Titre : Systèmes d'exploitation des ordinateurs
Auteur : CROCUS

Mots-clés : Systèmes d'exploitation (ordinateurs)
Description : XVII-[1]-364 p.; 24 cm
Adresse : Paris : Dunod, 1978
Cote de l'exemplaire : CNAM-BIB 8 Ca 2680

systèmes d'exploitation des ordinateurs

DUNOD
informatique

phase formation

- **Principes des ordinateurs,** par P. de Miribel
- **Fortran IV,** par M. Dreyfus
- **La pratique du fortran,** par M. Dreyfus et C. Gangloff
- **Cobol. Initiation et pratique,** par M. Barès et H. Ducasse
- **La construction de programmes structurés,** par J. Arsac
- **Let's talk D.P.; lexique d'informatique,** par J.P. Drieux et A. Jarlaud
- **Le langage de programmation PL/1,** par C. Berthet

phase spécialité

- **Bases de données : méthodes pratiques,** par D. Martin
- **Les fichiers,** par C. Jouffroy et C. Létang
- **Systèmes d'exploitation des ordinateurs,** par Crocus
- **Analyse fonctionnelle,** par H. Briand et C. Cochet

phase recherche

- **Programmation,** Actes du 2e colloque de l'Institut de Programmation, sous la direction de B. Robinet

et
l'Aide-mémoire Dunod informatique

systèmes d'exploitation des ordinateurs

CROCUS

Principes de conception

CROCUS
Nom collectif de : J. BELLINO, C. BÉTOURNÉ, J. BRIAT, B. CANET, E. CLEEMANN, J.-C. DERNIAME, J. FERRIÉ, C. KAISER, S. KRAKOWIAK, J. MOSSIÈRE, J.-P. VERJUS

DUNOD
phase spécialité | informatique

ISBN 2-04-001445-4

AVANT-PROPOS

L'idée de cet ouvrage est née lors d'une Ecole d'Eté d'Informatique organisée à Alès en 1971, sous le patronage de l'Association Française pour la Cybernétique Economique et Technique, pour permettre à des enseignants et à des chercheurs de réfléchir en commun à une présentation pédagogique des matières de l'informatique.

Un groupe de travail s'y est constitué pour rédiger des notes d'enseignement sur les systèmes d'exploitation. Un plan de cours détaillé publié aux Etats-Unis en juin 1971 par le « Cosine Committee of the Commission on Education of the National Academy of Engineering » sous le titre « An Undergraduate Course on Operating Systems Principles » a servi de document de départ et a permis de dépasser le stade des notes de cours. Divers membres du groupe ont pris pour base de leur enseignement la rédaction obtenue, ce qui a permis d'en mettre au point la présentation pédagogique. Enfin, cet ouvrage a constitué la matière des cours de trois Ecoles de Printemps sur les Systèmes d'Exploitation (Les Arcs, 1973 ; Auron, 1974 et 1975).

Ce livre est le résultat d'un travail fait en commun du début à la fin de la rédaction. C'est pourquoi un nom collectif, Crocus, a été choisi comme nom d'auteur par le groupe constitué de : J. Bellino (Centre Scientifique IBM de La Gaude), C. Bétourné (Université de Toulouse III (*)), J. Briat (Université de Grenoble I), B. Canet (Université de Rennes I), E. Cleemann (Université de Grenoble I), J. C. Derniame (Université de Nancy I), J. Ferrié (Université des Sciences et Techniques du Languedoc (*)), C. Kaiser (Conservatoire National des Arts et Métiers, Paris (*)), S. Krakowiak (Université de Grenoble I (*)), J. Mossière (Université de Grenoble I (*)) et J. P. Verjus (Université de Rennes I). Des observateurs ont participé aux travaux du groupe. Ce sont : G. Bazerque (Université de Toulouse I), J. C. Boussard (Université de Nice), C. Girault (Université de Paris VI) et C. Carrez (Université de Lille I). Nous tenons à remercier plus particulièrement ce dernier pour son rôle de contestataire permanent.

Nous exprimons notre reconnaissance à toutes les secrétaires, en particulier à Mmes G. Perez et M. Suard qui ont assuré une grande partie de la frappe des nombreuses versions intermédiaires du manuscrit, ainsi qu'à M. J. Riguet, qui s'est chargé de l'exécution de toutes les figures.

Nous remercions enfin les organismes d'appartenance des différents membres du groupe pour leur soutien matériel au cours de l'élaboration de cet ouvrage.

(*) Anciennement Iria.

AVANT-PROPOS DE LA SECONDE ÉDITION

L'évolution rapide des recherches et des réalisations dans le domaine des systèmes d'exploitation des ordinateurs a posé aux auteurs un délicat problème pour la préparation de cette seconde édition : dans quelle mesure fallait-il tenir compte des progrès réalisés depuis 1973, date d'achèvement du texte initial ?

Nous avons choisi de nous en tenir à une simple correction des erreurs matérielles relevées depuis la parution de la première édition. Une raison principale a motivé ce choix : l'évolution de l'enseignement de l'informatique a fait que la matière de cet ouvrage est maintenant intégrée, pour une large part, aux programmes de formation de base en informatique : maîtrise, Instituts de Programmation, Ecoles d'Ingénieurs. Cette formation met l'accent sur des principes généraux de conception dont la présentation nous paraît, pour l'essentiel, toujours d'actualité. L'abondance des travaux récents dans plusieurs domaines (protection, modèles, structuration des systèmes, ...) aurait pu justifier une refonte complète de certaines parties de l'ouvrage. Mais nous pensons que les concepts récemment introduits n'ont pas encore atteint une stabilité suffisante pour être intégrés à un enseignement de base, et que leur présentation relève encore, pour un temps, des enseignements spécialisés ou de la préparation à la recherche.

Depuis l'achèvement de la première édition, plusieurs ouvrages didactiques ont été publiés sur les systèmes d'exploitation. Nous avons inclus dans un complément bibliographique ceux qui nous paraissent, à des titres divers, les plus intéressants.

Nous insistons de nouveau sur l'importance que nous attachons, dans la présentation pédagogique de la matière de cet ouvrage, aux études de cas menées en parallèle : études de systèmes existants, projets de systèmes ou parties de systèmes. Quelques indications à ce sujet sont données dans le complément bibliographique.

Nous tenons enfin à remercier tous ceux qui nous ont aidés par leurs suggestions et critiques, et en particulier les enseignants, étudiants et stagiaires des divers cycles de formation où notre ouvrage a été utilisé.

Octobre 1976

PRÉAMBULE

L'informatique met en œuvre des ressources importantes et coûteuses, tant en ce qui concerne le matériel que les programmes. Un souci d'économie conduit à rendre ces ressources communes à un groupe de traitements. Cette fonction est remplie par un ensemble de programmes et de dispositifs câblés qui constituent le système d'exploitation (nous dirons plus simplement : le système). Ce système a pour charge de mettre à la disposition d'un groupe d'utilisateurs les ressources qu'il administre. Bien que les programmes inclus dans un système ne soient pas essentiellement différents des autres programmes, ils se particularisent par leur aspect dynamique et par la nécessité d'assurer l'indépendance mutuelle d'un ensemble d'utilisateurs. Le déroulement des programmes d'un système dépend de la nature de leurs données et de l'occurrence d'événements externes, ce qui rend le comportement d'un système difficilement prévisible et reproductible. Les problèmes qu'impliquent la mise en commun d'objets, leur partage, leur protection, la nécessité de les nommer, la synchronisation des actions qui peuvent être entreprises sur eux, prennent donc plus d'importance dans un système que dans d'autres programmes. Ces problèmes peuvent être abordés à propos d'un système particulier. De nombreux exemples de réalisations sont exposés dans la littérature technique, mais il est malaisé de dégager de ces descriptions, dont l'abord est souvent difficile, des principes généraux de conception. C'est à un tel effort de synthèse que nous avons tenté de contribuer.

Nous ne proposons pas dans cet ouvrage des règles de construction des systèmes d'exploitation, mais simplement des éléments pour leur conception. Plus exactement nous tentons de dégager, chaque fois que cela est possible, les principes qui s'appliquent à la conception des systèmes ou qui semblent devoir y contribuer dans les années à venir.

Ces principes intéressent les concepteurs de systèmes mais aussi ceux qui participent de près à leur évolution, qu'ils en assurent la maintenance ou l'exploitation. Ce travail leur est donc destiné, ainsi qu'aux étudiants et aux chercheurs spécialisés dans les problèmes liés aux systèmes d'exploitation ou, plus généralement, préoccupés par la conception ou l'utilisation de grands programmes. De façon plus précise, cet ouvrage s'adresse aux concepteurs, aux programmeurs de systèmes, aux étudiants en conception de systèmes ou en programmation avancée, ainsi qu'à tous les enseignants en informatique.

Nous supposons acquises la connaissance de l'anatomie d'un système simple ainsi que l'expérience de la programmation et de l'utilisation d'un système. Le lecteur devra avoir une idée claire du rôle d'un assembleur, d'un compilateur, d'un chargeur, d'un éditeur de liens, d'un interpréteur et d'un système de gestion de fichiers.

De même nous supposons connues les notions suivantes :

— l'organisation de l'unité de commande et de l'unité de traitement, les techniques d'adressage, la structure d'une instruction, le fonctionnement des mécanismes câblés d'exécution des instructions, les modes de fonctionnement (maître-esclave), les notions d'interruption et de déroutement ;

— l'organisation et les caractéristiques des principaux types de mémoire (circuits intégrés, tores, tambours, disques, bandes,...) ;

— le fonctionnement des différents types d'organes d'accès et leurs rapports avec l'unité de commande ;

— l'emploi des structures usuelles de données (tables, listes, piles, arborescences) et leur représentation en machine ;

— la structuration des programmes (récursivité, réentrance).

Il est également nécessaire de connaître un langage de programmation. Nous utilisons un langage inspiré d'ALGOL 60 pour la description des algorithmes, mais la connaissance d'un langage de niveau équivalent est suffisante pour leur compréhension.

La bibliographie qui figure à la fin de ce préambule recouvre à peu près les connaissances prérequises.

Cet ouvrage peut constituer un guide pour l'étude des principes de conception des systèmes d'exploitation des ordinateurs, mais il ne saurait être à lui seul suffisant. Il doit être complété par l'étude pragmatique d'un système réel. En particulier, il est recommandé aux enseignants d'illustrer les concepts par des exemples pris dans un ou plusieurs systèmes.

Des exercices repérés dans l'ordre de difficulté croissante par un nombre de 1 à 3 figurent à la fin de chaque chapitre. Leur but est de fournir l'occasion d'appliquer les connaissances acquises dans le cours et d'approfondir certains points non traités dans le corps de l'ouvrage. Pour la plupart des exercices, des schémas de solution sont regroupés *in fine*.

L'ouvrage contient enfin une bibliographie générale, avec regroupement des références par chapitre, et un index des termes le plus couramment utilisés.

BIBLIOGRAPHIE POUR LES CONNAISSANCES PRÉREQUISES

Arsac J., *Les systèmes de conduite des ordinateurs*, Dunod (2e édition, 1970).

Hopgood F. R. A., *Compiling techniques*, Macdonald Computer Monographs (1969).

Knuth D. E., *The art of computer programming*, **vol. 1** : Fundamental algorithms, Addison-Wesley (1968), en particulier 2.1 à 2.4.

Meinadier J. P., *Structure et fonctionnement des ordinateurs*, Larousse (1971).

Profit A., *Structure et technologie des ordinateurs*, Armand Colin (1970).

TABLE DES MATIÈRES

1

INTRODUCTION

La variété des formes externes que peuvent prendre les systèmes d'exploitation des ordinateurs et la diversité des fonctions qu'ils assurent rendent malaisée toute tentative de définition générale ou de classification rigoureuse. C'est pourquoi, après avoir donné une idée des principales fonctions d'un système et des aspects externes le plus souvent rencontrés, nous tenterons de dégager quelques caractéristiques communes qui guideront notre étude.

1.1 FONCTIONS ET ASPECTS EXTERNES DES SYSTÈMES

1.11 FONCTIONS D'UN SYSTÈME

Un système peut être examiné sous des points de vue très divers. On peut considérer qu'il remplit, vis-à-vis de ses utilisateurs, un certain nombre de fonctions dont la liste ci-dessous n'est pas exhaustive.

— Gestion et conservation de l'information : il s'agit d'offrir aux utilisateurs des moyens de créer, de retrouver et de détruire les objets sur lesquels ils veulent effectuer des opérations.

— Gestion de l'ensemble des ressources pour permettre l'exécution d'un programme : le système a pour rôle de créer un environnement nécessaire à l'exécution d'un travail.

— Gestion et partage de l'ensemble des ressources : le système est alors chargé de répartir ces ressources (matériels, informations et programmes) entre les usagers. Pour cela, il doit réaliser un ordonnancement des travaux.

— Extension de la machine câblée : le système a ici pour rôle de masquer certaines limitations ou imperfections du matériel, ou de simuler une machine

différente de la machine réelle. L'utilisateur a alors à sa disposition une « machine virtuelle » munie d'un « langage étendu », c'est-à-dire d'un mode d'expression mieux adapté que les seules instructions câblées. A l'aide de ce langage, il peut commander l'exécution de ses programmes et en effectuer la mise au point. Font aussi partie du langage étendu les commandes de l'opérateur et les directives utilisées pour la « génération » du système.

1.12 ASPECTS EXTERNES DES SYSTÈMES

Les systèmes se présentent sous un grand nombre d'aspects externes ; cette diversité reflète la variété des tâches à remplir et des caractéristiques des matériels utilisés. En dehors d'une classification historique [Rosin, 69], on peut classer les systèmes suivant la fonction principale qu'ils remplissent. On pourrait ainsi distinguer :

a) les systèmes orientés vers la commande de processus industriels (exemples : système de conduite d'un haut fourneau, système de guidage d'une fusée, central téléphonique),

b) les systèmes orientés vers la conservation et la gestion de grandes quantités d'information (exemples : systèmes de documentation automatique, système de gestion de comptes bancaires, systèmes de réservation de places),

c) les systèmes destinés à la création et à l'exploitation de programmes. Ces derniers systèmes peuvent eux-mêmes être classés suivant le degré d'interaction possible d'un utilisateur avec ses programmes (systèmes de traitement par trains ou systèmes conversationnels), suivant le mode d'entrée des programmes (local ou à distance, par fournées ou continu), suivant le mode de partage des ressources (mono- ou multiprogrammation), suivant les possibilités du langage étendu (langage unique ou langages multiples). Ces systèmes peuvent présenter à un degré variable certains aspects du type *a*) ou *b*) : ainsi, les contraintes de temps sont importantes pour un système comportant des usagers interactifs.

1.2 CARACTÉRISTIQUES COMMUNES

Les divers systèmes énumérés précédemment posent à leur concepteur les mêmes types de problèmes, bien qu'ils diffèrent par leurs objectifs, par leurs contraintes et par leur aspect externe. L'analyse de leur structure et de leur fonctionnement permet en effet de dégager un certain nombre de caractéristiques communes :

— gestion et partage d'un ensemble de ressources,
— désignation des objets et accès à l'information,
— coopération entre processus parallèles,
— protection des informations et fiabilité des programmes.

Dans le corps de l'ouvrage, nous tentons pour chacun de ces aspects de dégager les concepts utiles à la résolution des problèmes rencontrés. Lorsque l'état des connaissances le permet, nous proposons une approche aussi générale que possible, illustrée par des exemples pris dans des systèmes existants ; dans le cas contraire, nous suivons une démarche plus pragmatique en partant de réalisations particulières.

Les exemples sont, en général, empruntés à des systèmes importants fonctionnant sur des matériels moyens ou gros. Toutefois, les notions présentées s'appliquent également aux petits systèmes.

1.21 PARTAGE DES RESSOURCES PHYSIQUES

Des raisons économiques amènent les utilisateurs d'ordinateurs à mettre en commun leur matériel, ce qui pose le problème de la planification de son utilisation et de son partage. Une manière immédiate de résoudre ce problème consiste à admettre un seul programme à la fois en mémoire ; ce programme dispose donc pendant son déroulement de toutes les ressources de l'installation laissées libres par le système. L'utilisation de l'ordinateur peut alors être entièrement planifiée par le service d'exploitation : il suffit de réserver un temps suffisant à chacun des programmes et de prévoir l'ordre dans lequel ils seront exécutés.

Des considérations d'efficacité conduisent à adopter des méthodes de partage plus complexes.

Exemple. Soit un système de traitement par trains en monoprogrammation avec gestion simultanée des entrées-sorties, organisé comme suit : un seul programme d'utilisateur est présent à la fois en mémoire ; les entrées-sorties (y compris l'entrée des programmes eux-mêmes) ont lieu depuis (ou vers) une zone sur disque réservée à cet effet et sont effectuées par des programmes appelés « symbionts ».

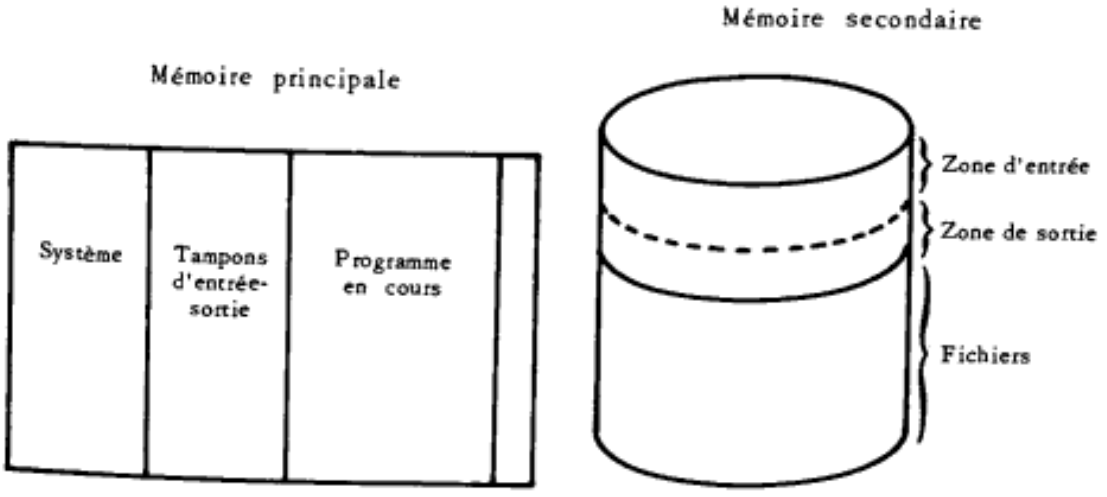


Figure 1. Exemple de partage de la mémoire.

Le disque est divisé en deux régions : l'une est réservée aux entrées-sorties, l'autre à la conservation des informations des utilisateurs (fichiers). De même, la partie de mémoire principale non occupée par le système est divisée en une zone de tampons d'entrée-sortie et une zone réservée au programme en cours d'exécution.

Une telle organisation, qui suppose l'existence d'une unité d'échange pouvant fonctionner en parallèle avec l'unité centrale, permet de réduire le temps global de traitement d'un ensemble de programmes.

Le système gère plusieurs files d'attente :

— la file des travaux en attente d'exécution (ces travaux sont rangés dans la zone « entrée » sur disque,

— les files des informations à sortir (il y a une file par type de périphérique).

Cet exemple simple permet de mettre en évidence des notions communes à de nombreux systèmes :

— la demande simultanée d'une même ressource par plusieurs utilisateurs conduit à un partage qui peut être séquentiel (exemple : l'unité centrale) ou simultané (exemple : le disque, la mémoire principale). Dans le cas de la mémoire, le partage peut être statique (les limites des différentes zones sont fixées une fois pour toutes) ou dynamique (les limites peuvent varier),

— à un instant donné, l'ensemble des demandes relatives à une ressource peut excéder la quantité disponible de cette ressource, et cette situation provoque l'attente des demandeurs.

D'une façon plus générale, il existe dans un système un ensemble de ressources utilisables et un ensemble de travaux (ou charge) dont le traitement entraîne des demandes de ces ressources. Le système est chargé de l'attribution des ressources suivant les objectifs qui ont été fixés à sa conception et qui peuvent consister à :

— mieux utiliser le matériel ou certaines parties du matériel,

— mieux satisfaire les utilisateurs, ce qui peut s'exprimer sous diverses formes (réduire le temps de réponse, respecter les échéances...).

Ces objectifs peuvent être contradictoires. La définition du système implique un certain nombre de choix de conception, qui influent sur les performances finales. Ainsi, dans l'exemple ci-dessus, les choix importants concernent :

— le mode de partage de la mémoire principale et de la mémoire secondaire (statique ou dynamique ? si dynamique, selon quel critère ?),

— le mode de gestion des différentes files d'attente (avec ou sans priorité ?).

Il est commode, pour chacune des ressources importantes d'un système (processeurs, mémoire centrale, mémoire secondaire), d'étudier séparément les stratégies individuelles permettant de la gérer. Les résultats d'une telle étude sont applicables dans les cas où les problèmes d'allocation des divers types de ressources peuvent être découplés ; mais le plus souvent ces problèmes interfèrent.

La mise en œuvre de plusieurs stratégies particulières indépendantes dans un système comportant plusieurs types de ressources peut conduire à des

conflits si elle est faite sans précautions. En particulier peuvent apparaître deux types de phénomènes :

— l'écroulement, ou la dégradation des performances, dû à une mauvaise gestion de l'ensemble mémoire-processeur dans un système multiprogrammé,

— l'interblocage d'un groupe de processus, situation dans laquelle chaque processus est en attente d'une ressource possédée par un autre processus du groupe.

Toute politique d'allocation de ressources doit tenir compte de ces dangers ; elle doit donc être conçue de manière globale.

1.22 GESTION DE L'INFORMATION

L'utilisateur d'un système informatique désire effectuer des traitements sur des objets ; ces objets peuvent être, entre autres, les fichiers contenant des programmes ou des données, les segments dans les systèmes à mémoire segmentée, les variables, tableaux et structures définis dans divers langages de programmation.

L'utilisateur désigne les objets qu'il veut employer grâce à des identificateurs. Pour qu'un objet puisse être traité dans un ordinateur, il faut lui associer des informations le désignant sans ambiguïté ; de toute manière, il faut qu'au moment du traitement effectif, l'objet puisse être localisé par le processeur chargé de ce traitement. Ces différentes informations de localisation ou de désignation constituent les noms de l'objet. Au cours de son existence, l'objet peut être désigné par des noms différents.

L'exemple qui suit, emprunté au langage de commande du système SIRIS 7 sur CII 10070, permet de préciser l'établissement de la correspondance entre identificateurs, noms et objets.

Exemple.

! Fortran si,go ⋮ $x = sin\ (y + z)$ *Write (273) x* ⋮	Compiler le programme qui suit et ranger le résultat de la compilation dans le fichier appelé *go*. Ecrire la valeur de x sur le support associé au descripteur *273*.
①- - - - - - - - - - - - -	
! Assign bib, fil, (nam,f4lib), (sts,old)	Assignation au descripteur *bib* du fichier existant (bibliothèque *Fortran*).
! Link *: Option (unsat, bib)*	Editer le programme contenu dans le fichier *go*, en allant chercher les références externes dans le fichier associé au descripteur *bib* ; placer le résultat (module de chargement) dans le fichier *go* (option par défaut).
②- - - - - - - - - - - - -	

```
! Assign 273,fil, (nam,resul), (unt,mt,
  (vol, 450))
```

Assignation au descripteur d'identificateur *273* d'un nouveau fichier *resul*, à créer sur la bande magnétique n° *450*.

③- - - - - - - - - - - - - -

```
! Run
```

Charger le programme édité contenu dans le fichier *go* et l'exécuter.

④- - - - - - - - - - - - - -

En 1, le compilateur a produit un programme translatable rangé dans le fichier *go*. Dans ce programme :

— le nom de *x* est une adresse relative (déplacement) par rapport à l'origine du programme,

— le nom *sin* n'est pas défini : l'identificateur *sin* figure dans une liste de références non satisfaites attachée au programme,

— le descripteur *273* n'a pas de valeur.

En 2, l'éditeur de liens a produit, dans le fichier *go*, un module de chargement (translatable) constitué en réunissant le texte initialement contenu dans le fichier *go* à ceux du fichier *f4lib* dont les identificateurs figuraient parmi les références non satisfaites de *go* : par exemple, *sin* est désigné maintenant par un déplacement relatif à l'origine du module de chargement et toutes les références à *sin* se font par ce nom.

En 3, le descripteur *273* a pour valeur la chaîne de caractères *'resul'* ; la correspondance entre l'identificateur *273* et le fichier appelé *resul* pourra donc être complètement établie à l'exécution.

En 4, les adresses en mémoire ont été fixées pour le module de chargement contenu dans *go* : maintenant *x* et *sin* désignent effectivement des objets.

Sur cet exemple, on peut constater que la correspondance entre identificateur et objet est établie en plusieurs étapes : on dit que l'identificateur est progressivement lié à l'objet qu'il désigne.

Le but de cette opération de liaison (« binding ») est essentiellement d'associer, de façon plus ou moins durable, l'objet à des emplacements adressables par un processeur, ce qui est la seule façon de le consulter ou de le modifier. La liaison est établie à l'aide d'une chaîne de noms partant de l'identificateur et aboutissant à l'objet désigné. Cette chaîne peut être construite en respectant le sens de l'identificateur vers l'objet : c'est le cas, dans l'exemple ci-dessus de la variable *x*. Elle peut aussi être construite dans un ordre différent : c'est le cas, dans notre exemple, de la liaison de l'identificateur *sin* lors de l'édition de liens, où sont reliées deux parties de la chaîne constituées à l'avance.

L'opération d'« assignation » des descripteurs de fichiers fournit le moyen de retarder jusqu'au stade de l'exécution le choix du fichier utilisé : un même programme peut être exécuté plusieurs fois avec des fichiers différents sans avoir à modifier son texte. De façon plus générale, le principe consistant à retarder les liaisons (« delay binding time ») permet une plus grande souplesse d'utilisation, qui se paye par une plus grande complexité et, parfois, une perte d'efficacité.

Dans de nombreux systèmes, d'autres opérations de liaison peuvent encore intervenir pendant l'exécution du programme : les adresses obtenues à l'édition de liens sont des adresses « virtuelles » et le mécanisme de transformation de ces adresses virtuelles en adresses réelles peut être plus complexe qu'une simple translation statique : production dynamique d'adresses par des mécanismes de segmentation par exemple.

On demande souvent à un système de permettre à plusieurs utilisateurs d'accéder à des informations communes. Ce partage pose des problèmes supplémentaires puisque l'indépendance des utilisateurs doit toujours être assurée. On peut concevoir deux façons de partager un objet : constituer de cet objet autant d'exemplaires distincts que nécessaire, ou bien permettre à chaque utilisateur d'accéder à l'exemplaire unique de l'objet. Dans le premier cas, il faut assurer la cohérence des différents exemplaires ; dans le second cas, on a encore le choix entre l'affectation à l'objet d'un nom différent pour chaque utilisateur, ou l'utilisation d'un nom commun.

Exemple. Une procédure partagée peut être recopiée en autant d'exemplaires qu'il y a de programmes qui l'utilisent. Une autre solution consiste à n'avoir qu'un exemplaire réentrant. Dans ce dernier cas, les différents noms qu'elle possède pour les différents programmes qui l'utilisent doivent en dernier ressort désigner le même objet.

1.23 COOPÉRATION DES PROCESSUS

Dans un système, plusieurs activités peuvent se dérouler simultanément. Ces activités résultent de l'exécution de programmes. Nous utiliserons pour les désigner le terme de processus.

Reprenons l'exemple, introduit en 1.21, d'un système de monoprogrammation avec « symbiont ». A un instant donné, on peut observer l'exécution d'un programme par l'unité centrale et d'une entrée-sortie par l'unité d'échange. Chacune de ces activités fait partie d'un processus. Le déroulement de chaque processus est déterminé par la suite d'instructions exécutée par l'organe actif correspondant, ou processeur (unité centrale ou unité d'échange).

Il est commode d'introduire un processus distinct pour une activité que l'on veut considérer comme indépendante. Une telle décomposition ne tient pas compte du fait que ces processus peuvent ou non se dérouler simultanément ; en particulier, elle ne tient pas compte du nombre de processeurs. On dit alors que les processus ainsi définis sont logiquement parallèles.

Notons que les notions de programme et de processus sont distinctes : chaque exécution d'un même programme correspond à un processus distinct ; si de plus ce programme est réentrant, ces exécutions peuvent être simultanées.

Dans l'exemple du 1.21, l'exécution du train de programmes d'utilisateurs et l'exécution du « symbiont » peuvent être considérés comme des processus logiquement parallèles : le processus *travail* et le processus *symbiont*. Toutefois, ils ne se déroulent pas toujours simultanément ; lorsque *symbiont* utilise l'unité

centrale, l'exécution de *travail* est suspendue. En dehors de ce conflit dû à une insuffisance de ressources, les deux processus ont d'autres interactions :

— le processus *symbiont* ne doit pas pouvoir accéder à un tampon que le processus *travail* est en train de remplir,

— lorsqu'un tampon de sortie est plein, *travail* doit réveiller *symbiont* si ce dernier est inactif,

— lorsqu'un nouveau programme est introduit dans la file d'entrée, *symbiont* doit réveiller *travail* si ce dernier est inactif.

Cet exemple met en évidence l'existence de différents types d'interaction entre processus :

— conflit pour l'accès à une ressource (unité centrale ou tampon d'entrée-sortie) qui ne peut être utilisée que par un seul processus à la fois (exclusion mutuelle),

— action directe (synchronisation) d'un processus sur un autre : mise en attente ou réveil.

On rencontre dans un système bien d'autres formes de parallélisme : par exemple les demandes de service faites par des utilisateurs depuis des consoles d'accès direct correspondent à des processus logiquement parallèles dont le déroulement peut être assuré par un système de multiprogrammation à un ou plusieurs processeurs.

On peut considérer un système d'exploitation comme un ensemble de processus parallèles pouvant interagir. Pour mettre en œuvre ces processus, on a deux problèmes à résoudre :

— écrire les programmes décrivant chaque activité individuelle,

— concevoir des mécanismes d'interactions permettant les différents types de coopération : exclusion mutuelle, synchronisation, communication d'information.

1.24 PROTECTION

La coexistence, à l'intérieur d'un système, d'informations appartenant à différents utilisateurs impose la protection de ces informations contre les erreurs de programmation ou contre les malveillances. Par exemple, les informations utilisées pour la gestion du système lui-même doivent être inaccessibles aux programmes des utilisateurs ; un utilisateur peut souhaiter n'autoriser la consultation ou la modification de ses informations privées qu'à certains utilisateurs explicitement spécifiés ; plusieurs utilisateurs peuvent ainsi avoir des droits différents sur une même ressource.

Plus généralement, le rôle d'un système de protection est de garantir, dans tous les cas, l'intégrité de certaines ressources protégées. Cette protection peut être mise en œuvre par différents mécanismes câblés ou programmés. Par exemple, de nombreux ordinateurs comportent deux modes d'exécution : maître et esclave, et certaines instructions (entrée-sortie, ...) ne peuvent être exécutées qu'en mode maître.

1.3 PROBLÈMES DE CONCEPTION ET D'ÉVALUATION

La conception des systèmes se trouve actuellement à une étape intermédiaire entre un stade empirique où elle est basée sur le savoir-faire et l'intuition, et un stade scientifique où elle pourrait s'appuyer sur des études théoriques conduisant à des méthodes de construction des systèmes. Deux approches, entre autres, sont envisageables :

1) faciliter la conception par une meilleure connaissance du comportement des systèmes existants, soit à l'aide de mesures, soit par la construction de modèles de comportement,

2) faciliter la réalisation proprement dite d'un système en définissant des techniques d'écriture pour les gros programmes dont la réalisation pose d'importants problèmes de documentation et de communication entre les participants.

1.31 MESURES ET MODÈLES DE SYSTÈMES

La conception et la mise au point d'un système sont facilitées par la connaissance d'informations quantitatives sur le système lui-même ou sur des systèmes existants analogues.

Différentes techniques de mesures, câblées ou programmées, permettent d'obtenir des informations :

— sur le comportement d'un système (temps de réponse, débit des travaux, utilisation des ressources, fréquence de certains événements),

— sur le comportement d'un utilisateur en mode interactif ou sur le comportement dynamique d'un programme.

Ces informations peuvent également être importantes pour choisir un matériel et un système, pour modifier une configuration, pour assurer la comptabilité de l'utilisation des ressources et pour optimiser les programmes.

Des renseignements sur le comportement d'un système peuvent également être obtenus par l'utilisation de modèles qui en fournissent une image approchée. Ces modèles peuvent être traités par le calcul, si leur complexité le permet, et fournir ainsi des formules directement utilisables ; si leur complexité est trop grande, ils peuvent être traités par simulation.

1.32 MÉTHODOLOGIE DE CONCEPTION

La réalisation d'un système nécessite l'intervention de nombreuses personnes et peut durer longtemps. Etant donné l'absence de techniques automatiques de construction, la fiabilité d'un système dépend beaucoup de la méthode suivie pour sa réalisation. Ainsi, une mauvaise documentation des programmes ou l'absence de conventions précises de liaison entre les constituants du système sont des sources importantes d'erreur et diminuent donc considérablement la fiabilité du produit.

La construction d'un programme peut être simplifiée si ses constituants sont décrits sous la forme de modules pouvant être combinés sans avoir à connaître les détails de leur réalisation interne.

Chaque module ne doit communiquer avec les autres qu'en suivant des règles bien définies : les spécifications d'interface. Ces règles doivent en particulier imposer une représentation cohérente des informations communes.

Il est souhaitable de disposer de méthodes d'analyse permettant la décomposition d'un système en modules. Deux méthodes peuvent être employées :

— une méthode de conception descendante consistant à définir l'implantation de la solution par étapes ; au cours de chacune d'elles, on complète les définitions de fonctions ou d'informations utilisées aux étapes précédentes,

— une méthode de conception ascendante qui utilise des fonctions ou des informations déjà décrites pour la réalisation de nouvelles fonctions.

Dans la pratique, on utilise alternativement l'une et l'autre méthode.

1.4 ORGANISATION DE L'OUVRAGE

Les divers aspects des systèmes qui viennent d'être considérés sont traités dans l'ordre suivant :

Chapitre 2 : Processus.
Chapitre 3 : Gestion de l'information.
Chapitre 4 : Gestion des ressources physiques.
Chapitre 5 : Protection.
Chapitre 6 : Mesures et modèles de systèmes.
Chapitre 7 : Méthodologie de conception et de réalisation.

2

LES PROCESSUS

2.1 INTRODUCTION

Lorsqu'on essaie d'analyser le fonctionnement d'un système d'exploitation, on se trouve en présence d'un ensemble d'activités multiples, simultanées ou non, et présentant de nombreuses interactions mutuelles. Ainsi, dans un système comprenant deux unités centrales, il est possible d'exécuter simultanément deux programmes ; une unité d'échange transfère de l'information indépendamment des unités centrales.

Pour décrire le fonctionnement d'un système, il est commode d'introduire la notion de processus, représentant une activité que l'on veut considérer comme élémentaire. Un processus représente l'exécution d'un programme comportant des instructions et des données : c'est une entité dynamique, créée à un instant donné, qui disparaît en général au bout d'un temps fini.

Un transfert d'information par une unité d'échange peut être considéré comme un processus ; ce transfert correspond à l'exécution d'un certain nombre de commandes envoyées à une unité de liaison.

L'objet de ce chapitre est de préciser la notion de processus, de montrer comment elle est mise en œuvre et comment est programmée la coordination entre processus. Aucun processus en effet n'est totalement isolé des autres ; à certains moments de son existence il communique avec d'autres processus, c'est-à-dire qu'il échange avec eux des signaux ou des informations ; parfois il les détruit, les arrête provisoirement, les fait repartir ; en outre, les organes de la machine, comme l'unité centrale ou la mémoire principale, doivent être partagés entre les processus, qui ne peuvent les monopoliser en général pendant toute leur existence.

Le problème général de la décomposition d'un système en processus ne nous intéresse pas ici (voir toutefois 2.24 à titre d'exemple). Nous supposons donné un ensemble de processus, sans nous soucier de leur nature ni de ce qu'ils font : seuls comptent les problèmes généraux de la création, de l'activation et de la coordination des processus.

Les notions présentées ci-après sont bien connues, au moins de nom. Le lecteur voudra bien admettre qu'il n'est pas possible de donner, dans l'état actuels des connaissances, une définition formelle des concepts introduits ; il devra donc faire appel à son expérience. Nous espérons que les notions s'éclairciront à mesure qu'il avancera dans le chapitre.

2.2 DÉFINITIONS

2.21 INSTRUCTIONS. PROCESSEUR. PROCESSUS

Comme tout système général, un système informatique peut être observé (ou décrit) à différents niveaux. Nous donnerons plus loin quelques exemples de niveaux d'observation usuels.

A un niveau donné s'exécutent des **programmes**, ensembles ordonnés d'**instructions**. La nature de l'instruction est fonction du langage considéré, et son exécution peut être complexe. L'instruction est considérée comme indécomposable (indivisible), c'est-à-dire qu'on s'interdit d'observer le système pendant l'exécution d'une instruction.

L'entité, câblée ou non, capable d'exécuter une instruction, est appelée **processeur.**

Enfin, un **processus séquentiel** (ou plus simplement **processus**), qui correspond à l'exécution d'un programme séquentiel, est une suite temporelle d'exécutions d'instructions.

Exemple 1. Le niveau d'observation le plus courant est celui où l'instruction est l'instruction (au sens usuel) de la machine, le processeur l'unité centrale ou un canal ; un processus représente alors l'exécution d'un programme écrit en langage de la machine.

Exemple 2. Dans une machine microprogrammée, on peut observer le système au niveau de la micro-instruction ; le processeur est alors l'organe chargé d'exécuter les micro-instructions et le processus est l'exécution d'une suite de micro-instructions. S'il y a parallélisme au niveau des micro-instructions, l'exécution d'une instruction (au sens de l'exemple 1) met en jeu une famille de processus.

Exemple 3. Dans un système permettant d'interpréter le langage APL, l'interpréteur APL est le processeur, l'instruction est l'instruction du langage APL et un processus est l'exécution d'un programme écrit en APL.

L'ensemble des variables et des procédures utilisables par un processus est le **vecteur d'état** de ce processus. Rappelons qu'on s'interdit d'observer le vecteur d'état d'un processus pendant l'exécution d'une instruction (qui prend toujours un temps fini). Par contre, entre deux instructions, il est possible d'accéder aux

données qui ont alors une valeur bien définie. Nous dirons que le processus se trouve alors en un **point observable** ; deux points observables consécutifs délimitent une instruction.

Certains éléments du vecteur d'état d'un processus ne sont accessibles que par un processus ; certains autres sont également accessibles par d'autres processus : on pourra distinguer des variables **locales** et des variables **globales.**

Exemple. Dans la plupart des systèmes, il existe une variable globale indiquant la date courante, accessible à tous les processus ; de même, les procédures de gestion de fichiers sont communes à tous les processus. En revanche, les variables et les procédures déclarées à l'intérieur du programme d'un processus sont locales à ce processus.

2.22 NOTION DE RESSOURCE. ÉTATS DES PROCESSUS

2.221 Ressources et états des processus

Pour qu'un processus puisse évoluer, il a besoin de procédures et de données, de mémoire destinée à les contenir, de l'unité centrale, éventuellement de fichiers et de périphériques. Nous appelons toutes ces entités des **ressources.** Comme les ressources du système sont en nombre limité, il n'est pas possible d'attribuer à chaque processus, dès sa création, toutes les ressources dont il aura besoin. On peut alors arriver à la situation où, parvenu en un point observable, un processus n'est pas en possession des ressources indispensables à l'exécution de l'instruction suivante. On dit que le processus est dans l'état **bloqué**. Par opposition, un processus qui dispose de toutes les ressources dont il a besoin pour exécuter l'instruction suivante est dit dans l'état **actif.**

Exemple 1. Un processus est bloqué :

— s'il ne dispose pas du processeur,

— si la prochaine instruction à exécuter se trouve dans une page non chargée en mémoire centrale.

Notons que le problème est généralement plus complexe : quand un processus est bloqué, le moniteur peut décider de lui retirer des ressources supplémentaires pour permettre à d'autres processus de progresser. Ainsi, quand un processus est bloqué en attente du chargement d'une page, le moniteur lui retire l'unité centrale au profit d'un autre processus.

Remarque. Dans ce qui précède, nous avons supposé que la détection des ressources manquantes avait lieu en un point observable. Dans la pratique, il n'en est pas tout à fait ainsi : on lance l'exécution de l'instruction, qui se termine anormalement du fait du manque d'une ressource, et on revient automatiquement au point observable précédent (cf. déroutement du CII 10070).

Dans un système dans lequel plusieurs processus coopèrent à la réalisation d'un même travail, un processus peut se trouver dans l'impossibilité de progresser pour une raison logique : l'attente d'un signal d'un autre processus.

Exemple 2. Le processus p fait un calcul et range le résultat dans un tampon, le processus q est chargé d'imprimer le contenu du tampon : le processus q ne peut s'exécuter que lorsque p a rempli le tampon.

Dans le premier exemple, le programmeur n'est pas conscient du blocage de son processus ; dans le second, au contraire, c'est lui qui programme explicitement l'attente. Quand on désire distinguer les deux causes de blocage, on les désigne respectivement sous les noms de blocage **technologique** et de blocage **intrinsèque** [Saltzer, 66].

Si l'on se place du point de vue du système, il est commode de considérer comme des ressources les signaux de synchronisation échangés par les processus : la notion de blocage se confond alors avec l'absence d'au moins une ressource nécessaire à l'exécution de l'instruction suivante.

Au contraire, du point de vue du programmeur qui ne se préoccupe que du blocage intrinsèque, il est commode de considérer que chaque processus s'exécute sur une **machine virtuelle** qui comprend virtuellement toutes les ressources nécessaires à l'exécution du processus. La correspondance dynamique entre les ressources de chaque machine virtuelle et les ressources physiques du système est laissée à la responsabilité du système.

Examinons maintenant du point de vue du système, les transitions entre les états actif et bloqué.

Un processus actif passe dans l'état bloqué dès qu'il lui manque une ressource nécessaire à l'exécution de l'instruction suivante. Un processus bloqué devient actif dès que toutes les ressources nécessaires sont rassemblées. Pratiquement, des informations sur les ressources allouées à un processus p font partie du vecteur d'état de ce processus ; les transitions entre états correspondent donc à la modification du vecteur d'état par d'autres processus (ou par le processus p lui-même, dans le cas de la transition actif → bloqué).

2.222 Accès aux ressources

Une ressource est dite **locale** à un processus si elle ne peut être utilisée que par ce processus ; elle doit obligatoirement disparaître à la destruction de ce processus puisqu'elle n'est plus utilisable. Une ressource qui n'est locale à aucun processus est dite **commune.**

Une ressource commune est dite **partageable avec n points d'accès** $(n \geqslant 1)$ si cette ressource peut être attribuée, au même instant, à n processus au plus. « Au même instant » signifie que si un observateur interrompait tous les processus et observait leurs vecteurs d'état, il constaterait que la ressource est utilisée par n d'entre eux au plus. Une ressource partageable à un point d'accès est dite **critique.**

Des processus sont dits **indépendants** s'ils n'ont que des ressources locales. Ils sont dits **parallèles** pour une ressource s'ils peuvent l'utiliser simultanément et en **exclusion mutuelle** s'il s'agit d'une ressource critique.

Exemple 1. Une unité centrale est une ressource à un seul point d'accès : tous les processus sont en exclusion mutuelle pour cette ressource.

Exemple 2. Si l'on ne considère que des processeurs virtuels, ressources locales à chaque processus, les processus sont indépendants. On peut dire par abus de langage qu'ils sont parallèles pour la ressource processeur physique.

Exemple 3. Un programme réentrant est une ressource à un nombre illimité de points d'accès.

Le mode d'accès à une ressource peut évoluer dynamiquement : un fichier est une ressource à une infinité de points d'accès quand il est ouvert en lecture, critique quand il est ouvert en écriture.

2.223 Pouvoir d'un processus

Nous appelons **pouvoir** d'un processus un ensemble d'informations définissant les ressources accessibles à ce processus, ainsi que leur mode d'accès. Le pouvoir permet de contrôler l'utilisation des ressources en fonction de l'identité du processus.

Exemple. Un processus en mode maître peut accéder aux ressources que sont les instructions privilégiées ; une clé d'écriture permet au processus d'écrire dans les pages ayant le verrou correspondant.

Le pouvoir d'un processus peut évoluer dynamiquement. Un problème de protection se pose quand le processus a besoin d'étendre son pouvoir, pour l'exécution d'une entrée-sortie par exemple (voir Chap. 5).

2.224 Contenu du vecteur d'état

Le vecteur d'état d'un processus contient grosso modo des informations de deux ordres :

— des informations utilisées explicitement par le processus (variables, procédures),
— des informations utilisées par le système pour gérer l'attribution des ressources ; il s'agit de la description des ressources attribuées ou demandées.

Exemple. Dans le système ESOPE sur CII 10070, le vecteur d'état d'un processus est défini par :
— le double-mot d'état de programme ou *PSD* (compteur ordinal, adresse virtuelle ou réelle,...),
— le contenu des 16 registres généraux,
— le contenu de la mémoire virtuelle.

Le pouvoir du processus est représenté par une partie du *PSD* (bit indiquant le mode-maître ou esclave, clé d'écriture) et un octet contenant l'autorisation d'emploi de certaines primitives du système.

2.23 RELATIONS ENTRE PROCESSUS

Considérons maintenant une famille de processus et leurs vecteurs d'état, à un niveau d'observation donné.

Deux processus sont en relation (ne sont pas indépendants) si leurs vecteurs d'état ont une intersection non vide : l'un des processus peut rendre une ressource accessible à l'autre, ou le priver de cette ressource, c'est-à-dire finalement que l'un des processus peut faire changer l'autre d'état.

2.231 **Création et destruction**

Un processus est une entité dynamique qui naît (lors du lancement de l'exécution d'un programme) et meurt (à la fin de cette exécution). Avant d'examiner les relations proprement dites entre processus, considérons tout d'abord les opérations de création et de destruction.

Créer un processus, c'est lui donner un nom et définir son vecteur d'état initial. Le nom permet au système et aux autres processus de désigner sans ambiguïté le nouveau processus. Dans le vecteur d'état initial il faut spécifier en particulier le programme, les données d'entrée et le pouvoir du processus. Si, comme c'est généralement le cas, le processus créé doit accomplir une certaine tâche au profit du processus créateur, les vecteurs d'état des deux processus doivent avoir certaines variables communes (données initiales, résultats). Plus délicate est l'attribution d'un pouvoir initial au processus créé : un problème de protection se pose quand le processus créé a un pouvoir supérieur à celui de son créateur (voir Chap. 5).

On définit récursivement la **descendance** d'un processus p de la façon suivante :

— un processus q créé par p appartient à la descendance de p,

— si un processus q appartient à la descendance de p, tout processus créé par q appartient à la descendance de p.

— il n'y a aucune autre manière de créer un processus appartenant à la descendance d'un processus p.

La destruction d'un processus peut intervenir de deux façons :

— destruction à sa propre initiative lorsqu'il parvient à la fin de son exécution,

— destruction à l'initiative du système (ou d'un autre processus) d'un processus dont on a détecté un mauvais fonctionnement. Dans ce dernier cas, on doit signaler au processus créateur du processus détruit p qu'il s'agit d'une terminaison anormale et détruire toute la descendance de p (ces processus pouvant utiliser des données de p).

A la destruction d'un processus, son vecteur d'état disparaît ; les ressources communes qu'il utilisait sont rendues disponibles pour d'autres processus, ses ressources locales sont détruites.

Les processus sont créés ou détruits soit à l'initiative du système, soit à celle d'un processus quelconque.

2.232 **Synchronisation et communication**

Deux aspects fondamentaux sont à considérer : la synchronisation proprement dite entre processus (c'est-à-dire le fait de permettre à un processus actif de changer d'état ou de faire changer d'état un autre processus) et la communication de données d'un processus à un autre.

Nous avons signalé en 2.222 l'existence de ressources critiques. Le premier problème de synchronisation que nous traiterons est celui de l'exclusion mutuelle à ces ressources critiques. Il s'agit bien sûr d'un cas particulier, mais que l'on rencontre assez souvent dans la pratique pour qu'il soit utile de l'examiner en détail. Nous présenterons ensuite quelques outils plus généraux de synchronisation.

En ce qui concerne la communication de données d'un processus à un autre, nous montrerons sur des exemples que la programmation devient très vite complexe si on se limite à l'emploi de variables communes et des mécanismes généraux de synchronisation. Des primitives plus riches seront alors décrites.

2.24 EXEMPLE DE DÉCOMPOSITION EN PROCESSUS

Ce paragraphe présente un exemple d'utilisation des processus. Nous avons retenu un sous-ensemble de système de gestion des entrées-sorties du système ESOPE [Baudet, 72] dans lequel nous avons supprimé des détails de programmation sans modifier le découpage en processus et la synchronisation entre ces processus.

Le système auquel on veut ajouter un système d'entrées-sorties est à accès multiple : à un instant donné, des processus de différents usagers coexistent. Les usagers peuvent conserver des informations (programmes ou données) dans des fichiers sur disques (un fichier est un ensemble d'articles, l'article est l'unité logique d'accès aux informations). On veut construire un dispositif permettant à un usager d'imprimer le contenu d'un fichier ; après la demande, l'impression a lieu à un instant dépendant uniquement des demandes en attente et l'usager ne reçoit aucun message en fin d'impression. Nous supposons en outre que le système comporte une seule imprimante et que les erreurs de fonctionnement (fin de papier, erreur de transmission) sont gérées par un opérateur qui peut en outre mettre en service ou hors service l'imprimante.

Pour exploiter au mieux le parallélisme entre les différentes unités du système, on introduit un processus pour chaque unité fonctionnant de manière autonome, c'est-à-dire :

— un processus attaché au disque, ou facteur, chargé de la lecture des articles de fichier,

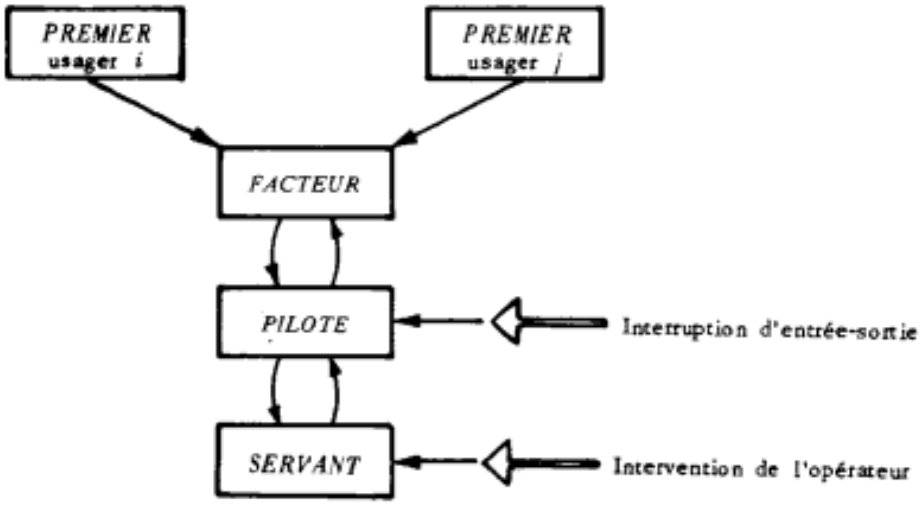
— un processus attaché à l'imprimante, ou pilote, chargé de l'impression des articles de fichier,

— un processus servant, associé à la console de l'opérateur.

Nous admettons enfin qu'à chaque usager du système correspond un processus, que nous appelons le processus premier de l'usager, chargé d'interpréter le langage de commande. C'est ce processus qui déclenche la demande de transfert.

Le couple facteur-pilote coopère à la réalisation des sorties de fichier tant que celles-ci se poursuivent normalement ; s'il se produit des erreurs de transmission, ou si l'opérateur désire arrêter un transfert, le processus servant intervient. Enfin, le processus premier n'intervient que pour transmettre la commande de transfert d'un fichier au couple facteur-pilote.

On obtient finalement le schéma suivant :



2.3 EXCLUSION MUTUELLE

2.31 INTRODUCTION AU PROBLÈME

Exemple 1. Un client d'un magasin a envoyé deux commandes distinctes portant sur des matériels différents. Ces deux commandes arrivent séparément au service de la comptabilité qui, pour chacune d'elles, établit une facture et tient à jour le compte du client. L'établissement des factures peut se faire indépendamment, dans n'importe quel ordre ou en même temps, mais on doit faire attention à ne pas modifier le compte en même temps, sinon on pourrait avoir la séquence suivante :

le compte, n, est lu pour la première facture ;
le compte, n, est lu pour la deuxième facture ;
le compte est modifié pour la première facture, il devient $n + n_1$;
le compte est modifié pour la deuxième facture et devient $n + n_2$.

La valeur finale du compte est $n + n_2$ au lieu de $n + n_1 + n_2$.

Exemple 2. Soit deux processus p et q qui produisent des données devant être imprimées sur une imprimante unique. L'emploi de cette imprimante par p exclut son emploi par q tant que l'impression pour p n'est pas terminée.

Exemple 3. En dehors de l'informatique, le même problème se retrouve dans le cas de trains ayant à circuler dans les deux sens sur un tronçon de voie unique.

Ces trois exemples illustrent la notion d'**exclusion mutuelle** : le compte du client doit être considéré comme une ressource à un seul point d'accès, de même que l'imprimante ou la voie unique.

Considérons la programmation, au niveau des processus, de l'exclusion mutuelle pour une ressource critique c donnée, et appelons **section critique** d'un processus, pour cette ressource, une phase du processus pendant laquelle la ressource c est utilisée et donc inaccessible aux autres processus.

Par hypothèse, les vitesses relatives des processus sont quelconques et inconnues ; nous supposons que tout processus sort de section critique au bout d'un temps fini.

Nous exigeons de la solution un certain nombre de propriétés :

a) à tous instant un processus au plus peut se trouver en section critique (par définition de la section critique),

b) si plusieurs processus sont bloqués en attente de la ressource critique, alors qu'aucun processus ne se trouve en section critique, l'un d'eux doit pouvoir y entrer au bout d'un temps fini (en d'autres termes, il faut éviter qu'un blocage mutuel des processus puisse durer indéfiniment),

c) si un processus est bloqué hors d'une section critique, ce blocage ne doit pas empêcher l'entrée d'un autre processus en sa section critique,

d) la solution doit être la même pour tous les processus, c'est-à-dire qu'aucun processus ne doit jouer de rôle privilégié.

Le lecteur comprendra mieux ces propriétés en étudiant le problème de Dekker (exercice 3). Si on disposait d'une instruction adéquate, le problème de l'exclusion mutuelle se résoudrait par :

exclusion mutuelle (section critique)

où *section critique* désigne une suite d'instructions utilisant la ressource critique. Les propriétés *a*), *b*), *c*), *d*) sont supposées vérifiées par l'instruction appelée ici *exclusion mutuelle*. Cette instruction se décompose en trois étapes :

```
exclusion mutuelle (section critique) : début
                                        entrée ;
                                        section critique ;
                                        sortie
                                        fin
```

Les instructions *entrée* et *sortie* doivent assurer le respect des propriétés *a*), *b*), *c*), *d*). La réalisation de ces instructions fait toujours appel, en dernier ressort, à un mécanisme câblé qui réalise une forme élémentaire d'exclusion mutuelle.

Nous présentons maintenant plusieurs schémas de réalisation.

2.32 ATTENTE ACTIVE

La solution la plus immédiate consiste à déclarer une variable p accessible aux processus, de valeur *1* ou *0* suivant que la ressource est occupée ou non ;

un processus doit consulter p pour pouvoir entrer en section critique, et remettre p à 0 en sortant.

Si l'on programme sans précautions, on se heurte alors à de nombreuses difficultés illustrées par l'exercice 3. Pour obtenir une solution simple, nous ferons appel à une instruction spéciale *Test And Set (TAS)* qui existe sur certaines machines ; cette instruction, agissant sur une variable m, peut se décrire ainsi :

```
instruction TAS(m) ;
début
bloquer l'accès à la cellule de mémoire m ;
lire le contenu de m ;
si m = 0 alors
   début
   m := 1 ;
   compteur ordinal := compteur ordinal + 2 ;
   commentaire : le compteur ordinal indique l'adresse de
                 l'instruction suivante du processus ;
   fin
sinon compteur ordinal := compteur ordinal + 1 ;
libérer l'accès à la cellule de mémoire m
fin
```

L'emploi de *TAS* conduit à la solution suivante :

Soit p la cellule de mémoire utilisée pour indiquer que la ressource critique R est occupée :

1) p est initialisée à 0.

2) La procédure *entrée* s'exprime par les deux instructions ci-après :

```
E : TAS(p) ;
aller à E
```

D'après ce qui précède, le processus ne pourra sortir de cette boucle, c'est-à-dire exécuter l'instruction suivant le branchement, que s'il trouve $p = 0$ pendant l'exécution de *TAS*.

3) La procédure *sortie* s'exprime par :

$$p := 0$$

La validité de cette solution tient à ce que le test et la mise à 1 de p ne peuvent être faits que par une seule instruction $TAS(p)$ à la fois. Le blocage de l'accès à la cellule de mémoire p assure, par câblage, l'exclusion mutuelle à la ressource critique.

Remarque. Pour programmer l'exclusion mutuelle à la ressource R, on a eu besoin d'un mécanisme élémentaire câblé d'exclusion mutuelle à une autre ressource p.

Dans la solution proposée, un processus bloqué sur p boucle sur l'instruction de test et monopolise un processeur, d'où le nom d'**attente active**. Cela est acceptable dans un système multiprocesseur si l'exclusion mutuelle survient rarement et dure peu. Nous allons étudier d'autres solutions où le processus bloqué perd l'unité centrale et entre dans une file d'attente.

2.33 LES VERROUS

Appelons **verrou** la variable p précédente et associons à un verrou une file d'attente $f(p)$. Si un processus ne peut entrer en section critique, il entre dans la file d'attente ; lorsqu'un processus sort de la section critique, un des processus de la file d'attente est activé, si celle-ci n'est pas vide ; il est inutile d'activer tous les processus à la fois car un seul pourra entrer en section critique. La valeur initiale de p est 0.

verrouiller (p) : *si* $p = 0$ *alors* $p := 1$ *sinon mettre le processus dans la file d'attente f(p), ce qui le fait passer à l'état bloqué ;*

déverrouiller (p) : *si f(p) n'est pas vide alors sortir un processus de f(p), ce qui le rend actif*
sinon $p := 0$;

Le verrou p et sa file $f(p)$ sont évidemment des ressources critiques qu'il faut protéger. Il est plus commode de considérer les deux procédures comme une seule ressource critique ; ce sont des procédures du système, car elles manipulent des files de processus ; nous les appellerons **primitives** parce qu'au niveau des processus appelants elles se comportent comme des instructions et leur exclusion mutuelle n'apparaît pas explicitement. Comme toute instruction, une primitive est indivisible pour l'observateur.

Comment résoudre l'exclusion mutuelle pour l'exécution des primitives ? Les solutions sont différentes suivant que le système possède un seul ou plusieurs processeurs :

a) Dans un système monoprocesseur, il suffit de rendre les procédures ininterruptibles ; on utilise le processeur comme ressource critique.

b) Dans un système multiprocesseur, cette condition ne suffit pas car elle n'empêche pas deux unités centrales de consulter et de modifier p et $f(p)$; on introduit donc une variable d'exclusion mutuelle dans les procédures, variable qui sera consultée par l'instruction *TAS* précédente. On retombe évidemment sur l'attente active mais cette attente dure au plus le temps d'une primitive ; par contre la section critique protégée par le verrou p peut durer un temps non négligeable et il devient rentable de programmer une file d'attente.

Une discussion plus détaillée sera donnée en 2.6.

2.34 LES SÉMAPHORES

Nous allons généraliser la solution précédente en utilisant une variable pouvant prendre des valeurs entières quelconques [Dijkstra, 67, 68].

2.341 **Définition**

Un **sémaphore** s est constitué d'une variable entière $e(s)$ et d'une file d'attente $f(s)$. La variable $e(s)$ peut prendre des valeurs entières positives, négatives ou nulles ; nous l'appellerons simplement la valeur du sémaphore. La politique de gestion de la file d'attente est laissée à la guise du concepteur du système.

Un sémaphore s est créé par une déclaration qui doit spécifier la valeur initiale $e0(s)$ de $e(s)$. Cette valeur initiale est nécessairement un entier non négatif. A la création d'un sémaphore, sa file $f(s)$ est toujours initialement vide.

On peut agir sur un sémaphore s par les deux seules primitives suivantes, qui sont des opérations indivisibles :

```
P(s) : début
       e(s) := e(s) − 1 ;
       si e(s) < 0 alors
         début
         commentaire : on suppose que cette primitive est exécutée par le
                       processus r ;
         état (r) := bloqué ;
         mettre le processus r dans la file f(s)
         fin
       fin ;
V(s) : début
       e(s) := e(s) + 1 ;
       si e(s) ⩽ 0 alors
         début
         sortir un processus de la file f(s) ;
         commentaire : soit q le processus sorti ;
         état (q) := actif
         fin
       fin ;
```

Remarque. La description de $V(s)$ n'indique pas comment se fait le choix du processus q, car ce choix dépend de la gestion des files d'attente qui varie selon le système. Cependant ce choix ne doit pas avoir d'influence sur le résultat final des actions entreprises par des processus coopérant à l'aide de P et V.

2.342 **Propriétés des sémaphores**

La définition des primitives P et V a les conséquences suivantes :

1) Un sémaphore ne peut être initialisé à une valeur négative, mais il peut devenir négatif après un certain nombre d'opérations P.

2) Soit $np(s)$ le nombre d'instructions P exécutées sur le sémaphore s,
$nv(s)$ le nombre d'instructions V exécutées sur le sémaphore s,
$e0(s)$ la valeur initiale du sémaphore s.

Il résulte de la définition de P et de V que

$$e(s) = e0(s) - np(s) + nv(s)$$

3) Soit $nf(s)$ le nombre de processus qui ont franchi la primitive $P(s)$, c'est-à-dire qui, ou bien n'ont pas été bloqués par celle-ci, ou bien ont été bloqués mais débloqués depuis ; à tout instant on a :

$$nf(s) \leqslant np(s)$$

Les effets de $P(s)$ et de $V(s)$ sur $nf(s)$ sont les suivants :

$P(s)$: $np(s) := np(s) + 1$;
si $np(s) \leqslant e0(s) + nv(s)$ alors $nf(s) := nf(s) + 1$;
commentaire : c'est le cas où $e(s) \geqslant 0$;

$V(s)$: $nv(s) := nv(s) + 1$;
si $np(s) \geqslant e0(s) + nv(s)$ alors $nf(s) := nf(s) + 1$;
commentaire : c'est le cas où $e(s) \leqslant 0$;

Théorème 1. L'exécution des primitives P et V laisse invariante la relation:

(1) $$nf(s) = \min (np(s), e0(s) + nv(s))$$

Supposons vérifiée la relation (1) et examinons l'effet de l'exécution de $P(s)$ et $V(s)$.

La relation (1) peut prendre deux formes suivant les valeurs relatives de $np(s)$ et de $e0(s) + nv(s)$. Nous aurons donc deux cas à examiner pour $P(s)$ et $V(s)$ (pour alléger l'écriture, nous supprimons dans les tableaux ci-après le nom du sémaphore s).

1) **Exécution de $P(s)$**

Forme initiale de la relation (1)	Relation après exécution de $np := np + 1$	Effet sur nf	Relations après exécution de P
$np < e0 + nv$ $\begin{cases} nf = np \\ nf < e0 + nv \end{cases}$	$np \leqslant e0 + nv$	$nf := nf + 1$	$nf = np$ $nf \leqslant e0 + nv$
$np \geqslant e0 + nv$ $\begin{cases} nf = e0 + nv \\ nf \leqslant np \end{cases}$	$np > e0 + nv$	*pas d'effet*	$nf = e0 + nv$ $nf < np$

On constate que dans chacun des cas la relation (1) reste vérifiée après exécution de $P(s)$.

2) **Exécution de $V(s)$**

Forme initiale de la relation (1)	Relation après exécution de $nv := nv + 1$	Effet sur nf	Relations après exécution de V
$np > e0 + nv \begin{cases} nf = e0 + nv \\ nf < np \end{cases}$	$np \geqslant e0 + nv$	$nf := nf + 1$	$nf = e0 + nv$ $nf \leqslant np$
$np \leqslant e0 + nv \begin{cases} nf = np \\ nf \leqslant e0 + nv \end{cases}$	$np < e0 + nv$	*pas d'effet*	$nf = np$ $nf < e0 + nv$

On constate à nouveau que la relation (1) demeure vraie dans chaque cas. Enfin la relation (1) est vraie pour les valeurs initiales qui sont :

$$np(s) = nv(s) = nf(s) = 0$$
$$e0(s) \geqslant 0 .$$

Cette relation, qui semble compliquée, s'interprète simplement en assimilant le sémaphore à une barrière : une opération P représente une demande de passage ; $e0 + nv$ représente le nombre total d'autorisations de passer jusqu'au moment présent ; la relation (1) exprime que le nombre effectif de passages égale le plus petit de deux nombres, à savoir le nombre de demandes et le nombre d'autorisations.

3) Si $e(s)$ est négative, sa valeur absolue égale le nombre des processus bloqués dans la file $f(s)$.

On a en effet (conséquence 2)

$$e(s) = e0(s) - np(s) + nv(s)$$

si $e(s) < 0$, on a $e0(s) + nv(s) < np(s)$

(1) donne alors :

$$nf(s) = e0(s) + nv(s)$$

et $$- e(s) = np(s) - nf(s)$$

4) Si $e(s)$ est positive ou nulle, sa valeur donne le nombre de processus pouvant franchir le sémaphore s sans se bloquer.

On trouvera dans [Habermann, 72] une étude théorique plus complète du problème.

2.343 Sémaphores d'exclusion mutuelle

L'exclusion mutuelle se résout comme suit : on introduit un sémaphore *mutex*, (abréviation pour « mutuelle exclusion »), initialisé à *1* et chaque

processus s'exécute selon le programme :

début
P(mutex) ;
section critique ;
V(mutex) ;
suite d'instructions ;
fin ;

Pour établir la validité de cette solution, il faut montrer :

— qu'à tout instant un processus au plus se trouve dans sa section critique,
— que lorsqu'aucun processus ne se trouve dans sa section critique, l'entrée en section critique se fait au bout d'un temps fini.

Théorème 2. A tout instant, un processus au plus se trouve dans sa section critique.

Le nombre de processus en section critique est égal à $nf(mutex) - nv(mutex)$.

Or, d'après le théorème 1 :

$$nf(mutex) = \min (np(mutex), 1 + nv(mutex))$$

d'où :

$$nf(mutex) - nv(mutex) \leqslant 1$$

Théorème 3. Si aucun processus ne se trouve en section critique, il n'y a pas de processus bloqué derrière le sémaphore d'exclusion mutuelle.

Si aucun processus ne se trouve en section critique, on a :

$$nf(mutex) = nv(mutex) \tag{2}$$

Si des processus attendent derrière *mutex*, on a :

$$nf(mutex) < np(mutex) \tag{3}$$

Les relations (1) et (3) donnent

$$nf(mutex) = nv(mutex) + 1$$

ce qui est incompatible avec (2).

2.35 DIFFICULTÉS DE L'EXCLUSION MUTUELLE

Les primitives présentées permettent de programmer aisément l'exclusion mutuelle entre processus, mais il ne suffit pas d'utiliser ces primitives pour garantir l'exclusion mutuelle ; il faut prendre certaines précautions :

— les modifications de verrous (en 2.33), de sémaphores (en 2.34) ne doivent se faire qu'à travers les primitives ; il est donc recommandé de protéger les tables des verrous ou des sémaphores contre l'écriture et de réserver aux primitives le droit d'y écrire.

— aucun processus ne doit pouvoir entrer dans une section critique, sans passer par le *P(mutex)* correspondant.

Si un processus est détruit en cours de section critique, il risque de bloquer d'autres processus. Il est alors nécessaire de détecter qu'il se trouve en section critique et de libérer artificiellement la section. Il peut être également utile de s'assurer qu'un processus ne reste qu'un temps fini à l'intérieur d'une section critique (cas de boucle ou de blocage).

Les solutions précédentes ne garantissent pas à tout processus d'entrer en section critique au bout d'un temps fini : si la file d'attente comporte des priorités, un processus de basse priorité risque d'attendre longtemps, voire indéfiniment. Par contre, si la file est gérée dans l'ordre des arrivées, tout processus entre nécessairement en section critique au bout d'un temps fini.

2.4 MÉCANISMES DE SYNCHRONISATION

2.41 GÉNÉRALITÉS

Les divers processus d'un système n'évoluent généralement pas indépendamment : il existe entre eux des relations qui dépendent de la logique de la tâche à accomplir et qui fixent leur déroulement dans le temps. Nous désignons l'ensemble de ces relations sous le terme de **synchronisation,** bien qu'elles ne fassent pas intervenir le temps comme mesure de durée, mais seulement comme moyen d'introduire une relation d'ordre entre des instructions exécutées par les processus.

Le problème de la synchronisation consiste donc à construire un mécanisme, indépendant des vitesses, permettant à un processus actif (soit p) :

— d'en bloquer un autre ou de se bloquer lui-même en attendant un signal d'un autre processus,

— d'activer un autre processus (soit q) en lui transmettant éventuellement de l'information.

Remarquons que, dans ce dernier cas, le processus q auquel est destiné le signal d'activation peut déjà se trouver à l'état actif ; il faut donc définir de façon plus précise l'effet de l'opération d'activation lorsqu'on se trouve dans cette circonstance. Deux possibilités se présentent :

a) le signal d'activation n'est pas mémorisé, et par conséquent il est perdu si le processus q ne l'attend pas,

b) le signal est mémorisé et le processus q ne se bloquera pas lors de la prochaine opération de blocage concernant ce processus.

A ce niveau de l'étude, nous n'avons fait aucune supposition sur les mécanismes qui permettent de réaliser ces opérations de synchronisation, appelées aussi primitives. Deux techniques au moins sont concevables :

a) le processus agit sur un autre processus en le désignant par son nom, ou bien agit sur lui-même : la synchronisation est dite **directe,**

b) le processus actionne un mécanisme qui agit sur d'autres processus : la synchronisation est alors **indirecte.**

Autrement dit, dans le premier cas l'identité du processus doit être un paramètre de l'opération d'activation (ou de blocage), alors que dans le second le nombre et l'identité des processus visés peuvent être inconnus du processus agissant.

2.42 MÉCANISMES D'ACTION DIRECTE

Les processus qui évoluent dans un système ne sont généralement pas tous au point ; certains processus peuvent par exemple boucler indéfiniment : il est alors indispensable de pouvoir les suspendre en les faisant passer à l'état bloqué. Pour réaliser explicitement le blocage d'un processus donné q, lorsque cette opération de synchronisation n'a pas été prévue au moment de l'écriture du programme, il est nécessaire de disposer d'un mécanisme d'action directe ; dans ce cas, l'identité du processus que l'on désire suspendre doit être un paramètre de la primitive de blocage.

Remarquons toutefois que même dans le cas où l'on dispose d'un mécanisme d'action directe pour réaliser le blocage d'un processus, l'instant où intervient cette action dans le cycle du processus à suspendre n'est pas toujours indifférent. En effet, pour assurer l'homogénéité des variables, certaines opérations exécutées par le processus à suspendre doivent être rendues logiquement ininterruptibles. Il est donc nécessaire, en pratique, de prévoir des dispositifs destinés à interdire qu'un processus soit bloqué par un autre pendant certaines phases de son activité (sections critiques par exemple).

Le mécanisme de synchronisation décrit dans [Saltzer, 66] est une bonne illustration d'un mécanisme d'action directe. Dans ce mécanisme, la coopération des processus est régie par deux opérations indivisibles *bloquer* et *éveiller*. En outre, à chaque processus q est associé un indicateur booléen noté *état (q)* qui indique à tout instant l'état, actif ou bloqué, du processus q.

La primitive *bloquer(q)* force le passage du processus q à l'état bloqué ; l'évolution du processus reprendra lorsque son état aura repris la valeur « actif ».

La primitive *éveiller(q)* a pour effet de rendre actif le processus q, s'il était bloqué ; toutefois si la primitive *éveiller(q)* est exécutée par un processus p alors que le processus q est encore actif le signal est perdu. Si on veut mémoriser un signal d'activation émis à l'intention du processus q, alors que celui-ci se trouve encore à l'état actif, on doit associer à chaque processus un indicateur booléen supplémentaire noté *témoin* (témoin d'éveil). Son effet sera de maintenir à l'état actif le processus q, lors de l'exécution de la prochaine primitive *bloquer(q)*.

Dans ces conditions, *bloquer(q)* provoquera le passage à l'état bloqué du processus q si et seulement si *témoin(q)* = *faux*.

Par contre si *témoin(q)* = *vrai*, il est remis à *faux* et l'effet de la primitive s'arrête là. L'algorithme de cette opération s'exprime comme suit :

bloquer(q) : *si* ¬ *témoin(q)* *alors*
état(q) := *bloqué*
sinon
témoin(q) := *faux*

Ainsi *bloquer(q)* exécuté par le processus *p* n'a aucune action sur ce processus ; tout se passe comme si le processus *q* exécutait *bloquer(q)* suivant l'algorithme précédent. Si le processus *q* est déjà bloqué, la primitive n'a aucun effet.

La primitive *éveiller(q)* active le processus *q*, s'il est bloqué. Par contre si le processus est encore actif lorsque la primitive est exécutée, *témoin(q)* est mis à *vrai* et l'activation est ainsi mémorisée. L'algorithme est le suivant :

éveiller(q) : *si* *état(q)* = *bloqué* *alors*
état(q) := *actif*
sinon *témoin(q)* := *vrai*

On note que le témoin d'éveil d'un processus peut mémoriser une — et une seule — activation éventuelle, alors que le processus est encore actif : c'est là une limitation de ce mécanisme.

2.43 MÉCANISMES D'ACTION INDIRECTE

Dans un mécanisme d'action directe, le nom du processus à synchroniser intervient explicitement comme paramètre des primitives d'activation ou de blocage ; dans un mécanisme d'action indirecte au contraire, la synchronisation met en jeu, non plus le nom du processus mais un ou plusieurs objets intermédiaires connus des processus coopérants, et manipulables par eux uniquement à travers des opérations indivisibles spécifiques. Ces objets intermédiaires qui appartiennent à la classe des données externes d'un processus portent les noms d'événements ou de sémaphores suivant la nature des opérations qui permettent de les manipuler.

2.431 **Synchronisation par événements**

Nous illustrerons le concept d'événement en nous référant aux langages de programmation qui permettent de manipuler ce concept.

Dans un langage de programmation évolué un événement est représenté par un identificateur ; il est créé par une déclaration qui fixe sa portée en tant qu'objet du langage. De plus un événement ne peut être manipulé que par certaines opérations particulières : ainsi un événement peut être attendu par le (ou les) processus qui y ont accès, ou bien il peut être déclenché. En outre, un événement peut être mémorisé ou non mémorisé.

a) *Evénement mémorisé*

Un **événement mémorisé** est représenté par une variable booléenne ; à un instant donné, la valeur, 1 ou 0, de cette variable traduit le fait que l'événement est ou n'est pas arrivé. Un processus se bloque si et seulement si l'événement qu'il attend n'est pas arrivé ; selon le système, le déclenchement d'un événement débloque un processus ou tous les processus qui l'attendent.

Cette notion de mémorisation peut, toutefois se traduire avec quelques différences dans les mécanismes proprement dits : en effet, l'événement mémorisé peut être remis à zéro

— soit explicitement, par un processus, au moyen d'une primitive particulière,

— soit implicitement dès qu'un processus qui l'attend est rendu actif.

Exemple. En PL/1, il est possible, en utilisant l'option *task*, d'initialiser des processus parallèles. Un événement est représenté par un symbole déclaré explicitement par l'attribut *event* ou implicitement lors de la création d'un processus (option *task*). L'affectation d'une valeur booléenne à un événement se fait au moyen d'une pseudo-variable de type *completion* de la façon suivante :

completion(evt) = *'0'B* ; indique que l'événement noté *evt* n'est pas arrivé ce qui équivaut à une « remise à zéro » de l'événement ;

completion(evt) = *'1'B* ; déclenche l'événement noté *evt*.

Un processus qui exécute l'instruction *wait(evt)* se bloque si et seulement si l'événement n'est pas arrivé, autrement dit si *completion(evt)* = *'0'B*.

De même que dans le cas des primitives *bloquer-éveiller*, on notera qu'on ne peut mémoriser qu'une activation d'un processus déjà actif.

b) *Evénement non mémorisé*

Lorsqu'il n'y a pas mémorisation, un événement émis alors qu'aucun processus ne l'attend est perdu. Par contre, si un ou plusieurs processus sont bloqués dans l'attente de cet événement au moment où il se produit, ces processus sont rendus actifs. Nous pouvons comparer l'événement non mémorisé à un message envoyé par radio, qui est reçu uniquement par les personnes à l'écoute à cet instant précis.

L'intérêt de l'événement non mémorisé, que l'on retrouve dans les langages spécialement conçus pour la commande de processus industriels, réside dans le fait que certaines informations prélevées sur des organes externes deviennent très rapidement caduques ; dans ces conditions il est souhaitable que le processus chargé de traiter ces informations soit activé uniquement lorsqu'il est bloqué en attente des dites informations, sinon elles sont perdues. Toutefois la notion d'événement non mémorisé est très délicate à manipuler, car elle met en jeu la vitesse des processus.

c) *Extensions de la notion d'événement*

Nous avons introduit le concept d'événement dans le cas le plus simple où la progression du processus dépend, en un point précis de son programme, de l'occurrence d'un événement et d'un seul. Cette façon d'envisager le problème est restrictive.

On peut en effet très bien imaginer que l'activation d'un processus soit associée à des entités plus complexes qu'un simple événement, par exemple à l'occurrence conjointe de deux événements ou à l'occurrence de l'un ou l'autre de deux événements et plus généralement à une expression booléenne d'événements. Cette idée est illustrée dans l'exemple suivant :

Exemple. Soit le programme PL/1 :

```
p1 : procedure ;
     ...
     call p2 event(e2) ;
     call p3 event(e3) ;
     wait (e2, e3) (i) ;
     ... ;
     end p1 ;
```

Dans ce programme, la tâche *p1*, pour nous conformer à la terminologie de PL/1 initialise au moyen d'une instruction *call* deux tâches parallèles, respectivement identifiées par *p2* et *p3*, avant d'exécuter une instruction *wait*. L'événement *e2* déclaré explicitement par l'attribut *event* à la création de la tâche *p2*, sera déclenché par la fin d'exécution de cette tâche ; il en est de même de l'événement *e3* pour la tâche *p3*.

Il en résulte que :

si $i = 2$ la tâche *p1* reste bloquée sur l'instruction *wait* jusqu'à l'arrivée des deux événements *e2* et *e3* ;

si $i = 1$ l'arrivée d'un seul des deux événements suffit à débloquer la tâche *p1* ;

si $i \leqslant 0$ l'instruction *wait* est sans effet et la tâche *p1* ne se bloque pas.

Il est parfois utile d'introduire le délai comme l'attente d'un événement particulier, et de subordonner l'évolution d'un processus à l'arrivée de cet événement. En PL/1, par exemple, un processus peut se bloquer lui-même pendant une période de temps finie, au moyen de l'instruction *delay* (< *expression élémentaire* >) où l'expression élémentaire, une fois évaluée et convertie, représente un nombre entier de millisecondes.

2.432 Synchronisation par sémaphores

Nous avons déjà rencontré le mécanisme des sémaphores pour résoudre le problème de l'exclusion d'accès à une ressource critique. Le même mécanisme est utilisable pour résoudre des problèmes généraux de synchronisation : un signal d'activation est envoyé par une primitive *V*, il est attendu par une primitive *P*.

Un sémaphore *s* est un **sémaphore privé** d'un processus *p* [Dijkstra, 68] si seul ce processus peut exécuter l'opération *P(s)* ; les autres processus ne peuvent agir sur *s* que par *V(s)*.

Ainsi un processus dont l'évolution est subordonnée à l'émission d'un signal par un autre processus se bloque, au moyen d'une primitive *P*, derrière son sémaphore privé initialisé à zéro. Le signal de réveil de ce processus bloqué est obtenu en faisant exécuter par un autre processus une opération *V* sur le même sémaphore.

Exemple 1. Relations d'ordre entre deux processus.

L'activation d'un processus *p* dont l'évolution est subordonnée à l'émission d'un signal par un processus *q* se programme comme suit, en introduisant le sémaphore *signal* initialisé à *0*.

sémaphore signal ; (valeur initiale = 0)

processus p : début	*processus q : début*
A1 ; . . . ; Ai ;	*B1 ; . . . ; Bj ;*
P(signal) ;	*V(signal) ;*
. . . ;	*. . . ;*
fin ;	*fin ;*

Dans cet exemple, deux cas peuvent se présenter :

— ou bien le processus *p* est déjà bloqué sur la primitive *P(signal)* lorsque le signal arrive — autrement dit lorsque le processus *q* exécute la primitive *V(signal)* — et le réveil est alors effectif,

— ou bien le processus *p* est actif lorsque le signal est émis (il exécute par exemple l'instruction *Ai*) et tout se passe comme si le signal était mémorisé ; en effet la valeur du sémaphore *signal* est passée à *1* et lorsque le processus *p* exécutera la primitive *P* il ne se bloquera pas.

Si l'on suppose maintenant que le processus *q* exécute *n* fois la primitive *V(signal)* alors que le processus *p* exécute l'instruction *Ai*, la valeur *n* prise par le sémaphore *signal* mémorisera l'arrivée des *n* signaux d'activation et le processus *p* disposera alors d'un potentiel de *n* activations : en conséquence son blocage sera effectif seulement lorsqu'il exécutera la *(n + 1)*-ième opération *P(signal)*, en supposant que le sémaphore n'ait pas été modifié entre temps.

A ce niveau, le sémaphore apparaît donc comme un mécanisme de synchronisation suffisamment général pour permettre, à la différence des mécanismes précédemment exposés, de mémoriser un nombre quelconque d'activations éventuelles alors que le processus auquel elles sont destinées se trouve encore à l'état actif.

Il est possible de combiner l'emploi des sémaphores d'exclusion mutuelle et des sémaphores privés, pour réaliser des modèles de synchronisation plus complexes. D'une façon générale, toutes les fois qu'un processus, pour poursuivre ou non son évolution, a besoin de connaître la valeur de certaines variables d'état, qui peuvent être modifiées par d'autres processus, il ne peut les consulter que dans une section critique. Comme il ne peut se bloquer à

l'intérieur de celle-ci, le schéma suivant est utilisé :

P(mutex) ;
modification et test des variables d'état ;
si non blocage alors V(sempriv) ;
V(mutex) ;
P(sempriv) ;

Si le test des variables d'état indique que le processus peut continuer, alors il exécute une opération *V* sur son sémaphore privé *sempriv* initialisé à *0*, sinon l'opération *V* est sautée. A la sortie de la section critique, les variables d'état indiquent aux autres processus si l'un d'eux doit ou non exécuter *V(sempriv)*.

La séquence des actions mises en jeu par un processus activateur s'écrit :

P(mutex) ;
modification et test des variables d'état, suivi éventuellement d'une opération V sur le sémaphore privé sempriv ;
V(mutex) ;

L'exemple suivant va nous permettre de préciser ces schémas.

Exemple 2. Problème des philosophes et des spaghetti [Dijkstra, 71].

Cinq philosophes, réunis pour philosopher, ont au moment du repas un problème pratique à résoudre ; en effet le repas est composé de spaghetti qui, selon le savoir-vivre de ces philosophes, se mangent avec deux fourchettes. Or, la table n'est dressée qu'avec une seule fourchette par couvert. Après quelques instants de réflexion, les philosophes décident d'adopter le rituel suivant :

1) Chaque philosophe prend place à un emplacement fixe.

2) Tout philosophe qui mange utilise la fourchette de droite et celle de gauche. Il ne peut pas en emprunter d'autres ; deux philosophes voisins ne peuvent donc pas manger en même temps.

3) A tout instant, chaque philosophe se trouve dans l'un des trois états suivants :

— ou bien il mange,

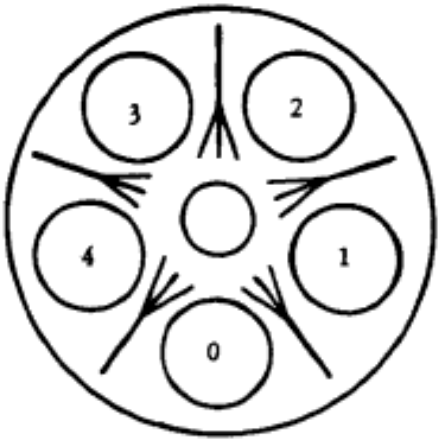
— ou bien il a décidé de manger, et ne peut satisfaire son désir par manque de fourchette ; dans ce cas il attend jusqu'à ce que les deux fourchettes (celle de droite et celle de gauche) soient disponibles,

— ou bien il pense et il a la politesse de n'utiliser aucune fourchette.

4) Initialement tous les philosophes pensent.

5) Tout philosophe qui mange cesse de manger au bout d'un temps fini.

Le comportement de chaque convive se réduit donc à une succession d'intervalles quelconques de réflexion et de ripaille ! Le rituel choisi satisfait les règles de savoir-vivre de ces philosophes, mais doit être complété si l'on veut éviter tout conflit : le cas typique d'un conflit conduisant à un blocage irrémédiable de la docte assemblée est celui où tous les commensaux décident de manger et saisissent au même moment leur



fourchette de droite, interdisant du même coup à leur voisin de droite l'usage de la seconde fourchette. De même un philosophe pourrait attendre indéfiniment s'il n'était pas averti de la libération des fourchettes dont il a besoin.

Le problème consiste donc à compléter le rituel choisi en faisant les hypothèses suivantes :

— les activités des philosophes, *penser* et *manger*, sont strictement séquentielles et n'ont d'interaction avec les activités des autres philosophes qu'au début et à la fin de leur exécution,

— les actions se déroulent à des vitesses quelconques, non nulles,

— le comportement de chaque convive peut être assimilé à un processus cyclique.

Une solution

Il s'agit d'élaborer un mécanisme qui permette à ces cinq processus de coopérer sans étreinte fatale ; notons tout de suite que, les philosophes jouant le même rôle, le mécanisme de synchronisation sera identique pour tous.

Associons 3 états à chaque philosophe i :

$c[i] = 0$ lorsqu'il pense ;
$c[i] = 1$ lorsqu'il voudrait bien manger, mais ne le peut pas par manque de fourchette ;
$c[i] = 2$ lorsqu'il mange.

Le passage de l'état $c[i] = 0$ à $c[i] = 2$ n'est possible, d'après l'hypothèse 2, que si

(4) $$(c[i + 1] \neq 2) \underline{et} (c[i - 1] \neq 2)\ (*)$$

Si cette condition n'est pas réalisée, le philosophe i passe dans l'état $c[i] = 1$ et se bloque. Cette impossibilité d'évolution du processus i est obtenue en introduisant un sémaphore privé, noté *sempriv*[i], initialisé à zéro.

Naturellement, le test de la condition (4) et les conséquences qui en résultent constituent une section critique à protéger par un sémaphore d'exclusion mutuelle, noté *mutex*.

(*) Dans cet exemple, les opérations sur les indices sont faites modulo 5.

Les actions effectuées par le philosophe i pour demander des fourchettes s'écrivent ainsi :

```
B1 : P(mutex) ;
     si (c[i + 1] ≠ 2) et (c[i − 1] ≠ 2) alors
          début
          c[i] := 2 ;
          V(sempriv[i])
          fin
     sinon c[i] := 1 ;
     V(mutex) ;
     P(sempriv[i]) ;
```

Si le test de la condition (4) indique que le processus i peut continuer, alors il exécute une opération $V(sempriv[i])$ — la valeur du sémaphore passe de 0 à 1 — sinon l'opération est sautée, laissant aux autres processus l'obligation d'exécuter une opération V à un moment favorable. Dans tous les cas, à la sortie de la phase critique protégée par *mutex*, la variable d'état $c[i]$ reflète le nouvel état dans lequel va passer le processus i et par conséquent l'obligation (ou non) pour les autres processus de le réveiller, s'il se trouve bloqué sur un sémaphore privé.

Le passage de l'état $c[i] = 2$ à $c[i] = 0$ entraîne le réveil des philosophes $(i + 1)$ et $(i - 1)$ si les deux conditions suivantes sont remplies :

— d'une part ces derniers avaient décidé de manger, autrement dit $c[k] = 1$ pour $k = i + 1$ et $i - 1$;

— d'autre part on est sûr qu'ils disposeront de l'autre fourchette c'est-à-dire $c[k] \neq 2$ pour $k = i + 2$ et $i - 2$.

La séquence des actions effectuées s'écrit alors :

```
B2 : P(mutex) ;
     c[i] := 0 ;
     si (c[i + 1] = 1) et (c[i + 2] ≠ 2) alors
          début
          c[i + 1] := 2 ;
          V(sempriv[i + 1])
          fin ;
     si (c[i − 1] = 1) et (c[i − 2] ≠ 2) alors
          début
          c[i − 1] := 2 ;
          V(sempriv[i − 1])
          fin ;
     V(mutex) ;
```

Nous donnons l'algorithme de synchronisation complet dans lequel les séquences *B1* et *B2* sont remplacées par une procédure unique *test(k)*.

```
entier tableau c[0 : 4] ; (valeurs initiales = 0)
sémaphore tableau sempriv[0 : 4] ; (valeurs initiales = 0)
sémaphore mutex ; (valeur initiale = 1)
procédure test(k) ; valeur k ; entier k ;
```

```
        si(c[k] = 1) et c[k + 1] ≠ 2) et (c[k − 1] ≠ 2) alors
            début
            c[k] := 2;
            V(sempriv[k])
            fin;
processus philosophe i
    Li : début
        penser;
                P(mutex);
                c[i] := 1;
                test(i);
                V(mutex);
                P(sempriv[i]);
        manger;
                 P(mutex);
                 c[i] := 0;
                 test(i − 1);
                 test(i + 1);
                 V(mutex);
        aller à Li;
        fin;
```

Remarque. Le lecteur pourra constater que cette solution présente un point faible puisqu'elle n'interdit pas à certains philosophes de se coaliser au détriment d'un autre ; si, par exemple, les philosophes qui se succèdent pour manger respectent indéfiniment l'ordre suivant :

(0,3), (0,2), (4,2), (0,2), (0,3) ... etc.

alors le philosophe *1* est condamné à mourir de faim !

2.44 CRITIQUE DES MÉCANISMES DE SYNCHRONISATION

Aucun des mécanismes étudiés précédemment ne peut répondre à tous les besoins, du moins de façon commode. En effet si les primitives introduites permettent de traduire la coopération des processus sous leur aspect temporel, elles ne fournissent aucun dispositif de communication d'un processus à l'autre ; en pratique cette insuffisance peut être masquée en introduisant des variables communes auxquelles il faut assurer l'exclusion mutuelle d'accès, ce qui alourdit le programme.

Evénements et sémaphores sont utiles lorsqu'un processus ignore, en raison de la nature du problème, l'identité des processus avec lesquels il coopère ; les interactions doivent toutefois avoir été prévues dès l'écriture du programme car événements et sémaphores ne fournissent aucun mécanisme permettant à un processus d'en bloquer un autre, si l'algorithme de ce dernier ne comporte à l'avance aucune primitive de blocage ; cela est particulièrement gênant lorsqu'on désire faire surveiller par un processus maître l'exécution d'un processus en cours de mise au point ou lorsqu'on désire suspendre un processus qui boucle.

Alors qu'un processus peut attendre plusieurs événements avec l'instruction *wait* généralisée, il ne peut agir que sur un seul sémaphore à la fois avec la primitive *P*. Alors que dans certains systèmes, un événement déclenché libère tous les processus qui l'attendent, la primitive *V* sur un sémaphore n'en libère qu'un à la fois ; pour pallier cette dernière limitation, on peut généraliser les primitives *P* et *V* en autorisant une variation quelconque (au lieu de ± 1) de la valeur du sémaphore (exercice 8).

2.5 COMMUNICATION ENTRE PROCESSUS

2.51 INTRODUCTION

La coopération de plusieurs processus à l'exécution d'une tâche commune nécessite en général une communication d'information entre ces processus. Les primitives de synchronisation étudiées précédemment réalisent un mode de communication où l'information transmise est réduite à la forme élémentaire d'une autorisation ou d'une interdiction de continuer l'exécution au-delà d'un certain point prédéterminé. Le message se réduit donc dans ce cas à un potentiel d'activation.

Ce mode de communication ne suffit pas à tous les besoins. Ainsi, lorsque les actions exécutées par un processus après son activation dépendent de l'identité du processus activateur, cette identité doit pouvoir être transmise au processus activé.

La communication d'information entre des processus implique l'accès de ces processus à un ensemble de variables globales constituant un univers commun. Si l'accès des processus à cet univers n'est soumis à aucune restriction *a priori*, les processus en communication doivent s'imposer un mode d'emploi des variables communes garantissant le bon fonctionnement de la communication. Il se pose alors des problèmes de sécurité, en particulier pour la protection des zones communes en cas de fonctionnement défectueux d'un processus. Un remède consiste à imposer que tout accès aux variables communes soit contrôlé. On est ainsi conduit à un type de solution où toutes les communications se font par des mécanismes spéciaux, sous le contrôle du système.

2.52 COMMUNICATION ENTRE PROCESSUS PAR VARIABLES COMMUNES

Les problèmes les plus généraux de la communication entre processus peuvent être résolus en rendant un ensemble de variables communes accessibles à tous les processus. Toutefois l'accès simultané de plusieurs processus à de telles variables pose des problèmes de cohérence qui ont été développés en 2.3 à propos de l'exclusion mutuelle. Les processus doivent donc s'imposer une

règle du jeu plus ou moins élaborée suivant la nature de la communication. Une règle simple consisterait à inclure tout accès à des données communes dans une section critique ; toutefois, des considérations d'efficacité amènent à réaliser des sections critiques aussi brèves que possible, et en particulier à éviter le blocage de processus à l'intérieur d'une telle section.

On peut faire deux remarques sur ce mode de communication général par accès à des variables communes :

— les règles de communications que doivent observer les processus ne peuvent leur être imposées car on n'a aucune garantie contre le non-respect de ces règles par un processus défectueux,

— la communication par consultation et par modification de variables communes se prête assez mal à une interaction du type « envoi de messages » ; un tel mode d'interaction entre des processus p et q serait par exemple :

1) p envoie un message à q (c'est-à-dire fournit de l'information et prévient q que cette information est disponible). Puis p peut attendre ou ne pas attendre un accusé de réception de q pour continuer.

2) q, qui attendait un message, reçoit le message de p et signale, ou ne signale pas, sa réception à p.

L'arrivée d'un message impliquant le réveil du destinataire en attente, on voit qu'un dispositif de synchronisation doit être incorporé dans le protocole de communication.

Les deux remarques qui précèdent vont guider le plan de notre étude. Nous examinerons d'abord la réalisation, à l'aide de variables communes, d'un dispositif de communication réunissant synchronisation et transmission d'information ; puis nous montrerons, à partir d'exemples, comment les contraintes de sécurité peuvent conduire à des mécanismes comportant un contrôle de la communication, supprimant en fait la notion de variables communes.

2.521 **Modèle du producteur et du consommateur**

Le schéma connu sous le nom de « modèle du producteur et du consommateur » permet de présenter les principaux problèmes de la communication entre processus par accès à des variables communes avec synchronisation. On considère deux processus, le producteur et le consommateur, qui se communiquent de l'information à travers une zone de mémoire, dans les conditions suivantes :

— l'information est constituée par des messages de taille constante,

— aucune hypothèse n'est faite sur les vitesses respectives des deux processus.

La zone de mémoire commune, ou tampon, a une capacité fixe de n messages $(n > 0)$. L'activité des deux processus se déroule schématiquement suivant

le cycle décrit ci-après :

PRODUCTEUR	*CONSOMMATEUR*
PROD : Produire un message ;	*CONS : Prélever un message*
Déposer un message	*dans le tampon ;*
dans le tampon ;	*Consommer le message ;*
aller à PROD ;	*aller à CONS ;*

On souhaite que la communication se déroule suivant les règles ci-après :

— exclusion mutuelle au niveau du message : le consommateur ne peut prélever un message que le producteur est en train de ranger ;

— le producteur ne peut pas placer un message dans le tampon si celui-ci est plein (on s'interdit de perdre des messages par surimpression) ; le producteur doit alors attendre ;

— le consommateur doit prélever tout message une fois et une seule ;

— si le producteur est en attente parce que le tampon est plein, il doit être prévenu dès que cette condition cesse d'être vraie ; il en est de même pour le consommateur et la condition « tampon vide ».

Pour représenter de façon plus précise l'état du système, introduisons deux variables caractérisant l'état du tampon en dehors des phases de communication proprement dites (les deux processus se trouvent donc dans leur phase de production ou de consommation) :

nplein = nombre de messages attendant d'être prélevés,
nvide = nombre d'emplacements disponibles dans le tampon.
Initialement, *nplein* = 0, *nvide* = *n*.
L'algorithme des deux processus s'écrit alors :

PRODUCTEUR	*CONSOMMATEUR*
entier nplein = 0, nvide = n ;	
PROD : Produire un message ;	*CONS : { nplein := nplein − 1 ;*
{ nvide := nvide − 1 ;	*si nplein = − 1 alors*
si nvide = − 1 alors	*attendre ; }*
attendre ; }	
Déposer le message ;	*Prélever un message ;*
{ nplein := nplein + 1 ;	*{ nvide := nvide + 1 ;*
si consommateur en attente alors	*si producteur en attente alors*
réveiller consommateur ; }	*réveiller producteur ; }*
aller à PROD ;	*Consommer le message ;*
	aller à CONS ;

Les parties notées entre accolades doivent se dérouler de façon indivisible, puisqu'elles comprennent le test et la modification de variables critiques.

Considérons à présent le test sur la condition « consommateur en attente » dans le processus producteur. On peut remarquer qu'en raison du caractère

indivisible de la séquence de début du consommateur, on peut remplacer la condition « consommateur en attente » par la condition $nplein = -1$ qui lui est alors équivalente (en fait, on compare *nplein* à *0* puisqu'on a fait $nplein := nplein + 1$). Le test et la modification de cette condition se font comme suit :

PRODUCTEUR	*CONSOMMATEUR*
⋮	⋮
$nplein := nplein + 1$;	$nplein := nplein - 1$;
si $nplein = 0$ *alors réveiller le consommateur* ;	*si* $nplein = -1$ *alors attendre* ;
⋮	⋮

Remarquons enfin que les conditions $nplein = 0$ et $nplein = -1$ entraînent respectivement $nplein \leqslant 0$ et $nplein < 0$. On voit alors que *nplein* fonctionne en fait comme un sémaphore avec la restriction, due à l'unicité du consommateur, qu'un processus au plus peut se trouver bloqué dans sa file. On peut faire la même remarque pour le compteur *nvide* et le producteur. L'algorithme des deux processus peut maintenant s'écrire :

sémaphore $nplein = 0, nvide = n$;

PROD : *Produire un message* ;	*CONS* : $P(nplein)$;
$P(nvide)$;	*Prélever un message* ;
Déposer le message ;	$V(nvide)$;
$V(nplein)$;	*Consommer le message* ;
aller à PROD ;	*aller à CONS* ;

Analysons le fonctionnement du système qui vient d'être décrit.
Reprenant les notations du 2.34, on note pour un sémaphore s :

$np(s)$ le nombre d'opérations P exécutées sur ce sémaphore,
$nv(s)$ le nombre d'opérations V exécutées sur ce sémaphore,
$nf(s)$ le nombre de fois qu'un processus a franchi une primitive $P(s)$.

On a d'après le théorème 1 :

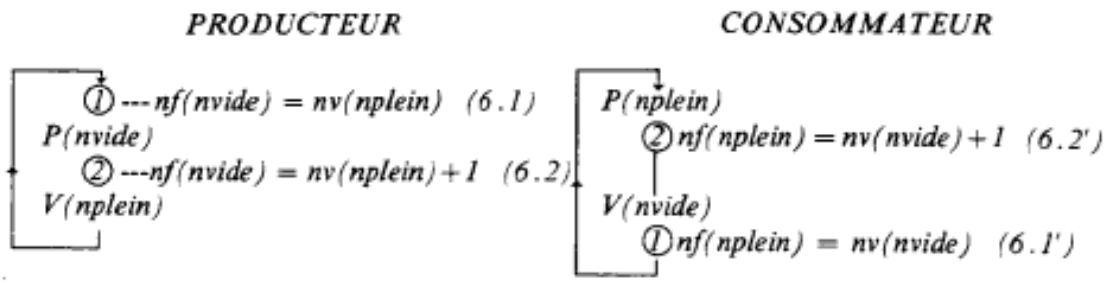
$$nf(s) = \min\,(np(s), e0(s) + nv(s))$$

Remplaçant successivement s par *nplein* et *nvide*, on obtient :

$$(5) \qquad \begin{cases} nf(nplein) = \min\,(np(nplein), nv(nplein)) \\ nf(nvide) = \min\,(np(nvide), n + nv(nvide)) \end{cases}$$

Pour chacun des deux processus, appelons phase 1 la phase de production ou de consommation du message, phase 2 la phase de dépôt ou de retrait du message.

Suivant la phase où se trouvent les deux processus, les variables *nf* et *nv* vérifient les relations illustrées sur le schéma ci-après :



Les relations de la phase 1 sont vraies à l'instant initial, tous les *nf* et *nv* étant nuls. Les transitions entre les relations des phases 1 et 2 résultent directement de l'effet des opérations *P* et *V*.

Nous démontrerons deux propriétés du système :

Propriété 1. Le producteur et le consommateur ne peuvent être bloqués simultanément.

Si tel était le cas, en effet, on aurait :

$$nf(nvide) < np(vide) \quad \text{(blocage du producteur)}$$
$$nf(nplein) < np(nplein) \quad \text{(blocage du consommateur)}$$

D'après (5), on aurait donc :

$$nf(nvide) = n + nv(nvide)$$
$$nf(nplein) = nv(nplein)$$

Combinant ces relations avec (6.1) et (6.1'), on obtient :

$$nv(nvide) = n + nv(nvide)$$

Cette égalité est incompatible avec l'hypothèse $n > 0$. L'interblocage est donc impossible.

Propriété 2. Lorsque les messages sont consommés dans l'ordre de leur production, le producteur et le consommateur n'opèrent jamais simultanément sur le même message.

Aucune hypothèse n'a jusqu'à présent été faite sur les procédures de dépôt et de retrait des messages.

Nous supposerons (ce cas se présente fréquemment dans la pratique) que les messages doivent être consommés dans l'ordre de leur production. Le tampon étant considéré comme formé de n cases numérotées de 0 à $n - 1$, une technique classique consiste à l'utiliser de façon circulaire, à l'aide de deux pointeurs :

queue : pointe vers la première case vide ;
tête : pointe vers la première case contenant un message à prélever.

Initialement, $tête = queue = 0$.

Les procédures de dépôt et de retrait d'un message s'écrivent :

déposer (message) :	*prélever (message) :*
tampon [queue] := message ;	*message := tampon [tête] ;*
queue := queue + 1 mod n ;	*tête := tête + 1 mod n ;*

Si le producteur et le consommateur opéraient simultanément sur le même message, on aurait l'égalité :

$$tête = queue,$$

les deux processus se trouvant dans leur phase 2 (dépôt ou retrait).

Les relations (6.2) et (6.2') sont alors vérifiées :

$$nf(nvide) = nv(nplein) + 1$$
$$nf(nplein) = nv(nvide) + 1$$

Les relations (5) permettent en outre d'écrire :

$$nf(nvide) \leqslant n + nv(nvide)$$
$$nf(nplein) \leqslant nv(nplein)$$

Eliminant les *nf* des 4 relations précédentes, on obtient :

$$0 < nv(nplein) - nv(nvide) < n \tag{7}$$

Or, les processus étant à leur phase 2, le pointeur *tête* (respectivement *queue*) a été augmenté de 1 à chaque franchissement de *nplein* (respectivement *nvide*).

On peut donc écrire :

$$tête = (tête)_0 + nf(nplein) \underline{mod}\ n = nf(nplein) \underline{mod}\ n$$
$$queue = (queue)_0 + nf(nvide) \underline{mod}\ n = nf(nvide) \underline{mod}\ n$$

L'égalité $tête = queue$ implique donc :

$$nf(nplein) - nf(nvide) = 0 \underline{mod}\ n$$

ou encore, d'après (6.2) et (6.2') :

$$nv(nplein) - nv(nvide) = 0 \underline{mod}\ n\ ,$$

en contradiction avec la double inégalité (7).

Le lecteur vérifiera que la relation $tête = queue$ peut être vérifiée quand un processus au moins ne se trouve pas dans sa phase 2. Elle traduit alors, comme à l'instant initial, le fait que le tampon ne contient aucun message.

Il est possible de démontrer simplement les résultats précédents sans faire appel au théorème 1 (exercice 9).

Dans la construction du mécanisme de synchronisation, l'hypothèse de l'unicité du producteur et du consommateur ne joue pas un rôle essentiel : le

même raisonnement s'appliquerait à un nombre quelconque de processus de l'une et l'autre classe (il suffit, par exemple, dans le programme des producteurs, de remplacer la condition « consommateur en attente » par « au moins un consommateur en attente », les sémaphores *nplein* et *nvide* pouvant alors prendre des valeurs inférieures à -1). Le schéma précédent pourrait donc s'appliquer à un ensemble de producteurs et de consommateurs partageant un tampon commun. Toutefois, rien ne garantit maintenant l'exclusion mutuelle de l'accès au tampon pour des processus d'une même classe, et cette exclusion doit être explicitement programmée.

Moyennant cette modification, l'exclusion mutuelle entre producteurs et consommateurs se démontre comme précédemment, en utilisant le fait qu'un producteur et un consommateur au plus peuvent simultanément se trouver dans leur phase de dépôt ou de retrait.

Le programme des producteurs et des consommateurs s'écrit :

```
        PRODUCTEUR                          CONSOMMATEUR
                    sémaphore nplein = 0, nvide = n,
                              mutexprod = 1, mutexcons = 1 ;
                    entier tête = 0, queue = 0 ;
PROD : Produire (message 1) ;        CONS : P(nplein) ;
       P(nvide) ;                           P(mutexcons) ;
       P(mutexprod) ;                       message 2 := tampon [tête] ;
       tampon[queue] := message 1 ;         tête := tête + 1 mod n ;
       queue := queue + 1 mod n ;           V(mutexcons) ;
       V(mutexprod) ;                       V(nvide) ;
       V(nplein) ;                          Consommer (message 2) ;
       aller à PROD ;                       aller à CONS ;
```

2.522 Communication par boîte aux lettres

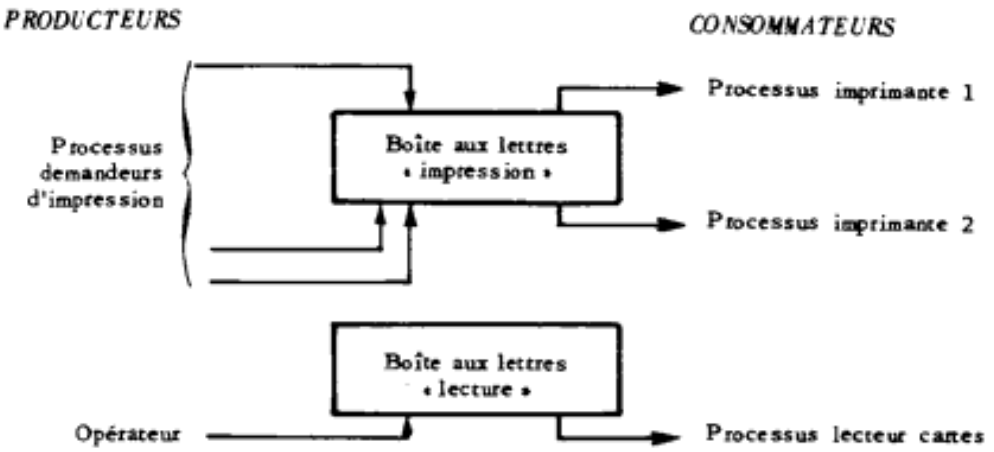
La communication entre processus suivant le schéma dit de la boîte aux lettres est une application directe du modèle du producteur et du consommateur. Une boîte aux lettres est une zone de mémoire permettant la communication entre ces processus suivant le schéma qui vient d'être défini. Remarquons que, lors du dépôt d'un message par un producteur, un consommateur quelconque se trouve activé, et pas nécessairement le destinataire prévu ; on voit donc que le dispositif n'est efficace que si tous les destinataires sont équivalents (c'est-à-dire s'il est indifférent, pour accomplir la tâche demandée, d'activer l'un quelconque de ces processus). Sinon, l'identité du destinataire doit faire partie du message, ce qui implique un tri à la réception. La solution généralement retenue consiste donc à prévoir une boîte aux lettres par classe de processus équivalents, évitant ainsi tout tri à la réception.

Exemple. Soit un système d'entrée-sortie gérant un lecteur de cartes et deux imprimantes, ces trois périphériques pouvant fonctionner en parallèle. Les opérations possibles sont :

— imprimer le contenu d'un fichier,
— lire des cartes et placer leur contenu dans un fichier.

L'impression d'un fichier peut être demandée par un processus quelconque (correspondant à l'exécution d'un programme d'utilisateur). La lecture de cartes ne peut être demandée, pour des raisons de sécurité de manipulation, que par une commande introduite à la console de l'opérateur.

Si on veut exploiter complètement le parallélisme, on prévoira trois processus consommateurs, un par périphérique ; les deux processus gérant les imprimantes sont équivalents, en ce sens qu'il est indifférent qu'un fichier soit imprimé sur l'une ou l'autre imprimante. Les demandes d'entrée-sortie utilisent deux boîtes aux lettres suivant le schéma ci-après :



Les messages transmis dans les boîtes aux lettres contiennent l'identité du fichier concerné par l'entrée-sortie et peuvent comporter des renseignements complémentaires (entrée en binaire ou symbolique, etc...).

Le fonctionnement d'une boîte aux lettres étant ainsi défini dans son principe, il reste à examiner les modalités de son implantation.

— Allocation d'espace pour la(ou les) boîte(s) aux lettres.

— Accès à la boîte aux lettres : celle-ci doit être adressable par tous les processus qui l'utilisent. Cette contrainte est à la source des principaux problèmes d'implantation d'un système de communication par boîtes aux lettres, en particulier dans le cas où chaque processus dispose d'un espace d'adressage distinct.

— Protection contre les actions intempestives.

Ces problèmes doivent trouver pour chaque implantation une solution spécifique dépendant des caractéristiques du système. Nous nous bornerons donc à quelques remarques générales :

— L'allocation d'espace peut se faire statiquement ; c'est la solution la plus simple. A la création d'un processus, l'espace nécessaire pour sa boîte aux lettres est réservé une fois pour toutes. Une solution moins coûteuse en espace consiste à allouer une zone fixe à l'ensemble des boîtes aux lettres et à partager dynamiquement cet espace suivant les demandes. Le cas de la saturation de la zone doit alors être prévu.

— Les problèmes d'accès et de protection peuvent être liés ; en effet, une façon d'assurer la protection est de n'autoriser l'accès à la boîte aux lettres qu'à travers une procédure assurant un contrôle de validité. Un contrôle préalable peut être fait sur l'identité du processus : par exemple, le droit d'accès aux diverses boîtes aux lettres du système peut faire partie du pouvoir des processus.

2.53 MÉCANISMES SPÉCIAUX DE COMMUNICATION

Nous allons maintenant étudier deux exemples de mécanismes spéciaux de communication. Leurs caractéristiques communes sont :

— le passage obligatoire par des procédures spéciales pour l'accès aux variables critiques,

— la présence d'un dispositif de synchronisation.

2.531 **Sémaphores avec messages** [Saal, 70]

Le principe du sémaphore avec message découle de la remarque suivante : lorsqu'un processus p active un processus q par une opération V sur son sémaphore privé, cette activation accompagne souvent une transmission d'information (par exemple l'identité de p, la nature de l'action requise de q par p, ...). On peut dès lors songer à inclure cette transmission d'information dans le mécanisme même des primitives de synchronisation.

On définit ainsi deux primitives P_M et V_M agissant sur des sémaphores et généralisant P et V :

L'opération V_M est l'opération V complétée par un envoi de message. L'opération P_M est l'opération P complétée par la réception d'un message. Pour la description de ces opérations, on introduira des variables de type *message* ; les opérations P_M et V_M seront représentées par des procédures ayant de telles variables comme paramètres.

Soit *message reçu* une variable de type *message* locale à un processus p.

L'opération

$$P_M(s, \textit{message reçu})$$

exécutée par p est équivalente à

$$\left\{ \begin{array}{l} P(s) ; \\ \text{affectation d'une valeur à } \textit{message reçu} ; \end{array} \right\}$$

De même, soit *message transmis* une variable de type *message* locale à un processus p. L'opération

$$V_M(s, \textit{message transmis})$$

exécutée par p équivaut à :

$$\left\{ \begin{array}{l} V(s) ; \\ \text{passage de la valeur de } \textit{message transmis} ; \text{ cette valeur sera} \\ \text{affectée au } \textit{message reçu} \text{ d'un processus } q \text{ qui franchira } P(s) \end{array} \right\}$$

De façon plus précise, on associe à chaque sémaphore s une file de processus $f_P(s)$ et une file de messages $f_M(s)$. On notera par une flèche ($\rightarrow$) le transfert d'un processus ou d'un message dans (ou hors de) la file correspondante. Les opérations P_M et V_M peuvent alors se décrire comme suit, en désignant par p le processus qui les exécute :

```
P_M(s, message reçu(p))
    début
    e(s) := e(s) - 1 ;
    si e(s) < 0 alors
        début
        état (p) := bloqué ;
        p → f_P(s)
        fin
    sinon
        début
        choisir m ∈ f_M(s) ;
        f_M(s) → m ;
        message reçu := m
        fin
    fin ;
```

```
V_M(s, message transmis)
début
e(s) := e(s) + 1 ;
si e(s) ⩽ 0 alors
    début
    choisir q ∈ f_P(s) ;
    message reçu (q) :=
    message transmis ;
    état (q) := actif ;
    f_P(s) → q
    fin
sinon
    message transmis → f_M(s)
fin ;
```

En admettant qu'initialement $e(s)$ soit nul et que les files $f_M(s)$ et $f_P(s)$ soient vides, le lecteur pourra établir la propriété suivante :

Après un nombre quelconque d'opérations $P_M(s)$ et $V_M(s)$:

Si $e(s) < 0$, la file $f_P(s)$ contient $|\, e(s) \,|$ processus, la file $f_M(s)$ est vide.
Si $e(s) \geqslant 0$, la file $f_M(s)$ contient $e(s)$ messages, la file $f_P(s)$ est vide.

On peut en tirer deux conséquences :

— il est possible de remplacer les deux files $f_P(s)$ et $f_M(s)$ par une file unique $f(s)$, dont les éléments sont des pointeurs sur des processus (si $e(s) < 0$) ou des messages (si $e(s) \geqslant 0$),

— si on initialise un sémaphore avec message à une valeur $n > 0$, on doit aussi placer n messages dans sa file $f(s)$.

Les problèmes d'allocation de mémoire pour les messages semblent restreindre ces primitives aux cas où les messages transmis ont un format standard ; il est en effet difficile, à l'intérieur de primitives qui doivent rester indivisibles, de mettre en œuvre des mécanismes très complexes d'allocation de zones de mémoire de longueur variable. D'autre part, l'opération V_M ne peut s'exécuter correctement quand la file f_M associée est pleine : on doit prévoir une réaction spéciale pour ce cas. L'exercice 10 donne un exemple d'utilisation des sémaphores avec messages.

2.532 Communication entre processus dans le système MU5

Le système MU5 [Morris, 69] est réalisé à l'Université de Manchester sur une

machine prototype. Dans ce système, tout processus est représenté par une machine logique comportant un processeur et une mémoire principale.

Une mémoire secondaire unique contient des informations communes, constituées en ensemble de segments. La mémoire principale d'une machine logique peut être mise en communication avec cette mémoire secondaire, permettant au processus correspondant d'accéder à un segment.

La communication entre machines logiques se fait par l'intermédiaire de canaux : toute machine logique possède 16 canaux, associés à 16 niveaux d'interruptions sur le processeur logique. Un nombre quelconque de messages peuvent être envoyés sur chaque canal; ils sont rangés dans une file. Chaque message comprend :

— l'identité du processus expéditeur,
— une zone spécifiant une demande de réponse sur un canal de l'expéditeur,
— un numéro de segment (éventuellement),
— une zone de message de 120 caractères,
— un lien de chaînage vers le message suivant sur le même canal.

Les messages courts (jusqu'à 120 caractères) sont passés directement dans la zone de message; les messages longs sont placés dans un segment dont le numéro est transmis. L'arrivée d'un message sur un canal provoque une interruption sur le niveau correspondant à ce canal.

Les processus disposent de certaines possibilités de filtrage sur les messages qu'ils reçoivent. Un processus peut en particulier :

— inhiber un niveau d'interruption correspondant à un canal; il devient alors sourd aux messages arrivant sur ce canal,
— établir une sélection sur les messages reçus, en modifiant l'état de ses canaux.

Un canal peut se trouver dans trois états :

— « ouvert » : tout processus peut envoyer des messages sur le canal,
— « réservé » : un processus spécifié, et lui seul, peut envoyer des messages sur le canal,
— « fermé » : aucun processus ne peut envoyer de message sur le canal.

Le processus expéditeur peut demander une réponse sur un canal spécifié; cette réponse lui est automatiquement transmise quel que soit l'état du canal. Toute erreur ou anomalie d'émission est signalée sur un canal fixé une fois pour toutes. En dehors des primitives d'action sur l'état des canaux, il existe une primitive d'envoi de message, qui a comme paramètres :

— le processus destinataire,
— le canal d'émission,
— le canal de réponse (éventuellement),
— le texte du message ou le numéro de segment,
— et une primitive de réception de message, qui a comme paramètre le canal de réception.

La commande de réception fait entrer le message dans la mémoire principale du destinataire ; si c'est un message long, le segment contenant le message est mis en communication avec la mémoire principale.

Les erreurs dans la procédure de transmission sont détectées par le superviseur (envoi de message à un processus inexistant, sur un canal fermé, etc...).

En résumé, le dispositif de communication du système MU5 se caractérise par une prise en charge complète de la communication entre processus par le système d'exploitation ; un mécanisme de synchronisation est associé à la communication, permettant l'attente de messages par le destinataire et le réveil de l'expéditeur ; les échanges sont personnalisés, c'est-à-dire que l'identité de l'expéditeur et du destinataire est toujours explicitement indiquée ; tout échange est protégé, grâce à la notion de canal, contre l'intervention de processus ne participant pas à cet échange ; enfin, les zones contenant les messages sont gérées par le système.

Un mécanisme voisin programmé est décrit dans [Brinch Hansen, 70].

2.6 IMPLANTATION DES PRIMITIVES DE SYNCHRONISATION

Ce paragraphe est consacré aux problèmes de programmation des primitives de synchronisation. Lorsqu'un processus se bloque en attente d'une ressource, il est inutile de lui garder l'unité centrale ; il en résulte qu'une primitive de synchronisation fait habituellement appel à l'allocation de processeur ; l'implantation des primitives et l'allocation de processeur aux processus sont ainsi étroitement liées.

Nous nous plaçons dans le cas où les primitives de synchronisation sont réalisées par des procédures du système exécutées, sous contrôle, par les processus qui en ont besoin. L'implantation des primitives doit, dans ce cas, résoudre trois problèmes :

— assurer une exclusion mutuelle aux variables d'état du système auxquelles accèdent les primitives. Les demandes d'accès viennent des processeurs câblés du système ou des divers niveaux d'interruption d'un même processeur câblé.

— interdire l'accès aux procédures et aux variables des primitives en dehors de tout appel normal.

— permettre le changement des mots d'état du processeur pour attribuer celui-ci au processus désigné par l'allocateur.

2.61 EXCLUSION MUTUELLE DANS LES PRIMITIVES

Nous admettrons que tous les processus ont accès, par l'intermédiaire de procédures du système, à des variables communes utilisées à des fins de synchronisation.

L'implantation des primitives dépend étroitement de la structure (mono- ou multiprocesseur) du calculateur et de la nature des instructions disponibles.

Nous allons donc étudier une suite de cas de complexité croissante :

a) On ne dispose que de la gamme d'instructions classique : les transferts entre mémoire et registres sont les seules opérations indivisibles utilisées. Celles-ci réalisent en effet le mécanisme élémentaire d'exclusion mutuelle mentionné en 2.32. Dans ce cas, il est théoriquement possible de bâtir des mécanismes de synchronisation en utilisant des variables booléennes (exercice 3). Cependant, cette solution est très limitative : un processus bloqué n'est pas interrompu, mais exécute une boucle d'attente, immobilisant donc inutilement son processeur et freinant l'accès à la mémoire. Elle n'est donc valable que pour un système multiprocesseur.

b) Le calculateur possède une seule unité centrale ; une séquence d'instructions peut être rendue indivisible en masquant l'unité centrale contre toute interruption. On utilise alors, pour passer dans l'état « interruptions masquées », une instruction d'appel au superviseur (*SVC* de l'IBM 360, *CAL* du CII 10070). A l'intérieur des séquences masquées, on teste et on modifie les variables communes et on change éventuellement les processus de file d'attente. La primitive se termine par l'activation de l'un des processus en attente seulement de l'unité centrale.

c) Le calculateur possède plusieurs unités centrales. Pour se ramener au cas précédent, il suffit de disposer d'un mécanisme garantissant qu'à un instant donné un processeur au plus peut exécuter une primitive de synchronisation. Pour assurer cette exclusion mutuelle, on préfère à l'utilisation de booléens l'emploi de l'instruction *TAS* qui garantit une meilleure efficacité.

Mais de toute façon, quand un processus veut exécuter une primitive alors qu'un autre, sur un processeur différent, en exécute déjà une, il faut le maintenir dans une boucle d'attente.

La programmation de l'exclusion mutuelle par l'instruction *TAS* a été décrite en 2.32. Avant l'entrée dans la section critique protégée par *TAS*, on masque le processeur contre toutes les interruptions de façon à ne pas interrompre un processus qui exécute une primitive. On procède ensuite comme en *b*). Les interruptions sont démasquées à la sortie de la section critique.

2.62 GESTION DES PROCESSUS

Les processus sont gérés selon le schéma général suivant :

— quand un processus passe dans l'état bloqué, on lui retire le processeur sur lequel il s'exécutait,

— le déblocage d'un processus, c'est-à-dire son passage à l'état actif, le rend de nouveau candidat à l'occupation d'un processeur. Si un processeur peut lui être alloué, le processus devient **élu** ; sinon, il doit attendre, et il est dit alors dans l'état **prêt**. Les états élu et prêt caractérisent donc la situation d'un processus actif vis-à-vis de la ressource processeur.

Le moniteur dispose d'une table des processus, chaque entrée contenant des informations propres à un processus. De plus les processus sont généralement chaînés dans diverses files d'attente, correspondant aux diverses ressources manquantes : processus en attente de mémoire, en attente de page, bloqués derrière un sémaphore, processus prêts, etc... Le blocage et le déblocage d'un processus se traduisent donc par des changements de file.

L'insuffisance de ressources physiques (registres généraux des processeurs par exemple) impose de disposer de zones de mémoire pour la sauvegarde de certaines fractions des vecteurs d'état des processus bloqués. Ainsi, à l'élection d'un processus (allocation d'un processeur), les registres propres à ce processeur sont chargés à partir des informations conservées dans la zone de sauvegarde ; inversement, lorsqu'un processus devient bloqué, les registres du processeur sur lequel il s'exécutait sont rangés.

La modification des registres internes du processeur ne peut être effectuée qu'avec des droits particuliers (mode maître par exemple).

Exemple. Pour le CII 10070, les registres à charger sont :

— les 16 registres généraux,
— la mémoire topographique,
— le double mot d'état de programme *(PSD)*.

Le *PSD* est chargé en dernier, au moyen d'une instruction spéciale *LPSD*. Cette instruction alloue l'unité centrale à un processus, dont l'exécution commence à l'adresse fixée par le *PSD*.

2.63 PROTECTION DES PRIMITIVES

La protection des primitives est en général assurée par deux mesures complémentaires.

— L'appel d'une procédure de primitive se fait par un point d'entrée unique (guichet, voir 5.1), nécessitant l'emploi d'une instruction d'appel au superviseur.

— Tout autre accès (lecture, écriture ou exécution) à la procédure d'une primitive ou à ses données est interdit. Cette interdiction est obtenue par l'emploi de clés et de verrous d'accès ou encore par l'existence d'espaces d'adressage disjoints (par exemple, dans ESOPE, les processus désignent, en mode « avec topographie » des adresses virtuelles et les primitives, en mode « sans topographie » des adresses physiques).

Dans certains cas, les primitives du système sont mises à la disposition des utilisateurs. Le contrôle de leur emploi est fait au guichet.

2.64 EXEMPLES

Dans les exemples qui suivent, nous présentons successivement :

— une implantation simplifiée de sémaphores sur un IBM 360 monoprocesseur ; l'absence de tout dispositif de protection rend cet exemple aisément compréhensible (et réalisable au cours d'une séance de travaux pratiques).

— la gestion des processus prêts dans le système SIRIS 8, qui montre comment on peut tirer parti d'un système câblé d'interruptions pour l'allocation de processeur.

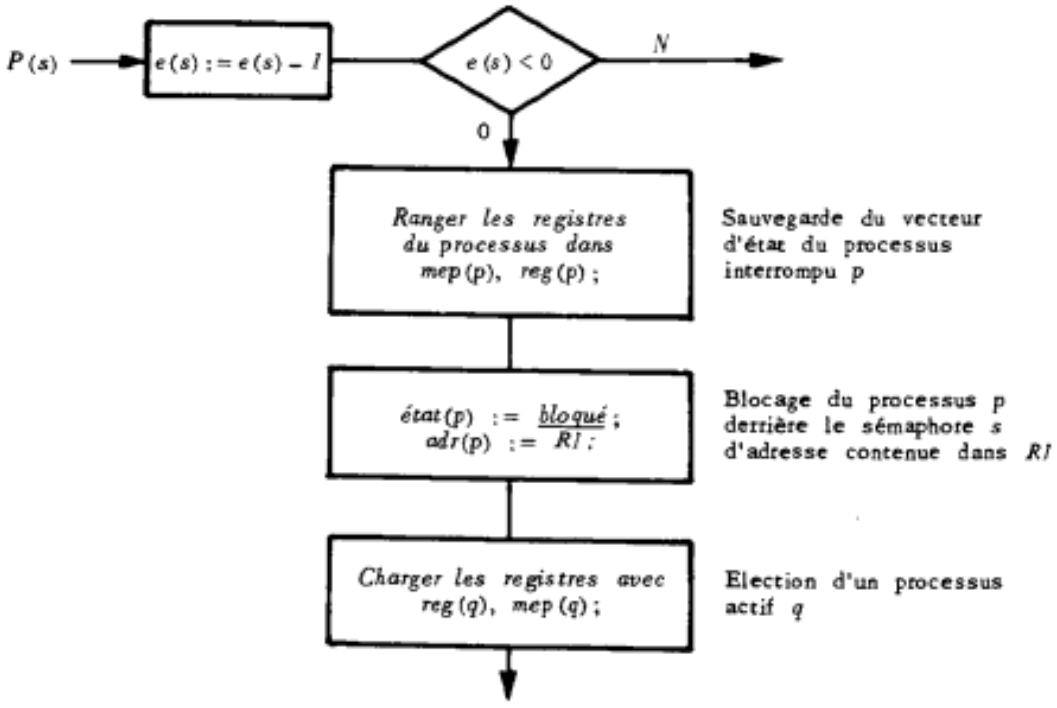
— l'implantation des files d'attente utilisées dans le système BURROUGHS B6500 pour la gestion des événements.

Exemple 1 : implantation de sémaphores [Wirth, 69].

Les processus sont chaînés en une seule file circulaire. Chaque vecteur d'état de processus, *vepro*, contient les champs suivants :

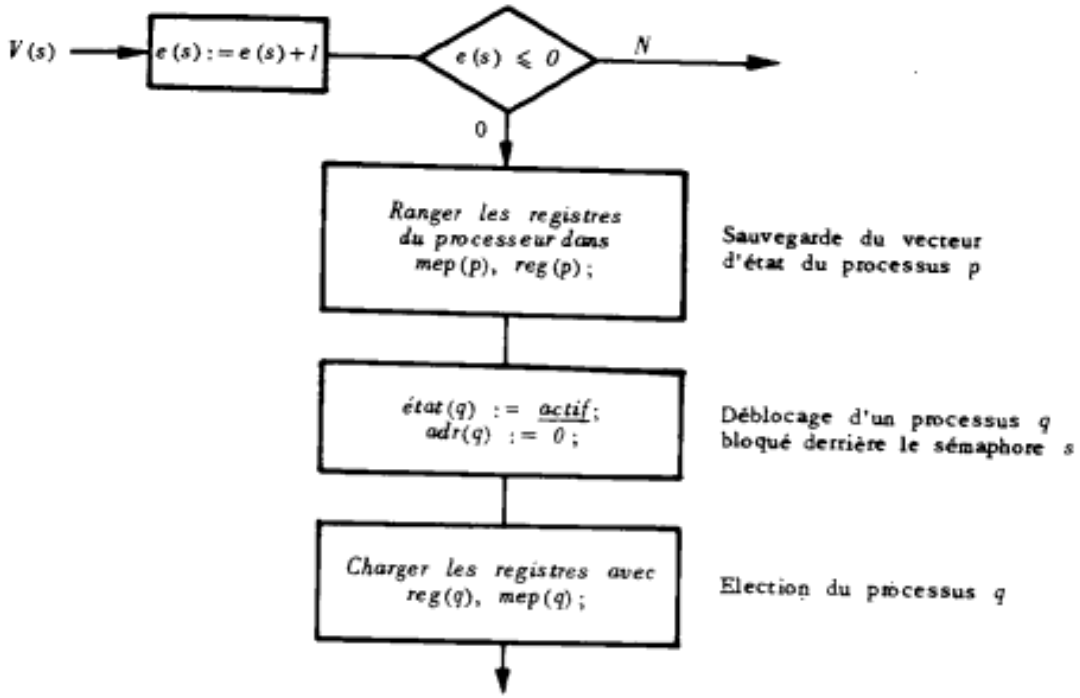
— *état* état du processus,
— *mep* mot d'état du programme,
— *reg* registres généraux,
— *adr* adresse de sémaphore,
— *suc* pointeur vers le vecteur d'état du processus suivant dans la file.

Lors de l'appel d'une primitive par l'intermédiaire d'un appel au superviseur (instruction *SVC*), le registre *R1* doit contenir l'adresse du sémaphore (*SVC 10* sert pour l'appel de *P*, *SVC 11* pour celui de *V*). L'instruction *SVC* déclenche une interruption qui permet à l'unité centrale de passer en mode maître. Les interruptions sont masquées et les actions ci-après sont entreprises :



Commentaires :

— On suppose qu'il existe toujours un processus actif au moins.
— Les *vepro* sont chaînés circulairement ; la recherche d'un processus actif consiste à balayer la file jusqu'à ce que l'on trouve un processus avec *état* = *actif*. La recherche d'un processus en attente de *s* impose en plus de consulter la valeur de *adr*.



— Quand une primitive *V* conduit au déblocage d'un processus, on arrête le processus élu et on alloue l'unité centrale au processus débloqué. Cela peut conduire à des modifications superflues de l'état du processeur.

La création et la destruction de sémaphores sont laissées à la responsabilité du programmeur qui les déclare comme entiers.

La création et la destruction de processus sont réalisées par deux appels au superviseur, *OPEN* et *CLOSE* :

— A l'appel de *OPEN*, on passe dans le registre *R2* l'adresse du nouveau *vepro* ; l'adresse de début du nouveau processus est aussi accessible. *OPEN* chaîne le *vepro* pointé par *R2* dans la file des processus, range le mot d'état de programme et les registres du processus créateur, et alloue l'unité centrale au processus créé.

— *CLOSE*, primitive sans paramètre, supprime le processus actif de la file circulaire ; l'unité centrale est alors allouée à un autre processus actif.

Exemple 2 : gestion des processus prêts dans le système SIRIS 8 [Boulle, 70].

On désire, dans ce système multiprocesseur fonctionnant sur l'IRIS 80 de la CII, utiliser le plus possible le système d'interruption pour la gestion des processus. Donnons d'abord un aperçu de ce système :

— Les différents processeurs sont identiques. Tout processus peut s'exécuter sur l'un quelconque des processeurs (et éventuellement en changer au cours de son exécution).

— Le système d'interruption est unique et fonctionne comme suit :

  — il existe plusieurs niveaux d'interruptions ; une priorité est associée par câblage à chaque niveau,

  — à la réception d'un signal, externe ou programmé, un niveau est activé, pour demander l'exécution du programme qui lui est associé à cet instant,

— quand un niveau n passe dans l'état actif, sa priorité est comparée automatiquement à la priorité des niveaux au cours de traitement sur les différents processeurs. Le programme associé à n ne peut s'exécuter que s'il y a un niveau moins prioritaire en cours de traitement ; dans ce cas, on interrompt le programme de traitement du niveau le moins prioritaire et on exécute le programme associé au niveau n.

Dans SIRIS 8, à chaque processus correspond une priorité, et donc un niveau d'interruption. On associe à chaque niveau une file d'attente de processus et on dispose en outre d'une pile dans laquelle on range les vecteurs d'état des processus interrompus.

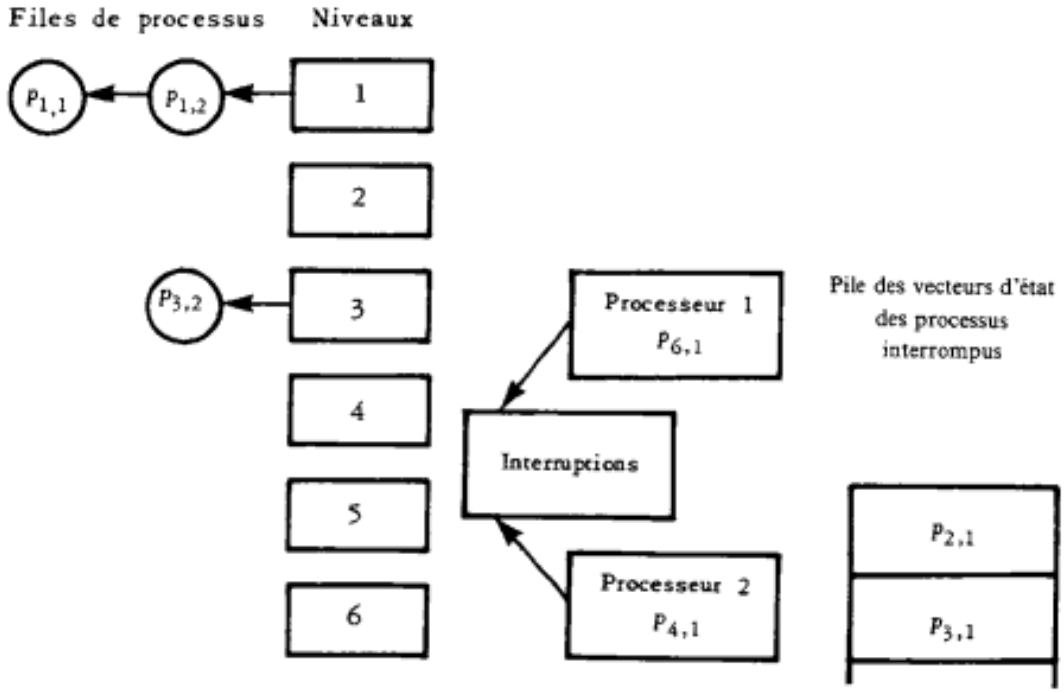


Figure 1. Gestion des processus dans le système SIRIS 8.

Quand un processus passe dans l'état prêt, il entre dans la file d'attente de son niveau qui est déclenché par programme. L'unité centrale est alors allouée par le système d'interruptions. Dans le cas où un processus est interrompu, son vecteur d'état est sauvegardé dans la pile.

Dans l'exemple ci-dessus (le niveau 1 étant le moins prioritaire), les processus $p_{1,1}$, $p_{1,2}$ et $p_{3,2}$ sont en attente dans les files des niveaux 1 et 3 ; les processus $p_{3,1}$ et $p_{2,1}$ sont interrompus et les processus $p_{4,1}$ et $p_{6,1}$ sont élus.

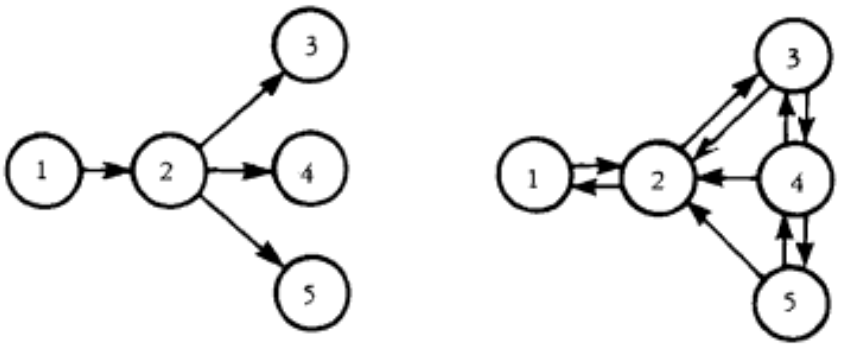
Si un processus associé au niveau 2 passe dans l'état prêt, il rejoint simplement la file de ce niveau ; par contre, si un processus $p_{5,1}$ associé au niveau 5 devient prêt, le processus $p_{4,1}$ est interrompu, son vecteur d'état est rangé dans la pile ; le processus $p_{5,1}$ peut alors s'exécuter.

Quand un processus se bloque intrinsèquement, son vecteur d'état est rangé dans une zone de mémoire propre ; si la file du niveau d'interruption associé est vide, le niveau est désactivé. Puis le système d'interruption active le processus p dont le vecteur d'état se trouve au sommet de la pile : c'est le dernier interrompu, donc en général

le plus prioritaire. Cependant, il est possible qu'un processus prêt q soit en attente dans la file d'un niveau plus prioritaire ; dans ce cas, le processus activé p est interrompu immédiatement au profit du processus q.

Exemple 3 : gestion des processus dans le système BURROUGHS B6500 [Cleary, 69].

Gestion des processus : dans le B6500, à chaque processus est associée une pile. Cette pile contient le vecteur d'état du processus et tous les liens de chaînage utilisés par le système. Le B6500 est étudiée plus en détail en 3.3. On entretient à chaque création ou destruction de processus une arborescence généalogique (Fig. 2).



a) Généalogie b) Liens de chaînage utilisés

Figure 2. Arborescence de processus dans le B6500.

A chaque destruction de processus, on détruit toute sa descendance.

Synchronisation : le système utilise des événements booléens, et gère un ensemble de files d'attente :

— une file des processus prêts,
— une file par événement.

Quand un processus prêt est élu, on remplace les liens de chaînage de la file des processus prêts par le numéro du processeur sur lequel il va s'exécuter ; les processus de la file prêt sont rangés par ordre de priorité (Fig. 3).

Les files de processus en attente d'événement ont une structure analogue : un champ supplémentaire A (Fig. 4) indique si l'événement s'est produit ou non.

Exclusion mutuelle : on dispose des primitives permettant de verrouiller ou de déverrouiller une ressource.

Interruptions programmées : les mécanismes étudiés jusqu'à présent permettent de programmer explicitement la coopération entre processus ; un processus actif ne peut se bloquer qu'à des stades de son exécution fixés une fois pour toutes. Au contraire, avec les interruptions programmées, un processus peut être interrompu à n'importe quel instant et obligé d'exécuter une procédure fixée au départ. On obtient donc une certaine souplesse de programmation et, en contrepartie, les mêmes problèmes de mise au point que dans les systèmes à interruptions câblées.

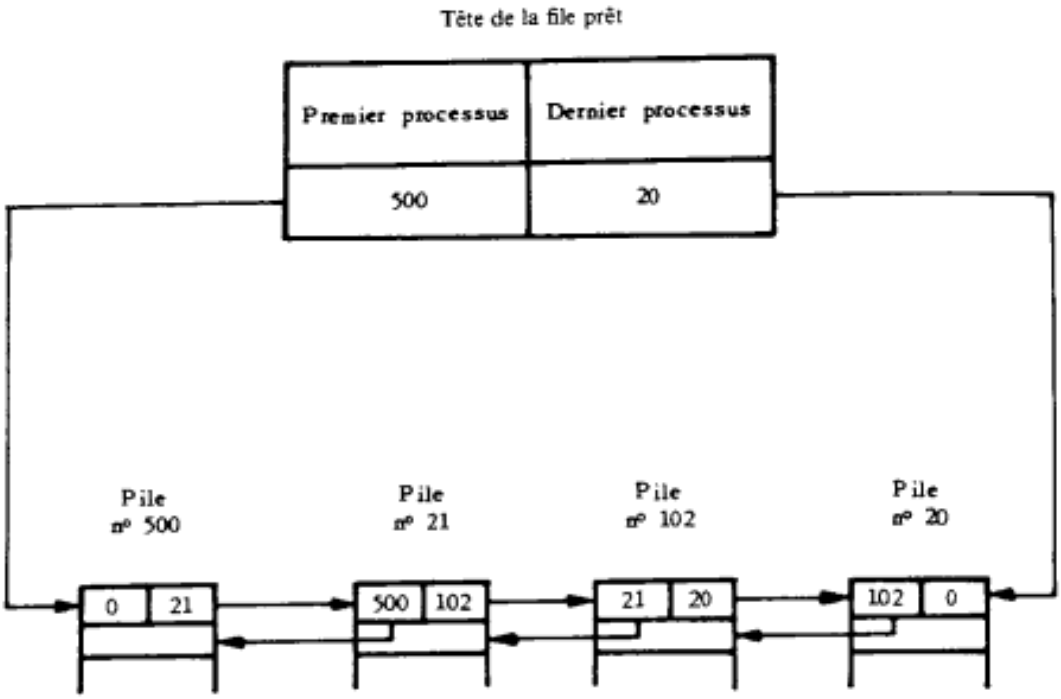


Figure 3. File de processus dans le B6500.

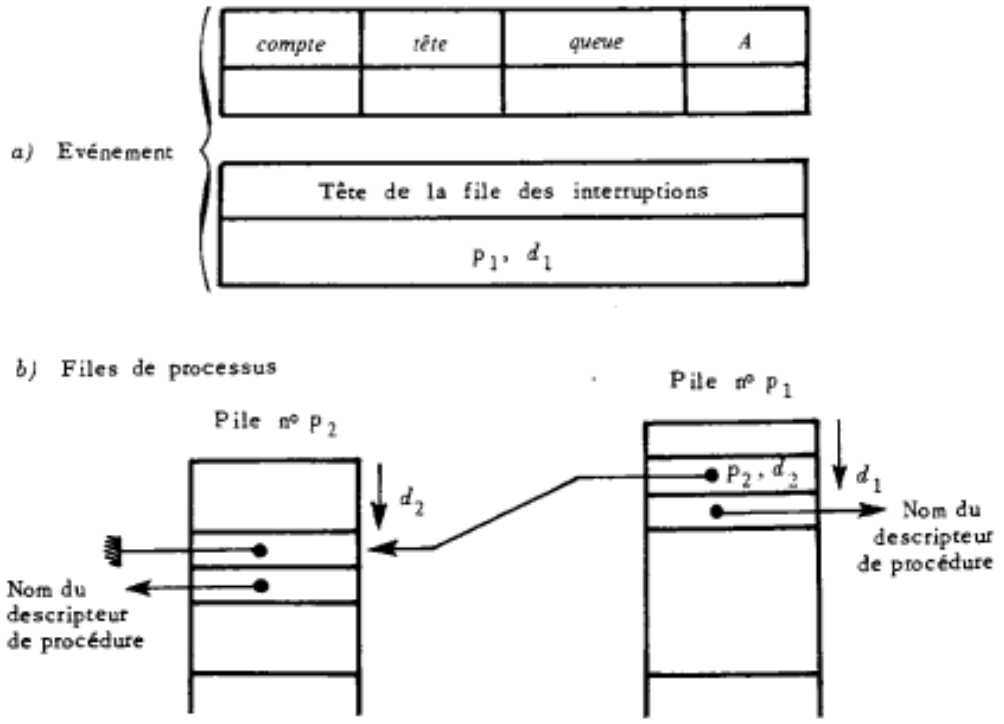


Figure 4. Gestion des interruptions programmées dans le B6500.

Les interruptions programmées de processus sont déclarées comme suit, en utilisant les événements :

event e ;
interrupt i : on e, instruction ;

On peut armer ou désarmer une interruption : si l'interruption *i* est armée, le processus exécute l'instruction suivant le *on* à chaque déclenchement de l'événement *e*, puis reprend le cours normal de son exécution.

Cela est réalisé comme suit : à un événement *e* sont associées deux files, celle des processus en attente de l'événement et celle des interruptions déclenchées par *e*. A un événement correspond un double-mot, le premier mot étant l'en-tête de la file des processus en attente, le second l'en-tête de la file des interruptions. Cette seconde file contient un pointeur vers le double-mot suivant et un pointeur vers le descripteur de la procédure à exécuter. La longueur de la file des interruptions est contenue dans un champ particulier, *compte*.

Un processus peut se trouver dans un nombre quelconque de files d'interruptions programmées. Chaque double-mot de la pile peut être utilisé ; pour cela, les liens de chaînage comprennent un numéro de pile et un déplacement à l'intérieur de la pile (permettant de localiser le double-mot de gestion de l'interruption).

Au déclenchement d'un événement, on effectue les actions suivantes :

— chaîner dans la file des processus prêts, à un rang correspondant à leur priorité, les processus en attente de l'événement,

— parcourir la file des interruptions, interrompre tous les processus qui ont armé l'interruption et appeler la procédure à exécuter comme s'il s'agissait d'une interruption câblée.

Il peut se faire, dans un système multiprocesseur, que l'un des processus à interrompre soit élu au moment de l'interruption. Pour tenir compte de cette éventualité, on ramène tous les processus en un point observable en interrompant systématiquement l'ensemble des processeurs (sauf celui qui exécute la séquence de déclenchement de l'interruption). Les processeurs sont alloués en priorité aux processus dont l'interruption a modifié l'état.

A l'arrivée d'une interruption, câblée ou programmée, le vecteur d'état du processus interrompu est sauvegardé et restauré par l'intermédiaire du mécanisme unique d'appel de procédure : une interruption équivaut à l'appel forcé d'une procédure.

2.7 PROBLÈMES DE PROTECTION

2.71 LES PROBLÈMES

Dans la plupart des systèmes coexistent deux classes de processus :

— des processus dont l'algorithme est fixé une fois pour toutes, et qui peuvent dans une certaine mesure être considérés comme fiables (il s'agit essentiellement des processus du moniteur),

— des processus dont on ne connaît absolument rien (processus des usagers) et qui doivent *a priori* être considérés comme suspects.

De toute façon, en l'absence de preuve rigoureuse de validité, on ne peut affirmer qu'un processus est définitivement exempt d'erreurs de programmation ; il est donc impératif de limiter les conséquences d'erreurs éventuelles.

Une mauvaise synchronisation est due, en général, à l'une des deux causes suivantes :

a) Les données servant à la gestion des processus et des primitives sont modifiées par un processus, sans utiliser les primitives prévues. La prévention de ce cas relève du problème plus général du pouvoir d'un processus d'accéder à la mémoire (voir Chap. 5).

b) Les primitives, bien qu'appelées correctement, sont mal utilisées par des processus. Il s'agit alors d'une erreur de programmation.

Donnons quelques exemples de fautes produites par un emploi incorrect des sémaphores :

— Immobilisation définitive d'une ressource par un processus : si *mutex* est un sémaphore utilisé pour assurer l'exclusion mutuelle à des tables globales du système et si un processus exécute un *P(mutex)* sans le faire suivre au bout d'un temps fini par un *V(mutex)*, alors les tables protégées ne sont plus accessibles.

— Interblocage : soit *a* et *b* deux sémaphores d'exclusion mutuelle. Si deux processus *p* et *q* peuvent exécuter respectivement *(P(a);P(b))* et *(P(b); P(a))*, il y a risque de blocage, chacun des deux processus devant attendre que l'autre libère son sémaphore avant de continuer (voir 4.7).

— Mauvaise utilisation d'un sémaphore privé : supposons que dans le système, on fasse appel à un processus d'entrée-sortie *e* par l'intermédiaire d'une primitive *V* sur le sémaphore privé *s* de ce processus. Supposons en outre qu'un processus *q* quelconque exécute par erreur la primitive *P(s)*. Alors le processus *q* rejoint la file d'attente de *s*. Lors d'une demande ultérieure d'entrée-sortie, le processus *q* sera activé au lieu du processus *e*.

— Interférences parasites : d'une façon plus générale, il faut garantir l'indépendance effective des processus logiquement indépendants ; en d'autres termes, une erreur commise par un processus ne doit pas perturber ou détruire des processus qui ne coopèrent pas avec lui.

2.72 QUELQUES REMÈDES

Pour éviter ces ennuis nous proposons la politique suivante, inspirée de [Brinch Hansen, 72].

1) Réservant à la gestion du parallélisme les primitives *P* et *V*, nous introduisons une instruction spéciale pour assurer l'exclusion mutuelle :

avec a faire instruction ;

dans laquelle *a* désigne une donnée partagée.

On peut traduire aisément l'instruction ci-dessus, en utilisant P et V :

P(mutex-a) ;
instruction ;
V(mutex-a) ;

mutex-a désigne un sémaphore associé à la donnée *a*.

2) Dans un système à accès multiple, il faut préserver l'indépendance des différentes machines virtuelles de chaque utilisateur. Nous définissons donc une partition de l'ensemble des processus en sous-ensembles ou **usagers** : un usager correspond soit au moniteur, soit à un utilisateur du système.

L'emploi des primitives de synchronisation se fait alors comme suit :

a) Chaque usager peut déclarer ses sémaphores privés et utiliser certaines de ses variables privées pour réaliser des exclusions mutuelles. Ce faisant, il ne risque de bloquer que sa propre machine virtuelle, ce qui ne saurait mettre en cause le fonctionnement global du système.

b) Un usager peut en autoriser d'autres à exécuter exclusivement des primitives V sur certains de ses sémaphores ; cette extension à l'usager de la notion de sémaphore privé permet une synchronisation entre usagers (demande de service au moniteur, par exemple).

Exemple. Le processus du moniteur chargé des entrées-sorties sur un périphérique est bloqué au repos derrière un sémaphore *ses*, accessible à tous les usagers. Lorsqu'un processus fait une demande d'entrée-sortie, il dépose un message dans un tampon, puis exécute *V(ses)*. Le message comprend les paramètres de l'entrée-sortie à exécuter et un nom de sémaphore sur lequel il faut exécuter en fin d'entrée-sortie une primitive V. Ce sémaphore a dû être déclaré accessible au moniteur. Pratiquement, les noms de sémaphores qui figurent dans les primitives sont qualifiés par le nom de l'usager lorsqu'ils appartiennent à un usager autre que celui qui exécute la primitive.

Ce fonctionnement peut être illustré par le schéma suivant :

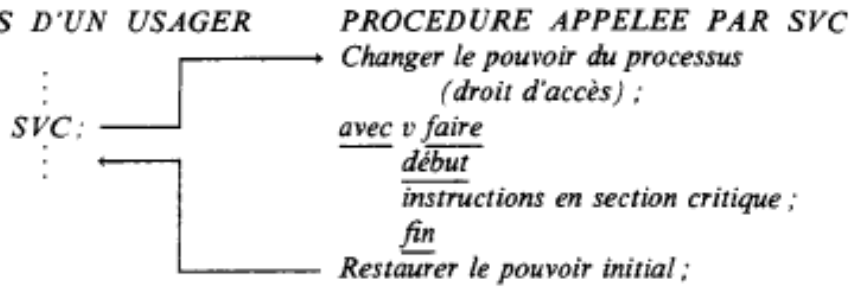
PROCESSUS DE L'USAGER u	*PROCESSUS DU MONITEUR*
Préparer le message d'entrée-sortie ;	*A : P(ses) ;*
V(monit.ses) ;	*Décoder le message d'entrée-sortie ;*
⋮	*Exécuter l'entrée-sortie ;*
P(sem) ;	*V(u.sem) ;*
	aller à A ;

On peut associer l'autorisation d'exécution de la primitive V soit à certains sémaphores, soit à certains usagers ou processus.

3) En ce qui concerne les exclusions mutuelles entre processus d'usagers différents, certaines précautions sont à prendre :

— Imposer à un processus d'entrer dans une section critique globale par une séquence déterminée d'instructions. On utilise pour cela, dans la pratique, une instruction d'appel au superviseur *(SVC)* suivant le schéma ci-après (voir 5.1 et 5.2).

Exemple



— S'assurer de la fiabilité des instructions contenues dans une section critique.

— Ne jamais détruire un processus à l'intérieur d'une section critique globale. On peut associer pour cela à chaque processus un compteur de sections critiques globales, compteur initialisé à *0*, augmenté de *1* à chaque entrée de *avec*, diminué de 1 à chaque sortie ; un processus ne peut être détruit que lorsque le compteur a la valeur *0*.

Quand un processus se trouve dans une section critique globale, on n'a pas intérêt à lui retirer le processeur au profit d'un autre processus, même plus prioritaire. En effet, le nouveau processus actif se bloquera si son programme comporte également une entrée dans la même section critique. Lors de l'allocation d'unité centrale, on peut tirer parti du compteur de sections critiques pour évaluer la priorité réelle d'un processus.

Ces indications n'empêcheront pas un usager de programmer ses exclusions mutuelles à l'aide de P et V, au risque de bloquer indéfiniment sa machine virtuelle ; par contre, on garantit que l'usager en difficulté n'aura pas d'influence néfaste sur les autres.

2.8 EXEMPLE DE COOPÉRATION DE PROCESSUS

Le problème de la gestion d'une imprimante dans le système ESOPE a été présenté en 2.24, où le système d'entrée-sortie a été décomposé en processus. Il reste à traiter en détail le problème de la coopération de ces processus. Pour des raisons de présentation, ce traitement est reporté au paragraphe 7.5.

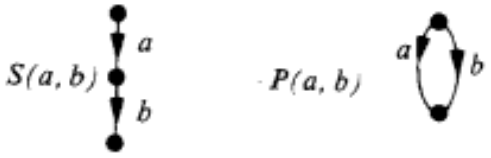
EXERCICES

1. [I]

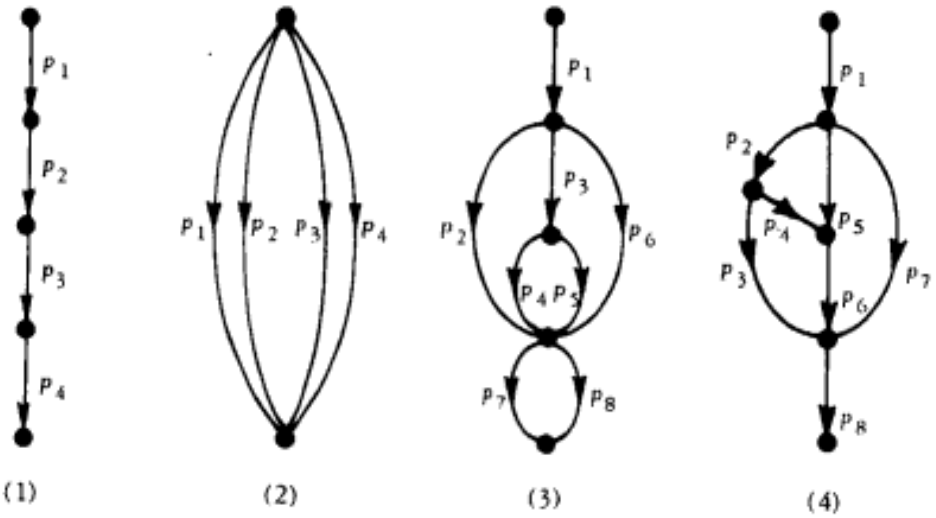
Un ensemble de processus coopérant à l'exécution d'une tâche commune peut être représenté par un graphe orienté dans lequel chaque arc représente l'évolution complète d'un processus. Le graphe ainsi obtenu est appelé graphe des processus pour la tâche

considérée. On introduit deux lois de composition $S(a,b)$ et $P(a,b)$ où a et b désignent des processus, telles que :

$S(a, b)$ représente l'exécution en série des processus a et b,
$P(a, b)$ représente leur exécution en parallèle.



En utilisant les seules fonctions S et P, donner, lorsque c'est possible, une description des graphes suivants :



(1) (2) (3) (4)

2. [1]

Dans le cas où les effets de bord n'imposent pas une évaluation séquentielle fixe, certaines sous-expressions d'une expression arithmétique peuvent être évaluées dans un ordre quelconque. On peut donc les évaluer en parallèle, si on dispose d'un nombre suffisant de processeurs.

Soit l'expression :

$$(a + b) * (c + d) - (e/f)$$

1) Donner la structure d'arbre correspondante.

2) En utilisant les conventions de l'exercice 1, donner un graphe des processus correspondant à une évaluation parallèle de cette expression. On cherchera à exploiter au mieux le parallélisme.

3) Donner une description de ce graphe en utilisant les seules fonctions S et P introduites dans l'exercice 1.

3. [3] *Programmation de l'exclusion mutuelle au moyen de variables communes* [Dijkstra, 67].

Le problème de l'exclusion mutuelle entre processus parallèles pour l'accès à une section critique a été posé en 2.31. Cet exercice a pour but de montrer les difficultés de la programmation de cette exclusion mutuelle lorsque les seules opérations indivi-

sibles dont on dispose sont l'affectation d'une valeur à une variable et le test de la valeur d'une variable (on exclut donc les opérations du type *TAS* et l'usage des sémaphores).

Le principe de la solution recherchée consiste à définir un ensemble de variables d'état, communes aux contextes des divers processus ; l'autorisation d'entrée en section critique sera définie par des tests sur ces variables, et l'attente éventuelle sera programmée comme une attente active, c'est-à-dire par répétition cyclique des tests.

I) On se limite initialement au cas de deux processus, notée p_1 et p_2.

1) On essaiera d'abord de construire une solution utilisant une variable booléenne unique c, égale à *vrai* si l'un des processus p_i se trouve dans sa section critique, à *faux* autrement.

Vérifier que l'exclusion mutuelle ne peut être programmée en utilisant cette seule variable.

2) On essaiera ensuite de construire une solution utilisant une variable commune unique t, avec la convention suivante :

$t = i$ si et seulement si le processus p_i est autorisé à s'engager dans sa section critique $(i = 1, 2)$.

— Ecrire le programme du processus p_i.

— Vérifier qu'on peut obtenir une solution répondant aux conditions *a*), *b*), *d*) du 2.31 mais non à la condition *c*).

3) Pour éviter la restriction précédente, on introduit maintenant une variable booléenne par processus ; soit $c[i]$ la variable attachée à p_i, avec la signification suivante :

$c[i]$ = *vrai* si le processus p_i est dans sa section critique ou demande à y entrer,

$c[i]$ = *faux* si le processus p_i est hors de sa section critique.

Le processus p_i a le pouvoir de lire et modifier $c[i]$, et de lire seulement $c[j]$ $(j \neq i)$.

— Ecrire le programme du processus p_i.

— Vérifier qu'on ne peut obtenir qu'une solution satisfaisant aux conditions *a*), *c*), *d*) ou *b*), *c*), *d*) du 2.31 mais non aux quatre conditions simultanément.

4) Une solution correcte peut être obtenue en combinant les solutions 2) et 3) ci-dessus, c'est-à-dire en complétant la solution 3) par l'introduction d'une variable supplémentaire t servant à régler les conflits à l'entrée de la section critique. Cette variable n'est modifiée qu'en fin de section critique. Le principe de la solution est le suivant : en cas de conflit (c'est-à-dire si $c[1]$ = *vrai et* $c[2]$ = *vrai*) les deux processus s'engagent dans une séquence d'attente où t garde une valeur constante. Le processus p_i tel que $i \neq t$ annule sa demande en faisant $c[i] :=$ *faux*, laissant donc l'autre processus entrer en section critique ; p_i attend ensuite dans une boucle que t soit remis à i avant de refaire sa demande d'entrée par $c[i] :=$ *vrai*.

— Ecrire le programme du processus p_i.

— Vérifier qu'on peut obtenir une solution satisfaisant aux quatre conditions du 2.31.

II) Généraliser la solution 4) au cas de n processus $p_1, \ldots, p_n$. On introduira comme précédemment un tableau booléen $c[1 : n]$ et une variable entière t avec les significations du 4) ; on remarquera toutefois que la variable t ne fonctionne plus en bascule si $n > 2$, et qu'on ne peut donc se contenter de la modifier en fin de section critique. On procédera comme suit : tout processus p_i devra avoir lui-même exécuté $t := i$ avant de signaler son intention d'entrer en section critique par $c[i] :=$ *vrai* et de consulter les $c[j]$, $j \neq i$. On devra s'assurer que la variable t, une fois égale à i, ne sera plus modifiée jusqu'à ce que p_i

sorte de sa section critique. On utilisera à cet effet un second tableau booléen $b[1 : n]$, avec :

$b[i] =$ *vrai* si { ou bien $t = i$; ou bien le processus p_i demande à faire $t := i$

$b[i] =$ *faux* si $t \neq i$ et si le processus p_i est hors de sa section critique.

On s'assurera que p_i ne peut exécuter $t := i$ que si $b[i] =$ *faux*.

4. [1]

Programmer, en utilisant les sémaphores, les deux cas de synchronisation décrits ci-après en PL/1 (cf. 2.431).

Cas 1 (condition *et*)

Processus *a*	Processus *b*	Processus *c*
wait (e1,e2) (2)	*completion (e1) = '1'B*	*completion (e2) = '1'B*

Cas 2 (condition *ou*)

Processus *a*	Processus *b*	Processus *c*
wait (e1,e2) (1)	*completion (e1) = '1'B*	*completion (e2) = '1'B*

5. [2] *Description d'un système d'interruption simple* [Denning, 71].

On considère un système d'interruption utilisant deux bascules : un masque m et une trappe t. L'interruption est masquée si $m = 0$ et autorisée si $m = 1$. L'arrivée d'un signal d'interruption se manifeste par une tentative de faire $t := 1$. Si l'interruption est démasquée, t passe à 1 immédiatement ; sinon, t passe à 1 au moment du démasquage. La mise de t à 1 entraîne l'éveil d'un processus cyclique de traitement d'interruption.

Décrire, à l'aide de sémaphores, la logique de ce dispositif câblé et du processus cyclique de traitement.

6.[3] *Problème des « lecteurs » et des « rédacteurs »* [Courtois, 71].

Le modèle des lecteurs et des rédacteurs schématise une situation rencontrée dans la gestion des fichiers partageables. Dans ce modèle, on considère des processus parallèles (n au plus) divisés en deux classes : les lecteurs et les rédacteurs. Ces processus peuvent se partager une ressource unique, le fichier. Ce fichier peut être lu simultanément par plusieurs lecteurs, tandis que les rédacteurs doivent y avoir un accès exclusif (un seul rédacteur peut y écrire et aucun lecteur ne peut lire pendant ce temps).

On note comme suit le programme des lecteurs et des rédacteurs :

LECTEURS	*REDACTEURS*
demande de lecture ;	*demande d'écriture ;*
lecture ;	*écriture ;*
fin de lecture ;	*fin d'écriture ;*

Les procédures *demande d'écriture* et *fin d'écriture* devront assurer l'exclusion mutuelle entre deux rédacteurs. Avec les procédures *demande de lecture* et *fin de lecture*, elles doivent assurer les règles de coopération entre la classe des lecteurs et la classe des rédacteurs.

Ecrire, avec des sémaphores, des opérations P et V, et des variables d'état le programme des lecteurs et des rédacteurs dans les quatre cas énumérés ci-après.

Cas 1. Priorité des lecteurs sur les rédacteurs, sans réquisition. Les lecteurs ont toujours priorité sur les rédacteurs sans réquisition ; le seul cas où un lecteur doive attendre est celui où un rédacteur occupe le fichier. Un rédacteur ne peut donc accéder au fichier que si aucun lecteur n'est en attente ou en cours de lecture. On autorise toute coalition de lecteurs pour occuper indéfiniment le fichier et en interdire l'accès aux rédacteurs.

a) Donner une solution en utilisant des variables d'état :

— le nombre *nl* de lecteurs occupant le fichier,
— un témoin *e* d'allocation du fichier à un des rédacteurs,
— des sémaphores privés *spriv(i)*, $i = 1, 2, \ldots, n$,
— un sémaphore d'exclusion mutuelle, *mutex*,
— deux files d'attente *filelect* et *filered* qui servent à gérer les numéros des processus en attente.

Cette gestion utilise les procédures *ajouter* et *ôter* qui sont données ci-après.

```
sémaphore tableau spriv [1 : n] = 0 [1 : n] ; commentaire il y a n processus ;
sémaphore mutex = 1 ;
entier nl = 0 ; booléen e = faux ;
structure doublet (entier compte file liste) filelect, filered ;
procédure ajouter (j, x) ; doublet x ; entier j ;
    début
    compte de x := compte de x + 1 ;
    mettre j dans la liste de x
    fin ;
procédure ôter (j, x) ; doublet x ; entier j ;
    début
    compte de x := compte de x − 1 ;
    j := élément ôté de liste de x
    fin ;
```

b) Rechercher une solution sans utiliser de sémaphore privé (il existe une solution ne nécessitant que 3 sémaphores).

Cas 2. Priorité des lecteurs sur les rédacteurs si et seulement si un lecteur occupe déjà le fichier. Quand aucun lecteur ne lit, les lecteurs et les rédacteurs ont même priorité. Par contre dès qu'un lecteur lit, tous les autres lecteurs peuvent lire, quel que soit le nombre de rédacteurs en attente. Les lecteurs ont toujours le droit de se coaliser pour monopoliser le fichier.

Cas 3. Priorité égale pour les lecteurs et rédacteurs. Aucune catégorie n'a priorité sur l'autre. Si un lecteur utilise le fichier, tous les lecteurs nouveaux qui arrivent y accèdent jusqu'à l'arrivée d'un rédacteur. A partir de ce moment, tous les nouveaux arrivants attendent, sans distinction de catégorie. Si un rédacteur utilise le fichier, tous

les nouveaux arrivants attendent également. Quand le rédacteur a fini, il réveille le premier processus en attente dans l'ordre (ici inconnu) des files d'attente. Si plusieurs lecteurs se suivent dans la file, ils accèdent ensemble au fichier. L'attente infinie est impossible dans ces conditions.

La solution, dans ce cas, se déduit comme suit de la solution 1 : les opérations *demande de lecture* et *demande d'écriture* ne sont plus indépendantes mais doivent maintenant provoquer une mise en attente dans une file commune, sans distinction entre lecteurs et rédacteurs, dès qu'un rédacteur attend ou utilise le fichier.

Cas 4. Priorité des rédacteurs sur les lecteurs. On donne cette fois la priorité aux rédacteurs : dès qu'un rédacteur réclame l'accès au fichier, il doit l'obtenir le plus tôt possible sans réquisition, c'est-à-dire à la fin de l'exécution des processus occupant le fichier au moment de la demande. Donc tout lecteur arrivé après que le fichier ait été demandé par un rédacteur doit attendre, même si des lecteurs utilisent encore le fichier. On notera que les rédacteurs peuvent cette fois se coaliser pour interdire indéfiniment aux lecteurs l'accès au fichier.

On devra compléter la solution 1 par un mécanisme assurant que l'arrivée du premier rédacteur pendant une lecture bloque l'accès au fichier pour les lecteurs jusqu'à ce que ce rédacteur ait fini d'écrire, ainsi que tous les rédacteurs arrivés après lui et ayant trouvé le fichier occupé par un rédacteur. On utilisera un mécanisme symétrique de celui du cas 1 pour l'ouverture du fichier aux lecteurs par le dernier rédacteur présent. On désignera par *nr* le nombre total de rédacteurs en attente ou en cours d'écriture.

7. [3] *Problèmes des feux de circulation.*

La circulation au carrefour de deux voies est réglée par des feux verts ou rouges. Quand le feu est vert pour une voie, les voitures qui y circulent peuvent traverser le carrefour ; quand le feu est rouge, elles doivent attendre (on admet que les voitures de chaque voie traversent le carrefour en ligne droite). On impose les conditions suivantes :

a) toute voiture se présentant au carrefour doit le franchir au bout d'un temps fini,

b) les feux de chaque voie passent alternativement du vert au rouge, chaque couleur étant maintenue pendant un temps fini,

c) à un instant donné, le carrefour ne doit contenir que des voitures d'une même voie.

Les arrivées sur les deux voies sont distribuées de façon quelconque.

Le fonctionnement de ce système est représenté par un ensemble de processus parallèles.

Un processus *changement* gère la commande des feux, et un processus particulier est associé à chaque voiture. La traversée du carrefour par une voiture de la voie i ($i = 1, 2$) correspond à l'exécution d'une procédure $traversée_i$ par le processus associé à cette voiture. On demande d'écrire le programme du processus *changement* et des procédures $traversée_1$ et $traversée_2$ dans les deux cas suivants :

Cas 1. Le carrefour peut contenir une voiture au plus à la fois.

Cas 2. Le carrefour peut contenir k voitures au plus à la fois.

Le fonctionnement correct des feux doit être maintenu quel que soit l'ordre d'arrivée des voitures ; la modification des feux par *changement* doit donc comporter les opérations suivantes :

— interdire aux nouvelles voitures arrivant sur la voie où le feu est vert de s'engager dans le carrefour (pour respecter la condition *b*)),

— attendre que les voitures engagées dans le carrefour en soient sorties avant d'ouvrir le passage sur l'autre voie (pour respecter la condition *c*)).

Dans le cas 1, on a un problème d'exclusion mutuelle ; dans le cas 2, on pourra remarquer que le problème est analogue à celui des lecteurs et rédacteurs (exercice 6, cas 1) et adopter une solution du même type.

8. [2] *Une extension des primitives P et V* [Vantilborgh. 72].

On définit sur un sémaphore s deux nouvelles opérations indivisibles $P(n, s)$ et $V(n, s)$, où n est un entier non négatif. Selon la notation habituelle, $e0(s)$ désigne la valeur initiale de s, $f(s)$ la file d'attente associée à s. On associe à chaque processus $p \in f(s)$ un entier non négatif noté $rang(p)$, inclus dans son vecteur d'état ; $rang(p)$ n'a pas de signification si p n'est pas bloqué.

L'effet de $P(n, s)$ et $V(n, s)$ se décrit comme suit :

```
P(n, s) : si n ⩽ e(s) alors
              e(s) := e(s) − n
          sinon
              début
              commentaire on désigne par p le processus appelant ;
              introduire p dans f(s) ;
              état(p) := bloqué ;
              rang(p) := n
              fin
V(n, s) : début
          e(s) := e(s) + n ;
          tant que q ∈ f(s) et rang(q) ⩽ e(s) faire
              début
              extraire de f(s) un processus p tel que rang(p) ⩽ e(s) ;
              état(p) := actif ;
              e(s) := e(s) − rang(p)
              fin
          fin
```

Aucune hypothèse supplémentaire n'est faite sur l'algorithme de choix de p dans $f(s)$.

1) Programmer avec les seules primitives $P(n, s)$ et $V(n, s)$ le cas 1 de l'exercice 6 (problème des lecteurs et rédacteurs). On supposera que le nombre total de processus (lecteurs et rédacteurs) dans le système est au plus égal à une valeur fixe n.

2) En supposant que l'algorithme d'extraction des processus de la file d'attente de s commence par les processus dont le rang est le plus petit, exprimer $P(n, s)$ et (Vn, s) en utilisant les primitives $P(s)$ et $V(s)$.

3) Soit un ensemble de processus, partitionné en n classes de niveaux de priorité distincts ; ces classes sont numérotées $1, 2, \ldots, n\text{-}1, n$ dans l'ordre des priorités décroissantes. Tous les processus sont en compétition pour l'utilisation d'une même ressource critique ; en cas de conflit d'accès, la ressource est allouée au processus le plus prioritaire.

a) Programmer en utilisant les seules primitives $P(n, s)$ et $V(n, s)$ l'entrée et la sortie de la section critique pour les processus de la classe i. Pour assurer l'exclusion mutuelle d'accès à la ressource, on introduira un sémaphore initialisé à une valeur telle que la

somme des rangs de deux (ou plusieurs) processus soit toujours supérieure à sa valeur courante.

b) Donner une autre solution à ce problème en utilisant les primitives $P(s)$ et $V(s)$.

9. [2]

Démontrer directement, sans utiliser le théorème 1 (2.34), les propriétés 1 et 2 (2.52) du modèle du producteur et du consommateur. On considérera à cet effet les nombres *nprod* et *ncons* de cycles complets d'exécution du producteur et du consommateur depuis l'instant initial et on établira les relations satisfaites par ces nombres dans les différentes phases d'exécution des deux processus.

10. [2] [Saal, 70].

On considère un ensemble de processus associés deux à deux dans une relation de producteur à consommateur. Les différents couples (p_i, c_i), $i = 1, \ldots, n$ sont indépendants, sauf en ce qui concerne l'allocation de leurs tampons d'échange t_i, qui se fait suivant la procédure ci-après :

— chaque tampon est constitué de cases de taille fixe, chaînées entre elles ; chaque case peut contenir un article (on appelle ainsi les objets produits et consommés),

— initialement, aucun tampon n'est constitué ; il existe une réserve contenant m cases,

— le producteur p_i puise une case dans la réserve, y place un article et chaîne la case en queue du tampon t_i,

— quand le consommateur c_i a prélevé un article dans t_i, il restitue la case correspondante à la réserve,

— si la réserve est vide, le producteur p_i doit attendre ; il est réveillé dès qu'une case devient disponible.

Une case étant repérée par une adresse, on dispose des procédures suivantes :

demander case(x) : extraire une case libre de la réserve ; affecter à x la valeur de l'adresse de la case,
restituer case(x) : restituer à la réserve la case d'adresse x,
chaîner(x, i) : chaîner la case d'adresse x à la queue du tampon t_i ;
extraire(x, i) : extraire la case de tête du tampon t_i ; affecter à x la valeur de l'adresse de cette case.

1) On suppose d'abord qu'il n'y a pas de limite (autre que celle imposée par la taille de la réserve) à la taille d'un tampon t_i. Ecrire le programme des processus t_i et c_i en utilisant les procédures ci-dessus, et les opérations P et V. On notera que les opérations comportant des manipulations de files de cases doivent se faire en exclusion mutuelle.

2) Ecrire le même programme en utilisant les sémaphores avec messages (voir 2.531), en transmettant comme message les adresses de cases. Montrer que l'on n'a plus besoin d'utiliser les procédures de manipulation de cases.

3) Comment modifier les programmes de 1) et 2) lorsqu'on impose maintenant une limite supérieure l au nombre de cases de chaque tampon t_i ? (une telle limitation aurait un but de sécurité, pour éviter l'épuisement de la réserve en cas d'incident dans un processus p_i ou c_i).

11. [3] *Sections critiques conditionnelles* [Hoare, 72b].

Rappelons (2.72) que l'instruction

avec v faire I

associe une instruction I à une variable partagée v ; les instructions faisant référence à la même variable v sont en exclusion mutuelle ;

v peut désigner une variable simple ou composée (tableau, structure).

Nous introduisons deux formes conditionnelles de cette instruction :

1) Forme 1 :

avec v *lorsque* b *faire* I

signifie que l'instruction I doit être exécutée en exclusion mutuelle à la variable v lorsque la condition b est vérifiée.

L'expression booléenne b ne contient que des constantes ou des éléments de la donnée v.

Pratiquement, l'instruction est interprétée comme suit par un processus p :

a) le processus p entre en section critique et évalue b ;

b) si b = *vrai*, p exécute I et sort de la section critique ;

c) si b = *faux*, p sort de la section critique et se bloque ; il sera réactivé lorsqu'un autre processus quittera la section critique et reprendra son exécution en *a*).

2) Forme 2 :

avec v *faire* I *et attendre* b

Le processus p exécute I de façon inconditionnelle, puis se bloque jusqu'à ce que la condition b ait la valeur *vrai*.

1) Programmer à l'aide des sections critiques conditionnelles le problème des philosophes aux spaghetti (voir 2.432).

2) Programmer à l'aide des sections critiques conditionnelles le cas 1 de l'exercice 6 (problème des lecteurs et rédacteurs).

3) Programmer l'implantation des sections critiques conditionnelles en utilisant les sémaphores et les primitives $P(s)$ et $V(s)$.

12.[2] *Une primitive P généralisée* [Patil, 71].

On définit sur un ensemble de sémaphores $s_1, s_2, \ldots, s_k$ une nouvelle opération indivisible $P(s_1, s_2, \ldots, s_k)$. Selon la notation habituelle, $e(s_k)$ désigne la valeur du sémaphore s_k. L'effet de la nouvelle opération est le suivant :

Si tous les sémaphores s_i $(1 \leqslant i \leqslant k)$ ont une valeur $e(s_i)$ positive, la primitive $P(s_1, \ldots, s_k)$ est exécutée : dans ce cas elle agit simultanément sur tous les sémaphores et elle décrémente leur valeur de un ; le processus appelant poursuit son exécution. Si un au moins des sémaphores s_i a une valeur négative ou nulle, la primitive n'est pas exécutée et le processus appelant se bloque. L'évolution du processus bloqué devient à nouveau possible lorsque tous les sémaphores ont repris une valeur positive ; le processus réexécute alors la primitive P.

1) Montrer que la primitive généralisée $P(s_1, s_2, \ldots, s_k)$ n'est pas équivalente à la séquence de primitives $P(s_1)$; $P(s_2)$; ... ; $P(s_k)$. Une primitive V généralisée est-elle nécessaire ?

2) Programmer avec la primitive P généralisée, le problème des philosophes aux spaghetti.

On trouvera à la fin du chapitre 7 des exercices sur l'exemple de coopération de processus (système d'entrée-sortie) présenté en 7.5.

3

GESTION DE L'INFORMATION

3.1 INTRODUCTION

Ce chapitre est consacré aux techniques qui permettent de gérer l'information présente dans un système. Cette information représente des **objets** sur lesquels on veut effectuer des traitements. Nous nous intéressons aux mécanismes qui permettent de créer, retrouver, modifier et détruire les représentations des objets. Nous avons écarté les aspects syntaxiques des langages offerts aux utilisateurs, y compris ceux du langage de commande. Les mécanismes de protection, qui contrôlent l'emploi des objets, apparaîtront au chapitre 5. Dans le présent chapitre, nous supposons souvent l'existence d'un compilateur pour imposer le respect des conventions d'accès aux objets.

Nous présentons dans l'introduction les principaux problèmes que soulève la gestion des représentations des objets. Nous exposons ensuite trois exemples de mécanismes employés, puis, en nous appuyant sur ces trois exemples, nous examinons les méthodes de représentation des objets en machine, l'accès à cette représentation et sa gestion en mémoire.

3.11 TERMINOLOGIE

Les objets sont concrétisés par une **représentation.** Nous distinguons la représentation externe employée par les utilisateurs du système et la représentation interne au système, qui tient compte de la nature du calculateur.

Pour alléger le texte, nous utilisons désormais le terme « objet » pour évoquer aussi bien l'objet lui-même que sa représentation externe ou interne.

La représentation des objets et la manière d'y accéder font appel à un certain nombre de **fonctions d'accès.** Nous examinons d'abord les fonctions associées à la représentation externe, puis celles qui sont associées à la représentation

interne et nous montrons comment on passe de la structure externe à la structure interne. Nous réservons pour noter ces fonctions les termes *désigner*, *repérer*, *renfermer*, *contenir* et *fournir*, qui ne sont employés dans ce chapitre qu'avec le sens précis qui leur est donné ci-après.

3.111 Représentation externe des objets

Les concepteurs et les utilisateurs d'un système peuvent définir des objets, grâce à un langage de programmation avec lequel il créent des **identificateurs.** On dit qu'un identificateur *désigne* un objet. Certains identificateurs, utilisés de façon universelle pour désigner des objets donnés, sont appelés **notations.** Une notation est un cas particulier de l'identificateur. Lorsqu'on a associé un identificateur à un objet, on peut assimiler l'identificateur à la représentation externe de l'objet.

Exemple 1. L'objet mathématique $E = \{ i \in N \mid 0 \leqslant i \leqslant 20 \text{ et } i \text{ premier} \}$ peut se définir en ALGOL 68 comme :

$$[1 : 9] \text{ ent premier} = (1, 2, 3, 5, 7, 11, 13, 17, 19)$$

Ici, *1, 2, ..., 19* sont des notations, *premier* est l'identificateur de l'objet.

Exemple 2. *vrai*, *faux*, *nil* sont des notations en ALGOL 68.

L'objet désigné par un identificateur peut être une constante ou peut servir d'intermédiaire pour accéder à un objet parmi d'autres. On dit dans ce cas que cet objet *repère* un autre objet.

L'emploi de *désigner* et de *repérer* est illustré par les exemples suivants.

Exemple 1. En FORTRAN, la notation *123* désigne une constante entière.

Exemple 2. En ALGOL 60, l'identificateur défini par *réel a* désigne un objet qui repère une valeur réelle.

Exemple 3. En ALGOL 68, un identificateur peut désigner un objet de mode *ent*, *rep ent*, *rep rep ent*, ... ce qui signifie qu'il désigne respectivement :

— une constante entière,
— un objet qui repère une constante entière,
— un objet qui repère un objet du mode *rep ent*.

On appelle **chaîne d'accès** à un objet une composition des fonctions *désigner* et *repérer* qui, à un instant donné, fait passer d'un identificateur à cet objet.

Dans une chaîne d'accès, un objet repéré par un autre objet peut être désigné par zéro, un ou plusieurs identificateurs. Dans pratiquement tous les langages de programmation, un identificateur permet d'accéder non seulement à l'objet qu'il désigne mais aussi à tout objet de la chaîne d'accès : en ce sens, un identificateur peut être la représentation externe de plusieurs objets.

Exemple. Considérons le programme ALGOL 68 suivant :

```
(a) début réel y := 3.2 ; rep réel xx := y ;
(b)     début tas réel x := 2.3 ;
(c)         xx := x ;
(d)     fin
(e)         y := y + xx ;
(f)         xx := y ;
(g) fin
```

Dans ce programme, nous avons défini trois identificateurs y, x et xx, qui désignent respectivement des objets Y et X de mode *rep réel* et un objet XX de mode *rep rep réel*. L'objet X désigné par x a une existence dans tout le programme, grâce à l'option *tas*. Par contre l'identificateur qui le désigne n'a d'existence que dans la région *(b-c-d)*.

A partir du point *(e)*, on note :

1) que l'objet X n'est plus désigné par aucun identificateur,

2) qu'il existe deux chaînes d'accès :

y désigne Y repère 3.2

xx désigne XX repère X repère 2.3

3) que dans l'instruction *(e)* :

— y, en partie gauche de l'affectation, donne accès à l'objet Y,
— y, en partie droite de l'affectation, donne accès à l'objet *3.2*,
— xx, quant à lui, donne ici accès à l'objet *2.3*.

4) que dans l'instruction *(f)* :

— xx donne accès à l'objet XX et y à l'objet Y (qui repère alors la constante *5.5*).

3.112 Représentation interne des objets

La mémoire d'un ordinateur est constituée d'**emplacements** qui ont à tout instant un **contenu.** Les emplacements sont de taille quelconque, accessibles comme un tout et généralement inconnus de l'utilisateur du système.

Nous utilisons encore pour les emplacements le terme *désigner* vu plus haut et nous appelons **nom** l'information qui désigne un emplacement.

Le processeur peut utiliser les noms pour lire ou modifier le contenu d'un emplacement nommé et il peut interpréter un contenu comme une instruction, comme une valeur entière, une chaîne de bits,..., ou comme un nom.

On doit convertir la représentation externe des objets et de leurs fonctions d'accès en une représentation interne s'appuyant sur des emplacements, des contenus et sur leurs fonctions d'accès.

Dans le système, la représentation externe est convertie en un couple *(emplacement, contenu)* et on appelle **nom de l'objet** le nom de l'emplacement. Une constante est un couple dont le contenu ne doit pas varier. On dit alors que l'emplacement *renferme* la représentation interne d'une constante.

Un objet qui repère un autre objet est un couple dont le contenu peut varier. Dans ce cas, on dit que l'emplacement *contient* la représentation d'un objet.

Cette représentation est soit un nom, soit la représentation interne d'une constante.

La conversion des fonctions d'accès aux objets se fait selon les règles suivantes :

FONCTIONS D'ACCÈS EXTERNE		FONCTIONS D'ACCÈS INTERNE
un identificateur *désigne* un objet	⇔	un nom *désigne* un emplacement
un objet est une constante	⇔	un emplacement *renferme* une constante
un objet *repère* un objet	⇔	un emplacement *contient* un nom qui *désigne* un emplacement

Exemple. Soit les deux instructions du langage d'assemblage du CII 10070.

a) *LW, R m*
b) *LW, R *m*

m est le nom (ici, l'adresse en mémoire) d'un emplacement qui *contient* une valeur d'adresse.

Soit *v* cette valeur. Les fonctions d'accès associées aux deux instructions sont :

a) *m désigne* un emplacement qui *contient* une valeur *(v)*

b) *m désigne* un emplacement qui *contient* le nom *v* qui *désigne* un emplacement qui *contient* une valeur.

L'application de ces règles peut être suivie d'une simplification.

La suite « nom *désigne* emplacement *renferme* constante » peut être remplacée par le seul élément « constante », lorsque l'emplacement qui contient le nom est de taille suffisante pour contenir la constante. Cette simplification n'est possible que parce que la suite des fonctions *désigne* et *renferme* ne relie le nom qu'à cette seule constante.

Exemple. Dans certaines machines, l'adressage dit « immédiat » permet de représenter une constante dans une instruction, au lieu d'un nom. Dans ce cas, lorsqu'un identificateur *désigne* une constante, celle-ci peut être représentée directement par une valeur dite « immédiate ». On peut éviter de convertir l'identificateur en un nom. C'est le seul cas où la représentation interne d'un objet n'est pas un couple *(emplacement, contenu)*.

Remarque 1. Lorsqu'un objet *repère* une constante, on peut le représenter par un emplacement qui *renferme* la représentation interne de la constante.

Remarque 2. La distinction entre les fonctions *renfermer* et *contenir* peut être assurée par un dispositif câblé de protection. Dans ce cas, on interdit toute écriture dans les emplacements qui renferment des constantes.

Remarque 3. Nous réservons le terme **adresse** pour les cellules de la mémoire physique. Dans certains systèmes, les instructions n'utilisent pas des adresses, mais des noms d'emplacements en mémoire virtuelle. L'objet représenté par un emplacement et son contenu n'a pas une adresse fixe mais peut se

déplacer en mémoire physique. Par définition, dans ce chapitre, le nom ne change pas lorsque change l'adresse de l'emplacement qu'il désigne. Le mouvement en mémoire physique est étudié au chapitre 4.

3.113 **Objets composés**

Nous n'avons considéré jusqu'ici que des objets simples et leur désignation. Un système comprend aussi des objets composés qui sont obtenus par composition d'autres objets. La représentation interne d'un objet composé s'appelle un **descripteur** ; celui-ci indique la nature de l'objet (nombre et nature des éléments, présence d'un ordre entre les éléments,...) et des emplacements qui le contiennent (nombre et classement des emplacements,...). On appelle **nom** d'un objet composé le nom de l'emplacement qui contient son descripteur.

Associée au descripteur de l'objet composé, une **fonction d'accès** permet de désigner soit l'objet composé lui-même, soit un ou plusieurs des objets qui le constituent. Cette fonction d'accès a des paramètres et *fournit* le nom d'un emplacement ou un contenu. Un objet auquel est associée une fonction d'accès est dit **accessible.**

Exemple 1. Un tableau en ALGOL 60 est un ensemble ordonné d'objets de même type. Si t désigne un tableau à deux dimensions, $t[i, j]$ est la fonction d'accès à un élément du tableau ; $t[i, j]$ fournit un entier si le tableau est de type « tableau d'entiers ».

Indiquons un descripteur possible pour ce tableau. Il contient une adresse origine et un triplet par paramètre. Chaque triplet précise :

— l'indice du paramètre (ici *1* ou *2*),
— la valeur actuelle de la borne inférieure du domaine de variation du paramètre,
— la valeur actuelle de la borne supérieure du domaine de variation du paramètre.

Ces informations permettent :

— de vérifier que les valeurs de i et j appartiennent aux domaines de variation,
— d'accéder à l'emplacement désigné, après un calcul faisant intervenir l'adresse origine, les valeurs de i, j et les valeurs du triplet.

Si v_1 et v_1' représentent les valeurs inférieures et supérieures du domaine de variation du premier paramètre, v_2 et v_2' celles du second paramètre, l'emplacement désigné par $t[i, j]$ est fourni par :

$$origine + (valeur(i) - v_1) * (v_2' - v_2 + 1) + valeur(j) - v_2$$

Exemple 2. Une structure, en PL/1, est un ensemble d'objets du langage. Si a est l'identificateur d'une structure et b l'identificateur d'un champ de la structure, $a.b$ est la fonction d'accès à un élément de la structure.

Une représentation interne de la structure peut être la réunion, en un ensemble d'emplacements consécutifs, des emplacements correspondant aux champs. Un tel emplacement peut contenir une valeur (nom ou constante) si le champ est un objet élémentaire, un descripteur si le champ est un objet composé. Le descripteur de la structure spécifie alors l'origine et la taille de cet ensemble d'emplacements. Chaque identificateur de champ est traduit en un indice, et la fonction d'accès est une indexation dans cet ensemble d'emplacements.

Exemple 3. La fonction d'accès aux composants d'un fichier se déduit généralement de l'organisation du fichier (fichier séquentiel, fichier à accès direct, ...). Considérons un fichier séquentiel comme un objet composé d'articles. Le descripteur du fichier est constitué de l'adresse d'origine des emplacements qui contiennent les articles (n° de dérouleur, adresse d'une piste) et d'un index (matérialisé par la tête de lecture du dérouleur, ou adresse de la piste courante). La fonction d'accès fournit l'article spécifié par l'index, lequel augmente à chaque accès (avancement de la bande, augmentation de l'adresse de piste courante). L'article peut être lui-même un tableau, une structure. L'accès à un objet composant nécessite alors la mise en œuvre d'une nouvelle fonction d'accès.

La fonction *fournir* peut entrer dans la composition d'une chaîne d'accès si celle-ci fait intervenir des objets composés. On utilise aussi le terme « chaîne d'accès » pour la composition de fonctions internes *désigner, renfermer, contenir, fournir* qui mène jusqu'à l'objet à partir du nom associé à un identificateur.

Exemple. Dans l'instruction suivante du CII 10070

LW, R m, X

où *X désigne* un registre d'index, la chaîne d'accès mise en œuvre est « { *m, X* } *fournit* un emplacement qui *contient* une valeur ».

On trouvera plusieurs approches de la représentation des objets et des fonctions d'accès prescrites dans les langages de programmation dans [Abrial, 72 ; Pair, 71, 72 ; Trilling, 73 ; Verjus, 73 ; Wegner, 71].

3.114 Durée de vie des objets

On appelle **durée de vie** d'un objet le temps pendant lequel il est accessible. La durée de vie d'un objet est précisée par le langage au moyen duquel on le fait apparaître ou disparaître. Un fichier est créé ou détruit par une opération explicite et sa durée de vie n'est pas liée au processus qui le crée. Un objet créé dans un bloc d'ALGOL 60 est implicitement détruit à la sortie de ce bloc. Un objet créé dans un programme FORTRAN est détruit quand l'exécution du programme est terminée.

Lorsqu'un objet est détruit, les emplacements de mémoire qui contenaient sa représentation interne deviennent disponibles pour d'autres objets. Si la représentation interne de l'objet subsiste à la destruction de l'objet, aucun nom ne doit la désigner.

Remarque. On ne doit pas confondre la durée de vie d'un objet avec celle de l'identificateur qui le désigne. Ainsi, dans certains systèmes, lorsque dans un processus donné on ferme un fichier, l'identificateur qui le désigne n'est plus utilisable bien que l'objet fichier existe encore. Ce n'est qu'à la destruction que l'objet fichier disparaît.

3.115 **Notion de segment**

Un **segment** est un objet composé constitué d'une suite linéaire d'emplacements numérotés *0, 1, 2, ...* Il sert à regrouper des objets de même nature, de même protection, de même durée de vie. C'est dans un segment que l'on représente souvent un objet composé. Le segment est généralement la plus petite unité partageable d'un système (au sens de 3.121). La taille d'un segment peut changer au cours du temps. Comme exemples d'objet rangé dans un segment, on peut citer : une procédure, un fichier, un tableau, une pile.

Certains segments sont désignés par les utilisateurs, d'autres sont créés par le système sans qu'il leur soit associé d'identificateur. Comme tout objet composé, un segment a un nom et un descripteur. Le nom désigne l'emplacement renfermant le descripteur.

3.116 **Procédure**

Plusieurs objets se rattachent à la notion de procédure. Nous précisons ici ceux qui nous sont utiles. A une procédure nous associons son texte-source, son objet-procédure, un segment-procédure et plusieurs procédurus.

Le **texte-source** est la suite de caractères qui est la représentation interne du texte écrit par un programmeur. Il est traité par un compilateur ou un interpréteur.

L'**objet-procédure** est la suite de constantes produite par le compilateur et destinée à être interprétée par un processeur comme des instructions, des valeurs initiales ou des valeurs constantes. C'est un objet composé qui est en général conservé dans un segment (ou un fichier), le **segment-procédure,** dont la durée de vie dépend d'opérations explicites de création et de destruction. Un segment-procédure peut contenir plusieurs objets-procédures.

Un traitement particulier d'un objet-procédure par un processeur, on dit aussi une **exécution** de la procédure, exécute une suite d'instructions qui est tout ou partie d'un processus (cf. 2.2). Une exécution de procédure crée des objets locaux qui ne servent qu'à cette exécution et utilise des paramètres et des objets externes.

En général les emplacements des objets locaux et des paramètres sont obtenus d'une manière standard pour un système donné. Si l'ensemble des objets externes qu'une procédure peut désigner change d'une exécution à l'autre, il faut indiquer sa composition avant chacune d'elles. Nous appelons **procédurus** le couple (objet-procédure, ensemble des objets externes), préparé avant toute exécution de procédure.

3.12 CONTRAINTES APPORTÉES PAR LE SYSTÈME

La nature des objets ne détermine pas entièrement leur représentation. On doit encore tenir compte du partage des objets entre plusieurs utilisateurs et de la limitation de la taille des matériels utilisés comme support.

3.121 **Partage des objets et utilisation des noms**

Nous disons (cf. 2.222) qu'un objet est partagé s'il est accessible à plusieurs processus. Le partage des objets intervient dans plusieurs circonstances :

a) *Partage d'un objet avec utilisation d'un même nom*

Soit deux processus qui partagent un objet. Pour que cet objet puisse être désigné par le même nom dans les deux processus, il faut réserver ce nom, que l'objet soit utilisé ou non. Tous les objets partageables entre deux processus ont donc des noms réservés. Il en résulte que l'ensemble des noms que peut utiliser un processus doit comprendre les noms de tous les objets partagés, comme c'est le cas dans le système ESOPE (cf. 3.4). On verra lors de l'étude du BURROUGHS B6700 comment on peut alléger cette contrainte (cf. 3.3).

b) *Partage d'un objet avec utilisation de noms différents*

Si on admet qu'un segment peut recevoir un nom différent dans chaque processus, l'ensemble des noms que peut utiliser un processus peut ne comprendre, en plus des noms des objets privés de ce processus, que les noms des objets partagés auxquels il accède. Il faut alors autre chose que le nom pour désigner un objet partagé. Dans le système CLICS, par exemple (cf. 3.2), les objets partagés sont rangés dans une arborescence (comme dans un fichier) et sont désignés de façon unique par l'identificateur du nœud (ou de la feuille) correspondant ; cet identificateur reçoit une représentation interne formée d'une chaîne de caractères, que le système est capable de transformer, au moment voulu et pour un processus donné en un nom propre au processus.

c) *Réutilisation des noms*

Soit une procédure réentrante utilisée par deux processus. Les identificateurs déclarés dans le texte-source de la procédure sont les mêmes pour tous les processus et sont transformés, à la compilation, en un nom unique. Pourtant, à l'exécution, certains noms, ceux des variables locales par exemple, doivent désigner des emplacements différents.

En résumé, le partage des objets ou des noms introduit les conditions suivantes :

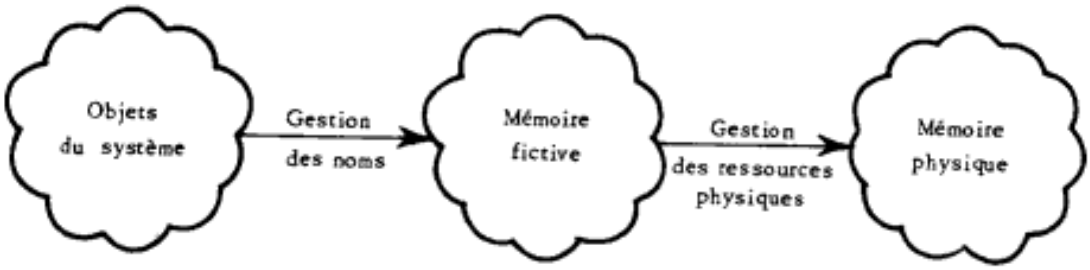
— un emplacement qui contient un objet partagé est parfois désigné par deux noms différents dans deux processus différents,

— le même nom doit pouvoir désigner des emplacements différents dans des processus différents.

3.122 **Interférence avec la gestion des ressources physiques**

Les mémoires physiques des systèmes sont limitées en taille et ont des délais d'accès très variables. L'existence de supports secondaires sur lesquels les processeurs centraux ne peuvent exécuter des instructions impose la mobilité des objets sur leur support. Cette mobilité complique la gestion de l'information en y mêlant la gestion des supports physiques. Pour séparer les deux fonctions,

on introduit parfois la notion de **mémoire fictive,** mémoire centrale hypothétique qui est suffisamment grande pour contenir tous les objets du système et qui est composée d'une suite d'emplacements numérotés *0, 1, 2, ..., N*. Chaque objet du système, et en particulier chaque **segment**, a alors une structure propre qui est appliquée dans la mémoire fictive. Cette application ne change pas pendant la durée de vie de l'objet, si bien que les objets ont un nom fixe dans la mémoire fictive. L'application de la mémoire fictive dans les supports physiques est faite par l'allocateur de mémoire à l'insu de la gestion des noms. On verra au chapitre 4 les diverses techniques possibles.



Remarque. La mémoire fictive est une notion attachée au système, tandis que la mémoire virtuelle (cf. 2.2) est en général associée à un processus seulement.

3.123 Représentation du système

Dans ce chapitre, le système est représenté comme un ensemble de processus manipulant des segments qui sont appliqués dans la mémoire fictive. Chaque segment a une taille variable, indépendante des autres segments. Leur ensemble constitue **l'espace des segments.**

3.13 MODIFICATIONS DE LA CHAINE D'ACCÈS A UN OBJET

Les objets que crée l'utilisateur sont désignés par un identificateur. L'objet qu'utilise le processus doit être rangé dans un emplacement et désigné par un nom. Ce nom fournit soit directement, soit par une succession de relations, l'emplacement de l'objet. On rappelle que la chaîne d'accès est la composition des relations *désigner, renfermer, contenir, fournir* qui vont du nom à l'objet ; on appelle **liaison** la construction de cette chaîne. Les différents éléments de cette chaîne ne sont pas tous établis en même temps. Lorsque la chaîne est complète, on dit que le nom et l'objet sont **liés.** Rappelons que l'objet qui est en bout de la chaîne d'accès peut être soit une constante (cas des opérations arithmétiques sur les entiers, par exemple), soit un nom (cas des opérations sur les pointeurs, par exemple).

Soit aRb et $bR'c$ deux maillons de la chaîne d'accès ; l'accès à c depuis a peut être réalisé de deux façons :

— par **substitution**, faite une fois pour toutes. La relation $aR''c = aRbR'c$ est représentée, b est alors perdu,

— par **chaînage**. La relation R'' n'est pas représentée et le cheminement $aRbR'c$ est effectué à chaque accès.

La substitution permet de gagner du temps lors de l'accès à l'objet, mais on perd de l'information. Le cheminement dans la chaîne d'accès peut être accéléré en gardant le résultat de cheminements partiels dans des registres associatifs.

La transformation de l'identificateur en un nom est toujours effectuée par le compilateur du langage externe (ou par un interpréteur). Lorsque la liaison ne peut être faite par le compilateur, celui-ci met en place les objets qui la permettront plus tard (par exemple, une chaîne de caractères représentant l'identificateur pour l'éditeur de liens).

3.131 Objets liés dès la compilation

Même lorsque les noms et les objets sont liés dès la compilation, la gestion des emplacements peut faire intervenir une succession de noms et de fonctions d'accès. Cette succession traduit le fait que les noms sont relatifs à l'ensemble des emplacements qu'ils peuvent désigner et qu'ils peuvent changer lorsque cet ensemble est modifié.

Exemple. Considérons le système SIRIS 7 sur CII 10070. Le compilateur range tous les objets locaux d'une section de programme dans un segment (module de chargement) et transforme chaque identificateur en un nom translatable, qui désigne un emplacement dans le segment. Le chargeur applique plusieurs segments dans la mémoire virtuelle et substitue aux noms translatables des noms virtuels qui désignent un emplacement en mémoire virtuelle. Durant l'exécution, le processeur transforme, à chaque référence, ce nom virtuel en une adresse en mémoire centrale. Il utilise pour cela une table (mémoire topographique) qui est entretenue par l'allocateur de mémoire. Comme les adresses varient selon la répartition dynamique de la mémoire, elles ne sont pas substituées aux noms virtuels. Ceux-ci sont conservés ainsi que, pour chaque processus, la table qui définit la relation entre les noms virtuels et les adresses.

3.132 Noms et objets libres après compilation

Certains objets ne peuvent être liés à la compilation. Ce sont les paramètres effectifs d'une procédure ou les objets externes.

a) *Les paramètres*

Une procédure est écrite en utilisant des paramètres formels qui sont des identificateurs désignant des objets fictifs. Ce n'est qu'à l'appel de la procédure que les objets réels sont désignés à l'aide des paramètres effectifs.

Le remplacement des paramètres formels par les paramètres effectifs ne peut pas toujours être fait à la compilation, car les paramètres effectifs peuvent n'être connus qu'à l'exécution. Une solution consiste à créer un nom intermédiaire qui désigne :

— *nil* avant l'appel, (*nil* est un nom fictif),
— le nom du paramètre effectif après l'appel.

Exemple. Soit, dans le système CLICS :

procédure p(f) ; début ... fin ;
p(e) ;

Les identificateurs *e* et *f* sont transformés à la compilation en *nom(e)* et *nom(f)* qui désignent chacun un emplacement. A l'appel de la procédure, l'emplacement désigné par *nom(f)* reçoit la valeur *nom(e)*. Aucune instruction de la procédure ne peut modifier le contenu de l'emplacement désigné par *nom(f)*.

b) *Les objets externes*

Les objets externes sont des objets qui ne sont pas créés dans le programme compilé. Le compilateur crée un objet composé intermédiaire, appelé **lien**. Un lien contient au moins la chaîne de caractères représentant l'identificateur et parfois des informations permettant de retrouver tous les noms désignant ce lien. A la compilation le nom de l'objet externe est le nom du lien auquel il est associé.

On appelle **édition de liens** la liaison des objets externes. Elle peut être effectuée avant l'exécution du programme (liaison statique), ou bien en cours d'exécution (liaison dynamique). On en verra un exemple en 3.2.6.

L'édition de liens peut :

— conserver le lien et y ajouter le nom de l'objet externe,
— supprimer le lien et substituer le nom de l'objet au nom du lien partout où il est employé.

Exemple 1. Une référence externe dans un programme en langage d'assemblage désigne un lien après assemblage. L'éditeur de liens substitue le nom de l'objet au nom du lien et détruit le lien.

Exemple 2. Un bloc de contrôle de fichier *(DCB)* est un lien. Il contient la chaîne de caractères représentant l'identificateur du fichier, son nom (ce peut être l'adresse d'un périphérique) et plus généralement une description d'un objet de type fichier (cf. 1.22).

3.2 GESTION DES NOMS DANS LE SYSTÈME CLICS

3.21 INTRODUCTION

Le système CLICS (Classroom Information and Computing Service) [Clark, 71a] est une présentation idéalisée à des fins pédagogiques du système MULTICS réalisé sur une machine HONEYWELL 645.

CLICS comporte un nombre fixe de processus. Chacun d'eux peut être associé à un utilisateur pour la durée d'une session (intervalle entre les appels des procédures *LOGIN* et *LOGOUT*). C'est cette durée que nous appellerons, par abus de langage, durée de vie du processus. Chaque processus exécute des

objets-procédure travaillant sur des objets collections de données. Tous ces objets, partageables pour la plupart par l'ensemble des processus, sont contenus dans des segments.

Chaque processus *désigne* des segments. Un segment donné doit être accessible par plusieurs processus à la fois. Cet accès est contrôlé par un système de protection.

Le moniteur de CLICS est un ensemble de procédures partagées qui sont exécutées par les processus associés aux utilisateurs. Chaque processus peut donc, à un instant donné, exécuter des fonctions du moniteur ou des procédures qui ont été écrites par un utilisateur. Une commande comme ! FORTRAN (cf. 1.22) se traduit par un appel de la procédure du moniteur qui constitue le compilateur FORTRAN.

Dans la suite de ce chapitre, nous présentons la mémoire segmentée et les différents objets qu'elle peut contenir, la façon dont un processus les nomme, et, enfin, la façon de réaliser l'édition de liens. Cette opération, comme il a été vu en 3.13, remplace l'identificateur d'un objet par un nom qui permet d'y accéder. Nous nous intéressons aux mécanismes d'accès, sans décrire les mécanismes de protection. Nous signalons cependant, quand c'est le cas, les dispositifs d'accès qui ne se justifient que pour des raisons de protection.

Quand nous évoquons dans le texte un segment particulier, nous appelons S l'objet, *TOTO* son identificateur et s son nom.

3.22 LA MÉMOIRE SEGMENTÉE

Un segment, dans CLICS, peut contenir de 1 à 2^{24} mots. Un objet, appelé **descripteur de segment** le situe dans une mémoire fictive (cf. 3.12) de 2^{40} mots. La gestion des ressources physiques n'est pas considérée ici.

Le descripteur de segment comprend :

— la longueur du segment,

— l'adresse, en mémoire fictive, du premier mot (base) du segment (codée sur 40 bits),

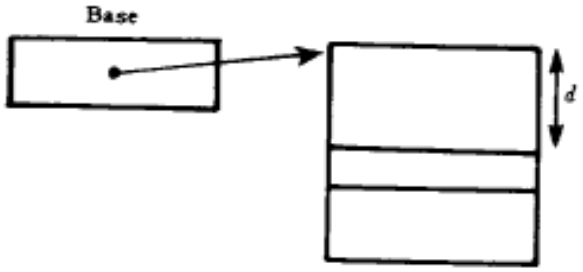
— des indicateurs utilisés par le mécanisme de protection.

Tout accès à un mot du segment fait intervenir le descripteur de segment.

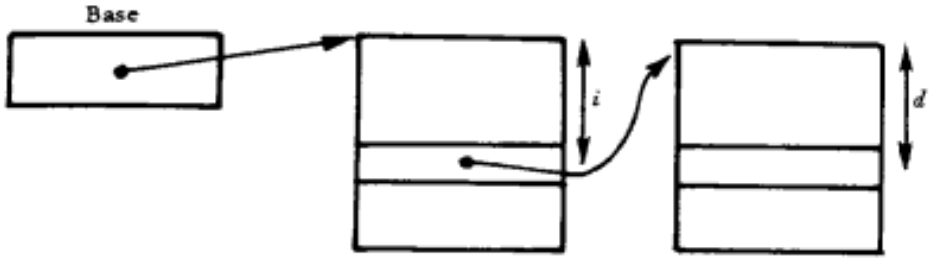
L'application des segments dans la mémoire fictive, fait passer d'un ensemble de suites, les segments, dans une suite unique, la mémoire fictive. Cette transformation nécessite une gestion de la mémoire fictive analogue à la gestion de la mémoire physique telle qu'elle est présentée en 4.44. Cette gestion est faite dans CLICS à l'aide du dispositif de pagination ci-après.

Les segments et la mémoire fictive sont découpés en pages de taille fixe (256 mots). Soit K le nombre de pages du segment et d un numéro d'emplacement dans une page :

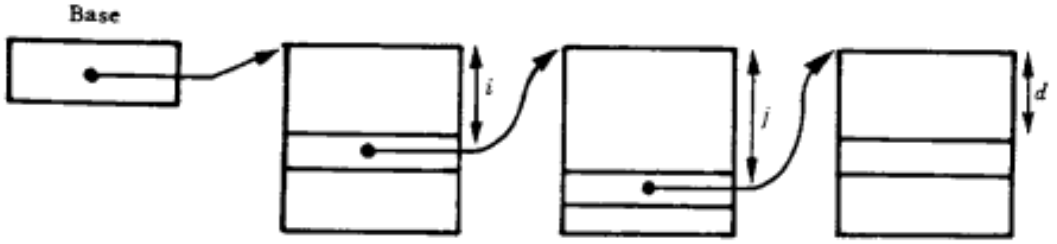
— si $K \leqslant 1$, l'adresse de base du segment est celle d'une page de mémoire fictive associée au segment.



— si $1 < K \leqslant 2^8$, l'adresse de base du segment est celle d'une page de mémoire fictive utilisée comme table de K pages. Chaque élément i de cette table est associé à la page numéro i du segment et désigne la page de mémoire fictive qui lui correspond.



— si $2^8 < K \leqslant 2^{16}$, l'adresse de base du segment est celle d'une page de mémoire fictive utilisée comme table de tables de pages. Chaque élément i de cette table est associé aux pages de numéros $256 * i$ à $256 * i + 255$ du segment et désigne une page de mémoire fictive qui contient une table de pages. L'élément j de cette dernière table est associé à la page de numéro $256 * i + j$ du segment et désigne la page de mémoire fictive qui lui correspond.



On constate que l'adresse de base d'un segment, contenue dans un descripteur, change lorsque la taille d'un segment atteint une page ou 256 pages.

Par la suite on emploiera « adresse fictive » pour « adresse en mémoire fictive » et, dans les schémas, on représentera l'adresse fictive du mot de numéro d d'un segment comme si le segment avait moins d'une page.

Remarque. L'accès à un mot de segment peut faire intervenir plusieurs consultations de table. Ces consultations peuvent être évitées en gardant le résultat des consultations les plus récentes dans des registres associatifs. Nous ne tenons pas compte de cette possibilité dans la suite du chapitre.

3.221 **Désignation d'un segment par un processus**

Tout segment est désigné par un nom, qui est le nom d'un emplacement contenant le descripteur du segment. Un segment peut avoir plusieurs noms ; chacun des emplacements correspondants contient alors une version du descripteur (Fig. 1).

Il serait souhaitable de n'accéder à un segment que par un descripteur unique plutôt que par de multiples versions de celui-ci. En effet chaque fois que la longueur ou la protection d'un segment changent, ces informations doivent être remises à jour dans toutes les versions de son descripteur.

Mais il existe plusieurs raisons pour multiplier les versions du descripteur d'un segment et partant pour avoir plusieurs noms pour un segment.

a) Soit un segment partagé par deux processus. Ce segment n'est pas nécessairement utilisé avec le même mode d'accès : un processus peut y écrire, l'autre seulement le lire. Aussi l'accès à un segment nécessite non seulement le couple (adresse, longueur) qui le situe mais également une information précisant le mode d'accès. C'est le descripteur qui fournit toutes ces informations : le couple est unique pour un segment donné, le mode d'accès est propre à chaque processus.

b) Soit deux processus qui désignent le même segment. Si on veut conserver un nom unique dans les deux processus, ce nom doit être réservé, que l'on accède ou non au segment. Les noms des segments partagés sont alors attribués de façon statique dans chaque processus. Pour utiliser moins de noms et en particulier pour ne pas nommer les segments non utilisés par un processus, les noms sont alloués dynamiquement à chaque processus. Il n'y a plus de raison pour que les segments partagés aient le même nom. Par contre il est nécessaire qu'ils aient le même couple (adresse, longueur).

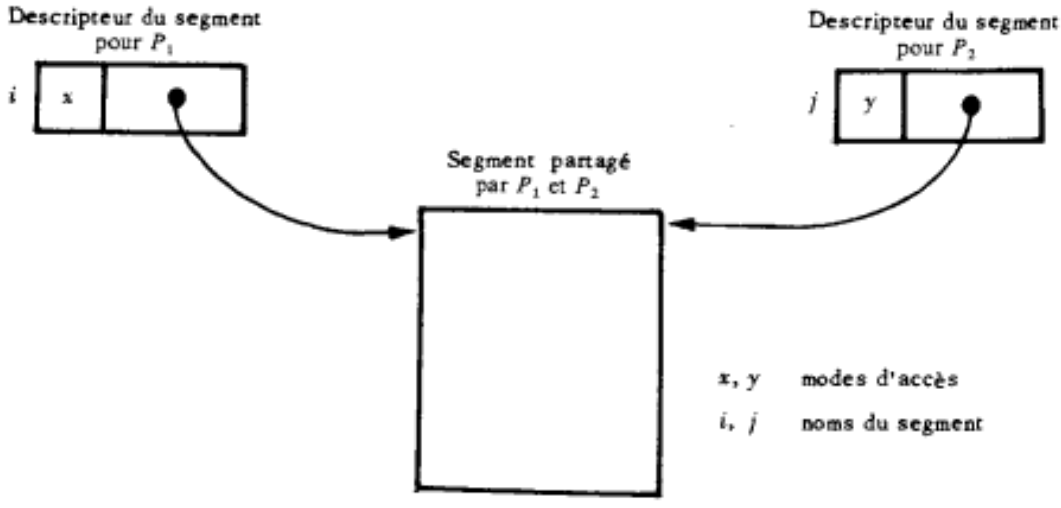


Figure 1. Noms d'un segment.

3.222 **Descriptif des segments d'un processus**

Le nom s_i par lequel un processus P_i désigne un segment S est un entier, compris entre 0 et $2^{16} - 1$, correspondant au mot de numéro s_i d'un segment appelé **descriptif du processus** P_i ; ce mot contient un descripteur du segment S. Si un autre processus P_j désigne S par le nom s_j, le descriptif de P_j a pour mot de numéro s_j un autre descripteur de S.

A chaque processus est associé un descriptif unique qui contient les descripteurs de tous les segments nommés par le processus. Le descripteur du descriptif est contenu dans un registre non programmable, le registre de base du descriptif.

Un processus désigne alors un mot d'un segment S par un couple (nom du segment, index dans le segment) : le nom du segment est propre au processus qui désigne le mot, l'index dans le segment est le numéro d'ordre du mot dans le segment considéré. Le mot de numéro d du segment S a ainsi le nom (s_i, d) pour le processus P_i.

Le nom d'un mot de segment, que nous appelons **adresse segmentée**, est codé sur 40 bits (16 pour le nom du segment, 24 pour l'index dans le segment). Le processeur traduit une adresse segmentée (s_i, d) de la manière suivante : le registre de base du descriptif repère le descriptif dont le mot de numéro s_i est la représentation pour P_i du segment S ; le mot de numéro d dans ce segment est le mot recherché.

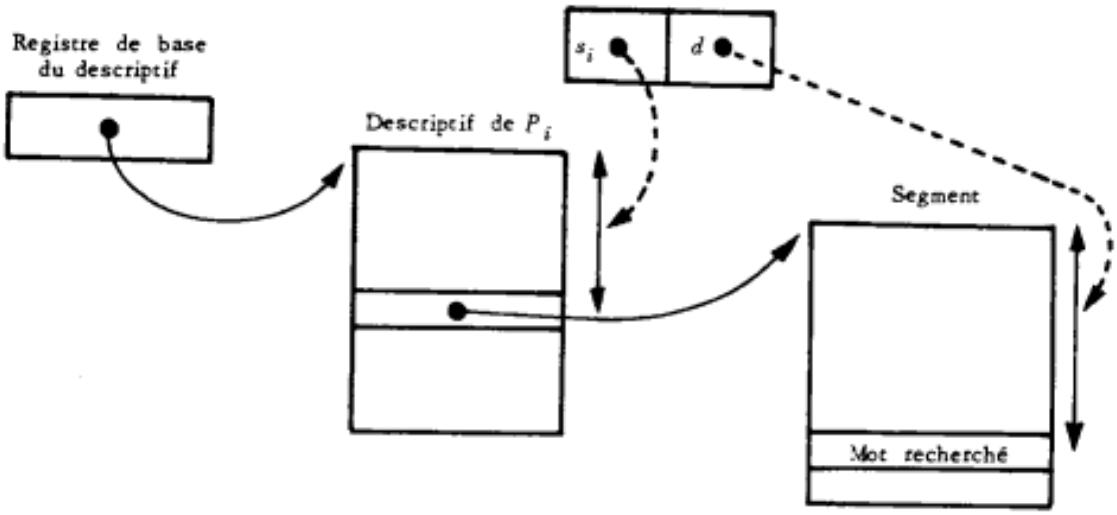


Figure 2. Adresse segmentée.

Par construction, les registres programmables et le descriptif représentent l'espace adressable du processus. Le descriptif n'est pas adressé comme les autres segments par le processus auquel il est associé (il n'a pas de descripteur dans le descriptif) et il n'est pas accessible aux autres processus.

C'est pourquoi nous séparerons dorénavant dans les schémas le descriptif, sans représenter son registre de base, et les segments partagés.

3.23 LANGAGE DE LA MACHINE ET OBJETS MANIPULÉS

Dans ce qui suit, nous observons un processus à un instant donné de son évolution et nous cherchons à montrer comment il accède aux objets (données, procédures) qu'il manipule. L'utilisateur désigne par un identificateur les objets créés par lui-même ou d'autres utilisateurs. Or le processus accède à ces objets par un nom. Nous verrons au paragraphe 3.26 quand et comment ce nom est obtenu. Auparavant, afin d'illustrer les mécanismes d'accès, indiquons le format des instructions de la machine et les objets que l'on veut manipuler.

3.231 Format des instructions

Le langage de la machine est constitué d'instructions de longueur fixe, contenues dans un mot. On considère ici que l'objet élémentaire que peut désigner un processeur est le mot d'un segment. Les instructions sont généralement composées de trois champs :

— Le code-opération. Lorsque nous aurons besoin d'illustrer une instruction, nous utiliserons un nom de code-opération très explicite.

— Le nom d'un registre (éventuellement). Chaque processus utilise 16 registres généraux notés *R0, R1, ..., R15 ;* les registres *R0, R3, R4* et *R5* ont une fonction très particulière que nous expliciterons ; *R1* et *R2* servent en cas d'interruption.

— Un champ adresse qui contient une valeur ou un nom d'opérande. Lorsque le champ adresse est un nom, nous l'écrivons entre parenthèses. Ainsi le format d'une instruction sera noté :

opération, R valeur
opération, R (nom)

On abrège souvent dans les schémas « *opération, R* » en « *op* ».

3.232 Les différents objets manipulés par l'exécution d'une procédure

Considérons une procédure *P*. En dehors des 16 registres généraux, on répartit en trois classes les objets qu'une procédure peut désigner au cours de l'exécution de *P*.

1) *Les objets externes à la procédure*

Ce sont des fichiers identifiés par un nom symbolique. Leur durée de vie est déterminée par des opérations explicites de création et de destruction.

Si on fait abstraction des problèmes de privilège d'accès, tous les fichiers (y compris le segment contenant *P*) sont accessibles depuis *P* avec les règles suivantes :

— si le fichier est une collection de données, on peut désigner chaque constituant de ce fichier pour le lire ou le modifier. Le fichier a pour support un segment appelé segment de données. Un constituant élémentaire est toujours un mot du segment contenant une valeur dite élémentaire (indépendamment de son type qui ne nous intéresse pas ici).

— si le fichier est un objet-procédure, on peut désigner certains points d'entrée de la procédure en vue de l'exécution du sous-ensemble de la procédure défini par ce point d'entrée. Le fichier a pour support un segment appelé **segment-procédure.** Un point d'entrée correspond à une instruction (contenue dans un mot) de ce segment-procédure.

2) *Les objets internes à la procédure*

Les objets internes à une procédure ne sont accessibles à un processus que pendant l'exécution de la procédure. Nous supposons que ces objets sont représentables sur un seul mot. On en distingue trois types :

a) Les objets **rémanents**, créés à la première exécution de la procédure par le processus et détruits lorsque le processus se termine. Chaque fois que *P* est exécutée par le processus, elle trouve les objets rémanents à la valeur que leur avait donnée l'exécution précédente ou, si c'est la première exécution, à une valeur initiale, fixée à la compilation et conservée dans le segment-procédure.

b) Les objets **locaux**, créés à chaque appel de la procédure par le processus et détruits au plus tard lorsque la procédure se termine. Les procédures pouvant s'appeler récursivement, les objets locaux sont gérés en pile. Chaque fois que *P* est exécutée par le processus, elle trouve les objets locaux à une valeur initiale fixée une fois pour toutes à la compilation.

c) Les **étiquettes**, qui correspondent aux noms d'emplacement du segment-procédure contenant des instructions. On peut considérer que tout objet à valeur élémentaire constante, soumis seulement à des opérations de lecture, appartient à cette classe.

3) *Les paramètres*

A tout paramètre est associé un objet qui a pour durée de vie l'exécution de la procédure et que seule la procédure peut désigner. Sa valeur est le nom d'un objet qui n'est pas interne à la procédure et dont la durée de vie est plus grande que celle de la procédure. Cet objet peut être :

— un objet interne à la procédure appelante,
— un paramètre de la procédure appelante,
— un objet externe.

Notons que les objets créés par une procédure appelante ne sont utilisables par la procédure appelée que si ce sont des objets externes ou que s'ils sont désignés par un paramètre.

Exemple. On désire compiler dans CLICS le programme ALGOL 60 suivant :

```
début procédure f ;
        début réel a ;
              procédure g ; début... a := a + 1 ...fin ;
              g ; a := a + 1
        fin
fin
```

Avec les mécanismes de CLICS, l'utilisation de a à l'intérieur de la procédure g n'est possible qu'en passant a comme paramètre à la procédure g ou en déclarant a comme externe à f et g.

La gestion de la structure de bloc d'ALGOL est étudiée dans l'exercice 4.

3.233 **Multiplicité des objets**

Le système doit satisfaire les deux objectifs suivants :

— chaque procédure et chaque collection de données est potentiellement partageable,

— les procédures peuvent s'appeler récursivement.

Il en résulte les conséquences suivantes :

a) Les objets à valeur constante sont protégés contre toute écriture.

b) Les objets externes, qu'il s'agisse de procédures ou de collections de données, ne sont pas recopiés.

L'utilisation d'un segment-procédure S par un processus P_i ne doit pas perturber son utilisation simultanée par le processus P_j. Par contre, l'utilisation par P_i d'un segment de données partagé en écriture peut interférer avec l'utilisation qu'en fait le processus P_j.

c) Les objets dont la durée de vie est égale à celle du processus (registres et objets rémanents) figurent en un exemplaire par processus. En effet l'utilisation que fait d'un tel objet le processus P_i ne doit pas interférer avec l'utilisation qu'en fait le processus P_j.

d) Les objets dont la durée de vie est égale à celle de la procédure (locaux et paramètres) figurent en un exemplaire par appel de procédure et par processus. Autrement dit, pour chaque processus P_i, il doit y avoir autant d'exemplaires d'un tel objet que d'exécutions en cours de la procédure qui l'utilise.

Le paragraphe suivant illustre la manière de nommer, dans une instruction, ces différents types d'objets.

3.24 ACCÈS AUX OBJETS

Considérons un segment S présent dans les descriptifs des processus P_i et P_j respectivement sous les noms s_i et s_j. Soit un objet d'emplacement d dans le segment S ; il a pour noms respectifs (s_i, d) pour P_i et (s_j, d) pour P_j. L'adressage indexé permet de désigner cet objet dans une instruction utilisable par plusieurs processus à la fois.

Il existe deux types d'indexation : l'adressage indexé **partiel** (noté $\underline{ip}$ dans le champ adresse) et l'adressage indexé **composé** (noté $\underline{ic}$). Le champ adresse d'une instruction est alors formé, quand c'est un nom :

— d'une marque d'indexation ($\underline{ip}$ ou $\underline{ic}$),

— d'un couple (nom de registre, déplacement), soit (R, dep), le nom du registre étant codé sur 4 bits et le déplacement sur 24.

Si le registre R contient l'adresse segmentée (s, d), alors l'adresse segmentée du mot recherché est respectivement :

(s, dep) si l'adressage est indexé partiel,
$(s, dep + d)$ si l'adressage est indexé composé.

Exemple. Si R contient $(30, 600)$, l'instruction

$$op \quad (\underline{ip}, R, 100)$$

se réfère à l'adresse segmentée $(30, 100)$, et

$$op \quad (\underline{ic}, R, 100)$$

se réfère à l'adresse segmentée $(30, 700)$.

3.241 Accès aux étiquettes du segment-procédure

Les instructions de branchement utilisent l'adressage indexé partiel ; le registre $R0$ contient à tout moment le nom par lequel le processus désigne le segment-procédure en cours d'exécution. Ainsi, toutes les instructions de branchement à une instruction du segment ont la forme :

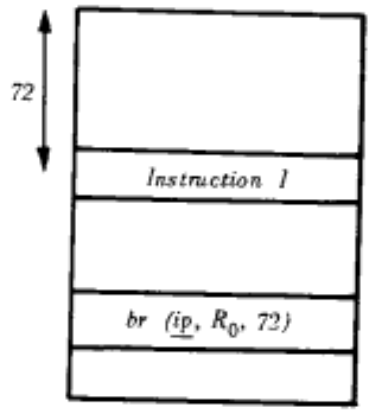
$$br \quad (\underline{ip}, R0, d)$$

où d a pour valeur le déplacement par rapport au début du segment-procédure.

Exemple. Considérons dans une procédure la séquence :

```
...
ALPHA   instruction I
...
        se brancher à ALPHA
...
```

Le segment-procédure correspondant est le suivant :



La figure 3 montre l'utilisation de ce segment par les processus P_i et P_j qui le désignent respectivement par les noms *251* et *146* :

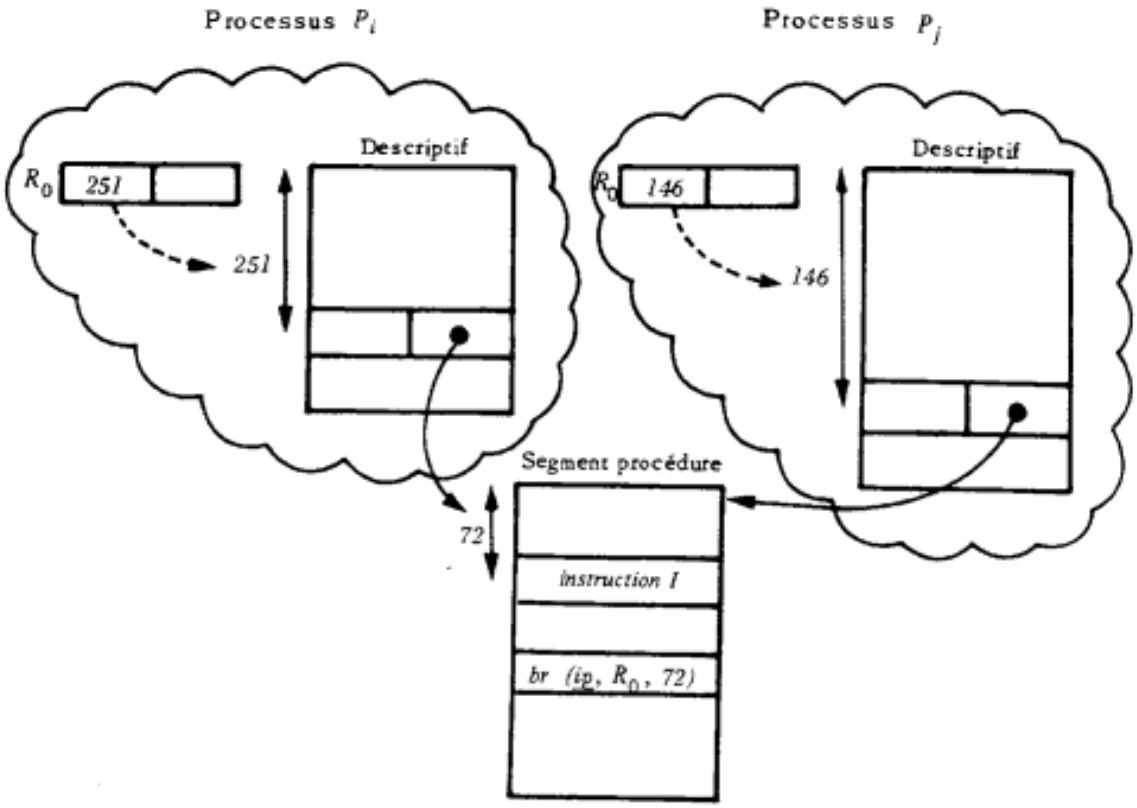


Figure 3. Accès au segment-procédure.

La partie déplacement de *R0* du processus P_i est à tout instant l'adresse relative (par rapport à la base du segment) de l'instruction en cours : *R0* est un compteur ordinal relatif.

Dans l'exemple précédent, après exécution par P_i de l'instruction de branchement considérée, le registre *R0* de P_i contient *(251, 72)*. Après exécution par P_j, son registre *R0* contient *(146, 72)*.

3.242 Accès aux objets rémanents

Rappelons que les objets rémanents ont la durée de vie du processus et qu'il doit y en avoir un exemplaire par processus. Ils ne peuvent être rangés dans le segment-procédure. Par contre, leur valeur initiale, qui doit être identique pour tous les processus, est rangée dans une zone du segment-procédure.

A la première exécution par un processus P_i d'un segment-procédure S, un nouveau segment est créé et inclus dans le descriptif de P_i. C'est le **segment de liaison** $L(S, i)$ du segment-procédure S pour le processus P_i. Les valeurs initiales des objets rémanents sont recopiées en tête de ce segment de liaison.

Notons que :

— ce segment est inaccessible aux autres processus,

— lors de toute nouvelle exécution du segment-procédure par le processus considéré, le même segment de liaison est utilisé (on verra en 3.252 comment il est retrouvé),

— chaque processus possède à un instant donné autant de segments de liaison qu'il connaît de segment-procédures différents.

En conséquence, les opérations de S sur des objets rémanents se réfèrent au segment de liaison.

Soit un segment S utilisé par deux processus P_i et P_j. Le segment $L(S, i)$ est nommé l_i par P_i et le segment $L(S, j)$ est nommé l_j par P_j. La référence aux objets rémanents s'effectue par indexation partielle, en utilisant un registre général, *R4* par convention, appelé **registre pointeur de liaison** qui contient le nom du segment de liaison de la procédure en cours d'exécution. Une opération sur un objet rémanent a la forme suivante :

op (ip, R4, déplacement)

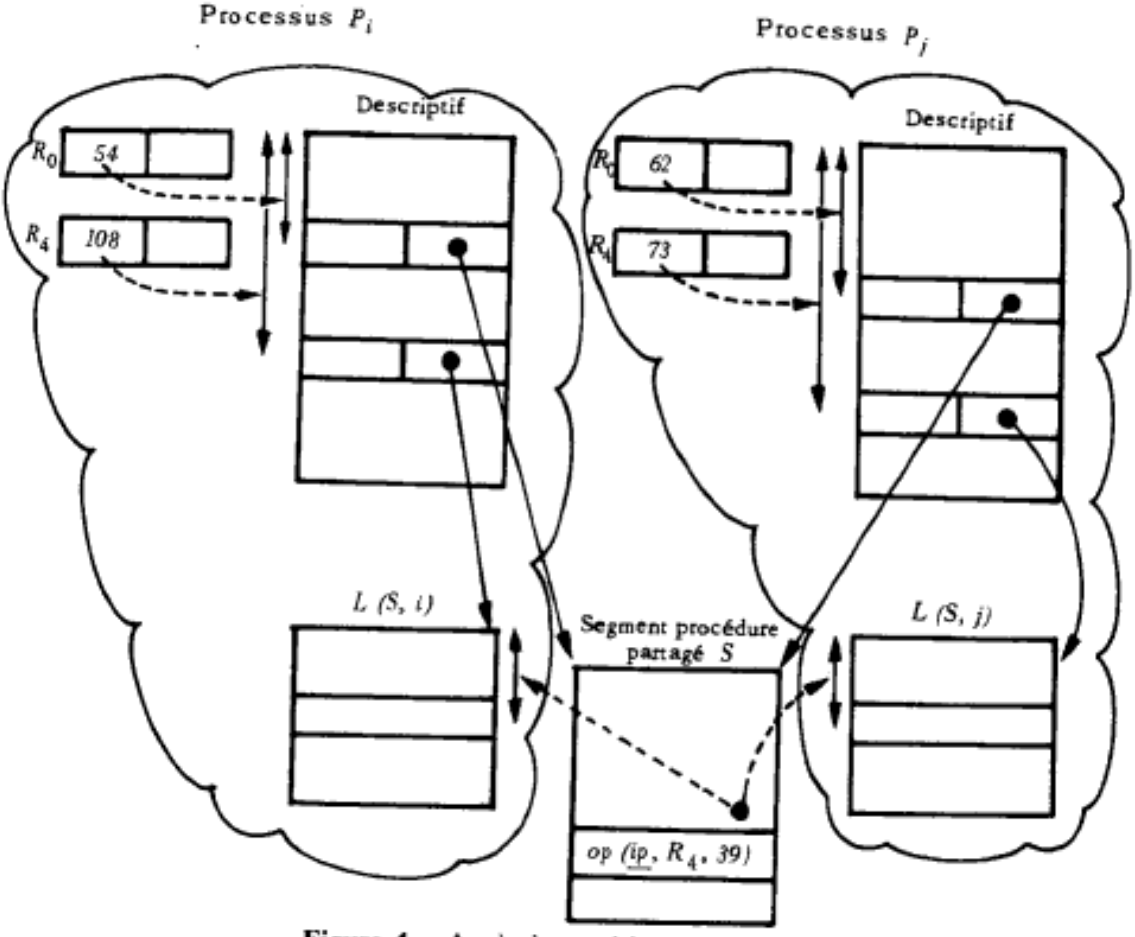


Figure 4. Accès à un objet rémanent.

Comme pour le descriptif, nous associons dorénavant les segments de liaison aux processus.

3.243 **Accès aux objets externes**

Les objets externes sont conservés dans les fichiers des utilisateurs. Tout objet qui n'est pas créé par une procédure est un objet externe à cette procédure. D'autre part, parmi les objets créés par une procédure, sont externes ceux qui ont une durée de vie supérieure à celle du processus. Nous avons vu que le segment de liaison contient les objets rémanents d'une procédure. Il contient également les références externes que peut effectuer la procédure. Soit deux processus P_i et P_j utilisant le segment-procédure S, lequel fait référence à un segment $S1$. $S1$ étant connu par P_i et P_j sous les noms s_i et s_j, on y accède par indirection en utilisant un mot du segment de liaison. Ce mot contient l'adresse segmentée de l'objet externe, c'est-à-dire :

— (s_i, déplacement dans $S1$) dans $L(S, i)$,
— (s_j, déplacement dans $S1$) dans $L(S, j)$.

Une instruction de S sur un objet externe s'écrit :

op * (*ip*, *R4*, déplacement dans segment de liaison) .

L'indirection (notée *) consiste à calculer par adressage indexé, l'adresse segmentée d'un emplacement qui contient l'adresse segmentée de l'objet référencé. Nous verrons ultérieurement à quel moment l'adresse segmentée (s, a) est rangée dans le segment de liaison et par quelle méthode.

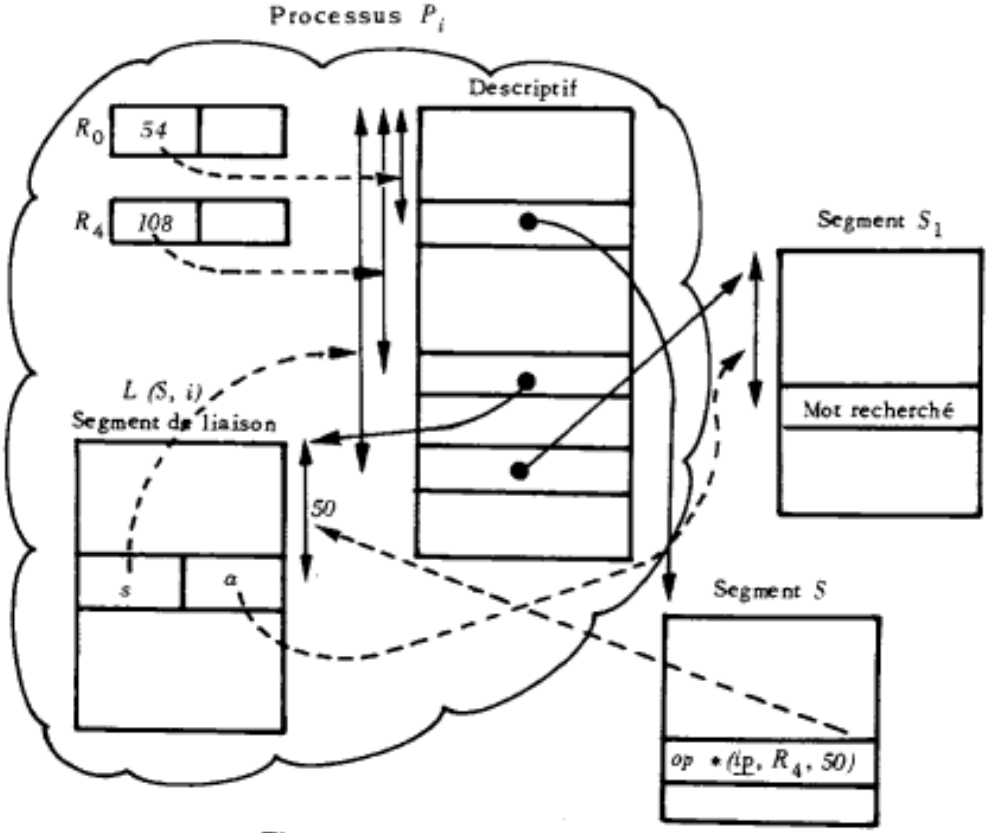


Figure 5. Accès à un objet externe.

3.244 Accès aux objets locaux

Rappelons que les objets locaux ont pour durée de vie l'exécution de la procédure et qu'ils doivent être gérés selon une structure de pile. Appelons région l'ensemble des emplacements qu'il est nécessaire de créer à cet effet dans la pile à l'appel d'une procédure donnée.

Pour chaque processus, un segment particulier constitue la pile. Le registre *R3*, appelé aussi **registre pointeur de pile**, contient l'adresse segmentée de la base de la dernière région empilée : celle de la procédure en cours d'exécution. Par ailleurs, le premier mot de la pile contient toujours l'adresse segmentée du sommet de pile. La pile associée au processus P_i sera notée p_i dans la suite.

Soit un segment-procédure S et un objet local A, auquel a été réservé le mot de numéro n d'une région. Soit p_i et p_j les piles associées aux processus P_i et P_j qui utilisent S. La référence à A est faite sous la forme :

$$op\ (\underline{ic},\ R3,\ n)$$

puisque le registre *R3* du processus P_i contient (p_i, d_i), adresse de base de la dernière région empilée par P_i et que le registre *R3* du processus P_j contient l'équivalent pour P_j.

Remarque. Cette méthode d'accès s'applique sans changement aux procédures récursives : à chaque appel de la procédure, le registre *R3* est mis à jour ; il est rétabli à chaque retour de procédure.

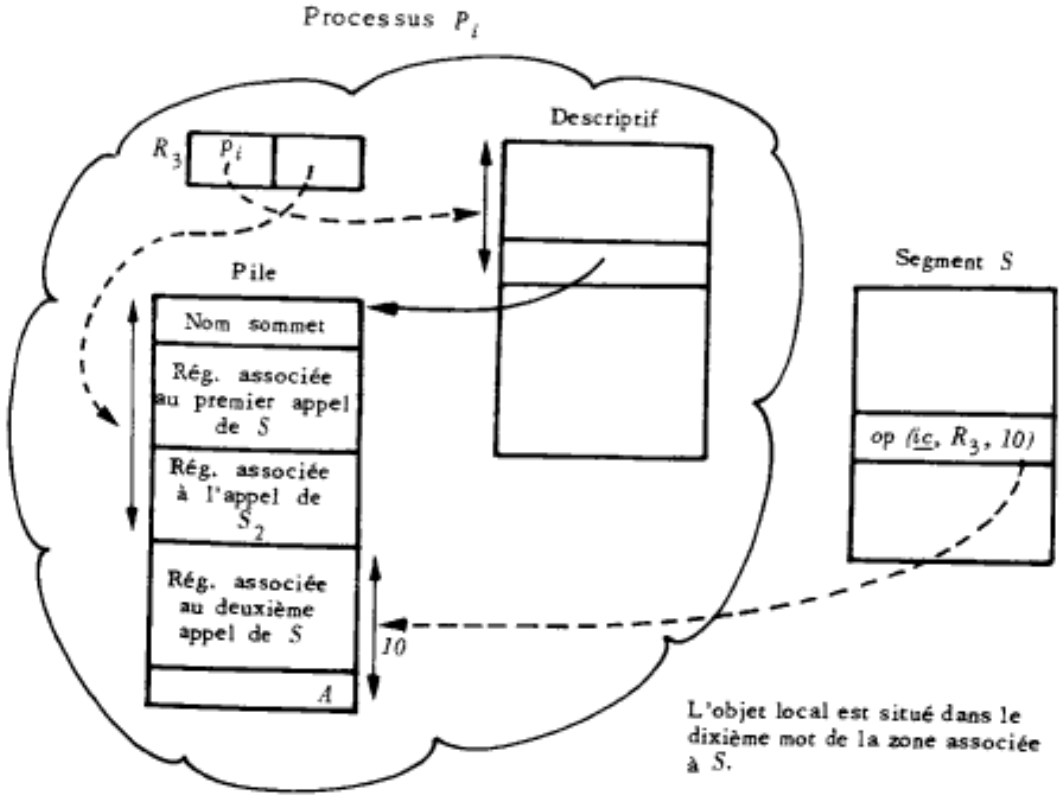


Figure 6. Accès à un objet local.

3.245 Accès aux paramètres

Rappelons que la durée de vie d'un paramètre effectif est au moins égale à celle de l'exécution de la procédure appelante. La procédure appelante calcule le nom du paramètre en tenant compte de sa nature :

— Objet externe : son nom *(s, d1)* est rangé dans le segment de liaison de la procédure appelante.

— Objet rémanent de la procédure appelante : son nom est une adresse segmentée *(l, d2)*, déduite du couple *(R4, d2)* et du contenu *l* du registre *R4*.

— Objet local de la procédure appelante : son nom est une adresse segmentée *(p, d3 + d4)* déduite du couple *(R3, d4)* et du contenu *(p, d3)* du registre *R3*.

— Valeur élémentaire : la procédure appelante crée un objet local auquel elle donne la valeur élémentaire considérée.

— Paramètre de la procédure appelante : nous verrons plus loin comment obtenir son nom.

L'accès aux paramètres dans la procédure appelée est illustré par l'exemple suivant :

Exemple. Soit l'appel de procédure $P(8, R, L, E)$ où 8 désigne une valeur élémentaire et R, L, E les identificateurs d'objets respectivement rémanent, local, externe. La procédure appelante détermine le nombre de paramètres et leurs noms et les range dans cinq mots successifs.

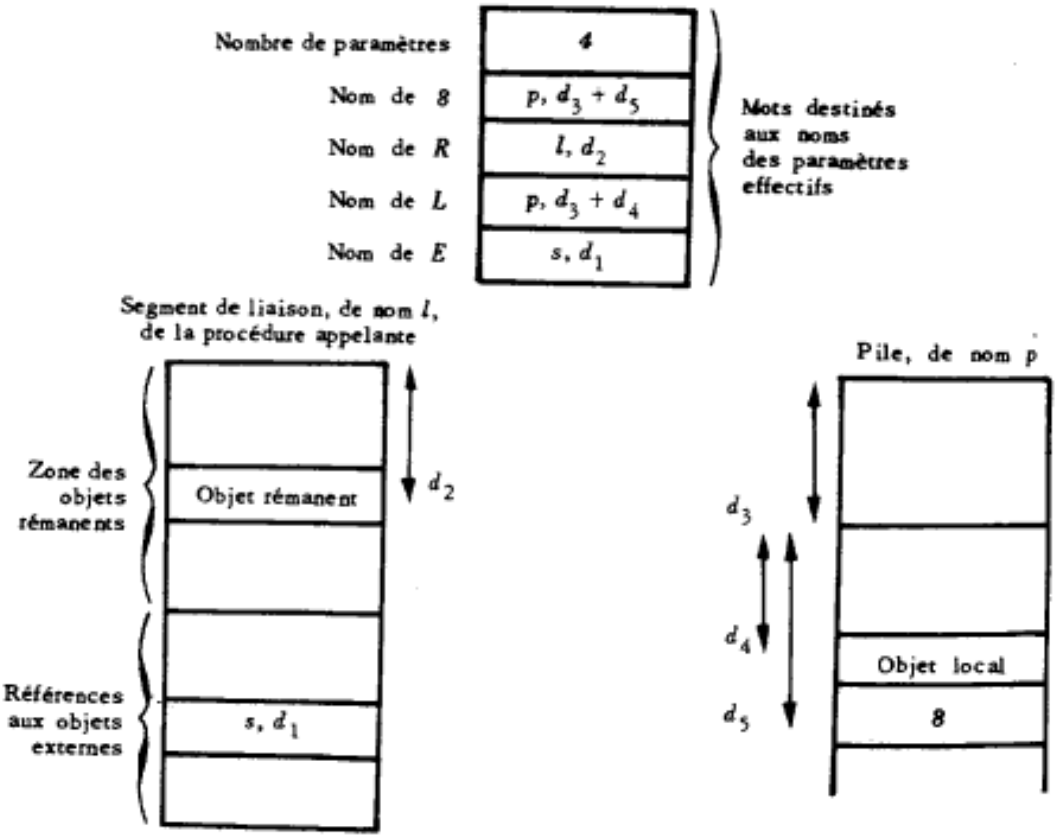


Figure 7. Accès aux paramètres.

Donnons à ces mots les noms *0, 1, 2, 3* et *4* et considérons un registre général, *reg*, appelé registre pointeur de paramètres, dont le contenu est le nom de la base du quintuplet. Dans la procédure appelée, le nom d'un paramètre formel est celui d'un mot qui contiendra à l'exécution le nom, calculé dynamiquement, du paramètre effectif. Ce double repérage nécessite que l'accès à tout paramètre formel se fasse par indirection. La procédure appelée utilise

$$*\,(ic, reg, i)$$

où i vaut *1, 2, 3* ou *4* selon le paramètre référencé. Ainsi si F est l'identificateur d'un paramètre formel, l'instruction

$$F := 3$$

se traduit dans le segment-procédure en

```
charger la valeur, RT      3
stocker le registre, RT    * (ic, reg, i)
```

où *RT* est un registre auxiliaire.

Reste à trouver un emplacement pour le quintuplet. La procédure appelante le range dans la pile, immédiatement après sa propre région. Les informations nécessaires au retour dans la procédure appelante et les objets locaux de la procédure appelée sont empilés également.

Ainsi à tout appel de procédure est associé, dans la pile, une région qui comprend :

— la zone des paramètres,
— la zone de retour,
— la zone des objets locaux.

L'adressage des paramètres, comme l'adressage des objets locaux, pourrait donc s'effectuer grâce au registre *R3*. En fait CLICS, comme MULTICS (cf. 5.2) est doté d'un mécanisme de 8 anneaux de protection. Le contrôle des accès dépend de l'anneau dans lequel se trouve la procédure et des protections associées au segment nommé. Une procédure peut changer d'anneau entre deux appels successifs. L'existence de ces anneaux de protection entraîne celle de :

— 8 piles par processus (car il y a 8 anneaux), appelées respectivement *0, 1, ..., 7*,
— un segment de liaison par anneau d'exécution de la procédure.

Comme une procédure appelante ne s'exécute pas forcément dans le même anneau que la procédure appelée, les paramètres n'appartiennent pas toujours à la même pile que les locaux de la procédure appelée. En conséquence, un registre supplémentaire, le registre *R5*, repère les paramètres tandis que le registre *R3* repère les locaux.

3.246 **Illustration des mécanismes d'accès**

Rappelons l'utilisation des registres du processus :

R0 : segment-procédure en cours d'exécution,
R3 : base du dernier bloc empilé,
R4 : segment de liaison associé à la procédure,
R5 : base du dernier groupe de paramètres empilés.

et la façon de faire référence aux objets :

— objet externe : lecture/écriture appel	* *(ip, R4, d)*
— objet rémanent : lecture/écriture	*(ip, R4, d)*
— objet local : lecture/écriture	*(ic, R3, d)*
— paramètre : lecture/écriture	* *(ic, R5, d)*
— instruction : se brancher à	*(ip, R0, d)*.

Dans l'illustration suivante (Fig. 8), nous représentons deux processus P_i et P_j en train d'exécuter un segment procédure S dans lequel figure tout l'éventail des types d'adressage. Nous avons arbitrairement supposé que P_i et P_j exécutent S dans le même anneau que la procédure qui l'appelle. (P_i dans l'anneau *1*, P_j dans l'anneau *4*.)

Pour lire le schéma, il est conseillé de considérer l'une après l'autre les cinq références qu'on trouve dans les instructions composant le segment partagé S, et pour chacune, de suivre tout le cheminement d'adressage conduisant à l'emplacement où se trouve l'objet référencé.

Chaque fois que les noms n'ont aucune raison d'être identiques pour P_i et P_j, nous les illustrons sous forme d'exemples numériques différents.

Remarque. Nous pouvons indiquer maintenant comment se fait le calcul du nom d'un paramètre qui correspond à un paramètre de la procédure appelante : c'est une adresse segmentée *(q, d5 + d6)* déduite du couple *(R5, d5)* et du contenu *(q, d6)* du registre *R5*.

3.25 APPEL ET RETOUR DE PROCÉDURE

Les mécanismes étudiés dans les paragraphes précédents permettent de résoudre les problèmes mis en jeu par l'appel et le retour d'une procédure :

— calcul de l'adresse segmentée des paramètres effectifs,
— sauvegarde du contexte de la procédure appelante,
— appel proprement dit,
— chargement du nouveau contexte,
— retour vers la procédure appelante,
— restauration du contexte de la procédure appelante.

Nous développons une solution possible à l'aide d'un exemple, celui de l'appel de la procédure

P(8, X, Y)

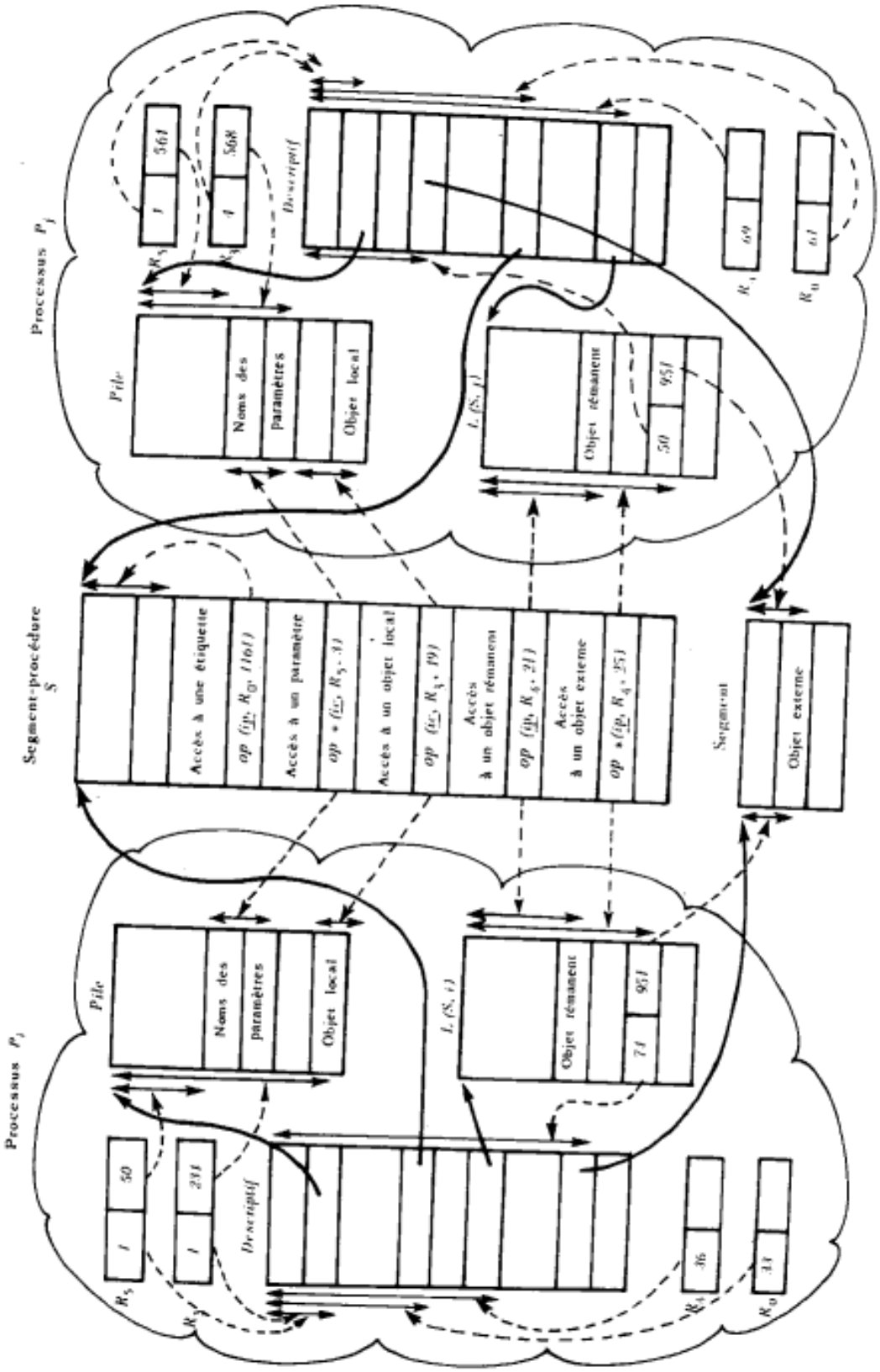


Figure 8. Illustration des mécanismes d'accès.

où *X* désigne le 13-ième objet rémanent de la procédure appelante et *Y* un objet externe désigné par le 20-ième mot du segment de liaison. Nous supposons que la procédure appelée et la procédure appelante ne s'exécutent pas dans le même anneau et qu'elles utilisent donc des piles différentes.

3.251 Calcul de l'adresse segmentée des paramètres effectifs

Le calcul des adresses segmentées des paramètres effectifs est fait par la procédure appelante. Ces adresses et un mot indiquant leur nombre sont rangés dans la pile de la procédure appelante, à la suite des objets locaux.

Supposons que les objets locaux occupent 14 mots de la pile. On range *8* dans le 15-ième mot. En désignant par *RT* un registre de travail, il y a successivement :

a) Rangement de *8*,

chargement immédiat, RT	*8*
rangement du registre, RT	*(ic, R3, 15)*

b) Rangement du nombre d'arguments :

chargement immédiat, RT	*3*
rangement du registre, RT	*(ic, R3, 16)*

c) Calcul du nom du premier paramètre. On utilise l'instruction *calcul adresse*, qui évalue l'adresse segmentée du champ opérande

calcul adresse, RT	*(ic, R3, 15)*
rangement du registre, RT	*(ic, R3, 17)*

d) Calcul du nom *X* du second paramètre

calcul adresse, RT	*(ip, R4, 13)*
rangement du registre, RT	*(ic, R3, 18)*

e) Calcul du nom *Y* du troisième paramètre

chargement mot, RT	*(ip, R4, 20)*
rangement du registre, RT	*(ic, R3, 19)*

3.252 Appel de procédure et changement de contexte

Seuls les contenus des registres *R0* et *R3* sont sauvegardés par l'exécution de l'instruction *appel de procédure*. Les autres registres doivent être sauvegardés par la procédure appelante.

a) Registres sauvegardés par la procédure appelante

Le contenu des registres à sauvegarder est rangé dans la pile. Si la procédure appelante a un segment de liaison, le contenu du registre *R4* doit être sauvegardé. Si elle a des paramètres, le contenu du registre *R5* doit être préservé. Dans tous les cas cette sauvegarde des registres est prévue dès la compilation.

Supposons par exemple qu'on doive sauvegarder les registres *R4*, *R5* et *R12* ; cela s'écrit :

rangement du registre, R4 *(ic, R3, 20)*
rangement du registre, R5 *(ic, R3, 21)*
rangement du registre, R12 *(ic, R3, 22)*

b) Mise à jour de l'adresse segmentée de sommet de pile

L'adresse segmentée du sommet de la pile est contenue dans le premier mot de la pile. Cette mise à jour se fait comme suit :

calcul adresse, RT *(ic, R3, 23)*
rangement du registre, RT *(ip, R3, 0).*

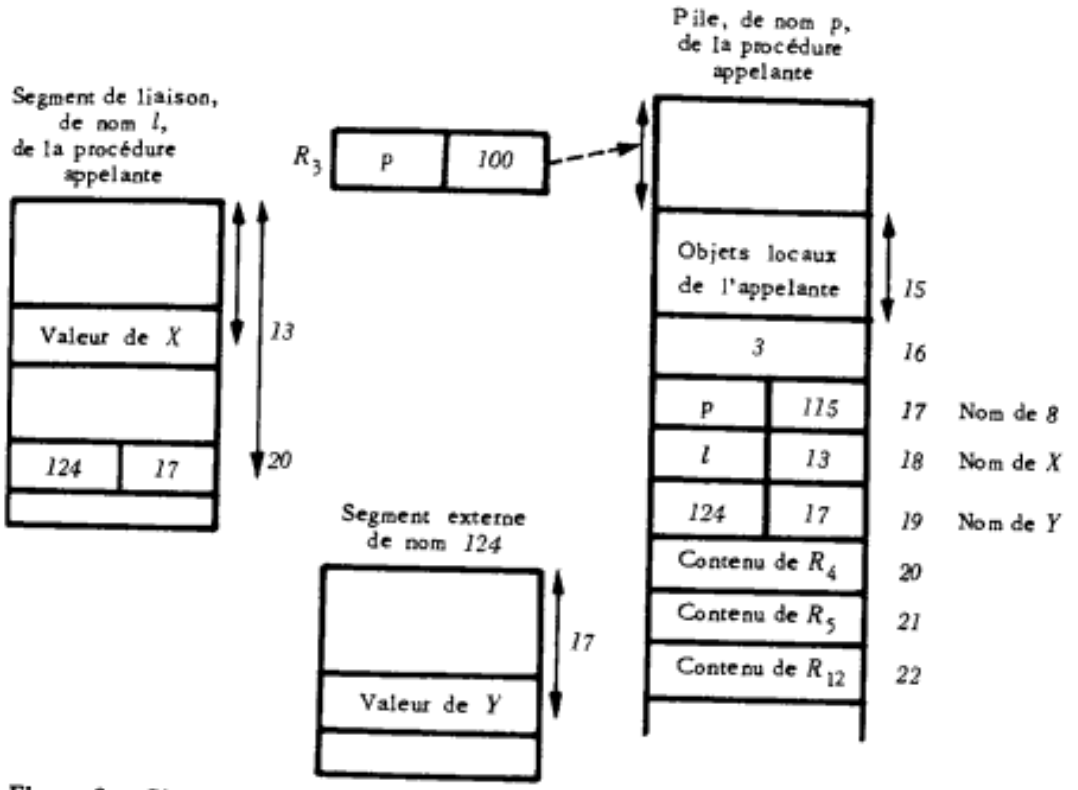


Figure 9. Situation avant l'exécution de l'instruction « appel de la procédure ».

c) Préparation du registre *R5* pointeur de paramètres pour la procédure appelée

calcul adresse, R5 *(ic, R3, 16)*

d) Exécution de l'instruction d'appel de procédure

L'instruction d'appel provoque le changement du contenu des registres *R0* et *R3*. Les anciens contenus sont rangés dans la pile associée à l'anneau d'exécution de la procédure appelée. Le registre *R0* reçoit l'adresse segmentée du point d'entrée de la procédure appelée. Le registre *R3* reçoit l'adresse segmentée

du sommet de pile de la procédure appelée, adresse qui, par construction, figure dans le premier mot de la pile.

Remarque. Le numéro de la pile utilisée par la procédure appelée est calculé par les mécanismes de protection. Nous n'examinons pas ici comment il est obtenu.

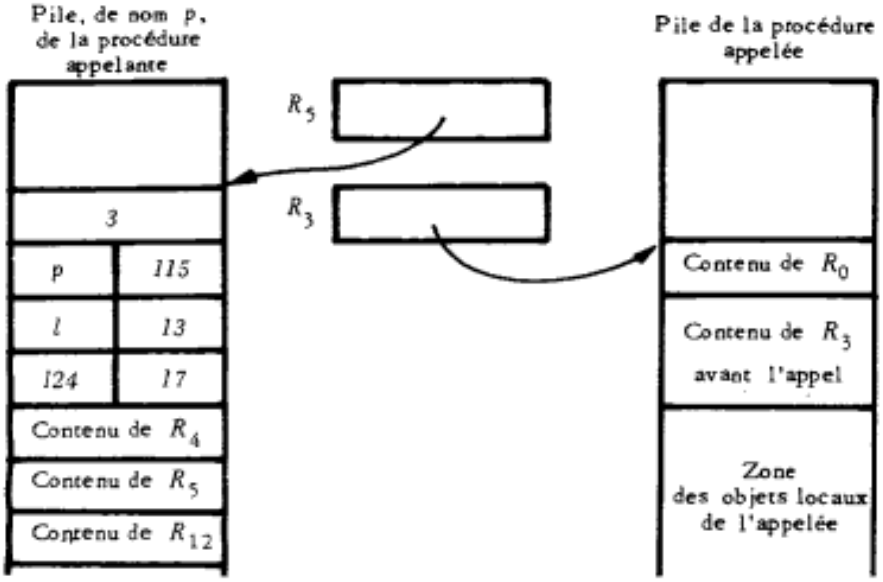


Figure 10. Situation après l'instruction d'appel de procédure.

e) Réservation de la zone des objets locaux à la procédure appelée

La réservation de la zone des objets locaux à la procédure appelée est obtenue en augmentant l'adresse segmentée du sommet de pile qui est contenue dans le premier mot de la pile. La taille *t* de cette zone est calculée à la compilation pour contenir *R0, R3* et les variables locales de la procédure. Il vient :

calcul adresse, RT	*(ic, R3, t)*
rangement du registre, RT	*(ip, R3, 0)*

f) Initialisation du registre pointeur du segment de liaison

Pour chaque anneau de protection, une table contient les numéros de segment de liaison des procédures. Cette table est repérée par le second mot de la pile associée à l'anneau. Elle fournit le contenu du registre *R4*, pointeur du segment de liaison.

3.253 **Retour à la procédure appelante**

a) Libération de la région des objets locaux à la procédure appelée

Pour détruire les objets locaux de la procédure appelée, la valeur du sommet de pile contenue dans le premier mot de la pile est modifiée. Sa nouvelle valeur est lue dans le registre *R3* qui repère la base de la région des objets locaux à détruire.

Il vient :

rangement du registre, R3 *(ip, R3, 0)*

b) Exécution de l'instruction de retour

L'instruction

retour procédure *(ic, R3, 0)*

charge les registres *R0* et *R3* avec les valeurs rangées en tête de la zone des objets locaux de la procédure appelée (cf. 3.252*d*).

c) Restauration des registres de la procédure appelante

R3 repère désormais la pile de la procédure appelante qui peut restaurer elle-même ses autres registres. Notre exemple donne :

chargement mot, R4 *(ic, R3, 20)*
chargement mot, R5 *(ic, R3, 21)*
chargement mot, R12 *(ic, R3, 22)*

3.26 LIAISONS DYNAMIQUES

Nous examinons maintenant un aspect fondamental de CLICS, en précisant les techniques mises en œuvre pour :

— assurer le partage des informations exécutables ou non,

— permettre l'écriture séparée de sections de code ou de données compilées indépendamment,

— gérer les objets de taille variable.

Il existe des langages de programmation (ALGOL 60, PL/1 ou ALGOL 68) où certaines liaisons entre segments ne peuvent être mises en place qu'à l'exécution. Ceci est dû au fait que ces langages permettent de traiter des objets de taille variable, des procédures récursives ou encore des objets créés et détruits dynamiquement.

D'autres liaisons qui sont mises en place dès la compilation dans des systèmes classiques, ne peuvent plus l'être dans CLICS à cause des contraintes de modularité (procédures compilées indépendamment), de partage et de protection (problèmes liés à l'accès partagé d'informations). De plus CLICS permet l'accès aux collections de données, considérées comme des segments.

Nous laissons de côté la traduction des identificateurs d'objets locaux, rémanents ou paramètres, car elle se fait au stade de la compilation ; par contre nous étudions l'identification des objets externes et les mécanismes mis en œuvre pour leur substituer un nom.

Nous supposons, dans ce qui suit, que S est un segment-procédure dont une instruction opère sur un objet externe à S situé dans le segment S'. Le programmeur ne peut pas désigner S' par un nom de segment puisque ce nom diffère d'un processus à l'autre. Le segment est alors désigné, sans ambiguïté dans le système, par un identificateur symbolique unique.

3.261 Remplacement de l'identificateur par un nom de segment : édition de liens

Soit *TOUAMOTOU* l'identificateur utilisé pour cataloguer S' à sa création. Cette création peut être postérieure au début de l'exécution du segment S qui le référence.

Le programmeur désigne le segment S' par *TOUAMOTOU*. Lorsqu'un processus P y fait référence c'est par un nom s', propre à P et à son descriptif. Examinons quand et comment est réalisée l'édition de liens qui fait passer de *TOUAMOTOU* à s'.

Comme cette édition ne peut être faite par le compilateur, celui-ci range la chaîne de caractères « *TOUAMOTOU* » dans une zone réservée de S.

On peut imaginer tout d'abord que l'édition de liens s'effectue au moment de l'adjonction de S dans le descriptif de P et avant le commencement de l'exécution de S. Il suffirait en effet de parcourir la liste des références externes de S et d'inclure dans le descriptif de P tous les segments nécessaires, c'est-à-dire tous ceux qui sont cités par S et qui ne figurent pas déjà dans le descriptif.

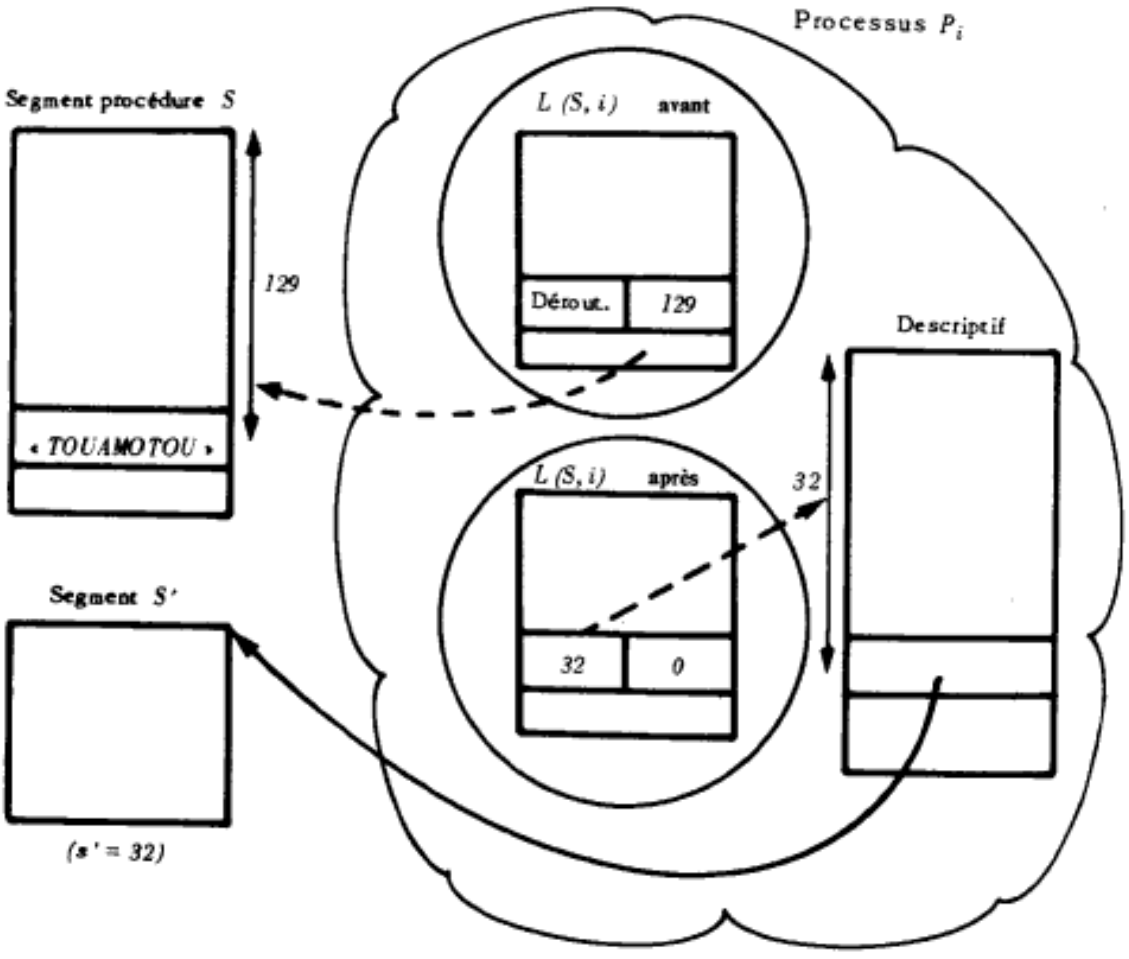


Figure 11. Effet de l'édition de liens dynamique.

Cette solution présente les inconvénients suivants :

— il peut se révéler inutile d'inclure systématiquement dans le descriptif tous les segments cités dans une procédure. En effet certains d'entre eux peuvent ne pas être utilisés à l'exécution,

— elle impose que tous les segments nommés par une procédure existent au moment de son appel. D'une part ce n'est pas nécessaire si S n'utilise pas S' lors de cette exécution particulière. D'autre part ce n'est pas possible si S' doit être créé par P ou par un autre processus avant d'être nommé dans S.

La solution adoptée dans CLICS consiste à faire la substitution de *TOUAMOTOU* par s' lors de la première référence à S' par S. Il s'agit donc d'une édition de liens dynamique. C'est le processeur qui doit détecter que le mot n'est pas encore lié. Pour cela un indicateur de déroutement est prévu dans le format des mots-machine. Quand il est présent, il se produit un déroutement, qui appelle l'éditeur de liens, et efface l'indicateur. Au retour de ce déroutement, l'instruction nommant S' est exécutée à nouveau.

L'état de l'indicateur de déroutement dépend du processus qui exécute le segment-procédure S et non pas de S. En effet comme S peut être partagé entre deux processus P_i et P_j, à un instant donné la substitution de *TOUAMOTOU* peut être déjà faite pour P_j (et le nom en est s''), mais non pour P_i. Pour cette raison l'indicateur de déroutement doit figurer dans le segment de liaison $L(S, i)$. Par ailleurs, ce n'est pas la chaîne de caractères « *TOUAMOTOU* » mais son adresse dans le segment S qui figure dans le segment de liaison et qui est associée à l'indicateur de déroutement. On fait référence à S' par indirection via le segment de liaison.

3.262 Référence à un segment-procédure

Soit un segment S faisant référence à un segment-procédure S'. Au premier appel de S' par la procédure S exécutée par le processus P_i, l'éditeur de liens est mis en œuvre. S' est alors connu par la chaîne de caractères « *TOUAMOTOU* » accessible par le segment de liaison $L(S, i)$. Si c'est le premier appel de S' par le processus P_i, l'éditeur de liens inclut S' dans le descriptif du processus P_i, créé et initialise un segment de liaison $L(S', i)$. Rappelons que $L(S', i)$ contient :

— les objets rémanents de la procédure S',

— les mots de liaison pour les références externes faites par S', que ces références soient à des données ou à des procédures.

L'état initial de $L(S', i)$, identique pour tous les processus, figure dans un modèle, présent dans le segment-procédure de S'.

Remarque 1. Lorsqu'il y a plusieurs points d'entrée, chacun d'eux reçoit un identificateur. Ceux-ci sont conservés dans une table rangée dans le segment S'. Toute référence symbolique à l'un de ces points d'entrée est préfixée par l'identificateur *TOUAMOTOU* du segment S'. L'éditeur de liens vérifie que le point d'entrée demandé figure bien dans la table des points d'entrée de S'.

Remarque *2.* Nous n'avons pas tenu compte des anneaux de protection, dont la présence complique les mécanismes indiqués. En particulier il y a autant de segments de liaison que d'anneaux de protection de la procédure S'.

Il en résulte qu'au premier appel de S' par le processus P_i dans l'anneau a, l'éditeur de liens peut découvrir (en utilisant la table accessible par le 2-ème mot de la pile de l'anneau correspondant, cf 3.252) que le segment S' est déjà présent dans le descriptif de P_i à la suite d'un appel de S' par P_i dans un anneau différent de a. Il doit alors créer un nouveau segment de liaison associé cette fois à l'anneau a.

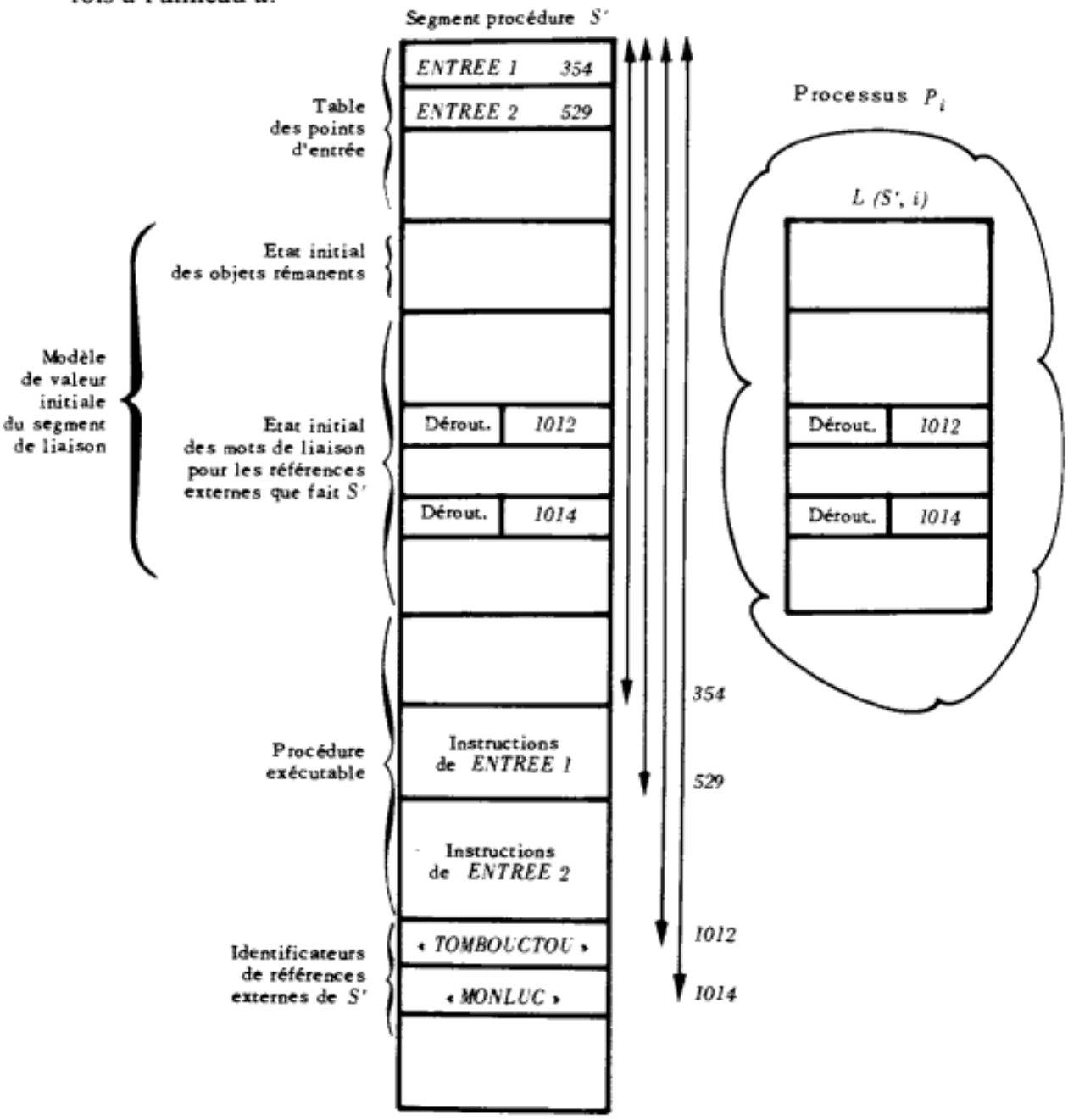


Figure 12. Image du segment de liaison à sa création.

Il en résulte aussi que les rémanents de CLICS ne peuvent, sans précaution, être utilisés comme des rémanents d'ALGOL 60. Ils servent surtout à conserver, en dehors de la pile, des variables « locales à une procédure » (comme en FORTRAN).

3.263 Catalogue des segments connus et catalogue général

Pour savoir si le segment *TOUAMOTOU* figure dans le descriptif du processus P_i, l'éditeur de liens consulte un catalogue donnant pour chaque segment accessible à P_i (c'est-à-dire dont le nom est présent dans le descriptif), l'identificateur associé. Ce catalogue, appelé catalogue des segments connus, est un segment particulier du processus P_i.

Lorsque *TOUAMOTOU* ne figure pas dans le catalogue des segments connus, il est recherché dans le catalogue général, unique dans le système, connu de tous les processus sous le même nom et qui contient les caractéristiques de tous les segments catalogués (identificateur, adresse de base dans la mémoire absolue, longueur, liste des contrôles et accès, etc...).

3.264 Gestion du descriptif

Rappelons que CLICS comporte un nombre fixe de processus et que chacun d'eux peut être associé à un utilisateur pour la durée d'une session.

Avant d'être associé à un utilisateur, le descriptif d'un processus P_i contient un petit nombre s_0 de noms de segments désignant, entre autres :

— les piles,
— le catalogue général et le catalogue des segments connus,
— l'éditeur de liens et les segments-procédure du moniteur,
— la procédure de gestion du terminal associé au processus.

Chaque segment S référencée par P_i pendant la session reçoit un nom lors de sa première référence et le garde pendant toute la durée de la session même si :

— le segment S n'est plus utilisé par P_i,
— le segment S est détruit.

Les noms de segments sont pris en séquence après s_0.

Pour éviter d'utiliser un segment détruit, un indicateur de déroutement est prévu dans chaque descripteur. Lorsque cet indicateur est présent, toute référence au segment entraîne un déroutement qui provoque l'appel d'une procédure du moniteur.

Le catalogue général des segments contient, pour chaque segment catalogué, la liste des processus qui le désignent par un nom et le nom donné par chaque processus. Lorsqu'un segment est détruit, cette liste permet de retrouver tous les processus qui l'utilisent et de placer l'indicateur de déroutement dans tous les descripteurs.

En fin de session d'un utilisateur, tous les segments de nom supérieur ou égal à s_0 sont enlevés du descriptif de P_i et les listes qui leur sont associées dans le catalogue général sont mises à jour.

3.3 GESTION DES NOMS DANS LE SYSTÈME BURROUGHS B6700

3.31 INTRODUCTION

Le système BURROUGHS B6500/6700 [Hauck, 68 ; Cleary, 69 ; Creech, 71 ; Organick, 71] possède les caractéristiques suivantes :

— marquage de tous les éléments de la mémoire par un préfixe,
— utilisation systématique de piles,
— partage des objets par le partage de leurs noms,
— création et destruction dynamique de processus.

Contrairement à CLICS, le système BURROUGHS B6700 ne permet pas l'édition de liens dynamique.

Le moniteur est composé d'un ensemble de procédures partagées connues de tous les processus.

Dans ce chapitre, nous présentons la pile associée à un processus et les objets qu'elle contient, les noms qui les désignent, leur partage entre processus grâce à une pile arborescente et l'utilisation de celle-ci pour la gestion de l'ensemble du système.

3.32 LE MATÉRIEL

3.321 **Notion de préfixe**

Dans le B6700, les premiers bits de chaque emplacement sont réservés pour indiquer la nature de l'objet représenté dans le reste de l'emplacement. Ils forment le préfixe de l'emplacement. Le préfixe peut indiquer, entre autres, une valeur, un nom ou un descripteur.

L'utilisation systématique des préfixes a des incidences :

— *sur l'adressage* : quand le processeur accède à un emplacement que son préfixe signale comme contenant un nom, alors que l'instruction nécessite une valeur, il effectue automatiquement un accès supplémentaire à l'emplacement désigné par le nom. Cette indirection automatique est répétée jusqu'à la rencontre d'un préfixe signalant une valeur.

— *sur la protection* : quand, après d'éventuelles indirections, le préfixe de la valeur n'est pas compatible avec le code opération de l'instruction, une erreur est signalée. On dispose ainsi d'une protection au niveau de l'emplacement.

— *sur le nombre des codes-opérations du langage de la machine* : les opérations effectuées dépendent du type des opérandes. Ceux-ci peuvent parfois ne pas être identiques : par exemple un opérande peut être en simple précision, l'autre en double précision. Il existe un code unique pour une opération donnée (par exemple l'addition). L'opération est complètement définie par le type de ses opérandes.

— *sur la compilation* : les compilateurs sont simplifiés, car ils n'ont pas à engendrer des ordres d'indirection et de conversion de type.

3.322 Les segments

La mémoire physique se compose d'une mémoire centrale de N emplacements de 51 bits ($N \leqslant 2^{20}$) et d'un disque ; elle est allouée par zones (cf. 4.43) à des segments, qui peuvent être de deux types :

— les **segments de données**, qui sont des tableaux, des éléments de fichiers ou des piles,

— les **segments-procédures** qui contiennent des instructions exécutables.

Un segment est représenté par un descripteur de segment qui occupe un emplacement. Ce descripteur contient, entre autres, les informations suivantes :

— un préfixe (*sd* pour un segment de données, *sp* pour un segment-procédure),

— l'adresse en mémoire physique de la base du segment,

— la taille du segment.

La donnée d'un couple (descripteur de segment, déplacement) permet à un processeur de calculer une adresse en mémoire physique par addition du déplacement à l'adresse de base. Au cours de ce calcul d'adresse, le processeur vérifie que le déplacement n'est pas supérieur à la taille du segment et qu'il n'y a pas violation de protection.

Quand un segment est déplacé en mémoire, la ou les versions de son descripteur doivent être retrouvées et mises à jour ; nous ne considérons pas ce problème dans ce chapitre.

3.323 Les processeurs physiques

Une installation peut comporter plusieurs processeurs. Chacun d'eux exécute des objets-procédures dont les instructions sont écrites en notation postfixée et gère en pile son espace de travail. Cette technique, décrite dans [Randell, 64], permet de traiter simplement la structure de bloc et les appels récursifs de procédures. Elle comporte certains avantages :

— la génération, par certains compilateurs, d'instructions en notation postfixée est facile,

— la structure de bloc permet une définition précise de l'espace de travail (cf. 4.6) et en facilite la gestion,

— beaucoup d'instructions trouvent leurs opérandes sur le sommet de la pile et leurs noms peuvent alors être implicites.

Le langage utilisé pour programmer le système est un dérivé d'ALGOL 60 ; le système dispose aussi de compilateurs pour d'autres langages évolués courants.

La pile d'un processeur est contenue dans un segment, dont le descripteur est réparti entre deux registres : le registre **base de pile** qui contient l'adresse de base du segment et le registre **plafond de pile** qui contient l'adresse du dernier

mot du segment. Ces deux registres délimitent le volume total réservé pour la pile. A tout instant, l'adresse du sommet de la pile est contenue dans un registre **sommet de pile**, mis à jour par câblage à chaque opération d'empilement ou de désempilement. Le plafond détermine la position limite du sommet : une tentative de dépassement provoque un déroutement, qui entraîne le réajustement du plafond ou la destruction du processus. D'autres registres permettant d'accéder aux objets de la pile seront présentés en 3.34.

Les instructions sont rangées dans des segments-procédures, suites d'emplacements portant chacun un préfixe noté *inst*.

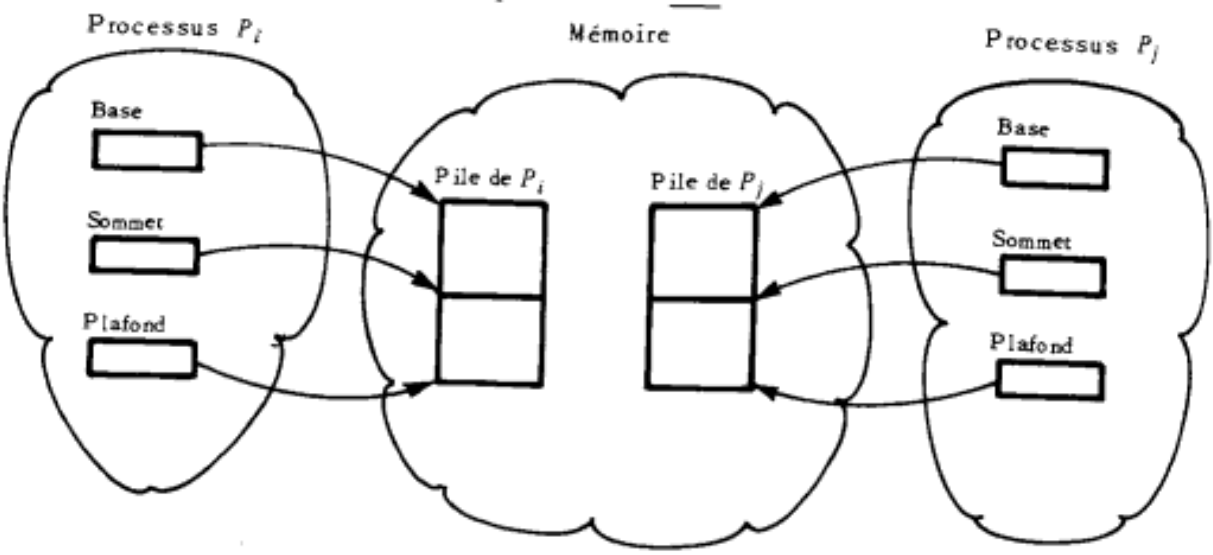


Figure 13. Pile d'un processus.

L'instruction qu'un processeur est en train d'exécuter est désignée par le **registre pointeur d'instruction** qui, entre autres :

— contient le nom de l'emplacement où se trouve le descripteur du segment-procédure courant,

— indique le déplacement de l'instruction courante, mesuré en octets par rapport à l'origine du segment-procédure courant.

Les interruptions sont traitées comme des appels de procédure inattendus, forcés par câblage. La sauvegarde de l'environnement au moment de l'interruption et sa restauration, une fois l'interruption prise en compte, sont réglées par le schéma d'appel et de retour de procédure.

3.33 REPRÉSENTATION DES OBJETS DU LANGAGE

Nous considérons ici un processus isolé ; tous les objets qui lui sont accessibles sont donc représentés dans sa pile.

3.331 **Objets simples**

Nous supposons qu'il n'existe qu'un seul type d'objets simples (au lieu des divers types : entier, réel, simple ou double précision, ...). Un emplacement contenant un objet simple porte le préfixe *val*. Quand un objet est créé sans

valeur initiale, l'emplacement correspondant reçoit un préfixe spécial signifiant *valeur indéterminée*. Une tentative de lecture d'une telle valeur provoque un déroutement.

3.332 **Tableaux**

Un tableau à une dimension est représenté dans la pile par un descripteur de segment qui renvoie au corps du tableau proprement dit, lui-même extérieur à la pile. Cette méthode permet de représenter des tableaux dont la taille peut varier au cours de leur durée de vie (Fig. 14).

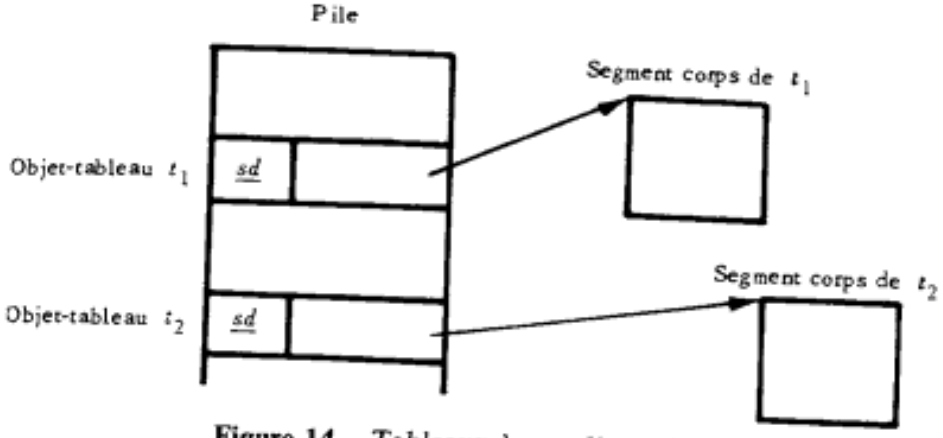


Figure 14. Tableaux à une dimension.

Un tableau à deux dimensions de taille $l \times m$ est représenté par le descripteur d'un segment de données de l mots, chacun d'eux contenant le descripteur d'un segment de données de m mots. Tous ces segments sont créés en même temps que le tableau.

Un tableau à n dimensions est représenté suivant le même principe. Un tableau peut donc être de taille variable selon une ou plusieurs de ses dimensions.

3.333 **Objets-procédures**

Chaque objet-procédure est conservé dans un segment-procédure et représenté par le descripteur de ce segment. On assimile tout bloc ALGOL 60 à une procédure et on lui associe donc un segment-procédure.

L'ensemble des descripteurs des segments-procédures utilisés par le processus, ou **descriptif,** est conservé dans la pile, dans une région dont la durée de vie est au moins égale à celle du processus.

3.34 ACCÈS AUX OBJETS

3.341 **Aspects lexicographiques**

On appelle **bloc** une suite d'instructions du texte source comprise entre les symboles *début* et *fin* et comportant des déclarations. Si on admet qu'un bloc peut contenir un autre bloc, la définition précédente doit être complétée

par une règle permettant d'associer par paires les symboles *début* et *fin* limitant un bloc. A cet effet, on définit un nombre k initialisé à -1 qui, lorsqu'on parcourt le texte source, augmente de *1* à chaque rencontre du symbole *début* et diminue de *1* à chaque rencontre du symbole *fin*. Dans un programme bien construit, k reste non négatif et sa valeur finale est -1 après rencontre du dernier symbole *fin*. Lorsqu'on pénètre dans un bloc après avoir franchi un *début*, la valeur courante de k est appelée **niveau d'emboîtement** du bloc ; le symbole *fin* qui termine le bloc est le premier symbole *fin* rencontré pour lequel la valeur courante de k est égale au niveau d'emboîtement du bloc. Dans le système BURROUGHS B6700, les niveaux *0* et *1* sont réservés au système ; le niveau d'emboîtement du bloc le plus externe d'un texte-source, c'est-à-dire du programme lui-même, est donc égal à *2*.

Nous appelons **identificateurs d'un bloc** les identificateurs dont la déclaration figure dans ce bloc à un niveau égal au niveau d'emboîtement du bloc. Le **lexique d'un bloc** est l'ensemble des identificateurs du bloc et des blocs englobants.

3.342 L'espace adressable

Chaque exécution d'un bloc crée deux espaces d'emplacements :

1) Une **région**, de taille connue à la compilation. Elle contient une **zone de liaison** et la collection des objets créés par le traitement des déclarations des identificateurs du bloc. Nous disons que la région est **issue** du bloc. A la sortie du bloc, cette région est détruite et les emplacements qu'elle occupait sont libérés. Le **niveau d'emboîtement d'une région** est le niveau d'emboîtement du bloc dont elle est issue.

2) Une zone d'évaluation des instructions, gérée en pile. Les emplacements de cette zone sont créés et libérés au fur et à mesure de l'évaluation des instructions. A la sortie du bloc, cette zone est de taille nulle.

Par le jeu des exécutions de blocs imbriqués et des appels de procédures, il existe en général à un instant donné plusieurs régions. Leurs durées de vie étant imbriquées, l'ensemble de ces régions est géré en pile ; il forme l'**espace adressable** du processus. La dernière région créée est la **région courante.**

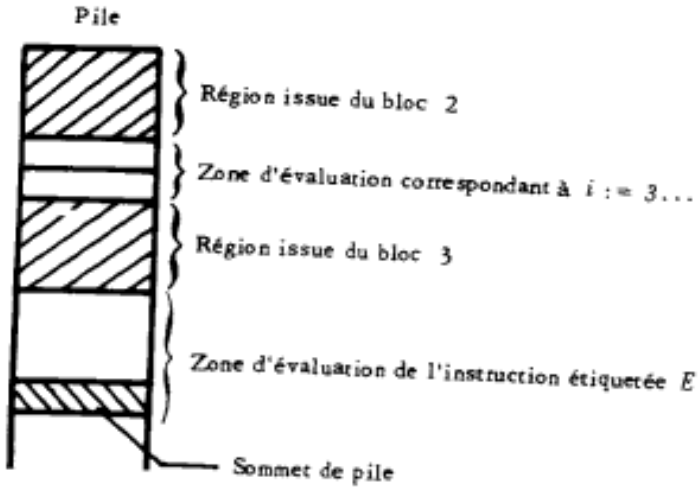
Remarque. Si une procédure est appelée récursivement, il existe à un moment donné plusieurs régions issues du même bloc.

La pile des régions et la pile des zones d'évaluation sont en fait fusionnées en une seule pile comme le montre l'exemple ci-dessous.

Exemple. Considérons le programme ALGOL 60 suivant

```
bloc 2 [ début entier i ; ...
         bloc 3 [ entier procédure p ; début entier j ; ... E : ... fin ;
                  i := 3 * p ...
         fin
```

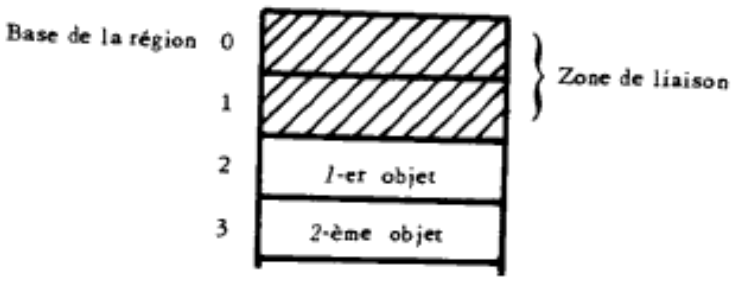
A l'exécution de l'instruction étiquetée *E*, la pile a la configuration suivante :



3.343 Environnement : accès par désignation

L'**environnement**, à un moment donné de l'exécution, est un ensemble de régions inclus dans l'espace adressable, tel qu'à tout identificateur (de paramètre formel ou non) appartenant au lexique du bloc en cours d'exécution correspond un objet de l'environnement. Lors d'une autre exécution du même bloc, le même identificateur désignera un autre objet, car l'environnement aura changé. Un identificateur désigne soit un objet du langage, soit un paramètre formel qui fournit l'information conduisant au paramètre effectif. L'environnement sera défini plus rigoureusement en 3.36 par la façon dont il varie à partir d'un état initial. Pour l'instant, constatons qu'il contient une région de chaque niveau d'emboîtement *i*, *i* allant de *0* jusqu'au niveau d'emboîtement de la région courante.

1) *Noms statiques* : le compilateur traduit chaque identificateur en un **nom statique** de la forme *(a, b)*, où *a* est le niveau d'emboîtement du bloc où est déclaré l'identificateur et *b* est un entier valant *k + 1* pour le *k*-ième objet déclaré dans le bloc. (Les emplacements *0* et *1* de la région étant systématiquement réservés à la zone de liaison, *b* doit commencer à la valeur *2*.)



2) *Registres d'environnement* : chaque processeur comporte une collection de **registres d'environnement** (« display registers ») qui contiennent une représentation de l'environnement à tout moment : le *n*-ième registre repère la base

de la région de niveau n de l'environnement ($0 \leqslant n \leqslant 31$ dans le système BURROUGHS B6700). Le contenu des registres d'environnement est mis à jour à chaque ouverture ou fermeture de bloc.

3) *Interprétation des noms statiques en fonction de l'environnement* : pour accéder à l'objet de nom statique (a, b), le processeur ajoute b au contenu du registre d'environnement de niveau a.

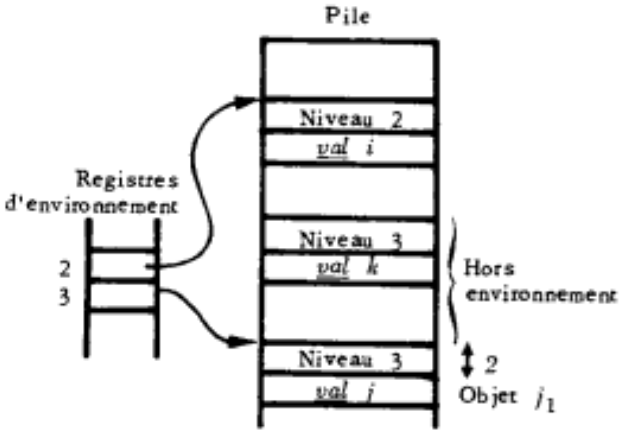
Exemple. Soit le programme ALGOL 60 :

```
début entier i ;
      procédure p ;
      début entier j ;
            i := i + 1 ; j := i ;
            si i < 3 alors p        ←— (2)
      fin ;
      début entier k ;
            i := 0 ; p              ←— (1)
      fin
fin
```

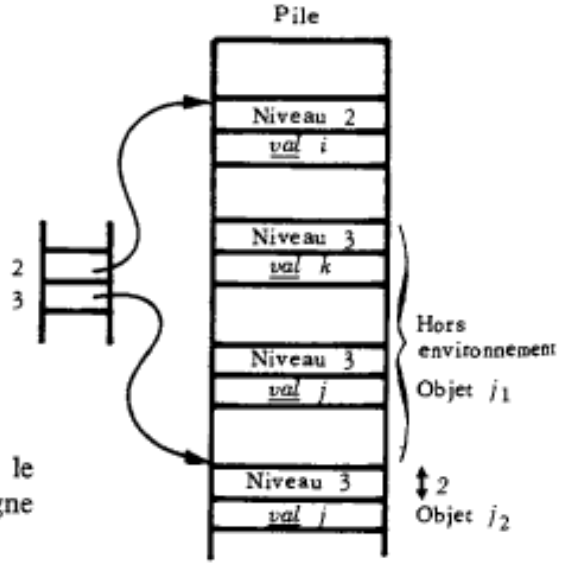
Après compilation, on a la correspondance suivante :

Identificateur	i	p	j	k
Nom statique	2,2	2,3	3,2	3,2

On remarque que j et k ont le même nom statique.



Après l'appel 1, à l'exécution de $j := i$, le nom statique $(3, 2)$ correspondant à j désigne l'objet j_1.



Après l'appel 2, l'environnement a changé et le même nom statique $(3, 2)$ désigne un autre objet, j_2.

3.344 Accès aux paramètres effectifs : noms dynamiques

L'identificateur d'un paramètre effectif ne peut figurer directement au sein du corps de la procédure appelée puisque le paramètre effectif varie en général d'un appel à l'autre. Seul est connu, à la compilation, le paramètre formel qui est traité comme un objet local à la procédure, à l'emplacement duquel on accède par désignation. L'information concernant le paramètre effectif ne peut être fournie qu'au moment de l'appel de la procédure. Le nom du paramètre effectif est alors rangé par la procédure appelante dans l'emplacement du paramètre formel.

L'interprétation de ce nom ne doit pas faire intervenir l'environnement car le paramètre effectif peut ne pas s'y trouver. Ce nom est un **nom dynamique** constitué d'un couple (b, d) où b est le déplacement, par rapport à la base de la pile, de la région où se trouve le paramètre effectif, et d est le déplacement du paramètre effectif par rapport à la base de cette région.

Un emplacement contenant un nom dynamique porte un préfixe noté *nd*. Pendant l'exécution de la procédure, chaque accès à l'emplacement du paramètre formel est automatiquement transformé, par le préfixe *nd*, en un accès indirect vers l'emplacement désigné par le nom dynamique.

Exemple. Dans le programme de la figure 15, pour exécuter $l := k$, le processus a besoin de la valeur de k ; mais k n'étant qu'un paramètre formel, il faut obtenir la valeur du paramètre effectif j, qui est hors de l'environnement au moment considéré. Pendant l'exécution de q, la pile a l'aspect illustré sur la figure 16.

```
début entier i ;
    procédure q(k) ; entier k ; début entier l ;
                                    ....
                                    l := k
                                fin
    début entier j ; ... ; q(j) ; ...
    fin
fin
```

Figure 15.

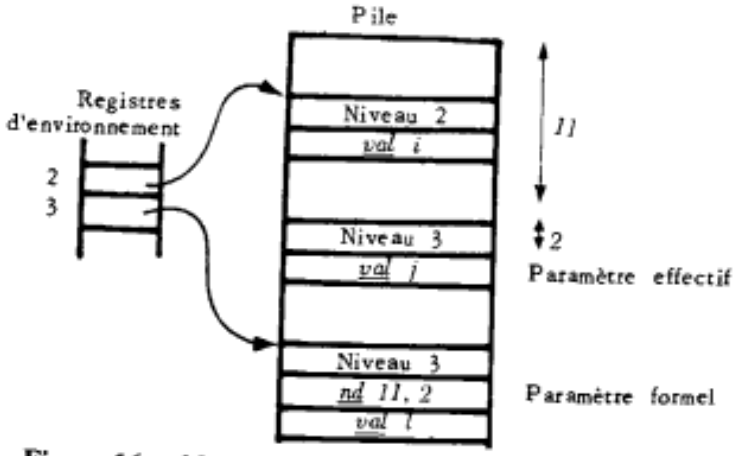


Figure 16. Nom dynamique d'un paramètre effectif.

Remarque. On verra dans le paragraphe suivant que c'est pour pouvoir traiter le cas des procédures passées en paramètres que le nom dynamique n'est pas simplement $b + d$.

3.35 PROCÉDURES

A chaque traitement d'une déclaration de procédure est créé un procédurus qui est l'association d'un objet-procédure *objproc* et d'un environnement *env*. ALGOL 60 précise que *env* est l'environnement courant au moment du traitement de la déclaration. Une même déclaration de procédure rencontrée plusieurs fois de suite par récursivité entraîne la création de plusieurs procédurus, qui ont tous la même composante *objproc* mais diffèrent par leur composante *env*. Quand un procédurus est créé, sa représentation est rangée dans la pile avec le préfixe *proc*.

a) La composante *objproc* figure dans l'emplacement du procédurus sous forme du nom statique de l'emplacement de la pile où est rangé le descripteur du segment-procédure contenant l'objet-procédure.

Exemple.

```
début procédure q ; début procédure p ; début ... fin ;
                                  procédure pp ; début ... fin ;
                                  q ;
                          fin ;
    q ;
fin
```

Après deux appels récursifs de q, on a la situation représentée sur la figure 17.

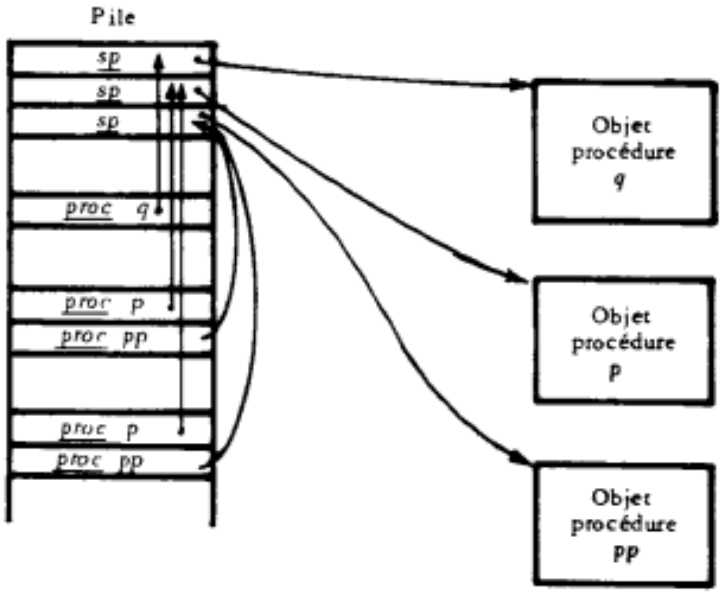


Figure 17. Procédurus et objet-procédure.

Remarque. Un emplacement préfixé *proc* pourrait contenir directement le descripteur de l'objet procédure, puisque l'édition de liens est effectuée avant l'exécution du programme. Mais cette solution conduirait à multiplier les descripteurs de segments et à compliquer leur recherche pour les mettre à jour en cas de déplacement des segments en mémoire physique.

b) La composante *env* du procédurus n'a pas besoin de figurer explicitement dans l'emplacement de chaque procédurus : en effet, tous les procédurus créés dans une région donnée ont la même composante *env*, qui est l'environnement associé à la région. Il suffit donc d'avoir, rangée en un seul exemplaire dans la zone de liaison de la région, l'information représentant *env*. Ce rôle est rempli par la chaîne statique, qui sera étudiée en 3.363. Cette économie a pour contrepartie une légère complication des noms dynamiques. Ils sont formés d'un couple dont les éléments désignent : l'un, la zone de liaison d'une région, l'autre un déplacement par rapport à cette zone.

Remarque. Cette généralisation du procédurus et la présence du préfixe de l'emplacement permettent d'utiliser les mêmes instructions de la machine pour désigner un objet par un nom ou par une procédure qui fournit l'objet. Seule la chaîne d'accès à l'objet change : la procédure est considérée comme une fonction d'accès à l'objet.

3.36 VARIATIONS D'ENVIRONNEMENT AUX APPELS ET RETOURS DE PROCÉDURES

L'environnement change à l'occasion des appels et des retours de procédures. Les entrées et les sorties de blocs en sont des cas particuliers que nous ne traitons pas.

Dans ce qui suit, on désigne par :

R(appelante) la région courante au moment de l'appel,
E(appelante) l'environnement associé à cette région,
R(appelée) la première région créée à la suite de l'appel.

3.361 Appel de procédure

Appelons **environnement initial** l'environnement auquel on ajoute *R(appelée)* pour former l'environnement pendant l'exécution du bloc le plus externe du corps de la procédure appelée. Par définition des langages à structure de bloc, l'environnement initial est constitué par la composante *env* (cf. 3.35) du procédurus appelé, que ce dernier ait été passé en paramètre ou non.

a) Si le procédurus appelé n'est pas un paramètre, *env* est forcément inclus dans *E(appelante)*. Plus précisément, si *n* est le niveau d'emboîtement de la région contenant le procédurus, *env* est constitué des régions de niveaux *0* à *n* de *E(appelante)*. Cette propriété rend très simple l'installation du nouvel environnement par le processeur : les registres d'environnement *0* à *n* n'ont

pas à être modifiés, le registre $n + 1$ désigne la nouvelle région *R(appelée)* et les contenus des registres $n + 2$, etc..., ne sont plus significatifs. On remarque qu'il n'est pas besoin d'accéder à la représentation du procédurus dans la pile : n est déduit de son nom statique et le nom de l'objet-procédure est connu dès la compilation.

Exemple. Soit le programme :

```
début procédure p ; début entier l ; ... fin ;
      début entier k ; ... ; p ; ... fin
fin
```

L'environnement avant et après l'appel de p sont illustrés sur la figure 18.

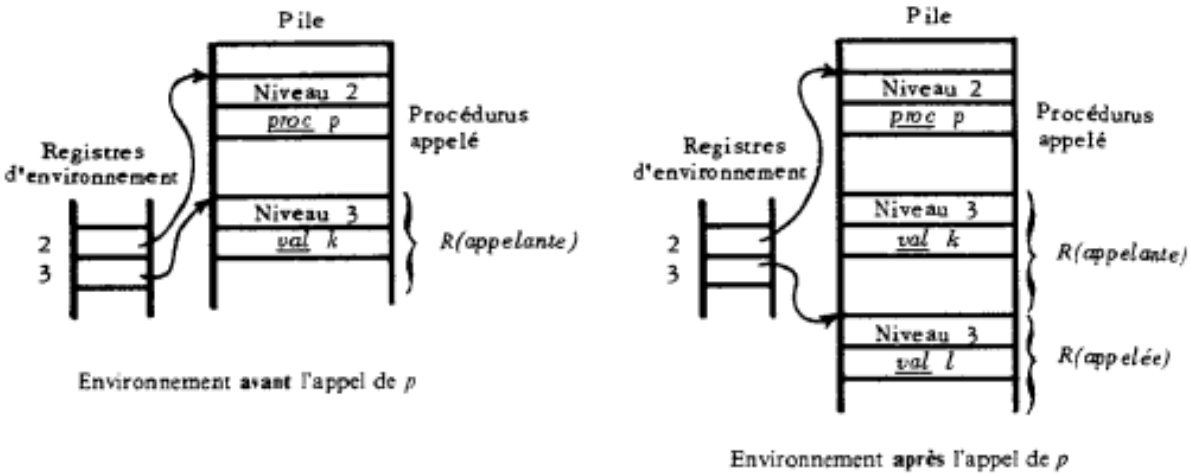


Figure 18. Appel d'une procédure qui n'a pas été passée en paramètre.

b) Si le procédurus appelé est un paramètre, le processeur doit aller chercher dans la pile les composantes *objproc* (cf. 3.35) et *env* du procédurus. On trouve *objproc* dans l'emplacement du procédurus, et *env* dans la zone de liaison de la région du procédurus.

En général, *env* n'est pas inclus dans *E(appelante)*, comme le montre l'exemple suivant :

Exemple. Soit le programme :

```
début procédure q( pp ) ; procédure pp ;
      début ... ; pp ; ... fin
      ...
      début procédure p ; début entier l ; ... fin ;
      q( p ) ;
      fin
fin
```

Au moment de son appel par la procédure q, le procédurus p passé comme paramètre effectif n'est pas dans l'environnement courant (Fig. 19).

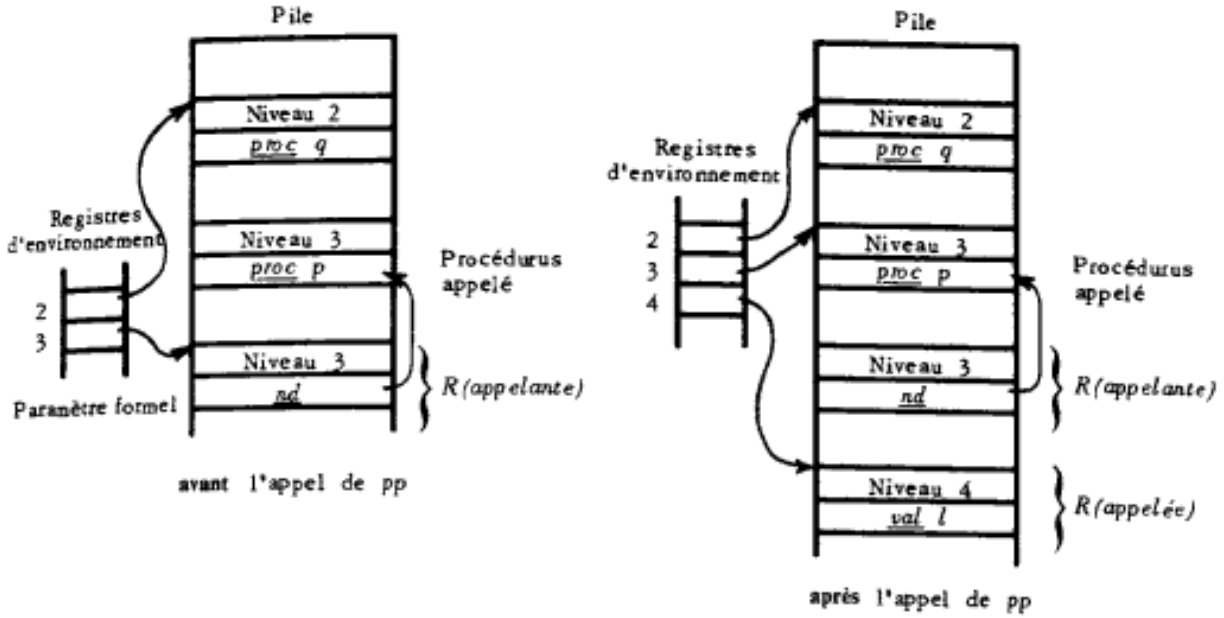


Figure 19. Appel d'une procédure passée en paramètre et qui n'est pas dans l'environnement.

3.362 **Retour de procédure**

Au retour d'une procédure, il faut rétablir *E(appelante)*. On dispose pour cela de la zone de liaison de *R(appelée)*. Une méthode immédiate serait d'y ranger le contenu des registres d'environnement significatifs de la procédure appelante ; mais, comme à chaque appel d'une procédure donnée l'environnement n'est pas toujours le même, les zones de liaison devraient avoir des tailles diverses, donc inconnues à la compilation. Aussi range-t-on dans *R(appelée)* uniquement un pointeur, appelé **élément de chaîne dynamique**, qui désigne la zone de liaison de *R(appelante)* ; c'est dans cette dernière que figure l'information permettant de reconstruire *E(appelante)*, sous une forme qui sera étudiée plus loin (chaîne statique).

Un registre propre à chaque processus, le **registre de région courante**, désigne constamment la base de la région courante ; son contenu fournit, à l'appel

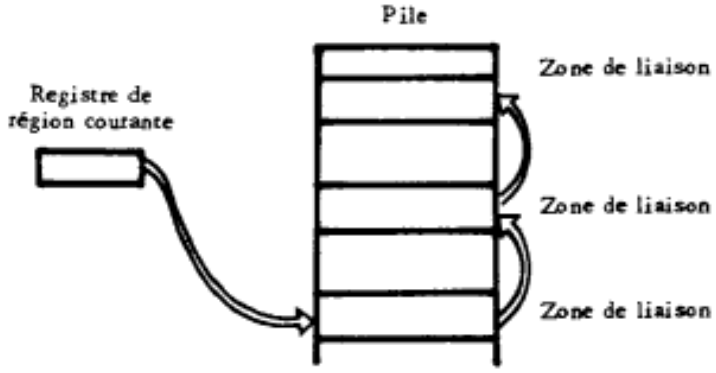


Figure 20. Chaîne dynamique.

d'une procédure, la valeur de l'élément de chaîne dynamique de la procédure appelée. La **chaîne dynamique** relie toutes les régions, dans l'ordre inverse de leur création (Fig. 20) ; elle ne sert qu'au retour de procédure (et donc en particulier à la sortie de bloc).

3.363 Chaîne statique

Nous avons vu deux circonstances dans lesquelles on doit retrouver l'environnement d'une région grâce à sa zone de liaison :

— à l'appel d'une procédure passée en paramètre,
— au retour d'une procédure.

Au lieu de recopier tous les registres d'environnement dans la zone de liaison, on n'en range qu'un seul, grâce à la propriété suivante des environnements, qui découle de la structure de bloc :

Soit R_n une région de niveau n. Appelons R_i la région qui appartient à l'environnement de R_n et qui est de niveau i. Il y en a une et une seule pour chaque i allant de 0 à n. Appelons $E(R_i)$ l'environnement associé à R_i. Nous constatons la propriété suivante :

$$E(R_i) = R_i \cup E(R_{i-1}), \qquad i = 1 \ldots n$$
$$E(R_0) = R_0$$

Supposons que la base de R_i soit connue. Il suffit, pour obtenir $E(R_i)$, d'obtenir la base de R_{i-1} et d'appliquer la formule de récurrence. La zone de liaison de R_i contient le nom de la base de R_{i-1}, sous forme d'un déplacement par rapport à la base de la pile.

La base de R_n est obtenue au départ :

— soit par la chaîne dynamique, en cas de retour de procédure,
— soit par le nom dynamique, dans le cas où une procédure est un paramètre.

La suite des noms qui figurent dans les zones de liaison et qui permettent de construire l'environnement associé à une région s'appelle la **chaîne statique.**

3.364 Zone de liaison

Le premier mot de la zone de liaison porte un préfixe noté $\underline{li}$ et contient

— le niveau d'emboîtement de la région,
— un élément de chaîne statique,
— un élément de chaîne dynamique.

Exemple. La figure 21 indique les chaînes statiques et dynamiques correspondant au programme de la figure 18.

3.365 Détail de l'appel de procédure

Nous étudions ici en détail les instructions correspondant à l'appel de procédure.

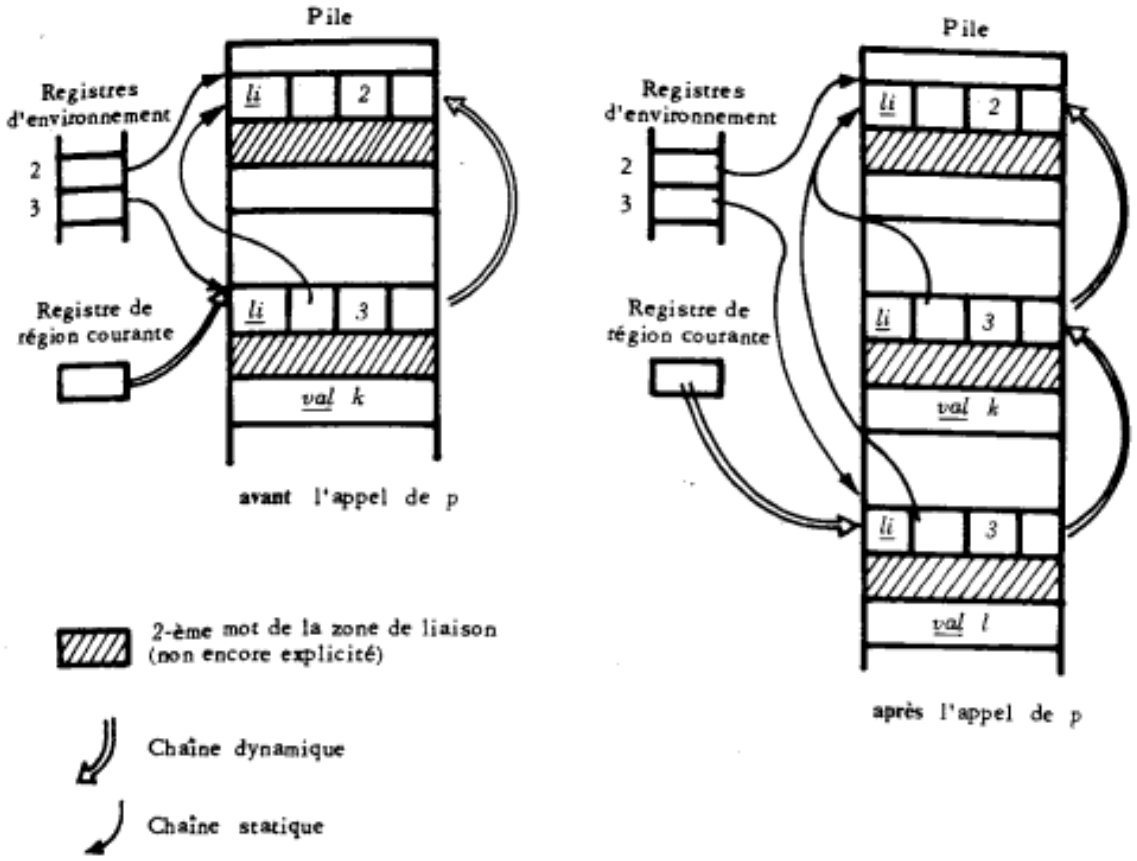


Figure 21. Chaîne statique et chaîne dynamique du programme de la figure 18.

1) Une première instruction réserve un mot en sommet de pile. Ce mot sera occupé par le premier mot de la zone de liaison de procédure à appeler.

2) L'instruction suivante empile temporairement le nom du procédurus à utiliser. Ce nom peut être un nom dynamique ; dans ce cas, il est recopié à partir de l'emplacement réservé au paramètre formel dans la région de la procédure-appelante.

3) Les paramètres effectifs sont ensuite évalués et leurs noms dynamiques empilés.

4) L'instruction d'appel proprement dite est exécutée. Si la procédure à appeler est dans l'environnement, son procédurus est désigné par un nom statique (n, d). Le registre d'environnement $n + 1$ reçoit l'adresse de base de la nouvelle zone de liaison. Les autres registres d'environnement restent valables et sont inchangés. Si la procédure est un paramètre, son procédurus est désigné par un nom dynamique (b, d). Dans la zone de liaison désignée par b, on trouve le niveau n de la région correspondante. Le registre d'environnement $n + 1$ reçoit l'adresse de base de la nouvelle zone de liaison. Les autres registres d'environnement n, $n - 1$, ..., reçoivent le contenu de la chaîne statique commençant en b. Dans tous les cas, le contenu des registres d'environnement de numéro supérieur à $n + 1$ n'est plus valide, et le programme n'y fait pas

référence. Le nombre $n + 1$ est rangé dans la nouvelle zone de liaison, avec le contenu du registre d'environnement n (chaîne statique) et le contenu du registre de région courante (chaîne dynamique). Le registre de région courante reçoit l'adresse de la base de la nouvelle région. Le contenu du registre pointeur d'instruction (cf. 3.322) est rangé dans le deuxième mot de la nouvelle zone de liaison avec un préfixe noté *ret*. Le nom de l'objet-procédure à exécuter est chargé dans le registre pointeur d'instruction. L'instruction suivante, désignée par ce registre, fait partie de la procédure appelée.

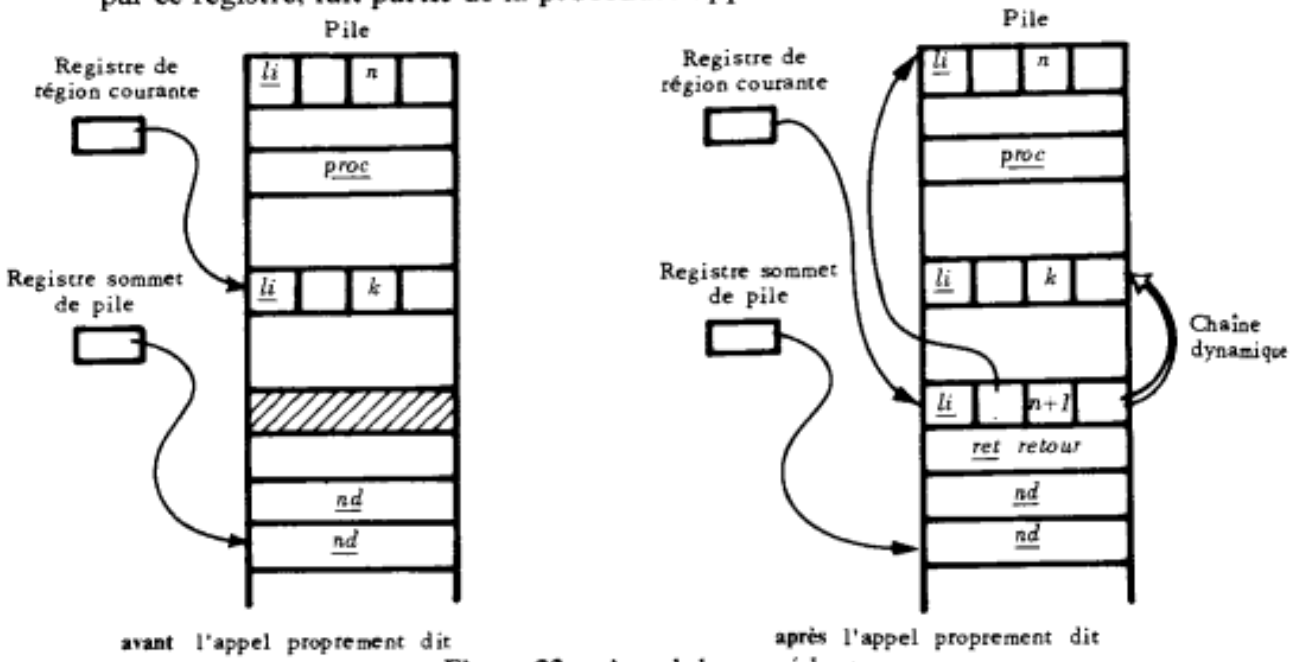


Figure 22. Appel de procédure.

3.37 PARTAGE DES OBJETS ENTRE UN PROCESSUS PÈRE ET SES PROCESSUS FILS

3.371 Création de processus

La création de processus se fait par un ordre d'activation sur une procédure.

Exemple.

```
début entier b ;
    processus truc ;
    procédure p(i) ; entier i ; début ... fin ;
    début entier m ;
        ...
        créer-activer truc sur p(m) ;
    fin
fin
```

Figure 23. Exemple de création de processus.

On utilise la terminologie **processus père** et **processus fils** pour évoquer le processus créateur et le processus créé.

Toutes les possibilités des appels de procédures sont permises lors de la création d'un processus sur une procédure : les paramètres sont autorisés, la procédure peut avoir été déclarée à n'importe quel niveau, elle peut être elle-même un paramètre.

A l'exécution, une création de processus diffère d'un appel de procédure par les points suivants :

— La nouvelle région (appelée) et les régions que pourra créer ultérieurement le processus fils, s'ouvrent non pas au sommet de la pile du père, mais dans une pile nouvelle, propre au fils.

— Un nouveau processeur virtuel est mis en activité. Ses registres reçoivent l'information permettant de gérer le nouveau processus ; en particulier son registre pointeur d'instruction reçoit le procédurus. Ce processus est lancé, et le processus père, dont les registres n'ont pas été modifiés, reprend immédiatement sa tâche.

Remarque. Le problème de l'allocation des processeurs réels est traité au chapitre 4.

3.372 Existence d'objets communs aux processus père et fils

La création d'un processus introduit une nouvelle source de création de régions issues d'un même bloc.

Soit p_i le processus père, p_j le processus fils créé sur la procédure S avec le procédurus t,
Q_i et Q_j leurs piles,
W_i et W_j les ensembles de régions qu'ils peuvent atteindre en dehors de leurs propres piles.

L'union des régions de Q_i et W_i forme l'**espace adressable** du processus p_i.

La structure de bloc implique les conséquences suivantes :

1) Soit $Z(i, j)$ l'ensemble des régions de Q_i qui existent au moment de la création de p_j. On appelle $Z(i, j)$ le **tronçon** de l'espace adressable de p_j dans Q_i. L'ensemble W_j est défini par

$$W_j = W_i \cup Z(i, j)$$

2) L'environnement initial du processus fils est composé des régions de W_j formant l'environnement associé à la région qui contient le procédurus t.

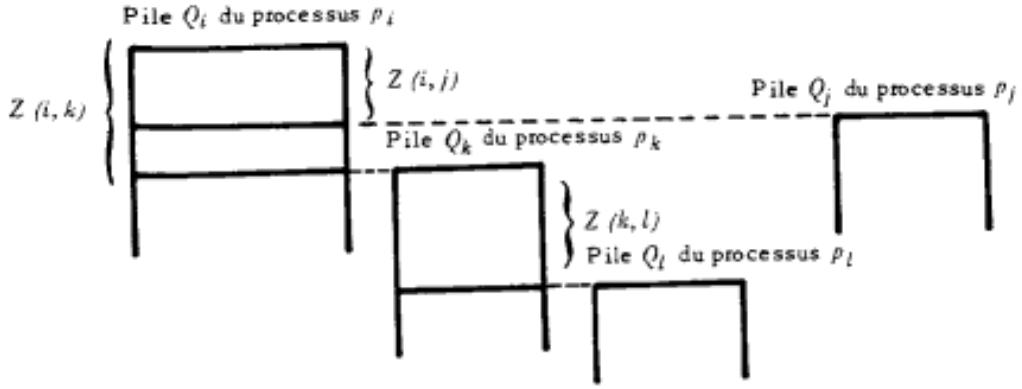
3) Un paramètre effectif de S est un objet quelconque de W_j.

4) Les objets créés par p_j sont tous dans Q_j.

5) L'environnement de p_j évolue à partir de l'environnement initial et des paramètres effectifs selon les règles édictées en 3.34, 3.35 et 3.36.

6) La fin du processus p_j se produit quand Q_j devient vide.

Ces nouvelles règles s'appliquent récursivement aux processus que le processus fils peut créer à son tour et aux processus ancêtres du processus père. Il existe un processus p_x tel que W_x soit vide. Les piles ainsi créées constituent une arborescence (Fig. 24).



p_i a créé p_j et p_k ; ce dernier a créé p_l. Le processus p_l peut avoir accès aux tronçons $Z(k, l)$ et $Z(i, k)$.

Figure 24. Arborescence de piles.

3.373 **Incidence sur les noms**

Pour tout processus le processeur doit gérer une pile et un espace adressable. La pile est gérée au moyen des registres base, sommet et plafond de pile décrits en 3.322. L'espace adressable et l'environnement, qui en est un sous-ensemble, étant composés de tronçons répartis dans diverses piles, le mode de désignation des objets doit en tenir compte.

a) Les éléments de la chaîne statique peuvent désigner des régions d'une autre pile. Le nom d'une région est alors un couple *(numéro de pile, déplacement de la région dans sa pile)*.

b) Les éléments de la chaîne dynamique d'un processus ne désignent jamais que des régions de sa pile. Quand la région ouverte à la création du processus se referme, le processus est détruit et il n'y a plus aucun retour à effectuer. Un élément de la chaîne dynamique désigne une région par son déplacement par rapport à la base de la pile.

c) Les noms statiques conservent la forme indiquée en 3.343. Ils sont interprétés en fonction du contenu des registres d'environnement.

d) Les noms dynamiques sont formés d'un triplet *(numéro de pile, déplacement de la région dans sa pile, déplacement de l'objet dans sa région)*.

e) Chaque pile est un segment. Les descripteurs de toutes les piles sont regroupés dans un **descriptif** de 1 024 éléments au plus. Le numéro de pile qui intervient dans les éléments de la chaîne statique et dans le nom dynamique est le déplacement, dans le descriptif, de l'emplacement où se trouve le descripteur de la pile considérée. Le descriptif est lui-même un segment dont le descripteur figure dans la pile de numéro zéro.

Remarque 1. Le double chainage :

— nom de la pile zéro dans le descriptif,
— nom du descriptif dans la pile zéro,

permet à l'allocateur de mémoire de déplacer le descriptif et la pile zéro.

La pile zéro est la pile du processus racine du système.

Remarque 2. On notera l'absence de protection câblée limitant l'accès d'une pile au seul tronçon autorisé au processus.

Exemple. Revenons au programme de la figure 23. Juste après l'instruction *créer-activer*, les processus père et fils sont dans l'état représenté sur la figure 25.

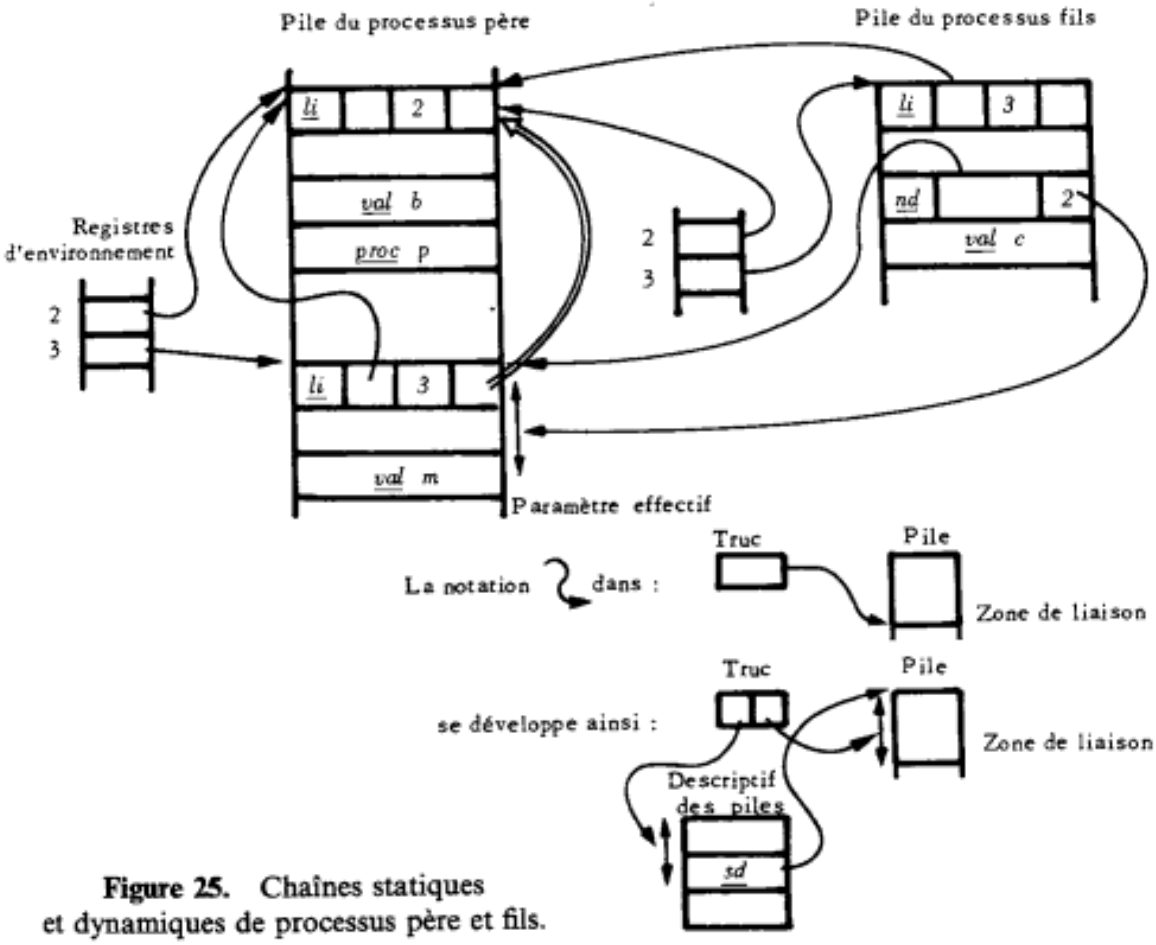


Figure 25. Chaînes statiques et dynamiques de processus père et fils.

3.374 Synchronisation

Les mécanismes fournis à l'utilisateur pour synchroniser ses processus parallèles ont été décrits en 2.63. La synchronisation peut être utilisée pour empêcher qu'un processus père ne détruise les régions de sa pile qui appartiennent à l'espace adressable d'un processus fils, alors que ce dernier n'est pas terminé. Le processus fils risquerait dans ce cas de désigner des objets disparus.

Par ailleurs, le système vérifie, à chaque fermeture de bloc, qu'il ne subsiste pas de processus qui ont été créés quand la région à détruire était courante (plus précisément les compilateurs engendrent à chaque fermeture de bloc un appel à une procédure du moniteur chargée de cette vérification). Cette protection programmée est la seule prévue.

3.38 INCLUSION DU MONITEUR DANS L'ARBORESCENCE DE PILES

3.381 **Partage des objets par une collection de processus**

Une collection C de processus se partage un objet si et seulement si celui-ci est dans l'espace adressable de chacun de ces processus. Etant donné la relation entre l'espace adressable d'un processus et celui de son père, ceci signifie qu'il doit exister un processus *p-créateur* qui a créé l'objet partagé, puis les processus de C (ou des ancêtres de ces processus).

Le processus *p-créateur* peut avoir pour seul rôle de créer les objets partagés puis les processus qui se les partagent. Comme la région créée pour les objets partagés doit avoir une durée de vie au moins égale à celle de tous les autres processus, le *p-créateur* est bloqué en attente de l'événement traduisant la fin de l'exécution de ceux-ci.

Exemple.

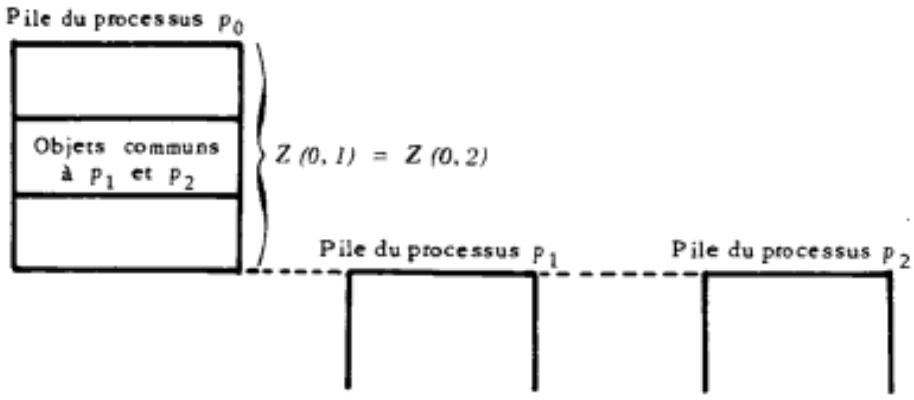


Figure 26. Objets mis en commun par un ancêtre commun.

Comme il est plus efficace de réaliser l'accès à un objet par désignation plutôt que par paramètre, on s'arrange pour que le procédurus sur lequel est créé chacun des processus ait tous les objets communs dans son environnement.

3.382 **Partage des objets communs à tous les processus**

Les procédures et les diverses tables qui constituent le moniteur sont considérées comme des objets partagés entre tous les processus.

Parmi les procédures, on peut citer :

— celles qui permettent à l'utilisateur de créer des segments-procédures à partir de textes-sources (compilateurs), de le cataloguer (gestion de fichiers) et de les nommer pour les exécuter,

— celles qui gèrent l'arborescence de processus,

— celle qui est appelée automatiquement après toute interruption afin d'en détecter la cause.

Nous avons déjà décrit une table du moniteur : le descriptif des piles.

Tous ces objets sont rangés dans la pile d'un **processus racine** dont la durée de vie est égale à celle du système et dont tous les autres processus sont les descendants. Cette pile ne contient jamais qu'une région de niveau d'emboitement zéro. Elle contient en particulier le descriptif des segments-procédures du moniteur et les procédurus associés à chacune des procédures (Fig. 27).

Exemple.

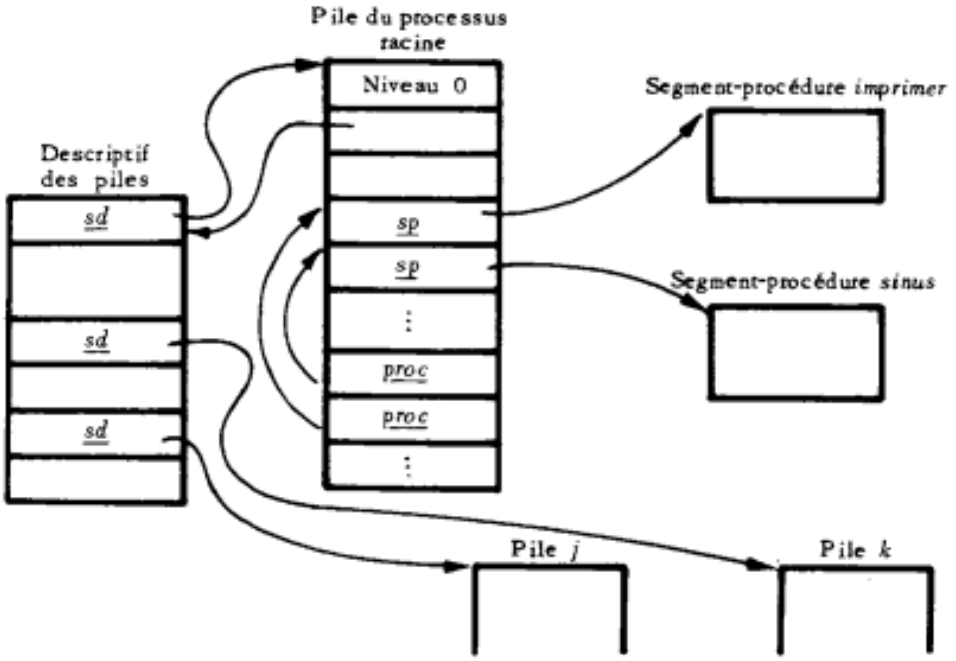


Figure 27. Processus racine.

3.383 Partage des procédures communes à plusieurs processus

Rappelons que l'édition de liens est entièrement effectuée avant l'exécution et qu'il n'y a donc pas d'édition dynamique. Nous considérons ici les procédures compilées qui correspondent aux programmes des utilisateurs. Le partage de tels programmes entre plusieurs processus nécessite la présence d'un tronçon commun *B*. Si *B* était la région racine, celle-ci contiendrait alors des objets dont la durée de vie serait inférieure à celle du système et qui seraient en nombre inconnu. Il faudrait introduire, au niveau du moniteur, une gestion

dynamique de leurs noms. Cette solution reviendrait aussi à rendre les procédures communes à tous les processus. Pour éviter cette gestion dynamique, le tronçon commun *B* est une région de niveau d'emboîtement *1* qui est créée à l'aide d'un processus auxiliaire.

Soit un segment-procédure *p* catalogué. Quand aucun processus ne l'utilise, son descripteur n'apparaît dans aucune pile. Lorsqu'un premier utilisateur en demande l'exécution, le système crée un processus auxiliaire *a* dont l'unique région est de niveau d'emboîtement *1*. Il inscrit dans cette région, à des emplacements prévus dès la compilation (de nom statique *1. k*), le descripteur de *p* et les descripteurs de tous les objets-procédures compilés en même temps que *p* et qui sont déclarés dans *p*. Ces descripteurs figurent dans le catalogue des fichiers. Le système complète ainsi un descriptif utilisable par tous les processus créés sur *p*. Un procédurus, constitué de l'objet procédure correspondant au bloc le plus externe et de l'environnement associé à la région est aussi créé dans cette région. Puis un processus est lancé sur ce procédurus comme fils de *a*. Si la procédure *p* est déjà utilisée, *a* n'a pas à être créé : on se contente de créer un nouveau fils pour *a*. Le nombre de processus utilisateurs de *p* est comptabilisé et *a* est détruit à la fin de l'exécution de son dernier fils.

Exemple. Dans la figure 28, deux processus *l* et *m* utilisent le même programme *p*. Un processus *n* utilise le programme *q*. Le processus *m* a appelé une procédure du moniteur, par exemple sous l'effet d'une interruption.

3.4 GESTION DE L'INFORMATION DANS LE SYSTÈME ESOPE

Les calculateurs décrits dans les deux exemples qui précèdent comportent des mécanismes d'adressage évolués, qui permettent de réduire l'importance des opérations programmées à mettre en œuvre par le système pour l'accès à l'information. Les techniques avancées utilisées sur ces matériels ne doivent pas décourager un concepteur de système disposant d'une machine moins perfectionnée. A partir des concepts qui viennent d'être développés, deux voies s'ouvrent à lui :

— réaliser une machine virtuelle en dotant son calculateur d'une extension programmée,

— restreindre les possibilités offertes aux utilisateurs pour tenir compte des limitations du matériel disponible.

Dans la pratique, la solution adoptée est souvent un compromis entre ces voies extrêmes.

A titre d'illustration, nous présentons une description schématique des mécanismes de gestion de l'information dans le système ESOPE, réalisé sur un calculateur CII 10070 [Bétourné, 70]. Aucun concept nouveau n'est ici introduit ; nous montrons comment certaines des idées développées dans 3.2 sont mises en œuvre compte tenu des restrictions apportées par le matériel.

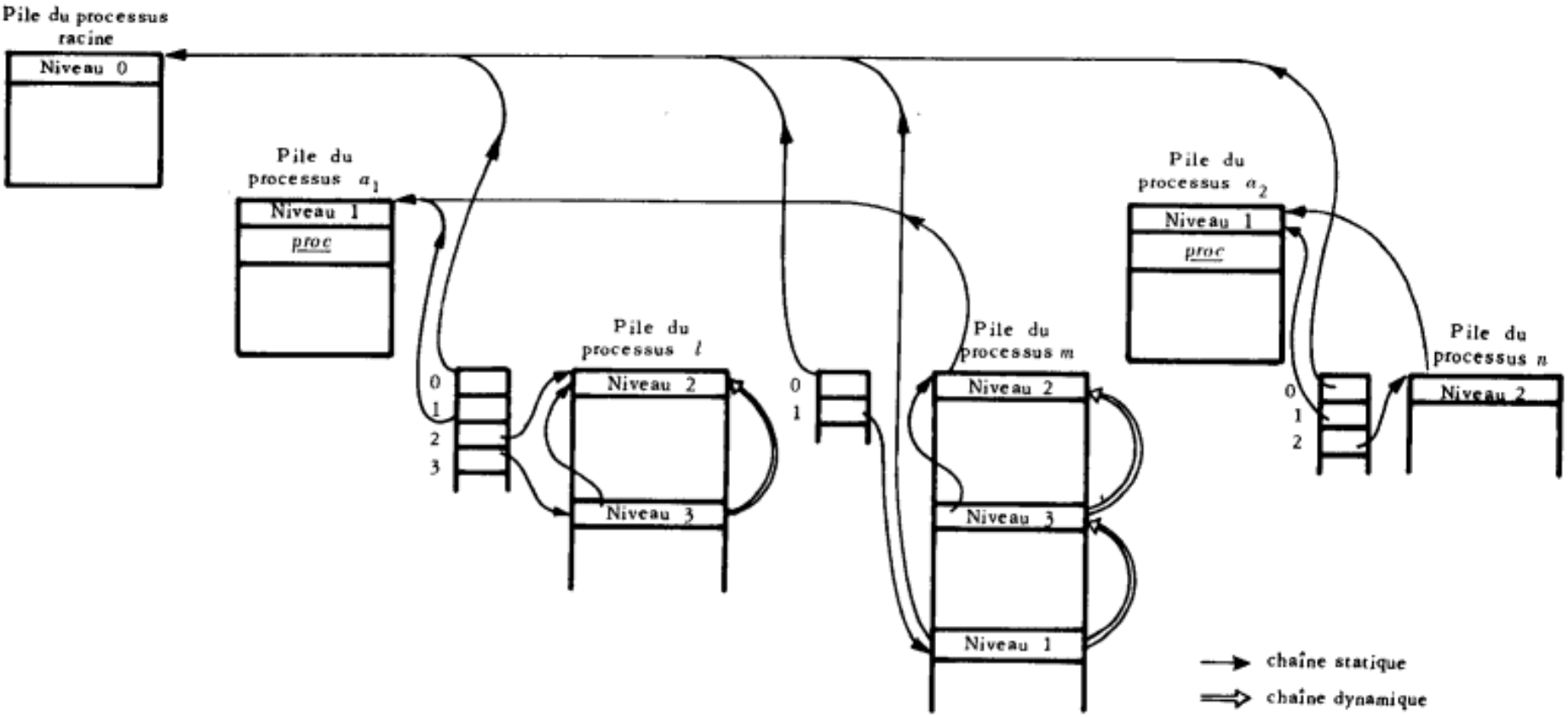


Figure 28. Partage de programmes.

3.41 LE MATÉRIEL

Nous présentons ici brièvement les caractéristiques du CII 10070 qui nous seront utiles dans ce chapitre.

3.411 La mémoire physique

Nous considérons la mémoire physique du 10070 comme composée de $N + P$ mots de 32 bits ; cette mémoire est constituée par :

— une mémoire principale directement adressable par l'unité centrale, comprenant N mots $(N \leqslant 2^{17})$,

— une mémoire secondaire (disques), non adressable par l'unité centrale, comprenant P mots.

L'unité d'information en mémoire physique est la page de 512 mots. Une page est repérée par une adresse physique A, qui peut désigner un emplacement en mémoire ou sur disque.

3.412 L'adressage topographique

Lorsque l'unité centrale se trouve dans le mode « avec topographie », ce que nous supposons dans tout ce qui suit, le mécanisme d'adressage fonctionne ainsi :

— une adresse de mot est constituée par un nombre représentable sur 17 bits,

— ce nombre est interprété comme un couple (numéro de page v, déplacement d).

Le numéro de page v désigne un registre de 8 bits dont le contenu φ est interprété comme un numéro de page physique en mémoire principale ; l'adresse finalement obtenue résulte de la concaténation de ce numéro avec le déplacement d initial (Fig. 29). L'ensemble des 256 registres correspondant à tous les numéros de pages possibles est appelé **mémoire topographique.** Ce mode d'adressage permet de gérer la mémoire physique par pagination (cf. 4.45).

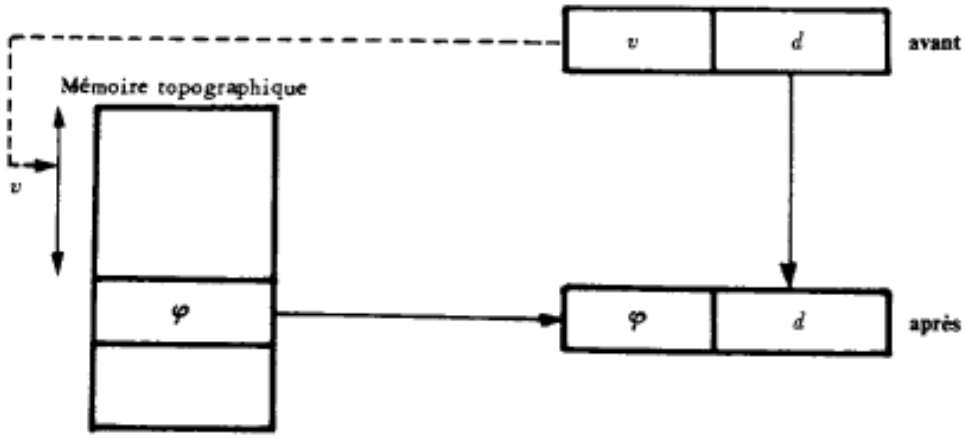


Figure 29. Transformation d'adresse par la mémoire topographique.

Lorsque le registre topographique de numéro v contient une valeur particulière de φ, notée *nil*, cette valeur n'est pas interprétée comme un numéro de page physique et il y a déroutement. Pour des raisons technologiques, la valeur *nil* est codée en utilisant les deux bits de protection associés à chaque registre de la mémoire topographique (cf. 4.45).

3.42 LA MÉMOIRE ADRESSABLE

Le système gère des **usagers,** définis comme des ensembles disjoints de processus. Nous décrirons successivement l'ensemble des objets existant dans le système et l'ensemble des objets accessibles à un instant donné à un usager.

3.421 L'espace des segments

Tout objet existant dans le système est représenté comme un **segment.** Un segment est un ensemble ordonné d'**articles** ; un article est une suite de 512 mots et a donc la taille d'une page de mémoire. Un article est désigné de façon unique par un couple (f, a) où :

f est le **nom interne** du segment,

a est le nom de l'article dans le segment.

Ce mode de désignation est interne au système. Le nombre de segments et le nombre d'articles par segment sont limités.

L'ensemble des couples (f, a) admissibles constitue l'**espace des segments.** Nous examinerons plus loin comment un segment est désigné par les utilisateurs du système, et comment est établie la correspondance entre ce mode de désignation et les noms internes.

3.422 La mémoire virtuelle

A un instant donné, l'information accessible à un usager est définie par sa **mémoire virtuelle.** On appelle ainsi l'ensemble des noms pouvant être construits avec 17 bits, longueur du champ « adresse de mot » dans les instructions du 10070 ; un tel nom, ou **adresse virtuelle,** s'interprète comme un couple (nom de page virtuelle (8 bits), déplacement). La mémoire virtuelle comprend donc 256 pages virtuelles. Les processus de l'usager doivent nécessairement désigner toute information par une adresse virtuelle.

3.43 DÉSIGNATION DES SEGMENTS

Jusqu'ici, nous n'avons désigné un segment que par son nom interne f. Ce nom est réservé au système pour nommer de façon unique un segment : il désigne une entrée dans une Table Générale des Segments *(TGS)*, unique pour le système, qui contient pour chaque segment existant dans le système un **descripteur** regroupant toutes ses caractéristiques (taille, protection, ...). Les usagers utilisent d'autres noms pour désigner les segments qu'ils manipulent.

a) *Nom temporaire*

Pendant qu'un usager utilise un segment, il peut être nécessaire de donner à ce segment une protection spécifique vis-à-vis des processus de l'usager, pendant la durée de cette utilisation.

Par ailleurs, dans certains cas (données temporaires) on souhaite pouvoir limiter la durée de vie d'un segment et, en particulier, la lier à celle d'un usager donné. C'est pourquoi un usager doit donner à chaque segment qu'il utilise un **nom temporaire** qui identifie le segment pour ce seul usager. Ce nom est utilisé comme paramètre dans les diverses procédures d'action sur les segments, qui sont :

— *créer* un segment de taille déterminée avec un nom temporaire donné,

— *ouvrir* un segment, c'est-à-dire autoriser l'accès à ses articles par des processus de l'usager avec une protection donnée,

— *fermer* un segment, c'est-à-dire interdire tout accès à ses articles,

— *détruire* un segment et rendre réutilisable son nom temporaire,

— *modifier* la taille d'un segment par création ou destruction d'articles.

La Table des Segments Utilisés *(TSU)* regroupe les segments utilisés par l'usager à un instant donné. Ces segments sont accessibles par leur nom temporaire.

Si un segment n'est utilisé que d'une façon temporaire (pour ranger des résultats intermédiaires par exemple), son nom temporaire suffit à le désigner à un usager. Si on désire lui donner une existence plus longue (et en particulier le faire survivre à la disparition de l'usager, permettre son partage entre usagers, ...), alors on doit utiliser un autre mode de désignation.

b) *Identificateur*

Un identificateur (cf. 3.1) permet à un utilisateur du système de désigner un segment, par exemple dans le langage de commande. Un identificateur résulte de la concaténation d'un indicatif de client (ou utilisateur) et d'un indicatif de segment (par exemple *DUPONT * TOTO*). Les identificateurs des segments d'un même client sont regroupés dans une table appelée **catalogue** du client. On introduit les opérations suivantes :

— *cataloguer* un segment de nom temporaire t (pour l'usager U) en le désignant par l'identificateur x ; le segment peut maintenant survivre à la perte du nom temporaire,

— *associer* le segment désigné par l'identificateur x avec le nom temporaire t pour l'usager U, avec un droit d'accès donné ; le segment peut maintenant être manipulé par l'usager U,

— *dissocier* un segment désigné par l'identificateur x du nom temporaire t pour l'usager U.

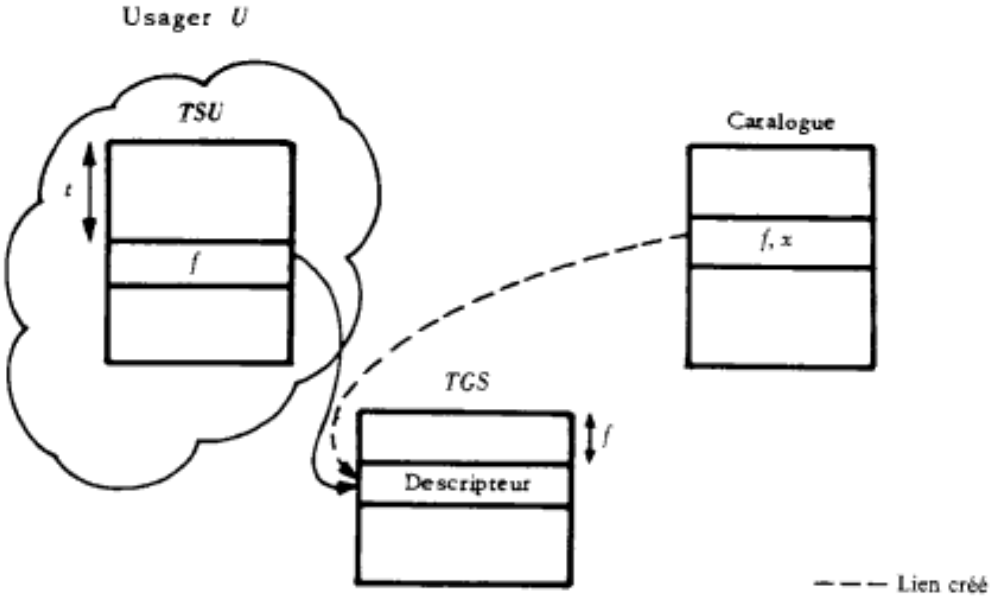


Figure 30. Effet de *cataloguer*.

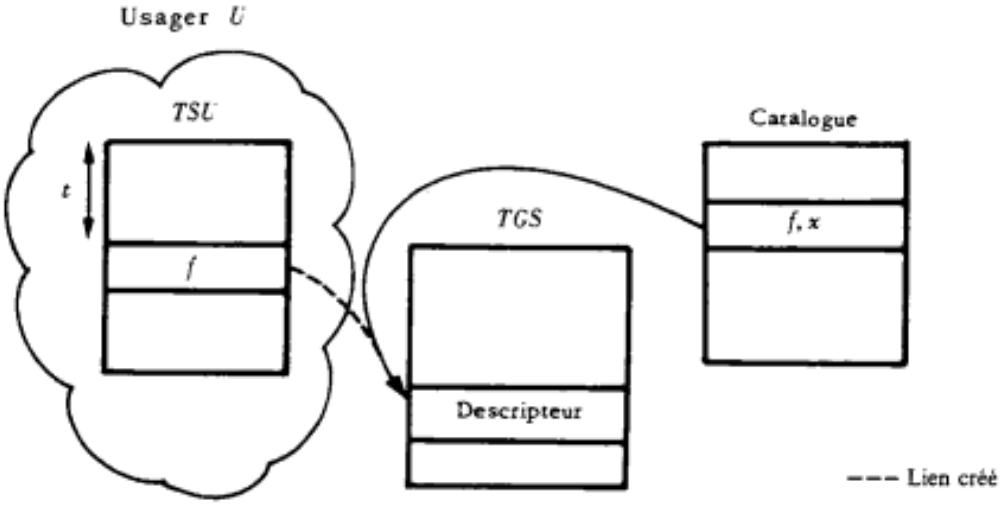


Figure 31. Effet d'*associer*.

3.44 ACCÈS A L'INFORMATION : LE COUPLAGE

3.441 **Principe du couplage**

On désigne sous le nom de **couplage** le mécanisme permettant à un usager d'atteindre, grâce au mode d'adressage fourni par sa mémoire virtuelle, un objet appartenant au sous-ensemble de l'espace des segments qui lui est couramment accessible, c'est-à-dire défini par la composition courante de sa *TSU* (ce mécanisme a été introduit pour la première fois dans le système GORDO [Anderson, 68]). Le couplage associe une page v de la mémoire

virtuelle d'un usager U à un article de segment (f, a). Un article ainsi couplé est connu des processus de l'usager U sous le nom v. Un article peut être couplé aux mémoires virtuelles de plusieurs usagers, et peut être connu de ces usagers sous des noms différents ; il peut également être couplé à plusieurs pages virtuelles d'un même usager, et donc connu de lui sous plusieurs noms différents. Nous examinerons plus loin les conditions de ce partage et les restrictions auxquelles il est soumis.

A un instant donné, l'information accessible aux processus d'un usager est définie par l'état de couplage de sa mémoire virtuelle, représenté dans la Table des Pages Virtuelles *(TPV)* de l'usager.

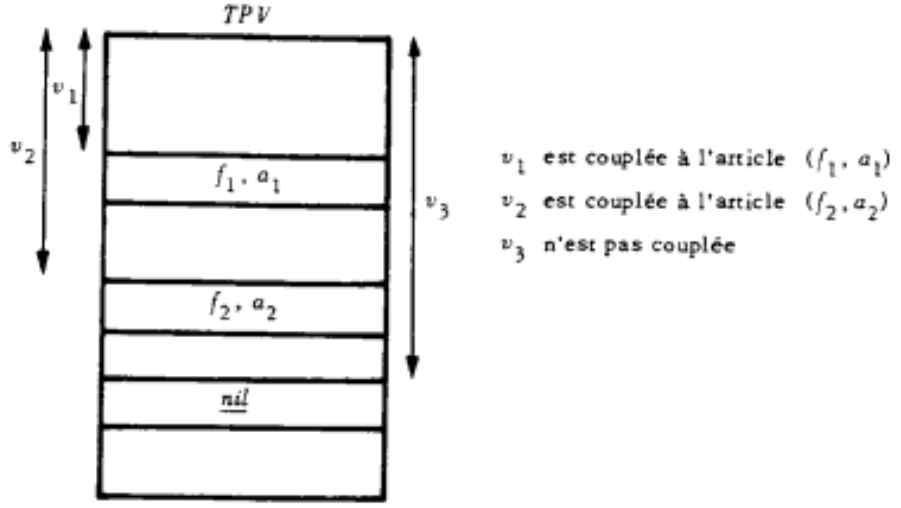


Figure 32. Etat de couplage d'une mémoire virtuelle.

On dispose des deux opérations suivantes :

— *coupler* à une page virtuelle un article d'un segment ouvert, avec une protection spécifiée,

— *découpler* une page virtuelle couplée, ce qui interdit tout accès à travers cette page.

Ces opérations permettent de modifier, à l'exécution, le contenu de la *TPV* et donc l'ensemble d'informations accessible à l'usager. Un nom de page virtuelle joue ainsi le rôle d'un repère pour des articles de l'espace des segments.

Remarque. Il y a une différence fondamentale entre l'emploi de la pagination tel qu'il apparaît ici et celui développé au chapitre 4. Ici, la mémoire virtuelle est utilisée comme un ensemble de repères vers l'espace des segments et le changement de valeur d'un repère n'entraîne aucun mouvement d'information.

3.442 Réalisation du couplage

Nous décrivons ici les structures de données utilisées pour la mise en œuvre du couplage, et les opérations sur ces données.

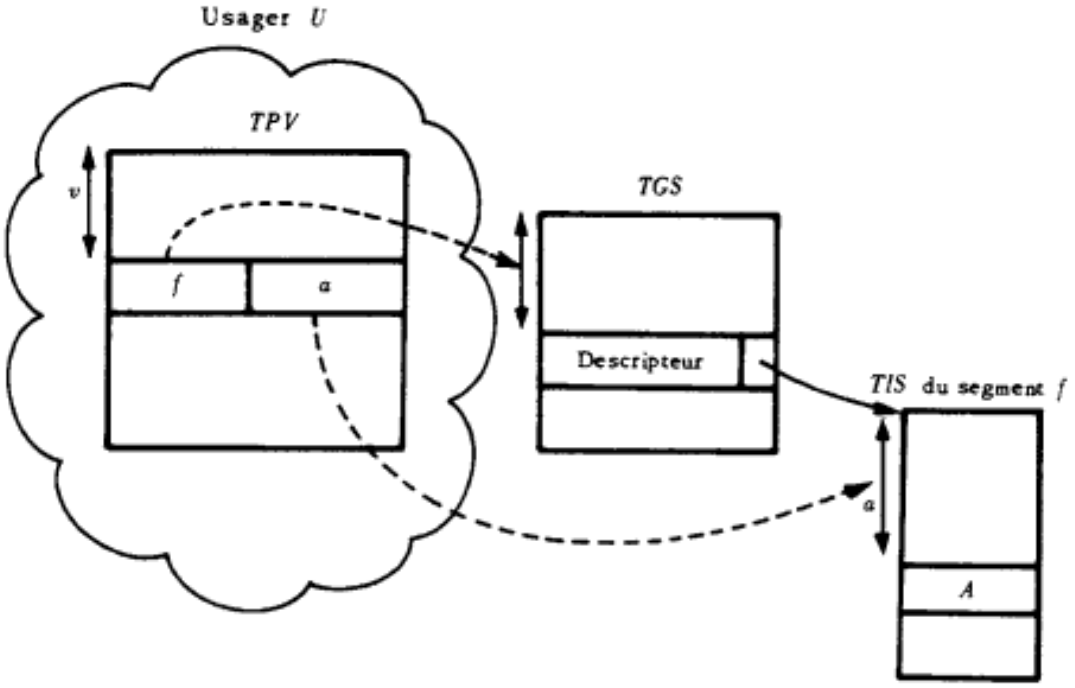


Figure 33. Localisation d'un article couplé.

La figure 33 montre comment on détermine la localisation physique de l'article *(f, a)* couplé à une page virtuelle *v*. Le nom *f* permet de retrouver dans la *TGS* le descripteur du segment *f*, qui comporte, entre autres, un pointeur vers une Table d'Implantation du Segment *(TIS)*. Celle-ci indique pour chaque article du segment l'adresse physique *A* où il est implanté. *A* est donc une adresse en mémoire principale ou sur disque, qui est tenue à jour par les processus du système chargés d'allouer les ressources physiques.

Dans le cas où *A* désigne une adresse de page en mémoire principale, soit φ, on peut utiliser la mémoire topographique comme moyen d'accélérer la localisation physique de l'article couplé à la page virtuelle *v*. Dans ce cas, en effet, φ est recopié dans le registre topographique du nom *v*, et l'adresse physique (φ, d) est obtenue directement à partir de (v, d) par le mécanisme câblé décrit en 3.412. Lorsque le registre de nom *v* contient *nil*, il y a déroutement, et l'adresse physique *A* est déterminée par la voie *TPV-TGS-TIS*. Le chargement d'une adresse φ dans un registre topographique a lieu :

— quand un article primitivement sur disque est chargé en mémoire principale,

— quand lors d'un accès par la voie *TPV-TGS-TIS* on trouve un *A* désignant une adresse en mémoire principale (par exemple parce que l'article avait déjà été chargé en mémoire pour le compte d'un autre usager).

Dans le cas où *A* désigne une adresse sur disque, il y a défaut de page et des dispositions sont prises pour transférer en mémoire principale l'article manquant.

3.443 **Contraintes**

La principale contrainte pour l'utilisation du couplage provient du fait qu'il n'existe pas de dispositif permettant la réimplantation d'une procédure dans la mémoire virtuelle. Après édition de liens, une procédure se trouve donc liée à des adresses virtuelles fixes ; il en résulte que les articles d'un segment contenant une procédure doivent toujours être couplés aux mêmes pages virtuelles.

De même une procédure, après l'édition de liens, doit toujours trouver ses données à des adresses fixes ; il est possible de tourner cette restriction, au prix d'un alourdissement du programme, en n'accédant aux données que par une indirection sur un registre suivie d'une indexation (voir exercice 10).

3.45 PARTAGE DES SEGMENTS

Aux restrictions près indiquées en 3.443 pour les segments de procédures, les structures de données et les mécanismes introduits permettent le partage d'un segment entre usagers suivant le schéma de la figure 34.

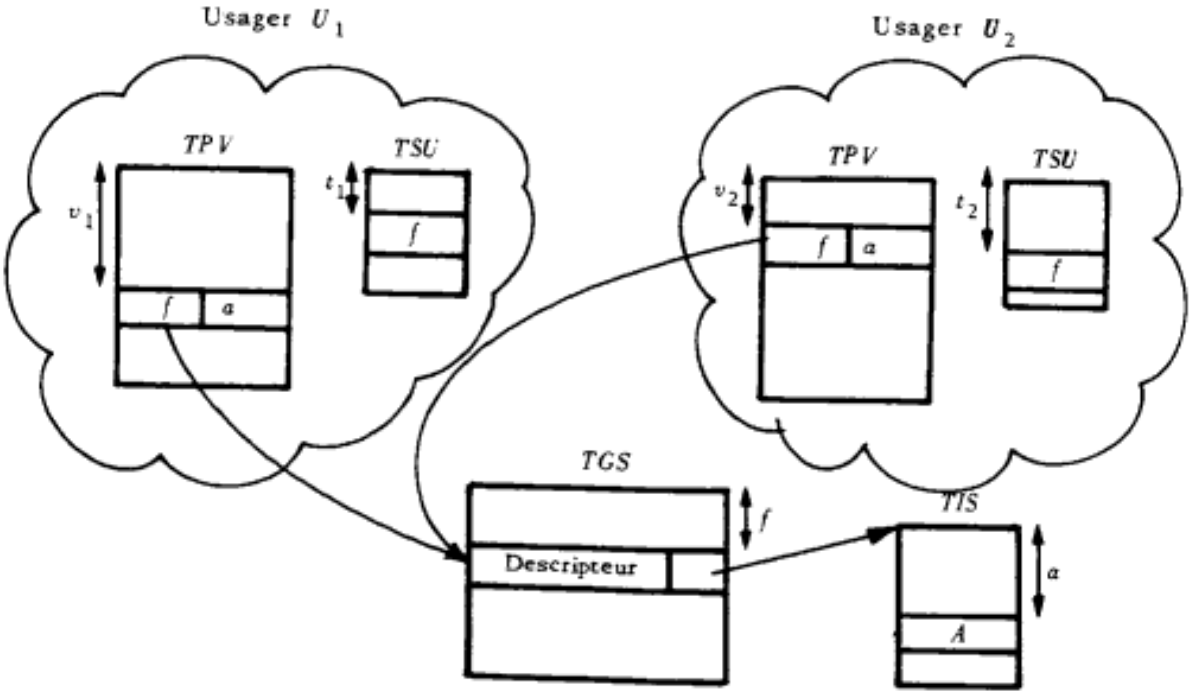


Figure 34. Partage d'un segment entre usagers.

Dans ce schéma, le segment de nom interne *f* est connu de l'usager *U1* sous le nom temporaire *t1*, de l'usager *U2* sous le nom temporaire *t2*. Son article *a* est couplé à la page virtuelle *v1* chez *U1*, *v2* chez *U2* (*v1* = *v2* s'il s'agit d'un segment-procédure, d'après 3.443). Le mécanisme d'accès décrit en 3.442 conduit à l'adresse physique *A* de l'article considéré.

3.46 UTILISATION DES MÉCANISMES DE GESTION DE L'INFORMATION

Nous illustrons par un exemple simple l'utilisation des mécanismes de gestion de l'information du système ESOPE. Soit à interpréter la commande

$$* \; EDIT \qquad SI = TOUAMOTOU,\ SO = TOMBOUCTOU$$

Cette commande appelle un éditeur de textes interactif, qui reçoit en entrée un texte symbolique contenu dans un segment « source » *TOUAMOTOU* et délivre en sortie un texte modifié dans un segment « objet » *TOMBOUCTOU*.

En ce qui concerne la gestion de l'information, l'interprétation de cette commande donne lieu aux opérations suivantes dans la mémoire virtuelle de l'usager qui a émis la commande :

1) *associer*, *ouvrir* et *coupler* le segment *EDIT-CODE* contenant le programme de l'éditeur de textes. Le segment *EDIT-CODE* est un segment partagé qui existe en exemplaire unique pour tout le système. Il contient un programme exécutable absolu ayant déjà subi l'édition de liens et doit donc être couplé à des pages virtuelles fixes.

2) *créer* un segment de travail (propre à l'usager appelant) destiné à contenir les données de travail de l'éditeur pour cet usager. Ces données sont désignées dans le programme par des adresses fixes ; on doit donc coupler le segment à des pages virtuelles spécifiées.

Le segment de travail est spécifique à une exécution particulière du segment-procédure *EDIT-CODE* par un processus d'un usager. Un nouvel exemplaire doit donc en être constitué à chaque exécution et, pour réutiliser l'espace en mémoire physique, il est détruit à la fin de l'exécution. Le segment de travail n'est désigné que par un nom temporaire (soit *t*).

3) Certaines des données qui figurent dans le segment de travail doivent être initialisées. Cette initialisation est faite après la création d'un exemplaire particulier de ce segment, en y recopiant le contenu d'un segment partagé *EDIT-DATA*, unique pour le système. On doit donc *associer* puis *ouvrir* *EDIT-DATA*, le *coupler* dans des pages de travail, transférer son contenu dans le segment *t*, et enfin le *fermer* et le *dissocier* puisqu'il ne sera plus utilisé.

Remarque. Dans la procédure de l'éditeur de textes figurent des instructions *(associer, coupler, ...)* qui désignent les segments *TOUAMOTOU* et *TOMBOUCTOU*.

3.5 REPRÉSENTATION ET GESTION DES OBJETS

Les exemples qui précèdent ont montré diverses solutions aux problèmes de la représentation interne et de la gestion des objets d'un système. Nous présentons ici les enseignements que nous en tirons pour la représentation des objets et l'accès aux emplacements qui les contiennent.

3.51 REPRÉSENTATION DES OBJETS

3.511 **Généralités**

Un objet est, suivant sa complexité, représenté sur un ou plusieurs emplacements. Il est caractérisé par les informations suivantes :

— son type,
— sa taille,
— sa durée de vie,
— la fonction d'accès aux composants de l'objet,
— les restrictions d'accès à cet objet,
— sa valeur.

a) *Le type.* Le type précise la nature de l'objet représenté et les opérations dont il peut être un opérande. Il peut être consulté à l'occasion de certains accès (affectation, opération, passage d'arguments), notamment lorsque la validité de ces accès n'a pu être vérifiée à la compilation.

Du type d'un objet, quand il est représenté, on peut dans certains cas déduire sa taille (exemple : un entier), les restrictions d'accès (exemple : une procédure), ou même la fonction d'accès aux composants (exemple : fichier séquentiel).

Exemple. Dans BURROUGHS B6700, c'est le préfixe qui détermine le type. Dans CLICS et dans ESOPE, le type n'est pas représenté.

b) *La taille.* La taille indique le nombre, fixe ou variable, des emplacements occupés par la représentation de l'objet. Elle est enregistrée lorsqu'elle n'est pas déterminée par le type.

c) *La durée de vie.* La durée de vie a été définie en 3.114 comme le temps pendant lequel un objet est accessible à l'aide d'un nom. La durée de vie d'un objet peut être déterminée automatiquement par des opérations implicites liées à l'exécution d'un programme ou spécifiée explicitement par une instruction du programme. Dans ce dernier cas, il est nécessaire de représenter l'indication de durée de vie afin de contrôler la validité de certaines opérations et de récupérer au moment opportun les emplacements libérés.

Exemple 1. Dans CLICS et dans BURROUGHS B6700, les objets locaux à un bloc ou à une procédure, qui ont même durée de vie, sont rangés dans une pile gérée en fonction de cette durée de vie.

Exemple 2. Un fichier utilisé par un processus qui n'a pas le pouvoir de le détruire a une durée de vie indépendante de celle de ce processus. Au contraire un fichier temporaire est détruit en même temps que le processus qui l'a créé.

d) *La fonction d'accès.* La fonction d'accès à un objet composé peut être définie comme l'ensemble des informations nécessaires pour atteindre soit l'objet composé lui-même, soit chaque élément de cet objet (cf. 3.113).

Exemple 1. Dans BURROUGHS B6700, un vecteur est représenté par un segment. La fonction d'accès se déduit de l'adresse de base du segment et de sa taille. Une matrice

(n, m) est représentée par n segments de taille m. La fonction d'accès est représentée par le contenu de $n + 1$ descripteurs : un descripteur primaire d'un segment S et n descripteurs de segment contenus dans le segment S.

On note que cette représentation est applicable à une collection de n enregistrements de taille variable.

Exemple 2. Dans CLICS, un segment de données peut être le résultat de la compilation d'une suite d'objets définis par programme. La fonction d'accès est complétée par une table appartenant au segment lui-même. Chaque ligne de la table comporte l'identificateur de l'objet et l'adresse de l'emplacement correspondant dans le segment.

e) *La restriction d'accès.* La restriction d'accès définit les opérations permises ou interdites sur l'objet. Elle est représentée lorsqu'elle n'est pas déterminée par le type, ce qui est le cas lorsqu'elle varie au cours du temps ou lorsqu'elle est fonction du processus qui désigne l'objet. Par exemple, on note dans la représentation d'un fichier s'il est accessible en écriture ou en lecture.

f) *La valeur.* La valeur d'un objet est une suite de bits, interprétable par un processeur ; c'est une constante ou un nom.

3.512 Décomposition de la représentation d'un objet

Généralement, on divise la représentation interne d'un objet composé en deux parties :

— une partie statique, ou descripteur, qui est de taille connue et fixe et dont la composition se déduit du type de l'objet,

— une partie dynamique de taille quelconque, parfois variable.

Cette taille et l'origine de la partie dynamique figurent dans la partie statique.

Exemple. Dans les trois systèmes étudiés, tous les objets composés (fichier, tableau, collection de données, prodécure) sont représentés par une partie statique (le descripteur) et une partie dynamique (un ensemble d'articles dans ESOPE, les segments dans BURROUGHS B6700 et CLICS).

3.513 Partage d'un objet

Le partage d'un objet est la possibilité pour plusieurs processus d'accéder à cet objet (cf. 3.121). On considère le cas où chaque processus peut désigner l'objet partagé par un nom qui lui est propre.

Le partage intervient sur les éléments de la représentation d'un objet :

a) le type, la taille, la durée de vie et la valeur sont uniques et donc partageables,

b) la fonction d'accès et la restriction d'accès ne sont pas nécessairement uniques. A moins qu'elles ne soient déterminées par le type de l'objet, il doit en exister un exemplaire par processus.

Le partage introduit une nouvelle décomposition de la représentation d'un objet :

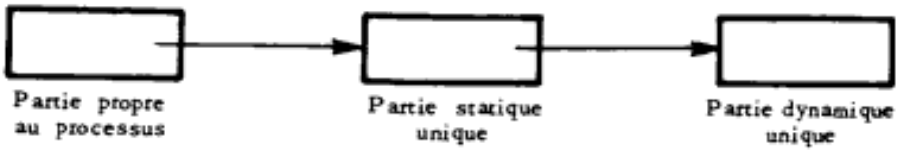
— une partie unique, commune à tous les processus,

— une partie propre à chaque processus.

D'où la schématisation suivante :

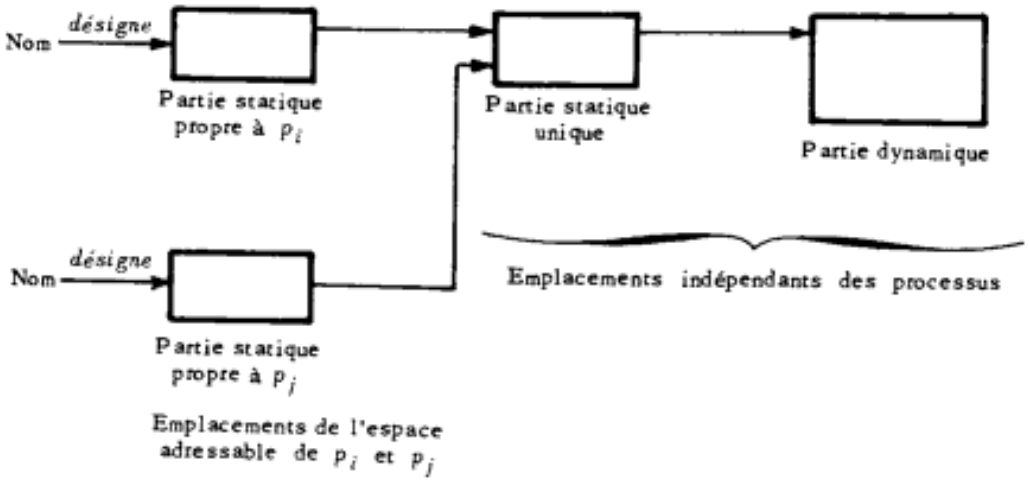
— objet simple

— objet composé



Dans ces schémas, la partie statique et la partie dynamique d'un objet partagé sont contenues dans des emplacements indépendants des processus. La partie propre au processus est contenue dans un emplacement attribué au processus ; elle est toujours désignée par un nom.

Le schéma de partage, par deux processus p_i et p_j, d'une valeur composée est le suivant :



Ce schéma n'est généralement pas appliqué tel quel dans les systèmes : en limitant sa généralité, on s'efforce d'accroître son efficacité, par exemple en supprimant un chaînage.

Exemple 1. Dans BURROUGHS B6700, les valeurs simples partagées ont un même nom. Leurs représentations, statiques et intrinsèques, sont rangées dans une pile commune aux processus.

Exemple 2. Dans CLICS, les valeurs simples ne sont pas partagées. Les valeurs composées comprennent un segment (partie intrinsèque dynamique) et un descripteur propre à chaque processus. Le descripteur contient des informations intrinsèques (taille, origine du segment) et des informations propres au processus (restrictions d'accès).

Exemple 3. Dans BURROUGHS B6700, les objets composés (segment de données) peuvent être partagés de deux manières :

— partage par nom (un seul descripteur dans une pile commune),
— partage par valeur (un descripteur par pile).

Exemple 4. Dans ESOPE, un objet composé partagé est un segment. Sa représentation contient quatre parties :

a) une partie propre au processus, contenant des restrictions d'accès propres au processus et un lien de chaînage vers le descripteur unique du segment,
b) le descripteur unique contenant toutes les caractéristiques statiques intrinsèques (taille, protection, et origine de la table d'implantation du segment),
c) la table d'implantation du segment, de taille dynamique, dont chaque emplacement contient l'adresse physique d'un article,
d) l'ensemble des articles.

3.52 ACCÈS AUX OBJETS

Les exemples des paragraphes 3.2 à 3.4 ont fait apparaître une grande diversité des formes de noms. Chaque fois que cela était utile pour désigner un objet dans un ensemble particulier, une nouvelle forme était introduite avec un nouveau qualificatif (statique, segmenté, virtuel, ...).

Nous allons, dans ce paragraphe, revenir sur les principaux aspects du nom et sur les formes rencontrées.

3.521 **Nom d'un objet**

Dans ce chapitre nous ne nous sommes intéressés qu'à la façon d'atteindre un objet, et non aux opérations qu'on lui fait subir. Pour effectuer une opération sur un objet d'un ensemble, il faut pouvoir le distinguer des autres et le désigner à l'organe qui exécute l'opération.

A un instant donné, la désignation d'un objet parmi d'autres est rendue possible par numérotation ; le numéro dans cet ensemble a été appelé « nom ». Cette numérotation a les propriétés suivantes :

— un nom ne désigne qu'un seul objet (dans cet ensemble, à un instant donné),
— tout objet est désigné par un nom.

Dans une machine, l'ensemble des emplacements est fini mais les processus créent un nombre arbitrairement grand d'objets qui doivent être représentés dans les emplacements ; ces derniers doivent donc être réutilisés. Les emplacements occupés par un objet deviennent réutilisables à la fin de la vie de cet objet.

Le découpage d'un ensemble d'objets en sous-ensembles présente certains avantages :

1) si les objets regroupés ont même durée de vie, on peut résoudre globalement les problèmes de réutilisation des emplacements correspondants.

Exemple. Dans BURROUGHS B6700, des objets peuvent être créés en début de bloc ; ils sont détruits à la fin du bloc correspondant. Les objets créés en début de bloc occupent une région et ont même durée de vie. Les problèmes de réutilisation des emplacements sont traités au niveau de la région : à la sortie d'un bloc, la région qui en est issue (région courante) est détruite et les emplacements qu'elle occupait sont libérés.

2) si le processus désigne pendant un certain temps des objets dans le même sous-ensemble (phénomène de localité, cf. 4.23), il peut accéder à un objet en conservant (par exemple dans un registre) le nom du sous-ensemble, constant pendant cette période, et en désignant l'objet par son nom dans le sous-ensemble. Par cet artifice, le champ-adresse d'une instruction peut être réduit.

Exemple 1. Dans CLICS, les registres *R0*, *R3*, *R4* sont utilisés dans ce but. Les objets rémanents d'une procédure qui s'exécute pour le compte d'un processus peuvent être désignés par celui-ci pendant toute la durée d'exécution de la procédure. Pendant cette durée, le registre pointeur de liaison *R4* contient l'adresse segmentée de l'origine du segment de liaison. Dans une instruction, un objet rémanent sera désigné par le nom de *R4* (4 bits seulement) et le déplacement dans le segment de liaison.

Il en est de même pour le registre pointeur de pile *(R3)* qui contient la base de la dernière zone de locaux rencontrée, et pour *R0* qui contient l'adresse relative (par rapport à la base du segment) de l'instruction en cours.

Exemple 2. Dans BURROUGHS l'ensemble des objets accessibles à un instant donné change en début et en fin de bloc. Les sous-ensembles introduits correspondent donc aux blocs. Pour désigner un objet dans une instruction, il suffira d'un nom statique (niveau d'emboîtement du bloc et rang de l'objet dans le bloc), le contenu des registres de contexte précisant le sous-ensemble. Le nom dynamique n'apparaît jamais dans une instruction ; les registres de contexte sont mis à jour en début et fin de bloc.

Soit un objet appartenant à un instant donné à deux ensembles ayant chacun sa propre numérotation. Pour passer d'une numérotation à l'autre, l'équivalence des deux noms de cet objet doit être exprimée dans un référentiel qui contient les deux ensembles précédents.

Exemple 1. Considérons le cas des externes dans CLICS. Soit S un segment auquel fait référence un segment S_0 pour un processus p_1. Le segment S est connu par p_1 sous le nom s_1 et peut être connu par un autre processus p_2 sous le nom s_2. On accède à S par l'intermédiaire du segment de liaison $L(S_0, 1)$. Lors de la première référence à S pour p_1, l'éditeur de liens dynamique utilise l'identificateur donné pour S à sa création et l'adresse de S en mémoire fictive.

Exemple 2. Dans BURROUGHS, un paramètre effectif peut être n'importe quel objet de la pile ; il peut donc se trouver en dehors du contexte pendant l'exécution de la procédure appelée. Pour le désigner, on utilise un nom dynamique, c'est-à-dire un nom dans la pile.

3.522 La mémoire fictive

La mémoire fictive est l'ensemble des emplacements du système. Nous l'avons choisie comme référentiel pour le partage des noms et nous avons supposé qu'un objet demeurait associé à un emplacement pendant toute la durée de vie de l'objet. Nous avons considéré que la mémoire fictive était linéaire ; elle pourrait tout aussi bien être segmentée. L'accès à un emplacement utilise l'adresse fictive.

3.523 Espace adressable d'un processus

L'espace adressable d'un processus à un instant donné est le sous-ensemble de la mémoire fictive constitué par les emplacements contenant les objets accessibles au processus à cet instant. Cet espace est nécessaire pour permettre le partage des procédures et des données dans la mémoire fictive et la création dynamique d'objets.

Exemple 1. Dans CLICS, le nom d'un emplacement de l'espace adressable a été appelé adresse segmentée. L'espace adressable est défini à tout instant par le descriptif.

Exemple 2. Dans BURROUGHS, l'espace adressable est formé :

— de la pile du processus et des tronçons de ses ancêtres ; ces éléments sont désignés par un nom dynamique,

— des segments dont le descripteur figure dans la pile ; on utilise alors un nom segmenté.

Exemple 3. Dans ESOPE, l'espace adressable est défini par la table des segments de l'usager *(TSU)*.

Dans ESOPE, nous avons rencontré un cas où le nombre maximal d'emplacements accessibles au processeur (mémoire virtuelle) est plus petit que l'espace adressable du processus. Le couplage permet d'utiliser, à des instants différents, la même adresse virtuelle pour désigner des objets différents de l'espace adressable.

L'adresse fictive d'un emplacement de l'espace adressable est obtenue à partir du couple :

— nom du registre contenant l'adresse fictive de l'origine de l'espace adressable,

— déplacement dans l'espace adressable.

3.524 L'environnement d'un processus

Dans certains langages de programmation, on a le moyen d'isoler dans l'ensemble des identificateurs d'un programme le sous-ensemble de ceux qui sont valides en un point donné. Ce sous-ensemble a été appelé lexique (cf. 3.34). Il reste en général stable pour une suite d'instructions (bloc ALGOL 60, procédure dans CLICS). Lorsqu'on exécute l'instruction qui se trouve au point considéré, l'ensemble des objets désignés par les identificateurs du lexique constitue l'environnement courant du processus.

Dans le cas où le nombre d'emplacements nécessaires pour contenir les objets créés au cours de l'exécution d'un programme est connu à la compilation, ces emplacements peuvent être alloués à ce stade (allocation statique). L'identificateur peut être converti en un nom de l'espace adressable, qui figure alors dans l'instruction. Le lexique n'a pas à être conservé à l'exécution.

Dans le cas où les objets créés par un programme sont gérés dynamiquement, leurs emplacements ne peuvent pas être connus lors de la compilation. On a alors le choix entre :

— conserver, ou même construire, le lexique lors de l'exécution (solution utilisée dans les interpréteurs),

— donner un numéro à chaque environnement et utiliser ce numéro dans la représentation des identificateurs.

Cette dernière solution évite de conserver les identificateurs du lexique. L'adresse fictive d'un emplacement est alors le triplet :

— nom du registre contenant l'adresse fictive de l'origine de l'espace adressable,

— nom du registre contenant le nom de la base de l'environnement dans l'espace adressable,

— nom de l'objet dans l'environnement.

Les objets de l'environnement sont les locaux, les paramètres et les externes. Les externes et les paramètres désignent des objets qui ne sont pas forcément dans l'environnement. Il faut donc les désigner dans l'espace adressable, référentiel commun à tous les environnements. Comme les objets de l'environnement n'ont pas même durée de vie, l'environnement est généralement segmenté en régions. Un descriptif de régions permet de les situer dans l'espace adressable.

Exemple. Dans CLICS, on peut associer à chaque anneau quatre régions, leur descriptif étant constitué par les registres *R0* (segment en cours d'exécution), *R3* (zone des locaux), *R4* (rémanents et références externes), *R5* (arguments).

EXERCICES

1. [1]

On suppose qu'il n'existe dans CLICS qu'un seul anneau ; il n'existe donc qu'un seul segment de liaison par procédure et qu'une pile par processus. On souhaite modifier CLICS pour ne conserver qu'un seul segment de liaison par processus. Dans ce cas, comment sont modifiés les mécanismes d'adressage ?

2. [1]

Dans les mêmes hypothèses que celles de l'exercice 1 (un seul anneau), on souhaite modifier CLICS pour supprimer la pile associée au processus. Les objets à gérer en pile sont répartis dans les segments de liaison. Dans ce cas, comment sont modifiés les mécanismes d'adressage ?

3. [2]

On souhaite modifier CLICS pour pouvoir réutiliser les noms des segments détruits. Quelles sont les opérations à effectuer lors de la destruction d'un segment ? Quelles sont les modifications du système qui faciliteraient ces opérations ?

4. [2]

On désire exécuter des programmes ALGOL 60 dans le système CLICS. On fait les hypothèses simplificatrices suivantes :

— tout bloc est une procédure sans paramètre,

— l'édition de lien entre toutes les procédures que contient le programme est faite avant l'exécution ; il n'existe pas de références à des segments de données,

— à tout objet-procédure est associé un seul segment et réciproquement. Le niveau d'emboîtement de la procédure est spécifié dans le descripteur de segment,

— l'évaluation des expressions s'effectue grâce à deux registres et à une zone gérée en pile,

— le compilateur interdit de passer une procédure en paramètre.

On demande :

1) Comment réaliser l'accès aux objets de l'environnement ?

2) Comment effectuer les commutations d'environnement ? On ne considérera pas le cas des branchements par étiquette.

5. [2]

Expliquer comment est rétabli l'environnement de la procédure appelante lors du retour de procédure dans le système BURROUGHS B6700.

6. [1]

Pourrait-on se passer des registres d'environnement et des registres de région courante dans le BURROUGHS B6700 ?

7. [1]

A un instant donné, 3 processus sont en cours dans le système BURROUGHS B6700. Deux d'entre eux, *u1* et *u2*, utilisent la procédure *p1*, le troisième, *u3*, utilise *p2*. Le programme suivant, qui contient les déclarations des objets du moniteur (par exemple, la procédure *sinus*) est une représentation de la situation décrite ci-dessus.

```
début   procédure sinus ; début
        procédure p11 ; début
                                    ... fin ;
                                    procédure p1 ; début ... fin ;
                                    créer processus u1 sur p1 ;
                                    créer processus u2 sur p1 ;
                                    attendre la fin de u1 et u2 ;
                        fin
        procédure p21 ; début       procédure p2 ; début ... fin ;
                                    créer processus u3 sur p2 ;
                                    attendre la fin de u3 ;
                        fin
        créer processus pseudo1 sur p11 ;
        créer processus pseudo2 sur p21 ;
        attendre la fin de pseudo1 et pseudo2 ;
fin
```

Que pensez-vous de cette interprétation ?

8. [3]

Imaginer les problèmes que pose la construction d'un système de fichiers dans BURROUGHS B6700 et décrire les fonctions à mettre en œuvre pour y accéder.

9. [2]

On rappelle que la primitive *coupler(v, a, f)* du système ESOPE (cf. 3.44) associe une page v de la mémoire virtuelle d'un usager à un article de segment (f, a), ce qui correspond à une opération vide dans le cas où la page v est déjà couplée à l'article (f, a). L'utilisation du couplage suppose une structuration logique du segment en articles de la taille d'une page. On désire maintenant masquer à l'usager le découpage du segment en articles, mais en contrepartie on veut lui donner la possibilité de gérer le segment comme un fichier séquentiel (du type « flot de caractères »). A cet effet on lui fournit la procédure :

écrire(u, n, f)

qui permet de copier dans le segment f, une chaîne de n caractères dont le premier est repéré dans la mémoire virtuelle par le nom u (u désigne une adresse d'octet). On demande de programmer cette procédure. On supposera que la taille d'une page est de l octets.

10. [2]

On a vu (cf. 3.443) que le problème du partage d'information (procédure ou données) dans la mémoire virtuelle du CII 10070 pouvait être résolu en éditant et en couplant chaque procédure et ses données aux mêmes adresses dans toutes les mémoires virtuelles où elles sont utilisées.

On se propose d'examiner plus généralement les façons de réaliser le partage de procédures et de données entre processus, en utilisant la mémoire virtuelle du CII 10070 et le mécanisme du couplage décrit en 3.44.

1) On suppose d'abord que chaque processus dispose d'une mémoire virtuelle propre. Dans quelles conditions peut-on éviter d'implanter aux mêmes adresses virtuelles les données partagées entre ces processus ?

2) On suppose maintenant que plusieurs processus peuvent travailler dans la même mémoire virtuelle. Quelles contraintes cela implique-t-il sur la programmation des procédures et des données partagées entre ces processus ?

4

GESTION DES RESSOURCES

4.1 NOTIONS GÉNÉRALES

Nous désignons par **ressource** tout élément dont peut avoir besoin un processus pour son exécution. Une fonction importante d'un système est la répartition entre ses utilisateurs des ressources qu'il administre. Le plus souvent, la somme des demandes à un instant donné est supérieure à la somme des ressources existantes, et cette situation entraîne l'attente de certains demandeurs. **L'allocateur** d'une ressource gère la file d'attente de cette ressource.

Du point de vue de l'allocation des ressources, un système est défini par un ensemble de files d'attente et par la politique adoptée pour gérer ces files : règles de passage d'une file à une autre, gestion interne de chaque file.

Lors de la conception du système, il s'agit, une fois définis les objectifs et recensées les contraintes, d'établir le schéma général des files d'attente et de déterminer les informations utilisées pour gérer ces files ainsi que les événements provoquant les transitions entre files.

Lors de la réalisation du système, il faut pouvoir représenter les ressources, les allocataires et les demandes. Il faut également tenir compte du fait que l'allocateur consomme lui aussi des ressources (principalement du temps d'unité centrale et de la place en mémoire). Il faut, par conséquent, envisager la manière dont ces ressources lui sont allouées : allocation particulière ou utilisation du mécanisme général.

Nous allons examiner les diverses ressources couramment utilisées et la manière de les représenter ; ensuite, nous envisagerons la nature des demandes de ressources et les réponses de l'allocateur à ces demandes.

4.11 EXEMPLES DE RESSOURCES

On peut classer les divers types de ressources selon certains critères non exclusifs les uns des autres.

a) Ressources physiques ou programmées

Certains types de ressources sont des organes de la machine : unités centrales, mémoire principale, canaux d'entrée-sortie, dispositifs périphériques. D'autres mettent en jeu la programmation : messages, programmes dits « de service » ou « utilitaires », fonctions ou procédures, fichiers. L'utilisation de ressources programmées implique celle de ressources physiques (mémoire par exemple).

b) Ressources à accès unique ou multiple (cf. 2.222)

Sont généralement à accès unique : les unités centrales, les zones de mémoire temporairement affectées à un processus, les fichiers privés ou dotés de protection d'accès, certains dispositifs périphériques (dérouleurs de bandes magnétiques, imprimantes, lecteurs-perforateurs de cartes, terminaux graphiques), les procédures ou programmes de service non réentrants.

Sont à accès multiple : les canaux d'entrée-sortie multiplexeurs, les procédures ou programmes de service réentrants, les fichiers en lecture (publics, ou privés avec autorisation par mot de passe).

c) Ensemble de ressources banalisées

Pour certaines catégories de ressources, qui existent en plusieurs exemplaires, le choix de l'unité allouée peut, dans certaines conditions, être sans importance pour le demandeur. Dans ce cas, les ressources sont dites **banalisées.**

Exemple. Les zones libres de mémoire principale, les dérouleurs de bandes magnétiques, les pistes de disques ou de tambours, les pages de mémoire principale sont des ressources banalisées.

d) Ressources virtuelles

Bien qu'un système d'exploitation dispose d'une quantité limitée de ressources, on peut donner à un processus l'illusion de posséder une ressource en propre alors qu'il n'y en a pas assez pour satisfaire toutes les demandes.

Exemple. Les imprimantes, qui existent en trop petit nombre, ne sont pas allouées aux processus ; ceux-ci disposent alors d'une imprimante virtuelle représentée par des zones de mémoire secondaire. Les échanges d'informations entre un processus et une imprimante réelle se font en deux étapes :

— transfert, sous le contrôle du processus, entre mémoire principale et imprimante virtuelle,

— transfert, sous le contrôle d'un processus du système d'exploitation, entre imprimante virtuelle et imprimante réelle (« spooling »).

De même, l'espace requis en mémoire par un processus peut n'être que partiellement en mémoire principale, le reste résidant sur une mémoire secondaire dédiée à la pagination ou au « va-et-vient » (cf. 4.432).

e) Ressources requérables

Une ressource est sujette à **réquisition** (« preemption ») si le système d'exploitation peut la retirer à un processus alors que celui-ci en a encore besoin. Selon le type de la ressource, le système peut, avec un coût variable suivant le type de ressource, effectuer les sauvegardes nécessaires de manière à être à même de restaurer ultérieurement l'état de la ressource avant la réquisition. Si ces mesures de sauvegarde n'ont pas été prises, le processus en question est détruit. Ainsi l'unité centrale est couramment réquisitionnée, avec sauvegarde du vecteur d'état du processus ; au contraire, il est rare d'appliquer ce procédé à un dérouleur de bande magnétique.

4.12 REPRÉSENTATION DES RESSOURCES

A toute ressource ou ensemble de ressources banalisées, il correspond dans le système une procédure d'allocation (l'allocateur) et un répertoire décrivant ces ressources. Le plus souvent, on définit pour chaque ressource une unité d'allocation (quantum), et on associe à chaque unité une entrée (descripteur) dans le répertoire de la ressource correspondante.

Exemple 1. Une mémoire paginée (cf. 4.45) est représentée par autant de descripteurs qu'il y a de blocs de mémoire allouables.

Exemple 2. Un ensemble d'unités de disques est représenté à raison d'une entrée par unité si le quantum d'allocation est l'unité complète. Ce même ensemble est représenté par autant d'entrées qu'il y a de pistes si le quantum d'allocation est la piste de disque.

Les informations contenues dans un descripteur dépendent de la nature de la ressource et de la façon dont elle est gérée. Le descripteur minimal se réduit à un bit représentant l'état, libre ou alloué, de la ressource.

Exemple. Dans l'allocation d'espace pour un disque, une chaîne de n bits peut représenter les n pistes du disque ; le numéro de la piste est ici déduit implicitement de la position du descripteur dans le répertoire de la ressource.

Les descripteurs contiennent fréquemment des informations autres que le simple état d'allocation ; citons quelques exemples.

— L'identification de l'allocataire (ou des allocataires pour les ressources partagées) permet de vérifier qu'un processus ne libère que des ressources qui lui appartiennent, ou rend possible la récupération de toutes les ressources allouées à un processus qui disparaît du système. Dans ce dernier cas, si le nombre de ressources de nature différente est important, il peut être préférable, en plus, de tenir à jour pour chaque allocataire une liste des ressources qu'il possède, plutôt que de devoir lors de la récupération explorer les répertoires de toutes les ressources du système.

— Les renseignements sur l'accès à la ressource peuvent être nécessaires (adresse physique d'une unité d'entrée-sortie), ou simplement utiles (liste des dispositifs facultatifs effectivement présents pour un terminal).

La structure d'un répertoire, c'est-à-dire la manière dont sont organisées les entrées, dépend principalement du nombre d'entrées et de la nature, constante ou variable, de ce nombre. Si le nombre d'éléments est invariant, ce qui est le cas le plus fréquent, le répertoire est un tableau : on représente ainsi par une table à n entrées les n blocs de mémoire d'une mémoire paginée. Si le nombre d'éléments est variable, le répertoire est généralement une liste dont les entrées sont créées dynamiquement. Une autre solution, possible si les variations du nombre d'entrées restent dans des limites connues et étroites, consiste à disposer d'une table contenant le nombre maximal d'entrées attendues.

Exemple. Une mémoire allouée par zones de tailles quelconques est l'exemple le plus courant d'une ressource à nombre variable d'éléments (cf. 4.442). Les allocations sont effectuées par groupes de cellules consécutives et l'élément d'allocation est créé à partir d'un élément libre de taille supérieure ou égale à la demande, ce qui revient à faire varier la taille et le nombre des éléments libres.

Le support du répertoire d'une ressource est généralement la mémoire principale ; toutefois si la durée de vie des allocations peut être supérieure à la durée d'une séance d'exploitation, ce répertoire doit être tenu à jour sur mémoire secondaire. Ainsi, l'état d'occupation des pistes d'un disque destinées à enregistrer des fichiers permanents se trouve sur le disque lui-même. Les procédures d'allocation sont alors plus complexes car il faut assurer la cohérence entre la copie du répertoire tenue à jour en mémoire principale (pour des raisons d'efficacité) et le répertoire lui-même sur mémoire secondaire (nécessaire au redémarrage du système).

4.13 ORIGINE ET FORME DES DEMANDES D'ALLOCATION

Une approche simple consisterait à admettre que les ressources sont affectées à des allocataires de type unique, par exemple les processus, et qu'un allocateur est activé sur demande explicite de ces derniers, soit pour acquérir une ressource, soit pour libérer une ressource précédemment acquise. Pour diverses raisons, la réalité est plus complexe : il existe dans un système des entités de nature et de durée de vie différentes liées à des notions telles que celle de travail, de processus, d'utilisateur, etc... et chacune de ces entités peut être allocataire de ressources ; de plus, les demandes de ressources ne sont pas toutes explicites (il faudrait par exemple qu'un processus dispose déjà du processeur « unité centrale » pour pouvoir demander un processeur).

Exemple. Dans un système multiprogrammé de traitement par train de travaux, certaines ressources peuvent être allouées globalement à un travail (unités d'entrée-sortie) alors que d'autres le sont individuellement aux processus composant le travail ;

en se plaçant dans ce cas au niveau des processus, on peut faire abstraction des ressources globales et considérer que les processus ne sont en compétition que pour l'utilisation des autres ressources (l'unité centrale, peut-être la mémoire, ...). La distinction entre ces deux niveaux d'allocation apparaît d'ailleurs nettement dans la structure des systèmes de ce type où l'on rencontre les deux fonctions de gestion des travaux d'une part, de gestion des processus d'autre part.

Les demandes de ressources peuvent prendre des formes très diverses.

Exemples.

— Un processus devient demandeur d'unité centrale à partir du moment où plus rien ne lui manque pour qu'il puisse s'exécuter : la fin du blocage du processus entraîne une demande d'unité centrale.

— Dans un système à mémoire paginée (cf. 4.45), un déroutement pour défaut de page peut être équivalent à la demande d'un nouvel.emplacement de page par le processus qui est à l'origine de ce déroutement.

— Dans un système à allocation statique de mémoire, l'indication de la taille de mémoire nécessaire, perforée sur une carte de commande accompagnant le travail, constitue une demande indirecte d'allocation de mémoire formulée par le programmeur.

L'identification de la ressource demandée peut être donnée avec plus ou moins de précision et, selon cette précision, l'allocateur dispose de plus ou moins de liberté.

Exemple 1 : allocation de mémoire à un travail.

La demande faite par le programmeur peut ne mentionner que la taille de la zone de mémoire nécessaire, laissant la possibilité à l'allocateur d'implanter le programme n'importe où ; elle peut aussi mentionner impérativement l'adresse d'implantation (cas des systèmes ne traitant que des programmes non translatables).

Exemple 2 : allocation d'espace secondaire dans le système OS/360.

Le programmeur voulant créer un fichier sur une unité à accès direct, peut faire la demande de réservation d'espace de diverses manières, en particulier :

— en précisant l'adresse physique de l'unité sur laquelle l'espace doit être alloué,

— en indiquant seulement le type de l'unité (par exemple disque 2314) lorsqu'il existe plusieurs unités du même type,

— en donnant seulement le groupe (par exemple toute unité à accès direct) auquel doit appartenir l'unité.

Les deux dernières formes permettent une meilleure utilisation des ressources en rendant possible la répartition des allocations sur l'ensemble des unités, ce qui permet en particulier d'équilibrer la charge des canaux.

4.14 FONCTIONS DE L'ALLOCATEUR

4.141 Généralités

La manière dont les ressources sont allouées dépend des objectifs choisis par le concepteur et des contraintes particulières liées à l'utilisation de chaque type de ressource. La bonne utilisation du matériel et la satisfaction des

utilisateurs sont les objectifs le plus souvent recherchés. Ces objectifs sont parfois contradictoires : par exemple, l'un des facteurs principaux tendant à la satisfaction de l'utilisateur est le temps de réponse du système ; ce temps de réponse est d'autant plus court que les ressources nécessaires ont une probabilité importante d'être disponibles ; dans ce cas elles sont sous-utilisées.

Le problème de l'allocation d'une ressource comporte dans le cas général deux aspects : gestion de la file d'attente des demandes (stratégie d'allocation) et choix de la ressource parmi les ressources banalisées (algorithme d'allocation).

Pour illustrer les choix possibles en matière de stratégies, prenons l'exemple d'une mémoire allouée par zones. Lorsqu'une zone devient libre, l'allocateur essaie de satisfaire les demandes de mémoire en attente ; il peut, selon les objectifs choisis :

— examiner ces demandes dans leur ordre d'arrivée (FIFO),
— associer à chaque demande la priorité attachée au processus demandeur,
— satisfaire la demande correspondant le mieux à la zone libre,
— satisfaire avec la zone libre le plus grand nombre possible de demandes (en commençant par les plus petites),
— satisfaire en premier lieu la demande la plus grosse présente dans la file, ce qui peut le conduire à geler la zone libérée jusqu'à ce que d'autres libérations lui fassent atteindre une taille suffisante.

Pour chaque type de ressource il existe généralement divers algorithmes possibles. Il est difficile de définir les meilleurs algorithmes, car dans l'évaluation de leur qualité interviennent des facteurs opposés tels que l'utilisation optimale de la ressource et le coût de l'algorithme.

Exemples

— Mémoire allouée par zones : les deux techniques les plus connues essaient de satisfaire une demande à partir, respectivement, de la première zone de taille suffisante et de la plus petite zone de taille suffisante.

— Unités d'entrée-sortie : à l'allocation au hasard de la première unité disponible, on peut opposer la recherche de l'unité qui donne la meilleure répartition de la charge des canaux.

— Espace secondaire sur disques à têtes mobiles : ici encore, à l'allocation au hasard, on peut opposer la méthode qui consiste à essayer d'allouer des emplacements proches de la position courante des têtes de lecture-écriture.

Remarquons qu'une solution donnant de bons résultats dans certaines conditions de fonctionnement peut se trouver mal adaptée dans d'autres. Le dernier exemple illustre assez bien ce fait : si la demande d'espace secondaire est effectuée par un processus au moment où il veut écrire sur cet emplacement (cas d'une allocation dynamique piste par piste, par exemple), l'allocation « sous les têtes » donnera de bons résultats ; c'est par contre un raffinement superflu si l'allocation d'espace à un fichier est effectuée globalement au début d'un travail.

4.142 Traitement d'une demande

Quelle que soit l'origine d'une demande de ressource et la manière dont elle a été formulée, cette demande finit par être présentée à l'allocateur correspondant qui peut, soit l'honorer immédiatement, soit la mettre dans une file d'attente pour un réexamen ultérieur, soit encore la rejeter.

a) Une demande est honorée et le catalogue de la ressource mis à jour, s'il est possible de la satisfaire, au besoin par réquisition, et s'il n'existe pas, par ailleurs, de conditions de non-allocation. De telles conditions correspondent au cas où il a été décidé de surseoir à toute nouvelle demande en provenance d'un demandeur.

Exemples

— Dans la gestion de l'espace secondaire, on peut tenir compte d'un coefficient de remplissage maximum au-delà duquel aucune allocation n'est plus effectuée, jusqu'à ce que ce coefficient soit redescendu à une certaine valeur.

— Dans la méthode de la « régulation de la charge » (cf. 4.62), l'observation d'un taux de pagination atteignant un certain seuil conduit le système à refuser l'introduction d'un nouveau travail en mémoire.

b) Une demande est mise en file d'attente si la ressource n'est pas disponible ou si des conditions particulières ne permettent pas de l'honorer. La file d'attente est constituée d'éléments identifiant le demandeur et les paramètres de la demande.

L'allocateur peut intervenir à nouveau dans trois circonstances pour tenter de satisfaire une ou plusieurs demandes de la file :

— lorsque l'état de la ressource est modifié à la suite d'une libération effectuée par l'un des allocataires,
— lorsqu'un événement extérieur donne le signal d'un réquisition possible,
— lorsqu'une condition de non-allocation disparaît.

Les instants d'intervention de l'allocateur font partie de la stratégie d'allocation.

c) Une demande est rejetée lorsqu'elle ne peut être satisfaite immédiatement et qu'il n'est pas prévu de file d'attente pour la ressource demandée. Ce rejet n'entraîne pas le blocage du demandeur, qui peut, au contraire, examiner la décision prise par l'allocateur et réagir en conséquence. Une telle forme de demande correspond en fait à un test de disponibilité de la ressource, couplé avec une allocation si la réponse est positive ; deux opérations sont ainsi incluses automatiquement dans une section critique (cf. Chap. 2).

Exemple. Dans un système conversationnel, où l'utilisation des bandes magnétiques est en général exceptionnelle, les dérouleurs ne sont pas alloués automatiquement à un utilisateur ; ce dernier, au moment où il va avoir besoin d'un dérouleur, peut en faire la demande au système par une commande particulière ; si un dérouleur est libre,

il lui est alloué, et un message lui indique qu'il peut maintenant l'utiliser ; si aucun dérouleur n'est libre, cela lui est également signalé par un message, et il peut alors décider de faire autre chose.

4.15 PRÉSENTATION DU CHAPITRE

Dans la suite de ce chapitre, nous introduirons la notion de charge d'un système pour caractériser la demande de ressources ; une bonne connaissance des caractéristiques de la charge permet d'améliorer le service rendu, dans le sens des critères retenus par les réalisateurs du système (temps de réponse, débit des programmes, etc...).

Ensuite, nous étudierons les stratégies d'allocation applicables pour le processeur, pour la mémoire lorsqu'elle est allouée par zones, pour la mémoire paginée et pour la mémoire secondaire. Dans chacun de ces cas, nous ne considérons qu'une seule ressource ; les stratégies proposées sont des **stratégies individuelles** qui ne tiennent pas compte des besoins du demandeur en autres ressources.

Ces stratégies individuelles sont très variées et dépendent de la nature de la ressource ; cependant, à un niveau d'allocation donné, par exemple au niveau des processus, il est souhaitable que les ressources spécifiques de ce niveau soient allouées de manière cohérente au moyen d'une **stratégie globale,** afin d'éviter les effets antagonistes de stratégies individuelles indépendantes. Après avoir mis en évidence le phénomène d'écroulement du système, qui peut apparaître lorsque le processeur et la mémoire paginée sont alloués au moyen de stratégies individuelles, nous présenterons des stratégies globales pour l'allocation combinée de ces deux ressources.

Enfin, nous introduirons le phénomène de **l'interblocage** : un groupe de processus est en interblocage si chaque processus attend une ressource allouée à un autre processus du groupe, et possède lui-même des ressources attendues par d'autres processus du groupe. Nous proposerons des solutions concernant la prévention, la détection et la guérison de l'interblocage.

4.2 CARACTÉRISTIQUES DE LA CHARGE D'UN SYSTÈME

4.21 INTRODUCTION

La **charge** d'un système désigne l'ensemble des demandes de ressources présentées à un instant donné par les travaux des utilisateurs. Les caractéristiques de cette charge sont connues du système de deux manières :

— à partir d'indications explicites fournies par le programmeur : paramètres mentionnés sur la carte de début d'un travail, ou bien indiqués lors d'une commande de travail conversationnel,

— à partir de mesures faites par le système lui-même à des instants plus ou moins rapprochés au cours du déroulement des processus (distribution des références à la mémoire, temps d'exécution, etc...).

Par ailleurs il est utile de constituer des ensembles de programmes formant des **charges-types** (cf. Chap. 6) pour faire des comparaisons entre systèmes, et évaluer leurs qualités.

Il est commode de distinguer deux niveaux d'observation de la charge :

— un niveau global ou macroscopique se situant à une échelle de temps correspondant à la durée d'un travail,

— un niveau fin dont l'échelle de temps est celle de la durée d'une instruction.

A ce dernier niveau, la ressource à laquelle on s'intéresse est la mémoire ; en effet, la dispersion des références qu'engendre le déroulement d'un processus a une incidence importante sur l'efficacité des algorithmes d'allocation de mémoire.

4.22 CARACTÉRISTIQUES GLOBALES DE LA CHARGE

Les courbes et les résultats de mesures de ce paragraphe ne constituent que des exemples particuliers ; en effet, de grandes variations existent d'un système à un autre en fonction des matériels et des types d'utilisateurs. De plus, pour un système donné, la charge peut dépendre de la période d'utilisation.

Nous tentons de montrer comment chaque type de mesures peut être exploité pour la conception d'un système, pour sa génération ou pour la prise en compte des travaux. L'utilisation des mesures pour l'auto-adaptation du système à la charge sera évoquée au paragraphe 4.6.

Examinons successivement quelques caractéristiques des demandes de ressources.

1) Taille de mémoire principale

L'exemple de la figure 1 montre la distribution des besoins des travaux en mémoire principale sur un calculateur géré en monoprogrammation [Le Faou, 73].

Ces résultats font apparaître une dispersion importante des tailles de mémoire nécessaires. Si l'on envisageait l'exécution de ces mêmes travaux avec un système multiprogrammé, il serait vraisemblablement plus efficace de prévoir des partitions de taille variable. Dans le même ordre d'idées, des mesures [Batson, 70] faites sur une machine BURROUGHS B5500 donnent une idée de la répartition et de l'utilisation de segments de différentes tailles (limitées à 1 023 mots) (Fig. 2).

La connaissance de la taille des travaux peut être utilisée dans l'écriture des algorithmes de gestion de la mémoire et dans le choix des paramètres de génération du système.

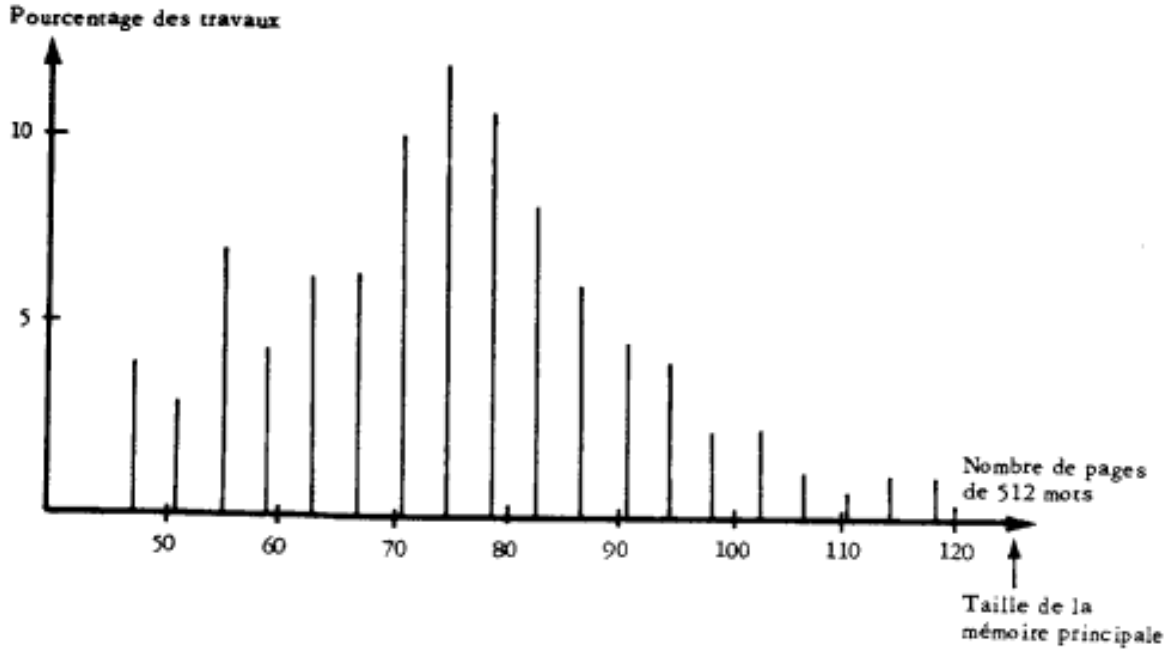


Figure 1. Utilisation de la mémoire principale [Le Faou, 73].

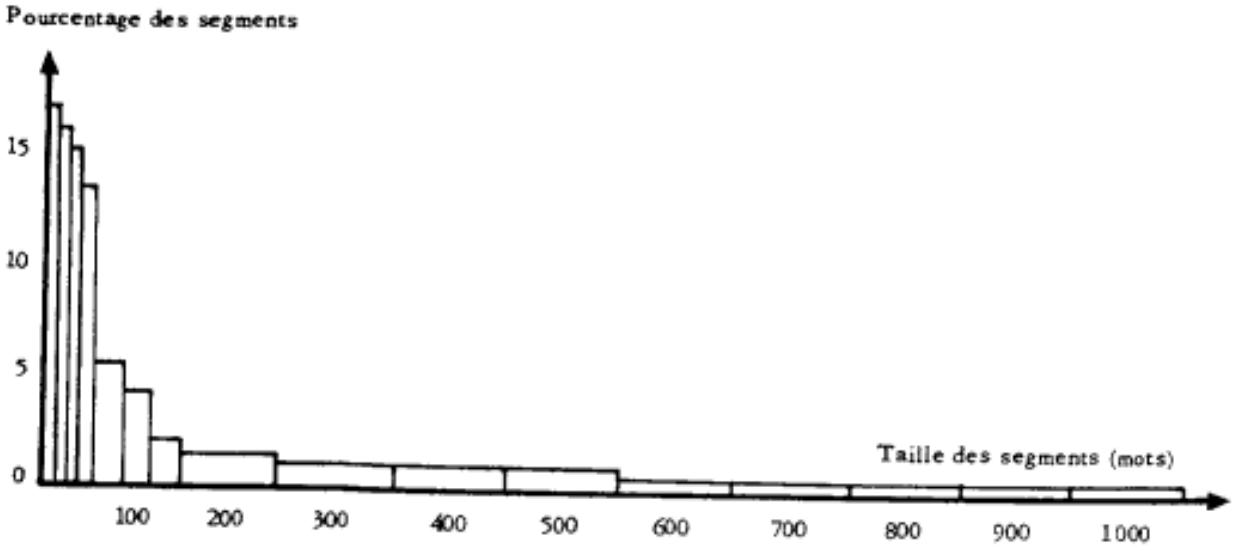


Figure 2. Distribution pour l'ensemble des segments en usage [Batson, 70].

2) Temps d'unité centrale

Les figures 3 et 4 permettent de comparer pour le système MTS [Arden, 69] les distributions des temps de calcul des travaux non interactifs et interactifs.

La connaissance du temps de calcul (ou d'une borne supérieure de ce temps) peut servir à la conception d'algorithmes d'allocation de l'unité centrale. Ainsi les résultats des figures 3 et 4 peuvent contribuer au choix d'un quantum (cf. 4.3) si on désire favoriser les travaux interactifs.

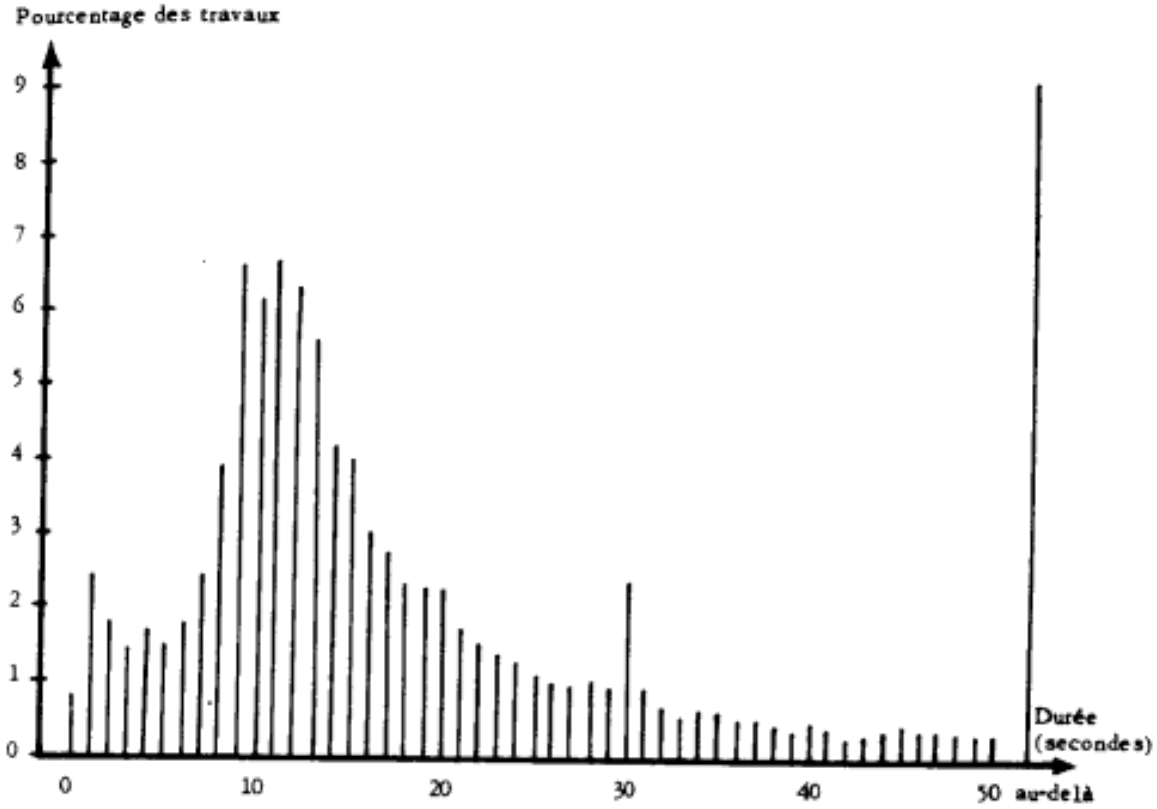


Figure 3. Temps de calcul des travaux non interactifs [Arden, 69].

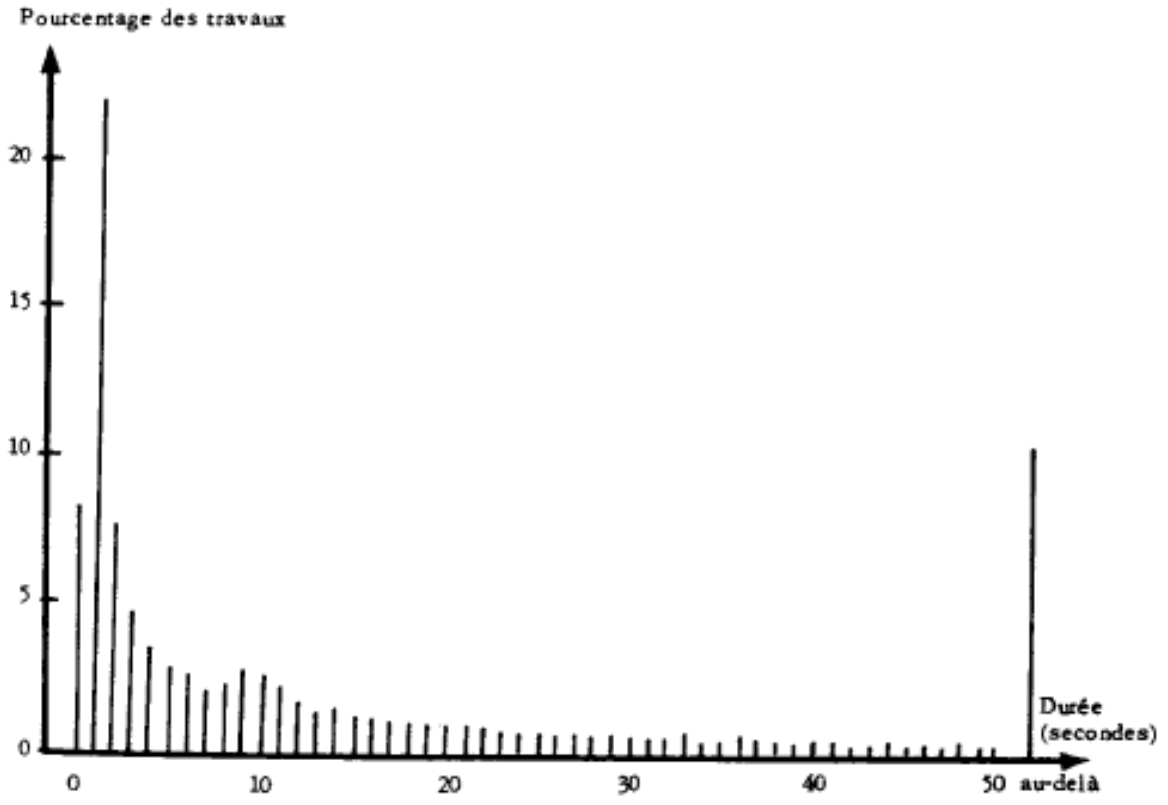


Figure 4. Temps de calcul des travaux interactifs [Arden, 69].

3) Comportement de l'utilisateur conversationnel

Pour un utilisateur conversationnel, on définit le **temps de réflexion** comme l'intervalle de temps entre le début de la réception d'une réponse à une commande et l'envoi d'une nouvelle commande. La figure 5 [Scherr, 65] montre un exemple de distribution de ce paramètre dans le système CTSS.

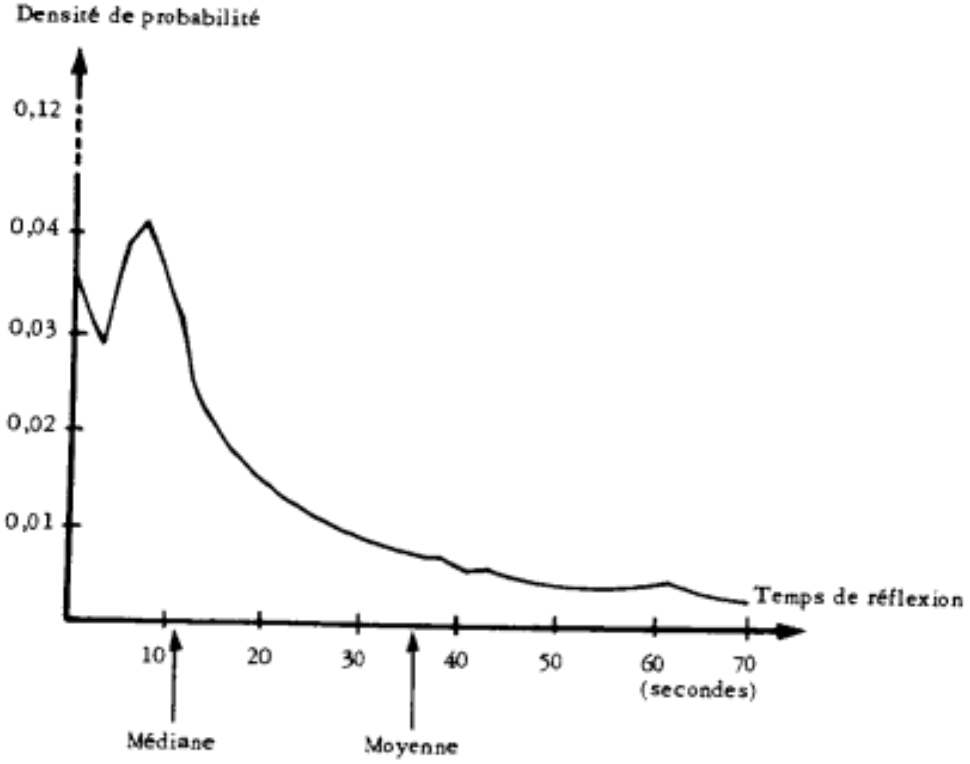


Figure 5. Distribution des temps de réflexion des utilisateurs [Scherr, 65].

Remarques. Le pic qui est au début de la courbe correspond à l'envoi de messages très brefs (retour-chariot) ; de même il faut compléter la courbe par le point de densité de probabilité 0,12 et de temps de réflexion nul ; ce point correspond à l'exécution, provoquée par un seul message, d'un ensemble de commandes pré-enregistrées.

Le temps de réflexion a une valeur relativement élevée, ce qui conduit à libérer la mémoire occupée par l'utilisateur en cours de réflexion.

4) Entrées-sorties

Dans un système multiprogrammé, il est intéressant de connaître les besoins relatifs des programmes, en entrées-sorties et en calcul, de manière à équilibrer la charge des différents processeurs.

Les courbes de la figure 6 montrent une grande disparité entre les comportements de deux compilateurs [Leroudier, 73] ; on pourrait équilibrer la charge des canaux d'entrée-sortie en multiprogrammant des compilations de programmes en FORTRAN et en PL/1.

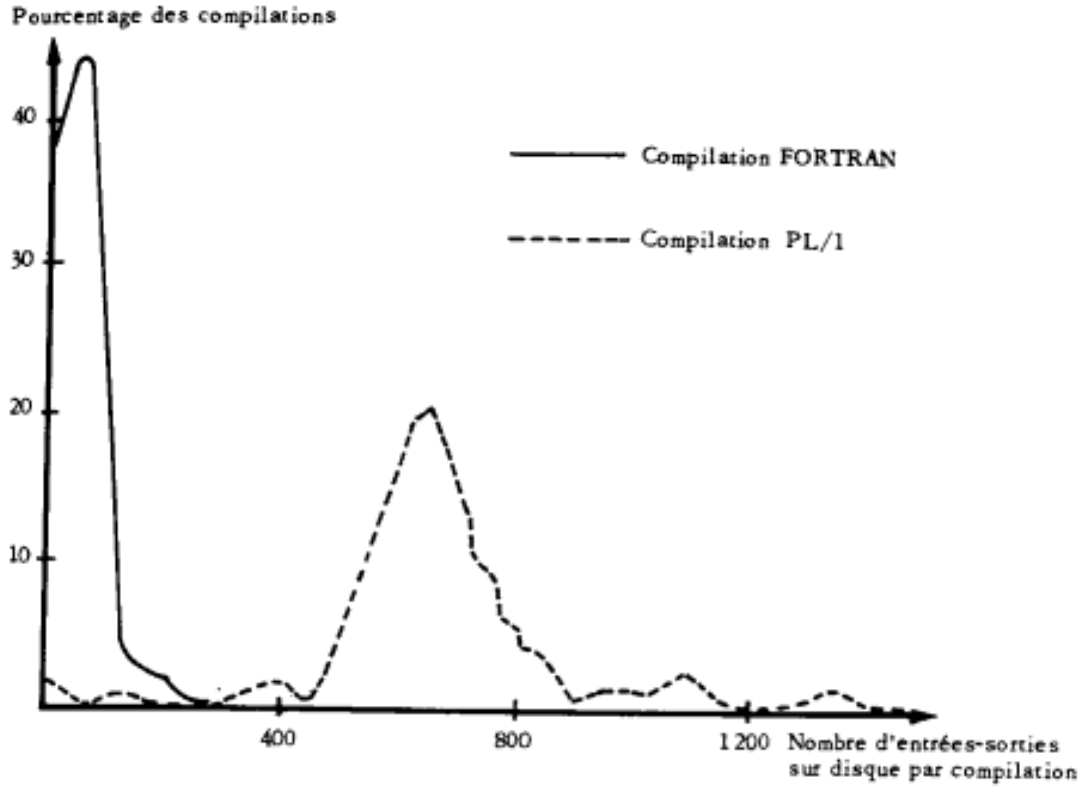


Figure 6. Nombre d'entrées-sorties sur disque pour des compilations FORTRAN et PL/1 [Leroudier, 73].

Le tableau qui suit présente, pour le système CP/CMS, les besoins en entrées-sorties des différentes classes de travaux soumis par les utilisateurs.

	Manipulations de fichiers	Editions de fichiers	Chargements Exécutions	Assemblages Compilations	Divers
Nb. E/S disques					
— Moyenne	19	65	78	490	34
— Ecart-type	70	120	150	800	240
Nb. E/S console					
— Moyenne	1,5	30	10	9	5,5
— Ecart-type	10	50	45	40	35
Temps UC (s)					
— Moyenne	0,1	0,5	3,4	8	0,1
— Ecart-type	0,5	1,3	30	11	1,5

Figure 7. Statistiques des entrées-sorties selon les types de travaux [Leroudier, 73].

Les valeurs moyennes sont en général peu représentatives car les écarts-types sont grands.

5) Taux d'utilisation des fonctions du système

Il est utile de connaître la fréquence relative d'emploi des diverses fonctions du système, chaque fonction étant caractérisée par un usage relativement stable des différentes ressources. Le tableau de la figure 8 donne un exemple de ce type de renseignements.

	Manipulations de fichiers	Editions de fichiers	Chargements Exécutions	Assemblages Compilations	Divers
Pourcentage des commandes	37	12	17	5	21
Pourcentage unité centrale	8	10	48	32	2,4
Pourcentage temps utilisateur	5	52	28	12	3

Figure 8. Statistiques des temps de calcul selon les types de travaux [Leroudier, 73].

Remarque. De même que dans l'exemple de la figure 5, on doit compléter la première ligne par une fraction de 8 % correspondant à l'exécution de commandes pré-enregistrées.

Les résultats de ce tableau montrent en particulier que les travaux d'édition de fichiers, auxquels les utilisateurs consacrent une fraction importante de leur temps, constituent une charge relativement faible pour le système ; ce renseignement contribue à l'évaluation du nombre de terminaux connectables.

4.23 COMPORTEMENT DYNAMIQUE DES PROGRAMMES. PROPRIÉTÉ DE LOCALITÉ

L'exécution d'un programme se traduit par une suite de références à des instructions et à des données. Nous considérons ces instructions et ces données comme occupant des emplacements dans un espace d'adressage linéaire unique (mémoire virtuelle), indépendamment de leur support physique. Nous considérons en outre que la mémoire virtuelle est découpée en blocs égaux d'emplacements consécutifs, et nous nous intéressons à la suite des références aux blocs. Pour un découpage en blocs d'une taille donnée, l'exécution d'un programme comprenant n blocs numérotés de 1 à n se traduit par une suite de références aux blocs :

$$\omega = r_1\ r_2\ \ldots\ r_i\ \ldots \quad \text{avec} \quad r_i = 1, 2, \ldots, n$$

où r_i est le numéro du bloc contenant la i-ième référence à la mémoire.

L'expérience montre que la suite des références possède les propriétés suivantes, vérifiées pour un grand nombre de programmes [Spirn, 72].

1) La distribution des références n'est pas uniforme sur l'ensemble de la mémoire : l'histogramme des références à la mémoire faites pendant l'exécution d'un programme présente généralement des pics marqués, correspondant aux zones le plus fréquemment utilisées (boucles, données localisées).

2) On peut diviser la suite des références en sous-suites consécutives ω_i (non nécessairement de même longueur), telles que si L_i est l'ensemble de blocs distincts référencés dans la sous-suite ω_i :

a) la distribution des références dans ω_i n'est pas uniforme ; de plus, le nombre de blocs de L_i est souvent petit par rapport au nombre total n de blocs du programme.

b) L_i et L_{i+1} possèdent généralement beaucoup de blocs en commun.

c) L_i et L_{i+j} tendent à devenir non corrélés pour de grandes valeurs de j : la connaissance de L_i donne peu d'information sur la composition de L_{i+j} pour de grandes valeurs de j.

On exprime ces propriétés en disant que la plupart des programmes présentent la **propriété de localité.** Cette propriété est liée à une certaine structuration, fréquemment rencontrée, des instructions et des données (instructions exécutées en séquence, boucles, accès regroupés aux données). Lorsque la structuration est différente (cas des données organisées en listes où les accès sont dispersés), la propriété de localité peut ne plus être vérifiée pour le programme ou pour les données.

Remarque. L'existence de la propriété de localité est, dans une large mesure, indépendante de la taille des blocs constituant la mémoire virtuelle. L'exercice 1 présente un modèle de programme à comportement local.

L'existence de la propriété de localité se traduit pratiquement par les conséquences suivantes :

1) Considérons un programme dont les instructions et les données sont initialement contenues dans une mémoire secondaire. Pour exécuter ce programme, il faut transférer ces instructions et ces données en mémoire principale ; nous supposons que ce transfert se fait par blocs de taille fixe (pages) et « à la demande » (cf. 4.45) : une page n'est chargée en mémoire principale qu'au moment du premier accès à l'information qu'elle contient ; cet accès provoque un déroutement **(défaut de page).** Si on considère l'évolution du nombre de défauts de page en fonction de la taille s de la mémoire principale, on constate que :

a) La courbe représentant en fonction de s le temps moyen $e(s)$ séparant deux défauts de page consécutifs n'est pas une droite (Fig. 9), mais a une allure en S. La seuil en deçà duquel l'intervalle entre défauts de page devient faible peut être notablement inférieur à la taille totale du programme.

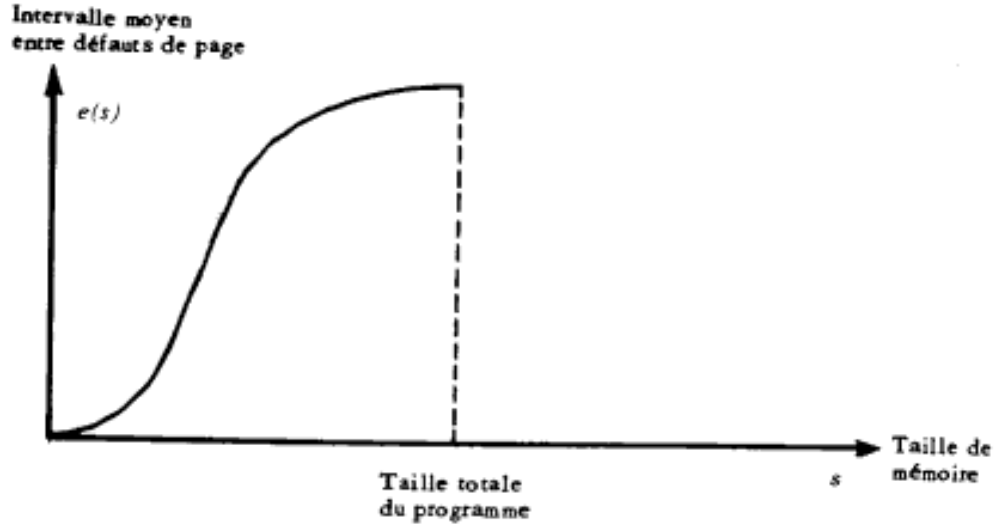


Figure 9. Intervalle moyen entre défauts de page en fonction de la taille de mémoire principale [Belady, 69].

Ce comportement peut s'expliquer en remarquant que, lorsque la mémoire est assez grande pour contenir le nombre de blocs distincts figurant dans un L_i donné, les défauts de page sont provoqués par le renouvellement de la population de L_i, qui, d'après la propriété 2*b*), est lent. Au contraire, le nombre de défauts de page devient élevé quand la mémoire est insuffisante pour contenir la totalité de L_i à un instant donné.

b) Un phénomène analogue est illustré par le résultat d'expériences faites sur le système IBM M44/44X [Brawn, 68]. La grandeur mesurée est ici le nombre de pages chargées depuis la mémoire secondaire. Lorsque s diminue, on constate que ce nombre croît d'abord très lentement, puis augmente très vite au-dessous d'un certain seuil (Fig. 10).

c) Les résultats donnés en *a*) et *b*) dépendent, en toute rigueur, de l'algorithme de remplacement utilisé pour la gestion de la mémoire (c'est-à-dire du choix de la page de mémoire principale à allouer lors d'un défaut de page, lorsque toutes les pages sont occupées). En fait, on constate que les résultats indiqués en *a*) et *b*) dépendent peu de l'algorithme de remplacement utilisé. En revanche, la manière dont le programme est écrit (ou produit par un compilateur) peut avoir une influence notable. Ainsi, en réduisant la dispersion des références, on peut réduire de façon importante la taille de mémoire nécessaire à une exécution efficace du programme [Brawn, 68].

2) Comme conséquence de la propriété 2*b*), les références faites par un programme dans un passé récent à ses instructions et à ses données sont souvent une bonne estimation des références qui seront faites dans un avenir

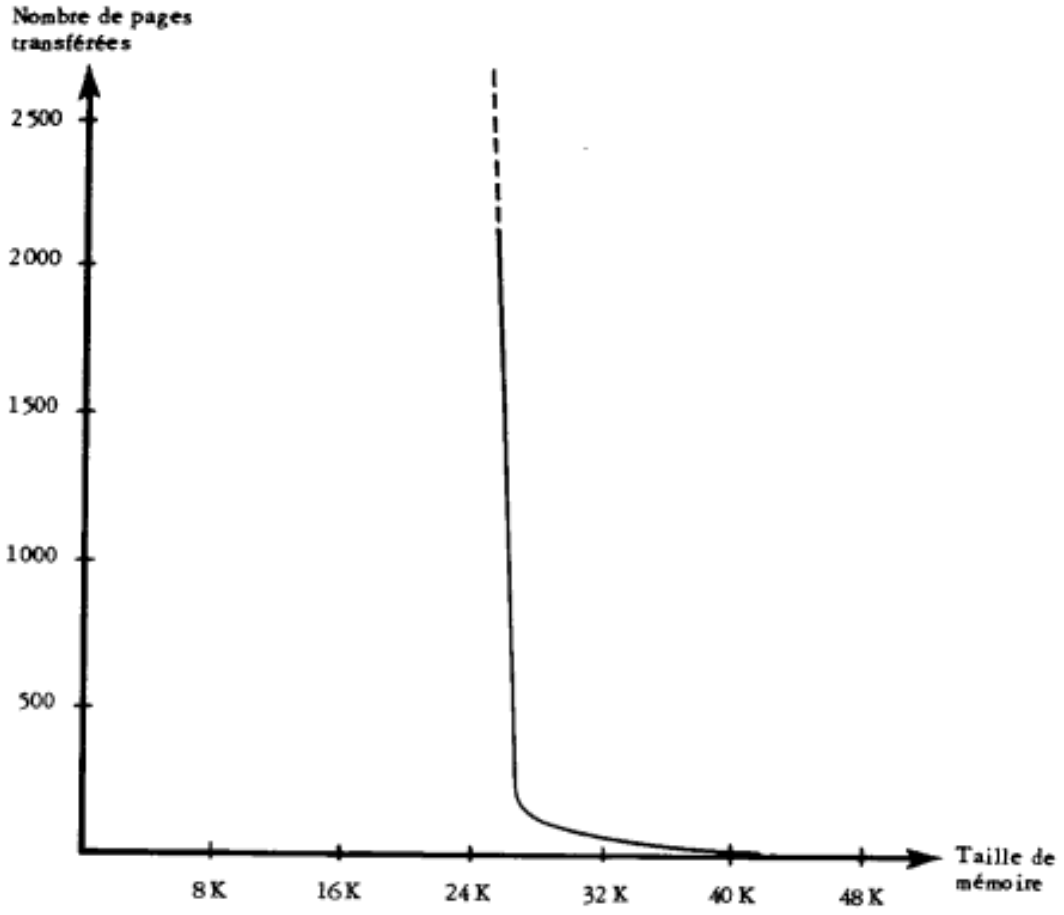


Figure 10. Variation du nombre de transferts de pages avec la taille de mémoire principale pour le système IBM M44/44X [Brawn, 68].

proche. Cette remarque (qui sera développée en 4.63) est à la base de l'usage des **antémémoires** (mémoires très rapides et de petite taille dans lesquelles sont conservées, à tout instant, les informations le plus récemment référencées). L'expérience montre qu'il y a une probabilité élevée pour qu'une nouvelle référence concerne une information présente dans l'antémémoire ; ainsi, le temps d'accès apparent à l'information est plus voisin du cycle de l'antémémoire que du cycle de la mémoire principale.

Exemple. Dans une série d'expériences de simulation destinées à évaluer l'efficacité de l'antémémoire sur l'ordinateur IBM 360/85 [Liptay, 68], on a mesuré, pour un ensemble de programmes constituant une charge type, la fréquence relative des accès pour lesquels l'information recherchée se trouvait dans l'antémémoire. Pour l'ensemble des programmes considérés, de taille comprise entre 16 K et 218 K octets, la fréquence relevée varie entre 92 % et 99 %, selon les programmes. L'antémémoire comportait 256 blocs de 64 octets et était gérée de manière à contenir à tout instant les 256 blocs contenant les références les plus récentes.

4.3 ALLOCATION DE PROCESSEUR RÉEL

Le problème de l'allocation de processeur peut être schématisé comme suit : au cours du temps, suivant une certaine loi d'arrivée, des processus demandent les services d'un processeur du système ; pour simplifier l'algorithme d'allocation, nous nous limitons au cas où tous les processeurs sont identiques. Le temps d'exécution d'un service demandé est connu ou non au moment de la demande.

L'objectif d'une stratégie d'allocation est la satisfaction des demandes, tout en respectant les contraintes imposées. Celles-ci peuvent être, entre autres :

— la garantie à chaque processus d'un temps donné d'allocation,
— le respect d'un ordre de priorité entre les processus demandeurs,
— l'exécution totale d'un processus avant une heure limite fixée à l'avance,
— l'annulation du travail de tout processus qui tente d'utiliser le processeur pendant un temps supérieur au maximum fixé par le programmeur ou le système.

4.31 CLASSIFICATION DES STRATÉGIES

Avec les notations de la figure 11, une stratégie d'allocation de processeur est définie par les opérations suivantes :

① : entrée d'un processus demandeur,
② : sélection d'un processus demandeur,
③ : sélection d'un processeur,
④ : interruption du service.

Les opérations ②, ③, ④ sont nécessairement liées : lorsqu'un processeur cesse de servir un processus donné, un processus demandeur peut (et donc doit) être servi. Nous ne nous intéressons pas à la sélection d'un processeur, ni à la façon dont les processus deviennent initialement candidats à l'allocation de processeur.

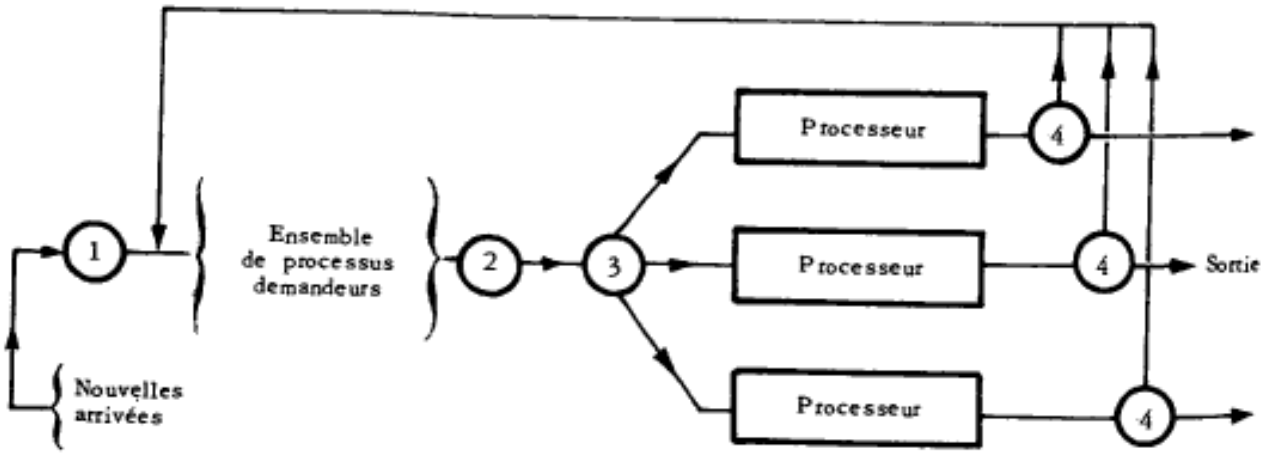


Figure 11. Schéma général de l'allocation de processeur.

Une première distinction peut être faite entre les stratégies mettant en jeu ou non la réquisition du processeur. Un second critère de classification des différentes stratégies est la nature des informations utilisées par celles-ci.

A côté des stratégies entièrement fixées *a priori*, on peut distinguer les stratégies utilisant des informations de diverses origines :

— informations attachées aux processus utilisateurs,
— informations recueillies au cours de l'activité du système.

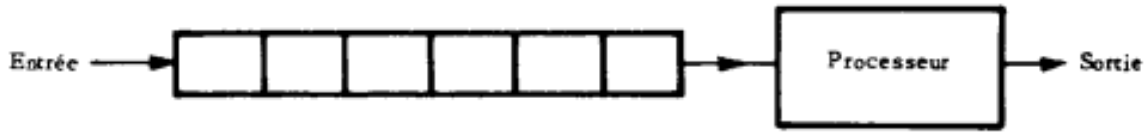
Les stratégies qui vont être examinées visent en général à réduire le temps de réponse (temps écoulé entre ① et la sortie) pour les utilisateurs dont le service exige un temps d'exécution court au détriment des utilisateurs qui ont des demandes plus importantes.

4.32 STRATÉGIES SANS RECYCLAGE DES TRAVAUX

La valeur exacte du temps d'exécution est en général inconnue au moment de la demande. On utilise une valeur estimée *a priori*, fournie ou non par le programmeur.

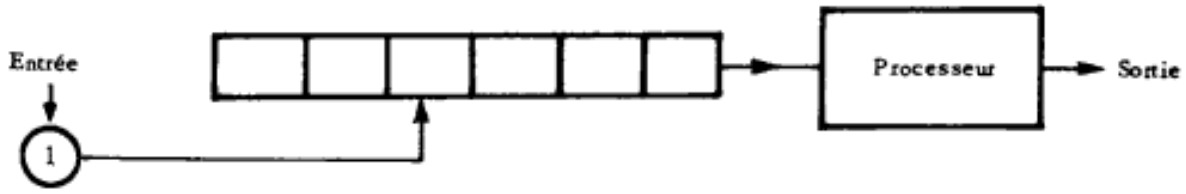
a) File d'attente simple

A titre de comparaison le schéma ci-dessous montre la stratégie la plus simple dite **file d'attente simple** ou FIFO (« First In, First Out », ou premier arrivé, premier servi). Cette stratégie ne tient aucun compte des temps d'exécution.



Elle est couramment utilisée dans les systèmes de traitement par train. Les travaux courts ont un temps de réponse élevé s'ils arrivent après des travaux longs.

b) File d'attente ordonnée suivant le temps estimé d'exécution



Quand un nouveau processus arrive, il est placé dans la file à l'endroit correspondant à son temps estimé d'exécution ; la file est ordonnée suivant les temps d'exécution croissants ; l'estimation est faite *a priori* et aucune correction n'est tentée en cours d'exécution. Le temps de réponse des travaux courts est ainsi diminué, mais les travaux longs sont retardés (indéfiniment

si le débit d'arrivée des travaux courts est assez grand). Cette stratégie peut être combinée avec un système de priorités croissantes avec le temps d'attente et un temps limite d'utilisation du processeur.

On peut également, avec cette méthode, utiliser une règle de réquisition du processeur : quand un nouveau travail arrive, son temps estimé d'exécution est comparé au temps estimé restant pour le travail en cours ; s'il est plus faible, le travail en cours est interrompu, et cède la place au nouvel arrivant ; le travail interrompu retourne dans la file d'attente, à l'emplacement correspondant à son temps restant d'exécution, c'est-à-dire en tête des processus qui attendent.

La réquisition accentue encore l'avantage donné aux travaux courts ; par contre à chaque interruption, il y a une perte de temps pour la gestion des vecteurs d'état.

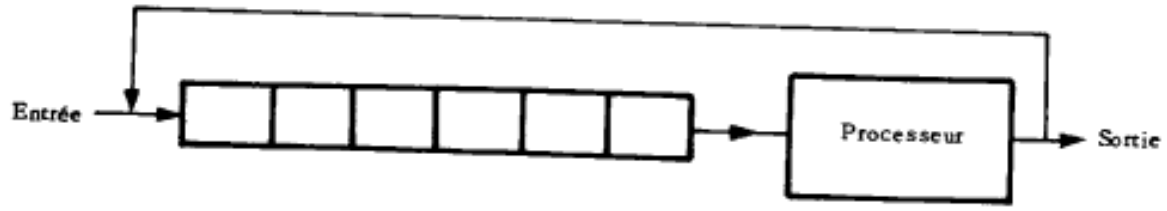
Par rapport à la méthode de la file d'attente simple, les deux stratégies précédentes abaissent la moyenne du temps de réponse des travaux courts, mais augmentent la dispersion autour de la moyenne globale. Elles présentent l'inconvénient de devoir se fier uniquement à un temps d'exécution estimé, qui peut être incorrect ou falsifié par des usagers de mauvaise foi.

4.33 STRATÉGIES AVEC RECYCLAGE DES TRAVAUX

Les méthodes précédentes sont peu adaptées aux conditions des systèmes conversationnels, où les demandes sont fréquentes et où les conséquences d'une erreur d'estimation peuvent être graves pour la qualité du service rendu. Dans une méthode avec recyclage, on ne poursuit pas l'exécution d'un processus jusqu'à son terme : au bout d'un temps très court, de l'ordre de quelques millisecondes, on interrompt ce dernier pour allouer le processeur à un autre processus ; les processus interrompus sont placés en file d'attente. Le nombre de ces files et la manière dont elles sont gérées distinguent les diverses stratégies développées dans ce paragraphe.

a) Balayage cyclique simple ou tourniquet

Dans la stratégie du **tourniquet** (« round robin »), le processeur est alloué successivement à chaque processus ; si au bout d'un temps fixé q, le processus ne s'est pas terminé, il est interrompu et placé en queue de la file des demandeurs.



L'intervalle de temps q, appelé **quantum,** est le paramètre de cette stratégie. Si q tend vers l'infini, on retrouve la stratégie de la file d'attente simple. Quant au modèle obtenu en faisant tendre q vers zéro, il a été étudié analytiquement

en raison de son intérêt théorique : tout s'y passe comme si chaque processus était servi (dès son arrivée) par un processeur dont la vitesse de calcul était divisée par le nombre de processus dans la file [Coffman, 68].

La méthode du tourniquet garantit que tout travail est servi au bout d'un temps fini. Son principal avantage est de limiter le délai de prise en compte (temps écoulé entre ① et ③), ce délai dépendant d'ailleurs de la charge. Dans un système en temps partagé, où la plupart des interactions usagers-système sont très brèves, le temps de réponse est du même ordre que le délai de prise en compte quand le quantum est fixé à une valeur telle que la plupart des interactions soient terminées en un seul quantum.

b) Recyclage à plusieurs files d'attente

On souhaite parfois accorder aux travaux courts un privilège plus grand que ne le permet la stratégie du tourniquet. Dans ce but, on introduit plusieurs files d'attente avec des valeurs de quantum éventuellement différentes.

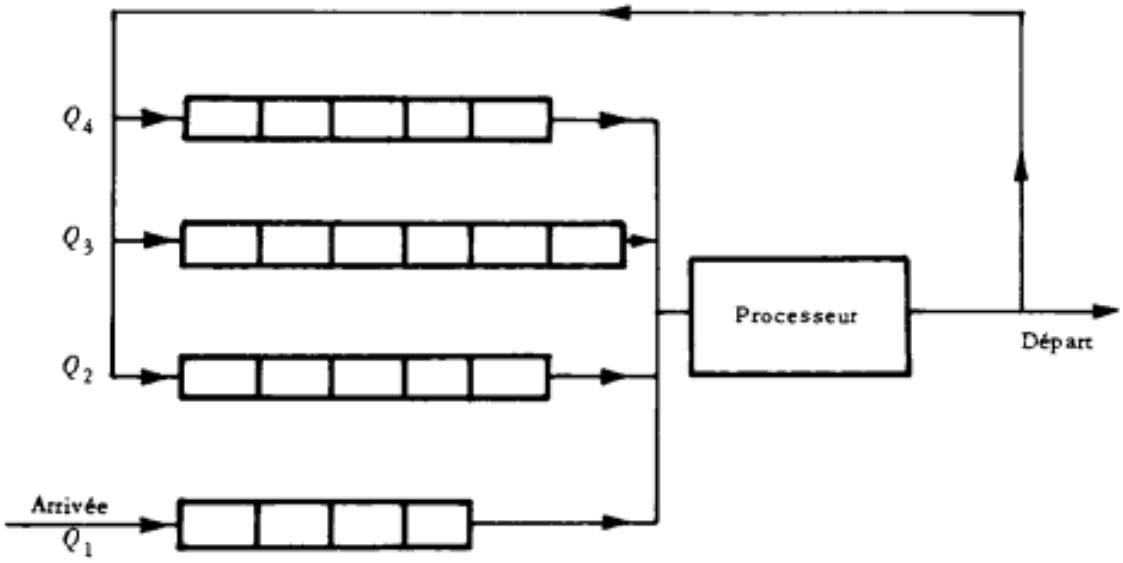


Figure 12. Recyclage à plusieurs files d'attente.

Les processus demandeurs sont rangés dans n files $Q_1, Q_2, \ldots, Q_n$. A chaque file Q_i est associé un quantum de temps q_i. Les nouveaux arrivants entrent dans la file Q_1. Le travail situé en tête de la file Q_i $(i > 1)$ ne peut être pris en compte que si toutes les files Q_j, $0 < j < i$ sont vides. Si un travail provenant de Q_i n'est pas terminé au bout du temps q_i, il passe en queue de Q_{i+1} ; les travaux sortant de Q_n y retournent. Enfin tout nouveau travail qui arrive dans la file Q_1 pendant l'exécution d'un travail de la file Q_i $(i > 1)$ est pris en compte dès l'expiration du quantum q_i. Dans la pratique, cette stratégie est généralement assortie d'une réquisition qui permet la prise en compte immédiate d'un travail arrivant dans une file Q_1 vide.

Les stratégies à plusieurs files permettent un ajustement plus souple que celles évoquées précédemment.

Exemple 1. $n = 2$, $q_1 = q_2 = q$.

C'est le cas le plus simple : il y a une file haute et une file basse, le choix étant défini par le fait que le temps d'exécution est plus grand ou plus petit que q. En faisant varier q, on peut privilégier une classe plus ou moins étroite de travaux.

Exemple 2. n quelconque, $q_k = 2^{k-1} q_1$.

Cette stratégie a été utilisée dans le système CTSS, n étant égal à 8 [Corbató, 62].

4.34 STRATÉGIES FONDÉES SUR LA NOTION DE PRIORITÉ

La priorité est un nombre attaché à chaque travail et définissant le degré d'urgence attribué à son exécution : elle est dite externe si elle est fixée avant la prise en charge (par exemple sur une carte de contrôle), et interne si elle est fixée ou modulée par l'algorithme de gestion des travaux. Un travail de priorité p_1 doit être pris en compte avant un travail de priorité p_2 si $p_2 < p_1$; il peut y avoir ou non réquisition.

Une stratégie fondée sur les priorités peut être mise en œuvre, soit en constituant une file unique où les travaux sont ordonnés d'après leur priorité, soit en constituant plusieurs files de travaux de même priorité, une file n'étant servie que lorsque les files de plus forte priorité sont vides. On peut également introduire la notion de priorité dans une stratégie avec recyclage, la priorité d'un travail diminuant après chaque quantum alloué à ce travail.

4.35 STRATÉGIE FONDÉE SUR LA NOTION D'ÉCHÉANCE

Dans certains cas (contrôle de processus industriels, simulation en temps réel, établissement de la paie d'une entreprise), on est amené à spécifier une **échéance** (« deadline »), c'est-à-dire une date limite à laquelle l'exécution d'un certain travail doit être terminée.

Pour que cette contrainte puisse être respectée, on doit connaître la durée d'exécution d'un tel travail ou tout au moins une limite supérieure. On est alors capable de déterminer pour les différents travaux comportant une échéance, l'heure avant laquelle ils doivent être entrepris pour respecter cette échéance. Le calcul doit tenir compte du fait qu'il peut exister plusieurs travaux de ce type, ainsi que d'autres travaux ne comportant pas cette contrainte. On peut alors se ramener à une stratégie à priorités en affectant aux travaux une priorité qui croît à mesure que l'heure d'échéance se rapproche. Quand un travail ne peut être terminé avant son échéance, deux stratégies sont possibles :

— détruire le travail, s'il n'a alors plus d'intérêt (traitement de mesures en temps réel),

— retarder l'échéance (paie d'une entreprise).

4.4 GESTION DE LA MÉMOIRE PRINCIPALE

4.41 INTRODUCTION

Nous regroupons sous le terme général de « gestion de la mémoire » tout ce qui concerne la gestion des unités servant de support à des informations. Nous nous intéressons seulement aux informations directement adressables par un programme en cours d'exécution et nous excluons celles qui sont rendues accessibles à ce programme par des opérations d'entrée-sortie demandées explicitement. Pour qu'une instruction élémentaire s'exécute, ses opérandes doivent se trouver en mémoire principale ou dans des registres ; de même, les instructions successives sont prélevées par un processeur dans la mémoire principale (la présence de mécanismes d'antémémoire ne change rien à ce fait puisque le processeur n'engendre que des adresses de mémoire principale).

Lorsque la mémoire principale est le seul support existant dans le système pour l'information adressable, on dit que l'on a une **mémoire uniforme** ; toute information qui disparaît de la mémoire perd définitivement la possibilité d'être adressée par un programme ; la taille de l'espace utilisable par un processus est alors inférieure ou égale à la taille de la mémoire principale, et cela quelle que soit la capacité d'adressage de la machine (donnée par le nombre de bits de l'adresse). La gestion d'une mémoire uniforme se ramène au choix d'un algorithme de placement : une fois placées en mémoire, les informations doivent y demeurer jusqu'au moment où elles cessent d'être utilisées. La seule forme possible de partage d'une mémoire uniforme entre plusieurs processus consiste en un découpage de cette mémoire et en une allocation de fractions différentes à chacun des processus.

Lorsque l'on veut partager la mémoire, ou une partie de celle-ci, sur la base d'un multiplexage entre les processus, en pratiquant la réquisition comme cela se fait pour une unité centrale, alors apparaît la nécessité d'une mémoire secondaire destinée à conserver les informations provisoirement chassées de la mémoire principale au profit d'autres informations : on a alors affaire à une **mémoire hiérarchisée.** La hiérarchie des mémoires comporte au minimum deux niveaux (la mémoire principale et une mémoire secondaire constituée généralement de disques ou tambours magnétiques), mais elle peut comporter plus de deux niveaux, dont chacun est en général caractérisé par rapport au précédent par un coût de stockage moindre, une capacité plus grande et un temps d'accès plus important. Dans une telle organisation, il s'agit de répartir l'information entre les niveaux de la hiérarchie de manière à ce que le maximum de références concerne les unités les plus rapides. On arrive ainsi à ce que le temps moyen d'accès à une information s'approche du temps d'accès de l'unité la plus rapide, et cela avec un coût proche de celui des unités les plus lentes [Mattson, 70]. La gestion d'une mémoire hiérarchisée fait intervenir des politiques de **placement** et de **remplacement** en

mémoire principale, d'allocation d'espace secondaire, et, dans le cas de plusieurs niveaux secondaires, de **migration** entre niveaux.

Dans ce qui suit, nous ne considérons que la gestion de la mémoire principale avec et sans mémoire secondaire, en laissant de côté la gestion de la mémoire secondaire elle-même, qui sera abordée en 4.5. Pour définir une gestion adaptée à des circonstances données, il faut en particulier tenir compte de la fréquence des demandes, de la durée d'allocation, des tailles demandées, du volume total de mémoire disponible et des caractéristiques des unités d'échange. Il faut également tenir compte des contraintes technologiques imposées à l'allocateur, notamment de celles liées à l'adressage de la mémoire physique et à la protection des informations qu'elle contient.

4.42 INCIDENCE DES MÉCANISMES D'ADRESSAGE

Examinons les problèmes posés par l'implantation des programmes en mémoire centrale. Un programme adresse un espace logique, ou virtuel ; lors de l'exécution, tout ou partie de cet espace a pour support la mémoire centrale. Une **fonction de topographie** F établit la correspondance entre l'espace virtuel et l'espace réel. La gestion de la mémoire réelle dépend étroitement de la façon dont cette correspondance est réalisée et du moment de sa réalisation [Mc Gee, 65]. Trois cas sont envisageables.

1) Lorsque la correspondance est réalisée lors de l'écriture, de la compilation ou de l'édition de liens du programme, le résultat est un module non translatable qui doit être chargé à une adresse précise, toujours la même, puisque les adresses qu'il contient sont des adresses de mémoire réelle. Dans ces conditions, le chargement d'un tel programme entraîne une demande de mémoire formulée en termes de taille et d'adresse, ce qui laisse peu d'initiatives à l'allocateur ; tout au plus, ce dernier peut-il associer à une zone fixe de mémoire réelle une file d'attente des programmes à charger dans cette zone ; la gestion de cette file dépend des autres ressources réclamées par les programmes. On dit dans ce cas qu'il y a **partition** de la mémoire.

Exemple. Le système DOS/360, qui utilise une mémoire uniforme, fait partie de cette catégorie.

2) Lorsque la correspondance est réalisée au moment du chargement en mémoire réelle, le résultat de la compilation ou de l'édition de liens est un module translatable ; un chargeur est capable de l'implanter n'importe où en mémoire centrale en effectuant une **réimplantation statique** qui fixe définitivement les adresses. L'adresse d'implantation est laissée à l'initiative de l'allocateur, ce qui permet une gestion plus efficace des ressources. Cependant, bien souvent, une réimplantation statique n'est pas suffisante pour permettre une bonne gestion de la mémoire : un programme ayant commencé son exécution ne peut plus être déplacé ; s'il est écrit en mémoire secondaire, il devra retrouver plus tard la même place en mémoire principale.

Exemple. Le système OS/360 pratique la réimplantation statique dans une mémoire qui, sans options particulières, est uniforme. Avec l'option « rollin-rollout », un programme implanté en mémoire peut être recopié sur mémoire secondaire pour autoriser son voisin du dessous à s'étendre en cours d'exécution vers les adresses supérieures. Pour poursuivre son exécution, ce programme doit être ramené à la même place lorsque la zone où il était implanté redevient libre.

3) Lorsque la correspondance est réalisée lors de l'exécution du programme, toute adresse de l'espace virtuel est transformée par la fonction topographique en adresse de l'espace réel au moment où elle est utilisée pour accéder à la mémoire. Cette fonction fournit une adresse réelle ou signale que l'adresse logique n'est associée à aucune adresse réelle. Cette traduction dynamique des adresses autorise la **réimplantation dynamique** (« relocation ») d'un programme au cours de son exécution [Randell, 68 ; Denning, 70].

Deux classes de mécanismes d'adressage permettent la réimplantation dynamique : les registres de base et la pagination. La réimplantation par registres de base impose de maintenir la contiguïté physique du programme réimplanté ; la pagination, au contraire, permet de découper ce programme en morceaux disjoints. A chacun de ces mécanismes correspondent des techniques d'allocation de mémoire particulières.

Dans ce qui suit, nous examinerons les stratégies d'allocation de mémoire aux travaux, en mémoire uniforme et en mémoire hiérarchisée. Elles utilisent deux techniques de gestion de la mémoire réelle : la gestion par zones de tailles variables et la gestion par blocs de taille fixe. Dans ce qui suit, nous réservons le terme de **zone** pour désigner un ensemble de mots consécutifs de taille quelconque.

Pour chacune de ces deux organisations, nous étudierons les mécanismes d'adressage permettant leur mise en œuvre et les techniques d'allocation applicables.

4.43 STRATÉGIES D'ALLOCATION DE LA MÉMOIRE AUX TRAVAUX

Nous nous plaçons d'abord dans le cas où les demandes de mémoire sont formulées, soit par la partie du système chargée de lancer un nouveau travail, soit par un processus d'un utilisateur. Pour améliorer l'utilisation de l'ensemble des ressources de l'installation, la plupart des systèmes pratiquent la multiprogrammation, c'est-à-dire le partage de la mémoire entre plusieurs processus.

4.431 Allocation en mémoire uniforme

Les besoins en mémoire des processus peuvent varier au cours de leur exécution. La solution qui consiste à laisser les processus effectuer eux-mêmes, après leur chargement initial, des demandes et des libérations de mémoire

présente un certain nombre d'inconvénients quand ces demandes sont honorées dans l'ensemble de la mémoire libre du système gérée en fonds commun.

1) Un phénomène d'interblocage (cf. 4.7) peut survenir : tous les processus sont en attente de mémoire, et aucun ne peut en libérer.

2) Cette technique d'allocation conduit à la **fragmentation** de la mémoire : la mémoire libre est divisée en de nombreuses zones disjointes et il peut arriver qu'une demande ne puisse être satisfaite dans une seule de ces zones, bien que l'espace libre total soit suffisant. Une fraction de la mémoire est ainsi inutilisable. La fragmentation pénalise donc tous les travaux.

3) Il faut garder trace, programme par programme, des zones allouées, pour permettre la récupération de l'espace (fin anormale du processus).

4) Rien n'empêche un programme erroné de boucler sur une demande de mémoire et de s'approprier ainsi tout l'espace.

Pour toutes ces raisons, on adopte fréquemment une stratégie plus rigide, en affectant à chaque processus une zone dont la taille représente le maximum estimé de ses besoins et en le contraignant à demeurer dans les limites de cette zone ; après le chargement initial dans la région allouée, il reste dans cette région une partie libre à partir de laquelle sont satisfaites les demandes ultérieures émises par le processus. Pour l'utilisation dynamique de la mémoire, le processus n'est plus alors en concurrence qu'avec lui-même et la fragmentation qu'il provoque est limitée à sa région.

Avec cette approche il existe deux façons de réaliser l'allocation des zones globales :

1) Le découpage de la mémoire en zones peut être défini une fois pour toutes lors de la création du système et modifiable seulement à certains moments (initialisation du système par exemple). Il en résulte parfois une mauvaise adaptation de la taille des zones aux besoins, car le mieux que l'on puisse faire est de choisir la plus petite zone possible.

Exemple. Le système OS/360 MFT utilise ce principe et découpe la mémoire en un certain nombre de zones de taille fixe (« partitions »).

2) La taille de la zone allouée à un programme peut être fixée lors de l'introduction de ce programme en mémoire. Cette solution fait réapparaître le problème de la fragmentation entre les zones mais permet par contre une meilleure utilisation de la mémoire en présence d'une charge variable ; en effet, on peut adapter exactement la taille des zones aux besoins et aussi faire varier dynamiquement le nombre de zones, donc le nombre de processus qui se partagent la mémoire. Nous verrons par ailleurs (4.442) que le compactage peut parfois résoudre le problème de la fragmentation.

Exemple. Le système OS/360 MVT alloue dynamiquement la mémoire par zones de taille variable (« régions »).

4.432 **Allocation en mémoire hiérarchisée**

Une mémoire uniforme n'est pas adaptée au multiplexage de l'unité centrale entre un nombre important de processus. En effet :

— comme ces processus coexistent en permanence dans la mémoire, la taille moyenne de mémoire disponible pour chacun d'eux est d'autant plus faible qu'ils sont plus nombreux,

— les processus inactifs occupent inutilement une partie de la mémoire principale,

— un processus ne peut pas disposer d'un espace d'adressage de taille supérieure à celle de la mémoire réelle.

L'utilisation d'une mémoire hiérarchisée permet de résoudre ces problèmes en multiplexant également la mémoire principale.

1) *Technique du va-et-vient global*

La méthode la plus courante consiste à utiliser le **va-et-vient** (« swapping ») qui consiste à vider sur une unité secondaire (disque, tambour magnétique, ...) le contenu de la zone de mémoire affectée à un processus puis à utiliser cette zone pour un autre processus ; lorsqu'on décide de reprendre l'exécution du premier processus, il faut recharger les informations nécessaires en mémoire centrale. Plusieurs problèmes sont liés à l'utilisation du va-et-vient.

a) Problème des entrées-sorties dans la zone de l'utilisateur. Si des entrées-sorties sont en cours au moment où le système décide de vider un programme, il faut attendre qu'elles se terminent ; pour limiter cette attente, on doit interdire aux utilisateurs de lancer dans leur espace propre des entrées-sorties dont la durée est longue ou indéterminée, comme la lecture d'une ligne sur un terminal ; de telles opérations doivent utiliser une zone de mémoire non soumise au va-et-vient.

b) Problème de la réactivation. Pour reprendre l'exécution d'un processus interrompu, il est nécessaire, en l'absence de mécanisme de réimplantation dynamique, de replacer son programme au même endroit. Cela peut entraîner le vidage de plusieurs autres programmes, à moins que l'on ne fasse du va-et-vient dans une partition fixe.

c) Problèmes de performances. Si chaque processus dispose de toute la mémoire non réservée au système, il n'est pas possible de faire du travail productif pendant les opérations de va-et-vient puisqu'il n'y a pas d'autre processus prêt. Pour que le système soit rentable, il faut alors que la durée d'une opération de va-et-vient soit faible devant l'intervalle de temps entre deux opérations de va-et-vient consécutives. A titre indicatif, il faut environ deux secondes pour vider et restaurer une mémoire de 256 K octets sur un disque de type 2314 relié à un IBM/360.

Si par contre la mémoire est partagée entre plusieurs processus, il est possible, pendant un transfert, d'activer un autre processus ; cependant, il se peut que

les avantages tirés de cette multiprogrammation soient illusoires si les opérations d'entrée-sortie qui réalisent le va-et-vient freinent trop l'unité centrale (cas d'une mise en commun d'organes entre l'unité centrale et les canaux qui entraîne un vol de cycles), ou empêchent le processus élu d'effectuer ses propres entrées-sorties.

2) *Technique du va-et-vient à la demande*

La technique du va-et-vient global amène à charger des parties de programme qui ne serviront pas pendant le quantum suivant et à vider des parties de programme non modifiées pendant le quantum écoulé. Cette remarque conduit à l'idée d'un mécanisme de « va-et-vient à la demande » qui permettrait de déterminer dynamiquement les parties utiles du programme et de ne les charger qu'au moment de leur utilisation.

Si l'on ne dispose pas d'un mécanisme de réimplantation dynamique, la technique du va-et-vient à la demande est d'une mise en œuvre difficile car une information donnée doit toujours être chargée dans les mêmes emplacements physiques ; toutefois, il existe des algorithmes permettant de minimiser les transferts [Hoare, 72a].

Si l'on dispose d'un mécanisme de réimplantation dynamique, on applique selon le cas une stratégie de « zone à la demande » (cf. 4.442) ou de « page à la demande » (cf. 4.453) ; la réimplantation permet alors d'utiliser au mieux la place libre. En particulier, si le mode d'adressage permet de définir des segments, il est naturel d'allouer une seule zone par segment (exemple : BURROUGHS B5500).

4.44 GESTION DE LA MÉMOIRE PAR ZONES

L'allocation de mémoire par zones de taille quelconque suppose l'existence d'un mécanisme permettant la réimplantation dynamique. Ce mécanisme est fourni par l'utilisation de registres de base.

4.441 Réimplantation dynamique par registres de base

Le principe de la réimplantation dynamique au moyen de registres de base est simple. Le contenu d'un registre particulier, ou **registre de base,** est automatiquement ajouté à toute adresse engendrée par un processus et le résultat est utilisé pour adresser la mémoire réelle. Un programme peut alors être chargé dans n'importe quel ensemble de cellules de mémoire consécutives ; il s'exécute correctement si le registre de base contient l'adresse d'implantation (dans le cas d'un programme implanté à partir de l'adresse virtuelle zéro). Les phases classiques d'édition de liens ou de chargement subsistent, mais elles produisent comme résultat un programme chargé dans un espace virtuel.

Déplacer globalement en mémoire centrale un programme dont l'exécution est commencée est alors possible, à la seule condition d'ajuster ensuite la valeur contenue dans le registre de base. L'utilisation de registres de base distincts pour le programme et pour les données permet leur réimplantation indépendante.

Remarque. Plusieurs machines utilisent l'adressage par registres de base, en particulier le CDC 6400, l'UNIVAC 1108 et le GE 635. L'IBM/360 n'appartient pas à la classe des machines pour lesquelles la réimplantation dynamique est possible sans conventions particulières. En effet, les registres de base sont accessibles par le programme qui peut les modifier ou ranger leur contenu pour les recharger plus tard ; cela revient à garder trace d'adresses absolues qui ne seraient plus valides après un déplacement du programme. De plus, le seul fait que les registres puissent contenir au gré du programmeur autre chose que des adresses rend impossible tout ajustement de ces registres après un déplacement du programme. Dans ces conditions, une réimplantation dynamique n'est possible que si les programmes respectent des conventions très strictes pour l'utilisation des registres. Le système CALL/360, par exemple, pratique la réimplantation dynamique des programmes ; le respect des conventions d'utilisation des registres est garanti par les compilateurs du système.

4.442 Algorithmes de gestion de la mémoire par zones

Nous considérons ici une mémoire allouée à la demande par zones de tailles quelconques : on doit alors résoudre les problèmes suivants [Knuth, 68 ; Randell, 69] :

— choix d'une représentation des zones,
— définition de critères de sélection d'une zone libre,
— politique de libération d'une zone occupée,
— décision à prendre quand aucune zone libre ne convient.

1) *Représentation des zones*

Une zone est définie par sa taille et son adresse de début, contenues dans un descripteur. Si le découpage de la mémoire est établi une fois pour toutes (nombre fixe de zones de tailles fixes), les descripteurs peuvent être rassemblés dans une table. Si par contre les tailles demandées sont variables, ils peuvent être placés dans les zones elles-mêmes et chaînés entre eux (liste simple ou circulaire). Les informations de gestion éventuellement contenues dans les zones allouées doivent être protégées.

Lorsqu'on alloue des zones dans une mémoire virtuelle ayant pour support une mémoire physique paginée, il est plus efficace de rassembler les descripteurs dans une table unique : on évite ainsi la dispersion des accès sur plusieurs pages.

L'ordre du chaînage a une influence sur l'efficacité des algorithmes. Le chaînage peut être construit dans l'ordre chronologique des libérations, mais le plus souvent on utilise l'un des deux classements suivants :

— classement par adresses croissantes ou décroissantes,
— classement par tailles croissantes ou décroissantes.

Le choix dépend de l'algorithme utilisé pour satisfaire une demande.

2) *Algorithmes de sélection*

Lorsqu'une demande est émise, une zone libre doit être choisie. Si les tailles des zones demandées ont une distribution quelconque, la probabilité de trouver une zone libre satisfaisant exactement la demande est pratiquement nulle. Une demande sera donc satisfaite le plus souvent à partir d'une zone libre plus grande ; la différence, ou **résidu,** est rattachée à la liste si elle n'est pas trop petite. Deux possibilités s'offrent, entre autres, quant au choix de la zone libre pour satisfaire une demande de n mots :

— prendre la première zone possible (« first fit »), c'est-à-dire parcourir la liste des zones libres jusqu'à ce que l'on en trouve une de taille $t \geqslant n$;

— prendre la plus petite zone possible (« best fit »), c'est-à-dire choisir celle qui donne le plus petit résidu.

Le temps pris par l'algorithme d'allocation dépend de la technique d'allocation choisie. Il peut être décomposé en deux parties : examen d'un certain nombre d'entrées pour le choix d'une zone libre et placement du résidu dans la liste. Si l'on choisit la plus petite zone possible, le classement par tailles évite de parcourir toute la liste. Le résidu reste à sa place s'il n'y a pas de classement ou si le classement est fait par adresses ; il doit être déplacé dans le classement par tailles.

Le phénomène d'accumulation de petits résidus en tête de liste ralentit la recherche. On utilise alors une liste circulaire qui permet de commencer l'exploration à partir de n'importe quel point de la liste.

La figure 13 montre un exemple d'allocation ; on remarque que la partie allouée se trouve à la fin de la zone choisie, pour faciliter la gestion des résidus.

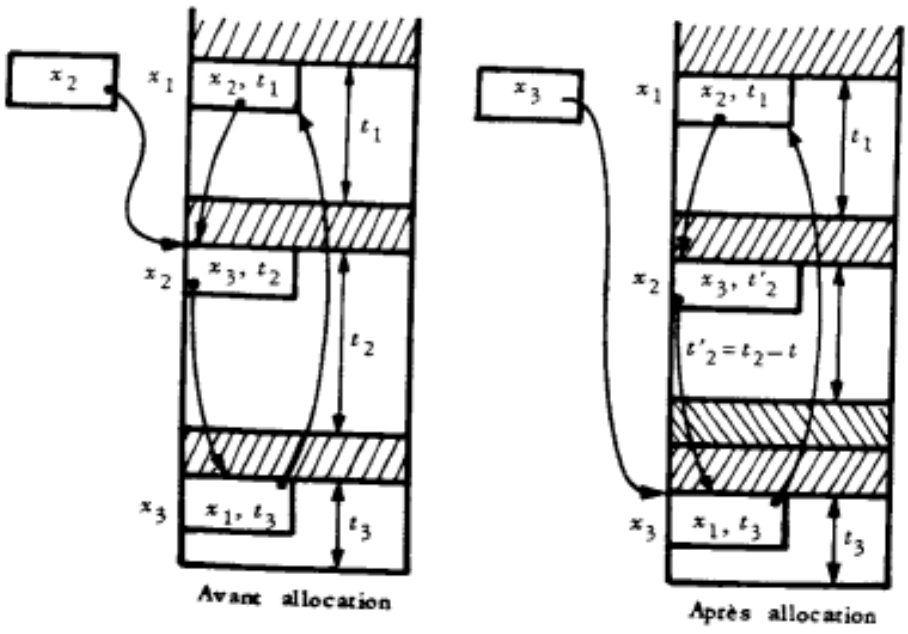


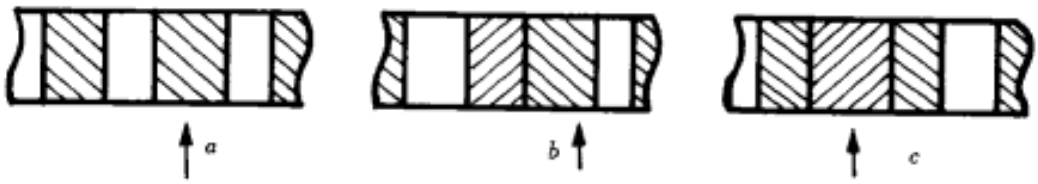
Figure 13. Allocation d'une zone de taille t (premier possible, liste circulaire).

On constate parfois que certaines tailles sont demandées plus fréquemment que les autres [Margolin, 71]. On améliore alors l'efficacité de l'allocation en réservant un certain nombre de zones possédant ces tailles privilégiées. Le mécanisme classique d'allocation reste utilisé en cas d'épuisement des zones réservées. Le nombre de zones à réserver pour chaque taille peut être déterminé par des mesures ou par une simulation du système.

Cette méthode de réservation, qui privilégie certaines tailles de zones, est couramment utilisée pour gérer la mémoire libre du système lui-même, quand ses besoins en mémoire sont connus à l'avance ; c'est le cas des blocs de mémoire acquis dynamiquement par le système pour gérer les entrées-sorties des utilisateurs.

3) *Libération d'une zone*

La libération d'une zone peut créer trois situations différentes selon que la zone libérée est entourée de deux zones libres (*a*), d'une zone allouée et d'une zone libre (*b*), ou de deux zones allouées (*c*).



Chaque fois que cela est possible (cas *a* et *b*), il est utile de regrouper, dès la libération, la zone libérée avec les zones libres voisines pour créer une seule zone libre de taille plus grande : on évite ainsi une fragmentation excessive de la mémoire. La rapidité de ce regroupement dépend encore du classement de la liste ainsi que des informations de gestion utilisées. Un classement qui est efficace pour l'allocation ne l'est pas forcément pour la libération.

4) *Fragmentation et compactage*

Dans une allocation du type « zone à la demande », le phénomène le plus gênant est celui de la fragmentation de la mémoire, qui se manifeste au bout d'un certain temps de fonctionnement ; il peut alors arriver que l'allocateur ne puisse trouver une zone de taille suffisante. Une solution consiste à compacter les zones allouées en les déplaçant toutes vers une extrémité de la mémoire, ce qui fait apparaître à l'autre bout une zone libre unique dont la taille est la somme des tailles des zones libres primitives.

Cette méthode, qui est coûteuse, n'est pas toujours applicable, car l'absence d'une possibilité de réimplantation dynamique interdit dans certains cas de déplacer l'information en mémoire. Par ailleurs, des résultats de simulation [Knuth, 68] ont montré que lorsque l'algorithme d'allocation ne peut satisfaire une demande, le taux de remplissage de la mémoire est souvent tel qu'après compactage on retombe très vite dans la même situation qu'avant ; le système consacre alors une grande partie de son temps au compactage.

Le compactage peut cependant être utilisé avec succès lorsque le nombre de zones allouées est faible et que la durée des allocations est importante ; cette situation correspond, en particulier, à l'allocation d'espace aux programmes dans un système de traitement par trains de travaux.

Exemple. Le système EXEC 8 sur UNIVAC 1108 compacte la mémoire chaque fois qu'un travail se termine, pour ne conserver qu'une seule zone libre située dans la partie haute de cette mémoire.

Le compactage peut être réalisé de trois manières :

— en utilisant des instructions de transfert de mémoire à mémoire ; cela ne permet pas à l'unité centrale de faire autre chose pendant le compactage,

— en utilisant les canaux d'entrée-sortie pour recopier sur une unité périphérique les zones à déplacer, et pour relire les parties recopiées dans une autre zone de mémoire,

— en couplant deux canaux d'entrée-sortie pour effectuer les transferts de mémoire à mémoire sans utiliser l'unité centrale.

4.45 GESTION DE LA MÉMOIRE PAR PAGES

La traduction dynamique d'adresses réalisée par l'emploi d'un registre de base impose que les programmes soient chargés en mémoire principale dans des cellules contiguës. Nous avons vu que les techniques d'allocation associées conduisent à la fragmentation de la mémoire (4.431). Pour l'éviter, il faut pouvoir implanter un programme dans plusieurs zones non contiguës. L'utilisation de registres de base multiples et explicites n'est guère envisageable, car cela supposerait que le nombre et la taille des zones, couvertes chacune par un registre de base, soient déterminés au moment de l'écriture du programme.

4.451 Mécanismes de pagination

La solution apportée par les mécanismes de pagination consiste à découper l'espace adressable, ou espace virtuel, en zones de taille fixe appelées **pages** et à associer implicitement l'équivalent d'un registre de base à chacune de ces pages. La mémoire réelle est également découpée en **cases** ayant la taille d'une page de sorte que chaque page peut être implantée dans n'importe quelle case de la mémoire réelle.

1) *Pagination à un niveau*

Une adresse est divisée en deux parties : un numéro de page p et un numéro de mot d, ou déplacement, à l'intérieur de la page. Les P bits de gauche d'une adresse de N bits fournissent le numéro de page, et les $N - P$ bits de droite le déplacement dans la page. La taille de la page est égale à 2^{N-P}.

La traduction des adresses utilise une fonction de topographie réalisée au moyen d'une **table de pages** qui est située en mémoire centrale ou dans des registres, et dans laquelle les entrées successives correspondent aux pages

virtuelles consécutives. La p-ième entrée de la table des pages contient le numéro r de la case où est implantée la page p, et éventuellement des indications supplémentaires. L'adresse réelle (r, d) d'un mot d'adresse virtuelle (p, d) est obtenue en remplaçant le numéro de page p par le numéro de case r trouvé dans la p-ième entrée.

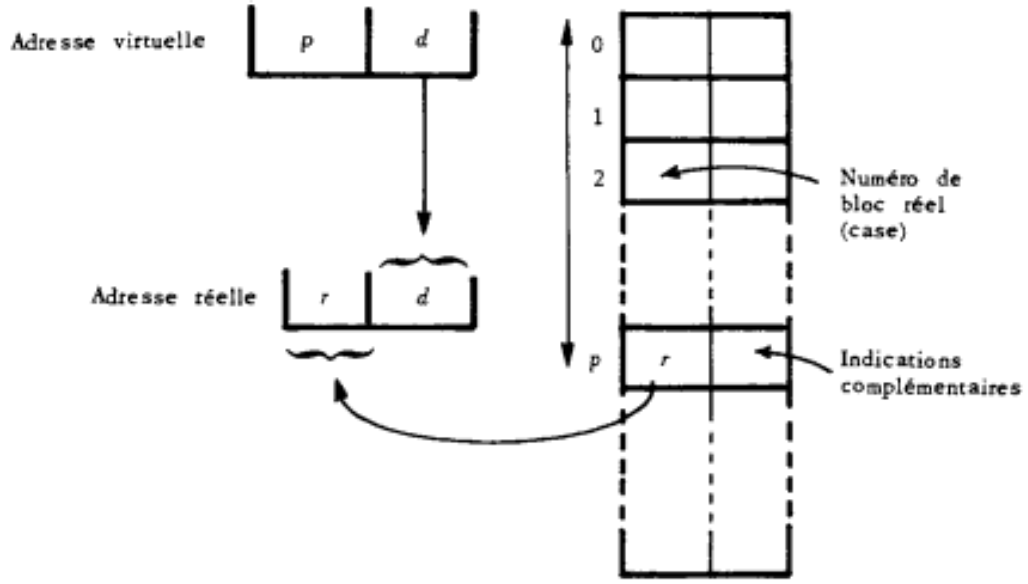


Figure 14. Calcul d'adresse avec pagination à un niveau.

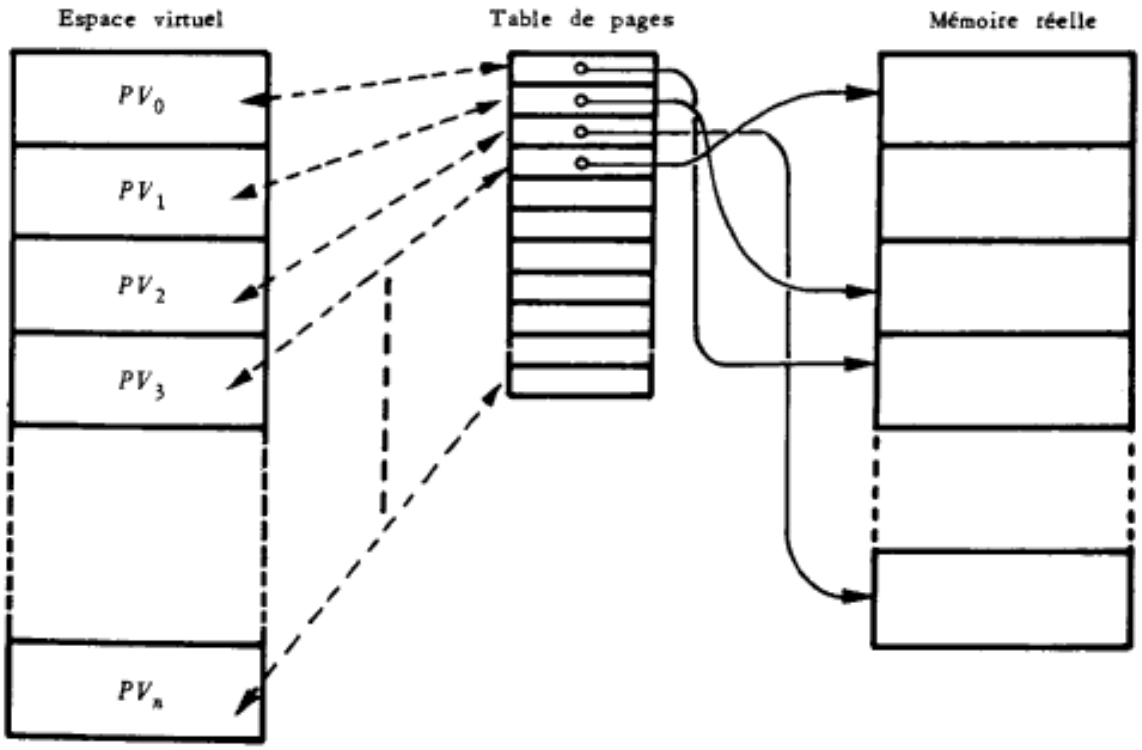


Figure 15. Correspondance entre espace virtuel et espace réel.

Lorsque le mécanisme de traduction des adresses utilise des tables de pages se trouvant en mémoire centrale, toute traduction entraîne une référence à la mémoire et ralentit donc notablement le processeur. Pour éviter cela, on peut implanter des mécanismes accélérateurs qui retiennent dans des registres à accès rapide (par exemple des registres associatifs) les numéros des pages les plus récemment référencées et les numéros de cases correspondantes. La traduction d'une adresse consiste alors à consulter d'abord ces registres et à n'effectuer l'accès aux tables que si le numéro de page virtuelle ne s'y trouve pas. Lorsqu'une adresse est obtenue par consultation des tables, le numéro de page virtuelle p et le numéro de case réelle r sont enregistrés dans l'un des registres, choisi selon un algorithme approprié. La propriété de localité (cf. 4.23) permet, avec un petit nombre de registres, de traduire la plupart des adresses sans avoir à consulter les tables en mémoire centrale.

Remarque. Il existe généralement un mode de fonctionnement du processeur, indiqué dans le mot d'état, qui ignore le mécanisme de traduction d'adresses. Les adresses émises par un programme sont alors utilisées directement pour accéder à la mémoire réelle.

Les tailles de pages choisies par les constructeurs sont en général toutes du même ordre de grandeur (de 512 à 4 096 caractères). Le choix de la taille des pages est guidé par les considérations suivantes.

a) L'allocation par pages fait apparaître un nouveau type de fragmentation, dit **fragmentation interne** [Randell, 69] : la dernière page d'un espace virtuel est, en moyenne, utilisée à moitié seulement, mais occupe une case complète. Cette perte de place est aggravée lorsque l'on charge dans un même espace virtuel diverses sections de programmes, implantées chacune, pour des raisons de protection ou d'efficacité, à partir d'un début de page. Plus les pages sont petites, plus la fragmentation interne est réduite.

b) La taille de la table des pages, pour un espace virtuel de taille donnée, est proportionnelle au nombre de pages de cet espace ; il est donc préférable, pour réduire cette taille, d'avoir peu de grandes pages plutôt que beaucoup de petites pages. Lorsque la table de pages est de petite taille, elle peut être constituée par des registres rapides ; sinon, il faut utiliser la mémoire principale comme support, et le problème du temps d'accès à cette table se pose alors.

c) Le temps de lecture d'une page à partir d'une mémoire secondaire (disque ou tambour magnétique) n'est pas proportionnel à la taille de cette page ; ce temps fait intervenir la durée de positionnement de l'unité de lecture qui est indépendante de la taille de la page, et la durée du transfert qui est proportionnelle à la taille mais qui ne constitue généralement qu'une faible partie du temps total (cf. 4.5). Pour réduire ce temps de lecture, il est donc préférable d'utiliser des grandes pages.

d) Pour une mémoire réelle donnée, la taille de page détermine le nombre de cases allouables ; cela conduit à utiliser des pages plus petites sur des machines dont la mémoire réelle est petite.

Certaines machines permettent l'utilisation de plusieurs tailles de pages (généralement 2). La taille utilisée est alors indiquée dans un registre spécifique ou dans les entrées de la table de pages.

Exemple. L'IBM/370 utilise, soit des pages de 2 048 octets, soit des pages de 4 096 octets. La taille choisie est indiquée dans le registre de contrôle 0, chargé par une instruction privilégiée.

Associées à chaque entrée de la table de pages, figurent fréquemment des informations de contrôle destinées à faciliter la mise en œuvre de la pagination. On peut citer :

— Un **bit d'invalidité** associé à chaque entrée. S'il est à 1, par exemple, un déroutement pour défaut de page se produit au moment où le processeur fait référence à cette entrée lors de la traduction d'une adresse virtuelle. Il est alors possible de n'amener les pages en mémoire réelle qu'au fur et à mesure de leur utilisation ; on réalise ce que l'on appelle le **chargement à la demande** (« demand paging ») qui garantit, en particulier, que l'on ne charge jamais des pages qui seront inutilisées. Le traitement d'un déroutement pour défaut de page consiste à obtenir une case libre en mémoire principale, à charger la page virtuelle manquante dans cette case, à garnir l'entrée de la table de pages avec le numéro de cette case et à remettre à 0 le bit d'invalidité. L'exécution du programme interrompu peut ensuite être relancée à partir de l'instruction qui avait provoqué le déroutement.

— Des **bits d'utilisation,** tenus à jour automatiquement. Ils permettent de connaître l'usage qui est fait d'une page (écriture, référence).

— Des **verrous d'accès** associés à chacune des pages virtuelles. Ils indiquent généralement si la page peut être lue, modifiée, exécutée. L'absence de bit d'invalidité peut être compensée par l'emploi d'un verrou interdisant tout accès à la page.

Exemple : mémoire topographique du CII 10070.

Dans la machine CII 10070 (3.41), une adresse virtuelle occupe 17 bits, ce qui permet d'adresser 128 K mots. La taille maximale de la mémoire réelle est également de 128 K mots. L'espace virtuel est découpé en 256 pages de 512 mots. Les 8 bits de gauche de l'adresse virtuelle fournissent le numéro de page et les 9 bits de droite le déplacement dans la page. La traduction dynamique des adresses est réalisée par une « mémoire topographique » constituée de 256 registres (de 8 bits) pouvant contenir chacun un numéro de case. Un verrou d'accès virtuel de 2 bits est associé à chaque entrée de la mémoire topographique ; sa signification est la suivante :

00 : tous accès autorisés,
01 : écriture interdite,
10 : lecture autorisée,
11 : aucun accès permis.

On peut utiliser la valeur *11* de ce verrou d'accès comme indicateur d'invalidité ; on interprète alors le déroutement pour violation de protection d'accès comme un défaut de page. La connaissance de la composition de l'espace virtuel d'un processus permet de distinguer un déroutement pour page manquante d'un déroutement pour accès hors des limites de cet espace virtuel.

2) *Pagination à deux niveaux*

Dans l'exemple précédent, la taille de l'espace virtuel et la taille maximale de la mémoire réelle sont identiques (128 K mots). Cependant, la taille de l'espace virtuel utilisable reste la même si l'on dispose d'une mémoire réelle plus petite (32 K mots par exemple). Les mécanismes de pagination fournissent donc un moyen d'adresser un espace virtuel plus grand que l'espace réel et limité seulement par le nombre de bits de l'adresse.

Si l'espace virtuel est grand, la table de pages associée est grande et peut même remplir complètement la mémoire réelle.

Exemple. Une machine à adressage par mot dont l'adresse occupe 24 bits permet d'adresser un espace virtuel de 16 384 K mots. Avec une taille de page de 512 mots, et en supposant qu'une entrée de la table de pages occupe un mot, la table décrivant un seul espace virtuel occupe 32 K mots en mémoire réelle.

La pagination à deux niveaux apporte une solution à ce problème de la taille de la table de pages, en permettant :

— de ne décrire par des tables de pages que la partie utile d'un espace virtuel (les zones inutilisées à l'intérieur de l'espace ne sont pas décrites),

— de partager entre plusieurs processus des descriptions de parties d'espace virtuel.

L'espace virtuel est découpé en un certain nombre de grands blocs de taille fixe, chacun contenant un nombre entier de pages. On a l'habitude d'appeler ces grands blocs des segments, bien que leur définition diffère fondamentalement de celle utilisée au chapitre 3. Pour ne pas créer de confusion, nous appellerons ces grands blocs des **hyperpages** dans la suite de ce texte.

Une adresse virtuelle est découpée en un numéro d'hyperpage h, un numéro de page p et un déplacement dans la page d. Les H bits de gauche d'une adresse de N bits constituent le numéro d'hyperpage, les P bits du milieu le numéro de page et les N-H-P bits de droite, le déplacement. Il y a 2^H hyperpages contenant chacune 2^P pages. La fonction de topographie est définie par une table d'hyperpages dont chaque entrée permet d'accéder à une table de pages qui ne décrit que la partie de l'espace virtuel couvert par cette hyperpage. Une entrée de la table des hyperpages peut également contenir :

— un bit d'invalidité qui indique qu'il n'y a pas de table de pages associée à cette hyperpage ;

— des droits d'accès à l'hyperpage ; le droit d'accès effectif à une page est fonction du droit d'accès à l'hyperpage qui la contient et d'un droit particulier attaché à chaque page.

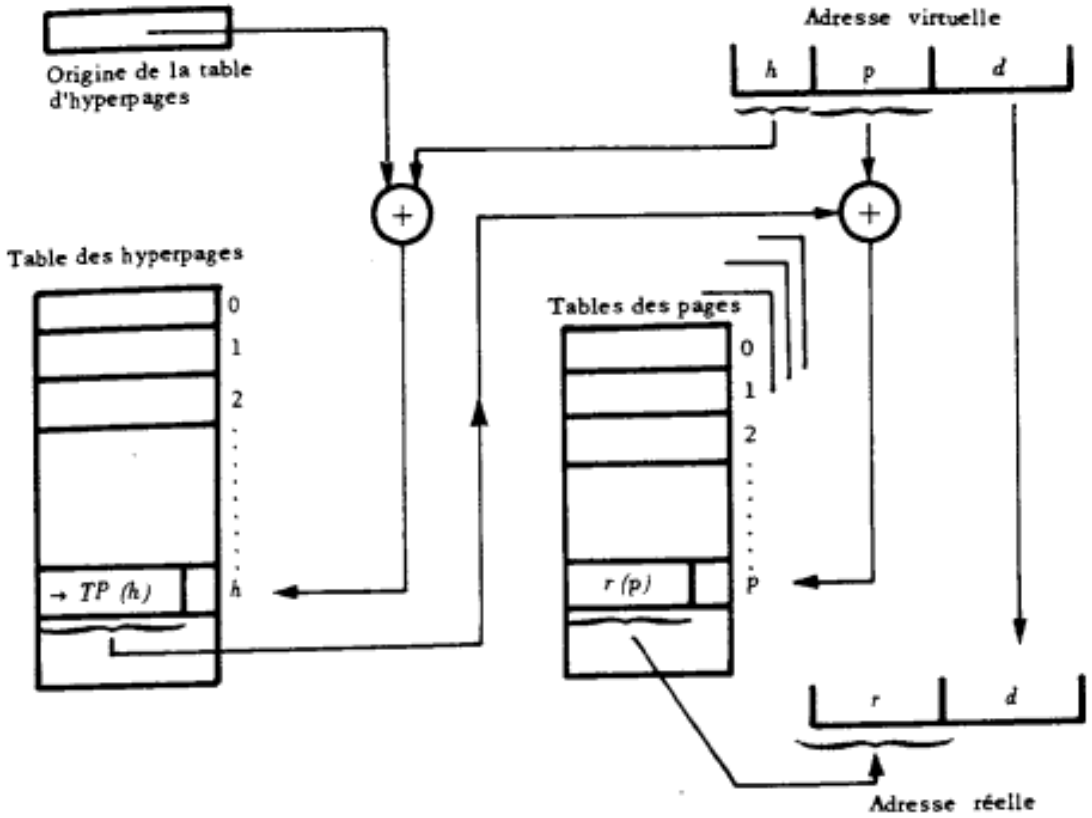


Figure 16. Calcul d'adresse avec une pagination à deux niveaux.

Exemple : pagination à deux niveaux de l'IBM/370.

Dans l'IBM/370 [IBM 72], une adresse virtuelle occupe 24 bits, ce qui permet d'adresser 16 384 K octets. L'espace virtuel est découpé en 16 hyperpages de 1 024 K octets ou en 256 hyperpages de 64 K octets. La taille des pages peut elle-même être choisie égale à 2 K octets ou à 4 K octets, ce qui donne quatre possibilités de structuration d'une adresse virtuelle. Le tableau suivant donne le nombre de bits utilisés par le numéro d'hyperpage H, le numéro de page P et le déplacement D.

H	P	D	
8	4	12	256 hyperpages contenant chacune 16 pages de 4 K
8	5	11	256 — — 32 — 2 K
4	8	12	16 — — 256 — 4 K
4	9	11	16 — — 512 — 2 K

La table des hyperpages et les tables de pages sont situées en mémoire principale. Des registres de contrôle, accessibles seulement en mode maître, contiennent la taille d'hyperpage, la taille de page et l'adresse de la table d'hyperpages définissant l'espace virtuel auquel le processus courant a accès.

Pour éviter une double référence à la mémoire lors de chaque traduction d'adresse, des mécanismes accélérateurs implantés différemment selon les modèles permettent de retenir dans des registres à accès rapide les couples (numéro d'hyperpage, numéro de page) récemment utilisés et le numéro de case réelle correspondante.

Remarque. En général, et c'est le cas pour l'exemple précédent, un découpage de l'espace virtuel en hyperpages et pages est tel qu'il y a continuité d'adresse entre le dernier mot de l'hyperpage n et le premier mot de l'hyperpage $n + 1$. En particulier, une séquence d'instructions à cheval sur deux hyperpages consécutives s'exécute sans discontinuité lors du passage d'une hyperpage à l'autre.

Il existe cependant des réalisations dans lesquelles les hyperpages sont disjointes : si l'on ajoute 1 par modification d'adresse à l'adresse du dernier mot d'une hyperpage, on retrouve l'adresse du premier mot de cette même hyperpage. Dans ces conditions, l'espace virtuel directement adressable par un processus a la taille d'une hyperpage et non la taille définie par le nombre de bits de l'adresse ; les liaisons entre hyperpages peuvent toutefois être assurées par le système.

Exemple. La machine RCA SPECTRA 70/46 utilise une adresse de 21 bits ce qui permet d'adresser 2 048 K octets. Cet espace virtuel est découpé en 8 hyperpages disjointes de 256 K octets chacune.

4.452 Représentation des espaces virtuels dans le système

La table de pages est utilisée directement par le mécanisme câblé de traduction d'adresses. Lorsqu'un défaut de page se produit, le système doit amener la page correspondante depuis la mémoire secondaire dans une case de la mémoire centrale. Il doit donc disposer d'une table, que nous appelons table auxiliaire, et qui contient autant d'entrées qu'il y a de pages dans l'espace virtuel considéré ; chaque entrée donne l'adresse de la page en mémoire secondaire (cf. 4.53). La table auxiliaire est également utilisée lorsque l'on doit recopier une case contenant une page modifiée. A un instant donné, il y a autant d'ensembles (table de pages, table auxiliaire) que d'espaces virtuels définis ; un seul de ces espaces est accessible au processeur.

Le système tient à jour par ailleurs une table d'allocation des cases. Lorsqu'une case est allouée à une page d'un espace virtuel, on indique dans l'entrée correspondante de la table d'allocation l'adresse de l'entrée de la table auxiliaire qui correspond à cette page. Lorsque l'on décide de recouvrir une case, on connaît ainsi directement l'adresse en mémoire secondaire où elle doit éventuellement être recopiée, ainsi que l'entrée de la table de pages à invalider. Le problème est un peu plus complexe lorsque des pages sont partagées par plusieurs espaces virtuels ; on peut dans ce cas par exemple chaîner entre elles les entrées des diverses tables auxiliaires correspondant à une page partagée, de manière à retrouver toutes les entrées de tables de pages à invalider lors d'un recouvrement.

4.453 Stratégie d'allocation des cases

Pour étudier la stratégie d'allocation des cases, nous utilisons le modèle suivant.

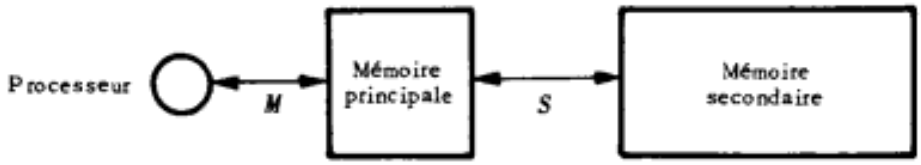
1) Le système est constitué d'un processeur et d'une mémoire à deux niveaux. La mémoire principale comprend m cases (emplacements de pages) allouables ; la mémoire secondaire est également divisée en cases et nous la supposons assez grande pour contenir toute l'information gérée par le système.

2) Le système est utilisé pour l'exécution d'un nombre fixe de processus ; l'exécution d'un processus p_i se traduit par une suite ω de références à un espace virtuel de n_i pages ($\Sigma n_i > m$)

$$\omega = \{r_1\, r_2\, r_3 \ldots r_t \ldots\}$$

$r_t = i$ signifie que la page i fait l'objet de la t-ième référence.

Soit M le temps d'accès à un mot d'une page chargée en mémoire principale et S le temps de transfert d'une page entre mémoire secondaire et mémoire principale.



Une caractéristique importante du système est le rapport $\Delta = S/M$. Dans l'état actuel de la technologie (mémoire principale à tores, mémoire secondaire sur tambour ou disque à têtes fixes), ce rapport est de l'ordre de 10^4. Toutefois, l'évolution de la technologie peut le faire descendre à des valeurs très inférieures, ce qui risque de modifier fondamentalement les idées actuelles sur la pagination.

L'objectif que l'on se fixe ici consiste à réduire le temps total d'exécution des processus présents. Il en découle quelques considérations intuitives.

1) Si, à un instant donné, un seul processus occupe la mémoire, tout chargement de page entraîne une période d'oisiveté de l'unité centrale de durée S. Pour avoir une bonne utilisation de l'unité centrale, on est conduit à admettre des pages de plusieurs processus en mémoire principale (le temps de commutation d'un processus à un autre est très bref, de l'ordre de M).

2) Plus le nombre de processus entre lesquels on partage la mémoire principale est grand, moins ces processus ont de cases à leur disposition. Il en résulte une plus grande quantité de défauts de pages ce qui crée une forte charge pour l'unité d'échange entre mémoires principale et secondaire.

Dans ce paragraphe, nous étudierons quand et où amener une page en mémoire principale. Deux stratégies de chargement de page sont envisageables :

— avant utilisation **(pré-chargement)** : on risque de charger des pages inutilement mais cette méthode permet d'utiliser l'unité d'échange au moment où elle est peu chargée et de transférer plusieurs pages à la fois ;

— au moment de la première référence **(chargement à la demande).**

Nous nous limitons dans ce qui suit à cette dernière stratégie qui est la plus fréquemment utilisée.

Quand le système a décidé de charger une page en mémoire principale, il doit déterminer dans quelle case la placer ; s'il n'existe pas de case libre il faut déterminer quelle page recouvrir en mémoire principale. L'algorithme qui détermine quelle page recouvrir s'appelle **algorithme de remplacement.** Il peut tenir compte des informations sur l'utilisation passée des pages présentes en mémoire principale.

4.454 Algorithmes de remplacement

De nombreux travaux ont été consacrés aux algorithmes de remplacement ; en particulier des études théoriques [Coffman, 73] ont été faites pour le cas où on se limite à un seul processus. Les algorithmes se différencient par les informations prises en compte et relatives à l'utilisation passée des pages (date de chargement, date de la dernière référence, ...). Citons :

1) **FIFO**(« First In First Out ») où l'on remplace la page la plus anciennement chargée,

2) **RAND** (« Random choice »), où l'on choisit au hasard la page à remplacer,

3) **LRU** (« Least Recently Used »), où l'on remplace la page la moins récemment utilisée,

4) **LFU** (« Least Frequently Used »), où l'on remplace la page la moins fréquemment utilisée.

L'algorithme optimal **OPT** minimise le nombre de chargements mais il suppose connue la chaîne des références ω du processus. Cet algorithme est le suivant :

— s'il existe des pages qui ne seront plus référencées, alors remplacer l'une de ces pages,

— sinon, remplacer la page qui sera référencée le plus tardivement.

Pour comparer les algorithmes de remplacement, définissons le taux de défauts de page par :

$$F = \frac{c(m, \omega)}{|\omega|}$$

où $c(m, \omega)$ désigne le nombre de remplacements de pages provoqués par le traitement, dans une mémoire de m cases, de la chaîne de références ω, et $|\omega|$ désigne le nombre d'éléments de ω.

En faisant la synthèse d'une série de mesures, on aboutit à des courbes dont l'allure générale est représentée sur la figure 17 [Denning, 70].

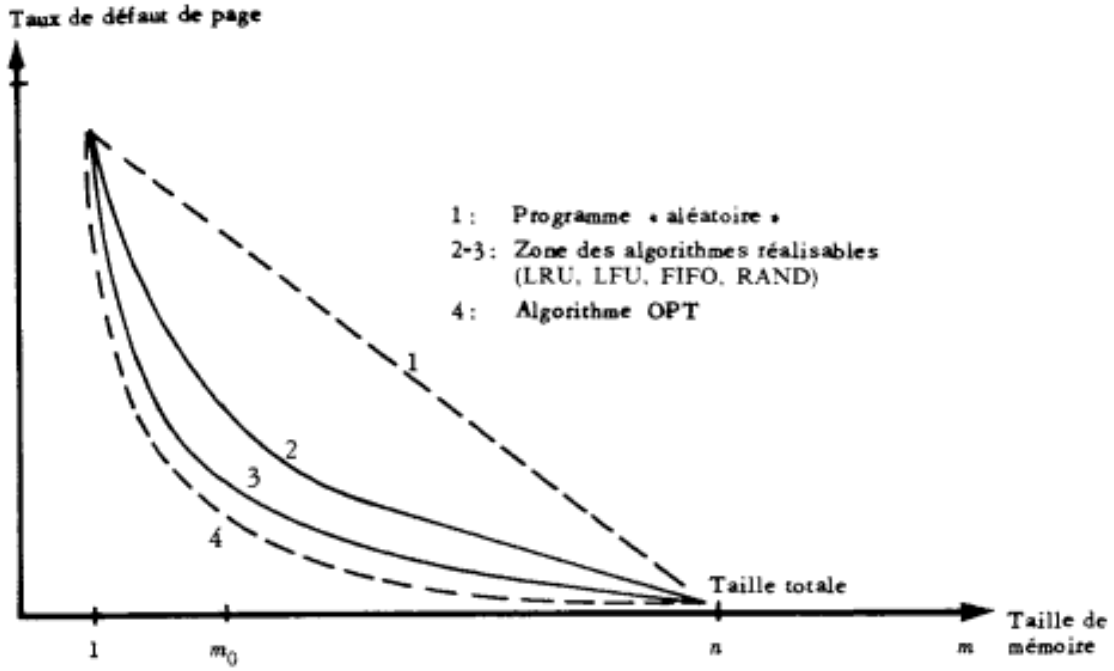


Figure 17. Influence de la taille de mémoire et de l'algorithme de remplacement sur le taux de défauts de page.

On constate qu'il existe une taille de mémoire m_0 en deçà de laquelle même OPT conduit à un nombre très élevé de défauts de page par unité de temps ; m_0 peut être notablement inférieur à la taille totale du programme. La droite « programme aléatoire » correspondrait à un programme où chaque page serait référencée avec la probabilité $1/n$, ce qui donne $F = (n - m)/n$ pour tout ω et pour tout algorithme de remplacement.

On pourrait s'attendre à ce que F soit une fonction non croissante de m pour un algorithme et une chaîne de références donnés. En fait, cette propriété n'est pas vraie dans le cas général : pour certains algorithmes (parmi lesquels FIFO), il existe des chaînes de références telles que

$$F(m_1, \omega) > F(m_2, \omega) \qquad \text{avec} \qquad m_1 > m_2$$

On appelle **anomalie de Belady** ce comportement contraire à l'intuition (exercice 7). Une condition suffisante pour qu'un algorithme ne présente pas cette anomalie est qu'il vérifie la propriété d'inclusion suivante :

$$S(m, \omega) \subseteq S(m + 1, \omega) \qquad \forall m, \omega$$

où $S(m, \omega)$ désigne l'ensemble des pages présentes en mémoire principale de taille m après traitement de la chaîne ω par l'algorithme considéré.

Si la même chaîne ω est traitée par cet algorithme dans deux mémoires de tailles différentes, la propriété d'inclusion exprime qu'à chaque étape de ce traitement (c'est-à-dire après une référence r_i donnée) toute page présente dans la mémoire la plus petite est également présente dans la mémoire la plus grande. On appelle **algorithme à pile** (« stack algorithm ») un algorithme vérifiant la propriété d'inclusion [Mattson, 70; Coffman, 71b]. L'intérêt de cette classe d'algorithmes est qu'ils permettent de prévoir le comportement d'un processus (nombre de défauts de page, identité des pages en mémoire à tout instant) dans une mémoire de taille m' si l'on connaît le comportement de ce même processus dans une mémoire de taille $m < m'$. Il est difficile de montrer dans le cas général qu'un algorithme est un algorithme à pile. Parmi les algorithmes usuels, LRU et OPT sont des algorithmes à pile, RAND et FIFO n'en sont pas (exercice 8).

4.455 Conclusions

Les mesures faites jusqu'à présent montrent (cf. 4.454) que les performances, en termes de mouvements de pages, sont plus influencées par la taille de m que par l'algorithme utilisé. De plus, il n'a pas été tenu compte dans cette étude du fait que seules les pages expulsées et modifiées devaient effectivement être recopiées sur mémoire secondaire. Il vaut mieux choisir un algorithme de réalisation simple, plutôt qu'un algorithme optimal complexe et coûteux dans sa mise en œuvre.

4.5 GESTION DE LA MÉMOIRE SECONDAIRE

4.51 INTRODUCTION

Dans ce paragraphe, nous nous intéressons aux mémoires secondaires rotatives que sont les tambours et les disques magnétiques, et nous considérons successivement deux emplois de ces mémoires.

1) Utilisation en tant que composant de la hiérarchie de mémoires et support d'une mémoire virtuelle (cf. 4.4). Lorsque la mémoire secondaire est utilisée à cette fin, on l'appelle généralement la **mémoire auxiliaire.** Nous nous limitons ici au cas d'une mémoire principale paginée. L'unité d'information échangée entre mémoire principale et mémoire auxiliaire est la page et l'unité d'allocation est la page ou un nombre entier de pages.

2) Utilisation en tant que support d'informations non directement adressables (fichiers par exemple). La mémoire secondaire prend alors souvent le nom de **mémoire externe.** L'unité d'information échangée entre mémoire principale et mémoire externe et l'unité d'allocation peuvent alors être variables.

Dans chacun de ces cas, nous étudierons les diverses techniques d'allocation d'espace et nous envisagerons l'optimisation des transferts avec la mémoire principale.

4.52 CARACTÉRISTIQUES PHYSIQUES DES UNITÉS

Nous considérons deux classes d'unités de mémoire secondaire, les tambours et les disques à têtes fixes d'une part, les disques à têtes mobiles d'autre part. Chaque unité peut être caractérisée par sa capacité et par le temps nécessaire pour amener en mémoire centrale une information résidant sur cette unité. La durée moyenne de lecture d'un bloc d'information fait intervenir le temps de positionnement des têtes de lecture-écriture (nul pour les tambours et les disques à têtes fixes), le temps d'attente du passage du début du bloc sous les têtes (en moyenne une demi-révolution) et le temps de transfert du bloc lui-même qui ne dépend que du débit de l'unité et de la taille du bloc. Pour les disques à têtes mobiles, on introduit la notion de **cylindre** : un cylindre est constitué par l'ensemble des pistes accessibles lorsque le bras portant les têtes de lecture-écriture (toutes solidaires) se trouve dans une position donnée. Cette notion peut être utilisée lors de l'allocation d'espace et lors de la réalisation des transferts pour minimiser les déplacements du bras.

Les deux tableaux qui suivent résument les caractéristiques d'unités représentatives de la technologie en 1973. Pour obtenir des valeurs comparables, nous avons ramené à la même unité (l'octet) les capacités et débits de ces unités.

1) Unités du type disque à têtes mobiles

	Capacité par unité (octets)	Temps moyen d'accès (ms)	Déplacement moyen des bras (ms)	Débit (octets/s)	Temps moyen pour lire 4 096 octets (ms)
IBM/3330 →	100×10^6	8,4	30	$0{,}8 \times 10^6$	43,4
UNIVAC → FASTRAND 2	99×10^6	35	58	$0{,}115 \times 10^6$	128

2) Unités du type disque à têtes fixes ou tambour

	Capacité par unité (octets)	Temps moyen d'accès (ms)	Débit (octets/s)	Temps moyen pour lire 4 096 octets (ms)
IBM/2305.2 →	11×10^6	5	$1{,}5 \times 10^6$	7,7
UNIVAC → FH-432	$1{,}18 \times 10^6$	4,3	$1{,}08 \times 10^6$	8,1

Il est également intéressant de comparer les coûts de ces deux types de mémoires, à quantité égale d'information emmagasinée et directement accessible (en excluant donc les informations se trouvant sur des disques non montés) : en prenant l'exemple des unités IBM 3330 et 2305.2, si le coût est égal à 1 pour la première il est de l'ordre de 50 pour la seconde ; par contre, les tableaux montrent que les temps de transfert pour un bloc de 4 096 octets sont seulement dans un rapport de 6 à 1. L'utilisation d'une unité à têtes fixes doit être justifiée par une augmentation du travail productif de l'unité centrale. Nous verrons en particulier (4.61) que le temps d'accès aux informations situées en mémoire auxiliaire est l'un des facteurs importants du phénomène d'écroulement. Une diminution de ce temps d'accès, en repoussant les limites d'apparition de ce phénomène, améliore l'utilisation de l'unité centrale.

Dans la suite de ce paragraphe, nous utiliserons le terme de **tambours** pour désigner à la fois les tambours et les disques à têtes fixes qui ont des caractéristiques identiques, et nous réserverons le terme de **disques** pour les disques à têtes mobiles. La capacité relativement faible des tambours conduit souvent à les associer à des disques dans la hiérarchie de mémoires. L'un des problèmes de l'allocation de mémoire secondaire consiste alors à répartir les informations entre ces deux niveaux.

4.53 GESTION DE LA MÉMOIRE AUXILIAIRE

4.531 **Allocation d'espace**

Les unités de mémoire auxiliaire sont divisées en cases, chaque case pouvant contenir une page virtuelle. En reprenant l'un des exemples précédents, il est possible, compte tenu des intervalles entre enregistrements, de loger 2 304 pages de 4 096 octets sur le tambour IBM 2305.2 et 23 400 pages sur un disque IBM 3330.

L'espace allouable est généralement représenté par une chaîne de bits, chaque bit correspondant à une case (cf. 4.12).

Nous avons vu (4.45) que tout espace virtuel est décrit dans le système par une table, dite table auxiliaire, permettant de situer les cases de mémoire auxiliaire associées aux pages de cet espace. L'allocation de mémoire auxiliaire à un espace virtuel, qui s'accompagne de la construction ou de la mise à jour de cette table, peut être réalisée à divers moments.

1) *Allocation statique lors de l'initialisation du système*

Si le nombre et la taille des espaces virtuels utilisés simultanément sont connus à l'avance, il est possible de définir lors de l'initialisation du système (et même éventuellement lors de sa génération), par quelle case de mémoire auxiliaire chacune des pages de ces espaces sera représentée. Cette méthode n'est à envisager que lorsqu'il n'existe qu'une seule unité de mémoire auxiliaire ; dans ce cas, en effet, le problème de la répartition dynamique entre les niveaux auxiliaires ne se pose pas et le problème de l'équilibrage dynamique

de la charge entre unités de même type non plus ; la solution de ces deux problèmes passe par une allocation dynamique.

Exemple. Le système DOS/VS sur IBM/370 utilise un seul espace virtuel de taille maximale égale à 16 384 K octets, découpé en régions (« partitions ») virtuelles dans lesquelles s'exécutent les travaux. La taille de page utilisée est égale à 2 048 octets. L'espace nécessaire sur mémoire auxiliaire (fichier de pagination), qui doit tenir sur une seule unité, est réservé, au plus tard, lors du chargement du système. Les cases sont allouées séquentiellement à partir d'un début de cylindre et l'adresse en mémoire auxiliaire associée à une page virtuelle se déduit du numéro de cette page (il n'y a pas de table auxiliaire). La taille du fichier de pagination est liée à la taille de l'espace virtuel définie lors de la génération du système.

2) *Allocation lors de la première utilisation d'une page*

La création par le système de l'espace virtuel d'un processus revient en fait à la création d'une table de pages qui est utilisée pour la traduction dynamique des adresses, et d'une table auxiliaire. Le contenu de l'espace virtuel est soit vide, soit initialisé avec certaines parties communes utilisables par tous les processus. Certaines entrées de la table auxiliaire peuvent ainsi déjà être garnies avec les adresses en mémoire auxiliaire des pages composant la partie de l'espace initialisée (que nous supposons allouée de façon permanente en mémoire auxiliaire). Lorsqu'une page, autre que l'une des pages initialisées, est référencée pour la première fois par le processus, un défaut de page se produit car l'entrée de la table de pages est invalidée. Comme l'entrée correspondante de la table auxiliaire indique que cette page n'a pas d'image secondaire, l'allocation peut être effectuée à ce moment-là, ou encore différée jusqu'au moment où il faudra recopier pour la première fois la case de mémoire principale que l'on alloue à cette page.

Exemple. Création d'un segment dans MULTICS.

Dans MULTICS, un segment comprend au maximum 64 pages de 1 024 mots. Lors de la création du segment, la table auxiliaire associée (« segment map ») est créée ; elle comporte l'identification de l'unité sur laquelle vont être allouées les cases pour ce segment (on impose aux diverses pages d'un segment de se trouver sur une même unité), et 64 entrées contenant comme adresse auxiliaire l'adresse d'une case fictive qui serait pleine de zéros. Lors du premier défaut de page pour une page virtuelle donnée, cette indication de case fictive conduit à remettre à zéro la case de mémoire principale que l'on alloue à cette page (aucun transfert n'est effectué), à allouer une case sur l'unité indiquée et à mettre à jour l'entrée correspondante. Sauf migration globale du segment (exemple ci-dessous), cette case secondaire est utilisée par la suite pour toutes les opérations de pagination concernant cette page virtuelle.

3) *Réallocation dynamique*

L'utilisation rationnelle de plusieurs niveaux de mémoire auxiliaire implique une distribution dynamique des pages dans les divers niveaux en fonction des fréquences d'utilisation de ces pages. Le support d'un espace virtuel peut être choisi globalement pour cet espace ou au niveau de chacune de ses pages.

a) Migration globale

Nous introduisons les notions d'espace virtuel temporaire et d'espaces virtuels permanents. Les espaces virtuels temporaires sont ceux dans lesquels s'exécutent les processus et qui disparaissent avec eux. A la disparition du processus, la mémoire auxiliaire associée à son espace est libérée (sauf cas particulier de l'allocation statique vue plus haut), et les emplacements correspondants sont réutilisables pour d'autres espaces virtuels. Le volume de mémoire auxiliaire nécessaire dans les systèmes qui ne gèrent que des espaces virtuels temporaires est relativement faible. Il suffit de pouvoir représenter la mémoire virtuelle des processus existant simultanément, et cela nécessite un nombre limité d'unités (le plus souvent une ou deux) connectées en permanence. Les espaces virtuels permanents, au contraire, sont créés par un processus et conservés après la disparition de ce dernier pour être réutilisés ultérieurement. Ce sont des segments, au sens du chapitre 3. La notion de mémoire externe disparaît, tout l'espace secondaire étant composé de segments décrits par des tables auxiliaires. Dans ces conditions, le volume de mémoire auxiliaire nécessaire pour stocker les informations permanentes peut devenir important et mettre en jeu de nombreux supports non simultanément connectables. Il importe alors, pour des raisons évidentes de performances, que toutes les pages d'un segment se trouvent sur un même support. Lors de la création d'un segment, un support initial est choisi ; ensuite, en fonction du taux d'utilisation de ce segment, on pourra décider de déplacer globalement ses pages vers un support plus rapide ou plus lent.

Exemple : migration d'un segment dans MULTICS.

La table auxiliaire décrite dans l'exemple précédent comporte une entrée supplémentaire contenant, lorsqu'une migration a été décidée, l'identification de la nouvelle unité, et zéro dans le cas contraire. A chacune des 64 entrées est associé un indicateur de page déplacée. Lorsqu'à la suite d'un défaut de page, une page doit être lue, elle l'est depuis l'ancienne ou la nouvelle résidence selon que l'indicateur est hors fonction ou en fonction. Lorsqu'une case de mémoire principale doit être recouverte et que la page qu'elle contient est une page non déplacée d'un segment en cours de migration, alors le recouvrement est précédé de :

— l'allocation d'une case sur la nouvelle unité,
— la libération de la case utilisée sur l'ancienne unité,
— la mise à jour de l'entrée correspondante de la table auxiliaire (y compris l'indicateur de page déplacée),
— l'écriture de la page depuis la mémoire principale sur cette nouvelle unité.

Nous voyons que cette méthode permet d'utiliser un segment en cours de migration. Lorsqu'on veut retirer du descriptif (par exemple à la fin d'un processus) un segment qui n'est pas en cours de migration, les cases de mémoire principale qu'il occupe sont libérées et les pages modifiées sont recopiées sur le support auxiliaire. Si le segment est en cours de migration, toutes ses pages résidant encore sur l'ancienne unité sont référencées pour les forcer à venir en mémoire principale. En fin de migration, il reste à mettre à jour, dans la table auxiliaire, l'identification de l'unité, l'entrée supplémentaire et les indicateurs de page déplacée.

Le mécanisme tel qu'il est décrit ici s'applique à un segment figurant dans le descriptif d'un processus. Pour déplacer un segment n'y figurant pas, il suffit de mettre l'identification de la nouvelle unité dans sa table auxiliaire, de mettre ce segment dans un descriptif puis de l'en retirer ; il se retrouvera automatiquement sur une nouvelle unité.

b) Réallocation lors de chaque recopie d'une page

Cette technique, utilisée pour des espaces virtuels temporaires, consiste, chaque fois qu'une page doit être recopiée en mémoire auxiliaire, à libérer l'ancien emplacement et à en allouer un nouveau. Cette réallocation dynamique a deux buts :

— assurer qu'à tout instant les pages les plus utilisées se trouvent sur les unités les plus rapides ou que la charge des unités de même type est équilibrée,

— optimiser les opérations d'écriture par un choix judicieux des emplacements alloués (cf. 4.532).

Les pages d'un même espace virtuel peuvent ainsi être réparties sur des unités différentes. Pour éviter que les unités rapides ne soient progressivement encombrées par des pages dont la fréquence d'utilisation est devenue faible (après une période d'activité plus intense qui a conduit à les écrire sur ces unités rapides), on peut, lorsque le taux d'occupation des unités rapides dépasse un seuil fixé, provoquer la migration sélective des pages peu utilisées.

Exemple : mémoire auxiliaire dans le système TSS/360 [IBM, 69].

Le système TSS (« Time Sharing System ») sur IBM/360-67 utilise comme premier niveau de mémoire auxiliaire 1 ou 2 tambours pouvant contenir chacun 900 pages et comme second niveau des disques, dont chaque unité peut recevoir 6 496 pages.

Lors de la création d'un processus on calcule le nombre de cases dont il pourra disposer sur tambour. Au cours d'une période d'activité de ce processus, les pages qui lui appartiennent et qui doivent être recopiées sont allouées si possible sur tambour sans tenir compte de la limite précédente. Cela permet de ne pas sous-utiliser les tambours si des cases y sont disponibles. Par contre, si le nombre des cases non allouées des tambours descend au-dessous d'une valeur déterminée, alors le processus (pris sur la liste des processus inactifs) dont l'excès de pages sur tambour par rapport à son allocation calculée est maximum voit certaines de ses pages déplacées vers les disques. Seules les pages référencées par le processus durant sa dernière tranche de temps restent sur tambour, jusqu'à concurrence du nombre maximal calculé. Si l'on ne trouve pas suffisamment de pages à déplacer appartenant à des processus inactifs, on cherche dans la liste des processus actifs. A la fin d'une tranche de temps allouée à un processus, toutes ses pages modifiées en mémoire principale sont écrites dans de nouveaux emplacements de mémoire auxiliaire et les anciens emplacements sont libérés.

4.532 Gestion des transferts pour un tambour

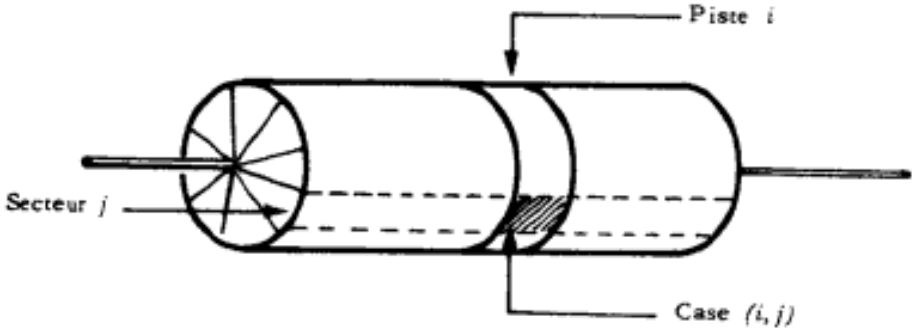
Chaque piste d'un tambour possède sa propre tête de lecture-écriture et est divisée en cases. Dans le cas où la place perdue sur chaque piste, du fait du découpage en un nombre entier de cases, est trop importante, on peut placer des cases à cheval sur deux pistes. On définit alors la **piste logique** comme étant formée du plus petit nombre de pistes consécutives contenant un nombre entier de cases.

Exemple. Le nombre de 2 304 pages que nous avons donné comme capacité de l'unité IBM 2305.2 (cf. 4.531) correspond au cas où l'on écrit huit pages sur trois pistes. L'unité comporte 864 pistes, soit 288 pistes logiques.

1) *Stratégies utilisées*

Dans ce qui suit, nous nous plaçons dans le cas où la piste logique et la piste physique sont identiques. Les techniques que nous développons s'étendent aisément au cas contraire.

Soit s le nombre de cases par piste. On dit que le tambour comporte s **secteurs.** L'adresse d'une case est complètement définie par son numéro de piste et son numéro de secteur.



Pour gérer la file d'attente des demandes de transfert, on utilise surtout les deux stratégies suivantes :

— la stratégie FIFO qui traite les demandes dans leur ordre d'arrivée,

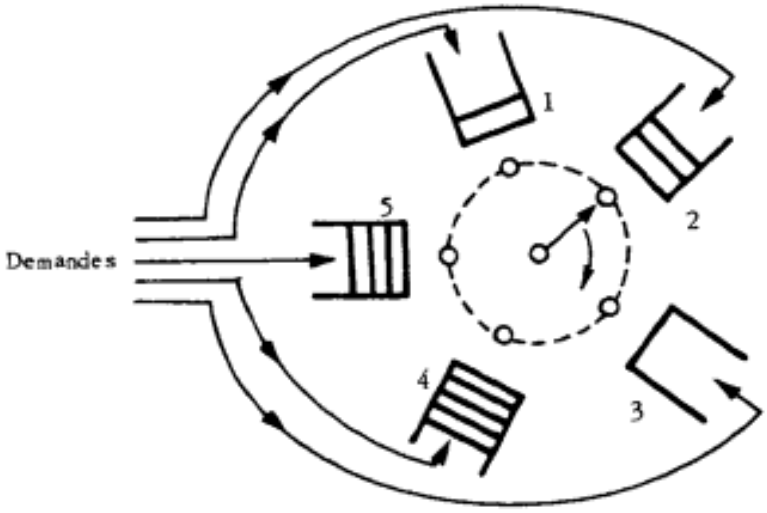
— la stratégie SATF (« Shortest Access Time First ») qui consiste à exécuter en premier lieu la demande qui nécessite le plus faible délai rotationnel [Abate, 69 ; Denning, 70].

Voyons comment cette dernière stratégie peut être mise en œuvre. On définit s files d'attente, une par secteur, rangées dans l'ordre de balayage des secteurs ; une demande de transfert de page est placée en queue de la file correspondant au secteur dans lequel se trouve la case concernée (« sector queueing »). Lorsqu'un transfert s'achève pour le secteur j, l'opération suivante est initialisée à partir de la tête de la première file non vide suivant la file j (modulo s). Chacune des files est ainsi gérée par une technique FIFO mais la gestion de l'ensemble des files s'apparente à une méthode de tourniquet (cf. 4.33).

Cette organisation suppose qu'il est possible d'initialiser un transfert pour le secteur $j + 1$ entre l'interruption signalant la fin d'un transfert pour le secteur j et le passage du début d'une case du secteur $j + 1$ sous les têtes de lecture. La taille des intervalles entre cases sur une piste doit donc avoir été choisie en conséquence. On peut également utiliser, lorsqu'il existe, le chaînage des commandes d'entrée-sortie. On prépare ainsi, en prélevant la demande se trouvant en tête de chaque file non vide, une chaîne de commandes correspondant à une révolution complète, et on lance cette chaîne de commandes.

Les intervalles entre cases doivent alors être seulement suffisants pour permettre la sélection électronique d'une tête de lecture-écriture. Un raffinement possible consiste à préparer une nouvelle chaîne pendant que s'exécute la précédente et à la raccrocher dynamiquement à cette dernière (technique utilisée par le système TSS/360). Remarquons cependant que l'utilisation du chaînage de commandes ne permet pas de manière simple de rajouter « au vol », dans une chaîne en cours d'exécution, une demande présentée pour un secteur non utilisé dans cette chaîne. Cet inconvénient est réduit en présence d'une forte charge, car la probabilité d'avoir des chaînes complètes est alors plus grande.

Un tambour géré suivant la stratégie SATF est parfois appelé **tambour de pagination** (« paging drum »).



L'efficacité d'une stratégie en présence d'une charge donnée peut se mesurer par le rapport entre le nombre de pages transférées par unité de temps et le nombre de cases passant sous les têtes de lecture-écriture pendant ce même temps. Soit L le nombre de demandes de transfert de page se trouvant en file d'attente. Nous supposons que L est constant (système en équilibre) et que les demandes de transfert sont équiprobables pour les s secteurs. On montre (exercice 4) que, lorsque les commandes ne sont pas chaînées, l'efficacité moyenne E_f de la stratégie FIFO, et E_t l'efficacité moyenne de la stratégie SATF, s'expriment, pour tout $L \geqslant 1$ par :

$$\begin{cases} E_f(L) = 2/(s+1) \\ E_t(L) \simeq (L+1)/(s+L+1) \qquad \text{(pour } s > 1\text{)} \end{cases}$$

Alors que l'efficacité de la stratégie FIFO est constante quelle que soit la charge, l'efficacité du tambour de pagination est d'autant plus grande que la charge est plus élevée. Pour tout $L > 1$ on a $E_t(L) > E_f(L)$.

Une caractéristique importante du tambour de pagination est que le temps moyen s'écoulant entre l'entrée d'une demande dans sa file d'attente et la

fin du transfert est diminué par rapport à FIFO, sans que le temps maximal soit supérieur à celui de FIFO. En effet, dans la stratégie SATF, les seules demandes qui peuvent retarder la prise en compte d'une demande particulière sont celles situées devant elle dans la même file.

2) *Incidence des algorithmes d'allocation d'espace*

Lorsque la demande correspond à une lecture, la case et donc le secteur sont, bien entendu, imposés. Pour une écriture, par contre, si le système pratique la réallocation dynamique à chaque recopie de page, alors la case allouée peut être choisie de manière à optimiser le transfert. Dans ce cas, l'allocateur fait partie du tambour de pagination. Le meilleur résultat est obtenu si l'on alloue la case dans le secteur qui a la plus courte file d'attente. Si plusieurs secteurs ont des files d'attente de longueur égale, on choisit celui qui passera le premier sous les têtes de lecture-écriture (on ne connaît pas toujours cette information).

Remarque 1. Dans ce qui précède, nous avons supposé que les demandes de transfert de page étaient toutes indépendantes et pouvaient donc être exécutées dans un ordre quelconque. Cela n'est vrai que si certaines précautions sont prises par l'algorithme de pagination. En particulier, lorsqu'une page modifiée doit être recouverte en mémoire principale, il ne faut pas ranger simultanément dans les files d'attente la demande d'écriture et la demande de lecture, car elles pourraient être exécutées dans l'ordre inverse.

Remarque 2. On peut chercher à privilégier, à l'intérieur d'un même secteur, les lectures par rapport aux écritures, en considérant qu'à une demande de lecture de page correspond généralement un processus bloqué qui mobilise des ressources. Le système MTS (« Michigan Terminal System ») sur IBM/360-67 gère les transferts de cette manière.

Remarque 3. On peut élever la charge instantanée du tambour, donc augmenter l'efficacité, en groupant les demandes de transfert.

Exemple. Dans le système TSS, toutes les pages modifiées par un processus sont recopiées à la fin de sa tranche de temps, plutôt que d'être recopiées une à une au fur et à mesure des besoins en cases de mémoire principale. Si une page ainsi recopiée n'a pas été recouverte lorsque le processus est à nouveau activé, elle est utilisable sans relecture.

4.533 Gestion des transferts pour un disque

Pour les disques, le temps de déplacement du bras portant les têtes de lecture-écriture est prépondérant par rapport au temps de lecture ou d'écriture et également par rapport au délai rotationnel. Les stratégies employées tiennent compte de cette caractéristique.

1) *Stratégie minimisant les déplacements du bras*

On peut penser à adopter la stratégie SATF pour les transferts de pages entre les disques et la mémoire principale. La notion de cylindre s'ajoutant à celle de secteur, cette stratégie revient alors à gérer une file d'attente par cylindre, et à l'intérieur d'un même cylindre une file par secteur. Lorsque le bras se trouve dans une position donnée, on exécute toutes les demandes se trouvant dans les files du cylindre correspondant, en gérant ces files à la manière du tambour de pagination. Lorsque ces files sont vides, on déplace le bras, dans un sens ou dans l'autre, vers le cylindre le plus proche comportant une file non vide.

Cette stratégie n'est en général pas utilisée pour deux raisons principales :

a) Le temps moyen d'exécution d'une demande est diminué, mais le temps maximum est augmenté. En effet, une demande concernant un cylindre éloigné du cylindre courant peut être retardée indéfiniment si des demandes concernant des cylindres plus proches arrivent avec une fréquence suffisamment grande. On voit que si un processus est en attente d'un transfert de page, cette stratégie peut par exemple aller à l'encontre de certaines règles de priorités établies entre les processus.

b) Au niveau de chaque cylindre, une technique du type « tambour de pagination » est souvent superflue, car, sauf cas particuliers, la probabilité pour qu'il y ait plus d'une demande par cylindre à un instant donné est faible, en raison du petit nombre de cases par cylindre.

2) *Stratégie de l'ascenseur*

Il est possible d'éviter que des demandes ne soient trop longtemps différées en utilisant la stratégie dite de « l'ascenseur ». Dans cette méthode, le bras se déplace dans un sens donné, par exemple de l'extérieur vers l'intérieur, s'arrête au-dessus de chaque cylindre pour lequel des demandes existent et les traite. Lorsque le dernier cylindre comportant une file non vide a été atteint et traité, on change le sens de déplacement et on recommence.

Il subsiste le risque qu'une suite de demandes très rapprochées pour le cylindre situé sous le bras empêche l'exécution des demandes relatives aux autres cylindres. Ce risque est peu probable en raison de la faible capacité d'un cylindre. Il peut, de toutes façons, être évité en ne considérant que les demandes qui se trouvent dans la file du cylindre au moment où ce dernier est atteint.

Ici encore, comme dans le cas des tambours, une liaison entre l'algorithme d'allocation d'espace secondaire et la gestion des transferts est souhaitable. En ce qui concerne les opérations d'écriture, une allocation dynamique des cases lors de chaque recopie de page permet de choisir la case allouée en fonction de la position courante du bras. Cela suggère de décrire l'espace allouable cylindre par cylindre et de rechercher une case dans le cylindre le plus proche possible du cylindre courant, en tenant éventuellement compte du sens de

déplacement lorsque la technique de l'ascenseur est employée. Il est plus difficile d'allouer les cases de manière à réduire le temps total de lecture puisque l'identité des pages lues dépend du comportement dynamique des processus. On peut cependant essayer de grouper dans des cylindres voisins les pages qui ont la plus grande probabilité d'être utilisées pendant un intervalle de temps donné.

4.54 GESTION DE LA MÉMOIRE EXTERNE

Nous considérons que la mémoire externe est utilisée pour contenir des **fichiers,** un fichier étant une collection d'objets que nous appelons **articles.**

Les articles sont lus, créés ou modifiés par les processus au moyen de commandes explicites de lecture ou d'écriture. Ces articles sont en général de taille variable. Ils peuvent être écrits dans des cases (de taille fixe) ou dans des zones (de tailles diverses).

L'emploi de zones permet *a priori* d'associer un article à un emplacement et donc d'accéder à cet article par un transfert unique dans lequel la seule information déplacée est celle composant l'article. En fait, on est souvent amené, pour des raisons d'économie de place ou de temps de transfert, à grouper plusieurs articles par zone ; on peut également être conduit à utiliser plusieurs zones pour un même article lorsque la taille de ce dernier est supérieure à la taille maximale de l'enregistrement que l'on peut créer sur l'unité de mémoire secondaire considérée.

L'emploi de cases, au contraire, présente, par rapport à la création de zones à la demande, des avantages importants.

1) Les algorithmes d'allocation et de libération sont simplifiés. Toutes les cases sont équivalentes, quel que soit le type d'unité de mémoire externe.

Dans le cas d'une mémoire principale paginée, si la taille de case est choisie égale à la taille de page, les transferts peuvent s'effectuer par des mécanismes de couplage (cf. 3.4) : pour réaliser un transfert d'une case de mémoire externe dans une page de l'espace virtuel, il suffit d'indiquer, dans la table auxiliaire décrivant cet espace, l'adresse de la case en mémoire externe et d'invalider l'entrée de la table de pages. Le mécanisme du défaut de page chargera la page en mémoire lorsqu'il y sera fait référence. Tous les transferts avec la mémoire principale sont ainsi placés sous le contrôle d'un mécanisme unique.

Du point de vue de la gestion de l'espace, il faut associer au fichier la liste des emplacements qu'il occupe. Cette information peut être contenue dans le descripteur du fichier, ensemble d'informations extérieures au fichier et permettant en particulier de localiser chacun de ses articles dans la mémoire externe. L'utilisation du chaînage entre emplacements ou suites d'emplacements pour représenter l'espace libre ou l'espace occupé par un fichier s'applique mal en mémoire secondaire, l'accès à chacun des maillons de la chaîne nécessitant une opération d'entrée-sortie.

Alors que l'unité de transfert avec la mémoire externe est la case ou la zone, l'unité d'allocation est généralement différente. Pour condenser la description de l'espace alloué à chaque fichier, et aussi pour appeler moins souvent la procédure d'allocation, on peut allouer en une seule fois plusieurs cases, pistes ou cylindres consécutifs qui seront ensuite utilisés pour la création de zones. L'espace occupé par un fichier est alors décrit sous la forme d'une suite d'extensions représentées chacune par un couple (adresse d'origine, nombre d'unités élémentaires). L'espace libre peut aussi constituer un fichier dont chacune des extensions décrit un ensemble d'unités élémentaires contiguës et libres. On retrouve alors des algorithmes d'allocation qui s'apparentent à ceux employés pour gérer la mémoire par zones (cf. 4.442).

Lorsqu'il existe plusieurs unités de mémoire externe et que le choix de l'unité n'est pas imposé à l'allocateur, ce dernier peut utiliser l'une des deux approches suivantes :

— allouer l'espace sur les unités les moins remplies,

— allouer l'espace de manière à équilibrer, en termes de transferts, l'activité des diverses unités.

Exemple. Dans le système OS/VS sur IBM/370, les fichiers temporaires (ceux, par exemple, qui sont utilisés pendant une compilation et qui disparaissent ensuite) sont alloués sur les unités qui ont été les moins sollicitées pendant l'intervalle de temps précédent.

4.6 STRATÉGIES GLOBALES

On constate à l'expérience que la mise en œuvre de stratégies individuelles et indépendantes pour l'allocation de processeur et de mémoire principale peut donner des résultats catastrophiques à partir d'une certaine charge du système. Une conception globale de l'allocation des ressources permet de maîtriser ces phénomènes de dégradation brutale des performances.

4.61 PHÉNOMÈNE D'ÉCROULEMENT DU SYSTÈME

Soit un système multiprogrammé, à mémoire paginée gérée selon une stratégie de remplacement appliquée à l'ensemble des processus. Quand le nombre des processus augmente, chacun d'eux reçoit en moyenne moins de cases et il est bloqué plus longtemps en attente de page : les temps de réponse sont augmentés et les performances du système se dégradent. Dans de nombreux cas, on constate que cette dégradation n'est pas progressive : il existe un seuil de charge au-delà duquel elle est très rapide et prend la forme d'un « écroulement » du système.

Ce phénomène est dû :

— à l'augmentation du nombre de transferts de pages,

— au fait que l'on peut retirer des pages à un processus qui est lui-même en attente de page.

L'écroulement peut recevoir une explication qualitative [Denning, 68a] reposant sur l'observation du premier de ces facteurs. Considérons un processus qui exécute V instructions et soit p la probabilité de défaut de page par instruction. Nous supposons que p n'est fonction que de la taille de mémoire principale allouée au processus et qu'un régime permanent est atteint (l'effet de l'initialisation de la mémoire est négligé). L'allure de la courbe représentative de p en fonction de la taille de mémoire allouée, obtenue expérimentalement, est donnée par la figure 18.

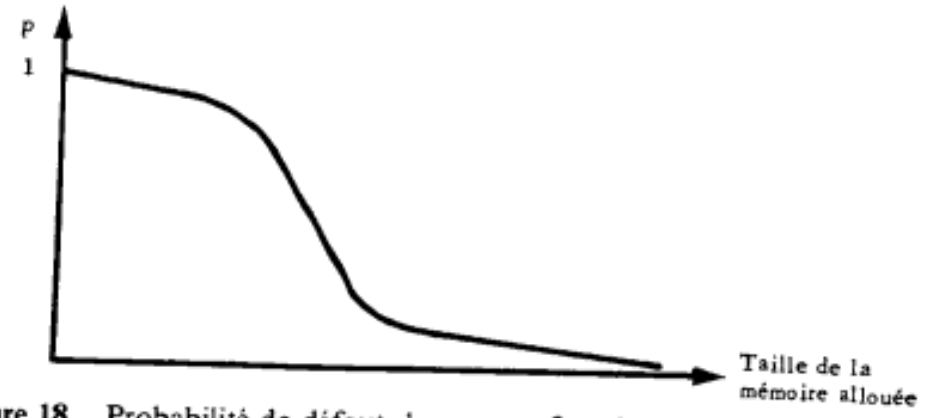


Figure 18. Probabilité de défaut de page en fonction de la taille de mémoire.

Supposons maintenant la mémoire partagée de manière égale entre n processus. Quand n augmente, la taille de la mémoire allouée à chacun d'eux décroît et p augmente. La courbe représentative de p en fonction du nombre de processus est donnée figure 19.

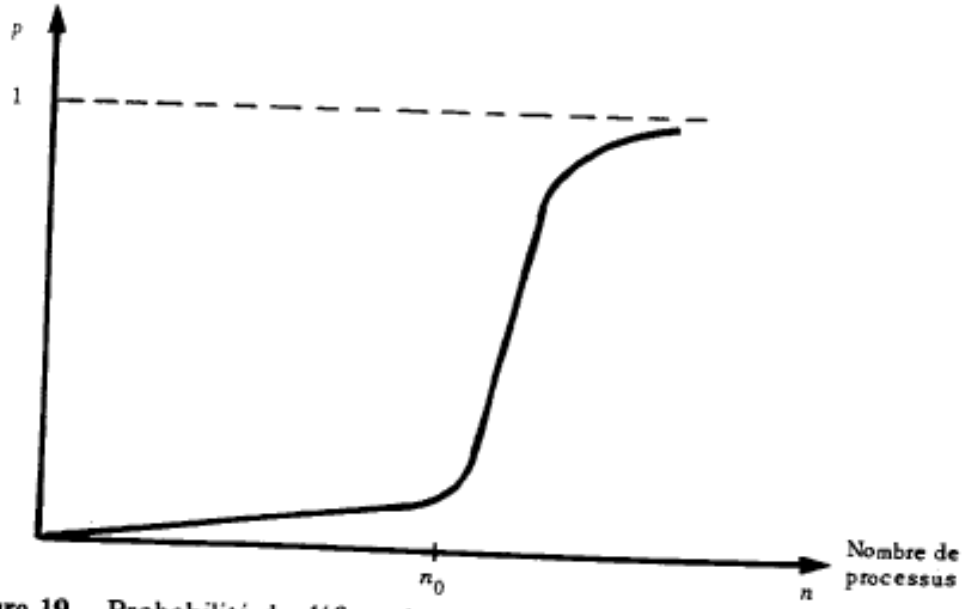


Figure 19. Probabilité de défaut de page en fonction du nombre de processus.

A partir d'un seuil n_0, l'introduction d'un processus supplémentaire provoque un accroissement brutal de p. Etudions l'influence de cette variation sur les performances du système. Définissons l'efficacité d'utilisation du processeur par un processus, $e(p)$, par :

$$e(p) = \frac{\textit{temps d'exécution}}{\textit{temps d'exécution} + \textit{temps d'attente de page}}$$

Soit S le temps, exprimé en nombre d'instructions, séparant la demande d'une page de son arrivée en mémoire. On peut écrire :

$$e(p) = \frac{V}{V + VpS} = \frac{1}{1 + pS}$$

La variation de $e(p)$ en fonction de p est représentée sur la figure 20.

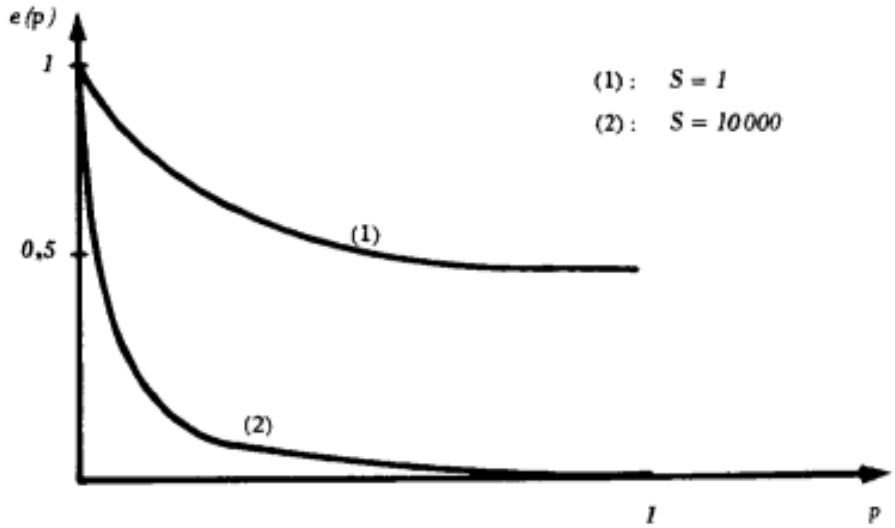


Figure 20. Influence du temps de transfert de page sur l'efficacité du processeur.

Si l'on part d'un état initial où p est faible, donc $e(p)$ élevé, on constate qu'un accroissement de p se traduit par une chute de $e(p)$, d'autant plus brutale que S est grand. Deux phénomènes contribuent donc à l'écroulement : l'accroissement rapide de la probabilité de défaut de page au-dessus d'une certaine charge et la valeur élevée de S.

Exemple. Lorsque p passe de 10^{-4} à 10^{-1}, $e(p)$ passe de 1 à $0{,}9$ si $S = 1$ et de $0{,}5$ à $0{,}0001$ si $S = 10^4$.

Les paragraphes qui suivent présentent deux approches qui permettent d'éviter l'écroulement des systèmes multiprogrammés. L'une est fondée sur une régulation de la charge du système, l'autre sur une régulation de la taille de mémoire allouée à chaque processus.

4.62 RÉGULATION DE LA CHARGE

Les méthodes de régulation de la charge (« load levelling ») consistent à agir sur le degré de multiprogrammation, c'est-à-dire sur le nombre de processus partiellement chargés en mémoire. Elles diffèrent par les points suivants.

1) Estimation de la charge : la mesure de charge utilisée pour la régulation peut mettre en jeu divers paramètres tels que le nombre de pages utilisées par processus, le taux d'activité de l'unité centrale et le nombre de transferts entre niveaux de mémoire.

2) Mode d'action sur le degré de multiprogrammation : l'action peut être brutale, et dans ce cas le régulateur chasse provisoirement un processus de la mémoire ou en introduit un autre, ou progressive, et alors un processus se voit retirer progressivement la mémoire qu'il occupe.

Diverses méthodes ont été expérimentées [Shils, 68 ; Wulf, 69 ; Alderson, 72] et certaines sont effectivement en usage dans des systèmes.

Exemple. Le système OS/VS sur IBM/370 utilise une méthode de régulation brutale déclenchée lorsqu'une nouvelle demande de page ne peut être satisfaite à partir d'une liste de pages disponibles.

Nous illustrons le principe de la régulation de charge par une description du système expérimental M44/44X [Shils, 68 ; Brawn, 68].

La charge est estimée par les deux paramètres suivants :

— l'activité relative de l'unité centrale pendant un intervalle de temps Δt,
— le nombre de pages remplacées pendant Δt.

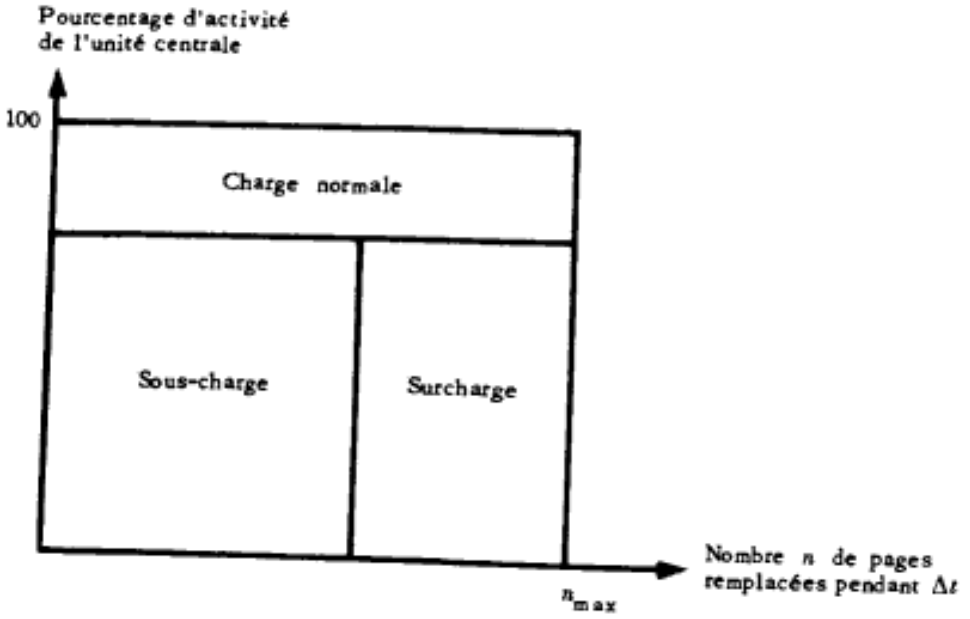


Figure 21. Diagramme d'activité du M44/44X [Shils, 68].

Le diagramme d'activité est représenté sur la figure 21. Il comprend trois parties :

— une zone de charge normale, caractérisée par une activité élevée de l'unité centrale,

— une zone de sous-charge, où le nombre de pages remplacées est faible et où l'unité centrale est relativement oisive,

— une zone de surcharge, qui correspond à un important remplacement de pages et à une faible activité de l'unité centrale.

Dans l'exemple étudié, les limites entre les différentes zones ont été déterminées expérimentalement et l'intervalle Δt fixé à 10 secondes.

Le régulateur inspecte, tous les Δt, l'activité du système et :

— en cas de surcharge, exclut un travail de la mémoire,

— en cas de sous-charge, admet en mémoire, soit un nouveau travail, soit un travail précédemment exclu, s'il en existe un,

— en cas de charge normale, ne fait rien.

L'effet du régulateur de charge est illustré par les deux matrices de la figure 22, qui donnent les probabilités de transition d'une zone à l'autre avec et sans régulateur. Par exemple, si le système est en surcharge, la probabilité de le

Zone initiale \ Zone finale	Sous-charge	Normal	Surcharge
Sous-charge	0,73	0,11	0,16
Normal	0,19	0,69	0,13
Surcharge	0,33	0,23	0,44

a) Probabilité de transition avec le régulateur.

Zone initiale \ Zone finale	Sous-charge	Normal	Surcharge
Sous-charge	0,83	0,09	0,05
Normal	0,10	0,67	0,23
Surcharge	0,04	0,19	0,77

b) Probabilité de transition sans le régulateur.

Figure 22. Matrices des probabilités de transition du M44/44X [Shils, 68].

retrouver encore en surcharge après un temps Δt est 77 % sans régulateur et seulement 44 % avec régulateur. Par contre, les effets du régulateur sont brutaux ; ainsi la probabilité de passer de la zone de charge normale à la zone de sous-charge, qui n'est que de 10 % sans régulateur passe à 19 % avec le régulateur ; de même la probabilité de passage de la zone de surcharge à la zone de sous-charge atteint 33 % avec régulateur, alors qu'elle est de 4 % sans régulateur. La régulation de charge, comme tout mécanisme de régulation, peut donc introduire des phénomènes d'instabilité [Wilkes, 71].

Les courbes de la figure 23 [Brawn, 68] montrent l'effet du régulateur de charge au cours d'expériences où plusieurs copies du même programme ont été multiprogrammées.

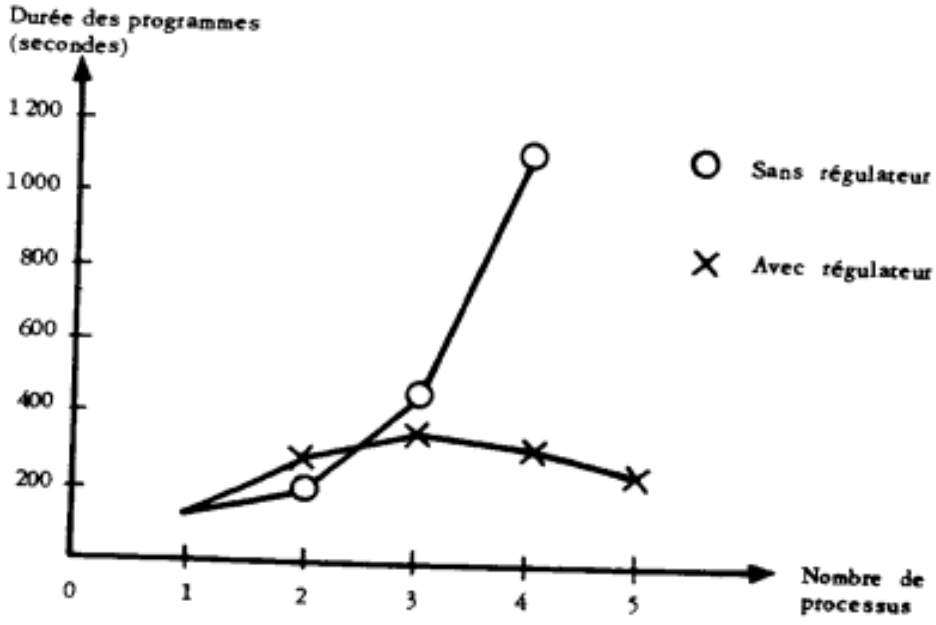


Figure 23. Influence du régulateur de charge [Brawn, 68].

4.63 STRATÉGIES FONDÉES SUR L'ESPACE DE TRAVAIL

La notion d'**espace de travail** (« working set ») est utilisée pour adapter, en fonction du comportement dynamique de chaque processus, la taille de mémoire qui lui est allouée. Nous décrivons ici la mise en œuvre de cette régulation.

4.631 Notion d'espace de travail

On appelle espace de travail $W(t, \tau)$ d'un processus, à l'instant t et pendant le temps τ, l'ensemble des pages référencées entre les instants $t - \tau$ et t. On trouvera certaines propriétés de l'espace de travail dans [Denning, 68*b*].

Lorsque les programmes présentent la propriété de localité, un processus a une probabilité beaucoup plus grande de se référer à des pages récemment

utilisées qu'à des pages quelconques. En conséquence, si le paramètre τ est adapté au comportement du programme, $W(t, \tau)$ est une bonne approximation de l'ensemble des pages qui vont être référencées entre t et $t + \tau$. On considère que τ est adapté au programme si pendant des périodes assez longues la taille et la composition de l'espace de travail varient lentement avec le temps (l'échelle de temps étant fixée par la durée moyenne d'exécution d'une instruction).

4.632 Mise en œuvre de stratégies fondées sur l'espace de travail

Dans une stratégie fondée sur l'espace de travail, on cherche à garantir que l'unité centrale n'est allouée à un processus que si les pages de son espace de travail sont chargées en mémoire. D'après la formule

$$e(p) = 1/(1 + pS) \qquad \text{(cf. 4.61)}$$

il faut essayer de maintenir p à une valeur aussi faible que possible pour prévenir l'écroulement du système dans le cas où S a une valeur élevée. Si chaque processus actif dispose à tout instant de son espace de travail en mémoire centrale, alors on est assuré que p reste petit, puisque $W(t, \tau)$ est une bonne approximation de $W(t + \tau, \tau)$.

Comme la détermination de l'identité des pages de l'espace de travail est coûteuse, on se contente généralement de réserver pour chaque processus un nombre de cases égal à la taille de son espace de travail. On admet alors en mémoire de taille M un nombre n de processus tel que

$$\sum_{i=1}^{n} | W_i(t, \tau) | \leqslant M$$

et on retrouve ainsi, comme en 4.61, un nombre maximal n_0 ; ce nombre est ajusté cette fois-ci en fonction du comportement individuel de chaque processus.

La valeur de τ est un paramètre important de ces stratégies ; on trouve dans [Denning, 68*b*] une approche analytique de son choix. Donnons ici quelques éléments d'appréciation.

1) La taille de l'espace de travail est une fonction non décroissante de τ.

2) Pour un τ relativement grand, l'espace de travail occupe beaucoup de pages ; le système doit donc comporter une grande mémoire principale et peut en contrepartie se contenter de mémoires secondaires à grand temps de transfert.

3) Inversement, si τ est petit, le système peut comporter une petite mémoire principale mais doit avoir des mémoires secondaires très rapides.

4.633 Détermination pratique de l'espace de travail

La détermination, en temps réel, de l'espace de travail est plus ou moins coûteuse selon les dispositifs câblés disponibles.

Si on ne dispose que d'un mécanisme de protection des pages virtuelles, on doit à l'instant $t - \tau$ interdire tout accès aux pages ; à chaque déroutement pour défaut de page on note la page utilisée et on autorise son accès. La complexité du programme d'analyse du déroutement rend cette technique peu efficace.

Si on dispose d'un mécanisme câblé faisant passer à 1, lors d'une référence à la page, le bit d'utilisation associé (cf. 4.451), il suffit de remettre tous ces bits à 0 à l'instant $t - \tau$, et d'exclure de l'espace de travail à l'instant t toutes les pages dont le bit d'utilisation est resté à 0.

Si on dispose non plus d'un seul bit mais d'un compteur associé à chaque page, alors on peut implanter dans de bonnes conditions des algorithmes LRU [Corbato, 69*a*].

D'autres techniques sont proposées dans [Denning, 68*c*].

4.634 Tentatives d'adaptation du comportement des programmes

Pour τ fixé, la taille $| W(t, \tau) |$ est d'autant plus petite que les références à la mémoire sont moins dispersées. On peut donc essayer d'améliorer le rendement global du système en organisant les programmes de manière à réduire la dispersion des références.

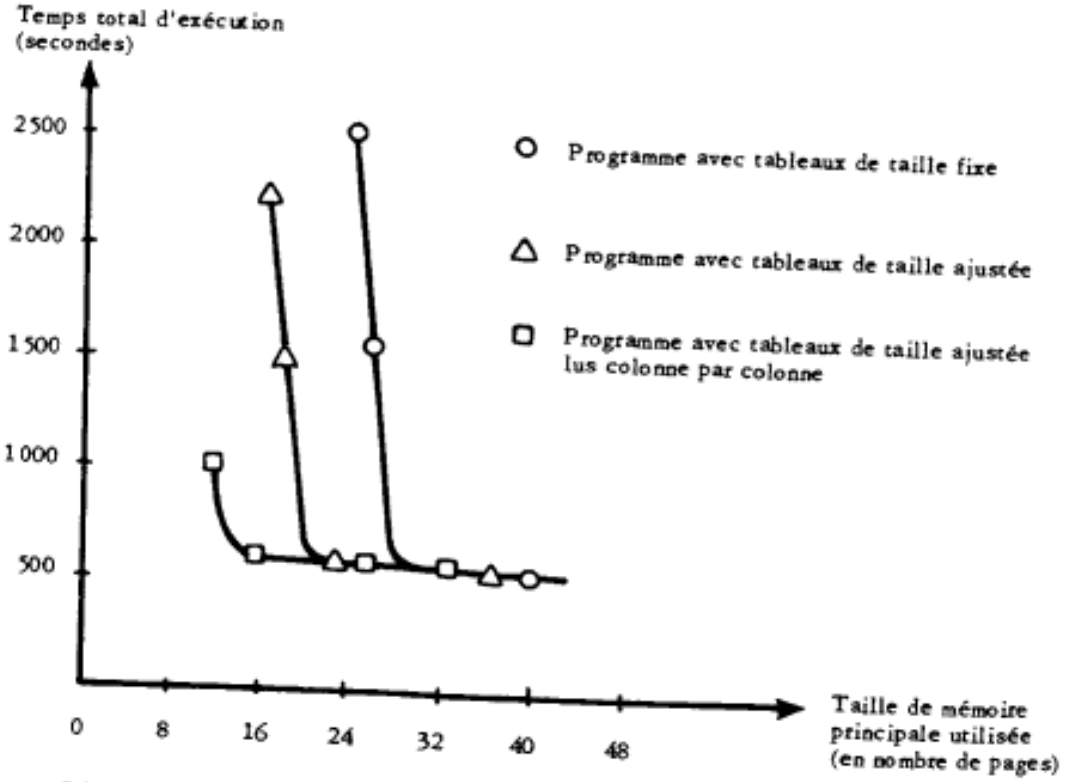


Figure 24. Influence du style de programmation sur le comportement d'un programme [Brawn, 68].

Deux approches ont été tentées.

1) Une permutation de l'ordre des modules d'un système [Comeau, 67 ; Hatfield, 71] a conduit, dans un cas particulier et après quelques essais, à une diminution de 50 % du nombre de défauts de page.

2) L'écriture complète de programmes en tenant compte de la pagination [Brawn, 68] a permis d'obtenir une diminution du temps d'exécution beaucoup plus importante que celle qui résultait de raffinements de l'algorithme de remplacement.

Exemple. Un programme d'inversion de matrice, basé sur la méthode de Gauss, a été écrit de trois façons différentes pour tenir compte de la pagination [Brawn, 68] :

a) avec des tableaux de taille fixe prévus pour des matrices de dimension maximale 150,

b) avec des tableaux ajustés selon les dimensions des matrices, et lus ligne par ligne,

c) avec des tableaux ajustés selon les dimensions des matrices et lus colonne par colonne.

Les temps d'exécution correspondant sont donnés figure 24.

4.7 INTERBLOCAGE

4.71 INTRODUCTION

Dès qu'on alloue des ressources non sujettes à réquisition à des processus qui s'exécutent concurremment, il apparaît un risque de blocage mutuel de ces processus lorsque, pour certains types de ressource, la demande totale est supérieure au nombre de points d'accès. En effet, les demandes de ressources de différents processus peuvent être satisfaites dans un ordre tel que deux ou plusieurs d'entre eux se bloquent indéfiniment : chacun de ces processus accapare des ressources tout en attendant celles possédées par les autres.

Exemple 1. Un processus p doit mettre à jour un fichier sur disque et attend que l'unique tourne-disque soit utilisable. Celui-ci est alloué à un processus q qui attend pour le libérer que le fichier soit mis à jour.

Exemple 2. Deux processus p et q utilisent des sémaphores $s1$, $s2$ initialisés à 1. Le processus p doit exécuter la séquence $P(s1)$; ... ; $P(s2)$ et le processus q la séquence $P(s2)$; ... ; $P(s1)$.

Si l'unité centrale est allouée de manière telle que p effectue $P(s1)$ puis que q effectue $P(s2)$, p et q se bloqueront dès l'opération P suivante.

Dans ces deux cas simples, les processus p et q sont **interbloqués.**

L'interblocage peut mettre en jeu un nombre important de processus et de ressources. Le déblocage de tous les processus **(guérison)** peut être très complexe, voire impossible sans destruction d'une partie ou de la totalité des processus interbloqués. Il est donc intéressant de rechercher des techniques d'allocation évitant l'apparition de l'interblocage **(prévention).**

4.72 DESCRIPTION INFORMELLE

Peut-on prévoir l'interblocage ? Caractérisons le degré de progression d'un processus à l'instant t par le nombre $T(t)$ d'instructions exécutées depuis l'instant initial, fonction non décroissante du temps. Soit deux processus p, q et $T1$, $T2$ leurs degrés de progression. Représentons la courbe de progression relative des deux processus, définie par l'équation $F(T1, T2) = 0$ obtenue par élimination de t (Fig. 25). Cette courbe peut présenter des segments parallèles aux axes indiquant qu'un des processus est bloqué.

Si p et q ont besoin au cours de leur progression d'une ressource R à un point d'accès, l'exclusion mutuelle des processus par rapport à R détermine un pavé P que la courbe F ne peut traverser.

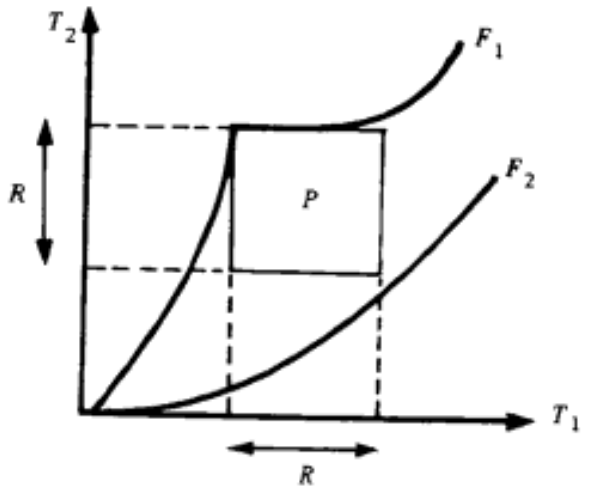


Figure 25. Utilisation d'une ressource par deux processus.

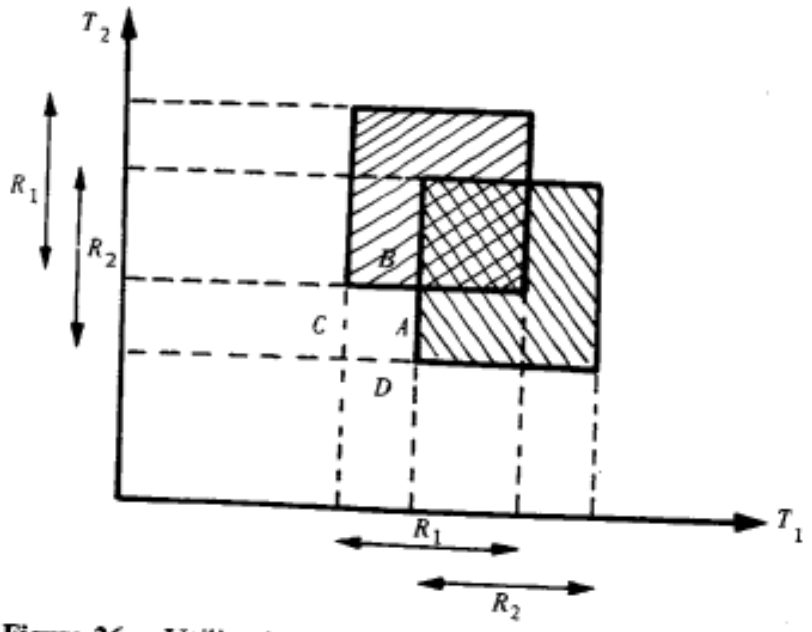


Figure 26. Utilisation de deux ressources avec interblocage.

Considérons maintenant (Fig. 26) le cas où les processus p et q demandent l'accès exclusif à deux ressources *R1* et *R2* de la manière suivante :

p demande *R1* puis *R2*,
q demande *R2* puis *R1*.

Bien que l'interblocage ne soit effectif qu'au moment où la courbe atteint les côtés A ou B, il devient inévitable dès qu'elle pénètre dans le pavé $ABCD$, c'est-à-dire dès allocation de la seconde ressource (côtés C ou D).

La prévention de l'interblocage semble possible en examinant la situation créée à chaque allocation de ressources à condition de connaître les demandes ultérieures de ressources des processus. Si les requêtes de p et q sur les ressources *R1* et *R2* sont faites dans le même ordre, l'interblocage ne peut apparaître (Fig. 27). On montrera dans la suite (cf. 4.742) que cette solution s'applique dans des cas plus généraux.

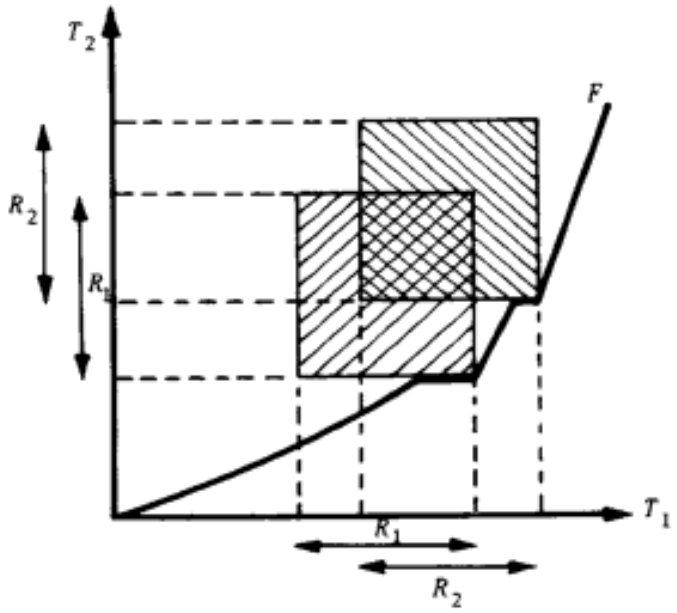


Figure 27. Utilisation de deux ressources sans interblocage.

4.73 FORMALISATION ET DÉFINITIONS

4.731 Système. Etat d'un système

Définition. Dans le contexte de ce chapitre, un système est constitué :

— d'un ensemble fini de processus séquentiels pouvant s'exécuter concurremment,

$$P = \{ p_1, \ldots, p_n \} ,$$

— d'un ensemble de classes de ressources à un point d'accès

$$E = \{ R_1, \ldots, R_m \} .$$

Nous appelons ici **classe** de ressources un ensemble de ressources banalisées.

L'état initial du système est décrit par le vecteur X donnant le nombre total de ressources existant dans chaque classe :

$$X = \begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix}.$$

Par hypothèse, nous supposons que :

— le vecteur X reste constant pendant la durée d'observation du système,

— tous les processus libèrent au bout d'un temps fini toutes les ressources qu'ils ont acquises.

Définition. A un instant t, l'état du système est défini par la matrice $A(t)$ des ressources allouées aux processus et par la matrice $D(t)$ des ressources demandées par les processus :

$$A(t) = \begin{pmatrix} a_{11}(t) & \dots & a_{1n}(t) \\ \vdots & & \vdots \\ a_{m1}(t) & \dots & a_{mn}(t) \end{pmatrix} = (A_1(t), \dots, A_n(t))$$

où $a_{ij}(t)$ est le nombre de ressources de la classe R_i allouées au processus p_j

$$D(t) = \begin{pmatrix} d_{11}(t) & \dots & d_{1n}(t) \\ \vdots & & \vdots \\ d_{m1}(t) & \dots & d_{mn}(t) \end{pmatrix} = (D_1(t), \dots, D_n(t))$$

où $d_{ij}(t)$ est le nombre de ressources de la classe R_i demandées par le processus p_j.

L'allocation des ressources, les changements d'état du système se font uniquement par les trois opérations suivantes :

— requête : un processus p_i fait une demande de ressources représentée par un vecteur N :

$$D_i(t) := D_i(t) + N$$

— acquisition : des ressources représentées par un vecteur M sont allouées au processus p_i :

$$A_i(t) := A_i(t) + M$$

— libération : un processus p_i libère des ressources représentées par un vecteur H :

$$A_i(t) := A_i(t) - H$$
$$D_i(t) := D_i(t) - H$$

Remarque 1. La matrice D n'est diminuée que lors d'une libération de ressources.

Remarque 2. Une requête n'est pas toujours immédiatement suivie d'une allocation mais peut être mise en attente ; cette remarque justifie l'introduction des deux matrices D et A.

Dans la suite nous utiliserons les notations ci-après :
Soit U et V des vecteurs à m éléments ; par définition :

$$U \leqslant V \Leftrightarrow U_i \leqslant V_i \qquad \forall i \in [1, m]$$
$$U < V \Leftrightarrow (U \leqslant V) \quad \underline{et} \quad (\exists i \text{ tel que } U_i < V_i)$$

Soit M et N des matrices à $m \times n$ éléments

$$M \leqslant N \Leftrightarrow M_i \leqslant N_i \qquad \forall i \in [1, n]$$
$$M < N \Leftrightarrow (M \leqslant N) \quad \underline{et} \quad (\exists i \text{ tel que } M_i < N_i)$$

Les conditions suivantes doivent être vérifiées :

— un processus donné ne peut demander plus de ressources qu'il n'en existe dans le système, ce qui se traduit par :

$$D_i(t) \leqslant X \qquad \forall i \in [1, n] \tag{1}$$

— un processus ne peut posséder à un instant donné plus de ressources qu'il n'en a demandé, ce qui se traduit par :

$$A_i(t) \leqslant D_i(t) \qquad \forall i \in [1, n] \tag{2}$$

— la somme des acquisitions de tous les processus à un instant donné ne peut dépasser la totalité des ressources du système, ce qui se traduit par :

$$\sum_{i=1}^{n} A_i(t) \leqslant X \tag{3}$$

Définition. L'état du système à un instant t est un **état réalisable** si et seulement si $A(t)$, $D(t)$ et X vérifient les relations (1), (2) et (3).

Les vecteurs $D_i(t)$ et $A_i(t)$ définissent l'état d'allocation des ressources au processus p_i. Celui-ci est bloqué tant que $D_i(t) > A_i(t)$.

Appelons $R(t)$ le vecteur décrivant le nombre de ressources disponibles à l'instant t dans chaque classe :

$$R(t) = X - \sum_{i=1}^{n} A_i(t)$$

La relation (3) peut s'écrire :

$$R(t) \geqslant 0$$

4.732 **Interblocage**

L'absence d'interblocage peut être garantie à l'instant t s'il existe à partir de cet instant une suite d'états réalisables du système tels que tous les processus obtiennent les ressources qu'ils demandent et s'exécutent jusqu'à leur fin. Nous allons démontrer (théorème 1) que trouver une telle suite d'états équivaut à ranger les processus dans un ordre tel que leurs demandes puissent être satisfaites s'ils s'exécutent dans cet ordre.

Soit S une telle suite de processus ; notons $S(i)$ le rang du processus p_i dans la suite S et supposons en outre qu'à l'instant t considéré, chaque processus ait demandé toutes les ressources nécessaires à son exécution. La demande non satisfaite du premier processus de la suite doit être inférieure aux ressources disponibles à l'instant t :

$$S(i_1) = 1 \qquad D_{i_1}(t) - A_{i_1}(t) \leqslant R(t)$$

Ce processus s'exécute puis libère ses ressources. Le second processus peut alors s'exécuter à condition que :

$$S(i_2) = 2 \qquad D_{i_2}(t) - A_{i_2}(t) \leqslant R(t) + A_{i_1}(t)$$

et ainsi de suite pour les autres processus. La suite S vérifie donc la relation :

$$\forall p_i \in P \qquad D_i(t) - A_i(t) \leqslant R(t) + \sum_{S(j) < S(i)} A_j(t)$$

Introduisons les définitions suivantes :

Définitions.

— Une suite S de processus est dite **saine** pour une demande $D(t)$ donnée, si et seulement si elle vérifie la relation :

(4) $$D_i(t) - A_i(t) \leqslant R(t) + \sum_{S(j) < S(i)} A_j(t) \qquad \forall p_i \in S$$

— Une suite S est dite **complète** sur P si et seulement si elle contient tous les processus de P.

— Un état réalisable du système est dit **sain** si et seulement s'il existe une suite saine complète de processus dans cet état.

Supposons qu'à l'instant t_0 les processus ont demandé toutes les ressources nécessaires à leur exécution.

Théorème 1. Si l'état du système est sain, alors il existe une suite d'états réalisables du système telle que tous les processus obtiendront les ressources qu'ils demandent, et réciproquement.

Soit S une suite saine pouvant être construite à l'instant initial t_0. Considérons la suite de n états $E_k = (A(t_k), D(t_k), X)$, définie comme suit :

$$A_i(t_k) = 0 \qquad \forall i \text{ tel que } S(i) < k$$
$$A_i(t_k) = D_i(t_0) \qquad i \text{ tel que } S(i) = k$$
$$A_i(t_k) = A_i(t_0) \qquad \forall i \text{ tel que } S(i) > k$$

$$D_i(t_k) = 0 \qquad \forall i \text{ tel que } S(i) < k$$
$$D_i(t_k) = D_i(t_0) \qquad \forall i \text{ tel que } S(i) \geqslant k$$

Le vecteur X est constant par hypothèse.

Montrons que ces états sont des états réalisables du système. L'état initial E_0 du système est sain et vérifie donc les relations (1), (2), (3) et (4),

$$D_i(t_0) \leqslant X \qquad \forall i \in [1, n]$$
$$A_i(t_0) \leqslant D_i(t_0) \qquad \forall i \in [1, n]$$
$$\sum_{i=1}^{n} A_i(t_0) \leqslant X$$
$$D_i(t_0) - A_i(t_0) \leqslant R(t_0) + \sum_{S(j) < S(i)} A_j(t_0) \qquad \forall i \in [1, n]$$

Par définition de la suite des états E_k, ceux-ci vérifient les conditions 1 et 2. Montrons qu'ils vérifient la condition 3. Soit $Y = \sum_{i=1}^{n} A_i(t_k)$. On a, par définition de $A(t_k)$:

$$Y = \sum_{S(i) < k} A_i(t_k) + \sum_{S(i) > k} A_i(t_k) + A_{i_k},$$

où i_k est tel que $S(i_k) = k$, ou encore :

$$Y = 0 + \sum_{S(i) > k} A_i(t_0) + D_{i_k}(t_0).$$

En remplaçant $D_{i_k}(t_0)$ par son majorant (relation (4)), il vient :

$$Y \leqslant R(t_0) + \sum_{S(j) \leqslant k} A_j(t_0) + \sum_{S(j) > k} A_j(t_0),$$

ou
$$Y \leqslant X$$

Les états E_k sont des états réalisables du système ; il est facile de vérifier que ces états sont sains.

Réciproquement, soit une suite d'états réalisables du système telle que tous les processus obtiennent les ressources qu'ils demandent. On peut extraire de cette suite la suite S_1 des états dans laquelle chaque état correspond à l'acquisition par un processus des ressources demandées

$$S_1 = (D(t_i), A(t_i), X) \qquad \forall i \in [1, n]$$

Chaque état $(D(t_i), A(t_i), X)$ est caractérisé par : $\exists k$ tel que $A_l(t_k) = D_l(t_k)$, où $A_l(t)$ est une fonction non décroissante pour $t \in [t_0, t_k]$.

Soit S la suite des processus définie comme suit :

$S(l) = k$ si et seulement si p_l est le processus dont l'annonce est satisfaite à l'instant t_k. Cette définition implique $A_l(t_k) \geqslant (A_l(t_0)$ pour $S(l) \geqslant k$.

Tous les états étant réalisables, on a :

$$\sum_{l=1}^{n} A_l(t_k) \leqslant X$$

Pour tout instant t_k, il existe p_l tel que

$$D_l(t_k) + \sum_{S(j) \neq S(l)} A_j(t_k) \leqslant X \qquad (S(l) = k)$$

d'où :

$$D_l(t_k) \leqslant X - \sum_{S(j) \neq S(l)} A_j(t_k)$$

$$D_l(t_k) \leqslant X - \sum_{S(j) > S(l)} A_j(t_k)$$

$$D_l(t_k) \leqslant X - \sum_{S(j) > S(l)} A_j(t_i) , \qquad \forall t_i \leqslant t_k$$

$$D_l(t_k) \leqslant R(t_i) + \sum_{S(j) \leqslant S(l)} A_j(t_i) , \quad \forall t_i \leqslant t_k .$$

Or on a, par hypothèse

$$D_l(t_k) = D_l(t_0) , \qquad \forall k$$

d'où :

$$D_l(t_i) \leqslant R(t_i) + \sum_{S(j) \leqslant S(l)} A_j(t_i) , \quad \forall t_i \leqslant t_k$$

La suite S est donc une suite saine du système à tout instant.

Le théorème 1 permet simplement de reconnaître par examen de l'état du système que celui-ci n'est pas en interblocage. Il ne garantit pas que les états suivants soient sains ; en effet, l'hypothèse qu'aucun des processus ne fera de nouvelles demandes n'est pas nécessairement vérifiée dans la réalité.

4.74 REMÈDES A L'INTERBLOCAGE

Une politique de prévention de l'interblocage doit contrôler les opérations d'acquisition de sorte que l'état du système reste sain. Une politique de guérison doit être capable de sortir le système d'un état d'interblocage en le remettant dans un état sain.

4.741 **Détection. Guérison**

Le premier problème à résoudre est de savoir si le système est en interblocage.

4.7411 *Détection*

La méthode générale consiste à essayer de construire une suite saine de processus. Si cette suite est complète, il n'y a pas d'interblocage, sinon les processus n'appartenant pas à cette suite sont interbloqués. Le théorème suivant facilite la recherche d'une suite saine complète :

Théorème 2. Si l'état du système est sain, toute suite saine incomplète S peut être prolongée en une suite saine complète.

Comme l'état du système est sain, il existe une suite saine complète Q :

$$D_k \leqslant R + \sum_{Q(l) \leqslant Q(k)} A_l \qquad \forall p_k \in Q \tag{5}$$

Soit S une suite saine :

$$D_k \leqslant R + \sum_{S(l) \leqslant S(k)} A_l \qquad \forall p_k \in S \tag{6}$$

Définissons une suite S_1 telle que :

$$S_1(k) = S(k) \qquad \forall p_k \in S$$

$$Q(l) \leqslant Q(k) \Rightarrow S_1(l) \leqslant S_1(k) \qquad \forall p_k, p_l \in Q - S$$

$$\sum_{Q(l) \leqslant Q(k)} A_l \leqslant \sum_{S_1(l) \leqslant S_1(k)} A_l \qquad \forall p_k \in Q - S \tag{7}$$

car tout p_l qui précède p_k dans Q le précède aussi dans S_1, par définition de S_1. Les relations (5) et (7) impliquent :

$$D_k \leqslant R + \sum_{S_1(l) \leqslant S_1(k)} A_l \qquad \forall p_k \in Q - S \tag{8}$$

Les relations (6) et (8) impliquent alors :

$$D_k \leqslant R + \sum_{S_1(l) \leqslant S_1(k)} A_l \qquad \forall p_k \in S_1$$

La suite S_1 est donc une suite saine complète.

Le théorème 1 indique que le système n'est pas interbloqué à un moment donné s'il existe une suite saine dans son état actuel.

Corollaire 1. S'il n'existe aucune suite saine pouvant être prolongée en une suite saine complète alors le système est interbloqué.

L'algorithme décrit ci-après recherche une suite saine complète. Précisons les notations :

{ } définit un ensemble,
$\varnothing$ désigne l'ensemble vide,
$A \oplus B$ désigne l'union de deux ensembles A et B
$A \ominus B$ désigne l'ensemble complémentaire de B par rapport à A
A^* désigne l'ensemble complémentaire de A par rapport à l'ensemble P des processus du système.

Algorithme 1

début booléen *étatsain* ; *ensemble* S, T ; *entier* i, j ;
$S := \varnothing$; *(initialisation de la suite saine)*
$T := P$; *(P = ensemble des processus du système)*

```
tant que T ≠ ∅ faire
début i := indice d'un élément de T ;
      T := T ⊖ { p_i } ;
      si D_i − A_i ≤ R + Σ_S A_j
      alors début S := S ⊕ { p_i } ;
                  T := S*
           fin
fin ;
étatsain := (S* = ∅)
fin
```

La suite S est initialement saine, car elle est vide. On recherche un élément de T pouvant être ajouté à S ; si un tel processus est trouvé, T est réinitialisé et le procédé est réitéré, sinon, d'après le corollaire, le système est interbloqué.

Dans l'hypothèse où le choix de n éléments dans T se fait dans l'ordre inverse de la suite S, le coût (temps d'exécution) de l'algorithme est proportionnel à $n + (n-1) + \cdots + 1 = n(n+1)/2$; c'est le cas le plus défavorable. Le coût peut être réduit en prenant comme état initial la suite saine de l'état précédent. Il est possible dans tous les cas de construire une suite saine par un algorithme de coût linéaire par rapport à n [Holt, 71]. L'inconvénient de cet algorithme réside en une phase importante d'initialisation (coût de l'ordre de n).

Sous cette forme, ces algorithmes sont utilisables pour une détection occasionnelle de l'interblocage, soit périodiquement, soit à la suite d'une baisse de performance du système, détectée par exemple par l'opérateur. Ils peuvent aussi être utilisés à chaque changement d'état du système pour vérifier que le nouvel état est sain. Tant qu'un processus ne fait pas de nouvelles requêtes, il existe au moins une façon d'allouer les ressources demandées : c'est l'ordre de la suite saine complète de l'état initial. Par contre, lorsqu'un processus fait une nouvelle demande, l'interblocage peut apparaître. La détection continue n'est donc nécessaire qu'au moment de l'exécution des requêtes.

Théorème 3. L'exécution d'une requête par un processus fait passer d'un état sain à un nouvel état sain s'il existe une suite saine contenant ce processus.

A et B étant deux suites, nous noterons A, B la suite obtenue en concaténant A et B dans cet ordre. Soit E_0 l'état initial à l'instant t_0 et p_i un processus non bloqué dans cet état ; on a :

$$D_i(t_0) = A_i(t_0)$$

Soit E_1 l'état final obtenu à l'instant t_1 par l'exécution d'une requête de p_i.

$$\begin{aligned} D_j(t_1) &= D_j(t_0) \qquad \forall j \neq i \\ A_j(t_1) &= A_j(t_0) \qquad \forall j \\ D_i(t_1) &> D_i(t_0) \end{aligned}$$

On a : $R(t_1) = R(t_0)$

puisqu'aucune acquisition n'a eu lieu.

Soit $S = S_1, \{ p_i \}$ une suite saine dans l'état E_1.

$$D_i(t_1) - A_i(t_1) \leqslant R(t_1) + \sum_{S(j) < S(i)} A_j(t_1) \qquad \forall p_i \in S$$

Les relations précédentes impliquent que S est une suite saine de l'état E_0. D'après le théorème 2, elle peut être prolongée en une suite saine complète S_2 pour l'état E_0. Montrons que S_2 est une suite saine de l'état E_1. Posons $S_2 = S, S_3$

$$D_k(t_0) - A_k(t_0) \leqslant R(t_0) + \sum_{S_2(j) < S_2(k)} A_j(t_0) \qquad \forall p_k \in S_3$$

Les propriétés de l'état E_1 impliquent :

$$D_k(t_1) - A_k(t_1) \leqslant R(t_1) + \sum_{S_2(j) < S_2(k)} A_j(t_1) \qquad \forall p_k \in S_3$$

S étant une suite saine dans l'état E_1, la relation précédente implique que S_2 est une suite saine complète dans l'état E_1.

Cette propriété est utilisée dans l'algorithme de détection continue qui suit :

Algorithme 2

```
début   booléen étatsain ; ensemble S, T ; entier i ;
        S := ∅ ; T := P ;
        tant que (p_k ∉ S) et (T ≠ ∅) faire
            début si p_k ∈ T alors i := k
                  sinon i := indice d'un élément de T ;
                  T := T ⊖ { p_i } ;
                  si D_i - A_i ≤ R + Σ_S A_j
                  alors début S := S ⊕ { p_i } ;
                              T := S*
                        fin
            fin ;
        étatsain := (p_k ∈ S)
fin
```

Des mesures ont montré que le temps d'exécution de cet algorithme est de l'ordre de n^2.

Cet algorithme permet la détection occasionnelle ou continue de l'interblocage. Il en existe d'autres qui s'appliquent à des cas plus simples [Holt, 71], tels que :

— ressources d'une seule classe,
— classes à une unité de ressources,
— requêtes d'une unité à la fois.

La mise en œuvre d'un algorithme de détection ne peut apporter qu'une information sur l'état du système : il est ou non en interblocage. S'il y est, il faut l'en sortir : c'est l'objet des méthodes de guérison.

4.7412 *Guérison*

Seule une action extérieure au mécanisme normal d'allocation peut sortir un système de l'interblocage, par la réquisition d'un certain nombre de ressources. Cette réquisition peut occasionner un coût non négligeable si elle entraîne la perte d'un travail et la destruction d'informations.

Il existe de nombreuses méthodes pratiques de guérison ; la plus simple consiste à détruire tous les processus interbloqués, une autre consiste à regrouper les processus en classes de même coût (priorité, type de travail). La réquisition des ressources est alors faite sur les processus des classes de plus faible coût jusqu'à ce que l'interblocage soit éliminé. En dépit de leur brutalité, ces méthodes sont d'un emploi courant.

Dans le cas d'une détection continue de l'interblocage, la solution la plus pratique consiste à détruire le processus qui vient de faire une requête. Cette méthode est insuffisante car elle ne détruit pas les processus qui causent réellement l'interblocage en monopolisant les ressources ; elle peut conduire à détruire tous les processus qui font une nouvelle demande, alors que la destruction d'un seul processus pourrait dénouer définitivement l'interblocage. Notons qu'il existe un algorithme [Shoshani, 70] qui permet de dénouer l'interblocage pour un coût minimal de réquisition. Malheureusement, son temps d'exécution et la taille de mémoire qu'il exige le rendent inutilisable en pratique.

Pour conclure, nous pouvons dire que s'il existe des algorithmes de détection de l'interblocage dont l'emploi est pratiquement concevable, l'insuffisance des techniques de guérison les rend inutilisables. Seules des méthodes de guérison brutales sont effectivement utilisées pour résoudre les blocages simples et peu fréquents (SIRIS 7).

Dans les cas plus critiques où la destruction des processus n'est pas possible, une politique de prévention doit être envisagée.

4.742 **Prévention**

Une politique de prévention introduit dans le mécanisme d'allocation des ressources une règle qui élimine toute situation pouvant conduire à un interblocage ; cela suppose que l'état initial du système est un état sain. Deux méthodes sont alors possibles :

— une méthode statique qui impose des restrictions aux demandes et aux acquisitions de ressources de façon à interdire une telle situation,

— une méthode dynamique qui reconnaît qu'une demande peut conduire à un interblocage et qui diffère l'allocation correspondante jusqu'à ce que le risque soit éliminé. Nous ne nous intéresserons pas ici à la façon dont un processus demandeur est mis en attente.

4.7421 *Prévention statique*

a) *Demande globale*

La méthode la plus simple est celle de la demande globale : tout processus doit demander et acquérir globalement ses ressources.

Dans ces conditions, il existe toujours une suite saine complète $S = S_1, S_2$, avec :

$$\forall p_i \in S_1 \qquad D_i - A_i = 0$$
$$\forall p_i \in S_2 \qquad A_i = 0$$

Tout nouveau processus arrivant dans le système est mis dans S_2 si on ne peut satisfaire sa requête, sinon il est mis dans S_1. La libération de ressources par un processus est suivie d'une recherche des processus de S_2 dont les demandes peuvent être intégralement satisfaites ; ces processus sont alors mis dans S_1. La nouvelle suite obtenue est toujours saine.

L'inconvénient majeur d'une telle méthode est de contraindre un processus à acquérir des ressources à un moment où il n'en a pas nécessairement l'usage (thésaurisation) ; cet inconvénient est réduit dans la méthode suivante, dite des classes ordonnées.

b) *Classes ordonnées*

Dans cette méthode [Havender, 68], les ressources sont regroupées en classes et l'ensemble de ces classes est ordonné. On impose alors à tout processus :

— de demander globalement les ressources de chaque classe qui lui sont nécessaires,

— de demander les ressources des différentes classes dans l'ordre des classes.

Un processus ne peut donc acquérir des ressources de la classe i que s'il ne possède pas de ressources des classes j telles que $j \geqslant i$.

Soit m le nombre de classes de ressources. Il existe toujours une suite complète $S = S_m \ldots S_l \ldots S_0$. Pour tout $p_k \in S_l$:

$$d_{jk} - a_{jk} = 0 \qquad \forall j \text{ tel que } j \leqslant l\,,$$
$$a_{jk} = 0 \qquad \forall j \text{ tel que } j > l\,.$$

Les suites S_m, $S_m\, S_{m-1}$, $S_m\, S_{m-1}\, S_{m-2}$, ..., sont des suites saines, et la suite S est une suite saine complète.

L'introduction d'un nouveau processus dans le système ne pose pas de problème : il peut toujours être placé dans la suite S_0. Lors d'une libération de ressources, le processus libérateur est éventuellement déplacé d'une suite S_i à une autre. La suite S est parcourue pour essayer de satisfaire partiellement ou totalement la demande d'autres processus en respectant les règles précédentes.

Une telle politique améliore la gestion des ressources. Les ressources les plus coûteuses doivent être placées dans les classes supérieures de manière à diminuer le temps pendant lequel elles sont inactives.

Exemple. La méthode des classes ordonnées est utilisée pour la gestion des fichiers, des périphériques et de la mémoire centrale dans le système OS/360. Il faut cependant remarquer que pour des raisons liées à la structure de ce système, la mémoire est allouée avant les périphériques.

Si l'on désire plus de souplesse, une méthode dynamique est préférable.

4.7422 *Prévention dynamique*

La méthode de prévention dynamique impose à tout processus de déclarer tous ses besoins (annonce) préalablement à son exécution et de rendre ses ressources au bout d'un temps fini. Elle a été introduite initialement sous le nom d'« algorithme du banquier » [Dijkstra, 67 ; Habermann, 69].

Dans un système à annonce, l'état est réalisable si les relations (1), (2), (3) sont vérifiées (4.731). Soit C la matrice des annonces où c_{ij} représente le nombre maximal de ressources de la classe R_i que pourra utiliser le processus p_j. On impose à un état réalisable de vérifier les nouvelles relations :

$$C_i \leqslant X \qquad \forall i \tag{9}$$

$$A_i(t) \leqslant C_i \qquad \forall i, t \tag{10}$$

$$D_i(t) \leqslant C_i \qquad \forall i, t \tag{11}$$

Cette méthode suppose qu'un processus peut exiger toutes les ressources annoncées. Une allocation ne sera alors effectuée que si elle maintient le système dans un état sain.

Définitions. Une suite S de processus est **fiable** si elle vérifie la relation :

$$C_i - A_i(t) \leqslant R(t) + \sum_{S(j) < S(i)} A_j(t) \qquad \forall p_i \in S \tag{12}$$

L'état du système est **fiable** s'il existe une suite fiable complète pour cet état.

La relation (11) implique que la condition (12) est plus forte que la condition (4) :

Théorème 4. Toute suite fiable S est une suite saine du système.

De même que les relations (1), (2), (3), (4) entraînent le théorème 2, les relations (1), (9), (10), (12) entraînent le

Théorème 5. Si l'état du système est fiable, toute suite fiable peut être prolongée en une suite fiable complète.

Le théorème suivant s'applique aux changements d'état :

Théorème 6. Une allocation de ressources à un processus p_k fait passer le système d'un état fiable à un nouvel état fiable s'il existe dans ce nouvel état une suite fiable contenant p_k.

Soit E_0 l'état du système à l'instant t_0 et E_1 l'état du système à l'instant t_1 obtenu par allocation de ressources au processus p_k. L'allocation de ressources à p_k se traduit par les relations :

$$A_l(t_0) = A_l(t_1) \qquad \forall l \neq k \tag{13}$$

$$A_k(t_1) > A_k(t_0) \tag{14}$$

$$R(t_1) + A_k(t_1) = R(t_0) + A_k(t_0) \tag{15}$$

Soit S une suite fiable à l'instant t_1 et contenant p_k

$$S = S_1, \{p_k\}$$

$$C_i - A_i(t_1) \leqslant R(t_1) + \sum_{S(l)<S(i)} A_l(t_1) \qquad \forall p_i \in S_1$$

$$C_k - A_k(t_1) \leqslant R(t_1) + \sum_{S(l)<S(i)} A_l(t_1)$$

Montrons qu'elle est aussi fiable à l'instant t_0. D'après les relations (13), (14), (15)

$$C_i - A_i(t_0) \leqslant R(t_0) + \sum_{S(l)<S(i)} A_l(t_0) \qquad \forall p_i \in S_1$$

La relation (15) entraîne :

$$C_k - A_k(t_1) - R(t_1) = C_k - A_k(t_0) - R(t_0)$$

d'où

$$C_k - A_k(t_0) \leqslant R(t_0) + \sum_{S(l)<S(k)} A_l(t_0)$$

La suite S est donc fiable à l'instant t_0 ; elle peut être prolongée en une suite fiable complète S_2 :

$$C_i - A_i(t_0) \leqslant R(t_0) + \sum_{S_2(l)<S_2(i)} A_l(t_0) \qquad \forall p_i \in S_2$$

Montrons que S_2 est aussi fiable à l'instant t_1. Posons $S_2 = S, S_3$.

$$C_i - A_i(t_0) \leqslant R(t_0) + \sum_{S_2(l)<S_2(i)} A_l(t_0) \qquad \forall p_i \in S_3$$

Les relations (13) et (15) entraînent :

$$C_i - A_i(t_1) \leqslant R(t_0) + A_k(t_0) + \sum_{\substack{S_2(l)<S_2(i) \\ \text{et } l \neq k}} A_l(t_1)$$

$$\leqslant R(t_1) + \sum_{S_2(l)<S_2(i)} A_l(t_1) \qquad \forall p_i \in S_3$$

La suite S étant fiable à l'instant t_1, la suite S_2 est une suite fiable complète à l'instant t_1.

Théorème 7. Si aucun processus ne libère ses ressources avant d'avoir reçu la totalité de son annonce, il existe au moins une manière d'allouer les ressources sans interblocage si et seulement si l'état du système reste fiable.

Remarquons que dans le cas où $C = D$, les concepts de suite fiable et de suite saine, d'état fiable et d'état sain sont équivalents.

Considérons une suite fiable complète. Si les processus exigent la totalité de leur annonce $(C = D)$, les conditions du théorème 1 sont vérifiées. Il existe donc une suite d'états fiables tels que les processus obtiennent leurs ressources. La réciproque est également vraie.

Corollaire. S'il est possible de trouver une façon d'allouer les ressources dans le cas le plus défavorable (chaque processus demandant toute son annonce), il est possible de trouver un ordre d'allocation dans tous les cas.

Si les processus ne s'interbloquent pas, c'est qu'il existe une suite fiable complète S telle que :

$$C_i \leqslant R(t) + \sum_{S(j)<S(i)} A_j(t) \qquad \forall i, t \in [1, n] .$$

Comme $D(t) \leqslant C$ pour tout t, S est une suite saine pour le système à tout instant.

Le théorème précédent et son corollaire signifient que tant que l'état du système reste fiable il existe au moins une façon d'allouer les ressources aux processus sans interblocage : elle consiste à satisfaire leur demande dans l'ordre de la suite fiable.

Il n'est pas toujours nécessaire de respecter cet ordre d'allocation. Il faut simplement s'assurer que toute allocation laisse le système dans un état fiable. Cela se traduit par la règle suivante :

Règle. Une allocation ne peut avoir lieu que si l'état résultant est un état fiable.

Dans la pratique, l'état initial du système est toujours fiable. On doit vérifier, avant toute allocation de ressources, que le nouvel état obtenu est fiable. On utilise pour cette vérification un algorithme analogue à celui décrit en 4.7411, en remplaçant la matrice des demandes par la matrice des annonces.

Il est important de noter qu'une suite fiable ne représente pas toujours l'ordre dans lequel les demandes doivent être satisfaites. L'état est dit fiable en ce sens qu'il existe au moins une façon d'allouer les ressources aux processus sans risque d'interblocage. L'allocation effective peut se faire de toute autre façon à la seule condition de conserver le système dans un état fiable. L'ordre dans lequel les requêtes des processus sont satisfaites dépend des stratégies individuelles d'allocation de chaque ressource. En tout état de cause, la prévention de l'interblocage est une stratégie globale qui doit prévaloir sur les stratégies individuelles.

Remarque.

— Des créations ou des libérations de ressources maintiennent le système dans un état fiable.

— Une diminution de l'annonce conserve la fiabilité d'un état ; toutefois une augmentation ne peut être autorisée que si elle maintient le système dans un état fiable.

— Lors de la création d'un nouveau processus, la suite obtenue en mettant ce processus en queue de la suite fiable de l'état qui précède la création est une suite fiable, quelle que soit l'annonce du nouveau processus. La création d'un processus est donc toujours possible.

— La méthode de l'annonce peut conduire à un écroulement des performances dès que le nombre de ressources libres décroît fortement, dès que les annonces sont voisines du nombre total de ressources du système et dès que les requêtes des processus approchent de leur annonce.

4.75 CONCLUSIONS

Selon quel critère doit-on choisir entre prévention de l'interblocage et détection, puis guérison ? Lorsque l'utilisation de la méthode des classes ordonnées entraîne une immobilisation des ressources tolérable, cette méthode semble devoir être retenue. Sinon, selon la fréquence probable des interblocages, on pourra retenir :

— une méthode de détection-guérison brutale (par exemple un simple rechargement du système dans le cas où l'interblocage survient rarement),

— la méthode de l'annonce dans le cas où l'interblocage risque d'être fréquent ou difficile à dénouer.

EXERCICES

1. [3] *Un modèle mathématique de comportement local* [Spirn, 72]

On se propose de construire un modèle simple de comportement de programme rendant compte de la propriété de localité (cf. 4.23). Etant donné un programme de n pages numérotées de 1 à n, on cherche une loi de génération d'une chaîne de références $r_1\ r_2 \ldots r_k$. On définit une variable de temps discrète par les instants des références successives, en disant que la référence r_k a lieu au temps k.

1) On demande de construire un modèle ayant la propriété suivante : quel que soit m $(0 \leqslant m \leqslant n)$, à tout instant t, la probabilité pour que la prochaine page référencée r_{t+1} appartienne à l'ensemble des m pages distinctes ayant fait l'objet des références les plus récentes est égale à une constante p_m indépendante de t. Préciser le mécanisme de génération des r_k.

2) Dans quelles conditions les chaînes de références engendrées par ce modèle possèdent-elles la propriété de localité ?

3) Particulariser le modèle pour obtenir le comportement suivant : à tout instant, la probabilité pour que la référence suivante appartienne à l'ensemble des l pages le

plus récemment référencées est égale à une constante λ. Quelle relation doit-il y avoir entre λ et l pour que le modèle représente effectivement un comportement local ? Quelle est la relation entre λ, l et les p_m du 1) ? Comment définirait-on les L_i (cf. 4.23) et quelle est la durée de vie moyenne d'un L_i ?

2. [1] *Règle des 50 %* [Knuth, 68]

Montrer que dans une mémoire gérée par zones, lorsque le nombre N de zones allouées est grand et que le système est en équilibre (c'est-à-dire que le nombre moyen de zones libres est constant), le nombre de zones libres M est approximativement égal à $N/2$.

3. [1] *Programmation des algorithmes de gestion de la mémoire par zones*

Programmer, dans un langage de votre choix, les algorithmes de gestion de la mémoire par zones correspondant à « la plus petite zone possible » et à « la première zone possible » (cf. 4.442).

4. [2] *Efficacité de la gestion des tambours*

Démontrer les formules exprimant l'efficacité de la gestion d'un tambour (cf. 4.532) :

a) avec file unique

$$E_f(L) = 2/(s + 1)$$

b) avec une file par secteur (tambour de pagination)

$$E_i(L) \simeq (L + 1)/(s + L + 1)$$

où s est le nombre de secteurs du tambour et L le nombre moyen de demandes de transferts de pages en attente.

5. [2] *Anomalie de Belady*

Pour un programme référençant cinq pages distinctes, construire une chaîne de références ω telle que $C(4, \omega) > C(3, \omega)$ pour l'algorithme FIFO.

6. [2]

Montrer que l'algorithme LRU possède la propriété d'inclusion (4.454).

7. [1] *Implantation d'un algorithme de remplacement de type* LRU [Belady, 66]

L'implantation de l'algorithme LRU nécessite à chaque accès un réarrangement dynamique des pages qui, faute de mécanisme câblé adéquat, se révèle très coûteux sinon impossible. On peut alors avoir recours à la technique suivante, plus grossière mais aisément réalisable.

A chaque page physique est associé un bit d'utilisation, noté U, qui est mis à 1 à chaque accès à la page. On peut alors définir une partition de l'ensemble des pages :

— pages récemment référencées,
— pages non récemment référencées.

1) Lorsqu'un remplacement devient nécessaire, quelle page doit-on choisir ? A quel instant peut-on remettre à zéro les bits U ?

2) On dispose maintenant d'un bit supplémentaire, le bit d'écriture E, qui est mis à 1 à chaque opération d'écriture dans la page. Au moment du remplacement, quelle page est-il préférable de choisir ?

8. [2] *Exemple de stratégie globale d'allocation de mémoire et d'unité centrale* [Belady, 69].

Si l'on considère uniquement la partie convexe de la courbe $e(s)$, représentant en fonction de la taille s de mémoire principale allouée, le temps moyen séparant deux défauts de page consécutifs (cf. 4.23) on a la relation suivante :

$$e(s) < [e(s - \Delta s) + e(s + \Delta s)]/2 \qquad \forall \Delta s$$

où Δs représente un accroissement de la taille de mémoire principale allouée au programme (s et Δs sont exprimés en nombre de pages).

1) Interpréter cette relation ; comment est-il possible, en jouant sur le temps d'allocation d'unité centrale, d'améliorer le temps d'exécution du programme ?

2) Considérons le cas où plusieurs processus sont en compétition pour l'utilisation de la mémoire et de l'unité centrale. L'algorithme de remplacement global utilisé est l'algorithme FIFO. En tenant compte du résultat précédent, de quelle façon doit-on modifier l'algorithme de remplacement FIFO, pour améliorer les performances globales du système ?

9. [2] *Représentation graphique de l'interblocage* [Holt, 71]

Un système (au sens du 4.731 et avec les mêmes notations) peut recevoir l'interprétation graphique suivante :

L'état d'allocation des ressources aux processus est représenté par un graphe où :

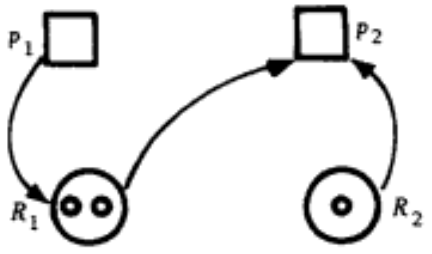
— les nœuds sont les éléments de l'ensemble des processus P et de l'ensemble des ressources E,

— la demande par un processus p_i d'une ressource de la classe R_j est représentée par un arc orienté (p_i, R_j),

— la possession, par un processus p_i d'une ressource de la classe R_j se traduit par un arc orienté (R_j, p_i).

La matrice A représente l'état d'allocation des ressources et la matrice $DN = D - A$ les requêtes insatisfaites.

Exemple



$$X = (2, 1) \qquad A = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \qquad D = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \qquad DN = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$$

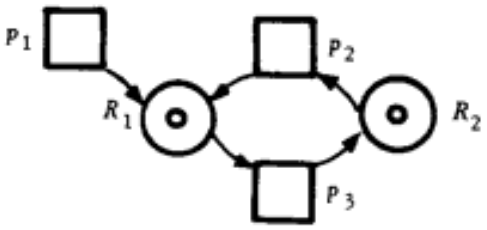
La demande d'une ressource par un processus donne lieu à la création d'un arc de requête qui change de sens lors de l'acquisition de la ressource et disparaît lors de la libération.

Nous dirons qu'un graphe d'état est réductible par tout nœud qui n'est ni un processus bloqué ni un nœud isolé ni une ressource. La réduction du graphe par un processus p est l'élimination de tous les arcs ayant p à une de leurs extrémités. Cette opération a le même effet sur le graphe que l'acquisition par ce processus de ses ressources, son exécution, puis la libération de toutes ses ressources. Le nouveau graphe obtenu est toujours un graphe d'état : il vérifie les conditions (1), (2) et (3) (cf. 4.731).

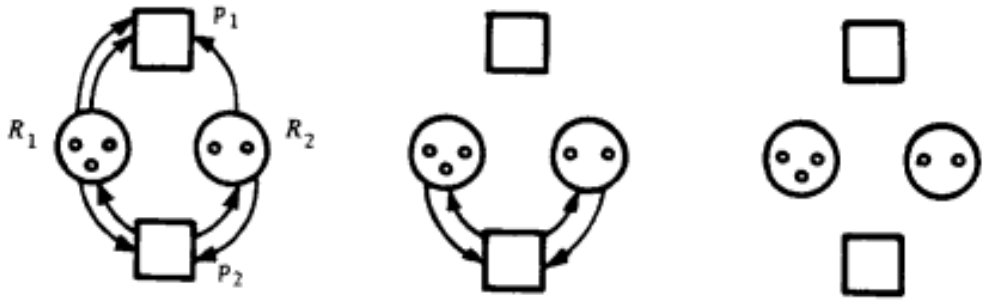
Nous appellerons **séquence** de réductions une suite de réductions du graphe par des processus tels que le graphe obtenu ne puisse plus être réduit par les autres processus du système ; le graphe est alors dit **irréductible.**

Si le graphe obtenu ne contient que des nœuds isolés, la séquence est **complète.** Le graphe initial est alors complètement réductible.

Exemple 1. Graphe irréductible.



Exemple 2. Réduction par la séquence (p_1, p_2).



Démontrer la **proposition 1 :** Si un graphe d'état G est réduit par deux suites S_1 et S_2 contenant les mêmes processus, alors les graphes H^1 et H^2 obtenus sont identiques.

10. [2]

Avec les notations de l'exercice [9], démontrer la **proposition 2 :**

Si un graphe d'état G réductible par un processus p_i peut aussi être réduit à un nouveau graphe H par une suite S de processus ne contenant pas p_i, alors H est encore réductible par p_i.

11. [2]

Avec les notations de l'exercice [9], démontrer la **proposition 3 :**

Si un graphe G peut être réduit au graphe H^1 par une séquence de réductions S_1 alors toute autre séquence de réductions S_2 est composée des mêmes processus et conduit à un graphe H^2 identique à H^1.

12. [2]

Avec les notations de l'exercice 9, démontrer la **proposition 4 :**

Une suite de réductions $S = \{ p_{q_1}, \ldots, p_{q_n} \}$ d'un graphe d'état G est une suite saine de processus et réciproquement.

13. [2]

Avec les notations de l'exercice 9 et en admettant la proposition 4, démontrer la **proposition 5 :**

La présence d'un circuit dans le graphe d'état est une condition nécessaire d'interblocage.

5

PROTECTION

5.1 PRÉSENTATION DU PROBLÈME

5.11 INTRODUCTION

Le contrôle du bon emploi de l'information et, d'une manière plus générale, des ressources d'un système est étudié sous le terme général de **protection** [Denning, 71 ; Wilkes, 68 ; Lampson, 71]. Comme il paraît naturel de faire jouer les dispositifs de contrôle au moment de l'accès à l'information ou de la demande de ressource, l'étude de la protection aurait pu être faite dans les chapitres qui en ont traité. Toutefois, l'analyse de la protection a montré qu'il existait une approche unique du problème. C'est elle que nous essaierons de dégager avant d'exposer quelques réalisations caractéristiques.

Nous écartons de notre présentation les problèmes politiques et sociaux que soulève la protection d'informations de caractère privé ou secret et nous nous penchons sur les seuls problèmes techniques.

La protection n'a pas comme rôle d'empêcher la production des erreurs, ou des malveillances, mais seulement leur incidence sur les objets protégés. Ainsi est assurée la protection des usagers entre eux, du système contre les usagers, des usagers contre le système et des processus d'un même usager entre eux. Aucun processus ne doit, par exemple, accéder à une donnée interne d'un autre processus sans en avoir reçu l'autorisation ; un processus qui utilise une procédure partagée, qu'elle soit du moniteur ou d'un autre usager, ne doit pouvoir l'exécuter sans contrôle ; un programme de mise au point doit être à l'abri des erreurs du programme qu'il essaie d'analyser.

Tout système de protection général satisfait les fonctions suivantes :

— Assurer l'indépendance des objets qui doivent rester logiquement indépendants. Cette indépendance peut être obtenue soit en ne donnant aucun

chemin vers l'information, soit en interdisant l'accès si le chemin existe. Dans le premier cas, l'information est inconnue : elle est en dehors de l'univers de l'utilisateur. C'est le cas par exemple en ALGOL pour les variables d'un bloc qui sont inconnues dans un autre bloc de même profondeur.

— Permettre une protection de l'utilisation de l'information, fonction de l'opération qui est tentée.

— Permettre une protection sélective de l'information partagée, en fonction de l'usager ou d'un groupe d'usagers.

Un bon principe pour lutter contre la propagation des erreurs est de limiter au strict nécessaire les pouvoirs d'un utilisateur.

Il n'existe pas, pour l'instant, de solution unique en matière de protection. Ceci explique la part importante que nous réservons, dans ce chapitre, à des réalisations particulières.

Dans une première partie, après avoir défini le problème, nous esquissons les propriétés d'un système de protection, et tentons de dégager à cette occasion quelques principes généraux de conception, valables à la fois pour des mécanismes câblés et programmés. A partir des concepts introduits, nous décrivons ensuite, dans le détail, les mécanismes de protection mis en jeu dans les deux systèmes ESOPE et MULTICS.

5.12 POSITION DU PROBLÈME

5.121 **Définitions**

Soit :

— un ensemble d'objets, $R = \{ r_i \}$: ce sont les éléments à protéger dont l'utilisation doit être contrôlée (par exemple fichiers, segments, pages, terminaux, processus, procédures, sémaphores),

— un ensemble d'utilisateurs, $U = \{ u_j \}$: ce sont les entités capables de produire ou de propager des erreurs (par exemple processus, client, procédure),

— une matrice M dont chaque ligne correspond à un utilisateur u_i et chaque colonne à un objet r_j.

Chaque élément M_{ij} de la matrice indique l'ensemble des modes d'accès de r_j par u_i. Le mode d'accès peut être donné ou calculé par une fonction, dépendante de l'utilisateur et de la ressource (exemples de modes d'accès : lecture ou écriture d'un fichier, création d'un processus, exécution d'un segment).

La matrice M définit les règles de protection à respecter. Nous supposons que chaque objet (et chaque utilisateur) est désigné par le même nom pendant toute la vie du système et que ce nom n'est pas réutilisé après destruction de l'objet.

Remarque 1. Une entité peut être à la fois objet et utilisateur : l'objet « processus d'impression » une fois activé, agit comme un utilisateur en demandant l'utilisation d'un tampon en mémoire.

Remarque 2. Certains objets doivent être demandés simultanément ; ainsi l'accès à un fichier nécessite un tampon en mémoire centrale.

La figure 1 est un exemple de matrice M. On note par λ l'interdiction d'emploi de l'objet par un utilisateur et ε l'absence de chemin de l'utilisateur vers l'objet.

U \ R	Fichiers f1	f2	f3	Consoles c1	c2	Processus p1	p2	p3
Processus 1	*lire écrire*	*lire*	*exécuter*	*allouer*	*allouer*	*bloquer détruire*	*activer*	*activer interrompre*
Processus 2	*propriétaire écrire*	λ	ε	*allouer*	ε	*activer*	*bloquer détruire*	*détruire*
Processus 3	*lire*	*écrire*	λ	λ	λ	*activer*	λ	*bloquer*
Processus 4	*lire*	ε	*écrire*	λ	*allouer*	λ	λ	λ

Figure 1. Exemple de matrice M.

On appelle :

— **droits** (« capabilities ») d'un utilisateur u_i sur un objet r_j, l'élément M_{ij},
— **pouvoir** d'un utilisateur u_i, la ligne M_i,
— **matrice des droits** (« access matrix »), la matrice M.

5.122 Limites du système de protection

Les droits traduisent les règles d'accès à un objet, propres à un utilisateur, et non les contraintes d'utilisation qui sont déterminées par les propriétés de l'objet, partageable, critique, ... (voir Chap. 4). Ces contraintes sont établies par ailleurs et ne sont pas considérées ici.

Les droits sont indépendants des actions entreprises par l'utilisateur en cas de refus. Le système de protection contrôle seulement si l'utilisateur a le droit ou non de se servir de l'objet avec l'accès qu'il précise.

La politique à adopter en cas d'erreur n'est pas non plus du ressort du système de protection. Par contre l'emploi des objets, lors de l'application de cette politique, doit aussi être contrôlé.

5.123 Variation du pouvoir d'un utilisateur : nécessité et limites

Le pouvoir, tel que nous l'avons introduit, définit pour chaque utilisateur l'ensemble de ses droits d'accès aux objets. Si le pouvoir associé à un processus reste constant durant tout le temps de son exécution, cela signifie que tous les programmes, toutes les procédures utilisés par ce processus ont les mêmes droits d'accès aux objets ; ces droits doivent donc être les droits les plus élevés qu'il utilisera bien qu'il n'en ait pas constamment besoin. La protection sera toutefois plus efficace si on peut lui donner à tout instant les pouvoirs minimaux dont il a besoin, et pas davantage.

Exemple 1. Lorsqu'un processus d'un utilisateur fait, en mode esclave, une opération d'entrée-sortie, il appelle généralement une procédure du système, dotée de droits plus étendus, qui vérifie les paramètres d'appel et provoque l'entrée-sortie proprement dite. Dès que l'entrée-sortie est terminée, le processus utilisateur retourne en mode esclave.

Le système de protection doit permettre la modification du pouvoir d'un processus d'utilisateur durant son exécution. Cette modification peut se faire dans le sens d'une augmentation (ou d'une diminution) de pouvoir et dans ce cas, les nouveaux droits d'accès aux objets sont un sur-ensemble (ou un sous-ensemble) des anciens droits d'accès. La modification du pouvoir peut aussi se traduire par l'acquisition de droits différents.

Exemple 2. Un processus a autorise un processus b à accéder à un segment de données d ; cet accès ne peut se faire qu'au moyen d'une procédure de contrôle p fournie par a et dotée de droits suffisants pour effectuer les références au segment de données d. Le mécanisme de protection doit aussi garantir que les données propres du processus b sont à l'abri des erreurs possibles de la procédure p. En conséquence, le pouvoir associé au processus b lorsqu'il utilise la procédure p, doit autoriser l'accès au segment d mais interdire tout accès aux données propres de b.

Il n'y a pas ici augmentation de pouvoir, mais acquisition d'un pouvoir différent par le processus b.

On ne peut pas modifier n'importe quand et n'importe comment le pouvoir d'un processus. De même, les droits d'un utilisateur sur un objet peuvent dépendre du chemin d'accès à l'objet ou de la façon dont l'objet est demandé.

Le système de protection peut parfois imposer que le chemin d'accès à un objet (ou un ensemble d'objets) débute par un point de passage obligé, appelé **guichet** (« gate »).

Exemple 3. L'utilisation d'une procédure p peut entraîner la demande d'autres objets a, b, ... Le système de protection remplit alors plusieurs fonctions :

— assurer que l'exécution de la procédure p ne peut commencer sans passer par un guichet,

— vérifier, au guichet, que l'utilisateur a le droit d'utiliser p,

— lui donner les moyens, en changeant son pouvoir, si c'est nécessaire pendant le temps qu'il utilise p, et pendant ce temps-là seulement, d'accéder aux objets a, b, ...

Pour simplifier le contrôle des valeurs prises par le pouvoir d'un utilisateur, on peut être amené à en limiter le nombre. On peut aussi introduire une relation d'ordre entre ces valeurs ; on verra une telle approche avec les anneaux de protection de MULTICS.

5.124 Problèmes à résoudre

D'autres aspects du problème, que nous n'avons pas évoqués, se posent toutefois au concepteur de système :

— comment utiliser la matrice des droits, ou comment effectuer le contrôle de l'accès à un objet ?

— comment conserver les informations de la matrice M ?

— comment régler la variation de la matrice M au cours du temps, et traduire la création de nouveaux objets ou la modification des droits d'un utilisateur ?

— comment protéger le système de protection ?

La matrice M traduit, en fait, les relations existant à un instant donné entre les utilisateurs et les objets du système. Le passage de la représentation des relations par une matrice M à la réalisation en machine dépend de la nature des objets mis en jeu. Plusieurs techniques, câblées ou programmées, peuvent être envisagées. C'est pourquoi, dans la pratique, le système de protection se trouve réparti à des niveaux différents. L'implantation d'un système de protection requiert toutefois l'existence d'un mécanisme câblé initial (protection de mémoire et/ou mode maître-esclave) pour assurer justement sa protection. On peut noter l'analogie qui existe ici avec le problème de l'exclusion mutuelle pour la solution duquel il est nécessaire de disposer d'un mécanisme de base (masquage des interruptions et instruction *TAS* dans le cas d'un multi-processeur).

Remarque. Il est toutefois possible de réaliser un système de protection, sur une machine qui ne dispose d'aucun mécanisme de protection câblé. Il suffit que le système interprète toutes les instructions des utilisateurs et qu'il contrôle l'emploi qu'ils font des objets du système.

5.13 EXEMPLES D'IMPLANTATION DE LA MATRICE DES DROITS

Nous exposons ci-après quelques techniques couramment utilisées pour implanter la matrice des droits. Le choix d'une technique dépend de la nature de la matrice M.

5.131 Liste d'accès

A chaque objet r est associée une liste des utilisateurs autorisés et de leurs droits, appelée **liste d'accès** (« access list »). Chaque élément de la liste est donc de la forme $(u, M(r, u))$. Cette liste est consultée à chaque tentative

d'accès à l'objet r. Cette méthode est couramment utilisée pour l'accès aux fichiers et n'est efficace que si la liste est courte. Diverses techniques d'accélération sont possibles :

— introduction de la liste complémentaire des utilisateurs non autorisés lorsqu'elle est plus courte que celle des utilisateurs autorisés,
— groupement des utilisateurs ayant les mêmes droits.

5.132 Liste des droits

La **liste des droits** est une représentation du pouvoir d'un utilisateur. A chaque utilisateur u est associée une liste (« capability list ») spécifiant, pour chaque objet auquel il peut accéder, les droits d'accès correspondants. Chaque élément se présente sous la forme $(r, M(r, u))$. Cette liste est parcourue à chaque tentative d'accès de l'utilisateur u. Elle peut être raccourcie avec les mêmes techniques que la liste d'accès. Un avantage de cette liste de droits est que, étant liée à l'utilisateur, elle n'est employée que lorsque celui-ci est actif. Ceci permet d'introduire des techniques d'accélération d'accès comme une mémoire associative, une antémémoire ou un mécanisme de couplage (voir Chap. 3 et 4). Si la liste est longue, elle peut être structurée en sous-listes, chacune étant associée à un mode d'utilisation. Par exemple,

— chaque sous-liste correspond à un mode d'accès : lecture, écriture, exécution, ...
— chaque sous-liste correspond à une classe d'objets : mémoire virtuelle, fichier, ...

5.133 Clés et verrous

Les utilisateurs sont regroupés et chaque groupe est identifié par un code ou **clé** remis à chacun de ses membres. De même, les objets réunis en groupement sont caractérisés par un **verrou** identique pour chaque élément d'un groupement. Il y a une clé et un verrou par mode d'accès possible. L'accès n'est permis que si la clé permet d'ouvrir le verrou. Dans ce cas, l'utilisateur est représenté par la clé et l'objet par le verrou. Le problème se simplifie beaucoup.

Exemple. Le système câblé de protection contre l'écriture dans un bloc de mémoire du calculateur CII 10070 met en jeu une technique de clés et verrous. L'écriture est permise si et seulement si :

$$(verrou = 0) \underline{ou} (clé = 0) \underline{ou} (clé = verrou)$$

Clé et *verrou* peuvent prendre 4 valeurs possibles. Ceci permet d'introduire 4 classes d'utilisateurs, et 4 zones de mémoire différemment protégées contre l'écriture.

5.134 Matrice des droits pour les modes maître et esclave

Les objets accédés sont ici les instructions de la machine. Le pouvoir d'un processus peut prendre deux valeurs, l'une où toutes les instructions sont autorisées et l'autre où certaines seulement le sont.

L'exemple suivant est issu du calculateur CII 10070. Le système de protection est câblé et intervient lors du décodage de l'instruction.

Objets = code opération

	LW	*STW*	...	*LPSD*	*HIO*	*SIO*	*CAL*
Mode maître	*exec*	*exec*		*exec*	*exec*	*exec*	*exec*
Mode esclave	*exec*	*exec*	...	λ	λ	λ	*exec*

exec : droit d'exécuter l'instruction

Figure 2. Modes maître et esclave du CII 10070.

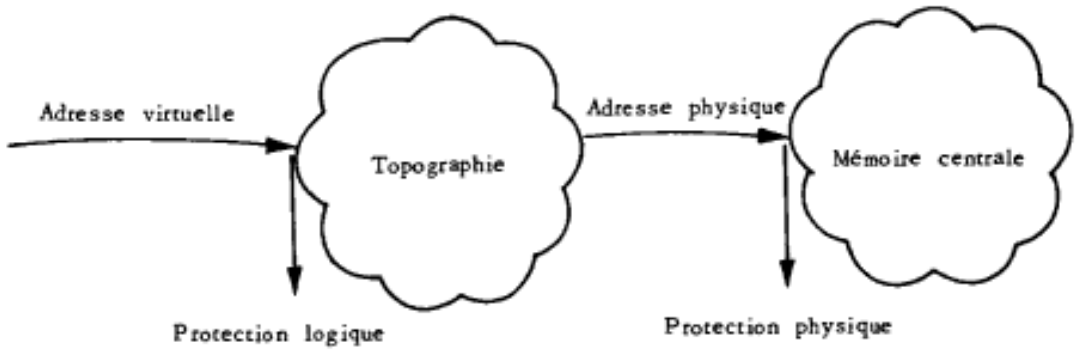
5.2 MÉCANISMES DE PROTECTION DANS LE SYSTÈME ESOPE

Il n'y a pas de mécanisme unique, mais plusieurs méthodes qui dépendent des objets à protéger et qui traduisent tout autant la diversité des mécanismes câblés que l'évolution des techniques de protection durant la réalisation du système.

5.21 RAPPELS SUR LE MATÉRIEL CII 10070

L'unité centrale peut fonctionner sous deux modes d'exécution, maître ou esclave (cf. 5.134) et utiliser deux modes d'adressage, avec ou sans topographie. Ces diverses options sont précisées dans le mot d'état de programme *(PSD)* de la machine. Seul un processus en mode maître peut exécuter des instructions d'entrée-sortie, agir sur les niveaux d'interruption, modifier le *PSD* et charger les registres de la mémoire topographique et les registres de protection de la mémoire.

Les restrictions d'accès à l'information peuvent être placées en deux endroits : en mémoire physique (lors de l'accès à l'information) ou en mémoire virtuelle (lors du calcul de l'adresse virtuelle).



A toute page physique est associé un verrou. Tout processus en exécution dispose d'une clé dans le *PSD*. Clés et verrous sont représentés sur deux bits et n'agissent que pour l'écriture (cf. 5.133). Toute tentative d'écriture non autorisée se traduit par un déroutement, quel que soit le mode d'exécution du processus fautif.

En mode « topographie », toute page virtuelle reçoit une protection logique de deux bits qui, pour tout processus, indique quel est l'accès permis dans cette page.

0 : tout accès permis,
1 : écriture interdite,
2 : écriture et exécution interdite,
3 : tout accès interdit.

Il en résulte que deux processus s'exécutant dans la même mémoire virtuelle possèdent les mêmes droits d'accès aux pages virtuelles. Cette protection n'existe pas en mode maître.

D'autre part les instructions *CAL* peuvent être employées pour définir des instructions ou des primitives nouvelles (ou pour appeler des procédures du moniteur). Elles permettent de réaliser des guichets et partant d'effectuer des branchements à des adresses prédéterminées avec modification de mode du *PSD*.

5.22 LA PROTECTION DANS LE SYSTÈME ESOPE

Le système ESOPE gère des **usagers** qui sont constitués chacun d'un ou de plusieurs processus et d'une mémoire virtuelle. Les programmeurs qui peuvent utiliser le système sont appelés **clients** du système et sont identifiés par leur nom ; lorsqu'ils utilisent le système, ils deviennent des usagers et reçoivent un numéro.

Les utilisateurs des objets sont les clients, les usagers ou les processus.

5.221 Utilisation des segments

Tout segment a un client propriétaire et un seul. Tous les segments d'un même propriétaire ont leur nom et leur adresse rangés dans un catalogue général. Pour chaque segment, le catalogue contient aussi :

— le mode d'accès que s'autorise le propriétaire,
— la liste des autres clients auxquels l'accès du segment a été autorisé et, pour chacun, le mode d'accès permis.

Seul le propriétaire d'un segment peut modifier l'entrée correspondante à son segment.

5.222 Protection de la mémoire virtuelle d'un usager

Chaque mémoire virtuelle forme un espace d'adressage indépendant (cf. 3.4). Les mémoires virtuelles ne communiquent que par les segments couplés à plusieurs d'entre elles. Dans ce cas le mode d'accès de chaque usager est précisé par les protections logiques de sa mémoire topographique.

A l'intérieur d'une mémoire virtuelle coexistent :

— une partie des données résidentes du moniteur,
— les procédures et les données de processus non résidents du moniteur,
— les procédures et les données des processus de l'usager.

On veut interdire :

— à tout processus, de modifier les procédures,
— aux processus de l'usager, d'accéder aux données du moniteur,
— aux processus du moniteur, d'accéder aux données des processus de l'usager.

Nous avons vu que deux processus parallèles du même usager ont les mêmes droits d'accès à une page virtuelle. Puisque chaque processus a sa propre *clé* d'écriture pour les pages physiques, on ne peut interdire que les processus lisent ou exécutent des pages qui leur sont interdites. Avec cette restriction, l'utilisation des *clés* et des *verrous* est faite comme suit :

	Données du moniteur (résidentes ou non) *verrou = 2*	Procédures du moniteur ou des usagers *verrou = 3*	Données des usagers *verrou = 1*
Processus usagers *clé = 1*	λ	λ	*écrit*
Processus du moniteur, résident ou non *clé = 2*	*écrit*	λ	λ

Figure 3. Utilisation des *clés* et *verrous* dans ESOPE.

5.223 **Pouvoir des processus**

A tout processus est associée une spécification de l'ensemble des actions qui lui sont permises. Les dispositifs câblés disponibles sont indiqués par plusieurs éléments du *PSD :*

— masques d'inhibition des déroutements et des interruptions,
— clé d'écriture en mémoire physique,
— indicateurs de mode : maître-esclave, avec ou sans topographie.

Ces dispositifs ont été complétés par un système programmé plus général, fondé sur une liste de droits, ou pouvoir, associé à chaque processus. A l'opposé des dispositifs câblés qui interviennent pour chaque instruction, les dispositifs programmés ne sont mis en jeu que pour les opérations *CAL* d'appel au superviseur.

Les processus sont divisés en 3 classes :

— les processus résidents du moniteur,
— les processus non résidents du moniteur,
— les processus des usagers.

A chaque classe est attachée une valeur de pouvoir à donner aux processus. Un processus peut recevoir un pouvoir différent de celui de sa classe lorsqu'il utilise, au moyen d'un *CAL*, des primitives ou des procédures du moniteur non résident. On donne (Fig. 4) un aperçu des divers pouvoirs.

Processus	Processus non résident du moniteur	Processus usager	Sémaphore moniteur	Sémaphore usager	Segment du système	Segment de l'usager
Moniteur résident	*créer détruire*	ε	*P, V*	ε	ε	ε
Moniteur non résident	λ	*créer détruire interrompre*	*P, V*	*créer détruire*	*ouvrir fermer coupler*	*fermer détruire*
Usager	λ	*créer détruire modifier*	ε (ou λ)	*P, V créer détruire*	λ	*créer coupler détruire*

Figure 4. Pouvoir des processus dans ESOPE.

Exemple. Considérons la matrice des droits précédente. Un processus de l'usager a le droit de *créer*, de *détruire* et de *modifier* les autres processus du même usager. Il n'a aucune action sur les processus non résidents du moniteur. Il peut agir sur les sémaphores du même usager, mais non sur ceux du moniteur bien que dans certains cas il connaisse leur nom (ce qui est noté λ). Il peut *créer, coupler, détruire* les segments de l'usager mais pas ceux du système (catalogues, segments des traducteurs, ...).

5.224 Changement du pouvoir d'un processus de l'usager

Le changement du pouvoir d'un processus d'un usager ne peut se produire que dans les cas suivants :

— lors de l'exécution d'une primitive,
— lors de l'appel d'une procédure du moniteur non résident et du retour correspondant.

a) Les primitives *(P, V, coupler, ...)* sont réalisées par des procédures qui ont accès uniquement aux données résidentes du moniteur et qui, pour cette raison, s'exécutent en adresses physiques. Elles sont appelées par une opération *CAL*. Cette opération assure simultanément plusieurs fonctions.

— Elle permet de réaliser un **guichet d'appel** : on vérifie que le processus a bien le droit d'utiliser la primitive et on fait commencer l'exécution de la procédure à une adresse qui est un point d'entrée de la procédure.
— Elle permet de changer le pouvoir du processus, pour lui permettre

d'accéder, par la procédure, aux données du moniteur (modification du *PSD* pour obtenir la clé d'écriture du moniteur).

— Elle permet de changer d'espace d'adressage (virtuel → physique).

Le retour de la primitive et partant la restauration du pouvoir du processus usager est effectué par la procédure proprement dite. Ce mécanisme est suffisant puisque la procédure dispose des droits nécessaires (mode maître) pour modifier le pouvoir *(PSD, ...)* du processus et effectuer le retour correspondant.

Etant donné que la primitive accède à des données communes, l'exclusion mutuelle nécessaire est réalisée ici par masquage des interruptions.

b) Les procédures du moniteur non résident sont couplées dans la mémoire virtuelle de chaque usager (cf. 3.441). Elles ont accès à des informations non résidentes du moniteur, par exemple le catalogue du système ou les catalogues des utilisateurs. Pour cette raison, l'exécution de telles procédures par un processus usager requiert des droits supérieurs aux siens. Ainsi un processus d'un usager ne peut se coupler aux articles du catalogue sans droits particuliers. Pour simplifier la présentation, nous supposons ici qu'il n'y a qu'un processus par usager.

L'appel d'une procédure du moniteur non résident est réalisé par une opération *CAL* qui a pour effet :

— de réaliser un guichet d'appel,

— de changer le pouvoir du processus usager (modification du *PSD* pour obtenir la clé d'écriture, autorisation de coupler le catalogue, autorisation d'agir sur les sémaphores du moniteur).

Ces actions nécessitent l'accès à des données résidentes du moniteur ; elles sont réalisées en adresses physiques, toutes interruptions masquées, après quoi le contrôle est passé en virtuel, au point d'entrée de la procédure.

La restauration du pouvoir du processus ne peut pas être assurée par la procédure qui, cette fois, n'en a pas le pouvoir. Elle utilise alors une opération *CAL* dont la fonction est :

— de réaliser un **guichet de retour** : ce guichet de retour — qui est différent du guichet d'appel — a dû être créé au moment de l'appel de la procédure ; il est associé au processus appelant. On vérifie alors que le processus a le droit d'utiliser ce guichet, en regardant s'il est créé ; dans le cas contraire une erreur est détectée.

— de restaurer le pouvoir du processus.

— de réaliser le retour proprement dit, dans la mémoire virtuelle du processus, en utilisant l'adresse de retour qui a été rangée dans le guichet de retour au moment de l'appel.

Pour les mêmes raisons que précédemment, ces actions sont effectuées en adresses physiques, toutes interruptions masquées.

5.3 MÉCANISME DE PROTECTION DANS LE SYSTÈME MULTICS

5.31 INTRODUCTION

MULTICS, comme CLICS, (cf. 3.2) utilise un mécanisme câblé de segmentation avec un dispositif de protection attaché à chaque segment. Le segment constitue la plus petite unité d'information à laquelle il est possible de donner une protection spécifique.

Pour appuyer la présentation qui va suivre, inspirée de [Schroeder, 72], nous introduisons les précisions suivantes.

— Pour chaque usager qui veut utiliser le système, on crée un processus et un espace adressable (ou descriptif) ; on associe à ce processus le nom de l'usager.

— L'information présente dans le système est constituée d'un ensemble de segments : une liste de contrôle d'accès (cf. 5.131) propre au segment, permet de connaître pour chaque usager les droits qu'il possède sur le segment. Cette liste détermine donc la colonne de la matrice M, relative à cet objet, ainsi que les évolutions possibles de cette colonne.

— Lorsqu'un processus fait référence à un segment, ce segment doit être introduit dans le descriptif du processus, s'il n'y était déjà ; cette opération qui est effectuée par une procédure du système, n'est effectivement réalisée que dans le cas où le nom de l'usager associé au processus demandeur figure dans la liste de contrôle d'accès du segment. Les droits de l'usager sont alors inscrits dans le descripteur du segment, attaché au processus.

L'évolution des droits d'un processus sur un segment est obtenue en associant au processus un pouvoir intrinsèque, indépendant des objets, et susceptible de varier suivant certaines conditions. Les valeurs possibles du pouvoir sont ordonnées ; elles sont appelées **anneaux** de protection (« rings »). Les droits d'accès d'un processus à un segment sont alors définis, pour chaque accès possible (lecture, écriture ou exécution), par la liste des anneaux dans lesquels doit se trouver le processus pour être autorisé à effectuer l'accès requis au segment.

Remarque. On peut, dans les calculateurs classiques, assimiler les modes maître et esclave à la présence de deux anneaux.

5.32 DÉFINITION ET PORTÉE DES ANNEAUX

Dans MULTICS le nombre d'anneaux a été limité à *8*. Les anneaux sont numérotés de *0* à *7*. Cette numérotation exprime en outre un ordre total sur les valeurs des pouvoirs possibles. Un processus possède les droits les plus étendus lorsqu'il s'exécute dans l'anneau *0*, et les droits les plus faibles lorsqu'il s'exécute dans l'anneau *7*.

La possibilité pour un processus d'accéder à un segment par exemple en lecture, en écriture ou en exécution, dépend de l'anneau dans lequel évolue le processus au moment où il effectue l'accès. Cet anneau est appelé par la suite **anneau courant.** L'ensemble des anneaux consécutifs, pour lesquels l'accès au segment est licite, forme la **parenthèse** de cet accès pour le segment. On définit pour chaque segment des parenthèses d'écriture, de lecture et d'exécution. Ces parenthèses sont prises dans la liste de contrôle d'accès du segment et recopiées dans le descripteur associé au processus, au moment où le segment est introduit dans le descriptif. Ces parenthèses sont propres à un couple segment-usager ; en d'autres termes, les parenthèses associées à un segment donné peuvent être différentes pour deux usagers autorisés à partager le segment. Une parenthèse peut être vide s'il n'existe aucun anneau où l'accès correspondant soit possible.

Schématiquement le contrôle de la référence à une adresse d'un segment est réalisé de la façon suivante en vérifiant :

— d'abord que la parenthèse du segment correspondante à l'accès demandé n'est pas vide,

— ensuite que l'anneau dans lequel évolue le processus, c'est-à-dire l'anneau courant, est bien inclus dans cette parenthèse d'accès.

La figure 5 illustre les parenthèses de protection associées à un segment de données modifiables mais non exécutables.

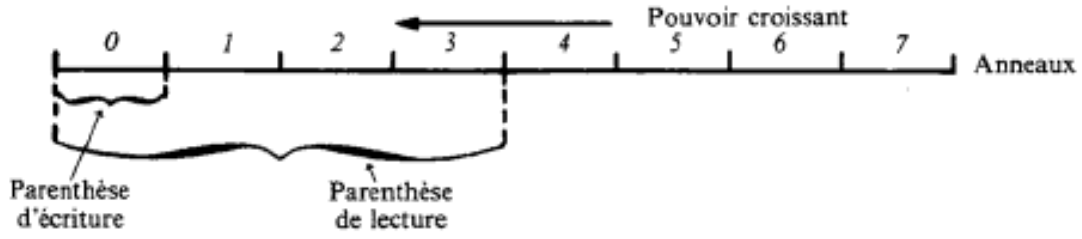


Figure 5. Parenthèses d'écriture et de lecture.

Exemple 1. Considérons le segment décrit dans la figure 5 pour un processus donné. Ce dernier pourra lire le segment s'il se trouve dans un anneau dont le numéro est compris entre *0* et *3*. Il pourra écrire dans le segment seulement s'il se trouve dans l'anneau *0* ; il ne pourra jamais l'exécuter.

La figure 6 illustre la parenthèse d'exécution associée à un segment-procédure.

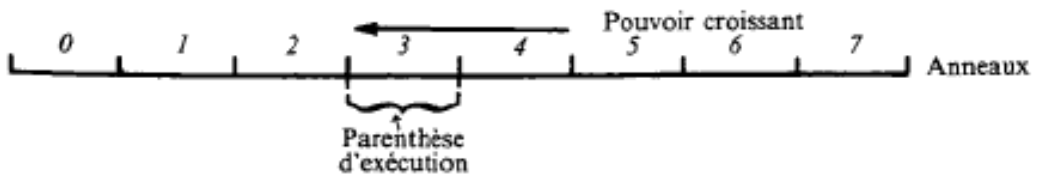


Figure 6. Parenthèse d'exécution.

Exemple 2. L'exécution, par le processus, de ce segment-procédure ne peut se faire que dans l'anneau *3*. Les conditions à satisfaire par un processus pour l'exécution d'une procédure seront étudiées au paragraphe suivant.

Remarque. Les parenthèses de lecture et d'écriture commencent toujours à l'anneau *0* ; par contre la parenthèse d'exécution peut commencer à un anneau quelconque. Ceci permet d'éviter l'exécution accidentelle d'un segment-procédure dans un anneau offrant davantage de pouvoir qu'il n'en faut normalement à la procédure pour référencer ses données. En pratique la plupart des segments-procédure ont une parenthèse d'exécution restreinte à un seul anneau.

5.33 CHANGEMENT DU POUVOIR D'UN PROCESSUS. NÉCESSITÉ DU GUICHET

Une instruction de branchement vers un segment-procédure peut être :

— soit un appel de procédure (instruction *CALL*),
— soit un retour de procédure (instruction *RETURN*),
— soit un branchement inconditionnel (instruction *GOTO*).

L'exécution par un processus d'une instruction *CALL* ou *RETURN* peut donner lieu au changement de son anneau d'exécution et partant à une modification de son pouvoir. L'instruction *GOTO*, par contre, ne peut donner lieu à aucun changement de pouvoir ; elle permet de se brancher vers des segments-procédure seulement dans le cas où leur parenthèse d'exécution contient l'anneau courant du processus.

5.331 Augmentation de pouvoir

Lorsqu'un processus, en exécution dans l'anneau n, appelle un segment-procédure dont la parenthèse d'exécution est $[m, u]$ avec $n > u$, le pouvoir du processus doit être augmenté ; cette augmentation implique un contrôle.

Pour cela on impose que l'exécution dans un nouvel anneau commence toujours à une (ou plusieurs) adresse(s) définie(s) du segment-procédure, appelée(s) guichet(s) ; ainsi seuls les branchements vers un guichet sont autorisés, et c'est seulement au guichet que l'anneau d'exécution du processus prend sa nouvelle valeur. On garantit ainsi que les programmes qui commencent à ces adresses référenceront leurs données avec les droits d'accès — ni trop faibles, ni trop forts — qui correspondent au nouvel anneau d'exécution. Les guichets sont spécifiés, en associant à chaque couple segment-usager une liste d'adresses de guichets ; ces guichets sont un sous-ensemble des points d'entrée externes (cf. 3.262).

De plus, afin de contrôler plus strictement toute augmentation de pouvoir, on définit pour tout segment exécutable et pour chaque usager, une **parenthèse d'appel.** La parenthèse d'appel d'un segment-procédure spécifie l'ensemble des anneaux de numéros consécutifs, immédiatement supérieurs à la paren-

thèse d'exécution du segment et à partir desquels tout branchement vers un guichet du segment est autorisé. On garantit ainsi que des segments qui sont exécutables dans un anneau doté de droits importants, ne pourront être utilisés par le processus s'il évolue dans un anneau doté d'un pouvoir trop faible. L'exemple suivant illustre l'appel d'un segment procédure, avec augmentation de pouvoir.

Exemple 3. La protection attachée au segment est donnée par la figure 7 :

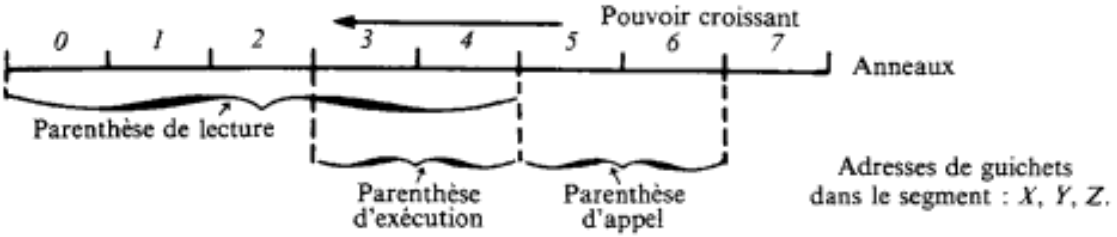


Figure 7. Exemple de protection d'un segment-procédure.

Toute tentative d'exécution de ce segment par le processus évoluant dans l'anneau *7* sera refusée. Si le processus évolue dans l'anneau *5* (ou *6*), il peut exécuter le segment, à la condition d'en commencer l'exécution par un des 3 guichets, dont les adresses respectives à l'intérieur du segment sont *X*, *Y* et *Z*. Dans ce cas le nouvel anneau d'exécution du processus est fixé à la valeur de la borne supérieure de la parenthèse d'exécution (ici *4*) du segment. Etant donné que le segment procédure a été construit pour travailler indifféremment dans les anneaux *3* et *4*, le choix de cette valeur peut paraître arbitraire ; il n'en demeure pas moins conforme à la philosophie des protections qui veut que l'on donne à un processus le pouvoir juste nécessaire à son exécution, mais pas davantage.

5.332 **Conservation du pouvoir**

Il y a conservation du pouvoir d'un processus lorsque celui-ci exécute une instruction de branchement *(CALL, RETURN* ou *GOTO)* vers un segment-procédure dont la parenthèse d'exécution inclut l'anneau d'exécution du processus. L'opération *CALL* donne cependant lieu à un contrôle par guichet de l'adresse de branchement. Ceci permet de prévenir des appels de procédure à des adresses qui ne seraient pas des points d'entrée. L'instruction *GOTO*, par contre, permet d'ignorer les guichets.

L'utilité de la conservation du pouvoir se pose pour l'utilisation de certaines procédures partagées, couramment employées (ce sont par exemple des sous-programmes de bibliothèque) qu'il est nécessaire d'exécuter avec autant de pouvoir que la procédure appelante, mais pas davantage. Afin d'éviter que tout appel d'une procédure de ce type ne provoque un changement de l'anneau d'exécution du processus, la parenthèse d'exécution qui leur est associée englobe plusieurs anneaux de numéros consécutifs. Ainsi l'exécution de ces procédures continuera à se faire dans l'anneau d'exécution du processus,

si celui-ci est inclus dans la parenthèse d'exécution de la procédure. Dans l'exemple 3, l'exécution du segment-procédure se fera indifféremment dans l'anneau *3* ou *4* suivant que le processus évolue au moment de l'appel dans l'un des anneaux *3* ou *4*.

5.333 Diminution de pouvoir

Lorsqu'un segment-procédure doté de la parenthèse d'exécution $[m, u]$ est appelé par un processus en exécution dans l'anneau n (avec $n < m$), cet appel s'accompagne d'une diminution du pouvoir du processus. Le changement de l'anneau d'exécution d'un processus dans le sens d'une diminution de pouvoir ne devrait pas nécessiter de contrôle, en lui-même. Cependant deux problèmes se posent :

— un problème de programmation : la procédure appelante peut très bien spécifier des paramètres que la procédure appelée n'a pas le pouvoir de référencer, dans son anneau d'exécution ;

— un problème de réalisation : le retour (par une instruction *RETURN*) vers la procédure appelante, se fera dans le sens d'une augmentation de pouvoir ; en conséquence c'est une opération qui doit être contrôlée avec soin.

Le premier problème peut être résolu en interdisant à la procédure appelante de spécifier des paramètres qui ne seraient pas accessibles à la procédure appelée. Une deuxième solution consiste à copier les paramètres passés dans des segments accessibles cette fois à la procédure appelée, puis à les recopier dans leur emplacement initial une fois le retour vers la procédure appelante effectué. Cette solution présente toutefois un inconvénient : elle ne permet plus le partage des paramètres entre différents processus.

Le deuxième problème peut être résolu en imposant que le retour vers la procédure appelante passe par un guichet. Ce guichet de retour — qui est différent du guichet d'appel — est associé à la procédure appelée ; il doit être créé au moment de l'appel et détruit lors du retour vers la procédure appelante. Dans le cas où la procédure appelée est une procédure récursive, les guichets de retour doivent être gérés dans une pile.

Remarque. Dans la pratique, les appels de procédures avec diminution de pouvoir ont une application limitée pour la raison essentielle suivante. Considérons une procédure p_r en exécution dans l'anneau r, qui appelle une procédure p_s, ayant pour domaine d'exécution l'anneau s (avec $r < s$) ; les paramètres rendus par la procédure p_s, déterminent l'évolution ultérieure de la procédure p_r. Dans la mesure où la procédure appelante accorde une grande confiance aux résultats obtenus (ils sont peut-être erronés) elle peut compromettre gravement la sécurité des données et des procédures également accessibles dans l'anneau r. Toutefois, les appels de procédures avec diminution de pouvoir sont utilisés sans danger lorsqu'il n'y a pas transmis-

sion de paramètres ; cette technique est employée dans MULTICS pour commencer l'exécution du programme d'un utilisateur à partir d'une procédure du moniteur.

5.34 IMPLANTATION CÂBLÉE DES ANNEAUX DE PROTECTION

Un projet de réalisation de mécanismes de protection câblés, bâtis sur la notion d'anneaux, est présenté dans [Schroeder, 72]. La description que nous en donnons présente l'organisation du processeur sous les seuls aspects qui ont trait au contrôle de l'accès à une adresse.

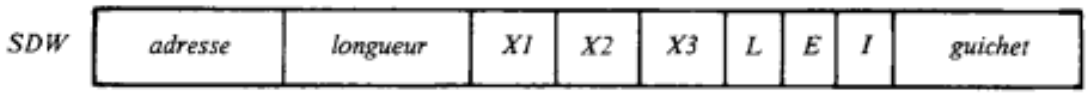
5.341 **Descripteur de segment**

L'ensemble des segments appartenant à la mémoire virtuelle d'un processus est défini par le descriptif des segments (cf. 3.222). Chaque descripteur (noté *SDW*) décrit un segment, et un seul, de la mémoire virtuelle du processus. Le numéro de segment est utilisé comme index pour accéder, dans le descriptif, au descripteur considéré. Chaque descripteur *SDW* contient l'adresse absolue du segment en mémoire, sa longueur et des indicateurs utilisés par le mécanisme de protection.

Dans l'implantation proposée, les définitions des parenthèses d'accès et des listes de guichets ont été restreintes afin :

— d'une part, de diminuer l'encombrement du descripteur,

— d'autre part, de simplifier les tests de validité d'accès effectués par le processeur.



Les 3 numéros d'anneaux, contenus dans les champs *(SDW. X1, SDW. X2, SDW. X3)* délimitent les parenthèses de lecture, d'écriture et d'exécution, de même que la parenthèse d'appel, de la façon suivante :

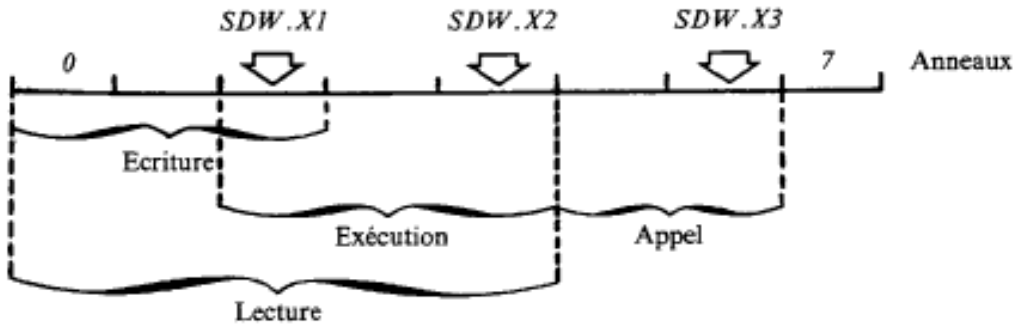


Figure 8. Disposition des parenthèses de protection.

Des témoins de lecture, d'écriture et d'exécution, respectivement donnés par les champs *SDW.L, SDW.E, SDW.I,* permettent en outre de spécifier si une parenthèse est vide ou non.

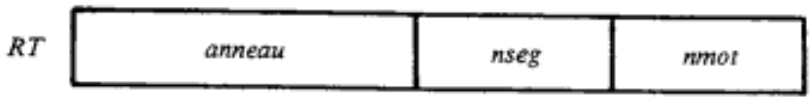
Pour simplifier la définition des listes de guichets, les possibilités ont été restreintes : par convention, seules les premières adresses d'un segment-procédure sont des adresses de guichets. La liste de k guichets, associée au couple segment-usager est donc déterminée par les k premières adresses du segment : c'est ce nombre k qui constitue le champ *SDW.guichet*. Il en résulte que deux usagers différents ne peuvent avoir des listes de guichets disjointes ; le premier (ou les premiers) guichet(s) d'une liste est (sont) nécessairement commun(s) à tous les usagers autorisés à partager le segment ; les guichets dont l'emploi est restreint à un petit nombre d'usagers (ou à un seul) sont situés en fin de liste.

On rappelle que les champs du *SDW* sont localisés par le nom de l'usager dans la liste de contrôle d'accès attachée à chaque segment. En outre un programme d'un usager, en exécution dans l'anneau n, ne peut pas spécifier, dans la liste de contrôle d'accès, des valeurs de *SDW.X1, SDW.X2, SDW.X3* inférieures à n.

5.342 Exécution d'une instruction

Nous avons vu que le compteur ordinal *CO* (c'est le registre *R0* dans CLICS) contenait l'adresse segmentée *(nseg, nmot)* de la prochaine instruction à exécuter. Il contient aussi le numéro d'anneau dans lequel s'exécute le processus.

Les tests de validité d'accès se font par l'intermédiaire d'un registre de travail *RT*, inaccessible au programmeur, composé de trois champs :



Nous verrons, par la suite, que tout nom est constitué de ces trois champs. Les registres pointeurs *RP*, accessibles au programme, sont les seuls registres, outre ceux déjà cités, à contenir des noms. Ils ne peuvent être chargés qu'au moyen d'instructions spéciales qui ont pour opérande un couple *(nseg, nmot)* ; l'anneau attaché à ce couple sera déterminé automatiquement au cours du calcul de l'adresse.

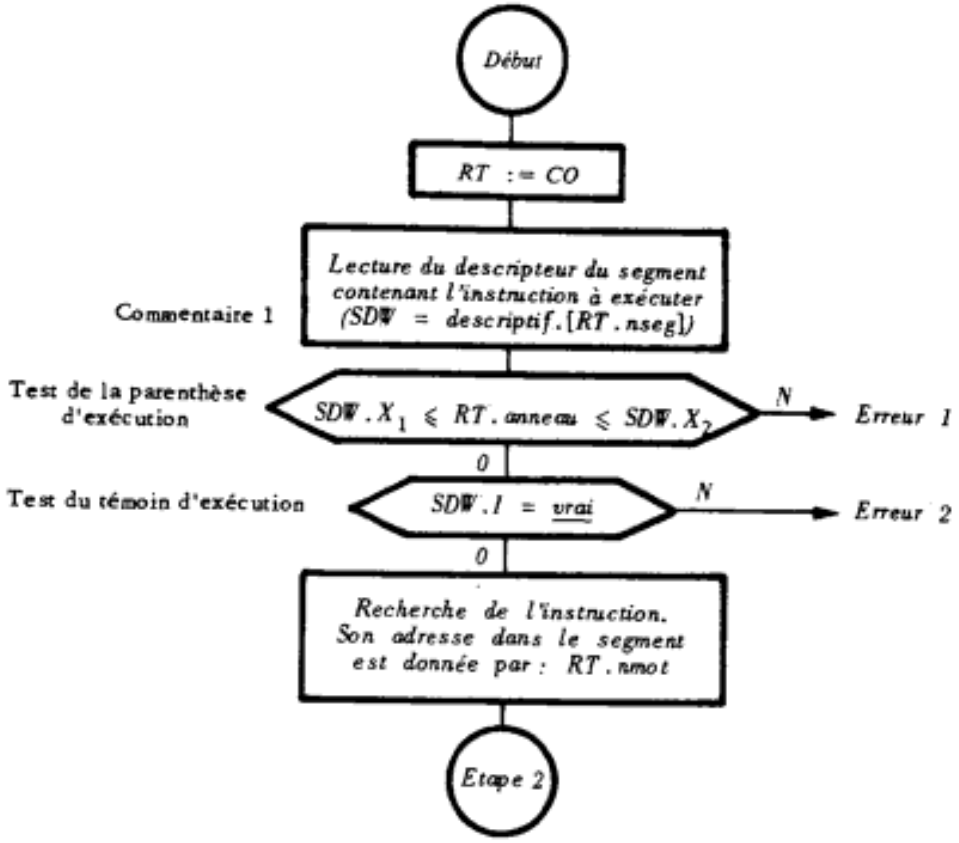
L'exécution d'une instruction est réalisée en 3 étapes :

— recherche de l'instruction à exécuter,
— formation de l'adresse segmentée de l'opérande,
— interprétation du code opération, avec accès à l'opérande.

Pour chaque étape, nous illustrerons par des organigrammes le contrôle exercé par le processeur lors de l'accès à une adresse.

a) *Recherche de l'instruction à exécuter*

L'adresse segmentée de l'instruction à exécuter est contenue initialement dans le registre *CO*. Le numéro d'anneau d'exécution courant *(CO.anneau)* qui définit le pouvoir actuel du processus est également contenu dans ce registre.



Erreur 1 : violation d'accès : l'anneau d'exécution courant n'est pas inclus dans la parenthèse d'exécution du segment référencé.
Erreur 2 : violation d'accès : exécution interdite.

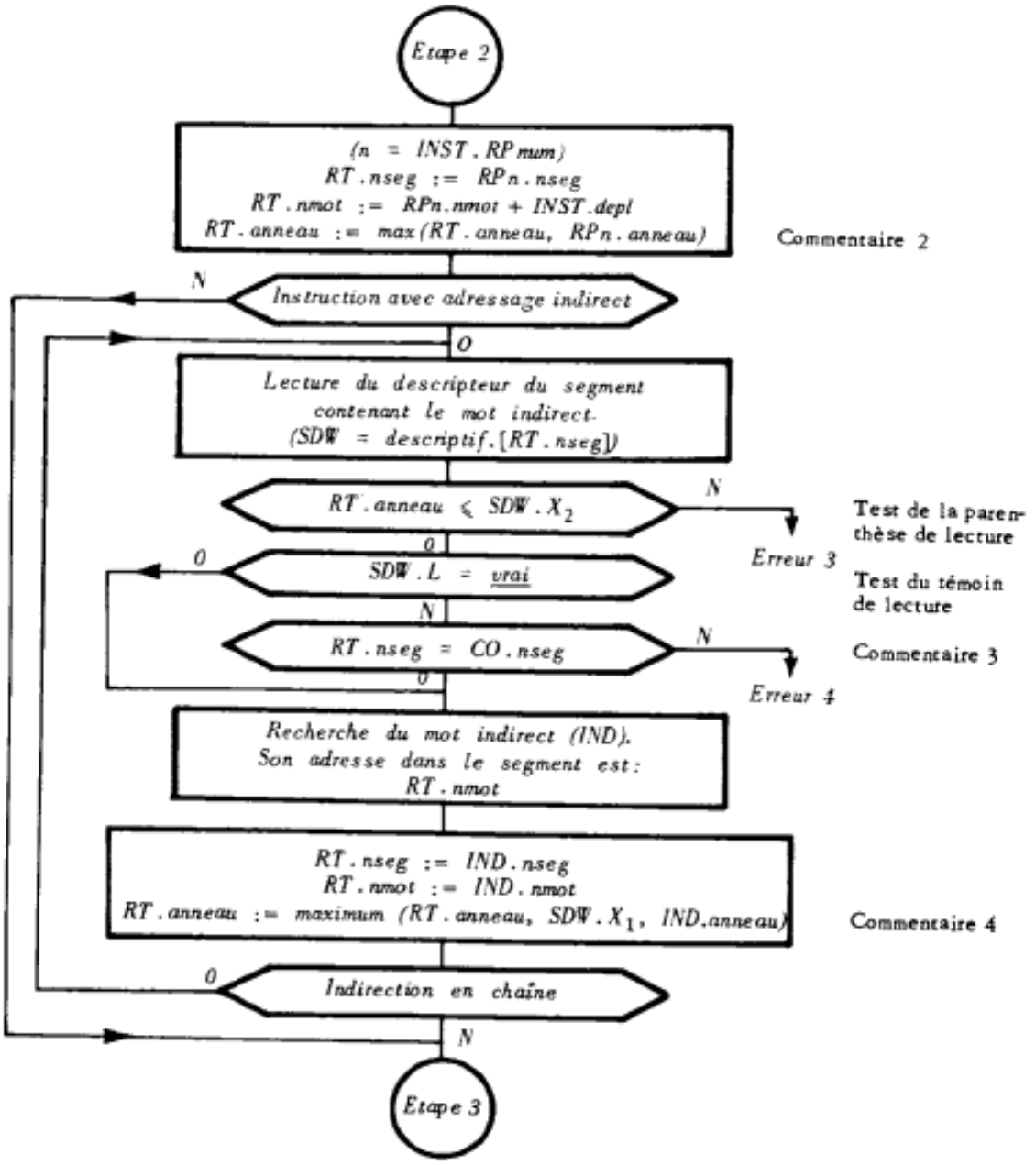
Commentaire 1 : le numéro du segment dans le descriptif du processus est donné par le champ *RT.nseg*.

Figure 9. Recherche de l'instruction.

b) *Formation de l'adresse virtuelle effective de l'opérande*

Lorsque l'instruction référence un opérande, son adresse segmentée est obtenue (cf. 3.24) en ajoutant le déplacement *(INST. dépl)*, spécifié dans l'instruction *INST*, au contenu du registre pointeur *RPn* spécifié dans l'instruction par le champ *INST.RPnum*. Ce registre contient également un numéro d'anneau qui permet de contrôler si l'accès demandé est licite ou non.

L'opérande définitif peut éventuellement être atteint à travers une suite d'indirections ; un contrôle est effectué par le processeur à chaque niveau d'indirection. Les adresses successives des mots indirects sont élaborées dans le registre de travail *RT*. L'accès aux différentes adresses qui mènent, via la chaîne d'indirection, à l'opérande définitif peut se faire avec un pouvoir différent mais toujours décroissant. Ce pouvoir est également rangé dans le registre *RT*.



Erreur 3 : violation d'accès : l'anneau effectif n'est pas inclus dans la parenthèse de lecture du segment contenant le mot indirect.
Erreur 4 : violation d'accès : lecture interdite par le témoin.

Commentaire 2 : le pouvoir avec lequel se fera l'accès à l'opérande ou au premier mot indirect sera inférieur ou égal au pouvoir défini par l'anneau d'exécution courant.

Commentaire 3 : le schéma suppose ici qu'il est possible de lire un mot dans le même segment, bien que le témoin de lecture ne soit pas positionné.

Commentaire 4 : chaque fois qu'un mot indirect est obtenu, on met à jour le pouvoir (*RT.anneau*) qui permettra de valider le prochain accès. Pour fixer sa nouvelle valeur, on considère que le mot indirect *(IND)* obtenu, a pu être modifié par une autre procédure du même processus influençant par là le résultat du calcul de l'adresse effective de l'opérande. Tenir compte de la parenthèse d'écriture *(SDW.X1)* de ce segment dans la mise à jour de *RT.anneau*, garantit que la référence à l'opérande sera contrôlée avec le pouvoir le plus faible qui pourrait avoir influé sur l'adresse effective.

Figure 10. Formation de l'adresse de l'opérande.

c) *Interprétation du code opération avec accès à l'opérande*

Au début de la présente étape (Fig. 11), on dispose dans le registre de travail *RT*

— de l'adresse segmentée de l'opérande,
— du numéro d'anneau, définissant l'accès requis à cet opérande.

Nous traitons l'exécution des instructions de branchement *(GOTO)* et des instructions d'écriture, en laissant de côté les instructions de lecture qui n'apportent rien de nouveau à la compréhension des mécanismes. Les instructions de branchement ne donnent lieu à aucun changement de pouvoir, mais seulement à un contrôle de validité d'exécution.

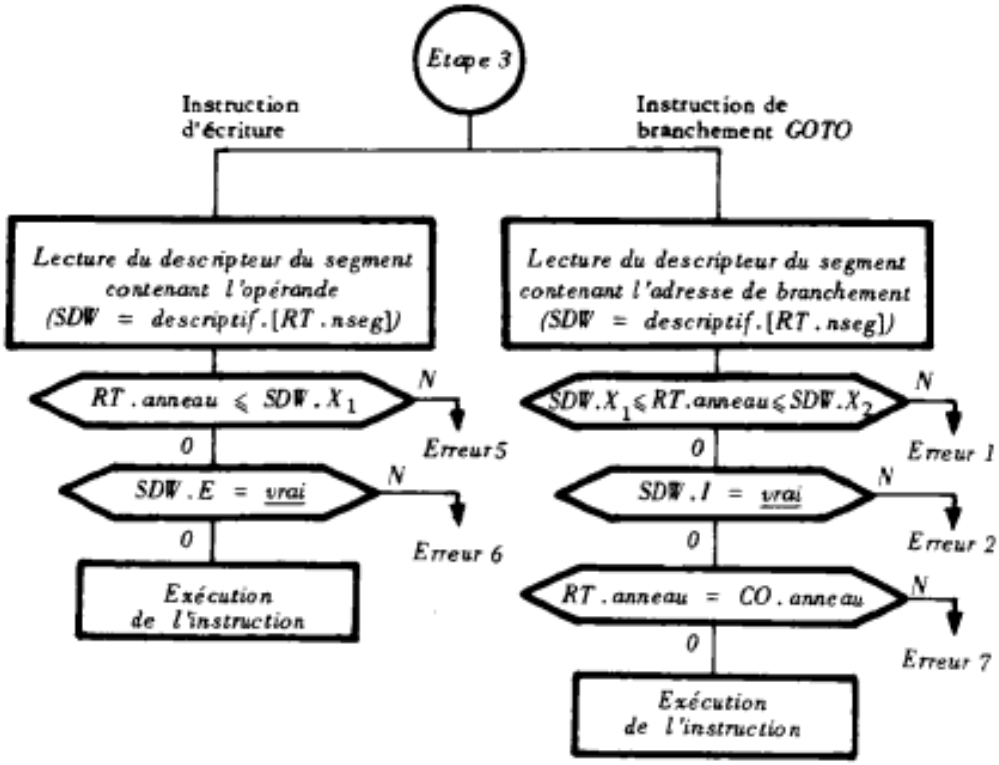
Remarque. Si l'instruction à interpréter est une instruction de chargement d'un registre pointeur *RP*, alors le contenu du registre *RT* est recopié dans ce registre pointeur. De cette façon, on est assuré que le numéro d'anneau qui est recopié dans le registre *RP*, est dans tous les cas supérieur (ou égal) à l'anneau courant du processus.

5.343 Appel et retour de procédure

L'appel d'une procédure s'accompagne de la passation de paramètres et du rangement de l'adresse de retour.

Pour qu'une procédure puisse être partagée par plusieurs processus, un mécanisme doit permettre à cette procédure de ranger ses données à des emplacements différents, suivant le processus qui l'exécute. Dans MULTICS, à cet effet, une pile de travail est associée à chaque processus. Chaque procédure utilise alors, pour désigner ses données, un registre pointeur de pile définissant dans la pile du processus pour le compte duquel elle s'exécute, la zone de travail qui lui a été allouée (dans CLICS le registre pointeur de pile est le registre *R3*).

Du point de vue de la protection, la zone de travail d'une procédure ne doit être accessible qu'aux procédures s'exécutant dans le même anneau que la procédure considérée ou dans un anneau doté d'un pouvoir supérieur. Pour



Erreur 1 : violation d'accès : (parenthèse d'exécution).
Erreur 2 : violation d'accès : (témoin d'exécution).
Erreur 5 : violation d'accès : (parenthèse d'écriture).
Erreur 6 : violation d'accès : (témoin d'écriture).
Erreur 7 : tentative de changement d'anneau d'exécution autrement que par les instructions *CALL* et *RETURN*.

Figure 11. Interprétation du code opération.

cette raison, l'implantation de la pile de travail associée à un processus est obtenue en créant autant de segments-pile qu'il y a d'anneaux d'exécution possibles (c'est-à-dire 8 segments). De même pour chaque procédure, il existe autant de segments de liaison qu'il existe d'anneaux d'exécution possibles (cf. 3.242).

Exemple. La zone de travail des procédures en exécution dans l'anneau n est implantée dans un segment-pile doté des parenthèses de lecture et d'écriture $[0, n]$. Ainsi les zones de travail de ces procédures ne sont pas accessibles à partir des procédures s'exécutant dans un anneau de numéro supérieur à n.

a) *Instruction CALL. Passation des paramètres*

L'instruction *CALL* permet d'effectuer simultanément l'appel d'une procédure et le changement d'anneau d'exécution du processus. Dans le cas où l'appel de la procédure correspond à une diminution de pouvoir, l'instruction *CALL* provoque un déroutement ; l'appel sera alors effectué par une procédure du moniteur. L'appel de procédure avec conservation de pouvoir peut être vu comme un cas particulier de l'appel avec augmentation de pouvoir.

Nous ne considérons donc que le cas d'une procédure p en exécution dans l'anneau n, appelant un segment-procédure q dont la parenthèse d'exécution est $[m, u]$ avec $n > u$. Le pouvoir du processus doit être augmenté.

Pour que l'appel de la procédure q soit satisfait, il faut :

— qu'il soit dirigé vers un guichet du segment q,

— que l'anneau d'exécution n soit inclus dans la parenthèse d'appel du segment q.

En supposant ces conditions remplies, examinons comment sont réalisés la transmission des paramètres ainsi que le retour correspondant vers la procédure p :

1) La procédure appelée possède par hypothèse un pouvoir suffisant pour accéder aux paramètres spécifiés par la procédure appelante p et ensuite pour retourner le contrôle à la procédure appelante.

2) La procédure appelée doit créer un bloc local dans le segment pile correspondant à son anneau d'exécution.

3) La procédure appelée doit avoir un moyen de contrôler les références aux paramètres qui lui sont passés ; en particulier elle ne doit pas être amenée à lire ou à écrire un paramètre pour lequel la procédure appelante n'aurait pas ce droit d'accès.

4) Enfin, la procédure appelée doit avoir un moyen d'identifier l'anneau dans lequel évolue la procédure appelante pour éviter de lui rendre le contrôle avec un pouvoir supérieur à celui qu'elle avait au moment de l'appel.

Le deuxième point est résolu en reliant implicitement le numéro du segment-pile à utiliser au numéro d'anneau de la procédure appelée. De cette façon le processeur calcule automatiquement le numéro du segment-pile associé à la procédure appelée et le lui communique. De plus, par convention, un mot particulier (ici le premier mot) de chaque segment-pile, désigne toujours le début du bloc disponible à l'intérieur du segment-pile ; la procédure appelée peut ainsi, à partir du numéro du segment-pile, construire son propre pointeur de pile et partant accéder à son bloc local. La préservation et la restauration de la valeur du pointeur de pile, associé à la procédure appelante, sont à la charge de la procédure appelée. Dans le cas qui nous intéresse ici, l'appel avec augmentation de pouvoir, cette convention ne viole pas les règles de protection puisque la procédure appelée possède davantage de pouvoir que la procédure appelante.

Pour résoudre le troisième point, on impose à la procédure appelante de construire, dans sa pile associée, une suite d'emplacements dont les contenus désignent les différents paramètres à passer ; elle doit fournir en outre, à la procédure appelée, l'adresse de début de cette suite, dans un registre pointeur *RPa* fixé par convention de programmation (dans CLICS c'est le registre *R5*). La procédure appelée peut alors désigner un paramètre en utilisant l'adressage indirect.

Puisque le numéro d'anneau spécifié dans le registre *RPa* a été chargé par la procédure appelante au moyen d'une instruction spéciale (cf. remarque du 5.342*c*)), sa valeur est supérieure ou égale à la valeur de l'anneau dans lequel s'exécutait la procédure appelante au moment de l'appel. Il s'ensuit que toute référence aux paramètres, par la procédure appelée, se fait avec un pouvoir identique (sinon inférieur) à celui de la procédure appelante. La manipulation des paramètres est protégée tant que la procédure appelée, dotée d'un pouvoir supérieur, ne fait pas d'action explicite pour augmenter les droits associés à une liste de paramètres (par exemple en modifiant le numéro d'anneau associé au mot indirect).

La solution du quatrième point (voir *b*) ci-après) consiste à ranger l'anneau de retour de la procédure appelante dans un registre accessible à la procédure appelée. Le retour vers la procédure appelante se fait dans cet anneau, via ce registre. Ce mécanisme est fiable puisqu'on est assuré qu'une procédure ne peut pas charger dans un registre pointeur un numéro d'anneau lui donnant des droits supérieurs aux siens.

Le cycle suivi par le processeur, au cours de l'instruction *CALL*, est donné par la figure 12. On suppose à ce stade de l'exécution que l'adresse effective de branchement spécifiée par l'instruction *CALL* se trouve dans le registre *RT*. Le registre *CO* contient l'adresse de l'instruction *CALL* et le numéro d'anneau d'exécution courant. *RT.anneau* désigne l'anneau dans lequel doit être commencée l'instruction *CALL*.

Ce schéma appelle quelques remarques.

L'accès demandé pour une instruction *CALL* est calculé par rapport au numéro d'anneau associé à l'adresse de branchement. Comme cette adresse est obtenue par l'intermédiaire d'un registre pointeur *RP* et par indirection, le numéro d'anneau associé, *RT.anneau*, a pu prendre une valeur plus élevée que celle de l'anneau dans lequel s'exécutait l'instruction. En conséquence un appel de procédure qui se présente comme s'il y avait augmentation de pouvoir par rapport à la valeur de *RT.anneau*, peut en fait être un appel avec diminution de pouvoir par rapport à l'anneau d'exécution courant *(CO.anneau)*. Puisque dans des circonstances normales, ceci représente une erreur, une violation d'accès est déclenchée en (A) même si l'anneau courant se trouve inclus dans la parenthèse d'exécution du segment appelé.

En (B), l'instruction *CALL* charge dans le registre *RPo* un pointeur vers le mot d'adresse *0* du segment-pile correspondant au nouvel anneau d'exécution. Par convention, on décide que le numéro du segment-pile est le même que le numéro du nouvel anneau d'exécution. A partir de là, la procédure appelée peut construire elle-même son pointeur-pile, ce qui lui permet de repérer son bloc local.

b) *Instruction RETURN. Détermination de l'anneau de retour*

Le retour d'une procédure et partant le changement d'anneau d'exécution du processus est réalisé par l'instruction *RETURN*. Un retour de procédure

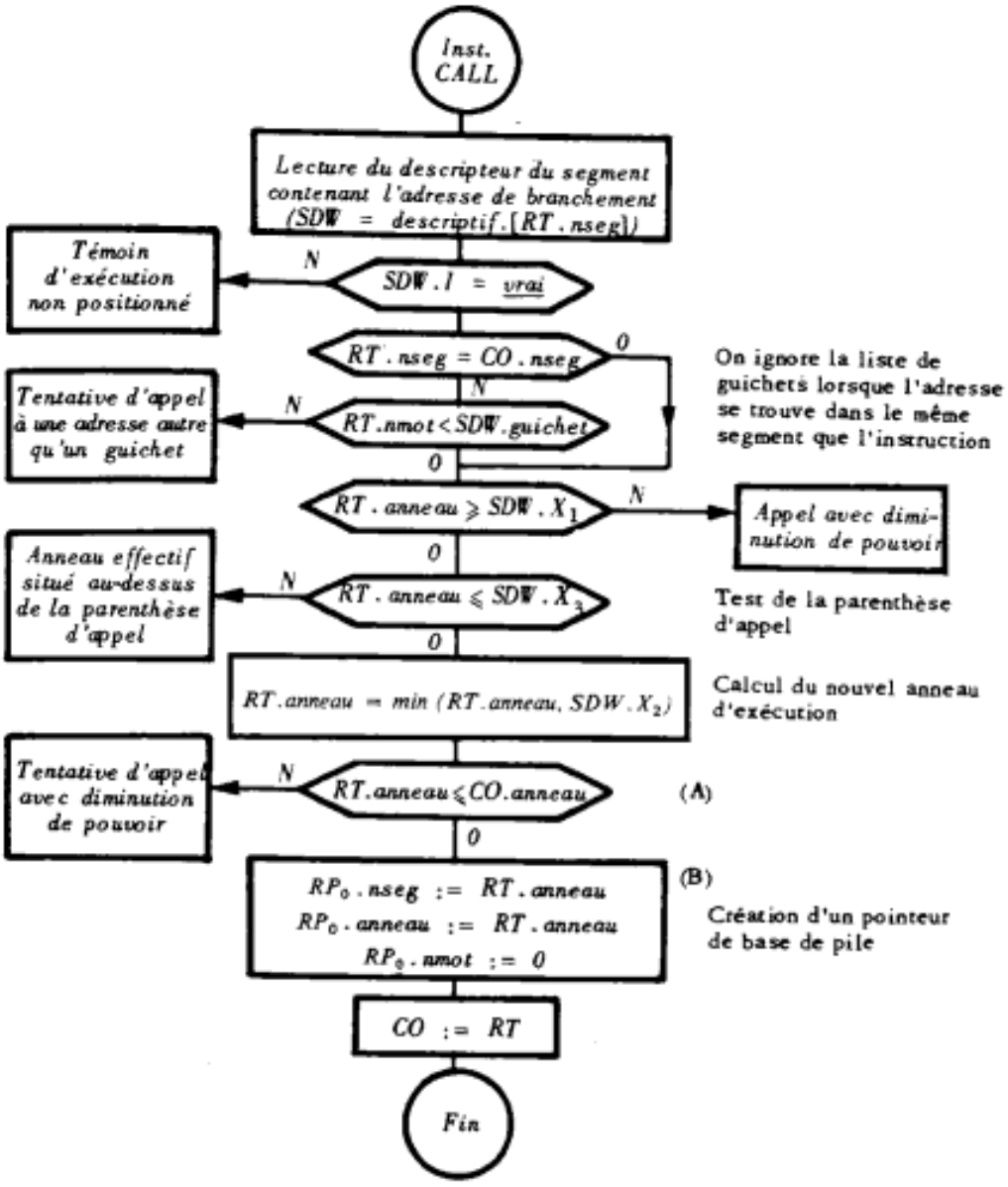


Figure 12. Interprétation de l'instruction *CALL*.

dans le sens d'une augmentation de pouvoir provoque un déroutement : le retour est alors à la charge d'une procédure du moniteur. Le retour avec conservation de pouvoir peut être vu comme un cas particulier du retour avec diminution de pouvoir.

La figure 13 illustre le fonctionnement de cette instruction ; à ce stade de l'exécution, on suppose que l'adresse de retour se trouve dans le registre de travail *RT*.

L'anneau de retour, *RT.anneau*, est celui qui est attaché à l'adresse de retour. Dans le cas d'un retour avec diminution de pouvoir, tous les numéros d'anneaux figurant dans les registres pointeurs *RP* sont remplacés par un numéro conférant un pouvoir inférieur (ou égal) à celui de l'anneau de retour.

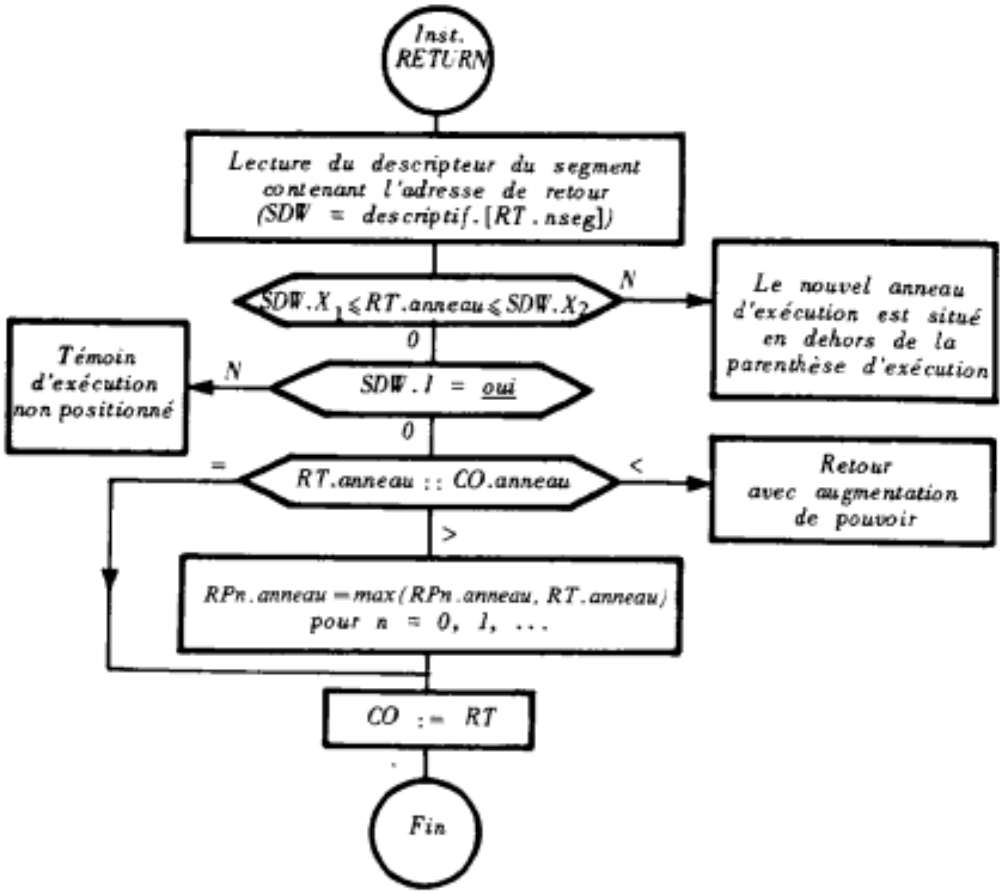


Figure 13. Interprétation de l'instruction *RETURN*.

Cette substitution, associée au fait que les registres *RP* ne peuvent être chargés que par des instructions spéciales, garantit que les numéros d'anneaux figurant dans ces registres ne confèreront pas un pouvoir supérieur à la procédure appelante.

Détermination de l'anneau de retour

Puisqu'après une instruction *CALL* les registres *RP* (excepté *RPo* qui est modifié par le *CALL*) sont encore ceux de la procédure appelante, ils contiennent des numéros d'anneaux qui définissent des pouvoirs inférieurs ou identiques à celui de la procédure appelante. Tout schéma de retour qui utilise comme anneau de retour une de ces valeurs, ne viole donc pas les règles de protection.

Avant d'exécuter l'instruction *RETURN*, la procédure appelée doit avoir rechargé le registre pointeur de pile de la procédure appelante à la valeur qu'il avait avant l'appel ; elle peut donc utiliser ce registre *RP* pour repérer indirectement l'adresse de retour. Ce mécanisme suppose bien sûr que la procédure appelante ait rangé l'adresse de retour à une position fixée dans son bloc local, avant d'exécuter l'instruction *CALL*. On garantit ainsi que l'instruction *RETURN* n'associe pas à l'adresse de retour un numéro d'anneau qui augmente le pouvoir de la procédure appelante.

Remarque 1. Lors d'un déroutement pour violation d'accès, le processeur change automatiquement d'anneau d'exécution (le nouvel anneau est l'anneau *0*) et transfère le contrôle à une adresse donnée du moniteur. Une instruction particulière permet de recharger ultérieurement l'état du processeur à la valeur qu'il avait au moment du déroutement.

Remarque 2. L'exécution des instructions privilégiées (opérations d'entrées-sorties, ...) n'est possible que dans l'anneau *0*, ce qui restreint leur utilisation aux procédures du moniteur.

5.35 CONCLUSIONS

La structure imbriquée des anneaux permet de hiérarchiser l'information (procédures et données) en fonction du niveau de protection souhaité. Ainsi un processus qui évolue dans un anneau donné a le pouvoir d'exécuter librement les segments-procédure, de lire ou de modifier les segments de données qui sont situés dans le même anneau (ou dans un anneau de pouvoir moindre) ; toute action sur des procédures ou des données qui sont situées dans un anneau doté de droits plus grands est, sinon interdite, du moins rigoureusement contrôlée.

Dans MULTICS, comme dans CLICS, il n'y a pas de processus moniteur mais uniquement des processus d'usagers ; les différentes fonctions du moniteur sont alors réparties dans chaque processus d'usager sous forme de segments-procédure dotés de droits différents suivant le niveau de protection souhaité. La décomposition du système, bâti sur ce principe est la suivante :

anneau *0* : c'est à ce niveau que sont implantées les fonctions vitales du système : procédures d'entrées-sorties, procédures réalisant le multiplexage de la mémoire et de l'unité centrale, ...
anneau *1* : on y trouve les divers services du moniteur : traducteurs, procédures de gestion de fichiers, ...
anneau *4* : à ce niveau s'exécutent les programmes des utilisateurs.

Remarque. Les anneaux *2* et *3* peuvent de la même façon être employés par un utilisateur pour définir des sous-systèmes protégés, partagés avec d'autres.

EXERCICE

1. [2]

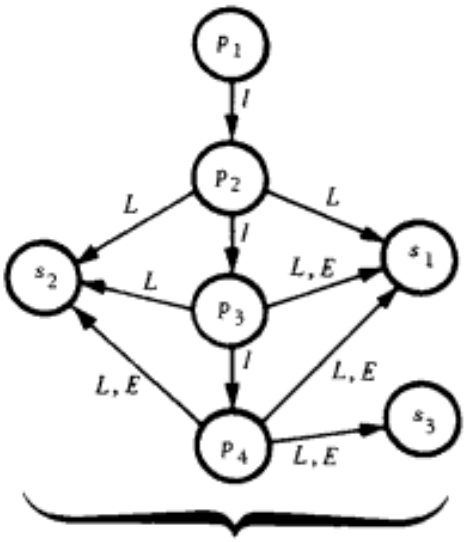
On considère 2 usagers u_1 et u_2 du système MULTICS. Ces usagers se partagent :

— des segments-procédure identifiés par p_1, p_2, p_3 *et* p_4,
— des segments de données identifiés par s_1, s_2 *et* s_3.

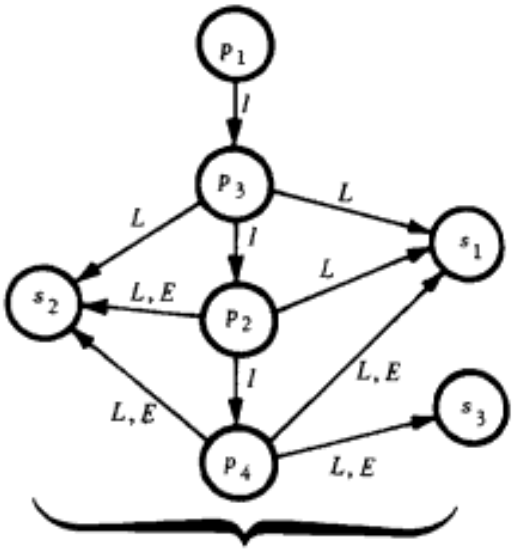
L'ordre d'appel des procédures, et les actions qu'elles effectuent, varient suivant l'usager u_1 ou u_2 pour le compte duquel elles s'exécutent. Ainsi la procédure p_1 appelle la procédure p_2 ou la procédure p_3 suivant qu'elle est exécutée par le processus u_1

ou par le processus u_2. De même la procédure p_2 peut lire le segment s_2 lorsqu'elle est exécutée par u_1 ; elle peut aussi le modifier lorsqu'elle est exécutée par u_2.

Toutes les interactions entre segments sont traduites sur les graphes suivants où les accès en lecture, écriture, exécution sont notés par des flèches respectivement étiquetées *L*, *E*, *I*.



Usager u_1



Usager u_2

On demande de définir, pour chaque segment et pour chaque usager, les parenthèses d'accès qui figurent dans la liste de contrôle d'accès du segment (on suppose que l'on dispose de 8 anneaux de protection). On rappelle que les parenthèses d'accès des procédures doivent traduire, pour chaque usager, l'ordre d'appel des procédures. On suppose que le moniteur qui traite les appels de procédure avec diminution de pouvoir les refuse pour les besoins de ce problème.

6

MESURES ET MODÈLES DE SYSTÈMES

6.1 INTRODUCTION

Ce chapitre est consacré aux divers modes d'analyse quantitative des systèmes. On y développe l'étude des systèmes par les modèles mathématiques et la simulation, puis on y décrit sommairement les techniques de mesure. On examine l'aide apportée par chacune de ces méthodes d'analyse à la conception et à la mise au point des systèmes.

6.11 INTÉRÊT ET IMPORTANCE DES ÉTUDES QUANTITATIVES

L'étude quantitative des systèmes permet de mieux comprendre leur fonctionnement en vue de l'améliorer. Le coût et la complexité du matériel et des programmes ont tendance à augmenter ; l'intérêt économique d'une bonne utilisation de ces ressources est donc certain. Indiquons les principaux champs d'application des techniques d'évaluation :

1) *Choix ou modification d'une configuration ou d'un système*

Le choix ou la modification d'un système complet nécessite la connaissance des conditions d'exploitation et des performances que l'on désire en obtenir. Des mesures sur les systèmes existants permettent parfois de prévoir quel sera le comportement du nouveau système. Les intérêts économiques mis en jeu demandent à cette prévision d'être aussi juste que possible. Une simulation

ou un modèle analytique permettent, entre autres, d'apporter des éléments de réponse aux questions suivantes :

— à coût donné, doit-on acheter un bloc de mémoire supplémentaire ou un nouveau tambour ?

— dans un système d'exploitation utilisant une partition fixe de la mémoire, comment choisir cette partition ?

2) *Comptabilité*

La connaissance du taux d'utilisation des diverses ressources d'une installation (temps d'unité centrale, mémoire centrale et secondaire, périphériques, traducteurs...) permet de répartir les frais de cette utilisation parmi une communauté d'usagers, en fonction des ressources effectivement utilisées par chacun. Les statistiques ainsi recueillies peuvent également servir de guide pour une extension du système, une modification des conditions d'utilisation, une amélioration du service fourni.

3) *Optimisation des programmes*

Les performances d'un programme accomplissant une tâche donnée peuvent varier dans des proportions considérables. Il est important de vérifier l'efficacité des programmes fréquemment utilisés. L'optimisation des performances est particulièrement importante pour les programmes (compilateurs ou systèmes d'exploitation) utilisés par un ensemble d'usagers. Des mesures permettent de déceler les parties les plus fréquemment utilisées. Lorsque le choix est possible entre plusieurs algorithmes, un modèle peut permettre de prévoir celui qui donnera les meilleurs résultats. Enfin tout programme est modifié au cours de son existence et il est utile de vérifier que ses performances ne sont pas altérées par ces modifications.

4) *Conception et construction de systèmes d'exploitation*

La conception d'un système est toujours fondée sur certaines hypothèses, parfois non explicitement formulées. Il est utile de disposer de renseignements quantitatifs sur des systèmes analogues au système projeté, afin de pouvoir établir ses spécifications et de formuler les hypothèses initiales. Celles-ci concernent aussi bien le fonctionnement interne du système que l'environnement dans lequel il doit travailler (charge, comportement des usagers). Au stade de la réalisation, les mesures permettent, par évaluations et modifications successives, d'obtenir les résultats attendus. Par ailleurs, si un modèle de simulation est utilisé pour aider aux choix de conception, la validité des résultats qu'il fournit dépend de celle des données de la simulation : il est donc utile que ces données puissent être étayées par un ensemble de mesures faites sur le système réel.

5) *Conception de matériel*

Plus encore que pour la conception de programmes, les modèles et les mesures sont utiles pour la conception du matériel car une modification du

câblage peut être très coûteuse et très longue à mettre en œuvre. La connaissance des caractéristiques de la charge est ici encore essentielle. Un exemple typique d'application est la détermination des fonctions qu'il est rentable de câbler ou de microprogrammer, dans des conditions données d'utilisation.

6) *Recherches sur les systèmes*

Le développement de modèles théoriques de programmes et de systèmes, et la compréhension du fonctionnement des systèmes existants nécessitent une bonne connaissance du comportement de ces programmes et de ces systèmes. On est donc amené à définir des grandeurs caractérisant ce comportement et à obtenir, par des mesures, des valeurs numériques de ces grandeurs.

6.12 MÉTHODES DE MESURE ET D'ÉVALUATION

Les méthodes d'étude quantitative des systèmes se divisent en deux classes, selon que l'on s'intéresse au système réel ou à une représentation de ce système ; dans chaque cas on dispose de moyens différents d'investigation. Schématiquement, on peut établir la classification suivante :

Etude des systèmes réels et des programmes :	mesures câblées mesures programmées
Représentation de systèmes et de programmes :	modèles de simulation modèles analytiques

Il est important de souligner qu'aucune de ces méthodes, en raison des limitations propres qui seront examinées plus loin, n'est suffisante à elle seule pour toutes les tâches d'évaluation. Les résultats les plus fructueux sont obtenus en combinant deux ou plusieurs des techniques ci-dessus.

Exemples. Les données d'entrée d'un modèle de simulation (ou des ordres de grandeur pour ces données) peuvent être obtenues par des mesures sur un système réel.

Un simulateur de charge programmé peut être utilisé pour étalonner un équipement de mesure câblé.

Dans un modèle de simulation, un sous-ensemble du système étudié peut être remplacé par un schéma mathématique, pour augmenter la rapidité d'exécution du programme de simulation.

6.2 LES MODÈLES DE SYSTÈME

6.21 LES OBJECTIFS DES MODÈLES

Pour étudier le comportement d'un système quelconque, que ce soit une usine ou un système d'exploitation d'ordinateur, on peut en construire un modèle contenant un certain nombre de paramètres ajustables. On n'envisage ici que la représentation d'un système d'exploitation ou d'une partie d'un tel système.

Suivant le degré de fidélité souhaité, on construit un modèle mathématique, en général très simplifié, ou bien un simulateur qui peut atteindre une grande complexité. L'intérêt du modèle analytique est qu'une formule fournit rapidement une réponse, pour toutes les valeurs des paramètres. Le simulateur a l'avantage de permettre l'étude de systèmes plus complexes, la modification d'algorithmes et l'utilisation de lois de probabilité quelconques pour lesquelles peu de propriétés mathématiques sont connues.

Une difficulté majeure des modèles provient du comportement aléatoire des systèmes. Les résultats obtenus sont des statistiques qui peuvent présenter des fluctuations telles qu'il ne soit pas possible de tirer des conclusions bien nettes. Pour cette raison, il est parfois plus utile de construire un modèle mathématique, même très simplifié, plutôt qu'un simulateur élaboré. Il faut noter en outre qu'un programme de simulation de grande taille peut se révéler coûteux à l'exécution et difficile à mettre au point, ce qui le rend peu maniable pour l'expérimentation.

6.22 EXEMPLES DE MODÈLES ANALYTIQUES

La recherche dans le domaine des modèles mathématiques se développe rapidement et dans des directions très diverses [Coffman, 73]. Citons quelques exemples : comportement global d'un système [Mc Kinney, 69 ; Coffman, 68], algorithmes de remplacement [Mattson, 70], comportement des programmes [Denning, 72], gestion des disques [Frank, 69].

6.221 **Échange de pages avec un disque à têtes fixes**

Nous nous intéressons uniquement au calcul des temps d'accès (cf. 4.532) pour deux politiques de gestion des demandes [Denning, 67].

Considérons un disque à têtes fixes, tournant uniformément avec une période de révolution r. Une piste de disque comprend un nombre entier m de secteurs, le secteur étant l'unité d'adressage. Nous poserons :

$$s = \frac{r}{m}$$

Nous supposons que la commutation d'une tête entre les modes lecture et écriture peut se faire pendant l'intervalle séparant deux secteurs consécutifs, si bien qu'il est inutile de distinguer le sens du transfert ; nous négligeons la vérification d'écriture.

Lorsque les demandes sont traitées dans l'ordre d'arrivée (politique FIFO), la première demande risque d'être éloignée de la position courante des têtes, d'où une perte de temps qui, dans le pire des cas, atteint un tour de disque ; une politique courante consiste à réordonner les demandes en attente, de façon à réduire les temps d'accès : lorsque le transfert d'un secteur est terminé, on choisit alors la demande la plus proche des têtes (politique SATF « Shortest Access Time First »). Nous allons calculer le temps d'accès moyen aux secteurs

pour les deux politiques FIFO et SATF ; nous supposons les demandes réparties uniformément sur le disque.

Le temps d'accès à un secteur donné, à partir d'un instant arbitraire, se décompose en une somme de deux variables aléatoires indépendantes :

— une variable t, de distribution uniforme $1/s$ sur $[0, s]$, représentant le temps d'accès au début du prochain secteur,

— une variable aléatoire discrète u représentant le temps d'accès au secteur demandé à partir du début du prochain secteur ; cette dernière variable peut prendre, avec la même probabilité $1/m$, les valeurs $0, s, \ldots, (m-1)\,s$.

Le temps d'accès moyen à un secteur quelconque, à partir d'un instant arbitraire, est donc égal à :

$$a = \int_0^s t \frac{\mathrm{d}t}{s} + \sum_{u=0}^{(m-1)s} u \frac{1}{m} = \frac{s}{2} + (m-1)\frac{s}{2} = \frac{r}{2}$$

Ce résultat était aisément prévisible. En se plaçant sur une limite de secteur et non plus à un instant arbitraire, le temps d'accès moyen est $(m-1)\frac{s}{2}$.

Considérons maintenant un paquet de n demandes indépendantes, ordonnées en fonction de la politique suivie. Les nouvelles demandes arrivant pendant le transfert sont ignorées tant que le paquet n'est pas épuisé.

Calculons la moyenne du temps total d'accès en fonction de n.

a) Politique FIFO

Le temps d'accès moyen de la première demande est $r/2$, comme nous venons de le voir ; celui des suivantes est égal à $(m-1)\frac{s}{2}$ car, après exécution d'une demande, les têtes se trouvent sur une limite de secteur. Pour les n demandes le temps d'accès total est donc, en moyenne :

$$A_F(n) = \frac{r}{2} + (n-1)\frac{m-1}{2}s$$

b) Politique SATF

Nous commencerons par démontrer un résultat préliminaire :

Soit un ensemble de h variables aléatoires indépendantes $t_1, \ldots, t_h$, de même distribution $p(t)$. Considérons la variable aléatoire x, toujours égale à la plus petite des variables t_i :

$$x = \min \{ t_1, t_2, \ldots, t_h \}$$

Nous avons, Pr désignant une probabilité :

$$\begin{aligned} \Pr(x > u) &= \Pr(t_1 > u, t_2 > u, \ldots, t_h > u) \\ &= (\Pr(t > u))^h \\ &= (G(u))^h \end{aligned}$$

où

$$G(u) = \Pr(t > u) = \int_u^\infty p(t)\,dt$$

D'où le résultat : la moyenne du minimum est

$$\bar{x} = \int_0^\infty \Pr(x > u)\,du = \int_0^\infty (G(u))^h\,du$$

Soit a le temps d'accès à un secteur donné, en se plaçant sur une limite de secteur. La fonction de répartition $\Pr(a < u)$ est une fonction en escalier :

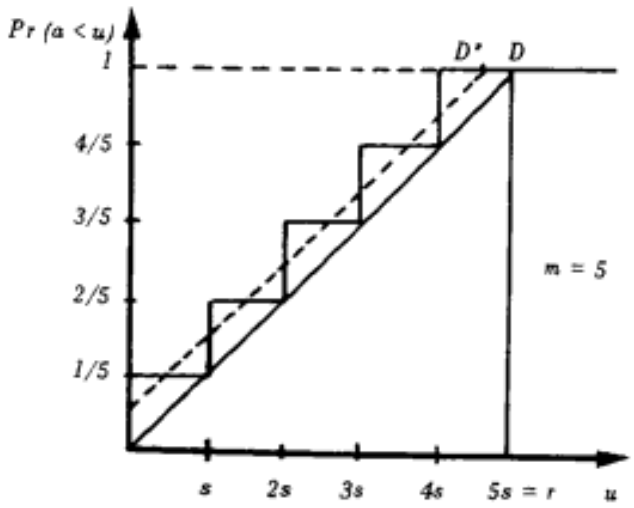


Figure 1. Fonction de répartition du temps d'accès à un secteur.

Remplaçons l'escalier par la droite en tirets D' passant par le milieu des marches ; nous obtenons :

$$\Pr(a < u) \simeq \begin{cases} 0 & u \leqslant 0 \\ \dfrac{1}{2m} + \dfrac{u}{r} & 0 < u \leqslant U \quad \text{où} \quad U = \left(m - \dfrac{1}{2}\right) s \\ 1 & u > U \end{cases}$$

En appliquant le résultat précédent nous obtenons la moyenne du temps d'accès au secteur le plus proche :

$$a_S(n) \simeq \int_0^U \left(1 - \frac{1}{2m} - \frac{u}{r}\right)^n du = \frac{r}{n+1}\left(1 - \frac{1}{2m}\right)^{n+1}$$

Si l'on se place en un point quelconque du disque et non plus sur une limite de secteur, la fonction de répartition est représentée par la droite D d'équa-

tion u/r, pour $0 \leqslant u \leqslant r$; le temps d'accès au secteur le plus proche a pour moyenne

$$\int_0^r \left(1 - \frac{u}{r}\right)^n \mathrm{d}u = \frac{r}{n+1}$$

Au total, pour les n demandes, le temps d'accès a pour moyenne approchée (exercice 1) :

$$A_S(n) \simeq \frac{r}{n+1} + a_S(n-1) + a_S(n-2) + \cdots + a_S(1)$$

La figure 2 montre clairement le gain obtenu avec la politique SATF.

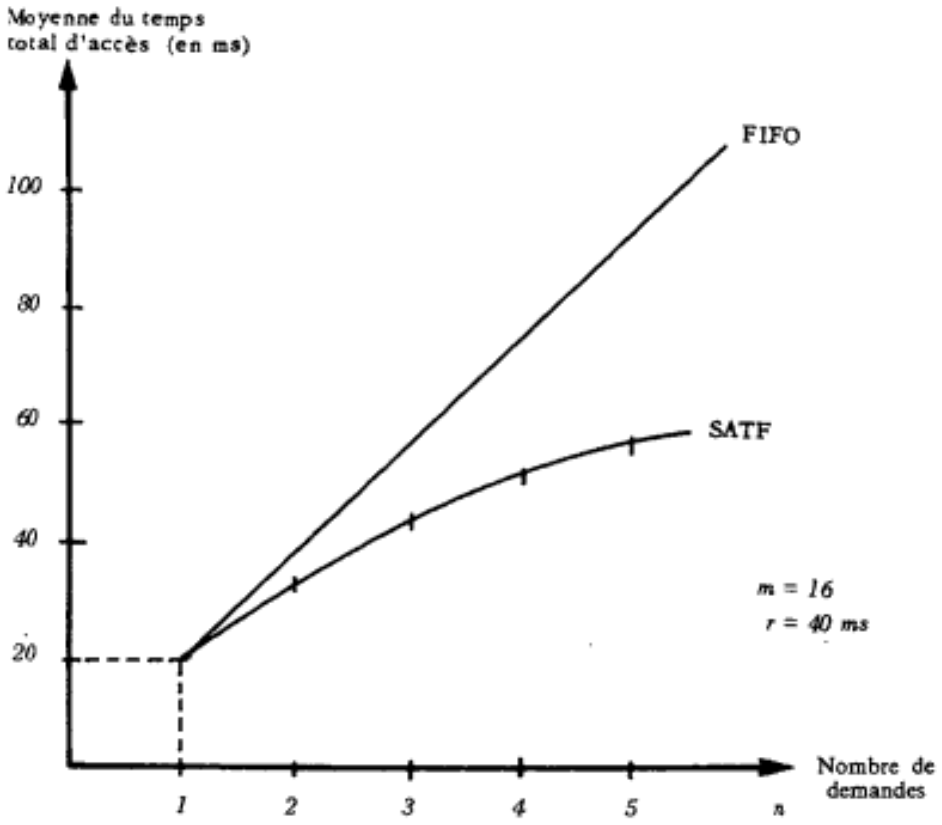


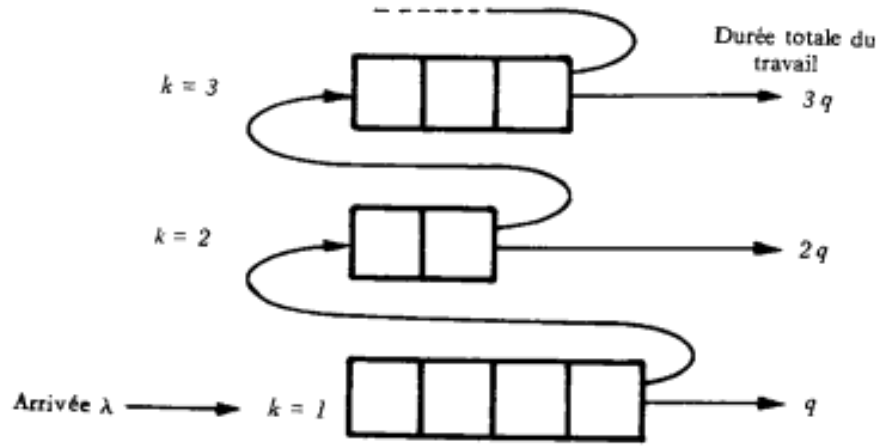
Figure 2. Comparaison de FIFO et SATF dans le modèle du disque.

6.222 **Un modèle d'allocation de processeur** [Coffmann, 73 ; Schrage, 66]

Considérons un système monoprocesseur dans lequel le temps d'unité centrale est alloué par quantum q entre les différents travaux. La politique SET (« Shortest Elapsed Time ») a pour objectif de favoriser les travaux de faible durée, sans connaître par avance le temps d'exécution des travaux.

La politique SET est la suivante : tout travail reçoit un seul quantum q à la fois, sans réquisition. Un travail libère l'unité centrale lorsqu'il est terminé ou arrivé en fin de quantum. Dès que l'unité centrale est libre, on active le travail ayant reçu le moins de quanta.

Il s'agit donc d'une politique d'ordonnancement à plusieurs niveaux (cf. 4.3). Les travaux sont rangés dans des files de priorités différentes : tous les travaux non terminés et ayant épuisé k quanta entrent dans la file $k + 1$, dans l'ordre d'arrivée (FIFO). La priorité décroît avec le numéro de la file k.



Nous supposons que les demandes arrivent suivant une distribution de Poisson, de débit moyen λ. Leurs temps d'exécution suivent une distribution discrète quelconque, g_i représentant la probabilité de durer i quanta.

Nous nous proposons de calculer le temps de réponse moyen $W(k)$ pour un travail de k quanta. Ce temps de réponse se décompose en deux parties :

— le temps d'attente moyen A, dû aux autres demandes,
— le temps d'exécution.

Nous négligerons le temps de commutation d'un travail à un autre.

Nous avons donc :

(1) $$W(k) = A + kq$$

Considérons un travail donné t, de k quanta ; ce travail va attendre d'une part parce qu'il y a des travaux commencés avant lui et non encore terminés, d'autre part parce que de nouveaux travaux arriveront pendant son traitement et recevront leurs premiers quanta en priorité.

Nous décomposons le temps d'attente A en :

(2) $$A = A' + A''$$

— A' est le temps moyen mis pour terminer le quantum en cours et pour servir toutes les demandes arrivées avant t et situées dans les k premières files d'attente.

— A'' est le temps pour servir les demandes arrivées après t, jusqu'au $(k-1)$-ième quantum inclus ; à partir du moment où t entame son dernier quantum k les nouvelles arrivées n'ont plus à être prises en compte.

a) Calcul de A''

Soit $G(n)$ la fonction de répartition des temps d'exécution, c'est-à-dire

$$G(n) = \sum_{i=1}^{n} g_i$$

Le temps moyen d'unité centrale consommé par une demande ayant reçu au plus n quanta s'écrit :

$$S(n) = \sum_{i=1}^{n} iqg_i + nq(1 - G(n))$$

le second terme correspondant aux travaux de plus de n quanta.

Pendant l'attente de durée A du travail t et l'exécution de ses $k-1$ premiers quanta, le nombre d'arrivées croît en moyenne de :

$$\lambda[A + (k-1)\,q]$$

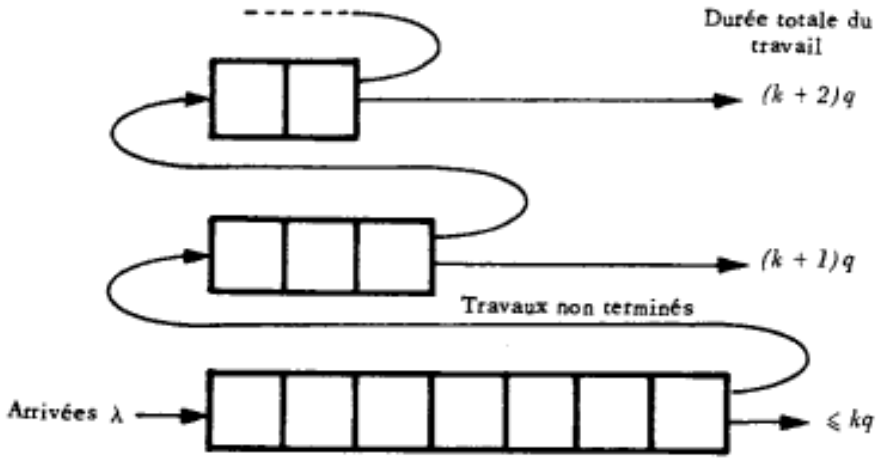
Tant que t n'aura pas reçu son dernier quantum, chacun des nouveaux travaux prendra, en moyenne, $S(k-1)$ quanta ; d'où :

$$A'' = \lambda[A + (k-1)\,q]\,S(k-1) \tag{3}$$

b) Calcul de A'.

Nous allons modifier artificiellement la politique de façon à faciliter le calcul du terme A'.

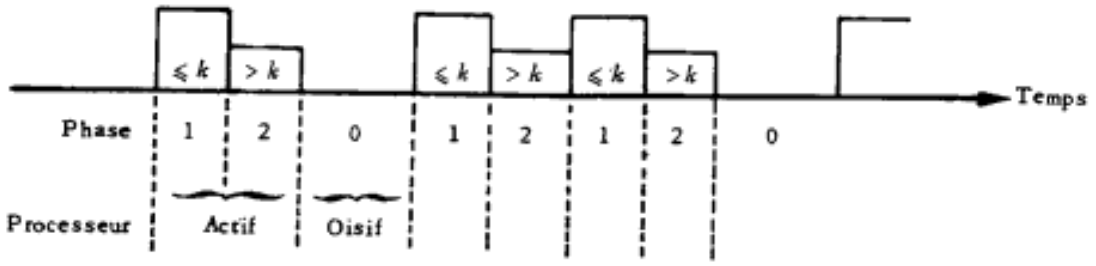
Considérons donc une politique où toutes les demandes dans la première file peuvent recevoir jusqu'à k quanta, mais un seul quantum dans les files de priorité inférieure :



Un travail donné t de k quanta attendra donc uniquement dans la première file et sera servi en une seule fois.

Soit A_1 le temps d'attente de t ; ce temps d'attente est dû aux demandes précédant t dans la première file et au travail en cours. Nous allons montrer que l'attente A_1 est égale à l'attente A' dans la première politique.

Portons, sur l'axe du temps, les périodes oisives de l'unité centrale, désignées par les phases 0 sur la figure ci-dessous :



Dans les deux politiques, le temps de gestion des travaux est négligé ; le temps total d'exécution d'un groupe donné de travaux ne dépend donc pas de la politique : la distribution des intervalles d'oisiveté est la même pour les deux politiques.

Dans un intervalle d'activité, nous distinguons les phases de type 1 pendant lesquelles les travaux reçoivent uniquement des quanta d'ordre k au plus, et les phases de type 2 où les quanta sont d'ordre supérieur à k ; les distributions de ces phases sont également indépendantes de la politique choisie.

En résumé la distribution des trois phases est la même dans les deux politiques.

Reprenons la demande t de k quanta :

— si elle se présente alors que le système est dans une période d'oisiveté, l'attente est nulle dans les deux politiques ;

— si elle se présente alors que le système est dans une phase 2, la demande t attend la fin du quantum en cours, et cette attente est la même dans les deux politiques ;

— si elle se présente alors que le système est dans une phase 1, on peut ne considérer que les travaux arrivés avant t et ayant reçu moins de k quanta. Le temps d'exécution de ces travaux jusqu'au k-ième quantum est indépendant de la politique. Le temps d'attente de t est égal au temps total d'exécution de ces travaux, identique dans les deux politiques, moins la somme des temps déjà reçus. Cette somme est égale au temps séparant le début de la phase 1 de l'arrivée de t ; elle est donc la même dans les deux politiques. En conséquence, le temps d'attente de t est le même.

Dans les trois cas, l'attente de t est indépendante de la politique. Il en est de même des probabilités respectives de ces cas. Il en résulte que :

$$A_1 = A'$$

Nous pouvons écrire, pour la seconde politique :

$$A_1 = R(k) + mS(k)$$

où :

— $R(k)$ est le temps moyen pour terminer le travail en cours,
— m la longueur moyenne de la première file.

Nous ferons appel à la relation de Little [Little, 61] qui s'applique à un système en équilibre, dans lequel entrent des demandes avec un débit moyen d'arrivée a ; si S est le temps moyen de résidence dans le système alors le nombre moyen L de demandes dans le système est égal à :

$$L = aS$$

quelle que soit la façon dont les demandes sont traitées (exercice 2).

Nous obtenons ici :

$$m = \lambda A_1$$

Il vient :

$$A_1 = \frac{R(k)}{1 - \lambda S(k)} \tag{4}$$

Soit $S_2(k)$ la moyenne du carré du temps de service jusqu'au k-ième quantum :

$$S_2(k) = \sum_{i=1}^{k} (iq)^2 g_i + (kq)^2 [1 - G(k)]$$

Nous admettrons le résultat suivant, sans démonstration :

$$R(k) = \frac{\lambda}{2}\left(S_2(k) + q^2 \sum_{i=k}^{\infty} [1 - G(i)]\right) \tag{5}$$

En substituant (4) et (3) dans (2) nous obtenons l'équation :

$$A = \lambda[A + (k - 1)\, q]\, S(k - 1) + \frac{R(k)}{1 - \lambda S(k)}$$

d'où nous pouvons extraire la valeur de A.

Finalement le temps de réponse moyen (1) s'écrit :

$$W(k) = \frac{\lambda}{2} \frac{S_2(k) + q^2 \sum_{i=k}^{\infty} [1 - G(i)]}{[1 - \lambda S(k)]\,[1 - \lambda S(k - 1)]} + \frac{(k - 1)\, q}{1 - \lambda S(k - 1)} + q$$

Dans le cas d'une politique FIFO, le temps de réponse est donné par la formule suivante [Conway, 67] :

$$W = S + \frac{\lambda}{2} \frac{S_2}{1 - \lambda S}$$

où

$$S = \sum_{i=1}^{\infty} (iq)\, g_i \quad \text{et} \quad S_2 = \sum_{i=1}^{\infty} (iq)^2\, g_i$$

La figure 3 compare les politiques SET et FIFO ; elle montre que le temps de réponse des travaux d'un quantum demeure voisin du quantum dans la politique SET.

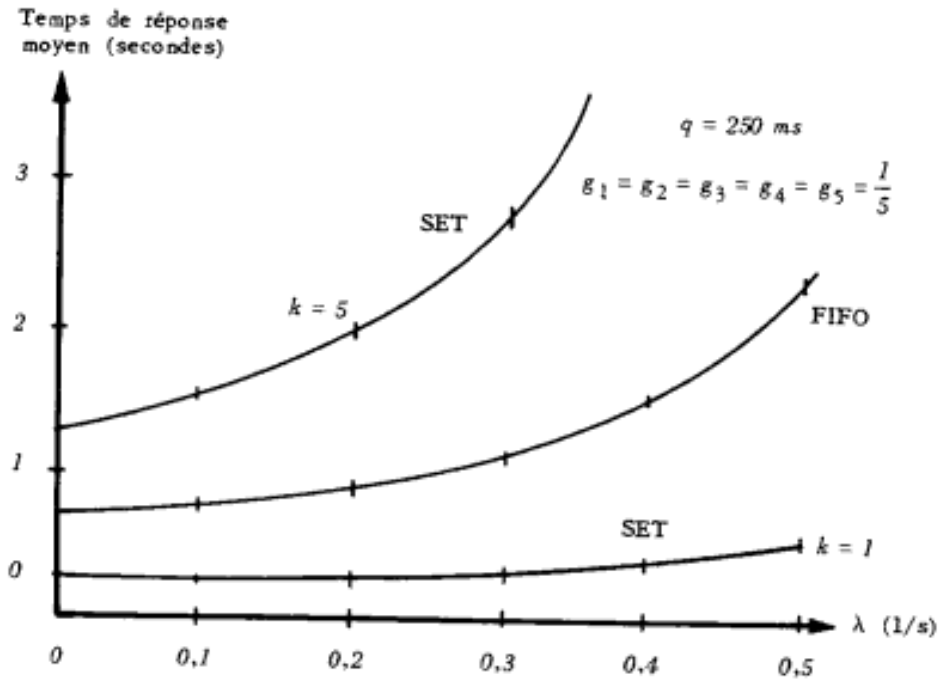


Figure 3. Comparaison des politiques FIFO et SET pour l'allocation de processeur.

6.223 Un modèle de système conversationnel

La complexité d'un système n'implique pas nécessairement qu'il faille un modèle compliqué pour calculer certaines de ses caractéristiques. C'est ainsi que Scherr a construit un modèle qui, en dépit d'hypothèses simplificatrices, représente convenablement le temps de réponse du système CTSS [Scherr, 65] (Fig. 4).

Dans ce système, les demandes provenant des consoles sont traitées en FIFO et une seule à la fois (monoprogrammation).

L'usager conversationnel a un comportement cyclique avec une phase de réflexion (usager oisif), pendant laquelle le programmeur réfléchit et tape sa demande, suivie d'une phase d'attente de la réponse (usager actif).

Dans ce modèle le temps de réflexion est représenté par une distribution exponentielle, de moyenne R, soit :

$$\Pr\{\text{réflexion} < t\} = 1 - \exp\left(-\frac{t}{R}\right)$$

Le temps de traitement d'une demande est représenté également par une distribution exponentielle, de moyenne T ; ce temps de traitement comprend le temps de chargement du programme correspondant et le temps de son exécution.

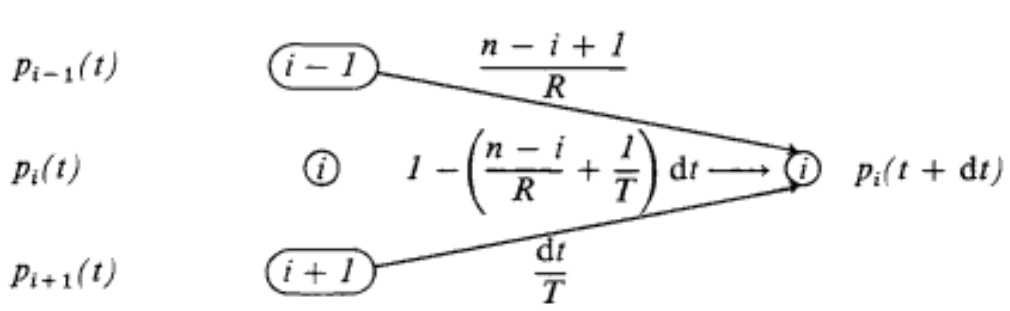
Si un usager donné est oisif à l'instant t, la probabilité qu'il quitte l'état oisif entre t et $t + \mathrm{d}t$ est égale à $\mathrm{d}t/R$; s'il est actif à l'instant t, la probabilité qu'il devienne oisif est : $\mathrm{d}t/T$.

Soit n le nombre de consoles en interaction, nombre que nous supposerons constant. A un instant donné t, on observe $0, 1, \ldots$ ou n usagers actifs ; soit $p_0(t), \ldots, p_n(t)$ les probabilités respectives d'observer ces divers états à l'instant t. A l'instant $t + \mathrm{d}t$ ces probabilités ont changé, car pendant l'intervalle $\mathrm{d}t$ le premier usager actif a pu devenir oisif, et un ou plusieurs usagers, oisifs à l'instant t, ont pu devenir actifs. Nous négligerons les transitions avec changement d'état de deux usagers ou plus, car les probabilités correspondantes sont de l'ordre de $(\mathrm{d}t)^2$.

Supposons qu'à l'instant $t + \mathrm{d}t$ on observe i usagers actifs, avec $i \neq 0$ et $i \neq n$: au temps t il y avait $i - 1$, i ou $i + 1$ usagers. La probabilité de la transition $i + 1 \to i$ est égale à $\mathrm{d}t/T$. Celle de $i - 1 \to i$ est égale à

$$\frac{n - (i - 1)}{R}\,\mathrm{d}t\,,$$

car il y a $n - (i - 1)$ usagers à l'instant t susceptibles de passer à l'état actif. Enfin la probabilité de la transition $i \to i$ égale 1 moins la probabilité de quitter l'état i, à savoir : $\left(\frac{1}{T} + \frac{n - i}{R}\right)\mathrm{d}t$.



Appliquons le théorème des probabilités composées :

$$\begin{aligned}\Pr(i, t + \mathrm{d}t) = {} & \Pr(i - 1, t) \times \Pr(i - 1 \to i) \\ & + \Pr(i, t) \times \Pr(i \to i) \qquad (i \neq 0, \quad i \neq n) \\ & + \Pr(i + 1, t) \times \Pr(i + 1 \to i)\end{aligned}$$

d'où

$$p_i(t + dt) = p_i(t) + dt\left[- p_i(t)\left(\frac{n - i}{R} + \frac{1}{T}\right) + \right.$$
$$\left. + p_{i-1}(t)\frac{n - i + 1}{R} + p_{i+1}(t)\frac{1}{T}\right]$$

On vérifiera que pour $i = 0$ et $i = n$ on a :

$$p_0(t + dt) = p_0(t) + dt\left[- p_0(t)\frac{n}{R} + p_1(t)\frac{1}{T}\right]$$
$$p_n(t + dt) = p_n(t) + dt\left[- p_n(t)\frac{1}{T} + p_{n-1}(t)\frac{1}{R}\right]$$

Ces équations peuvent s'écrire en utilisant les dérivées :

$$p_i'(t) = - p_i(t)\left[\frac{n - i}{R} + \frac{1}{T}\right] + p_{i-1}(t)\frac{n - i + 1}{R} + p_{i+1}(t)\frac{1}{T}$$
$$p_0'(t) = - p_0(t)\frac{n}{R} + p_1(t)\frac{1}{T}$$
$$p_n'(t) = - p_n(t)\frac{1}{T} + p_{n-1}(t)\frac{1}{R}$$

Au bout d'un certain temps, le système atteint un régime d'équilibre si n reste constant ; ce régime d'équilibre est caractérisé par des probabilités indépendantes du temps, soit :

$$p_i(t + dt) = p_i(t) = p_i \qquad \forall i$$

d'où

$$p_i'(t) = 0 \qquad \forall i$$

Le système différentiel se réduit à un système d'équations linéaires qui donne, après quelques calculs algébriques :

$$p_i = \frac{n\,!}{(n - i)\,!}\left(\frac{T}{R}\right)^i p_0$$

Utilisant la relation de normalisation :

$$\sum_{i=0}^{n} p_i = 1$$

on en déduit :

$$\frac{1}{p_0} = \sum_{i=0}^{n} \frac{n\,!}{(n - i)\,!}\left(\frac{T}{R}\right)^i$$

La quantité p_0 représente la fraction du temps pendant laquelle le système est oisif, c'est-à-dire sans usagers actifs.

Le nombre moyen d'usagers actifs, à l'équilibre, s'écrit :

$$q = \sum_{i=0}^{n} i p_i$$

Le temps de réponse moyen W représente, par définition, la durée moyenne de la période d'activité d'un usager. Pour le calculer nous remarquerons qu'il y a, en moyenne et par unité de temps, autant d'usagers entrant dans l'état actif que d'usagers quittant l'état actif ou, ce qui revient au même, entrant dans l'état oisif.

Appliquons la formule de Little (cf. 6.222) aux deux ensembles d'usagers actifs et oisifs. Considérons uniquement des valeurs moyennes : s'il y a q usagers qui restent pendant une durée W dans l'état actif et si λ est le débit des arrivées dans l'état actif, nous avons :

$$q = \lambda W$$

De même, s'il y a $n - q$ usagers oisifs, qui demeurent pendant une durée R dans cet état et si μ est le débit des arrivées dans l'état oisif nous avons :

$$n - q = \mu R$$

Ecrivons que $\lambda = \mu$, soit :

$$\frac{q}{W} = \frac{n - q}{R}$$

d'où l'on peut tirer, en utilisant la définition de q (exercice 3) :

$$W = \frac{nT}{1 - p_0} - R$$

Cette expression peut s'écrire sous une forme facilement interprétable :

$$1 - p_0 = \frac{nT}{W + R}$$

La quantité $1 - p_0$ représente l'activité relative moyenne du système. Un cycle d'usager dure en moyenne $W + R$; pendant un cycle moyen les n usagers sont passés une fois et une seule en moyenne par l'état oisif ; durant ce cycle le système a été actif pendant une durée totale nT.

6.23 EXEMPLES DE SIMULATION

Nous n'entrerons pas dans les détails de mise en œuvre d'une simulation. On pourra consulter à titre d'exemple [Scherr, 65] et [Pinkerton, 68]. Signalons qu'il existe de nombreux langages spécialisés (SIMULA, SIMSCRIPT, GPSS, ...).

Le domaine d'application de la simulation n'est pas limité *a priori* : configuration d'un IBM 360/67 [Nielsen, 67], algorithmes d'ordonnancement

pour le TSOS du RCA [Oppenheimer, 68], organisation d'une mémoire [Pirtle, 67], échanges de pages dans le système ESOPE [Bétourné, 72], comparaison de divers algorithmes de gestion de disques [Teorey, 72], etc...

Il paraît utile d'exposer quelques résultats pour montrer l'intérêt de l'outil, ce qui nous permettra en outre d'illustrer des notions présentées dans les chapitres précédents.

Exemple 1 : le système CTSS [Scherr, 65].

Ce système conversationnel a été simulé par Scherr qui est en outre parvenu à formuler un modèle analytique simple (cf. 6.223). Le système réel a fourni les paramètres caractéristiques de la charge (cf. 4.2).

La figure 4 montre des points expérimentaux obtenus par mesure sur le système réel. En ordonnée, on a porté le temps de réponse moyen divisé par le temps moyen d'unité centrale par demande. La simulation et la théorie sont en bon accord avec les mesures.

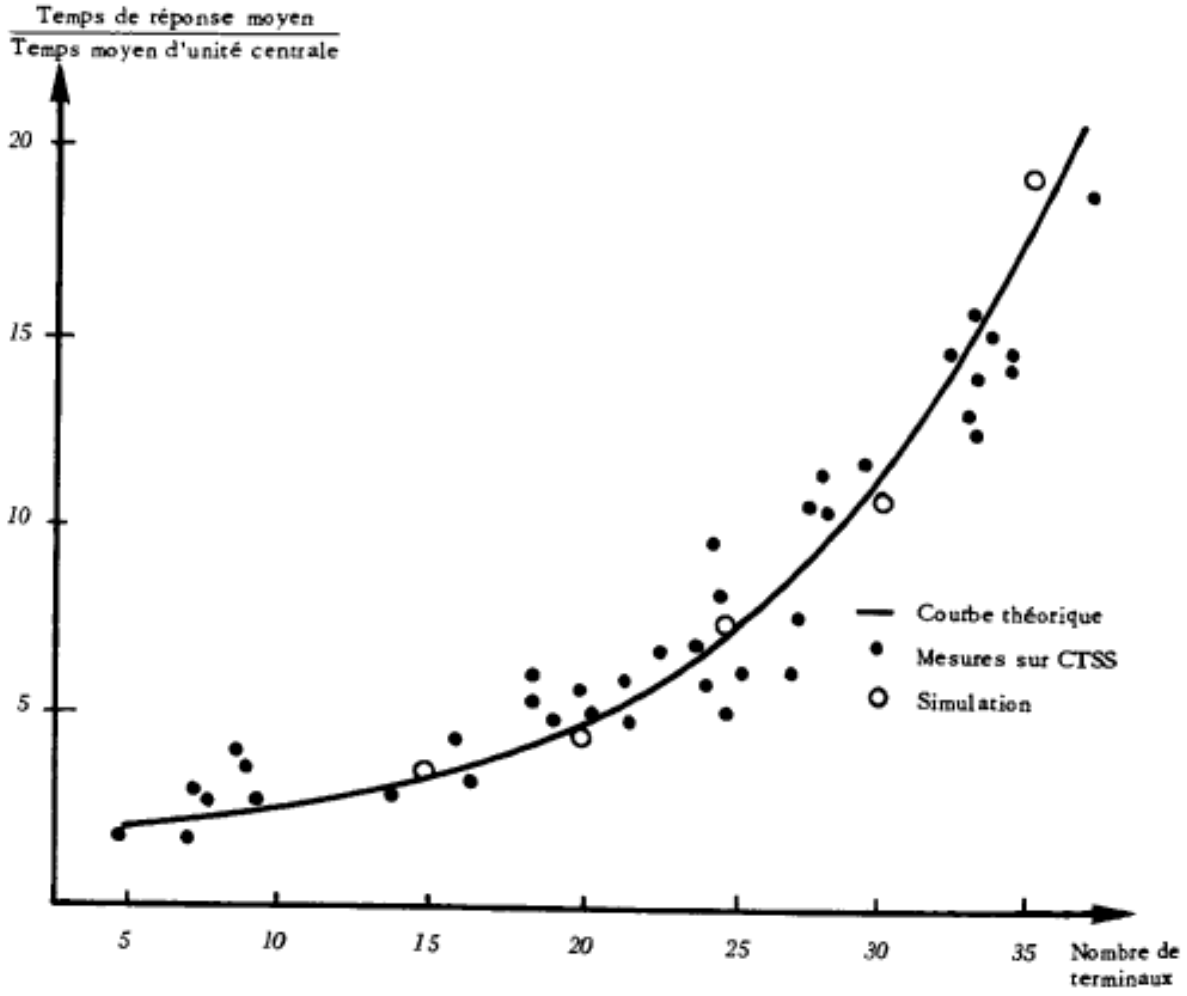


Figure 4. Temps de réponse de CTSS en fonction de la charge [Scherr, 65].

Exemple 2 : le système TSOS [De Meis, 69].

Le système TSOS de RCA utilise un ordinateur SPECTRA 70/46. Il gère des travaux conversationnels en multiprogrammation. L'allocation de mémoire se fait par page avec chargement à la demande (cf. 4.441).

Les programmes utilisateurs sont simulés par des processus cycliques qui déterminent la chaîne des références aux pages de la mémoire, tant en lecture qu'en écriture.

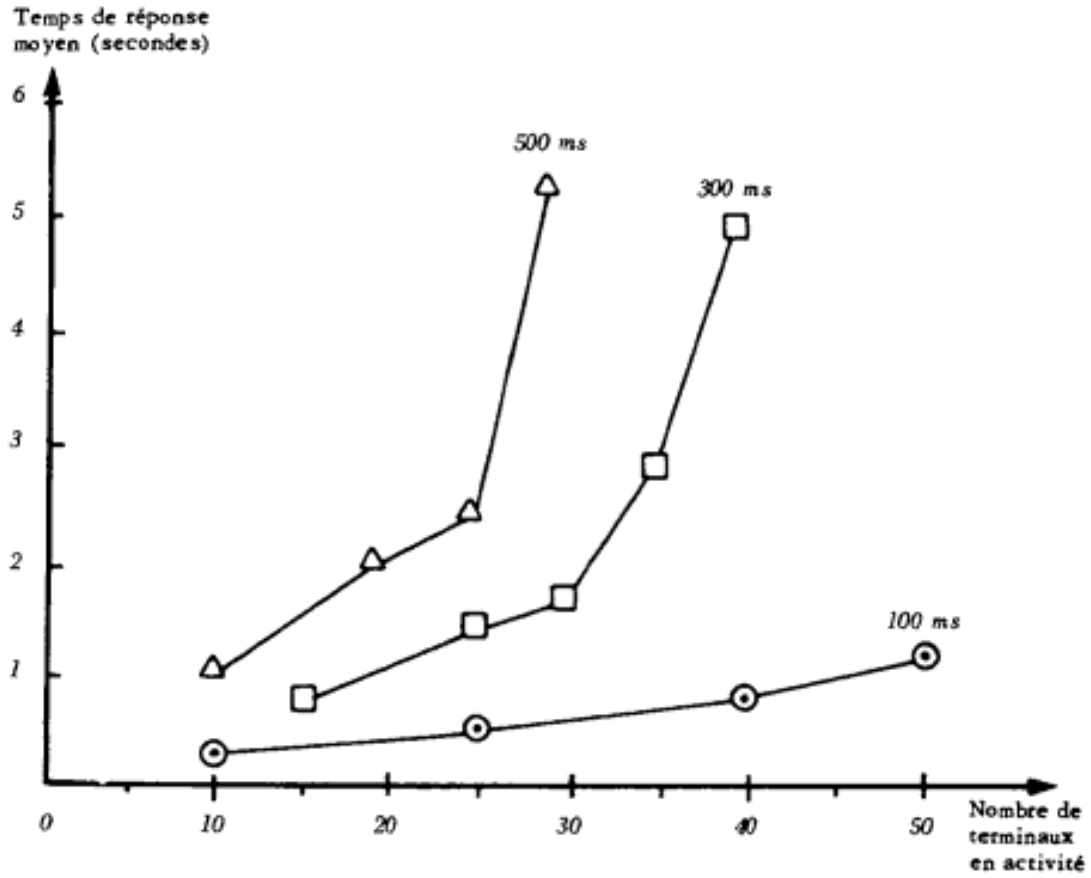


Figure 5. Temps de réponse de TSOS en fonction de la charge et du temps moyen d'unité centrale par interaction [De Meis, 69].

La figure 5 montre l'influence du temps moyen d'unité centrale par demande sur le temps de réponse moyen. On note pour les deux courbes supérieures une montée rapide, le coude correspondant à une utilisation de l'unité centrale dépassant 90 %.

Le phénomène de l'écroulement du système (cf. 4.61) est remarquablement illustré par la figure 6, où l'on constate une montée brutale du temps de réponse pour une charge de 44 consoles. Cette rupture est due à une saturation de la mémoire, entraînant des vols de pages entre les tâches. Cette succession de vols, une fois amorcée, va en s'amplifiant ; les échanges avec la mémoire secondaire se multiplient et l'activité de l'unité centrale chute brusquement (de 80 à 50 % lors de l'écroulement). On évite ce phénomène en pratiquant une politique de réservation : une tâche n'est admise en mémoire qu'à la condition qu'il y ait suffisamment de place libre pour son espace de travail, et qu'elle puisse ainsi acquérir l'essentiel de ses pages sans devoir en voler aux autres.

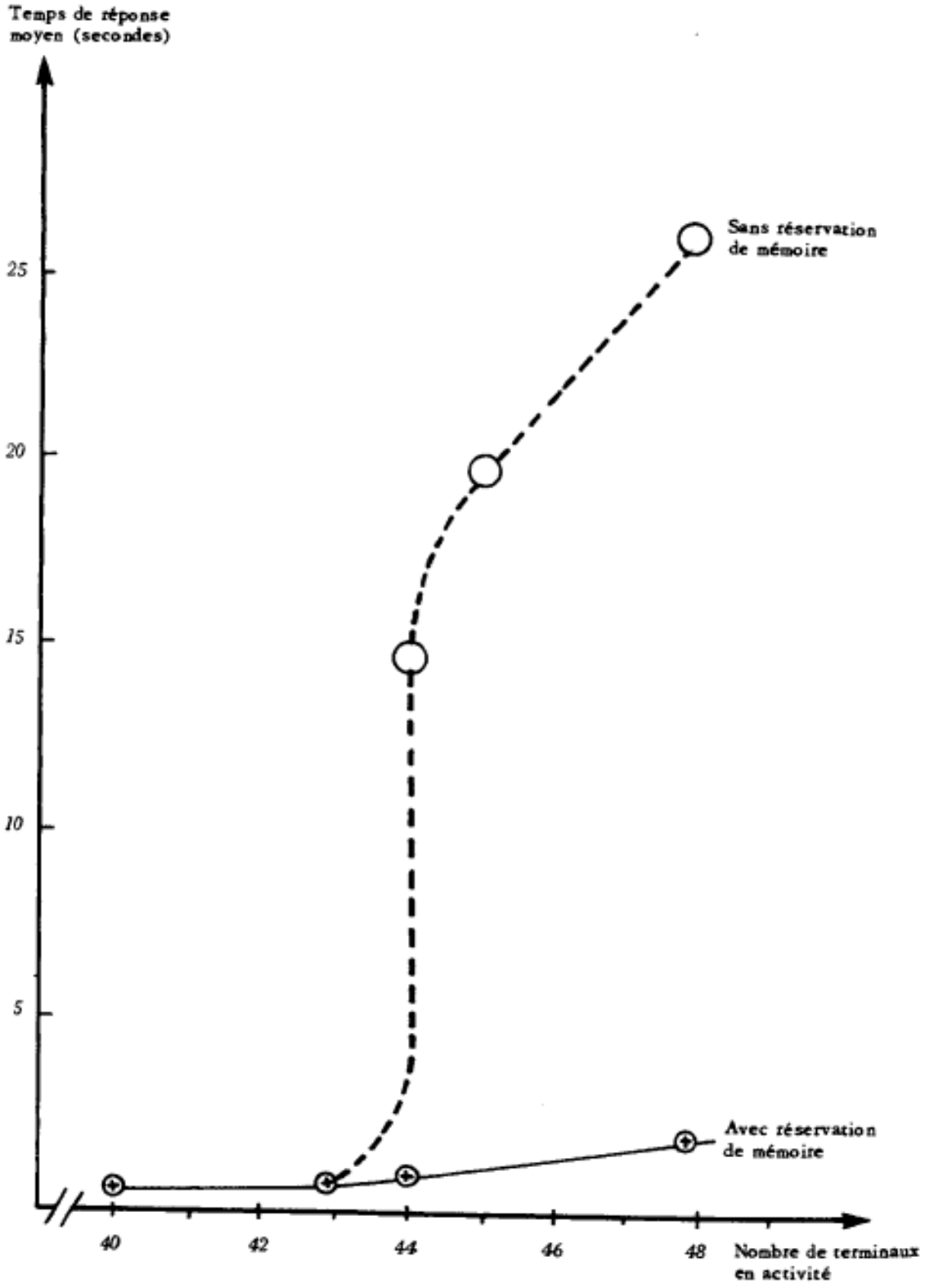


Figure 6. Simulation de l'écroulement du système TSOS [De Meis, 69].

6.3 MESURES SUR LES SYSTÈMES RÉELS

Les mesures sur un système réel servent, comme celles obtenues avec un modèle, à améliorer les performances ; en outre le système peut les exploiter dynamiquement, pour contrôler la charge par rétroaction (cf. 4.6). Un système réel permet de mesurer les caractéristiques des demandes (cf. 4.2) et de vérifier la validité d'un modèle.

6.31 NATURE DES MESURES

Les données élémentaires collectées par les mécanismes de mesure sont, pour la plupart, des contenus de mémoire, des événements, des comptages d'événements (fréquences). On peut classer ces données en quatre groupes :

a) les mesures d'intervalles de temps et les mesures instantanées de grandeurs autres que le temps (longueur d'une file d'attente, volume de mémoire occupé, longueur d'un message, ...),

b) les dénombrements dans le temps (fréquence d'événements),

c) les contenus d'ensembles à un instant donné (files d'attente, mémoires virtuelles, ...),

d) les suites d'événements, représentant la succession des actions dans le temps.

Un nombre seul est inutilisable : il faut connaître le contexte dans lequel il a été obtenu. Pour relier la mesure à l'évolution du système, on peut :

— soit noter l'heure de la mesure,
— soit se référer à des événements connus du système.

Par exemple, on peut mesurer la longueur d'une file d'attente, périodiquement toutes les 100 ms, ou encore chaque fois qu'un processus donné est activé. On distinguera donc deux méthodes d'acquisition de données :

— les mesures non synchronisées avec les changements d'état du système (mesures périodiques par exemple),
— les mesures synchronisées avec l'évolution du système ; dans ce cas, on possède une information supplémentaire qui est en corrélation avec l'activité du système.

6.32 MÉTHODOLOGIE DES MESURES

Pour contrôler la validité des mesures diverses méthodes sont possibles :

— multiplier les points de mesure de façon à mesurer une même grandeur de plusieurs façons indépendantes,
— mesurer indépendamment des grandeurs reliées entre elles et vérifier que la relation entre ces grandeurs est satisfaite,
— comparer les résultats avec les prédictions d'un simulateur ou d'un modèle mathématique,

— faire des mesures sous une charge artificielle, de caractéristiques très simples, permettant de calculer les résultats par avance.

Lors de la prise des mesures, il est utile de connaître leurs ordres de grandeurs de façon à détecter dès que possible toute anomalie.

La reproduction d'une expérience est un problème délicat en raison de la nature aléatoire des phénomènes ; en outre le système peut évoluer et il est alors nécessaire de connaître sans ambiguïté et d'enregistrer, à chaque expérimentation, l'état de la version utilisée.

6.33 MÉCANISMES DE MESURE

6.331 **Généralités**

On peut distinguer les mécanismes câblés, externes au système, et les mécanismes programmés. Il n'y a pas en fait de distinction nette entre ces deux classes : en effet, l'appareillage peut être entièrement externe ou encore utiliser une partie du système, à savoir des alimentations ou un canal ; quant aux mesures programmées, elles font appel à des mécanismes câblés internes au système tels que les horloges, voire à des instructions câblées spécialement conçues pour faciliter ces mesures.

Nous distinguerons cependant, pour la commodité de l'exposé, mécanismes externes et mécanismes internes, ces derniers étant soit câblés, soit programmés.

6.332 **Appareillage de mesure externe**

Un montage classique comporte un ensemble de sondes de haute impédance connectables aux circuits de la machine, un choix de circuits logiques permettant de combiner les signaux des sondes, un système d'enregistrement sur bande magnétique et des horloges de haute fréquence (1 MHz) pour noter l'heure des événements enregistrés. Parfois un ordinateur est chargé de traiter les données en temps réel.

L'intérêt de cette technique est d'éviter les interférences avec le système étudié. On enregistre une grande quantité d'information qui est traitée ultérieurement. En combinant les sondes entre elles, au moyen de circuits électroniques, on diminue le nombre d'enregistrements en ne conservant que l'information utile. Il est possible d'obtenir des mesures très fines, au niveau de la microseconde, sur toutes les parties du système.

Etant donné l'importance croissante des mesures, on peut supposer que les machines futures se prêteront plus aisément à l'expérimentation ; un panneau de connexion, fourni par le constructeur, protégera la machine contre les fausses manœuvres ; des mécanismes divers permettront la collecte de renseignements sur l'utilisation des processeurs, des canaux et des mémoires ; quelques mesures bien choisies, affichables sur le panneau de commande, avertiront l'opérateur des situations anormales.

Exemple 1 : mesure d'activité d'une unité centrale.

On suppose qu'une bascule indique si le processeur est oisif ou non (Fig. 7) :

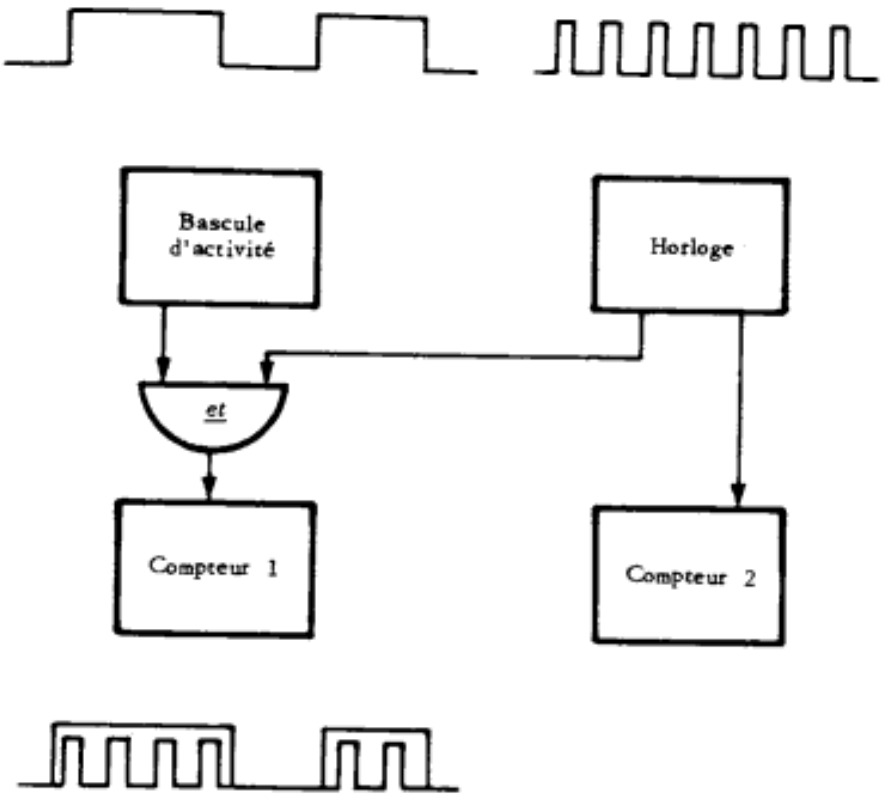


Figure 7. Mesure de l'activité d'un processeur.

Un circuit *et* entre la bascule et une horloge permet de mesurer la durée totale d'activité dans le compteur 1 ; le compteur 2 donne le temps total de la mesure et le rapport compteur 1/compteur 2 mesure l'activité relative du processeur.

Exemple 2 : le TS/SPAR pour IBM 360 [Schulman, 67].

Cet appareil, conçu pour étudier l'activité d'un IBM 360/67, contrôle jusqu'à 256 sondes et 48 compteurs d'événements ; le système comporte en outre un enregistreur sur bande magnétique et des horloges.

De nombreuses mesures sont possibles :

a) activité des processeurs : temps passé en attente, en mode maître, en mode esclave ; fréquence et types des interruptions par processeur ; fréquence d'exécution de programmes donnés, des instructions *SVC* ;

b) multitraitement : nombre moyen d'arrêts d'une tâche ; nombre de pages demandées, libérées et modifiées pendant la tranche de temps ; temps de réponse ;

c) programmes et données partagés : ralentissement des processeurs en cas de conflit d'accès à la mémoire ; fréquence d'utilisation des programmes ; pourcentage de temps de processeur en exécution de procédure réentrante ;

d) utilisation de la mémoire associative : pourcentage de transformations d'adresses effectuées par la mémoire associative ; nombre de modifications de la mémoire associative ;

e) entrées-sorties : activité de chaque canal ; fréquence et durée des ordres de lecture et d'écriture pour chaque périphérique ; recouvrement d'activité entre un canal et une unité centrale ; trafic de pages avec les disques et les tambours ; nombre d'instructions d'entrées/sorties exécutées.

6.333 Mécanismes câblés internes au système

L'appareillage de mesure peut utiliser des ressources propres au système, comme des horloges ou un canal. Un montage intéressant consiste à faire jouer à l'appareil le rôle d'un périphérique, connecté à un canal. Dans certains cas, les mesures sont exécutées par un ordinateur satellite, adapté aux problèmes en temps réel ; de cette façon le système a la possibilité de commander les mesures, de les lire pour les traiter, voire de les utiliser pour améliorer son propre fonctionnement.

Exemple 1 : les horloges du CII 10070.

Le CII 10070 possède 4 horloges, déclenchant périodiquement des interruptions ; une interruption d'horloge est acquittée par une seule instruction *MTW* (« Modify and Test Word ») qui augmente de 1 un compteur de temps ; l'instruction *MTW* peut adresser indirectement le compteur, si bien qu'il est possible de changer dynamiquement le compteur affecté à l'horloge, en changeant simplement la valeur du pointeur sur le compteur (Fig. 8).

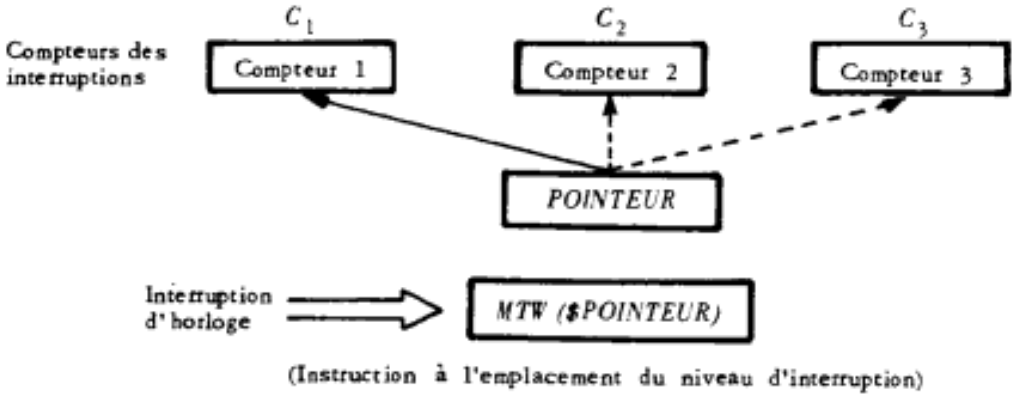


Figure 8. Partage d'une horloge par plusieurs compteurs.

Considérons, à titre d'application, la mesure du parallélisme entre l'unité centrale et le canal ; si l'on ne distingue que deux états, actif et oisif, pour chacun de ces deux processeurs, la mesure se programme très simplement en utilisant 4 compteurs contigus en mémoire, correspondant aux 4 cas suivants :

Compteurs	*C1*	*C2*	*C3*	*C4*
Unité centrale Canal	active actif	active oisif	oisive actif	oisive oisif

On constate qu'il suffit de faire ± 2 sur le pointeur du compteur lorsque l'unité centrale change d'état, et ± 1 lorsque le canal change d'état ; dans tous les cas une seule instruction suffit pour la transition (*MTW* sur le pointeur du compteur).

Exemple 2 : système MULTICS.

Le système MULTICS utilise trois mécanismes de mesure [Saltzer, 70] :

a) une horloge calendrier de 52 bits, de résolution 1 µs, indépendante des processeurs, donne le temps du système, à partir de l'année 1900 ; elle permet de dater sans ambiguïté tout événement du système ;

b) une horloge, attachée à chaque unité centrale, mesure le temps virtuel du processus, c'est-à-dire le temps qu'il faudrait à la tâche pour s'exécuter si elle n'était jamais interrompue et si elle était seule à utiliser la mémoire ; la différence entre le temps réel d'exécution, donné par l'horloge calendrier, et le temps virtuel est due aux interruptions et aux conflits d'accès à la mémoire ;

c) un canal est affecté à la sortie des mesures ; celui-ci exécute indéfiniment un programme cyclique et sort les données sur un ordinateur DEC PDP-8 pourvu d'un écran de visualisation.

Exemple 3 : ordinateur IBM/370.

L'ordinateur IBM/370 possède un ensemble de 16 classes d'instructions d'appel au moniteur (instruction *MC*) ; un masque de 16 bits permet de mettre en service sélectivement chacune des classes. Une instruction *MC* provoque un déroutement si elle est en service, sinon elle est sans effet. Ces instructions permettent de réaliser aisément diverses mesures : fréquence d'utilisation ou durée d'exécution d'un programme donné, traces, etc., ...

6.334 Mesures programmées

L'implantation de mesures programmées suppose une connaissance approfondie du système. Elle est facilitée par la présence d'interfaces normalisés et de points de passage obligé bien définis. On a donc intérêt à prévoir les mesures dès la conception du système.

Une difficulté sérieuse provient de la perturbation due aux mesures elles-mêmes ; cette interférence n'est pas simple à évaluer, car il ne suffit pas de calculer la durée de la mesure, la perturbation étant également fonction de la fréquence de ces mesures.

Les mesures par échantillonnage sont simples à implanter, bien que l'information puisse ne pas être aisément accessible à tout instant. La période d'échantillonnage ne doit pas être synchronisée avec le comportement dynamique du système étudié, sinon les mesures seront plus ou moins déformées ; une telle coïncidence semble d'ailleurs peu probable pour un système complexe, de comportement très variable. Si par exemple on enregistre périodiquement la valeur du compteur ordinal, on a une mesure de la fréquence d'utilisation des différents programmes du système ; si on analyse le code opératoire de l'instruction courante on obtient les fréquences d'utilisation des instructions du répertoire de la machine.

6.34 UTILISATION DES MESURES

Diverses applications possibles des mesures ont été énumérées en 6.11. Nous en développons ici deux à titre d'exemple : l'évaluation des systèmes et l'amélioration de leurs performances.

6.341 **Evaluation des systèmes**

Le problème de l'évaluation des systèmes se présente sous deux aspects : définition des grandeurs caractérisant les performances et spécification des caractéristiques de la charge du système. Toute estimation de performance doit en effet se référer à une charge bien définie.

Les performances d'un système d'exploitation peuvent s'exprimer, par exemple, en terme de temps de réponse, cette grandeur n'étant généralement pas considérée en valeur absolue, mais compte tenu de la durée du travail demandé. La valeur absolue du temps de réponse prend toute son importance dans le cas des travaux conversationnels où la durée du travail demandé est généralement négligeable vis-à-vis du temps de réponse considéré comme admissible (quelques secondes).

Deux grandeurs caractérisant globalement le fonctionnement d'un système sont le débit des travaux et le taux d'utilisation des diverses ressources. Cette dernière grandeur doit être interprétée avec précaution : le taux d'utilisation d'une ressource donnée dépend non seulement de la demande mais aussi de la gestion de l'ensemble des ressources par le moniteur.

En général, la charge réelle d'un système est variable et non reproductible. On est donc amené à définir des charges artificielles satisfaisant aux conditions suivantes :

— être représentatives, dans un sens à préciser, des conditions d'exploitation réelles du système à évaluer,
— être reproductibles,
— être faciles à paramétrer.

Donnons un aperçu des principales techniques utilisées. Le lecteur pourra se reporter à [Lucas, 71] et [Ferrari, 72] pour une étude plus complète et une bibliographie.

a) Assortiment d'instructions

Un **assortiment d'instructions** (« instruction mix ») est une charge simulée où des instructions de divers types (arithmétiques, comparaison, transfert) figurent chacune avec une fréquence déterminée, fréquence qui peut être fournie par des mesures sur une charge réelle. On essaie ainsi de caractériser un type donné d'application. Pour un assortiment simple, le temps d'exécution peut être calculé à la main.

b) Noyaux

La méthode du **noyau** (« kernel ») est une extension de la méthode précédente, utilisant cette fois un ensemble des sous-programmes.

Les méthodes *a)* et *b)* servent surtout à comparer entre elles des performances d'unités centrales différentes ; ne comportant pas d'opérations d'entrée-sortie, elles ne peuvent servir à évaluer des systèmes.

c) Jeux d'essai

Un **jeu d'essai** d'un système (« benchmark ») est un ensemble de programmes considéré comme représentatif d'un certain type de charge. Les jeux d'essai peuvent comporter des programmes réels ou des programmes fabriqués spécialement pour cet essai. Selon leur composition, ils peuvent être utilisés pour mettre à l'épreuve toutes les parties d'un système, y compris les programmes de service (compilateurs) et les entrées/sorties.

d) Simulateurs de charge

Un **simulateur de charge** est un programme utilisant de façon connue et paramétrée les différentes ressources d'un système. On peut ainsi étudier l'effet, sur les performances d'un système, de divers paramètres de la charge (fraction du temps consacrée au calcul, aux opérations sur des fichiers, distribution et type d'accès des références à la mémoire...). Dans le cas d'un système conversationnel à accès multiple, le simulateur de charge doit en outre simuler le comportement des divers usagers ; il doit donc comporter ou utiliser un dispositif de gestion de processus parallèles. Une difficulté consiste à séparer, dans les mesures, l'effet de la charge simulée et l'effet de la gestion interne du simulateur, qui utilise lui-même les ressources du système. Cette difficulté peut être évitée en faisant exécuter le programme de simulation sur un autre calculateur relié au système mesuré. Cette technique se prête bien à la simulation d'usagers conversationnels et, plus généralement, de tout phénomène dépendant du temps réel.

6.342 **Amélioration des performances**

Un exemple de l'utilisation des mesures pour améliorer les performances d'un programme a été indiqué au paragraphe 6.11 : si l'on sait à quelles parties d'un programme est consacrée la majorité du temps d'exécution, on peut porter son effort sur l'optimisation de ces parties ; si le programme s'exécute dans une mémoire paginée, des mesures peuvent détecter une dispersion des références aux pages, dont la correction pourra améliorer les performances du programme.

Une autre application concerne l'amélioration des performances d'un système d'exploitation, pour lequel des goulots d'étranglement pourront être décelés par des mesures. Un exemple en est donné dans [Cantrell, 68] : les auteurs y signalent un gain de 10 à 50 % sur le débit des travaux d'un système multiprogrammé obtenu à la suite de modifications mineures suggérées par des mesures. Deux exemples de telles modifications sont :

— la réservation pour le moniteur d'un espace de mémoire dépendant de sa taille (variable suivant les configurations), au lieu d'un espace fixé une fois pour toutes ;

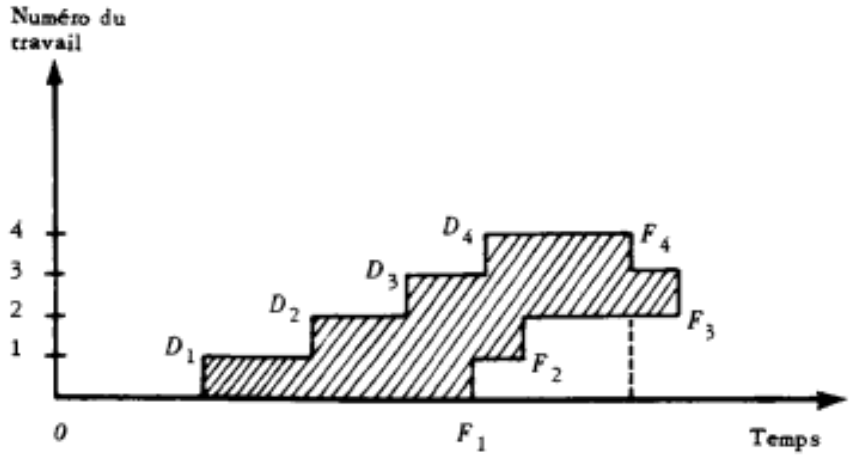
— la modification de la valeur du quantum d'unité centrale pour éviter que des travaux effectuant beaucoup d'entrées-sorties ne soient freinés par des travaux utilisant surtout l'unité centrale. Le gain est obtenu en choisissant un quantum inférieur au temps moyen d'exécution d'une demande d'entrée-sortie.

EXERCICES

1. [1]

Pour la politique SATF de gestion des disques (cf. 6.221), faire le calcul exact du temps d'accès moyen en intégrant sur l'escalier. Vérifier la formule générale pour les cas particuliers $n = 1$ et $n = 2$.

2. [2] *Formule de Little*



On considère un système exécutant des travaux suivant une politique quelconque. Les travaux arrivent suivant une distribution quelconque ; ils commencent aux instants $D_1, D_2, \ldots$ et se terminent aux instants $F_1, F_2, \ldots$ dans un ordre qui n'est pas nécessairement celui des arrivées. L'intervalle $F_i - D_i$ représente le temps de résidence R_i pour le travail i.

a) Soit $N(t)$ le nombre de travaux dans le système à l'instant t ; la zone hachurée, délimitée par la fin F_n du n-ième travail, a pour aire :

$$\int_0^{F_n} N(t)\, \mathrm{d}t$$

A l'instant F_n les $n - 1$ premiers travaux ne sont pas tous nécessairement terminés. Soit $u_i(t)$ la fraction de temps de résidence restant encore au temps t pour le travail i ; montrer que :

$$\int_0^{F_n} N(t)\, \mathrm{d}t = \sum_{i=1}^{n} (1 - u_i(F_n))\, R_i$$

b) Soit $\overline{N}(0, F_n)$ le nombre moyen de travaux dans le système entre les instants 0 et F_n, c'est-à-dire :

$$\overline{N}(0, F_n) = \frac{1}{F_n} \int_0^{F_n} N(t)\, \mathrm{d}t$$

Soit R_{max} le plus grand des R_i, $i \leqslant n$.
Montrer que :

$$\overline{N}(0, F_n)\left[\frac{D_n}{n} + \frac{R_n}{n}\right] \leqslant \frac{1}{n}\sum_i R_i$$

$$\overline{N}(0, F_n)\left[\frac{D_n}{n} + \frac{R_n}{n}\right] \geqslant \frac{1}{n}\sum_i R_i - \frac{N(F_n)\, R_{max}}{n}$$

En déduire que, si le système atteint un équilibre, alors :

$$\overline{N} = \overline{R}\lambda$$

où λ est le nombre moyen d'arrivées par unité de temps.

3. [2] *Modèle de Scherr* (cf. 6.223)

Calculer la probabilité p_i à partir du système d'équations linéaires du régime d'équilibre. Calculer le temps de réponse moyen W.
En utilisant :

$$\sum_{i=0}^{n} (n - i)\, p_i + \sum_{i=0}^{n} i p_i = n$$

montrer que :

$$n - q = (1 - p_0)\frac{R}{T}$$

7

MÉTHODOLOGIE DE CONCEPTION ET DE RÉALISATION

Nous nous intéressons dans ce chapitre aux moyens d'augmenter la **fiabilité** des systèmes, la fiabilité d'un objet étant définie par la probabilité pour que cet objet remplisse pendant un temps donné des fonctions spécifiées dans certaines conditions d'exploitation. Nous avons déjà étudié au chapitre 5 comment on pouvait améliorer cette fiabilité en incluant dans les systèmes des dispositifs de protection destinés à détecter les erreurs en cours de fonctionnement et à limiter leurs conséquences. Nous allons examiner à présent certaines méthodes de conception, de réalisation et de mise au point qui tendent à accroître la probabilité de bon fonctionnement des systèmes produits par ces méthodes.

7.1 INTRODUCTION

A l'heure actuelle, la mise au point des programmes (c'est-à-dire la recherche et la correction des erreurs) tend à consommer la majeure partie du temps et de l'énergie des programmeurs. Au-delà d'un certain degré de complexité, on peut dire qu'un programme ne sort jamais de sa phase de mise au point, c'est-à-dire qu'il y subsiste toujours des erreurs. Nous nous intéressons aux moyens d'améliorer cette situation, en ayant surtout en vue le cas des gros programmes, où le problème se pose avec le plus d'acuité. Nous ne tenterons pas ici de fixer des seuils de taille, de durée de réalisation, de complexité, de

nombre de programmeurs, au-dessus desquels un programme peut être qualifié de « gros », nous contentant de mentionner que tous ces facteurs interviennent.

On peut distinguer schématiquement trois étapes dans le traitement d'un problème sur ordinateur :

1) établissement d'un modèle par formalisation d'un problème concret,

2) description d'un algorithme de traitement dans un langage non ambigu, qui peut ou non être destiné à l'interprétation par une machine ; tous les renseignements nécessaires à la construction de cet algorithme sont contenus dans le modèle établi à la première étape,

3) construction d'un programme exécutable, dont l'interprétation met en œuvre l'algorithme sur une machine donnée.

Nous considérons ici la dernière de ces étapes. Pour que le programme résolve effectivement le problème posé, il doit satisfaire à un certain nombre de propriétés qui définissent son comportement dans des circonstances déterminées. On appelle **spécifications** l'ensemble de ces propriétés. Elles doivent caractériser le comportement du programme de façon complète et, de préférence, simple et non redondante. Le problème de l'établissement des spécifications ne nous intéresse pas ici ; remarquons simplement que le passage du modèle à l'algorithme et de l'algorithme au programme peuvent introduire des distorsions dues aux limitations introduites par le langage ou par son implantation.

Exemple. Les propriétés des opérations arithmétiques sur des nombres représentés dans des mots de longueur finie ne sont pas identiques à celles des opérations usuelles ; ainsi, si $a = b$ est interprété comme $| a - b | \leqslant 2^{-n}$, alors $a = b$ et $b = c$ n'entraînent pas nécessairement $a = c$.

Un programme est dit **valide** s'il répond à ses spécifications. Pour être utilisable, on demande en outre à un programme de posséder certaines propriétés qui influent sur son mode de réalisation :

— respect de contraintes économiques : minimisation d'une fonction de coût à définir par le concepteur, encombrement, date limite d'achèvement, etc.,

— possibilité de compréhension et de modification.

Cette dernière propriété est particulièrement importante. L'expérience montre en effet que les spécifications des programmes évoluent, que des erreurs sont découvertes pendant leur utilisation. Au cours de son existence, tout programme doit être adapté aux modifications de ses conditions d'utilisation. Une condition pour qu'un programme soit aisément modifiable est que l'éventualité d'une modification soit explicitement prévue dans sa construction. Cela peut être réalisé de deux façons :

— en restreignant les possibilités de modification à un choix dans un ensemble fixé une fois pour toutes (assemblage conditionnel),

— en concevant le programme de manière à faciliter le remplacement ou l'addition de parties de programmes (modularité).

Une autre condition pour qu'un programme soit modifiable est qu'il en existe une description permettant de comprendre sa structure et son fonctionnement, à un niveau de détail suffisant. Cette description est souvent donnée sous forme de commentaires mais elle n'est pas nécessairement distincte du texte du programme lui-même.

Compte tenu des contraintes ci-dessus, on cherche donc à réaliser des programmes valides. Le problème de la validité peut être abordé sous deux aspects :

— étant donné un programme existant, établir sa validité vis-à-vis d'un ensemble donné de spécifications,

— étant donné un ensemble de spécifications, construire un programme qui les vérifie.

Pour établir la validité d'un programme existant, on pourrait imaginer de faire exécuter le programme dans les conditions définies par ses spécifications, et de vérifier que les résultats obtenus sont ceux spécifiés. Dans la pratique, une telle démarche est impossible, car le nombre de cas à examiner serait très grand : ainsi, la vérification d'une procédure de racine carrée devrait comprendre l'essai de tous les nombres réels non négatifs représentables ! Suivant la formule de Dijkstra, les tests peuvent servir à détecter la présence d'erreurs, non à prouver leur absence. La validité d'un programme ne peut donc en général se déduire du seul examen de son comportement, elle doit être démontrée à partir des propriétés de la structure du programme. L'état actuel des connaissances est encore loin de permettre une telle démarche ; nous donnons en 7.2 quelques indications sur les techniques utilisées dans les preuves de validité.

En ce qui concerne la construction d'un programme à partir de ses spécifications, on ne dispose pas non plus d'un moyen automatique. Toutefois, on peut utiliser des méthodes qui permettent, à défaut d'une certitude, d'accroître considérablement le degré de confiance que l'on peut avoir dans les programmes construits, tout en rendant plus aisées leur modification et leur mise au point. Ces méthodes sont examinées en 7.3 sous le titre général de « Programmation structurée ». Les langages utilisés pour l'écriture de systèmes et les outils de mise au point sont examinés en 7.4. Enfin, un exemple tiré d'un problème réel illustre en 7.5 l'application des méthodes présentées.

7.2 VALIDITÉ DES PROGRAMMES

Nous donnons ici un aperçu des méthodes utilisées pour démontrer qu'un programme est valide, c'est-à-dire qu'il vérifie ses spécifications. Nous supposons que celles-ci sont données sous la forme de relations, ou **assertions** [Floyd, 67], entre les variables d'entrée et les variables de sortie du programme ; nous supposons en outre que le programme ne comporte pas d'exécution parallèle.

Plus précisément, nous définissons :

— une assertion d'entrée spécifiant les relations existant par hypothèse entre les données du programme,

— une assertion de sortie spécifiant la relation souhaitée entre les variables de sortie et celles d'entrée, ou entre certaines variables de sortie.

Exemple 1. Soit une procédure de calcul de la racine carrée y d'un nombre donné x, avec une précision ε.

Assertion d'entrée : $x \geqslant 0$.

Assertion de sortie : $| y^2 - x | < \varepsilon$.

Exemple 2. Soit à spécifier dans un système à mémoire paginée la procédure *loc* de recherche de la localisation physique x d'une page virtuelle v, la mémoire virtuelle contenant *nmax* pages. A la sortie de la procédure, la variable booléenne *mem* indique si x doit être interprété comme une adresse en mémoire ou sur disque. L'en-tête de cette procédure s'écrit :

procédure loc(v, mem, x) ;
entier v, x ; booléen mem ;

Assertion d'entrée : $0 \leqslant v < nmax$.

Assertion de sortie : *(mem et (v = mémoire(x))) | (¬ mem et (v = disque(x)))*

mémoire et *disque* représentent des tables d'implantation des pages en mémoire principale et sur disque.

Il est naturel de considérer un programme comme une suite d'étapes ; à chaque étape correspondent des assertions d'entrée et de sortie. Prouver la validité d'un programme (ou d'une étape), c'est démontrer :

— que le programme se termine, c'est-à-dire qu'il atteint en un nombre fini d'opérations l'un des points de sortie prévus,

— que les assertions correspondant à ce point de sortie sont vérifiées.

Exemple. Calcul de a^b (J. King, cité dans [Dijkstra, 72]).

Etant donné deux entiers a et b $(a > 0, b \geqslant 0)$, la suite d'instructions ci-après affecte à z la valeur a^b (on suppose déclarés les entiers x, y, z, a, b).

```
x := a ; y := b ; z := 1 ;
tant que y ≠ 0 faire
    début
    si impair(y) alors
        début
        y := y - 1 ;
        z := z * x
        fin ;
    y := y/2 ; x := x * x
    fin ;
```

La fonction booléenne *impair(y)* prend la valeur *vrai* si et seulement si l'entier y est impair.

Remarque. L'interprétation de cette suite d'instructions fait appel à un certain nombre d'axiomes implicitement admis, mais qu'il faudrait en toute rigueur énumérer dans une preuve complète de validité. Ces axiomes traduisent sous forme d'assertions les propriétés des divers opérateurs introduits et de la représentation des nombres. Ainsi, une propriété de l'opérateur $:=$ est traduite par le fait que l'assertion de sortie pour $x := y$ est

$$(x = y) = \underline{vrai}$$

L'assertion d'entrée est :

(1) $$a > 0 \quad \underline{et} \quad b \geqslant 0$$

L'assertion de sortie est :

(2) $$z = a^b$$

La démonstration comporte deux étapes :

1) Désignant par I l'instruction composée suivant *tant que ... faire*, nous allons d'abord démontrer que l'exécution de I laisse invariante la relation :

(3) $$x > 0 \quad \underline{et} \quad y \geqslant 0 \quad \underline{et} \quad a^b = z * x^y$$

D'après (1), cette relation est vraie avant la première exécution de I.

Nous distinguerons deux cas pour la preuve d'invariance :

a) y impair. Posons $y = 2p + 1$ $(p \geqslant 0)$ et désignons par x', y', z' les nouvelles valeurs prises par x, y, z après exécution de I :

$$x' = x^2 ; \quad y' = p ; \quad z' = z * x$$

On a donc

$$z' * x'^{y'} = z * x * (x^2)^p = z * x^{2p+1} = z * x^y$$

et

$$x' > 0 , \quad y' \geqslant 0$$

Si x, y, z vérifient (3), alors x', y', z' vérifient (3) également.

b) y pair. Posons $y = 2p$ $(p \geqslant 0)$. Avec les mêmes notations qu'en *a*), on trouve :

$$x' = x^2 ; \quad y' = p ; \quad z' = z$$

d'où

$$z' * x'^{y'} = z * (x^2)^p = z * x^{2p} = z * x^y$$

et

$$x' > 0 , \quad y' \geqslant 0$$

et l'invariance de (3) est encore assurée.

On remarque que y a toujours une valeur paire avant l'exécution de $y := y/2$, ce qui garantit que y conserve des valeurs entières.

2) Nous allons à présent démontrer que la boucle *tant que* se termine. Si l'on note y_0 la valeur initiale de y et y_i sa valeur après la i-ième exécution de I, alors on a pour tout $i \geqslant 0$, ou bien $y_i = 0$, ou bien $y_{i+1} < y_i$. Les y_i étant non négatifs, il existe donc un entier k tel que $y_k = 0$, ce qui garantit l'arrêt de la boucle après k itérations. La

relation (1), vraie initialement, reste vraie après k exécutions de I. Les valeurs finales de x, y, z vérifient donc :

$$x > 0 \quad \underline{et} \quad y = 0 \quad \underline{et} \quad a^b = z * x^0$$

L'assertion de sortie $z = a^b$ est donc vérifiée.

On pourra trouver dans [Hoare, 71] un autre exemple de preuve de validité, pour un programme plus complexe (tri).

Les méthodes de preuve de validité de programmes sont encore très loin d'être entrées dans la pratique courante, mais on peut prévoir à long terme le développement d'outils (probablement interactifs) permettant la construction de la preuve parallèlement à celle du programme [Floyd, 71 ; Snowdon, 71].

La possibilité d'exécution d'opérations parallèles introduit des difficultés supplémentaires dans les preuves de validité. Pour écrire les assertions, on a besoin d'introduire des axiomes supplémentaires liés au temps (indivisibilité de certaines opérations). Cela peut être fait en spécifiant un mécanisme de synchronisation entre processus, les démonstrations ne s'appliquant alors qu'aux ensembles de processus utilisant ce mécanisme. C'est ainsi que le fonctionnement correct du système producteur-consommateur (cf. 2.5) a pu être démontré en utilisant un théorème sur les sémaphores. On trouvera dans [Dennis, 70] plusieurs tentatives de formalisation des propriétés des processus.

Dans tous les cas présentant quelque intérêt pratique, la complexité de la démonstration est telle qu'elle exclut à l'heure actuelle l'utilisation courante des preuves de validité, notamment dans les systèmes. Nous développons donc dans ce qui suit une voie d'approche moins formelle.

7.3 PROGRAMMATION STRUCTURÉE

Considérons maintenant le problème sous un autre angle : au lieu de prouver qu'un programme donné satisfait certaines spécifications, on se propose de construire un programme satisfaisant des spécifications données.

Une solution idéale serait naturellement de construire automatiquement le programme. Ainsi [Simon, 63] décrit un constructeur automatique de programmes IPL V dont les spécifications sont exprimées dans un sous-ensemble de l'anglais. L'impossibilité où l'on se trouve en général d'exprimer formellement les spécifications d'un problème rend une telle solution inexploitable dans la pratique.

On tente donc, de façon plus pragmatique, de tirer parti de la liberté dont on dispose dans l'écriture du programme pour lui donner une structure propre à faciliter à la fois la construction du programme et la démonstration de sa validité. On désigne sous le terme général de **programmation structurée**

[Dijkstra, 72 ; Mills, 72 ; Wirth, 73], un ensemble de méthodes mises en œuvre pour atteindre ces objectifs. On peut citer en particulier :

— la décomposition des programmes en sous-ensembles pour aboutir à des éléments de complexité acceptable,
— le choix d'une décomposition telle que les interactions entre sous-ensembles soient les plus simples possibles,
— la spécification pour chaque partie de programme d'assertions d'entrée et de sortie.

Dans ce qui suit, nous allons développer de façon plus détaillée les méthodes de décomposition et de spécification des programmes. Nous présenterons d'abord des méthodes applicables aux gros programmes séquentiels, puis nous essaierons de les généraliser aux cas où le parallélisme intervient.

7.31 PROGRAMMES SÉQUENTIELS

Il est généralement admis que la complexité de la réalisation d'un programme (sans chercher à définir précisément cette notion) croît beaucoup plus rapidement que le nombre d'instructions de ce programme. Dans ces conditions, si on parvient à décomposer la réalisation d'un programme d'une part en la réalisation d'un ensemble de parties plus simples et d'autre part, en l'assemblage de ces parties, on aura réduit la complexité globale de l'opération. Nous allons examiner deux méthodes de décomposition.

7.311 **Modules**

La décomposition d'un programme en modules est considérée comme classique depuis la réalisation de l'OS/360 [Mealy, 66]. Un **module** est un « morceau de programme » à plusieurs entrées et plusieurs sorties pouvant réaliser un ensemble de fonctions. La décomposition offre les avantages suivants :

— simplification de la conception : on peut définir d'abord la fonction réalisée par chaque module et ses relations avec les autres modules ; on ne s'intéresse qu'ensuite à la réalisation des différents modules,
— facilité de modification du programme par remplacement de modules : la réalisation de chaque module est indépendante de la réalisation des autres,
— accélération de l'implantation si on peut affecter une équipe à la réalisation de chaque module,
— accélération de la mise au point.

1) *Décomposition en modules*

On ne connaît pas de méthode générale de décomposition des programmes. On doit donc s'appuyer sur l'expérience, c'est-à-dire suivre dans la pratique une démarche essai-échec-nouvelle décomposition. On demande généralement

à une décomposition les qualités suivantes :

— la taille présumée de chaque module doit être assez petite pour que la réalisation d'un module puisse être confiée à une équipe réduite, voire à une seule personne,

— si l'on représente par un graphe les interactions entre modules (les modules étant des sommets, les arcs orientés représentant les appels d'un module par un autre), ce graphe doit être le plus simple possible pour faciliter la démonstration de la validité du système.

Chaque module obtenu par décomposition peut lui-même, s'il est trop complexe, faire l'objet d'une nouvelle décomposition.

2) *Spécifications d'un module. Interface*

Une décomposition en modules doit s'accompagner d'une spécification précise et complète de la façon dont le module se comporte vis-à-vis des autres modules. Une telle spécification est appelée **interface.** Pour conserver les avantages de la décomposition, et en particulier permettre le remplacement d'un module par un module de même interface, il est impératif qu'un module donné ne soit accessible aux autres modules qu'en utilisant son interface. Un programme appelant un module ne doit pas, en particulier, exploiter des renseignements sur la réalisation interne du module qui ne font pas partie de l'interface. Une façon efficace de parvenir à ce but [Parnas, 71] consiste à laisser les programmeurs d'un module dans l'ignorance de la réalisation des autres (cette méthode autoritaire pouvant sans doute être remplacée par une auto-discipline des programmeurs).

Une définition plus formelle des notions de module et d'interface est proposée dans [Parnas, 72]. Un module est considéré comme un dispositif pouvant se trouver dans un certain nombre d'états. Chaque état est défini par les valeurs d'un ensemble de variables d'état. Les changements d'état sont provoqués par l'appel de procédures d'utilisation (avec ou sans paramètres). L'interface est définie comme l'ensemble des variables d'état et des procédures d'utilisation.

Les variables d'état peuvent être consultées, mais ne peuvent être modifiées que par l'appel des procédures d'utilisation ; la donnée de leurs valeurs initiales fait partie de l'interface. Le module peut comporter en outre des procédures et des variables internes, inaccessibles de l'extérieur du module, mais qui peuvent apparaître dans les procédures d'utilisation. Le traitement des cas d'erreur (utilisation incorrecte du module) doit être envisagé. Une solution possible consiste à prévoir lors de la programmation du module tous les cas possibles d'erreur et à appeler dans chaque cas une procédure appropriée extérieure au module. Ces procédures d'erreur doivent être spécifiées au moment de l'appel du module.

Exemple. Module de gestion d'une pile [Parnas, 72].

On définit ici les spécifications d'un module chargé de la gestion d'une pile. L'interface du module est définie comme suit :

Variables	Type	Valeur initiale
sommet *hauteur*	*entier* *entier*	*nil* *0*

Procédures	Paramètres	Effet
empiler(a)	*entier a*	*si hauteur = max* *alors err1 sinon* *début sommet := a ;* *hauteur := hauteur + 1* *fin*
désempiler	—	*si hauteur = 0 alors err2 ;* la séquence *(empiler(a) ; désempiler)* équivaut à une action vide s'il n'y a pas d'appel de *err1*.

max est la hauteur maximale de la pile, *err1*, *err2* sont des procédures d'erreur, *empiler* et *désempiler* sont les fonctions de changement d'état ; les variables d'état sont *sommet* (valeur courante de l'élément en sommet de pile) et *hauteur* (hauteur courante de la pile). La valeur *nil* correspond à une pile vide.

Remarque. La définition implicite de la fonction *désempiler* peut à première vue surprendre : elle a en fait été réduite au strict minimum nécessaire pour éviter toute hypothèse sur le mode de réalisation du module. En effet, une définition explicite de *désempiler* nécessiterait d'expliciter le mécanisme de passage d'un élément de la pile au suivant et au précédent, mécanisme qui doit rester inconnu à l'extérieur du module. Le mode de spécification ici utilisé a deux avantages :

— laisser toute liberté au réalisateur pour la gestion interne de la pile (tableau, liste chaînée, ...)

— éviter que les utilisateurs du module utilisent (voire modifient) les variables du module autrement qu'à travers l'interface.

7.312 Niveaux

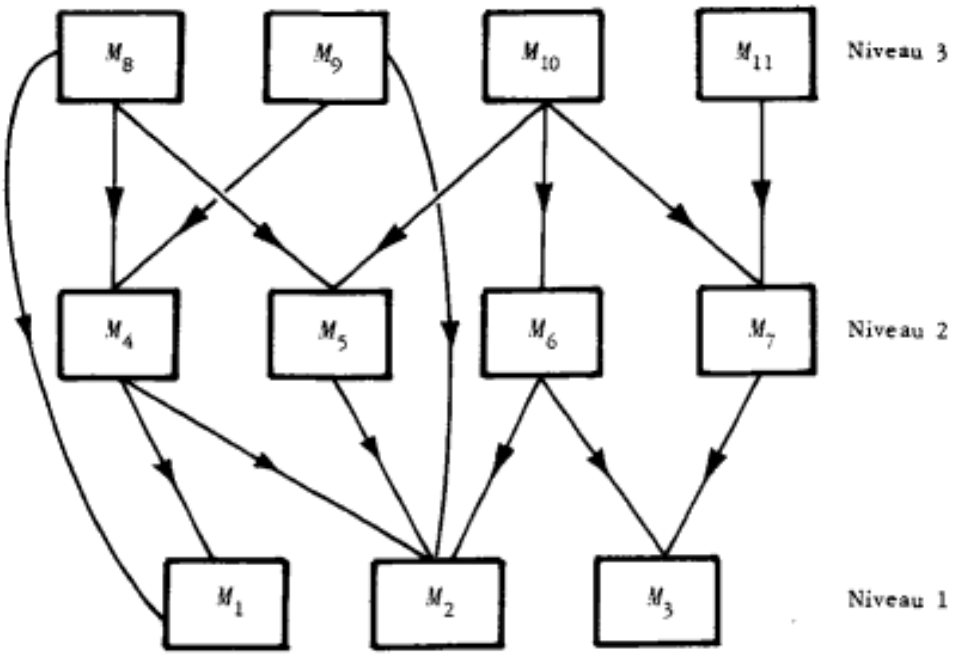
La décomposition en niveaux [Zurcher, 68 ; Wirth, 71b ; Dijkstra, 72] ne diffère pas fondamentalement de la méthode précédente : on se restreint à des découpages en modules tels que le graphe des interactions soit sans circuit.

On peut alors classer les différents modules par niveaux numérotés dans l'ordre des entiers naturels de la façon suivante :

— au niveau *1*, on place les modules qui n'en appellent aucun autre,

— au niveau *2*, les modules qui n'appellent que des modules du niveau *1*,

— au niveau *i*, les modules qui n'appellent que des modules des niveaux *1, 2, ..., i − 1*.



Si on se place à un niveau donné, l'ensemble des fonctions introduites à ce niveau définit un langage ou, ce qui revient au même, une machine. Les modules de niveau inférieur au niveau considéré forment l'interpréteur du langage (ou l'implantation de la machine) ; les modules de niveau supérieur forment un programme écrit dans ce langage (ou s'exécutent sur cette machine). Le système lui-même fournit à ses utilisateurs un langage ou une machine.

On peut utiliser la conception par niveaux pour simuler des ressources virtuelles à partir de ressources physiques. Ces ressources virtuelles peuvent être utilisées à des niveaux supérieurs à celui de leur définition. Les commentaires du 7.311 sur la définition et l'interface des modules restent valables. De plus, maintenant, on dispose de deux approches pour définir le découpage :

— **conception descendante** (« stepwise refinement ») : on part du résultat que l'on souhaite obtenir et on définit par étapes son implantation ; à chaque étape, certaines fonctions sont complètement définies alors que l'implantation des autres reste floue et sera précisée dans une étape ultérieure,

— **conception ascendante** : on part de la machine réelle et on construit des modules qui étendent progressivement son jeu d'instructions et de données pour obtenir une machine de plus en plus proche de celle désirée.

Comme la mise en œuvre d'une seule de ces approches est délicate (et demande en tout cas une grande expérience de la conception de programmes comparables), on procède dans la pratique alternativement de haut en bas et de bas en haut, jusqu'à ce que l'on obtienne une solution convenable ; par contre, la programmation se fait généralement de bas en haut : on met d'abord au point les modules du niveau *1*, puis on intègre ceux du niveau *2* et ainsi de suite.

Exemple. Un programmeur désirant construire un programme d'analyse syntaxique utilisant des piles pourra commencer par étendre son langage en définissant des instructions de gestion de pile comme dans l'exemple du 7.311. Il devra réaliser ces instructions avec les outils dont il dispose.

7.32 PROGRAMMES PARALLÈLES

Une des difficultés des méthodes exposées ci-dessus réside dans la part d'arbitraire de la décomposition en modules. Lorsqu'on admet la possibilité d'exécution parallèle de certaines phases de travail, on introduit un degré de liberté supplémentaire car la décomposition en processus est elle aussi, dans une certaine mesure, arbitraire. La conception et la réalisation sont donc beaucoup plus complexes. Il semble que l'on puisse procéder de la façon suivante : au départ, on fixe la décomposition en processus du système global, ainsi que les primitives de synchronisation que l'on va implanter. On emploie pour cela une approche par niveaux : les niveaux inférieurs doivent fournir des outils adéquats pour la programmation des autres niveaux. On commence ainsi par programmer le niveau le plus bas où l'on implante la gestion des processus et des sémaphores ; puis on ajoute à chaque niveau un ou plusieurs processus de gestion de nouvelles ressources réelles ou virtuelles. On retrouve ainsi les systèmes à « noyau extensible » [Brinch Hansen, 70], dans lesquels le réalisateur du système ne fournit qu'un certain nombre d'outils (synchronisation, gestion des fichiers, ...) qui correspondent aux niveaux inférieurs d'un système complet. L'utilisateur peut compléter le système de façon adaptée à ses besoins propres.

Exemple. Donnons pour terminer un exemple de décomposition en niveaux d'un système réel : le système THE [Dijkstra, 68] ; on rappelle qu'il s'agit d'un système multiprogrammé à nombre fixe de processus, utilisant le sémaphore comme mécanisme de synchronisation.

Niveau 0. On désire tout d'abord multiprogrammer l'unité centrale entre plusieurs processus ; on implante donc au niveau le plus bas les primitives P et V qui réalisent l'allocation de l'unité centrale. Pour assurer une répartition équitable du temps d'unité centrale, on ajoute la programmation d'une horloge permettant de changer périodiquement le processus élu. A partir du niveau *1*, le nombre de processeurs réels est sans importance. Corrélativement, si on change le nombre de processeurs, on n'aura à modifier que le niveau *0*.

Niveau 1. Au niveau *1*, on introduit le processus de gestion du tambour, qui transfère des pages du tambour vers la mémoire centrale à la demande des processus de niveau supérieur. Ce processus est un processus cyclique ; pendant l'exécution d'une entrée-sortie, il est bloqué derrière un sémaphore (le système d'interruption de la machine ELX8 sur laquelle est implanté le THE est très proche du sémaphore). Au-dessus du niveau *1*, on dispose donc d'une mémoire virtuelle.

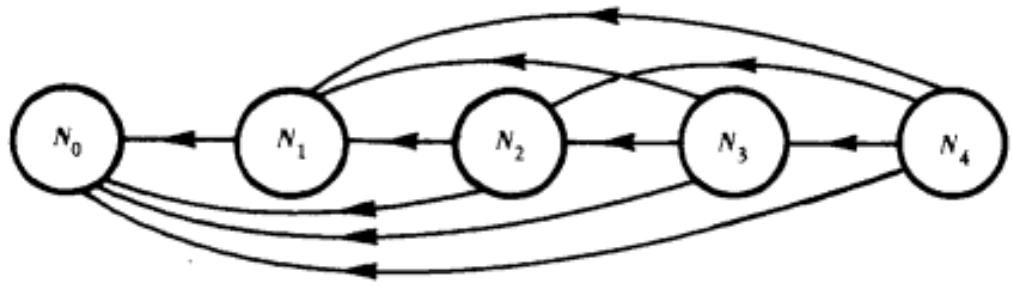
Niveau 2. Au niveau *2* se trouve le processus de gestion de la console de l'opérateur (cf. 7.5). Ce processus, chargé de gérer les conversations entre l'opérateur et les divers processus de niveaux supérieurs, est placé au niveau *2* pour qu'il puisse utiliser la

mémoire virtuelle. Si ce processus avait été placé au niveau *1*, il aurait fallu lui réserver une zone fixe de mémoire centrale et donc condenser les différents messages disponibles. Au-dessus du niveau *2*, les processus peuvent communiquer avec l'opérateur.

Niveau 3. A ce niveau se placent les processus d'entrées-sorties. Les entrées-sorties se font par l'intermédiaire de fichiers sur tambour (auxquels les processus du niveau *4* ont accès par des requêtes au processus de gestion du tambour) ; les processus du niveau *3* sont chargés de remplir (ou de vider) les fichiers tampons à partir (sur) des périphériques. Les demandes d'entrées-sorties sont faites par des processus de niveau supérieur ; pendant l'exécution physique des transferts, les processus d'entrées-sorties sont bloqués derrière un sémaphore. On les place à un niveau supérieur à *2* car ils utilisent la mémoire virtuelle (niveau *1*) et communiquent avec l'opérateur (niveau *2*). Au-dessus du niveau *3*, les processus peuvent exécuter des entrées-sorties.

Niveau 4. A ce niveau se placent les processus des différents usagers. Comme ceux-ci font appel aux processus du niveau *3*, on doit les placer à un niveau supérieur, c'est-à-dire au niveau *4*. Remarquons que les processus des usagers appellent également les processus des niveaux *1* et *2* et utilisent évidemment les mécanismes du niveau *0*.

D'où le graphe des interactions entre niveaux :



7.4 OUTILS D'ÉCRITURE ET DE MISE AU POINT

Nous nous intéressons ici aux modalités pratiques de la réalisation des systèmes. Il s'agit d'un champ d'étude encore peu exploré, et faisant l'objet de nombreuses recherches. Nous nous restreindrons à deux aspects particuliers : les langages d'écriture de systèmes et les outils d'aide à la mise au point.

7.41 LANGAGES D'ÉCRITURE DE SYSTÈMES

Un système doit remplir des fonctions spécifiques, souvent très dépendantes du matériel (gestion des dispositifs câblés de protection, des interruptions, des déroutements, des entrées-sorties, ...), et ces fonctions ne sont pas toujours aisément descriptibles dans un langage de programmation. On peut être ainsi amené à concevoir l'ensemble des outils de construction de système comme un autre système. Ce système de construction peut être considéré comme un sous-ensemble du système à construire (en constituant ainsi un

premier niveau) ou conçu comme entièrement distinct. Dans tous les cas, on doit disposer d'un langage permettant d'exprimer les algorithmes du système à construire. Ce langage doit permettre de construire des programmes dont les caractéristiques ne dépendent pas de celles du système de construction. Nous allons examiner divers aspects des langages de programmation et tenter de dégager des propriétés souhaitables pour un langage destiné à la réalisation de systèmes ; nous passerons ensuite en revue les principaux types de langages utilisés.

7.411 Caractéristiques des langages

Sans chercher à être exhaustifs, nous énumérons ici les caractéristiques qui nous paraissent les plus importantes.

1) *Adéquation*. Le langage doit être adapté au mode de conception et à la nature des algorithmes qu'il exprime : si des processus parallèles interviennent, le langage doit pouvoir exprimer le parallélisme. Ce problème n'est pas résolu actuellement de façon satisfaisante. Le langage doit non seulement pouvoir décrire des propriétés des objets manipulés mais comporter un moyen de garantir à l'exécution que ces propriétés sont satisfaites. Par ailleurs, les programmes de systèmes sont le plus souvent constitués à partir de morceaux séparés. Un langage d'écriture de systèmes doit donc pouvoir décrire aisément les liaisons entre modules ou au minimum permettre d'assembler aisément des morceaux de programmes compilés séparément.

2) *Clarté*. Un langage de programmation est destiné aussi bien à la communication entre hommes qu'à l'interprétation par une machine. De plus, il influence le mode de pensée de ceux qui l'utilisent. La clarté est donc une propriété à rechercher, pour les instructions comme pour les données :

— instructions : le langage ne doit pas permettre de constructions dont l'effet n'est pas immédiatement apparent (effets de bord, par exemple). Les instructions de séquencement (« control statements ») doivent permettre une programmation structurée qui facilite la lecture et réduit le risque d'erreur (utilisation des constructions *tant que* ... *faire*, *répéter* ... *jusqu'à*, ... plutôt que *aller à*),

— données : le langage doit permettre de décrire des données composées (tableaux, classes) et d'associer à toute donnée une procédure d'accès imposant des conditions d'utilisation bien déterminées.

3) *Efficacité*. L'efficacité à l'exécution du programme résultant doit être recherchée, mais non au détriment de la clarté et de la sécurité. Ainsi, on ne doit pas systématiquement exclure diverses opérations à l'exécution (contrôle des indices de tableaux, gestion d'une pile) qui permettent des possibilités étendues de vérification. L'efficacité du compilateur est également un facteur important pour des programmes qui doivent être fréquemment modifiés et recompilés.

7.412 **Classification des langages**

Nous énumérons, par ordre d'éloignement croissant du langage de la machine, les principaux types de langages utilisés pour l'écriture de systèmes.

1) *Langages d'assemblage*

Les opérations décrites dans un langage d'assemblage sont les instructions d'une machine donnée ; aussi ces langages se prêtent-ils assez mal à une programmation claire et structurée en raison de la nécessité de décrire les algorithmes à l'aide d'opérations très élémentaires. De même les structures de données sont rudimentaires et déterminées par le matériel. On peut remédier dans une certaine mesure à ces défauts, par :

— l'utilisation d'un macro-générateur,
— le respect d'une stricte discipline de programmation.

Exemple. On peut implanter des structures de données munies d'un mécanisme d'accès si on s'impose de n'accéder à ces données que par un jeu de macro-instructions. Certains langages facilitent cette discipline par des directives permettant d'adjoindre au répertoire d'instructions un ensemble donné de macro-instructions (directive *SYSTEM* du METASYMBOL sur CII 10070) ou de limiter la portée des identificateurs.

En résumé, les langages d'assemblage permettent d'utiliser toutes les possibilités d'une machine, mais au prix d'un travail souvent fastidieux et sujet aux erreurs. Deux voies sont envisageables pour remédier à ces défauts :

— utilisation de macro-assembleurs, qui ne limitent pas les possibilités de génération,
— utilisation de langages de plus haut niveau, qui restreignent les possibilités de génération.

2) *Langages de type* PL360

Le langage PL360 [Wirth, 68] et les langages analogues conçus pour d'autres machines représentent une tentative de compromis entre la clarté d'expression et la souplesse de génération. Les langages de ce type sont conçus pour une machine donnée. Leurs caractéristiques peuvent se décrire comme suit :

— les instructions de séquencement du type *si* ... *alors*, *tant que* ... *faire* ... sont commandées par des tests sur des expressions booléennes, évitant la programmation détaillée des branchements et comparaisons ;
— les registres sont représentés par des identificateurs, les opérations de chargement, de rangement et les opérations arithmétiques et logiques par des instructions analogues à celles d'ALGOL 60 ;
— des structures de blocs définissent la portée des déclarations ;
— les structures de données sont réduites à des variables simples ou à des tableaux d'éléments adressables sur la machine donnée (mots, octets...) ;
— l'utilisation des sous-programmes, représentés sous forme de procédures, est analogue à celle d'ALGOL 60 ;

— toute instruction du répertoire de la machine peut être engendrée ;
— le code objet engendré par toute instruction est connu précisément.

Cette dernière caractéristique n'est pas en fait utilisée très souvent.

Par rapport aux langages d'assemblage, le gain en clarté apporté par les instructions de séquencement, les instructions d'affectation et l'écriture des expressions est considérable. Il subsiste des inconvénients : nécessité de gérer les registres (risques d'erreur), structures de données rudimentaires, pauvreté du mécanisme d'appel de procédures, absence de tout contrôle à l'exécution. Des extensions sont possibles permettant de remédier dans une certaine mesure à ces défauts :

— possibilité de laisser la gestion des registres (ou de certains registres) à la charge du compilateur,
— mécanisme de définitions de données plus fines (champs dans un mot) et d'accès à ces données.

3) *Langages de haut niveau*

Etant donné les inconvénients signalés plus haut, la tendance actuelle est d'utiliser des langages de niveau plus élevé que les langages d'assemblage ou de type PL360. Les premières tentatives en ce sens ont été l'écriture des systèmes MCP BURROUGHS B5500 (en ALGOL étendu [Burroughs, 64]) et MULTICS (en EPL, sous-ensemble de PL/1 [Corbato, 69b]). On assiste à diverses tentatives de définition d'un langage de haut niveau adapté à l'écriture de systèmes, l'effort portant plus particulièrement sur les points suivants :

— structures de données : définition conjointe des structures de données et de leur algorithme d'accès, y compris la protection de cet accès ; inclusion dans le langage de la notion de pointeur ; possibilité de définir de nouveaux types de variables et d'en créer et détruire dynamiquement des représentants,
— procédures : mécanismes de transmission de paramètres ; spécification du mode d'accès des paramètres.

On pourra trouver dans [Clark, 71b ; Wirth, 71a ; Wulf, 71 ; Berthaud, 72 ; Ichbiah, 72], des descriptions de langages de haut niveau conçus en vue de l'écriture de systèmes.

7.42 OUTILS DE MISE AU POINT

Dans les paragraphes précédents, nous avons passé en revue deux approches en vue d'obtenir des programmes valides : la preuve *a priori* de leur validité et la programmation structurée. Comme les techniques de preuve restent encore inapplicables pratiquement, surtout aux gros programmes, on doit se contenter de vérifier expérimentalement (dans une certaine mesure) que chaque programme répond à ses spécifications ; il ne suffit pas de détecter la présence d'erreurs, il faut encore disposer de renseignements permettant de situer et de corriger rapidement ces erreurs. Quant aux techniques de

programmation structurée, leur application systématique facilite aussi bien la mise au point expérimentale que la démonstration de la validité : dans le cas d'une conception par niveaux, par exemple, on commence par mettre au point les programmes du niveau *0*, puis ceux du niveau *1* en admettant que le niveau *0* est valide, et ainsi de suite.

Pour une étude détaillée des outils de mise au point, le lecteur peut se reporter à [Evans, 66 ; Gaines, 69 ; Rustin, 71]. Nous nous contentons de donner ici quelques idées générales et le principe de leur application à la mise au point des systèmes d'exploitation.

Pour détecter puis corriger les erreurs, le programmeur a besoin de renseignements sur l'exécution de son programme dans les instants précédant et suivant l'apparition de l'erreur ; les outils de mise au point ont pour rôle de lui fournir ces renseignements. Il peut s'agir de la valeur de certaines variables, prélevées lors de phases déterminées de l'exécution, de la suite des instructions exécutées, de mesures de performances (la connaissance des régions de programmes les plus fréquemment exécutées permet d'améliorer les performances globales en reprogrammant soigneusement ces régions).

Un outil de mise au point doit posséder plusieurs propriétés :

1) Toute information imprimée doit l'être sous une forme la plus proche possible de celle du langage source (si le « dump » hexadécimal peut être accepté, faute de mieux, lorsque l'on programme dans le langage de la machine, il est pratiquement inutilisable si on écrit dans un langage de haut niveau).

2) La mise au point doit pouvoir être limitée à certaines variables ou régions.

3) Les performances du programme engendré par le compilateur ne doivent pas être trop dégradées par l'outil de mise au point ; en particulier les procédures non analysées doivent occuper leur taille de mémoire définitive et s'exécuter à la vitesse maximale.

4) La sélection des variables et des régions de programme que l'on désire analyser doit se faire avec le moins possible de modifications du programme source ; l'emploi de cartes de commandes de mise au point et d'options particulières du compilateur paraît approprié. Il est également intéressant de disposer de certaines instructions de mise au point conditionnelle, ne donnant des informations que lorsqu'une certaine condition est réalisée (expression booléenne de variables du programme ou condition anormale : débordement d'un tableau, d'une liste d'instructions *cas*, utilisation d'un pointeur qui ne repère plus rien, ...).

5) La structure du langage de mise au point doit autant que possible être identique à celle du langage source. Les deux langages manipulent en effet les mêmes structures de données. Signalons l'importance du choix des options par défaut pour la sortie des résultats de mise au point : entre une feuille blanche et des centaines de pages imprimées, un compromis satisfaisant n'est pas facile à trouver.

Lors de l'écriture d'un système d'exploitation, la mise au point se fait généralement en deux phases : en utilisant un système préexistant, on écrit et on met au point des modules isolés ; puis on intègre progressivement les différents modules sur la machine pour laquelle on construit le système. On a tout intérêt à munir le système préexistant d'un bon compilateur assorti d'outils de mise au point adéquats et à utiliser ces outils le plus longtemps possible en simulant l'environnement définitif du programme. L'expérience montre que le coût de réalisation de ce simulateur est très largement compensé par le gain de temps à l'intégration ; on limite ainsi les heures de travail au pupitre de la machine, qui gênent d'autres utilisateurs de l'ordinateur. Quant au système en construction, il est fondamental d'y inclure dès le départ un certain nombre d'outils de mise au point, même si ces outils sont rudimentaires aux niveaux inférieurs.

Citons entre autres :

— l'impression sous une forme claire des structures de données globales du système lors de la détection de certaines conditions ou à la demande de l'opérateur (interruption au pupitre). Notons que la mise au point est facilitée si on détecte les erreurs le plus tôt possible ;

Exemple. Dans une procédure d'extraction d'un élément d'une liste, il est judicieux de toujours prévoir le cas où la liste est vide, même si des précautions extérieures semblent garantir que la liste n'est jamais vide lors de l'appel de cette procédure.

— l'enregistrement de l'histoire récente du système : dans un système à processus parallèles, il est difficile, sinon impossible, de reconstituer la suite des différents processus exécutés avant l'apparition d'une erreur ; on évite aisément cette recherche en notant à chaque allocation de processeur le nom du processus allocataire, l'adresse à laquelle on lance l'exécution, et l'heure de l'allocation.

En ce qui concerne la correction des erreurs détectées, il est utile de pouvoir modifier le programme-objet en mémoire centrale pour permettre une expérimentation rapide. Toutefois, il est recommandé de reporter ces modifications dans le texte-source. Des outils permettent de faciliter ces différentes corrections.

— Au niveau du programme-objet : commandes permettant de modifier des mots de la mémoire, ou d'insérer des instructions dans un programme. Ces commandes doivent être placées au premier niveau du système construit.

— Au niveau du programme-source : gestion de fichiers permanents, éditeurs de texte, possibilités de compilation séparée.

7.43 TECHNOLOGIE DE LA PROGRAMMATION

Nous n'aborderons pas ici les problèmes divers que pose la construction de systèmes informatiques en tant que technique industrielle, et qui ont donné lieu à de nombreuses publications. Le lecteur pourra consulter en particulier [Corbato, 68 ; Naur, 69 ; Buxton, 70 ; Weinberg, 71].

7.5 EXEMPLE : RÉALISATION D'UN SYSTÈME D'ENTRÉE-SORTIE

Nous nous proposons dans cet exemple d'illustrer les concepts suivants :

— spécification d'un module et de son interface,

— réalisation d'un module (découpage en processus coopérants, communications entre processus, conception descendante des algorithmes).

Le programme étudié est une version simplifiée du système de gestion des entrées-sorties du système ESOPE [Baudet, 72] dans lequel nous avons supprimé des détails de programmation, mais laissé inchangés le découpage en processus et la synchronisation entre ces processus.

Le système auquel on veut ajouter un système d'entrées-sorties est à accès multiple : à un instant donné, des processus de différents usagers coexistent. Les usagers peuvent conserver des informations (programmes ou données) dans des fichiers sur disques (un fichier est un ensemble d'articles, l'article est l'unité logique d'accès aux informations). On veut construire un programme permettant à un usager d'imprimer le contenu d'un fichier ; après la demande, l'impression a lieu à un instant dépendant uniquement des demandes en attente et l'usager ne reçoit aucun message en fin d'impression. Nous supposons de plus que le système comporte une seule imprimante et que les erreurs de fonctionnement (fin de papier, erreur de transmission) sont gérées par un opérateur qui peut également mettre l'imprimante en service ou hors service.

7.51 SPÉCIFICATION DU MODULE D'ENTRÉE-SORTIE

Avant de spécifier le module d'entrée-sortie, donnons quelques compléments sur la notion d'interface, ainsi qu'un schéma général de conception qui sera utilisé dans le reste du paragraphe.

1) Lorsqu'on spécifie l'interface d'un module, on ne précise que la nature des éléments qui le constituent, ainsi que les relations qui doivent exister entre ces éléments à l'entrée et à la sortie du module. Cette définition est indépendante du mode de réalisation choisi. Selon la complexité du module, on peut se contenter de cette définition ou donner des directives de réalisation.

2) Les éléments de l'interface d'un module peuvent être accessibles soit par tous les autres modules avec les mêmes conventions, soit par tous les modules, mais avec des conventions différentes selon l'appelant, soit par certains modules seulement. Dans les deux derniers cas, la notion d'interface n'est plus suffisante : il faut parler de l'interface entre module appelant et module appelé ou plus simplement d'interface appelant-appelé.

En conséquence, spécifier un module, c'est :

— fournir une description de la machine de base, c'est-à-dire des instructions utilisables pour la programmation du module,

— définir avec précision le mode d'utilisation et éventuellement le contenu du module.

Le mode d'utilisation est défini par l'interface du module, c'est-à-dire l'ensemble des procédures et des données (entrées et résultats) accessibles de l'extérieur du module, ainsi que leurs conventions d'utilisation.

Des indications sur le contenu du module sont données par :

— des directives pour son implantation (nombre de processus par exemple),

— la liste des sous-ensembles du module dont le mode de réalisation est sans importance pour le système global.

Dans l'exemple décrit dans ce paragraphe, nous disposons avant la réalisation du module d'entrée-sortie d'une machine de base dont les instructions sont interprétées soit par câblage, soit par des programmes implantés à des niveaux inférieurs au niveau considéré. Le rôle du nouveau module est de compléter la machine de base par l'introduction d'une instruction

imprimer(nom de fichier)

permettant à un usager de sortir sur l'imprimante le contenu du fichier nommé.

7.511 La machine de base

La machine de base interprète les instructions câblées du calculateur CII 10070 (l'adressage et le jeu d'instructions sont classiques) et les extensions programmées suivantes :

— gestion de processus parallèles synchronisés à l'aide de sémaphores et des primitives P et V,

— utilisation de primitives permettant d'accéder à des fichiers sur disques.

Ces extensions programmées sont utilisables par l'intermédiaire d'instructions d'appel au superviseur (instruction *CAL*).

7.512 Conséquence de l'extension souhaitée

On désire qu'un processus d'un usager puisse demander l'impression d'un fichier. Le module chargé de réaliser l'impression doit exploiter les possibilités de parallélisme entre les différentes unités d'entrée-sortie (disque et imprimante) ; d'autre part, il ne doit pas empêcher le processus demandeur de poursuivre son exécution. Le système doit être protégé contre toutes les erreurs commises par l'usager dans l'appel de la procédure d'impression.

L'extension souhaitée a donc plusieurs conséquences.

1) Pour permettre au processus appelant de poursuivre son exécution, le transfert du fichier doit être réalisé par une famille de processus (dits processus d'entrée-sortie) distincts du processus appelant. Ce dernier n'exécute que la procédure, écrite dans le langage de la machine de base, permettant de faire appel aux processus d'entrée-sortie.

2) Pour tirer parti du parallélisme, nous devons introduire au moins un processus d'entrée-sortie par organe périphérique et mémoriser les demandes de transfert qui ne peuvent être satisfaites immédiatement, c'est-à-dire celles qui surviennent alors qu'un fichier est en cours d'impression.

3) Pour protéger le système contre les erreurs ou les malveillances des usagers, un processus utilisateur ne doit pas avoir accès directement aux variables constituant l'interface du module d'entrée-sortie ; on lui impose de passer par une procédure appelée par l'intermédiaire d'un appel au superviseur.

7.513 L'interface du module d'entrée-sortie

A la suite des considérations précédentes, l'interface du module d'entrée-sortie comprend :

— une zone de mémoire destinée à conserver les demandes d'impression en attente : la boîte aux lettres,

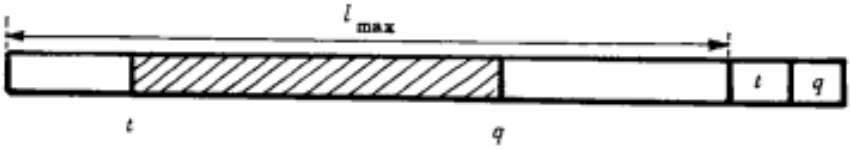
— un moyen de signaler aux processus d'entrée-sortie qu'une demande vient d'être déposée dans la boîte aux lettres.

La structure des fichiers à imprimer n'appartient pas à l'interface du module d'entrée-sortie. Les processus d'entrée-sortie, comme les processus utilisateurs ont accès aux fichiers par l'intermédiaire d'instructions de la machine de base.

1) *La boîte aux lettres*

La boîte aux lettres est un fichier du système, de nom *BAL*. Elle est gérée en tampon circulaire et contient des messages de longueur fixe (6 mots = 24 caractères = longueur maximale d'un nom de fichier).

Sa structure d'ensemble est la suivante :



— les conditions initiales sont $t = q = 0$,

— un nouveau message est placé en queue :

$$BAL\ (q) := message\ ;$$
$$q := q + 6\ \underline{mod}\ lmax\ ;$$

— les messages sont extraits en tête par un processus d'entrée-sortie :

$$message\text{-}reçu := BAL(t)\ ;$$
$$t := t + 6\ \underline{mod}\ lmax\ ;$$

— la boîte aux lettres est pleine si $t = q + 6\ \underline{mod}\ lmax$. Les actions entreprises par un processus demandeur qui trouve la boîte aux lettres pleine sont indépendantes des processus d'entrée-sortie,

— enfin, comme la boîte aux lettres est un fichier accessible en lecture-écriture à plusieurs processus, tout accès doit être inclus dans une section critique ; on utilise pour cela un sémaphore *smutexbal* de valeur initiale *1*.

2) *Synchronisation entre processus demandeur et processus d'entrée-sortie*

On utilise simplement un sémaphore privé *ses* d'un processus d'entrée-sortie. Quand aucune demande de transfert n'est en cours de traitement, ni en attente, ce processus est bloqué derrière le sémaphore *ses*. Après dépôt d'un message dans la boîte aux lettres, le processus demandeur exécute un *V(ses)* qui débloque le processus d'entrée-sortie.

3) *Protection*

L'ensemble des processus demandeurs d'entrée-sortie ne pouvant être considéré comme fiable, il n'est pas question de donner à ces processus un accès direct à l'interface du module d'entrée-sortie. On interdit donc aux processus des usagers d'accéder à la boîte aux lettres et au sémaphore *ses* autrement que par l'intermédiaire d'une procédure particulière dotée d'un pouvoir suffisant. Cette procédure doit alors être appelée par une instruction d'appel au superviseur jouant le rôle d'un guichet d'appel (cf. 5.224). Les actions suivantes y sont effectuées :

— vérification des droits du processus à demander le transfert du fichier,
— modification du pouvoir du processus pour qu'il puisse accéder à la boîte aux lettres,
— accès à la boîte aux lettres ; si celle-ci est pleine, il y a retour au processus demandeur avec un code d'erreur, sinon on note dans la boîte aux lettres le nom du fichier à imprimer et on exécute un *V(ses)*.

Nous ne décrivons pas ici la programmation du guichet d'appel.

En résumé, le système comprend une procédure effectuant l'accès à la boîte aux lettres et l'instruction *V(ses)*. L'instruction *imprimer* du langage de l'utilisateur est traduite à la compilation en un appel au superviseur et tout se passe comme si *message* et *nom de fichier* étaient les paramètres formel et effectif de la procédure.

7.514 Choix laissés au réalisateur du module d'entrée-sortie

Les spécifications précédentes (que nous avons volontairement réduites au minimum) laissent au réalisateur du module un bon nombre d'initiatives. Signalons seulement les principales :

— découpage des modules en processus (dans la mesure où le parallélisme est exploité) et choix des modes de communication entre ces processus,
— traitement des erreurs d'entrée-sortie et communication avec l'opérateur,
— détails de réalisation (mise en page des en-tête de fichiers sur l'imprimante, par exemple).

7.52 CONCEPTION DU MODULE D'ENTRÉE-SORTIE

Nous appliquons à la conception d'un module particulier la même approche que pour un système global, à savoir :

— découpage en modules (les critères de décomposition sont les mêmes que dans le cas général),

— définition globale des interfaces des différents modules,

— conception descendante de chaque module et définition plus fine des interfaces.

7.521 **Décomposition**

Pour exploiter au mieux le parallélisme entre les différentes unités du système, nous introduisons un processus pour chaque organe fonctionnant de manière autonome, c'est-à-dire :

— un processus attaché au disque, ou *FACTEUR*, chargé de la lecture des articles de fichier,

— un processus attaché à l'imprimante, ou *PILOTE*, chargé de l'impression des articles de fichier,

— un processus *SERVANT*, associé à la console de l'opérateur.

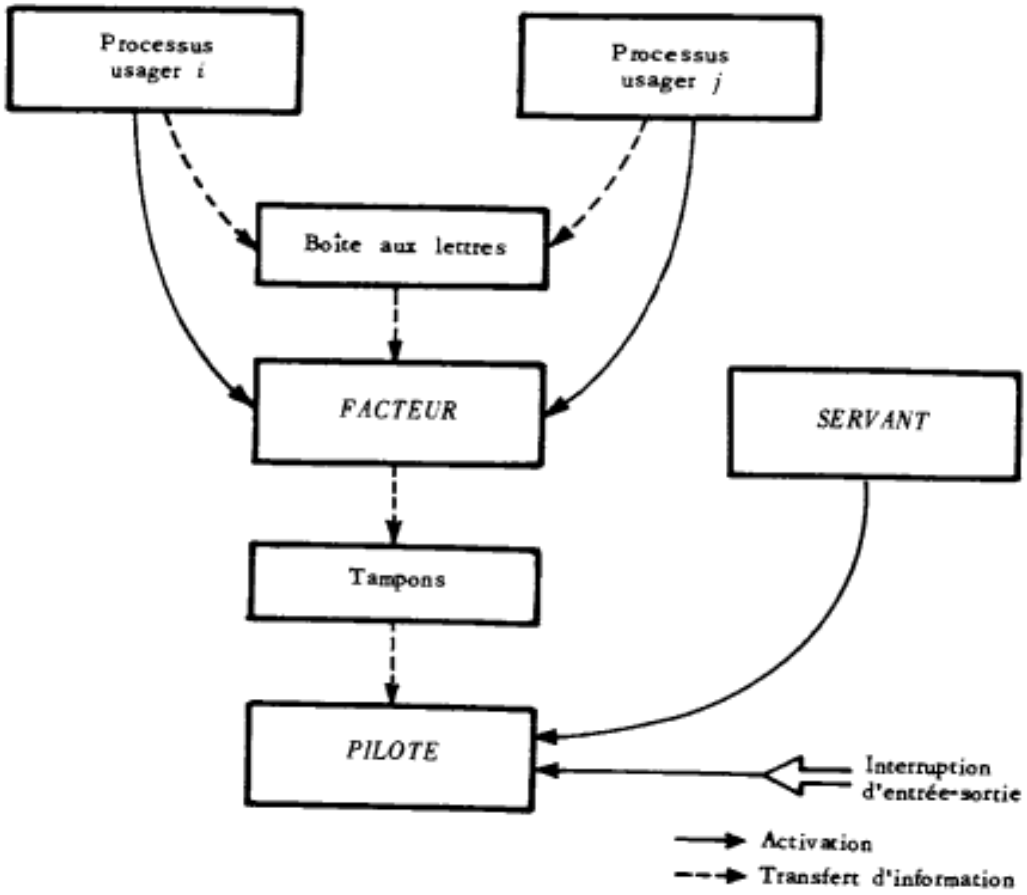


Figure 1. Schéma d'ensemble du système d'entrée-sortie.

Le couple *FACTEUR-PILOTE* coopère à la réalisation des sorties de fichier tant que celles-ci se poursuivent normalement ; s'il se produit des erreurs de transmission, ou si l'opérateur arrête un transfert, le processus *SERVANT* intervient. Enfin, un processus d'un usager n'intervient que pour transmettre la commande de transfert d'un fichier au couple *FACTEUR-PILOTE*.

La boîte aux lettres est représentée par un fichier sur disque ; en conséquence, c'est le processus *FACTEUR* qui est responsable de sa consultation. De même, le sémaphore *ses* introduit au paragraphe précédent est un sémaphore privé du processus *FACTEUR*.

Examinons brièvement deux autres décompositions possibles et les raisons qui nous ont amené à les écarter.

1) Il est possible d'introduire un nombre de processus supérieur au minimum requis pour assurer le fonctionnement simultané des différents organes : par exemple, un processus supplémentaire accède à la boîte aux lettres, puis active le processus *FACTEUR*. Cette solution n'est intéressante que si les usagers peuvent demander des transferts de fichier sur plusieurs organes par l'intermédiaire d'une seule boîte aux lettres.

2) Il est possible d'utiliser le fait que le calculateur CII 10070 est mono-processeur et dispose d'une seule interruption d'entrée-sortie. Dans ces conditions, un processus unique, activé par l'interruption d'entrée-sortie (via un sémaphore), peut être programmé de manière à assurer un fonctionnement simultané du disque et de l'imprimante. Des arguments d'efficacité sont souvent invoqués à l'appui de cette solution qui présente néanmoins des inconvénients majeurs :

— elle revient à programmer sous une forme *ad hoc* des mécanismes de synchronisation qui existent déjà dans la machine de base,

— la mise au point est délicate car les problèmes de synchronisation ne sont pas résolus au moyen de mécanismes généraux dûment éprouvés,

— la clarté due à la décomposition est perdue : le programme final est moins facilement modifiable.

La décomposition en modules que nous retenons correspond à la décomposition en processus. Ce choix est justifié par la simplicité des interfaces et de la programmation de chaque processus.

7.522 **Interfaces**

1) *Interface FACTEUR-PILOTE*

Le *FACTEUR* et le *PILOTE* ne peuvent fonctionner en parallélisme réel que si l'on dispose de plusieurs tampons d'entrée-sortie (on ne peut à un instant donné remplir un tampon à partir du disque et le vider sur l'imprimante).

Soit n le nombre des tampons disponibles.

La technique du producteur-consommateur est bien adaptée au problème de la synchronisation *FACTEUR-PILOTE*. Nous l'employons donc, le *FACTEUR* étant le producteur, le *PILOTE* le consommateur ; *sprod* (valeur initiale *n*) et *scons* (valeur initiale *0*) sont les deux sémaphores utilisés. L'interface entre ces deux processus se réduit donc :

— à l'ensemble des tampons, dont les adresses et les longueurs sont connues des deux processus,
— aux sémaphores *sprod* et *scons*.

2) *Interface du processus SERVANT*

Le processus *SERVANT* est associé à la console de l'opérateur ; il peut être activé de deux façons :

— à la demande de l'opérateur (interruption pupitre), qui désire intervenir sur le déroulement des entrées-sorties,
— à la demande d'un processus quelconque, pour sortir un message sur la console.

Dans le premier cas, on ne désire prendre en compte qu'une seule demande à la fois et un booléen, *itpup*, suffit à mémoriser la demande. Au contraire, dans le second cas, on utilise une boîte aux lettres intermédiaire, *casier*, gérée suivant le schéma producteur-consommateur à l'aide de deux procédures :

extraire-message(m) : prélever un message dans *casier* et affecter sa valeur à *m*,
déposer-message(m) : déposer le message *m* dans *casier*.

Dans ce schéma, le producteur est unique ; la présence de plusieurs périphériques nécessiterait un schéma à producteurs multiples.

Au repos, le *SERVANT* est bloqué derrière son sémaphore privé *sservant* ; toute demande d'intervention doit donc s'accompagner d'un *V(sservant)*. L'interface du processus *SERVANT* comprend donc :

— le sémaphore *sservant*,
— le booléen *itpup* (accessible à l'opérateur seul),
— la procédure *déposer-message*.

7.523 **Conception des différents modules**

1) *FACTEUR et PILOTE*

Une première approche de la programmation du *FACTEUR* et du *PILOTE* est la suivante :

```
FACTEUR : P(ses) ;
          prélever un
          message dans
          la boîte aux lettres ;
          tant que ∃ articles faire

PILOTE : P(scons) ;
         imprimer le tampon ;
         V(sprod) ;
         aller à PILOTE ;
```

```
début
P(sprod) ;
transférer un article
du disque dans une
page tampon ;
V(scons)
fin ;
aller à FACTEUR ;
```

L'exclusion mutuelle à chacun des tampons a été démontrée au chapitre 2 (modèle du producteur et du consommateur).

En ce qui concerne le processus *FACTEUR*, l'opération *transférer un article dans une page tampon* se fait par une instruction de la machine de base ; quant à la ligne *prélever un message dans la boîte aux lettres*, elle peut s'écrire, en admettant que la boîte aux lettres est un fichier d'un seul article :

```
P(smutexbal) ;
transférer l'article boîte aux lettres dans un tampon ;
message-reçu := BAL(t) ;
t := t + 6 mod lmax ;
vider le tampon dans le fichier boîte aux lettres ;
V(smutexbal) ;
```

En ce qui concerne le processus *PILOTE*, précisons la ligne *imprimer le tampon*. Chaque tampon contient un ensemble de lignes à imprimer. On a choisi, pour alléger la description, de ne pas imprimer plus d'une ligne par instruction d'entrée-sortie. En attendant la fin de l'exécution de l'impression, le *PILOTE* se bloque derrière un sémaphore *sit* (valeur initiale *0*) ; le traitement de l'interruption d'entrée-sortie exécute un *V(sit)*.

imprimer le tampon peut alors s'écrire :

```
        n := 0 ;
INCR :  tant que n < nmax faire
            début
            n := n + 1 ;
            lancer la sortie de la ligne n ;
            P(sit) ;
            traitement des erreurs
            fin ;
```

Les opérations *lancer la sortie de la ligne n* et *traitement des erreurs* seront définies plus tard, en liaison avec le processus *SERVANT*.

2) *SERVANT*

Le processus *SERVANT* est un processus cyclique bloqué au repos derrière son sémaphore privé *sservant*. Nous admettrons que les interventions de l'opérateur bénéficient d'une priorité plus grande que les sorties de message.

La procédure associée au processus *SERVANT* a donc la structure suivante :

```
SERVANT :  P(sservant) ;
           si itpup alors
               debut
               lire une ligne sur la console ;
               décoder le message entré ;
               exécuter l'action demandée ;
               itpup := faux
               fin
           sinon
               début
               extraire-message(m) ;
               imprimer le message m
               fin ;
           aller à SERVANT ;
```

Pendant chaque entrée-sortie de message, le *SERVANT* est bloqué derrière un sémaphore *sitservant* (valeur initiale *0*) ; l'interruption de fin d'entrée-sortie sur la console déclenche un *V(sitservant)*.

Nous allons maintenant exposer deux actions exécutées par le *SERVANT*, ainsi que les instructions correspondantes du processus *PILOTE* : le traitement des erreurs d'entrée-sortie et le verrouillage de l'imprimante.

3) *Traitement des erreurs d'entrée-sortie*

Trois classes d'anomalies peuvent se produire :

— passage en mode manuel (appui sur le bouton « manuel » de l'imprimante, fin de papier). Si l'imprimante est en mode manuel, une demande de sortie *(SIO)* est acceptée ; elle doit attendre pour s'exécuter que l'opérateur ait appuyé sur le bouton « automatique » de l'imprimante. Dans ce cas, le *SERVANT* se contente donc de demander à l'opérateur d'intervenir.

— imprimante non opérationnelle (par exemple à la suite d'une panne de secteur). Le lancement d'une entrée-sortie est alors refusé.

— erreur de transfert.

Dans les deux derniers cas, il faut avant de relancer la sortie que l'opérateur appuie sur le bouton « remise à zéro » de l'imprimante. On lui donne alors la possibilité soit de reprendre l'impression de la ligne erronée, soit de poursuivre purement et simplement l'impression du fichier. En attendant l'intervention de l'opérateur, le *PILOTE* se bloque derrière un sémaphore *sop* ; on mémorise ce blocage au moyen d'un booléen *attend-op*. Dans le processus *PILOTE*, la ligne *lancer la sortie de la ligne n* s'écrit donc :

```
SORTIE :  SIO ;
          si manuel alors
              déposer-message (« imprimante en manuel ») ;
```

```
si non opérationnelle alors
début
attend-op := vrai ;
déposer-message (« impr. non opérationnelle ») ;
P(sop) ;
si réponse opérateur = « continuer » alors
aller à INCR sinon aller à SORTIE
fin
```

De même, la ligne *traitement des erreurs* s'écrit :

```
si erreur de transfert alors
début
attend-op := vrai ;
déposer-message (« erreur de transfert ») ;
P(sop) ;
si réponse opérateur = « continuer » alors aller à INCR
sinon aller à SORTIE
fin
```

Quant à l'action du processus *SERVANT*, elle peut s'écrire :

```
si ¬ attend-op alors erreur
sinon
    début
    réponse opérateur := message lu à la console ;
    attend-op := faux ;
    V(sop)
    fin
```

Remarque 1. Le booléen *attend-op*, la chaîne *réponse-opérateur* et le sémaphore *sop* font partie de l'interface *PILOTE-SERVANT*.

Remarque 2. Le sémaphore *sop* est un sémaphore privé du processus *PILOTE* ; les variables *attend-op* et *réponse-opérateur* peuvent être placées indifféremment dans l'un quelconque des deux modules.

4) *Verrouillage de l'imprimante*

On désire parfois arrêter l'imprimante entre deux transferts de fichier, pour permettre certaines opérations d'entretien. On prévoit donc deux commandes utilisables par l'opérateur, le verrouillage et le déverrouillage de l'imprimante.

Une première approche est la suivante :

— on positionne au verrouillage un booléen *verrou*,

— le *FACTEUR* consulte *verrou* au sortir de sa position de repos *P(ses)* ; s'il le trouve *vrai*, il se bloque derrière un sémaphore *sbloc* (valeur initiale *0*),

— au déverrouillage, il faut bien sûr remettre *verrou* à *faux* ; de plus, si le facteur s'est bloqué, il faut exécuter un *V(sbloc)*. On doit donc disposer d'un indicateur supplémentaire *facteur-bloqué* positionné par le facteur avant son blocage, et faire attention aux exclusions mutuelles d'accès à cet indicateur.

Dans une deuxième approche, on désire se dispenser de l'indicateur *facteur-bloqué* et exécuter systématiquement *V(sbloc)* au déverrouillage. Pour cela, on peut faire une analogie avec un passage à niveau : au verrouillage, on baisse la barrière, au déverrouillage on la relève. Quand une voiture trouve la barrière fermée, elle s'arrête et repart quand la barrière est levée. Avec les sémaphores, on obtient les instructions suivantes (*sbloc* a pour valeur initiale *1*) :

```
VERROUILLAGE                    DEVERROUILLAGE
si ¬ verrou alors               si verrou alors
    debut                           debut
    verrou := vrai ;                verrou := faux ;
    P(sbloc)                        V(sbloc)
    fin ;                           fin ;

                FACTEUR
                P(ses) ;
                P(sbloc) ;
                V(sbloc) ;
```

Remarques

— Le sémaphore *sbloc* appartient à l'interface *FACTEUR-SERVANT*.

— Dans le *FACTEUR*, on fait suivre le *P(sbloc)* d'un *V(sbloc)* pour ramener le sémaphore à sa valeur initiale de *1*.

— On peut dans le *FACTEUR* tirer parti de l'indicateur *verrou* pour n'exécuter les primitives sur *sbloc* qu'au moment opportun ; *verrou* appartient alors à l'interface *FACTEUR-SERVANT*.

7.524 Récapitulation

La figure 2 donne un schéma général des relations entre processus dans le système d'entrée-sortie.

Donnons pour terminer la liste des modules et de leurs interfaces :

1) *Module d'entrée-sortie*

— sémaphores *ses*, *smutexbal*,
— boîte aux lettres : fichier de nom *BAL*.

2) *Sous-modules*

a) *FACTEUR*

— sémaphores *sprod*, *scons*, *ses*,
— tampons d'entrée-sortie.

} interface *(FACTEUR-PILOTE)*

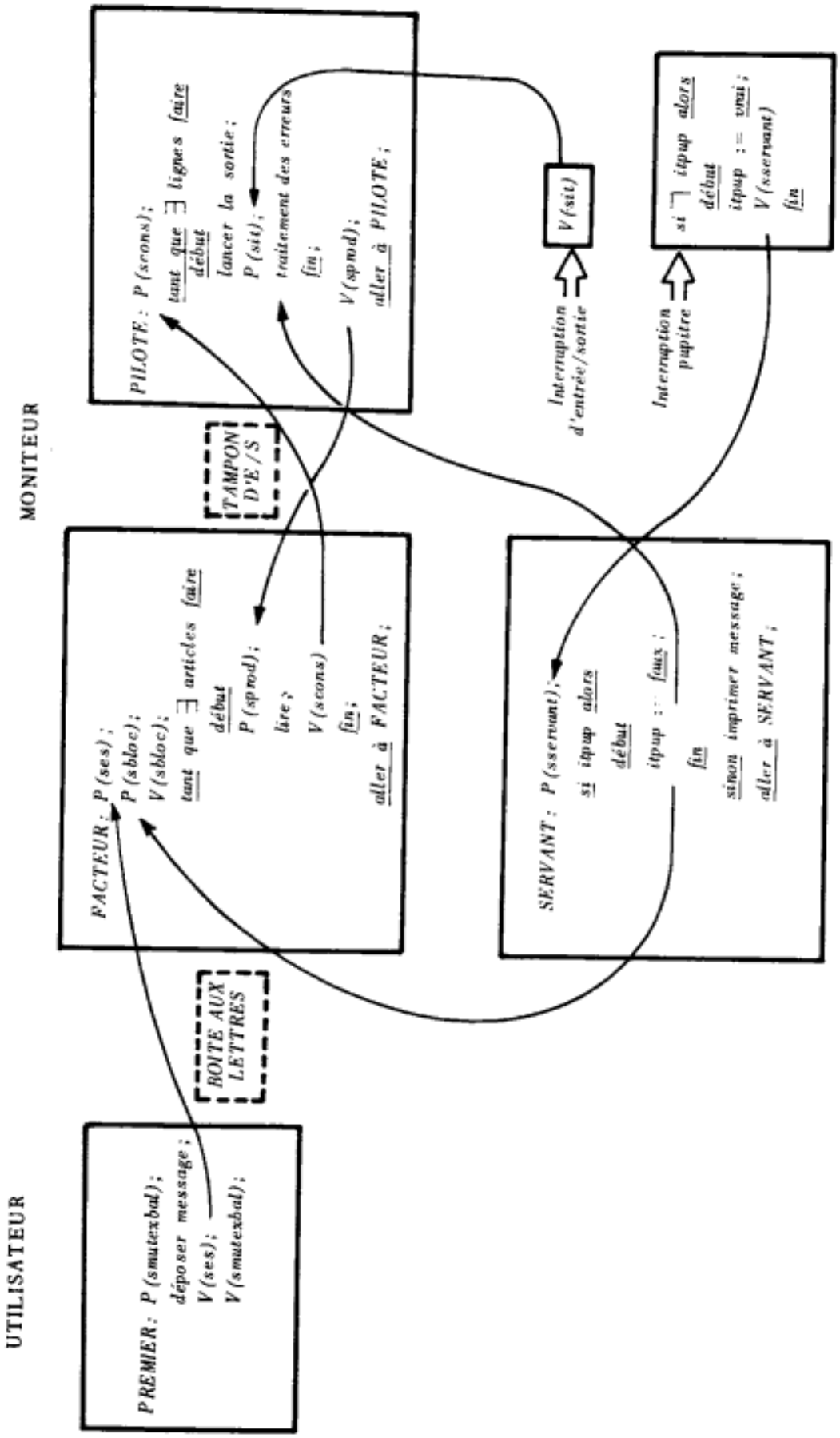


Figure 2. Schéma général du système d'entrée-sortie.

b) *PILOTE*

— sémaphores *sprod, scons,*
— tampons d'entrée-sortie,
} interface *(PILOTE-FACTEUR)*

— sémaphores *sop, sbloc,*
— booléen *attend-op,*
— chaîne *réponse-opérateur.*
} interface *(PILOTE-SERVANT)*

c) *SERVANT*

— sémaphore *sservant,*
— procédure *déposer-message,*
} interface *(PILOTE-SERVANT)*

— booléen *itpup.* interface *(SERVANT-OPERATEUR)*

Toutes les variables de communication entre le *PILOTE* et le *SERVANT* sont définies dans le programme associé au processus *PILOTE*; dans ces conditions, l'extension à un système comportant plusieurs périphériques, donc plusieurs processus *PILOTE*, est immédiate.

EXERCICES

Les quatre exercices qui suivent se rapportent au modèle d'entrée-sortie présenté en 7.5. La terminologie utilisée est celle de 7.5.

1. [1] *Transmission des messages du processus SERVANT* (cf. 7.522)

Ecrire les procédures utilisées pour la communication avec le processus *SERVANT*. On prévoira le cas de producteurs multiples.

déposer-message place un message dans la boîte aux lettres (*casier*) du *SERVANT*, *extraire-message* est appelée par le *SERVANT* pour prélever un message dans *casier*.

2. [2] *Verrouillage de l'imprimante*

a) En utilisant la première des deux méthodes exposées en 7.523*d*), écrire :

— les commandes de verrouillage et de déverrouillage de l'imprimante,
— la phase initiale du processus *FACTEUR* (test après activation).

b) On suppose maintenant que le système gère deux imprimantes.
Modifier la séquence de verrouillage de façon que la mise hors service d'une imprimante n'empêche pas l'impression d'un fichier sur l'autre.

3. [2] *Arrêt de l'entrée-sortie en cours*

Implanter une commande permettant à l'opérateur d'arrêter un transfert en cours ; si une demande de transfert est en attente, elle doit alors être prise en compte immédiatement.

4. [2] *Signal de fin d'impression*

On désire que l'usager soit prévenu à la fin de l'impression (et puisse donc attendre celle-ci). Modifier les programmes en conséquence.

SOLUTIONS DES EXERCICES

CHAPITRE 2

Exercice 1

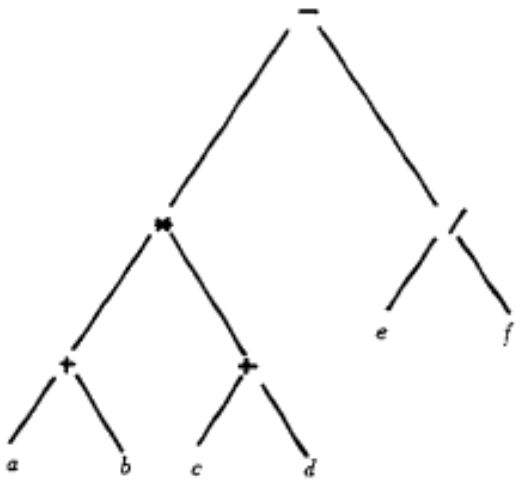
1) $S(p_1, S(p_2, S(p_3, p_4)))$.

2) $P(p_1, P(p_2, P(p_3, p_4)))$.

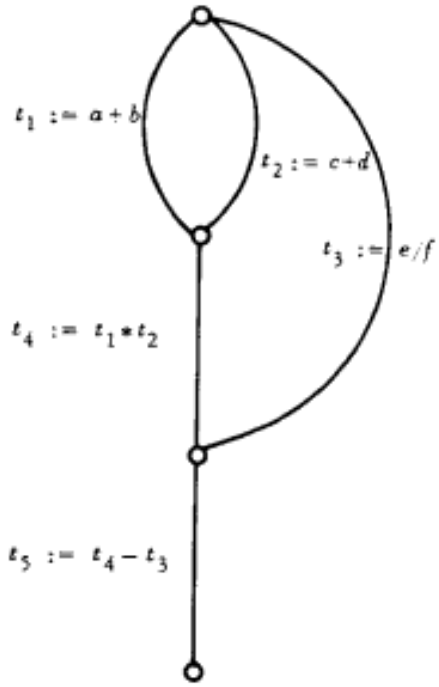
3) $S(p_1, S(P(p_2, P(S(p_3, P(p_4, p_5)), p_6)), P(p_7, p_8)))$.

4) Description impossible.

Exercice 2



1) Structure d'arbre



2) Graphe des processus

3) $S(P(S(P(t_1, t_2), t_4), t_3), t_5)$.

Exercice 3

1) Cas de deux processus.

1) Une seule variable booléenne commune. Pour l'écriture du programme de p_i on désigne « l'autre processus » par p_j, c'est-à-dire $j = 1$ si $i = 2$ et $j = 2$ si $i = 1$ ($j = 3 - i$).

```
contexte commun : booléen c ;
                  c := faux ;
processus pi :    début                          (i = 1, 2)
          Ai :    si c alors aller à Ai ;
                  c := vrai ;
                  section critique i ;
                  c := faux ;
                  reste du programme i ;
                  aller à Ai
                  fin ;
```

Supposons que $c = faux$. Si chaque processus teste c avant que l'autre ne lui ait affecté la valeur *vrai*, les deux processus s'engagent dans leur section critique. L'exclusion mutuelle n'est donc pas assurée.

2) Une seule variable commune, fonctionnant en bascule.

```
contexte commun : entier t ;
                  t := 1 ;
processus pi :    début                          (i = 1, 2)
          Ai :    si t = j alors aller à Ai ;
                  section critique i ;
                  t := j ;
                  reste du programme i ;
                  aller à Ai
                  fin ;
```

Le processus p_i ne peut entrer dans sa section critique que si $t = i$; l'exclusion mutuelle est donc garantie par l'indivisibilité de l'opération d'accès à t. Toutefois, la modification de t à chaque fin de section critique impose un fonctionnement en bascule des deux processus, incompatible avec la propriété d'indépendance (condition *c*) du 2.31) de la solution. En particulier, l'arrêt de p_i dans la partie *reste du programme i* empêche p_j d'exécuter plus d'une fois sa section critique.

3) Deux variables communes.
On pose encore $j = 3 - i$. Une solution conforme aux conventions de l'énoncé peut s'écrire :

contexte commun :

```
                booléen tableau c[1 : 2];
                c[1] := c[2] := faux;
processus p_i :  début                                  (i = 1, 2)
        Ai :    si c[j] alors aller à Ai;
                c[i] := vrai;
                section critique i;
                c[i] := faux;
                reste du programme i;
                aller à Ai
                fin;
```

On a traduit ici le fait qu'un processus p_i ne peut entrer en section critique que si $\neg\, c[j]$. Considérons toutefois la séquence suivante :

1) p_i consulte $c[j]$ et trouve *faux* ;
2) p_j consulte $c[i]$ et trouve *faux* ;
3) p_j fait $c[j] :=$ *vrai* et entre dans sa section critique ;
4) p_i fait $c[i] :=$ *vrai* et entre dans sa section critique.

L'exclusion mutuelle n'est donc pas garantie, et la solution est inacceptable. On peut songer à modifier le programme de façon à faire l'affectation de $c[i]$ avant le test de $c[j]$, remplaçant les deux instructions suivant *Ai* par :

```
Ai : c[i] := vrai;
     si c[j] alors aller à Ai;
```

L'exclusion mutuelle est cette fois garantie : en effet, si p_i trouve $c[j] =$ *faux*, p_j ne se trouve pas en section critique, et ne peut y entrer puisque $c[i] =$ *vrai* au moment du test de $c[j]$. Mais la condition *b*) n'est maintenant plus vérifiée. Soit en effet la séquence suivante :

1) p_i exécute l'instruction étiquetée *Ai* : $c[i] :=$ *vrai* ;
2) p_j exécute l'instruction étiquetée *Aj* : $c[j] :=$ *vrai* ;
3) p_j consulte $c[i]$, trouve *vrai* et réexécute *Aj* ;
4) p_i consulte $c[i]$, trouve *vrai* et réexécute *Ai*.

Les deux processus sont désormais engagés dans une boucle infinie. On pourrait encore songer à améliorer la dernière solution en remettant temporairement $c[i]$ à *faux* à l'intérieur de la boucle d'attente de p_i pour laisser passer p_j :

```
Ai : c[i] := vrai;
     si c[j] alors
          début
          c[i] := faux;
          aller à Ai
          fin;
```

Le lecteur vérifiera que la possibilité d'attente infinie subsiste, en faisant exécuter les deux processus à la même vitesse, avec une instruction de retard.

4) Trois variables communes.

On pose toujours $j = 3 - i$.

```
contexte commun : entier t ;
                  booléen tableau c[1 : 2] ;
                  t := 1 ; c[1] := c[2] := faux ;

processus p_i :   début                                   (i = 1, 2)
            Ai :  c[i] := vrai ;
            Li :  si c[j] alors
                      début
                      si t = i alors aller à Li ;
                      c[i] := faux ;
            Bi :      si t = j alors aller à Bi ;
                      aller à Ai
                      fin ;
                  section critique i ;
                  t := j ; c[i] := faux ;
                  reste du programme i ;
                  aller à Ai
                  fin ;
```

La solution précédente est due à Dekker [Dijkstra, 67]. Montrons qu'elle satisfait aux conditions requises. Nous distinguons deux cas.

1) Il n'y a pas conflit d'accès à la section critique, c'est-à-dire que l'un des processus (soit p_i) exécute *Li* alors que $c[j] = faux$; p_i entre alors en section critique ; comme $c[i]$ a alors la valeur *vrai* et la gardera tant que p_i sera en section critique, p_j ne pourra entrer en section critique avant que p_i en soit sorti.

2) Il y a conflit d'accès, c'est-à-dire que l'exécution des instructions *Ai*, *Li* est entrelacée avec celle de *Aj*, *Lj*. On utilise alors le fait que la variable t n'est pas modifiée par l'exécution des instructions *Ai*, *Li*. Si $t = i$, alors p_j entrera dans la boucle étiquetée *Bj* après avoir remis $c[j]$ à *faux* : cela permet donc l'entrée en section critique de p_i, qui était engagé dans la boucle *Li*. A la fin de sa section critique, p_i remet t à la valeur j, ce qui libère p_j de la boucle *Bj*, lui permettant de tenter à nouveau l'entrée en section critique par *Aj*. On évite donc aussi bien l'attente infinie que l'entrée simultanée puisque le processus p_i et lui seul entre en section critique en cas de conflit.

2) Cas de n processus.

La solution ci-après a été publiée dans [Dijkstra, 65].

contexte commun : *entier* t ;
booléen tableau $b[0 : n], c[0 : n]$;
pour $t := 0$ *pas* 1 *jusqu'à* n *faire*
$b[t] := c[t] :=$ *faux* ;
$t := 0$;

processus p_i : *début entier* j ; $(i = 1, 2, \ldots, n)$
Ai : $b[i] :=$ *vrai* ;
Li : *si* $t \neq i$ *alors*
début
$c[i] :=$ *faux* ;
si $\neg\, b[t]$ *alors* $t := i$;
aller à Li
fin ;
$c[i] :=$ *vrai* ;
pour $j := 1$ *pas* 1 *jusqu'à* n *faire*
début
si $j \neq i$ *et* $c[j]$ *alors aller à* Li
fin ;
section critique i ;
$t := 0$; $c[i] := b[i] :=$ *faux* ;
reste du programme i ;
aller à Ai
fin ;

On remarquera que la variable t peut prendre la valeur 0. Cela est dû au souci de conserver une solution symétrique ; si en effet t n'était pas remis à 0 à la fin de la section critique d'un processus p_k, celui-ci resterait privilégié en cas de conflit lors de sa prochaine entrée en section critique ; les variables $b[0]$ et $c[0]$, initialisées à *faux*, ne sont jamais modifiées.

Le lecteur pourra consulter [Dijkstra, 65] pour l'explication de cette solution, ainsi que [Knuth, 66] pour une solution modifiée, garantissant que tout processus demandant à entrer en section critique y parvient dans un temps fini (on notera que cette condition est plus contraignante que la condition *b*) du 2.31, qui spécifie simplement que tout conflit d'accès est résolu en un temps fini).

Exercice 4

Cas a

```
                    sémaphore s = 0 ;
Processus 1         Processus 2         Processus 3
    .                   .                   .
    .                   .                   .
    .                   .                   .
  P(s) ;              V(s) ;              V(s) ;
  P(s) ;                .                   .
    .                   .                   .
    .                   .                   .
    .
```

Cas b

```
                    sémaphore s = 0 ;
Processus 1         Processus 2         Processus 3
    .                   .                   .
    .                   .                   .
    .                   .                   .
  P(s) ;              V(s) ;              V(s) ;
    .                   .                   .
    .                   .                   .
    .                   .                   .
```

Exercice 5

```
sémaphore m = 1, t = 0 ;
booléen masque = faux ;
```

Matériel :

arrivée du signal d'interruption :

```
P(m) ;
V(t) ;
V(m) ;
```

masquage de l'interruption :

```
si ¬ masque alors
    début
    masque := vrai ;
    P(m)
    fin
```

démasquage de l'interruption :

```
si masque alors
  début
  masque := faux ;
  V(m)
  fin
```

Cette solution suppose que le masquage et le démasquage ne peuvent se faire en parallèle.

Processus cyclique de traitement de l'interruption :

```
A : P(t) ;
    procédure de traitement ;
    aller à A ;
```

Exercice 6
[Courtois, 71]

Cas 1

Solution *a*) (utilisation de sémaphores privés).
Procédures de lecture et d'écriture utilisées par le processus *i* :

DEMANDE DE LECTURE

```
début
P(mutex) ;
si ¬ e alors
    début
    nl := nl+1 ;
    V(spriv[i])
    fin
sinon ajouter(i, filelect) ;
V(mutex) ;
P(spriv[i])
fin ;
```

DEMANDE D'ECRITURE

```
début
P(mutex) ;
si (¬ e) et (nl = 0) alors
    début
    e := vrai ;
    V(spriv[i])
    fin
sinon ajouter(i, filered) ;
V(mutex) ;
P(spriv[i])
fin ;
```

FIN DE LECTURE

```
début
entier j ;
P(mutex) ;
nl := nl−1 ;
si (nl = 0) et (compte de filered ¬ = 0) alors
    début
    ôter (j, filered) ;
    e := vrai ;
    V(spriv[j])
    fin ;
V(mutex)
fin ;
```

FIN D'ECRITURE

```
début
entier j ;
P(mutex) ;
e := faux ;
```

```
si (compte de filelect ¬ = 0) alors
    répéter
        début
        nl := nl+1;
        ôter(j, filelect);
        V(spriv[j])
        fin
    jusqu'à compte de filelect = 0
sinon si (compte de filered ¬ = 0) alors
    début
    e := vrai;
    ôter(j, filered);
    V(spriv[j])
    fin
fin;
```

Solution *b*)

```
entier nl = 0;
sémaphore mutex1 = 1, mutex2 = 1, w = 1;
```

LECTEURS	*REDACTEURS*
P(mutex1)	*P(mutex2);*
nl := nl+1;	*P(w);*
si nl = 1 alors P(w);	...
V(mutex1);	*écrire;*
...	...
lire;	*V(w);*
...	*V(mutex2);*
P(mutex1);	
nl := nl−1;	
si nl = 0 alors V(w);	
V(mutex1);	

Remarques

1) *w* est un sémaphore d'exclusion mutuelle pour tous les rédacteurs. Il sert au premier lecteur qui occupe le fichier et au dernier qui le libère.

2) *mutex1* est un sémaphore d'exclusion mutuelle pour les lecteurs seulement. Il protège le compte de lecteurs *nl*. Si un lecteur est bloqué par *w*, tous les autres lecteurs le seront par *mutex1*.

3) *mutex2* permet de garantir la priorité des lecteurs. Sans *mutex2* on pourrait avoir plusieurs rédacteurs en attente sur *w*, avant le premier lecteur bloqué par *w*.

Cas 2

```
entier nl = 0 ;
sémaphore mutex = 1, w = 1 ;
```

LECTEURS	*REDACTEURS*
P(mutex) ;	
nl := nl + 1 ;	
si nl = 1 alors P(w) ;	*P(w) ;*
V(mutex) ;	...
...	*écriture*
lecture	...
...	*V(w) ;*
P(mutex) ;	
nl := nl − 1 ;	
si nl = 0 alors V(w) ;	
V(mutex) ;	

Remarques

1) *w* est un sémaphore d'exclusion mutuelle pour tous les rédacteurs. Il sert aussi au premier lecteur qui occupe la ressource et au dernier qui la libère.

2) *mutex* est un sémaphore d'exclusion mutuelle pour les lecteurs seuls ; il protège le compte de lecteurs, *nl*.

Cas 3

```
entier nl = 0 ;
sémaphore mutex = 1, r = 1, w = 1 ;
```

LECTEURS	*REDACTEURS*
P(r) ;	
P(mutex) ;	
nl := nl + 1 ;	
si nl = 1 alors P(w) ;	
V(mutex) ;	*P(r) ;*
V(r) ;	*P(w) ;*
...	...
lecture ;	*écriture ;*
...	...
P(mutex) ;	*V(w) ;*
nl := nl − 1 ;	*V(r) ;*
si nl = 0 alors V(w) ;	
V(mutex) ;	

Remarques

1) *r* bloque tous les lecteurs nouveaux dès qu'un rédacteur est arrivé.

2) *w* bloque toute écriture tant que les lectures ne sont pas finies. Il y a au plus un rédacteur bloqué dans la file associée à *w*.

3) Dès qu'un lecteur passe ou est réveillé, il incrémente le compte des lecteurs, bloque éventuellement la ressource et réveille le processus suivant de la file associée à *r*.

4) Comme l'ordre *P(r)*, *P(w)* est respecté dans chaque processus. Il ne peut y avoir de blocage par étreinte fatale.

5) Pour rendre la solution plus symétrique, le rédacteur peut exécuter la primitive *V(r)* immédiatement après *P(w)*.

6) Cette solution n'assure de service selon l'ordre d'arrivée (FIFO) que si toutes les files sont gérées selon ce principe.

Cas 4

```
entier nl = 0, nr = 0 ;
sémaphore mutex1 = 1, mutex2 = 1, mutex3 = 1,
          w = 1, r = 1 ;
```

LECTEURS	*REDACTEURS*
P(mutex3) ;	*P(mutex2) ;*
P(r) ;	*nr := nr + 1 ;*
P(mutex1) ;	*si nr = 1 alors P(r) ;*
nl := nl + 1 ;	*V(mutex2) ;*
si nl = 1 alors P(w) ;	*P(w) ;*
V(mutex1) ;	
V(r) ;	
V(mutex3) ;	
...	...
lecture ;	*écriture ;*
...	...
P(mutex1) ;	*V(w) ;*
nl := nl − 1 ;	*P(mutex2) ;*
si nl = 0 alors V(w) ;	*nr := nr − 1 ;*
V(mutex1) ;	*si nr = 0 alors V(r) ;*
	V(mutex2) ;

Remarques

1) *mutex1* et *w* jouent le même rôle que *mutex* et *w* dans le cas 1,

2) *r* est utilisé par les rédacteurs pour se réserver l'accès à la ressource, tout comme *w* par les lecteurs dans le cas 1. Le premier rédacteur, en faisant *P(r)*, bloque les lecteurs avant leur entrée dans la section critique contrôlée

par *mutex1*. Il est important qu'il le fasse en dehors de cette section car *mutex1* contrôle une autre section critique dans laquelle les lecteurs libèrent la ressource.

3) Sans *mutex3*, on pourrait trouver un rédacteur et plusieurs lecteurs dans la file d'attente de *r*. On n'assurerait pas la priorité absolue aux rédacteurs ; *mutex3* garantit qu'un lecteur au plus utilise *r*. Donc la file d'attente de *r* ne peut plus contenir qu'un lecteur ou qu'un rédacteur (pendant qu'un lecteur incrémente *nl*).

Exercice 7

Cas 1. Le carrefour peut contenir une voiture.

```
sémaphore mutex1 = 1, mutex2 = 1, feu1 = 1, feu2 = 0 ;

TRAVERSEE 1          CHANGEMENT                          TRAVERSEE 2
P(mutex1) ;            booléen a = vrai ;                P(mutex2) ;
    P(feu1) ;        C : attendre(m) ;                       P(feu2) ;
    traversée          commentaire m est le délai d'attente ;  traversée
    du carrefour ;     si a alors                              du carrefour ;
    V(feu1) ;              début                               V(feu2) ;
V(mutex1) ;                P(feu1) ;                       V(mutex2) ;
                           V(feu2) ;
                           a := faux
                           fin
                       sinon
                           début
                           P(feu2) ;
                           V(feu1) ;
                           a := vrai
                           fin ;
                       aller à C ;
```

Remarques.

1) *feu1* et *feu2* règlent chaque file. Ils évitent aussi toute modification de feu tant qu'une voiture est dans le carrefour.

2) *mutex1* et *mutex2* ont pour fonction de prévenir toute coalition de voitures voulant bloquer le feu. En conséquence il y a au plus une voiture bloquée par *feu1* ou *feu2*.

3) *m* indique un délai d'attente.

Cas 2. Le carrefour peut contenir *k* voitures.

```
sémaphore mutex1 = k, mutex2 = k, feu1 = 1, feu2 = 0 ;
sémaphore mutex = 1, w = 1 ;
entier n = 0 ;
```

```
TRAVERSEE 1
P(mutex1) ;
    P(feu1) ;
        P(mutex) ;
        n := n + 1 ;
        si n = 1 alors P(w) ;
        V(mutex) ;
    V(feu1) ;

traversée du
carrefour ;

    P(mutex) ;
    n := n - 1 ;
    si n = 0 alors V(w) ;
    V(mutex) ;
V(mutex1) ;
```

```
CHANGEMENT
  booléen a = vrai ;
C : attendre(m) ;
  si a alors
      début
      P(feu1) ;
      P(w) ;
      V(feu2) ;
      V(w) ;
      a := faux
      fin
  sinon
      début
      P(feu2) ;
      P(w) ;
      V(feu1) ;
      V(w) ;
      a := vrai
      fin ;
  aller à C ;
```

```
TRAVERSEE 2
P(mutex2) ;
    P(feu2) ;
        P(mutex) ;
        n := n + 1 ;
        si n = 1 alors P(w) ;
        V(mutex) ;
    V(feu2) ;

traversée du
carrefour ;

    P(mutex) ;
    n := n - 1 ;
    si n = 0 alors V(w) ;
    V(mutex) ;
V(mutex2) ;
```

1) *mutex1* et *mutex2* ne laissent passer que k voitures au plus.

2) *feu1* et *feu2* servent à arrêter les voitures lorsque les feux vont changer. Le processus *CHANGEMENT* n'est jamais bloqué longtemps derrière *feu1* (ou *feu2*) car s'il est bloqué aucune voiture ne peut se bloquer par $P(w)$.

3) w sert à bloquer le changement de feu tant qu'il y a des voitures dans le carrefour. A cause des deux primitives $P(feu)$, $P(w)$, seul le processus *CHANGEMENT* peut être bloqué par w.

Exercice 8 [Vantilborgh, 72]

1)

```
sémaphore w = n ;

LECTEURS          REDACTEURS
P(1, w) ;         P(n, w) ;
lecture ;         écriture ;
V(1, w) ;         V(n, w) ;
    :                 :
```

2) Posons :

ns : valeur maximum prise par le rang n

bloque[i] : nombre de processus de rang i, bloqués dans la file d'attente du sémaphore s.

Il vient :

```
sémaphore mutexs = 1 ; sémaphore tableau sempriv[1 : ns] = 0[1 : ns] ;
entier n, s, i, j ; entier tableau bloque[1 : ns] = 0[1 : ns] ;

programme de P(n, s) : P(mutexs) ;
                       si e(s) ⩾ n alors début e(s) := e(s) − n ; V(mutexs) fin
                                  sinon début
                                        bloque[n] := bloque[n] + 1 ;
                                        V(mutexs) ;
                                        P(sempriv[n])
                                        fin ;

programme de V(n, s) : P(mutexs) ;
                       e(s) := e(s) + n ;
                       pour i := 1 pas 1 jusqu'à ns faire
                            pour j := bloque[i] pas − 1 jusqu'à 1 faire
                                 si e(s) ⩾ i alors début
                                                   bloque[i] := bloque[i] − 1 ;
                                                   e(s) := e(s) − i ;
                                                   V(sempriv[i])
                                                   fin
                                             sinon aller à FIN ;
                       FIN : V(mutexs) ;
```

3)

a)

```
sémaphore priorité = 2 n − 1 ;

  PROCESSUS CLASSE i

  P(n + i − 1, priorité) ;
  utilisation de la ressource ;
  V(i, priorité) ;
  pour j := 1 pas 1 jusqu'à n − 1 faire
  V(1, priorité) ;
```

b)

```
sémaphore mutex = 1 ; sémaphore tableau sempriv[1 : n] = 0[1 : n] ;
entier disponible ; entier tableau bloque[1 : n] = 0[1 : n] ;
  disponible := 1 ;

        PROCESSUS CLASSE i

        P(mutex) ;
        si disponible = 1 alors début
                          disponible := 0 ;
                          V(mutex)
                          fin
```

```
sinon début
        bloque[i] := bloque[i] + 1 ;
        V(mutex) ;
        P(sempriv[i])
        fin ;
utilisation de la ressource ;
P(mutex) ;
pour j := 1 pas 1 jusqu'à n faire
      si bloque[j] > 0 alors début
                  bloque[j] := bloque[j] − 1 ;
                  V(sempriv[j]) ;
                  aller à FIN
                  fin ;
disponible := 1 ;
FIN : V(mutex) ;
```

Exercice 9

Soit avec les hypothèses de 2.411, *nprod* (respectivement *ncons*) le nombre de cycles complets (phase 1 + phase 2) exécutés par le producteur (respectivement consommateur) depuis l'instant initial. On se place dans une situation où les deux processus ont exécuté l'opération P sur leurs sémaphores respectifs mais n'ont pas encore exécuté l'opération V du même cycle. On a alors :

$$nvide = n - nprod + ncons - 1$$
$$nplein = nprod - ncons - 1$$

Par ailleurs, on a les relations suivantes entre les valeurs de *nplein* (respectivement *nvide*) et l'état du producteur (respectivement consommateur) :

$nplein = -1$: producteur bloqué
$nplein \geqslant 0$: producteur dans sa phase 2

On a donc le tableau suivant :

CONS \ *PROD*	Phase de dépôt (2)	Bloqué
Phase de retrait (2)	$0 < nprod - ncons < n$	$nprod - ncons = 0$
Bloqué	$nprod - ncons = n$	$\begin{cases} nprod - ncons = 0 \\ nprod - ncons = n \end{cases}$

Les propriétés 1 et 2 se déduisent immédiatement de ce tableau.

Propriété 1. L'hypothèse que les deux processus sont simultanément bloqués conduit à deux relations incompatibles si $n > 0$.

Propriété 2. La relation *tête = queue*, avec les deux processus en phase de dépôt ou retrait implique *nprod = ncons* mod *n*, incompatible avec

$$0 < nprod - ncons < n .$$

Exercice 10

1)

```
                sémaphore mutexres = 1, réserve = m ;
                sémaphore tableau mutex[1:n] = 1[1:n],
                                  nplein[1:n] = 0[1:n] ;
        début entier x ;                          début entier x ;
PROD_i : Produire(article) ;          CONS_i : P(nplein[i]) ;
        P(réserve) ;                           P(mutex[i]) ;
        P(mutexres) ;                          Extraire(x, i) ;
        Demander-case(x) ;                     V(mutex[i]) ;
        V(mutexres) ;                          Prélever(article, x) ;
        Déposer(article, x) ;                  P(mutexres) ;
        P(mutex[i]) ;                          Restituer case (x) ;
        Chaîner(x, i) ;                        V(mutexres) ;
        V(mutex[i]) ;                          V(réserve) ;
        V(nplein[i]) ;                         Consommer(article) ;
        aller à PROD_i                         aller à CONS_i
        fin                                    fin
```

2) On note *msémaphore* le déclarateur de sémaphore avec message.

```
        msémaphore réserve = m ;
        Initialiser la file de messages de « réserve » avec les
        adresses des m cases de la réserve ;
        msémaphore tableau nplein[1:n] = 0[1:n] ;
        début entier x ;                          début entier x ;
PROD_i : Produire(article) ;          CONS_i : P_M(nplein[i], x) ;
        P_M(réserve, x) ;                      Prélever(article, x) ;
        Déposer(article, x) ;                  V_M(réserve, x) ;
        V_M(nplein[i], x) ;                    Consommer(article) ;
        aller à PROD_i                         aller à CONS_i
        fin                                    fin
```

On notera que les procédures permettant de réaliser le chaînage des cases sont devenues inutiles : les chaînages sont en effet établis dans les files de messages des sémaphores.

3) Si le nombre de cases d'un tampon t_i est limité à l, il suffit d'introduire le tableau suivant :

sémaphore tableau nvide[1:n] = l[1:n] ;

Dans le programme des producteurs, après l'instruction *Produire(article)*, ajouter :

P(nvide[i]) ;

Dans le programme des consommateurs, avant l'instruction

Consommer(article),

ajouter :

V(nvide[i]) ;

Exercice 11

1)
```
booléen tableau c[0:4] = faux[0:4] ;
                PHILOSOPHE i
          L : penser ;
              avec c lorsque ¬ c[i − 1] et ¬ c[i + 1] faire
                    c[i] := vrai ;
              manger ;
              avec c faire c[i] := faux ;
              aller à L ;
```

2) [Brinch Hansen, 72]

Nous donnons la solution dans le cas où les rédacteurs sont plus prioritaires que les lecteurs. On désigne par v une variable composée constituée de deux entiers nl et nr ; l'instruction *avec v faire* assure l'exclusion mutuelle à nl et nr ; la variable w est une variable simple.

```
LECTEURS                          REDACTEURS
avec v lorsque nr = 0 faire       avec v faire
nl := nl + 1 ;                    nr := nr + 1 et attendre nl = 0 ;
lecture ;                         avec w faire écriture ;
avec v faire nl := nl − 1 ;       avec v faire nr := nr − 1 ;
```

3)
```
début entier j, compte ;
      sémaphore mutex= 1, s = 0 ;
      compte := 0 ;
       A : P(mutex) ;
           si b alors
               début
               I ;
               pour j := 1 pas 1 jusqu'à compte faire V(s) ;
               compte := 0 ;
               V(mutex)
               fin
```

```
sinon
    début
    compte := compte + 1 ;
    V(mutex) ;
    P(s) ;
    aller à A
    fin
fin
```

Commentaires

On suppose que l'évaluation de b ne provoque pas d'effets de bord sur les éléments de la variable v ; sinon il faut activer tous les processus en attente de s à chaque sortie de section critique.

2) On suppose que chaque processus est laissé dans l'ignorance des conditions attendues par les autres ; on ne peut alors que réveiller tous les processus en attente.

Exercice 12

```
sémaphore tableau mutex[0:4] = 1[0:4] ;
PROCESSUS PHILOSOPHE i
Li : début
    penser ;
    P(mutex[i], mutex[i + 1]) ;
    manger ;
    V(mutex[i]) ;
    V(mutex[i + 1]) ;
    aller à Li
    fin
```

CHAPITRE 3

Exercice 1

S'il n'existe qu'un seul anneau de protection, les objets rémanents et les liens associés à une procédure sont en un seul exemplaire, dans un seul segment de liaison associé au segment procédure. Appelons $Z(p)$ la région groupant ces emplacements pour un segment-procédure p donné.

S'il n'existe qu'un seul segment de liaison, soit L, par processus, il faut ranger l'ensemble des régions $Z(p)$ dans ce segment. On rappelle que :

a) $Z(p)$ est de taille fixe,
b) sa durée de vie est égale à celle du processus.

On fait abstraction ici des problèmes de taille. Autrement dit, si $|\ Z(p)\ |$ est la taille d'une région, on suppose que, pour tous les p nommés par le processus :

$$\Sigma\ |\ Z(p)\ | < 2^{24}$$

Quatre problèmes se posent :

a) comment le processus connaît-il le segment de liaison L ?
b) comment créer une région $Z(p)$?
c) comment lier $Z(p)$ au segment-procédure ?
d) comment désigner l'un des emplacements de $Z(p)$?

a) *Désignation de L*

On peut supposer que lorsque le processus est créé, deux segments sont créés :

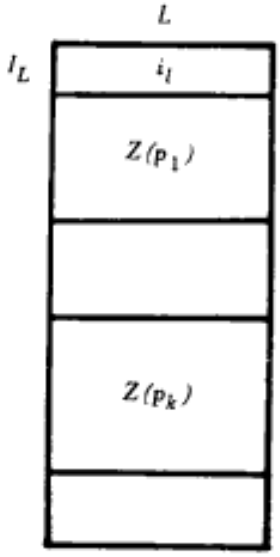
— l'un est destiné à contenir la pile unique (puisqu'il n'y a qu'un seul anneau) ; il a pour nom *0*,

— l'autre est destiné à contenir les régions $Z(p_i)$; il a pour nom *1* par exemple.

Par la suite, on désigne par l le nom du segment de liaison.

b) *Création*

Soit k régions $Z(p_i)$ créées successivement dans L. Considérons le moment où un segment-procédure p_{k+1} est amené dans l'espace adressable du processus (une entrée est créée dans le descriptif). Soit I_L le premier mot du segment de liaison, dont le contenu repère le premier emplacement libre dans L et soit i_l ce contenu. Les opérations nécessaires, effectuées par l'éditeur de liens à la première référence au segment-procédure p_{k+1}, sont les suivantes :



a) recopier le modèle de $Z(p_{k+1})$, rangé dans p_{k+1}, dans les emplacements :

$$(l, i_l) \text{ à } (l, i_l + |Z(p_{k+1})| - 1)$$

b) faire :

$$I_L := i_l + |Z(p_{k+1})| .$$

c) *Le catalogue des segments connus*

Considérons un second appel à un segment-procédure déjà lié, soit p_i. L'éditeur de liens, appelé pour transformer l'identificateur associé à p_i en un nom consulte le catalogue des segments connus. L'examen de ce catalogue permet de déduire le nom du segment p_i et le nom de $Z(p_i)$: ce nom est en théorie un couple (l, i_l), mais, en pratique, il peut se réduire à la seule donnée de i_l.

d) *Adressage*

Considérons le registre *R4*, contenant le couple *(l, i)* (au lieu du seul nom du segment de liaison). Alors toute désignation d'un objet externe et d'un objet rémanent s'effectue par indexation composée, soit respectivement :

* *(ic, R4, d)* si le nom de l'objet externe est rangé dans le *(d + 1)*-ième mot de $Z(p_l)$

(ic, R4, d) si l'objet rémanent est le contenu du *(d + 1)*-ième mot de $Z(p_i)$.

Remarque. La valeur i_l pourrait être rangée dans le descripteur de L, en tant que taille de L. Cette solution implique une succession de modifications de taille de L qu'il est souhaitable d'éviter.

Exercice 2

S'il n'existe qu'un seul anneau de protection, une seule pile est nécessaire. Dans ce cas, au prix de quelques aménagements de l'appel de procédure, un seul registre suffit pour désigner la région des locaux et celle des arguments (dans ce cas la zone de sauvegarde des registres doit être de taille fixe, ou encore elle doit être située après les objets locaux ou avant les arguments).

Si le segment contenant la pile est supprimé et que celle-ci est répartie dans les segments de liaison, tout se passe comme si on avait autant de piles que de procédures différentes. Aussi la solution la plus simple est de ne pas modifier la répartition des informations entre la pile de l'appelant et celle de l'appelé.

Considérons l'appel d'une procédure p_j par une procédure p_i. Etant donné un segment-procédure p_j, on appelle :

— L_j le segment de liaison associé et I_j son premier mot,
— $Z(p_j)$ la région des objets rémanents et des liens,
— $V(p_j)$ la région des objets locaux, $A(p_j)$ celle des paramètres,
— $S_1(p_i)$ et $S_2(p_i)$ les zones de sauvegarde des registres représentant l'environnement de p_i.

S_1 est la zone de sauvegarde garnie par programme, S_2 est la zone de sauvegarde garnie automatiquement à l'appel de procédure. Avant l'appel de p_j, on suppose qu'on a la situation décrite sur la figure 1.

L'appel de procédure agit exactement comme décrit en 3.25 ; on aura donc, après l'appel la situation indiquée sur la figure 2.

Pendant l'exécution d'une procédure p_k, les registres *R3* et *R4* contiennent respectivement (l_k, d) et $(l_k, 0)$, c'est-à-dire une adresse segmentée désignant un emplacement du segment de liaison.

L'adressage des locaux s'effectue par indexation composée sur *R3*. L'adressage dans $Z(p_k)$ s'effectue par indexation partielle sur *R4*. Il en résulte qu'un seul registre suffit pour les deux types d'adressage, par exemple le registre *R3*. Ainsi, grâce à cette technique, on peut supprimer un registre, le registre *R4*.

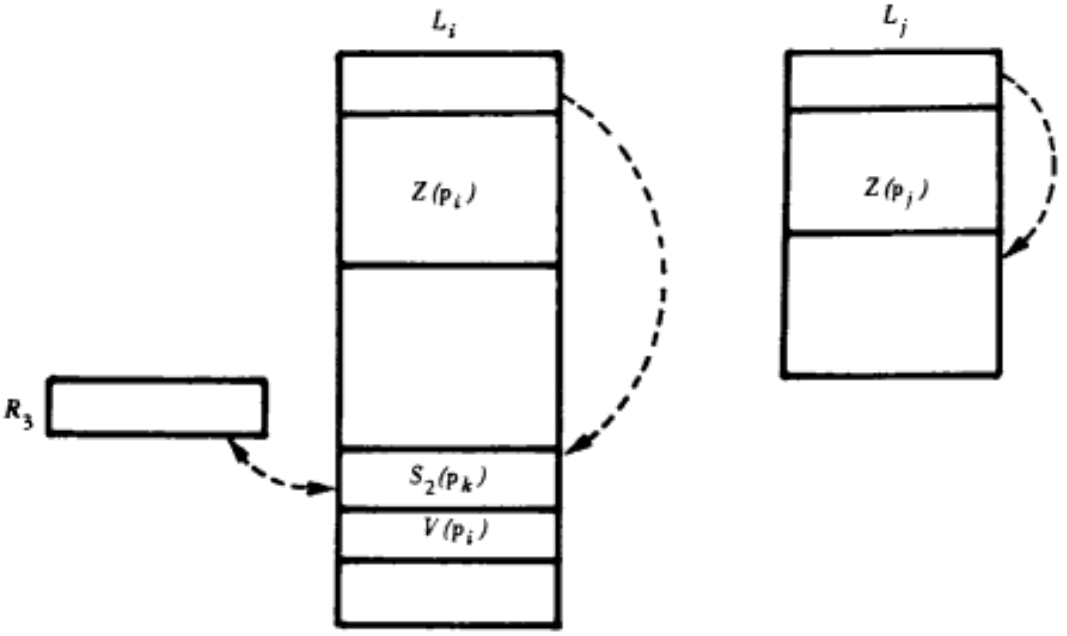


Figure 1.

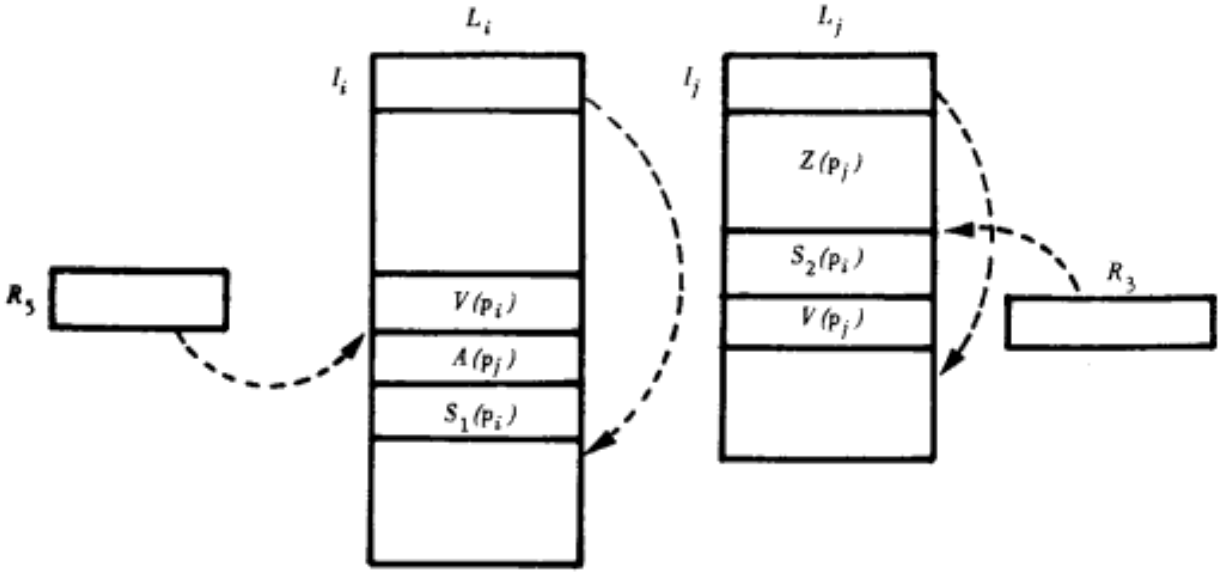


Figure 2.

Exercice 3

Lorsqu'un segment est détruit, son nom devient disponible. Lorsqu'il s'agit d'un segment-procédure, les noms de ses segments de liaison (un par anneau d'appel effectué) deviennent aussi disponibles.

Pour pouvoir les réutiliser sans risque, il faut être certain qu'aucune chaîne d'accès ne peut plus conduire aux segments détruits.

Soit S le segment détruit, d'identificateur *TOUAMOTOU*. Toutes les références à S sont faites :

— dans S, par des instructions de branchement, si S est un segment-procédure,

— dans des segments-procédure S' qui ont comme objets externes des objets de S,

— dans des segments-procédure S'' qui ont comme paramètres effectifs des objets de S.

On peut retrouver les segments-procédure S' à l'aide du catalogue général des segments. En effet on y trouve pour l'identificateur *TOUAMOTOU*, la liste des processus qui le désignent par un nom et le nom donné par chaque processus. Soit p_i un tel processus et s_i le nom par lequel il désigne S. On procède comme suit :

Dans tous les segments de liaison $L(S', i)$ associés aux segments-procédure S' ayant un nom dans p_i, on recherche, dans la région des noms des objets externes tous les emplacements qui contiennent une adresse segmentée avec le nom de segment s_i. Dans tous ces emplacements, on remet l'indicateur de déroutement.

Cette opération est à effectuer pour tous les processus p_i qui sont cités dans le catalogue général, dans la liste associée à l'identificateur *TOUAMOTOU*.

Il est beaucoup plus difficile de retrouver les paramètres effectifs qui sont des objets de S, car il faut faire une recherche dans les piles, dans les zones des paramètres effectifs de chaque procédure en cours d'exécution.

Une façon de faciliter cette recherche serait de mettre dans les emplacements réservés aux paramètres effectifs qui sont des objets externes, non pas les noms de ceux-ci, mais le nom du lien qui figure dans le segment de liaison. On se ramènerait alors au cas précédent. Cette méthode n'est réalisable que si l'adressage indirect à deux niveaux est possible.

Exercice 4

1 Accès aux objets

1.1 *Désignation d'une procédure*

Considérons les n segments-procédure numérotés, selon un ordre fixé par le compilateur, de 0 à $n - 1$. Au début de l'exécution, les n descripteurs sont rangés dans des emplacements de numéros $e, e + 1, \ldots, e + n - 1$ dans le descriptif. Ces emplacements sont inconnus du compilateur. Aussi, une procédure est désignée, dans le programme, par un nom

$$(a, i)$$

où i est un entier compris entre 0 et $n - 1$. Le contenu du registre de numéro p désigne la base d'une zone de liaison contenant les n adresses segmentées $(e + i, 0)$, $i = 0, \ldots, n - 1$.

Cette zone peut indifféremment être rangée dans un segment de liaison, unique pour un processus, ou en début de pile. Soit, par convention, $a = 12$.

1.2 *Désignation des objets de l'environnement d'une procédure*

Soit p une procédure de niveau d'emboîtement i. A l'aide de noms de la forme (k, x), où k est un numéro de registre et x un indice, elle désigne :

a) Les objets locaux et les paramètres de toutes les procédures de l'environnement. Par définition, celui-ci est formé de i régions de la pile, chaque région groupant les objets locaux et les paramètres d'une procédure de niveau d'emboîtement j $(1 \leqslant j \leqslant i)$.

b) Ses propres instructions, dans le segment-procédure.

c) Les procédures de l'environnement.

d) La zone d'évaluation, rangée en sommet de pile, et elle-même gérée en pile.

On choisit les conventions suivantes :

— *R0* est le compteur ordinal ; son contenu désigne l'instruction en cours de la procédure active.

— Le contenu des registres *R1* à *Ri* désigne la région de l'environnement de de niveau d'emboîtement i. Les registres *R1* à *R11* sont disponibles à cet effet, ce qui fixe à *11* la valeur maximale du niveau i.

— Le contenu de *R12* désigne la zone des descripteurs de segment-procédure,

— Le contenu de *R13* désigne le sommet de la zone d'évaluation.

— *R14* et *R15* sont utilisés pour les calculs.

On a ainsi (voir Fig. 1) :

— nom d'une étiquette locale à la procédure : $(\underline{ip}, 0, x)$,

— nom d'un objet local de l'environnement : $(\underline{ic}, i, x)$, $1 \leqslant i \leqslant 11$,

— nom d'un paramètre de l'environnement : $*\,(\underline{ic}, i, x)$, $1 \leqslant i \leqslant 11$,

— nom d'une procédure (elle appartient à l'environnement par construction) : $(\underline{ip}, 12, i)$,

— nom du sommet de pile $(\underline{ic}, 13, 0)$.

2 Commutation d'environnement

Dans cette mise en œuvre, on observe :

— que les registres *R0* (compteur ordinal), *R14* et *R15* sont à sauvegarder. Une zone de liaison (hachurée sur la figure) contient la valeur de ces registres,

— que le sommet de pile est constamment désigné par le contenu de *R13*.

2.1 *Appel d'une procédure*

Par définition, depuis la procédure p de niveau d'emboîtement i, on ne peut appeler qu'une procédure q de niveau d'emboîtement j, avec $1 \leqslant j$ $i + 1$.

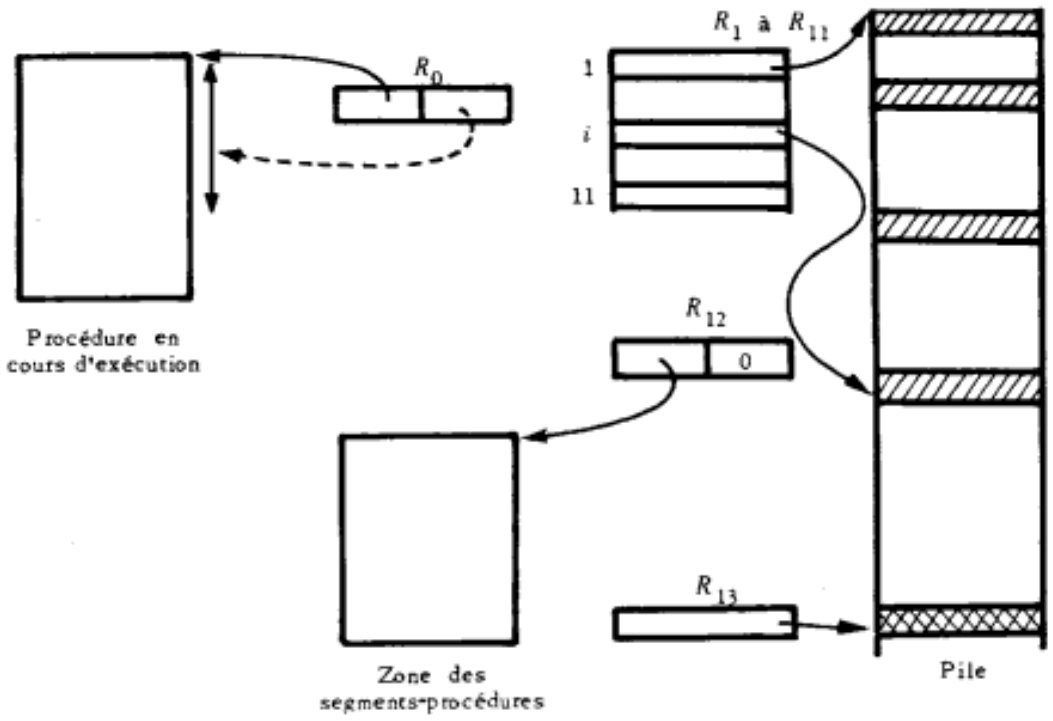


Figure 1.

L'effet de l'appel de procédure est le suivant :

— création d'une zone de liaison en tête de pile, et sauvegarde des registres *R0*, *R14* et *R15*,

— évaluation des paramètres et rangement de leur adresse segmentée après la zone de liaison ; le k-ième paramètre est rangé à l'adresse $(\underline{ic}, 13, v)$, avec v = taille de la zone de liaison + k.

— rangement de i et du contenu du registre Ri (partie déplacement uniquement) dans la zone de liaison,

— rangement de l'adresse segmentée du premier mot de la nouvelle zone de liaison dans le registre Rj,

— mise à jour de *R0*, création des locaux de q et mise à jour de *R13*.

2.2 *Retour de procédure*

Le contenu de Rj désigne la zone de liaison de la région courante. Celle-ci contient le niveau d'emboitement i de la procédure appelante et l'adresse zl de la zone de liaison de la région associée.

Si $i = j-1$, les contenus des registres $1, 2, \ldots, i$ désignent déjà les régions de l'environnement de retour. Dans le cas contraire $(i \geqslant j)$, il faut remettre à jour les contenus des registres $i, i-1, \ldots, j$. L'adresse zl est alors rangée dans le registre i ; dans $pile[zl]$, on trouve l'adresse de la zone de liaison à ranger dans le registre $i-1$, etc...

R0, *R14* et *R15* sont remis à jour grâce à la zone de liaison. *R13* prend comme valeur le contenu du registre j avant commutation diminué de *1*.

Exercice 5

Le deuxième mot de la zone de liaison de la région appelée permet de rétablir le registre pointeur d'instruction. L'élément de chaîne dynamique, qui contient un déplacement d, permet de retrouver la zone de liaison *L(appelante)* de la région appelante, où figure le niveau d'emboîtement n de la région appelante. Le registre d'environnement d'indice n reçoit la valeur d ; ceux d'indice $n - 1$, $n - 2$, etc..., reçoivent des valeurs obtenues d'après la chaîne statique commençant dans *L(appelante)*. Le registre de sommet de pile reçoit le contenu du registre de région courante diminué de 1 ; ce dernier registre reçoit ensuite la valeur d. L'exécution peut ensuite reprendre.

Exercice 6

Les registres d'environnement ne sont pas indispensables puisque dans la région courante se trouve aussi la représentation de son environnement, sous forme de la chaîne statique. L'utilisation de la chaîne statique serait évidemment peu intéressante car il faudrait éventuellement effectuer plusieurs accès dans la pile (en lisant la chaîne statique) avant d'accéder à un objet. La connaissance de la région courante est indispensable.

Exercice 7

On voit qu'il faut écrire deux procédures *p11* et *p22* qui n'apparaissent pas dans le système BURROUGHS (elles devraient se trouver représentées dans le processus-racine), parce qu'elles n'ont pas la même durée de vie que les autres objets du processus-racine.

Exercice 9

A chaque segment de nom f, on associe :

— le numéro d'article courant (c'est celui qui est en cours de remplissage) noté $a(f)$,

— un index de remplissage, noté $r(f)$, pointant vers le premier caractère non écrit $(0 \leqslant r \leqslant l)$ dans l'article.

La procédure proprement dite utilise une page virtuelle de numéro v comme espace de travail. Elle s'écrit :

```
procédure écrire (u, n, f) ; entier u, n, f ; valeur u, n, f ;
    début
    entier long, dep, nombre ;
    long := n ;
    tant que long > 0 faire
        début
        coupler (v, a(f), f) ;
        si r(f) + long ⩾ l alors
            début
            dep := 0 ;
```

```
            nombre := l - r(f);
            a(f) := a(f) + 1
            fin
        sinon
            début
            nombre := long;
            dep := r(f) + long
            fin;
        transférer (nombre, u, v * l + r(f));
        u := u + nombre;
        r(f) := dep;
        long := long - nombre
    fin
fin
```

Remarque. La procédure *transférer* (n, u_1, u_2) recopie n octets débutant à l'adresse d'octet u_1 dans les n octets suivant l'adresse d'octet u_2 (instruction *MBS* du 10070).

Exercice 10

1 Cas d'une mémoire virtuelle par processus

a) *Partage d'une procédure P*

Pour assurer la réentrance, étant donné que l'on ne dispose d'aucun mécanisme de réimplantation en mémoire virtuelle, il suffit d'éditer et de coupler la procédure P aux mêmes adresses dans les différentes mémoires virtuelles, et d'associer les données à des pages physiques différentes (Fig. 1).

b) *Partage des données D*

Le partage des données entre deux procédures différentes associées à des processus distincts est résolu de façon différente suivant que :

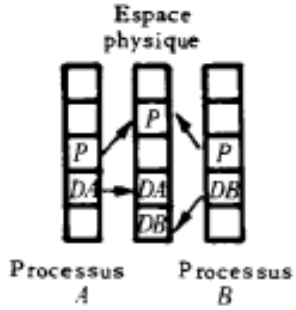
— les données ne contiennent aucune référence à des adresses,

— les données contiennent des adresses qui font référence à la procédure appelante,

— les données contiennent des adresses qui font référence aux données elles-mêmes.

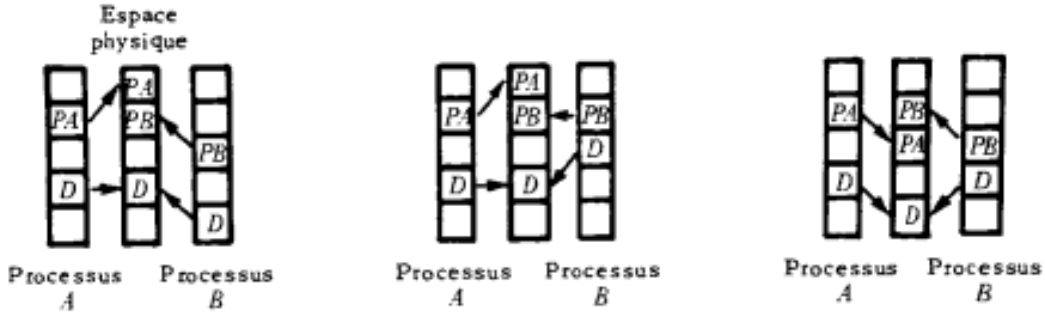
Si les données ne contiennent aucune référence à des adresses, elles peuvent être couplées à des adresses différentes, dans les deux mémoires virtuelles ; il suffit que les procédures appelantes soient éditées pour les nommer à ces adresses (Fig. 2a).

Si les données contiennent des adresses qui font référence à des adresses de la procédure appelante, elles peuvent être couplées à des pages virtuelles différentes ; par contre les procédures doivent être éditées et couplées aux mêmes adresses dans les deux mémoires virtuelles (Fig. 2b).



DA représente les données du processus *A*
DB représente les données du processus *B*

Figure 1. Partage d'une procédure *P* entre 2 processus *A* et *B* associés à des mémoires virtuelles différentes.



PA représente la procédure associée au processus *A*
PB représente la procédure associée au processus *B*

Figure 2. Partage d'une zone de données *D* entre deux processus *A* et *B*.

Si les données contiennent des adresses qui font référence aux données elles-mêmes (par exemple par adressage indirect), elles doivent être éditées et couplées aux mêmes adresses dans les deux mémoires virtuelles (Fig. 2c).

2 Cas de plusieurs processus travaillant dans la même mémoire virtuelle

a) *Partage d'une procédure P*

Dans ce schéma l'ensemble des registres constitue le seul espace d'adressage propre à chaque processus. Cette restriction n'interdit pas le partage d'une même procédure *P* ; elle impose simplement de tenir compte de certaines règles dans le codage de la procédure.

Ainsi la réentrance peut être assurée,

— soit en programmant la procédure de telle façon que son espace de travail soit restreint aux registres seuls, si cette restriction est possible,

— soit en programmant la procédure de façon qu'elle puisse désigner ses données alternativement avec des adresses différentes suivant le processus qui l'exécute ; dans ce cas l'adresse de la base des données propres à chaque processus doit constituer un paramètre de la procédure.

b) *Partage des données D*

Le partage des données entre différents processus travaillant dans la même mémoire virtuelle est résolu sans mécanisme particulier par le seul fait que les processus ont accès à un espace commun d'adressage.

CHAPITRE 4

Exercice 1

1) Le mécanisme de génération de la chaîne r_k peut être décrit comme suit [Spirn, 72] : on entretient une pile (dite pile *LRU*) telle que le numéro de la page la plus récemment référencée figure à tout instant au sommet. Soit $P_t = x_1\, x_2 \ldots x_n$ (les x_i sont des numéros de page) la composition de la pile à l'instant t, en partant du sommet. Si la référence r_{t+1} est la page x_k, le nouvel état de la pile est : $P_{t+1} = x_k\, x_1\, x_2 \ldots x_{k-1}\, x_{k+1} \ldots x_n$. Soit $a_i(t)$ la probabilité à l'instant t pour que la référence r_{t+1} soit la page $x_i (t > i)$. On doit avoir, quel que soit $m (0 < m \leqslant n)$:

$$\begin{aligned} a_1(t) &= p_1 \\ a_1(t) + a_2(t) &= p_2 \\ &\cdots\cdots\cdots\cdots \\ a_1(t) + a_2(t) + \cdots + a_m(t) &= p_m \end{aligned}$$

d'où :

$$a_1(t) = p_1 \qquad \text{et} \qquad a_i(t) = p_i - p_{i-1} \qquad \text{pour} \qquad i = 2, 3, \ldots, n$$

Les a_i sont donc des constantes indépendantes de t (sauf pendant la phase d'initialisation de la pile *LRU*, où on assigne une probabilité égale aux pages non encore référencées). La donnée des a_i (ou des p_i) définit complètement le modèle.

2) Le modèle a un comportement de localité si :

$$a_1 \geqslant a_2 \geqslant \cdots \geqslant a_n$$

Les références récentes sont d'autant plus privilégiées que les a_i d'indice faible sont élevés. On doit avoir $\Sigma a_i = p_n = 1$.

3) Les l pages le plus récemment référencées sont les l pages au sommet de la pile *LRU*. On a donc pour le modèle proposé :

$$\begin{aligned} a_1 = a_2 = \cdots = a_l &= \lambda/l \\ a_{l+1} = a_{l+2} = \cdots = a_n &= (1 - \lambda)/(n - l) \end{aligned}$$

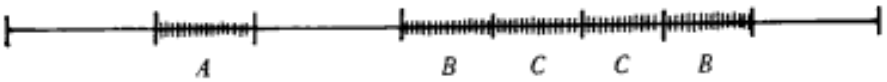
La propriété de localité s'exprime par :

$$\lambda/l > (1 - \lambda)/(n - l)$$

Le modèle est entièrement défini par la donnée de l et λ. L_i est défini à tout instant comme l'ensemble des l dernières pages référencées, et la durée de vie moyenne de L_i (intervalle de temps pendant lequel L_i garde la même composition) a pour valeur moyenne l/λ.

Exercice 2

Les N zones allouées se répartissent en 3 classes suivant qu'elles sont contiguës à 2, 1, ou 0 zones libres (voir schéma).



Soit N_A, N_B, N_C le nombre de zones de chaque classe et M le nombre de zones libres, à un instant donné. Nous supposerons que l'allocation se fait toujours à partir d'une des extrémités d'une zone libre et qu'une zone libérée est fusionnée avec toute zone libre adjacente. On a les relations :

$N = N_A + N_B + N_C$

$M = \frac{1}{2}(2N_A + N_B)$, à ± 1 près selon la configuration aux extrémités de la mémoire.

Soit p la probabilité pour qu'une demande de zone ne soit pas satisfaite par une zone libre de même taille (en général $p \simeq 1$).

La diminution ΔM_1 du nombre de zones libres, consécutive à une allocation est en moyenne :

$$\Delta M_1 = (1 - p)$$

L'augmentation ΔM_2 du nombre de zones libres, consécutive à une libération est :

$$\Delta M_2 = \frac{N_B}{N}.0 + \frac{N_A}{N}(-1) + \frac{N_C}{N}(1) = \frac{N - 2M}{N}$$

Lorsque le système est en équilibre, $\Delta M_1 = \Delta M_2$; il vient :

$$M = \tfrac{1}{2} pN$$

Exercice 4

Nous considérons un système en équilibre, avec $L > 1$ de façon que la file d'attente ne soit jamais vide ; dans ce cas le temps d'accès à une page demandée se mesure en nombres entiers de pages, mis à part le premier transfert.

Nous appliquerons la formule de Little (cf. 6.222) : soit λ le débit moyen des arrivées des demandes et soit S le temps moyen de résidence d'une demande dans la file ; la formule de Little nous donne

$$L = \lambda S$$

Lorsqu'une demande arrive dans la file elle trouve L demandes devant elle en moyenne ; soit e le temps moyen d'exécution d'une demande, ce temps comprenant le temps d'accès moyen a et le temps de transfert t ;

$$e = a + t$$

Le temps de résidence d'une demande est donc, en moyenne

$$S = Le$$

Introduisons le temps de révolution R du disque ; nous avons

$$t = \frac{R}{s}$$

L'efficacité est égale au rapport

$$E = \frac{\lambda}{\left(\frac{s}{R}\right)} = \frac{\lambda R}{s}$$

ce qui peut s'écrire

$$E = \frac{L}{S}\frac{R}{s} = \frac{R}{se} = \frac{R}{sa + R}$$

En FIFO le temps d'accès moyen a, lorsqu'on se place sur une limite de page, est égal à

$$a = \frac{R}{s}(0 + 1 + 2 + \cdots + s - 1) \; /s = \frac{R}{2s}(s - 1)$$

d'où

$$E_f = \frac{2}{s + 1}$$

Avec la politique SATF, le temps moyen d'accès est approximativement (cf. 6.231)

$$a \simeq \frac{R}{L + 1}\left(1 - \frac{1}{2s}\right)^{L+1} \simeq \frac{R}{L + 1}$$

d'où

$$E_t \simeq \frac{L + 1}{s + L + 1}$$

Exercice 5

Soit la chaine de références [Belady, 69]

$$\omega = 1 \quad 2 \quad 3 \quad 4 \quad 1 \quad 2 \quad 5 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5$$

On obtient avec $m = 3$ et $m = 4$ pour FIFO et pour LRU à titre de comparaison

FIFO

	1	2	3	4	1	2	5	5	5	3	4	4
		1	2	3	4	1	2	2	2	5	3	3
$m = 3$			1	2	3	4	1	1	1	2	5	5

$C(3, \omega) = 9$

	1	2	3	4	4	4	5	1	2	3	4	5
		1	2	3	3	3	4	5	1	2	3	4
$m = 4$			1	2	2	2	3	4	5	1	2	3
				1	1	1	2	3	4	5	1	2

$C(4, \omega) = 10$

LRU

	1	2	3	4	1	2	5	1	2	3	4	5
		1	2	3	4	1	2	5	1	2	3	4
$m = 3$			1	2	3	4	1	2	5	1	2	3

$C(3, \omega) = 10$

	1	2	3	4	1	2	5	1	2	3	4	5
		1	2	3	4	1	2	5	1	2	3	4
$m = 4$			1	2	3	4	1	2	5	1	2	3
				1	2	3	4	4	4	5	1	2

$C(4, \omega) = 8$

Exercice 6

Comme $S(m, \omega)$ contient toujours les m pages les plus récemment référencées, $S(m, \omega)$ est un sous-ensemble de $S(m + 1, \omega)$, ce qui démontre la propriété (cf. exercice 5).

Exercice 7

1) Les bits U peuvent être remis à 0

— soit au moment où intervient le remplacement, et dans le cas où tous les bits U ont été mis à 1,

— soit immédiatement (et automatiquement) dès que le dernier bit U est mis à 1.

Dans les 2 cas, le dernier bit mis à 1 n'est pas remis à 0.

2) Les pages sont réparties en 4 groupes disjoints suivant la valeur du couple (U, E). Ces groupes sont classés comme suit : $(0, 0)$, $(0, 1)$, $(1, 0)$, $(1, 1)$.

La page à remplacer est choisie, de façon aléatoire , dans le premier groupe non vide (en commençant par le groupe $(0, 0)$).

Exercice 8

1) La relation de l'énoncé s'interprète comme suit : pour une taille s de mémoire principale allouée au programme pendant un temps t donné, il est préférable (en nombre d'instructions exécutées) d'exécuter le programme en lui allouant pendant un temps $t/2$, une taille de mémoire égale à $(s - \Delta s)$ et pendant $t/2$ une taille de mémoire égale à $(s + \Delta s)$, plutôt que de lui allouer pendant un temps t une taille de mémoire constante s.

2) Pour tenir compte du résultat précédent on peut privilégier pendant un intervalle de temps donné, chacun des processus en compétition. On peut, par exemple, décider que les pages de mémoire principale appartenant au processus privilégié, ne seront pas choisies par l'algorithme de remplacement pendant une période de p remplacements successifs ; après quoi on décide de privilégier un autre processus. En conséquence, pendant un certain temps (le temps requis pour effectuer p remplacements) l'espace alloué au processus privilégié augmentera de Δs pages ; après quoi le processus perdra ses pages plus rapidement que si un simple algorithme FIFO était utilisé. L'algorithme ainsi modifié est appelé BIFO (« biased FIFO »).

Exercice 9

Les deux graphes H_1 et H_2 sont caractérisés par les mêmes matrices d'allocation et de demande. En effet, puisque les suites S_1 et S_2 contiennent les mêmes processus à l'ordre près,

$$A^{S_1} = A^{S_2}$$

où A^S est la matrice décrivant les allocations aux processus de la suite S.

$$DN^{S_1} = DN^{S_2}$$

où DN^S est la matrice décrivant les demandes non satisfaites des processus de la suite S.

On a donc :

$$A^{H_1} = A - A^{S_1} = A - A^{S_2} = A^{H_2}$$
$$DN^{H_1} = DN - DN^{S_1} = DN - DN^{S_2} = DN^{H_2}$$

Exercice 10

Appelons R^G les ressources libres dans l'état G :

$$R^G = X - \sum_{j=1}^{n} A_j^G$$

Le graphe G est réduit par S dans le graphe H :

$$A^H = A^G - A^S$$

donc
$$R^H = X - \sum_{j=1}^{n} A_j^H \geqslant X - \sum_{j=1}^{n} A_j^G = R^G$$

Mais le graphe G est réductible par p_i, donc :

$$DN_i \leqslant R^G \leqslant R^H$$

H est aussi réductible par p_i.

Exercice 11

Supposons $H_1 \neq H_2$.

La proposition 1 implique que les suites S_1 et S_2 ne contiennent pas les mêmes processus. Supposons qu'il existe un processus p_i tel que :

$$p_i \in S_1 \qquad \text{et} \qquad p_i \notin S_2$$

Remarquons que les rôles de S_1 et S_2 sont symétriques. Soit S le début de la suite S_1 jusqu'au processus p_i exclu (S est éventuellement vide).

Tous les processus de S appartiennent aussi à S_2 :

$$R^G + \sum_{p_j \in S_2} A_j \geqslant R^G + \sum_{p_j \in S} A_j \geqslant DN_i$$

Si après réduction de G par S le graphe obtenu est encore réductible par p_i, alors *a fortiori* le graphe obtenu après réduction de G par S_2 est encore réductible par p_i. Cela est contraire à l'hypothèse que S_2 est une séquence. Donc tout processus de S_1 appartient à S_2 et, par symétrie, tout processus de S_2 appartient à S_1.

La proposition 1 implique alors $H_1 = H_2$.

Exercice 12

Soit $G_0 = G$ et soit G_k le graphe obtenu après réduction de G_{k-1} par le processus p_{q_k}.

1) Soit S une suite de réductions. Lorsqu'on réduit G_0 par p_{q_1}, on obtient :

$$DN_{q_1} \leqslant R^0 \qquad \text{et} \qquad R^1 = R^0 + A_{q_1}$$

où R^k est le vecteur des ressources libres dans l'état G_k. A la k-ième réduction (par p_{q_k}) on a :

$$DN_{q_k} \leqslant R^{k-1} \qquad \text{et} \qquad R^k = R^{k-1} + A_{q_k}$$

ou encore

$$DN_{q_k} \leqslant R^0 + \sum_{h<k} A_{q_h}.$$

Soit $S(i) = k$ l'indice de p_i dans S. Alors :

$$\{ q_h \mid 1 \leqslant h < k \} = \{ j \mid S(j) = h \ \underline{et} \ 1 \leqslant h < k \}$$
$$= \{ j \mid S(j) < S(i) \},$$

d'où :

$$DN_i \leqslant R^0 + \sum_{S(j) < S(i)} A_j$$

Ceci est vrai pour tout p_i de S ; cette suite S est donc saine.

2) Soit S une suite saine, p_i un processus appartenant à S, et soit $S(i)$ l'indice de p_i dans S,

$$DN_i \leqslant R + \sum_{S(j) < S(i)} A_j$$

Pour $i = q_1$, soit $S(i) = 1$, on a :

$$DN_{q_1} \leqslant R^0 = R$$

Le graphe G_0 $(= G)$ peut être réduit par P_{q_1} et

$$R^1 = R^0 + A_{q_1}$$

Pour $i = q_k$, on a :

$$DN_i = DN_{q_k} \leqslant R^0 + \sum_{S(j) < S(i)} A_j$$

Raisonnons par récurrence et admettons que

$$R^{k-1} = R^0 + \sum_{S(j) < S(q_{k-1})} A_j$$

Alors :

$$DN_{q_k} \leqslant R^{k-1}$$

et le graphe G_{k-1} peut être réduit par un processus p_i tel que $S(i) = q_k$ et le graphe G_k obtenu vérifie bien la relation admise. S est donc une suite de réductions.

Exercice 13

Il peut exister des composantes connexes dans le graphe d'état, c'est-à-dire des ensembles disjoints de processus demandant ou possédant des ensembles disjoints de ressources. Considérons une de ces composantes G et supposons qu'il n'y existe pas de circuit.

On démontre en théorie des graphes le théorème suivant : s'il n'existe pas de circuit dans un graphe connexe, il existe une suite non vide S de nœuds du graphe telle que le nœud N précède le nœud M dans S si et seulement si il existe un chemin de N à M dans le graphe. Les premiers nœuds sont les racines, les derniers sont les feuilles.

On peut extraire de S la suite S_1 des processus, par élimination des nœuds ressources. Le graphe G peut être réduit par la suite S_2 inverse de la suite S_1

$$S_1 = (p_{q_1}, \ldots, p_{q_m}) \qquad S_2 = (p_{q_m}, \ldots, p_{q_1})$$

$G_0 = G$ peut être réduit par P_{q_m} en un graphe G_m.

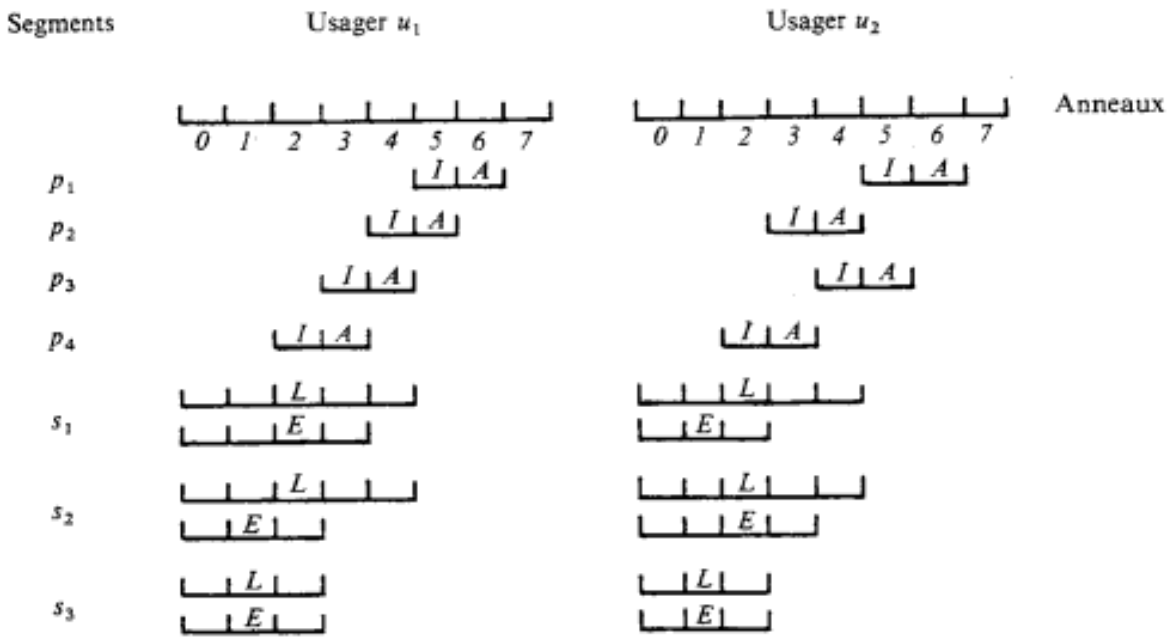
En effet si P_{q_m} a des requêtes en attente, ce ne peut être que sur des ressources feuilles dans la suite S, c'est-à-dire des ressources toutes libres. De même G_{i+1} peut être réduit en un graphe G_i par p_{q_i}, car les demandes insatisfaites de p_{q_i} ne peuvent concerner que des ressources feuilles dans S ou des ressources suivant p_{q_i} dans S, c'est-à-dire possédées par les processus $p_{q_{i+1}}, \ldots, p_{q_m}$. Ces ressources ont alors été libérées lors des réductions précédentes.

S_2 est une séquence de réduction complète du graphe connexe G. Il en est de même pour toutes les composantes connexes du graphe d'état. L'ensemble des séquences S_2 constitue une séquence de réduction complète du graphe d'état. Cette séquence est aussi une suite saine (proposition 4), et le système n'est pas en interblocage. L'existence d'un circuit dans le graphe d'état est bien une condition nécessaire d'interblocage.

CHAPITRE 5

Exercice 1

Une disposition possible des parenthèses d'accès, pour chaque couple segment-usager est, en notant A la parenthèse d'appel :



Remarque. Cette solution nécessite 4 anneaux au minimum *(2, 3, 4* et *5)*. Toute solution obtenue par translation est également valable.

CHAPITRE 6

Exercice 1

En intégrant sur l'escalier (Chap. 6, Fig. 1), on obtient :

$$a(n) = \sum_{j=1}^{m} \frac{r}{m} \left(1 - \frac{j}{m}\right)^n$$

$$= \frac{r}{m^{n+1}} \sum_{j=1}^{m-1} j^n$$

Pour $n = 1$, un raisonnement élémentaire conduit à :

$$a(1) = \frac{r}{m}\left[\frac{0}{m} + \frac{1}{m} + \cdots + \frac{m-1}{m}\right] = \frac{r}{m^2}\sum_{j=1}^{m-1} j$$

Exercice 2

a) La formule :

$$\sum_{i=1}^{n} (1 - u_i(F_n))\, R_i$$

s'obtient aisément en décomposant la surface hachurée en rectangles horizontaux.

b) Soit $N(0, n)$ l'aire de la zone :

$$N(0, n) = \sum_{i}^{n} R_i - \sum_{i}^{n} u_i(F_n)\, R_i$$

On a :

$$\bar{N}(0, n)\, F_n = N(0, n)$$

d'où

$$\bar{N}(0, n)\left(\frac{D_n}{n} + \frac{R_n}{n}\right) = \frac{\sum R_i}{n} - \frac{1}{n}\sum_{i} u_i(F_n)\, R_i$$

$$= \bar{R} - \frac{1}{n}\sum_{i}^{n} u_i(F_n)\, R_i$$

Il vient :

$$\bar{N}(0, n)\left(\frac{D_n}{n} + \frac{R_n}{n}\right) \leqslant \bar{R}$$

En remplaçant R_i par $R_{\max}$, on obtient :

$$\sum_{i}^{n} u_i(F_n)\, R_i \leqslant N(F_n)\, R_{\max}$$

d'où

$$\bar{N}(0, n)\left(\frac{D_n}{n} + \frac{R_n}{n}\right) \geqslant \bar{R} - \frac{1}{n} N(F_n)\, R_{\max}$$

Si le système atteint un état d'équilibre, le terme R_n/n tend vers zéro lorsque n augmente ; les termes $N(F_n)$ et $R_{\max}$ sont finis ; donc

$$\bar{N}(0, n)\,\frac{D_n}{n} \to \bar{R} \quad \text{lorsque} \quad n \to \infty$$

Le terme D_n/n représente l'intervalle moyen $1/\lambda$ entre deux arrivées successives, d'où la formule de Little :

$$\bar{R} = \frac{\bar{N}}{\lambda}$$

Remarque. Cette démonstration donne une interprétation simple de la relation de Little ; en considérant un intervalle de temps très grand T, on peut écrire :

$$\bar{N}T = (\lambda T)\,\bar{R}$$

Cette égalité s'obtient intuitivement en évaluant de deux façons l'aire de la zone hachurée : en prenant les sections verticales d'une part, les rectangles horizontaux d'autre part.

Exercice 3

Le système d'équations linéaires s'écrit :

$$\frac{p_1}{T} - p_0 \frac{n}{R} = 0$$

$$\frac{p_2}{T} - p_1\left[\frac{n-1}{R} + \frac{1}{T}\right] + p_0 \frac{n}{R} = 0$$

$$\frac{p_3}{T} - p_2\left[\frac{n-2}{R} + \frac{1}{T}\right] + p_1 \frac{n-1}{R} = 0$$

...

D'où

$$\frac{p_1}{T} = p_0 \frac{n}{R}$$

En additionnant les deux premières équations :

$$\frac{p_2}{T} = p_1 \frac{n-1}{R}$$

En additionnant la seconde et la troisième :

$$\frac{p_3}{T} = p_2 \frac{n-2}{R}$$

...

Il vient :

$$p_i = p_0\, n(n-1) \ldots (n-i+1) \left(\frac{T}{R}\right)^i = p_0 \frac{n!}{(n-i)!} \left(\frac{T}{R}\right)^i$$

Nous avons l'identité :

$$\sum_{i=0}^{n} (n-i)\, p_i + \sum_{i=0}^{n} i p_i = n \sum_{i=0}^{n} p_i = n$$

Soit :

$$n - q = \sum_{i=0}^{n} (n-i) \frac{n!}{(n-i)!} p_0 \left(\frac{T}{R}\right)^i$$

La somme peut encore s'écrire, en ignorant le temps nul pour $i = n$:

$$n - q = p_0 \frac{R}{T} \sum_{i=0}^{n-1} \frac{n\,!}{(n-i-1)\,!} \left(\frac{T}{R}\right)^{i+1}$$

Faisons le changement de variable $j = i + 1$:

$$\begin{aligned} n - q &= p_0 \frac{R}{T} \left[\sum_{j=1}^{n} \frac{n\,!}{(n-j)\,!} \left(\frac{T}{R}\right)^{j} + \left(\frac{n\,!}{(n-0)\,!} \left(\frac{T}{R}\right)^{0} - 1 \right) \right] \\ &= p_0 \frac{R}{T} \left[\sum_{j=0}^{n} \frac{n\,!}{(n-j)\,!} \left(\frac{T}{R}\right)^{j} - 1 \right] = p_0 \frac{R}{T} \left[\frac{1}{p_0} - 1 \right] \end{aligned}$$

Nous obtenons :

$$n - q = (1 - p_0) \frac{R}{T}$$

d'où :

$$W = \frac{qR}{n-q} = \frac{R}{(1-p_0)\frac{R}{T}} \left[n - (1-p_0)\frac{R}{T} \right] = \frac{nT}{1-p_0} - R$$

CHAPITRE 7

Exercice 1

```
extraire-message(m) : lire le message m dans casier ;
                      V(sprodcasier) ;
déposer-message(m)  : P(sprodcasier) ;
                      P(smutexcasier) ;
                      copier le message m dans casier ;
                      V(smutexcasier) ;
                      V(sservant) ;
```

Exercice 2

Il faut remarquer que le verrouillage et le déverrouillage, exécutés par un processus unique, sont en exclusion mutuelle.

a)

```
VERROUILLAGE
si ¬ verrou alors
    verrou := vrai ;
```

```
DEVERROUILLAGE
P(smutex-verrou) ;
si verrou alors
  début
  verrou := faux ;
  si facteur-bloque alors
    début
    facteur-bloque := faux ;
    V(sbloc)
    fin
  fin ;
V(smutex-verrou) ;
```

```
FACTEUR : P(ses) ;
          P(smutex-verrou) ;
          si ¬ verrou alors V(sbloc)
          sinon facteur-bloque := vrai ;
          V(smutex-verrou) ;
          P(sbloc) ;
```

Remarque. La section critique protégée par *smutex-verrou* évite la suite d'événements suivante :

— verrouillage,
— test de *verrou* par le *FACTEUR*, qui le trouve *vrai*,
— déverrouillage,
— blocage du *FACTEUR* derrière *sbloc*.

b)

```
VERROUILLAGE
si ¬ verrou alors
    début
    verrou := vrai ;
    P(sbloc)
    fin
```

```
DEVERROUILLAGE
si verrou alors
    début
    verrou := faux ;
    V(sbloc)
    fin
```

```
FACTEUR : P(ses) ;
          si verrou alors
             début
             V(ses ) ;
             P(sbloc) ;
             V(sbloc) ;
             aller à FACTEUR
             fin
```

Exercice 3

Les idées directrices sont les suivantes :

1) L'arrêt du transfert ne peut pas être instantané : on peut arrêter le *FACTEUR* (production), mais le *PILOTE* doit consommer tous les tampons produits de façon à ramener les sémaphores *sprod* et *scons* à leurs valeurs initiales.

2) On souhaite que l'opérateur ait l'impression que l'entrée-sortie est arrêtée dès l'introduction de la commande d'arrêt ; pour cela, on n'imprime effectivement une ligne que si un booléen *avort* est à la valeur *faux*.

3) Dans le schéma simplifié donné ici (un *SIO* pour chaque ligne), on peut se contenter d'un arrêt « à une ligne près » ; il est alors inutile d'arrêter physiquement une entrée-sortie (par l'instruction *HIO*).

D'où la solution suivante :

```
FACTEUR : P(ses)
          tant que (∃articles) et (¬ avort) faire
              début
              P(sprod) ;
                  .
                  .
                  .
              V(scons)
              fin ;
          avort := faux ;
          aller à FACTEUR ;

PILOTE :  P(scons) ;
          n := 0 ;
          tant que (n < nmax) et (¬ avort) faire
              début
                  .
                  .
                  .
              fin ;
VE :      V(sprod) ;
          aller à PILOTE ;
```

Dans l'analyse d'une réponse de l'opérateur, on commence par consulter *avort*.

```
si avort alors aller à VE
sinon si réponse opérateur = « continuer » alors ...
```

Dans le processus *SERVANT*, on réalise les actions suivantes après détection de la commande d'arrêt :

```
avort := vrai ;
si attend-op alors
    début
    attend-op := faux ;
    V(sop)
    fin
```

Remarque. Si on se contente de ce schéma, on peut demander des interventions superflues à l'opérateur, par exemple :

— entrée dans la boucle *tant que*,
— mise à *vrai* de *avort* par le processus *SERVANT*,
— détection d'une imprimante non opérationnelle par le *PILOTE*, et demande d'intervention de l'opérateur.

On peut pallier cet inconvénient en rendant indivisible le test d'*avort* et le lancement de la sortie (avec un sémaphore *savort* de valeur initiale *1*).

Exercice 4

Le message transmis dans la boîte aux lettres doit comporter le nom d'un sémaphore *sattente*, supposé initialement à la valeur *0*. Le processus demandeur attend la fin du transfert en exécutant *P(sattente)* ; lorsque le *PILOTE* a terminé la lecture du fichier sur disque, il effectue un *V(sattente)* avant de se brancher à *FACTEUR*.

BIBLIOGRAPHIE

Abate J., Dubner H.
69 : Optimizing the performance of a drum-like storage, *IEEE Trans. on Computers* C18, 11 (1969), pp. 992-997.

Abrial J. R.
72 : *Structures de données et de programmes*, Cours de Maîtrise d'Informatique, Université de Grenoble (1972).

Alderson A., Lynch W. C., Randell B.
72 : Thrashing in a multiprogrammed paging system, in *Operating Systems Techniques*, Hoare et Perrott, ed., Academic Press (1972), pp. 152-167.

Anderson G., Bertram K., Conn R., Malmquist K., Millstein R., Tokubo S.
68 : Design of a time-sharing system allowing interactive graphics. *Proc. ACM Nat. Conf.* (1968).

Arden B., Boettner D.
69 : Measurement and performance of a multiprogramming system, *Second Symposium on Operating Systems Principles*, Princeton (1969),

Avi-Itzhak B., Maxwell W. L., Miller L. W.
65 : Queuing with alternating priorities, *Operations Research* 13, 2 (1965). pp. 306-318.

Bacchus P., Pair C., Peccoud D., (éditeurs).
73 : *Manuel Algol 68*, par le groupe Algol de l'AFCET, à paraître chez Hermann.

Batson A., Shy-Ming Ju, Wood D. C.
70 : Measurements of segment size, *CACM* 13, 3 (1970), pp. 155-159.

Baudet G., Ferrié J., Kaiser C., Mossière J.
72 : Entrées-sorties dans un système à mémoire virtuelle, *Congrès AFCET*, Grenoble (1972).

Belady L. A.
66 : A study of replacement algorithms for a virtual storage computer, *IBM Syst. J.* 5, 2 (1966), pp. 78-101.
69 : An anomaly in space-time characteristics of certain programs running in paging machines, *CACM* 12, 6 (1969), pp. 349-363.

Berthaud M., Clauzel D., Jacolin M.
72 : *GSL : définition du langage*, Etude FF2.0133, Centre Scientifique IBM, Grenoble (1972).

Bétourné C.
70 : —, Boulenger J., Ferrié J., Kaiser C., Krakowiak S., Mossière J.
Notion d'espace virtuel dans le système ESOPE, *Congrès AFCET*, Paris.
72 : —, Krakowiak S.
Simulation de l'allocation de ressources dans un système conversationnel à mémoire virtuelle paginée, *Congrès AFCET*, Grenoble (1972).

Boulle C.
70 : Certains aspects de réalisation de SIRIS 8, *Congrès AFCET*, Paris (1970).

Brinch Hansen P.
70 : The nucleus of a multiprogrammed system, *CACM* 13, 4 (1970).
72 : A comparison of two synchronizing concepts, *Acta Informatica* 1, 3 (1972), pp. 190-199.

Brawn B. S., Gustavson F. G.
68 : Program behavior in a paging environment, *Proc. AFIPS FJCC* (1968), pp. 1019-1032.

Burroughs.
64 : *Burroughs B5500 Information Processing Systems Reference Manual*, Burroughs Corp. (1964).

Buxton J., Randell B., (editors).
70 : *Software engineering techniques* (Rome 1969), NATO Science Committee. (1970).

Cantrell H. N., Ellison A. L.
68 : Multiprogramming system performance measurement and analysis, *Proc. AFIPS SJCC* (1968).

Clark D., Graham R., Saltzer J., Schroeder M.
71a : *The classroom information and computing service*, MAC TR-80, MIT (1971).

Clark B. L., Horning J. J.
71b : The system language for project SUE, *ACM Sigplan Symposium on Languages for Systems Implementations*, Purdue University (1971).

Cleary J. C.
69 : Process handling on BURROUGHS B6500, *Proc. Fourth Australian Computer Conference*, Adelaïde (1969).

Coffman E. G.
68 : —, Kleinrock L.
Feedback queuing models for time-shared systems, *JACM* 15, 4 (1968), pp. 549-576.
71a : —, Elphick N. J., Shoshani A.
System deadlocks, *Computing Surveys* 3, 2 (1971).
71b : —, Randell B.
Performance predictions for extended paged memories, *Acta Informatica* 1, 1 (1971).
73 : —, Denning P. J.
Operating systems theory, Prentice-Hall (1973).

Comeau L.
67 : A study of the effect of user program optimization in a paging system, *First ACM Symposium on Operating Systems Principles*, Gatlinburg (1967).

Conway R. W., Maxwell W. L., Miller L. W.
67 : *Theory of Scheduling*, Addison-Wesley (1967).

Corbató F. J.
62 : An experimental time-sharing system, *Proc. AFIPS SJCC* (1962).
68 : *Sensitive issues in the design of multi-use systems*, MAC M-383, MIT (1968).
69a : A paging experiment with the MULTICS system, in *In Honor of P. M. Morse*, MIT Press (1969).
69b : PL/1 as a tool for system programming, *Datamation* (may 1969).

Courtois P. J., Heymans F., Parnas D. L.
71 : Concurrent control with « Readers » and « Writers », *CACM* 14, 10 (1971).

Creech B.
71 : Architecture of the B6500, *Infotech Report* 2, (1971).

Dennis, J. B. (editor).
70 : *Record of the Project MAC Conference on Concurrent Processes and Parallel Computation*, Woods Hole, ACM (1970).

De Meis W. M., Weizer N.
69 : Measurement and analysis of a demand paging time-sharing system, *Proc. 24th ACM Nat. Conf.* (1969).

Denning P. J.
67 : Effects of scheduling on file memory operations, *Proc. AFIPS SJCC* (1967), pp. 9-21.
68a : Thrashing, its causes and prevention, *Proc. AFIPS FJCC* (1968), pp. 915-922.
68b : *Resource allocation in a multiprocess computer system*, Ph. D. Thesis MAC TR-50, MIT (1968).
68c : The working set model for program behavior, *CACM* 11, 5 (1968), pp. 323-333.
70 : Virtual memory, *Computing Surveys* 2, 3 (1970).
71 : Third generation computer systems, *Computing Surveys* 3, 4 (1971).
72 : —, Schwartz S. C.
Properties of the working set model, *CACM* 15, 3 (1972).

Dijkstra E. W.
65 : Solution of a problem in concurrent programming control, *CACM* 8, 9 (1965).
67 : Cooperating sequential processes, in *Programming Languages*, F. Genuys, ed., Academic Press (1967).
68 : The structure of the THE multiprogramming system, *CACM* 11, 5 (1968).
71 : Hierarchical ordering of sequential processes, *Acta Informatica* 1, 2 (1971).
72 : Notes on structured programming, in *Structured Programming*, Dahl, Dijkstra, Hoare, APIC Studies in Data Processing, Academic Press (1972).

Evans T. G., Darley D. L.
66 : On-line debugging techniques : a survey, *Proc. AFIPS FJCC* (1966).

Ferrari D.
72 : Workload characterization and selection in computer performance measurement, *Computer* (July-August 1972).

Frank H.
69 : Analysis and optimization of disk storage devices for time-sharing systems, *JACM* 16, 4 (1969), pp. 602-620.

Freibergs I. F.
68 : The dynamic behavior of programs, *Proc. AFIPS FJCC* (1968), pp. 1163-1167.

Floyd R. W.
67 : Assigning meanings to programs, *Proc. Symposia in Applied Mathematics* 19, American Math. Soc. (1967), pp. 19-32.
71 : Towards the interactive design of correct programs, *Proc. IFIP Congress*, Ljubljana (1971).

Gaines R. S.
69 : *The debugging of computer programs*, Ph. D. Thesis, Princeton (1969).

Habermann A. N.
69 : Prevention of system deadlocks, *CACM* 12, 7 (1969), pp. 373-377.
72 : Synchronization of communicating processes, *CACM* 15, 3 (1972).

Hatfield D. J., Gerald J.
71 : Program restructuring for virtual memory, *IBM Syst. J.* 10, 3 (1971).

Hauck E., Dent. B.
68 : Burroughs B6500/B6700 stack mechanism, *Proc. AFIPS SJCC* (1968).

Havender J. W.
68 : Avoiding deadlocks in multitasking systems, *IBM Syst. J.* 7, 2 (1968), pp. 74-84.

Hoare C. A. R.
71 : Proof of a program, « FIND », *CACM* 14, 1 (1971).
72a : A survey of store management techniques, in *Operating Systems Techniques*, Hoare et Perrott, ed., Academic Press (1972), pp. 117-151.
72b : Towards a theory of parallel programming, in *Operating Systems Techniques*, Hoare et Perrott, ed., Academic Press (1972), pp. 61-71.

Holt, Richard C.
71 : *On deadlocks in computer systems*, Ph. D. Thesis, Cornell University (1971).

IBM
69 : *IBM System/360 Time-Sharing System, System Logic Summary PLM*, GY28-2009 (1969).
72 : *IBM System/370 Principles of Operations*, GA22-7000 (1972).

Ichbiah J. D., Rissen J. P., Héliard J. C.
72 : *Spécifications de définition de LIS*, Rapport STG-0-59T, CII (1972).

Knuth D. E.
66 : Additional comments on a problem in concurrent programming control, *CACM* 9, 5 (1966).
68 : *The art of computer programming*, vol 1, Addison Wesley (1968), pp. 435-451.

Lampson B. W.
71 : Protection, *Proc. Fifth Annual Princeton Conference on Information Sciences and Systems* (1971).

Le Faou J.
73 : *Mesure et analyse d'un système informatique*, Thèse de 3e cycle, Rennes (1973).

Leroudier J.
73 : *Une analyse de système*, Thèse de 3e cycle, Grenoble (1973).

Liptay J. S.
68 : Structural aspects of the System/360 model 85, II : the cache, *IBM Syst. J.* 7, 1 (1968).

Little J. D. C.
61 : A proof of the queuing formula $L = \lambda W$, *Operations Research* 9, 3 (1961).

Lucas H. C.
71 : Performance evaluation and monitoring, *Computing Surveys* 3, 3 (1971).

Mc Gee W. C.
65 : On dynamic program relocation, *IBM Syst. J.* 4, 3 (1965).

Mc Kinney J. M.
69 : A survey of analytical time-sharing models, *Computing Surveys* 1, 2 (1969).

Margolin B. H., Parmelee R. P., Schatzoff M.
71 : Analysis of free-storage algorithms, *IBM Syst. J.* 10, 4 (1971).

Mattson R. L., Gecsei J., Slutz D. R., Traiger I. L.
70 : Evaluation techniques for storage hierarchies, *IBM Syst. J.* 9, 2 (1970), pp. 78-117.

Mealy, G. H.
66 : The functional structure of OS/360, *IBM Syst. J.* 5, 1 (1966).

Mills, H. D.
72 : *Mathematical foundations for structured programming*, Report FSC 72-6012, IBM (1972).

Morris D., Detlefsen G. D.
69 : A virtual processor for real-time operation, in *Software engineering*, Tou, ed., Vol. 1, COINS Academic Press (1969).

Naur P., Randell B. (editors).
69 : *Software engineering techniques*, Garmisch (1968), NATO Science Committee.

Nielsen N. R.
67 : The simulation of time-sharing systems, *CACM* 10, 7 (1967).

Oppenheimer G., Weizer N.
68 : Resource management for a medium-scale time-sharing operating system, *CACM* 11, 5 (1968).

Organick E., Cleary J.
71 : A date structure model of the B6700 computer system, *ACM Sigplan Notices.*

Pair C.
71 : *Structures de données*, Ecole d'Eté AFCET, Alès (1971).
72 : *Compilation*, Ecole d'Eté AFCET, Neuchâtel (1972). (S'adresser à l'Université de Nancy).

Parnas D. L.
71 : A data structure model of the B6700 computer system, *ACM Sigplan Notices.* Ljubljana (1971).
72 : A technique for software module specification with examples, *CACM* 15, 5 (1972).

Patil S. S.
71 : *Limitations and capabilities of Dijkstra's semaphore primitives for coordination among processes*, Comp. Struct. Group Memo 57, Project MAC, MIT.

Pinkerton T. B.
68 : *Program behavior and control in virtual storage computer systems*, TR-4, University of Michigan, (1968).
Pirtle N. W.
67 : Intercommunication of processes and memory, *Proc. AFIPS FJCC* (1967).
Randell B.
—, Russell L. J.
64 : *Algol 60 Implementation*, Academic Press (1964).
—, Kuehner C. J.
68 : Dynamic storage allocation systems, *CACM* 11, 5 (1968), pp. 297-306.
69 : A note on storage fragmentation and program segmentation, *CACM* 12, 7 (1969), pp. 365-372.
Rosin R. F.
69 : Supervisory and monitor systems, *Computing Surveys*, 1, 1 (1969).
Rustin R. (editor).
71 : *Debugging techniques in large systems*, Courant Computer Sciences Symposium, 1, Prentice-Hall (1971).
Saal H. J., Riddle W. E.
70 : *Communicating semaphores*, SLAC, CGTM 117, Stanford (1970).
Saltzer J. H.
66 : *Traffic control in a multiplexed computer system*, Ph. D. Thesis, MAC-TR-30, MIT, Cambridge (1966).
70 : —, Gintell J. W.
The instrumentation of MULTICS, *CACM* 13, 8 (1970), pp. 495-500.
Scherr A. L.
65 : *An analysis of time-shared computer systems*, Ph. D. Thesis, MIT (1965), réimprimé par MIT Press (1971).
Schulman F. D.
67 : Hardware measurement device for IBM System/360, *Proc. ACM Nat. Conf.* (1967).
Schrage L. E.
66 : *Some queuing models for a time-shared facility*, Ph. D. Thesis, Cornell University (1966).
Schroeder M. D., Saltzer J. H.
72 : A hardware architecture for implementing protection rings, *CACM* 15, 3 (1972).
Shils A. J.
68 : *The load leveler*, IBM Research Report RC 2233 (1968).
Shoshani A., Coffman E. G.
70 : Prevention, detection and recovery from system deadlocks, *Proc. Fourth Princeton Conf. on Information Sciences and Systems* (1970).
Simon H. A.
63 : Experiments with a heuristic compiler, *CACM* 10, 4 (1963), pp. 493-506.
Snowdon R. A.
71 : *PEARL : an interactive system for the preparation and validation of structured programs*, TR-28, Computing Lab., University of Newcastle upon-Tyne.

Spirn J. R., Denning P. J.
72 : Experiments with program locality, *Proc. AFIPS FJCC* (1972).

Takaćs L.
62 : *Introduction to the theory of queues*, Oxford University Press (1962).

Teorey T. J., Pinkerton T. B.
72 : A comparative analysis of disk scheduling policies, *CACM* 15, 3 (1972).

Trilling L.
73 : Traitement des procédures en ALGOL 68, *Journées ALGOL 68*, Université de Paris VI (1973).

Vantilborgh H., Van Lamsveerde A.
72 : On an extension of Dijkstra's semaphore primitives, *Inf. Proc. Letters*, 1 (1972).

Verjus J. P.
73 : Relations entre un programme ALGOL 68 et l'environnement, *Journées ALGOL 68*, Université de Paris VI (1973).

Watson R. W.
70 : *Timesharing system design principles*, Mc Graw-Hill (1970).

Wegner P.
71 : Data structure models for programming languages, *ACM Sigplan Notices* (1971).

Weinberg G.
71 : *The psychology of computer programming*, Van Nostrand (1971).

Wilkes M. V.
68 : *Time sharing computer systems*, Macdonald (1968).
71 : Automatic load adjustment in time-sharing systems, in *Séminaires IRIA, Structure et Programmation des calculateurs* (1971), pp. 127-138.

Wirth N.
68 : PL/360, a programming language for the 360 computers *JACM*, 15, 1 (1968).
69 : On multiprogramming, machine coding and computer organization, *CACM* 12, 9 (1969).
71a : The programming language PASCAL, *Acta Informatica* 1, 2 (1971).
71b : Program development by stepwise refinement, *CACM* 14, 4 (1971).
73 : *Systematic programming, an introduction*, Prentice Hall (1973).

Wulf W. A.
69 : Performance monitors for multiprogramming systems, *Second Symposium on Operating Systems Principles*, Princeton (1969).
71 : —, Russell D. B., Habermann A. N.
BLISS, a language for systems programming, *CACM* 14, 12 (1971).

Zurcher F. W., Randell B.
68 : Iterative multi-level modelling : a methodology for computer system design, *Proc. IFIP Congress*, Edinburgh (1968).

Complément bibliographique pour la seconde édition

1. OUVRAGES GÉNÉRAUX

Brinch Hansen P.
73 : *Operating systems principles*, Prentice-Hall (1973).

Graham R. M.
75 : *Principles of systems programming*, Wiley (1975).

Habermann A. N.
76 : *Introduction to operating systems design*, Science Research Associates (1976).

Madnick S. E., Donovan J. J.
74 : *Operating systems*, Mc Graw-Hill (1974).

Shaw A. C.
74 : *The logical design of operating systems*, Prentice-Hall (1974).

[Graham 75] présente une introduction générale à la conception du logiciel de base, et contient un exposé élémentaire mais complet sur les systèmes d'exploitation. [Brinch Hansen 73], [Shaw 74] et [Habermann 76] mettent l'accent sur les principes de conception des systèmes d'exploitation et contiennent également des études de cas. [Madnick 74] est plutôt orienté vers les aspects technologiques des systèmes d'exploitation, illustrés par des exemples tirés de systèmes IBM.

2. ÉTUDES DE CAS

Banino J. S., Ferrié J., Kaiser C., Lanciaux D.
78 : Contrôle de l'accès aux objets partagés dans les systèmes informatiques, Monographies AFCET (1978).

Brinch Hansen P.
77 : The architecture of concurrent program. Prentice Hall (1977).

Cohen E., Jefferson D.
75 : Protection in the HYDRA operating system, *Proc. 5th ACM Symposium on Operating Systems Principles*, Austin (1975).

England D. M.
75 : Capability concept, mechanism and structure in system 250, *RAIRO Informatique*, B-3 (1975), pp. 47-62.

IBM
73 : Functional structure of IBM virtual storage operating systems (3 articles), *IBM Syst. J.* 4, (1973), pp. 368-411.

Lauesen S.
75 : A large semaphore-based operating system, *CACM* 18, 7 (1975), pp. 377-389.

Ritchie D. M., Thompson K.
74 : The UNIX time-sharing system, *CACM* 17, 7 (1974).

Whitfield H., Wight A. S.
74 : EMAS : The Edinburgh Multi-Access System, *Computer J.* 16, 4 (1974), pp. 331-346.

Wulf W., Levin R., Pierson C.
75 : An overview of the HYDRA operating system, *Proc. 5th ACM Symposium on Operating Systems Principles*, Austin (1975).

[Brinch Hansen 77] décrit un langage de haut niveau conçu pour l'écriture de systèmes et illustre son usage par plusieurs études de cas traitées en détail. [Ritchie 74] décrit un système d'exploitation fonctionnant en temps partagé sur un mini-ordinateur. L'étude des systèmes HYDRA [Cohen 75, Wulf 75] et Plessey 250 [England 75] illustre la mise en œuvre de schémas de protection programmés ou câblés utilisant des descripteurs (« capability »). Une étude de synthèse sur l'adressage et la protection dans les systèmes, illustrée par l'analyse des systèmes MULTICS et Plessey 250, est présentée dans [Banino 78]. [Lauesen 75] donne un exemple d'utilisation des sémaphores dans la construction d'un système d'exploitation. On trouvera dans [IBM 73] et [Whitfield 74] l'application de méthodes d'auto-adaptation pour la gestion d'un système multiprogrammé. [Whitfield 74) contient en outre des résultats de mesure sur le dispositif de régulation.

77 : The architectural of concurrent program. Prentice Hall (1977).

Classement par chapitre des références bibliographiques

CHAPITRE 1

Rosin, 69 ; Watson, 70 ; Wilkes 68.

CHAPITRE 2

Baudet, 72 ; Boulle, 70 ; Brinch Hansen (70, 72) ; Cleary, 69 ; Courtois, 71 ; Dijkstra (65, 67, 68, 71) ; Habermann, 72 ; Hoare, 72b ; Knuth, 66 ; Morris, 69 ; Patil, 71 ; Saal, 70 ; Saltzer, 66 ; Vantilborgh, 72 ; Wirth, 69.

CHAPITRE 3

Abrial, 72 ; Anderson, 68 ; Bacchus, 73 ; Bétourné, 70 ; Clark, 71a ; Cleary, 69 ; Creech, 71 ; Hauck, 68 ; Organick, 71 ; Pair (71, 72) ; Randell, 64 ; Trilling, 73 ; Verjus, 73 ; Wegner, 71.

CHAPITRE 4

Abate, 69 ; Alderson, 72 ; Arden, 69 ; Batson, 70 ; Belady, 66 ; Belady, 69 ; Brawn, 68 ; Coffman (71, 73) ; Comeau, 67 ; Corbatò (62, 69a) ; Denning (68a, 68b, 68c, 70) ; Dijkstra, 67 ; Freibergs, 68 ; Habermann, 68 ; Hatfield, 71 ; Havender, 68 ; Hoare, 72a ; Holt, 71 ; IBM (69, 72) ; Knuth, 68 ; Le Faou, 73 ; Leroudier, 73 ; Liptay, 68 ; Mc Gee, 65 ; Margolin, 71 ; Mattson, 70 ; Oppenheimer, 68 ; Randell (68, 69) ; Scherr, 67 ; Shils, 68 ; Shoshani, 70 ; Spirn, 72 ; Wilkes, 71 ; Wulf 69.

CHAPITRE 5

Denning, 71 ; Lampson, 71 ; Schroeder, 72 ; Watson, 71 ; Wilkes, 68.

CHAPITRE 6

Abate, 69 ; Arden, 69 ; Avi-Itzhak, 65 ; Bétourné, 72 ; Cantrell, 68 ; Coffman (68, 73) ; Conway, 67 ; De Meis, 69 ; Denning (67, 68a, 72) ; Ferrari, 72 ; Frank, 69 ; Little, 61 ; Lucas, 71 ; Mc Kinney, 69 ; Mattson, 70 ; Nielsen, 67 ; Oppenheimer, 68 ; Pinkerton, 68 ; Pirtle, 67 ; Saltzer, 70 ; Scherr, 65 ; Schulman, 67 ; Takacs, 62 ; Teorey, 72 ; Wulf, 69.

CHAPITRE 7

Baudet, 72 ; Berthaud, 72 ; Brinch Hansen, 70 ; Burroughs, 64 ; Buxton, 70 ; Clark, 71b ; Corbato (68, 69b) ; Dennis, 70 ; Dijkstra (68, 72) ; Evans, 66 ; Floyd (67, 71) ; Gaines, 69 ; Hoare, 71 ; Ichbiah, 72 ; Mealy, 66 ; Mills, 72 ; Naur, 69 ; Parnas (71, 72) ; Rustin, 71 ; Schrage, 66 ; Simon, 63 ; Snowdon, 71 ; Weinberg, 71 ; Wirth (68, 71a, 71b, 73) ; Wulf, 71 ; Zurcher, 68.

INDEX

Imprimé en France. — Imprimerie JOUVE, 17, rue du Louvre, 75001 PARIS
Dépôt légal : 3e trimestre 1978

Extrait de notre catalogue :

ANALYSE ORGANIQUE

Informatique de gestion

par C. COCHET et A. GALLIOT

Tome 1 : Les données : 156 pages, 15,5 × 24, broché.
Tome 2 : Les traitements : 148 pages, 15,5 × 24, broché.

Reflet d'une longue expérience pratique des auteurs, le thème essentiel de cet ouvrage est l'aspect organique de l'analyse en Informatique de gestion.

Le premier tome expose les principaux problèmes de l'analyse et de la codification. Au travers de l'étude des fichiers et des tables, il traite aussi de l'organisation physique et logique des données.

Le deuxième tome explique comment architecturer les traitements pour que l'application automatisée atteigne pleinement son but. Il développe également les aspects relatifs à la mise en œuvre des chaînes et au contenu des dossiers.

Exposé abordable et efficace, l'ouvrage est marqué par un souci de progressivité et de pédagogie qui le destine à de nombreux lecteurs.

Les utilisateurs, présents ou futurs, de l'informatique y trouveront des explications claires sur les processus de conception et de réalisation d'une application automatisée.

Les analystes, débutants ou confirmés y puiseront les uns de nouvelles connaissances, les autres une assise plus large pour leur expérience.

Aide-mémoire Dunod

INFORMATIQUE

par Charles BERTHET

256 pages, Format poche 13 × 18, broché

Le plan de cet aide-mémoire se réfère à la pratique quotidienne de l'informaticien : après avoir resitué la place de l'informatique dans le champ plus vaste des systèmes d'information des organisations, Charles Berthet a choisi de regrouper les éléments les plus fréquemment utiles, en éliminant les concepts trop théoriques et peu mis en œuvre, ou trop spécialisés.

Il convenait d'abord de préciser la représentation et l'organisation des données. L'auteur a donc rappelé la façon dont les principaux constructeurs ont résolu ce problème. Puis il a résumé les concepts essentiels relatifs aux fichiers, et, plus généralement, aux banques de données.

La pratique de l'informatique est, concrètement, celle de la programmation. C'est pourquoi une part relativement importante de ce livre concerne les trois langages les plus utilisés : *Fortran*, *Cobol* et *PL/1*. Le développement de la téléinformatique est considérable, parce que vital : un chapitre a été consacré à ses techniques particulières.

Essentiellement pratique, cet aide-mémoire est pour l'informaticien un guide et une référence précieux, un compagnon indispensable.

Sommaire