

systèmes d'exploitation des ordinateurs

8° ca 26 80

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 2^e 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

© BORDAS, Paris, 1975 — n° 0316 780 207
ISBN 2-04-001445-4

"Toute représentation ou reproduction, intégrale ou partielle, faite sans le consentement de l'auteur, ou de ses ayants-droit, ou ayants-cause, est illicite (loi du 11 mars 1957, alinéa 1^{er} de l'article 40). Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait une contrefaçon sanctionnée par les articles 425 et suivants du Code pénal. La loi du 11 mars 1957 n'autorise, aux termes des alinéas 2 et 3 de l'article 41, que les copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective d'une part, et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration".

DUNOD

phase spécialité

informatique

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. Béourné (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 particulissent 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.

VIII Systèmes d'exploitation des ordinateurs

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 (récurivité, 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 PRÉREQUISSES

- Arsac J., *Les systèmes de conduite des ordinateurs*, Dunod (2^e 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

Avant-propos	V
Avant-propos de la seconde édition	VI
Préambule	VII
CHAPITRE 1. Introduction	1
1.1 <i>Fonctions et aspects externes des systèmes</i>	1
1.11 Fonctions d'un système	1
1.12 Aspects externes des systèmes	2
1.2 <i>Caractéristiques communes</i>	2
1.21 Partage des ressources physiques	3
1.22 Gestion de l'information	5
1.23 Coopération des processus	7
1.24 Protection	8
1.3 <i>Problèmes de conception et d'évaluation</i>	9
1.31 Mesures et modèles de systèmes	9
1.32 Méthodologie de conception	9
1.4 <i>Organisation de l'ouvrage</i>	10
CHAPITRE 2. Les processus	11
2.1 <i>Introduction</i>	11
2.2 <i>Définitions</i>	12
2.21 Instructions. Processeur. Processus	12
2.22 Notion de ressource. États des processus	13
2.221 Ressources et états des processus	13
2.222 Accès aux ressources	14
2.223 Pouvoir d'un processus	15
2.224 Contenu du vecteur d'état	15
2.23 Relations entre processus	16
2.231 Création et destruction	16
2.232 Synchronisation et communication	17
2.24 Exemple de décomposition en processus	17

X Table des matières

2.3 <i>Exclusion mutuelle</i>	18
2.31 Introduction au problème	18
2.32 Attente active	19
2.33 Les verrous	21
2.34 Les sémaphores	22
2.341 Définition	22
2.342 Propriétés des sémaphores	22
2.343 Sémaphores d'exclusion mutuelle	24
2.35 Difficultés de l'exclusion mutuelle	25
2.4 <i>Mécanismes de synchronisation</i>	26
2.41 Généralités	26
2.42 Mécanismes d'action directe	27
2.43 Mécanismes d'action indirecte	28
2.431 Synchronisation par événements	28
2.432 Synchronisation par sémaphores	30
2.44 Critique des mécanismes de synchronisation	35
2.5 <i>Communication entre processus</i>	36
2.51 Introduction	36
2.52 Communication entre processus par variables communes	36
2.521 Modèle du producteur et du consommateur	37
2.522 Communication par boîte aux lettres	42
2.53 Mécanismes spéciaux de communication	44
2.531 Sémaphores avec messages	44
2.532 Communication entre processus dans le système MUS	45
2.6 <i>Implantation des primitives de synchronisation</i>	47
2.61 Exclusion mutuelle dans les primitives	47
2.62 Gestion des processus	48
2.63 Protection des primitives	49
2.64 Exemples	49
2.7 <i>Problèmes de protection</i>	55
2.71 Les problèmes	55
2.72 Quelques remèdes	56
2.8 <i>Exemple de coopération de processus</i>	58
EXERCICES	58

	<i>Table des matières</i>	XI
CHAPITRE 3. Gestion de l'information		
3.1 Introduction	67	
3.11 Terminologie	67	
3.111 Représentation externe des objets	68	
3.112 Représentation interne des objets	69	
3.113 Objets composés	71	
3.114 Durée de vie des objets	72	
3.115 Notion de segment	73	
3.116 Procédure	73	
3.12 Contraintes apportées par le système	73	
3.121 Partage des objets et utilisation des noms	74	
3.122 Interférence avec la gestion des ressources physiques	74	
3.123 Représentation du système	75	
3.13 Modifications de la chaîne d'accès à un objet	75	
3.131 Objets liés dès la compilation	76	
3.132 Noms et objets libres après compilation	76	
3.2 Gestion des noms dans le système CLICS	77	
3.21 Introduction	77	
3.22 La mémoire segmentée	78	
3.221 Désignation d'un segment par un processus	80	
3.222 Descriptif des segments d'un processus	81	
3.23 Langage machine et objets manipulés	82	
3.231 Format des instructions	82	
3.232 Les différents objets manipulés par l'exécution d'une procédure	82	
3.233 Multiplicité des objets	84	
3.24 Accès aux objets	84	
3.241 Accès aux étiquettes du segment-procédure	85	
3.242 Accès aux objets rémanents	86	
3.243 Accès aux objets externes	88	
3.244 Accès aux objets locaux	89	
3.245 Accès aux paramètres	90	
3.246 Illustration des mécanismes d'accès	92	
3.25 Appel et retour de procédure	92	
3.251 Calcul de l'adresse segmentée des paramètres effectifs	94	
3.252 Appel de procédure et changement de contexte	94	
3.253 Retour à la procédure appelante	96	
3.26 Liaisons dynamiques	97	
3.261 Remplacement de l'identificateur par un nom de segment : édition de liens	98	

	<i>XII Table des matières</i>	
3.262 Référence à un segment-procédure	99	
3.263 Catalogue des segments connus et catalogue général	101	
3.264 Gestion du descriptif	101	
3.3 Gestion des noms dans le système BURROUGHS B 6700	102	
3.31 Introduction	102	
3.32 Le matériel	102	
3.321 Notion de préfixe	102	
3.322 Les segments	103	
3.323 Les processeurs physiques	103	
3.33 Représentation des objets du langage	104	
3.331 Objets simples	104	
3.332 Tableaux	105	
3.333 Objets-procédures	105	
3.34 Accès aux objets	105	
3.341 Aspects lexicographiques	105	
3.342 L'espace adressable	106	
3.343 Environnement : accès par désignation	107	
3.344 Accès aux paramètres effectifs : noms dynamiques	109	
3.35 Procédures	109	
3.36 Variations d'environnement aux appels et retours de procédures	111	
3.361 Appel de procédure	111	
3.362 Retour de procédure	113	
3.363 Chaîne statique	114	
3.364 Zone de liaison	114	
3.365 Détail de l'appel de procédure	114	
3.37 Partage des objets entre un processus père et ses processus fils	116	
3.371 Création de processus	116	
3.372 Existence d'objets communs aux processus père et fils	117	
3.373 Incidence sur les noms	118	
3.374 Synchronisation	119	
3.38 Inclusion du moniteur dans l'arborescence de piles	120	
3.381 Partage des objets par une collection de processus	120	
3.382 Partage des objets communs à tous les processus	120	
3.383 Partage des procédures communes à plusieurs processus	121	
3.4 Gestion de l'information dans le système ESOPE	122	
3.41 Le matériel	124	
3.411 La mémoire physique	124	
3.412 L'adressage topographique	124	

Table des matières XIII

3.42 La mémoire adressable	125
3.421 L'espace des segments	125
3.422 La mémoire virtuelle.....	125
3.43 Désignation des segments	125
3.44 Accès à l'information : le couplage.....	127
3.441 Principe du couplage	127
3.442 Réalisation du couplage	128
3.443 Contraintes.....	130
3.45 Partage des segments	130
3.46 Utilisation des mécanismes de gestion de l'information.....	131
3.5 <i>Représentation et gestion des objets</i>	131
3.51 Représentation des objets	132
3.511 Généralités	132
3.512 Décomposition de la représentation d'un objet.....	133
3.513 Partage d'un objet.....	133
3.52 Accès aux objets	135
3.521 Nom d'un objet	135
3.522 La mémoire fictive	137
3.523 Espace adressable d'un processus	137
3.524 L'environnement d'un processus.....	137
EXERCICES	138
CHAPITRE 4. Gestion des ressources	141
4.1 <i>Notions générales</i>	141
4.11 Exemples de ressources	142
4.12 Représentation des ressources	143
4.13 Origine et forme des demandes d'allocation.....	144
4.14 Fonctions de l'allocateur	145
4.141 Généralités	145
4.142 Traitement d'une demande	147
4.15 Présentation du chapitre	148
4.2 <i>Caractéristiques de la charge d'un système</i>	148
4.21 Introduction	148
4.22 Caractéristiques globales de la charge.....	149
4.23 Comportement dynamique des programmes. Propriété de localité.....	154

XIV *Table des matières*

4.3 <i>Allocation de processeur réel</i>	158
4.31 Classification des stratégies	158
4.32 Stratégies sans recyclage des travaux	159
4.33 Stratégies avec recyclage des travaux	160
4.34 Stratégies fondées sur la notion de priorité	162
4.35 Stratégies fondées sur la notion d'échéance	162
4.4 <i>Gestion de la mémoire principale</i>	163
4.41 Introduction	163
4.42 Incidence des mécanismes d'adressage	164
4.43 Stratégies d'allocation de la mémoire aux travaux	165
4.431 Allocation en mémoire uniforme	165
4.432 Allocation en mémoire hiérarchisée	167
4.44 Gestion de la mémoire par zones	168
4.441 Réimplantation dynamique par registres de base	168
4.442 Algorithmes de gestion de la mémoire par zones	169
4.45 Gestion de la mémoire par pages	172
4.451 Mécanismes de pagination	172
4.452 Représentation des espaces virtuels dans le système	178
4.453 Stratégie d'allocation des cases	179
4.454 Algorithmes de remplacement	180
4.455 Conclusions	182
4.5 <i>Gestion de la mémoire secondaire</i>	182
4.51 Introduction	182
4.52 Caractéristiques physiques des unités	183
4.53 Gestion de la mémoire auxiliaire	184
4.531 Allocation d'espace	184
4.532 Gestion des transferts pour un tambour	187
4.533 Gestion des transferts pour un disque	190
4.54 Gestion de la mémoire externe	192
4.6 <i>Stratégies globales</i>	193
4.61 Phénomène d'écoulement du système	193
4.62 Régulation de la charge	196
4.63 Stratégies fondées sur l'espace de travail	198
4.631 Notion d'espace de travail	198
4.632 Mise en œuvre de stratégies fondées sur l'espace de travail	199
4.633 Détermination pratique de l'espace de travail	199
4.634 Tentatives d'adaptation du comportement des programmes	200

<i>Table des matières</i>	XV
4.7 Interblocage	201
4.71 Introduction	201
4.72 Description informelle	202
4.73 Formalisation et définitions.....	203
4.731 Système. Etat d'un système	203
4.732 Interblocage	206
4.74 Remèdes à l'interblocage	208
4.741 Détection. Guérison	208
4.7411 Détection	208
4.7412 Guérison	212
4.742 Prévention	212
4.7421 Prévention statique	213
4.7422 Prévention dynamique	214
4.75 Conclusions	217
EXERCICES	217
CHAPITRE 5. Protection	223
5.1 Présentation du problème	223
5.11 Introduction	223
5.12 Position du problème	224
5.121 Définitions	224
5.122 Limites du système de protection.....	225
5.123 Variation du pouvoir d'un utilisateur : nécessité et limites	226
5.124 Problème à résoudre	227
5.13 Exemples d'implantation de la matrice des droits.....	227
5.131 Liste d'accès	227
5.132 Liste des droits	228
5.133 Clés et verrous	228
5.134 Mode maître, mode esclave	228
5.2 Mécanismes de protection dans le système ESOPE	229
5.21 Rappels sur le matériel CII 10070.....	229
5.22 La protection dans le système ESOPE	230
5.221 Utilisation des segments	230
5.222 Protection de la mémoire virtuelle d'un usager.....	230
5.223 Pouvoir des processus	231
5.224 Changement du pouvoir d'un processus de l'usager.....	232
5.3 Mécanisme de protection dans le système MULTICS	234
5.31 Introduction	234
5.32 Définition et portée des anneaux	234

<i>XVI Table des matières</i>	
5.33 Changement du pouvoir d'un processus. Nécessité du guichet..	236
5.331 Augmentation de pouvoir	236
5.332 Conservation du pouvoir	237
5.333 Diminution de pouvoir	238
5.34 Implantation câblée des anneaux de protection	239
5.341 Le descripteur de segment	239
5.342 Exécution d'une instruction	240
5.343 Appel et retour de procédure	243
5.3431 Instruction CALL. Passation des paramètres	244
5.3432 Instruction RETURN. Détermination de l'anneau de retour	246
5.35 Conclusions	249
EXERCICES	249
CHAPITRE 6. Mesures et modèles de systèmes	251
6.1 Introduction	251
6.11 Intérêt et importance des études quantitatives	251
6.12 Méthodes de mesure et d'évaluation	253
6.2 Les modèles de système	253
6.21 Les objectifs des modèles	253
6.22 Exemples de modèles analytiques	254
6.221 Echange de pages avec un disque à têtes fixes	254
6.222 Un modèle d'allocation de processeur	257
6.223 Un modèle de système conversationnel	262
6.23 Exemples de simulation	265
6.3 Mesures sur les systèmes réels	269
6.31 Nature des mesures	269
6.32 Méthodologie des mesures	269
6.33 Mécanismes de mesure	270
6.331 Généralités	270
6.332 Appareillage de mesure externe	270
6.333 Mécanismes câblés internes au système	272
6.334 Mesures programmées	273
6.34 Utilisation des mesures	274
6.341 Evaluation des systèmes	274
6.342 Amélioration des performances	275
EXERCICES	276

Table des matières XVII

CHAPITRE 7. Méthodologie de conception et de réalisation	279
7.1 Introduction	279
7.2 Validité des programmes	281
7.3 Programmation structurée.....	284
7.31 Programmes séquentiels	285
7.311 Modules	285
7.312 Niveaux.....	287
7.32 Programmes parallèles	289
7.4 Outils d'écriture et de mise au point.....	290
7.41 Langages d'écriture de systèmes	290
7.411 Caractéristiques des langages	291
7.412 Classification des langages	292
7.42 Outils de mise au point	293
7.43 Technologie de la programmation	295
7.5 Exemple : réalisation d'un système d'entrée-sortie	296
7.51 Spécification du module d'entrée-sortie.....	296
7.511 La machine de base	297
7.512 Conséquence de l'extension souhaitée.....	297
7.513 L'interface du module d'entrée-sortie.....	298
7.514 Choix laissés au réalisateur du module d'entrée-sortie ..	299
7.52 Conception du module d'entrée-sortie.....	300
7.521 Décomposition.....	300
7.522 Interfaces	301
7.523 Conception des différents modules	302
7.524 Récapitulation	306
EXERCICES	308
SOLUTIONS DES EXERCICES	309
BIBLIOGRAPHIE	349
INDEX	357

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

1.1.1 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

2 Systèmes d'exploitation des ordinateurs

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.

1.1.2 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 ».

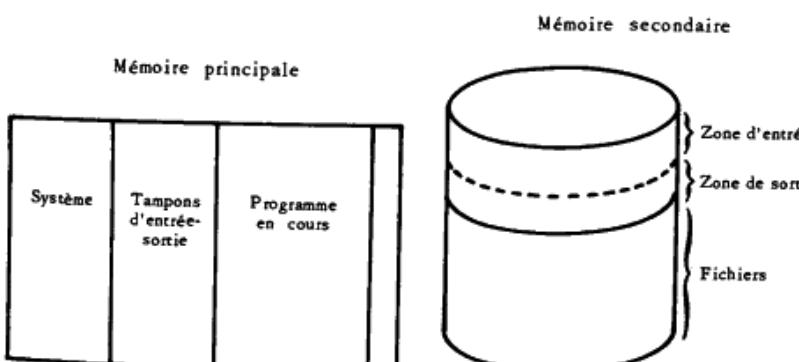


Figure 1. Exemple de partage de la mémoire.

4 Systèmes d'exploitation des ordinateurs

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'éroulement, 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 identificateur 273.

①-----

! Assign bib, fil, (nam,f4lib), (sts,old)

Assignation au identificateur *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 identificateur *bib* ; placer le résultat (module de chargement) dans le fichier *go* (option par défaut).

②-----

6 Systèmes d'exploitation des ordinateurs

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

③-----

! Run

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 identificateur 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 identificateur 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 identificateurs 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.

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

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

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

8 Systèmes d'exploitation des ordinateurs

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.

10 Systèmes d'exploitation des ordinateurs

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.

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.

12 Systèmes d'exploitation des ordinateurs

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. déroulement 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.

14 Systèmes d'exploitation des ordinateurs

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.

Exampons 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* ≥ 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 grossièrement 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 Crédit 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.

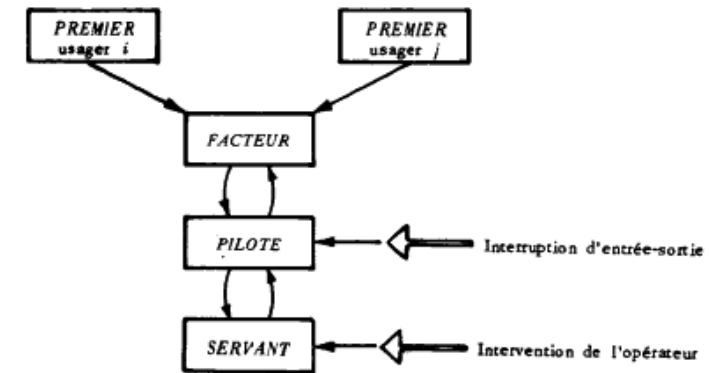
Pour exploiter au mieux le parallélisme entre les différentes unités du système, on introduit un processus pour chaque unité fonctionnant de manière autonome, c'est-à-dire :

- 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 ;

20 Systèmes d'exploitation des ordinateurs

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.

22 Systèmes d'exploitation des ordinateurs

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 ;
               $\text{état } (r) := \underline{\text{bloqué}}$ ;
              mettre le processus r dans la file f(s)
          fin;
      fin;
V(s) : début
       $e(s) := e(s) + 1$ ;
      si  $e(s) \leq 0$  alors
          début
              sortir un processus de la file f(s) ;
              commentaire : soit q le processus sorti ;
               $\text{état } (q) := \underline{\text{actif}}$ 
          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 :

$$nf(s) \leq np(s)$$

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

$$P(s) : np(s) := np(s) + 1;$$

si $np(s) \leq e0(s) + nv(s)$ alors $nf(s) := nf(s) + 1$;

commentaire : c'est le cas où $e(s) \geq 0$;

$$V(s) : nv(s) := nv(s) + 1;$$

si $np(s) \geq e0(s) + nv(s)$ alors $nf(s) := nf(s) + 1$;

commentaire : c'est le cas où $e(s) \leq 0$;

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

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

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
$np < e0 + nv \begin{cases} nf = np \\ nf < e0 + nv \end{cases}$	$np \leq e0 + nv$	$nf := nf + 1$	$nf = np$ $nf \leq e0 + nv$
$np \geq e0 + nv \begin{cases} nf = e0 + nv \\ nf \leq np \end{cases}$	$np > e0 + nv$	<i>pas d'effet</i>	$nf = e0 + nv$ $nf < np$

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
$np > e0 + nv \begin{cases} nf = e0 + nv \\ nf < np \end{cases}$	$np \geq e0 + nv$	$nf := nf + 1$	$nf = e0 + nv$ $nf \leq np$
$np \leq e0 + nv \begin{cases} nf = np \\ nf \leq e0 + nv \end{cases}$	$np < e0 + nv$	<i>pas d'effet</i>	$nf = np$ $nf < e0 + nv$

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 :

$$np(s) = nv(s) = nf(s) = 0$$

$$e0(s) \geq 0.$$

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)

$$\begin{aligned} e(s) &= e0(s) - np(s) + nv(s) \\ \text{si } e(s) < 0, \text{ on a } e0(s) + nv(s) &< np(s) \end{aligned}$$

(1) donne alors :

$$\begin{aligned} nf(s) &= e0(s) + nv(s) \\ \text{et } -e(s) &= np(s) - nf(s) \end{aligned}$$

- 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 :

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

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 :

$$(2) \quad nf(mutex) = nv(mutex)$$

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

$$(3) \quad nf(mutex) < np(mutex)$$

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 réservier 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'*up* 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 ininteruptibles. 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) = vrai.

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  $\neg$  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) Événement 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) Événement 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 pl;
```

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* ≤ 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.

<i>sémaphore signal : (valeur initiale = 0)</i>	
<i>processus p : début</i>	<i>processus q : début</i>
$A_1 ; \dots ; A_i ;$	$B_1 ; \dots ; B_j ;$
$P(signal) ;$	$V(signal) ;$
$\dots ;$	$\dots ;$
$fin ;$	$fin ;$

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 A_i) 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 A_i , 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. À 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) ;

```

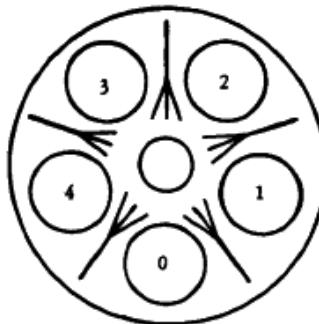
L'exemple suivant va nous permettre de préciser ces schémas.

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 docile 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

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

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.

Événements 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

PROD : Produire un message ;
Déposer un message
dans le tampon ;
aller à PROD ;

CONSOMMATEUR

CONS : Prélever un message
dans le tampon ;
Consommer le message ;
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) :

n_{plein} = nombre de messages attendant d'être prélevés,
 n_{vide} = nombre d'emplacements disponibles dans le tampon.

Initialement, $n_{\text{plein}} = 0$, $n_{\text{vide}} = n$.

L'algorithme des deux processus s'écrit alors :

PRODUCTEUR

entier $n_{\text{plein}} = 0$, $n_{\text{vide}} = n$;

PROD : Produire un message ; CONS : $\left\{ \begin{array}{l} n_{\text{plein}} := n_{\text{plein}} - 1; \\ \text{si } n_{\text{plein}} = -1 \text{ alors} \\ \quad \text{attendre} \end{array} \right\}$
 $\left\{ \begin{array}{l} n_{\text{vide}} := n_{\text{vide}} - 1; \\ \text{si } n_{\text{vide}} = -1 \text{ alors} \\ \quad \text{attendre} \end{array} \right\}$
 Déposer le message ;
 $n_{\text{plein}} := n_{\text{plein}} + 1$;
 $\left\{ \begin{array}{l} \text{si consommateur en attente alors} \\ \quad \text{réveiller consommateur} \\ \quad \text{aller à PROD} \end{array} \right\}$
 $\left\{ \begin{array}{l} n_{\text{vide}} := n_{\text{vide}} + 1 \\ \text{si producteur en attente alors} \\ \quad \text{réveiller producteur} \\ \quad \text{aller à CONS} \end{array} \right\}$

CONSOMMATEUR

$\left\{ \begin{array}{l} n_{\text{plein}} := n_{\text{plein}} - 1; \\ \text{si } n_{\text{plein}} = -1 \text{ alors} \\ \quad \text{attendre} \end{array} \right\}$

Prélever un message ;
 $n_{\text{vide}} := n_{\text{vide}} + 1$;
 $\left\{ \begin{array}{l} \text{si producteur en attente alors} \\ \quad \text{réveiller producteur} \\ \quad \text{aller à CONS} \end{array} \right\}$

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 $n_{plein} = -1$ qui lui est alors équivalente (en fait, on compare n_{plein} à 0 puisqu'on a fait $n_{plein} = n_{plein} + 1$). Le test et la modification de cette condition se font comme suit :

PRODUCTEUR	CONSOMMATEUR
:	:
$\left\{ \begin{array}{l} n_{plein} := n_{plein} + 1; \\ \text{si } n_{plein} = 0 \text{ alors réveiller} \end{array} \right\}$	$\left\{ \begin{array}{l} n_{plein} := n_{plein} - 1; \\ \text{si } n_{plein} = -1 \text{ alors} \\ \quad \text{attendre}; \end{array} \right\}$
:	:
:	:

Remarquons enfin que les conditions $n_{plein} = 0$ et $n_{plein} = -1$ entraînent respectivement $n_{plein} \leq 0$ et $n_{plein} < 0$. On voit alors que n_{plein} 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 n_{vide} et le producteur. L'algorithme des deux processus peut maintenant s'écrire :

<u>sémaphore</u> $n_{plein} = 0, n_{vide} = n;$ PROD : Produire un message ; <i>P</i> (n_{vide}); <i>Déposer le message</i> ; <i>V</i> (n_{plein}); <u>aller à PROD</u> ;	CONS : $P(n_{plein})$; <i>Prélever un message</i> ; <i>V</i> (n_{vide}); <i>Consommer le message</i> ; <u>aller à CONS</u> ;
---	--

Analysons le fonctionnement du système qui vient d'être décrit.

Retenant 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 :

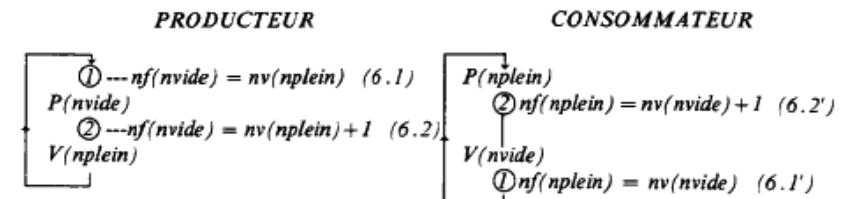
$$nf(s) = \min(np(s), e0(s) + nv(s))$$

Remplaçant successivement s par n_{plein} et n_{vide} , on obtient :

$$(5) \quad \left\{ \begin{array}{l} nf(n_{plein}) = \min(np(n_{plein}), nv(n_{plein})) \\ nf(n_{vide}) = \min(np(n_{vide}), n + nv(n_{vide})) \end{array} \right.$$

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 :

$$\begin{aligned} nf(nvide) &< np(vide) && (\text{bloquage du producteur}) \\ nf(nplein) &< np(nplein) && (\text{bloquage du consommateur}) \end{aligned}$$

D'après (5), on aurait donc :

$$\begin{aligned} nf(nvide) &= n + nv(nvide) \\ nf(nplein) &= nv(nplein) \end{aligned}$$

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 supposons (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 :

<i>déposer (message) :</i>	<i>prélever (message) :</i>
<i>tampon [queue] := message ;</i>	<i>message := tampon [tête] ;</i>
<i>queue := queue + 1 mod n ;</i>	<i>tête := tête + 1 mod n ;</i>

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 :

$$\begin{aligned} nf(nvide) &= nv(nplein) + 1 \\ nf(nplein) &= nv(nvide) + 1 \end{aligned}$$

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

$$\begin{aligned} nf(nvide) &\leq n + nv(nvide) \\ nf(nplein) &\leq nv(nplein) \end{aligned}$$

Éliminant les nf des 4 relations précédentes, on obtient :

$$(7) \quad 0 < nv(nplein) - nv(nvide) < n$$

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 :

$$\begin{aligned} tête &= (tête)_0 + nf(nplein) \text{ mod } n = nf(nplein) \text{ mod } n \\ queue &= (queue)_0 + nf(nvide) \text{ mod } n = nf(nvide) \text{ mod } n \end{aligned}$$

L'égalité $tête = queue$ implique donc :

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

ou encore, d'après (6.2) et (6.2') :

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

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 :

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

2.522 Communication par boîte aux lettres

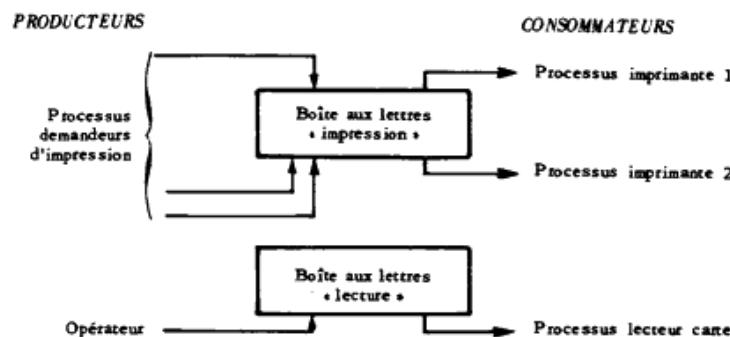
La communication entre processus suivant le schéma dit de la boîte aux lettres est une application directe du modèle du producteur et du consommateur. Une boîte aux lettres est une zone de mémoire permettant la communication entre ces processus suivant le schéma qui vient d'être défini. Remarquons que, 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 Séma phores avec messages [Saal, 70]

Le principe du séma phore avec message découle de la remarque suivante : lorsqu'un processus p active un processus q par une opération V sur son séma phore 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 séma phores 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

$P_M(s, \text{message reçu})$

exécutée par p est équivalente à

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

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

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

exécutée par p équivaut à :

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

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 :

$P_M(s, \text{message reçu}(p))$	$V_M(s, \text{message transmis})$
<u>début</u>	<u>début</u>
$e(s) := e(s) - 1;$	$e(s) := e(s) + 1;$
<u>si</u> $e(s) < 0$ <u>alors</u>	<u>si</u> $e(s) \leq 0$ <u>alors</u>
<u>début</u>	<u>début</u>
$\text{état } (p) := \text{bloqué};$	choisir $q \in f_p(s);$
$p \rightarrow f_p(s)$	$\text{message reçu } (q) :=$
<u>fin</u>	$\text{message transmis};$
<u>sinon</u>	$\text{état } (q) := \text{actif};$
<u>début</u>	$f_p(s) \rightarrow q$
choisir $m \in f_M(s);$	<u>fin</u>
$f_M(s) \rightarrow m;$	<u>sinon</u>
$\text{message reçu} := m$	$\text{message transmis} \rightarrow f_M(s)$
<u>fin</u>	<u>fin</u>
<u>fin</u> ;	

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 $e(s) \geq 0$, 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 $e(s) \geq 0$),

— 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 invisibles, 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 MUS

Le système MUS [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 chainage 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 MUS 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.

2.6.1 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.

2.6.2 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 chainés 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.

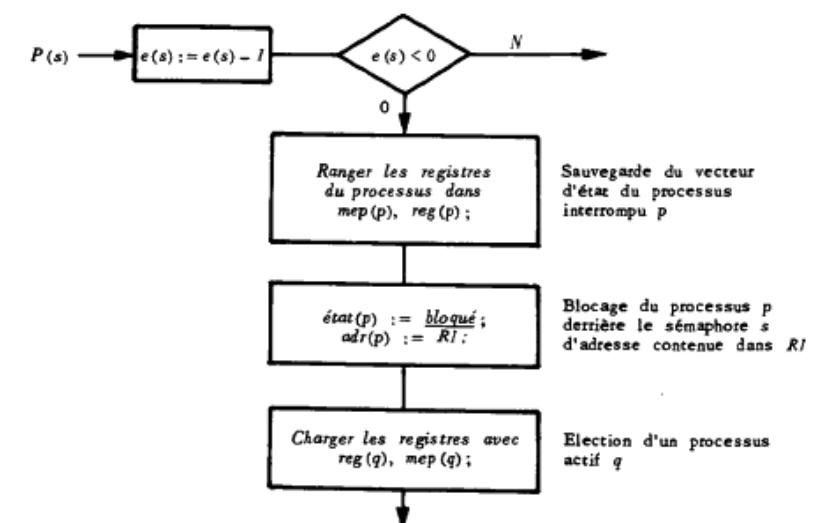
— l'implantation des files d'attente utilisées dans le système BURROUGHS B6500 pour la gestion des événements.

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

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

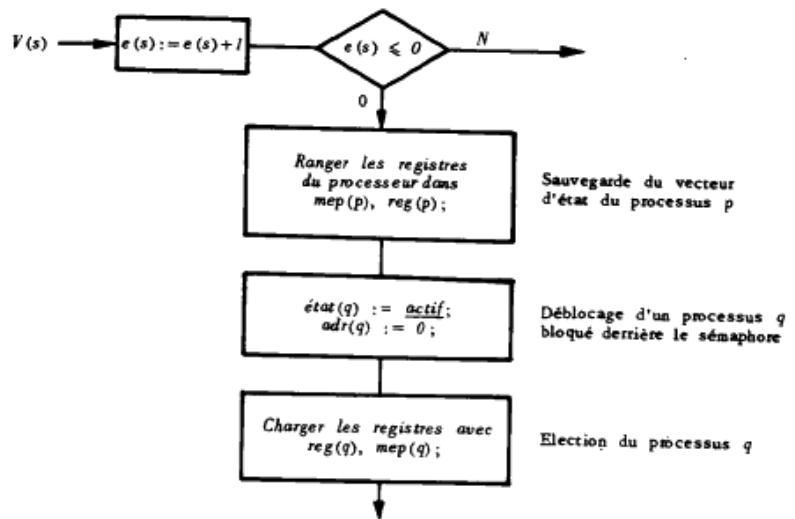
- *état* état du processus,
- *mep* mot d'état du programme,
- *reg* registres généraux,
- *adr* adresse de sémaphore,
- *suc* pointeur vers le vecteur d'état du processus suivant dans la file.

Lors de l'appel d'une primitive par l'intermédiaire d'un appel au superviseur (instruction *SVC*), le registre *RI* 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 *vproto* sont chainés 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* chaine 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 [Boule, 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.

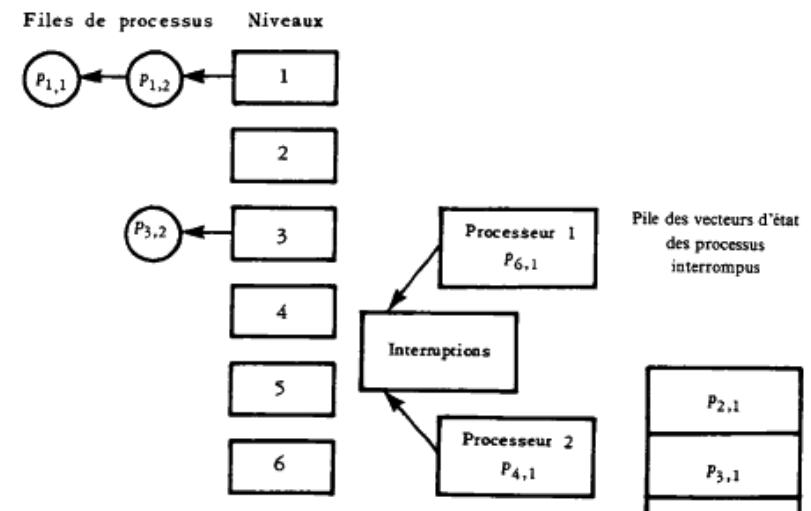


Figure 1. Gestion des processus dans le système SIRIS 8.

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 $p_{5,1}$ 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 étudié plus en détail en 3.3. On entretient à chaque création ou destruction de processus une arborescence généalogique (Fig. 2).

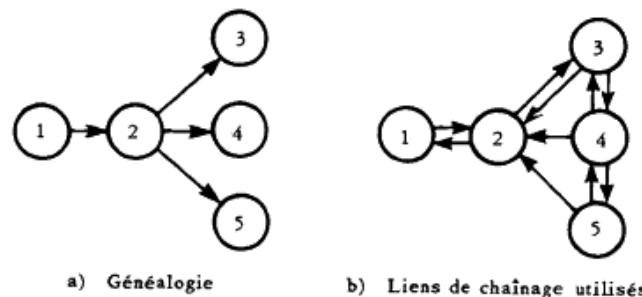


Figure 2. Arborescence de processus dans le B6500.

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.

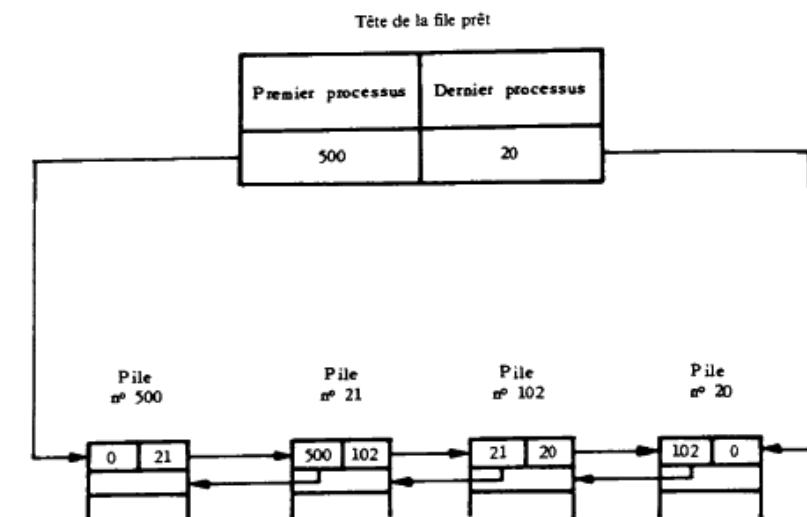


Figure 3. File de processus dans le B6500.

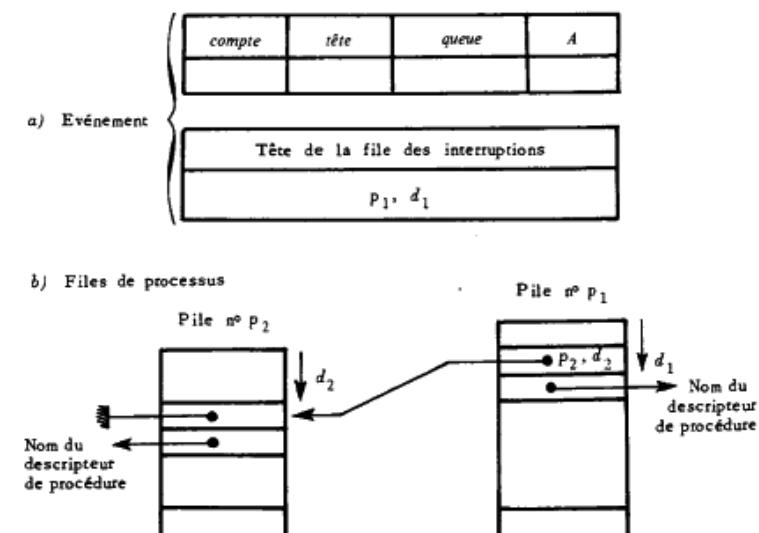


Figure 4. Gestion des interruptions programmées dans le B6500.

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 chainage 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 :

- chainer 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 sauvégarde 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

2.7.1 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.

2.7.2 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*
Préparer le message d'entrée-sortie;
V(monit.ses);

⋮

P(sem);

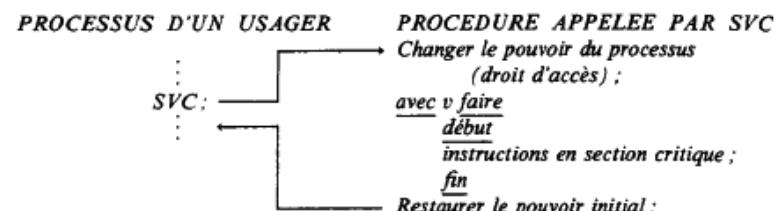
PROCESSUS DU MONITEUR
A : P(ses);
Décoder le message d'entrée-sortie;
Exécuter l'entrée-sortie;
V(u.sem);
aller à A;

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.

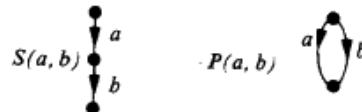
EXERCICES

1. [1]

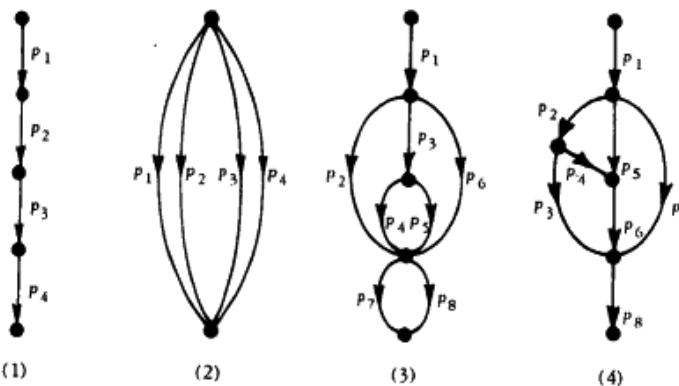
Un ensemble de processus coopérant à l'exécution d'une tâche commune peut être représenté par un graphe orienté dans lequel chaque arc représente l'évolution complète d'un processus. Le graphe ainsi obtenu est appelé graphe des processus pour la tâche

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 :



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 :

$c[i] = \text{vrai}$ si le processus p_i est dans sa section critique ou demande à y entrer,
 $c[i] = \text{faux}$ 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) 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 $c[1] = \text{vrai}$ et $c[2] = \text{vrai}$) 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 $c[i] := \text{faux}$, 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 $c[i] := \text{vrai}$.

— 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, \dots, p_n . On introduira comme précédemment un tableau booléen $c[I : n]$ 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 $c[i] := \text{vrai}$ 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 :

$$b[i] = \begin{cases} \text{vrai si} & \text{ou bien } t = i \\ & \text{ou bien le processus } p_i \text{ demande à faire } t := i \\ \text{faux si } t \neq i \text{ et si le processus } p_i \text{ est hors de sa section critique.} \end{cases}$$

On s'assurera que p_i ne peut exécuter $t := i$ que si $b[i] = \text{vrai}$.

4. [1]

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

Cas 1 (condition et)

Processus a	Processus b	Processus c
:	:	:
<u>wait</u> (e1,e2) (2)	<u>completion</u> (e1) = '1'B	<u>completion</u> (e2) = '1'B
:	:	:
<i>Cas 2 (condition ou)</i>		

Processus a	Processus b	Processus c
:	:	:
<u>wait</u> (e1,e2) (1)	<u>completion</u> (e1) = '1'B	<u>completion</u> (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 écrire et aucun lecteur ne peut lire pendant ce temps).

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

LECTEURS	REDACTEURS
<u>demande de lecture</u> ;	<u>demande d'écriture</u> ;
lecture ;	écriture ;
<u>fin de lecture</u> ;	<u>fin d'écriture</u> ;

62 Systèmes d'exploitation des ordinateurs

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 doit 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 $spriv(i)$, $i = 1, 2, \dots, n$,
- 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 n_r 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 :

- toute voiture se présentant au carrefour doit le franchir au bout d'un temps fini,
- les feux de chaque voie passent alternativement du vert au rouge, chaque couleur étant maintenue pendant un temps fini,
- 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 *traversée₁* et *traversée₂*, 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, $e_0(s)$ 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 :

$P(n, s) : \underline{\text{si }} n \leq e(s) \text{ alors}$
 $\quad e(s) := e(s) - n$

sinon

début
commentaire on désigne par p le processus appelant ;
introduire p dans $f(s)$;
 $\text{état}(p) := \underline{\text{bloqué}}$;
 $\text{rang}(p) := n$
fin

$V(n, s) : \underline{\text{début}}$
 $\quad e(s) := e(s) + n ;$
tant que $q \in f(s)$ et $\text{rang}(q) \leq e(s)$ faire

début
extraire de $f(s)$ un processus p tel que $\text{rang}(p) \leq e(s)$;
 $\text{état}(p) := \underline{\text{actif}}$;
 $e(s) := e(s) - \text{rang}(p)$
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 $V(n, s)$ 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 $1, 2, \dots, n-1, n$ 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) , $i = 1, \dots, n$ 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, chainées 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 :

demande 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 :

- le processus p entre en section critique et évalue b ;
- si $b = \text{vrai}$, p exécute I et sort de la section critique ;
- si $b = \text{faux}$, 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, \dots, s_k une nouvelle opération indivisible $P(s_1, s_2, \dots, s_k)$. Selon la notation habituelle, $e(s_i)$ désigne la valeur du sémaphore s_i . L'effet de la nouvelle opération est le suivant :

Si tous les sémaphores s_i ($1 \leq i \leq k$) ont une valeur $e(s_i)$ positive, la primitive $P(s_1, \dots, s_k)$ 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 $P(s_1, s_2, \dots, s_k)$ 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.

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 évoyer 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 $E = \{ i \in N \mid 0 \leq i \leq 20 \text{ et } i \text{ premier} \}$ peut se définir en ALGOL 68 comme :

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

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 <i>désigne</i> un objet	\Leftrightarrow	un nom <i>désigne</i> un emplacement
un objet est une constante	\Leftrightarrow	un emplacement <i>renferme</i> une constante
un objet <i>repère</i> un objet	\Leftrightarrow	un emplacement <i>contient</i> un nom qui <i>désigne</i> un emplacement

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

- a) $LW, R \quad m$
- b) $LW, R \quad *m$

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

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

- a) m désigne un emplacement qui *contient* une valeur (r)
- 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 :

$$\text{origine} + (\text{valeur}(i) - v_1) * (v'_2 - v_2 + 1) + \text{valeur}(j) - v_2$$

Exemple 2. Une structure, en PL/I, 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

$LW, R \quad m, X$

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 réservé 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 réentrant 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.

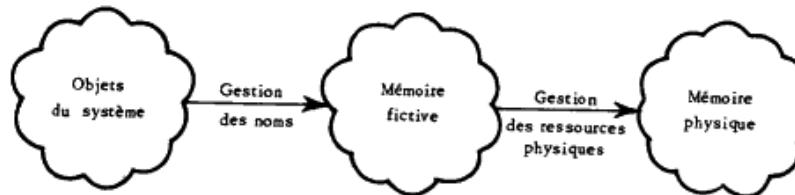
En résumé, le partage des objets ou des noms introduit les conditions suivantes :

- 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 $0, 1, 2, \dots, N$. 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

3.2.1 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.

3.2.2 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 :

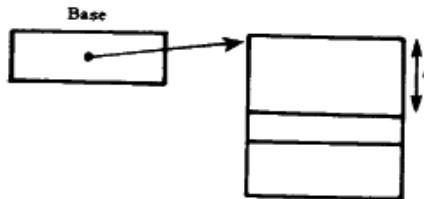
- la longueur du segment,
- l'adresse, en mémoire fictive, du premier mot (base) du segment (codée sur 40 bits),
- des indicateurs utilisés par le mécanisme de protection.

Tout accès à un mot du segment fait intervenir le descripteur de segment.

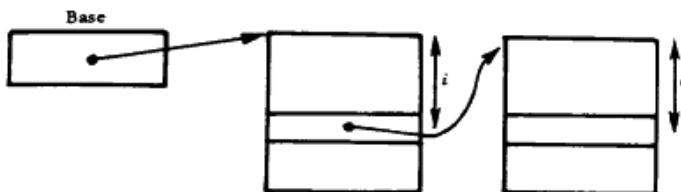
L'application des segments dans la mémoire fictive, fait passer d'un ensemble de suites, les segments, dans une suite unique, la mémoire fictive. Cette transformation nécessite une gestion de la mémoire fictive analogue à la gestion de la mémoire physique telle qu'elle est présentée en 4.44. Cette gestion 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 :

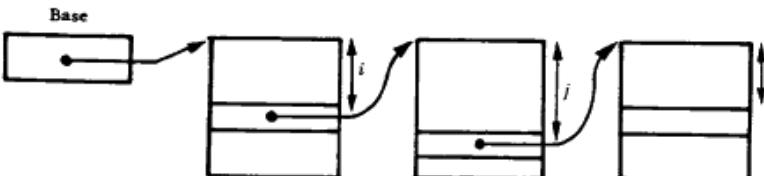
- si $K \leq 1$, l'adresse de base du segment est celle d'une page de mémoire fictive associée au segment.



— si $1 < K \leq 2^8$, 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 $2^8 < K \leq 2^{16}$, 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).

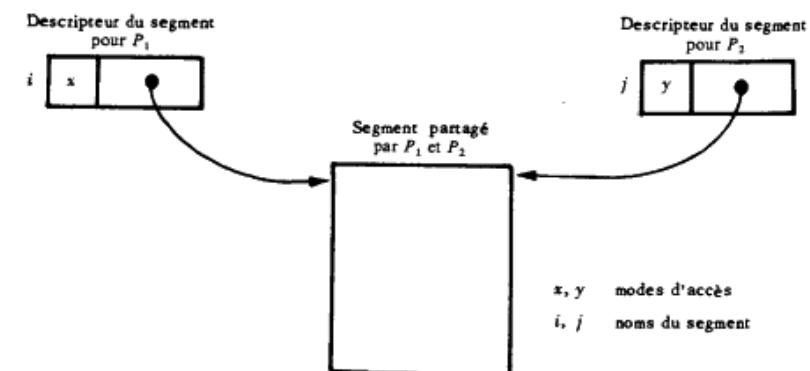


Figure 1. Noms d'un segment.

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.

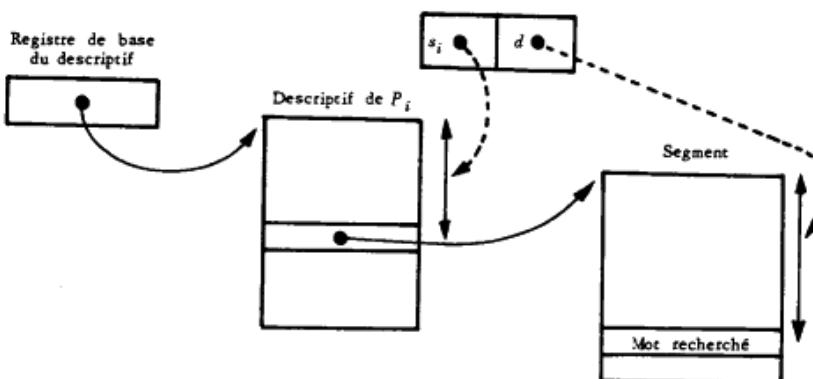


Figure 2. Adresse segmentée.

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 $R0, R1, \dots, R15$; 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).