# Méthodes pour l'écriture des systèmes d'exploitation

Jacques Mossière

# THESE



*présentée à*

**Université Scientifique et Médicale de Grenoble**

**Institut National Polytechnique de Grenoble**

*pour obtenir le grade de*

DOCTEUR es SCIENCES
Mathématiques

Jacques MOSSIERE



## METHODES POUR L'ECRITURE DES SYSTEMES D'EXPLOITATION.

Thèse soutenue le 23 septembre 1977 devant la Commission d'Examen :

Président : L. BOLLIET

Examinateurs : J.C. BOUSSARD
S. KRAKOWIAK
M. SINTZOFF
J.P. VERJUS

| | | |
|---|---|---|
| MM. | RAYNAUD Hervé | M.I.A.G. |
| | REBECQ Jacques | Biologie (CUS) |
| | REYMOND Jean-Charles | Chirurgie générale |
| | RICHARD Lucien | Biologie végétale |
| Mme. | RINAUDO Marguerite | Chimie macromoléculaire |
| MM. | ROBERT André | Chimie papetière |
| | SARRAZIN Roger | Anatomie et chirurgie |
| | SARROT-REYNAULD Jean | Géologie |
| | SIROT Louis | Chirurgie générale |
| Mme. | SOUTIF Jeanne | Physique générale |
| MM. | STIEGLITZ Paul | Anesthésiologie |
| | VIALON Pierre | Géologie |
| | VAN CUTSEM Bernard | Mathématiques appliquées |

## MAITRES DE CONFERENCES ET MAITRES DE CONFERENCES AGREGES

| | | |
|---|---|---|
| MM. | ARMAND Yves | Chimie (IUT I) |
| | BACHELOT Yvan | Endocrinologie |
| | BARGE Michel | Neuro-chirurgie |
| | BEGUIN Claude | Chimie organique |
| Mme | BERIEL Hélène | Pharmacodynamie |
| MM. | BOST Michel | Pédiatrie |
| | BOUCHARLAT Jacques | Psychiatrie adultes |
| Mme. | BOUCHE Liane | Mathématiques (CUS) |
| MM. | BRODEAU François | Mathématiques (IUT B) (Personne étrangère habilitée à être directeur de thèse) |
| | CHAMBAZ Edmond | Biochimie médicale |
| | CHAMPETIER Jean | Anatomie et organogénèse |
| | CHARDON Michel | Géographie |
| | CHERADAME Hervé | Chimie papetière |
| | CHIAVERINA Jean | Biologie appliquée (EFP) |
| | CONTAMIN Charles | Chirurgie thoracique et cardio-vasculaire |
| | CORDONNIER Daniel | Néphrologie |
| | COULOMB Max | Radiologie |
| | CROUZET Guy | Radiologie |
| | CYROT Michel | Physique du solide |
| | DENIS Bernard | Cardiologie |
| | DOUCE Roland | Physiologie végétale |
| | DUSSAUD René | Mathématiques (CUS) |
| Mme. | ETERRADOSSI Jacqueline | Physiologie |
| MM. | FAURE Jacques | Médecine légale |
| | FAURE Gilbert | Urologie |
| | GAUTIER Robert | Chirurgie générale |
| | GIDON Maurice | Géologie |
| | GROS Yves | Physique (IUT I) |
| | GUIGNIER Michel | Thérapeutique |
| | GUITTON Jacques | Chimie |
| | HICTER Pierre | Chimie |
| | JALBERT Pierre | Histologie |
| | JULIEN-LAVILLAVROY Claude | O.R.L. |
| | KOLODIE Lucien | Hématologie |
| | LE NOC Pierre | Bactériologie-virologie |
| | MACHE Régis | Physiologie végétale |
| | MAGNIN Robert | Hygiène et médecine préventive |
| | MALLION Jean-Michel | Médecine du travail |
| | MARECHAL Jean | Mécanique (IUT I) |
| | MARTIN-BOUYER Michel | Chimie (CUS) |
| | MICHOULIER Jean | Physique (IUT I) |

UNIVERSITE SCIENTIFIQUE
ET MEDICALE DE GRENOBLE

---

Monsieur Gabriel CAU : Président
Monsieur Pierre JULLIEN : Vice Président

---

## MEMBRES DU CORPS ENSEIGNANT DE L'U.S.M.G.

### PROFESSEURS TITULAIRES

| | | |
|---|---|---|
| MM. | AMBLARD Pierre | Clinique de dermatologie |
| | ARNAUD Paul | Chimie |
| | ARVIEU Robert | I.S.N. |
| | AUBERT Guy | Physique |
| | AYANT Yves | Physique approfondie |
| Mme. | BARBIER Marie-Jeanne | Electrochimie |
| MM. | BARBIER Jean-Claude | Physique expérimentale |
| | BARBIER Reynold | Géologie appliquée |
| | BARJON Robert | Physique nucléaire |
| | BARNOU Fernand | Biosynthèse de la cellulose |
| | BARRA Jean-René | Statistiques |
| | BARRIE Joseph | Clinique chirurgicale |
| | BEAUDOING André | Clinique de pédiatrie et puériculture |
| | BELORIZKY Elie | Physique |
| | BERNARD Alain | Mathématiques pures |
| Mme. | BERTRANDIAS Françoise | Mathématiques pures |
| MM. | BERTRANDIAS Jean-Paul | Mathématiques pures |
| | BEZEZ Henri | Pathologie chirurgicale |
| | BLAMBERT Maurice | Mathématiques pures |
| | BOLLIET Louis | Informatique (IUT B) |
| | BONNET Jean-Louis | Clinique ophtalmologique |
| | BONNET-EYMARD Joseph | Clinique gastro-entérologique |
| Mme. | BONNIER Marie-Jeanne | Chimie générale |
| MM. | BOUCHERLE André | Chimie et toxicologie |
| | BOUCHEZ Robert | Physique nucléaire |
| | BOUSSARD Jean-Claude | Mathématiques appliquées |
| | BOUTET DE MONVEL Louis | Mathématiques pures |
| | BRAVARD Yves | Géographie |
| | CABANEL Guy | Clinique rhumatologique et hydrologique |
| | CALAS François | Anatomie |
| | CARLIER Georges | Biologie végétale |
| | CARRAZ Gilbert | Biologie animale et pharmocodynamie |
| | CAU Gabriel | Médecine légale et toxicologie |
| | CAUQUIS Georges | Chimie organique |
| | CHABAUTY Claude | Mathématiques pures |
| | CHARACHON Robert | Clinique oto-rhino-laryngologique |
| | CHATEAU Robert | Clinique de neurologie |
| | CHIBON Pierre | Biologie animale |
| | COEUR André | Pharmacie chimique et chimie analytique |
| | CONTAMIN Robert | Clinique gynécologique |
| | COUDERC Pierre | Anatomie pathologique |

| | | |
|---|---|---|
| Mme. | DEBELMAS Anne-Marie | Matière médicale |
| MM. | DEBELMAS Jacques | Géologie générale |
| | DEGRANGE Charles | Zoologie |
| | DELORMAS Pierre | Pneumophtisiologie |
| | DEPORTES Charles | Chimie minérale |
| | DESRE Pierre | Métallurgie |
| | DESSAUX Georges | Physiologie animale |
| | DODU Jacques | Mécanique appliquée (IUT I) |
| | DOLIQUE Jean-Michel | Physique des plasmas |
| | DREYFUS Bernard | Thermodynamique |
| | DUCROS Pierre | Cristallographie |
| | GAGNAIRE Didier | Chimie physique |
| | GALVANI Octave | Mathématiques pures |
| | GASTINEL Noël | Analyse numérique |
| | GAVEND Michel | Pharmacologie |
| | GEINDRE Michel | Electroradiologie |
| | GERBER Robert | Mathématiques pures |
| | GERMAIN Jean-Pierre | Mécanique |
| | GIRAUD Pierre | Géologie |
| | JANIN Bernard | Géographie |
| | KAHANE André | Physique générale |
| | KOSZUL Jean-Louis | Mathématiques pures |
| | KLEIN Joseph | Mathématiques pures |
| | KRAVTCHENKO Julien | Mécanique |
| | KUNTZMANN Jean | Mathématiques appliquées |
| | LACAZE Albert | Thermodynamique |
| | LACHARME Jean | Biologie végétale |
| Mme. | LAJZEROWICZ Janine | Physique |
| MM. | LAJZEROWICZ Joseph | Physique |
| | LATREILLE René | Chirurgie générale |
| | LATURAZE Jean | Biochimie pharmaceutique |
| | LAURENT Pierre-Jean | Mathématiques Appliquées |
| | LEDRU Jean | Clinique médicale B |
| | LE ROY Philippe | Mécanique (IUT I) |
| | LLIBOUTRY Louis | Géophysique |
| | LOISEAUX Pierre | Sciences nucléaires |
| | LONGEQUEUE Jean-Pierre | Physique nucléaire |
| | LOUP Jean | Géographie |
| Melle | LUTZ Elisabeth | Mathématiques pures |
| MM. | MALINAS Yves | Clinique obstétricale |
| | MARTIN-NOEL Pierre | Clinique cardiologique |
| | MAZARE Yves | Clinique médicale A |
| | MICHEL Robert | Minéralogie et pétrographie |
| | MICOUD Max | Clinique maladies infectieuses |
| | MOURIQUAND Claude | Histologie |
| | MOUSSA André | Chimie nucléaire |
| | NOZIERES Philippe | Spectrométrie physique |
| | OZENDA Paul | Botanique |
| | PAYAN Jean-Jacques | Mathématiques pures |
| | PEBAY-PEYROULA Jean-Claude | Physique |
| | PERRET Jean | Semeiologie médicale (Neurologie) |
| | RASSAT André | Chimie systématique |
| | RENARD Michel | Thermodynamique |
| | REVOL Michel | Urologie |
| | RINALDI Renaud | Physique |
| | DE ROUGEMONT Jacques | Neuro-chirurgie |
| | SEIGNEURIN Raymond | Microbiologie et Hygiène |
| | SENGEL Philippe | Zoologie |
| | SIBILLE Robert | Construction mécanique (IUT I) |

| | | |
|---|---|---|
| MM. | SOUTIF Michel | Physique générale |
| | TANCHE Maurice | Physiologie |
| | TRAYNARD Philippe | Chimie générale |
| | VAILLANT François | Zoologie |
| | VALENTIN Jacques | Physique nucléaire |
| | VAUQUOIS Bernard | Calcul électronique |
| Mme. | VERAIN Alice | Pharmacie galénique |
| MM. | VERAIN André | Physique |
| | VEYRET Paul | Géographie |
| | VIGNAIS Pierre | Biochimie médicale |

PROFESSEURS ASSOCIES

| | | |
|---|---|---|
| MM. | CRABBE Pierre | CERMO |
| | DEMBICKI Eugéniuz | Mécanique |
| | JOHNSON Thomas | Mathématiques appliquées |
| | PENNEY Thomas | Physique |

PROFESSEURS SANS CHAIRE

| | | |
|---|---|---|
| Melle | AGNIUS-DELORD Claudine | Physique pharmaceutique |
| | ALARY Josette | Chimie analytique |
| MM. | AMBROISE-THOMAS Pierre | Parasitologie |
| | ARMAND Gilbert | Géographie |
| | BENZAKEN Claude | Mathématiques appliquées |
| | BIAREZ Jean-Pierre | Mécanique |
| | BILLET Jean | Géographie |
| | BOUCHET Yves | Anatomie |
| | BRUGEL Lucien | Energétique (IUT I) |
| | BUISSON René | Physique (IUT I) |
| | BUTEL Jean | Orthopédie |
| | COHEN ADDAD Pierre | Spectrométrie physique |
| | COLOMB Maurice | Biochimie |
| | CONTE René | Physique (IUT I) |
| | DELOBEL Claude | M.I.A.G. |
| | DEPASSEL Roger | Mécanique des fluides |
| | FONTAINE Jean-Marc | Mathématiques pures |
| | GAUTRON René | Chimie |
| | GIDON Paul | Géologie et minéralogie |
| | GLENAT René | Chimie organique |
| | GROULADE Joseph | Biologie médicale |
| | HACQUES Gérard | Calcul numérique |
| | HOLLARD Daniel | Hématologie |
| | HUGONOT Robert | Hygiène et médecine préventive |
| | IDELMAN Simon | Physiologie animale |
| | JOLY Jean-René | Mathématiques pures |
| | JULLIEN Pierre | Mathématiques appliquées |
| Mme. | KAHANE Josette | Physique |
| MM. | KRAKOWIACK Sacha | Mathématiques appliquées |
| | KUHN Gérard | Physique (IUT I) |
| | LUU DUC Cuong | Chimie organique |
| | MAYNARD Roger | Physique du solide |
| Mme. | MINIER Colette | Physique (IUT I) |
| MM. | PELMONT Jean | Biochimie |
| | PERRIAUX Jean-Jacques | Géologie et minéralogie |
| | PFISTER Jean-Claude | Physique du solide |
| Melle | PIERY Yvette | Physiologie animale |

| | | |
|---|---|---|
| MM. | NEGRE Robert | Mécanique (IUT I) |
| | NEMOZ Alain | Thermodynamique |
| | NOUGARET Marcel | Automatique (IUT I) |
| | PARAMELLE Bernard | Pneumologie |
| | PECCOUD François | Analyse (IUT B) (Personnalité étrangère habilité à être directeur de thèse) |
| | PEFFEN René | Métallurgie (IUT I) |
| | PERRIER Guy | Géophysique-Glaciologie |
| | PHELIP Xavier | Rhumatologie |
| | RACHAIL Michel | Médecine interne |
| | RACINET Claude | Gynécologie et obstétrique |
| | RAMBAUD André | Hygiène et hydrologie (Pharmacie) |
| | RAMBAUD Pierre | Pédiatrie |
| | RAPHAEL Bernard | Stomatologie |
| Mme. | RENAUDET Jacqueline | Bactériologie (Pharmacie) |
| MM. | ROBERT Jean-Bernard | Chimie physique |
| | Romier Guy | Mathématiques (IUT B) (Personnalité étrangère habilité à être directeur de thèse) |
| | SCHAERER René | Cancérologie |
| | SHOM Jean-Claude | Chimie générale |
| | STOEBNER Pierre | Anatomie pathologie |
| | VROUSOS Constantin | Radiologie |

MAITRES DE CONFERENCES ASSOCIES

| | | |
|---|---|---|
| MM. | DEVINE Roderick | Spectro physique |
| | HODGES Christopher | Transition de phases |

*Fait à SAINT MARTIN D'HERES, NOVEMBRE 1976.*

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Monsieur Philippe TRAYNARD : Président

Monsieur Pierre-Jean LAURENT : Vice Président

---

## PROFESSEURS TITULAIRES

| | | |
|---|---|---|
| MM. | BENOIT Jean | Radioélectricité |
| | BESSON Jean | Electrochimie |
| | BLOCH Daniel | Physique du solide |
| | BONNETAIN Lucien | Chimie minérale |
| | BONNIER Etienne | Electrochimie et électrométallurgie |
| | BOUDOURIS Georges | Radioélectricité |
| | BRISSONNEAU Pierre | Physique du solide |
| | BUYLE-BODIN Maurice | Electronique |
| | COUMES André | Radioélectricité |
| | DURAND Francis | Métallurgie |
| | FELICI Noël | Electrostatique |
| | FOULARD Claude | Automatique |
| | LESPINARD Georges | Mécanique |
| | MOREAU René | Mécanique |
| | PARIAUD Jean-Charles | Chimie-Physique |
| | PAUTHENET René | Physique du solide |
| | PERRET René | Servomécanismes |
| | POLOUJADOFF Michel | Electrotechnique |
| | SILBER Robert | Mécanique des fluides |

## PROFESSEUR ASSOCIE

| | | |
|---|---|---|
| M. | ROUXEL Roland | Automatique |

## PROFESSEURS SANS CHAIRE

| | | |
|---|---|---|
| MM. | BLIMAN Samuel | Electronique |
| | BOUVARD Maurice | Génie mécanique |
| | COHEN Joseph | Electrotechnique |
| | LACOUME Jean-Louis | Géophysique |
| | LANCIA Roland | Electronique |
| | ROBERT François | Analyse Numérique |
| | VEILLON Gérard | Informatique fondamentale et appliquée |
| | ZADWORNY François | Electronique |

MAITRES DE CONFERENCES

| | | |
|---|---|---|
| MM. | ANCEAU François | Mathématiques appliquées |
| | CHARTIER Germain | Electronique |
| | GUYOT Pierre | Chimie minérale |
| | IVANES Marcel | Electrotechnique |
| | JOUBERT Jean-Claude | Physique du solide |
| | MORET Roger | Electrotechnique nucléaire |
| | PIERRARD Jean-Marie | Mécanique |
| | SABONNADIERE Jean-Claude | Informatique fondamentale et appliquée |
| Mme. | SAUCIER Gabrièle | Informatique fondamentale et appliquée |

MAITRE DE CONFERENCES ASSOCIE

| | | |
|---|---|---|
| M. | LANDAU Ioan | Automatique |

CHERCHEURS DU C.N.R.S. (Directeur et Maîtres de Recherche)

| | | |
|---|---|---|
| MM | FRUCHART Robert | Directeur de Recherche |
| | ANSARA Ibrahim | Maître de Recherche |
| | CARRE René | Maître de Recherche |
| | DRIOLE Jean | Maître de Recherche |
| | MATHIEU Jean-Claude | Maître de Recherche |
| | MUNIER Jacques | Maître de Recherche |

*Je remercie Monsieur BOLLIET, Professeur à l'Université de Grenoble, qui m'a fait l'honneur de présider le jury de cette thèse.*

*Je remercie Messieurs BOUSSARD, Professeur à l'Université de Nice, et SINTZOFF, Ingénieur au M.B.L.E., qui se sont intéressés à mon travail et ont accepté de le juger.*

*Jean-Pierre VERJUS a su trouver les arguments nécessaires pour que j'entreprenne la rédaction de cette thèse; il m'a aidé pendant sa réalisation, tant par ses critiques qu'en me déchargeant d'une partie de mon travail. Pour celà, je tiens à lui exprimer ma gratitude.*

*Dès mon premier travail de recherche, j'ai eu la chance de collaborer avec Sacha KRAKOWIAK. Ses conseils et ses encouragements ont toujours été précieux; son rôle modérateur a permis de maintenir au cours de plusieurs projets une ambiance chaleureuse.*

*La recherche en systèmes d'exploitation est toujours un travail d'équipe; le contenu de ce travail doit donc beaucoup à tous ceux avec qui j'ai pu travailler ou échanger des idées pendant ces dernières années. Je me dois de mentionner en particulier tous mes collègues de l'équipe SESAME :*
*Bernard CASSAGNE, Jean-Louis CHEVAL, Flaviu CRISTIAN, Martine LUCAS et Jean MONTUELLE.*

*Xavier ROUSSET de PINA a dépassé son rôle de compagnon de cordée pour se charger d'une part de mon travail d'enseignant; je l'en remercie sincèrement.*

*Je remercie enfin Madame TREVISAN et tout le personnel du service de reproduction pour la réalisation matérielle de cet ouvrage.*

# CHAPITRE 1

## AVANT - PROPOS

L'objet de ce travail est l'étude de méthodes et d'outils de conception et d'écriture de systèmes d'exploitation. Dans cet avant-propos, nous allons tout d'abord montrer en quoi ce sujet est actuel, nous expliciterons ensuite notre façon d'aborder ces problèmes puis nous présenterons les différents chapitres de cette thèse.

La conception et l'écriture des systèmes d'exploitation sont souvent considérées comme une affaire de spécialistes : quelques équipes travaillant chez les constructeurs d'ordinateurs fournissent des programmes mystérieux, dont la mise en oeuvre demanderait plus de savoir-faire et d'astuces que de connaissances. Seuls quelques initiés, les "ingénieurs-systèmes", seraient capables d'effectuer l'entretien et les modifications de ces programmes, opérations dont tout le monde admet la nécessité.

Cette vision primitive tend heureusement à disparaître : d'une part la différenciation des besoins des utilisateurs, d'autre part l'évolution des matériels (mini et microordinateurs) amènent de plus en plus de gens, soit à modifier des systèmes d'exploitation, soit à concevoir et à écrire des systèmes adaptés à leurs besoins propres. Un système d'exploitation n'est alors plus écrit une fois et une seule pour un calculateur donné ; c'est un programme comme les autres, à ceci près qu'il est en général plus gros et plus difficile à écrire et surtout à mettre au point qu'un programme d'application ordinaire. Il est même couramment admis qu'un grand système d'exploitation est représentatif des constructions les plus complexes que l'esprit humain ait à concevoir.

Dans ces conditions, il importe de rechercher des méthodes de construction et de mettre au point des outils aptes à simplifier et à accélérer le travail des programmeurs : on assiste ainsi à la conception de langages de description et de spécification de systèmes, de langages d'écriture de systèmes; de nombreuses méthodes de structuration, de construction et de mise au point de systèmes d'exploitation sont essayées ; etc... c'est dans ce courant de recherches que notre travail s'inscrit.

En raison de la complexité du sujet, nous avons suivi une démarche assez empirique. Dans un premier temps, nous avons participé à la conception, à l'écriture et à la mise au point d'un système d'exploitation complet. Pendant cette ph[illegible] l'objectif prioritaire était la construction du système; nous avons pu cependant en tirer quelques idées sur les méthodes à suivre et les outils dont il serait intéressant de disposer pour mener à bien un projet de construction de système. Dans un second temps, nous avons alors choisi de mettre l'accent sur ces méthodes et ces outils;lorsque nous avons écrit des parties de systèmes, c'était beaucoup plus pour vérifier la validité de nos idées que pour disposer d'un système particulier.

C'est le bilan de cette double expérience que nous présentons ici. Plus précisément, nous avons retenu trois problèmes que nous considérons comme importants, et sur lesquels nous estimons avoir eu un apport original : la construction de systèmes à l'aide de modules élémentaires, les langages d'écriture de systèmes et l'expression du parallélisme. Ce sont ces trois points que nous allons développer, en nous appuyant sur notre expérience propre, et en situant nos travaux par rapport à ceux menés par d'autres équipes. Comme les projets auxquels nous avons participé ont fait l'objet de diverses publications, nous n'avons pas repris en détail tous les résultats obtenus : nous nous sommes contentés de rendre cet ouvrage compréhensible en lui-même; quant au reste, le lecteur pourra se reporter à la bibliographie. En ce sens, cette thèse doit être considérée comme intermédiaire entre une thèse au sens classique du terme et une thèse "sur travaux".

Ce travail est articulé de la façon suivante.

Au chapitre 2, nous décrivons les problèmes que pose la programmation des systèmes d'exploitation, justifiant ainsi le choix des points que nous avons décidé de traiter.

Au chapitre 3, nous présentons brièvement les projets auxquels nous avons participé ; nous avons regroupé en fin de ce chapitre la liste de toutes les publications consacrées à ces projets.

Nous abordons alors nos trois sujets d'étude. Au chapitre 4, nous traitons de la décomposition des systèmes d'exploitation, et de la construction de systèmes par assemblage de constituants; au chapitre 5 nous étudions les langages d'écriture de systèmes et au chapitre 6 les questions d'expression du parallèlisme. Ces trois chapitres

ont été écrits de manière à être lus de manière autonome. On ne s'étonnera donc pas d'y trouver quelques redites.

Enfin, nous récapitulons en conclusion les résultats obtenus et mentionnons une liste de problèmes actuellement non résolus de façon satisfaisante, dont l'étude pourrait constituer un prolongement de ce travail.

# CHAPITRE 2

# LA PROGRAMMATION DES SYSTEMES D'EXPLOITATION

## 2.1. INTRODUCTION

De nombreuses définitions des systèmes d'exploitation peuvent être trouvées dans la littérature (voir par exemple [Hoare 72a]). Les auteurs s'accordent en général sur deux aspects, donnant la primauté tantôt à l'un, tantôt à l'autre :

a) Partage de ressources : "La fonction première du système d'exploitation d'un calculateur est le partage du matériel qu'il gère entre un ensemble d'usagers qui font des demandes de ressources non prévues à l'avance; ce partage doit être effectué de manière efficace, fiable et discrète" [Hoare 72a].

b) Machine virtuelle : "Le système a pour rôle de masquer certaines limitations ou imperfections du matériel, ou de simuler une machine différente de la machine réelle. L'utilisateur a alors à sa disposition une "machine virtuelle" munie d'un "langage étendu", c'est-à-dire d'un mode d'expression mieux adapté que les seules instructions câblées [Crocus 75].

Dans les deux définitions ci-dessus, il est implicite qu'un système est un ensemble de programmes s'exécutant sur un matériel donné ; l'écriture de ces programmes est délicate à cause de leur taille et de la prise en compte des caractéristiques du matériel (entrées-sorties sur des périphériques variés, traitement d'interruptions, ...)

En conséquence, la réalisation d'un système d'exploitation pose des problèmes de divers ordres :

- problèmes de conception et d'architecture globale (choix des principales fonctions ou services à offrir aux utilisateurs, étude d'algorithmes de partage de ressources, de mécanismes de synchronisation, etc...),
- problèmes de programmation (écriture et mise au point de programmes réalisant les fonctions définies lors de la conception),
- problèmes d'entretien et de modifications (correction des erreurs détectées au cours de l'exploitation, adaptation du système selon l'évolution des besoins des utilisateurs).

La distinction entre ces problèmes n'est en pratique pas aussi nette, elle est néanmoins suffisamment marquée pour que l'on puisse les étudier séparément.

Notre étude est surtout consacrée au second de ces problèmes : la programmation des systèmes d'exploitation. Dans ce chapitre, nous allons donc montrer les difficultés de cette programmation, situer la phase de programmation par rapport à celles de conception et d'entretien et justifier le choix des problèmes traités par la suite. Cette présentation est faite en deux temps. Dans le premier, nous abordons les problèmes de programmation des gros programmes, dont les systèmes d'exploitation constituent un cas particulier ; dans le second, nous prenons en compte les spécificités des systèmes d'exploitation.

## 2.2. LES SYSTEMES D'EXPLOITATION EN TANT QUE GROS PROGRAMMES

A l'exception de certains systèmes très spécialisés, les systèmes d'exploitation sont en général de "gros" programmes. Il faut entendre par là que leur taille peut atteindre quelques centaines de milliers d'instruction et que leur réalisation peut impliquer la collaboration de quelques dizaines de personnes pendant plusieurs années. Ces valeurs sont actuellement extrêmes, mais on peut noter qu'un système prototype comme ESOPE contient quelques milliers d'instructions, et que sa réalisation a occupé cinq personnes pendant environ trois ans, qu'un petit système comme GMS [Bellino 73] occupe 48K octets de mémoire et que sa réalisation a demandé deux personnes et demie pendant deux ans.

Des informations chiffrées sur la conception et la réalisation des gros programmes peuvent être trouvées dans [Naur69, Buxton 70].

Tous les problèmes que pose la réalisation d'un système vont donc être amplifiés par la taille du programme à écrire. Les difficultés croissent beaucoup plus vite que la taille du programme ; de plus, il est difficile de déduire des méthodes d'écriture de gros programmes sans expérimentation en vraie grandeur (voir par exemple [Dijkstra 72]).

La réalisation d'un gros programme comprend schématiquement trois phases (conception, écriture et mise au point, entretien) que nous examinons successivement.

### 2.2.1. Conception

On appelle généralement phase de conception la phase de travail qui, à partir d'objectifs généraux plus ou moins flous, permet :

(i) de préciser ces objectifs
(ii) de définir la structure globale du programme à écrire : principaux constituants (modules, processus), communications entre ces constituants.

Des algorithmes ne sont précisés dans cette phase que s'ils influent sur la structure d'ensemble du programme ; les autres sont laissés à la charge du réalisateur de chaque constituant.

a) Spécifications

On appelle spécifications d'un programme le résultat de la phase de conception. Les spécifications permettent de programmer, puis d'assembler les différentes parties du programme. Plus précisément, elles peuvent prendre diverses formes :

- programme équivalent (dont l'exécution produit le même effet)
- assertions (pré- et post-conditions) portant sur l'état des variables du programme avant et après son exécution. Ces assertions peuvent être exprimées de façon formelle (prédicats logiques) ou informelle (langue naturelle).
- spécifications fonctionnelles.

Pour être utiles, les spécifications d'un programme doivent posséder l'une au moins des propriétés suivantes :

- faciliter la construction du programme (par un moyen automatique ou non),
- faciliter la compréhension du programme et servir ainsi de documentation,
- permettre d'établir (par un moyen automatique ou non) des propriétés du programme.

En pratique, on est à peu près capable de spécifier formellement un petit programme (algorithme à caractère mathématique, mais aussi aspect particulier d'un système). Pour un système d'exploitation complet, on se contente en général d'un texte en langage courant ; la longueur de ce texte peut atteindre, voir dépasser, celle d'un gros livre.

b) Décomposition

(i) Abstraction et raffinement

La décomposition en parties est couramment utilisée pour faciliter la compréhension, la conception et la construction d'ensembles complexes. Cette méthode a donc trouvé de nombreux usages en programmation et plusieurs tentatives ont été faites pour faciliter son utilisation rationnelle. Ont ainsi été introduites dans la conception des programmes les notions d'abstraction et de niveaux d'abstraction, de raffinement, de conception descendante, de machine abstraite, de module [Dijkstra 72, Wirth 71]. De telles notions sont en fait un prolongement de la notion de spécification. L'opération d'abstraction consiste à définir une machine (abstraite, c'est-à-dire non nécessairement réalisée physiquement). Définir une machine, c'est choisir un répertoire de modèles d'objets et de modèles d'actions portant sur ces objets, pour réaliser un ensemble de fonctions définies par le concepteur. L'opération de raffinement peut se définir ainsi : étant donnée la description d'une action dans un langage (c'est-à-dire au moyen du répertoire d'actions et d'objets d'une machine abstraite), décrire la même action en explicitant au moyen d'un autre langage les actions et objets utilisés. La conception d'un programme de quelque complexité est une combinaison, consciente ou non, de ces deux démarches.

(ii) Modules

Une autre approche des problèmes de décomposition est la programmation modulaire : les programmes sont décomposés en parties, ou modules, les avantages attendus étant les suivants :

- Meilleure compréhension du fonctionnement du programme

Chaque module remplit une fonction déterminée, dont la définition est indépendante de son mode de réalisation; le problème de la réalisation d'une fonction particulière par un module peut être disjoint du problème de la connexion du module au reste du système.

- Facilité de production

Une fois la décomposition définie, les diverses parties du système peuvent être réalisées en parallèle par des équipes distinctes ayant peu d'inter-

actions. Les communications d'information sont restreintes au strict nécessaire; en particulier, les informations sur le fonctionnement interne d'un module peuvent et doivent rester inconnues des utilisateurs de ce module.

- facilité d'entretien (cf. § 2.2.3.)

L'application à la conception des programmes de techniques de décomposition n'est pas nouvelle. Depuis [Mealy 66], de nombreuses publications sont consacrées à la décomposition des programmes, par exemple [Zürcher 68, Wirth 71, Dijkstra 72, Jackson 75, Myers 75].

Au chapitre 4, nous traiterons de la décomposition des systèmes et de leur réalisation par assemblage de parties élémentaires. Cet assemblage est décrit à l'aide d'un langage particulier, le langage de connexion.

### 2.2.2. Ecriture et mise au point

Bien qu'elle se révèle efficace, la décomposition ne résoud pas tous les problèmes. Etant donné les spécifications d'une partie, il reste à construire un programme qui les vérifie. On a besoin pour celà d'un langage de programmation. On demande tout d'abord à ce langage de permettre une écriture et une mise au point facile des programmes, mais aussi de faciliter les modifications ultérieures (cf. § 2.2.3.). Le programme produit doit donc être le plus clair possible.

De plus, une démarche naturelle pour la rationalisation de la production de logiciel semble être d'introduire, dans les langages servant à cette production, les notions et méthodes jugées utiles à la conception pour la classe de programmes considérée. Par exemple, si l'on souhaite utiliser une méthode de conception descendante, une structure modulaire, et des processu parallèles, il semble utile de faire passer ces notions sous la forme de constructions ou d'objets du langage utilisé. Cette démarche a l'avantage d'obliger à expliciter des notions souvent mal définies pour les faire entrer dans le moule contraignant d'une syntaxe.

La tendance actuelle est à la définition de langages évolués adaptés à l'écriture des gros programmes (et en particulier des systèmes d'exploitation). Nous allons étudier maintenant les incidences sur la programmation d'une phase d'exploitation et d'entretien assez longue (§ 2.2.3.), puis les caractéristiques des systèmes d'exploitation proprement dits (§ 2.3.). Quelques aspects des langages d'écriture de systèmes sont étudiés au chapitre 5.

### 2.2.3. Entretien - Modifications

Deux raisons principales imposent un certain entretien des gros programmes au cours de leur exécution : la présence d'erreurs résiduelles et l'évolution des besoins des utilisateurs.

Compte-tenu du caractère informel des spécifications des gros programmes, on ne peut envisager de prouver a priori leur validité. En conséquence, quel que soit le soin apporté à la mise au point, des erreurs subsisteront lors de l'exploitation. Elles peuvent provenir soit de spécifications incohérentes, soit de simples erreurs de codage.

Quant aux besoins des utilisateurs, ils vont d'autant plus varier que l'exploitation du programme sera plus longue. Et tout gros programme doit être exploité plusieurs années pour amortir le coût de sa réalisation. On aura par exemple à prendre en compte de nouveaux périphériques, à ajuster des paramètres d'exploitation, à mettre en oeuvre de nouvelles fonctions; on doit aussi souvent reprogrammer certaines parties dont la réalisation a été jugée non satisfaisante à l'expérience.

Dans les deux cas, des modifications doivent être apportées au programme, et ces modifications sont en général effectuées par des programmeurs distincts des réalisateurs du système.

En conséquence, tout programme doit être compréhensible. C'est-à-dire qu'on doit disposer d'un document réunissant l'ensemble des spécifications des différentes parties; pour chaque partie, on doit disposer du texte du programme, dans lequel les spécifications sont rappelées sous forme de commentaires. Le rôle des principales parties du programme doit être également explicité, même si on

utilise un langage évolué. Enfin, la détection des erreurs peut être facilitée si on dispose de la liste des essais effectués.

Quant aux modifications proprement dites, certaines sont prévues dès la conception. On réalise alors plutôt une famille de systèmes qu'un système particulier. On laisse ainsi chaque utilisateur définir la configuration de son calculateur, des tailles de tables, des paramètres d'exploitation, mais aussi des algorithmes particuliers adaptés à ses besoins propres.

Lorsque les paramétrages peuvent être reflétés dans les programmes eux-mêmes (par exemple sous forme de déclarations de constantes), ces modifications sont quasi-automatiques. A l'inverse, des modifications délicates sont celles qui remettent en cause la structure d'ensemble du programme : si on veut les réaliser en conservant une structure logique, elles imposent de reprendre la phase de conception et donc probablement une bonne part de la programmation.

## 2.3. CARACTERISTIQUES DES SYSTEMES D'EXPLOITATION

### 2.3.1. Existence d'activités parallèles

Dans la plupart des calculateurs, différents traitements peuvent être effectués simultanément : l'utilisation de canaux d'entrée-sortie tend à se généraliser même sur les minicalculateurs, certaines machines comportent plusieurs unités centrales, d'autres sont construites spécialement pour permettre un haut degré de parallélisme.

D'autre part, les systèmes d'exploitation sont en général utilisés concurremment par plusieurs usagers, et il faut assurer l'indépendance logique des travaux soumis par ces usages.

En conséquence, le programme d'un système d'exploitation ne peut se réduire à un programme à exécution séquentielle ; au contraire, on doit prendre en compte plusieurs exécutions simultanées : on décompose alors un système en une famille d'entités, chacune d'elle à exécution séquentielle : les processus. Si les processus d'un système étaient indépendants, la complication ne serait pas très grande; l'utilisation d'un dispositif de protection de mémoire et un allocateur d'unité centrale en tourniquet permettraient d'exécuter correctement ces processus. En fait, les processus ne sont pas totalement indépendants pour deux

classes de raisons :

- raisons intrinsèques, des processus coopérent à la réalisation d'une même fonction ,
- raisons technologiques, les processus se partagent soit les ressources physiques du calculateur, soit des programmes particuliers ; le partage de programmes (compilateurs) est particulièrement fréquent entre processus d'usagers différents

Le programmeur d'un système d'exploitation doit donc disposer d'un langage lui permettant de décrire des processus et les relations qui doivent exister entre ces processus. Cette expression du parallélisme est un point très délicat de la programmation des systèmes : l'existence simultanée de plusieurs processus dont les vitesses relatives ne sont guère prévisibles pose des problèmes de mise au point, car les erreurs sont souvent difficiles à reconstituer. Les instructions d'expression du parallélisme doivent alors reposer sur des mécanismes indépendants des vitesses d'exécution et permettre de vérifier statiquement certaines propriétés qui restent invariantes lors de l'exécution du système.

Quand on analyse les relations entre processus dans un système réel, on distingue trois schémas distincts de synchronisation :
- l'exclusion mutuelle,
- le couplage de processus par un modèle producteur-consommateur,
- l'allocation de ressources.

Ces trois problèmes sont définis précisément au chapitre 6 ; c'est à partir des solutions à ces problèmes que nous y comparons différentes instructions d'expression du parallélisme. La notion de processus en tant qu'unité de décomposition d'un système est étudiée au chapitre 4.

### 2.3.2. Mise en oeuvre de toutes les fonctions d'un calculateur

Les instructions disponibles sur un calculateur peuvent schématiquement se répartir en deux classes :

- les instructions qui définissent une machine simplifiée connue du programmeur (transferts entre registres et mémoire, arithmétique fixe ou flottante, branchements, ...)
- les instructions de commande du fonctionnement du calculateur, qui permettent de simuler sur la machine réelle plusieurs exemplaires de la machine simplifiée.
  (on retrouve souvent cette répartition au niveau de la protection lorsqu'on distingue les instructions privilégiées des autres).

Dans la seconde classe se trouvent les instructions de commande des entrées-sorties, des interruptions, de la protection de mémoire, de mise à jour du mot d'état de programme. On peut la caractériser par les propriétés suivantes :

- grande hétérogénéité dans les formats des instructions,
- opérandes devant se trouver souvent à des emplacements imposés par le matériel,
- format des opérandes (mot d'état de programme, commande d'entrée-sortie,...) imposé par le matériel.

A cause de ces dernières propriétés, il est difficile de produire ces instructions par l'intermédiaire d'un compilateur. Si bien qu'on a considéré longtemps qu'un système d'exploitation devait être écrit ou bien en langage d'assemblage, ou bien dans un langage donnant accès à toutes les possibilités d'une machine (PL360). Deux types de solutions se dessinent aujourd'hui :

- inclusion dans un langage de programmation de caractéristiques dépendant de la machine,
- définition des données et des instructions en deux temps, définition logique et directives d'implantation.

Nous traiterons de ces solutions au chapitre 5.

### 2.3.3. Problèmes d'efficacité

En général, on demande à un programme d'être efficace, c'est-à-dire de s'exécuter en consommant le moins possible de ressources. Ce souhait est particulièrement important lorsque le programme est un système d'exploitation qui ne doit pas monopoliser à son profit les ressources du calculateur. Par contre, on ne doit pas en faire de déduction trop simpliste, par exemple "pour être efficace, tout système doit être écrit en langage d'assemblage", ou encore "tout programme écrit en langage évolué est inefficace". Les différentes parties d'un système ayant des caractéristiques d'exécution très différentes, il convient tout d'abord de préciser les caractéristiques de la partie étudiée (fréquence d'utilisation, priorité, ...) avant tout choix de réalisation. L'efficacité d'un programme donné va alors dépendre de l'algorithme employé (ce que l'on pourrait appeler efficacité intrinsèque) et de la méthode de programmation (efficacité technologique). Seule l'efficacité technologique nous préoccupe ici. Nous en discuterons au chapitre 5.

### 2.3.4. Structures de données

Tout langage de programmation permet la manipulation de classes d'objets. Ces objets prennent leurs valeurs dans certains ensembles et on peut effectuer sur eux des opérations bien définies. De manière informelle, nous appelons ici type la définition d'une classe d'objets (c'est-à-dire de l'ensemble des valeurs possibles de ces objets) ainsi que des opérations que l'on peut effectuer sur ces objets.

On peut dès lors distinguer deux catégories de langages de programmation : ceux qui offrent au programmeur un répertoire de types fixé une fois pour toutes et ceux qui, à partir de types préexistants, dits types primitifs, permettent la définition de nouveaux types d'objets.

Lors de la définition d'un langage de programmation, le choix des types des données utilisables est l'un des points les plus délicats : si ces types sont mal adaptés aux problèmes à traiter, le programmeur consacrera une part notable de son activité à la traduction des données de son problème en terme des structures disponibles ; de plus une telle traduction ne peut que diminuer la lisibilité et les possibilités de modification de programmes.

Lorsque l'on vise une application bien définie, des choix judicieux sont possibles (tableaux en calcul scientifique par exemple). Par contre, en "programmation de systèmes", la nature des objets utilisés est difficilement prévisible. Après s'être longtemps contenté d'utiliser le langage de la machine, on semble s'orienter actuellement vers les langages permettant la définition de nouveaux types.

La définition des structures de données dans différents langages est étudiée au chapitre 5.

## 2.4. CONCLUSION

Essayons de résumer les principales caractéristiques de la programmation de système, telles que nous avons pu les dégager dans ce chapitre.

Tout d'abord, les systèmes d'exploitation sont de gros programmes qui sont fréquemment modifiés pendant leur phase d'utilisation. En conséquence, on doit attacher beaucoup d'importance aux facilités d'entretien et de modification.

Ils font en général coexister plusieurs activités parallèles, ce qui pose des problèmes de mise au point car certaines erreurs sont difficiles à reproduire. Les données manipulées sont peu prévisibles a priori, ce qui suggère de les programmer dans un langage permettant de définir de nouveaux types.

Enfin, on demande aux systèmes d'exploitation d'être le plus efficace possible.

De plus, tous ces problèmes ne sont pas disjoints (par exemple, l'existence d'activités parallèles complique la conception d'un système et les accès aux données) ; leurs solutions sont parfois contradictoires (la conservation d'une structure modulaire à l'exécution peut entraîner une perte d'efficacité).

Il nous faut donc terminer en admettant que la réalisation de systèmes d'exploitation est une activité difficile. Elle doit donc être abordée en disposant de méthodes et d'outils propres à la simplifier.

# CHAPITRE 3

# PRESENTATION GENERALE DES TRAVAUX EFFECTUES

## 3.1. INTRODUCTION

Nous présentons dans ce chapitre les différents projets auxquels nous avons participé. Pour chacun d'eux, nous nous attacherons à montrer les objectifs, les points les plus originaux et les apports que nous en avons personnellement retirés. Par contre, on ne trouvera pas ici une description extensive de ces projets, qui ne ferait que reprendre d'autres publications. Quant aux points précis nécessaires à notre argumentation, ils seront décrits à leur juste place, dans les chapitres 4 à 6.

Nos travaux seront exposés en trois paragraphes : la plus grande place sera accordée aux deux premiers, décrivant les deux projets (ESOPE, SESAME) auxquels nous avons participé : projets d'une certaine importance matérielle (4 ou 5 personnes pendant plusieurs années), dont nous avons pu retirer une bonne connaissance des problèmes que pose l'écriture des gros programmes. Nous avons regroupé dans le troisième paragraphe des travaux plus ponctuels.

## 3.2. LE PROJET ESOPE

ESOPE (Exploitation Simultanée d'un Ordinateur et de ses PEriphériques) est un système d'exploitation prototype étudié et réalisé à l'IRIA de fin 1968 à 1971.

Les spécifications générales du système sont des plus classiques : gestion de travaux conversationnels introduits à partir de téléimprimeurs, d'un travail de fond et d'un travail en temps réel de forte priorité. La machine sur lequel il est réalisé (CII 10070) est assez répandue et a été ou sera l'objet de nombreuses études de systèmes d'exploitation (BPM, BTM, UTS, GORDO, SAM, SAR, SIRIS 7). L'intérêt essentiel du projet n'est donc pas à rechercher dans le produit réalisé, d'autant plus que, pour des raisons non-techniques, ce produit ne sera jamais mis en exploitation réelle. Cependant, deux raisons justifient qu'ESOPE apparaisse dans cette étude.

a) Il s'agit d'une expérience réelle de programmation d'un système d'exploitation complet. En ce sens, l'apport d'un tel projet, au moins pour ses participants, est immense : comme le signale en particulier Dijkstra [Dijkstra 72], il n'est

pas possible de saisir la complexité d'un grand système à partir d'expériences restreintes. Nous ferons donc appel à des exemples tirés d'ESOPE toutes les fois que nous aurons besoin d'étudier des problèmes réels.

b) Avec quelques années de recul, certaines solutions semblent toujours intéressantes. On peut noter :

- l'utilisation systématique des sémaphores pour la synchronisation entre processus, même pour la réalisation du système d'entrées-sorties [Baudet 72]. L'introduction d'une primitive de synchronisation supplémentaire, D(s), semble toutefois superflue.

- l'utilisation de l'espace virtuel et du défaut de page pour l'accès aux fichiers, solution qui est maintenant largement répandue (appelée couplage dans ESOPE).

- la gestion globale de la mémoire et de l'unité centrale par des algorithmes permettant de s'adapter dynamiquement à la charge du système. Cependant, compte-tenu de l'arrêt du projet, ces algorithmes n'ont jamais été complètement validés (ils se trouvent de toutes façons hors des sujets développés dans ce travail).

- la notion d'usager, famille de processus coopérant de façon étroite à la réalisation d'un travail et descendant d'un même ancêtre, le processus premier de l'usager.

On pourra aborder ESOPE par les articles de presentation générale [Bétourné 70a, 70b]. Une description détaillée est fournie dans les thèses [Ferrié 71, Kaiser 73, Krakowiak 73, Mossière 71]. Les premières évaluations du projet se trouvent dans [Boulenger 74, Kaiser 74a, 74b].

## 3.3. LE PROJET SESAME

Dans le cadre d'ESOPE, nos efforts se sont surtout concentrés sur le système à produire. Naturellement, de nombreux problèmes de méthode se sont posés au cours de la réalisation. Ces problèmes ont été résolus "au jour le jour", et les décisions prises ne paraissent,avec un peu de recul,plus toujours judicieuses.

Les principales difficultés provenaient :

- des insuffisances du langage de programmation utilisé, le LP 10070 (langage de la famille du PL 360); notre expérience d'utilisation de ce langage est évaluée dans [Boulenger 74],

- de l'absence d'outils de conservation et d'assemblage de programmes ; la gestion des différentes versions du système, les passages d'une version à une autre étaient assurés "manuellement",

- d'une méthode de construction insuffisamment réfléchie : l'accent a été délibérément mis sur la division du système en processus parallèles; la décomposition en programmes compilés séparément s'est faite de façon empirique. Ces problèmes de construction sont étudiés au chapitre 4.

Il nous a donc paru intéressant de tirer parti de l'expérience acquise pour définir un projet centré sur les méthodes d'écriture de systèmes d'exploitation. C'est ainsi qu'est né SESAME, Système d'Ecriture de Systèmes par Assemblage de Modules Eléméntaires.

Le projet comporte deux volets, le développement d'outils d'écriture de systèmes et la validation de ces outils par la programmation de systèmes prototypes.

### 3.3.1. Outils d'écriture

Les systèmes d'exploitation, gros programmes complexes, ne peuvent être écrits et mis au point d'une seule pièce. Au contraire, leur réalisation implique la mise au point de nombreux morceaux indépendants, ou modules, qui sont progressivement assemblés pour former un système. De plus, on ne construit que rarement un système unique ; on prévoit dès le départ toute une classe de systèmes voisins. Dans ces conditions, les divers modules, comme certains assemblages, doivent être paramétrés.

En conséquence, SESAME offre :

- un langage d'écriture de modules, conçu comme une extension à Pascal [Jensen 74]; les modules de SESAME sont analogues à ceux décrits par Parnas [Parnas 72a]

- un langage, dit langage de connexion, distinct du précédent, décrivant la liste des modules constituant un système et les connexions existant entre ces modules.

- un bibliothécaire, permettant de conserver et de mettre à jour un ensemble de modules et de programmes de connexion. En effet, lors de la décomposition d'un gros programme, on limite souvent le nombre des morceaux pour en simplifier la gestion. Nous espérons que la mise à la disposition des usagers d'un système de conservation adéquat va permettre de lever cette restriction.

### 3.3.2. Validation

Pour montrer la validité de nos idées, nous avons entrepris la programmation d'un ensemble de systèmes :

- le bibliothécaire a été conçu et écrit de façon modulaire [Cristian 77],
- pour montrer l'utilité réelle du paramétrage, nous avons décrit toute une classe de systèmes à traitement par trains [Montuelle 77],
- un système complet (traitement par trains avec flots d'entrée et de sortie simultanés au traitement) a été réalisé; ce dernier programme devrait pouvoir être utilisé pour l'enseignement des systèmes d'exploitation : la structure modulaire se prête aussi bien à l'exposé magistral de sous-systèmes qu'à la définition de travaux pratiques.

A l'occasion de ces réalisations, nous avons effectué un travail de réflexion sur les principes de la conception modulaire [Cheval 77, Montuelle 77]. La conception s'effectue en deux phases :

- définition d' "objets abstraits" indépendamment de considérations de réalisations (un module réalise un objet abstrait).
- explicitation des mécanismes (création, connexion et destruction des objets abstraits) dont on avait admis de disposer dans la première phase (on introduit alors de nouveaux modules).

Ce processus peut être itéré jusqu'à ce que la réalisation de tous les objets abstraits ait été définie.

### 3.3.3. Structure globale

On trouvera une présentation d'ensemble de SESAME dans [Cheval 76a]; [Lucas 77] et [Montuelle 77] fournissent une description extensive du projet.

Actuellement, une version prototype est en fin de mise au point. Elle implique l'utilisation de deux machines, un IRIS 80 et un T1600.

Notre système de réalisation de systèmes (bibliothécaire, compilateur du langage d'écriture des modules, connecteur) est implanté sur l'IRIS 80, ce qui nous permet de profiter des avantages offerts par une grosse machine. Avec cet outil, nous produisons des systèmes d'exploitation exécutables sur le T1600. Plus précisément, nous avons décidé de munir systématiquement le T1600 d'un noyau réalisant des fonctions élémentaires (gestion de processus, allocation de mémoire). Le compilateur traduit certaines instructions en appels à ce noyau. En conséquence, la nature des instructions offertes par le noyau, comme leurs interfaces, sont figées; par contre leur réalisation pourra varier d'un système à l'autre.

Remarque : Compte-tenu de la nature élémentaire des instructions du noyau, les restrictions aux systèmes produits nous paraissent peu importantes.

Nos premières expériences de construction modulaire sont encourageantes. Les limitations du langage d'écriture des modules (pas de données dynamiques, rigidité de la notion de type) viennent de Pascal. Le langage de connexion permet de décrire commodément les assemblages de modules. Une généralisation aurait été souvent appréciée : l'inclusion dans le langage d'écriture des modules d'instruction de connexion dynamique, qui permettraient de reconfigurer les systèmes produits. Cette "orthogonalisation" serait indiscutable dans une réalisation faisant intervenir une seule machine. Elle est plus contestable sur notre prototype car elle reviendrait à figer une structure de bibliothécaire sur le T1600.

Nous reviendrons dans ce travail sur les aspects les plus originaux du projet, principalement aux chapitres 4 et 5.

## 3.4. AUTRES TRAVAUX

Nos autres sources d'expérience proviennent d'une part de la participation à un groupe de travail sur les systèmes, de nos tâches d'enseignement d'autre part.

### 3.4.1. Groupe de travail

C'est dans le groupe CROCUS que nous avons eu l'occasion de confronter nos idées de façon un peu systématique, et surtout nous avons pu découvrir d'autres façons d'aborder les problèmes de système et en particulier deux extrêmes :

- la méthode "linguistique" du groupe de Rennes, fondée sur l'utilisation exclusive d'Algol 68 en tant que langage d'écriture du système et langage unique d'utilisation [Trilling 72].
- l'approche IBM composant des systèmes monstrueux à partir de "modules", de tables ou de macros et qui (devrait-on dire par quel miracle) fonctionnent à la satisfaction générale des utilisateurs [Mealy 66].

### 3.4.2. Les retombées de l'enseignement

Quant à nos activités d'enseignement, elles nous ont essentiellement permis de vérifier certaines idées lors de l'encadrement de projets. Nous avons pu ainsi surveiller la réalisation de deux systèmes simples mais complets : GIMMIC [De Courcel 74], système à processus parallèles construit sur un IBM 360 et PROSPER [Herodin 77], système modulaire implanté sur un T1600. C'est également dans ce cadre [Lucas 74] que nous avons abordé l'étude des primitives de synchronisation dans les langages de haut-niveau.

## PUBLICATIONS SUR ESOPE

[Baudet 72] : BAUDET G., FERRIE J., KAISER C., MOSSIERE J.

"Entrées-sorties dans un système à mémoire virtuelle"

*Congrès AFCET*, Grenoble (1972)

[Bétourné 70a] : BETOURNE C., BOULENGER J., FERRIE J., KAISER C., KRAKOWIAK S. MOSSIERE J.

- Présentation générale du système ESOPE
- Espace virtuel dans le système ESOPE
- Allocation de ressources dans le système ESOPE

*Congrès AFCET*, Paris (1970)

[Bétourné 70b] : BETOURNE C., BOULENGER J., FERRIE J., KAISER C., KRAKOWIAK S. MOSSIERE J.

"Process Management and resource sharing in the multiaccess system ESOPE"

*CACM* 13,12 (1970)

[Bétourné 71] : BETOURNE C., FERRIE J., KAISER C., KRAKOWIAK S., MOSSIERE J.

"System design and implementation using parallel processes"

*Proc. IFIP Congress*, Ljubljana (1971)

[Bétourné 72] : BETOURNE C., KRAKOWIAK S.

"Simulation de l'allocation de ressources dans un système conversationnel à mémoire virtuelle paginée"

*Congrès AFCET*, Grenoble (1972)

[Bétourné 75] : BETOURNE C., KRAKOWIAK S.

"Mesures sur un système conversationnel"

*RAIRO, AFCET* (1975)

[Boulenger 74] : BOULENGER J. KRONENTAL M.

"An experience in systems programming using a PL360-like language"

*Proc. IFIP WG-2.4 meeting*, La Grande Motte (1974)

[Ferrié 72] : FERRIE J., MOSSIERE J.

"ESOPE : gestion des processus et partage des ressources"

*cahier de l'IRIA* n°8 (1972)

[Kaiser 73] : KAISER C.

*"Conception et réalisation de systèmes à accès multiple : gestion du parallèlisme"*

Thèse de Doctorat es-Sciences, Paris (1973)

[Kaiser 74a] : KAISER C., KRAKOWIAK S.
"Analyse de quelques pannes d'un système d'exploitation"
*Proc. International Symposium on Operating Systems*, IRIA (1974)

[Kaiser 74b] : KAISER C., KRAKOWIAK S.
"Design and implementation of a time-sharing system : a critical appraisal"
*Proc. IFIP Congress*, Stockholm (1974)

[Krakowiak 73] : KRAKOWIAK S.
*"Conception et réalisation de systèmes à accès multiples : allocation de ressources"*
Thèse de Doctorat es-Sciences, Paris (1973)

## PUBLICATIONS SUR SESAME

[Cheval 76a] : CHEVAL J.L., CRISTIAN F., KRAKOWIAK S., LUCAS Ma., MONTUELLE J., MOSSIERE J.
*Conception modulaire des systèmes d'exploitation*
Rapport de recherche n° 32, Mathématiques Appliquées-Informatique, Grenoble (1976)

[Cheval 76b] : CHEVAL J.L., CRISTIAN F., KRAKOWIAK S., LUCAS Ma., MONTUELLE J., MOSSIERE J.
"Un système d'aide à l'écriture des systèmes d'exploitation"
*Congrès AFCET*, Paris (1976)

[Cheval 77] : CHEVAL J.L., CRISTIAN F., KRAKOWIAK S., MONTUELLE J., MOSSIERE J.
"An experiment in modular program design"
*Proc. IFIP Congress*, Toronto (1977)

[Cristian 76] : CRISTIAN F., GAUDUEL F.
*"Réalisation d'un bibliothécaire pour le projet SESAME"*
Rapport 3ème année, ENSIMAG, Grenoble (1976)

[Cristian 77] : CRISTIAN F.
A case study in modular design"
*Proc. International Computing Symposium*, Liège (1977)

[Krakowiak 74] : KRAKOWIAK S., MOSSIERE J.
*Quelques problèmes de la programmation modulaire.*
Note interne (1974)

[Krakowiak 75] : KRAKOWIAK S., MOSSIERE J.
*Systèmes d'aide à la programmation de systèmes*
Journée sur les systèmes d'aide à la programmation, IRIA (1975)

[Krakowiak 76] : KRAKOWIAK S., LUCAS Ma., MONTUELLE J., MOSSIERE J.
"A modular approach to the structured design of operating systems"
*Proc. MRI Symposium on Computer Software Engineering*, Polytechnic Institute, New York (1976)

[Lucas 75] : LUCAS Ma.
*Primitives de synchronisation pour langages de haut niveau*
Séminaire de programmation de Grenoble (1975)

[Lucas 77] : LUCAS Ma.
*Conception modulaire des systèmes d'exploitation. Outils pour la programmation modulaire*
Thèse 3ème cycle, Grenoble (1977)

[Montuelle 77] : MONTUELLE J.
*Conception modulaire des systèmes d'exploitation. Méthodes et exemple d'application*
Thèse docteur-ingénieur, Grenoble (1977)

[Mossière 75] : MOSSIERE J.
*Quelques outils d'aide à la programmation des systèmes d'exploitation*
Séminaire de programmation de Grenoble (1975)

[Sesame 77] : Equipe SESAME
"Implantation d'une structure modulaire sur minordinateur, réalisation d'un noyau de multiprogrammation"
*BIGRE* n° 5 (1977)

# CHAPITRE 4

# PROBLEMES DE DECOMPOSITION ET D'ASSEMBLAGE

## 4.1. INTRODUCTION

Les systèmes d'exploitation sont de gros programmes complexes qu'il est impossible d'appréhender globalement. Que l'on s'intéresse à leur spécification, leur écriture, leur mise au point, ou plus simplement à la compréhension de leur fonctionnement, une méthode s'impose : diviser le système en un ensemble de parties de façon que chaque partie soit compréhensible, de même que les relations entre parties.

Pour effectuer cette division, deux "écoles" s'affrontent :

- partir du problème le plus général que l'on cherche à résoudre, et le décomposer en un petit nombre de problèmes plus simples, que l'on réduit à leur tour;
- partir de la machine (ou du langage) dont on dispose et enrichir progressivement l'ensemble des données et des opérations définies sur cette machine, en espérant qu'après quelques étapes d'enrichissements, on aura une solution au problème initial.

Cette distinction nous paraît assez académique : on ne dispose à l'heure actuelle d'aucun algorithme de conception de systèmes : on procède par essais, en alternant des phases de décomposition et d'enrichissement, et on se fonde sur l'expérience pour limiter le nombre d'itérations nécessaires.

Si nous la mentionnons cependant, c'est parce qu'elle a conduit à la mise en oeuvre de deux séries de méthodes et d'outils de conception et d'écriture de systèmes. L'idée de décomposition a conduit à la notion de module, puis d'assemblage de modules ; l'enrichissement progressif a conduit à la notion de machine abstraite, puis de types abstraits. Modules et types abstraits ont été inclus dans des langages de programmation et sont couramment utilisés à l'heure actuelle. Cette distinction se retrouve également dans notre travail : dans ce chapitre, nous nous concentrerons plutôt sur la décomposition d'un système en modules; la notion de type abstrait sera abordée au chapitre 5. Il est clair que cette séparation est assez floue, et comporte une certaine part d'arbitraire.

Le découpage d'un système peut intervenir à différents niveaux. Comme les textes écrits par les programmeurs passent toujours par des phases de traduction ou d'édition de liens, il est possible de concevoir des décompositions au niveau des programmes sources, décompositions qui n'existent plus au moment de l'exécution. C'est à notre sens regrettable car on peut tirer parti du découpage pour limiter la propagation des erreurs. (Une protection de haut niveau contre les erreurs peut cependant jouer le même rôle).

Nous n'allons pas essayer dans ce chapitre de montrer un exemple de conception d'un système complet (on pourra cependant consulter [Cheval 77] pour avoir une idée des méthodes que nous préconisons). Notre propos est plus modeste. Dans un premier temps, nous allons répertorier les différents moyens de décomposition que l'on peut songer à utiliser ; nous montrerons ensuite, comment les utiliser pratiquement. Dans un dernier point, nous étudierons, là encore à partir d'exemples, le processus de construction et de mise au point des systèmes, c'est-à-dire l'ensemble des méthodes et des outils que l'on utilise pour réunir progressivement les différentes parties de façon à avoir un système complet.

Remarque : Lorsque dans ce chapitre nous parlerons d'assemblage de parties, il s'agira toujours de la réunion de ces parties pour former un ensemble plus gros. En conséquence, nous parlerons de compilation pour désigner la traduction d'un langage "d'assemblage".

## 4.2. LES PRINCIPAUX MOYENS DE DECOMPOSITION

Nous avons classé les moyens de décompositions en deux groupes logiques : ceux employés dans les programmes séquentiels et ceux employés dans les programmes parallèles ; nous traiterons ensuite des compilations séparées que, par opposition aux autres moyens, on pourrait qualifier de technologiques.

### 4.2.1. Programmes séquentiels

a) Les macro-instructions

Dans les langages de bas niveau, et en particulier dans les langages ne comportant pas la notion de type de données, les macro-instructions sont à peu près le seul outil de structuration des programmes. Elles apportent un certain palliatif à l'absence de la notion de type.

Leur rôle en tant que moyen de décomposition est assez modeste :
- il n'en subsiste aucune trace après compilation,
- leur emploi judicieux est soumis à la bonne volonté des programmeurs.

b) Les procédures

Introduites au départ pour des problèmes de réutilisation de parties communes, les procédures sont devenues l'outil essentiel de conception des programmes par raffinements successifs [Dijkstra 72, Wirth 73].

Elles permettent de distinguer la définition d'une action (faisant éventuellement intervenir des paramètres) de son utilisation. Ainsi, les variables ou les procédures locales à la procédure considérée ne sont pas visibles de l'extérieur et peuvent être changées sans que l'on ait à modifier le reste du programme.

On peut prévoir de plus d'inclure systématiquement des contrôles de validité des paramètres de façon à limiter la propagation des erreurs à l'exécution.

Sans remettre en question l'intérêt de la notion de procédure, son emploi en tant qu'unité de décomposition d'un système est limité : on a rarement besoin de définir une procédure isolément. Considérons l'exemple très simple ci-dessous, qui montre bien les limites de la notion.

Soit un programme de traitement d'une suite de caractères. Il est souvent commode de définir la suite par deux fonctions.

*function carcour : char ; {donne le caractère courant}*

*procedure carsuiv ; {fait passer au caractère suivant}*

Remarque : *carcour* peut-être une variable de type caractère au lieu d'une fonction rendant un résultat de type caractère.

Supposons en outre que les caractères à traiter se trouvent physiquement sur des cartes et qu'on dispose d'une procédure de lecture de carte :

*procedure lirecarte (var buf : array 1..80 of char).*

Pour programmer les procédures *carcour* et *carsuiv*, il faut alors disposer d'une zone de mémoire *tampon* pouvant contenir une carte et d'un pointeur *p* vers le caractère courant dans cette zone. Lorsqu'on atteint la fin d'une carte, la procédure *carsuiv* provoque automatiquement la lecture de la carte suivante.

Dans les langages à structure de blocs, les variables *tampon* et *p* doivent être globales aux procédures *lirecarte*, *carcour* et *carsuiv* et seront souvent en pratique déclarées au niveau le plus externe.

En conséquence, on pourrait y faire référence de n'importe quel point du programme, alors qu'on a pris soin d'introduire *carcour* et *carsuiv* pour s'abstraire des détails matériels de la chaîne à traiter.

Remarque : En PL/1, on aurait pu déclarer *tampon* et *p* avec les attributs *static external*, mais il faut répéter la déclaration dans chacune des procédures où ces variables sont utilisées, d'où un nouveau risque d'erreur.

Une solution satisfaisante à ce problème est fournie par la notion de classe, enrichie ensuite pour conduire au module.

c) Les classes

Une classe de SIMULA [Dahl 70] est un modèle permettant de construire, soit à la compilation, soit dynamiquement, des ensembles d'objets de mêmes propriétés. Plus précisément, on trouve dans une classe des déclarations de variables et de procédures opérant sur ces variables (on exprime ainsi qu'une variable est commune à plusieurs procédures).

Lorsqu'on a créé un objet d'une classe, les constituants de cet objet, c'est-à-dire les variables aussi bien que les procédures sont accessibles de l'extérieur. Autrement dit, les variables représentant un objet d'une classe peuvent être modifiées par des procédures distinctes de celles définies dans la classe, ce qu'il serait intéressant d'exclure pour des raisons de clarté et de protection. Cette restriction a été effectuée dans les versions plus récentes de SIMULA (attribut *hidden*) ; on en arrive ainsi à la notion de module.

d) Les modules

Un module est un programme dont seuls certains objets (dits externes) sont accessibles de l'extérieur. Ces objets externes peuvent a priori être exécutables (procédures) ou non (données). Partant de l'idée qu'une donnée ne doit être définie (pour des raisons de sécurité) qu'en même temps que les opérations autorisées sur elle, nous restreindrons, par convention, les objets externes d'un module à être des procédures (pouvant être des fonctions, c'est-à-dire fournir une valeur appartenant à un type spécifié). Cette convention revient à interdire la modification d'une donnée d'un module autrement que par l'appel d'une procédure externe, la lecture étant également assurée par un appel de fonction. Une convention équivalente [Parnas 72a] consiste à admettre des données externes accessibles en lecture seule depuis l'extérieur du module. Les paramètres d'une procédure externe ne peuvent être passés que par valeur.

Un module comprend donc :

- un ensemble de procédures externes, accessibles depuis l'extérieur du module
- un ensemble de procédures } internes, inaccessibles directement depuis
- un ensemble de données } l'extérieur du module.

La définition des procédures externes (nom et éventuellement type, nombre et type des paramètres, description de l'effet) constitue l'interface du module. Le terme de spécification est également employé. Lorsque nous parlerons de "spécification(s)", sans préciser, ce terme sera synonyme d' "interface" ; le terme de "spécification de réalisation" s'applique à la description d'une réalisation particulière d'un module.

Avant la première utilisation d'un module, ses données internes doivent être dans un état initial bien défini. La définition de cet état doit faire partie des spécifications du module, soit directement, soit comme résultat de l'exécution d'une procédure d'initialisation spécifiée.

A tout instant (en dehors de l'exécution d'une procédure externe), les variables internes du module ont des valeurs dont l'ensemble définit l'état du module à cet instant. Une autre façon de définir l'état, qui ne fait pas intervenir les variables internes, consiste à donner la suite d'appels de procédures (avec les valeurs des paramètres) entre l'instant initial (ou tout instant où l'état est déjà défini) et l'instant actuel. L'état d'un module résume donc, en ce qui concerne son comportement futur, toute l'histoire antérieure du module.

La notion de module n'a pas encore été largement intégrée aux langages de programmation. Elle était introduite dans le PL/1 modifié de MULTICS [Clark 71]. En dehors de SESAME, on peut citer en particulier MODULA [Wirth 77] et EUCLIDE [Lampson 77].

Remarque : Le terme module est souvent employé de façon vague (module d'assemblage, module de chargement). Dans ce travail, nous nous efforcerons de le restreindre à la définition donnée ci-dessus, sauf dans quelques cas mentionnés explicitement.

### 4.2.2. Programmes parallèles : les processus

Considérons maintenant des programmes dans lesquels certaines phases peuvent s'exécuter simultanément. On peut se ramener au cas précédent en distinguant des fonctions à exécution séquentielle, les processus. On doit alors faire face à deux problèmes :

- décomposer un système en processus, et définir les communications et les relations de synchronisation entre ces processus,
- décomposer les programmes en unités élémentaires (procédures, modules), indépendamment du nombre des processus mis en jeu.

On dispose donc de deux décompositions, qui doivent être compatibles : on ne peut imaginer, pour des raisons d'efficacité évidentes, qu'au cours de l'exécution d'un processus, on passe à chaque instruction d'un module à un autre. Nous discuterons sur des exemples (cf. § 4.3.), des différents choix possibles. Il nous faut par contre insister sur certaines difficultés introduites par le parallélisme, ainsi que sur certaines ambiguïtés de la notion de processus.

1) Problèmes de communication

Tant que les unités communicantes sont des procédures ou des modules appelés séquentiellement, les problèmes de communication sont bien connus : on peut communiquer soit par variables globales, avec les effets de bord que cela peut impliquer, soit par paramètres, avec différents modes de passage. Le problème se complique lorsque les objets communicants sont des processus de durées de vie quelconques. Il est facile de communiquer un message à un processus par une technique de boîte aux lettres [CROCUS 75], ou de créer un processus en lui passant des paramètres par valeur. Tous les autres modes de communication sont dangereux, car ils risquent de conduire à une tentative d'adressage d'objets disparus.

Par exemple, dans la Burroughs B 6700 [ORGANICK 71], un processus peut désigner des variables du processus qui l'a créé. Si ce dernier processus disparaît, ou simplement quitte le bloc dans lequel il a créé son fils, des variables théoriquement accessibles n'existent plus.

2) Différentes utilisations de la notion de processus

Comme la notion de processus constitue un outil commode de décomposition et d'analyse de système, on l'utilise souvent sans que cela soit imposé par la prise en compte des simultanéités d'exécution de divers processeurs.

On s'abstrait alors de contraintes séquentielles non intrinsèques : dès que des activités pourraient se dérouler en parallèle si l'on disposait d'assez de processeurs, on associe à chacune d'elles un processus. Ainsi, dans le système MAS [Briat 76], toutes les fonctions sont exécutées de façon asynchrone par des processus communicant par boites aux lettres.

De même, lorsqu'on dispose d'un calculateur comportant une unité centrale et une unité d'échange multiple, on associe souvent un processus à chaque opération d'entrée-sortie, bien que l'unité centrale soit employée très rarement, et en réponse à la même interruption d'entrée-sortie.

On constate donc que les systèmes comprennent un nombre de processus élevé ; à un instant donné, la plupart d'entre eux sont bloqués. La perte d'efficacité qui en découle est amplement compensée par la clarté de description et la sûreté de fonctionnement qu'apportent les processus.

### 4.2.3. Les compilations séparées

En règle générale, un système n'est pas compilé en une seule fois. Différentes parties sont compilées indépendamment les unes des autres, puis réunies dans une ou plusieurs opérations d'éditions de liens. L'introduction de compilations séparées a bien sûr des justifications.

- Différents langages peuvent être utilisés pour écrire un système (on peut programmer la plupart des algorithmes en langage de haut niveau, et en langage machine les quelques programmes dépendant de la structure du calculateur dont on dispose).
- Au cours de la mise au point d'un système, les modifications sont fréquentes et on doit diminuer le coût de ces modifications. Comme une édition de liens est souvent plus rapide qu'une compilation, on préfère n'avoir à compiler qu'une partie du système, les programmes modifiés, et effectuer une édition de liens globales.
- Différentes parties d'un système peuvent s'exécuter dans des espaces d'adressage différents (espace physique et espaces virtuels par exemple). A chacun de ces espaces doit correspondre au moins une édition de liens, et donc une compilation.
- Même dans le cas d'un système écrit en un seul langage (dont on dispose d'un compilateur rapide) et s'exécutant dans un seul espace d'adressage, une compilation unique est peu envisageable en pratique : un compilateur ne peut traduire que des programmes de taille limitée, et cette limite est en général inférieure à la taille d'un système d'exploitation, même simple.

Pour toutes ces raisons, l'écrivain de systèmes a besoin des compilations séparées. Il n'est pas question d'en nier les inconvénients, que nous allons développer maintenant, et auxquels il faut trouver des palliatifs.

Les compilations séparées présentent deux classes d'inconvénients : elles imposent la duplication de certaines portions de programmes, et empêchent certaines vérifications effectuées normalement par un compilateur.

- duplication : Les différentes compilations d'un même système ont en commun des déclarations de constantes de types et de variables globales. Lorsqu'on effectue une modification de ces parties communes, il importe de répercuter la modification dans toutes les parties. Sinon, on introduit une erreur qui sera par la suite difficile à expliquer.

  Une solution consiste à ne gérer qu'un seul texte de ces parties communes, et à faire une référence explicite à ce texte à chaque compilation (cf. 4.4.3)

- vérification : Dans une déclaration d'un langage de programmation, on peut associer à une variable un certain type qui définit les opérations que l'on peut effectuer sur la variable. Il est communément admis que la notion de type apporte une certaine sécurité de programmation. Dans le cadre de compilations multiples, des problèmes surviennent :

  - si un type non primitif est défini dans un programme, comment peut-on s'assurer que c'est bien un même type qui est défini dans deux parties compilées séparément.

  - les éditeurs de liens effectuent en général la correspondance entre les symboles externes en ne considérant que des identificateurs, à l'exclusion de tout type. On peut dès lors définir dans un programme une procédure $X$ et $y$ faire référence dans un autre comme à une variable entière. Une erreur ne sera détectée (peut-être ?) qu'à l'exécution.

Toute solution à ces problèmes implique :

- que des types puissent être définis dans un univers de durée de vie supérieure à une compilation,
- que la concordance des types soit vérifiée lors de l'assemblage des différentes parties.

### 4.2.4. Récapitulation

En résumé, un concepteur de systèmes a donc à sa disposition des unités élémentaires de décomposition de programmes : procédures, modules, processus et compilations séparées. Nous ne donnons pas ici de fil directeur pour arriver à une décomposition satisfaisante, dans la mesure où il en existerait une et où il serait

possible de donner une définition acceptable par tous de "la" décomposition satisfaisante. Contentons nous de mentionner les points que nous considérons comme essentiels pour juger d'une décomposition.

1) Le premier rôle d'une unité de décomposition est de définir une certaine "machine abstraite", indépendamment de sa réalisation pratique. Un programme sera d'autant plus adaptable que les différentes unités qui le composent seront d'emploi plus général (au prix peut-être d'une légère perte d'efficacité).

2) Plus les spécifications d'une machine abstraite - son interface - seront simples, plus la machine pourra être l'objet d'une réalisation et d'une mise au point autonomes.

3) Lorsqu'on peut faire des hypothèses sur les modifications les plus probables du programme, il faut en tenir compte dans le découpage de façon à ce qu'une modification ne concerne qu'un petit nombre de parties. Nous verrons en 4.3.1. un contre exemple.

4) Si l'on dispose d'outils d'assemblage bien adaptés, un critère fondé sur la taille de l'unité élémentaire n'aura que peu d'intérêt. A notre avis, c'est l'inadaptation des langages de commandes actuels qui conduit à diviser les systèmes en unités de grande taille. Des unités dont le programme occupe quelques feuilles d'imprimantes sont beaucoup plus agréables à écrire. S'il est possible de les assembler progressivement, la tâche de mise au point globale n'en sera pas alourdie.

5) L'accent doit être mis sur la clarté de la décomposition et sur les vérifications effectuées lors des assemblages (concordance de types).

## 4.3. EXEMPLES

Pour illustrer les notions précédemment introduites, nous avons retenu trois exemples : un compilateur du langage PASCAL, un système d'exploitation et un programme modulaire construit à l'aide de SESAME.

### 4.3.1. Un programme séquentiel : le compilateur PASCAL-SFER

Le compilateur du langage PASCAL [Jensen 74] développé dans le cadre du projet SFER est un compilateur en un passage dirigé par une grammaire LL(1). Le découpage en procédures est assez immédiat : à chaque symbole non terminal est associée

une procédure chargée de son analyse ; ces différentes procédures s'appellent, éventuellement récursivement. La production de code objet se fait au fur et à mesure de l'analyse, par l'intermédiaire de procédures ad hoc.

D'où une première décomposition du compilateur en un ensemble de procédures. Pour des raisons d'efficacité, la communication entre procédures se fait le plus possible par l'intermédiaire de variables globales.

A ce niveau, on peut faire deux critiques :

- on a sacrifié à l'efficacité la sûreté qu'apportent les passages de paramètres;
- la production de code-objet est disséminée dans toutes les procédures d'analyse.

De plus, le compilateur est écrit en quatre parties compilées séparément :

- l'une écrite en META-SYMBOL IRIS 80 réglant les problèmes d'entrée-sortie et d'interface avec le système d'exploitation SIRIS 8 de l'IRIS 80,
- trois écrites en PASCAL :
  - une racine comprenant des déclarations de tables globales et les procédures d'analyse lexicographique et de production de code objet,
  - le traitement des déclarations,
  - le traitement des instructions.

Pour limiter la taille de mémoire occupée par le compilateur, ces deux dernières parties se recouvrent en mémoire.

Le choix de ce découpage et de la structure de recouvrement découle de la structure d'un programme PASCAL : un programme (ou une procédure) se compose d'une suite de déclarations, puis d'une suite d'instructions; en conséquence, la composition de la mémoire ne sera modifiée que lors de la compilation des déclarations de procédures.

Les variables communes aux trois parties sont déclarées dans la racine (elles sont nombreuses puisqu'on les utilise pour la communication entre procédures). Leurs déclarations doivent être répétées dans les deux autres parties de façon à ce que le compilateur connaisse les types de toutes les variables. Donc à chaque changement de caractéristique d'une variable commune, il faut répercuter manuellement la modification dans les deux autres parties.

Ce découpage simple du compilateur pose quelques problèmes de modification : s'il est assez facile de modifier la syntaxe du langage source, le changement de machine objet implique des corrections dans l'ensemble du compilateur. On aurait pu préparer cette classe de modifications en produisant du code objet pour une machine fictive et en traduisant le langage de la machine fictive en celui de la machine objet dans une partie bien définie du compilateur.

Nous avons donc montré que dans le cas d'un compilateur, le choix d'une méthode de compilation et la définition d'un langage conduisent assez naturellement à une décomposition. Nous allons maintenant constater que dans le cas d'un système d'exploitation, les règles de décomposition sont plus floues.

### 4.3.2. Décompositions du système ESOPE

Le système ESOPE [Ferrié 72] a été écrit dans un langage du type PL 360, le LP 10070 ; il est possible de compiler séparément des portions de programmes qui seront reliées dans une phase d'édition de liens. Le système a été conçu et fonctionne comme une famille de processus coopérants ; l'ensemble des processus est partitionné en classes appelées usagers. Enfin, les programmes peuvent s'exécuter selon deux modes d'adressage, adressage absolu ou adressage virtuel par l'intermédiaire d'un mécanisme câblé de traduction dynamique d'adresse.

Nous trouvons donc dans ESOPE la plupart des unités de décomposition présentées en 4.2. et l'étude des différentes combinaisons de ces unités doit avoir une portée assez générale.

Examinons tout d'abord la décomposition en processus.

a) Processus (cf. [Ferrié 71], chapitres 3,4,5)

On distingue de prime abord dans ESOPE deux catégories de processus : ceux qui s'exécutent pour le compte des utilisateurs du système, dit processus usagers, et ceux qui assurent le partage des ressources entre utilisateurs, dits processus du moniteur. Les processus usagers s'exécutent en général en adressage virtuel, les processus du moniteur en adresses absolues. Considérons successivement ces deux catégories :

a.1. Processus usagers

L'existence d'au moins un processus par utilisateur se justifie pour rendre compte de l'indépendance logique qui existe entre les différents utilisateurs d'un système en temps partagé. Chaque utilisateur peut déclarer plusieurs processus; ces différents processus s'exécutent dans un même espace virtuel - l'ensemble d'un espace virtuel et des processus qui s'exécutent dans cet espace est un usager. Un usager est associé à chaque utilisateur du système, de plus un usager particulier gère les périphériques lents.

A l'intérieur de chaque usager, on distingue le processus premier, créé en même temps que l'usager et jouant le rôle du moniteur non résident, des autres processus ou traducteurs. Il n'y a aucune simultanéité entre ces processus, le processus premier se bloque dès qu'il lance un traducteur; dès qu'il reprend le contrôle, il commence par arrêter tous les autres processus de l'usager.

La distinction entre processus premier et traducteur n'a donc rien à voir avec le parallélisme. Elle est introduite pour des raisons de sécurité (le processus premier reprend le contrôle en cas d'erreur) et de protection (les différents processus n'ont pas les mêmes pouvoirs).

En pratique, les usagers associés aux utilisateurs conversationnels comportent un seul processus traducteur. Il en va autrement de l'usager d'entrée-sortie.

a.2. Les processus de l'usager d'entrée-sortie [Baudet 72]

Dans l'usager d'entrée-sortie, on veut rendre compte du parallélisme physique des différents périphériques. L'usager d'entrée-sortie assure des transferts entre fichiers et périphériques : on associe à chaque transfert un couple de proces-

sus assurant respectivement la transmission fichier-mémoire centrale et la transmission mémoire centrale-fichier et on utilise plusieurs tampons de façon à assurer un parallélisme réel entre le disque et les périphériques.

Naturellement, il n'y a aucune simultanéité réelle au niveau de l'unité centrale, car le calculateur est monoprocesseur. Mais ces différents processus sont en général bloqués en attente de fin d'entrée-sortie et la simultanéité est réelle au niveau des canaux.

On aurait pu obtenir la même efficacité avec un seul processus d'entrée-sortie conservant sous forme de tables l'état des différents transferts (ce qu'on nomme habituellement le superviseur d'entrée-sortie). Nous pensons cependant que la clarté de la description et la facilité de programmation qu'ont apportées l'introduction de nombreux processus justifient amplement ce découpage.

### a.3. Les processus du moniteur

Nous avons également essayé de décomposer en processus les différentes fonctions du moniteur :

- arrivée (processus initiateur) et départ (processus balayeur) d'un utilisateur
- lancement d'une demande de page (processus demande de page)
- gestion des transferts de pages (processus exécuter-requête et acquitter-requête),
- transfert du contrôle d'un usager à un autre (processus changement).

Cette décomposition nous paraît maintenant injustifiée pour différentes raisons :

- il n'y a aucune simultanéité réelle (monoprocesseur) ; même si l'on disposait de plusieurs processeurs, ces processus qui contiennent de nombreuses sections critiques pour l'accès aux tables communes, auraient peu de phases simultanées,
- le découpage est artificiel : pour donner une description claire de l'allocateur de ressources, on s'appuie, non pas sur les différents processus, mais sur les transitions d'états des différents usagers et des pages physiques de la mémoire [Mossière 71].
- Le découpage introduit certaines inefficacités. Nous avons introduit des priorités entre ces différents processus. La prise en compte de ces priorités conduit à des commutations superflues du processeur lorsqu'il y a contradiction entre les priorités et les règles d'exclusion mutuelle.

Un seul processus du moniteur aurait pu assurer les mêmes fonctions, avec une sélection de la tâche à effectuer selon la valeur d'un message d'entrée, de façon analogue à la solution retenue dans OURS [Briat 76]. Si nous avons introduit ce découpage, c'est parce que le processus était pour nous l'unité de décomposition "normale" d'un système, puisque seule disponible.

En résumé, les processus sont introduits pour trois raisons différentes :
- prise en compte du parallélisme logique,
- prise en compte du parallélisme physique,
- décomposition d'un programme.

Seules les deux premières raisons semblent justifiées, si toutefois, on dispose d'outils de décomposition adéquats.

La décomposition d'un système en processus parallèles, si elle facilite sa conception et sa compréhension, est matériellement insuffisante pour la réalisation : le programme est trop gros pour être compilé en une seule fois; il serait de plus peu économique de tout recompiler à chaque modification. Nous devons donc diviser le système en ensembles compilés séparément.

b) Compilations séparées

Une seule règle est impérative pour la définition de compilations séparées : il faut compiler séparément des programmes devant s'exécuter dans des espaces virtuels distincts. Pour le reste, la liberté est totale, et nous avons défini les différentes compilations de façon à limiter les références entre compilations, mais avec un critère supplémentaire souvent décisif : la portabilité physique des paquets de cartes.

C'est ainsi que :
- les processus du moniteur sont regroupés dans une seule unité de compilation, à l'exception de la gestion des disques;
- le processus premier des usagers conversationnels comporte trois parties compilées séparément;
- les processus d'entrée-sortie sont compilés en deux groupes : ceux assurant les transferts entre fichier et mémoire (qui s'exécutent en adresses virtuelles) et ceux assurant les transferts entre la mémoire et les périphériques, qui s'exécutent en adresses absolues.

Il faut également avoir recours à plusieurs éditions de liens :
- une pour les programmes s'exécutant en adresses absolues,
- une pour les programmes s'exécutant en adresses virtuelles.

Le problème principal vient des communications entre les différentes parties.

Les déclarations des constantes globales doivent être répétées dans chaque parties. De même, les variables globales sont situées dans des zones de mémoire accessibles aussi bien en adresses absolues que depuis chaque espace virtuel. La composition de ces zones communes est également définie par des valeurs de constantes.

Comme le langage d'écriture dont nous disposions ne permettait pas d'insérer dans les programmes des lignes prélevées dans un fichier, ces déclarations devaient être physiquement répétées dans chaque programme. D'où un risque d'erreur à chaque modification de ces déclarations, et de nombreuses erreurs effectives.

Les variables et les procédures définies dans un programme et utilisées dans un autre doivent faire l'objet :

- d'une déclaration dans le premier programme, spécifiant un nom, un type et précisant que le symbole nommé doit pouvoir être utilisé de l'extérieur,
- d'une référence dans le second programme, spécifiant elle aussi un nom et un type.

Mais l'éditeur de liens n'assure que la correspondance entre les noms et ignore les déclarations de types. On peut ainsi définir un objet dans un programme et l'utiliser avec un type différent de celui de sa définition, ce qui permet peut-être des astuces, mais fournit surtout une source d'erreurs difficiles à détecter.

En résumé, deux idées essentielles se dégagent de cette description :

- la notion de processus est bien adaptée à la description de problèmes de parallélisme, aussi bien logique que physique,
- les difficultés des compilations séparées proviennent d'une part du langage de programmation (l'absence d'un univers global aux compilations impose une répétition des éléments globaux), d'autre part, de l'éditeur de liens qui ne vérifie pas la concordance des types des symboles externes.

Dans SESAME, nous avons donc cherché à limiter l'emploi des processus à l'expression du parallélisme et à remédier aux insuffisances des compilations séparées (cf. 4.3.3.).

### 4.3.3. Processus, modules et compilations séparées dans les systèmes produits par SESAME.

Les modules de SESAME, ensemble de données et des procédures qui les manipulent correspondent à la définition donnée en 4.2.1. Cette notion est incluse dans le langage de programmation;de plus, le module est choisi comme unité de compilation.

Remarque : Ce choix peut être contesté; par exemple, le langage MODULA [Wirth 76] ne prévoit aucune compilation séparée. Notre justification est alors la suivante :

1) Les compilations séparées sont nécessaires à la construction d'un gros système (cf. 4.2.3.) ;
2) Un module, qui regroupe des données et les procédures de traitement de ces données doit être compilé comme un tout pour simplifier les vérifications de cohérence ;
3) En conséquence, une unité de compilation va contenir un nombre entier de modules. Pour des raisons de paramétrage, nous nous restreignons à la compilation d'un seul module (cf. 4.4.3.a.).

Dans SESAME, l'unité de décomposition des programmes, tant logique que technologique, est donc le module. De plus, on introduira des processus pour les deux raisons étudiées plus haut :
- exploitation du parallélisme réel entre diverses unités,
- expression de l'indépendance logique entre différents travaux.

Il nous reste à examiner comment on va introduire des créations et des destructions de processus par rapport aux appels et retours de modules. Il n'est naturellement pas question d'une indépendance complète entre les deux notions : créer un processus dans un module pour exécuter un algorithme figurant dans un second module, puis le détruire dans un troisième ne permet guère de déduction sur les propriétés du programme, surtout si les créations et destructions interviennent dans des instructions conditionnelles ou itératives.

Deux solutions cohérentes sont concevables.

a) Associer un processus à une activation de procédure. Deux appels de procédure doivent alors être inclus dans le langage de programmation, l'appel "synchrone" habituel et un appel "asynchrone" qui est en fait une création de processus ; le processus créé est détruit lors du retour de la procédure.

Cette solution permet de créer un nombre de processus variable selon l'exécution; par contre, le processus créateur et le processus créé s'exécutent simultanément, ce qui pose des problèmes de contrôle de l'accès aux objets partagés.

b) Disposer d'une instruction génératrice de parallélisme *(parbegin...parend)* dans le corps d'une procédure d'un module.

C'est cette deuxième solution que nous avons retenue, car elle nous paraît la plus conforme à l'idée de module : un module comprend la définition de fonctions opérant sur les données du module; on peut mettre en évidence des activités parallèles à l'intérieur de chaque fonction. De plus, les durées de vie et le nombre des processus créés sont indiqués clairement dans le texte des programmes. Enfin, comme le processus père (celui où figure le *parbegin-parend)* est bloqué pendant l'exécution des processus fils, on peut permettre aux processus fils d'accéder aux variables du père sans avoir besoin de mettre en place un contrôle coûteux sur les durées de vie des objets.

On pourra trouver dans [Cristian 77] un exemple de décomposition en modules d'un programme séquentiel et dans [Herodin 77] la description de la réalisation d'un système modulaire complet.

Dans un système modulaire, l'utilisation de la notion de processus est donc limitée. Quant à la façon dont les différents modules sont assemblés, nous allons l'étudier maintenant dans un cadre un peu plus général : la construction et la mise au point d'un système complet.

## 4.4. CONSTRUCTION D'UN SYSTEME - TECHNIQUES D'ASSEMBLAGE

Dès que les programmes atteignent une certaine complexité, leur mise au point est facilitée si on peut les composer progressivement, en essayant chaque composition partielle avant de l'enrichir. On obtient ainsi une méthode de construction par couches successives. Dans le cas des systèmes d'exploitation, sa mise en oeuvre est délicate : on doit en effet augmenter progressivement le nombre de procédures ou de modules et le nombre de processus. Différentes conceptions sont donc possibles, que nous allons illustrer maintenant.

*N.B. : dans les paragraphes 4.4.1. et 4.4.2., le terme module a un sens beaucoup plus flou que dans les paragraphes précédents. Dans le 4.4.1., il correspond simplement à un morceau de programme, avec des instructions et des données; dans le 4.4.2., il correspond à une unité de compilation.*

### 4.4.1. Extension par addition de processus

Le système THE [Dijkstra 68] est structuré en niveaux, chaque niveau pouvant être considéré comme la réunion du niveau inférieur et d'un nouveau module. Sa mise au point s'est effectuée niveau après niveau.

Ces niveaux ont les propriétés suivantes.

a) Au niveau le plus bas, on réalise sur la machine physique une machine à processus.

b) Lorsqu'on passe d'un niveau au niveau supérieur, on ajoute un certain nombre de processus.
   Exemple : au niveau 1, on implante la gestion du tambour, au niveau 2, le processus de gestion de la console de l'opérateur, qui utilise la gestion du tambour.

c) Les communications entre niveaux semblent se réduire à des appels aux processus implantés à des niveaux inférieurs, en utilisant des primitives de synchronisation (niveau 0, accessible à tous).

En terme de modules, on peut donc dire qu'un module particulier gère les différents processus; les communications entre modules se font par appel de processus. La solution est viable car THE est un système simple qui a pu être réalisé par un petit nombre (5) de modules.

### 4.4.2. Extension par enrichissement des algorithmes des processus

Le système ESOPE a également été réalisé par versions de complexité croissante. Les règles de passage d'une version à une autre n'ont pas été aussi systématiques que dans THE.

a) Dans toutes les versions a figuré un niveau bas de gestion du parallélisme ; ce programme a été mis au point une fois pour toutes et n'a plus évolué par la suite.

b) La version finale du système a été obtenue par l'addition à la version précédente (système conversationnel) d'un usager chargé de la gestion des périphériques lents.

c) Les versions de réalisation du système conversationnel comportaient les mêmes processus, avec les mêmes instructions de synchronisation, mais des algorithmes de plus en plus riches. Le passage d'une version à une autre se traduisait par le remplacement de la gestion statique d'une ressource par une gestion dynamique.

Exemples : l'avant dernière version comportait les allocateurs de mémoire et d'unité centrale définitifs, mais les espaces virtuels des usagers étaient fixés à l'initialisation du système. La dernière version a été construite en ajoutant une gestion de fichiers et des instructions de composition dynamique de l'espace virtuel.

De façon plus précise, la réalisation d'une nouvelle version était préparée par la mise au point de "sous-versions" comprenant des extensions très limitées; on opérait ensuite une fusion de ces différentes sous-versions.

Les résultats pratiques peuvent être résumés comme suit :

- L'introduction de l'usager de gestion des périphériques lents n'a en elle-même pas pas posé de problème. Les problèmes, comme pour tous les changements de version, sont venus de la mise à jour des tables et des constantes communes (introduction de nouveaux sémaphores en particulier).
- La mise au point du schéma global de synchronisation a été facilitée par le caractère rudimentaire des algorithmes des processus. Par la suite nous n'avons pas eu de problème de synchronisation.
- Les changements de versions étaient plus délicats, car ils concernaient pratiquement tous les modules du système. Il est vrai que ces opérations étaient rendues plus pénibles par l'absence d'outils adaptés de conservation et d'édition de programmes.

En résumé, les principales difficultés de construction d'ESOPE ont été provoquées par :

- l'absence d'un bibliothécaire pour la conservation et l'édition des programmes,
- l'impossibilité de pouvoir définir à un emplacement unique des constantes communes à plusieurs modules,
- la non vérification par l'éditeur de liens des types des symboles externes reliés.

Dans SESAME, nous avons donc cherché à supprimer ces difficultés.

### 4.4.3. Le langage de connexion de SESAME

a) Quelques éléments sur SESAME [Cheval 76b]

Pour produire des systèmes à l'aide de SESAME, un usager commence par écrire, dans un langage proche de PASCAL, des modules; ces modules sont ensuite assemblés pour former un système complet. La deuxième phase de la construction est l'objet de ce paragraphe; pour en permettre la description, il nous faut exposer brièvement la définition de nos modules.

Un module est une unité de construction de programmes. Il se compose d'un ensemble de données et de procédures de manipulation de ces données. Seules certaines de ces procédures, dites externes, sont accessibles de l'extérieur du module; les données d'un module ne sont accessibles que par les procédures de ce module.

L'un des objectifs du projet est la mise à la disposition des utilisateurs d'un ensemble de modules qu'ils pourront utiliser pour la constitution de leurs programmes (voir aussi [Corwin 72]). Pour que cette utilisation soit effective, il est utile de conserver non pas des modules, mais des générateurs, ou modèles de modules permettant d'engendrer toute une classe de modules. Un module particulier est défini par les valeurs, fixées à la compilation, d'un ensemble de paramètres. De plus, les procédures extérieures à un module ne sont pas toujours désignées par leur nom réel, mais peuvent l'être par un nom conventionnel, de signification locale au module. La procédure est alors dite fictive. Lors de la construction d'un système particulier, on associe au nom conventionnel le nom réel de la procédure que l'on souhaite effectivement appeler. Ce mode de désignation et le mécanisme de liaison associé sont analogues à ceux utilisés dans les systèmes classiques pour la désignation des fichiers (DCB).

Exemple :

```
pattern table (&nmax : integer ; &t1, &t2 : type) ;
dummy procedure erreur1 ; dummy procedure erreur 2 ;
      type pair = record clé : &t1 ;
                         info : &t2
                  end ;
      var n : integer := 0 ;              {nombre d'éléments dans la table}
          a : array [1..&nmax] of pair ; {représentation interne de la table}
      function index (c : &t1) : integer;{rechercher un élément de clé c}
      var i : integer ;
      begin
          i := 1 ;
          while (i<= n) and (a[i].clé ≠ c) do i := i+1 ;
          index := i ;
      end {index};
ext procedure entrer(c : &t1 ; inf : &t2) ;  {entrer un élément dans la table}
      var i : integer ;
      begin
          if n = &nmax then
                erreur1
          else
              begin
                  i := index(c) ;
                  if i<= n then
                     erreur2
                  else
                      begin
                          n := n+1 ;
                          with a[n] do
                          begin clé := c ; info := inf end
                          end
                      end
              end
        end {entrer};
ext procédure supprimer(c : &t1) ; {supprimer l'élément de clé c}
    :
ext function val(c : &t1) : &t2 ; {valeur du champ info d'un élément de clé c}
    :
end {table}.
```

Ce programme est le texte d'un modèle de module (pattern), qui décrit la gestion d'une table. Ce programme a les propriétés suivantes :

- les noms des procédures externes sont préfixés par *ext*.
- les noms des procédures fictives sont préfixés par *dummy*.
- les variables internes du module représentent son état (tel qu'il résulte de l'initialisation et des appels successifs des procédures externes). Ces variables internes sont inconnues des utilisateurs du module.
- un modèle de module n'est pas directement exécutable : c'est une description commune à une famille de modules. Chaque module de la famille peut être engendré par macrogénération, en affectant des valeurs aux métavariables (identificateurs dont le nom commence par & et dont la liste suit le nom du modèle de module).

b) Notion d'environnement

Pour éviter des erreurs de recopie, les déclarations communes à plusieurs modules ne doivent figurer dans le système qu'à un seul exemplaire. Dans un système modulaire, les variables représentent l'état des modules et les déclarations de variables sont donc locales à chaque module. En conséquence, les déclarations communes ne pourront correspondre qu'à des déclarations de constantes, de types, et, à la limite, de fonctions ou de procédures sans variable globale.

Tout usager de SESAME peut confier au bibliothécaire des groupes de telles déclarations, que nous appelons environnements. En tête des déclarations de chaque modèle de module, le programmeur peut spécifier une liste d'environnements. Les déclarations de ces environnements sont insérées dans le module par le compilateur, au même niveau que les déclarations globales du module.

Autrement dit, si un identificateur apparaît dans un environnement et dans un module, il y aura détection d'une double définition. Nous estimons en effet, que les environnements seront le plus souvent définis au niveau d'un projet et qu'un programmeur ne doit pas pouvoir modifier sa constitution; ceci pourrait arriver, volontairement ou non, si un environnement était défini à un niveau lexicographique externe au programme utilisateur, et si des règles de portée s'appliquaient.

Remarque : La notion d'environnement est analogue à celle de "prologue standard" d'ALGOL 68.

c) Le langage de connexion

A partir des éléments de construction que sont les modèles de modules, représentation des entités définies à la conception, le langage de connexion permet de définir :

- la liste des modules constituant le système produit, ainsi que les valeurs des métavariables permettant d'obtenir ces modules à partir des modèles correspondants,
- la liste des connexions a établir entre les modules, c'est-à-dire la mise en correspondance des noms de procédures fictives avec des noms réels de procédures externes.

Un programme de connexion est donc composé d'une suite de déclarations énumérant des modules et d'une suite d'affectations définissant des connexions entre ces modules. Compte-tenu de son originalité (voir cependant [De Remer 75]), nous allons faire une présentation détaillée de ce langage.

(i) - Instructions d'affectation

*<affectation> ::=*
*<nom de module>.<nom de proc fictive> := <nom de module>.<nom de proc>*

Une instruction d'affectation définit la procédure externe que l'on veut faire correspondre à une procédure fictive.

A l'interprétation d'une affectation, on vérifie que les types des paramètres et des résultats de la procédure fictive et de la procédure externe coïncident. Cette vérification est pour l'instant limitée aux types de bases et à ceux figurant dans les environnements.

(ii) - Déclaration

*<déclaration> ::= <identificateur> : module = <expression de module>*
*<expression de module> ::= <nom du modèle> {( <liste de métavariables>)}*
*| !<nom de module>*

Une déclaration associe un nom à un module. Ce module peut être soit un module binaire de bibliothèque *(!<nom de module>)*, soit le résultat de la compilation d'un modèle de module.

L'interprétation d'une déclaration de la première forme est un appel au compilateur du langage d'écriture des modules.

(iii) Définition du point d'entrée

*<point d'entrée> ::= start <nom de module>.<nom de proc. externe>*

L'exécution du système produit doit commencer par la procédure nommée dans l'instruction start.

Exemple :

On dispose en bibliothèque de deux modèles de modules :

```
pattern table (&N : integer ; &T : type) ;
        dummy procedure error ;
        ext procedure entrer ; ...
        ext procedure sortir ; ...
        .
pattern exemple (&Z : integer) ;
        dummy procedure lire ;
        dummy procedure écrire ;
        ext procedure vasy ; ...
```

et d'un module binaire *erreur* contenant une définition de procédure externe de nom *faute*. On peut utiliser ces composants dans le programme de connexion ci-dessous :

```
tab  : module = table (50, real) ;
ex   : module = exemple (100) ;
err  : module = ! erreur ;
tab.error := err.faute ;
ex.lire := tab.entrer ;
ex.écrire := tab.sortir ;
start ex.vasy ;
```

(iv) Tableaux de modules. Instruction itérative

Les systèmes comportent souvent des familles de composants remplissant des fonctions voisines (gestion de périphériques par exemple). Il est commode de désigner par un seul nom tous les modules d'une telle famille, et de sélectionner l'un d'eux par indexation. On introduit pour cela la notion de tableau de modules.

*<decl. de tableau> ::= <identificateur> : array [<index>] of*
*module = <liste de couples>*
*<index>>::= <expr.entière> .. <expr.entière>*
*<liste de couples> ::= <couple> | <liste de couples>, <couple>*
*<couple> ::= <expr.entière> : <expr.de module>*
*|<index> : <expr. de module>*

Exemples :

1) *tab : array [2..3] of module =*
*2 : table (50, integer);*
*3 : table (100, real) ;*

définit un tableau de 2 modules compilés à partir du modèle *table*. Ces tableaux sont respectivement désignés par *tab*[2] et *tab*[3].

2) *table : array [1..100] of module =*
*1 : table (50, integer),*
*2.. 100 ; table (100, real) ;*

définit un tableau de 100 modules dont les 99 derniers éléments sont identiques.

Pour abréger l'écriture des connexions entre éléments de tableaux, on introduit une instruction itérative

*<instr.itérative> ::=*
*foreach <ident> in <qqchose> do <instruction> ;*
*<qqchose>>::= <index> | <identificateur de tableau>*

Exemple : *foreach i in tab do tab*[*i*] . *error := err.faute ;*

(v) Procédures de connexion

Les programmes de connexion introduits jusqu'à présent peuvent être considérés comme une suite de commandes qu'un utilisateur de SESAME tape à sa console, commandes qui sont immédiatement interprétées.

Même si l'on admet qu'un programme de connexion peut être conservé en bibliothèque, l'intérêt de cette conservation est discutable car le programme ne comporte aucun paramètre.

En conséquence, nous introduisons des procédures de connexion permettant de donner un nom à un groupe de déclarations et d'instructions de connexion puis d'utiliser des groupes préalablement définis.

Ces procédures de connexion peuvent comporter des paramètres (identificateurs commençant par &). Lors de l'appel d'une procédure, la correspondance paramètre formel - paramètre effectif est effectuée par remplacement lexicographique

```
<déclaration de procédure> ::=
  proc <identificateur> (<liste de paramètres formels>) ;
       <texte de connexion>
  pend
```

Exemple : Le programme de connexion

```
proc tableau (&x : integer ; &bidon : procedure) ;
     tab : module = table (50, real) ;
     ex : module = exemple (&x) ;
     tab.error := &bidon ;
     ex.lire := tab.entrer ;
     ex.écrire := tab.sortir ;
pend ;
err : module = ! erreur ;
tableau (100,err.faute) ;
start ex.vasy ;
```

est équivalent à celui donné en (iii).

Les procédures de connexion sont des objets conservés par le bibliothécaire.

## 4.5. CONCLUSION

Nous avons étudié dans ce chapitre différents constituants des systèmes d'exploitation (et plus particulièrement les modules et les processus) et les problèmes que pose l'assemblage de ces constituants pour la construction d'un système.

Les exemples étudiés montrent qu'on ne peut pas se contenter d'un seul type de constituant : deux unités de décompositions logiques sont nécessaires, l'une pour diviser le texte des programmes, l'autre pour rendre compte des activités parallèles ; enfin, la construction pratique d'un système d'exploitation, comme de tout logiciel réaliste, implique que l'on puisse compiler séparément les différents constituants, pour les assembler dans une phase d'édition de liens.

On peut alors dégager deux règles :

Règle 1 : Lorsque plusieurs moyens de décomposition doivent coexister, faire jouer à chacun d'eux le rôle pour lequel on l'a introduit.

Règle 2 : Pour constuire un système, ne pas se contenter d'appliquer une méthodologie de production, mais intégrer cette méthodologie dans des outils adéquats.

Ces deux règles sont respectées dans SESAME : la seule unité de décomposition est le module, qui est en outre l'unité de compilation ; les processus servent à exprimer l'existence d'activités parallèles, logiques ou physiques. Enfin, l'utilisation d'un langage de connexion permet la construction progressive des systèmes d'exploitation.

# CHAPITRE 5

# SUR LES LANGAGES D'ECRITURE DE SYSTEMES

## 5.1. INTRODUCTION

Il est généralement admis que la structure du langage utilisé influe considérablement sur le mode de pensée de ceux qui l'utilisent. Il en est ainsi en programmation où l'emploi d'un langage (de programmation) adapté aux problèmes que l'on cherche à résoudre simplifie la conception et la mise au point des programmes, et améliore leur qualité.

Nous avons examiné au chapitre 2 les caractéristiques de la programmation des systèmes d'exploitation. Rappelons en brièvement les conclusions :

- les systèmes d'exploitation sont de gros programmes en évolution constante, qui doivent donc être clairs et aisément modifiables,
- on doit décrire des activités parallèles ;
- les données manipulées sont de structures très variables et peu prévisibles, leur implantation exacte est parfois imposée par le matériel ;
- toutes les instructions spéciales (entrées-sorties, traitement des interruptions, ...) doivent pouvoir être utilisées.

Trois classes de langage sont utilisées pour l'écriture des systèmes d'exploitation, énumérons les par ordre d'éloignement croissant du langage de la machine [CROCUS 75].

1) Langages d'assemblage

Les opérations de ces langages, celles d'une machine donnée, sont trop élémentaires pour décrire des algorithmes de façon claire. Les données manipulées se limitent à des chaînes de bits de longueurs fixées par construction (octet, demi-mot,...). Il est possible en langage d'assemblage d'exploiter toutes les possibilités d'une machine, mais au prix d'un travail assez fastidieux. De plus, ces langages encouragent les programmeurs à faire preuve de virtuosité, ce qui obscurcit les programmes.

L'utilisation systématique d'un macro-assembleur permet de remédier partiellement à ces défauts :

- on peut définir les accès à des données structurées par une famille de macro-instructions ; lorsqu'on modifie la structure, il suffit de mettre à jour les macros correspondantes.
- le respect de normes de programmation est facilité si l'on met à la disposition des programmeurs une bibliothèque de macro-instructions (appel et retour de procédures, gestion d'une pile de travail, ...).

2) Langages de type PL 360

Le langage PL 360 [Wirth 68] et les langages analogues constituent une solution de compromis entre les langages d'assemblage et les langages évolués. Si la structure d'ensemble (instructions de séquencement, structure de blocs, affectations) est celle d'ALGOL, on a conservé la pauvreté des structures de données des assembleurs. L'absence (voulue) de gestion à l'exécution interdit les procédures récursives et impose un passage de paramètres par registres. Par contre, toute instruction du répertoire de la machine peut être engendrée et le code objet correspondant à chaque instruction est connu précisément.

Pour la construction d'ESOPE, nous avons utilisé un langage de cette famille, le LP 10070. Une évaluation détaillée de notre expérience est faite dans [Boulenger 74a, 74b] . Rappelons-en quelques conclusions.

Par rapport aux langages d'assemblage, le gain en clarté, en rapidité d'écriture et de mise au point est considérable. Des inconvénients subsistent : gestion "manuelle" des registres, structures de données et de procédures rudimentaires, absence de vérifications dynamiques.

Des améliorations sont bien sûr possibles sur chacun de ces points, mais on préfère actuellement s'orienter vers des langages évolués.

3) Langages évolués

Les premières tentatives d'utilisation de langages évolués pour l'écriture de systèmes d'exploitation sont à notre connaissance la production des systèmes MCP BURROUGHS B5500 (en ALGOL étendu [Burroughs 64]) et MULTICS (en EPL, sous ensemble de PL/1 [Corbato 69]). Aujourd'hui, de nombreux langages ont été définis, et continuent de l'être, en vue de l'écriture de systèmes d'exploitation. Examinons donc les principales caractéristiques de ces langages et la nature des recherches en cours.

En ce qui concerne les instructions, un consensus s'est à peu près dégagé; l'effet des instructions doit être immédiatement apparent et permettre l'établissement statique de propriétés des programmes; les effets de bord, qui rendent les programmes peu compréhensibles, sont à proscrire lorsque certaines séquences sont exécutées en parallèle; les instructions d'itération *tantque... faire* et *répéter ... jusqu'à* sont communément admises. Quelques problèmes restent en suspens :

- La querelle "idéologique" de la proscription du *aller à* : doit-on conserver certaines utilisations de l'instruction de branchement inconditionnel, ou imaginer des substituts. Ce point est développé dans [Knuth 74].

- La définition de certaines instructions d'itération *(foreach)* ou de traitements d'erreur.

La situation est beaucoup moins nette en ce qui concerne les objets manipulés : on constate que d'une part la structure de bloc, d'autre part la définition a priori de quelques types primitifs et de règles de composition (tableaux, structures) sont insuffisantes. L'effort actuel porte sur la définition de classes d'objets de façon telle que les détails de la représentation des objets soient cachés aux utilisateurs et que les opérations exécutables sur l'objet soient incluses dans la définition. Aucune solution ne s'est encore imposée.

Nous allons donc centrer ce chapitre sur la définition des structures de données dans les langages d'écriture de systèmes. Pour celà, nous définissons tout d'abord les différentes notions utiles; nous situons ensuite, dans le cadre fourni par ces définitions, les principales études actuelles. Nous traitons enfin des questions d'efficacité et de mise en oeuvre des particularités physiques des machines (inclusion d'instructions non classiques dans un langage de haut niveau).

## 5.2. DEFINITION DE STRUCTURES DE DONNEES DANS LES LANGAGES EVOLUES

Les différents concepts utilisés pour la définition de structures de données sont assez rarement définis avec précision. De plus, des auteurs différents désignent des objets différents sous le même nom. Dans ces conditions, nous allons commencer notre étude par une définition des principales constructions introduites. Ce paragraphe est un résumé de [Mossière 74], note interne non diffusée.

Remarque : La terminologie adoptée est propre à l'auteur : il ne saurait en être autrement puisque le même nom peut recouvrir des concepts différents dans des articles distincts. Nous espérons que la lecture de ce chapitre n'en sera pas rendue trop ardue.

### 5.2.1. Types

#### 5.2.1.1. Types et variables

Soit $E = \{e_i\}$ un ensemble d'éléments $e_i$. On dit que $e_i$ est une valeur du type E, ou encore une constante de type E.

Dans un langage de programmation, une variable X est dite de type E si elle ne peut repérer qu'une valeur de type E. On associe un nom de variable à un type au moyen d'une déclaration. On postule l'existence d'une opération primitive sur les variables, l'affectation, qui a la forme :

<nom de variable de type E> := <valeur de type E>

#### 5.2.1.2. Opérateurs - expressions

Une loi de composition interne sur E est une application de $E \times E$ dans E. Un opérateur sur E sert à dénoter une loi de composition interne. A l'aide de valeurs d'opérateurs et de règles de parenthésages ou de priorité, on peut former des expressions sur E, dont le résultat est encore une valeur appartenant à E.

On utilise en général, par conservatisme, la notation infixée :

<opérande> <opérateur> <opérande>.

Il est possible de lui substituer une notation préfixée :

<opérateur> <opérande> <opérande>

ou fonctionnelle :

<opérateur> (<opérande>,<opérande>).

on peut alors utiliser des opérateurs à nombre quelconque d'opérandes et dont chaque opérante appartient à un type particulier (application de $E_1 \times E_2 \times \ldots \times E_n$ dans $E_{n+1}$).

5.2.1.3. Types primitifs - types construits

Nous postulons maintenant l'existence d'un certain nombre de types dits types primitifs, ainsi que des opérateurs permettant d'effectuer des calculs sur les valeurs des types primitifs. Lorsque les types primitifs sont inadéquats pour la résolution d'un problème, une solution consiste à offrir des moyens de définition de nouveaux types, dits types construits, à partir des types primitifs.

La définition de nouveaux types implique :
- la possibilité de définir de nouveaux ensembles de valeurs,
- la possibilité de définir des opérateurs sur ces valeurs.

Par contre, les définitions des déclarations de variables et de l'instruction d'affectation sont inchangées.

Pour définir de nouveaux ensembles de valeur, deux méthodes de construction sont surtout utilisées :
- le tableau permettant de considérer globalement un ensemble de n valeurs d'un même ensemble E. On retrouve ainsi le produit cartésien $E^n$.
- la structure permettant de considérer globalement un ensemble de n valeurs, chacune d'ellesappartenant à un certain ensemble $E_i$ (i allant de 1 à n). On retrouve là encore le produit cartésien

$$E_1 \times E_2 \times \ldots \times E_n.$$

Les deux notions (structure et tableau) ne sont donc pas fondamentalement distinctes; leur distinction est plutôt une question d'habitude et de syntaxe.

Pour utiliser des éléments de types construits, il nous faut disposer de deux outils complémentaires :
- la sélection d'une des valeurs constitutives d'une valeur composée (par indices pour les tableaux, par champs pour les structures),
- la définition d'opérateurs portant globalement sur les valeurs composées (fonctions dans lesquelles on utilise les sélecteurs et les opérateurs primitifs).

### 5.2.1.4. Conclusions sur la notion de type - Ses limites

Nous avons défini un type comme un ensemble de valeurs et deux classes d'objets figurant dans les programmes, les constantes et les variables. L'intérêt de cette approche vient de son caractère élémentaire : aucune hypothèse n'est faite (et donc aucune restriction n'est apportée) en ce qui concerne d'une part les associations entre types et opérateurs, d'autre part les problèmes de création, de destruction et de visibilité des variables.

a) On peut regrouper dans une même définition de type ensemble de valeurs et opérateurs, et restreindre les opérations sur le type à celles figurant dans la définition. Cette restriction n'a d'intérêt qu'en ce qui concerne les variables, mais elle est alors imposée également sur les constantes. De plus elle pose des problèmes quand on veut définir un opérateur dont les deux opérandes sont de types différents : où doit-on placer la définition de l'opérateur, dans l'un des deux types, dans les deux ?

b) Les choix concernant les questions de durée de vie et de visibilité des variables sont délicats. Cette question est détaillée dans [Dijkstra 76].

   La structure de bloc qui définit en même temps la durée de vie et la visibilité de la variable est insuffisante : tous les objets de longue durée de vie doivent être déclarés au niveau externe, et sont dès lors accessibles dans l'ensemble du programme. De plus, il est possible de définir dans des blocs internes des variables de même nom que les variables globales : l'accès aux variables globales est alors impossible, ce qui offre une certaine sécurité, mais nuit à la clarté du programme.

Dijkstra propose d'énumérer en tête de chaque bloc les objets externes utilisés dans le bloc ; des solutions voisines ont été retenues dans MODULA [Wirth 77] et EUCLID [Lampson 77].

Nous avons préféré une solution un peu plus radicale, mais imposant une écriture moins lourde : une variable n'est visible qu'à l'intérieur d'une entité particulière, le module, dans lequel sont définies des variables et des procédures. Les procédures externes utilisées, comme celles accessibles de l'extérieur, sont énumérées dans la définition du module.

### 5.2.2. Les modules

Les notions de variables et de types construits permettent de construire des objets de structure quelconque. Cependant, tous les constituants d'un tel objet sont librement accessibles : les seuls contrôles de validité concernent la cohérence des types dans les affectations. On introduit la notion de module pour restreindre les opérations possibles sur un ensemble de variables.

#### 5.2.2.1. Définition

Un module est un doublet (S,F) dans lequel
S est un ensemble fini d'états,
$F = \{f_i\}$ est un ensemble de fonctions
$f_i : S \times P_i \rightarrow S$, $P_i$ étant l'ensemble des paramètres de la fonction $f_i$.

En pratique, l'état d'un module est représenté par un ensemble de variables, dites variables d'état du module, dont les types sont choisis selon les caractéristiques du module à construire. Les fonctions $f_i$ sont un ensemble de procédures ou de fonctions (au sens courant du terme dans les langages de programmation) qui permettent de consulter l'état du module, ou de provoquer un changement d'état. Les paramètres de ces procédures sont passés par valeur.

De l'extérieur d'un module, on ne peut atteindre les variables d'état que par l'intermédiaire des procédures du module. Le but recherché, la restriction des possibilités de transformation, est donc atteint.

Remarque : La définition ci-dessus est un peu différente de celle donnée dans [Parnas 72a] : Parnas autorise un accès en lecture seule aux variables d'un module. Deux raisons nous ont conduit à cette restriction supplémentaire : rendre l'utilisation d'un module indépendante de sa réalisation et simplifier les contrôles d'accès aux variables.

#### 5.2.2.2. Propriétés

a) Un module est une unité de construction de programme. En conséquence, tout module d'un programme existe en un seul exemplaire. Pour construire un programme (ou un système) complet, on assemble des modules préexistants. Cet assemblage peut intervenir soit à la compilation, soit dans une phase d'édition de liens.

b) Les passages de contrôle d'un module à un autre sont des appels ou des retours de procédures. Dans le cas de programmes où coexistent plusieurs processus, les exécutions des procédures d'un module doivent en général être synchronisées. Un autre schéma est alors envisageable : associer un processus à chaque module et réaliser les transferts de contrôle comme des appels de processus avec dépôt d'un message dans une boîte aux lettres associée au module. Lorsqu'on se propose d'écrire des systèmes d'exploitation, cette dernière solution peut sembler multiplier inutilement le nombre de processus (cf. chapitre 4). Elle a été cependant retenue dans le système MAS [Briat 76] pour diminuer le rôle des piles associées à chaque processus.

c) La notion de module présente une analogie avec celle de variable : ce sont deux dispositifs pouvant se trouver dans un nombre fini d'états; les transitions entre états sont provoquées par l'exécution d'instructions. Cependant, la façon dont on les emploie les différencie nettement :

- les modules sont des entités de construction de programme; il n'est pas possible de définir des opérateurs sur les modules. Les restrictions d'accès portent sur les modules.
- les valeurs des variables sont accessibles et modifiables sans restrictions ; on utilise des variables pour représenter l'état d'un module.

5.2.2.3. Exemple

Considérons une pile de nombres entiers. Le module associé comprend :
- la réservation de mémoire pour la pile et le pointeur de sommet de pile,
- une fonction de consultation de la valeur du sommet,
- deux procédures de changement d'état, par empilement ou désempilement.

5.2.3.2. Définition

Alors que dans un module, tout est défini dès l'écriture, on laissera figurer dans le texte d'un modèle de module un certain nombre d'inconnues dites méta-variables. Lors de la création d'un module engendré par le modèle, on spécifie les valeurs que doivent prendre les méta-variables pour cette réalisation particulière.

Si les métavariables peuvent a priori apparaître n'importe où dans le texte du modèle de module, on n'obtient des textes compréhensibles que si l'on se limite à des notations de type ou de valeur.

5.2.3.3. Exemple

On se propose maintenant de réaliser une pile analogue à celle du § 5.2.2.3., mais permettant de conserver des valeurs de type quelconque et en nombre maximal quelconque. La définition du module va donc faire intervenir deux méta-variables, l'une $\&t_1$ dénotant un type, l'autre *&nmax* une valeur entière. Il vient :

```
pattern pile (&t1 : type ; &nmax : integer) ;
var st : array [1 .. &nmax] of &t1 ;
    sp : integer := 0 ;
function sommet : &t1 ;
    if sp = 0 then sommet := nil else sommet := st [sp] ;
function empiler (x : &t1) : boolean ;
    if sp = &nmax then empiler := false
    else
       begin
       sp := sp+1 ; st [sp] := x ;
       empiler := true
       end ;
function désempiler : boolean ;
      {inchangée}
end
```

#### 5.2.3.4. Commentaires

De même qu'à partir d'un type T, on peut engendrer une classe de variables de type T, il est possible à partir d'un modèle de module d'engendrer toute une classe de modules. Il existe donc une certaine analogie entre les types et les modèles de modules. Lorsqu'on rassemble dans une définition de type ensemble de valeurs et opérations, ce que l'on appelle souvent "type abstrait", il y a même confusion des deux notions. Notre distinction a l'avantage de mettre l'accent sur les différences entre variables et modules.

## 5.3. DEFINITION DE STRUCTURES DE DONNEES DANS QUELQUES LANGAGES RECENTS

### 5.3.1. Introduction

Le modèle présenté en 5.2. est naturellement inspiré de SESAME. Nous commencerons donc ce paragraphe en précisant les possibilités effectives de PASCAL et de langages dérivés, SESAME, MODULA [Wirth 77] et EUCLID [Lampson 77]. Nous examinerons ensuite trois projets choisis parmi les multiples travaux consacrés aux définitions de types et aux langages d'écriture de systèmes : EL1 [Weghreit 74], CLU [Liskov 74a, 74b] et ALPHARD [Wulf 76]. Ces travaux ont été retenus essentiellement à cause de leur notoriété actuelle.

### 5.3.2. Structures de données dans PASCAL et SESAME

Nous commençons par rappeler quelques caractéristiques de PASCAL pour montrer ensuite les apports de SESAME. PASCAL offre à ses utilisateurs quelques types de base *(integer, boolean, real, char)* et la possibilité de définir de nouveaux types par trois constructions : le tableau, la structure et l'ensemble.

Etant donné un type, on peut définir un objet de ce type de deux façons :

- par une déclaration, l'objet est alors désigné par un identificateur,
- par une instruction de création dynamique, *new*, et l'objet est alors accessible par l'intermédiaire d'un pointeur.

Cependant, l'usage des types construits est restreint :

- il n'existe aucune notation de valeur de type construit, ni possibilité de définition d'opérateurs sur de nouveaux types. En pratique, on doit toujours effectuer opérations et affectations sur des composants élémentaires.
- les fonctions ne peuvent calculer que des résultats appartenant à un type de base ; en ce qui concerne les types construits, on doit utiliser des paramètres passés par référence.
- les types sont trop cloisonnés : seules quelques conversions *(integer-real)* sont prévues une fois pour toutes dans le compilateur; l'utilisation des tableaux est particulièrement restrictive : les bornes doivent être fixées à la compilation, même pour les paramètres formels des procédures.

Il existe une possibilité de paramétrage des programmes, la définition de constantes. Là encore, cette facilité présente des restrictions :

- on ne peut définir que des constantes appartenant à des types de base (ce qui est cohérent avec le reste du langage),
- on ne peut donner à une constante qu'une valeur immédiate, à l'exclusion de toute expression, calculée à la compilation, faisant intervenir des constantes préalablement définies.

Cette dernière restriction est particulièrement gênante en programmation de systèmes, où les tailles de certaines structures de données sont souvent calculées en fonction d'autres paramètres. En PASCAL, on est obligé de décrire comme indépendantes des constantes qui ne le sont pas, avec tous les risques d'erreur que celà présente lors des modifications.

Dans SESAME, nous avons décidé, pour faciliter les communications avec d'autres équipes, d'utiliser pour l'expression des algorithmes un langage très voisin de PASCAL. Dès lors, il n'était plus question de revenir sur les limites de la notion de type (des modifications "minimales" aux types de PASCAL ont été effectuées dans EUCLID [Lampson 77, Popek 77]. En ce qui concerne les types et les constantes, nous nous sommes contentés d'introduire :

- les expressions de constantes, pour lever la restriction la plus intolérable de PASCAL,
- les environnements (cf. chapitre 4), qui deviennent indispensables dès l'introduction des compilations séparées.

Par contre, nous avons fait porter notre effort sur l'introduction des modules et des modèles de modules (cf. § 5.2., 5.2.3.)

SESAME permet de conserver des modèles de modules :

- la construction d'un module particulier à partir d'un modèle de module (substitution de "paramètres" effectifs aux métavariables) est effectuée par le compilateur (dans sa procédure d'analyse lexicographique).
- aucune vérification syntaxique n'est effectuée sur les modèles de modules ; seule la correction des modules construits est vérifiée à la compilation.

Rappelons enfin (cf. Chapitre 4) que les modules d'un système sont compilés séparément, que les relations entre les modules sont exprimées dans un langage distinct du langage d'écriture, le langage de connexion, et qu'on vérifie à la connexion si les objets que l'on connecte sont compatibles.

En même temps que nous définissions SESAME, un autre prolongement de PASCAL était défini par Wirth : le langage MODULA. On trouve dans ce langage une nouvelle construction, le module, dont la définition est voisine de la nôtre. Plus précisément, on définit en tête de chaque module la liste des objets accessibles de l'extérieur. En conséquence :

- si seules des procédures sont accessibles, alors le module est identique à celui de SESAME, mais il n'existe aucun paramétrage,
- si l'on définit un type $T$ comme externe, on ne passe que le nom du type, et non la représentation : une variable de type $T$ ne pourra alors être manipulée que par des procédures déclarées dans le module où $T$ est défini. On se rapproche alors des types abstraits de CLU et ALPHARD (cf.§5.3.3.).

MODULA comprend de plus des éléments permettant de décrire et de synchroniser des processus parallèles : un processus est créé par appel asynchrone d'une procédure, la synchronisation est assurée par des moniteurs [Hoare 74].

Signalons enfin qu'un programme MODULA est compilé en une seule fois. Nous ne reviendrons pas ici sur les limites d'un tel choix (cf. chapitre 4).

### 5.3.3. Autres langages

a) Le langage EL1 [Weghreit 74] est un langage extensible dans lequel la définition de nouveaux types est particulièrement générale. En particulier, les limites de la notion de type de PASCAL, telles que nous les avons exposées plus haut sont levées dans EL1.

Quelques constructions se rapprochent de nos idées.

1) On dispose dans EL1 de deux traducteurs compatibles, un interpréteur et un compilateur. Le compilateur peut être appelé comme un sous-programme lors d'une interprétation. Supposons alors que l'on dispose d'un programme comportant des variables libres, il est possible de le compiler plusieurs fois en affectant à chaque compilation des valeurs particulières aux variables libres.

L'analogie avec les modèles de modules est forte. Cependant, un seul langage est utilisé. De plus, les phases de production de programme et d'exécution ne sont pas toujours distinctes dans EL1, alors que c'est un impératif lors de l'écriture des systèmes d'exploitation.

2) Une notion se rapproche des modules, la définition de type par son comportement ("programmer specified mode behavior") qui permet d'associer à un objet ayant une certaine représentation un ensemble d'opérations. Cependant, la nature de ces opérations est fixée une fois pour toutes dans le langage. Ont été retenues les conversions, l'affectation, l'accès à certains composants de l'objet, l'impression et la création. On peut de plus avoir accès à la représentation d'un objet, et utiliser alors des opérations non associées au type.

b) Les idées de départ de CLU [Liskov 74a, 74b] et ALPHARD [Wulf 76] sont assez voisines. La notion fondamentale est celle de type abstrait qui comprend la définition d'une classe d'objets et des opérations que l'on peut effectuer sur les objets de cette classe. Les types abstraits doivent vérifier trois propriétés :

- la définition d'un type doit comprendre la définition de toutes les opérations que l'on peut effectuer sur des objets de ce type,
- l'utilisateur d'un type abstrait n'a pas besoin de connaître la représentation d'un objet de ce type,
- l'utilisateur ne peut manipuler un objet qu'à travers les opérations définies dans le type, toute manipulation de la représentation en mémoire étant exclue.

Il est possible de créer des objets d'un type donné et de les faire désigner par des variables, la variable ne devant pas être considérée comme le contenant d'une valeur, mais plutôt comme un pointeur vers un objet. Ainsi, une affectation se traduira soit par la recopie d'un objet, soit par l'établissement d'un nouveau lien vers un objet existant, et non par la modification d'un objet. Une modification de la valeur d'un objet ne peut être faite que par l'intermédiaire d'une opération figurant dans le type abstrait de l'objet.

Le compilateur utilise la définition des types abstraits ("cluster" dans CLU, "form" dans ALPHARD) pour traduire les opérations effectuées sur les objets. A priori, on peut supposer que l'ensemble des opérations définies dans un type subsiste à l'exécution sous forme de sous-programme; ces sous-programmes peuvent opérer sur l'ensemble des objets du type, sauf dans certains cas de paramétrage. Ainsi, la structure des programmes offerte par les types abstraits ne subsiste pas à l'éxécution.

En ce qui concerne la compilation, on précise dans CLU que des compilations séparées sont possibles, à condition de fournir lors de chaque compilation la liste de tous les modules implantant les types abstraits utilisés. On trouve

dans [Jones 76] deux schémas possibles :

- la traditionnelle compilation unique,
- des compilations séparées avec un bibliothécaire pour conserver les objets et des contrôles de cohérence effectués soit à la compilation, soit à l'exécution.

On retrouve alors un terrain familier.

Exemple :

Cet exemple, tiré de [Liskov 74b] définit (partiellement) une classe d'ensembles d'éléments de type quelconque.

```
set = cluster[etype : type] is create, insert, remove, has, equal, copy ;
      rep = array of etype ;

      create = oper( ) returns cvt ;
           return rep$create ( );
      end create ;

      insert = oper (s : cvt, i: etype) ;
          if search(s,i) ≤ rep$high(s)  then return ;
          rep$extend(s,i);
          return ;
      end insert ;

      search = oper(s : rep, i : etype) returns int ;
           for j : int := rep$low(s) to rep$high(s) by 1 do
               if etype$equal(i,s[j])  then return j ;
           return rep$high(s) + 1 ;
      end search ;
       ⋮
end set
```

La première ligne définit le nom du type abstrait *set*, le paramètre de génération *etype* et les opérations exécutables *create*, *insert*, ...

On déclare ensuite, que l'ensemble est représenté par un tableau, puis on définit les opérations sur les objets du type abstrait. On peut remarquer que l'on peut définir pour celà des procédures locales au type abstrait (*search*). Pour

désigner une opération d'un type abstrait, on utilise une notation qualifiée. Ainsi :

*etype $ equal(i,s[j])*

est un appel à la procédure *equal* du type abstrait *etype*, avec les paramètres *i* et *s[j ]*.

### 5.3.4. Conclusion

La première conclusion que l'on peut tirer des paragraphes précédents, c'est que les idées essentielles en matière de structuration des données étaient déjà présentes dans SIMULA et EL1. L'apport effectif de CLU ou ALPHARD est finalement assez restreint. Quant à SESAME, il est bien clair que les modèles de modules sont eux aussi directement issus de SIMULA: son intérêt principal est de fournir un outil de construction progressive de systèmes, sans que l'on ait à compiler tout l'ensemble à chaque modification.

En second lieu, on observe deux tendances, selon que la notion de module subsiste ou non à l'exécution. Dans CLU ou ALPHARD, les programmes sont transformés par un compilateur en un programme dans lequel le cloisonnement dû aux types abstraits a disparu. Au contraire, dans SESAME et à un certain dégré dans MODULA, la notion de module subsiste à l'exécution et à la limite, on peut dire que les modules de SESAME sont avant tout une structure d'exécution, et que SESAME est concevable sans compilateur, avec des modules écrits en langage d'assemblage.

Remarque : La structure de moniteur [Brinch Hansen 73, Hoare 74] utilisée pour la synchronisation entre processus parallèles possède des propriétés analogues ; on peut inclure ce concept dans un langage de programmation, ou se contenter d'en respecter les règles.

La structuration à l'exécution offre des avantages :

1) Elle peut être associée à un système de protection qui vérifie que les règles d'accès aux objets sont bien vérifiées [Montuelle 77].

2) La construction progressive des systèmes par assemblage de modules en est facilitée.

3) On peut étendre la connexion statique des modules par une connexion dynamique au cours de l'exécution des systèmes produits.

Son inconvénient essentiel est une certaine perte d'efficacité; elle provient surtout des recopies occasionnées par le passage des paramètres par valeur.

## 5.4. PROGRAMMATION DE SYSTEMES ET PARTICULARITES DES CALCULATEURS

Lorsqu'on envisage de programmer des systèmes d'exploitation en langage évolué, deux objections sont fréquemment avancées : un système d'exploitation doit être efficace, il n'est pas aisé de décrire certaines particularités de la machine dans un langage distinct du langage d'assemblage. Nous allons donc donner quelques réflexions sur ces deux points, sans pour autant prétendre les épuiser.

### 5.4.1. Sur les questions d'efficacité

Pour la plupart des programmeurs, il est admis que programmer en langage évolué est synonyme d'inefficacité lors de l'exécution : un programme écrit à la main par un programmeur serait beaucoup plus rapide que celui produit par un compilateur. Comme un système d'exploitation est un programme exécuté très fréquemment, il est peu tolérable d'y perdre du temps et on ne pourrait donc le programmer qu'en langage machine.

Nous allons réfuter cet argument en deux temps :

1) en montrant ses limites,
2) en montrant que même s'il est vrai dans toute sa généralité, il ne fait aucun obstacle à la programmation en langage évolué de la quasi totalité d'un système.

Pour ne pas fausser la discussion, nous ne parlerons naturellement pas de la clarté, ni de la rapidité d'écriture et de mise au point, points pour lesquels la supériorité des langages de haut niveau est admise.

Tout d'abord, l'affirmation du début du paragraphe est beaucoup trop générale : l'efficacité du FORTRAN n'est sans doute pas la même selon que l'on programme une inversion de matrice ou une analyse syntaxique. De même, passer un paramètre par nom en ALGOL 60, ou par valeur-résultat en ALGOLW n'aura pas la même efficacité, surtout si l'accès à l'argument impose des calculs complexes. De même, si l'on étudie des programmes écrits à la main par des programmeurs, surtout après quelques modifications, il est bien rare de ne pas y relever des inefficacités.

Autrement dit, on ne peut parler de l'efficacité d'un langage en général : il faut considérer les éléments du langage, la classe des problèmes que l'on veut résoudre, et le programmeur qui les résoud. On trouve donc déjà un avan-

tage à passer par un compilateur : le programme produit a une qualité constante quels que soient le programmeur et le nombre de modifications effectuées : si l'on modifie un programme, une recompilation conserve la qualité du programme objet ; au contraire, une modification d'un programme en langage d'assemblage fait souvent disparaître les "optimisations" de la version initiale.

L'efficacité d'un langage va alors dépendre de l'adaptation des constructions qu'il contient aux problèmes à résoudre. Et il n'y a dès lors aucune raison de principe pour une inefficacité. On peut au contraire citer W.A. Wulf [Wulf 75]qui énonce : "en ce qui concerne l'efficacité, le seul critère que je reconnaisse pour un compilateur, c'est qu'il produise un code meilleur que celui écrit à la main par un bon programmeur".

Supposons maintenant que l'on dispose d'un langage inefficace. On sait [ Morrison 73] que les divers programmes constituant un système ne sont pas utilisés de manière uniforme : certaines séquences ne sont exécutées que quelques fois par jour (procédures d'initialisation, de redémarrage après erreur), d'autres plusieurs dizaines de fois par seconde (analyse d'interruption). Des mesures permettent de délimiter les séquences d'instructions très fréquemment parcourues. Comme ces séquences sont en général peu nombreuses, il est aisé de les réécrire en langage d'assemblage, en soignant leur efficacité.

### 5.4.2. Prise en compte des particularités des calculateurs

Certains modules des systèmes d'exploitation doivent opérer sur des données dont le format ou l'adresse sont imposées par le calculateur. On peut citer par exemple les mots d'état de programme, les programmes des canaux dont la structure dépend du calculateur, ou les zones servant à échanger les mots d'état en cas d'interruption, dont l'adresse est fixe.

De même, différents modes de contrôle d'exécution (appels au superviseur, déroutement, interruptions) doivent être pris en compte.

Lorsqu'on programme en langage d'assemblage, le traitement de ces particularités n'offre guère de difficultés, puisque toutes les caractéristiques de la machine sont utilisables. Il en va autrement en langage évolué, puisque la programmation est simplifiée par une certaine abstraction de la machine sur lequel le langage est implanté.

Différentes solutions ont été proposées pour sortir de ce cercle vicieux.

a) Inclusion dans le langage de caratéristiques du calculateur

L'idée de permettre au programmeur de "court-circuiter" les schémas de traduction d'un compilateur n'est pas nouvelle : bien des compilateurs FORTRAN autorisent l'insertion de lignes codées en langage machine. De même, le langage PL/1 permet le traitement de certains déroutements (ON-condition), c'est-à-dire que l'on peut appeler certaines procédures autrement que par un branchement avec sauvegarde de l'adresse de retour.

Dans cet ordre d'idées, quelques caractéristiques ont été ajoutées au langage MODULA [Wirth 76] pour le traitement des entrées-sorties sur PDP/11 : certains processus dits "device processes" peuvent se mettre en attente de la fin d'opérations d'entrée-sortie par une instruction particulière, "doio". Les "device processes" communiquent avec les autres processus par un moniteur spécialisé appelé "device module". De plus, on peut déclarer des variables et les associer aux registres tampons des périphériques ; on donne en général le type chaîne de bits à de telles variables (remarquons au passage que la chaîne de bits, de longueur égale à celle du mot de la machine, est la seule caractéristique de MODULA qui permette de programmer des traitements sur des structures de données de format imposé).

Cette approche ne nous paraît guère satisfaisante. Wirth donne deux indications pour qu'elle reste raisonnable :

- introduire le moins possible de caractéristiques de la machine,
- les exprimer de façon analogue aux concepts indépendants de la machine, lorsque c'est faisable et économiquement possible.

Il nous semble que nous serions obligés d'introduire, pour la programmation d'un système sur un gros calculateur, un grand nombre de "gadgets" assez disparates. D'autres solutions doivent donc être explorées.

b) Définition des données en deux phases : description logique et spécification de l'implantation.

Dans certains langages, la définition de structures de données est effectuée par deux séries de déclarations qui se complètent : dans une première déclaration, on précise la nature logique du type ou de la variable déclaré; dans

une seconde, on indique un schéma d'implantation pour l'élément. Lorsque le schéma d'implantation est omis, des règles systématiques sont imposées par le compilateur.

Cette idée existe à l'état embryonnaire dans PASCAL, où l'attribut *packed* préfixant une déclaration de type précise au compilateur que les composants d'un objet de ce type doivent être le plus possible tassés en mémoire (au prix d'un alourdissement des instructions d'accès). Elle a été reprise dans les langages LIS [Ichbiah 74a, 74b ]et LEST [Bétourné 76].

Par exemple, on peut décrire en LEST un mot d'état de programme par les instructions suivantes :

```
psw-scheme : scheme bit (32=(0,4);(4,1);(5,1);(6,1);(15;17));
psw : class record
      (cc : number in (0:15);
      mask,map,slave:flag ;
      address:location) with psw-scheme ;
```

La première instruction *(scheme)* définit un certain nombre de champs dans une chaîne de bits, chaque champ étant désigné par un couple $(o,\ell)$ $o$ étant le numéro du premier bit du champ et $\ell$ son nombre de bits.

Cette seconde façon de procéder nous parait très intéressante : on peut écrire un programme et le mettre au point sans se préoccuper des contraintes imposées sur les données. On donnera une description de l'implantation lorsque le programme sera au point. De plus, cette méthode peut être d'emploi assez général il est possible de composer aisément différentes versions d'un même programme, et de comparer leurs performances (vitesse d'exécution, taille de mémoire occupée).

c) Ecriture de procédures en langage machine

Enfin, même si l'on a la possibilité de compiler séparément les différents modules d'un programme, on peut se contenter d'écrire en langage-machine les procédures de traitement qui font intervenir les caractéristiques du calculateur. C'est pratiquement possible, compte-tenu du petit nombre des procédures à écrire. Ainsi, dans le système SOLO [Brinch Hansen 76], moins de 4 % du programme est écrit en langage machine, le reste en langage évolué (concurrent PASCAL). C'est également cette solution que nous avons pour l'instant retenue dans SESAME.

## 5.5. CONCLUSION

Ce chapitre sur les langages d'écriture de systèmes a été centré sur la définition des structures de données dans ces langages.

La tendance actuelle consiste à regrouper dans une même entité (type abstrait, module, modèle de module) la définition d'un ensemble de données et des opérations que l'on peut effectuer sur cet ensemble. Cette définition permet en général la création d'un nombre quelconque d'objets différenciés par les valeurs de certains paramètres.

La structuration fournie par ce regroupement peut ou non subsister à l'exécution. Dans les deux cas, elle induit des règles de visibilité des objets :

- les données ne sont modifiables qu'à travers les opérations définies dans la même entité; à la rigueur elles peuvent être accédées en lecture seule sans contrôle.
- les objets externes utilisés dans une entité doivent être énumérés en tête de l'entité.

En ce qui concerne plus spécifiquement SESAME :

- les modules subsistent à l'exécution; ils sont construits à partir de modèles de modules en donnant des valeurs à des meta-variables ;
- les modules sont compilés séparément ;
- les objets d'un module accessibles de l'extérieur, comme ceux définis dans d'autres modules et utilisés dans le module courant sont énumérés dans le module, certains noms restent libres après compilation :les noms des procédures fictives.

Enfin, les problèmes d'efficacité ou d'accès à toutes les possibilités d'une machine sont à notre avis souvent surestimés. Des solutions satisfaisantes existent, et en premier lieu le recours à quelques procédures écrites en langage d'assemblage.

Terminons ce chapitre sur une note un peu pessimiste. Notre conviction est que la plus grande partie des systèmes d'exploitation devrait être programmée en langage évolué. Nous devons bien constater, malgré le succès de projets prototypes, qu'il n'en est pas encore ainsi, et que l'inertie des programmeurs est considérable.

# CHAPITRE 6

# L'EXPRESSION DE LA SYNCHRONISATION ENTRE PROCESSUS PARALLELES

## 6.1. Le point sur la situation actuelle

### 6.1.1. Introduction

La synchronisation entre processus parallèles est sans aucun doute le domaine des systèmes d'exploitation qui a été le plus exploré. Depuis 1965 environ, de nombreux articles ont été publiés, que l'on peut répartir en trois classes :

- la définition de primitives élémentaires de synchronisation en considérant comme élémentaires des instructions que l'on peut câbler avec un coût raisonnable,
- la définition d'instructions de synchronisation adaptées aux langages évolués,
- la définition de méthodes de construction de programmes parallèles et la recherche de méthodes de preuves de correction [Howard 76, Owicki 76, Sintzoff 75].

A l'heure actuelle, on peut dire que la définition de primitives élémentaires n'est plus un sujet de recherche dans les systèmes non répartis. Il est admis que des solutions satisfaisantes existent. Le sémaphore et les primitives P et V [Dijkstra 67] servent de référence et sont couramment utilisés pour décrire des algorithmes.

Par contre, de nombreuses solutions sont proposées pour les langages évolués. Nous allons essayer ici d'évaluer ces différentes propositions. En conséquence, après un bref exposé des constructions étudiées, nous allons définir, puis appliquer une méthode d'évaluation.

### 6.1.2. Synchronisation dans les langages de haut niveau

On peut distinguer trois groupes d'instructions de synchronisation :

- les sections critiques conditionnelles [Hoare 72], dans lesquelles le contrôle du parallèlisme est explicité complètement,
- les moniteurs [Brinch Hansen 73, Hoare 74] dans lesquels l'exclusion mutuelle est implicite et les autres causes de synchronisation explicites,

- les expressions de chemin [ Campbell 74, Haberman 75, Flon 76] et les compteurs de synchronisation [Robert 77] dans lesquels on exprime la synchronisation comme un texte que l'on juxtapose à la compilation aux programmes proprement dits.

Rappelons brièvement les définitions de ces instructions.

a) Les sections critiques conditionnelles

On distingue classiquement trois formes de cette instruction.

α) La section critique inconditionnelle

*with* *V* *do* *I*

exprime que l'orsqu'une instruction *I* fait référence à une variable partagée *V*, alors l'exécution de *I* doit se faire dans une section critique associée à *V*.

β) La section critique conditionnelle

*with* *V* *when* *B* *do* *I*

signifie que l'instruction *I* doit être exécutée dans une section critique associée à la variable *V* lorsque la condition *B* est vérifiée.

L'expression booléenne *B* ne contient que des constantes ou des éléments de *V*.

En pratique, l'instruction est exécutée de la façon suivante par un processus *p*.

- *p* entre en section critique et évalue *B*.
- si *B* = *vrai*, *p* exécute *I* puis sort de la section critique.
- si *B* = *faux*, *p* sort de la section critique et se bloque ; il sera réactivé lorsqu'un autre processus quittera la section critique ; il reprendra alors l'exécution de l'instruction à son début.

γ) Lorsqu'un processus exécute

*with* *V* *do* *I* *await* *B*

il exécute *I* en exclusion mutuelle et de façon inconditionnelle, puis se bloque jusqu'à ce que l'expression booléenne *B* ait la valeur *vrai*.

b) Les moniteurs

Un moniteur est la réunion d'un ensemble de données (les données du moniteur) et de procédures (les procédures du moniteur). Les données d'un moniteur ne sont accessibles que par l'intermédiaire des procédures du moniteur; les procédures d'un moniteur ne peuvent désigner que leurs variables locales, leurs paramètres et bien sûr les données du moniteur.

On associe à un moniteur deux sortes de synchronisation :

- une synchronisation implicite : les procédures d'un même moniteur sont exécutées en exclusion mutuelle les unes par rapport aux autres ;
- une synchronisation explicite pour le blocage et l'activation des processus. On introduit pour cela des variables spéciales, dites conditions, sur lesquelles on peut effectuer deux opérations, *wait* et *signal*.

Si $c$ est une variable condition et $p$ le processus qui exécute une procédure d'un moniteur, alors :

- $c$.*wait* bloque le processus $p$, le range dans la file d'attente de $c$ et libère la section critique du moniteur.
- $c$.*signal* n'a pas d'effet si la file de $c$ est vide, sinon active un processus $q$ de cette file d'attente et lui donne le contrôle sans lâcher la section critique du moniteur (le processus $p$ qui exécute $c$.*signal* est bloqué jusqu'à ce que $q$ quitte le moniteur, par une sortie de procédure ou un nouveau blocage).

c) Les expressions de chemins

Considérons un système divisé en un ensemble de modules. Les expressions de chemin permettent de décrire, à côté des procédures du module , l'ensemble des séquences possibles pour l'exécution de ces procédures.

Les principales relations entre exécutions de procédure sont :

- la séquence notée $P;Q$

exprimant que la procédure *Q* doit être exécutée après la procédure *P*.
(N.B. : la synchronisation s'applique à l'ordre d'exécution des procédures ; *P* et *Q* peuvent être exécutés par des processus différents) ;

- la sélection notée *P + Q*
  exprimant que l'on peut exécuter soit *P*, soit *Q*.

A l'aide du parenthésage habituel, on peut composer séquence et sélection pour obtenir des expressions plus complexes.

Exemple : *P + (Q;R) + S* indique que l'on peut exécuter soit *P*, soit *Q* suivie de *R*, soit *S*.

Les délimiteurs *path* et *end* marquent le début et la fin d'une expression de chemin; ils indiquent de plus que le chemin peut être exécuté un nombre quelconque de fois. On peut exprimer cette dernière propriété pour une partie de chemin grâce à l'opérateur *.

Exemple : *P;(Q;R)*;S* autorise les séquences :
*P;S*
*P;Q;R;S*
*P;Q;R;Q;R;S*
etc...

Quelques extensions à ces mécanismes de base ont été proposées :

- les chemins conditionnels [Habermann 75] permettent de sélectionner un élément de chemin selon les valeurs courantes des variables du module.

Dans le chemin :

*<cond 1> : <elem 1>,..., <cond N> : <elem N>,{<elem N+1>}*

on sélectionne l'élément le plus à gauche dont la condition est vraie. L'élément N+1, facultatif, est sélectionné lorsque toutes les conditions sont fausses.

- les conditions bloquantes [Lucas 75] permettent à un processus de se bloquer soit avant, soit après une exécution de procédure en attendant qu'une certaine condition prenne la valeur vrai.

- les chemins avec comptage [Flon 76] permettent d'exprimer des relations entre les nombres d'exécutions des procédures.

  Exemple : Le chemin *path (acquire-release)*$^n$ *end* impose que la différence entre les nombres d'exécutions de *acquire* et *release* soit compris entre $0$ et et $n$.

Les chemins se rapprochent alors des compteurs de synchronisation ci-dessous.

d) Les compteurs de synchronisation

Dans cette approche, les procédures sont groupées dans des modules de traitement; à chaque module de traitement peut être associé un module de contrôle qui définit les règles de synchronisation associées à ce module.

Dans le module de contrôle figurent :

(i) un ensemble de queues, partition de l'ensemble des procédures, dans lesquelles seront chaînées des demandes d'exécution de procédures que l'on doit différer,

(ii) des compteurs d'état, associés à chaque procédure $P$ du module de traitement et définis comme suit :

  $\rho(P)$ nombre total de demandes d'exécution de $P$ depuis l'initialisation du module,

  $\alpha(P)$ nombre total d'autorisations d'exécutions de $P$ depuis l'initialisation du module,

  $\lambda(P)$ nombre total de terminaisons de $P$ depuis l'initialisation du module,

  $\nu(P) = \alpha(P) - \lambda(P)$, nombre d'exécutions courantes de $P$

  $\phi(P) = \rho(P) - \alpha(P)$, nombre de demandes d'exécutions reçues, mais non encore exécutées.

Le module de contrôle ne fait qu'enregistrer les demandes d'exécutions et les terminaisons de procédures. Par contre, il peut différer les autorisations d'exécutions. Pour cela, on associe à chaque procédure un ensemble de conditions qui doivent être vraies lorsque l'on autorise une exécution de cette procédure.

Exemple : La condition $\nu(P) + \nu(Q) = 0$ exprime que $P$ et $Q$ doivent être exécutées en section critique.

## 6.2. Propositions pour une évaluation des instructions de synchronisation

### 6.2.1. Généralités

Lorsqu'on se propose de comparer des instructions de synchronisation (voir par exemple [Brinch Hansen 72, Bekkers 74]), ou d'en justifier l'introduction de nouvelles, trois aspects doivent être discutés.

- La puissance de chaque construction, c'est-à-dire qu'on doit caractériser la classe de problèmes susceptibles d'être résolus à l'aide de cette construction;
- l'efficacité de sa mise en oeuvre sur des calculateurs;
- la clarté de l'expression et la commodité d'emploi de chaque construction.

Pour les deux premiers points, il est commode d'utiliser les sémaphores comme mécanisme de référence : on essaie de montrer que l'on peut définir des sémaphores à l'aide de la nouvelle instruction, puis que l'on peut en donner une implantation en termes de sémaphores. On a ainsi montré que les sémaphores et la nouvelle instruction ont des puissances équivalentes ; de plus, comme il est admis que l'on peut implanter efficacement des sémaphores, on peut étudier l'efficacité des constructions sur la traduction en termes de sémaphores.

L'estimation de la commodité d'emploi, de la clarté de l'expression, de la sécurité et des vérifications offertes est plus délicate. On essaie en général de résoudre certains problèmes de synchronisation et on compare leurs différentes solutions. La validité de la comparaison repose alors sur le choix des problèmes retenus. L'idéal serait bien sûr d'écrire un système complet avec chaque instruction, mais l'ampleur du travail à réaliser rend cette idée impraticable.

Nous allons donc retenir une solution moyenne : extraire des systèmes d'exploitation un certain nombre de problèmes de synchronisation réels. Ces problèmes réels sont en général plus simples à résoudre que des problèmes académiques (par exemple les lecteurs rédacteurs de [Courtois 71]) ; du point de vue du praticien, nous pensons que les résultats obtenus sont plus fidèles.

### 6.2.2. Les problèmes classiques de synchronisation

Nous distinguons trois problèmes fondamentaux de synchronisation : l'exclusion mutuelle, le couplage de deux processus selon le modèle du producteur et du consommateur et l'allocateur. Explicitons donc ces trois problèmes.

a) L'exclusion mutuelle [Dijkstra 67, Crocus 75]

Considérons deux processus cycliques. Pendant une phase particulière de leur exécution, dite section critique, chacun d'eux a besoin d'utiliser une ressource à un seul point d'accès.

Les vitesses relatives des processus sont quelconques; tout processus sort de section critique au bout d'un temps fini.

Une solution au problème doit vérifier quatre propriétés :

(i) à tout instant, un processus au plus peut se trouver en section critique;
(ii) si les deux processus sont bloqués en attente de la ressource critique, l'un d'eux doit pouvoir y entrer au bout d'un temps fini;
(iii) si un processus est bloqué lors d'une section critique, ce blocage ne doit pas empêcher l'entrée en section critique de l'autre processus;
(iv) aucun processus ne doit jouer de rôle privilégié.

Rappelons la solution classique de ce problème à l'aide de sémaphores.

Soit *mutex* un sémaphore de valeur initiale 1, le programme de chaque processus s'écrit :

```
P(mutex);
   :
{section critique}
   :
V(mutex);
```

Ce problème (et la solution ci-dessus) se généralise à un nombre quelconque de processus.

b) Le modèle du producteur consommateur [Dijkstra 67, Crocus 75]

Soit deux processus, appelés producteur et consommateur, qui se communiquent de l'information à travers une zone de mémoire.

Les messages transmis au consommateur par le producteur ont une taille constante et la zone de mémoire commune, ou tampon, a une capacité fixe de $n$ messages ($n > 0$). Les vitesses des deux processus sont quelconques.

La communication doit respecter les règles suivantes :

(i) le consommateur ne peut prélever un message que le producteur est en train de déposer ;

(ii) le producteur ne peut pas placer un message dans le tampon lorsque celui-ci est plein - il doit attendre ;

(iii) le consommateur doit prélever tout message une fois et une seule ;

(iv) si le producteur (respectivement le consommateur) est en attente parce que le tampon est plein (respectivement vide), il doit être réveillé dès que cette condition cesse d'être vraie.

Avec deux sémaphores *nvide* (valeur initiale $n$) et *nplein* (valeur initiale $0$), la solution s'écrit :

| Producteur | Consommateur |
|---|---|
| *while true do* | *while true do* |
| *begin* | *begin* |
| *Produire un message;* | *P(nplein);* |
| *P(nvide);* | *Prélever un message;* |
| *Déposer le message;* | *V(nvide);* |
| *V(nplein)* | *Consommer le message* |
| *end;* | *end;* |

c) L'allocateur de ressources

Considérons un ensemble de ressources banalisées de cardinal *nmax*. Chaque unité de ressources est à un point d'accès.

Des processus ont besoin pour s'exécuter de $x$ exemplaires de la ressource ($0 < x \leq nmax$). Lorsqu'un processus demande $x$ exemplaires, et que ce nombre $x$ est supérieur au nombre de ressources couramment disponibles, aucune allocation n'a lieu et le processus doit se bloquer. Il devra être réveillé lorsque $x$ ressources au moins seront devenues disponibles à la suite de libérations de ressources.

Nous admettons de plus que tout processus libère les ressources qui lui ont été allouées (et elles seules) au bout d'un temps fini. Lors d'une libération de ressources, la politique de satisfaction des processus en attente, s'il y en a, est quelconque. On pourra par exemple servir les processus dans l'ordre de leur arrivée, ou bien satisfaire, le plus possible de processus bloqués.

En pratique, nous aurons à écrire deux procédures :

*procédure allouer (x : integer; p : process-identifier) ;*
*{le processus p demande x unités de ressources}*

*procédure libérer (x : integer; p : process-identifier) ;*
*{le processus p libère x unités de ressources}*

et les instructions de blocage ou de réveil seront incluses dans ces procédures.

Nous ne nous intéressons ni à la façon dont on transmet à un processus l'identité des ressources allouées, ni à la prévention de l'interblocage (la politique adoptée permettrait d'introduire une prévention, par exemple par l'algorithme d'Habermann [Habermann 69]) ?

Donnons une solution de ce problème à l'aide de sémaphores.

L'état du système est représenté par une variable *nlibre* de valeur initiale *nmax*. Cette variable est augmentée (respectivement diminuée) à chaque allocation (respectivement libération) de la quantité de ressource allouée (respectivement libérée). Lorsque des processus sont bloqués en attente d'allocation, on conserve dans une liste leur nom et la quantité de ressources qu'ils demandent. On dispose de plus d'un sémaphore d'exclusion mutuelle *mutex* et d'un sémaphore privé par processus.

```
var nlibre : integer :=nmax;
    mutex : semaphore := 1;
    sempriv : array [processidentifier] of semaphore := 0;
 {procédures d'accès à la liste B des processus bloqués}
 procédure chaîner (p : process identifier; i : integer) ;
    {range dans B un doublet nom de processus p, demande i du processus p}
 procédure ôter (var p : processidentifier ; var i : integer) ;
    {ôte le premier doublet de B, restitue le nom p du processus extrait et sa
     demande i }
function tête : integer ;
    {fournit la valeur de la demande du premier processus de B; si la liste est
     vide, rend une valeur supérieure à nmax}
{procédure d'allocation-libération}
procédure allouer (x : integer ; p : processidentifier) ;
    begin  P(mutex);
    if  x > nlibre then chaîner (p,x)
    else begin nlibre := nlibre -x; V(sempriv[p]) end;
    V(mutex) ;
    P(sempriv[p])
    end ;
procédure libérer (x : integer; p : processidentifier) ;
    var q : processidentifier ;
        y : integer ;
    begin
    P(mutex)
    nlibre := nlibre + x ;
    {activation de processus bloqués, si c'est possible}
    while tête < nlibre do
          begin
          ôter (q,y) ;
          nlibre := nlibre - y ;
          V(sempriv[q])
          end ;
     V(mutex) ;
    end ;
```

Commentaires

1) Dans ce problème, la décision de blocage d'un processus dépend de l'état du système et d'un paramètre de la procédure *allouer*. Le blocage est donc détecté dans une section critique; naturellement, pour permettre à d'autres processus de s'exécuter, le processus ne se bloque effectivement qu'après être sorti de la section critique.

2) Si l'on s'intéresse à l'identité des ressources allouées, chaque processus $q$ activé doit entrer à nouveau en section critique pour prélever des unités de ressources effectivement disponibles.

   Il faut alors :

   - soit garantir qu'entre l'activation de $q$ et son entrée en section critique aucune allocation ne peut avoir lieu,
   - soit forcer le processus $p$ à reprendre la procédure *allouer* à son début (on court alors le risque de l'avoir réveillé inutilement).

3) Le problème académique des philosophes aux spaghetti [Dijkstra 67] est un cas particulier du problème de l'allocateur de ressources.

4) Dans le cas d'une ressource unique, on n'assure plus que le blocage ou le réveil des processus. Si l'on dispose d'un tel allocateur par processus, on retrouve la notion de sémaphore privé.

## 6.3. Solutions aux problèmes précédents

Nous donnons ici les différentes solutions sans commentaire. Une discussion d'ensemble est présentée en 6.4.

### 6.3.1. Exclusion mutuelle

Soit $V$ le descripteur de la ressource à point d'accès unique, $P_1$ et $P_2$ les procédures utilisant cette ressource dans chacun des processus.

a) Section critique conditionnelle

*with V do section critique ;*

b) Moniteurs

Il suffit de regrouper $P_1$ et $P_2$ dans un même moniteur pour qu'elles s'exécutent en exclusion mutuelle.

c) Expressions de chemin

*path* $P_1 + P_2$ *end ;*

d) Compteurs de synchronisation :

*condition* $P_i$ $(i=1,2)$ : $v(P_1) + v(P_2) = 0$

### 6.3.2. Producteur - Consommateur

Soit *plein* le nombre de cases occupées du tampon, produire et consommer les procédures qui assurent le dépôt et le prélèvement d'un message dans le tampon. Les programmes du producteur et du consommateur sont de la forme :

| Producteur | Consommateur |
|---|---|
| *while true do* | *while true do* |
| *begin* | *begin* |
| *préparer un message ;* | *consommateur ;* |
| *producteur* | *traitement du message* |
| *end ;* | *end ;* |

a) Section critique conditionnelle

```
procédure producteur ;
     with plein when plein < n do
        begin
        plein := plein + 1 ;
        produire
        end ;

procédure consommateur ;
     with plein when plein > 0 do
        begin
        plein := plein - 1 ;
        consommer
        end ;
```

b) Moniteur

```
tampon : monitor ;
   var plein : 0 ..  n ;
       nonvide, nonplein : condition ;
   procédure producteur ;
       begin
       if plein = n then nonplein . wait ;
       produire ;
       plein := plein + 1 ;
       nonvide . signal
       end ;
   procédure consommateur ;
       begin
       if plein = 0 then nonvide . wait ;
       consommer ;
       plein := plein - 1 ;
       non plein . signal
       end
   endmonitor ;
```

c) Expressions de chemin

Les trois articles [Campbell 74, Habermann 75, Flon 76] consacrés aux expressions de chemin développent des notations différentes. Nous donnons ici les solutions au problème tirées des deux derniers articles, la variante [Campbell 74] semblant abandonnée.

c1. expressions de chemin avec sélection conditionnelle

*path* [*plein = 0 : produire, plein = n : consommer, produire + consommer*] *end*

c2. expressions de chemin avec comptage

*path* $(produire - consommer)^n$ *end* ;

d) Compteurs de synchronisation

On exprime tout d'abord l'exclusion mutuelle à *produire* et *consommer*; de plus, le nombre de messages présents dans le tampon est égal à la différence entre le nombre d'autorisations d'exécution de *produire* et le nombre d'autorisations d'exécution de *consommer*.

D'où les deux invariants :

$$\nu(produire) + \nu(consommer) \leq 1$$
$$0 \leq \alpha(produire) - \alpha(consommer) \leq n$$

On en déduit les conditions d'autorisations des deux procédures :

- *condition(produire)* : $\nu(produire) + \nu(consommer) = 0$
  *et* $\alpha(produire) - \alpha(consommer) < n$
- *condition(consommer)* : $\nu(produire) + \nu(consommer) = 0$
  *et* $\alpha(produire) - \alpha(consommer) > 0$

Remarque : Les compteurs de synchronisation n'imposent pas une exclusion mutuelle à *produire* et *consommer*, contrairement aux moniteurs. On pourrait se contenter d'exprimer séparément les exclusions mutuelles entre producteurs et consommateurs.

### 6.3.3. Allocateur de ressources

L'état du système est représenté par une variable *nlibre* de valeur initiale *nmax*. Comme en 6.2.2., on pourra introduire une liste de doublets (nom de processus, demande) et les procédures d'accès *chaîner*, *ôter* et *tête*.

a) Sections critiques conditionnelles

On dispose d'une solution triviale :

```
a1    var nlibre : integer := nmax ;
      procédure allouer (x : integer) ;
          with nlibre when x < nlibre do
              nlibre := nlibre - x ;
      procédure libérer (x : integer) ;
          with nlibre do nlibre := nlibre + x ;
```

Mais l'ordre dans lequel les ressources sont allouées aux processus bloqués ne dépend que de l'implantation des sections critiques conditionnelles.

On peut alors préférer la solution suivante, où les processus sont activés nominalement.

```
a2  var nlibre : integer := max ;
      bloqué : array [processidentifier] of boolean := false ;
    {déclaration de procédure chaîner, ôter et tête comme en 6.2.2.}
    procédure allouer (x : integer ; p : processidentifier) ;
      begin
      with nlibre do
        if x > nlibre then
          begin
          chaîner (p,x) ;
          with bloqué do bloqué [p] := true
          end ;
      with bloqué when ¬ bloqué [p] do ;
      {blocage en attente de bloqué [p] = false}
      end ;
```

```
procédure libérer (x : integer ; p : processidentifier) ;
    var q : processidentifier ;
        y : integer ;
    begin
    with nlibre do
        begin
        nlibre := nlibre + x ;
        while tête ≤ nlibre do
            begin
            ôter (q,y) ;
            nlibre := nlibre - y ;
            with bloqué do bloqué [q] := false
            end
        end
    end ;
```

b) Moniteurs

On peut distinguer là aussi deux solutions, une solution immédiate (b1) valide si le nombre de processus bloqués est petit et une solution calquée sur celle donnée en 6.2.2.

```
b1  allocateur : monitor ;
    var nlibre : integer := nmax ;
        bloqué : condition ;
    ext procédure allouer (x : integer) ;
        begin
        while x > nlibre do
            begin
            bloqué . wait ; bloqué . signal
            end ;{réveil de tous les processus bloqués à chaque libération}
        nlibre := nlibre - x
        end ;
    ext procédure libérer (x : integer) ;
        begin
        nlibre := nlibre + x ;
        bloqué . signal
        end ;
    endmonitor ;
```

```
b2  allocateur : monitor ;
     var nlibre : integer := nmax ;
         bloqué : array [processidentifier] of condition ;
     {déclaration des procédures chaîner ôter et tête}

    ext procedure allouer (x : integer ; p : processidentifier) ;
        begin
        if x > nlibre then
           begin
           chaîner (p,x) ;
           bloqué [p] . wait
           end ;
        nlibre := nlibre - x
        end ;

    ext procédure libérer (x : integer ; p : processidentifier) ;
        var q : processidentifier ;
            y : integer ;
        begin
        nlibre := nlibre + x ;
        while tête ≤ nlibre do
              begin
              ôter (q,y) ;
              bloqué[q] . signal
              end
        end ;

    endmonitor ;
```

c) Expressions de chemin

Les expressions de chemins, comme les compteurs de synchronisation, ne permettent pas d'écrire une version intuitive de l'allocateur. Les solutions données sont calquées sur celle du 6.2.2.

```
type  sem =          {correspond à peu près à un sémaphore}
      begin procedure bloquer ; {vide} ;
            procedure lancer ; {vide} ;
            path lancer ; bloquer end
      end ;
var nlibre : integer := nmax ;
    sempriv : array [processidentifier] of sem ;
{déclarations des procédures chaîner, ôter et tête}
procédure demander (x : integer; p : processidentifier) ;
     begin
     if x > nlibre then chaîner (p,x)
     else begin nlibre := nlibre - x ;
                sempriv[p]. lancer
          end
    end ;
ext procédure allouer (x : integer ; p : processidentifier) ;
    begin
    demander(x,p) ;
    sempriv[p]. bloquer
    end ;
ext procedure libérer (x : integer; p : processidentifier) ;
    var q : processidentifier ;
        y : integer ;
    begin
    x := x * nlibre ;
    while tête ≤ nlibre do
          begin
          ôter (q,y)
          nlibre := nlibre - y
          sempriv[q]. lancer
          end
    end ;
path demander + libérer end ;
```

d) Compteurs de synchronisation

*type sem =*

*begin procedure bloquer ; {vide} ;*

*procedure lancer ; {vide}*

*condition (bloquer) : ν(bloquer) + ν(lancer) = 0 and α(lancer) - α(bloquer)>0;*

*condition (lancer): ν(bloquer) + ν(lancer) = 0*

*end ;*

*{les déclarations sont alors identiques aux précédentes}*

*condition(demander) : ν(demander) + ν(libérer) + φ(libérer) = 0 ;*

*condition(libérer) : ν(demander) + ν(libérer) = 0 ;*

*{de plus, les libérations sont prioritaires sur les demandes d'allocation}*

## 6.4. Evaluation

Nous allons tout d'abord faire le bilan de nos expériences sur la commodité des différentes instructions. Des considérations d'efficacité donneront des éléments supplémentaires de choix.

### 6.4.1. Souplesse des différentes instructions

Les problèmes de l'exclusion mutuelle et du producteur - consommateur sont bien résolus avec les quatre types d'instructions. Par contre, si le problème de l'allocateur est convenablement traité avec les sections critiques conditionnelles ou les moniteurs, les solutions apportées par les expressions de chemin et les compteurs de synchronisation sont beaucoup plus lourdes.

En effet, l'idée d'exprimer la synchronisation à côté des algorithmes et au niveau des procédures est séduisante à une condition : le découpage d'un problème en procédures doit être guidé par des raisons logiques ; il ne doit pas être modifié par des considérations de synchronisation. Et nous constatons :

1) La procédure allouer doit être coupée en deux parties, l'une (demander) exécutée en section critique, l'autre se réduisant à un appel à demander et à un blocage éventuel.

2) Pour bloquer un processus particulier, on lui fait exécuter une procédure vide; cette exécution ne sera autorisée que lorsqu'un autre processus aura exécuté une seconde procédure tout aussi vide que la précédente.

Plus généralement, les expressions de chemin et les compteurs de synchronisation sont mal adaptés aux problèmes dans lesquels les conditions de synchronisation font intervenir des variables d'état des modules et des paramètres des procédures. Lorsque l'on compare expressions de chemins et compteurs de synchronisation, on constate qu'on introduit moins de procédures vides avec les compteurs qu'avec les expressions de chemin, du moins dans la version initiale [Campbell 74]. Mais les versions [Habermann 75] et [Flon 76] paraissent équivalentes aux compteurs.

Personnellement, nous avons cependant une légère préférence pour les compteurs, et ceci pour deux raisons :

1) leur apprentissage est plus simple que celui des expressions de chemin,

2) les promoteurs des expressions de chemin ont tendance à introduire un nouveau "gadget" pour chaque nouveau problème. On est ainsi passé des accolades permettant de résoudre les problèmes de lecteurs-rédacteurs aux expressions conditionnelles, puis aux comptages; pour quand un allocateur incorporé ?

En ce qui concerne les moniteurs, on leur reproche souvent de ne pas permettre la résolution de problèmes qui ne se ramènent pas à des exclusions mutuelles. Considérons donc un problème représentatif de cette classe, celui des lecteurs-rédacteurs [Courtois 71]. Un moniteur unique ne permet évidemment pas de le résoudre ; on doit distinguer deux éléments, le module d'accès au fichier, avec les procédures lire et écrire, et un moniteur contrôlant les droits d'accès au fichier.

Le corps de chaque procédure du module fichier est précédé et suivi par un appel à une procédure du moniteur assurant les changements d'état des variables décrivant le fichier et les blocages éventuels des processus.

Nous donnons ci-dessous une solution dans le cas où les lecteurs ont priorité sur les rédacteurs.

```
module fichier ;
     {déclaration du fichier}
     ext procedure lire ;
         begin
         accès.demlect ;
         {lecture effective}
         accès.finlect
         end ;
     ext procedure écrire ;
         begin
         accès.demecr ;
         {écriture effective}
         accès.finecr
         end ;
     endmodule
accès : monitor ;
  var nl,nr : integer := 0 ; {comptent les lecteurs et les rédacteurs}
      ecr : boolean := false ;{true  si une écriture est en cours}
      lire, écrire : condition ;
  ext procédure demlect ;
      begin nl := nl + 1 ;
      if ecr then begin
                  lire.wait ;
                  lire.signal {réveille le lecteur suivant}
                  end
      end ;
   ext procédure finlect ;
       begin nl := nl - 1 ;
       if (nl = 0) and (nr>0) then begin ecr := true; écrire.signal end
       end ;
   ext procédure demecr ;
       begin nr := nr + 1 ;
       if ecr or (nl>0) then écrire.wait
       else ecr := true
       end ;
```

```
ext procédure finecr
    begin nr := nr - 1 ;
    if nl >0 then begin ecr := false, lire.signal end
    else if nr > 0 then écrire.signal
    else ecr := false.
    end ;
endmonitor ;
```

Cette solution est bien sûr un peu longue à écrire. Cependant, on peut facilement la modifier pour implanter une stratégie quelconque de partage du fichier.

Remarque : La séparation entre le module fichier et le moniteur de gestion des accès se retrouve lorsqu'on utilise d'autres instructions de synchronisation (voir par exemple [Bekkers 77]).

### 6.4.2. Efficacité de la mise en oeuvre

Malgré un recul de quatre à cinq ans, on dispose finalement d'assez peu de renseignements sur l'efficacité des instructions. Signalons cependant quelques résultats.

1) Une traduction en termes de sémaphores a été donnée pour les sections critiques conditionnelles [Crocus 75], les moniteurs [Hoare 74] et les expressions de chemin [Campbell 74].

2) L'implantation des sections critiques conditionnelles implique une certaine forme d'attente active : lorsqu'une condition ne peut être évaluée que dans le processus où elle figure, alors on doit réveiller à chaque sortie de section critique tous les processus bloqués. Il est cependant possible [Lucas 74] d'évaluer les conditions dans le processus qui quitte la section critique. On explore toujours la liste des processus bloqués, mais on n'active un processus qu'à coup sûr.

3) Hoare [Hoare 73] a proposé diverses implantations des moniteurs. Même la traduction la plus générale est proche de ce que produirait à la main un bon programmeur. De plus, les moniteurs ont été intégrés à un langage de programmation, Concurrent PASCAL, qui a été utilisé pour construire un système complet [Brinch Hansen 76].

4) Lorsqu'on traduit des expressions de chemins par des prologues et des épilogues de procédures utilisant des sémaphores [Campbell 74], alors le nombre de sémaphores introduits est trop grand : on ne tient pas compte du fait qu'à l'intérieur de chaque processus, les procédures sont appelées dans un ordre bien déterminé.

5) Dans les compteurs de synchronisation, il est facile de traduire une condition d'autorisation par un prologue de procédure. Par contre, à chaque sortie de procédure, il faut rechercher parmi les processus bloqués ceux dont l'état a changé (ou les réveiller systématiquement pour qu'ils réévaluent eux-mêmes leurs conditions).

### 6.4.3. Conclusion

Supposons que l'on veuille introduire des primitives de synchronisation dans un langage de haut niveau, quelle instruction va-t-on choisir ?

On doit d'abord opter pour un schéma classique, où les attentes et les réveils s'expriment explicitement (moniteurs) ou pour un ordonnancement des procédures exécutées.

L'idée d'exprimer la synchronisation au niveau des procédures semble intéressante, car elle favorise la modularité : on peut même imaginer qu'à un algorithme unique, on fasse correspondre différentes synchronisations selon le contexte mis en jeu. La réalité est moins séduisante : des nécessités de synchronisation conduisent à redéfinir le découpage d'un algorithme en procédures, voire à introduire des procédures vides. Et on retrouve ainsi les attentes explicites dont on avait voulu se passer.

Si l'on retient un schéma classique, on optera pour les moniteurs lorsque les considérations d'efficacité sont prépondérantes. S'ils n'apportent ni la puissance, ni l'élégance des sections critiques conditionnelles, tous les problèmes que nous avons cherché à résoudre ont pu l'être dans de bonnes conditions (solution trouvée rapidement, et d'une efficacité correcte). C'est d'ailleurs cette solution de sécurité que nous avons retenue dans SESAME, après plusieurs tentatives d'utilisation des expressions de chemin.

Par contre, dans le cadre d'un système à processeurs suffisamment nombreux pour que certaines formes d'attente active soient tolérables, nous nous orienterions sans doute vers une implantation des sections critiques conditionnelles.

Une voie est peut-être à explorer : essayer de faire coexister plusieurs instructions dans un même système. On pourrait ainsi utiliser les compteurs là où ils sont bien adaptés et réserver les moniteurs aux cas où la synchronisation met en jeu les paramètres des procédures.

# CHAPITRE 7

# C O N C L U S I O N

Nous nous étions proposé dans ce travail de traiter de quelques problèmes de programmation des systèmes d'exploitation : les méthodes de décomposition de systèmes et leur construction par assemblage de parties, les langages d'écriture de systèmes et l'expression du parallélisme. Il nous faut donc maintenant faire le bilan de notre travail, le situer par rapport à d'autres travaux sur les systèmes d'exploitation et envisager quelques prolongements.

## 7.1. RECAPITULATION DES RESULTATS OBTENUS

### 7.1.1. Intégration des méthodes de conception et d'écriture aux outils de programmation

En premier lieu, notre expérience nous prouve que des méthodes de conception et d'écriture de système, quelles que soient leur qualité, sont insuffisantes : l'expérience d'ESOPE montre que les programmeurs ont tendance à violer les règles de programmation, même si ces règles ont été fixées d'un commun accord. En conséquence, les méthodes de conception et d'écriture doivent être intégrées aux outils mis à la disposition des programmeurs, qui se trouvent dès lors obligés de les suivre.

Remarques :

1) On procède de cette façon en ce qui concerne la visibilité des objets dans les langages à structure de bloc : on **ne** peut adresser dans un bloc une variable déclarée dans un bloc disjoint; il est donc superflu de protéger cette variable.

2) La seule autre solution viable (?) consiste à imposer le respect des règles par des moyens coercitifs(on laisse les programmeurs dans l'ignorance de tout ce qui n'est pas nécessaire à leur travail - choix globaux de réalisation, caractéristiques d'un module particulier ; le respect des règles est vérifié par le chef de projet, ...)

### 7.1.2. Modularité

Le module, ensemble de déclarations de variables et de procédures de manipulation de ces variables se révèle une entité commode de décomposition, et surtout de construction de systèmes d'exploitation. Le cloisonnement entre les modules doit être vérifié soit statiquement (par le compilateur ou l'éditeur de liens), soit dynamiquement. En particulier, les paramètres des procédures externes à un module doivent être passés par valeur; lorsqu'on autorise un accès direct à des variables d'un module, cet accès doit être limité à des lectures.

Un aspect encore peu exploré de la programmation modulaire est la composition de programmes par assemblage de modules préexistants. Cette nouvelle méthode impose de disposer d'un outil de conservation et de recherche de modules (le bibliothécaire dans SESAME). De plus, pour que la réutilisation de modules soit effective, il importe de laisser à l'utilisateur le choix de certains paramètres. Ce problème de paramétrage est résolu dans SESAME par les métavariables (types des données et tailles des tableaux) et les procédures fictives (établissement des connexions entre modules). Mais à notre avis, le problème de la réutilisation des modules est un problème de conception : lorsqu'on écrit un module, on doit essayer de se dégager de son problème immédiat pour composer un module d'emploi général, au prix peut-être d'une perte d'efficacité. Et on ne dispose actuellement d'aucune règle pour effectuer cette généralisation.

Plaçons-nous maintenant dans le contexte des systèmes d'exploitation. Nous verrons en 7.2. que le fait de conserver à l'exécution la structure modulaire offre des avantages en ce qui concerne la protection. Il faut alors gérer deux structures d'exécution différentes, les processus et les modules; une façon harmonieuse de le faire consiste à laisser à chaque notion une seule fonction (un processus est une entité introduite pour des raisons de parallélisme physique ou logique, un module une entité de décomposition et de protection) et à définir les processus de façon liée à la structure modulaire (soit appel asynchrone à une procédure, soit *parbegin parend)*.

### 7.1.3. Langages d'écriture de systèmes

Compte-tenu de la clarté et de la rapidité d'écriture et de mise au point qu'ils apportent, les langages évolués devraient être le plus souvent utilisés pour

l'écriture des systèmes d'exploitation. Dans la conception d'un langage en vue de l'écriture d'un système d'exploitation, on veillera particulièrement à éviter les constructions par trop inefficaces. Par contre, une gestion à l'exécution ne doit pas être rejetée systématiquement, elle doit simplment être compatible avec tous les systèmes que l'on envisage de produire.

Dès que l'on envisage la construction de gros systèmes, des possibilités de compilation séparée, jointes à des vérifications de cohérence à la liaison, nous semblent indispensables. Un langage de connexion, distinct du langage d'écriture des modules, permet d'énumérer les modules constituant un système et de spécifier les valeurs des paramètres et les liaisons entre modules.

On doit également prévoir que certaines parties des systèmes sont très dépendantes de particularités de la machine. On évitera d'abâtardir le langage utilisé en y intégrant des propriétés de la machine et on s'orientera plutôt vers une programmation à deux niveaux distinguant l'algorithme des traits d'implantation. Plus simplement, on pourra souvent se contenter de quelques sous-programmes écrits en langage machine.

## 7.2. SECURITE DES SYSTEMES - PROTECTION A LA COMPILATION OU A L'EXECUTION

Tous les travaux récents sur les systèmes d'exploitation tendent d'une part à faciliter leur conception et leur écriture, d'autre part à améliorer leur qualité de fonctionnement en limitant sinon les erreurs, du moins les conséquences de ces erreurs. Plus précisément, on peut distinguer deux approches :

- l'une fondée sur la vérification (a priori ou à la compilation) de la correction du système d'exploitation,
- l'autre fondée sur l'addition aux ordinateurs de dispositifs câblés de protection.

Notre travail se situe plutôt dans le premier de ces courants.

Si chaque approche possède ses inconditionnels, un point de vue plus nuancé s'impose à notre avis. Tout d'abord, les techniques de preuves de programme sont à l'heure actuelle inapplicables pour prouver la correction d'un gros programme, système d'exploitation ou compilateur. En ce qui concerne l'accès aux objets, on doit donc se reposer soit sur les règles de portée imposées par un compilateur,

soit sur des contrôles dynamiques soit sur les deux. Les contrôles effectués par un compilateur, qui n'augmentent pas le temps d'exécution des programmes, ni ne demandent de dispositifs câblés appropriés sont efficaces à deux conditions :

- le compilateur doit être au point,
- l'ordinateur doit fonctionner conformément aux spécifications.

Si la première de ces conditions peut être réalisée (avec beaucoup de difficultés pendant la phase de mise au point du compilateur), on ne peut envisager actuellement de machine totalement dépourvue de panne. En conséquence, les deux approches doivent être utilisées conjointement : on effectue statiquement un maximum de vérifications et des dispositifs de protection assurent une détection et empêchent la propagation des erreurs résiduelles.

Lorsque la structure modulaire est conservée à l'exécution, c'est elle qui va diriger l'utilisation des mécanismes de protection : les objets accessibles au cours de l'exécution d'un module sont bien définis, l'environnement accessible ne change que lors des transferts de contrôle d'un module à un autre et les dispositifs de protection peuvent être utilisés efficacement.

## 7.3. PROBLEMES OUVERTS ET PROLONGEMENTS

Restent donc à examiner les prolongements possibles de travail.

Tout d'abord, certains problèmes actuels n'ont toujours pas de solution satisfaisante : nous pensons principalement à l'expression de la synchronisation et aux traitements des erreurs.

Quand nous avons abordé l'étude de la synchronisation dans le contexte des programmes modulaires, nous avions l'intention de rechercher des instructions permettant d'exprimer les règles de synchronisation à côté des programmes des modules ; certaines instructions de synchronisation auraient alors pu être changées selon le contexte d'utilisation du module, sans que l'on ait à modifier le module lui-même. L'objectif est toujours valable, mais aucune solution satisfaisante ne s'est encore dégagée.

En ce qui concerne la détection et le traitement des erreurs, les "solutions" actuelles sont disparates : certaines détections sont effectuées par câblages, d'autres par des tests programmés implicites ou définis par le programmeur. Certaines actions de reprise sont imposés, d'autres peuvent être modifiées dynamiquement, etc...

Dans un premier temps, il importe de dégager un schéma uniforme de détection d'erreurs, et d'activation de procédures de traitement; ce schéma pourra être alors intégré aux outils d'écriture disponibles. Une réflexion plus fondamentale concerne la nature des actions à effectuer lorsqu'une erreur a été détectée; il serait en tout cas intéressant de déterminer quelles sont les parties du système qui peuvent continuer à fonctionner, quels sont les modules à remplacer. Or, la simple localisation du module dans lequel l'erreur s'est produite est déjà difficile.

Ces deux problèmes, synchronisation et traitement d'erreur, sont certes fondamentaux, mais la découverte de solutions admises par tout le monde est probablement assez éloignée. Des prolongements plus immédiats de notre travail sont envisageables.

En premier lieu, il serait intéressant de définir une architecture de machine adaptée à la programmation modulaire, de même que l'on a défini des architectures adaptées à la protection; ces deux classes de machine ne sont d'ailleurs probablement pas fondamentalement différentes.

De même, un des problèmes du découpage en modules consiste à limiter les transferts (aussi bien de contrôle que d'informations) entre modules. Comme ces transferts sont peu coûteux dans un système centralisé, on n'y prête pas toujours l'attention souhaitée. Il serait alors intéressant d'étudier une architecture dans laquelle chaque module serait réalisé par un microordinateur, et les communications entre modules par des lignes de communication. Dans le même ordre d'idées, il serait intéressant d'étudier, sous l'angle de la programmation modulaire, des systèmes multicalculateurs répartis sur plusieurs sites éloignés (la répartition n'est plus un choix de réutilisation comme ci-dessus mais une contrainte d'utilisation). Le problème est de savoir si l'on peut se contenter de modules locaux à chaque site, ou si l'on a besoin de faire intervenir des entités englobant plusieurs sites.

Quoi qu'il en soit, nous pensons avoir montré dans ce travail que la programmation des systèmes d'exploitation ne doit pas être considérée comme une activité de spécialistes. Au contraire, pourvu que l'on dispose d'outils convenables, il devrait être possible de construire des systèmes d'exploitation adaptés aux besoins de chaque installation informatique.

# B I B L I O G R A P H I E  G E N E R A L E


Armstrong 73 — ARMSTRONG R.M.
*Modular programming in COBOL*, Wiley (1973)

Baudet 72 — BAUDET G., FERRIE J., KAISER C., MOSSIERE J.
Entrées-sorties dans un système à mémoire virtuelle, *Congrès AFCET*, Grenoble (1972)

Bekkers 74 — BEKKERS Y.
*A comparison of two high level synchronizing concepts*, The Queen's University of Belfast (1974)

Bekkers 77 — BEKKERS Y., BRIAT J., VERJUS J.P.
Construction of a synchronization scheme by independent definition of parallelism, *Proc. IFIP TC2 Working conference on constructing quality software*, Novosibirsk (1977)

Bellino 73 — BELLINO J.
*Mécanismes de base dans les systèmes superviseurs : conception et réalisation d'un système à accès multiples*, Thèse, Grenoble (1973)

Bétourné 70a — BETOURNE C., BOULENGER J., FERRIE J., KAISER C., KRAKOWIAK S., MOSSIERE J.
Présentation générale du système ESOPE. Espace virtuel dans le système ESOPE. Allocation de ressources dans le système ESOPE *Congrès AFCET*, Paris (1970)

Bétourné 70b — BETOURNE C., BOULENGER J., FERRIE J., KAISER C., KRAKOWIAK S., MOSSIERE J.
Process management and resource sharing in the multiaccess system ESOPE, *CACM* 13, 12 (1970)

Bétourné 71 — BETOURNE C., FERRIE J., KAISER C., KRAKOWIAK S., MOSSIERE J.
System design and implementation using parallel processes, *Proc. IFIP Congress*, Ljubljana (1971)

Bétourné 72 — BETOURNE C., KRAKOWIAK S.
Simulation de l'allocation de ressources dans un système conversationnel à mémoire virtuelle paginée, *Congrès AFCET*, Grenoble (1972)

Bétourné 75 — BETOURNE C., KRAKOWIAK S.
Mesures sur un système conversationnel, *RAIRO, AFCET* (1975)

Bétourné 76 — BETOURNE C., FERAUD L., JOULIA J., RIGAUD J.M.
Le langage d'écriture de systèmes "LEST", *Congrès AFCET*, Paris (1976)

Boulenger 74a — BOULENGER J.
Ecriture de systèmes d'exploitation à l'aide d'un langage de type PL 360, Rapport de recherche n° 48, IRIA (1974)

Boulenger 74b — BOULENGER J., KRONENTAL M.
An experience in systems programming using a PL 360 like language, *Proc. IFIP WG 2.4 Meeting*, La Grande Motte (1974)

Briat 76 — BRIAT J., ROUSSET de PINA X., VERJUS J.P.
Le projet OURS : ses principes, *Congrès AFCET*, Paris (1976)

Brinch Hansen 72a — BRINCH HANSEN P.
Structured multiprogramming, *CACM*, 15,7 (1972)

Brinch Hansen 72b — BRINCH HANSEN P.
A comparison of two synchronizing concepts, *Acta Informatica*, 1,3 (1972)

Brinch Hansen 73 — BRINCH HANSEN P.
Concurrent programming concepts, *Computing surveys*, 5,4 (1973)

Brinch Hansen 74a — BRINCH HANSEN P.
*Concurrent PASCAL, a programming language for operating system design*, California Institute of Technology, TR n° 10 (1974)

Brinch Hansen 74b BRINCH HANSEN P.
*Deamy : a structured operating system*, California Institute of Technology, (1974)

Brinch Hansen 75 BRINCH HANSEN P.
The programming language concurrent PASCAL, *Int. summer school on language hierarchies and interfaces*, Marktoberdorf (1975)

Brinch Hansen 76 BRINCH HANSEN P.
The SOLO operating system, *Software-Practice and Experience*, vol. 6, pp. 139-205, (1976)

Burroughs 64 BURROUGHS Corp.
*Burroughs B5500 information processing systems reference manual* (1964)

Buxton 70 BUXTON J., RANDELL B. (Eds)
*Software engineering techniques* (Rome 1969), NATO Science committee (1970)

Campbell 74 CAMPBELL R.H., HABERMANN A.N.
The specification of process synchronization by path expressions *Coll. sur les aspects théoriques et pratiques des systèmes d'exploitation*, IRIA, Paris (1974)

Cheval 76a CHEVAL J.L., CRISTIAN F., KRAKOWIAK S., LUCAS Ma., MONTUELLE J., MOSSIERE J.
*Conception modulaire des systèmes d'exploitation*, Rapport de Recherche n° 32, IMAG, Grenoble (1976)

Cheval 76b CHEVAL J.L., CRISTIAN F., KRAKOWIAK S., LUCAS Ma., MONTUELLE J., MOSSIERE J.
Un système d'aide à l'écriture des systèmes d'exploitation, *Congrès AFCET*, Paris (1976)

Cheval 77 — CHEVAL J.L., CRISTIAN F., KRAKOWIAK S., MONTUELLE J., MOSSIERE J. An experiment in modular program design, *Proc. IFIP Congress*, Toronto (1977)

Clark 71 — CLARK D., GRAHAM R., SALTZER J., SCHROEDER M. *The classroom information and computing service*, MAC TR 80, MIT (1971)

Corbato 69 — CORBATO F.J., PL/1 as a tool for systems programming, *Datamation* (1969)

Corwin 72 — CORWIN P., WULF W.A. *A software laboratory*, Carnegie-Mellon University (1972)

Courtois 71 — COURTOIS P.J., HEYMANS F., PARNAS D.L. Concurrent control with "Readers" and "Writers", *CACM* 14,10 (1971)

Cristian 76 — CRISTIAN F., GAUDUEL F. *Réalisation d'un bibliothécaire pour le projet SESAME*, Projet de 3ème année ENSIMAG, Grenoble (1976)

Cristian 77 — CRISTIAN F. A case study in modular design, *Proc. Int. Computing symposium*, Liège (1977)

Crocus 75 — CROCUS *Systèmes d'exploitation des ordinateurs. Principes de conception* Dunod (1975)

Dahl 70 — DAHL O.J., MYRHAUG B., NYGAARD K. *The SIMULA 67 common base language*, Norwegian Computing center (1970)

Dahl 72 — DAHL O.J., DIJKSTRA E.W., HOARE C.A.R. *Structured programming*, Academic Press (1972)

De Courcel 74 — DE COURCEL P., GUILLORY A., MONTEL D., MONTUELLE J. *Réalisation d'un mini-système d'exploitation*, Projet de 3ème année ENSIMAG, Grenoble (1974)

De Remer 75 — DE REMER F., KRON H.
Programming in the large vs. programming in the small, Proc. Int. Conference on Reliable Software, *SIGPLAN Notices* 10,6 (1975)

Derniame 74 — DERNIAME J.C.
*Le projet CIVA, un système de programmation modulaire,* Thèse, Nancy (1974)

Dijkstra 67 — DIJKSTRA E.W.
Cooperating sequential processes, in *Programming languages,* F. Genuys Ed., Academic Press (1967)

Dijkstra 68 — DIJKSTRA E.W.
The structure of the THE - Multiprogramming system, *CACM* 11,5 (1968)

Dijkstra 71 — DIJKSTRA E.W.
Hierarchical ordering of sequential processes, *Acta Informatica* 1,2 (1971)

Dijkstra 72 — DIJKSTRA E.W.
Notes on structured programming in Dahl 71

Dijkstra 76 — DIJKSTRA E.W.
*A discipline of programming,* Prentice Hall (1976)

Ferrié 71 — FERRIE J.
*La gestion des processus dans un système à partage de ressources,* Thèse de Docteur-Ingénieur, Toulouse (1971)

Ferrié 72 — FERRIE J., MOSSIERE J.
*ESOPE : gestion des processus et partage de ressources,* cahier de l'IRIA n° 8 (1972)

Flon 76 — FLON L., HABERMANN A.N.
Towards the construction of verifiable software systems, *Proc. ACM symposium on data (abstraction, definition and structure)* (1976)

Habermann 69 — HABERMANN A.N.
Prevention of system deadlocks, *CACM* 12,7 (1969)

Habermann 75 — HABERMANN A.N.
*Path expressions*, Carnegie Mellon University (1975)

Henderson 76 — HENDERSON P.
*Some thoughts on program structuring*, Computing Laboratory, Univ. of Newcastle upon Tyne, (1976)

Hérodin 77 — HERODIN J.M., GUYOT F.
*MORSE : réalisation modulaire d'un système d'exploitation*, Projet de 3ème année ENSIMAG, Grenoble (1977)

Hoare 72a — HOARE C.A.R.
Operating systems : their purpose, objectives, functions and scope, in *Operating systems techniques*, Hoare & Perrott eds, Academic Press (1972)

Hoare 72b — HOARE C.A.R.
Towards a theory of parallel programming in *Operating systems techniques*, Hoare & Perrott eds, Academic Press (1972)

Hoare 74 — HOARE C.A.R.
Monitors : an operating system structuring concept, *CACM* 17,10 (1974)

Howard 76 — HOWARD J.H.
Proving monitors, *CACM* 19,5 (1976)

Ichbiah 74a — ICHBIAH J.D., RISSEN J.P., HELIARD J.C.
The two level approach to data independent programming in the LIS system implementation language, *Machine oriented higher level languages*, Van der Poel & Maarsen eds, North Holland (1974)

Ichbiah 74b — ICHBIAH J.D., RISSEN J.P., HELIARD J.C., COUSOT P.
*The system implementation lauguage LIS*, Technical report 4549E/En, CII (1974)

Jackson 75 — JACKSON M.A.
*Principles of programm design*, Academic Press (1975)

Jensen 74 — JENSEN K., WIRTH N.
*PASCAL : user manual and report*, Lecture notes in computer science n° 18, Springer (1974)

Jones 76 — JONES A., LISKOV B.
*An access control facility for programming languages*, Carnegie Mellon University (1976)

Kaiser 73 — KAISER C.
*Conception et réalisation de systèmes à accès multiples : gestion du parallélisme*, Thèse, Paris (1973)

Kaiser 74a — KAISER C., KRAKOWIAK S.
Analyse de quelques pannes d'un système d'exploitation, *Coll. sur les aspects théoriques et pratiques des systèmes d'exploitation*, IRIA, Paris (1974)

Kaiser 74b — KAISER C., KRAKOWIAK S.
Design and implementation of a time-sharing system : a critical appraisal, *Proc. IFIP congress*, Stockholm (1974)

Knuth 74 — KNUTH D.E.
Structured programming with goto statements, *Computing Surveys* 6,4 (1974)

Krakowiak 73 — KRAKOWIAK S.
*Conception et réalisation de systèmes à accès multiples : allocation de ressources*, Thèse, Paris (1973)

Krakowiak 74 — KRAKOWIAK S., MOSSIERE J.
*Quelques problèmes de la programmation modulaire*, note interne, Grenoble (1974)

Krakowiak 75 — KRAKOWIAK S., MOSSIERE J.
Systèmes d'aide à la programmation de systèmes, *Journée sur les systèmes d'aide à la programmation*, IRIA, Paris (1975)

Krakowiak 76 — KRAKOWIAK S., LUCAS Ma., MONTUELLE J., MOSSIERE J.
A modular approach to the structured design of operating systems, *Proc. MRI symposium on computer software engineering*, Polytechnic Institute, New York (1976)

Lampson 77 — LAMPSON B.W., HORNING J.J., LONDON R.L., MITCHELL J.G., POPEK G.J.
Euclid report, *ACM SIGPLAN notices*, (1977)

Liskov 74a — LISKOV B., ZILLES S.
Programming with abstract data types. Proc. of a symposium on very high level languages, *ACM SIGPLAN notices* (1974)

Liskov 74b — LISKOV B.
*A note on CLU*, Computation structures groupe memo 112, MIT project MAC (1974)

Lucas 74 — LUCAS Ma.
*Expression du parallélisme dans les langages de haut niveau*, Rapport de DEA, Grenoble (1974)

Lucas 75 — LUCAS Ma.
*Primitives de synchronisation pour langages de haut niveau*, Séminaire de programmation, Grenoble (1975)

Lucas 77 — LUCAS Ma.
*Conception modulaire des systèmes d'exploitation. Outils pour la programmation modulaire,* Thèse de 3ème cycle, Grenoble (1977)

Maynard 70 — MAYNARD B.
*Modular programming,* Butterworths (1970)

Mealy 66 — MEALY G.H.
The functional structure of OS 360, *IBM Systems Journal* 5,1 (1966)

Montuelle 77 — MONTUELLE J.
*Conception modulaire des systèmes d'exploitation. Méthodes et exemple d'application,* Thèse Docteur-Ingénieur, Grenoble (1977)

Morrison 73 — MORRISON J.E.
User program performance in virtual storage systems, *IBM Systems journal,* 12.3 (1973)

Mossière 71 — MOSSIERE J.
*Allocation de ressources dans un système à accès multiples,* Thèse 3ème cycle, Paris (1971) in [Ferrié 72]

Mossière 74 — MOSSIERE J.
*Sur les notions de types, de modules, ...* Note interne, Grenoble (1974)

Mossière 75 — MOSSIERE J.
*Quelques outils d'aide à la programmation des systèmes d'exploitation,* Séminaire de programmation, Grenoble (1975)

Myers 75 — MYERS G.J.
*Reliable software through composite design*, Petrocelli/charter (1975)

Naur 69 — NAUR P., RANDELL B. (Eds)
*Software engineering* (Garmisch 1968), NATO Science committee (1969)

Organick 71 — ORGANICK E.I., CLEARY J.G.
A data structure model of the B 6700 computer system, *ACM SIGPLAN Notices* (1969)

Owicki 76 — OWICKI S., GRIES D.
Verifying properties of parallel programs : an axiomatic approach, *CACM* 19,5 (1976)

Parnas 72a — PARNAS D.L.
A technique for software module specification with examples, *CACM* 15,5 (1972)

Parnas 72b — PARNAS D.L.
On the criteria to be used in decomposing a system into modules *CACM* 15,12 (1972)

Popek 77 — POPEK G.J., HORNING J.J., LAMPSON B.W., MITCHELL J.G., LONDON R.L.
Notes on the design of EUCLID, Proc. of a conference on language design for reliable software, *ACM SIGPLAN notices* 12,3 (1977)

Robert 77 — ROBERT P., VERJUS J.P.
Towards autonomous descriptions of synchronization modules, *Proc. IFIP congress*, Toronto (1977)

Sesame 77 — SESAME (Equipe)
Implantation d'une structure modulaire sur miniordinateur. Réalisation d'un noyau de multiprogrammation, *BIGRE* n° 5 (1977)

# CLASSEMENT PAR CHAPITRE DES REFERENCES BIBLIOGRAPHIQUES

La plupart des références bibliographiques sont citées dans le corps du texte. Cependant, nous avons ajouté quelques titres supplémentaires. En conséquence, nous donnons ici une récapitulation par chapitre.

CHAPITRE 2

Bellino 73, Buxton 70, Crocus 75, Dijkstra 72, Hoare 72a, Jackson 75, Mealy 66, Myers 75, Naur 69, Wirth 71, Zurcher 68.

CHAPITRE 3

De Courcel 74, Dijkstra 72, Herodin 77, Jensen 74, Lucas 74, Parnas 72a, Parnas 72b, Trilling 72.

CHAPITRE 4

Armstrong 73, Baudet 72, Briat 76, Brinch Hansen 74a, Cheval 76b, Cheval 77, Corwin 72, Cristian 77, Crocus 75, Dahl 70, De Remer 75, Derniame 74, Dijkstra 68, Dijkstra 72, Ferrié 71, Ferrié 72, Henderson 76, Herodin 77, Jackson 75, Jensen 74, Maynard 70, Mossière 71, Myers 75, Organick 71, Parnas 72a, Parnas 72b, Wirth 71, Wirth 73, Wirth 77, Zurcher 68

CHAPITRE 5

Bétourné 76, Boulenger 74a, Boulenger 74b, Brinch Hansen 76, Burroughs 64, Corbato 69, Crocus 75, Dahl 70, Dijkstra 76, Hoare 74, Ichbiah 74a, Ichbiah 74b, Jones 76, Knuth 74, Lampson 77, Liskof 74a, Liskov 74b, Montuelle 77, Morrison 73, Mossière 74, Parnas 72a, Popek 77, Wegbreit 74, Wirth 68, Wirth 77, Wulf 75, Wulf 76

CHAPITRE 6

Bekkers 74, Bekkers 77, Brinch Hansen 72a, Brinch Hansen 72b, Brinch Hansen 73, Brinch Hansen 74b, Brinch Hansen 75, Brinch Hansen 76, Campbell 74, Courtois 71, Crocus 75, Dijkstra 67, Dijkstra 71, Flon 76, Habermann 69, Habermann 75, Hoare 72b, Hoare 74, Howard 76, Lucas 74, Lucas 75, Owicki 76, Robert 77, Sintzoff 75.