Mécanisme d'adressage et de protection dans les systèmes informatiques

Serge Guiboud-Ribaud

▶ To cite this version:

Serge Guiboud-Ribaud. Mécanisme d'adressage et de protection dans les systèmes informatiques. Réseaux et télécommunications [cs.NI]. Institut National Polytechnique de Grenoble - INPG; Université Joseph-Fourier - Grenoble I, 1975. Français. NNT : . tel-00004557

HAL Id: tel-00004557
https://theses.hal.science/tel-00004557

Submitted on 6 Feb 2004

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° d'ordre : A.0.11.195

THESE

présentée à

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

POUR OBTENIR LE GRADE DE

DOCTEUR D'ETAT

Serge GUIBOUD-RIBAUD

MECANISMES D'ADRESSAGE ET DE PROTECTION DANS LES SYSTEMES INFORMATIQUES

APPLICATION AU NOYAU GEMAU

Thèse soutenue le 30 juin 1975 devant la commission d'examen :

Président	N. GASTINEL
Examinateur	L. BOLLIET
Rapporteur	C. KAISER
Examinateurs	S. KRAKOWIAK
	F.H. RAYMOND
	J.P. VERJUS

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE		INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
M. Michel SOUTIF	Présidents	M. Louis NEEL
M. Gabriel CAU	Vice-Présidents	MM. Lucien BONNETAIN Jean BENOIT

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

PROFESSEURS TITULAIRES

MM.	ANGLES D'AURIAC Paul	Mécanique des fluides
	ARNAUD Paul	Chimie
	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
	BARNOUD Fernand	Biosynthèse de la cellulose
	BARRA Jean-René	Statistiques
	BARRIE Joseph	Clinique chirurgicale
	BEAUDOING André	Clinique de Pédiatrie et Puériculture
	BERNARD Alain	Mathématiques Pures
Mme	BERTRANDIAS Françoise	Mathématiques Pures
MM.	BEZES Henri	Pathologie chirurgicale
	BLAMBERT Maurice	Mathématiques Pures
	BOLLIET Louis	Informatique (IUT B)
	BONNET Georges	Electrotechnique
	BONNET Jean-Louis	Clinique ophtalmologique
	BONNET-EYMARD Joseph	Pathologie médicale
	BOUCHERLE André	Chimie et toxicologie
	BOUCHEZ Robert	Physique nucléaire
	BOUSSARD Jean-Claude	Mathématiques appliquées
	BRAVARD Yves	Géographie
	CABANEL Guy	Clinique rhumatologique et hydrologie
	CALAS François	Anatomie
	CARLIER Georges	Biologie végétale
	CARRAZ Gilbert	Biologie animale et pharmacodynamie
	CAU Gabriel	Médecine légale et toxicologie
	CAUQUIS Georges	Chimie organique
	CHABAUTY Claude	Mathématiques Pures
	CHARACHON Robert	Clinique Oto-Rhino-Laryngologique
	CHATEAU Robert	Thérapeutique (Neurologie)
	CHIBON Pierre	Biologie animale
	COEUR André	Pharmacie chimique et chimie analytique
	CONTAMIN Robert	Clinique gynécologique
	COUDERC Pierre	Anatomie pathologique
	CRAYA Antoine	Mécanique
Mme	DEBELMAS Anne-Marie	Matière médicale
MM.	DEBERMAS Jacques	Géologie générale
	DEGRANGE Charles	Zoologie
	DELORMAS Pierre	Pneumo-Phtisiologie
	DEPORTES Charles	Chimie minérale
	DESRE Pierre	Métallurgie
	DESSAUX Georges	Physiologie animale
	DODU Jacques	Mécanique appliquée

MM.	DOLIQUE Jean-Michel	Physique des plasmas
	DREYFUS Bernard	Thermodynamique
	DRUCROS Pierre	Cristallographie
	DUGOIS Pierre	Clinique de dermatologie et syphiligraphie
	FAU René	Clinique neuro-psychiatrique
	GAGNAIRE Didier	Chimie physique
	GALLISSOT François	Mathématiques pures
	GALVANI Octave	Mathématiques pures
	GASTINEL Noël	Mathématiques appliquées
	GAVEND Michel	Pharmacologie
	GEINDRE Michel	Electroradiologie
	GERBER Robert	Mathématiques pures
	GERMAIN Jean-Pierre	Mécanique
	GIRAUD Pierre	Géologie
	JANIN Bernard	Géographie
	KAHANE André	Physique Générale
	KLEIN Joseph	Mathématiques pures
	KOSZUL Jean-Louis	Mathématiques pures
	KRAVTCHENKO Julien	Mécanique
	KUNTZMANN Jean	Mathématiques appliquées
	LACAZE Albert	Thermodynamique
	LACHARME Jean	Biologie végétale
	LAJZEROWICZ Joseph	Physique
	LATREILLE René	Chirurgie générale
	LATURAZE Jean	Biochimie pharmaceutique
	LAURENT Pierre-Jean	Mathématiques appliquées
	LEDRU Jean	Clinique médicale B
	LLIBOUTRY Louis	Géophysique
	LONGEQUEUE Jean-Pierre	Physique nucléaire
	LOUP Jean	Géographie
Mle	LUTZ Elisabeth	Mathématiques pures
	MALGRANGE Bernard	Mathématiques pures
	MALINAS Yves	Clinique obstétricale
	MARTIN-NOEL Pierre	Seméiologie médicale
	MAZARE Yves	Clinique médicale A
	MICHEL Robert	Minéralogie et pétrographie
	MICOUD Max	Clinique maladies infectieuses
	MOURIQUAND Claude	Histologie
	MOUSSA André	Chimie nucléaire
	MULLER Jean-Michel	Thérapeutique (néphrologie)
	NEEL Louis	Physique du solide
	OZENDA Paul	Botanique
	PAYAN Jean-Jacques	Mathématiques pures
	PEBAY-PEYROULA Jean-Claude	Physique
	RASSAT André	Chimie systématique
	RENARD Michel	Thermodynamique
	RINALDI Renaud	Physique
	DE ROUGEMONT Jacques	Neuro-chirurgie
	SEIGNEURIN Raymond	Microbiologie et hygiène
	SENGEL Philippe	Zoologie
	SIBILLE Robert	Construction mécanique
	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
	YOCCOZ Jean	Physique nucléaire théorique

PROFESSEURS ASSOCIES

MM.	CHEEKE John	Thermodynamique
	COPPENS Philip	Physique
	CORCOS Gilles	Mécanique
	CRABBE Pierre	CERMO
	GILLESPIE John	I.S.N.
	ROCKAFELLAR Ralph	Mathématiques appliquées

PROFESSEURS SANS CHAIRE

Mlle	AGNIUS-DELORD Claudine	Physique pharmaceutique
	ALARY Josette	Chimie analytique
MM.	AMBROISE-THOMAS Pierre	Parasitologie
	BELORIZKY Elie	Physique
	BENZAKEN Claude	Mathématiques appliquées
	BERTRANDIAS Jean-Paul	Mathématiques pures
	BIAREZ Jean-Pierre	Mécanique
	BILLET Jean	Géographie
Mme	BONNIER Jane	Chimie générale
MM.	BOUCHET Yves	Anatomie
	BRUGEL Lucien	Energétique
	CONTE René	Physique
	DEPASSEL Roger	Mécanique des fluides
	GAUTHIER Yves	Sciences biologiques
	GAUTRON René	Chimie
	GIDON Paul	Géologie et Minéralogie
	GLENAT René	Chimie organique
	GROULADE Joseph	Biochimie médicale
	HACQUES Gérard	Calcul numérique
	HOLLARD Daniel	Hématologie
	HUGONOT Robert	Hygiène et Méd. Préventive
	IDELMAN Simon	Physiologie animale
	JOLY Jean-René	Mathématiques pures
	JULLIEN Pierre	Mathématiques appliquées
Mme	KAHANE Josette	Physique
MM.	KUHN Gérard	Physique
	LOISEAUX Jean	Physique nucléaire
	LUU-DUC-Cuong	Chimie organique
	MAYNARD Roger	Physique du solide
	PELMONT Jean	Biochimie
	PERRIAUX Jean-Jacques	Géologie et minéralogie
	PFISTER Jean-Claude	Physique du solide
Mlle	PIERY Yvette	Physiologie animale
MM.	RAYNAUD Hervé	Mathématiques appliquées
	REBECQ Jacques	Biologie (CUS)
	REVOL Michel	Urologie
	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.	VIALON Pierre	Géologie
	VAN CUTSEM Bernard	Mathématiques appliquées

MAITRES DE CONFERENCES ET MAITRES DE CONFERENCES AGREGES

MM.	AMBLARD Pierre	Dermatologie
	ARMAND Gilbert	Géographie
	ARMAND Yves	Chimie
	BARGE Michel	Neurochirurgie
	BEGUIN Claude	Chimie organique
Mme	BERIEL Hélène	Pharmacodynamique
M.	BOUCHARLAT Jacques	Psychiatrie adultes
Mme	BOUCHE Liane	Mathématiques (CUS)
MM.	BRODEAU François	Mathématiques (IUT B)
	BUISSON Roger	Physique
	BUTEL Jean	Orthopédie
	CHAMBAZ Edmond	Biochimie médicale
	CHAMPETIER Jean	Anatomie et organogénèse
	CHARDON Michel	Géographie
	CHERADAME Hervé	Chimie papetière
	CHIAVERINA Jean	Biologie appliquée (EFP)
	COHEN-ADDAD Jean-Pierre	Spectrométrie physique
	COLOMB Maurice	Biochimie médicale
	CORDONNIER Daniel	Néphrologie
	COULOMB Max	Radiologie
	CROUZET Guy	Radiologie
	CYROT Michel	Physique du solide
	DELOBEL Claude	M.I.A.G.
	DENIS Bernard	Cardiologie
	DOUCE Roland	Physiologie végétale
	DUSSAUD René	Mathématiques (CUS)
Mme	ETERRADOSSI Jacqueline	Physiologie
MM.	FAURE Jacques	Médecine légale
	FONTAINE Jean-Marc	Mathématiques pures
	GAUTIER Robert	Chirurgie générale
	GENSAC Pierre	Botanique
	GIDON Maurice	Géologie
	GRIFFITHS Michaël	Mathématiques appliquées
	GROS Yves	Physique (stag.)
	GUITTON Jacques	Chimie
	HICTER Pierre	Chimie
	IVANES Marcel	Electricité
	JALBERT Pierre	Histologie
	KOLODIE Lucien	Hématologie
	KRAKOWIAK Sacha	Mathématiques appliquées
Mme	LAJZEROWICZ Jeannine	Physique
MM.	LEROY Philippe	Mathématiques
	MACHE Régis	Physiologie végétale
	MAGNIN Robert	Hygiène et médecine préventive
	MARECHAL Jean	Mécanique
	MARTIN-BOUYER Michel	Chimie (CUS)
	MICHOULIER Jean	Physique (IUT A)
Mme	MINIER Colette	Physique
MM.	NEGRE Robert	Mécanique
	NEMOZ Alain	Thermodynamique
	PARAMELLE Bernard	Pneumologie
	PECCOUD François	Analyse (IUT B)
	PEFFEN René	Métallurgie
	PERRET Jean	Neurologie
	PHELIP Xavier	Rhumatologie
	RACHAIL Michel	Médecine interne
	RACINET Claude	Gynécologie et obstétrique
	RAMBAUD Pierre	Pédiatrie
Mme	RENAUDET Jacqueline	Bactériologie
MM.	ROBERT Jean-Bernard	Chimie-Physique

MM.	ROMIER Guy	Mathématiques (IUT B)
	SHOM Jean-Claude	Chimie générale
	STIEGLITZ Paul	Anesthésiologie
	STOEBNER Pierre	Anatomie pathologique
	VROUSOS Constantin	Radiologie

MAITRES DE CONFERENCES ASSOCIES

MM.	COLE Antony	Sciences nucléaires
	FORELL César	Mécanique
	MOORSANI Kishin	Physique

CHARGES DE FONCTIONS DE MAITRES DE CONFERENCES

MM.	BOST Michel	Pédiatrie
	CONTAMIN Charles	Chirurgie thoracique et cardio-vasculaire
	FAURE Gilbert	Urologie
	MALLION Jean-Michel	Médecine du travail
	ROCHAT Jacques	Hygiène et hydrologie

Fait à Saint Martin d'Hères, OCTOBRE 1974.

"MEMBRES DU CORPS ENSEIGNANT DE L'I.N.P.G."

PROFESSEURS TITULAIRES

MM.	BENOIT Jean	Radioélectricité
	BESSON Jean	ELectrochimie
	BONNETAIN Lucien	Chimie Minérale
	BONNIER Etienne	Electrochimie, Electrométallurgie
	BRISSONNEAU Pierre	Physique du solide
	BUYLE-BODIN Maurice	ELectronique
	COUMES André	Radioélectricité
	FELICI Noël	Electrostatique
	PAUTHENET René	Physique du solide
	PERRET René	Servomécanismes
	SANTON Lucien	Mécanique
	SILBER Robert	Mécanique des fluides

PROFESSEUR ASSOCIE

M.	BOUDOURIS Georges	Radioélectricité

PROFESSEURS SANS CHAIRE

MM.	BLIMAN Samuel	Electronique
	BLOCH Daniel	Physique du solide et cristallographie
	COHEN Joseph	Electrotechnique
	DURAND Françis	Metallurgie
	MOREAU René	Mécanique
	POLOUJADOFF Michel	ELectrotechnique
	VEILLON Gérard	Informatique fondamentale et appliquée
	ZADWORNY François	Electronique

MAITRES DE CONFERENCES

MM.	BOUVARD Maurice	Génie mécanique
	CHARTIER Germain	Electronique
	FOULARD Claude	Automatique
	GUYOT Pierre	Chimie minérale
	JOUBERT Jean Claude	Physique du solide
	LACOUME Jean Louis	Géophisique
	LANCIA Roland	Physique atomique
	LESPINARD Georges	Mécanique
	MORET Roger	Electrotechnique nucléaire
	ROBERT François	Analyse numérique
	SABONNADIERE Jean Claude	Informatique fondamentale et appliquée
Mme	SAUCIER Gabriêle	Informatique fondamentale et appliquée

MAITRE DE CONFERENCES ASSOCIE

M.	LANDAU Ioan Doré	Automatique

CHARGE DE FONCTIONS DE MAITRES DE CONFERENCES

M.	ANCEAU François	Mathématiques appliquées

Fait à Saint Martin d'Hères JANVIER 1974

Je désire par ces quelques mots, remercier les personnes qui ont contribué à l'aboutissement de ce travail :

Monsieur le Professeur Noel GASTINEL qui a accepté de présider le jury et qui a soutenu avec ardeur les travaux concernant GEMAU,

Monsieur le Professeur Louis BOLLIET qui m'a dirigé depuis mes premiers travaux et qui m'a constamment encourage et guidé depuis,

Monsieur le Professeur François H. RAYMOND qui m'a fait l'amitié de critiquer et de soutenir les différents travaux,

la mémoire de Monsieur Alain GREBERT qui, par de longues et fructueuses discussicns, m'a apporté une nouvelle conception de l'informatique à l'origine directe des résultats obtenus ici,

Messieurs François ANCEAU, Claude BOKSENBAUM, Claude KAISER, Sacha KRAKOWIAK, Robert MAHL, Guy MAZARE et Jean-Pierre VERJUS, pour leurs nombreuses critiques et suggestions,

les membres de l'équipe GEMAU qui ont contribué directement au projet :

mon ami Claude OTRAGE qui a partagé avec moi la responsabilité du projet,

Messieurs Roland BALTER, Jacques BRIAT, Jacques CHARLET, Philippe DARONDEAU, Jacky ESTUBLIER, Alain INGLESE, Christian JULLIEN, Pierre LAFORGUE, Jacques LECOURVOISIER, Xavier ROUSSET de PINA, Harm SMIT et Madame Irène VATTON, qui ont su, dans des circonstances parfois difficiles, donner le meilleur d'eux-même au projet,

Mesdames CHALAND et BONNEFOY pour la réalisation matérielle de cette thèse, et le service tirage de l'IMAG.

A Gisèle, Véronique
et Stéphane.

S O M M A I R E

PROLOGUE

1. Historique —— p. 1

2. Plan de l'ouvrage —— p. 4

PREMIERE PARTIE : PHILOSOPHIE D'UN SYSTEME

Chapitre 1 : Introduction, philosophie d'un système

1. Motivations —— p. 6

   1.1. Problèmes informatiques

   1.2. Notion de noyau

2. Les besoins de l'utilisateur —— p. 13

   2.1. Moyens de calcul

   2.2. Conservation de l'information

   2.3. Partage de l'information

   2.4. Adaptabilité, substitution

   2.5. Communications

   2.6. Conclusion

3. Le point de vue du concepteur —— p. 17

   3.1. Banalisation des ressources

   3.2. Interpréteurs

   3.3. Mémoire virtuelle segmentée et symbolique

   3.4. Le rôle du concepteur, choix des compromis

   3.5. Notion de sous-système

   3.6. Partage contrôlé de l'information

   3.7. Conclusion : propriétés d'un noyau

4. Concepts généraux d'un système : notre point de vue — p. 27
4.1. Objets manipulés par un système
4.2. Contextes d'adressage
4.3. Relations de désignation
4.4. Structure de l'espace des objets
4.5. Existence de mécanismes semblables dans les systèmes existants

5. Objectifs de GEMAU — p. 43
5.1. Objectifs pratiques
5.2. Objectifs théoriques

DEUXIEME PARTIE : GEMAU

Chapitre 2 : Espaces d'exécution et d'objets de GEMAU

1. Conventions — p. 46

2. Espace d'objets, espace d'exécution p. 47
2.1. Espace d'objets
2.1.1. Rappels et définitions
2.1.2. Exemple de structure de noms
2.2. Espace d'exécution
2.2.1. Définitions et rappels

3. Les objets — p. 52
3.1. Durée de vie d'un objet
3.2. Modes de désignation d'un objet
3.3. Opérateurs sur les objets

4. Modifications de l'espace d'objets — p. 55
4.1. Création d'un nouvel objet
4.2. Destruction d'un objet
4.3. Modification de la description d'un objet
4.4. Obtention d'informations sur un objet
4.5. Exemple

5. Modifications de l'espace d'exécution ———— p. 62

5.1. Exécution d'une procédure

5.2. Fin d'exécution d'une procédure

5.3. Capture d'objets dans l'espace d'exécution

5.4. Libération d'objets capturés

5.5. Exemples

5.6. Exemple d'utilisation des noms locaux

5.7. Moment de liaison

6. Transfert entre les différents espaces ———— p. 73

6.1. Opérateur

6.2. Exemple

7. Protection ———— p. 76

7.1. Introduction

7.2. Restrictions d'adressage

7.3. Restrictions d'utilisation

7.4. Définition

7.5. Exemple

7.6. Portée des restrictions

7.7. Cas de l'opération de liaison

7.8. Accumulation de protections

7.9. Exemple

8. Synchronisation : accès concurrentiel aux objets ———— p. 82

9. Récapitulation sur les objets ———— p. 83

9.1. Objet annuaire

9.2. Objet segment

9.3. Objet périphérique

9.4. Exemple

10. Environnement ———— p. 88
- 10.1. Définition
- 10.2. Exemple
- 10.3. Prérogatives
- 10.4. Relation prérogative - protection
- 10.5. Les primitives en tant qu'objets

11. Liens ———— p. 94
- 11.1. Motivations
- 11.2. Restrictions d'accès
- 11.3. Valeur des liens
- 11.4. Exemple
- 11.5. Cas des liens référençant des liens
- 11.6. Définition
- 11.7. Destruction (ou substitution) des objets référencés au travers d'un lien
- 11.8. Boucles
- 11.9. Evaluation dans le cas des portes
- 11.10. Destruction d'un lien
- 11.11. Transfert d'une arborescence contenant des liens
- 11.12. Lien implicite d'annuaire

12. Cohérence des objets ———— p. 120
- 12.1. Définition du problème
- 12.2. Valeurs discrètes
- 12.3. Opérateur de mise à jour

13. Disponibilité des sous-systèmes ———— p. 124
- 13.1. Notion de volume
- 13.2. Manipulation des volumes
- 13.3. Définition finale de la primitive CREATE
- 13.4. Définition finale d'un objet annuaire
- 13.5. Création d'un annuaire
- 13.6. Remarques

Chapitre 3 : Exemple d'utilisation de GEMAU
Sous-systèmes de gestion d'entrées-sorties

1. Définition des entrées-sorties ———— p. 132
   1.1. Classification des périphériques
   1.2. Définition des périphériques "utilisateurs"
2. Programmation des périphériques ———— p. 137
   2.1. Entrée-sortie
   2.2. Fonctions spéciales
   2.3. Exemple
   2.4. Récapitulatif sur les périphériques
   2.5. Virtualisation d'un périphérique
3. Définition des symbionts d'entrées-sorties ———— p. 143
4. Symbiont de sortie ———— p. 145
   4.1. Problème
   4.2. Différents environnements
   4.3. Fonctions
   4.4. Gestion de la file d'attente
   4.5. Procédure principale du symbiont de sortie
   4.6. Point d'entrée depuis un environnement utilisateur
   4.7. Point d'entrée depuis l'environnement opérateur
   4.8. Exemple
   4.9. Remarque générale
5. Autres exemples de symbionts ———— p. 159
   5.1. Symbiont d'entrée
   5.2. Processus console
6. Conclusion ———— p. 162

TROISIEME PARTIE : COMPARAISONS, ETUDES PARTICULIERES

Chapitre 4 : Désignation, adressage, liaison - signification des noms

1. Définition du problème ——— p. 164
   1.1. Nature évolutive des noms
   1.2. Substitution
   1.3. Propriétés des espaces d'exécution et d'objets

2. Désignation dans les espaces d'exécution ——— p. 170
   2.1. Problèmes de désignation et de référence
   2.2. Espace d'exécution statique
   2.3. Espace d'exécution statique avec section de liaison
   2.4. Information préfixée dans les segments
   2.5. Espace d'exécution récursif, machines récursives
   2.6. Espace d'exécution évolutif
   2.7. Conclusion
   2.8. Utilisation de court-circuits : les registres de base

3. Liaison espace d'exécution - espace d'objets ——— p. 209
   3.1. Définitions
   3.2. Moment de liaison
   3.3. Liaison implicite à la première référence lors de l'exécution (MULTICS
   3.4. Liaison explicite à l'exécution (GEMAU)
   3.5. Liaison avant l'exécution : squelette d'espace d'exécution
   3.6. Liaison avant l'exécution : insertion dans l'espace d'objets
   3.7. Mécanismes et politiques

Chapitre 5 : Protection, relations adressage - protection

1. Le point de vue de l'utilisateur ________ p. 228
   1.1. Introduction
   1.2. Propriétés de la protection
2. Le point de vue du système ________ p. 232
   2.1. Introduction
   2.2. Liaison versus accès
   2.3. Types de protection
   2.4. Aspect dynamique de la protection
   2.5. Règles de changement de domaines
3. Modèle théorique de protection ________ p. 239
4. Protection à l'exécution ________ p. 243
   4.1. Espace d'exécution statique
   4.2. Espace d'exécution évolutif
   4.3. Espace d'exécution récursif
   4.4. Conclusion
5. Relations entre la structure de l'espace des objets et la protection
   5.1. Espace d'objets de MULTICS
   5.2. Cas de GEMAU
   5.3. Résolution des différents mécanismes à l'aide de GEMAU
6. Méfiance réciproque ________ p. 264
   6.1. Définition et solution dans le cas de GEMAU
   6.2. Cas des autres systèmes à noms universels
   6.3. Espaces récursifs

7. Résiliation de protection ———— p. 272
7.1. Contrôle de la diffusion des descripteurs
7.2. Propriétés d'une chaîne d'accès
7.3. Résiliation d'accès
7.4. Résiliation d'accès dans GEMAU
7.5. Propriétés supplémentaires de la résiliation dans GEMAU

8. Substitution, liens non résolus ———— p. 280

9. Conclusion ———— p. 282
9.1. Récapitulatif des propriétés de protection
9.2. Commentaires

QUATRIEME PARTIE : CONCLUSIONS, EXTENSIONS

Chapitre 6 : Conclusions

1. Domaines abordés par l'étude GEMAU ———— p. 285
1.1. Notions importantes
1.1.1. Existence de deux espaces de désignation
1.1.2. Notions de porte et d'environnement
1.1.3. Notion de lien
1.2. Autres aspects de GEMAU
1.2.1. Gestion de processus
1.2.2. Technologie du logiciel : réalisation du noyau sur 10070
1.3. Utilité pratique d'un noyau

2. Limites de l'étude, perspectives ———— p. 293
2.1. Limitations dues à GEMAU, aspects non abordés
2.1.1. Objets construits
2.1.2. Contrôle de type lors d'appels procéduraux
2.1.3. Synchronisation, interblocage
2.1.4. Fiabilité, reconfiguration
2.1.5. Gestion prédictive de ressources

2.2. Limitation des objectifs initiaux
2.2.1. Machines orientées vers les systèmes, conception de systèmes
2.2.2. Distribution
2.2.3. Outils d'écriture de systèmes
3. Synthèse ______ p. 299
3.1. Résultats thécriques
3.2. Résultats pratiques

ANNEXES

Annexe 1. Références bibliographiques ______ p. 301
Annexe 2. Documentation, présentations GEMAU - SPS ______ p. 311
Annexe 3. Glossaire ______ p. 319

PROLOGUE

1. HISTORIQUE

Mon intention n'est pas de faire un mémoire sur l'ensemble des travaux effectués sur les systèmes de SPARTACUS à GEMAU, mais plutôt de faire le point et de montrer les étapes successives et contributions dans le domaine des systèmes.

A Grenoble (IMAG, 1967-1968), conjointement au projet DIAMAG, nous avons mené (C. Boksenbaum et moi-même) une étude d'un système de temps partagé sur petit calculateur DEC PDP-8, le système SPARTACUS [BO 67], [BO 68] [GU 68]. Cette étude a abouti à la réalisation effective d'un système utilisé (encore de nos jours) pour l'initiation des étudiants aux ordinateurs, et à certains de leurs travaux pratiques. Ce petit système qui est conversationnel et supporte 24 consoles, fut parmi l'un des tout premiers de ce type à fonctionner en France. C'est grâce à lui que j'ai pu découvrir un ensemble de problèmes "systèmes" et que j'ai ressenti le besoin de les approfondir plus.

Cet approfondissement fut possible grâce à un séjour prolongé chez Burroughs Corp. (Paoli, PA, USA, 1969-1971) où avec C. Gaylor et D. Grubb, nous avons conçu et réalisé un prototype de système conversationnel et de gestion de base de données (ECHO [EC 69]) pour le B8500 ; un prototype d'ECHO fut réalisé sur une machine du type B3500. Cette étude a permis d'améliorer mes connaissances dans le domaine des systèmes et de leurs applications et d'aborder des aspects importants tels que la protection, l'utilisation simultanée et les réseaux. C'est en particulier dans ce projet, où les langages de commande et de programmation sont les mêmes, que j'ai été amené à considérer le problème des processeurs spécialisés et à voir de façon plus précise la notion d'objet.

Le retour en France (à Grenoble) m'a conduit vers une étude des problèmes liés aux langages (problèmes d'extensibilité) puis à démarrer, avec Claude Otrage, une seconde étude sur les systèmes prenant en compte des relations entre les langages et les systèmes.

A l'origine (1.1.1972), le but de l'étude GEMAU était d'étudier le concept de machines virtuelles et de montrer la possibilité de réaliser ce type de système sur IRIS 80, d'ailleurs le nom GEMAU porte les séquelles de cette origine : "GEnérateur de Machines AUtonomes". Durant le cours de l'année 1972, plusieurs modifications ont amené une modulation de ces objectifs :

a. la réalisation de machines virtuelles équivalentes à du matériel réel, possède un domaine d'application limité ;

b. le coût de production d'un système et le prix attaché à la fiabilité de celui-ci, militent en faveur de l'introduction d'une architecture "fonctionnelle" qui oriente et contraint l'écriture de systèmes dans des structures prédéterminées.

Les diverses expériences précédant l'étude GEMAU ont montré que la structure des systèmes a une influence directe sur le coût, la mise en oeuvre et la maintenance de ceux-ci. En particulier, il est nécessaire de définir des outils pour faciliter la construction de systèmes d'exploitation. Dans ce but, nous avons voulu comprendre d'abord ce qu'est un système, le formaliser et en effectuer une réalisation simple, afin de prouver la faisabilité d'un système réel à partir d'une définition formelle indépendante du matériel.

Pour cela, un prototype a été réalisé sur CII 10070, ainsi qu'un certain nombre de systèmes simples : système conversationnel, système de traitement par lots, système de gestion de périphériques.

GEMAU a été défini a priori, sans tenir compte des contraintes de réalisation particulières, seules des contraintes limitatives ont été prises dans le prototype afin de n'étudier la faisabilité que de certains aspects et de tenir compte des particularités précises du 10070.

L'étude a été menée en même temps que d'autres études parallèles effectuées à l'étranger et a abouti à des résultats comparables avec, sur beaucoup d'études, l'avantage d'être plus qu'une étude "papier".

Dans GEMAU nous avons été amenés à considérer un certain nombre d'aspects : seule une étude complète des mécanismes d'adressage et de protection est abordée ici. Les autres aspects peuvent être trouvés dans certains articles et dans les travaux des membres du groupe GEMAU (cf. Annexe 2).

Les aspects sur lesquels j'ai plus particulièrement travaillé sont décrits dans l'annexe 2 sous les numéros A.1, B.2 (première partie), B.4 (deuxième partie), B.5, C.1 à C.4, F.4, F.5, G.5 (deuxième partie).

La partie SPS (Système à Processeurs Spécialisés) est un début d'extension du projet GEMAU en vue d'une application matérielle. Cette partie n'est pas abordée ici, on peut cependant consulter les travaux cités en F de l'annexe 2.

2. PLAN DE L'OUVRAGE

Cet ouvrage se présente sous la forme de trois grandes parties :

Première partie : PHILOSOPHIE D'UN SYSTEME

Chapitre 1 : Introduction générale, définition des termes et concepts utilisés. Présentation de la philosophie générale de conception des systèmes, motivations et objectifs.

Deuxième partie : GEMAU

Chapitre 2 : Application de la philosophie générale à la définition de l'adressage et de la protection dans GEMAU.

Chapitre 3 : Exemple d'utilisation de l'adressage tel qu'il a été défini au chapitre 2, plus particulièrement dans le cas de systèmes de gestion de périphériques.

Troisième partie : COMPARAISONS, ETUDES PARTICULIERES

Chapitre 4 : Définition des problèmes généraux d'adressage et de liaison dans les systèmes, études de diverses solutions et comparaison avec les solutions de GEMAU : avantages et inconvénients.

Chapitre 5 : Définition des problèmes généraux de protection, étude de diverses solutions et comparaison avec GEMAU. Liaison avec l'adressage.

Quatrième partie : CONCLUSIONS, EXTENSIONS

Cette partie récapitule les avantages et les inconvénients des solutions choisies, montre les objectifs atteints et définit les axes de recherche importants pour effectuer des extensions à GEMAU.

PREMIERE PARTIE

P H I L O S O P H I E D ' U N S Y S T E M E

Chapitre 1

INTRODUCTION, PHILOSOPHIE D'UN SYSTEME

Nous analysons les motivations générales de l'étude et en particulier la grande complexité des systèmes actuels et l'évolution des compromis classiques.

L'idée de *noyau* de système en tant qu'ensemble cohérent d'outils simples est introduite, en se basant sur le fait qu'il faut adapter les systèmes aux applications plutôt que l'inverse.

Pour cela, certains besoins universels des utilisateurs sont définis et une déduction est faite sur les contraintes apportées au concepteur de systèmes. Un certain nombre de propriétés des systèmes sont alors données.

Puis les concepts généraux des systèmes sont introduits, notamment la notion d'objet et les relations entre ces objets (adressage). Et enfin les objectifs pratiques et théoriques de l'étude GEMAU.

1. MOTIVATIONS

1.1. Problèmes informatiques

La complexité d'un système informatique considéré comme agrégat de matériel, logiciel et opérateurs humains, a pour conséquence la difficulté pour les concepteurs de systèmes de saisir les problèmes liés à leurs travaux.

Les résultats les plus fréquents sont, d'une part, la durée très longue des temps de conception, d'écriture et de mise au point des systèmes et, d'autre part, le manque de fiabilité des produits résultants (cf. Software Engineering [SO 69], [SO 70]).

Comme les problèmes à résoudre sont généralement complexes et très variés, il est difficile de leur trouver une solution générale. En considérant les systèmes dits "universels", on s'aperçoit que ceux-ci doivent résoudre des problèmes différents souvent incorrectement sinon nullement définis lors de la conception dudit système (car les solutions qui correspondent à un certain nombre de domaines d'application ne sont pas forcément transposables directement à d'autres domaines).

De plus, l'ensemble des connaissances acquises dans les systèmes et les concepts utilisés sont fondés sur des compromis (matériel, logiciel) qui étaient valables pour une certaine technologie. Or les termes de ces compromis semblent maintenant inversés ou sur le point de l'être. En particulier, une évolution technologique rapide s'est opérée au cours de ces dernières années : on a vu par exemple décroître de façon considérable le coût des petits ordinateurs (cf. figure 1). Cette figure ne donne cependant qu'une idée très limitée du phénomène, car la chute des prix prévue pour les composants est beaucoup plus vertigineuse. Ce développement de la technologie n'a pas, pour l'instant, d'équivalent dans le développement du logiciel.

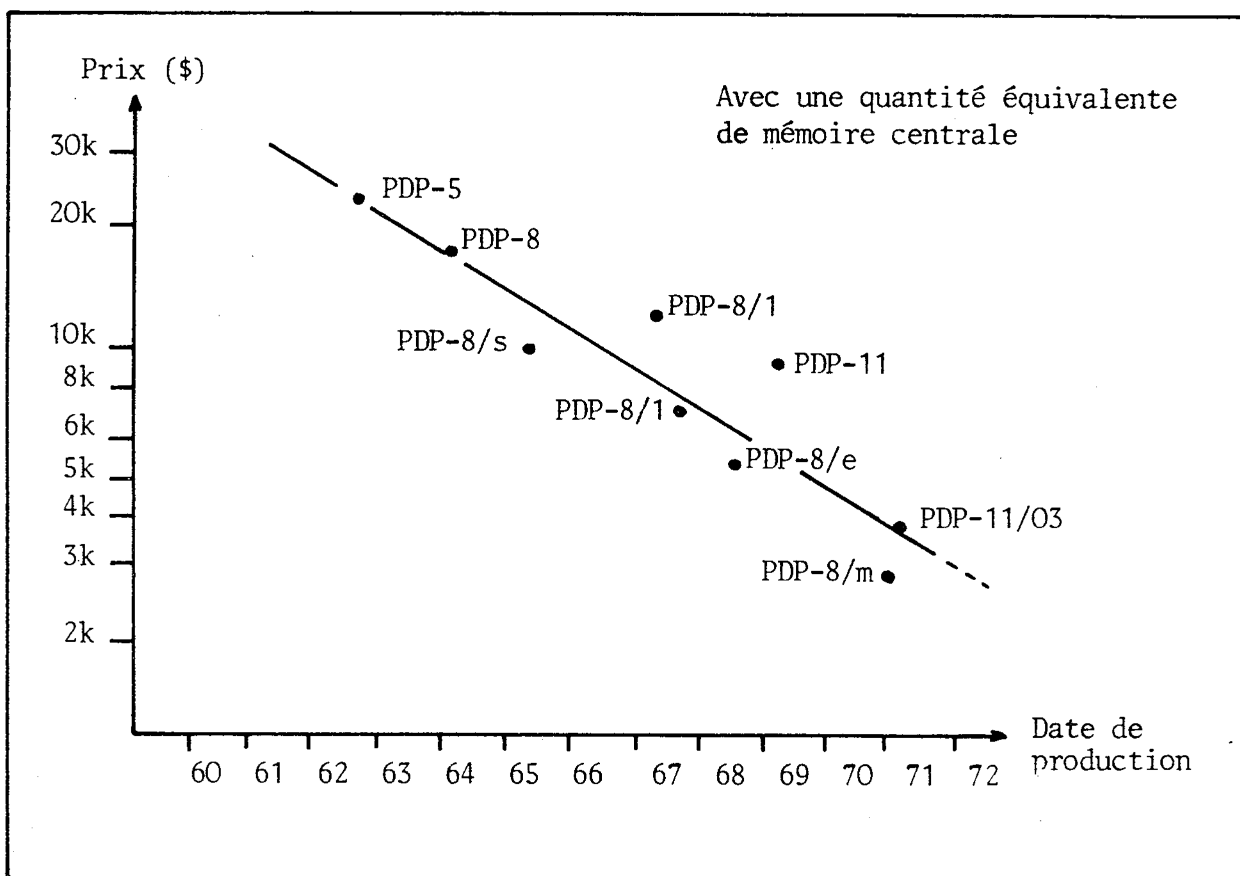


Figure 1 : Evolution des prix de quelques petits ordinateurs

La figure 2 (inspirée de [SO 69]), bien qu'ancienne, montre le coût de construction (en nombre de lignes de programme) de quelques systèmes d'exploitation. Sachant que le nombre d'instructions écrites et testées par programmeur est très faible (quel que soit le langage de programmation), ceci donne une idée du nombre de programmeurs nécessaires au développement de tels systèmes.

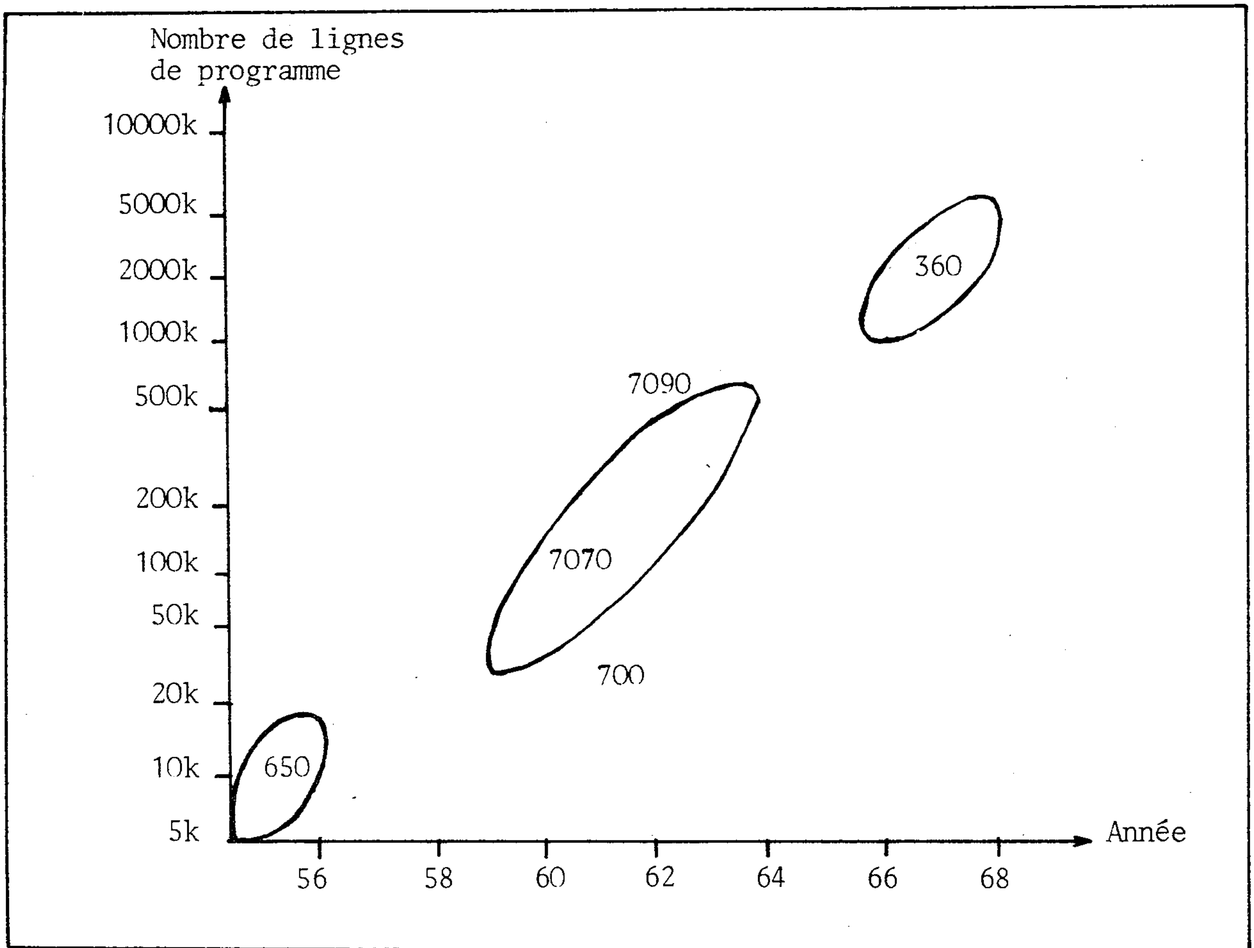


Figure 2 : Le mur de la complexité

Considérons maintenant le coût d'un ordinateur qui peut être défini par la formule suivante :

coût d'utilisation =	coût du matériel	(1)
	+ $\frac{1}{n}$ [coût du développement (matériel + logiciel)]	(2)
	+ coût de maintenance	(3)

n = nombre de produits vendus.

En comparant cette formule aux figures 1 et 2, on s'aperçoit que le facteur (1) est décroissant et qu'il faut par conséquent faire porter nos efforts sur les facteurs (2) et (3).

Les coûts de développement viennent de la conception générale de la machine et sont fonction d'un certain nombre de contraintes logicielles et matérielles, comme par exemple la compatibilité avec des machines existantes, la possibilité de changements ultérieurs (expansion possible), la fiabilité et le domaine des applications à résoudre (créneau). Ils incluent aussi les coûts de réalisation de matériel et de logiciel.

Il semble que les compromis puissent être remis en cause et que l'on puisse admettre un surcroît de matériel au profit de la facilité de conception et de la fiabilité de l'ensemble.

Nous nous sommes intéressés aux systèmes par l'observation d'un certain nombre de systèmes informatiques (MULTICS, Burroughs B6500/6700/7500, CP 67 sur IBM 360/67), en essayant d'en dégager les règles essentielles pour mieux comprendre la réalisation pratique des systèmes ou de l'architecture des calculateurs.

De plus, nous nous sommes intéressés en parallèle, à des études "papier" (i.e. sans réalisation) qui traitent des problèmes des systèmes (Dennis, Lampson pour la protection, Dijkstra pour la méthodologie et la synchronisation, ..).

L'étude de systèmes commerciaux existant a donné ([SO 69], [SO 70]) l'exemple de systèmes lourds, peu extensibles et complexes. Ce degré de complexité tend à faire augmenter le nombre de programmeurs associés à un projet, ainsi que la taille des programmes résultants. Tout ceci introduit immédiatement le problème de la communication entre les programmeurs et donne une bonne approximation des difficultés de mise au point et de maintenance de tels systèmes.

Une des raisons majeures de ces difficultés réside dans le fait que les entités et les opérations définies dans la plupart des systèmes courants sont généralement éloignées (pour ne pas dire différentes) des entités et opérations que l'utilisateur désire [GA 74]. Il s'agit donc de réduire la distance entre les applications d'un utilisateur et le système. Le système étant considéré ici comme un ensemble de fonctions (logicielles, matérielles) qui supervisent les opérations de la machine, assurent leur coordination et facilitent l'utilisation de la machine. On peut facilement envisager l'idée d'élever le niveau du matériel pour mieux répondre aux besoins des systèmes [MA 72]. Cette approche a déjà été utilisée depuis longtemps dans le domaine des machines spécialisées pour le traitement des langages de haut niveau, ce qui a contribué à diminuer la distance entre les utilisateurs et la machine [GA 74]. De telles machines sont spécialisées pour optimiser l'exécution de programmes écrits en langage de haut niveau, c'est-à-dire que leur structure est très semblable aux constructions du langage pour lequel la machine est définie. Comme l'architecture interne est très proche des besoins du langage, le compilateur nécessaire est beaucoup plus simple que sur une machine traditionnelle. Ces machines possèdent des instructions spécifiques et des structures de données adaptées au langage à traiter ; beaucoup de fonctions effectuées normalement par les systèmes sont alors réalisées par le matériel [MA 72].

1.2. Notion de noyau

L'impossibilité à prévoir toutes les utilisations possibles (domaines d'application) nous amène à l'idée de base concernant les études que nous avons menées : *il faut fournir un ensemble d'outils simples permettant d'écrire des systèmes d'exploitation, de les adapter aux besoins, plutôt que résoudre des problèmes académiques dus aux difficultés rencontrées dans les solutions existantes.*

Ces outils doivent permettre la définition des entités manipulées par un système et les règles qui régissent ces manipulations. La définition formelle de l'ensemble des entités et règles est appelée une *machine abstraite*, l'automate permettant de réaliser une telle machine étant appelé un *noyau*.

Les règles de manipulation des entités définissent les moyens de communication [BA 71] entre les systèmes et le monde extérieur, ou entre les systèmes eux-mêmes. Sans communication un programme n'a aucune utilité, il ne peut ni obtenir des données à traiter, ni fournir ses résultats. Les mécanismes de communication ont été (et sont encore) séparés en diverses catégories selon l'entité avec laquelle la communication doit s'établir. Ainsi trouve-t-on parmi les diverses entités :

. les périphériques d'entrées-sorties (lecteur de cartes, imprimantes, bandes magnétiques, etc ..),

. les terminaux qui, bien que périphériques d'entrées-sorties, sont généralement traités différemment, ils ont leur propre système de gestion. De plus, selon le type de terminal, la communication s'effectue différemment par exemple si le terminal fonctionne en mode "message" ou "caractère", le type de la ligne sur lequel le terminal est connecté, le type du terminal lui-même (code des caractères : ASCII, EBCDIC, .., ou caractères spéciaux différents). L'utilisateur d'un terminal doit, en général connaître des spécifications précises de son terminal.

. Les fichiers de "travail" qui n'existent souvent que pour pallier à la pénurie de mémoire centrale,

. les programmes de l'utilisateur qui sont directement appelés par une instruction spéciale,

. les programmes de service du système qui sont appelés par une autre instruction spéciale et qui s'exécutent avec des prérogatives différentes de celles de l'utilisateur,

. etc ..

Dans la plupart des systèmes existants, il y a un nombre très divers de mécanismes, souvent plus que le nombre d'entités différentes et chacun de ces mécanismes a une syntaxe d'appel différente de celle des autres. Un programme qui demande une communication avec une de ces entités est donc obligé de connaître le type de cette entité.

Ainsi il est très difficile de réaliser un programme qui puisse accepter ses données, indifféremment depuis un lecteur de cartes, un terminal ou un autre programme, sans avoir prévu tous les cas possibles lors de la programmation. Le premier objectif que nous nous fixons est la définition de *mécanismes cohérents* de communication entre les entités manipulé par un système. Ce que nous voulons ce sont effectivement des mécanismes et non des politiques, car il faut éviter de définir toute solution qui hypothèque trop l'utilisation de la machine (y compris le système d'exploitation). Nous pensons que l'unification de ces mécanismes ne peut qu'être un pas vers la définition de *systèmes plus simples* et à *échelle humaine*, c'est-à-dire tels qu'il soit possible à un être humain de comprendre facilement ce qu'est un système.

Nous avons vu qu'un des problèmes des systèmes actuels est la distance entre la machine et l'utilisateur ; pour réduire cette distance nous allons définir un noyau et pour cela nous étudierons successivement les points suivants :

a. les besoins des utilisateurs (§ 2),

b. puis ce que le concepteur d'un système peut tirer des besoins de l'utilisateur et les propriétés nécessaires (§ 3),

c. et de là nous déduirons les concepts généraux à mettre en oeuvre dans un noyau et une structure générale qui permette de résoudre les points a. et b. En particulier, nous présenterons une définition précise des entités manipulées par un noyau.

2. LES BESOINS DE L'UTILISATEUR

2.1. Moyens de calcul

Un des premiers effets de l'introduction de l'informatique a concerné le domaine du calcul numérique. Depuis, une évolution très nette s'est effectuée, mais on demande toujours à l'ordinateur d'être un outil de traitement et seule la définition de ce moyen de traitement (numérique, chaîne de caractères, gestion, ..) change d'une application à l'autre.

L'on peut dire que l'intérêt de l'utilisateur réside dans l'utilisation *d'un langage de programmation adapté à ses propres besoins.*

2.2. Conservation de l'information

En plus du besoin originel de moyens de calcul, on demande de plus en plus à un système de pouvoir conserver de l'information pendant le calcul et même en dehors des moments de calcul. Pour cela, il faut pouvoir *désigner* les ensembles d'information (objets) afin de les retrouver par la suite, ces objets pouvant être éventuellement des programme. Cette désignation doit être *symbolique* : nous entendons par là qu'elle doit être telle qu'elle n'introduise pas de relations inutiles entre deux ensembles qui n'en possèdent pas au niveau de l'application. L'utilisateur doit être libre de choisir les noms des objets qu'il veut sans prendre en considération ceux choisis par d'autres utilisateurs, qui peuvent d'ailleurs être les mêmes noms. Les objets doivent avoir une taille variable dans le temps, c'est-à-dire qu'ils peuvent contenir plus ou moins d'information. Ces objets doivent aussi pouvoir être créés et détruits dynamiquement au gré de l'utilisateur.

2.3. Partage de l'information

Un utilisateur peut utiliser une machine pour une application indépendante, mais il peut aussi vouloir *communiquer* et *partager de l'information* avec d'autres utilisateurs. Les raisons de ce partage peuvent être diverses

a. utiliser les travaux déjà effectués par d'autres utilisateurs et conservés dans le système,

b. consulter ou mettre à jour des données, comme par exemple dans le cas d'application du type : réservation de places,
systèmes de vente par correspondance,
gestion de stocks,
tout système de bases de données ..

c. mettre à la disposition de la communauté des utilisateurs (ou seulement de certains utilisateurs) des programmes ou des fichiers.

Ce partage est un partage logique qui correspond à une nécessité pour l'utilisateur et il ne faut pas le confondre avec le partage, pour des raisons d'efficacité, de ressources physiques, comme par exemple le fait que certaines machines demandent qu'un programme soit entièrement résident en mémoire centrale pour pouvoir l'exécuter.

Le partage logique peut être considéré soit séquentiellement, soit simultanément, c'est-à-dire qu'un même objet peut être utilisé par deux utilisateurs différents en des instants distincts, ou au même instant. Du point de vue de l'utilisateur, les problèmes ne se posent pas exactement dans les mêmes termes. En effet, celui-ci partage de l'information, il peut être amené à modifier cette information qui ne peut alors être utilisée par un autre utilisateur qu'une fois la modification terminée. Par exemple, dans un système de réservation de places, la mise à jour d'une réservation s'effectue en deux temps : vérifier qu'il y a de la place et ensuite prendre les places demandées. Dans un partage séquentiel, le problème est résolu, mais dans un partage simultané, il faut que l'utilisateur possède des moyens d'indiquer au système comment effectuer la *synchronisation* des différents utilisateurs pour l'utilisation de l'objet.

A partir du moment où un utilisateur partage de l'information avec d'autres utilisateurs, il faut aussi pouvoir *protéger* cette information. Par exemple, certains utilisateurs ne pourront accéder à certaines informations qu'en lecture, alors que d'autres pourront les accéder aussi bien en lecture qu'en écriture.

2.4. Adaptabilité, substitution

Une des raisons essentielles du partage est la possibilité de pouvoir réaliser de grosses applications à plusieurs utilisateurs, en particulier de pouvoir utiliser le travail des autres. Ceci pose le problème de l'évolution d'une telle application, en particulier de son *adaptabilité* à des besoins nouveaux découverts au cours de sa réalisation.

Un second aspect du partage concerne la méthodologie de réalisation de tels projets où il devient indispensable pour un utilisateur de développer sa propre partie de l'application en parallèle avec le développement des autres parties. Il faut alors pouvoir utiliser des procédures (objets) qui simulent, de façon généralement très simplifiée, les procédures qui sont en cours de développement par d'autres utilisateurs. Une fois que les procédures développées par ces autres utilisateurs sont au point, il faut les intégrer dans cette partie de l'application, ceci se fait par la *substitution* d'objets. Cette substitution doit être effectuée en ligne pendant que le système s'exécute.

Cet objectif est très fortement justifié par la remarque de Corbato, Saltzer et Clingen [CO 72] (p. 573) :

> "Because the system is so large, the most powerful maintenance tool available was chosen - the system itself. With all the system modules stored on-line, it is easy to manipulate the many components of different versions of the system. Thus it has been possible to maintain steadily for the last year or so a pace of installing 5 or 10 new or modified systems modules a day. Some three-quarter of these changes can be installed while the system is in operation. /.../ This on-line maintenance capability was proven indispensable for the rapid development and maintenance of MULTICS since it permits constant upgrading of the user interface without interrupting the service. We are just beginning to see the instances of user-written applications ..."

2.5. Communications

Durant la mise au point de son programme, un utilisateur a besoin d'outils qui lui permettent de dialoguer avec la machine, par exemple pour récupérer des erreurs de programmation ou pour arrêter une exécution et consulter son programme. Ces possibilités sont fournies par des mécanismes dits *"points de controle"* qui permettent de définir les actions à prendre sur certains événements (erreur, arrêt, ..) pendant le déroulement du programme.

Ce problème de mise au point nécessite par ailleurs la présence de mécanismes de *protection*. En effet, même pour un seul utilisateurs, sans partage d'information avec d'autres utilisateurs, il existe des besoins de protection ; par exemple la mise au point d'une nouvelle procédure nécessite que toute erreur se produisant dans cette procédure ait un effet aussi limité que possible sur l'ensemble des données et procédures déjà mises au point ; pour cela, des outils permettant un accès restreint sont très utiles.

2.6. Conclusion

En conclusion on peut dire que :

les besoins qu'un utilisateur désire voir satisfaits par un système sont les suivants :

- moyens de calcul,
- conservation et désignation d'information,
- possibilités de partage d'information avec d'autres utilisateurs, ce qui nécessite :
  - une protection adéquate de l'information,
  - une synchronisation sur l'accès à l'information,
- adaptabilité par substitution dynamique,
- communication avec le système et avec le monde extérieur,
- éventuellement possibilité d'exécutions parallèles.

3. LE POINT DE VUE DU CONCEPTEUR

3.1. Banalisation des ressources

Il existe deux façons générales de considérer les systèmes d'exploitation : d'une part,

a. comme une machine abstraite qui fournit des facilités et ressources à un niveau plus élevé que celui d'une machine (matériel),
et d'autre part,

b. comme un allocateur de ressources physiques afin d'utiliser efficacement la machine.

Si le premier point de vue correspond plus à notre conception que le second, il est cependant intéressant de remarquer que dans les deux cas le terme de ressources est utilisé. Le second point de vue correspond plus à la définition de techniques efficaces d'allocation ; nous n'en parlerons donc pas si ce n'est de temps en temps pour citer des exemples de réalisation.

Nous définissons par *objet* une ressource quelle que soit sa réalisation.

Le point de départ que nous considérons est fourni par la définition de l'ensemble des besoins des utilisateurs, le rôle du concepteur de système étant de résoudre les problèmes matériels posés au niveau de la solution des besoins. Le rôle du système est donc de réaliser la correspondance entre le *monde virtuel* tel que le désire l'utilisateur et le *monde réel* tel qu'il existe.

Un des problèmes liés à cette correspondance vient de la pénurie des ressources physiques dont la mémoire centrale est un exemple flagrant. En effet, c'est tout simplement une contrainte de réalisation qui nécessite la présence entière d'un programme en mémoire centrale en vue de son exécution. Cette contrainte a déjà disparu aujourd'hui avec l'apparition de la notion de mémoire virtuelle. L'utilisateur désire une *mémoire virtuelle* [DA 65] indépendante de la machine, la responsabilité de la gestion de cette mémoire sur divers espaces physiques (mémoire centrale, mémoire secondaire) étant entièrement du ressort du système. Cette mémoire virtuelle doit être suffisamment grande pour permettre des applications variées.

Ceci n'est qu'un exemple parmi d'autres qui montrent qu'un utilisateur dépense une énergie souvent considérable à gérer des ressources physiques, donc à traiter des aspects sans liaison avec le problème à résoudre. Ainsi, l'introduction de systèmes de multiprogrammation et de partage de temps est-elle déjà un pas vers la gestion du temps, temps qui dès maintenant est très souvent une ressource transparente à l'utilisateur. Il en est de même pour la mémoire centrale, ainsi que nous venons de le voir.

Un aspect plus général de gestion qui doit être transparente à l'utilisateur correspond au multiplexage des ressources dont la gestion du temps est un exemple. Il en est de même, d'une part, pour l'utilisation partagée de certaines ressources comme la mémoire (centrale, secondaire), dans la mesure où ce partage correspond à une tentative de meilleure utilisation des ressources physiques et, d'autre part, pour l'existence de plusieurs processeurs banalisés qui correspondent à une augmentation des performances du système tout entier.

En conclusion, on peut dire que l'utilisateur veut être libéré de contraintes physiques et ne veut plus qu'utiliser des ressources propres à son problème, en ignorant comment ces ressources sont éventuellement multiplexées (dans le temps ou dans l'espace).

3.2. Interpréteurs

Nous avons vu que le premier besoin de l'utilisateur concerne la définition d'un langage de programmation, c'est-à-dire l'existence d'un moyen de traitement. Ce langage se traduit par la définition d'un certain nombre de structures de données et d'instructions pour manipuler ces données. Le composant chargé de mettre en oeuvre ces instructions (traitement) s'appelle *un processeur* ou *un interpréteur*.

Considérons un ordinateur du type IBM S/360 : cette machine possède un adressage par octet (8 chiffres binaires), ainsi que 16 registres généraux pour l'adressage (base), l'arithmétique et l'indexation. Il existe un ensemble d'instructions de syntaxe et sémantique différentes. Chaque instruction est lue depuis une zone où réside le programme vers le processeur 360 pour y être exécutée. La définition précise de ces différentes instructions et des données manipulées constitue un langage et l'interpréteur associé est un processeur S/360. Ce qui est important ici c'est que l'on puisse réaliser des processeurs S/360 de différentes façons : à l'aide de matériel, de logiciel, ou de microprogrammation. Dans tous les cas, il faut que l'interpréteur corresponde au langage S/360, mais la réalisation effective ne doit avoir d'influence que sur la performance finale et le prix de réalisation, mais aucunement sur la définition du langage.

On peut ainsi imaginer une installation qui désire utiliser un certain langage et que, dans un premier temps, l'interpréteur correspondant soit réalisé entièrement par du logiciel, puis, la charge augmentant, que l'on puisse remplacer cette réalisation par une autre réalisation entièrement cablée. *Dans tous les cas, il ne faut pas que l'utilisateur ait à reprogrammer les applications écrites dans le langage considéré.* Cette modification des compromis classiques a été grandement utilisée pour certains langages tels que ALGOL [HA 68], FORTRAN [BA 67], EULER [WE 67], APL [HA 71] et PASCAL [AN 74a]. Nous ne nous intéressons pas ici aux techniques propres à la réalisation de tels processeurs spécialisés, mais beaucoup plus au fait que les compromis logiciel - matériel peuvent être très différents de ce qu'ils sont sur du matériel classique.

On peut donc envisager de façon très simple une manière de définir une gamme de machines, où toutes les machines ayant la même définition logique, pourraient être composées de processeurs logiciels ou matériels, ou un mélange des deux. Du point de vue de l'utilisateur il s'agit toujours de la même machine, du point de vue du concepteur il faut posséder des outils pour permettre cette expansion ; cet outil doit donc permettre de *substituer* des processeurs à d'autres processeurs.

Un second problème lié aux interpréteurs est celui qui correspond à la notion de *niveau d'interpréteur,* c'est-à-dire au fait qu'un interpréteur peut être réalisé dans un langage particulier, donc être un programme pour l'interpréteur correspondant au second langage. Nous supposons seulement qu'un interpréteur est primitif, c'est-à-dire que nous ne nous intéressons pas ici à la réalisation effective de tels interpréteurs ; le lecteur intéressé peut consulter [DA 73], [AN 74], [DE 72], ainsi que [F.4 et F.5 de l'annexe 2], pour des études plus détaillées du problème.

3.3. Mémoire virtuelle segmentée et symbolique

Nous avons vu que l'utilisateur demande de pouvoir désigner et conserver de l'information (objets) ; il y a deux solutions générales à ce problème un système classique de fichiers et une *mémoire virtuelle segmentée et symbolique.* Il faut comprendre ici le terme "mémoire virtuelle" comme beaucoup plus vaste que le sens originel d'extension de mémoire centrale qui correspond au besoin de libérer l'utilisateur de contraintes matérielles dues à l'exiguité de la mémoire centrale. Nous pensons qu'il faut étendre cette notion à tout objet manipulable, en particulier la notion classique de fichier comme objet distinct de la mémoire ne tient plus : un fichier est un ensemble structuré de données au même titre que les données manipulées localement par un processeur. Cette tendance à l'uniformisation, bien qu'ancienne [DA 68], n'a cependant été que peu mise en pratique [BE 70], [VE 73], [OR 72], [HA 73]. Nous pensons qu'il faut aller plus loin et mettre dans cette mémoire tout type d'objet, y compris les périphériques. L'objectif du concepteur est donc d'homogénéiser, du point de vue de la désignation, les différents éléments d'un système, c'est la définition du *concept d'objet.*

La terminologie "mémoire segmentée" signifie que la mémoire n'est pas linéaire par nature, car les objets qui la composent sont de nature (et donc de taille et caractéristiques) différente. De plus, en règle générale, seule l'identification de deux objets distincts (programmes, fichiers, périphériques, autres données ..) est importante pour l'utilisateur car c'est généralement une coïncidence si deux objets sont situés linéairement l'un par rapport à l'autre. C'est la raison pour laquelle nous employons le terme *symbolique*. La représentation réelle des noms de ces objets est donc seulement une convention dans les systèmes, ce peut être soit une chaîne de caractères, soit un entier, soit toute autre convention.

Il faut remarquer que le terme mémoire virtuelle segmentée est très souvent utilisé dans les systèmes classiques dans le sens d'une mémoire virtuelle paginée, c'est-à-dire avec deux niveaux de tables pour la réalisation : table des segments et pour chaque segment, table des pages. Pour nous ce point de vue est très restrictif et concerne uniquement des techniques de réalisation de mémoire virtuelle comme extension de la mémoire centrale et ne nous intéresse pas ici.

3.4. Le rôle du concepteur, choix des compromis

Ce que nous venons de voir dicte la méthodologie à suivre par le concepteur de système, à savoir fournir un ensemble d'outils fonctionnels, à mesurer et évaluer les applications décrites à l'aide de ces outils, et alors seulement à définir les compromis de la réalisation. En particulier, on peut concevoir, pour des objectifs de performances ou de prix différents, des réalisations différentes. Chacune de ces réalisations admettant des changements futurs comme par exemple, un processeur qui peut être réalisé dans un premier temps de façon logicielle et être ensuite remplacé par un ou plusieurs composants matériels.

Toute solution définissant les compromis a priori n'aboutit qu'à forcer l'application à se plier à ces compromis, ceci est le cas dans la réalisation de GEMAU sur 10070 - IRIS 80, où la mémoire virtuelle ne peut qu'être paginée, ce qui force les applications à regrouper les procédures en ensembles d'importance assez conséquente. Pour ce cas précis, une référence aux résultats obtenus par Burroughs où la taille des procédures (logiques) est très faible (de l'ordre de quelques dizaines de mots), permet de penser que les choix à faire pour la construction d'un matériel adapté donneraient un résultat très différent de celui obtenu dans le cas de la pagination, au moins pour la définition des procédures.

Le rôle du concepteur de systèmes est donc de fournir des outils permettant toute une gamme de réalisations, donc de définir fonctionnellement le système avant de choisir les compromis.

3.5. Notion de sous-système

Si nous considérons un certain nombre de systèmes classiques, nous assistons à une certaine évolution. Dans cette évolution on peut citer notamment : les systèmes en monoprogrammation, les systèmes en multiprogrammation, l'introduction d'une application particulière telle que l'accès à distance, l'accès conversationnel par l'intermédiaire de techniques de temps partagé, etc ..

On s'aperçoit rapidement que tous ces différents types de systèmes font appel à des concepts similaires, d'où l'idée de définir *un ensemble d'outils plus synthétiques* qui ne sont pas des systèmes en eux-mêmes, mais permettent de construire les différents types de systèmes dont nous venons de parler.

A titre d'exemple, considérons les rapports qui existent entre un système d'exploitation et un utilisateur de ce système. L'utilisateur demande un certain nombre de services et le système gère les objets nécessaires à ce service. Très souvent, dans les systèmes classiques, on peut être amené à définir, pour un utilisateur, une application complexe qui gère plusieurs "sous-utilisateurs". Par exemple, cet utilisateur peut être une base de données (service de réservation de places, service de gestion intégrée dans une entreprise). Dans ce cas, l'utilisateur joue le rôle d'un système pour ses "sous-utilisateurs", en particulier ces derniers seront sous la responsabilité administrative et comptable du premier. Le système global ignorant même l'existence de ces sous-utilisateurs. Nous appelons *sous-système* un tel utilisateur. Dans ce cas, le sous-système et ses utilisateurs jouent un rôle similaire respectivement au système et sous-système ; cette relativité est schématisée par l'exemple de la figure 3.

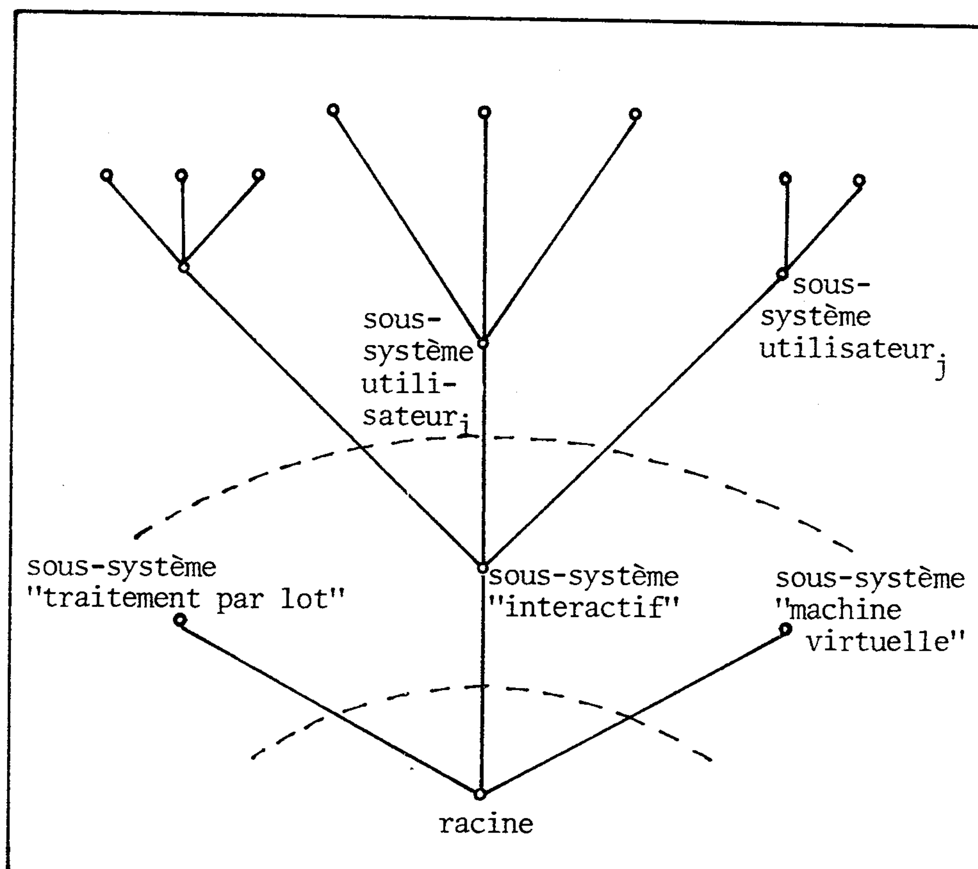


Figure 3 : relativité de la notion système-utilisateur (hiérarchie)

Alors que les mécanismes nécessaires à la réalisation des "sous-utilisateurs" par le sous-système sont les mêmes que ceux nécessaires à la réalisation des sous-systèmes, il faut très souvent dans les systèmes actuels redéfinir de nouveaux mécanismes car les mécanismes de base du système ne sont pas accessibles aux différents sous-systèmes. Par exemple, il faut redéfinir une notion d'activité indépendante et un moniteur permettant d'allouer les ressources à ces activités, qui ne sont qu'une seule activité du point de vue du système (par exemple un sous-système temps partagé). D'où une duplication d'algorithmes semblables à tous les niveaux de réalisation ; nous pensons qu'un noyau incluant des mécanismes d'extensibilité est une réponse à ce problème. Il est important de remarquer que l'existence d'un noyau n'empêche pas un utilisateur de bâtir un sous-système comme dans les systèmes classiques (c'est-à-dire en recréant tout), mais ceci ne peut résulter que d'une volonté bien définie de la part du concepteur de ce sous-système. Il y a une analogie très importante entre cette hiérarchie de sous-systèmes et la notion de hiérarchie d'interpréteur [DE 72].

3.6. Partage contrôlé de l'information

Nous avons vu que la mémoire virtuelle segmentée et symbolique, ainsi que la notion de sous-système, permettent de résoudre certains besoins des utilisateurs (application ou conception de sous-systèmes). Cependant nous avons ignoré jusqu'à maintenant le problème du partage de l'information. Etant donné que nous voulons définir un ensemble de mécanismes utilisables par différents utilisateurs, nous utilisons la même mémoire virtuelle pour tous, avec une structure hiérarchisée comme pour les sous-systèmes.

Cette hiérarchie stricte ne permet pas de résoudre le partage d'information entre des sous-systèmes différents, il faut donc offrir des mécanismes qui permettent depuis un sous-système d'accéder à un autre sous-système indépendant, c'est-à-dire que la structure des différents utilisateurs n'est plus une hiérarchie stricte et ne peut donc pas être traduite par une arborescence comme celle fournie par la figure 3. Nous utilisons pour cela une structure de graphe qui permet de définir des relations de partage entre différents sous-systèmes. La figure 4 donne un exemple intuitif d'un tel type de structure.

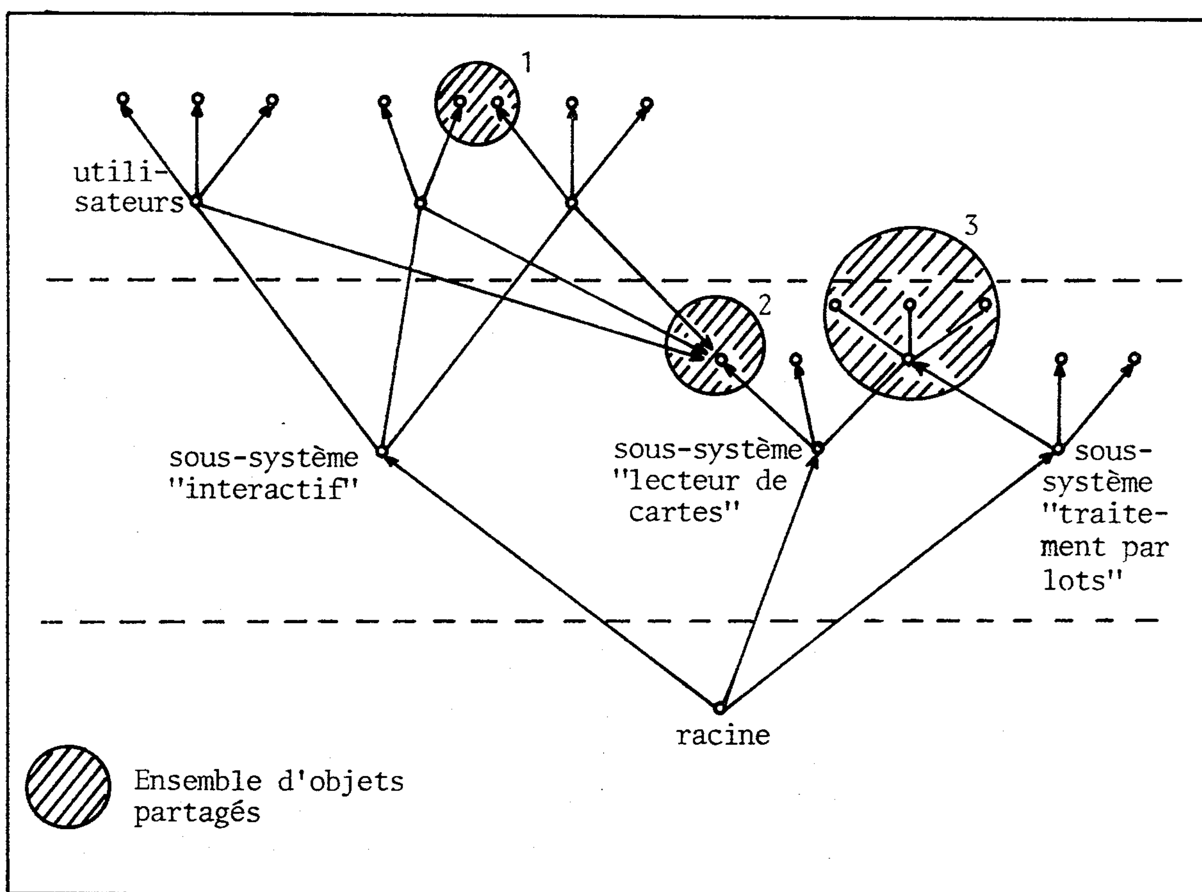


Figure 4 : Partage entre utilisateurs

Dans cette figure, nous avons représenté trois ensembles d'objets partagés entre différents sous-systèmes :

1 : entre deux utilisateurs du sous-système "interactif"

2 : entre les utilisateurs du sous-système "interactif" et un composant du sous-système "lecteur de cartes"

3 : entre les sous-systèmes "lecteur de cartes" et "traitement par lots".

Un même objet (ou ensemble d'objets) est donc accessible par différents sous-systèmes ; il faut prévoir la protection à associer aux différents accès, par exemple tel objet qui ne peut être qu'exécuté par tel utilisateur et lu par tel autre, ce partage contrôlé inclut la synchronisation éventuelle sur l'objet.

3.7. Conclusion : propriétés d'un noyau

D'après tout ce que nous venons de voir (besoins de l'utilisateur, concepts communs aux sous-systèmes), on peut dire que

l'objectif final est la définition et la réalisation d'un noyau de système qui soit *un outil de base de recherches*.

Les mécanismes de ce noyau doivent avoir les propriétés suivantes :

- permettre l'utilisation simultanée,
- assurer la cohérence des mécanismes de communication,
- permettre une grande souplesse, c'est-à-dire fournir la possibilité d'extension et de substitution dynamique,
- permettre un partage naturel, c'est-à-dire définir des outils de protection et de synchronisation,

4. CONCEPTS GENERAUX D'UN SYSTEME : NOTRE POINT DE VUE

Nous allons essayer de voir plus en détail ce qu'est un système, c'est-à-dire de définir avec plus de précision le domaine "système". Cette définition, qui est bien entendu personnelle, doit avoir les propriétés définies au paragraphe 3.8.

4.1. Objets manipulés par un système

Nous considérons qu'un *système est formé d'un ensemble d'objets* et que seule l'existence des objets et des relations qui les lient a une importance. En particulier, la sémantique propre à chacun des objets n'a pas d'intérêt du point de vue du système.

Considérons par exemple une pile ; nous pouvons définir un objet de type "pile" à l'aide des deux opérations 'DEPOSER' et 'RETIRER' (une information) ; cependant la réalisation physique (pointeurs, informations de controle, etc ..) d'un objet de type pile n'a pas d'importance pour l'utilisateur de 'DEPOSER' et 'RETIRER'. Un objet est donc défini par son *type*, c'est-à-dire par l'ensemble des valeurs qu'il peut prendre ainsi que par les opérations qui peuvent être effectuées sur cet objet. Nous appelerons l'ensemble des opérations possibles sur un objet le *mode* de cet objet. L'objet est aussi défini par une seconde caractéristique qui est sa *valeur*, qui est le contenu de l'objet ; ce contenu ne peut être manipulé que par l'intermédiaire des opérateurs du mode de l'objet. La valeur qùi n'a aucune signification du point de vue du système, est tout simplement la représentation interne de l'objet.

Considérons la figure 5 où nous avons représenté deux objets : UC et MC. Considérons l'objet MC, c'est une mémoire centrale et les deux opérations possibles sont :

valeur ← LIRE(adresse)

et

ECRIRE(adresse,valeur).

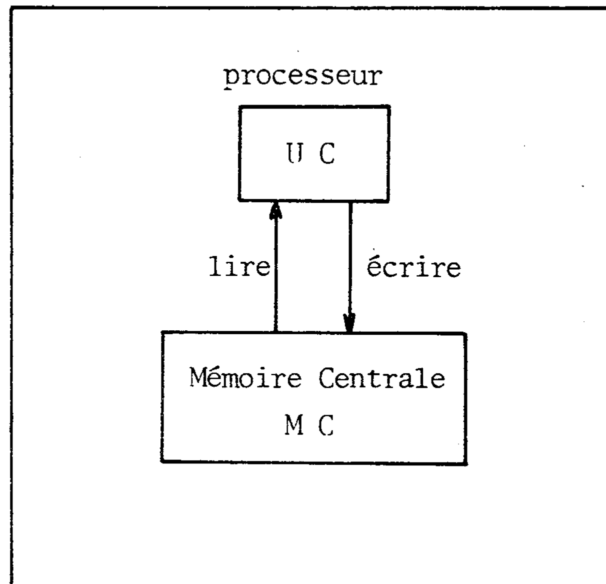


Figure 5

La valeur de l'objet MC est définie par son contenu interne et cette valeur peut seulement être accédée par une des deux opérations possibles. L'exécution de LIRE(100) fournit le contenu de l'adresse 100 et cela même si la mémoire est réalisée physiquement par des techniques d'entrelacement, c'est-à-dire que les adresses paires et impaires sont dans des blocs physiques différents. Un autre type d'organisation qui existe sur certaines machines, consiste à réaliser dans une certaine technologie certaines adresses de la mémoire centrale et dans une autre technologie les autres adresses. Pour des raisons de compromis économiques, on peut changer la répartition des adresses entre les deux technologies, sans rien changer au monde extérieur, c'est-à-dire que les communications avec la mémoire MC se font toujours par LIRE ou par ECRIRE. Du point de vue de l'utilisateur ceci se comprend facilement car si il a rangé la valeur k à l'adresse i, il veut que LIRE(i) retourne k quels que soient l'endroit et la façon dont cette valeur a été sauvegardée.

Comme un objet peut être accédé simultanément par plusieurs utilisateurs, il faut prévoir une nouvelle caractéristique de cet objet, c'est le *contrôleur*. Le noyau doit en effet pouvoir choisir, en cas de conflit, quel est l'accès (ou les accès) qui sera autorisé. Considérons la figure 6 qui représente une mémoire centrale à laquelle sont connectés trois processeurs différents (P1, P2 et P3). Supposons que P1 et P2 essaient d'accéder simultanément à la mémoire pour le mot 100 respectivement en lecture et en écriture et que P3 essaie de lire le mot 101.

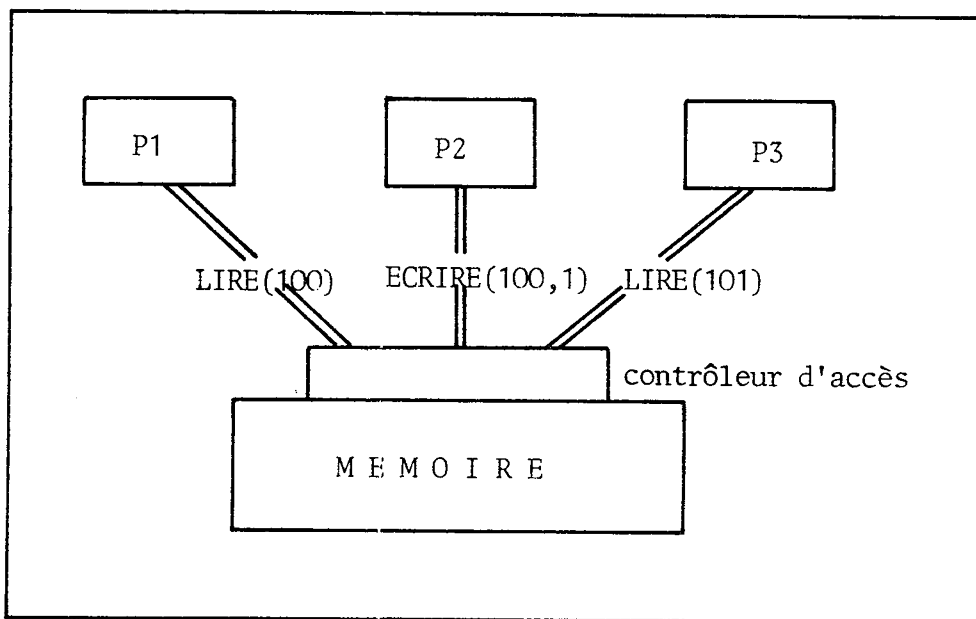


Figure 6 : Synchronisation sur l'accès à un objet

Cet exemple suppose qu'il y ait un mécanisme de synchronisation sur l'accès à la mémoire, c'est le contrôleur d'accès qui définit les priorités, i.e. il choisit l'ordre dans lequel ces accès doivent être effectués sans qu'aucun des processeurs P1, P2 ou P3 n'en soit conscient. Tout se passe du point de vue de l'accès comme si chacun de ces processeurs était seul branché sur la mémoire.

Si maintenant nous reprenons l'exemple précédent en y incluant un entrelacement entre les adresses paires et impaires, le contrôleur permettra l'accès simultané aux mots 100 et 101 par des processeurs différents, mais devra effectuer une synchronisation entre P1 et P2 pour l'accès au mot 100. Il est important de remarquer que ce point de vue impose que la mémoire soit un processeur comme un autre.

Nous voyons ici une distinction qui est effectuée très souvent entre les *propriétés intrinsèques* d'un objet et ses *propriétés technologiques*. Les propriétés intrinsèques concernent ce qui intéresse l'utilisateur de l'objet et les propriétés technologiques ce qui intéresse la réalisation des objets. Si nous revenons au problème de la relativité entre le système et les sous-systèmes, entre un sous-système et les utilisateurs, .., les propriétés intrinsèques sont définies au niveau de l'utilisateur concerné (sous-système, utilisateur) et peuvent être réalisées par des propriétés technologiques au niveau inférieur (respectivement système et sous-système). La figure 7 représente de façon schématique la définition d'un objet.

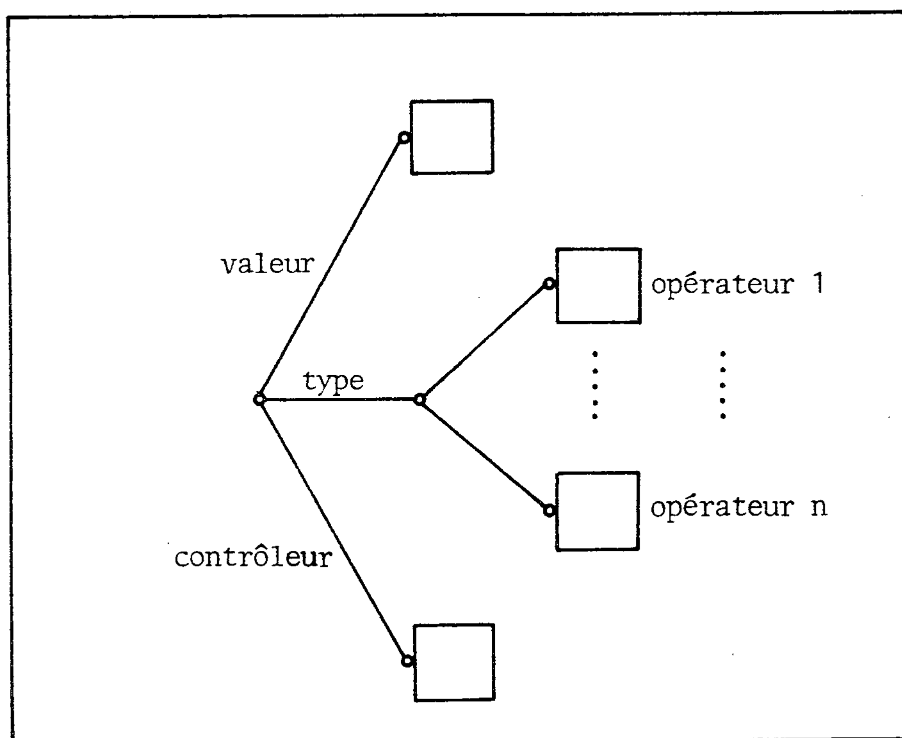


Figure 7 : Composants d'un objet

Parmi les exemples d'objets manipulés par un sous-système, on trouve les annuaires ou bibliothèques d'objets, les procédures, les compilateurs, les périphériques d'entrées-sorties, les fonctions des sous-systèmes.

La protection sur un objet peut être facilement réalisée en supprimant (temporairement) l'utilisation de certains opérateurs du mode de cet objet.

Nous appelons l'ensemble de tous les objets existants dans un système, *l'espace des objets* ⁎. Le noyau est responsable de la conservation de ces objets et des moyens de retrouver ces objets par une certaine identification. C'est ce problème que nous allons traiter dès maintenant.

4.2. Contextes d'adressage

Il peut exister, ainsi que nous venons de le voir, plusieurs utilisations indépendantes et simultanées d'un objet par l'intermédiaire des opérateurs de son mode. Nous appelons *processus* une de ces activités indépendantes. Un processus est caractérisé à un instant donné par l'exécution d'une procédure, cette exécution étant effectuée par un processeur. Nous allons d'abord nous intéresser aux problèmes d'adressage (désignation) des objets par un processus.

Considérons pour cela le programme ALGOL 60 donné par la figure 8 et plus particulièrement la procédure P1. Cette procédure utilise plusieurs variables :

- . locale à la procédure X (3)
- . globale à la procédure Y (2)
- . paramètre formel A

⁎ Dans un certain nombre d'articles nous avions utilisé le terme *"espace d'adressage"* pour désigner la structuration de l'espace des objets ; nous préférons ne plus utiliser ce terme à cause des confusions possibles avec une acceptation de plus en plus générale qui a une signification similaire à celle de l'espace d'exécution.

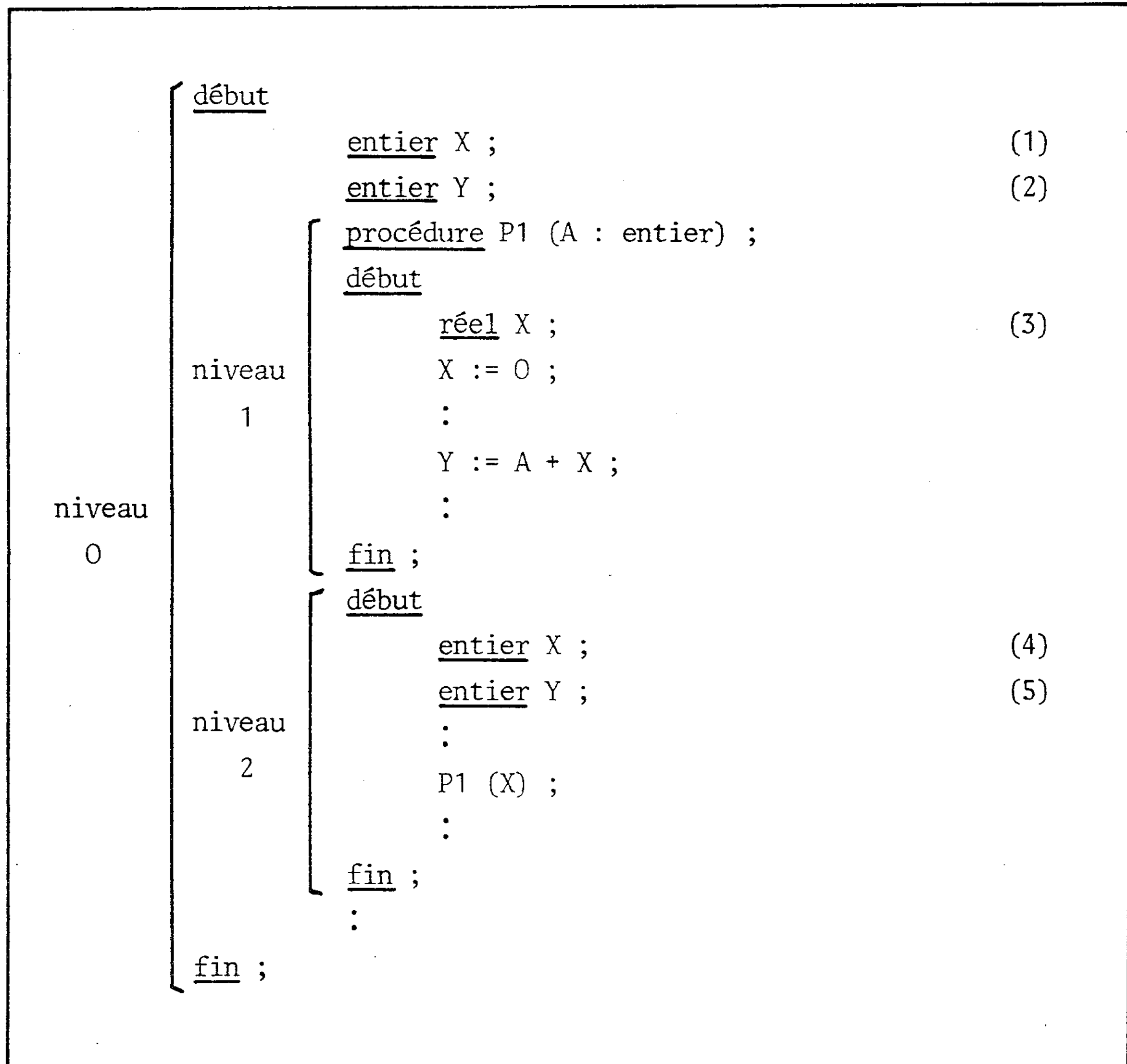


Figure 8 : Contextes d'exécution et d'appel

Lors d'un appel de procédure, le paramètre formel est remplacé par le paramètre réel, puis l'exécution peut se poursuivre. Nous appelons *état d'exécution* l'ensemble des informations nécessaires à l'exécution de la procédure, ce qui inclut par exemple l'adresse de l'instruction en cours. En ALGOL 60 les appels de procédures peuvent être récursifs, ce qui signifie que quand P1 appelle P1 (depuis l'intérieur de lui-même), un nouvel *état d'exécution* est créé ; cet état possède une nouvelle variable (X) locale à la procédure ; cette nouvelle activation de variable est différente de l'activation précédente. Un nouvel emplacement mémoire est affecté à X. L'ensemble des *états d'exécution* constitue *l'état d'un processus*. Les états d'exécution sont créés par des appels de procédures et détruits au retour de ces procédures.

Si nous considérons l'exemple précédent (figure 8) nous nous apercevons qu'il existe plusieurs variables de nom X. Nous allons examiner maintenant le problème qui consiste à déterminer quelle est la variable X utilisée dans les différentes instructions qui utilisent le nom X. Considérons l'appel à P1, la variable considérée est l'entier *numéroté 4*, tandis que dans le corps de la procédure P1 la variable de même nom est le nombre réel *numéroté 3*. De même pour la variable Y, il s'agit au moment de l'appel, de l'entier *numéroté 5* et, lors de l'exécution, de l'entier *numéroté 2*. Cet exemple montre l'existence de *contextes de noms* qui ne référencent pas la même variable (ou objet). Le contexte d'appel (ici les variables 4 et 5) est utilisé pour résoudre le nom des paramètres réels, et le contexte d'exécution de la procédure (ici les variables 2 et 3) est utilisé pour résoudre les noms locaux et globaux. Le contexte des noms fait aussi partie de l'état d'exécution.

Cet exemple montre le caractère dynamique d'un processus, par exemple le contexte des noms évolue dans le temps en fonction des appels de procédure. La figure 9 donne un schéma plus complet.

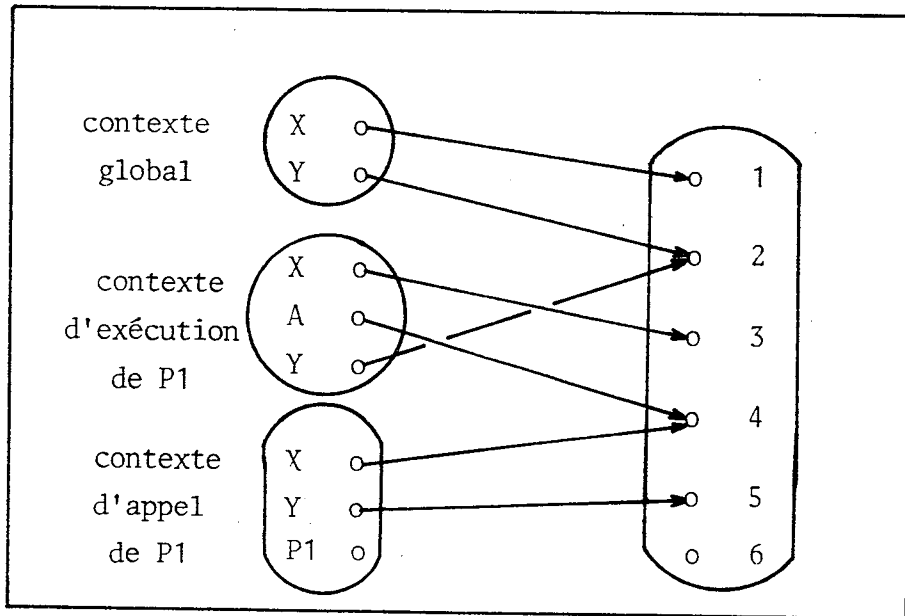


Figure 9 : Différents contextes d'adressage

Il est important de voir que le *nom* d'un objet est le moyen de retrouver cet objet, le rôle du noyau étant justement d'assurer la correspondance entre les noms et les objets. Un nom est donc un symbole dont la signification varie en fonction du contexte.

L'exemple que nous venons de traiter montre en effet l'existence de conflits sur les noms, i.e. que le même nom peut référencer différents objets selon le contexte d'exécution. Dans notre explication, pour spécifier de façon plus précise de quels objets nous parlions, nous avons utilisé la terminologie *"objet numéroté i"*. Ce problème, qui s'est posé lors de l'explication, se retrouve au niveau du système. Nous appelons ce nombre *le nom universel* de l'objet et pour n'avoir aucun conflit sur ce nom, *ce nom doit être unique,* c'est-à-dire qu'il est créé une fois pour toutes quand un objet est créé et n'est jamais réutilisé même après la destruction de l'objet. Ce nom est logique et n'est pas lié à la position physique de l'objet, en particulier, le noyau peut déplacer, pour des raisons de performances, l'objet dans l'espace réel à condition de maintenir la correspondance entre le nom universel et l'objet.

La figure 10 schématise les différents types de noms dans le cas de l'exemple donné par la figure 8.

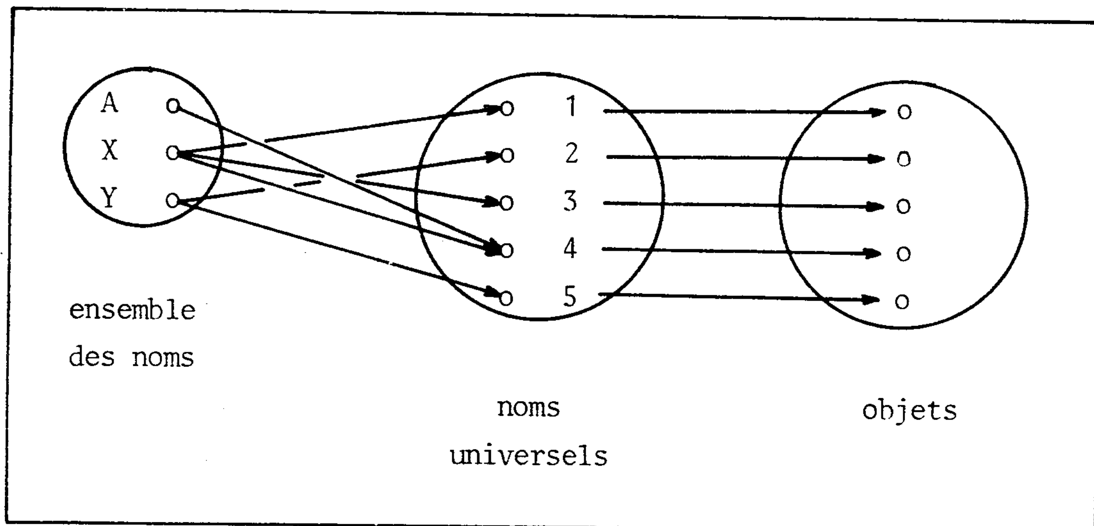


Figure 10 : Noms universels

4.3. Relations de désignation

Nous sommes donc en présence de deux grandes classes d'entités de nature différente : ce sont les objets et les processus. Les processus sont des entités actives qui ont besoin d'un certain nombre d'objets* pour pouvoir s'exécuter. Cependant pour pouvoir utiliser ces objets il faut un certain moyen de désigner ces objets, ceci est fait au moyen d'un *nom*. L'exemple ALGOL 60 précédent a montré la nécessité de tels noms et nous avons découvert l'existence de deux catégories de noms différents. Un *nom* est un *symbole,* ce symbole est en général interprété (par rapport à un certain contexte) et fournit un autre nom (i.e. interprétable par rapport à un autre contexte). Ainsi, dans l'exemple en ALGOL 60, nous avons vu qu'on établit une correspondance entre un nom et un nom universel.

* Ces objets sont souvent appelés "ressources" dans certains systèmes.

Le nom universel est, probablement, interprété lui-même par le système pour donner une adresse (par exemple sur un périphérique) ; cette adresse est elle-même un nom interprété par le matériel pour obtenir l'information elle-même à un certain emplacement physique. Cet exemple montre bien la relativité de la signification d'un nom.

Pour reprendre l'exemple ALGOL 60, à un instant donné l'état d'exécution d'un processus comporte l'ensemble des objets manipulables par ce processus : ce sont les paramètres effectifs de la procédure, les variables locales à cette procédure et les variables globales. Cet ensemble d'objets manipulables ne représente qu'un sur-ensemble des objets effectivement utilisés par l'exécution.

Nous n'avons vu pour l'instant que les problèmes de noms liés à l'existence d'un seul processus, ce problème est le même pour tous. Nous appelons *espace d'exécution courant* d'un processus l'ensemble de tous les noms d'objets accessibles à un instant donné par le processus ; nous appelons *nom local* le nom d'un objet par rapport à l'espace d'exécution d'un processus.

Pour beaucoup de systèmes, les objets de type fichier peuvent être accédés par tous les processus qui connaissent leurs noms ; nous voulons au contraire donner des règles qui renforcent la portée des noms, c'est-à-dire qu'un processus ne puisse accéder que ce qui lui est nécessaire. De plus, dans l'exemple du fichier, les règles d'adressage ne sont pas les mêmes pour les fichiers que pour les autres types d'objets (procédure par exemple), or il ne faut pas oublier que notre objectif est une uniformisation du concept d'objet et de sa désignation.

Les mêmes objets peuvent aussi avoir des noms locaux différents dans les différents espaces d'exécution. L'exemple ALGOL montre aussi que le contexte d'adressage évolue dans le temps, donc un processus est composé de plusieurs espaces d'exécution.

De plus, on peut facilement réaliser la protection en l'associant au nom local d'un objet, ce qui permet de résoudre le besoin de l'utilisateur qui veut plusieurs protections différentes sur le même objet.

Nous avons beaucoup parlé de noms d'objets, il faut maintenant se pencher sur la structure de l'espace des objets.

4.4. Structure de l'espace des objets

Un processus ne peut accéder qu'aux objets qu'il peut nommer, c'est-à-dire qui possèdent un nom local dans son expace d'exécution. Certains de ces objets peuvent représenter des collections d'objets, nous les appelons alors *annuaires* ; chaque objet possède une identification dans un annuaire, identification appelée *nom d'entrée*. A un instant donné le nom par lequel un processus peut désigner un objet est obligatoirement un nom local éventuellement qualifié par l'ensemble des noms d'entrée dans les différents annuaires permettant de l'atteindre. Ce nom est appelé *nom partiel*.

La structure de l'espace des objets doit être du type de celle définie pour les sous-systèmes par la figure 4. La règle essentielle est :

qu'un processus ne peut utiliser que des objets qu'il peut nommer et réciproquement.

Cette structure de l'espace des objets a une existence indépendamment des processus. Il existe donc un annuaire privilégié appelé la *racine*.

A un instant donné, pour un processus donné, l'espace des objets est divisé en deux : ceux qu'il peut utiliser (c'est-à-dire qui sont désignables) et ceux qu'il ne peut utiliser.

Un processus étant une entité dynamique (cf. exemple ALGOL de la figure 8) avec un contexte qui évolue dans le temps, est représenté par un ensemble d'espaces d'exécution dont l'un est l'espace courant. L'évolution de cet espace ne peut s'effectuer indépendamment d'une évolution dans l'espace des objets ; en effet, le changement de contexte doit traduire une modification de l'ensemble des objets accessibles. La structure naturelle pour cet espace est une structure de graphe telle qu'elle a été définie au § 3.6.

Si nous considérons une procédure, on distingue au moins deux ensembles d'objets accessibles par cette procédure : les paramètres et les objets non locaux à la procédure, c'est-à-dire ceux dont le nom n'est connu que dans le contexte de l'exécution. Pour cela il faut affecter à une procédure un annuaire d'exécution, cet annuaire s'appelle *l'environnement de la procédure*. Lors du changement d'espace d'exécution, il faut associer l'environnement correspondant à la procédure, il existe donc un objet particulier dans un espace d'exécution appelé *l'environnement* du processus. Un *espace d'exécution* est donc composé des paramètres effectifs de la procédure en exécution et de l'environnement associé.

La signification d'un nom ne doit pas changer pendant l'interprétation par des procédures indépendantes, il faut donc pouvoir associer un nom à un objet pour une durée déterminée. En effet, pour traduire la propriété de substitution dynamique, il faut définir à quels instants cette substitution peut être effectuée et aussi à quel instant cette substitution devient effective.

Considérons une version d'un compilateur que l'on veut remplacer dynamiquement par une nouvelle version ; il faut que le concepteur puisse effectuer la substitution dynamiquement sans savoir si des utilisateurs utilisent effectivement le compilateur. Du point de vue de l'utilisateur du compilateur, il n'est pas concevable d'avoir un remplacement de l'ancienne version, durant son exécution, par la nouvelle version. Il faut donc un mécanisme qui préserve les exécutions en cours de l'objet (compilateur) et qui assure que toute nouvelle exécution prenne le nouvel objet, i.e. celui qui a été substitué. Le problème est d'ailleurs identique pour les destructions d'objets.

Une solution à ce problème peut être trouvée facilement en ayant des noms différents au niveau de l'utilisation et de la conservation. Une façon naturelle consiste à utiliser les possibilités offertes par l'existence des espaces d'exécution et d'objet, en prenant comme *nom d'utilisation un nom local et comme nom de conservation un nom partiel*. Pour cela, il faut pouvoir affecter un nom local à un objet, l'opération qui permet cette affectation est appelée la *liaison*.

D'où la règle simple :

la substitution ou la destruction d'un objet est effectuée dans l'espace des objets et n'affecte aucunement les objets déjà liés à un espace d'exécution.

Les objets substitués sont de nouveaux objets qui possèdent les mêmes noms partiels dans l'espace des objets. De même qu'il existe une opération de liaison, il existe une opération inverse appelée *déliaison*.

A la différence de nombreux autres systèmes, nous avons voulu utiliser au maximum la structure arborescente de l'espace des objets. En particulier, un sous-système peut être décrit par un annuaire, les utilisateurs de ce sous-système par d'autres annuaires. L'activité d'un annuaire est représentée par un ou plusieurs processus définis par une pile d'espace d'exécution, chacun de ces processus évoluant dans l'espace des objets pouvant par exemple appeler des procédures du sous-système concerné (ou même d'un autre sous-système), la structure arborescente permettant de contrôler ces évolutions et de protéger les différents sous-systèmes.

4.5. Existence de mécanismes semblables dans les systèmes existants

Nous allons montrer sur un exemple simple que les différents aspects que nous venons de développer existent déjà, sous une forme plus ou moins complète, à l'heure actuelle.

Considérons un système dans lequel on définit une mémoire virtuelle (de type classique) associée à un processus et où l'appel d'une fonction fournie par le système (par exemple demande de service de type : accès à un fichier) s'effectue à l'aide d'une instruction particulière appelée "appel moniteur" qui provoque un déroutement.

A ce moment l'espace d'exécution du processus change, en particulier au niveau de l'adressage qui passe très souvent en mode réel (i.e. la totalité de la mémoire centrale est alors accessible) et du mode de fonctionnement privilégié (les instructions spéciales telles que lancement d'entrées-sorties deviennent possibles). La figure 11 schématise l'état du processus à l'instant t_1 (i.e. avant l'appel au moniteur) et à l'instant t_2 (i.e. après l'appel au moniteur).

Les procédures du système exécutées après l'exécution de l'appel au moniteur ont des possibilités d'adressage différentes de celles de la procédure ayant effectué l'appel (instant t_1) ; elles peuvent notamment modifier l'état d'exécution de l'instant t_1, par exemple si l'appel au moniteur au lieu d'être explicite avait été implicite, comme dans le cas d'une "faute de page". Après résolution de la faute de page, en particulier modification dans l'espace d'exécution de t_1 de la table des pages associées, il y a retour à la procédure interrompue. On a ici une sauvegarde suivie d'une restauration de l'espace d'exécution, c'est un appel procédural.

Considérons maintenant le cas où l'appel au moniteur consiste à demander l'exécution d'un nouvel objet (i.e. de fabriquer une nouvelle table de pages pour l'exécution d'un objet différent de la procédure appelante), le retour de l'appel moniteur s'effectue seulement après fabrication d'un nouvel espace d'exécution, éventuellement différent de celui de l'instant t_1. Dans ce cas, plusieurs solutions sont possibles :

. sauvegarde de l'espace d'exécution t_1 et restauration lors de la fin de l'exécution de la nouvelle procédure : c'est le cas classique des appels procéduraux, on utilise alors (au niveau du noyau (ou moniteur)), une pile des espaces d'exécution ;

. remplacement de l'espace d'exécution t_1 par le nouvel espace d'exécution : c'est le cas d'un branchement, il n'est alors plus possible de restaurer plus tard cet espace du temps t_1 ;

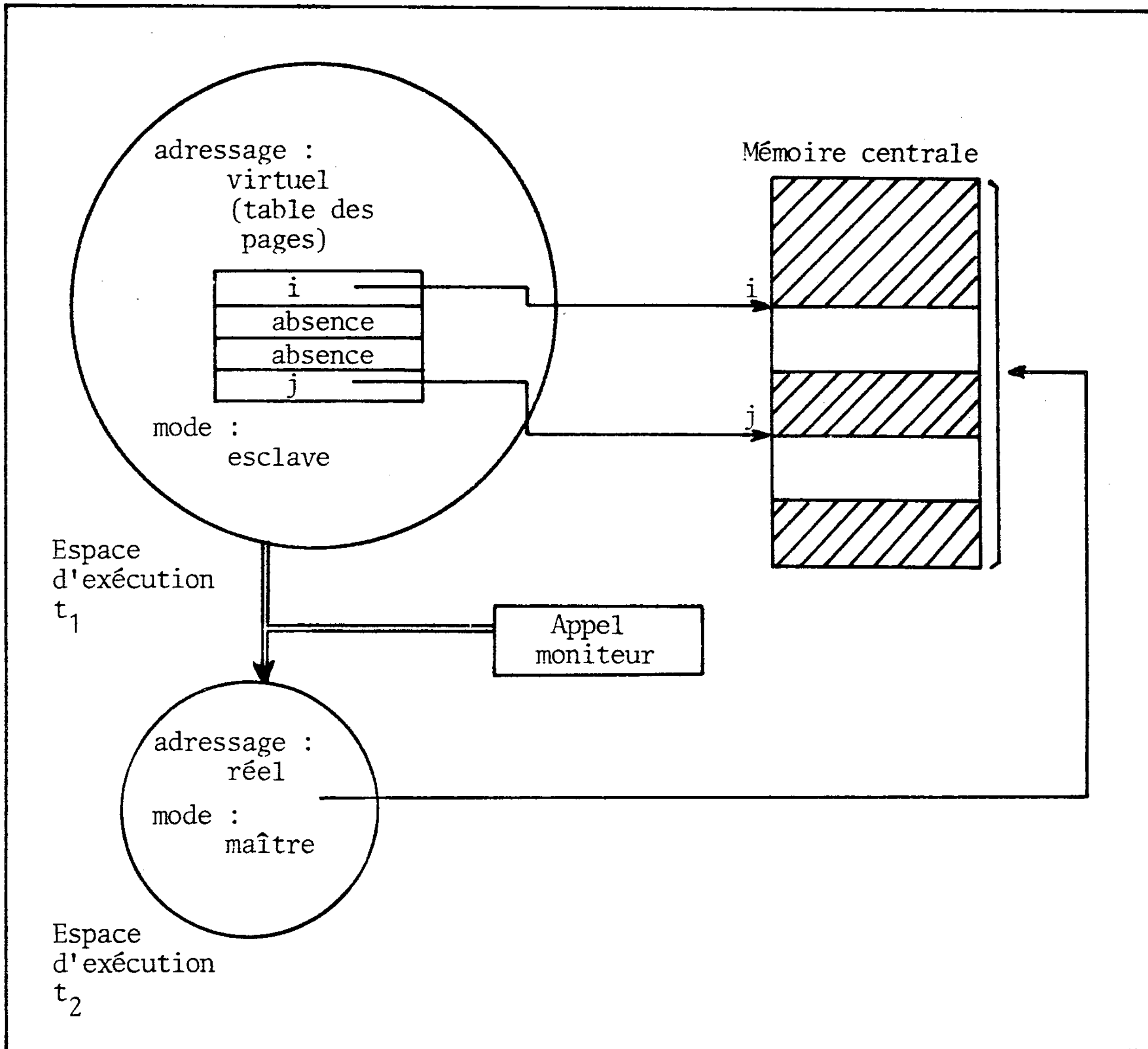


Figure 11 : Transition d'espace d'exécution

. création d'un processus auquel on associe le nouvel espace d'exécution et retour dans le cas du processus appelant à l'espace d'exécution du temps t_1 : c'est le cas de la création d'une nouvelle exécution qui se déroule alors en parallèle avec l'ancienne.

On voit sur cet exemple qu'il y a un choix à faire quant à l'évolution des espaces d'exécution d'un processus. De façon générale, on peut dire qu'un changement d'espace d'exécution n'a lieu que sur appel explicite d'une procédure couramment active dans un espace d'exécution.

DEUXIEME PARTIE

G E M A U

5. OBJECTIFS DE GEMAU

5.1. Objectifs pratiques

Nous allons maintenant essayer de définir les objectifs pratiques que nous nous sommes fixés dans le projet GEMAU.

Nous voulons en effet qu'un système possède les propriétés évoquées dans ce chapitre, et définir un exemple de noyau qui possède ces propriétés. Pour cela nous n'avons pas voulu que cette étude reste une étude formelle, nous avons voulu la réaliser et utiliser cette réalisation pour écrire un certain nombre de sous-systèmes de façon à valider les idées avancées. Cet objectif de réalisation a imposé de ne pouvoir aller jusqu'au fond de certains mécanismes, par exemple la notion de mode et spécialement de mode construit. Nous essaierons dans la conclusion de cet ouvrage de déterminer les limitations et de montrer les extensions possibles au noyau. Une étude d'un système de façon plus formelle et qui pousse à fond l'idée de mode, est menée principalement par Ph. Darondeau [DA 74 b].

Nous avons mené l'étude en nous attaquant aux points suivants :

a. définition des objets élémentaires : type, protection, ..

b. définition des opérateurs de manipulation de ces objets,

c. définition de la structure de l'espace des objets, ainsi que des règles de création et de destruction d'objets,

ces trois points sont en fait la définition d'un langage qui est le langage de commande du noyau [BR 74], [DA 74 c],

d. définition d'une méthodologie pour la construction des systèmes [DA 74 a], c'est-à-dire :

. comment décrire les spécifications internes d'un système ?

. comment être sûr que ces spécifications sont correctes et qu'elles sont réalisées sans distorsion ?

. comment fournir des outils pratiques d'aide à la mise au point ?

5.2. Objectifs théoriques

L'objet de cette étude est de montrer (sinon de démontrer) que

> les propriétés de protection, partage, substitution et synchronisation peuvent être résolues *facilement et efficacement* par la définition d'un schéma d'adressage approprié.

Pour cela nous allons donner l'exemple des espaces d'exécution et d'objets fournis par le noyau et montrer ensuite comment le noyau de GEMAU peut être utilisé ; de plus, on se reportera utilement à la thèse de P. Laforgue [LA 75] et à [IN 74] pour une description détaillée de plusieurs sous-systèmes écrits en utilisant le noyau GEMAU.

Chapitre 2

ESPACES D'EXECUTION ET D'OBJETS DE GEMAU

Nous définissons plus précisément les objets manipulés dans GEMAU, leurs relations réciproques et l'adressage de ces objets.

Les propriétés liées aux objets (synchronisation, type, environnement) et celles liées aux noms (protection) sont développées, ainsi que les notions fondamentales d'environnement, de sous-systèmes et de liens entre sous-systèmes.

Ce chapitre se termine par une étude des mécanismes liés à la cohérence et à la disponibilité des sous-systèmes.

1. CONVENTIONS

Nous définissons les diverses notations employées tout au long de ce chapitre. Nous désignons :

- . les noms locaux par LNi
- . les registres généraux (pour les valeurs immédiates) par RGi*
- . les objets par des identificateurs classiques
- . les chaînes de caractères entre apostrophes (')
- . les noms universels par un nombre entier positif, sans que les séquences de nombres aient une signification d'un exemple à l'autre.

Les définitions des objets, des primitives, etc .., seront données par une notation très simple de style BNF (Backus Naur Form).

Le caractère / ('divisé par') indique l'alternance des possibilités.

Les crochets suivis de trois points ([ ]...) indiquant la présence optionnelle et la répétition éventuelle (0 ou plusieurs fois).

---

* Ceci est un artifice de notation car une valeur immédiate peut être considérée comme un objet de type particulier et donc être référencée par un nom local.

2. ESPACE D'OBJETS, ESPACE D'EXECUTION

2.1. Espace des objets

2.1.1. Rappels et définitions

Une structure arborescente permet de réaliser simplement les fonctions de désignation, de partage et de protection que nous avons vues précédemment.

Nous appelons *annuaire* les noeuds (objets) non terminaux de l'arborescence. Un annuaire est le descriptif d'un ensemble d'objets qui lui sont directement rattachés.

Un *segment* est un objet de type donnée ou programme et un *périphérique* est un objet de type périphérique ou entrée-sortie.

Un annuaire permet de nommer symboliquement des objets qui peuvent être des segments, des annuaires ou d'autres types à définir.

Un objet possède donc un *nom d'entrée* dans un annuaire. Ce nom permet de désigner de façon *unique* l'objet de cet annuaire.

Nous appelons *nom partiel* d'un objet par rapport à un annuaire, la liste de tous les noms d'entrée des objets situés sur le chemin issu de cet annuaire et menant à l'objet. Il existe un annuaire prédéfini appelé *racine*.

Chaque objet est défini dans un annuaire (exception faite de la racine) par un *descripteur* qui contient les caractéristiques dudit objet. On y trouve notamment :

. le nom d'entrée,

. la protection,

. le type de l'objet : annuaire, segment, périphérique,

. le nom universel de l'objet.

Le contrôle de l'espace des objets, et en particulier la définition des noms partiels connus, est réalisé par le mécanisme *d'environnement*. La racine est l'environnement initial. Un environnement est un annuaire distingué. Une procédure est située dans un environnement donné si dans son nom partiel à partir de cet environnement ne figurent pas d'autres environnements. S'il en existe, ceux-ci sont dits *englobés*, le premier étant dit *englobant*.

Un environnement fournit à une procédure y étant située, un sous-ensemble de l'espace d'adressage. Tout nom émis par la procédure est résolu comme un nom partiel à partir de cet environnement ; l'ensemble des objets nommables à partir d'un environnement définit son *espace d'objets visible*.

2.1.2. Exemple de structure de noms

Considérons la figure 1. L'objet 1 est la racine ; 1, 4 et 6 sont des environnements.

Les noms partiels de l'objet 8 sont respectivement : C.B.A, B.A et A dans ces environnements (1, 4, 6).

L'objet 5 a pour noms, respectivement C.A et A dans 1 et 4 mais n'est pas nommable par une procédure située dans l'environnement 6.

Nous avons les ensembles d'objets visibles suivants :

Ev (1) = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11 }

Ev (4) = {5, 6, 7, 8, 9, 10, 11}

Ev (6) = {8, 9 }

Nous voyons qu'il est possible de nommer les mêmes objets depuis des environnements différents, s'ils sont visibles depuis ces environnements, c'est cette propriété qui permet le partage.

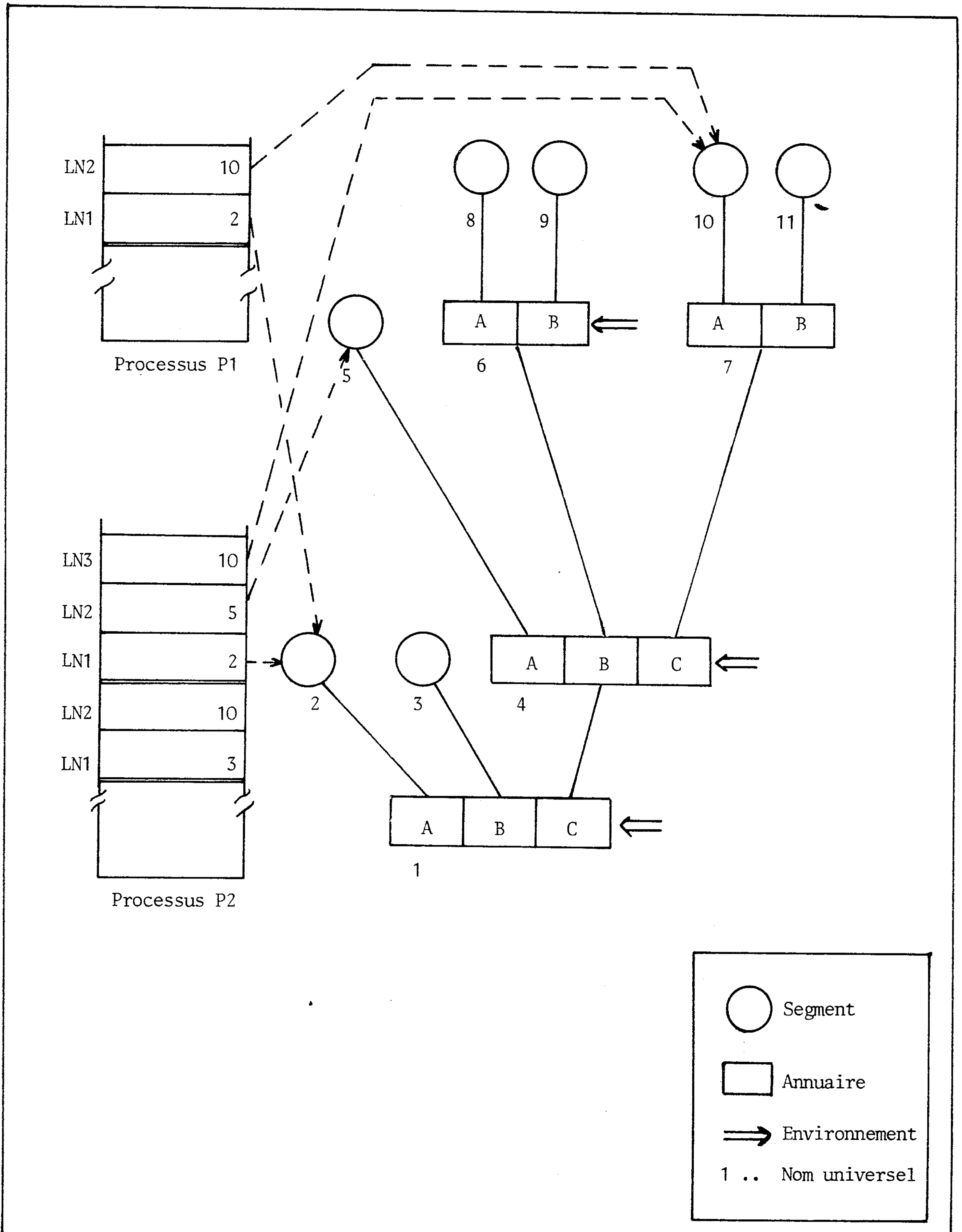


Figure 1 : Structure des noms

2.2. Espace d'exécution

2.2.1. Définitions et rappels

Une procédure est parfaitement définie par ses *noms locaux* dont l'un d'eux (LNO) représente son *environnement* (courant).

Les noms locaux peuvent se référer à des objets provenant éventuellement d'autres environnements que l'environnement courant.

L'espace visible d'un environnement est l'ensemble des objets adressables à partir de l'environnement.

Nous appelons *primitives* les opérateurs qui permettent de manipuler les objets relativement à un espace de noms (exécution ou objet). A tout instant une procédure possède un ensemble de primitives possibles, cet ensemble s'appelle les *prérogatives* de la procédure. Il paraît naturel de lier les prérogatives d'une procédure à l'environnement qui définit les objets manipulables.

Un processus est réalisé par une *pile de contrôle* qui contient les espaces d'exécution de ce processus. Ce schéma est inspiré de Evans et Leclerc [EV 67], c'est-à-dire que toute communication se fait par passage de paramètres, y compris l'environnement qui est un paramètre implicite. Le sommet de la pile constitue l'espace d'exécution courant du processus.

L'environnement de la procédure en cours d'exécution est appelé *environnement courant* du processus (ε_c dans les figures, c'est le nom local LNO).

La figure 1 donne un exemple d'espace d'exécution. Remarquons dans cet exemple, que l'état courant du processus P2 peut être le résultat de l'exécution dans l'objet 3 de :

CALL(LNO.A,LNO.C.A,LNO.C.C.A)

ou

CALL(LNO.A,LNO.C.A,LN2)

La création d'un processus exige la fourniture d'une procédure initiale, de ses arguments et d'un environnement de référence. C'est celui du processus père au moment de l'ordre de création du fils.

L'espace d'exécution du fils est construit à partir de l'environnement courant du père. Ceci garantit qu'un processus créé n'outrepassera jamais les prérogatives du processus créateur.

La fin d'un processus n'est effective qu'au retour de sa procédure initiale.

Pour des raisons de contrôle de processus, nous privilégions la relation de filiation entre processus : un processus père est responsable de ses fils. Toute autre relation entre processus ne peut alors être définie qu'au moyen des objets de l'espace d'objets.

Cette structure hiérarchique exige l'existence a priori d'un processus *ancêtre* prédéfini dont l'environnement initial est la racine de l'espace d'objets.

Dans la figure 2, nous trouvons le processus ancêtre et son environnement (0). Il assure la génération du système à partir des programme et périphérique initiaux.

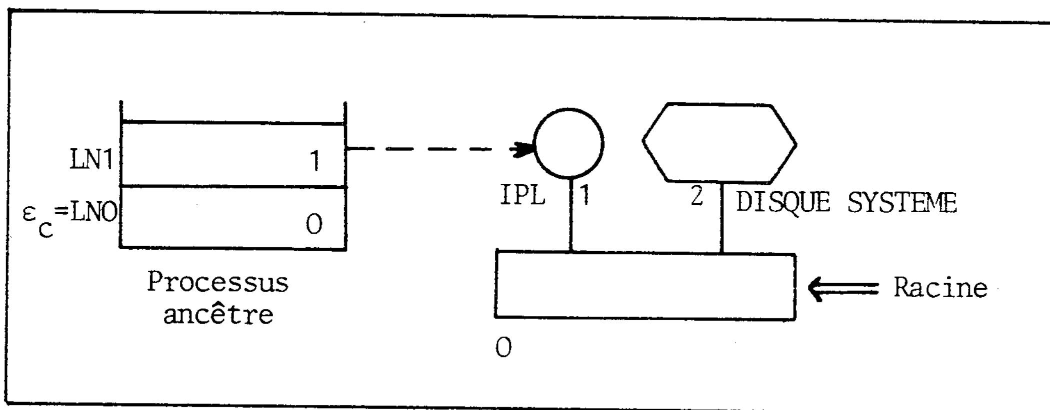


Figure 2 : Environnement initial du processus ancêtre

3. LES OBJETS

3.1. Durée de vie d'un objet

Un objet est défini par un nom d'entrée dans un annuaire, son existence est liée à l'existence de ce nom d'entrée et à celle de l'annuaire.

Cependant, il n'y a aucune raison pour qu'un processus ne puisse utiliser des objets qui n'appartiennent pas à l'espace d'objets et qui, par conséquent, ne sont pas partageables avec d'autres processus ; ce sont des *objets temporaires*.

La destruction physique d'un objet est effectuée quand il n'y a plus aucun nom local qui référence l'objet et que, de plus, cet objet n'a pas (ou plus) de nom d'entrée.

Nous verrons ultérieurement que la primitive de destruction d'un objet effectue seulement une destruction logique, c'est-à-dire détruit un nom d'entrée.

3.2. Modes de désignation d'un objet

Ainsi que nous venons de le voir, il existe un certain nombre de façons de désigner un objet :

. par un nom local (LNi)

. par un nom partiel, c'est-à-dire un nom local suivi d'une notation pointée ; par convention, dans le cas où ce nom local est LNO (environnement courant), nous l'omettons dans les exemples.

On peut ainsi définir :

< désignation simple d'objet >::= <nom local >[< nom d'entrée >]...

Un nom partiel d'un objet (pour un processus) représente la liste de tous les noms d'entrée des annuaires situés sur le chemin issu d'un annuaire d'origine vers l'objet désiré. Cet annuaire d'origine est soit l'environnement courant (LNO) du processus, soit un annuaire désignable par un nom local dans l'espace d'exécution.

La figure 3 définit un processus P ayant pour environnement courant l'annuaire 0. Le nom local LN2 référence l'annuaire 2. Le nom partiel de l'objet 3 peut être : B.A (1)

ou LN2.A (2)

Dans le cas (1) il s'agit d'un nom partiel par rapport à l'annuaire 0 (environnement courant)*, dans le cas (2), d'un nom partiel par rapport à l'annuaire 2.

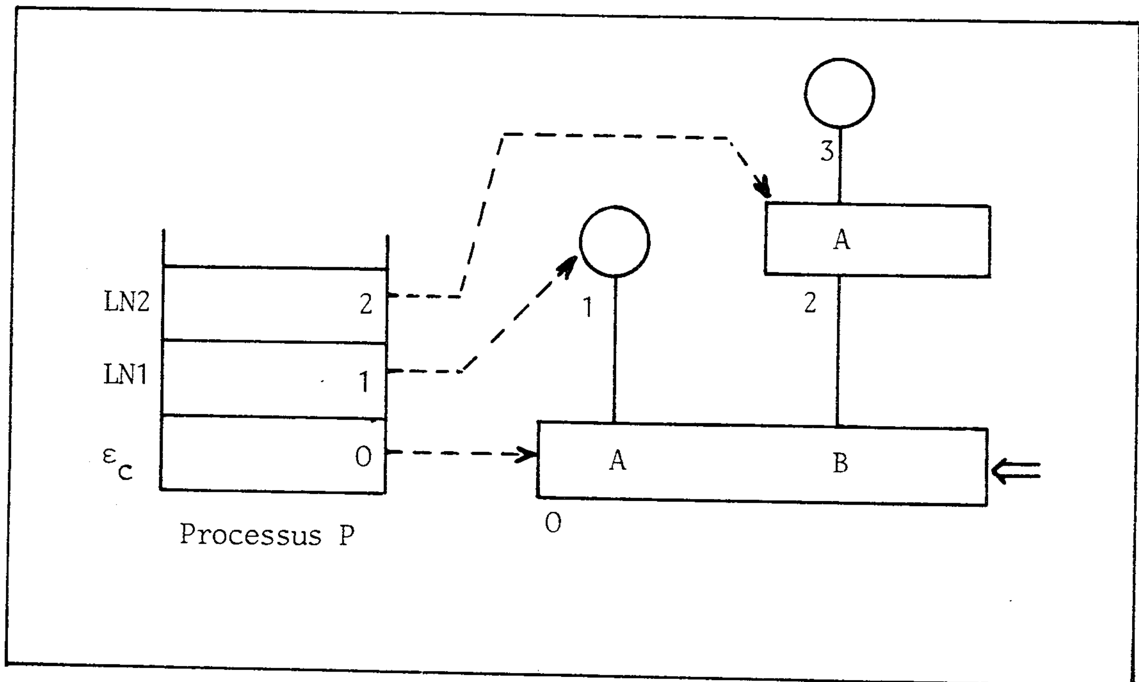


Figure 3 : Nom partiel d'un objet

* Dans ce cas, pour des raisons de lisibilité, nous omettons très souvent le nom local LN0.

Cependant, il existe une troisième façon de désigner un objet, c'est par sa description ; ce sera par exemple le cas de la création d'un objet qui pourra être de tel ou tel type, avoir comme valeur initiale celle d'un autre objet du même type.

De plus, une désignation peut référencer un objet (comme dans les exemples précédents) ou une copie d'un objet. Ceci représente pour les appels de procédures les deux cas classiques de passage par nom et par valeur.

On a ainsi :

< désignation d'objet >::= [*valeur*]< désignation simple d'objet > / < description d'objet >

où <description d'objet > dépend du type de l'objet et sera définie plus en détail pour chacun des types primitifs connus du noyau.

3.3. Opérateurs sur les objets

Le noyau fournit un certain nombre de primitives (opérateurs) permettant de manipuler et d'accéder les éléments de l'espace des objets. Ces primitives peuvent être divisées en deux classes :

. celles des primitives qui permettent de modifier l'espace d'objets (création d'objet, destruction, transfert, ..),

. et celles qui permettent d'établir une correspondance entre les objets et les espaces d'exécution (appels de procédures, liaison, création d'objets temporaires, destruction de noms locaux, ..).

Cette dernière classe de primitives ignore la signification sémantique des objets manipulés, en particulier la structuration interne des segments est inconnue.

4. MODIFICATIONS DE L'ESPACE DES OBJETS

4.1. Création d'un nouvel objet

```
CREATE(< désignation d'annuaire >
      ,< nom d'entrée >
      ,< désignation d'objet >)
```

permet d'entrer un nouvel objet dans l'annuaire désigné par < désignation d'annuaire >, cet objet possède le nom d'entrée indiqué comme second paramètre de la primitive de création. Le type et toutes les caractéristiques du nouvel objet sont définis par la < désignation d'objet >. Cette désignation peut être soit une description, soit définir la copie (*valeur*) d'un objet déjà existant (même type, mais de protections identiques ou différentes, par exemple), soit être simplement un nouveau nom d'entrée pour l'objet indiqué. Ce dernier cas sera complètement ingoré jusqu'au paragraphe 11 du présent chapitre (liens).

4.2. Destruction d'un objet

```
DELETE(< désignation d'objet >)
```

permet de retirer un objet d'un annuaire (destruction d'un nom d'entrée), cependant comme nous l'avons vu précédemment, la destruction physique de l'objet n'intervient que quand il n'y a plus aucun nom local référençant cet objet.

4.3. Modification de la description d'un objet

MODIFY(< désignation d'objet > ,< description d'objet >)

permet de changer en totalité (ou en partie) les caractéristiques contenues dans le descripteur de l'objet désigné.

4.4. Obtention d'informations sur un objet

GETINFO(< désignation d'objet >)

permet d'obtenir les informations contenues dans la description d'un objet. Il est important de noter que l'obtention de ces informations ne permet absolument pas de modifier l'objet ou de l'accéder autrement que par l'accès normal, c'est-à-dire sans faire jouer les protections, changements d'environnements, etc ..

4.5. Exemple

Considérons la figure 4 ; nous allons illustrer l'utilisation des différentes primitives.

Cette figure illustre un espace d'objets dans lequel il y a sept objets et un espace d'exécution d'un processus P avec trois noms locaux. L'environnement courant est l'annuaire 1. Nous supposons aussi qu'aucun nom local de ce processus (ou d'autres processus) ne référence les objets 3, 4 et 5 à l'exception de LN1, LN2 et LN3 de l'espace d'exécution courant du processus P.

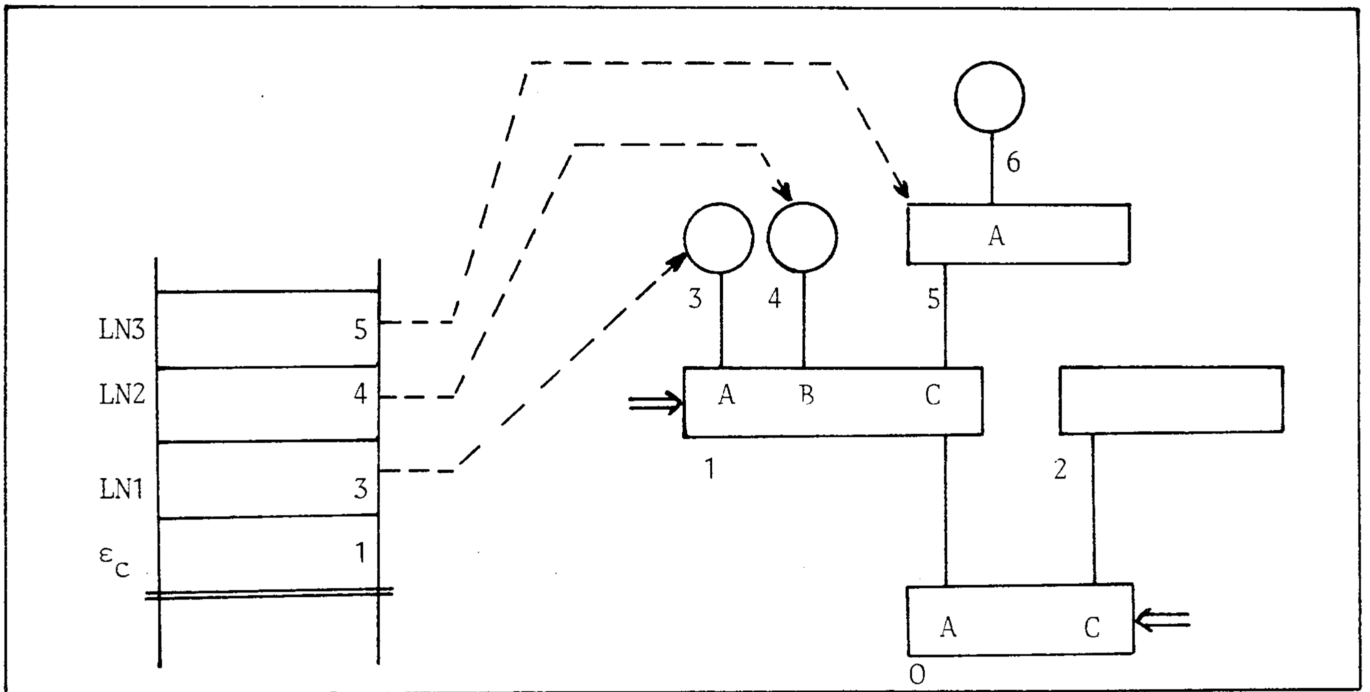


Figure 4.a.

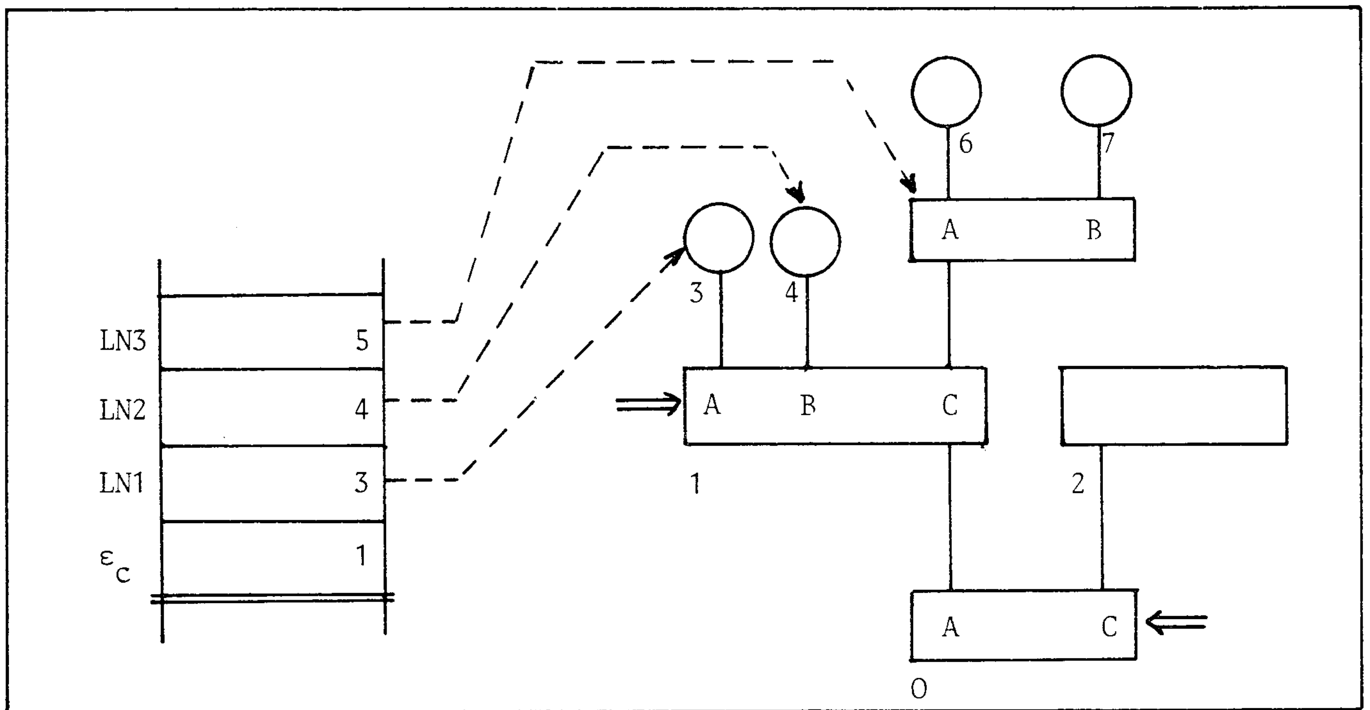


Figure 4.b.

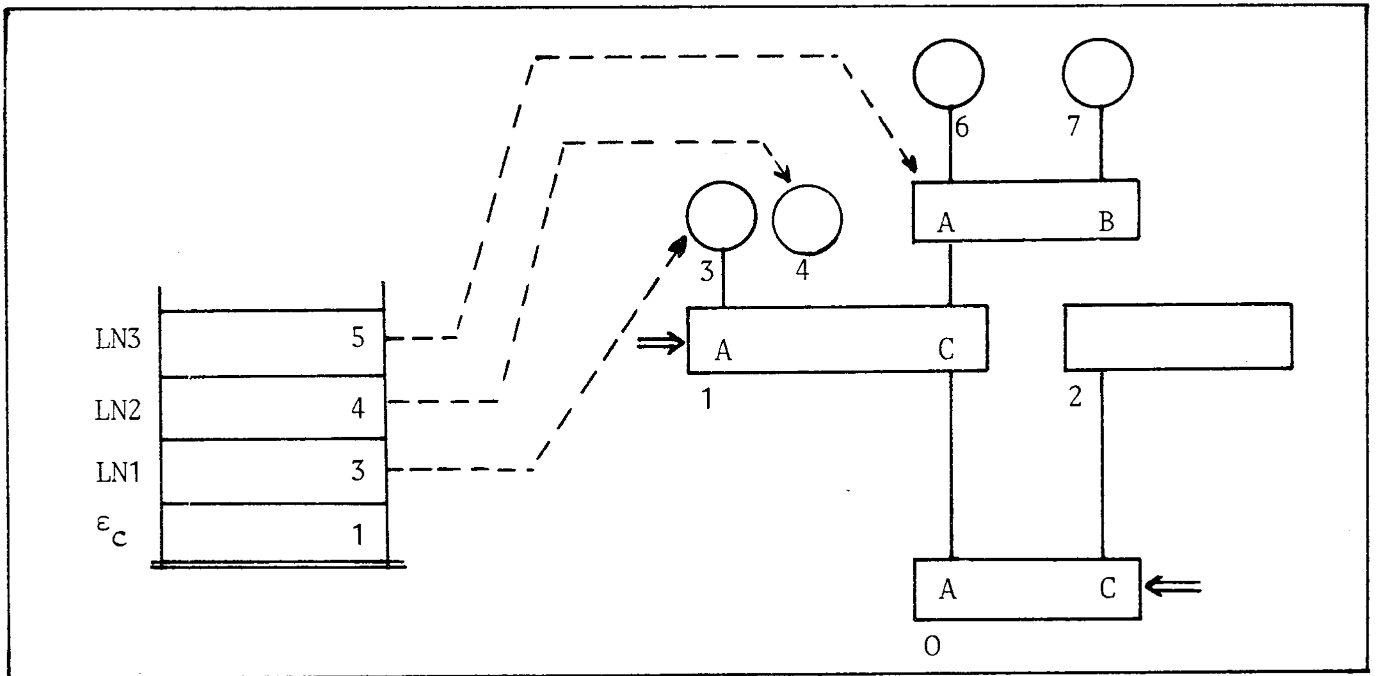


Figure 4.c.

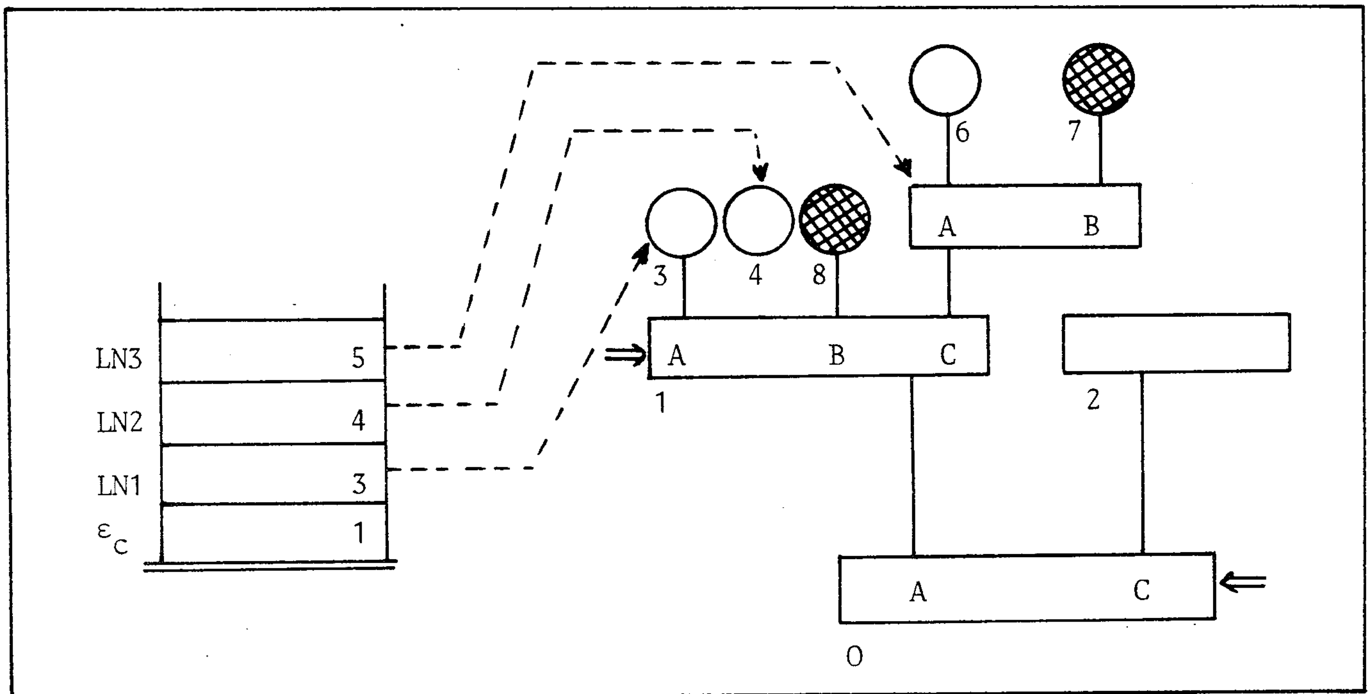


Figure 4.d.

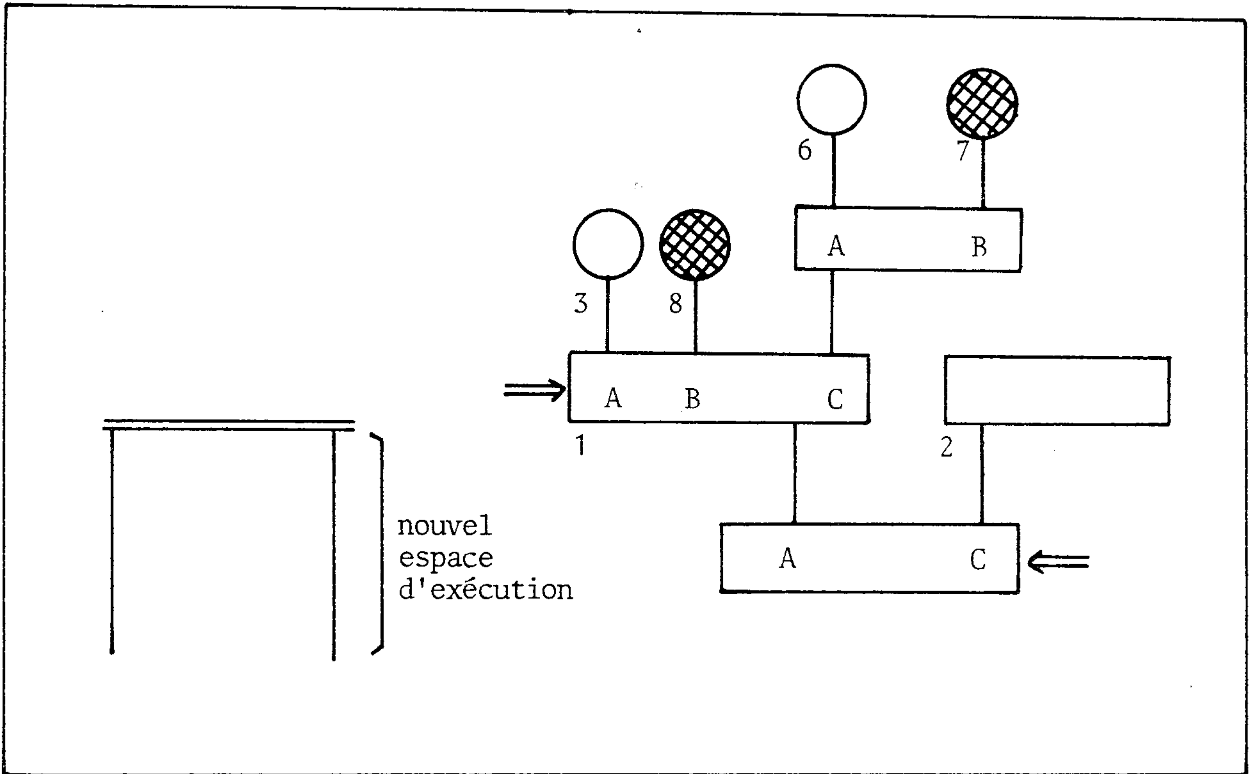


Figure 4.e.

L'exécution de

CREATE(C,'B',type = 'segment')

a ajouté un nouvel objet (7) de type segment dans l'annuaire 5 sous le nom d'entrée B (figure 4.b).

Il est important de remarquer que le nom partiel de l'annuaire C est résolu par rapport à l'environnement courant, ce qui ne provoque aucune confusion avec l'annuaire 2 qui a le nom d'entrée C dans l'environnement 0. Le processus P ne peut, à cet instant et dans cet état, créer aucun objet dans l'annuaire 2.

On remarquera que la création du segment 7 aurait pu être effectué par

CREATE(LN3,'B',type = 'segment')

Dans ce cas, la désignation d'annuaire est un nom local et la résolution est effectuée automatiquement en utilisant l'annuaire 5 (LN3), sans aucune référence à l'environnement courant (1).

DELETE(B) supprime le nom d'entrée B dans l'annuaire qui est l'environnement courant (LN0). L'objet correspondant n'est pas détruit physiquement dès cet instant, car LN2 le référence toujours, cependant LN2 est le seul moyen d'accéder et d'utiliser l'objet 4 car aucun nom partiel ne permet plus de l'atteindre (figure 4.c).

CREATE(,'B',*valeur* C.B) crée dans l'environnement courant (1) un nouvel objet (8) avec le nom d'entrée B. Le type, la description et la valeur de cet objet sont les mêmes que ceux de l'objet 7 (C.B par rapport à 1) (figure 4.d).

GETINFO(B) retournerait :

type = 'segment'
nom d'entrée = B
protection =
...

On remarque de plus, qu'il n'y a aucune confusion possible entre les objets 4 et 8 qui sont deux objets différents et qui possèdent seulement en commun le fait d'avoir eu le même nom d'entrée en des instants différents. De même, il n'y a aucune confusion entre 7 et 8, l'un étant seulement la copie instantanée de l'autre à un instant donné, mais ils ont chacun leur évolution propre.

Après l'instruction qui termine l'exécution de la procédure 3 et retire l'espace d'exécution de la pile de contrôle, plus aucun nom local ne référence l'objet 4 qui peut maintenant être physiquement détruit (figure 4.e).

5. MODIFICATIONS DE L'ESPACE D'EXECUTION

5.1. Exécution d'une procédure

< nom local >]...[< valeur immédiate >] ... ← CALL
(< désignation d'objet exécutable >
[,< désignation d'objet >] ...)
(< valeur immédiate >] ...)

permet de construire un espace d'exécution à partir des paramètres de la primitive CALL.

Les noms locaux correspondant aux paramètres sont créés, en particulier LN1[*] qui est l'objet exécutable (qui peut être un segment de programme ou un périphérique).

Les paramètres du CALL sont résolus uniquement par rapport à l'espace d'exécution lors de l'appel (noms locaux dont l'environnement courant).

L'environnement de l'objet exécutable devient le nouvel environnement courant du processus, l'espace d'exécution précédent étant sauvegardé dans la pile de contrôle.

L'objet exécutable peut fournir des objets comme résultat, dans ce cas ces résultats sont affectés à la liste des noms locaux fournie dans la partie gauche de l'appel (CALL) ; ces noms locaux sont ceux de l'espace d'exécution de l'appelant.

Il n'y a qu'un seul point d'entrée par objet exécutable.

Les informations nécessaires pour le retour en séquence après l'appel sont aussi sauvegardées dans la pile de contrôle.

Parmi les paramètres de cette primitive, on trouve des valeurs immédiates qui sont simplement des chaînes de chiffres binaires. Conceptuellement, ces valeurs immédiates représentent un nouveau type d'objet et peuvent donc être passées dans des noms locaux. Toutefois, dans la représentation présente, nous pouvons pour des raisons d'écriture et aussi parce que cela est plus proche du prototype existant, admettre l'existence d'un ensemble de registres généraux dans lesquels on peut transférer les paramètres par valeur immédiate ; on utilise alors RG_i comme notation pour désigner la $i^{ème}$ valeur immédiate d'une liste de paramètres.

---

[*] par convention.

5.2. Fin d'exécution d'une procédure

RETURN([< désignation d'objet >]...)[(< valeur immédiate >)]...

permet de terminer l'exécution d'un objet exécutable (essentiellement une procédure dans ce cas).

L'espace d'exécution courant est détruit après recopie éventuelle des références indiquées dans RETURN dans les noms locaux indiqués dans le CALL correspondant à cette exécution.

Toutes les désignations d'objets, dans le cas des noms locaux et partiels, sont résolues par rapport à l'espace d'exécution de la procédure qui exécute le RETURN (appelée).

La liste des objets résultats est affectée en séquence à la liste des objets en partie gauche du CALL. De même pour les valeurs immédiates. Les noms locaux en pargie gauche du CALL sont définis dans l'espace d'exécution correspondant à la procédure qui a effectué le CALL.

L'exécution reprend en séquence après l'instruction CALL qui avait initialisé l'exécution de la procédure qui s'achève.

5.3. Capture d'objets dans l'espace d'exécution

[< nom local >]... ← BIND([< désignation d'objet >]...)

permet de créer un ensemble de noms locaux référençant l'ensemble des objets désignés et ceci dans l'espace d'exécution courant.

Si la désignation possède l'indication 'valeur', alors un objet est créé (avec le nom local indiqué) comme la copie de l'objet désigné. De même la désignation peut être tout simplement une description.

Ces deux derniers cas permettent de créer des objets temporaires, c'est-à-dire des objets qui n'ont aucun nom d'entrée dans l'espace d'objets ;   leur durée de vie est donc limitée à l'existence d'un nom local pour référencer cet objet.

5.4. Libération d'objets capturés

FREE(< nom local >)

permet de détruire un nom local, à la seule condition que ce nom local n'ait pas été créé comme paramètre d'un appel de procédure.

5.5. Exemples

Considérons la figure 5 ; nous allons illustrer l'utilisation des différentes primitives de modification de l'espace d'exécution sur cette figure. Au début, l'espace d'exécution du processus P est constitué de l'environnement courant (0) et de la procédure (1) connue sous le nom local LN1 (figure 5.a).

(LN2,LN3) ← CALL(B.A,C.B,*valeur* C.A)*

demande l'exécution du segment 4 avec l'annuaire 7 en paramètre et une copie du segment 6. L'espace d'exécution (au moment de l'interprétation de la primitive CALL) est utilisé pour résoudre les noms partiels B.A, C.B et C.A. Puis cet espace est sauvegardé dans la pile des contextes, le nouvel espace contient les noms locaux LN1, LN2 et LN3 (référençant respectivement les objets 4, 7 et 8 qui est la copie de 6) et a comme environnement courant l'objet 2 (figure 5.b).

CREATE(B,'A',type=segment)

crée un segment (9) dans l'annuaire (5) sous le nom d'entrée A.

CREATE(LN2,'A',type=segment)

crée un segment (10) dans l'annuaire (7) sous le nom d'entrée A. Il est important de remarquer que l'annuaire 7 n'est accessible qu'au moyen du nom local LN2 et qu'il n'est pas dans l'espace visible du processus P à cet instant (figure 5.c).

---

* LN2 et LN3 sont réservés, à ce stade, dans l'espace d'exécution de l'appelant, mais ne seront affectés qu'au retour de la procédure appelée. Par souci de clarté, nous omettons cette réservation dans les figures.

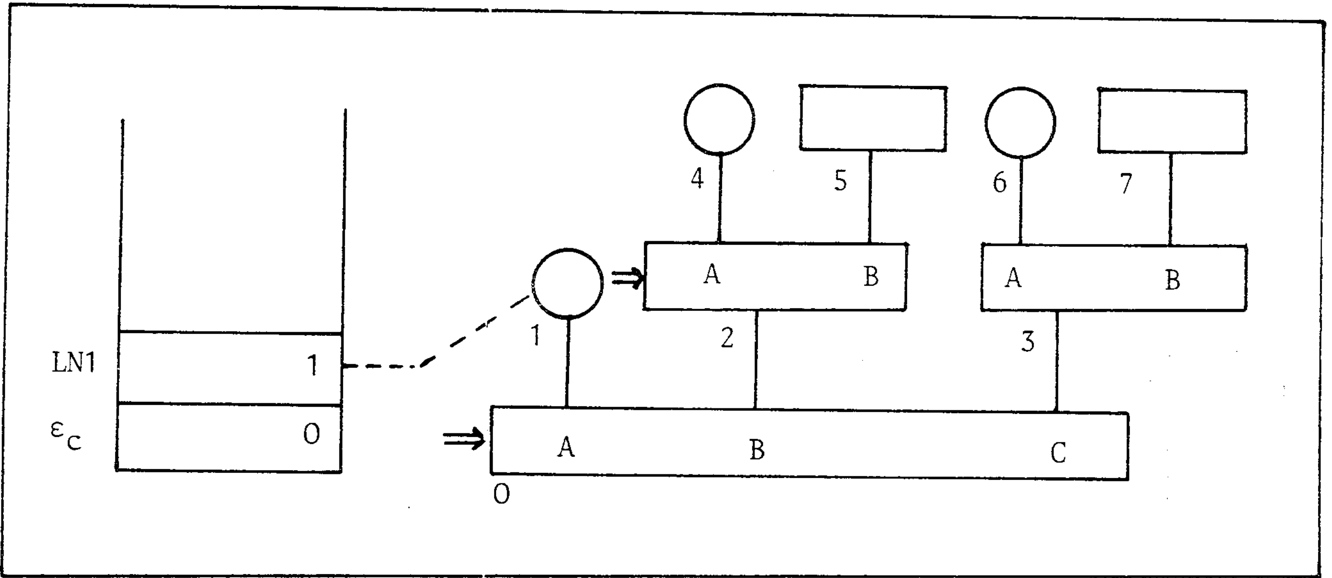


Figure 5.a.

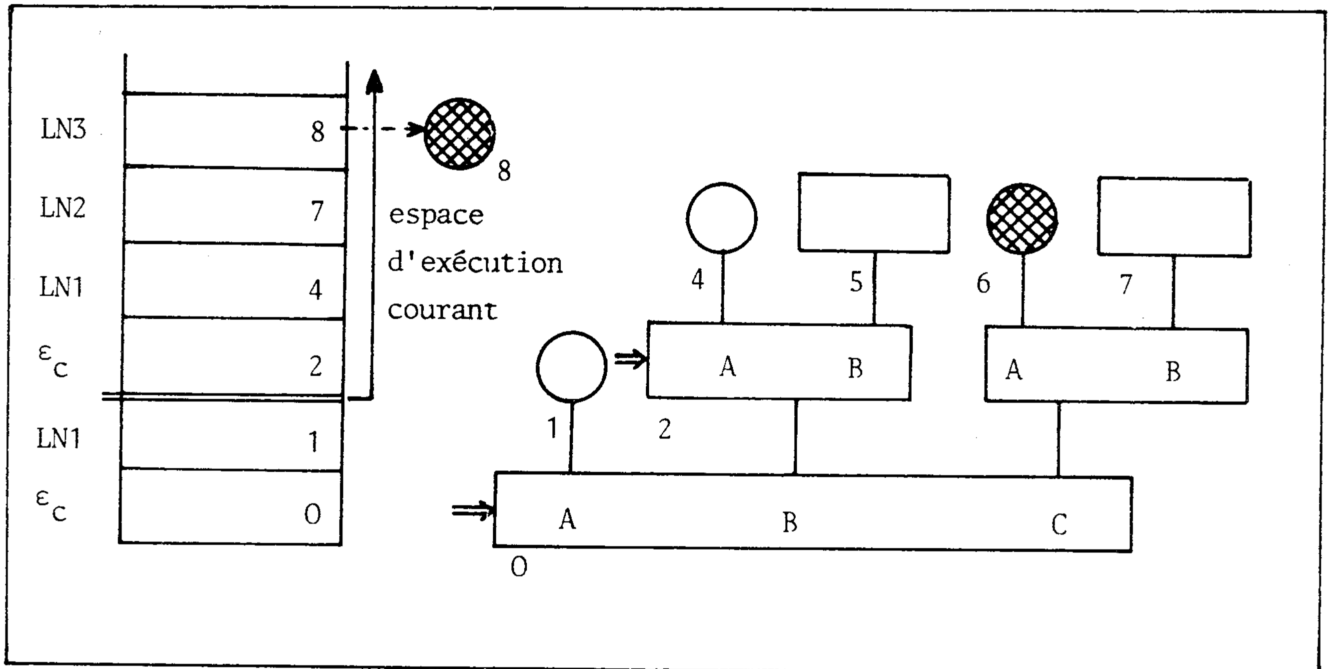


Figure 5.b.

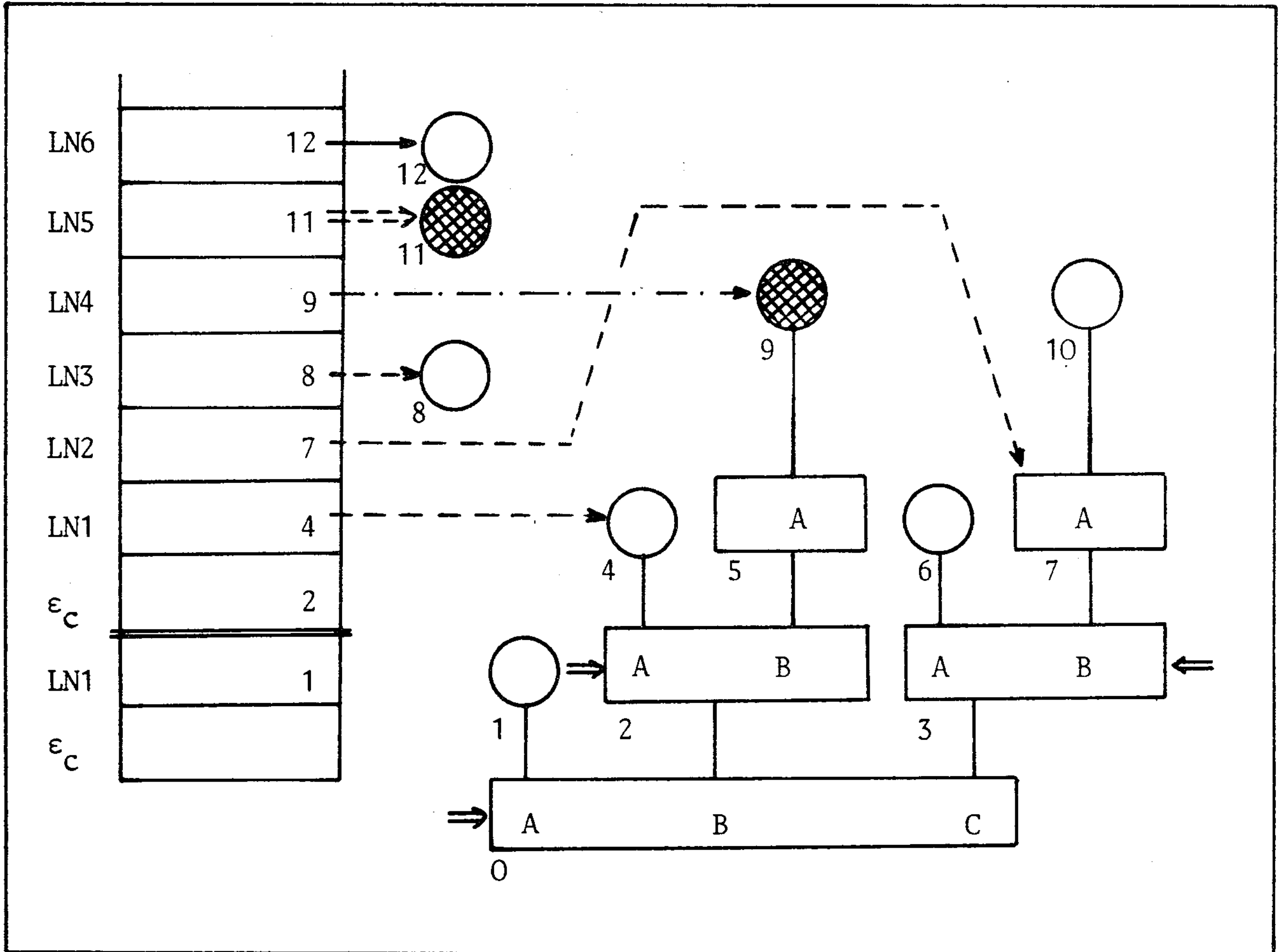


Figure 5.c.

LN4 ← BIND(B.A)

crée le nom local LN4 comme référençant le segment (9)

LN5 ← BIND(*valeur* LN4)

crée le nom local LN5 comme référençant un nouveau segment temporaire (11) de valeur initiale égale à celle de l'objet 9.

LN6 ← BIND(type=segment)

crée le nom local LN6 comme référençant un objet temporaire (12) de type segment (figure 5.c).

RETURN(LN5,B)

détruit l'espace d'exécution courant, après avoir transmis comme résultat les objets 11 (segment temporaire) et 5 (annuaire). Ces objets sont maintenant référencés par les noms locaux LN2 et LN3. Les objets 8 et 12 (temporaires) n'ayant plus aucune référence disparaissent (figure 5.d).

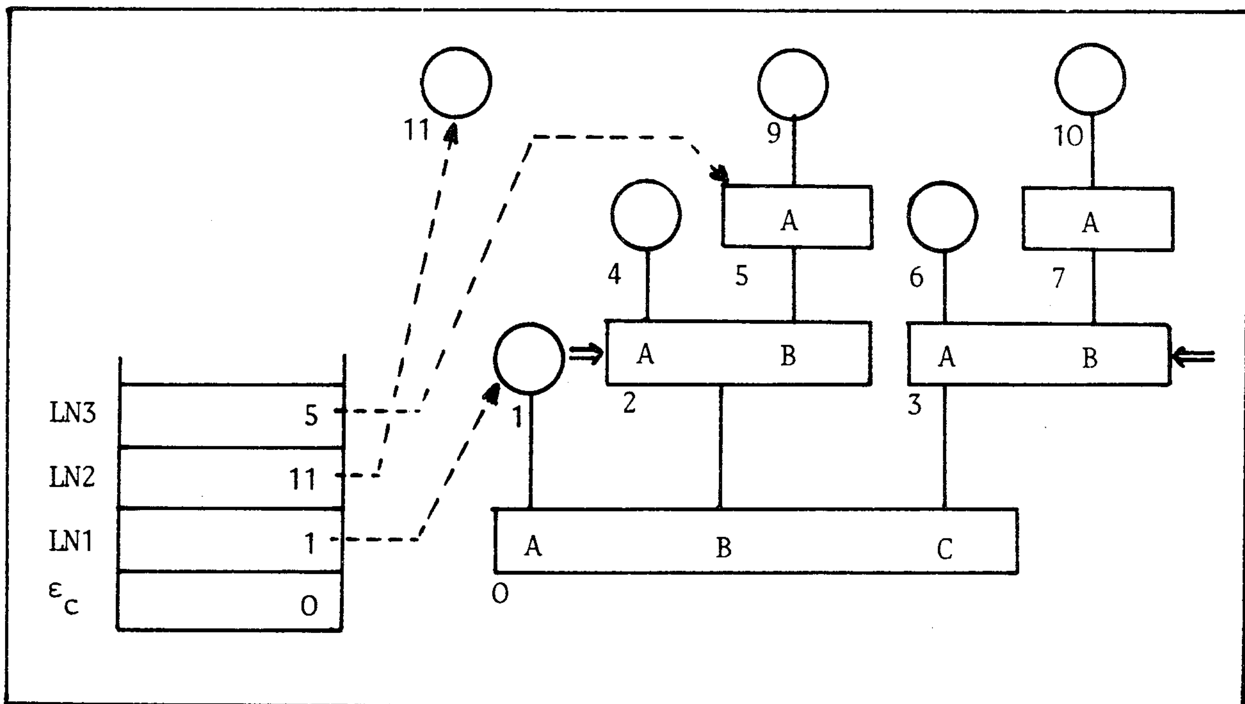


Figure 5.d.

Figure 5.e.

FREE(LN2)
détruit le nom local LN2 qui référence un segment temporaire et comme aucun autre nom local ne le référence, le segment 11 disparaît.
Ce segment ayant été créé par le processus P ne peut aucunement être transmis à un autre processus (figure 5.e).
Mais si par contre nous avions effectué
FREE(LN3)
alors la référence à l'annuaire 5 aurait disparu, mais cet annuaire existerait toujours.

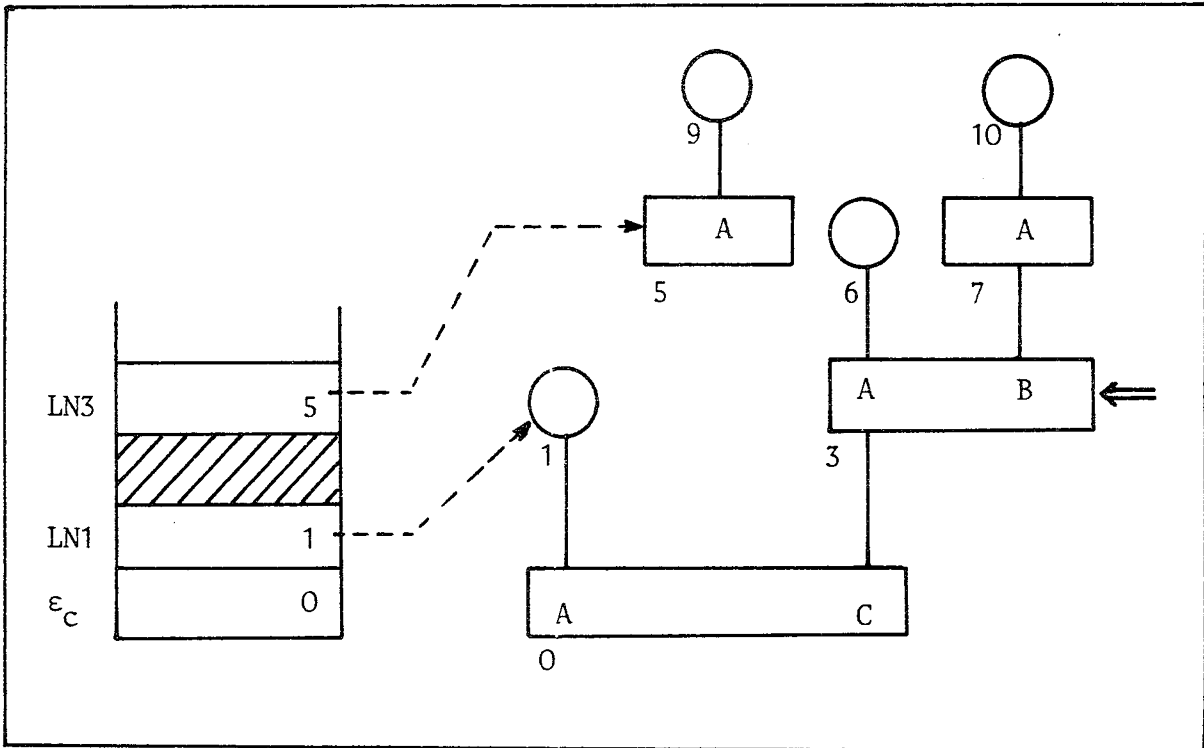


Figure 5.f.

DELETE(B)

détruit l'arborescence complète qui a comme racine l'annuaire de nom d'entrée B (i.e. 2, car nous sommes dans l'environnement 0).
Cependant, l'annuaire 5 est lié par LN3, ce qui veut dire que l'espace visible EV(5) ne peut pas être détruit physiquement. Ainsi, l'objet 9 peut être accédé par le processus P en utilisant le nom LN3.A, mais aucun autre processus (sauf s'ils ont lié les objets 9, 5, dans un nom local avant que P n'ait détruit 9) ne pourra jamais accéder 5 et 9.
Cependant, les objets 2 et 4 sont détruits physiquement (s'ils ne sont référencés par aucun nom local) dès le moment du DELETE (figure 5.f).
L'objet 9 peut cependant être détruit immédiatement par le processus P par DELETE(LN3.A).

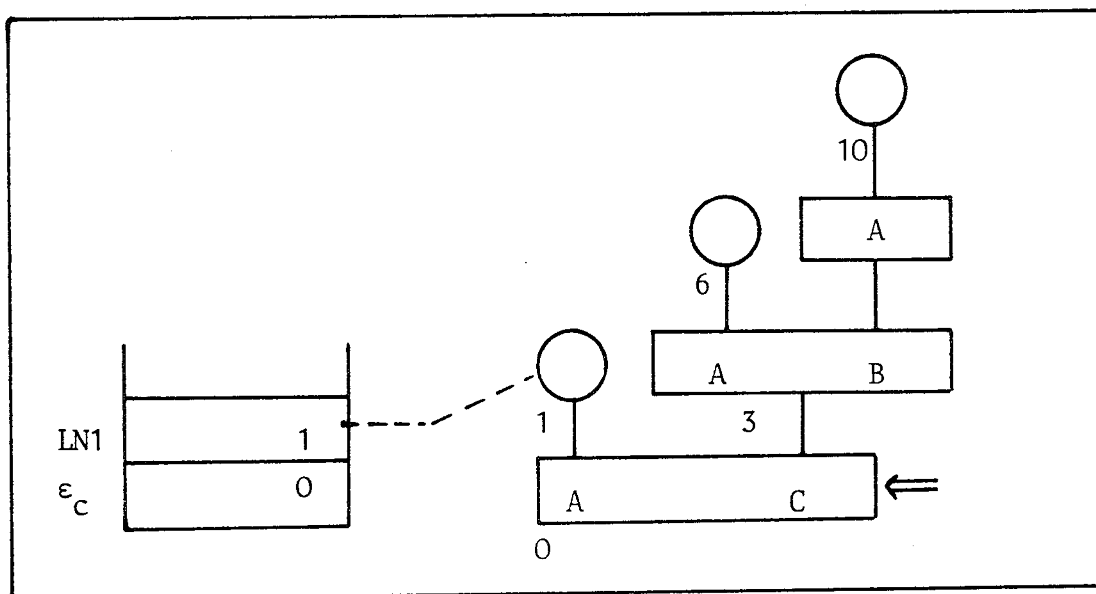


Figure 5.g.

FREE(LN3)
permet la destruction physique des objets 5 et 9 (figure 5.g).

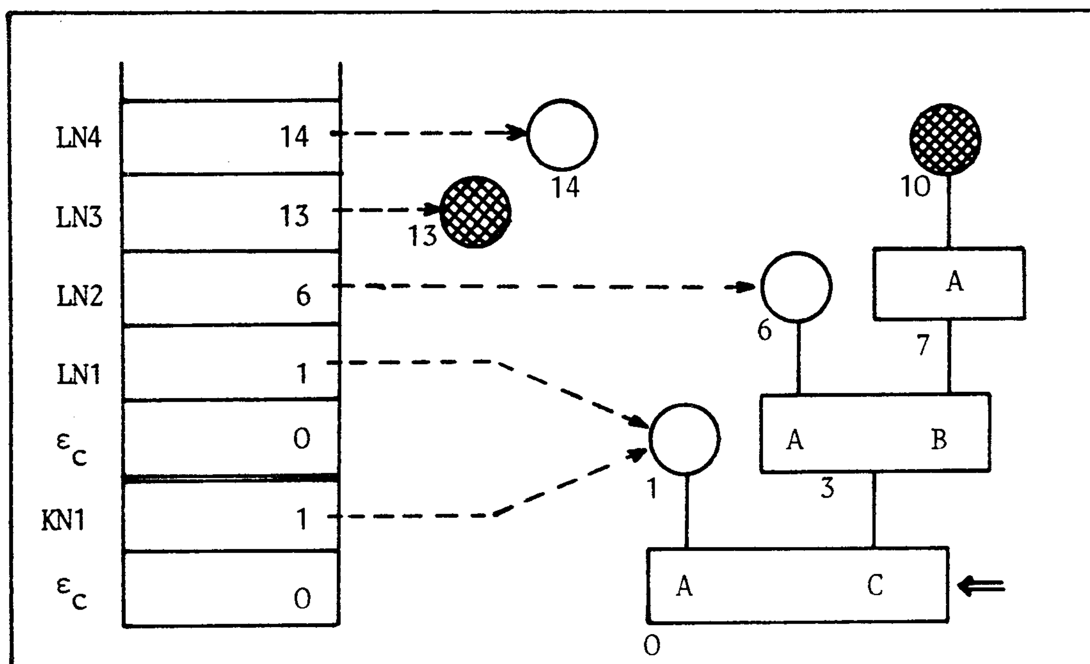


Figure 5.h.

CALL(LN1,C.A,*valeur* C.B.A,(type=segment))
permet la création d'un nouvel espace d'exécution et montre la variété des paramètres d'appel (figure 5.h).

5.6. Exemple d'utilisation des noms locaux

Nous donnons ici un exemple très simplifié d'organisation de l'adressage d'un processeur. Cet exemple ne sert ici qu'à donner une idée subjective au lecteur de la réalité et de l'utilisation de mécanismes décrits précédemment.

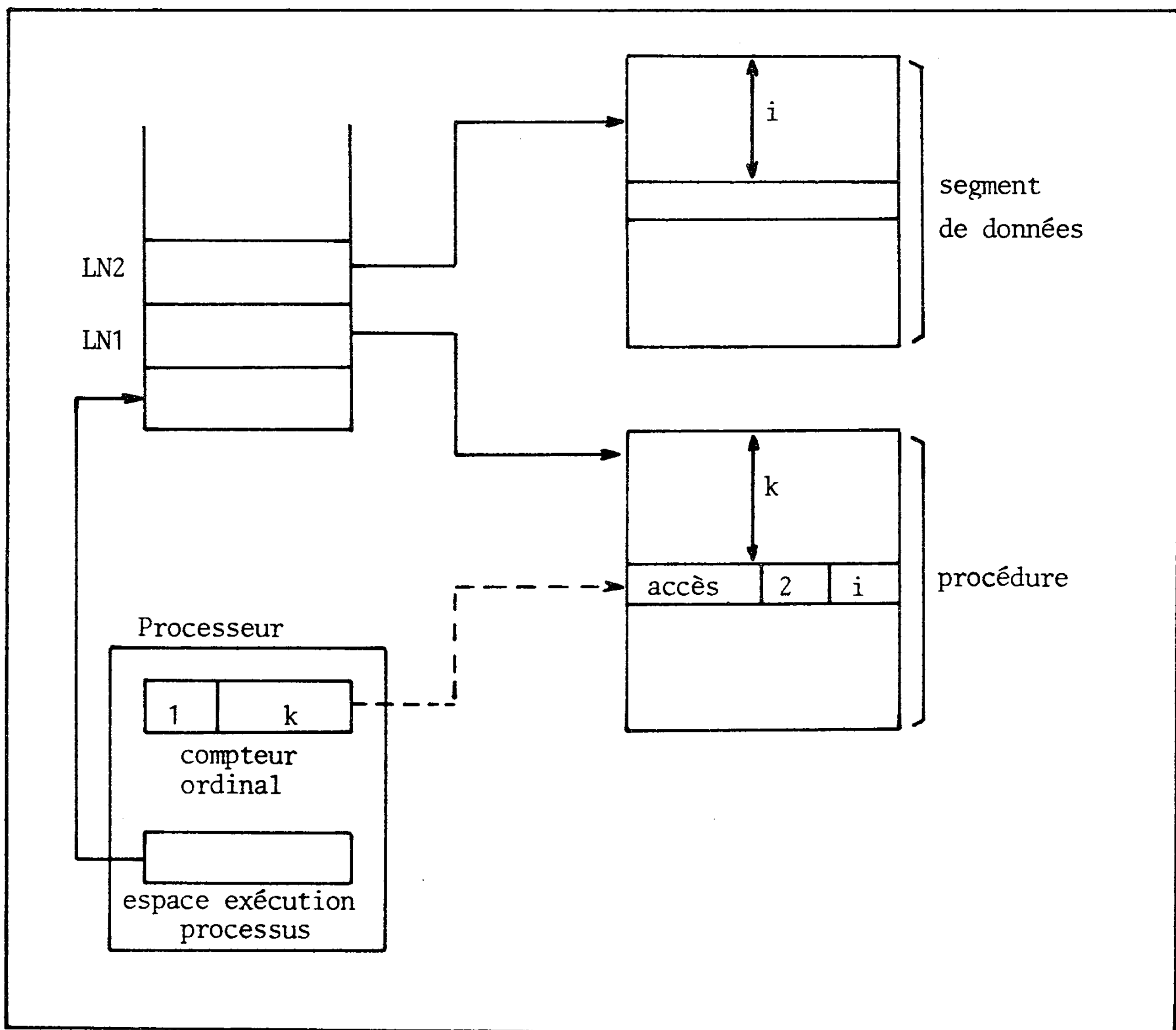


Figure 6 : Utilisation des noms locaux
par un processeur

La figure 6 donne un exemple d'un processus ayant l'exécution d'une procédure et d'un segment de données.

Une instruction de la procédure est de la forme :

< code opération >[,< operande >]...

où au moins l'un des opérandes est du format

< nom local >,< déplacement >

Par exemple, dans la figure 4, l'instruction k accède le $i^{\text{ème}}$ élément (mot, octet, ..) du segment de nom local LN2.

Dans le processeur on a un registre qui désigne l'espace d'exécution courant (dans le cas de la réalisation sur IRIS 80, il s'agit du pointeur sur une table de segment) et un registre qui désigne l'instruction courante.

Il est facile d'imaginer que l'espace d'exécution est une table de segments et que chacun des noms locaux référence une table de pages qui décrit chacun des segments. Les mécanismes de segmentation IRIS 80, 360/67 et GE 645 sont très facilement utilisables ici.

Le chapitre 4 discute plus en détail des mécanismes d'adressage des processeurs.

5.7. Moment de liaison

Les primitives que nous venons de voir encouragent l'identification de l'utilisation logique d'un objet à son utilisation physique, mais permettent toutefois de séparer ces utilisations pour garantir la cohérence d'une exécution. En effet, entre deux moments de liaison, une substitution d'objet est possible. La liaison doit alors être maintenue tant qu'une substitution est néfaste (cf. Chapitre 4).

6. TRANSFERTS ENTRE LES DIFFERENTS ESPACES

6.1. Primitive

Nous avons vu jusqu'à maintenant un certain nombre de méthodes pour établir des connexions entre les espaces d'objets et d'exécution, en particulier la primitive BIND qui permet soit de référencer un objet existant, soit de référencer un nouvel objet (temporaire).

Le problème qui se pose maintenant est celui de l'incorporation dans l'espace des objets d'un objet temporaire et plus généralement du déplacement d'objet dans l'arborescence.

```
TRANSFER(< designation d'annuaire >]
        ,< nom d'entrée >          ]  destination
        ,< désignation d'objet >)  ]  origine
```

détruit le nom d'entrée de l'objet désigné (dans son annuaire de création) et crée une nouvelle entrée dans l'annuaire désigné ; cependant le nom universel de l'objet n'est pas changé.

6.2. Exemple

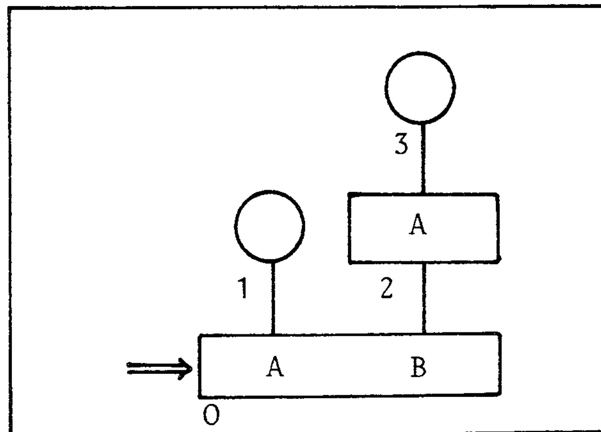


Figure 7.a.

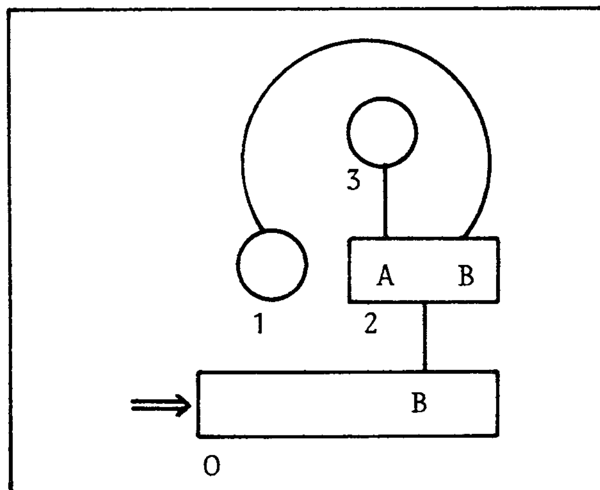


Figure 7.b.

Considérons la figure 7 et supposons que l'on effectue :

TRANSFER(B,'B',A)

le résultat est donné par la figure 7.b, c'est-à-dire que l'objet 1 a le nom d'entrée B dans l'annuaire 2, au lieu du nom A dans l'annuaire 0.

Si l'objet transféré était référencé dans des espaces d'exécution, il n'y a aucun problème car la référence d'un nom local a lieu uniquement par nom universel (qui dans ce cas n'a pas changé).

De même (figure 8.a) :

LNi ← BIND(type=segment)

crée un objet temporaire de nom universel 4 ;

et TRANSFER(,'A',LNi)

permet l'insertion dans l'arborescence de l'objet 4. Cependant, comme cet objet (temporaire) n'avait aucun nom d'entrée, aucune destruction de descripteur n'est effectuée (figure 8.b). Cette opération est souvent désignée dans les systèmes classiques sous le nom de "catalogage".

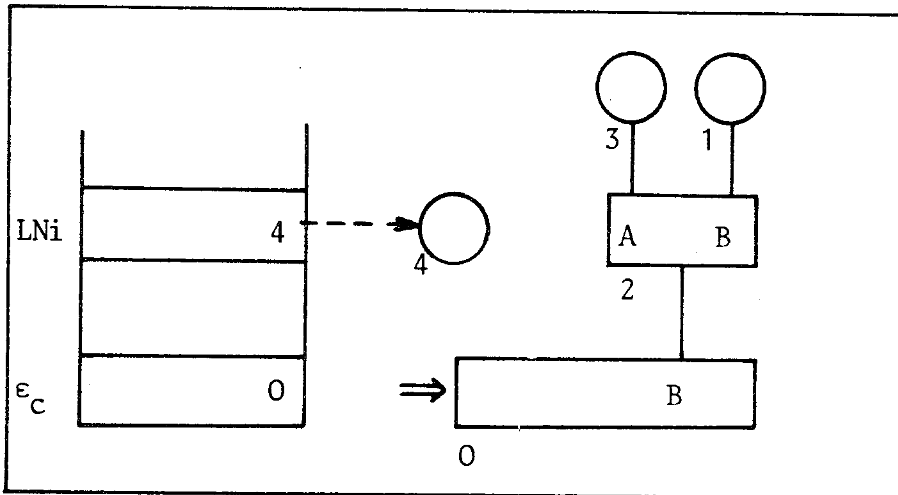


Figure 8.a.

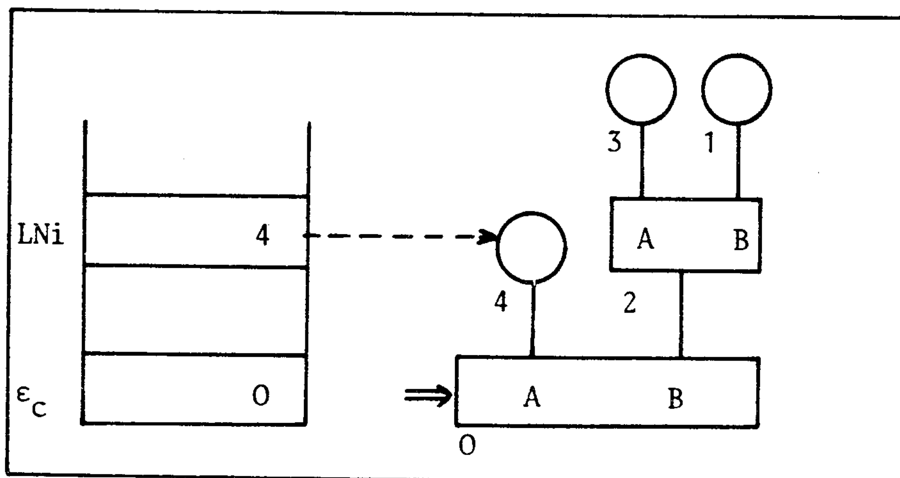


Figure 8.b.

7. PROTECTION

7.1. Introduction

Le type d'un objet définit les opérateurs qui lui sont applicables ; la protection consiste simplement à réduire l'utilisation de certains de ces opérateurs.

Lors de l'utilisation d'un objet par un processus, il faut que le noyau connaisse le type d'accès nécessaire à la manipulation désirée de l'objet, par exemple pour une lecture, une écriture ou une exécution d'un segment.

Il existe un certain nombre d'accès élémentaires à chacun desquels est associé un mécanisme de protection. Pour chaque type d'accès le descripteur de l'objet possède :

. type de l'accès,

. clé d'accès : l'accès est autorisé ou non.

Les types d'accès sont classés en deux catégories :

. accès liés à l'espace d'adressage,

. accès liés à l'utilisation.

7.2. Restrictions d'adressage

Ce sont les restrictions qui s'appliquent à l'objet en tant qu'élément de l'espace d'adressage.

Les types de protection sont liés aux différentes primitives, on trouve

D : DELETE l'objet peut ou non être détruit

T : TRANSFER

X : exécution (CALL)

M : MODIFY

C : copie : l'objet peut (ou non) être copié, par exemple lors d'une création.

7.3. Restrictions d'utilisation

Ce sont celles qui s'appliquent lors de l'utilisation d'un objet, c'est-à-dire celles qui dépendent du type de l'objet. Par exemple, un segment pourra être en "lecture seule" ou "exécution".

Ces protections sont réalisées par le processeur qui interprète le type de l'objet. Pour un segment, ce sera par exemple directement le matériel.

Si on considère le 10070 (ou IRIS 80) pour un segment la protection peut être :

- tout accès autorisé (lecture, écriture, exécution)
- lecture et exécution autorisés
- lecture (seulement) autorisée

Pour un périphérique, ce sera par exemple :

- lecture (transferts autorisés en entrée, i.e. depuis le périphérique vers un segment)
- lecture et écriture
- autres, i.e. dépendant du périphérique, tels que par exemple lecture arrière, etc ..

7.4. Définition

```
< restriction >::=< restriction d'adressage >/< restriction d'utilisation>
< restriction d'adressage >::= D / T / X / M / C
< restriction d'utilisation> ::= sera définie pour chaque type d'objet
```

Dans tous les exemples, l'absence d'indication de protection indique qu'aucune restriction ne s'applique à l'objet.

Nous utiliserons indifféremment les notations suivantes :

DTX : destruction, transfert, exécution *autorisés*
(seules les opérations indiquées sont possibles à l'exclusion de toute autre)

ou ¬M¬C : modification et copies *interdites* (toutes autres opérations possibles)

ou rien : tout autorisé.

Par exemple, un segment ayant les restrictions d'adressages limitées à X et aucune restriction d'utilisation, ne pourra qu'être exécuté par CALL et pendant l'exécution la protection du processeur sera nulle (programme non réentrant).

Cependant, avant l'exécution, même si le segment est lié dans un nom local, il sera impossible d'effectuer quelque modification que ce soit à l'intérieur de ce segment.

7.5. Exemple

Considérons la figure 9 où l'objet 3 possède le nom d'entrée A dans l'annuaire 2 et une protection contre la destruction (¬D).

Si un processus P est décrit par la pile de contrôle donnée par la figure 9, alors il ne peut effectuer DELETE(A).

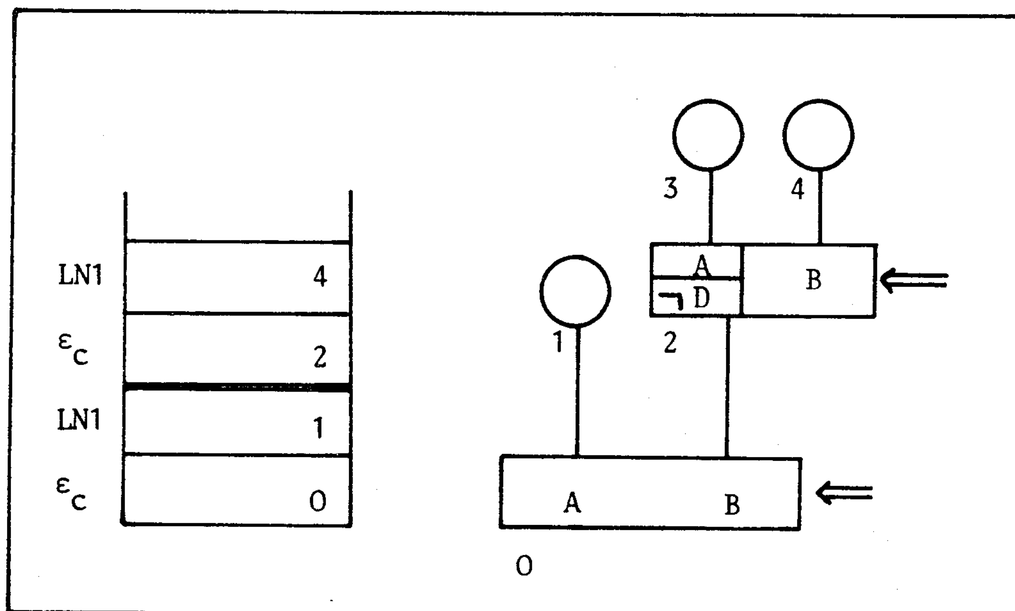


Figure 9

7.6. Portée des restrictions

Il est souhaitable, dans certains cas, de pouvoir ne pas tenir compte de la protection. C'est essentiellement le cas de l'interdiction de détruire un objet, car si un processus crée un objet avec une protection en destruction dans son environnement courant, il ne pourra pas le détruire.

Il existe donc un moyen de destruction qui est lié à la structure arborescente, c'est-à-dire que l'on peut indiquer, dans une destruction, que la protection ne doit pas s'appliquer et ceci ne marque que si l'objet protégé est dans un environnement strictement englobé depuis l'environnement courant. Lorsque l'objet est accédé en utilisant un nom partiel, toutes les protections placées sur le chemin (annuaires intermédiaires) sont ignorées.

De plus, pour tout objet désigné depuis un environnement englobant l'environnement de l'objet, il est possible de spécifier l'absence de restriction.

Si nous reprenons l'exemple précédent (figure 9), après l'exécution de RETURN, le processus P se retrouvera dans l'environnement 0 et la procédure 1 peut détruire l'objet B.A si elle le veut par :

DELETE(B.A{ aucune restriction })

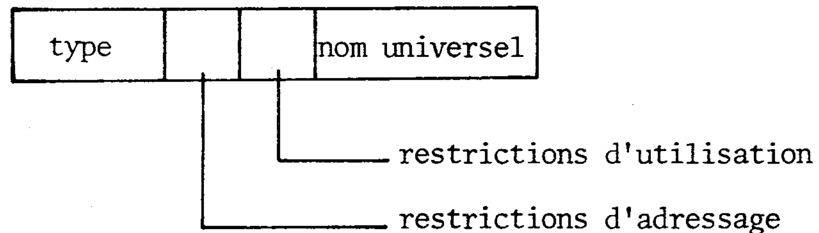
Cette opération est impossible pour la procédure 4. La destruction n'est possible que parce que 0 est un environnement strictement englobant de l'environnement 2. Ceci est une conséquence de la hiérarchie de l'espace des objets et réintroduit la notion de propriétaire : quoi que l'on fasse, il doit toujours exister quelqu'un qui puisse le défaire.

7.7. Cas de l'opération de liaison

Chaque fois qu'un objet est capturé, les restrictions de l'objet capturé (dans son descripteur) sont recopiées dans le nom local qui sert à la liaison. Ainsi on peut définir :

```
< nom local >  ::= < type de l'objet >
                 , < restrictions d'adressage >
                 , < restrictions d'utilisation >
                 , < nom universel >
```

Dans les dessins, les noms locaux seront désormais représentés par :



Dans la littérature de langue anglaise, ce type de référence est plus connu sous le nom de "capability" (cf. chapitres sur l'adressage et la protection dans cet ouvrage).

7.8. Accumulation de protections

Lors de l'utilisation d'un objet au moyen d'une des primitives précédentes, il doit toujours être possible de définir plus de restrictions que l'objet n'en a réellement. Par exemple, une opération de liaison (BIND) peut très bien spécifier que l'objet est en écriture interdite s'il s'agit d'un segment.

Reprenons l'exemple précédent (figure 9) ; il est possible d'effectuer

LN2 ← BIND(B{ ¬ D })

ce qui impose que

DELETE(LN2)

soit refusé alors que

DELETE(B)

est accepté dans ce cas.

On a dans cet exemple :

LN2 =	segment	¬D		4

Désignation d'objet :

Toutes ces accumulations sont possibles chaque fois qu'on est en présence d'une primitive admettant pour paramètre une désignation d'objet.

D'où la définition plus complète de la désignation d'objet :

< désignation d'objet > ::= ([*valeur*] < désignation simple d'objet >
/ < description d'objet >)
[{ restriction }]...

7.9. Exemple

Considérons la figure 10 où l'objet 2 est en restriction d'adressage égale à exécution seule (CALL), par contre n'a pas de restrictions d'utilisation.

LN2 ← BIND(B)

crée un nom local LN2 qui référence l'objet 2.

Cependant il est impossible (pour le processeur 10070 ou IRIS 80) d'accéder la valeur de 2. Mais,

CALL(LN2)

permet alors, à 2, de se modifier.

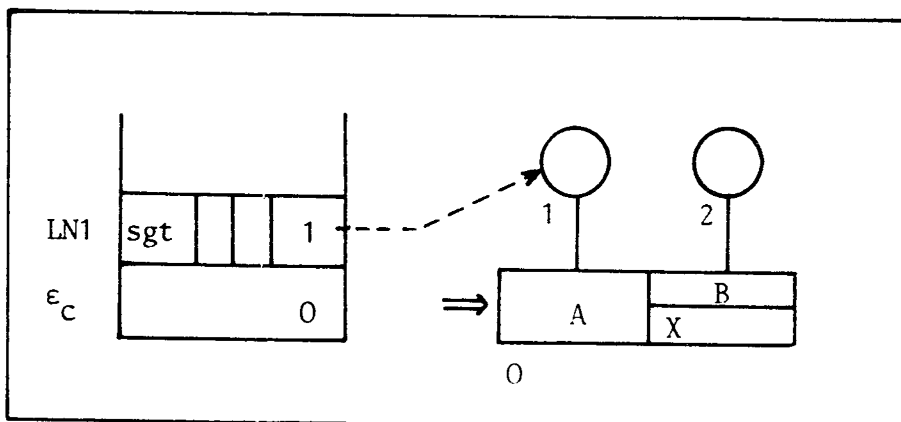


Figure 10

8. SYNCHRONISATION : ACCES CONCURRENTIEL AUX OBJETS

La structure de l'espace d'objets a été conçue de façon à permettre un partage aisé des objets. Toutefois, il est possible qu'un objet ne puisse être lié que par un processus et un seul à un instant donné, par exemple pour un objet de type programme non réentrant, ou éventuellement de type périphérique. Une telle propriété est une caractéristique intrinsèque de l'objet.

On appelle *degré de multiprogrammation* associé à un objet, le nombre maximal de processus différents pouvant lier ou manipuler cet objet au même instant.

La synchronisation est effectuée au niveau de l'objet ; il n'est pas possible de définir des synchronisations plus fines, par exemple sur des enregistrements, si l'objet est un fichier (segment). Dans ce cas, la synchronisation s'effectue en donnant un degré de multiprogrammation fini au programme de manipulation de ce fichier.

Les processus demandant l'objet sont mis en attente si le nombre de processus l'ayant capturé à cet instant est égal au degré de multiprogrammation. Ils sont débloqués au gré des libérations de l'objet par les processus l'utilisant. Ceci est valable pour tous les opérateurs applicables à l'objet, y compris la destruction.

Le choix du processus à débloquer quand l'objet est libéré n'est pas pris en compte à ce niveau. Seule une réalisation particulière peut effectuer des choix en fonction de différents critères (par exemple priorité entre les processus, priorité entre les différents opérateurs, ...).

9. RECAPITULATION SUR LES OBJETS

9.1. Objet annuaire

```
< objet annuaire > ::=< descriptif d'annuaire >[,< référence d'objet >] ..
< descriptif d'annuaire > ::= < type > = 'annuaire'
                              ,< restriction d'utilisation > =
                                 CREATE/DELETE/TRANSFER/GETINFO/MODIFY
                              ,< taille >
< référence d'objet > ::= < nom d'entrée >
                          ,< restriction d'adressage >
                          ,< degré de multiprogrammation >
                          ,< nom universel >
< nom universel > ::= < objet annuaire >/< objet segment >/
                      < objet périphérique >/< objet lien >
```

Le descriptif annuaire est la caractéristique intrinsèque d'un annuaire. Sa valeur (référence objet) est l'ensemble des descripteurs d'objets du point de vue de l'espace d'adressage (caractéristiques extrinsèques).

La valeur d'un objet dépend du type de l'objet ; par exemple pour un annuaire < valeur > = [< référence d'objet >]...

La taille d'un annuaire est le nombre maximal de noms d'entrée autorisé.

Il a paru intéressant d'effectuer des restrictions plus fines que celles indiquées précédemment, notamment pour la création d'objet ; dans ce cas, la protection porte sur la création de chaque type d'objet. Par exemple :

CD : création d'annuaire

CS : création de segment

CP : création de périphérique

CL : création de lien *

La primitive MODIFY porte sur les éléments suivants :

- . restrictions d'adressage d'utilisation de l'annuaire
- . taille
- . degré de multiprogrammation.

* Les objets de type lien seront examinés au paragraphe 11.

9.2. Objet segment

```
< objet segment > ::= < type > = 'segment'
                    ,< restriction d'utilisation > =
                       LIRE/LIRE,EXECUTER/LIRE,ECRIRE,EXECUTER
                    ,< taille >
                    ,< valeur>
```

C'est un objet de type 'chaîne de mots de 32 chiffres binaires', utilisable par un processeur 10070 ou IRIS 80.

< taille > est le nombre maximal de mots autorisés pour un segment. La taille réelle, à un instant donné, est comprise entre 0 et < taille >.

Dans le prototype, cette taille maximale est soit de 16 K mots pour le 10070, soit de 128 K mots pour l'IRIS 80. Pour des raisons d'implémentation, cette taille peut être définie en termes d'unité supérieure à celle du mot (512 mots) à cause de la pagination par exemple. Ce problème est considéré comme hors du sujet traité ici.

Dans le cas d'un segment, la primitive MODIFY porte sur les éléments suivants :

. restrictions,
. taille,
. degré de multiprogrammation.

9.3. Objet périphérique

< objet périphérique > ::= < type > = 'périphérique'
, < restriction d'utilisation > =
INPUT/OUTPUT/TEST/HALT/ORDER
,< valeur>

Un objet de type périphérique est uniquement exécutable (CALL), il permet de réaliser différentes fonctions (INPUT, OUTPUT, ORDER, ...) sur un périphérique réel.

Sa valeur est un moyen, pour le noyau, de retrouver l'emplacement physique du périphérique (par exemple, adresse de canal, d'unité d'échange) et son type de façon à pouvoir générer les programmes d'entrées-sorties physiques correctes.

Le chapitre 3 traite d'une façon générale les problèmes liés aux périphériques. Sachons seulement que les paramètres fournis à un objet de type périphérique sont uniquement des valeurs immédiates. Ainsi, si LPT est un objet périphérique (imprimante) et BUF un segment contenant la donnée à transférer,

CALL(LPT,BUF)('OUTPUT',D,ℓ)

permet de transférer une zone de 'ℓ' caractères situés à l'emplacement 'D' dans le segment BUF vers l'imprimante LPT.

Il est bien entendu que la création d'un périphérique, qui est lié à l'existence de ressources physiques de la machine, ne peut être effectuée par n'importe quel utilisateur ou sous-système. La protection peut être effectuée par la restriction d'utilisation, sur les annuaires, à des créations d'objets non périphériques (¬ CP).

9.4. Exemple de réalisation

Considérons la figure 11

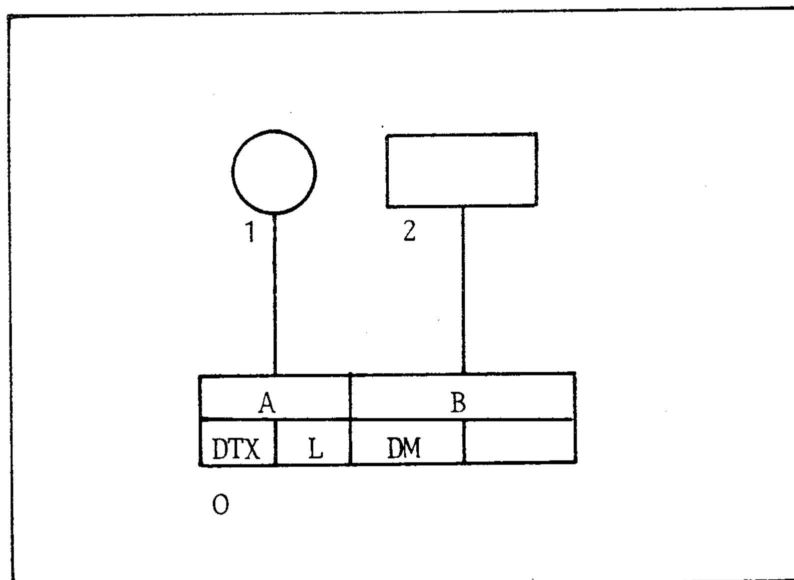


Figure 11

nous aurons la représentation suivante sur IRIS 80 :

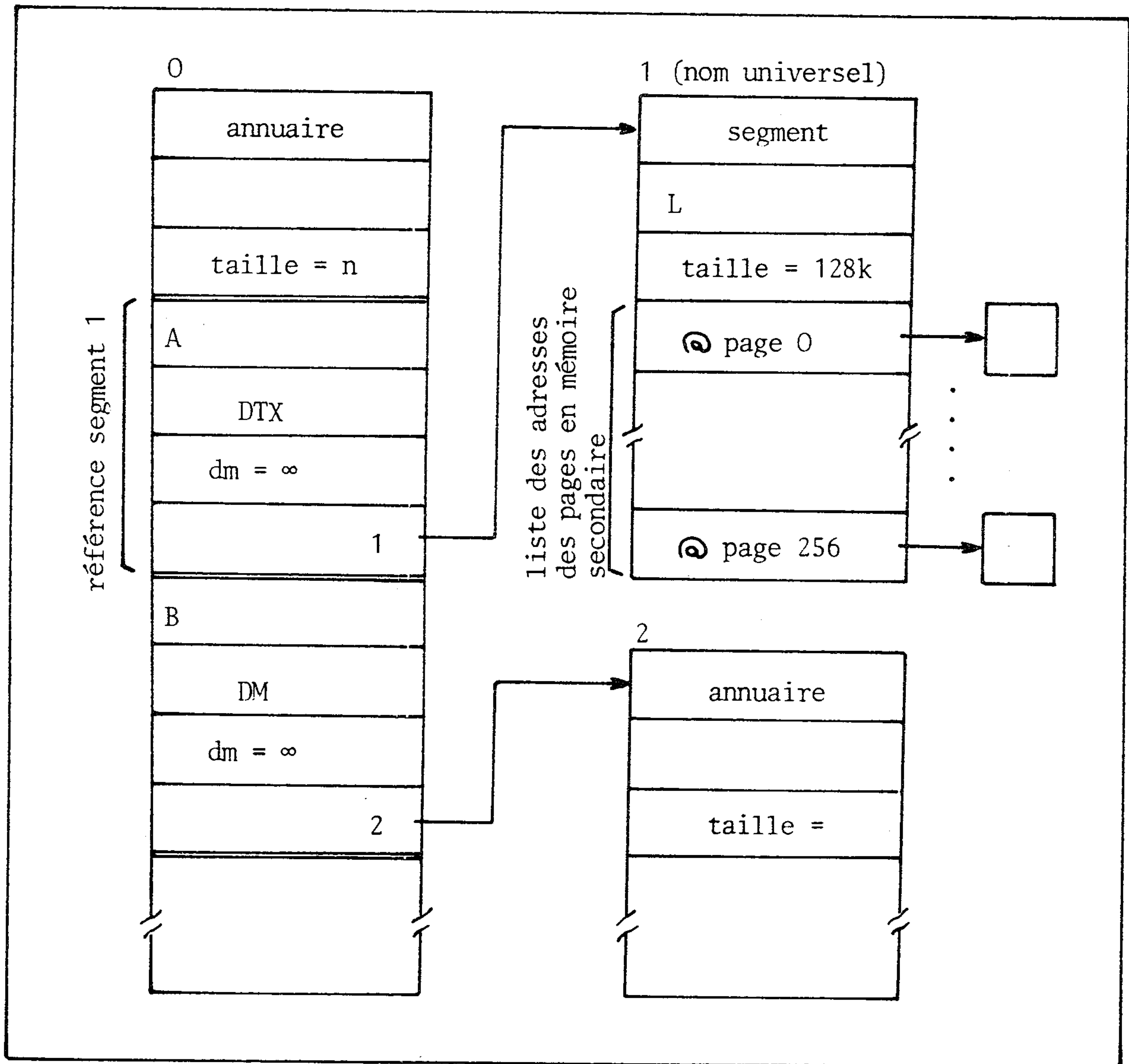


Figure 12

Dans ce schéma, nous n'avons représenté que les informations données à la définition ; il y a bien d'autres informations, telles que la table des pages, le nombre d'entrées effectives dans un annuaire, etc ..

10. ENVIRONNEMENT

10.1. Définition

Rappelons que l'environnement est l'annuaire de référence pour la résolution des noms partiels utilisés par une procédure ; il fixe aussi les pouvoirs ou prérogatives de la procédure.

```
< objet environnement > ::= < objet annuaire >
                           ,< attribut > = 'porte'
                           ,< prérogatives >
```

Les prérogatives sont un sous-ensemble des primitives du noyau.

Dans certains exemples précédents, nous avons vu que l'appel de certaines procédures provoquait un changement d'environnement du processus.

Figure 13

Lors de l'exécution d'une procédure, une question se pose : quel est l'environnement à utiliser pour résoudre les noms symboliques d'objets qui seront référencés par l'exécution de la procédure ?

Prenons l'exemple de la figure 13 et supposons que le processus en cours d'exécution ait comme environnement 0 et exécute CALL(A.A).

Ceci va exécuter les instructions du segment 4. Si 4, durant son exécution, référence un objet de nom LNO.B, s'agit-il de l'objet 3 ou de l'objet 1 ?

Le mécanisme de changement d'environnement est réalisé automatiquement par addition d'un *attribut* dans la description des annuaires ; cet attribut s'appelle '*porte*'.

Les annuaires possédant cet attribut et rencontrés lors de l'exécution de la primitive CALL, changent l'environnement qui devient automatiquement l'annuaire possédant l'attribut porte. Ce mécanisme ne joue que pour le premier paramètre d'un CALL (procédure).

Ainsi, pour reprendre l'exemple précédent, pour obtenir avec le nom B l'objet 3, il faut définir l'annuaire 2 comme ayant l'attribut *porte*, c'est-à-dire comme étant un environnement.

Si le nombre d'annuaires intermédiaires est supérieur à un et qu'un certain nombre possèdent l'attribut porte, seul le dernier annuaire ayant l'attribut porte devient le nouvel environnement.

Cette notion de porte est fondamentale dans GEMAU et implique que l'on associe au nom local des objets segments (en exécution possible) un élément supplémentaire : l'environnement d'exécution de cette procédure, i.e. lors d'une opération BIND l'indication de l'environnement correspondant à la dernière porte rencontrée ou de l'absence de porte. Ceci introduit l'aspect "historique" d'un chemin d'accès.

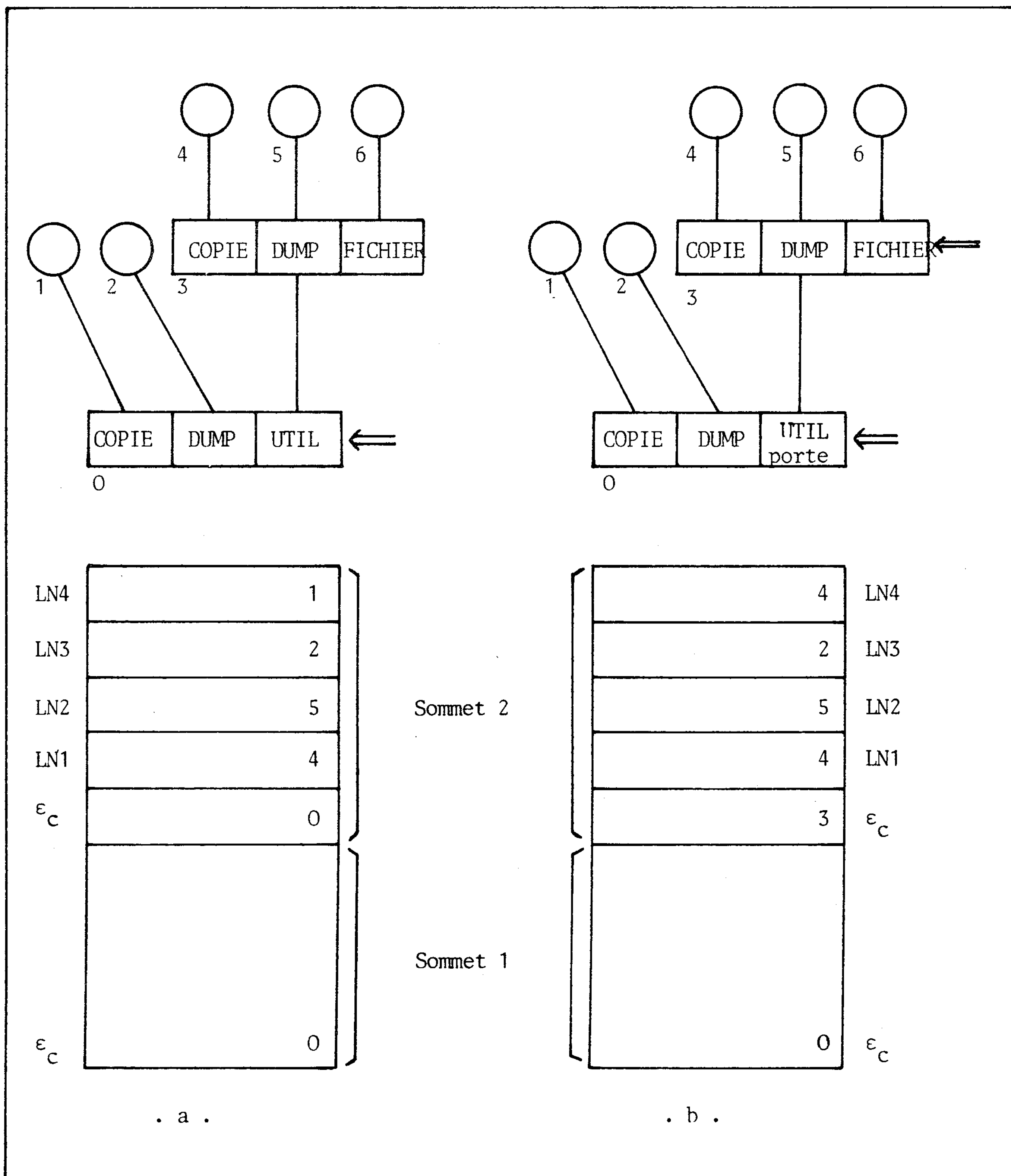


Figure 14 : Mécanismes des environnements

10.2. Exemple (Figure 14)

Dans le cas 14.b., l'annuaire de nom d'entrée UTIL dans 0 possède l'attribut environnement (porte), ce qui n'est pas le cas dans 14.a.

L'état 1 est l'état initial et l'environnement courant (ε_c) associé au processus est 0.

Supposons que l'on ait :

CALL(UTIL.COPIE,UTIL.DUMP,DUMP)

l'état courant du processus est alors l'état 2. L'environnement courant est soit 0 (14.a), soit 3 (14.b).

Supposons que le programme 4 comporte

LN4 ← BIND(COPIE)

le résultat est respectivement 1 et 4.

Dans le cas 14.b., l'objet 2 qui a été passé en tant que paramètre lors de l'appel n'est utilisable que sous le nom local LN3, mais n'a pas de dénomination dans l'environnement 3.

10.3. Prérogatives

Sur la plupart des machines actuelles, il existe deux modes de fonctionnement : maître / esclave.

A chacun de ces modes, des prérogatives différentes sont associées ; c'est le cas des instructions privilégiées.

Les prérogatives d'un processus sont les opérateurs (objets) utilisables par ce processus. Elles sont définies à un instant donné par l'espace d'exécution et l'environnement courant qui lui est associé.

A sa création, un environnement est muni d'une liste de primitives. Ce sont les prérogatives de l'environnement. Ces prérogatives sont un sous-ensemble des prérogatives du processus créant l'environnement.

Nous imposons de plus aux prérogatives de suivre les règles d'inclusion des espaces visibles.

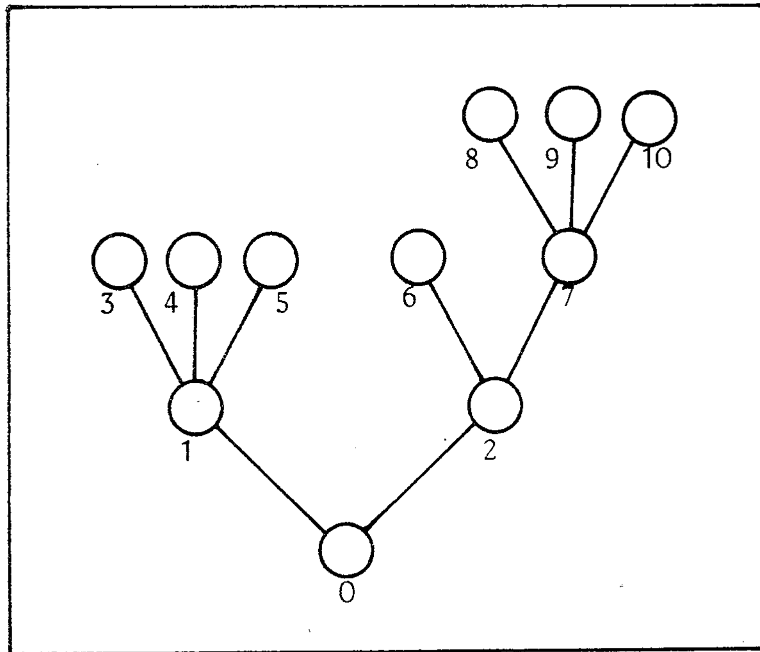


Figure 15 : Inclusion de prérogatives

Par exemple, dans la figure 15, les prérogatives de l'environnement 7 ainsi que celles de l'environnement 1, sont incluses dans celles de l'environnement 0 car l'espace visible de 7 comme celui de 1 est inclus dans l'espace visible de 0.

Ceci signifie que les prérogatives iront en décroissant en avançant le long d'une branche de l'espace des objets.

10.4. Relation prérogative - protection

Les prérogatives sont définies par l'environnement courant du processus. Tout changement d'environnement entraîne un changement des prérogatives du processus.

La protection (ensemble des opérateurs applicables) est spécifique du nom d'un objet. Elle est valide dans un domaine bien défini, c'est-à-dire quand l'objet considéré se trouve dans l'environnement courant. Si cet objet est dans un environnement englobé, la seule protection qui lui est associée est alors celle liée à son type, ou alors une protection définie au niveau de l'appel de procédure (restriction d'accès) ; dans ce cas, la protection est associée au nom local et est relative à l'exécution considérée.

10.5. Les primitives en tant qu'objets

De ce qui a été dit précédemment, il résulte que les primitives sont des objets comme les autres. Ceci nécessite seulement l'existence des opérations CALL et RETURN à un niveau vraiment primitif.

On pourrait d'ailleurs, dans une extension future, considérer que si une primitive donnée (par exemple DELETE) n'existait pas dans un environnement donné, alors le noyau rechercherait si aucun objet exécutable de nom d'entrée 'DELETE' n'existe dans l'environnement courant. Ceci est une extension souhaitable dans le prototype.

11. LIENS

11.1. Motivations

L'environnement permet de définir un cadre local pour la résolution des noms d'objets référencés par les processus et fixer les prérogatives.

Il est indispensable d'introduire un moyen permettant, depuis une branche de l'arborescence, l'accès aux objets situés sur une autre branche. On désire en effet pouvoir accéder à un même objet depuis des environnements différents avec des protections différentes, ou bien augmenter localement des prérogatives pour un travail déterminé.

C'est notamment le cas de programmes bibliothèque que l'on désire mettre à la disposition des utilisateurs avec des protections différentes selon l'environnement d'accès initial ; de plus, on ne désire pas dupliquer ces programmes : d'une part, parce que cela ne sert à rien d'avoir de multiples copies sur mémoire secondaire d'un même programme, et d'autre part, cela permet de ne remettre à jour qu'une seule et même copie d'un programme lors de modifications, sans avoir à connaître toutes les références à ce programme.

L'introduction de références supplémentaires dans l'espace d'objets permet de généraliser les modes privilégiés et non-privilégiés de machines classiques.

En effet, nous avons vu que des environnements différents peuvent posséder des prérogatives différentes. Par exemple, pour contrôler la création d'objet par un processus dans un certain environnement, on peut fournir un lien vers une procédure située dans un environnement possédant le pouvoir de créer des objets. Le lien vu de cette façon est un point d'entrée dans un certain environnement.

Cela correspond à deux cas précis :

. notion de point d'entrée dans un sous-système (1)

. notion de point d'entrée dans le système (2).

Ainsi que nous l'avons vu lors de la définition de la primitive CREATE (§ 4.1.), nous n'avons pas traité le cas où

< désignation d'objet > = < désignation simple d'objet >

Ce cas revient à définir un second nom d'entrée pour un objet donné. Nous appelons ces références supplémentaires des *liens*.

Un des problèmes importants à résoudre est celui du contrôle de la diffusion des liens. En effet, à partir du moment où on peut définir un lien sur un objet, il n'est plus possible de contrôler si de nouveaux liens sont créés à l'aide de ce lien.

Par exemple, dans la figure 16.a. un objet 5 est défini par un nom d'entrée (A) dans l'annuaire 2. Un processus peut créer un lien dans l'environnement 1 vers 5 par

CREATE(A,'C',B.A)

Le résultat est donné par la figure 16.b.

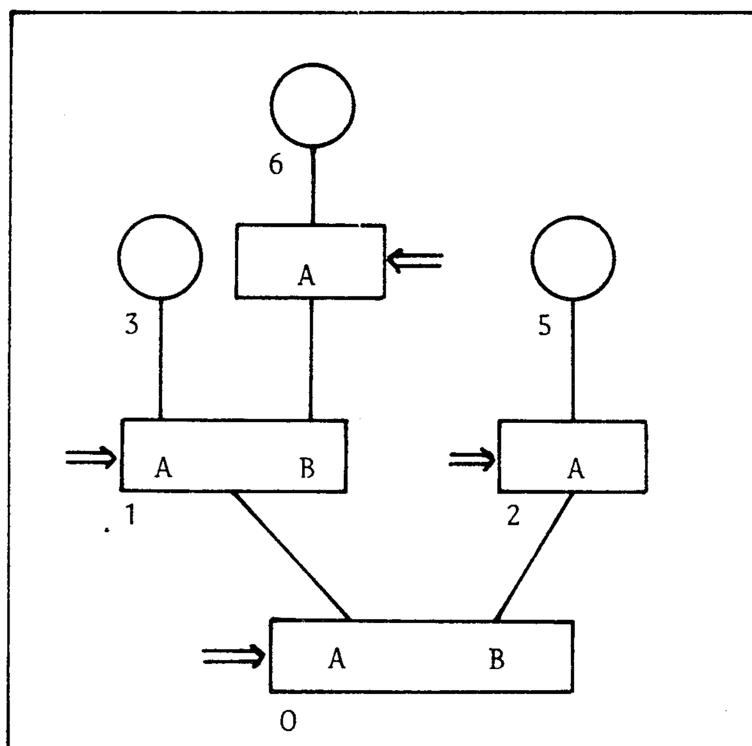


Figure 16.a.

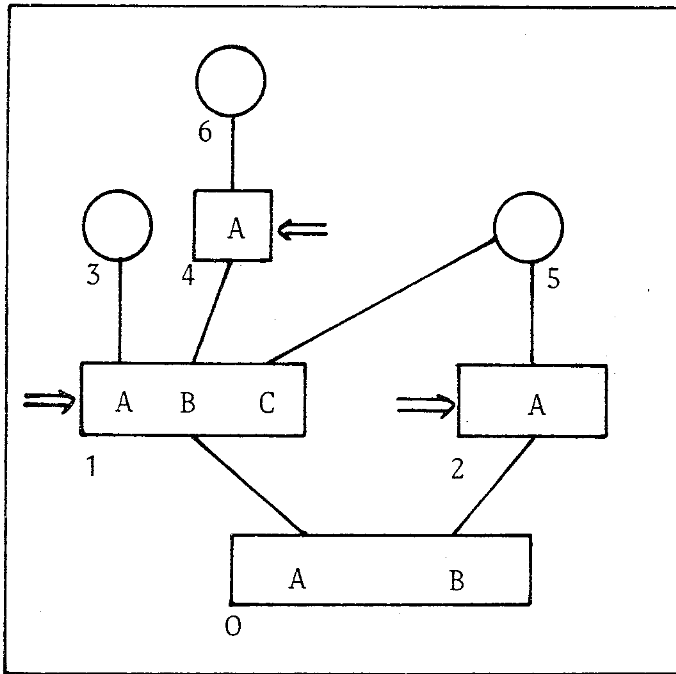


Figure 16.b.

Considérons maintenant un processus situé dans l'environnement 1, il peut maintenant accéder l'objet 5 par l'intermédiaire du nom partiel C. Il peut aussi créer un lien dans l'environnement 4 à l'aide de

CREATE(B,'B',C)

Le résultat est donné par la figure 16.c.

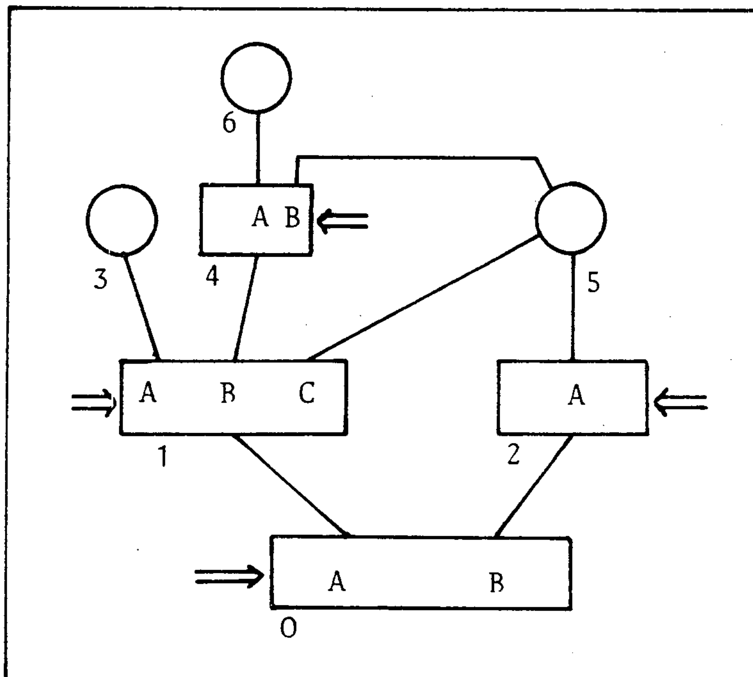


Figure 16.c.

Supposons maintenant que le processus situé dans l'environnement 0 veuille révoquer l'accès à l'objet 5, s'il détruit le nom d'entrée 'C' dans l'annuaire 1, il n'a pas retiré l'utilisation de l'objet 5 depuis l'espace visible de 1. En fait, il est impossible à ce processus de connaître tous les noms d'entrée sur l'objet 5 car n'importe quel autre processus ayant connaissance, à un instant donné, d'un nom d'entrée d'un objet, peut créer un lien, c'est-à-dire une copie du descripteur. On peut dire qu'avec un tel mécanisme il est impossible de contrôler la diffusion des références et donc de retirer un accès qui a été autorisé à un instant. C'est pour cela qu'*un lien est en fait une indirection, c'est donc un nouvel objet.*

Dans notre cas, le résultat des opérations effectuées dans l'exemple précédent est schématisé par la figure 16.d.

La suppression du lien 'C' dans l'environnement 1 rompt la chaîne des indirections.

De plus, pour permettre de conserver la propriété de substitution (cf. Chapitre 1, § 2.4.), il est nécessaire que le lien soit une référence symbolique (cf. § 11.3.). Il est important de voir qu'il y a un nom partiel privilégié qui est celui défini à la création de l'objet.

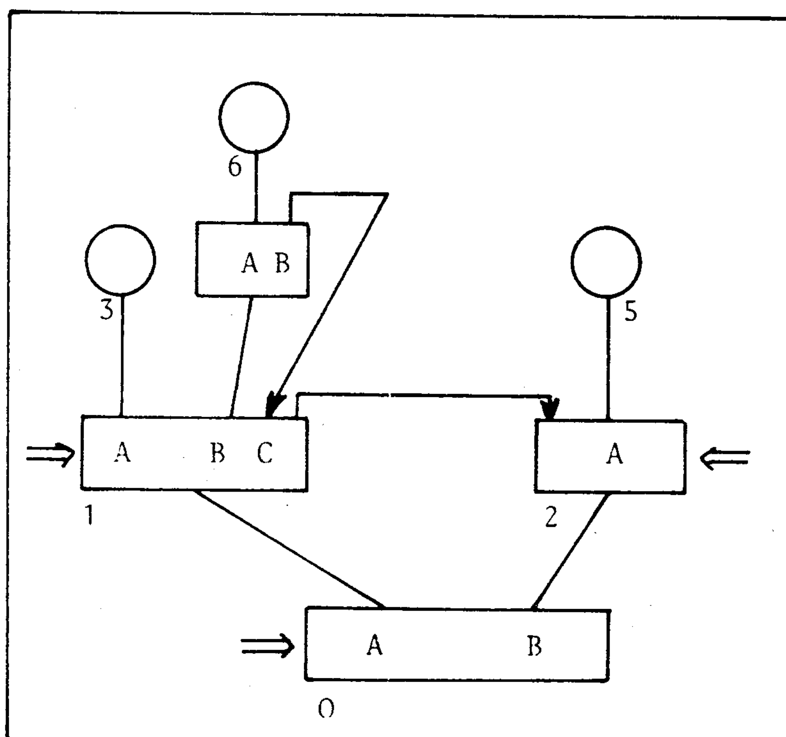


Figure 16.d.

11.2. Restrictions d'accès

Lors de la création d'un lien il faut fournir le nom de ce lien et le nom de l'objet référencé.

Les protections d'accès au lien sont définies comme pour un objet normal.

Cependant, en même temps que l'on définit le nom de l'objet référencé, on peut définir les restrictions d'utilisation à appliquer à l'objet final.

Quand le noyau, dans l'algorithme d'évaluation d'un nom d'objet, trouve un lien, il vérifie que les restrictions associées à la référence (par opposition à celles associées à l'objet lien) permettent l'accès ; le noyau continue ensuite à avancer dans l'arbre.

Le mécanisme d'environnements inclus les uns dans les autres ne joue plus quand il y a des liens, ce qui fait qu'il n'est pas possible de retirer toute protection à un objet quelconque s'il est accédé par un lien. C'est une protection absolue*.

Il faut noter que l'avantage de plusieurs noms réside dans différentes restrictions d'accès, en ce sens que les protections sont alors associées au chemin d'accès plutôt qu'à l'objet.

Un lien peut pointer sur un objet de type quelconque. Le processus créant un lien ne peut créer ce lien que dans son environnement vers un objet de son environnement, à cause de la structure même des noms.

Toute application d'une primitive sur un lien est une application sur l'objet terminal et non sur le lien.

---

* Il faut noter cependant que pour tout objet de l'espace des objets, il existe un nom partiel privilégié, par construction, qui ne comporte aucun lien. On peut donc toujours trouver un environnement permettant l'accès sans restriction, sauf pour la racine.

11.3. Valeur des liens

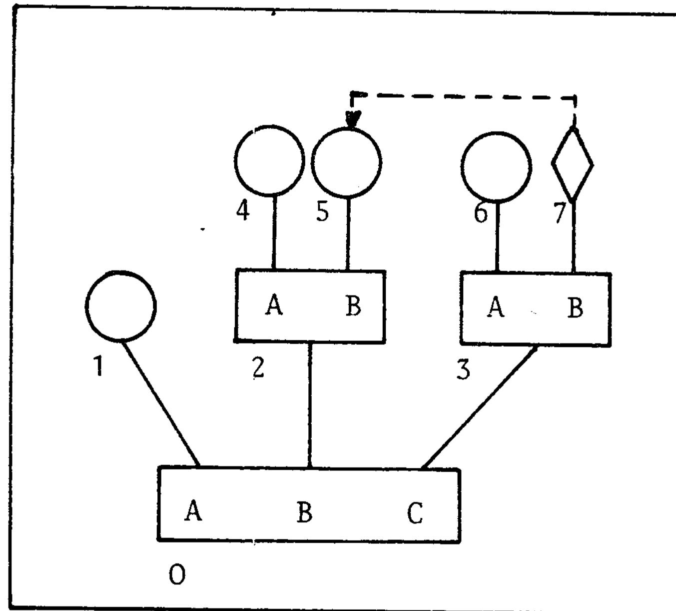


Figure 17.a.

Considérons par exemple la figure 17 et supposons l'existence d'un processus dont l'environnement courant soit 0. Ce processus veut créer le lien 7 vers le segment 5 ; il effectue ceci par

CREATE(C,'B',B.B)

Ceci crée un objet intermédiaire (7) de type lien qui contient en particulier les restrictions associées au lien.

La valeur interne du lien peut être le nom universel de l'objet référencé, c'est-à-dire 5.

Dans ce cas toute substitution ultérieure de l'objet B (5) sera impossible, car une destruction de B puis la création d'un nouvel objet B dans l'annuaire 2 fournira un objet nouveau (son nom universel ne sera pas 5). Il faut donc que la référence soit symbolique, c'est-à-dire soit

< nom universel >, < nom partiel >

La question qui se pose est celle très générale de l'indirection, à savoir une référence designe-t-elle un objet (nom universel) ou une position (nom partiel) ? Selon la solution choisie, la substitution sera (nom partiel) ou non (nom universel) possible.

La solution idéale consiste donc à pouvoir spécifier quelle est la partie de la référence qui doit être résolue lors de la création du lien.

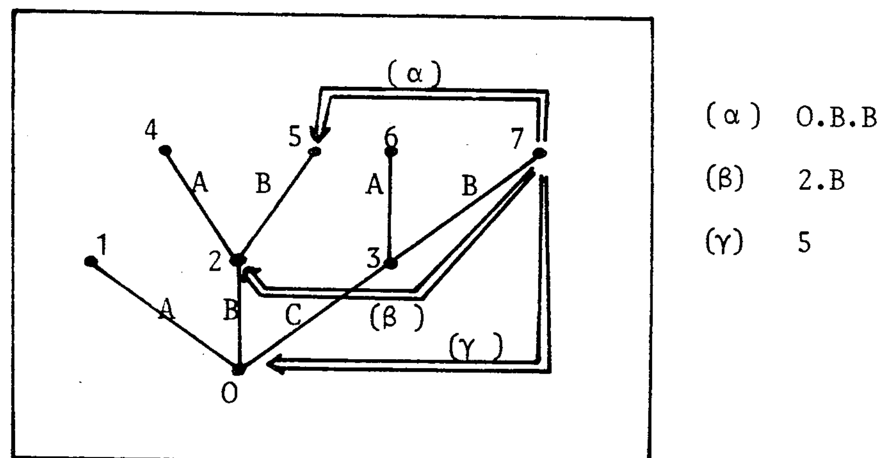


Figure 17.b.

La figure 17.b. schématise les trois différentes liaisons possibles pour la référence du lien 7. Nous utilisons pour cela :

CREATE(C,'B',(X,Y))

avec :

X : désignation de la partie résolue

Y : nom partiel (symbolique) non résolu à la création du lien

X et Y étant relatifs à l'objet à atteindre par le lien, nous avons les valeurs suivantes possibles :

(α)	X,Y	:	,'B.B'	valeur du lien 0.B.B
(β)		:	B,'B'	valeur du lien 2.B
(γ)		:	B.B	valeur du lien 5

On voit que dans le mécanisme proposé ci-dessus, on établit des liens symboliques, c'est-à-dire que l'on peut toujours changer l'objet pointé par un autre objet (lien/annuaire/segment/périphérique), sans changer la valeur interne du lien.

Ceci est très important dans un système où l'on veut remplacer des programmes utilisateurs sans changer la valeur des liens. Ceci permet la modularité et explique qu'il n'y ait pas de vérification de l'existence de la partie non liée de l'objet.

L'intérêt de définir de façon précise la partie résolue est aussi dû à l'existence des portes ; en effet, lors de l'évaluation d'un nom partiel utilisant des liens, seules les portes trouvées sur le chemin d'accès symbolique (i.e. non résolu) sont prises en compte (cf. § 11.10).

11.4. Exemple

Considérons la figure 18.a. qui définit sept objets d'un espace d'objets et l'espace d'exécution d'un processus P.

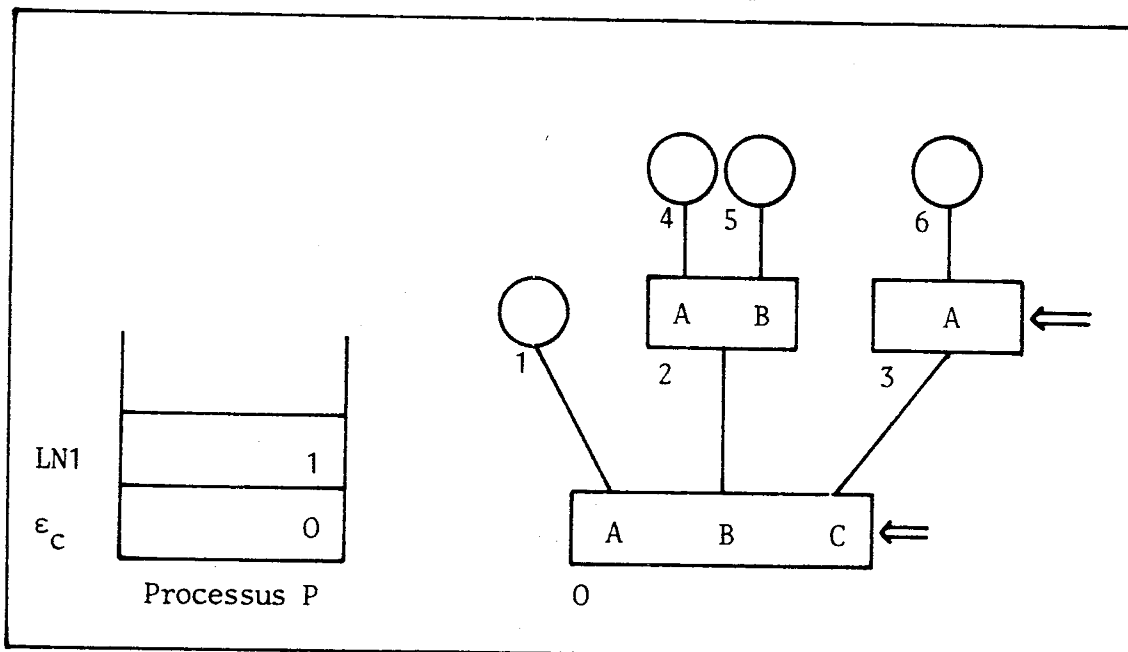


Figure 18.a.

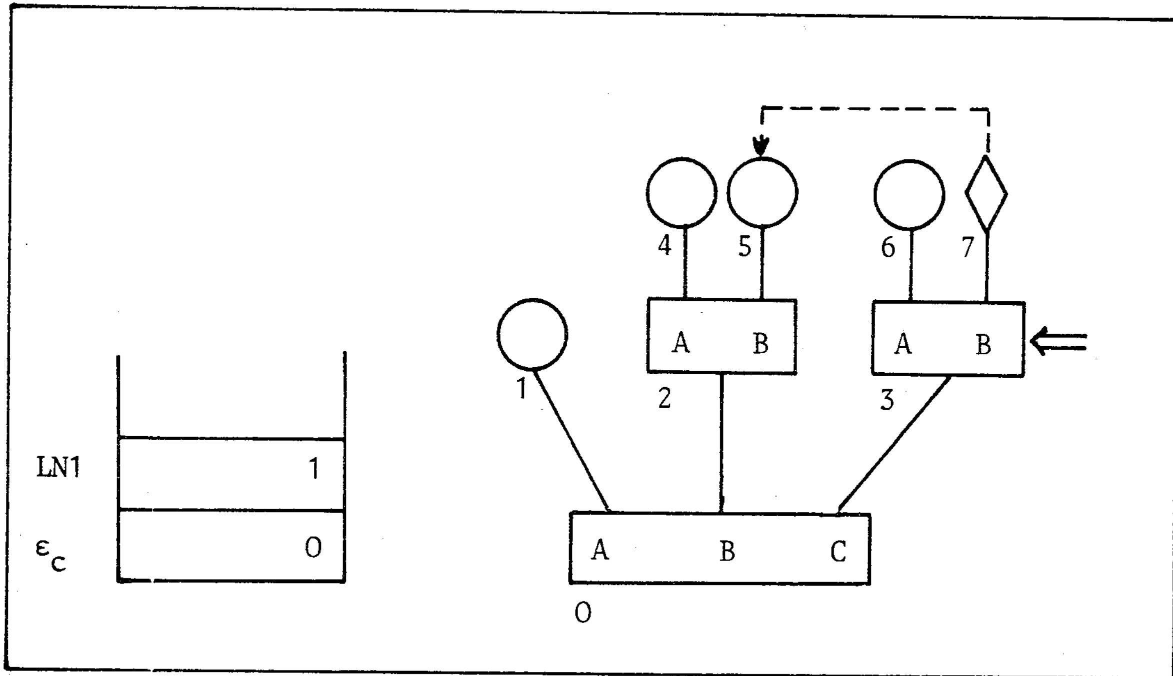


Figure 18.b.

L'exécution de

CREATE(C,'B',(,'B.B'))

crée dans l'annuaire C (C est un nom partiel par rapport à l'environnement courant) un objet de type lien qui référence l'objet B.B (5). Ainsi, l'objet 5 peut être désigné par le processus P (dans l'environnement 0) par

. B.B

. C.B

Il faut noter que la référence est une référence à l'objet O.B.B. (figure 18.b).

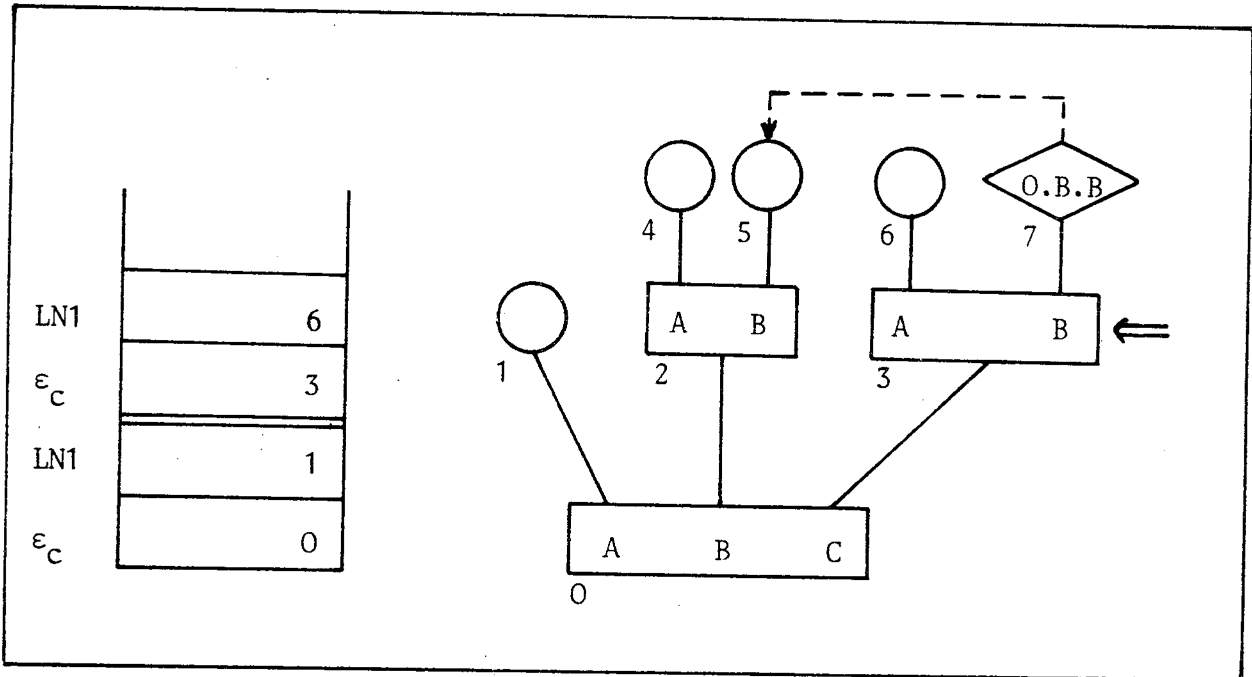


Figure 18.c.

Supposons que l'on exécute

CALL(C.A)

ceci va activer la procédure 6, l'espace d'exécution du processus étant sauvegardé, et le nouvel environnement courant devient 3. L'espace visible est composé des deux objets A et B, c'est-à-dire 6 et 7. Ainsi, l'objet 5 est accessible depuis l'environnement 3 à travers le lien B (7), mais les objets 0, 1, 2 et 4 ne sont pas accessibles depuis cet environnement (figure 18.c).

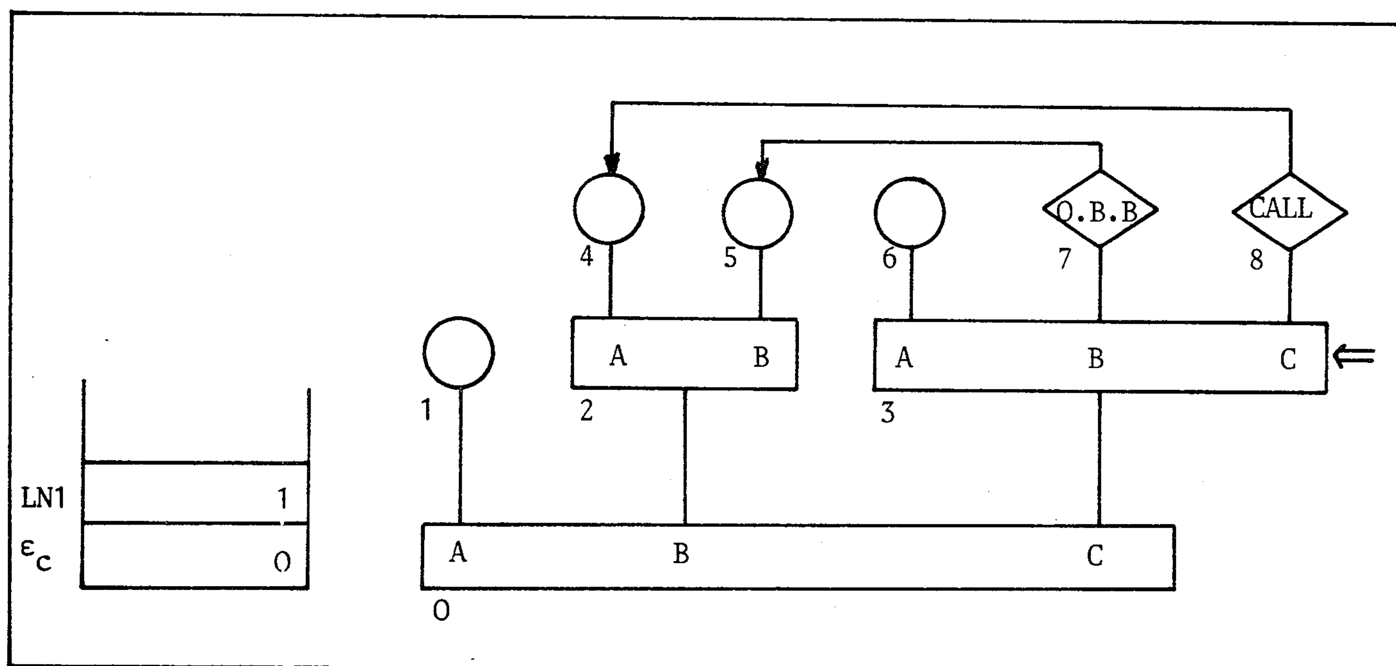


Figure 18.d.

Considérons maintenant

RETURN effectué par 6,

CREATE(C,'C',(B.A{ X }))

qui est effectué par 1, crée un objet de type lien dans l'annuaire 3 sous le nom d'entrée C. Ce lien référence l'objet 4, mais tout accès à l'objet 4 est limité à un appel de procédure (CALL), c'est-à-dire que

CALL(C.C) est autorisé

alors que

DELETE(C.C) est refusé

(figure 18.d).

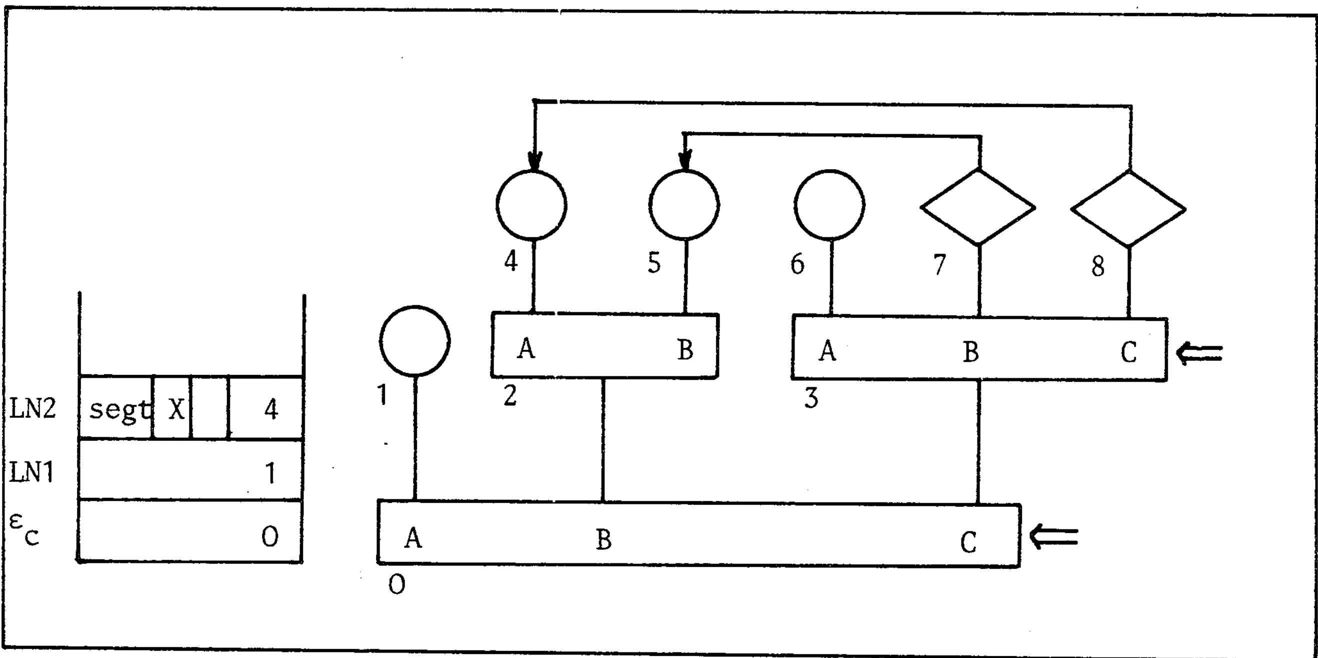


Figure 18.e.

LN2 ← BIND(C.C)

affecte au nom local LN2 l'objet 4, avec la restriction "appel seulement". Toute manipulation de l'objet à travers LN2 autre que CALL provoquera une erreur (figure 18.e).

11.5. <u>Cas des liens référençant des liens</u>

Considérons la figure précédente (figure 18.e) et supposons que l'on veuille, dans l'environnement 3, créer un nouvel environnement D (9) et dans cet environnement un accès à l'objet C (8). Supposons aussi que cela soit effectué par un processus situé lui-même dans 3. Ce processus voit l'objet C comme un segment et ignore qu'il s'agit d'un lien. On peut avoir

CREATE(,'D',(type=annuaire,attribut='porte'))

CREATE(D,'A',C)

L'objet 10 qui est le lien a-t-il comme partie résolue de la référence les noms universels 4 ou 8 ?

Nous avons choisi, si la partie résolue désigne un lien, de prendre cette valeur de façon à conserver les propriétés de substitution et de révocation (cf. § 11.1). Ici on obtient donc l'objet 8.

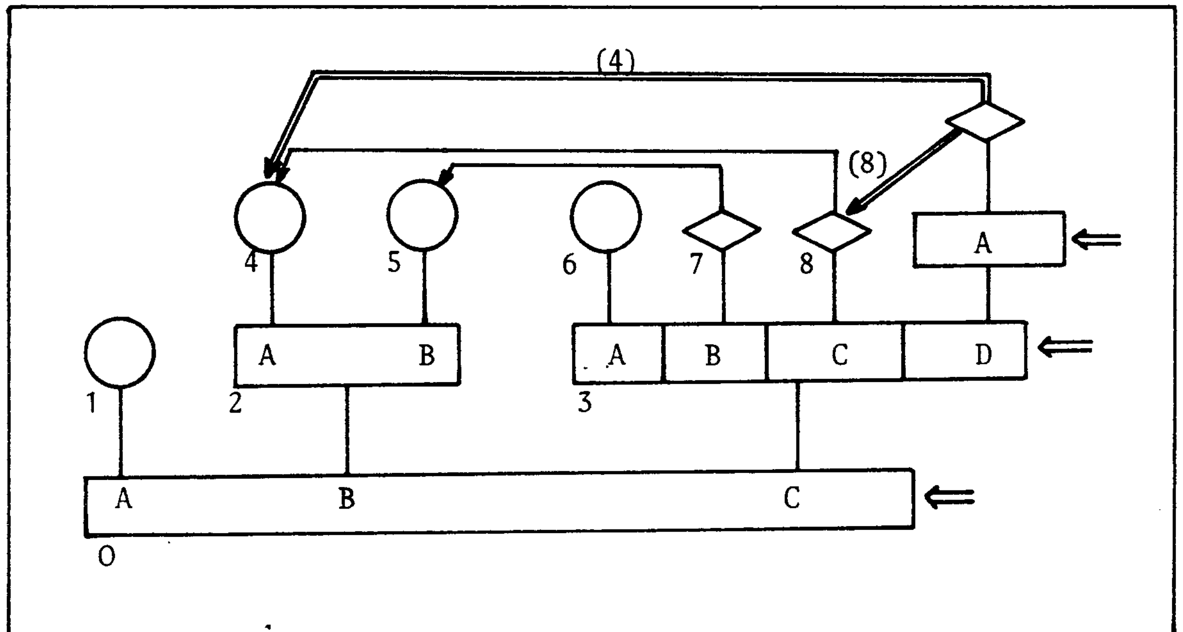


Figure 19

11.6. Définition

	< objet lien > ::= < type > = 'lien'	
	,< restriction >	
objet référencé {	,< nom universel >	partie résolue
	,< nom partiel >	

11.7. Problème de la destruction (ou du remplacement) des objets référencés au travers d'un lien

Considérons la figure 20.a. où les objets 7 et 8 sont des liens qui ont été créés par un processus situé dans l'environnement 0. La création s'est effectuée par :

CREATE(C,'B',(,'B.B'))

et

CREATE(C,'C',(B.B,))

ce qui a donné les valeurs 0.B.B et 4 aux valeurs des liens 7 et 8.

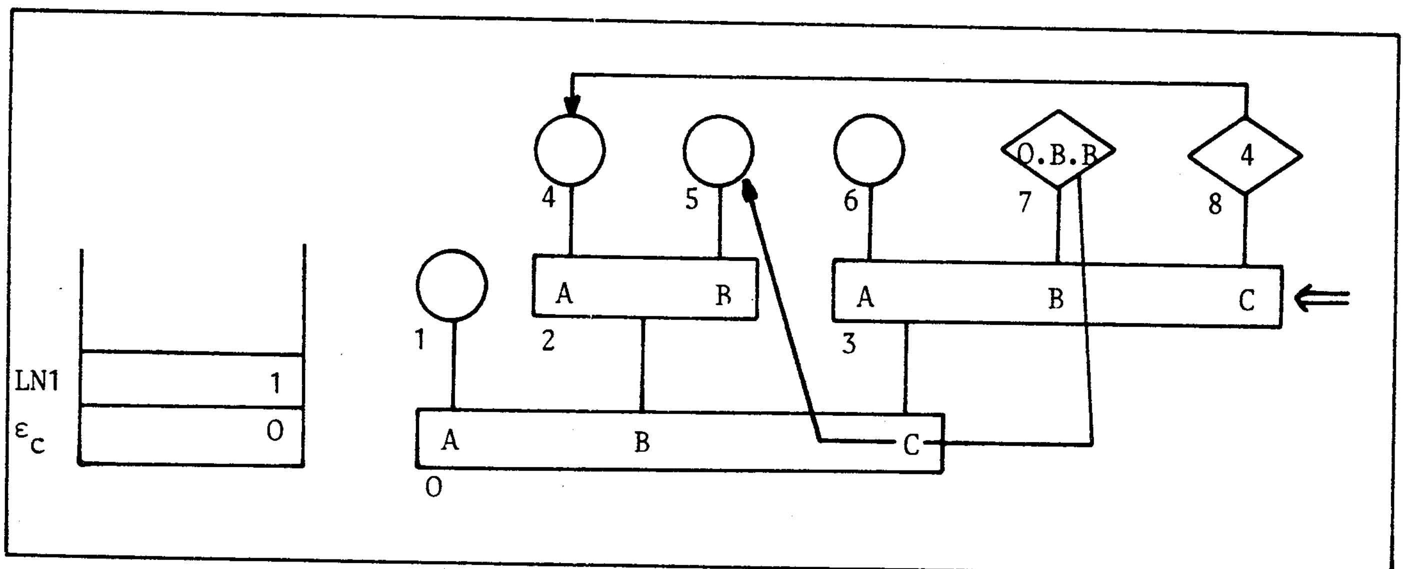


Figure 20.a.

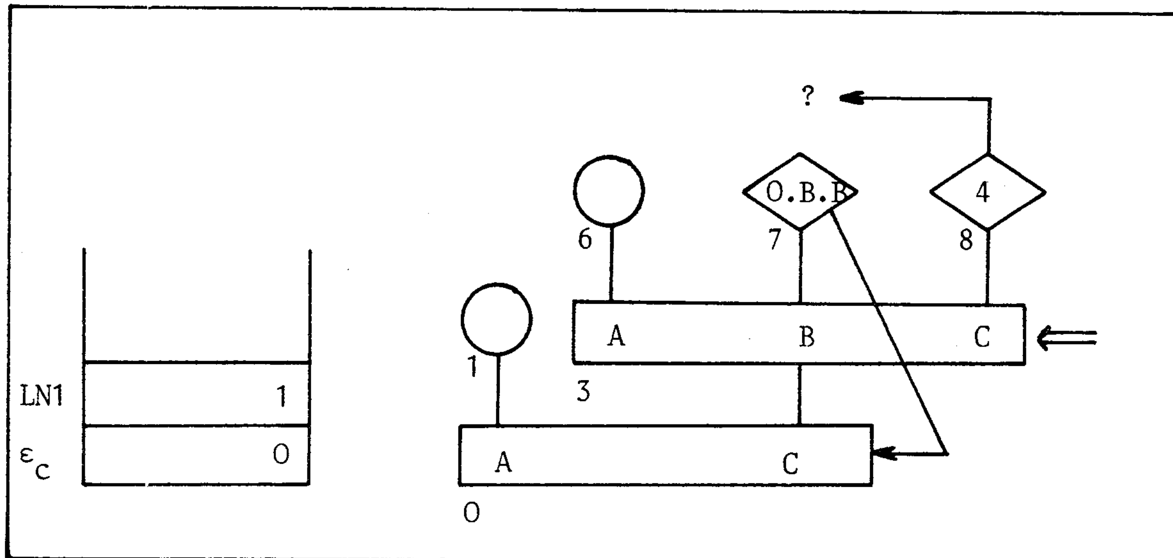


Figure 20.b.

Supposons que

DELETE(B)

détruit l'arborescence ayant comme racine l'annuaire 2, c'est-à-dire les objets 2, 4 et 5.

CALL(C.B)

retournera une erreur car si l'objet 0 est toujours présent, aucun objet de nom d'entrée B n'existe dans cet environnement.

CALL(C.C)

retournera aussi une erreur car l'objet de nom universel 4 n'a plus d'existence (figure 20.b).

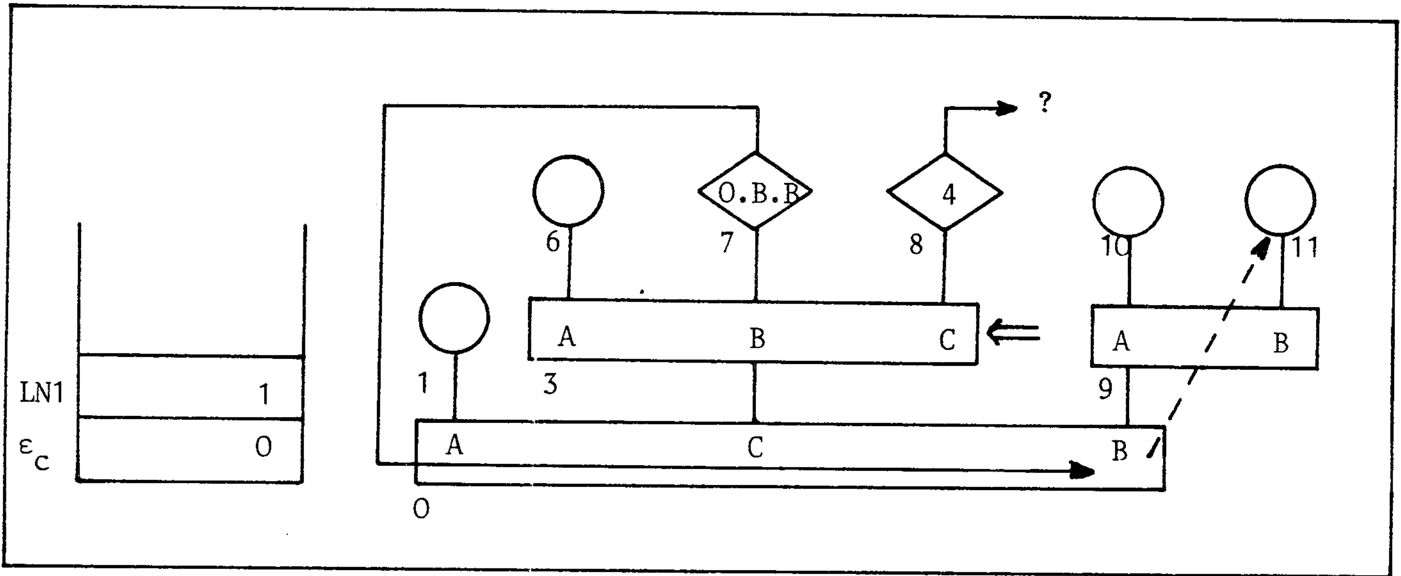


Figure 20.c.

Supposons qu'on ait ensuite :

CREATE(,'B',(type=annuaire)) → crée objet 9

CREATE(B,'A',(type=segment)) → crée objet 10

CREATE(B,'B',(type=segment)) → crée objet 11

alors (figure 20.c)

CALL(C.B)

entraîne l'exécution de l'objet 10, car l'objet 0 (environnement) existe et il existe aussi un nom partiel B.A dans cet environnement.

Cependant

CALL(C.C)

entraîne toujours une erreur.

Cet exemple appelle quelques remarques :

a. il y a une différence entre la liaison par l'intermédiaire d'un nom local et la liaison par l'intermédiaire d'un lien.

En effet, quand un objet est lié dans un nom local sa destruction physique est retardée, c'est-à-dire qu'il faut garder un compteur de liaison (nombre de noms locaux référençant cet objet) et à chaque libération d'un nom local, décrémenter ce compteur, et à chaque liaison (même après une destruction logique car on peut toujours effectuer LNi ← BIND(LNj)) incrémenter ce compteur. Dès que le compteur devient nul, l'objet peut être détruit physiquement. Alors qu'il n'y a pas de compte de liaison pour les liens.

On se reportera utilement au chapitre sur la protection pour une discussion de ce problème et en particulier pour voir les limitations entraînées par les choix de GEMAU.

b. Les noms universels sont uniques et ne peuvent absolument pas être créés ou modifiés directement par des procédures, sauf par l'intermédiaire des primitives fournies spécialement à cet effet (CREATE, DELETE, TRANSFER).

11.8. Boucles

Un des problèmes importants de tout système à indirections multiples est celui du bouclage, à savoir un enchaînement de liens qui pointent respectivement les uns sur les autres.

Plusieurs solutions à ce problème :

. conserver le nom universel de tous les liens rencontrés lors de l'évaluation d'un nom et vérifier à chaque fois que l'on trouve un lien que celui-ci n'a pas déjà été utilisé par cette évaluation ;

. compter le nombre de liens et, au-delà d'une certaine limite, décréter qu'il y a bouclage (principe de l'horloge de garde).

Dans le prototype nous avons choisi la deuxième solution.

11.9. Evaluation dans le cas des portes

Il faut pouvoir indiquer dans quel environnement le processus doit s'exécuter lors de l'accès d'une procédure au travers d'un lien.

Ceci est effectué par la définition d'un attribut 'porte' sur le lien.

Lors de l'évaluation d'un nom partiel, pour l'objet à exécuter il y a, chaque fois qu'une porte est passée, changement de l'environnement courant du processus. Quand l'objet final est atteint, la dernière porte passée fournit le nouvel environnement, c'est-à-dire celui qui sera utilisé pendant l'exécution du segment.

Considérons la figure 21 et supposons qu'un processus situé dans l'environnement 12 demande l'exécution de A.A par

CALL(A.A)

ceci va amener l'objet 3 à être exécuté dans LN1, le nouvel environnement étant 9.

En effet, l'évaluation suit les étapes suivantes :

- sauvegarder l'espace d'exécution courant
- $\varepsilon_c \leftarrow 12$ nouvel environnement courant = environnement actuel
- évaluer(12.A.A)
- évaluer(13.B.B.A) l'évaluation peut continuer car la restriction sur le lien autorise l'exécution
- $\varepsilon_c \leftarrow 11$
- évaluer(11.B.A)
- $\varepsilon_c \leftarrow 9$
- évaluer(9.A)
- LN1 ← 3, { CALL seulement}

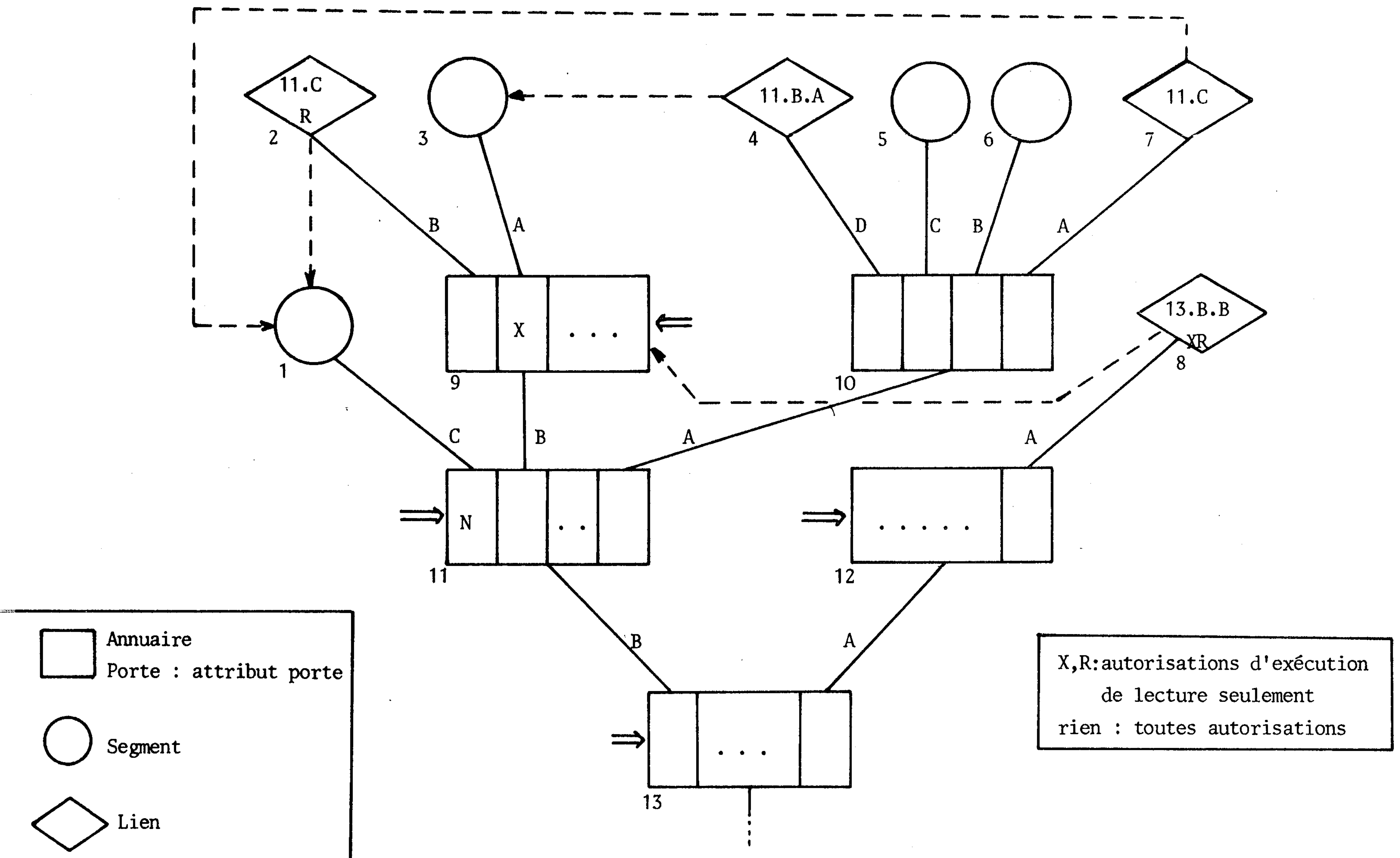


Figure 21

Exemple :

Reprenons l'exemple précédent en tenant compte des restrictions d'utilisation (environnement d'appel : 12)

CALL(A.A,A.B{ LIRE })

. Evaluation du nom de la procédure dans LN1 :
  est identique à celle de l'exemple précédent

. Evaluation du premier paramètre dans LN2 :

LN2 ← 12
evaluer(12.A.B)
évaluer(13.B.B.B)   peut être fait car LIRE est autorisé
LN2 ← 11{ LIRE }
évaluer(11.B.B)
LN2 ← 9 {LIRE }
évaluer(9.B)
évaluer(11.C)   peut passer car LIRE est autorisé
LN2 ← 1{ LIRE }

On voit dans cet exemple que l'objet 1 ne peut être accédé qu'en lecture depuis l'environnement 9, mais est sans protection depuis les environnements 11 et 13.

De même, les objets de l'annuaire 9 ne peuvent être que lus ou exécutés par un processus se trouvant dans l'environnement 12.

Le type de protection en utilisation considéré ici est non restrictif, le même mécanisme marche pour les protections sur accès de manipulation. Par exemple, pour un processus en 12 :

DELETE(A.B)

est refusé.

11.11. Destruction d'un lien

A partir des divers exemples précédents on a pu se rendre compte que les opérateurs (primitives) appliqués sur un lien considèrent ce lien comme une indirection et s'appliquant en fait sur l'objet terminal.

Ceci est vrai spécialement pour la primitive DELETE. En effet, l'ensemble des opérateurs définis jusqu'à présent ne permet pas de détruire un objet de type lien.

Si on considère la figure 22,

DELETE(B.A)

détruira l'objet 1 et non l'objet 3.

Il est donc nécessaire d'introduire une primitive spéciale de destruction d'un lien :

DELETELINK(< désignation de lien >)

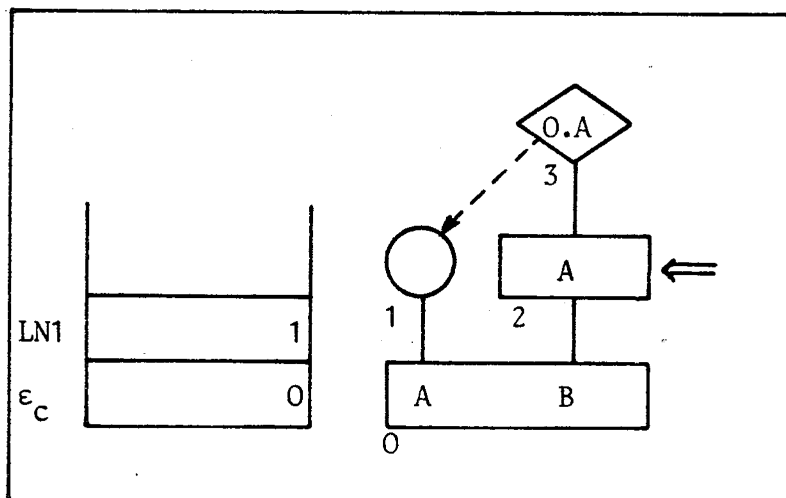


Figure 22

Pour reprendre la figure précédente
DELETELINK(B.A)
détruit l'objet 3 car c'est un lien.

Il en est de même pour modifier un lien :

MODIFYLINK(< désignation de lien > ,< description >)

et de même pour obtenir de l'information sur un lien :

GETINFOLINK(< désignation de lien >)

11.12. Transfert d'une arborescence contenant des liens

Considérons la figure 23.a. et supposons que le processus P, dans l'environnement 0, effectue :

TRANSFER(A,'A',B.B)

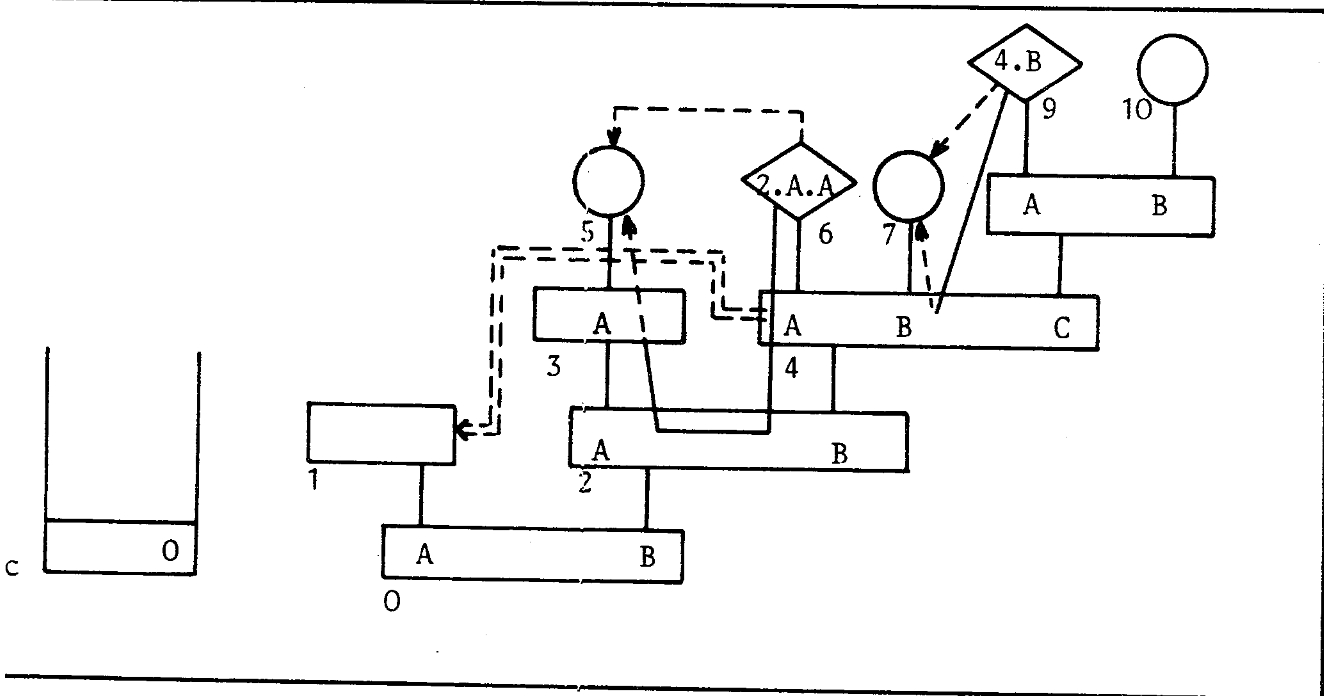


Figure 23 .a.

Les liens contenus dans l'arborescence transférée contiennent une référence à un objet et une référence symbolique. La référence résolue peut soit désigner un objet qui appartient à l'arborescence transportée (9), soit désigner un objet à l'extérieur de l'arborescence transportée (6). Le résultat final est donné par la figure 23.b.

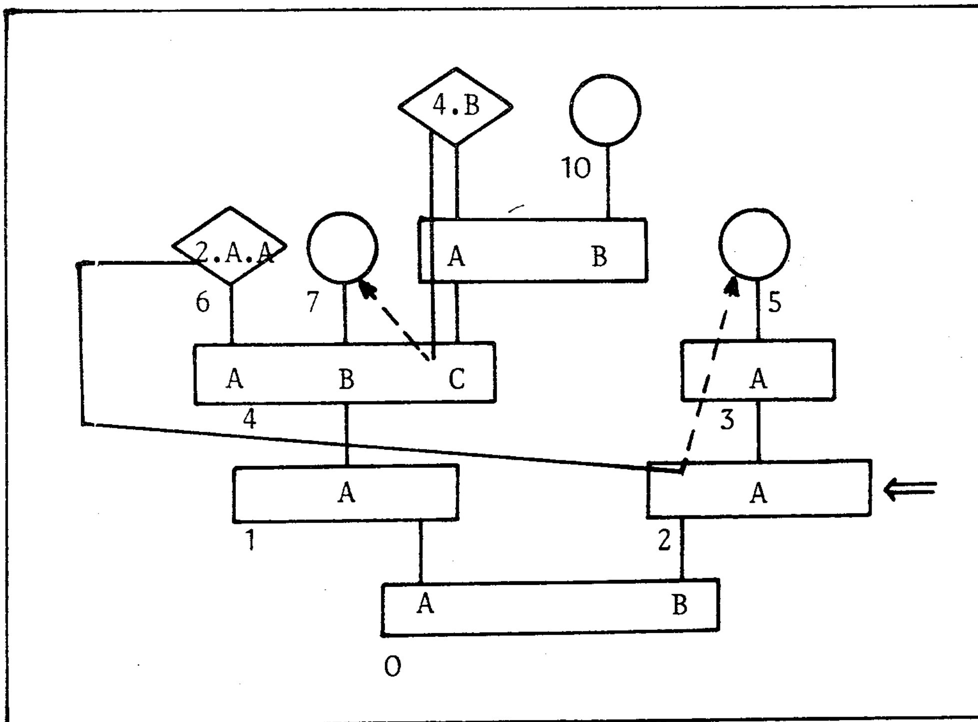


Figure 23.b.

Il est possible de parcourir l'arborescence pour modifier les liens référençant des objets extérieurs au sous-arbre, mais ceci serait en contradiction avec les propriétés désirées de la valeur d'un lien, à savoir que la partie résolue de la référence sert à désigner *une réalisation non substituable* d'un objet.

Ici c'est le cas du lien 6 qui, s'il est accédé, permet d'obtenir l'objet 5. Tous les mécanismes habituels des liens s'appliquent,en particulier les changements d'environnement éventuels (portes).

11.13. Lien implicite d'annuaire

La structure arborescente de résolution des noms partiels peut être trop rigide pour certaines applications et on peut désirer définir une certaine fonction de recherche des noms d'entrée.

Ceci est très fréquent et se trouve à des niveaux très différents dans les systèmes existants : 'bibliothèques' standard pour la résolution des références non résolues dans une édition de liens statique.

On peut aussi vouloir définir une structure de noms différents : par exemple, structure de bloc d'Algol.

Il suffit d'ajouter à un annuaire un lien (de nom prédéfini, par exemple nul) qui indique un annuaire dans lequel la recherche doit être effectuée si le nombre d'entrée cherché ne se trouve pas dans l'annuaire courant. Ce lien répond aux mêmes règles que les liens habituels, en particulier en ce qui concerne les restrictions sur l'objet référencé ; on peut ainsi par exemple n'autoriser que CALL sur des objets situés dans des annuaires référencés par des liens implicites. Si l'objet référencé n'est pas un annuaire, la recherche (lors de l'évaluation d'un nom partiel) s'arrête et tout se passe comme si le nom recherché n'existe pas dans l'annuaire initial.

Le lien est un lien vers un annuaire et non pas vers une procédure, car un annuaire est un objet primitif dans GEMAU ; dans le cadre d'extensions, cette restriction pourrait être levée avec l'introduction des objets construits (cf. chapitre 6, § 3.1.1.).

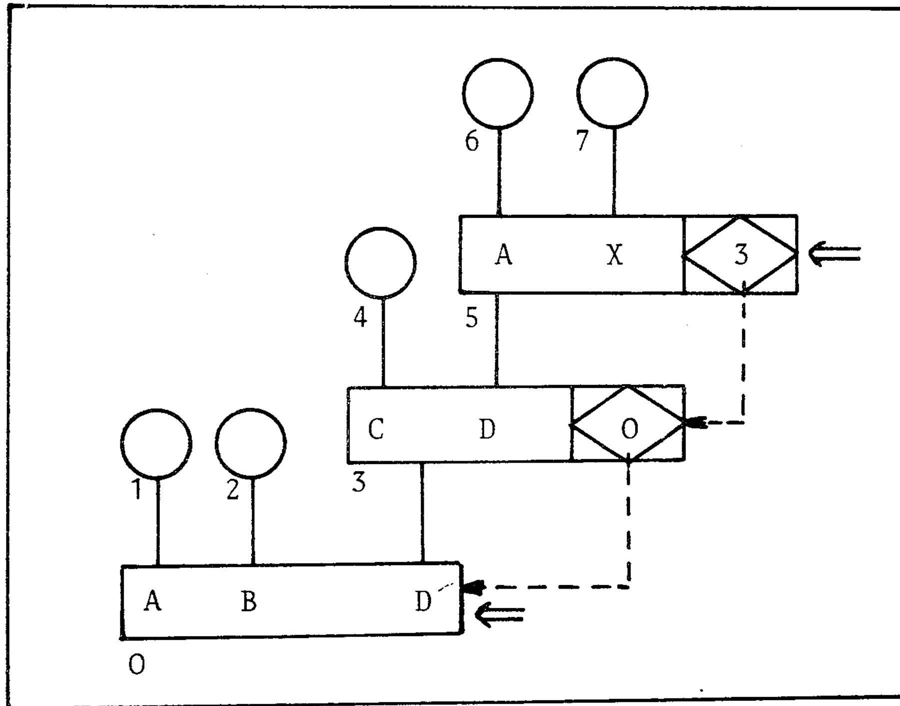


Figure 24

Considérons l'exemple donné par la figure 24. Un processus situé dans l'environnement 5 obtiendrait les objets suivants :

pour le nom partiel A : 6
" B : 2
" C : 4
" Y : refusé

Cet exemple représente une structure de bloc de type Algol. En effet, si nous prenons l'évolution du nom partiel B : il y a recherche dans l'environnement courant (5) et comme aucun objet de nom d'entrée B n'existe, le noyau effectue une recherche dans l'objet donné par le lien implicite (i.e. 3) et effectue la recherche dans cet objet qui doit être de type annuaire. Et ainsi de suite. Si aucun lien implicite n'existe, alors le noyau considère que le nom d'entrée n'existe pas.

Ces liens implicites permettent aussi de générer des boucles infinies, pour cela la détection effectuée par le noyau est identique à celle des liens normaux.

La figure 25 donne un exemple de boucle temporaire, c'est-à-dire qui disparaîtra quand un objet de nom d'entrée 'X' existera dans l'annuaire (sauf, éventuellement, s'il s'agit d'un lien créant lui aussi une boucle).

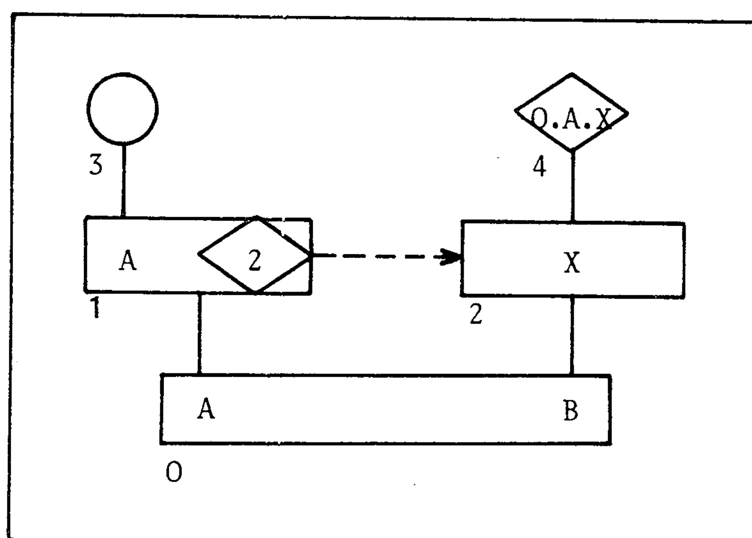


Figure 25

Les liens implicites augmentent aussi le nombre possible de noms partiels sur un objet. Par exemple, dans la figure traitant de la structure de bloc (figure 24), depuis l'environnement 0 l'objet 2 a les noms :

B,D.B,D.D.B

Ce lien possède tout simplement un nom standard dans l'annuaire qui le possède. Ce qui permet de le manipuler par les moyens existants, en particulier de le créer et de le détruire.

12. COHERENCE DES OBJETS

12.1. Définition du problème

Considérons un utilisateur à sa console en train d'éditer (mettre à jour) un fichier d'images de cartes. L'édition se compose de l'insertion de nouvelles cartes, de la destruction de certaines cartes et même de la modification du contenu de certaines autres cartes.

Dans notre terminologie, un fichier est un objet nous supposons qu'il est réalisé par un segment. Ce fichier comporte donc une structure interne avec un certain nombre de variables indiquant l'état du fichier. Nous supposons aussi que la structure des enregistrements est une structure de liste de façon à faciliter l'insertion et le retrait des différents enregistrements.

Pour opérer le retrait d'un enregistrement, il faut que la procédure d'édition mette à jour les pointeurs avant et arrière du fichier et ceci en au moins deux temps : mise à jour des pointeurs avant puis arrière. Durant cette mise à jour, l'état du fichier n'est plus cohérent.

En effet, les modifications du fichier (pour une mise à jour, par exemple) se traduisent généralement par une série d'instructions élémentaires. En particulier si les objets sont réalisés à l'aide de mécanismes de pagination, certaines parties de l'objet peuvent être dans des états différents sur les différents supports "mémoire".

Dans le cas où l'objet est utilisé en simultanéité, le degré de multiprogrammation doit permettre d'assurer que deux utilisations successives ont lieu sur un objet cohérent. Ainsi, dans tous les cas (mono-utilisateur ou multi-utilisateurs avec degré de multiprogrammation égal à un), on se ramène au cas de la monoprogrammation.

Si de plus nous supposons que le système s'arrête pour une cause matérielle ou logicielle, plusieurs hypothèses sont alors possibles :

. l'arrêt intervient entre deux modifications de l'objet, sa valeur est toujours définie,

. l'arrêt intervient pendant le déroulement d'une opération sur l'objet, la valeur de ce dernier peut toujours être cohérente, mais ceci ne peut être garanti.

Le problème découle du fait qu'en général, après une erreur, le système repart dans un état sûr, qui est éventuellement différent de l'état au moment de l'erreur. Il y a donc, souvent, perte d'information.

Ne pouvant reprendre le traitement exactement là où il se trouvait au moment de l'arrêt, il faut déterminer la dernière valeur correcte de l'objet. Ce qui veut dire que, pour des raisons de fiabilité, il faut conserver les différentes valeurs correctes, nous appelons chacune de ces valeurs *une valeur discrète* de l'objet.

12.2. Valeurs discrètes

Ainsi que nous venons de le voir, un objet possède un ensemble de valeurs discrètes dans le temps. Le problème posé précédemment ne peut être résolu que si chaque modification de la valeur d'un objet modifie non pas la valeur discrète, mais une copie de cette valeur discrète. Cette copie est appelée *valeur courante* de l'objet. A chaque instant, nous pouvons posséder deux images différentes d'un objet : une *image permanente* (la valeur discrète) et une *image temporaire* (la valeur courante).

A la fin d'une opération de modification de l'objet, la valeur courante devient la nouvelle valeur discrète, alors que l'ancienne disparaît. On peut établir une analogie avec les différentes versions d'un fichier dans les systèmes classiques.

D'où la division des espaces d'objets en deux sous-ensembles : l'espace permanent qui contient les valeurs discrètes des objets et l'espace temporaire qui contient leurs valeurs courantes. La figure 26 schématise cette distinction en deux espaces.

Les objets temporaires n'ont pas de valeurs discrètes dans un espace permanent car ils n'existent que pour une activation donnée, l'espace temporaire est réinitialisé à chaque redémarrage.

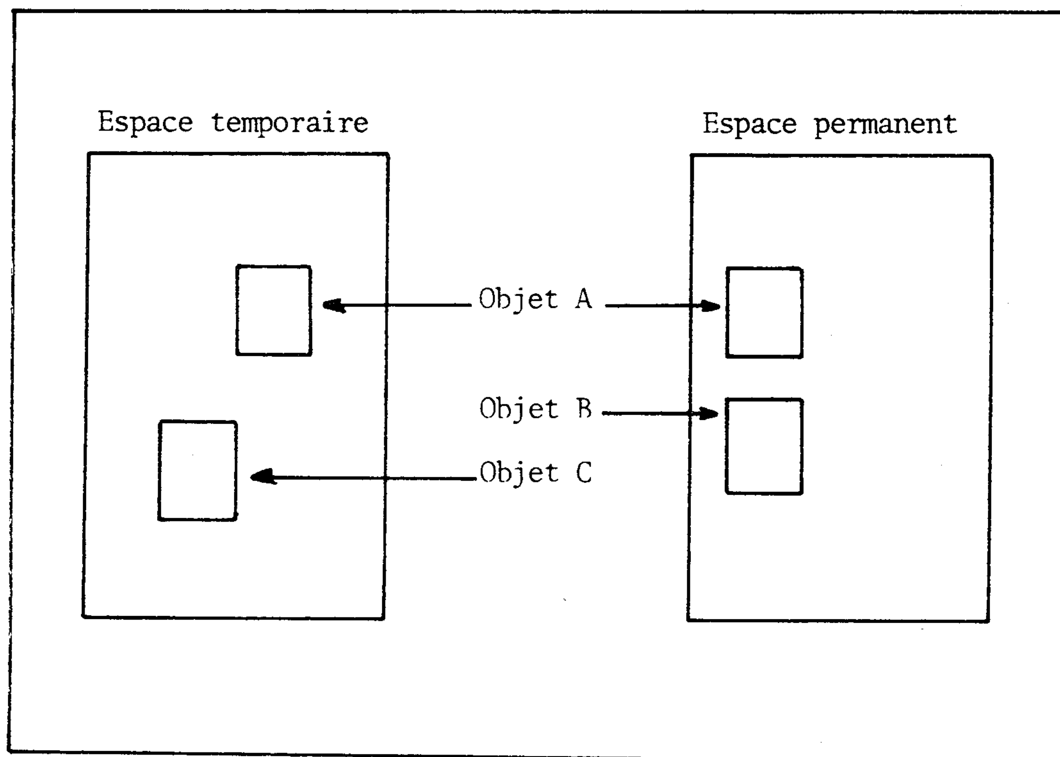


Figure 26 : Espaces permanent et temporaire

12.3. Opérateurs de mise à jour

Il existe deux primitives, SAVE et QUIT, qui permettent de mettre à jour la valeur discrète.

SAVE([< désignation d'objet >]...)

définit comme nouvelle valeur discrète (permanente) des objets désignés l'image temporaire de ces mêmes objets.

QUIT([< désignation d'objet >]...)

supprime la valeur temporaire des objets désignés. Cette primitive permet de repartir de la valeur discrète, c'est-à-dire que toutes les modifications de l'objet contenues dans la partie temporaire sont oubliées.

Ces deux primitives permettent une mise à jour simultanée de plusieurs objets.

La question qui se pose est de savoir quand effectuer une des deux opérations SAVE et QUIT, et par corollaire qui du noyau ou de l'utilisateur (i.e. système, sous-système, ..) prend la décision.

Pour l'instant de mise à jour, il semble naturel que celle-ci s'effectue en liaison avec les modifications de l'espace d'exécution, c'est-à-dire lors d'opérations BIND ou FREE, lors d'appel ou de retour d'exécution (CALL ou RETURN). On pourrait très bien imaginer des primitives FREE et RETURN dans lesquelles on indiquerait pour chaque objet s'il faut effectuer SAVE, QUIT ou ne rien faire. Nous avons choisi de laisser l'utilisateur (i.e. le système, le sous-système, ..) libre de décider et donc ces opérations doivent être effectuées *explicitement*, pour des raisons de simplicité.

13. DISPONIBILITE DES SOUS-SYSTEMES

13.1. Notion de volume

Chaque objet (segment, annuaire, périphérique, ..) est situé dans un espace mémoire permanent, le nom universel donnant un moyen de trouver l'emplacement dans cette mémoire.

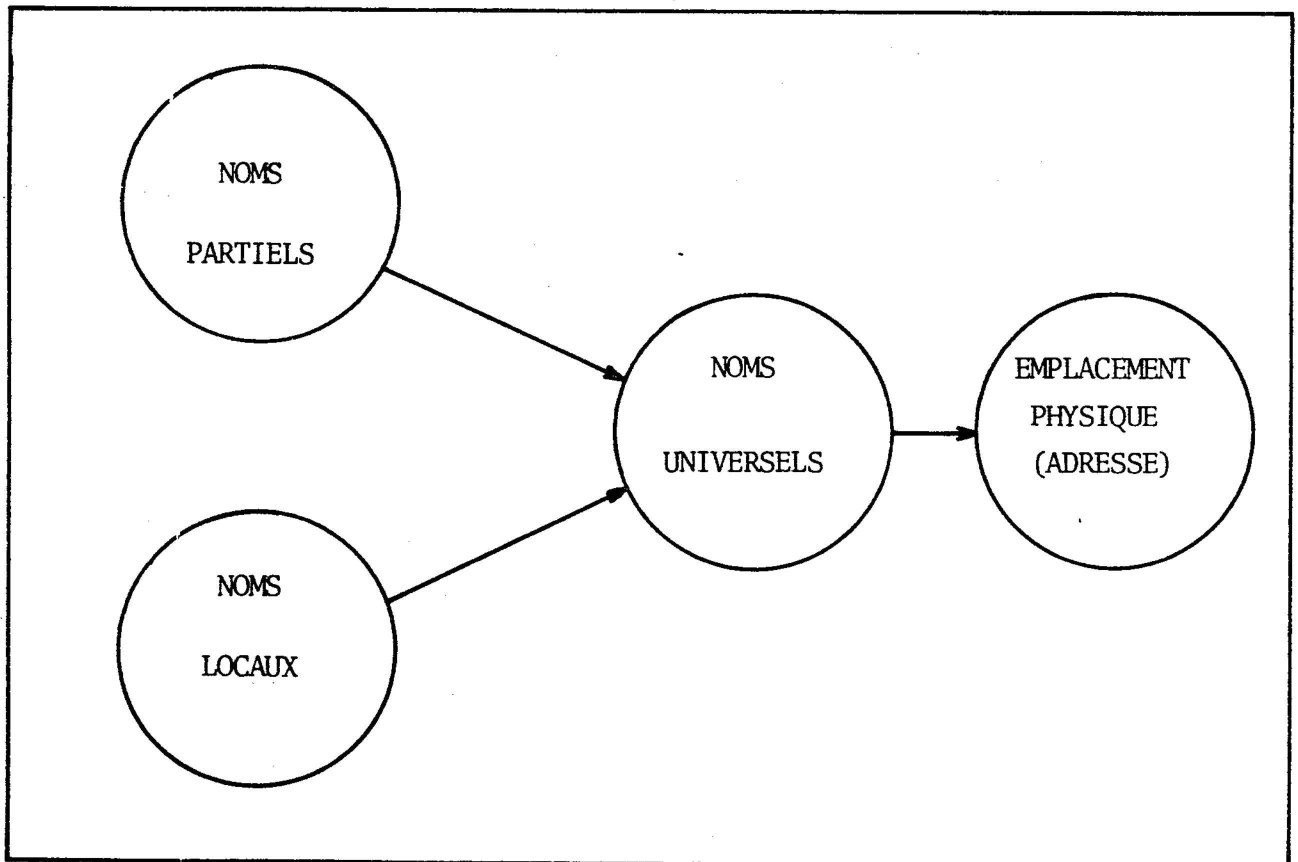


Figure 27 : Chaîne des différents noms d'un objet

La mémoire permanente peut être réalisée de façons extrêmement variées et c'est le rôle du noyau de définir la hiérarchie éventuelle de cette mémoire. Le noyau gère donc l'accès à l'information. Cependant, il ne faut pas oublier que nous voulons faciliter la construction de sous-systèmes et donc permettre à ces sous-systèmes de définir éventuellement des politiques d'allocation variées. L'utilisateur (système, sous-système, ou autre) du noyau doit donc pouvoir choisir le lieu de résidence de ses objets, par exemple que tous les fichiers soient sur un même support.

Ceci conduit à la notion de *volume* qui est, du point de vue de la mémoire secondaire, identique à la notion de segment. Un volume est un objet et est donc adressable en tant que tel, sa valeur est un ensemble d'autres objets. L'espace mémoire permanente est donc maintenant multidimensionnel.

Ainsi une adresse en mémoire permanente est-elle :

< identification de volume >, < déplacement dans le volume >

Une définition plus précise d'un objet volume est donnée par :

```
< objet volume > ::= < type > = 'volume'
                   , < taille >
                   , < identification symbolique >
< identification symbolique > ::= chaîne de caractères
```

Lors de la création d'un objet nous imposons la contrainte qu'un objet de type annuaire (ou environnement) doit contenir une liste des volumes sur lesquels les objets décrits dans l'annuaire (ou l'environnement) devront obligatoirement résider.

Si cette liste n'est pas fournie, il s'agit alors de la même liste que celle de l'annuaire père de l'annuaire en question.

L'adressage, par l'utilisateur, des volumes répond au schéma général d'adressage. Lors de la création d'un objet, il peut spécifier le volume d'implantation s'il possède l'objet de type volume correspondant dans son espace visible (espace d'exécution ou d'objets).

13.2. Manipulation des volumes

Considérons l'espace d'objets donné par la figure 28.a. Nous avons représenté cinq objets de noms universels 0 à 4 et respectivement de type : environnement, segment et volumes (2, 3, 4). L'environnement 0 est situé sur le volume VOL0.

L'opération, par un processus situé dans 0,

```
CREATE(LNO,'DIR',(type='annuaire'
                  ,attribut='porte')
                  ,volume=VOL1)
                 ,volume=VOL1)
```

crée un environnement (5) de nom d'entrée DIR et situé sur le volum VOL1 (i.e. de nom universel 2). Tout objet créé à partir de DIR(5) sera sur le volume de nom universel 2, sauf spécification contraire.

Ainsi, pour le même processus, les opérations

```
   CREATE(LNO.DIR,'Y',(type='segment'))
et CREATE(LNO.DIR,'Z',(type='segment'),volume=LNO.VOL2)
```

créent successivement les segments 6 et 7 sur les volumes respectifs 2 et 3.

Cependant, un processus situé dans 5 ne peut pas créer un objet hors du volume 2, car il ne peut nommer aucun volume du système (il n'en possède ni dans son espace d'objets, ni dans son espace d'exécution).

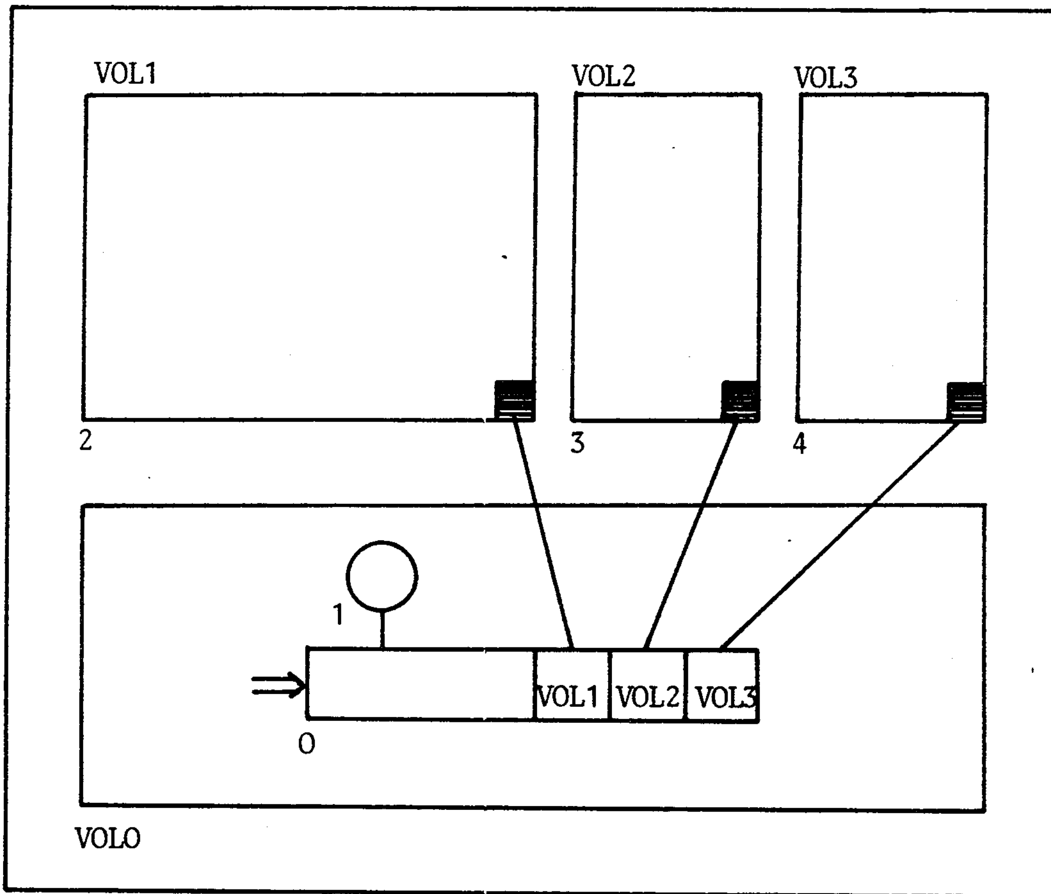


Figure 28.a. : Volumes

Cependant, le processus situé dans 0 peut transmettre à l'environnement 5 un lien vers le volume 4 par

CREATE(LNO.DIR,'T',(LNO.VOL3))

dans ce cas, à partir de l'environnement 5 on peut créer des objets situés soit sur 2 soit sur 4, par

CREATE(LNO.,'X',(type=...),volume=LNO.T)

La figure 28.b. récapitule les différentes opérations indiquées ci-dessus et donne les volumes sur lesquels les différents objets sont implantés.

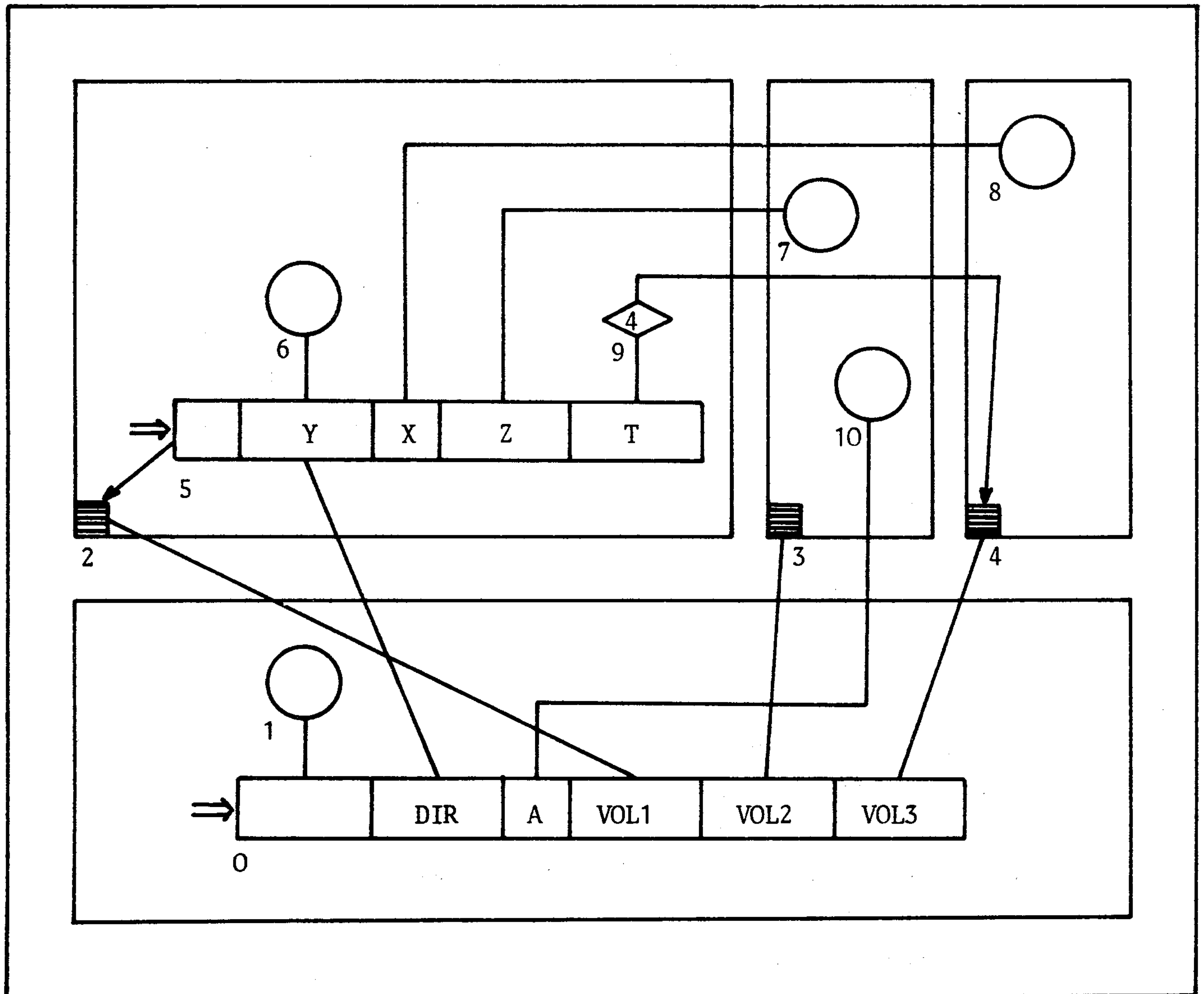


Figure 28.b. : Exemple d'utilisation de volumes

13.3. Définition finale de la primitive CREATE (cf. § 4.1.)

```
CREATE( < désignation simple d'annuaire >
        , < nom d'entrée >
        , < désignation d'objet >
        [, < désignation de volume >])
```

13.4. Définition finale d'un objet annuaire (cf. § 9.1.)

```
< descriptif d'annuaire > ::= < type >= 'annuaire'
                              , < restriction d'utilisation > =
                                CREATE/DELETE/TRANSFER/GETINGO/MODIFY
                              , < taille >
                              [, < désignation de volume >]  ...
```

Quand on crée un annuaire (ou un environnement), on peut spécifier une liste de volumes sur lesquels les objets devront être implantés.
La création d'un objet de cet annuaire s'effectue donc, sauf contre-ordre, sur un des volumes spécifiés et dans l'ordre de spécification, tout en gardant la contrainte qu'un segment doive résider entièrement sur le même volume.

13.5. Création d'un volume

La création d'un objet de type volume va préparer le formattage de l'élément (disque, tambour) ; il faut donc indiquer un périphérique sur lequel se trouve l'élément à formatter. Cette opération n'est pas une opération courante, en particulier il faut prendre toutes les précautions possibles lors du formattage pour ne pas détruire un volume existant.
L'identification symbolique du volume correspond à l'idée classique de "label".

13.6. Remarques

Les notions avancées ici appellent quelques commentaires :

. le mécanisme de volume s'intègre parfaitement dans la structure d'adressage des objets ;

. les espaces physiques ne sont pas imbriqués, contrairement aux machines récursives (cf. Chapitre 4, § 2.5.), bien que l'espace d'objets ait une structure hiérarchique.

Chapitre 3

EXEMPLE D'UTILISATION DE GEMAU

SOUS-SYSTEMES DE GESTION D'ENTREES-SORTIES

Nous définissons plus en détail les objets de type périphérique et nous montrons comment il est possible de réaliser, à l'aide des objets et opérateurs de GEMAU, des sous-systèmes de gestion d'entrées-sorties.

1. DEFINITION DES ENTREES/SORTIES.

1.1. Classification des périphériques.

Les ressources en organes d'entrée-sortie d'une machine peuvent être divisées en deux classes :

a) La première classe est réservée au noyau et sert au fonctionnement et partage de la machine entre les différents processus ; C'est en particulier le cas de la mémoire secondaire qui sert à réaliser l'arborescence (annuaire) et à conserver les segments.

Cette classe de ressources d'entrées-sorties est inconnue des utilisateurs et sa gestion est effectuée par le noyau en fonction de fréquences d'utilisation et d'un certain nombre d'autres critères (en général d'optimisation). La localisation des objets conservés sur cet espace est en général ignorée des utilisateurs, et il est cependant intéressant de pouvoir indiquer qu'un objet où un ensemble d'objets, sont à un endroit spécifique, ainsi que nous l'avons vu (c'est la notion de *volume*).

b) La deuxième classe est donnée à l'utilisateur, ce sont en particulier les objets de type périphérique, ces objets sont définis dans un environnement donné, ce qui dépend essentiellement du système initial et des processus utilisant ces périphériques ; Leur utilisation est effectuée à l'aide de la primitive CALL.

Désormais le terme *périphérique* s'applique seulement aux objets de cette classe.

1.2. Définition des périphériques "utilisateurs".

Il est très important de remarquer que la notion de fichier traditionnel disparaît dans une optique GEMAU, c'est-à-dire qu'un fichier est une structure de données et est représentée uniquement dans un segment. Le fichier traditionnel en tant que mémoire de manoeuvre avec des opérateurs d'accès particuliers, n'a plus de raison d'être, car ces opérateurs font intervenir explicitement des transferts d'information. Cependant l'aspect de périphériques en tant que moyens de communiquer avec le monde extérieur (terminaux, lecteurs de cartes

imprimantes, etc..) subsiste, et il n'y a aucune raison pour que ces périphériques ne soient utilisables directement par des programmes utilisateurs. Ce ne sont pas uniquement des périphériques standarts car ils peuvent être aussi de n'importe quel type (périphériques spéciaux, bandes magnétiques, disques magnétiques, rubans perforés, tables traçantes, etc...)

Le noyau ne s'intéresse aucunement à la gestion logique des périphériques donnés à l'utilisateur (i.e décrits dans l'espace d'objets), et en particulier il doit être possible de piloter n'importe quel périphérique à l'aide d'un programme utilisateur (i.e. hors du noyau). Le seul rôle du noyau est d'effectuer les entrées-sorties physiques, donc de gérer les conflits d'accès à un périphérique et de fabriquer les programmes canaux nécessaires. Cette position que nous prenons vis à vis des périphériques permet d'ajouter un nouveau type de périphérique à la machine avec un minimum de modification dans le noyau, car la manipulation effective de ce périphérique est faite par des programmes utilisateurs. Cette position est imposée par les arguments suivants :

a) les périphériques sont considérés comme des objets sur lesquels les opérateurs applicables sont :
   - INPUT (transférer depuis le périphérique vers un segment de données)
   - OUTPUT (transférer depuis un segment de données vers un périphérique).
   - ordre spécial (propre à certains périphériques tels que : rebobiner, espace arrière...)
   - HALT

   Ces opérateurs sont répartis en deux classes : celle des opérateurs qui entraînent des transferts de données entre un (ou des) segment(s) et un périphérique,et celle des opérateurs qui changent l'état d'un périphérique. Les opérations de transfert demandent la spécification d'une zone de mémoire, où ranger ou prendre les données. Cette zone est définie de façon virtuelle (c'est-à-dire en terme de déplacement dans un segment). C'est le rôle du noyau de transformer ces demandes en programme-canaux avec les contraintes physiques de la machine, en particulier la présence en mémoire centrale de la zone au moment de l'entrée-sortie et la gestion des conflits d'accès sur la configuration d'entrées-sorties. La notion physique de chemin d'accès (canal, unité de contrôle, ...) a complètement disparue.

b) Les périphériques étant des objets situés dans l'espace d' objets sont donc soumis aux mêmes règles que tous les autres objets, en particulier pour la définition de leur visibilité et de leur protection, ainsi que de la visibilité et de la protection des objets contenant les zones de transfert d'entrée-sortie.

. Il est impossible à un processus ne possèdant pas effectivement un périphérique réel, de procéder arbitrairement à des entrées-sorties ;

. Il doit être possible de passer un objet de type périphérique d'un environnement à un autre, à l'aide d'un objet de type lien ou d'une copie ;

L'affectation de périphériques réels à un environnement résoud au niveau des sous-systèmes, le problème de l'allocation. Cette affectation dépend uniquement du sous-système et absolument pas du noyau. En particulier, si un sous-système donne deux liens dans deux environnements différents (utilisateurs) vers le même périphérique,des processus différents peuvent effectuer des entrées-sorties séquentielles sur ce même périphérique. Pour des raisons de spécificité d'utilisation, un périphérique de type disque, imprimante ou lecteur de cartes, doit pouvoir être donné directement à un utilisateur (ou sous-système).

c) Le traitement spécifique des erreurs apparaissant durant l'utilisation d'un périphérique est laissé au programme utilisateur, car lui seul manipule le périphérique. Cependant, dans le cadre de l'utilisation d'un sous-système, ce sous-système peut fournir, à titre de service, les procédures de support de ces périphériques et de récupération de leurs erreurs.

Ce point permet aussi d'effectuer, depuis certains sous-systèmes, une maintenance et des tests de périphériques "en ligne".

d) Les périphériques doivent pouvoir être virtuels, c'est-à-dire que l'on doit pouvoir remplacer un périphérique réel par une procédure qui prend les mêmes paramètres d'appel et fournit (vue de la procédure appelante) un résultat similaire à celui d'un périphérique. En particulier, il n'y aura plus aucune différence pour un compilateur si son entrée s'effectue depuis un périphérique réel (terminal, lecteur de cartes, périphérique situé sur un réseau

d'ordinateurs)ou depuis une procédure simulant un périphérique (cf § 3). Ceci est vrai pour n'importe quel programme utilisateur ou système. En particulier on peut changer dynamiquement la réalisation d'un périphérique (réel ou virtuel) par substitution comme pour n'importe quel autre type d'objet.

La programmation d'entrées-sorties dans un segment de programme doit rester inchangée,que la valeur effective de l'objet pris pour périphérique soit une simulation de ce périphérique (virtuel) ou le périphérique lui-même (réel). C'est ainsi que la primitive CALL s'adressant au périphérique peut soit conduire à un appel de procédure (virtuel) soit être interprétée directement par le noyau (réel).

L'intérêt de la virtualisation des périphériques est multiple. Hormis l'impossibilité de l'utilisation simultanée d'un même lecteur de lecteur (ou imprimante) par plusieurs utilisateurs (cf. § 3.1 du présent chapitre sur les symbionts), il peut être très utile de simuler un périphérique. D'abord pour tester un programme et notamment les procédures d'erreur car il est plus facile de simuler des conditions anormales par logiciel qu'à l'aide du matériel. De plus, il est possible d'essayer des procédures de support de périphériques particuliers sans posséder réellement ces périphériques. De même, pour certains tests répétitifs on peut ainsi éviter de bloquer une ressource périphérique pendant un temps très long ; ainsi supposons que nous ayons écrit un programme qui traite des données en provenance d'un autre système, ces données sont passées par l'intermédiaire de bandes magnétiques. Durant toute la phase de mise au point du traitement il est inutile d'attendre qu'un dérouleur de bande soit disponible pour découvrir que le programme comporte une erreur après la lecture du premier enregistrement. L'utilisation d'un simulateur, au demeurant très simple, de bande magnétique contenant suffisamment d'enregistrement permet d'avancer la phase de mise au point sans dépendre de l'utilisation de ressources physiques partagées pour lesquelles il existe souvent une pénurie importante.

e) Une demande d'entrée-sortie s'effectue comme un appel de procédure (CALL) et est donc toujours synchrone, c'est-à-dire qu'il n'y a pas de parallélisme entre un processus demandant une entrée (ou une sortie) et le déroulement de

de l'entrée (ou de la sortie) elle-même. Cette position s'explique simplement dans le cas de systèmes multi-utilisateurs car elle permet d'effectuer le parallèlisme à un niveau plus fin, c'est-à-dire au niveau du noyau. De plus la programmation d'entrées-sorties synchrones est infiniment plus simple que celles des entrées-sorties asynchrones.

Cependant, cette position n'interdit absolument pas la programmation des asynchronismes par création de processus parallèles effectuant eux-même l'entrée-sortie de façon synchrone, et provoquant l'occurence d'un évènemen lors de l'achèvement de cette entrée-sortie. Dans ce cas on remplace l'appe CALL par un appel FORK (cf. [BR 74]). Une autre solution qui utilise la nc tion de symbiont est décrite dans ce chapitre.

2. PROGRAMMATION DES PERIPHERIQUES

Un objet de type périphérique est uniquement exécutable (opérateur CALL), il permet de réaliser un certain nombre de fonctions, comme : entrée, sortie, ordre spécial (par exemple : arrêt, rebobinage,...) et test d'état. Il est possible, comme pour les segments de protéger l'utilisation d'un périphérique au niveau de ces opérateurs,par exemple un périphérique peut être utilisable en entrée seulement (transfert depuis le périphérique vers un segment), toute sortie étant interdite. Chacune des fonctions demande un certain nombre de paramètres qui sont de deux types : segment et valeurs immédiates.

2.1. Entrée-sortie.

```
CALL (< désignation de périphérique>, <désignation de segment>)
('INPUT'/'OUTPUT',<déplacement>,<longueur>[,<divers>]...)
```

où

<déplacement> : déplacement (en caractères) depuis le début du segment (LN2) de la zone où ranger la donnée.

<longueur> : longueur de la zone tampon (i.e nombre de caractères à transférer).

<divers> : tous paramètres spécifiques à un périphérique donné. Par exemple pour un terminal ou un lecteur de cartes une indication de transcodage (lecture binaire, EBCDIC, ASCII), pour un terminal le caractère de fin de ligne, etc...

2.2. Fonctions spéciales.

```
CALL (<désignation de périphérique>)
     ('ORDER'/'HALT'/'TEST' [,<divers>]...)
```

ou

a) HALT : arrêt de toutes opérations sur le périphérique.

b) TEST : fournit comme résultat une valeur immédiate qui indique l'état actuel du périphérique (en mode manuel, en cours de transfert,...)

c) ORDER : opération spéciale sur le périphérique. Il faut un certain nombre de paramètres supplémentaires qui dépendent du type de périphérique. Par exemple dans le cas d'une bande magnétique :

. rebobinage.
. espace avant/arrière et nombre d'espaces désirés.
. saut d'un fichier en avant ou en arrière.
  (i.e jusqu'à une marque de bande).
. écriture d'une marque de bande.

2.3. Exemple (figure 1)

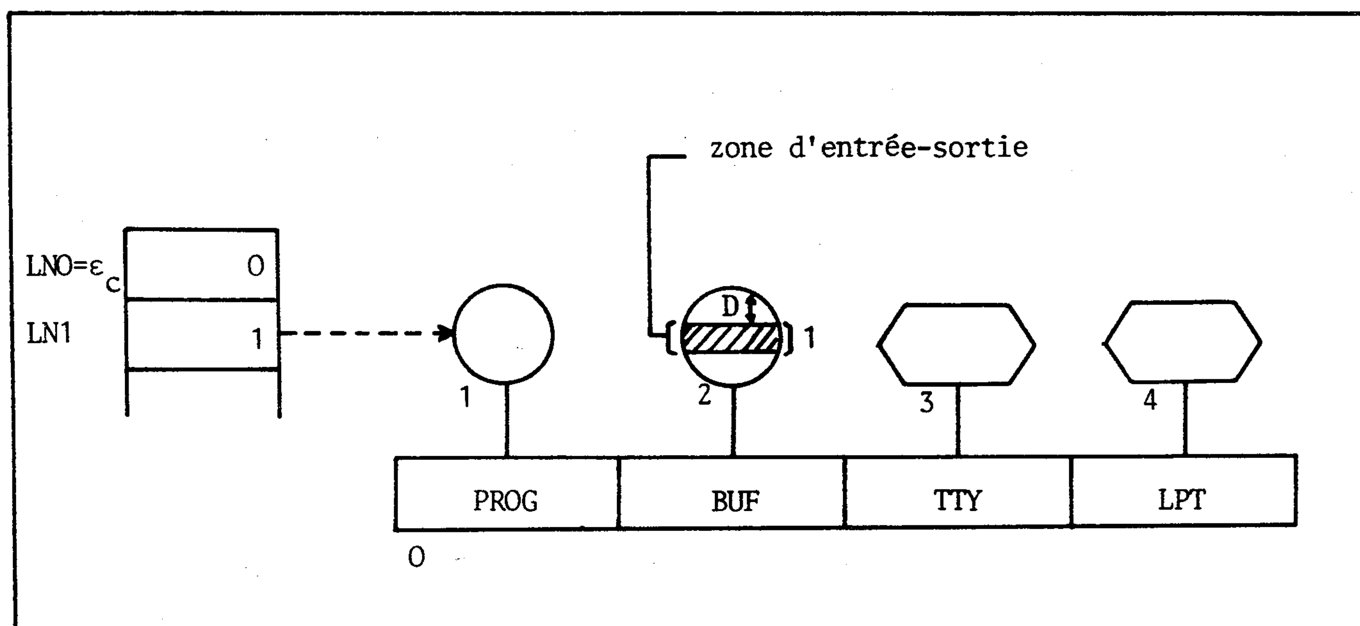


Figure 1 : Programmation des périphériques

Considérons la figure 1 où les objets 4 et 5 sont du type *périphérique*, TTY est un terminal et LPT une imprimante.

Une demande d'impression d'une zone du segment BUF sur LPT s'effectue par

CALL (LPT, BUF)('OUTPUT', D, 1)

où D est le déplacement de la zone par rapport au début du segment 2, et l la longueur . De même une demande de lecture depuis TTY s'effectue par :

CALL(TTY, BUF ('INPUT', D, l, '↲'))

Dans ce cas, le paramètre '↲' indique le caractère de fin de ligne en entrée.

Toutes les informations passées en valeurs immédiates sont vérifiées par le périphérique lui-même,i.e en dehors de la fonction,des déplacements et longueur qui sont vérifiés par le noyau (le tampon doit être entièrement dans le même segment), toutes les autres informations dépendent du type du périphérique.

Les règles habituelles sur la manipulation des objets s'appliquent ici, en particulier la liaison et la protection.

Par exemple :

LN2 ← BIND (LPT)

CALL (LN2, BUF) ('OUTPUT', D , l)

2.4. Récapitulatif sur les périphériques.

<objet périphérique> :: =<type>= 'périphérique'
,<restriction d'utilisation> =
INPUT/OUTPUT/TEST/ORDER/HALT
,<valeur>

2.5. Virtualisation d'un périphérique.

Considérons la figure 2

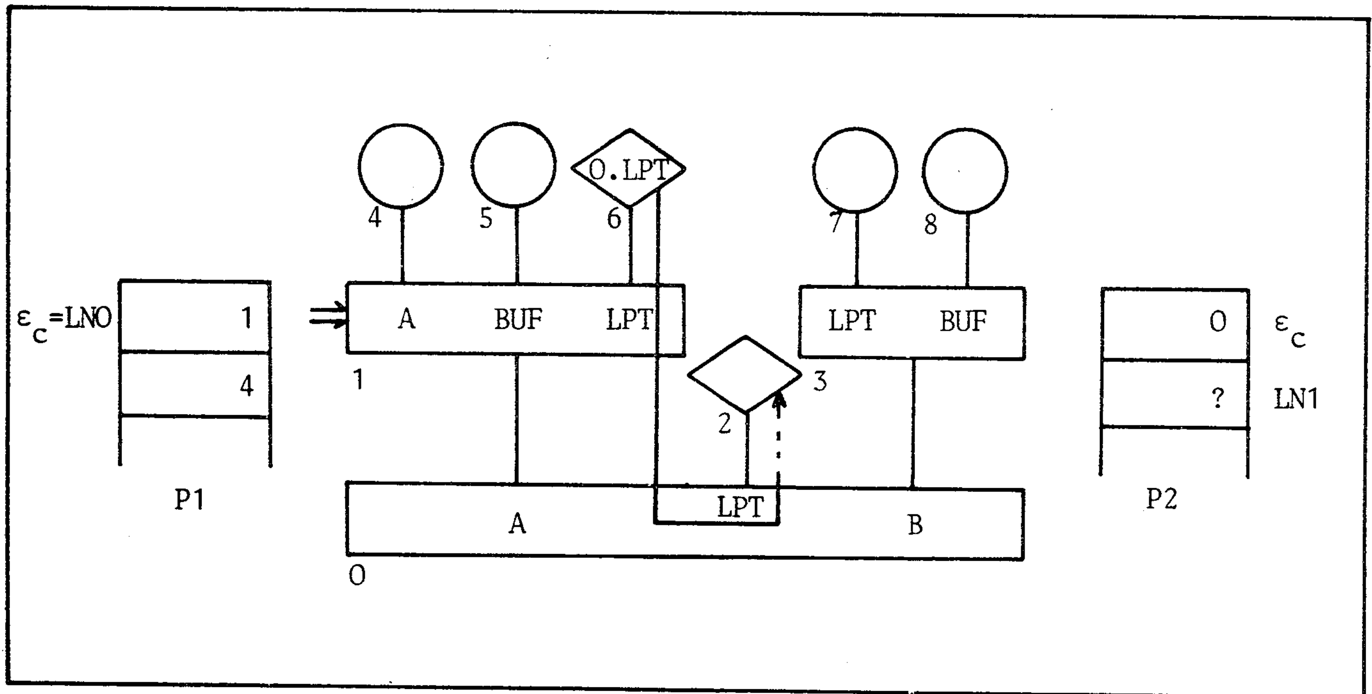


Figure 2 : Virtualisation des périphériques

où un processus P1 s'exécute dans la procédure 4, i.e dans l'environnement courant 1, dans cet environnement il existe un lien LPT (6) vers l'objet O.LPT (i.e un périphérique). Toute entrée-sortie sur 6 s'effectue par

CALL(LPT, BUF)('OUTPUT', D, 1)

Supposons qu'un deuxième processus P2 situé dans l'environnement O effectue

DELETELINK (A. LPT)
CREATE(A, 'LPT', (,'B.LPT'))

Ceci va remplacer la valeur interne du lien LPT de O.LPT par O.B.LPT.

On a le schéma donné par la figure 3

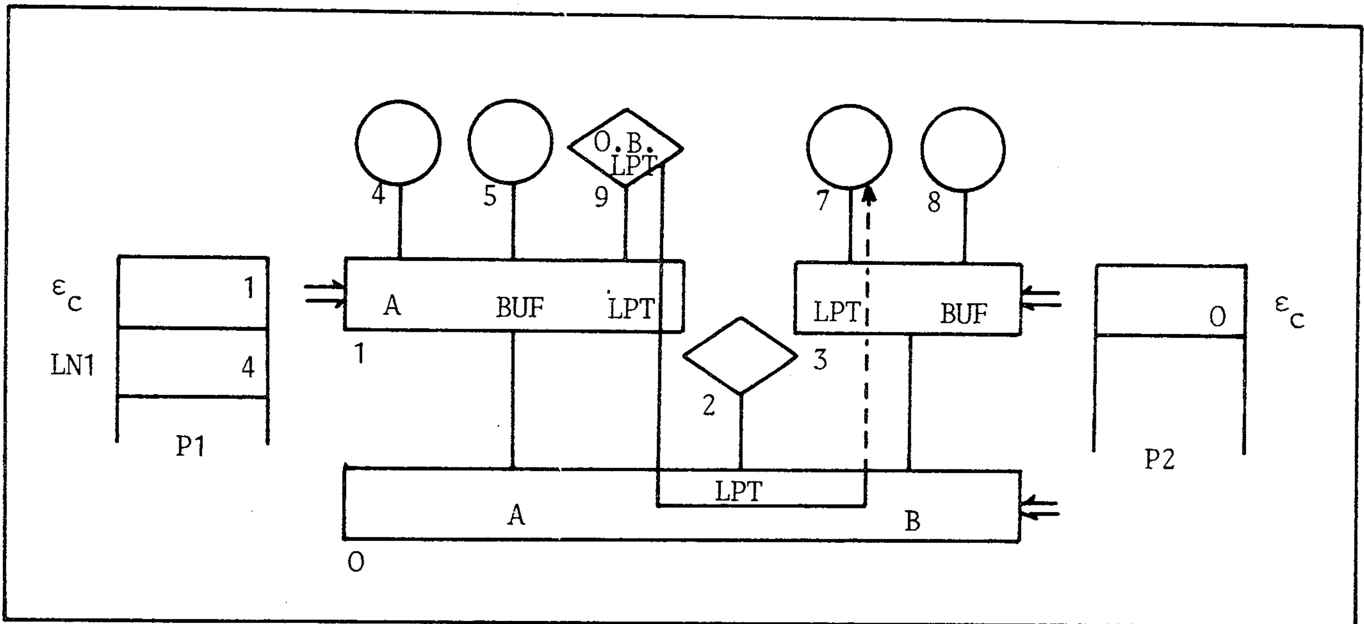


Figure 3

L'objet O.B.LPT est une procédure, cette procédure peut tout simplement effectuer l'opération de sortie, en, transférant l'information depuis l'objet BUF (qui sera pour elle le nom local LN2) vers l'objet 8 (BUF).

En effet

CALL(LPT,BUF)('OUTPUT', D, 1)

pousse l'espace d'exécution de P1, et fabrique un nouvel espace composé de LN1 (7) et de LN2 (5). La procédure 7 peut très bien effectuer

LN3 ← BIND(BUF)

qui affecte au nom local LN3 l'objet 8 et ensuite effectue un transfert depuis la zone (D, 1) de LN2 vers une zone de LN3.

Ceci est schématisé par la figure 4. Cependant toutes les vérifications quant à la validité de l'ordre ('OUTPUT'), et des autres paramètres (D, 1, LN2) doivent être effectuées par la procédure de simulation (ici 7).

Quand le transfert est effectué, la procédure se termine par

RETURN ( ) (informations)

en fournissant les valeurs immédiates indiquant l'acceptation ou refus de la demande 'OUTPUT', par exemple si D ou D + 1 sont supérieurs à la taille du segment contenant la zone tampon.

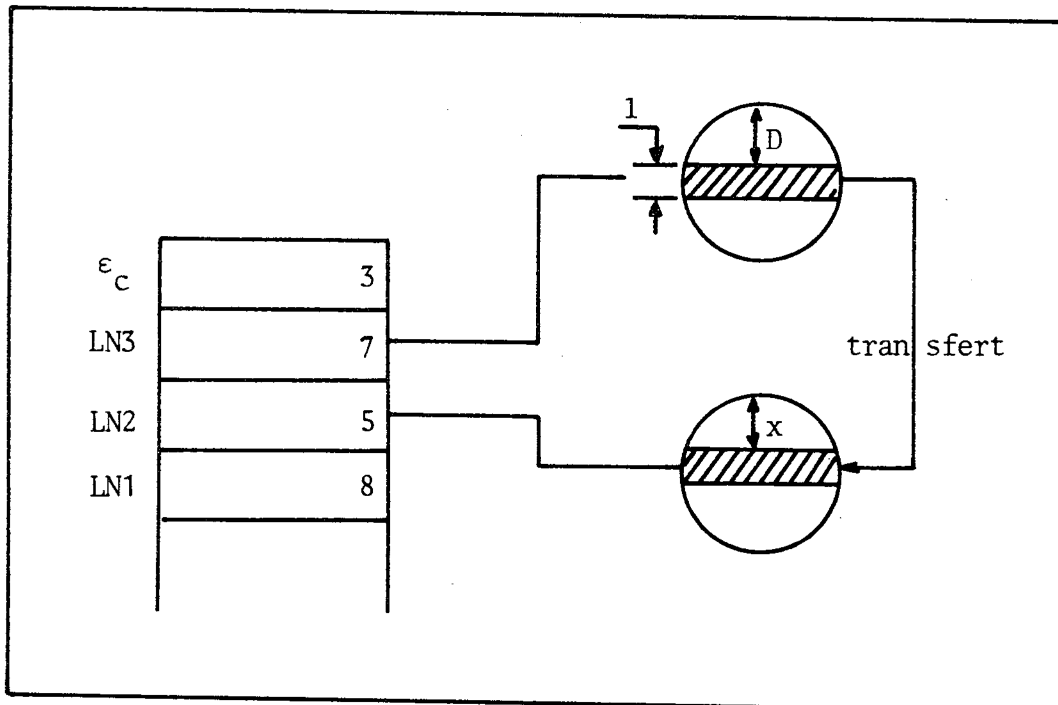


Figure 4

3. DEFINITION DES SYMBIONTS D'ENTREE-SORTIE

Dans un système multi-utilisateur à partage de ressources, il n'est en général, pas possible d'affecter certains périphériques à un utilisateur particulier. C'est le cas par exemple de périphériques lents du type lecteur (ou perforateur) de cartes ou du type imprimante. La raison principale est que la lecture (ou l'impression ou la perforation) se fait sur la base d'un enregistrement à la fois (carte, ligne), et non sur la base d'un ensemble d'enregistrements. Ce cas est semblable à celui des dérouleurs de bandes magnétiques, mais par contre, la solution généralement adoptée pour les périphériques lents est différente de celle choisie pour les bandes magnétiques. Ce que l'on utilise ce sont des périphériques virtuels tels qu'ils ont été définis précédemment. Ainsi prenons le cas d'une imprimante, chaque utilisateur peut disposer d'un (ou plusieurs) simulateur d'imprimante, et toute impression d'une ligne s'effectue par mouvement d'information entre segments comme nous l'avons vu au paragraphe précédent. Cependant, quand un utilisateur a terminé son impression, il faut sortir effectivement la liste de lignes sur une imprimante réelle (ou éventuellement tout autre type de périphérique). Cela peut être fait soit par le processus de l'utilisateur, soit par un processus spécial appelé *symbiont*. La raison du choix d'un processus spécial réside dans le fait que les activités de l'utilisateur peuvent se dérouler parallèlement à l'impression des listes. Dans ce cas, il faut définir une file d'attente où les processus utilisateur déposent les imprimantes virtuelles prêtes à être imprimées physiquement, et dans laquelle le processus d'impression vient chercher une imprimante virtuelle dès qu'il a terminé l'impression correspondant à une autre imprimante virtuelle. Il est possible d'avoir autant de processus 'symbiont' que de périphériques réels. Le problème traité ici est connu sous le nom des "producteurs-consommateurs" et a donné lieu à un certain nombre d'études, particulièrement sur la synchronisation. La solution en est classique, c'est un des exemples que nous traitons dans ce chapitre dans le cadre de GEMAU. En particulier, nous étudierons les relations d'adressage. La figure 5 schématise ce problème.

Dans cette figure les deux processus P1 et P2 déposent des lignes dans des imprimantes virtuelles respectives, puis déposent une demande d'impression physique dans la file d'attente FA. Nous verrons en particulier les problèmes de méfiance réciproque importants qui se posent dans le cas des symbionts.

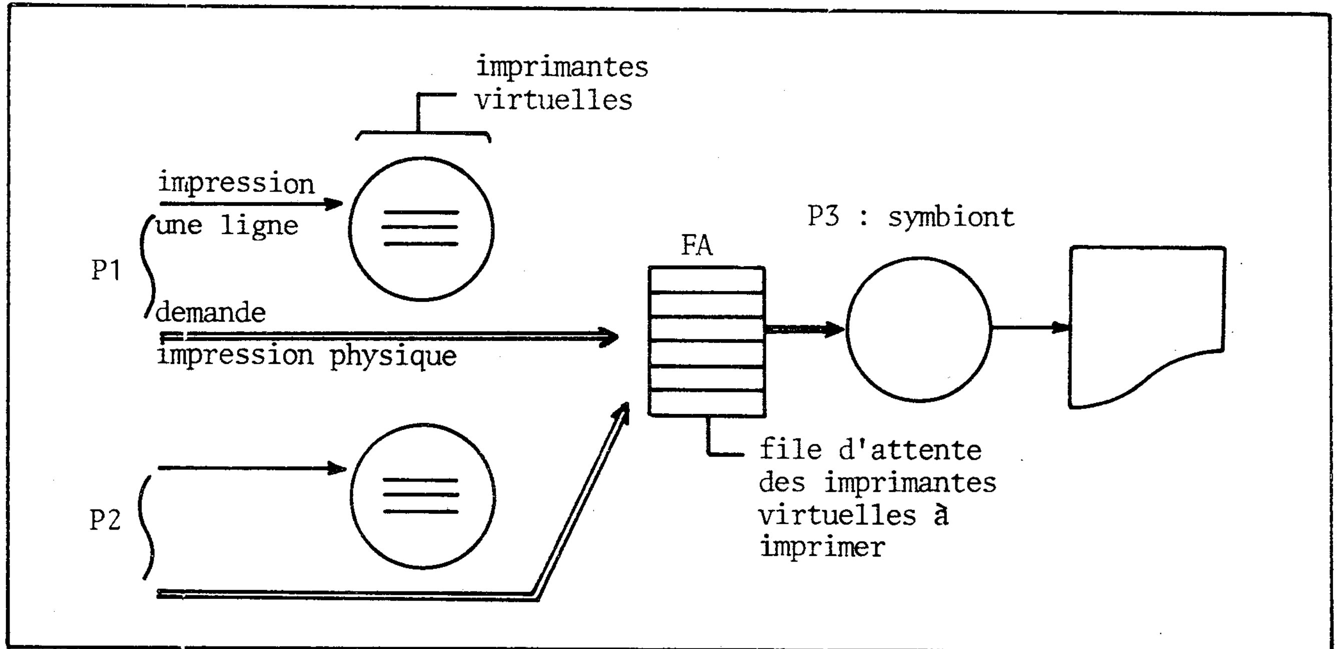


Figure 5 : Symbiont de sortie

Le problème du symbiont est semblable dans le cas des organes d'entrée, mais n'est pas complètement symétrique ainsi que nous le verrons au paragraphe § 5.1.

Ce mécanisme du producteur-consommateur peut être étendu à d'autres périphériques, comme par exemple la console de l'opérateur. Ce cas est matérialisé par la figure 6 où les processus utilisateurs P1 et P2 déposent des messages dans un segment FA qu'un troisième processus vide, à la vitesse d'impression réelle, sur la console réelle. Le segment FA sert de mémoire tampon et permet d'avoir des vitesses d'exécution (dépôt et retrait) différentes. Ce cas est traité au paragraphe § 5.2.

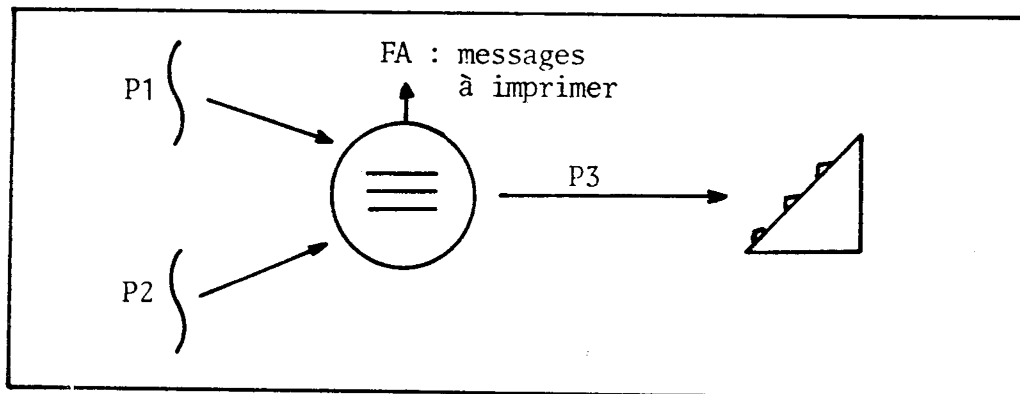


Figure 6 : Parallélisme entre impression réelle et virtuelle

4 - SYMBIONT DE SORTIE.

4.1. Problème.

Ce que veut l'utilisateur c'est un certain nombre d'imprimantes virtuelles et la possibilité de les fournir à un processus qui les imprimera successivement sur un périphérique de sortie. En particulier on veut étendre la notion d'impression en différé à tout un ensemble de segmentsde formats différents, par exemple on peut vouloir imprimer un segment-donnée de structure imprimante virtuelle (LPT), de structure éditeur de texte (EDIT), de structure 'bibliothèque de procédures de langage de commande' (LIB), etc... On trouvera dans la thèse de P. Laforgue [LA 75] une définition plus précise des différents types de fichiers réalisés sur des segments GEMAU.

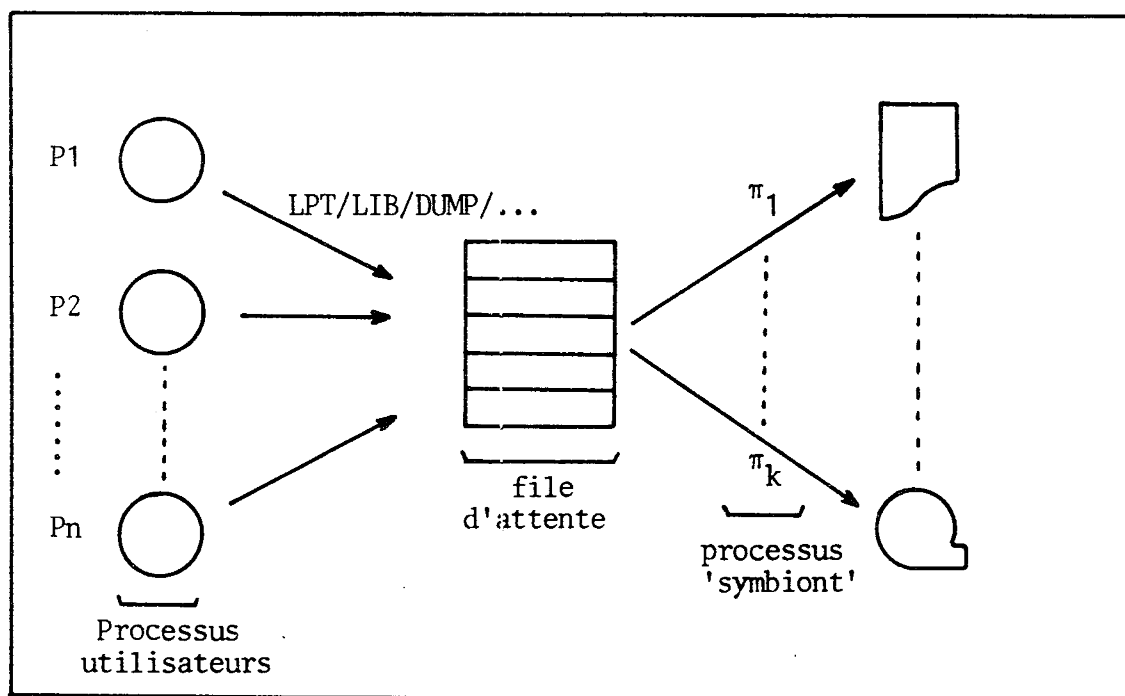


Figure 7 - Symbionts de sortie

La figure 7 symbolise les fonctions du symbiont de sortie. On remarque qu'il peut exister plusieurs symbionts de sortie.

4.2. - Différents environnements.

Nous allons maintenant étudier les relations entre les différents environnements de l'espace d'adressage correspondant respectivement à :

- Un des environnements utilisateur.
- Le sous-système : "Symbiont" de sortie.
- Le sous-système : "opérateur".
- La racine.

Ces relations sont schématisées par la figure 8.

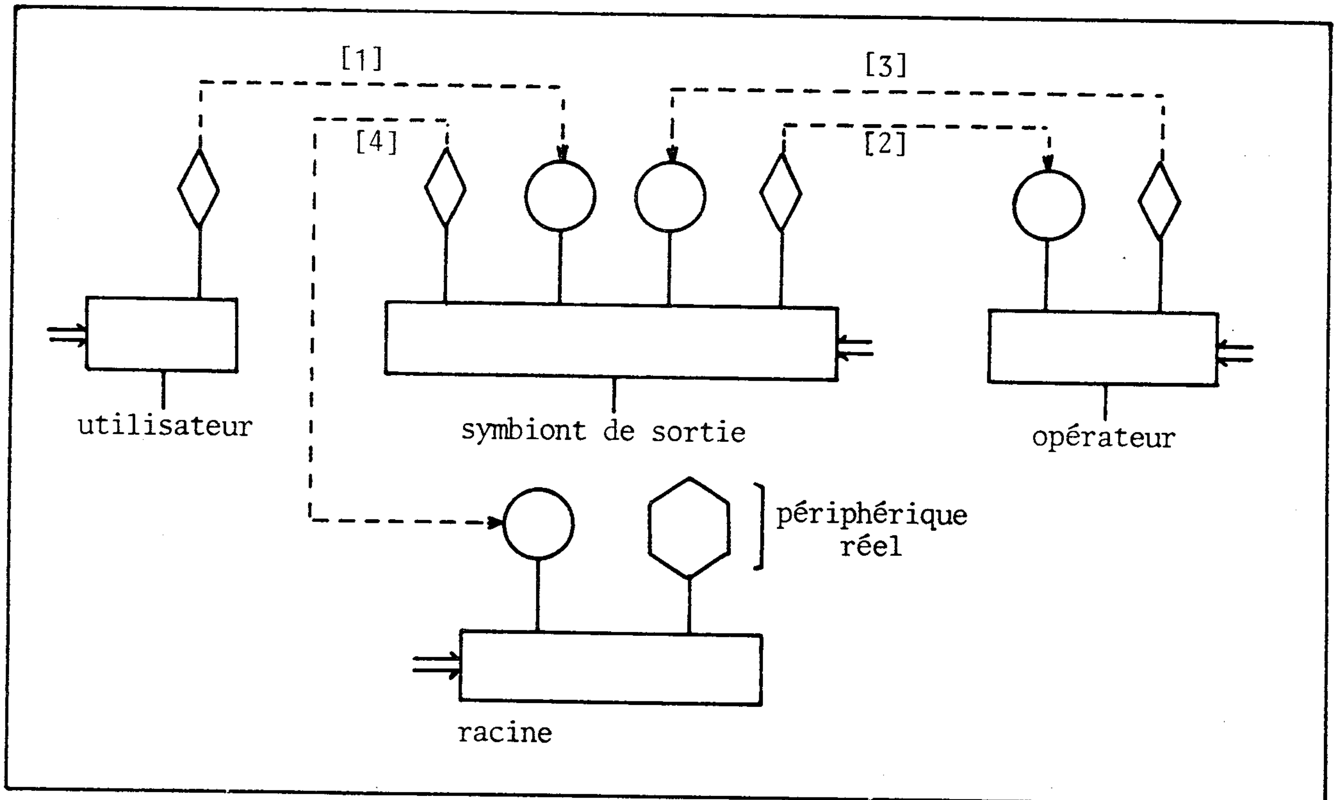


Figure 8 - relations entre les différents environnements

[1] appel par l'utilisateur d'une procédure (OFFLINE), située dans l'environnement symbiont de sortie, pour demander l'impression d'un segment. Il y aura transfert ou copie de ce segment depuis l'environnement utilisateur vers l'environnement symbiont. Le fait qu'il y ait transfert ou copie est dû à l'existence de besoins à la fois d'imprimante virtuelle (on n'a plus besoin de conserver la liste chez l'utilisateur quand l'impression est virtuelle) et aussi de service (on veut l'état actuelle d'une bibliothèque,

ce que l'on imprime c'est donc une copie de cette bibliothèque et non la bibliothèque elle-même). La copie est due au fait qu'il y a exécution parallèle du processus utilisateur et du processus symbiont, en particulier l'impression physique peut être très différée et pendant ce temps la valeur du segment (en cas de non copie) serait changée.

[2] Dépot par le symbiont de sortie de messages destinés à l'opérateur. Point d'entrée dans l'environnement opérateur : la procédure OPERATOR.

[3] Point d'entrée de l'opérateur dans l'environnement 'symbiont de sortie' à l'aide de la procédure OUTPUT (pour arrêt d'impression, ou destruction de fichiers en attente, etc...).

[4] lien vers des procédures systèmes, en particulier TIMEOUT.

4.3. Fonctions.

Les fonctions demandées à un symbiont de sortie sont les suivantes :

a) pour l'utilisateur : demande d'impression ([1])
CALL(OFFLINE, <désignation de segment>(<format>,<titre>,...)
où
<format> : type d'impression désirée, par exemple :
- 'DUMP' pour impression hexadécimale du contenu du segment
- 'LIB' pour impression d'une bibliothèque de procédure du langage de commande.
- 'LPT' pour impression d'une imprimante virtuelle.
- etc...

<titre> : chaîne de caractères quelconque qui servira de titre au listage.

etc. : toutes autres information de détail propres à une réalisation effective.

b) pour l'opérateur : demande d'arrêt d'impression d'un fichier ou destruction d'un segment en attente... ([3])
CALL (OUTPUT)(<fonction>[,<utilisateur>]...)

où

<fonction>::= 'STOP' pour arrêt d'une impression d'une imprimante.
'REMOVE' pour destruction d'un segment à imprimer.

L'environnement du symbiont de sortie comprend donc des procédures qui sont les points d'entrée (OFFLINE, OUTPUT), et des procédures propres. Parmi celles-ci, on considère celle qui réalise l'impression physique des segments (SPOOL), et celle qui sert à gérer les files d'attente des segments à imprimer (NAME). Cette dernière procédure qui peut être utilisée pour le compte de différents processus (utilisateur, symbiont, opérateur) manipule un segment de données qui contient la file d'attente des segments (FA). La procédure NAME doit posséder un degré de multiprogrammation égal à un,car elle manipule des données partagées et est donc en section critique.

4.4. Gestion de la file d'attente.

a) Fonctions.

C'est une procédure (NAME)en degré de multiprogrammation égal à un, qui gère la file d'attente des segments à imprimer. Les fonctions à réaliser par cette procédure sont donc l'introduction d'un nouveau segment (ENTER), l'obtention d'un segment (READY) ou le retrait d'un segment (REMOVE). En plus de la gestion de la file d'attente, cette procédure gère aussi un annuaire (FILE) dans lequel sont transférés les segments à imprimer. La figure 9 schématise les différentes fonctions de NAME et les processus qui les effectuent, les flèches indiquent les fonctions connues par chaque processus (similitude avec la notion de visibilité).

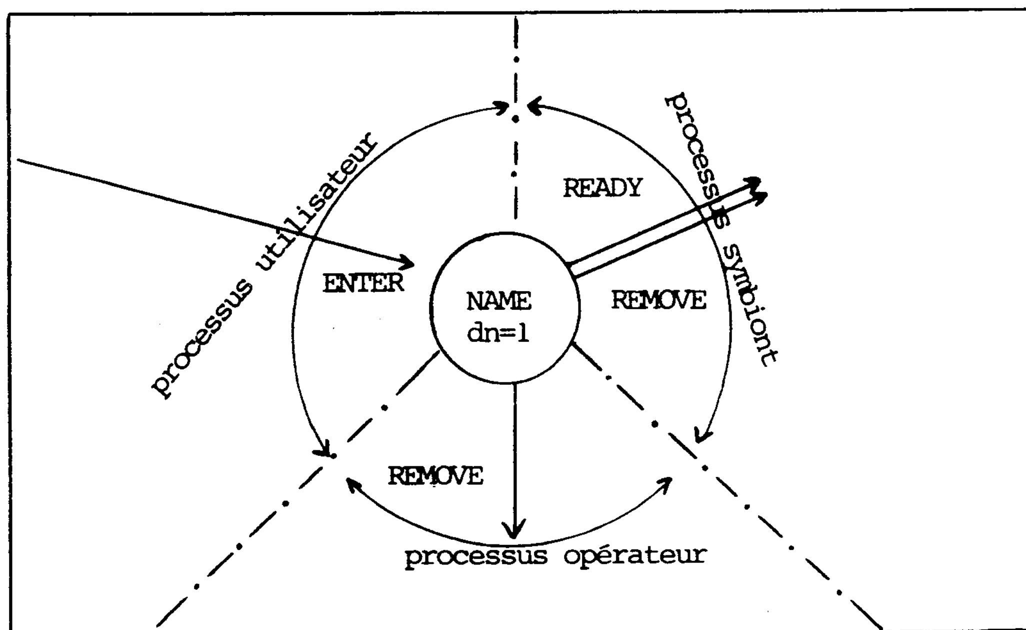


Figure 9 - Procédure de gestion de la file d'attente

L'appel de la procédure NAME est la suivante

[<fichier2>]←CALL(NAME[,<fichier1>])(<fonction>,<RG1>,...<RGn>)
avec
<fonction>::=ENTER/READY/REMOVE
<fichier1>::=segment à mettre en file d'attente (cas 'ENTER')
<fichier2>::=segment à imprimer (cas 'READY')

Cette procédure est locale à l'environnement symbiont et ne peut être appelée directement depuis un environnement disjoint.

La figure 10 donne les objets manipulés par cette procédure dans l'environnement 'symbiont'.

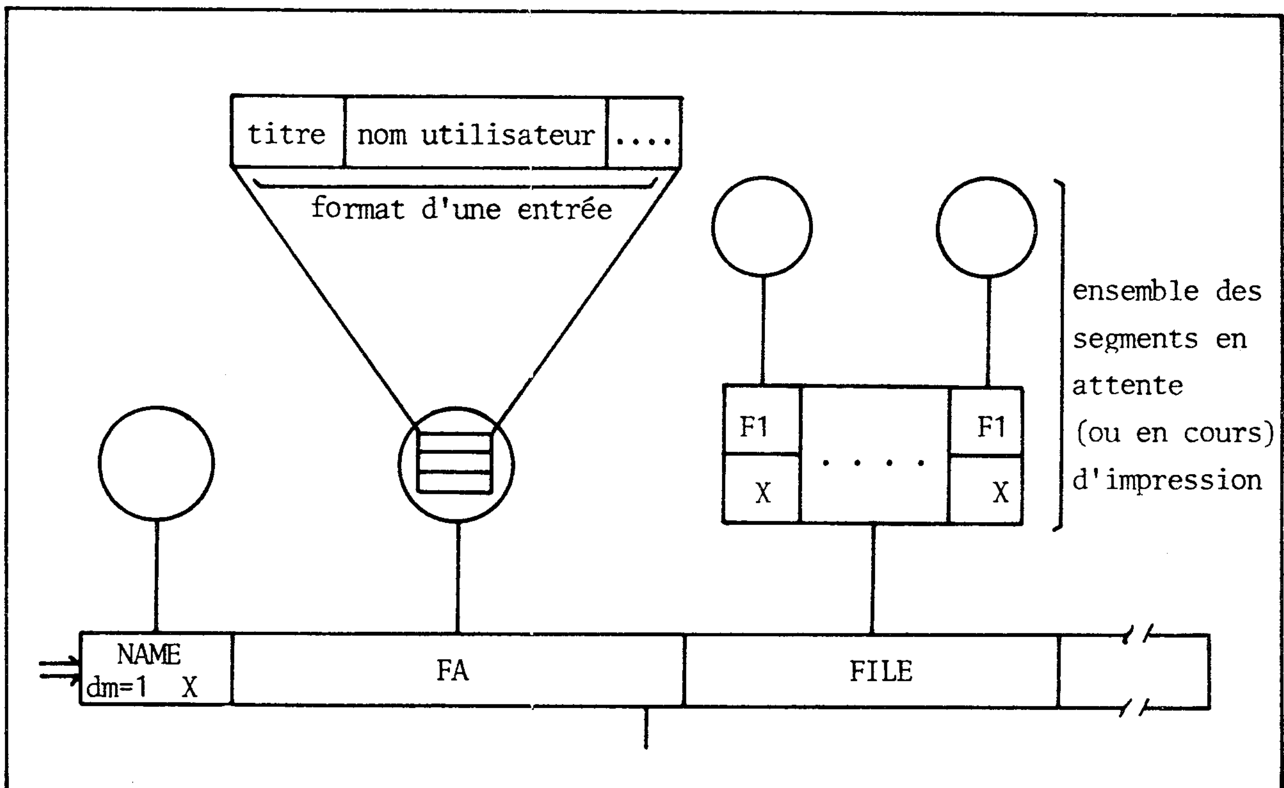


Figure 10 - Objets manipulés par NAME.

b) Algorithme

```
LN3 ← BIND(FA)
case  RGO of
ENTER : begin
            comment  F_i  est une variable dans FA qui indique le nom
            du segment courant dans FILE ;
            . ajouter indication nouveau fichier dans LN3(FA), son type
              (RG1)
            . incrémenter F_i
            . TRANSFER (FILE . F_i, LN2)

        end
or
READY : begin
            . trouver un fichier non encore utilisé dans LN3, soit nom_i
              son nom,le marquer utilisé
            . RETURN (FILE.nom_i)(nom_i)

        end
or
REMOVE : begin
            . trouver si le fichier du nom spécifié dans RG1 existe, si
              oui le retirer
            . DELETE (FILE,[RG1])
            . SAVE (FA)

        end ;

esac ;
```

c) Remarques

. NAME est la procédure d'accès à l'objet FA qui contient l'identification des segments à imprimer ainsi que leurs types.

. La fonction 'READY' comprend dans la pratique des paramètres supplémentaires pour permettre d'effectuer un choix sur le segment à prendre parmi tous ceux qui sont prêts et non encore utilisés. En particulier si on a plusieurs processus 'symbionts', l'un peut imprimer uniquement les fichiers inférieurs à une certaine taille (nombre de lignes) ou en

priorité les fichiers les plus petits, tandis qu'un autre symbiont voit les fichiers dans l'ordre d'arrivée (sauf pour ceux qui auraient été pris par le premier symbiont).

. Dans la pratique, de nombreuses informations supplémentaires sont fournies lors de la fonction ENTER, et en particulier, le nom de l'utilisateur, et les informations comptables.

. Des tests sur la présence et la validité des objets passés en paramètre sont aussi effectués sur le programme réel , nous les avons omis dans cet exemple par souci de simplicité et de clarté.

. La primitive SAVE est utilisée dans le cas ENTER pour sauvegarder l'état de la file d'attente FA ainsi si une panne se produit la file d'attente contient le nouveau segment et ce segment pourra être imprimé lors du redémarrage du système. De même chaque fois qu'un fichier est supprimé. Par contre un problème se pose dans le cas d'une panne survenant durant l'impression de certains segments, il ne faut pas perdre ces segments lors du redémarrage, en particulier l'indication dans FA,du fait qu'ils sont en cours d'utilisation,peut avoir été sauvegardée si un ENTER ou un REMOVE (d'un autre segment) a eu lieu pendant l'impression. Cet ENTER (ou REMOVE) peut être effectué par des processus utilisateurs (ou symbionts). Pour cela nous avons rajouté, dans NAME, une fonction supplémentaire (RESTART) dont le rôle est de balayer la file d'attente FA et de retirer l'indication "en cours d'utilisation" des segments qui la possède. Cette fonction est exécutée lors du lancement du premier symbiont de sortie. Cette fonction n'est pas schématisée dans l'algorithme décrit précédemment. Une autre façon de faire serait de donner dans l'environnement du symbiont l'accès à un objet qui indique le numéro de séance du système, et d'avoir un appel à cet objet dans NAME pour vérification avec une valeur conservée dans FA qui indiquerait le numéro de la séance où a eu lieu la dernière modification. Si les numéros sont différents alors la fonction RESTART est activée par NAME elle-même, puis le numéro dans FA est mis à jour.

L'algorithme décrit peut faire qu'il existe une entrée dans FA pour un segment alors que le segment a été physiquement détruit. Cela signifie qu'il y a eu une panne entre la destruction du segment et la sauvegarde de FA

(cas de la fonction REMOVE) ou entre la sauvegarde de FA et le transfert d'un segment (cas de la fonction ENTER). Il suffit de rajouter des tests dans les fonctions ENTER et REMOVE pour ne pas tenir compte de l'entrée si le fichier est détruit.

4.5. Procédure principale du symbiont de sortie.

a) définition.

C'est la procédure qui demande l'obtention d'un segment à imprimer (appel à NAME), et qui procède à l'impression sur périphérique réel. Un processus symbiont est crée avec la procédure SPOOL comme procédure initiale(LN1) et avec comme paramètre (nom local LN2) le périphérique réel considéré. Cette procédure effectue l'impression, et quand il n'y a plus rien à imprimer (le segment résultant de l'appel à NAME pour la fonction READY n'existe pas) le processus se met en attente pour un certain délai, puis appelle à nouveau NAME pour voir s'il y a quelque segment à imprimer, et ainsi de suite. L'appel pour une mise en attente s'effectue à l'aide d'un objet système, appelé TIMEOUT auquel on fournit comme paramètre la durée de suspension du processus.

b) algorithme.

```
SPOOL  :      . LN3 ← BIND (type = 'segment') ;
              . comment  liaison d'un segment temporaire au processus pour
                contenir les variables de travail de la procédure SPOOL ;

[1] SPOOL1 :  . LN4 ← CALL(NAME)('READY') ;
              . if LN4 non affecté then
              . begin
                        . CALL (TIMEOUT)(15) ;
                        . goto SPOOL1 ;
              . end ;
              . impression sur LN2 du segment contenu dans LN4 en fonction
                de son type (LPT/DUMP/LIB/...)
              . CALL (NAME,LN4)('REMOVE');
[2]           . CALL (OPERATOR)(message);
              . FREE (LN4);
              . goto SPOOL1 ;
```

c) Remarques.

. la procédure SPOOL est entièrement réentrante, et est la procédure initiale de tous les processus 'symbiont de sortie'.

. l'appel à NAME ([1]) pour obtenir un segment peut comporter, dans la réalité, des paramètres supplémentaires qui permettent d'effectuer un choix sur le segment à imprimer et non pas de prendre le premier dans la file d'attente.

. l'appel à l'opérateur ([2]) sera explicité plus en détail au paragraphe § 5.2. c'est tout simplement l'équivalent d'une demande d'entrées-sorties.

. Dans la pratique, il existe de nombreux autres appels effectués par SPOOL pour :
- définir les informations comptables (lien vers un environnement spécialisé).
- obtenir la date et l'heure afin de les indiquer sur la liste d'impression.
- traiter des erreurs qui peuvent se produire pendant l'impression réelle (arrêt du périphérique LN2, absence de papier sur l'imprimante, etc...) et communiquer alors avec la console opérateur pour obtenir une intervention manuelle.

d) Protection.

La procédure est en exécution seule (X)

4.6. Point d'entrée depuis un environnement utilisateur.

Il est effectué par la procédure OFFLINE dont l'algorithme est le suivant :

. vérification des paramètres.
. CALL(NAME,LN2)('ENTER', RGO, RG1...)

Son rôle est en fait de contrôler l'accès à la procédure NAME, de façon à ce qu'un utilisateur ne puisse modifier la file d'attente FA autrement que par la fonction ENTER. Ce qui n'aurait pas été le cas si le lien chez l'utilisateur avait été directement sur l'objet NAME. Ceci est une illustration du problème de la méfiance mutuelle. Si l'utilisateur avait directement accès à NAME il pourrait modifier la file d'attente sans aucun controle en utilisant les fonctions READY et REMOVE. De plus, cette solution évite qu'un utilisateur puisse lier, pour un temps indéterminé, la procédure NAME dans son espace d'exécution et ainsi bloquer tous les autres processus (utilisateurs, symbionts et opérateurs) qui auraient besoin d'accéder FA par l'intermédiaire de NAME. Ceci, parce que la procédure NAME a un degré de multiprogrammation égal à un.

4.7. Point d'entrée depuis l'environnement opérateur.

Il est réalisé par la procédure OUTPUT. L'algorithme de cette procédure répond aux mêmes impératifs que ceux exprimés dans OFFLINE (limitation de l'accès).

4.8. Exemple.

Considérons la figure 11, qui représente les espaces d'adressage d'un utilisateur, des symbionts de sortie et de la racine.

Les processus existants sont PU_i (utilisateur) et les deux symbionts PS1 et PS2, chacun d'eux ayant la même procédure mais des périphériques différents (1 et 2) ainsi que des objets temporaires (LN3 = 100 et 101) pour contenir les variables de travail propres à chacune des activations de la procédure 15.

L'environnement concernant le sous-système opérateur pour l'envoi des messages à l'opérateur et le sous-système de comptabilité n'ont pas été représentés sur cette figure pour ne pas en compliquer la lisibilité.

Supposons que l'utilisateur demande l'impression du segment FILE(32), par l'intermédiaire de l'appel :

CALL(OFFLINE, FILE)('DUMP','erreur')

format titre

l'état de PU_i sera alors donné par la figure 12. Durant l'exécution de OFFLINE (11) on effectue (cf. § 4.7)

CALL(NAME,LN2)('ENTER', RG0, RG1)

qui va transférer l'objet 32 depuis l'annuaire 30 vers l'annuaire 14 sous

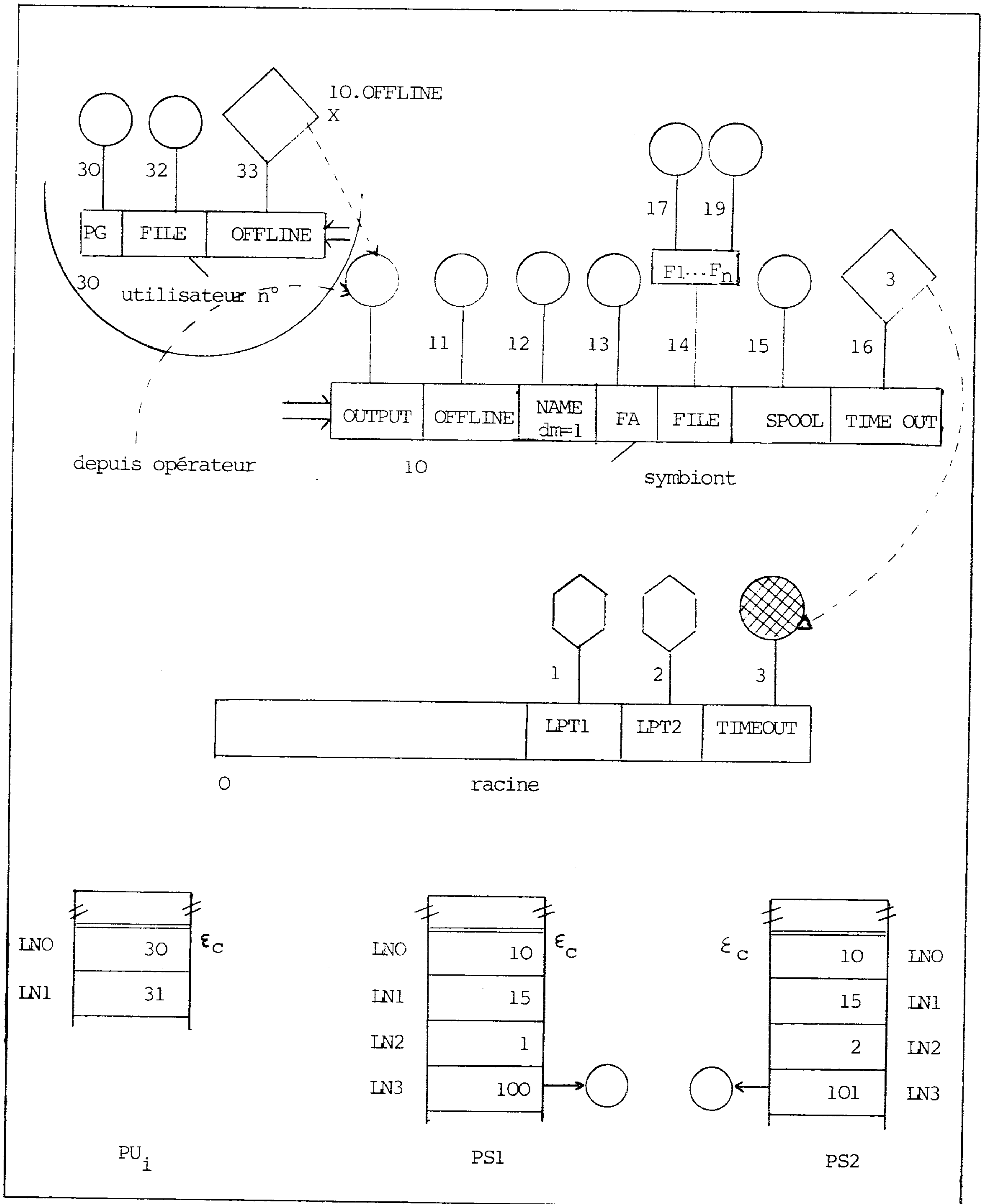
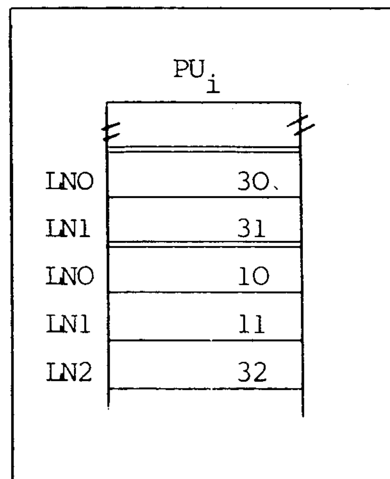


Figure 11 - Environnement 'symbiont de sortie'



le nom F_i, les valeurs de RG0 et RG1 sont ici respectivement 'DUMP' et 'erreur'.

Figure 12 : processus utilisateur exécutant OFFLINE.

Remarque :

Si l'utilisateur veut seulement l'impression de son segment sans destruction, il doit faire l'appel par

CALL (OFFLINE, value FILE) ('DUMP', 'erreur') afin de transférer une copie au symbiont de sortie.

Qui fait quoi ?

Les segments dans l'annuaire 14 (FILE) sont crées par le processus utilisateur et détruit par un processus symbiont (ou éventuellement par le processus opérateur).

4.9. Remarque générale

Pour les processus symbiont LN2 représente le périphérique sur lequel effectuer l'impression ; tout cela marche de la même façon, que ces périphériques soient réels (imprimante, bande magnétique, perforateur de cartes, même terminal lourd à distance, ..) ou virtuels (procédure). En particulier on peut essayer facilement une nouvelle version du symbiont en créant un environnement symbiont chez un utilisateur puis en le testant à l'aide de l'ancien symbiont, c'est-à-dire en fournissant comme paramètre à SPOOL une imprimante virtuelle. C'est ainsi qu'ont été mises au point les différentes versions des symbionts sur le prototype. Ainsi, dans le premier essai, un utilisateur pouvait posséder en propre l'imprimante réelle (par création d'un

lien, ceci est schématisé par la figure 13).Ensuite, une imprimante virtuelle a été introduite par utilisateur, et un système de symbiont rudimentaire a été développé, et des liens chez les utilisateurs crées vers la procédure OFFLINE qui effectuait elle-même dans un premier temps l'impression des imprimantes virtuelles c'est-à-dire, qu'il n'y avait pas de processus symbiont ni de procédure NAME, ce cas est schématisé par la figure 14. Dans ce cas, la procédure OFFLINE possédait un degré de multiprogrammation égal à un pour permettre l'impression en séquence des différentes imprimantes virtuelles. Ensuite, la version décrite dans ce paragraphe a été développée, il a alors suffit de substituer dynamiquement la procédure OFFLINE précédente par la nouvelle procédure après insertion dans l'environnement 'symbiont' des objets NAME, SPOOL, FA, FILE, etc... Cette substitution dynamique n'a demandé aucun changement dans les liens possédés par les utilisateurs. Puis toute nouvelle modification s'est ensuite effectuée par substitution dans l'environnement 'symbiont de sortie'. Ce qui est intéressant c'est qu'on peut développer dans un environnement n'ayant pas plus de prérogatives qu'un utilisateur, un sous-système de type symbiont, le tester et le mettre au point à l'aide de l'ancien sans perturber le fonctionnement des autres utilisateurs.

Il est important de remarquer qu'entre la figure 13 et la figure 14, l'impression d'une ligne par un programme n'a pas changé pour l'utilisateur : c'est toujours un appel à LPT avec les paramètres tels que nous les avons vus au paragraphe 2.1. Le seul changement logique est l'introduction d'une demande explicite d'impression physique. Cependant, dans le cas de la figure 13, il fallait demander un lien vers une imprimante alors que cette contrainte n'existe plus dans la figure 14, mais il faut en contre partie demander l'impression physique. Au point de vue des temps d'exécution la solution de la figure 14 est un progrès appréciable mais avec le problème d'une attente sur l'impression par OFFLINE. Ce dernier inconvénient est levé par la solution finale.

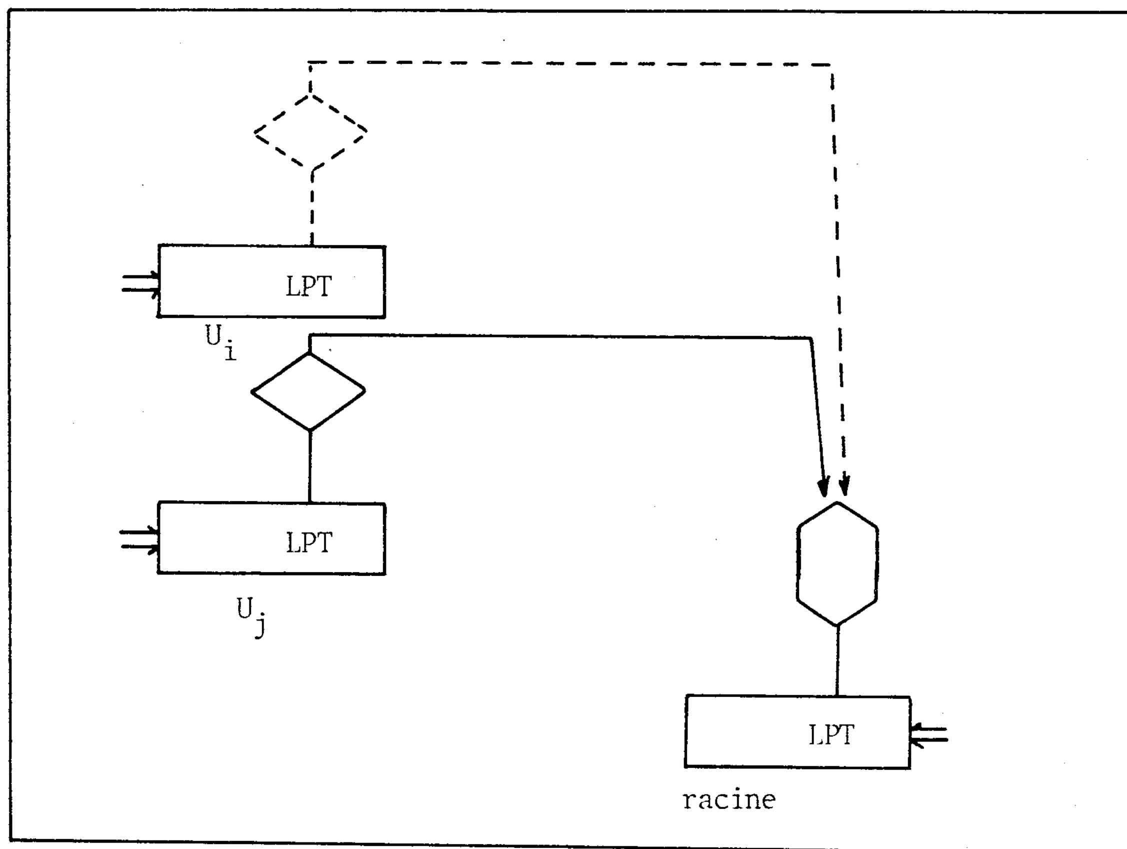


Figure 13 : Symbiont originel :
l'imprimante réelle affectée successivement à chaque utilisateur

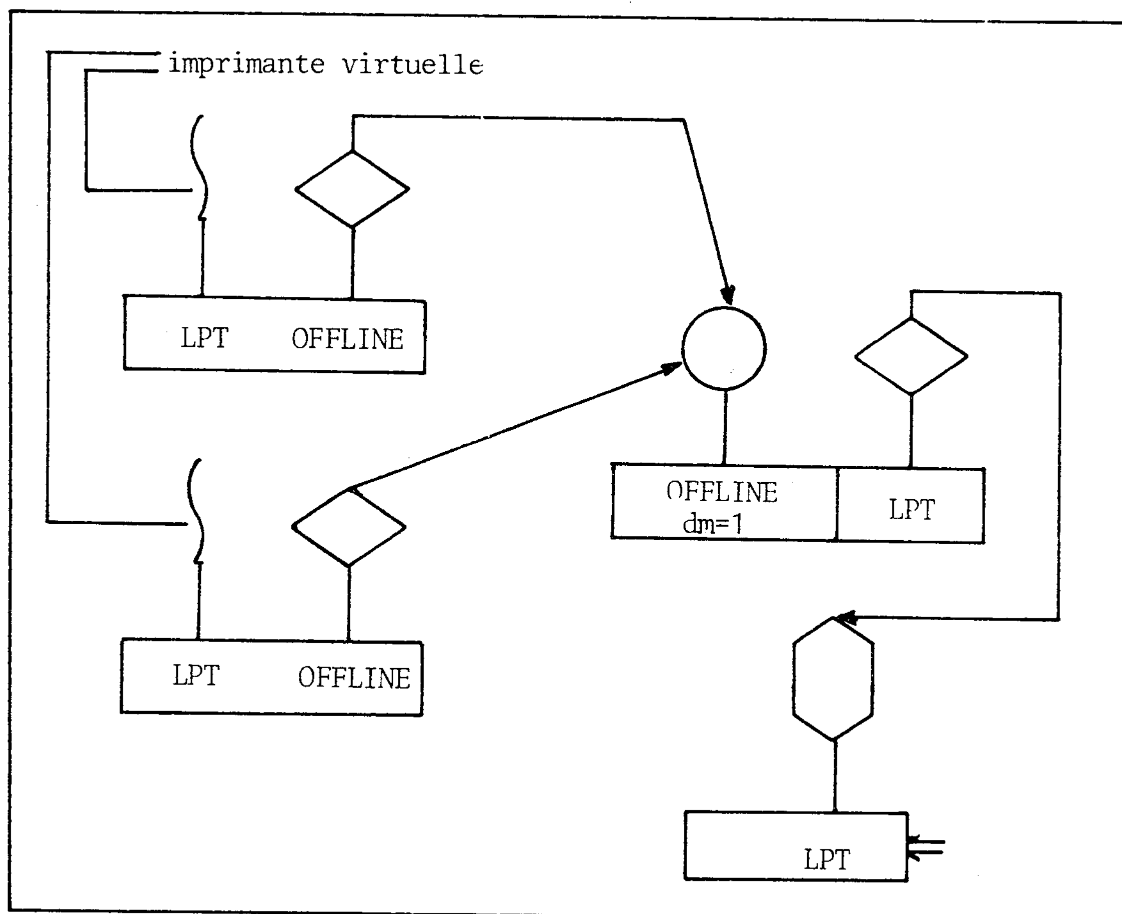


Figure 14 : Symbiont où l'impression est effectuée
successivement pour chaque imprimante réelle

5. AUTRES EXEMPLES DE SYMBIONTS.

5.1. Symbiont d'entrée.

Ce que l'on veut, c'est créer une fonction inverse de celle des symbionts de sortie. On veut, à partir d'un périphérique réel, créer des segments de données et les affecter à des environnements utilisateurs.

Pour cela on a donc un processus qui lit des enregistrements, les place dans un segment temporaire puis transfert ensuite ce segment dans l'environnement utilisateur. L'utilisateur peut ensuite utiliser un lecteur virtuel pour lire les cartes rangées dans ce segment. La figure 15 donne schématiquement les relations entre les différents environnements.

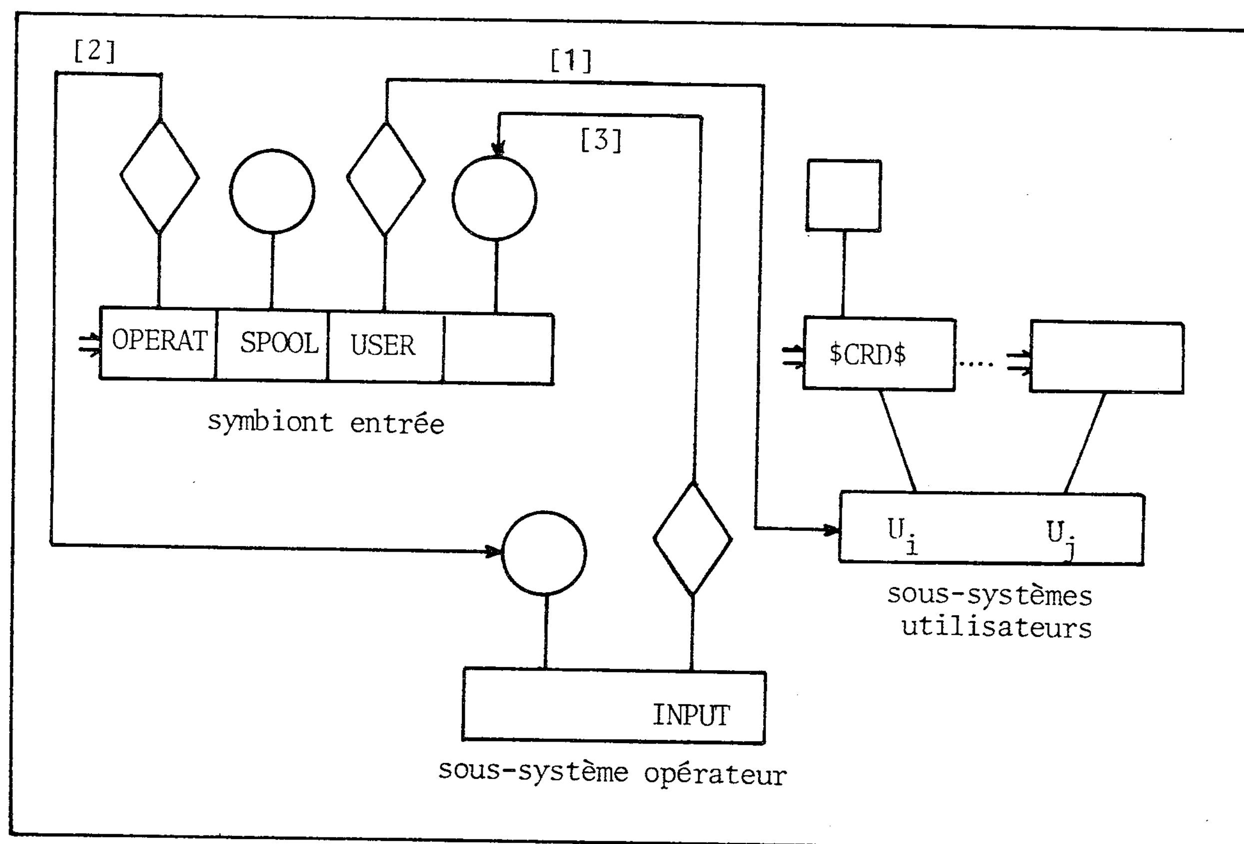


Figure 15 : Relations entre environnements dans le cas du symbiont d'entrée

En particulier on a :

. le lien [1] vers l'annuaire des utilisateurs
. le lien [2] vers l'opérateur pour l'envoi de messages sur la console opérateur.
. le lien [3] qui est une entrée dans le symbiont d'entrée pour permettre à l'opérateur d'avoir une action sur le déroulement de ce symbiont.

Lors de la lecture d'un paquet de cartes, le symbiont détermine, à l'aide d'une carte contrôle, le nom de l'utilisateur auquel ce paquet appartient, le nom du segment à créer chez l'utilisateur et le format de stockage de l'information.

Pour éviter d'introduire des segments n'importe où dans l'environnement de l'utilisateur on peut prendre la convention que ces segments sont à mettre dans un annuaire de nom prédéfini (par exemple CRD) dans l'environnement de l'utilisateur. En créant (ou en ne créant pas) un tel annuaire, l'utilisateur permet (ou ne permet pas) la lecture de cartes à transférer dans son environnement.

5.2. Processus console.

Nous avons parlé à plusieurs reprises d'envoyer des messages à l'opérateur, ceci s'effectue de façon identique à l'envoi de segments aux symbionts de sortie, sauf qu'ici, les messages seront de taille inférieur à un segment. On utilise une technique analogue à celle de la procédure NAME, et les informations dans la file d'attente (FA) sont directement les messages.

Un processus console effectue le vidage de cette file. La procédure équivalente à NAME est donc avec un degré de multiprogrammation égal à un.

On peut remarquer qu'il est très facile au processus de vidage de trier les messages et de les envoyer, selon les cas, vers des destinations différentes. En particulier on peut envoyer sur une console graphique uniquement les messages d'erreurs, cependant que l'on envoietous les messages sur une imprimante ou une bande magnétique.

L'espace d'adressage et d'exécution permet de réaliser très facilement la duplication d'une même sortie, ou une expédition sélective : il suffit pour cela de créer un périphérique virtuel

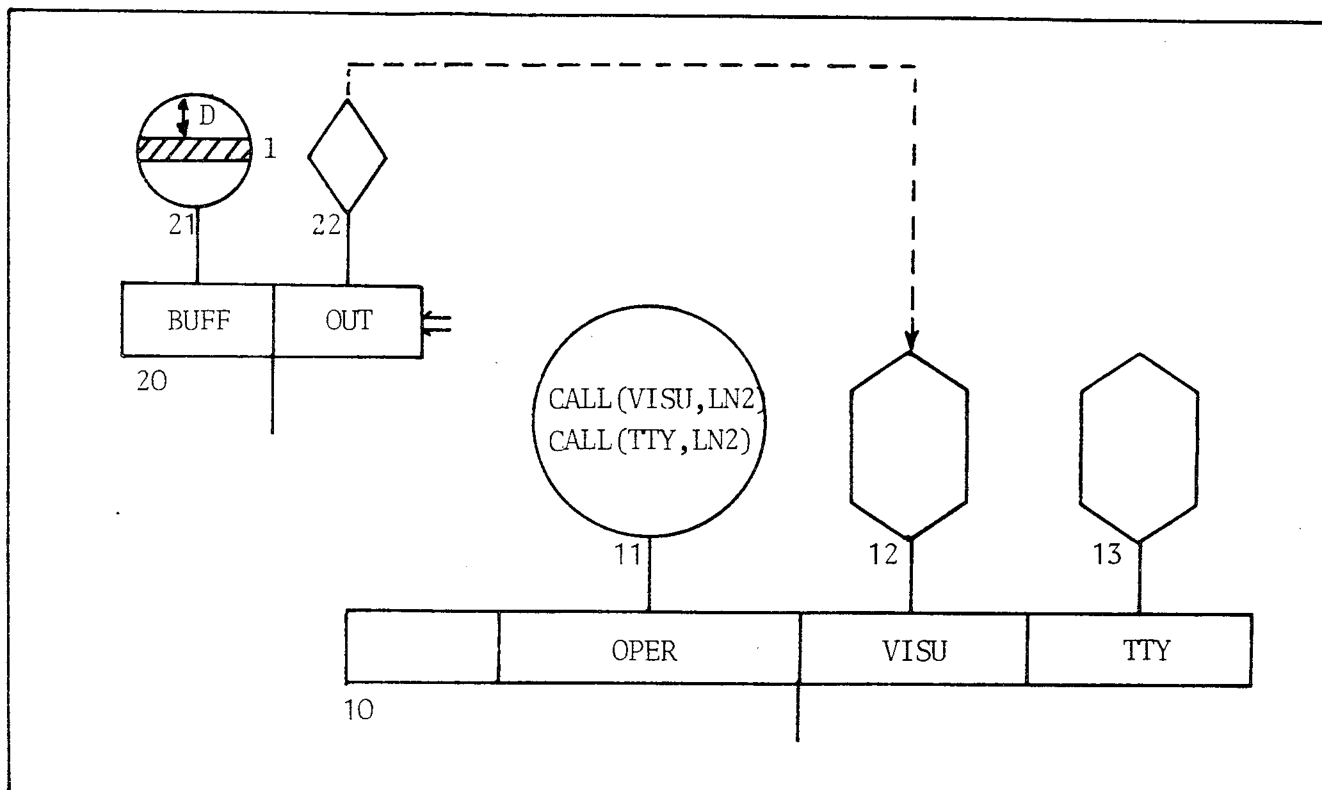


Figure 16 :

La figure 16 donne un tel exemple. Dans l'environnement 20 on a un lien vers l'objet 12.

Toute demande d'impression s'effectue par

CALL(OUT,BUFF)('OUTPUT', D,1)

dans ce cas le contenu de la zone (D,1) de BUFF sera envoyé sur le périphérique VISU. Supposons que l'on remplace le lien 22 par un lien de même nom d'entrée mais vers 11 et que la procédure 11 comporte

CALL (VISU,LN2)(RG0, RG1, RG2)
CALL (TTY,LN2)(RG0,RG1,RG2)

alors le même message sera envoyé à VISU et TTY . L'utilisateur (processus dans l'environnement 20)est complètement ignorant de ce fait.

6. CONCLUSION

Les problèmes abordés dans ce chapitre ont été aisément résolus en utilisant l'espace des objets de GEMAU. En particulier la protection, la fiabilité et la modularité ont été pleinement utilisées. Cependant, on s'aperçoit d'une gêne au niveau de la synchronisation entre processus qui est effectuée par attente "quasi-active", i.e. en utilisant une horloge de réveil. Une première solution extrêmement simple à réaliser et très suffisante dans les problèmes que nous venons de traiter, est l'introduction d'un nouveau type d'objet : le sémaphore. Cet objet serait uniquement exécutable et possèderait les opérations désormais classiques : P, V et I [DI 68]. Une seconde solution plus générale s'inspirerait des travaux de Hoare [HO 74 a], en introduisant la notion de moniteur.

De plus, la réalisation des symbionts s'est avérée très simple et très rapide ; ceci est surtout dû aux primitives de l'espace des objets. La propriété de substitution dynamique est aussi utile dans le développement continu et en ligne de ces symbionts. Dans ce dernier cas, il faut aussi faire intervenir comme élément important la banalisation qui permet de définir et d'expérimenter des sous-systèmes en tant qu'utilisateurs sans avoir besoin de prendre la machine pour faire ces tests et sans perturber l'exécution des autres utilisateurs.

La banalisation des objets est aussi un atout considérable avec l'introduction des réseaux d'ordinateurs, car on peut définir tous les périphériques que l'on veut et l'utilisateur peut effectuer ses entrées (ou sorties) depuis n'importe quel objet en ignorant complètement la localisation réelle de l'objet.

La réalisation effective, sur le prototype 10070, suit les grandes lignes générales de ce que nous venons de voir, les différences viennent de ce que ce prototype procure des contraintes importantes : taille limitée des segments (64 K caractères) et temps d'interprétation des CALL.

TROISIEME PARTIE

COMPARAISONS, ETUDES PARTICULIERES

Chapitre 4

DESIGNATION, ADRESSAGE, LIAISON, SIGNIFICATION DES NOMS

Nous posons le problème général de la signification d'un nom et de ses transformations éventuelles. La désignation, à l'exécution, est étudiée pour des espaces statiques, évolutifs et récursifs. Dans chacun de ces cas, les avantages et inconvénients sont énumérés et un certain nombre d'exemples donnés.

Nous étudions ensuite plus précisément le problème de la liaison entre une exécution et les objets nécessaires à cette exécution, avec la définition du moment de liaison : implicite, explicite, avant ou pendant l'exécution.

Nous montrons comment les choix faits dans GEMAU se placent par rapport aux diverses solutions.

1. DEFINITION DU PROBLEME

1.1. Nature évolutive des noms

Considérons un utilisateur qui désire résoudre un problème sur une machine donnée. Durant tout le processus qui va de la conception à l'exécution du programme résultant, il y a un certain nombre d'étapes dont la figure 1, volontairement simplifiée, donne un exemple :

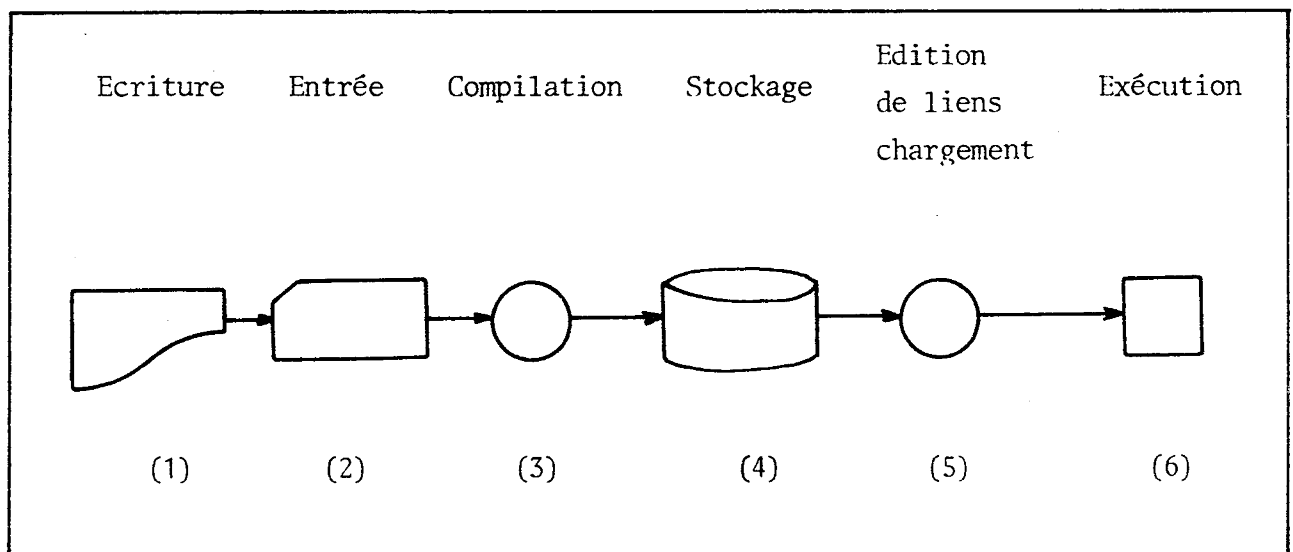


Figure 1 : Evolution d'un programme

Ce sont :

(1) écriture du programme, choix des algorithmes, des variables, du langage de programmation,

(2) entrée du programme dans la machine : perforation de cartes, entrée depuis un terminal, ..,

(3) compilation du programme

(4) sauvegarde du programme qui vient d'être compilé dans un fichier

(5) chargement du programme en mémoire centrale avec, possiblement, une translation des adresses d'exécution (ceci est fonction de l'adressage de la machine),

(6) exécution de ce programme par le matériel.

A chacune de ces étapes il y a disparition d'un certain degré de liberté, ce phénomène étant très souvent irréversible. Par exemple, sur les machines de type IBM 360 ou CII 10070, avec les systèmes OS 360 [PR 72] et SIRIS 8. Ces liaisons sont :

Etape 1 : aux variables du problème on donne des noms symboliques (identificateurs) en fonction du langage choisi ;

Etape 3 : ces identificateurs sont transformés en déplacements (appelés très souvent : adresses translatables) par rapport au début du programme ;

Etape 5 : les déplacements sont transformés en adresses (dans un espace d'exécution linéaire.

La figure 2 donne un exemple de telles transformations :

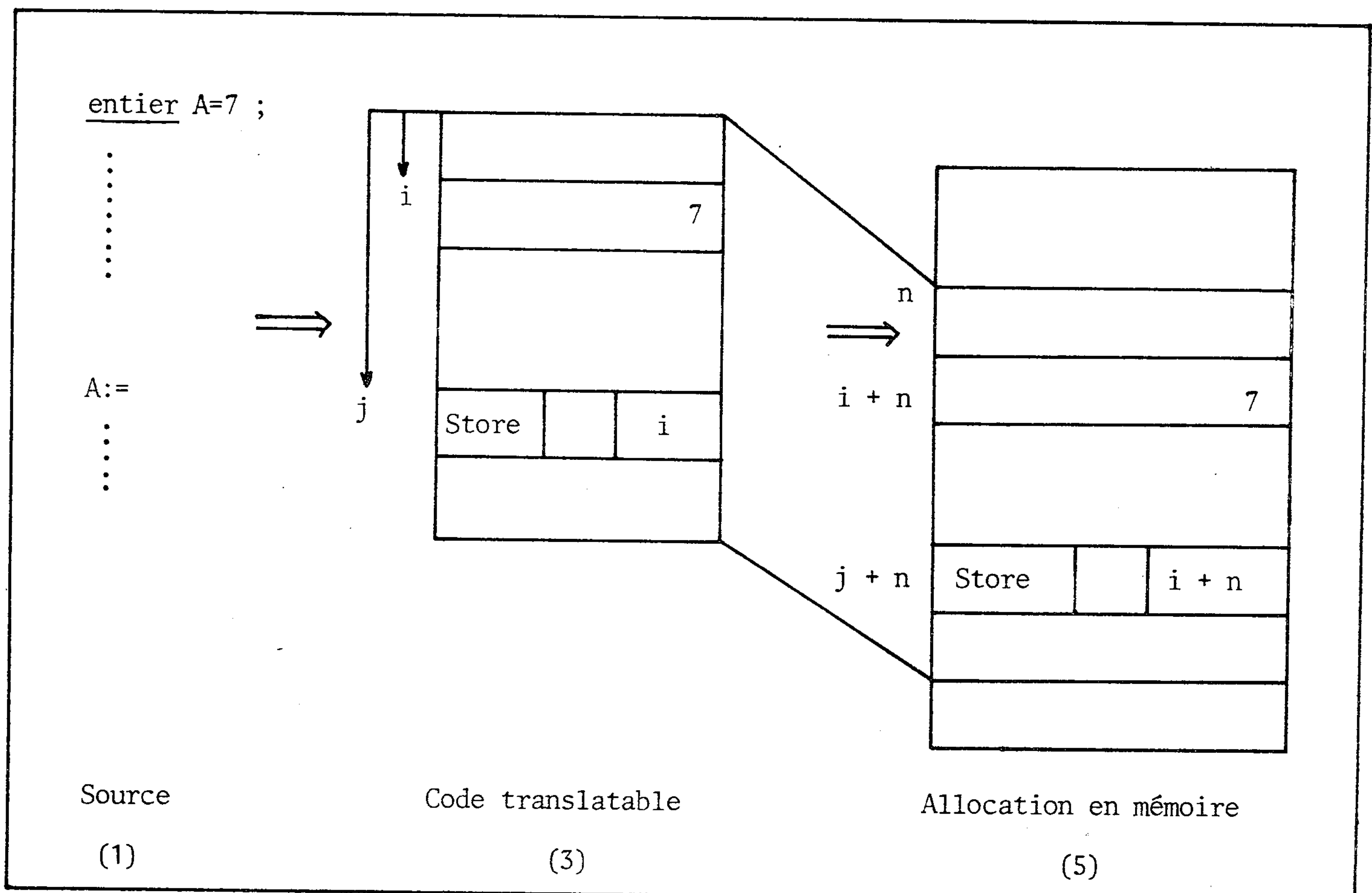


Figure 2 : Transformations successives des noms

L'irréversibilité se traduit par le fait qu'il est impossible de savoir que l'adresse i + n réfère l'entier A du programme source initial.

1.2. Substitution

Considérons maintenant le même programme, mais supposons que ce programme fasse appel à une procédure "multiplication de matrices" ; l'utilisateur peut incorporer cette procédure à différents instants pendant les étapes précédentes. Par exemple :

Etape 1 : en incorporant dans son programme source le code (source) de la procédure ;

Etape 2 : en incorporant dans son paquet de cartes un autre paquet de cartes appartenant à une bibliothèque de procédures pré-perforées ;

Etape 3 : à la compilation, si le compilateur offre des possibilités de copier des programmes (en source) depuis des fichiers ;

Etape 5 : après la compilation, mais avant le chargement en mémoire centrale, en incorporant une procédure de multiplication de matrice déjà compilée (cette étape est généralement connue sous le nom d'édition de liens : OS 360 [PR 72], SIRIS 7) ;

Etape 6 : à l'exécution où a lieu la recherche des objets appartenant à des références non résolues (cas de MULTICS [CO 72], GEMAU).

Ces différentes étapes sont un sous-ensemble de celles schématisées par la figure 1.

Dans la réalité il peut exister d'autres mécanismes, par exemple l'absence des phases 4 et 5, dans le cas où un compilateur (ou un assembleur) génère directement des instructions exécutables.

La résolution des références par l'incorporation d'un sous-programme dans un programme s'appelle *la liaison* du sous-programme avec le programme, cette liaison pouvant être effectuée en des instants variables. La liaison est la transformation d'un type de nom en un autre type de nom.

Considérons maintenant le problème de la substitution : par exemple on veut remplacer la procédure de "multiplication de matrices" par une autre procédure acceptant les mêmes paramètres, mais réalisée selon un algorithme (interne) différent ; il est facile de voir que la substitution doit s'opérer à l'étape où a été faite la liaison et qu'il faut ensuite refaire toutes les étapes ultérieures. Plus la liaison est tardive, plus la substitution est aisée. Bien que le retard de l'instant de liaison augmente les coûts de réalisation, il augmente la liberté et la flexibilité de l'allocation des ressources (cf. Denning [DE 71]).

La transformation de nom est faite dans un contexte particulier et il n'y a aucune ambiguité sur le nouveau nom (le nom transformé). En ALGOL 60, dans un programme, chaque bloc forme un contexte séparé et le contexte d'un bloc est celui du bloc immédiatement englobant plus les déclarations apparaissant dans le bloc interne. Dans un tel langage, le compilateur peut effectuer directement une transformation de nom. Cependant, ce n'est plus le cas si on permet dans le langage des créations dynamiques de variable, par exemple :

si < condition > alors entier x finsi ;

Ce type de déclaration dynamique existe pour les systèmes classiques de fichiers (opération OPEN). Dans ce cas, il est impossible de résoudre la correspondance entre un nom et un objet (nom universel) avant l'exécution.

L'instant de liaison reporté à l'étape 6 est généralement utilisé pour les objets de type fichier dans tous les systèmes classiques où ceci est effectué lors de l'ouverture du fichier. De plus, certains langages ont des problèmes inhérents à leur structure : par exemple, en ALGOL, la taille d'un tableau n'est connue qu'à l'exécution, dans ce cas la liaison ne pourra donc être réalisée que le plus tard possible. Nous nous proposons de généraliser ce mécanisme à tous les objets.

Nous pouvons définir maintenant de façon plus précise ce qu'est un nom et une liaison. Un nom est tout simplement une chaîne de chiffres binaires (c'est un *contenant*) et l'opération de liaison consiste à affecter un *contenu* (objet) à ce nom. Ce qu'il est important de voir, c'est la relativité de la notion de liaison, car le contenu peut être lui-même un nom à interpréter (donc à lier) par rapport à un autre contexte de noms.

1.3. Propriétés des espaces d'exécution et d'objets

Le fait que les adresses (ou noms) ne soient pas indépendantes de contextes d'exécution amène un certain nombre de problèmes qui sont agravés par le fait que des objets différents puissent être partagés par des utilisateurs (processus) différents. Ces problèmes sont essentiellement ceux de la désignation des contenus d'objets (à l'exécution), des références entre objets et du moment de liaison des objets. Ces problèmes ont été traités dans de nombreux articles, parmi lesquels on peut citer Dennis [DE 65], Fabry [FA 74].

Tous ces problèmes ont trait à la gestion de mémoire, pour laquelle on peut noter ceux liés à l'hétérogénéité des supports de mémoire auxquels nous ne nous intéressons que peu ou pas du tout dans le présent ouvrage, et ceux qui sont liés aux mécanismes permettant une programmation modulaire. Ce sont ces derniers que nous avons évoqués et que nous traitons ici.

Si nous considérons un objet, celui-ci doit avoir des relations avec le monde extérieur (relation de désignation et de références). Les mécanismes permettant de mettre en oeuvre ces relations doivent posséder les propriétés suivantes :

a. une procédure doit pouvoir accéder à ses paramètres à l'aide de la même instruction, quel que soit le processus pour lequel elle s'exécute ;

b. une procédure doit pouvoir accéder à des données rémanentes dans les mêmes conditions ;

c. un objet doit pouvoir référencer tout ou partie d'un autre objet.

Toute solution possédant les propriétés a. et b. prohibent l'utilisation classique de l'édition de lien (étape 5) où l'on fige par recopie les différents objets nécessaires à une exécution, car dans ce cas on ne partage plus les objets eux-mêmes, mais plutôt des copies.

Beaucoup d'études traitent uniquement le problème de la désignation dans l'espace d'exécution et des références inter-objets sans liaison avec l'espace des objets (système de gestion de fichiers, nomenclature). Dans toutes ces études, il faut construire en plus de l'espace d'exécution un noyau de système avec des règles de désignation pour un espace d'objets. Parmi les études qui ont attaqué les deux aspects simultanément (intégration des espaces d'exécution et d'objets), nous ne connaissons que celles de Vanderbilt [VA 69] et de SAR [VE 73]. Toutes les autres études, telles que celles de Lauer [LA 74], Fabry [FA 74], PP250 [WI 72], MULTICS* [CO 72], de Spier [SP 74], ne traitent que de l'aspect "espace d'exécution".

Nous allons dans ce chapitre étudier successivement le problème de la désignation et des références à l'exécution (§ 2), puis le problème de la liaison entre les objets et l'espace d'exécution (§ 3).

---

* Dans ce cas les espaces d'objets et d'exécution sont juxtaposés l'un à l'autre, mais aucun n'utilise les propriétés de l'autre.

2. DESIGNATION DANS LES ESPACES D'EXECUTION

2.1. Problèmes de désignation et de référence

Avant l'apparition de la translation dynamique d'adresse (pagination, adressage par base et déplacement type B3500 ou UNIVAC 1108, dispositif de réallocation type 10070, segmentation), un programme possédait un emplacement fixe en mémoire centrale et dans ce cas les adresses étaient figées juste avant le chargement (étape 5) et toute utilisation en multi-programmation obligeait soit à laisser le programme résident en mémoire centrale durant toute sa vie, soit à le transférer vers un support extérieur avec obligation de le remettre aux mêmes emplacements en mémoire centrale lors du transfert de retour.

Puis des mécanismes de translation dynamique d'adresses furent introduits de façon à rendre les programmes indépendants de l'emplacement physique occupé en mémoire centrale. Ce fut l'apparition de la notion de mémoire virtuelle (cf. Wilkes [WI 68]). Cependant, chaque utilisateur se voyait affecter une mémoire virtuelle indépendamment des mémoires virtuelles des autres utilisateurs et les problèmes d'allocation (édition de lien) à l'intérieur d'une même mémoire virtuelle se posaient (car cette mémoire est encore linéaire). Tout ceci n'est pas très grave, dans la mesure où on relie ensemble dans un espace donné des programmes et des données que l'on recopie physiquement dans cet espace (même si ces programmes et ces données sont les mêmes que ceux utilisés dans un autre espace virtuel).

La figure 3 donne un exemple de ce cas. L'espace virtuel de l'utilisateur 1 est composé des objets A, B et E ; celui de l'utilisateur 2 des objets A, C, D et E. Les adresses virtuelles (d'implantation) correspondant aux mêmes objets peuvent être identiques (A) ou différentes (E). Les objets A et E qui sont tous deux dans deux espaces distincts sont dupliqués car ils ont des adresses distinctes et de plus ils peuvent éventuellement comporter des variables propres à chaque exécution et la valeur d'une de ces variables peut très bien être une adresse virtuelle, donc dépendre de l'espace virtuel.

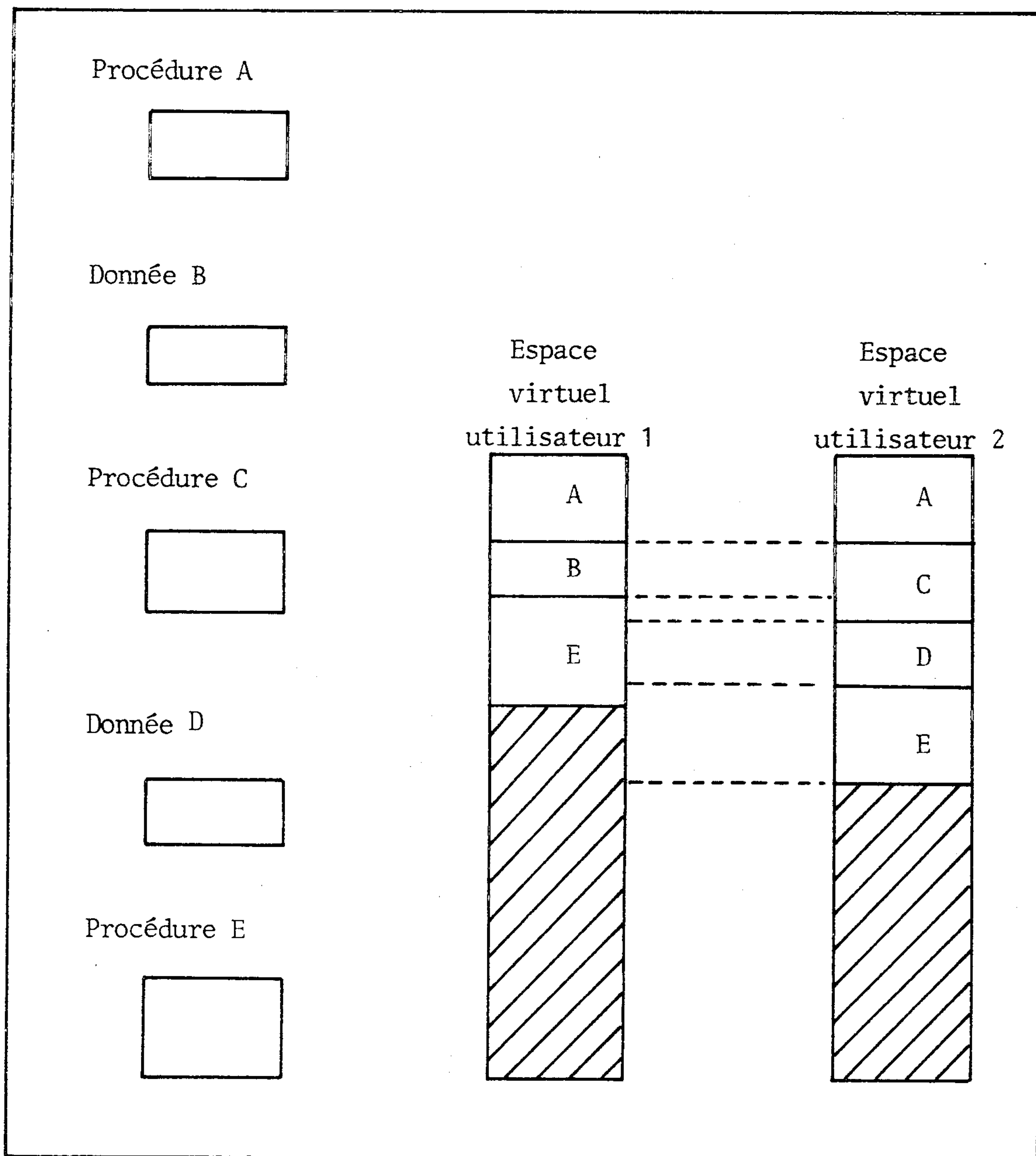


Figure 3 : Allocation dans des espaces virtuels indépendants

Considérons maintenant le problème qui consiste à vouloir introduire une interaction entre ces deux utilisateurs, par exemple vouloir une seule copie de A et E. N'avoir qu'une copie de A et E peut être le résultat des considérations suivantes :

. non duplication de procédure pour des raisons économiques (moins de place occupée sur l'espace réel) ou logiques (problème de la substitution dynamique) ;

. non duplication de données à cause des problèmes de mise à jour qui se posent.

Dans ce cas, chaque adresse manipulée est interprétée par le dispositif de translation dynamique et est donc indépendante du contexte d'exécution.

Introduisons maintenant l'utilisation de la segmentation telle qu'elle a été définie au chapitre 1 et par Dennis [DE 65]. Dans ce cas, le problème est identique, car il faut affecter à chaque objet un numéro (ou nom) de segment unique à un instant donné. Dans notre cas, la segmentation permet de s'affranchir de la taille d'un objet qui peut varier dans le temps (grossir, diminuer).

Une identification dans un segment est donnée par :

< nom local >, < déplacement >

où <nom local > : nom local du segment

< déplacement > : déplacement dans le segment.

L'avantage de la segmentation réside aussi dans la possibilité d'introduction de types d'objets différents de ceux de type procédure ou donnée (cf. chapitres 1 et 2).

Nous ne nous intéressons pas plus avant aux différents avantages de la segmentation dans ce paragraphe. Nous supposons que chaque utilisateur et pour être plus précis chaque processus, est défini par un ensemble de noms locaux (espace d'exécution), ensemble qui est très souvent matérialisé dans la pratique par une table de segments. Nous supposons, pour l'instant, que la correspondance entre un identificateur (programme source) et un nom local (numéro de segment) est effectuée par un compilateur. Cependant, le problème de conflits de noms (cf. figure 3) sur les identificateurs subsiste toujours et est transposé au niveau des noms locaux.

Nous nous intéressons à ce qui est généré par un compilateur ou un assembleur et à la façon dont sont résolus (et transformés) les problèmes de désignation au moment de l'exécution.

Nous n'indiquons pas, dans ce paragraphe, comment un nom local est chargé avec un nom universel (cf. § 3), ni comment on peut retrouver physiquement un objet étant donné son nom universel. Le lecteur peut consulter notamment [FE 74] [LI 73] [WI 68] [WI 72] et dans le cas de GEMAU (Annexe 2 : D.1 et D.2).

Nous supposons qu'une adresse est toujours donnée par :

< nom local >, < déplacement >

mais nous verrons au paragraphe 2.8. quelques idées sur la façon de réduire l'encombrement de telles adresses dans une instruction par l'introduction de *surnoms* ou de *registres de base*.

Dans le cadre d'un même exemple, nous allons examiner plusieurs solutions possibles pour les problèmes de désignation et de références inter-objets. Ces solutions ont respectivement pour nom :

a. espace d'exécution statique,

b. espace d'exécution statique avec section de liaison,

c. information préfixée dans les segments,

d. espace d'exécution récursif,

e. espace d'exécution évolutif.

On peut dire que ces solutions se divisent en deux grandes classes selon que le contenu des noms locaux est directement manipulable (c, d) ou non (a, b, e). Toutes ces solutions (sauf a) ont cependant un point commun qui est qu'un nouvel espace d'exécution est défini à chaque appel de procédure.

2.2. Espace d'exécution statique

Considérons l'exemple défini par la figure 4 :

```
procédure A ;
début
   externe D : segment, E : procédure ;
   :
   D[i] := j ;
   appeler E ;
   :
fin ;
```

Figure 4

Nous définissons par la déclaration "externe" un objet externe à la procédure ; par D[i] le $i^{ème}$ mot du segment de nom "D" ; les lettres minuscules i et j représentent des valeurs immédiates entières.

La solution d'un espace d'exécution statique consiste à imposer à tous les objets le même nom local dans tous les espaces d'exécution des différents processus. Dans l'exemple précédent c'est relativement facile, mais avec l'inconvénient que chaque objet doive avoir un nom unique qui soit le même pour tous les utilisateurs et que ce nom unique soit connu de tous les utilisateurs. Ceci est une contrainte sérieuse et un abandon de l'idée qui consiste à définir des applications à l'aide de programmes écrits indépendamment les un des autres. En particulier, la propriété de modularité ne peut plus être respectée.

Cependant, il y a beaucoup plus grave, c'est le problème des objets de type 'procédure' avec paramètres. Considérons l'exemple de la figure 5 où '<u>local</u>' définit un objet qui n'a de signification que pour la procédure A.

```
procédure A (B : segment) ;
début
   externe D : segment, E : procédure ;
   local F : segment ;
   :
   :
   F[k] := D[i] := B[j] ;
   appeler E ;
   :
   :
fin ;
```

Figure 5

L'affectation d'un nom local unique aux objets A, D et E ne pose pas de problème. Par contre, quel nom affecter à B, car c'est seulement au moment de l'appel à A que cette affectation sera connue.

Une solution consiste aussi à affecter un nom (dans l'espace d'exécution) unique pour B et lors de l'exécution, à charger ce nom avec le paramètre effectif. Le nom local de F peut être affecté de la même façon*.

---

* Ces différentes affectations sont faites par le noyau lors de l'appel de procédure.

Considérons l'espace d'exécution donné par la figure 6.a. Dans cet exemple, nous supposons que les numéros (nom locaux) suivants ont été affectés :

A - 1, B - 2, D - 3, E - 4, F - 5, G - 6,

cependant, 2 est un numéro auquel aucun segment n'est physiquement affecté (paramètre formel) avant tout appel à A ; il en est de même pour 5 (objet local à la procédure A).

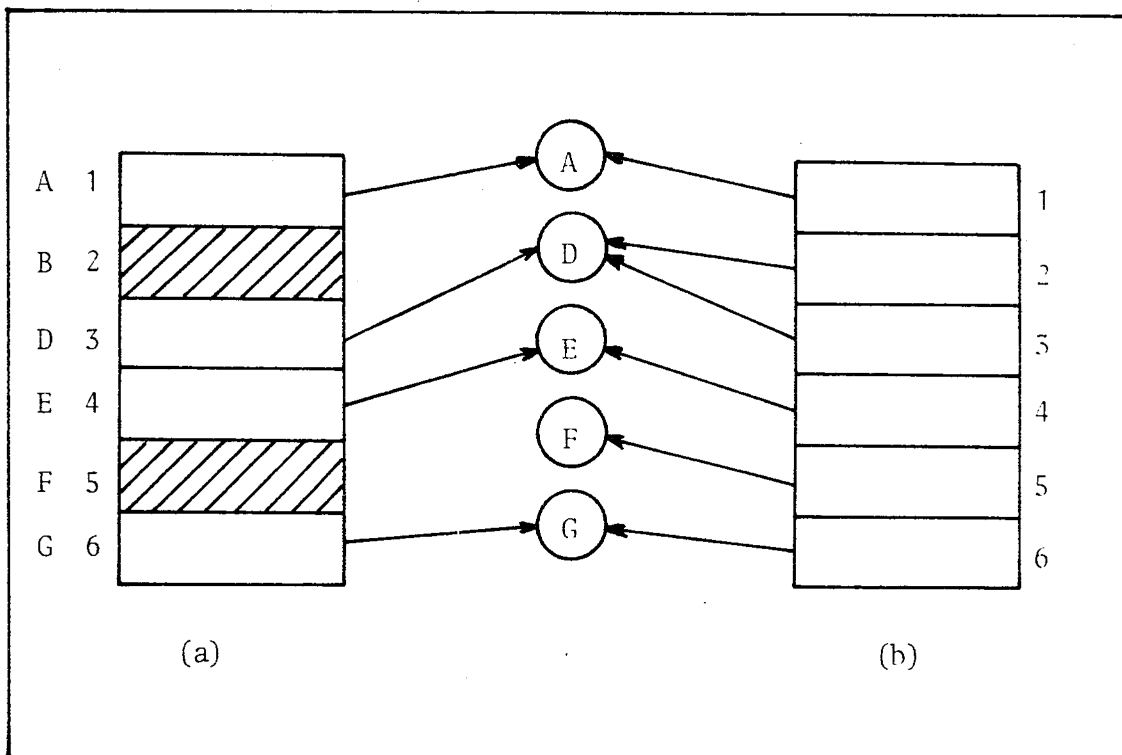


Figure 6 : Passage de paramètres dans l'espace d'exécution statique

La figure 6.b. donne l'état de l'espace d'exécution après l'exécution de appeler(1,3), i.e. en langage source : appeler(A,D).

Cette solution ne marche pas pour les appels récursifs dans le cas d'objets locaux, car alors on ne peut utiliser le même nom local (et par conséquent le même objet) pour des activations différentes de la procédure. De plus, pour permettre plusieurs appels successifs (i.e. l'un après l'autre) de A avec des paramètres différents, il faut "délier" le nom local 2 après chaque exécution de A ; une solution identique peut être prise pour le nom local 5. La figure 7 donne les espaces d'exécution des utilisateurs U1 et U2 après l'exécution respective de appeler(1,3) et appeler(1,6).

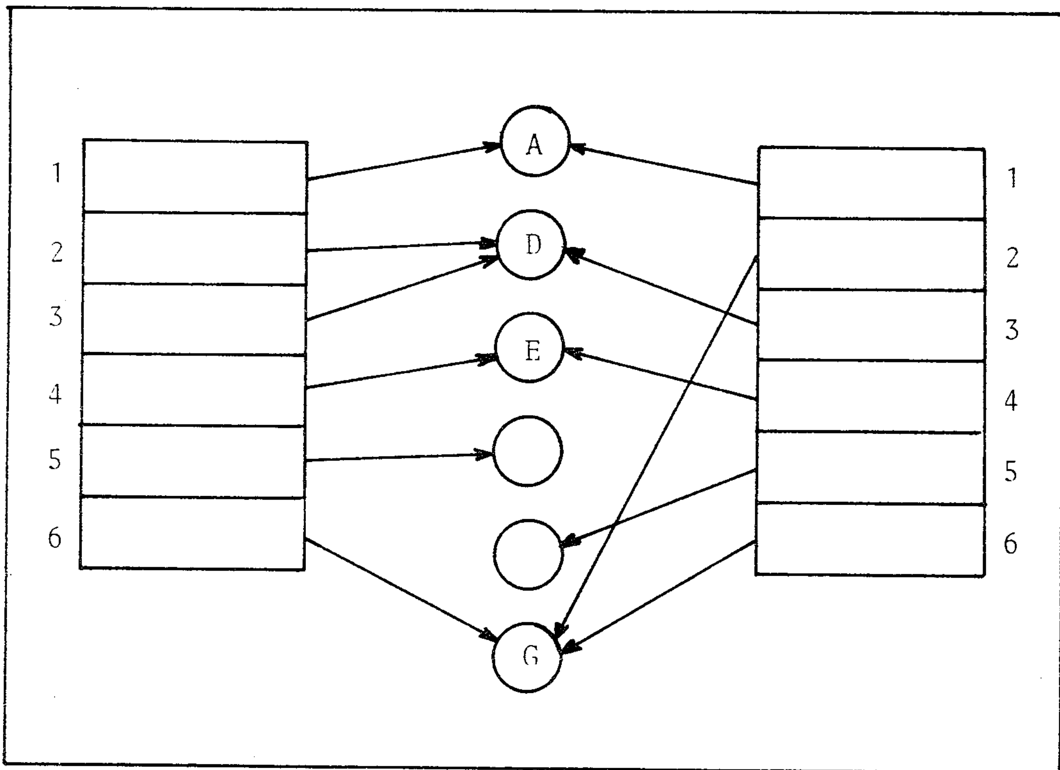


Figure 7 : Partage avec des espaces d'exécution statiques

Cette solution n'est en fait acceptable que dans l'une au moins des trois conditions suivantes :

a. les compilations indépendantes sont prohibées (cas des machines Burroughs),

b. le nombre des objets partagés est réduit,

c. le nombre des noms locaux (espace d'exécution) est "infini".

La condition a. est contraire aux propriétés de modularité.

La condition b. permet d'adopter une solution dans laquelle on décide de réserver un sous-ensemble de noms locaux aux objets partagés et un autre aux objets non partagés. Cette solution est donnée par la figure 8 ; c'est celle qui est utilisée par SIRIS 8 ou VM 370 ; par exemple les objets partagés correspondent aux programmes et données du système, aux compilateurs, etc ...

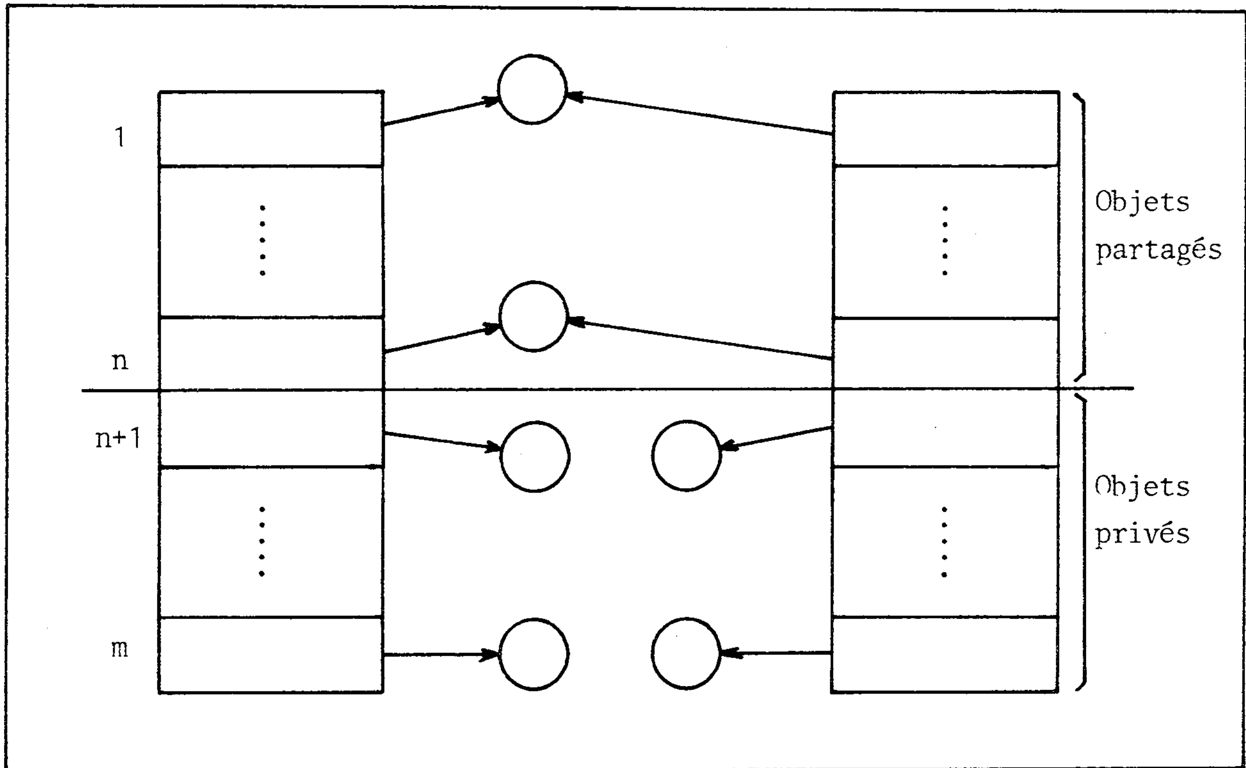


Figure 8 : Espace d'exécution statique partitionné

Avec une telle solution, le même nom local peut correspondre à des objets différents (objets privés).

La condition c. ne peut être atteinte que si l'on sait définir ce qu'est un "infini" raisonnable (2^{24}, 2^{32}, 2^{64}, ?) et plus le nombre maximal de noms locaux est grand, plus la taille occupée par une adresse est grande (< nom local >, < déplacement >). On trouvera au § 2.8. une solution pour réduire l'encombrement des adresses à l'intérieur des instructions.

Le problème le plus important consiste à être sûr que le maximum choisi est effectivement suffisant, de façon à ne pas avoir à effectuer de réaffectation des noms locaux uniques. Dans le cadre des systèmes, on peut citer ici l'utilisation d'une méthode analogue à celle des structures de recouvrement ("overlay").

Considérons le programme formé des procédures A, B, C, D, E et F et supposons que les relations entre ces objets soient fournies par la figure 9. Les objets situés sur la même ligne horizontale n'ont aucune relation entre eux. Les affectations de noms locaux peuvent être :

A : 1
B,C : 2
D,E,F : 3

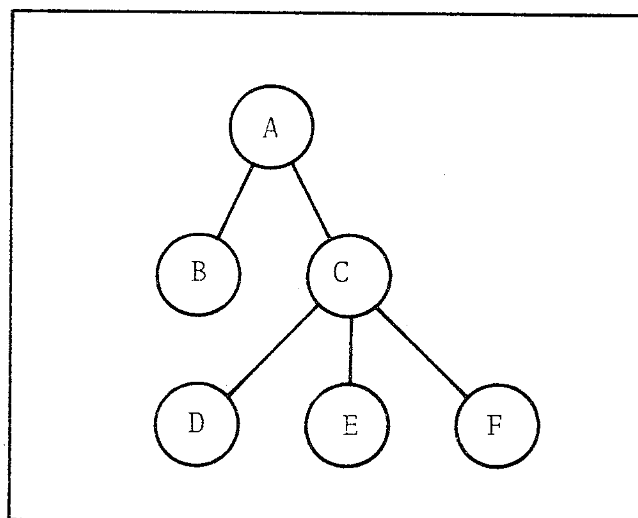


Figure 9 : Structure de recouvrement

L'espace d'exécution d'un utilisateur peut être l'un des quatre suivants :

A, B
A, C, D
A, C, E
A, C, F
| | |
LN1 LN2 LN3

Dans ce cas, il faut seulement fournir un moyen de charger un nom local avec le nom universel d'un objet.

Un dernier problème réside dans le choix des noms locaux, choix qui ne peut être laissé aux différents utilisateurs à cause des conflits possibles de noms ; il faut donc demander au système l'attribution automatique des noms locaux (mécanisme équivalent aux noms universels), ce qui explique que cette solution soit difficilement utilisable pour des systèmes universels.

2.3. Espace d'exécution statique avec section de liaison

Dans cette solution, on associe à chaque procédure un segment particulier appelé *section de liaison* et toutes les instructions de la procédure comportent des adresses qui sont des index dans cette section. Chacune des entrées dans cette section comporte le nom local de l'objet réel.

Ainsi, pour reprendre l'exemple de la figure 5, les index sont 1, 2, 3, 4 et 5 pour respectivement A, B, D, E et F. Il faut rajouter au niveau du matériel un registre qui indique le nom local de la procédure (et le déplacement courant : c'est le compteur ordinal CO), et un second registre (SL) qui indique le nom local de la section de liaison.

La figure 10 donne la représentation pour l'exemple précédent, i.e. pour appeler(A,D).

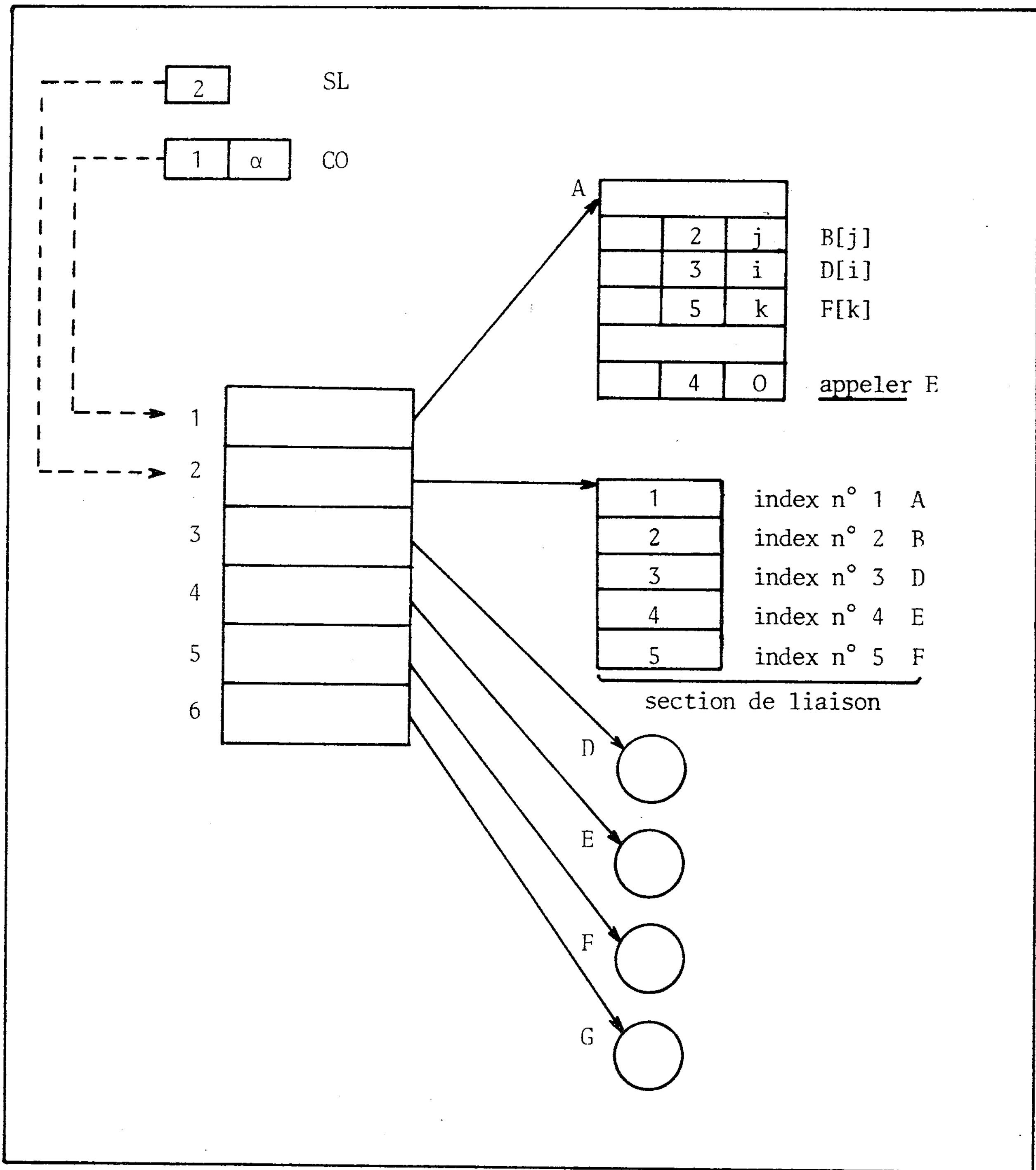


Figure 10 : Section de liaison

Cette section de liaison est créée à chaque appel de procédure et détruite lors du retour. Il faut donc avoir un modèle (section de liaison statique) associé à la procédure pour pouvoir remplir les index avec les valeurs réelles qui leur sont affectées ; cette affectation est faite par le système.

Une telle solution offre l'avantage de rendre les procédures indépendantes de leur position dans l'espace d'exécution pour pouvoir s'exécuter. Cependant, cette solution a l'inconvénient de nécessiter une indirection supplémentaire dans le dispositif de traduction dynamique d'adresse. La figure 11 donne le schéma de translation d'une adresse (d'instruction) en un nom local et un déplacement. Ce mécanisme n'existait pas dans la solution à adressage uniforme.

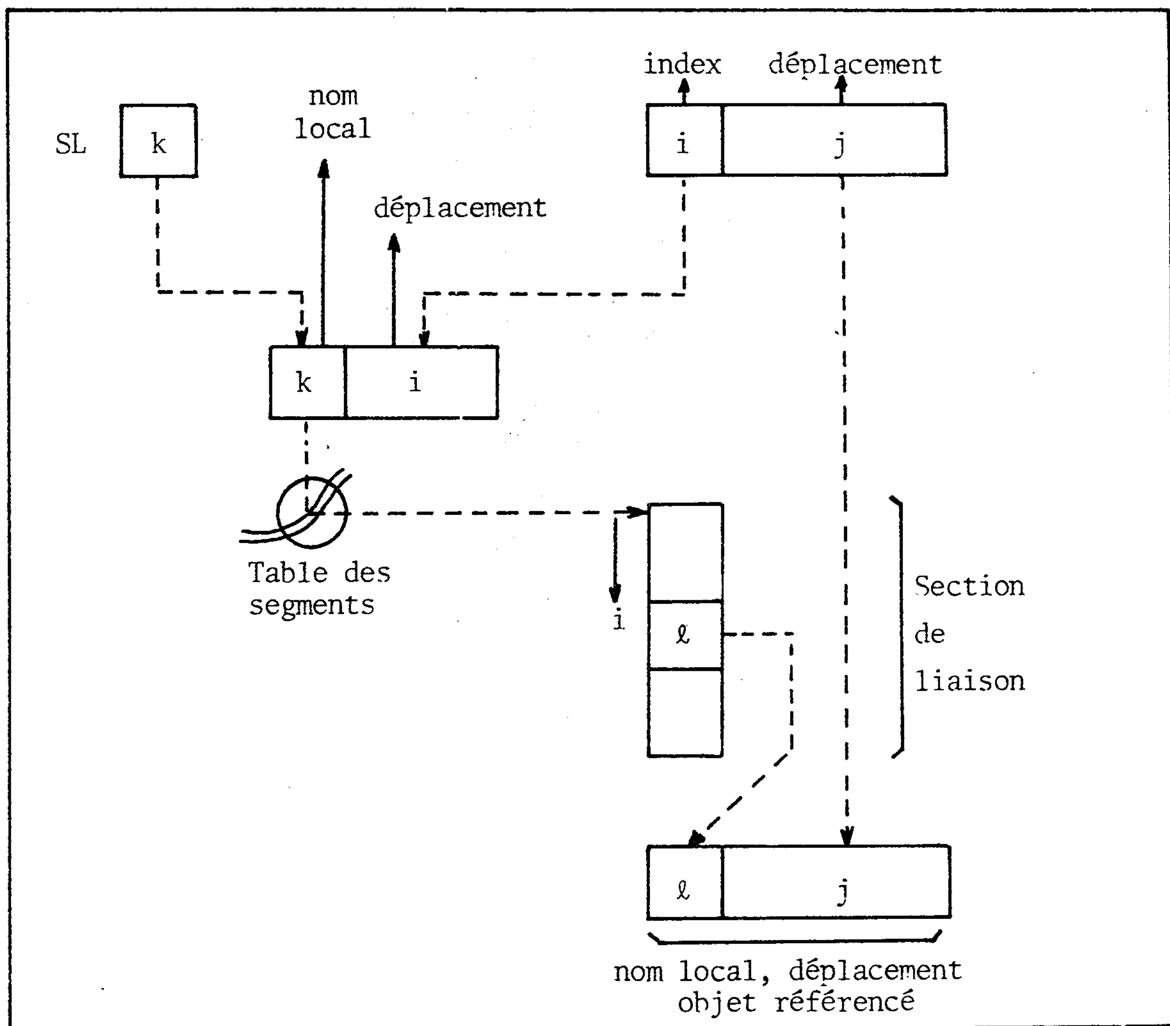


Figure 11 : Transformation d'adresse par section de liaison

Une solution améliorée de ce mécanisme peut être faite en conservant dans le registre SL non pas un nom local mais le nom universel de l'objet qui contient la section de liaison.

Pour en revenir à l'exemple 5, l'état des deux utilisateurs U1 et U2 après exécution de appeler(A,D) et appeler(A,G) donne la figure 12, dans le cas de la solution d'espace d'exécution statique avec section de liaison.

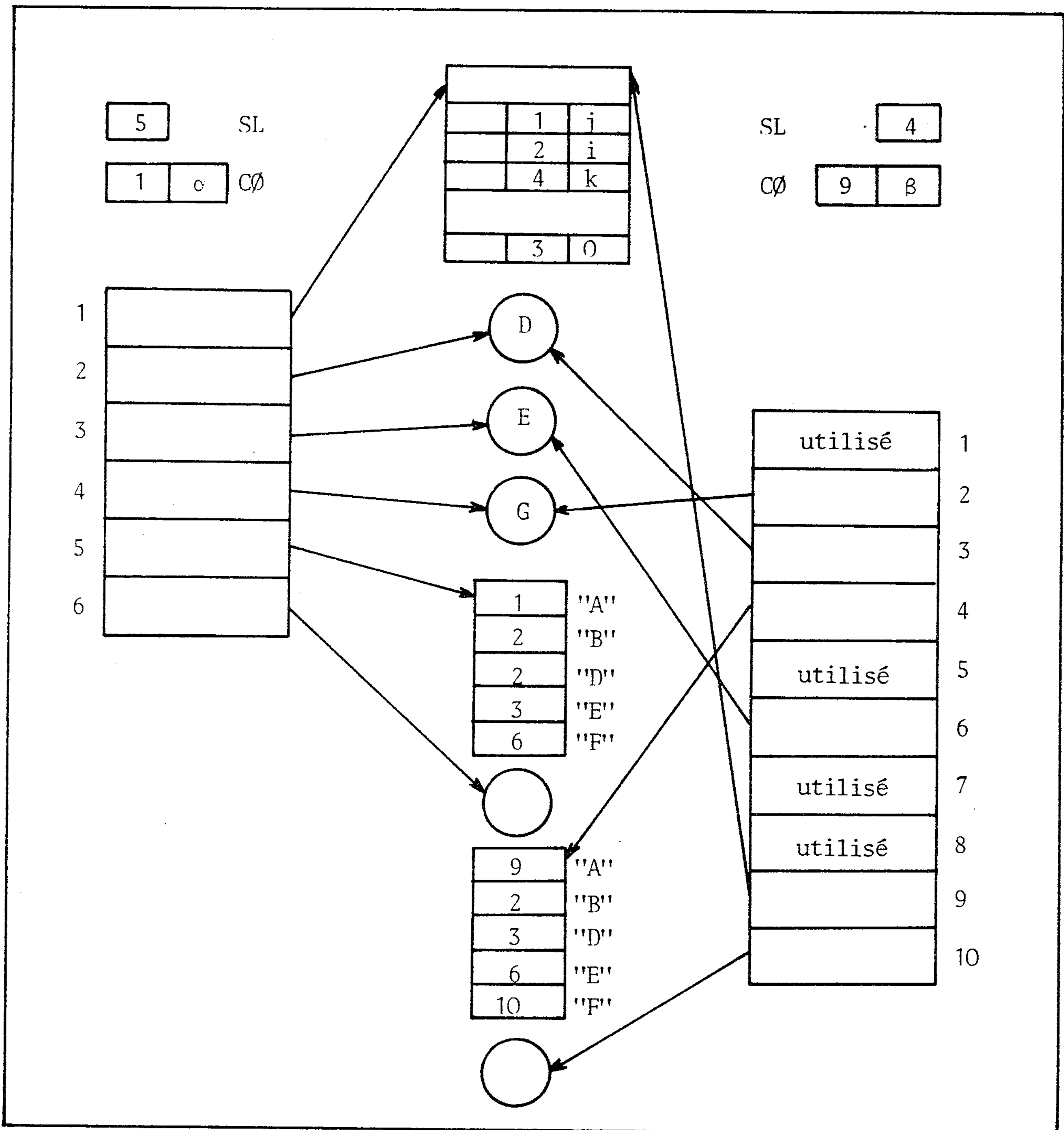


Figure 12 : Partage dans le cas des espaces d'exécution statiques avec section de liaison

L'appel de procédure nécessite, dans ce cas, la possibilité de conserver les registres SL et CØ et de les restaurer lors du retour de procédure.

Un schéma d'appel récursif ou d'appel de sous-programme (type FORTRAN) peut être réalisé. En effet, si on crée une section de liaison dynamique à chaque appel de procédure, on peut alors réaliser des procédures classiques récursives. Si par contre on ne crée cette section de liaison que lors du premier appel à la procédure et que l'on utilise ensuite toujours la même section de liaison, on a un appel de type sous-programmes (i.e. avec variables locales rémanentes).

Cette solution (de section de liaison) qui a été adoptée pour MULTICS (cf. Daley et Dennis [DA 68] et Organick [OR 72]), est présentée ici de façon beaucoup plus simple qu'elle n'est réalisée dans MULTICS, en particulier l'adressage par l'intermédiaire de la section de liaison est effectué par adressage indirect.

On peut mélanger aussi les deux types d'adresse :

< index >, < déplacement > et < nom local >, < déplacement >.

Dans ce cas, on peut récupérer les avantages de la solution 1 sans en avoir les inconvénients.

Nous verrons au chapitre suivant que l'inconvénient majeur de cette solution réside dans les problèmes liés à la protection, car cette idée de section de liaison permet de n'adresser directement qu'une partie de l'espace d'exécution tout en permettant d'adresser directement des objets non visibles par la procédure à l'aide d'adresse du type :

< nom local >, < déplacement >.

2.4. Information préfixée dans les segments

Un des inconvénients majeurs des deux types de solutions précédentes réside dans le fait que toutes les adresses manipulées par une procédure sont de la forme :

< nom local >, < déplacement >.

Fabry [FA 74], Lindsay [LI 73], Halton [HA 72] et Williams (Plessey PP 250) [WI 72] proposent une solution alternée avec l'introduction "d'adresse absolue" (au sens nom universel).

La solution proposée consiste à pouvoir mélanger dans un même segment des données (ou/et des instructions) avec des descripteurs d'objets ("capability"). Cependant il ne faut pas que ces descripteurs puissent être manipulés comme des données normales, car ils contiennent notamment le nom universel d'un objet ainsi que la protection qui est associée à cet objet (équivalent d'un nom local). Pour cela on introduit un champ supplémentaire pour chaque mot des segments, ce champ est appelé *préfixe* (en anglais "*tag field*") et indique le type du mot : descripteur ou donnée. Il est impossible de fabriquer ou de modifier un descripteur à l'aide d'instructions classiques, notamment de type arithmétique. Il faut introduire de nouvelles instructions comme les primitives GEMAU et des registres spéciaux (noms locaux dans un espace d'exécution), ce qui est la solution du PP 250.

Dans une seconde solution préconisée par Fabry, tous les registres sont banalisés (registres généraux, noms locaux, ..) et le même système de préfixe indique le type de l'information conservée dans un registre. Ceci correspond assez bien à une solution classique d'adressage par registre de base et déplacement (type IBM 360) dans laquelle on interdirait la manipulation des registres affectés à une adresse par des instructions non spécialisées. Dans le cas où un descripteur est accédé par une instruction arithmétique, un déroutement se produit.

Dans la résolution de l'exemple 5, on peut mettre, directement à la compilation, les noms universels de D et E dans le segment A en utilisant des descripteurs. Par contre, ceci n'est possible ni pour B qui est un paramètre formel, ni pour F qui est un segment local à la procédure A. Ces descripteurs peuvent être utilisés partout où des références apparaissent. La figure 13 donne un exemple pour la procédure A, dans le cas où l'objet E a été lié directement lors de la compilation de l'objet A.

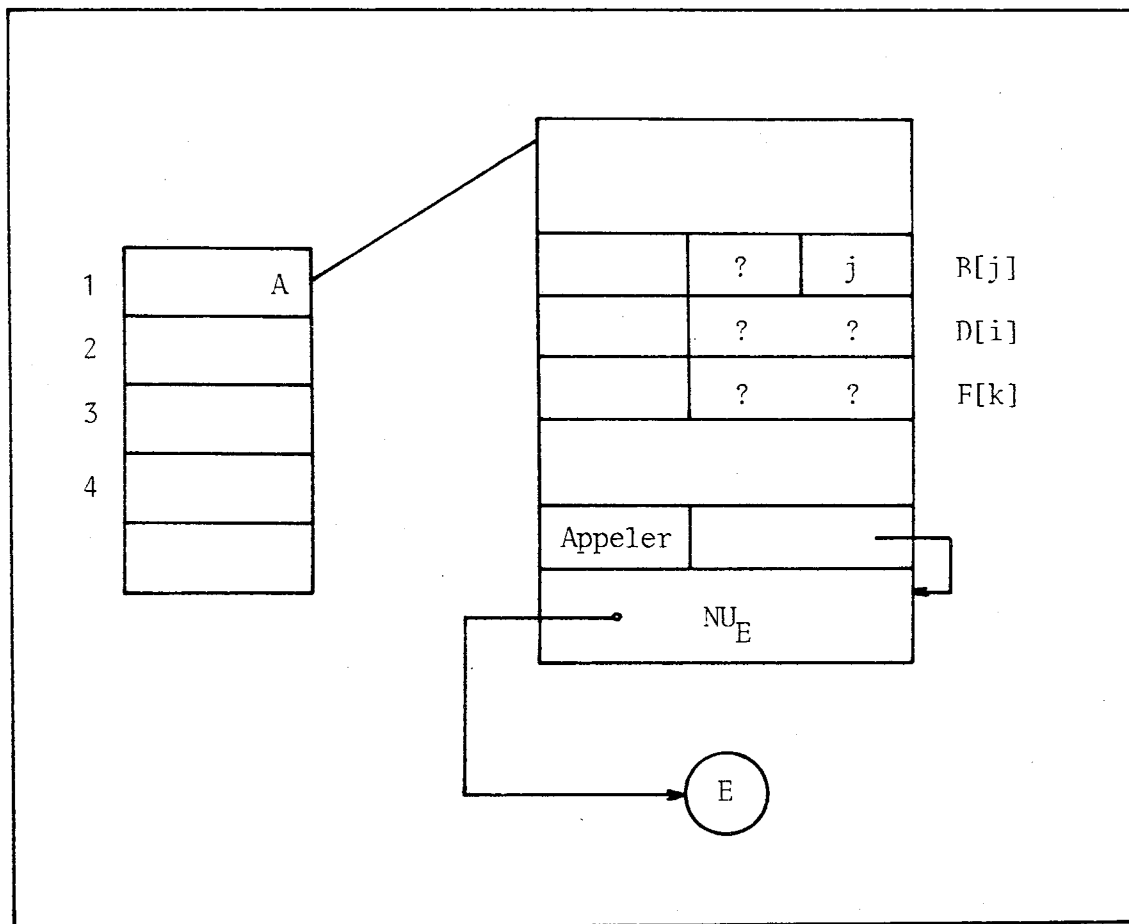


Figure 13 : Adressage par descripteur préfixé

Le problème est plus compliqué si on veut effectuer la compilation de D[i], car il faut pouvoir indiquer dans la partie adresse de l'instruction de référence à D[i], une information du type < nom local >, < déplacement > pour référencer le descripteur de D, et il faut de plus un moyen d'indiquer le déplacement i dans ce segment D.

La solution consiste à placer, dans les instructions de A, une instruction supplémentaire pour charger un nom local (ou un registre banalisé pour Fabry) avec le descripteur de D, puis à utiliser ce nom local pour toute référence ultérieure à D.

De même que l'on peut charger un registre ou un nom local avec un descripteur depuis un segment, l'opération inverse est possible. On est ramenés à la gestion manuelle (i.e. par le programme) des registres d'adressage, ce qui nous semble être un pas en arrière important dans la mesure où, du côté des langages, on tend de plus en plus à s'affranchir de la gestion manuelle de registres en utilisant des structures plus naturelles (piles, listes, ..).

Cette solution, pour les paramètres et les objets locaux, peut utiliser l'une des trois solutions suivantes :

- espace d'exécution statique (cas de Fabry [FA 74]),
- espace d'exécution avec section de liaison,
- espace d'exécution évolutif (cas du PP 250 [WI 72]).

Par exemple, dans le cas de la procédure A et de la procédure E liée à la compilation de A, on aurait la génération donnée par la figure 14. Dans ce cas, les noms locaux affectés à A, B, D et F sont respectivement 1, 2, 3 et 4.

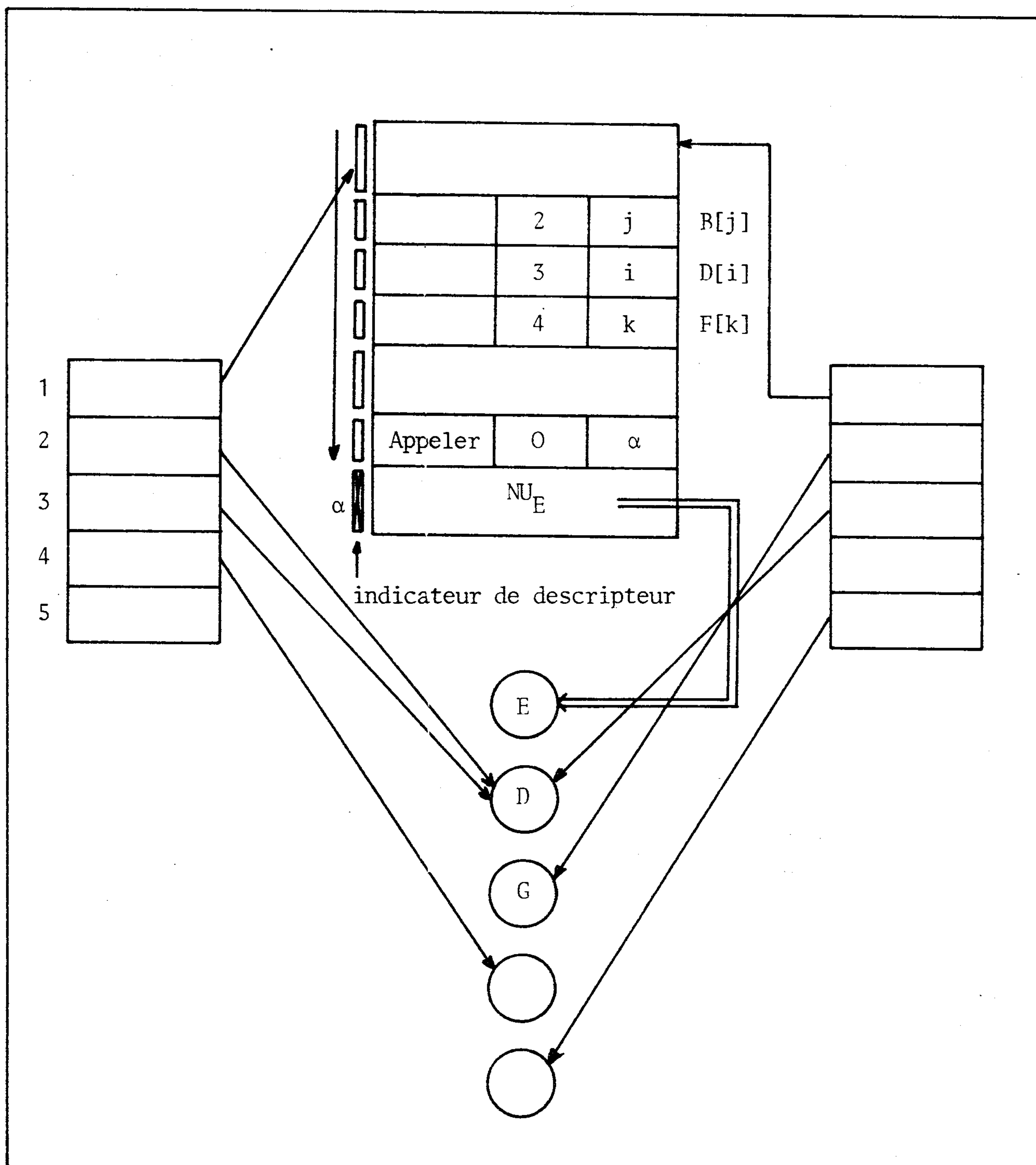


Figure 14 : Partage dans le cas d'espace d'exécution évolutif

2.5. Espace d'exécution récursif, machines récursives

La plupart des modèles que nous avons vus jusqu'à maintenant possédaient une caractéristique commune : l'existence d'un nom universel unique pour désigner un objet du système. Cette solution a été rejetée par un certain nombre d'études (Auroux [AU 72], Goldberg [GO 73] et Lauer [LA 74]) qui proposent une solution alternée connue sous le nom *d'espaces d'exécution récursifs*. Nous allons développer ici un modèle simple d'espaces récursifs et montrer la solution préconisée par Lauer.

Considérons une machine comme la B3500 de Burroughs, où l'espace d'exécution d'un processus utilisateur est définie par un seul segment décrit par une paire de registres (matériels) : base et limite (cf. figure 15). Toute adresse utilisée dans un programme est relative au début de ce programme, son adresse réelle est donc fournie par l'addition du contenu de la base à cette adresse virtuelle. La seule condition imposée est que l'adresse finale soit inférieure à la limite. Cette solution est très limitative, car on ne peut avoir qu'un seul niveau d'adressage : en effet, pour plusieurs utilisateurs il suffit de définir des partitions indépendantes. Cette solution ne permet pas le partage entre utilisateurs, la seule solution possible serait celle donnée par la figure 16.

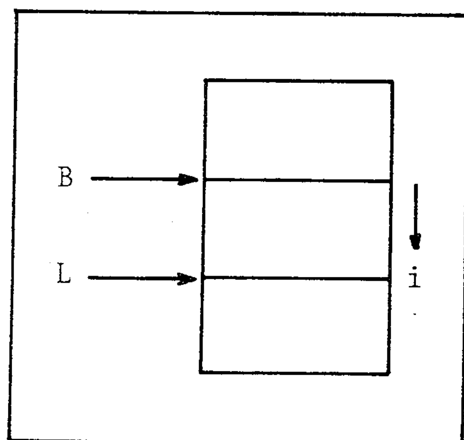


Figure 15 : Adressage B3500

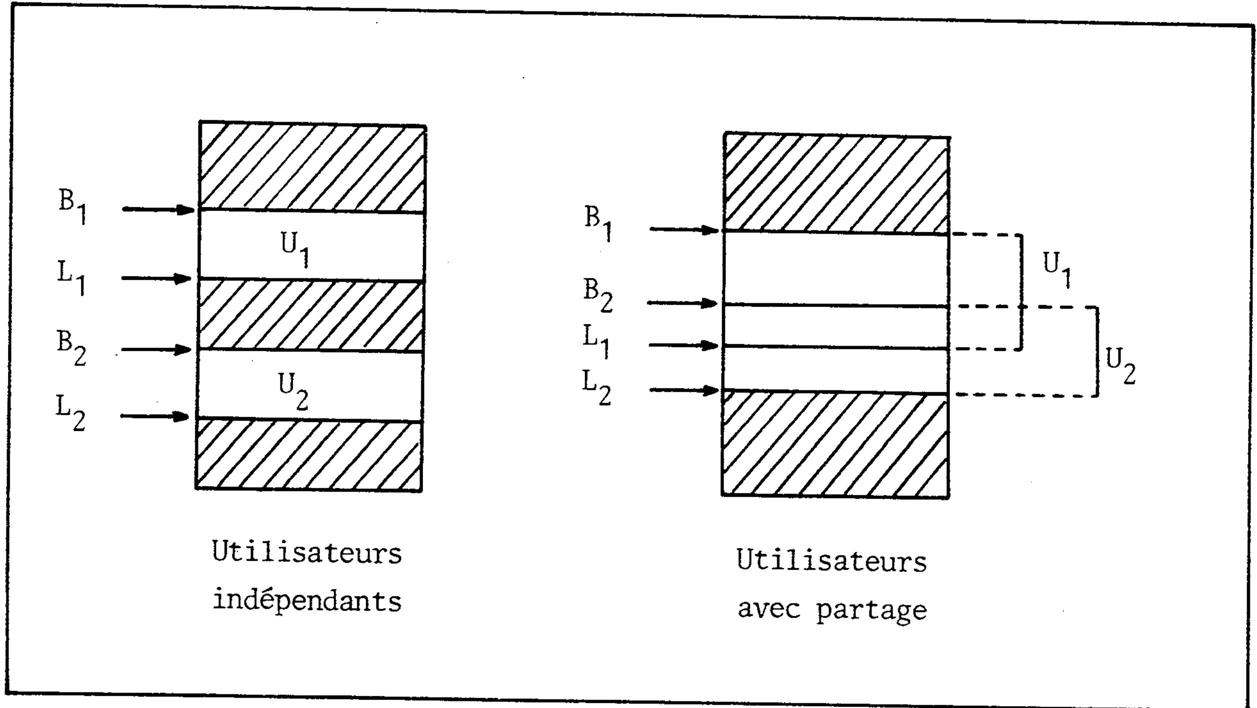


Figure 16 : Utilisation B3500 en multiprogrammation

On veut, sinon définir le partage entre deux utilisateurs, du moins pouvoir définir un schéma hiérarchique des espaces mémoire. C'est-à-dire avoir des niveaux aussi bien de processus que d'adressage, chacun de ces niveaux étant défini en termes du niveau inférieur. La figure 17 définit la hiérarchie entre les systèmes, sous-systèmes et utilisateurs.

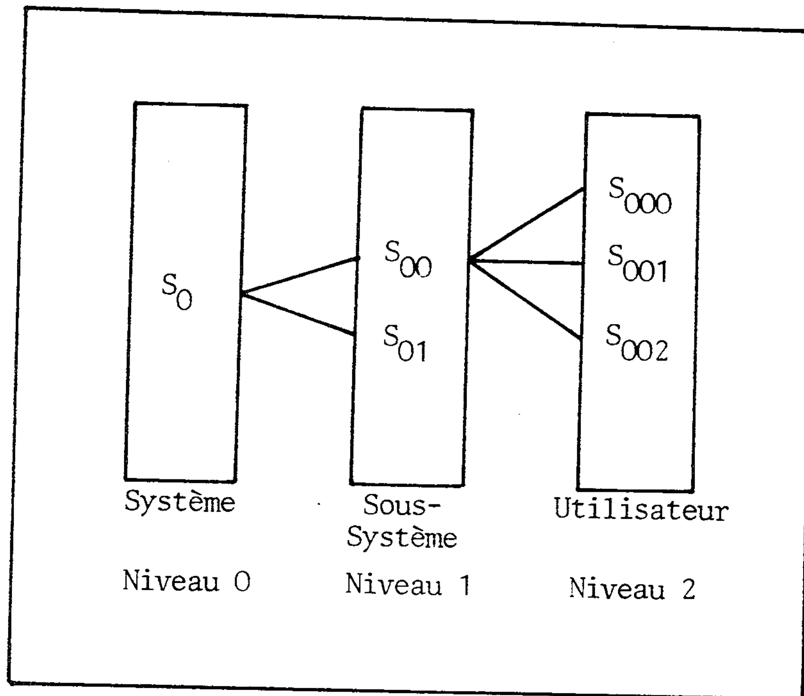


Figure 17 : Hiérarchie système - sous-système

Pour chacun des sous-systèmes, nous définissons l'espace d'exécution en termes de son espace d'exécution courant (celui du créateur du sous-système).

La figure 18 donne un exemple d'espaces emboités pour la hiérarchie définie par la figure 17.

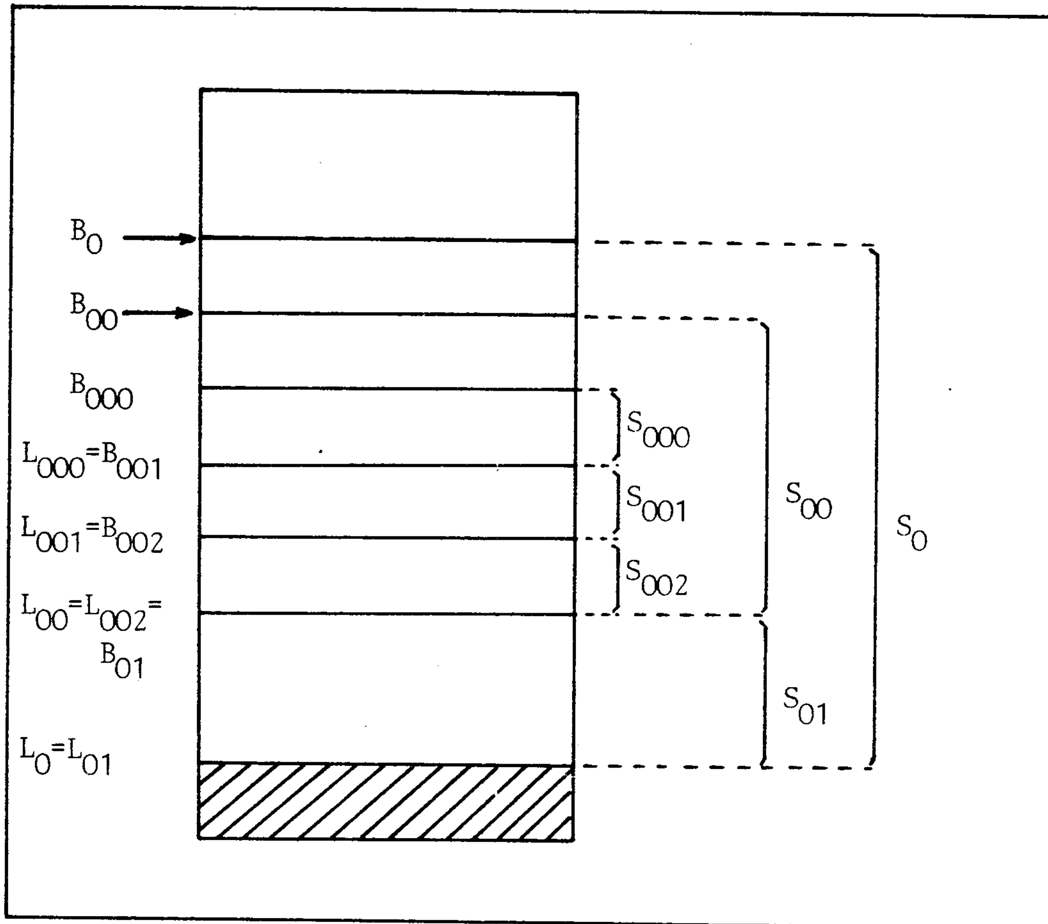


Figure 18 : Espaces emboîtés

S_0 est défini en termes d'adresses réelles $(b_0, l_0) \rightarrow B_0 = b_0$, $L_0 = l_0$

où B_i : registre de base

L_i : registre limite

b_i : adresse

l_i : adresse

S_{00} est défini par S_0 en termes d'adresses relatives à S_0, c'est-à-dire par (b_{00}, l_{00}).

Ce qui donne pour la définition réelle de l'espace, c'est-à-dire lors de l'accès par S_{00} :

$B_{00} = b_0 + b_{00}$, $L_{00} = b_0 + l_{00}$

à condition que $b_{00} \leqslant l_{00} \leqslant l_0 - b_0$

et ainsi de suite. Il suffit, sur la B3500, de conserver une pile des couples (b_i, l_i) des différents niveaux et avant d'activer une procédure à un niveau donné, de fabriquer les registres de base et limite par combinaisons successives :

$B_{ijk..n} = b_i + b_{ij} + b_{ijk} + \ldots + b_{ijk..n}$

$L_{ijk..n} = b_i + b_{ij} + \ldots\ldots\ldots\ldots + l_{ijk..n}$

Lauer propose que cette transformation soit faite par le matériel pour chaque accès. De plus, il étend l'espace d'exécution à être un ensemble de segments, c'est-à-dire qu'une adresse est de la forme

< nom local >, < déplacement >

Au niveau le plus bas, ces adresses sont des adresses physiques. La création d'un sous-système S_i se fait par la définition d'une table qui contient la définition des noms locaux en termes des adresses du niveau courant. Ainsi, par exemple, on peut définir la table (ensemble des noms locaux) comme composée de deux segments :

$S_i = ((1,X,x),(1,Y,y))$

ce qui signifie que S_i est composé de deux segments ayant pour noms locaux respectivement 1 et 2, comme adresses de début X et Y, et comme longueur x et y (X et Y étant des adresses relatives au segment de nom local 1 du sous-système courant).

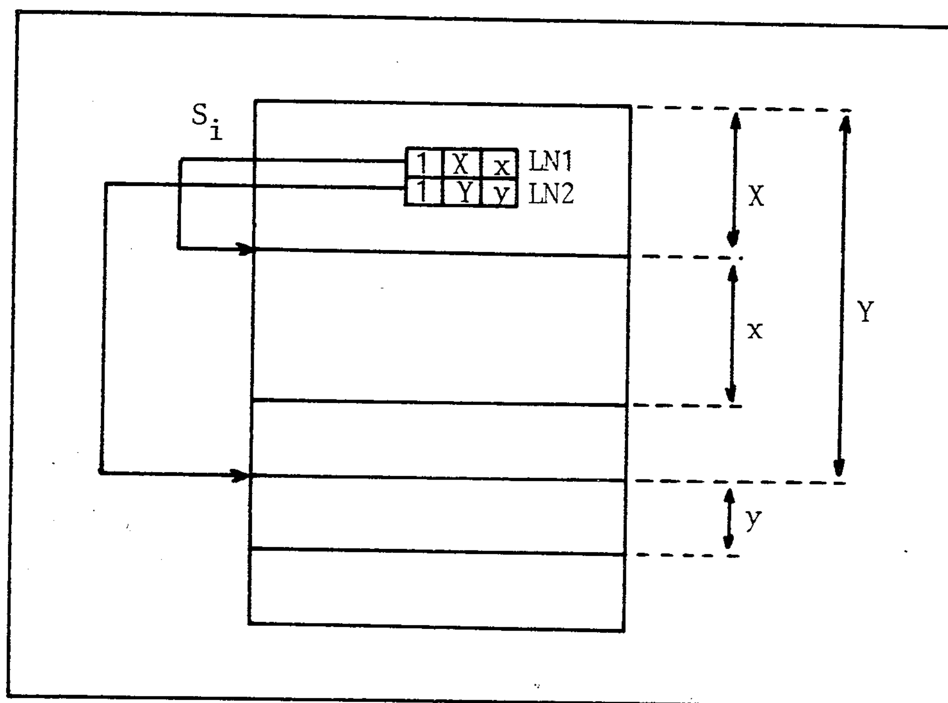


Figure 19 : Définition de S_i

L'espace d'exécution S_{ij} construit par S_i peut être défini par (cf. figure 20) :

S_{ij} = ((1,A,a),(2,B,b),(1,A+a,c))

Dans ce cas, l'accès au mot d'adresse (2,ℓ) par un processus s'exécutant dans S_{ij} est donné par :

évaluer(2,ℓ) / S_{ij}

évaluer((2,B,b) + ℓ / S_i

i.e.

évaluer((2,B+ℓ) / S_i ce qui implique $B+\ell \leqslant b$

évaluer((1,Y,y),B+ℓ) / matériel

i.e.

évaluer(1,Y+B+ℓ) / matériel si $X+B+\ell \leqslant y$

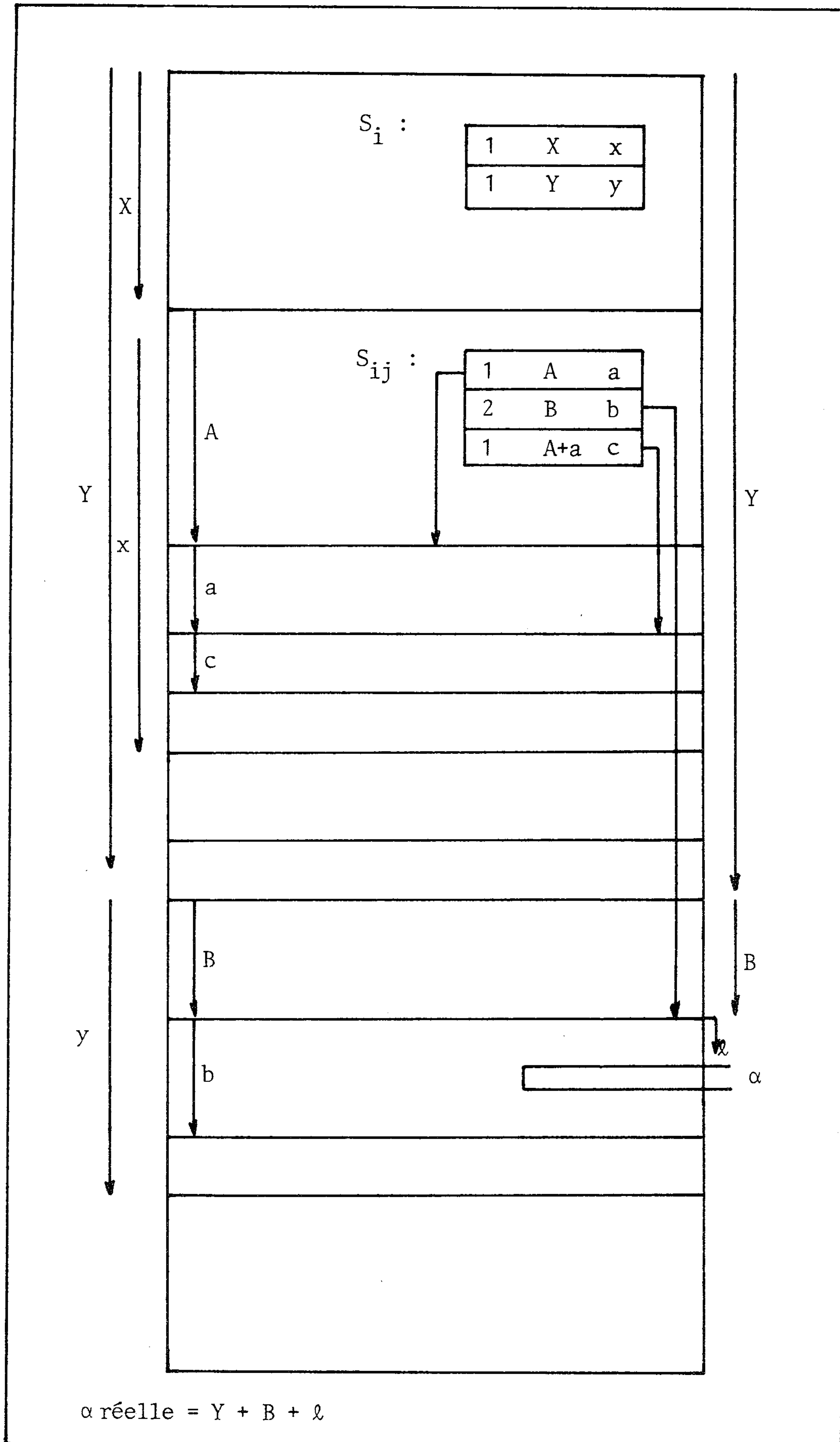


Figure 20 : Espaces d'exécution récursifs

En fait la règle est très simple : il suffit d'additionner toutes les adresses de début de segments qui contiennent le mot terminal (à accéder). Ces adresses sont toujours relatives au segment immédiatement englobant, i.e. du sous-système englobant.

De même que dans les systèmes à descripteur,on peut inclure dans les noms locaux un champ pour définir la protection. Cependant, la différence essentielle est qu'ici la table est fabriquée par le sous-système créateur qui peut la modifier à volonté et l'appel du sous-système correspondant est une instruction non privilégiée. Ceci est l'équivalent d'une primitive CALL de GEMAU, avec en paramètre un nouvel espace d'exécution (table des noms locaux). Les entrées (noms locaux) sont quelconques, elles ont simplement un format comme les tables de pages et de segments pour les mémoires virtuelles paginées classiques. Le contenu de ces entrées, pour la partie adresse, est relatif à un sous-système, ce qui élimine le besoin de noms universels ; il est alors possible d'accéder des emplacements qui ne sont pas dans le sous-système, les pointeurs dans les espaces d'exécution ne peuvent pas être contrefaits par l'utilisateur s'ils sont définis par le sous-système englobant.

Pour chaque processus le noyau maintient une pile de la table des noms locaux du sous-système courant et de tous les sous-systèmes englobants. Durant l'accès, le matériel vérifie la validité des accès dans les tables intermédiaires (mécanisme équivalent à celui de l'évaluation des liens dans GEMAU) et en cas de refus, le niveau de sous-système où le refus a eut lieu permet de connaître le traitement à effectuer.

La figure 21 donne un exemple de résolution du problème de la procédure A (solution du problème de la figure 5) dans le cas des espaces d'exécution récursifs.

Nous supposons que l'espace d'exécution du sous-système S est donné par les noms locaux

LN1 espace de travail
LN2 A
LN3 D
LN4 E
LN5 G

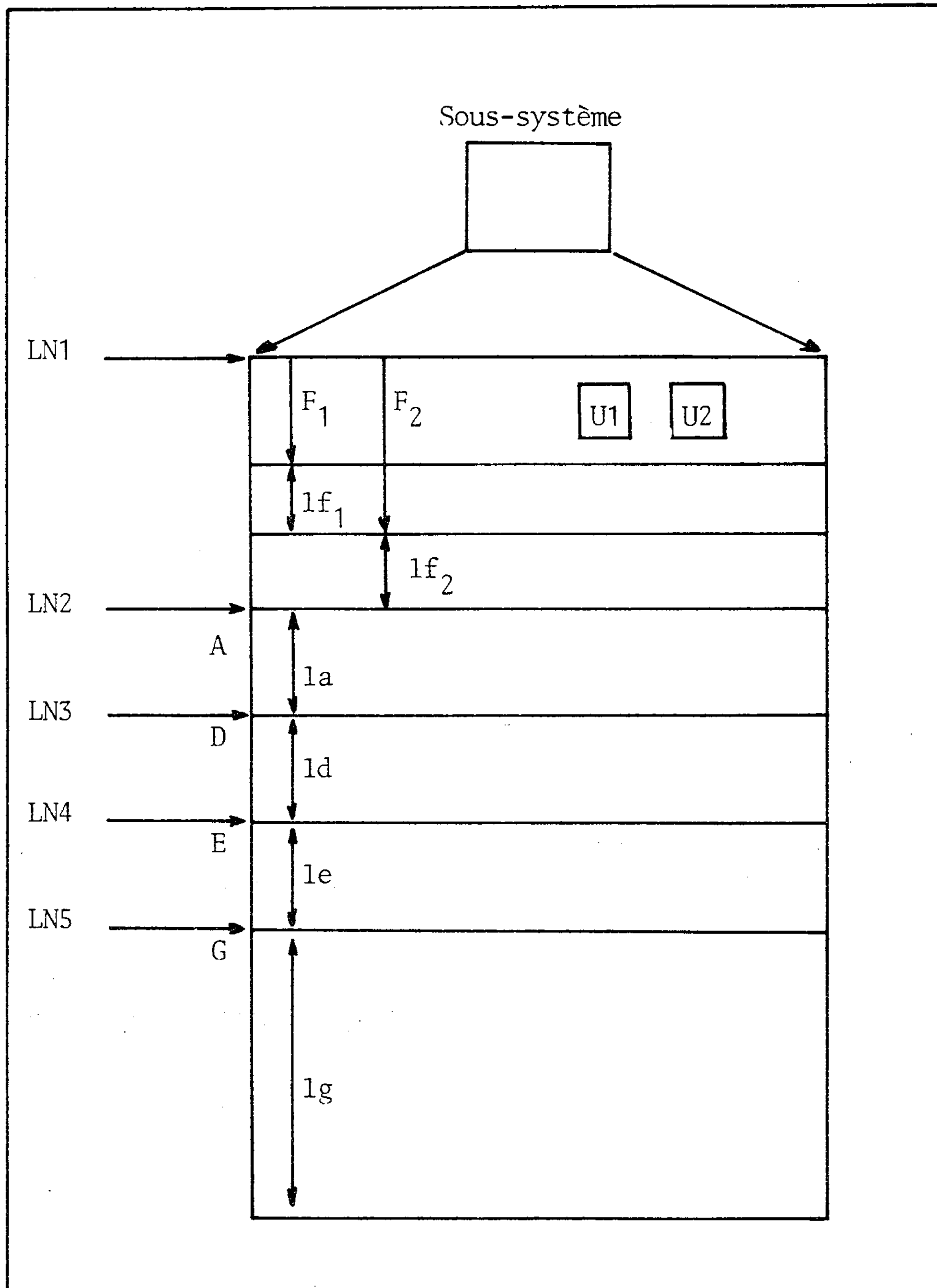


Figure 21 : Solution du partage dans les machines récursives

L'exécution de <u>appeler</u>(A,D) par U1 et <u>appeler</u>(A,G) par U2 se traduit par la fabrication des espaces d'exécution suivants :

LN1 :	2,0,1a	2,0,1a	A
LN2 :	3,0,1d	5,0,1g	B
LN3 :	3,0,1d	3,0,1d	D
LN4 :	4,0,1e	4,0,1e	E
LN5 :	$1,F_1,1f_1$	$1,F_2,1f_2$	F

Les objets temporaires sont pris dans un poöl commun (ici le segment LN1 du sous-système).

Le partage de segments entre deux sous-systèmes est très facile à réaliser.

Le système proposé par Lauer peut être facilement étendu à des objets de type différent de celui d'un segment, par exemple : périphérique, sémaphore, .. Ceci existe déjà dans certaines machines où les périphériques sont adressables comme des positions mémoire ; dans ce cas une instruction d'entrée-sortie est un transfert d'une adresse à une autre adresse, les deux adresses pouvant correspondre à deux périphériques, auquel cas un transfert de données s'effectue sans transiter par la mémoire.

La solution proposée par Lauer est très séduisante et est déjà utilisée (de façon simple) par les systèmes de machines virtuelles [AU 72] ; ses avantages sont les suivants :

. simplicité,

. possibilité pour un système d'exploitation de fonctionner sous lui-même ;

. définition par niveaux d'interprétation.

Cependant, les problèmes soulevés sont :

. impossibilité d'utilisation dans le cas des multiprocesseurs à cause de la duplication des piles de noms locaux (si des processus fonctionnent à divers niveaux), car toute modification d'un élément d'une table de noms locaux doit être transmise à tous les processus utilisant (quelque part dans leur pile) le nom local ;

. partage d'objets entre des niveaux d'interprétation sans passer par toute la chaîne des tables intermédiaires : i.e. difficulté de définition d'objets ayant un rôle semblable aux liens de GEMAU.

2.6. Espace d'exécution évolutif

Cette solution est une amélioration de la solution fournie par les sections de liaison. Si nous supposons que dans les sections de liaison nous mettons les noms universels des objets, nous n'avons plus besoin de conserver l'espace d'exécution unique dans lequel nous faisions nos indirections. L'espace d'exécution est alors la section de liaison et l'on possède une pile de ces espaces d'exécution. La figure 22 donne l'exemple des deux utilisateurs U1 et U2 après l'exécution de appeler(A,D) et de appeler(A,G).

Dans ce cas, les adresses manipulées par la procédure sont du type < nom local >,< déplacement >, où le nom local est propre à l'exécution de la procédure. Le compilateur a pu générer les noms locaux 1, 2, 3, 4 et 5 pour A, B, D, E et F.

Différents systèmes utilisent un mécanisme similaire : on peut citer HYDRA [WY 74], les travaux de Evans et Leclerc [EV 67], les suggestions de Needham [NE 74], Spier [SP 74] et GEMAU. De tous ces travaux, seuls GEMAU, HYDRA et Spier ont fait l'objet d'une expérimentation réelle à ce jour. Needham propose quatre espaces d'exécution simultanés pour un même processus : un pour les objets globaux au processus, un pour les objets rémanents de la procédure courante et utilisés par tous les processus, un pour les données locales à la procédure, et un pour les arguments (paramètres réels) de la procédure courante.

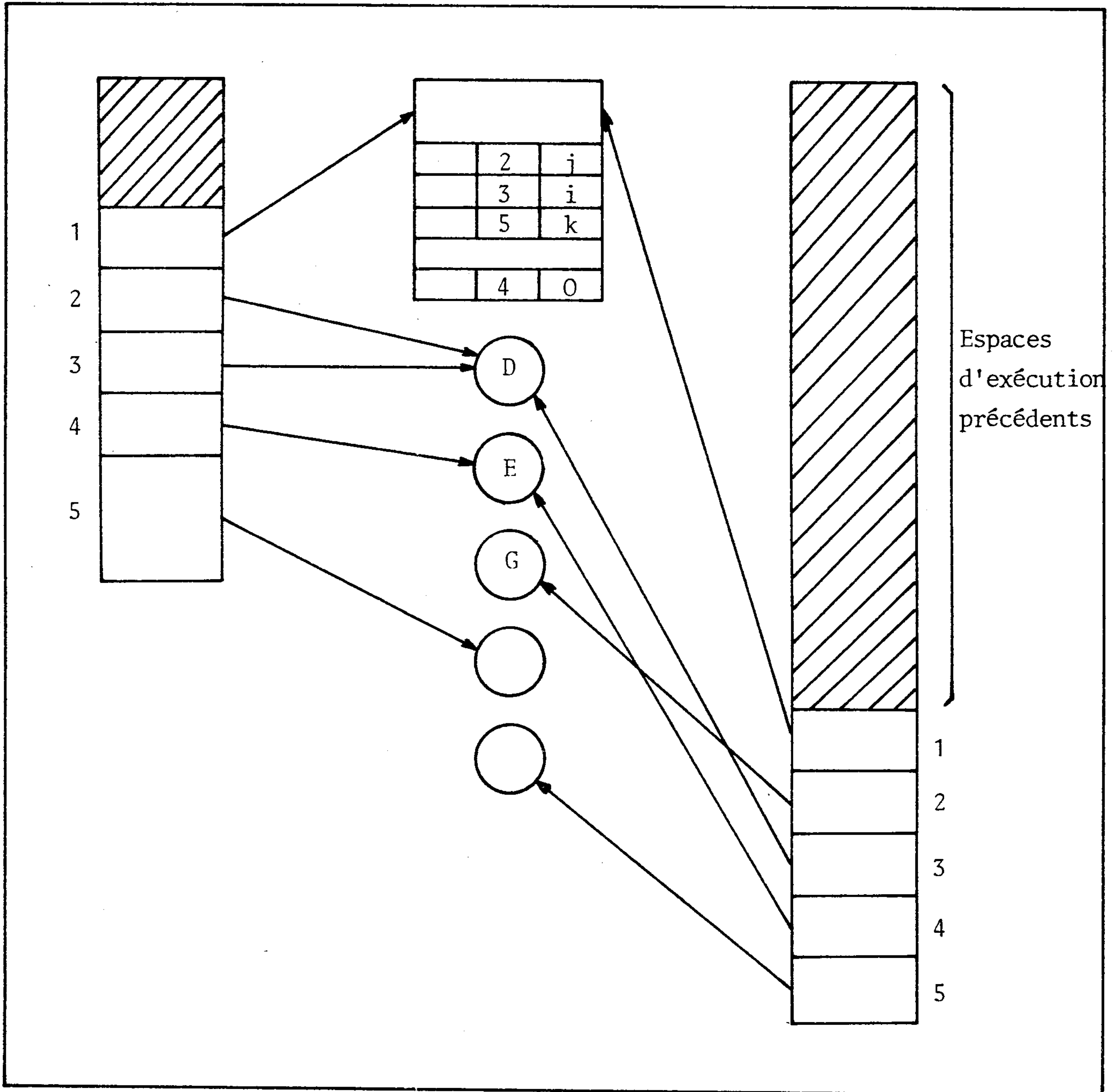


Figure 22 : Partage avec espaces d'exécution évolutifs

Dans le cas de HYDRA, une procédure contient une liste de références à d'autres objets, c'est une entité statique. Un objet de type LNS (Local Name Space) est l'équivalent de notre espace d'exécution. Il y a un seul et unique LNS par appel effectif de procédure et cet LNS disparaît quand l'exécution de la procédure est achevée.

Spier [SP 74], qui s'inspire fortement des travaux de Evans et Leclerc, a réalisé une expérience sur PDP 11/45.

2.7. Conclusion

Nous allons maintenant étudier successivement les trois aspects suivants :

a. propriétés relatives des différents types d'espaces d'exécution,

b. nature intrinsèquement évolutive des espaces d'exécution,

c. relations avec les espaces d'objets,

d. descripteurs regroupés ou répartis dans les objets.

a. Propriétés relatives des différents espaces d'exécution

Ainsi que nous venons de le voir, il y a trois degrés de liberté possibles :

. descripteur préfixé ou non,
. espace d'exécution évolutif ou non,
. existence, ou non, de noms universels.

Ceci donne donc huit solutions possibles qui sont données par la figure 23.

préfixé	évolutif	nom universel	Exemple de système
√	√	√	PP250
√	√		B6500
√		√	
√			
	√	√	GEMAU, HYDRA
	√		Machines récursives
		√	MULTICS ※
			OS 360, SIRIS 7/8, systèmes classiques

※ en fait, en partie évolutif par le biais de la section de liaison

Figure 23

Considérons les propriétés des différents types d'espaces d'exécution :

. existence d'adresses différentes dans des espaces d'exécution différents (évolutif)
. existence de références inter-segments (préfixé)
. accès possible (simple) aux paramètres
. récursivité (évolutif)
. variables locales à chaque exécution
. possibilité de multitraitement (universel)
. hiérarchie stricte (non universel), i.e. impossibilité de traverser des couches d'interprétation
. protection effectuée simplement (évolutif) ; dans le cas des espaces non évolutifs nous verrons, au chapitre suivant, qu'il faut surimposer des mécanismes supplémentaires pour assurer la protection.

La constatation simple que l'on peut faire dès maintenant, est qu'un espace statique est difficilement utilisable dans un cadre général.

b. Nature intrinsèquement évolutive des différents types d'espaces d'exécution

C'est le fait qu'un appel de procédure change l'espace d'exécution courant. Cette nature évolutive peut être considérée sous les deux points de vue différents : économique et fonctionnel. D'un point de vue économique, l'aspect évolutif favorise la connaissance des objets nécessaires à une exécution à un instant donné. Du fait que cet ensemble évolue en fonction des appels de procédure, on peut mettre la connaissance de cet ensemble à profit pour améliorer les performances du système. De plus, le problème du nombre maximal de noms locaux qui se pose dans les espaces statiques disparaît car à un instant donné ce nombre est généralement petit.

Le point de vue fonctionnel a été décrit en détail au chapitre 1, et dans Evans et Leclerc [EV 67], HYDRA [WU 74].

Cette nature évolutive est intéressante, car la majorité des systèmes utilisent une pile pour les appels de procédure, d'où notre but qui a été d'utiliser le mécanisme à fond et non d'en laisser la définition à l'utilisateur.

De plus, la solution statique est incluse dans la solution évolutive, car il suffit, par exemple dans le cas de GEMAU, de ne pas effectuer les appels de procédure par l'intermédiaire de la primitive CALL, mais plutôt par des branchements d'un objet à l'autre. Dans ce cas, la protection n'est plus assurée.

La solution statique avec section de liaison a été introduite pour rendre cette nature évolutive, d'où la tendance naturelle vers la solution des espaces d'exécution évolutifs.

c. Relations avec l'espace des objets

Un des gros inconvénients de l'espace évolutif est la difficulté à définir les références inter-segments, car seules les références par l'intermédiaire de noms locaux sont possibles. Nous verrons au paragraphe suivant (§ 3) que ce problème peut être résolu simplement par l'intégration d'un espace d'objets (nomenclature) à l'espace d'exécution.

d. Descripteurs regroupés ou répartis dans les objets

Les systèmes à objets et noms universels utilisent tous des mécanismes de descripteurs pour adresser les objets. On a alors le choix entre deux grandes classes de solutions : les descripteurs d'une part, sont regroupés dans des objets spécifiques comme des annuaires (cas de GEMAU, MULTICS, HYDRA, Ferrié), ou d'autre part, peuvent être inclus dans des objets de n'importe quel type, comme par exemple des segments de programme ou de données (cas de Fabry).

Nous allons essayer de comparer ces différentes solutions et déterminer leurs avantages et inconvénients respectifs.

. Références intersegments

La solution proposée par Fabry permet de définir des références "absolues" intersegments de données et l'on peut ainsi désigner n'importe quel objet de l'espace des objets quelle que soit la structure. Dans le cas de GEMAU ou de MULTICS, comme les descripteurs sont regroupés en des objets particuliers, les références intersegments sont faites par l'intermédiaire de l'espace des objets en utilisant des noms partiels. Dans le cas de MULTICS on peut désigner n'importe quel objet à condition d'en donner le nom partiel depuis la racine. Dans GEMAU, comme un processus ne peut généralement désigner la racine, cette solution n'est pas possible, mais on utilise alors pour cela les iens ; dans ce cas, les références dépendront toujours de l'environnement d'exécution et ne sont donc que des références relatives et non absolues. Cependant, les avantages de Fabry peuvent, si cela est nécessaire, être retrouvés en regroupant les références (descripteurs) d'un segment dans un objet associé, c'est le cas des squelettes d'espace d'exécution (cf. § 3.5.).

. Protection

Une solution du type de celle proposée par Fabry présente cependant un inconvénient majeur qui est l'impossibilité de transférer vers l'extérieur du système des segments où données et descripteurs sont mélangés. Le problème se pose en fait lors du transfert retour (i.e. restauration) de ces segments. En effet, s'il y a des références absolues (i.e. toutes les adresses ne sont pas relatives au contexte d'exécution), il n'est pas possible de transférer ces dernières car la falsification des descripteurs est toujours possible dans le monde extérieur. Dans le cas où les segments sont regroupés dans des objets particuliers, ce problème disparaît car il n'est alors pas possible de transférer ces objets vers, ou depuis, le monde extérieur : ils sont créés, maintenus et détruits par le noyau.

2.8. Utilisation de courts-circuits : les registres de base

Nous nous limitons au cas des objets de type segment.

Pour le processeur l'adresse d'une cellule de mémoire est définie par <ln >,< d > où ln est le nom local d'un segment et d un déplacement à l'intérieur du segment.

Dans une réalisation quelconque, cette adresse (< ln >, < d >) peut être grande car 24 et 32 chiffres binaires sont déjà des tailles normales pour une adresse dans les matériels existants. D'où, du point de vue du matériel, le besoin de limiter la taille de cette adresse dans une instruction.

De plus, si on considère une procédure en exécution à un instant donné, les données à accéder sont très localisées ; c'est-à-dire que la probabilité d'accéder une cellule proche de la dernière cellule accédée est plus grande que celle d'accéder une cellule lointaine. Transposée au niveau des segments, cette loi risque d'être encore plus vérifiée, c'est-à-dire que le nombre de segments nécessaires à un instant donné est faible.

Dans un système tel que GEMAU, l'espace d'exécution fournit l'ensemble des objets directement accessibles par le matériel. Cependant, même dans ce cas, la taille < ln >, < d > peut encore être très grande. Une solution pour réduire cette taille existe avec la définition de fenêtre sur l'espace d'exécution. Cette fenêtre permet de nommer seulement un sous-ensemble de cet espace ; les registres de base, comme dans l'IBM 360, sont souvent utilisés pour cela.

Une adresse directe, pour le processeur, est maintenant :

< rb >,< d1 >

au lieu de

< ln >,< d >

où

(nombre de< rb >) < (nombre de <ln >)

(taille de <d1 >) < (taille de <d >)

Il existe donc des instructions permettant de charger une adresse (< ln >,< d> dans un registre de base. A un instant donné l'espace *directement* adressable par le processeur est inférieur à la taille suivante :

(nombre de < rb >) x $2^{\text{taille de} < d1 >}$

La figure 24 donne un exemple d'espace directement adressable ; les différentes façons d'accèder les zones hachurées (non accessibles) sont les suivantes :

- par changement du contenu d'un registre de base de façon à avoir la zone à accéder dans l'espace direct du registre de base,
- par adressage indexé,
- par adressage indirect.

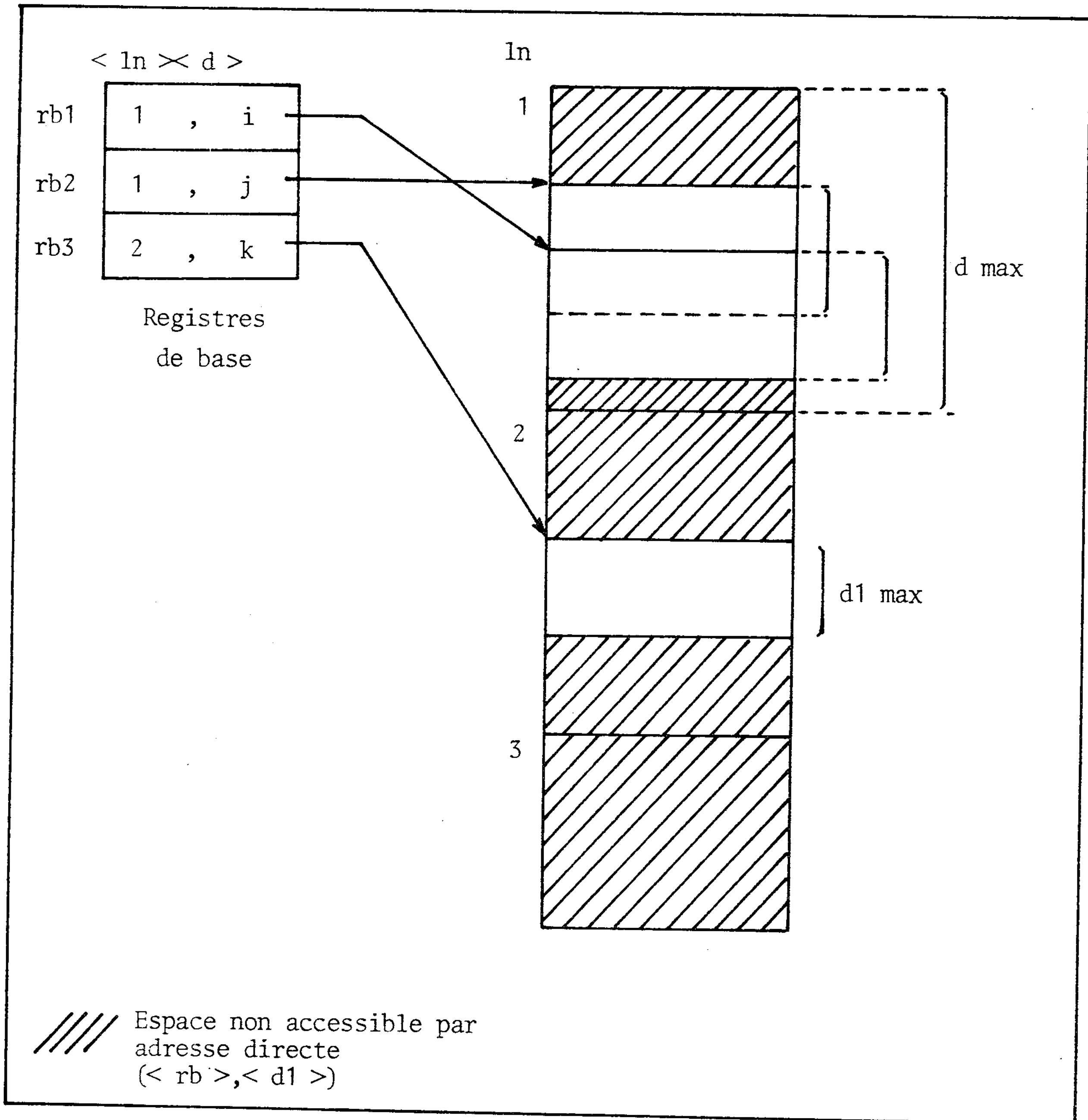


Figure 24 : Registres de base

Par exemple, dans le prototype GEMAU sur 10070, on a :

. taille maximale d'un segment = 16 k mots

. nombre maximal d'objets dans un espace d'exécution (LN) = 8

Dans ce cas, < rb > = < ln>. Ceci est extrêmement limitatif, spécialement du fait que l'adresse virtuelle maximale d'une unité centrale est de 128 k (8 x 16 k) et qu'il n'y a aucun registre de base dans le matériel.

Pour la réalisation sur IRIS 80, on a :

. taille maximale d'un segment = 128 k mots

. nombre maximal d'objets dans un espace d'exécution (nombre de LN = 128).

Ainsi, < ln >,< d > a les tailles suivantes :

< ln > : 7 } < ln >,< d > : 24 chiffres binaires
< d > : 17 }

Il y a sept registres de base et un déplacement < d1 > de 16 k mots seulement. Ainsi

< rb >,< d1 > → 3 + 14 = 17 chiffres binaires.

Une adresse (0,i) dans une instruction est le i^{ème} mot du segment de nom local 0.

Dans cette réalisation, l'espace d'exécution complet est de 128 x 128 k = 16 384 k mots, et la fenêtre est de 8 x 16 k = 128 k (i.e. 1/128 de l'espace d'exécution maximal).

Il existe une instruction pour charger une adresse (< ln >,< d >) de 24 chiffres binaires. De plus, l'IRIS 80 possède l'indexation et l'indirection.

Remarque :

Ce type de solution présente toutefois l'inconvénient de nécessiter une gestion "manuelle" des registres de base, alors que l'un des objectifs était de laisser le noyau gérer les noms locaux.

3. LIAISON ESPACE D'EXECUTION - ESPACE D'OBJETS

3.1. Définitions

Les aspects et propriétés des espaces d'exécution que nous venons d'examiner permettent de répondre aux deux questions suivantes :

. comment une procédure désigne-t-elle les contenus des objets qu'elle utilise ?

. comment peut-on établir des références inter-segments ?

Nous avons vu que sauf dans un cas particulier (descripteur préfixé), nous ne pouvions répondre à la seconde question.

De plus, un problème n'a jamais été abordé lors du paragraphe précédent : comment associe-t-on un nom universel (ou un objet) à un nom local ? C'est-à-dire quand lie-t-on un objet à un espace d'exécution ? Dans le cas de Fabry, la réponse est immédiate : des instructions permettent de charger des descripteurs depuis des cellules d'un segment, cependant il faut bien avoir fabriqué auparavant ces descripteurs, d'où le besoin d'un espace d'objets même dans ce cas simple.

Si on considère le Plessey PP250 [HA 72] [WI 72] qui fonctionne selon les principes énoncés par Fabry, les concepteurs ont construit un système de fichiers [EN 72] [GO 72] qui permet d'établir la correspondance entre un espace d'exécution (qui est évolutif) et l'espace des objets.

On peut faire dès maintenant une remarque générale : tous les systèmes conçus uniquement autour d'un espace d'exécution ont demandé la création d'un espace d'objets surimposé à l'espace d'exécution. Pour exemple on peut citer :

. MULTICS [GR 68], [OR 72]

. PP250 [EN 72], [GO 72]

A l'inverse, un certain nombre d'autres études ont mélangé les deux espaces, ce sont :

. SAR [VE 73]

. HYDRA [WU 74]

. Vanderbilt [VA 69]

. GEMAU

Il existe un certain nombre d'objectifs qui ne sont pas complètement résolus par un espace d'exécution seul, ce sont :

a. conservation d'information pour une durée supérieure à la durée de vie des espaces d'exécution. Cette conservation se double du besoin de moyens de désigner,"symboliquement" si possible, les objets.

b. Partage contrôlé des procédures, données et autres objets. Ceci rejoint un des objectifs des espaces d'exécution, mais une différence essentielle existe au niveau de la protection en ce sens que la protection associée à un objet est une protection potentielle alors que celle associée à l'accès (durant l'exécution) est réelle et peut être différente de la protection potentielle. Pour plus de détails, on consultera le chapitre suivant.

c. Création, destruction, croissance et réduction des objets pendant l'évolution d'un processus.

d. Modularité : c'est la possibilité pour des utilisateurs différents d'utiliser le travail des autres sans connaissance de la réalisation interne de ce travail (procédures, données, autres objets). Cet objectif existe pour les espaces d'exécution et en fait tout le paragraphe 2 présente des solutions qui répondent à cet objectif.

Le dernier aspect qui peut être traité par intégration des deux espaces, est celui lié à la notion de contexte et de changement de contexte. En effet, il n'y a pas de raison pour que les contraintes de visibilité des espaces d'exécution ne s'accompagnent de contraintes similaires pour les espaces d'objets : aspect évolutif des espaces d'exécution, à chacun desquels correspond un environnement.

Ce qui est très important, c'est que la structure de l'espace d'objets et les règles de désignation dans cet espace suivent les mêmes structure et règles que celles de l'espace d'exécution, sinon il y a rupture de l'homogénéité du mécanisme et demande d'introduction de mécanismes supplémentaires.

C'est par exemple le cas de MULTICS où la structure de l'espace d'objets est indépendante de la structure de l'espace d'exécution, d'où un certain nombre de problèmes, en particulier pour la protection (cf. chapitre suivant).

Ceci est extrêmement simple avec GEMAU, dans la mesure où tout objet n'est adressable que par l'intermédiaire d'un nom local, éventuellement suivi d'un nom partiel (on rappelle que l'environnement courant est défini par le nom local LNO).

3.2. Moment de liaison

Nous avons vu dans les exemples précédents comment on pouvait associer un nom local à un objet lors de la compilation. La question qui se pose maintenant est de savoir quand charger ce nom local avec le nom unique de l'objet (ou "capability"), c'est-à-dire quand effectuer la liaison d'un objet à un espace d'exécution.

Les points qui nous concernent ici sont les étapes 3, 4 et 6 de la figure 1. Eliminons momentanément l'étape 3 (compilation) pour ne considérer que les étapes 4 et 6. Si nous voulons permettre un degré de généralité suffisant dans la programmation, il faut permettre à une procédure d'adresser les objets manipulables de façon symbolique et de même, pour toute nouvelle procédure appelée, ignorer quels sont les objets manipulés par cette procédure. Nous éliminons la liaison à l'étape 4 qui peut toujours être réalisée de façon classique (édition de liens sous OS 360 ou SIRIS 7/8 par exemple) et qui est toujours possible à l'intérieur d'un même segment pour regrouper des éléments compilés indépendamment. Ce regroupement s'effectue par recopie dans un segment des différents éléments.

Presser et White [PR 72] décrivent en détail les mécanismes nécessaires à ce type d'édition de liens qui existe aussi dans MULTICS pour des raisons économiques (problème de performances, cf. Organick [OR 72] pp. 57-58) à cause de la grandeur de la taille des segments qui sont obligatoirement paginés.

Le point 6 est beaucoup plus intéressant car il faut savoir comment la correspondance entre un nom (partiel) d'objet et un nom local est établie. Il y a deux solutions possibles : soit lors de la première référence à l'objet qui se fait toujours de façon symbolique (cas de MULTICS), soit explicitement lors de l'évaluation des paramètres pour un appel ou lors d'une commande explicite (cas de GEMAU). Ces deux solutions qui sont classiques dans les langages, correspondent au moment d'évaluation des paramètres (cf. Fraser [FR 71]). Nous examinerons le cas de la liaison à l'exécution successivement pour MULTICS (§ 3.3.) et pour GEMAU (§ 3.4.). Puis nous étudierons ensuite le problème de la liaison avant l'exécution.

Considérons la solution qui consiste à préfixer l'information contenue dans les segments ; cette solution offre une plage très étendue de possibilités de choix du moment de liaison. Spécialement pour des choix effectués lors de la compilation, où il est alors facile d'associer un objet particulier à une référence dans une procédure (cf. § 2.4. et figure 13). Dans le cas de MULTICS ou GEMAU, cette solution semble difficile.

Eliminons d'office la possibilité de liaison type "édition de liens classique" qui est de toute façon postérieure à la compilation et qui ne concerne pas les références inter-segments. Dans le cas de GEMAU, il y a deux possibilités que nous examinerons successivement. La première, qui n'a pas été réalisée dans la définition actuelle et qui peut être considérée comme une extension possible, est inspirée de Evans et Leclerc, et HYDRA (§ 3.5.). La seconde prend en compte les particularités de l'intégration espace d'objets - espace d'exécution (§ 3.6.).

3.3. Liaison implicite à la première référence lors de l'exécution (MULTICS)

La solution que nous présentons ici est fonctionnelle, la réalisation en étant plus complexe dans le détail dans le cas de MULTICS. L'idée de base est très simple : la première fois qu'une procédure accède à un objet par un nom symbolique, la référence (ou l'appel) est déroutée vers le noyau qui remplace (dynamiquement) le nom symbolique par un nom local (appelé dans MULTICS, un numéro de segment). Dans ce cas, il faut qu'un descripteur d'objet contienne soit le nom universel de l'objet, soit un nom symbolique, c'est-à-dire que dans l'entrée (correspondant à un index) de la section de liaison contienne les informations données par la figure 25, où nous n'indiquons que les informations qui sont nécessaires à la liaison.

L'indicateur indique, si la valeur est L, que l'objet possède le nom local indiqué et si sa valeur est S qu'il n'y a pas encore eu accès à l'objet. Lors de la résolution d'une adresse du type (< index >,< déplacement >), si la valeur de l'indicateur est L alors tout se passe exactement comme dans les exemples de l'espace d'exécution statique avec section de liaison (cf. § 2.3.). Si l'indicateur est S alors le matériel déclenche un déroutement et abandonne l'opération que l'on tentait d'effectuer.

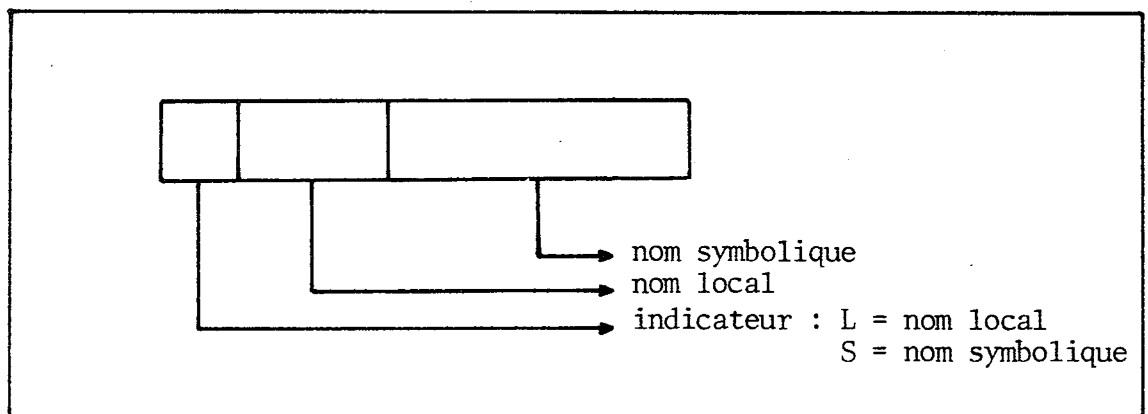


Figure 25 : Informations de liaison dans MULTICS

Le déroutement provoque un appel du noyau qui tente d'effectuer la liaison de l'objet représenté par le nom symbolique avec l'objet lui-même (nom universel) ; en d'autres termes, il s'agit d'affecter un nom local à l'objet et de le placer dans l'index, puis de mettre l'indicateur sur L et relancer l'opération qui avait provoqué le déroutement. Durant l'évaluation du nom symbolique, il faut déterminer si l'objet n'est pas déjà lié au processus, auquel cas il suffit de mettre son nom local dans l'index, sinon il faut affecter un nouveau nom local à l'objet et l'on se trouve ramené au cas d'un objet déjà lié. Pour cela, il faut donc posséder une table par processus qui indique quels sont les objets déjà liés à ce processus. Cette table comporte autant d'entrées que la table des segments du processus. Les figures 26 et 27 donnent, dans le temps, les diverses transformations qui s'effectuent pour le cas de appeler E (cette figure reprend l'exemple de la figure 5).

La première exécution de appeler(3,0)provoque un déroutement (figure 26) qui se traduit par l'évaluation du nom symbolique E, l'introduction de l'objet E dans l'espace d'exécution (par exemple sous le nom local LN5).

Toute nouvelle exécution de appeler(3,0) dans la procédure A effectuera directement l'appel à E sans plus jamais passer par le mécanisme de liaison, car l'indicateur est positionné sur L dans la section de liaison.

Cette méthode présente un inconvénient majeur qui est l'encrassement de la table des segments par des objets liés. Cet encrassement, qui est dû à l'existence conjointe d'adresse du type (< index >,< déplacement >) et (< nom local >,< déplacement >) fait qu'il y a dans la table des segments des objets non utilisés. De plus, le nombre maximal de noms locaux est forcément limité, ne serait-ce que pour pouvoir coder facilement les instructions.

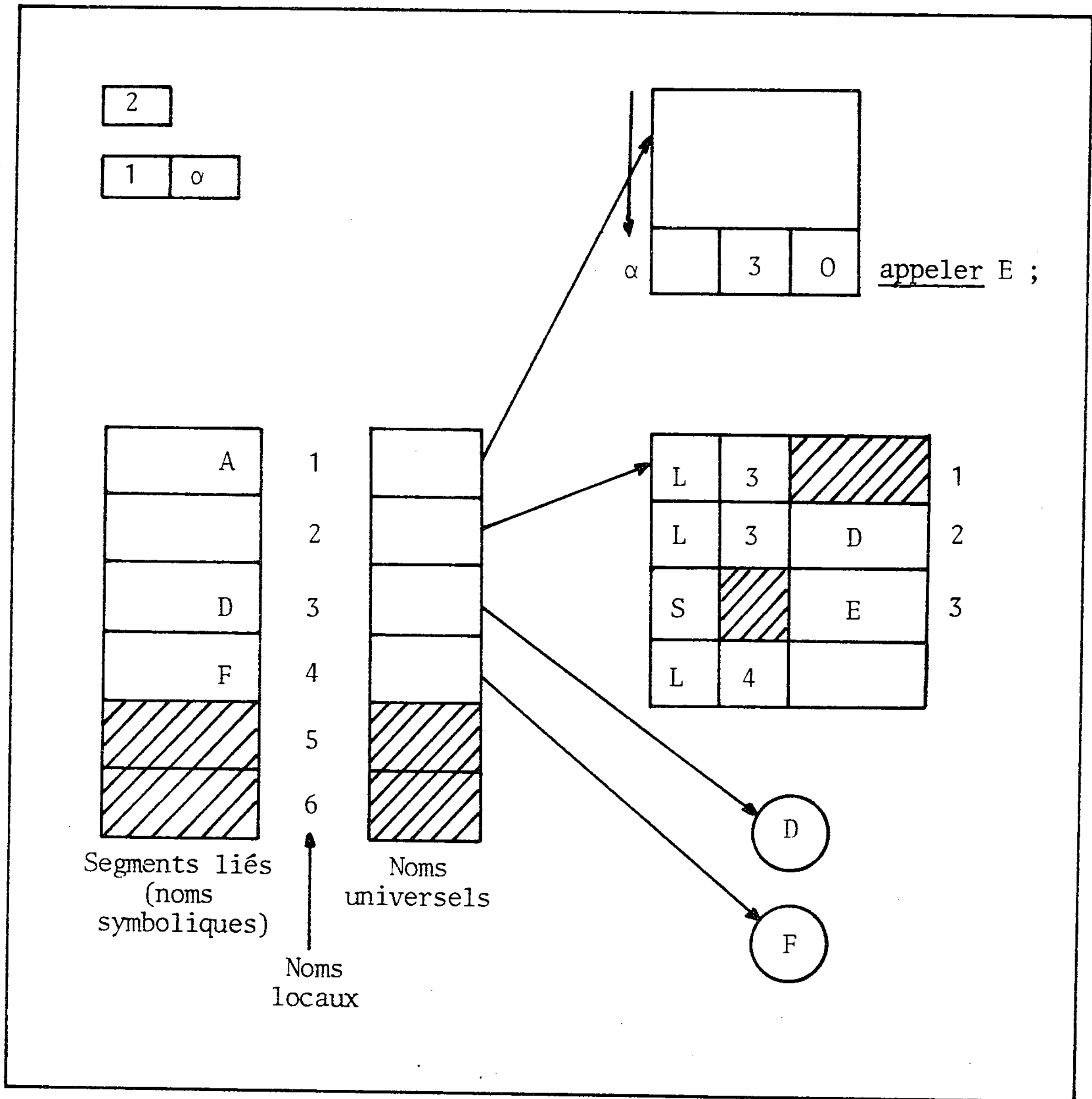


Figure 26 : Section de liaison avec référence non résolue

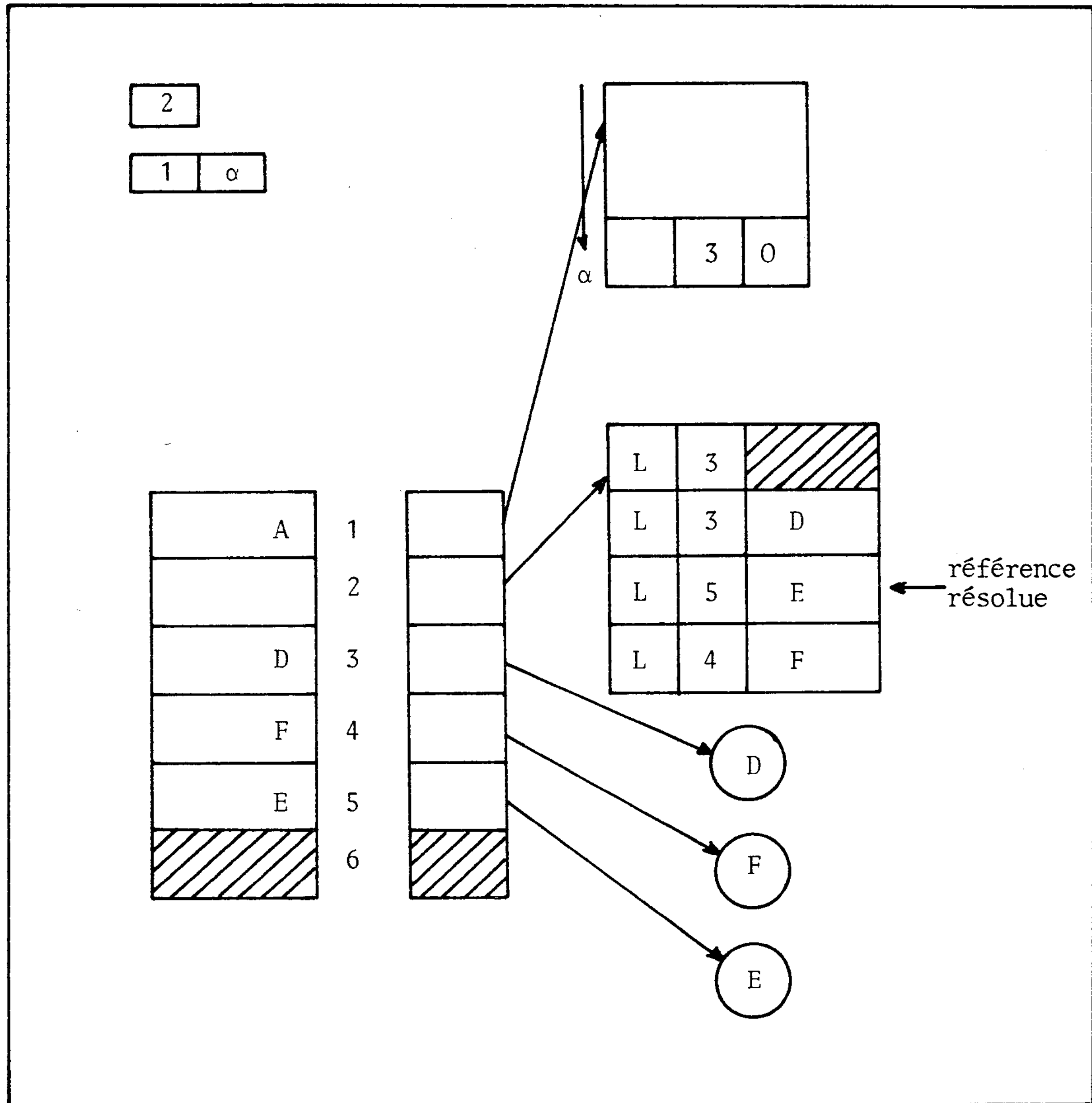


Figure 27 : Section de liaison après résolution de référence

Si la liaison est automatique, il semble difficile de rendre l'opération inverse ("déliaison") automatique, sauf éventuellement pour certains cas tels que la destruction d'objets locaux à une procédure, et encore à condition que l'on ait choisi le mécanisme de section de liaison récursif, c'est-à-dire avec activation d'une nouvelle section à chaque appel de procédure.

Le cas non récursif, choisi par MULTICS, fait que la section de liaison associée au processus l'est à la première activation de la procédure, et les références résolues de la section le restent pour la durée de vie du processus. La "déliaison" impose la connaissance du moment où un objet n'est plus utilisé, ce qui peut être fait en conservant au niveau de la table des segments un compteur de liaison de l'objet au processus (nombre d'index le référençant). Il suffit d'incrémenter ce compteur chaque fois que le mécanisme de liaison implicite est en oeuvre et de le décrémenter chaque fois que l'on supprime un index, c'est-à-dire une section de liaison. En fait, ceci est difficilement réalisable dans le cas de MULTICS, d'une part car il n'y a qu'une section de liaison créée lors de la première activation et d'autre part, parce que l'on peut utiliser des adresses du type < nom local >,< déplacement > n'importe où dans les segments et qu'il faudrait retrouver toutes ces adresses. De plus, la réallocation de noms locaux est contraire aux objectifs de MULTICS.

3.4. Liaison explicite à l'exécution (GEMAU)

Dans le cas de GEMAU, ce problème d'encrassement n'existe plus, car la limitation sur le nombre maximal de noms locaux n'est valable que pour un espace d'exécution (i.e. l'exécution d'une procédure) et à chaque changement d'espace d'exécution on a toujours le même nombre maximal de noms locaux disponibles.

L'encrassement dû au maintien d'objets inutiles disparaît de lui-même grâce à la structure en pile des espaces d'exécution. Par contre, il faut, comme dans MULTICS, maintenir une table des objets liés au processus, ainsi qu'un compteur de liaison pour chacun des objets pour résoudre le problème, d'une part de la synchronisation par degré de multiprogrammation et, d'autre part de la destruction physique des objets.

De plus, la connaissance explicite de la liaison et de la "déliaison" d'un objet dans un espace d'exécution favorise la propriété de substitution d'objets. En effet, dans le cas de MULTICS, un objet est généralement lié à un processus pour toute la durée de vie de ce processus. Toute substitution de l'objet par un autre équivalent ne peut être prise en compte par ce processus.

Considérons les deux programmes donnés par la figure 28.

```
π1 : .
     :
     répéter n fois
     début
         :
         appeler(X) ;
         :
     fin ;
     :

π2 : .
     :
     LNi ← lier(X) ;
           :
           répéter n fois
           début
               :
               appeler(LNi) ;
               :
           fin ;
           :
```

Figure 28

Dans le cas de MULTICS $\pi1$ est équivalent à $\pi2$, car la première référence à l'objet X lie cet objet à l'espace d'exécution du processus. Dans le cas de GEMAU, il est possible de choisir l'une des deux solutions : pour $\pi1$ c'est l'objet de nom X qui est exécuté à chaque itération dans la boucle, et si entre deux itérations l'objet est remplacé par un autre objet équivalent de même nom X, alors le nouvel objet X sera exécuté au prochain appel. Pour le programme $\pi2$, l'objet X est lié à l'espace d'exécution du processus et, même en cas de substitution de X, c'est toujours l'objet initial qui sera référencé dans la boucle.

3.5. Liaison avant l'exécution : squelette d'espace d'exécution

Il est intéressant de considérer de près l'article de Evans et Leclerc [EV 67]. A une procédure définie par un ensemble d'instructions et un ensemble de paramètres formels, on peut associer un espace paramètre (selon la terminologie de Evans et Leclerc) qui contienne la description de chacun des paramètres formels et des objets globaux, comme pour la définition statique de la section de liaison associée à une procédure. Certains des éléments de cette section peuvent être liés dès la compilation. Dans le cas de GEMAU ou de MULTICS, cette section statique servirait de valeur initiale à une section dynamique lors de l'activation de la procédure, les paramètres actuels étant résolus soit selon la méthode MULTICS (liaison au premier accès), soit selon la méthode GEMAU (liaison au moment de l'appel de procédure). Cette solution est représentée par les figures 29 et 30. La figure 29 donne le squelette d'un espace d'exécution associé à la procédure A (de l'exemple de la figure 5), où les objets D et E ont des références résolues lors de la compilation de la procédure A.

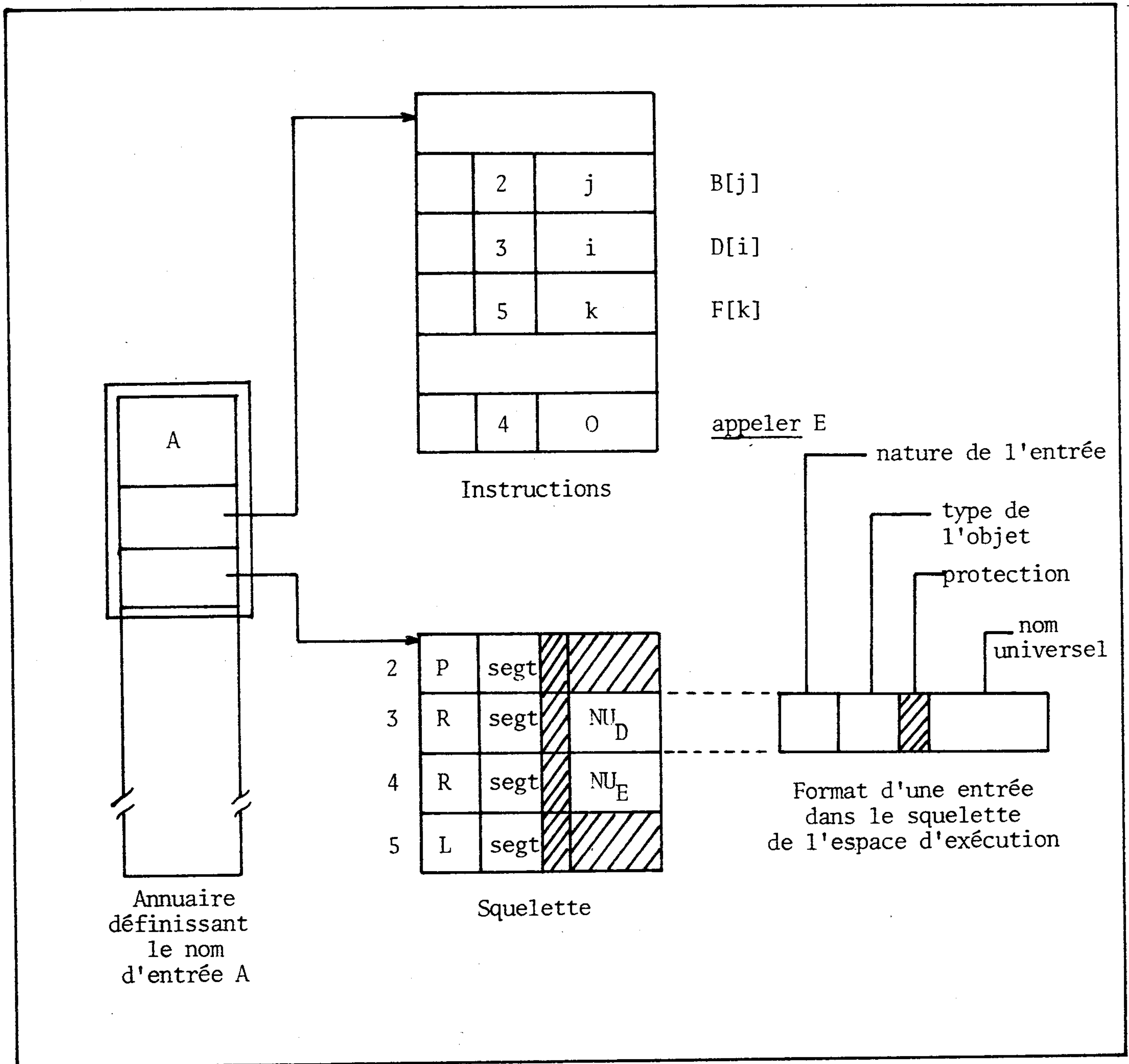


Figure 29 : Squelette d'espace d'exécution associé à A

La nature d'une entrée dans le squelette peut avoir l'une des trois valeurs suivantes : P, R ou L.

P = paramètre formel, dans ce cas il n'y a pas de nom universel associé,

R = entrée résolue, c'est-à-dire que l'objet de nom universel indiqué est lié à la procédure,

L = objet local, c'est-à-dire qui doit être créé à chaque activation de la procédure.

La figure 29 représente la solution dans le cas de GEMAU pour la procédure A, la figure 30 représente le résultat de appeler(A,D) avant le début effectif de l'exécution de A.

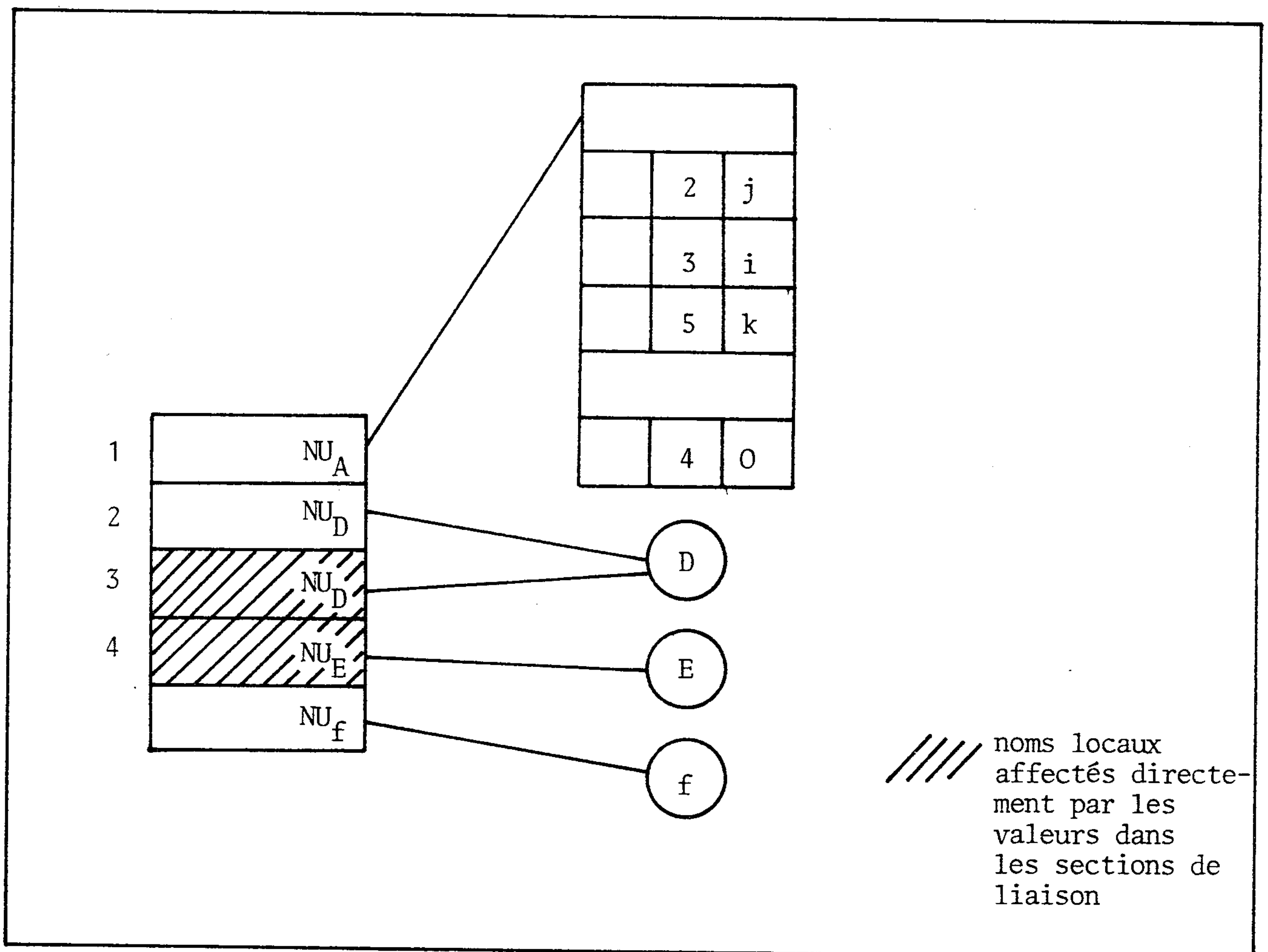


Figure 30 : Espace d'exécution initial

Cette solution n'a pas été réalisée dans GEMAU, c'est une extension possible. Elle permet d'effectuer une vérification sur le type des paramètres réels, vérification qui doit être effectuée si on le désire, par la procédure ellè-même à l'heure actuelle. Un des problèmes essentiels de cette vérification dynamique réside dans la perte de la possibilité de substitution d'un objet par un objet ayant la même fonction mais réalisé différemment (par exemple la virtualisation des périphériques), ce qui motive en partie sa non-incorporation dans GEMAU. Cependant, on pourrait rajouter un type particulier, dans le squelette, dont la signification serait : "n'importe quoi" et seuls les paramètres dont le type serait explicitement spécifié ne pourraient être "virtualisés".

Cette solution, qui est réalisée dans HYDRA est aussi adoptée par Ferrié [FE 74]. Dans le cas de HYDRA, une procédure contient une description des paramètres formels et un ensemble de descripteurs d'objets qui sont liés à la procédure. Ces deux descriptions sont utilisées comme un squelette pour la fabrication du LNS ("local name space") lors de l'activation de la procédure. La première description sert à la vérification des types et la seconde à la définition de l'espace d'exécution initial, y compris la protection réelle (cf. chapitre sur la protection § 4.2.).

3.6. Liaison avant l'exécution : insertion dans l'espace des objets

Si nous reprenons l'exemple de la figure 1, nous voyons qu'il y a une liaison à la compilation (étape 4), dans le cas de GEMAU cette liaison consiste non seulement à changer les identificateurs symboliques du programme source, mais aussi à insérer la procédure une fois compilée dans l'espace des objets, ce qui permet de résoudre les variables libres (externes par exemple) à l'exécution en fonction de l'endroit où cette procédure est définie dans l'espace des objets.

En considérant l'exemple de la figure 5, nous allons étudier les différentes possibilités.

a. Liaison effectuée par le compilateur

Son rôle est de transformer les noms symboliques des paramètres et des objets locaux en des noms locaux. Par exemple, dans notre cas, faire correspondre LN1, LN2 et LN3 respectivement à A, B et F. Par contre, les objets D et E qui sont externes ne sont pas connus à l'instant de la compilation. La compilation de appeler(E) se transforme en une primitive CALL(E), mais la compilation de D[i] impose d'avoir, dans une instruction, une adresse de type (< nom local >,< déplacement >) ; il faut donc que le compilateur rajoute un appel à la primitive BIND pour l'objet D avec le nom local 4.

Il faut remarquer que tout ceci n'est qu'une génération de code, et qu'aucune des primitives dont nous venons de parler n'est effectuée. La figure 31 donne l'exemple d'une compilation possible. Dans cette figure le code opération 'SVC' est un appel au noyau, avec comme adresse celle d'une zone contenant les paramètres de la primitive à exécuter.

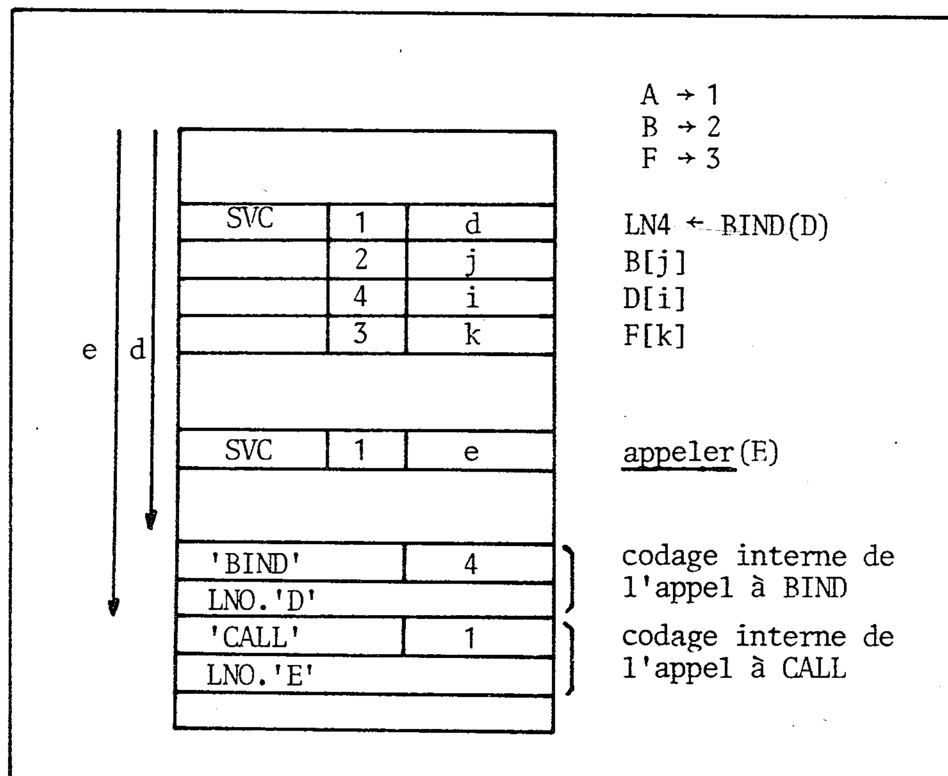


Figure 31 : liaison à la compilation

b. Insertion dans l'espace des objets

Cette procédure, une fois compilée, doit être, éventuellement, mise dans l'espace des objets. Si cette procédure est simplement un objet temporaire, on rappelle que son environnement d'exécution est le même que l'environnement d'appel* ; dans ce cas, les objets D et E de l'environnement d'appel sont utilisés. L'insertion dans l'espace des objets peut être effectuée de deux manières différentes :

(i) la procédure est définie dans un environnement, auquel cas cet environnement sera celui utilisé lors de l'exécution de la procédure. Les objets D et E seront ceux définis dans cet environnement ;

(ii) la procédure est définie dans un annuaire, auquel cas l'environnement d'exécution sera celui de la procédure appelante (de A) et les objets D et E seront ceux de l'environnement d'appel.

Lors de l'exécution de la procédure de nom partiel A les objets connus sous les noms partiels D et E sont uniques dans le cas de la figure 32 (cas i). Par contre la figure 33 (cas ii) montre que la même procédure (15) connue sous les noms d'entrée A dans les environnements 1, 2 et 3, va s'exécuter dans l'un de ces environnements quand elle est appelée depuis l'un d'eux (principe du mécanisme de porte). Dans ce cas, on peut dire que l'on a autant de sections de liaison que d'environnements d'appel différents. Ainsi :

. appel depuis l'environnement 1 : D → 5, E → 6
. appel depuis l'environnement 2 : D → 11, E → 9
. appel depuis l'environnement 3 : D → 11, E → 13

* Conséquence de la notion de "porte".

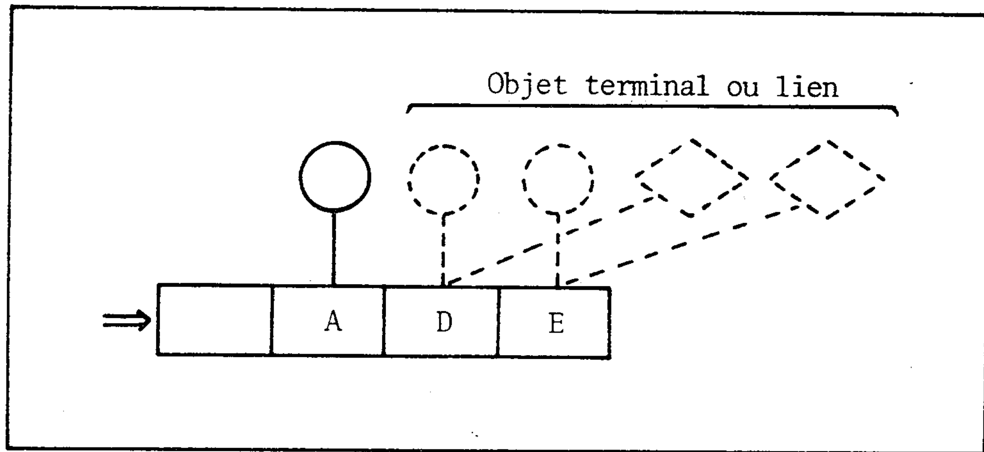


Figure 32 : Une seule section de liaison

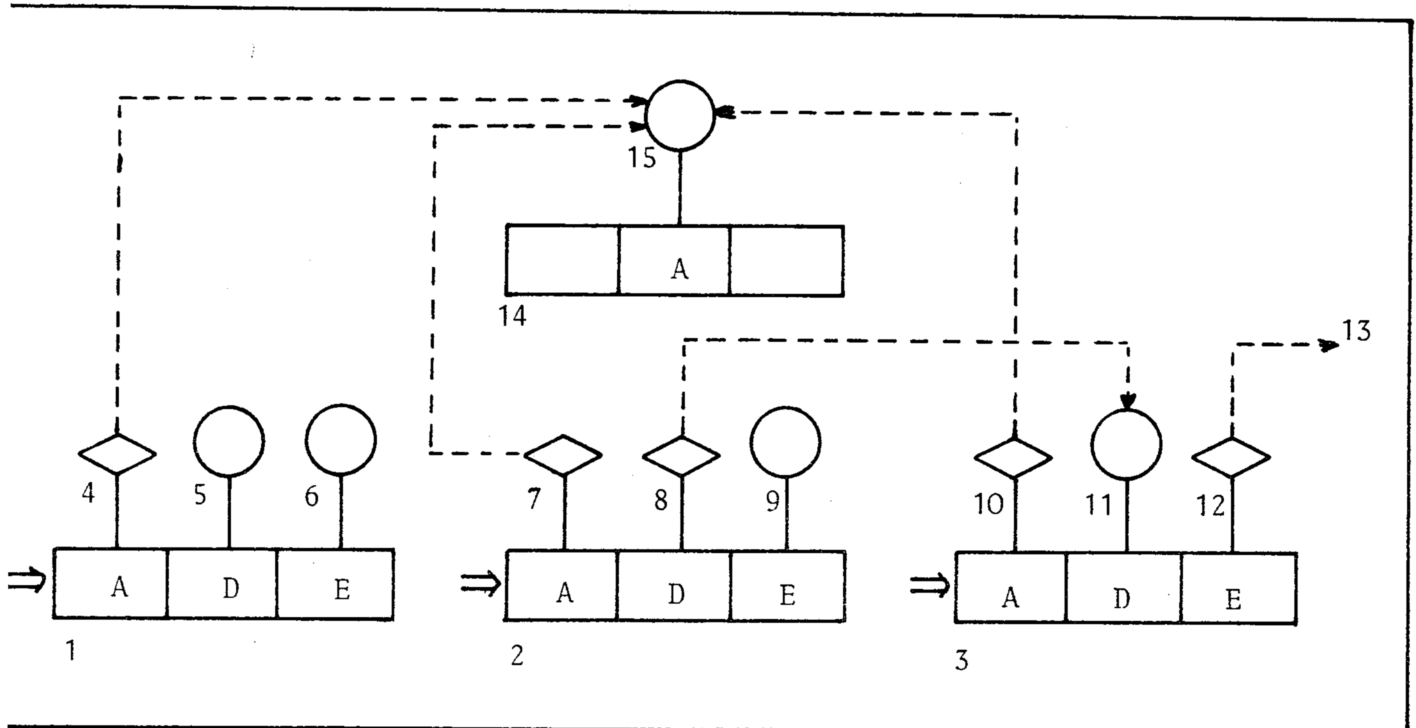


Figure 33 : Plusieurs sections de liaison

3.7. Mécanismes et politiques

Si nous considérons les deux approches différentes sur le problème de la liaison (implicite dans MULTICS, explicite dans GEMAU) on peut faire plusieurs remarques :

. la liaison implicite à la première référence est un choix qui est imposé à l'utilisateur et il n'y a aucun moyen simple pour effectuer la déliaison ; ce choix est déjà une politique.

. Le problème de la déliaison dans le cas de la liaison implicite peut être moins crucial si la structure de l'espace d'exécution est évolutive. En effet, on connaît dans ce cas un instant obligatoire de déliaison, c'est le retour de procédure. Le problème de l'encrassement des espaces d'exécution n'est plus alors aussi critique.

. L'utilisation de la liaison explicite permet de réaliser une politique de liaison à la première référence (en utilisant les mécanismes de déroutement), alors que la réalisation de la liaison explicite est impossible si on dispose seulement du mécanisme évolué de la liaison implicite.

Chapitre 5

PROTECTION

RELATIONS ADRESSAGE - PROTECTION

Nous étudions le problème de la protection des objets dans un système, en partant du point de vue de l'utilisateur pour aboutir à la notion de domaine de protection, à l'aspect dynamique de la protection et de l'instant de protection.

Ensuite, nous décrivons rapidement un modèle théorique classique, avant d'étudier de façon très précise les relations entre la protection et les structures d'adressage vues au chapitre précédent.

Certains problèmes particuliers sont analysés de façon détaillée : méfiance réciproque, résiliation de droits d'accès, liens non résolus.

1. LE POINT DE VUE DE L'UTILISATEUR

1.1. Introduction.

La protection de l'information et de tous types d'objets résulte de la volonté d'un utilisateur de limiter et de contrôler l'accès aux objets qu'il conserve dans un système. Cette protection va depuis l'indépendance complète vis à vis d'autres utilisateurs jusqu'au partage de l'information avec ces mêmes utilisateurs. Dans le premier cas personne, à l'exception du propriétaire, ne peut lire ou modifier l'information qui est considérée comme privée. Dans le second cas, toute une gamme d'accès peut être désirée, en effet, pour chacun de ses objets un utilisateur peut vouloir indiquer en particulier si celui-ci :

- est privé
- est public
- est public mais ne peut être détruit que par le propriétaire
- n'est utilisable que par certaines classes d'utilisateurs, et pour chacune de ces classes seuls certains types d'accès sont autorisés ; par exemple l'objet peut être recopié, éxécuté, lu, modifié, etc...
- ne peut être utilisé que si celui qui l'accède fournit un mot de passe correct.
- ne peut être accédé qu'après validation de certaines conditions propres à l'utilisateur,et auxquelles le système n'aurait pas pensé. Par exemple tel objet ne peut être accédé que par telle personne tous les jours ouvrables entre 8 et 18 heures, à condition toutefois de fournir un mot de passe changeant chaque jour.

Ces divers exemples montrent que la protection désirée par un utilisateur peut être très variable. Il importe donc de définir les différentes classes de protection que désire un utilisateur, ce sont :

a) la protection contre les erreurs d'autres utilisateurs (ou processus) : indépendance des exécutions.

b) la protection contre lui-même.

c) l'accès à ses propres objets, par d'autres utilisateurs, d'une façon controtrolée par lui même.

d) la méfiance réciproque.

e) la protection contre les pannes du système.

a) Indépendance entre processus.

C'est le fait que l'exécution d'un processus utilisateur puisse se dérouler sans que des erreurs de fonctionnement, ou des tentatives malhonnêtes, dans d'autres processus aient un effet quelconque pour le processus de l'utilisateur s'il ne partage rien explicitement avec les autres processus.

b) Protection contre soi-même.

La protection , c'est aussi une série de mécanismes qui doivent pouvoir s'appliquer à l'utilisateur lui-même. Par exemple, pour protéger des programmes et données correctes pendant la mise au point de nouveaux programmes ;

c) Accès partagé.

C'est la possibilité de permettre l'utilisation de certains objets par d'autres utilisateurs en indiquant au système quelles en sont les utilisations possibles. Cette protection se situe au niveau de l'espace des objets pour indiquer les possibilités d'accès et au niveau de l'espace d'exécution pour que le système prenne en compte ces possibilités. La permission d'utilisation donnée par un utilisateur appelle la possibilité, pour ce même utilisateur, de résilier cette permission. C'est ce que nous traiterons au paragraphe sur la réalisation des droits d'accès (§ 7).

d) Méfiance réciproque.

Un autre type de protection est celui de la méfiance mutuelle ; c'est le cas où un utilisateur (A) utilise un service fourni par le système ou par un autre utilisateur (B). Le système (ou l'autre utilisateur) veut se protéger contre des erreurs de l'utilisateur de l'objet (A), mais en contrepartie l'utilisateur (A) ne veut pas qu'une erreur dans le service rendu ait des conséquences néfastes sur les objets que lui même (A) possède. En d'autre termes, il faut que l'erreur dans le service ait des effets limités, c'est-à-dire qu'ils puissent être controlés par l'utilisateur du service.

e) Protection contre les pannes du système (fiabilité).

Cette protection contre les incidents du système, correspond à la nécessité d'assurer l'intégrité des informations conservées par le système. Cet aspect n'est pas traité dans le présent ouvrage.

1.2. Propriétés de la protection.

Le système va donc fournir un certain nombre de mécanismes pour permettre à l'utilisateur de réaliser ses différentes classes de protection (cf § 2.1.). Il faut, d'un point de vue de l'utilisateur, que ces mécanismes possèdent un certain nombre de propriétés qui sont :

- libre contrôle des mécanismes
- garantie d'utilisation
- transparence
- impossibilité à contrefaire

a) Libre contrôle des mécanismes.

Cette propriété part du principe qu'un utilisateur qui crée un objet (et qui le possède donc) doit pouvoir définir toutes les restrictions d'accès qu'il désire et aussi pouvoir résilier toute protection de son propre chef.

b) Garantie d'utilisation.

Il faut que les mécanismes permettent d'assurer l'utilisateur d'un service qu'une fois obtenu l'accès à ce service, ce dernier ne lui soit retiré de façon impromptue.A partir du moment où l'on a donné accès à un objet (en fait un service) a-t-on le droit de retirer cet accès ? Vanderbilt [VA 69] considère que le partage, c'est-à-dire la fourniture d'un service, est un contrat et que l'on ne peut rompre le contrat car les utilisateurs qui utilisent ce service construisent des applications qui dépendent de ce service. Le retrait de ce service est une rupture de contrat, et Vanderbilt indique que la résiliation d'un tel contrat n'est pas souhaitable. Lampson [LA 71] , Graham et Denning [GR 72] considèrent que cette résiliation doit être possible et est même désirée à cause de la méfiance réciproque qui existe entre les utilisateurs.

Quant à nous, nous pensons que les deux aspects sont justifiés et qu'il faut donc fournir les outils en conséquence ; par contre nous ne considérons pas le cas où les différents utilisateurs ne respectent pas leurs contrats, c'est-à-dire que la mauvaise utilisation des outils a des implications, éventuellement juridiques, hors du domaine technique traité ici.

c) Transparence des mécanismes.

Les mécanismes de protection ne doivent pas imposer de contraintes supplémentaires à la programmation pour l'utilisateur d'objets partagés. Une fois définit les modalités de partage et de protection, le système doit effectuer une vérification constante sans que l'utilisateur d'un service soit conscient de cette vérification.

d) Impossibilité à contrefaire.

La contrefaçon, c'est-à-dire la violation des mécanismes, doit être interdite de façon très stricte, ou à l'extrême limite l'utilisateur doit bénéficier de moyens propres à prendre des mesures exceptionnelles dans ces cas (cf. a). Ce point ne considère pas le cas où l'utilisateur a fourni certains accès droits à un utilisateur qui les utilise de façon "malhonnête mais autorisée" C'est un problème de coopération réciproque : si deux utilisateurs ont des droits équivalents sur un objet, rien ne permet au système de vérifier que ces droits sont utilisés "honnêtement".

2. LE POINT DE VUE DU SYSTEME.

2.1. Introduction.

Nous avons vu que ce qui est demandé par l'utilisateur couvre un domaine assez vaste et variable, il n'est donc pas possible de fournir toutes les classes possibles de protection. Ce qui nous intéresse c'est donc la définition d'un ensemble cohérent de mécanismes pour permettre de construire les classes désirées par l'utilisateur ; nous ne développerons ici qu'un aspect technique. Il s'agit essentiellement de contrôler comment les processus accèdent aux objets.

En effet, ainsi que nous l'avons vu dans le chapitre 1, il n'y a aucune différence dans les relations entre un utilisateur et le système, entre le système et les sous-systèmes, etc... Ce qui veut dire que les mécanismes de protection doivent être identiques à tous les niveaux (i.e. système, sous-système, utilisateur,...) Dans la construction des moyens de protection nécessaire à un utilisateur, il est très utile que le sous-système puisse s'appuyer sur les mêmes mécanismes que ceux sur lesquels s'appuyait le système pour assurer la protection entre les différents sous-systèmes. Le problème est similaire au niveau de l'utilisateur qui peut être lui-même composé de plusieurs processus.

Pour définir la protection, il importe d'abord de définir quelles sont les entités manipulées et leurs interactions. Les entités à protéger sont les *objets*, tandis que les entités qui les manipulent sont les *processus*. La protection sera définie par les règles qui permettent aux processus d'accéder aux objets. Ces règles forment le coeur de tout système de protection, elles doivent être :

- simples, c'est-à-dire faciles à comprendre et à manipuler de façon à être sûr qu'il n'y a aucun moyen de tromper la protection.

- fonctionnelles, c'est-à-dire être naturelles pour ne pas être une gêne pour l'utilisateur.

- complètes, c'est-à-dire ne pas permettre les accès non autorisés.

- générales pour permettre le partage contrôlé entre les utilisateurs (ou sous-systèmes).

Le point important qui a été très bien souligné dans HYDRA [WU 74], est que la protection est liée aux problèmes d'adressage. A ce sujet on trouvera un essai de classification des mécanismes de protection en fonction de la structure d'adressage dans Moore [MO 73]. Nous essaierons de montrer que le choix de cet adressage influence directement la protection car des erreurs de conception ou une conception incomplète oblige dans certains cas à surajouter des mécanismes supplémentaires.

La protection qui nous intéresse concerne l'objet dans son ensemble et non des parties internes de cet objet, ainsi nous ne considèrons pas les mécanismes classiques telles que protection des pages de mémoire centrale, ou des instructions que l'on peut exécuter (mode maître ou esclave). C'est la position unanime de toutes les études récentes ou actuelles sur la protection, on peut pour cela s'intéresser à Dennis et Van Horn [DE 66], Denning [DE 70], Fabry[FA 74], Graham et Denning [GR 72], England [EN 72], Wulf et al. [WU 74], Ferrié [FE 74], Schroeder et Saltzer [SC 72] , Vanderbilt [VA 69] Lampson [LA 71] pour ne citer que les études les plus importantes.

2.2. Liaison versus accès.

Pour effectuer une protection, il faut connaître à tout instant quels sont les pouvoirs que possède un processus ainsi que les possibilités d'établir et (ou) de modifier ces pouvoirs. A cet effet, il faut distinguer très précisément le problème de la visibilité de l'objet à protéger par un processus, et le problème de l'accès réel par le processus à cet objet. En effet, un objet situé dans un espace d'objets possède un type qui permet de connaître les opérations applicables sur l'objet. Ce type est une caractéristique *intrinsèque* de l'objet.

L'objet possède aussi un ou plusieurs noms, et à chacun de ces noms est associée une protection qui est une restriction sur les opérateurs applicables à l'objet. Cette protection est une protection *potentielle* : Un processus possède l'objet à un instant donné, dans son espace visible si cet objet possède un nom dans l'environnement courant du processus ; Cette visibilité offre donc une possibilité d'accès à l'objet, en permettant soit de le manipuler soit de le lier au processus pour pouvoir l'accéder directement. L'accès réel à l'objet (par exemple l'accès aux différents mots si c'est un segment) ne peut être effectué que sur des objets liés. Il y a donc deux temps dans un mécanisme de protection :

- la liaison dans l'espace d'exécution
- l'accès proprement dit aux objets.

La liaison d'un objet correspond à une *authentification*, c'est-à-dire à la vérification de la validité de cette liaison. Lors de la liason d'un objet on peut spécifier des restrictions supplémentaires sur les opérateurs de l'objet, par exemple un objet ayant tout accès autorisé dans l'espace des objets peut être lié en lecture seule dans l'espace d'exécution. C'est la restriction dans l'espace d'exécution qui est prise en compte par le matériel pour l'accès final au contenu de l'objet. La liaison permet de définir, pour un processus donné, un accès sélectif à un objet. Il est bien évident que le même objet peut être lié, au même processus ou à des processus différents, avec des protections différentes.

Un des points importants de GEMAU c'est l'intégration de l'espace des objets (fichiers, autres objets,...) et de l'espace d'exécution dans le mécanisme de protection. En particulier la visibilité (dans l'espace des objets) définit une potentialité d'accès et l'existence de noms différents (avec des protections différentes) permet de définir des accès différents à un même objet. L'association d'un nom local à un objet définit la protection réelle d'utilisation de l'objet.

L'intégration des deux espaces n'existe que dans MULTICS * , dans tous les autres systèmes il est construit indépendamment de l'espace d'exécution d'où son aspect rudimentaire du côté de la protection ce qui se traduit soit par une brèche dans le mécanisme de protection, soit par l'introduction d'un mécanisme supplémentaire et surimposé.

On se reportera au chapitre 2 § 7 pour comprendre avec plus de détails ces deux aspects dans le cas de GEMAU.

<u>Remarque</u> : moment de protection.

Nous avons vu deux moments privilégiés de protection, l'un lors de la liaison, l'autre lors de l'accès. Cependant, bien que hors du cadre traité ici, il est intéressant de citer une autre méthode de protection, à savoir que la protection peut être faite en partie par la machine (à l'exécution) et en partie par le compilateur, comme c'est le cas pour la B6500 [CL 69] . Cette solution n'est possible que si aucun utilisateur ne peut générer directement des instructions "machine" sans passer par les compilateurs.

---

* Pour ne citer que le cas des réalisations, Vanderbilt [VA 69] a intégré dans un modèle théorique les deux aspects.

2.3. Types de protection.

On peut considérer deux aspects fondamentaux au problème de la protection :

a) comment assurer la protection de l'information, c'est-à-dire des objets.

b) comment assurer la protection des mécanismes définis en a).

La protection de l'information est l'aspect le plus souvent cité dans les différents travaux et dans les systèmes classiques (S/360, OS360,...). De plus, il faut considérer, sous cet aspect, comment la protection est définie pour un objet et comment un processus peut accéder à l'objet.

Le second aspect (b) est fondamental, car si les mécanismes de protection ne sont pas eux-mêmes protégés c'est une brèche ouverte dans le système de protection. Ce dernier aspect concerne le contrôle de la mise en place d'une protection (et de son retrait). La solution qui est très souvent utilisée consiste à ne permettre le positionnement de ces mécanismes qu'au système lui-même (mode maître), ce qui est contraire aux idées que nous émettons ici car alors le système est un ensemble de programmes privilégiés. Ce que nous voulons c'est définir une hiérarchie et donc un moyen d'établir cette hiérarchie et de la protéger. Il y a à cela deux solutions : soit avoir des opérateurs de manipulation de protection et il faut appliquer la protection à ces mécanismes (cf Ferrié [FE 74], GEMAU), soit définir à un niveau inférieur la protection du niveau supérieur : ce sont les machines récursives de H.C. Lauer [LA 72], [LA 74], et les études que nous avons menées dans le cadre de la B8500 [EC 69].

2.4. Aspect dynamique de la protection.

Considérons le problème de la compilation d'un programme. L'utilisateur (en fait un processus) effectue un appel au compilateur en lui passant comme paramètres l'ensemble de ses propres objets qui sont nécessaires à la compilation. Ce sont par exemple les fichiers "source" et "imprimante", ainsi que le segment qui va contenir le programme compilé. Lors de l'exécution du compilateur, il ne faut pas que ce dernier puisse accéder aux objets de l'utilisateur qui ne lui sont pas nécessaires, et même l'accès aux objets nécessaires doit pouvoir être protégé (lecture seule, destruction interdite, etc...).

Ce dernier aspect de la protection a été traité au paragraphe précédent. Par contre le compilateur doit pouvoir soit accéder à des objets qui ne sont pas accessibles par l'utilisateur, soit effectuer des entrées-sorties sur des périphériques inconnus de l'utilisateur.

Ce problème de protection est identique au problème classique des modes *maître et esclave* (privilégié ou non). Lors d'un appel au système, le processus change de mode et peut effectuer un certain nombre d'opérations qui n'étaient pas possibles auparavant. Cependant, nous voulons généraliser le mécanisme car, de même qu'il faut protéger le système contre l'utilisateur, il faut protéger l'utilisateur contre le système. Ce problème très important que nous avons déjà vu est connu sous le nom de *méfiance mutuelle*.

Ces différents exemples montrent *l'aspect essentiellement dynamique* de la protection, car les pouvoirs d'un processus sur les objets changent dans le temps (appels procéduraux).

Un *domaine* est défini comme l'ensemble des objets accessibles par un processus à un instant donné. Durant l'exécution un processus peut changer dynamiquement de domaine de protection. Ce qui caractérise le point de vue du système par opposition à celui de l'utilisateur c'est cet aspect dynamique de la protection appliquée à un processus donné. L'idée des domaines de protection est d'indiquer les objets accessibles, la façon dont ils peuvent être accédés, ainsi que les règles strictes de changement de domaine de protection pour un processus donné.

Nous voyons qu'un domaine est donc un ensemble d'objets munis de leurs droits d'accès. A un instant donné un processus est situé dans un domaine particulier et les seuls objets utilisables par ce processus sont ceux du domaine. Par exemple dans GEMAU un domaine est constitué par l'espace d'exécution courant et l'espace visible défini par l'environnement courant. En fait il y a des liens très étroits avec les problèmes d'adressage que nous avons vus au chapitre précédent : d'une façon générale, le domaine est constitué des différents ensembles d'objets nécessaires à l'exécution d'une procédure. Ces ensembles sont composés des objets :

. accessibles par toutes les procédures du processus.
. passés en paramètre
. locaux à la procédure
. rémanents à la procédure.

Nous ne reviendrons pas plus en détail sur ces différents ensembles qui ont été examinés au chapitre précédent.

2.5. Règles de changement de domaines.

Ce sont les règles de changement de domaines qui constituent l'ossature d'un système de protection. Les changements peuvent être effectués de diverses manières, mais doivent toujours être contrôlés. Ce sont :

. des appels procéduraux
. des appels de coroutine
. des branchements sans retour ("*goto*" non local)
. des retours de procédure.

Nous ne nous intéressons pas spécialement aux opérateurs eux-mêmes mais plutôt à l'instant où a lieu ce changement. En effet un appel de procédure correspond à deux types différents de besoins :

. programme utilitaire
. programme de service

Un programme utilitaire a généralement autant de pouvoirs que l'appelant, et ne nécessite pas des pouvoirs spéciaux. De plus, ce programme peut accéder aux objets du domaine de l'appelant.

Un programme de service a besoin de pouvoirs différents, par exemple s'il s'agit de gestion de périphériques, ce programme doit pouvoir effectuer des entrées-sorties alors que l'appelant n'en avait pas le pouvoir, en particulier le programme appelé doit pouvoir accéder à des objets situés dans un

domaine qui ne le permettait pas à l'appelant. Ce dernier point impose l'existence d'objet privilégiés : les *points d'entrée*. Ce sont les seuls objets du domaine accessibles depuis le monde extérieur. Une fois que l'on a appelé l'un de ces objets le domaine du processus appelant change pour devenir celui de l'objet accédé. La fin d'exécution de cet objet, dans le cas d'appels procéduraux, entraîne le retour vers l'objet appelant avec le domaine avant l'appel. Le cas des branchements sans retour permet simplement la substitution d'un domaine par un autre. Cette notion de point d'entrée (appelée aussi "procédure protégée" dans [HO 72]), se retrouve dans la plupart des systèmes de protection.

Nous examinerons ces règles sur un certain nombre d'exemples.

3. MODELE THEORIQUE DE PROTECTION

Nous consacrons un paragraphe particulier au modèle théorique développé par Lampson [LA 71] et repris par Graham et Denning [GR 72], vu l'importance de ce modèle.

Dans ce modèle les objets ont une identification unique, et les entités manipulant les objets sont appelées sujet et sont aussi des objets. La protection est définie par une *matrice d'accès* qui représente pour chaque sujet et chaque objet les opérations possibles. La figure 1 donne un exemple de matrice d'accès. Pour simplifier dans un premier temps, nous considérerons que les sujets sont des processus. Dans la matrice d'accès sont indiquées les relations entre les processus et les différents objets. La figure 1 indique les accès (ou opérations) que chaque processus peut effectuer à un instant donné.

	P1	P2	01	02	03
P1			lire	propriétaire écrire	lire
P2			écrire détruire propriétaire	pas d'accès	propriétaire

Figure 1 : Matrice d'accès

Dans leur modèle Graham et Denning ont dû introduire la notion de *propriétaire* : tout objet a un sujet qui en est le propriétaire, il ne peut y avoir qu'un seul propriétaire par objet (cf. le paragraphe sur la résiliation d'accès pour une discussion complète sur ce sujet). Un des problèmes essentiels dû aux systèmes de protection est l'aspect dynamique de cette matrice d'accès. Il faut fournir la possibilité de donner accès à certains objets par certains sujets. Ceci peut être fait par le propriétaire de l'objet, mais il faut permettre à d'autres sujets de pouvoir transférer ou non certains droits d'accès qu'ils possèdent sur cet objet. Ceci est réalisé par l'existence

d'un attribut spécial appelé "*copie*" qui permet la copie des droits d'accès. La création ou la destruction d'un objet entraine l'addition, ou la suppression d'une nouvelle colonne dans la matrice d'accès. Le créateur devient automatiquement le propriétaire de l'objet. La création d'un sujet entraîne l'addition d'une nouvelle ligne et d'une nouvelle colonne dans la matrice d'accès. Ainsi on peut définir une hiérarchie des sujets.

Considérons auparavant les différentes façon de représenter la matrice d'accès. Il y a en gros deux possibilités : par colonnes ou par lignes.

a) par colonnes

On associe à chaque objet la liste des sujets qui peuvent l'accéder avec le type d'accès autorisé. Ce qui caractérise cette solution c'est que tout processus peut nommer (accéder) potentiellement tout objet. C'est la solution adoptée par MULTICS, où le sujet est identifié par un numéro d'utilisateur. La figure 2 correspond à la définition de la matrice d'accès donnée par la figure 1. Cette solution est connue sous le nom de liste d'accès.

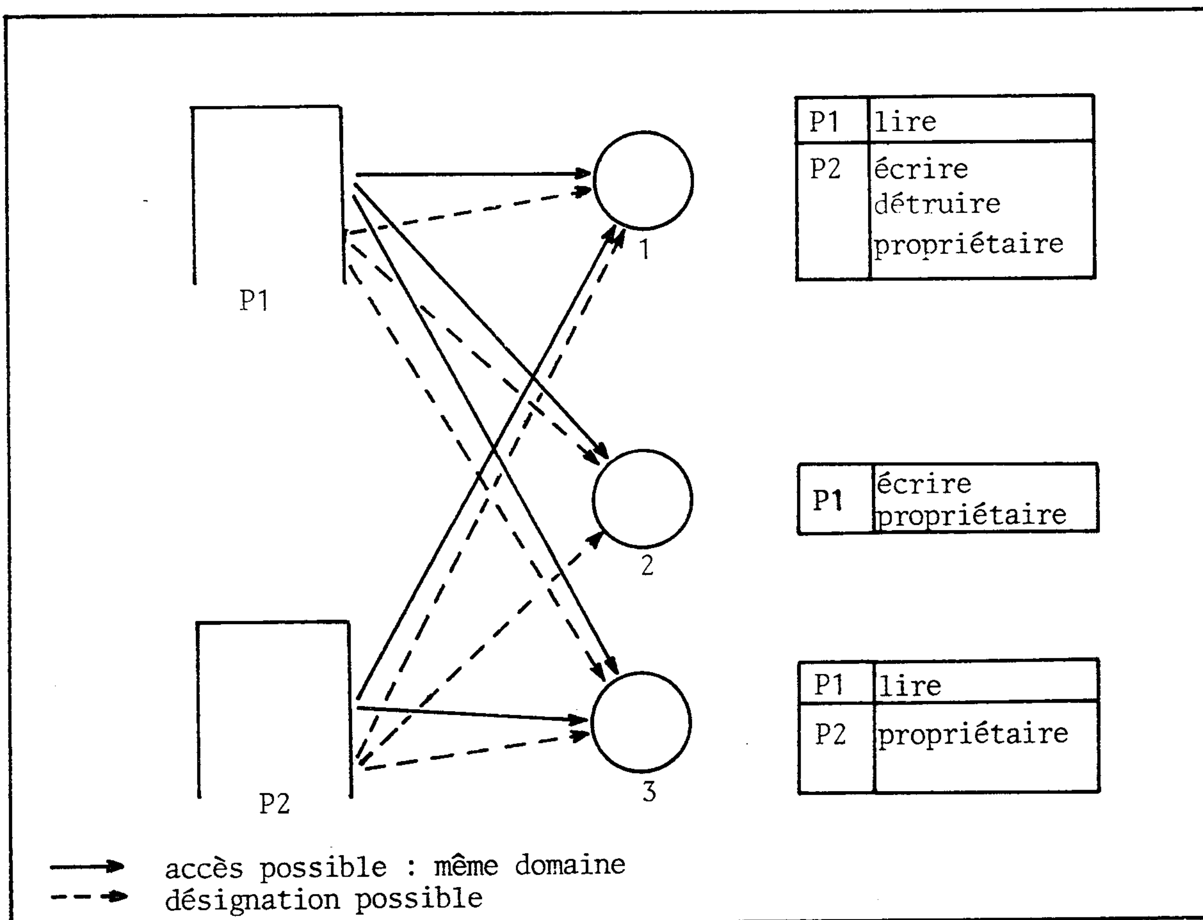


Figure 2 : Liste d'accès

b) par lignes.

C'est associer à chaque sujet la liste des objets accessibles, avec leurs droits d'accès. Cela correspond, en gros, aux environnements de GEMAU ou à la C-List ("capability list") de Dennis et Van Horn [DE 66].

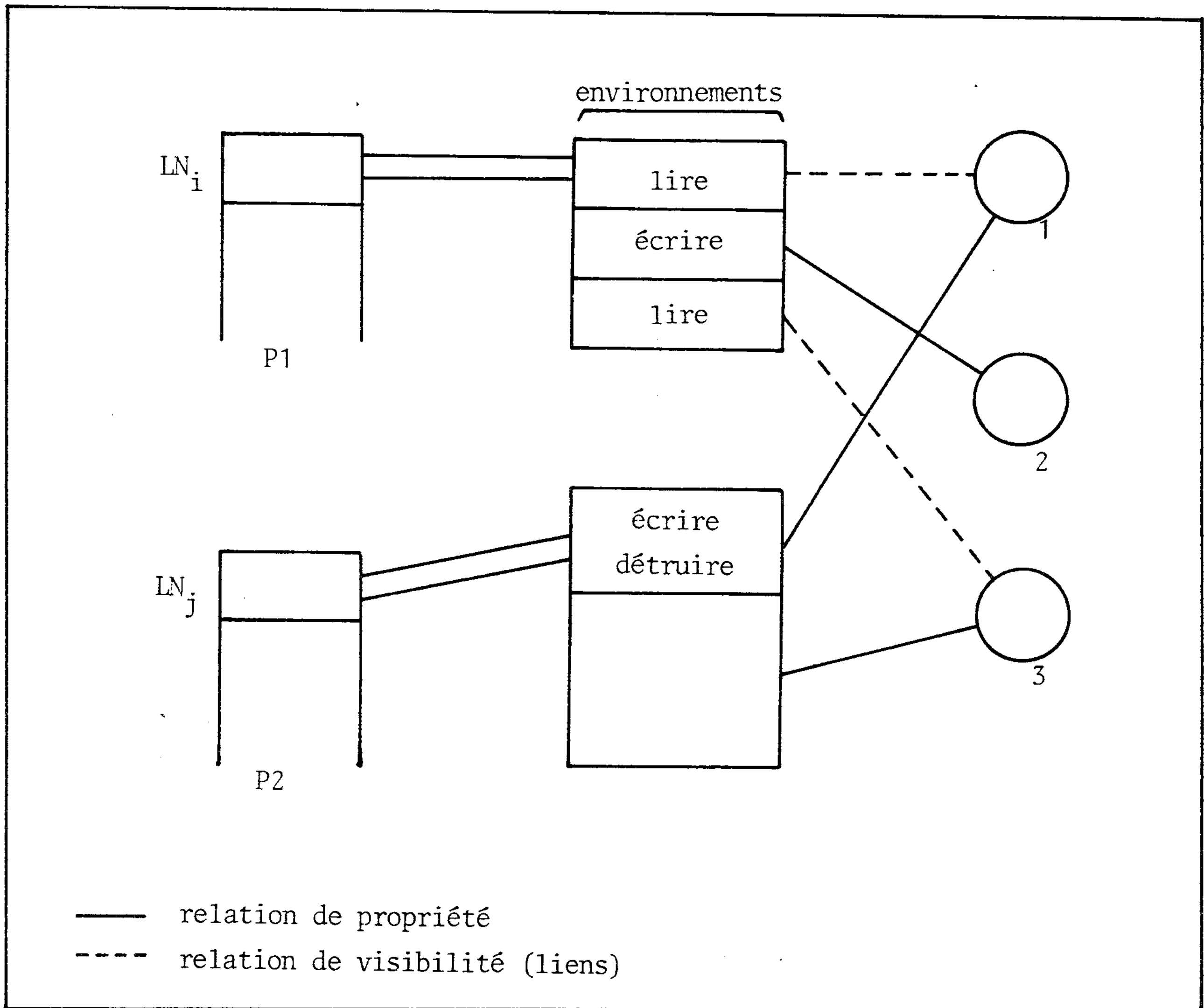


Figure 3 : Environnements

Dans ce cas les relations entre les sujets définissent les relations entre environnements. De plus les objets peuvent être nommés différemment dans des environnements différents, alors qu'aucune hypothèse n'est faite à ce sujet dans la réalisation par colonnes.

Nous voyons apparaître sur la figure 3 un point important dans un espace de désignation, à savoir que celui-ci est un méta-graphe traduisant deux relations :

. l'appartenance à une arborescence,

. la notion de visibilité qui est un graphe quelconque.

La propriété est un droit différent de la visibilité. Ce point est fondamental dans GEMAU.

En fait cette dernière solution peut être réalisée différemment si l'on adopte une solution du style de celle proposée par Fabry [FA 74]. Il s'agit de mélanger dans les segments données et descripteurs, au lieu de regrouper les descripteurs dans des objets spéciaux (annuaire, environnement, C-list ...). Cette solution impose les machines dites "à descripteurs" où chacune des informations en mémoire est préfixée par un indicateur pour spécifier s'il s'agit d'une donnée ou d'un descripteur. On peut ainsi transmettre des descripteurs entre des objets différents.

Un premier problème qui apparaît d'entrée avec cette solution est lié au transfert de la valeur d'un segment vers le monde extérieur, si ce segment comporte des descripteurs. Il sera toujours possible de transférer des descripteurs vers l'extérieur, mais le transfert vers un segment de cette valeur n'est plus possible, car il n'y a plus aucune sécurité sur les nouveaux descripteurs vu que la copie sur le monde extérieur a pu être modifiée sans aucun problème. Nous verrons un deuxième problème à cette solution dans le cadre du contrôle de la diffusion des droits d'accès (descripteurs) sur un objet (cf § 7).

4. PROTECTION A L'EXECUTION.

Nous ne considérons dans ce paragraphe que des objets liés à un espace d'exécution, sans nous poser le problème de la liaison dans cet espace depuis un autre espace (celui des objets en particulier).

4.1. Espace d'exécution statique.

Nous avons vu qu'un des aspects fondamentaux de la protection est son dynamisme, dans le cas d'un espace d'exécution statique le changement de procédure (appel, retour) ne change rien à la visibilité des objets liés dans l'espace d'exécution. Il faut donc introduire des mécanismes supplémentaires, en surimpression, pour limiter l'accès uniquement à certains objets :

Nous allons étudier un ensemble de solutions.

a) mot de passe.

Il faut que la procédure appelante fournisse un code, ensuite le système vérifie que le code permet l'accès au domaine indiqué, car lors de la création du domaine soit le mot de passe permettant l'accès est indiqué, soit la procédure accédée vérifie elle-même la validité de l'accès. Ce type de solution présente l'inconvénient de se prêter facilement à la contrefaçon. En effet, il est facile d'obtenir l'accès une fois que l'on a deviné le mot de passe. Lampson suggère une solution [LA 69] où les mots de passe sont des objets d'un nouveau type qui peuvent seulement être créés, détruits ou transmis comme les autres descripteurs d'objets ("capability"). Dans ce cas, il suffit d'avoir un objet de type mot de passe, et l'accès ne peut se faire que si on fournit le descripteur du mot de passe (par un nom local).

b) clé d'accès.

Une clé (chaîne de chiffres binaires, chaîne de caractères, entiers, ...) est associée à un processus. De même à chaque objet on associe une liste de couples : (clé, protection); cette liste est appelée liste d'accès. Un domaine est constitué par l'ensemble des objets qui possèdent la même clé dans leur liste d'accès. Lors d'un accès par un processus, on vérifie l'existence des clés identiques et d'opérations possibles. La figure 4 donne l'exemple de deux processus P1 et P2 de clés courantes respectivement X et Y. Les domaines respectifs de P1 et P2 sont : {1, 2} et {1, 3, 4}.

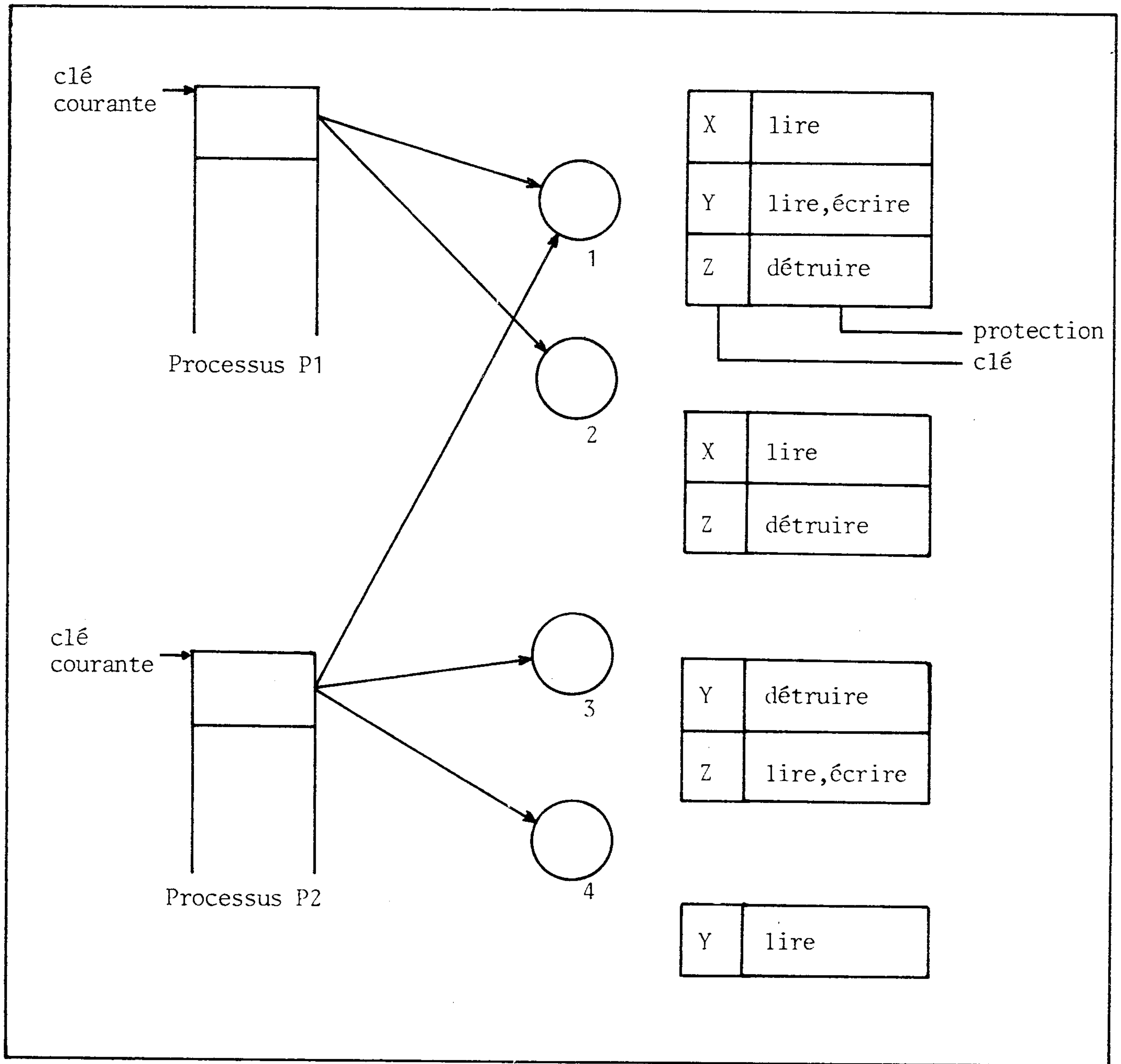


Figure 4 : Clés de protection

Cette solution offre l'inconvénient de ne pas être complètement sûre, en effet, on pourrait imaginer qu'un processus puisse fabriquer une fausse clé. Dans ce cas, on peut utiliser le mécanisme de descripteur décrit par Lampson [LA 69], c'est-à-dire qu'une clé est l'équivalente d'un nom universel et qu'elle ne peut être manipulée qu'à l'aide de certains opérateurs.

Cette solution correspond à une réalisation de la matrice d'accès par colonnes.

c) <u>anneaux</u>

Le mécanisme des anneaux de MULTICS est une variante du mécanisme des clés d'accès. Dans ce cas on veut généraliser la notion maître-esclave, et établir une hiérarchie entre les objets. La clé est un simple entier compris entre 0 et n (dans le matériel n = 7 : cf Schroeder [SC 72]). Un numéro d'anneau est associé au processus, soit i ce numéro ; le processus n'a aucun accès aux objets de numéro d'anneau j tel que $i > j$, et un accès si $i \leq j$. Dans ce dernier cas les restrictions d'accès liées au nom de l'objet sont à prendre en compte. Ainsi ce n'est pas parce qu'un objet est dans un rang d'anneau supérieur que toutes les opérations sont possibles. En fait, à un objet on associe trois numéros d'anneau : n1, n2 et n3 (avec $n1 \leq n2 \leq n3$).Dans le cas de MULTICS les objets sont uniquement des segments et les opérateurs possibles sur ces objets sont : lecture, écriture, exécution.

La règle est simple :

- $i \leq n1$: tout accès possible
- $n1 < i \leq 2$: lecture/exécution autorisées
- $i > n2$: pas d'accès

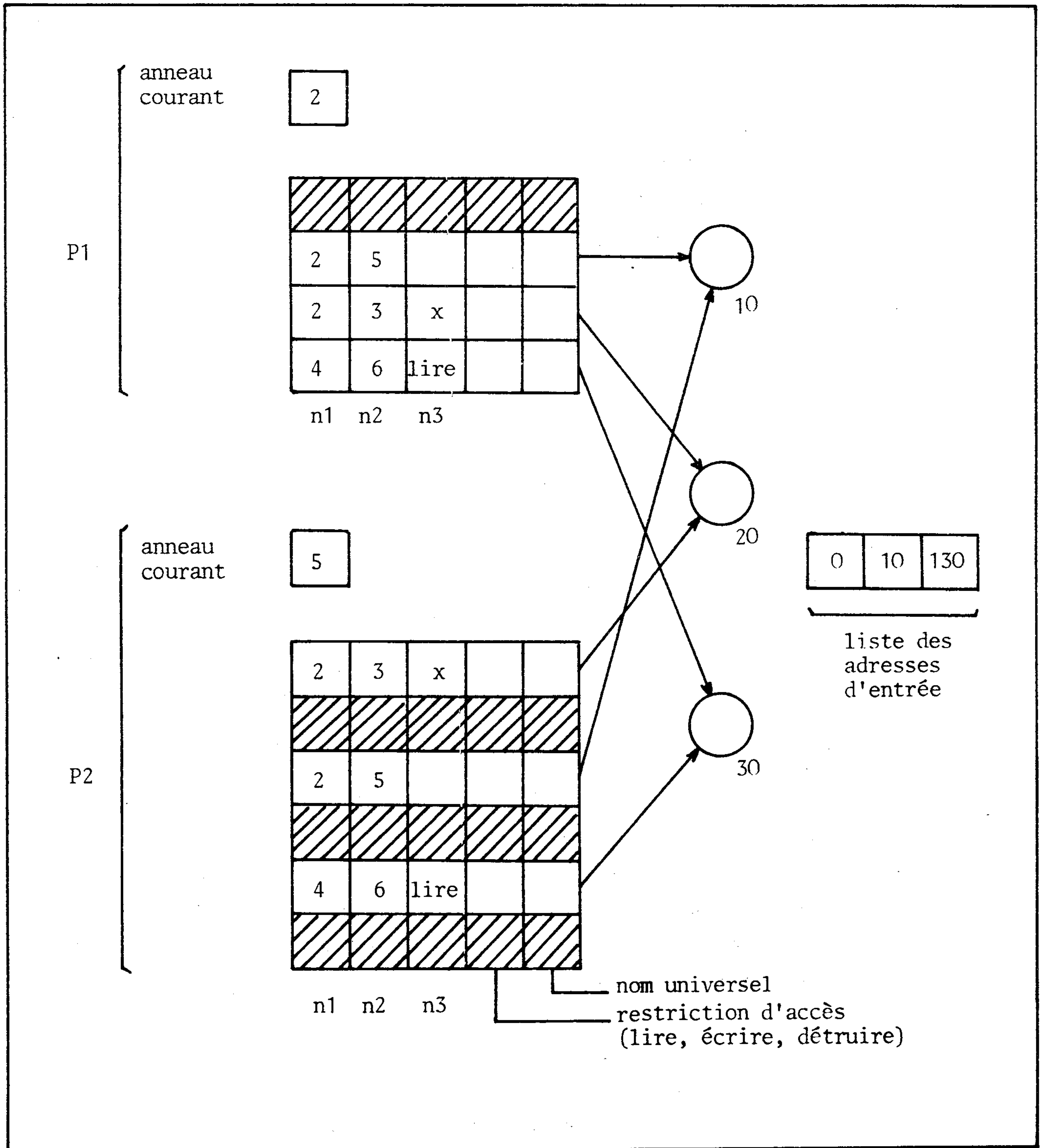


Figure 5 : Anneaux de MULTICS

La figure 5 donne l'exemple de 2 processus P1 et P2, de numéros d'anneau respectifs : 2 et 5.

P1 peut accéder aux objets 10, 20 et 30 en tout accès,

P2 peut accéder aux objets 10 et 30 en lecture et exécution.

Cette accessibilité peut être modulée par les restrictions d'accès sur l'objet (cf. § 5.1. du même chapitre). Par exemple, sur la figure 5 l'objet 3 est défini comme ayant des restrictions qui ne permettent que la lecture. Donc les processus P1 et P2 ne pourront jamais accéder qu'en lecture seule à condition que leur numéro d'anneau soit inférieur à 6.

Le troisième numéro d'anneau (n3) joue un rôle dans la définition des changements de domaine et sert pour les adresses d'entrée* autorisées dans un segment de type procédure. Si le numéro d'anneau du processus est i, et que l'on a la relation

$$n2 < i \leq n3$$

alors l'accès à l'objet ne peut être qu'un appel procédural ou un transfert sans retour, mais cet appel (ou ce transfert) ne peut se faire que vers l'une des adresses d'entrée spécifiées.

L'anneau résultant lors d'un appel ou transfert (nouveau domaine) est donné par le tableau suivant où i représente le numéro d'anneau de l'appelant.

$i \leq n1$	n1
$n1 < i \leq n2$	i
$n2 < i \leq n3$	n2 à condition que ce soit un point d'entrée (adresse)
$n3 < i$	aucun accès

* ces adresses sont appelées "point d'entrée" dans MULTICS, nous préférons les appeler "adresse" car il ne s'agit pas vraiment des points d'entrées dans un domaine au sens où nous l'avons défini au paragraphe § 2.5. Il s'agit d'adresses, autorisées en accès depuis l'extérieur, dans un seul et même segment.

Reprenons l'exemple de la figure précédente, ainsi si l'objet 2 possède comme numéro d'anneau n3 la valeur 6 alors le processus P2 peut exécuter cet objet (par appel à l'aide du nom local dans lequel la procédure a été liée) à condition que l'appel soit sur l'une des trois adresses d'entrée (déplacement à indiquer dans l'instruction d'appel) i.e. ici 0, 10 ou 130. L'anneau courant de P2 devient alors n2, c'est-à-dire 3.

Par contre, P1 peut accéder n'importe quelle partie de l'objet 2 et en particulier l'exécuter même en dehors des adresses d'entrée.

Pour conserver l'aspect procédural des appels de domaines, il faut ajouter une* nouvelle pile par processus pour conserver les anneaux successifs.

Schroeder et Saltzer indiquent que leur objectif est la généralisation du mode maître/esclave. Cependant leur solution ne marche que parce que tous les objets sont des segments, c'est-à-dire que l'on connaît les opérateurs applicables : lire, écrire et exécuter ; et que l'on peut, dès lors, les ordonner les uns par rapport aux autres. Si on essaie de généraliser en introduisant des types d'objets différents alors il faudrait étendre le mécanisme d'anneaux en classant les opérateurs applicables et en définissant autant de numéros d'anneaux que d'opérateurs.

Cependant l'espace d'exécution étant statique on peut considérer que l'un des objectifs cherchés a été atteint, c'est-à-dire la définition d'un mécanisme qui puisse se réaliser facilement à l'aide de matériel.

Mais l'aspect évolutif de la sélectivité des accès n'est pas possible, car il faudrait pouvoir changer n1 et n2, si l'on voulait qu'à un instant donné un segment ne soit accédé qu'avec une restriction de protection, par exemple passer en paramètre, à une procédure, un segment en restreignant les opérateurs possibles à la lecture seule.

---

* dans la réalité il en faut 7 : une par anneau de protection

d) RC4000

Une solution plus générale est introduite dans le RC4000 [BR 69] . En effet,avec la solution de MULTICS on ne peut résoudre le problème suivant : soit quatre segments A, B, C et D où nous supposons que :

A peut accéder A,C et D

B peut accéder A,B et C

C peut accéder C

Comment décrire cette accessibilité en MULTICS ?

On peut décrire ces relations sous la forme du graphe donné par la figure 6, et mettre chacun des segments dans un numéro d'anneau différent Pour A et B, le problème ne se pose pas, il suffit de donner à B un numéro inférieur à A.

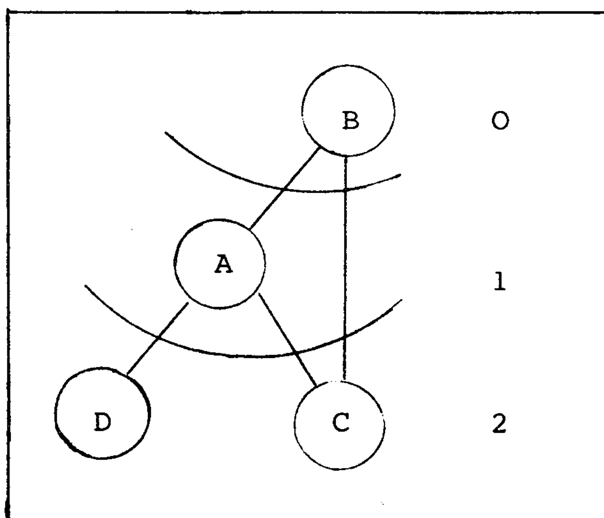


Figure 6.

Cependant, pour D et C, le problème est plus complexe, car il faut que leurs numéros soient supérieurs à celui de A, mais de plus D et C n'ayant aucune relation entre eux on ne peut traduire ce fait avec des numéros d'anneau. La solution utilisée dans le RC4000 est d'affecter à chaque segment* une clé, qui est un nombre compris entre 0 et $2^n - 1$ (sur RC4000: n=3). Et d'associer au processus un registre de protection à 2^n entrées. L'accès et la modification des mots de clé k est autorisée si la Kième entrée du

* dans la pratique cette clé est associée à chaque mot.

registre de protection l'autorise (= 1). Dans le cas de notre exemple il suffit d'affecter les clés 0, 1, 2 et 3 aux mots de B, A, C et D. L'exécution de A, B, C et D nécessite donc pour chacune de ces exécutions, les valeurs suivantes du registre de protection :

clé	0	1	2	3
A	1	0	0	0
B	0	0	0	1
C	1	1	0	1
D	1	1	1	0

Cette solution souffre cependant du manque de sélectivité lors de l'accès, il faudrait alors définir plusieurs registres de protection par processus, un pour chaque opération (ou type d'opération).

4.2. Espace d'exécution évolutif

Dans le cas des espaces d'exécution évolutif, l'aspect dynamique des domaines est rendu de facon naturelle. Si nous considérons les espaces statiques, nous remarquons les problèmes suivants :

a) Comme il n'y a pas d'aspect dynamique dans ce type d'espace, il faut ajouter une pile dans le processus pour définir la succession des différents domaines (anneaux par exemple), de même qu'il fallait une pile (section de liaison) pour traduire la succession des différents appels de procédures avec leurs paramètres associés.

b) La difficulté pour un même processus d'accèder à un objet avec une certaine protection à un instant donné et une protection différente (restriction) à un autre instant. Ce changement de protection est lié à l'objet (propriété intrinsèque) et non à l'utilisation de l'objet. Ainsi dans MULTICS on ne peut restreindre la protection d'un objet car cet objet ne possède qu'un seul nom local dans l'espace d'exécution.

Tous les systèmes à espace d'exécution évolutif résolvent simplement les problèmes a) et b). Rappelons à cet effet qu'un nom local contient les informations suivantes :

<nom universel de l'objet>
<type de l'objet>
<protection>

Ainsi, le même objet peut être lié dans les espaces différents ou dans le même espace sous des noms locaux différents avec une protection différente.

Le changement de domaine de protection se fait lors d'un appel (CALL) ou retour (RETURN) de procédure.

a) HYDRA [WU 74]

Comme dans GEMAU, la protection peut être divisée en deux classes l'une dépendante du type de l'objet ("auxiliary rights") l'autre indépendante ("Kernel rights"). La première classe est définie uniquement par le créateur du type. Dans tous les cas, le rôle du noyau est simplement de vérifier que les informations de protection autorisent l'accès sans avoir à connaître la signification exacte de ces informations.

Le changement de domaine s'effectue par appel de procédure (CALL). Les noms locaux contiennent la protection actuelle sur l'objet (les deux classes) et le nom universel de l'objet. Dans le cas d'un objet de type procédure, cet objet contient un squelette d'espace d'exécution ainsi que nous l'avons vu au chapitre précédent. Ce squelette décrit les paramètres formels, et en particulier sur le plan de la protection deux informations : clé d'accès et protection réelle. Lors d'un appel de procédure le noyau vérifie que la protection actuelle des paramètres réels est un sous-ensemble de la clé de chacun des paramètres formels. Si cette vérification est correcte alors l'accès est autorisé et les protections affectées aux noms locaux sont celles indiquées dans le squelette . La figure 7 donne un exemple dans le cas de HYDRA. En particulier les protections associées aux objets LN2, LN3 et LN5 dans l'espace d'exécution de l'appelant (sous-sommet) sont respectivement pY, pZ et pV. Pour que l'appel (*appeler*(LN2, LN3, LN5)) puisse s'effectuer il faut que les protections actuelles (pY, pZ, pV) soient inclues dans les clés associées dans le squelette (y, z, v).

La solution réalisée par HYDRA implique que la procédure appelée puisse accéder aux paramètres avec un pouvoir plus grand que n'en avait l'appelant, en effet il n'y a aucune relation entre la protection du paramètre réel et celle définie dans le squelette pour être prise en compte lors

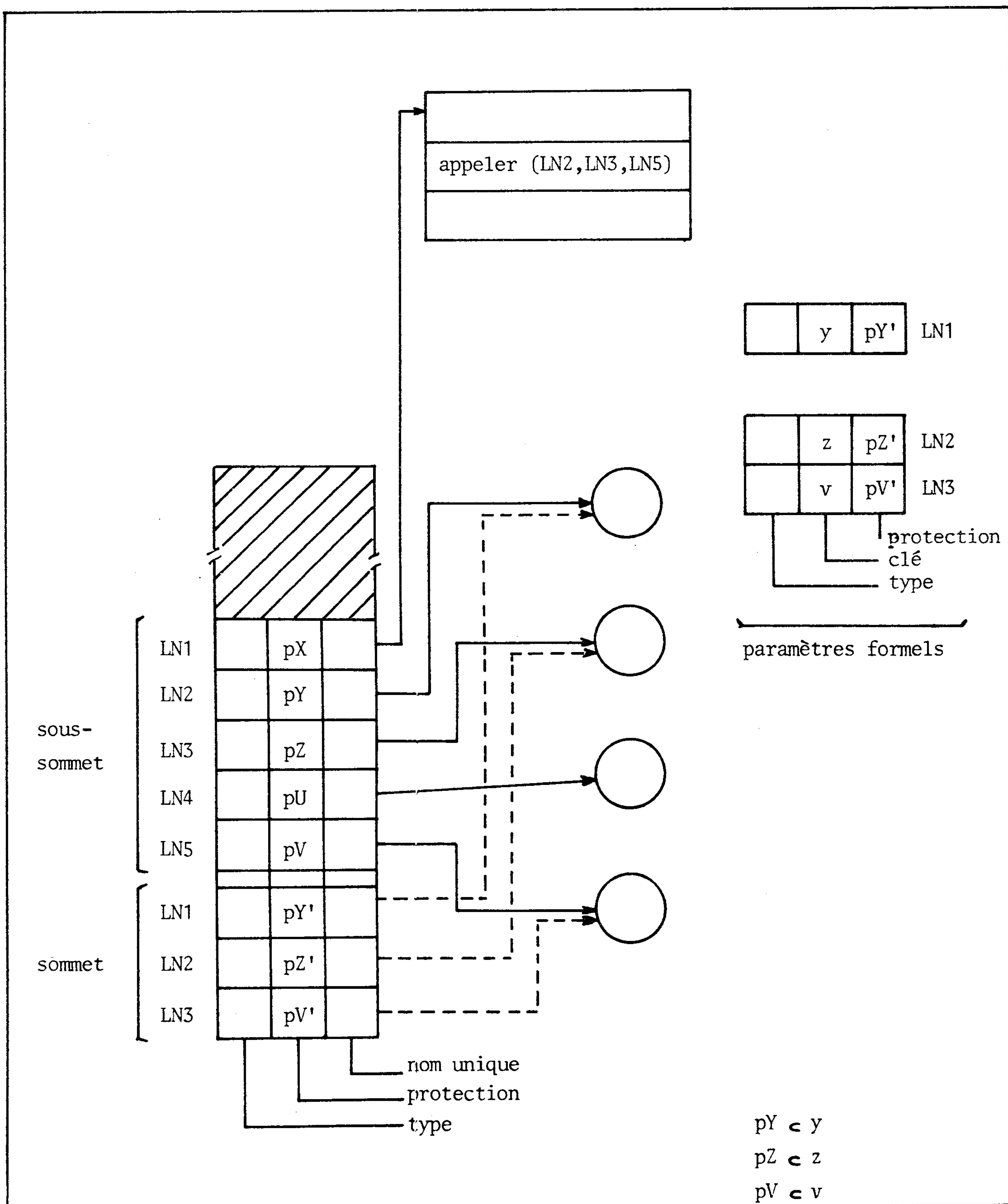


Figure 7 : Protection lors d'appel de procédure dans HYDRA

de l'exécution. Mais en aucun cas l'appelant ne peut obtenir cette dernière protection car elle est indiquée dans un espace d'exécution différent.

Cependant, cette solution n'est pas complètement sûre, car il n'y a plus aucune protection sur les objets passés en paramètres, il faut alors ajouter un mécanisme supplémentaire qui est l'autorisation de créer des objets de type procédure. Et même dans ce cas, l'utilisateur peut mettre la protection qu'il désire dans le squelette

De plus il n'y a aucun moyen, comme dans GEMAU, pour que l'appelant puisse restreindre ce qui pourra être fait avec les paramètres qu'il fournit à l'appel, car la protection est définie par l'appelé.

La solution énoncée ici s'apparente fortement à celle des clés décrite au paragraphe 4.1.

b) GEMAU

Nous ne reviendrons pas en détail sur le fonctionnement de la protection dans GEMAU qui a été décrit au chapitre 2 (§ 7). Cependant, nous reprenons l'exemple de HYDRA (figure 7), et nous montrons sur la figure 8 le fonctionnement de ce même exemple. Plusieurs remarques sont possibles :

- lors de l'appel on peut spécifier des restrictions sur les protections des paramètres, la protection finale étant l'intersection des opérations possibles (ou l'union des restrictions). On ne possède que des restrictions et aucune extension ;
- la protection terminale est fournie par l'appelant ;

- bien entendu, il faut que l'objet à exécuter ne possède pas de restriction sur CALL.

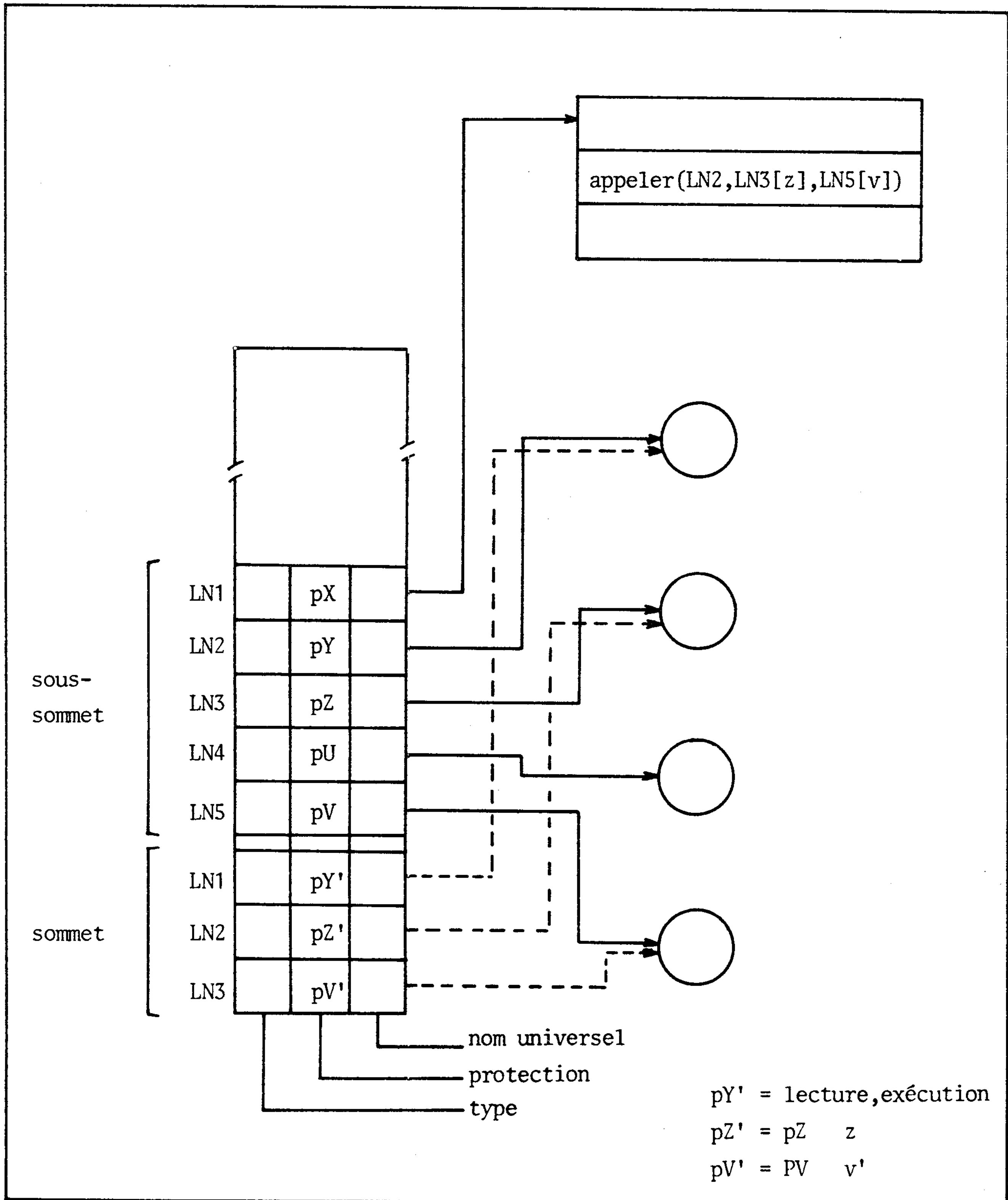


Figure 8 : Protection lors d'appel de procédure dans GEMAU

4.3. Espace d'exécution récursif

Ainsi que nous l'avons vu au chapitre précédent, la différence fondamentale entre les espaces d'exécution évolutifs et récursifs réside dans la façon dont sont fabriqués les noms locaux.

Il suffit donc de compléter les informations nécessaires d'un espace d'exécution récursif. Ces informations sont :

- nom local du segment (dans l'espace courant)
- adresse de début du nouveau segment
- longueur de ce segment
- restriction d'utilisation : lire, écrire, exécuter

La figure 9 schématise dans le cas d'un espace récursif la solution de la figure 8 (cas de GEMAU). Dans ce cas l'utilisateur fabrique lui-même son nouvel espace d'exécution (LN1, LN2 et LN3). Lors de l'exécution effective, le matériel fabrique l'adresse réelle par cumul des adresses (nom locaux) des sous systèmes intermédiaires. De même le matériel va cumuler les restrictions rencontrées dans les différents noms locaux. Ce qui est une solution équivalente, du moins du strict point de vue de la protection des objets dans l'espace d'exécution, à celle de GEMAU.

La protection de l'objet désigné LN2 dans l'espace d'exécution courant est donnée par :

4.4. Conclusion.

Lors de l'exécution l'aspect dynamique de la protection est réalisée très simplement par l'aspect procédural et donc par les espaces d'exécutions évolutifs ou récursifs. Dans tous les autres cas, il faut surimposer un mécanisme supplémentaire.

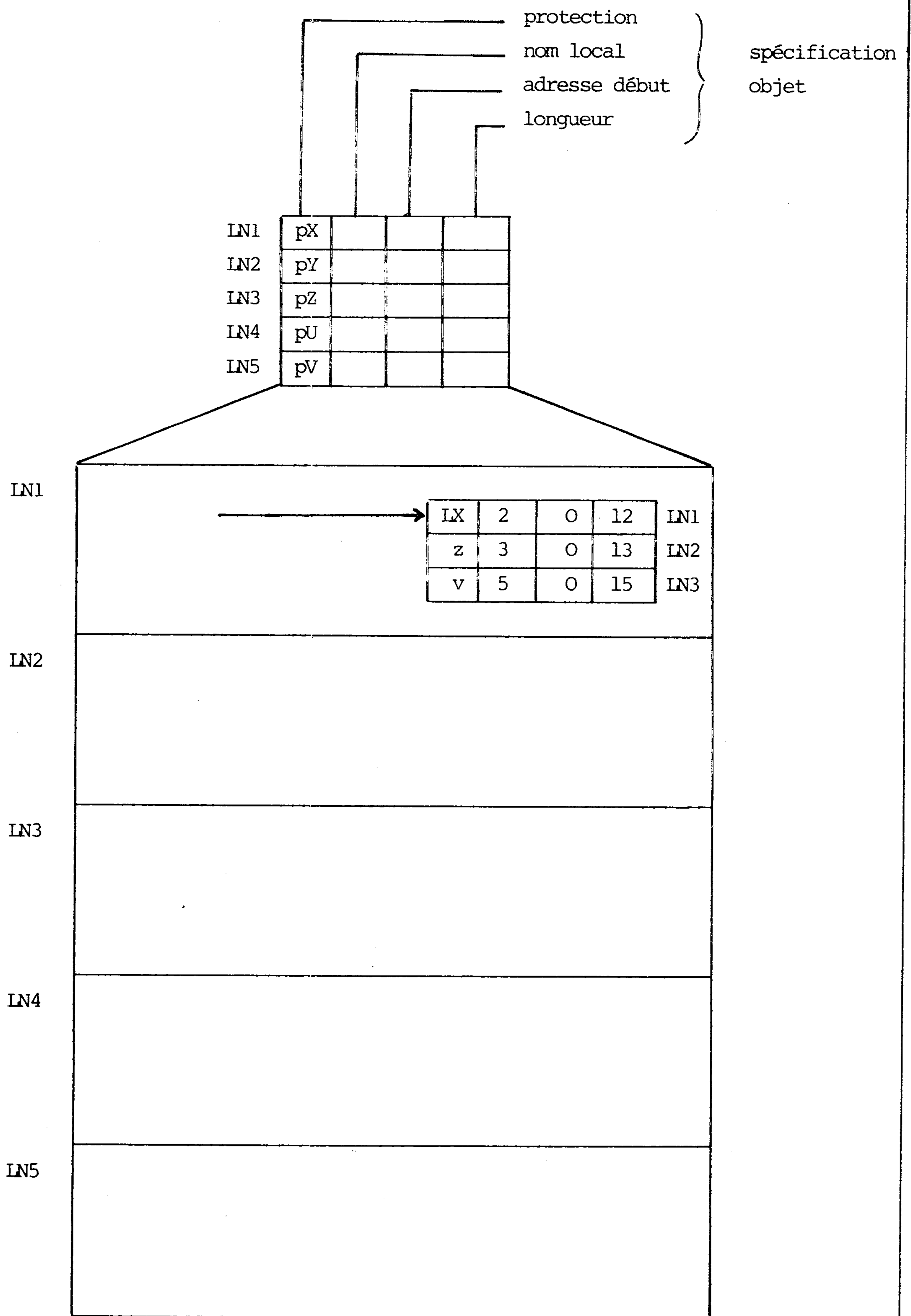


Figure 9 - Protection sur appel de procédure dans le cas d'espace récursif.

5. RELATIONS ENTRE LA STRUCTURE DE L'ESPACE DES OBJETS ET LA PROTECTION.

Nous voulons montrer que le choix de la structure de cet espace influence la protection. Dans le cas de l'espace des objets , la protection est liée de façon intrinsèque aux noms et objets indépendemment de l'utilisation de ces objets. Pour cela nous étudions le cas de MULTICS qui est assez connu puis nous montrons dans le cas de GEMAU comment les problèmes posés par MULTICS sont résolus.

5.1. Espace des objets de MULTICS.

Ce système est décrit dans de nombreux articles seuls Graham [GR 68], Schroeder et Saltzer [SC 72] ainsi que Organick [OR 72] nous intéressent ici.

La structure adoptée est une structure arborescente comme celle de GEMAU. Cette structure est décrite par Daley et Neumann [DA 65] . Il y a deux types d'objets : annuaires et segments, les segments étant des objets terminaux. Un objet possède une entrée "branch" dans un annuaire, des entrées spéciales appelées "liens" permettent de définir des noms différents. Les liens de MULTICS jouent un rôle analogue aux liens complètement résolus de GEMAU.

Un objet peut être désigné de plusieurs façons différentes :

- par un nom partiel absolu, qui est la liste des noms d'entrée depuis la racine jusqu'à l'objet.
- par un nom partiel relatif à un annuaire de référence. Il y a un tel annuaire associé avec chaque processus, cet annuaire est un simple annuaire de travail que l'on peut (à l'aide de primitives) changer à tout instant sans aucun contrôle de la part du système. Un nom partiel relatif permet soit d'aller directement vers les objets terminaux, soit remonter vers la racine et ensuite d'aller vers les objets terminaux.

Ce qu'il est important de savoir c'est qu'à tout instant un processus peut désigner n'importe quel objet dans l'espace d'adressage, nous nous retrouvons donc dans une situation semblable à celle rencontrée dans les espaces d'exécution statiques. Il faut donc ajouter un mécanisme supplémentaire pour permettre de contrôler l'accès des objets par les processus. Pour cela on associe à chaque objet une liste d'accès dont chaque entrée est de la forme :

- nom d'utilisateur ou de classe d'utilisateurs
- protection associée à cette entrée.

Dans ce cas, un processus ne peut jamais changer de domaine de protection, car celui-ci est défini par l'ensemble des objets qu'il peut accéder, c'est-à-dire pour lesquels le nom de l'utilisateur (ou de sa classe) figure dans la liste d'accès. Il n'est pas possible de réaliser l'aspect dynamique des domaines de protections, sauf peut être en créant de nouveaux processus et en demandant un service à un processus qui est dans un domaine différent. Il suffit alors que les deux domaines aient une intersection non nulle pour se transmettre des paramètres. Ce n'est pas la solution qui a été retenue, mais plutôt celle proposée par Graham [GR 68] : les anneaux que nous avons déjà vu au paragraphe § 4.1 . A chaque objet (segment dans le cas de MULTICS) on associe un triplet : (n1, n2, n3). Où n1 et n2 jouent le rôle d'anneaux pour délimiter les possibilités de lecture, écriture et exécution.

Le troisième numéro sert pour les adresses d'entrée protégées dans la procédure.

Si on considère un segment, celui-ci est défini dans un annuaire par :

- un ensemble de noms symboliques
- un nom unique
- n1, n2
- une liste d'adressesd'entrées
- n3
- une liste d'accès (. nom d'utilisateur/classe
  . protection).

Il importe de remarquer que le mécanisme d'anneau cohabite avec les listes d'accès, et que pour pouvoir accéder un objet en écriture,il faut non seulement obéir aux règles des anneaux, mais avoir une autorisation d'écriture (pour cet utilisateur) dans la liste d'accès associée à l'objet.

La réalisation de MULTICS s'effectue avec une pile de contrôle par numéro d'anneau et par processus.

Le reproche que l'on peut faire à une telle solution est de définir un ordre absolu, même entre des parties de l'arbre qui sont disjointes. Ce problème , en plus de la hiérarchie est dû à deux raisons :
- possibilité de nommer tous les objets de l'arbre.
- identification des processus (liste d'accès).

5.2. Cas de GEMAU.

Dans ce cas, la désignation dans l'espace des objets s'effectue toujours à partir d'un descripteur comme dans l'espace d'exécution :
- environnement courant (LNO)
- nom local (LN i)

Il n'y a aucun moyen de nommer des objets pour lesquels il n'existe aucun nom local ou nom partiel issu d'un nom local de l'espace d'exécution courant. Comme on ne peut pas nommer depuis la racine ou remonter vers la racine comme dans MULTICS la protection est absolue et il n'y a pas besoin de mécanismes supplémentaires.

Il n'est pas possible de traduire une structure comme celle de GEMAU à l'aide de mécanismes tels que, mot de passe, clé d'accès, anneau et RC4000.

5.3. Résolution des différents mécanismes à l'aide de GEMAU.

Nous allons montrer comment on peut réaliser les différents mécanismes de protection à l'aide des différents espaces de GEMAU. Ces différentes solutions ne sont ni uniques ni optimales, leurs buts sont simplement d'en montrer la faisabilité.

a) mot de passe.

A chaque objet est associé un mot de passe, pour réaliser ce mécanisme en GEMAU, il suffit, dans l'environnement d'un processus d'avoir un lien "mot de passe" qui référence un annuaire ayant tous les objets du même mot de passe (cf. figure 10).

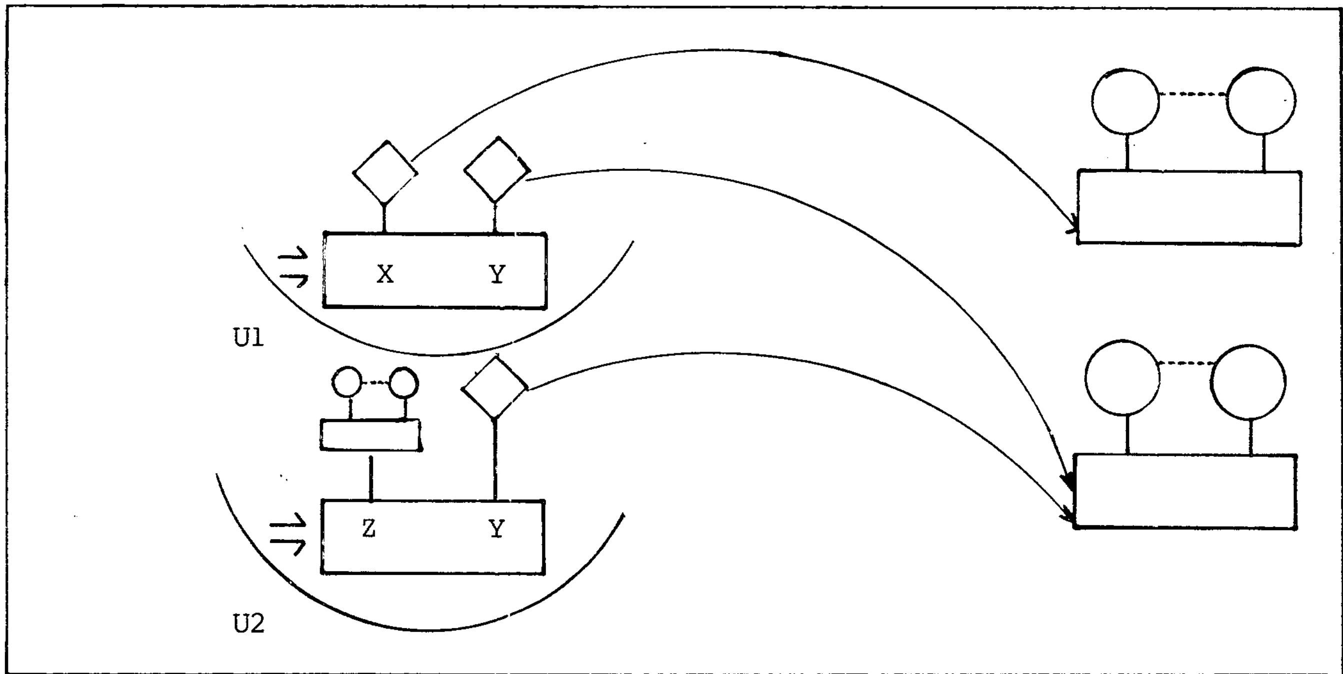


Figure 10 - Solution du mot de passe

Un utilisateur ayant le droit d'utiliser certain mot de passe, est défini par la liste de ses mots de passe. Chacun de ces mots correspond à un annuaire (par l'intermédiaire ou non d'un lien). Le nom d'un objet est donc préfixé par son mot de passe.

b) Clé d'accès.

Reprenons l'exemple donné par la figure 4 (§ 4.1), une solution "GEMAU" en est donnée par la figure 11.

La clé associée à un processus est un environnement qui contient des liens vers tous les objets possèdant la clé, sur chacun de ces liens on indique la protection à appliquer (lire, écrire,...).

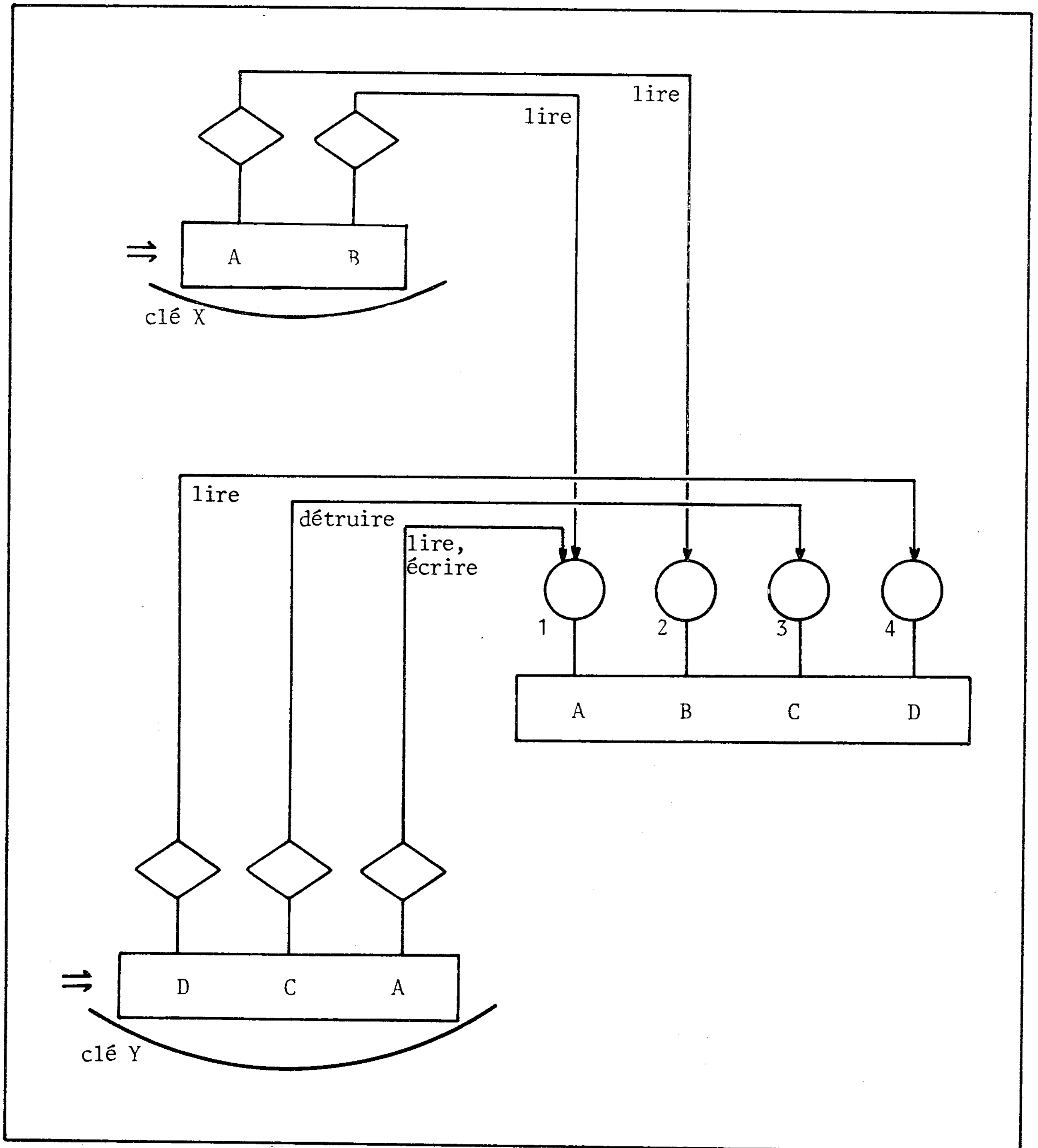


Figure 11 : Solution de la clé d'accès

c) Anneaux de MULTICS

On va définir autant d'environnement que de numéros d'anneaux, et dans chacun de ces environnements on définit les objets accessibles et leur type d'accès à l'aide de liens. Ainsi considérons un objet de numéros d'anneaux:

$$n1 = 2, \quad n2 = 3, \quad n3 = 4$$

Un processus depuis les anneaux 1 et 2 peut accéder à cet objet sans rextrictions, depuis l'anneau 3 en lecture et exécution seulement, depuis l'anneau 4 en exécution seule, et sans accès depuis d'autres anneaux. La figure 12 schématise cette solution.

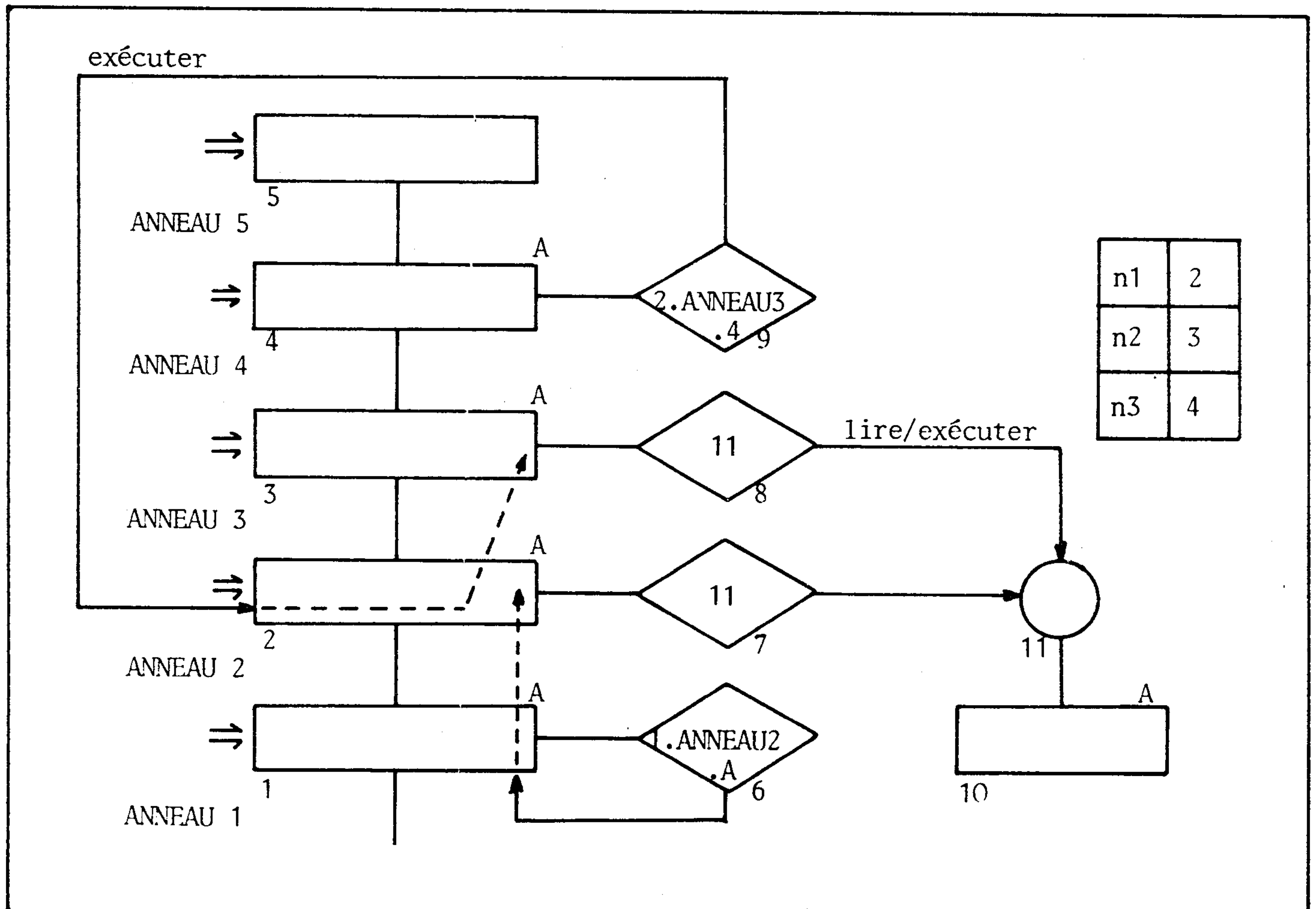


Figure 12 : Solution des anneaux de MULTICS

Cette solution nécessite quelques commentaires :

- les anneaux supérieurs à n3 ne possèdent aucun lien vers l'objet l1.
- les anneaux situés entre n1 et n2 (i.e. ici 2 et 3) ne provoquent pas de changement d'anneaux lors d'une exécution. La valeur du lien est donc directement l1.
- les anneaux inférieurs à n1 (2) doivent provoquer le changement d'anneau lors d'un appel, il faut donc utiliser le mécanisme des liens non résolus pour rentrer dans l'environnement ANNEAU 2 ; d'où la valeur du lien :

1. ANNEAU2. A

- les anneaux situés entre n2 et n3 (ici 4) doivent provoquer un changement d'anneau (donc d'environnement) et être seulement en exécution. D'où la valeur du lien.

2. ANNEAU3. A

Lors d'une exécution le nouvel anneau est ANNEAU3

- il y a une difficulté à traduire la notion d'adresse d'entrée car cette notion n'existe pas dans GEMAU.

d) RC4000

Le problème posé par la figure 6 est facilement traduisible en GEMAU, car on peut décrire un graphe

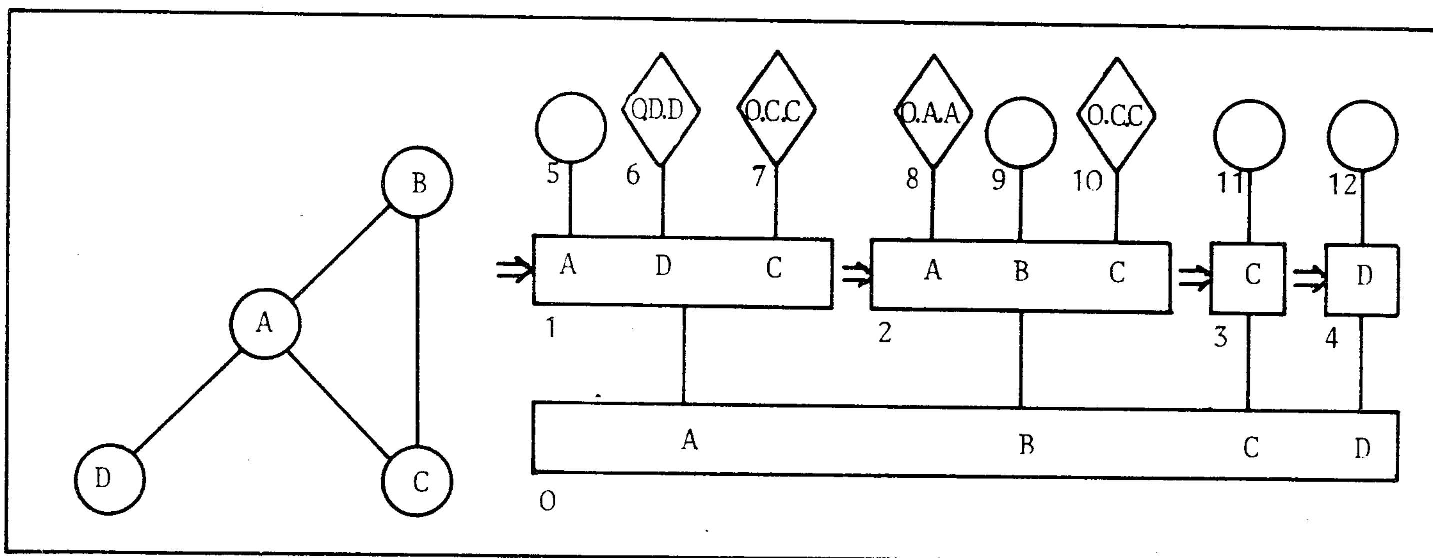


Figure 13 - Solution RC4000

La solution donnée par la figure 13 résoud bien le problème, en effet depuis B on peut accéder A, B et C. L'accès (exécution de A ou C provoque un changement d'environnement (vers 1 ou 3).

6. MEFIANCE RECIPROQUE.

6.1. Définition et solution dans le cas de GEMAU.

Considérons deux utilisateurs U1 et U2. U1 possède un objet, par exemple un fichier X, auquel il veut autoriser l'accès par U2, tout en gardant la possibilité de contrôler à tout instant ce que fait U2 avec ce fichier. Pour cela U1 donne à U2 accès à une fonction de contrôle F_x située dans son propre espace. F_x est un point d'entrée dans U1 pour U2, par l'intermédiaire du lien f_x dans U2 (cf. figure 14)

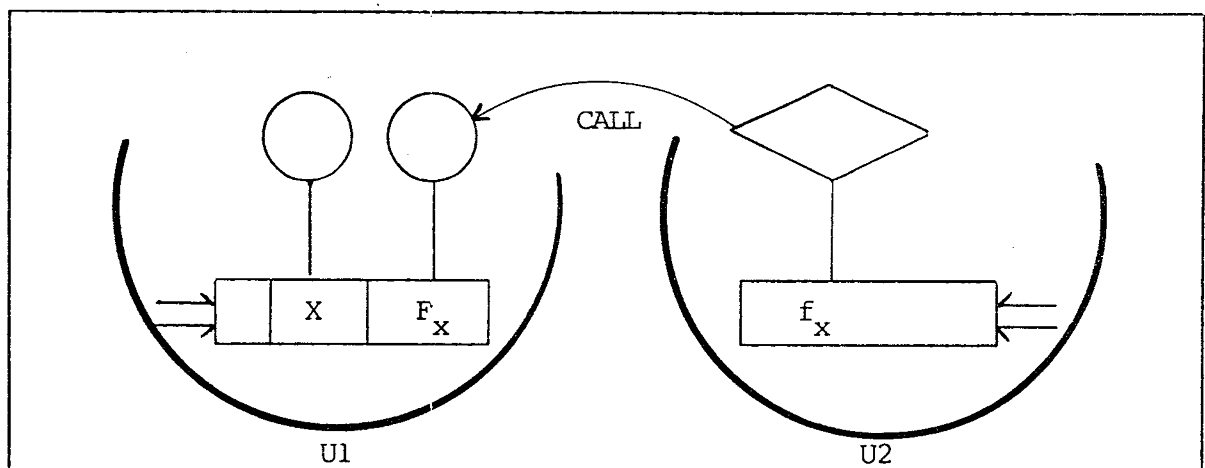


Figure 14 - Relation entre sous-systèmes

Cependant, U1 ne veut pas que U2 puisse modifier, lire ou même copier F_x dans son espace. Pour cela le lien sur F_x est défini avec des protections qui interdisent la destruction, le transfert, la modification ou la copie. C'est-à-dire que la seule primitive applicable sur f_x est CALL.

La fonction F_x est définie dans un environnement à l'intérieur du sous-système U1 d'où elle peut accéder à X. Un processus situé dans U2 ne peut jamais qu'activer F_x par l'intermédiaire de f_x et provoquer ainsi un changement d'environnement. Ceci définit parfaitement la méfiance de U1 par rapport à U2 : U1 est complètement protégé des erreurs de procédures situées dans U2.

Considérons maintenant le point de vue de U2, qui veut bien utiliser le fichier X par l'intermédiaire de f_x, mais il ne veut pas que par erreur, volontaire, ou involontaire, F_x puisse accéder aux objets qui lui appartiennent (i.e. à U2).

En effet, il est possible que U1 n'ait pas défini d'environnement et que l'exécution de F_x ait lieu dans l'environnement de l'appelant (cf. chapitre 2 § 10). Pour cela U1 va se protéger obligeant F_x à s'exécuter dans un environnement spécial (de mise au point) si par hasard F_x ne s'exécutait pas dans l'environnement U1. La figure 15 donne la solution GEMAU à ce problème.

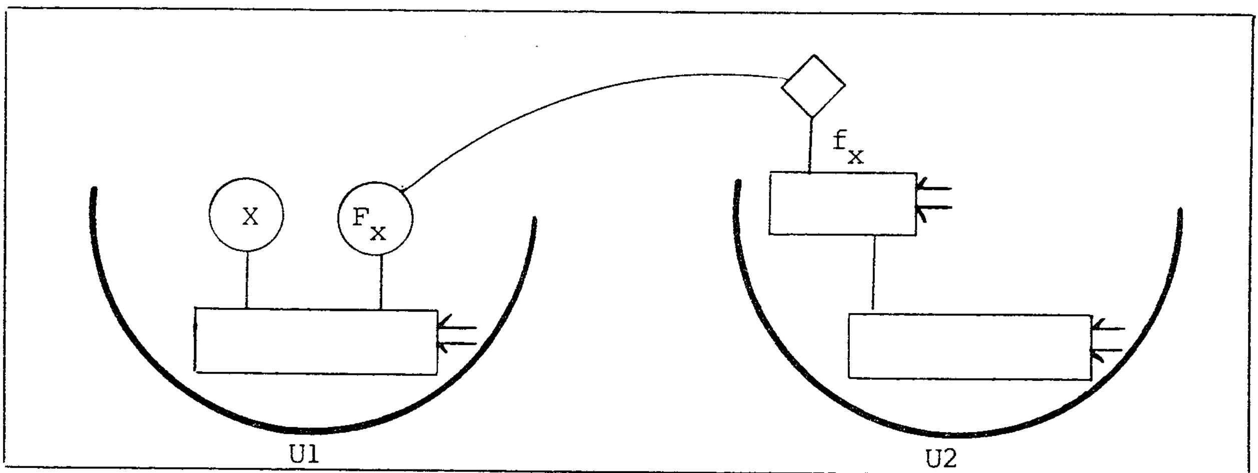


Figure 15 - Méfiance réciproque.

La seule question qui reste en suspend est la définition du lien f_x vers F_x. Ce lien ne peut être défini que depuis un environnement qui englobe U1 et U2, il sera généralement fourni directement au niveau de U2, ce qui imposera à U2 de définir un second lien vers f_x soit f'_x (cf. figure 16) afin que U_i soit l'environnement d'exécution si F_x ne s'exécute pas dans U_1. (méfiance de U2 vis à vis de U1).

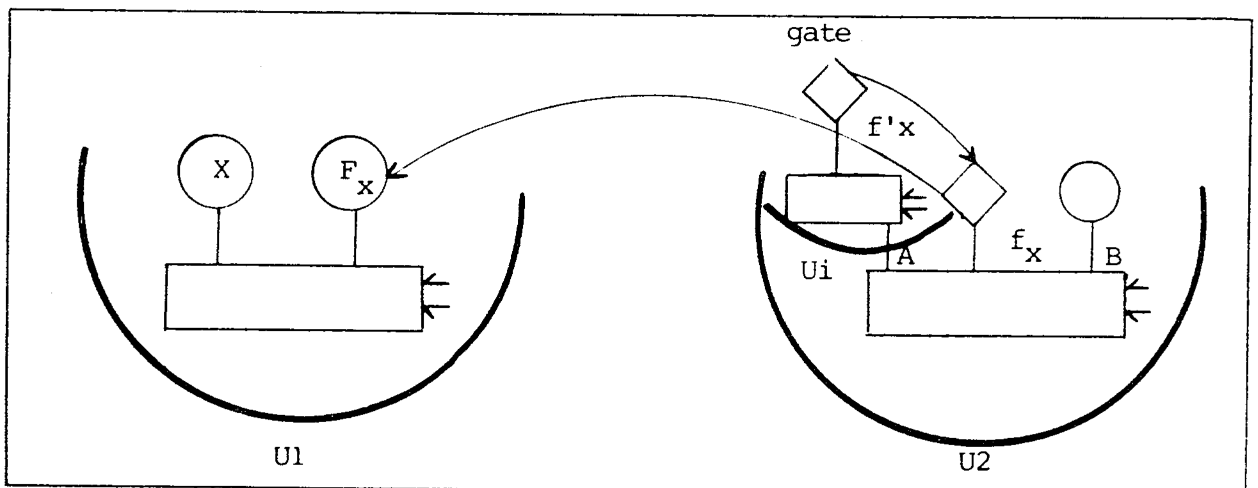


Figure 16 -

Considérons un processus P situé dans l'environnement U2, l'appel par ce processus à la fonction F_x par CALL (A. f'_x, B) donne comme état de la pile d'exécution l'une des deux possibilités de la figure 17. Dans le cas a) F_x s'exécute dans l'environnement U1, et donc toute action de F_x ne peut que modifier B ou des objets situés dans U1. Tous les autres objets de U2 sont inaccessibles.

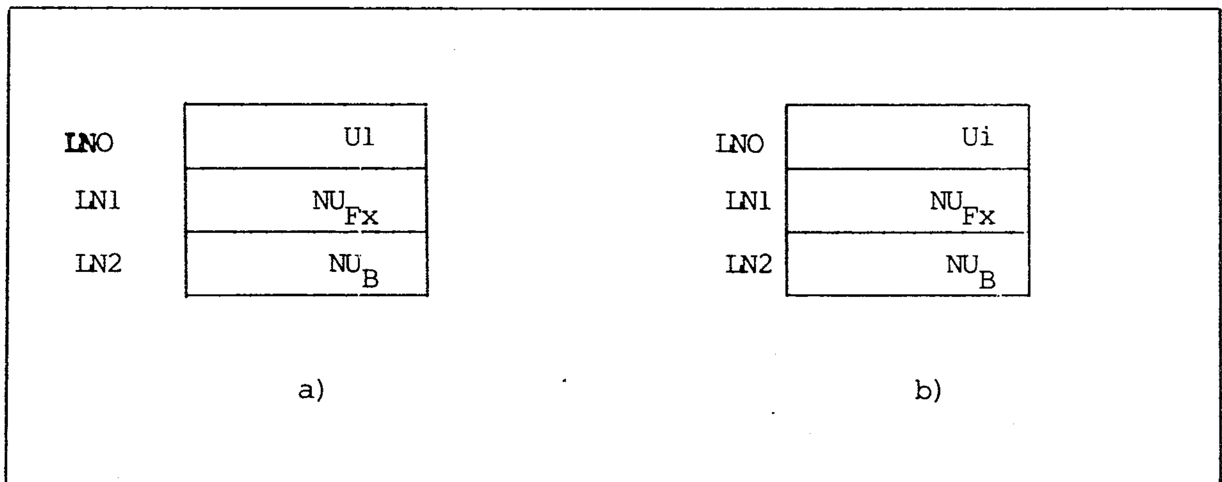


Figure 17 -

En fait, même lors de l'appel à F_x, U2 qui a un contrôle absolu sur les paramètres d'appel peut toujours restreindre les fonctions applicables sur le paramètre LN2 de l'appel ; U2 est donc complètement protégé vis à vis de F_x.

Le cas b de la figure 17 représente l'état de la pile quand U1 fournit F_x comme une procédure "bibliothèque" et non comme un point d'entrée dans U1. Dans ce cas, l'environnement d'exécution est U_i. Les seuls objets accessibles par F_x sont alors LN2 (B) et ceux de l'environnement U_i, U2 est donc encore parfaitement protégé des actions de F_x. Une troisième possibilité aurait pu avoir lieu, c'est le cas où F_x n'est pas un point d'entrée dans U1, mais où la chaîne des liens passe par un environnement. U2 est cependant toujours parfaitement protégé car cet environnement est extérieur à U2 (cf. figure 18).

Nous avons vu en détail comment on pouvait décrire la méfiance mutuelle dans un espace des objets cependant le mécanisme doit essentiellement fonctionner lors de l'accès réel, donc en utilisant l'espace d'exécution.

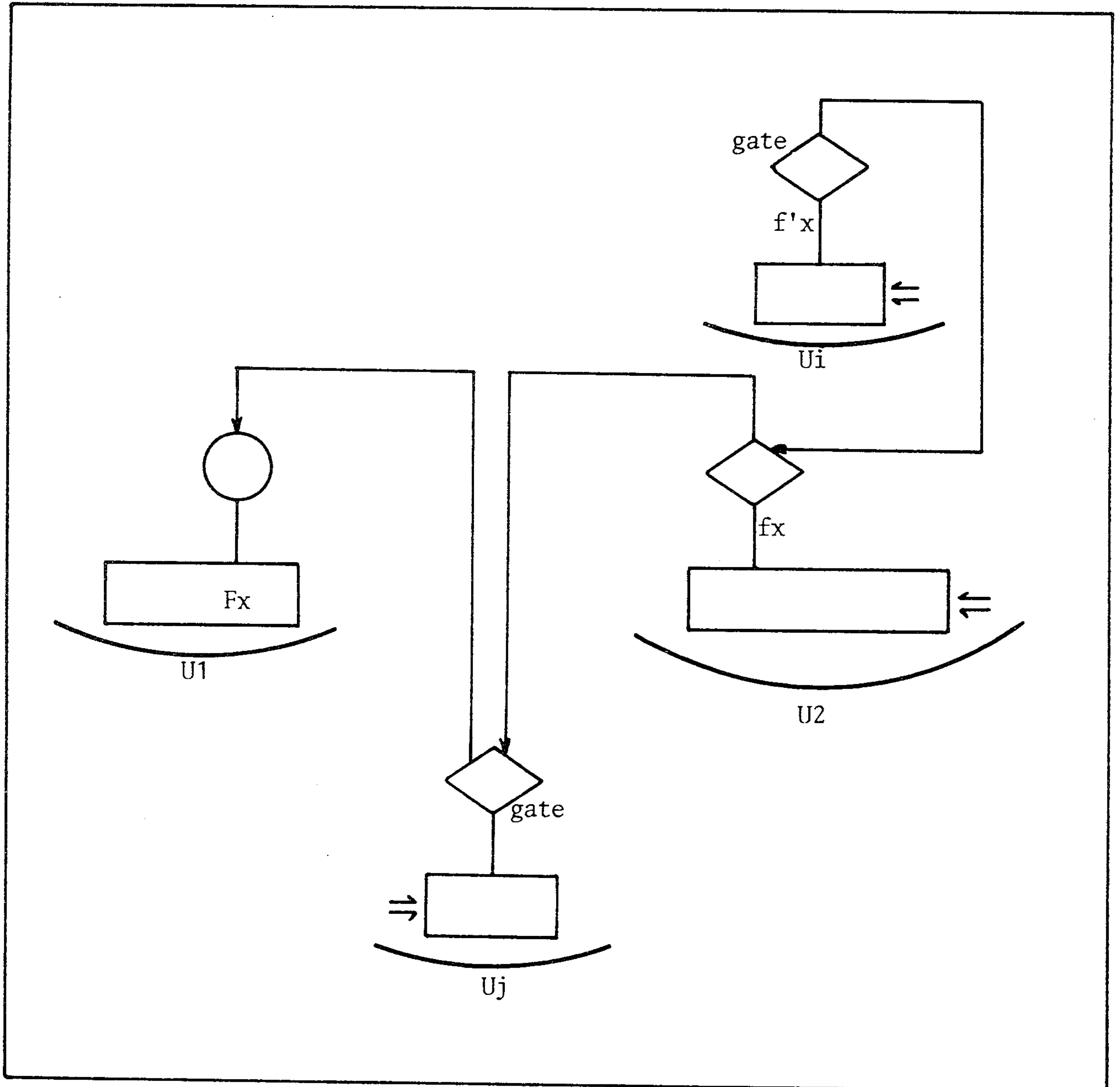


Figure 18

Il faut cependant remarquer qu'à cause de la structure hiérarchique il n'y a pas de méfiance vis à vis d'un environnement englobant.

6.2. Cas des autres systèmes à noms universels.

La méfiance réciproque ne peut pas être réalisée dans les systèmes tels que MULTICS, car il n'y a pas dans ce système de domaines disjoints à cause du mécanisme des anneaux. Il n'est en effet pas possible de définir U1 dans un domaine d'anneau i et U2 dans j, car si :

i < j	U1 peut accéder tout objet de U2
i > j	U2 peut accéder tout objet de U1
i = j	U1 et U2 peuvent accéder mutuellement les objets appartenant à l'autre.

De même le mécanisme des listes d'accès ne permet pas non plus de résoudre ce problème.

Dans le cas de HYDRA [WU 74] ce problème peut être résolu,cependant il n'est pas possible de restreindre les opérations applicables sur un objet passé en paramètre car HYDRA définit, associé à chaque procédure, une description des paramètres et une liste des protections à utiliser lors de l'exécution de la procédure. Dans ce cas, il n'est pas possible de protéger B. Cependant ceci n'est pas très gênant car il suffit à U2 de passer un objet intermédiaire et ensuite d'effectuer lui-même le transfert(après le retour de F_x) de cet objet intermédiaire vers B après vérification éventuelle de la validité de l'information fournie. D'ailleurs il faudra faire de même dans GEMAU si F_x est sensée écrire dans IN2.

Des études plus récentes (Ferrié [FE 74]) permettent de résoudre le problème de la méfiance mutuelle en utilisant toutes des mécanismes de protection à descripteurs.

6.3. Espaces récursifs.

Le cas des machines récursives est plus complexe car il n'y a de changements de domaine que d'une façon hiérarchique. Il n'est pas possible de définir un domaine associé à une procédure. Cependant on peut imaginer une solution à ce problème.

La figure 19 représente comment pourrait être réalisée la méfiance mutuelle entre U1 et U2. Uo est le système englobant U1 et U2. L'activation d'une procédure dans le domaine U1 est effectuée par :

appeler(U1)

où U1 représente l'adresse (en termes de Uo) de la table décrivant le domaine U1. Le nom local 2 de ce domaine représente F_x. L'accès à F_x avec le paramètre B est fait par U1 à l'aide de

appeler(U11)

où U11 est l'adresse (en terme de U1) de la table décrivant un domaine composé de F_x et B. (i.e des noms locaux 2 et 3).

Cependant l'exécution de cette instruction fournit une faute car F_x est en tout accès interdit, et le contrôle est repris par Uo. Uo analyse la cause de l'erreur, fabrique un nouveau domaine U2 avec F_x, X et B. Le dernier paramètre étant pris dans U11 (dont l'adresse est transmise par l'erreur). La figure 20 représente l'état de la machine lors de l'exécution de F_x.

Dans le cas d'une machine récursive on peut construire des mécanismes de protections mutuelles, mais ces mécanismes ne sont pas fournis par le noyau et sont donnés par une interprétation d'un sous-système par le sous-système englobant. Il ne faut pas oublier qu'une machine récursive est un noyau plus élémentaire que GEMAU c'est-à-dire qu'il réalise seulement les espaces d'exécution, et qu'il faut donc construire sur lui-même un espace d'objets.

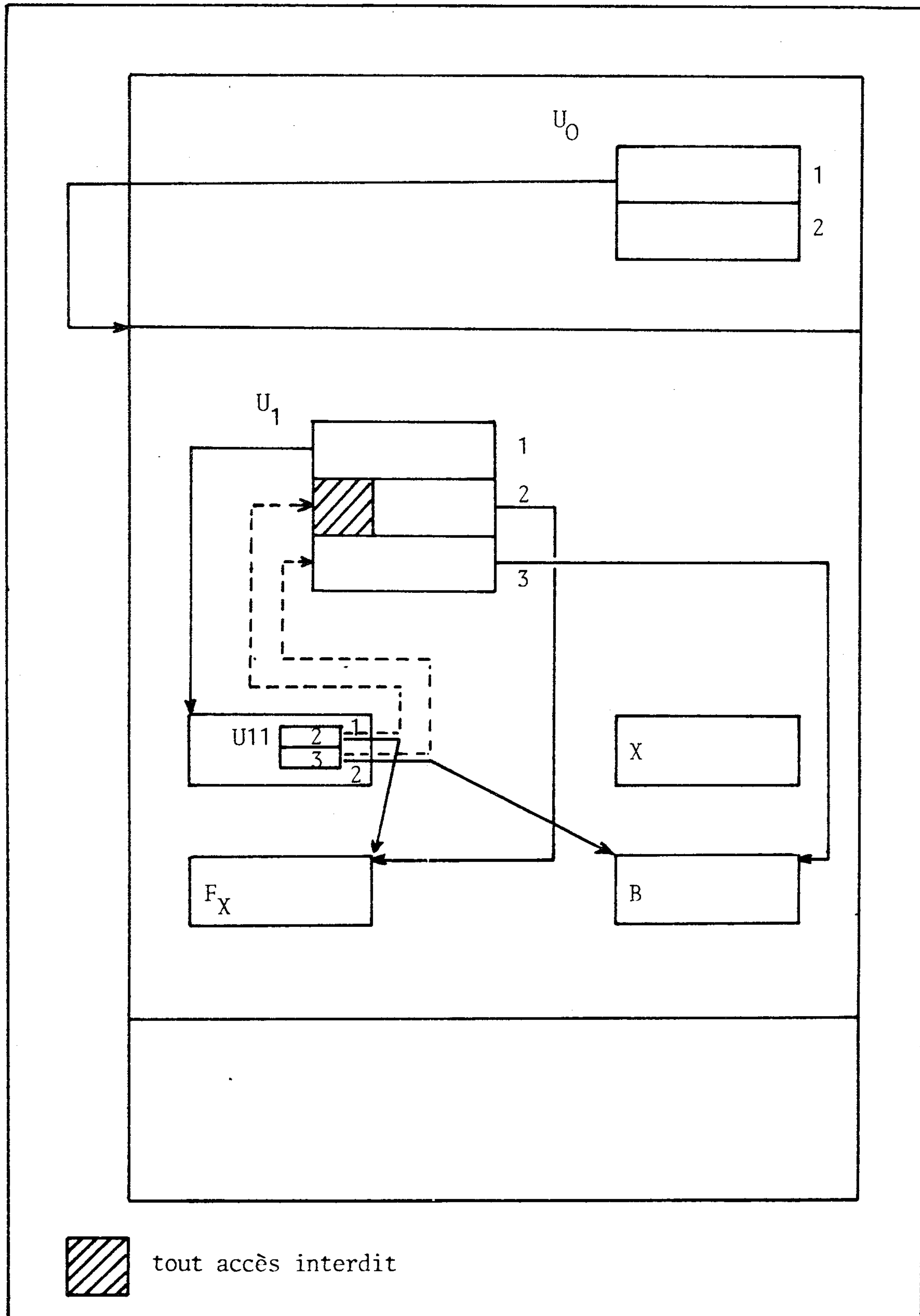


Figure 19 : Etat initial, cas des machines récursives

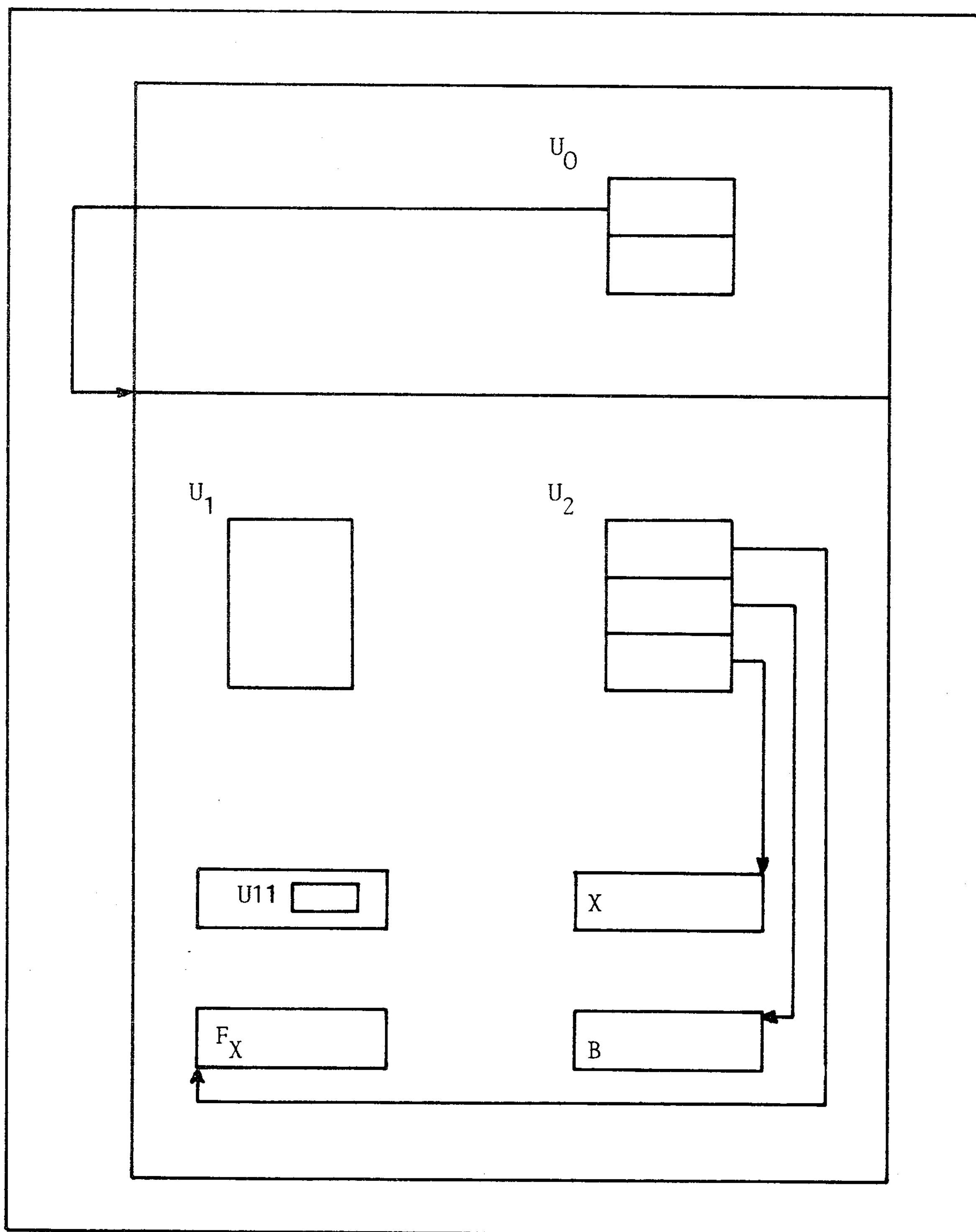


Figure 20 : Exécution de F_X

7. RESILIATION DE PROTECTION.

7.1. Contrôle de la diffusion des descripteurs.

Les divers systèmes et études que nous avons vus insistent presque tous sur l'importance des problèmes de partage et de protection ; et en particulier sur les problèmes d'accès partagé entre deux domaines disjoints. Il doit donc exister des moyens de désigner un objet d'un domaine U2 à partir d'un domaine U1: Les liens de GEMAU, les descripteurs de Fabry [FA 74] et du PP250 [WI 72]sont autant d'exemples.

Considérons le modèle de Graham et Denning, un objet possède un propriétaire qui peut donner l'accès de l'objet à un autre sujet ou retirer cet accès. A partir du moment où l'accès est autorisé un sujet peut copier les attributs d'accès dans n'importe quel autre sujet pour cet objet. Un indicateur "*copie*" est introduit pour chaque attribut d'accès pour autoriser ou non le transfert de l'attribut. Ainsi Graham et Denning [GR 72] peuvent empêcher la diffusion d'un attribut en interdisant la copie.

Dès maintenant se pose le problème de permettre la diffusion d'attributs d'accès (descripteurs) sur un objet X, et de pouvoir ensuite retirer ces accès soit à tous les sujets soit à certains d'entre eux.

Cette diffusion peut être effectuée dans les deux espaces : exécution et objets . La diffusion des descripteurs dans les espaces d'exécution a lieu chaque fois qu'il y a liaison d'un objet soit par CALL, BIND ou RETURN. La diffusion dans l'espace des objets a lieu par la création de liens, c'est-à-dire par la possibilité de fournir plusieurs noms partiels à un même objet.

Les aspects de la protection que nous allons traiter maintenant concernent *la distribution contrôlée des attributs d'accès* et *la résiliation éventuelle des attributs distribués.*

Au chapitre 2 (espace d'exécution et des objets : § 2.1) nous avons montré pourquoi il est nécessaire d'introduire des objets de type particulier : *les liens*. Ce problème est extrêmement général, en effet, la distribution des descripteurs sur un objet peut être faite de deux façons différentes : la valeur de la référence du descripteur est le nom universel

de l'objet (figure 21.a) ou le nom universel d'un autre descripteur. C'est l'introduction de la notion *d'indirection* et de *chaîne d'accès* (cf. Verjus[VE 73] Graham et Denning [GR 72]).

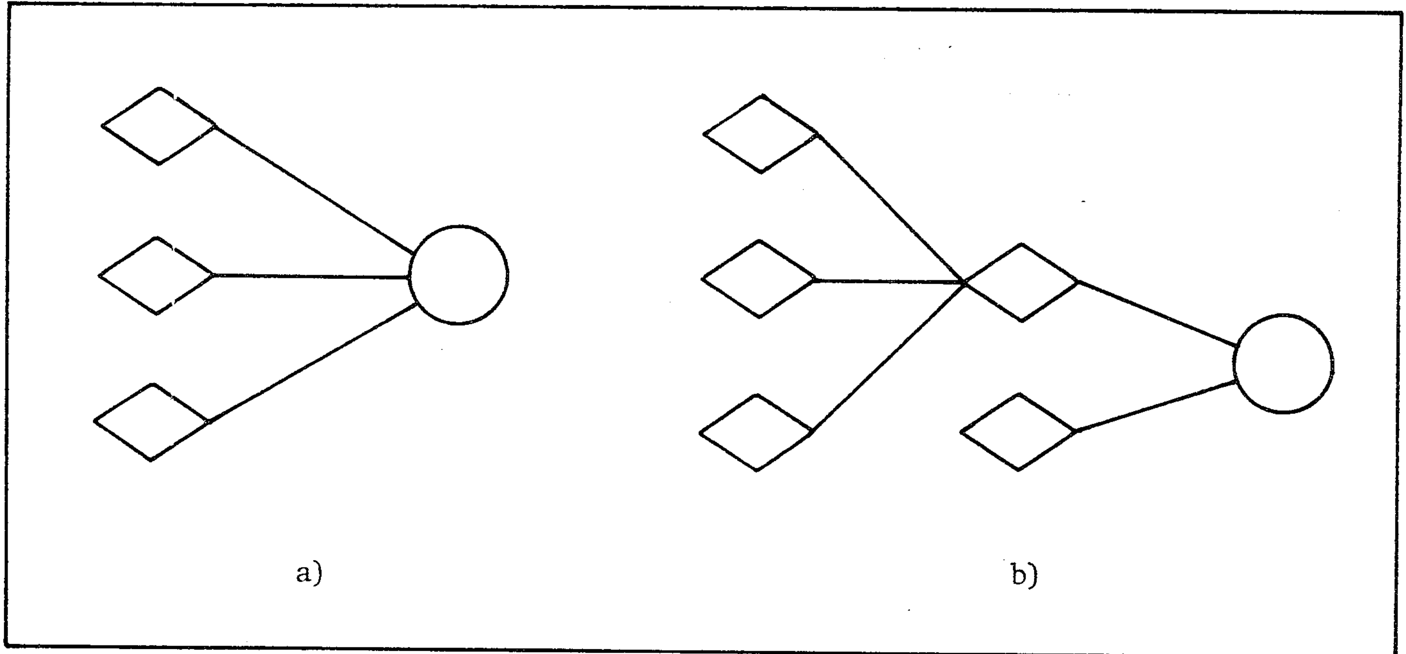


Figure 21 : Distribution des descripteurs

Ce problème est un problème très ancien qui a été traité dans les langages, en particulier très bien résolu sur les machines à descripteurs du type B3500, B6500.

Si l'on considère le cas donné par la figure 21.a, les problèmes qui se posent sont :

a) arrêt de la diffusion des descripteurs
b) résiliation de certains descripteurs.

Le point a) est résolu par Graham et Denning [GR 72] avec l'introduction de l'indicateur de copie. Par contre le point b) n'est pas résolu correctement, et la seule résiliation vraiment possible est la destruction de l'objet.

En fait, la résiliation sélective est possible à condition de conserver l'identification de chacune des distributions ; cette conservation ne peut être effectuée que par celui qui distribue. Si on considère le cas décrit dans la figure 21.b, la résiliation de l'accès s'effectue en rompant un élément dans la chaîne d'accès. Il ne faut pas interdire la diffusion des descripteurs, c'est le cas des liens dans GEMAU. Cependant comme pour a) le problème de descripteurs sur des objets détruits (objet terminal, ou lien : i.e. autre descripteur) subsiste toujours.

7.2. Propriétés d'une chaîne d'accès.

. Un objet de type lien doit être complètement *transparent* à l'utilisation. C'est-à-dire que les opérateurs applicables sur un objet doivent être utilisés de la même façon, que l'objet soit accessible directement ou au travers d'une chaîne d'accès.

. Un objet de type lien doit permettre de définir des restrictions d'accès sur l'objet terminal, c'est donc une protection *sélective* selon la chaîne d'accès.

. Une chaîne d'accès peut être créee sur tout objet accessible, y compris sur une autre chaîne d'accès. C'est la propriété *de libre distribution* des chaînes d'accès.

Les problèmes posés par les chaînes d'accès concernent l'accès aux descripteurs intermédiaires. En effet, nous avons vu que la transparence d'accès étant une propriété souhaitable, il importe de savoir comment on peut détruire ou modifier un lien. Il est donc nécessaire d'avoir des opérateurs de manipulation d'un lien. A un utilisateur donné il n'est possible d'agir que sur l'objet terminal et sur les objets intermédiaires qu'il peut nommer ; il n'y a pas d'accès possible à la chaîne d'accès en tant que telle, mais plutôt en tant que structure d'adressage (il n'y a pas d'accès au lien suivant un lien donné).

Nous avons présenté sur la figure 22 quatre environnements définissant les espaces visibles U1, U2, U3 et U4. Depuis U3, on peut agir sur les objets 2, 3, 6 et sur l'objet terminal 4, mais depuis U1 on ne peut agir que sur les objets 1 et 4 et non sur les objets 2 ou 3. Cette figure s'applique dans le cas de GEMAU.

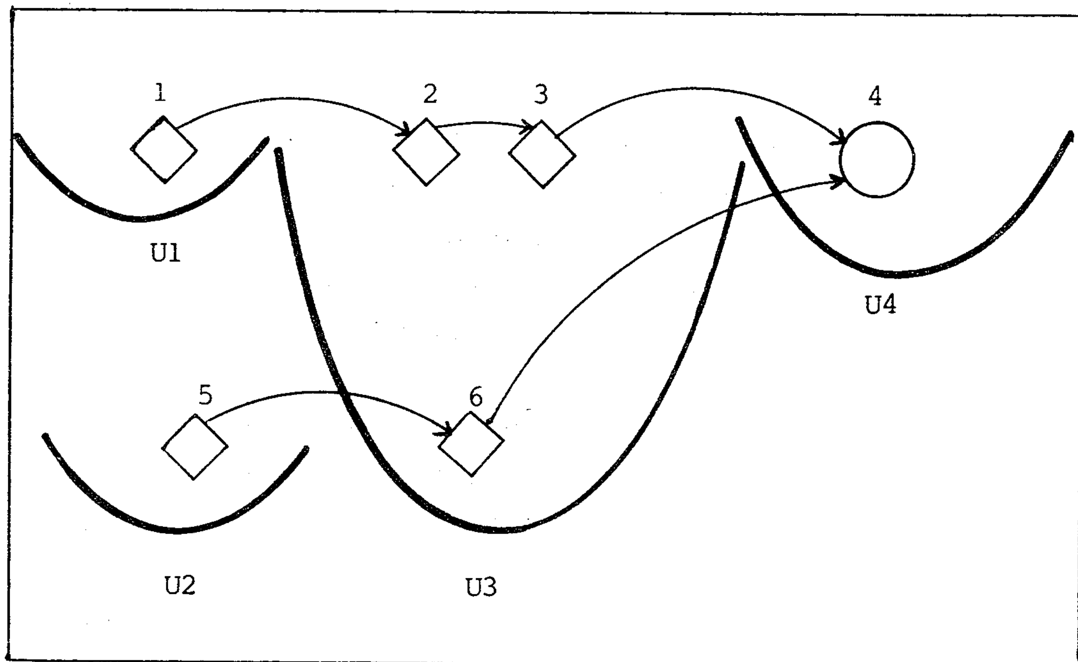


Figure 22 - Différentes chaînes d'accès

remarque : cas de MULTICS

Si nous considérons la figure 22, ces différentes chaînes (1, 2, 3, 4 et 5, 6, 4) sont possibles cependant on se heurte à un problème.

U4 a donné la permission à U3 d'accéder à l'objet 4.
U3 a donné la permission à U2 d'accéder à l'objet 6.

L'accès à l'objet 4 par U2 peut être impossible si, tout simplement, U4 n'a pas donné une autorisation d'accès à l'utilisateur U2 dans la liste d'accès de l'objet 4. Dans ce cas, i.e si on utilise les listes d'accès, si la libre diffusion des descripteurs est toujours possible, l'accès peut être impossible à partir de certains éléments de la chaîne d'accès.

Cependant le problème de la résiliation ne se pose pas car de toute façon quelle que soit la chaîne d'accès, un processus peut toujours accéder à n'importe quel objet (en le nommant).

7.3. Résiliation d'accès.

La résiliation d'une chaîne d'accès doit répondre aux propriétés suivantes :

a) avoir *un effet immédiat* pour tout nouvel accès,

b) être *sélective*, c'est-à-dire ne porter que sur une chaîne d'accès et non sur toutes les chaînes d'accès à l'objet,

c) être *cohérente*, c'est-à-dire ne pas perturber les exécutions en cours qui utilisent l'objet terminal.

L'effet immédiat est obtenu simplement par le fait que l'opération de destruction d'un lien est une opération qui porte sur l'espace des objets et est effectuée par le noyau. Toutes les opérations de modification et d'accès à l'espace des objets sont effectuées de façon exclusives les unes des autres. Ainsi, pour CALL (A, B, C.D.E, X, LN4) la liste des paramètres est évaluée en *exclusion* sur l'espace des objets Comme un lien ne peut être lié en tant que tel dans un nom local, et qu'il ne sert qu'à l'accès le fait que les opérations sur l'espace des objets soient exclusives permet de fournir la propriété c). En effet, ce que l'on détruit c'est un lien et non l'objet.

La propriété de sélectivité découle tout naturellement de la structure arborescente de l'espace des objets.

7.4. Résiliation d'accès dans GEMAU.

Nous allons montrer comment on peut réaliser la révocation dans GEMAU.

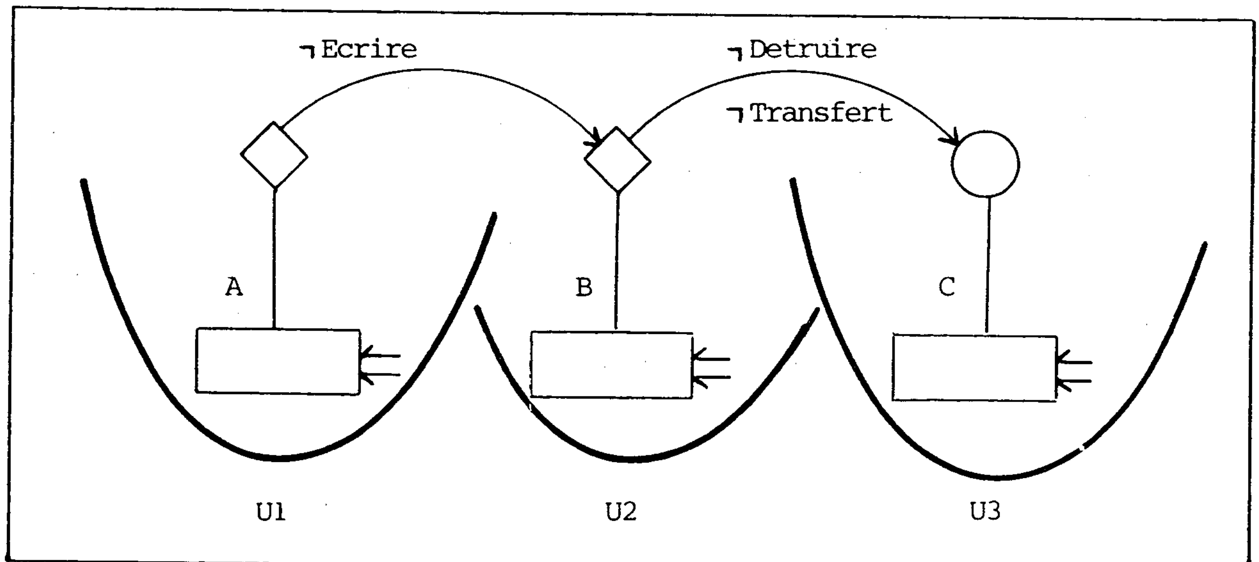


Figure 22 - Chaîne d'accès

Considérons l'exemple donné par la figure 22 : un objet C est référencé par B,lui même référencé par A. A, B et C sont dans des environnements différents.

Le lien B vers C possède une restriction de destruction et de transfert et le lien A vers B une restriction d'écriture. L'accès à C par U1 est interdit en écriture, destruction et transfert, l'accès par U2 est interdit en destruction et transfert.

L'interdiction de TRANSFER sur le lien B permet d'empêcher le déplacement de C dans l'arborescence.

U2 peut résilier son accès à C, et par la même occasion celui de U1 à C par

DELETELINK (B) ;

La modification des privilèges peut se faire par

MODIFYLINK(B,...) ;

Si l'on veut interdire la résiliation de l'accès par U2, il suffit de définir le lien B en destruction interdite.

Si U2 veut conserver l'accès et interdire celui de U1 il y a deux solutions :

a) addition d'un lien supplémentaire dans B (figure 23)
b) destruction du lien A de U1 (figure 23)

a) il suffit à U2 d'effectuer

DELETELINK (D)

ou

MODIFYLINK (D,...)

pour changer les droits d'accès de U1 à C. sans effet sur les droits de U2 sur C.

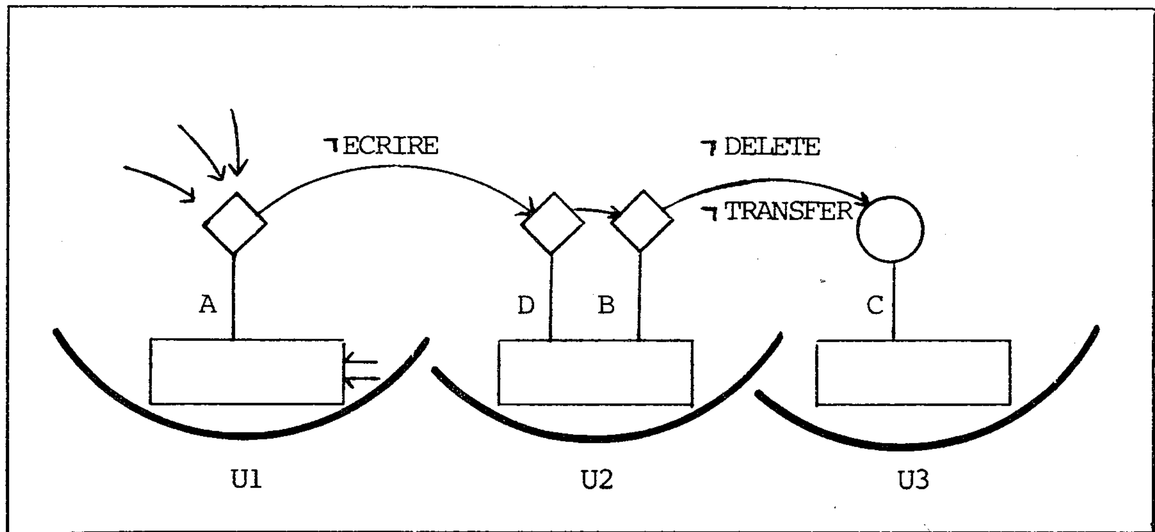


Figure 23 - Résiliation par double lien

b) il faut que U2 puisse voir le lien A de U1 par inclusion d'environnement.

La destruction du lien A dans U1 supprime la chaîne d'accès de U1 à C dans U3, même si les liens avaient été diffusés dans U1 sur A.

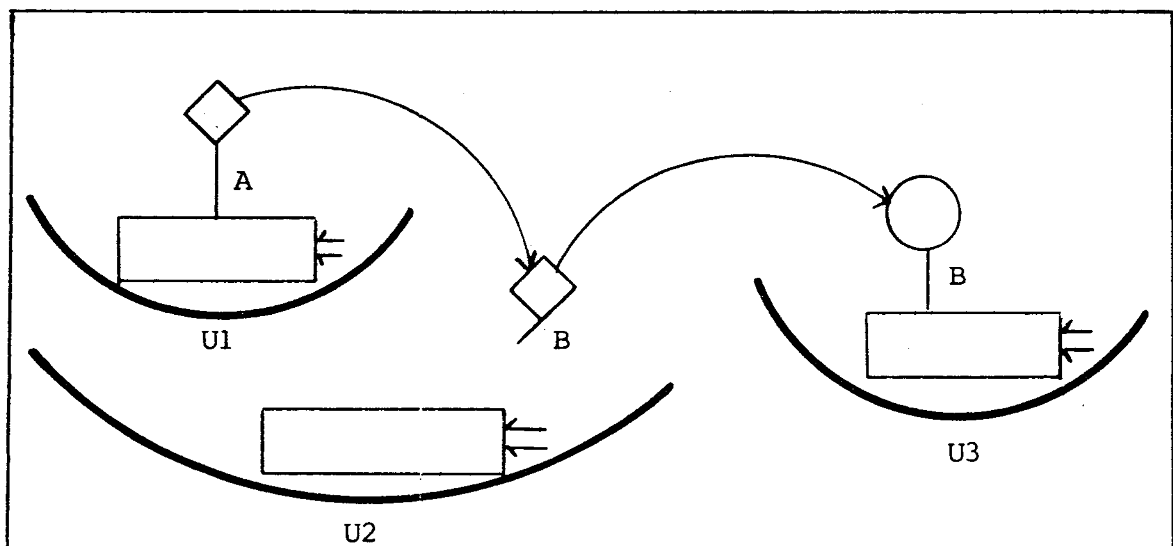


Figure 24 : Résiliation de liens inclus

7.5. Propriétés supplémentaires de la résiliation dans GEMAU.

La résiliation n'a pas besoin d'être effectuée par une instruction spéciale, il suffit de posséder un maillon de la chaîne (i.e de pouvoir le nommer), il n'y a pas besoin d'un descripteur spécial permettant cette résiliation.

Cependant Fabry [RE 74] interdit la résiliation du droit de résilier ; dans GEMAU, ce droit est toujours résiliable, il suffit de définir le lien lui-même en interdiction de destruction. Aussi, si dans l'exemple de la figure 23, le lien B (pas l'objet référencé), est en destruction interdite, U2 ne pourra jamais détruire ce lien et donc rompre la chaîne.

8. SUBSTITUTION, LIENS NON RESOLUS.

Nous avons vu dans le chapitre sur l'espace d'adressage que les liens étaient des indirections (§ 11.1) et comportaient une partie non résolue (§ 11.3) pour permettre la substitution. La valeur de la référence d'un lien est :

<nom universel> [, <nom partiel>]

Nous allons maintenant analyser les liens non résolus d'un point de vue de protection, et en particulier, en liaison avec la résiliation de chaîne d'accès.

Considérons l'exemple donné par la figure 25

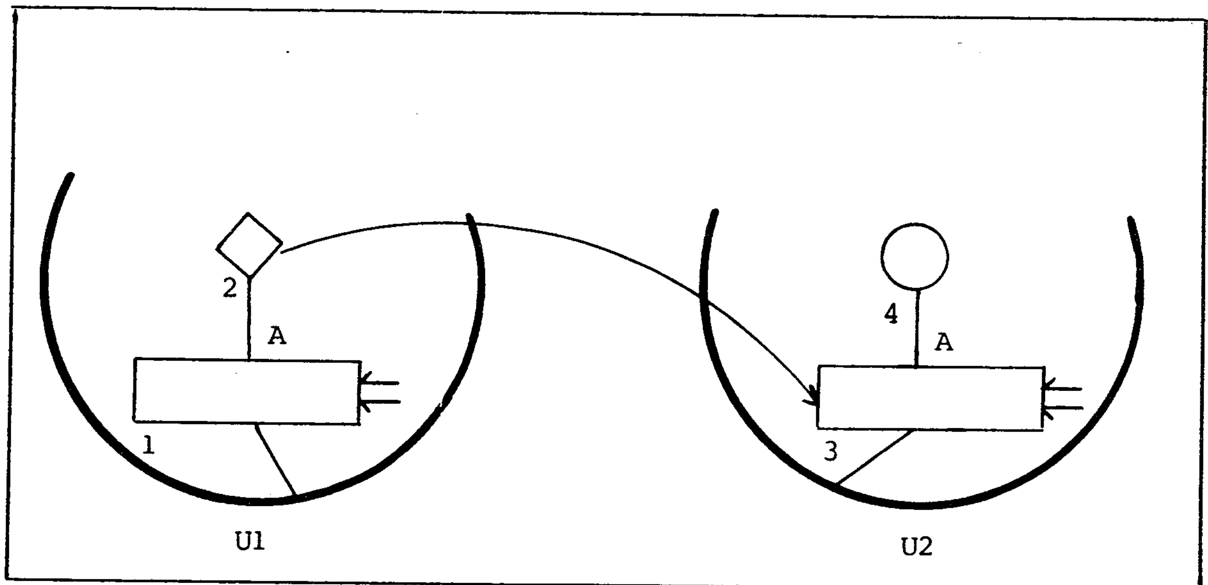


Figure 25 - Lien sur annuaire

où dans l'environnement U1, un lien A référence l'environnement U2, de façon résolue. Considérons un processus dans U1, tel que

CREATE (LNO, 'B', A.A)

crée un lien B dans U1 vers A.A, i.e. l'objet 4. Dans ce cas on a pu créer un lien vers l'objet 4 qui ne passe plus par la chaîne d'accès 2 (figure 26)

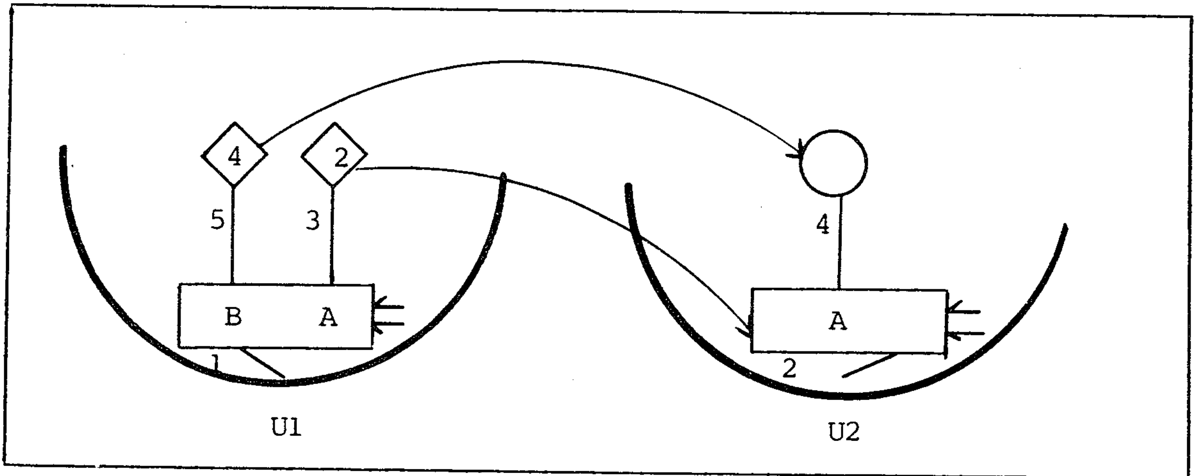


Figure 20 - Liaison hors chaîne d'accès

La solution consiste à utiliser les restrictions d'adressage en interdisant la copie de descripteur (¬ C) de l'objet 4, soit directement associé au nom d'entrée A, soit dans la valeur du lien 3. Dans ce dernier cas, il est impossible de créer dans U1 un lien résolu vers 4. Par contre la création de

CREATE (LNO, 'B', (A, 'A'))

marche (son résultat est donné par la figure 27),

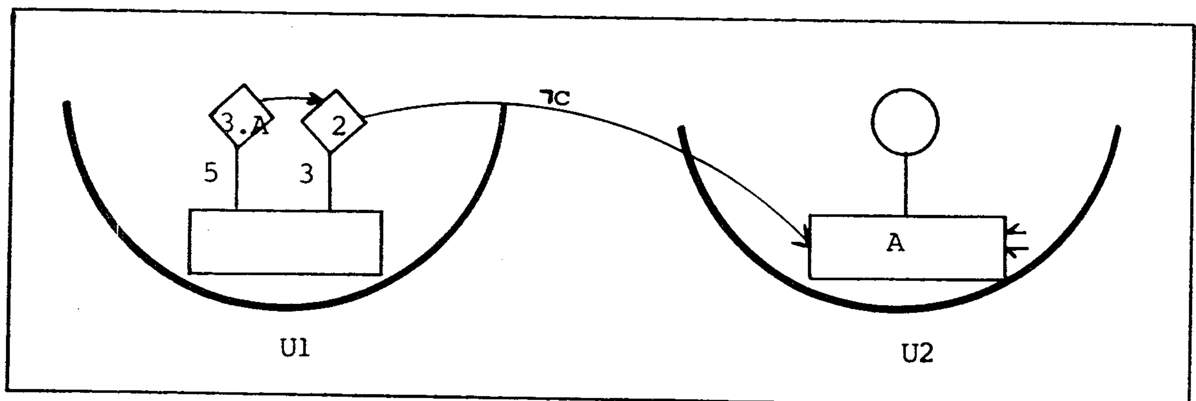


Figure 27 - Utilisation de liens non résolus.

Car il est alors possible de créer des liens dont la partie résolue soit le lien 3, aucune vérification supplémentaire (i.e au delà du lien 3) n'étant effectuée sur la valeur du lien 3.

9. CONCLUSION

9.1. Récapitulatif des propriétés de protection.

Nous avons défini en divers endroits de ce chapitre les propriétés que nous estimions qu'un système de protection doit posséder. Nous allons maintenant résumer ces propriétés pour GEMAU.

a) point de vue de l'utilisateur.

- libre contrôle des mécanismes : oui
- garantie d'utilisation : oui fournie par l'existence des 2 espaces
- transparence des mécanismes : oui
- impossibilité à contrefaire : oui

b) point de vue du système.

- possibilité d'isolation : oui
- sélectivité : oui
- méfiance réciproque : oui
- hiérarchie stricte : non

9.2. Commentaires

Le système de protection fourni par GEMAU procure des outils pour permettre la réalisation des différents mécanismes demandés par les utilisateurs et existant dans les systèmes actuels.

Mais de plus, il prend en compte des aspects pour lesquels des solutions ne sont apparues que récemment, ce qui est le cas de la résiliation ou de la méfiance réciproque. Il prend aussi en compte des aspects qui sont peu ou pas abordés à l'heure actuelle : c'est le cas du mécanisme de substitution intégrée à la protection.

La puissance de ce mécanisme est dûe pour une grande part à l'intégration dans le même système de protection des espaces d'exécution et des objets. Seul MULTICS ayant réalisée une telle solution à ce jour, cependant, le grand avantage de GEMAU réside dans le choix de l'adressage dans une structure arborescente (depuis un noeud vers une branche) qui utilise au maximum les propriétés de l'arborescence pour la protection ; et en particulier, il n'est plus nécessaire d'ajouter un mécanisme d'anneaux.

Lauer [LA 74] utilise aussi entièrement les propriétés de l'arborescence, mais d'une façon extrêmement stricte, alors que la structure de GEMAU est en fait un graphe orienté pour permettre de traverser les niveaux (sous-systèmes).

De plus, l'existence des deux espaces permet de résoudre des problèmes tels que la destruction et la substitution dynamiques qui n'affectent que les objets non liés, les objets liés continuant à être utilisés avec leur ancienne valeur.

Dans les systèmes où l'intégration des espaces d'exécution et des objets n'est pas réalisée, il faut construire l'espace des objets à partir de l'espace d'exécution, d'où très souvent l'aspect rudimentaire de cet espace ce qui se traduit soit par une perte de protection, soit par l'introduction d'un mécanisme surimposé de protection. A ce propos, on peut consulter les F-espaces de Hans [HA 73] et le système PP250 [EN 72].

Les mécanismes généraux de protection et d'adressage du langage ne s'appliquent plus dans ce cas-là. La perte de protection n'est pas trop grave, car si l'on peut accéder n'importe quel livre, le mécanisme de l'espace d'exécution permet de limiter la propagation des erreurs. Cependant, il n'est pas possible d'assurer simplement le secret de ces objets, parce que le mécanisme d'environnement n'est pas intégré à l'espace des objets.

Dans le cas du système PP250, il faut pouvoir fabriquer des descripteurs ("capabilities") pour permettre la fabrication d'un espace des objets. Il est donc nécessaire d'ajouter des instructions privilégiées ce qui est une introduction d'un mécanisme supplémentaire à celui de l'adressage : introduction d'un mode "maître/esclave".

QUATRIEME PARTIE

CONCLUSIONS, EXTENSIONS

Chapitre 6

CONCLUSIONS

Après avoir rappelé les notions de base, puis les aspects non traités dans cet ouvrage, nous montrons les domaines abordés dans GEMAU.

Nous essayons ensuite de définir les limites de l'étude et d'indiquer quelles pourraient être les directions futures de recherche : d'une part à moyen terme (extensions à GEMAU) et d'autre part à long terme (nouvelles études sur les systèmes).

Nous terminons ensuite sur une synthèse des résultats.

1. DOMAINES ABORDES PAR L'ETUDE GEMAU

1.1. Notions importantes

Tout au long de cet ouvrage nous avons introduit un certain nombre de points qui nous semblent fondamentaux et en particulier :

. l'existence intégrée de deux espaces (d'objets et d'exécution),

. les notions de porte et d'environnement dans l'espace des objets,

. les notions de liens (résolus et non résolus).

Nous allons examiner successivement chacun de ces points.

1.1.1. Existence de deux espaces de désignation

Dans GEMAU les deux espaces (d'exécution et d'objets) ont été construits de façon à ce que chacun d'eux utilise les propriétés de l'autre. Ainsi, l'espace d'exécution a une structure naturellement évolutive (aspect procédural) et l'espace d'objets, grâce à sa structure arborescente, définit la notion des espaces visibles.

Les problèmes d'adressage, de partage et de protection sont résolus de façon uniforme par ces deux espaces et la définition précise des relations entre les deux. Les mécanismes de liaison entre ces deux espaces sont peu nombreux et ne définissent aucune politique d'emploi (cf. Chapitre 4 § 3.).

De même, le choix qui consiste à regrouper les descripteurs dans des objets particuliers (annuaires) est très satisfaisant par rapport aux solutions où les objets peuvent contenir un mélange de descripteurs et de données (cas proposé par Fabry). Dans ce dernier cas, il n'est pas possible de transférer depuis (ou vers) le monde extérieur de tels objets sous peine de perdre soit de l'information, soit certaines propriétés de protection (cf. chapitre 4, § 2.7.).

Du point de vue de la protection, on peut dire que l'espace des objets permet de définir la protection réelle appliquée à un objet en fonction du chemin d'accès.

De façon générale, on peut dire que les systèmes qui n'abordent pas simultanément les espaces d'exécution et d'objets nécessitent l'introduction de mécanismes de protection supplémentaires et surimposés : anneaux, clés, listes d'accès, ..

1.1.2. Notion de porte et d'environnement

La structure arborescente découle de l'axiome initial, à savoir que l'utilisateur ne peut accéder qu'aux objets qu'il peut nommer et réciproquement ne nommer que ceux auxquels il peut accéder. Cette structure impose la notion d'environnement qui permet d'enfermer un utilisateur dans un domaine défini d'objets. Un utilisateur peut changer dynamiquement d'environnement, mais uniquement de façon contrôlée, c'est-à-dire chaque fois qu'il franchit une porte (cf. chapitre 2, § 10).

1.1.3. Notion de lien

La structure arborescente de l'espace des objets traduit les relations d'appartenance des objets à un annuaire et la relation de visibilité (environnement). Cependant, cette structure arborescente traduit une hiérarchie stricte entre les objets ; le concept de lien (cf. chapitre 2, § 11), permet de définir des relations d'accessibilité entre des environnements disjoints. On peut donc partager ainsi des objets.

La notion de lien utilisée conjointement avec le mécanisme des portes permet le transfert de contrôle entre un environnement et un autre et ce de façon éventuellement non hiérarchisée.

On peut aussi résoudre simplement le problème de la méfiance mutuelle entre deux environnements disjoints par utilisation des liens et des portes (cf. chapitre 5, § 6). La méfiance réciproque n'est possible que pour des environnements disjoints et ne joue pas s'il y a une relation d'appartenance (englobant, englobé) entre les deux environnements. C'est donc le rôle du concepteur, qui définit l'espace des objets, que de choisir le type de relations désiré : les propriétés finales d'un objet étant différentes selon sa position dans l'espace des objets.

La notion de lien non résolu liée à celle de porte est importante, car on peut ainsi définir de façon très précise l'environnement d'exécution d'une procédure, environnement qui varie selon la valeur des liens. Par le biais des liens non résolus, on peut trouver une solution au difficile problème de la destruction des objets ou de leur substitution "en ligne" (cf. chapitre 2 § 11.7. et chapitre 5 § 8). Cette propriété de substitution qui peut paraître académique s'est avérée extrêmement utile dans l'utilisation pratique du noyau et la construction de sous-systèmes [LA 75].

1.2. Autres aspects de GEMAU

Nous allons essayer de décrire brièvement ici les aspects, autres que ceux de l'adressage et de la protection, abordés dans le cadre de l'étude GEMAU.

1.2.1. Gestion des processus

La création dynamique des processus est intégrée à la structure de l'espace des objets, de même que la notion de point de contrôle (i.e. d'événement). Cet aspect de gestion des activités parallèles et du contrôle sont décrits dans [BR 74] et dans la documentation GEMAU (cf. Annexe 2 : B.2, C.12 et C.16).

1.2.2. Technologie du logiciel : réalisation du noyau sur 10070

L'objet manipulé à l'intérieur du noyau est un module. Nous avons voulu qu'à la conception la connaissance précise des interfaces (équivalents aux opérateurs associés à un mode) entre les différents modules (procédures et paramètres), soit définie avant toute programmation. En plus de ces interfaces, les relations entre les objets sont définies de façon précise, mais avec une structure différente de celle de l'espace des objets. En effet, les propriétés recherchées et les contraintes imposées ne sont pas forcément identiques à celles que nous avons pu définir et voir tout au long de cet ouvrage.

Une forme de pensée similaire à celle développée ici a été utilisée pour réaliser un noyau de GEMAU sur CII 10070 [DA 74 a].

Nous avons donc défini un "noyau" du noyau de GEMAU, qui permette de définir les modules, les interfaces et les structures nécessaires.

Un module est un ensemble de données associées aux procédures d'accès à ces données. Un module ne peut jamais accéder lui-même à une information contenue dans un autre module. Il doit toujours appeler la procédure d'accès correspondante de cet autre module. En conséquence :

a. une fois fixée la sémantique de ce module, c'est-à-dire l'ensemble des procédures d'accès, la réalisation de ce module peut être laissée aux soins d'un réalisateur particulier ;

b. tout module peut être substitué par un module équivalent ayant des performances améliorées ;

c. tout ensemble de données partagées dans le noyau se trouve réuni en un seul module ;

d. un code en section critique est toujours défini au niveau du module sur l'ensemble des procédures d'accès du module (sémaphore, niveau d'interruption, adressage réel ou virtuel pour la réentrance du code). Il n'appartient donc pas au programme du module de manipuler ces notions.

Chaque module constitue à lui seul un "environnement" : en effet, il est plongé dans un ensemble d'autres modules qui sont les "objets visibles" lorsqu'on exécute le module. Ainsi, dans le noyau, chaque application d'une fonction (d'accès) sur un module fait changer le processus d'espace visible (cf. figure 1).

L'espace de désignation du noyau est donc constitué d'un assemblage statique d'objets. Cet espace forme un graphe sans cycle afin de faciliter la compréhension de l'ensemble du fonctionnement du noyau (ce procédé n'est pas sans analogie avec les méthodes de la programmation structurée [DI 68].

La figure 2 schématise les différentes couches qui composent le prototype GEMAU tel qu'il a été réalisé sur CII 10070. Dans cette figure ce que nous appelons 'extension hardware' est la réalisation du noyau de noyau. Ce prototype est entièrement décrit dans la documentation GEMAU (cf. Annexe 2 : C.1 à C.16).

1.3. Utilité pratique d'un tel noyau

Ayant défini ce noyau, il faut maintenant déterminer si la structure et les notions mises en place permettent effectivement d'atteindre les objectifs plus généraux définis à l'origine : à savoir la construction de systèmes d'exploitation.

Pour étudier ce point nous nous basons sur l'expérience acquise dans la réalisation de sous-systèmes sur le prototype 10070. Cette expérience comprend la conception et l'écriture de :

. deux sous-systèmes "utilisateurs" : conversationnel et traitement par lots (cf. Annexe 2 : E.2, E.4, et [LA 75]) ;

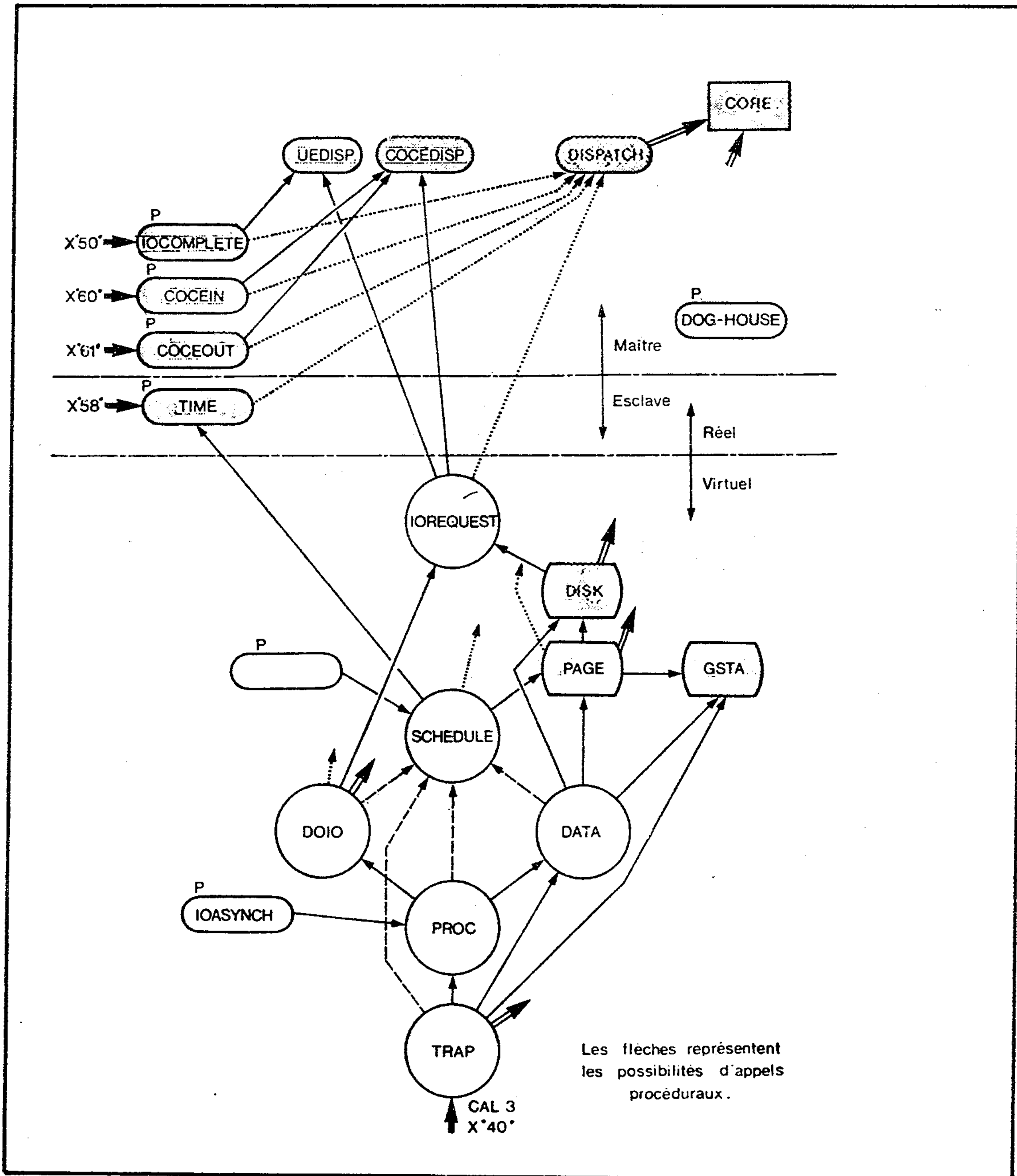


Figure 1 : Noyau de GEMAU

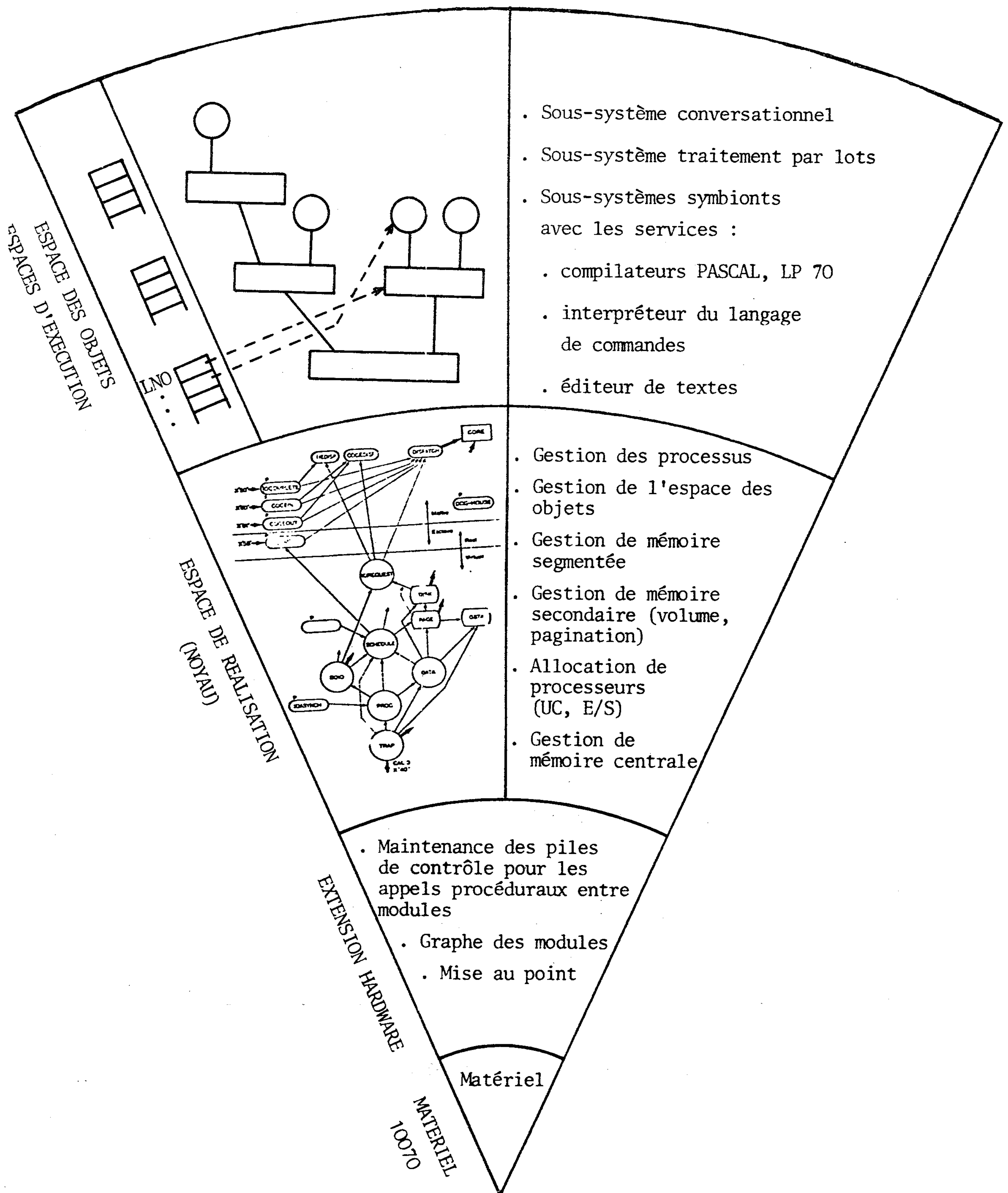


Figure 2 : Différentes couches de réalisation

. trois sous-systèmes de gestion d'entrées-sorties : symbionts lecteur de cartes, imprimante et console opérateur ;

. un interpréteur de langage de commandes (cf. Annexe 2 : E.1, E.3) ;

. le transport de deux compilateurs : PASCAL et LP 70 depuis SIRIS 7 vers GEMAU.

P. Laforgue dans sa thèse [LA 75], analyse cette expérience que nous rappelons brièvement ici :

. la structure de GEMAU n'introduit pas de contraintes spéciales pour la réalisation de sous-systèmes ; ce point est en faveur des primitives et de la structure de désignation choisie ;

. la structure hiérarchique couplée avec le concept des liens est bien adaptée à la réalisation de sous-systèmes ;

. l'aspect symbolique de l'adressage, joint à la propriété de substitution, s'est avéré efficace pour la conception, la mise au point et la maintenance des différents composants des sous-systèmes.

Les difficultés rencontrées par P. Laforgue ont permis d'avoir une influence sur la définition de GEMAU. Une autre étude effectuée par A. Inglese (Annexe 2 : E.5), montre la structure d'un sous-système de la machine virtuelle utilisant GEMAU.

2. LIMITES DE L'ETUDE, PERSPECTIVES

Les études que nous avons menées correspondent à certains objectifs et n'étudient qu'une partie des problèmes. Nous nous heurtons à deux classes de limitations : l'une due à l'étude compte tenu des objectifs, l'autre due à la définition des objectifs.

2.1. Limitations dues à GEMAU, aspects non abordés

2.1.1. Objets construits

Nous avons défini la substitution de façon à pouvoir remplacer un objet (ou ensemble d'objets) par un autre équivalent, cet objet (ou ensemble d'objets) ne pouvant être que primitif.

Il semble intéressant d'étudier un mécanisme où de nouveaux types d'objets puissent être fabriqués, par exemple un type fichier où les opérateurs applicables au lieu d'être primitifs seraient eux-mêmes des objets (notamment des procédures écrites dans un certain langage). On trouvera dans [DA 73] et [DA 74 b] une formalisation de ce concept d'objet construit.

2.1.2. Contrôle de type lors d'appels procéduraux

Le problème du contrôle a été abordé en partie quand nous avons étudié la possibilité de définir un squelette d'espace d'exécution associé à une procédure (cf. chapitre 4 § 3.5.). Ce type d'extension est souhaitable à condition de conserver la possibilité de substitution. Nous renvoyons le lecteur au paragraphe précité pour une discussion plus complète sur ce problème.

2.1.3. Synchronisation, interblocage

Le seul degré de multiprogrammation n'est pas suffisant pour définir des mécanismes de synchronisation évolués. Ainsi, le problème des "readers and writers" ne peut être résolu par GEMAU. Il n'est pas non plus possible d'effectuer des synchronisations explicites entre processus sans recourir à la création de nouveaux types primitifs d'objets, comme par exemple les sémaphores (ce qui est d'ailleurs relativement facile). Cependant, une solution plus intéressante à ce problème pourrait être trouvée dans les moniteurs de Hoare [HO 74 a].

Un autre type de problème qui n'est pas résolu par GEMAU concerne l'interblocage et plus spécialement les problèmes pouvant naître de l'utilisation des degrés de multiprogrammation. On peut dire dès maintenant qu'une partie importante des recherches futures sur les systèmes à objets devrait concerner la prévention des interblocages. L'association des fonctions de synchronisation aux objets plutôt qu'aux programmes utilisant ces objets devrait aider cette étude dans la mesure où l'on peut contrôler cette synchronisation.

2.1.4. Fiabilité, reconfiguration

Nous avons constaté que la fiabilité n'est pas seulement un problème de réalisation, mais que des outils adéquats doivent être prévus lors de la définition d'un noyau. Dans le modèle proposé ces outils sont intégrés à la définition fonctionnelle.

De plus, s'il est vrai que ces outils apportent un coût supplémentaire dans les temps d'exécution et dans la taille des espaces mémoire, la véritable question est de chiffrer la perte (incohérence) de certains objets et de comparer ce coût à celui ajouté par la recherche de fiabilité. Dans notre cas, le choix s'est clairement porté en faveur d'une plus grande fiabilité des systèmes.

Nous n'avons pas abordé ici le problème de la fiabilité d'autres éléments tels que les périphériques et les hiérarchies de mémoire :

. les objets de type périphérique sont définis à un instant donné dans certain environnement et sont utilisés par un ou plusieurs processus. La structure même de l'adressage et de la protection permet de réduire les erreurs dues à ces éléments (périphériques), car une erreur ne peut pas se propager à l'extérieur de l'espace d'exécution courant (sauf erreur dans le noyau).

. L'expérience du prototype nous a montré la fragilité des systèmes où l'on utilise des hiérarchies de mémoire pour réaliser l'espace des objets ; il faut donc étudier le problème de la fiabilité du noyau et en particulier la possibilité de fonctionnements dégradés.

Les structures de recouvrement d'erreurs selon [HO 74 b] semblent pouvoir procurer des éléments de réponse à ces problèmes de fiabilité.

2.1.5. Gestion prédictive de ressources

La connaissance du comportement de programmes stockés dans l'espace des objets permet une gestion a priori des ressources du système et en particulier de la mémoire. En effet, à un instant donné on connaît une enveloppe supérieure des besoins immédiats d'un processus (l'ensemble des objets liés à son espace d'exécution courant) et on connaît aussi les instants où cette enveloppe est susceptible de changer (BIND, FREE, CALL, RETURN) ; on peut donc associer des indications de comportement sur chaque objet et espérer ainsi lors d'appels de procédure, effectuer une gestion prédictive des ressources. On utilise alors des informations qui sont relatives à des objets et non à des processus.

Une autre possibilité qui peut être fournie par l'étude actuelle et le prototype, concerne les réponses possibles aux questions suivantes :

a. quel est le coût effectif d'un adressage symbolique et en particulier la fréquence des appels aux annuaires et l'influence des piles ?

b. quelle est la taille des différents autres composants et la fréquence de leurs accès ?

c. quelle est l'influence de la gestion prédictive de ressources ?

2.2. Limitation des objectifs initiaux, perspectives

L'étude menée dans le cadre de GEMAU comporte un certain nombre d'enseignements, mais souffre aussi de certaines limitations inhérentes d'une part aux objectifs initiaux, et d'autre part à la contrainte d'expérimentation sur du matériel standard. Nous allons considérer successivement les cas suivants :

- machines orientées vers les systèmes,
- distribution des processeurs et de l'adressage,
- langage d'écriture de systèmes.

2.2.1. Machines orientées vers les systèmes, conception de systèmes

C'est la définition de tout un ensemble de concepts et de mécanismes cablés ou logiciels nécessaires afin de mettre en oeuvre les concepts "système". Suivant cette tendance, de plus en plus de concepts sont concrétisés dans le matériel [WI 72], [WU 74], [SP 74]. Ce domaine comprend :

- la définition d'opérateurs système,
- la définition de structures de programmes au niveau de l'adressage (structure de la mémoire) et de l'intégration de mécanismes de protection et de partage,

. la définition des structures de contrôle (processus, événements) ainsi que les moyens de synchronisation.

2.2.2. Distribution

La banalisation des objets et leur substitution, doivent faciliter la réalisation de problèmes actuels entraînés par la distribution des processeurs (multiprocesseurs, réseaux d'ordinateurs, hiérarchie de mémoire, ..).

Considérons le problème des entrées-sorties. Dans GEMAU, nous avons supprimé la notion de périphérique (ou de fichier classique) dans tous les cas où cette notion correspondait à un besoin de capacité mémoire (temporaire ou permanente). Cependant les périphériques d'entrées-sorties existent en ce qui concerne la communication avec le monde extérieur et sont alors équivalents à des procédures.

Une autre solution consiste à considérer les périphériques comme des processeurs : dans ce cas on généralise la notion de programme canal et un périphérique est alors un processeur ayant une syntaxe et une sémantique dont les niveaux sont plus élevés que ceux des programmes canaux actuels. Cette solution offre l'avantage de généraliser la notion de processeur spécialisé (cf. Annexe 2 : F.4 et F.5), mais impose l'existence d'objets construits. A chaque objet on associe le nom universel du processeur qui le traite, que ce processeur soit matériel ou logiciel.

Dans la même catégorie de problème, on peut citer le cas des symbionts qui doivent pouvoir faire partie de l'espace des objets ; ce sont des processeurs spécialisés qui traitent les segments à imprimer ou à lire.

La réalisation des objets peut être effectuée à l'aide de techniques de hiérarchie de mémoire ; dans ce cas certains processeurs pourraient faire partie de cette hiérarchie, en particulier les processeurs symbionts.

D'ailleurs, dans le même ordre d'idée, on peut citer l'intégration dans un réseau d'ordinateurs qui est possible au niveau de l'objet car celui-ci pourrait être localisé sur tel ou tel site sans que l'utilisateur le sache (si ce n'est au niveau des performances éventuelles) ; il peut même correspondre à un objet situé sur un autre espace d'objets ayant une structure et des propriétés différentes de celles de GEMAU.

2.2.3. Outils d'écriture de systèmes

Au cours de cette étude nous avons été amené à définir une méthodologie de conception du noyau (cf. § 1.3.2.). Cependant, pour la réalisation effective, nous avons véritablement manqué d'outils efficaces d'écriture et de mise au point. En particulier, un des aspects fondamentaux à étudier concerne la définition de langages de description et de langages d'écriture de systèmes.

3. SYNTHESE

Nous allons essayer de dégager brièvement les résultats de l'étude tant sur le plan théorique que sur le plan pratique.

3.1. Résultats théoriques

Compte tenu des objectifs fixés et des diverses comparaisons effectuées dans cet ouvrage, nous pouvons affirmer que les problèmes rencontrés dans les systèmes peuvent être résolus par la définition d'espaces de désignation adaptés et réduire ainsi le nombre de mécanismes nécessaires.

Les concepts d'espace d'exécution et d'objets, d'environnement et de porte, ainsi que de liens, sont la base du formalisme décrit ici.

Nous avons découvert au cours de cette étude un certain nombre d'horizons nouveaux (cf. § 2.2.), il faut donc considérer les travaux décrits ici non comme un aboutissement, mais plutôt comme une image instantanée de certaines connaissances.

3.2. Résultats pratiques

Un prototype du noyau fonctionne sur CII 10070 (ou IRIS 80) au CICG depuis le début de l'année 1974 et un système conversationnel simple depuis Août de la même année.

Cette réalisation a permis d'une part de montrer qu'il est parfaitement concevable et viable d'avoir des systèmes à objets et, d'autre part, de mieux comprendre ce type de système.

Ce prototype comprend tous les concepts décrits ici, mais possède certaines restrictions dues au fait que nous voulions tester uniquement des idées. Cependant, la fabrication d'un véritable système opérationnel utilisant les mécanismes câblés de l'IRIS 80 devrait être un banc de test pour expérimenter sur une échelle réelle des systèmes à objets et d'en procurer les moyens de les évaluer.

ANNEXES

BIBLIOGRAPHIE GENERALE

BIBLIOGRAPHIE GEMAU

GLOSSAIRE

Annexe 1

BIBLIOGRAPHIE GENERALE

[AN 74 a] F. ANCEAU, D. FORTIER & J.P. SCHOELLKOPF
Conception descendante des machines : application à une machine PASCAL
Journées AFCET, Machines orientées langage, machines orientées système, 2-3 Mai 1974

[AN 74 b] F. ANCEAU
Contribution à l'étude des systèmes hiérarchisés de ressources dans l'architecture des machines informatiques
Thèse d'état, Grenoble, 5 Décembre 1974

[AU 72] A. AUROUX
Généralisation du concept d'adressage, application au système CP 67
Congrès AFCET 1972, Grenoble, pp. 211-223

[BA 67 a] R.M. BALZER
Dataless Programming
FJCC 1967, pp. 535-544

[BA 67 b] T.R. BASHKOW, A. SASSON & A. KRONFIELD
A System Design for a FORTRAN Machine
IEEE Transactions on Electronic Computer, August 1967, pp. 485-499

[BA 71] R.M. BALZER
PORTS : a Method for Dynamic Interprogram Communication and Job Control
SJCC Vol 38 (1971), pp. 485-489

[BE 70] C. BETOURNE, J. BOULENGER, J. FERRIE, C. KAISER, S. KRAKOWIAK & J. MOSSIERE
Process Management and resource sharing in the multiaccess system 'ESOPE'
C.ACM, 13,12, December 1970, pp. 727-733

[BE 72] A. BENSOUSSAN, C.T. CLINGEN & R.C. DALEY
The MULTICS Virtual Memory : Concepts and Design
C.ACM 15,5, May 1972, pp. 308 ; & Proceedings Second Symposium on Operating Systems, Princeton, October 1969, pp. 30-42

[BO 67] C. BOKSENBAUM & S. GUIBOUD RIBAUD
Etude et réalisation d'un système pédagogique de programmation en ligne et en temps partagé
Congrès AFIRO, Nancy, Mai 1967 et projet de fin d'étude

[BO 68] C. BOKSENBAUM & S. GUIBOUD RIBAUD
SPARTACUS : Système en temps partagé conversationnel à usage scolaire, ou la programmation en mode esclave
Colloque Université de Grenoble, 1968, Monographies d'informatique n° 6 (AFCET), Dunod, pp. 15-24

[BR 69] P. BRINCH HANSEN
RC4000 Software Multiprogramming System
A/S Regencentrallen, Copenhagen, 1969

[BR 73] P. BRINCH HANSEN
Concurrent programming concepts
ACM Computing Surveys, 5,4, December 1973

[BR 74] J. BRIAT & S. GUIBOUD RIBAUD
Espace d'adressage et espace d'exécution du système GEMAU
International Symposium on Operating Systems Theory and Practice, Paris, April 23-25 1974, pp. 131-167

[CL 69] J.C. CLEARY
Process Handling on Burroughs 6500
Proceedings Fourth Australian Computer Conference, Adelaide 1969

[CO 72] F.J. CORBATO, J.H. SALTZER & C.T. CLINGEN
MULTICS, The first seven years
Proceedings SJCC 1972, pp. 571-583

[CR 75] CROCUS, ouvrage collectif
Systèmes d'exploitation des ordinateurs, principes de conception
Dunod, Collection Université

[DA 65] R.C. DALEY & P.G. NEWMANN
A general-purpose file system for secondary storage
FJCC 1965, pp. 213-229

[DA 68] R.C. DALEY & J.B. DENNIS
Virtual memory, processes & sharing in MULTICS
C.ACM, 11,5, May 1968, pp. 306-312

[DA 73] P. DARONDEAU
NAMES & MODES, Noyau d'architecture et métalangage pour l'élaboration de systèmes modulaires organisés sur la définition d'environnements spécialisés
Rapport ENSIMAG, Juillet 1973

[DA 74 a] P. DARONDEAU, S. GUIBOUD RIBAUD & C. OTRAGE
Application de la structuration de système à la conception interne du système GEMAU
Colloque sur la Programmation, Paris, 9-11 Avril 1974, Springer Verlag 9, pp. 280-297

[DA 74 b] P. DARONDEAU, S. GUIBOUD RIBAUD & C. OTRAGE
A formal model for system-oriented architecture
IFIP 1974, Stockholm, August 1974, pp. 71-75

[DA 74 c] P. DARONDEAU, S. GUIBOUD RIBAUD, C. OTRAGE & H. SMIT
Architecture de calculateurs orientés vers les systèmes
RAIRO B2 - 1974, pp. 93-126 (Première partie); Deuxième et Troisième parties à paraître

[DE 65] J.B. DENNIS
Segmentation and the Design of Multiprogrammed Computer Systems
J.ACM 12,4 October 1965, pp. 589-602

[DE 66] J.B. DENNIS & E.C. VAN HORN
Programming Semantics for multiprogrammed computations
C.ACM 9,3, March 1966, pp. 143-155

[DE 70] P.J. DENNING
Virtual Memory
Computing Surveys 2,3, September 1970, pp. 153-189

[DE 71] P.J. DENNING .
Third Generation Computer Systems
Computing Surveys, vol 3, n° 4, December 1971, pp. 175-216

[DE 72] J.B. DENNIS
The design and construction of software
Advanced Course on Software Engineering, February 21 - March 4, 1972, Munich

[DI 68] E.W. DIJKSTRA
THE Multiprogramming system
C.ACM 9,3, March 1966

[EC 69] Interactive Systems "ECHO"
Documentation interne Burroughs, LCSO (Paoli) PA, 1969

[EN 72] D.M. ENGLAND
Operating system of System 250
IEEE, International Switching Symposium Record, Cambridge, Massachusetts, 6-9 June 1972, pp. 525-529

[EV 67] D.C. EVANS & J.Y. LECLERC
Address mapping and the control of access in interactive computer
AFIPS, SJCC 1967, pp. 23-30

[FA 74] R.S. FABRY
Capability-based addressing
C.ACM 17,7, July 1974, pp. 403-412

[FE 74] J. FERRIE, C. KAISER, D. LANCIAUX & B. MARTIN
An extensible structure for protected system design
International Workshop on Protection in Operating Systems, IRIA, August 13-14 1974, pp. 83-105

[FR 71] A.G. FRASER
On the meaning of names in programming systems
C.ACM 14,6, June 1971, pp. 409-417

[GA 74] U.O. GAGLIARDI
Software-related advances in computer hardware
Computer Architecture News (ACM Special Interest Group)
3,2, June 1974, pp. 16-36

[GO 72] H.A. GOODIERS
System 250, Processing philosophy
IERE, Conference on Computer Systems & Technology, London,
27-29 October 1972, pp. 169-180

[GO 73] R.P. GOLDBERG
Architecture of virtual machines
Proceedings AFIPS 1973, pp. 309-318

[GR 68] R.M. GRAHAM
Protection in an information processing utility
C.ACM, vol 11 n° 5, May 1968, pp. 365-369

[GR 72] G.S. GRAHAM & P.J. DENNING
Protection, principles and practice
AFIPS Conference Proceedings, 40, 1972, SJCC

[GU 68] S. GUIBOUD RIBAUD
SPARTACUS : Système multiconversationnel sur PDP 8
Manuel d'Emploi, Note ENSIMAG, 1968

[HA 68] E.A. HAUCK & B.A. DENT
Burroughs B 6500 / B 7500 stack mechanism
Proceedings AFIPS SJCC 1968, pp. 245-251

[HA 71] A. HASSITT, J.W. LAGESCHULTE & L.E. LYON
Implementation of a high level language machine
Proceedings of the Fourth Annual Microprogrammed Workshop,
Santa Cruz, California, September 1971

[HA 72] D. HALTON
Hardware of the System 250 for communication control
IEEE, International Switching Symposium Record, Cambridge, Massachusetts, 6-9 June 1972, pp. 530-536

[HA 73] C. HANS
Contribution à l'architecture de mécanismes élémentaires pour certains systèmes générateurs de machines virtuelles
Thèse d'Etat, Grenoble, 24 Novembre 1974

[HO 72] J.R. HORTON
Capability management
The problem of protection in systems : use of capabilities
Séminaires IRIA, 1972, pp. 175-200

[HO 74 a] C.A.R. HOARE
Monitors : an operating system structuring concept
C.ACM 17,10, October 1974, pp. 549-557

[HO 74 b] J.C. HORNING, H.C. LAUER, P.M. MELLIAR-SMITH & B. RANDELL
A program structure for error detection and recovery
Proceedings "Aspects Théoriques et Pratiques des Systèmes d'Exploitation", IRIA, Avril 1974, pp. 177-193

[IN 74] A. INGLESE
Description d'un système de machines virtuelles
Document ENSIMAG, Grenoble, 25 Juin 1974

[LA 69] B.H. LAMPSON
Dynamic protection structures
FJCC 1969, pp. 27-38

[LA 71] B.W. LAMPSON
Protection
Proceedings Fifth Princeton Symposium on Information Sciences and Systems, Princeton University, March 1971, pp. 437-443, reprinted in Operating Systems Review, 8,1, January 1974, pp. 18-24

[LA 72] J.C. LAUER & C.R. SNOW
Is supervisory state necessary ?
Proceedings of the ACM - AIA ICS, Venice, 1972

[LA 74] H.C. LAUER
Protection and hierarchical addressing structure
International Workshop on Protection in Operating Systems, IRIA, August 1974, pp. 137-148

[LA 75] P. LAFORGUE
Construction de sous-systèmes utilisant une machine abstraite. Réalisation autour du noyau GEMAU
Thèse de Docteur-Ingénieur, Grenoble, 1er Février 1975

[LI 73] B. LINDSAY
Suggestions for an extensible capability-based machine architecture
International Workshop on Computer Architecture, Grenoble, June 1973

[MA 72] R.L. MANDELL
Hardware/Software trade-offs reasons and directions
FJCC 1972, pp. 453-459

[MO 73 a] B.J. MOORE
A classification of central processor architecture
International Workshop on Computer Architecture, Grenoble, June 1973

[MO 73 b] J.H. MORRIS
Protection in programming languages
C.ACM 16,1, January 1973, pp. 15-21

[NE 72] R.M. NEEDHAM
Protection systems and protection implementations
Proceedings AFIPS FJCC 1972, pp. 571-578

[NE 74] R.M. NEEDHAM & M.V. WILKES
Domains of protection and the management of processes
The Computer Journal 17,2, May 1974, pp. 117-123

[OR 72] E.I. ORGANICK
The MULTICS System : an examination of its structure
MIT Press, 1972

[PR 72] L. PRESSER & J.R. WHITE
Linkers and Loaders
Computing Surveys 4,3, September 1972, pp. 149-167

[RE 74] D.D. REDELL & R.S. FABRY
Selective revocation of capabilities
International Workshop on Protection in Operating Systems, IRIA, August 1974, pp. 197-209

[SA 74] J. SALTZER
Protection and the control of information sharing in MULTICS
C.ACM 17,7, July 1974, pp. 388-402

[SC 72] M.D. SCHROEDER & J. SALTZER
A hardware architecture for implementing protection rings
C.ACM 15,3, March 1972, pp. 157-170

[SO 69]
[SO 70] Software Engineering
NATO Science Committee, Garmisch 1969 and Rome 1970

[SP 74] M.J. SPIER, T.N. HASTINGS & D.N. CUTLER
A Storage Mapping Technique for the Implementation of Protective Domains
Software Practice and Experience, vol 3, 1974, pp. 215-230

[VA 69] D.H. VANDERBILT
Controlled information sharing in a computer utility
Project MAC, Internal Report MCA TR 67, October 1969

[VE 73] J.P. VERJUS
Nature et composition des objets manipulés dans un système de programmation
Thèse d'Etat, Rennes, Septembre 1973

[WE 67] H. WEBER
Implementation of EULER on the System/360 model 30
C.ACM 8,9, September 1967, pp. 547-558

[WI 68] M.V. WILKES
Time-Sharing computer system
Mac Donald, London 1968

[WI 72] R.K. WILLIAMS
System 250. Basic concepts
Proceedings Conference on Computer Systems and Technology, IERE, London, October 1972, pp. 157-168

[WU 74] W. WULF, E. COHEN, W. CORWIN, A. JONES, R. LEVIN, C. PIERSON & F. POLLAK
HYDRA : The kernel of a multiprocessor operating system
C.ACM 17,6, June 1974, pp. 337-345

[ZA 71] R. ZACKS, D. STEINGART & J. MOORE
A firmware APL Time-Sharing system
SJCC 1971, pp. 179-191

Annexe 2

D O C U M E N T A T I O N   G E M A U - S P S

P R E S E N T A T I O N S   D I V E R S E S

A. DOCUMENTS DE PRESENTATION

A.1. Présentation de GEMAU, Janvier 1973

A.2. Présentation de SPS, Janvier 1973

B. PUBLICATIONS

B.1. Application de la structuration de système à la conception interne du système GEMAU
Colloque sur la Programmation, Avril 1974, Paris, Springer Verlag 19
Ph. DARONDEAU, S. GUIBOUD RIBAUD & C. OTRAGE

B.2. Espace d'adressage et espace d'exécution du système GEMAU
International Symposium on Operating Systems Theory and Practice
Paris, 23-25 Avril 1974,
S. GUIBOUD RIBAUD & J. BRIAT

B.3. A formal model for system-oriented architecture
IFIP 1974, Stockholm, Août 1974
Ph. DARONDEAU, S. GUIBOUD RIBAUD & C. OTRAGE

B.4. Architecture de calculateurs orientés vers les systèmes
RAIRO B2, 1974
Ph. DARONDEAU, S. GUIBOUD RIBAUD, C. OTRAGE & H. SMIT

B.5. Introduction to system architecture : software
CERN School of Computing, Godoysund, Norvège, Août 1974
S. GUIBOUD RIBAUD

B.6. Construction de sous-systèmes utilisant une machine abstraite. Réalisation autour du noyau GEMAU
Thèse de Docteur Ingénieur, Grenoble, 1er Février 1975
P. LAFORGUE

C. SPECIFICATIONS DE REALISATION DU NOYAU DE GEMAU

C.1. Organisation et contrôle général du noyau
Serge GUIBOUD RIBAUD, 30 Mai 1973

C.2. Système de mise au point
Serge GUIBOUD RIBAUD & Jacques CHARLET, 3 Mai 1973

C.3. Description informelle de la structure du noyau
Serge GUIBOUD RIBAUD, 30 Juillet 1973

C.4. Gestion de la mémoire réelle
Serge GUIBOUD RIBAUD & Xavier ROUSSET de PINA, 4 Juin 1973

C.5. Modules et processus de gestion de l'unité centrale
Xavier ROUSSET de PINA, 16 Juin 1973

C.6. Gestion des entrées-sorties physiques :
UEDISP, COCEDISP, IOREQUEST, IOCOMPLETE, COCEIN, COCEOUT
Jacques LECOURVOISIER, 10 Avril 1973

C.7. Comptage du temps réel
Serge GUIBOUD RIBAUD, 5 Mai 1973

C.8. Allocation et gestion de la mémoire secondaire (DISK)
Alain INGLESE, 13 Avril 1973

C.9. Gestion de la mémoire virtuelle (PAGE)
Alain INGLESE, 18 Mai 1973

C.10. Allocation des ressources (SCHEDULER)
Xavier ROUSSET de PINA, 30 Octobre 1973

C.11. Gestion de l'espace d'adressage (DATA)
Roland BALTER, 20 Septembre 1973

C.12. Module et processus de synchronisation logique
Jacques BRIAT, 31 Juillet 1973

C.13. Gestion des entrées-sorties logiques
Harm SMIT, 18 Avril 1973

C.14. Gestion des déroutements. Module TRAP
Roland BALTER, 19 Juin 1973

C.15. Gestion des segments en mémoire centrale. Module GSTA
Roland BALTER, 19 Avril 1973

C.16. Exemples de cheminement du contrôle dans le noyau
Jacques BRIAT & Claude OTRAGE

D. DOCUMENTS INTERNES - NOYAU DE GEMAU

D.1. Mécanismes de base de l'accès à l'espace d'adressage du système GEMAU
Alain INGLESE, 1er Mars 1973

D.2. Réalisation de l'espace d'adressage de GEMAU
Roland BALTER, 1er Mars 1974

D.3. The problem of access revocation and object substitution in capability systems
Roland BALTER, Jacques BRIAT & Xavier ROUSSET de PINA, Janvier 1975

E. DOCUMENTS INTERNES - UTILISATION DE GEMAU

E.1. Langage de commande
Jacques CHARLET, 1er Janvier 1974

E.2. Construction d'un système sur machine abstraite : le système Conversationnel SSC sur GEMAU
Pierre LAFORGUE, 1er Janvier 1974

E.3. Ecriture des primitives du langage de commande
Jacques CHARLET, 24 Janvier 1973

E.4. Description externe du système conversationnel
Pierre LAFORGUE, 3 Mai 1973

E.5. Description d'un système de machines virtuelles
Alain INGLESE, 25 Juin 1974

F. S.P.S.

F.1. Structures de données dans les langages ALGOL 68, EL/1, PASCAL, SIMULA 67
Robert Van REETH, Rapport DEA, Juin 1972

F.2. Pré-étude de machine PASCAL
François ANCEAU & Guy BOULAYE, Décembre 1972

F.3. Emulation
Guy BOULAYE, Décembre 1972

F.4. Introduction des notions de mode, processus et opérateur de contrôle dans une structure de système
Serge GUIBOUD RIBAUD & Claude OTRAGE, 19 Avril 1973, DI EA

F.5. Moniteurs associés à un objet
Serge GUIBOUD RIBAUD & Claude OTRAGE, SPS INF

G. DIVERS

G.1. Etude d'un langage pour systèmes interactifs
Christian JULLIEN & Jacques LECOURVOISIER, Rapport DEA, Juin 1972

G.2. Langage de spécification fonctionnelle et simulation de systèmes (Rapport de définition)
Philippe DARONDEAU, Rapport ENSIMAG, Juillet 1972

G.3. A methodology for the global design of information systems, System specification and simulation language
Philippe DARONDEAU, International Workshop on Computer Architecture, Grenoble, Juin 1972

G.4. Names & Modes. Noyau d'architecture et métalangage pour l'élaboration de systèmes modulaires organisés sur la définition d'environnements spécialisés
Philippe DARONDEAU, Rapport ENSIMAG, Juillet 1973

G.5. Architecture des calculateurs
Guy BOULAYE, Serge GUIBOUD RIBAUD & Claude OTRAGE
Ecole d'Eté de l'AFCET, Grenade, Juillet 1973

H. PRESENTATIONS, COMMUNICATIONS et SEMINAIRES

H.1. Projet GEMAU
S. GUIBOUD RIBAUD, Colloque Franco-Soviétique sur les Systèmes, Novosibirsk, Septembre 1972

H.2. Espace d'adressage du système GEMAU
S. GUIBOUD RIBAUD & C. OTRAGE, Séminaire Université de Rennes, Janvier 1973
Séminaire IRIA, Février 1973,
Séminaire de Programmation, Université de Grenoble, 1972

H.3. Langage de spécification fonctionnelle et simulation de systèmes
Ph. DARONDEAU, Séminaire ENSIMAG, Mars 1973

H.4. Architecture des calculateurs
G. BOULAYE, S. GUIBOUD RIBAUD & C. OTRAGE, Ecole d'Eté de l'AFCET Grenade, Juillet 1973

H.5. Comment un espace de noms peut être utilisé pour l'allocation des ressources. Application à GEMAU
X. ROUSSET de PINA, Séminaire ENSIMAG, 23 Novembre 1973

H.6. Espace d'adressage de GEMAU et allocation de ressources
X. ROUSSET de PINA, Séminaire Université de Clermont Ferrand, Novembre 1973

H.7. Introduction des modes et moniteurs dans une structure de système
Ph. DARONDEAU, Séminaire ENSIMAG, Décembre 1973

H.8. La spécification : une approche scientifique de la conception des systèmes
Ph. DARONDEAU, Ecole d'Eté de l'AFCET, Grenade, Juillet 1973

H.9. A methodology for the global design of information systems. System specification and simulation language
Ph. DARONDEAU, International Workshop on Computer Architecture, Grenoble, Juin 1973

H.10. Systèmes à la carte et multiprocesseurs
S. GUIBOUD RIBAUD & C. OTRAGE, Journées IRIA, Structures résultant d'un groupement de processeurs, Saint Pierre de Chartreuse, Novembre 1973, pp. 94-105

H.11. Présentation de GEMAU
Présentation à la Direction Technique de CII, Grenoble, Janvier 1974

H.12. Journées GEMAU
Présentation générale et démonstration auprès de l'Université de Grenoble, 14-15 Janvier 1974

H.13. Journées de comparaison des systèmes GEMAU et SAR
Université de Rennes, 31 Janvier-1er Février 1974

H.14. Espace d'adressage et espace d'exécution du système GEMAU
H. SMIT
Méthodologie de réalisation, allocation de ressources
X. ROUSSET de PINA
C.E.R.T., Toulouse, 8 Février 1974

H.15. Journées GEMAU
IRIA, 18-19 Mars 1974

H.16. Présentation de GEMAU
J. BRIAT, P. LAFORGUE, J. LECOURVOISIER, Séminaire Université de Nancy, Avril 1974

H.17. Concepts d'objets, de modes et d'espace de noms dans un système
Ph. DARONDEAU, Séminaire Université de Nancy, Avril 1974

H.18. Etude comparative de la nomenclature et de l'adressage dans GEMAU, MULTICS, B6500
R. BALTER, Mai 1974

H.19. Description d'un système de machines virtuelles
A. INGLESE, ENSIMAG, Mai 1974

H.20. Evaluation d'un langage de spécification
Ch. de MONTETY, ENSIMAG, Juin 1974

H.21. Cours GEMAU
CICG, Juin 1974

H.22. Cours Méthodologie et réalisation de GEMAU
CICG, Octobre-Novembre 1974

H.23. Présentation de GEMAU
R. BALTER, J. BRIAT, S. GUIBOUD RIBAUD & C. JULLIEN, CNET Lannion, 14-15 Novembre 1974

H.24. Conférences AEA
Ecole des Mines de Saint Etienne, Mars-Avril 1974

H.25. Cours système. Application à GEMAU
CERI Alger, Année scolaire 1974-1975

H.26. Révocation des droits d'accès et substitution dans les systèmes à adressage par descripteur
R. BALTER, J. BRIAT & X. ROUSSET de PINA, Séminaire ENSIMAG, 17 Janvier 1975

Annexe 3

G L O S S A I R E

Accès : voir chaîne d'accès, résilisation d'accès, matrice d'accès

Ancêtre : processus initial, prédéfini, ascendant de tous les processus du système

Annuaire (objet) : catalogue d'objets

Capability : terme anglais pour descripteur

Chaîne d'accès : succession de liens permettant d'atteindre un objet

Contexte : tout moyen permettant de connaître la signification d'un nom. Cf. environnement

Contrôleur : composant d'un objet, lié à la synchronisation

Degré de multiprogrammation : nombre maximal de processus pouvant lier ou manipuler un objet (propriété intrinsèque d'un objet)

Déliaison : rupture de liaison entre un espace d'exécution et un objet (libération d'un nom local). Cf. Liaison

Dénomination : cf. désignation

Descripteur : ensemble d'informations (nommables) permettant d'accéder à un objet, manipulable uniquement à l'aide de primitives

Désignation : action d'associer un nom (quelconque) à un objet (synonyme : dénomination)

Domaine : ensemble des objets accessibles, munis de leurs droits d'accès, par un processeur à un instant donné

Environnement : annuaire distingué servant de référence pour la résolution de noms partiels émis par un processus

Espace d'exécution : ensemble des objets liés à l'exécution d'une procédure à un instant donné

Espace d'objets : structure arborescente de l'espace des noms partiels

Espace permanent : espace de résidence des valeurs discrètes des segments et des descriptions des objets de l'espace des objets

Espace temporaire : espace de modification des objets entre deux valeurs discrètes

Espace visible : ensemble de tous les objets désignables depuis un environnement donné

Interpréteur : voir processeur

Liaison : action d'associer un nom local à un objet : liaison à un espace d'exécution et, par contrecoup, à un processus

Lien (objet) : objet permettant d'accéder à un autre objet (indirection)

Lien implicite (d'annuaire) : objet de nom partiel prédéfini permettant de définir des règles de recherche particulières pour les noms d'entrée

Lien résolu : valeur d'un objet de type lien qui indique le nom universel d'un objet, c'est donc la partie non substituable de la référence

Machine abstraite : définition formelle de l'ensemble des objets et des règles de manipulation

Manipulation (primitives de) : ensemble des opérateurs applicables sur un annuaire

Méfiance mutuelle : propriété de protection relative entre un utilisateur et un sous-système (ou entre deux sous-systèmes)

Nom : toute représentation servant à désigner un objet

Nom d'entrée : nom d'un objet dans un annuaire (nom de conservation d'un objet)

Nom local : nom d'un objet lié à un processus, c'est un index dans un espace d'exécution (nom permettant l'utilisation d'un objet)

Nom partiel	: nom d'un objet par rapport à un environnement (espace d'objets). C'est une liste de noms d'entrée
Nom universel	: désignation unique d'un objet indépendamment de tout espace
Noyau	: automate permettant de réaliser les différents espaces et les primitives de manipulation et d'utilisation des objets
Objet	: entité élémentaire
Opérateur	: fonction d'utilisation d'un objet
Périphérique (objet)	: objet représentant un organe d'entrée-sortie désignable par les processus
Pile de contrôle	: structure contenant les espaces d'exécution d'un processus
Porte	: attribut permettant de définir un annuaire comme étant un environnement
Prérogatives	: ensemble des primitives utilisables par un processus. Elles sont définies dans un environnement.
Primitives	: opérateur de base de la machine abstraite
Processeur	: robot d'exécution d'un programme
Processus	: exécution d'un programme séquentiel
Protection	: ensemble de mécanismes permettant de définir ce qui est possible à un instant donné pour un processus donné sur un objet (ou ensemble d'objets)
Racine	: annuaire d'origine de l'espace des objets
Résiliation (d'accès)	: rupture d'une chaîne d'accès
Restrictions (d'adressage ou de manipulation)	: masque sur les primitives de manipulation pour un objet donné (cf. protection)
Restrictions (d'utilisation)	: masque sur les différents opérateurs associés à un objet

Section de liaison : mécanisme permettant de rendre évolutif un espace d'exécution statique

Segment (objet) : objet de type procédure ou donnée. Associé à un processeur

Sous-système : tout ensemble d'objets réalisé dans l'espace des objets à partir d'un environnement donné

Substitution : propriété de remplacement (dans l'espace des objets) d'un objet donné par un autre possédant les mêmes opérateurs

Utilisateur : voir sous-système

Valeur : un des composants d'un objet, représentant son contenu

Valeur discrète (d'un segment) : valeur cohérente d'un objet à un instant donné

Volume (objet) : objet permettant une structuration de l'espace permanent. Espace physique de résidence.