J.R. ABRIAL

STRUCTURES DE DONNEES

1970-71

Chapitre 1

Point de vue épistentiel

1 Point de vue existentiel

1.1 Exemple de la vie courante

Nous allons faire une analogie entre une donnée et une personne vivante.

Une personne habite (ou est d') un certain lieu, elle porte un nom et elle a des caractéristiques qui la différencie des autres personnes. Ces caractéristiques ont certaines valeurs. Par ailleurs avant sa naissance une personne était dans le néant, elle y retourne à sa mort. Enfin, au cours de sa vie une personne noue toute sorte de relations avec d'autres personnes.

La vie d'une personne est essentiellement une série d'actions qui consistent à

- Pe faire changer de lieu
- Pa faire changer de valeur
- éventuellement la faire changer de nom.

La naissance d'une personne est une opération qui consiste à

- lui donner un nom
- lui fournir un lien
- lui donner quelques premières valeurs

La mort d'une personne est une opération qui la rend inexistante (sauf dans les mémoires) donc (constitue) à lui supprimer :

- nom
- lien
- valeur

Lorsque deux personnes parlent d'une troisième elles peuvent la désigner de diverses façons. Par exemple Napoléon sera désigné par

- l'Empereur des Français (de France)
- Bonaparte
- le petit caporal

Voici d'autres exemples :

Définition par liens :

Roi de France
Duchesse de Lamballe
Pépé le Moco
l'homme de la Mayenne
le locataire du 5e
le lit 104 - B (hopital)

Définition par Noms

Louis XIV
Buffalo - Bill
de Funès
Dupont
1 - 38 - 11 - 78 - 646 - 010

Définition par valeur

L'homme à l'oreille cassée
L'homme au Marteau de Fer
la vésicule biliaire (hopital)

Sur ces exemples nous pouvons voir apparaître les caractéristiques de ces différents

modes de désignation :

La désignation par lieu est non ambiguë à un instant donné mais elle n'a pas tout à fait indéfiniment

Exemple : Un roi de France peut être décédé, un malade peut guérir le CS 104 - 3

La désignation par valeur est très ambiguë et non stable dans le temps -

La désignation par nom est (en principe) la seule qui soit toujours valable - En fait le nom est souvent la juxtaposition d'un certain nombre de valeurs ou d'éléments du lien de telle sorte qu'ils forment un tout suffisamment représentatif pour être non ambiguë : Ceci est typiquement le cas du N° de Sécurité Sociale qui est un mélange de la date de naissance, du sexe et d'un numéro d'ordre et une clef et du lieu de

naissance d'autre part - (14)

Un dernier exemple montre à quel point ces modes de designation sont courants : On trouve bien un Botin par nom, un Botin par rue et un Botin par profession.

On peut enfin designer une personne par une relation par rapport à une ou plusieurs autres :

Exemple :

d'amant de Lady Chatterley

le Père du Régiment

Cette manière peut être ambigue ou non suivant la nature de la relation

On peut aussi designer une personne par l'intermédiaire d'un lien

Exemple lorsque l'on trouve l'avis suivant devant une porte

dose : " Changement d'adresse - d'une nouvelle adresse est - - - "
on peut alors désigner une personne par l'adresse indiqué à une certaine adresse etc - - -

1.2 Données enregistrées et Données calculées

Avant d'étudier approfondissement la structure d'une donnée nous allons éliminer un cas particulier.

En informatique une donnée peut être enregistrée (stockée en mémoire) ou calculée (si l'on donne le mode de calcul)

Exemple : soit la donnée dont le nom est sin(30).

Si sin est un tableau, sin(30) indique le lieu où est stockée la valeur correspondante

Si sin est un programme sin(30)

est un appel du programme avec comme paramètre actuel la valeur 30 – Le résultat de l'exécution de ce programme est la valeur de fin(30) –

En fait, que l'on soit dans l'un ou l'autre cas, les problèmes sont les mêmes : Dans le 2ème cas on recherche une donnée particulière : le programme fin fait l'on désigne par son nom.

Par la suite, sauf cas explicite, nous considérerons que toutes les données sont enregistrées –

1.3 Les espaces d'une donnée

Comme une personne une donnée a un nom, un lieu (celui où elle est stockée) et une valeur (stockée dans le lieu) –

Nous allons définir avec précision ces trois notions –

1.3.1 Espace des liens

Nous parlerons d'un lien dès que nous pourrons stocker de l'information - L'unité de lien est donc la position binaire -

La mémoire (ou espace des liens) est donc une réserve de positions binaires

Une donnée peut avoir besoin de plusieurs positions binaires pour être stockée - Le lien d'une donnée est donc caractérisé par sa taille (en nombre de position binaire) - La taille d'un lien pour une même donnée peut varier dans le temps -

Une autre caractéristique d'un lien d'application est le nom de la donnée qui l'occupe - Si ce nom n'existe pas on dira que le lien est libre - On dit aussi que c'est un trou -

Un lien a un nom qui "représente" ce lien par rapport à tous les autres - Le nom d'un lien

est son adresse - L'adresse d'un bloc est en général un paramètre pour un algorithme de selection -

Cet algorithme de selection peut être purement hardware

Exemple :

circuit de selection des registres

circuit de selection de la mémoire centrale -

ou un melange software - hardware

Exemple

handler + armoire de controle pour une mémoire secondaire -

L'espace des blocs dans une installation donnée est généralement

- non homogène
- dynamique

Il est non homogène dès qu'il existe plusieurs algorithmes de selection différents

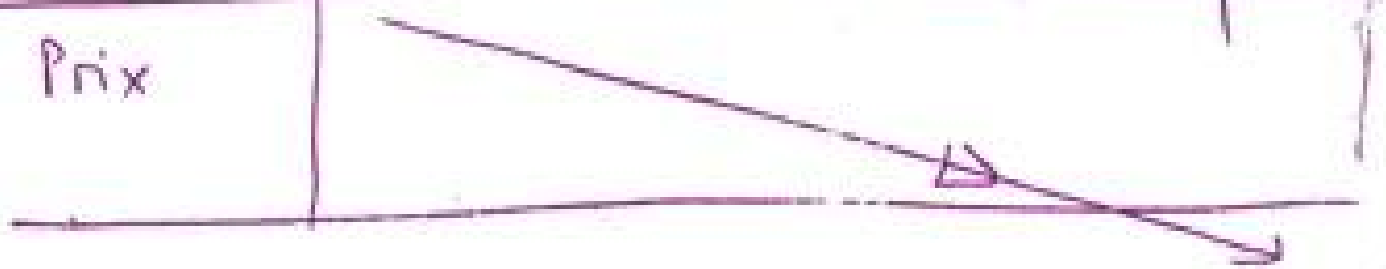
Il est dynamique si sa taille globale peut varier.

On appelle <u>support</u> le sous ensemble d'un espace des biens qui est homogène.

A tout support correspond un <u>temps d'accès moyen</u> et une <u>durée de séjour moyen</u> de l'information.

Le tableau ci-dessous récapitule les diverses caractéristiques.

	Registre	Tores	Tambour	Disque	bande
Taille (en octet)	100	100 K à 1000 K	1 000 K à 10.000 K	∞	∞
Temps d'accès moyen	ns	µs	10 ms	100 ms	1 mn
Durée de séjour moyen	ns	10 ms à mn	10 ms à heure	∞	∞
dynamique	non	non	non	oui	oui
Prix					



On voit sur ce tableau que les supports disque et bande sont les seuls à pouvoir stocker de l'information en permanence et que leur structure dynamique leur permet de trouver les pulsations (naissance, mort) des données.

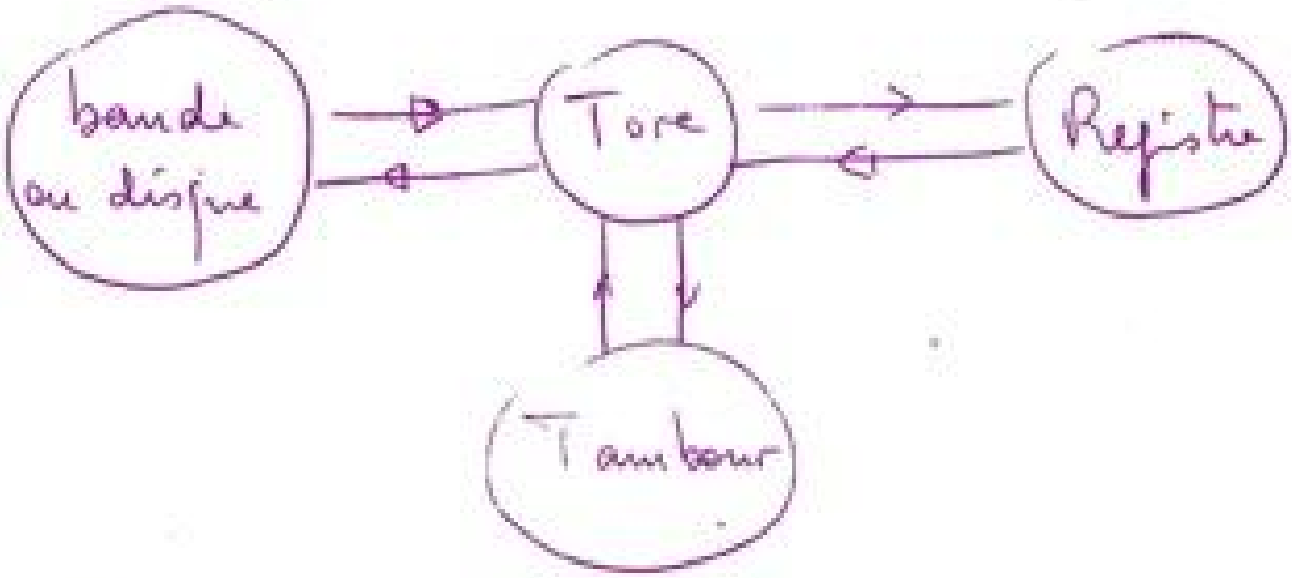
Les supports Tambour et Tore servent à stocker momentanément de l'information venant des bandes ou disque et qui sert de réservoir sous pression.

Le support Registre contient l'information en cours de traitements par l'unité de calcul.

On voit donc que, en fait, le véritable espace des liens est le support disque (ou bande) et qu'il existe ensuite une hiérarchie des supports

de plus en plus rapide, de plus en (2)
plus petit et de plus en plus cher.

Suivant l'usage que l'on fait d'une donnée elle va donc subir des déplacements correspondants au graphe suivant



Pour une même donnée on pourra par exemple avoir le voyage suivant

Disque - Tore - Registre - Tore - Registre - Tore - Tambour - Tore - Registre - Tore - Tambour - Tore - Disque

(22)

La Gestion de l'espace des lieux
a donc en fait en plusieurs
tâches bien séparées :

~~[illegible]~~

Comme nous l'avons vu un lieu
peut être caractérisé par

- son nom (son adresse)
- ses caractéristiques qui sont
  - sa taille
  - le fait qu'une donnée l'occupe ou non
  - le nom (éventuellement) de la donnée qui l'occupe

En fait pour tous les lieux d'un
support, ces éléments sont rassemblés
et constituent un ensemble de donnée
en soi que l'on appelle <u>la table</u>

d'occupation du support

Généralement le nom de la donnée qui occupe un bien est le nom du bien qui occupe cette donnée dans le support situé à un niveau juste inférieur dans la hiérarchie

Exemple :

En mémoire centrale on pourra trouver une table des buffers (un buffer est un bien en mémoire centrale) contenant pour chaque buffer
- son adresse (en mémoire centrale)
- sa taille
- l'adresse tampon de la donnée allouée

Sur le tambour on pourra alors trouver une table d'occupation (cette table pourra être en mémoire centrale) contenant pour chaque bien sur le

tambour :

! son adresse (sur le tambour)
son taille
l'adresse disque de la donnée associée. (ou la bande)

En fin on trouvera sur le disque une table d'occupation du volume contenant pour chaque bloc de ce disque (ou de cette bande)
son adresse (disque)
sa taille
son nom (son "adresse" dans le monde extérieur)

La gestion des blocs sur un support consiste donc à gérer la table d'occupation correspondante de façon à ce que à chaque naissance (venue d'une nouvelle donnée dans un support de niveau inférieur dans la hiérarchie) la nouvelle donnée puisse trouver

un bien fini (de taille suffisante) - (21)

Deux cas peuvent se présenter

a) En cherchant dans la table d'occupation on trouve un bien libre et de taille suffisante - C'est ce bien que la donnée nouvelle va maintenant occuper

β) En cherchant dans la table d'occupation on ne trouve pas un bien de taille suffisante - On peut alors réorganiser les biens sur le support (par exemple compactage) de façon à transformer plusieurs petits trous en un trou unique plus grand - Après réorganisation 2 cas peuvent encore se présenter

(31) La réorganisation donne satisfaction ; c'est à dire qu'elle a pu produire un bien de taille suffisante pour la nouvelle donnée -

(2a) La réorganisation ne donne pas satisfaction (saturation du support) – Si le support n'est pas dynamique c'est qu'il occupe une position haute dans la hiérarchie (tore ou tambour) – On va alors dans ce cas dégrader une ou plusieurs données de ce support en les faisant rejoindre un support de niveau inférieur jusqu'à ce que l'on obtienne un lieu de taille suffisante pour la nouvelle donnée.

Exemple : Dans le cas d'une saturation des tores on fait redescendre une donnée sur tambour

– dans le cas d'une saturation des tambours on fait redescendre une donnée sur disque

– Si le support qui sature est dynamique on peut l'agrandir

en mettant la nouvelle donnée sur un nouveau volume (nouveau disque, nouvelle bande) -

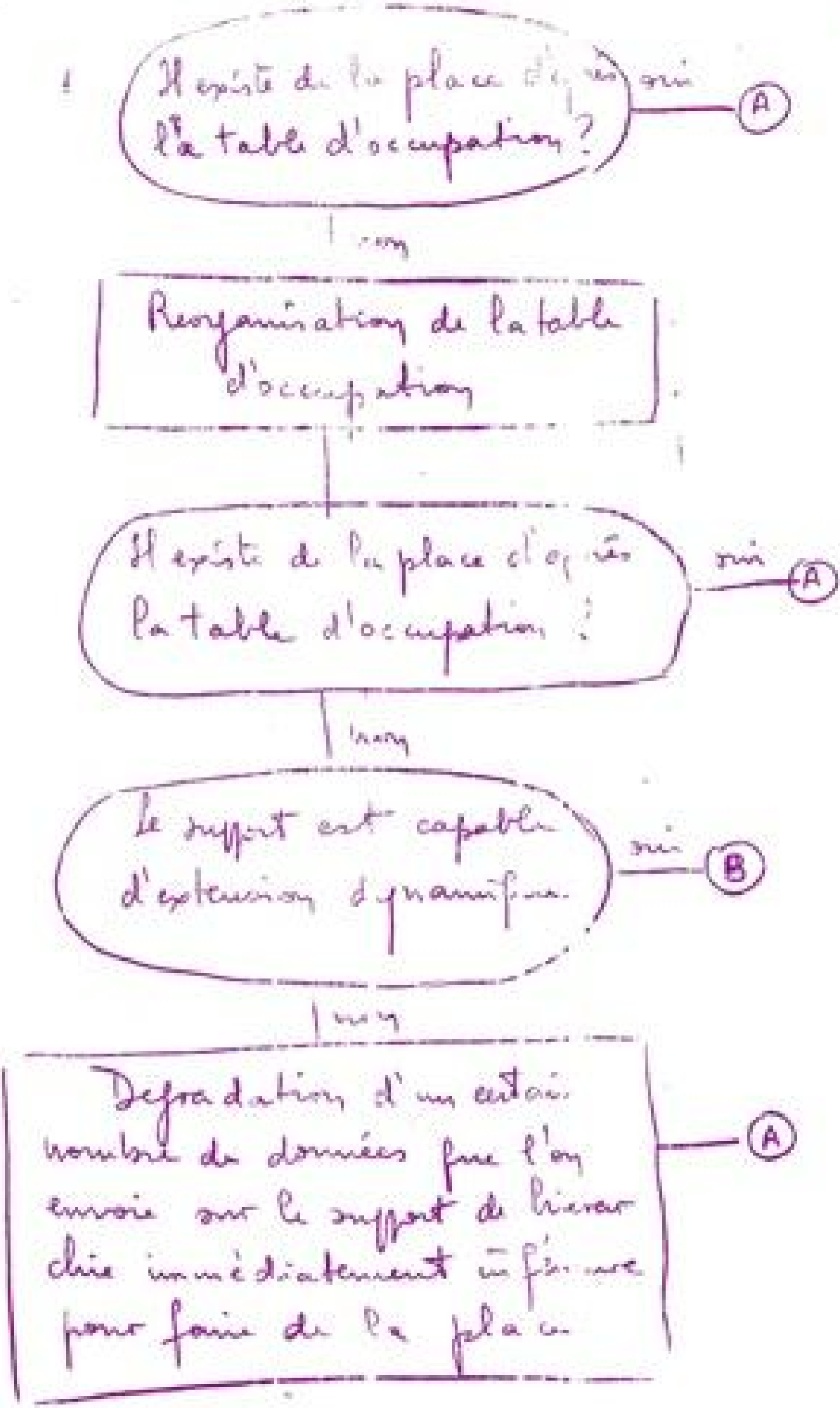
A ce stade il se pose un nouveau problème : En effet si le support est dynamique il n'en est pas de même du support du support

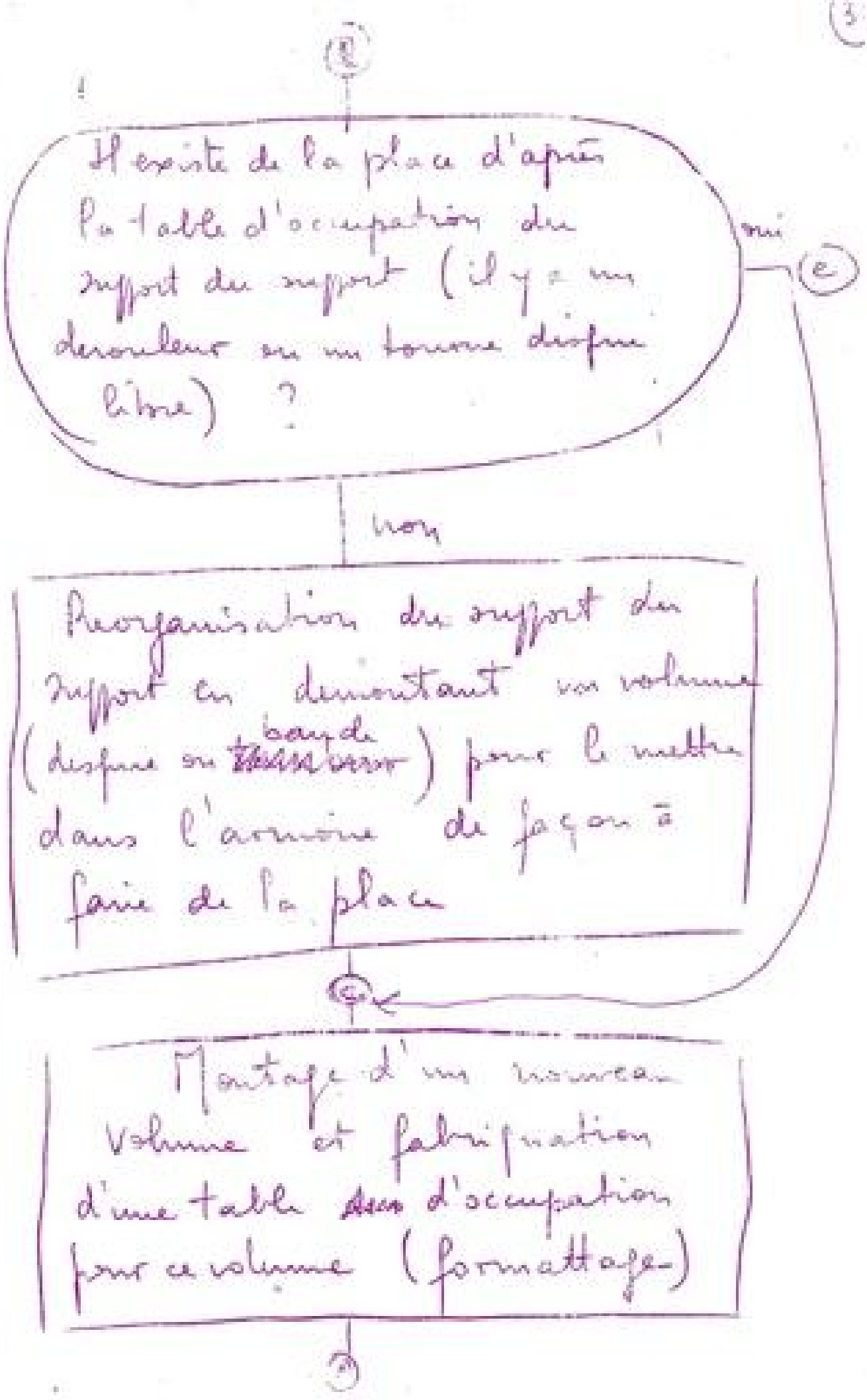
Exemple : Dans une installation de disque le nombre de disques contenant de l'information est a priori quelconque ; par contre le nombre de tourne disque est lui constant -

L'extension dynamique est donc une opération longue qui nécessitera généralement le démontage d'un disque (ou d'une bande) et le remontage d'une autre -

On peut dire que au niveau du support de support on a affaire par

analogie avec celle des supports une (2)
hiérarchie entre
le tourne disque (ou le deroulant)
et l'armoire à disque (ou à bande)
Les tables d'occupation correspondantes
sont respectivement
les étiquettes sur les disques/ou tambour
la liste collée sur la porte de l'armoire
Quant au "programme" de gestion
et de selection des supports son rôle
est joué par l'opérateur -
d'organigramme fait ici recapitule
les opérations de gestion pratiquée
à la naissance d'une donnée sur un
support


Il existe de la place dans la table d'occupation ?
oui
A
non
Reorganisation de la table d'occupation
Il existe de la place dans la table d'occupation ?
oui
A
non
Le support est capable d'extension dynamique
oui
B
non
Defragmentation d'un certain nombre de données pour l'on envoie sur le support de stockage immédiatement inférieure pour faire de la place
A


Il existe de la place d'après la table d'occupation du support du support (il y a un déroulement ou un tronçon disponible) ?
oui
non
Reorganisation du support du support en démontant un volume (disparu ou bande) pour le mettre dans l'armoire de façon à faire de la place
Montage d'un nouveau volume et fabrication d'une table d'occupation pour ce volume (formattage)

Nous avons donné ici un schéma très général des opérations qui peuvent se passer à la naissance d'une donnée. Nous verrons plus loin au § 1-4 les schémas correspondants à d'autres opérations : mise à jour, mort etc...

Les solutions techniques données à ce schéma général sont extrêmement variables d'un système à l'autre.

Il n'est pas question ici d'en faire une étude exhaustive ; nous donnerons cependant quelques détails au § 2.2.2.

Le terme "table d'occupation" que nous avons employé ici est à prendre dans un sens très général - la table d'occupation est un système de données servant à décrire les liens d'un support -

Le terme "table d'occupation" sera à mettre au même niveau que "catalogue" lorsque nous étudierons l'espace des noms au § 1.3.2 ou "table des symboles" lorsque nous étudierons l'espace des valeurs au § 1-3.3

Table d'occupation et hiérarchie des supports sont les notions les plus importantes de ce §.

1.3.2 L'Espace des Noms

L'espace des noms a longtemps (et est encore) confondu avec l'espace des liens - Cette confusion vient du fait

que les données ont d'abord été statiques et à durée de vie limitée : elles naissaient toutes en même temps elles mourraient toutes en même temps -

Par ailleurs, les données ne subissaient pas d'extension dynamique (leur "taille" restait constante) -

Dans ces conditions on comprend que le lien d'une donnée était unique pour chaque donnée, on pouvait leur donner comme nom & leur adresse

Au paragraphe précédent nous avons vu comment

- à un même instant une donnée pouvait occuper plusieurs lieux différents (sur différents supports)
- à deux instants différents une même donnée pouvait occuper deux lieux différents sur un même support (ceci est dû essentiellement aux

possibilités de réorganisation) --

Il apparaît donc souhaitable d'avoir pour chaque "donnée" d'abord qui représente d'une donnée (l'existence d'une donnée) indépendamment de l'espace (les différents supports) et indépendamment du temps - "le" fuel "donnée" est le nom de la donnée par analogie avec les propriétés (presque toujours vraies) du nom d'un être vivant -

Quel profit on ne donne pas de nom aux données, c'est le vrai qui en tient compte et ceci même pour des données dynamiques - La programmation de tel système est très délicate On imagine volontiers toutes les précautions qu'il faut prendre lorsque l'on change une donnée de place sur un support ou qu'on la fait

A propos du nom d'une donnée, on peut se poser la question suivante : qui et comment donne-t-on son nom à une nouvelle donnée qui vient de naître ?

Rappelons d'abord la propriété essentielle d'un nom - La donnée qui le porte est la seule à le porter pendant toute la durée de sa vie -.

Un humain peut être à l'origine de la naissance d'une donnée - (par exemple création d'un fichier par un programme) - le nom est alors proposé par cet humain au système de gestion des noms qui regarde si ce nom proposé n'est pas déjà porté par une autre donnée

Les noms données par les humains aux données sont généralement des chaines de caractères alphanumériques le test par lequel on valide un nom se fait en consultant un catalogue - le catalogue est l'équivalent de la table d'occupation présente dans chaque support, pour les noms généralement, outre le nom, le catalogue contient pour chaque donnée l'adresse correspondante dans la table d'occupation sur le support de plus bas niveau (support permanent) où peut se trouver cette donnée -

Ainsi on voit comment, grâce à la table d'occupation, on peut trouver connaissant l'adresse le nom d'une donnée et inversement comment grâce au catalogue on peut trouver

grâce au nom, l'adresse d'une donnée sur son support de plus bas niveau ...

Un programme peut être à l'origine de la naissance d'une donnée - Au lieu de proposer un nom alphanumérique on peut alors chercher directement dans le catalogue un nom non utilisé - Comme par ailleurs un programme manie plus naturellement de l'information binaire que de l'information alphanumérique les noms donnés par programme seront de préférence binaire -

Comme par ailleurs l'espace des noms ressemble assez à un espace des êtres homogène (cf les expressions sur l'analogie entre catalogue et table d'occupation),

On appelle nouveau l'espace des noms binaires, l'espace virtuel -

Ceci est une abstraction très commode et nous permet de donner les propriétés d'un espace virtuel

(a) d'adresse virtuelle d'une donnée est unique pendant toute la vie de la donnée

(b) il existe un moyen permettant de savoir si une adresse virtuelle correspond ou non à une donnée vivante

(c) l'espace virtuel ne doit jamais être saturé - Comme il est homogène (par définition) il doit pouvoir être extensible (ou alors être extrêmement grand)

(d) Il doit exister une correspondance entre l'espace virtuel et la table d'occupation des supports -

Dans la pratique on trouve beaucoup de faux espace virtuels qui n'ont pas la propriété d'être des espaces de

homme en particulier parce que la propriété fondamentale ⓐ n'est pas satisfaite. Ces faux espaces virtuels servent uniquement à faire une homothétie fatale artificielle de l'ensemble (tore + tambour). - Ils rendent cependant de grands services! -

Il existe 2 sortes de vrais espaces virtuels :

α) Espaces virtuels pour lesquels la correspondance ⓓ est une fonction enregistrée. Le système MULTICS sur le GE-645 est un prototype d'un tel système. La mémoire virtuelle est en fait une mémoire à 2 dimensions : chaque adresse virtuelle est composée d'une adresse de segment et d'une adresse de ligne à l'intérieur d'un segment. La correspondance ⓓ se fait alors au moyen d'une

Table des segments puis pour chaque segment d'une "table des pages". Nous reviendrons sur ces techniques au § 2.2.2.2. Le test ⓑ s'effectue alors soit en détectant un débordement dans les Tables précédentes soit en trouvant une ligne vide dans la table des segments.

β) Espaces virtuels pour lesquels la correspondance ⓓ est une fonction calculée. Le système SOCRATE est un prototype d'un tel système. Le "calcul" se fait en effectuant un hash-coding sur l'adresse virtuelle. Nous reviendrons sur ces techniques au § 2.2.2.2. Le test ⓑ s'effectue en consultant une chaîne de bits ou en détectant une donnée indéfinie. Dans beaucoup de cas une donnée si elle est créée par un programme peut avoir un nom alpha-numérique.

donné par un humain. Dans ce cas il est fréquent d'avoir 2 noms pour une donnée

un nom externe alphanumérique
un nom interne (son adresse virtuelle)

Il est souhaitable bien entendu pour l'un puisse passer de l'un à l'autre et vice-versa -

Par extension on appellera catalogue l'étendue la correspondance entre le nom externe et l'adresse virtuelle.

Dans MULTICS le catalogue (étendu) est enregistré ; dans LOCRATE le catalogue (étendu) est calculé (un hashcoding de le nom externe donne le nom interne) -

1.3.3 Espace des valeurs

La valeur d'une donnée est l'information qui est stockée dans le lieu de cette donnée. Au chapitre 5 (Pour le modèle relationnel) nous élargirons cette notion de valeur par celle de relation.

Une donnée n'a de sens que par rapport à l'interprétation que l'on fait de sa valeur.

Exemple : Dans un système automatisé de gestion du personnel d'une entreprise, tôt ou tard on aura pour la couleur des cheveux des gens est une information qui n'intéressera pas. (Cette donnée n'est pas interprétable par le programme de paie par exemple)

Cette remarque paraît très triviale. Elle n'est cependant pas toujours faite par les responsables de la structure des fichiers de gestion qui font stocker

des informations sans elle-même qu'il capte seulement un interpréteur (humain ou programme) capable de voir à leur diverses valeurs.

La manière par laquelle les diverses valeurs d'une donnée peut être représenté sur son lieu est une convention fixé il faut établir avec les interpréteurs de la donnée. Cette convention s'appelle la codification de la donnée

Exemple: le sexe d'une personne peut être masculin ou féminin. Le responsable d'un fichier peut décider que la codification est 1 pour masculin, 2 pour féminin. S'il n'y a pas passage de convention (de préférence écrite) avec le programmeur qui se sert de ce fichier dans un programme (interpréteur) dont le comportement dépend (conditionnel) du sexe des personnes, on risque bien

que le résultat de l'exécution du programme risque d'être complètement faux. Ce cas d'erreur est très fréquent.

Parmi les éléments de la codification d'une donnée il est fondamental de fournir la convention correspondant à une information indéfinie. Bien des erreurs arrivent parce que des règles précises n'étaient pas prescrites : souvent "indéfini" est codifié par zéro.

La valeur d'une donnée peut changer au cours de la vie d'une donnée.

Le changement de la valeur d'une donnée peut être éventuellement accompagné d'une variation (augmentation ou diminution) de la taille de son Bien.

On voit donc en résumé que toute donnée s'accompagne nécessairement

d'un certain nombre de règles définies par

- le domaine de variation des valeurs de la donnée
- la codification de ces valeurs
- la manière par laquelle une donnée peut se développer ou se rétracter.

L'ensemble de ces règles constitue ce que l'on appelle le type de la donnée - Le type d'une donnée peut, par analogie, se comparer au "programme génétique" d'un être vivant

Le chapitre 2 (Point de vue morphologique) consistera en une étude détaillée des types de données -

Les données qui sont d'un même type portent un nom générique -

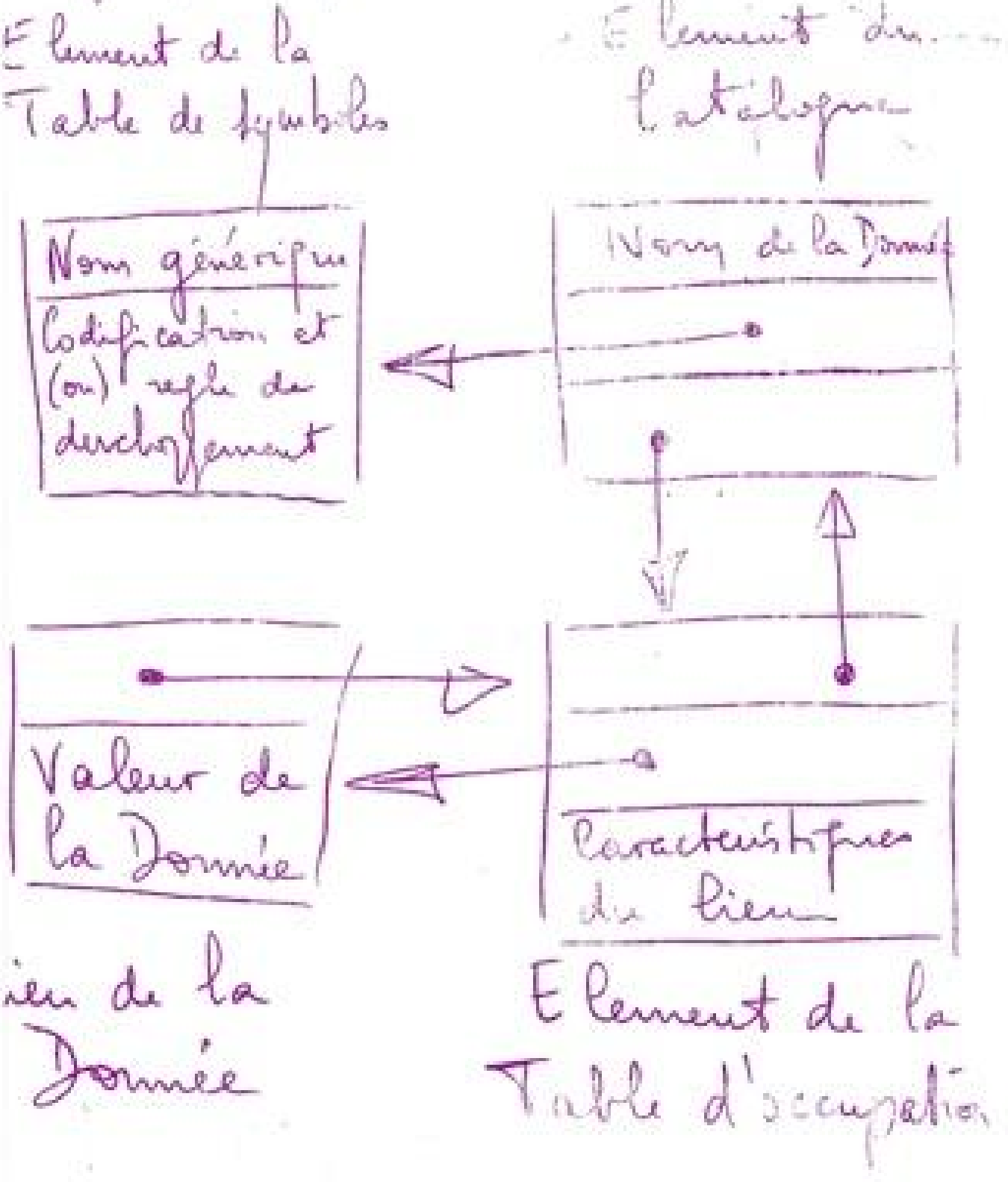
le nom générique et le type d'une donnée sont définis bien entendu avant la naissance de cette donnée. Cette opération de définition s'appelle une <u>déclaration</u> ; nous y reviendrons au § 1.4.1. (Définition).

Généralement on confond le nom générique d'une donnée avec son nom. Ceci n'est en fait pas gênant à un instant donné dans un système. Il n'y a pas une seule donnée correspondant à un certain type : on peut alors lui donner pour nom son nom générique. Dans d'autres cas (Algol), seule la dernière donnée créée d'un certain type est accessible, on lui donne donc par abus le nom générique comme nom.

L'ensemble des types d'un système constitue un système de données en soi que l'on appelle généralement

1.3.4 Récapitulation et Exemples

Nous pouvons récapituler les 3 derniers paragraphes par le schéma suivant



Les ⟶ indiquées sur le schéma ont uniquement le sens d'une relation (non forcément réalisée par un parleur)

Les réalisations pratiques correspondant à ce schéma sont extrêmement diverses : parfois certains éléments du schéma sont totalement absents ; parfois ces éléments sont plus ou moins ; parfois des intermédiaires viennent compliquer (souvent de façon inextricable) ce schéma de base. (à croire que leurs auteurs ont voulu réaliser une véritable "jungle" de parleurs et de liens de façon à ce que les "apprentis" s'y perdent)

Nous allons étudier maintenant quelques exemples typiques

1.3.4.1 Exemples : Data Set

(Données de l'"operating system" IBM)

ⓐ Table des symboles

La Table des symboles est fragmentée. Chaque élément en est constitué par le Data Control Block (DCB)

ⓑ Nom générique

Le nom générique est le DDNAME du Data Set

ⓒ Catalogue

Le catalogue est représenté par le Job File Control Block (JFCB) qui contient les DD Statements et en particulier l'association entre

le nom générique (DD NAME)
le nom (DSNAME)
le lien (Volume)

Parfois si le Data Set est catalogué (CATLG), l'association entre le nom et le lien se fait dans ce fui est appelé effectivement le CATALOGUE.

d) Nom de la Donnée

- nom de la donnée est son DSNAME

e) Lieu et Table d'occupation

lieu de la Donnée se trouve sur un volume. La table d'occupation s'appelle Volume Table of occupation (VTOC) chaque élément de la VTOC s'appelle Data Set Control Block (DSCB).

- DSCB contient le DSNAME, la section du lieu du data set sur le volume et les caractéristiques (taille, formatation etc...) du lieu -

f) Création et suppression

opération de création se fait à aide d'un DD statement (NEW) : le Data set ne doit plus exister indiquée dans le DD statement la mention DELETE

② Référence

Les documents suivants donnent tous les détails sur l'organisation de ce système de données :

IBM System/360 Operating System Job Control Language page 24 (DD Statement)

IBM System/360 Operating System Supervisor and Data Management Services page 65 ... (Interface with the Operating System).

1.3.4.2 Exemple 2 : Enregistrement dans un fichier indexé séquentiel

② Table des symboles

La Table des symboles contient la (ou les) structure(s) des enregistrements du fichier. Si le fichier est créé et utilisé par COBOL, cette structure est définie dans la DATA DIVISION du programme COBOL.

Si le fichier est utilisé en Pl/1, la structure de l'enregistrement est généralement définie par une STRUCTURE (déclaration de donnée composée) la table des symboles sera donc dans ces 2 cas, la "Table des symboles" des ces compilateurs -

Si le Fichier est utilisé en assembleur la table des symboles n'existe pas : la structure de l'enregistrement est donnée de façon externe au programmeur -

① Nom générique

Le nom générique est le nom du fichier

② Catalogue

Le catalogue est représenté par les INDEX qui contiennent à différents niveaux les listes des clés (keys) de différents enregistrement ainsi que un pointeur vers leur position sur le Data set

④ Nom de la Donnée

La donnée # est : ici un enregistrement (RECORD) - de nom de la donnée est sa clé (KEY) -

⑤ Lien et Table d'occupation

Le lien de la donnée se trouve dans un Data set (Voir § précédent)
En toute enfin la table d'occupation n'existe pas puisque l'index donne une correspondance entre nom et lien - Cependant l'index ne donne pas une correspondance partielle (toutes les clés ne figurent pas dans l'index : on cherche d'abord dans l'index où les clés sont classées puis séquentiellement dans le Data set à partir de l'enregistrement dont la clé est la plus voisine de celle de l'enregistrement cherché : d'où le nom

indépé séquentiel) -

En fait au niveau de chaque enregistrement (dans le lien lui même) on trouve des informations concernant l'organisation de ce lien : la taille de l'enregistrement (si elle taille est variable) et le fait qu'il correspond à un enregistrement inexistant (X FF' : delete code)

① Création et suppression

L'opération de création se fait à l'aide d'une opération WRITE indiquant une clé qui n'existe pas déjà dans le fichier

L'opération de suppression se fait à l'aide d'une opération de mise à zéro en mettant la valeur du "delete code" - Dans cette organisation la suppression n'entraine pas la destruction

tion automatique des bloc —

⑨ Références

Le document suivant donne les détails sur l'organisation de ce système de données

IBM System /360 Operating System Supervisor and Data Management Services page 111 ... (Processing an Indexed Sequential Data Set)

1.34.3 Exemple 3 : Allocation automatique des blocs en PL/1

ⓐ Table des symboles

Le type d'un bloc PL/1 est l'ensemble des déclarations faites à l'intérieur de ce bloc — C'est la Table des Symboles du Compilateur PL/1 qui contient ces informations

ⓑ Nom générique

Si le bloc correspond à une procédure le nom générique du bloc est le nom de la procédure —

Dans le cas contraire le bloc n'a pas de nom générique.

ⓒ Catalogue

Il n'y a pas de catalogue car il n'y a jamais qu'un seul bloc qui a un nom : le dernier créé.

ⓓ Nom

Le nom d'un bloc n'existe que pour le dernier créé : c'est R13 (le Registre 13 contient l'adresse absolue de l'allocation du bloc).

ⓔ Lien et Table d'occupation

Le lien d'un bloc s'appelle une Dynamic Storage Area (DSA). Il se trouve dans une zone de mémoire centrale réservé à l'expansion dynamique des programmes PL/1 (Subpool).

Des éléments d'organisation de la DSA se trouvent dans la DSA elle même (longueur, chainage) - Par

ailleurs une liste des éléments libres (5)
de la "subpool" indiquant les éléments
qui peuvent constituer une DSA à
la création d'un bloc.

(f) Création suppression

La création d'un bloc se fait
par une opération appelée PROLOGUE.
Cette opération appelle une routine
de la bibliothèque PL/1 qui est un
nommé IHESADA, laquelle se
sert de la macro DS : getmain

La suppression d'un bloc se fait par-
l'opération EPILOGUE faisant
appel à IHESAFA laquelle utilise
la macro DS : freemain.

(g) Références

On trouvera de plus amples détails sur
l'organisation de ce système de données
dans :

IBM System /360 Operating System
PL/1 Programmer's Guide (Run Time stack) page 130

et dans:
IBM System/360 Operating system
supervisor and Data Management
services page 44 ... (mais Storage
Manager).

17.4.4 Exemple 4: Données basées
en PL/1 (Based Data)

① Table de Symbole

Celle du compilateur ou si la donnée
est une structure on a un "DOPE VECTOR"
qui décrit la structure de la valeur
de la donnée - (ou mieux de la classe de la donnée)

② Nom générique

Ce nom générique est le nom de la
donnée basée à sa déclaration

③ Catalogue

Il n'existe pas car on ne donne pas
(malheureusement) de nom à la
création

④ Nom de ③

(e) Lien et Table d'occupation

Le lien pour une donnée basée de PL/1 se trouve à l'intérieur d'une zone appelée AREA.

La Table d'occupation est dispersée dans l'AREA. Chaque élément (de l'AREA) est

- soit occupé par une donnée basée (Allocated) dont la taille est indiquée
- soit libre (free) - les éléments libres de l'AREA forment une liste libre dont les éléments sont chaînés entre eux par taille croissante.

(f) Création et suppression

La création se fait à l'aide de l'opération ALLOCATE et la suppression à l'aide de l'opération FREE.

Une variable POINTEUR (faisant office de nom) contient l'adresse de la donnée basée lors de sa création.

Cependant on ne peut pas dire pour

le nom de la variable POINTER
est le nom de la donnée basée
car un instant donné plusieurs POINTER
peuvent être associés à une même donnée
et de plus un POINTER peut changer
d'association.

(f) References

De plus amples détails sur ce type
de données peuvent être trouvés dans
- IBM System /360 Operating System
PL/1 Programmer's guide - page 145...
(Allocation and Release of Storage in an
Area)
- IBM System /360 Operating System
PL/1 Language Reference Manual - page 171...
(Based Variables and List Processing)

1.3.4.5 Exemples : Record class en Algol W

Ce type de donnée est très proche
d'une donnée basée de PL/1

Les variables de type REFERENCE, jouent le rôle des variables de type POINTER de PL/1

La différence vient des opérations de création et de suppression. En Algol W la création est explicite comme en PL/1 mais il n'existe pas d'opérateur spécial, c'est l'affectation d'une "Record class Identifier" à une variable de référence qui en tient lieu.

La suppression explicite en PL/1 (FREE) est ici implicite : un record d'Algol W est implicitement supprimé lorsqu'il n'existe plus aucune variable de type référence pointant sur lui.

1.3.4 C Exemple 6 : Système de pagination (6)

décrit dans l'article : Use of Fast and Slow memory in List Processing Languages par J. Cohen.

(a) Table de symboles

Il n'y a pas de table de symboles. Toutes les "données" sont de même type et sont des pages de longueur fixe

(b) Noms génériques

Pas de noms génériques.

(c) Catalogue

Le Catalogue est un Tableau indexé par l'adresse disque de la page.

(d) Nom

L'adresse disque de la page constitue son nom.

(e) Lien et Table d'occupation

Le lien d'une page se trouve en mémoire centrale dans une zone fixe pouvant contenir un certain nombre de pages.

e) Table d'occupation donne pour chaque position dans la zone dynamique des pages : le nom de la page qui l'occupe et d'autres renseignements annexes —

(f) Création et suppression

Ces opérations correspondent à l'entrée d'une page de la mémoire secondaire dans une position en mémoire centrale et à la sortie d'une page en mémoire secondaire (ou éventuellement à reconnaissance d'une position par sa sortie d'une autre page)

(g) Références

Pour les plus amples renseignements voir l'article "Use of Fast and slow memories in List Processing languages" par J. Cl. C A.C.M. Vol 10. N°2 - Feb. 67

1.3.4.7 Exemple : Entités dans le système Socrate —

(a) Table des symboles

La table des symboles est la structu-

Interne de la Banque de Donnée - (b)
Une entité a une description complète de ses composants dans la structure Interne -

(b) Nom générique

Le nom générique est le nom de l'entité donné lors de la définition de la Banque de Donnée

(c) Catalogue

Le catalogue n'existe pas explicitement : c'est l'ensemble des adresses virtuelles - Via correspondance entre une entrée du catalogue $\frac{b}{12}$ et le fait par hashcoding - On peut dire donc que le catalogue a lieu d'être enregistré comme habituellement est calculé par le programme de hashcoding -

(d) Nom

Le nom d'une donnée est son adresse virtuelle -

e) Lien et Table d'occupation

Le lien est dans l'Espace réel -
La Table d'occupation des liens est rassemblée (Technique du dictionnaire rassemblé) ou dispersée -
Le lien d'une donnée est en fait éclatée en diverses sous-pages (blocs de longueur fixe) dans l'espace réel -
La Table d'occupation fait les sous-pages -

f) Création / suppression

Les opérations sont explicites (b, t)

g) Références

Pour de plus amples détails voir Projet Socrate - Université de Grenoble -

1.4 Opérations sur une donnée

Nous venons d'introduire pour une donnée trois espaces :

- Nom
- Temps
- Valeurs

Si nous introduisons de plus l'espace vide Λ, nous pouvons alors envisager les opérations sur une donnée comme des opérations faisant passer d'un espace au même ou à un autre, le ou les autres espaces valeurs dans le ou les autres espaces restant constantes.

Nous pouvons alors schématiser ces opérations sur la matrice de transition suivante

	L	N	V	Λ
L	Deplacement	Designation plurielle	accès plurielle	Deallocation
N	localisation logique	changement de nom ?	accès logique	suppression
V	localisation associative	désignation associative	Mise à jour	Devalorisation
Λ	allocation	baptême	Initialisation	

Avant d'étudier ces diverses opérations nous allons d'abord considérer l'opération de définition.

1.b.1 Définition

L'opération de définition est celle qui consiste à "déclarer" un nouveau type (une nouvelle classe) de donnée - C'est une opération de création

dans le système de donnée constituée par
- nom générique (Nom)
- Table des symboles (Lien)
- Organisation et codification (Valeur)

Exemple
Lorsque l'on fait la déclaration suivante en PL/1 :

DCL TOTO FIXED BINARY (31) ;

On crée une nouvelle classe de données de nom générique TOTO et d'organisation FIXED BINARY (31)

C'est bien pour le système de données explicité plus haut la création d'une donnée puisque l'on a une opération de baptême et d'initialisation - c'est le compilateur qui fait l'allocation de cette "méta donnée" dans la Table des symboles

Lors de la définition on fixe généralement la codification des données futures de cette classe : Ainsi dans

l'exemple ci dessus les données de la classe TOTO seront des nombres binaires entiers codés sur 31 bits.

La codification n'est pas toujours aussi explicite. Ainsi en ALGOL on écrit par exemple "INTEGER A;" c'est alors la version du compilateur APPOL avec laquelle on travaille qui définit implicitement la codification des futures données de la classe.

Lorsque les données de la classe devront porter chacune un nom, la codification de celui-ci peut être définie lors de la définition de la classe.

. Exemple :

Lorsque l'on veut définir un fichier sous DS on emploie la macro DCB. Parmi les (nombreux) paramètres de cette macro figure bien entendu le DDNAME

qui est le nom générique et aussi un paramètre KEYLEN qui donne la longueur (en caractères) de la clé des enregistrements. On a donc bien ici, lors de la définition, une spécification de la codification du nom des futurs données de la classe.

La définition d'une classe de données peut être une opération ponctuelle (qui se fait en une seule fois) : c'est le cas des données définies dans un programme écrit dans un langage de haut niveau (c'est la déclaration). Par contre, malheureusement, une classe de données peut être définie en plusieurs fois dans certains cas. Ceci rend beaucoup plus compliqué le programme de définition et surtout des erreurs.

Exemple: un data set (IBM). (?) peut en effet être partiellement défini dans une macro DCB, mais aussi dans le DDSTATEMENT des cartes de contrôle (paramètre introduit par DCB) ~~[illegible]~~ ~~[illegible]~~ enfin d'autres informations de définitions peuvent être contenues dans la Table d'occupation. Bien entendu ces diverses sources d'information définissant une même classe de données (un même fichier) peuvent être incohérentes et les auteurs ont donc défini des règles de précédence. (Pour de plus amples détails voir le document IBM System/360 Operating System Supervisor and Data Management Services page 65. (Interface with the Operating System))

Pouvoir définir une donnée en plusieurs fois est évidemment plus souple mais cela complique peut être arbitrairement l'organisation des données.

1.4.2 Opérations de création (Allocation, baptême, initialisation)

De ces trois opérations c'est l'allocation qui est la plus importante et qui existe toujours.

Créer une donnée suppose qu'il en existe un nom générique. Dans la traduction on trouve deux classes de création :

- ou bien la Table des symboles est présente (c'est le cas pour les fichiers en général)
- ou bien la table des symboles n'est pas présente (c'est le cas pour les données définies dans un programme)

Dans tous les cas l'allocation consiste à chercher dans la Table d'occupation des liens, un lien qui est apte à contenir la nouvelle donnée - ~~en~~ l'information concernant la taille du lien se trouve, si dans la table des symboles, soit dans les éléments de valeurs fournies par l'initialisation - (si la Table des symboles n'est pas présente lors de la création d'information concernant la taille du lien est indiqué dans un programme de création généré par le compilateur pour cette classe de données)

d'allocation d'une donnée peut être une opération automatique (c'est le cas des données déclarées à l'intérieur d'un bloc en PL/1 ou en Algol) ou manuelle (intervention humaine explicite ou par l'intermé

d'aire d'un programme : c'est le cas des données basées de PL/1, des enregistrements de fichiers et des data sets).

On appelle souvent dans la littérature le moment où l'on "accroche" une donnée nouvelle à son nom générique le "binding-time"

Exemple : - les données dites STATIC de PL/1 ont leur "binding-time" au début de l'exécution du programme dans le quel elle sont définies (quel fusionnent leur position dans ce programme)

- les données dites AUTOMATIC de PL/1 ont leur "binding-time" à l'entrée dans le bloc où elle sont définies.

Le baptême, s'il a lieu, se passe toujours "en même temps" que l'allocation. Comme l'allocation, le baptême peut être explicite (le nom de la donnée est fournie

(ou par une attribution de valeur) : le baptême est refusé si la même fonction existe déjà pour une autre donnée - c'est le baptême par exclusion le cas de la création d'un enregistrement de fichier muni d'une clé d'accès -

Le baptême peut être automatique ; dans ce cas c'est le système lui-même qui fournit un nom (une adresse virtuelle) à la donnée nouvellement créée - Dans ce cas le système est obligé de connaître les noms déjà affectés pour donner à la nouvelle donnée un nouveau nom - Dans le système SOCRATE une chaîne de bits sert à indiquer quels sont les noms affectés et ceux qui ne le sont pas - (Pour de plus amples détails voir Propel Socrate page 197 (Technique Hatfield)

d'initialisation (fournitures d'une première valeur à la donnée) est une opération qui peut avoir lieu en même temps que l'allocation et

le baptême : c'est le cas par exemple des données dont la déclaration porte l'attribut INIT en PL/1. Il n'en est pas toujours ainsi = très souvent l'initialisation a lieu après la création proprement dite. Dans ce cas entre le moment de la création et le moment de l'initialisation il se passe un certain temps pendant lequel la donnée existe (puis qu'elle a été allouée et baptisée) et est accessible mais n'a pas de valeur. Que se passe-t-il alors si l'on demande sa valeur.

Dans la pratique 2 cas se présentent dans le 1er (le plus fréquent) cas, on a une erreur (le programme s'arrête): il est interdit de manipuler des données indéfinies (c'est le cas des données PL/1) dans le 2e (Banque de Données Converse travailles) une donnée indéfinie (non initialisée) est une donnée licite au même titre que une autre. Il peut en

effet très intéressant de connaître les noms des données qui n'ont pas de valeurs de façon à pouvoir repérer cette lacune...

Dans ce 2ème cas, l'opération inverse de l'initialisation, la dévalorisation est donc une opération licite qui n'entraîne pas automatiquement la suppression de la donnée - On peut alors avoir pour une même donnée plusieurs initialisations et/ou plusieurs dévalorisations.

1.4.3 Opérations de suppression (Deallocation, débaptisation)

La suppression d'une donnée est une opération toujours beaucoup plus complexe que sa création.

Nous verrons en effet au chapitre 3 (Point de vue relationnel) qu'une donnée n'est jamais isolée comme elle apparaît dans ce chapitre ici - En fait, outre les valeurs portées d'une donnée, il

Le nom au sens de la vie de l'une (?) données toute sorte de liaisons -

Les relations se manifestent dans la pratique par des pointeurs (des données dont la valeur est une adresse : un lien) ou par des références (des données dont la valeur est le nom d'une autre) (Attention! le sens que nous donnons ici au terme référence est assez particulier et ne correspond pas toujours à celui - là que l'on trouve dans la pratique où l'on confond souvent référence avec pointeur. cf Algol w)

Bien entendu, lorsqu'une donnée meurt, son lien et son nom doivent disparaître - Or il se trouve que par le jeu des relations que nous venons d'exposer ces noms et liens sont invoqués par des paramètres référence et des pointeurs

Aussi avant de supprimer une donnée (ou la détruire) il est indispensable de rendre indéfini tous les paramètres ou objets ces fois

supprimer cette donnée - Toute la complexité de la suppression vient de cette obligation

Comme pour la création, la suppression peut être explicite (cas d'un enregistrement d'un fichier) ou implicite (cas d'une donnée AUTOMATIC de PL/1 supprimée automatiquement à la sortie du bloc où elle est définie) - Il existe un autre cas de suppression implicite intéressant. une donnée est supprimée lorsqu'elle n'est plus accessible par aucun pointeur - C'est le cas des couples (CAR-CDR) de LISP - Nous y reviendrons au chapitre 3 -

Lorsqu'une donnée est supprimée, sa désallocation peut être immédiate ou différée - (le 2ème cas est celui des enregistrements indexés séquentiels où la suppression consiste seulement à marquer que l'enregistrement fut il est supprimé : X'FF' en tête du

l'enregistrement : - de sorte que la désallocation est différée le lieu de toutes les données est réorganisé lorsque la fragmentation devient trop importante (Voir § 1.3.1. Espace des temps) - c'est au cours de cette réorganisation qu'on fait bien en même temps toutes les désallocations qui avaient été différées depuis la dernière réorganisation - : Au lieu de faire les désallocations coup par coup on les fait toutes ensemble -

Les techniques de garbage-collection que nous verrons au chapitre 3 font partie de ces méthodes de désallocations différées -

La débogération consiste, s'il y a lieu, à supprimer le nom de la donnée du catalogue (s'il existe) - Marche

1.4.4 Opérations d'accès

(accès physique, accès logique, accès associatif)

Les opérations d'accès (logique ou physique) sont celles qui à partir d'un nom ou d'une adresse donnent la valeur d'une donnée.

Par extension on appellera accès associatif l'opération qui à partir de la connaissance d'une partie seulement de la valeur d'une donnée fournit la totalité de la valeur de la donnée.

Les opérations d'accès sont donc les suivantes

L $\longrightarrow$ V accès physique

N $\longrightarrow$ V accès logique

V $\longrightarrow$ V accès associatif

1.4.4.1 Accès physique

L'opération L $\longrightarrow$ V est toujours faite par le hardware.

En fait si l'on reflechit au processus exact de cette opération, on s'aperçoit qu'elle est plutôt une opération L $\longrightarrow$ L.

c'est à dire un déplacement.

L'opération L → V est le produit d'un certain nombre de déplacements pour amener la donnée (ou une partie de celle-ci) à être sur un support où un interpréteur peut la prendre en compte.

Si l'interpréteur est humain, le lieu final peut être un papier d'imprimante, un écran cathodique, une carte perforée interprétée etc...

Si l'interpréteur est un programme, le lieu final est un registre.

Dans tous les cas la mémoire centrale est la plaque tournante de l'information : on ne peut pas malheureusement faire communiquer directement un disque avec une imprimante ou un perforateur de carte (mais peut être cela existe-t-il après tout ?...)

Par contre les opérations RD, WD (write direct, read direct) permettent

en principe de passer directement d'un (8)
qui permet périphérique spécialement
connecté à un registre (et inversement).
(Ceci au prix d'un ralentissement considérable de l'Unité Centrale) —

Ce rôle de plaque tournante fonctionnel
lement par la mémoire centrale a
amené les constructeurs à faire
des développements techniques allant
dans le sens de la multiprogrammation
ou dans le sens du ~~[illegible]~~ multitraitement —

En effet les successions de déplacements : Disque - Tore - Imprimante ou
Disque - Tore - Perforateur ou
Imprimante - Tore - Disque ou
Perforateur - Tore - Disque ou
les mêmes avec un Tambour (ou un disque à tête fixe) sont des opérations de routine extrêmement fréquentes et longues - On aimerait

donc fortement perturbent le moins possible le fonctionnement de l'Unité centrale. Avec la 3ème génération, ces opérations peuvent se faire en parallèle avec le fonctionnement de l'Unité centrale grâce au Canal qui est indépendant. Il est cependant nécessaire d'initialiser ces opérations et de gérer la zone Tore servant d'intermédiaire.

La solution multiprogrammation consiste à écrire de petits programmes appelés les <u>synchronisants</u> qui

- sont spécifiques d'un couple (par exemple disque - perforateur)
- possèdent chacun une petite zone de travail en mémoire centrale.

Les synchronisants sont réglés par le jeu des interruptions de fin de transfert : ils opèrent par "bouffée" successives.

Le fonctionnement normal de l'Unité Centrale est donc interrompu de temps en temps pour permettre à un synchronisant d'agir

tialiser. Le transfert d'une banque d'une donnée (ou d'un tambour) vers la zone Tore ou de sa zone Tore vers un périphérique tout ou l'inverse.

Certains constructeurs (peut être même tous) ont pensé que l'action des symbionts perturbait par trop le fonctionnement de l'Unité Centrale et ils ont prévu (ou plusieurs) mettre un petit ordinateur (+) chargé de faire marcher les symbionts. C'est par exemple la solution choisie par CDC sur la série 6000. C'est aussi le rôle du 360/40 lorsqu'il "pilote" un 360/65 avec le système MSP/MVT.

En ce qui concerne l'opération L → V pour laquelle la destination finale est un registre dont nous avons déjà fait au § 1.3.1 (Espace des liens) un certain nombre de développements.

1.4.2 L'accès logique

L'accès logique (opération $N \longrightarrow V$) est toujours le produit d'une opération $N \longrightarrow L$ par une opération $L \longrightarrow V$

Il y a lieu de distinguer tout de suite 2 cas :

① Le catalogue n'existe pas sous quelque forme que ce soit (les données n'ont pas de noms) : La seule manière de faire un accès logique est alors de désigner la donnée par son nom générique (qui lui doit exister évidemment sinon il n'est plus possible de faire un accès dit logique) —

Ce cas est celui des enregistrements dans un fichier séquentiel ou bien des données d'Algol ou de Pl/1 —

Puisque l'on désigne la donnée par son nom générique et non par son nom il y a la plupart du temps une ambiguïté gênante à une liaison

la donnée (à un nom générique) il (2)
correspond plusieurs données —
La seule façon de lever cette ambi-
guité consiste à établir une convention de façon à ce qu'au nom générique il ne corresponde qu'une seule donnée de la classe —
Dans le cas d'un fichier séquentiel la convention est la suivante : l'invocation du nom générique pour accéder à une donnée (opération GET) fournit automatiquement la donnée qui suit dans la séquence celle qui a été accédée la plus récemment —
Dans le cas des données d'Algol ou des données AUTOMATIC de PL/1, l'invocation du nom générique fournit la dernière donnée créée de ce type (pour tout ce nom générique)
Dans ces deux cas il existe donc une convention qui évolue dynamiquement au cours du temps — Par extension du

Cas où il y a un point de rendez-vous
Contact qui établit la liaison entre le nom fichier et l'unité donnée accessible appartenant à cette classe

Dans le cas d'un fichier séquentiel ce point connecté se trouve dans l'élément de la Table des symboles (c'est à dire dans le D.C.B.)

Dans le cas d'une donnée PL/1 ou Algol 68 le point connecté est formé par un registre (qui en fait pointe sur la zone de mémoire correspondant au dernier bloc créé - la donnée elle même est représentée par un déplacement dans cette zone)

(b) Le Catalogue existe -

Dans ce cas l'accès Bâtime consiste à chercher l'élément du catalogue qui contient le nom, puis suivant la réalisation (technique) de la liaison entre

l'élément du catalogue et le lien de la donnée à accéder plus ou moins directement à celle-ci -

Rappelons que le catalogue peut être
- totalement incorporé à la donnée (le nom de la donnée apparaît alors comme une valeur particulière et l'accès logique comme un cas particulier d'un accès associatif) -
- séparé de la donnée mais donnant accès à celle-ci sans passer par une table d'occupation (celle-ci est en fait dans ce cas confondue avec la donnée : la taille de la donnée est en tête du lien de celle-ci -) - Ce cas est celui d'un cheminement d'un fichier indexé séquentiel -
- séparé de la donnée mais ne donnant accès à la donnée que par l'intermédiaire d'une Table d'occupation des liens séparée des données. C'est le cas des Data sets -

Dans tous les cas la première opération à effectuer est une recherche dans le catalogue -

En fait si l'on considère le catalogue comme un système de données en soi et un nom comme la valeur d'une donnée, cette opération est une recherche associative dans le catalogue -

Nous renvoyons au chapitre 2 (§ 2.2.2.1.2) sur les techniques de recherche -

Etant donnée l'importance (fréquence d'emploi, temps de mise en œuvre) des opérations de recherche dans un catalogue, on a assisté depuis les débuts de l'informatique à une prolifération extraordinaire des méthodes de recherche associatives -

Nous allons cependant donner ici un aperçu rapide de la manière dont on fait classer ces méthodes -

Notation : Trois éléments interviennent pour analyser les méthodes de recherche

- le temps moyen de recherche $\overline{t}$ en dénombrement
- la taille du catalogue : E
- le nombre d'éléments actifs du catalogue : N

On introduit souvent en plus le coût du catalogue $C = E \cdot \overline{t}$

Ce coût est intéressant car en général les valeurs de E et $\overline{t}$ varient en sens inverse d'une méthode à l'autre -

En effet si l'on veut construire un catalogue ayant un temps de recherche $\overline{t}$ minimum (on a alors évidemment $\overline{t} = 1$) il est nécessaire d'avoir un catalogue dont la taille est égal au nombre de valeurs possible que peut porter un nom - (On imagine la taille d'un tel catalogue si le nom est un numéro de sécurité sociale $\approx 10^{13}$) - le catalogue est évidemment extrêmement vide

Si l'on veut par contre construire un (9)
catalogue qui a un encombrement minimum on disposera les noms puis le construira en séquence. On aura alors

$$\begin{cases} \bar{E} = N \\ \bar{t} = \frac{N}{2} \end{cases}$$

On peut grosso modo diviser les méthodes de recherche en 3 classes

(a) Celles pour lesquelles le temps de recherche est indépendant de N. Parmi ces méthodes on trouve l'indexation directe en plus haut, celles qui utilisent des mémoires associatives (dans l'état actuel de l'art, vu le prix de ces mémoires, l'usage en est réduit à de très petits catalogues cf la mémoire associative du 360/67), celles qui simulent par software les mémoires associatives : ce sont les méthodes de hashcoding. Un hashcode est une fonction qui fait passer du nom cherché à un index dans le catalogue

l'indexation pure est un hashcode dont la fonction est l'identité - Le problème du hashcode est donc de choisir suivant la structure des noms une fonction qui minimise l'encombrement -

La fonction de hashcode est donc une fonction "many-one" (c'est à dire que plusieurs noms peuvent donner la même index) - Lorsque deux noms donnent le même index on a une collision

Les variations qui existent entre les diverses méthodes de hashcoding correspondent à l'emploi de diverses classes de fonctions de hashcoding et aux diverses méthodes de réglement des collisions

En toute rigueur, à cause des collisions, le temps moyen de recherche $\overline{E}$ dépend de N et du rapport du coefficient de remplissage : $\frac{N}{E}$ - (d'un catalogue)

On peut montrer que si les noms sont régulièrement disposés dans leur domaine de variation on a : $\overline{E} = \frac{3}{2}\frac{N}{E} \leq \frac{3}{2}$

Il existe une littérature considérable

sur les méthodes de hash coding - Les méthodes des hash coding ont d'abord été developpées pour les compilateurs qui ont également souvent à rechercher les noms générés par des données dans la Table des Symboles -

Pour de plus amples details sur les methodes des hash coding on peut consulter les articles suivants

- Scatter Storage Techniques par MORRIS (CACM Vol 11 N1 Janvier 1968)
- An Improved hash code for Scatter Storage par MAURER (CACM Vol 11 N1 Janvier 1968)
- File Structures using hashing functions par E. COFFMAN (CACM Vol 13 N7 July 197-)

et le chapitre 4 (Table) de l'ouvrage Techniques de Computation par F.R.A. HOPGOOD (Dunod - Traduction 1970)

(b) Celles pour lesquelles le temps d'accès est proportionnel à Log N

le type même de ces méthodes est la recherche binaire bien connue (on dit aussi souvent recherche dichotomique) - Ces méthodes nécessitent ~~[illegible]~~ que les éléments du catalogue soient classés - Ceci explique pourquoi on présente souvent dans la littérature en même temps les méthodes de recherche et les méthodes de classements -

Nous entendons au § 2.2.3 (Arbres) sur les méthodes de recherche utilisant la construction d'un arbre -

Pour de plus amples détails sur ces méthodes voir

- l'ouvrage de I. FLORES - Computer sorting
- Le Numéro spécial des Communications de l'A.C.M. (Vol 6, N5 May 63) consacré aux problèmes de classement et de recherche -

© Celles pour lesquelles le temps de recherche est proportionnel à N (méthodes purement séquentielles) - Elles sont comme nous l'avons dit d'un très mauvais rendement en temps mais d'un très bon encombrement optimum - De plus ces méthodes sont obligatoires si la catalogue est sur un support séquentiel par essence (bande magnétique) - Elles sont donc souvent les seules possibles pour des installations utilisant de petits ordinateurs (à temps d'unité central peu cher) avec des mémoires secondaires des bandes magnétiques - C'est la méthode du pauvre - - -

1.4.3.3 Récapitulations et autres opérations

Au cours des paragraphes 1.4.3.1 et 1.4.3.2 nous venons d'étudier les opérations suivantes

- L ⟶ V accès logique (dont nous avons vu qu'elle était en fait une opération L ⟶ L)

. $N \to \bar{V}$ a ceci près que dans (1) nous avions vu que elle était en fait le produit d'une opération $N \to L$ par une opération $L \to \bar{V}$

. $N \to L$ (cf $N \to \bar{V}$)

. $\bar{V} \to L$ que nous avons étudié à propos de l'opération $N \to L$ puisque est très proche -

Il reste à étudier l'opération

$L \to N$ et l'opération $V \to N$.

(cette dernière peut se décomposer en

$V \to L$ puis $L \to N$.)

L'opération $L \to N$ ne pose aucun problème si le nom de la donnée est stocké dans le lien de celle-ci (c'est ce que je pense la plupart du temps mais s'il y a une catalogue) - si ce n'est pas le cas il existe toujours soit directement soit par l'intermédiaire de la table d'occupation un moyen pour "remonter"

depuis le bien jusqu'à l'élément dans le catalogue (donc jusqu'au nom) (R2)

1.5 Bibliographie

- Another Look at Data
  .G. H. MEALY F.J.C.C. 1967 (pp 525-534)
- Data Structure and their representation in Storage. M. D'IMPERIO - Annual Review in Automatic Programming Vol 5. Pergamon Press New York - Spring 1969 -
- IBM System/360 Operating System - Supervisor and Data Management Services
- IBM System/360 Operating System - PL/1 (F) Language Reference Manual -
- A use of fast and slow memories in list-processing languages. J. COHEN C.A.C.M. Vol 10. N.2 Feb 67
- Scatter Storage techniques
  R. MORRIS C.A.C.M. Vol 11. N1. Janv. 1968
- Projet SOCRATE
  J.R. ABRIAL, G. REAUMÉ etc.... Université de Grenoble - Août 1970

2. Point de vue morphologique

Dans le chapitre précédent nous avions étudié les structures de données sous un angle existentiel sans préciser ce qui était la "valeur" et le "lien" d'une donnée.

Nous avions schématisé à l'extrême ces deux espaces en faisant sous entendre que le lien d'une donnée était une boîte et sa valeur, le contenu de cette boîte

Dans la pratique il en est tout autrement : le lien d'une donnée n'est pas souvent valable en une seule boîte mais de plus il peut varier dynamiquement au cours de la vie de la donnée

Dans ce chapitre nous allons étudier la structure des données d'un point de vue morphologique ; c'est à dire analyser les différentes formes que peuvent prendre les ~~données~~ valeurs des données et voir

comment techniquement on peut réaliser au mieux les liens entre pendants —

Plutôt que de commencer par une étude formelle nous allons d'abord considérer deux "études de cas" fondamentales : les listes linéaires puis les arbres - A propos des listes linéaires nous ferons d'assez longues digressions concernant les pointeurs et les files d'attente -

Nous terminerons le chapitre par une étude plus théorique en nous amenant à comparer la morphologie des données avec celle des langages de Programmation.

2. Liste linéaire (Introduction)

Une donnée est une liste linéaire si l'on peut décomposer sa valeur en une suite ordonnée d'éléments -

Nous parlerons de liste linéaire par opposition au terme "liste" : dans une liste linéaire chaque élément est un tout inséparable à

la structure dufuel on ne s'intéresse pas! –

Une liste linéaire est la plus simple des données à structure de valeur dynamique ; elle possède un défaut de liberté: le son nombre d'éléments –

On voit donc fcn en plus, des opérations que nous avons étudiées au chapitre précédent (création, suppression, accès) il vient s'ajouter un certain nombre de nouvelles opérations spécifiques de la structure de liste linéaire

– les opérations de croissance (extension, diminution ⁂ : ajouter ou enlever des éléments)
– les opérations de selection

Les dernières servent à isoler un ou plusieurs éléments de façon à pouvoir les lire, changer leur valeur ou les enlever – Nous allons donc les étudier d'abord –

Le tableau ci dessous résume un certain

Sélection	du premier du dernier du n^{me} de n'importe quel de tous	élément conditionné
Condition	rien condition sur la valeur condition sur la position	
Condition sur la position	juste en n^{me} position n'importe où	avant après — élément sélectionné

Un système général de gestion de liste linéaire devrait permettre de réaliser simplement toutes ces opérations sur une liste linéaire définie par exemple par son nom.

A noter que l'opération d'ajout d'un élément ne fait bien entendu par rapport à la position d'un élément existant.

(103)

Operation	erreur de nom de liste	erreur de rang	erreur de valeur	erreur de destruction	Commentaire
Creer(l)	son nom existe déjà	/	/	X	le résultat de cette opération est la création d'une nouvelle liste c'est l'opération (baptême + allocation du chapitre 1 –
supprimer(l)	le nom l n'existe pas	/	/	/	suppression de la liste de nom l (debaptisation + deallocation) cf ch. 1
valeur(l, r)	"	Il n'existe pas d'élément de rang r On doit avoir $1 \leq r \leq \text{card}(l)$	/	/	le résultat de cette opération est la valeur de l'élément de rang r de la liste de nom l
rangmin(l, v, r1, r2)	"	On doit avoir $1 \leq r1 \leq \text{card}(l)$ et $r2 \geq r1$	Il n'existe pas de tel élément	/	Cette opération a pour résultat le rang r minimum des éléments de la liste l ayant pour valeur v et tels que : $r1 \leq r \leq r2$
rangmax(l, v, r1, r2)	"	On doit avoir $1 \leq r2 \leq \text{card}(l)$ et $r1 \leq r2$	"	/	Cette opération a pour résultat le rang r maximum des éléments de la liste l ayant pour valeur v et tels que $r1 \leq r \leq r2$
miseajour(l, r, v)	"	on doit avoir $1 \leq r \leq \text{card}(l)$	v ne peut être une valeur licite pour un élément de la liste l	/	Cette opération remplace la valeur de l'élément de rang r de la liste l par la valeur v
ajouter(l, r)	"	On doit avoir $1 \leq r \leq \text{card}(l)+1$	/	X	Cette opération ajoute un nouvel élément à la liste de nom l – cet élément prend le rang r et a une valeur

Opération	Les erreurs				Commentaires
	de nom de liste	de rang	de valeur	de saturation	
enlève (l, r)	Ce nom l n'existe pas	on doit avoir $1 \leq r \leq \text{card}(l)$	/	/	Cette opération enlève de la liste de nom l l'élément de rang r. $\text{card}(l) \leftarrow \text{card}(l) - 1$
$\text{Card}(l)$	"	/	/	/	Cette opération a pour valeur le nombre d'éléments de la liste l

Cet ensemble d'opérations représentent un répertoire souhaitable pour un système général de gestion de listes linéaires.

Les solutions techniques que l'on rencontre dans la pratique sont très variables suivant la manière dont les listes sont allouées en mémoire. Mais il est assez rare de trouver un système permettant effectivement toutes ces opérations.

La manière dont nous avons organisé ces opérations est assez typique de la façon dont on devrait travailler en programmation [illegible].

Si l'on part en effet d'une machine
booléenne pourvue d'une mémoire centrale,
et d'un certain nombre de mémoires de con
taires ainsi que d'un répertoire d'instruc
tions et si l'on implémente à partir de
ces possibilités les 9 opérations ci dessus,
on s'est fabriqué une nouvelle machine
munie d'une mémoire centrale ostensible qui
n'est pas un simple vecteur mais un
ensemble pouvant comporter de liste linéaire.

Ainsi la génération des programmes
(et donc des programmes) qui se servent
de cette nouvelle machine ont l'impression
d'avoir à leur disposition un hardware
plus élaboré que le hardware de base.

Ils peuvent à leur tour construire de
nouvelles opérations etc....

On voit que les erreurs que nous avons
indiquées ici sont l'équivalent pour cette
nouvelle machine des erreurs que l'on
peut faire en utilisant le hardware

de base comme : une division par zéro, une tentative d'atteindre une adresse inexistante etc...

Dans la partie suivante traite ces erreurs d'utilisation du hardware par un déroulement vers un ~~point~~ programme implanté en mémoire en une zone constante.

Il est souhaitable que les erreurs rencontrées dans l'emploi des instructions de la nouvelle machine soient traitées d'une façon analogue.

A titre d'exemple on peut écrire à l'aide des opérations précédentes l'algorithme : dans (l1, l2) qui a pour résultat (dans un registre r0) le rang (minimum) dans la liste de nom l1 à partir duquel on trouve une suite d'éléments consécutifs identiques à ceux de la liste de nom l2

```
dans (l1, l2)
    r0 ← 1
α:  v2 ← valeur(l2, 1)
    r0 ← rangmin(l1, v2, r0, card(l1))
    r1 ← r0 + 1
    r2 ← 2
β:  v2 ← valeur(l2, r2)
    v1 ← valeur(l1, r1)
    if v1 = v2
        then [ r1 ← r1 + 1
               r2 ← r2 + 1
               if r2 > card(l2)
                   then goto fin
                   else goto β ]
        else [ r0 ← r0 + 1
               goto α ]
fin:
```

Si l'on sort de l'algorithme avec une erreur (detectée par l'une des fonctions de base) c'est que l2 n'est pas compris dans l1

Dans certains systèmes de gestion de liste linéaire, seule un certain nombre des opérations du tableau ci dessous sont permises et constituent des cas particuliers fondamentaux

Pile

Une pile est une liste linéaire sur laquelle seules les opérations suivantes sont permises

opérations générales	opération - pile
créer	créer
supprimer (l)	supprimer (l)
enlève (l, card(l))	retire (l)
ajoute (l, card(l)+1)	pousse (l)
mise_jour (l, card(l), v)	met (l, v)
valeur (l, card(l))	valpile (l)

File d'attente

Une file d'attente est une liste linéaire sur laquelle seules les opérations suivantes sont permises :

Opération générale	Opération - File
Créer	créer
supprimer (l)	supprimer (l)
enlève (l, 1)	tête (l)
ajoute (l, card(l)+1)	pousse (l)
mise à jour (l, card(l), v)	met (l, v)
valeur (l, 1)	valfile (l)

2.2 Pile

La pile est un des mécanismes les plus largement répandus en programmation système.

Nous allons illustrer l'emploi d'une pile pour l'écriture de programmes réentrants.

Un programme réentrant est un programme qui n'est pas modifié par sa propre exécution. Que peut-on modifier dans l'exécution d'un programme ? On peut modifier

ⓐ les instructions de l'algorithme

A u début de l'informatique, il était courant de voir circuler "artistes" écrits des programmes qui modifiaient leurs instructions en cours d'exécution (ajustage télécommandé etc...)

Ce genre d'acrobatie a à peu près (heureusement) complètement disparu car on s'est rendu compte que les programmes ainsi écrits étaient illisibles et très difficiles à mettre au point; & de plus ils ne pouvaient pas être rendus réentrants

(b) les zones de travail de l'algorithme. Celles-ci sont au nombre de trois : (b1) la zone où sont disposés les paramètres d'appel de l'algorithme

(b2) la zone où sont disposés les paramètres de résultat de l'algorithme

(b3) la zone où sont disposées les variables de travail proprement dites.

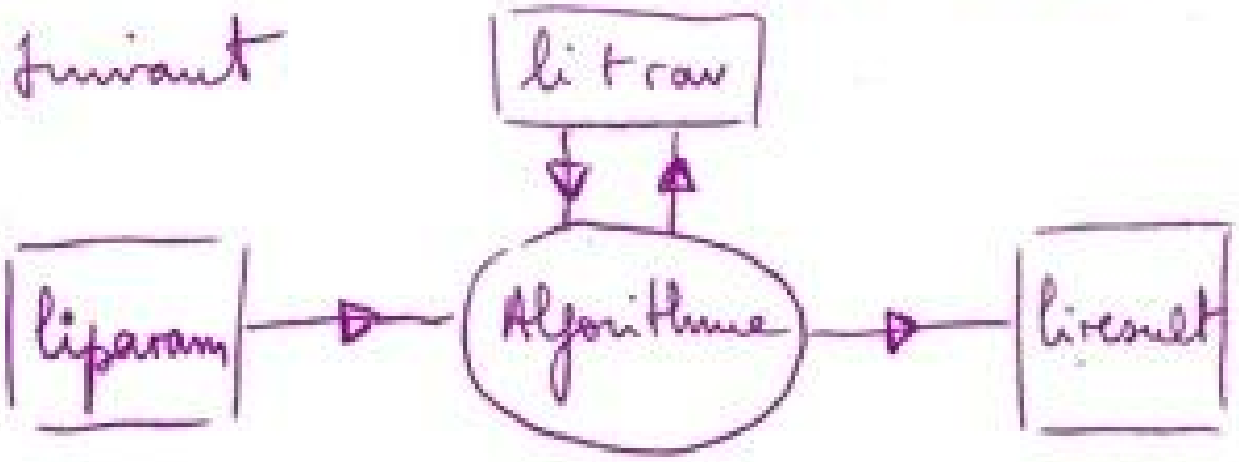
Bien entendu, suivant les cas, ces zones b1, b2, b3 peuvent être inexistantes.

Rendre un programme réentrant c'est donc faire en sorte que les 3 zones b1, b2, b3 ne soient pas contenues dans l'algorithme lui même -

En fait ces zones sont des listes, nous les appellerons par la suite :

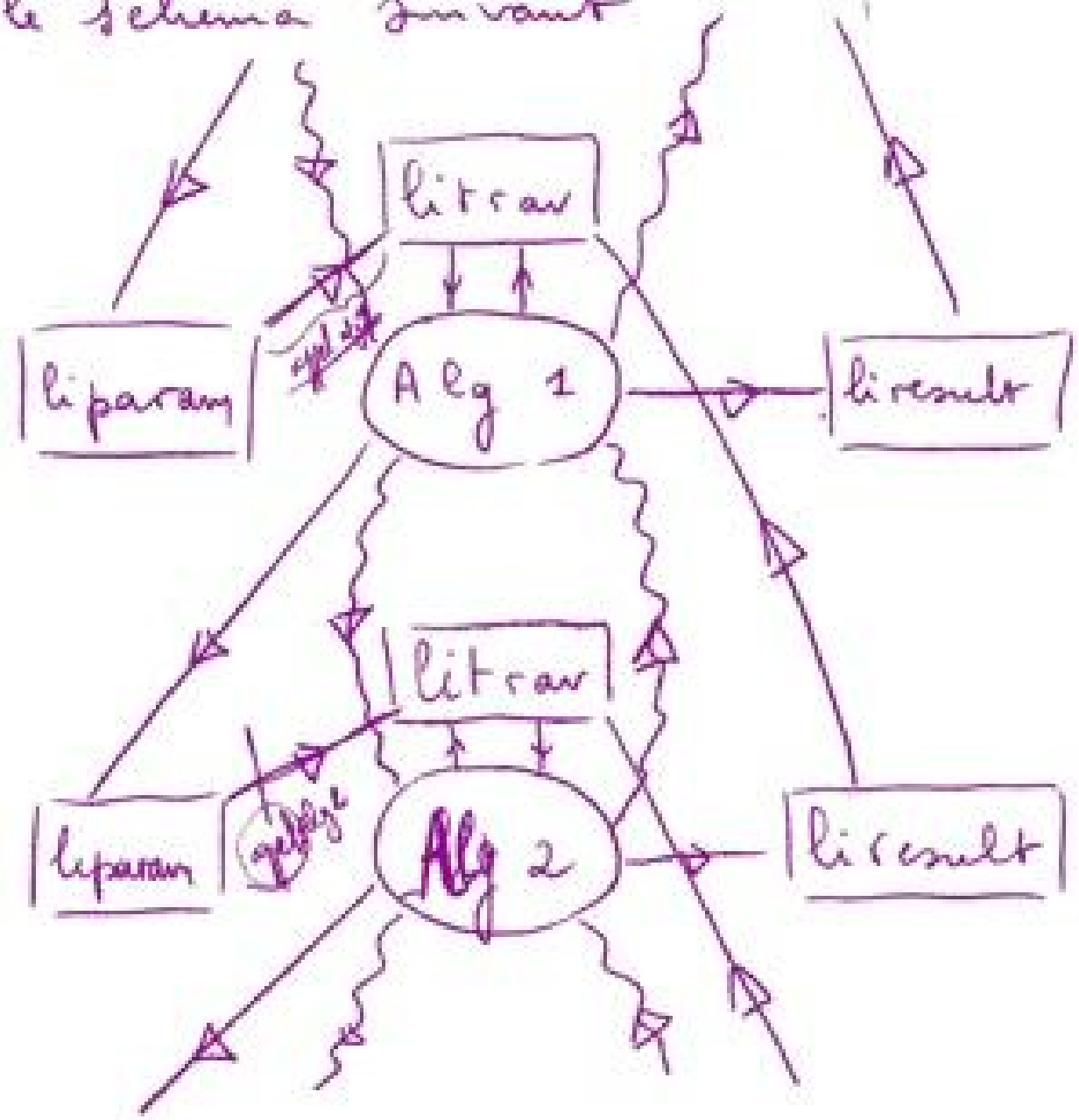
- liparam
- liresult
- litraw

Le schéma d'exécution d'un algorithme réentrant doit donc être le suivant



En fait ce schéma se complique car un algorithme peut en appeler un autre avec retour (sous-programme) ou sans retour (passage de contrôle).

Dans le 1er cas on peut dessiner le schéma suivant



Sur le schéma nous avons supposé que

(a). au début de l'exécution d'un algorithme sa liste de paramètre était transférée dans litraw de façon à libérer la liste liparam pour un autre appel

(b). Au cours de l'exécution, l'algorithme travaille avec litraw (en la faisant grandir dynamiquement éventuellement) et il élabore ses résultats dans liresult ; s'il désire appeller un autre algorithme il ~~élabore la liste de paramètres~~ élabore la liste de paramètres dans liparam avant de faire l'appel proprement dit

(c) à la fin de l'exécution de l'algorithme on doit enlever de litraw l'environnement de son exécution c'est à dire les paramètres

puis nous avions transféré (en ⓐ) et (11ᴇ) les variables de travail puis nous y avions mis (en Ⓑ) - Puis l'on doit mettre dans litrav le resultat (le mot [illegible]) et enfin redonner le controle à l'algorithme appellant -

On voit donc que litrav forme bien comme une pile -

Dans la pratique litrav est généralement constituée par les registres de l'unité centrale qui ne sont pas en nombre suffisant pour constituer une pile de taille suffisante. On doit donc constituer une pile indépendante -

Dans les conventions de l'O.S (IBM) la pile est formée par une suite de SAVEAREAS doublement chainés entre elles; nous reviendrons sur

ces techniques au § # 2.3 (Allocation des listes linéaires)

Nous allons maintenant à l'aide des opérations générales définies au § 2.1 pour les piles définir de nouvelles opérations permettant de réaliser le mécanisme du schéma ci dessus -

Rappel des opérations sur les piles

pousse(l) : ajoute un nouvel élément à la fin

retire(l) : enlève le dernier élément

met(l, v) : donne la valeur v au dernier élément

valpile(l) : a pour résultat la valeur du dernier élément

Très souvent on fait successivement les 2 opérations pousse(l) et met(l, v) - Nous introduisons donc l'opération :

insert (l, v)

pousse (l)
met (l, v)

Nous avons aussi besoin des deux opérations sauve (l1, l2) et restaure (l1, l2) qui vide la pile l2 dans la pile l1 en ajoutant en plus dans l1 la taille initiale de la pile l2
La 2e est l'opération inverse (on l'on suppose donc que le dernier élément de la pile l1 contient le nombre d'éléments à insérer dans la pile l2)
A la fin de sauve (l1, l2) la pile l2 est vide -

sauve (l1, l2)

```
trav ← card (l2)
for i = 1, 1, trav
    do [ insert (l1, valpile (l2))
         retire (l2) ]
insert (l1, trav)
```

```
retourne (l1, l2)
    trav ← valpile (l1)
    retire (l2)
    for i = (1, 1, trav)
        do [insert (l2, valpile (l1))
            retire (l1)
```

A l'aide de ces 2 nouvelles operations nous allons maintenant construire les operations d'appel et de retour Dans chaque cas nous considerons 3 variantes

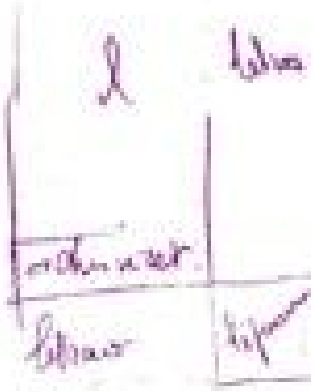
- cas général (liparam ou liresult)
- cas special 1 (un seul parametre, un seul resultat)
- cas special 2 (~~un seul~~ pas de parametre pas de resultat)

La liste l est la pile. Elle est implicite -

Nous avons donc les operations suivantes

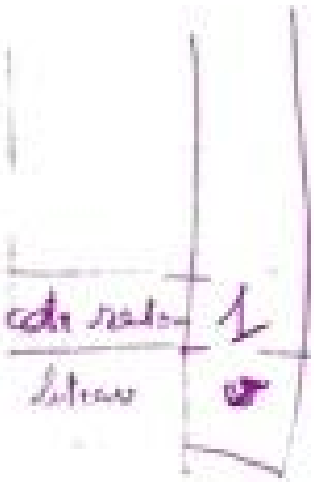
appel ret liste (adprog, adret)

sauve (l, litrav)
insert (l, adret)
sauve (litrav, liparam)
goto adprog



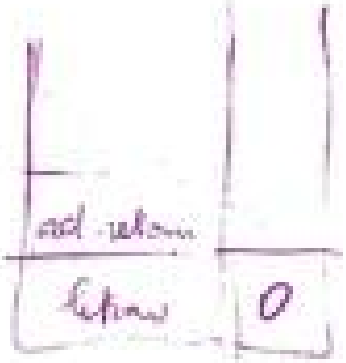
appel ret (adprog, adret, v)

sauve (l, litrav)
insert (l, adret)
insert (litrav, v)
insert (litrav, 1)
goto adprog



appel sans (adprog, adret)

sauve (l, litrav)
insert (l, adret)
insert (litrav, 0)
goto adprog



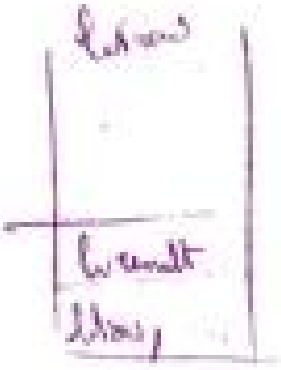
A noter que dans les 2 cas précédents nous avons mis 1 ou 0 dans litrav de façon à être cohérent avec le cas général.

Adret, l'adresse de retour est stockée juste derrière les variables de travail et va permettre de "remonter" à l'algorithme appellant

Pour les opérations suivantes nous aurons besoin de l'opération suivante qui vide complètement une pile

```
nettoie (l)
    trav ← card(l)
    for i = (1, 1, trav)
        do [ retire (l)
```

```
retour liste
    trav ← valpile (l)      } sauvegarde dans trav la valeur de la pile
    retire (l)              } sauvegarde dans trav la valeur de la pile
    nettoie (litrav)        } vide la pile litrav
    restaure (l, litrav)    } restaure la pile litrav dans la pile l
    sauve (litrav, liresult) } sauvegarde la valeur de retour
    goto trav
```



<u>retour (v)</u>

```
    trav ← valpile (l)
    retire (l)
    nettoie (litrav)
    restaure (l, litrav)    { trav ← valpile(l)
                            { retire (l)
                            { [illegible]
    insert (litrav, v)
    insert (litrav, 1)
    goto trav
```

<u>retoursans</u>

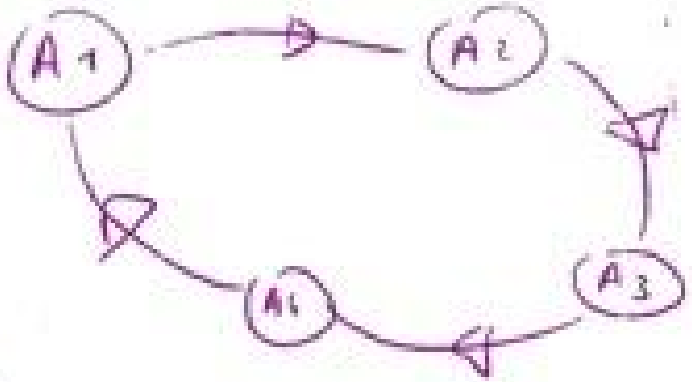
```
    trav ← valpile (l)
    retire (l)
    nettoie (litrav)
    restaure (l, litrav)
    insert (litrav, 0)
    goto trav
```

Le tableau ci dessous donne une récapitulation générale de toutes les opérations que nous venons d'introduire

(21)

opération	commentaires
insert (l, v)	ajoute un nouvel élément à une pile avec initialisation de cet élément à la valeur v
initpile (l)	vide complètement une pile
sauve (l1, l2)	vide la pile l2 dans la pile l1 en ajoutant la taille initiale de l2
restaure (l1, l2)	opération inverse de la précédente
appelretliste (adprog, adret)	appel de l'algorithme adprog et retour plus tard dans l'algorithme adret (avec une liste de paramètre dans laparam)
appelret (adprog, adret, v)	même opération avec un seul paramètre v
appelsans (adprog, adret)	même opération sans paramètre
retourliste	retour dans un algorithme avec une liste de résultat dans liresult
retour (v)	même opération avec ~~liresult~~ mise v
~~retourvide~~	même opération sans résultat

En multiprogrammation on a besoin (ici) de procédures (algorithmes) réentrantes lorsque l'on a des algorithmes récursifs c'est à dire telle que la série des appels que l'on peut trouver forme une boucle



On a l'impression que de tels appels vont à l'exécution produire un cycle infini : ce ci n'est pas sûr si les appels sont conditionnés (Remarque : A noter que ceci est une condition nécessaire pour que une série d'algorithmes ne boucle pas indéfiniment, nous savons (cf ch 0) que elle n'est pas suffisante)

En multiprogrammation (nous y reviendrons au § 2.3) on a besoin d'algorithmes réentrants dès que l'on veut les partager entre plusieurs utilisateurs —

Notons enfin que en mono programmation on a besoin d'une pile dès que l'on a des appels avec retour entre algorithmes (car il faut stocker quelque part les points de retour) - En fait dans bien des cas la pile "accompagne" l'execution des algorithmes - Un bon exemple de ceci est l'étude de l'exemple 5 page 12 (Non-reentrable save Area chaining) de l'exemple 9 page 16 (passing control with return) et de l'exemple 13 (page 18) (establishing a return code) du document IBM System/360 Operating System - Supervisor and Data management Services -

~~Nous allons maintenant donner~~
~~leurs exemples~~

A noter que dans ce dernier cas on a une pile mais elle est stockée dans les algorithmes eux même - Si

l'on veut rendre les algorithmes récursifs on doit fabriquer la pile à l'extérieur des algorithmes en utilisant les macros getmain et freemain (cf exemple page 12. Reentrable save area chaining)

Nous allons maintenant étudier 2 exemples de programmes récursifs pour illustrer l'emploi des opérations que nous venons d'introduire

Exemple 1

Soit l'algorithme suivant

```
fact (N)
        if N = 1 then 1
                 else N x fact (N-1)
```

On le "traduit" comme suit

```
fact : if valeur (litraw, 1) = 1
            then retour (1)
            else appel fact, α, valeur(litraw,1)-1)
α : retour (valeur (litraw,1) x valeur (litraw, 3))
```

l'appel de l'algorithme se fait comme suit

insert (trace, 0) → met 0 dans trace

appel ret (fact, fin, v) → met dans v la valeur de fact dans l → met dans trace la valeur de fact dans l

fin :

Trace

si l'on fait v = 4 on a la trace suivante

passage	l	l'trace
		#
→ fact	fin	4, 1
→ fact	3; fin, 4, 1	3, 1
→ fact	2; fin, 4, 1, 3, 2, 1	2, 1
→ fact	1; fin, 4, 1, 3, 2, 1, 2, 1	1, 1
← fact → d	2; fin, 4, 1, 3, 2, 1, 2, 1	2, 1, 1, 2
← fact → d	3; fin, 4, 1, 3, 2, 1	3, 2, 1, 1
← fact → d	4; fin, 4, 1	4, 1, 6, 1
fin		24, 1

Exemple 2 : cet exemple correspond à l'analyse descendante et à l'interprétation du langage défini par la grammaire suivante

Expr ::= Terme | Terme + Expr

Terme ::= Facteur | Facteur x Terme

Facteur ::= (Expr) | Nombre

On suppose que la chaine d'entrée se trouve dans une file appellée source l'écriture "Algol" est la suivante

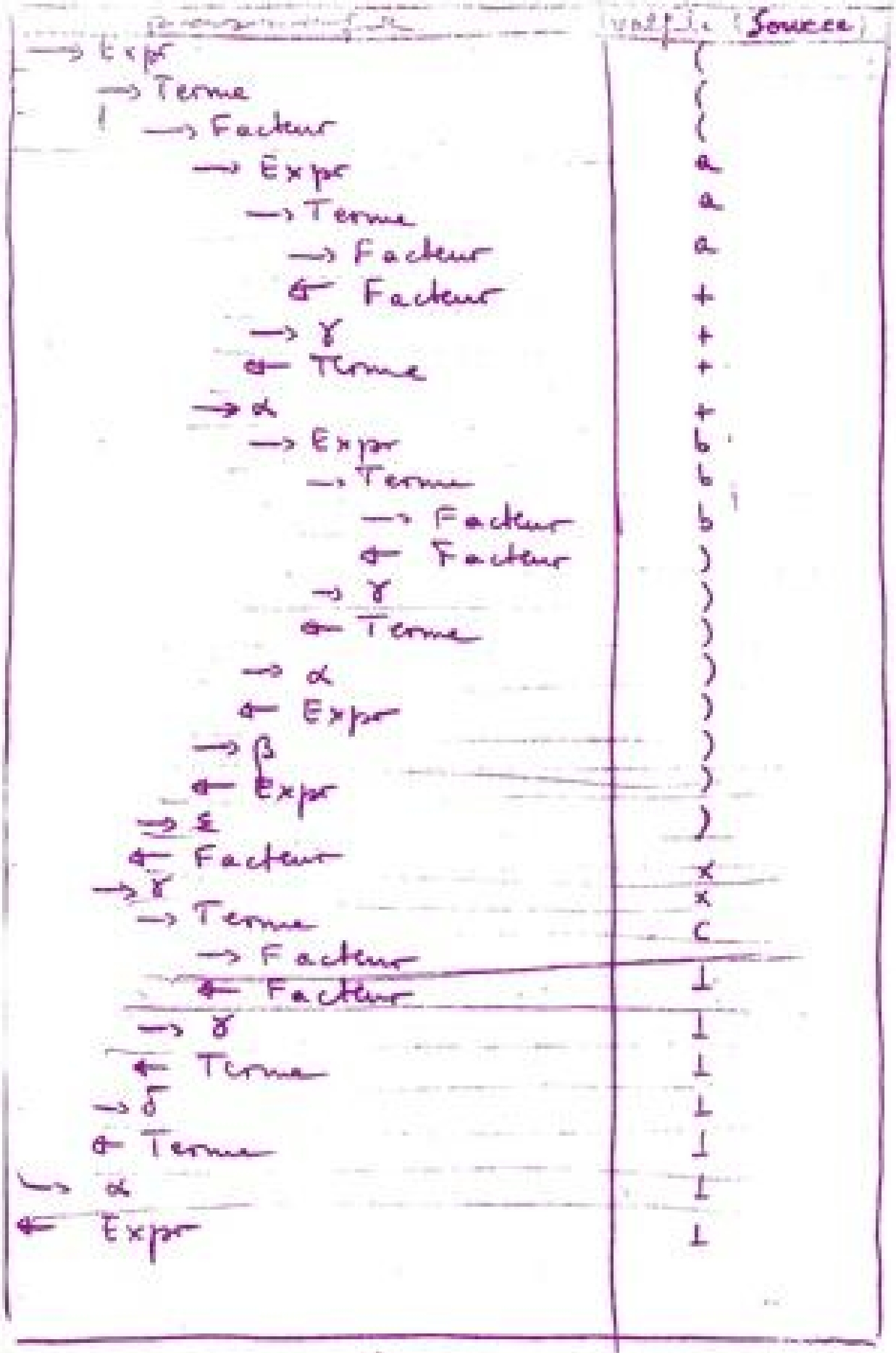
```
Expr
    trav ← Terme
    if symbole = '+'
        then [nouveau resultat ← trav + Expr
        else [resultat ← trav
Terme
    trav ← Facteur
    if symbole = 'x'
        then [nouveau resultat ← trav x Terme
        else [resultat ← trav
Facteur
    if symbole = '('
        then [nouveau trav ← Expr
```

```
        [ nouveau resultat ← trav
else    [ trav ← symbole
        [ nouveau resultat ← trav
```

de la "traduction" de cet ensemble d'algorithmes est la suivante :

```
Expr : appelretsaut (Terme, α)
α    : if valfile (source) = '+'
           then [ lire (source)
                [ appelretsaut (Expr, β)
       retour (valeur (litrav (1))
β    : retour (valeur (litrav, 1) + valeur (litrav, 2))
Terme: appelretsaut (Facteur, γ)
γ    : if valfile (source) = 'x'
           then [ lire (source)
                [ appelretsaut (Terme, δ)
       retour (valeur (litrav, 1))
δ    : retour (valeur (litrav, 1) x valeur (litrav, 3))
Facteur: if valfile (source) = '('
           then [ lire (source)
                [ appelretsaut (Expr, ε)
       ranger (litrav, valfile (source))
       lire (source)
       retour (valeur (litrav, 1))
ε    : lire (source)
       retour (valeur (litrav, 1))
```

La trace de cet algorithme pour la chaine source (a + b) x c ⊥ est la suivante



$(a \times b) + c \cdot \perp$

2.3 File d'attente

Comme la pile, la file d'attente est un des mécanismes les plus répandus en programmation système.

Au paragraphe précédent nous avons vu comment à l'aide d'une pile nous pouvions assurer la réentrance d'un algorithme.

Au cours de l'exécution du dernier exemple nous avons rencontré plusieurs fois l'instruction "Tire (source)". Il se peut que la liste linéaire "source" ne soit pas en mémoire: il est nécessaire alors d'effectuer une opération d'entrée pour aller chercher le premier élément de "source". Pendant ce temps l'exécution doit évidemment s'arrêter: l'exécution attend. Au lieu de stopper l'unité

l'entrée, on peut mettre à profit cette attente pour lui faire exécuter un autre ensemble d'algorithmes (par exemple le même mais pour une autre liste d'entrée forcée) - Cette nouvelle exécution va elle-même être bloquée à son tour et on peut mettre à profit etc...

On voit donc qu'il va se former des files d'attente d'exécution bloquée

Nous allons formaliser un peu ces phénomènes dans les pages qui suivent en introduisant d'abord un certain nombre de définitions puis de nouvelles opérations élémentaires -

Définitions

@ Client : Nous appellons CLIENT l'exécution d'un certain nombre d'algorithmes - Un client porte un nom et est associé à une pile. Cette

Pla s'appelle la contexte du client

(b) Point de service : un POINT DE SERVICE est un "endroit" où un client peut être servi. Plusieurs clients peuvent vouloir se faire servir au même point de service. Un client est servi à un point de service par un SERVEUR (Voir c). Il peut y avoir plusieurs serveurs par point de service. Et un point de service on aura 3 files d'attente

(b1) une file pour client (point de service) qui est la file d'attente des nouveaux clients arrivant à ce point de service

(b2) une file pour client (point de service) qui est la file d'attente de clients qui avaient commencé de se faire servir par un serveur du point de service puis qui étaient partis se faire servir à un autre point de service

et fini maintenant revient (3x) continuer de se faire servir ~~=~~ à ce point initial -

(63) Une file serveur libre (point de service)

c'est la file des serveurs de ce point de service qui n'ont rien à faire (qui n'ont pas de client à servir)

Un serveur d'un point de service peut n'avoir rien à faire pour deux raisons (additionnelles)

(α) le nombre de clients s/v à ce point de service est nul : N(point de service) = 0.

(β) le nombre de clients dont on a déjà commencé le service à ce point de service a dépassé un seuil fixé à l'avance - Le seuil s'appelle le degré de multiprogrammation max.

mmm : DM (point de service) est une constante du point de service : le nombre de client dont on a déjà commencé le service (et qui n'est pas fini) s'écrit D (point de service)

les 2 files (b1) et (b2) (pleine client bi et file vide client) sont situées dans une salle d'attente. Pour des raisons qui apparaîtront plus clairement par la suite, cette salle d'attente peut être fermée à clé de temps en temps pour certaines périodes très courtes.

On a donc :

salle d'attente (point de service) → 0 si ouverte
→ 1 si fermée à clé

(c) Serveur Un SERVEUR est associé par construction à un point de service. Le nombre de serveurs associés à un point de service est une constante fini d'appelle le degré de multi-traitement maximum : DT (point de service)

Un serveur peut être en train de dormir s'il appartient à la file serveur libre de son point de service

On a donc :

dort (serveur) → 1 s'il dort
→ 0 s'il sert un client

Le point de service auquel est associé un serveur s'appelle service (serveur) -

On a donc la règle

si dort(serveur) = 1 alors

serveur ∈ file serveur libre(service(serveur))

Pour des raisons qui apparaîtront plus clairement par la suite, chaque serveur veut pouvoir être tranquille (non dérangé) pendant certaines périodes de temps bien comptés.

On a donc

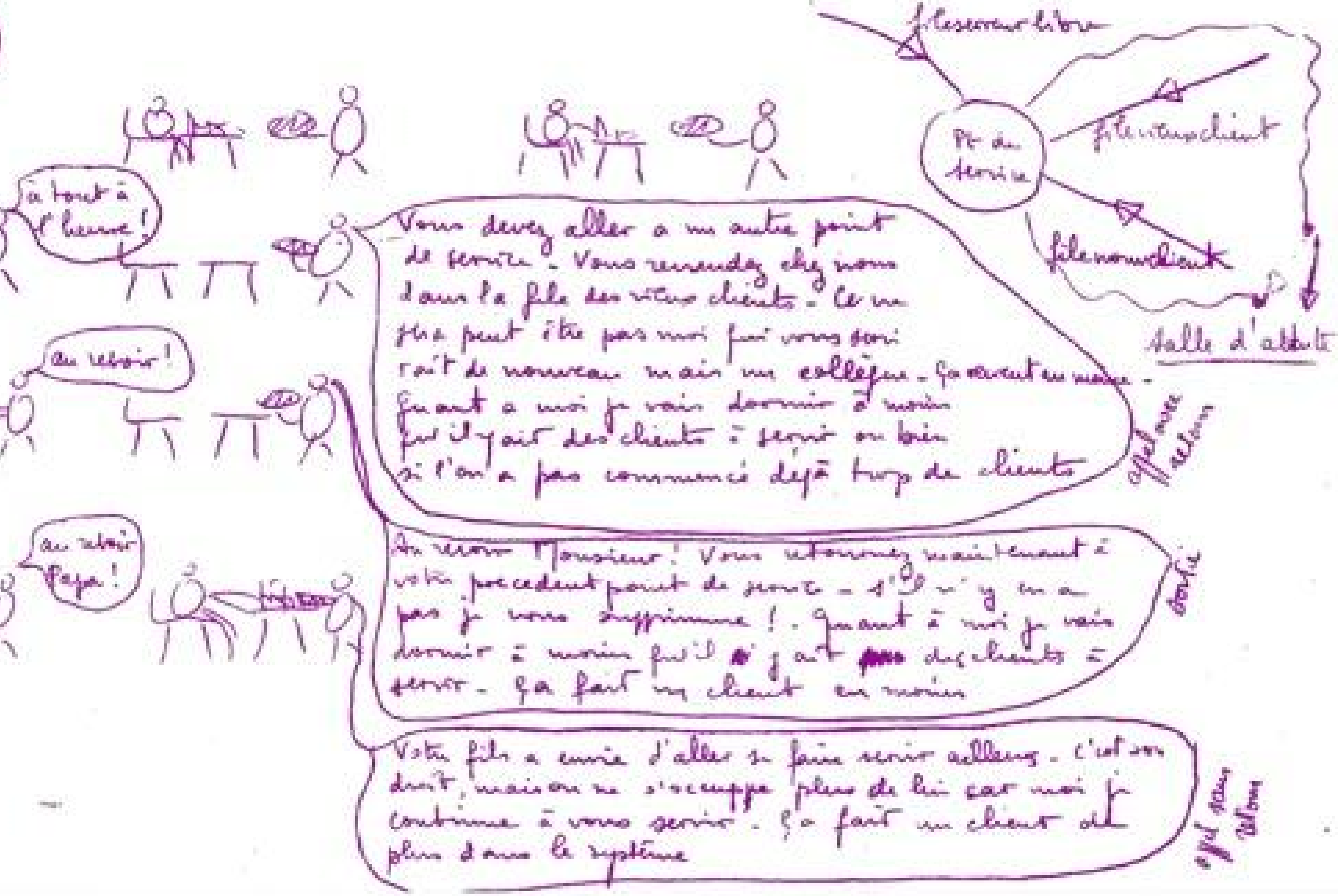
attention(serveur) → 0 peut être dérangé
→ 1 ne déranger sous aucun prétexte

Enfin lorsqu'un serveur ne dort pas c'est qu'il sert un client. Le client est client(serveur) -

Le tableau ci dessous récapitule la structure de données que nous venons d'introduire -

Élément	Propriété	Rôle ou fonction
client		contexte d'exécution d'algorithmes
point de service	salle d'attente	o/n ouverte ou fermée
	file_nouveau_clients	clients nouveaux en attente
	file_ancs_clients	clients anciens en attente
	file_serveur_libre	serveurs libres dont (nom)
	DM	Nombre maximum de clients que l'on peut commencer de servir. Degré de Multiprogrammation
	DT	Nombre de serveurs. Degré de multitraitement
	D	Nombre de clients que l'on a commencé de servir
	N	card (file_nouveau_clients) + card (file_ancs_clients) + nombre de client servi
serveur	attention	o/n on peut déranger ou ne pas déranger
	dort	o/n est en client en dort
	client	client du serveur
	service	point de service du serveur

Le schéma ci dessous concrétise quelques unes des activités à un point de service


file serveur libre
Pt de service
file vieux client
file nouv client
salle d'attente
à tout à l'heure !
au revoir !
au revoir Papa !
Vous devez aller à un autre point de service. Vous revendez chez vous dans la file des vieux clients. Ce ne sera peut être pas moi qui vous servira de nouveau mais un collègue. Ça revient au même. Quant à moi je vais dormir à moins qu'il y ait des clients à servir ou bien si l'on a pas commencé déjà trop de clients
appel avec retour
Au revoir Monsieur ! Vous retournez maintenant à votre précédent point de service. S'il n'y en a pas je vous supprime ! Quant à moi je vais dormir à moins qu'il y ait des clients à servir. Ça fait un client en moins
sortie
Votre fils a envie d'aller se faire servir ailleurs. C'est son droit, mais on ne s'occuppe plus de lui car moi je continue à vous servir. Ça fait un client de plus dans le système
appel sans retour

Avant de définir avec précision les opérations de base correspondant aux "conversations" du schéma ci dessus, il convient de prévenir l'usage abusif des salles d'attente : un seul client à la fois n'est autorisé à entrer dans une salle d'attente —

De façon à garantir ce ci nous définir sons deux opérations :

bloq (porte) et débloq (porte)

bloq (porte)

K: cc ← porte
porte ← 1

Test & set

if cc = 1 then goto K

Les deux opérations cc ← porte et porte ← 1 sont faites de façon ininterruptible c'est à dire qu'un seul serveur ne peut la faire à la fois —

En 360, ces 2 opérations accolées s'appellent Test & Set — Pour de plus amples détails

voir le document IBM System /360 Operating System : Principle of Operation page –

d'opération dblq est triviale.

dblq (porte)

porte ← 0

Nous allons maintenant écrire les algorithmes des 3 opérations décrites dans le schéma :

Appel avec retour (serveur, service appelé)

Sortie (serveur, service retour)

Appel sans retour (serveur, service appelé)

On suppose pour chacun des 3 cas, un certain nombre d'opérations de routine du genre sauve ou restaure qui ont en lien juste avant ces opérations et qui rendent le contexte du client cohérent avec elles –

Appel avec retour (serveur, service appelé)

attention (serveur) ← 1

— blq (salle d'attente (service appelé))

— insert (file nouv client (service appelé), client (serveur))

— N (service appelé) ← N (service appelé) + 1

if D (service appelé) < DM (service appelé)
and card (file serveur libre (service appelé)) ≠ 0
then
  dont (val file (file serveur libre (service appelé)))
    ← 0
  tire (file serveur libre (service appelé))
  débq (salle d'attente (service appelé))
  goto a

débq (salle d'attente (service appelé))

a: blq (salle d'attente (service (serveur)))

— N (service (serveur)) ← N (service (serveur)) − 1

if N (service (serveur)) = 0 or D (service (serveur))
  = DM (service (serveur))
then
  dont (serveur) ← 1
  ...

attente (serveur) ← 0
goto scheduler

goto scheduler

Sortie (serveur, service retour)

attente (serveur) ← 1
if service retour = 0
then [supprimer (client (serveur))
goto B
bloq (salle d'attente (service retour))
insert (file vide client (service retour), client (serveur))
N (service retour) ← N (service retour) + 1
if card (file serveur libre (service retour)) ≠ 0
then [dort (valide (file serveur libre (service retour))
← 0
tire (file serveur libre (service retour))
debloq (salle d'attente (service retour))
goto B
debloq (salle d'attente (service retour))
B: bloq (salle d'attente (service serveur))
N (service (serveur)) ← N (service (serveur)) − 1
D (service (serveur)) ← D (service (serveur)) − 1

if N(service(serveur)) = 0

Hmm [dont(serveur) ← 1
débloq(salledattente(service(serveur)))
attention(serveur) ← 0
goto scheduler]

else goto scheduler

appelsansretour(serveur, service appelé)

client ← créer
attention(serveur) ← 1
bloq(salledattente(service appelé))
insert(fileserveurclient(service appelé), client)
N(service appelé) ← N(service appelé) + 1
if D(service appelé) < DM(service appelé)
and card(fileserveurlibre(service appelé)) ≠ 0
Hmm [dont(valfile(fileserveurlibre(service appelé)) ← 0
tête(fileserveurlibre(service appelé))
débloq(salledattente(service appelé))
goto 8]
débloq(salle d'attente(service appelé))
8 : attention(serveur) ← 0

scheduler (serveur)

blq (dort (serveur))
attention (serveur) ← 1
blq (salle d'attente (service (serveur)))

schedsmi : dort (serveur) ← 0
if card (filevieuxclient (service (serveur)) = 0
then client ← choixde (service (serveur),
filenouvclient (service (serveur)))
enleve (filenouvclient (service (serveur)),
client)
client (serveur) ← client
D (service (serveur)) ← D (service (serveur))
+ 1
else client ← choixde (service (serveur),
filevieuxclient (service (serveur))
enleve (filevieuxclient (service (serveur)),
client)
dblq (salle d'attente (service (serveur)))
attention (serveur) ← 0
sert (client (serveur))

Analyse du scheduler

Nous allons "suivre" un serveur -

Lorsque dort(serveur) = 1, ce serveur tourne en rond sur l'instruction Blq(dort(serveur)). Dès que dort(serveur) passe à la valeur 0 ce serveur sort de la boucle -

⇒ Dort(serveur) ← 0 est une instruction qui fait se passer un certain temps dans appel retour en appel sans retour (lorsque D < Dm et que la file serveur libre n'est pas vide) pour le service qui soit dans sortie pour le service retour

· Dès que le serveur peut devenir actif il bloque la salle d'attente de son service et choisit un client soit dans file serveur client soit dans file service client (il essaie d'abord d'en

choisir un dans file vieux client)
Le choix est une question de politique
qui peut varier d'un ~~~~ point de
service à l'autre -

Avant de servir son client le serveur débloque la salle d'attente -
Si le serveur a pris un client dans file nouveau client : augmente de 1 le nombre de clients commencés pour son point de service -
Si au cours du travail, le client doit s'en aller (avec retour) vers un nouveau point de service, son serveur le place dans la file d'attente des nouveaux clients de ce ~~~~ point de service et si les conditions sont remplies il réveille l'un des serveurs endormis de ce point de service
Celui ci peut alors prendre en charge immédiatement le client - Quant au serveur qui vient de perdre son client

il peut eventuellement aller s'endormir ou au contraire reprendre immédiatement un autre client (schedown)

Si la fin du travail arrive (sortie) le serveur renvoie eventuellement (sinon il le tue) son client à son point de service précédent et il va lui même s'endormir ou prendre un autre client.

Les operations consistant à endormir ou a reveiller un serveur sont standards nous pouvons donc les écrire :

```
endort(serveur)
    dort(serveur) ← insert(fileserveurlibre(service(serveur)), serveur)}
    dltq(salledattente(service(serveur)))
    attention(serveur) ← 0
    goto scheduler
```

et de même :

réveille (point de service, retour)

dort (valfile (filedeserveurlibre (point de service)))

⟵ 0

tire (filedeserveurlibre (point de service))

dlibq (salledattente (point de service))

goto retour

On peut introduire de nouvelles opérations communes :

parcontrole (serveur, service appelé)

[première partie de Appelavecretour (jusqu'en α)]

[deuxième partie de sortie (à partir de β)]

Antispeciale (serveur, service retour)

[première partie de sortie (jusqu'en β)]

[deuxième partie de Appelavecretour (à partir de α)]

→ V (pointdeservice)

blq (salledattente (pointdeservice))

D (pointdeservice) ← D(pointdeservice) − 1

if N(pointdeservice) > 0 and

Card (fileserveurlibre(pointdeservice)) ≠ 0

then reveille (pointdeservice, δ)

δ :

On reconnait ici l'implantation de l'operation V de Dijkstra -

Pour l'operation P il suffit de faire

: Appelaveceretour (serveur, pointdeservice)

le travail fait a ce point de service etant alors sortie speciale en plus haut

Pour plus de detail sur les operations P et V de Dijkstra voir l'article Co-operating Sequential processes du livre

Programming Languages (F. Genuys) (48)
(Académic Press 1968)

Tout ce que nous avons dit jusqu'à maintenant supposait que les serveurs dont il était question travaillaient - En fait dans la réalité ceci n'est pas vrai. Car pour travailler un serveur quel qu'il soit a besoin d'un hardware -

De plus il serait maintenant économique de mobiliser un hardware pour un serveur qui bouclerait sur la première instruction de schéduler (Attq (dont (serveur)))

Nous sommes donc obligé d'organiser les serveurs en file d'attente (en attente du hardware)

En fait nous n'avons pas qu'un hardware mais plusieurs classes de hardware qui regroupent plusieurs hardwares

identiques -

Nous allons donc introduire les nouvelles structures suivantes

ⓐ classedehardware

Une classe de hardware est constituée par une file hard (classedehardware) qui est une liste des hardwares qui constituent la classe - Les serveurs qui (de la classe) attendent l'un quelconque des hardware forment une file de garçon (classedehardware)

Cette file de garçon est dans une salle des gardes dont on veut pouvoir bloquer la porte pendant un temps très court - Enfin il existe un organe charger de faire fonctionner cette classe de hardware c'est le Regulateur (classedehardware) Pour travailler il a besoin d'un indicateur appelé tourniquet (classedehardware)

ⓑ hardware

Chaque hardware lorsqu'il travaille est

emprunté par un serveur fini est
emprunteur (hardware) et de plus
il a un compteur d'adresse (hardware) -

ⓒ serveur On ajoute à la structure de serveur fini nous avons déjà construit les éléments suivants :

adr (serveur) fini contient un point de reprise pour ce serveur (une adresse) et hard (serveur) fini indique le hardware emprunté par ce serveur - On a :

emprunteur (hard (serveur)) = serveur
hard (emprunteur (hardware)) = hardware

ⓓ Point de service On ajoute à la structure de point de service fini nous avions déjà constitué l'élément

classe (point_de_service) fini indique la classe de hardware dont se servent les serveurs de ce point de service -

Les différents programmes fini nous avons écrit sont enchaînés sauf en dort et réveille fini alternativement :

endort (serveur)

dort (serveur) ← 1
insert (fileserveurlibre (service (serveur)), serveur)
dblq (salledattente (servi (serveur)))
attention (serveur) ← 0
dblq (tourneurond (classe (servi (serveur))))
goto scheduler

Reveille (point de service, retour)

garçon ← valfile (fileserveurlibre (point de service))
dort (garçon) ← 0 réveille garçon
tire (fileserveurlibre (point de service)) enlève
dblq (salle d attente (servi (serveur)))
attention (serveur) ← 0 par sécurité
blq (salle de garde (classe (point de service)))
ajoute (file garçon (classe (point de service)), garçon)
dblq (salle de garde (classe (point de service)))
dblq (tourneurond (classe (point de service)))
attention (serveur) ← 1 on perd la tâche
goto retour

Le Regulateur est alors

Regulateur (classe de hardware)

```
1: blq (tourneurond (classe de hardware))
   blq (salledesfardes (classe de hardware))
   meilleurgarçon ← choixtenseur (filegarçon (classedehardware))
   moinsbonhardware ← choixhardware (flehard (classedehardware
   if digne (meilleurgarçon, moinsbonhardware)
   then sacrifié ← tamponneur (moinsbonhardware)
        blq (attention (sacrifié))
        STOP (moinsbonhardware)
        dblq (attention (sacrifié))
        adr (sacrifié) ← compteur d'adresse (
                         moinsbonhardware)
        compteur_d'adresse (moinsbonhardware) ←
                         adr (meilleurgarçon)
        tamponneur (moinsbonhardware) ← meilleurgarçon
        enleve (filegarçon (classe de hardware), meilleurgarçon)
        ajoute (filegarçon (classe de hardware), sacrifié)
        DEMARRE (moinsbonhardware)
   dblq (salledesfardes (classe de hardware))
   goto 1
```

Analyse du Régulateur (classe de hardware)

Nous allons suivre un hardware dont le Régulateur est développé (tourne rond ⟵ ○ sur intervention de endort ou réveille), il bloque la file des gardes et joue à deux algorithmes qui définissent sa politique : il choisit parmi [les serveurs qui attendent un hardware dans file de garage celui qui lui paraît le meilleur] (meilleur garage) et il choisit parmi tous les hardwares celui qui lui semble le moins bon (par exemple parce qu'il travaille avec un serveur d'un point de service peu important ou pour tout autre raison dépendant de sa politique).

Il regarde ensuite si le meilleur garage est digne du moins bon hardware (c'est encore une fonction de politique). Dans l'affirmative, il va sacrifier l'emprunteur

du nouveau hardware, c'est à dire
fait le ~~redémarrage~~ dans file de fonction -
Pour cela il arrête le nouveau hardware
lui change son serveur et le redemarre -
Enfin Regulateur debloque la salle
des gardes puis boucle sur R -
Le Regulateur comme nous l'avons
dit peut être (indirectement) activé
soit par endort soit par reveille.
Lorsqu'il est activé par endort : il
est evident que le Regulateur aura
Tendance à choisir comme nouveau
hardware celui qui jusqu'à
pour ~~serveur~~ emprunteur le serveur que
l'on vient d'endormir - c'est pourquoi
on ne va pas tourner sur file (dont
(serveur)) dans scheduler -
Lorsqu'il est activé par réveille, le
regulateur remet eventuellement en
cause l'affectation des hardwares aux
serveurs en tenant compte du nouveau
serveur qui vient d'entrer dans file de pour

On remarque que dans le Regulateur (15)
on ne stoppe le hardware sacrifié lors-
lorsque l'operation blq (attention (sacrifié))
le permet. Ceci est necessaire afin
d'eviter que on ne mette dans le degats un
un serveur qui a bloqué une salle d'attente (point de sortie) car alors le point
de sortie serait inutilisable jusqu'à
ce que le serveur sacrifié emprunte
de nouveau un hardware —

Il faut remarquer que le blocage
sur attention (sacrifié) ainsi
que le blocage sur salle des gardes
n'entraine pas d'etreinte fatale
entre le Regulateur et un hardware
car dans cette
dernière operation on a pris soin
de faire attention (serveur) ⟵
(ce qui debloque eventuellement le
Regulateur) juste avant de faire
blq (salle des gardes) —

Il faut noter la très forte analogie qui existe entre les éléments du tableau ci dessous

Point de service	classe de hardware
serveur	hardware
client	serveur
file nouveau client et file attente client	file de paquets
file serveur libre + autre serveur du point de service	file hard
appel	réveil
sortie	endort
salle d'attente	salle des paquets
Scheduler	Régulateur
choix client	choix serveur

Nous allons enfin introduire une dernière opération qui permet à un serveur de changer de classe de hardware

```
change (serveur, classe1, classe2, adresse)
    lmpermuteur (hard (serveur)) ← 0
    blq (salle des gardes (classe 2))
        ajoute (file d'attente (classe2), serveur)
        adr (serveur) ← adresse
    dblq (salle des gardes (classe 2))
    dblq (tournevis (classe 2))
    dblq (tournevis (classe 1))
    goto 0
0 : goto 0
```

On suppose un serveur travaille avec un hardware de la classe 1 et un qui est pour une raison ou une autre plus autorisé à travailler avec lui (par exemple le hardware n'est pas assez puissant : il n'a plus le répertoire d'instruction suffisant pour que le serveur puisse continuer à

travailler de façon satisfaisante pour son client -). Le serveur va alors changer de classe de hardware pour emprunter un exemplaire de la classe 2 — Les opérations de changement terminées les deux régulateurs sont informés — Le Hardware de la classe 1 qui a perdu son serveur n'a alors plus rien d'autre à faire que à bouder — On se doute bien que le Régulateur de la classe va le désigner tout de suite comme moins bon hardware et que il va rapidement faire un travail plus profitable à la communauté des serveurs et des clients — Notons enfin que il peut être possible à un organe extérieur aux hardwares et aux Régulateurs de faire de temps en temps l'opération d'échange (transparente) de classe de hardware pour l'une quelconque des classes de hardware

Le Régulateur ainsi défini peut ! remettre en cause les affectations de ses ~~[illegible]~~ hardwares aux serveurs. Lorsque l'organe extérieur fini accède aux différentes pour une certaine classe de hardware est une horloge travaillant à intervalle régulier on dit que les hardwares de cette classes travaillent en time slicing (tranche de temps)

Dans la pratique des systèmes existant actuellement on trouve généralement 2 classes de hardwares :

- les Unités Centrales
- les Canaux

ⓐ Unité Centrale

Nous allons d'abord considérer le cas d'une installation (3e génération) n'ayant qu'une seule Unité Centrale.

Le problème est compliqué par le fait

- Que une Unité Centrale cache en fait 2 hardwares distincts (ou plus) :
  - l'Unité Centrale en mode normal
  - l'Unité Centrale en mode maître

- Que les Régulateurs de chacune de ces 2 classes de hardware (mode normal, mode maître) s'exécutent sur l'Unité Centrale elle même et non dans un organe indépendant comme nous l'avons décrit dans le système "théorique".

L'opération :

Change (serveur, mode normal, mode maître, adresse)

s'appelle un appel superviseur (SVC)

L'opération :

Change (serveur, mode maître, mode normal, adresse d'accès)

s'appelle un changement d'état programme (MMU)

En résumé on a 4 organes distincts

qui se partagent l'usage de l'Unité (16)
centrale

- le Régulateur (mode-maître)
- le Régulateur (mode-normale)
- l'Unité hard (mode-maître)
- l'Unité hard (mode normal)

Bien entendu ces organes sont en compétition pour utiliser l'Unité Centrale : il existe donc une hiérarchie entre ces organes qui est la suivante

① Régulateur (mode-maître)
② hard (mode-maître)
③ Régulateur (mode-normal)
④ hard (mode-normal)

On s'aperçoit que l'on a ici une complication assez arbitraire qui tend d'ailleurs à disparaître

Dans le cas d'un système avec plusieurs Unité Centrales il n'est plus possible de faire exécuter les Régulateurs sur une

Unité Centrale (laquelle ?) - On voit alors apparaître un ~~[illegible]~~ organe indépendant qui est le Régulateur - C'est le cas par exemple des 360/67 biprocesseur ou de l'IRIS-80 (CII)

⑥ Canal

L'"histoire" du canal depuis la 1ère génération jusqu'à nos jours -- serait à rapprocher de l'histoire d'un organe dans une espèce vivante sous la pression de l'évolution sélective -

On peut voir comment le canal très ~~[illegible]~~ simplifié et totalement incorporé à l'Unité Centrale lors de la 2ème génération a toujours tendu à prendre de plus en plus son indépendance -

Si nous observons où en est l'évolution du canal dans un système 3ème génération

classique (IBM/360) nous voyons qu'il est un organe hybride qui n'a pas encore réussi à se séparer totalement de l'Unité Centrale et qui doit encore s'en servir.

Autrement dit nous ne pouvons pas (malheureusement) identifier un Canal à un pur de service c'est pourquoi nous sommes obligé de prati quer des changements de hardware comme

Charge (serveur, mode maître, canal, adresse)

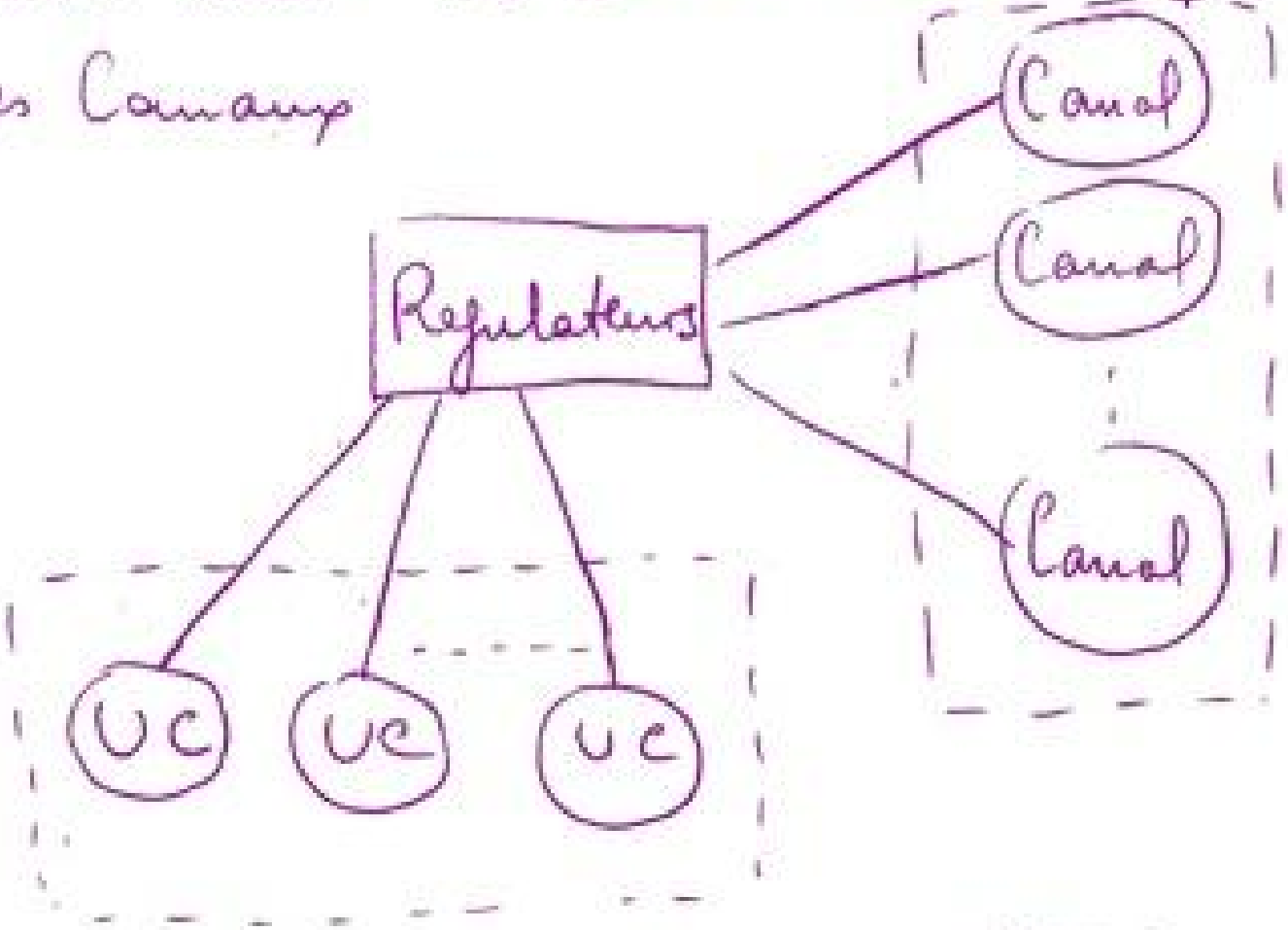
Ceci s'appelle généralement un START I/O (SIO)

et comme

charge (serveur, canal, mode maître, adresse)

Ceci s'appelle généralement une interruption Canal

Mais l'evolution qui va dans le sens de une identification des paramètres Canaux à une seule point de service est irreversible en même temps que les travaux de Regulation se regroupent dans un même organe chargé de réguler aussi bien les Unités Centrales que les Canaux



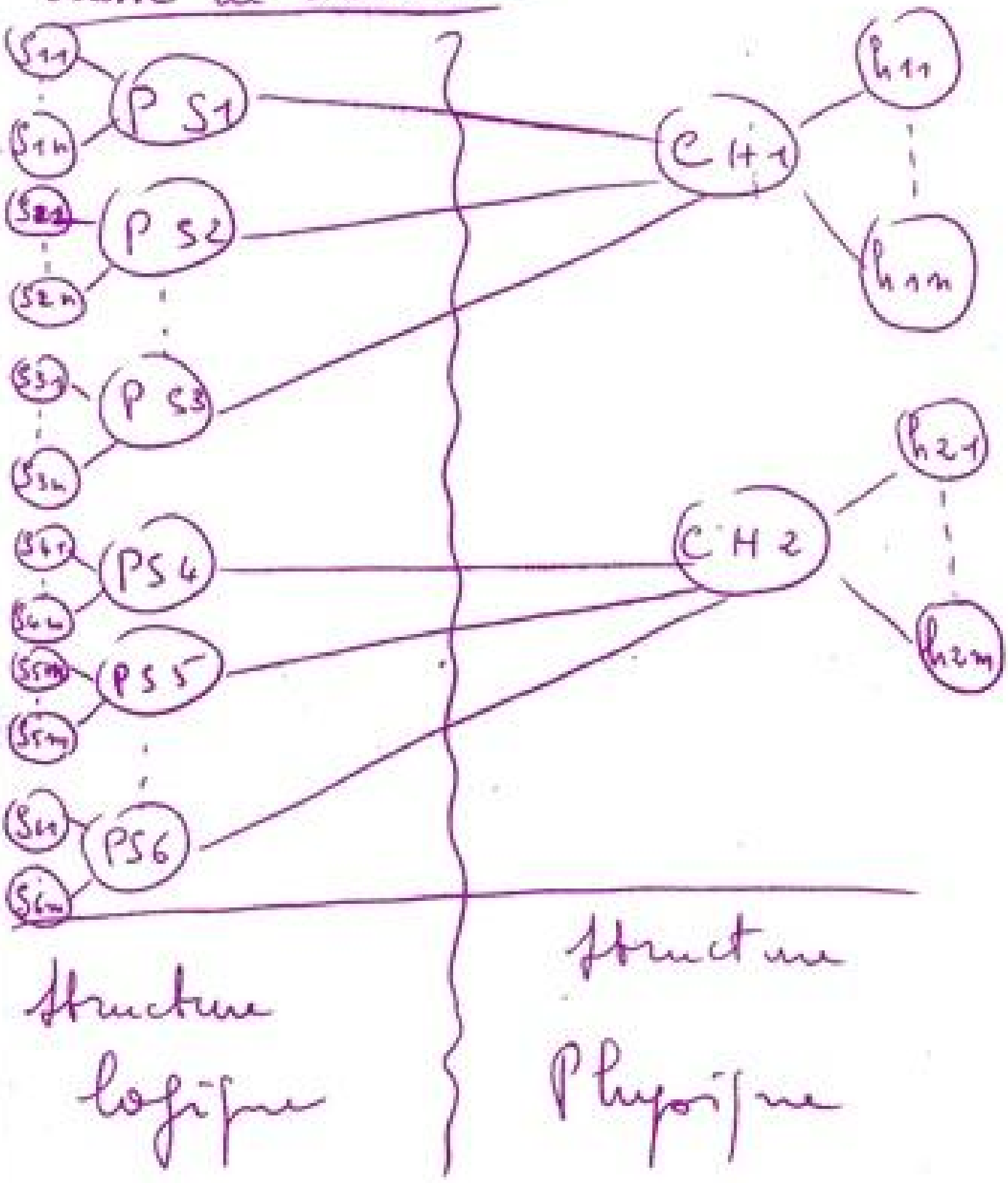
Avec la tendance vers ce schéma disparaît de plus en plus le besoin des opérations changement (fermeture, ouverture, adressage). On tend donc à avoir des systèmes

Un fonctionnement sans interruption

Dans de tels systèmes à un point de

temps il ne correspond qu'une seule

classe de hardware



le nombre de serveurs d'un point de fenêtre (degré de multitraitement) est égal au nombre de hardware de la classe mixte correspondant à ce point de service (Voir schema : m, n)

Lorsque un client (on appelle parfois dans la littérature un client un processus) passe d'un point de service PSi à un point de service PSj on effectue éventuellement l'opération réveille (PSj, retour) (Voir cette opération page 151) - On voit par là au cours de cette opération on place un serveur endormi dans la file d'attente de la classe de hardware de PSj (opération soulignée plus bas) débloque le régulateur de cette classe de hardware. Si le serveur réveillé est déjà d'une autre hardware de la classe il va effectuer

1 scheduler (puisqu'il dormait il
pseudo-tournait sur blq (dort (serveur)))
se débloquer (puisqu'il est réveillé)
et éventuellement (changements dans
1 scheduler page 142) choisir ce
client - (il sera distant page choisi + tard)

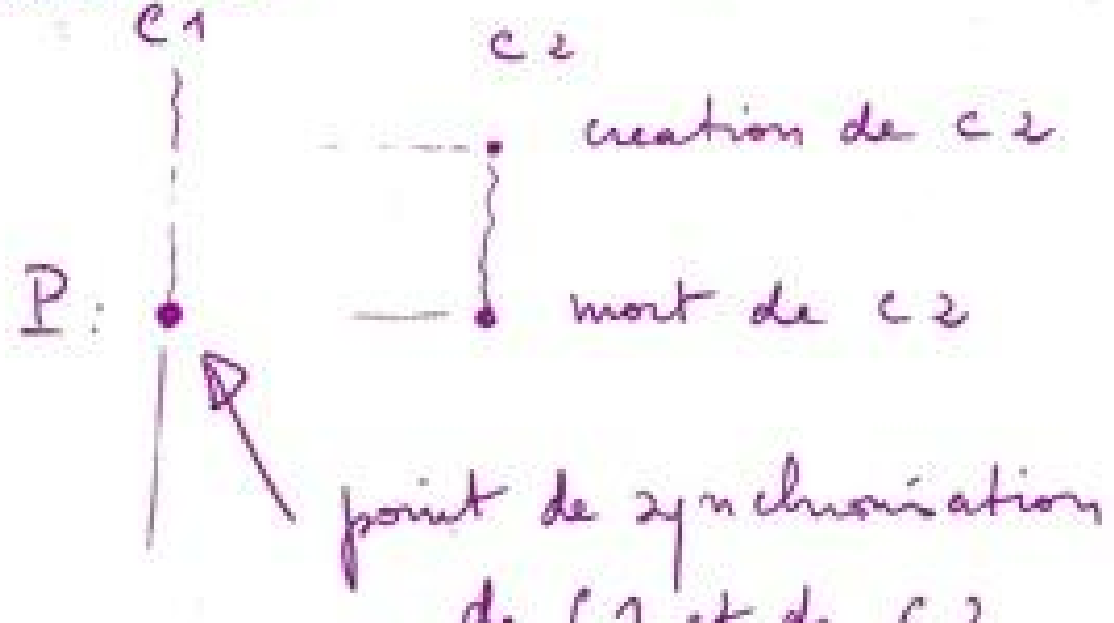
On voit donc que le changement
de hardware pour un client
se fait "en douceur" à l'occasion d'un
changement de serveur et non par une
opération change beaucoup plus brutale -
à noter que dans ce cas on peut
appeler un serveur un hardware
(ou processeur) virtuel -

Problème de synchronisation

Nous allons enfin considérer un dernier
problème : celui de la synchronisation
de 2 clients

Supposons qu'un client c1 soit servi à un point de service PS1 et qu'il crée un autre client c2 (comme dans un appel sans retour) pour être servi à un 2e point de service PS2.

On veut alors pouvoir synchroniser les deux clients c1 et c2 c'est à dire qu'une fois que c2 est créé on veut que c1 puisse continuer jusqu'à un certain point P à partir duquel il est nécessaire que c2 soit mort (qu'il ait fait SORTIE) pour continuer



Deux cas sont alors possible

(a) C1 arrive au point de synchronisation alors que C2 est déjà mort: C1 peut donc continuer sans plus attendre

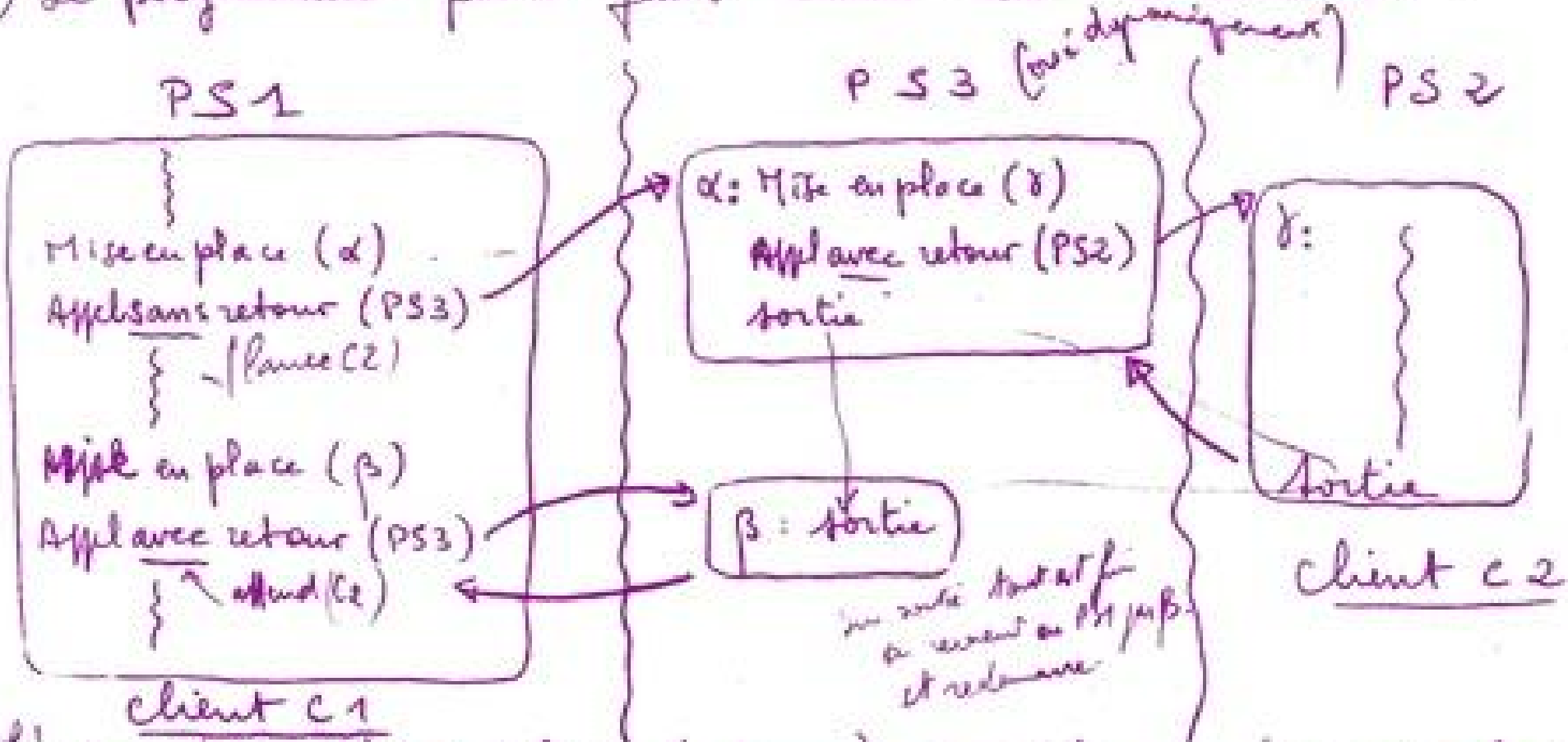
(b) C1 arrive au point de synchronisation alors que C2 n'est pas encore mort - C1 doit attendre.

Dans le cas (b), l'attente de C1 doit avoir lieu comme toute les phases d'attente dans une file, or toutes les files d'attente de client n'existent que par rapport à des points de service

On voit donc la necessité d'avoir un point de service supplementaire PS3 qui sert à synchroniser C1 et C2

Ce point de service PS3 doit avoir un degré de multiprogrammation maximum egal à 1 : DM(PS3) = 1

(170) Le "programme" que il faut écrire est le suivant :



l'opération Mise en place (adresse) consiste à insérer dans la pile du client l'adresse d'un programme - (cf § 2.1)

On voit sur cet exemple que il est nécessaire que le point de service PS3 ne dépende que du client C1 et non

du programme exécuté par le serveur (171)
de C1.

Ceci amène deux opérations supplémentaires :

- Creerps
- Supprimerps (point de reprise)

qui permettent de créer et supprimer dynamiquement des points de reprises.

2.4 : Allocation des listes linéaires

Nous allons décrire différentes méthodes d'allocation de listes linéaires en faisant différentes hypothèses quant à la structure de la mémoire.

Nous supposerons d'abord que nous avons une mémoire homogène (un seul algorithme de sélection) adressable au niveau d'atome de mémoire que nous appellerons des mots (sans préciser leur taille). Nous supposerons de plus les listes que nous voulons allouer sont des suites de mots : un élément de la liste (élément log.) correspond à un atome de mémoire (élément physique)

Dans la 2e hypothèse nous supposons que la mémoire n'est plus homogène elle comporte 2 niveaux ; le premier est analogue au précédent et le second plus grand n'est pas directement interprétable par le hardware mais doit au préalable transférer une partie de son contenu dans le premier.

A noter que ceci n'est pas toujours le cas dans les hiérarchies de mémoires : les "extended core memories" constituent des mémoires d'un niveau inférieur à celui des mémoires tore et pourtant leur contenu est directement interprétable par le hardware - Le système time-sharing ACME de l'hopital

universitaire de Stanford utilise une taille mémoire étendue en fonction de mémoire secondaire rapide — (8 µsec de temps d'accès)

2.4.1 Plein mémoire homogène

Nous allons étudier 7 méthodes différentes et les comparer sur 6 critères différents qui sont les suivants

- facilité d'allocation
- facilité de désallocation
- indexation / Recherche
- ajout (d'éléments) au milieu de la liste
- place occupée
- possibilité d'extension en mémoire secondaire (étudié plus en détail au paragraphe 2.4.4)

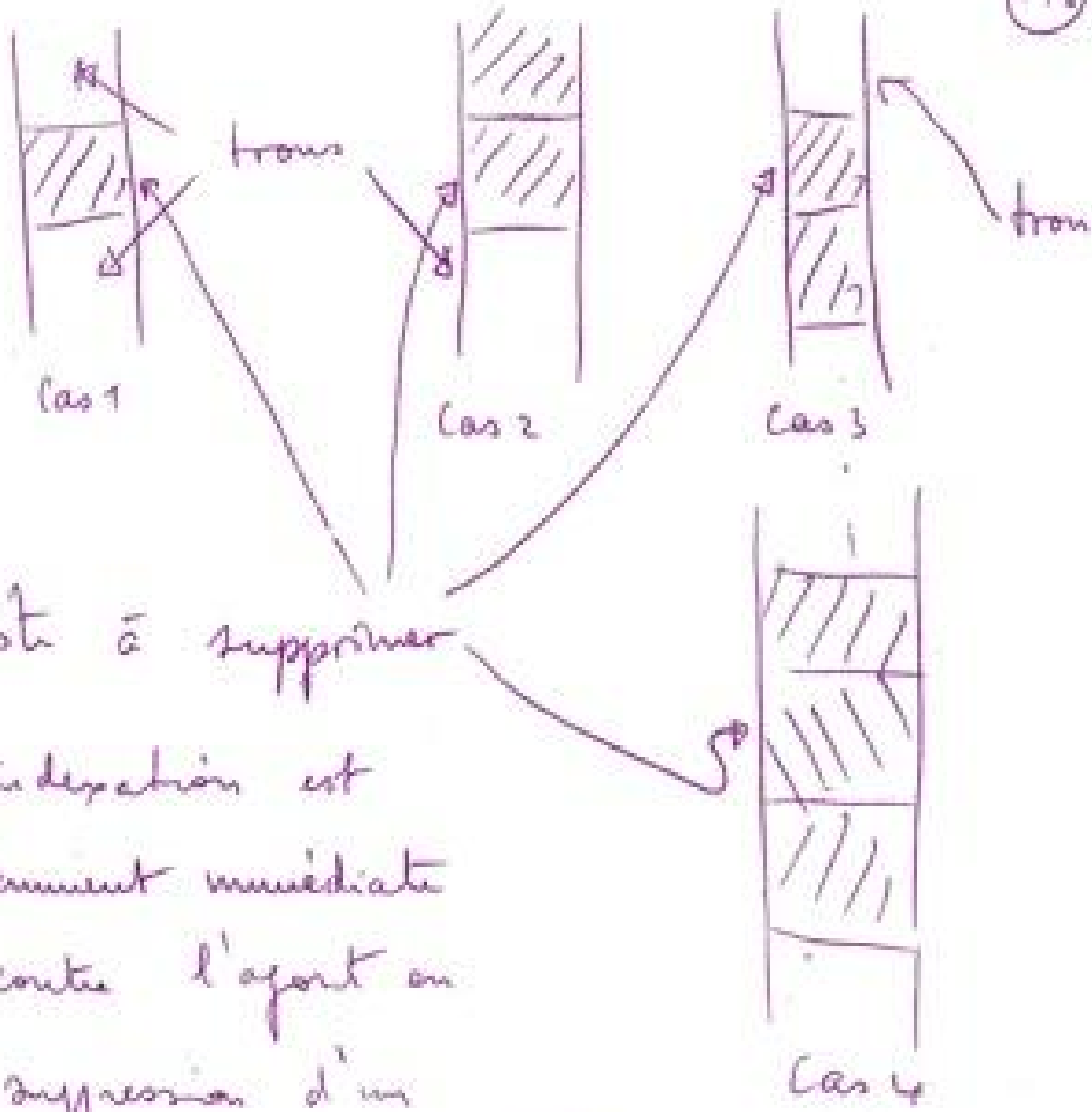
① Méthode pratiquement séquentielle :

la liste est allouée séquentiellement

la manière

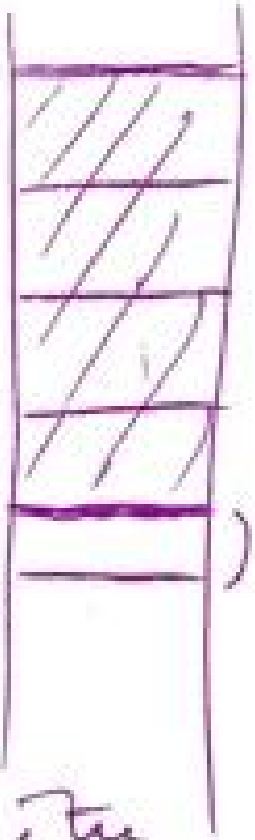
l'allocation ou la création font des opérations éventuellement complexe car il est à chaque fois de trouver un trou de taille suffisante ou alors de faire des opérations de compactage (très coûteuse) entre les différentes listes linéaires de façon à concaténer entre eux les petits trous pour en former de plus gros -

La désallocation par contre est simple : on doit seulement vérifier pour la désallocation d'une liste entre dans l'un des 4 cas suivants et reformatter l'espace en conséquences :



L'indexation est
nécessairement immédiate
car contre l'agent en
la suppression d'un
élément au milieu peut être
assez complexe (éventuellement obligation
de compactage)
la place occupée est minimale et
les possibilités d'extension en mémoire
secondaire sont complexes

② Méthode purement séquentielle avec allocation par fonctions de mots de taille fixe



Cette méthode est une simple variante de la méthode précédente de façon à améliorer la consommation éventuelle ;) perte

Ceci se paie par une perte de place qui peut être plus ou moins importante suivant le rapport du reste de la division de la taille de la liste par la taille du pavé, à la taille du pavé $\mod(t, p)/p$.

3) Allocation par groupe de mots de taille variable

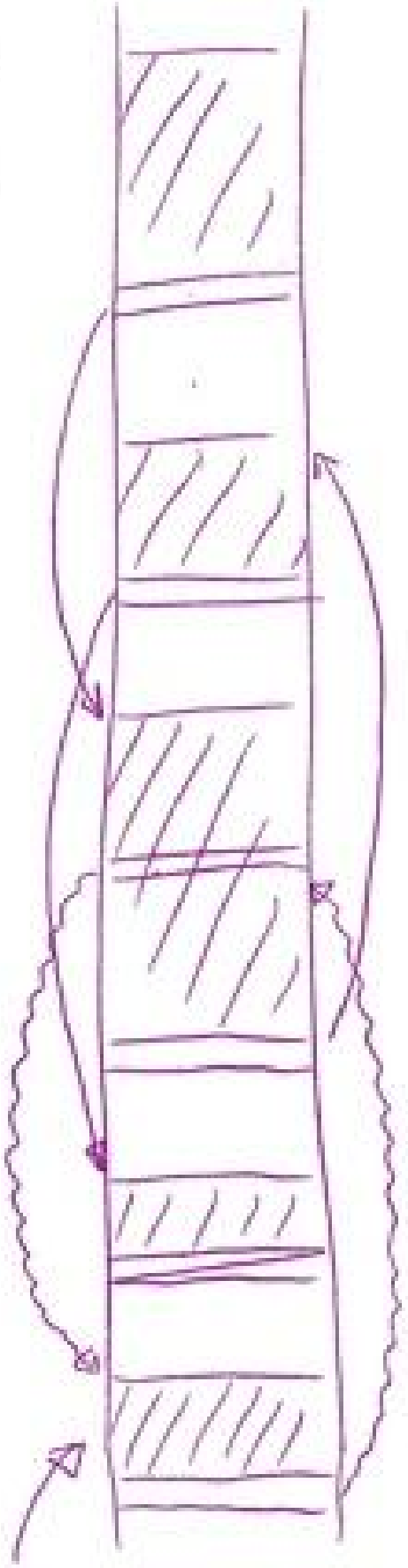
De façon à améliorer les problèmes d'allocation des méthodes précédentes on peut fractionner une liste en différentes parties séquentielles de taille variable, chaque partie étant reliée à la mémoire par un pointeur.

Cette méthode se caractérise par le fait que l'on peut de temps en temps réorganiser les listes de façon à les rendre parfaitement séquentielles.

Les caractéristiques d'une telle méthode sont donc variables.

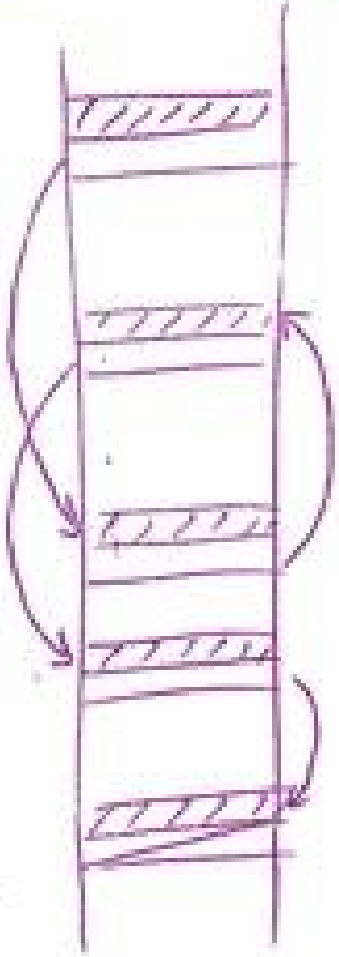
Son principal défaut consiste en l'impossibilité d'indexer en général

par contre il est facile d'ajouter un (ou mieux plusieurs) éléments au milieu d'une liste organisée de cette façon : comme d'habitude le schéma ci contre. on tronçonne (en 2) la partie concernée et l'on "plante" un pointeur (avec recopie du mot écrasé par le pointeur dans la partie ajoutée) vers le nouveau groupe puis l'on chaîne à la 2e partie de l'ensemble coupé —



groupes de mots puis l'on ajoute "au milieu"

④ Méthode provisoirement chaînée



Très pratique pour ajouter des éléments au milieu, cette méthode interdit toute indexation et prend beaucoup de place ((509) en pointeurs)

5) Méthode chaînée par éléments de plus d'un mot de taille fixe

C'est une variante de la méthode précédente dans la quelle on chaine entre eux non plus chaque mot mais des groupes de taille constante de mots.

Prend moins de place que la méthode précédente et permet une certaine indexation.

si . p_1 est la ϕ taille de chaque groupe

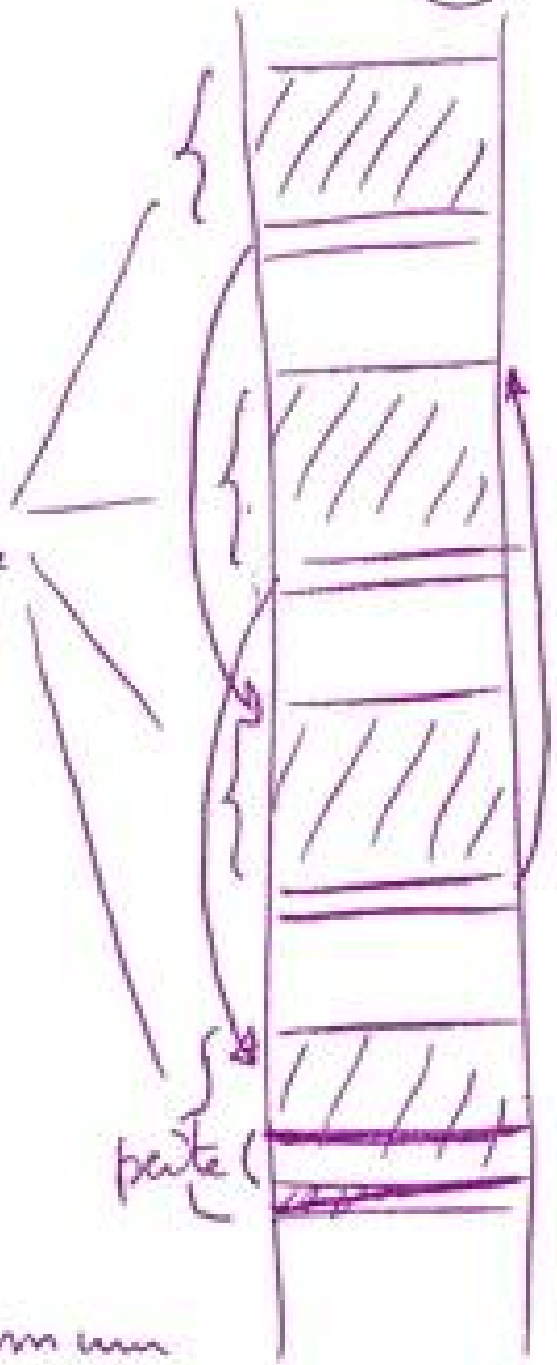
. t est la taille de la liste

la place "perdue" est :

$$\left[\frac{t}{p}\right] + \frac{p}{2}$$

pour la pointeur

Valeur maximum de la perte en fin de liste

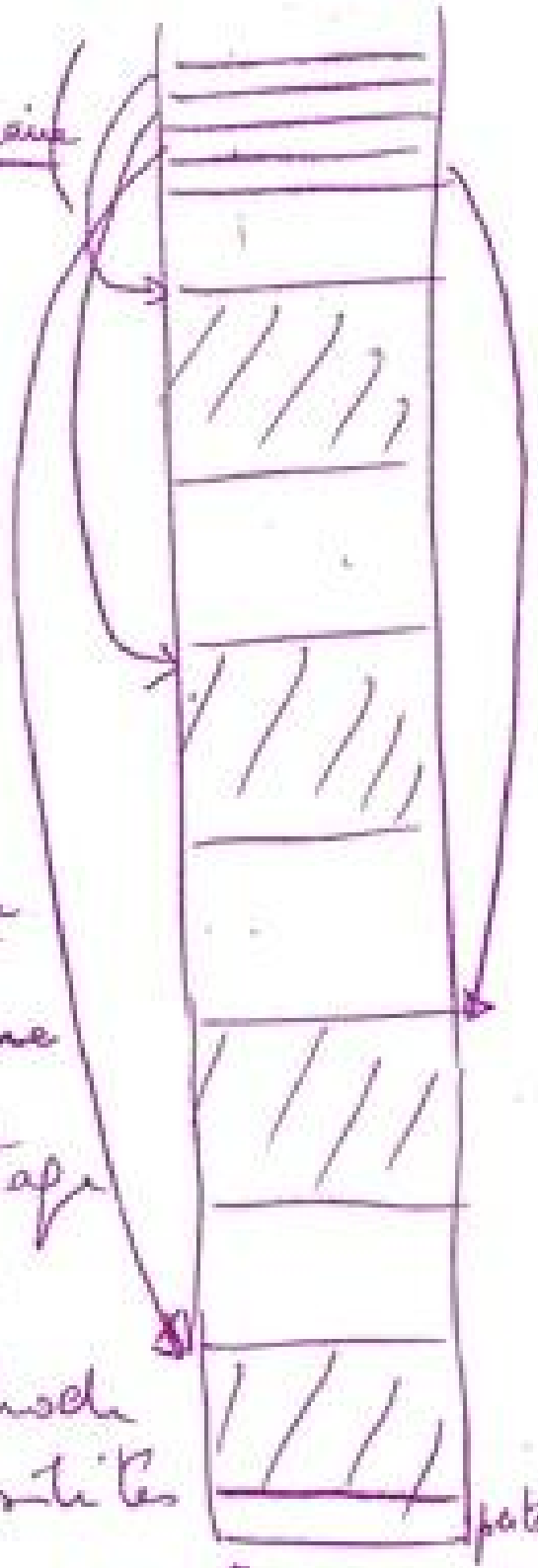


Il existe donc un optimum de la taille p approximativement pour $p = \sqrt{2t}$

Avec cette méthode on peut commencer de pratiquer une extension en mémoire secondaire au niveau de chaque groupe

6) Méthode du dictionnaire (papination)

Cette méthode apparaît comme une extension de la méthode précédente : au lieu de chaîner entre elles les différentes pages comme dans la méthode précédente on regroupe ces pointeurs dans une liste linéaire ~~informatique~~ : le dictionnaire



Elle présente l'avantage par rapport à la précédente de réduire les possibilités d'indexation (en fait double indexation)

les possibilités peuvent d'ailleurs être facilement implémentées par hardware -

Par contre on retombe sur une nouvelle série de difficultés car pour le dictionnaire doit être une liste refermée (Méthode ①) on peut donc être amené pour conserver les possibilités d'indexation du dictionnaire à le parcourir lui même et donc à envisager un dictionnaire du dictionnaire etc - - -

Cette méthode peut être assez coûteuse en place par contre son extension à la mémoire secondaire est simple

7) Méthode du système SOCRATE

Cette méthode vise à avoir les avantages de la précédente (indexation)

Sans avoir les inconvénients de la méthode 1 (risque d'avoir à faire du compactage) ni les inconvénients de la méthode précédente (allocation donc pagination du dictionnaire)

On suppose que l'espace mémoire est extrêmement grand (par exemple 2^{32} mots > 2 milliards de mots ...) - Les diverses listes sont allouées statiquement dans cet espace - Leur nombre et leur taille sont donc fixes (main tenus fixes). Cet espace s'appelle l'espace virtuel - Le véritable espace mémoire s'appelle l'espace réel -

La fonction fmi permet de passer de l'adresse virtuelle d'un mot à son adresse réelle et une fonction de replacement prise de projection du

l'espace virtuel sur l'espace réel

comme! Si on définit le

schéma ci contre -

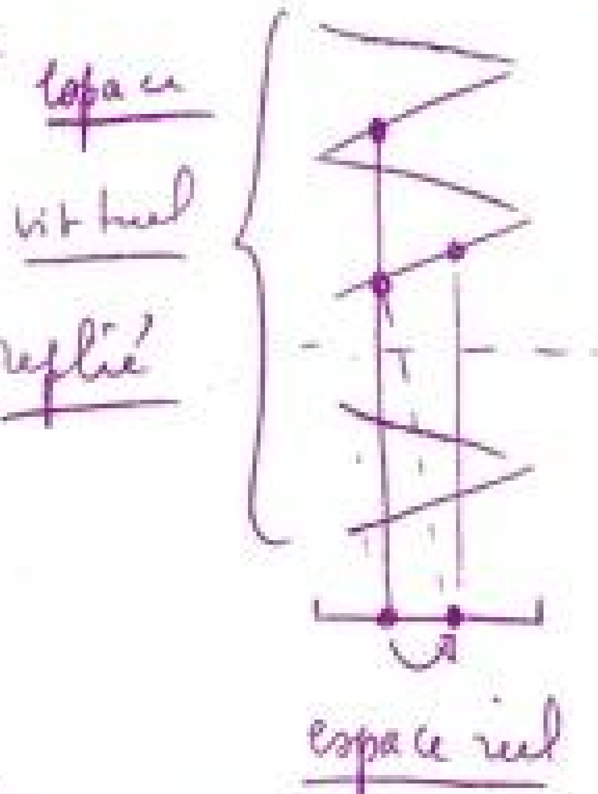
Cette fonction

entraîne bien entendu

plusieurs mots virtuels

à avoir la même

projection réelle -



Pour résoudre ce problème on chaîne

entre eux dans l'espace réel les

mots qui sont ~~[illegible]~~ issus

de mots virtuels ayant la même

projection - De façon à optimiser

on peut comme dans la méthode 3

(par rapport à la méthode 2) grouper

plusieurs mots virtuels consécutifs

~~[illegible]~~ de taille fixe de façon

à avoir un pointeur de chaînage

non plus pour un mot mais pour plusieurs - Appelons page ce groupe de mots -

L'espace réelle est donc formé d'une suite de pages qui peuvent être dans l'un des états suivants

libre : cette page réelle ne correspond à aucune page virtuelle

Occupée et bien placée : cette page réelle correspond à une page virtuelle dont elle est la projection.

Occupée et mal placée : cette page réelle correspond à une page virtuelle mais sa position n'est pas la projection de cette page virtuelle : on l'appelle un squatter -

Lorsque l'on lit un mot d'une page virtuelle trois cas peuvent se présenter :

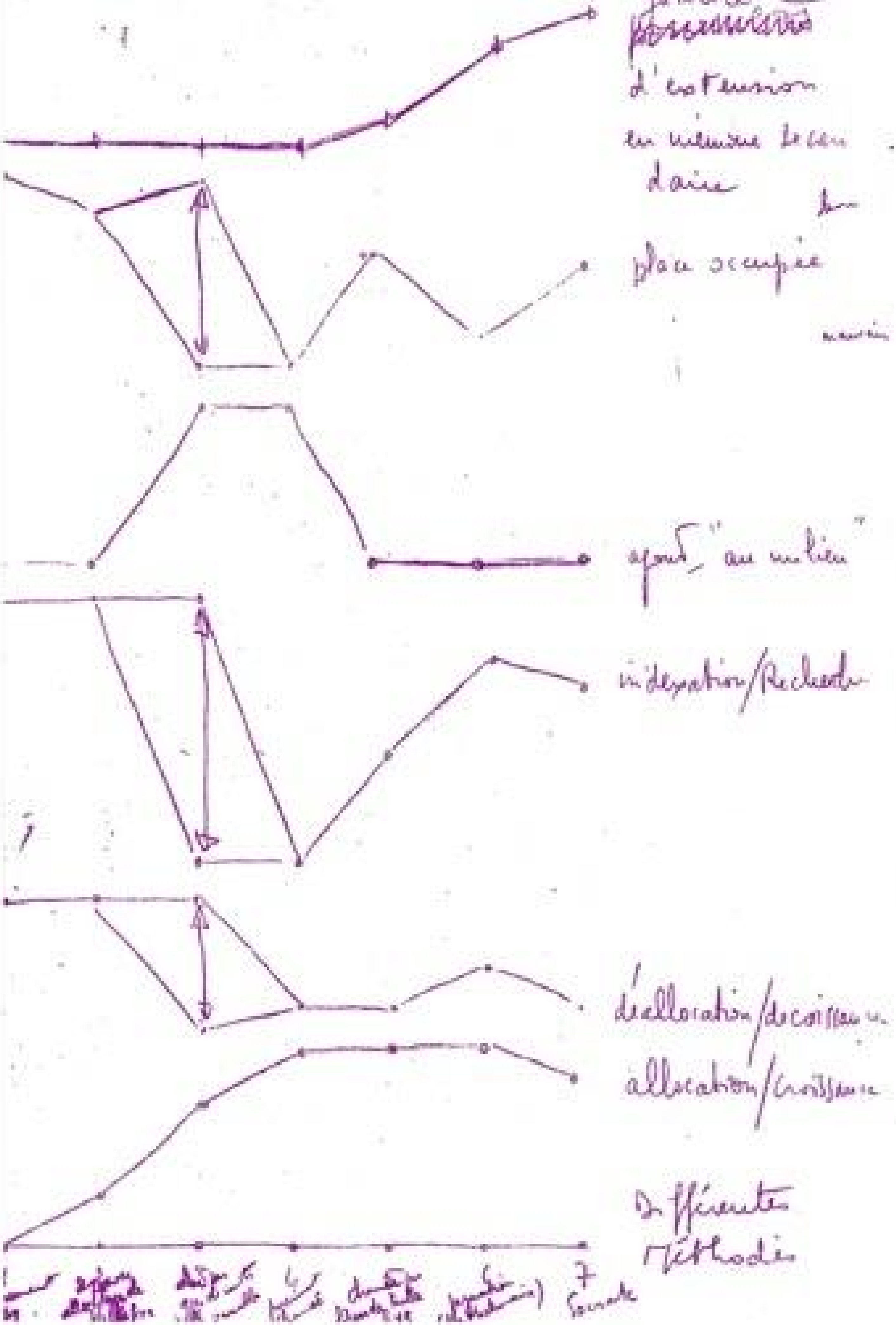
- la page projection est libre : la page virtuelle n'existe pas

- la page projection est occupée et bien placée : on lit directement le mot

- la page projection est occupée et mal placée : on se déplace avec le chariot pour trouver (éventuellement) la bonne page vide -

Lorsque l'on veut écrire (un mot d') dans une page virtuelle on peut se retrouver dans l'un de ces 3 cas : dans le troisième on cherche ~~a~~ une page libre (dans l'espace réel) que l'on choisit aux autres systèmes -

Nous allons maintenant analyser ces diverses méthodes dans les diagrammes comparatifs qui suivent


facilité
d'extension
en mémoire secondaire
place occupée
accès "au milieu"
indexation/Recherche
désallocation/décroissance
allocation/croissance
Difficultés
méthodes

Sur la base de ces diagrammes (quelque peu subjectifs il est vrai) nous allons étudier les cas pratiques d'emploi de ces différentes méthodes

Pour la méthode 3 on peut avoir une infinité de points suivants l'état de la fragmentation

Méthodes 1 et 2 trois - On a représenté les extrêmes sur certains diagrammes

Lorsque l'on a peu ou pas d'allocation dynamique (utilisée), le peu de place et les facilités d'indexation procurés par ces méthodes les rendent intéressantes -

Les compilateurs allouent de cette façon les données statiques en mémoire centrale - A noter aussi que les ingénieurs FETMAIN et FREEMAIN (IBM) travaillent de cette façon On les emploie aussi parfois dans les systèmes en temps réel pour

(de mémoire centrale)
gérer des listes éventuellement dynamiques (on emploie autant alors la méthode 2) - De façon à ne pas perdre de temps en compactage lorsque l'on veut agrandir on crée une nouvelle liste on utilise les temps morts pour opérer des compactages même si cela n'est pas nécessaire : En effet les systèmes temps réel sont caractérisés par une courbe de charge très variable donc de nombreux temps morts -

Méthodes 3 et 4

L'intérêt essentiel de ces méthodes réside dans le fait de pouvoir agrandir ou diminuer des listes par le milieu - Si par ailleurs

les besoins en indexation ne sont pas fondamentaux par rapport aux besoins en extension dynamiques on choisira ces méthodes de préférence aux autres -

C'est le cas des langages de listes. la méthode 4 tend maintenant à supplanter la méthode 3 à cause de l'économie de place qui peut être très importante

Voir à ce sujet les articles suivants :

- Compact list representation : definition, garbage collection and system implementation par W.J. Hansen CACM Vol 12 N 9 sept 69
- A non recursive list compacting algorithm par C.J. Cheney CACM V.13 N 11 Nov 70

Méthode 5

La méthode 5 est intéressante car elle permet une allocation / déallocation simple (on chaîne un nouveau bloc ou on en détache un), une certaine indépendance et n'utilise pas trop de place.

Cette méthode est en particulier assez efficace pour manipuler des files d'attente en mémoire centrale ou des piles. En plus généralement des listes linéaires dont on veut modifier que les noms voisins de l'extrémité. Il suffit d'ajouter un registre qui pointe sur le dernier bloc chaîné.

C'est la méthode qui tend à supplanter l'allocation

dynamique de PL/1 (Voir aussi
cf le paragraphe intitulé
"Run time stack" page 132 du
document IBM System /360
Operating System PL/1 (F) : Program
mers guide)

Cette méthode est aussi celle généralement préconisée pour stocker les contextes d'exécution (les registres) des programmes écrits en assembleur.

Voir en particulier le paragraphe intitulé "Providing a save area" page 11 du document IBM System/360 Operating System : Supervisor and Data Management Services —

Méthode 6 (pagination)

Les facilités conjuguées d'indexation et d'allocation constituent le mérite majeur de cette méthode. Les possibilités d'extension à une mémoire secondaire et sa réalisation facile par hardware l'ont rendue très populaire. Elle est implémentée sur de très nombreuses machines : CII 10070 et IRIS 80, IBM 360/67 et au dessus, GE 645 etc ---

Méthode 7

Cette méthode présente un bon compromis, elle est plus économique en place que la méthode 6 mais son implémentation par hardware est problématique — d'allocation

statique dans l'espace virtuel présente un très grand intérêt pour la simplification des programmes : en particulier l'adresse virtuelle d'une liste est son nom et le catalogue peut se réduire à une chaine de bits. Nous reviendrons sur ces points dans le paragraphe consacré aux catalogues — (§ 2.4.3)

2.4.2 Représentation des tables d'occupation

Le terme "Table d'occupation" sert à désigner un objet donnée en soi qui représente l'état d'occupation d'un support (ici homogène)

Il y a deux cas à distinguer suivant que les éléments (libres ou occupés) sont de taille variable (multiples

1, 2 et 3) ou de Tailles fixe
(méthodes 4, 5, 6 et 7)

Dans tous les cas la "table d'occupation" ne représente généralement que les éléments libres (les trous)

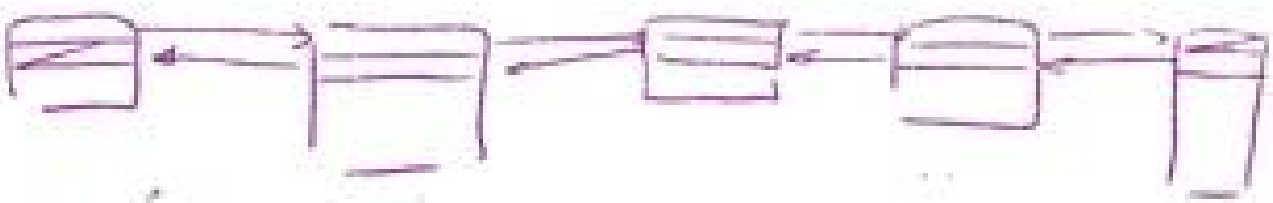
1er cas : éléments de longueur variable

chaque trou contient sa propre taille - les ~~éléments~~ trous sont chaînés entre eux.

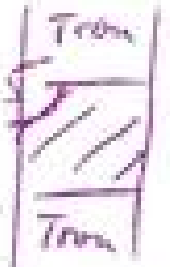
On peut le faire de 3 façons différentes

① par ordre chronologique

dès qu'une suite de mots devient libre on la chaîne (éventuellement doublement) à la fin de la liste des trous.

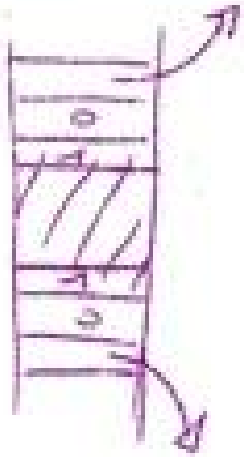


Cette méthode est rapide mais
ne fonctionne pas la concaténation
des trous devient lorsque cela
est nécessaire comme dans
la libération de la zone
hachurée ci-contre car il n'y a



aucune raison pour les trous adjacents
soient
le soit dans la liste des trous.

De façon à résoudre ce problème
on peut mettre un code spécial
(par exemple 0) à la fin et au
début d'un trou.

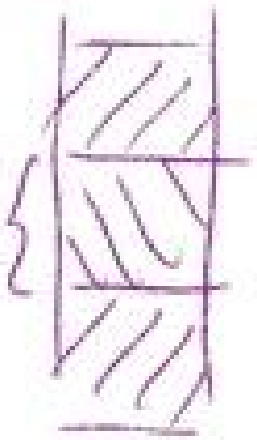


(Mais alors il faut mettre
aussi un code spécial : par
exemple 1, à la fin et
au début d'une zone pleine)

Lorsque deux trous se fondent en un seul on est donc amené à remplacer la chaîne des trous : c'est pourquoi il est nécessaire de disposer d'une double chaînage

(b) par ordre d'adresse croissante

De façon à simplifier les problèmes précédents on peut chaîner les trous par ordre d'adresse croissante - Cela amène une perte de temps supplémentaire lorsque l'on est dans la zone de chaînage courte car il faut faire une recherche dans la chaîne des trous pour insérer le nouveau trou à la bonne place.



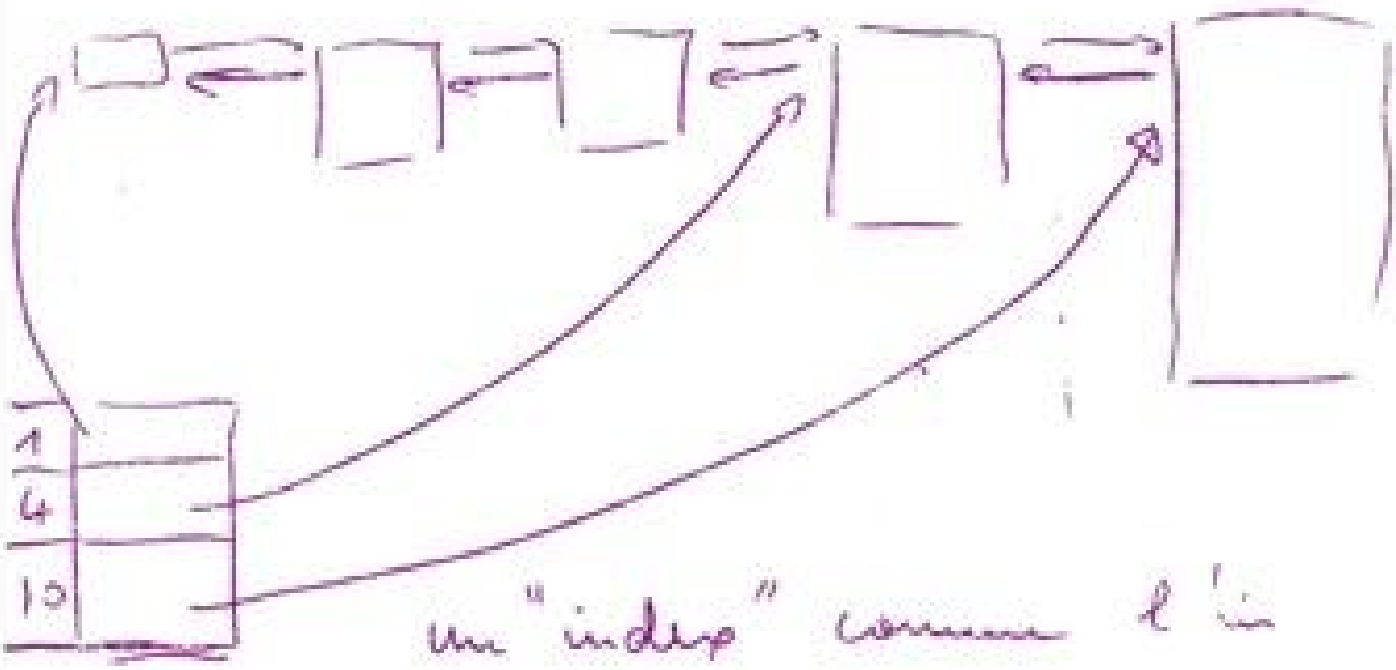
c) par taille croissante

Lorsque l'on a besoin d'une zone d'une certaine taille on peut employer l'une ou l'autre des 2 methodes

- "first fit" : on cherche dans la liste des trous le premier d'entre eux qui a une taille au moins egale à celle désirée - Cette methode favorise la fragmentation et on peut lui preferer la suivante

- "best fit" : on cherche dans la liste des trous le meilleur d'entre eux c'est à dire le plus petit parmi ceux qui conviennent - Cette deuxième methode peut être longue car il faut chercher un minimum c'est pourquoi on

dressement des trous par taille
constante peut être intéressant



un "index" comme l'in
dique le schéma ci dessus peut
encore améliorer cette recherche —

Une étude poussée des méthodes
"best fit" et "first fit" et
de diverses implémentations de la
dernière peut être trouvée dans
l'ouvrage: "the Art of Computer
Programming" de D. Knuth (Vol 1)

Addison-Wesley) au paragraphe .5 page 435 (Dynamic storage allocation) - Voir aussi le paragraphe "Allocation and release of storage in an Area" page 145 du manuel IBM system/360 operating system PL/1 (F) Programmer's guide

2. Cas : éléments de longueur fixe.

Dans ce cas la plupart des problèmes rencontrés précédemment n'existent plus ; les pages libres forment une liste libre implémentée

en

De façon à grouper les allocations consécutives dans une même zone (intéressant surtout lorsque l'on emploie une mémoire secondaire) on peut employer la méthode de la "bit map" c'est à dire

une chaine de bits contenant autant de bits qu'il y a de pages : un bit égal à 1 indique que la page correspondante est occupée, on trouve zéro dans le cas contraire.

On peut alors rechercher facilement un certain nombre de pages consécutives libres en promenant un masque sur la chaine de bits.

2.4.3 Représentation des catalogues

Le catalogue est un système de donnée en soi qui contient pour chaque donnée (au minimum) deux valeurs : son nom, et un pointeur sur son lieu.

Rappelons que le catalogue sert à deux "moments" :

- lorsqu'on baptise une donnée pour vérifier que le nom que l'on donne n'est pas déjà celui d'une autre donnée déjà existante

- lorsque l'on veut faire un accès logique à une donnée pour pouvoir trouver son lieu à partir de son nom

Dans les 2 cas la vertu principale d'un catalogue doit être la rapidité de consultation.

Implémenter un catalogue sous la forme d'une liste linéaire séquentielle (ordinaire) est souvent (une méthode) encore trop lente car : il faut parcourir

toute la liste.

C'est pourquoi les catalogues sont

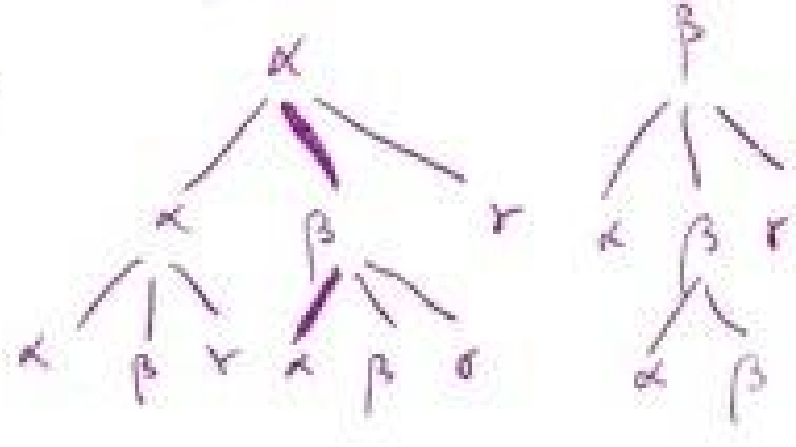
(a) des listes linéaires séquentielles classées par ~~ordre~~ ordre ~~alpha~~ des noms (pour pouvoir faire des recherches dichotomiques) ;

(b) des structures en arbres (ou même en forêt) : un nom est alors en fait une suite de chaînes alphanumériques - Les noeuds fils d'un noeud donné sont classés par ordre - C'est le cas de la plupart des catalogues de fichiers dans les systèmes d'exploitation classiques

Exemple :

Un nom peut être

$\alpha . \beta . \alpha$



Les "index" des clés des enregistrements des fichiers indexés séquentiels sont organisés avec de cette façon

(c) des listes linéaires à support séquentiel mais remplies de façon pseudo-aléatoire par hashcoding

Cette méthode est rapide mais elle prend beaucoup de place - De plus lorsque le ~~dictionnaire~~ catalogue hashcodé est saturé on ne peut plus l'étendre dynamiquement sans en faire une reorganisation complète

C'est pourquoi cette méthode est intéressante lorsque l'on peut se fier avec certitude (ou imposer aux utilisateurs) un nombre maxi

nuum de noms à entrer dans le catalogue et lorsque d'autre sont le catalogue (qui prend beaucoup de place) a une durée de vie limitée.

C'est typiquement le cas pour les catalogues qui contiennent les noms générés des données utilisés dans les programmes assemblés ou compilés : c'est à dire les Table des symboles

les compilateurs ou assembleurs

④ Cas particuliers

Dans le système SOCRATE, l'adresse virtuelle d'une donnée est son nom et le mécanisme de passage de l'adresse virtuelle à l'adresse réelle

se fait par programme : la 2e fonction du catalogue est donc inutile.

La première fonction est realisée par une chaine de bits.

Exemple : si l'on veut réaliser un système pouvant supporter 1024 listes linéaires on devra utiliser une chaine de 128 octets. Chaque bit indique à la liste correspondante existe ou non.

Si l'on desire par ailleurs donner un nom externe alphanumérique à une donnée on peut realiser un catalogue hashcodé contenant pour chaque donnée son nom externe et son nom interne (son adresse interne). Cependant on minimise la place occupé par ce 2e catalogue

car il est lui-même "virtualisé" (208) et les trous qu'il comporte n'ont pas leur contrepartie dans l'espace réel —

2.4.4 Milieu mémoire non homogène

Comme nous l'avons précisé plus haut nous allons considérer le cas de 2 milieux :

- l'un le milieu tore ayant un temps d'accès de l'ordre de 1 µsec
- l'autre le disque ayant un temps d'accès de l'ordre de 100 ms (on pourrait aussi considérer le cas du Tambour ou du disque à tête fixe ayant un temps d'accès de l'ordre de 10 ms)

Toute la difficulté de l'allocation dans ce cas vient du rapport 10000 (ou 1000000) entre les temps d'accès —

Le milieu Tore ne sert pas pour les traitements, les listes sont normalement stockées sur le Disque —

On peut considérer plusieurs cas :

@ la taille du tore est suffisante pour contenir entièrement la liste linéaire stockée de façon permanente sur le disque On entre alors toute la liste en mémoire centrale lorsque l'on en a besoin — Si l'on désire accéder

a une autre liste ou la mettre
la mémoire centrale s'il y a encore
de la place pour contenir entièrement
cette nouvelle liste ou bien dans
le cas contraire on remet sur
disque l'ancienne pour faire de
la place — Cette méthode est
connue sous le nom de swapping —
Elle a été utilisé dans les premiers
systèmes fonctionnant en multiprogram-
mation (les listes linéaires en
question étant alors des programmes)
Un particulier pour la Burroughs
B 5000 et les suivantes —

(b) La taille du tore n'est
pas suffisante pour contenir la
liste sur laquelle on veut

travailler mais on est capable d'indiquer a priori la manière dont on va employer la liste (ou plus largement la structure de donnée) de façon à faire des entrées partielles en mémoire centrale qui vont se recouvrir les unes les autres - On place ces entrées partielles dans une zone tampon (buffer) dont la taille est la plus grande taille parmi celles des structures partielles que l'on traite - Dans le cas où la structure en question est un programme on dit que l'on

fait de l'"overlaye" - Dans le cas où la structure en question est une donnée on a affaire à un fichier dont les différents enregistrements se retrouvent en mémoire centrale sur le buffer - Dans ce dernier cas de façon à rentabiliser l'accès disque, on peut (surtout lorsque l'on doit pratiquer un parcours séquentiel sur les enregistrements) rentrer plusieurs enregistrements sur le buffer en une seule opération d'accès : le nombre d'enregistrements que l'on rentre simultanément s'appelle le facteur de blocage -

© La mémoire totale ne peut contenir la totalité de chaque liste et de plus il est impossible de prévoir à l'avance le comportement du programme de traitement à l'égard de la liste. On pratique alors une transformation du tore et du disque par pagination.

Le disque est divisé en un certain nombre de pages - Le tore est divisé de même en un certain nombre de cadres de la même taille que les pages -

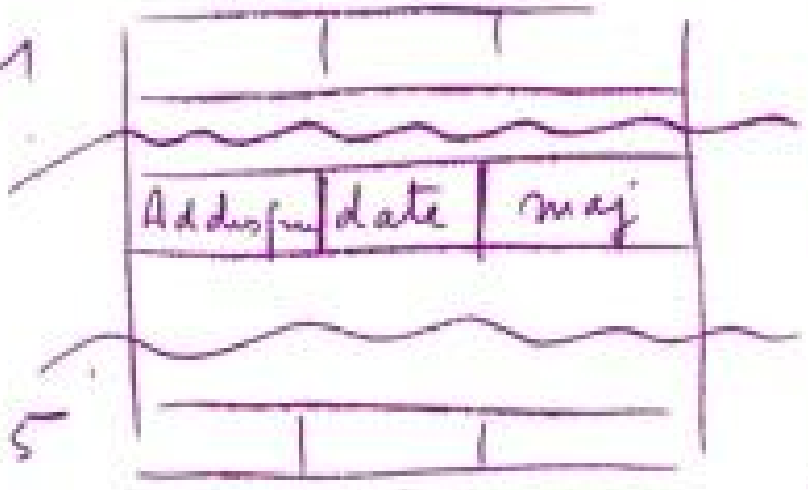
Supposons pour fixer les idées une zone de mémoire centrale de 150 K octets et que l'on pagine à 100 Millions d'octets sur disque, on aura

alors : la taille d'une page est de 2 Koctets :

75 cadres en mémoire centrale

50.000 pages sur disque

Les cadres sont gérés par une table d'occupation comportant pour chacune des 75 entrées l'adresse disque de la page correspondante ainsi que des renseignements sur la date de sa consultation et le fait qu'elle a été mise à jour ou non.



Le fonctionnement est très simple :
lorsque l'on veut consulter une page
on cherche dans la table d'occupation
des cadres si elle est là
- 1° oui on la consulte
- 2° non on choisit un cadre
pour servir de place à la page que
l'on va amener depuis le disque —
De nouveau 3 cas
- le cadre choisi est inoccupé
- le cadre choisi est occupé par
une page non modifiée : on la recouvre
= le cadre choisi est occupé par
une page modifiée : on la recopie sur
le disque avant d'utiliser le cadre
L'algorithme que nous venons de

décide s'appelle un algorithme
de "demand paging" par opposi
tion aux algorithmes où l'on a
toujours "un coup d'avance" de façon
à avoir systématiquement un cadre
libre en mémoire centrale lorsque
l'on a besoin de mettre une nou
velle page -
L'algorithme de choix des cadre
à libérer ou "page removal algorithms"
peut faire appel à différentes
critères : le but visé est
évidemment d'éviter d'utiliser
le cadre qui contient une
page qui va être à nouveau

dans un proche avenir (le cas d'un cadre ne contenant pas de page ou se rencontrant pas en dehors de la période de tout à fait initiale de fonctionnement) Un critère classique consiste à choisir comme cadre celui qui contient la page la plus anciennement consultée. De façon à ce que ce cadre soit déterminé rapidement on peut chaîner (doublement) entre eux les cadres par ordre de consultation (chaque fois que l'on consulte un cadre on le place en queue de la file) : celui qui est le premier de la file est le plus anciennement consulté)

la disparition d'un milieu mémoire intermédiaire entre le Tore et le disque modifie fondamentalement le comportement général de tels systèmes :

On peut placer des milieux intermédiaires soit "près" du tore ("extended core memories" vus plus haut) soit près du disque (Tambour ou disque à têtes fixes) - Pour des raisons de prix on trouve généralement des Tambours ou des disques à têtes fixes -

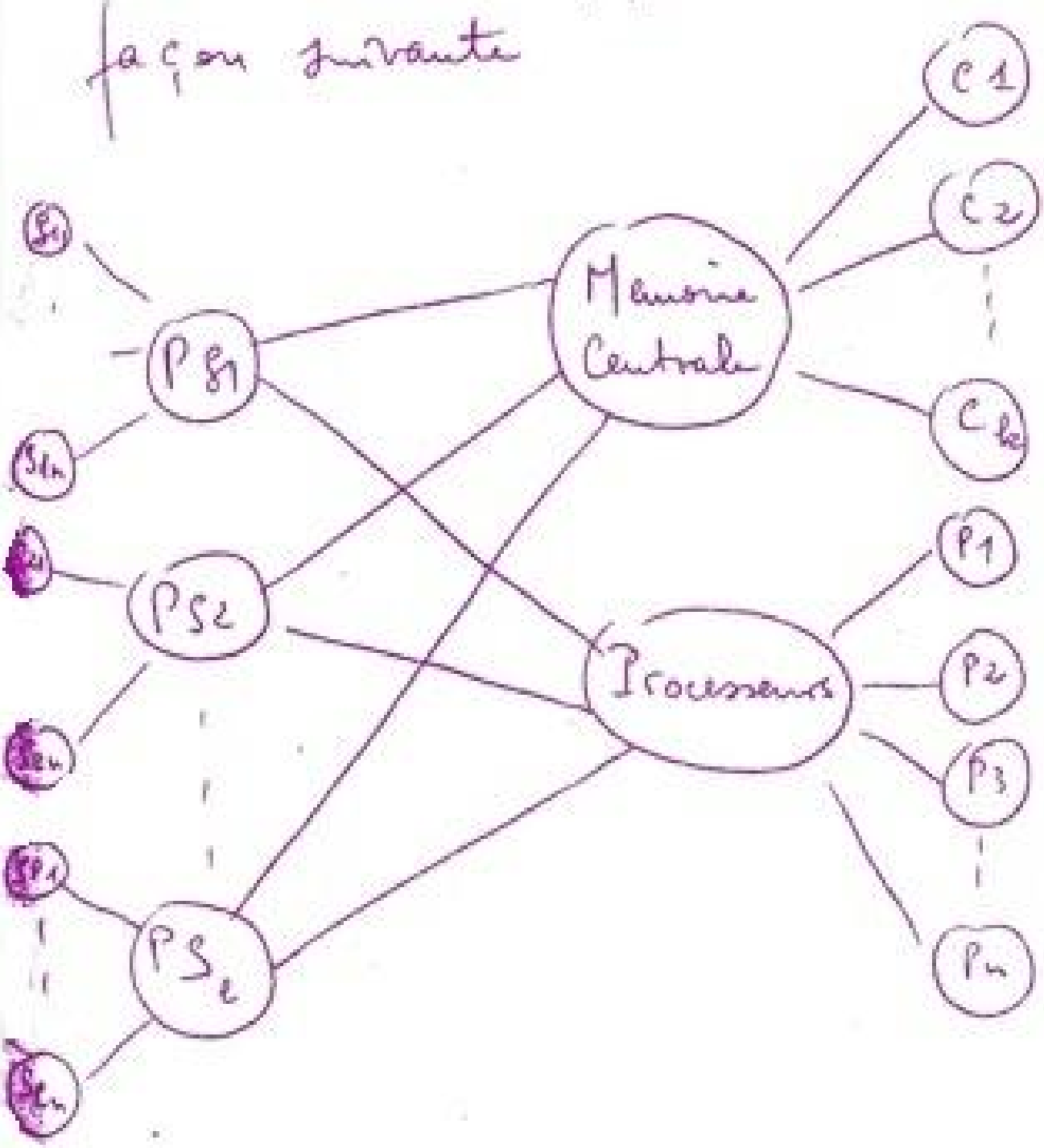
Pour une étude très complète de toutes ces techniques on pourra se reporter avec profit à l'article

suivant : "Virtual memory" (219)
par P. J. Denning Computer
Surveys Vol 2 N. 3 Sept 70 -

J'espère que le lecteur qui réfléchit n'aura pas manqué de remarquer l'analogie très grande qui existe entre le "page removal algorithm" et le Régulateur de cet page (152) - En fait il s'agit du même problème : on est en présence dans les 2 cas d'une ressource quantifiée : la classe de hardware dont les quantas sont les différents hardware

de la classe et la mémoire centrale dont les flèches sont à les différents cadres.

Il convient donc d'améliorer le schéma de la page (165) de la façon suivante



Chaque serveur ne se trouve plus dans une seule file d'attente : # file de garçon (processeurs) mais il se trouve aussi dans la file d'attente file de garçon (mémoire centrale)

les deux Régulateurs :

Régulateur (processeur) et Régulateur (mémoire centrale) ne doivent donc pas travailler indépendamment mais au contraire se coupler de façon à donner de façon harmonieuse un processeur et un (ou plusieurs) cadre(s) au même serveur et ceci le plus simultanément possible de façon à ce qu'on ne

se trouve pas dans la situation
(la + fréquente dans la pratique)
d'un client fini/un processeur et
pas de mémoire ou l'inverse -
Ceci est un des problèmes les plus
difficile à résoudre actuellement
dans l'écriture des systèmes -

De la même façon pour nous
avions suggéré la réalisation
d'un organe unique chargé de réguler
toutes les classes de hardware
on pourrait adjoindre
à cet organe la tâche supplémentaire
de réguler la mémoire centrale -
Ceci faciliterait d'autant plus
les comparaisons entre régulations.