

## Principes des Systèmes d'Exploitation

Michel Billaud [michel.billaud@u-bordeaux.fr](mailto:michel.billaud@u-bordeaux.fr)  
Département Informatique - IUT - Université de Bordeaux  
Novembre 2014

Ce document est copiable et distribuable librement et gratuitement à la condition expresse que son contenu ne soit modifié en aucune façon, et en particulier que le nom de son auteur et de son institution d'origine continuent à y figurer.

# Table des matières

<b>1</b>	<b>Ordinateurs et systèmes</b>	<b>3</b>
1.1	Le premier ordinateur	3
1.1.1	Les calculateurs électroniques	3
1.1.2	Mémoire à tube de Williams-Kilburn	3
1.1.3	Architecture et programmation	4
1.1.4	Une démonstration probante	4
1.1.5	Les suites	5
1.2	Structure d'un ordinateur simple	5
1.2.1	La mémoire	5
1.2.2	Le processeur	6
1.2.3	Les périphériques	8
1.2.4	Les outils de programmation	8
1.3	Utilisation en mono-tâche	9
1.3.1	Moniteur d'enchaînement des travaux	9
1.3.2	Planification de travaux	10
1.3.3	Modifications matérielles nécessaires	10
1.3.4	Déroulement d'un appel système	10
1.4	Profiter des temps morts : la multiprogrammation	10
1.4.1	Motivation économique	10
1.4.2	Étude d'un cas	11
1.4.3	La mise en oeuvre du multitâche	12
1.5	Fonctionnement d'un centre de calcul	12
1.6	Travailler à plusieurs : le temps partagé	13
1.7	De nos jours...	14
<b>2</b>	<b>Les processus</b>	<b>15</b>
2.1	Introduction	15
2.2	Histoire	15
2.2.1	Multitâche	15
2.2.2	Exemple : intérêt du multi-tâches	16
2.2.3	Du <i>batch</i> au <i>time sharing</i>	16
2.2.4	Support matériel du multi-tâches : les interruptions	16
2.2.5	Support logiciel	17
2.3	Définitions	17
2.3.1	Les processus	17
2.3.2	Les états des processus	17
2.3.3	Changements d'état	17
2.3.4	Multitâche coopératif	18
2.3.5	Scénario détaillé	18
2.3.6	Multitâche préemptif	19
2.3.7	Multitâche préemptif et utilisation interactive	20
2.3.8	Interruptions et multitâche, en résumé	20
2.4	Politiques d'ordonnancement	20
2.4.1	Ordonnanceur	20
2.4.2	Critères d'évaluation	20
2.4.3	Tourniquet	21
2.4.4	Priorités	21
2.4.5	Priorités variables	21
2.4.6	Files multiples	22
<b>3</b>	<b>Gestion de la Mémoire</b>	<b>23</b>
3.1	Mémoire et multi-programmation	23
3.1.1	Motivation	23
3.1.2	Rappel : fonctionnement de la mémoire	23
3.2	Adresses logiques et physiques	24
3.2.1	Le chargement des processus en mémoire	24
3.2.2	Adresses logiques	25
3.2.3	Protection mémoire	25
3.2.4	Memory Management Unit	26
3.3	Gestion d'un espace mémoire linéaire	26
3.3.1	Exemple	26
3.3.2	Problème : la fragmentation	26
3.3.3	Solutions curatives	27
3.3.4	Solutions préventives	27
3.3.5	Buddy system (blocs compagnons)	27
3.3.6	Partitions de taille fixe	28
3.4	Va-et-vient sur disque	28
3.5	La mémoire segmentée	28
3.5.1	Segments	29
3.5.2	Espace d'adressage	29

3.5.3	La réalisation	29
3.5.4	Tables locale/globale des segments	30
3.5.5	Espace mémoire paginé	31
3.5.6	Un circuit MMU : le MC68851	32
3.6	Mémoire virtuelle paginée	32
3.6.1	Principe	32
3.6.2	MMU pour la mémoire virtuelle paginée	33
3.6.3	Les algorithmes de remplacement de page	33
3.6.4	Remplacement à tour de rôle	33
3.6.5	Algorithme LRU : least recently used	34
3.6.6	NRU, remplacement d'une page non récemment utilisée	35
3.6.7	Algorithme de la seconde chance	35
3.6.8	Algorithme du vieillissement	35
3.6.9	Réservation d'espace d'échange	35
3.6.10	Intérêt du swap ?	36
3.7	Mémoire virtuelle segmentée-paginée	36
<b>4</b>	<b>Gestion des fichiers</b>	<b>38</b>
4.1	Les fichiers	38
4.2	Les disques	38
4.2.1	Disque, tête	39
4.2.2	Avec plusieurs plateaux	40
4.2.3	Quelques chiffres	40
4.3	Gestion des entrées/sorties sur disque	40
4.3.1	Sur un système mono-tâche	40
4.3.2	Sur un système multitâche	40
4.3.3	E/S : Premier arrivé, premier servi	40
4.3.4	E/S : plus court déplacement	41
4.3.5	Politique de l'ascenseur	41
4.3.6	E/S : Deadline driven disk scheduler	41
4.3.7	Conclusion	41
4.4	Systèmes RAID	41
4.4.1	Principe et objectifs	41
4.4.2	NRAID (JBOD)	42
4.4.3	RAID 0 : agrégation par bandes	42
4.4.4	RAID 1 : miroir	43
4.4.5	Disques de rechange, reconstruction	43
4.4.6	RAID 4 : agrégation par bandes avec parité	44
4.4.7	RAID 5 : agrégation par bandes avec parité répartie	44
4.4.8	Combinaisons : RAID 10	44
4.4.9	RAID 51	45
4.4.10	RAID 50	45
4.4.11	Choix d'un système RAID	45
4.5	Systèmes de fichiers	45
4.5.1	Fonctions du SGF	45
4.5.2	Catalogue de fichiers	45
4.5.3	FAT : file allocation table	46
4.5.4	Tables des blocs Unix	46
4.5.5	Représentation des répertoires	47
4.5.6	Autres caractéristiques des SGF	47

# Chapitre 1

## Ordinateurs et systèmes

### 1.1 Le premier ordinateur

Les ordinateurs tels que nous les connaissons sont des objets qui s'inscrivent dans une longue suite d'inventions et de combinaisons de technologies diverses.

Si on définit l'ordinateur comme

un appareil électronique qui fait des calculs en suivant les instructions d'un programme enregistré

le premier ordinateur construit est très probablement le *Manchester Small Scale Experimental Machine (SSEM)*<sup>1</sup>,



qui a tourné pour la première fois le 21 juin 1948. Il avait été réalisé par Tom Kilburn et Geoff Tootill, dans l'équipe de Freddie Williams, professeur d'électrotechnique à l'Université de Manchester.

Voir le reportage tourné par la BBC en 1948, sur <http://news.bbc.co.uk/2/hi/technology/7465115.stm>

#### 1.1.1 Les calculateurs électroniques

**Les calculateurs électroniques à programme** existaient déjà depuis une quinzaine d'années, mais leurs programmes n'étaient pas enregistrés en mémoire. Ils étaient soit externes (bande ou carte perforées), soit figés. Par exemple sur l'ENIAC (1946), des interrupteurs à tourner dans tableau de connexion pour réaliser les 0 et les 1 d'une mémoire morte.

1. surnommé "Baby", voir <http://www.computer50.org/mark1/new.baby.html>

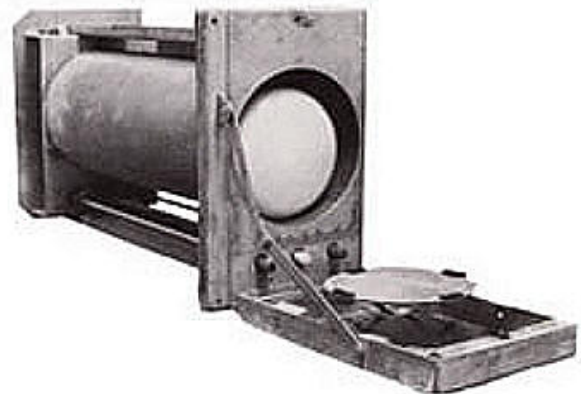
2. Voir l'article de Wikipedia consacré à la mécanique

Cette manière de faire dérivait assez naturellement des machines à traiter les cartes perforées, utilisées depuis la fin du XIX<sup>e</sup> siècle.<sup>2</sup>

En réalité le SSEM était un prototype destiné à tester l'utilisabilité d'une mémoire à tube cathodique inventée par F. Williams.

#### 1.1.2 Mémoire à tube de Williams-Kilburn

Cette innovation utilisait un tube d'oscilloscope standard, dans lequel un faisceau d'électrons permettrait d'allumer des points de phosphore sur l'écran, avec une certaine rémanence. Le tube de Williams-Kilburn utilise la propriété suivante : quand le faisceau bombarde un point de l'écran, des électrons secondaires sont éjectés par le phosphore, en quantité différente selon que le point est ou non déjà allumé. En mesurant la tension sur une plaque métallique devant l'écran, on peut connaître l'état du point.



En 1947, l'équipe de Williams avait réussi à stocker 2048 bits sur un écran pendant des heures, ce qui promettait une technologie de mémoire rapide, bon marché, basée sur des composants standards, destinée aux calculateurs.

L'idée est donc venue assez naturellement de fabriquer un ordinateur simple avec une mémoire à tube pour en tester la fiabilité dans une machine qui effectue plusieurs milliers de lectures/écritures par seconde. Jusque là, le tube avait été testé en adressant les bits par un jeu d'interrupteurs manuels...

### 1.1.3 Architecture et programmation

L'architecture du SSEM est très simple. En termes modernes, c'est une machine 32 bits, avec une mémoire de 32 mots de 32 bits (1024 bits stockés sur un tube), extensible à 8192.

Les calculs se font en nombre entiers en notation complément à deux.

Deux tubes étaient utilisés pour les registres spéciaux :

- l'un pour l'accumulateur A (32 bits) sur lequel se font les opérations
- l'autre pour CI (control instruction) qui contient d'adresse de l'instruction en cours, et PI (present instruction), l'instruction elle-même

Un dernier tube (sans plaque) dupliquait le premier, permettant de voir les bits en mémoire<sup>3</sup>.

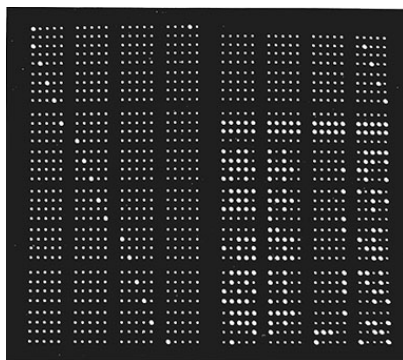


Fig. 6. A typical Storage Pattern on C.R.T.

La mémoire du SSEM

Le jeu d'instructions était réduit à 7 instructions d'un format unique : 3 bits pour le code opération, et 13 bits pour l'adresse S de l'opérande. Les 16 derniers bits étaient inutilisés. Les 7 opérations étaient

- LDN S (load negative)  $A = -\text{Mem}[S]$ , qui charge dans l'accumulateur l'opposé du contenu d'un mot mémoire
- SUB S (subtract)  $A = A - \text{Mem}[S]$ , qui soustrait le contenu d'un mot
- STO S (store)  $\text{Mem}[S] = A$ , qui copie en mémoire le contenu de l'accumulateur
- CMP (compare) If  $A < 0$ ,  $CI = CI + 1$ , qui saute l'instruction suivante si l'accumulateur est négatif,
- JMP S (jump)  $CI = \text{Mem}[S] + 1$ , qui provoque un saut indirect, à l'adresse contenue dans un mot de la mémoire.
- JRP S (jump relative)  $CI = CI + \text{Mem}[S] + 1$ , pour un saut indirect relatif.
- HLT (halt) qui arrête l'ordinateur.

Ce jeu d'instruction a été choisi parce qu'il était réalisable avec un minimum de circuits électroniques. Il ne simplifie évidemment pas la vie du programmeur. Par exemple, pour additionner deux nombres  $x$  et  $y$  situés aux adresses 20 et 21, il faut 4 opérations en passant par une variable temporaire (d'adresse 22)

```
0 LDN 20 ; A contient -x
1 SUB 21 ; A contient -x-y
2 STO 22 ; Mem[22] contient -x-y
3 LDN 22 ; A contient -(-x-y) = x+y
```

Voici un exemple plus complexe, écrit en utilisant des adresses symboliques : le calcul du maximum de deux nombres  $X$  et  $Y$  et le rangement dans  $Z$ .

# calculer la différence

```
0 LDN X ; A = -x
1 STO TMP ; TMP = -x
2 LDN TMP ; A = x
3 SUB Y
```

# selon la différence, charger -X ou -Y  
# dans l'accumulateur

```
4 CMP
5 JMP I8 ; si A positif
6 LDN Y
7 JMP I9
8 LDN X
```

# et envoyer l'opposé de l'accumulateur dans Z

```
9 STO TMP
10 LDN TMP
11 STO Z
12 HLT
```

# Les variables

```
13 X = 42
14 Y = 15
15 TMP = 0
16 Z = 0
```

# les adresses de saut

```
17 i8 = 7
18 i9 = 8
```

Remarque : pour aller à l'instruction 9, l'instruction 7 charge le mot d'adresse 17 dans le CI, auquel il ajoute 1 comme après toute instruction. C'est pourquoi le mot 17 contient 8, l'adresse qui précède celle de l'endroit où le programme doit se poursuivre.

### 1.1.4 Une démonstration probante

Le programme précédent suffit à occuper plus de la moitié de la mémoire disponible sur le SSEM, qui n'a évidemment jamais servi à faire des calculs très complexes. Le clou du spectacle était un programme de 17 instructions<sup>4</sup> pour trouver le plus grand diviseur propre d'un nombre  $N$ , en essayant de le diviser successivement par  $N-1$ ,  $N-2$  etc., la division étant elle-même réalisée par soustraction successives.

3. les bits de poids fort sont à droite, contrairement à la notation habituelle des nombres en binaire

4. La légende de l'université de Manchester dit que c'est le seul programme que le professeur Williams ait jamais écrit.



Kilburn et Williams devant la console du SSEM

Il a fallu 52 minutes de calcul (et 3,5 millions d'opérations) pour établir que le plus grand facteur propre de  $2^{18}$  était  $2^{17}$ . Ce que tout le monde savait déjà évidemment. Sur l'écran phosphorescent, une puissance de deux est facile à lire : un point allumé sur une ligne éteinte.

Peu importe : l'objectif était de montrer que la mémoire était fiable : peu importe les calculs du moment que le résultat est correct après des millions d'opérations.

### 1.1.5 Les suites

Un vrai ordinateur est sorti de ces travaux, *Manchester Automatic Digital Machine* (MADM), opérationnel en avril 1949, et qui a donné naissance aux machines du constructeur Ferranti.

Quant aux tubes de Williams, ils ont été utilisés comme mémoires dans quelques ordinateurs célèbres (UNIVAC 1103, Whirlwind, IBM 701, IBM 702 ...) avant d'être rapidement supplantés par les mémoires à tores de ferrite, qui ont dominé le marché pendant 20 ans de 1955 à 1975, avant d'être remplacées par les mémoires à semi-conducteurs que nous utilisons aujourd'hui.

## 1.2 Structure d'un ordinateur simple

Schématiquement, un ordinateur est composé

- d'un **processeur**, circuit électronique capable d'exécuter une à une (mais très vite) les instructions d'un programme
- d'une **mémoire centrale**, circuit qui sert à mémoriser les données et les programmes pendant leur exécution
- des **périphériques** : imprimante, carte réseau, carte graphique, disque dur etc. reliés par des contrôleurs d'interface.

Le rôle du Baby étant de tester la fiabilité des mémoires à tubes de Williams-Kilburn, il était dépourvu de périphérique.

Dans ce chapitre, nous présentons ces divers éléments de façon très simplifiée.

### 1.2.1 La mémoire

Différentes technologies ont été utilisées pour réaliser les mémoires des ordinateurs : bascules bistables à base de tubes

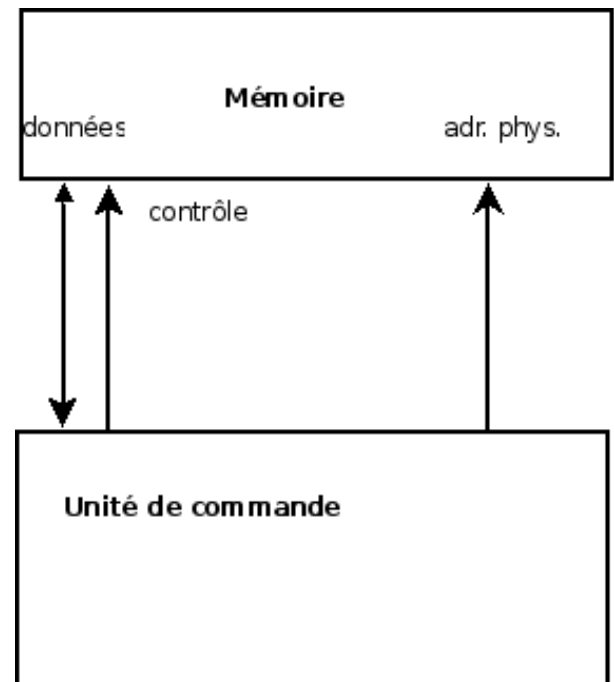
(triodes), mémoires à tores de ferrite, à transistors, circuits intégrés etc.

Indépendamment des technologies, la mémoire est un organe qui a pour fonction de stocker et restituer des "mots binaires" repérés par une adresse.

Les mots sont de taille fixe<sup>5</sup>, par exemple l'ATLAS (réalisé en 1962 conjointement par l'université de Manchester, Ferranti et Plessey) avait une mémoire de 16384 mots de 48 bits. Les micro-processeurs des années 70 étaient souvent des machines à octets (mot = 8 bits), de nos jours ce sont des mots de 32 ou 64 bits.

La mémoire communique avec le reste de l'ordinateur par 3 bus (groupes de fils)

- le **bus de contrôle** (il faudrait dire bus de commande), qui indique à la mémoire l'opération que l'on veut effectuer : lecture ou écriture ;
- le **bus de données**, bidirectionnel, qui sert à émettre et recevoir les mots ;
- le **bus d'adresses**, qui indique à la mémoire l'adresse concernée.



La mémoire et ses trois bus

#### Opérations :

- **lecture** : pour consulter l'adresse A en mémoire, le processeur place le nombre A sur le bus d'adresses, et envoie le signal de contrôle "lecture". Après un petit délai de réponse, le contenu du mot d'adresse A est présent sur le bus de données.
- **écriture** : pour envoyer un mot M à l'adresse A, le processeur place A sur le bus d'adresses, M sur le bus de données et active l'ordre d'écriture.

5. Il y a eu bien sûr quelques exceptions, qui ont été des échecs. Dans l'histoire des ordinateurs, beaucoup de choses ont été essayées.

**Remarque :** sur certaines machines (c'était le cas du processeur 8088 qui équipait les premiers PC d'IBM<sup>6</sup>), les bus de données et d'adresses sont *multiplexés*, ils partagent des fils. L'avantage est de minimiser le nombre de connexions entre circuits intégrés (et du nombre de pattes), l'inconvénient est que les données et les adresses ne sont pas être transmis en même temps, ce qui se fait au détriment des performances. Un signal de commande supplémentaire précise si l'information qui circule est une adresse ou une donnée.

## 1.2.2 Le processeur

Un processeur est un dispositif électronique relativement simple, composé de circuits logiques divers. Il communique avec la mémoire (voir plus haut) et les périphériques par des bus de données, d'adresses, et de commande.

Son rôle est d'exécuter, les unes après les autres, des instructions qui sont stockées en mémoire.

### Instructions et registres

Le **compteur de programme**<sup>7</sup> est un registre<sup>8</sup> qui contient l'adresse de la prochaine instruction à exécuter. C'est un *compteur* parce que, la plupart du temps, on va lui ajouter 1 pour passer à l'instruction suivante.

La première action du processeur est de lire en mémoire le mot qui contient cette instruction, et de la placer dans un **registre d'instruction**, où il sera décodé.<sup>9</sup>

Par exemple, on aura peut être lu le mot de 32 bits **000110000100001100000000000101010** qui, sur un PowerPC 32 bits, se décompose en **001110 00010 00011 00000000 000101010** ce qui représente

- les 6 premiers bits : le code de l'opération "ajouter une constante"
- un numéro de registre destination (registre 2) sur 5 bits
- un numéro de registre source (registre 3)
- une constante (42) codée sur 16 bits

et qui signifie : ajoutez au registre de travail numéro 3 la valeur 42, et placez le résultat dans le registre 2, ce qu'on écrirait en *langage d'assemblage*

```
addi 2,3,42
```

L'exécution de l'opération ci-dessus fera appel à différents autres circuits

- les **registres généraux**, qui stockent des valeurs intermédiaires (il y en a 32 sur le PowerPC). C'est la généralisation de l'accumulateur A du SSEM.
- une **unité arithmétique**, qui sera ici chargée de s'occuper de l'addition.

Certaines opérations permettront des transferts entre registres et mémoire, par exemple

```
stw 5,0,1234
```

envoie (*store*) le contenu du registre général 5 à l'adresse 1234 de la mémoire.

Il y a également des instructions pour comparer le contenu de registres, et d'autres qui changent le cours de l'exécution si une condition est remplie, en indiquant le numéro de la prochaine instruction à exécuter.

En fait, les instructions de comparaison positionnent des indicateurs booléens dans un **registre de condition** qui mémorisent le résultat de la comparaison (inférieur, supérieur, égal ?)

En fonction de ces indicateurs, les **instructions de branchement conditionnel** incrémentent le compteur de programme, ou lui affectent une autre adresse.

Rappel : sur le SSEM, le bit de signe de l'accumulateur était l'unique indicateur de condition, utilisé par l'instruction CMP.

À ces opérations s'ajoutent des **instructions d'entrée-sortie**, permettant le dialogue avec des circuits **contrôleurs de périphériques**, ainsi que des instructions spéciales que nous verrons plus tard.

Tous ces circuits fonctionnent sous le contrôle d'un **séquenceur**, qui enchaîne les différentes étapes :

- envoyer le contenu du PC sur le bus d'adresse, et un ordre de lecture
- copier la valeur présente sur le bus de données dans le registre d'instruction RI, et indiquer la fin de lecture
- décoder l'instruction contenue dans le RI
- l'exécuter. Si il s'agit de "ajouter 42 au registre 3" :
  - envoyer le contenu du registre 3, la valeur 42 et l'ordre d'addition à l'unité arithmétique (circuit de calcul)
  - copier le résultat dans le registre 3
  - ajouter 1 au PC
  - et recommencer

## Architecture interne d'un processeur

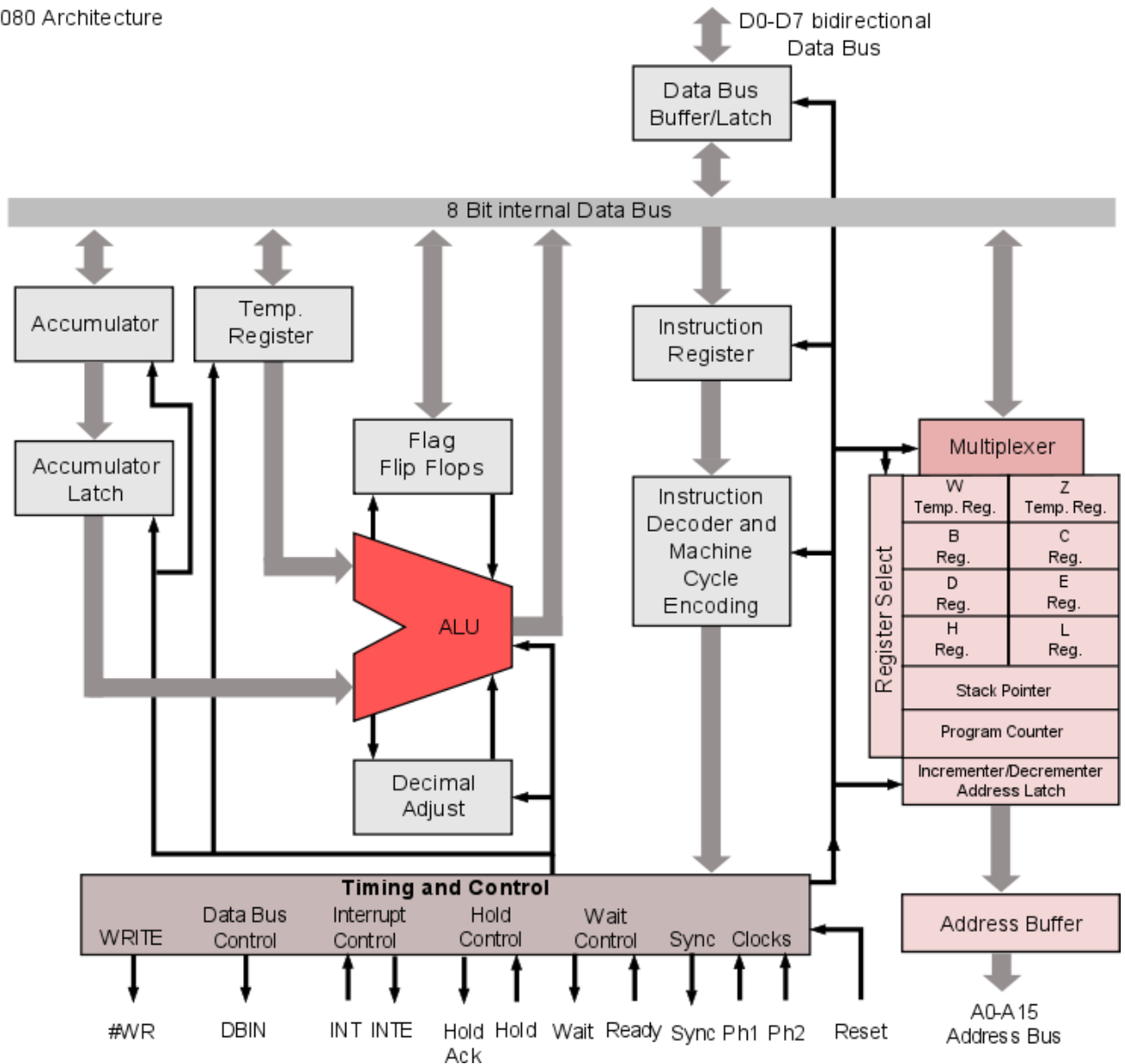
Le schéma ci-dessous montre l'architecture interne du processeur 8080 mis sur le marché par Intel en Avril 1974. C'est le second micro-processeur 8 bit, après le 8008.

6. Le processeur 8086 d'INTEL possédait des bus séparés, le 8088 qui en était dérivé était à la fois plus cher à fabriquer (c'est un 8086 avec en plus des circuits de multiplexage/démultiplexage) et moins performant. Ceci dit, le 8086 nécessitait une famille de circuits associés 16 bits (contrôleurs d'interruption, etc), alors que le 8088 pouvait se contenter des circuits 8 bits qui étaient déjà produits en masse pour les microprocesseurs 8080 et 8085 qui équipaient les micro-ordinateurs les plus courants de l'époque (sous CP/M).

7. ou pointeur d'instruction, compteur ordinal...

8. en électronique numérique, un *registre* est un circuit qui mémorise quelques bits d'information. Il est possible d'y charger une valeur, et de la relire ensuite.

9. La terminologie du SSEM les désignait par CI (current instruction) et PI (present instruction)



On retrouve

- en haut, le bus de données de 8 bits
- en bas à droite le bus d'adresse 16 bits,
- à gauche l'accumulateur A
- au centre l'unité arithmétique et logique (ALU)
- le registre d'instruction
- à droite une "banque de registres", dont le compteur de programme, et des registres de travail (B,C,D ...)

### Le modèle du programmeur

Dans un ordinateur, certains registres sont utilisables directement par le programmeur (comme le registre accumulateur par exemple), et d'autres ont un rôle interne (le registre d'instruction).

Ce qu'on appelle **modèle du programmeur**, c'est la partie qui est utilisable directement par le programmeur

- le compteur de programme,
- les registres généraux et spécialisés
- les registres de condition

- les différentes catégories d'instruction
  - chargement/ rangement en mémoire
  - arithmétique
  - logique : et, ou, décalages...
  - tests
  - branchements conditionnels et inconditionnels,
  - ...

Par exemple, dans le 8080, le registre C est utilisable comme opérande d'une instruction, par exemple `MOV A,C` (copie de C dans A), il fait donc partir du modèle contrairement au registre TEMP qui sert d'intermédiaire dans cer-



taines instructions. Par exemple l'instruction ADD B (ajouter le contenu du registre B à l'accumulateur) se déroule en deux temps

1. copier B dans TEMP, et A dans le registre "latch" (autre registre temporaire), pour les présenter en entrée de l'UAL,
2. envoyer le résultat de LATCH+TEMP, sortant de l'UAL, dans A.

**Exercice 1.** Pourquoi y a-t-il deux étapes, et non une seule ?

### La programmation en langage machine

Un exemple de programme en pseudo-assembleur vous montre le niveau de détail auquel il faut descendre quand on programme dans un *langage machine* : la séquence ci-dessous, qui commence à l'adresse 100, calcule la somme des entiers de 1 à N, en supposant que N est dans le registre 3 et que le résultat doit aller dans le registre 4.

```
100  mettre la valeur 0 dans r4
101  comparer r3 et la valeur 0
102  si égal, aller à 106
103  ajouter r3 à r4
104  ajouter la valeur -1 à r3
105  aller à 101
106  ...
```

Après l'exécution de l'instruction d'adresse 102, le compteur de programme vaudra 103 ou 106, selon la valeur des indicateurs positionnés par l'instruction précédente (101).

**Exercice 2.** Écrivez un programme du même type pour le SSEM, en essayant de le faire tenir sur 32 mots.

Comme vous le voyez, un processeur n'est pas bien compliqué. C'est un assemblage de quelques circuits de base : registres, additionneurs, etc. qui sait exécuter des instructions de base très élémentaires.

Ce qui est compliqué c'est de combiner des instructions aussi rudimentaires pour effectuer des traitements utiles (qui ne sont pas forcément simples, eux). Affronter cette complexité, c'est la spécificité du travail du programmeur.

### 1.2.3 Les périphériques

Enfin, dès les années 60, une grande variété de périphériques permet l'entrée, la sortie et le stockage des données, ainsi que la communication.

Sur les premiers ordinateurs, les périphériques courants étaient

- les lecteurs de rubans perforés, support fragile et peu commode hérité des télécriteurs, abandonnés rapidement.
- Les lecteurs et perforateurs de cartes (hérités de la mécanographie),
- les imprimantes ;
- les lecteurs de bandes magnétiques.
- des terminaux interactifs : machines à écrire électriques, voire écrans cathodiques.

Assez rapidement, on en est venu à utiliser un petit ordinateur auxiliaire - dit "frontal" - pour recopier les cartes perforées sur des bandes magnétiques, de lecture bien plus rapide. Et inversement, les résultats de l'ordinateur principal étaient transférés sur des bandes magnétiques que le frontal se chargeait de faire imprimer ou perforer.

Ainsi on économisait le temps précieux du gros ordinateur.

### 1.2.4 Les outils de programmation

Dans les premiers temps de l'informatique, le programmeur écrivait ses programmes en binaire, en utilisant la liste des instructions de l'ordinateur (instruction set) et en les codant lui-même en binaire. Pour des programmes de quelques dizaines d'instructions, c'était encore envisageable.

Il est ensuite apparu que cette activité de codage, éminemment fastidieuse, était trop sujette à erreurs. Il était donc préférable d'utiliser un programme pour faire mécaniquement ce codage sans risque d'erreur.

L'histoire officielle dit que l'idée d'utiliser un programme de traduction vient de John Neumann en 1945, mais selon d'autres sources<sup>10</sup>, Von Neumann était en fait initialement opposé à cette idée (émise par un étudiant), parce que cela gaspillait le précieux temps de calcul de l'ordinateur, alors qu'on pouvait très bien confier ce travail à des étudiants modestement rémunérés.

Un programme écrit en *langage d'assemblage* se présente comme une suite d'instructions utilisant les codes mnémotechniques, comme `addi` pour "add immediate value" sur le PowerPC. Parfois c'est plus obscur, comme `lwz xu` (load word with zero indexed with update). L'*assembleur* est le programme de traduction<sup>11</sup>, qui assemble les traductions de chaque instruction.

On est ensuite passé (au milieu des années 50) à la traduction automatique de formules mathématiques, puis de programmes complets, avec le langage FORTRAN (Formula translator). Là aussi, l'intérêt n'était pas évident pour tout le monde. Le même John Von Neumann a déclaré, quand on lui a présenté FORTRAN en 1954 « why would you want more than machine language ? »

Inversement, l'arrivée de langages de haut niveau a parfois soulevé un enthousiasme excessif. En effet, il suffisait d'écrire

```
multiply PRIX—UNITAIRE
      by QUANTITE giving PRIX.
add PRIX to TOTAL.
```

10. <https://beacon.salemstate.edu/~tevens/VonNeuma.htm>

11. mais on dit souvent, par métonymie, "programmer en assembleur"

là où, autrefois, un professionnel barbu grassement rémunéré produisait des lignes de code absolument incompréhensibles

```
load  PU
mult  QTE
store PRIX
add   TOT
store TOT
```

y compris par son chef de service, incapable d'en vérifier la qualité.

Quand COBOL a été annoncé au début des années 60, certains y ont vu, un peu vite, la fin du métier de programmeur.

C'était en effet la fin d'un certain type de programmation, mais il reste que même si le code est écrit avec des phrases anglaises et semble facile à relire après une formation d'une semaine, la difficulté est en réalité dans l'algorithmique et dans l'organisation du code, composé d'une multitude d'opérations simples. La programmation reste un métier à part entière, qui ne s'improvise pas.

## 1.3 Utilisation en mono-tâche

Les faibles capacités des premiers ordinateurs (quelques dizaines de kilo-octets de mémoire) ne permettaient que de faire exécuter un programme à la fois.

Un opérateur était donc chargé de mettre le travail fourni par les utilisateurs (cartes perforées ou bande magnétique) dans la mémoire de la machine, de lancer l'exécution et de récupérer les résultats imprimés (ou enregistrer). Et de recommencer avec le travail suivant.

Chaque travail disposait donc de l'intégralité des ressources de la machine.

**Exercice 3.** Les premières machines, expérimentales, étaient utilisées en mono-tâche à la demande. Quand un utilisateur arrivait avec un travail à faire passer sur le calculateur, il devait attendre que les précédents aient libéré la place avant d'utiliser la machine.

Imaginons l'arrivée de 4 utilisateurs :

- John arrive à 8h00, avec un travail qui dure 40 mn
- Grace arrive à 8h10, avec un travail de 30 mn
- Alan arrive à 8h20, avec un travail de 1 h 5 mn
- Niklaus arrive à 8h40, avec un travail de 25 mn

1. Quel est le “temps de service” pour chaque utilisateur (durée entre son arrivée en salle d'attente et la fin de son travail), si ils passent dans l'ordre d'arrivée (politique FIFO, *first-in first-out*) ? Calculez le temps de service moyen.
2. Mêmes questions si les utilisateurs décident de faire passer en premier celui qui <sup>a</sup> a le travail le plus court (politique dite “du plus courts temps d'exécution”).

a. parmi ceux qui sont présents

### 1.3.1 Moniteur d'enchaînement des travaux

Pour éviter de perdre du temps, on a vite eu l'idée d'automatiser l'enchaînement des travaux. Un petit programme, toujours présent en mémoire, assurait la lecture du travail suivant dès qu'un travail était terminé, et gagnait ainsi de précieuses minutes.

Cet embryon de système peut prendre la forme d'une boucle de quelques instructions pour copier en mémoire les cartes de l'exécutable à charger, avant de lui transférer le contrôle. Un programme utilisateur qui se termine normalement doit simplement relancer le moniteur.

En début de journée (et après chaque crash), l'opérateur manipule les interrupteurs de la console pour entrer ce “chargeur” en mémoire. Dans une version plus élaborée, c'est un dispositif électronique qui lit un ruban perforé contenant le chargeur : il suffit d'appuyer sur un bouton pour “recharger le chargeur”.

Encore mieux : le programme sur bande perforée peut servir à charger un système plus volumineux, depuis un autre périphérique plus rapide (bande, disque ou tambour magnétique). C'est ce qu'on appelle le **bootstrapping**, ou **amorçage** : la procédure de démarrage d'un ordinateur, qui comporte notamment le chargement du programme initial, et qui peut se faire en plusieurs étapes.

Ce programme résident comportait aussi des sous-programmes (par exemple lecture-écriture sur bande magnétique, sur disque etc.) qui pouvaient être appelés par les programmes des utilisateurs, et qu'il n'était donc pas nécessaire de recharger avec chaque travail.

12. Du moins présentées comme telles par le Directeur Informatique, qui trouve là un moyen d'asseoir sa position stratégique dans l'entreprise en négociant sa collaboration avec d'autres Directeurs..

### 1.3.2 Planification de travaux

Dans les entreprises, des informaticiens étaient chargés de la planification de l'exploitation : un certain nombre de travaux devaient "passer" sur l'ordinateur, à eux de décider quand et dans quel ordre, en tenant compte de diverses contraintes :

- la taille mémoire de chaque programme, et celle de la machine,
- les périphériques utilisés : sur une installation à 3 lecteurs de bandes, on ne peut pas faire tourner en même temps 2 programmes qui ont chacun besoin de 2 lecteurs.
- les priorités définies par l'entreprise<sup>12</sup>

et en optimisant à la fois la satisfaction de chaque service demandeur, et la rentabilisation des matériels.

**Exercice 4.** Soit une installation avec un ordinateur mono-tâche. A partir de 8h00, on doit faire passer trois travaux A, B, C qui durent respectivement 1h, 30 min, et 45min. Et les utilisateurs sont évidemment pressés d'obtenir les résultats.

Quel est le temps d'attente moyen des utilisateurs si on les fait passer dans cet ordre sur l'ordinateur (mono-tâche). Dans l'ordre inverse ? Quel est l'ordre optimal ?

### 1.3.3 Modifications matérielles nécessaires

Malheureusement, la coexistence en mémoire du "superviseur" et du travail utilisateur introduit de nouveaux problèmes. En effet, un programme utilisateur "buggé" (intentionnellement ou pas) peut

- altérer la partie de la mémoire réservée au superviseur, conduisant au plantage de la machine
- utiliser de façon incorrecte les instructions d'entrées-sorties (accès illégaux à des fichiers, périphériques endommagés, etc.)

Ceci a conduit à quelques modifications du processeur, suggérées dès la fin des années 50

- le processeur possède deux modes de fonctionnement "maître" (ou superviseur, ou privilégié) et le mode "esclave" (normal). Ceci est matérialisé par une bascule 1 bit.
- le superviseur s'exécute en mode maître, et les programmes utilisateurs en mode esclave.
- en mode maître, le processeur a accès à toute la mémoire, et peut exécuter toutes les instructions de la machine.
- en mode esclave, la zone mémoire accessible par le processeur est restreinte : deux registres indiquent le début de cette zone, et sont constamment comparés avec le compteur ordinal.
- en mode esclave, certaines instructions (par exemple les instructions d'entrée-sortie directes) ne peuvent pas être exécutées<sup>13</sup>.

- Quand un programme viole ces règles d'accès, une **exception** se produit : le contrôle est rendu au superviseur, à une adresse fixée au départ. Le superviseur examine donc la situation et décide des suites à donner (reprendre le programme, y mettre fin etc).
- pour appeler les sous-programmes du superviseur, un programme utilise une instruction spéciale ("syscall", trap logiciel, ...), qui provoque aussi une exception. Le superviseur se charge alors d'effectuer (en mode privilégié) l'opération demandée, avant de rendre la main au programme appelant.
- Quand une exception se produit, le contenu du compteur de programme est automatiquement sauvegardé, soit dans une pile en mémoire, soit dans un registre spécial. Ceci permet de reprendre éventuellement l'exécution là où elle en était arrêtée

### 1.3.4 Déroulement d'un appel système

**Point de vue du programmeur d'application** Par exemple, sous MS-DOS, pour faire afficher une chaîne de caractères il fallait

- placer le nombre 9 dans le registre AH
- placer l'adresse de la chaîne dans la paire de registres DS :DX
- appeler l'instruction INT 21H

**Déroulement** L'exécution de l'interruption 33 (21H) déclenche l'appel au système d'exploitation, dans la **routine de traitement de l'interruption 21H**. Ce sous programme consulte le registre AH qui indique la fonction demandée : 9 = affichage d'une chaîne. Une fois cette fonction affectée (par copie de caractères dans la mémoire de l'écran), le système place le code de retour 24H (36) dans le registre AL, et rend la main au programme utilisateur.

## 1.4 Profiter des temps morts : la multiprogrammation

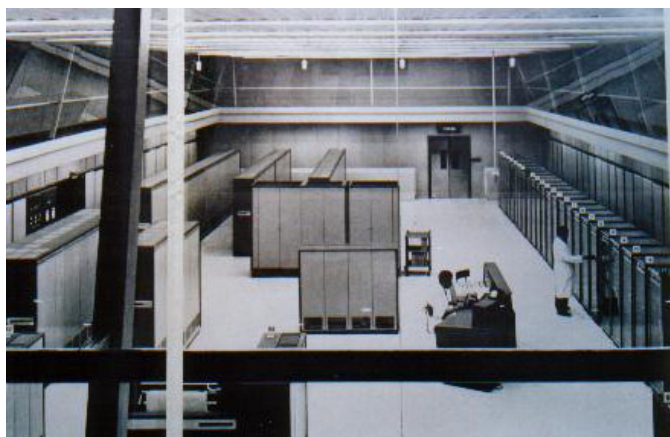
### 1.4.1 Motivation économique

A l'époque (début des années 60) les ordinateurs coûtent une fortune, on essaie donc de les rentabiliser au maximum.

Les machines sont composées d'une unité centrale (processeur et mémoire) et de périphériques : lecteurs de cartes perforées, de bandes, imprimantes etc. Or on observe que les opérations d'entrées-sorties sont extrêmement lentes par rapport aux possibilités d'un processeur.

Prenons par exemple le Gamma 60 dont le premier exemplaire a été livré par la société Bull à la SNCF en 1958, avec 6 imprimantes et 16 dérouleurs de bande, il occupait 360m<sup>2</sup>.

13. Dans le Stretch d'IBM, les ingénieurs avaient oublié de rendre privilégiée l'instruction qui permet de passer en mode maître. Doh !



source : <http://histoireinform.com/Histoire/+infos2/chr4infa.htm>

Pour les périphériques :

- le lecteur de ruban fonctionnait à 300 caractères par seconde
- les cartes perforées de 80 colonnes étaient lues à 300 cartes/mn
- les imprimantes fonctionnaient à 300 lignes par minute

et une instruction (opération sur nombres de 10 chiffres) prenait de 100 à 500 microsecondes, soit des dizaines de milliers par seconde.

Dans ces conditions, il est clair que le processeur est le plus souvent en attente d'une E/S.

L'idée est donc de faire cohabiter plusieurs tâches dans la mémoire : quand la tâche "active" demande à lire des cartes sur le lecteur, on met à profit le temps libre du processeur pour faire avancer une autre tâche.

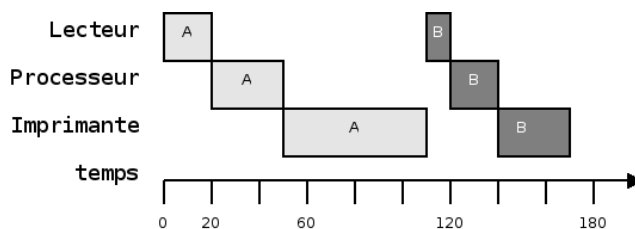
## 1.4.2 Étude d'un cas

Imaginons deux tâches A et B :

- la chargement de A depuis le lecteur de cartes dure 20 secondes, elle fait du calcul pendant 30 secondes, et l'impression des résultats prend 1 minute;
- le chargement de la seconde B dure 10 secondes, son calcul 20 secondes et l'impression 30 secondes.

Le graphique ci-contre montre ce qui se passe sous le contrôle d'un "moniteur d'enchaînement de travaux". La tâche B n'est chargée en mémoire que quand A s'est terminée ( $t = 110s$ ) et se termine à  $t = 170s$ .

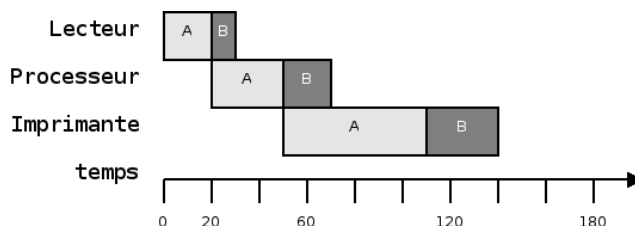
Le processeur a travaillé  $30 + 20 = 50s$ , soit un taux d'occupation de  $50/170 = 29,4\%$ .



### Exercice 5.

- calculez le taux d'occupation du lecteur de cartes
- calculez le taux d'occupation de l'imprimante.

Voici maintenant le déroulement dans un système multitâche; la tâche B est chargée dès que le lecteur a été libéré, puis est exécutée quand le processeur est libre, etc.



### Exercice 6.

- Calculez les taux d'occupation, comparez avec les chiffres précédents.
- Même question si on commence par exécuter B au lieu de A.
- Imaginons qu'il s'y ajoute une troisième tâche C semblable à B. Représentez le déroulement dans les deux cas (enchaînement séquentiel et multitâche). Comparez les chiffres.

### 1.4.3 La mise en oeuvre du multitâche

Pour mettre en oeuvre efficacement le multitâche, il faut que le processeur ne soit pas bloqué en attente des opérations d'entrée-sorties, qui doivent se dérouler en parallèle avec les calculs.

On confie donc le pilotage de périphériques à des circuits spécialisés, à qui le processeur enverra des commandes (requêtes d'E/S), et qui préviendront le processeur, par un **signal d'interruption**, quand la requête est terminée. Les interruptions sont traitées comme les exceptions vues plus haut.

Le fonctionnement par interruption décharge ainsi le processeur de la surveillance des périphériques, qui peut consacrer son temps à l'avancement des autres tâches.

L'idée des interruptions est apparue en 1955. La NASA possédait un Univac 1103 pour ses besoins de calculs scientifiques (traitement de données d'essais en tunnel de soufflerie avec Boeing) et ses applications administratives.<sup>14</sup>

Le programme de collecte de données était chargé en mémoire et présent pendant que les applications de gestion tournaient. Quand les tests en soufflerie étaient prêts, un bouton poussoir situé dans le hangar permettait d'activer le programme "instantanément"<sup>15</sup> : le contenu des différents registres de l'ordinateur (compteur ordinal, conditions, ...) était alors sauvegardé et le contrôle était transféré au programme de collecte de données.

## 1.5 Fonctionnement d'un centre de calcul

Les premiers systèmes multi-tâches sont toujours destinés au *traitement par lots (batch processing)*, mode d'exploitation qui est assez similaire à *'enchaînement automatique des travaux* décrit plus haut : on "enfourne" dans la machine une suite de travaux à réaliser, qui ressortent une fois terminés.

La différence, c'est qu'avec le multitâche, le travail N+1 peut être chargé en mémoire avant que le travail N ne soit terminé : on charge autant de programmes que possible pour remplir la mémoire, pour avoir un meilleur rendement et ne pas gaspiller le temps du processeur. Ils ne se terminent pas forcément dans l'ordre où ils ont commencé.

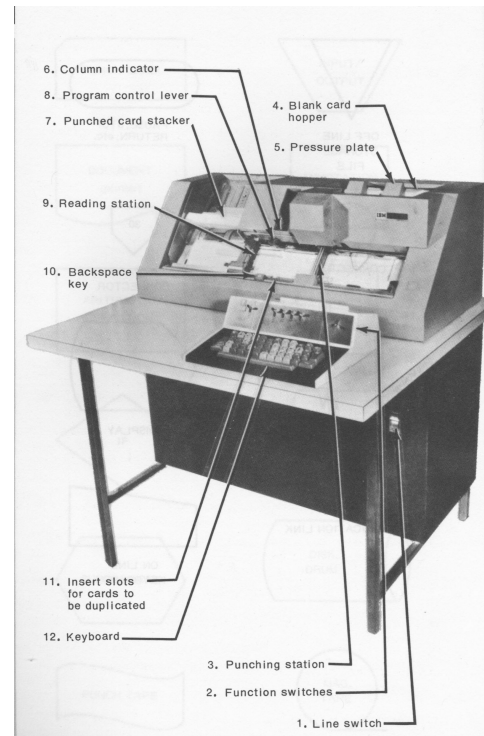
Dans les années 70, les étudiants en informatique de Bordeaux 1 travaillaient de la façon suivante

1. il fallait d'abord écrire le programme sur le papier
2. puis se rendre dans une salle où se trouvaient quelques perforatrices comme l'IBM 29 :

14. <http://www.cs.clemson.edu/~mark/interrupts.html>

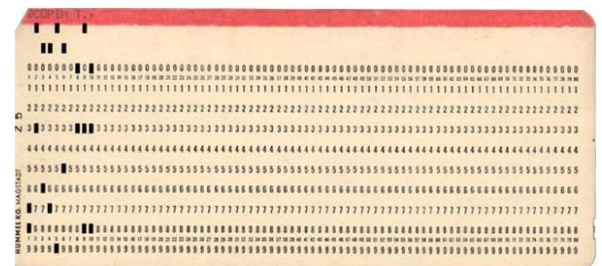
15. la solution précédente consistait à réserver l'utilisation de l'ordinateur à l'avance, et à téléphoner à l'opérateur pour qu'il lance le programme le moment donné

16. avec 4 perfos, il fallait évidemment faire la queue, un peu moins la nuit et le week-end



(source : <http://www.math-cs.gordon.edu/courses/cs323/FORTRAN/keypunch.html>)

pour transcrire le programme sur des cartes perforées<sup>16</sup>. Chaque carte contenait une ligne de 80 caractères.



3. les cartes, entourées par un élastique, étaient placés dans un bac qu'un étudiant (rémunéré) allait porter au Centre de Calcul Interuniversitaire, où se trouvait l'ordinateur IRIS 80, trois ou quatre fois par jour. Les bacs de cartes étaient alors confiés à un opérateur, qui les plaçait dans le lecteur de cartes, lançait le traitement et récupérait (bien plus tard) les cartes avec les listings de résultats.



Source <http://www.feb-patrimoine.com/projet/iris80/iris80.htm>.

4. il en profitait pour ramener les travaux précédents, avec les listings de résultats.
5. en récupérant son travail, l'étudiant constatait généralement qu'il manquait un point-virgule quelque part : ne restait plus qu'à trouver où, remplacer la carte fautive, et remettre le paquet dans le bac de départ, pour avoir le résultat quelques heures plus tard.

Dans ces conditions, il était évidemment préférable de réfléchir avant de taper, et de se relire soigneusement plusieurs fois avant de mettre les cartes dans le bac...

Les étudiants de troisième cycle, et les chercheurs en mathématiques et informatique, disposaient quant à eux d'une petite salle avec quelques **terminaux conversationnels** hétéroclites : telex Olivetti, terminaux à écran cathodiques (HP 2621), clavier avec imprimante thermique, écran graphique Textronix 4027, reliées directement au centre de calcul par des lignes à 9600 bit/s. De quoi travailler très confortablement.

## 1.6 Travailler à plusieurs : le temps partagé

En effet un nouveau besoin est apparu avec l'utilisation de terminaux interactifs : telex transformés, machines à écrire électriques, et écrans alphanumériques. Au départ les utilisateurs peuvent soumettre de nouvelles tâches dans le traitement par lots, mais les programmes interactifs amènent une contrainte supplémentaire : chaque utilisateur doit avoir l'impression d'utiliser une machine "réactive" : si un collègue lance un programme de calcul lourd (quelques milliers de décimales de  $\pi$ )<sup>17</sup>, ça ne doit pas empêcher les autres de travailler en monopolisant le temps du processeur.

C'est la prise en compte de cette contrainte qui conduit au *time sharing* : le temps du processeur est partagé "équitablement" entre les utilisateurs.

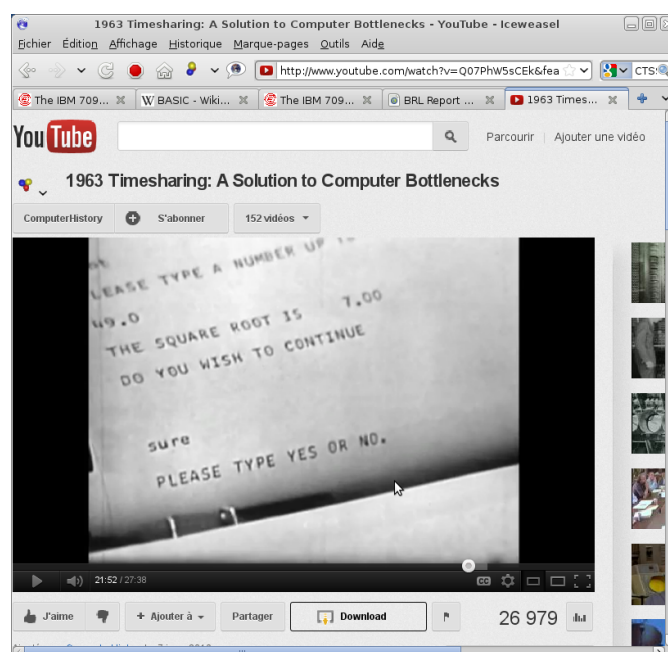
Parmi les premiers systèmes, le plus connu est CTSS (Compatible Time Sharing System) issu du projet MAC

17. ou un programme qui boucle, comme ça arrive parfois.

18. John McCarthy, décédé en octobre 2011, est un des pionniers de l'informatique : systèmes d'exploitation, intelligence artificielle, programmation symbolique et fonctionnelle, etc. Lire sa biographie sur Wikipedia. Parmi ses articles, un mémo "A Time Sharing Operator Program for Our Projected IBM 709" daté du 1er janvier 1959. Voir ses souvenirs sur le time-sharing dans <http://www-formal.stanford.edu/jmc/history/timesharing/timesharing.html>

(Multi Access Computer) de John McCarthy au MIT<sup>18</sup>. Ce système multi-utilisateurs (à partir de 1961, en production en 1964) tournait sur un IBM 7094 modifié, avec une mémoire de 2 fois 32K mots de 36 bits. Lire par exemple "The IBM 7094 and CTSS" par Tom Van Vleck, <http://www.multicians.org/thvv/7094.html>

**A voir absolument :** le reportage "1963 Timesharing : A Solution to Computer Bottlenecks" (27 minutes) sur <http://www.youtube.com/watch?v=Q07PhW5sCEk>, avec une longue interview de Fernando J. Corbato, responsable du projet, qui explique le fonctionnement du temps partagé, suivie par une démonstration.



Une démonstration plus courte <http://www.youtube.com/watch?v=sjnmckVnLi0> (Robert Fano explains scientific computing), à partir de 5 :20.

**Exercice 7.** Imaginons que 3 utilisateurs d'un système en temps partagé lancent en même temps des travaux qui nécessitent 10 minutes de calcul chacun.

1. si ces travaux sont envoyés dans une file d'attente pour être traités un par un, le premier utilisateur aura sa réponse dans 10 minutes, le second dans 20 minutes, et le troisième dans 30, d'où un temps d'attente moyen de 20 minutes ;
2. si ils se déroulent en temps partagé, ils dureront tous les trois 30 minutes.

Dans ce contexte, pourquoi les utilisateurs préfèrent-ils quand même la seconde solution ?



## 1.7 De nos jours...

Les systèmes d'exploitation modernes<sup>19</sup> sont tous capables de faire exécuter plusieurs tâches en même temps. Sous Unix, la commande `top` vous permet de voir les tâches en cours : sur un ordinateur personnel vous constaterez qu'il en a au moins une bonne centaine, plusieurs milliers sur un serveur.

Les solutions qui permettent le fonctionnement en multi-tâches ont été trouvées, mises en place et généralisées, dès le début des années 60 : interruptions, partage de la mémoire, etc.

Dans les années 70 et 80, on a assisté à un recul ap-

parent : les premiers micro-ordinateurs étaient destinés à un usage personnel.

Les contraintes de coût qui ne permettaient qu'une faible capacité mémoire (dizaines ou centaines de kilo-octets) expliquent la réapparition, pendant une dizaine d'années, des systèmes mono-tâches, comme CP/M (Kildall, 1977) et MS-DOS (1981).

**Exercice 8.** Avez-vous vraiment besoin d'un système multi-tâches, avec protection mémoire etc, dans votre smartphone ?

---

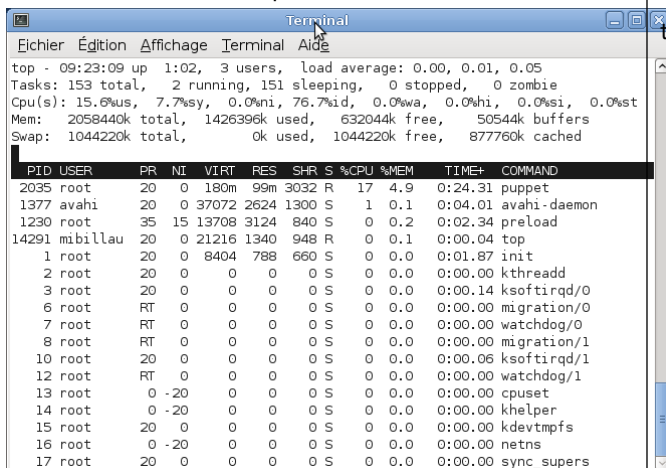
19. à l'exception de quelques systèmes embarqués ou de super-calculateurs

# Chapitre 2

## Les processus

### 2.1 Introduction

Les systèmes d'exploitation modernes sont tous capables de faire exécuter plusieurs tâches en même temps. Sous Unix, la commande `top` vous permet de voir les tâches en cours : sur un ordinateur personnel vous constaterez qu'il en a au moins une bonne centaine, plusieurs milliers sur un serveur.



```
top - 09:23:09 up 1:02, 3 users, load average: 0.00, 0.01, 0.05
Tasks: 153 total, 2 running, 151 sleeping, 0 stopped, 0 zombie
Cpu(s): 15.6%us, 7.7%sy, 0.0%ni, 76.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 2058440k total, 1426396k used, 632044k free, 50544k buffers
Swap: 1044220k total, 0k used, 1044220k free, 877760k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2035	root	20	0	180m	99m	3032	R	17	4.9	0:24.31	puppet
1377	avahi	20	0	37072	2624	1300	S	1	0.1	0:04.01	avahi-daemon
1230	root	35	15	13708	3124	840	S	0	0.2	0:02.34	preload
14291	mibillau	20	0	21216	1340	948	R	0	0.1	0:00.04	top
1	root	20	0	8404	788	660	S	0	0.0	0:01.87	init
2	root	20	0	0	0	0	S	0	0.0	0:00.00	kthreadd
3	root	20	0	0	0	0	S	0	0.0	0:00.14	ksoftirqd/0
6	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/0
7	root	RT	0	0	0	0	S	0	0.0	0:00.00	watchdog/0
8	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/1
10	root	20	0	0	0	0	S	0	0.0	0:00.06	ksoftirqd/1
12	root	RT	0	0	0	0	S	0	0.0	0:00.00	watchdog/1
13	root	0	-20	0	0	0	S	0	0.0	0:00.00	cpuset
14	root	0	-20	0	0	0	S	0	0.0	0:00.00	khelper
15	root	20	0	0	0	0	S	0	0.0	0:00.00	kdevtmpfs
16	root	0	-20	0	0	0	S	0	0.0	0:00.00	netns
17	root	20	0	0	0	0	S	0	0.0	0:00.00	sync_supers

Mais matériellement, un processeur ne peut exécuter qu'une instruction à la fois. Même si les ordinateurs possèdent plusieurs processeurs, on est loin du compte : il n'y a pas un processeur par programme.<sup>1</sup>

Le déroulement parallèle de ces tâches est donc une *illusion* : en réalité le processeur consacre un peu de temps "faire avancer" une tâche, puis passe à une autre etc. à tour de rôle. C'est la rapidité de cette alternance, quelques millisecondes par tâche, qui donne l'impression, à notre échelle, que tout avance en même temps.

Une des fonctions importantes d'un système multitâche est donc de gérer les *commutations de contexte* : il doit

- noter l'état de la tâche en cours (sauvegarde du contexte)
- choisir une des tâches (ordonnancement)
- la relancer dans l'état où elle était arrêtée (restauration du contexte)

tion du contexte)

### 2.2 Histoire

#### 2.2.1 Multitâche

Les systèmes d'exploitation multi-tâches sont apparus très rapidement dans l'histoire de l'informatique.

- Ordinateur Gamma 60 de la société Bull, en 1958  
[http://fr.wikipedia.org/wiki/Gamma\\_60](http://fr.wikipedia.org/wiki/Gamma_60).
- Ordinateur LEO III de la société Lyons, 1961.



La société Lyons regroupait une chaîne de restaurants, des hôtels et des activités dans l'alimentaire (biscuits). Grande utilisatrice de machines à cartes perforées pour sa gestion, elle a compris avant beaucoup d'autres l'intérêt des recherches qui étaient menées sur les calculateurs électroniques (EDSAC, à Cambridge). Elle a donc embauché un technicien radar, loin de son cœur de métier,<sup>2</sup> pour développer ses propres ordinateurs pour ses applications de gestion. Le LEO I (Lyons Electronic Office) est sorti en 1951, et d'autres modèles ont suivi qui ont été commercialisés hors de la société Lyons.<sup>3</sup>

1. Dans la suite du cours, pour simplifier, on considère des machines à un seul processeur. Avec plusieurs processeurs, il y a des complications intéressantes, mais les principes de base sont les mêmes.

2. Ce n'est pas un cas isolé : Honeywell (spécialiste de la régulation de chauffage) et Boeing (avions) ont aussi fabriqué des ordinateurs. Et plus tard le premier micro-ordinateur a été fabriqué en France pour l'INRA (Institut National de Recherche Agronomique), le Micral en 1972.

3. Sur le site <http://www.leo-computers.org.uk/> des anciens employés de LEO Computers, vous trouverez de nombreuses photos et descriptions (et même des enregistrements de l'ambiance des salles machine).



**L'objectif** du multi-tâches est évident : si plusieurs programmes s'exécutent en même temps, ils utilisent les divers périphériques en parallèle, ce qui rentabilise au mieux l'instal-

lation informatique : on augmente le nombre de programmes que l'on peut faire exécuter dans une journée d'utilisation.

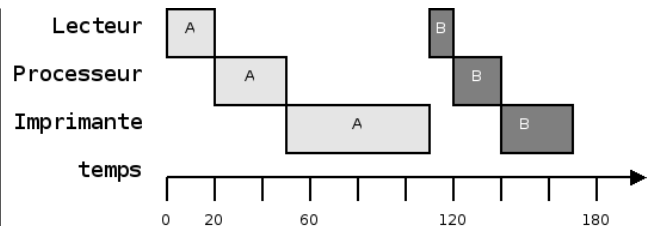
## 2.2.2 Exemple : intérêt du multi-tâches

Imaginons deux tâches A et B :

- le chargement de A depuis le lecteur de cartes dure 20 secondes, elle fait du calcul pendant 30 secondes, et l'impression des résultats prend 1 minute ;
- le chargement de la seconde B dure 10 secondes, son calcul 20 secondes et l'impression 30 secondes.

Le graphique ci-contre montre ce qui se passe sous le contrôle d'un "moniteur d'enchaînement de travaux". La tâche B n'est chargée en mémoire que quand A s'est terminée ( $t = 110s$ ) et se termine à  $t = 170s$ .

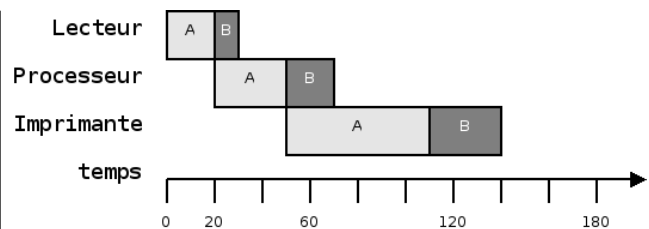
Le processeur a travaillé  $30 + 20 = 50s$ , soit un taux d'occupation de  $50/170 = 29,4\%$ .



### Exercice 9.

- calculez le taux d'occupation du lecteur de cartes
- calculez le taux d'occupation de l'imprimante.

Voici maintenant le déroulement dans un système multi-tâches ; la tâche B est chargée dès que le lecteur a été libéré, puis est exécutée quand le processeur est libre, etc.



### Exercice 10.

- Calculez les taux d'occupation, comparez avec les chiffres précédents.
- Même question si on commence par exécuter B au lieu de A.
- Imaginons qu'il s'y ajoute une troisième tâche C semblable à B. Représentez le déroulement dans les deux cas (enchaînement séquentiel et multi-tâches). Comparez les chiffres.

## 2.2.3 Du batch au time sharing

**Traitement conversationnel.** Un nouveau besoin apparaît à la fin des années 50 : avec l'utilisation de terminaux interactifs : telex transformés, machines à écrire électriques, et écrans alphanumériques. Chaque utilisateur doit avoir l'impression d'utiliser une machine "réactive" : si un collègue lance un programme de calcul lourd (quelques milliers de décimales de  $\pi$ )<sup>4</sup>, cela ne doit pas empêcher les autres de travailler en monopolisant complètement le temps du processeur.

C'est la prise en compte de cette contrainte qui conduit au *time sharing* : le temps du processeur doit être partagé "équitablement" entre les utilisateurs.

## 2.2.4 Support matériel du multi-tâches : les interruptions

Pour être réalisé, le multi-tâche nécessite l'ajout de quelques quelques fonctionnalités techniques sur le matériel.

En particulier, il ne serait pas raisonnable de devoir interroger constamment les périphériques pour savoir si les opérations qu'on leur a confiées (et qui sont attendues par certaines tâches) sont terminées ou non. Cette *boucle d'attente active* consommerait beaucoup de temps du processeur, temps que

4. ou un programme qui boucle, comme ça arrive parfois.

l'on souhaite consacrer à l'exécution de programmes "utiles".

L'ordinateur comporte donc des circuits spécialisés (contrôleurs de périphériques, en terminologie moderne) à qui le processeur confie l'exécution des entrées-sorties. Quand l'opération est terminée, le contrôleur envoie une **interruption**, signal électrique qui provoque le déroutement vers une "routine de traitement de l'interruption" située à une adresse convenue.

Dans un ordinateur multi-tâches, les interruptions "réveillent" le système d'exploitation, qui peut alors débloquer les tâches qui attendaient la fin de ces opérations.

D'autres mécanismes matériels sont également nécessaires pour la multiprogrammation, en particulier il faut *protéger l'espace mémoire de chaque tâche*, pour éviter qu'une autre tâche y accède indûment. La gestion de la mémoire fait l'objet d'un autre chapitre.

### 2.2.5 Support logiciel

Avant la multiprogrammation, les systèmes d'exploitation étaient des *moniteurs d'enchaînement de travaux* assez très rudimentaires chargés au début de la mémoire au démarrage de la machine, et dont le rôle était simplement :

1. de lire un programme exécutable (par exemple sur une bande magnétique) et de le copier un peu plus loin en mémoire,
2. de lancer son exécution,
3. et passer au suivant quand le programme s'est achevé.

Avec plusieurs tâches présentes simultanément, le système d'exploitation devient plus complexe : il doit traiter les interruptions provenant des périphériques, faire les commutations de contexte, gérer le partage de la mémoire, etc.

## 2.3 Définitions

### 2.3.1 Les processus

Le **processus** est une entité abstraite qui sert à représenter un **programme en cours d'exécution**.

Un processus regroupe :

- le **code** du programme : un espace mémoire contenant les instructions du programme ;
- un espace mémoire pour les **données** de travail (variables, pile, tas) ;
- d'**autres ressources** : descripteurs de fichiers ouverts, des ports réseau, etc.
- des **droits d'accès**

Le noyau du système d'exploitation détient une **table des processus** qui décrit l'état des processus présents dans la mémoire.

Pour chaque processus, il y a un **bloc de contrôle** (PCB, *process control block*) contient

- l'identifiant du processus
- son état : actif, prêt ou bloqué (voir plus loin)
- les valeurs des registres

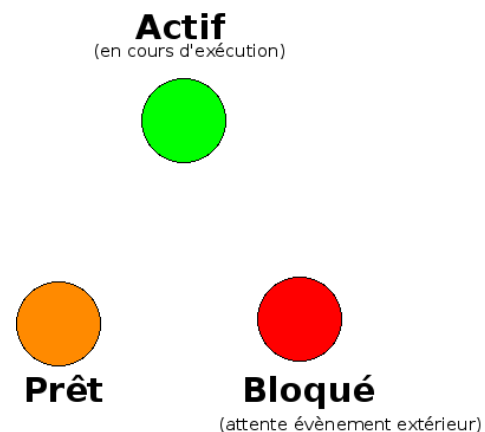
- le compteur ordinal (numéro de la prochaine instruction à exécuter)
- ...

Ces informations donnent la "photographie" d'un programme au moment où il a été interrompu, et permettront de reprendre son exécution exactement là où il était arrêté, avec le même contenu dans chaque registre.

### 2.3.2 Les états des processus

Trois états sont possibles pour un processus :

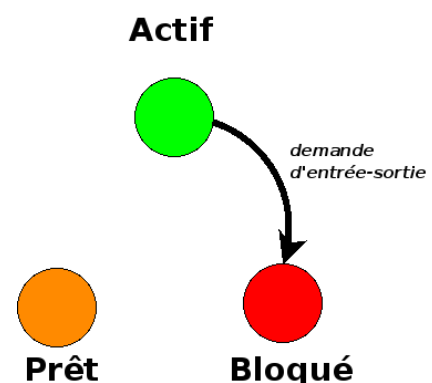
- **actif** quand le processeur est en train d'exécuter une de ses instructions. Dans un système mono-processeur, un seul processus peut être actif à la fois ;
- **bloqué** quand il est en attente d'un événement, par exemple une lecture de données ;
- **prêt** si il n'est ni actif ni bloqué.



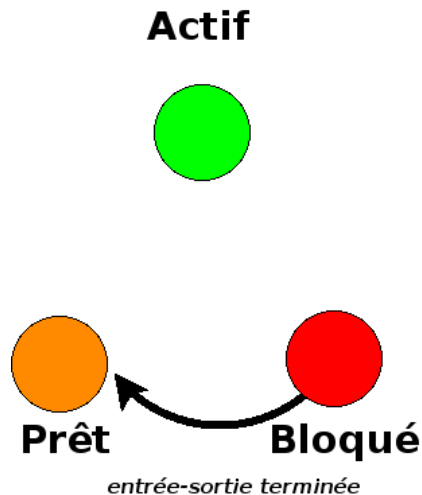
### 2.3.3 Changements d'état

1. Quand le processus actif a besoin de faire une opération d'entrée-sortie (E/S), il en fait la demande auprès du système d'exploitation par un "appel système" (sous Unix : `read`, `write`, etc.).

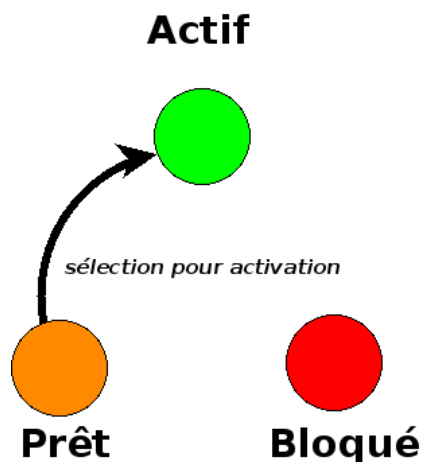
Si il faut attendre la fin de cette opération pour continuer (par exemple pour une lecture), le système change l'état du processus, qui devient **bloqué**. On parle d'opération *bloquante*.



2. lorsqu'un périphérique signale au processeur qu'une opération d'E/S est achevée, le système d'exploitation "débloque" le processus qui attendait la fin de cette opération. Le processus est alors marqué comme **prêt**.



3. enfin, le système d'exploitation peut choisir un processus prêt pour le rendre **actif**.

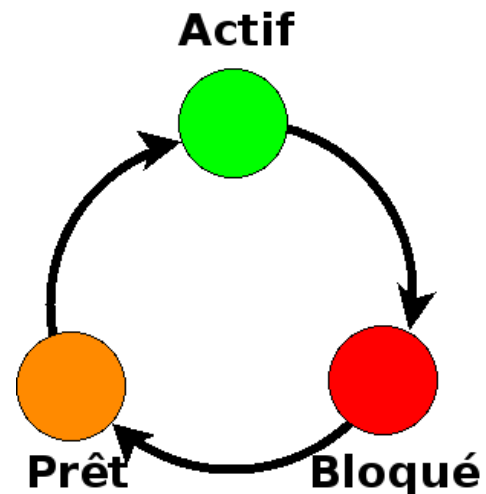


Ce changement peut se faire quand le processus actif se termine ou se bloque, et "laisse sa place".

À un moment donné il peut y avoir plusieurs processus prêts : le choix du processus à activer est fait par un module appelé **ordonnanceur (scheduler)**. Diverses **politiques d'ordonnancement** sont envisageables, nous les verrons en détail plus loin.

### 2.3.4 Multitâche coopératif

Dans un système multi-tâches dit "coopératif", les changements d'états d'un processus se font donc selon le cycle suivant, qui résume les transitions vues plus haut :



On observe que le processus qui est actif le reste tant qu'il ne demande pas d'opérations d'entrée-sortie.

Dans un tel système, Il est donc nécessaire que les programmes soient bien écrits pour

- ne pas boucler indéfiniment
- faire des entrées-sorties de temps en temps pour "bien se comporter" envers les autres processus et leur laisser des occasions de s'activer.

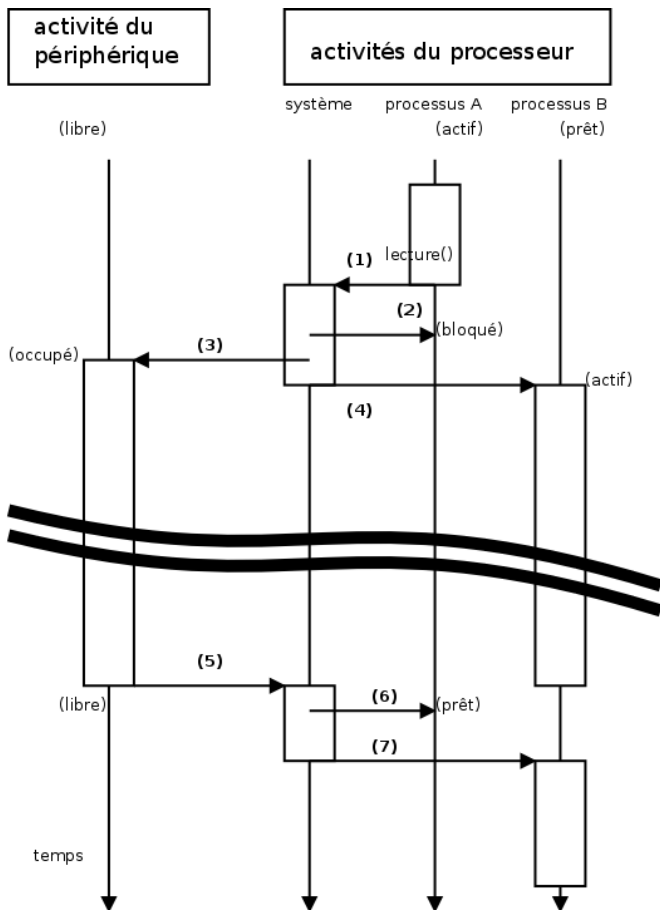
Évidemment ce genre de système marche assez mal en pratique. C'est ce qu'on trouvait dans Windows jusqu'à la version 3.11, et Mac OS jusqu'à MAC OS 9, plus de trente ans après l'invention du "vrai" multitâche !

### 2.3.5 Scénario détaillé

Le schéma ci-dessous montre le déroulement détaillé d'une opération d'entrée-sortie. On suppose qu'il y a au départ un processus A qui demande une lecture sur un périphérique inoccupé, et un processus B qui est prêt.

La scénario montre de haut en bas les "lignes de vie" des différentes activités : une pour le périphérique, et trois pour le processeur, pour distinguer ce qui relève du système, et des 2 processus.

Les flèches horizontales montrent les causalités.



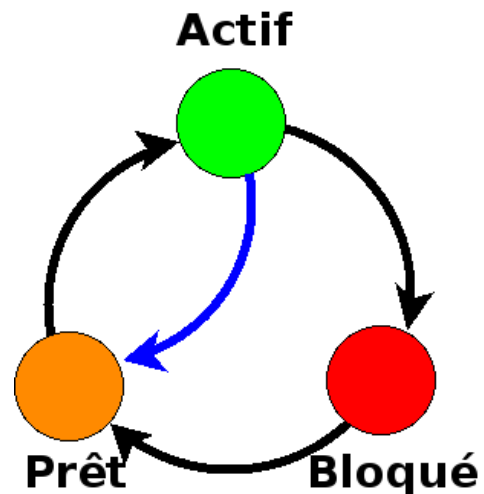
1. A fait un appel au système d'exploitation pour demander une lecture.
2. le système marque le processus A comme bloqué
3. il ordonne au périphérique de se mettre au travail
4. il active B
5. le périphérique signale la fin de l'opération
6. l'interruption rend la main au système d'exploitation, qui marque A comme prêt
7. le système rend la main au processus B

### 2.3.6 Multitâche préemptif

Le multi-tâches préemptif, qui traite correctement les problèmes de partage du temps, a été proposé très tôt par McCarthy et Teager. Dans un papier de 1959, McCarthy écrit que "l'idée n'est pas vraiment nouvelle".

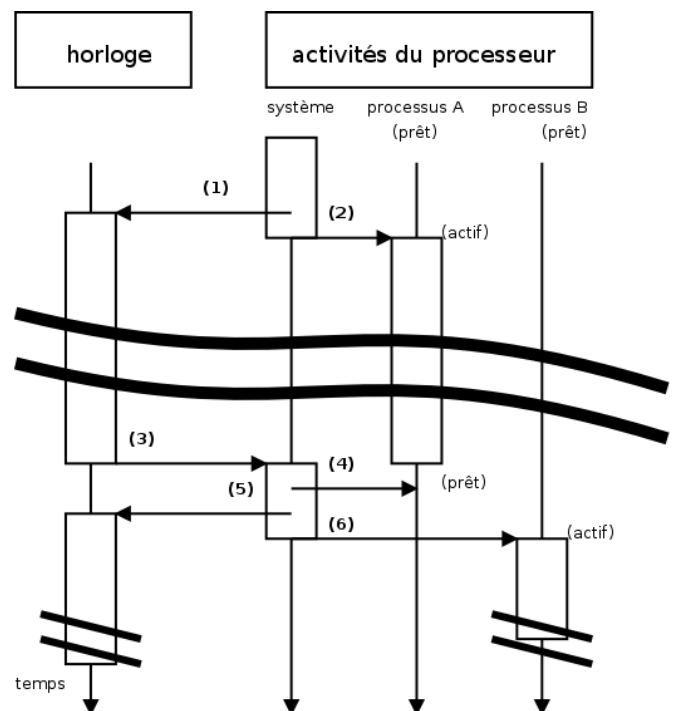
En fait, il suffit d'ajouter un circuit d'horloge qui envoie une interruption au bout d'un certain délai. Quand le système active un processus, il programme cette horloge pour un certain **quantum de temps**. Il peut alors se passer deux choses

- soit le processus actif fait un appel système pour faire une E/S ou se terminer, et donc il "passe la main", au moins provisoirement, au système d'exploitation.
- soit il fait uniquement du calcul, et se trouve donc interrompu, automatiquement, par le signal d'horloge quand le quantum de temps est épuisé. Le processus est alors marqué comme prêt.



C'est ce qu'on appelle une **préemption**, capacité pour le système d'exploitation d'interrompre une tâche en cours pour activer une tâche plus prioritaire. Notez que ceci n'interdit pas que le processus ainsi préempté soit aussitôt réactivé, si l'ordonnanceur détermine qu'il est le plus prioritaire.

Vu autrement, voici le déroulement d'une préemption, à partir du moment où l'ordonnanceur a déterminé qu'il fallait activer le processus A :



1. le circuit d'horloge est armé pour un quantum de temps fixé
2. le processus A est activé, et se met à faire du calcul "indéfiniment"
3. le quantum est épuisé, l'horloge envoie une interruption
4. le système reprend la main
5. le processus A est marqué "prêt"
6. le processus B est choisi et activé, l'horloge est armée.

Dans ce système dit “multi-tâches préemptif” on a donc la garantie que le système d’exploitation reprend la main régulièrement, ce qui lui donnera l’occasion de distribuer le temps équitablement entre les processus non-bloqués, sans laisser un processus actif monopoliser le processeur.

Le multitâche préemptif est présent sur tous les ordinateurs multitâches depuis les années 60. On mesure donc à quel point, pendant la décennie 1980-1990, les principaux systèmes pour micro-ordinateurs (Windows et MacOS) étaient littéralement préhistoriques.

En effet, ces petits systèmes ont revécu, trente ans plus tard, l’histoire de l’informatique :

- au départ, petites machines à capacité très limitées
- utilisées par un seul programme à la fois, une seule personne à la fois. Dans le cas de la micro-informatique, l’objectif commercial était de vendre un ordinateur par personne, surtout pas de le partager à plusieurs !

L’évolution vers le multi-tâche a été forcée par l’utilisation d’interfaces graphiques (si on a un multi-fenêtrage, on veut fatalement y faire tourner plusieurs programmes). Le multitâche coopératif est assez facile à réaliser par modification d’un système mono-tâche. Par contre le passage au multitâche préemptif nécessitait une refonte complète du système, ainsi que du catalogue d’applications.<sup>5</sup>

### 2.3.7 Multitâche préemptif et utilisation interactive

Soient deux utilisateurs X et Y d’un ordinateur en temps-partagé. Ils lancent tous les deux, à peu près en même temps, un programme qui fait une minute de calcul.

- Dans le cas d’un système coopératif, le calcul de l’un sera effectué pendant une minute, et alors commencera le calcul du second.
- Dans un système préemptif, les deux calculs alterneront par petites tranches correspondant au quantum de temps (valeurs courantes entre 1/50 et 1/1000 de seconde). Ils finiront donc tous les deux au bout de deux minutes.

En résumé, dans le cas coopératif, l’attente moyenne des deux utilisateurs sera  $\frac{60+120}{2} = 90s$ , et avec le système coopératif  $\frac{120+120}{2} = 120s$ .

**Exercice 11.** L’attente moyenne est-elle un bon critère pour mesurer la satisfaction des utilisateurs d’un système en temps partagé ? Pouvez-vous proposer mieux ?

5. On le sait assez peu, mais la société Microsoft prévoyait, après le lancement de DOS 3.0, de faire évoluer son catalogue vers Xenix, un système UNIX dont elle avait acheté les droits à ATT à la fin des années 70 de façon à pouvoir enfin remplacer DOS par un vrai système multitâches. Ce plan a été abandonné vers 1985, quand Microsoft et IBM ont commencé à développer ensemble OS/2 (nom de code “CP/DOS”) qui devait préserver la compatibilité avec les applications existantes. En effet, commercialement, il est mal avisé de forcer les clients à racheter tout leur parc logiciel pour bénéficier d’un nouveau système, fut-il techniquement bien meilleur. La stratégie OS/2 a aussi été abandonnée au profit du développement de Windows NT, qui regroupe en fait les versions Windows 2000, XP, 2003, Vista, Home Server, Server 2008, et Windows 7.

### 2.3.8 Interruptions et multitâche, en résumé

Une interruption est un signal qui détourne le processeur de sa boucle d’exécution normale (lire une instruction, l’exécuter, passer à la suivante), pour effectuer un traitement particulier (routine de traitement d’interruption).

Les interruptions sont causées par

- les **périphériques** (fin d’exécution de requête)
- des **signaux d’horloge**
- des **événements extérieurs**
- déroutements en cas d’**erreur** (accès illégal à la mémoire, division par zéro ...)
- **interruptions logicielles** provoquées par instruction spéciale

Dans un système d’exploitation multitâches, les interruptions sont traitées par le système, qui agit sur l’état des processus :

- venant d’un périphérique d’E/S : le système fait passer le processus demandeur à l’état prêt ;
- venant de l’horloge (épuisement du quantum de temps), le système fait passer le processus actif à l’état prêt ;
- interruption d’erreur (division par zéro etc) : le processus actif est supprimé.

## 2.4 Politiques d’ordonnancement

### 2.4.1 Ordonnanceur

Dans un système d’exploitation multi-tâches, l’**ordonnanceur** (*scheduler*) est un composant du noyau, qui a pour fonction de choisir un des processus prêts pour l’activer.

L’ordonnanceur applique une **politique d’ordonnement**, algorithme ou heuristique qui est censé donner “de bons résultats”.

### 2.4.2 Critères d’évaluation

Une politique d’ordonnement peut être évoluée selon plusieurs critères. On peut en effet souhaiter avoir

- **équité** : chaque processus dispose d’une part équitable du temps global,
- **aucun n’est empêché de tourner** (par exemple par une coalition de processus prioritaires)
- **minimiser le temps de réponse** : les utilisateurs interactifs souhaitent un système “réactif”. Quand ils lancent des commandes courtes, la réponse doit être rapide.
- **minimiser le temps d’exécution** : les commandes longues ne s’éternisent pas.
- **maximiser le rendement** : on peut lancer davantage de travaux dans la journée.

Mais malheureusement

- ces objectifs sont contradictoires,
- le comportement des processus ne peut pas être prévu

Il n'y a donc **pas de politique optimale** valable dans tous les cas. On se contente d'**heuristiques**, dont l'expérience montre qu'elles fonctionnent bien (ou pas) dans des contextes voisins.

Qui plus est, pour chaque méthode il sera souvent possible de construire un "scénario pathologique" pour lequel les choses se passent mal, du point de vue d'un critère particulier. La difficulté est d'évaluer la probabilité avec laquelle de tels cas peuvent se produire, dans le contexte d'utilisation visé, ainsi que les conséquences.

Que des programmes d'affichage puissent "lagger" quelques secondes de temps en temps n'a pas la même importance pour une animation en flash dans un navigateur, et sur une console d'aiguilleur du ciel.

### 2.4.3 Tourniquet

L'algorithme du **tourniquet** (synonymes : *round robin*, FIFO, premier arrivé-premier servi, ...)

**Principe :**

- le système détient une liste des processus prêts ;
- on choisit, pour l'activer, le premier processus de la liste ;
- à la fin de son quantum de temps, un processus actif est placé en fin de liste

**Propriétés :** cette heuristique garantit une **équité** entre les processus, qui ont tous une occasion de tourner.

**Exercice 12.** Soit trois processus A, B et C à comportements périodiques : ils font du calcul, une opération d'E/S et recommencent (un très grand nombre de fois).

- Pour A le calcul dure 10 ms, 10 ms pour B, et 45 ms pour C

- une opération d'E/S de 20 ms

Étudiez le déroulement pendant les 150 premières ms

- pour un ordonnancement circulaire (tourniquet) sans réquisition
- pour un ordonnancement avec réquisition (quantum 20 ms)

Comparez les taux d'utilisation des CPU dans les deux cas.

### 2.4.4 Priorités

Les priorités permettent de favoriser certains travaux.

On affecte un niveau de priorité (numérique) à chaque processus. Sous Unix, ce sont les numéros faibles qui sont les plus prioritaires.

**Principe**

- on choisit le processus de priorité la plus élevée
- si il y a plusieurs processus du même niveau, ils sont sélectionnés à tour de rôle (tourniquet)

**Propriétés**

- il y a un risque de **coalition** : si il y a beaucoup de processus prioritaires qui font du calcul, ils occupent le temps du processus à eux seuls, empêchant les autres processus de tourner.

**Exercice 13.** Soient deux processus A, B au fonctionnement cyclique : ils font du calcul (durées  $d_a$ ,  $d_b$ ), une entrée-sortie (durée  $d_{es}$ ), et recommencent. Un processus C, moins prioritaire, ne fait que du calcul. À quelle condition apparaît-il une coalition entre A et B qui empêche C de s'exécuter ?

**Exercice 14.** Soient trois processus A, B et C, qui ont un comportement répétitif : ils font un peu de calcul pour une durée  $t$  (respectivement 10ms, 15ms et 45ms), puis une opération d'entrée-sortie (qui dure 20ms), et recommencent. Le processeur d'entrées-sorties traite les requêtes séquentiellement, dans l'ordre où il les reçoit.

1. En supposant un ordonnancement par tourniquet sans réquisition, évaluez le taux d'occupation de la CPU, et du processeur d'entrées-sorties. Étudiez l'équité de l'ordonnancement.
2. Même question, avec un ordonnancement pré-emptif avec tourniquet, et un quantum fixé à 20 ms.
3. Même question, avec un ordonnancement pré-emptif avec priorités (dans l'ordre décroissant B, A, C), et un quantum fixé à 20 ms.

### 2.4.5 Priorités variables

En pratique on utilise des systèmes avec des priorités variables.

Exemple d'un tel système :

- chaque processus se voit accorder une priorité initiale (qui peut dépendre de l'utilisateur)
- la priorité baisse chaque fois que le processus termine son quantum de temps
- elle revient à son niveau initial après chaque entrée-sortie.

Ainsi

- les processus qui font beaucoup de calcul sont pénalisés, leur priorité baisse.
- les processus courts sont favorisés, ce qui donne aux utilisateurs interactifs une impression de réactivité.

- de même les processus qui font beaucoup d'E/S (et donc chargent peu le processeur), ont davantage d'occasions de tourner.

**Exercice 15.** Au département informatique nous avons eu un problème avec un système d'exploitation (SysVr4) qui remontait les priorités les processus qui faisaient des entrées-sorties.

Les étudiants devaient écrire du code pour afficher les requêtes d'une base de données, avec un algorithme du style

```
requete <= "select * from voitures"
curseur <= lancer-requete(req)

tant que code-erreur() == 100
  faire
    afficher valeur(curseur)
    avancer(curseur)
```

Dans ce code, les opérations sur la base de données communiquent par l'intermédiaire d'un "*pipe*" avec un processus qui accède effectivement aux fichiers de la base.

Certains étudiants ont inversé le test d'arrêt. Leur programme entrait donc dans une boucle de conversation avec l'autre processus, qui répondait immédiatement avec un code d'erreur différent de 100.

Quelles sont les conséquences pour les autres utilisateurs ?

## 2.4.6 Files multiples

On définit des **classes** de processus

- à chaque classe correspond une liste (FIFO) de processus

- chaque classe est sélectionnée régulièrement

On décidera par exemple d'avoir 3 classes : A (prioritaire), B (normal), C (non prioritaire), à qui on accordera respectivement 50 %, 30 % et 20 % du temps de calcul. Chaque classe sera gérée selon le principe du tourniquet,

Par exemple, pour les 5 premières activations l'ordonnateur choisira un processus prêt de la classe A, pour les 3 suivantes dans la classe B, etc, et recommencera.

Ce système a l'avantage

- de respecter les priorités,
- d'interdire les coalitions.

Dans l'exploitation en traitement par lots, les classes permettaient une segmentation de la clientèle des centres de calculs : les tarifs étaient plus élevés pour les travaux dans la classe la plus prioritaires. Les clients "en classe économique" devaient attendre les résultats plus longtemps.

## Chapitre 3

# Gestion de la Mémoire

### 3.1 Mémoire et multi-programmation

#### 3.1.1 Motivation

Rappelons que, dès les débuts de l'informatique, il y avait une motivation d'ordre économique : tirer le meilleur profit d'un matériel qui, à l'époque, était extrêmement coûteux.



(début années 60 : le calculateur Burroughs B5000)

On s'est aperçu rapidement que le goulet d'étranglement (*bottleneck*) du système informatique (processeur, mémoire, périphériques)



une unité de traitement ...

n'était pas la partie la vitesse de calcul du processeur, mais la lenteur relative des périphérique : la partie la plus coûteuse (le processeur) passait son temps à attendre la fin des entrées-sorties.



des périphériques ...

D'où l'idée de partager le temps de calcul entre plusieurs programmes, ce qui permet de mieux rentabiliser les équipements (meilleurs taux d'occupation).

Nous allons voir dans ce chapitre les techniques qui ont été mises en place au cours du temps pour permettre la présence de plusieurs processus en mémoire.

L'aboutissement en est ce qu'on appelle la **mémoire virtuelle**, qui est de nos jours, en général, une combinaison de techniques de *segmentation*, de *pagination* et de va-et-vient sur disque. à la demande.

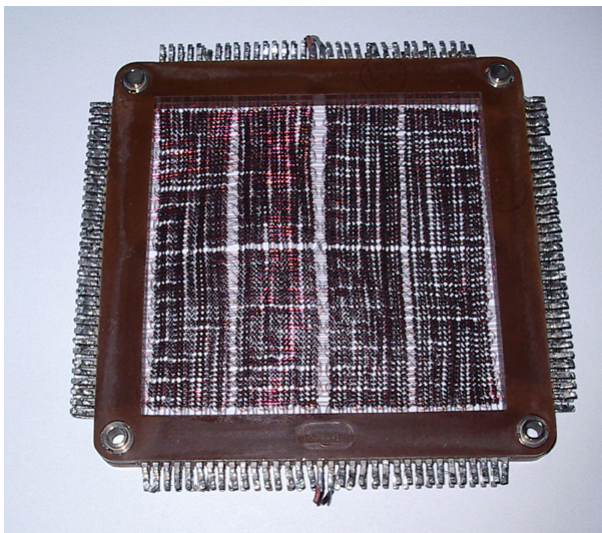
**Repères historiques :** le principe de la mémoire virtuelle a été exposé en 1962 dans un article de James Kilburn (Manchester) décrivant l'ordinateur Atlas. Dans les années 1970, cette technique était utilisée dans tous les ordinateurs<sup>1</sup>.

#### 3.1.2 Rappel : fonctionnement de la mémoire

Différentes technologies ont été utilisées pour réaliser les mémoires des ordinateurs : bascules bistables à base de tubes (triodes), mémoires à tores de ferrite, à transistors, circuits intégrés etc.

1. à l'exception de cas particuliers, comme les machines embarquées et certains super-calculateurs





mémoire à tores de ferrite) ...

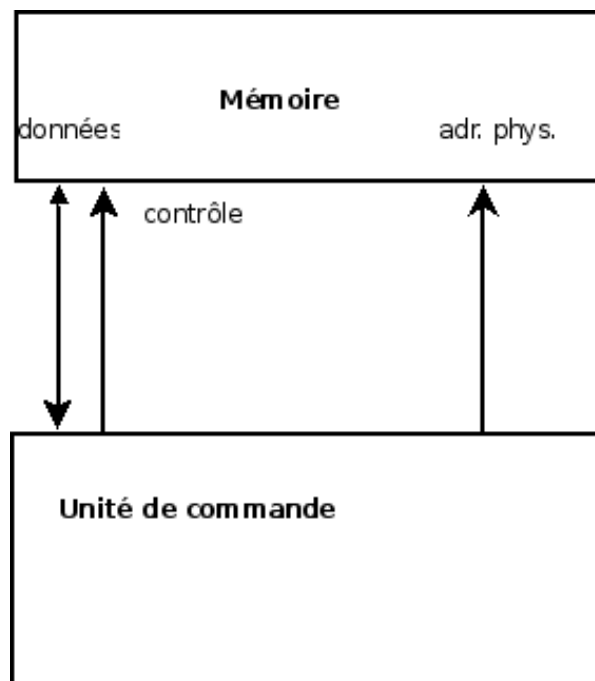
Indépendamment des technologies, la mémoire est un organe qui a pour fonction de stocker et restituer des "mots binaires" repérés par une adresse.

Les mots sont de taille fixe, par exemple l'ATLAS avait une mémoire de 16384 mots de 48 bits. Les micro-processeurs des années 75 étaient souvent des machines à octets (mot = 8 bits), de nos jours ce sont des mots de 32 ou 64 bits.

La mémoire communique avec le reste de l'ordinateur par 3 bus (groupes de fils)

- le **bus de contrôle** (il faudrait dire bus de commande), qui indique à la mémoire l'opération que l'on veut effectuer ;
- le **bus de données**, bidirectionnel, qui sert à émettre et recevoir les mots ;
- le **bus d'adresses**, qui indique à la mémoire l'adresse concernée.

2. **Description sommaire** : processeur très simple à un accumulateur. Les instructions contiennent l'adresse de l'opérande : pour charger/additionner/etc. une constante (zéro, un...) on utilise une variable contenant cette valeur. La comparaison *cmp* positionne des *indicateurs* (inférieur, supérieur, égal) qui sont utilisés par les instructions de branchement conditionnels (exemple : *bgt* = aller à une adresse si plus grand - *greater than*). L'instruction *b* est un branchement sans condition ; la directive *word* sert à réserver un mot, avec une valeur initiale.



La mémoire et ses trois bus

#### Opérations :

- **lecture** : pour consulter le contenu de la mémoire à l'adresse A, le processeur place le nombre A sur le bus d'adresses, et envoie le signal de contrôle "lecture". Après un petit délai de réponse, le contenu du mot d'adresse A est présenté par la mémoire sur le bus de données.
- **écriture** : pour envoyer un mot M à l'adresse A, le processeur place A sur le bus d'adresses, M sur le bus de données et active l'ordre d'écriture.

## 3.2 Adresses logiques et physiques

### 3.2.1 Le chargement des processus en mémoire

Pour lancer un programme, un système d'exploitation multi-tâche doit

- réserver de la place en mémoire
- y placer une copie de l'exécutable,
- ajouter une entrée dans la table des processus, et la marquer comme prête.

Un même programme pourra donc être chargé à des endroits différents, selon les programmes qui ont déjà été chargés.

Un nouveau problème se pose : un programme est composé d'instructions élémentaires. Voici un exemple de programme écrit en *langage d'assemblage* pour un processeur imaginaire.<sup>2</sup> Exemple de programme :

```

load    zero
store   somme
load    un
boucle :
store   i
cmp     n
bgt     fin      ; sauter à fin si >
add     somme
store   somme
load    i
b       boucle ; aller à boucle
fin :
halt
n      word 42
i      word 0
somme  word 0
zero   word 0
un     word 1

```

Les instructions font référence à des adresses. En supposant que chaque instruction tienne sur un mot, et que ce programme soit chargé à l'adresse 100, l'instruction de saut "b boucle" fait référence à l'adresse 103.

Si le codage de "b boucle" contient explicitement la constante 103, ce programme ne fonctionnera pas correctement si il est chargé à une autre adresse physique que 100.

**Exercice 16.** Faites tourner ce programme en fixant  $n$  à 4. Que fait-il, en général ?

Deux solutions ont été proposées pour ce problème

1. au moment du chargement d'un programme, modifier les mots qui contiennent des adresses pour tenir compte de début de la zone attribuée au processus.
2. différencier deux notions d'adresses :
  - l'*adresse logique* : le début de la boucle est le cinquième mot du programme (adresse logique 4, puisque la numérotation commence à 0)
  - l'*adresse physique* : ce mot se trouve dans la "case" 105 de mémoire.

La première technique est faisable sans modification du processeur, mais elle nécessite de garder une liste, à l'assemblage, des emplacements qui contiennent des adresses, pour que le chargeur puisse les modifier au chargement.

La seconde technique ne nécessite qu'une légère modification du processeur, que nous allons détailler ci-dessous.

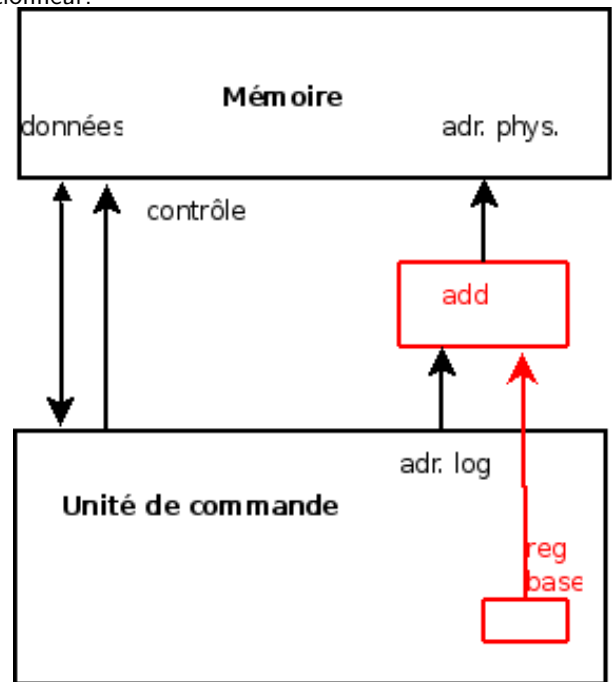
### 3.2.2 Adresses logiques

Avec les adresses logiques, tous les programmes sont compilés comme si ils devaient être chargés à partir de l'adresse 0 de la mémoire. Tout se passe comme si chaque programme utilisateur s'exécutait dans une mémoire qui lui appartient.

En réalité, les adresses logiques qu'il utilise sont traduites en adresses physiques par un circuit MMU (*memory manage-*

*ment unit*) avant d'être transmises à la mémoire : l'adresse logique est ajoutée au contenu d'un **registre de base** contenant l'adresse de début de la zone mémoire allouée au processus.

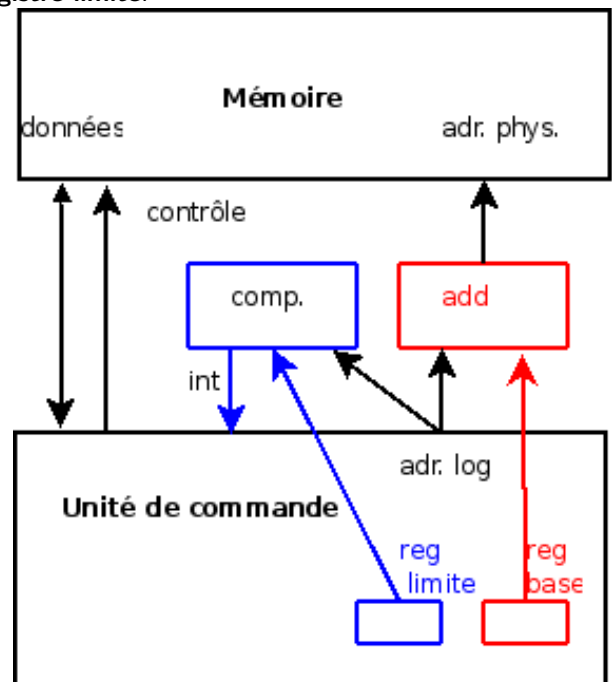
Cette modification ne nécessite qu'un registre et un additionneur.



avec registre de base

### 3.2.3 Protection mémoire

Quelques circuits supplémentaires fournissent une protection de la mémoire : un **comparateur**<sup>3</sup> vérifie que l'adresse logique émise par le processeur est inférieure au contenu d'un **registre limite**.



3. c'est un soustracteur où on ne regarde que le signe du résultat

avec registre limite

Ceci garantit que le programme utilisateur reste confiné dans son espace mémoire.

### 3.2.4 Memory Management Unit

À ce qui précède il convient d'ajouter qu'en mode privilégié le processeur a normalement accès à toute la mémoire, avec des instructions privilégiées qui lui permettent d'accéder explicitement à des données de l'espace utilisateur.

Par exemple, un appel système permet à un programme utilisateur de demander l'écriture d'une chaîne de caractères, en précisant dans un registre l'adresse de début du tableau de caractères. Cet appel système donne la main au système d'exploitation, qui doit interpréter le contenu de ce registre comme une adresse dans l'espace utilisateur, et non dans le sien.

On peut donc voir la MMU comme un circuit ayant

— **comme entrées :**

- une adresse logique AL
- le contenu du registre de base RB,
- le contenu du registre limite RL
- un bit EU précisant l'espace d'adressage utilisé (= 1 si utilisateur, ou instruction spéciale d'accès à l'espace utilisateur en mode privilégié)

— **comme sorties :**

- une adresse physique AP
- un signal INT d'interruption si une violation mémoire s'est produite

— les **équations logiques :**

$$\begin{aligned} \text{AP} &= \text{si EU} \\ &\quad \text{alors AL} + \text{RB} \\ &\quad \text{sinon AL} \\ \text{INT} &= \text{EU et (AL} \geq \text{RL)} \end{aligned}$$

## 3.3 Gestion d'un espace mémoire linéaire

Dans ce qui précède, chaque processus dispose d'un "espace d'adressage linéaire" (ou plat) : l'ensemble des adresses logiques est une séquence de nombres, sans trous ; et il lui correspond un "bloc" d'adresses contiguës en mémoire physique.

Dans un contexte de multiprogrammation, il va falloir gérer l'allocation et la libération de ces blocs en mémoire, au fur et à mesure que les programmes sont chargés et déchargés de la mémoire.

Pour simplifier l'explication, on se limite ici à un bloc par processus, mais sur de nombreuses machines, le processeur comporte plusieurs paires de registres base+limite, et

les adresses logiques se réfèrent à l'un ou l'autre des "segments" : segment de code, segment de données, segment de pile, etc.<sup>4</sup> La problématique sera la même : où charger les segments quand on en a besoin ?

### 3.3.1 Exemple

Supposons que nous disposions d'une mémoire centrale de 64K mots (confortable pour les années 60)

Espace mémoire



Espace mémoire de 64K mots

et que nous souhaitions y charger 3 programmes de tailles respectives 10 K, 30 K et 15K. Une idée assez naturelle est de réserver un espace mémoire consécutif pour chaque programme, et de placer les programmes les uns à la suite des autres.

Espace mémoire



Placement consécutif

Cette stratégie d'**allocation contiguë** paraît simple, mais elle pose un problème. A priori, on ne sait pas dans quel ordre les processus vont se terminer. Si c'est P1 qui se termine le premier, la mémoire libre sera de 19K, mais il sera impossible de placer un processus de 19K, puisque nous aurons deux blocs libres de 10K (anciennement occupé par P1) et 9K, qui ne sont pas consécutifs.

Il faudra donc

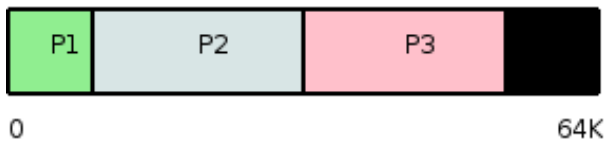
- prendre des mesures si il se produit (déplacer les processus ?)
- essayer d'éviter que ce problème ne se produise, du moins trop souvent,
- ou alors remettre en cause l'idée de l'allocation contiguë.

### 3.3.2 Problème : la fragmentation

Le scénario ci-dessous montre l'évolution de l'usage de la mémoire pendant qu'apparaissent et disparaissent des processus de tailles diverses :

4. Ceci permet à la fois d'étendre les capacités d'adressage en mode utilisateur (une adresse sur 16 bits donne théoriquement accès à 64K mots dans un espace : avec 4 segments c'est 256K), et aussi de partager des segments entre plusieurs processus. Par exemple, sur un système en temps partagé, on n'aura en mémoire qu'un seul exemplaire du segment de code pour l'éditeur de textes, utilisé par des dizaines de personnes simultanément.

Espace mémoire



(0) 3 processus sont présents, P2 se termine ...

Espace mémoire



(1) allocation de P4 ...

Espace mémoire



(2) allocation de P5 ...

Espace mémoire



(3) libération de P4 ...

Espace mémoire



(4) l'espace libre est fragmenté

L'allocation et la libération de blocs de tailles diverses provoque une **fragmentation** de l'espace mémoire disponible. Cette fragmentation peut rendre impossible le chargement en mémoire d'un processus alors que l'espace libre total est en théorie suffisant.

### 3.3.3 Solutions curatives

Quand cette situation se produit, on peut envisager de décaler les blocs alloués pour les ramener en début de mémoire, et ainsi regrouper toutes les zones libres en une seule. C'est possible si le processeur distingue adresses logiques et

adresses physiques; le code des programmes est alors "relogeable" (*relocatable*) : il suffit de le recopier ailleurs et de changer le contenu du registre de base.

L'inconvénient de cette opération de "ménage" est qu'elle oblige à interrompre les processus concernés pendant ce temps, provoquant des à-coups désagréables dans un contexte d'utilisation interactive.

**Exercice 17.** Supposons qu'un ordinateur de 2Mo effectue un compactage toutes les secondes. Si il faut  $0.5\mu s$  pour copier un octet, et si la taille moyenne des zones libres est égale à 0.4 fois celle des zones allouées, quelle fraction du temps du processeur est consacrée au compactage ?

### 3.3.4 Solutions préventives

L'exemple ci-dessus illustre la stratégie d'allocation dite "*first-fit*" : les zones libres sont triées par ordre d'adresses, et on choisit la première zone qui soit assez grande. Une partie de cette zone, correspondant à la demande, est allouée, et le reste est replacé dans la liste des zones libres.

L'algorithme du *meilleur ajustement* (*best fit*) consiste à retenir la plus petite zone qui soit assez grande. Son intention est d'éviter de fractionner les grands blocs. Cependant les simulations montrent qu'il fait perdre davantage de place que le "*best fit*" : en effet il a tendance à créer de multiples petites zones libres, trop petites pour être utilisables.

Inversement, l'algorithme du *plus grand résidu* (*worst fit*) choisit systématiquement le plus grand bloc libre. En pratique, ça ne donne pas non plus de bons résultats.

**Exercice 18.** Allocation contiguë La mémoire d'un système contient des zones libres de 10K, 4K, 20K, 18K, 7K, 9K, 12 K et 15K mots (dans l'ordre des adresses).

- Quelles zones l'algorithme *first fit* choisit-il pour des demandes successives d'allocation de (a) 10K, (b) 12K, (c) 9K.
- Même question pour les deux autres algorithmes.

### 3.3.5 Buddy system (blocs compagnons)

Une autre approche : on évite de gérer des blocs de tailles trop différentes. Pour cela on arrondit les demandes d'allocation mémoire à la puissance de 2 supérieure (au dessus d'une valeur minimale raisonnable, par exemple 1K octet).

Pour chaque taille, on maintient une liste de blocs :  $L_{10}$  (blocs de  $2^{10} = 1024$  octets),  $L_{11}$  (2048 octets), etc. qui commencent tous à une adresse qui est un multiple de leur taille. Par exemple les blocs de  $L_{11}$  peuvent commencer à l'adresse 0, 2K, 4K, 6K, etc.

**Pour allouer** un espace de 3000 octets, la demande est arrondie à la puissance de 2 supérieure, soit  $2^{12} = 4096$ .

On prend donc un bloc libre dans la liste  $L_{12}$  des blocs libres de 4096 octets (si il n'y en a pas, on prendra dans la liste du dessus  $L_{13}$ , etc.) commençant par exemple à l'adresse  $12288 = 3 \times 2^{12}$ .

L'excédent par rapport à la demande est de  $4096 - 3000 = 1096$  : on peut en retirer un bloc de 1024 octets qui est remis dans  $L_{10}$ . Le bloc réellement alloué est de  $4096 - 1024 = 3072$  octets.

**Pour restituer** ce bloc de 3072 octets, il est fragmenté en un bloc de taille 2048 ( $2^{11}$ ) commençant à l'adresse  $12288 = 3 \times 2^{12} = 6 \times 2^{11}$  qui est remis dans  $L_{12}$ , et un bloc de 1024  $2^{10}$  commençant en  $12288 + 2048 = 3 \times 2^{12} + 2^{11} = 14 \times 2^{10}$  destiné à  $L_{10}$ .

Or ce dernier bloc est la moitié d'un bloc deux fois plus gros. Si l'autre moitié (son bloc compagnon) - qui commence en  $15 \times 2^{10}$  - est libre, on peut les fusionner en un bloc libre de  $2^{11}$  qui ira dans  $L_{11}$ .

Lors de la restitution, le bloc de 640 est à nouveau fragmenté en  $B_1$  (512) et  $B_2$  (128). On consulte la liste  $L_7$  pour voir si le *bloc compagnon*  $B'_2$  de  $B_2$  y figure pas ( $B_2$  est la moitié d'un bloc de 256 octets, dont le compagnon est précisément l'autre moitié). Si il y est, on en profite pour fusionner les deux compagnons pour faire un bloc libre de 256, dont on cherche le compagnon dans  $L_8$ , etc.

En pratique, ce système donne de bons résultats.

### 3.3.6 Partitions de taille fixe

Une autre approche est l'utilisation de partitions de taille fixe.

Au démarrage, l'administrateur fixe un partage de la mémoire en partitions. Dans une partition ne tournera qu'un programme qui lui sera affecté soit explicitement, soit automatiquement (la plus petite partition assez grande).

Le compactage est évité au prix d'une certaine perte de place.

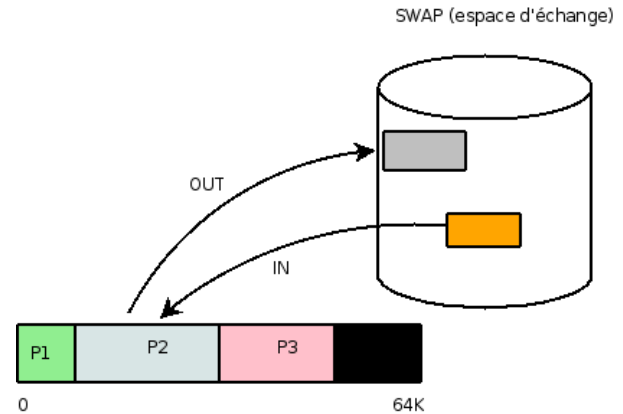
Au début des années 80 le département de maths-info de Bordeaux 1 possédait un mini-ordinateur LSI 11-23 (Plessey), clone du PDP 11-23 de DEC, qui fonctionnait sous TSX-Plus. Avec ses 256 Ko de mémoire, il permettait à 5 ou 6 utilisateurs de travailler en temps partagé depuis des terminaux alphanumériques ou graphiques (Tektronix 4014 à écran phosphore, très utilisé par les maths-applis, ou 4027 en couleur) reliées par des lignes série. Au démarrage, l'administrateur pouvait fixer la taille de la partition mémoire allouée à chaque poste de travail.

## 3.4 Va-et-vient sur disque

Dans un contexte d'utilisation en temps partagé conversationnel, on s'aperçoit vite que l'utilisateur humain est de loin le périphérique le plus lent du système. Il lui arrive de

lancer un programme (qui prend de la place en mémoire) et de partir prendre un café (ou partir en week-end) alors que le programme attend une réponse.

Dans ces conditions, on peut avoir l'idée de "mettre de côté" son programme en faisant une copie sur disque de l'espace mémoire qu'il utilise, espace qui sera bien plus utile si on le prête à des programmes qui tournent vraiment.



Échanges mémoire ↔ disque

Quand l'utilisateur reviendra et se décidera à appuyer sur une touche, le système ramènera le programme en mémoire, et poursuivra son exécution.

Dans la mesure où un accès disque ne prend que quelques centièmes de seconde, le va-et-vient (swapping) du programme entre la mémoire vive et le disque est imperceptible pour l'utilisateur interactif.

Économiquement, il faut aussi considérer le ratio des prix entre mémoire centrale et disque : pour fixer les idées, début avril 2012, 1 Go de mémoire vive coûte environ 20 euros, et un disque de 1 To environ 100 euros, soit un rapport de 200.<sup>5</sup>

Le *swapping* est donc une manière économique d'étendre la capacité mémoire d'une machine, pour y faire tourner davantage de programmes "simultanément".

## 3.5 La mémoire segmentée

Nous avons déjà vu l'idée de différencier adresses logiques et physiques dans le cas d'un adressage "plat" (linéaire) : à chaque processus correspond un "bloc" (ou segment) d'adresses contiguës de la mémoire physique qui sont vues, par chaque processus utilisateur comme un espace logique commençant à l'adresse 0.

L'astuce était d'employer un *registre de base* qui indique l'emplacement physique du premier mot de cette zone ; l'adresse physique d'un mot s'obtient en ajoutant l'adresse logique au contenu de ce registre. Par exemple, si le registre de base contient 1734, le mot d'adresse logique 55 est situé à l'emplacement  $1734 + 55 = 1789$  de la mémoire physique.

5. Ce rapport a évidemment beaucoup fluctué dans l'histoire de l'informatique, mais il est clair qu'il revient moins cher de "loger" les processus inactifs sur disque qu'en mémoire vive. Et le gain était d'autant plus clair à une époque où les sommes en jeu n'étaient pas des dizaines d'euros, mais des dizaines de milliers de dollars.

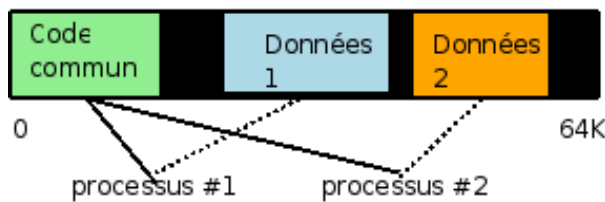


### 3.5.1 Segments

Le besoin est vite apparu de permettre au processeur d'accéder à plusieurs segments. Par exemple, dans un usage interactif, il se peut que plusieurs utilisateurs fassent tourner le même programme. Dans ce cas, il serait dommage de charger autant d'exemplaires du code exécutable que d'utilisateurs ; il est plus judicieux de considérer que l'espace mémoire d'un programme se compose de deux segments :

- le segment de code contenant les instructions (qui peut être partagé),
- et le segment de données où se trouvent les variables propres à chaque instance du programme.

Espace mémoire



Deux processus exécutent le même code

Plus généralement, les programmes présents en mémoire, même si ils sont différents, font souvent appel à des bibliothèques (*libraries*), par exemple les fonctions mathématiques. Chaque bibliothèque peut être chargée en mémoire comme un segment de code qui sera éventuellement partagé.

Ce besoin est apparu très rapidement : la première machine commerciale à utiliser la segmentation est le B5000 de Burroughs, sorti en 1961.

### 3.5.2 Espace d'adressage

On s'éloigne donc de l'adressage plat, où les adresses logiques d'un processus sont dans un intervalle  $[0..n]$ . Dans une mémoire segmentée, les adresses sont maintenant formées d'un couple (s,p) avec

- un numéro de segment s
- une position p dans le segment (on parle d'*offset*, ou de déplacement)

Le numéro de segment est toujours placé dans les bits de poids forts de l'adresse.

**Exemple, le PDP 10** utilisait des adresses de 18 bits comportant

- un bit pour le numéro de segment
- 17 bits pour la position dans le segment

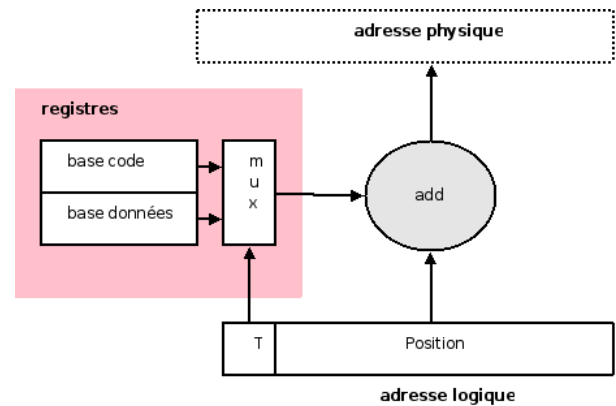
ce qui veut dire que l'espace mémoire logique se décomposait en 2 segments, et que le processeur comportait deux registres de base (et deux registres limite).

numéro de bit	17	16	15	...	1	0
	s	p	p	..	p	p

d'au plus  $2^{17} = 128\text{ K}$  mots chacun. Les adresses du "segment bas" commencent à 0 à 128K-1, celles du "segment haut" à partir de 128K. L'espace d'adressage n'est plus

linéaire : si le processus a un segment bas de 4K et un segment haut de 8K, les adresses logiques valides sont dans deux plages séparées : de 0 à 4K-1 et de 128K à 136K-1.

Selon la valeur du bit 17, on sélectionne l'un ou l'autre des registres de base



un multiplexeur sélectionne le registre de base à utiliser

**Exercice 19.** Soit une machine dont les adresses, sur 24 bits, comportent 4 bits pour le numéro de segment.

1. combien y a-t-il de segments logiques ?
2. quelle est la taille maximum d'un segment ?
3. quelle est la taille de l'espace d'adressage logique ?
4. quelle est l'adresse logique correspondant à l'octet 2 du segment 5 ?
5. donnez le numéro de segment et l'offset de l'adresse logique 0x600042.

### 3.5.3 La réalisation

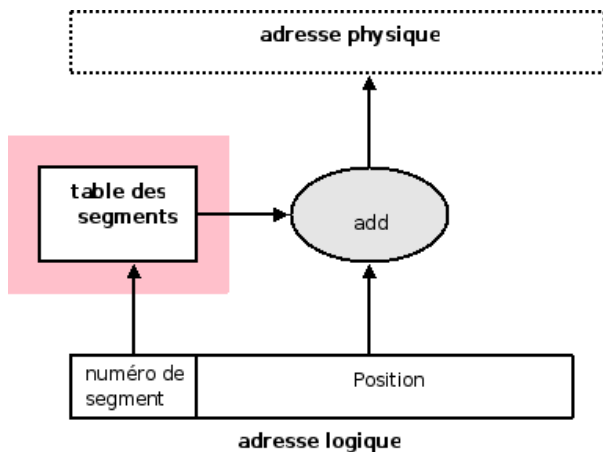
Prenons le cas d'une machine dont les adresses sur 20 bits, (notées  $AL[19:0]$ ) comportent 2 bits pour le numéro de segment  $AL[19:18]$  et 18 bits  $AL[17:0]$  pour le déplacement.

Cette machine a donc une table de segments composée de 4 registres de base  $RB[i]$  et autant de registres limites  $RL[i]$ .

L'adresse logique se calcule donc en additionnant

- le contenu du registre correspondant au numéro de segment
- le déplacement contenu en partie basse de l'adresse

$$AL = RB[AL[19:18]] + AL[17:0]$$



utilisation d'une table des segments

et une violation de protection mémoire est détectée si le déplacement atteint ou excède la limite du segment.

INT = 1 si AL[17:0] >= RL[AL[19:18]]  
0 sinon

**La taille des registres limite** est logiquement, sur cet exemple, de 18 bits, puisqu'ils servent à vérifier un déplacement qui est lui-même exprimé sur 18 bits. Par contre on se sait rien sur la taille des registres de base. En effet, si l'espace adressable par un processus est limitée à  $2^{20} = 1\text{M}$  mots à travers 4 segments de 256K mots, la mémoire physique peut être plus grande pour accueillir plusieurs processus. Par exemple, on peut avoir des adresses physiques sur 24 bits, ce qui donne une capacité de 16M mots pour la mémoire physique.

**Exercice 20.** Le processeur 8086 utilisait 4 registres de base CS, SS, DS et ES<sup>a</sup>, des déplacements sur 16 bits, et des adresses physiques sur 20 bits. Les segments commençaient à des adresses multiples de 16, les registres de base contenaient les 16 bits de poids fort qui étaient virtuellement complétées par 4 zéros binaires.

1. quelle est la capacité maximum de la mémoire physique ?
2. quelle est la taille de l'espace d'adressage d'un programme à un moment donné ?<sup>b</sup>
3. si CS contient 0x5432, quelle est l'adresse physique correspondant à l'adresse 0x0123 du segment de code ?

a. segments de code, de pile, de données et supplémentaire  
b. les programmes pouvant modifier leurs registres de base, ils ont accès à tout

### 3.5.4 Tables locale/globale des segments

En réalité c'est un petit peu plus compliqué. Si un programme a été compilé pour utiliser 3 segments - par exemple

0 = le code, 1 = les données et 2 = la bibliothèque mathématique -, la numérotation utilisée pour les segments est **locale** au processus : la bibliothèque mathématique peut être le segment 2 de ce processus, et le segment 4 d'un autre.

Par contre, les informations sur les segments (position en mémoire, taille, attributs de protection,...) sont **globales** à tout le système.

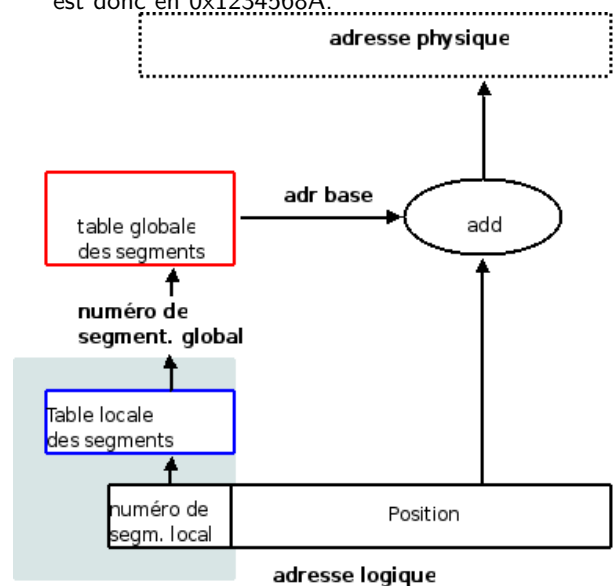
On utilise donc deux tables

- la **table globale des segments** (TGS), qui garde trace de tous les segments chargés en mémoire
- une **table locale des segments** (TLS) pour le processus en cours, dans laquelle se fait la correspondance entre le numéro de segment utilisé dans les adresses logiques, et le numéro global de segment

La traduction d'une adresse logique (s,d) passe donc par

- la consultation de TLS[s] pour trouver le numéro g de segment global correspondant à s. Par exemple on trouvera que le segment local 2 est le segment global numéro 17.

- la consultation de la TGS[g] pour récupérer l'adresse de base et la taille du segment. Par exemple le segment global 17 commence en 0x12345678.
- l'addition de l'adresse de base au déplacement pour obtenir l'adresse physique. L'adresse 12 du segment 2 est donc en 0x1234568A.



Génération d'adresses, TLS + TGS

La table globale des segments contient des **descripteurs**, qui indiquent pour chaque segment

- la position du segment en mémoire
- sa longueur
- les droits d'accès (lecture, écriture, exécution)

En effet il est assez simple, au niveau du processeur d'émettre des signaux qui indiquent son état (recherche d'une instruction, exécution, ...) La MMU peut donc vérifier que le processeur n'essaie pas d'exécuter des instructions dans un segment qui est censé ne contenir que des données, ce qui correspond à une lecture en mémoire pendant la phase d'exécution sur un segment non-exécutable.

C'est très utile notamment pour lutter contre les attaques par "débordement de buffer". Dans une telle attaque, un utilisateur malveillant envoie des données d'une taille supérieure à celle de la variable tampon destinée à les recevoir<sup>6</sup>. En effet, cette variable tampon se trouve sur la pile d'exécution, où se trouve également l'adresse à laquelle la fonction devra retourner.

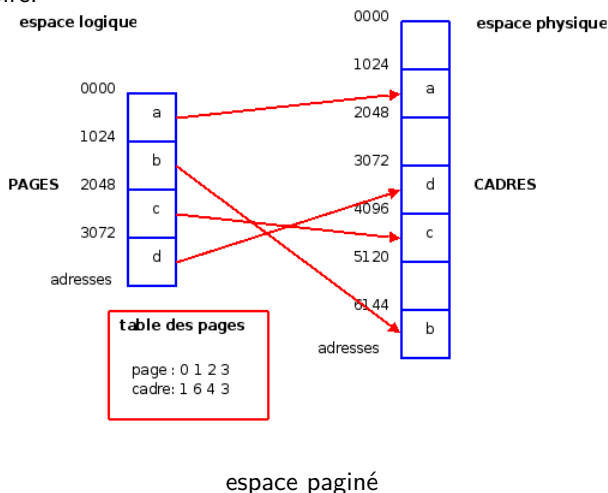
En choisissant soigneusement les données envoyées, le pirate s'arrange pour charger du code en mémoire, et remplacer l'adresse de retour de la fonction pour qu'elle désigne ce code. Quand la fonction fera son "return", c'est le "code malicieux" qui sera exécuté.

Voir article "dépassement de tampon" sur Wikipédia.

### 3.5.5 Espace mémoire paginé

Revenons au problème qui se posait pour la gestion de la mémoire, inhérent au fait de gérer des segments de tailles diverses, qui apparaissent et disparaissent au cours de l'évolution du système, en laissant des trous inexploitable.

Une solution a été trouvée, là encore très tôt dans l'histoire de l'informatique : considérer chaque segment non comme un espace linéaire, mais comme une suite de *pages*, qui ne sont pas forcément placées séquentiellement en mémoire.



À la différence des segments, les pages sont de même taille, qui est une puissance de 2 (valeurs courantes = 1, 2, 4, 16K), et elles sont alignées sur un multiple de la taille de page. En supposant des pages de 4K, la page 0 est en 0, la page 1 en 4K, la page 2 en 8K, etc.

Considérons par exemple une machine dont les adresses logiques sont sur 16 bits (espace d'adressage de 64 Kmots), avec des pages de 4K mots. La position d'un mot dans une page sera donc un nombre compris entre 0 et 4K-1, qui sera exprimé sur 12 bits (puisque  $2^{12} = 4K$ ).

Une adresse logique se composera donc

- de  $16 - 12 = 4$  bits pour le numéro de page, qui sera compris entre 0 et  $15 = 2^4 - 1$  ;
- de 12 bits pour le déplacement dans la page.

### Bits d'une adresse

Numéro de page	Position dans la page
----------------	-----------------------

format d'adresse logique

On a vu que la mémoire physique peut être de taille différente, par exemple 256K mots, et comporte donc  $\frac{256K}{4K} = 64$  cadres de page<sup>7</sup>

#### Exercice 21.

- quelle est la taille d'une adresse physique ?
- combien de bits dans un numéro de cadre de page ?

La **table des pages** indique où se trouvent les cadres de page en mémoire

page	0	1	2	3	4	5	6	7
cadre	6	7	23	1	42	17	18	62

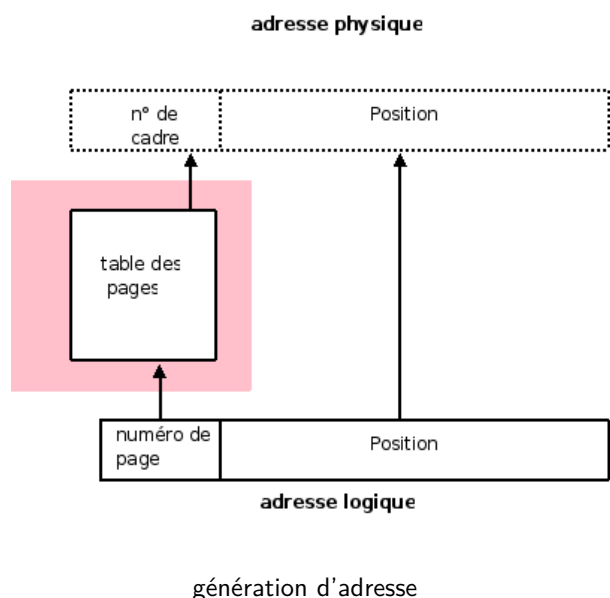
Quand un processus s'exécute, sa table des pages est chargée dans les registres de page de la MMU, qui effectue la traduction.

L'adresse physique est formée en concaténant

- le contenu du registre correspondant au numéro de page (en poids forts de l'adresse logique)
- la position dans la page (bits de poids faibles de l'adresse logique)

$$AP[17 : 12] = RP[ \quad AP[15 : 12] \quad ]$$

$$AP[11 : 0] = AL[11 : 0]$$



génération d'adresse

6. Il y a là une faute du programmeur qui n'a pas contrôlé la taille des données reçues avant de les placer dans le tampon

7. on parle de pages pour l'espace logique, et de cadres de pages pour l'espace physique



**Exercice 22.** Déterminer les adresses physiques correspondant aux adresses logiques 0x00F0, 0x32AB, 0x1030.

### 3.5.6 Un circuit MMU : le MC68851

Le MC68851, un des premiers circuits MMU pour microprocesseurs, était destiné à compléter le microprocesseur 68020 qui équipait notamment les MacIntosh II et LC, les Amiga 1200 et la console CD32, les stations graphiques IRIS et Sun-3.

Vous pouvez trouver sa “datasheet” sur internet : <http://pdf1.alldatasheet.com/datasheet-pdf/view/4166/MOTOROLA/MC68851.html>

Voici quelques-unes de ses fonctionnalités

- Full 32-bit logical to physical address ;
- Wide selection of page size from 256 Bytes to 32 KBytes ;
- fully associative, 64-entry, on chip address translation cache ;
- automatic update of the on-chip translation cache from external translation tables ;
- multiple tasks supported simultaneously ;

que nous allons examiner, et qui permettront de mieux comprendre comment fonctionne un vrai circuit MMU.

**Adresses logiques et physiques sur 32 bits :** la MMU s’intercale entre le processeur et la mémoire. Les bus d’adresses peuvent en principe être de taille différentes, ici ce sont les mêmes. À quoi cela sert-il ?

- il faut d’abord penser qu’en 1984 (année de sortie du 68020), les machines auxquelles ce processeur était destiné disposaient au mieux de quelques méga-octets de mémoire.
- cependant, rien n’empêche un programmeur d’utiliser la plage d’adresses logiques à sa guise. Sur un système paginé on peut simuler la segmentation en utilisant par exemple les 8 bits de poids forts comme numéros de segments. Cela n’impliquera pas que le segment 0xFF commence physiquement à l’adresse 0xFF000000, ni que la machine dispose de 4 G ( $2^{32}$ ) octets de mémoire réelle.
- une des rôles de la MMU est donc de “mapper” les adresses logiques utilisées (espace logique qui comporte beaucoup de trous) sur la plage d’adresses physiques qui est bien plus petite.

**Taille de pages de 256 à 32 K octets :** selon le type d’utilisation on pourra avoir intérêt à avoir des petites pages ou des grandes pages, et le circuit était configurable pour correspondre aux besoins.

- pendant l’allocation mémoire, la taille demandée est arrondi au multiple suivant de la taille des pages. Par exemple, pour 20K octets on utilisera 80 pages de 256 octets, ou 1 page de

32K. Dans ce dernier cas, on gaspille 12 Ko. En moyenne, le gaspillage est d’une demi-page.

**Exercice 23.** Un des arguments contre la pagination (opposée à la mémoire partitionnée) était que ça faisait perdre une place non négligeable en mémoire. C’est à relativiser selon la taille de la mémoire, des programmes, etc. Comparez les pourcentages de perte dans ces situations :

- une machine des années 60, avec 64Ko de mémoire (pages de 4Ko) qui fait tourner des programmes de quelques kilo-octets
- un PC actuel (pages de 4K à 2M) utilisé comme serveur
- un PC domestique

et tirez une conclusion, dans chaque contexte d’utilisation.

**Support simultané de tâches multiples :** lors d’un changement de contexte, le processeur indique à la MMU un numéro de tâche sur 3 bits. Les adresses logiques qui suivront seront alors interprétées par rapport à l’espace logique de cette tâche.

**Table associative** à 64 entrées pour la génération d’adresses, mise à jour automatiquement depuis des tables en mémoire :

- la MMU ne contient pas toute la table des pages, mais seulement 64 éléments qui contiennent chacun un numéro de page logique (combiné au numéro de tâche) et le numéro de la page physique. Le reste de la table réside dans la mémoire physique.
- lorsque le système d’exploitation effectue un changement de tâche, il indique à la MMU l’adresse de la “racine” de la table des pages de la tâche. La MMU garde une racine pour chaque tâche.
- lors d’une traduction d’adresse, le numéro de page logique (combiné avec le numéro de tâche) est comparé aux 64 numéros de pages logiques présents dans le “cache” de la MMU.<sup>8</sup> Si le numéro de page logique est absent du cache, la MMU va chercher le numéro de page dans la table en mémoire, pour remplacer le contenu du registre utilisé le moins récemment.

## 3.6 Mémoire virtuelle paginée

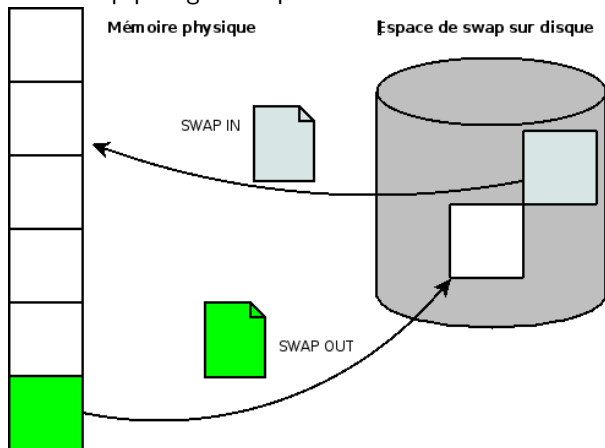
### 3.6.1 Principe

La pagination permet d’apporter une grosse amélioration de la technique de va-et-vient sur disque exposée plus haut. En effet, pour relancer un programme qui a été “swappé” sur le disque, il suffit de recharger depuis le disque la page qu’il

8. C’est une comparaison **simultanée** grâce à 64 circuits comparateurs qui travaillent en parallèle, un par registre. C’est ce qu’on appelle une mémoire associative

était en train d'exécuter, et non l'intégralité de son espace mémoire.

L'espace d'échange sur disque, qui est lui-même structuré en "pages" de même taille. L'ensemble, mémoire vive + espace d'échange, constitue une mémoire virtuelle qui peut être beaucoup plus grande que la seule mémoire vive.



Swapping

On constate par ailleurs que, pendant l'exécution d'un programme, celui-ci ne fait référence qu'à quelques pages à la fois. Il est assez probable qu'après avoir exécuté l'instruction N, il passe à l'instruction N+1, où à une autre instruction proche : en pratique la plupart des boucles et de sauts ne vont pas très loin, le plus souvent la destination est dans la même page.

Automatiquement, les pages qui ont été transférées sur disque y restent tant qu'on n'en a pas besoin, et libèrent autant de place pour les pages les plus "actives" des processus, on fera donc une économie de place, qui permettra d'avoir davantage de processus "présents" (processus prêts, avec leur page active déjà présente en mémoire vive). D'où de meilleures performances qui si on chargeait en totalité l'espace mémoire d'un processus.

### 3.6.2 MMU pour la mémoire virtuelle paginée

Une page peut être présente en mémoire ou non, et si elle ne l'est pas, elle peut être sur le disque. La table des pages indique donc le numéro de cadre de page, et le numéro de bloc dans l'espace disque réservé au swap.

La table des pages pour la mémoire paginée comporte donc ces informations

page logique	0	1	2	.....
cadre (physique)	17	-	3	....
sur disque	42	13	44	...

La MMU contiendra, comme précédemment, une table de registres pour la correspondance entre numéro de page et numéro de cadre, et des indicateurs de présence pour les pages. Une interruption de "défaut de page"<sup>9</sup> sera émise si un accès est tenté à une page absente.

9. qui signifie que la page fait défaut (elle manque), et non pas qu'elle est défectueuse

10. ce qui est très éloigné des ordres de grandeurs réels : un mini-ordinateur typique des années 80, avec quelques mega-octets de mémoire, a des centaines ou des milliers de cadres de page.

Exemple :

```

DEFAULT_DE_PAGE = non PRESENT[ AP[15 :12]
]
AP[17 :12] = RP[ AP[15 :12] ]
AP[11 : 0] = AL[11 :0]

```

L'interruption provoquera l'intervention du système d'exploitation qui

1. consultera la table des pages pour trouver l'emplacement, sur disque, de la page manquante
2. trouvera un cadre de page libre en mémoire.
3. procédera au chargement de la page en mémoire
4. mettra à jour la table des pages
5. relancera le processus interrompu

Pour trouver un cadre de page, il faudra souvent (dès que la mémoire réelle sera pleine) faire de la place en transférant sur disque une page présente. La manoeuvre de remplacement comportera donc deux accès disque, l'un pour sauver sur disque la page qui est en mémoire, l'autre pour lire sur disque la page à ramener en mémoire.

Deux accès disques représentent dans le meilleur des cas quelques centièmes de seconde, ce qui est très long à l'échelle de l'ordinateur. On cherchera donc à minimiser le nombre de défauts de page.

### 3.6.3 Les algorithmes de remplacement de page

Dans le cadre de ce mécanisme très général il reste à définir la méthode pour choisir (second point) le cadre de page dans lequel la page manquante va être amenée en mémoire. C'est l'objet des **algorithmes de remplacement de page**.

En réalité ce sont plus des heuristiques que des algorithmes : un bon algorithme devrait trouver une solution optimale, or c'est impossible dans l'absolu : à un moment donné on décide de remplacer telle page plutôt qu'une autre, c'est seulement dans le futur, selon la suite des événements, que cette décision apparaîtra judicieuse ou non.

Or le futur est, en grande partie, imprévisible. On utilisera donc des **heuristiques**, méthodes dont l'objectif est d'obtenir des résultats acceptables dans la plupart des cas.

### 3.6.4 Remplacement à tour de rôle

Le premier algorithme, appelé aussi FIFO (first-in first out, premier entré premier sorti) consiste à utiliser à tour de rôle chacun des cadres de pages. C'est une méthode simpliste, qui ne donne pas de bons résultats (nous verrons pourquoi), que nous exposons surtout pour introduire quelques idées.

L'exécution d'un programme provoque de nombreux accès successifs à la mémoire. Ces adresses correspondent à des pages, dont nous retiendrons les numéros, qui constituent la

séquence de références de pages. Par exemple un processus fera accès aux pages 22,33,22,17,44,18,22,12,33,22,...

Pour simplifier l'explication, prenons un mémoire physique de 4 cadres de pages seulement.<sup>10</sup>

Un tableau nous servira à montrer quelle page occupe quel cadre, au fur et à mesure du déroulement de la séquence d'accès :

Après quelques étapes, le tableau est ainsi rempli

temps page	1	2	3	4	5	6	7	8	9	10
cadre 1	22	22	22	22	22	18	18			
cadre 2	-	33	33	33	33	33	22			
cadre 3	-	-	-	17	17	17	17			
cadre 4	-	-	-	-	44	44	44			

- au début tous les cadres de pages sont vides
- aux temps 1,2,4 et 5 les cadres de pages sont chargés successivement avec les pages demandés (les défauts de page sont encadrés).
- remarquez qu'au temps 3 la page demandée est déjà présente, il n'y a donc pas d'accès au disque.
- à t=6 tous les cadres de pages sont occupés. L'algorithme choisit de charger la page manquante dans le cadre 1.
- même problème ensuite : on passe au cadre 2

#### Exercice 24.

- Complétez le tableau.
- Combien de défauts de page ?

**Exercice 25.** On appelle **oracle** un algorithme (hypothétique) qui serait capable de prendre les meilleures décisions en tenant compte de l'avenir. Que ferait un oracle dans ce cas ?

**Anomalie de Belady** . Il est raisonnable de penser que plus on dispose de mémoire, moins on aura de défauts de page.

Dans les années 60 on pensait que c'était une règle générale, mais en 1969 Laszlo Belady a trouvé un contre-exemple<sup>11</sup> :

**Exercice 26.** Soit la séquence de références 1 2 3 4 1 2 5 1 2 3 4 5

- calculez le nombre de défauts de page obtenus avec l'algorithme FIFO pour 3 cadres de pages
- même question pour 4 cadres de pages

C'est un exemple de "scénario pathologique" qui montre qu'une affirmation généralement acceptée ("il vaut mieux être riche et en bonne santé que pauvre et malade") n'est pas vraie dans tous les cas. Cependant l'existence de contre-exemples n'empêche pas l'affirmation d'être vraie presque tout le temps.

**Note.** On a longtemps estimé que les séquences pathologiques produisaient au maximum deux fois plus de défauts de page. Deux chercheurs ont montré en 2010 (<http://arxiv.org/abs/1003.1336>) que le ratio des nombres de défauts de page n'est pas borné.

### 3.6.5 Algorithme LRU : least recently used

L'algorithme FIFO ci dessus ne donne pas de bons résultats en pratique, parce qu'il ne tient pas compte d'une propriété importante du comportement des programmes, la **localité**.

**Principe de localité** : pendant le déroulement d'un vrai programme, il est très probable qu'après avoir exécuté une instruction, la suivante soit très proche, presque certainement dans la même page. Et qu'après avoir accédé à un élément de tableau, on passe au suivant.

Cette propriété de localité fait que si un programme a besoin d'accéder à une page, c'est très probablement une des pages qu'il a utilisées il y a peu de temps.

On met à profit ce principe, en conservant les pages qui ont été utilisées le plus récemment. L'algorithme qui en découle consiste donc à choisir le cadre qui contient **la page accédée le moins récemment**, d'où son nom (LRU = least recently used).

Le tableau ci-dessous montre le déroulement de l'algorithme LRU.

temps page	1	2	3	4	5	6	7	8	9	10
cadre 1	22	22	22	22	22	22	22			
cadre 2	-	33	33	33	33	18	18			
cadre 3	-	-	-	17	17	17	17			
cadre 4	-	-	-	-	44	44	44			

- Les caractères gras marquent les accès.
- à t=6, on choisit le cadre 2 pour lequel le dernier accès a eu lieu à t=2, contre 3,4 et 5 pour les autres.

#### Exercice 27.

- Complétez le tableau.
- Combien y a t'il de défauts de page ?

**Réalisation matérielle** Pour pouvoir appliquer cet algorithme il faut ajouter un peu de matériel dans la MMU : pour chaque page un *estampille*, registre qui contiendra la date de dernier accès (*horodatage*). Cette estampille sera mise à jour à chaque accès en y copiant la valeur d'un compteur qui s'incrémente sous le contrôle d'une horloge.

Le système d'exploitation interrogera la MMU pour relever les compteurs quand un défaut de page se produira.

11. probablement en essayant de démontrer que c'était vrai

**Exercice 28.** En théorie ce compteur, mis à zéro au démarrage, devrait être incrémenté à chaque instruction. De quelle taille devrait-il être pour une machine qui exécute 1 million d'instructions par seconde, et qui fonctionne sans arrêt pendant une année ?

En pratique, les transistors coûtent cher sur une MMU, et on se contentera d'une précision moindre, en ne stockant que les bits de poids fort du compteur. Nous verrons plus loin l'algorithme NRU (*not recently used*), dans lequel on se contente d'un bit pour différencier les pages récemment accédées de celles qui ne le sont pas.

**Scénarios pathologiques** : en général les algorithmes basés sur la localité marchent bien en pratique, parce que c'est un comportement qu'on observe sur les vrais programmes. Cependant il est tout-à-fait possible de construire des contre-exemples.

**Exercice 29.** Soit l'affirmation "le LRU fait moins de défauts de page que que FIFO". Considérons un système avec 3 cadres de pages. La séquence de références de pages commence par 11, 22, 33. Si on la prolonge par 11 et 44, l'algorithme FIFO placera la page 44 dans le cadre 1, et LRU dans le cadre 2.

- À partir de là, les situations ayant divergé, continuez la séquence avec des numéros bien choisis pour que la stratégie LRU conduise à plus de défauts de page que FIFO.
- Votre conclusion ?

Attention : l'existence de contre-exemples ne prouve pas que LRU n'est pas meilleur que FIFO, seulement qu'il peut exister des cas pathologiques où ça se passe moins bien que ce qu'on pensait. Or ces cas, précisément, sont construits en ignorant le principe de localité, et en raisonnant sur un nombre totalement irréaliste de cadres de pages<sup>12</sup>. On est donc loin de la "réalité statistique".

### 3.6.6 NRU, remplacement d'une page non récemment utilisée

Deux bits sont associés à chaque page : R indique si la page a été *référéncée* (en lecture ou en écriture), M qu'elle a été modifiée. Ces deux bits sont positionnés à chaque référence mémoire.

Périodiquement, le système remet les bits R des pages à 0. Si le bit R d'une page est à 1, on peut donc en conclure que la page a été référencée récemment.

On obtient donc 4 catégories de pages :

1. pages non référencées, non modifiées (R=0, M=0),
2. pages non référencées, modifiées (R=0, M=1),
3. pages référencées, non modifiées (R=1, M=0),
4. pages référencées et modifiées (R=1, M=1).

<sup>12</sup>. Un petit programme pour PC occupe quelques mégas octets, soit des milliers de pages. La probabilité d'apparition d'un scénario pathologique est assez faible.

L'algorithme NRU (*not recently used*) choisit au hasard une des pages de la première catégorie non vide.

En pratique cet algorithme est facile à implémenter, relativement efficace, et ses performances sont généralement suffisantes.

### 3.6.7 Algorithme de la seconde chance

Variante de l'algorithme FIFO, en utilisant le bit R. Lors d'un défaut de page, l'algorithme teste le bit R de la page la plus ancienne. Si il est à 1, la page est remise en fin de liste avec son bit R mis à 0 (comme si elle venait d'être chargée), et l'algorithme regarde la page suivante, jusqu'à trouver une page ayant le bit R à 0, qui sera retirée.

En pratique on utilise une liste circulaire. Exercice : justifier son autre appellation : *algorithme de l'horloge*.

### 3.6.8 Algorithme du vieillissement

On utilise le bit de référence R, et un registre à décalage vers la droite de n bits pour chaque page. À intervalles réguliers, le bit R est transféré dans le registre à décalage, puis remis à 0.

Le registre à décalage permet donc de savoir si des références ont été faites lors des n dernières périodes de temps. Par approximation grossière, un grand nombre dans le registre (beaucoup de bits 1 à gauche) indique qu'une page a été récemment beaucoup utilisée.

### 3.6.9 Réserveation d'espace d'échange

On peut se poser la question : où vont les pages évacuées ? Leur a-t-on préalablement réservé un espace sur le disque, ou non ?

La réponse dépend de la philosophie adoptée pour la mémoire virtuelle. L'ancienne école considérait que les processus "habitaient" dans l'espace disque réservé pour le swapping, et qu'ils ne faisaient que passer provisoirement dans la mémoire vive, considérée comme un *cache* du disque dur. Dès son lancement, un processus se voyait donc réserver une place dans l'espace de swap, qui devait être au moins aussi grand que la mémoire vive.

On peut aussi considérer que l'espace de swap est une extension de la mémoire. Un processus est chargé en mémoire sans se préoccuper de réserver de l'espace sur disque. Il pourrait donc théoriquement arriver qu'on ne puisse pas évacuer une page de la mémoire vers le disque, si le swap est plein.

**La taille du swap** Lorsqu'ils installent une nouvelle machine, les administrateurs débutants se posent inmanquablement cette question : quelle taille donner à l'espace d'échange (swap) ?

À cette question il existe une seule réponse techniquement exacte : il faut donner un espace suffisant pour qu'à tout moment, tous les processus rentrent. Malheureusement, on ne

peut pas deviner comment la machine sera utilisée dans l'avenir, et on doit donc se baser sur des observations (comment la machine actuelle est-elle utilisée ? Quels programmes ? Combien d'utilisateurs ? etc.) pour faire des prévisions.

Mais comme le disait le regretté Pierre Dac, *la prévision est un art difficile surtout en ce qui concerne l'avenir*. Les administrateurs s'en tiennent donc souvent à une estimation grossière (*rule of thumb*) qui consiste à prendre un swap deux fois plus grand que la mémoire réelle. **Cette règle n'a aucune justification technique**. Par contre elle est extrêmement utile : elle évite de perdre du temps à des supputations oiseuses sur l'usage futur de la machine. Si le swap s'avère trop petit à l'usage, il sera toujours possible de l'agrandir ultérieurement (quitte à acheter un disque supplémentaire). Si il est trop grand, quelques centaines de mégaoctets seront gaspillés, ce qui n'est pas un drame. De nos jours, l'espace disque ne coûte pas très cher.

Dans des temps plus anciens (années 70-80), le matériel était beaucoup plus cher, et les acheteurs étaient beaucoup plus attentifs à la constitution de configurations équilibrées, savant dosage entre un processeur d'une puissance suffisante, d'une mémoire assez spacieuse, et de disques assez gros, le tout devant tenir dans un budget limité.

Le raisonnement était le suivant : choisir une mémoire assez grande pour avoir un taux de pagination quasi nul sous une charge normale. Ceci implique déjà d'avoir un swap plus grand que la mémoire (voir plus haut). En doublant cette taille, on permet théoriquement de lancer deux fois plus de processus, en charge de pointe, mais le taux d'échange disque/mémoire sera beaucoup plus élevé, au point de paralyser la machine (*thrashing*, facile à reconnaître au bruit du disque qui glougloute constamment) si tous les processus prêts sont en défaut de page, et le taux d'occupation du processeur baisse énormément (une fois atteint le sommet de 100 %, il ne peut que baisser) : c'est le disque qui devient le goulet d'étranglement. Si on en arrive là fréquemment, c'est qu'on a mal estimé la charge : il vaut mieux se contenter d'un processeur moins puissant et utiliser la différence de prix pour gonfler la mémoire.

### 3.6.10 Intérêt du swap ?

De nos jours, le prix de la mémoire est tellement faible qu'on pourrait envisager, dans beaucoup de situations, de se passer de swap.

Mais sur un ordinateur personnel, l'espace de swap peut aussi servir à sauver le contenu de la mémoire quand l'appareil est mis en veille, ce qui permettra de redémarrer l'appareil beaucoup plus vite que passer par la longue procédure de boot.

Une autre utilisation des mécanismes du swap (table de pages) est de pouvoir traiter des fichiers comme si c'étaient des tableaux en mémoire. Voir manuel de la fonction Unix `mmap()`. La table des pages est alors modifiée pour qu'une partie de l'espace logique du processus corresponde à des blocs situés, non dans l'espace d'échange, mais dans un fichier.

Dans la même veine, des processus peuvent demander à

partager de la mémoire commune (appel `shmget()`) : leurs tables de pages désignent alors des blocs communs.

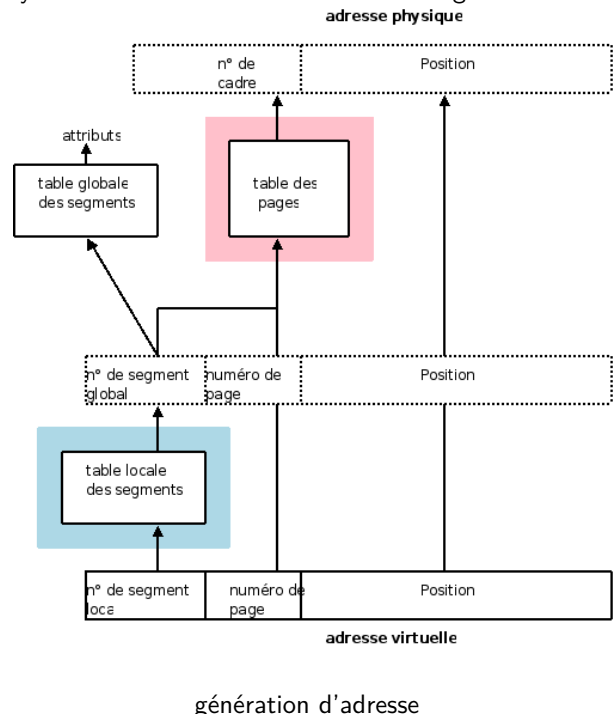
## 3.7 Mémoire virtuelle segmentée-paginée

La mémoire virtuelle segmentée-paginée est une combinaison des techniques précédentes :

- l'espace logique d'un processus est composé de segments
- chaque segment est découpé en pages
- les pages peuvent être présentes en mémoire, ou dans l'espace d'échange sur disque.

Elle est utilisée couramment depuis le GE-645 (1968) et l'IBM 360/67 (1969)

La figure ci-dessous montre la génération d'adresse d'un tel système utilisant une table locale des segments :



**Exercice 30.** Sur le schéma ci-dessus l'adresse virtuelle (en hexadécimal) `0x2A0073CB` correspond au segment 42 ; le système a des pages de 4K mots.

- quelle est la taille maximum d'un segment ?
- Combien a-t-il de pages ?

Soit l'adresse réelle `0xA20F3CB`

- que dire sur la taille de la mémoire réelle ?
- quel est le numéro de cadre de page ?
- imaginez le contenu des différentes tables pour que l'adresse physique soit le résultat de la génération d'adresse à partir de l'adresse logique ci-dessus.

Le GE-645 de General Electric est l'ordinateur sur lequel a été développé le système Multics (Lire le

manuel : [http://www.bitsavers.org/pdf/ge/GE-645/GE-645\\_SystemMan\\_Jan68.pdf](http://www.bitsavers.org/pdf/ge/GE-645/GE-645_SystemMan_Jan68.pdf)).

- les adresses absolues sont sur 24 bits, permettant d'utiliser jusqu'à 16 millions de mots. La documentation précise «*Memories of this size are not available for initial GE-645 systems. The 16-million-word addressing capability exists to facilitate future growth of the system.* » (p. 10)
- les adresses logiques sont sur 34 bits (le GE-645 est une machine mots de 36 bits) :
  - 18 bits pour le numéro de segment
  - 6 pour le numéro de page
  - 10 pour le déplacement dans la page

#### Exercice 31.

- Quelle est la taille maximale d'une page ? D'un segment ?
- la taille de l'espace logique d'un processus ?

**Exercice 32.** La page Wikipedia consacré à l'IBM 360-67 précise

Dynamic Address Translation (DAT) with support for 24 or 32-bit virtual addresses using segment and page tables (up to 16 segments each containing up to 256 4096 byte pages)

Interprétez cette phrase.



IBM 360/67 (1969)

**Exercice 33.** Les processeurs ARM équipent la grande majorité des smartphones et autres téléphones portables.

- Trouvez un modèle de processeur qui soit équipé d'une MMU, et sa documentation sur Internet.
- Quelle est la taille des adresses physiques ? Des adresses logiques ?
- Quelles tailles de pages sont disponibles ?
- Expliquez un des schémas décrivant la génération d'adresses.

Attention, la terminologie change selon les constructeurs. Établir la correspondance avec les termes du cours.

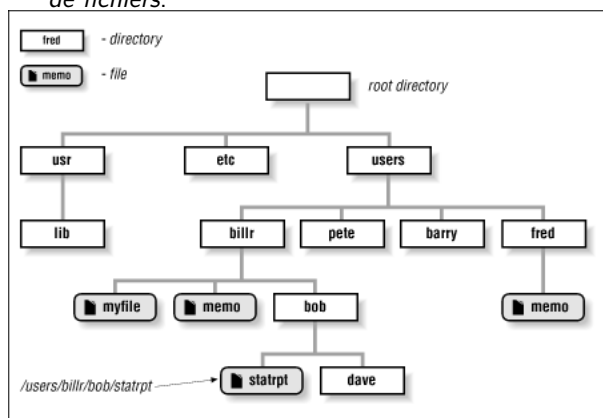
# Chapitre 4

## Gestion des fichiers

### 4.1 Les fichiers

Qu'est-ce qu'un fichier ?

- Un fichier est une suite de données structurées codées sur un support.
- Les fichiers sont la plupart du temps conservés sur des *mémoires de masse* tels que les disques durs.
- Les fichiers sont classés dans des *répertoires*, chaque répertoire peut contenir d'autres répertoires, formant ainsi une *organisation arborescente* appelée *système de fichiers*.



un système de fichiers

Autrefois les systèmes d'exploitation prenaient en charge différents types de fichiers :

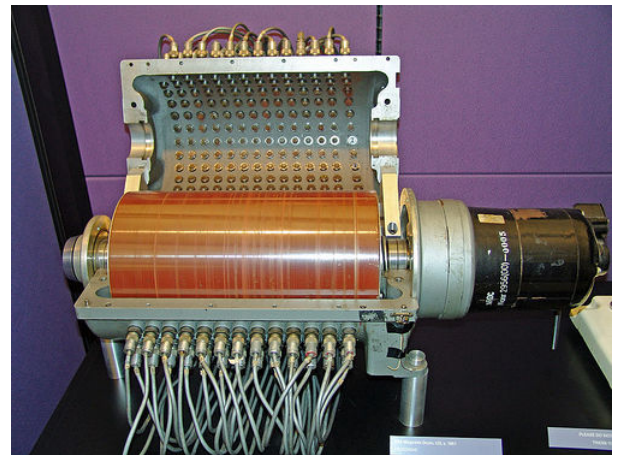
- les fichiers de texte, composés de lignes de taille variable ;
- les fichiers séquentiels, composés d'enregistrements
- les fichiers relatifs, contenant des enregistrements accessible directement par leur numéros (le système d'exploitation fournit des fonctions d'accès du type : "lire l'enregistrement numéro 25")
- les fichiers indexés, dont les enregistrements comportent une *clé* (par exemple la plaque d'immatriculation pour un fichier de cartes grises). On peut alors demander à lire l'enregistrement concernant la plaque "1234AB56".
- ...

De nos jours, la plupart des systèmes connaissent un seul type : les fichiers sont une séquence d'octets, avec des opérations (`read`, `write`) pour lire et écrire à partir d'une position

courante, et se positionner (`seek`) à un endroit déterminé (numéro d'octet). Les fonctionnalités des fichiers indexés, par exemple, sont fournies par des bibliothèques spécialisées (voir par exemple la page de manuel `dbopen`).

### 4.2 Les disques

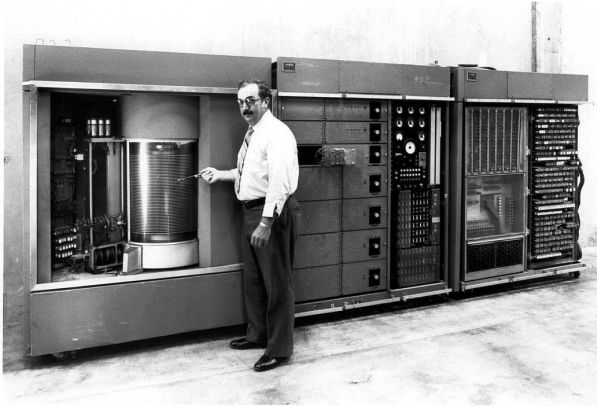
Les disques magnétiques que nous connaissons dérivent des mémoires à tambours magnétiques, des cylindres en rotation couverts d'un enduit ferro-magnétique, avec des têtes magnétiques qui assuraient la lecture et l'écriture.



ancêtre : le tambour magnétique

En 1953, un ingénieur d'IBM a eu l'idée de superposer des plateaux sur un axe, et d'ajouter une tête de lecture-écriture mobile, portée par un bras. Le bras pouvait changer de disque en moins d'une seconde.





Prototype d'IBM

La production commerciale du RAMAC 305 (Random Access Method of Accounting and Control) a commencé en juin 1957.

Caractéristiques :

- 59 disques de 24 pouces
- 2 têtes pouvant se déplacer d'un plateau à l'autre
- capacité totale de 5 megaoctets.
- prix de 10000 dollars par megaoctet.

De nombreuses améliorations ont été apportées :

- l'utilisation de plusieurs têtes de lectures portées par un même bras (une par face de disque) dispense de déplacer la tête d'un disque à l'autre.
- un coussin d'air permet de "faire flotter" les têtes en évitant qu'elles rentrent en contact avec le disque, incident appelé "atterrissage" qui endommage irrémédiablement la surface magnétique.

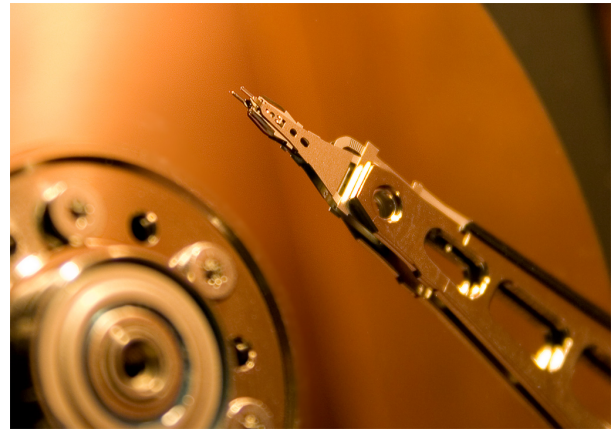
L'augmentation de la précision mécanique du mouvement des têtes et de la vitesse de rotation du disque (entre 3600 et 15000 tours/mn) a permis d'augmenter la capacité : 5 Mo en 1956, 1 Go en 1982, 25 Go en 1998, 500 Go en 2005, 2 To en 2009, 4 To en 2011.



Une unité de disque récente (80GB), début du siècle

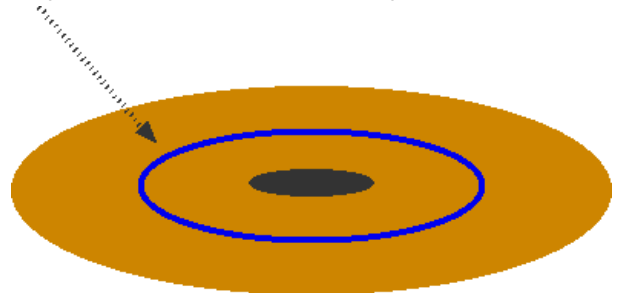
De nos jours, on trouve des disques de capacité 4 To en format 3 pouces et demi (Hitachi 7K4000).

#### 4.2.1 Disque, tête



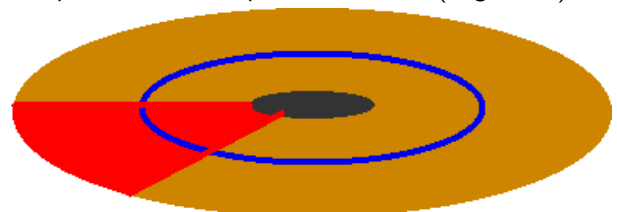
Disque, tête de lecture, bras

La position d'une tête définit une *piste*



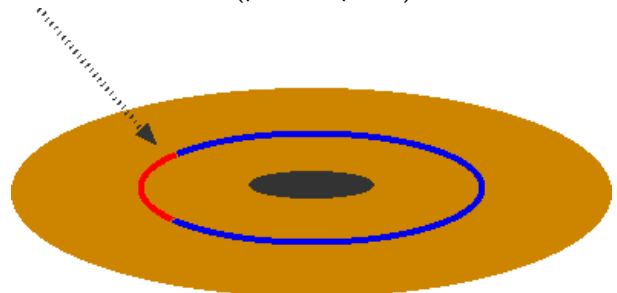
Une piste

Les pistes sont découpées en *secteurs* (angulaires)



Un secteur

Un secteur contient (pour simplifier) un *bloc de données*.



Un bloc

Changer de piste implique un *mouvement* de la tête, changer de secteur une rotation.

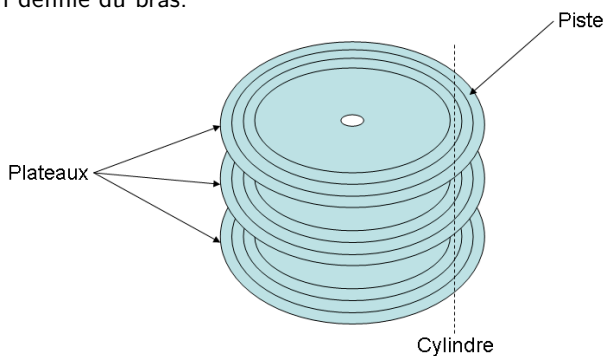


#### 4.2.2 Avec plusieurs plateaux



Un axe, un bras, plusieurs plateaux

On appelle *cylindre* l'ensemble des pistes pour une position définie du bras.



Cylindre = pistes  
pour une position du bras

- Accéder à un autre cylindre implique un mouvement du bras, qui est lent (à l'échelle de l'ordinateur)
- Changer de tête se fait par commutation électronique d'une tête à l'autre.

À retenir : le disque magnétique est un médium à *temps d'accès non uniforme* : le délai pour accéder à une information dépend de la position de cette information, et de la position courant des têtes de lecture.

Dans ce délai il faut intégrer

- le déplacement du bras (sélection du cylindre)
- la rotation (sélection du bloc)

Le temps de commutation (sélection de la piste) est négligeable.

Le délai le plus long vient du mouvement de la tête. Il en résulte un principe de localité : pour minimiser les déplacements de tête, et donc les temps d'accès, il vaut mieux grouper les données sur des cylindres proches.

#### 4.2.3 Quelques chiffres

couche magnétique	10-20 nm
vitesse de rotation	5400-15000 tours/min
capacité	120 GB -2 TB
temps d'accès	2-15 ms
vitesse de transfert	70 MB/sec (7200 t/min)

### 4.3 Gestion des entrées/sorties sur disque

#### 4.3.1 Sur un système mono-tâche

Sur un système mono-tâche : il n'y a pas (trop) de problème, on déplace la tête à la demande là où on en a besoin.

Bien évidemment les performances sont meilleures si les blocs d'un fichier sont regroupés. Il est donc prudent, parfois, de *défragmenter* les systèmes de fichier pour regrouper les blocs.

Autrefois on profitait de la sauvegarde pour copier le disque original, fichier par fichier, sur un disque vierge, ce qui regroupait les blocs. Le disque original était alors démonté et conservé comme sauvegarde.

Bien entendu, une telle sauvegarde se faisait dans le cadre d'une maintenance planifiée : pendant quelques heures l'ordinateur était inaccessible aux utilisateurs.

#### 4.3.2 Sur un système multitâche

Sur un système multitâches il est possible que plusieurs processus soient bloqués pour avoir demandé une opération sur le même disque. Il y a donc plusieurs demandes en attente : laquelle servir en premier ? C'est le problème d'*ordonnancement des mouvements du bras du disque*.

#### 4.3.3 E/S : Premier arrivé, premier servi

Une première politique peut être facilement imaginée : exécuter les demandes dans l'ordre où elles ont été effectuées.

Exemple : trois processus qui lisent et traitent des données situées dans des blocs consécutifs : P1 commence au bloc 100, puis 101, 102, etc. Idem pour P2 et P3 qui commencent respectivement à 200 et 500.

Déroulement :

- P1 demande la lecture du bloc 100, qui commence à s'exécuter ; P2 demande le bloc 200, P3 le bloc 400
- la lecture se termine, la lecture du bloc 200 s'exécute, P1 demande le bloc 101.
- la lecture se termine, la lecture du bloc 500 s'exécute, P2 demande le bloc 201.
- la lecture du bloc 101 commence, P3 demande le bloc 501, ...
- etc.

Successivement, on lit donc les blocs 100, 200, 500, 101, 201, 501, 102, ...

Cette politique "premier arrivé - premier servi" est

- *simple* à mettre en oeuvre
- *équitable* ...
- mais *pas efficace*

#### 4.3.4 E/S : plus court déplacement

Cette politique consiste à exécuter la requête qui demandera le moindre déplacement.

Sur le même exemple, après avoir lu les blocs 100 et 200 on a en attente des demandes pour 101 et 500. La position 101 est la plus proche de la position courante (200), on retourne donc à P1.

Il en résulte l'ordre suivant : 100, 200, 101, 201, 102, 202 ... et on constate que les lectures de P3 ne sont pas faites en raison de la coalition entre P1 et P2.

En résumé, la politique "déplacement le plus court" est

- simple à mettre en oeuvre
- efficace ...
- mais pas équitable

#### 4.3.5 Politique de l'ascenseur

La politique dite "de l'ascenseur" est un compromis intéressant.

L'algorithme de gestion est à deux phases

- dans la phase montante, on traite la requête qui concerne le plus petit de bloc strictement supérieur à la position courante. Si il n'y en a pas, on passe en phase descendante.
- dans la phase descendante, on traite la requête pour le plus grand numéro de bloc strictement inférieur à la position courante. Si il n'y en a pas on passe en phase montante.

Déroulement :

- (↑) 100, 200, 500,
- (↓) 201, 101,
- (↑) 202, 501,
- (↓) 203, 102,
- (↑) 204, 502
- (↓) ...

La politique de l'"ascenseur" est

- assez équitable ...
- assez efficace

#### 4.3.6 E/S : Deadline driven disk scheduler

Une idée plus moderne : on fixe un délai maximum, après lequel l'exécution d'une requête devient urgente.

Algorithme :

- si il y existe des requêtes urgentes, on traite la plus ancienne (qui est aussi la plus urgente)
- si il n'y en a pas, on traite la requête qui demande le moins de mouvement.

La politique "deadline driven scheduling" est

- efficace
- et équitable
- ... et brevetée

**Exercice 34.** Traduire en français correct le titre du brevet *United States Patent 5787482* de 1998 :

Subject: Deadline driven disk scheduler method and apparatus with thresholded most urgent request queue scan window

#### 4.3.7 Conclusion

- Une problématique spécifique, par rapport à la gestion des données en mémoire, est de tenir compte du temps d'accès aux données, qui n'est pas uniforme.
- Il y a un compromis à trouver entre *efficacité* et *équité*
- quelques heuristiques simples pour l'ordonnancement :
  1. premier arrivé, premier servi
  2. plus court déplacement
  3. ascenseur
  4. *deadline scheduling*

### 4.4 Systèmes RAID

#### 4.4.1 Principe et objectifs

Les systèmes RAID (*Redundant array of inexpensive disks*) sont composés de plusieurs disques groupés dans un boîtier ou un *rack*<sup>1</sup>.



un boîtier RAID



1. Boîtier que l'on boulonne dans une armoire informatique de taille normalisée

## un rack RAID

L'objectif initial était de réaliser des systèmes de stockage de grande capacité en combinant des "petits" disques bon marchés, plutôt qu'en achetant de gros disques, qui n'étaient disponible que sur marché du matériel professionnel, et donc beaucoup plus coûteux.

On verra que grouper des disques permet non seulement d'étendre la *capacité*, mais aussi d'améliorer la *fiabilité* (par l'introduction d'une redondance) et les *performances*, en faisant travailler les disques en parallèle.

Actuellement on traduit RAID par Redundant Array of Independent Disks, les systèmes n'étant pas forcément composés de disques bon marché.

### Connexion des disques

- les disques peuvent être connectés de la manière habituelle, et le RAID géré par des modules spécifiques du système d'exploitation
- les disques peuvent être raccordés à une carte d'extension qui prend en charge les fonctions du RAID
- la carte mère peut intégrer la fonction RAID,
- les disques peuvent également être intégrés à un boîtier (ou rack) extérieur, qui se comporte comme un disque unique.

Il existe un grand nombre de manières de combiner plusieurs disques physiques pour réaliser un système de stockage.

### 4.4.2 NRAID (JBOD)

Le degré zéro du RAID consiste à "concaténer" de petits disques pour en faire un gros. Par exemple, avec 4 disques de 500 Go on obtient un système de 2 To. Les premiers secteurs du système sont sur le disque 1, les suivants sur le disque 2, etc.

Ceci permet de répartir les données, mais n'assure aucune redondance, d'où les noms usuels NRAID et JBOD

- NRAID *is Not RAID*
- JBOD *just a bunch of disks*

**Exercice 35.** Les disques sont-ils nécessairement de même taille ?

On notera qu'il est possible de mener plusieurs requêtes d'entrées-sorties simultanément sur le système, si par chance elles concernent des disques différents. Ceci a un impact positif sur les performances.<sup>2</sup>

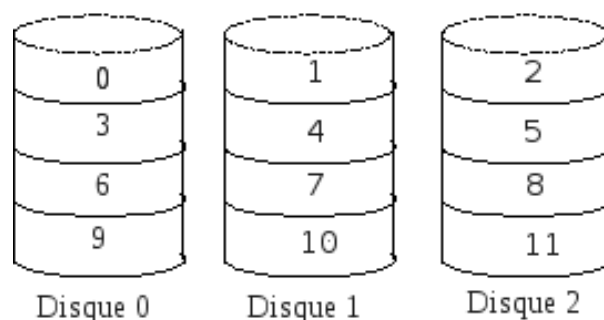
En cas de défaillance d'un disque, les données sont compromises. Les fichiers qui ont des blocs sur le disque défaillant sont irrécupérables.

### 4.4.3 RAID 0 : agrégation par bandes

Le RAID 0, ou stripping, combine des disques de même taille. A la différence du NRAID, les données sont réparties par *bandes* : le premier bloc du système est sur le premier disque, le second sur le second disque etc.

2. par rapport à l'utilisation d'un disque unique de 2 To

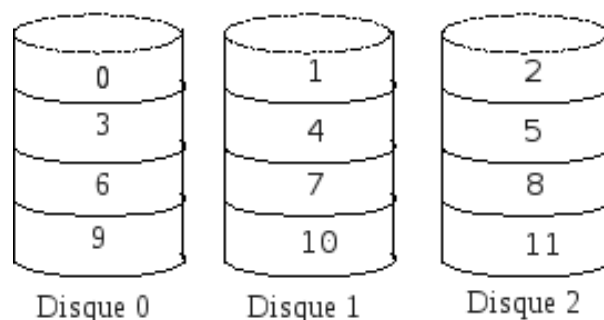
### Striping



RAID 0 : agrégation par bandes

Ceci permet une meilleure répartition de la charge de travail, en tenant compte également du principe de localité : après avoir lu un bloc on a très probablement besoin du suivant ; il est donc possible, en parallèle, de le lire de façon anticipée.

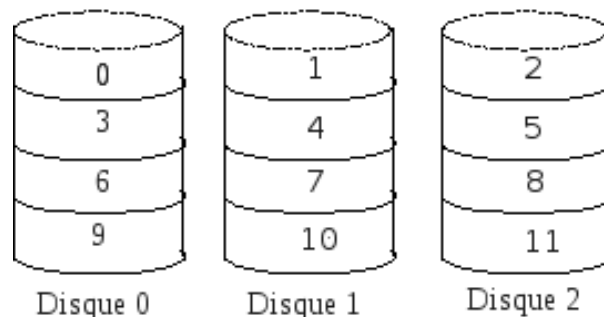
### Striping



Demande de lecture 7 :  
possibilité de *lecture anticipée* de 8 et 9  
dans le même temps

On ne peut pas anticiper les écritures, mais plusieurs écritures en même temps, à condition qu'elles se déroulent sur des disques différents.

### Striping



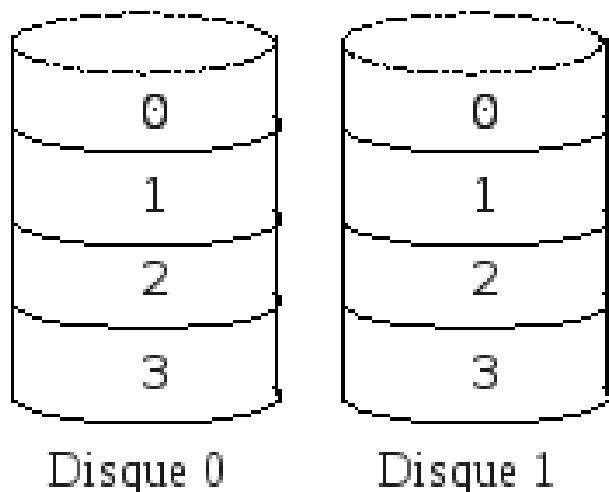
**exemple**  
lecture/écriture de 2, 6 et 11

En cas de défaillance d'un disque les données sont irrécupérables.

#### 4.4.4 RAID 1 : miroir

Dans une configuration en miroir, les données sont stockées en double (ou plus).

### Miroir



RAID 1 : miroir

Avec 2 disques de 500 Go chacun, on fait donc un système RAID 1 de 500 Go.

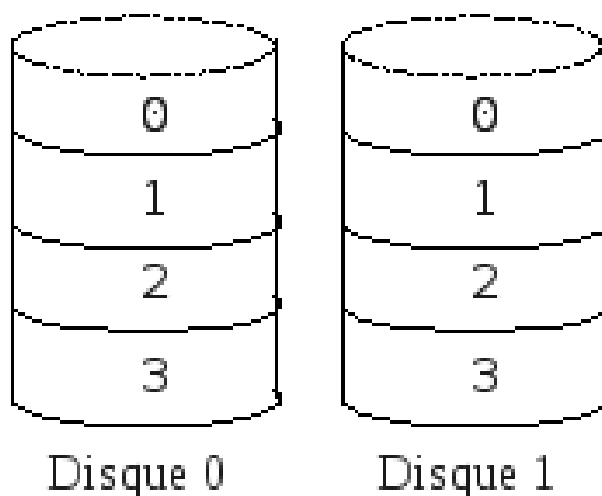
L'objectif est bien sûr d'améliorer la redondance : si un des disques tombe en panne, on peut retrouver les données sur l'autre disque.

**Exercice 36.** Si les disques ont une probabilité de panne de 5/100 par an, quelle est la probabilité de perdre les données dans un système RAID de 2 disques ?

**Exercice 37.** Peut-on faire du RAID 1 avec 3 disques ?

Toute opération d'écriture sur le système se traduit obligatoirement par des écritures sur tous les disques, menées simultanément, ce qui ne prend pas plus de temps qu'une seule écriture.

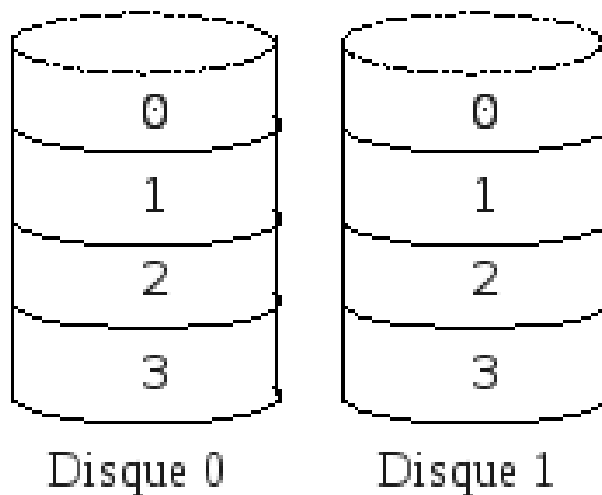
### Miroir



*Écriture* : doit se faire en parallèle

Par contre, puisque les données sont en double, on peut lire des blocs à 2 adresses différentes en mêmes temps.

### Miroir



*Lecture* : deux à la fois

Globalement, on peut donc mener deux fois plus de lectures qu'avec un seul disque : il y a donc un gain de performances en lecture.

#### 4.4.5 Disques de rechange, reconstruction

Un boîtier RAID possède en général des emplacements supplémentaires pour des disques de rechange (*spare disks*).

Par exemple un boîtier en RAID 1 à deux disques peut avoir 3 emplacements. Normalement 2 disques (D1, D2) sont actifs et fonctionnent en miroir, le dernier D3 est arrêté.

Si un problème est détecté sur un disque actif - par exemple D2 - le système de disques peut encore fonctionner en mode dégradé (le fonction miroir n'est plus assurée)

sur D1. Le boîtier procède alors à la *reconstruction* automatique du miroir : il met alors automatiquement en route le disque D3, et y fait une copie du disque D1 qui est encore valide. Quand cette copie est terminée, le système fonctionne à nouveau en miroir sur D1 et D3.

En général le boîtier possède aussi une interface réseau, et fait remonter une alerte à l'administrateur pour qu'il remplace rapidement le disque défectueux par un nouveau disque de rechange.

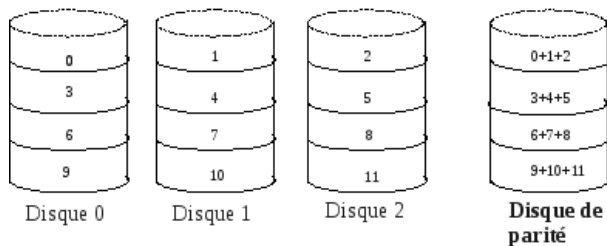
#### 4.4.6 RAID 4 : agrégation par bandes avec parité

Le principe du RAID 4 est une amélioration du principe du miroir. Au lieu de dupliquer les données, on utilise un système de "parité" qui permettra de reconstituer le disque manquant.

Partons d'un système RAID 0 à trois disques A, B, C : les trois premiers blocs du système sont sur les premiers blocs des disques, les trois blocs suivants sur les seconds blocs etc. Ajoutons un quatrième disque D, sur lequel nous stockerons, dans le Nième bloc, un ou-exclusif des Nièmes blocs des 3 disques de données.

$$D_n = A_n \oplus B_n \oplus C_n$$

**Striping + parité**



RAID 4 : agrégation par bandes avec parité

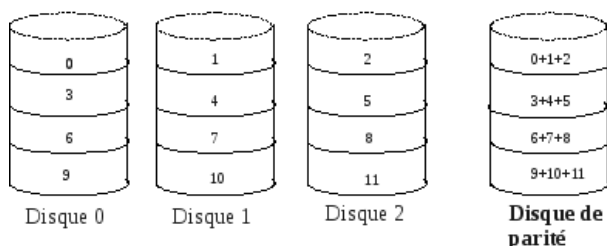
Si le disque A tombe en panne, il sera possible de reconstituer son contenu. En effet, les propriétés de l'algèbre de Boole font que le bloc  $A_n$  est le ou-exclusif des autres blocs de la même "bande" :

$$A_n = B_n \oplus C_n \oplus D_n$$

Le RAID 4 permet donc d'avoir de la redondance sans doubler la taille totale des disques. Les systèmes usuels ont 4 ou 5 disques, la place disque nécessaire à la redondance est donc de l'ordre de 20 ou 25%.

Il est possible de mener de front plusieurs lectures, d'où une possibilité de gain de performances.

**Striping + parité**

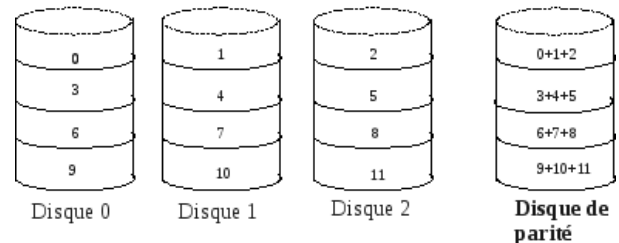


Lectures en parallèle

Par contre, pour modifier un bloc, il faut modifier aussi le bloc "de parité", et pour le calculer, il faut avoir lu les blocs de la bande. Une écriture peut donc prendre le temps de deux accès disque.

Remarquez aussi que le disque de parité est sollicité à chaque écriture. Il constitue donc un goulet d'étranglement.

**Striping + parité**

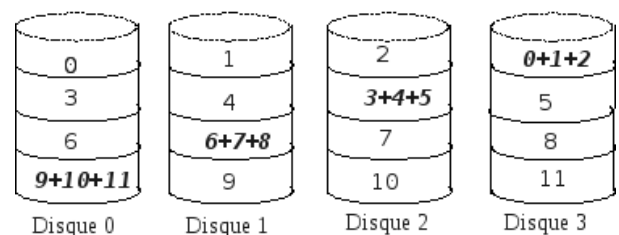


*goulet d'étranglement en écriture*  
(accès au disque de parité)

#### 4.4.7 RAID 5 : agrégation par bandes avec parité répartie

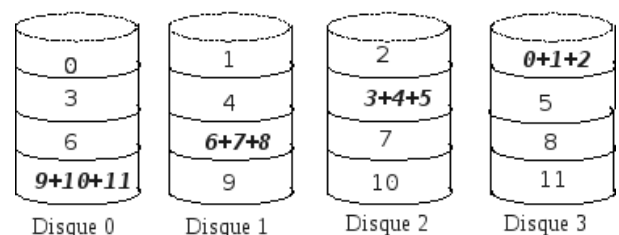
Le RAID 5 est une amélioration du RAID 4 pour mieux répartir la charge : selon les bandes, le bloc de parité n'est pas situé sur le même disque.

**Striping + parité + répartition**



RAID 5 : agrégation par bandes avec parité répartie

**Striping + parité + répartition**



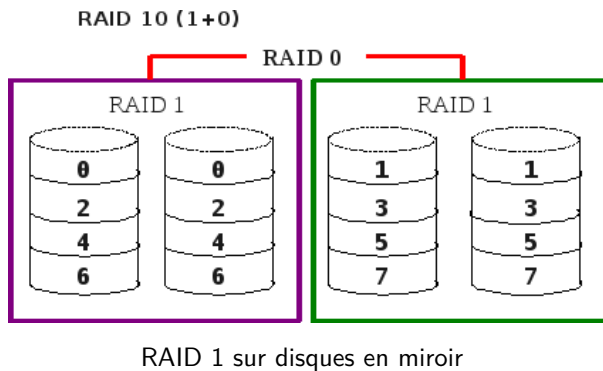
la charge en écriture est répartie

Ceci conduit à de meilleurs performances.

#### 4.4.8 Combinaisons : RAID 10

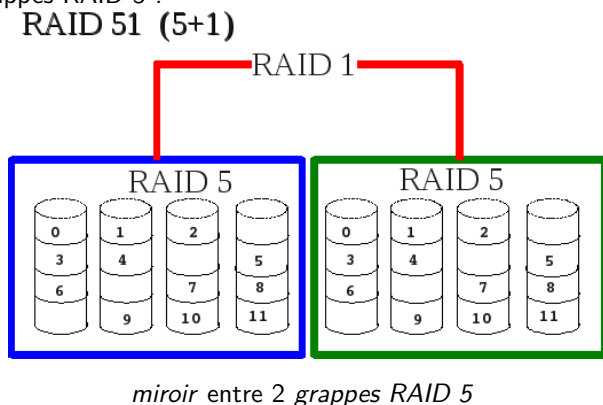
Les techniques vues précédemment sont combinées entre elles pour combiner, selon les besoins, redondance et performances.

Par exemple, le RAID 10 (ou RAID 1+0) est une "grappe" RAID 0 (striping) de paires de disques en miroir :



#### 4.4.9 RAID 51

De la même façon, le RAID 51 est un miroir de deux grappes RAID 5 :



#### 4.4.10 RAID 50

Laissé en exercice :

**Exercice 38.** Dessinez un système de stockage RAID 50. Comparez performance et fiabilité avec le RAID 51

#### 4.4.11 Choix d'un système RAID

Les systèmes RAID servent à réaliser des systèmes de stockages de données, avec des objectifs de capacité, de fiabilité et de performances qui sont supérieurs à ceux des disques individuels. C'est une technologie qui est de nos jours indispensable en environnement professionnel.

Les objectifs sont atteints en combinant diverses techniques : la duplication, la répartition, la parité.

De multiples combinaisons sont possibles : pour faire le bon choix dans un contexte donné, il faut les comprendre.

### 4.5 Systèmes de fichiers

Les *systèmes de fichiers* sont des structures de données, stockées sur disque, qui représentent des fichiers, des répertoires etc.

Vu par l'utilisateur du système (c'est à dire le programmeur d'applications), un fichier possède un contenu, et des méta-données :

- sa taille
- son propriétaire
- les droits d'accès
- la date de création
- date de dernier accès
- ...

#### 4.5.1 Fonctions du SGF

Le Système de Gestion de Fichiers (SGF) contient des fonctions pour

- Manipulation des fichiers : créer/détruire des fichiers, ...
- Allouer de la place sur mémoires secondaires
- Localiser des fichiers : accès au contenu
- Sécurité et contrôle des fichiers
- Fiabilité en cas de panne
- ...

On s'intéresse ici à la représentation des fichiers et des répertoires sur les disques.

#### 4.5.2 Catalogue de fichiers

Les premiers disques avaient une capacité très faible (quelques méga octets). Selon un principe constant en histoire de l'informatique,

*faire du vieux avec du neuf*

ils ont été d'abord gérés en utilisant les méthodes issues des technologies précédentes, à savoir les bandes magnétiques.

Les bandes magnétiques sont un support à accès essentiellement séquentiel (il existe quand même un moyen de "bobiner" plus vite, en avant ou en arrière) sur lequel les fichiers sont stockés les uns après les autres. En début de bande, on met un premier petit fichier, qui contient le nom de la bande, et la liste des fichiers présents.

Les premiers disques étaient gérés de la même façon (VTOC = Volume Table of Contents, d'IBM) : au début du disque se trouve une table, qui contient les noms, positions de départ (numéro de bloc) et taille de chaque fichier. On trouve aussi la liste des espaces inutilisés, pour pouvoir allouer de nouveaux fichiers.

Cette manière de faire possède tous les inconvénients que l'on connaît pour la gestion de la mémoire par blocs contigus : il se produit fatalement une certaine fragmentation, et il est difficile d'allonger un fichier existant.

Par contre, pour une utilisation en mono-tâche les performances sont très bonnes : les données d'un même fichier sont proches les unes des autres, ce qui minimise le temps d'accès ?

A noter : avec des disques de taille aussi réduite, il n'apparaît pas nécessaire d'avoir une organisation hiérarchisée sous forme de répertoires.

### 4.5.3 FAT : file allocation table

Une autre solution est d'utiliser deux tables

- une table d'allocation qui indique les méta-données de chaque fichier, et la position (index) de son premier bloc ;
- une table d'index donnant, pour chaque bloc, la position de son successeur.

La table d'index est chargée en mémoire au montage du disque.

Ceci permet de gérer les fichiers de manière similaire à la mémoire paginée : les blocs d'un fichier ne sont pas contigus, ce qui fait qu'on ne rencontre pas le problème de "gruyérisation".

Par contre, la dispersion sur le disque des blocs d'un même fichier peut avoir un impact important sur les performances, parce que le disque est un support à temps d'accès non-uniforme.

Il y a donc diverses stratégies au niveau du système pour minimiser la dispersion. Par exemple le disque sera géré par zones, le système prenant soin d'essayer garder un certain pourcentage de place libre dans chaque zone, pour pouvoir y loger de préférence les blocs supplémentaires des fichiers qui font déjà partie de la zone.

**Exercice 39.** Soit un disque de 10 Mo, géré par blocs de 2 Ko. Quelle serait la taille de la table d'index ?

### 4.5.4 Tables des blocs Unix

Unix est une vaste famille de système d'exploitations, qui supporte une multitude de types de systèmes de fichiers.

**Exercice 40.** Sur les machines Linux regardez dans `/lib/modules/*/kernel/fs` la liste des pilotes qui prennent en charge les systèmes de fichiers. Recherchez à quoi correspondent `aufs`, `jfs`, `ntfs` ?

Dans ce qui suit nous décrivons les principes utilisés dans les premiers systèmes Unix : l'espace disque comporte deux zones

- les méta-données
- les blocs de données (contenu des fichiers)

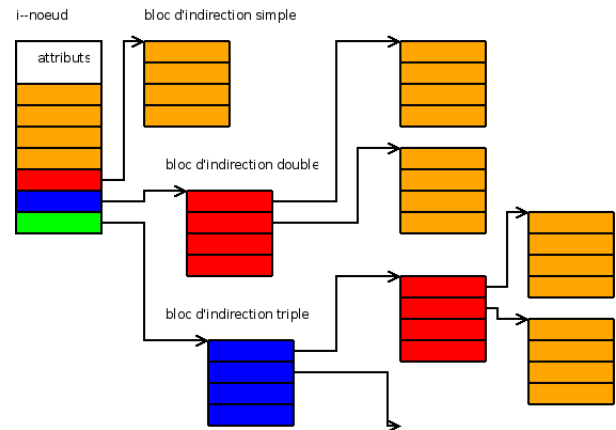
À chaque fichier est associé un *i-node* (noeud d'information)

- qui contient des attributs (taille, propriétaire, droits...)
- qui permet de retrouver les adresses de ses blocs de données

Un fichier n'a pas de nom intrinsèque, il est repéré par son numéro d'i-node. Nous verrons plus loin que ce sont les répertoires qui servent à faire la correspondance entre des noms et des numéros d'i-nodes.

Pour pouvoir retrouver les données, l'i-node contient

- l'adresse des premiers blocs du fichier, ce qui suffit pour les petits fichiers
- l'adresse d'un *bloc d'indirection simple* qui contient d'autres adresses de blocs de données.
- l'adresse d'un *bloc d'indirection double* qui contient des adresses de blocs d'indirection simple<sup>3</sup>
- l'adresse d'un *bloc d'indirection triple* qui contient des adresses de blocs d'indirection double<sup>4</sup>



I-nodes et blocs d'indirection

**Un exemple** Cette structure de données, qui peut paraître un peu baroque, permet d'accéder efficacement au début d'un fichier indépendamment de sa longueur.

Supposons le cas d'une machine avec

- un disque avec des blocs de 4 Ko ( $2^{12}$ )
- des entiers sur 32 bits

En théorie, un système de fichiers, avec des numéros de blocs sur 32 bits, peut contenir jusqu'à 4 milliards ( $2^{32}$ ) de blocs, ce qui représente un espace de  $2^{32} \times 2^{12} = 2^{44}$  octets, soit 16 To.

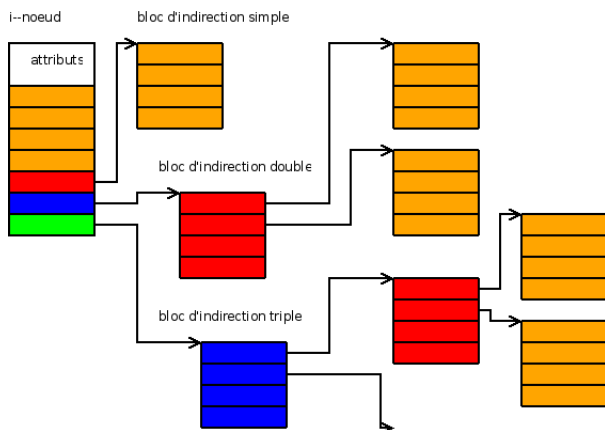
Le système de fichiers a donc une capacité maximale de 16 To. Voyons maintenant la taille maximale d'un fichier :

- les numéros de blocs sont sur 32 bits, donc 4 octets
- un bloc d'indirection simple est de la taille d'un bloc du disque soit 4 Ko. Il peut donc contenir les adresses de 1024 blocs de données ( $2^{10}$ ), soit 4 Mo de données.
- un bloc d'indirection secondaire permettra de référencer les  $2^{20}$  blocs suivants (4 Go), et le bloc tertiaire  $2^{30}$  blocs (4 To).

3. qui contiennent d'autres adresses de blocs de données.

4. qui contiennent des blocs d'indirection simple qui contiennent d'autres adresses de blocs de données.





Pour la plupart des accès, une indirection suffit

#### 4.5.5 Représentation des répertoires

Plusieurs approches sont possibles pour la représentation des répertoires. On peut les considérer comme une extension de la notion de table des fichiers, ou bien les regarder comme des fichiers spéciaux.

##### Catalogues de fichiers

Avec cette approche (CP/M, MS/DOS, Windows...), les "catalogues" sont matérialisés par des entrées spéciales de la table des fichiers qui renvoient vers d'autres parties de la table.

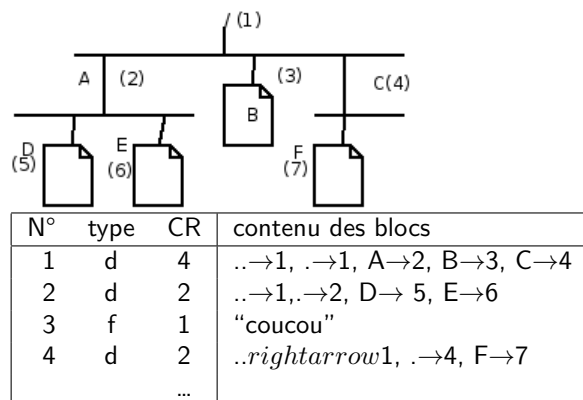
Ceci interdit les *liens* comme on les connaît sous Unix, c'est à dire qu'un même fichier soit visible à des endroits différents de l'arborescence. En effet, chaque entrée du catalogue correspond à des méta-données distinctes; dupliquer des méta-données conduirait rapidement à des incohérences lors de la modification d'un fichier.

##### Comme des fichiers de données

C'est la représentation utilisée par UNIX : chaque i-node contient un indicateur qui précise le type de l'objet, qui peut être

- un fichier normal. Les blocs de données correspondent au contenu du fichier.
- un répertoire. Dans ce cas les blocs contiennent une table de correspondance entre les noms des objets présents dans ce répertoire, et leurs numéros d'i-node
- un lien symbolique. L'i-node contient le chemin d'accès.
- un périphérique,
- ...

**Exemple** Table des i-nodes



Le compteur de références (CR) indique combien de fois un objet est cité dans les répertoires. Quand il n'est plus cité, il est inaccessible et donc on peut récupérer la place qu'il occupe.

**Exercice 41.** Sur ce schéma, étudiez l'effet successif des commandes

- `ln /B /A/G`
- `rm /B`
- `rm /C/F`
- `mkdir /C/X`

**Gestion des blocs libres** Le système possède

- une liste des blocs libres
- un tableau de marquage des blocs occupés

**Vérification du système de fichiers** L'utilitaire traditionnellement appelée *fsck* (*File System Check*) sous UNIX effectue un parcours de l'arborescence, et de la liste des blocs libres, pour en vérifier la cohérence.

1. vérification des i-nœuds, des blocs et des tailles
2. vérification de la structure des répertoires
3. vérification de la connectivité des répertoires
4. vérification des compteurs de référence
5. vérification de l'information du sommaire de groupe

Certaines corrections sont effectuées automatiquement :

- Dans le cas où des répertoires contiennent des références à des i-nodes détruits, les références sont supprimées.
- Dans le cas inverse où des i-nodes "vivants" ne sont plus référencés, ils sont rattachés arbitrairement au répertoire "lost+found".

D'autres nécessitent l'intervention de l'administrateur, par exemple quand des fichiers font usage de blocs qui sont marqués comme libres. Il faut alors choisir entre supprimer le fichier, ou rattacher le bloc au fichier.

#### 4.5.6 Autres caractéristiques des SGF

Les systèmes de fichiers modernes ont d'autres fonctionnalités importantes.

## Journalisation

### Journal :

- garde une trace des opérations d'écriture non terminées
- fournit un *point de reprise* au cas où l'opération serait interrompue (plantage, coupure de courant) pendant une écriture sur disque.

#### Avantages

- pas de pertes d'informations
- reprise plus rapide (évite le *fsck*)

### Snapshots (instantané)

Les "instantanés" sont des copies (virtuelles) de l'état du système de fichiers à un moment donné.

**Besoin** Ceci est utile en particulier pour les sauvegardes. Quand on fait une sauvegarde avec *tar* par exemple, ce programme établit la liste des fichiers à sauvegarder avec leurs noms, leurs tailles etc, puis construit une archive contenant cette liste et le contenu des fichiers.

Or il se peut que les données à sauvegarder "bougent", parce que les utilisateurs (ou des programmes) ajoutent, suppriment ou modifient les fichiers entre la constitution de la liste et la construction de l'archive; ce qui peut conduire à une archive inexploitable.

**Fonctionnement** Quand on demande un "snapshot", le SGF prend note de toutes les modifications apportées au système de fichier "actif" à partir d'un moment précis. Le snapshot est constitué d'une table des sauvegardes des blocs qui ont été modifiés sur le système de fichiers "actif". Lorsqu'on demande le bloc numéro *n* du snapshot, le SGF renvoie la sauvegarde du bloc *n* si il y en a eu une, et sinon le bloc *n* du système de fichiers. La commande de sauvegarde peut donc, à travers le "cliché", accéder aux données dans l'état où elles étaient à un moment précis.

En quelque sorte, c'est une duplication partielle du système de fichiers, au fur et à mesure des modifications; le snapshot est effectué instantanément (au moment de sa création, la table des blocs modifiés est vide), et prend peu de place quand le système de fichiers est peu actif.