STRUCTURE DES ORDINATEURS ET DES SYSTEMES

Johann PAVIS (selon source : M. CASTANET)

Table des matières

1 Introduction

1.1 Définition

Informatique : traitement de l'information

objet traité : information → représentation, modélisation, traitement, algorithme, transfert, visualisation.

1.2 Vers le concept d'ordinateur

homme → aide → *physique*
→ *calculatoire*

aide physique	aide calculatoire
• machines "simples"	• machine à calculer (Pascal)
• machine à programme fixe	• machine à calculer avec "programmes"
• machine à programme "externe"	• machine mécanographique(trieuse)
• robot	• ordinateur

1.3 Générations d'ordinateur

1ère **génération** → **45/55** Technologie : à lampes

Support : cartes perforées, rubans magnétiques

Langage : langage machine

2nde **génération** → **55/65** Technologie : transistors, circuits à composants discrets

Support : cartes perforées, bandes, terminaux imprimants, disques et tambours magnétiques

Langage : Fortran, Cobol, Lisp

Très nombreux constructeurs.

3ème **génération** → **65 / 75** Technologie : circuits intégrés

Support : disques magnétiques, terminaux (écran + clavier)

Langage : très nombreux à se développer

Notion de gamme d'ordinateurs.

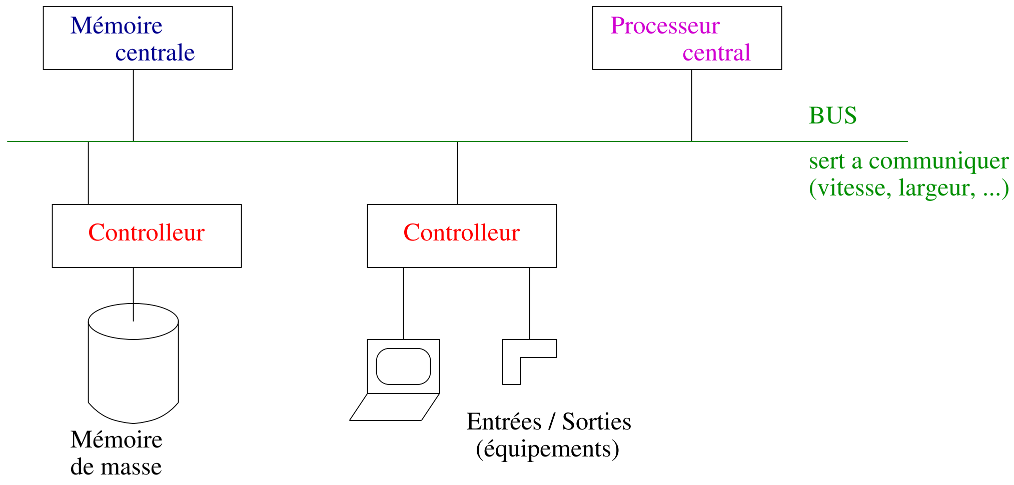
4ème **génération** → **75 - ?** Technologie : circuits VLSI (Very Large Scale Integration)

Support : disques, disquettes, CD

Architecture : architecture répartie, à l'inverse de l'ordinateur central (→ réseau, serveur, ...), répartition mondiale (*Internet*)

Interface homme / machine → image et son

1.4 Architecture générale d'un ordinateur



1.4.1 Mémoire centrale

- stocker des $\left\{ \begin{array}{l} \text{données} \\ \text{programmes} \end{array} \right.$ (infos)
- capacité
- temps d'accès

1.4.2 Processeur central

- calcule ou traite des informations
- exécution d'instructions
- rapidité (nombre d'instructions à la seconde)

1.4.3 Mémoire de masse

- stocke des informations
- capacité de plusieurs Go
- temps d'accès : qqs *ms*

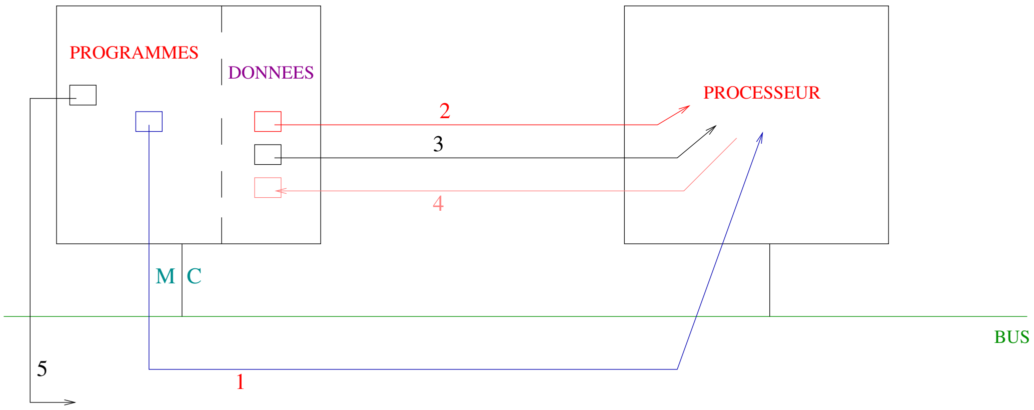
1.4.4 Contrôleur

→ processeur spécialisé

Équipements d'entrée / sortie

- interface homme / machine
- conversion d'informations

1.5 Fonctionnement d'une machine de type **Von Neumann**



2 Composantes de l'architecture

2.1 Codage de l'information

- informations : caractères, images, chiffres, instructions, ...
- codage de base : codage binaire $\{0{,}1\}$, codages dérivés (octal, hexadécimal, ...) (→ conversions possibles)
  - codages des nombres entiers $\boxed{1\ 0\ 0\ 1\ 1\ 0\ 1\ 1}$

  Pour les nombres relatifs, on utilise le codage suivant pour le premier bit : $\begin{cases} \text{positif} \longrightarrow 1 \\ \text{négatif} \longrightarrow 0 \end{cases}$

  Représentation

signe (1*bit*)	*valeur* (7*bits*)

  → complément à un

Nombre initial

0	1 0 1 1 0 0 1

↓

complément à 1

1	0 1 0 0 1 1 0

→ complément à deux

Nombre initial

0	1 0 1 1 0 0 1

↓

complément à 1

1	0 1 0 0 1 1 0

↓

complément à 2

1	0 1 0 0 1 1 1

n (exposant)	m ("valeur")

• codage des réels

→ $0, m * 2^n$

Il y a cette fois un double signe :
⋆ pour l'exposant
⋆ le nombre réel

• codage des caractères → ASCII : sur 7 bits (128 caractères codables)
→ EBCDIC : sur 8 bits (256 caractères codables)
→ codes étendus

caractères spéciaux (2 ou 4 bits)	code caractère

• codage des images ⋆ pixel (1 point de l'image)
⋆ noir et blanc (→ 1 pixel = 1 bit)
⋆ gris sur 128 niveaux (→ 1 pixel = 7 bits)
⋆ codage RGB sur 256 couleurs (→ 8 bits/couleur ⇒ 24 bits/pixel)

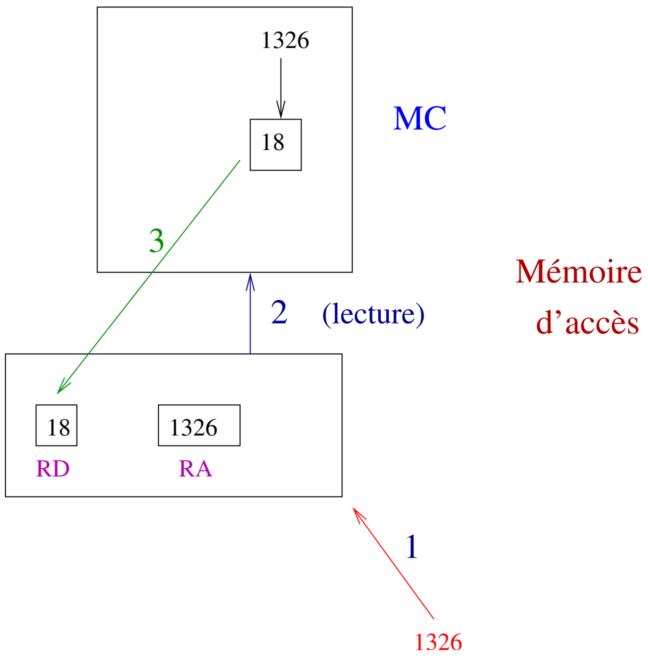
Remarque Comment différencier une information ?

10110011100 ? → tout dépend de la sémantique qui la traite.
⇒ exécuter l'information : **instruction**
⇒ additionner avec un entier : **nombre entier**
etc

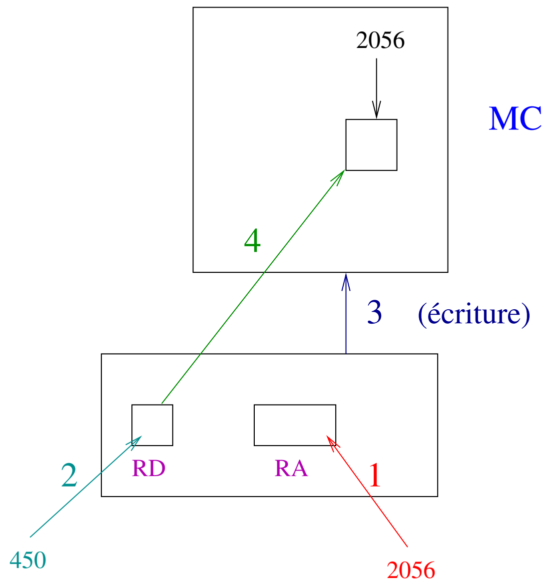
2.2 Structure et fonctionnement de la mémoire centrale

⋆ Pôle : stockage d'informations
⋆ Structure logique : ensemble de cases mémoires
⋆ Case : taille de 8,16,32 ou 64 bits
⋆ Adresse d'une case : mémoire grande ⇒ adresse longue
⋆ Opérations de base :

Lecture :



Écriture :



Remarque
lecture → écrase l'ancien contenu du registre de donnée (RD)
écriture → conserve le contenu du RD après l'écriture

⋆ Temps d'accès et cycle :

temps d'accès : qqs ms

cycle mémoire : > temps d'accès (au maximum, le double)

⋆ Protection des données : mémoire non fiable ou transfert non fiable

bit de parité : $\boxed{1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1}$

Le dernier bit est le bit de parité. On rajoute, au besoin, un 1 afin d'en obtenir un **nombre pair**.
Contrôle de Hamming n bits d'informations + k bits de contrôle (k tel que $2^k > n + k$).
Les k bits sont imbriqués dans les bits de positions 2^r où $r \in [0, k-1]$.
Chaque r bit contrôle la parité des autres r bits :
le bit p est contrôlé par les bits $b_1, b_2, \ldots, b_j$ de sorte que : $b_1 + \ldots + b_j = p$

bit 1 contrôle	:	1,3,5,7,9,11
bit 2 contrôle	:	2,3,6,7,9,10,11
bit 4 contrôle	:	4,5,6,7
bit 8 contrôle	:	8,9,10,11

Le bit 5 est contrôlé par les bits 4 et 1 (en effet : $5 = 4 + 1$).

Exemple : Caractère ASCII

1	0	1	1	0	1	1

On protège les données

1	1	1	0	0	1	1	0	0	1	1
$\downarrow$ 2^0	$\downarrow$ 2^1		$\downarrow$ 2^2				$\downarrow$ 2^3			

Une erreur se produit : le $7^{ème}$ bit devient 0

1	1	1	0	0	1	**0**	0	0	1	1
$\downarrow$ 2^0	$\downarrow$ 2^1		$\downarrow$ 2^2				$\downarrow$ 2^3			

A la réception, on effectue le contrôle de Hamming : on compte le nombre de 1 qui suivent chaque bit de contrôle (*i.e.* ceux qu'ils contrôlent (*cf* ci-avant)).

bit 8 $\Rightarrow$ OK
bit 4 $\Rightarrow$ FAUX
bit 2 $\Rightarrow$ FAUX
bit 1 $\Rightarrow$ FAUX

$4 + 2 + 1 = 7 \Rightarrow$ c'est le bit 7 qui est faux. Il "redevient" 1.

⋆ Types de mémoires – RAM → lecture <u>et</u> écriture
- ROM → lecture uniquement
- PROM
- REPROM
- Tore de ferrite → deux positions stables
- Transistors → mémoires statiques et dynamiques

⋆ Optimisation ***Technologie*** : taille + accès

Architecturale :
- *lecture groupée* (lecture anticipée)
- *lecture entrelacée* (lit le paquet qui suit celui demandé)
- *lecture multi-bloc* (idem, mais avec plus de blocs)
- *mémoire cache* (antémémoire)

Dans ce cas, la mémoire centrale, **lente**, envoie les informations nécessaires à l'antémémoire. La mémoire centrale est à cet effet divisée en lignes et colonnes. Chaque bloc de l'antémémoire correspond à une colonne de la mémoire centrale.

Il existe un élément qui indique le numéro de la ligne du présent en antémémoire ($Pos \rightarrow Vecteur$).

Lecture (adresse) `adr : l c m`

initialement `copie MC ↔ AM`
ensuite `adr l' c' m'`
`pos(c') = l'`

Deux stratégies possibles :
▷ écrire simultanément en mémoire centrale et antémémoire
▷ on écrit seulement en antémémoire, on effectue une copie du bloc dans la mémoire centrale par mise à jour
⇒ le temps d'accès apparent est amélioré !

2.3 Processeur central

⋆ Exécute des instructions
⋆ Jeu d'instructions (50 à 500) : addition, multiplication, décalage, test, branchement, chargement mémoire, ...
Il existe deux types principaux de processeurs :
$\begin{cases} \text{RISC (Reduced Install Set Computer)} \\ \text{CISC (Complex Install Set Computer)} \end{cases}$
⋆ Instruction Paramètres :
code instruction @ 1^{er} param , @ $2^{ème}$ param , @ résultat, @ instruction suivante
Optimisation :
On n'indique pas d'adresse de résultat, en considérant que le résultat est stocké à l'adresse du second paramètre. On n'indique pas non plus l'adresse de l'instruction suivante, qui est sous-entendue.
En général, les instructions sont codées sur 32 bits :

32 bits

code	registre	adresse arg 2

⋆ Vitesse d'exécution :
→ vitesse unitaire par instruction
→ vitesse globale
Les unités utilisées sont le *Mips*, le *Mflops*, le *benchmark*, ...
⋆ Composants :

Registres :
stockage provisoire

Registres d'index :
stocker un indice, puis faire progresser cet indice en changeant le contenu du registre

Compteur ordinal (*PC* pour *Program Counter*) :
permet de connaître l'adresse de l'instruction suivante

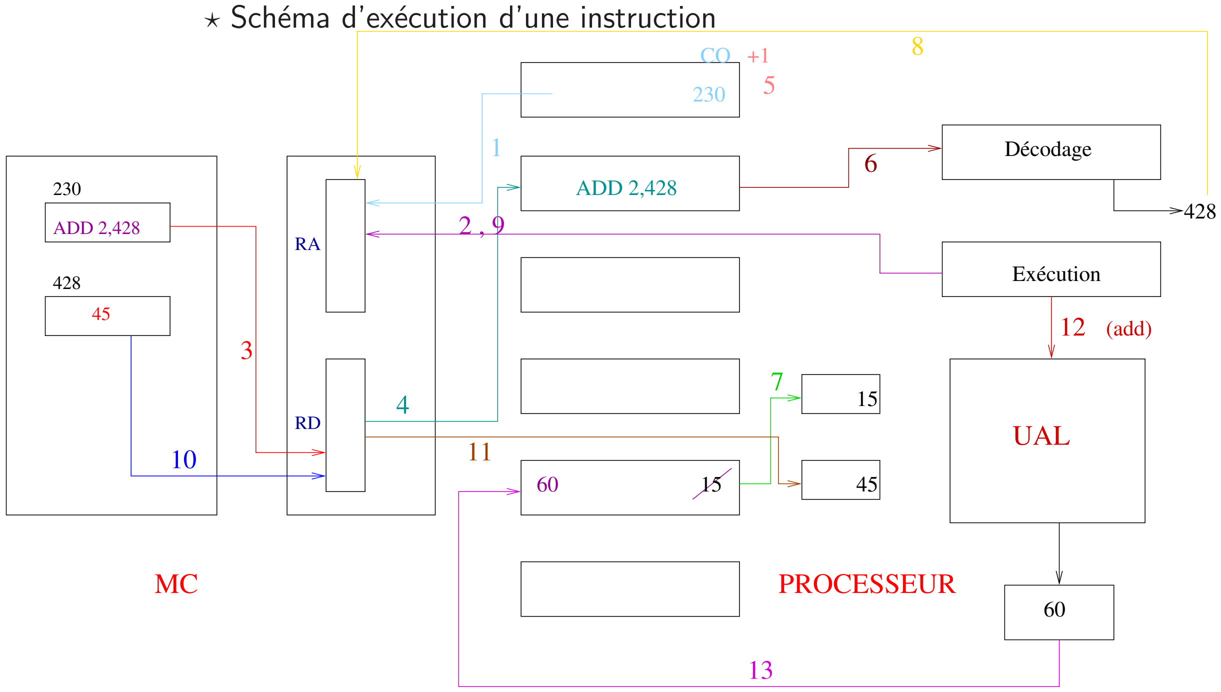
Circuits de décodage et d'exécution

UAL (Unité Arithmétique et Logique) :
calcule arithmétiques, avec 4 instructions de base

Horloge :

caractérisée par sa fréquence. Elle permet de cadencer les micro-opérations à l'intérieur du processeur

⋆ Schéma d'exécution d'une instruction

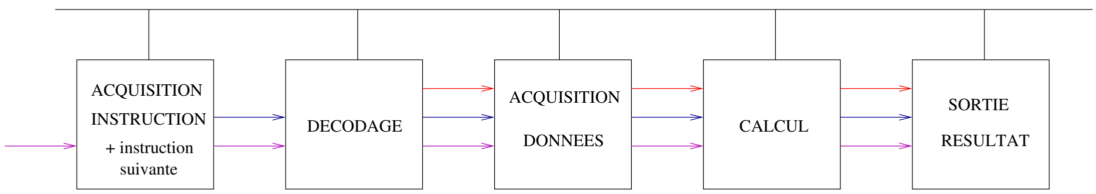


On effectue ici l'addition du contenu du registre 2 et du contenu de l'adresse 428. Le résultat est placé dans le registre 2.
1 à 5 → phase d'**acquisition** de l'instruction
5 à 13 → phase d'**exécution** de l'instruction

⋆ Types de processeur :

→ SISD (Single Instruction Single Datestream) : architecture "classique"

→ MISD (Multiple Instructions Single Datestream) : système en architecture "pipeline"



→ SIMD : *array processor* (processeur vectoriel)
Très utilisé par le calcul scientifique, le calcul d'images, ...

→ MIMD : massivement parallèle

→ RISC / CISC (Reduced / Complex Instructions Set Computer) :

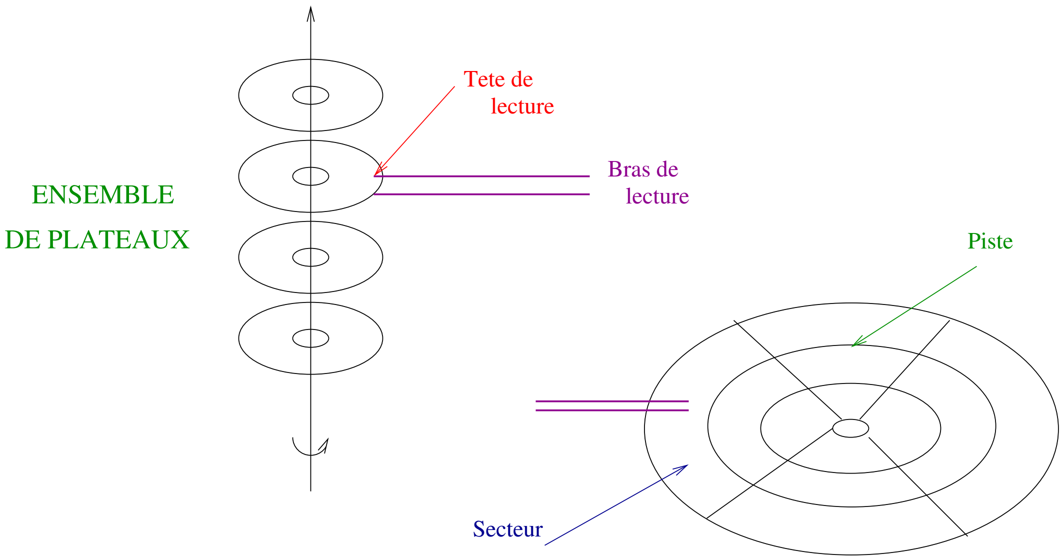
◦ *Type RISC* : une centaine d'instructions au maximum
◦ *Type CISC* : plusieurs centaines d'instruction, parfois longues
◦ *Comparaison des jeux d'instructions* :

```
(CISC)  add @1 , @2 , @3
         load r1 , @1
(RISC)   load r2 , @2
         add r1 , r2
         store r1 , @3
```

◦ *Comparaison avec l'architecture pipeline* :
5 sous-processeurs : chargement, décodage, chargement opérande, exécution, sortie du résultat.
Dans le cas de l'architecture pipeline, il faut veiller à décaler les instructions pour éviter les conflits

2.4 Mémoires auxiliaires

⋆ Rôle : stockage d'informations
⋆ Technologie : magnétique (particule à 2 "positions", 0 ou 1), optique, laser
⋆ Disque magnétique :
→ *Opérations* : lecture / écriture
→ *Structure générale* :



⋆ Capacité d'un disque :
→ *Densité d'informations* : en *bpi* (*byte per inch*) ou *tpi* (*track per inch*). La tête de lecture doit être la plus petite possible et le plus

près possible du disque. La capacité dépend du nombre de pistes et de plateaux.

→ *Capacité globale* : plusieurs dizaines de *Go*

⋆ Temps d'accès : adresse disque (numéro du cylindre, numéro de face, numéro de secteur, position), puis déplacement des bras ($\simeq 10ms$), sélection électronique, délai rotationnel (1/2 tour de disque) (dépend de la vitesse de rotation : 6 à $10ms$)

⋆ Architecture de disque (pour la protection des données)

→ *Architecture de type RAID* (Redondance And Independant Disc) :

RAID 0 → pas de redondance
RAID 1 → disques miroirs
RAID 2 → contrôle ECC (Hamming), ajout de disques de contrôle
RAID 3 → bits de parité (octet) : 1 bit supplémentaire permettant de rajouter de l'information de contrôle
RAID 4 → entrelacement de bits de parité
RAID 5 → entrelacement d'un bloc de parité
RAID 6 → ≡ RAID 5 et code de Reed / Solomon (correction de 2 erreurs)

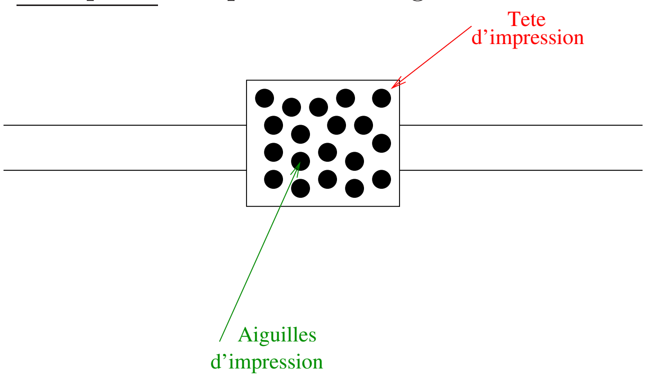
⋆ Autres mémoires :

∘ *Disquette* : technologie similaire au disque
∘ *Cartouches magnétiques* : sur bande magnétique, recopie, sauvegarde de disques, ...(jusqu'à $100Gb$)
∘ *CD* : distribution de logicielles, possibilité de sauvegarde
∘ *Mémoires magnéto-optique* : CD réinscriptibles

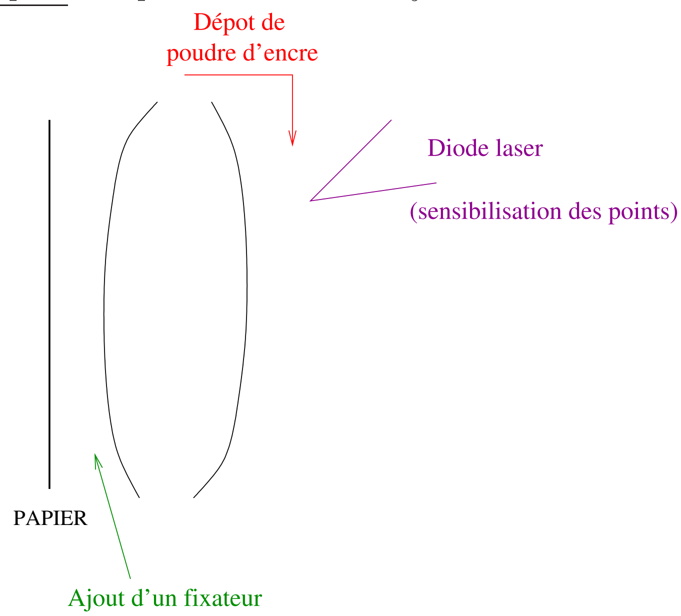
⋆ Unités d'entrée / sortie : Leur rôle est l'interface homme / machine, de rendre les informations internes utilisables par l'homme.

⋆ Unités de sortie :

• *Imprimante* a. impacts : imprimante à aiguilles



b. sans impact : imprimante laser ou à jet d'encre



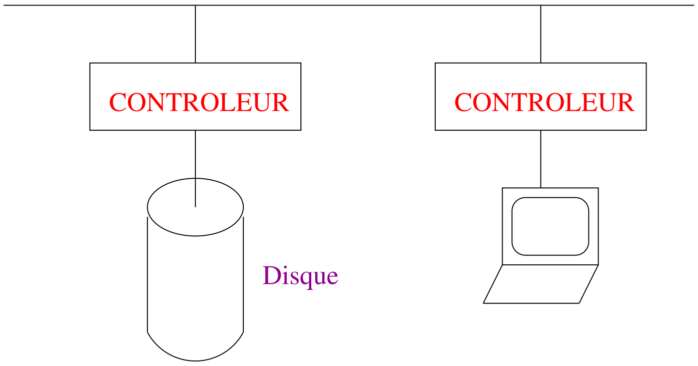
- *Écrans* a. alphanumérique La mémoire ROM transcode l'alphabet. L'impression se fait par ligne (ensemble de caractères), par le biais d'un faisceau qui éclaire l'écran point par point.

  b. graphique La représentation d'une page écran dans une mémoire vidéo. Il faut donc coder chaque point de l'image. Pour l'affichage sur l'écran, c'est également un faisceau qui balaie les points (selon les informations graphiques).
- *Synthétiseurs de sons*
- *Traceur* (table traçante)

⋆ Unités d'entrée :

- *Clavier* touche → symbole → codage
  gestion des rebonds (gestion des répétitions de touche)
  gestion des touches multiples (jusqu'à trois touches)
  touches de fonction → chaîne de caractères
- *Souris* repérage position
  choix de fonctions
- *Analyseur de sons*

2.5 Contrôleurs



- ⋆ Contrôleur : processeur spécialisé pour la gestion des périphériques (mémoires de masse, unités d'entrée / sortie, ...)
- ⋆ Bus : 1 ou plusieurs bus. Il se caractérise par sa largeur (vitesse de transfert). C'est un système à trois états : *lecture, écriture* et *bloqué*.
  C'est par lui que passe l'ensemble des fonctions de contrôle.

3 Système d'exploitation

3.1 Généralités

⋆ Définitions :

Système d'exploitation (ou OS, pour Operating System) :

C'est un ensemble de programmes (gestion des ressources d'un ordinateur) et les services pour le développement et l'utilisation des programmes.

Types de systèmes :

- mono/multi : système mono-constructeur (VM[IBM], GECOS[Bull], ...). Pour les multiconstructeurs, on dispose des systèmes Unix (Linux, ...).
  On a également la possibilité d'avoir deux systèmes d'exploitation sur la même machine.
- mono/multi-tâches : un système mono-tâche ne traite qu'une seule tâche à la fois, pendant un temps donné (avant de passer à la main au processus suivant). Ce processus a à sa disposition toutes les ressources du système.
  Les systèmes multi-tâches sont nettement plus complexes.

Historique des systèmes d'exploitation :

- $1^{ème}$ génération : pas de système d'exploitation
- 2^{nde} génération : système de gestion des cycles de traitement, lecture des programmes, traitement à effectuer, exécution ⇒ résultats.
  Système "*batch*" :
  ```
  $JOB nom_prg + prgeurs
  $fortran (cartes perforées)
  $load
  $load (données)
  $end
  ```
- $3^{ème}$ génération : système complexe et multi-tâches. Démarrage des systèmes Unix.
  Nouvelle dimension utilisable : traitement interactif des données.
- $4^{ème}$ génération : système distribué. Interface homme/machine améliorée. Performances de plus en plus grandes.

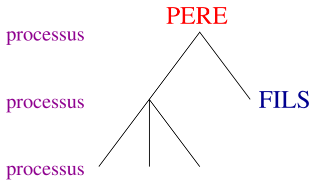
⋆ Concepts généraux des systèmes d'exploitation :

- *Approche objet* : 2 types d'objets :
  → processus : codes exécutables
  → fichiers : données identifiées
- *Processus* :
  - Code exécutable identifié : par le **système d'exploitation** (*pid*) ou par l'**utilisateur** (fonction de traitement).
  - Structuration :

    Espace mémoire accessible

pile
↕
données
code

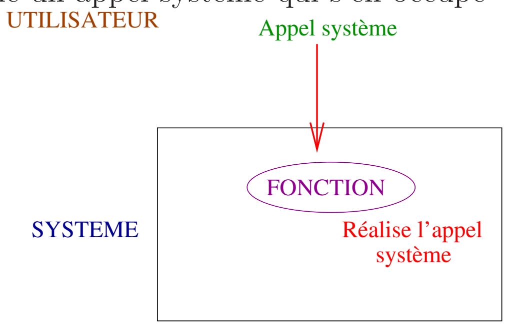
    → zone libre (permet d'ajuster la pile/les données)
  - États d'un processus : en exécution (X), en attente (A) ou prêt (P)
  - Naissance et mort d'un processus :
    - notion d'appel système (requête) : opération (lié à l'objet)
      Pour de nombreux systèmes : notion de hiérarchie de processus :



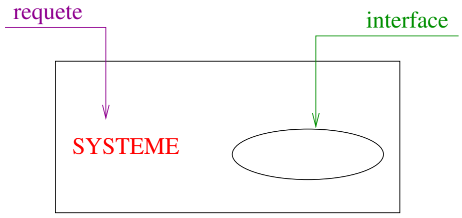
    - destruction d'un processus (tuer, *kill*) : problème de gestion dans le cas de la hiérarchie de processus (tuer une feuille ne pose pas de problème, à l'inverse de tuer un processus "*noeud*").

Il y a deux stratégies : tuer **tout le sous-arbre**, ou **attendre la fin des processus fils** avant de tuer le processus.

– communication de processus : à chaque opération sur un processus, on associe un appel système qui s'en occupe



- Fichiers :
  - ensemble de **données structurées**
  - interface de **gestion**



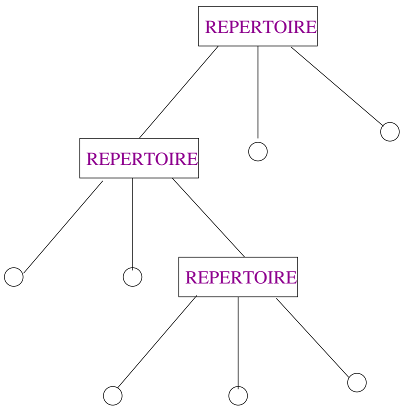
  - ensemble de requêtes : création, destruction, ouverture, fermeture, lecture, écriture, autres spécificités
  - organisation des fichiers : notion de répertoire

entrée $\longrightarrow$

Nom	localisation

répertoire

  - types de répertoire
    $\longrightarrow$ répertoire à plat (tous les noms doivent être différents)
    $\longrightarrow$ répertoire hiérarchisé



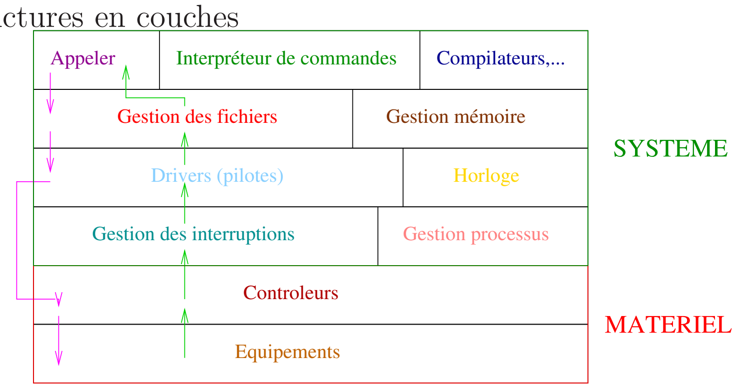
◦ expression d'un chemin :
→ utilisation de 2 opérateurs : **descente** ({/} , {\} , {>} , {(}) et **montée** ({\} , {/} , {<} , {)})
→ chemin absolu : chemin à partir de la **racine**
→ chemin relatif : nécessité de notion de **répertoire courant** (chemin complexe)

◦ protection des fichiers :
→ technique de mots de passe
→ technique de protection globale, par triplet de triplet (pour les **utilisateurs** et les **opérations**)
→ accès direct de l'utilisateur au système

◦ langage de commandes : nécessité d'utiliser un interpréteur de commandes

- Structure d'un système et fonctionnalités :

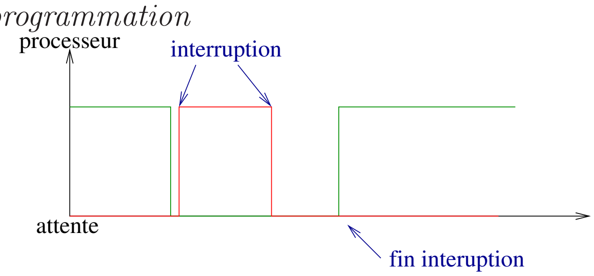
◦ Structures en couches

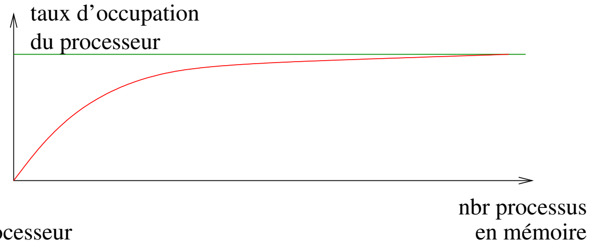


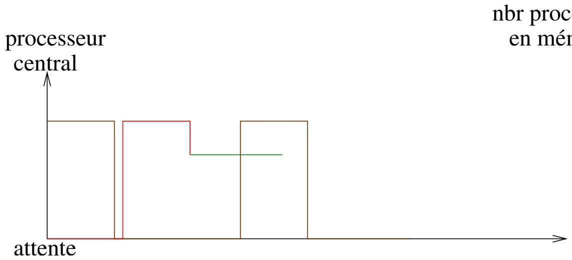
◦ Fonctionnalités d'un système
– *Mono-programmation, -utilisateur*

$$\text{Applications} \longrightarrow \boxed{\text{SYSTÈME}} \longrightarrow \text{Résultats}$$
$$\updownarrow$$
$$\text{données}$$

– *multi-programmation*

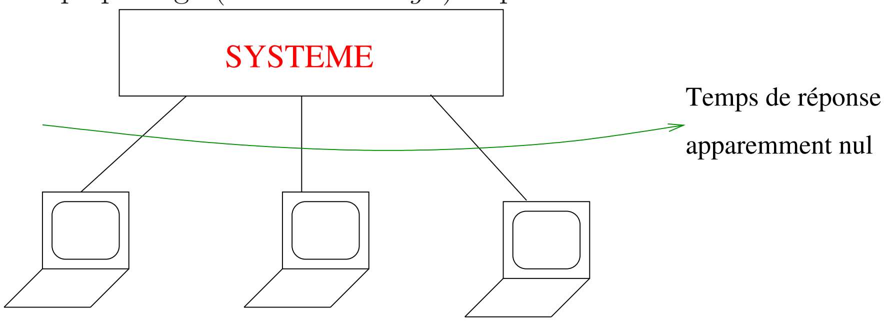


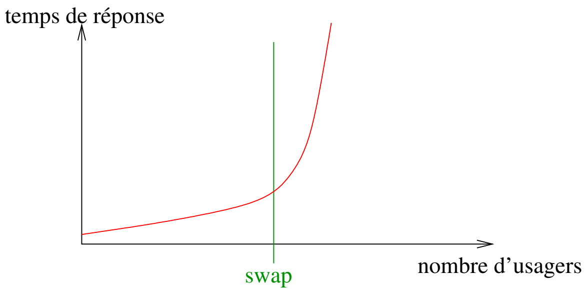




∘ Nouvelles fonctionnalités : gestion des interruptions, gestion des processus, gestion des mémoires

∘ Temps partagé (“*Time sharing*”) : optimisation de l'utilisation



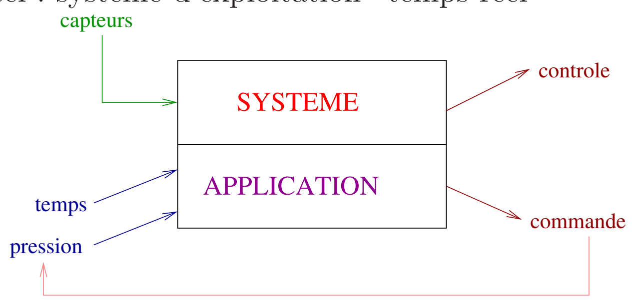


⇒ Nouvelles fonctionnalités : gestion des **usagers** et du **temps**.

∘ Muli-tâches

Les opérations sont découpées en plusieurs **processus coopérant**
⇒ Nouvelles fonctionnalités : gestion et **communication** inter-processus

◦ Temps réel : système d'exploitation "temps réel"



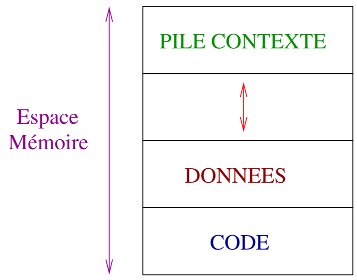
⇒ Nouvelles fonctionnalités : tâches **haute priorité**, respect du **temps de réponse**

Remarque : les contraintes du temps réel sont très fortes

3.2 Processus

⋆ Généralités :

- *Processus* : code identifié par le système (*pid*) et l'utilisateur (*fonction*)
- *Structure* :

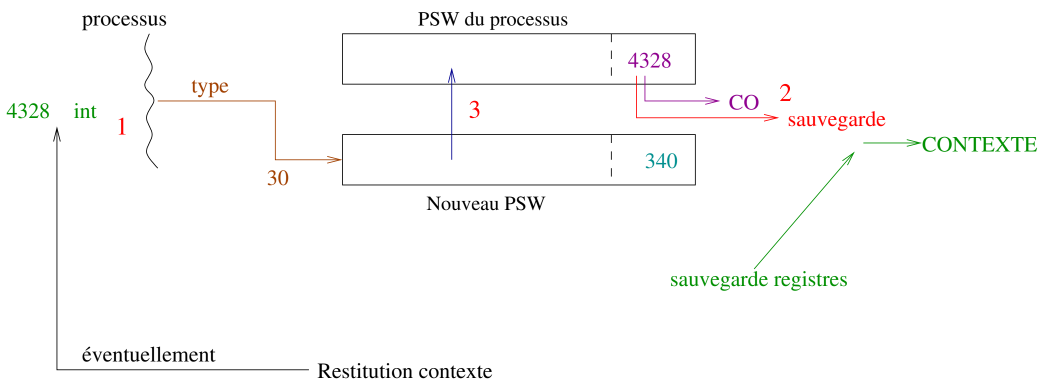


- *liens fichier/processus* fichier entrée → $\boxed{\text{processus}}$ → fichier sortie
  ↕
  fichiers de travail
- *Vie d'un processus* ◦ Création, destruction
  ◦ Évolution :
  - X → en exécution (processus *central* s'il nécessite toutes les ressources)
  - P → prêt (toutes les ressources, sauf le processeur)
  - A → en attente

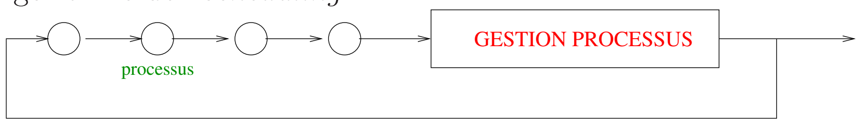
  ◦ Système multi-tâches : primitives de communication entre les différents processus (primitives accès ressources)

⋆ Mécanisme d'interruption :

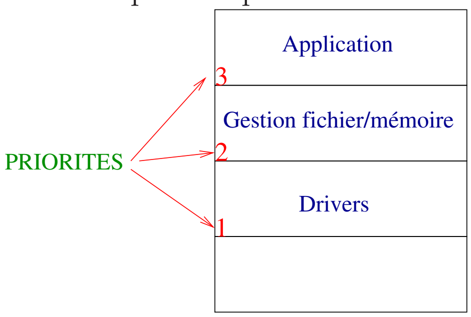
- *Principe* ∘ vie ⇒ nombreuses interruptions
  ∘ interruption *Rightarrow* **sauvegarder** le contexte / **restituer** le contexte
- *Contexte* ∘ compteur ordinal
  ∘ priorité
  ∘ indicateurs de condition
  ∘ masque d'interruption
  ∘ privilège (par exemple, mode `maître` / `esclave`)
  ∘ registres de processus
- *Types d'interruption* ∘ erreur machine
  ∘ erreur programme
  ∘ appel système ou interruption logicielle
  ∘ interruption externe
  ∘ interruption horloge
  (l'ensemble forme ce qu'on appelle l'"*état système* (PSW))"
- *Fonctionnement d'une interruption* A chaque type d'interruption ⇒ zone mémoire : nouveau PSW



- *Choix du processus : $P \rightarrow X$* ∘ file d'attente des processus
  ∘ algorithme de "*scheduling*"



Exemple d'implémentation

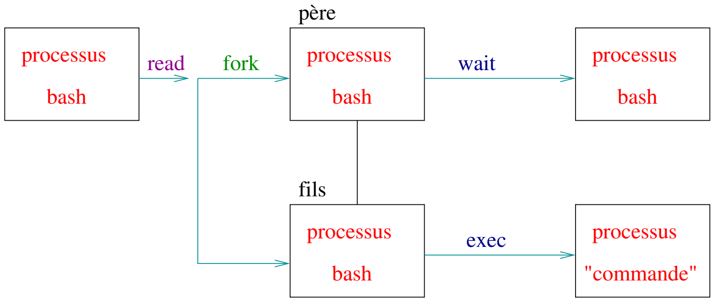


Il y a trois files de priorité (priorité 1, priorité 2, priorité 3). Le principe de l'algorithme de scheduling est l'épuisement des files de priorité par ordre croissant

⋆ Naissance et mort d'un processus :

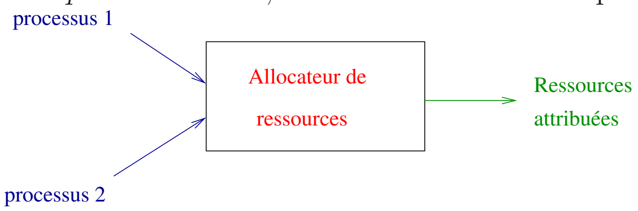
- *Primitive* : requête au système
- *Types de conception* :
  - mono-processus : appli ⇒ processus
  - multi-processus
- *Exemple type UNIX*
  - `fork` (renvoie 0 dans le *pid* du processus fils et renvoie le *pid* du fils dans celui du père)
  - `wait` (attente)
  - `exec(ve)` (remplace le code d'un processus)
- *Exemple processus* bash

```
while (true) {
   read (command,param)
   if (fork() != 0) {
        wait(&status)
      } else {
        exec(ve) (command,param) } }
```

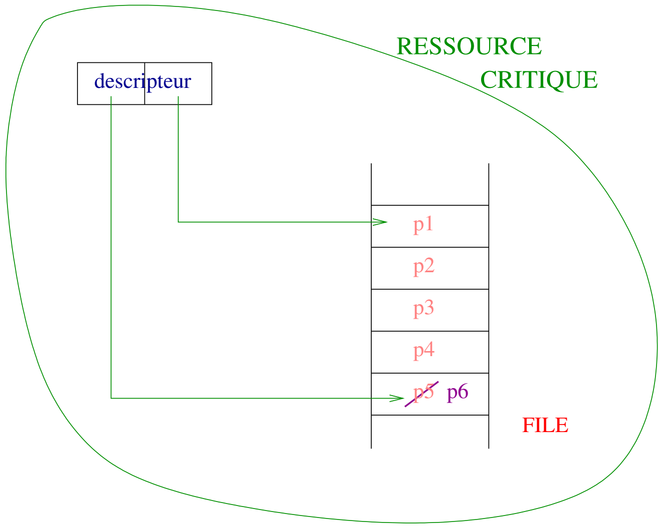


⋆ <u>Gestion des ressources critiques</u> :

- *Ressource critique* : utilisable, attribuable → un seul procédé

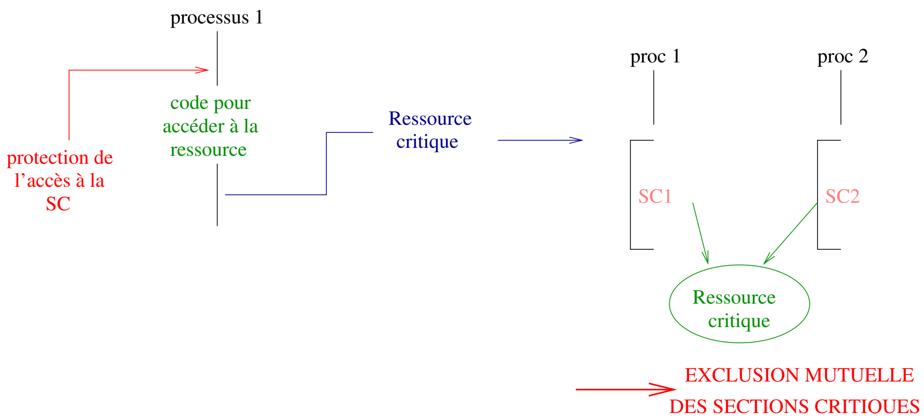


Exemple :

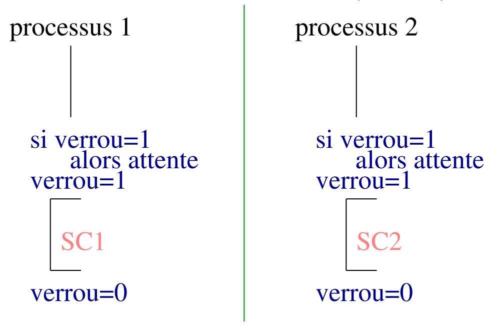


Interruption du processus 5. On met le processus 6 en file ⇒ perte du processus 5 !

- *Section critique (SC)* : protéger l'accès à une ressource critique ⇒ décalage : protéger l'accès au code permettant l'accès à la ressource critique



- *Solution naïve* : verrou à deux positions (0 et 1)



- *Solution* ***test and set*** : test et positionnement d'un verrou en une seule instruction
- *Sémaphore* (Dijstra) ◦ Définition :

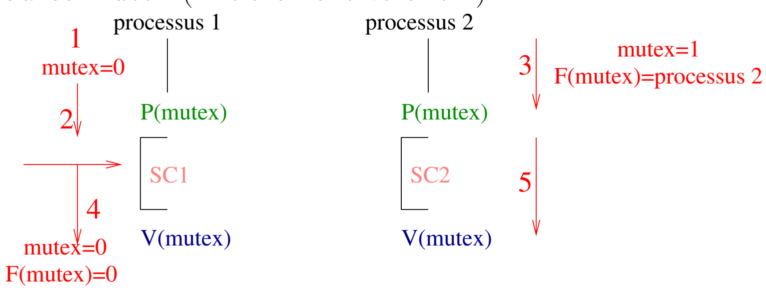
  entier signé, file d'attente $F(s)$ des processus bloqués, 2 primitives P et V.

  P(s) : $s' \leftarrow s + 1$ et si $s < 0$ mise de s dans la file

  V(s) : $s' \leftarrow s + 1$ et si $s \leq 0$ déblocage du processus $F(s)$

  ◦ Utilisation pour l'exclusion mutuelle

  Soit une ressource critique. On crée un sémaphore attaché à cette ressource mutex (initialement valant 1)

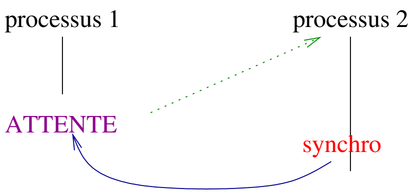


  ◦ Problème d'interruptibilité

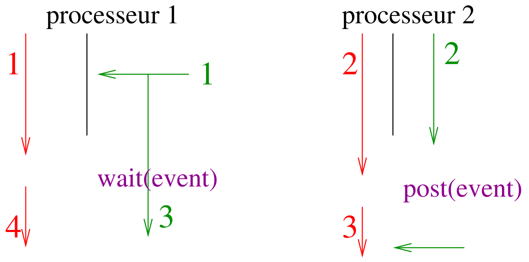
  1. P et V sont des **instructions machines**
  2. P et V sont des appels systèmes. Ils peuvent bloquer tous les interrupteurs
  3. dans le cas du multi-processus, il faut penser à protéger P et V par un **tas**

⋆ <u>Synchronisation de processus</u> :

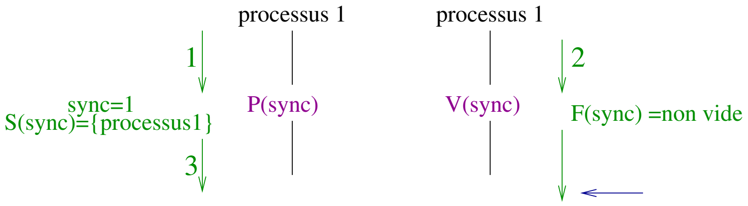
- *Principe* :



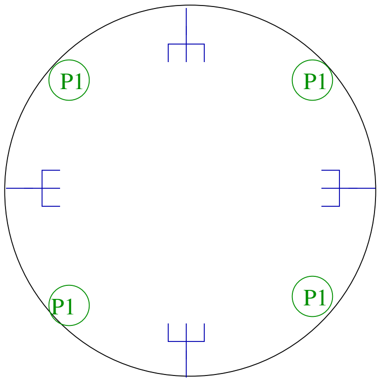
- *Technique* ***wait and post*** Objet : événement (`event`)
  2 primitives : `wait(event)` et `post(event)`
  Programmation :



- *Sémaphore* Sémaphore : `sync` (initialement à 1)
  Programmation :



- *Exemple de synchronisation multiple* : les philosophes et les spaghettis
  Cycle de vie des philosophes : pense, veut manger, mange (... et ainsi de suite)



  La ressource critique est la fourchette.
  Pour le philosophe : pense, acquiert fourchettes, mange, restitue fourchettes (retour au départ)
  Programmation générale : sémaphores

`sempriv[i]` $\rightarrow$ synchronisation entre φ_i

`mutex` $\rightarrow$ protection section critique (initialement à 1)

```
acquisition fourchettes
   si etat[φ_{i+1}]=2 ou etat[φ_{i-1}]=2
   alors etat[φ_i]=1 et blocage P(sempriv[i])
   sinon etat[φ_i]=2
V(mutex)
```

Transformation du programme (pour gérer les problèmes d'interruption) :

```
P(mutex)
si (etat[φ_{i+1}]≠2 et etat[φ_{i-1}]≠2)
alors etat[φ_i]=2 et V(sempriv[i])
sinon etat[φ_i]=1
V(mutex)
P(sempriv[i])
```

Restitution des fourchettes

```
etat[φ_i]=0
P(mutex)
si (etat[φ_{i-1}]=1 et etat[φ_{i-2}]≠2)
alors etat[φ_{i-1}]=2 et V(sempriv[i-1])
si (etat[φ]≠1 et etat[φ_{i+2}]≠2)
alors etat[φ_{i+1}]=2 et V(sempriv[i+1])
V(mutex)
```

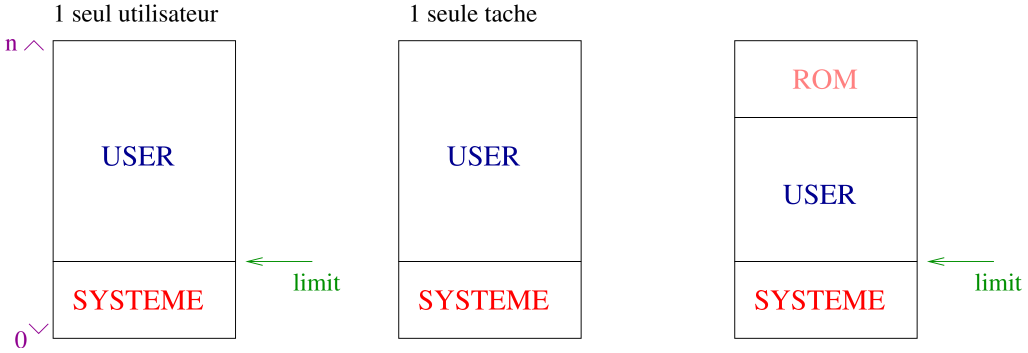
3.3 Gestion de la mémoire centrale

⋆ Généralités :

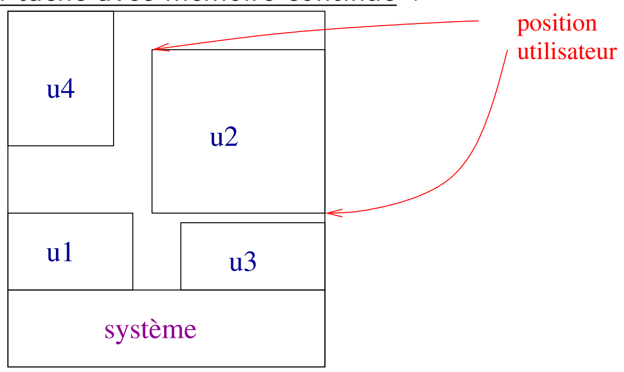
Applications	
Gestion fichiers	**Gestion mémoire**
Drivers	
Interruption	Processus

Ressources à gérer partagées ⇒ gestion du partage d'**allocation** et de **restitution**, protection.

⋆ Gestion mono-tâche et mono-programmation :



⋆ Gestion multi-tâche avec mémoire continue :



- *Phase d'allocation* :
  - processus → besoin de mémoire t
    ⇒ allocation d'espace ≥ t
  - pour l'allocation, le système gère une table d'allocation :

id / processus	début	fin	taille
1234	5238	8238	3000
3456	12500	18500	6000

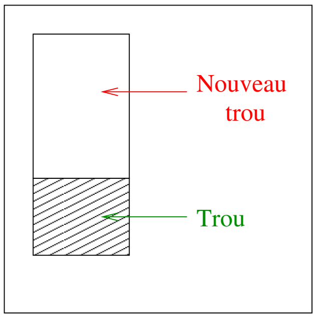
⇒ déduction de l'idée de **trous** :

trous	début	fin	taille
	8239	12499	**4260**

∘ algo :

```
tri table_alloc
deb(alloc(i)) - fin(alloc(i-1))
⇒ trou
```

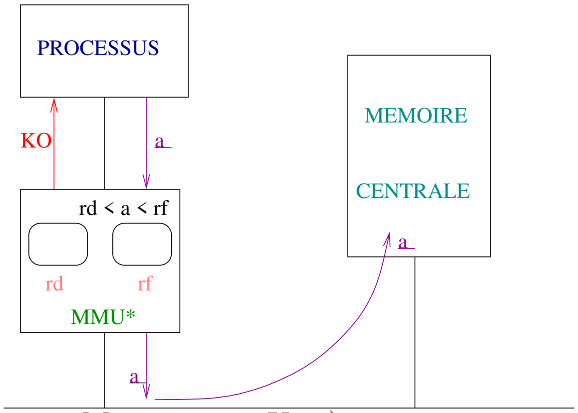
Pour un processus de taille t, on cherche le plus petit processus de taille ≥ t, puis on modifie la table en créant un nouveau trou :



3.3.1 Phase d'exécution

⋆ Processus implanté entre deux limites : début fin

⋆ Architecture :



(* MMU : Memory Management Unit)

⋆ A l'initialisation :

processus P (prêt) → X (exécuter), rd ← début et rf ← fin.

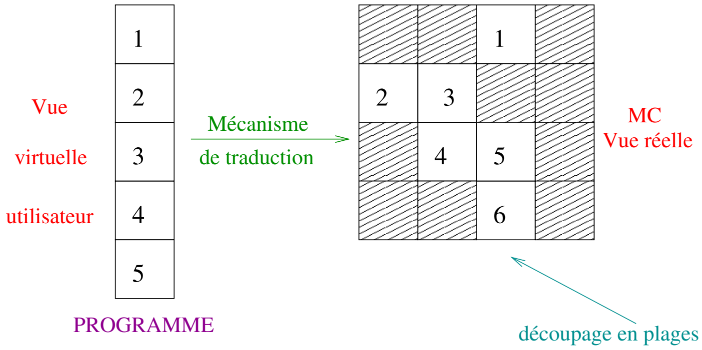
⋆ Accès mémoire :

Le processeur accède à la mémoire d'adresse a. Le fait que (rd) ≤ a ≤ (rf) assure la protection du système.

⋆ Adressage virtuel :

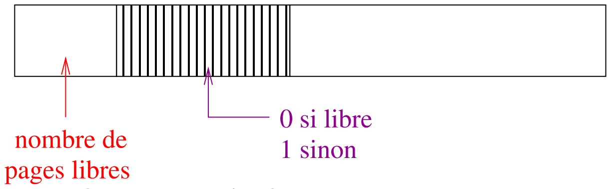
La méthode précédente présente des inconvénients et ce sont des plages continues de mémoire qui sont occupées (⇒ ensemble de trous)

Principe :



3.3.2 Phase d'allocation et chargement

⋆ Vecteur d'occupation de la mémoire centrale (VO) :



Soit un processus de n pages à charger.

Si n ≤ nombre pages libres (VO)

Alors allocation et chargement

Sinon échec, attente libération de pages

Pour allouer de la place pour des pages, on utilise le **vecteur de traduction** (VT). Ce vecteur contient les numéros des pages réelles, indicées par les numéros des pages virtuelles :

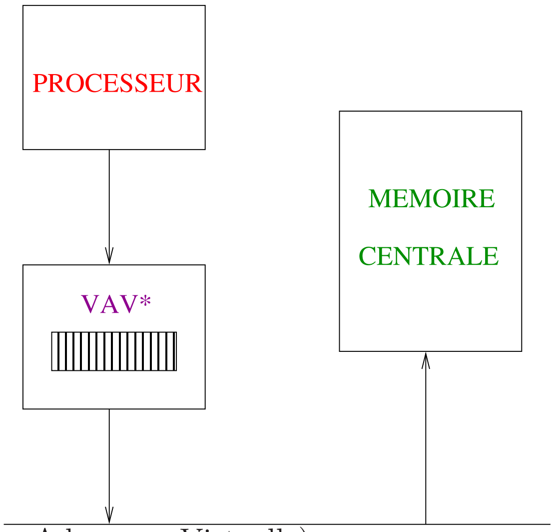
VT	3	5	6	10	11
	1	2	3	4	5

⋆ Mise à jour de VO

Il existe également une phase de désallocation des pages à la fin de chaque processus.

3.3.3 Phase d'exécution

⋆ Architecture



(*VAV : Vecteur Adressage Virtuelle)

⋆ Initialisation Processus P → X

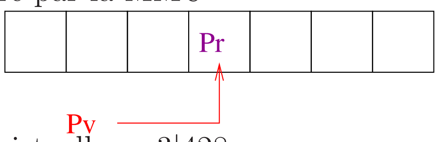
VT du processus envoyé vers le VAV

⋆ Accès mémoire $a_v = p_v \cdot m$

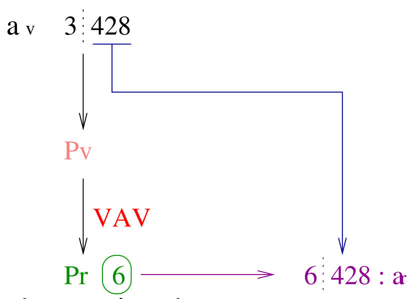
Exemple : page de 1K

Pour une page de 4K, il faut compter 12 bits (au lieu des 10 bits nécessaires précédemment)

⇒ Découpage assuré par la MMU



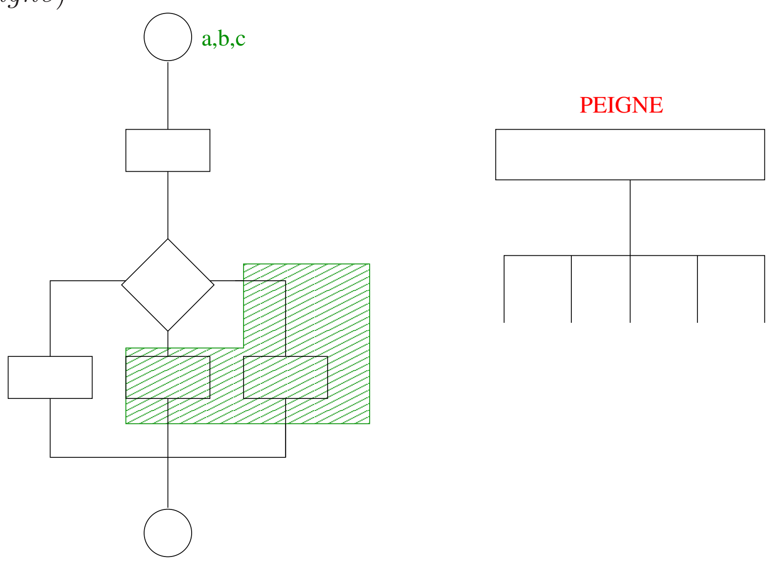
Exemple : adresse virtuelle a_v 3|428



⇒ Assure la **protection mémoire**

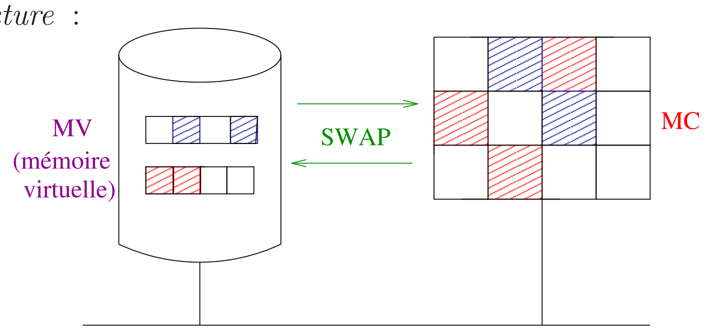
⋆ <u>Technique de mémoire virtuelle</u> :

- *Principe* : pour une exécution donnée (sur des données d'entrées), on ne parcourt que très partiellement le code du processus (technique du *peigne*)

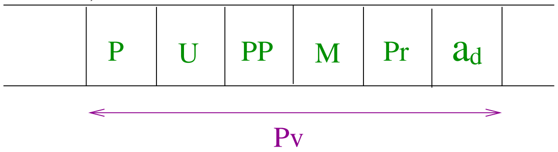


On utilise l'hypothèse que seul 10% du code est réellement utilisé pour un jeu de données en entrées.

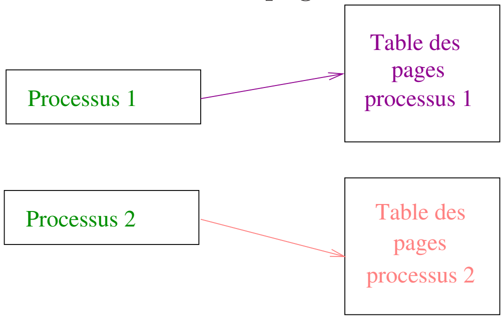
- *Architecture* :



- *Table des pages : MMU* On a déjà vu comment était créée l'adresse virtuelle ($a_v = p_v \cdot m$), avec :

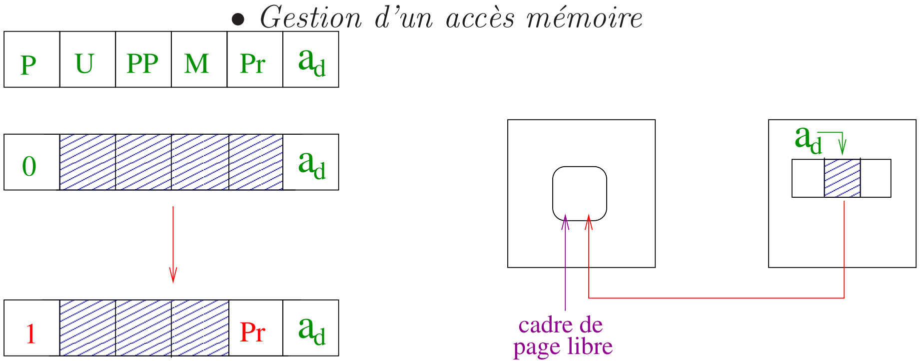


P $\longrightarrow$ page présente en mémoire centrale (1 ou 0)
U $\longrightarrow$ page utilisée (1 ou 0)
PP $\longrightarrow$ protection (1 ou 0)
M $\longrightarrow$ page modifiée (depuis changement) (1 ou 0)
P_r $\longrightarrow$ numéro de page réelle
a_d $\longrightarrow$ adresse disque (MV) de la page
Un processus accède à <u>**sa**</u> table de pages :



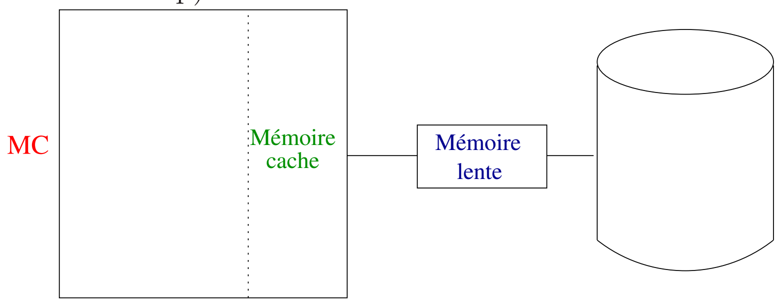
Un processus accède à la table des pages par le biais de PV.
<u>Si</u> P=1
<u>Alors</u> a_r : $p_r \cdot m$
<u>Sinon</u> défaut de page
        <u>Si</u> page libre (par p_r)
        <u>Alors</u> lire page en MV (par a_d) et la mettre dans le cadre p_r

- *Gestion d'un accès mémoire*



Pour libérer un cadre de page : mise de la page MC sur le disque (MV).
Se pose le problème de la page à la libérer $\Rightarrow$ on regarde les bits M

et U (M=1 et U=1 est le cas le plus favorable)
Remarque : Dans le cas des petites mémoires centrales, il y a risque d'effondrement du système
On cherche à optimisé les accès à la mémoire virtuelle ($\rightarrow$ diminution du nombre de swap) :



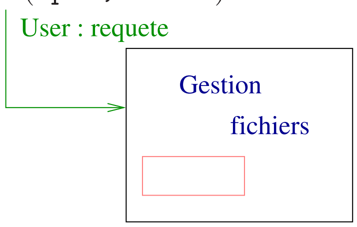
$\Rightarrow$ évite parfois l'accès au disque
$\Rightarrow$ optimise l'écriture sur le disque (plusieurs pages simultanément)

3.4 Gestion des fichiers

⋆ Position :

Applications	
Gestion fichiers	Gestion mémoires
Drivers	
...	
Contrôleur	

⋆ Organisation et primitive • *Primitives pour la gestion de fichiers* :
∘ création, suppression (`rm, ...`)
∘ ouverture, fermeture (`open, close`)



∘ opérateurs particuliers

• *Organisation et accès* :
∘ 2 types d'organisations : **séquentielle** (triée ou non) et **aléatoire**
∘ 2 types d'accès possibles : **séquentiel** ou **direct**

→ <u>Organisation séquentielle et accès séquentiel</u> :

Il apparaît deux problèmes : comment **ajouter** une fiche et comment **supprimer** une fiche (**tassement** (suppression physique) ou **bit de suppression** (suppression logique))

→ <u>Organisation séquentielle et accès direct</u> :

Ensemble de fiches avec **clé** et **accès par clé**

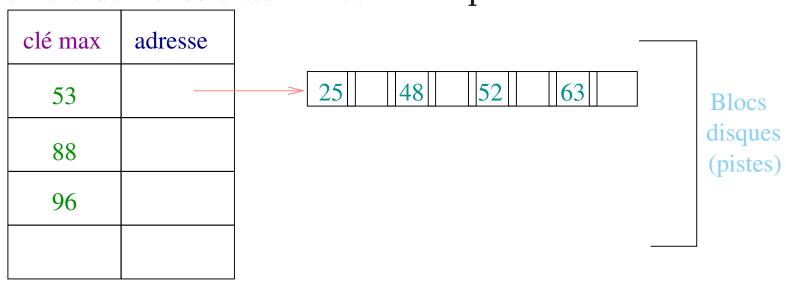


Table d'index

Pour modifier un fichier

- <u>ajout</u> : placement de la nouvelle fiche dans la position requise, décalage local, création de blocs de débordement
- <u>suppression</u> : suppression logique (bit de suppression) et fonction de réorganisation périodique

→ <u>Organisation aléatoire et accès direct</u> :

Avec table d'index :

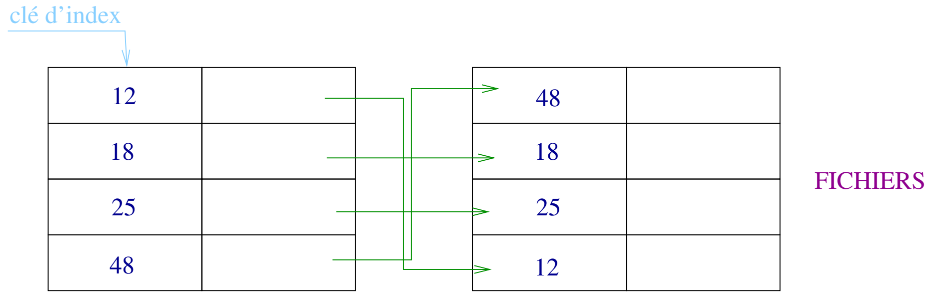


Table d'index

<u>Accès(clé)</u> : recherche la clé dans la table. L'adresse donne la position du fichier

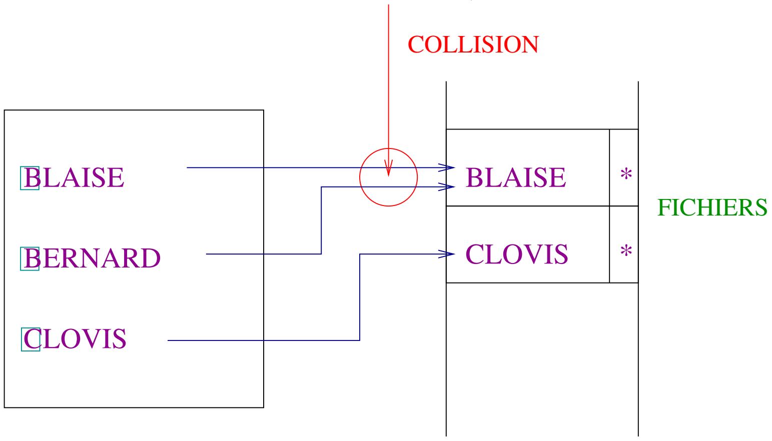
<u>Modification</u> : l'ajout se fait en fin de fichiers (réorganisation de la table d'index), la suppression est une suppression logique (à laquelle set adjointe une fonction de réorganisation périodique)

◇ Avec fonction de hachage (“*hash coding*”)

clé ↔ adresse

a. Fonction hachage identité

b. Par calcul “simpliste”
On peut utiliser, par exemple, la fonction de hachage qui prend le code ASCII du premier caractère (128 espaces de stockage)



Accès (clé)
↓
(fct hachage)
↓
Position
↓
Comparaison clé /
clé de la fiche
↓
Si ≠ → collision ?
↓
Recherche au plus près

⇒ Recherche d’une fonction de hachage telle que le nombre de collisions et le taux d’occupation de la mémoire soient minimaux.

c. Calculs sur la clé
Découpage de clé
On peut, par exemple, diviser la clé par la taille de l’espace de stockage. Le reste de cette division donnera alors la position

◇ Autre technique : fonction aléatoire
On a toujours le problème de la gestion des collisions
⇒ algo au plus près (et indication des collisions), algo de chaînage, algo de tirage aléatoire

⋆ <u>Organisation physique des informations d’un fichier</u> • *Stratégies* :
- ◦ en continu (mais obligation de fixer une taille maximum)
- ◦ en discontinu (utilisation du disque)

- *Techniques de blocs discontinus* :
  - fichier : ensemble de blocs
  - disque : *idem*
  - taille d'un bloc : quelques **kilo-octets**
  - gestion des blocs libres : allocation d'un bloc à un fichier utilisateur ⇒ NÉCESSITE de connaître les blocs libres

    a. Solution bit-map

    On utilise une **table binaire** à n positions (si le disque dispose de n blocs), 0 ou 1.

    b. Solution test de blocs libres

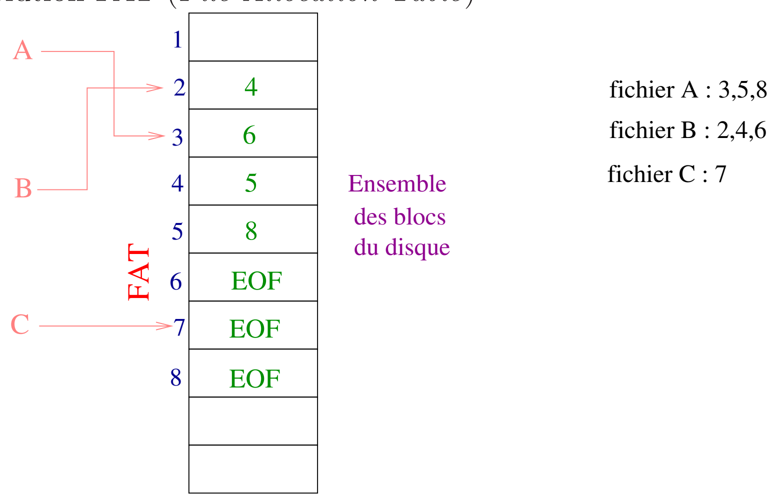
    Exemple : 10 M-blocs → liste de 10M positions

    ⇒ 40 M-octets (40 K-blocs) pour stocker **tous les blocs libres**

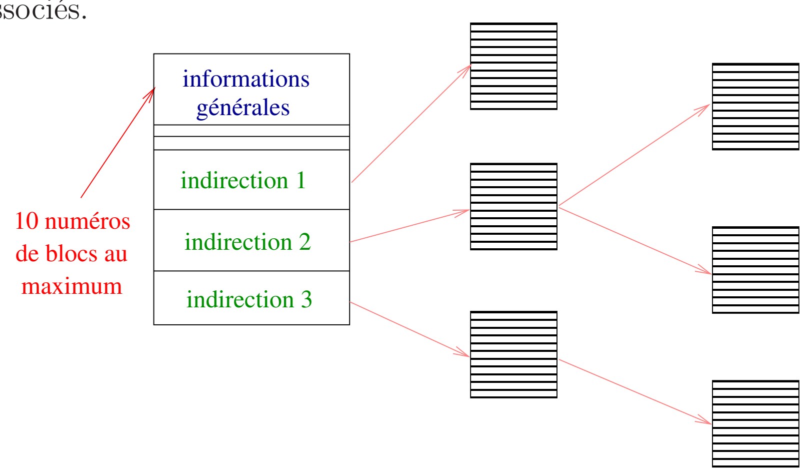
    Cette méthode est **beaucoup** plus rapide que la précédente.
  - Classification des blocs d'un fichier :

    (un fichier est un ensemble de blocs d'informations)

    a. Solution FAT (*File Allocation Table*)

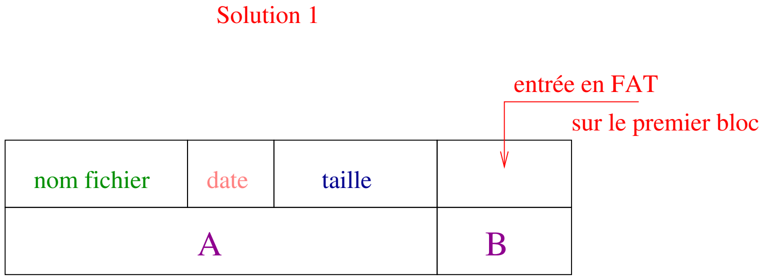


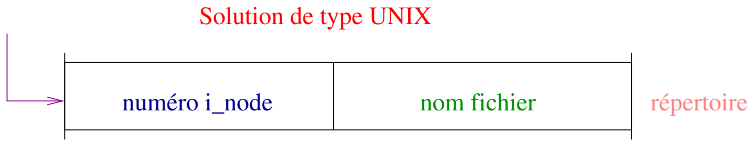
b. Solution de type UNIX
Fichier → e_node (dans un bloc) et infos générales, liste des blocs associés.



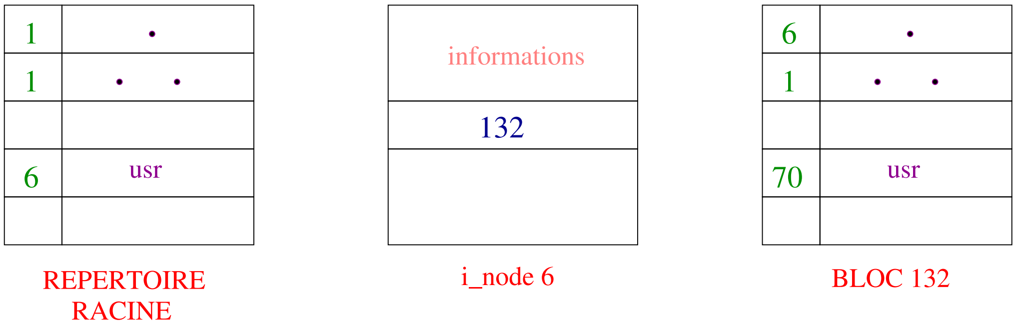
Exemple : bloc de taille 1K
indirection 0 → 10 blocs au max
indirection 1 → 10 + 256 blocs au max
indirection 2 → 10 + 256 + 256^2 blocs au max
⋮
Repérage des blocs d'un fichier à partir d'un répertoire

- fichiernom (`identfichier`) : ensemble des blocs associés au fichier
- répertoire : liste des fichiers



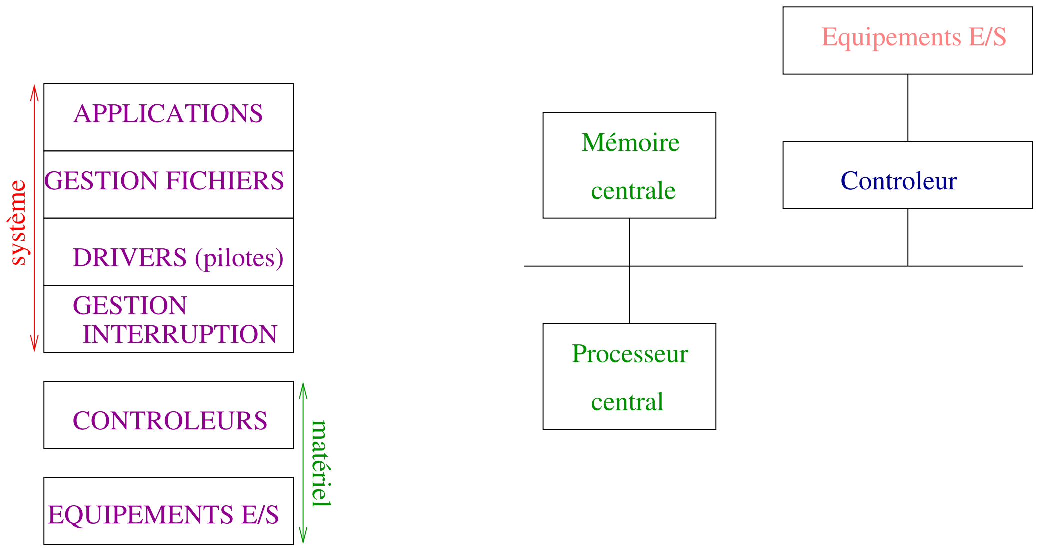


Exemple :



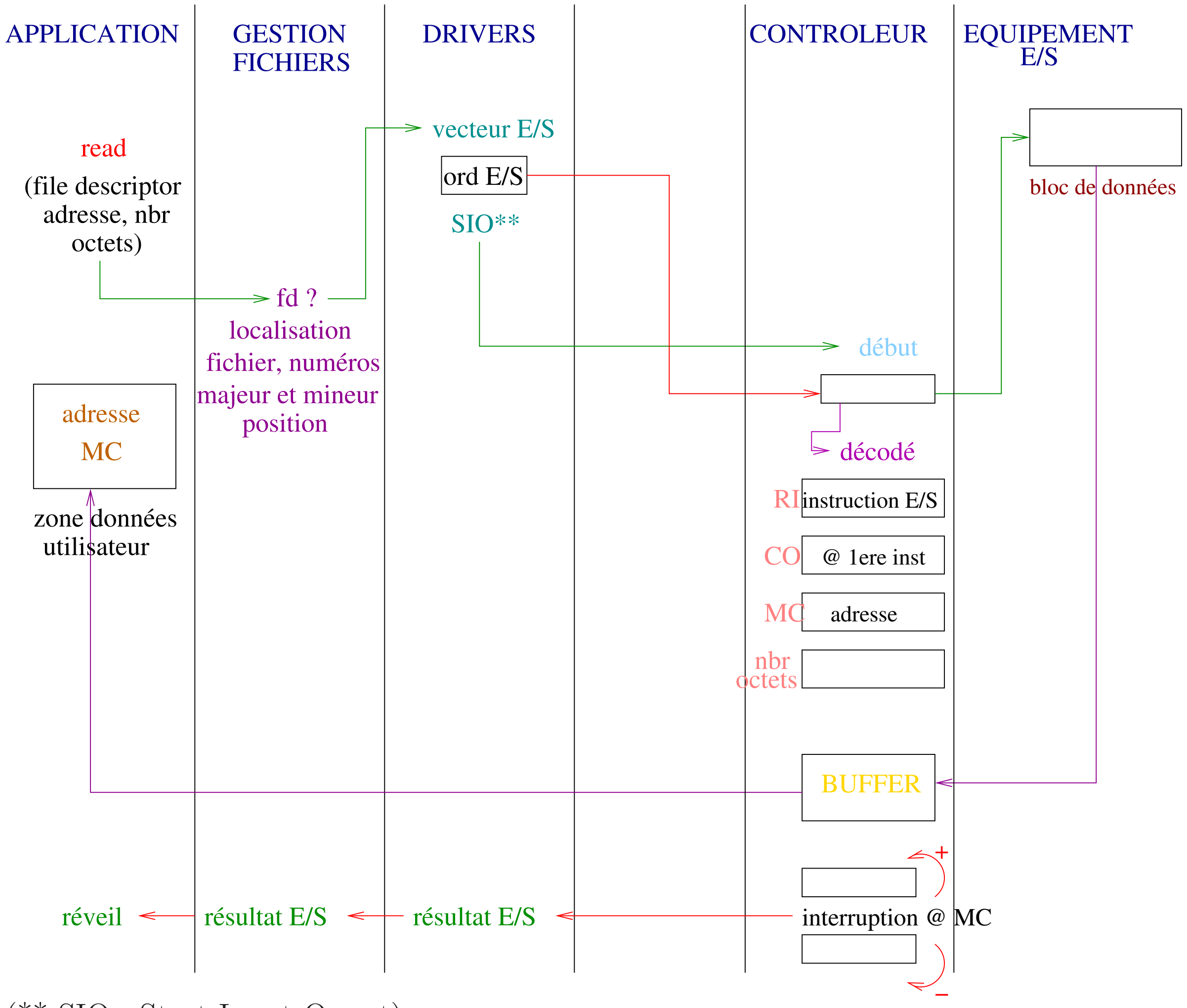
3.5 Gestion des entrées / sorties

⋆ Architecture :



- la gestion des fichiers reçoit une commande, une requête (`read` par exemple)
- le driver reçoit un message (de la gestion des fichiers) spécifique du type d'équipement d'entrée / sortie
- la gestion des interruptions reçoit du contrôleur des informations de fin d'utilisation de l'équipement E/S
- le contrôleur gère un ensemble d'équipement E/S de même type ($\Rightarrow$ processeur spécialisé)

⋆ Schéma d'exécution d'une entrée / sortie :



(** SIO : Start Input Ouput)

⋆ Forme générale d'un driver :

```
message mess ;
is_task ;
{ int r,caller ;
  initialize() ;
  while(true)
     { receive(any,&mess)
       caller=mess.source ;
       switch(mess.type)
          case read.r=do_read() ;break ;
          case write.r=do_write() ;break ;
       mess.type=task.reply ;
       mess.rep.status=r ;
       send(caller,&mess) ; }
  }
```

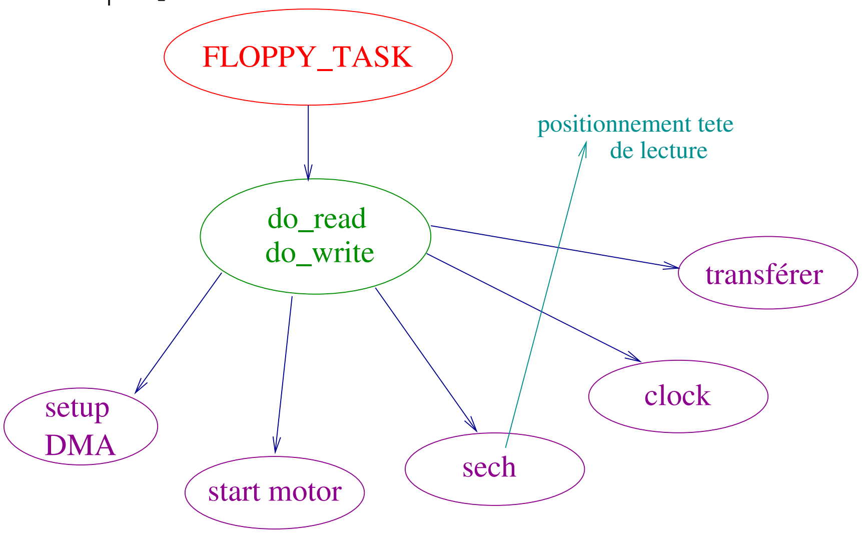
⋆ Exemple d'un driver (lecteur de disquette)

- *Lecture ou écriture d'un bloc de données* ($\simeq$ secteur)
- *Demande* (appel entrant)

message	`type`	opération demandée
	`device`	numéro mineur
	`position`	adresse sur la disquette
	`source`	appelant
	`adress`	adresse mémoire centrale où transférer les données
	`count`	nombre d'octets

- *Réponse* (appel sortant)

message	`type`	task_reply
	`proc_nr`	appelant
	`rep_status`	nombre d'octets transférés ou code erreur



Dès le lancement du `start motor` (processus parfois long), le driver s'"endort" (interruption) jusqu'à ce que le contrôleur reçoive l'information sur la vitesse (stabilisée) du périphérique.
En cas de non sollicitation du driver pendant un un certain laps de temps, le signal `clock` permet de stopper le `start motor`