# Système d'exploitation

Cours : Raymond Namyst
TD : A. Guermouche

Ecrit par : Marie-Luce ALLEE

27 novembre 2015

**Résumé**

***Objectif :***

— "Comment fonctionne un système d'exploitation ?"
— "Pourquoi ça marche comme ça ?"

***Notation :***

TP : NACHOS → 3 projets à rendre
Contrôle continu = $(\frac{Nachos1+Nachos2+Nachos3+DS}{4})$
Note final = MAX $(\frac{CC+EX}{2}$, EX)

***Biblio :***

"Système d'Exploitation" _ A. Tanenbaum
"Structure des système d'exploitation" _ A.Silberschatz
"Le noyau Linux" _ Bovet & Cesati

# Table des matières

# Contexte

A l'origine ce qu'on appelait un *système d'exploitation* consistait en un ensemble de routine résidant en mémoire.
(Aujourd'hui, on appelle ça une bibliothèque)
→ pilotes de périphériques

*Historiques de l'ordinateur :*
(1945 ∼ 55) - Un ordinateur est une salle entière
- *interface* : des interrupteurs puis des cartes perforées (∼ 50)
- *sortie* : au début des ampoules

(1955 ∼ 65) - arrivée des transistors
- arrivée des bandes magnétiques
(↪ permet de lire aléatoirement les données)
→ $1^e$ fois où on parle de "système"

Le "système" est une moniteur qui enchaine les travaux séquentielle-ment.

(1965 ∼..) - arrivé de la multiprogrammation (plusieurs travaux appartenant à des utilisateurs différents, en même temps en mémoire)
*Objectif* : recouvrir les temps d'entrées-sorties

*Exemple* de système multitâche : MULTICS (conçu par MIT, Bill, Général Electrics)
1969 : UNIX

# Chapitre 1

# Introduction aux systèmes d'exploitation

Un *système*, c'est une machine abstraite qui ajoute des services par rapport à un matériel.

- arbitrage de l'accès aux ressources
- virtualisation du matériel
- primitives de plus haut niveau que le matériel (fichiers, processus, espaces d'adressage,...)

**Question.** *Que se passe t-il au démarrage d'une machine ?*

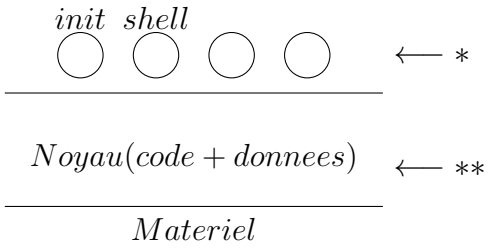
1. Le processeur exécute un programme contenue dans le "BIOS" (Basic Input Output System), code qui se trouve dans une puce mémoire non volatile → dans la mémoire morte (ROM) de la carte mère.
   Le *BIOS* contient le nom du périphérique où se trouve le "système d'exploitation"

   **Remarque.** *boot : est juste un gestionnaire de système*
2. Le BIOS charge le "système" en mémoire et l'exécute

   ↓

   le noyau du système

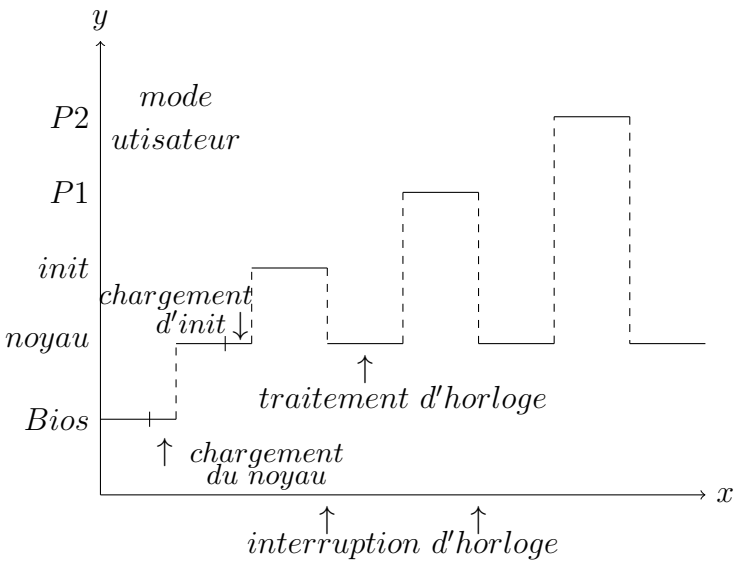
**Répresentation.**



**Légende :**
* : processus ($\simeq$ programme binaire en cours d'exécution)
** : Dans le Noyau, on y trouve :
- la gestion de processus
- la gestion de la mémoire
- les pilotes de périphériques

**Remarque.** *:* init *est le premier processus lancé. Il contient l liste des processus à lancer.*

**Démarrage & gestion des processus**




- Le traitement d'horloge est de très petite durée.
- Le noyau lors du traitement d'horloge choisit de redonner la main à init ou à un processus en mode utilisateur (ici P1 ou P2)

**Question.** *Que fait un processeur lorsqu'il reçoit une interruption ?*

1. les interruptions sont numérotées
2. l'interruption provoque une sauvegarde des registres et effectue un saut à une adresse qu'il récupère dans la table des vecteurs d'interruption

**Question.** *Que ne doit pas pouvoir faire un processus ?*

- il ne faut pas qu'ils puissent accéder à toute la mémoire
- il ne faut pas qu'ils aient accès à la totalité du jeu d'instruction du processus

Les processus fonctionnent selon (au moins) 2 modes :
- le mode noyau ("privilégié")

— le mode utilisateur ("protégé")
concernant le mode utilisateur, certaines instructions ne sont pas exécutables.
Il existe un bit dans un registre du processeur qui permet de configurer le mode

Lorsque le processeur tente d'exécuter une instruction illégale, une interruption est déclenchée (SIG ILL)
De plus, lors d'une interruption, le processeur passe en mode noyau.
Une interruption particulière est réservée aux processus pour lui permettre de basculer volontairement dans le noyau.
On appelle ça un appel système ($\simeq$ 300 sur les OS contemporains)

Pour indiquer l'appel demandé, le processus utilise un registre.

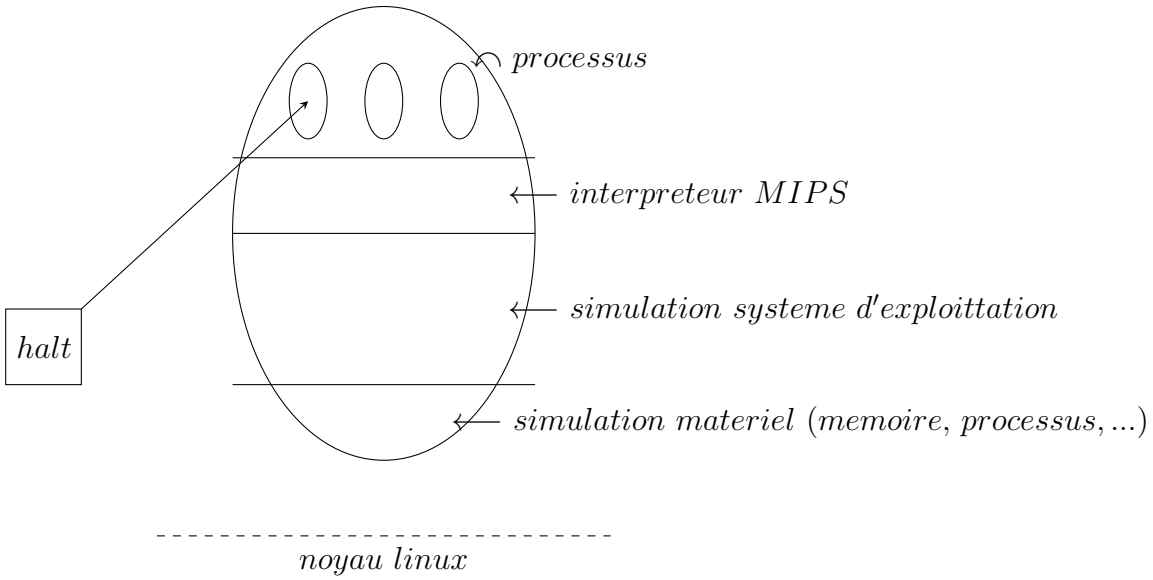
```
open()
{
 movl 5, %eax
 int 80h      --> passe en mode noyau
}
unistdh
```

# Chapitre 2

# NACHOS

## 2.1 NACHOS est un processus

*Représentation du processus NACHOS*




**Remarque.** *:*

- *L'interpréteur MIPS est la pour éviter de passer la main au noyau linux.*
- *halt est du binaire*
- *NACHOS est un processus récursif*

## 2.2 Répertoires principaux dans NACHOS

nachos/machine/ : processus pour le matériel

nachos/thread/ : processus pour le noyau

nachos/userprog/ : processus pour la mémoire

nachos/text/

**Description des répertoires plus en détail :**

- Répertoire thread

```
cd nachos/thread
./nachos
```

Résultat de ces commandes : le noyau est lancé puis un test se fait pour vérifier si le noyau marche puis le processus s'éteint

- Répertoire userprog

```
cd nachos/userprog
./nachos -x ../test/halt/   --> lance le programme halt sur Nachos

 main()
{
 Halt();
}
```

Que fait Halt() ? ?
Son code se trouve dans nachos/test/start.s (.s : c'est de l'assembleur)

```
Halt : addiu SC$\_$Halt, \$0, \$2     --> en IA-32 : mov SC$\_$Halt, r2
       syscall
       j 31            --> équivaut à un return
```

**Remarque.** *Assembleur MIPS a été choisit car c'est l'assembleur le plus simple.*

- Dans nachos/test/, on trouve tous les programmes utilisateurs

Ces programmes doivent donc être en MIPS, on utilise donc un compilateur "croisé" qui génère du code MIPS

- fonction ExceptionHandler() nachos/userprog/exception.cc a une fonction Exception-Handler()

```
ExceptionHandler()
{
regarde quel erreur, appel système est appelé
.
.
switch(n)
..
 case SC_Halt() := exit();
 case : <ERREUR>
..
}
```

- Pour modéliser la mémoire machine, un tableau est utilisé → mainMemory[..]

## 2.3 Représentation matériel de NACHOS :

— Machine
(registre MIPS + mémoire + ...)
— Interrupt
— Console (Entrée / Sortie)

- *Représentation du système dans NACHOS*

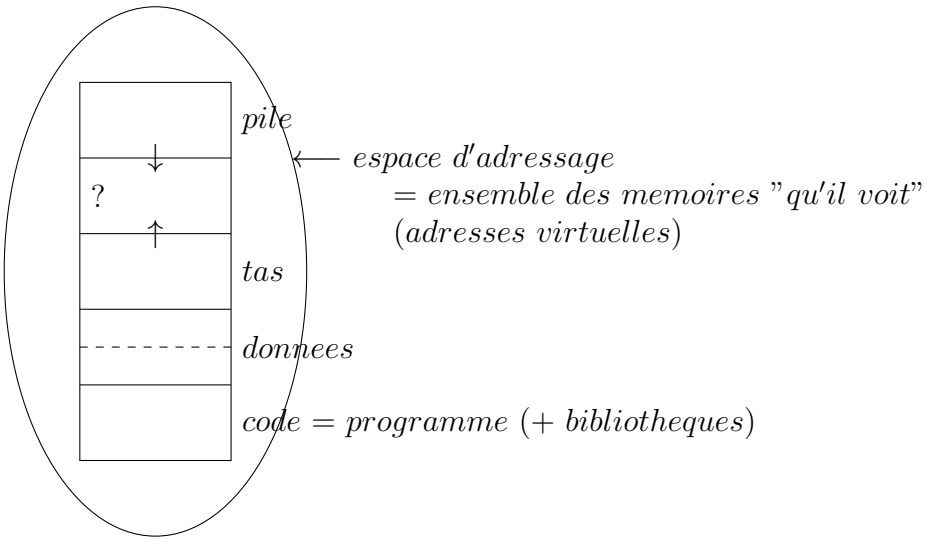
Nachos possède les classes suivantes :
— Thread
— Scheduler
Il maintient une liste de processeurs, c'est lui qui choisit qui prend la main après une interruption = ordonnancement
— Semaphore
Sert à la synchronisation
— AddrSpace
→ représente l'espace mémoire des processus

# Chapitre 3

# Gestion de processus

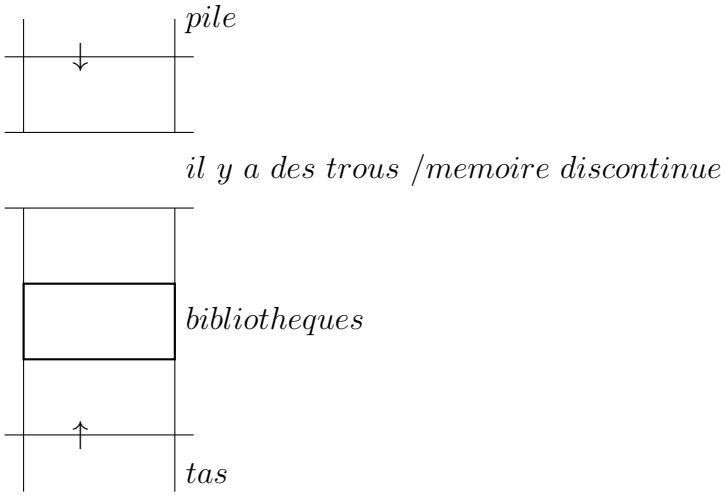
**Rappel.** *un processus est une instance de programme en cours d'exécution.*

**Représentation de l'espace d'adressage d'un processus :**



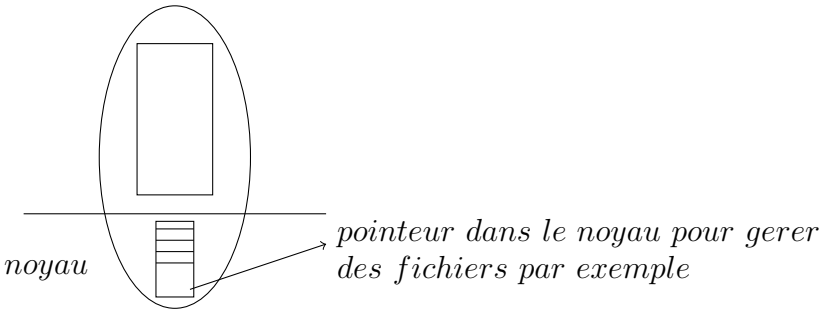

**Rappel.**
- *le code et une partie des données sont chargés en mémoire au démarrage du processus.*
- *les bibliothèques sont chargées que lors de leur première utilisation, c'est-à-dire pas forcément toutes au démarrage. (Raison : les bibliothèques sont de très gros fichiers).*
- *le tas est un segment de données.*

• Zoom sur la partie " ?" :
Comment est organisé cette partie ?

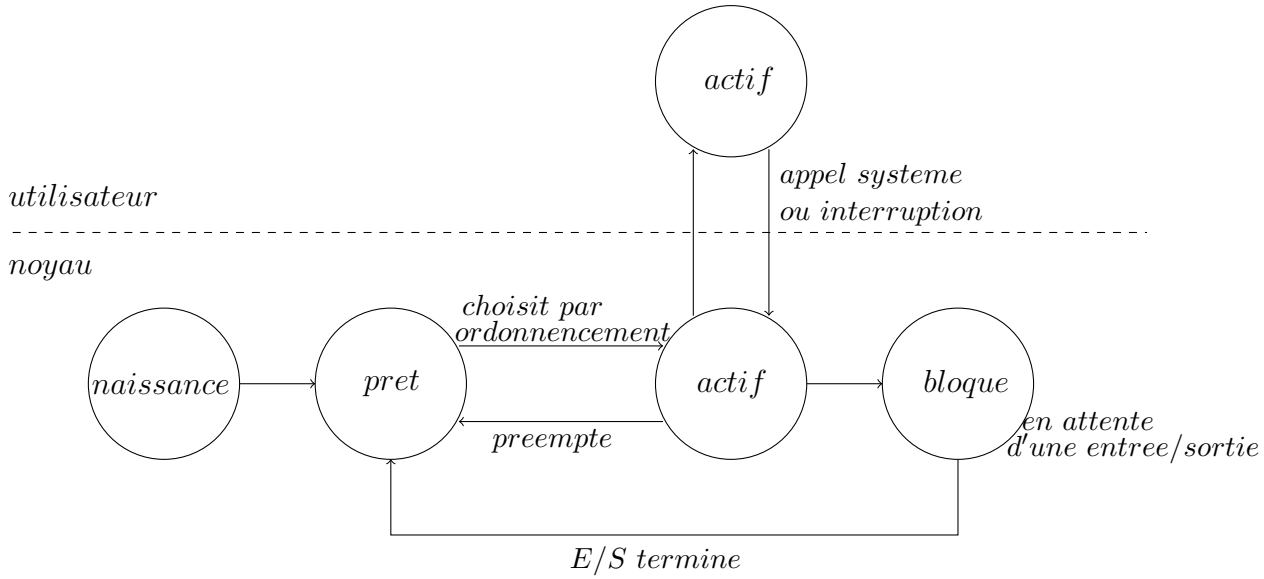



• Le noyau alloue une structure pour chaque processus dans laquelle il stocke :
— un pid : indice ds le tableau pour savoir ou il est
— une priorité
— une sauvegarde des registres
— un couple : utilisateur réel (celui qui a lancé le processus)/ utilisateur effectif
— une cartographie mémoire du processus (pour savoir une fois le processus fini ce qu'on peut libérer comme mémoire ou encore quelle taille maxime peut être donner à ce processus)

**Répresentation.**



## 3.1 Différents états d'un processus




**Remarque.**
— *Le nombre de processus actifs en même temps ≤ au nombre de cœur sur la machine.*
— *Même lors d'un appel système, le noyau peut décider de passer la main à un autre processus.*
— *Préempté : les registres du processus actif sont enregistrés puis le processus est remis en prêt.*
— *Exemple d'un processus bloqué : emacs*
*Comment sort un processus de l'état bloqué ?*

- *processus actif : gcc, processus bloqué : emacs*

1. *l'utilisateur tape sur une touche → interruption*
2. *gcc passe alors en noyau & réveille emacs qui passe en processus*
3. *le noyau fait l'échange entre emacs et gcc*

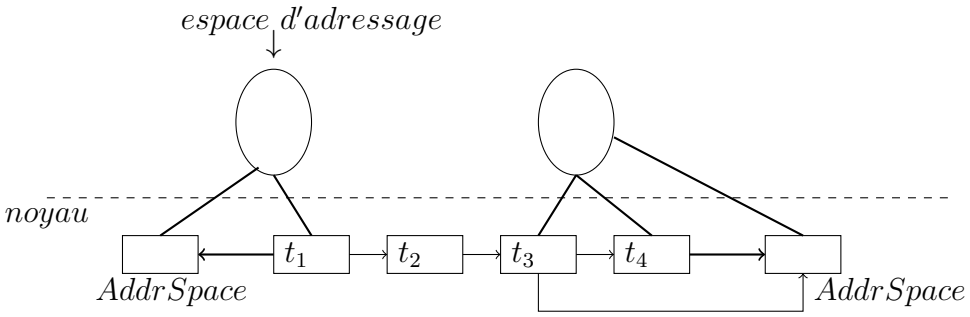
La transition entre l'état actif et l'état prêt s'appelle un *changement de contexte* (switch_to)

```
switch_to(Thread *current, Thread *next)
{

- sauver les registre du processus dans la structure current
- charge les valeurs de la structure next dans les registres

}
```

En réalité, dans un noyau tout est thread. Les processus n'existent plus (ils ont disparus dans les années 2000)
Les threads stockent la même chose qu'un processus, la cryptographie mémoire en moins.

Quelques différences :
— Il existe des threads qui n'agissent que dans le noyau, d'autres entre le noyau & dans le mode utilisateur.
— Deux threads peuvent avoir un même espace d'adressage.
— La mémoire est représentée dans AddrSpace.

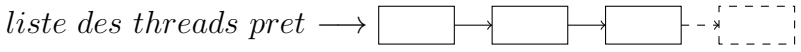
**Répresentation.**




## 3.2 Ordonnancement des processus

L'ordonnancement des processus est un compromis entre :
— maximiser l'occupation du processus
— le système doit être le plus interactif possible
— les calculs doivent être les plus performants possibles (ex : compilation rapide)

### 3.2.1 Ordonnancement le plus simple à mettre en œuvre : FIFO (First In, First Out

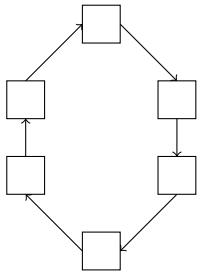



Lorsque le processus actif meurt ou se bloque, il est retiré de la liste et on passe au nouveau premier de la liste.
Les nouveaux processus sont rajoutés à la fin de la liste.
On trouve ce genre d'ordonnancement dans NACHOS (justement !) ou encore dans les JVM (Java Virtual Machine)

**Point négatif :** risque de famine.
Le premier processus peut faire une boucle sans fin par exemple.

### 3.2.2 FIFO + horloge qui provoque des changements de contexte = tourniquet (Round Robin)

**Répresentation.**



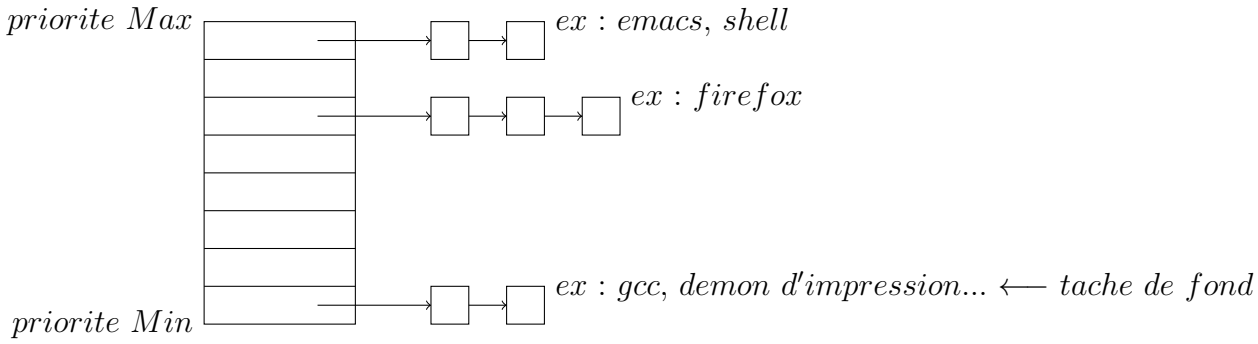
**Question.** *Est-ce que UNIX pourrait être comme ça ?*

Imaginons qu'il y ait 100 processus. Un des processus est Firefox (dont l'action est la lecture d'une vidéo), si on lui donne la main qu'après les 100 autres processus : il n'aura la main que 10ms toutes les secondes. Ce n'est bien évidement pas possible.

**Point positif :** plus de famine

**Point négatif :** pas d'interactivité

### 3.2.3 Algorithme avec file de priorité




Chaque processus a une priorité fixe.

⇒ utilisé pour les systèmes à temps-réel (ex : dispositif de pilotage)

Pour UNIX :
Il est difficile d'ordonnancer les processus. On ne sait pas le faire.

**Exemple.** *Firefox est embêtant, on ne sait pas ce qu'il va faire : google, vidéo (ce qui donne des priorités différentes).*

**Idée.** *Pour faire un système interactif, il faudrait pouvoir "modifier dynamiquement" les priorités.*

### 3.2.4 Algorithme utilisés au sein des systèmes de "traitement par lots" (batch)

**Objectif :** Minimiser le temps de restitution moyen
→ temps de restitution moyen = temps_de_fin - temps_de_soumission

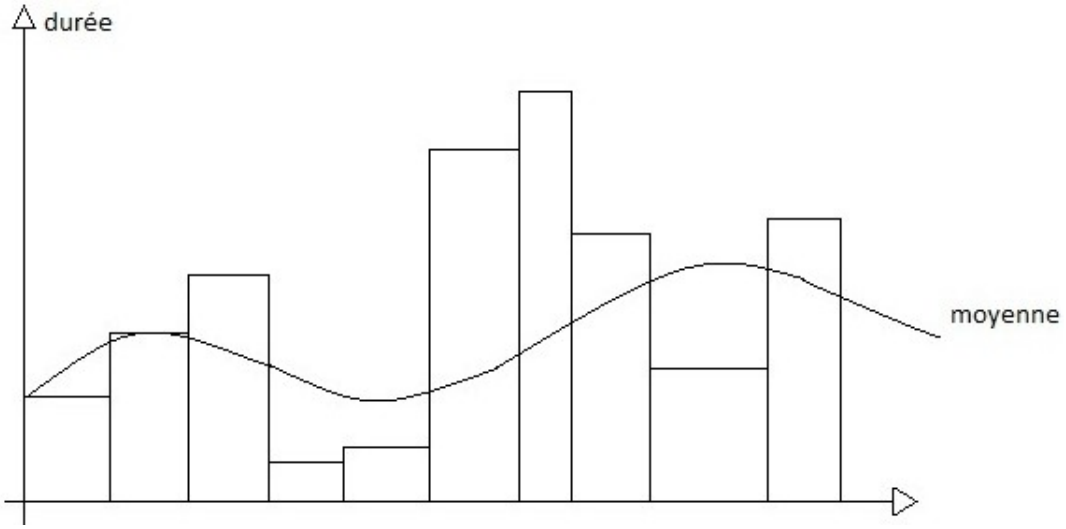
Supposons qu'on ait quatre taches : a,b,c,d
durée de taches : $t_a$, $t_b$, $t_c$, $t_d$
Si le système exécute a puis b, puis c, et enfin d, alors :

temps restitution(a) = $t_a$
temps restitution(b) = $t_a + t_b$
temps restitution(c) = $t_a + t_b + t_c$
temps restitution(d) = $t_a + t_b + t_c + t_d$
moyen = $\frac{4t_a+3t_b+2t_c+t_d}{4}$
↪ c'est minimal ssi $t_a < t_b < t_c < t_d$

**Idée.** *estimer la durée du prochain cycle d'utilisateur du CPU en s'appuyant sur le passé (une moyenne du passé)*

Estimation$_n$ = $\alpha$ . durée_observée$_{n-1}$ + (1-$\alpha$) Estimation$_{n-1}$

$\alpha = \frac{1}{2}$, E= $\frac{1}{2}$ . durée$_{n-1}$ + $\frac{1}{4}$ . durée$_{n-2}$ + $\frac{1}{8}$ . durée$_{n-3}$ + ...




**Idée.** *La priorité est inversement proportionnelle à l'estimation (on favorise les plus courts)* $/!\backslash$ *lorsque le processus n'est pas actif, il faut continuer à atténuer son estimation.*

**Exemple.** *Linux 2.4*

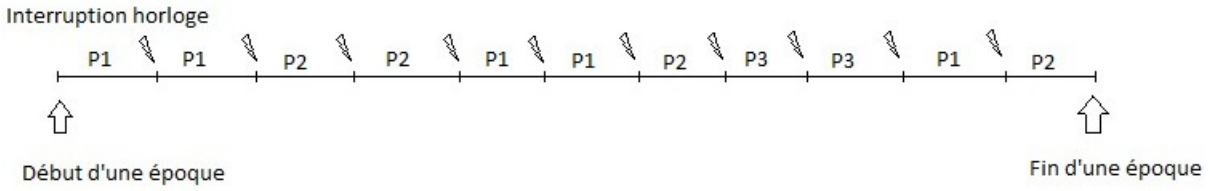
Sous UNIX, la priorité d'un processus comporte deux aspects :

1. priorité statique (nice)
2. priorité dynamique : dépend du comportement du processus

Sous Linux,la priorité statique donne droit à un certain nombre de quanta de temps durant une époque.

Données initiales :

| | |
|---|---|
| $P_1$ : 5 crédits $\rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow \emptyset$ | 5 |
| $P_2$ : 4 crédits $\rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow \emptyset$ | 4 |
| $P_3$ : 2 crédits $\rightarrow 1 \rightarrow \emptyset$ | 2 |

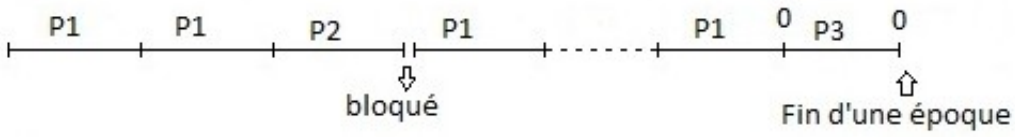



1. $P_1$ : 5$\rightarrow$ 4 : $P_1$ a droit à une deuxième tranche de temps.
2. $P_1$ : $4 \rightarrow 3$ & $P_2$ : 3 :Égalité des priorités entre $P_1$ et $P_2$. $P_1$ est choisit car certaines données sont déjà dans la mémoire cache.
3. ...

*Cas où P2 se bloque :*

Données initiales :

| | |
|---|---|
| $P_1$ : 5 crédits $\rightarrow 4 \rightarrow 3... \rightarrow \emptyset$ | 5 |
| ($P_2$) : 4 crédits | $4 + \frac{4}{2}$ |
| $P_3$ : 2 crédits ... $\rightarrow \emptyset$ | 2 |




A la fin d'une époque, on donne à chaque processus :

$$\frac{creditrestant}{2} + creditsinitiaux$$

$$\Rightarrow \frac{c + \frac{c+\frac{C}{2}}{2}}{2} + C \rightarrow c + \frac{C}{2} + \frac{C}{4} + ...2C$$

### 3.2.5 Système utilisé dans UNIX (av)

**Problème** : lorsque le nombre de processeurs augmente, l'algorithme devient cher. Chaque processeur exécute un goodness sur chaque prêt. On a donc un ensemble de données partagées qui doivent être accéder "un par un".
Si il y a plusieurs (milliers) de processeurs, ils ne peuvent pas scheduler en même temps.

$\Rightarrow$ il faut synchroniser

**Solution :** on casse la liste de processus en petits morceaux qu'on donne à chaque processeurs.
D'autres problèmes sont tout de même à gérer.

Il faut partager équitablement $\rightarrow$ la priorité moyenne doit être la même sur tous les processeurs
Il se peut aussi que toutes les taches soit finies sur un CPU. Dans ce cas il faut gérer les "voles" de taches aux autres processeurs tout en gardant une vision globale sur les priorités (même si moins précise que lorsqu'il y a un unique processeur).

## 3.3 Synchronisation

### 3.3.1 Préemption, concurrence, parallélisme

*Préemption :* le fait d'interrompre un thread/processus et de le retirer du processeur

*Concurrence :* l'exécution qui est "entrelacé" de plusieurs threads/processus

*Parallélisme :* lorsque plusieurs threads/processus progressent simultanément sur des processus différents

Regardons comment NACHOS fonctionne :

— préemption ?
./nachos -rs <seed> : permet de demander des interruptions d'horloge (<seed> pour créer un générateur de nombres aléatoires)
↪ essaye de créer un currentThread → Yield() ; (→ préemption)

Dans beaucoup de fonctions de NACHOS :

```
{
  int oldlevel = interrupt -> setLeveel(IntOff);  <- de manière indirecte,
   .                                                interrupt -> oneTick();
   .
   .
  interrupt ->setLevel(oldLevel);
}
```

Arrêt des interruptions d'horloge dans ce code.

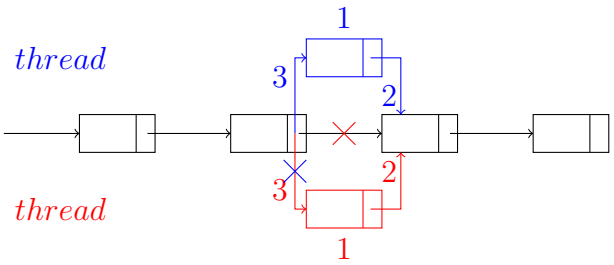
— concurrence : oui
— parallélisme : non (raison : déterminisme)

**Question.** *Quel est le principal problème posé par la concurrence entre threads ? (c'est-à-dire des processus qui partagent de la mémoire)*

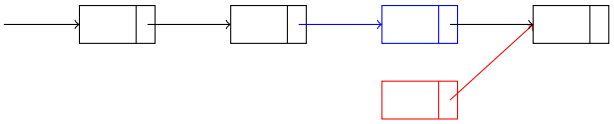
Lorsque les threads accèdent à une même donnée de manière concurrente (ex : malloc/free, malloc gère une liste de chainées).

**Exemple.** *manipulation de listes chainées*

Insertion de deux threads à peu prés en même temps :



Résultat :



→ les modifications de la liste ne doivent pas s'effectuer en même temps.

**Exemple.** *d'une instruction simple exécutée en même temps.*

```
                              int = 0;
thread 1                                    thread 2
for (i=0; i$<$100; i++)                     for (i=0; i$<$100; i++)
    n++;                                        n++;
```

```
                          peint n? 200 : 100
```

En C : n++, en assembleur $\begin{cases} load@n, r1 \\ incr1 \\ saver1, @n \end{cases}$ Si pas toujours synchrone on pourrait penser que 100< n < 200

| | thread 1 | thread 2 | |
|---|---|---|---|
| | lad r | | |
| n=1 | | | |
| | | lad r | ) |
| | | inc r1 | ) x99 |
| | | save r1 | )→ n=99 |
| n=1 | save r1 | | |
| | | load r1 | |
| | | inc r1 | |
| 99 x ( | | | |
| | | save r1 | n=2 |

$2 \leq n \leq 200$

⇒ même une instruction telle que "n++" doit être exécutée en prenant des précautions !
n++ n'est pas une instruction "atomique".
Une instruction *atomique* est une instruction qu'un processeur sait exécuter d'un seul tenant, c'est-à-dire qui est non interruptible ni "entrelaçable"

Pour résoudre ce problème, on crée une section critique.

### 3.3.2 Section critique

*Section critique :* portion de code qui ne doit être exécuter que par au plus un thread à la fois.

```
bool occupé = FALSE;

entrer_sc()         <------ le premier thread "passe"
{                            les autres attendent
 while(occupé == TRUE) ;
 occupé =TRUE;
}

sortir_sc()
{
 occupé = FALSE;
}
```

Problème si deux treads/processus passent `occupé` à TRUE en même temps, le problème n'est pas résolu.

- Solution (Peterson) pour deux threads

```
bool drapeau[2] = {FALSE,FALSE};
int tour = 0;

thread i
entrer_sc()
{
  drapeau[i] = TRUE;
  tour = i;
  attendre(drapeau[1-i] == FALSE ou tour == 1-i);  //1-i = "l'autre"
}

sortir_sc()
{
  drapeau[i] = FALSE;
}
```

La variable `tour` joue le rôle de l'arbitre et désigne qui est arrivé en dernier.

Généraliser "Peterson" à n processus est possible, mais c'est couteux et limité à un nombre de processus bornés.

→ les architectes ont introduit de nouvelles instructions dans le processus

**Exemple.** *l'instruction "test and set"*

```
int test_and_set(int *verrou)
{
 int old = *verrou;
 *verrou = 1;
 return old;
}
```

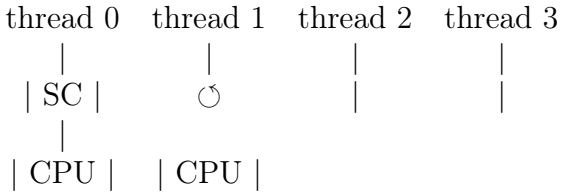
$\Longrightarrow$ test_and_set : instruction atomique

```
enter_sc()
{
  while(test_and_set(&verrou) == 1);
}

sortir_sc()
{
 verrou = 0;
}
```

**Amélioration :**

```
entrer_sc()
{
  while(test_and_set(&verrou) == 1)
     while(verrou == 1);
}
```

Mais il reste encore quelques problèmes.
Explication sur un exemple :




— pour le thread 1 : gaspillage d'un quantum de temps à chaque fois qu'il est ordonné.

— pour le thread 0 : le détendeur d'une section critique peut être préempté

Test_and_Set : instruction utilisée comme brique de base pour construire des schémas de synchronisation plus complexe

⊖ interférences avec l'ordonnancement

⊖ attente active

⊖ synchronisation plus complexe

### 3.3.3 Sémaphores et barrières

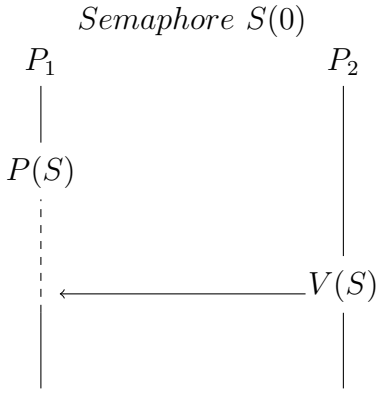
Dijsktra a proposé une autre solutions : des sémaphores.
Des sémaphores peuvent être vus comme des boites avec des jetons où le nombre de jetons $\geq 0$.

Les sémaphores ont 2 fonctions :
— Proberen (down) : puis-je ? en français
— Verhorgen (up) : Vas-y ! en français.

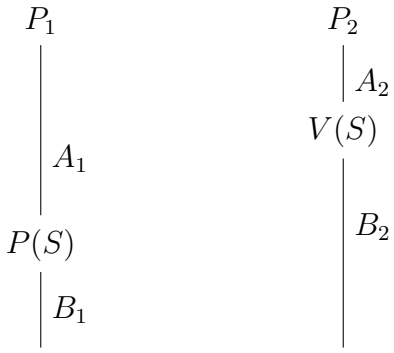
```
P(sem)
{
  sem -> value --;
  if(sem->value < 0)
     se bloque dans la liste sem -> liste
}

V(sem)
{
 sem->value++;
 if(sem->value <= 0)
    réveille un processus de la liste sem -> liste
}
```

`value` représente le nombre de jetons
Les fonctions P() et S() sont bien sur atomiques, sinon ça ne fonctionnerait pas.

**Exemple.** *1^e cas :*

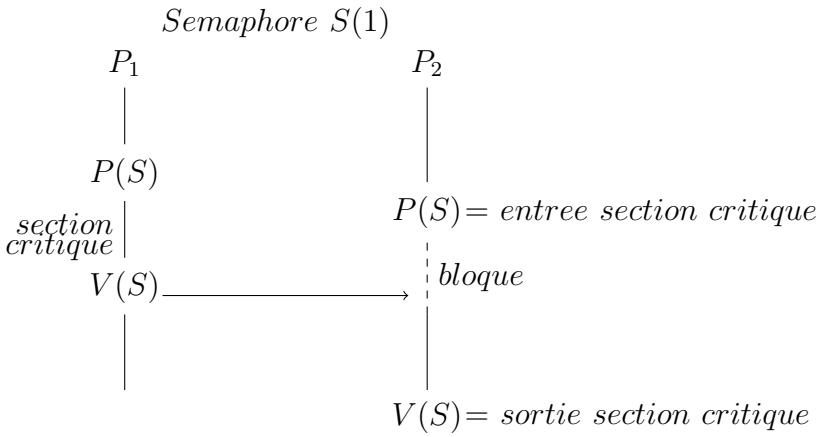



<u>2e cas :</u>




*L'opération par $P_1$ n'est pas bloquante.*
*$B_1$ débute toujours après la fin de $A_2$.*

<u>3e cas :</u>

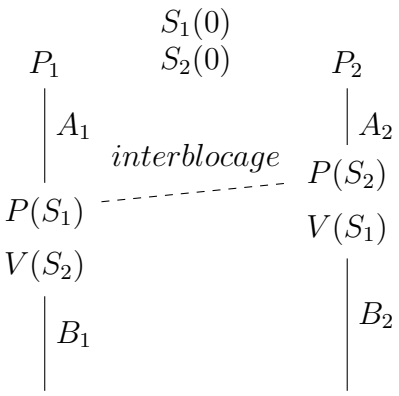



Par convention, on nomme ces sémaphores : mutex (Mutual Exclusion). Ils sont initialisés à 1 et cherchent "juste" à gérer les sections critiques.

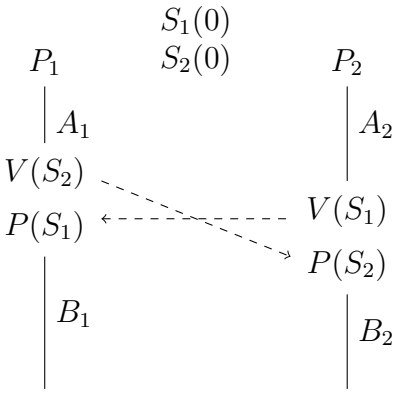
On veut $A_1 \ll B_2$ et $A_2 \ll B_1$.

**Idée.** *On utilise un sémaphore par processus.*

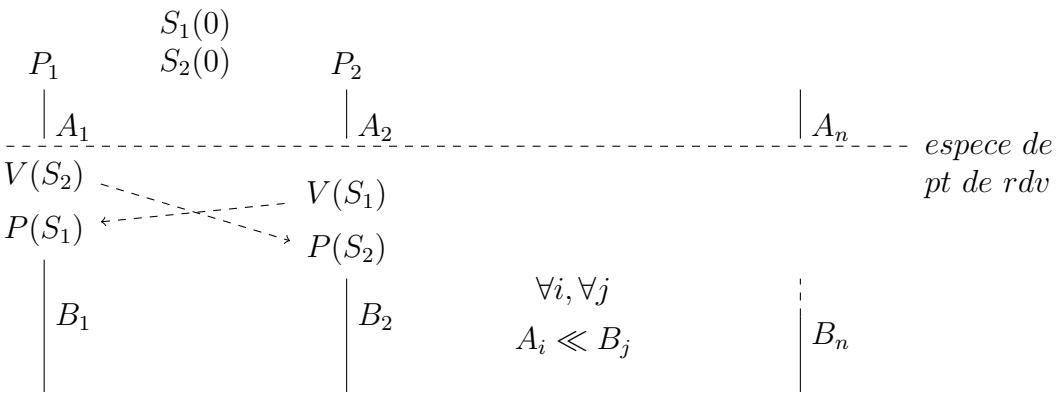
$1^e$ cas :




$\Rightarrow$ ne fonctionne pas. Interblocage.




Généralisation à n processus.



```
S[N] = {0, ..., 0};
Pi:
   |
   | Ai
   |
  for (j=0; j<N;j++)
    if(i!=j)
       V(S[j])
  for(j=0; j<N-1;j++)
     P(S[i]);
   |
   |
   |
```

Barrière de synchronisation : c'est un rendez vous où on attend que tous les processus aient atteint un certain point.

• On va maintenant essayer de voir si on peut utiliser un seul sémaphore pour bloquer les processus.

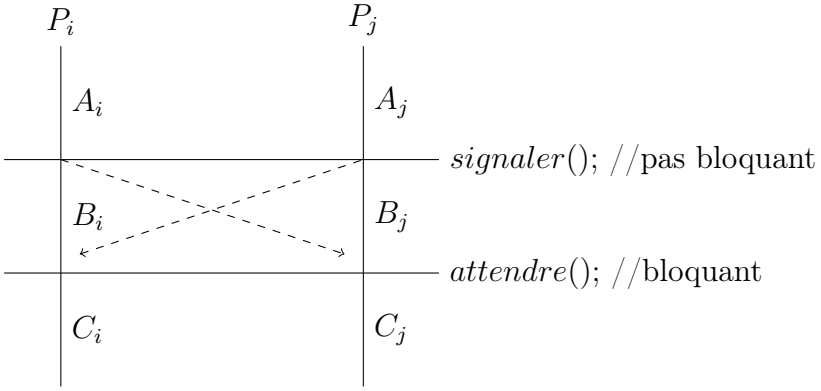
```
semaphores wait(0), mutex(1)
int n=0;

void barrière(int MAX)
{
 P(mutex);
 n++;
 V(mutex);
 if (n == MAX){
   for (i = 0; i < MAX; i++)
     V(wait);
   n = 0;
 }
 else
 {
   P(wait);
 }
}
```

V(mutex) ne peut pas être mis après le n++, car entre le moment où on incrémente n et le moment où on teste sa valeur, il peut se passer des choses.
On ne peut laisser passer que n - 1 éléments car on ne peut ajouter le dernier. En effet si on avait ajouté le dernier élément avec un code comme celui-ci :

```
semaphores wait(0), mutex(1)
int n = 0;

void barrière(int MAX)
{
 P(mutex);
 n++;
 if (n == MAX){
   V(mutex);
   for (i = 0; i < MAX - 1; i++)
     V(wait);
   n = 0;
 }
 else
 {
   V(mutex);
   P(wait);
 }
}
```

On ne peut pas le mettre à la fin de la fonction car P(wait) bloquerait et ne redonnerait jamais la main à V(mutex) et il y aurait donc un interblocage.

*Barrière en deux temps*




```
wait(0)
mutex(1)
int n=0;

signaler()                          |  attendre()
```

```
{                                   |  {
  P(mutex);                         |    P(wait);
  n++;                              |  }
  if(n==MAX)                        |
  {                                 |
    for(i=0; i>MAX; i++)            |
       V(wait);                     |
   n=0                              |
  }                                 |
  V(mutex);                         |
}                                   |
```

**Problème** : un processus peut manger le jeton d'un autre et passé une barrière en plus.

**Solution :**

**Idée.** *utiliser des données distinctes pour les barrières impairs et les barrières paires*

On peut définir une variable en utilisant le mot `thread` devant et dans ce cas, la variable est définie dans le thread en question.

Un cas très connu : `errno` qui indique s'il y a une erreur. Elle est définie dans chaque thread par `thread int errno` ce qui permet donc d'avoir un `errno` par programme.

Dans le cas de notre exemple, on déclare donc

```
thread int tour
semaphore mutex[2] = {1,1}, wait[2]={0,0}
int nb[2]={0,0}

void barrière(int MAX)
{
 P(mutex[tour]);
 nb[tour]++;

 if (nb[tour] == MAX)
 {
   V(mutex[tour]);
   for( i = 0; i < MAX; i++)
     V(wait[tour]);
   n=0;
 }

 else
```

```
 {
   V(mutex[tour]);
   P(wait[tour]);
 }

 tour = 1-tour;
}
```

On peut donc dans le programme ajouter `[tour]` dans toutes les variables utilisées. La variable tour étant « safe » car elle est propre au thread, on peut mettre à jour `tour` n'importe où : pas besoin de mutex. En d'autres termes, on peut mettre à jour la variable à la fin du else :

**Remarque.** *De nos jours les sémaphores ne sont pas toujours utilisés, on verra plus tard d'autres moyens.*

## 3.4 Quelques problèmes de synchronisation

**Exemple.** *Les philosophes : ces gens ne font que penser dormir et manger.*

*On considère un nombre n de ces gens là, et on se donne une table sur laquelle sont disposées n assiettes ainsi qu'une fourchette à la gauche de chaque assiette.*

*On remarque donc que cette disposition revient à dire que pour une assiette donnée, on a deux fourchettes à disposition.*

*Lorsque les philosophes se lèvent pour manger, ils ont besoin de 2 fourchettes. Comment optimiser le moment du repas à l'aide des notions présentées plus haut ?*

L'idée consistant à dire que dès qu'un philosophe arrive il commence par prendre la fourchette de gauche aboutit à quelques problèmes : si tous arrivent en même temps, personne ne mange.

Une solution pourrait être de bloquer le nombre de sages pouvant s'approcher de la table.

### 3.4.1 Tube/Pipe ([Shift]-[Alt]-[L] ou [AltGr]-[6])

#### Présentation

Un pipe peut s'apparenter à un tube dans lequel on pourra placer des données pour qu'un programme puisse les utiliser en attendant à l'autre bout du tube.

Nous allons raisonner comme suit : soit un tube que l'on peut remplir avec `put` et que l'on peut vider avec `get`. Le tube a une longueur maximale qui fait que le premier processus est bloqué si le tube est plein (`put` impossible), de même le second bloque si le tube est vide (`get` impossible).

Ci dessous sont donnés deux exemples typiques d'utilisation de pipe dans Unix.

**Exemple.** *Pour lister un répertoire et rechercher par exemple les fichiers* `pdf` *on utilise la commande suivante :*

```
ls | grep pdf
```

`ls` *va lister tous les fichiers et dossiers du répertoire pour ensuite les placer les uns après les autres dans le tube. En sortie de pipe,* `grep` *va rechercher le motif* `pdf` *dans chacun des éléments qu'il extrait à l'aide de* `get`

**Exemple.** *Dans le même esprit, on cherche à lister tous les réseaux wifi trouvés par la commande* `iwlist eth1 scan` *sachant qu'on ne veut que l'ESSID du réseau. La commande précédente donnant bien trop d'information, on va utiliser* `grep` *et* `sed` *pour arriver à nos fins.*

```
iwlist eth1 scan | grep ESSID | sed -e 's/^.*ESSID://g'
```

*Le premier pipe nous permet ici de passer la sortie de* `iwlist` *à* `grep` *qui va lire chaque ligne et placer dans un nouveau tube toutes celles contenant le motif ESSID.*

*Ensuite* `sed` *fait le remplacement et nous affiche les réseaux dans la sortie standard.*

### Le problème du Producteurs-Consommateurs (ou comment faire fonctionner un tube)

Le but est ici de construire à l'aide de sémaphores et de mutex les tubes ainsi que leur fonctionnement.

Un processus producteur (rempli le tube) va appeler une fonction `prod` et le consommateur (vide le tube) va appeler `cons` dont les squelettes sont donnés ci-dessous :

```
Prod(e)
{
      ....
      put(e);
   ....
}
```

```
Cons(*e)
{
      ....
      get(e);
   ....
}
```

On pourra commencer par utiliser un sémaphore `wait_cons(0)` (que l'on initialise à zéro), le consommateur ne pouvant pas faire `get` si le tube est vide. On peut donc se dire qu'un jeton équivaut à un élément et on aboutit donc au premier code suivant :

```
Prod(e)
{
    ....
    put(e);
    V(wait_cons);
}
```

```
Cons(*e)
{
    ....
    P(wait_cons);
    get(e);
    ....
}
```

Le point dont il a été question ici (ne pas tenter de sortir un élément alors que le tube est vide) est donc réglé puisque le code fonctionne : le nombre de jetons est le même que le nombre d'éléments dans le tube.

**Remarque.** *Attention : à un instant donné, on ne peut pas connaitre le nombre de jetons d'un sémaphore, juste savoir s'il est à zéro ou non...*

Maintenant, il faut aussi penser que le tube doit bloquer s'il est plein. On va donc ajouter un sémaphore qui va compter la place restante (par rapport à max : la taille du tube). D'où un sémaphore `wait_prod(Max)` (initialisé à MAX) et on peut compléter le code de la manière suivante :

```
Prod(e)
{
    P(wait_prod);
    put(e);
    V(wait_cons);
}
```

```
Cons(*e)
{
    P(wait_cons);
    get(e);
    V(wait_prod);
}
```

On a maintenant un nouveau problème : `put` et `get` peuvent-ils être effectués en même temps ? Cela demanderait un accès au même moment, de deux manières à la même structure (par exemple une liste).

Le code proposé ci-dessus ne fonctionne donc que si `get` et `put` sont supposées robustes. Il est possible que l'implémentation de `put` \`get` ne permette pas :

— un `get` et un `put` exécutés simultanément
— 2 `put` simultanément
— 2 `get` simultanément

Les deux derniers étant les cas les plus gênants. On va donc utiliser une section critique (avec un mutex) pour résoudre ce problème.

Tout d'abord, on ne peut pas faire fonctionner le code suivant car le producteur serait bloquer...

```
Prod(e)
{
    P(mutex);
    P(wait_prod);
    put(e);
    V(wait_cons);
    V(mutex);
}
```

```
Cons(*e)
{
    P(mutex);
    P(wait_cons);
    get(e);
    V(wait_prod);
    V(mutex);
}
```

pour ne pas être bloquant on fait :

```
Prod(e)
{
    P(wait_prod);
    P(mutex);
    put(e);
    V(mutex);
    V(wait_cons);
}
```

```
Cons(*e)
{
    P(wait_cons);
    P(mutex);
    get(e);
    V(mutex);
    V(wait_prod);
}
```

Et on résout ainsi les trois problèmes d'un seul coup. Si on veut juste supprimer le double `get/put`, on va utiliser deux mutex : `mutex_prod` et `mutex_cons` et on entoure comme précédemment.

```
Prod(e)
{
    P(wait_prod);
    P(mutex_prod);
    put(e);
    V(mutex_prod);
    V(wait_cons);
}
```

```
Cons(*e)
{
    P(wait_cons);
    P(mutex_cons);
    get(e);
    V(mutex_cons);
    V(wait_prod);
}
```

C'est de cette façon que sont gérer les tubes sous linux.

### 3.4.2 Lecteurs rédacteurs

On considère une structure de données partagées sur laquelle des threads peuvent :
— lire la structure ;
— modifier la structure et la lire.
Et le but est de gérer les lectures/écritures sur ces données.

La première idée est d'utiliser un mutex pour résoudre ce problème et donc ajouter une section critique à chaque fois que nécessaire. Mais il existe des actions que l'on peut autoriser en accès simultané : les lectures.

Par contre on interdit :
— les écritures simultanées ;
— les écritures et lectures simultanées.

#### Solution

On se donne les squelettes suivants :

```
lecteur()
{....
    lire();
....}
```

```
redacteur()
{....
    ecrire();
....}
```

On suppose que lire et écrire ne sont pas deux fonctions protégées (au sens vu dans la sections précédente).

On va commencer par se donner un mutex `mutex_redac(1)`.

On veut le placer dans `redacteur()` en encadrant à `ecrire()` par `P(mutex_redac)` et `V(mutex_redac)`, et le problème de l'écriture est résolu.

Il faut maintenant gérer le problème de `lire()`. On va essayer de réunir toutes les lectures en même temps pour obtenir « une seule section critique ». On va donc trouver

qui est la première lecture, cette dernière bloquera toute les autres et lorsqu'un droit de lecture sera donné, toute les demandes en attente seront traitées.

On ajoute donc `int nb=0` qui nous donne le rang de la lecture. On va quand même ajouter un mutex pour ne pas incrémenter la variable `nb` en même temps. On crée le mutex : `mutex_lect(1)`.

Le jeton `mutex_redac` ne devra être rendu que lorsque la dernière lecture sera effectuée. On va simplement ajouter un test pour savoir si on est bien la dernière lecture à l'aide de la variable `nb`.

Le mutex `mutex_lect` doit être relâcher assez tard car le premier processus demandant une lecture est considéré comme un éclaireur. On doit donc bloquer « tout le monde » (comprendre les demandes de lecture) tant que l'éclaireur ne peut pas lire. Donc en relâchant le mutex avec `V(mutex_lect)`, on va libérer tous les mutex bloqués en cascade.

Donc tant que `nb` est non nul, on aura toujours quelqu'un qui utilise `lire()`...On obtient donc :

```
lecteur()
{
    P(mutex_lect);
    nb++;

    // eclaireur
    if (nb == 1)
    {
      P(mutex_redac);
    }

    V(mutex_lect)

    lire(); // on profite de notre droit

    P(mutex_lect);

    nb--;

    if (nb == 0){
        V(mutex_redac);
    }

    V(mutex_lect);
}
```

Ceci fonctionne, mais apriori, les lecteurs peuvent garder la main indéfiniment et les nouveaux rédacteurs ne pourront jamais faire leur travail.

Notre problème de synchronisation est résolu mais n'est pas du tout équitable en particulier pour les rédacteurs.

On va maintenant créer un nouveau sémaphore : `sas` qui va jouer le rôle d'entonnoir. L'idée principale ici est que les rédacteurs ne doivent « pas perdre leur place », en particulier ne pas se faire doubler par une lecture plus récente qu'eux. On fait donc :

```
lecteur()
{
    P(sas);
    P(mutex_lect);
    nb++;

    // Eclaireur
    if (nb == 1)
    {
        P(mutex_redac);
    }

    V(mutex_lect);

    V(sas);

    // on profite de notre droit
    lire();

    P(mutex_lect);
    nb--;

    // On termine avec l'éclaireur
    if (nb == 0){
        V(mutex_redac);
    }
    V(mutex_lect);
}
```

```
redacteur()
{
    P(sas);
    P(mutex_redac);
    ecrire();
    V(sas);
    P(mutex_redac);
}
```

## 3.5 Moniteur de synchronisation (Hoare)

Les premiers moniteurs se définissaient de la manière suivante en `simula` :

```
Begin moniteur "m"
    int variable,...;
    procedure p1
        variable++;
    procedure p2
```

```
        ...
end
```

où les procédures s'effectuent en exclusion mutuelle.

Sous Unix, l'utilisation des moniteurs s'effectue au travers d'une bibliothèque. C'est un type que l'on appelle `mutex_t` que l'on utilise avec : `mutex m;`.

On fait `mutex_lock(&m);` pour entrer en section critique, ce qui ressemble à un `P(mutex)` et on sort de la section critique par `mutex_unclock(&m)` qui correspond à `V(mutex)`.

Les synchronisations bloquantes s'effectuent au moyen d'objets différents qui sont des « conditions ».

Aprés avoir déclaré notre moniteur, on fait `cond_t c;` pour ajouter des conditions.

**Exemple.** *Un processus entre en section critique et se bloque :*

```
...
mutex_lock(&m);
...
if (...)
    cond_wait(&c,&m);
...
mutex_unlock(&m)
```

*Avec un* `wait`*, on est toujours bloqué, et comme la condition est utilisée avec un moniteur* `m`*, on doit le passer en argument.*

*Dans notre cas, le processus se bloque et sort de la section critique (temporairement).*

*Un jour, quelqu'un va débloquer le* `wait` *(chose qui sera vu plus loin) et on exécute le code qui suit. Par contre il faudra avoir ré-acquis le verrou pour pouvoir continuer l'exécution.*

Généralement, le code sera encadré par des `_lock()` et `_unlock()`.

Le réveil de processus s'effectue au moyen de :

— `cond_signal(&c)` qui réveille un processus s'il y en a au moins 1 bloqué, sinon rien ne se passe ;

— `cond_bcast(&c)` qui réveille tous les processus bloqués sur `c`.

Pour faire une barrière, on fait

```
int nb =0;
mutex m;
cond c;
barriere()
{
    mutex_lock(&m);
    nb++;
```

```
    if (nb == MAX)
    {
        nb = 0;
        cond_bcast(&c);
    }
    else
    {
        cond_wait(&c,&m);
    }

    mutex_unlock(&m);
}
```

```
   signaler()
   {
    mutex_lock(&m);
    nb++;

    if(nb == MAX)
      cond_bcast(&c);

    mutex_unlock(&m);
   }
```

```
attendre()
{
 mutex_lock(&m);

 if(nb < MAX)
  cond_wait(&c,&m);

 mutex_unlock(&m);
}
```

Ce code marche mais pas si appeler plusieurs fois. (comme pour les sémaphores).

Producteur/Consommateur rappel : cas d'un tube de taille MAX à deux fonctions put() et get(). But : synchroniser les fonctions put() et get().

```
   mutex m;
cond prod, cons;
int nb = 0;
produire()
{
  mutex_lock(&m);
  if(nb == MAX)
    cond_wait(&prod,&m);
  put();
  nb ++;
  cond_signal(&cons);
  mutex_unlock(&m);
}
```

```
Consommer()
{
 mutex_lock(&m);
 if (nb == 0)
    cond_wait(&cons, &m);
 get();
 nb--;
 cond_signal(&prod);
 mutex_unlock(&m);
}
```

Ce code marche s'il y a qu'un seul consommateur, un seul producteur. Problème avec cette implémentation, un premier consommateur arrive à `cond_wait(&cons,&m)` et se bloque, un deuxième consommateur arrive au même au moment que le producteur fait un `put()`. Le deuxième consommateur double le premier, fait un `get()`. Or le producteur exécute un

`cond_signal(&cons)`, le premier consommateur se libère et va exécuter son `get()`. Mais pas possible plus d'élément ! => Gros Problème.

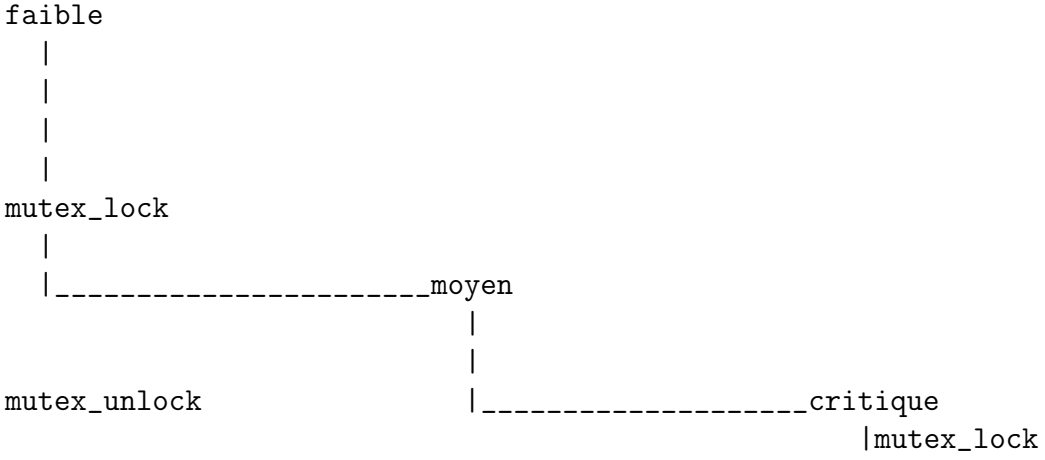
**Solution :** on rajoute un while.

```
produire()
{
  mutex_lock(&m);
  while(nb == MAX)
    cond_wait(&prod,&m);
  put();
   ...
}
```

```
Consommer()
{
 mutex_lock(&m);
 while (nb == 0)
    cond_wait(&cons, &m);
 get();
 ...
}
```

On s'aperçoit avec ce problème que les moniteurs ne sont pas si faciles à utiliser. Cependant ils sont plus souvent utiliser que les sémaphores. Les moniteurs sont préférés parce qu'ils sont plus clairs, on voit mieux les choses qu'avec des sémaphores.

1997, Robot PathFinder
Plusieurs exécutions de trois types : *Faible*, *Moyen* et *Critique* utilisent de la mémoire partagée
Il est périodique de durée courte.
S'il y a pas d'exécution pendant longtemps on reboot.




Le problème il peut avoir des inversions de priorité
**Solution :** l'héritage de priorité c'est à dire le détenteur du mutex voit sa priorité réajustée au processus le plus prioritaire qui attend

`mutex_unlock` ne fonctionne que si on détient le verrou.

**Exemple.**

```
void g()
{
 mutex_lock(&m);



 mutex_unlock(&m);
}
```

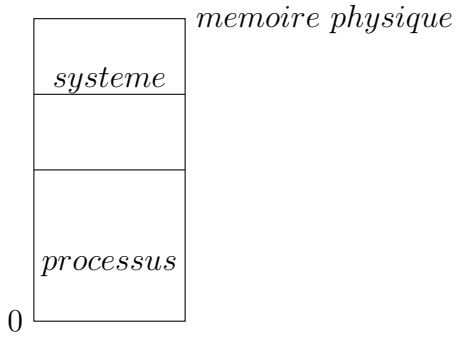
```
f()
{
 mutex_lock(&m);

 g();

 mutex_unlock(&m);
}
```

Le `mutex_lock` dans g() => bloque
Dans les vrais systèmes, une option permet aux mutexs de fonctionner en cas d'appels récursifs.

# Chapitre 4

# Gestion de mémoire

A l'origine, les ordinateurs exécutaient un seul processus à la fois.



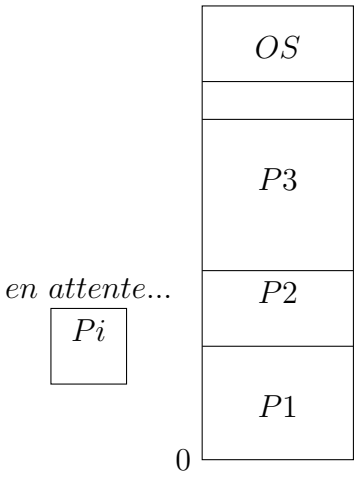

Exemple de système qui gère un seul processus : MS-DOS

**Question.** *Pourquoi des gens ont voulus mettre plusieurs processus sur une machine alors que ça marchait bien avec un seul processus ?*

$\longrightarrow$ rentabilité du système.

p : probabilité de faire une E/S dans un processus
proba que le processus calcule : 1-p
si n processus se trouvent simultanément en mémoire : 1-$P^n$.

**Idée.** *(depuis le système IBM OS/360) :*
*faire cohabiter plusieurs processus simultanément.*




```
i :
mov @i, %ax
jmp suite
```

**Problème :** le compilateur ne connait pas l'adresse à laquelle le processus va être chargé

**Une solution :** utiliser des partitions, fixer de la mémoire
Le compilateur détermine à l'avance la partition dans laquelle le processus sera chargé.
De nouveau un **Problème :** gaspillage
De plus, on ne sait pas "à l'avance" (= compilation) à quel endroit va charger un processus.

Le compilateur suppose que le processus est chargé à l'adresse 0.
→ Le compilateur joint une liste d'emplacement dans le code binaire qu'il faut translater.
→ L'OS fait un travail qui peut être long
→ / !\ Accès mémoire en dehors du processus.
La vérification à la compilation est impossible dès qu'il y a des choses comme des tableaux ou des pointeurs.

**Idée.** *insérer des instructions supplémentaires pour vérifier que les accès de mémoire sont corrects.*

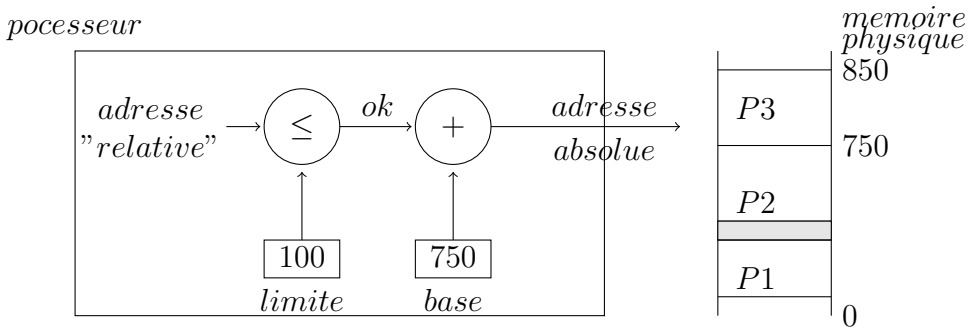
→ effectuer par le compilateur. (langage Ada : langage qui vérifie chaque accès de mémoire)
**Inconvénient :** Très couteux. (Pour chaque écriture : deux comparaisons en haut de la mémoire et en dessous)

Évolution des processeurs : ajout de deux registres spéciaux :
— base
— limite

Le compilateur génère des "adresse relatives" (= il suppose que le processus est chargé à l'adresse 0).
L'OS positionne les registres limite et base.

Lors d'un changement de contexte,on sauvegarde les registres du processus. Pareil pour ces nouveaux registres, ils sont enregistrés quand on passe la main à un autre processus.

Fonctionnement :



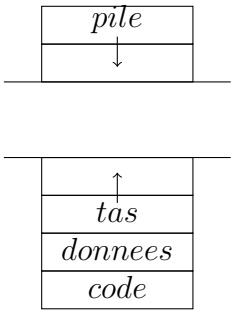

**Bonus :**
⊕ les processus peuvent être placés à n'importe quelle adresse
⊕ un processus ne peut pas faire d'accès en dehors de son espace d'adressage
Malgré tout il risque encore des problèmes. En effet, il reste encore des trous dans la pile de mémoire.

Lorsqu'on veut allouer un processus, on alloue sur le plus grand trou. Contrairement à la première pensée qui d'insérer dans le trou le plus proche de la taille du processus. Si ça fonctionnait de cette façon ça créerait un nouveau trou encore plus petit qui sera lui inutilisable plus tard car trop petit.

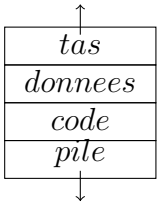
**Question.** *Mais que ce passe-t-il quand tous les trous sont de tailles inférieurs au processus ?*

Une solution est de déplacer la mémoire de chaque processus de façon contigüe pour ne créer qu'un unique bloc de mémoire.
→ ce qui coute chère dans cette solution, c'est la copie de mémoire Du point de vue des processus, seul le registre de base doit être réajusté pour chaque processus.
Cette méthode était utilisée dans le passé mais désormais on ne fait plus ce genre de choses.

De plus, une supposition a été faite jusqu'à maintenant qui n'est pas vrai. En effet, le processus n'est pas de taille fixe.
De nos jours, il y a deux morceaux du processus qui augmentent : le tas et la pile.




Le schéma moderne n'est pas possible pour le moment on ne peut pas se permettre d'avoir des bornes. Puisqu'on a pas assez de mémoire.
**Une solution :** pour éviter de borner la pile de processus :




Il reste malgré tout une question.

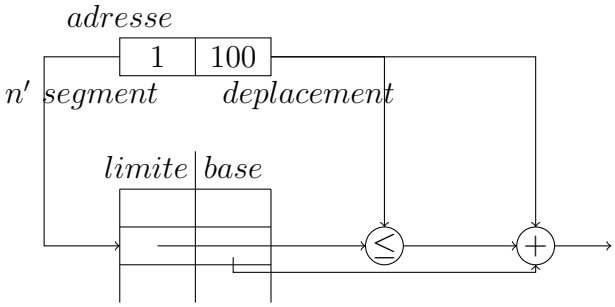
**Question.** *Comment gérer la croissance de l'espace d'adressage ?*

**Une solution :** morceler l'espace d'adressage en plusieurs parties logiques : pile, tas et code + données. En effet de petits morceaux sont plus faciles à insérer qu'un gros bloc. Mais le processeur n'est pas encore fait pour une telle solution.
Pour gérer ça, il est nécessaire que chaque bout des processus est un registre limite et un registre base.

Les adresses sont maintenant de la forme : numéro de segment, adresse relative à ce segment.
Une table est utilisée pour stockés ces adresses de segments.

Sur les processeur x86, les segments sont nommés :
— CS "code segment"
— DS "data segment"
— SS "stack segment"
— ES "extra segment"

On doit donc écrire : `mov ds:100, %eax` pour déplacer l'adresse 100 du segment de donnée dans eax. Quand on fait des push, pop, call, on n'a pas besoin d'ajouter les segments, le processeurs s'occupe de tout.




Cela fonctionne toujours de la même façon.

Avec une telle implémentation, les processus ne peuvent pas partagés des données entre eux.
La solution a été de créer un segment de mémoire partagée. Ce nouveau segment permet à plusieurs processus de partagés de la mémoire en configurant un segment de manière identique. (ES)

**Question.** *Est-ce qu'une autre partie de la mémoire peut être partagée ?*

Les segments permettent à plusieurs processus exécutant le même programme de partager le même segment de code.
Ça suppose que l'on puisse garantir que le segment de code n'est accessible qu'en lecture seule.
Ainsi, dans le tableau comportant les registres limites et bases ont rajoute un 3 bits donnant des droits d'accès (read, write, exécute).
Le contrôleur mémoire sait toujours ce que veut faire le processeur donc on peut bloquer l'accès avec notre nouveau tableau.

Ce mécanisme que l'on vient d'introduire s'appelle la *mémoire segmentée* ou *segmentation.*
Ce système a été très utilisé par MS-DOS.

Actuellement, ce mécanisme n'est plus utilisé.

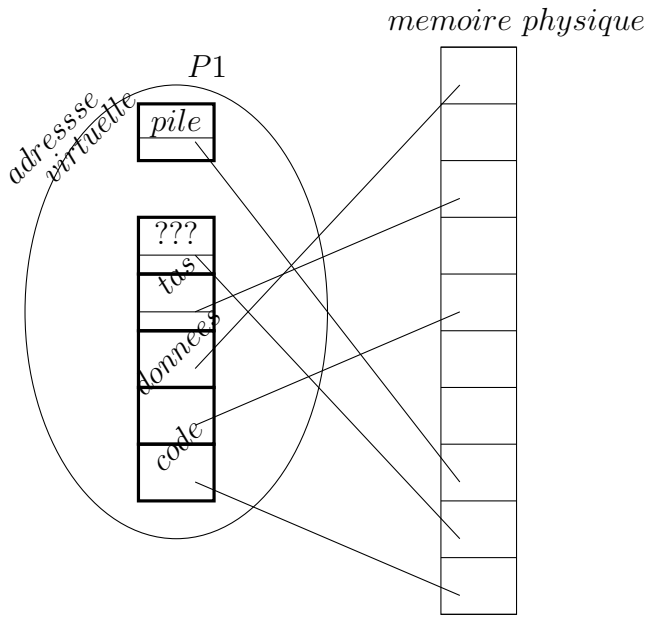
**Question.** *Pourquoi n'utilise-t-on plus ce système ?*

- ⊖ Grâce à ce système, on alloue beaucoup plus facilement la mémoire. Malgré tout, on a pas géré le problème de fragmentation de la mémoire (pleins de petits trous). (trop chère de compacter la mémoire.)
- ⊖ La croissance des segments est toujours très couteuse.

**Choix retenue :** La mémoire est découpée en petits cadres tous de même tailles (= page). Un processus ne peut pas allouer moins d'une page.

Pour créer un processus, il va falloir calculer le nombre de page nécessaire. On ne demande pas de page pour prévoir l'augmentation de la pile. Les pages alloués ne seront pas nécessairement alloués contigument dans la mémoire.
Il n'y a pas de gaspillage du point de vue du système.
A la demande d'une page, le processeur donne la première disponible.

**Exemple.** *le tas peut être divisé sur plusieurs pages discontinues.*




**Question.** *Comment réduire les adresses virtuelles de l'espace d'adressage en numéro de page ? Qui fait la conversion ?*

Une adresse virtuelle (32 bits) c'est $2^{32}$ octets différents soit $2^{32} = 2^{2} * 2^{30} = 4$Go de mémoire.
Les processus ne vont pas tout demander au système (heureusement sinon saturation) d'où l'intérêt de l'espace laissé entre pile et tas.
Une page c'est environ 4 Ko

L'adresse virtuelle est représentée sur 32 bits où 20 bits corresponde au numéro de page et les 12 bits restants au déplacement.

Pour traduire, une adresse virtuelle en adresse physique, il suffit de convertir le numéro de page virtuel en numéro de page physique (le déplacement reste inchangé). Le seul moyen est d'utiliser une table.
Cette table possède $2^{20}$ lignes où chaque ligne représentant un numéro de page virtuelle (20 bits soit 3 octets en pratique 4 octets). La table fait donc 4Mo par processus. Elle ne

peut pas être stockée dans la mémoire du processeur, trop grosse. Elle se trouve donc dans la mémoire physique. (RAM)

Un nouvel élément est ajouté au processeur le MMU (Memory Management Unit) qui est responsable de l'accès à la mémoire demandée par le processeur. Dans la table, il reste un peu de place (les 20 bits ne prennent pas 4 octets) ⇒ on s'en sert pour notifier les droit (R,W,X)

**Question.** *Comment dire à la table qu'une page n'a pas été allouée ?*

On ajoute à la table à chaque adresse un bit "valid"
Si la page a laquelle la MMU essaye d'accéder n'est pas valide, elle déclenche une Interruption : `page fault`

**Question.** *Combien coute l'accès mémoire ?*

```
mov (100), %eax ---> encodée sur au moins 5 octets = 2mots mémoire
```

$\longrightarrow$ 3 lectures par chargement d'exécution
L'accès à la table des pages est prohibitif.

**Idée.** *utiliser un cache qui mémorisent les conversions les plus récentes (page virtuelle, page physique)*

Ce cache se trouve dans la MMU, on l'appelle TLB, Translation Lookaside Buffer. Il permet de pas aller en mémoire pour faire la conversion. On place dans le TLB les données les plus récentes. En effet, on considère qu'il y a beaucoup de boucles dans les processus donc l'information récente va surement être réutilisée.
Le TLB est en fait un simple tableau à 6 colonnes : page virtuelle, page physique, droit (r, w, x) et valide
Lorsque le cache ne possède pas l'information, la traduction est effectuée via la table des pages puis le résultat est conservé dans le TLB. Le cas échéant, on abandonne l'information la plus ancienne.

**Question.** *Quelle taille fait le TLB ?*

Une table de 32 lignes ? C'est suffisant car ça coute 32*4 Ko de données.
On ne doit pas prendre un cache trop grand pour avoir un temps d'accès rapide et une machine qui ne coute pas trop chère.
On parle de "TLB-miss" lorsqu'on est face à un échec de la recherche dans le TLB. Les architectes recherchent à obtenir 1% de TLB-miss pour avoir un cache efficace.

Lors d'un changement de contexte,

— il faut mettre à jour le registre `pageTable` de la MMU
— on vide le TLB (c'est comme ça que marche les pentiums)

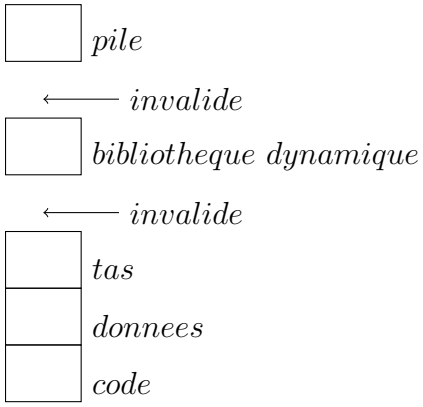
Dans le cas où $P_1 \to P_2 \to P_1$ et $P_2$ n'a besoin que de deux entrées dans le TLB, c'est dommage de tout vider et de tout recharger ensuite. Mais en même temps rien nous dit que $P_1$ va utiliser les mêmes entrées lorsqu'il reprend la main.

Les architectes ont donc repensé le TLB, ils ont ajouté une colonne spécifiant le pid du processus. On voit donc que la MMU possède aussi le pid du processus en cours. Mais ce pid ne sert à rien pour le MMU, c'est l'adresse qui est dans la page-table qui identifie de manière unique le processus.

**Remarque.** *Dans NACHOS, il y a un TLB. Par contre dans les sources, on peut utiliser un TLB sans table des pages ou une table des pages sans TLB*

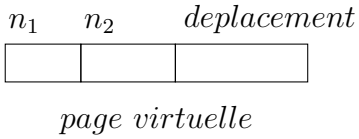
Retour aux tables des pages et plus précisément au problème suivant : Chaque table des pages "pèse" 4Mo ! Or dans les pages stockées, certaines sont parfois invalides et donc ne nous intéresse pas.
En effet il y a un nombre de pages valides $\ll 50\%$

Premier constat : les pages invalides sont plutôt en blocs. Les adresses invalides sont en fait placées entre la pile et le tas, autour des bibliothèques dynamiques.



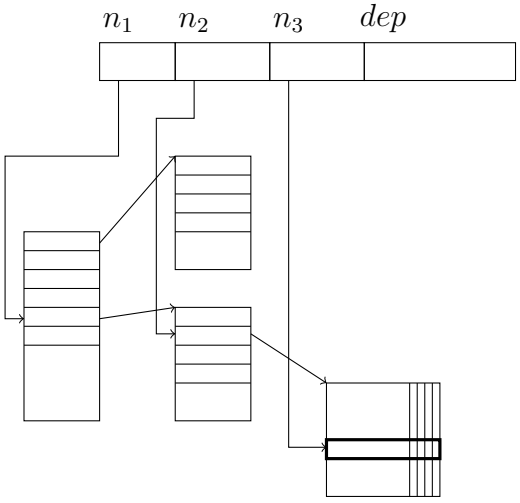

**Idée.** *On peut faire une nouvelle table qui contient des pointeurs vers des morceaux contigus de la table des pages*

**Exemple.** *On coupe la table des pages en 4, on a donc une table d'indirection de 4 lignes à laquelle a accès la MMU. Précédemment, l'adresse virtuelle était composé de la page virtuelle et le déplacement :* `| PV | D |`*. Les adresses commencent par 00 puis par 01, 10 et les dernières par 11. Ce sont les bit de poids fort. Les bits de poids fort nous donne la case de la table d'indirection et ensuite on utilise les bits restants pour se déplacer.*




La MMU possède un pointeur sur une table qui contient $2^{n_1}$ entrées. Chaque entrée est un pointeur sur une table d'indirection où on peut se repérer avec les $n_2$ bits restants. Puis avec le déplacement pour obtenir l'adresse physique correspondante.

Système à 3 niveaux :




**Question.** *Combien ce système coute par rapport à l'ancien ?*

On sait que la table d'adressage fait environ 4Mo.

— La table de premier niveau qui contient des pointeurs (4 octets) "pèse" : 16 lignes * 4 = 64 octets

— Les tables de second niveau : 16 tables * 256 entrées * 4 octets

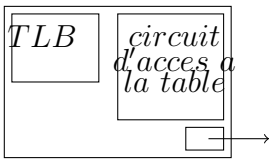
Ce qui nous fait environ 16Ko utilisés en plus. Ce qui n'est pas énorme pour la machine. Cependant avec ce système, un accès mémoire coute 3 accès préalables c'est beaucoup plus couteux.

**Question.** *Pourquoi faire ça ?*

Ce qui est intéressant, c'est que maintenant on peut ne pas allouer une table si elle ne contient que des adresses invalides. En effet, si toutes les pages sont invalides dans une table du dernier niveau, on donne un pointeur nul à la table du niveau au dessus pour signifier qu'on a pas besoin d'allouer de place pour ces pages.
Si pour un niveau, une table ne contient que les pointeurs nuls, on remonte les niveaux en attribuant à la table au dessus aussi un pointeur nul. Et ainsi de suite.
Plus on raffine les niveaux de pages, plus on gagne de la mémoire pour la table des pages

Reste un problème, si une page est valide sur une table du dernier niveau, il faut allouer toutes les tables des niveaux précèdent dont elle dépend, c'est ce qui coute un peu cher.

**Conclusion :**
Une table des pages hiérarchique permet d'économiser de la mémoire en n'allouant pas toutes les sous tables..
Par contre, les conversion coutent cher... Ainsi qu'en circuit. Dans la MMU, le circuit d'accès à la table des pages est très complexe.




**Anecdote :** MIPS a choisit d'utiliser uniquement un TLB (plus grand) , et de supprimer le circuit d'accès à la table qui coute cher
Lorsqu'une conversion ne se trouve pas dans le TLB il effectue un interruption. Le noyau effectue la conversion de façon logicielle, puis renseigne le TLB.
Ça coute très chère mais le pari de MIPS était que cette opération n'est quasiment jamais effectuée puisque presque toutes les conversions sont dans le TLB

## 4.1 Optimisation de la gestion mémoire

**Idée.** *économiser de la place en allouant pas tout de suite toutes les pages d'un processus, on appelle ça l'*allocation paresseuse

**Question.** *Comment faire ?*

Les pages pas allouées sont des pages invalides. Mais normalement quand on fait un accès à une page invalide, ça déclenche un `pagefault`. Comment faire la distinction entre une page invalide et une page non allouée

1. Le système n'alloue pas certaines pages bien qu'elles soient "légitimes" → la table des pages indique "invalide"
2. Lorsque la MMU déclenche une interruption de type "Page Fault", le noyau détermine dans quel segment a lieu "Page Fault"
   (a) si en dehors → SIGSEGV
   (b) l'accès porte sur un segment connu, mais l'accès n'est pas conforme aux droits du segment... → SIGSEGV

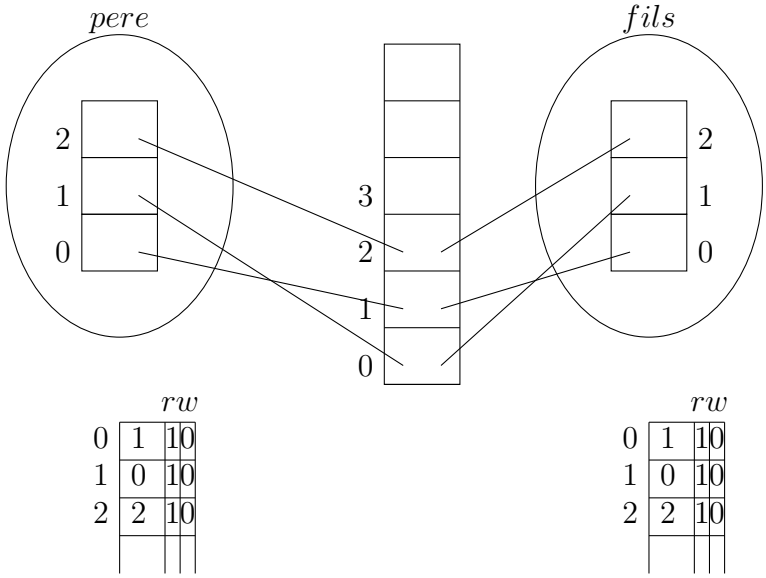
(c) l'accès est correct en théorie → il s'agit d'une page dont l'allocation a été différée, donc le noyau doit corriger
action 1 : allouer une page en mémoire physique et la remet à 0.
action 2 : dans la table des pages qui vient d'avoir un problème, il met l'adresse physique et bit valide à 1
action 3 : on retente l'accès à la page et il n'y a plus de `Page Fault`

**Cas particulier du Fork :**
Fork c'est l'appel système qui clone un processus, il duplique l'espace d'adressage à l'identique. Les valeurs vont changer petit à petit pour différencier le père et le fils.
Le exec(" ") permet d'effacer l'héritage du père et de recommencer.
Du coup c'est dommage d'avoir tout copié pour l'effacer juste après.
Au lieu de copier, on va partager les mêmes pages physiques temporairement, accessible en Readonly

**Question.** *Comment optimiser tout cas ?*

On fait de l'allocation paresseuse :
**1$^e$ cas :** on retarde au maximum l'allocation des pages demandées par un processus
**2$^e$ cas :** optimisation du fork + exec : éviter la recopie des pages lors du fork, en utilisant un partage des pages en lecture seulement

**Fonctionnement sur un exemple :**
On a un processus père avec 3 pages physiques/virtuelles, avec une table de pages dont les droits sont à rw.
Puis un processus fils est créé (par fork) qui partagent les mêmes pages physiques mais qui a bien sur sa propre table de pages.
Du coup, le bit w sur les tables de pages du père et du fils sont mis à 0.




**Question.** *Que se passe-t-il lorsqu'un des processus tente une écriture ?*

Imaginons que le père veut écrire dans la page 0, la MMU regarde dans la table des pages et la elle voit rouge car pas accès d'écriture.

1. une interruption "page-fault" est générée par la MMU
2. le noyau alloue un nouvelle page physique et y recopie le contenu de la page d'origine Un MemCopy est fait de la page physique 1 à la page 3
3. le noyau met à jour l'entrée de la table des pages de manière à utiliser la nouvelle page en mode rw (comme initialement av le fork)
4. au retour de l'interruption, le processus ré-exécute l'instruction fautive

**Question.** *Il peut avoir plus d'un fils. Comment savoir quand une page n'est lu que par un seul processus ?*

En effet, si il ne reste plus qu'un seul processus qui a le droit de lire la page partagé, une fois qu'il voudra la modifier, il va bêtement copier la page. On a du gaspillage.
Le noyau maintient une table indexée par le numéro de pages physiques dans lequel il mémorise :

— pages libres : oui/non
— un compteur indiquant le nombre de tables de pages qui ont accès à cette page physique.

Le dernier processus se aura une erreur mais le noyau lui dira de passer le bit w à 1, et c'est reparti.

Cette méthode est utilisée dans tous les systèmes modernes, tel linux. Une autre méthode possible aurait été d'avoir une table référençant chaque processus. Mais bien trop

couteux.

**3$^e$ cas** : grossissement automatique de la pile
La pile a pour taille max 8 Mo. Mais rares sont les processus qui ont besoin de toute cette mémoire.
En fait toute la pile n'est pas allouée au départ. Les allocations sont faites au fur et à mesure des besoins lors de l'exécution.

Exemple :

```
int main(){
 int i;
 int *ptr = &i;
 ptr -=1000;
 *ptr = 12;
}
```

Ça ne marche pas !
⇒ Erreur

```
int main(){
 int i;
 int *ptr = &i;
 ptr -=1000;
 printf("   ");
 *ptr = 12;
}
```

Ça marche !

En effet, une fois la page allouée, elle est gagnée.
Ainsi tout accès sur la page est effectuée, même s'il se fait après le esp.
On vérifie la place du esp, que dans le cas d'un "page-fault". Dans ce cas, on vérifie si l'accès mémoire est proche de l'esp, si c'est le cas on alloue la page sinon erreur.

# Chapitre 5

# Pagination sur disque

On utilise RAM + disque pour stocker davantage de pages.

| | |
|---|---|
| RAM : 4Go/s (débit) | Disque : 40Mo/s (débit) |
| $\simeq$ nano seconde (temps d'accès) | 3ms (temps d'accès = latence) |

Le disque est simplement utilisé pour stocker des pages dont on ne se sert pas actuellement
Les pages stockés sur le disque ne sont pas accessibles actuellement, donc la table des pages indique "invalide".
Lorsque le processus tente un accès l'une de ces pages, on a une interruption "page-fault"
Si il y a de la place en mémoire physique, le noyau procède comme pour l'allocation paresseuse, avec une lecture disque en plus. (on alloue une nouvelle page puis on charge la page du disque sur la mémoire physique)

**Question.** *Où le noyau mémorise-t-il l'emplacement des pages sur disque ?*

**Remarque.** *un disque est divisé en bloc (plus petit qu'une page). L'espace où sont stockées les pages est l'espace de swap.*

Une solution : on stocke le numéro de bloc dans la table des pages à la place de l'adresse physique. Ainsi il est facile de distinguer une vraie page invalide (adresse à 0) d'une page alloué sur le disque (adresse $> 0$).

Le vrai problème se pose lorsque la mémoire est pleine.
Lorsqu'il s'agit de ramener une page depuis le disque, il faut d'abord "faire de la place".
= choisir une page "victime" et l'évincer.

**Question.** *Comment choisir la victime ?*

Une page victime est une page dont le processeur ne se servait pas. Idéalement, il faut choisir la page dont le prochain accès est le plus lointain.

Impossible à prédire. Mais différentes algorithmes ont été créés pour s'en approcher.

**Idée.** *Algorithme FIFO : les pages sont évincées de la mémoire dans le même ordre qu'elles y sont entrées.*

On choisit la page qui réside en mémoire depuis le plus longtemps. ON pénéalise les pages allouée depuis le plus de temps, on ne sait pas si elle n'est pas encore utilisée. Ce qui compte, ce n'est pas la date d'allocation mais se sont plutot les statistques d'accès. POur avoir les statistiques d'accès sur les pages physiques qui se trouvent en mémoire, la MMU peut nous aider en faisant un compteur. On va regarder dans la table des pages si on ne peut pas noter cette information. On peut prendre commme statistiques, le nombre d'accès à une page ou la date du dernier accès.

Malheureusement il a une mauvaise propriété, c'est l'anomalie de Belady (cf internet). Le nombre de default de pages peut augmenter si on augmente la quantité de RAM.

**Idée.** *associer un compteur aux pages : chaque accès incrémente le compteur*

La MMU peut mettre à jour ce compteur.
Choisir une victime = choisir la page qui possède le plus petit.
→ NFU "Not Frequently Used"
On peut choisir la stratégie "Not Frequently Used (NFU)" ou la Least Recently Used (LRU), NFU n'est pas otimal car on peut avoir un processus qui fait un grand accès mémoire sur une durée t courte et qui n'y accèdera plus après. Il faut donc regarder la dernière date d'accès. La MMU donne seulement un bit nommé "access". On doit se déboruiller avec ça pour simuler une date. Il faut également remttre les statistiques à zéro pardois.

**Problème :** c'est aussi un mauvais critère

Modification de NFU. On utilise une date d'accès au lieu d'un compteur.
date dernier accès $\simeq$ nombre de cycle écoulés
Mais ça devient couteux à stocker.
Un compteur est présent dans chaque table des pages. Tous les 10ms, par ex, la MMU ajoute un 1 aux pages dont on a eu accès. On peut voir ça comme une approximation de la date de dernier accès.

**Une solution** moins couteuse sur le même principe. La MMU stocke un bit quand on fait un accès (nouveau champ : access(ed) dans la table des pages). Il faut ensuite bien l'utiliser et remettre tous à 0 au bout d'un moment. (Sinon tout à 0 et ce n'est plus exploitable)
Un moyen d'utiliser ce nouveau champ, toutes les ms, par ex, ces bits sont stockés dans des tableaux . On peut ensuite comparer ces tableaux afin de trouver la victime (= LRU : Least Recently Used)

**Remarque.** *Linux se contente simplement du bit accès, c'est-à-dire une page non accédée récemment est une victime comme une autre.*

**Question.** *LRU en pratique : dans quelle(s) tables(s) cherche-t-on ?*

1. soit les processus sacrifient leurs propres pages
2. soit les processus volent des pages autrs

Choix des pages "victimes" lorsqu'il est nécessaire d'évincer une page.
Si on trace le nombre de default page par seconde en fonction du nombre de pages, on s'aperçoit que la courbe démarre très haut mais décroit très vite pour finir par presque stagner.
Un tel graphe nous permet de voir le nombre de pages dit min qu'un processus à besoin pour fonctionner de façon acceptable et le nombre de pages dit max au delà du quel, il travaille trop confortablement.
Ce schéma est propre à chaque processus
Pour trouver notre "victime", on cherche la page à l'intérieur du processus qui est dans la position la plus confortable

**Remarque.** *Le cas de linux : on choisit simplement le plus gros processus pour faire son LRU*

S'il y a trop de processus (même des tous petits à une page), il est possible que chaque processus soit dans le min. Dans un tel cas, on courre à la catastrophe car ils font tous des accès disques et pique des pages au détriment des autres. (exemple : fork-bombe)

Lorsqu'on a besoin de libérer une page physique (get-free-page)

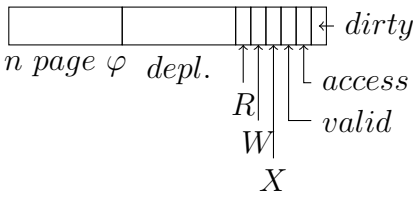
1. choisir une page victime
2. trouve un emplacement sur disque libre
3. écriture de la page sur disque
4. effacement de la page et retour de la fonction

Dans le cas d'un défaut de page, à la suite de get-free-page, il faut :

1. lire la page depuis le disque
2. mettre à jour la table des pages du processus

Afin d'éviter de recopier une page sur le disque alors qu'elle n'a pas été modifiée par le processeur, un bit est ajouté dans la table des pages. C'est le bit Dirty qui est supporté par certains processeurs mais pas tous. Ce bit est positionné à 1 lorsque la page est modifiée.

On a désormais comme entrée de la page des tables :




**Question.** *Dans le cas de l'éviction d'une page "dirty", peut-on éviter l'attente ?*

Si on évince une page "dirty", il faut attendre le temps de la copie de la page sur le disque pour pouvoir disposer de la place libérée.

**Idée.** *On maintient un ensemble de pages libres au delà d'un seuil fixé.*

Par exemple, 10% des pages sont libres.

Pour maintenir ce seuil fixé, périodiquement, le noyau s'assure qu'il possède un ensemble suffisant.
Si ce n'est pas le cas, il anticipe des évictions de pages.
Pour maintenir cet ensemble libre, le noyau utilise un thread qui se réveille périodiquement (kswapd). Possible grâce à la DMA : Direct Memory Access.

La DMA est un procédé informatique où des données circulant de ou vers un périphérique (ici un disque dur) sont transférées directement par un contrôleur adapté vers la mémoire principale de la machine, sans intervention du microprocesseur si ce n'est pour lancer et conclure le transfert.

Ainsi lors du transfert de page de la mémoire au disque, le CPU peut faire autre chose.

**Question.** *Si tous les processus sont dans min (qu'il y en a trop). Qu'est ce qu'on peut faire ?*

utilisation de l'algorithme : gang scheduler