3I010 - Notes de cours
Introduction aux systèmes d'exploitation

BARON Benjamin

Le 18 janvier 2016

D'après le cours de Pierre Sens

Table des matières

Chapitre 1

Introduction

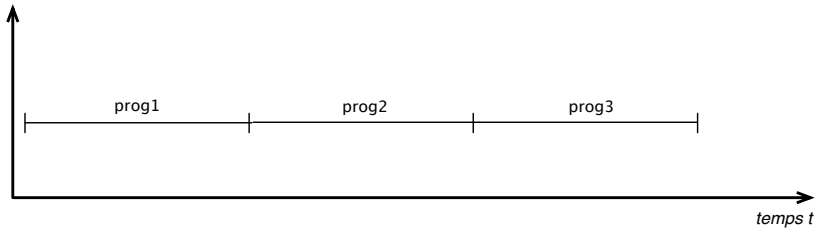
1.1 Historique

1940 Ordinateur à tubes. Il n'y a pas de système.

Exemple. ENIAC : ~20 000 tubes ⇒ 2 octets

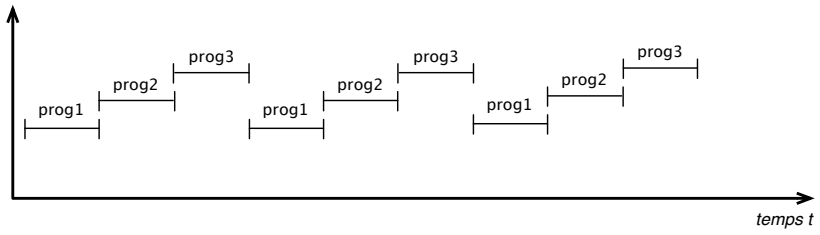
1960 Ordinateur à transistors composé d'un *mainframe* (gros système).

Exemple. ATLAS (60) : traitement par lots (*batch*)



1962 Mainframe à temps partagé (*time sharing*) : CTSS (MIT). Il y a un partage du temps pour tous les programmes : *quantum* alloué pour chaque programme.

Exemple. MUTICS (65), OS1360 (IBM, 64)
+100 utilisateurs pour un ordinateur



1969 Mini-ordinateur : UNICS → UNIX

+10 utilisateurs pour un ordinateur, temps partagé, systèmes en langage machine (assembleur)

1973 Langage C pour ré-écrire UNIX. Il a fallu environ 10 000 lignes de code C pour écrire le premier système d'exploitation UNIX.

1980 Micro-ordinateur (ordinateur dédié à un utilisateur particulier). Il s'agit d'**une** station de travail pour **un** utilisateur.

Exemple. Systèmes d'exploitation pour PC (*Personnal Computer*) ~batch : CP/M, MS-DOS...

1990 – 91 UNIX pour PC (*opensource*) : Mimix, Linx, FreeBSD

1993 MS-DOS devient Windows (équivalent au batch : mono-tâche, mono-utilisateur).

La branche NT de Windows crée Windows NT, un système d'exploitation multi-tâches, multi-utilisateurs

2000 Fin de la branche "MS-DOS – Windows" (3.1, 95, 98, Millenium) de Microsoft. Seule la branche NT de Microsoft subsiste (NT, NT4, 2000, XP, Vista, 7, 8).

2009 Sortie de Windows 7 (NT)

NT → 2000 → XP → Vista → Windows 7 (NT) → Windows 8

1.2 Définitions

Définition (Programme)**.** Expression d'un algorithme à l'aide d'un langage de programmation.

Définition (Tâche)**.** Conjonction d'un programme et des données auxquelles il s'applique.

Définition (Processus)**.** Un processus est l'exécution d'une tâche. Il est caractérisé par la **tâche** et son **contexte d'exécution**, c'est-à-dire la valeur du compteur ordinal, les valeurs des registres, etc.

Définition (Monoprogrammation vs. multiprogrammation)**.** En monoprogrammation, il y a un seul processus à la fois en mémoire. Lorsqu'une tâche est soumise et que le processeur est disponible, on la charge en mémoire puis on exécute le processus associé jusqu'à ce qu'il soit terminé. On passe alors à la tâche suivante.

En multiprogrammation, il peut y avoir plusieurs processus à la fois en mémoire. Une tâche soumise est chargée en mémoire s'il y a de la place et donne naissance à un processus. Un processus est prêt s'il n'est pas en attente d'un événement extérieur (fin d'entrée-sortie, libération de ressource, etc.). Les processus prêts s'exécutent à tour de rôle, on parle de commutation de processus.

Remarque. La multiprogrammation peut être utilisée en batch ou en temps partagé.

Définition (Batch vs. temps partagé)**.** En batch, il n'y a commutation que si le processus actif doit effectuer une entrée-sortie.

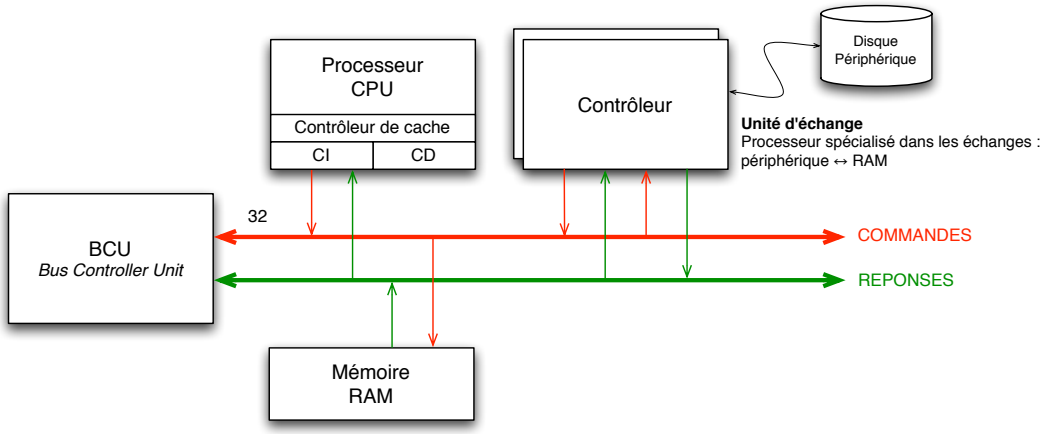
En temps partagé, il y a commutation si le processus actif est en attente d'un événement ou s'il a épuisé son quantum. Un quantum est une durée d'exécution maximale (de l'ordre de 10 à 200 ms).

Définition (*Overhead*)**.** Temps passé (perdu) à effectuer une commutation entre deux processus en temps partagé.

Définition (Unité d'échange)**.** Une unité d'échange sert à effectuer des transferts entre la mémoire principale et un périphérique sans utiliser le processeur central (DMA – *Direct Memory Access*). Elle reçoit un ordre du processeur, effectue le transfert, puis avise le processeur de la fin de l'échange. Durant le transfert, le processeur peut faire d'autres calculs.

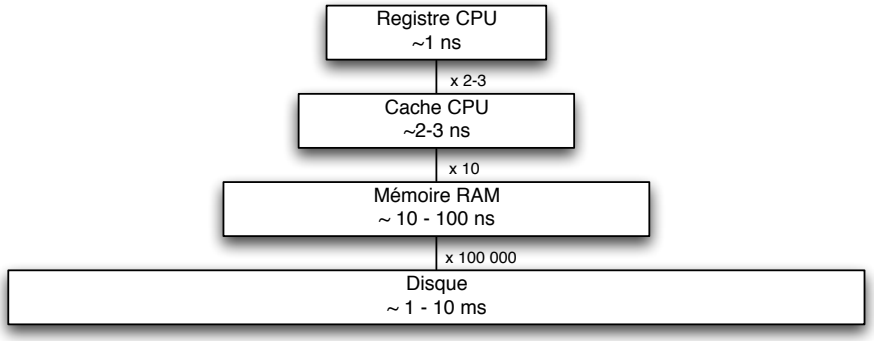
1.3 Composants d'un ordinateur

1.3.1 Architecture de base



Remarque. Il y a un vrai parallélisme entre les traitements sur le CPU et les traitements sur les contrôleurs (qui sont des processeurs spécialisés).

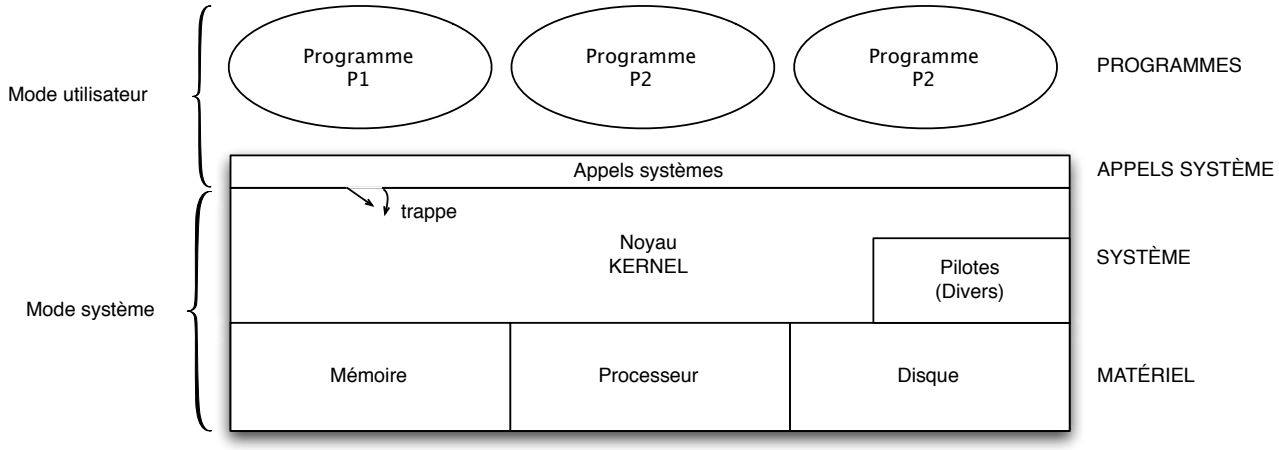
1.3.2 Hiérarchie des temps d'accès



1.4 Les systèmes d'exploitation

Un système d'exploitation (ou OS – *Operating System*) est :
– Un programme qui gère l'ordinateur ;
– Un programme qui donne la vision d'une machine virtuelle (cache la complexité de la machine).

1.4.1 Architecture



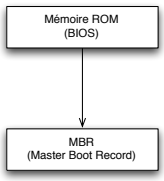
1.4.2 Modes d'exécution

Il existe deux modes d'exécution :
– Mode utilisateur : accès restreint (confinement) aux ressources. Il s'agit de l'exécution des codes des programmes ;
– Mode système : accès privilégié à toutes les ressources du système (noyau + programmes utilisateurs). Il s'agit de l'exécution des codes du noyau.

Les **trappes** permettent le passage d'un mode à l'autre.

1.4.3 Démarrage (boot)

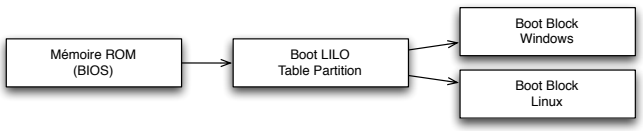
L'allumage de l'ordinateur correspond à l'exécution d'un programme contenu dans la mémoire ROM (mémoire morte ≠ mémoire vive – RAM) sur le CPU.



Les différentes phases pour charger le système d'exploitation en mémoire :

1. Vérifier les périphériques ;
2. Exécuter le premier bloc des disques (MBR) → *Boot block* ;
3. Charger le système d'exploitation.

Multi-boot : remplacer le boot block par un programme fait un aiguillage (eg. GRUB, LILO…).



Chapitre 2

Interruptions

2.1 Introduction

Définition (Interruption)**.** Une interruption (IT) est un signal généré par le **matériel** pour notifier un événement au système d'exploitation.

Le traitement des interruptions est ensuite effectué par le système d'exploitation.

Classification des interruptions

Quand ?

1. Évènement interne : lié à l'exécution du programme en cours
   Exemple. Erreur de programmation (division par zéro, erreur d'adressage...).
2. Évènement externe : indépendant de l'exécution du programme en cours.
   Exemple. Interruptions générées par des périphériques (contrôleur, clavier...) :
   – Fin de transfert (contrôleur de disque) ;
   – Horloge.

Types d'interruptions

1. Interruptions liées aux entrées/sorties (**I**nput/**O**utput ou E/S) : transfert de données des périphériques vers la mémoire.
   → liés au contrôleur (Fin, Prêt, Erreur...).
2. Interruptions externes
   → horloge, clavier...
3. Appels système : interruptions/trappes générées explicitement par les programmes
4. Déroutements : évènements internes, erreurs pendant l'exécution d'un programme.
5. Erreurs matérielles

Priorités exemple d'UNIX — définies par le constructeur

+ prioritaire	0	Horloge
	1	Disque : I/O depuis le disque
	2	Console (souris, clavier...)
	3	Autres périphériques
- prioritaire	4	Appels systèmes

2.2 Traitement d'interruptions

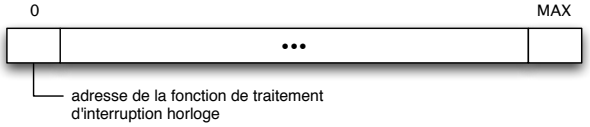
A chaque interruption, une commutation du mot d'état est automatiquement générée par le matériel.

État utilisateur $\xrightarrow{\text{Trappe}}$ État système

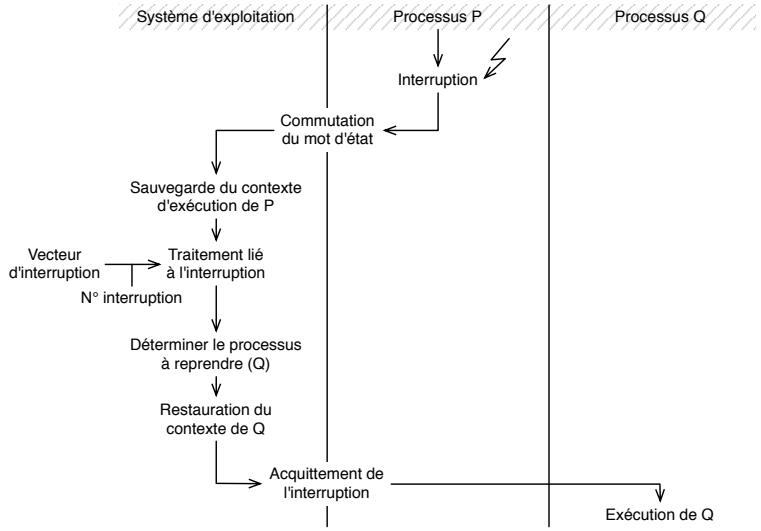
Définition (Mot d'état)**.** Un mot d'état ou PSW (*Processor Status Word*) représente :

– Etat du processeur (actif/inactif) ;
– Mode du processeur (utilisateur/système) ;
– Masque d'interruption ;
– PC/CO : compteur ordinal = la prochaine instruction à exécuter.

Définition (Vecteur d'interruptions)**.** Un vecteur d'interruption est un tableau des adresses des fonctions liées à chaque interruptions prise en compte par le matériel. Les fonctions regroupent les routines d'interruption propre à chaque interruption.

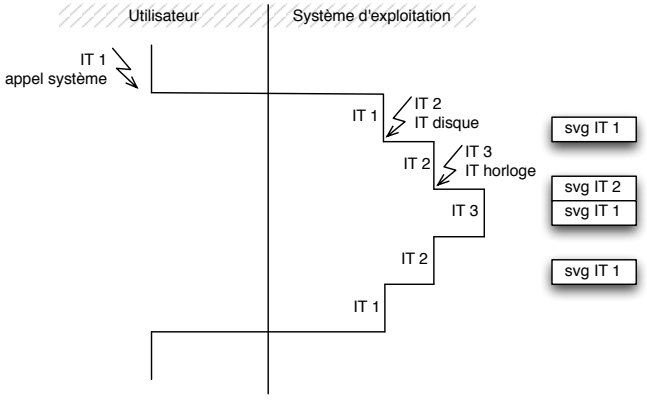


Remarque. En général, les systèmes d'exploitation redéfinissent les fonctions du vecteur d'interruption.



2.3 Pile d'interruption

Des interruptions de priorité supérieure strictement peuvent survenir lors du traitement d'une interruption. Le système utilise une pile pour gérer les interruptions. La pile d'interruption permet un empilement de différentes interruptions.



2.4 Masquage/Désarmement d'interruption

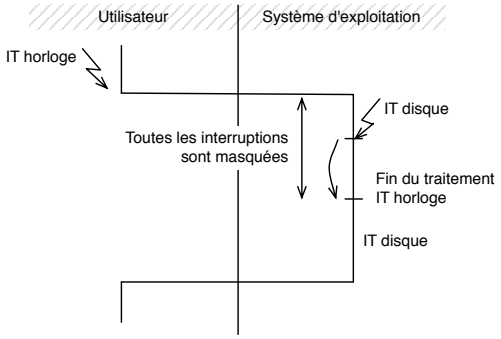
Définition (Désarmement d'une interruption)**.** C'est une **annulation** du traitement d'une interruption ou la prise en compte d'une interruption (eg. utile dans le cas d'un périphérique défaillant qui génère des interruptions inutiles).

Définition (Masquage d'une interruption)**.** Le masquage d'une interruption permet de **retarder** le traitement d'unes interruption.

Remarque. Le masquage d'une interruption permet de masquer toutes les interruptions de priorité inférieure ou égale à celle de l'interruption masquée.

Pendant le traitement d'une interruption, les interruptions de priorité inférieure ou égale sont masquées.

Des fonctions du noyau permettent de masquer des interruptions (exemple la fonction SPL (*Set Priority Level* dans des UNIX).



Chapitre 3

Ordonnancement

L'ordonnancement permet la gestion de l'accès des processus (tâches) au processeur.

3.1 Introduction

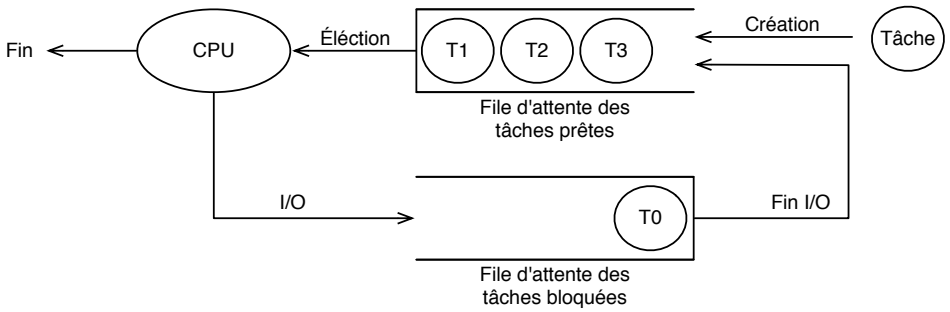
Un algorithme d'ordonnancement (*scheduling*) permet de choisir parmi un ensemble de tâches prêtes celle qui va occuper le processeur. Cet algorithme peut ou non utiliser des priorités affectées aux tâches.

Définition (Famine)**.** La famine survient lorsque les critères utilisés par l'algorithme d'ordonnancement (alors inéquitable) pour déterminer la tâche élue sont tels que l'une des tâches n'est jamais choisie, et ne peut donc pas terminer son exécution. Un bon algorithme d'ordonnancement doit à tout prix éviter les problèmes de famine.

Il existe deux modèles de programmation :
- Mono-programmation (*monotask*) : un seul processus en mémoire ⇒ pas d'ordonnancement.
- Multi-programmation (*multitask*) : plusieurs processus en mémoire à un instant donné en concurrence pour l'accès au processeur.

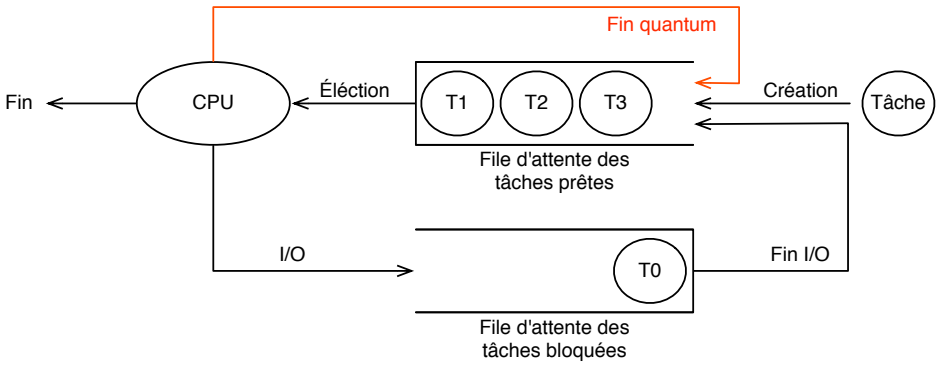
3.1.1 Types d'ordonnancement

Mode Batch La tâche élue conserve le processeur jusqu'à sa terminaison : elle ne peut être interrompue ni par l'arrivée d'une autre tâche, ni par une interruption horloge (eg. FCFS, SJN).
Elle reste sur le processeur jusqu'à sa fin (mono-programmation) **ou** son blocage (eg. I/O : transfert sur/depuis le disque).

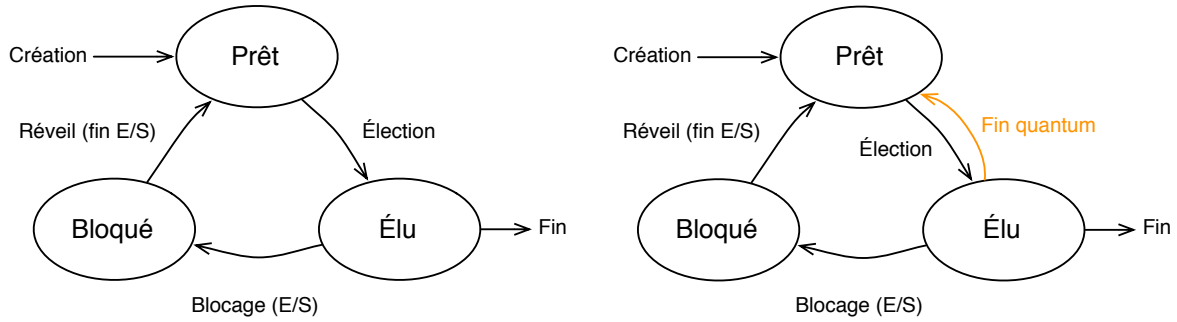


Mode temps partagé Un **quantum** d'exécution correspond au temps maximal d'exécution continue d'une tâche (ie. sans interruption).

Remarque. Un quantum n'est pas une unité d'allocation de temps sur le processeur. Il s'agit d'une borne maximale du temps qu'un processus peut rester en continu sur le processeur. A la fin du quantum, le processus doit quitter le processeur et céder sa place à un autre processus.



3.1.2 Graphe d'état d'un processus



(a) Mode Batch

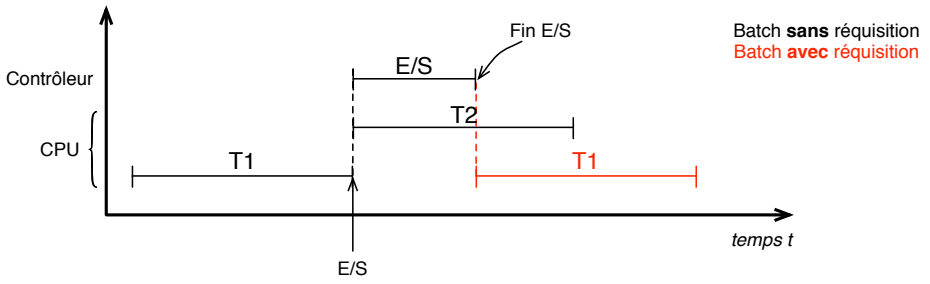
(b) Mode temps partagé

FIGURE 3.1: Graphe d'état d'un processus

3.1.3 Réquisition

Définition (Réquisition)**.** Cela correspond à la possibilité pour une tâche plus prioritaire d'utiliser le processeur. Ainsi, seule l'arrivée d'une tâche plus prioritaire peut interrompre la tâche élue (eg. PSJN). La tâche élue est alors remise en tête de la file des tâches prêtes.

Soient deux tâches T1, T2 telles que la priorité de T1 soit supérieure à celle de T2.



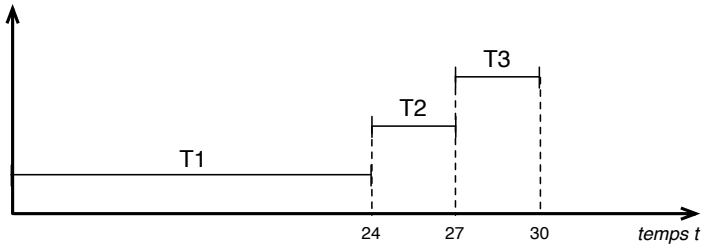
3.2 Algorithmes d'ordonnancement – mode Batch

Remarque. On considérera un système sans quantum, avec ou sans réquisition.

3.2.1 Premier arrivé, premier servi – FIFO (*First In, First Out*) ou FCFS (*First Come, First Served*)

Principe. Les tâches sont traitées dans leur ordre d'arrivée.

Considérons trois tâches T1, T2, T3 telles que :
– T1 est crée à $t = 0$, de durée 24 secondes ;
– T2 est crée à $t = 0$, de durée 3 secondes ;
– T3 est crée à $t = 0$, de durée 3 secondes.



Tâches créées dans l'ordre : T1, T2, T3.

Calcul des différents temps de mesure pour une tâche :
– Temps de réponse : $T_{r_{\mathrm{T}_i}}$ = date de fin de T_i − date de création de T_i ;
– Temps d'attente : $T_{a_{\mathrm{T}_i}}$ = temps de réponse de T_i − durée de T_i.

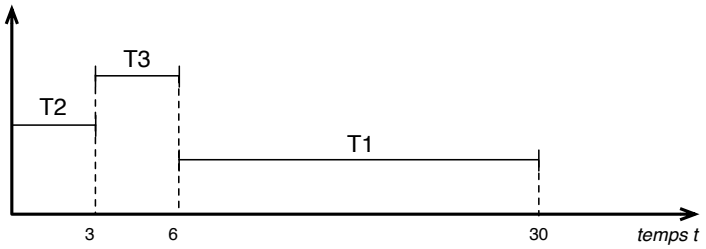
Tâche	Temps de réponse	Temps d'attente
T1	$T_{r_{\mathrm{T}_1}} = 24 - 0 = 24$ s	$T_{a_{\mathrm{T}_1}} = 24 - 24 = 0$ s
T2	$T_{r_{\mathrm{T}_2}} = 27 - 0 = 27$ s	$T_{a_{\mathrm{T}_2}} = 27 - 3 = 24$ s
T3	$T_{r_{\mathrm{T}_3}} = 30 - 0 = 30$ s	$T_{a_{\mathrm{T}_3}} = 30 - 3 = 27$ s

Temps d'attente moyen : 17 secondes.

3.2.2 Plus courts travaux d'abord – SJF (*Shortest Job First*) ou SJN (*Shortest Job Next*)

Principe. Le système choisit la tâche dont le temps d'exécution restant est le plus court (temps d'exécution initial de la tâche est supposé connu).

Considérons trois tâches T1, T2, T3 telles que :
– T1 est crée à $t = 0$, de durée 24 secondes ;
– T2 est crée à $t = 0$, de durée 3 secondes ;
– T3 est crée à $t = 0$, de durée 3 secondes.



Tâche	Temps de réponse	Temps d'attente
T1	$T_{r_{T_1}} = 30 - 0 = 30$ s	$T_{a_{T_1}} = 30 - 24 = 6$ s
T2	$T_{r_{T_2}} = 3 - 0 = 3$ s	$T_{a_{T_2}} = 3 - 3 = 0$ s
T3	$T_{r_{T_3}} = 6 - 0 = 6$ s	$T_{a_{T_3}} = 6 - 3 = 3$ s

Temps d'attente moyen : 3 secondes.

Conclusion. La stratégie SJF (*Shortest Job First*) est meilleure que la stratégie FIFO (*First In, First Out*).

Quelques problèmes se posent :
- Le temps d'exécution est rarement connu à l'avance ;
- Risque de famine des tâches longues.

3.3 Algorithmes à temps partagé

Utilisation de la notion de quantum (*time-slice*).

3.3.1 Implémentation

Implémentation d'algorithmes à temps partagé : Utilisation des registres horloge (RH et Rtempo) ou PIT (*Programmable Interrupt Timer*).

Définition. L'horloge (*Timer*) est assimilable à un contrôleur. C'est un composant externe au processeur qui décrémente le registre horloge RH à une certaine fréquence (~1MHz – toutes les μs).

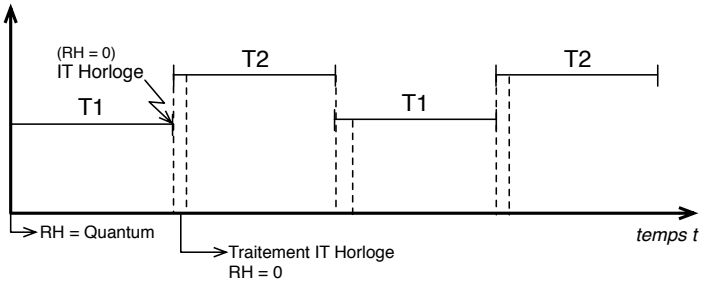
Si RH = 0, alors le Timer lève une interruption horloge, qui sera ensuite traitée par le système d'exploitation.

Le Timer a pour rôle d'implémenter les quantum, mettre à jour l'heure de la machine, gérer les alarmes.

Traitement d'une interruption horloge (interruption la plus prioritaire du système) :
- Traiter les alarmes ;
- Mise à jour de l'horloge ;
- Élire une nouvelle tâche ;
- Réinitialiser le registre RH.

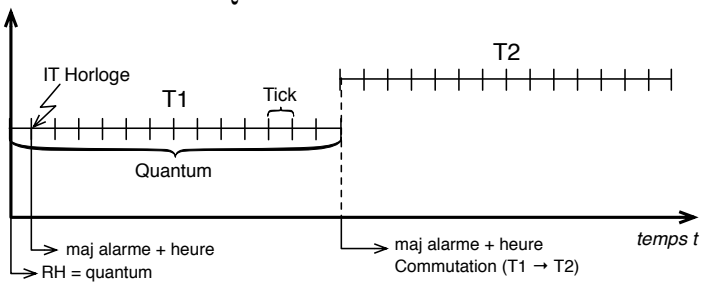
Remarque. Le registre RH est un registre privilégié.

Le quantum est lié à la perception de parallélisme de l'utilisateur. De ce fait, le quantum a une durée d'environ 100 ms.



Définition (Tick)**.** Un Tick (ou *jiffies* au sens de Linux) est la valeur (temps) entre deux interruptions horloge. Un tick est un paramètre système initialisé à environ 1 – 10 ms.

$$\text{Quantum} = N \times \text{Tick}$$



Une expiration de quantum aura ainsi lieu tous les N ticks.

3.3.2 Algorithme du tourniquet – RR (*Round Robbin*)

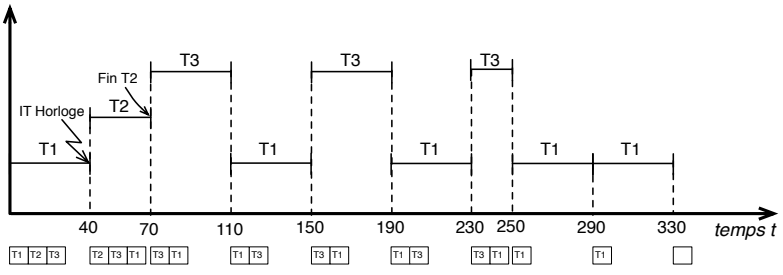
Accès des tâches à tour de rôle au processeur. Cet algorithme nécessite une file des tâches prêtes.

Création d'une tâche Insertion en queue ;

Election d'une tâche (N ticks) Choisir la tâche en tête de la file des tâches prêtes ;

Fin de quantum (N ticks) Réinsertion de la tâche élue en queue. Celle-ci devient une tâche prête.

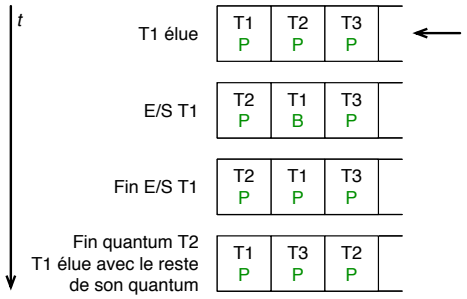
Soient trois tâches T1, T2, T3 créées dans cet ordre à $t = 0$:
– T1 dure 200 ms ;
– T2 dure 30 ms ;
– T3 dure 100 ms.
Considérons un quantum de durée 40 ms et un temps de commutation négligeable (< 1 ms).



Temps d'attente des différentes tâches :
– $T_{a_{\mathrm{T}_1}} = (330 - 0) - 200 = 130$ ms ;
– $T_{a_{\mathrm{T}_2}} = (70 - 0) - 30 = 40$ ms ;
– $T_{a_{\mathrm{T}_3}} = (250 - 0) - 100 = 150$ ms ;

Lors d'une entrée / sortie, la tâche passe à l'état bloqué. A la fin de l'E/S, la tâche passe à l'état prêt. Il faudrait alors donner une priorité plus forte aux tâches réveillées (ie. qui sortent de l'état bloqué) en maintenant la position de la tâche dans la file. Au réveil de la tâche, le quantum restant de celle-ci lui est alloué.

Exemple. Soit la file des tâches suivante (P : état prêt, B : état bloqué) :



Avantages / inconvénients de cet ordonnancement :

- Equitable, simple ;
- Pas de famine : si une tâche T est prête, soit n la taille de la file devant T et q la quantum, alors T attend au plus $n \times q$;
- Trop équitable : pas de priorités entre les tâches.

3.3.3 Priorités

Priorité fixe (statique)

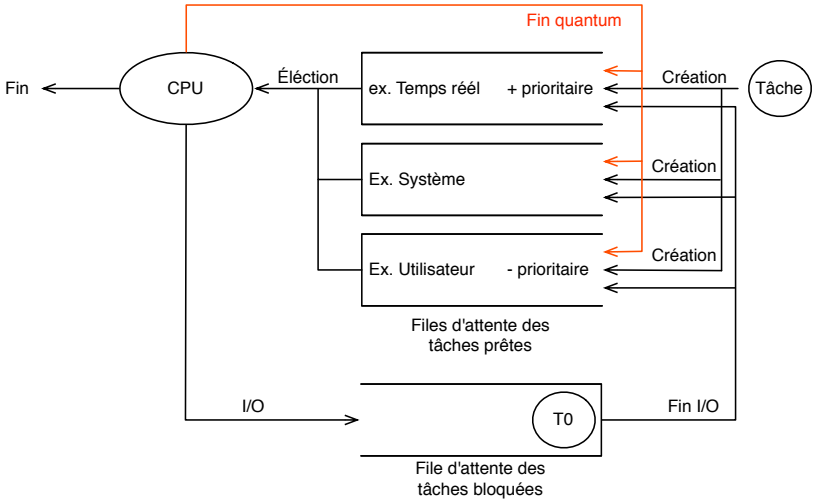
Chaque tâche dispose à sa création d'une priorité qui conserve la même valeur tout au long de son exécution.

Considérons une priorité fixe avec N niveaux de priorités et une file par niveau de priorité.

Création d'une tâche Attribuer un niveau à la tâche, insertion de celle-ci en queue de la file correspondante.

Election d'une tâche Choisir la tâche prête de la file non vide la plus prioritaire.

Fin du quantum La tâche élue est ré-insérée en queue de sa file.



Avantages / inconvénients de cet ordonnancement :

- Permet de prendre en compte les tâches prioritaires ;
- Pas équitable : risque de famine des tâches les moins prioritaires.

Priorité dynamique

Les priorités affectées aux tâches changent en cours d'exécution. Plusieurs algorithmes sont alors possibles suivant le critère de changement de priorité.

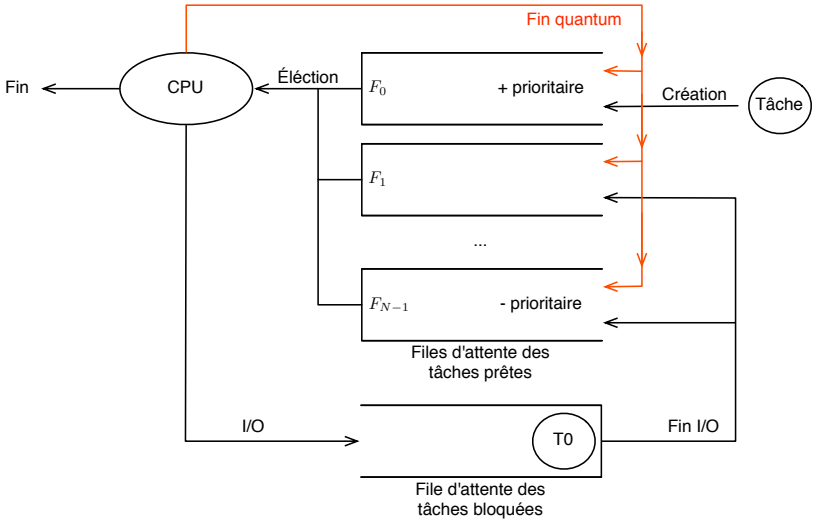
Considérons une priorité dynamique avec N files de F_0 à F_{N-1} (où la file F_0 est la plus prioritaire).

Création d'une tâche Insertion de la tâche en queue de la file F_0.

Election d'une tâche Choisir la tâche prête de la file non vide la plus prioritaire (utilisation d'une boucle d'attente `while`).

Fin quantum La tâche élue est ré-insérée en queue de la **file suivante**.

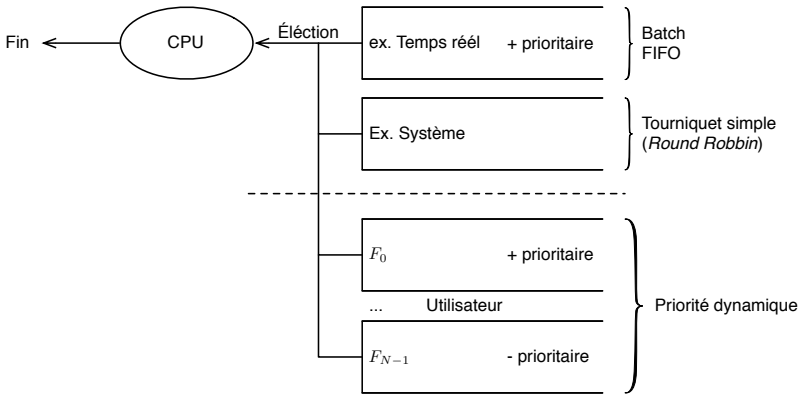
Remarque. Les tâches issues de la file F_{N-1} sont ré-insérées en queue de la file F_0.



Avantages / inconvénients de la priorité dynamique :

- **Plus** une tâche est élue, **moins** elle est prioritaire.
- Une tâche ancienne est donc moins prioritaire ;
- Approximation de SJF (*Shortest Job First*) sans connaître à l'avance la durée des tâches.
- Risque de famine des tâches longues si il y a un flot continu de création de tâches.
- Résolution de la famine : régulièrement (tous les N quantum ~1 seconde), le système remonte la priorité des toutes les tâches prêtes.

3.4 Ordonnancement dans les systèmes "modernes"



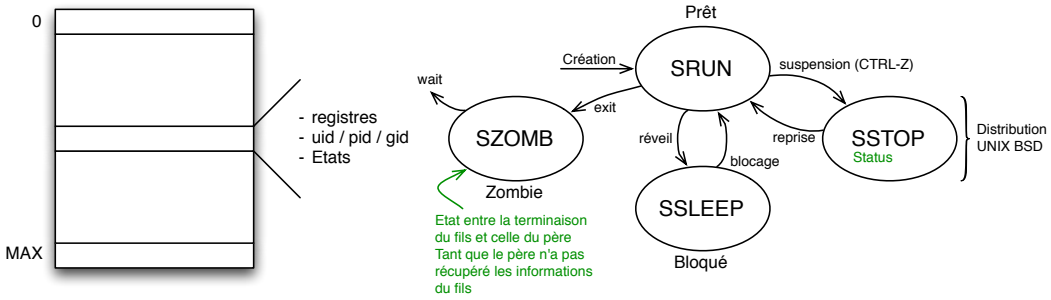
Chapitre 4

Processus UNIX

4.1 Introduction

Un processus en mémoire correspond à :
– Une pile (variables locales) – R/W ;
– Un segment données (variables globales, tas) – R/W ;
– Un code – R.
Contexte matériel du processus : registres.

Informations système :
– pid : numéro de processus (unique) ;
– uid : numéro d'utilisateur ayant exécuté le processus ;
– gid : numéro du groupe auquel appartient l'utilisateur ayant lancé le processus.
Le système contient une table globale regroupant tous les processus : `proc[]` ou `task[]`.



4.2 Primitives permettant de manipuler les processus

Afin de savoir quels sont les directives pré-processeur `#include` à utiliser, s'informer avec la commande `man <commande>` du terminal.

En général, utiliser :

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
```

4.2.1 Création d'un processus

```
int main(int argc, char * argv[], [char * envp[]])
```
nombre d'arguments ← argc
arguments → argv[]
variables d'environnement → envp[]

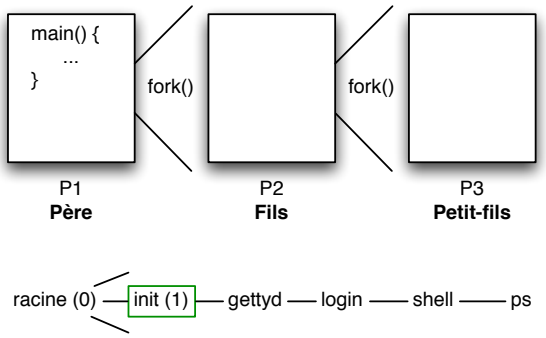
```
return (int) ret;
```
0 (EXIT_SUCCES) : OK
≠ 0 (EXIT_FAILURE) : KO

Soit l'exécutable `./mon_prog` créé à la suite de la compilation. On a alors :

`argv[0]`	`argv[1]`	`argv[2]`	...	`argv[argc-1]`	`argv[argc]`
`./mon_prog`	`arg1`	`arg2`	...	`argN`	`NULL`

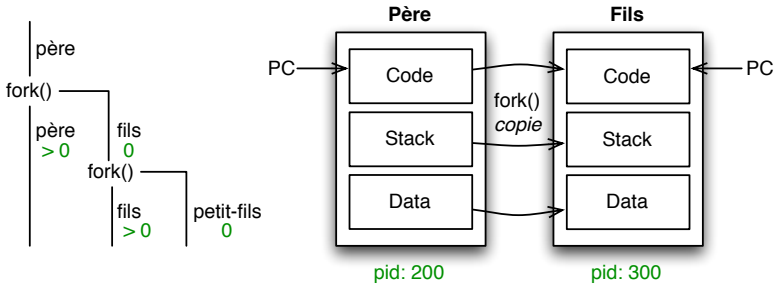
4.2.2 Création de nouveaux processus

Notion de filiation :



Primitive (appel système) `pid_t fork();` création par duplication du père

Remarque. Le type `pid_t` est assimilable au type `int`.
Il n'y a pas de partage de variables une fois le processus père dupliqué.



La valeur de retour de `fork()` permet de distinguer le processus père du processus fils créé :
– > 0 : pid du processus fils ; valeur retournée au processus père ;
– 0 : valeur retournée au processus fils (pour savoir son pid : `pid_t getpid()`) ;
– -1 : erreur sur la fonction `fork()` (la table des processus est probablement remplie).
Il existe également d'autres primitives relatives aux processus :
– `pid_t getpid(void)` : retourne le pid du processus courant ;
– `pid_t getppid(void)` : retourne le pi du père (parent) du processus courant.

Exemple. Soit le code suivant :

```
int main(int argc, char * argv[]) {
        pid_t p;
        int a = 10;
        p = fork();

        if(p == -1) { // Erreur
                perror("Erreur sur le fork.\n");
                return 1;
```

```
        }

        if(p == 0) { /* le fils */
                printf("Je suis le fils %d, pere : %d.\n", getpid(), getppid());
                a++;
                return 0; // a = 11
        } else { /* le pere */
                printf("Pere de %d.\n", p);
                return 0; // a = 10
        }
}
```

4.2.3 Terminaison

La terminaison d'un processus s'effectue avec la primitive `void exit(int etat);`

Le processus se termine en retournant la valeur entière `etat`. Il passe alors à l'état *zombie* jusqu'à ce que son père récupère son état de sortie (à l'aide de la primitive `wait(&status)`).

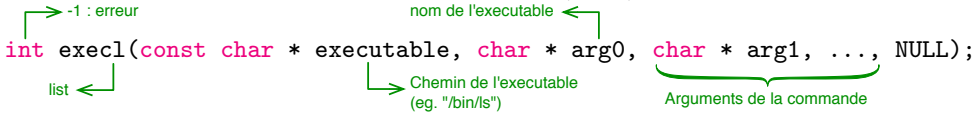
Remarque. Les valeurs `EXIT_SUCCESS` (0) et `EXIT_FAILURE` ($\neq 0$) sont définies avec le C Standard (ISO C99).

4.2.4 Recouvrement (primitive exec)

Définition (Recouvrement)**.** Le recouvrement permet à un processus de changer (écraser) son code. De ce fait, le processus va exécuter un nouveau `main`.

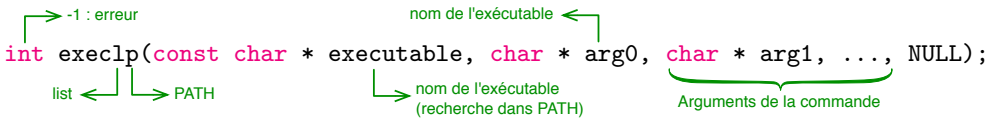
Remarque. Le recouvrement ne permet pas de créer de nouveau processus. En effet, le processus garde le même pid et le même père.
Pour créer un nouveau processus : `fork()` puis `exec()`.

Primitives Les différentes primitives de recouvrement (`exec`) sont présentées dans la suite.

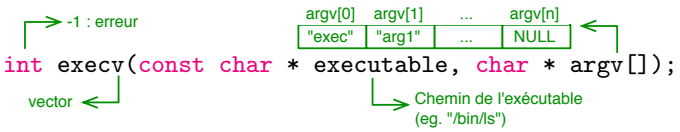


Exemple. Soit le code suivant :

```
p = fork();
if(p == -1) { // Erreur
        perror("Erreur sur le fork.\n");
        return 1;
}
if(p == 0){ /* le fils */
        execl("/bin/ls", "ls", "f1", NULL);
        perror("pb exec");
        exit(1);
}
```

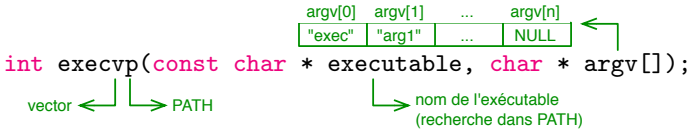


Exemple. `execlp("ls", "ls", "f1", NULL);`



Exemple. Soit le code suivant :

```
char * argv[3] = {"ls", "f1", NULL};
execv("/bin/ls", argv);
```



Exemple. Soit le code suivant :

```
char * argv[3] = {"ls", "f1", NULL};
execvp("ls", argv);
```

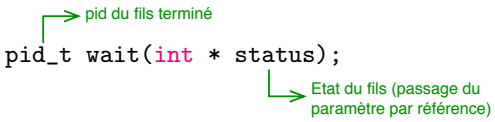
Remarque. Il existe également les primitives `execle` et `execve` qui permettent aussi de modifier les variables d'environnement du processus à exécuter.

4.2.5 Synchronisation père/fils (primitive wait)

La primitive `pid_t wait(int * etat)` permet au père d'attendre la fin d'un seul fils quelconque.

Remarque. S'il faut attendre la fin de n fils, il faut faire une boucle `for` contenant n itérations appelant à chaque fois la primitive `wait`. En général, il faut autant de `wait` que de `fork`.

L'argument `etat` passé par référence à la primitive `wait` n'est pas forcément alloué dynamiquement (avec `malloc`). Il est possible de l'allouer statiquement, puis de le passer par référence.



Argument et valeur de retour :
– Valeur de retour : pid du processus fils terminé ;
– Paramètre de sortie `status` passé par référence : état du processus fils (valeur de retour de `exit`).
Macros définies dans `<wait.h>` manipulant le paramètre `etat` :
– `WEXITSTATUS(status)` : valeur de retour ;
– `WIFEXITED(status)` : valeur $\neq 0$ si fin normale.

Exemple. Soit le code suivant :

```
pid_t p, d;
int e;
p = fork();
if(p == -1) { // Erreur
        perror("Erreur sur le fork.\n");
        return 1;
}

if(p == 0){ /* le fils */
        exit(2);
} else { /* le pere */
        d = wait(&e); // wait(NULL) -- pere bloque
        if(WIFEXITED(e)){ // WEXITSTATUS(e) == 0
```

```
                printf("Fin normale de %d.\n", d);
        } else {
                printf("Probleme fils : %d, etat : %d.\n", d, WEXITSTATUS(e));
        }
}
```

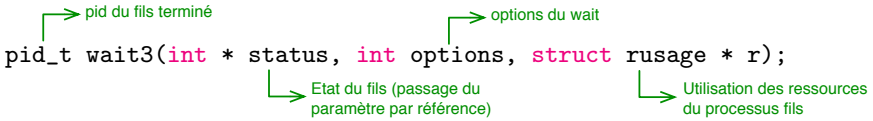
Remarque. La primitive `wait` retourne -1 si pas de fils.

Le fils terminé reste à l'état de zombie jusqu'à ce que le père fasse `wait`.

Exemple. Créer un zombie pendant 30 secondes (environ)

```
p = fork();
if(p == -1) { // Erreur
        perror("Erreur sur le fork.\n");
        return 1;
}

if(p == 0){ /* le fils */
        exit(0);
} else { /* le pere */
        sleep(30);
        wait(NULL);
}
```

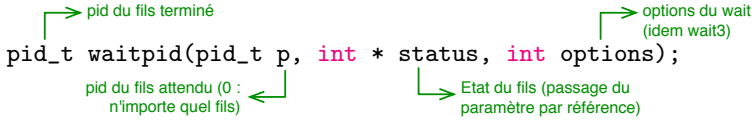
Remarque. Si le père est terminé, ses fils (orphelins) sont "adoptés" par 'init (processus fils du processus racine dont le pid est égal à 1).



Arguments :

- `options` :
  - 0 pas d'option ;
  - `WNOHANG`, non bloquant retourne 0 si fils pas terminé.
- Structure `struct rusage` (`man getrusage`) :

```
struct rusage {
        struct timeval ru_utime; /* CPU consomme par le fils en mode user */
        strcut timeval ru_stime; /* CPU consomme par le fils en mode system */
}
```



Arguments :

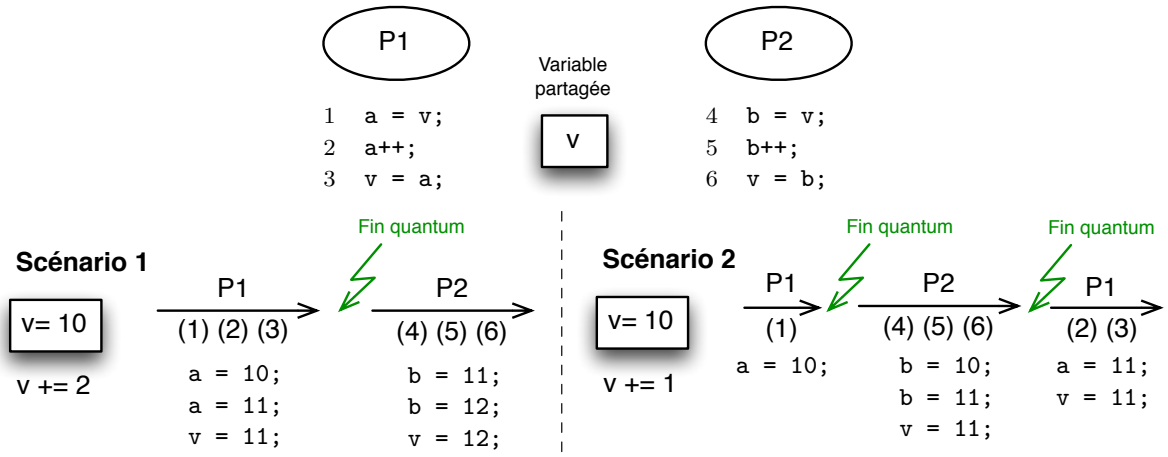
- `p` : pid du fils attendu (0 : n'importe quel fils) ;
- `options` : idem que `wait3`.

Chapitre 5

Synchronisation entre tâches

5.1 Définitions

On considère un système :
– Multi-programmation (multi-tâches) en temps partagé ;
– Plusieurs tâches partagent des données (variables).



On a alors affaire à un code non déterministe (plusieurs issues différentes non prévisibles).

Définition (Ressource critique)**.** Une ressource critique est une ressource partagée entre plusieurs processus et dont l'accès doit être contrôlé pour préserver la cohérence de son contenu (variable `v`, fichier, etc.). En particulier, ce contrôle peut impliquer un accès en exclusion mutuelle, c'est-à-dire que la ressource n'est jamais accédée par plus d'un processus à la fois.

Définition (Sections critiques – SC)**.** Les sections critiques sont des portions de codes manipulant des ressources critiques (dans le cas de l'exemple, il s'agit des sections (1)(2)(3) et (4)(5)(6)). Ces portions de code doivent être exécutée de manière indivisible.

Définition (Indivisibilité)**.** L'indivisibilité signifie que deux sections critiques concernant une même ressource ne sont jamais exécutées simultanément mais toujours séquentiellement (dans un ordre arbitraire). Il y a ainsi exclusivité entre ces séquences d'instructions.

Définition (Exclusion mutuelle)**.** L'exclusion mutuelle est le mécanisme qui assure qu'un processus au plus est en section critique sur une même variable.

Définition (Synchronisation)**.** L'exclusion mutuelle n'est qu'un cas particulier de synchronisation. Une synchronisation est une opération influant sur l'avancement d'un ensemble de processus :
– Etablissement d'un ordre d'occurrence pour certaines opérations (envoyer / recevoir) ;
– Respect d'une condition (rendez-vous, exclusion mutuelle, etc.).

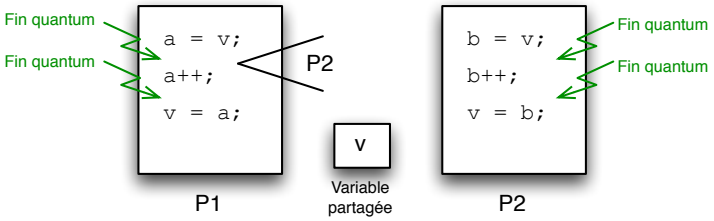
Définition (Attente active)**.** L'attente active est la mobilisation d'un processeur pour l'exécution répétitive d'une primitive de synchronisation jusqu'à ce que la condition de synchronisation permette la continuation normale du processus.

Définition (Interblocage (*deadlock*))**.** L'interblocage se produit lorsque deux processus concurrents s'attendent mutuellement. Les processus bloqués dans cet état le sont alors définitivement.

5.2 Mécanisme d'exclusion mutuelle

Propriétés des algorithmes d'exclusion mutuelle :
– Sûreté : un processus au plus en section critique ;
– Vivacité : toutes les demandes d'accès aux sections critiques doivent être satisfaites.

```
EnterSC();
        /* Section Critique */
SortirSC();
```



Les primitives `EntrerSC()` et `SortirSC()` doivent être rendues indivisibles (opérations atomiques) pour garder la cohérence de leur contenu. De ce fait, il faut masquer les interruptions système pendant leur exécution et les rendre primitives système.

5.2.1 Masquage IT Horloge

Le masquage de l'interruption horloge permet de désactiver le temps partagé pendant la section critique :
– `EntrerSC()` : Masque l'IT Horloge ;
– `SortirSC()` : Démasquer l'IT Horloge.
Plusieurs problèmes se posent alors :
– Trop dangereux – risque de monopoliser le processeur ;
– Exclut **tous** les processus ;
– Le masque d'IT Horloge est utilisé en interne par le système.

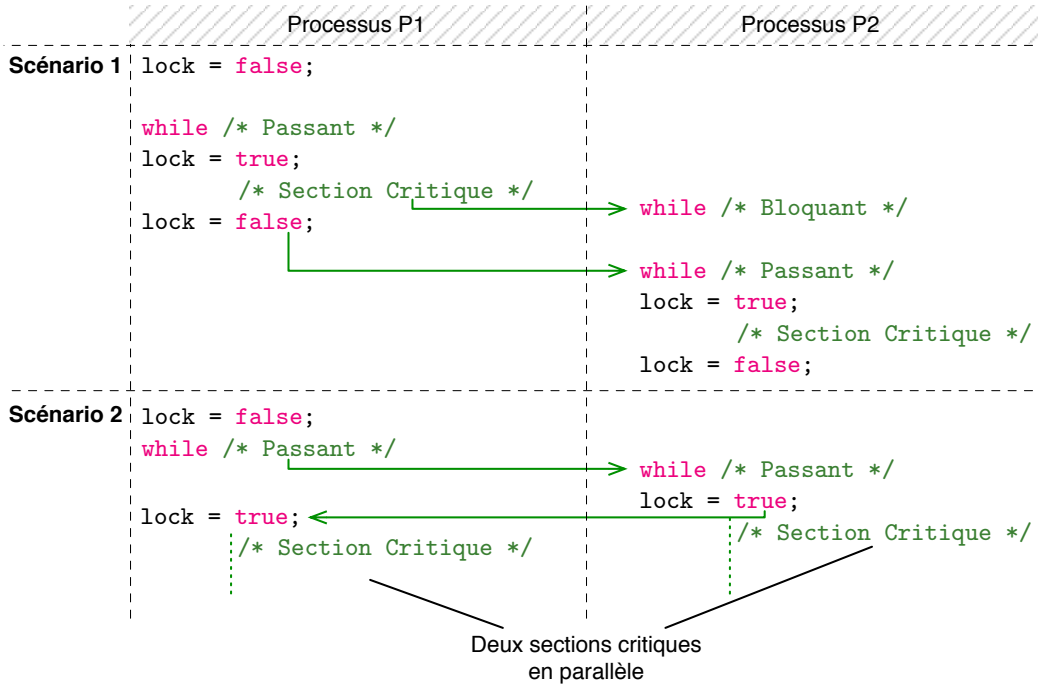
5.2.2 Variables de synchronisation

Des variables partagées indiquent si les ressources critiques sont disponibles.

Solution 1. Une variable booléenne

Soit la variable `bool lock` ayant pour valeur :
– `true` si la ressource est verrouillée ;
– `false` si la ressource est non verrouillée.

```
                    bool lock = false;

EnterSC()           while(lock == true);
Attente active      lock = true;

                    /* Section Critique */

SortirSC()          lock = false;
```



Solution 2. Algorithme de Peterson (1981)

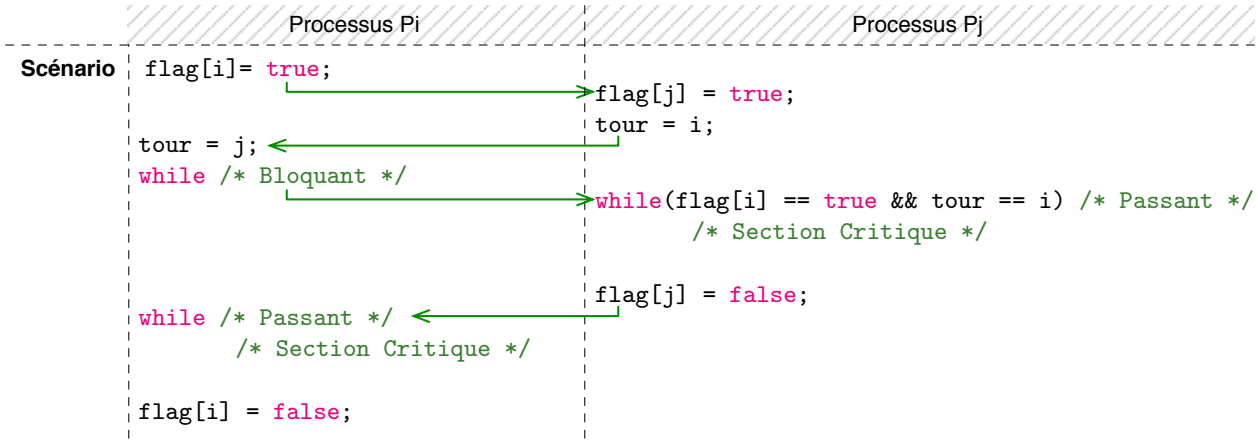
L'algorithme de Peterson assure l'exclusion mutuelle pour deux processus P_i et P_j (généralisable). Cet algorithme est basé sur l'attente active des deux processus P_i et P_j.

Soient les variables partagées :
- Un tableau de booléen `flag` indique si un processus est demandeur de section critique (SC) ;
- Une variable `tour` indique à qui le tour.

```
bool flag[2] = {false, false};
int tour;

EnterSC()          flag[i] = true;
Attente active     tour = j;
                   while(flag[j] == true && tour == j);

                   /* Section Critique */

SortirSC()         flag[i] = false;
```

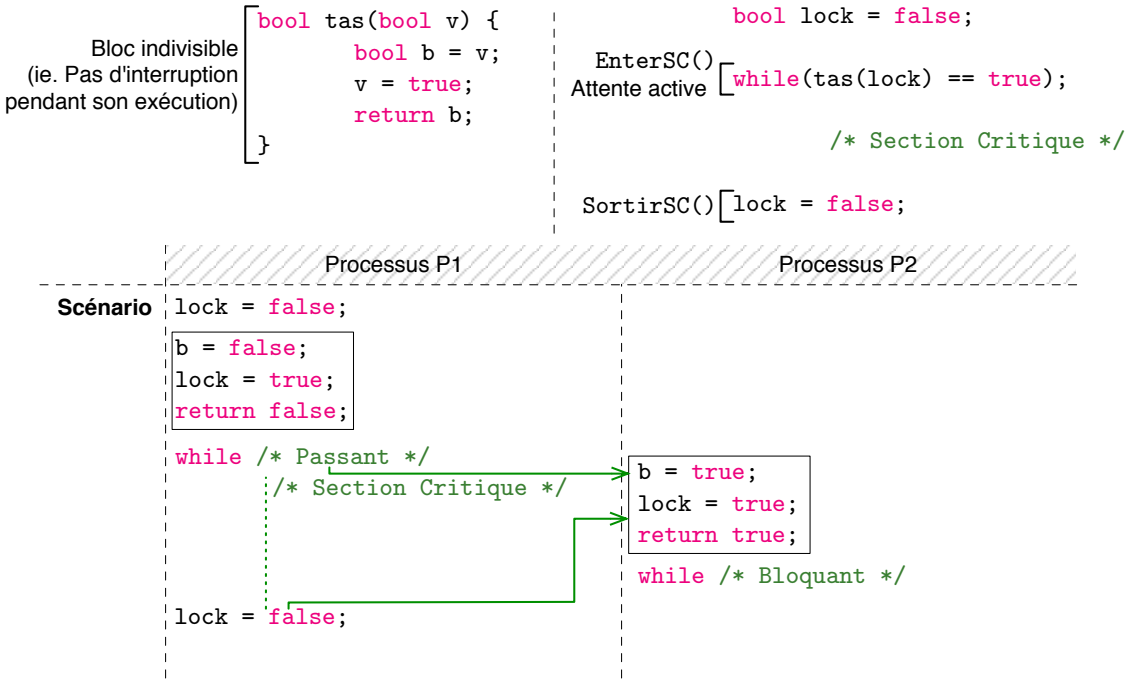
Cas 1 Le processus P_i demande seul l'accès à la section critique.
`flag[j] == false` et `while` passant.
Le processus P_i a accès à la section critique.

Cas 2 Les processus P_i et P_j demandent l'accès à la section critique.
La variable `tour` fixe l'ordre.

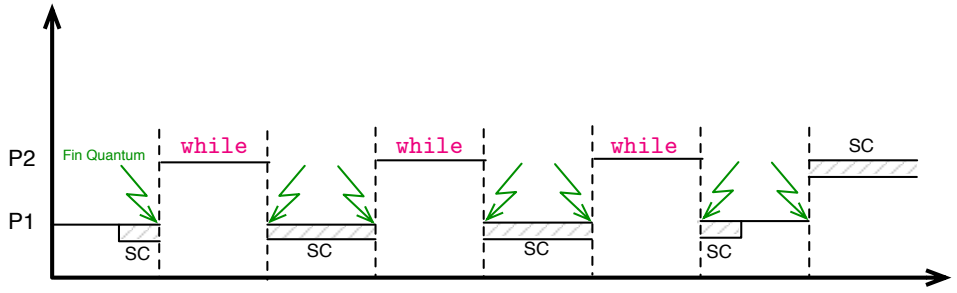


Solution 3. Test-and-set (t-a-s)

Définition (Test-and-set)**.** Un bloc d'instructions indivisible pour positionner et tester une variable. Il ne peut pas y avoir d'interruption pendant l'exécution du code de test-and-set.



Problèmes de performance Problème de performance des solution à base d'attente active.



Ces solutions sont environ 2 fois plus lentes que les solutions sans attente active.

5.3 Sémaphores

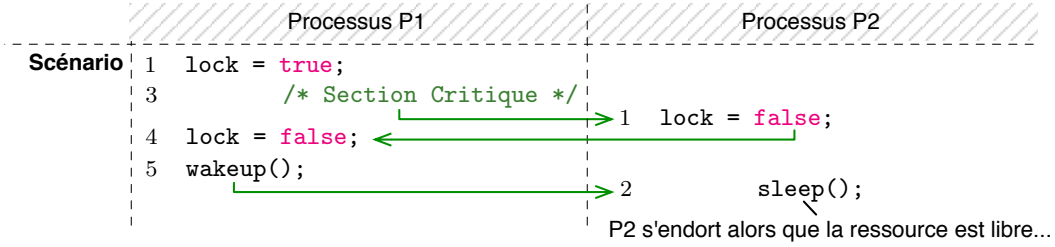
Les sémaphores permettent d'éviter les attentes actives.

5.3.1 Suspension/Réveil

Considérons deux primitives :
– `sleep()` : endormir un processus (passe de l'état prêt à l'état bloqué) ;
– `wakeup()` : réveiller un processus.

Exemple. Soit le code suivant (Ne marche pas !!) :

```
1 if(tas(lock) == true)
2         sleep();
3 /* Section critique */
4 lock = false;
5 wakeup();
```



5.3.2 Sémaphore : Dijkstra 76

Un sémaphore est une structure gérée par le système d'exploitation permettant de contrôler l'accès à une ressource en évitant l'attente active. Celui-ci est composé de :
– Un compteur (`cpt`) = nombre de processus autorisés ;
– Une file d'attente (`file`) pour les processus non autorisés.
Considérons les trois primitives indivisibles suivantes :

`Init(sem, cpt)`	Création d'un sémaphore `sem` dont le compteur est initialisé à `cpt`.
`DS(sem)`	Destruction d'un sémaphore. Si la file n'est pas vide un traitement d'erreur doit être effectué.
`P(sem)`	Demande d'acquisition d'une ressource. Si aucune ressource n'est disponible, le processus est bloqué. Décrémente le compteur du sémaphore `sem` ⇒ demande un accès "Puis-je ?" (Bloquant) – P : *proboren* décrémenter NL
`V(sem)`	Libération d'une ressource. Si la file d'attente n'est pas vide, un processus est débloqué. Incrémente le compteur du sémaphore `sem` ⇒ libère un accès "Vas-y !" : Reveil (non Bloquant) – V : *verhogen* incrémenter NL

```
Init (int val, SEM s){
        s.cpt = val;
        s.file = VIDE;
}
```

Remarque. Attention `val >= 0`

Les fonctions `P` et `V` sont des blocs indivisibles de code. Elles ne doivent pas être interrompues.

```
P(SEM s){
    s.cpt --;
    if(s.cpt < 0){
        inserer(processus_courant, s.file);
        sleep();
    }
}
```

```
V(SEM s){
    s.cpt ++;
    if(s.cpt <= 0){
        p = retirer(s.file);
        wakeup(p);
    }
}
```

A propos de la valeur `cpt` :
– `cpt >= 0` : `cpt` correspond au nombre de processus autorisés à accéder à la section critique ;
– `cpt < 0` : `|cpt|` est le nombre de processus bloqués dans la file.

5.4 Problèmes de synchronisation classiques

5.4.1 Exclusion mutuelle

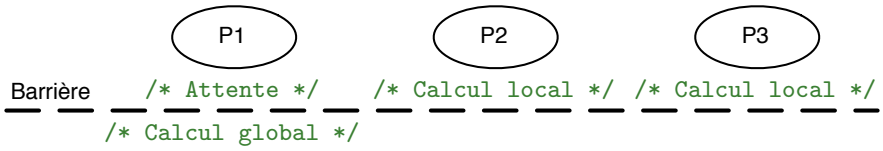
Définition (Exclusion mutuelle)**.** Autoriser un seul processus à la fois à accéder à la section critique.

Soit un sémaphore `MUTEX` initialisé de la façon suivante : `init(MUTEX, 1);`

```
EnterSC() P(MUTEX);
          /* Section Critique */
SortirSC() V(MUTEX);
```

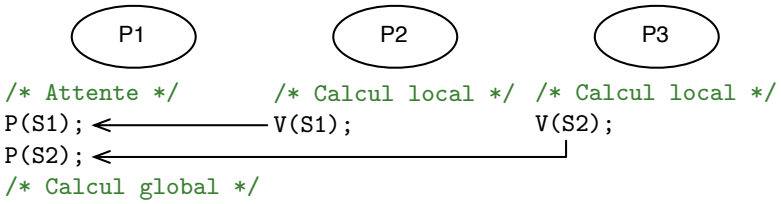
5.4.2 Barrière de synchronisation

Définition (Barrière de synchronisation)**.** Un processus attend les autres à un point de synchronisation : la barrière.

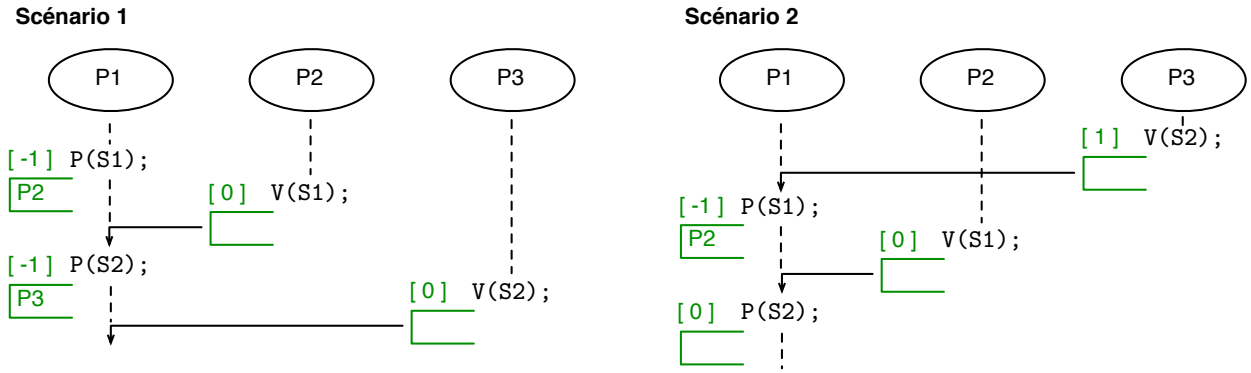


Considérons deux sémaphores :
– Un sémaphore S1 pour attendre le processus P2 ;
– Un sémaphore S2 pour attendre le processus P3.

```
init(S1, 0); init(S2, 0);
```

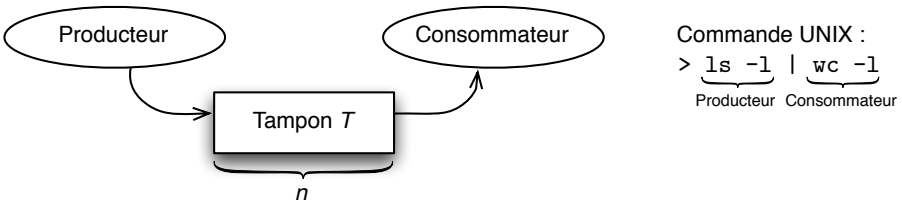


Remarque. Il y a un compteur propre à chaque sémaphore. En général, il faut autant de `P` que de `V`.

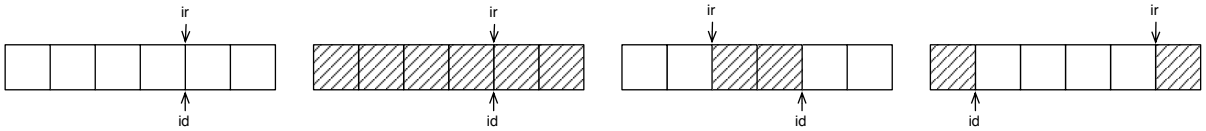


5.4.3 Producteur / consommateur

Considérons un tampon T partagé avec N cases. Rôles des processus :
- Processus producteurs : produisent dans T ;
- Processus consommateurs : consomment dans T.

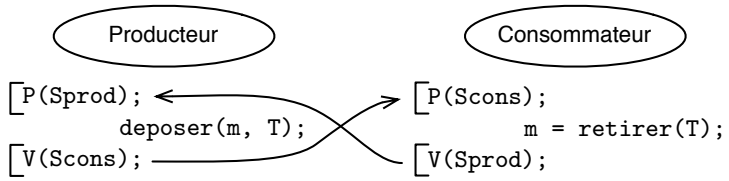


Gestion des indices `ir` (commun aux consommateurs) et `id` (commun aux producteurs) liés au tampon.



Les producteurs sont bloqués si le tampon est plein ⇒ un sémaphore `Sprod` est alors nécessaire pour bloquer les producteurs : `init(Sprod, 1);`

Les consommateurs sont bloqués si le tampon T est vide ⇒ un sémaphore `Scons` est alors nécessaire pour bloquer les consommateurs : `init(Scons, 0);`



Les variables `id` et `ir` sont partagées. De plus, nous souhaitons avoir une exécution la plus parallèle possible.

Considérons alors les sémaphores suivants :
- Les sémaphores `MUTEXP` et `MUTEXC` d'exclusion mutuelle pour respectivement les processus producteurs et les processus consommateurs, initialisés à 1 : `init(MUTEXP, 1)` et `init(MUTEXC, 1)` ;
- Le tableau de sémaphores `C[1..n]` de taille `n` et initialisés à 1 : `for(i=0; i<n; i++)init(C[i], 1);`. Avec ce tableau, un producteur en train de produire un message n'empêche pas d'autres producteurs d'acquérir des cases vides. De plus, il permet de garantir qu'un producteur ne produira pas dans une case en train d'être consommée.

Code des fonctions `deposer()` et `retirer()` satisfaisant aux conditions liées au problème.

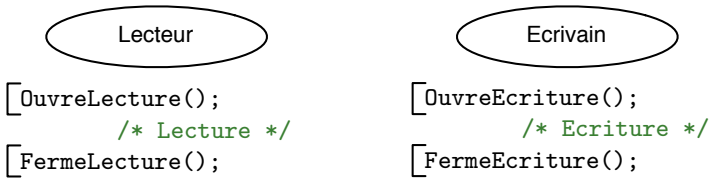
```
void deposer(message * m) {
    int i;
    P(MUTEXP);
        i = id; id = (id+1) % n;
        P(C[i]);
    V(MUTEXP);
    T[i] = m;
    V(C[i]);
}
```

```
message * retirer() {
    int i;
    P(MUTEXC);
        i = ir; ir = (ir+1) % n;
        P(C[i]);
    V(MUTEXC);
    message * m = T[i];
    V(C[i]);
    return m;
}
```

5.4.4 Lecteur / écrivain

Un fichier est partagé par des processus lecteurs et écrivains.

Contraintes. N lecteurs et 0 écrivain ou (XOR) un écrivain et 0 lecteur.



Le premier lecteur verrouille le fichier en écriture et le dernier lecteur déverrouille le fichier en écriture
⇒ Variable partagée comptant le nombre de lecteurs en train d'accéder au fichier : `int nblect = 0;`

Sémaphores nécessaires :
- Un `MUTEX` pour protéger la variable `nblect` : `init(MUTEX, 1);`
- Un sémaphore `E` pour bloquer les écrivains (un écrivain à la fois peut accéder au fichier) : `init(E, 1);`

Problème. Il existe un risque de famine pour l'écrivain.

```
OuvreLecture() {
    P(MUTEX);
    nblect++;
    if(nblect == 1) P(E);
    V(MUTEX);
}
```

```
FermeLecture() {
    P(MUTEX);
    nblect--;
    if(nblect == 0) V(E);
    V(MUTEX);
}
```

```
OuvreEcriture() {
    P(E);
}
FermeEcriture() {
    V(E);
}
```

Utilisation d'une file (FIFO) pour assurer l'équité dans l'accès des processus à la section critique : lors de leur arrivée, tous les processus sont rangés dans une même file. Utilisation d'un sémaphore `FILE` tel que `init(FILE, 1)`.
- Extraction d'un écrivain ;
- Extraction du premier lecteur d'un groupe : autoriser les lecteurs suivant dans la file jusqu'à ce que le processus en tête de file soit un écrivain.

```
OuvreLecture() {
    P(FILE);
        P(MUTEX);
        nblect++;
        if(nblect == 1) P(E);
        V(MUTEX);
    V(FILE);
}
```

```
FermeLecture() {
    P(MUTEX);
    nblect--;
    if(nblect == 0) V(E);
    V(MUTEX);
}
```

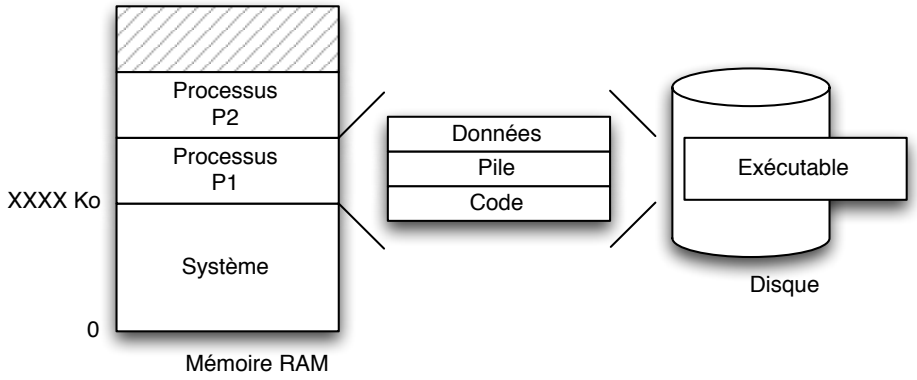
```
OuvreEcriture() {
    P(FILE);
    P(E);
}
FermeEcriture() {
    V(E);
    V(FILE);
}
```

Chapitre 6

Gestion de la mémoire

6.1 Introduction

Organisation de la mémoire.

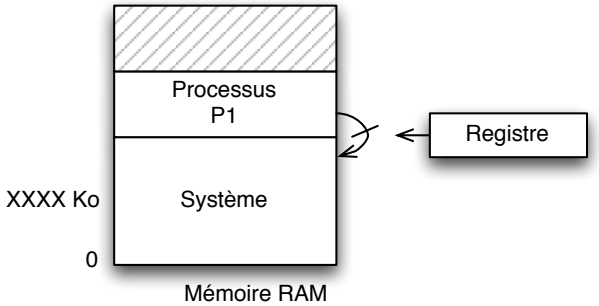


6.2 Mémoire linéaire

Avec la mémoire linéaire, les processus occupent des espaces contigus en mémoire. La totalité du programme et les données d'un processus doivent être chargées en mémoire pour pouvoir exécuter le processus.

6.2.1 Mono-programmation

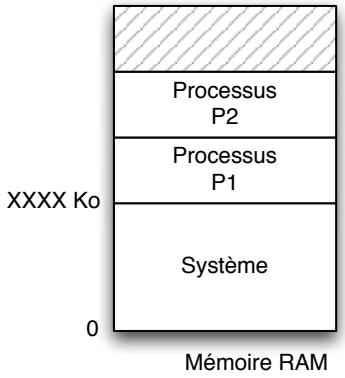
En mono-programmation, il n'y a qu'un seul processus en mémoire. Un registre barrière est testé par le processeur (CPU) pour vérifier, à chaque accès mémoire, la validité des adresses soumises au processeur.



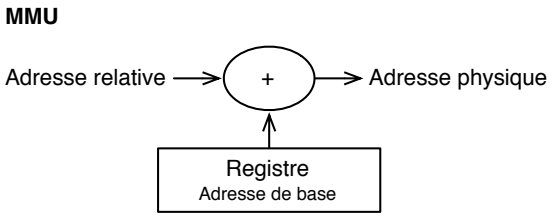
La partie du processeur qui gère la mémoire est la MMU (*Memory Management Unit*).

6.2.2 Multi-programmation

En multi-programmation, plusieurs processus partagent la mémoire. Les processus peuvent être relogeables (ie. Leur emplacement peut changer d'une exécution à une autre).



De ce fait, les adresses sont relatives par rapport au début du processus. Il existe donc des mécanismes de traduction d'adresses relative en adresse physique (traduction effectuée par la MMU).

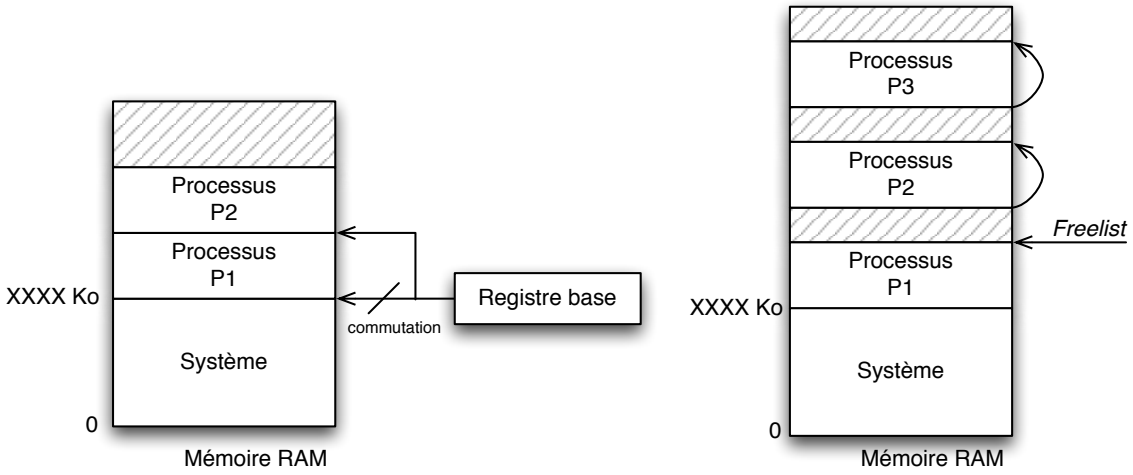


Traduction relative ⇒ physique par le CPU (MMU) Positionnement registre de base fait par le système à la commutation de processus

6.2.3 Allocation mémoire

L'allocation mémoire est gérée par le système.

Objectif. Trouver un emplacement contigu en mémoire au moment de la création d'un processus.
⇒ Le système maintient donc une liste des emplacements libres (*freelist*).

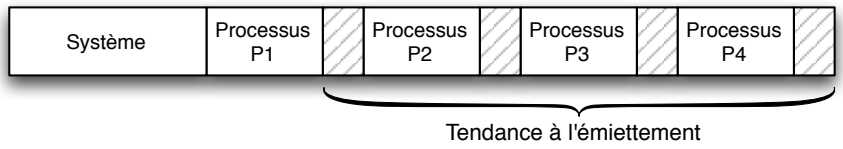


Il existe trois stratégies d'allocation :

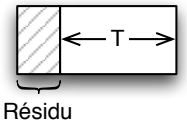
First fit Choisir dans la liste le premier emplacement qui convient (taille > taille processus alloué).

Avantages / inconvénients :

- Simple ;
- Tendance à l'émiettement au début de la mémoire (miette : emplacement trop petit) ;
- Ne minimise pas le nombre de trous (emplacements).



Best fit Choisir l'emplacement qui convient le mieux (l'emplacement qui minimise le résidu).



Avantages / inconvénients :

- Meilleure utilisation de la mémoire, moins de trous, *freelist* plus petite) ;
- Plus complexe : parcourir toute la liste ;
- Tendance à un fort émiettement (si taille du résidu > 0).

Worst fit Choisir l'emplacement le plus grand ⇒ Maximiser le résidu.

Avantages / inconvénients :

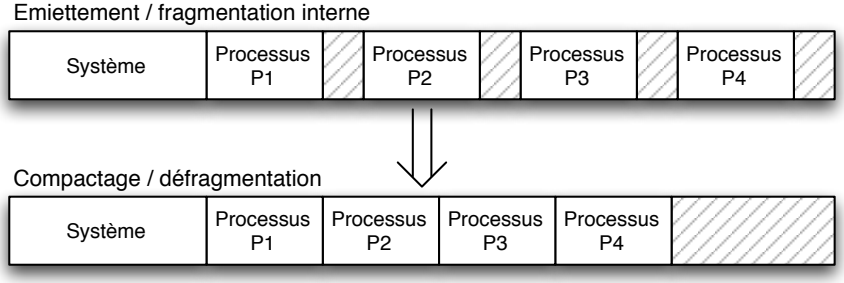
- Moins de miettes ;
- Plus de trous dans la mémoire.

Brenchmarks : les stratégies *First fit* et *Best fit* sont les meilleures (*First fit* > *Best fit* >> *Worst fit*).

Définition (Fragmentation interne)**.** Si la taille d'un fichier n'est pas un multiple de la taille d'un bloc (ie. `taille_fichier % taille_bloc != 0`), alors le système de fichiers allouera un nouveau bloc qui ne sera pas rempli entièrement, gaspillant ainsi de l'espace disque en provoquant de l'émiettement.

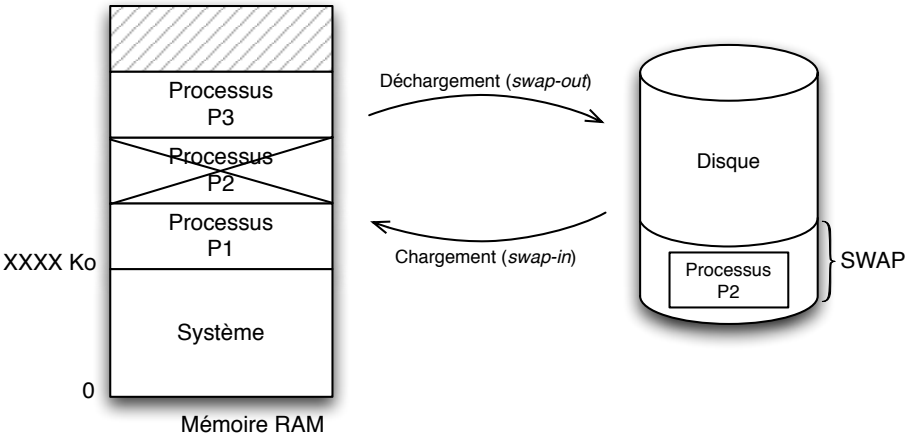
Définition (Fragmentation externe)**.** Fait d'avoir les blocs constituant le fichier répartis de façon non contiguë sur tout le disque, réduisant ainsi les performances d'accès à ce fichier, du fait de l'augmentation des mouvements des têtes de lecture du disque nécessaire à l'accès à ce fichier.

Problème 1.1. Si trop d'émiettement (quelle que soit la stratégie utilisée), alors l'allocation n'est plus possible. Il faut alors réorganiser la mémoire avec un compactage / défragmentation :



Problème 1.2. Le nombre de processus limité est directement proportionnel à la capacité de la mémoire physique.

Idée. Définir une zone de disque particulière comme extension de la mémoire (beaucoup plus lent : rapport de 100 000 à 1 000 000) ⇒ zone de swap.



6.2.4 Stratégie de *swap*

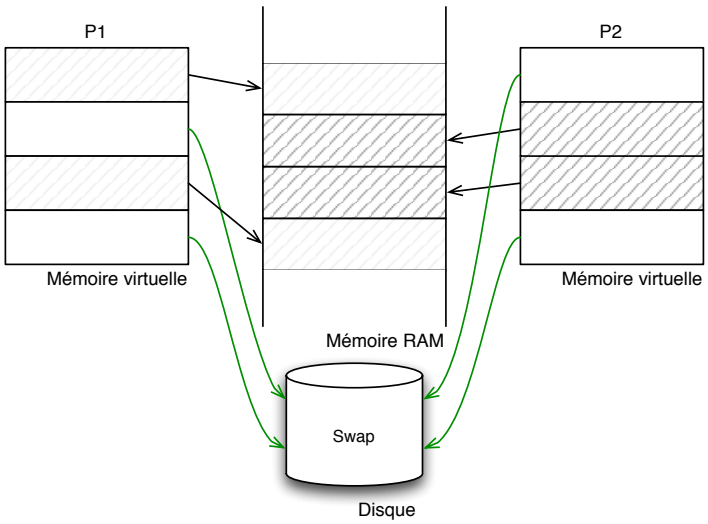
***Swap-out* (déchargement)**

- Intervient en cas de saturation mémoire ;
- Choisir un processus "victime" selon les critères suivants :
  - Un processus à l'état **bloqué** ;
  - Durée à l'état bloqué (un processus bloqué depuis longtemps) ;
  - Taille (gros processus en priorité).

***Swap-in* (chargement)** Dès qu'un processus "swappé" redevient prêt, il est rechargé en mémoire. Il y a éventuellement des déchargements de d'autres processus bloqués.

6.3 Mémoire virtuelle

Idée. Présence non contiguë et partielle des processus en mémoire. La traduction adresse virtuelle ⇒ adresse physique est effectuée par la MMU (*Memory Management Unit*) qui se trouve au niveau du processeur.



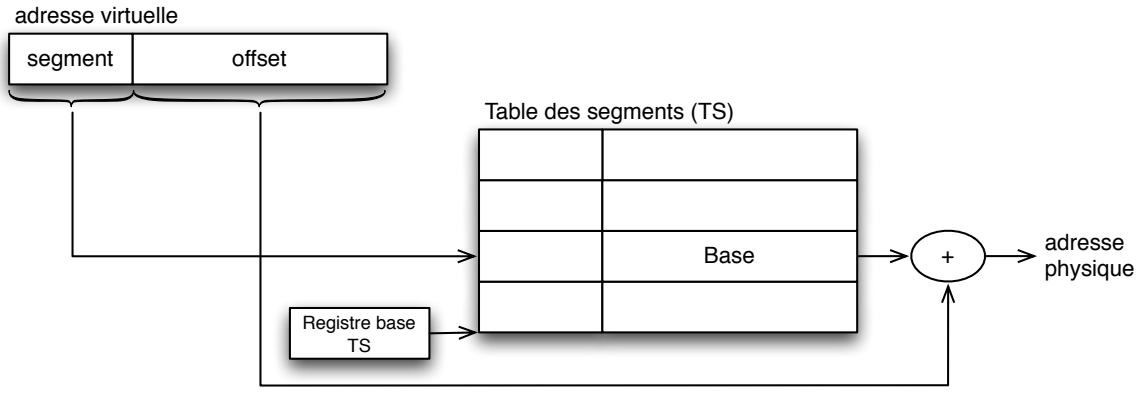
6.3.1 Mémoire segmentée

Les processus sont découpés en **segments**. Un segment représente une zone mémoire pour la tâche. Il est associé à un aspect spécifique de la représentation de la tâche en mémoire (eg. code, données, pile, etc.). Un segment est chargé entièrement dans les zones de mémoire contiguës.

Définition (Segment)**.** Une suite d'emplacements **consécutifs** en mémoire et de **taille variable**.

Exemple. Il existe trois segments :
– Un segment code (`.text`) ;
– Un segment pile (`.stack`) ;
– Un segment données (`.data`).

La traduction nécessite une table des segments par **processus**.

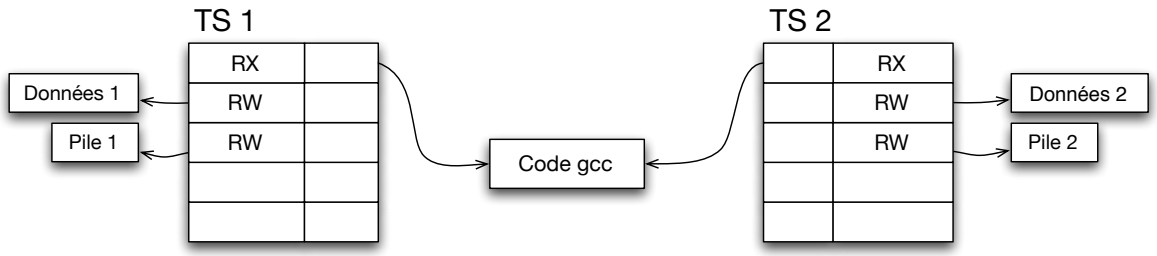


Entrée de la table des segments se présente sous la forme : `<P, Droits, Taille, Base>` :
– `P` : présence (0 : non présent, 1 : présent) ;
– `droits` : R (*Read*), W (*Write*), X (*eXecutable*).

La MMU (processeur) vérifie pour un couple `<segment, offset>` :
– `Presence == 1 ?` ⇒ 0 ⇒ IT défaut de segment (système) ;
– `Droits OK ?` ⇒ Problème ⇒ violation segment (système) ;
– `offset > taille` ⇒ Non ⇒ IT débordement mémoire (système) ;
– Traduction (matériel).

Avantages / inconvénients de la mémoire segmentée :

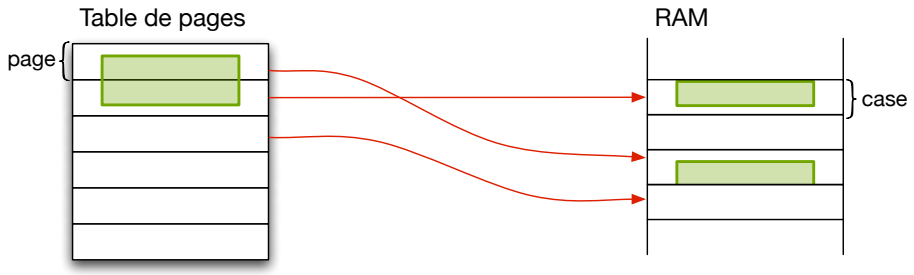
- Partage de segment en lecture.
  Exemple. Les codes P1, P2 partagent `gcc` :



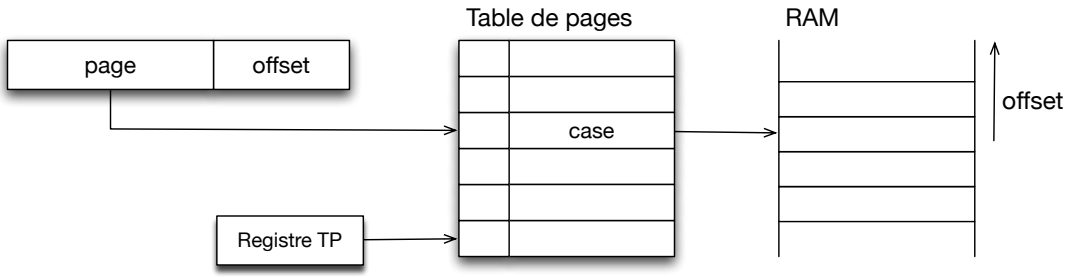
- Allocation des segments : trouver un emplacement consécutif en mémoire physique (*First fit*, *Best fit*, *Worst fit*).
  ⇒ Problème de fragmentation + compactage.

6.3.2 Mémoire paginée

Mémoire virtuelle (espace virtuel) découpée en **pages** de **tailles fixes**.
Mémoire physique découpée en cases (*frames*) Taille page = taille case.



La traduction adresse virtuelle ⇒ adresse physique est effectuée par la MMU.



Remarque. La taille de page doit être une puissance de 2.
L'adresse physique est égale à `case * taille`.

Exemple. Soit une taille de page égale à 512 octets et une adresse virtuelle de base égale à 2 050 :
– `page` = 2 050 ÷ 512 = 4 ;
– `offset` = 2 050 mod 512 = 2.

Entrée de la table des pages : `<Presence, droits, evincee, modifiee, Case>`
– `Present` : 0 : non présent, 1 : présent ;
– `droits` : RWX ;
– `evincer` : 0 : non verrouillée, 1 : verrouillée ;
– `modifiee` : 0 : non modifiée, 1 : modifiée.
Une version abrégée : `<P, droits, V, M, Case>`

Remarque. Les champs sont dépendant de l'architecture du processeur.

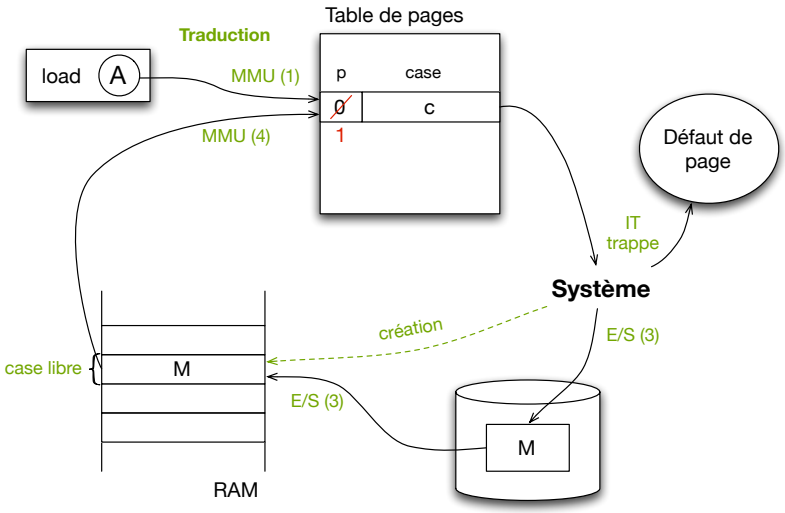
Les pages sont chargées **à la demande** au fur et à mesure des accès. Accès à une page avec `P = 0` ⇒ MMU lève l'interruption "défaut de page" (*page fault*).

Le système maintient une liste de cases libres.
⇒ Lorsque le nombre de cases libre est trop faible, le système doit libérer des cases en déchargeant une partie des cases ramenées en mémoire sur disque, dans la zone de swap.

6.3.3 Remplacement des pages

Permet de réguler l'activité mémoire. Ce mécanisme intervient en cas de pénurie de mémoire.

Objectif. Il y a deux objectifs à atteindre :
– Choisir une page (victime) à évincer sur le disque, dans la zone de swap (si la page a été modifiée) ;
– Utiliser la case libérée.



Algorithmes de remplacement de page

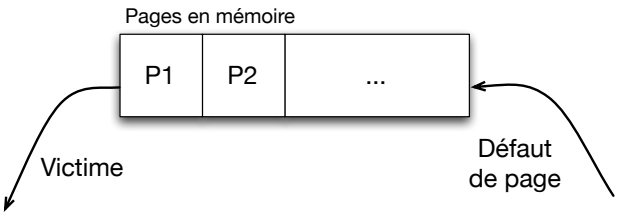
Objectif. Choisir une bonne victime en minimisant le nombre de défauts de page.

Il existe trois types d'algorithme.

Optimal (théorique) : choisir la page qui sera utilisée le plus tardivement. ⇒ Connaissance du futur.

FIFO Choisir la page la plus vieille.

- Simple à implémenter :



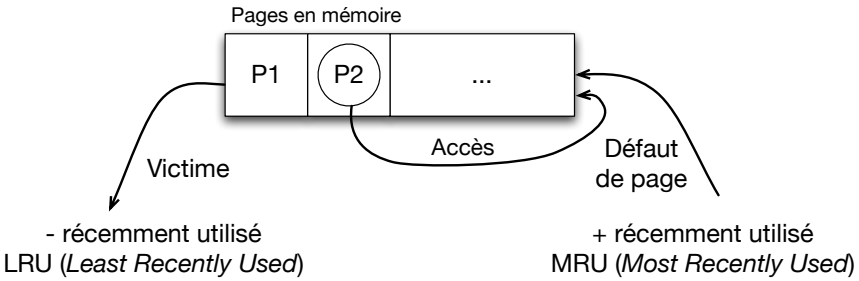
- Peu performant car on ne prend pas en compte l'utilisation des pages.

LRU (*Least Recently Used*) Choisir la page la moins récemment utilisée.

Idée. Utiliser la propriété de **localité temporelle** : une page utilisée il y a longtemps a peu de chance d'être utilisée dans un futur proche.

⇒ Notion de *working set* : ensemble des pages les plus utilisées.

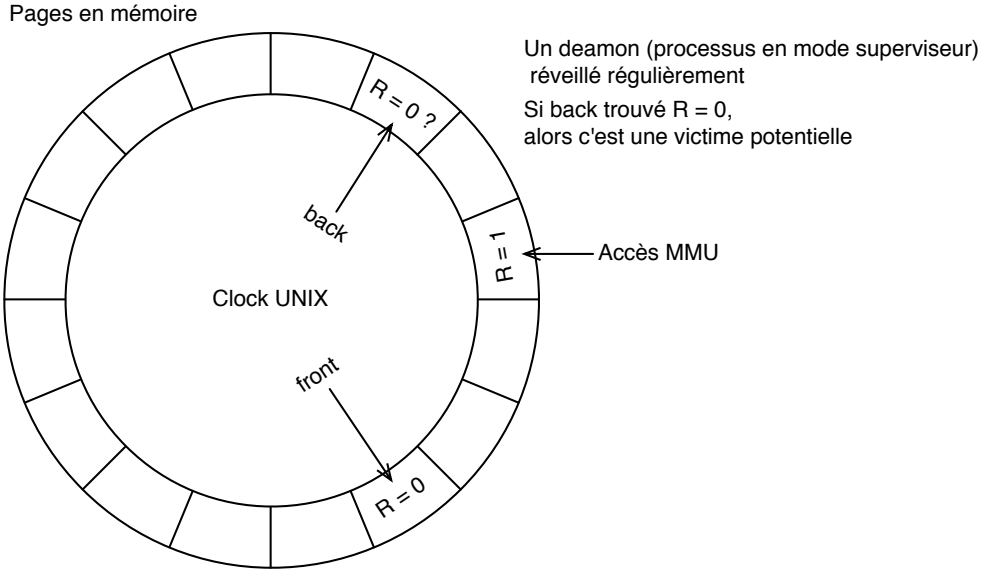
- Bon choix (défauts de pages faibles) ;
- Très (trop) coûteux à implémenter : nécessité de tracer tous les accès en mémoire.



Il existe une approximation LRU en NRU (*Not Recently Used*)
⇒ Ajout de bits supplémentaires dans la table des pages :
– *Reference bit* (`R`) positionné à 1 par la MMU à chaque accès de page (en lecture ou en écriture) ;
– *Acces bit* (`A`) positionné à 1 par la MMU à chaque modification de la page concernée (écriture).

Principe (NRU – *Not Recently Used*)**.**
– Le système re-positionne régulièrement les bits `R` à 0 ;
– `R` testés plus tard, si `R` toujours à 0 ⇒ page plus récemment utilisée.

Exemple. Clock (UNIX)

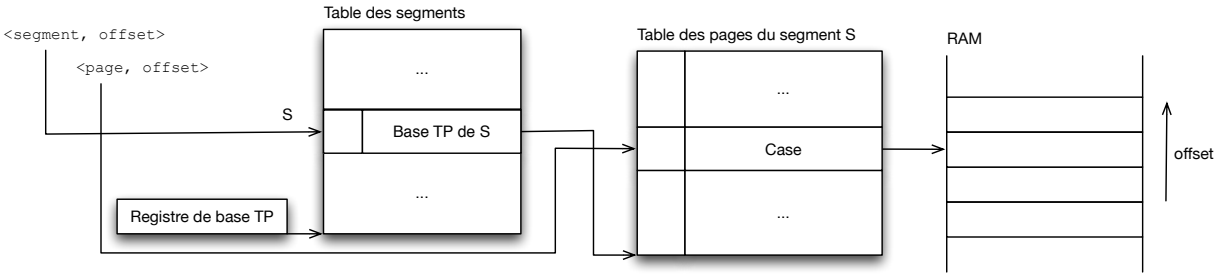


Deux types de mémoire pour l'instant :
– **Mémoire paginée** : espace d'adressage du processus découpé en blocs (pages) de taille fixe.
– **Mémoire segmentée** : espace d'adressage du processus découpé en blocs (segments) de tailles variables.

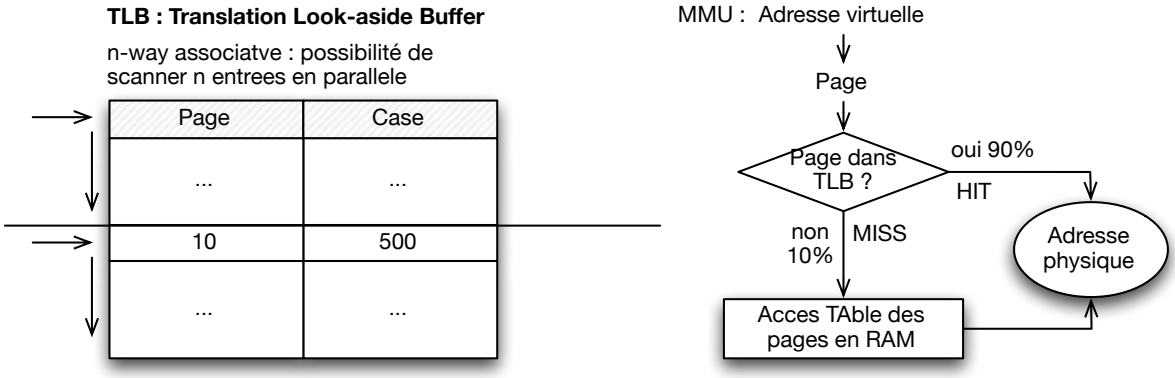
6.4 Mémoire segmentée paginée

Idée. Lever la contrainte de contiguïté des segments en mémoire physique. ⇒ Segments découpés en pages.

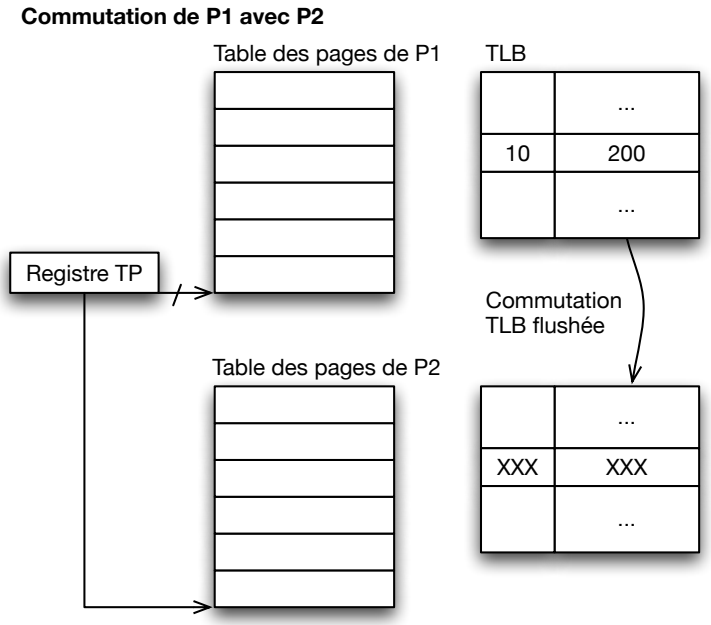
Schéma "classique" : une table des pages par segment :
– Adresse virtuelle définie par : `<segment, offset>` ;
– Déplacement à l'intérieur du segment défini par : `<page, offset>`.


<segment, offset>
<page, offset>
S
Table des segments
...
Base TP de S
...
Registre de base TP
Table des pages du segment S
...
Case
...
RAM
offset

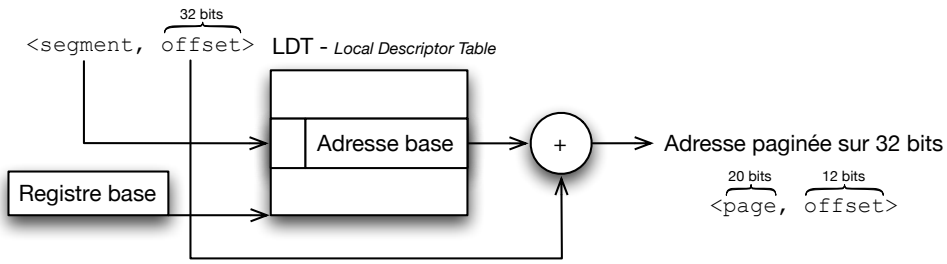
Cache des adresses traduites : TLB (*Translation Look-Aside Buffer*) Mémoire associative interne au processeur (pas en RAM) pour mémoriser les pages déjà traduites.



- TLB consultée à chaque traduction ;
- Mise à jour de la TLB :
  - Nouvelle page ;
  - Vidage (flush) des caches à chaque commutation de processus (pas le cas des threads).

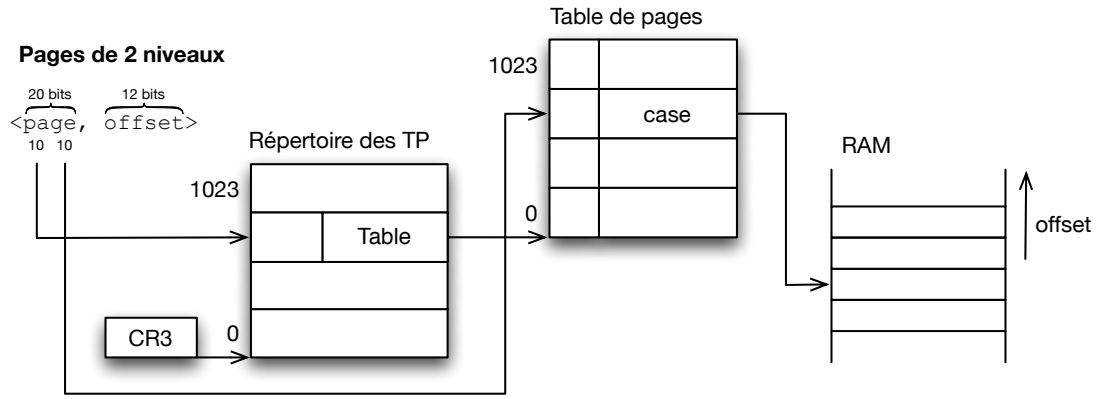


Exemple. Dans le processeur Intel 32 bits, la mémoire est **segmenté paginée** : une table des segments par processus : LDT (*Local Descriptor Table*).


32 bits
<segment, offset>
LDT - Local Descriptor Table
Adresse base
+
Adresse paginée sur 32 bits
Registre base
20 bits
12 bits
<page, offset>

Les pages ont une taille de 4 Ko, donc l'*offset* est codé sur 12 bits. Puisque l'adresse est codée sur 32 bits, le numéro de page est codé sur 20 bits, ce qui donne lieu à 2^{20} pages, soit plus de 1 million de pages potentielles.
Une table des pages à 1 million d'entrées aurait une taille égale à $2^{20} \times 4 = 2^{22}$ octets = 4 Mo.

Les architectures Intel utilise donc des pages multi-niveaux (2 pour l'architecture 32 bits, jusqu'à 4 niveaux de pages pour les architectures à 64 bits).



Chapitre 7

Entrées / sorties Disque

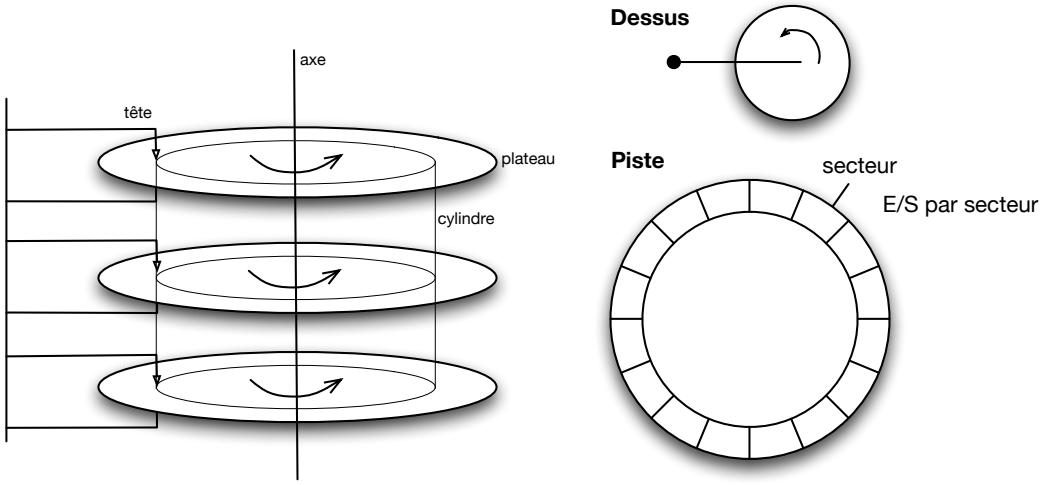
7.1 Définitions

Il existe deux types de périphériques :

Mode bloc (disque) Lecture / Ecriture par bloc de manière indépendante ⇒ Mode adressable.

Mode caractère (console, imprimante, réseau) Pas de structuration du flux ⇒ Accès séquentiel.

Architecture d'un disque



Accès aux données (piste, secteur) :
- Positionnement sur la piste (*seek time*) ;
- Attendre le passage du secteur sous la piste ;
- Transfert.

7.2 Ordonnancement des requêtes d'E/S

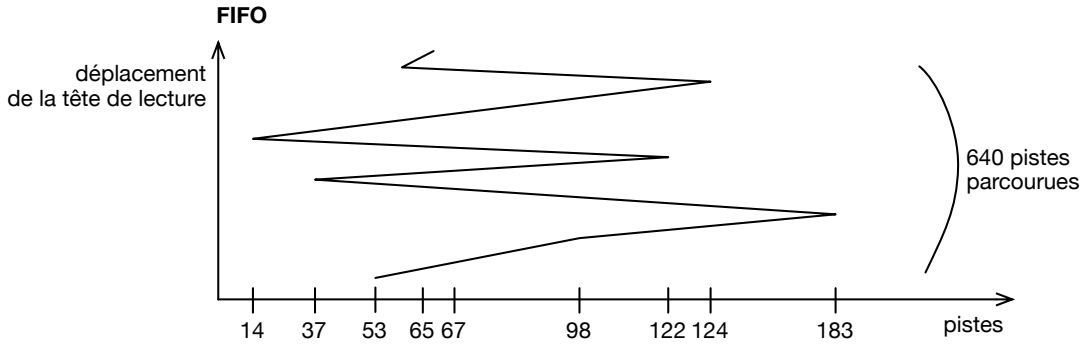
A chaque disque, est associée une file d'attente des requêtes (numéro de piste). Ces files d'attente sont gérées par le système à l'aide des pilotes/drivers.

Objectif. Limiter les déplacements des têtes de lecture des disques durs.

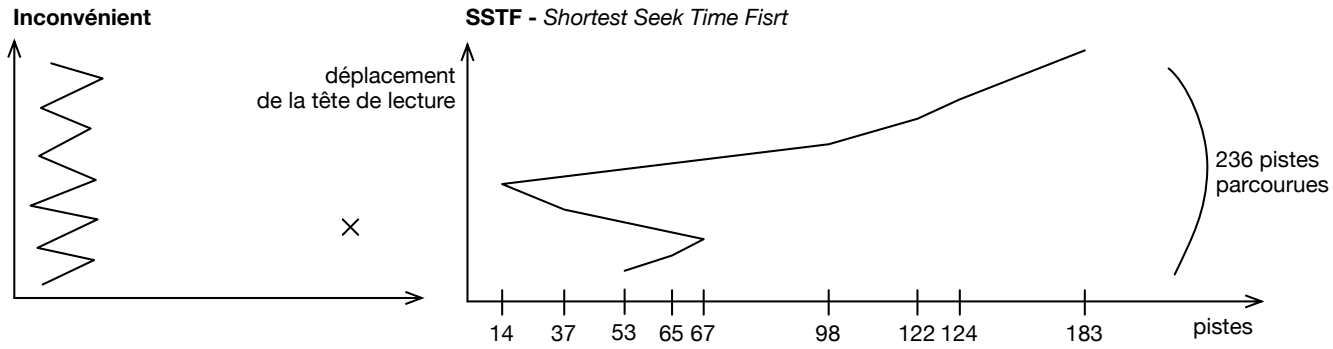
7.2.1 Premier arrivé, premier servi – FIFO / FCFS

Les requêtes sont servies dans l'ordre.

Exemple. Supposons la tête de lecture sur la piste 53.
Contenu de la file d'attente : 98, 183, 37, 122, 14, 124, 65, 67.



7.2.2 Plus court déplacement d'abord – SSTF *Shortest Seek Time First*

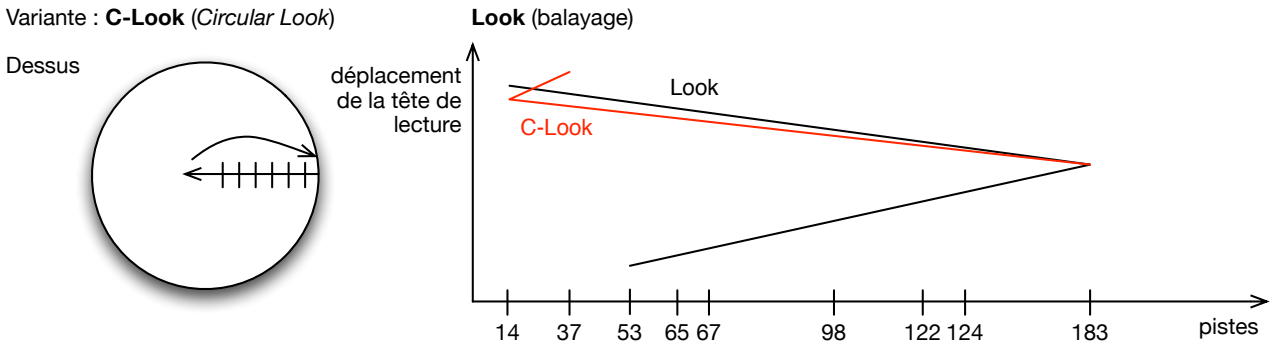


Avantages / inconvénients :

- Performant ;
- Risque de famine des requêtes lointaines.

7.2.3 Algorithme à balayage (idem ascenseurs – Look)

Balayage du disque entre les deux requêtes les plus éloignées. Requêtes servies au passage de la tête de lecture. Il existe une variante : C-Look (*Circular Look*).

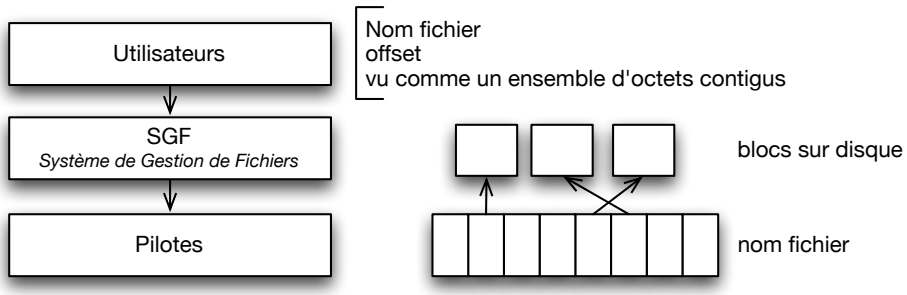


Chapitre 8

Systèmes de gestion de fichiers

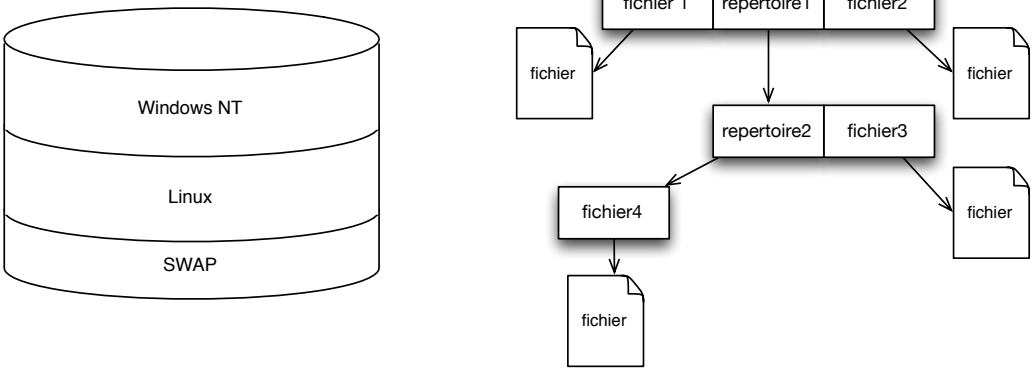
Un système de gestion de fichiers (FS – *file system*) est une façon de stocker les informations et de les organiser dans des fichiers sur ce que l'on appelle des mémoires secondaires (disque dur, CD-ROM, clé USB, SSD, disquette, etc.).

8.1 Introduction



Un Système de Gestion de Fichiers doit pouvoir assumer plusieurs rôles :
- Accès aux données : conversion `<nom, offset>` en bloc sur disque ;
- Gestion de l'espace libre.
- Allocation de la place sur mémoires secondaires : les fichiers étant de taille différente et cette taille pouvant être dynamique, le SGF alloue à chaque fichier un nombre variable de granules de mémoire secondaire de taille fixe (blocs) ;
- Manipulation des fichiers : des opérations sont définies pour permettre la manipulation des fichiers par les programmes d'application, à savoir : créer/détruire des fichiers, insérer, supprimer et modifier un article dans un fichier ;
- Sécurité et contrôle des fichiers : le SGF permet le partage des fichiers par différents programmes d'applications tout en assurant la sécurité et la confidentialité des données.

Un disque est divisé en partitions. Chaque partition du disque possède ses propres structures.
Une partition contient des fichiers regroupés en répertoires ⇒ Ces fichiers et répertoires forment une arborescence.

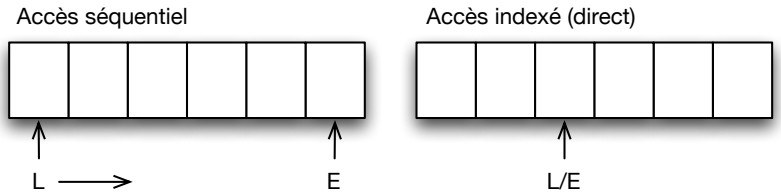


Mode d'accès aux fichiers :

Accès séquentiel

- Lecture d'un flux de données
- Ecriture en fin de fichier

Accès indexé (direct) Accès direct aux blocs

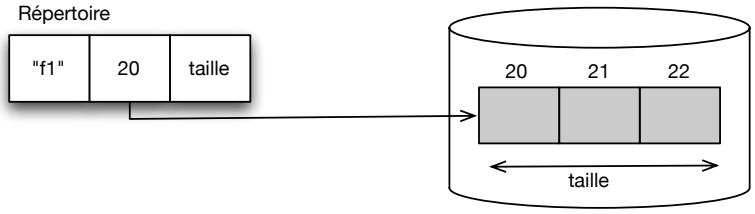


8.2 Politique d'allocation

On veut gérer l'allocation de données sur le disque pour les fichiers utilisateurs. Le disque est structuré en **blocs**. La politique d'allocation des blocs permet de définir la structuration des blocs de données du système de gestion de fichiers.

8.2.1 Allocation contiguë (séquentielle)

Un fichier est un ensemble de blocs sur le disque (pas forcément consécutifs). Si l'on veut optimiser d'accès séquentiel aux fichiers, il faut minimiser les déplacements de la tête de lecture du disque. De ce fait, il faut utiliser un bitmap pour gérer l'espace libre sur le disque afin d'avoir des fichiers alloués sur des blocs contigües.

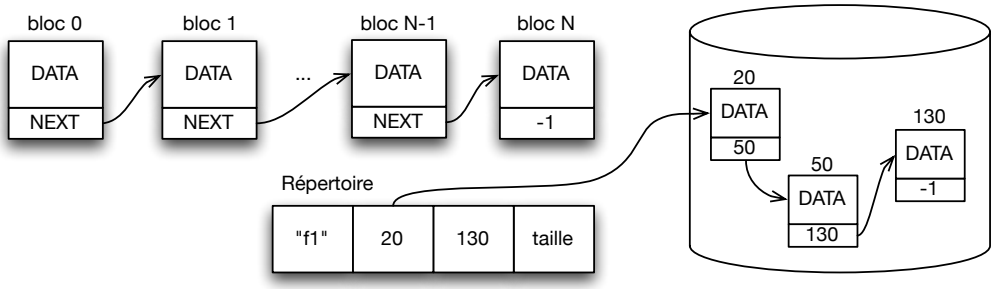


Avantages /inconvénients :

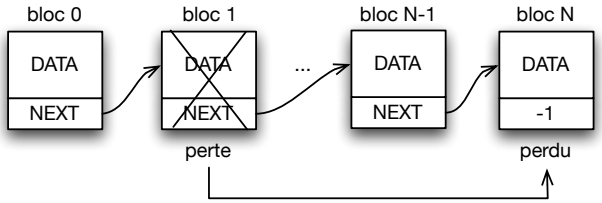
- Simple, performant (déplacement des têtes de lecture du disque dur) ;
- Accès séquentiel direct : facilement implémentable ;
- Allocation : défragmentation régulière ;
- Changement de taille lourd.

8.2.2 Allocation chainée

Un fichier est considéré comme étant une suite de blocs chainés entre eux.



Avantages/inconvénients :
- Allocation simple ;
- Changement de taille ;
- Accès séquentiel ;
- Pas d'accès direct ;
- Fiabilité : parte d'un bloc ⇒ Perte de la fin du fichier.



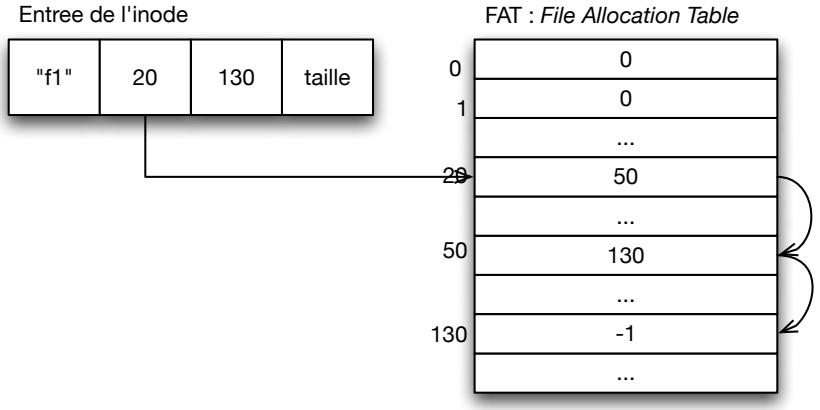
Il existe une variante à l'allocation chaînée : table d'allocation de fichiers ou FAT (*File Allocation Table*).

8.2.3 FAT (File Allocation Table

La FAT (*File Allocation Table*) est une structure de données permettant de gérer à la fois l'espace libre et l'espace alloué. Il s'agit d'un tableau avec une entrée par bloc stocké en début de partition.

Dans un répertoire, on trouve pour chaque fichier le numéro du premier bloc occupé par le fichier. L'entrée correspondante de la FAT contient l'adresse du $2^{\text{ème}}$ bloc occupé, etc.
L'entrée de la FAT correspondant du dernier bloc occupé contient une valeur spéciale "fin de fichier" (EOF – *End Of File*). Les entrées correspondant aux blocs vides sont égales à 0.

En résumé, une entrée par bloc prend pour valeur :
– 0 : bloc libre ;
– -1 : dernier bloc ;
– > 0 : numéro de bloc suivant (chaînage).



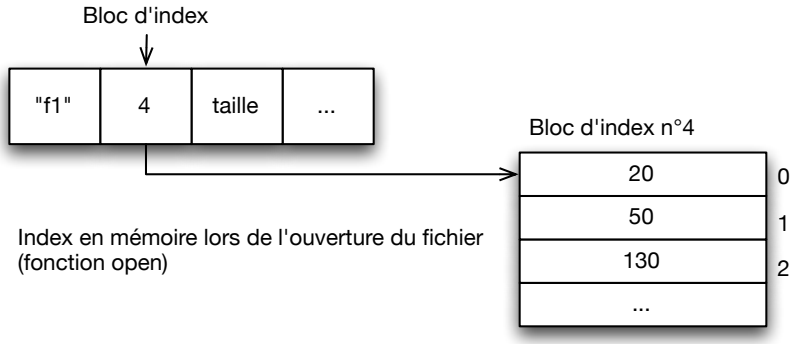
Avantages / inconvénients :
- Accès direct possible (lecture d'entrées de la FAT) ;
- Fiabilité : réplication de la FAT ;
- Accès direct peu performant ;
- Déplacement de la tête de lecture : accès à la FAT en début de partition.

8.2.4 Allocation indexée

L'allocation indexée consiste à un index par fichier contenant la liste des blocs composant le fichier.

Index simple

Un bloc d'index est alloué par fichier.



Avantages / inconvénients :
- Accès séquentiel/direct ;
- Perte d'un bloc par fichier ⇒ Important car beaucoup de petits fichiers ;
- Taille limitée à la capacité du bloc d'index.

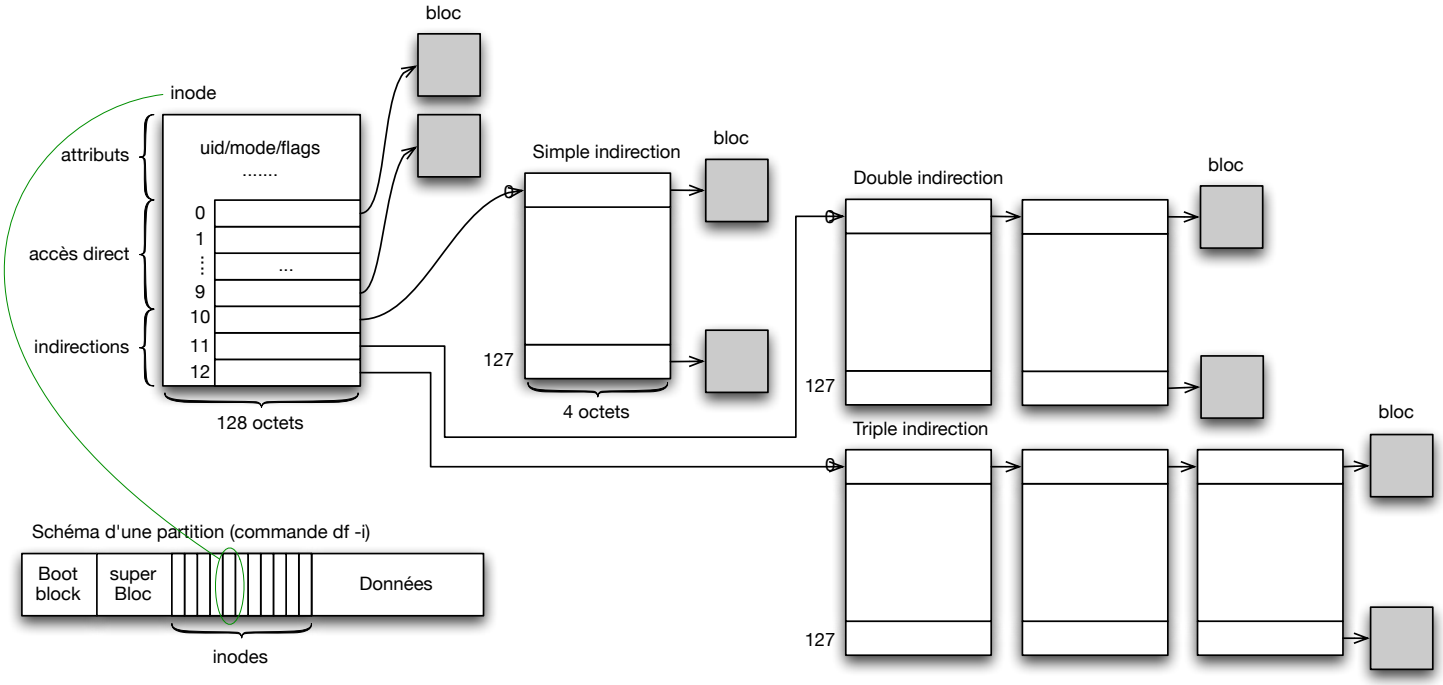
Index multi-niveau (UNIX)

Dans le système de gestion des fichiers d'UNIX, à chaque fichier est associée une structure de données appelée *inode*. Dans cette structure, on trouve une liste de blocs contenant 13 entrées :
– Les 10 premières entrées référencent des blocs de données sur le disque (**accès direct**) ;
– La 11^{e} référence un bloc de contrôle sui référence des blocs de données (**simple indirection**) ;
– La 12^{e} référence un bloc de contrôle qui référence des blocs de contrôle qui référencent des blocs de données (**double indirection**) ;
– La 13^{e} référence un bloc de contrôle qui référence des blocs de contrôle qui référencent des blocs de contrôle qui référencent des blocs de données (**triple indirection**).
Chaque bloc de contrôle permet de référencer au maximum **128** blocs (de contrôle ou de données).
La taille max d'un fichier est alors donnée par la formule :

$$\left(10 + 128 + 128^2 + 128^3\right) \times 512 = 1082201088 \text{ octets} \geqslant 1 \text{ Go}$$

Remarque. La taille d'un bloc de contrôle est de $128 \times 4 = 512$ octets qui était la taille UNIX initiale d'un bloc.

Attributs d'un fichier selon la norme POSIX (métadonnées) :
– La taille du fichier en octets ;
– Identifiant du périphérique contenant le fichier ;
– L'identifiant du propriétaire du fichier (UID) ;
– L'identifiant du groupe auquel appartient le fichier (GID) ;
– Le numéro d'*inode* qui identifie le fichier dans le système de fichier ;
– Le mode du fichier (RWX) ;
– Timestamp pour :
  – La date de dernière modification `ctime` de l'inode (affichée par la commande `stat` ou par `ls -lc`, modification des droits du fichier) ;
  – La date de dernière modification du fichier `mtime` (affichée par le classique `ls -l`) ;
  – La date de dernier accès `atime` (affichée par la commande `stat` ou par `ls -lu`).
– Un compteur indiquant le nombre de liens physiques sur cet inode (`Nlinks`) ;
– Liste des blocs compost le fichier.
Ces métadonnées forment une structure *inode*.



Remarque. Les inodes ne contiennent pas les noms de fichier.

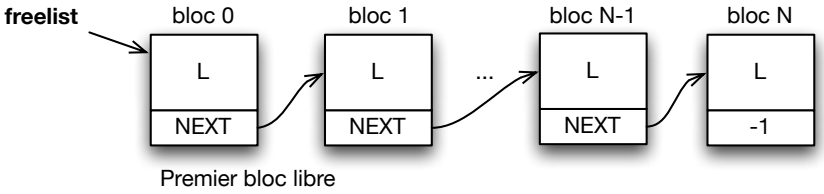
Avantages / inconvénients :
- Accès rapide aux petits fichiers ;
- Gestion des gros fichiers (simple, double, triple indirection) ;
- Accès aux blocs non uniforme.

8.3 Gestion d'espace libre

Objectif. La gestion de l'espace libre (ie. des blocs qui sont libres) permet de trouver ces blocs libres.

8.3.1 Liste chainée

Une *freelist* des blocs libres.



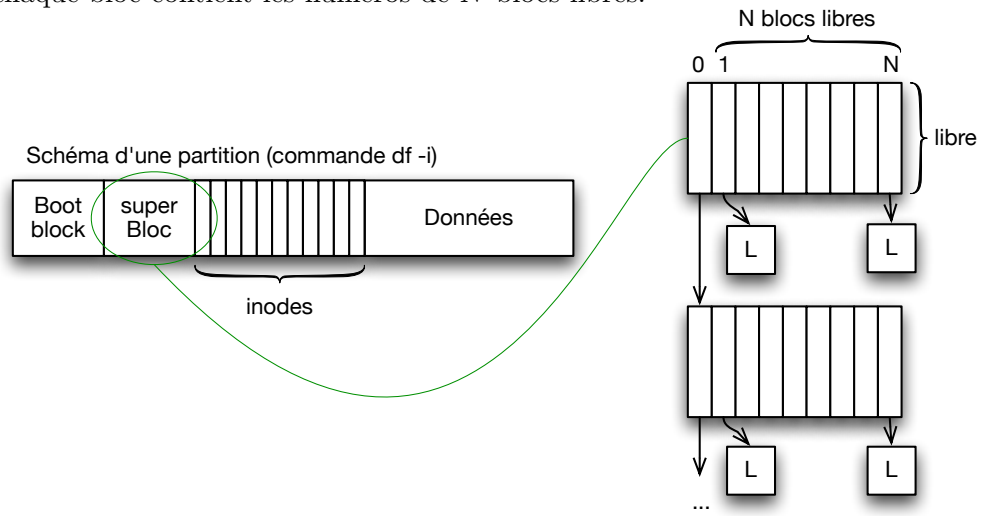
Avantages / inconvénients :
- Simple à implémenter ;
- Fragile ;
- Peu performant pour des demandes groupées.

8.3.2 Regroupement

Un bloc spécial (*superbloc*) contient (entre autres) :
– En première position (indice 0) : le numéro du prochain bloc contenant une liste de blocs libres ;

– De l'indice 1 à l'indice $N+1$: une liste de N blocs libres

De ce fait, chaque bloc contient les numéros de N blocs libres.



Avantages / inconvénients :

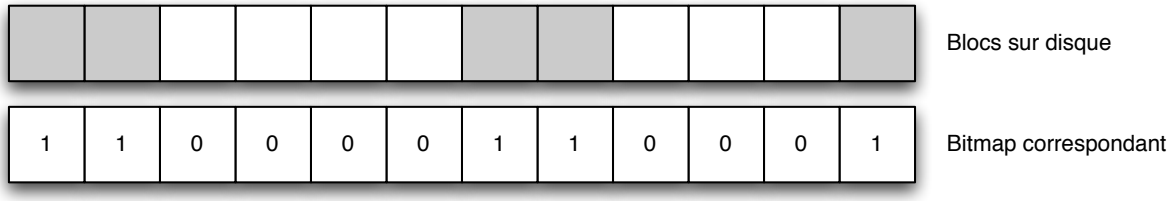
- Fournir rapidement beaucoup de blocs libres ;
- Faible occupation mémoire ;
- Difficile de fournir des blocs libres proches sur le disque ⇒ Fragmentation importante.

8.3.3 Vecteur linéaire (bitmap)

Une entrée par bloc codée sur 1 bit :

– 0 : bloc libre ;

– 1 : bloc occupé.



Allocation de N blocs pour un même fichier : trouver une séquence de N zéros dans le bitmap. Il existe plusieurs stratégies possibles : *Best Fit*, *First Fit*, *Worst Fit*...

Avantages / inconvénients :

- Limite la fragmentation : il est possible d'allouer facilement n blocs contigües (il suffit de maximiser le nombre de 0 consécutifs) ;
- Taille : importante pour les grosses partitions.