

# Systemes informatiques et applications concurrentes

## Partie 2 - Processus et ordonnancement

Ivan KURZWEG

6 mars 2010

**Systemes informatiques et applications concurrentes**

by Ivan KURZWEG

Copyright © 2010 Ivan KURZWEG, [ivan.kurzweg@gmail.com](mailto:ivan.kurzweg@gmail.com), 2010

Permission to use, copy, modify, and distribute this documentation *for any purpose with or without fee* is here by granted, provided that *the above copyright notice and this permission notice appear in all copies*.

The documentation is provided "as is" and the author disclaims all warranties with regard to this documentation including all implied warranties of merchantability and fitness. In no event shall the author be liable for any special, direct, indirect, or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this documentation.

## Table des matières

<b>1</b>	<b>Notions de processus et de <i>Threads</i></b>	<b>1</b>
1.1	Définitions . . . . .	1
1.2	Etats des processus . . . . .	1
1.3	Bloc de contrôle des processus . . . . .	2
1.4	<i>Threads</i> . . . . .	2
<b>2</b>	<b>Ordonnancement des processus</b>	<b>2</b>
2.1	Critères d'ordonnancement . . . . .	2
2.2	Principes . . . . .	3
2.3	Algorithmes d'ordonnancement . . . . .	3
2.3.1	FIFO - <i>First In First Out</i> . . . . .	4
2.3.2	SJF - <i>Shortest First Job</i> . . . . .	5
2.3.3	SR - <i>Shortest Remaining Time</i> . . . . .	5
2.3.4	RR - <i>Round Robin</i> . . . . .	5
2.3.5	Avec priorité . . . . .	6
2.3.6	MQS - <i>Multilevel Queue Scheduling</i> . . . . .	6
2.3.7	MFQ - <i>Multilevel Feedback Queues</i> . . . . .	7
<b>3</b>	<b>Implémentation sous <i>Unix</i> et <i>FreeBSD</i></b>	<b>8</b>
3.1	Processus . . . . .	8
3.1.1	Hiérarchie de processus . . . . .	8
3.1.2	Création des processus . . . . .	9
3.1.3	Terminaison des processus . . . . .	10
3.2	Etats du processus sous <i>Unix</i> . . . . .	12
3.3	<i>Threads</i> . . . . .	13
3.4	L'ordonnancement sous <i>FreeBSD 8.0</i> . . . . .	14
3.4.1	Le 4BSD Scheduler . . . . .	14
3.4.2	ULE Schedule . . . . .	15
<b>4</b>	<b>Exercices</b>	<b>15</b>
4.1	Processus . . . . .	15
4.1.1	Exercice 1 . . . . .	15
4.1.2	Exercice 2 . . . . .	15
4.2	Ordonnancement . . . . .	16
4.2.1	Exercice 1 (source : Joelle Delacroix - Cnam) . . . . .	16
4.2.2	Exercice 2 (source : Christian Carrez - Cnam) . . . . .	16
<b>5</b>	<b>Références</b>	<b>17</b>
5.1	Références . . . . .	17

## Table des figures

1	Etats d'un processus . . . . .	1
2	Commutations . . . . .	3
3	Cycle de vie des processus en ordonnancement préempté . . . . .	4
4	FIFO . . . . .	4
5	SJF . . . . .	5
6	RR . . . . .	5
7	Avec priorité . . . . .	6
8	MQS . . . . .	7
9	MFQ . . . . .	7
10	Etats d'un processus . . . . .	13
11	Chronogramme . . . . .	17

### Résumé

Partie 2 du cours "Systèmes informatiques et applications concurrentes". Cette seconde partie du cours précise la notion de processus abordée en première partie du cours.

## 1 Notions de processus et de *Threads*

Nous avons vu dans la première partie de cours que le processeur pouvait exécuter une séquence d'instructions d'un programme, puis une séquence d'instructions d'un autre programme, avant de revenir au premier, et ainsi de suite. Si la durée d'exécution des séquences est suffisamment réduite, les utilisateurs ont la sensation que l'ensemble des programmes s'exécute en parallèle : on parle de *pseudo-parallélisme*. La gestion du passage d'un programme en exécution à un autre (*commutation*), implique le concept de *processus*, que nous allons approfondir dans ce chapitre.

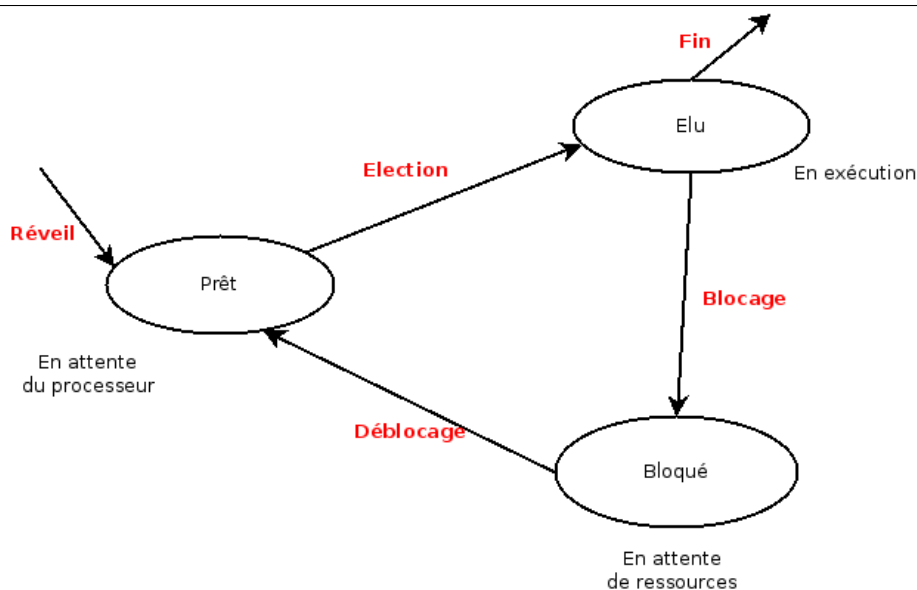
### 1.1 Définitions

- Un processus est un *programme en cours d'exécution* auquel est associé un environnement processeur et un environnement mémoire appelés contexte du processus.
- Un processus est l'instance dynamique d'un programme et incarne le fil d'exécution de celui-ci dans un *espace d'adressage protégé* (objets propres : ensemble des instructions et données accessibles)
- Un programme *réentrant* est un programme pour lequel il peut exister plusieurs processus en même temps

### 1.2 Etats des processus

Nous l'avons vu dans la première partie du cours, il est nécessaire de *bloquer* un processus quand il est en attente de ressources. Le processeur doit donc redevenir libre pour un autre processus, qui était auparavant bloqué. Le cycle de vie d'un processus peut donc s'illustrer par le schéma suivant :

FIG. 1 Etats d'un processus



Un processus est toujours créé dans l'état *prêt*. Lorsqu'il obtient le processeur, il est dans l'état *élu*, c'est l'exécution de ses instructions. Lors de cette exécution, le processus peut demander à accéder à une ressource qui n'est pas immédiatement disponible : le processus ne peut pas poursuivre son exécution tant qu'il n'a pas obtenu la ressource. Le processus quitte alors le processeur et passe dans l'état *bloqué*. L'état bloqué est l'état d'attente d'une ressource autre que le processeur. Dès que la ressource est obtenue, le processus est donc à même de reprendre son exécution. Il est dans l'attente du processeur, de nouveau l'état *prêt*. Le passage de l'état *prêt* vers l'état *élu* constitue l'*opération d'élection*. Le passage de l'état *élu* vers l'état *bloqué* est l'*opération de blocage*. Le passage de l'état *bloqué* vers l'état *prêt* est l'*opération de déblocage*.

Nous verrons que ce schéma peut être enrichi, avec des états supplémentaires liés à la gestion de la mémoire et à l'exécution d'instructions particulières.

### 1.3 Bloc de contrôle des processus

Le système d'exploitation stocke les informations relatives à un processus dans un bloc de contrôle, qui va permettre la sauvegarde et la restauration du contexte mémoire et processeur lors des opérations de *commutations de contexte*. Le bloc de contrôle d'un processus (PCB) contient les informations suivantes :

- *un identificateur unique du processus* (un entier) : le PID
- *l'état courant du processus* (élu, prêt, bloqué)
- *le contexte processeur du processus* : la valeur du compteur ordinal ("à quelle instruction il est arrivé"), la valeur des autres registres du processeur
- *le contexte mémoire* : ce sont des informations mémoire qui permettent de trouver le code et les données du processus en mémoire centrale
- des informations diverses de comptabilisation pour les statistiques sur les performances système
- des informations liées à l'ordonnancement du processus. Le PCB permet la sauvegarde et la restauration du contexte mémoire et du contexte processeur lors des opérations de commutations de contexte. Nous verrons ces concepts plus en détail dans le chapitre suivant.

### 1.4 Threads

Le processus léger, *Thread*, constitue une extension du modèle de processus. Un processus classique est constitué d'un espace d'adressage avec un seul fil d'exécution, ce fil d'exécution étant représenté par une valeur de compteur ordinal et une pile d'exécution.

Un *thread* est un espace d'adressage dans lequel plusieurs fils d'exécution peuvent évoluer en parallèle. Ces fils d'exécution sont chacun caractérisés par une valeur de compteur ordinal propre et une pile d'exécution privée.

De cette définition, on tire plusieurs avantages :

- Les *threads* d'un même fil partagent code et données : les commutations lors d'un changement d'état d'un thread sont allégées. On ne change que les registres et le compteur ordinal, pas la mémoire centrale
- Les partages de données sont facilités : dans le cas de processus distincts, les partages de données ne sont pas inclus dans le langage, et nécessitent la mise en oeuvre de services fournis par le système d'exploitation, lourds et peu souples.

Par contre, les *threads* partageant les mêmes données, il est généralement nécessaire de synchroniser et protéger les accès aux données, ce que nous verrons plus en avant dans le cours.

## 2 Ordonnancement des processus

Le système d'exploitation doit donc gérer l'ensemble des processus, dans le but de maintenir le taux d'utilisation du processeur le plus élevé possible. Pour cela, le système d'exploitation met en oeuvre une *politique d'ordonnancement*, algorithme régissant la commutation des tâches.

### 2.1 Critères d'ordonnancement

L'algorithme d'ordonnancement doit identifier le processus qui sera élu, de manière à assurer la "meilleure" performance du système. Selon les algorithmes, et selon les objectifs recherchés, différents critères vont être optimisés, parmi lesquels :

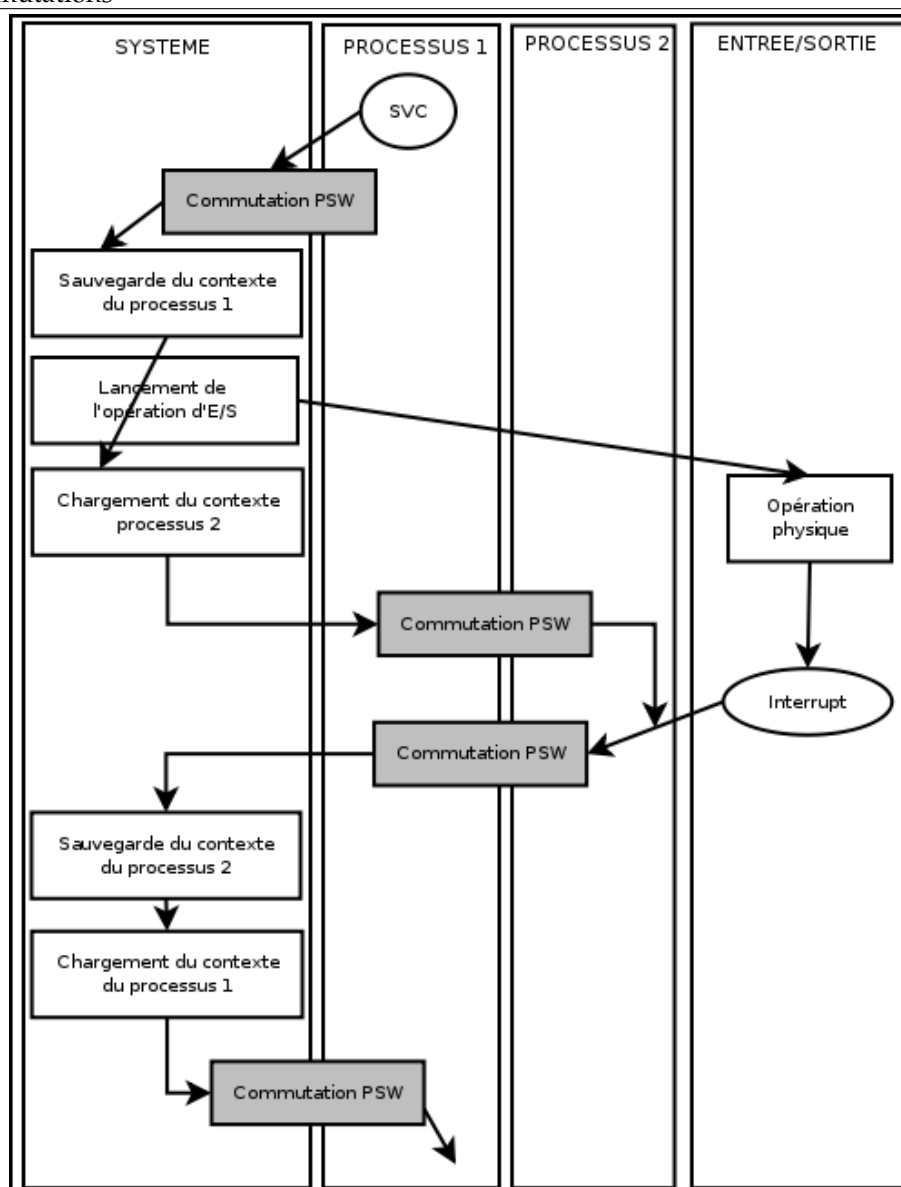
- *Utilisation de l'UC* : il s'agit de prendre en compte le temps pendant lequel l'unité centrale exécute un processus.
- *Utilisation répartie* : c'est le pourcentage de temps pendant lequel est utilisé l'ensemble des ressources. On évalue l'utilisation de la mémoire, des E/S, .. plutôt que le temps processeur.
- *Débit* : c'est le nombre de processus pouvant être exécutés par le système durant une période de temps donnée. Le calcul du débit doit prendre en compte la longueur moyenne d'un processus. Sur des systèmes aux processus longs, le débit est inférieur à celui des systèmes aux processus courts.

- *Temps de rotation* : durée moyenne de l'exécution d'un processus. Il est inversement proportionnel au débit.
- *Temps d'attente* : durée moyenne d'attente des processus du proceseur.
- *Temps de réponse* : le temps moyen pour répondre aux entrées de l'utilisateur (systèmes interactifs).
- *Équité* : degré auquel tous les processeurs reçoivent une chance égale de s'exécuter. On évalue entre autre le fait de ne pas permettre à un processus de souffrir de *famine*, à savoir rester bloqué indéfiniment.

## 2.2 Principes

Un processus est donc successivement en phases de calcul, et en phase d'entrées-sorties. L'idée est donc de tenter de se faire recouvrir une phase d'E/S d'un processus avec des phases de calcul d'un autre processus :

FIG. 2 Commutations

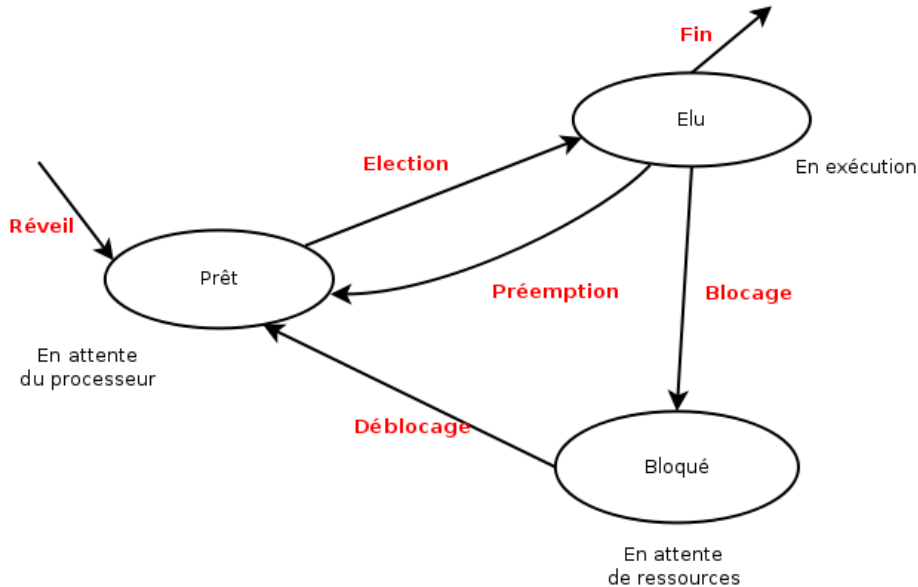


## 2.3 Algorithmes d'ordonnancement

Au début de l'informatique, l'ordonnancement était souvent non préemptif, ou coopératif : un processus conservait le contrôle de l'unité centrale jusqu'à ce qu'il se bloque ou qu'il se termine. Pour des

travaux par lots, le système convenait, le temps de réponse n'ayant pas grande importance. Sur les systèmes interactifs actuels, c'est l'ordonnancement préemptif qui est utilisé : le système d'exploitation peut *préempter* un processus avant qu'il ne se bloque ou ne se termine, afin d'attribuer le processeur à un autre processus. On peut ainsi compléter le schéma du premier chapitre.

FIG. 3 Cycle de vie des processus en ordonnancement préemptif



On recense six algorithmes d'ordonnancement répandus :

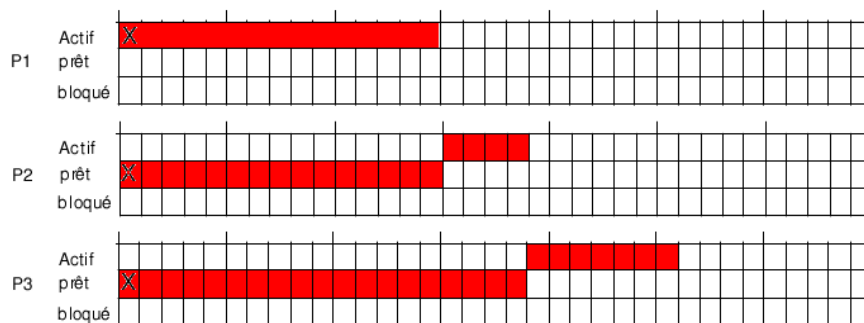
### 2.3.1 FIFO - *First In First Out*

Le FIFO est le plus simple des algorithmes d'ordonnancement. En effet, l'ordonnanceur (non préemptif), utilise une pile dans laquelle sont stockés dans l'ordre d'arrivée les processus en attente du processeur.

#### Exemple 2.1 Exemple d'ordonnancement FIFO

Soit trois processus P1, P2, P3 faisant uniquement du calcul. Les temps d'exécution des trois processus sont respectivement 15, 4 et 7 millisecondes. Le chronogramme d'exécution des processus peut être représenté de la manière suivante :

FIG. 4 FIFO



L'ordonnancement FIFO a donc tendance à privilégier les processus longs ou tributaires de l'unité centrale

### 2.3.2 SJF - Shortest First Job

L'ordonnement basé sur l'algorithme du travail le plus court d'abord est également non préemptif. A partir d'une estimation du temps nécessaire à l'exécution des processus, le prochain élu est donc le plus court. Ce qui nous donne pour l'exemple précédent (en supposant qu'au départ les trois processus soient bloqués) :

---

#### Exemple 2.2 Exemple d'ordonnement SJF

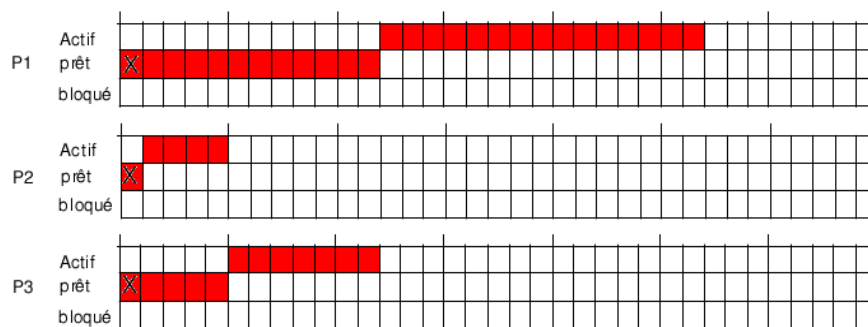
---



---

**FIG. 5 SJF**


---



Cet algorithme privilégie les programmes courts.

### 2.3.3 SR - Shortest Remaining Time

L'ordonnement du temps restant le plus court est la version préemptif de l'algorithme SJF : chaque fois qu'un processus passe dans l'état prêt (à sa création ou à la fin d'une E/S), on compare la valeur estimée du temps de traitement restant à celle du processus en cours. On préempte le processus en cours si son temps restant est plus long. L'algorithme privilégie les programmes courts, mais il peut y avoir un risque de famine pour les programmes longs.

### 2.3.4 RR - Round Robin

L'ordonnement par tourniquet (*Round Robin*) est préemptif, et sélectionne le processus attendant depuis le plus longtemps. Un quantum de temps est spécifié, et à chacun de ces intervalles, on choisit une nouvelle sélection. Plus le quantum est faible, plus les temps de réponse sont courts, mais plus les commutations sont fréquentes, d'où une dégradation possible des performances.

---

#### Exemple 2.3 Exemple d'ordonnement RR

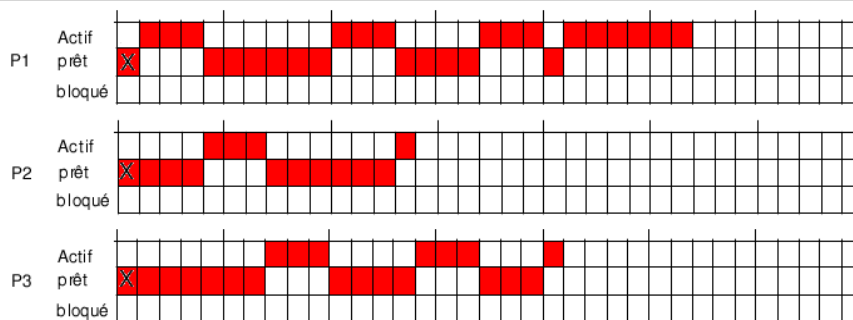
---

Nous reprenons l'exemple précédent, en supposant que P1, P2 et P3 ont été soumis dans cet ordre. Le quantum est fixé à 3ms.

---

**FIG. 6 RR**


---



### 2.3.5 Avec priorité

Pour l'ordonnement préemptif à priorité, on affecte une valeur à chaque processus. Le processus élu est celui qui à la priorité la plus élevée (valeur la plus faible), et en cas d'égalité, on utilise FIFO. Les priorités peuvent être fixées en fonction des caractéristiques du processus (beaucoup d'E/S, utilisation de la mémoire,...), de l'utilisateur, ou même d'une priorité fixée par l'administrateur ou l'utilisateur (commande `nice` sous *Unix*). Pour éviter le risque de famine des processus de trop faible priorité, on peut augmenter la priorité en fonction du temps d'attente écoulé.

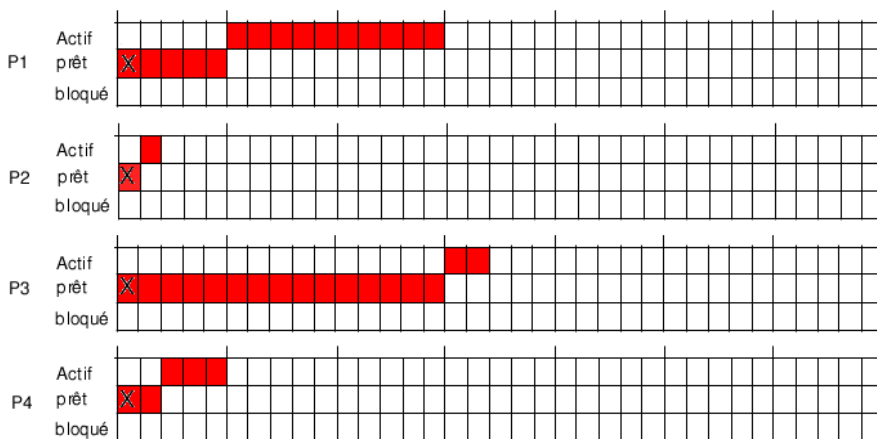
---

#### Exemple 2.4 Exemple d'ordonnement avec priorité

Soit P1,P2,P3 et P4 des processus de temps d'exécution respectifs 10,1,2,3, et de priorité respectives 3,1,3,2.

---

FIG. 7 Avec priorité



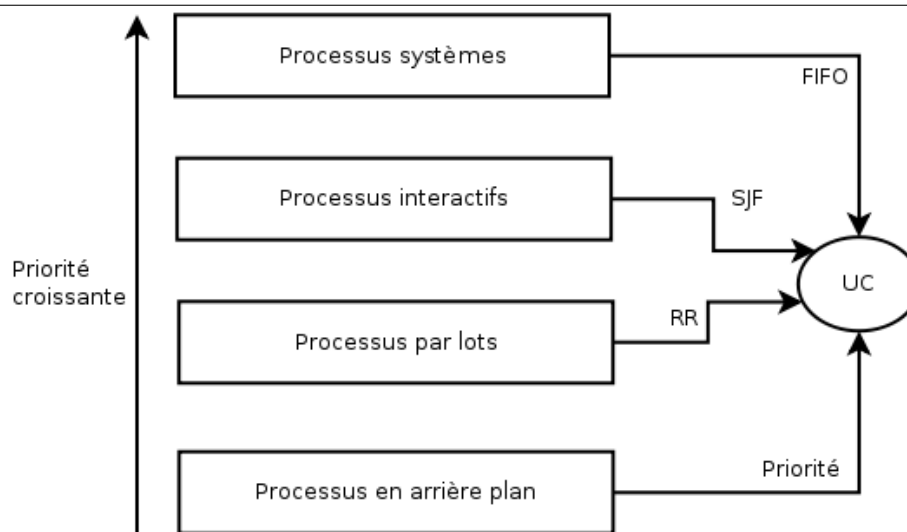

---

### 2.3.6 MQS - Multilevel Queue Scheduling

L'ordonnement par files multi-niveaux est une extension de l'ordonnement avec priorité, mais en plaçant les processus de même priorité dans des files d'attente distinctes. Par exemple, les systèmes en temps partagé supportent généralement la notion de processus en premier (*foreground*) et en arrière plan (*background*). Les premiers sont souvent des processus interactifs, alors que les seconds n'ont besoin de s'exécuter que si le processeur est libre. Donc les deux types de processus nécessitent deux ordonnancements différents.

**Exemple 2.5** Exemple de MQS

FIG. 8 MQS



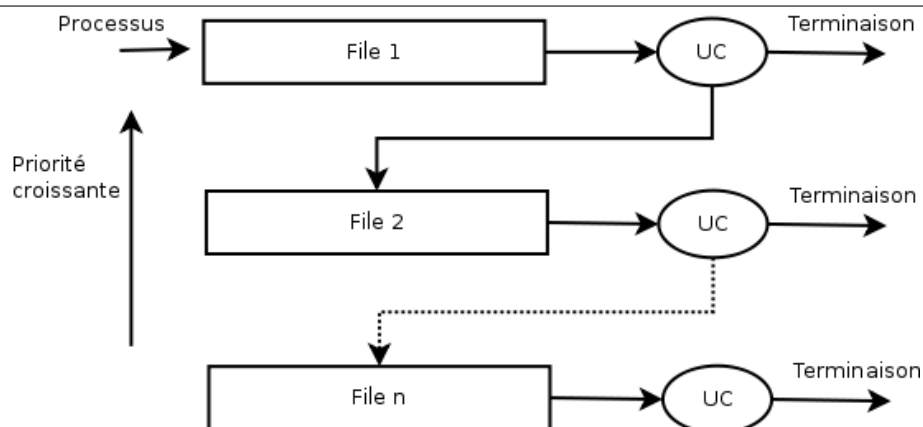
Dans cet exemple, chaque file de processus est ordonnancée par un algorithme différent.

Les processus des files de plus basse priorité ne peuvent s'exécuter que lorsque les processus de la file supérieure sont tous bloqués. Il y a donc risque de famine.

**2.3.7 MFQ - Multilevel Feedback Queues**

L'ordonnement par *files multi-niveaux à retour* résout le problème des MQS en permettant aux processus de changer de file au cours du temps. On sépare ainsi les processus selon leur temps d'occupation de l'unité centrale. Si un processus consomme trop de temps processeur, il est alors déplacé vers une file de priorité moindre. Les files de plus basse priorité ont un temps processeur moindre.

FIG. 9 MFQ



Les implémentations des MFQ diffèrent selon :

- Le nombre de files
- L'ordonnement de chaque file
- la méthode utilisée pour déterminer la descente d'un processus en file inférieure
- la méthode utilisée pour déterminer la montée d'un processus en file supérieure
- la méthode utilisée pour choisir la file initiale d'un processus

## 3 Implémentation sous *Unix* et *FreeBSD*

### 3.1 Processus

#### 3.1.1 Hiérarchie de processus

Les système *Unix* reposent entièrement sur une arborescence de processus "père" et de processus "fils" : chaque processus se duplique, puis par recouvrement de son code d'origine, exécute un code spécifique. Le premier processus est le processus 0, directement créé au démarrage du système d'exploitation. Ce processus 0 crée entre autre par duplication (`fork`), le processus 1, aussi appelé processus `init`. Ce dernier va à son tour se dupliquer, pour lancer des processus critiques pour le système, dont certains *daemons* (gestion des terminaux, *daemons* réseaux, ...)

---

#### Exemple 3.1 Liste des processus

---

```
ikare@kaitan:~ > ps -axo pid,ppid,user,command | sort -k 2
1      0 root      /sbin/init --
15     0 root      [TIMER]
10     0 root      [audit]
18     0 root      [bufdaemon]
22     0 root      [flowcleaner]
4      0 root      [g_down]
2      0 root      [g_event]
3      0 root      [g_up]
11     0 root      [idle]
12     0 root      [intr]
0      0 root      [kernel]
13     0 root      [ng_queue]
8      0 root      [pagedaemon]
17     0 root      [pagezero]
7      0 root      [pfpurge]
6      0 root      [sctp_iterator]
21     0 root      [softdepflush]
20     0 root      [syncer]
16     0 root      [usb]
9      0 root      [vmdaemon]
19     0 root      [vnlru]
5      0 root      [xpt_thrd]
14     0 root      [yarrow]
1378   1 haldaemon /usr/local/sbin/hald
1896   1 ikare     /usr/local/bin/dbus-daemon --fork --print-pid 5
3714   1 ikare     /usr/local/libexec/gam_server
1898   1 ikare     /usr/local/libexec/gconfd-2
1210   1 messagebus /usr/local/bin/dbus-daemon --system
694    1 root      /sbin/devd
1360   1 root      /usr/libexec/getty Pc ttyv1
1361   1 root      /usr/libexec/getty Pc ttyv2
1362   1 root      /usr/libexec/getty Pc ttyv3
1363   1 root      /usr/libexec/getty Pc ttyv4
1364   1 root      /usr/libexec/getty Pc ttyv5
1365   1 root      /usr/libexec/getty Pc ttyv6
1366   1 root      /usr/libexec/getty Pc ttyv7
1381   1 root      /usr/local/sbin/console-kit-daemon
1265   1 root      /usr/local/sbin/httpd -DNOHTTACCEPT
1309   1 root      /usr/sbin/cron -s
1289   1 root      /usr/sbin/sshd
842    1 root      /usr/sbin/syslogd -s
121    1 root      adjkerntz -i
```

---

L'extrait précédent montre ainsi une partie de l'arborescence des processus sur une station *FreeBSD*. Chaque processus est identifié par un PID (*Process Identifier*) et connaît le PID de son processus père (PPID).

### 3.1.2 Création des processus

La création d'un processus se fait à l'aide de la fonction `fork()`. Cette fonction duplique le processus. Le processus fils est alors un nouveau processus, exécutant le code depuis le *fork*, et de manière concurrente au père.

---

#### Exemple 3.2 Création d'un processus

---

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

main()
{
    pid_t ret;
    ret = fork();
    if (ret == 0)
    {
        printf("je suis le fils mon pid est %d\n", getpid());
        printf("Mon pere est le process %d\n", getppid());
    } else
    {
        printf("je suis le pere mon pid est %d\n", getpid());
        printf("Mon fils est le process %d\n", ret);
    }
    exit(0);
}

ikare@kaitan:exemple > ./a.out
je suis le pere mon pid est 58483
Mon fils est le process 58484
je suis le fils mon pid est 58484
Mon pere est le process 58483
```

---

L'exemple ci-dessus montre une manière de différencier les processus en utilisant le code de retour du `fork()`. Ce code est égal à 0 pour le fils, et égal au PID du fils pour le père.

Les deux processus précédents exécutent le même code, mais dans la plupart des cas, les fils vont se différencier de leur père. Ceci est réalisé à l'aide de primitives de recouvrement, qui vont charger un autre code pour le processus fils. C'est la famille des primitives `exec`, dont nous n'allons présenter qu'un seul exemple :

**Exemple 3.3** PrIMITIVE de recouvrement

```

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

main()
{
    pid_t ret;
    ret = fork();
    if (ret == 0)
    {
        printf("je suis le fils mon pid est %d\n", getpid());
        printf("Mon pere est le process %d\n", getppid());
        execlp("date", "date", NULL);
    } else
    {
        printf("je suis le pere mon pid est %d\n", getpid());
        printf("Mon fils est le process %d\n", ret);
        wait();
    }
    printf("Seul le pere %d continue le fil du programme\n", ret);
    exit(0);
}

je suis le pere mon pid est 59638
Mon fils est le process 59639
je suis le fils mon pid est 59639
Mon pere est le process 59638
Wed Mar  3 17:15:02 RET 2010
Seul le pere 59863 continue le fil du programme

```

Dans l'exemple précédent, le fils continue l'exécution du code après l'appel à la fonction `fork()`. L'appel de la primitive `exec` va *recouvrir* le code du fils.

**3.1.3** Terminaison des processus

Un appel à la primitive `exit()` entraîne la terminaison du processus, avec un code de retour. Un processus qui se termine passe dans l'état zombi, et reste dans cet état tant que son père n'a pas pris en compte sa terminaison. Lorsqu'un processus se termine, le système efface tout son contexte, mais conserve son entrée dans la table des processus. Le processus père attend la mort de son fils par la primitive `wait()`, récupère alors le code de retour et détruit l'entrée dans la table des processus.

Un processus orphelin (le père s'est terminé avant le fils), est toujours adopté par le processus `init`.

---

**Exemple 3.4** Terminaison de processus

---

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

main()
{
    pid_t pid;
    printf("je suis %d et je vais creer un fils \n",getpid()) ;
    pid = fork() ;
    if (pid == 0) {
        printf("je suis %d le fils et je vis \n",getpid()) ;
        printf("je meurs .... adieu mon pere \n") ;
        exit(0) ;
    }
    else {
        printf("je suis le pere de %d \n",pid) ;
        wait();
        printf("je me suicide .... mon fils est mort\n") ;
    }
}
```

```
ikare@kaitan:exemple > ./a.out
je suis 90711 et je vais creer un fils
je suis le pere de 90712
je suis 90712 le fils et je vis
je meurs .... adieu mon pere
je me suicide .... mon fils est mort
```

---

**Exemple 3.5** Processus zombi

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

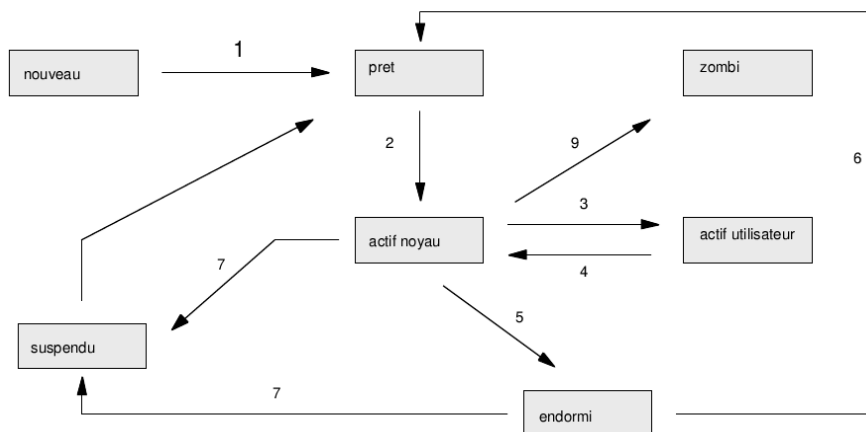
main()
{
    pid_t pid;
    printf("je suis %dd et je vais creer un fils \n",getpid() );
    printf("je bouclerai ensuite a l'infini \n") ;
    pid = fork() ;
    if (pid == 0) {
        printf("je suis %d le fils et je vis \n",getpid()) ;
        sleep(10) ;
        printf("je me suicide .... \n") ;
        exit(0) ;
    }
    else {
        for(;;) ;
    }
}
```

```
ikare@kaitan:exemple > ./a.out &
[1] 82050
ikare@kaitan:exemple > je suis 82050d et je vais creer un fils
je bouclerai ensuite a l'infini
je suis 82051 le fils et je vis
ps
  PID  TT  STAT      TIME COMMAND
  1522  v0  I        0:00.02 -tcsh (tcsh)
  82050  pb  R        0:01.04 ./a.out
  82051  pb  S        0:00.00 ./a.out
je me suicide ....
ikare@kaitan:exemple > ps
  PID  TT  STAT      TIME COMMAND
  1522  v0  I        0:00.02 -tcsh (tcsh)
  82050  pb  R        0:11.74 ./a.out
  82051  pb  Z        0:00.00 <defunct>
ikare@kaitan:exemple > kill -15 82050
ikare@kaitan:exemple > ps
  PID  TT  STAT      TIME COMMAND
  1522  v0  I        0:00.02 -tcsh (tcsh)
[1] + Terminated          ./a.out
```

**3.2 Etats du processus sous *Unix***

Des exemples précédents, nous pouvons donc préciser les états d'un processus sous *Unix* :

FIG. 10 Etats d'un processus



1. 1. Le processus est créé, il est en attente du processeur
2. 2-4. Le processus est passé en mode *superviseur*
3. 3. Mode normal d'exécution : actif utilisateur
4. 5. Le processus est bloqué
5. 6. En attente du processeur
6. 7. Arrêté par un signal
7. 9. Processus terminé, mais PCB toujours présent (fin non prise en compte par son père)

### 3.3 Threads

Les *threads*, processus légers, sont une extension du modèle des processus : au sein d'un même processus vont pouvoir s'exécuter plusieurs fils d'exécution, partageant le même espace d'adressage. L'exemple suivant montre effectivement deux *threads* modifiant une variable commune. Dans le cas des processus lourds, cette variable aurait été dupliquée pour chaque fil d'exécution.

**Exemple 3.6** Threads*threads*

```

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

int i;

void simple_func() {
    i+=10;
    printf("Je suis le thread fils %d, voici i : %d\n",getpid(),i);
    i+=20;
    printf("Je suis le thread fils %d, voici i : %d \n",getpid(),i);
}

main()
{
    pthread_t t_id;int ret;
    i=0;
    ret = pthread_create(&t_id, NULL, (void *(*())simple_func,NULL);
    i+=1000;
    printf("Je suis le thread initial %d, voici i : %d\n",getpid(),i);
    i+=2000;
    printf("Je suis le thread initial %d, voici i : %d\n",getpid(),i);
    pthread_join(t_id,NULL);
}

ikare@kaitan:exemple > gcc -pthread thread1.c ❶
ikare@kaitan:exemple > ./a.out
Je suis le thread initial 38475, voici i : 1000
Je suis le thread initial 38475, voici i : 3010
Je suis le thread fils 38475, voici i : 1010
Je suis le thread fils 38475, voici i : 3030

```

1. La compilation sous *FreeBSD* nécessite le *flag* `-pthread`.

**3.4 L'ordonnancement sous FreeBSD 8.0**

*FreeBSD* a hérité dès son "fork" de la branche 4.3BSD de l'implémentation de l'ordonnanceur de la famille BSD, le *4BSD Scheduler*. Cet ordonnanceur était une adaptation du traditionnel ordonnanceur Unix, un Round-Robin à priorités. Apparue avec *FreeBSD* 5.0, l'ordonnanceur ULE est resté optionnel (nécessitait la recompilation du noyau) jusqu'à la version 7.1.

**3.4.1 Le 4BSD Scheduler**

Cet ordonnanceur a été conçu pour des systèmes interactifs et par lots, c'est-à-dire pour favoriser les processus interactifs tout en équilibrant le temps processeur donné aux processus longs (compilateurs, programme de calculs, ...). Il s'agit d'un *Round-Robin* à priorités, ces priorités étant calculées sur une estimation du temps utilisé par le processus, et de son niveau de priorité système (*nice*).

Cette priorité est recalculée toutes les secondes, selon le principe suivant :

```
prio = CPU-Usage + nice + base
```

La base est une valeur associée à la cause de blocage du processus, et calculée lorsque le processus repasse en attente. Le processus qui a la valeur de priorité la plus basse est élu.

Cet algorithme donne de bons résultats dans un contexte interactif et avec peu de processus en cours. En cas de montée en charge, le recalcul des priorités toutes les secondes devient pénalisant pour le système. De plus, cet ordonnanceur ne tire pas partie des nouvelles capacités des architectures multi-processeurs. L'implémentation sous *FreeBSD* apporta quelques améliorations dans les temps de réponses même sous haute charge, mais sans nouvelle prise en compte des architectures récentes.

### 3.4.2 ULE Schedule

L'ordonnanceur ULE est désormais l'algorithme par défaut de *FreeBSD*. Il supporte maintenant les nouvelles fonctionnalités des processeurs multicœurs et des cartes multiprocesseurs, et permet d'obtenir des temps d'exécution constants quelque soit le nombre de processus. Il est également capable d'identifier clairement les tâches interactives, en leur fournissant un temps de réponse le plus court possible.

L'ordonnanceur repose maintenant sur :

- trois files par processeurs :
  - Une file pour les processus "idle" (peu prioritaires, tels qu'écrans de veille, etc...). Cette file n'est prise en compte que si les deux autres files sont vides.
  - une file "current" dont sont élus (par priorité) les processus jusqu'à la vider. Y sont insérés les processus interactifs et temps-réels.
  - une file "next" qui sera échangée avec la queue "current" quand celle-ci sera vidée. Y sont insérés les processus qui ne rentrent pas dans les classes précédemment citées.
- deux algorithmes de d'équilibrage de charge sur plusieurs CPU
- un "détecteur d'interactivité", permettant de définir par rapport au ratio de temps d'exécution et de blocage si un processus est interactif ou non.
- un estimateur de charge CPU, principalement utilisé par les commandes systèmes **ps**, **top** et autres.
- un calculateur de priorité, dont le rôle est principalement d'éviter les cas de famine.

## 4 Exercices

### 4.1 Processus

#### 4.1.1 Exercice 1

Ecrire un bout de code qui crée 15 processus avec les contraintes suivantes :

- le processus initial est l'un des 15 ;
- un processus donné a zéro, deux ou trois fils (mais jamais un seul) ;
- tous les fils du processus initial ont eux-même des fils ;
- le code est le plus simple possible et contient le plus petit nombre de `fork()` possibles.

#### 4.1.2 Exercice 2

Quelles traces génère le programme suivant ? Expliquez ..

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

int i;
main()
{
    pid_t ret;
    i=0;
    ret = fork();
    if (ret == 0)
    {
        i+=10;
        printf("fils %d\n", i);
        i+=20;
        printf("fils %d\n", i);
    }
    else
    {
        i+=1000;
        printf("pere %d\n", i);
    }
}
```

```

        i+=2000;
        printf("pere %d\n",i);
        wait();
    }
}

```

## 4.2 Ordonnement

### 4.2.1 Exercice 1 (source : Joelle Delacroix - Cnam)

Cinq travaux A, B, C, D et E sont soumis à un processeur dans cet ordre, mais quasi simultanément. Ces travaux ne font pas d'entrées-sorties. Leurs durées respectives sont 10, 6, 2, 4 et 8 secondes. Déterminer les temps de réponse de chacun des travaux, ainsi que le temps de réponse moyen pour :

1. l'algorithme FIFO
2. l'algorithme SJF
3. l'algorithme à priorité, avec  $P(A)=3$ ,  $P(B)=5$ ,  $P(C)=2$ ,  $P(D)=1$ ,  $P(E)=4$

### 4.2.2 Exercice 2 (source : Christian Carrez - Cnam)

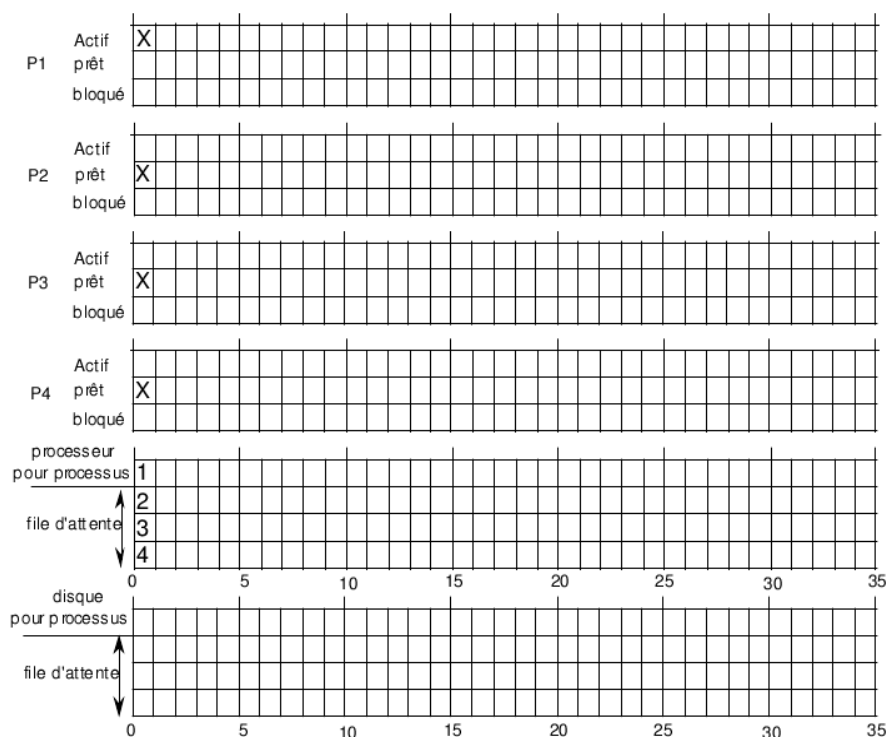
On considère un système monoprocesseur et les 4 processus P1, P2, P3 et P4 qui effectuent du calcul et des entrées/sorties avec un disque selon les temps donnés ci-dessous :

Processus P1	Processus P2
Calcul : 3 unités de temps	Calcul : 4 unités de temps
E/S : 7 unités de temps	E/S : 2 unités de temps
Calcul : 2 unités de temps	Calcul : 3 unités de temps
E/S : 1 unité de temps	E/S : 1 unité de temps
Calcul : 1 unité de temps	Calcul : 1 unité de temps
Processus P3	Processus P4
Calcul : 2 unités de temps	Calcul : 7 unités de temps
E/S : 3 unités de temps	
Calcul : 2 unités de temps	

Remplissez le chronogramme pour chacun des cas suivants :

1. On considère que l'ordonnement sur le processeur se fait selon une politique FIFO : le processus élu à un instant  $t$  est celui qui est le plus anciennement dans l'état prêt. Initialement, l'ordre de soumission des processus est P1, puis P2, puis P3, puis P4. De même, on considère que l'ordre de services des requêtes d'E/S pour le disque se fait selon une politique FIFO.
2. On considère maintenant que l'ordonnement sur le processeur se fait selon une politique à priorité préemptible : le processus élu à un instant  $t$  est celui qui le processus prêt de plus forte priorité. On donne priorité  $(P1) > \text{priorité} (P3) > \text{priorité} (P2) > \text{priorité} (P4)$ . On considère que l'ordre de services des requêtes d'E/S pour le disque se fait toujours selon une politique FIFO.
3. On considère toujours que l'ordonnement sur le processeur se fait selon une politique à priorité préemptible : l'ordre des priorités des 4 processus reste inchangé. On considère maintenant que l'ordre de services des requêtes d'E/S pour le disque se fait également selon la priorité des processus : le processus commençant une E/S est celui de plus forte priorité parmi ceux en état d'attente du disque. Une opération d'E/S commencée ne peut pas être préemptée.

FIG. 11 Chronogramme



## 5 Références

### 5.1 Références

- [1] Paolo ZanellaYves Ligier, *Architecture et technologie des ordinateurs* , **Dunod** , isdn : ISBN 2 10 003801 X, 199,2002.
- [2] Matt WelshKalle DalheimerLar Kaufman, *Le systeme Linux* , **O'Reilly** , 1995, 1997, 1999, 2000.
- [3] Marshall Kirk McKusick, *The Design And Implementation Of The FreeBSD Operating System*, Hardcover , 2003.
- [4] Joelle Delacroix, *Linux* , **Dunod** , 2003.
- [5] Jean-Marie Rifflet, *La programmation sous Unix*, EdiScience , 1986, 1989, 1993, 2002.
- [6] Michael W. Lucas, *Absolute FreeBSD: The Complete Guide to FreeBSD*, **No Starch Press** , 2007.
- [7] Michael W. Lucas, *FreeBSD 7.0*, **No Starch Press** , 2007.
- [8] Nicholas P. Carter, *Architecture de l'ordinateur*, **EdiScience - Schaum's** , 2002.
- [9] J. Archer Harris, *Systèmes d'exploitation*, **EdiScience - Schaum's** , 2002.
- [10] Eric Steven Raymond, *The art of Unix Programming*, **Thyrsus Enterprises** , 2003.
- [11] Jeff Roberson , *ULE : A Modern Scheduler For FreeBSD*, **USENIX Association** , 2003.