PROCESSUS

Le concept de processus est le plus important dans un système d'exploitation. C'est une abstraction d'un programme eu exécution. La conception et la réalisation d'un système repose sur ce concept.

Une fois un programme transformé en code exécutable (chapitre précédent), il faut le charger et l'exécuter. On dit qu'il faut le transformer en processus car les systèmes d'exploitation ne reconnaissent d'autres entités exécutables que les processus. Le processus est une entité manipulée par tout système d'exploitation.

Ce chapitre explique comment les systèmes d'exploitation manipulent ces processus. On commence d'abord par définir le modèle de processus qui repose sur l'identité d'un processus exprimée par son contexte et ses états et sur les structures de données utiles pour gérer les processus. Le modèle de processus est un formalisme de base des systèmes d'exploitation. Il montre la représentation machine des processus.

On cite ensuite tous les algorithmes de scheduling des processus avec une sélection d'un algorithme pour un système particulier.

Les processus peuvent coopérer pour la réalisation d'une tâche donnée en partageant certaines ressources. Ces accès concurrents aux ressources partagées affectent la cohérence des données. Pour la maintenance de cette cohérence, divers mécanismes sont utilisés pour assurer une exécution ordonnée des processus. On citera également en fin de ce chapitre ces principaux mécanismes et leur implémentation en milieu des processus.

Les problèmes en relation seront cités dans les chapitres concernés tels que les interblocages et la gestion de la mémoire.

III.1 MODELE DE PROCESSUS

III.1.1 Introduction

Tous les ordinateurs modernes peuvent effectuer plusieurs tâches à la fois. Dans un système multiprogrammé, le processeur peut aussi commuter d'un programme à un autre en exécutant chaque programme pendant quelques ***ms.*** Mais le processeur, à un instant donné, n'exécute réellement qu'un programme à la fois et donne, ainsi, aux utilisateurs une impression de parallélisme.

Cette commutation de contexte '***Context Switching***' est qualifiée de pseudo parallélisme pour la distinguer du vrai parallélisme qui se produit lorsque le processeur travaille en même temps que certains périphériques d'E/S. Le contrôle de plusieurs activités parallèles est une tâche difficile, c'est pourquoi le modèle de processus a été amélioré au cours du temps pour devenir plus simple d'emploi.

III.1.2 Modèle de processus

Un système d'exploitation est organisé en un certain nombre de processus séquentiels ou simplement processus.

III.1.3 Définition d'un processus

Un processus est un programme en exécution . Il est défini par son PCB (Process Control Block) qui représente le contexte du processus, son code, ses données et sa pile(fig.III1).

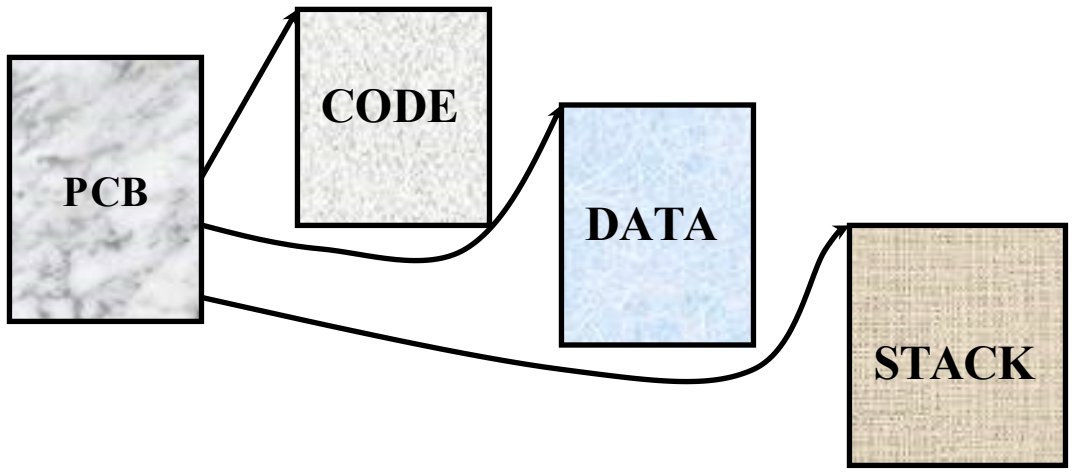


Fig.III.1 Structures d'un processus

III.1.3.1 PCB

Le PCB contient toutes les informations nécessaires à la commutation de contexte regroupées sous forme compactée pour faciliter leur transfert vers les registres du processeur. Ce transfert est réalisé par des instructions assembleur. Le PCB est pointé par un registre spécial du processeur. Il est conforme et dépendant du processeur. C'est la partie du système la plus en relation avec le hardware. Il est divisé en un PCB hardware et un PCB software. Il spécifie :

- L'état du processus
- Le PC et le PSL
- Les registres du processeur
- Une partie pour le Scheduling
  - La priorité du processus
  - Les pointeurs sur les files d'attente du Scheduling
  - Autres informations nécessaires au scheduling
- Une partie pour la gestion de la mémoire
  - Le pointeur sur le code(table des pages du code)
  - Le pointeur sur les données
  - Le pointeur sur la pile
  - Autres informations nécessaires à la gestion de la mémoire
- Une partie pour la comptabilisation
  - La priorité du processus
  - Les limites de temps
  - Le temps CPU et le temps réel utilisés
  - Identité du processus etc..
- Une partie pour les E/S
  - Les périphériques alloués au processus
  - Les fichiers ouverts par ce processus

III.1.3.2 Code

C'est un segment constitué de plusieurs pages en mémoire renfermant le code du programme à exécuter par le processus. Ce n'est pas nécessairement la totalité du code exécutable généré par la translation. Une partie de ce code suffit à l'initialisation du processus. Le reste du code sera chargé au fur et à mesure de l'évolution du processus. On doit respecter le quota mémoire associé au processus. Comme on travaille en relocation, le code n'est pas fixe en mémoire. Pour cela, il nécessite un pointeur dans le PCB. Le PC pointe toujours sur l'instruction courante du code en exécution.

III.1.3.3 Data

C'est comme le segment de code mais cette fois ci il renferme les données manipulées par le processus. En général, on affecte le même espace pour les data et la pile(voir chapitre IV). Il est géré par le système de la même façon que le code. Une partie des

données est chargée en mémoire pour permettre l'initialisation du processus, le reste suit conformément à l'évolution du processus.

III.1.3.4 Stack

La pile est utilisée par un processus lors de l'appel de sous programme surtout car on doit sauvegarder l'état du programme appelant avant de passer la main au programme appelé. Cette sauvegarde s'effectue pratiquement sur une pile. L'usage particulier de la pile par un processus est souvent possible. Elle dispose d'un pointeur de sommet de pile et d'une adresse début en mémoire

III.1.4 Etat d'un processus

Les processus bien qu'étant des entités indépendantes doivent interagir avec d'autres processus. Cette interaction nécessite d'identifier un processus par son état. Un processus se bloque lorsqu'il ne peut pas, pour une raison logique, poursuivre son exécution. Un processus élu peut être arrêté, même s'il peut poursuivre son exécution, si le système décide d'allouer le processeur à un autre processus. Enfin, l'état du système dépend des états de ses processus.

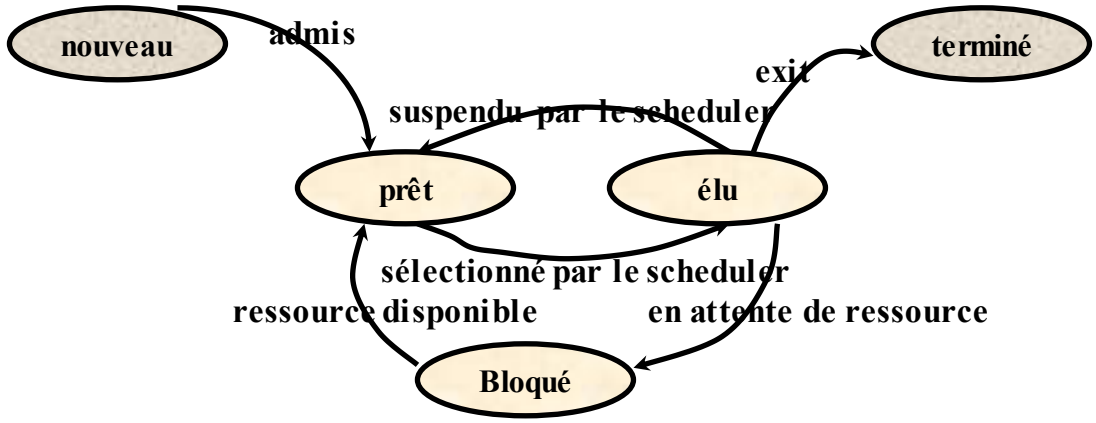


Fig.III.2 Diagramme des états d'un processus

Un processus suit toujours le diagramme des états de la fig.III.2.Il peut être dans un des états suivants :

- Nouveau: processus en cours de création pouvant être admis par le système.
- Terminé : processus terminant son exécution et quittant le système.
- Elu : processus en cours d'exécution occupant le processeur.
- Prêt : processus en attente d'élection pour un processeur par le scheduler.
- Bloqué: processus attendant un événement de mise en disponibilité de ressource.

Il suit les transitions suivantes :

- Admis : processus accepté par le système.
- Exit : processus quittant le système.
  - Suspendu par le scheduler : processus disposant de toutes les ressources mais le scheduler décide d'exécuter un autre processus (fin de quantum par exemple).
  - Sélectionné parle Scheduler : le scheduler choisit ce processus.
  - Ressource disponible : l'événement signalant la disponibilité d'une ressource ayant provoquée la suspension du processus a eu lieu. Le processus n'a plus de rester bloqué.
  - En attente de ressource : le processus demande une ressource non disponible. Il passe dans un état d'attente de ressource. Cet état se distingue selon le nombre de ressources.

III.1.5 Hiérarchie des processus

Dans la plupart des systèmes, il faut pouvoir créer et détruire dynamiquement des processus. A l'initialisation du système, on crée tous les processus qui s'avèrent utiles et nécessaires. Pour cela, tout système doit fournir des mécanismes de création et de destruction de processus. Aussi, on doit pouvoir permettre à tout processus d'utiliser ces mécanismes ; c'est à dire créer ses propres processus qui sont appelés dans ce cas processus fils dont il devient le père et pouvoir les détruire à tout moment.

Quand un processus crée un fils celui-ci hérite de toutes les propriétés du père. Le système crée un copie conforme du processus père (PCB). Le code et les data peuvent cependant changer. Le fils peut à son tour créer ces propres fils. Un processus n'a qu'un seul père mais plusieurs fils. Ce qui conduit à une arborescence de processus (fig.III.3).

A l'initialisation du système, un processus particulier appelé (INIT) est chargé à partir du disque système. Il lit un fichier de configuration où il trouve l'ensemble des processus qu'il faut créer selon le matériel installé. Il crée un processus pour terminal. Ces processus attendent alors leurs connexions éventuelles et lorsqu'une connexion a eu lieu, ils lancent le langage de commandes (SHELL). Le Shell crée pour chaque commande un processus. La fin d'une commande termine un processus. Tous les processus font partie d'une même arborescence dont INIT est la racine.

Dans le cas de la fig.III.3, Le processus **init** crée les processus **user$_1$**, **user$_2$**, **user$_3$**, …, **user$_n$** . Le processus **user$_1$**, par l'exécution de son shell lance la commande **vi** en créant un nouveau processus. Le processus **user$_3$** lance la commande **vi&** qui crée un processus pour l'exécution de l'éditeur de texte **vi** en mode asynchrone et remet la main au shell qui lance ensuite un autre processus pour l'exécution du programme **paie.** Celui-ci, à son tour, lance en parallèle les programmes **salaire**, **compta** et **gestion** qui nécessitent chacun un processus. Le processus **user$_n$** comme **user$_3$** lance **vi&** et **ps** .

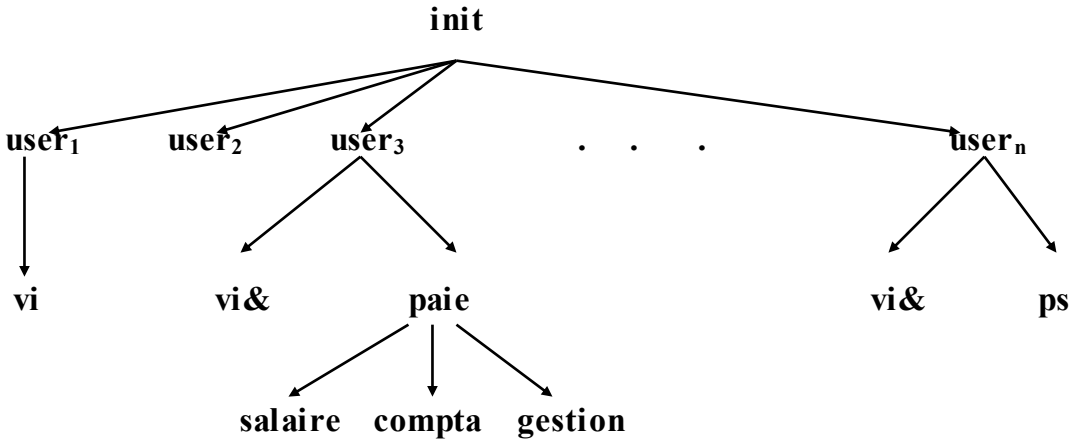


fig.III.3 Hiérarchie de processus

III.1.6 Implémentation du modèle de processus

Donc tout système d'exploitation est représenté par un ensemble de processus. Certains sont des processus système et les autres sont des processus utilisateurs. Ils sont organisés selon figIII.4

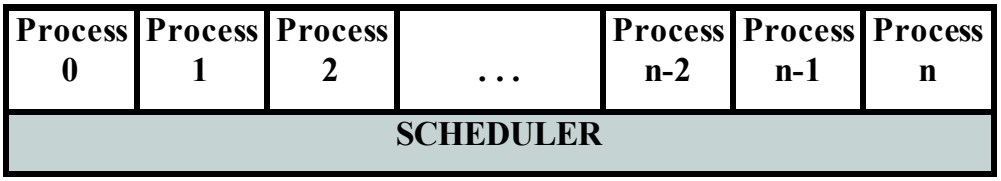


Fig.III.4 Modèle de processus

La couche la plus basse d'un système structuré en processus se charge des interruptions et effectue le scheduling. Le reste du système est constitué de processus.

La gestion des interruptions, la suspension et la relance des processus selon le diagramme des états font partie intégrante du scheduling. Cette couche porte le nom de noyau de synchronisation ou tout simplement KERNEL.

Dans un modèle de processus, le schéma suivant est toujours respecté suite à une interruption :

- Empilement du PC et PSL (opération hardware)
- Chargement du nouveau PC à partir du vecteur de l'interruption (hardware)
- Exécution de la routine d'it
  - Sauvegarde des registres du processeur dans le PCB (assembleur)
  - Sélection de la nouvelle pile (assembleur)
  - Changement de l'état du processus (driver d'E/S) à prêt (C)
  - Sélection du processus par le scheduler
  - Fin de la procédure C
  - Lancement du processus élu par la routine assembleur

III.1.7 Threads

A l'origine l'implémentation des co-routines nécessitait plusieurs processus hors un programme sous forme de co-routines n'est en réalité qu'un seul processus. Ceci assure une bonne protection des données et allège énormément le système. Aussi, il existe plusieurs situations où il serait utile de partager des ressources et de pouvoir y accéder d'une manière concurrentielle. La routine d'accès à la ressource peut être implémentée sous forme de thread s'exécutant dans le même espace d'adresse que le processus accédant à cette ressource via la routine. Ce concept est implémenté par tous les systèmes actuels. Un processus, dans ce cas une tâche, est donc constitué de plusieurs threads partageant les mêmes données, la pile et parfois le même code.

III.1.7.1 Définition d'un Thread

C'est un flot de contrôle séquentiel indépendant associé à un processus. Il permet de développer des programmes dans lesquels plusieurs flots de contrôle peuvent être spécifiés. Ce concept (figIII.5) :

- améliore la performance
- permet une meilleure structuration de l'application

exemple :

Lorsqu'une activité exécute une entrée/sortie et est en attente de la fin de cette opération, une autre activité peut exécuter un autre traitement.

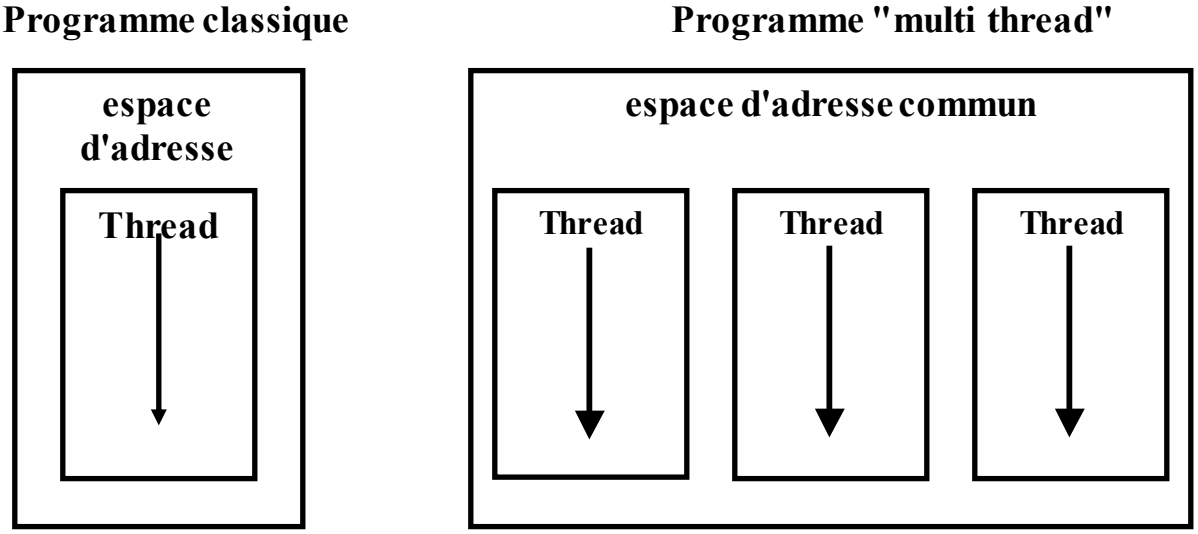


Fig.III.5 Concept de Thread

III.1.7.2 conception d'applications Multi thread

• Modèle Boss/worker

La fonction du Boss est de donner des tâches à effectuer aux threads travailleurs. Lorsqu'un thread travailleur a terminé, il interrompt le chef pour lui indiquer qu'il a terminé. Le chef interroge régulièrement les travailleurs pour savoir où ils en sont.

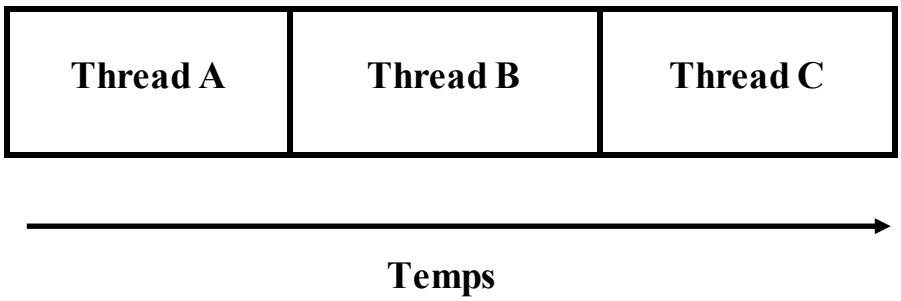
• Modèle crew

La tâche à réaliser est divisée en plusieurs threads qui exécutent en parallèle une partie du travail.

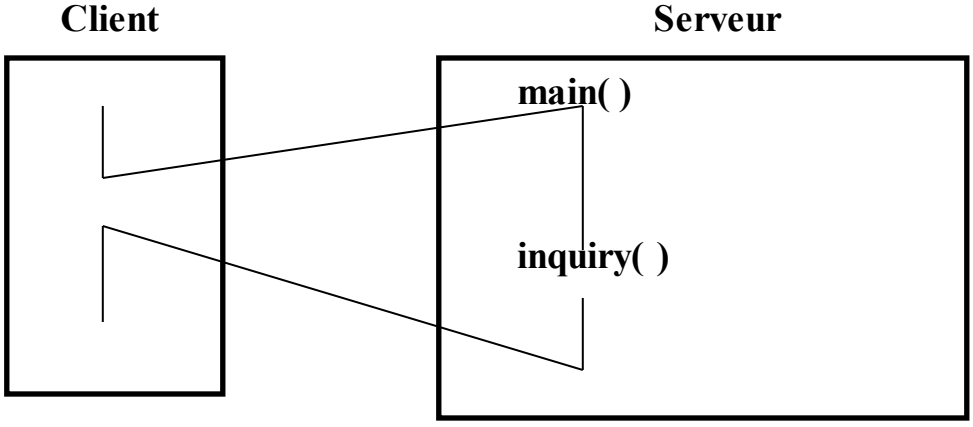
<table>
<tr><td rowspan="3">Début</td><td>Thread A</td><td rowspan="3">Fin</td></tr>
<tr><td>Thread B</td></tr>
<tr><td>Thread C</td></tr>
</table>

• Le modèle pipeline

Une tâche est divisée en étapes. Les étapes sont effectuées en séquence, le résultat d'une étape correspondant aux données en entrée de l'étape suivante



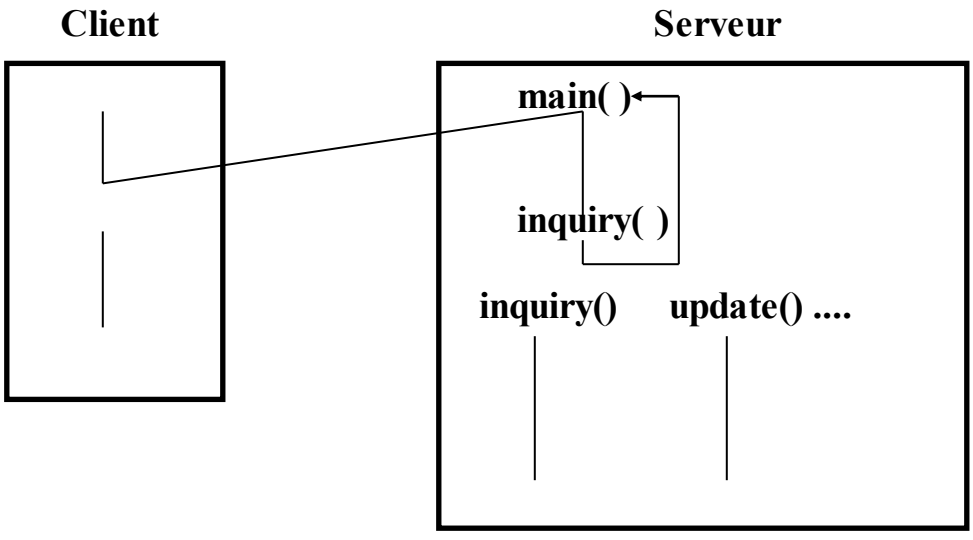
III.1.7.3 Exemple de serveur "mono-thread" :



```
void main()
{
   do {
       select();
       switch (msg.req) {
       case inq:
               inquiry(state_entry);
               break;
       case update :
                 ....
         }
      } until cows_come_home;
}

void inquiry (state_table_pt state entry)
{ switch (state_entry->state) {
     case initialize :
                  .....
     case guts :
                  .....
     case final :
                 ....
   }
}
```

III.1.7.4 Exemple de serveur "multi-thread"



```
void main()
{
    do {
        select ();
        switch (msg.req){
                    case inq :
                         create(...,inquiry,...);
                         break;
                    case update :
                                ...
                   }
    } until cows_come_home;
}

void inquiry ()
{
    allocate_this;
    ....
}
```

III.1.7.5 Structure d'un thread

La structure de données représentant un thread est définie par :

- son propre identificateur
- une politique et une priorité d'ordonnancement
- une partie privée :
  - pile d'exécution,
  - valeur de registre,
  - mémoire locale
  - mémoire locale
- une partie partagée : texte du programme,
- variables statiques, fichiers ouverts, sockets, etc....

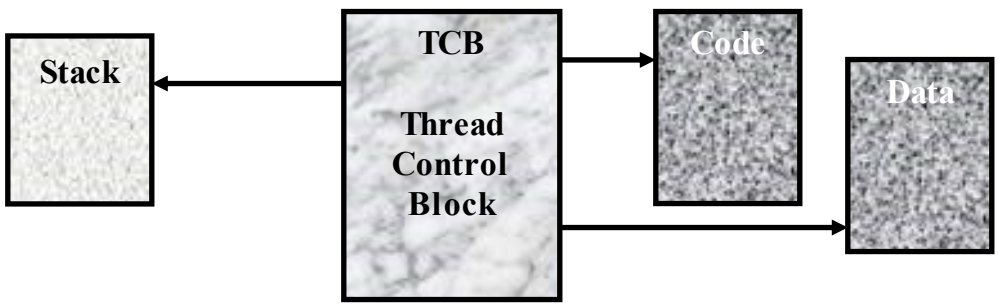


Fig III.5 Structure d'un thread

III.1.7.6 Etats d'un thread

Un thread peut être dans l'un des six états suivants (fig.III.6) :

- **Ready** : le thread est en attente d'exécution
- **Stand by** : un thread parmi les Readys est sélectionné pour s'exécuter sur un processeur. Un thread par processeur. Le dispatcher, dans les conditions convenables, exécute le context switching. Un seul thread par processeur peut être dans l'état Stand by.
- **Running** : Suite au context switching, le thread rentre en exécution jusqu'à ce que :
  - Un thread prioritaire arrive.
  - Le quantum expire
  - Le thread se termine.
  - Il entre volontairement dans l'état waiting
- **Waiting** : il entre cet état si :
  - Il attend volontairement un objet pour synchroniser son exécution..
  - Au compte du système I/O
  - L'utilisateur demande au thread de suspendre son exécution..

  Quand l'attente se termine, il passe à l'état Ready.
- **Transition** : ressources non disponibles.
- **Terminated** : fin d'exécution.

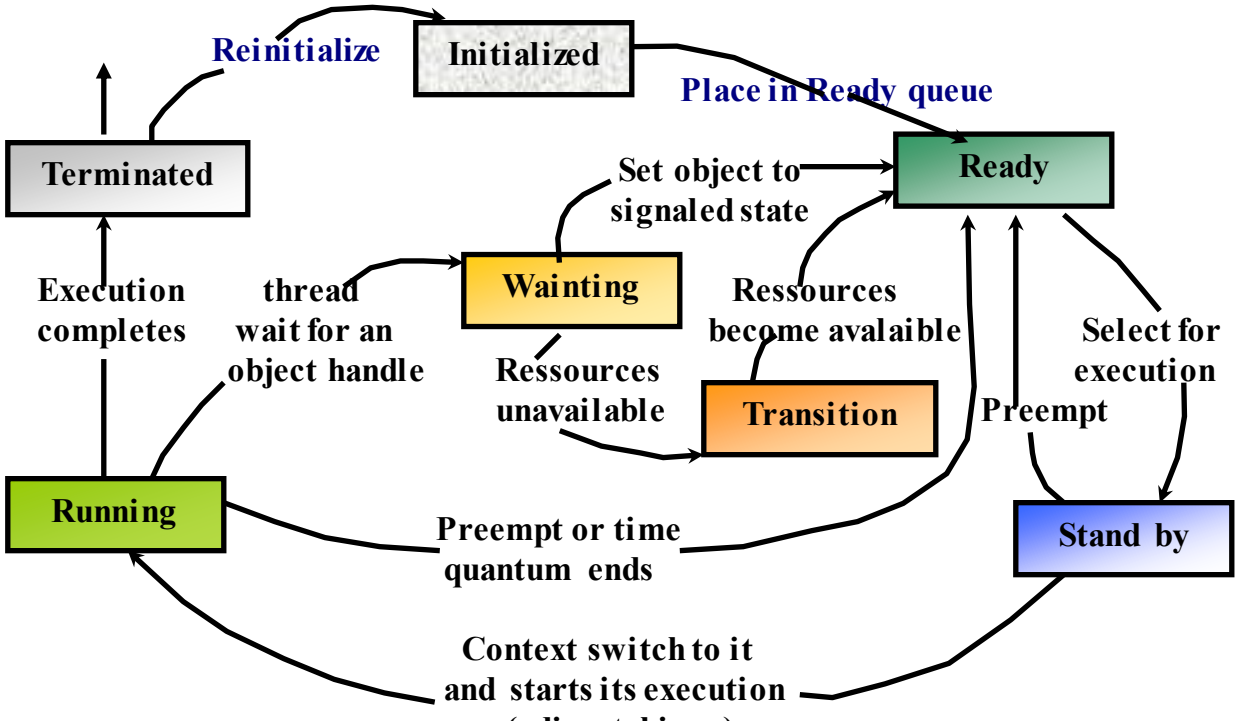


Fig III.6 Diagramme des états d'un thread

Sur une machine monoprocesseur, un seul thread est actif à la fois. Sur une machine multiprocesseur, plusieurs threads peuvent être actifs (parallélisme réel).

III.1.7.7 Problèmes

Certains problèmes peuvent se présenter dans trois domaines:

• Au niveau du traitement des signaux :

Un signal peut être capturé, ignoré ou mis en attente. Certains signaux sont synchrones : ils sont causés par le thread actif suite à une erreur (une violation de segmentation , une erreur de calcul en virgule flottante). Quand un signal synchrone arrive, il est géré par le thread courant, le processus entier est alors tué, sauf si le signal est ignoré ou capturé. D'autres sont asynchrones : ils sont causés par des processus externes (l'appui par l'utilisateur sur la touche DEL pour interrompre le processus en cours). Quand un signal asynchrone arrive, le système vérifie qu'aucun thread n'est en attente de ce signal. Si c'est le cas il est passé à tous les threads pouvant le gérer.

• La plupart des procédures de la librairie standard ne sont pas réentrantes:

il se peut qu'un thread soit occupé à faire de l'allocation mémoire lorsqu'il y a une commutation de contexte (changement de thread). A ce moment là, les structures internes du gestionnaire de la mémoire sont incohérentes ... ce qui pose des problèmes au nouveau du thread actif qui essaie de faire de l'allocation mémoire....On résout ce problème en ajoutant à quelques procédures (surtout E/S) des enveloppes qui garantissent un accès en exclusion mutuelle. Pour les autres procédures, une exclusion mutuelle globale rend certain que seul un thread à un instant est actif dans la librairie. Les procédures comme **read, fork**,... sont des procédures enveloppées (primitives).

• La variable d'erreur *errno*

les appels système d'Unix retournent leur état d'erreur dans une variable globale ***errno***. Si l'un des threads fait un appel système, mais, juste après que celui-ci soit terminé, un autre thread est exécuté et fait également un appel système, l'original de la variable **errno** est perdu. Une interface de gestion des variables d'erreur est ajoutée. C'est une macro qui permet au programmeur de consulter la version spécifique à chaque thread de la variable ***errno***.

III.2 SCHEDULING

Cette partie se distingue par deux sections pour mettre en œuvre le scheduling. On doit d'abord présenter les différentes structures de données utilisées par les algorithmes de scheduling ensuite citer ces derniers selon leurs caractéristiques et leur utilité.

III.2.1 Structures de données :

Dans un système, les processus (threads) sont ordonnés en listes. Ceux en attente de mémoire se trouvent sur une liste en mémoire secondaire. Les autres se divisent en classes selon les différents états du diagramme des processus : (fig.III.7)

- Ready Queue : renferme tous les processus prêts en attente d'exécution.
- Device Queues : renferment par classe l'ensemble des processus en attente de ressources. Pour chaque type d'E/S, il existe une liste associée.

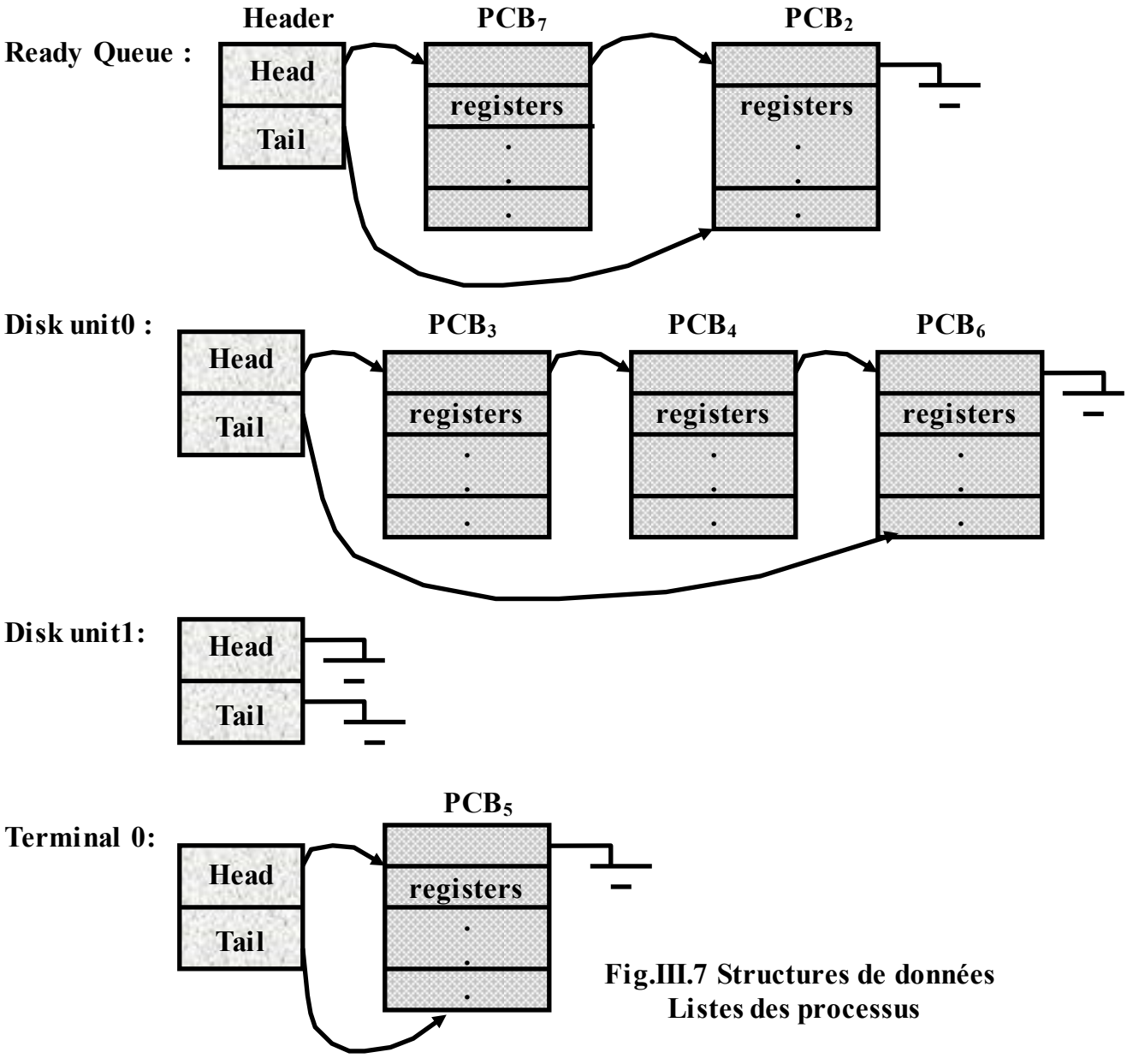


Fig.III.7 Structures de données
Listes des processus

III.2.2 Context Switching :

Le fait de retirer le CPU à un processus et l'affecter à un autre est le context switching « commutation de contexte » (fig III.8). Le temps pris par cette opération est important et touche aux performances du système. Le context switching dépend de la rapidité de la mémoire et de l'existence d'instructions spéciales (chargement et sauvegardes des PCB en une seule instruction machine). Il varie de 1 à 100ms.

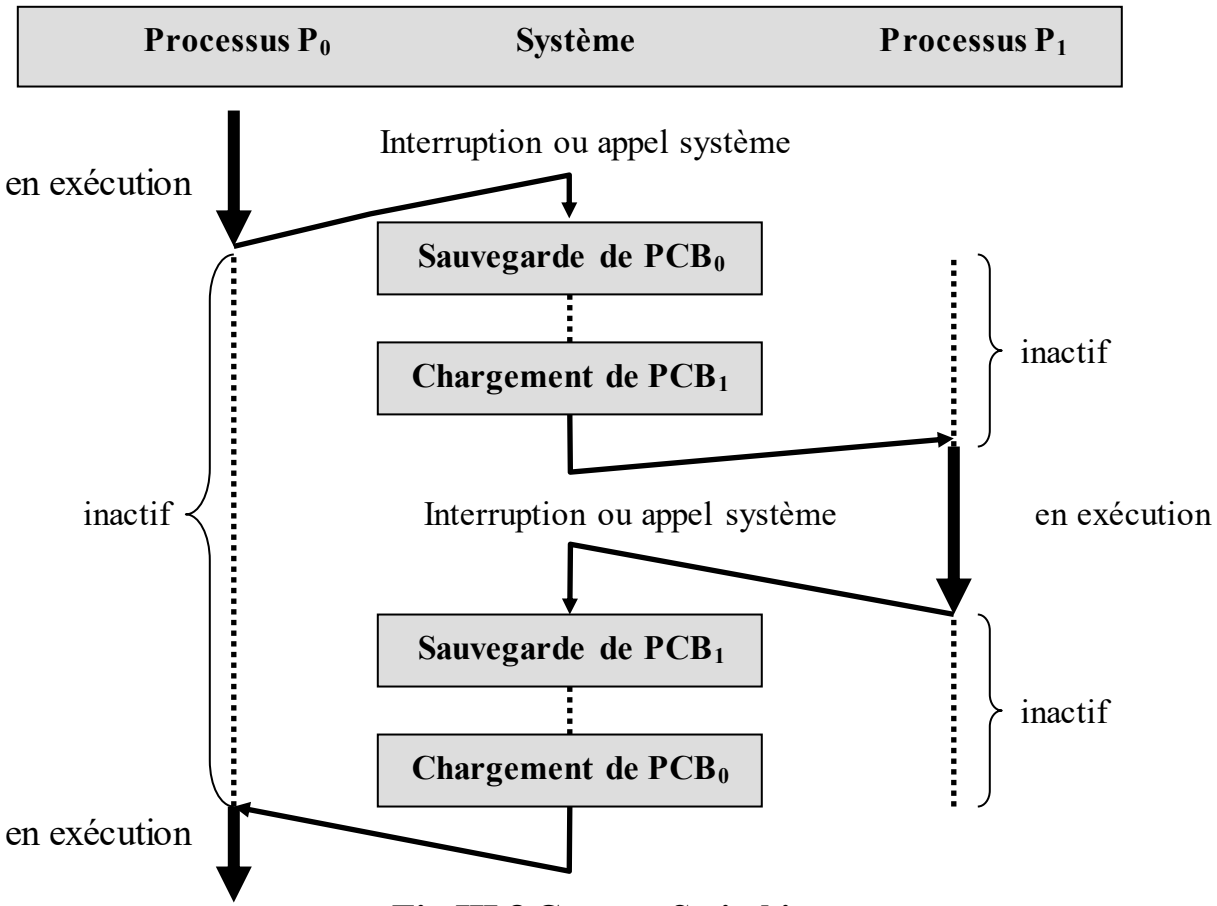


Fig.III.8 Context Switching

III.2.3 Algorithmes de Scheduling :

On utilise souvent le diagramme de la fig.III.9 pour représenter et expliquer le scheduling.

Un nouveau processus est placé dans la **Ready Queue** , il attend dans cette liste sa sélection pour l'exécution. Quand il dispose du processeur (élu) un parmi les événements suivants se produit :

- Le processus demande une I/O, il est placé dans la I/O Queue.
- Le processus crée un fils et attend qu'il termine.
- Le processus doit être forcé de quitter le CPU suite à une it (ou le temps expire)

Le processus continue dans ce diagramme jusqu'à sa fin.

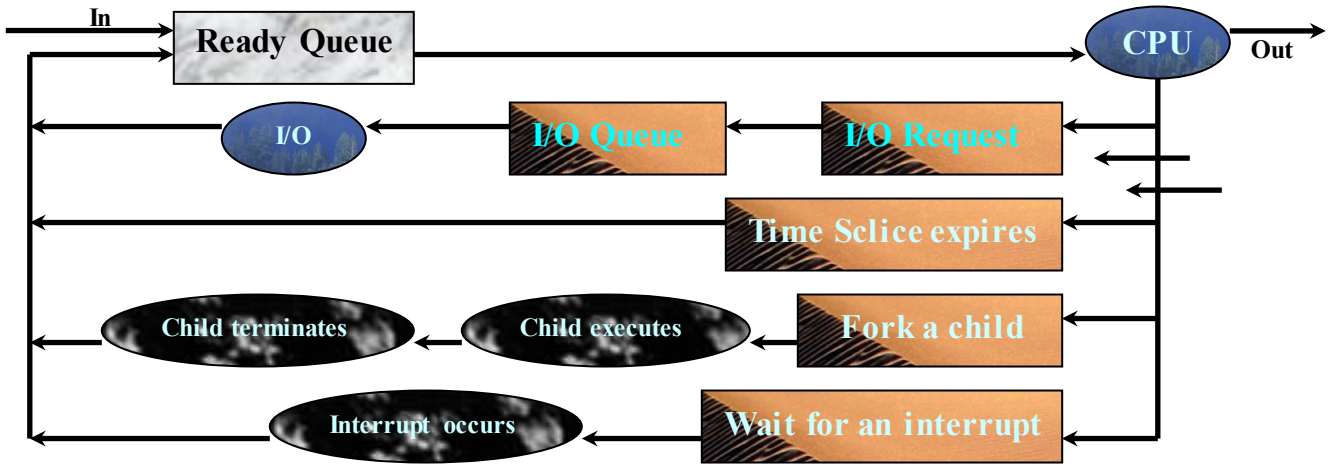


Fig.III.9 Diagramme des transitions

III.2.3.1Types de Schedulers :

La sélection des processus pour l'exécution est réalisée par le scheduler. Il existe deux types de schedulers :

- Le job scheduler : souvent appelé long term scheduler, sélectionne le job et le ramène en mémoire pour exécution.
- Le CPU scheduler : ou short term scheduler, sélectionne parmi les processus prêts, le plus approprié et lui alloue le CPU.

La distinction enter ces deux types est due à la fréquence d'utilisation. Le CPU scheduler ou tout simplement scheduler doit être très rapide puisqu'il est très fréquent. Le job scheduler contrôle le degré de multiprogrammation (le nombre de processus en MC). Si ce degré est stable, le nombre de processus entrant doit être égal au nombre de processus sortant.

III.2.3.2 Caractéristiques du Scheduler :

L'algorithme doit remplir certaines caractéristiques :

- Equitabilité : s'assurer que chaque processus reçoit sa part du temps processeur.
- Efficacité : le processeur doit travailler à 100% du temps.
- Temps de réponse : maximiser le temps de réponse en interactif.
- Temps d'attente : minimiser le temps d'attente des processus.
- Rendement : maximiser le nombre de processus par heure.

Il est généralement difficile de satisfaire toutes ces contraintes vu leurs critères contradictoires. Le meilleur algorithme est celui qui répond d'une façon optimale à ces contraintes.

III.2.3.3 Décisions du Scheduler :

Le scheduler doit décider :

1. quand un processus passe de l'état **élu** à l'état **bloqué** .
2. « « « « « « « « « **prêt** .
3. « « « « « « bloqué « «.
4. « « « se termine.

III.2.3.4 Non preemptive Scheduler :

Une fois le CPU alloué à un processus, celui-ci le retient jusqu'à la fin de son exécution ou jusqu'à passer à un état bloqué (décisions 1 et 4).

III.2.3.5 Preemptive Scheduler :

Renferme les décisions 1 à 4.

6.3.2 LE DISPATCHER

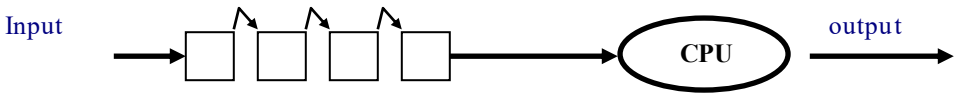
C'est un composant impliqué dans la fonction de scheduling. Il donne le contrôle du CPU au processus élu par le scheduler. Sa fonction inclut :

- Le context switching
- Changement du mode user au mode kernel
- Initialisation du PC par l'adresse début du programme utilisateur.

III.2.4 Algorithmes de Scheduling :

III.2.4.1.1 FCFS :

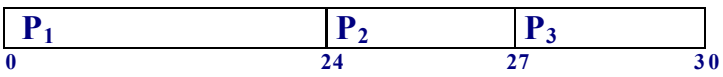
Le FCFS (first come first served) utilise une file d'attente simple où le premier venu est le premier servi.



Exemple :

Processus	temps CPU
P_1	24
P_2	3
P_3	3

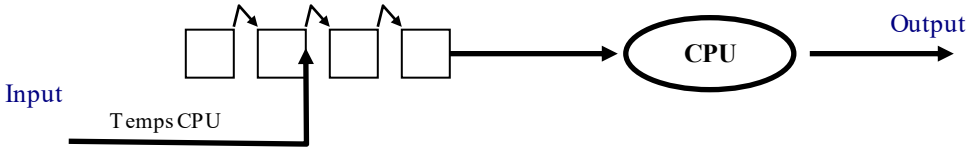
Si on considère que l'ordre d'arrivée est P_1, P_2, P_3, le diagramme de Gantt est :



	P_1	P_2	P_3	*moyen*
Temps attente	0	24	27	17
Turnaround	24	27	30	27

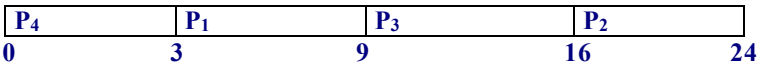
III.2.4.1.2 SJF :

Le SJF (Shortest Job First) utilise une file ordonnée selon le temps d'exécution croissant (le plus court en tête de liste et le plus lent en queue de liste). Si deux processus possèdent le même temps estimé, ils sont traités en FCFS. L'estimation est faite à priori.



Exemple1 : SJF Non Préemptif

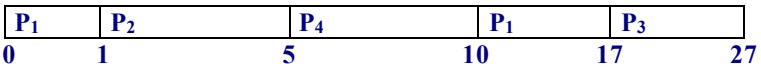
Processus	temps_CPU
P_1	6
P_2	8
P_3	7
P_4	3



	P_1	P_2	P_3	P_4	*moyen*
Temps attente	3	16	9	0	7
Turnaround	9	24	16	3	13

Exemple2 : SJF Préemptif

Processus	temps_CPU	Temps_d'arrivée
P_1	8	0
P_2	4	1
P_3	9	2
P_4	5	3

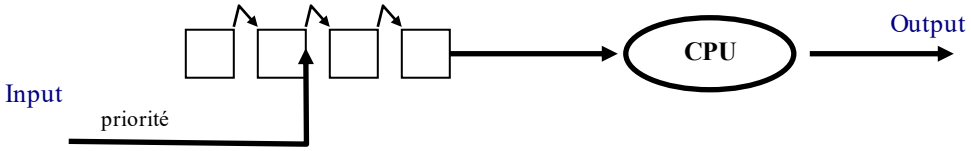


	P_1	P_2	P_3	P_4	*moyen*
Temps attente	9	0	15	2	6.5
Turnaround	17	4	25	7	13.25

III.2.4.1.3 Priority :

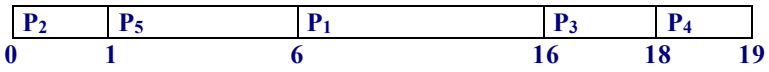
Une priorité est associée à chaque processus, le CPU est alloué au processus prioritaire. Les processus de même priorité obéissent à un FCFS. La priorité la plus

élevée est définie par convention. 0 peut être la plus haute ou la plus basse selon les systèmes. Les nouveaux processus sont placés dans la file selon leur priorité.



Exemple1 : Priority Non Préemptif

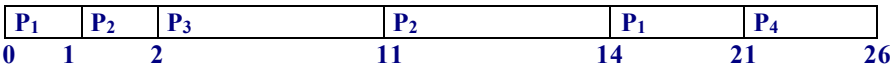
Processus	temps CPU	Priorité
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2



	P_1	P_2	P_3	P_4	P_5	moyen
Temps attente	6	0	16	18	1	8.2
Turnaround	16	1	18	19	6	12

Exemple2 : Priority Préemptif

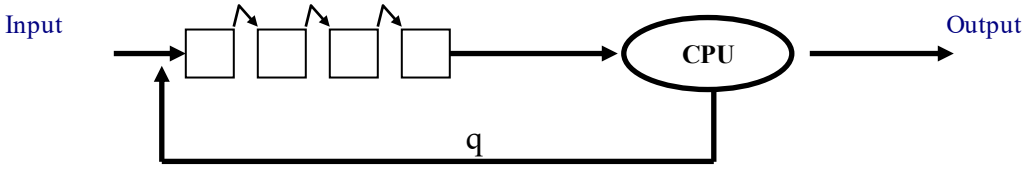
Processus	temps CPU	Priorité	Temps d'arrivée
P_1	8	3	0
P_2	4	2	1
P_3	9	1	2
P_4	5	3	3



	P_1	P_2	P_3	P_4	moyen
Temps attente	13	9	0	18	10
Turnaround	21	13	9	23	16.5

III.4.2.3 Round-Robin :

Il est caractérisé par un quantum de temps qui varie de 10 à 100ms. La Ready-queue est FIFO et circulaire. Les nouveaux processus sont ajoutés en queue de liste.



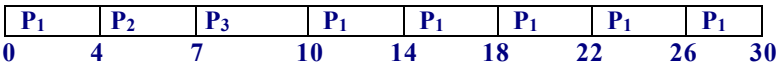
Le CPU saisit le processus en tête de liste et initialise l'horloge à la valeur du quantum. Celle-ci se décrémente automatiquement :

- Le temps CPU du processus est inférieur au quantum et le processus quitte volontairement le CPU avant q. On passe au processus suivant.
- L'horloge atteint 0 et émet une interruption qui provoque un context swiching. Le processus en cours passe en queue de liste car son temps CPU dépasse q et celui en tête de liste est sélectionné.

Exemple :

Processus	temps CPU
P_1	24
P_2	3
P_3	3

Si on considère que l'ordre d'arrivée est P_1, P_2, P_3 à t=0 et que le quantum q=4 alors le diagramme de Gantt est :

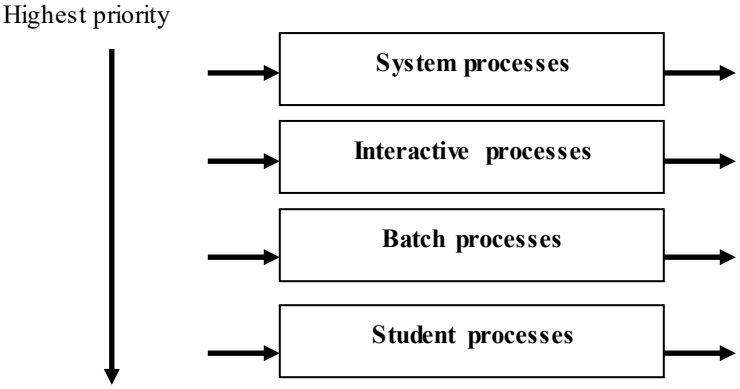


	P_1	P_2	P_3	moyen
Temps attente	6	4	7	5.6
Turnaround	30	7	10	15.6

III.2.4.2.4 MQS (Multilevel Queue Scheduling) :

Dans cet algorithme, on divise la Ready queue en plusieurs listes. Cette division dépend du type de processus et de ces propriétés telles que la taille de la mémoire etc…

Exemple :

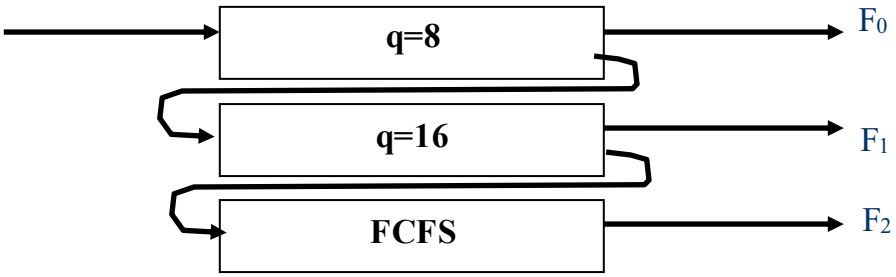


Chaque liste dispose de son propre algorithme de scheduling.
Il faut en plus un scheduling entre les listes.
Les processus sont assignés à une liste et y restent jusqu'à la fin de leur exécution. Mais cette approche n'est pas flexible (le processus ne peut changer de liste).

III.2.4.2.5 MFQS (Multilevel Feedback Queue Scheduling) :

MFQS permet à un processus de changer de listes. L'idée est basée sur le temps CPU nécessaire au processus. Si un processus nécessite beaucoup de temps, il décroît dans les listes jusqu'à la moins prioritaire. Un processus qui a beaucoup attendu dans une liste peut grimper vers les priorités supérieures. Ceci permet d'éviter le âging.

Exemple :
Considérons un MFQS à 3 listes F_0, F_1,F_2.



Tout nouveau processus est servi dans la liste F_0. Si F_0 est vide le scheduler exécute F_1. Si F_1 est également vide, il exécute F_2.

Si un processus arrive dans F_0 ,il exécute la préemption des processus de F_1 (tous les processus de F_1 sont suspendus) et dispose du CPU. Egalement un processus arrivant dans F_1 fait la même chose avec les processus de F_2 .

Un processus de F_0 a un quantum q=8ms. S'il nécessite plus de 8ms, il passe dans F_1. quand F_0 devient vide, les processus de F_1 (en particulier le processus arrivant de F_0) sont pris en compte prenant chacun un quantum q=16ms. S'il ne se termine pas, il passe dans F_2 où il séjourne jusqu'à ce que F_0 et F_1 deviennent vides. On applique alors à F_2 un FCFS.

Un MFQS est défini par :

- Le nombre de listes.
- L'algorithme de scheduling de chaque liste.
- La méthode utilisée pour dégrader un processus.
- La méthode utilisée pour favoriser un processus.
- L'algorithme utilisé pour déterminer la liste appropriée.

Le MFQS est très général et très complexe.

III.4.3 Opérations sur les processus

Les processus dans la plupart des systèmes peuvent s'exécuter simultanément, et peuvent être créés et supprimés de manière dynamique. Ainsi, tout système d'exploitation doit prévoir un mécanisme de création et de terminaison de processus.

4.3.1 Création des processus

La primitive fork()

Un processus peut créer plusieurs nouveaux processus, via un appel système de création de processus, au cours de l'exécution. Le processus créateur est le processus père, et les nouveaux procédés sont les fils de ce processus. Chaque nouveau processus peut, à son tour, créer d'autres processus, formant ainsi une arborescence de processus (figure 7).

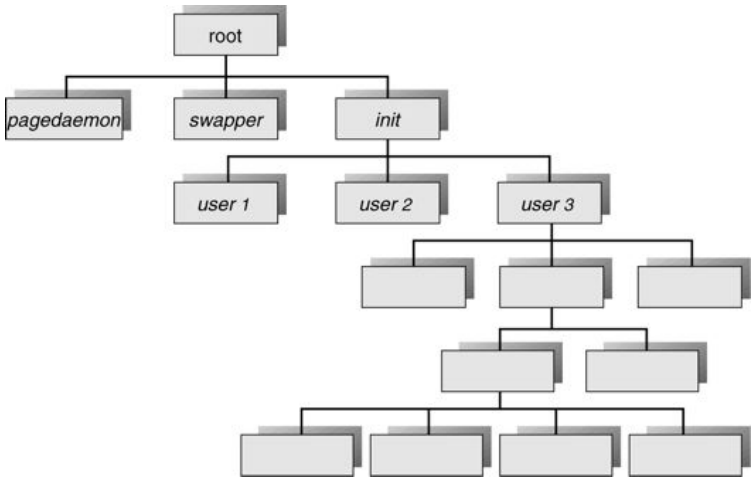


Figure 7 Une arborescence de processus UNIX.

En général, un processus a besoin de ressources (CPU, mémoire, fichiers, périphériques d'E/S) pour s'exécuter. Lors de la création de processus fils, le fils peut obtenir ses ressources directement du système d'exploitation, ou bien se restreindre à un sous-ensemble des ressources du processus père. Le père peut partitionner ses ressources parmi ses fils, ou il peut partager certaines de ses ressources (mémoire, fichiers) avec plusieurs de ses fils. Restreindre un fils à un sous-ensemble de ressources du père empêche tout processus de surcharger le système en sous-processus.

En plus des différentes ressources physiques et logiques qu"un processus obtient à sa création, les données en entrée peuvent être transmises par le père à son fils. Par exemple, un processus veut afficher un fichier F1à l'écran. A sa création, il recevra du père le nom du fichier F1qu'il ouvrira et affichera. Il peut aussi obtenir le nom du périphérique de sortie. Certains systèmes passent les ressources aux fils. Sur de tels systèmes, le processus recevra le fichier F1 et le périphérique de sortie, il s'occupe simplement du transfert entre les deux ressources (le fichier F1 et le périphérique d'E/S).

En termes d'exécution, quand un processus crée un fils :

- les deux processus sont indépendants et peuvent s'exécuter concurremment;
- le processus père est en attente de la fin d'exécution du processus fils;
- le processus lancé remplace le processus père qui disparaît.

En termes d'espace d'adressage du nouveau processus:

1. Le fils est une copie du père (il a le même code et les mêmes données que le père).
2. Le fils a son propre code.

Un processus fils hérite de la plupart des attributs de son père, à l'exception de :

- son PID, qui est unique et attribué par le système d'exploitation,
- le PID de son père,
- ses temps d'exécution initialisés à la valeur nulle,
- les signaux en attente de traitement,
- la priorité initialisée à une valeur standard,
- les verrous sur les fichiers.

Tous les autres attributs du père sont hérités, et notamment la table des descripteurs de fichiers.

Pour illustrer ces différences sous UNIX, chaque processus est identifié par son ***pid***, un nombre entier unique. Un nouveau processus est créé par l'appel système **fork()**. Le nouveau processus est constitué d'une copie de l'espace d'adressage du père. Ce mécanisme permet au père de communiquer facilement avec son fils. Après l'exécution du **fork()**, deux processus exécutent la suite du code. C'est ce qui provoque deux fois l'affichage du ***pid***. On remarque qu'un des processus a gardé le ***pid*** de départ (le père) alors que l'autre en a un nouveau (le fils) qui correspond à la valeur de retour de l'appel à **fork()**. Pour le processus père, la valeur de retour du **fork()** est le ***pid*** du fils alors que pour le fils, elle vaut *0*. C'est cette différence de valeur de retour du **fork()** qui permet, *dans un même programme source*, de différencier le code exécuté par le fils du code exécuté par le père.

Généralement, l'appel système **exec()** est utilisé après un **fork()** par l'un des deux processus pour remplacer l'espace mémoire du processus avec un nouveau programme. L'appel **exec()** charge un fichier binaire en mémoire (destruction de l'image mémoire du programme contenant **exec()**) et commence son exécution. De cette manière, les deux processus sont capables de communiquer mais chacun exécute son propre code. Le père peut ensuite créer davantage de fils, ou, il peut émettre un **wait()** si elle n'a rien à faire alors que le fils s'exécute. Le **wait()** déplace le processus vers la file d'attente des processus bloqués jusqu'à ce que le fils termine.

Le programme C ci-dessous, figure 8, illustre les appels décrits précédemment. Nous avons maintenant deux processus différents exécutant une copie du même programme. La valeur du ***pid*** du fils est zéro, alors que pour le père c'est un entier positif. Le processus fils charge et exécute la commande (/bin/ls) en utilisant **execlp()**, (**execlp ()** est une variante d'**exec()**). Le parent attend que le fils termine, grâce au **wait()**. Lorsque le fils se termine, le père reprend à partir de l'appel de **wait()** et termine par **exit(0)**.

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

int main(int argc, char *argv[])
{
int pid;

    /* fork another process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       exit(-1);
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
       exit(0);
    }
}
```

À l'issue d'un **fork()** les deux processus s'exécutent simultanément.

On souhaite généralement exécuter des tâches distinctes dans le processus père et le processus fils. La valeur retournée par **fork()** est donc très importante pour différencier le processus père du processus fils. Un test permet d'exécuter des parties de code différentes dans le père et le fils.

Le système VMS de la DEC, en revanche, crée un nouveau processus, lui charge un nouveau programme, et lance son exécution. Windows NT supporte les deux modèles: l'espace d'adresse du parent peut être dupliqué, ou le parent peut spécifier le nom d'un programme à charger dans l'espace d'adressage du nouveau processus.

La primitive exit ()

void exit (int *status*)

La primitive **exit()** met fin au processus qui l'a émis, avec un code de retour ***status***. Tous les descripteurs de fichiers ouverts sont fermés ainsi que tous les flots de la bibliothèque d'E/S standard.

Si le processus à des fils lorsque **exit()** est appelée, ils ne sont pas modifiés mais comme le processus père prend fin, le nom de leur processus père est changé en 1, qui est l'identifiant du processus **init**.

Ce processus **init** est l'ancêtre de tous les processus du système excepté le processus 1 lui-même ainsi que le processus 0 chargé de l'ordonnancement des processus.

En d'autres termes, c'est l'ancêtre de tous les processus qui adopte tous les orphelins.

Par convention, un code de retour égal à zéro signifie que le processus s'est terminé correctement, et un code non nul (généralement 1) signifie qu'une erreur s'est produite. Seul l'octet de droite de l'entier ***status*** est remonté au processus père. On est donc limité à

256 valeurs pour communiquer un code retour à son processus père.

Le père du processus qui effectue un **exit()** reçoit son code retour à travers un appel à **wait()**.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main (void)
{
  int i ;
  for (i=0 ; i < 4 ; i++) {
    int retour ;
    retour = fork() ;
    switch (retour) {
      case -1 : /* erreur */
        perror ("erreur fork\n") ;
        exit(1) ;
      case 0 : /* fils */
        printf ("fils : %d\n", i) ;
      default : /* pere */
        printf ("pere : \n") ;
    }
  }
}
```

La primitive wait()

```
#include <sys/types.h>
#include <sys/wait.h>
int  wait (int * terminaison)
```

Lorsque le fils se termine, si son père ne l'attend pas, le fils passe à l'état defunct (ou zombie) dans la table des processus.

L'élimination d'un processus terminé de la table ne peut se faire que par son père, grâce à la fonction **wait()**. Avec cette instruction :

- Le père se bloque en attente de la fin d'un fils.
- Elle rendra le ***pid*** du premier fils mort trouvé.
- La valeur du code de sortie est reliée au paramètre d'exit de ce fils.

On peut donc utiliser l'instruction **wait()** pour connaître la valeur éventuelle de retour, fournie par **exit()**, d'un processus. Ce mode d'utilisation est analogue à celui d'une fonction. **wait()** rend –1 en cas d'erreur.

Un processus exécutant l'appel système **wait()** est endormi jusqu'à la terminaison d'un de ses fils. Lorsque cela se produit, le père est réveillé et wait() renvoie le PID du fils qui vient de mourir.

La primitive waitpid()

L'appel système waitpid() permet de tester la terminaison d'un processus particulier, dont on connaît le PID.

#include <sys/types.h>
#include <sys/wait.h>
int waitpid (int pid, int *terminaison, int options)

La primitive permet de tester, en bloquant ou non le processus appelant, la terminaison d'un processus particulier ou appartenant à un groupe de processus donné et de récupérer les informations relatives à sa terminaison à l'adresse terminaison.

Plus précisément le paramètre ***pid*** permet de sélectionner le processus attendu de la manière suivante :

- <-1 tout processus fils dans le groupe | ***pid*** |
- -1 tout processus fils
- 0 tout processus fils du même groupe que l'appelant
- > 0 processus fils d'identité ***pid***

Le paramètre **options** est une combinaison bit à bit des valeurs suivantes. La valeur WNOHANG permet au processus appelant de ne pas être bloqué si le processus demandé n'est pas terminé.

La fonction renvoie :

- -1 en cas d'erreur,
- 0 en cas d'échec (processus demandé existant mais non terminé) en mode non bloquant,
- 0 le numéro du processus fils zombi pris en compte.

Interprétation de la valeur renvoyée

La valeur de l'entier *terminaison, au retour d'un appel réussi de l'une des primitives wait() ou waitpid(), permet d'obtenir des informations sur la terminaison ou l'état du processus.

De manière générale sous UNIX, pour un processus qui s'est terminé normalement c'est-à-dire par un appel exit(n), l'octet de poids faible de n est récupéré dans le second octet de *terminaison, ce que symbolise le schéma ci-dessous sur une machine où les entiers sont codés sur 32 bits.

Entier n du processus fils.

octet 0	**octet 1**	**octet 2**	**octet 3**

Entier *terminaison récupéré par le processus père

? ? ? ? ? ? ? ?	**octet 0**	**? ? ? ? ? ? ? ?**	**? ? ? ? ? ? ? ?**

Pour un processus qui s'est terminé accidentellement à cause du signal de numéro ***sig***, l'entier *terminaison est égal à ***sig***, éventuellement augmenté de la valeur décimale 128 si une image mémoire (fichier core) du processus a été créée à sa terminaison.

L'interprétation de la valeur récupérée doit, pour des raisons de portabilité être réalisée par l'intermédiaire de macro-fonctions prédéfinies, appelées avec la valeur fournie au retour de l'appel à wait ou waitpid.

Fonction	**Interprétation**
WIFEXITED	Valeur non nulle si le processus s'est terminé normalement.
WEXITSTATUS	Fournit le code de retour du processus si celui-ci s'est terminé nor
WIFSIGNALED	Valeur non nulle si le processus s'est terminé à cause d'un signale.
WTERMSIG	Fournit le numéro du signal ayant provoqué la terminaison du processus.

```
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>

main (void)
{
 pid_t pid ;
 int status ;

 pid = fork () ;

 switch (pid) {
  case -1 :
   perror ("fork") ;
   exit (1) ;
  case 0 : /* le fils */
   printf ("processus fils\n") ;
   exit (2) ;
  default : /* le pere */
   printf ("pere: a cree processus %d\n", pid) ;
   wait (&status) ;
   if (WIFEXITED (status))
    printf ("fils termine normalement: status = %d\n",
          WEXITSTATUS (status)) ;
   else
    printf ("fils termine anormalement\n") ;
 }
}
```

```
% ./status_fils
pere: a cree processus 907
processus fils
fils termine normalement: status = 2
```

4. La primitive system ()

4.1. Description de la primitive system()

int system (char *_ch_)

La primitive **system()** lance l'exécution d'un processus SHELL interprétant la commande passée en argument dans la chaîne de caractères ***ch***. Cette primitive crée pour cela un nouveau processus, qui se termine avec la fin de la commande. Le processus à l'origine de l'appel de **system()** est suspendu jusqu'à la fin de la commande.

5. Recouvrement du code du processus initial

5.1. Principe du recouvrement

La création d'un nouveau processus ne peut se faire que par le « recouvrement » du code d'un processus existant. Cela se fait à l'aide d'une des fonctions de la famille **exec()** qui permet de faire exécuter par un processus un autre programme que celui d'origine.

Lorsqu'un processus exécute un appel **exec()**, il charge un autre programme exécutable en conservant le même environnement système :

- Du point de vue du système, il s'agit toujours du même processus : il a conservé le même ***pid***.
- Du point de vue de l'utilisateur, le processus qui exécute **exec()** disparaît, au profit d'un nouveau processus disposant du même environnement système et qui débute alors son exécution.

Les deux fonctions de base sont:

int execl (char *ref, char *arg0, …, char *argn, 0)
int execv (char *ref, char *argv [ ]);

Il existe d'autres fonctions analogues dont les arguments sont légèrement différents : **execle()**, **execlp()**, **execv()**, **execcve()**, **execvp()**.

Ces fonctions rendent –1 en cas d'échec.

5.2. Primitive execl()

Dans le cas de **execl()**, les arguments de la commande à lancer sont fournis sous la forme d'une liste terminée par un pointeur nul :

- ***ref*** est une chaîne de caractères donnant le chemin absolu du nouveau programme à substituer et à exécuter.
- ***arg0, arg1, …, argn*** sont les arguments du programme.
- Le premier argument, ***arg0***, reprend en fait le nom du programme.

```
void main()
{  execl("/bin/ls", "ls", "-l", NULL);
 printf("Erreur lors de l'appel à ls \n"); }
```

Remarque:

La partie de code qui suit l'appel d'une primitive de la famille **exec()** ne s'exécute pas, car le processus où elle se trouve est remplacé par un autre. Ce code ne sert donc que dans le cas où la primitive **exec()** n'a pas pu lancer le processus de remplacement (le nom du programme était incorrect par exemple).

5.3. Primitive execv()

Dans le cas de **execv()**, les arguments de la commande sont sous la forme d'un vecteur de pointeurs (de type **argv[ ]**) dont chaque élément pointe sur un argument, le vecteur étant terminé par le pointeur NULL :

- ***ref*** est une chaîne de caractères donnant l'adresse du nouveau programme à substituer et à exécuter,
- ***argv [ ]*** contient la liste des arguments.

```
#include <stdio.h>
#define NMAX  5
void main () {
    char *argv [NMAX];
    argv [0] = "ls";
    argv [1] = "-l";
    argv [2] = NULL;
    execv ("/bin/ls", argv);
   printf("Erreur lors de l'appel à ls \n"); }
```

% cat exec1.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>

char *path = "./exec2" ;
char *argval[] = {"exec2", "foo", "bar", NULL} ;

int main (int argc, char *argv[], char *envp[])
{
 pid_t pid ;

 switch (retour = fork()) {
   case -1 :
     perror ("fork") ;
     exit (1) ;
```

```
  case 0 :
    printf ("processus fils va exécuter exec\n") ;
    execve (path, argval, envp) ;
    perror ("execve") ;
    exit (1) ;
  default :
    printf ("père: a crée processus %d\n", retour) ;
    wait(0) ;
    printf ("père: a reçu terminaison fils\n") ;
 }
}
```

6. Primitives d'accès aux caractéristiques générales d'un processus

Nous rappelons ci-dessous la liste des principales caractéristiques d'un processus que l'on trouve dans le bloc de contrôle UNIX et un certain nombre de primitives permettant leur consultation ou leur modification. A la création d'un processus un très grand nombre de ses attributs sont hérités du processus père, c'est-à-dire ont comme valeur initiale la valeur actuelle des attributs correspondants dans le processus père.

6.1. Identité du processus et celle de son père

Un processus a accès respectivement à son PID et à celui de son père par l'intermédiaire respectivement de la fonction getpid() et getppid().

```
#include <unistd.h>
int getpid(void); int getppid(void);
% cat pid.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int
main (void)
{
 pid_t pid_fils ;

 switch (pid_fils = fork ()) {
  case -1 :
    perror ("Fork ") ;
    exit (1) ;
  case 0 : /* le fils */
    printf ("Je suis le fils, PID %d, PPID %d\n",
          getpid (), getppid ()) ;
    exit (0) ;
  default : /* le pere */
    printf ("Je suis le pere, PID %d. Mon fils a le pid %d\n",
          getpid (), pid_fils);
```

```
    (void) wait (NULL) ;
    exit (0) ;
  }
}
```

```
% ./pid
Je suis le pere, PID 8078. Mon fils a le pid 8079
Je suis le fils, PID 8079, PPID 8078
```

6.2. Répertoire de travail du processus

Deux fonctions sont disponibles pour obtenir et changer le répertoire de travail d'un processus. La connaissance de ce répertoire est importante, car tous les chemins relatifs utilisés sont exprimés par rapport à celui-ci. La fonction chdir() permet de changer le répertoire de travail, la fonction getcwd() de récupérer le chemin absolu du répertoire de travail courant.

#include <unistd.h> int chdir(const char *chemin);
char * getcwd(char * buf, unsigned long taille);

Le paramètre taille correspond à la longueur maximale du chemin à récupérer. Si cette longueur n'est pas suffisante, la fonction va échouer.

6.3. Masque de création des fichiers

Le masque de création de fichiers umask() permet d'interdire le positionnement de certaines permissions. Les permissions demandées lors de création de fichiers par les commandes creat ou open, sont « masquées » par le umask. La fonction umask() permet de modifier la valeur du « umask » tout en retournant la valeur du masque précédent.

#include <unistd.h> unsigned int umask(unsigned int masque);

6.4. La table des descripteurs de fichier

Chaque processus peut posséder à un instant donné un maximum de OPEN_MAX (constante définie dans <limits.h>) descripteurs de fichiers. Un processus hérite à sa création de tous les descripteurs de son père. Il a donc accès aux mêmes entrées/sorties. Il peut acquérir de nouveaux descripteurs par l'intermédiaire des primitives standards (open, creat, dup, ..) ou en libérer par la primitive close.

6.5. Liens du processus avec les utilisateurs

Ces liens unissent le processus :

- d'une part à des utilisateurs individuels (identifiés par des constantes de type uid_t)
- et d'autre part à des groupes d'utilisateurs (identifiés par des constantes de type gid_t)

Il existe 4 liens différents étant tous hérités du processus père à la création du processus :

- Le propriétaire réel.

- Le propriétaire effectif. x Le groupe propriétaire réel.
- Le groupe propriétaire effectif. Ces valeurs peuvent éventuellement être modifiées au cours de l'exécution du processus

Différence entre propriétaire réel et effectif

Le propriétaire réel est l'utilisateur ayant lancé le processus. Le propriétaire effectif est utilisé pour vérifier certains droits d'accès du processus vis-à-vis des différents objets du système (fichiers et envoi de signaux à destination d'autres processus). Normalement le propriétaire effectif correspond au numéro du propriétaire réel. On peut cependant positionner l'utilisateur effectif d'un processus au propriétaire du fichier exécutable à l'aide du set-uid bit. Si celui-ci a été positionné sur le fichier exécutable, alors le fichier s'exécutera avec les droits du propriétaire et non avec les droits de celui qui l'a lancé. Cette technique permet notamment d'autoriser exceptionnellement un utilisateur quelconque à lancer un programme système (nécessitant les droits d'exécution root) sans avoir à se connecter en tant que root.

Consultation des valeurs

La fonction getuid() (respectivement getgid()) retourne le numéro d'identification de l'utilisateur (respectivement le numéro d'identification du groupe) à qui appartient le processus. Ce numéro peut être différent de celui du propriétaire du fichier exécutable.

#include <unistd.h> #include <sys/types.h> int getuid(void);
int getgid(void);

Il en est de même pour le propriétaire effectif au travers des fonctions geteuid et getegid.

#include <unistd.h> #include <sys/types.h> int geteuid(void);
int getegid(void);

Changement des valeurs

La fonction setuid() (respectivement setgid()) permet de modifier à la fois le propriétaire réel et le propriétaire effectif d'un processus. Cette possibilité est offerte en générale au superutilisateur du système (root dont le numéro d'identification est 0).

C'est de cette manière que sont initialisées les valeurs des propriétaires des processus SHELL créés au cours de la procédure login de connexion des utilisateurs.

#include <unistd.h> #include <sys/types.h> int setuid (int uid)
int setgid (int gid)

CHAPITRE TROISIÈME
GESTION DES PROCESSUS
EXERCICES

EXERCICE1 :

Les ordinateurs CDC 6600 pouvaient traiter 10 processus d'E/S simultanément grâce à une forme particulière du modèle du tourniquet appelé partage du processeur. Une commutation de processus se produisait après chaque instruction. La 1ère instruction venait du processus 1, la 2ème du processus 2, etc.. la commutation était effectuée par le matériel et le temps utilisé par le système était égal à 0. Si un processus a besoin de T secondes pour s'exécuter lorsqu'il est seul, quel temps lui faut-il si le processeur est partagé entre N processus ?.

ECERCICE2 :

Des mesures sur un système ont montré que les processus s'exécutent en moyenne pendant un temps T avant de se bloquer sur une E/S. une commutation de processus se fait en S secondes, temps qui est effectivement perdu. Pour un ordonnancement circulaire avec un quantum Q, donner une formule qui évalue l'efficacité du processeur pour les valeurs de Q suivantes :

a) $Q=\infty$ b) $Q>t$ c) $S<Q<t$ d) $Q=S$ e) $Q=0$.

EXERCICE3 :

Cinq travaux A à E arrivent pratiquement en même temps dans un centre de calcul. Leur temps d'exécution respectif est estimé à 10,6,2,4 et 8. Leurs priorités sont 3, 5, 2, 1 et 4, la valeur 5 correspond à la priorité la plus élevée.

Déterminer le temps moyen d'attente pour chacun des algorithmes d'ordonnancement suivants, sans tenir compte du temps perdu lors de la commutation des processus : Tourniquet, ordonnancement avec priorité, premier arrivé premier servi, plus court d'abord.

EXERCICE4 :

Un processus exécuté sur CTSS dure 30 quantums. Combien de va et vient (SWAP) y aurait-il en comptant le premier chargement en mémoire.

EXERCICE5 :

Cinq travaux attendent d'être exécutés. Leurs temps d'exécution sont 9, 6, 3, 5 et X . dans quel ordre doivent-ils être lancés pour minimiser le temps de réponse moyen ?.

EXERCICE6 :

On utilise l'algorithme du vieillissement avec a=1/2 pour prévoir les temps d'exécution. les quatre dernières exécutions ont donné dans l'ordre 40, 20, 40 et 15ms. Quel temps exécution peut-on prévoir pour l'exécution suivante ?.

EXERCICE 7 :

Soit le tableau de processus suivant :

Processus	Temps(mn)	Priorité	Date-Arrivée
P1	5	3	8 :00
P2	4	1	8 :00
P3	3	2	8 :02
P4	6	3	9 :00
P5	2	4	9 :05

On suppose que les priorité sont croissantes.

1) donner 2 diagrammes de Gantt correspondants à l'exécution de ces processus en utilisant les algorithmes SJF et priorité sans préemption.
2) donner 3 diagrammes de Gantt correspondants à l'exécution de ces processus en utilisant les algorithmes SJF, priorité et Roud Robbin (quantum=1) avec préemption.

Les algorithmes sont accompagnés par le Turnarround moyen et le temps d'attente moyen.

EXERCICE8 :

Supposons qu'un ordinateur a 2M de mémoire vive dont 512K sont occupées par le système d'exploitation (soit ¼ de la mémoire) et où chaque programme occupe 512K.

1- tracer la courbe de la multiprogrammation pour une moyenne des E/S de 60%
2- quelle serait l'amélioration du rendement si on ajoute 1M de mémoire
3- soient 5 programmes donnés par :

Job	Arrivée	Temps CPU
1	9 :30	5
2	9 :37	3
3	9 :50	7
4	9 :55	2
5	10 :05	2

a) donner la table d'utilisation du processeur par les 5 programmes pour un temps d'attente moyen des E/S de 60%
b) donner le graphe des séquences des événements liés à l'exécution des programmes spécifiant les durées d'exécution.

EXERCICE9 :

Job	**Temps d'arrivés**	Temps CPU

1	8 :00	6
2	8 :10	5
3	8 :15	4
4	8 :17	6
5	8 :23	3

Le système fonctionne à 40% d'E/S.

1) Donner le tableau d'utilisation du CPU
2) Donner les séquences d'événements pour les cinq jobs.
3) Quel est le nombre minimal de jobs qu'il faut en mémoire pour maximiser le taux d'utilisation du CPU ? quelle est la taille de la mémoire correspondante si la taille moyenne des processus est de 50K ?
4) Ces processus sont exécutés dans 256K de mémoire centrale avec P1=45K, P2=55K, P3=70K, P4=22K, P5=64K. ce système utilise la formule $U_{n+1}=U_n+2U_{n-1}$ avec $U_0=1$ et $U_1=2$, représenter par un tableau les différentes séquences du Buddy System à 8h25. combien de blocs restent-ils et quelles sont les tailles et les adresses à 8h20 et 8h25.

EXERCICE10 :

Soient 5 processus

Processus	Temps d'arrivée	Temps CPU	Priorité
1	9 :30	8	3
2	9 :30	6	2
3	α	β	δ
4	9 :40	6	2
5	9 :40	8	3

1) pour β=4, trouver α tel que le turnarround moyen soit minimal pour le SJF préemptif.
2) Pour α=9h35 et β=4, trouver δ tel que le temps d'attente moyen soit minimal pour le priorité préemptif.
3) Pour q=2 et β=4, donner α maximal pour que le 3ème processus se termine le 3ème avec le RR.
4) Pour q=2 et α=9h35, donner β maximal pour que le 3ème processus se termine le 1er.

EXERCICE11 :

Soit un MFQS à n files F_i i=1..n, à chaque F_i est associé un quantum $q_i=2*q_{i-1}$. Les nouveaux processus entrent dans F_1. Le processus en tête de Fi ne peut être pris en compte que si toutes les Fj (0<j<i) sont vides. Si un processus de Fi n'est pas terminé au bout de qi, il passe dans Fi+1, le processus sortant de Fn y retourne. Le système est préemptif, ce qui permet la prise en compte d'un processus arrivant dans Fi une fois le quantum qj de Fj (i<j) expire.

Soit le tableau des processus suivant :

Processus	Temps CPU	Arrivée
P1	6	8 :00
P2	5	8 :15
P3	3	8 :15
P4	4	8 :10
P5	6	8 :10
P6	7	8 :05
P7	3	8 :02
P8	6	8 :07

1) Donner le diagramme de Gantt pour un SJF préemptif et son turnarround
2) Donner le diagramme de Gantt et le turnarroud pour n=2 et q=3
3) Donner le diagramme de Gantt et son turnarround pour n=3 et q=1
4) Pourquoi les deux systèmes précédents se comportent-ils de la même manière.
5) Que se passe-t-il pour n>3 ?

EXERCICE12:

(A) Soit un système à processeur unique et soient 5 processus s'exécutant en multiprogrammation, et soit le tableau de processus suivant :

Job	Temps d'exécution	Temps d'arrivée	Priorité
P1	10	9h: 30	3
P2	3	9h: 34	4
P3	7	9h: 32	2
P4	6	9h: 40	2
P5	4	9h: 41	1

A-1) Donner deux (2) diagrammes de Gantt correspondants à l'exécution de ces processus en :

SJF (le plus court d'abord) avec préemption

Round Robbin avec le quantum q= 2

A-2) Calculer le Temps turnaround moyen (temps de réponse) pour chacun des algorithmes.

Calculer le temps d'attente moyen pour chacun des algorithmes.

B) Soit l'algorithme SETF "Shortest Elapsed Time First" décrit comme suit:

- Un processus reçoit un seul quantum à la fois (sans réquisition)
- Un processus libère le CPU:
  - Soit parce qu'il se termine
  - Soit son quantum est terminé.
- Dès que le CPU devient libre, **on active le processus ayant reçu jusqu'à présent le moins de quantum**.

Soit le tableau suivant :

Job	Temps d'exécution	Temps d'arrivée
P1	8:00	10

P2	8:00	5
P3	8:04	8
P4	8:06	6
P5	8:12	4

Donner le diagramme de Gantt correspondant et déterminer le temps Turnaround moyen

C) Soit le diagramme de Gantt suivant:

P1	P2	P4	P2	P5	P1	P3
8h:00	8h:02	8h:03	8h:05	8h:06	8h:10	8h:***

Et soit le tableau des Temps d'arrivée de chaque processus :

Temps-arrivée	8:00	8:02	8:00	8:03	8:05

Déterminer pour le diagramme de Gantt donné plus haut, l'algorithme d'ordonnancement appliqué.

EXERCICE13 :

Soit l'algorithme **SETF** « Shortest Elapsed Time First » décrit comme suit :
Un processus reçoit un seul quantum à la fois sans réquisition pendant le quantum. Un processus libère le CPU lorsqu'il a terminé ou a épuisé son quantum. Dés que le CPU devient libre, on active le processus ayant reçu jusqu'à présent le moins de quantum.
Et soit le tableau suivant :

Job	Arrivée CPU	temps
P1	**8h :00**	**5**
P2	**8h :00**	**8**
P3	**8h :02**	**4**
P4	**8h :05**	**6**
P5	**8h :10**	**4**
P6	**8h :10**	**6**

1) Donner le diagramme de Gantt correspondant et déterminer le temps d'attente moyen.
2) Cet algorithme s'implémente, en réalité et plus simplement comme un **MFQS** à plusieurs files.
Décrire le **MFQS**, dans sa forme générale, équivalent au **SETF** avec un nombre de files **n**

- décrire l'algorithme de chaque file
- expliquer le passage d'un processus d'une file à une autre.

3) Appliquer le MFQS à l'exemple du tableau

- bien déterminer **n** de façon à éviter les files inutiles
- bien déterminer le quantum pour chaque file
- bien expliquer le passage d'une file à une autre

4) Pour valider votre MFQS, déterminer son diagramme de Gantt et son temps d'attente moyen.

5) Comparer les résultats (temps d'attente) et dire lequel est meilleur. Expliquer ?

EXERCICE14 :

Soit le tableau suivant :

Job	Arrivée	temps CPU
P1	8h :00	4
P2	8h :00	6
P3	8h :05	5
P4	8h :10	6
P5	8h :10	4

1. Compléter le diagramme de Gantt de la réponse 1.
2. de quel algorithme s'agit-il ?

Diagramme de Gantt

P1	P2	P1	P2	P1	P3	P	P2	P3	P	P4	P5	P	P	P4	P5	P2	P3	P4	P5	P	P	P4	P	P
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Exercice15 :

Soit le tableau suivant :

Job	Temps CPU	Priorité	Arrivée	Mémoire
P1	4	2	8:00	10k
P2	6	4	8:00	12k
P3	a	b	8:05	c
P4	6	4	8:10	12k
P5	4	2	8:10	10k

a) Pour b=2 donner le diagramme de Gantt par priorité avec 50% d'E/S

b) Pour b=4 même question

c) Donner le turnarround des questions précédentes et trouver a tel que ces turnarround soient égaux.

d) Pour a précédent et b=3 donner les séquences des évenements en supposant que les jobs arrivent avec la priorité pi(tableau) et chaque job prenne un temps pi*t du temps CPU.

Exercice16 :

Soit un système à 4 processeurs M1, M2, M3 et M4. Chaque jod s'exécute sur les processeurs dans le même ordre M1, M2, M3, M4. M1 utilise le Round Robbin(q=2), M2 le SJF, M3 Priority et M4 le Round Robbin (q=1). Soit le tableau suivant :

				Temps CPU			
Job	**Arrivée**	**Priorité**	**Mémoire**	**M1**	**M2**	**M3**	**M4**
J1	8h :02	3	8k	4	5	8	9
J2	8h :00	3	9k	2	3	2	2
J3	8h :06	4	12k	5	2	3	3

J4	8h :04	2	10k	2	3	2	3
J5	8h :07	3	6k	2	1	3	2

1) Donner le diagramme de Gant de chaque machine, le turnaround et le temps d'attente.
2) Ce système est muni d'une mémoire commune de 40k, gérée par partitions fixes. Chaque machine charge toute seule ses jobs. Un job terminé sur une machine retourne en mémoire secondaire. Les machines ont une priorité (les jobs de M1 sont chargés avant ceux de M2, M2 avant M3 et M3 avant M4). Le système est non préemptif. R-R, SJF, PR sont résidents comme indiqué sur le schéma ci-dessous.

   Donner l'état de la mémoire à travers le temps, le diagramme de Gant de chaque machine équivalant, le turnaround et le temps d'attente.

3) le système est comme précédemment, mais cette fois-ci, il est géré par le First-Fit. R-R, SJF, PR appartiennent à l'OS résident en mémoire selon le schéma ci-dessous.

Exercice17 :

Soit le tableau suivant :

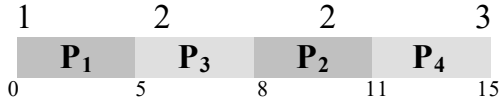
Processus	Arrivée	Temps CPU	Priorité	%E/S
P1	8 :00	8	2	50
P2	8 :03	7	1	30
P3	8 :05	6	3	40
P4	8 :10	9	1	40
P5	8 :12	4	3	50
P6	8 :15	5	2	30

Soit un MFQS à 3 files F1, F2, et F3. les processus entrent tous dans F1 et chacun grimpe jusqu'à la file correspondante à sa priorité. Un processus dans une file Fi prend un quantum qi=(p-i+1)*q où p est la priorité du processus et q est donné. Les files s'exécutent en R-R le quantum q égal au quantum du processus élu.

1- Donner le diagramme de Gantt pour le R-R simple avec q=2
2- Donner le diagramme de Gantt pour le MFQS avec q=1
3- Donner la séquence des événements en R-R simple avec les % E/S donné au tableau q=2
4- Pour une moyenne des E/S calculée selon le tableau, donner la séquence des événements pour le MFQS.

Exercice18 :

Partie 1 : Pour le diagramme de Gantt :



Et les Algorithmes :

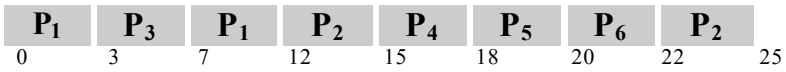
1-FCFS
2-SJF Non préemptif
3-SJF Préemptif
4-Priority Non préemptif
5-Priority Préemptif
6-Aucun

Pour le Tableau suivant :

P1	P2	P3	P4	algorithmes	TM
8h:00	8h:05	8h:02	8h:10	1 2 3 4 5 6	
8h:00	8h:05	8h:00	8h:05	1 2 3 4 5 6	
8h:00	8h:03	8h:02	8h:01	1 2 3 4 5 6	
8h:00	8h:02	8h:03	8h:00	1 2 3 4 5 6	

1- Pour les temps donnés d'arrivée, déterminer pour chaque ligne les algorithmes qui répondent au diagramme de Gantt (Noter que plusieurs algorithmes peuvent répondre). Barrer les numéros des algorithmes qui ne répondent pas.
2- Déterminer le Turnaround Moyen T M correspondant.

Partie 2 : Soit le diagramme de Gantt



1- Déterminer les plus petits temps d'arrivée des processus pour qu'ils s'exécutent en **SJF** selon le diagramme précédent.
2- Déterminer les plus petites priorités et les plus petits temps d'arrivée des processus pour qu'ils s'exécutent par **PRIORITY** selon le diagramme précédent.