1. Notion de processus

On considère le cas d'un ordinateur comportant un processeur ***unique*** et une mémoire adressable par ce processeur. L'état de la machine est modifié par l'exécution d'instructions par le processeur. L'exécution d'une instruction donne lieu à une action. L'action est définie par un état initial et un état final.
On suppose les actions *indivisibles ou atomiques* : on ne peut observer l'état de la machine pendant cette exécution. Seuls le début et la fin de l'exécution sont des *points observables* (donc interruptibles). Un événement est un état associé à une date donnée :

(Point observable, date) = événement

Une action « a » est caractérisée par 2 événements : début(a) et fin(a) tels que début(a) < fin(a).

L'exécution d'un programme donnera une suite d'actions correspondant aux instructions qu'il contient :
a1,a2,a3, …,an avec début(ai+1)>fin(ai)

Cette suite est dite processus séquentiel ou processus du programme.
La suite début(ai),fin(ai) est dite la trace temporelle (ou histoire) du processus ainsi défini.

D'une manière non formelle, un processus est un programme en exécution. Mais nous allons voir que cette définition n'est pas complète.

2. Contexte d'un processus

C'est l'ensemble des informations que les actions du processus peuvent consulter ou modifier. Ces informations sont :

1. Contexte processeur (mot d'état et registres généraux) ;
2. Contexte mémoire : c'est l'espace de travail qui comprend :
   - Les segments procédures
   - Les segments données
   - Les piles d'exécution.

3. Ensembles d'attributs du processus :
   - **Nom** : nom interne désignant le PCB du processus, qui sera discuté plus tard
   - **Priorité** : en fonction du scheduling (discuté dans le chapitre suivant)
   - **Droits** : opérations permises par le processus

3. Relations entre processus

Les contextes des différents processus peuvent avoir des parties communes. Deux processus ayant des contextes différents sont dits *indépendants.* Dans le cas contraire, ils sont dits *dépendants.*

La partie du contexte qui n'appartient à aucun autre processus est dite **contexte privé**. Les processus peuvent s'échanger de l'information en s'envoyant des messages (IPC ou communication inter processus).

4. Modes d'exécution des processus

Afin d'exécuter des programmes, il faut mettre à leur disposition les ressources nécessaires à leur exécution :

- Processeur
- Espace mémoire
- Données nécessaires.

Une solution consiste à exécuter un seul processus à la fois et à lui allouer les ressources nécessaires. Un autre programme n'est alors exécuté qu'à la fin du précèdent. Le schéma 1 résume ce mode d'exécution **séquentiel**. Cette solution est coûteuse car elle monopolise toute la machine pour un seul programme.

Une autre solution est d'exécuter un ensemble de processus en mode dit " en **compétition** " ou " en concurrence ". Cette solution repose sur le schéma 2.

Enfin, on peut exécuter les programmes **en parallèle**, mais ceci nécessite autant de *processeurs* que de programmes. Le schéma 3 résume cette situation.

Schéma 1 : Exécution entièrement séquentielle

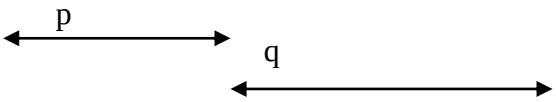


Schéma 2 : Exécution alternée ou pseudo-parallélisme

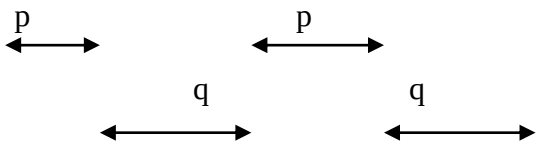
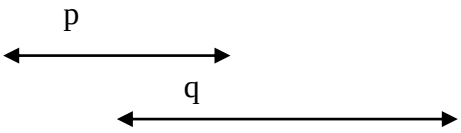


Schéma 3 : Exécution en parallélisme réel



5. Les niveaux d'observation

Un processus P est une suite d'actions a_i. Donc : P=a1, a2, a3, …, an
On peut écrire :

P= (a1, a2),(a3,a4), …, (an-1,an)

Ou encore : P=A1,A2,A3, … ,Am
Où Aj=(ai,ai+1)

Les unités d'exécution Aj sont *moins fines* que les ai. Les unités d'exécution les plus fines possibles sont dites unités d'exécution (ou actions) élémentaires ou atomiques. Le niveau d'observation *le plus fin* est dit **niveau de base**.

Ainsi, si l'exécution d'un programme est, pour l'utilisateur, une ***action unique*** (observable en tant que tel), alors les **schémas 2 et 3** sont ***identiques*** (exécution **parallèle**) et l'on a :

Fin(p)>deb(q) et fin(q) > deb(p)

Le **schéma 1** est celui d'une exécution séquentielle.
Par contre, au ***niveau de base***, les schémas **2 et 3** ***sont différents*** (2 est **séquentiel**, 3 est **parallèle**). En revanche, les schémas **1 et 2** sont ***identiques*** (exécution **séquentielle**).

6. Ordonnancement des processus

Quand un processus rentre dans le système, il est placé dans une file d'attente de travaux ou **pool des travaux**, se trouvant dans la mémoire secondaire (un disque par exemple). Elle contient tous les processus du système.

Les processus résidents en mémoire et qui sont prêts à s'exécuter sont maintenus dans une **file d'attente des processus prêts**.

Le PCB d'un processus possède un pointeur vers le PCB du processus suivant dans la file.
Les processus en attente d'une E/S sont mis dans une file d'attente rattachée au périphérique correspondant : **file d'attente du périphérique**.

Un nouveau processus est placé dans la file d'attente des processus prêts. Il sera sélectionné (distribué) pour son exécution. Dans ce cas, un des événements suivants peut se produire :

- Le processus pourrait émettre une requête d'E/S et sera mis dans une file d'attente d'E/S
- Le processus pourrait créer un processus fils et attendre sa fin
- Le processus pourrait être enlevé de force de l'UC (interruption) et remis dans la file d'attente des processus prêts.

Un processus passe entre les diverses files d'attente d'ordonnancement ou **scheduling** pendant toute sa durée de vie. Il doit être sélectionné à partir de ces files d'attente par le SE. Le processus de sélection s'appelle le ***scheduleur.***

Système de traitement par lots : les travaux sont nombreux pour être exécutés simultanément. Ces processus sont « spoolés » sur une mémoire secondaire (un disque) où ils sont maintenus afin d'être exécutés plus tard. Le **scheduleur « à long terme »** ou scheduleur de travaux sélectionne les travaux de ce pool et les charge en mémoire centrale pour être exécutés.

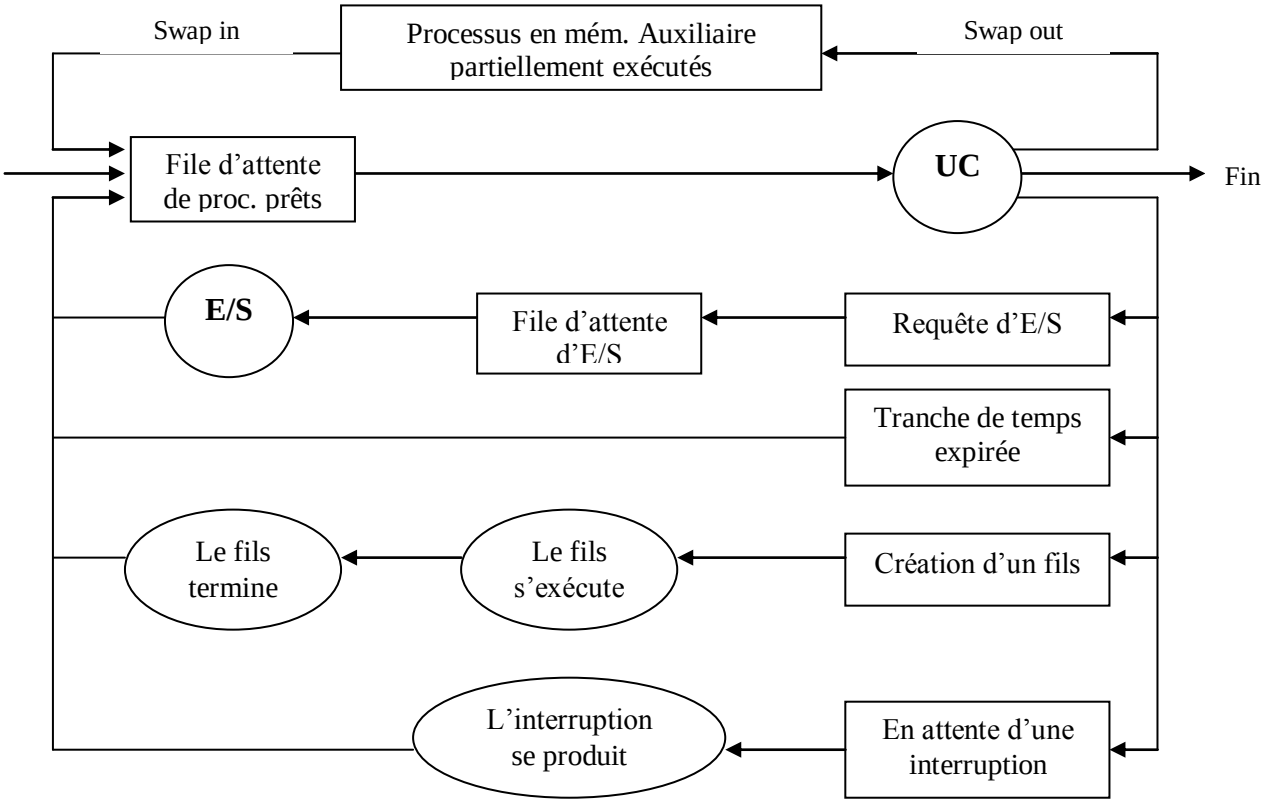


Fig. 1 : Représentation du diagramme des files d'attente du scheduling des processus

Système multiprogrammé : il existe plusieurs processus en mémoire centrale. Le **scheduleur « à court terme »** ou scheduleur de l'UC choisit parmi les processus prêts et alloue l'UC à l'un d'eux.

La différence entre les deux scheduleurs est la fréquence de leurs exécutions : le scheduleur à court terme sélectionne à raison d'un processus/ms à 100 ms. Le scheduleur à long terme s'exécute beaucoup moins fréquemment (plusieurs minutes) ; il mesure le degré de multiprogrammation (nombre de processus en mémoire) ; un degré stable veut dire :

Vitesse de création de processus
=
vitesse de départ de processus.

Le scheduleur à long terme doit faire un « bon mélange » de processus ; en effet, un processus est soit tributaire de l'UC ou des E/S. Si tous les processus sont tributaires des E/S, la file d'attente des processus prêts sera toujours presque vide et le scheduleur à court terme ne fera presque rien. En revanche, si tous les processus sont tributaires de l'UC, la file d'attente des E/S sera toujours presque vide, les périphériques resteront inutilisés.
Dans les deux cas, le système n'est pas équilibré.

Cas particulier : Systèmes en *temps partagé.*
En général, le SLT est absent. Seul le SCT fonctionne. Ainsi, chaque processus est placé dans la mémoire centrale pour le SCC.

La stabilité de ces systèmes dépend des limitations physiques (nombre de terminaux, consensus des utilisateurs). Cependant, ces systèmes peuvent avoir recours à un intermédiaire, le **scheduleur « à moyen terme ».**

Principe : Supprimer momentanément des processus de la mémoire (à la demande de l'UC) pour réduire le taux de multiprogrammation. Plus tard, le processus sera réintroduit dans la mémoire centrale et son exécution peut reprendre la ou elle s'était arrêtée. Ce schéma est celui du ***swapping*** (to swap=échanger). On utilise la mémoire auxiliaire (disque) pour réaliser le swapping.

7. Etats des processus

Il est important de distinguer entre un processus et le programme qui lui correspond. On dit que le programme (représenté par un *code source* ou *objet*) est la ***section texte*** du processus. Nous mettons l'accent sur le fait qu'un programme par lui-même n'est pas un processus ; un programme est une entité ***passive***, comme les contenus d'un fichier stocké sur disque, tandis qu'un processus est une entité ***active*** évoluant dans le temps.
Un processus peut connaître quatre états en *ordonnancement court terme*:

- **Bloqué** (blocked) : état d'un processus qui attend l'occurrence d'un événement, comme une fin d'E/S, avant de pouvoir continuer à s'exécuter.
- **Prêt** (ready) : état d'un processus qui n'est pas alloué à l'UC, mais qui est prêt à être exécuté. Il se trouve toujours en mémoire centrale.
- **En exécution** (running) : état d'un processus exécuté sur une UC.
- **Terminé** (terminated) : état d'un processus qui a terminé son exécution, mais dont l'enregistrement est conservé par le système d'exploitation.

Il existe aussi deux états en *ordonnancement moyen terme* :

- **Permuté – bloqué** (swapped – blocked) : état d'un processus qui attend l'occurrence d'un événement et qui a été extrait vers la mémoire auxiliaire.
- **Permuté – prêt** (swapped – ready) : état d'un processus qui est prêt à être exécuté, mais qui est placé dans la mémoire auxiliaire.

Et aussi un état en *ordonnancement long terme* :

- **Suspendu** (held) : état d'un processus qui a été créé, mais qui n'est pas pris en compte pour le chargement en mémoire ou l'exécution. C'est un nouveau processus. Cet état ne peut être vécu qu'une seule fois par un processus.

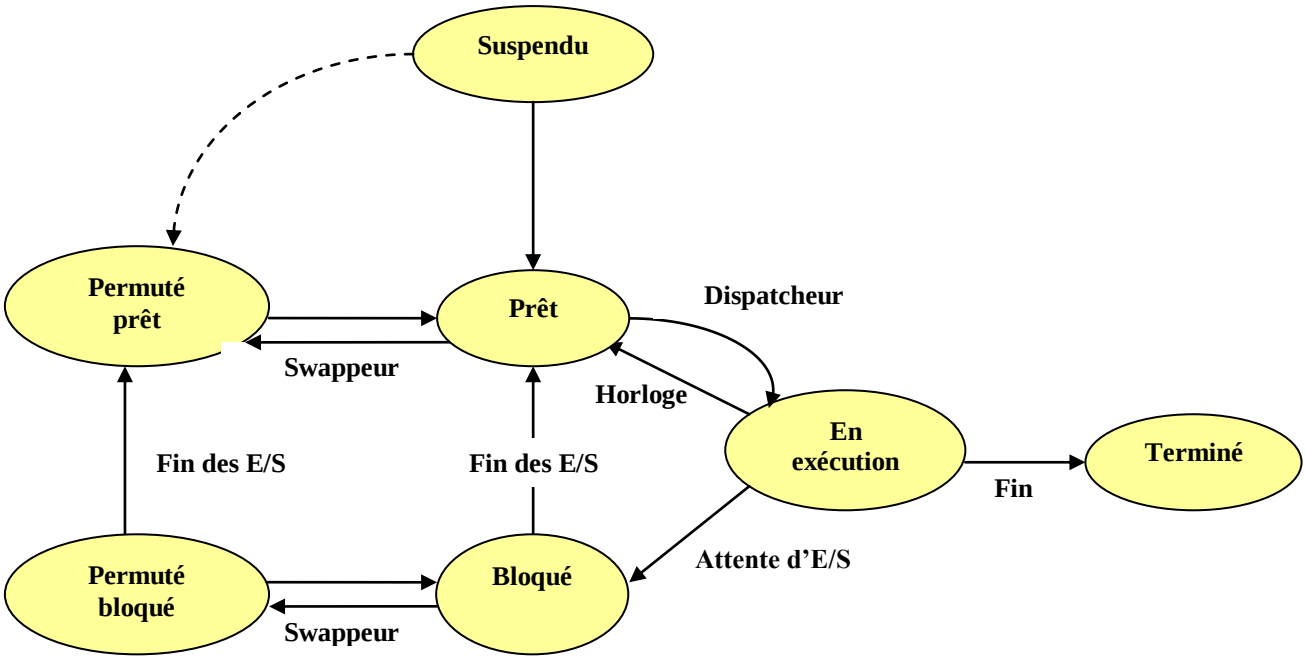


Fig. 2 : Schéma de transition entre les états des processus

8. Bloc de contrôle de processus (PCB)

Chaque processus est représenté dans le système d'exploitation par un **bloc de contrôle de processus (en anglais " Process Control Bloc " ou PCB).** Concrètement, le nom d'un processus est le nom de son PCB : *le processus n'existe que par son PCB*.
Le PCB contient plusieurs informations concernant le processus, parmi lesquelles nous citons :

- **L'état du processus** : nouveau, prêt, en exécution, en attente, arrêté, etc.
- **Le compteur d'instructions** : le compteur indique l'adresse de l'instruction suivante devant être exécutées par ce processus ;
- **Les registres de l'UC** : les registres varient en nombre et en type selon l'architecture de l'ordinateur. Ils englobent des registres du *processeur* (accumulateurs, registres d'index, pointeurs de pile, registres généraux). Les contenus de ces registres doivent être sauvegardés lors de chaque interruption pour permettre au processus de continuer ;
- **Informations sur le scheduling de l'UC** : ces informations comprennent la priorité du processus, des pointeurs sur les files d'attente de scheduling (discutées dans le chapitre suivant) ;
- **Informations sur la gestion mémoire** : ces informations peuvent inclure les valeurs des registres base et limite, les tables de pages ou tables de segments selon le système de mémoire utilisé (discutées lors du chapitre sur la mémoire) ;
- **Informations de comptabilisation** : ces informations tiennent en compte la quantité de temps processeur et temps réel utilisés, les limites de temps, les numéros de compte (des utilisateurs), les numéros des travaux ou de processus, etc ;
- **Informations sur l'état des entrées/sorties** : l'information englobe la liste de périphériques d'entrées/sorties (comme les unités de bande, les unités de disque) allouées à ce processus, une liste de fichiers ouverts, etc.

En clair, le PCB sert simplement comme gisement pour toute information pouvant varier d'un processus à un autre.

Commutation de contexte (context switching)

Le fait de commuter l'UC sur un autre processus demande de sauvegarder l'état de l'ancien processus (son PCB) et de charger l'état sauvegardé du nouveau processus : c'est la **commutation de contexte**. Elle varie, en temps, entre 1 et 1000 µs, selon les systèmes. Pendant la commutation, le système ne fait rien.
Dans les systèmes à ensembles multiples de registres, la commutation consiste à changer le pointeur vers l'ensemble courant de registres. Dans la plupart des systèmes, on utilise simplement la mémoire centrale pour commuter.

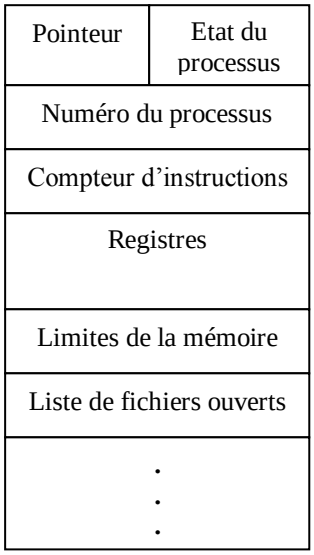


Fig. 3 : Bloc de contrôle d'un processus

9. Opérations sur les processus

Un processus est vu comme un objet par le système d'exploitation. Cet objet est identifié par un identificateur unique (PID) et sur lequel on peut appliquer les opérations suivantes :

- Création
- Destruction
- Blocage
- Réveil
- Activation (par le dispatcher)
- Désactivation (fin de quantum)
- Suspension …

Un noyau de système se compose de toutes ces opérations.

9.1. Création de processus

Cette opération consiste à :

- Allouer un descripteur à un processus (PCB)
- Affecter un identificateur unique au processus (PID)
- Initialiser le descripteur (PCB) : programme à exécuter, pile d'exécution, mémoire centrale utilisé par le programme et les données du processus, état du processus, priorité du processus, autres ressources…etc.

La création de processus peut se limiter aux processus système ou généralisée à tous les processus. Dans ce cas, pendant son exécution, un processus peut créer plusieurs nouveaux processus en exécutant des appels système de création de processus. Le processus effectuant la création est dit **processus père**, tandis que les nouveaux processus sont ses **fils.** Chacun des nouveaux processus peut encore créer d'autres processus (fils) créant ainsi un arbre de processus. Un processus père doit toujours connaître l'identité de ses fils.

9.2. Destruction de processus

La destruction d'un processus ne peut être effectuée que par le processus père ou un processus système. Cette opération consiste à :

- Libérer les ressources occupées par le processus
- Détruire ou non la descendance du processus (Unix ne détruit pas les processus fils)
- Libérer son descripteur (PCB), qui pourra être réutilisé
- Le PID est supprimé mais ne peut plus être réutilisé

9.3. Exemple d'Unix

Sous **Unix**, chaque processus est identifié par son identificateur de processus ou **PID**, un entier unique. L'appel système **fork()** est utilisé pour créer un processus fils. L'appel système **execve()** est utilisé après fork pour remplacer le contexte du processus courant par celui du nouveau processus. L'espace mémoire du fils remplace celui du père permettant ainsi aux deux processus de communiquer. L'appel système **wait()** permet au père de sortir de la file des processus prêt jusqu'à la fin du fils.

Quand un processus exécute sa dernière instruction, il sort du système en faisant un appel système **exit()**. Enfin, un processus père peut terminer brutalement un fils en effectuant l'appel système **abort()**. Les utilisateurs ne peuvent pas tuer des processus fils (instruction **kill**).

10. Les threads

Un processus est défini par les ressources qu'il utilise et par l'emplacement (mémoire) dans lequel il s'exécute.
Le partage de ressources entre processus et l'exécution dans le même espace d'adresses est un challenge pour les systèmes multiprocessus.

10.1. Définition

Un **thread** ou processus de poids léger (LWP = Light Weight Process) est une unité de base d'utilisation de l'UC.
Comme un processus, un thread possède un compteur d'instructions, des registres et un espace pile.
Un processus défini une tâche. Une tâche est un ensemble de threads. Un thread appartient à une seule tâche.
En particulier, un processus de poids lourd est une tâche avec un seul thread.

10.2. Avantages des threads

- Commutation de contexte moins coûteuse ;
- La commutation de contexte ne nécessite pas un travail de gestion mémoire (on utilise le même espace d'adresses du processus qui contient le thread).

10.3. Threads niveau utilisateur

Ce sont des threads disponibles dans des *bibliothèques* utilisateur (API), au lieu par appels système. Leur avantage est que la commutation de contexte entre threads utilisateurs n'interrompt pas le noyau du système d'exploitation et s'effectue donc plus rapidement.

Application

Serveur de fichiers traitant plusieurs requêtes (une requête est traitée par un thread au lieu par un processus entier ; on bloque un thread effectuant un accès disque et on commute vers un autre pour traiter la requête suivante).

Inconvénient des threads utilisateur

Si le noyau est mono-thread, tout thread qui effectue un appel système bloque toute la tâche (processus) jusqu'à ce que l'appel système retourne car le noyau schedule seulement des processus (il ne connaît pas les threads).

10.4. Multithread Vs Multiprocessus

Dans un noyau multiprocessus :

- Chaque processus est indépendant des autres (contexte different) ;
- Chaque processus possède son propre compteur d'instructions, ses propres registres, pile et espace d'adresses.

Exemple : Serveur de fichiers sur un PC mono-processeur

Si un processus serveur est bloqué par un accès disque, un autre processus serveur (pour le même service) ne peut opérer car il doit utiliser le même espace d'adresse (il doit être indépendant).

Dans un noyau multithreads :

- Les threads fonctionnent comme les processus : ils ont des *états* (bloqué, prêt, en exécution), se partagent l'UC et un seul thread est actif (en exécution) à la fois.
- Chaque thread dans un processus possède son propre compteur d'instructions et sa propre pile.
- Les threads peuvent créer des fils.
- Si un thread est bloqué, un autre peut s'exécuter.

Différence Threads - Processus

- Les threads ne sont pas indépendants :un thread peut accéder à chaque adresse dans la tâche (processus).
- Un thread peut lire ou écrire sur n'importe quelle pile des autres threads (de la tâche).
- Il n'y a pas de protection nécessaire entre les threads d'une tâche : les threads sont conçus pour s'assister entre eux.

Reprenons l'exemple du serveur de fichiers. Si une tâche exécute plusieurs threads, pendant qu'un thread de serveur est bloqué et attend, un deuxième thread dans la même tâche peut s'exécuter d'où une capacité de traitement du serveur supérieure et des performances améliorées : les threads d'un même processus serveur coopèrent.

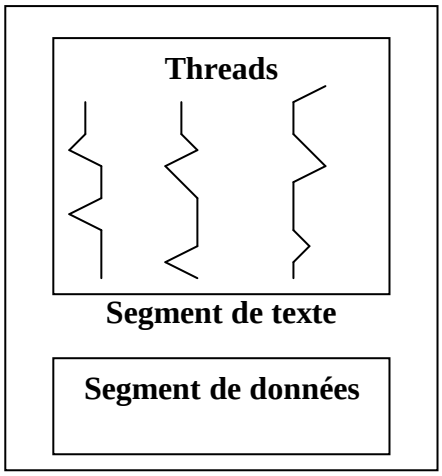


Fig. 4 : Une tâche (processus entier)

10.5. Implémentation des threads

Il existe deux approches d'implémentation :

1. Les threads peuvent être supportés par le noyau. Dans ce cas, les appels système sont similaires aux processus.
2. Les threads peuvent être supportés au-dessus du noyau : ensemble d'appels de bibliothèques au niveau utilisateur.

Exemple

Soit deux processus P_1 et P_2 avec respectivement 1 thread et 100 threads.
Mode thread utilisateur : le thread de P_1 s'exécute 100 fois plus rapidement qu'un thread de P_2.
Mode thread noyau : le processus P_2 reçoit 100 fois plus de temps processeur que P_1. mais une perte de temps en commutation de contexte (commuter 100 fois au lieu de deux fois).

Une solution consiste à faire un **compromis** entre les deux approches. Ainsi, le système Unix de *Sun*, **Solaris** a, pour la première fois implémenté un noyau multithreads avec la capacité de gérer des threads utilisateurs. Solaris gère ainsi deux types de threads : les **threads noyau** et les **threads utilisateur**.