

Systemes d'Exploitations Avancés

Chapitre II :

Gestion des Processus et Threads

Amine DHRAIEF

Mastère professionnel en Modélisation, Bases de
Données et Intégration des Systemes

ESEN, Univ. Manouba

Introduction

- Les **premiers S.E** autorisaient un **seul programme** à être exécuter à la fois. Un tel programme avait un **contrôle complet du système** et un accès à toutes les ressources du système
 - Les S.E **actuels** permettent à **plusieurs programmes** d'être charger en mémoire et exécuter en même temps.
- Cette évolution a nécessité un **contrôle plus strict** et un **cloisonnement plus rigoureux** des différents programmes.

Introduction

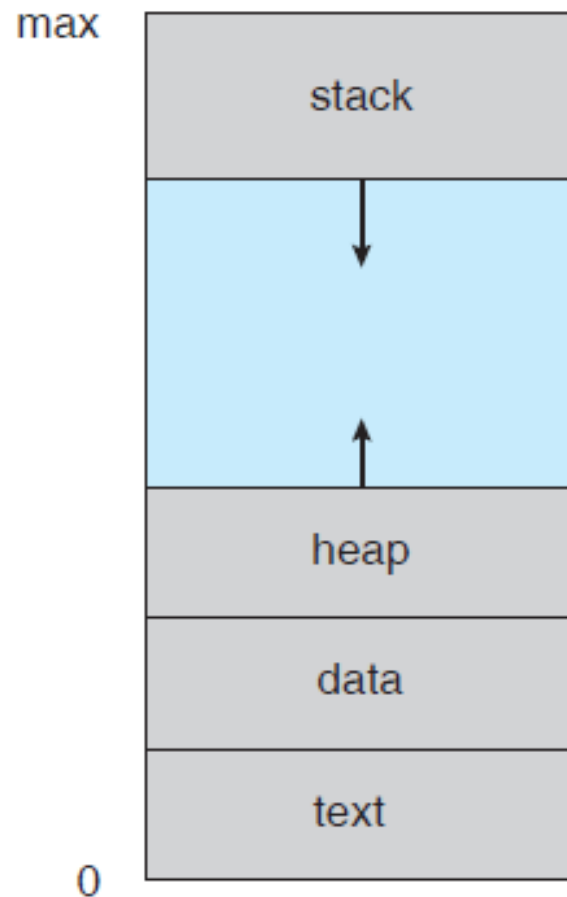
- Ces besoins ont conduit à la création de la notion de **processus**, qui est un « **programme en exécution** ».
 - Un processus est l'unité fondamentale dans le cadre d'un S.E moderne temps partagé.
- Un système se compose d'un ensemble de processus:
 - Les **processus du S.E** exécutent le code du S.E
 - et des **processus utilisateurs** exécutent le code de l'utilisateur.
- Tous ces processus peuvent **s'exécuter en même temps**.
 - En commutant le processeur entre les processus, le système d'exploitation peut rendre l'ordinateur plus productif.

Le Concept de Processus

- Un processus est **souvent** définit comme **étant un programme en exécution**.
 - Un **processus** est plus que le code du programme (*text section*)
- Un processus inclut une représentation de l'activité en cours du programme :
 - Le **contenu des registres du processeur**.
 - Le **program counter** (PC) (un registre qui contient l'adresse mémoire de l'instruction en cours)
- Un processus comprend également
 - **une pile d'exécution** qui contient des données provisoires (tels que la fonction, les paramètres, des adresses de retour, et les variables locales),
 - et une section de données (**data section**), qui contient les variables globales.
- Un processus peut également comprendre un tas (**heap**) qui est un mémoire allouée dynamiquement lors de l'exécution processus.

LE CONCEPT DE PROCESSUS

Le Concept de Processus



**Représentation
du processus en
mémoire**

Le Concept de Processus

Programme vs. Processus

- **Le programme n'est pas en soi un processus,**
 - **un programme est une entité passive**, comme un fichier contenant une liste d'instructions stockées sur le disque (souvent appelé un fichier exécutable),
 - **alors qu'un processus est une entité active**, avec un programme counter spécifiant la prochaine instruction à exécuter et un ensemble de ressources associées.
 - Un **programme** devient un **processus** lorsqu'un fichier exécutable est **chargé en mémoire**.
- Même si **deux processus** peuvent être associés à un même programme, ils sont néanmoins **considérés comme deux séquences d'exécution séparées**.
 - Par exemple, plusieurs utilisateurs peuvent exécuter différentes copies du logiciel de courrier électronique.
 - Chacune d'elles est un processus distinct, et bien que les sections de texte sont équivalentes, les data section, la pile, le heap sont différents.

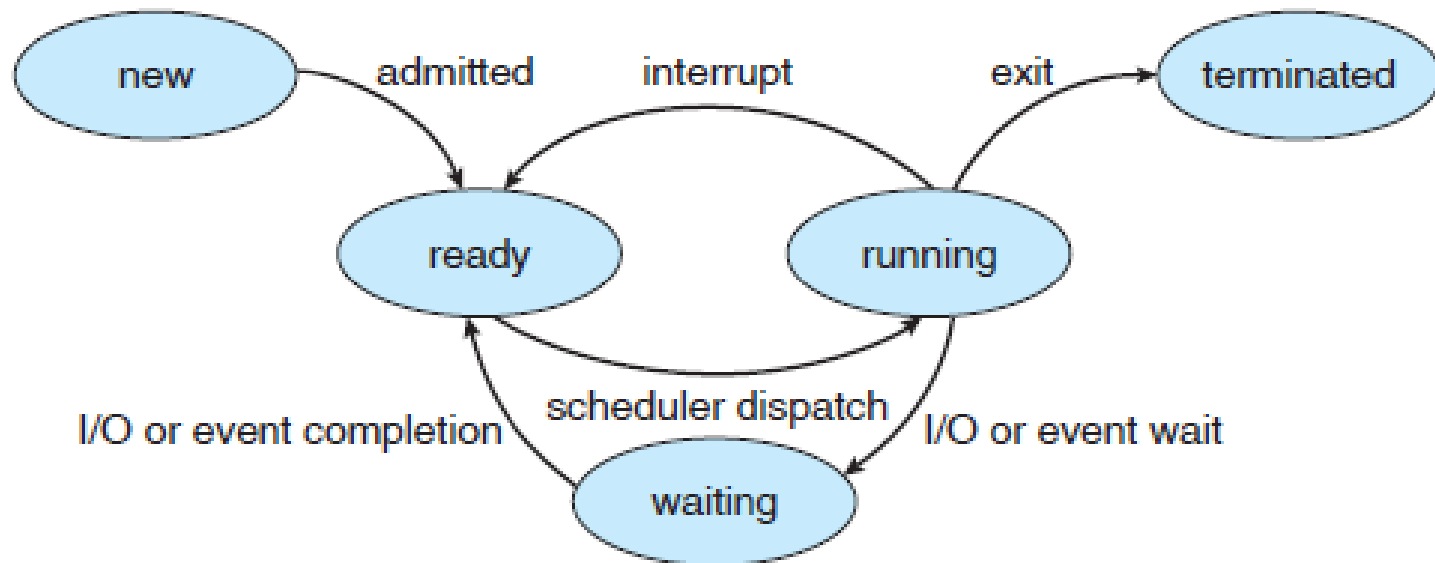
Le Concept de Processus

Les états d'un processus

- Durant son exécution, **un processus change d'état**
- Chaque processus peut être dans l'un des états suivants:
 - **New**: Le processus est en cours de création.
 - **Running**: Les instructions sont en cours d'exécution.
 - **Waiting**: Le processus est en attente d'un événement de se produire (par exemple une E / S, l'achèvement ou la réception d'un signal).
 - **Ready**: Le processus est en attente d'affectation à un processeur.
 - **Terminated**: Le processus a fini d'exécution

Le Concept de Processus

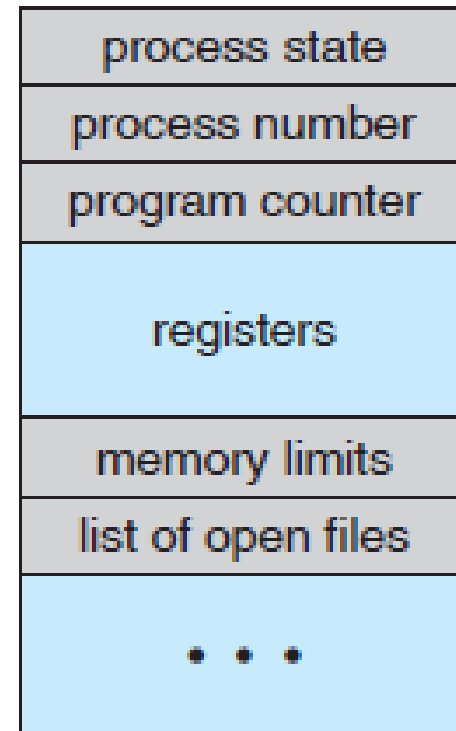
Les états d'un processus



Le Concept de Processus

Process Control Block

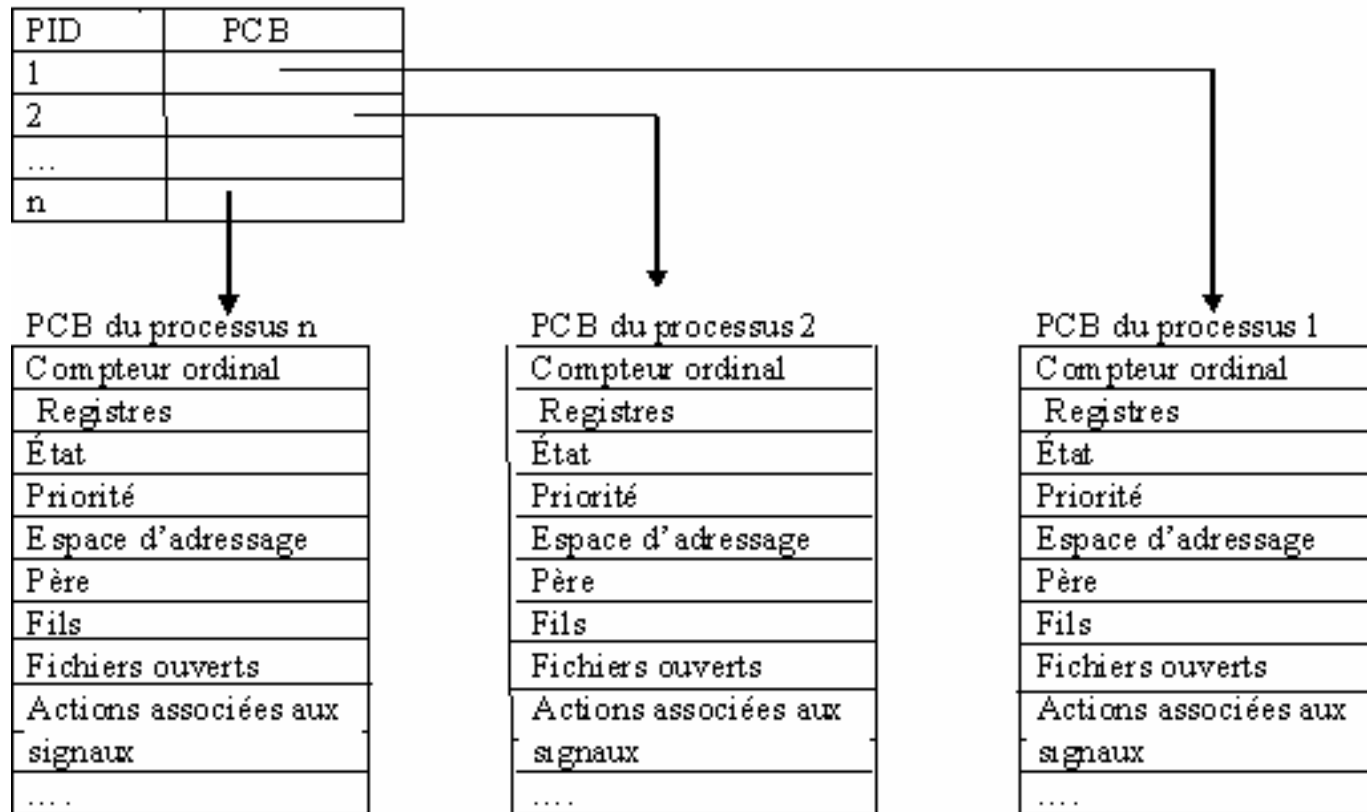
- Chaque processus est représenté dans le système d'exploitation par un bloc de contrôle de processus (**Process Control Block - PCB**) appelé aussi un **task-control block**
- Le système d'exploitation maintient dans une table appelée «**table des processus**»
 - les informations sur tous les processus créés (une entrée par processus : PCB).



Le Concept de Processus

Process Control Block

Table des processus



Le Concept de Processus

Process Control Block

- **L'état du processus:** L'état peut être New, Ready, Running, Waiting,...
- **Program counter:** Le compteur indique l'adresse de l'instruction suivante à exécuter pour le processus.
- **Les registres CPU:** Les registres varient en nombre et en type, en fonction de l'architecture informatique.
 - Ils comprennent les accumulateurs, les registres d'index, pointeurs de pile et les registres à usage général
 - ces registres doivent être sauvegardé si une interruption se produit, pour permettre au processus de se poursuivre correctement par la suite

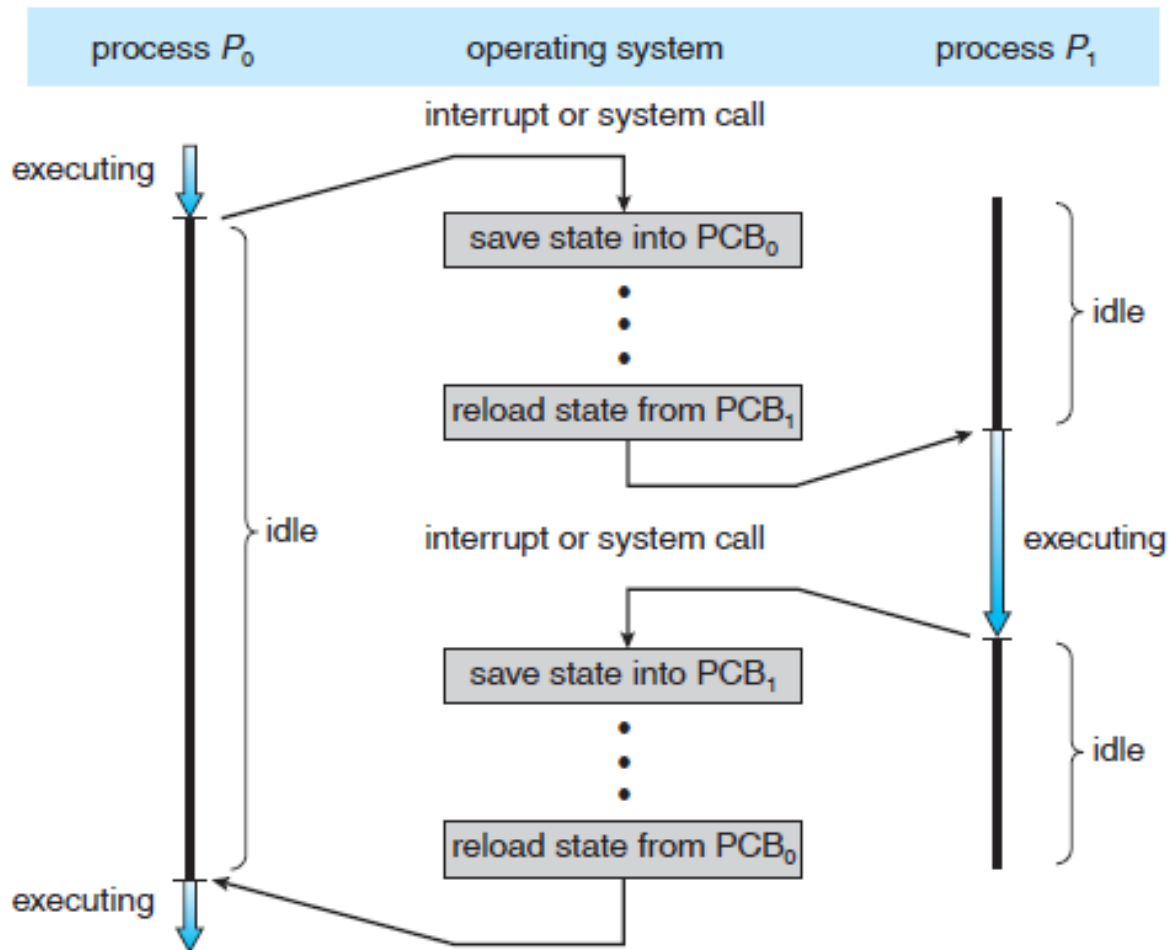
Le Concept de Processus

Process Control Block

- **Les informations relatives à l'ordonnancement du CPU:** Comprennent la priorité du processus, les pointeurs vers des files d'attente d'ordonnancement, et les autres paramètres d'ordonnancement.
- **Les Informations relatives à la gestion de la mémoire:** Ces informations peuvent inclure les tables de page, les tables de segments...
- **Les informations d'état des E/S:** Comprend la liste des périphériques E/S allouée au processus, la liste de fichiers ouverts, ...

Le Concept de Processus

Process Control Block



Le Concept de Processus

Process Control Block

- **Dans le système d'exploitation Linux, le PCB est représenté par la structure C `task_struct`.**
- Cette structure contient toutes les informations nécessaires l'information de représentation d'un processus:
 - L'état du processus
 - son ordonnancement
 - la gestion de mémoire
 - la liste des fichiers ouverts
 - et pointeurs vers le processus parent et l'un de ses enfants.
 - Parent d'un processus est le processus qui l'a créé;
 - ses enfants sont tous les processus qu'il crée

Le Concept de Processus

Process Control Block

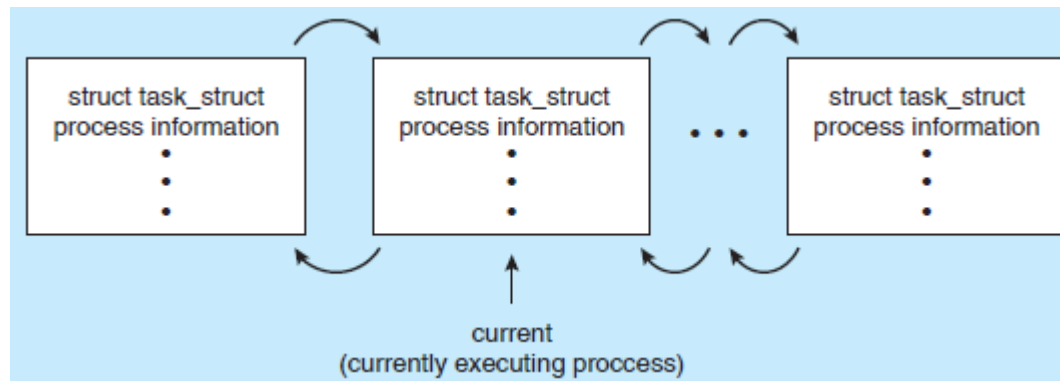
- Quelques champ de la task_struct (/linux/sched.h)
 - (<http://lxr.linux.no/#linux+v3.5.4/include/linux/sched.h#L1229>)

```
pid_t pid; /* process identifier */
long state; /* state of the process */
unsigned int time slice /* scheduling information */
struct task_struct *parent; /*this process's parent */
struct list_head children; /*this process's children */
struct files_struct *files; /*list of open files */
struct mm_struct *mm; /*address space of this process*/
```


Le Concept de Processus

Process Control Block

- L'état d'un processus est représenté par *long state* dans cette structure.
- Dans le noyau Linux, tous les processus actifs sont représentés l'aide d'une liste doublement chaînée des ***task_struct***
- Le noyau maintient un pointeur ***current*** du processus en cours d'exécution sur le système.



Le Concept de Processus

Process Control Block

- Supposons que le système voudrais modifier l'état du processus actuellement en cours d'exécution à la nouvelle valeur ***new_state***.
- ***Current*** pointant vers le processus en cours d'exécution, l'état est changé comme suit:

```
current->state = new_state;
```

PROCESS SCHEDULING

Process Scheduling

- Dans un système multi-utilisateurs à temps partagé, **plusieurs processus** peuvent être présents en mémoire centrale **en attente d'exécution**.
- Si **plusieurs processus sont prêts**, le système d'exploitation **doit gérer l'allocation du processeur aux différents processus à exécuter**.
- **C'est l'ordonnanceur qui s'acquitte de cette tâche.**
- **Un ordonnanceur fait face à deux problèmes :**
 - le choix du processus à exécuter, et
 - le temps d'allocation du processeur au processus choisi.

Process Scheduling

S.E préemptif

- Un **système d'exploitation multitâche** est **préemptif**
 - lorsque celui-ci **peut arrêter** (réquisition) à tout moment **n'importe quelle application** pour passer la main à la suivante.
- Dans les systèmes d'exploitation préemptifs **on peut lancer plusieurs applications à la fois et passer de l'une à l'autre**, voire lancer une application pendant qu'une autre effectue un travail.
- **le noyau garde toujours le contrôle**
 - se réserve le droit de fermer les applications qui monopolisent les ressources du système. Ainsi les blocages du système sont inexistants

Process Scheduling

S.E coopératif

- Un **système d'exploitation multitâche est coopératif**
 - Lorsqu'il permet à **plusieurs applications de fonctionner**
 - et **d'occuper des plages mémoire**, laissant le soin à ces **applications de gérer cette occupation**,
 - au **risque de bloquer tout le système**.

Process Scheduling

Scheduling Queues

- Dès que les processus sont pris en charge par le S.E, ils sont placés dans une file d'attente (***job_queue***), qui contient tous les processus dans le S.E.
- Les processus qui sont stockées dans la mémoire principale et **sont prêtes et en attente d'exécution** sont conservés dans une liste appelée ***ready_queue***
 - Cette file d'attente est généralement implémenté sous la forme **d'une liste chaînée**.
 - La ***ready_queue*** contient un en-tête ayants deux pointeurs: le premier vers le **premier** PCB, un deuxième vers le **dernier** PCB de la liste
 - Chaque PCB comporte un pointeur qui pointe vers le **prochain** PCB de la ***ready_queue***

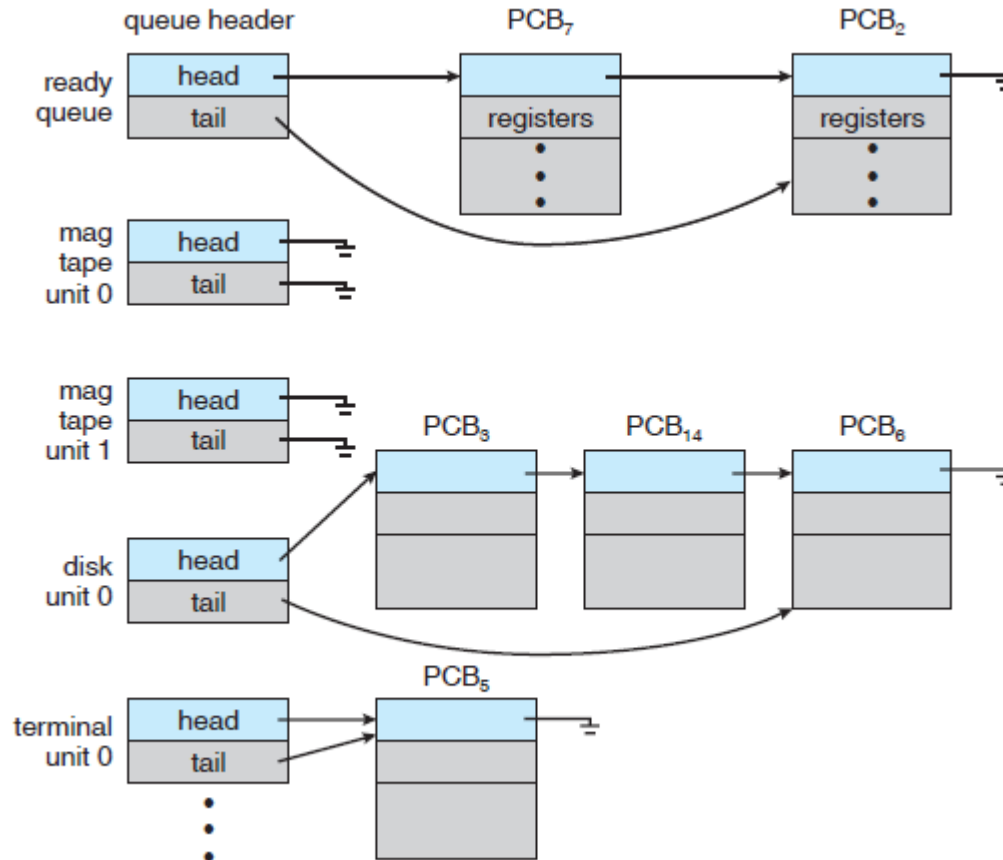
Process Scheduling

Scheduling Queues

- Le système comprend également **d'autres files d'attente**.
 - Quand **un processus se voit allouer le CPU**, il exécute pendant un certain temps et quitte, s'interrompt ou attend l'occurrence d'un événement particulier, **tel que l'achèvement d'une demande d'E/S**.
- Supposons que le processus effectue une demande d'E/S à un **device** partagé, tel qu'un disque.
 - Comme il existe de nombreux processus dans le système, **le disque peut être occupé** avec la demande d'E/S d'un autre processus.
 - Le **processus** peut donc **avoir à attendre le disque**.
 - La liste des processus en attente d'un **device** d'E/S est appelé **device queue**.
 - Chaque device a sa propre **device queue**.

Process Scheduling

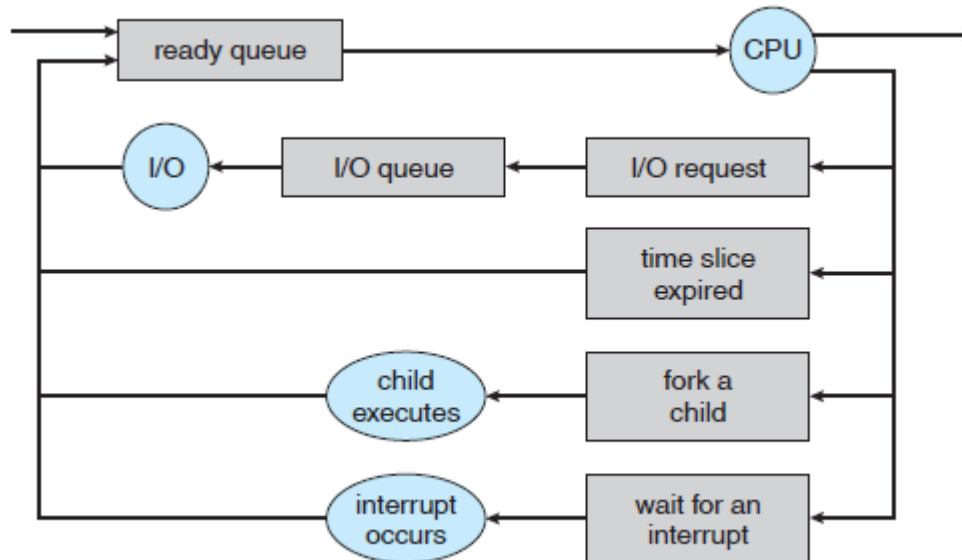
Scheduling Queues



Process Scheduling

Scheduling Queues

- Une représentation commune de l'ordonnancement des processus est un diagramme de files d'attente



- Chaque rectangle représente une file d'attente.
 - deux types des files d'attente sont présentes: la **ready_queue** et un ensemble de **device_queue**.
 - les cercles représentent les ressources qui servent les files d'attente, et les flèches indiquent les flux de processus dans le système.

Process Scheduling

Scheduling Queues

- Un nouveau processus est d'abord mis dans la **ready_queue**. Il attend dans cette file jusqu'à ce qu'il soit sélectionné pour l'exécution, ou est dispatché.
- Une fois que le processus se voit alloué le CPU, et est exécuté, un de ces événements pourrait se produire:
 - Le processus pourrait **émettre une demande d'E/S** et ensuite être placé dans **une file d'attente d'E/S**.
 - Le processus pourrait **créer un nouveau sous-processus** et attendre que la fin du sous-processus
 - Le processus peut subir **une interruption**, enlevé de force du CPU, et mis en attente dans une **device_queue**

Process Scheduling Schedulers

- Un processus migre entre **différentes files d'attente d'ordonnancement** tout au long de sa **durée de vie**.
 - Le S.E doit sélectionner, à des fins de planification, les processus de ces files d'attente.
- **Le processus de sélection est effectué par l'ordonnanceur (scheduler).**

Process Scheduling Schedulers

- **L'ordonnanceur à long terme fait la sélection de programmes à admettre dans le système pour leur exécution.**
- Les programmes admis deviennent des processus à l'état **prêt**.
- **L'admission dépend de la capacité** du système (degré de multiprogrammation) et du niveau de performance requis.

Process Scheduling Schedulers

- **Généralement le nombre de processus soumis est supérieur à la capacité de traitement immédiat du système.**
 - Ces processus sont en attente sur un dispositif de stockage de masse (généralement un disque), où ils sont conservés pour une exécution ultérieure.
- L'ordonnanceur à **long terme**, ou ***job scheduler***, sélectionne les processus de ce pool et **les charge en mémoire pour une exécution ultérieure.**
- L'ordonnanceur à **court terme**, ou ***CPU scheduler***, sélectionne **parmi les processus qui sont prêts à être exécuter et alloue le CPU à l'un d'eux.**

Process Scheduling Schedulers

- **L'ordonnanceur à court terme a pour tâche la gestion des processus prêts.**
 - Il sélectionne en fonction d'une certaine politique le **prochain processus à être exécuter.**
 - Il effectue aussi **le changement de contexte des processus.**
 - Il peut implanter **un ordonnancement préemptif, non préemptif, ou coopératif.**
- **L'ordonnanceur est activé par un événement :**
 - interruption du temporisateur,
 - interruption d'un périphérique,
 - appel système ou signal.

Process Scheduling Schedulers

- **La principale distinction entre ces deux ordonnanceurs réside dans la fréquence d'exécution.**
 - L'ordonnanceur à court terme doit **fréquemment** sélectionner un nouveau processus pour la CPU.
 - Un processus peut s'exécuter pour seulement quelques millisecondes avant une demande d'E/S.
- **Souvent, l'ordonnanceur à court terme s'exécute au moins une fois tous les 100 millisecondes.**
 - En raison de la courte période entre les exécutions, l'ordonnanceur à court terme doit être rapide.
 - S'il faut 10 millisecondes pour décider d'exécuter un processus pour 100 millisecondes, $10 / (100 + 10) = 9 \%$ **de la CPU est utilisé (perdu) simplement pour planifier le travail.**

Process Scheduling Schedulers

- **L'ordonnanceur à long terme s'exécute beaucoup moins fréquemment**
 - des minutes peuvent séparer la création de deux processus.
- **L'ordonnanceur à long terme contrôle le degré de multiprogrammation (le nombre de processus en mémoire).**
- **Si le degré de multiprogrammation est stable**
 - alors le taux moyen de création de processus doit être égal à la moyenne des taux de départ du processus qui quittent le système.
- **L'ordonnanceur à long terme est invoqué lorsqu'un processus quitte le système.**

Process Scheduling Schedulers

- **Les objectifs d'un ordonnanceur sont, entre autres :**
 - S'assurer que chaque processus en attente d'exécution reçoive sa part de temps processeur.
 - Minimiser le temps de réponse.
 - Utiliser le processeur à 100%.
 - Utilisation équilibrée des ressources.
 - Prendre en compte des priorités.
 - Être prédictibles.

Process Scheduling Schedulers

- La plupart des processus peuvent être décrits comme des processus limités par les E/S (**I/O-bound process**) ou des processus limités par le CPU (**CPU-bound process**).
- **Un processus est CPU-bound**
 - Génère rarement des E/S
 - Passe plus de temps à faire des calculs sur le CPU
 - **sa progression est limitée par le CPU**: si la CPU était plus rapide, il aurait terminé son exécution plus rapidement.
- **Un processus est I/O-bound**
 - Génère un nombre important d'E/S (fichiers/accès disque/..)
 - Sa progression est limitée par les E/S

Process Scheduling Schedulers

- Il est important que l'**ordonnanceur** à long terme fasse une sélection équilibrée entre les deux types de processus
- **Si tous les processus sélectionnés sont des I/O-bound**
 - la **ready-queue** serait presque toujours vide
 - L'ordonnanceur à court terme aura très peu à faire
- **Si tous les processus sélectionnés sont des CPU-bound**
 - la **file d'attente d'E/S** sera presque toujours vides,
 - les devices (disques/...) seront sous utilisées
 - Encore une fois le système sera déséquilibrée.
- **Le système avec les meilleure performance devra avoir une combinaison de processus CPU-bound et I/O-bound.**

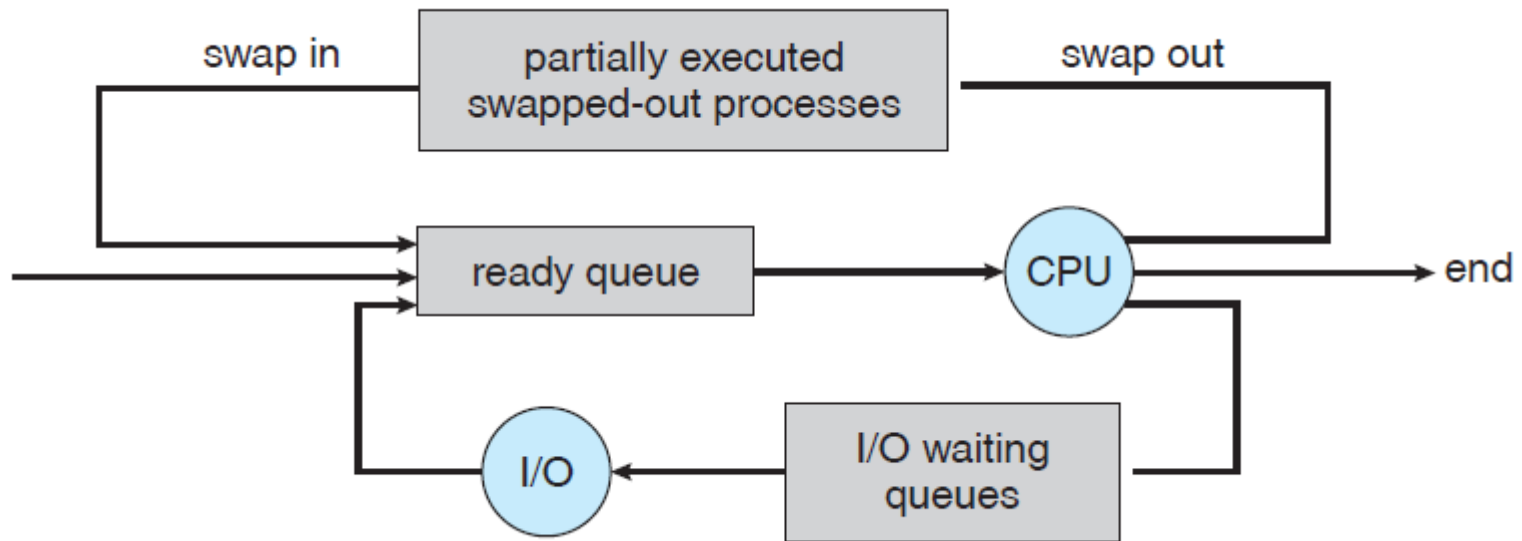
Process Scheduling Schedulers

- **Sur certains systèmes, l'ordonnanceur à long terme peut être absente ou minime.**
 - Par exemple, les systèmes à temps partagé comme **UNIX et Microsoft Windows n'ont souvent pas d'ordonnanceur à long terme,**
 - Ils chargent simplement **chaque nouveau processus dans la mémoire de l'ordonnanceur à court terme.**
 - La stabilité de ces systèmes dépend soit **d'une limite physique** (tel que le nombre de terminaux disponibles) **ou de l'auto-ajustement de l'utilisateur humain** (quittent leurs sessions en raison d'une baisse de performances du système).
- **Ces systèmes introduisent un niveau intermédiaire d'ordonnanceur: les ordonnanceurs à moyen terme**

Process Scheduling Schedulers

- L'idée principale derrière **un ordonnanceur à moyen terme** est que, parfois, **il peut être avantageux d'éliminer les processus de la mémoire** (et de la contention sur le CPU) et donc de **réduire le degré de multiprogrammation**.
- Plus tard, **le processus peut être réintroduit dans la mémoire, et son l'exécution peut être poursuivie où il s'était arrêté**.
- Ce schéma est appelé **swapping**.

Process Scheduling Schedulers



Process Scheduling

Context Switch

- Lorsqu'une interruption se produit, le système doit **sauvegarder le contexte actuel du processus en cours d'exécution** sur le processeur de telle sorte **que il peut restaurer ce contexte lorsque son traitement est effectué**,
 - Le contexte est représenté par le PCB du processus
- De manière générique, le S.E procède à une **sauvegarde d'état (*state save*)** du CPU (que ce soit en user mode ou en kernel mode) puis effectue **une restauration de l'état (*state restore*)** pour reprendre une exécution.

Process Scheduling

Context Switch

- **Switcher la CPU d'un processus à un autre nécessite**
 - l'exécution d'une **sauvegarde d'état du processus en cours**
 - et une **restauration état d'un processus suivant.**
- Cette tâche est connue sous le nom d'un **commutation de contexte.**

Process Scheduling

Context Switch

- **Lorsqu'une commutation de contexte se produit**
 - le noyau enregistre le contexte de l'ancien processus dans son PCB
 - et charge le contexte sauvegardé du nouveau processus pour s'exécuter.
- **Le temps de commutation de contexte est considéré comme un overhead**
 - Le S.E n'exécute aucune tâche durant ce temps.
 - Sa vitesse varie selon les machine, il dépend de la vitesse de la mémoire, le nombre de registres qui doivent être copié, et l'existence d'instructions particulières (comme une seule instruction pour charger ou stocker tous les registres).
 - Les vitesses typiques sont quelques millisecondes.

LES OPERATIONS SUR LES PROCESSUS

Les Operations sur les Processus

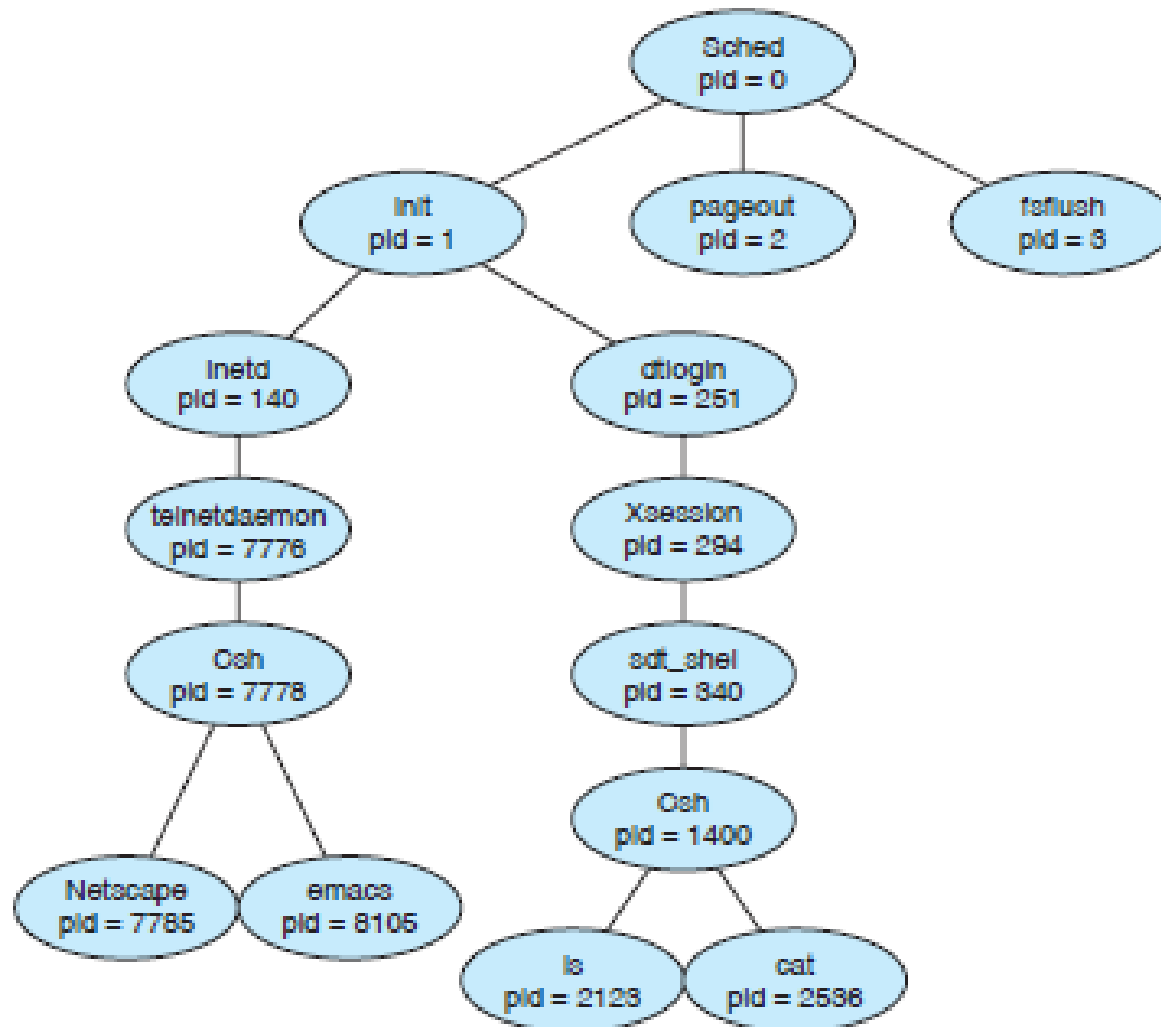
- **Les processus dans la plupart des systèmes peuvent s'exécuter en même temps, et ils peuvent être créés et supprimés de manière dynamique.**
- **Ainsi, ces systèmes doivent fournir une mécanisme de création de processus et de terminaison**

Création de processus

- Un processus (**processus parent**) peut créer plusieurs nouveaux processus (**processus fils**), via un appel système au cours de l'exécution.
- Chacun de ces de nouveaux procédés peuvent à leur tour créer d'autres processus, formant ainsi **un arbre de processus**.
- La plupart des systèmes d'exploitation **identifie un processus** via un ***PID***, qui est typiquement un nombre `integer`.

Création de processus

arbre des processus dans Solaris



Naissance

Création de processus

arbre des processus dans Solaris

- **Sous Solaris, le processus au sommet de l'arbre est le processus `sched`, avec un pid de 0.**
- **Le processus `sched` crée plusieurs enfants, y compris les processus `pageout` et `fsflush`. Ces processus sont responsables de la gestion de la mémoire et des systèmes de fichiers.**
- **Le processus `sched` crée également le processus `init`, qui sert que le processus parent racine pour tous les processus utilisateur.**
- **`Init` a deux enfants: `netd` et `dtlogin`.**
 - `inetd` est chargé des services réseaux tels que telnet et ftp
 - `dtlogin` est le processus qui fournit à l'utilisateur un écran de connexion.
 - Quand un utilisateur se connecte, `dtlogin` crée une session X-Windows (`xsession`), qui à son tour crée le processus `sdt_shel`.

Création de processus

- **En général, un processus a besoin de certaines ressources (temps CPU, la mémoire, les fichiers, périphériques E/S) pour accomplir sa tâche.**
- **Quand un processus crée un sous-processus, le sous-processus peut être en mesure d'obtenir ses ressources directement à partir de l'exploitation système, ou bien il peut être limitée à un sous-ensemble des ressources du processus père.**
 - Le père peut avoir à partitionner ses ressources entre ses enfants, ou il peut être en mesure de partager certaines ressources (comme la mémoire ou les fichiers) entre plusieurs de ses enfants.

Création de processus

- **Quand un processus crée un nouveau processus, il existe deux possibilités d'exécution:**
 - Le parent continue d'exécuter **en même temps** que ses enfants.
 - Le parent **attend** jusqu'à ce que tout ou partie de ses enfants ont pris fin.
- **Il y a aussi deux possibilités pour la gestion de l'espace d'adressage du nouveau processus:**
 - Le processus fils **est une copie** du processus parent (il a le même programme et les données en tant que parent).
 - Le processus fils **a un nouveau programme chargé**.

Création de processus

- Pour illustrer ces différences, nous allons d'abord examiner le système d'exploitation UNIX.
- **Un nouveau processus est créé par `fork()` du système appelant.**
 - **Le nouveau processus comprend une copie de l'espace d'adressage de l'original processus.**
 - Ce mécanisme permet au processus parent de communiquer facilement avec son processus enfant.
 - **Les deux processus (le parent et l'enfant) poursuivre l'exécution à l'instruction après le `fork()`, avec une différence: le code retour du `fork()` est égal à zéro pour le nouveau (enfant) processus, alors que le PID (non nulle) de l'enfant est retourné au processus père.**

Création de processus

- En règle générale, l'appel système **`exec()`** est utilisée après un appel système **`fork()`** par l'un des deux processus afin de remplacer l'espace mémoire du processus avec un nouveau programme.
- L'appel système **`exec()`** charge un fichier binaire en mémoire (destruction de l'image mémoire du programme contenant l'appel système **`exec()`**) et commence son exécution.
- De cette manière, les deux processus sont en mesure de communiquer puis se séparent.
- Le parent peut alors créer plus d'enfants, ou, si il n'a rien d'autre à faire alors que l'enfant s'exécute, il peut émettre un appel système **`wait()`** afin de mettre hors de la **`ready_queue`** jusqu'à la terminaison du processus fils.

Makefile

```
CC=gcc
CFLAGS=-W -Wall -ansi -pedantic
EXEC=exemple1
all: exemple1

exemple1.o: exemple1.c
    $(CC) -o exemple1.o -c exemple1.c $(CFLAGS)

clean:
    rm -rf *.o

mrproper: clean
    rm -rf $(EXEC)
```

Création de processus

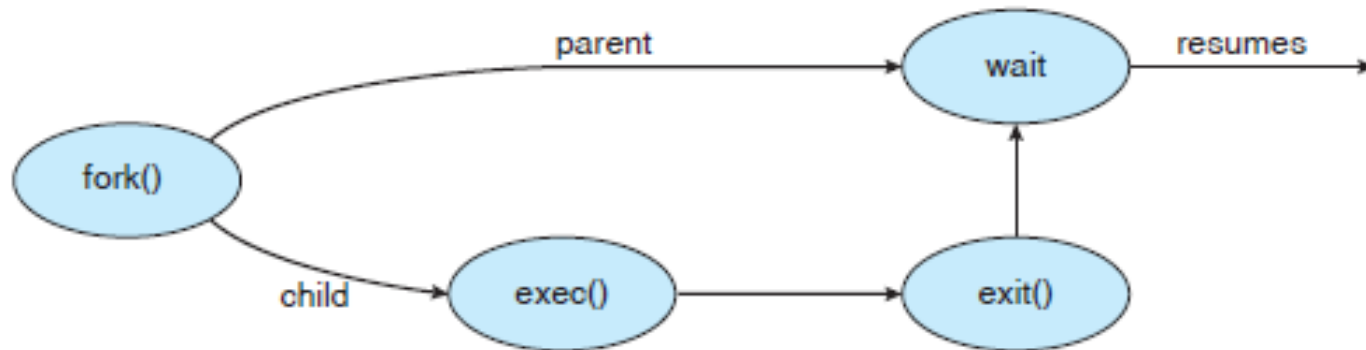
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) {
        /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) {
        /* child process */
        printf("Child Process says Hello !\n");
        execlp("/bin/ls", "ls", NULL);
    }
    else {
        /* parent process */
        /* parent will wait for the child to complete */
        printf("Parent Process says Hello !\n");
        wait(NULL);
        printf("Child Complete");
    }
    return 0;
}
```

Création de processus



Identification par le PID

- Le premier processus du système, `init`, est créé directement par le noyau au démarrage.
- La seule manière, ensuite, de créer un nouveau processus est d'appeler l'appel-système `fork()`, qui va dupliquer le processus appelant.
- Au retour de cet appel-système, deux processus identiques continueront d'exécuter le code à la suite de `fork()`.
 - La différence essentielle entre ces deux processus est un numéro d'identification.
 - On distingue ainsi le processus original, qu'on nomme traditionnellement le processus père, et la nouvelle copie. Le processus fils.
- L'appel-système `fork()` est déclaré dans `<unistd.h>`, ainsi :
`pid_t fork(void) ;`

Identification par le PID

- **Les deux processus pouvant être distingués par leur numéro d'identification PID (Process Identifier), il est possible d'exécuter deux codes différents au retour de l'appel-système `fork()`.**
 - Par exemple. le processus fils peut demander à être remplacé par le code d'un autre programme exécutable se trouvant sur le disque. C'est exactement ce que fait un shell habituellement.
- **Pour connaître son propre identifiant PID, on utilise l'appel-système `getpid()`, qui ne prend pas d'argument et renvoie une valeur de type `pid_t`. Il s'agit, bien entendu, du PID du processus appelant.**
- **Cet appel-système déclaré dans `<unistd.h>`, est l'un des rares qui n'échouent jamais :**

```
pid_t getpid (void) ;
```

Identification par le PID

- Ce numéro de PID est celui que nous avons vu affiché en première colonne de la commande `ps`. La distinction entre processus père et fils peut se faire directement au retour de l'appel `fork()`.
- Celui-ci, en effet, renvoie une valeur de type `pid_t`, qui vaut zéro si on se trouve dans le processus fils, est négative en cas d'erreur, et correspond au PID du fils si on se trouve dans le processus père.

Identification par le PID

- **Voici en effet un point important : dans la plupart des applications courantes, la création d'un processus fils a pour but de faire dialoguer deux parties indépendantes du programme (à l'aide de signaux, de tubes, de mémoire partagée...).**
- **Le processus fils peut aisément accéder au PID de son père (noté PPID pour Parent PID) grâce à l'appel-système `getppid()`, déclaré dans `<unistd.h>` :**

```
pid_t getppid (void);
```
- **Cette routine se comporte comme `getpid()`, mais renvoie le PID du père du processus appelant. Par contre, le processus père ne peut connaître le numéro du nouveau processus créé qu'au moment du retour du `fork()`.**

Identification par le PID

- On peut examiner la hiérarchie des processus en cours sur le système avec le champ PPID de la commande `ps axj`:

```
$ ps axj
PPID  PID  PGID  SID  TTY  TPGID  STAT  UID  TIME  COMMAND
0      1      0      0    ?      -1    S      0   0:03  init
1      2      1      1    ?      -1    SW     0   0:03  (kflushd)
1      3      1      1    ?      -1    SW<    0   0:00  (kswapd)
1      4      1      1    ?      -1    SW     0   0:00  (nfsiod)
[...]
1     296     296     296    6     296    SW     0   0:00  (mingetty)
297    301     301     297    ?      -1    S      0  45:56  usr/X11R6/bin/X
297  25884  25884     297    ?      -1    S      0   0:00  (xdm)
```

Identification par le PID

- **Lorsqu'un processus est créé par `fork()`, il dispose d'une copie des données de son père, mais également de l'environnement de celui-ci et d'un certain nombre d'autres éléments (table des descripteurs de fichiers, etc.). On parle alors d'héritage du père.**
- **Notons que, sous Linux, l'appel-système `fork()` est très économe car il utilise une méthode de « copie sur écriture ».**
 - Cela signifie que toutes les données qui doivent être dupliquées pour chaque processus (descripteurs de fichier, mémoire allouée...) ne seront pas immédiatement recopiées. Tant qu'aucun des deux processus n'a modifié des informations dans ces pages mémoire, il n'y en a qu'un seul exemplaire sur le système.
- **Par contre, dès que l'un des processus réalise une écriture dans la zone concernée, le noyau assure la véritable duplication des données. Une création de processus par `fork()` n'a donc qu'un coût très faible en termes de ressources système.**

Identification par le PID

- En cas d'erreur, `fork()` renvoie la valeur -1, et la variable globale `errno` contient le code d'erreur, défini dans `<errno.h>`, ou plus exactement dans `<asm/errno.h>`, qui est inclus par le précédent fichier d'en-tête.
- Ce code d'erreur peut être
 - soit **ENOMEM**, qui indique que le noyau n'a plus assez de mémoire disponible pour créer un nouveau processus,
 - soit **EAGAIN**, qui signale que le système n'a plus de place libre dans sa table des processus mais qu'il y en aura probablement sous peu. Un processus est donc autorisé à réitérer sa demande de duplication lorsqu'il a obtenu un code d'erreur **EAGAIN**.

Identification par le PID

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/wait.h>
int main (void)
{
    pid_t pid_fils;

    do {
        pid_fils = fork ( ) ;
    }while ((pid_fils == -1) && (errno == EAGAIN));

    if (pid_fils == -1) {
        fprintf(stderr, "fork( ) impossible, errno= %d\n", errno);
        return (1);
    }

    if (pid_fils == 0) {
        fprintf (stdout, "Fils : PID=%d, PPID=%d\n", getpid( ), getppid( ))
        return (0);
    } else {
        fprintf (stdout, "Père :PID=%d,PPID=%d,PID FILS= %d\n",getpid( )
, getppid( ), pid_fils);
        wait(NULL);
        return(0);
    }
    return 0;
}
```

Identification par le PID

- **Dans notre exemple, l'appel-système `fork()` boucle si le noyau n'a plus assez de place dans sa table interne pour créer un nouveau processus.**
- **Dans ce cas, le système est déjà probablement dans une situation assez critique, et il n'est pas utile de gâcher des ressources CPU en effectuant une boucle hystérique sur `fork()`.**
 - Il serait préférable d'introduire un délai d'attente dans notre code pour ne pas réitérer notre demande immédiatement, et attendre ainsi pendant quelques secondes que le système revienne dans un état plus calme

Identification par le PID

- **On remarquera que nous avons introduit un appel-système `wait(NULL)` à la fin du code du père. Nous en reparlerons ultérieurement, mais on peut d'ores et déjà noter que cela permet d'attendre la fin de l'exécution du fils.**
- **Si nous n'avions pas employé cet appel système, le processus père aurait pu se terminer avant son fils, redonnant la main au shell, qui aurait alors affiché son symbole d'invite (\$) avant que le fils n'ait imprimé ses informations.**

Identification par le PID (pas de wait)

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/wait.h>
int main (void)
{
    pid_t pid_fils;
    int i;

    do {
        pid_fils = fork ( ) ;
    }while ((pid_fils == -1) && (errno == EAGAIN));

    if (pid_fils == -1) {
        fprintf(stderr, "fork( ) impossible, errno= %d\n", errno);
        return (1);
    }

    if (pid_fils == 0) {

        for (i=0;i<10;i++)
            fprintf (stdout, "Fils : PID=%d, PPID=%d\n",getpid( ),getppid( ))
        return (0);

    } else {
        fprintf (stdout, "SANS WAIT, Père :PID=%d,PPID=%d,PID FILS=
%d\n",getpid( ) ,getppid( ),pid_fils);
        return(0);
    }
    return 0;
}
```

Identification de l'utilisateur correspondant au processus

- **À l'opposé des systèmes mono-utilisateurs (Dos. Windows 95/98...), un système Unix est particulièrement orienté vers l'identification de ses utilisateurs.**
 - Toute activité entreprise par un utilisateur est soumise à des contrôles stricts quant aux permissions qui lui sont attribuées.
- **Pour cela, chaque processus s'exécute sous une identité précise. Dans la plupart des cas, il s'agit de l'identité de l'utilisateur qui a invoqué le processus et qui est définie par une valeur numérique : l'UID (User Identifier).**

Identification de l'utilisateur correspondant au processus

- Il existe trois identifiants d'utilisateur par processus : l'UID réel, l'UID effectif, et l'UID sauvé.
 - L'UID réel est celui de l'utilisateur ayant lancé le programme.
 - L'UID effectif est celui qui correspond aux privilèges accordés au processus.
 - L'UID sauvé est une copie de l'ancien UID effectif lorsque celui-ci est modifié par le processus.
- Les appels-système `getuid()` et `geteuid()` permettent respectivement d'obtenir l'UID réel et l'UID effectif du processus appelant. Ils sont déclarés dans `<unistd.h>`, ainsi :

```
uid_t getuid (void) ;  
uid_t geteuid(void) ;
```

Identification de l'utilisateur correspondant au processus

- Il s'agit d'une propriété qui est appliquée aux fichiers et répertoires d'un système d'exploitation UNIX. Grâce à cette propriété, un processus exécutant un tel fichier peut s'exécuter au nom d'un autre utilisateur.
- Quand un fichier exécutable est propriété de l'utilisateur root, et est rendu setuid, tout processus exécutant ce fichier peut effectuer ses tâches avec les permissions associées au root, ce qui peut constituer un risque de sécurité pour la machine, s'il existe une faille dans ce programme.
- En effet, un hacker pourrait utiliser cette faille pour s'arroger des droits d'administrateur et effectuer des opérations réservées, par exemple en se créant un compte d'accès illimité en temps et en pouvoirs.

Identification de l'utilisateur correspondant au processus

amine@amine-PC: ~

```
amine@amine-PC:~$ ls -l /usr/bin/passwd
```

```
-rwsr-xr-x 1 root root 47032 juil. 15 20:29 /usr/bin/passwd
```

```
amine@amine-PC:~$
```

Identification de l'utilisateur correspondant au processus

- **Le type `uid_t` correspondant au retour des fonctions `getuid()` et `geteuid()` est défini dans `<sys/types.h>`.**
 - Selon les systèmes, il s'agit d'un unsigned int, unsigned short ou unsigned long.
 - Nous utilisons donc la conversion `%u` pour `fprintf()` qui doit fonctionner dans la plupart des cas

Identification de l'utilisateur correspondant au processus

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

int main (void)
{
    fprintf (stdout, " UID réel = %u, UID
effectif = %u\n", getuid( ), geteuid( ));
    return (0);
}
```


Identification de l'utilisateur correspondant au processus

```
amine@amine-PC: ~  
amine@amine-PC:~$ ls -l exemple4  
-rwxrwxr-x 1 amine amine 8706 oct. 20 09:43 exemple4  
amine@amine-PC:~$ ./exemple4  
UID réel = 1000, UID effectif = 1000  
amine@amine-PC:~$ sudo ./exemple4  
[sudo] password for amine:  
UID réel = 0, UID effectif = 0  
amine@amine-PC:~$ chmod u+s exemple4  
amine@amine-PC:~$ sudo ./exemple4  
UID réel = 0, UID effectif = 1000  
amine@amine-PC:~$ ./exemple4  
UID réel = 1000, UID effectif = 1000  
amine@amine-PC:~$ chmod u-s exemple4  
amine@amine-PC:~$ sudo ./exemple4  
UID réel = 0, UID effectif = 0  
amine@amine-PC:~$
```

Identification groupe d'utilisateurs du processus

- Chaque utilisateur du système appartient à un ou plusieurs groupes.
- Ces derniers sont définis dans le fichier `/etc/groups`.
- Un processus fait donc également partie des groupes de l'utilisateur qui l'a lancé. Comme nous l'avons vu avec les **UID**, un processus dispose donc de plusieurs GID (*Group Identifier*) *réel, effectif, sauvé, ainsi que de* **GID** supplémentaires si l'utilisateur qui a lancé le processus appartient à plusieurs groupes.
- Le GID réel correspond au groupe principal de l'utilisateur ayant lancé le programme (celui qui est mentionné dans `/etc/passwd`).
- Le GID effectif peut être différent du GID réel si le fichier exécutable dispose de l'attribut Set-GID (`chmod g+s`). C'est le GID effectif qui est utilisé par le noyau pour vérifier les autorisations d'accès aux fichiers.

Identification groupe d'utilisateurs du processus

- La lecture de ces GID se fait symétriquement à celle des UID avec les appels-système `getgid()` et `getegid()`.
- La modification (sous réserve d'avoir les autorisations nécessaires) peut se faire à l'aide des appels `setgid()`, `setegid()` et `setregid()`.
- Les fonctions `getgid()` et `setgid()` sont compatibles avec Posix.1, les autres avec BSD. Les prototypes de ces fonctions sont présents dans `<unistd.h>`, le type `gid_t` étant défini dans `<sys/types.h>` :

```
gid_t getgid (void);
gid_t getegid (void);
int setgid (gid_t egid);
int setegid (gid_t egid);
int setregid (gid_t rgid, gid_t egid);
```
- Les deux premières fonctions renvoient le GID demandé. les deux dernières renvoient 0 si elle réussissent et -1 en cas d'échec.

Identification groupe d'utilisateurs du processus

- **L'ensemble complet des groupes auxquels appartient un utilisateur est indiqué dans /etc/groups (en fait, c'est une table inversée puisqu'on y trouve la liste des utilisateurs appartenant à chaque groupe).**
- **Un processus peut obtenir cette liste en utilisant l'appel système**
 - `getgroups()` :
 - `int getgroups (int taille, gid_t liste []);`
 - Celui-ci prend deux arguments, une dimension et une table. Le premier argument indique la taille (en nombre d'entrées) de la table fournie en second argument. L'appel-système va remplir le tableau avec la liste des GID supplémentaires du processus.
 - Si le tableau est trop petit, `getgroups()` échoue (renvoie `—1` et remplit `errno`), sauf si la taille est nulle ; auquel cas, il renvoie le nombre de groupes supplémentaires du processus.

Identification groupe d'utilisateurs du processus

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

int main (void)
{
    int taille;
    gid_t * table_gid = NULL;
    int i;

    if ((taille = getgroups (0, NULL)) < 0) {
        fprintf (stderr, "Erreur getgroups, errno =%d\n", errno);
        return (1);
    }

    if ((table_gid = calloc (taille, sizeof (gid_t)))==NULL) {
        fprintf (stderr, "Erreur calloc, errno = %d\n", errno);
        return (1);
    }

    if (getgroups (taille, table_gid) < 0) {
        fprintf (stderr, "Erreur getgroups, errno %d\n", errno);
        return (1);
    }

    for (i = 0; i < taille; i++)
        fprintf (stdout, "%u " , table_gid [i]);

    fprintf (stdout, "\n");
    free (table_gid);
    return (0);
}
```

Identification du groupe de processus

- Les processus sont organisés en groupes. Rappelons qu'il ne faut pas confondre les groupes de processus avec les groupes d'utilisateurs que nous venons de voir, auxquels appartiennent les processus.
- Les groupes de processus ont pour principale utilité de permettre l'envoi global de signaux à un ensemble de processus. Ceci est notamment utile aux interpréteurs de commandes, qui l'emploient pour implémenter le contrôle des jobs.
- Pour savoir à quel groupe appartient un processus donné, on utilise l'appel-système `getpgid()`, déclaré dans `<unistd.h>`:
 - `pid_t getpgid (pid_t pid);`
- Celui-ci prend en argument le PID du processus visé et renvoie son numéro de groupe, ou `—1` si le processus mentionné n'existe pas. Avec la bibliothèque Glibc 2, `getpgid()` n'est défini dans `<unistd.h>` que si la constante symbolique `GNU_SOURCE` est déclarée avant l'inclusion.

Identification du groupe de processus

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

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

    if (argc == 1) {
        fprintf(stdout, "%d : %d\n", getpid(), getpgid(0));
        return 0;
    }
    for (i = 1; i < argc; i++) {
        if (sscanf(argv[i], "%ld", & pid) != 1) {
            fprintf(stderr, "PID invalide : %s\n", argv[i]);
        } else {
            pgid = (long) getpgid((pid_t) pid);
            if (pgid == -1)
                fprintf(stderr, "%ld inexistant\n", pid);
            else
                fprintf(stderr, "%ld : %ld\n", pid, pgid);
        }
    }
    return 0;
}
```

Mort naturelle et Zombie

Terminaison d'un processus

- **Un processus peut se terminer normalement ou anormalement.**
- **Dans le premier cas, l'application est abandonnée à la demande de l'utilisateur, ou la tâche à accomplir est finie.**
- **Dans le second cas, un dysfonctionnement est découvert, qui est si sérieux qu'il ne permet pas au programme de continuer son travail**

Terminaison normale d'un processus

- Un programme peut se finir de plusieurs manières. La plus simple est de revenir de la fonction `main()` en renvoyant un compte rendu d'exécution sous forme de valeur entière.
- Cette valeur est lue par le processus père, qui peut en tirer les conséquences adéquates. Par convention, un programme qui réussit à effectuer son travail renvoie une valeur nulle, tandis que les cas d'échec sont indiqués par des codes de retour non nuls (et qui peuvent être documentés avec l'application).
- Il est possible d'employer les constantes symboliques `EXIT_SUCCESS` ou `EXIT_FAILURE` définies dans `<stdlib.h>`. Ceci a l'avantage d'adapter automatiquement le comportement du programme, même sur les systèmes non-Posix, où ces constantes ne sont pas nécessairement 0 et 1.

Terminaison normale d'un processus

- Une autre façon de terminer un programme normalement est d'utiliser la fonction `exit()`.
 - `void exit (int code) :`
- On lui transmet en argument le code de retour pour le processus père. L'effet est strictement égal à celui d'un retour depuis la fonction `main()`, à la différence que `exit()` peut être invoquée depuis n'importe quelle partie du programme (notamment depuis les routines de traitement d'erreur).
- Lorsqu'on utilise uniquement une terminaison avec `exit()` dans un programme, le compilateur se plaint que la fin de la fonction `main()` est atteinte alors qu'aucune valeur n'a été renvoyée.

Terminaison normale d'un processus

```
#include <stdlib.h>
void sortie (void);

int main (void)
{
    sortie( );
}

void sortie (void)
{
    exit (EXIT_FAILURE);
}
```

Terminaison normale d'un processus

- **Déclenche à la compilation l'avertissement suivant :**
 - exemple_exit_1.c: In function 'main':
 - exemple_exit_1.c:9: warning: control reaches end of non-void function
- **Si nous avons directement mis `exit()` dans la fonction `main()`, le compilateur l'aurait reconnu et aurait supprimé cet avertissement**

Terminaison normale d'un processus

- Pour éviter ce message, on peut être tenté de déclarer `main()` comme une fonction de type `void`.
- Sous Linux, cela ne pose pas de problème, mais un tel programme pourrait ne pas être portable sur d'autres systèmes qui exigent que `main()` renvoie une valeur:

```
#include <stdlib.h>
void main (void)
{
    exit (EXIT_SUCCESS) ;
}
```

Terminaison normale d'un processus

- **D'ailleurs, le compilateur gcc avertit que `main()` doit normalement être de type `int`:**
 - `exemple_exit_2.c:5: warning: return type of 'main' is not 'int'`

Terminaison normale d'un processus

- **Ayons donc comme règle de bonne conduite, ou plutôt de bonne lisibilité**
 - de toujours déclarer `main()` comme étant de type `int`,
 - et ajoutons systématiquement un `return(0)` ou `return (EXIT_SUCCESS)` à la fin de cette routine.
 - C'est une bonne habitude à prendre, même si nous sortons toujours du programme en invoquant `exit()`.
- **Lorsqu'un processus se termine normalement, en revenant de `main()` ou en invoquant `exit()`, la bibliothèque C effectue les opérations suivantes :**
 - Elle appelle toutes les fonctions qui ont été enregistrées à l'aide des routines `atexit()` et `on_exit()`. Il est possible, grâce aux routines `atexit()` et `on_exit()` de faire enregistrer des fonctions qui seront automatiquement invoquées lorsque le processus se terminera normalement, c'est-à-dire par un retour de la fonction `main()` ou par un appel de la fonction `exit()`
 - Elle ferme tous les flux d'entrée-sortie, en écrivant effectivement toutes les données qui étaient en attente dans les buffers.
 - Elle supprime les fichiers créés par la fonction `tmpfile()`.
 - Elle invoque l'appel-système `exit()` qui terminera le processus

Orphelin et Zombie

- **Processus orphelins :**

si un processus père meurt avant son fils ce dernier devient orphelin.

- **Processus zombies :**

si un fils se termine tout en disposant toujours d'un PID celui-ci devient un processus zombie (ex : sans que son père ait lu son code de sortie (via la wait))

Orphelin et Zombie

- Un processus fils peut devenir orphelin si son père termine avant lui, auquel cas le noyau s'arrange pour le « faire adopter » par un processus système (*INIT*), le processus fils peut donc lui transmettre son statut de terminaison.
- Un processus est dit zombie s'il s'est achevé, mais qui dispose toujours d'un identifiant de processus (PID) et reste donc encore visible dans la table des processus. On parle aussi de processus défunt.
- Au moment de la terminaison d'un processus, le système désalloue les ressources que possède encore celui-ci mais ne détruit pas son bloc de contrôle. Le système passe ensuite l'état du processus à la valeur **TASK_ZOMBIE** (représenté généralement par un Z dans la colonne « statut » lors du listage des processus par la commande ps).
- Le signal **SIGCHLD** est alors envoyé au processus père du processus qui s'est terminé, afin de l'informer de ce changement. Dès que le processus père a obtenu le code de fin du processus achevé au moyen des appels systèmes `wait` ou `waitpid`, le processus terminé est définitivement supprimé de la table des processus.

Terminaison normale d'un processus

- **SIGCHLD:** est un signal utilisé pour réveiller un processus dont un des fils vient de mourir.
- **SIGHUP:** correspond habituellement à la déconnexion (Hang Up) du processus
- **SIGCONT:** permet de relancer un processus stoppé.

Terminaison normale d'un processus

- **L'appel-système `_exit()` exécute – pour ce qui concerne le programmeur applicatif – les tâches suivantes :**
 - Il ferme les descripteurs de fichiers.
 - Les processus fils sont adoptés par le processus 1 (init), qui lira leur code de retour dès qu'ils se finiront pour éviter qu'ils ne restent à l'état zombie de manière prolongée.
 - Le processus père reçoit un signal SIGCHLD.

Terminaison normale d'un processus

- **Le système se livre également à des tâches de libération des ressources verrouillées, de comptabilisation éventuelle des processus, etc.**
- **Le processus devient alors un zombie, c'est-à-dire qu'il attend que son processus père lise son code de retour. Si le processus père ignore explicitement SIGCHLD, le noyau effectue automatiquement cette lecture.**
- **Si le processus père s'est déjà terminé, init adopte temporairement le zombie, juste le temps de lire son code de retour. Une fois cette lecture effectuée, le processus est éliminé de la liste des tâches sur le système.**

Terminaison anormale d'un processus

- **Un programme peut également se terminer de manière anormale.**
- **Ceci est le cas par exemple lorsqu'un processus exécute une instruction illégale, ou qu'il essaye d'accéder au contenu d'un pointeur mal initialisé.**
- **Ces actions déclenchent un signal qui, par défaut, arrête le processus en créant un fichier d'image mémoire core.**

Terminaison anormale d'un processus

- Une manière « propre » d'interrompre anormalement un programme (par exemple lorsqu'un bogue est découvert) est d'invoquer la fonction `abort`).
 - `void abort (void)`
- Celle-ci envoie immédiatement au processus le signal `SIGABRT`, en le débloquent s'il le faut, et en rétablissant le comportement par défaut si le signal est ignoré.
- Le problème de la fonction `abort`() ou des arrêts dus à des signaux est qu'il est difficile de déterminer ensuite à quel endroit du programme le dysfonctionnement a eu lieu.

Terminaison anormale d'un processus

- Il est possible d'autopsier le fichier core (à condition d'avoir inclus les informations de débogage lors de la compilation avec l'option -g de gcc), mais c'est une tâche parfois ardue.
- Une autre manière de détecter automatiquement les bogues est d'utiliser systématiquement la fonction `assert()` dans les parties critiques du programme.
 - Il s'agit d'une macro, définie dans `<assert.h>`. et qui évalue l'expression qu'on lui transmet en argument.
 - Si l'expression est vraie, elle ne fait rien.
 - Par contre, si elle est fausse, `assert()` arrête le programme après avoir écrit un message sur la sortie d'erreur standard, indiquant le fichier source concerné, la ligne de code et le texte de l'assertion ayant échoué. Il est alors très facile de se reporter au point décrit pour rechercher le bogue.
- Pour analyser le core dumped
 - `gcc -g -o myfile myfile.c`
 - `gdb myfile core`

Terminaison anormale d'un processus

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

void fonction_reussissant (int i);
void fonction_echouant (int i);

int main (void)
{
    fonction_reussissant(5);
    fonction_echouant(5);
    return (EXIT_SUCCESS);
}

void fonction_reussissant (int i)
{
    /* Cette fonction nécessite que i soit positif */
    assert (i >= 0);
    fprintf (stdout, "Ok, i est positif \n");
}

void fonction_echouant (int i)
{
    /* Cette fonction nécessite que i soit négatif */
    assert (i <= 0);
    fprintf (stdout, "Ok, i est négatif \n");
}
```

.

Attendre la fin d'un processus fils

- Il est primordial dans les scripts de pouvoir déterminer si une commande a réussi à effectuer son travail correctement ou non.
- On imagine donc l'importance qui peut être portée à la lecture du code de retour d'un processus. Cette importance est telle qu'un processus qui se termine passe automatiquement par un état spécial, zombie, en attendant que le processus père ait lu son code de retour.
- Si le processus père ne lit pas le code de retour de son fils, celui-ci peut rester indéfiniment à l'état zombie.
- Voici un exemple, dans lequel le processus fils attend deux secondes avant de se terminer. Tandis que le processus père affiche régulièrement l'état de son fils en invoquant la commande ps.

Attendre la fin d'un processus fils

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

int main (void)
{
    pid_t pid;
    char  commande[128];

    if ((pid = fork()) < 0) {
        fprintf(stderr, "echec fork()\n");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        /* processus fils */
        sleep(2);
        fprintf(stdout, "Le processus fils %ld se termine\n",
(long) getpid());
        exit(EXIT_SUCCESS);
    }
}
```

Attendre la fin d'un processus fils

```
} else {  
    /* processus pere */  
    snprintf(commande, 128, "ps %ld", (long)pid);  
    system(commande);  
    sleep(1);  
    system(commande);  
    sleep(1);  
    system(commande);  
    sleep(1);  
    system(commande);  
    sleep(1);  
    system(commande);  
    sleep(1);  
    system(commande);  
}  
return EXIT_SUCCESS;  
}
```

Attendre la fin d'un processus fils

```
$ ./exemple_zombie_1
PID    TTY    STAT   TIME   COMMAND
949    pts/0  S      0:00   ./exemple_zombie_1
PID    TTY    STAT   TIME   COMMAND
949    pts/0  S      0:00   ./exemple_zombie_1
Le processus fils 949 se termine
PID    TTY    STAT   TIME   COMMAND
949    pts/0  Z      0:00   [exemple_zombie_<defunct>]
PID    TTY    STAT   TIME   COMMAND
949    pts/0  Z      0:00   [exemple_zombie_<defunct>]
PID    TTY    STAT   TIME   COMMAND
949    pts/0  Z      0:00   [exemple_zombie_<defunct>]
PID    TTY    STAT   TIME   COMMAND
949    pts/0  Z      0:00   [exemple_zombie_<defunct>]
$ ps 949
PID    TTY    STAT   TIME   COMMAND
$
```

Attendre la fin d'un processus fils

- Lorsque le processus père se finit, on invoque manuellement la commande ps, et on s'aperçoit que le fils zombie a disparu.
- Dans ce cas, le processus numéro 1, init, adopte le processus fils orphelin et lit son code de retour, ce qui provoque sa disparition.
- Dans ce second exemple, le processus père va se terminer au bout de 2 secondes, alors que le fils va continuer à afficher régulièrement le PID de son père.

Attendre la fin d'un processus fils

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

int main (void)
{
    pid_t pid;

    if ((pid = fork()) < 0) {
        fprintf(stderr, "echec fork()\n");
        exit(EXIT_FAILURE);
    }

    if (pid != 0) {
        /* processus père */
        fprintf(stdout, "Pere : mon PID est %ld\n", (long)getpid());
        sleep(2);
        fprintf(stdout, "Pere : je me termine\n");
        exit(EXIT_SUCCESS);
    }
}
```

Attendre la fin d'un processus fils

```
} else {  
    /* processus fils */  
    fprintf(stdout, "Fils : mon pere est %ld\n",  
(long)getppid());  
    sleep(1);  
    fprintf(stdout, "Fils : mon pere est %ld\n",  
(long)getppid());  
    sleep(1);  
    fprintf(stdout, "Fils : mon pere est %ld\n",  
(long)getppid());  
    sleep(1);  
    fprintf(stdout, "Fils : mon pere est %ld\n",  
(long)getppid());  
    sleep(1);  
    fprintf(stdout, "Fils : mon pere est %ld\n",  
(long)getppid());  
    }  
    return EXIT_SUCCESS;  
}
```


Attendre la fin d'un processus fils

- L'exécution suivante montre bien que le processus 1 adopte le processus fils dès que le père se termine. Au passage, on remarquera que, aussitôt le processus père terminé, le shell reprend la main et affiche immédiatement son symbole d'accueil (\$) :

```
$ ./exemple_zombie_2
Père : mon PID est 1006
Fils : mon père est 1006
Fils : mon père est 1006
Père : je me termine
$ Fils : mon père est 1
Fils : mon père est 1
Fils : mon père est 1
```

Attendre la fin d'un processus fils

- Pour lire le code de retour d'un processus fils, il existe quatre fonctions : `wait()`, `waitpid()`, `wait3()` et `wait4()`. Les trois premières sont d'ailleurs des fonctions de bibliothèque implémentées en invoquant `wait4()` qui est le seul véritable appel-système.
- La fonction `wait()` est déclarée dans `<sys/wait. h>`, ainsi :
 - `pid_t wait (int * status);`
- Lorsqu'on l'invoque, elle bloque le processus appelant jusqu'à ce qu'un de ses fils se termine. Elle renvoie alors le PID du fils terminé. Si le pointeur `status` est non NULL, il est renseigné ce une valeur informant sur les circonstances de la mort du fils.
 - Si un processus fils était déjà en attente à l'état zombie, `wait()` revient immédiatement.
 - Si on n'est pas intéressé par les circonstances de la fin du processus fils, il est tout à fait possible de fournir un argument NULL.

LES THREADS

Présentation

- **Le concept de processus se traduit par deux caractéristiques:**

1) La propriété de ses ressources :

- Un processus a un espace d'adressage virtuel qui reflète les attributs du PCB (data, texte, PID, PC,...)
- L'OS protège cet espace de toutes interférences extérieur

2) Ordonnancement / exécution :

- L'exécution d'un processus suit un chemin d'exécution (une trace)
- Un processus a un état (Ready, Running,...)
- Une priorité
- Est ordonnancé par l'OS

Présentation

- **Ces deux caractéristiques sont totalement indépendantes**
 - Dans les OS modernes elle agissent sur des entités différentes.
- **La propriété de ses ressources → processus**
- **Ordonnancement / exécution :→ threads**

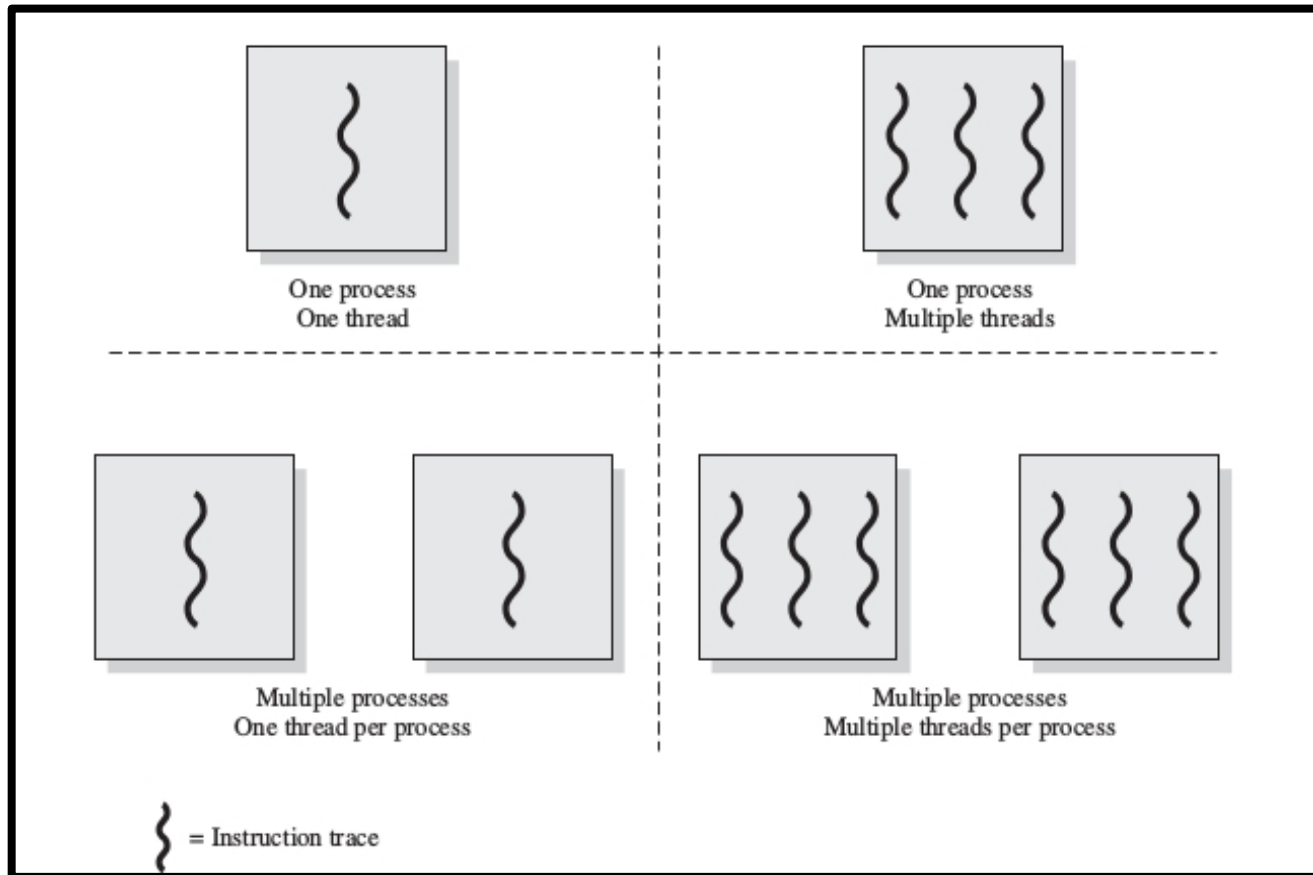
Le Multithreading

Le Multithreading : La capacité d'un système d'exploitation à avoir simultanément des chemins d'exécution multiples au sein d'un processus unique

Le Multithreading

- **Le mot thread peut se traduire par « fil d'exécution », c'est-à-dire un déroulement particulier du code du programme qui se produit parallèlement à d'autres entités en cours de progression.**
- **Les threads sont généralement présentés en premier lieu comme des processus allégés ne réclamant que peu de ressources pour les changements de contexte**
- **Il faut ajouter à ceci un point important : les différents threads d'une application partagent un même espace d'adressage en ce qui concerne leurs données.**
- **La vision du programmeur est d'ailleurs plus orientée sur ce dernier point que sur la simplicité de commutation des tâches.**

Le Multithreading



Monothread

Multithread

Le Multithreading

- **En première analyse, on peut imaginer les threads comme des processus partageant les mêmes données statiques et dynamiques.**
 - **Chaque thread dispose personnellement d'une pile et d'un contexte d'exécution contenant les registres du processeur et un compteur d'instruction.**
 - **Les méthodes de communication entre les threads sont alors naturellement plus simples que les communications entre processus.**
 - **En contrepartie, l'accès concurrentiel aux mêmes données nécessite une synchronisation pour éviter les interférences, ce qui complique certaines portions de code.**

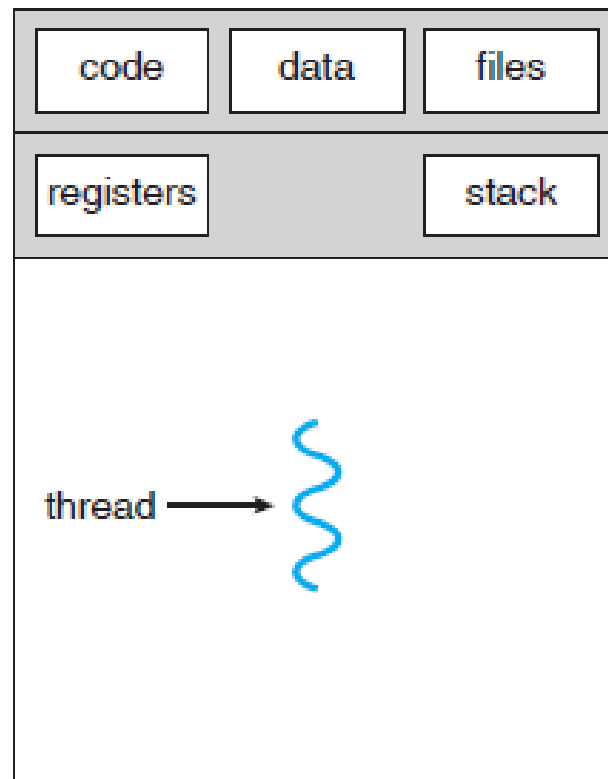
Le Multithreading

- **Les threads ne sont intéressants que dans les applications assurant plusieurs tâches en parallèle.**
- **Si chacune des opérations effectuées par un logiciel doit attendre la fin d'une opération précédente avant de pouvoir démarrer, il est totalement inutile d'essayer une approche multithread.**

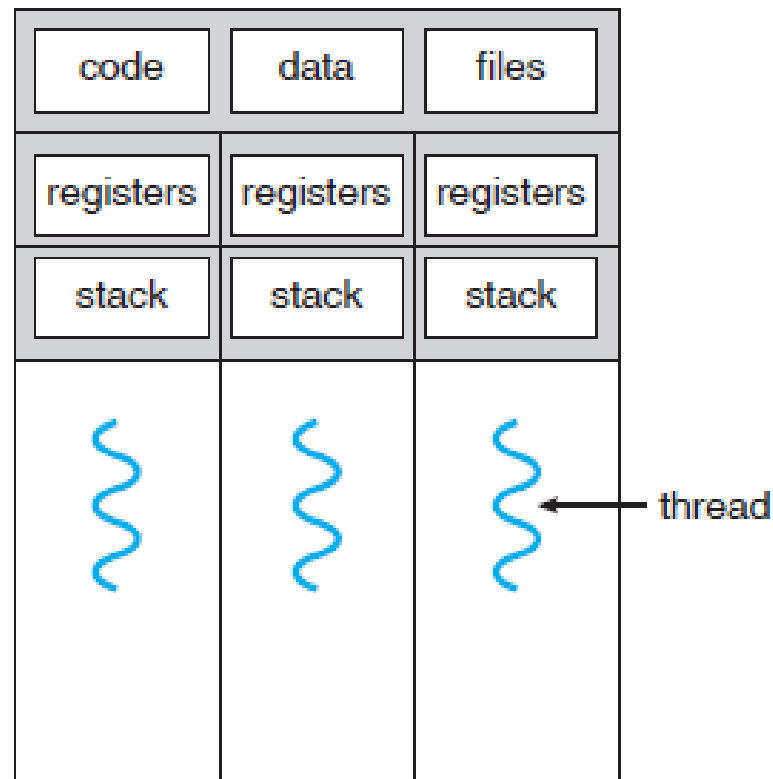
Définition

- **Un thread est l'unité de base de l'utilisation du CPU,**
 - il comporte un identifiant de thread,
 - Un program counter
 - Un ensemble de registres, et une pile.
- **Il partage avec les autres threads appartenant au même processus**
 - la section de code,
 - la section de données,
 - et d'autre ressources du système d'exploitation ressources, telles que les fichiers ouverts et les signaux.
- **Un processus traditionnel (poids lourd) a un seul thread.**
 - Si un processus a plusieurs threads, il peut effectuer plus d'une tâche à la fois.

Définition



single-threaded process



multithreaded process

Motivation

- **De nombreux logiciels qui s'exécutent sur nos PC/MAC sont multi-threadé. Une application est généralement mis en œuvre comme un processus distinct avec plusieurs threads de contrôle.**
- **Un thread du navigateur Web peut avoir la gestion de l'affichage alors qu'un autre thread récupère les données du réseau.**

Avantage

- **Réactivité:** Rendre une application Multithreadé peut permettre à une partie de continuer à fonctionner même si une autre partie est bloqué ou effectue une opération longue
- **Partage des ressources:** Les processus peuvent se partager des ressources grâce à des techniques telles que la mémoire partagée ou la transmission de messages. Ces techniques doivent être explicitement implémenté par le programmeur. Cependant, les threads partagent la mémoire et les ressources du processus auxquels ils appartiennent par défaut.

Avantage

- **Economie:** Alloué de la mémoire et de ressources pour la création des processus est coûteux. Vu que les threads partagent les ressources du processus auxquels elle ils appartiennent, il est plus économique de créer et d'exécuter des commutations de contexte .
- **Scalabilité (passage à l'échelle):** Les avantages du multithreading peuvent être considérablement augmenté dans une architecture multiprocesseur, où les threads peuvent s'exécuter en parallèle sur des processeurs différents

Le parallélisme

- On distingue deux types de parallélisme:
 - ***Task parallelism*** (le parallélisme par flot d'instructions également nommé parallélisme de traitement ou de contrôle) : plusieurs instructions différentes sont exécutées simultanément
 - Machines MIMD (Multiple Instructions, Multiple Data).
 - ***Data parallelism*** (parallélisme de données) : les mêmes opérations sont répétées sur des données différentes
 - Machines SIMD (Single Instruction, Multiple Data)

Le parallélisme

- ***Data parallelism***

- la distribution des sous-ensembles des mêmes données sur plusieurs cœurs de calcul et en effectuant la même opération sur chaque cœur.

- **Exemple : la somme des éléments d'un tableau de taille N**

- Sur une machine single-core : un seul thread effectue la somme de $[0]$ à $[N-1]$
- Sur une machine dual-core : un thread A effectue la somme de $[0]$ à $[N/2 - 1]$ et un second thread B effectue la somme de $[N/2]$ à $[N-1]$.

Le parallélisme

- ***Task parallelism***

- Implique la distribution des tâches (threads) entre les différents cœurs.
- Chaque thread effectuant une tâche particulière
- Différents threads peuvent agir sur les mêmes données ou sur des données différentes.

- **Exemple : statistique sur un tableau**

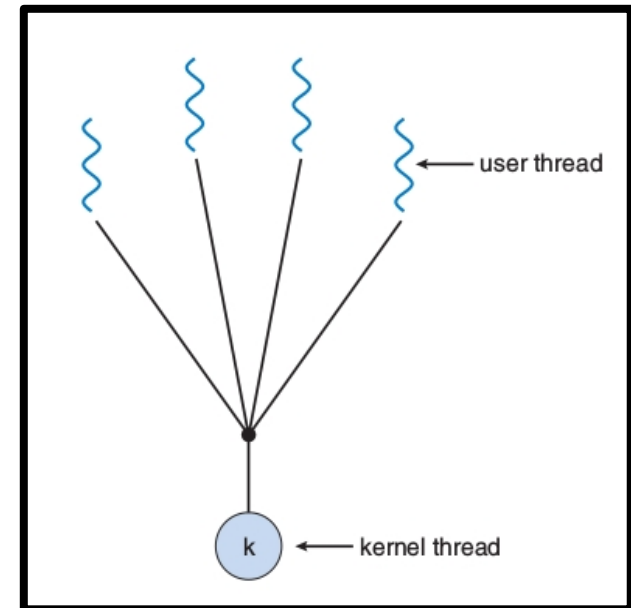
- Thread A → Somme
- Thread B → Min
- Thread C → Max
- Thread D → Moyenne

Modèle de multithreading

- On distingue deux types de threads
 - **User-level threads (ULT)** : la gestion des threads se fait au niveau du user space
 - **Kernel-level threads (KLT)** : la gestion des threads se fait au niveau du noyau de l'OS.
- Plusieurs relations sont possibles entre l'ULT et le KLT : Many-to-One, One-to-One et Many-to-Many

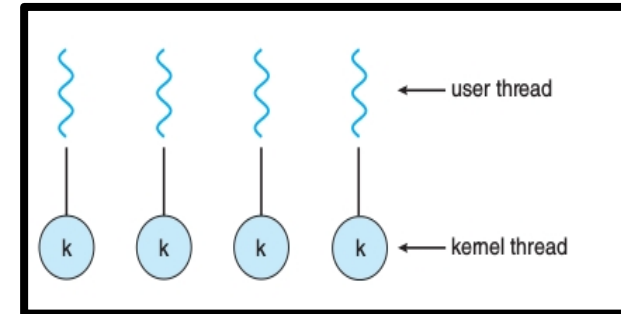
Modèle de multithreading : Many-to-One

- **Fait correspondre plusieurs ULT à un seul KLT**
 - La gestion des threads est fait par une bibliothèque au niveau du user space.
 - Le KLT se bloque si un thread effectue un appel système bloquant
 - Un seul KLT s'exécutant, les ULTs ne peuvent pas s'exécuter en parallèle sur une machine multi-cœur
→ modèle de multithreading abandonné



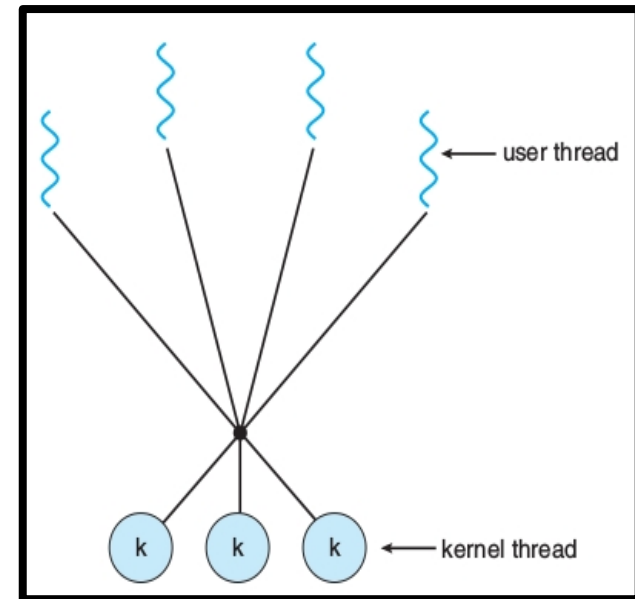
Modèle de multithreading : One-to-One

- Fait correspondre à chaque ULT un seul KLT
- Si un thread effectue un appel système bloquant, les autres threads peuvent encore évoluer
 - Il permet plus de concurrence que le modèle many-to-one
- Il permet à plusieurs threads de s'exécuter en parallèle sur des architectures multiprocesseurs.
- L'inconvénient majeur de ce modèle est qu'il crée un KLT pour chaque ULT → les performances du systèmes peuvent se dégrader rapidement
 - Linux limite les KLT dans ce cas de figures



Modèle de multithreading : Many-to-Many

- **Multiplexe plusieurs ULT en un nombre inférieur ou égale de KLT**
- **Le modèle Many-to-one ne permet pas la concurrence**
- **Le modèle One-to-one permet la concurrence mais le développeur devra faire attention à ne pas créer trop de KLT pour ne pas dégrader les performance du système**
- **Le modèle Many-to-Many dépasse ses deux limites**
 - Les développeurs peuvent créer autant de ULT qu'ils le désirent
 - Les threads peuvent s'exécuter en parallèle sur des machines multiprocesseurs.
 - Un appel système bloquant ne bloque pas les autres threads.



Exemple de Programmation Multithreadé avec Pthread

Création de threads

- Il existe des appels-système qui permettent dans un contexte multithread de créer un nouveau thread, d'attendre la fin de son exécution, et de mettre fin au thread en cours.
- Un type `pthread_t` est utilisé pour distinguer les différents threads d'une application, à la manière des PID qui permettent d'identifier les processus.
- Dans la bibliothèque LinuxThreads, le type `pthread_t` est implémenté sous forme d'un `unsigned long`, mais sur d'autres systèmes il peut s'agir d'un type structuré.

Création de threads

- On se disciplinera donc pour employer systématiquement la fonction `pthread_equal ()` lorsqu'on voudra comparer deux identifiants de threads.
 - `Int pthread_equal (pthread_t thread_1, pthread_t thread_2);`
- Cette fonction renvoie une valeur non nulle s'ils sont égaux.

Création de threads

- **Lors de la création d'un nouveau thread, on emploie la fonction `pthread_create()`.**
 - Celle-ci donne naissance à un nouveau fil d'exécution, qui va démarrer en invoquant la routine dont on passe le nom en argument.
 - Lorsque cette routine se termine, le thread est éliminé.
 - Cette routine fonctionne donc un peu comme la fonction `main()` des programmes C.
 - Pour cette raison, le fil d'exécution original du processus est nommé thread principal (main thread).
- **Le prototype de `pthread_create()` est le suivant :**
 - `int pthread_create (pthread_t * thread, pthread_attr_t * attributs, void * (* fonction) (void * argument), void * argument);`

Création de threads

- **Le premier argument est un pointeur qui sera initialisé par la routine avec l'identifiant du nouveau thread.**
-
- **Le second argument correspond aux attributs dont on désire doter le nouveau thread.**
- **Le troisième argument est un pointeur représentant la fonction principale du nouveau thread.**
 - Celle-ci est invoquée dès la création du thread et reçoit en argument le pointeur passé en dernière position dans `pthread_create()`.
 - Le type de l'argument étant `void *`, on pourra le transformer en n'importe quel type de pointeur pour passer un argument au thread

Création de threads

- Lorsque la fonction principale d'un thread se termine, celui-ci est éliminé.
- Cette fonction doit renvoyer une valeur de type `void *` qui pourra être récupérée dans un autre fil d'exécution.
- Il est aussi possible d'invoquer la fonction `pthread_exit()`, qui met fin au thread appelant tout en renvoyant le pointeur `void *` passé en argument.
- On ne doit naturellement pas invoquer `exit()`, qui mettrait fin à toute l'application et pas uniquement au thread appelant.
 - `void pthread_exit (void * retour);`

Création de threads

- Pour récupérer la valeur de retour d'un thread terminé, on utilise la fonction `pthread_join()`
- Celle-ci suspend l'exécution du thread appelant jusqu'à la terminaison du thread indiqué en argument.
- Elle remplit alors le pointeur passé en seconde position avec la valeur de retour du thread fini.
 - `int pthread_join (pthread_t thread, void ** retour) ;`

Recap'

- **Création de threads**

```
int pthread_create (pthread_t  
*thread , pthread_attr_t *attr,  
void *nomfonction, void *arg );
```

- Le service `pthread_create()` crée un thread qui exécute la fonction `nomfonction` avec l'argument `arg` et les attributs `attr`. Les attributs permettent de spécifier la taille de la pile, la priorité, la politique de planification, etc. Il y a plusieurs formes de modification des attributs.

Recap'

- **Suspension de threads**

```
int pthread_join(pthread_t *thid,  
void **valeur_de_retour);
```

- pthread_join() suspend l'exécution d'un thread jusqu'à ce que termine le thread avec l'identificateur thid. Il retourne l'état de terminaison du thread.

Recap'

- **Terminaison de threads**

```
void pthread_exit(void  
*valeur_de_retour);
```

- pthread_exit() permet à un thread de terminer son exécution, en retournant l'état de terminaison

Recap'

- **Création de threads**

```
int pthread_create (pthread_t  
*thread , pthread_attr_t *attr,  
void *nomfonction, void *arg );
```

- Le service `pthread_create()` crée un thread qui exécute la fonction `nomfonction` avec l'argument `arg` et les attributs `attr`. Les attributs permettent de spécifier la taille de la pile, la priorité, la politique de planification, etc. Il y a plusieurs formes de modification des attributs.

Rappel sur les pointeurs

- le langage C permet de manipuler des adresses par l'intermédiaire de variables nommées "pointeurs".
- **Considérons les instructions :**

```
int * ad ;
```

```
int n ;
```

```
n = 20 ;
```

```
ad = &n ;
```

```
*ad = 30 ;
```

Rappel sur les pointeurs

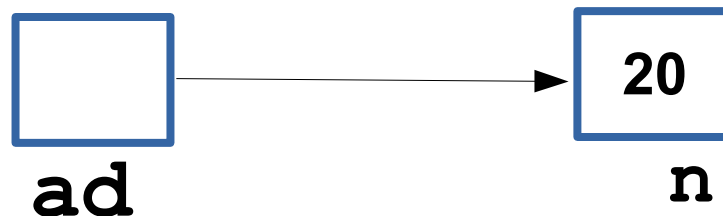
```
int * ad ;
```

- Réserve une variable nommée `ad` comme étant un "pointeur" sur des entiers.
- `*` est un opérateur qui désigne le contenu de l'adresse qui le suit.

Rappel sur les pointeurs

`ad = &n ;`

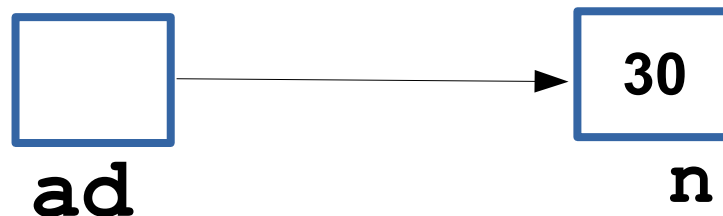
- Affecte à la variable `ad` la valeur de l'expression `&n`. L'opérateur `&` est un opérateur unaire qui fournit comme résultat l'adresse de son opérande.
- cette instruction place dans la variable `ad` l'adresse de la variable `n`



Rappel sur les pointeurs

`*ad = 30 ;`

- signifie : affecter à `*ad` la valeur 30.
- Or `*ad` représente l'entier ayant pour adresse `ad`. Après exécution de cette instruction, la situation est la suivante :



UTILISATION DE POINTEURS SUR DES FONCTIONS

- **En C, comme dans la plupart des autres langages, il n'est pas possible de placer le nom d'une fonction dans une variable.**
- **En revanche, on peut y définir une variable destinée à "pointer sur une fonction", c'est-à-dire à contenir son adresse.**

Paramétrage d'appel de fonctions

- **Considérez cette déclaration :**

```
int (* adf) (double, int) ;
```

- **Elle spécifie que :**

- **(* adf) est une fonction à deux arguments (de type double et int) fournissant un résultat de type int,**

- **donc que :**

- **adf est un pointeur sur une fonction à deux arguments (double et int) fournissant un résultat de type int.**

Paramétrage d'appel de fonctions

- **Si, par exemple, `fct1` et `fct2` sont des fonctions ayant les prototypes suivants :**

```
int fct1 (double, int) ;
```

```
int fct2 (double, int) ;
```

- **les affectations suivantes ont alors un sens :**

```
adf = fct1 ;
```

```
adf = fct2 ;
```

- **Elles placent, dans `adf`, l'adresse de la fonction correspondante (`fct1` ou `fct2`).**

Paramétrage d'appel de fonctions

- **Dans ces conditions, il devient possible de programmer un "appel de fonction variable" (c'est-à-dire que la fonction appelée peut varier au fil de l'exécution du programme) par une instruction telle que :**

```
(* adf) (5.35, 4) ;
```

- **Celle-ci, appelle la fonction dont l'adresse figure actuellement dans adf, en lui transmettant les valeurs indiquées (5.35 et 4). Suivant le cas, cette instruction sera donc équivalente à l'une des deux suivantes :**

```
fct1 (5.35, 4) ;
```

```
fct2 (5.35, 4) ;
```

Fonctions transmises en argument

- **Supposez que nous souhaitions écrire une fonction permettant de calculer l'intégrale d'une fonction quelconque suivant une méthode numérique donnée. Une telle fonction que nous supposerons nommée `integ` posséderait alors un en-tête de ce genre :**

```
float integ ( float(*f)(float), ..... )
```

- **Le premier argument muet correspond ici à l'adresse de la fonction dont on cherche à calculer l'intégrale. Sa déclaration peut s'interpréter ainsi :**
 - `(*f)(float)` est de type `float`
 - `(*f)` est donc une fonction recevant un argument de type `float` et fournissant un résultat de type `float`,

Fonctions transmises en argument

- **f est donc un pointeur sur une fonction recevant un argument de type float et fournissant un résultat de type float**
- **Au sein de la définition de la fonction `integ`, il sera possible d'appeler la fonction dont on aura ainsi reçu l'adresse de la façon suivante :**

`(*f) (x)`

- **Notez bien qu'il ne faut surtout pas écrire `f(x)`, car `f` désigne ici un pointeur contenant l'adresse d'une fonction, et non pas directement l'adresse d'une fonction**

Fonctions transmises en argument

- **L'utilisation de la fonction `integ` ne présente pas de difficultés particulières. Elle pourrait se présenter ainsi :**

```
main()  
{  
    float fct1(float), fct2(float) ;  
    ...  
    res1 = integ (fct1, ..... ) ;  
    ...  
    res2 = integ (fct2, ..... ) ;  
    ...  
}
```

Exemple 1 de pointeur sur fonction

```
ptr-fct.c (~/.Dropbox/Cours/Cour...thread/exemple-introductifs) - VIM
#include <stdio.h>

int SaisisEntier(void)
{
    int n;
    int ret=-1;

    while (ret<0)
    {
        printf("Veuillez entrer un entier : ");
        ret=scanf("%d", &n);
    }
    return n;
}

void AfficheEntier(int n)
{
    printf("L'entier n vaut %d\n", n);
}

int main(void)
{
    void (*foncAff)(int); /* déclaration d'un pointeur foncAff */
    int (*foncSais)(void); /*déclaration d'un pointeur foncSais */
    int entier;

    foncSais = SaisisEntier; /* affectation d'une fonction */
    foncAff = AfficheEntier; /* affectation d'une fonction */

    entier = foncSais(); /* on exécute la fonction */
    foncAff(entier); /* on exécute la fonction */

    return 0;
}
```

1_1 Top

Exemple 2 de pointeur sur fonction

```
ptr-fct-2.c (~/.Dropbox/Cours/Co...thread/exemple-introductifs) - VIM
#include <stdio.h>

int SaisisEntier(void)
{
    int n;
    int ret=-1;

    while (ret<0)
    {
        printf("Veuillez entrer un entier : ");
        ret=scanf("%d", &n);
    }
    getchar();/*vider le buffer du clavier */
    /*fflush(stdin); peut aussi faire l'affaire*/
    return n;
}

void AfficheDecimal(int n)
{
    printf("L'entier n vaut %d \n", n);
}

void AfficheHexa(int n)
{
    printf("L'entier n vaut %x\n", n);
}

void ExecAffiche(void (*foncAff)(int), int n)
{
    foncAff(n); /* exécution du paramètre */
}
```

1,1 Top

Exemple 2 de pointeur sur fonction

```
ptr-fct-2.c (~/.Dropbox/Cours/Co...thread/exemple-introductifs) - VIM
int main(void)
{
    int (*foncSais)(void); /*déclaration d'un pointeur foncSais */
    int entier;
    char rep;

    foncSais = SaisisEntier; /* affectation d'une fonction */
    entier = foncSais(); /* on exécute la fonction */

    printf("Voulez-vous afficher l'entier n en décimal (d) ou en hexa (x) ?\n");
    rep = getchar();
    /* passage de la fonction en paramètre : */
    if (rep == 'd')
        ExecAffiche(AfficheDecimal, entier);
    if (rep == 'x')
        ExecAffiche(AfficheHexa, entier);

    return 0;
}

54,0-1 Bot
```

EXEMPLE1 DE CRÉATION DE THREADS

- Makefile

```
EXEC = exemple-pthread-create-1
```

```
CFLAGS = -O
```

```
LIBS = -lpthread
```

```
CC = gcc
```

```
all: exemple-pthread-create-1
```

```
exemple-pthread-create-1: exemple-pthread-create-1.o
```

```
$(CC) -o exemple-pthread-create-1 exemple-pthread-create-1.o $(LIBS)
```

```
exemple-pthread-create-1.o: exemple-pthread-create-1.c
```

```
$(CC) $(CFLAGS) -c exemple-pthread-create-1.c
```


EXEMPLE1 DE CRÉATION DE THREADS

```
exemple-pthread-create-1.c (~/D...-II/Cours/Chapitre2/threads) - VIM
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void * fonction_thread(void * arg);

int main (void)
{
    pthread_t thr;
    if (pthread_create(& thr, NULL, fonction_thread, NULL) != 0) {
        fprintf(stderr, "Erreur dans pthread_create\n");
        exit(EXIT_FAILURE);
    }
    while (1) {
        fprintf(stderr, "Thread Main\n");
        sleep(1);
    }
}

void * fonction_thread(void * arg)
{
    while (1) {
        fprintf(stderr, "Nouveau Thread\n");
        sleep(1);
    }
}
```

EXEMPLE1 DE CRÉATION DE THREADS

```
amine@amine-PC: ~  
amine@amine-PC:~$ ./exemple-pthread-create-1  
Thread Main  
Nouveau Thread  
Thread Main  
Nouveau Thread  
Thread Main  
Nouveau Thread  
Thread Main  
Nouveau Thread  
Thread Main  
Nouveau Thread  
Thread Main  
Nouveau Thread
```

Exemple PID du thread

```
thread-pid.c (~/.Dropbo.../exemple-creation) - VIM
#include <unistd.h>
#include <pthread.h>
#include <stdio.h>

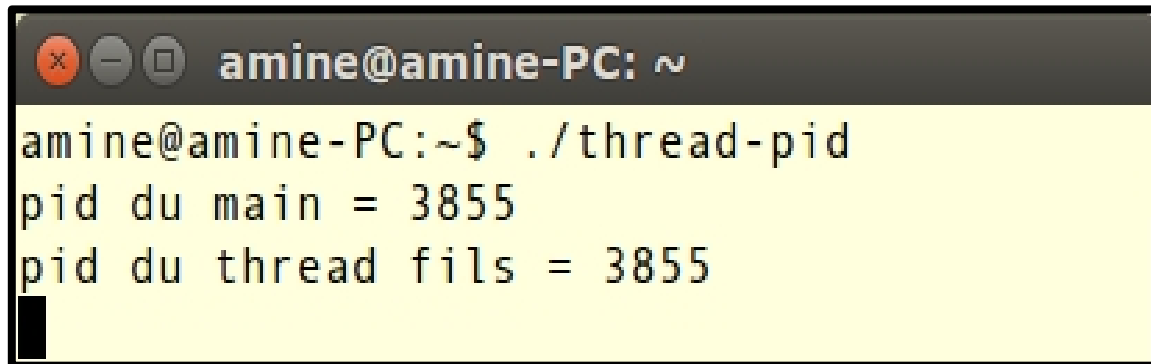
void *fonction(void* arg)
{
    printf("pid du thread fils = %d\n", getpid());
    while(1)

    return NULL;
}

int main()
{
    pthread_t thread;
    printf("pid du main = %d\n", getpid());
    pthread_create(&thread, NULL, &fonction, NULL);
    while(1);
    return 0;
}
```

1,1 All

Exemple PID du thread



```
amine@amine-PC: ~  
amine@amine-PC:~$ ./thread-pid  
pid du main = 3855  
pid du thread fils = 3855
```

Variables globales dans les threads

```
thread-glob.c (~/.Dropbox/Cours...e2-thread/exemple-creation) - VIM
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int glob = 0 ;
void* decrement(void* );
void* increment(void* );

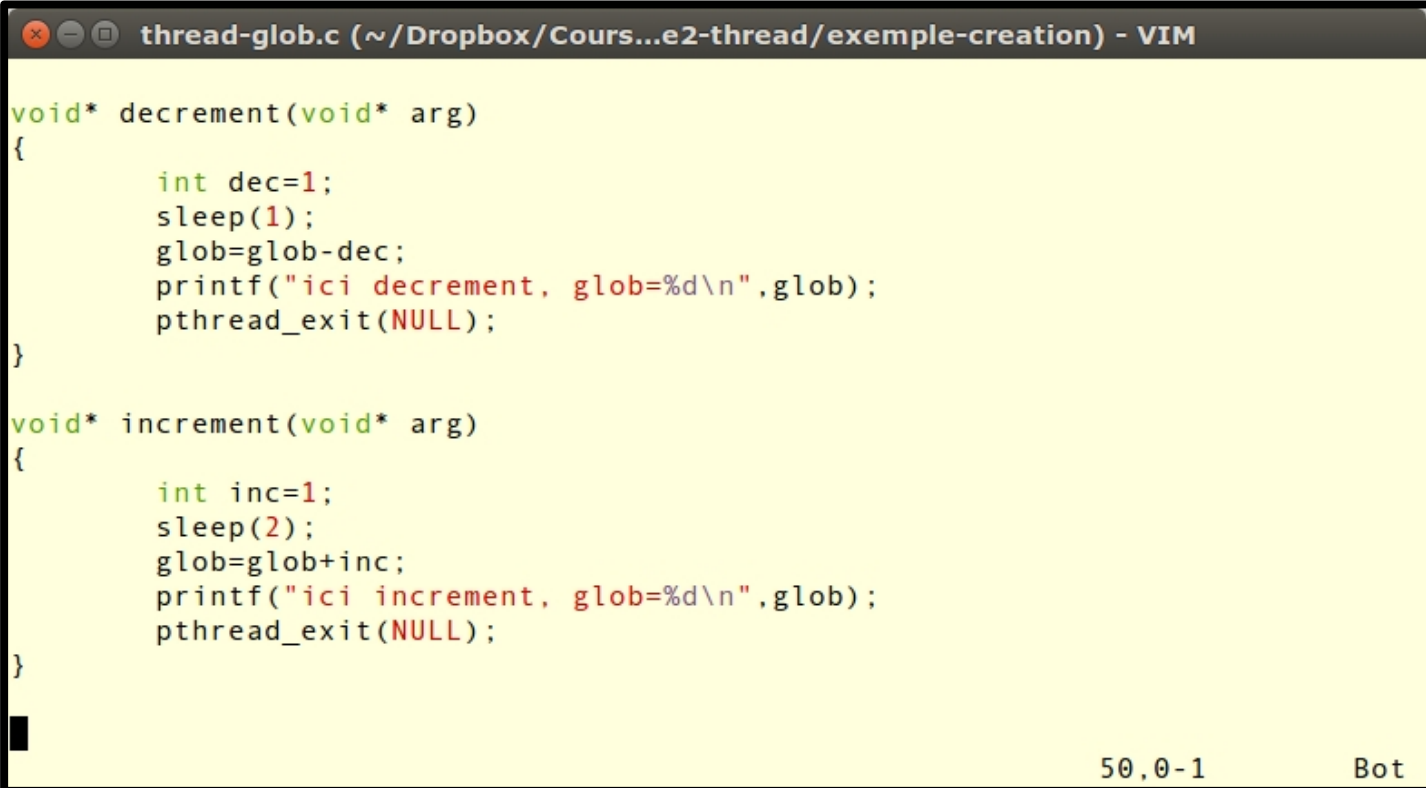
int main()
{
    pthread_t tid1, tid2;
    printf( "ici main[%d] , glob = %d\n ", getpid() , glob ) ;
    if (pthread_create(&tid1, NULL, increment, NULL) != 0) {
        fprintf(stderr, "Erreur dans pthread_create\n");
        exit(EXIT_FAILURE);
    }
    printf( "ici main, création de thread 1 avec succes\n " ) ;

    if (pthread_create(&tid2, NULL, decrement, NULL) != 0) {
        fprintf(stderr, "Erreur dans pthread_create\n");
        exit(EXIT_FAILURE);
    }
    printf( "ici main, création de thread2  avec succes\n " ) ;

    pthread_join(tid1 ,NULL) ;
    pthread_join(tid2 ,NULL) ;
    printf( "ici main, fin des threads, glob=%d\n",  glob ) ;
    return 0 ;
}

1,1 Top
```

Variables globales dans les threads

A screenshot of a VIM editor window. The title bar at the top reads "thread-glob.c (~/.Dropbox/Cours...e2-thread/exemple-creation) - VIM". The editor area has a yellow background and contains C code. The code defines two functions: "decrement" and "increment", both taking a "void*" argument. The "decrement" function sets a local variable "dec" to 1, sleeps for 1 second, decrements the global variable "glob", prints the value, and exits the thread. The "increment" function sets a local variable "inc" to 1, sleeps for 2 seconds, increments the global variable "glob", prints the value, and exits the thread. At the bottom right of the editor, the text "50,0-1" and "Bot" are visible.

```
thread-glob.c (~/.Dropbox/Cours...e2-thread/exemple-creation) - VIM

void* decrement(void* arg)
{
    int dec=1;
    sleep(1);
    glob=glob-dec;
    printf("ici decrement, glob=%d\n",glob);
    pthread_exit(NULL);
}

void* increment(void* arg)
{
    int inc=1;
    sleep(2);
    glob=glob+inc;
    printf("ici increment, glob=%d\n",glob);
    pthread_exit(NULL);
}

50,0-1 Bot
```

Variables globales dans les threads

```
amine@amine-PC: ~  
amine@amine-PC:~$ ./thread-glob  
ici main[6772] , glob = 0  
ici main, création de thread 1 avec succes  
ici main, création de thread 2 avec succes  
ici decrement, glob=-1  
ici increment, glob=0  
ici main, fin des threads, glob=0  
amine@amine-PC:~$
```

Passage de paramètre entre des threads et une fonction main (int)

```
int i = 42;
pthread_create(..., myfunc, (void *)&i);
.
.
.
void *myfunc(void *vptr_value) {
int value = *((int *)vptr_value);
.
.
.
}
```


Passage de paramètre entre des threads et une fonction main (C-string)

```
char *str = "ESEN";
```

```
pthread_create(..., my_func, (void *)str);
```

```
.  
. .
```

```
void *myfunc(void *vp_ptr_value) {
```

```
char *str = (char *)vp_ptr_value;
```

```
.  
. .
```

Passage de paramètre entre des threads et une fonction main (array)

```
Int tab[10];
```

```
pthread_create(..., my_func, (void *)tab);
```

```
.  
.   
.
```

```
void *myfunc(void *vp_ptr_value) {  
int *tableau = (int *)vp_ptr_value;
```

```
.  
.   
.
```

EXEMPLE 2 DE CRÉATION DE THREADS

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

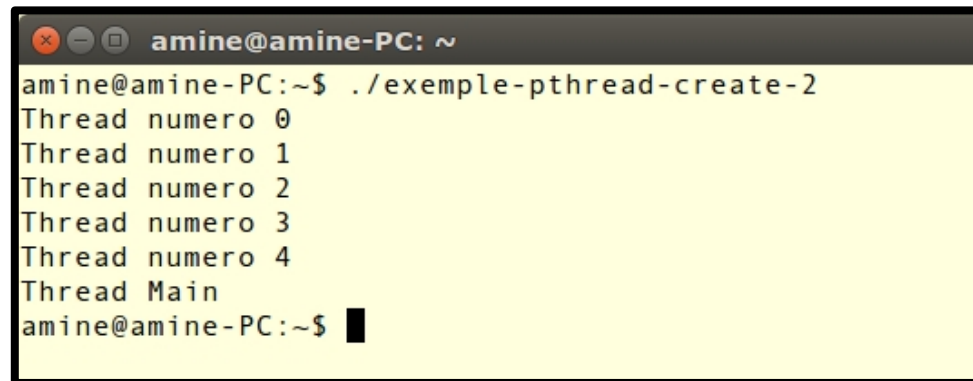
void * fonction_thread(void * arg);

#define NB_THREADS 5

int main (void)
{
    pthread_t thr[NB_THREADS];
    int i;
    for (i = 0; i < NB_THREADS; i ++) {
        if (pthread_create(& thr[i], NULL, fonction_thread, (void *)& i) != 0) {
            fprintf(stderr, "Erreur dans pthread_create\n");
            exit(EXIT_FAILURE);
        }
    }
    sleep(1);
    for (i = 0; i < NB_THREADS; i ++) {
        pthread_join(thr[i], NULL) ;
    }
    fprintf(stderr, "Thread Main\n");
    return 0;
}

void * fonction_thread(void * arg)
{
    int num = *((int *)arg);
    fprintf(stderr, "Thread numero %d\n", num);
    pthread_exit(NULL);
}
```

EXEMPLE 2 DE CRÉATION DE THREADS

A terminal window with a dark title bar containing the text 'amine@amine-PC: ~'. The terminal has a yellow background and displays the following text: 'amine@amine-PC:~\$./exemple-pthread-create-2', followed by five lines of 'Thread numero 0' through 'Thread numero 4', then 'Thread Main', and finally 'amine@amine-PC:~\$' with a black cursor block.

```
amine@amine-PC: ~  
amine@amine-PC:~$ ./exemple-pthread-create-2  
Thread numero 0  
Thread numero 1  
Thread numero 2  
Thread numero 3  
Thread numero 4  
Thread Main  
amine@amine-PC:~$ █
```

EXEMPLE 3 DE CRÉATION DE THREADS

```
exemple-pthread-create-3.c (~/D...-II/Cours/Chapitre2/threads) - VIM
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void * fonction_thread(void * arg);

typedef struct {
    int X;
    int Y;
} coordonnee_t;

int main (void)
{
    pthread_t thr;
    coordonnee_t * coord;

    coord = malloc(sizeof(coordonnee_t));
    if (coord == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }
    coord->X=10;
    coord->Y=20;
    if (pthread_create(& thr, NULL, fonction_thread, coord) != 0) {
        fprintf(stderr, "Erreur dans pthread_create\n");
        exit(EXIT_FAILURE);
    }
    while (1) {
        fprintf(stderr, "Thread Main\n");
        sleep(1);
    }
}
```

EXEMPLE 3 DE CRÉATION DE THREADS

```
exemple-pthread-create-3.c (~/D...-II/Cours/Chapitre2/threads) - VIM
int main (void)
{
    pthread_t thr;
    coordonnee_t * coord;

    coord = malloc(sizeof(coordonnee_t));
    if (coord == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }
    coord->X=10;
    coord->Y=20;
    if (pthread_create(& thr, NULL, fonction_thread, coord) != 0) {
        fprintf(stderr, "Erreur dans pthread_create\n");
        exit(EXIT_FAILURE);
    }
    while (1) {
        fprintf(stderr, "Thread Main\n");
        sleep(1);
    }
}

void * fonction_thread(void * arg)
{
    coordonnee_t * coord = (coordonnee_t *) arg;
    int X = coord->X;
    int Y = coord->Y;
    free(coord);
    while (1) {
        fprintf(stderr, "Thread X=%d, Y=%d\n", X, Y);
        sleep(1);
    }
}
```

47,0-1 Bot

EXEMPLE 3 DE CRÉATION DE THREADS

[illegible]

Passage d'une structure de donnée

```
thread-strt-2.c (~/.Dropbox/Cou...re2-thread/exemple-creation) - VIM
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS 8
char* messages[NUM_THREADS];

typedef struct thread_data {
    int thread_id;
    int sum;
    char *message;
} tdata_t;

void* PrintHello(void* threadarg){
    int taskId,sum;
    char* hello_msg;
    tdata_t *my_data;

    sleep(1);

    my_data=(tdata_t*)threadarg;

    taskId=my_data->thread_id;
    sum=my_data->sum;
    hello_msg=my_data->message;

    printf("Thread %d:  %s Sum=%d \n\n",taskId, hello_msg, sum);
    free(threadarg);
    pthread_exit(NULL);
}
```

1,1 Top

Passage d'une structure de donnée

```
thread-strt-2.c (~/.Dropbox/Cou...re2-thread/exemple-creation) - VIM
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS 8
char* messages[NUM_THREADS];

typedef struct thread_data {
    int thread_id;
    int sum;
    char *message;
} tdata_t;

void* PrintHello(void* threadarg){
    int taskId,sum;
    char* hello_msg;
    tdata_t *my_data;

    sleep(1);

    my_data=(tdata_t*)threadarg;

    taskId=my_data->thread_id;
    sum=my_data->sum;
    hello_msg=my_data->message;

    printf("Thread %d:  %s Sum=%d \n\n",taskId, hello_msg, sum);
    free(threadarg);
    pthread_exit(NULL);
}
```

1,1 Top

Passage d'une structure de donnée

```
amine@amine-PC: ~  
amine@amine-PC:~$ ./thread-struct-2  
  
Thread Main: Creating thread 0  
Thread 0:  Arabic: Essalmou Alikum! Sum=0  
  
Thread Main: Creating thread 1  
Thread 1:  English: Hello World! Sum=1  
  
Thread Main: Creating thread 2  
Thread 2:  French: Bonjour, le monde! Sum=3  
  
Thread Main: Creating thread 3  
Thread 3:  Spanish: Hola al mundo! Sum=6  
  
Thread Main: Creating thread 4  
Thread 4:  German: Guten Tag, Welt! Sum=10  
  
Thread Main: Creating thread 5  
Thread 5:  Russian: Zdravstvyye, mir! Sum=15  
  
Thread Main: Creating thread 6  
Thread 6:  Japan: Sekai e konnichiwa! Sum=21  
  
Thread Main: Creating thread 7  
Thread 7:  Latin: Orbis, te saluto! Sum=28  
  
amine@amine-PC:~$
```

EXEMPLE 4 DE CRÉATION DE THREADS

```
exemple-pthread-create-4.c + (~/.Dro...apitre2-thread/exemple-creation) - VIM
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void * fonction_thread(void * arg);

#define NB_THREADS 5

int compteur = 0;

int main (void)
{
    pthread_t thr[NB_THREADS];
    int i;
    for (i = 0; i < NB_THREADS; i ++) {
        if (pthread_create(& thr[i], NULL, fonction_thread, (void *)&i) != 0) {
            fprintf(stderr, "Erreur dans pthread_create\n");
            exit(EXIT_FAILURE);
        }
    }
    while (1) {
        fprintf(stderr, "Thread Main, compteur = %d\n", compteur);
        sleep(1);
    }
}

void * fonction_thread(void * arg)
{
    int num = *((int*) arg);
    while (1) {
        fprintf(stderr, "Thread numero %d, compteur = %d \n", num+1, compteur);
        compteur ++;
        sleep(1);
    }
}
```

EXEMPLE 4 DE CRÉATION DE THREADS

```
amine@amine-PC: ~  
amine@amine-PC:~$ ./exemple-pthread-create-4  
Thread numero 1, compteur = 0  
Thread numero 5, compteur = 1  
Thread numero 3, compteur = 0  
Thread numero 4, compteur = 0  
Thread Main, compteur = 0  
Thread numero 2, compteur = 0  
Thread numero 1, compteur = 5  
Thread numero 5, compteur = 5  
Thread numero 4, compteur = 5  
Thread numero 3, compteur = 6  
Thread Main, compteur = 8  
Thread numero 2, compteur = 9  
Thread numero 1, compteur = 10  
Thread numero 4, compteur = 10  
Thread numero 2, compteur = 12  
Thread Main, compteur = 11  
Thread numero 3, compteur = 11  
Thread numero 5, compteur = 10  
Thread numero 1, compteur = 15  
Thread numero 2, compteur = 15  
Thread numero 4, compteur = 15  
Thread Main, compteur = 18  
Thread numero 3, compteur = 18
```

EXEMPLE 4 DE CRÉATION DE THREADS

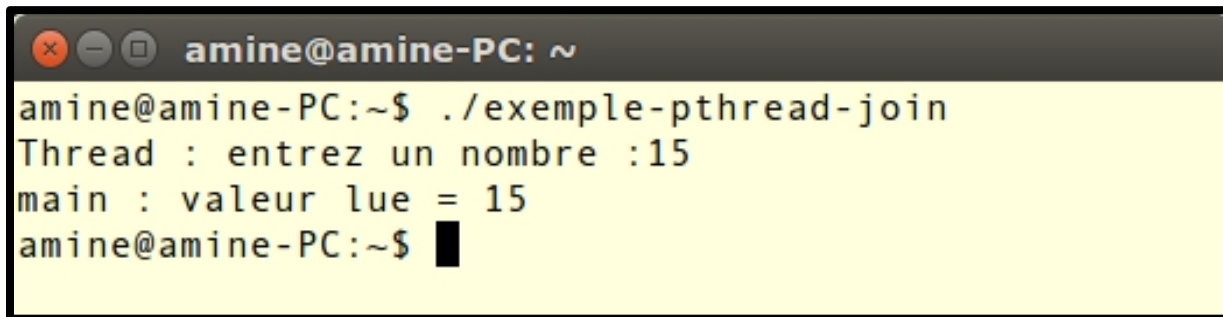
- Cette application est très mal conçue car les différents threads modifient la même variable globale sans se préoccuper les uns des autres.
- Et c'est justement l'essence même de la programmation multithread d'éviter ce genre de situation

Création de threads

- **Le programme suivant n'utilise qu'un seul thread autre que le fil d'exécution principal ; il s'agit simplement de vérifier le comportement des fonctions `pthread_join()` et `pthread_exit()`.**
 - `pthread_join`: attend la fin d'un thread
 - `pthread_exit`: termine le thread appeleant
- **Nous sous-traitons la lecture d'une valeur au clavier dans un fil d'exécution secondaire. Le fil principal pourrait en profiter pour réaliser d'autres opérations.**

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
void * fn_thread (void * inutile)
{
    char chaine [128];
    int i = 0;
    fprintf (stdout, "Thread : entrez un nombre :");
    while (fgets (chaine, 128, stdin) != NULL)
        if (sscanf (chaine, "%d", & i) != 1)
            fprintf (stdout, "un nombre SVP \n");
        else
            break;
    pthread_exit ((void *) i);
}
int main (void)
{
    int i;
    int ret;
    void * retour;
    pthread_t thread;
    if ((ret = pthread_create (& thread, NULL, fn_thread, NULL)) != 0)
    {
        fprintf (stderr, "%s\n", strerror (ret));
        exit (1);
    }
    pthread_join (thread , & retour);
    if (retour != PTHREAD_CANCELED)
    {
        i = (int) retour;
        fprintf (stdout, "main : valeur lue = %d\n", i);
    }
}
```

Création de threads



```
amine@amine-PC: ~  
amine@amine-PC:~$ ./exemple-pthread-join  
Thread : entrez un nombre :15  
main : valeur lue = 15  
amine@amine-PC:~$
```

A terminal window with a dark grey title bar containing the text "amine@amine-PC: ~". The main area has a yellow background and displays the following text: "amine@amine-PC:~\$./exemple-pthread-join", "Thread : entrez un nombre :15", "main : valeur lue = 15", and "amine@amine-PC:~\$ " followed by a black cursor block.

Création de threads

- **Lorsqu'un thread ne renvoie pas de valeur intéressante et qu'il n'a pas besoin d'être attendu par un autre thread, on peut employer la fonction `pthread_detach()`, qui lui permet de disparaître automatiquement du système quand il se termine.**
 - `int pthread_detach (pthread_t thread);`
- **Cela autorise la libération immédiate des ressources privées du thread (pile et variables automatiques). Le mécanisme n'est pas sans rappeler le passage des processus à l'état Zombie en attendant qu'on lise leur code de retour.**

Création de threads

- **Un thread peut très bien invoquer `pthread_detach()` à propos d'un autre thread de l'application.**
 - Contrairement aux processus, il n'y a pas de notion de hiérarchie chez les threads ni d'autorisations particulières pour modifier les paramètres d'un autre fil d'exécution.
 - Cette fonction échoue si le thread n'existe pas ou s'il est déjà détaché.
- **Il n'est pas possible d'attendre, avec `pthread_join()`, la fin d'un thread donné parmi tous ceux qui se déroulent.**
 - Ce genre de fonctionnalité pourrait être utile pour attendre que tous les threads d'un certain ensemble se terminent.
 - On peut obtenir le même résultat en utilisant des threads détachés qui décrémentent un compteur global avant de se terminer, le thread principal surveillant alors l'état de ce compteur.
 - Des précautions devront être prises pour l'accès au compteur global, comme nous le verrons plus tard.

Création de threads

- Pour connaître son propre identifiant, un thread invoque la fonction `pthread_self()`, qui lui renvoie une valeur de type `pthread_t` :
 - `pthread_t pthread_self (void);`
- Il lui est alors possible de comparer avec `pthread_equal()`, son identité avec une variable globale indiquant une tâche à accomplir

EXERCICE

Exercice I: Gestion des processus : — commandes ps, top, awk, kill

- Les paramètres de la commande ps peuvent varier d'un système Unix à un autre
 1. Redirigez la sortie standard de « ps -Al » sur un fichier nommé « process » sans détruire son ancien contenu. Visualisez-le.
 2. Affichez le nombre total de processus (bloqués, actif) et la proportion du temps CPU qu'ils utilisent.
 3. Affichez les noms des processus fils du processus '0' (*ind. Utilisez ps et awk*)
 4. Listez toutes les informations sur les processus dont vous êtes propriétaire (*ps et grep*).
-

Exercice II: Fork(), wait*(), and exit()

1. Combien de processus sont créés par le programme, bidon.c, suivant :

```
main() { fork() ; fork() ; fork() ;}
```

•

Exercice II: Fork(), wait*(), and exit()

- Soit le code suivant : `test.c`

```
main(int argc, char *argv[]) {  
    int c = 5;  
    int child = fork();  
    if (child == 0) c += 5;  
    else {  
        child = fork();  
        c += 10;  
        if(child) c += 5;    }  
}
```

•

Exercice II: Fork(), wait*(), and exit()

1. Combien y-a-t'il de copies de c.
2. Quelles sont leurs valeurs à l'exécution du programme ?
3. On remplace `if(child) c+=5;` par `if (child) execlp("../test..../test.,NULL);` Que se passe-t-il ?

Exercice 3 : Processus Unix (DS 11/2010)

```
main() {  
    int PID, PID1, PID2;  
  
    PID1 = fork();  
  
    PID2 = fork();  
  
    if (PID1 == 0) {  
        PID = getpid();  
        printf("A = %d\n", PID); }  
    else    printf("B = %d\n", PID1);  
  
    if (PID2 == 0) {  
        PID = getpid();  
        printf("C = %d\n", PID);}  
    else    printf("D = %d\n", PID2);  
}
```

Exercice 3 : Processus Unix (DS 11/2010)

1. Représentez, sous la forme d'un arbre, les processus créés par ce programme et leurs relations de parenté.
2. Donnez le résultat de l'exécution de ce programme, pour chaque processus, en utilisant des valeurs de PID que vous inventerez.

Exercice 4.

- **Écrire un programme qui va créer un deuxième processus. Le père va afficher les majuscules à l'écran et le fils les minuscules. Ici, le travail effectué par les 2 processus est trop court et il n'y a pas d'entrelacement des exécutions. Pensez à mettre un "\n" à la fin de chaque écriture afin de vider le buffer !**

Exercice 5.

- **Écrire un programme qui va créer un deuxième processus. Le père et le fils comptent de 0 à 100000 et l'affiche à l'écran. Le père place un P devant son comptage et le fils un F. Analysez le travail de l'ordonnanceur.**

Exercice 6 : Processus Zombie/orphelin ?

- I. Créez une descendance complète de processus avec trois niveaux de branches (grand-père, père et fils).
- II. Que ce passe t-il si on coupe la branche au niveau du grand-père. Tuez le processus correspondant et vérifiez l'état des processus restants. Existe-t-il des processus orphelins ? A quels processus ont-ils été rattachés ?
- III. Stopper maintenant le processus au niveau père. Reprendre les mêmes questions du cas précédent. Pourquoi un processus ne peut-il rester sans processus père ?
- IV. Que ce passe t-il si on coupe la branche au niveau du fils. Tuez le processus correspondant et vérifiez l'état des processus restants. Existe-t-il des processus zombies ?
- V. Stopper maintenant le processus au niveau 2 (père). Vérifiez l'état des processus restants ? Existe-t-il des processus zombies ?

•

Exercice 7.

- Une variante de commandes exec (execl(e/p), execv(e/p)) permettant d'exécuter des processus en remplaçant un processus par le nouveau code d'un autre. Attention : il n'y a pas création d'un nouveau processus mais recouvrement de l'ancien processus par le nouveau.
- Faites exec ps. Que se passe-t-il ? Donnez une explication. Vérifiez votre explication en étudiant la commande shell exec sh. (Quel est le pid de ce shell ?)

Exercice 7

- **Soit le programme C suivant :**

```
#include <unistd.h>
void main() {
    pid_t p1, p2, p3, p4;
    int i;
    if ((p1=fork())==0)
    if ((p2=fork())==0)
        printf("Process p2, mon num est %d \n", getpid());
    else execlp("a", "a", NULL);
    if ((p3=fork())==0) { execlp("b","b", NULL); }
    if ((p4=fork())==0) { execlp("c","c", NULL); }
    sleep(5);
    /* attente de terminaison des fils*/
    while((i=waitpid(-1,NULL,0))>0)
        printf(" \nFils %d termine\n ",i) ;
}
```

-
-

Exercice 7

- **Tracer l'arborescence des processus créés par ce programme si les programmes a, b et c se terminent tous par l'instruction : `exit(2)`;**
- **Pour simplifier, nous supposons que :**
 - les programmes a, b, c durent respectivement 2s, 3s et 1s
 - tous les processus partagent le même processeur
 - un processus qui obtient le processeur le garde jusqu'à ce qu'il se termine sauf dans le cas où il passe à l'état endormi (exécute la fonction `sleep()`).

Exercice 8

- La création et le lancement du thread se fait par :

```
pthread_t th1 ;  
int ret ;
```

```
pthread_create (&th1, NULL, runDuThread, "1");  
if (th1 == NULL) {  
    fprintf (stderr, "pthread_create error 1\n") ; exit(0) ;  
}
```

- Le thread exécutera alors la fonction *runDuThread* dont le prototype est :
 - *void* runDuthread (void *param) ;*

Exercice 8

- Cette fonction est à écrire par le programmeur pour décrire le comportement du thread. Le paramètre *param* est un pointeur dont la valeur est celle passée en argument (le 4ème) de la fonction *pthread_create*. Il permet de passer des données au thread.
- Si la fonction main se termine, le programme et tous les threads lancés se terminent aussi. Il faut donc s'assurer avant de terminer le programme que tous les threads ont fini leur travail. L'attente de la terminaison d'un thread se fait comme ceci :
 - `(void) pthread_join (th1, (void *)&ret) ;`
- Le paramètre *ret* contiendra la valeur retournée par la fonction *pthread_exit (int val)* à exécuter avant de terminer un thread.
- Écrire un programme qui lance 2 threads. L'un écrira les 26 minuscules à l'écran et l'autre les 26 majuscules.
- Écrire un programme qui initialise une variable globale à 0 et crée 2 threads. Chacun des threads va incrémenter la variable N fois. Afficher la valeur de la variable à la fin de l'exécution de chacun des threads.

Exercice 9

- The Fibonacci sequence is the series of numbers 0, 1, 1, 2, 3, 5, 8,
- Formally, it can be expressed as:
 - $f_{ib}0 = 0$
 - $f_{ib}1 = 1$
 - $f_{ib}n = f_{ib}n-1 + f_{ib}n-2$

Exercise 9

- Write a C program using the `fork()` system call that generates the Fibonacci sequence in the child process. The number of the sequence will be provided in the command line.
-
- For example, if 5 is provided, the first five numbers in the Fibonacci sequence will be output by the child process. Because the parent and child processes have their own copies of the data, it will be necessary for the child to output the sequence.
-
- Have the parent invoke the `wait()` call to wait for the child process to complete before exiting the program. Perform necessary error checking to ensure that a non-negative number is passed on the command line.

Exercise 9

- Write a multithreaded program that generates the Fibonacci sequence.
- This program should work as follows: The user will enter on the command line the number of Fibonacci numbers that the program is to generate.
-
- The program will then create a separate thread that will generate the Fibonacci numbers, placing the sequence in data that can be shared by the threads (an array is probably the most convenient data structure).
-
- When the thread finishes execution, the parent thread will output the sequence generated by the child thread. Because the parent thread cannot begin outputting the Fibonacci sequence until the child thread finishes, this will require having the parent thread wait for the child thread to finish

FIN