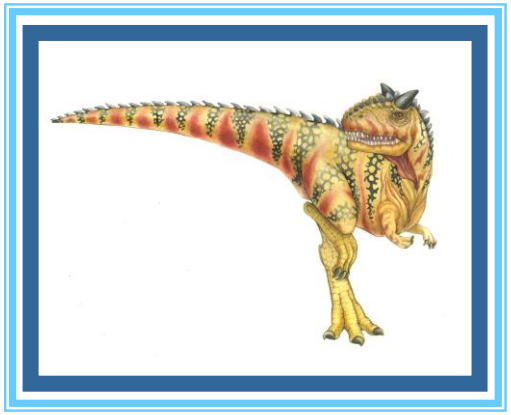
Chapter 2: Processes



Outline

- Introduction
- Les processus
  - Définition
  - Types
  - Caractéristiques
  - Hiérarchie
  - Etats
  - Le bloc de contrôle (PCB)
- L'ordonnancement des processus
  - Présentation
  - Allocation du processeur
  - Algorithmes d'ordonnancement

- Process Concept
- Process Scheduling
- Operations on Processes

- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems
- Communication in Client-Server Systems

Concept de processus

- Un **programme** est une entité **passive** stockée sur le disque (fichier exécutable).

Description statique d'un travail

≠

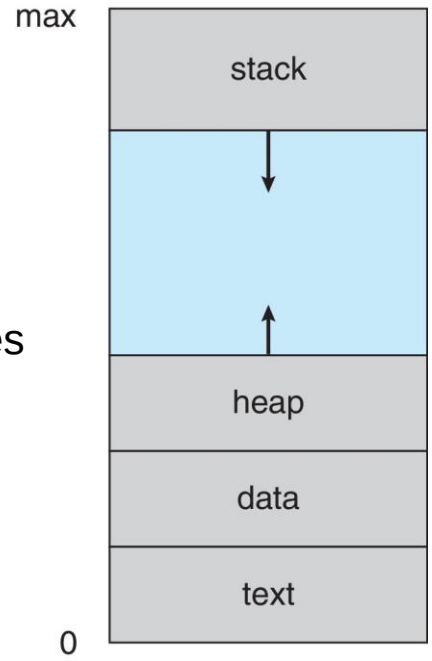
- Un **processus** est une entité **active** en cours d'exécution.

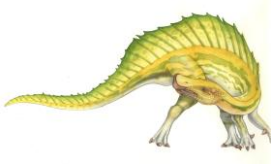
Instance d'une tache en cours d'exécution

- **Transformation d'un programme en processus**
  - Un programme devient un processus lorsqu'il est **chargé en mémoire**.
  - L'exécution peut être **déclenchée** via une interface graphique (**clic de souris**) ou en **ligne de commande**.
- **Un programme peut générer plusieurs processus**
  - Si un même programme est exécuté plusieurs fois, il correspond à plusieurs processus
  - Plusieurs utilisateurs peuvent exécuter le même programme simultanément, créant ainsi plusieurs processus distincts.
- Un **processus** peut **communiquer** des informations **avec d'autres processus.**

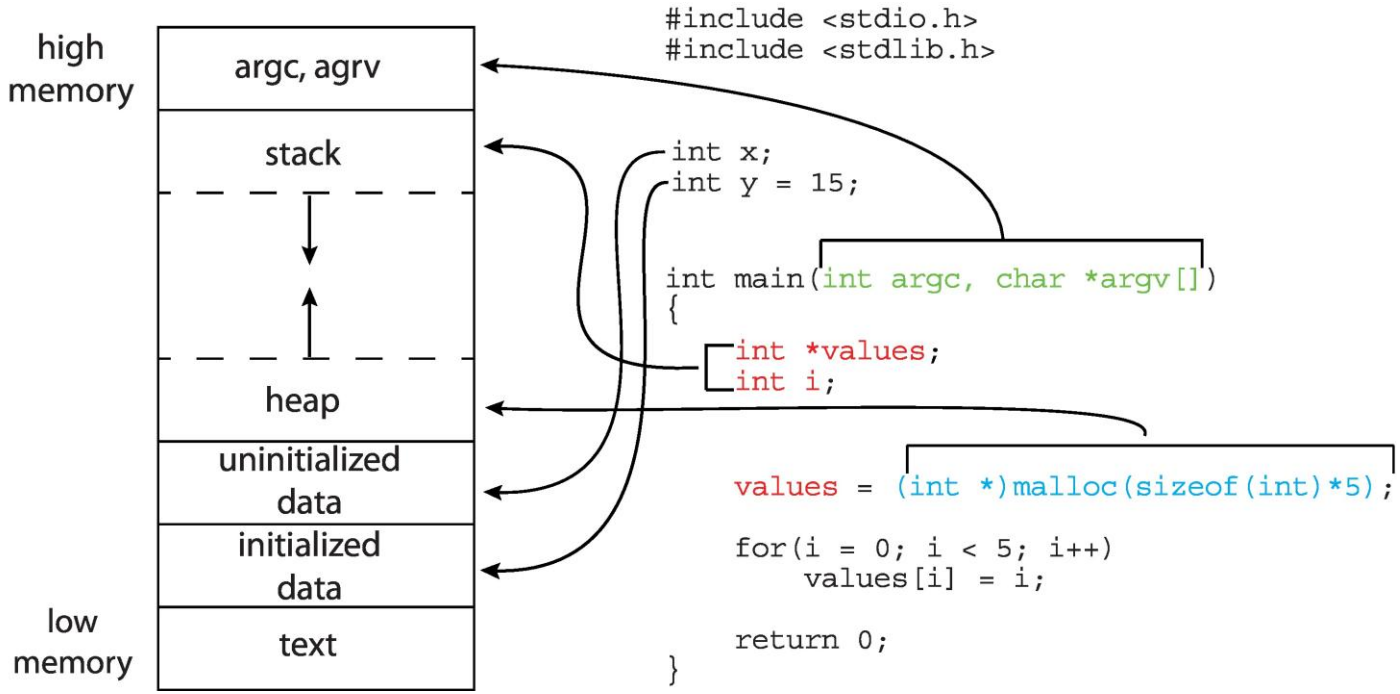
Concept de processus

- **Composants d'un processus**
  - Code du programme (appelé **text section**)
  - **État actuel** (compteur ordinal, registres du processeur)
  - **Pile (Stack)** : mémoire temporaire
    - Contient les paramètres des fonctions, les adresses de retour et les variables locales
  - **Section de données** : contient les variables globales
  - **Tas (Heap):** mémoire allouée dynamiquement à l'exécution





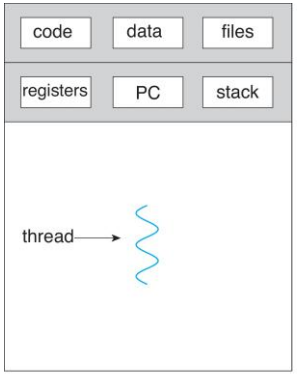
Organisation de la Mémoire d'un Programme C



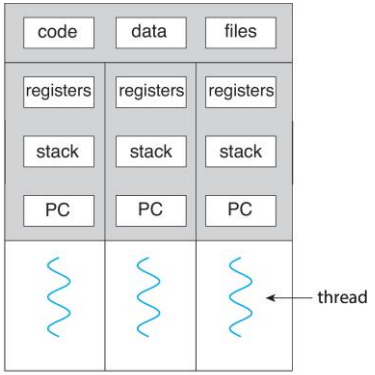
Concept de processus

- Un **processus** peut être **composé d'un ou de plusieurs processus légers** (**threads**).
  - « *Un thread est une unité d'exécution rattachée à un processus, chargée d'en exécuter une partie.* »
  - **Ex**: pour un même document MS-Word, plusieurs threads: **Interaction avec le clavier**, **sauvegarde régulière** du travail, **contrôle d'orthographe**…)

- Un **processus possède** un ensemble de **ressources** (**code**, **fichiers**, **périphériques**…) que ses **threads partagent.**
- Cependant, **chaque thread dispose** :
  - d'un **compteur programme** (pour le suivi des instructions à exécuter)
  - de **registres systèmes** (pour les variables de travail en cours)
  - **d'une pile** (pour l'historique de l'exécution)



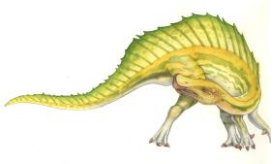
single-threaded process



multithreaded process

Types de processus

- Le système d'exploitation manipule deux types de processus :
  - **Processus système** : processus lancé par le système (**init** processus père des tous les processus du système)
  - **Processus utilisateur** : processus lancé par l'utilisateur (commande ou programme utilisateur)



Création des processus

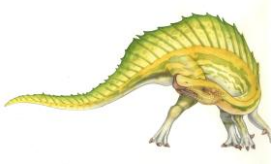
- Avant le lancement du système, le **chargeur de démarrage** (ou **d'amorce**) fait appel à la fonction **start_kernel()** du fichier **init/main.c**.

- Cette fonction va créer le tout premier processus : le **swapper** (ou **Processus 0**, ou encore **sched pour scheduler**) qui occupera la première entrée de la table des processus.

- Le **swapper** va ensuite créer **deux autres processus**, le processus **init** et le processus [**kthreadd**] et va s'endormir.

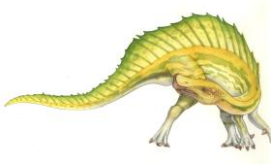
- À partir de ce moment-là, nous pouvons considérer qu'il existe **2 espaces au sein du système**. Un **espace utilisateur** avec **init** au sommet de la hiérarchie et un **espace noyau** avec [**kthreadd**].

  - **Swapper (pid = 0) qui gère la mémoire;**
    - **Init (pid = 1) qui crée tous les autres processus**
    - [**kthreadd**] **(pid = 2) le démon de thread du noyau**



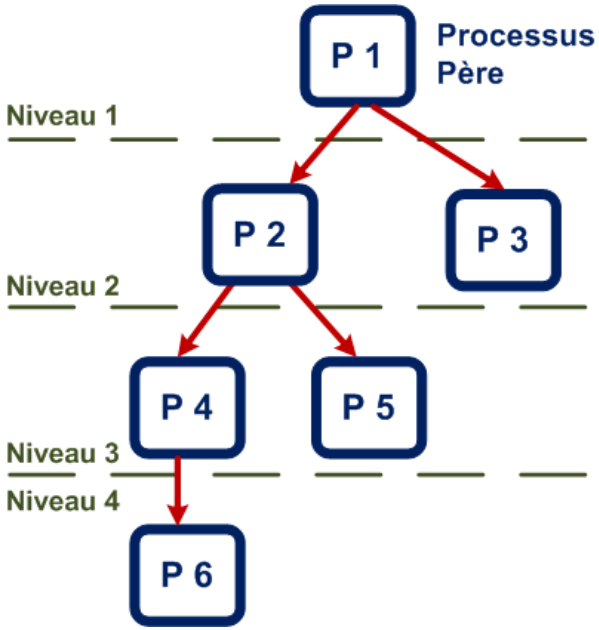
Caractéristiques des processus

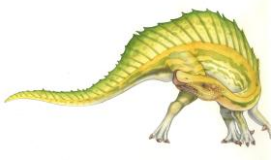
- Depuis sa création, un processus possède notamment les caractéristiques suivantes :
  - **PID** : identificateur du processus (numéro unique)
  - **PPID** : identificateur du processus père
  - **UID** : identificateur de l'utilisateur qui a lancé le processus
  - **GID** : identificateur du groupe de l'utilisateur qui a lancé le processus
  - **etc.**



Hiérarchie des processus

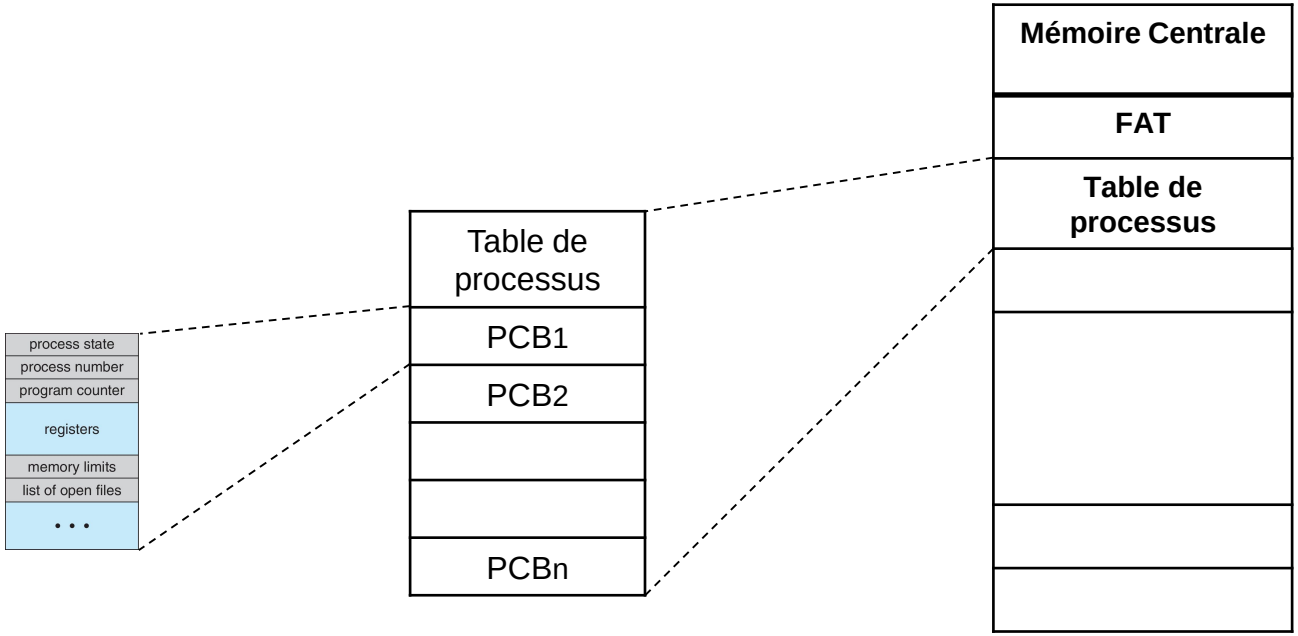
- Un processus **père** peut créer d'autres processus **fils**.
- Ce dernier, à son tour, crée d'autres processus.
- Un processus a un seul père mais peut avoir plusieurs fils.

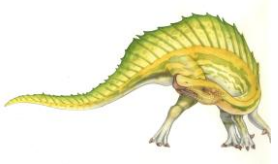




Bloc de Contrôle du Processus (PCB)

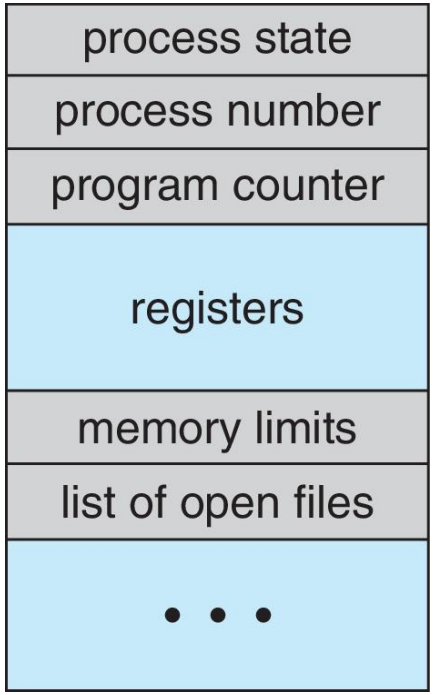
- Pour localiser et gérer tous les processus, le SE maintient **une structure de données** appelée «**table des processus**» qui contient les informations sur tous les processus créés.
- Le Bloc de Contrôle de Processus (**Process control Bloc ou PCB**) est une entrée dans cette table, composée principalement de:





Bloc de Contrôle du Processus (PCB)

- **PID** (Process ID) : Identifiant unique du processus.
- **PPID** L'ID du processus parent
- **UID** L'ID de l'utilisateur du processus
- **État** : En cours d'exécution, en attente, suspendu, etc.
- **Compteur de programme** : Adresse de la prochaine instruction à exécuter.
- **Registres CPU** : Sauvegarde des registres du processeur liés au processus.
- **Priorité et planification** : Informations pour l'ordonnancement du processus.
- **Mémoire** : Adresses mémoire allouées (pile, tas, segments de code et de données).
- **Fichiers ouverts** : Liste des fichiers et périphériques utilisés.
- **Informations sur les ressources** : Temps CPU consommé, statistiques d'utilisation.

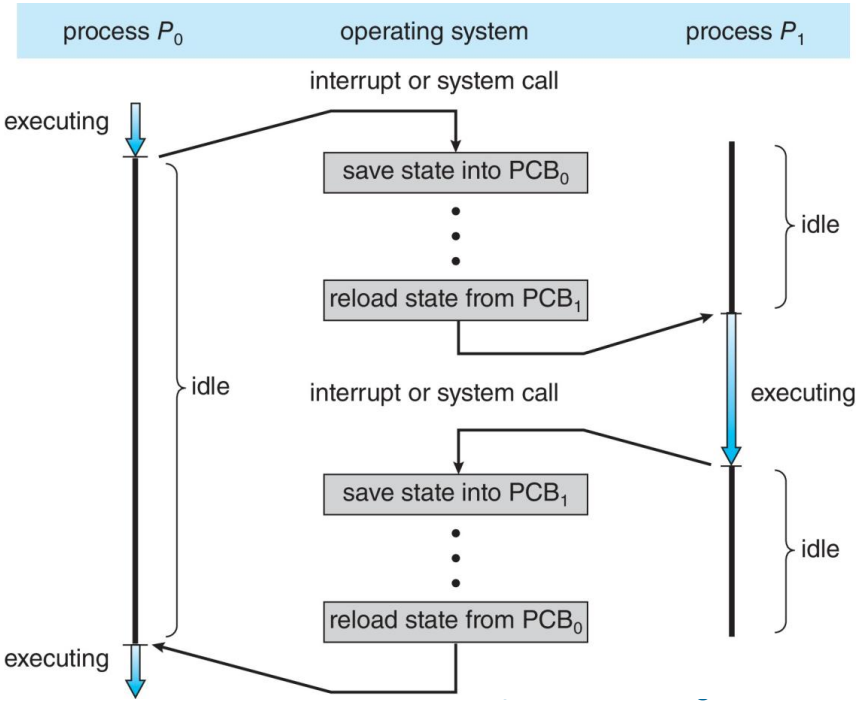


NB. Cette structure permet au noyau de gérer efficacement les processus.

Changement de contexte

- **Pourquoi a-t-on besoin de toutes ces données (càd PCB)?**
  - Dans un système **multiprogrammé**, on a souvent besoin **d'interrompre** un processus et de redonner le contrôle de l'UCT à un autre processus.
  - Il **faut mémoriser toutes les informations nécessaires** pour pouvoir **relancer le processus** courant dans le même état.
  - Le processus en cours est interrompu et un ordonnanceur est appelé. Ce dernier s'exécute en mode noyau (kernel) pour pouvoir manipuler les PCB.

Le changement de contexte a un coût: il va consommer de la mémoire et des cycles UCT, pour décharger le PCB du processus qui était en cours d'exécution et charger le PCB du processus qui va s'exécuter.

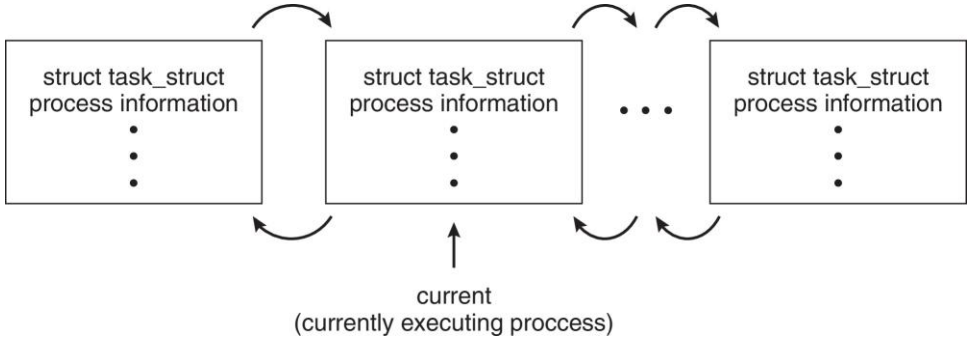


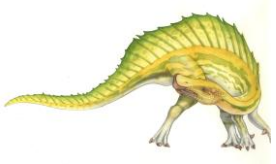
Représentation d'un processus sous Linux

Chaque processus sous Linux est représenté par une structure de données appelée **task_struct**, qui contient diverses informations essentielles :

structure `task_struct`

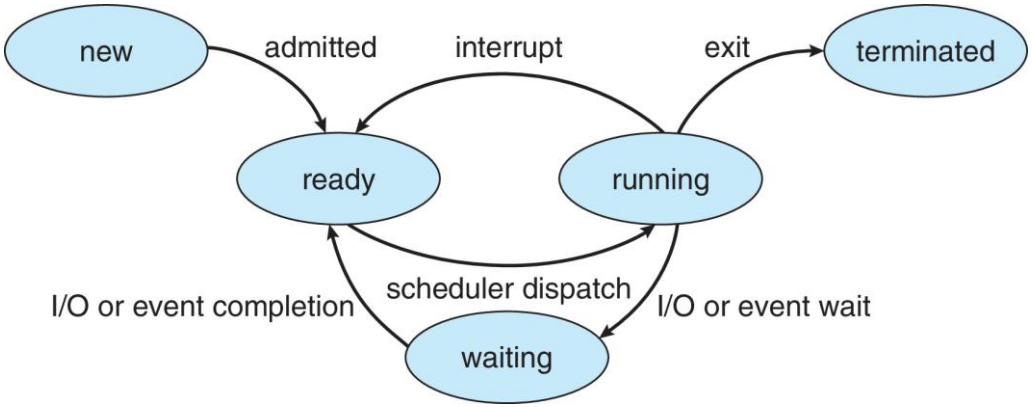
```
pid t_pid;                      /* process identifier */
long state;                     /* state of the process */
unsigned int time_slice         /* scheduling information */
struct task_struct *parent;     /* this process's parent */
struct list_head children;      /* this process's children */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

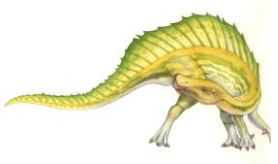




Les États d'un Processus

- Lorsqu'un processus s'exécute, il **change d'état** :
  - **Nouveau (New)** : le processus est en cours de création.
  - **En cours d'exécution (Running)** : les instructions sont en train d'être exécutées (**Actif** , **élu**)
    En attente (Waiting) : le processus attend un événement.) (**suspendu , bloqué**)
  - **Prêt (Ready)** : le processus attend d'être affecté à un processeur (**éligible**)
  - **Terminé (Terminated)** : l'exécution du processus est achevée
  - **Zombie**: un processus qui a été totalement désalloué de la mémoire mais reste toutefois présent dans la table des processus. Il est rattaché au processus numero 1 (init) qui s'occupera de le tuer





Les États d'un Processus

Lister des informations sur les processus

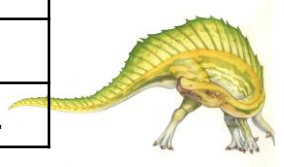
- **Commande ps -aux**

```
ayounes@vm-ubuntu:~$ ps au
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
ayounes     2570  0.0  0.1 235964  6228 tty2     Ssl+ 22:30   0:00 /usr/libexec/gdm-wayland-session env GNOME_SHELL_SESSION_MODE=ubuntu
ayounes     2574  0.0  0.4 298492 16752 tty2     Sl+  22:30   0:00 /usr/libexec/gnome-session-binary --session=ubuntu
ayounes     3544  0.0  0.1  13100  6592 pts/0    Ss   22:30   0:00 bash
ayounes     5319  0.0  0.1  13820  4668 pts/0    R+   23:51   0:00 ps au
ayounes@vm-ubuntu:~$
```

- **The process ID or PID**: a unique identification number used to refer to the process.
- **The parent process ID or PPID**: the number of the process (PID) that started this process.
- **Terminal or TTY** terminal to which the process is connected
- **STAT**: status du processus

Certains états des processus

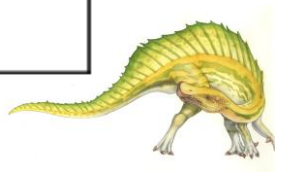
R	exécution (running)
I	endormi (> à 20s)
S	endormi (< à 20s)
D	en attend d'une opération sur disque
T	interrompu
Z	defunct ("zombie") process, terminated but not reaped by its parent.



Les États d'un Processus

Lister des informations sur les processus

Autres états	
>	priorité supérieure à la normale
N	priorité normale
<	priorité inférieure à la normale
A	demande d'un remplacement aléatoire
S	demande de remplacement de page de type FIFO
V	interrompu pendant une opération vfork
E	essai d'exécution d'une commande exit
L	pages verrouillées dans un core
X	en cours d'exécution pas à pas
s	maître de session
W	mis en mémoire swap
+	processus au premier plan



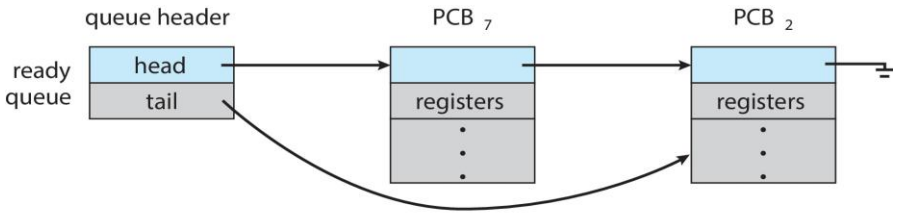
L'ordonnancement (Scheduling)

- **L'ordonnancement** sélectionne un processus pour exécution sur le CPU et gère ses transitions d'état.

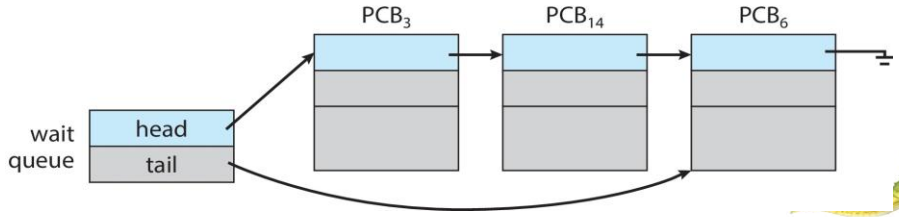
- **Objectif** : **Optimiser l'utilisation du CPU et basculer rapidement les processus**.

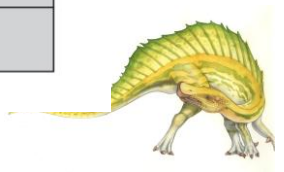
- **Il maintient des files d'attente :**
  - **File prête** (**Ready queue**): : Processus en mémoire, prêts à s'exécuter.
  - **File d'attente** (**Wait queues**): : Processus en attente d'un événement (ex. : E/S).
  - **Les processus passent d'une file à l'autre.**

Prêt (Ready Queue)

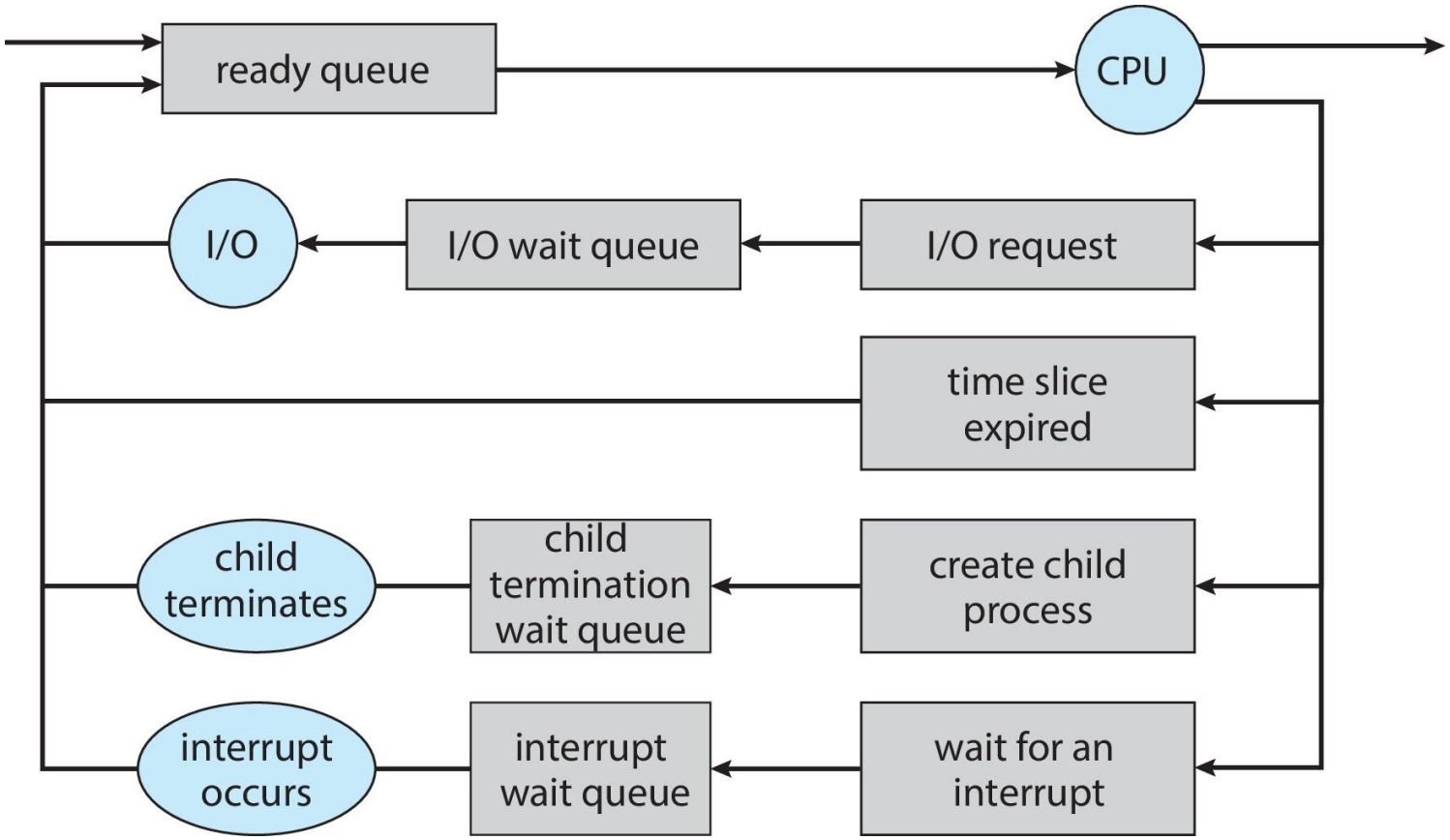


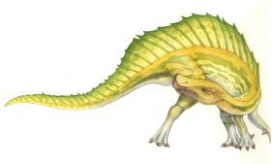
En attente (Waiting queue)





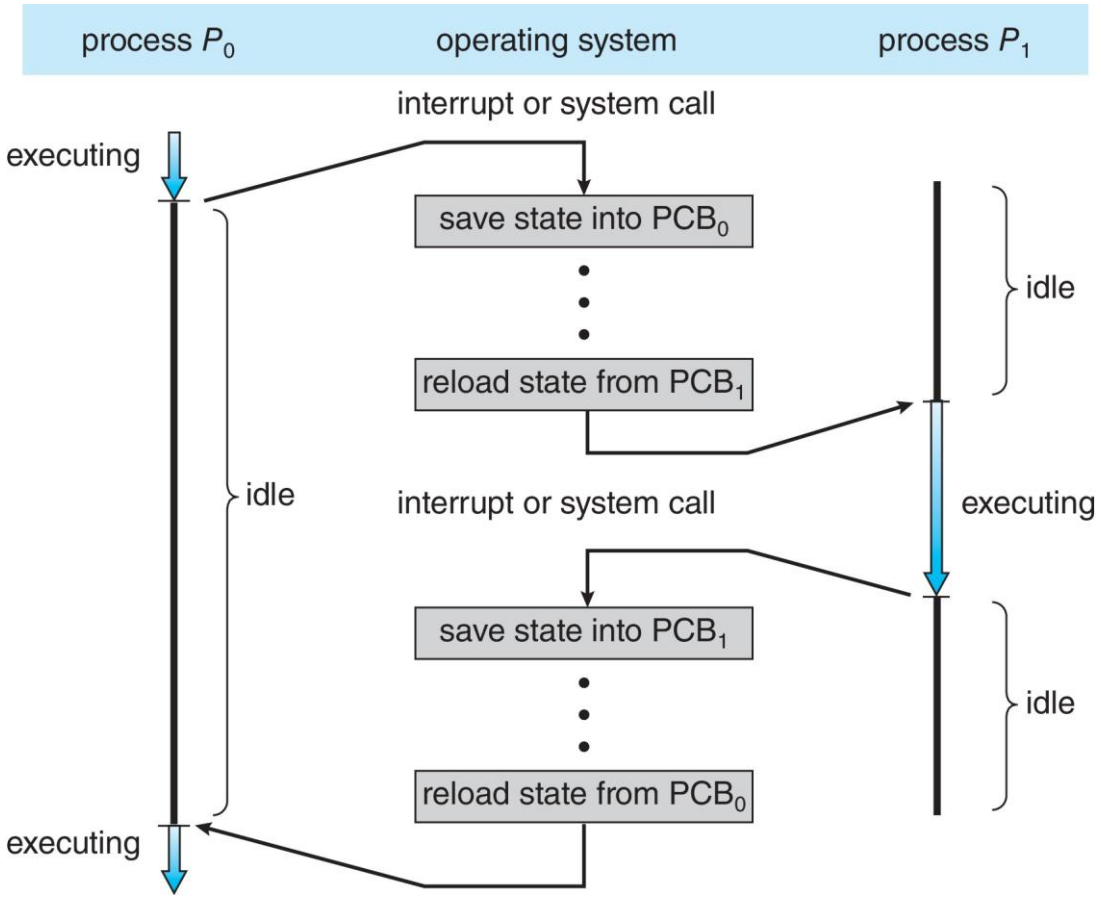
Représentation de l'ordonnancement des processus





Commutation du processeur d'un processus à l'autre

Un **commutation de contexte** **(context switch)** se produit lorsque le processeur passe d'un processus à un autre.



Commutation de contexte

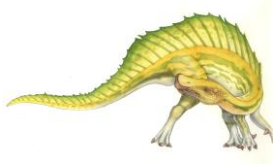
- Lorsque le CPU passe à un autre processus, **il doit sauvegarder l'état de l'ancien** processus et charger celui du nouveau (changement de contexte).

- L'état du processus est stocké dans le **PCB** (Process Control Block).

- Le **changement de contexte** est une surcharge : **aucun travail utile n'est effectué pendant cette opération.**

  - Plus l'OS et le **PCB** sont complexes ➔ plus le changement de contexte est long.

  - Le temps de commutation dépend du matériel ➔ certains processeurs ont plusieurs ensembles de registres, permettant de charger plusieurs contextes simultanément.

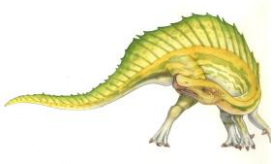


Opérations sur les processus

Le système d'exploitation gère les processus via deux opérations principales

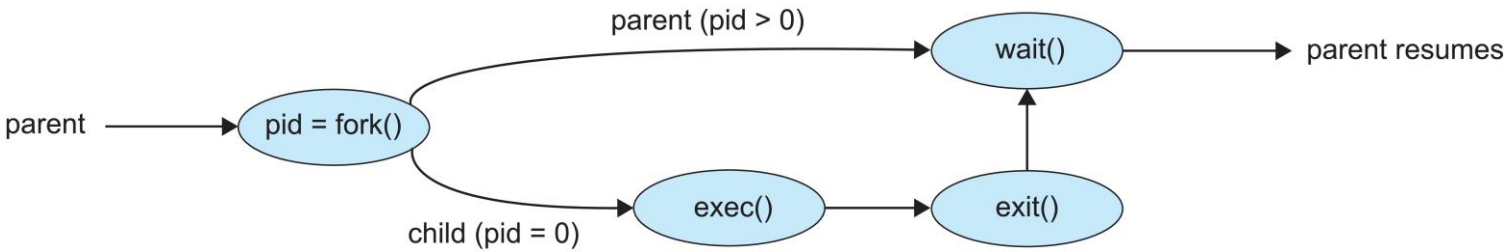
- **Création de processus** : Lancement d'un nouveau processus.
  - `fork()` : **duplication** d'un processus existant pour créer un nouveau processus fils.
  - `exec()` : **remplace** un programme en cours d'exécution par un nouveau programme dans le processus fils.
  - `clone()` **(dans certains systèmes)** : Permet de créer un processus avec un contrôle plus fin sur les ressources partagées

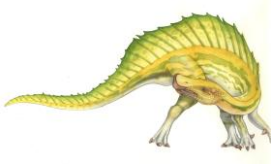
- **Terminaison de processus** : Arrêt et suppression d'un processus.
  - `exit()` : **Termine** un processus et libère ses ressources.
  - `wait()` ou `waitpid()` : Permet au processus père **d'attendre la fin** d'un processus **fils** et de récupérer son statut de terminaison.
  - `abort()` : **Terminer le processus courant** de manière **brutale** et immédiate
  - `kill()` : **Envoie un signal** pour forcer la terminaison d'un processus.



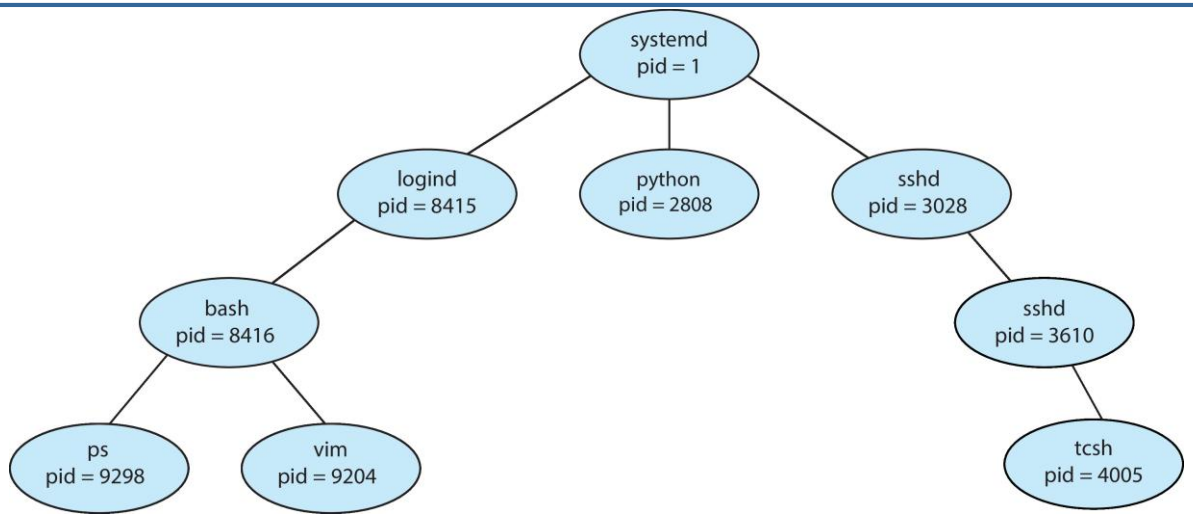
Création des processus

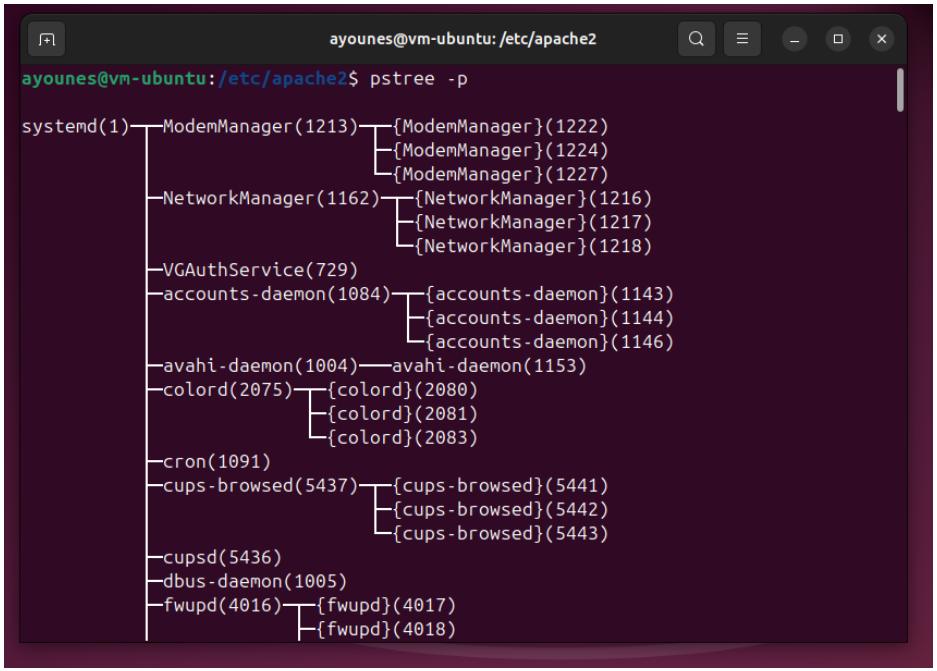
- Un processus parent crée des processus enfants, qui peuvent à leur tour en créer d'autres, formant une **arborescence de processus.**
- Chaque processus est identifié et géré par un **PID (Process Identifier).**
- **Partage des ressources :**
  - Le parent et les enfants partagent **toutes** les ressources.
  - Les enfants partagent un **une partie** des ressources du parent.
  - Le parent et les enfants ne partagent **aucune** ressource.
- **Options d'exécution :**
  - Le parent et les enfants s'exécutent en **parallèle**.
  - Le parent **attend** la fin des enfants avant de continuer.
- **Espace d'adressage :**
  - L'enfant est une **copie du parent** au départ.
  - L'enfant peut charger un **nouveau programme** dans son espace mémoire.

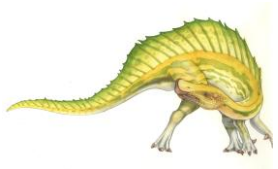




A Tree of Processes in Linux







La primitive fork()

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

pid_t fork(void);
```

- L'appel à **`fork()`** **duplique** le processus.
- **L'exécution continue** dans les deux processus **après l'appel à fork().**
- Tout se passe **comme si les deux processus avaient appelé fork().**
- **La seule différence** (outre le **`PID`** et le **`PPID`**) est la valeur retournée par **`fork(`**

- **La fonction fork retourne** une valeur de type **`pid_t`**. Il s'agit généralement d'un **`int`** ; il et déclaré dans **`<sys/types.h>`**
  - **dans le processus père** (celui qui l'avait appelé): **`fork()`** **retourne** le **`PID du processus fils créé;`**
  - **dans le processus fils**: **`fork()`** **retourne** **`0;`**
  - Le fork **peut échouer** par manque de mémoire ou si l'utilisateur a déjà créé trop de processus; dans ce cas, aucun fils n'est créé et : **`fork()`** **retourne** **`-1.`**

La primitive fork()

- **Exemple 0**

exemple0.c
/home/ayounes/tps

Ouvrir — Enregistrer

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

int main(void) {

printf("Hello avant fork()\n");
fork();
printf("Hello apres fork()\n");

return 0;
}
```

C — Largeur des tabulations : 8 — Lig 7, Col 9 — INS

ayounes@vm-ubuntu: ~/tps

```
ayounes@vm-ubuntu:~/tps$ gcc exemple0.c -o exemple0.o
ayounes@vm-ubuntu:~/tps$ ./exemple0.o
Hello avant fork()
Hello apres fork()
Hello apres fork()
ayounes@vm-ubuntu:~/tps$
```

La primitive fork()

- **Exemple 1**

exemple1.c — /home/ayounes/tps

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

int main()
{
  int i;
  printf("Dans le corps du programme.\n");
  i=fork();
  printf("Apres le fork, i=%d\n",i);
  return 0;
}
```

```
ayounes@vm-ubuntu:~/tps$ gcc exemple1.c -o exemple1.o
ayounes@vm-ubuntu:~/tps$ ./exemple1.o
Dans le corps du programme.
Apres le fork, i=14214
Apres le fork, i=0
ayounes@vm-ubuntu:~/tps$
```

La primitive fork()

- **Exemple 2**

*exemple2.c
/home/ayounes/tps

Ouvrir | Enregistrer

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

int main() {
        pid_t pid = fork();
        if (pid == -1) {
                // Il y a une erreur
                perror("fork");
                return EXIT_FAILURE;
        } else if (pid == 0) {
                // On est dans le fils
                printf("On est dans le fils : Mon PID est %i et celui de mon père est %i\n", getpid(), getppid());
        } else {
                // On est dans le père
                printf("On est dans le père: Mon PID est %i et celui de mon fils est %i\n", getpid(), pid);
        }
        return EXIT_SUCCESS;
}
```

C | Largeur des tabulations : 8 | Lig 7, Col 28 | INS

ayounes@vm-ubuntu: ~/tps

```
ayounes@vm-ubuntu:~/tps$ gcc exemple2.c -o exemple2.o
ayounes@vm-ubuntu:~/tps$ ./exemple2.o
On est dans le père: Mon PID est 14533 et celui de mon fils est 14534
On est dans le fils : Mon PID est 14534 et celui de mon père est 14533
ayounes@vm-ubuntu:~/tps$
```

Termination d'un Processus

- Un processus exécute sa dernière instruction, puis demande au système d'exploitation sa suppression via l'appel système **`exit()`.**
  - Il retourne un **statut** au parent via **`wait()`.**
  - Le système d'exploitation libère ses **ressources**.

- Le parent peut terminer un processus enfant via **`abort()`** pour plusieurs raisons :
  - L'enfant dépasse ses ressources allouées.
  - La tâche assignée à l'enfant n'est plus nécessaire.
  - Le parent se termine et l'OS interdit aux enfants de continuer sans lui.

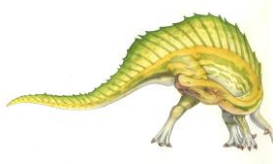
Termination d'un Processus

- Certains systèmes d'exploitation ne permettent pas à un processus enfant d'exister si son parent s'est terminé.
  - Si un processus se termine, tous ses enfants et descendants doivent également arrêtés (**termination en cascade**).
  - Cette terminaison est initiée par le **système d'exploitation**.

- Un parent peut attendre la fin d'un enfant via l'appel system **`wait()`**. L'appel renvoie des informations sur le statut et le PID (identifiant) du processus qui s'est terminé :

```
pid = wait(&status);
```

- Si aucun parent n'attend (pas de **`wait()`**), le processus devient un **zombie**.
- Si le parent se termine sans attendre, l'enfant devient un **orphelin**.



C Program Forking Separate Process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.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 */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

La primitive fork()

- **Exemple 3**

Ouvrir — *exemple3.c — /home/ayounes/tps — Enregistrer

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

int main() {
        pid_t pid = fork();
        if (pid == -1) {
                // Il y a une erreur
                fprintf(stderr,"Fork Failed\n");
                return 1;
        } else if (pid == 0) {
                // On est dans le fils
                execlp("/bin/ls","ls",NULL);
        } else {
                // On est dans le père
                wait(NULL);
                printf("le fils terminer\n");
        }
        return 0;
}
```

C — Largeur des tabulations : 8 — Lig 14, Col 37 — INS

ayounes@vm-ubuntu: ~/tps

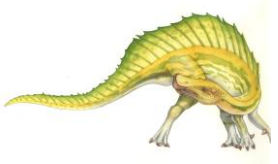
```
ayounes@vm-ubuntu:~/tps$ gcc exemple3.c -o exemple3.o
ayounes@vm-ubuntu:~/tps$ ./exemple3.o

a.out       exemple0.o  exemple1.o  exemple2.o  exemple3.o
exemple0.c  exemple1.c  exemple2.c  exemple3.c  test.c
le fils terminer
ayounes@vm-ubuntu:~/tps$
```

Creating a Separate Process via Windows API

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

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
      fprintf(stderr, "Create Process Failed");
      return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Hiérarchie des Processus sous Android

- Les systèmes d'exploitation mobiles doivent souvent terminer des processus pour récupérer des ressources système, telles que la mémoire.
- Par ordre d'importance, du plus au moins important :
  1. Processus en premier plan (**Foreground process**)
  2. Processus visible (**Visible process**)
  3. Processus de service (**Service process**)
  4. Processus en arrière-plan (**Background process**)
  5. Processus vide (**Empty process**)
- Android commence par supprimer les processus les moins importants.



Multiprocess Architecture – Chrome Browser

- De nombreux navigateurs Web fonctionnaient avec un **seul processus** (certains le font encore).
  - Si un site Web plante, tout le navigateur peut se bloquer ou crasher.
- **Google Chrome** utilise une **architecture multiprocessus** avec 3 types de processus :
  1. **Processus du navigateur** : Gère l'interface utilisateur, le disque et les entrées/sorties réseau.
  2. **Processus de rendu** : Affiche les pages Web, gère HTML et JavaScript.
     - Un nouveau processus de rendu est créé pour chaque site ouvert.
     - Fonctionne en **sandbox**, limitant l'accès au disque et au réseau pour renforcer la sécurité.
  3. **Processus de plug-in** : Un processus distinct pour chaque type de plug-in