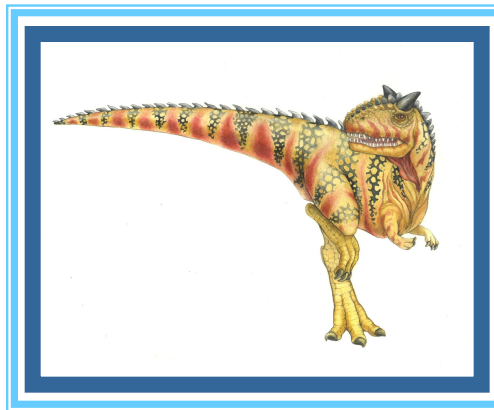


Chapter 3: Les Processus





Plan

- Introduction
- Les processus
 - Concept
 - Types
 - Création des premiers processus
 - Caractéristiques
 - Hiérarchie
 - Le bloc de contrôle (PCB)
 - Les états d'un processus
- Changement de contexte
- L'ordonnancement
- Communication interprocessus
- Opérations sur les processus
 - La primitive fork()
 - Termination d'un Processus
 - Commandes de base





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 tâche
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**.

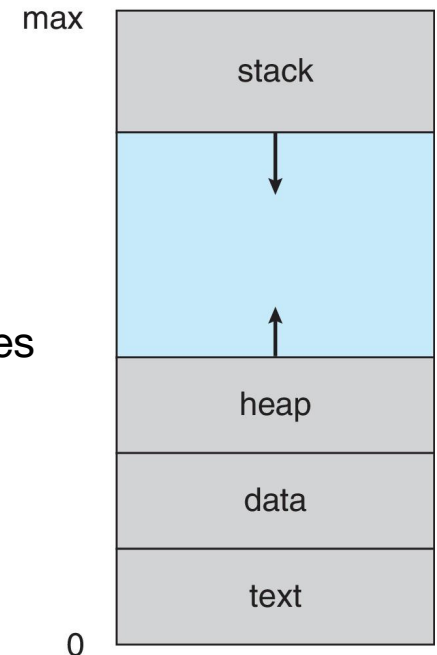




Concept de processus

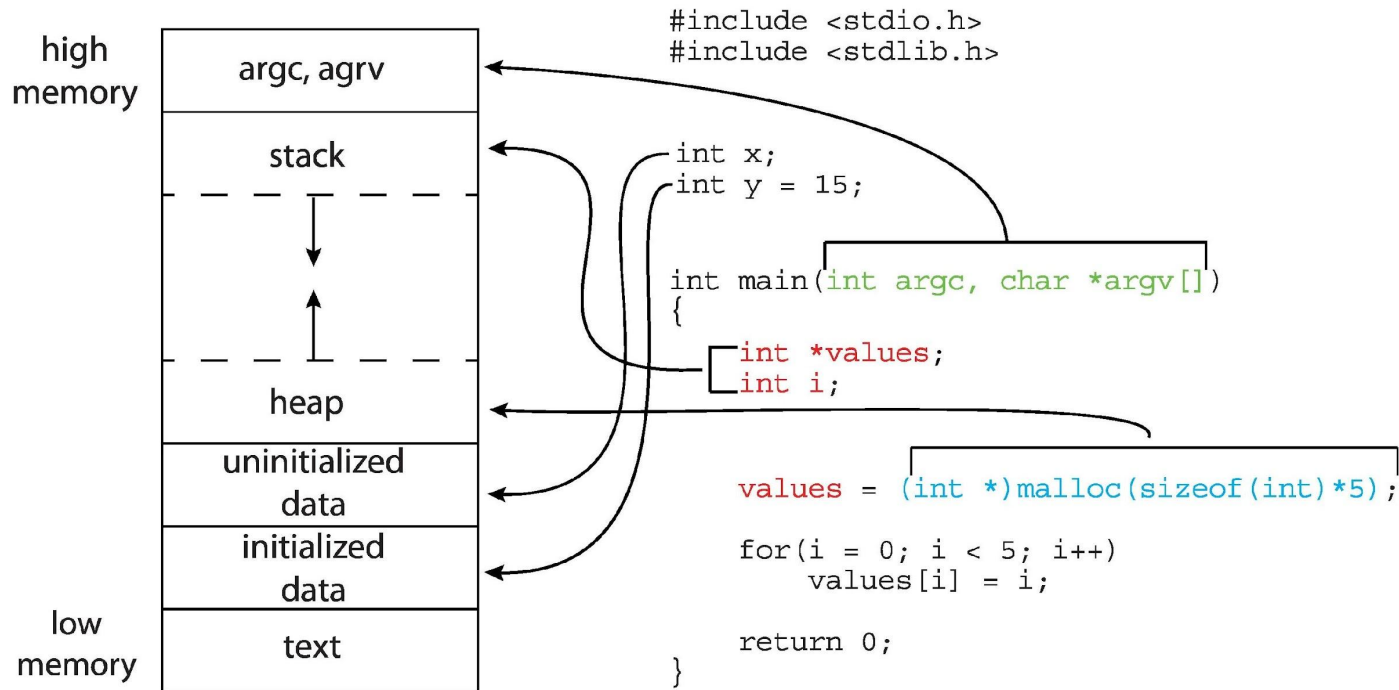
- **Composants d'un processus**

- 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





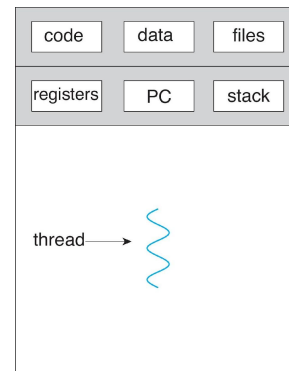
Organisation de la Mémoire d'un Programme C



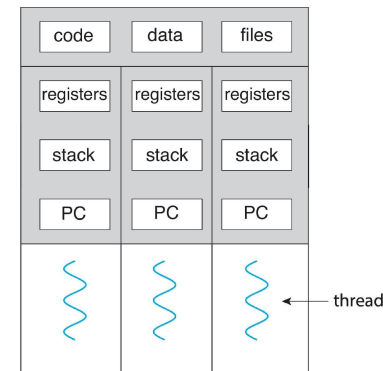


Concept de processus

- 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 Premiers 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;
 - 4 **Init** (pid = 1) qui crée tous les autres processus
 - 4 `[kthreadd]` (pid = 2) le démon de thread du noyau





Caractéristiques des processus

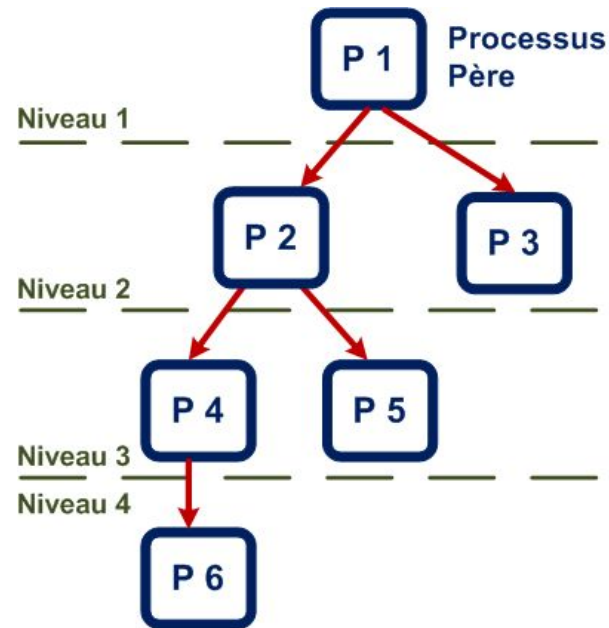
- Depuis sa création, un processus possède notamment les caractéristiques suivantes :
 - **PID** : identificateur du processus (numéro unique)
 - **PPID** : identificateur du processus père
 - **UID** : identificateur de l'utilisateur qui a lancé le processus
 - **GID** : identificateur du groupe de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





Hiérarchie des processus

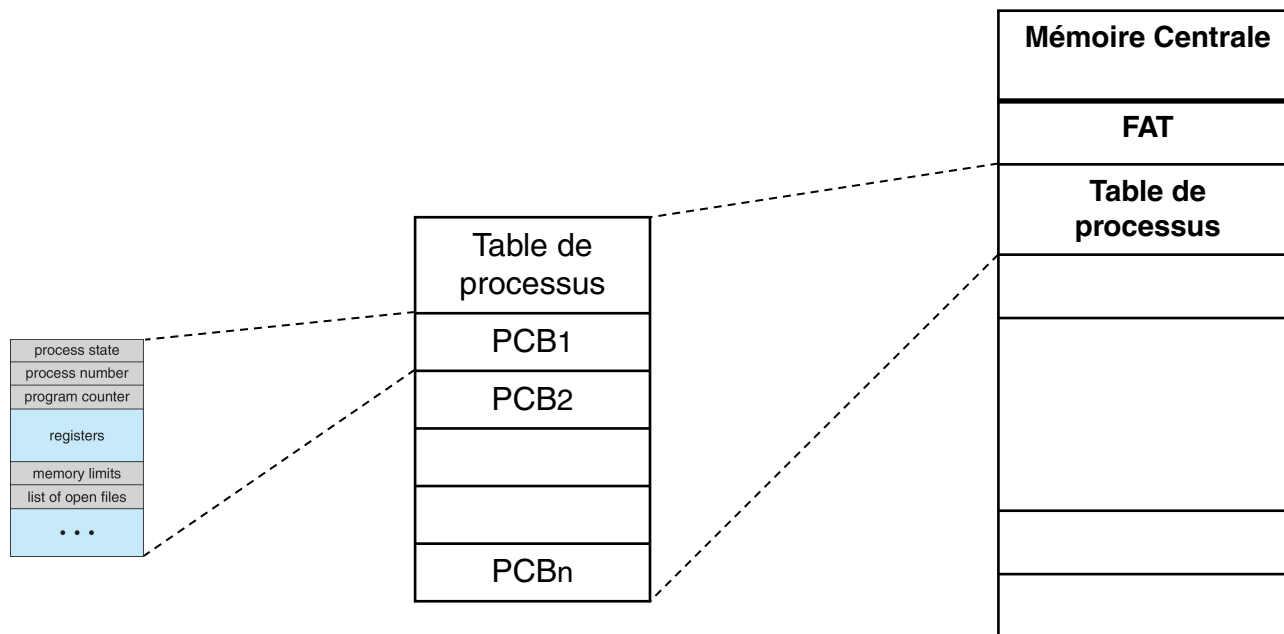
- Un processus **père** peut créer d'autres processus **fil**s.
- Ce dernier, à son tour, crée d'autres processus.
- Un processus a un seul père mais peut avoir plusieurs fils.





Bloc de Contrôle du Processus (PCB)

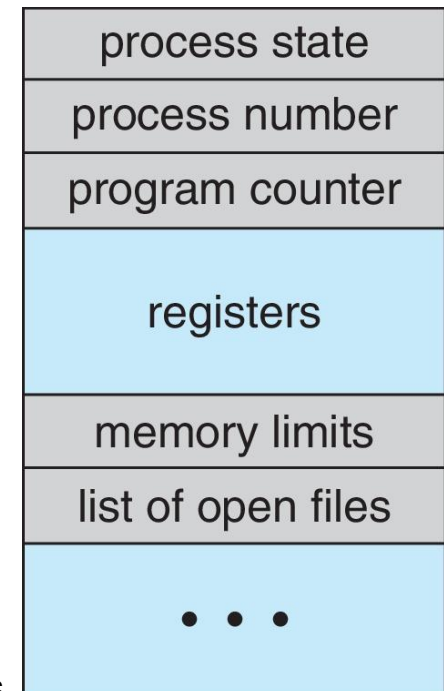
- Pour localiser et gérer tous les processus, le SE maintient **une structure de données** appelée «**table des processus**» qui contient les informations sur tous les processus créés.
- Le Bloc de Contrôle de Processus (**Process control Bloc ou PCB**) est une entrée dans cette table, composée principalement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



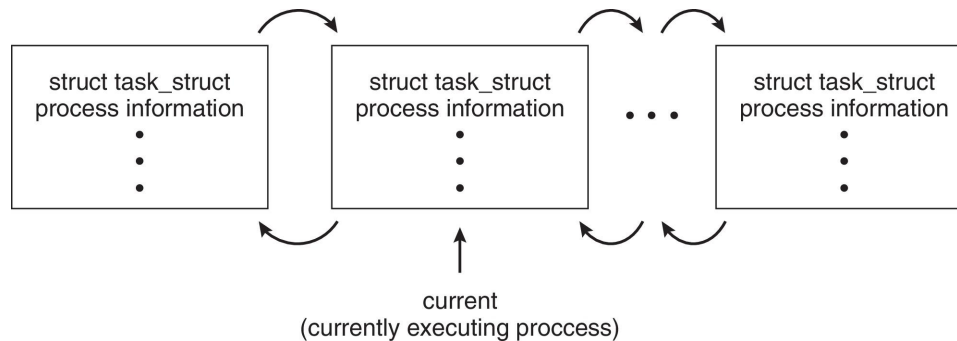


Représentation d'un processus sous Linux

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

```
structure task_struct
```

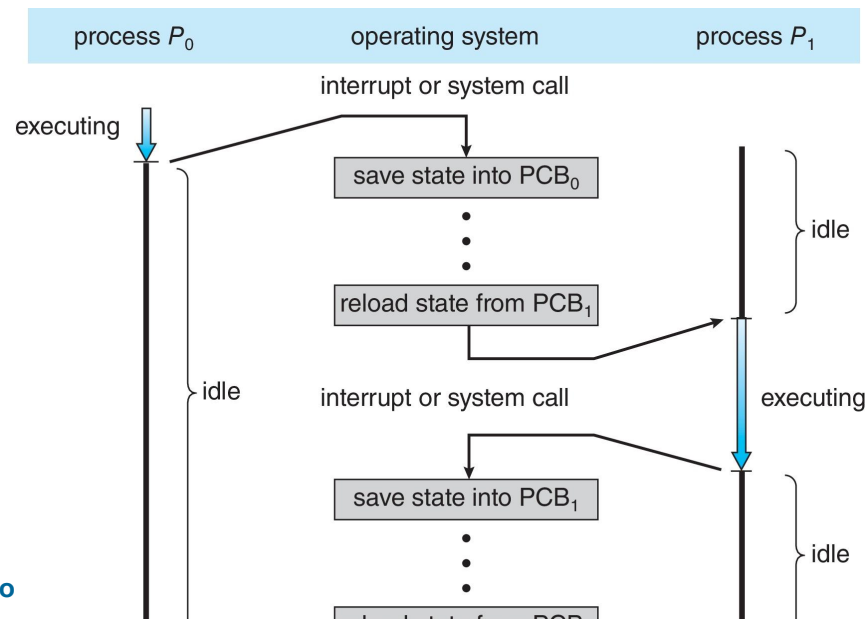
```
pid t_pid;                /* process identifier */
long state;               /* state of the process */
unsigned int time_slice   /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files;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





Changement de contexte

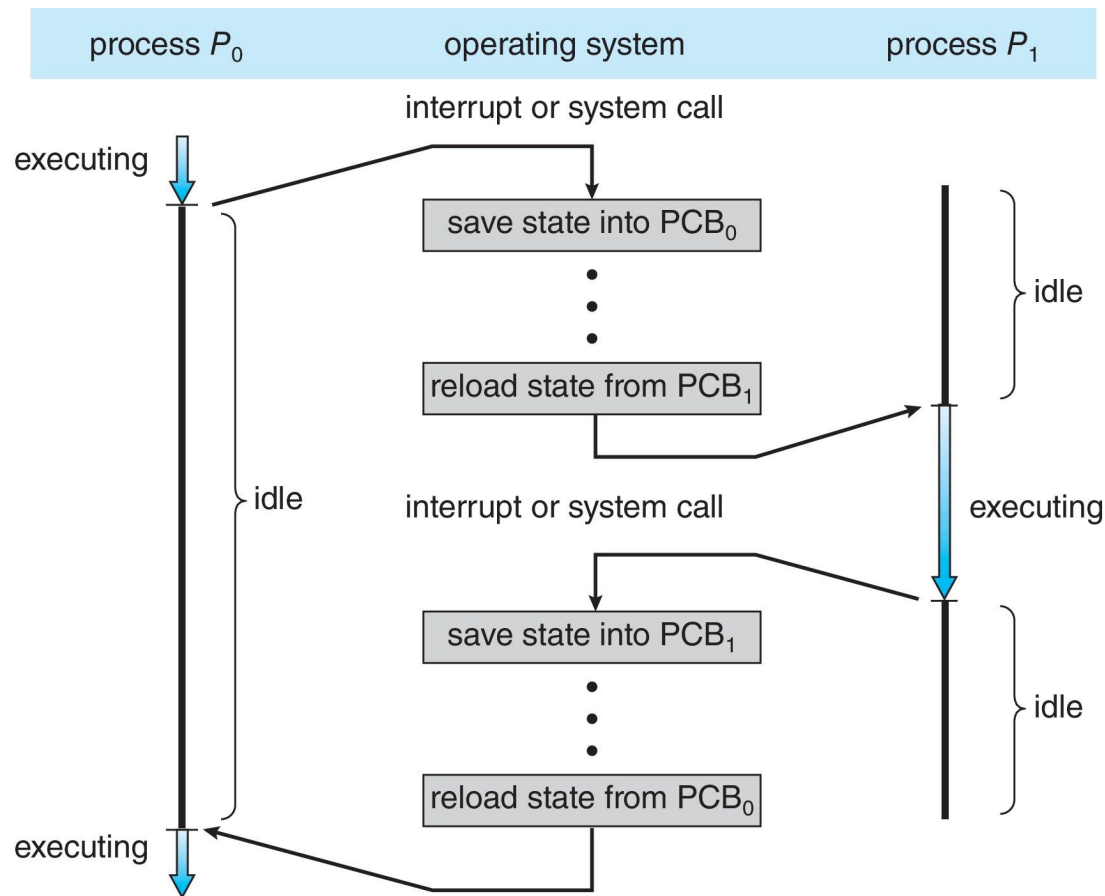
- Lorsque le CPU passe à un autre processus, **il doit sauvegarder l'état de l'ancien processus** et charger celui du nouveau (**changement de contexte**).
- **L'état du processus est stocké dans le PCB** (Process Control Block).
- **Pourquoi a-t-on besoin de toutes ces données (càd PCB)?**
 - Dans un 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.





Changement de contexte

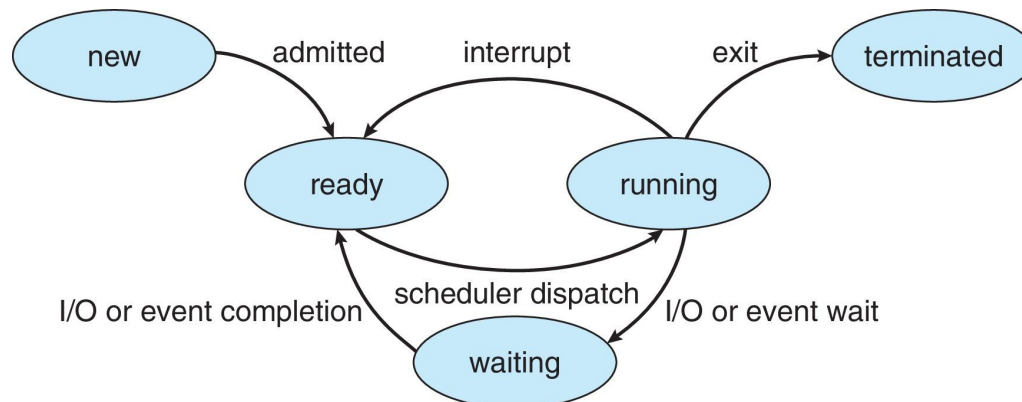
Un **commutation de contexte (context switch)** se produit lorsque le processeur passe d'un processus à un autre.





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

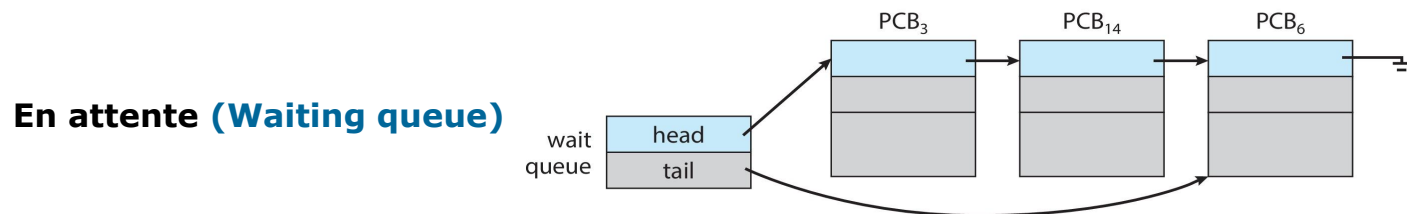
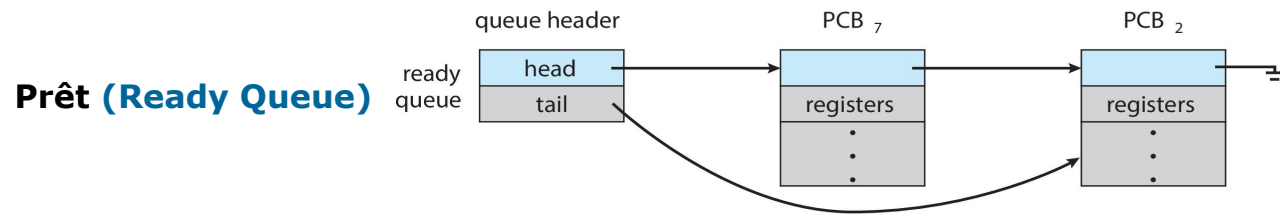
Autres états	
>	priorité supérieure à la normale
N	priorité normale
<	priorité inférieure à la normale
A	demande d'un remplacement aléatoire
S	demande de remplacement de page de type FIFO
V	interrompu pendant une opération vfork
E	essai d'exécution d'une commande exit
L	pages verrouillées dans un core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





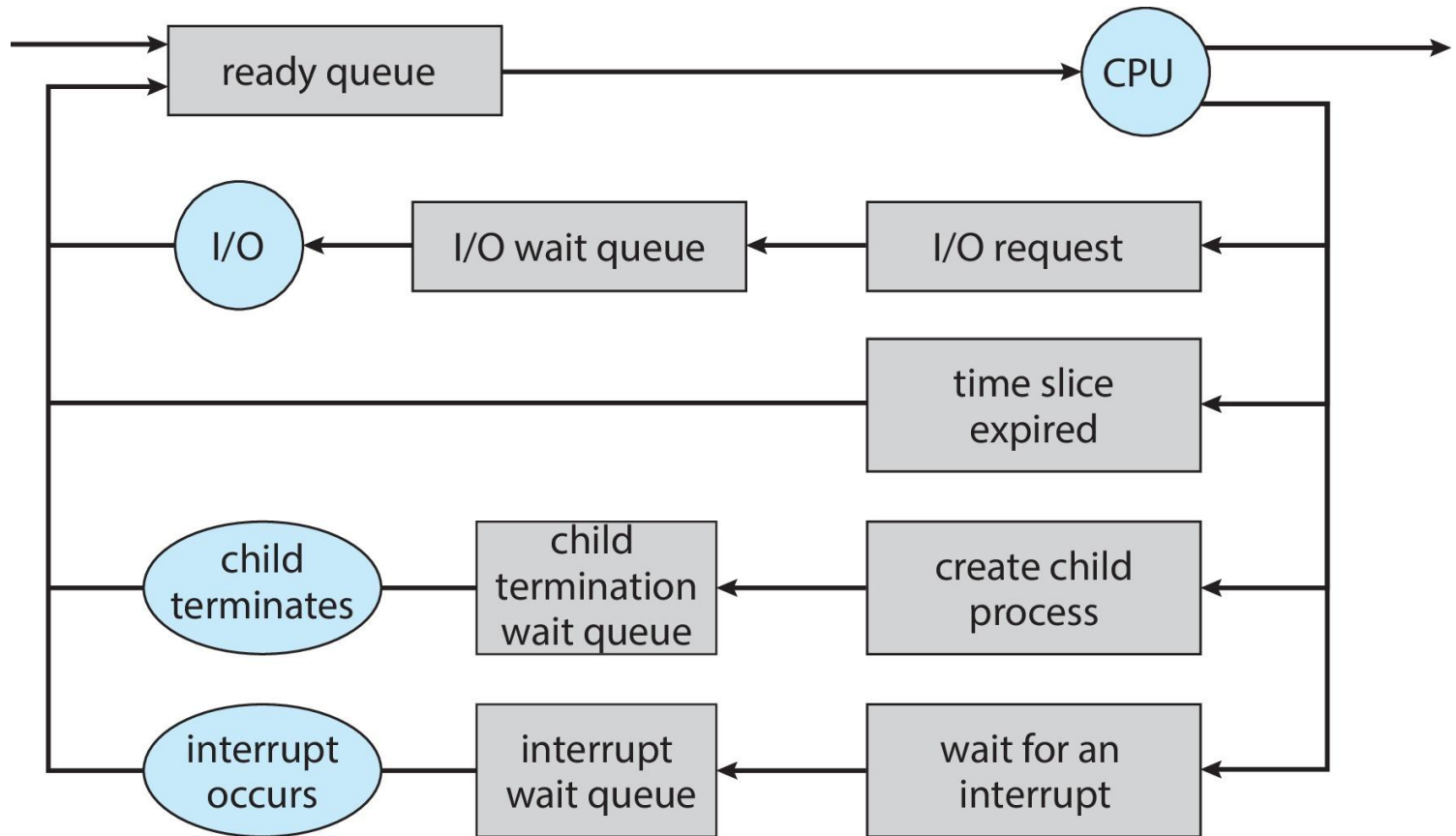
L'ordonnancement (Scheduling)

- **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.**





Représentation de l'ordonnancement des processus





L'Arbre des Processus sous Linux

Dans Linux, tous les processus sont organisés sous forme d'un **arbre hiérarchique**, où chaque processus a un **processus parent** et peut avoir des **processus enfants**.

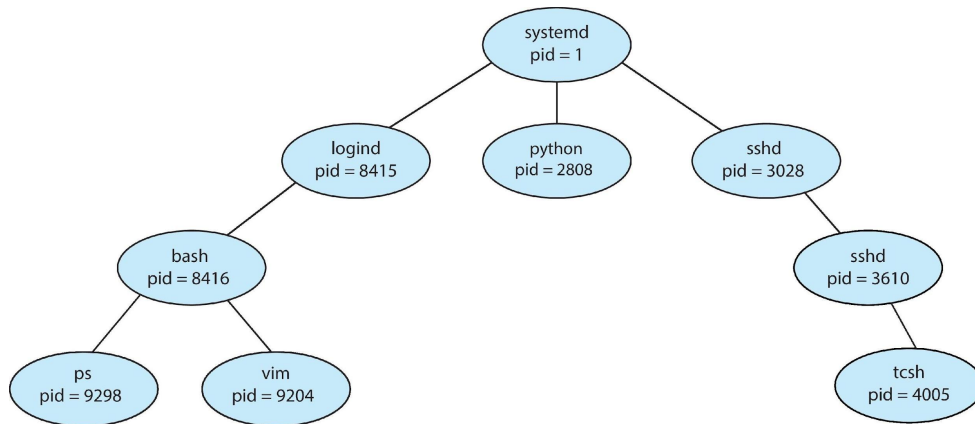
- Le Processus Racine (**init** ou **systemd**)
 - Le tout premier processus (**PID = 1**) est **init** (ou **systemd** sur les distributions modernes).
 - Il est lancé au démarrage du système et crée tous les autres processus.
- Création des Processus
 - Un processus peut en créer d'autres via **fork()**.
 - Le processus parent et le processus enfant partagent certaines ressources.
- Types de Processus
 - Processus de l'utilisateur : exécute des applications (ex. : navigateur, éditeur de texte).
 - Processus du noyau : gère les tâches système (ex. : kthreadd pour les threads du noyau).
- Visualisation de l'Arbre des Processus
 - Commande **ps -e --forest** ou **pstree** pour voir la hiérarchie des processus.

Ce modèle structuré permet une gestion efficace des processus et des ressources du système





L'Arbre des Processus sous Linux



```
ayounes@vm-ubuntu: /etc/apache2
ayounes@vm-ubuntu: /etc/apache2$ pstree -p
systemd(1)─ModemManager(1213)─{ModemManager}(1222)
                        │{ModemManager}(1224)
                        │{ModemManager}(1227)
                        └─NetworkManager(1162)─{NetworkManager}(1216)
                                                │{NetworkManager}(1217)
                                                └─{NetworkManager}(1218)
─VGAuthService(729)
─accounts-daemon(1084)─{accounts-daemon}(1143)
                     │{accounts-daemon}(1144)
                     └─{accounts-daemon}(1146)
─avahi-daemon(1004)─avahi-daemon(1153)
─colord(2075)─{colord}(2080)
             │{colord}(2081)
             └─{colord}(2083)
─cron(1091)
─cups-browsed(5437)─{cups-browsed}(5441)
                  │{cups-browsed}(5442)
                  └─{cups-browsed}(5443)
─cupsd(5436)
─dbus-daemon(1005)
─fwupd(4016)─{fwupd}(4017)
            │{fwupd}(4018)
```





Opérations sur les processus

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

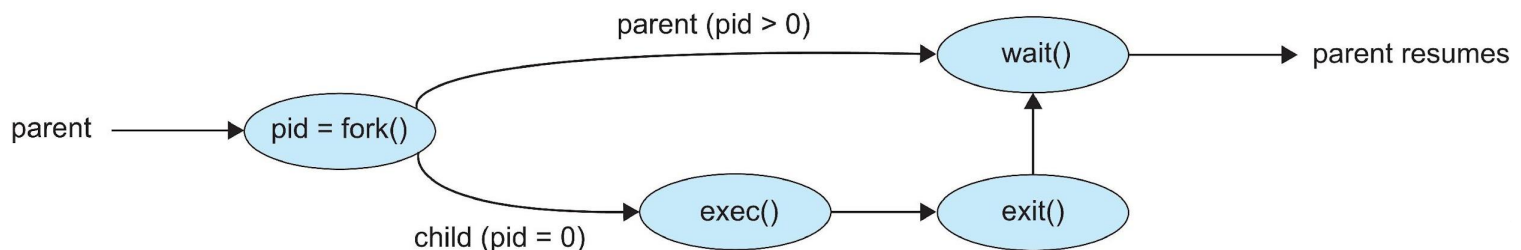
- ● **Création de processus** : Lancement d'un nouveau processus.
 - **fork()** : **duplication** d'un processus existant pour créer un nouveau processus fils.
 - **exec()** : **remplace** un programme en cours d'exécution par un nouveau programme dans le processus fils.
 - **clone()** (**dans certains 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.





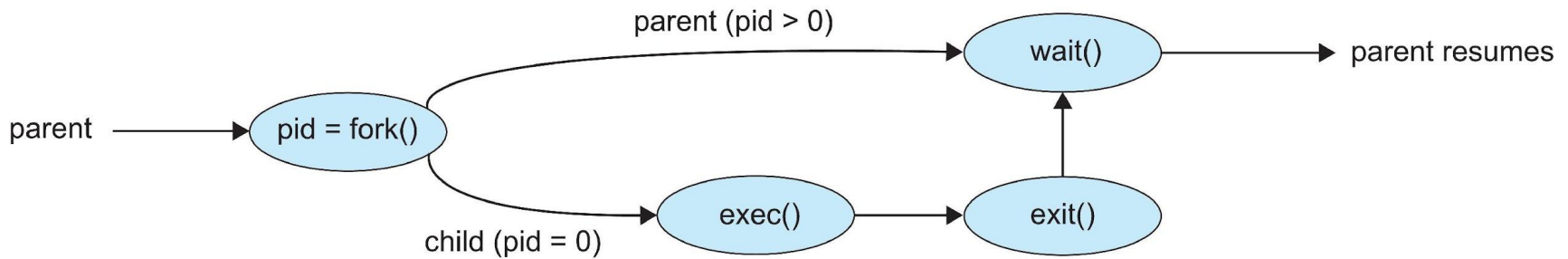
Création des processus

- 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.





Création des processus



- En UNIX-Linux, le **fork()** crée un processus identique au père, excepté son **pid**
- **exec()** peut exécuter un nouveau programme





La primitive fork()

- 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()`

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

pid_t fork(void);
```

- **La fonction `fork` retourne** une valeur de type `pid_t`. Il s'agit généralement d'un `int` ; il est 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 1

```
exemple0.c
/home/ayounes/tps

Ouvrir  Enregistrer

1 #include <sys/types.h>
2 #include <stdio.h>
3 #include <unistd.h>
4
5 int main(void) {
6
7     printf("Hello avant fork()\n");
8     fork();
9     printf("Hello apres fork()\n");
10
11     return 0;
12 }
```

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 2

```
exemple1.c
/home/ayounes/tps
Enregistrer

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4
5 int main()
6 {
7     int i;
8     printf("Dans le corps du programme.\n");
9     i=fork();
10    printf("Après le fork, i=%d\n",i);
11    return 0;
12 }
```

C ▾ Largeur des tabulations : 8 ▾ Lig 6, Col 2 INS

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





La primitive fork() Exemple 3

```
Ouvrir  [icon]  *exemple2.c  /home/ayounes/tps  Enregistrer  [menu]  [close]  [maximize]  [restore down]

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5
6 int main() {
7     pid_t pid = fork();
8     if (pid == -1) {
9         // Il y a une erreur
10        perror("fork");
11        return EXIT_FAILURE;
12    } else if (pid == 0) {
13        // On est dans le fils
14        printf("On est dans le fils : Mon PID est %i et celui de mon père est %i\n", getpid(), getppid());
15    } else {
16        // On est dans le père
17        printf("On est dans le père: Mon PID est %i et celui de mon fils est %i\n", getpid(), pid);
18    }
19    return EXIT_SUCCESS;
20 }
```

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

```
[icon]  ayounes@vm-ubuntu: ~/tps  [search]  [menu]  [close]  [maximize]  [restore down]

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$
```





Synchronisation de processus

- **wait()**
 - Permet au processus d'attendre la terminaison d'un de ses fils
- **waitpid()**
 - Permet au processus appelant d'attendre de manière sélective la terminaison d'un autre processus.
- **sleep()** : Mise en sommeil d'un processus
 - **sleep(n)** suspend l'exécution du processus appelant pour une durée de **n secondes**





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**.





Programme C : Processus séparé de fork

```
#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 4

```
Ouvrir  [icon]  *exemple3.c  /home/ayounes/tps  Enregistrer  [icon]  [icon]  [icon]  [icon]

1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/types.h>
4 #include <wait.h>
5
6 int main() {
7     pid_t pid = fork();
8     if (pid == -1) {
9         // Il y a une erreur
10        fprintf(stderr, "Fork Failed\n");
11        return 1;
12    } else if (pid == 0) {
13        // On est dans le fils
14        execlp("/bin/ls", "ls", NULL);
15    } else {
16        // On est dans le père
17        wait(NULL);
18        printf("le fils terminer\n");
19    }
20    return 0;
21 }
```

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

```
[icon]  ayounes@vm-ubuntu: ~/tps  [icon]  [icon]  [icon]  [icon]

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$ [ ]
```





Création d'un Processus Séparé via l'API Windows

```
#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, etc.

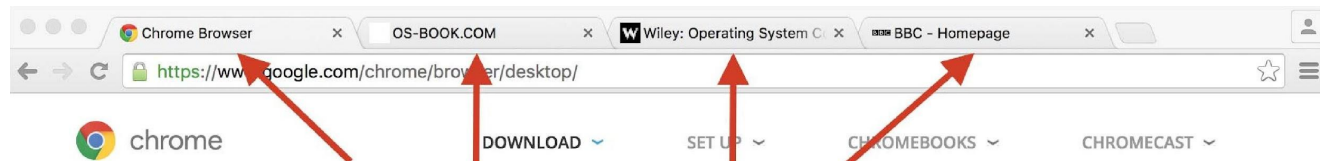
- **Par ordre d'importance, du plus au moins important :**
 1. Processus en premier plan (**Foreground process**)
 2. Processus visible (**Visible process**)
 3. Processus de service (**Service process**)
 4. Processus en arrière-plan (**Background process**)
 5. Processus vide (**Empty process**)
- Android commence par supprimer les processus les moins importants.





Multiprocess Architecture – Chrome Browser

- De nombreux navigateurs Web fonctionnent 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 multiprocesseur** 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.
 - 4 Un nouveau processus de rendu est créé pour chaque site ouvert.
 - 4 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



Each tab represents a separate process.





Communication interprocessus

- Les processus d'un système peuvent être **indépendants** ou **coopératifs**.
- **Un processus coopérant** peut affecter ou être affecté par d'autres processus, notamment par le partage de données.
- Raisons justifiant les processus coopératifs :
 - Partage d'informations
 - Accélération des calculs
 - Modularité
 - Commodité
- Les processus coopératifs nécessitent une communication interprocessus **interprocess communication (IPC)**
- **Deux modèles d'IPC**
 - Mémoire partagée (**Shared memory**)
 - Transmission de messages (**Message passing**)

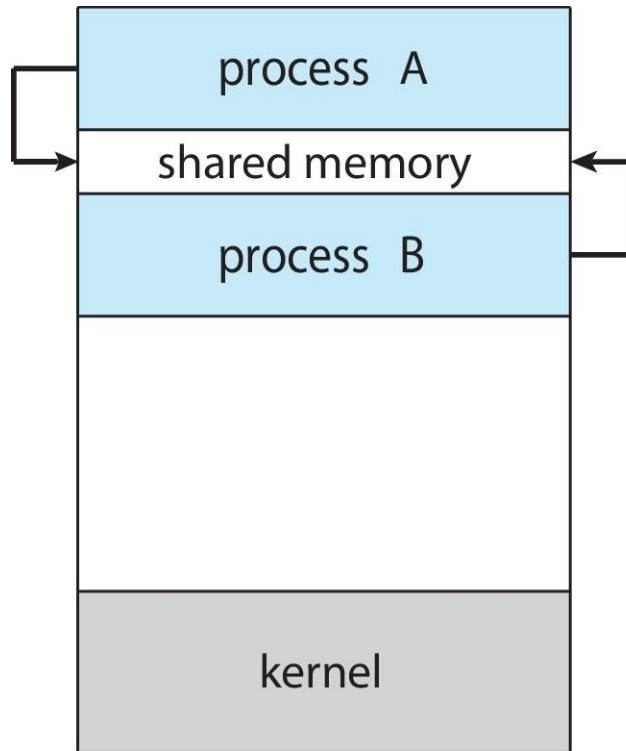




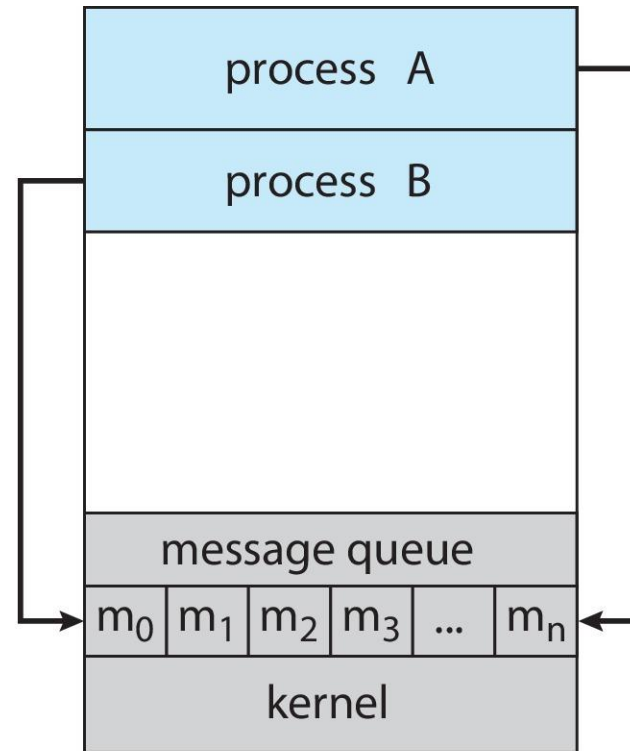
Communication interprocessus

(a) Shared memory.

(b) Message passing.



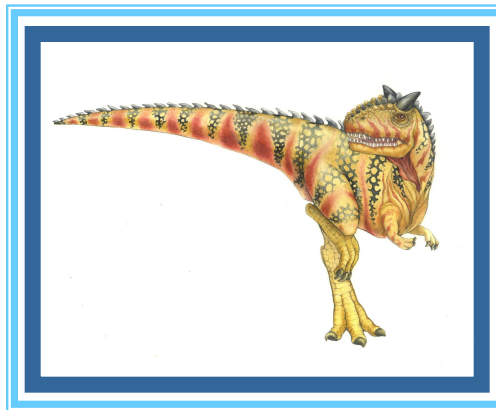
(a)



(b)



Les Commandes de base





Regroupement des processus

- **Enchaînement séquentiel de processus :**

`cmd1 ; cmd2 ; cmd3 ; ... Cmdn`

Les commandes peuvent être enchaînées séquentiellement en utilisant un point virgule.

- **Exécution sous condition d'erreur :**

`cmd1 || cmd2 || cmd3 || ... || cmdn`

Si cmd1 ne se termine pas correctement, alors cmd2 est exécutée, et ainsi de suite.

- **Exécution sous condition de réussite :**

`cmd1 && cmd2 && cmd3 && ... && cmdn`

Si cmd1 se termine correctement, alors cmd2 est exécutée, et ainsi de suite.

```
ayounes@vm-ubuntu: /bin
ayounes@vm-ubuntu:~$ echo "hello"; cd /bin ; pwd ; sleep 15&
hello
/bin
[1] 3652
ayounes@vm-ubuntu: /bin$
```





Redirection

Il est possible de **rediriger** ces fichiers standards vers des fichiers différents en utilisant des **signes de redirection**:

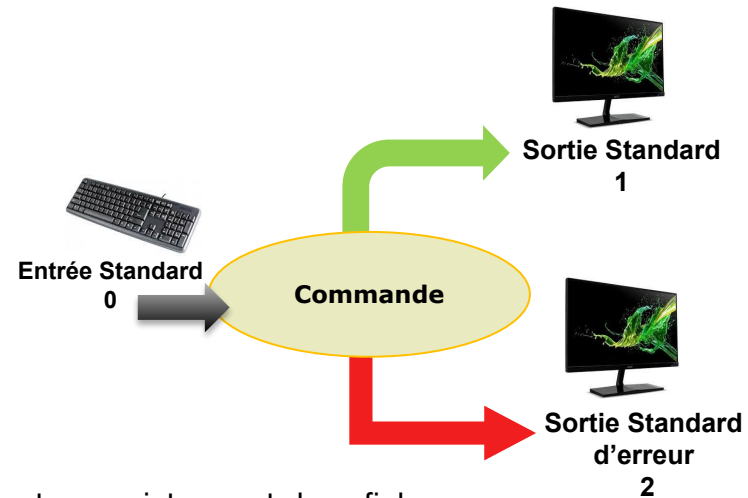
- **<** pour rediriger l'entrée du processus,
- **n>** et **n>>** pour rediriger la sortie du processus.

avec **n** : le descripteur de la sortie standard

- **n=1**: sortie standard(résultat de la commande)
- **n=2**: sortie standard d'erreur
- **n=&**: sortie standard et sortie standard d'erreur.

Remarques:

- par défaut **n=1**
- Commande **n>fich** :si (fich n'existe pas) alors création de fich et enregistrement dans fich.
 - si (fich existe) alors écrasement de son contenu
- Commande **n>>fich** :si (fich n'existe pas) alors création de fich et enregistrement dans fich
 - si (fich existe) alors ajout à la fin de fich



```
ayounes@vm-ubuntu:~$ ls -la 1> resultat.txt 2>erreur.txt
ayounes@vm-ubuntu:~$ cat resultat.txt
total 104
drwxr-x--- 18 ayounes ayounes 4096 mars 19 23:00 .
drwxr-xr-x  3 root    root    4096 oct. 14 22:15 ..
-rw-----  1 ayounes ayounes 2681 mars 19 22:08 .bash_history
-rw-r--r--  1 ayounes ayounes  220 mars 31 2024 .bash_logout
-rw-r--r--  1 ayounes ayounes 3771 mars 31 2024 .bashrc
drwxr-xr-x  4 ayounes ayounes 4096 mars 11 14:36 Bureau
drwx----- 13 ayounes ayounes 4096 févr. 18 10:51 .cache
```





Lancement des Processus

Pour lancer un processus on écrit son **nom d'appel** (**nom du programme** ou **commande**);

Exemple: **sleep 30** // permet de créer un processus qui attend 30 secondes;

Pour lancer un processus en **arrière plan**, il faut suivre son nom d'appel par un **&**

Exemple: **sleep 30&** // crée un processus qui attend 30s en arrière plan

```
19 mars 18:08
ayounes@vm-ubuntu: ~
ayounes@vm-ubuntu:~$ ps
  PID TTY          TIME CMD
 3568 pts/0    00:00:00 bash
 6065 pts/0    00:00:00 ps
ayounes@vm-ubuntu:~$ sleep 30&
[1] 6066
ayounes@vm-ubuntu:~$ ps
  PID TTY          TIME CMD
 3568 pts/0    00:00:00 bash
 6066 pts/0    00:00:00 sleep
 6067 pts/0    00:00:00 ps
ayounes@vm-ubuntu:~$
```





Création des processus

```
ayounes@vm-ubuntu: ~  
top - 00:55:21 up 2:25, 1 user, load average: 0,03, 0,07, 0,07  
Tâches: 324 total, 1 en cours, 323 en veille, 0 arrêté, 0 zombie  
%Cpu(s): 2,1 ut, 1,1 sy, 0,0 ni, 96,8 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st  
MiB Mem : 3866,7 total, 419,1 libr, 2151,7 util, 1633,8 tamp/cache  
MiB Éch: 3764,0 total, 3763,7 libr, 0,3 util, 1715,1 dispo Mem
```

PID	UTIL.	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TEMPS+	COM.
2708	ayounes	20	0	3914720	305432	145460	S	1,7	7,7	1:08.03	gnome-shell
740	root	20	0	244980	9444	8036	S	1,0	0,2	0:27.58	vmtoolsd
5446	www-data	20	0	253896	62868	46976	S	1,0	1,6	0:04.41	apache2
4270	ayounes	20	0	3160256	396248	172688	S	0,7	10,0	0:37.54	firefox
5126	ayounes	20	0	2962076	362704	114932	S	0,7	9,2	0:54.77	Isolated Web Co
745	root	-51	0	0	0	0	S	0,3	0,0	0:00.20	irq/57-vmw_vmci
2937	ayounes	20	0	215180	38916	30980	S	0,3	1,0	0:28.02	vmtoolsd
3537	ayounes	20	0	703748	61216	47764	S	0,3	1,5	0:09.88	gnome-terminal-
5122	ayounes	20	0	2422176	88100	71076	S	0,3	2,2	0:03.69	Isolated Servic
5426	root	20	0	0	0	0	I	0,3	0,0	0:02.93	kworker/0:2-events
5761	root	20	0	0	0	0	I	0,3	0,0	0:00.65	kworker/u513:1-events_unbound
1	root	20	0	23156	14332	9468	S	0,0	0,4	0:02.65	systemd
2	root	20	0	0	0	0	S	0,0	0,0	0:00.03	kthreadd
3	root	20	0	0	0	0	S	0,0	0,0	0:00.00	pool_workqueue_release
4	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	kworker/R-rcu_gp
5	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	kworker/R-sync_wq
6	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	kworker/R-slub_flushwq
7	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	kworker/R-netns

- **PID** : Identifiant unique du processus.
- **USER** : Utilisateur propriétaire du processus.
- **PR (Priority)** : Priorité du processus.
- **NI (Nice)** : Valeur d'ajustement de la priorité du processus.
- **VIRT (Virtual Memory)** : Mémoire virtuelle utilisée par le processus.
- **RES (Resident Memory)** : Mémoire physique réellement utilisée.
- **SHR (Shared Memory)** : Mémoire partagée avec d'autres processus.
- **S (State)** : État du processus (R=en cours d'exécution, S=en veille, D=attente, Z=zombie, T=stoppé).
- **%CPU** : Pourcentage d'utilisation du CPU par le processus.
- **%MEM** : Pourcentage d'utilisation de la mémoire RAM.
- **TIME+** : Temps total d'exécution du processus (hh:mm:ss).
- **COMMAND** : Nom de la commande ou du programme en cours d'exécution.





Lancement des Processus

La commande **ps** affiche les processus courants, Les options de la commande **ps** sont les suivantes:

- **Affichage des processus de l'utilisateur**
 - **ps** → Affiche les processus en cours de l'utilisateur actuel dans le terminal.
 - **ps -u utilisateur** → Affiche les processus appartenant à un utilisateur spécifique.
- **Affichage de tous les processus**
 - **ps -e** ou **ps -A** → Affiche tous les processus du système.
 - **ps aux** → Affiche tous les processus en détail avec les utilisateurs associés.
 - **ps -ef** → Affiche tous les processus avec un formatage détaillé.
- **Filtrage des processus**
 - **ps -C nom_du_processus** → Affiche les processus correspondant à un nom spécifique.
 - **ps -p PID** → Affiche des informations sur un processus spécifique avec son PID.
 - **ps --ppid PPID** → Affiche les processus ayant un PPID (Parent Process ID) donné.
- **Affichage détaillé**
 - **ps -l** → Affiche des informations détaillées sur les processus de l'utilisateur.
 - **ps -f** → Format détaillé avec l'arborescence des processus parents.
 - **ps -o format** → Permet de spécifier les colonnes affichées (ex: **ps -o pid,cmd,%cpu,%mem**).
- **Affichage dynamique et interactif**
 - **ps -T** → Affiche les threads d'un processus.
 - **ps --forest** → Affiche les processus sous forme d'arbre.





Threads

- Jusqu'à présent, un processus utilise un seul **fil d'exécution**.
- Imaginons maintenant qu'un processus puisse avoir plusieurs compteurs de programme (Program Counters), ce qui permettrait à plusieurs parties du programme de s'exécuter en même temps.
 - Ces différentes lignes d'exécution sont appelées **threads**.
- Pour gérer ces threads, le système doit stocker des informations spécifiques à chacun, comme plusieurs compteurs de programme dans le **Bloc de Contrôle du Processus (PCB)**.
- Détails et concepts avancés à voir dans le **Chapitre 4**.



Fin du chapitre 3

