

# Gestion de processus



# Les Caractéristiques d'un processus

## ❖ Concepts du processeur :

- ✓ Exécution d'un programme
- ✓ Les tâches du système sont de : trouver le fichier sur le disque, en fonction de l'organisation des
- ✓ fichiers ; trouver de la place en mémoire
- ✓ Charger le programme en mémoire
- ✓ Trouver le point d'entrée du programme (**main()**)
- ✓ Exécuter la première instruction du programme
- ✓ Continuer, et suivre le déroulement du programme (compteur ordinal ou pointeur d'instruction)
- ✓ A la fin du programme, libérer les ressources qu'il occupe ; obtenir et traiter son résultat.

=> **D'où la nécessité d'utiliser une notion qui définit une exécution d'un programme**

**« le Processus ».**



# Les Caractéristiques d'un processus

- ✓ On sait que les activités parallèles ont permis d'améliorer la rentabilité des machines. **Le parallélisme** peut être lié à des activités spécifiques, et donc obtenu par **des processeurs spécialisés**. Par exemple, un transfert d'entrées-sorties peut être exécuté en même temps qu'un calcul interne au processeur. Il peut aussi être le moyen d'obtenir plus de puissance de calcul dans le même temps imparti, en utilisant plusieurs processeurs de calcul.
- ✓ Plus généralement, un processeur peut exécuter une séquence d'instructions d'un programme, puis ensuite une séquence d'instructions d'un autre programme, avant de revenir au premier, et ainsi de suite. Si la durée d'exécution des séquences est assez petite, les utilisateurs auront l'impression que les deux programmes s'exécutent en **parallèle**. On parle alors de **pseudo-parallélisme**.



# Les Caractéristiques d'un processus

- ✓ **Un processus** est un programme en cours d'exécution. Un processus a besoin de ressources matérielles : l'unité centrale, la mémoire centrale et l'accès à des périphériques d'entrées /sorties . Les caractéristiques d'un processus sont comme suivant :

## Caractéristiques Dynamiques

- Priorité
- Environnement d'exécution...
- Quantité de ressources consommées
- (temps unité centrale utilisé...)



## Caractéristiques Statiques « Fixe au cours de sa vie »

- Un numéro unique: **PID (Process Identifier )**,
- Un propriétaire déterminant les droits d'accès du processus aux ressources : ouverture de fichiers...
- Un **processus parent PPID** dont il hérite la plupart des caractéristiques,
- Un terminal d'attache pour les entrées/sorties.
- Le seul qui ne suit pas cette règle est le premier processus lancé sur le système le processus **init** qui n'a pas de père et qui a pour **PID=1**.

# Les Caractéristiques d'un processus

## ❖ Identifiant d'un processus :

Chaque processus d'un système Linux est identifié par son identifiant de processus unique, appelé **pid (process ID)**. Les identifiants de processus sont **des nombres de 16 bits** assignés de façon séquentielle par Linux aux processus nouvellement créés. Chaque processus a également un processus parent (sauf le processus spécial **systemd**). Vous pouvez donc vous représenter les processus d'un système Linux comme un arbre, le processus **systemd** étant la racine. L'identifiant de processus parent (**parent process ID**), ou **ppid**, est simplement l'identifiant du père de processus.



```
root@hayat-VirtualBox: /home/hayat# pstree
systemd--ModemManager--2*[{ModemManager}]
--NetworkManager--2*[{NetworkManager}]
--3*[VBoxClient--VBoxClient--2*[{VBoxClient}]]
--VBoxClient--VBoxClient--3*[{VBoxClient}]
--VBoxService--8*[{VBoxService}]
--accounts-daemon--2*[{accounts-daemon}]
--acpid
--avahi-daemon--avahi-daemon
--colord--2*[{colord}]
--cron
--cups-browsed--2*[{cups-browsed}]
--cupsd
--dbus-daemon
--fwupd--4*[{fwupd}]
--gdm3--gdm-session-wor--gdm-x-session--Xorg--5*[{Xorg}]
--gnome-session-b--ssh-agent
--2*[{gnome+
--2*[{gdm-x-session}]
--2*[{gdm3}]
--gnome-keyring-d--3*[{gnome-keyring-d}]
--2*[{kerneloops}]
--networkd-dispat
--packagekitd--PK-Backend--sh--update-motd-upd--apt-check--lsb_re+
--3*[{packagekitd}]
--polkitd--2*[{polkitd}]
--rsyslogd--3*[{rsyslogd}]
```

# Les Caractéristiques d'un processus

## ❖ Creation d'un processus :

Deux techniques courantes sont utilisées pour créer de nouveaux processus :

- ✓ **La première** est relativement simple mais doit être utilisée avec modération car elle est peu performante et présente des risques de sécurité considérables. **(la fonction system)**
- ✓ **La seconde technique** est plus complexe mais offre une flexibilité, une rapidité et une sécurité plus grandes.  
**(La fonction fork et exec)**

### A. La fonction system :

La fonction **system** de la bibliothèque standard propose une manière simple d'exécuter une commande depuis un programme. En fait, la fonction **system** crée un sous-processus dans lequel s'exécute le shell Bourne standard (**/bin/sh**) et passe la commande à ce shell pour qu'il l'exécute.

*La fonction system system.c :*

```
int main ()  
{  
    int return_value;  
    return_value = system ("ls -l /");  
    return return_value }  

```



# Les Caractéristiques d'un processus

## ❖ Creation d'un processus :

### B. La fonction fork :

Linux offre une fonction, **fork**, qui produit un processus fils qui est l'exacte copie de son processus parent. Lorsqu'un programme appelle **fork**, **une copie du processus, appelée processus fils, est créée**. Pour créer un nouveau processus, vous utilisez tout d'abord **fork** pour créer une copie du processus courant. Puis vous utilisez **exec** pour transformer un de ces processus en une instance du programme que vous voulez créer.

Le processus parent continue d'exécuter le programme à partir de l'endroit où **fork** a été appelé. Le processus fils exécute lui aussi le même programme à partir du même endroit.

**La différence entre les deux c'est que** : le processus fils est un nouveau processus et dispose donc **d'un nouvel identifiant de processus, distinct de celui de l'identifiant de son processus parent**. Un moyen pour un programme de savoir s'il fait partie du processus père ou du processus fils est d'appeler la méthode **getpid**.





# Les Caractéristiques d'un processus

## ❖ Creation d'un processus :

### B. La fonction fork :

Cependant, la fonction **fork** renvoie des valeurs différentes aux processus parent et enfant -- un processus « rentre » dans le **fork** et deuxième en « ressort » avec des valeurs **de retour différentes**. **La valeur de retour dans le processus père est l'identifiant du processus fils**. La valeur de retour dans le processus fils est zéro. Comme aucun processus n'a l'identifiant zéro, il est facile pour le programme de déterminer s'il s'exécute au sein du processus père ou du processus fils.

#### ➤ Exemple :

#### Utilisation de la fonction « fork »

```
int main ()
{
    pid_t child_pid;
    printf ("ID de processus du programme principal : %d\n", (int) getpid ());
    child_pid = fork ();
    if (child_pid != 0) {
        printf ("je suis le processus parent, ID : %d\n", (int) getpid ());
        printf ("Identifiant du processus fils : %d\n", (int) child_pid);
    }
    else
        printf ("je suis le processus fils, ID : %d\n", (int) getpid ());
    return 0;
}
```





# Les types de processus

✓ Deux types de processus existent:

1. **Les processus utilisateurs**, => issues du Shell de Connexion;
2. Les processus « **démons** » : Ces processus démons assurent un service, ils sont souvent lancés au démarrage de la machine

Exemple des services assurés par des processus daemon sont l'impression, les tâches périodiques, les communications, la comptabilité, le suivi de tâche. Il y'a des processus particuliers comme :

## Le scheduler

- Affectation des ressources aux processus
- Gérer la priorité des processus
- Charger de l'affectation du processeur aux différents processus actifs

## Le syncer

- Chargé de l'actualisation des informations contenues sur les disques
- Il est hyper important d'activer le syncer avant d'arrêter le système, sinon, les partitions ne sont pas stables.

## Le swapper

- **Le mécanisme de swap** repose sur la notion de **page mémoire**. Une page est une unité élémentaire de mémoire pour l'opération de swap. Lorsqu'un processus demande à accéder à une adresse mémoire, le système commence par vérifier si celle-ci est présente en page mémoire. Si oui, le processus y accède immédiatement, sinon, il reçoit un signal page-fault, lui intimant de s'interrompre. Et de reprendre son exécution.



# La visualisation d'un processus

✓ On peut visualiser les processus qui tournent sur une machine avec la commande : **ps (options)**

⇒ les options les plus intéressantes sont :

✓ **-e** (affichage de tous les processus)

✓ **-f** (affichage détaillée). => La commande **ps -ef** donne un truc du genre :

UID	PID	PPID	C	STIME	TTY	TIME	COMMAND
root	1	0	0	Dec 6	?	1:02	init
...							
jean	319	300	0	10:30:30	?	0:02	/usr/dt/bin/dtsession
olivier	321	319	0	10:30:34	ttyp1	0:02	csh
olivier	324	321	0	10:32:12	ttyp1	0:00	ps -ef

le premier processus à pour PID 321, le deuxième 324. Vous noterez que le PPID du process " ps -ef " est 321 qui correspond au shell

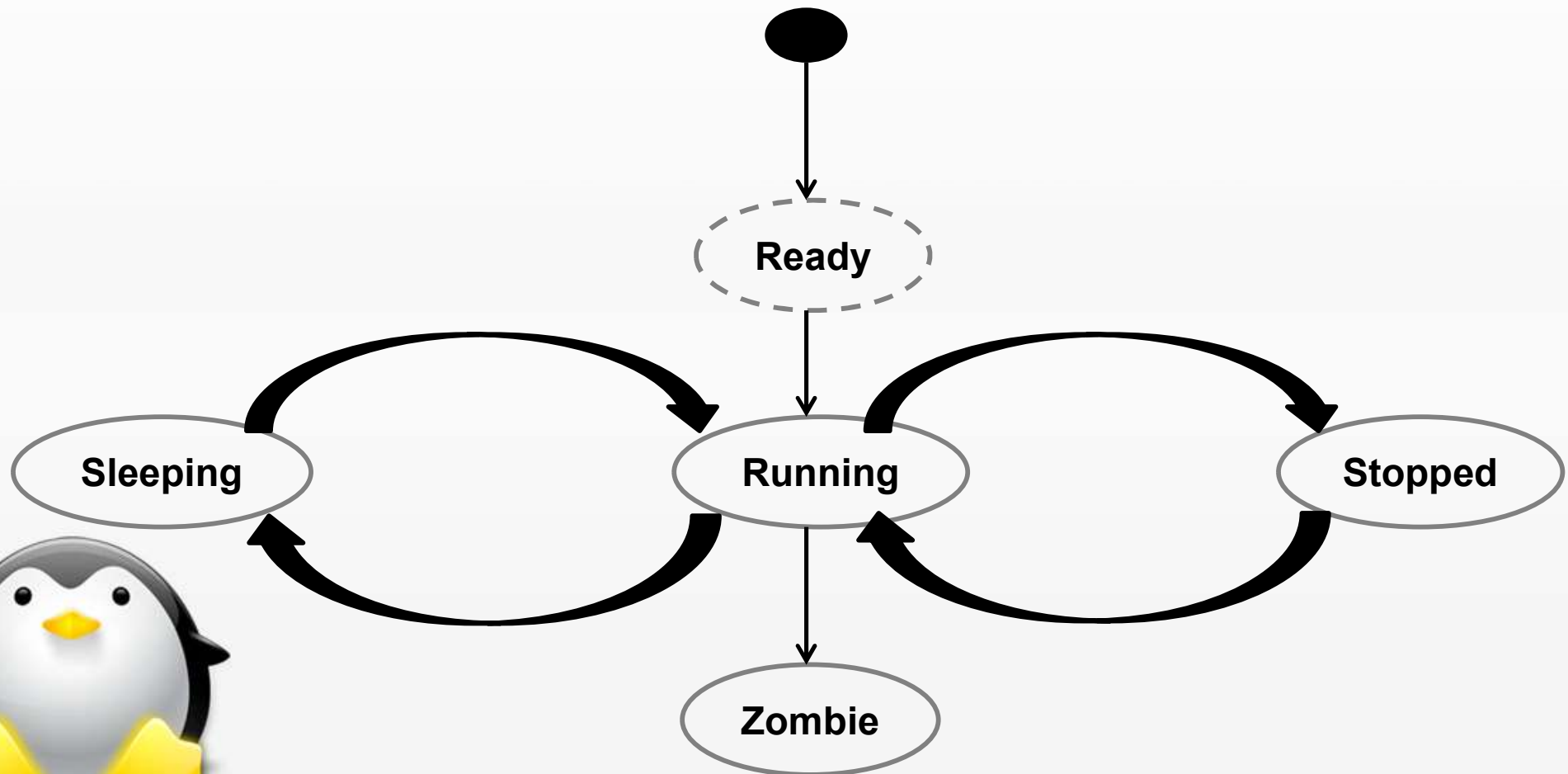
- La signification des différentes colonnes est la suivante:

- ✓ **UID** nom de l'utilisateur qui a lancé le process
- ✓ **PID** correspond au numéro du process - **COMMAND** correspond à son nom
- ✓ **PPID** correspond au numéro du process parent
- ✓ **C** au facteur de priorité : plus la valeur est grande, plus le processus est prioritaire
- ✓ **STIME** correspond à l'heure de lancement du processus
- ✓ **TTY** correspond au nom du terminal - **TIME** correspond à la durée de son traitement



# Les états d'un processus

1. Un processus est une entité dynamique il change son état au cours du temps. Typiquement, on recense cinq états de processus; soit il est en : **Ready, Running, Stopped, Sleeping, Zombie**



# Les états d'un processus

1. **Ready** : Cet état correspond à un processus qui est prêt à s'exécuter mais ne peut pas encore prendre possession du processeur car celui-ci est déjà pris par un autre processus.
2. **Running** : En état d'exécution, donc en pleine possession du processeur.
3. **Stopped** : Le processus a reçu un signal qui l'a stoppé.
4. **Sleeping** : Endormi, en attente d'un évènement, comme la fin d'une entrée sortie (écriture sur un terminal, ...).
5. **Zombie** : Un processus père crée des processus fils à travers un **fork**. Le processus fils termine son exécution, mais le père n'en ai pas encore conscient. Le processus fils se retrouve dans un état zombie, inerte, mais occupe toujours une place dans la table de processus du système.



# Les Ordonnancements d'un Processus

## Le processus d'ordonnancement Scheduler

- ✓ Il est précisément chargé de l'affectation du processeur aux différents processus actifs qui sont rangés dans une liste où se voit attribué une priorité.
- ✓ Lorsque le processus courant est « exécutif » du processeur, le prochain processus à disposer du processeur est celui dont l'ordre de priorité est la plus faible dans la liste.
- ✓ Cette même priorité détermine également du laps de temps accordé au processus.
- ✓ Une fois son temps passé, le processus est réintroduit dans la liste avec un ordre de priorité très élevée.

Vous pouvez utiliser la commande **nice**, pour lancer un programme avec une priorité d'ordonnancement **différente de zéro** en spécifiant la valeur de la priorité d'ordonnancement avec l'option **-n**. Par exemple : vous pourriez invoquer la commande **sort input.txt > output.txt**, une opération de tri longue, avec une priorité réduite afin qu'elle ne ralentisse pas trop le système:

```
nice -n 10 sort input.txt > output.txt
```



# Les Ordonnancements d'un Processus

## Le processus d'ordonnancement Scheduler

- ✓ Le Scheduler adopte trois politiques d'ordonnancement pour gérer l'exécution des processus par le système :
- ✓ **SCHED\_FIFO** : FIFO pour *First In First Out*, premier arrivé premier servit. Dans cette politique, plusieurs files d'attente de processus sont créées chacune avec une priorité différente. Le processeur est affecté en priorité au premier processus qui le demande et ne le relâchera que lorsque :

1- Il aura fini de s'exécuter.

2- Stopper.

3- Endormi en attente d'un évènement.

Ce processus doit évidemment faire partie de la file d'attente de plus haute priorité. Lorsque le processus relâche le processeur, celui-ci sera affecté au processus suivant qui est à l'état Ready et dans la même file d'attente s'il y'en a un, sinon le Scheduler choisit un autre dans la file d'attente de priorité inférieure.



# Les Ordonnancements d'un Processus

## Le processus d'ordonnancement Scheduler

✓ **SCHED\_RR** : Symétrique à **SCHED\_FIFO** en tout point sauf qu'ici chaque processus se voit attribuer une tranche de temps d'exécution qu'il ne pourra pas dépasser. A la fin de cette tranche de temps, il est relégué à la fin de ça file d'attente et le processeur sera réquisitionné pour un autre processus qui est à l'état Ready et de même priorité. Si un tel processus n'existe pas, un autre de priorité inférieure est sélectionné.

⇒ Les processus sous les politiques **SCHED\_FIFO** et **SCHED\_RR** sont considérés comme des processus en « **temps réel** » et sont donc plus prioritaires par rapport aux processus qui sont sous la politique par défaut **SCHED\_OTHER**. Vu que le domaine de priorité des processus en **temps-réel** est par définition augmenté par rapport au processus par défaut.

⇒ **SCHED\_OTHER** : Correspond à la politique par défaut du noyau Linux, les processus se voit attribuer dès leurs création une priorité dite « **statique** » qui peut être modifié via des appels systèmes. Avec cette priorité **statique** et en prenant en compte plusieurs autres facteurs, une priorité « **dynamique** » est calculée et en fonction de cette dernière le **Scheduler** choisi un processus et lui attribue le processeur pour un temps d'exécution maximal.





# Les commandes des processus

- ✓ Un processus est une entité dynamique il change son état au cours du temps. Typiquement, on recense trois états de processus

Commande UNIX	Description
<code>ps -ef</code>	Faire un affichage détaillé de tous les processus
<code>ps -u « user »</code>	Voir les processus d'utilisateur « user »
<code>ps -Al</code>	Permettre d'avoir une sortie assez riche
<code>man ps</code>	Aide sur la commande ps
<code>pstree</code>	Visualiser l'arborescence des processus
<code>top</code>	Visualiser dynamiquement les caractéristiques et les états des processus exécutés en temps réels
<code>kill -signal PID</code>	Exemple : <code>kill -9 25466</code> demande l'arrêt du processus de PID 25446
<code>ps aux</code>	Permettre d'afficher toutes les informations associées à tous les processus en cours d'exécution dans le système
<code>nice &lt;-nbre&gt; &lt;proc&gt;</code>	Définir la priorité avec un ordre nbre d'un processus proc à son lancement.

# La communication interprocessus

✓ La communication interprocessus (IPC) regroupe un ensemble de mécanismes permettant à des processus actifs de communiquer. Ces mécanismes peuvent être classés en trois catégories :

1. Les mécanismes permettant l'échange de données entre les processus ;
2. Les mécanismes permettant la synchronisation entre les processus (notamment pour gérer le principe de section critique) ;
3. Les mécanismes permettant l'échange de données et la synchronisation entre les processus.

⇒ La mémoire principale d'un ordinateur peut aussi être utilisée pour échanger des données entre plusieurs processus concurrents. Selon le type de processus, les mécanismes utilisés ne sont pas les mêmes :

⇒ Dans le cas de processus lourds (process en anglais), les espaces mémoires des processus ne sont pas partagés. On utilise alors un mécanisme de partage de mémoire, ( comme les segments de mémoire partagée dans Unix )



⇒ Dans le cas de processus légers (thread en anglais), l'espace mémoire des processus est partagé, la mémoire peut donc être utilisée directement.

# La communication interprocessus

## La synchronisation des processus

✓ Les mécanismes de synchronisation sont utilisés pour résoudre les problèmes de sections critiques et plus généralement pour bloquer et débloquer des processus suivant certaines conditions; parmi eux il y'a :

1. **Les verrous** : permettent de bloquer tout ou une partie d'un fichier. Ces blocages peuvent être réalisés soit pour les opérations de lecture, soit d'écriture, soit pour les deux.
2. **Les sémaphores** : sont un mécanisme plus général, ils ne sont pas associés à un type particulier de ressource et permettent de limiter l'accès concurrent à une section critique avec un certain nombre de processus. Pour ce faire les sémaphores utilisent deux fonctions : **P** et **V**, et un **compteur**. La fonction **P** décrémente le compteur, si le compteur est nul le processus est bloqué. La fonction **V** incrémente le compteur et débloque l'un des processus bloqués.
3. **Les signaux** : sont à l'origine destinés à tuer (terminer) un processus dans certaines conditions,



**Le problème des mécanismes de synchronisation est que les processus ne sont bloqués que s'ils les utilisent. De plus, leur utilisation est difficile et entraîne des problèmes d'inter-blocage (tous les processus sont bloqués).**

# La communication interprocessus

## Les signaux :

- ✓ Les Signaux constituent un moyen de communication entre processus. **Mais ils ne permettent pas d'échanger des données**; par exemple **le signal SIGKILL** tue un processus qui effectue un accès à une zone de mémoire qu'il n'a pas allouée. Les signaux peuvent cependant être dérivés vers d'autres fonctions. Le blocage d'un processus se fait alors en demandant l'attente de l'arrivée d'un signal et le déblocage consiste à envoyer un message au processus.
- ✓ **Le signal est un événement asynchrone** : il peut arriver n'importe quand il doit être traité pour dériver le processus via des fonctions prédéfinies

⇒ **Pour réaliser de telles échanges, il faut utiliser des fichiers.** Deux processus UNIX peuvent communiquer par un fichier spécial appelé **tube** ou **(pipe)**. Un processus y met des données et un autre les prend. C'est bien un fichier ( $\Leftrightarrow$  à un inode), mais d'un genre spécial « **invisible** », car il n'existe dans aucun répertoire  $\Rightarrow$  il appartient au système qui le crée et le détruit après usage. Comme c'est un moyen de communication, il lui correspond à un mécanisme de synchronisation



# La communication interprocessus

## Traitement d'un signal « Envoie d'un signal »

- ✓ Un processus peut recevoir **un signal** de deux façons différentes :
- ✓ Un **signal** lui est envoyé par un autre processus par l'intermédiaire de **l'appel système kill ()**. **Par Exemple: le Shell envoie le signal SIGKILL si la commande kill -9 pid est frappé ;**
- ✓ L'exécution du processus a levé une trappe (exécution d'une instruction non autorisée : division par 0) et le gestionnaire d'exception associé positionne un signal pour signaler l'erreur détectée.
- ✓ **Exemple: l'occurrence d'une division par zéro amène le gestionnaire d'exception divide\_error () à positionner le signal SIGFPE.**
- ✓ Un signal ainsi délivré mais pas encore pris en compte par le processus destinataire est qualifié de signal pendant.



# La communication interprocessus

## Traitement d'un signal « Prise en compte d'un signal »

- ✓ Un processus peut recevoir **un signal** de deux façons différentes :
- ✓ Un **signal** lui est envoyé par un autre processus par l'intermédiaire de **l'appel système kill ()**. **Par Exemple: le Shell envoie le signal SIGKILL si la commande kill -9 pid est frappé ;**
- ✓ La prise en compte d'un signal par un processus s'effectue :
  1. Lorsque celui-ci s'apprête à quitter le mode noyau pour repasser en mode utilisateur
  2. Avant de passer dans un état bloqué.
  3. En sortant de l'état bloqué.
  4. Lorsque le signal est ignoré, aucune action n'est entreprise au moment de sa prise en compte. Mais une exception existe cependant concernant **le signal SIGCHLD**. Lorsque ce signal est ignoré, le noyau force le processus à consulter les informations concernant ces fils zombies de manière à ce que leur prise en compte soit réalisée et que les blocs de contrôle associés soient détruits.



# La communication interprocessus

## Traitement d'un signal « Prise en compte d'un signal »

5. Exécuter l'action par défaut Le système d'exploitation associe à chaque signal une action par défaut (Gestionnaire par défaut du signal) ; ces actions par défaut sont de 5 natures :

6. **Abort: abandon ou terminaison du processus et génération d'un fichier core** contenant son contexte d'exécution, ce fichier core est exploitable par l'outil débogueur : SIGFPE, SIGQUIT... ;

7. Exécution d'une procédure spécifique : Le processus ne peut pas modifier l'action par défaut de **SIGKILL** et **SIGSTOP**. Par contre pour des signaux telles que **SIGINT, SIGUSR1, SIGUSR2**..., l'action par défaut peut être modifiée, pour cela le processus doit associer au signal une autre procédure de gestion *c'est le Handler*.

Quand un signal arrive et *le Handler* sera pris en compte par le noyau le processus revient en mode utilisateur pour exécuter la procédure associée, à la fin de l'exécution de la procédure le processus poursuit son exécution à son point de déroutement vers le mode noyau.

8. Un processus fils n'hérite pas des signaux pendants de son père, mais il hérite les gestionnaires (Handler). En cas de recouvrement du code hérité du père (le fils exécute la fonction execlp...), les gestionnaires par défaut sont réinstallés pour tous les signaux du fils.





# La communication interprocessus

## Les signaux :

✓ Il y'a 64 signaux, Les signaux 1 à 31 correspondent aux signaux classiques.

Numéro de signal	Nom du signal	Description
1	SIGHUP	Instruction (HANG UP) – Fin de session
2	SIGINT	Interruption
3	SIGQUIT	Instruction (QUIT)
4	SIGILL	Instruction illégale
5	SIGTRAP	Trace trap
6	SIGABRT	(ANSI) Instruction (ABORT)
6	SIGIOT	(BSD) IOT Trap
7	SIGBUS	Bus error
8	SIGFPE	Floating-point exception - Exception arithmétique
9	SIGKILL	Instruction (KILL) - termine le processus immédiatement
10	SIGUSR1	Signal utilisateur 1

# La communication interprocessus

## Les signaux :

Les numéros 32 à 63 correspondent aux **signaux temps réel**. **Les 32 premiers signaux** (standards) qui sont utilisés couramment :

Numéro de signal	Nom du signal	Description
11	SIGSEGV	Violation de mémoire
12	SIGUSR2	Signal utilisateur 2
13	SIGPIPE	Broken PIPE - Erreur PIPE sans lecteur
14	SIGALRM	Alarme horloge
15	SIGTERM	Signal de terminaison
16	SIGSTKFLT	Stack Fault
17	SIGCHLD ou SIGCLD	modification du statut d'un processus fils
18	SIGCONT	Demande de reprise du processus
19	SIGSTOP	Demande de suspension imbloquable
20	SIGTSTP	Demande de suspension depuis le clavier
21	SIGTTIN	lecture terminal en arrière-plan

# La communication interprocessus

## Les signaux :

Numéro de signal	Nom du signal	Description
22	SIGTTOU	écriture terminal en arrière-plan
23	SIGURG	évènement urgent sur socket
24	SIGXCPU	temps maximum CPU écoulé
25	SIGXFSZ	taille maximale de fichier atteinte
26	SIGVTALRM	alarme horloge virtuelle
27	SIGPROF	Profiling alarm clock
28	SIGWINCH	changement de taille de fenêtre
29	SIGPOLL (System V)	occurrence d'un évènement attendu
30	SIGPWR	Power failure restart
31	SIGSYS	Erreur d'appel système
32	SIGUNUSED	Non utilisé

# La communication interprocessus

## Les signaux :

- ✓ Les Signaux constituent un moyen de communication entre processus. **Mais ils ne permettent pas d'échanger des données**; par exemple **le signal SIGSEGV** tue un processus qui effectue un accès à une zone de mémoire qu'il n'a pas allouée. Les signaux peuvent cependant être déroutés vers d'autres fonctions. Le blocage d'un processus se fait alors en demandant l'attente de l'arrivée d'un signal et le déblocage consiste à envoyer un message au processus.
- ⇒ Pour réaliser de telles échanges, il faut utiliser des fichiers. Deux processus UNIX peuvent communiquer par un fichier spécial appelé **tube** ou **(pipe)**. Un processus y met des données et un autre les prend. C'est bien un fichier ( $\Leftrightarrow$  à un inode), mais d'un genre spécial « **invisible** », car il n'existe dans aucun répertoire  $\Rightarrow$  il appartient au système qui le crée et le détruit après usage. Comme c'est un moyen de communication, il lui correspond à un mécanisme de synchronisation

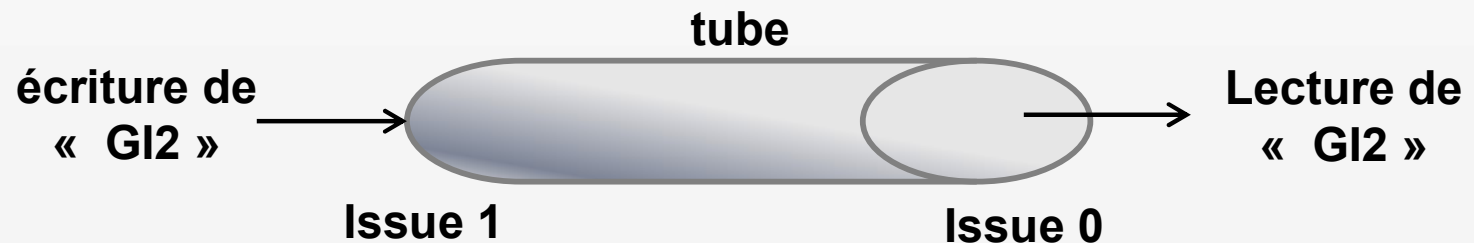


# La communication interprocessus

## Les tubes :

✓ **Un tube** est désigné par des descripteurs et manipulé par les primitives **read()** et **write()**. Mais avec les particularités suivantes:

1. La communication est unidirectionnelle: on écrit à un bout et on lit à l'autre (d'où le nom de tube). Cela entraîne qu'il faut au moins deux descripteurs pour manipuler un tube.
2. La communication est faite en mode **FIFO (first in first out)**. **Premier écrit, premier lu.**
3. Ce qui est lu quitte définitivement le tube et ne peut être relu. De même, ce qui est écrit est définitivement écrit.
4. La transmission est faite en mode flot continu d'octets. L'envoi consécutif des deux séquences, **par exemple "abcd" et "efg"** est semblable à **"abcdefg"** et peut être lu en totalité ou en morceaux comme **"ab"**, **"cde"** et **"fg"**.
5. Pour fonctionner, un tube doit avoir au moins un lecteur et un écrivain.



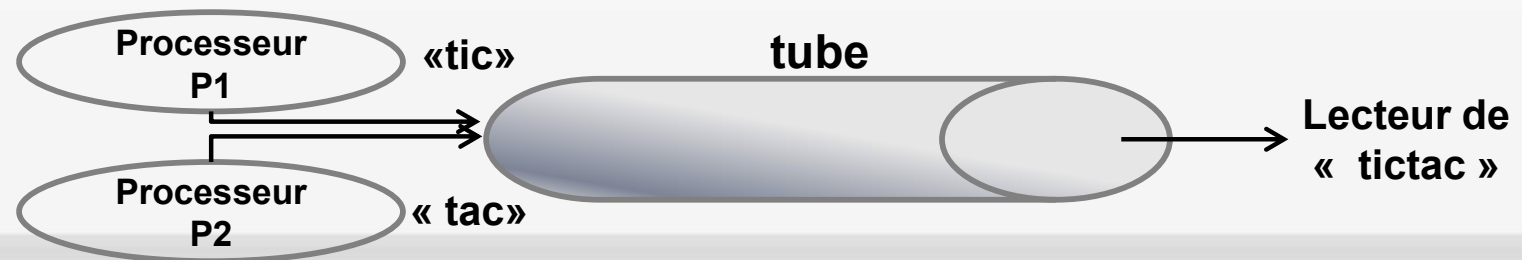
# La communication interprocessus

## Les tubes :

- ✓ Les processus qui **lisent/écrivent** dans un tube sont en général des processus concurrents (**i.e exécutifs en temps réel**) et doivent être liés dans une même généalogie (**i.e Père/Fils**).
- ✓ **En effet, un tube, et ses descripteurs, font partie des caractéristiques hérités par un processus après un fork().**

=> Pour que deux processus puissent donc communiquer des données par un tube, il faut que tous les deux disposent du tube, et pour cela descendre d'un même père (ou ancêtre commun) qui crée le tube. Ce dernier pouvant être lui-même l'un des processus communiquant. (Une coopération semblable mais entre processus quelconques est réalisable par un mécanisme de tubes nommés, qui sont des fichiers disques "réels").

=> On utilise **un tube** à l'aide **de deux descripteurs entiers** : **un en écriture pour l'écrivain**, et **un en lecture pour le lecteur**. La création de tels descripteurs du tube se fait à l'aide de la primitive **pipe()**. La lecture et l'écriture se font avec les primitives **read()** et **write()**.



# La communication interprocessus

## L'utilisation des tubes :

### a) Création d'un Tube: Primitive pipe()

- par exemple : `int pipe (int p[2]);`

crée un tube et lui associe deux descripteurs rendus dans un tableau de deux entiers **p**. Par définition **p[0]** est le descripteur pour lire (sortie du tube) et **p[1]** celui pour écrire (entrée du tube). **La valeur retour de pipe() est 0 en cas de succès, -1 sinon (trop de descripteurs actifs, ou de fichiers ouverts, etc...)**

### b) Lecture dans un Tube: Primitive read()

On peut lire dans un tube à l'aide de la primitive classique **read()**. On utilise le descripteur **p[0]** rendu par **pipe()**. Dans l'exemple suivant :

```
char buf[100];  
int p[2];  
...  
read(p[0], buf, 20);
```

on a une demande de lecture de **20** caractères dans le tube **p**. Les caractères lus sont rendus disponibles dans le tableau des caractères **buf**. Leur nombre est la valeur retour de **read()**.





# La communication interprocessus

## L'utilisation des tubes :

### b) Lecture dans un Tube: Primitive read()

Une précaution voudrait qu'un processus ferme systématiquement les descripteurs dont il n'a pas besoin: ici on a besoin de lire dans le tube p. On doit fermer le descripteur p[1].

```
close(p[1]);
```

```
read(p[0], buf, 20);
```

Cela permet d'éviter des erreurs aboutissant parfois à des situations d'inter-blocage entre processus: des processus communiquent, mais chacun attend que l'autre commence.



# La communication interprocessus

## L'utilisation des tubes :

### c) Ecriture dans un Tube: Primitive write()

On peut écrire dans un tube avec la primitive classique write(). L'écriture est faite en utilisant le descripteur p[1] cette fois-ci. Selon cet exemple :

```
char buf[100];  
int p[2];  
...  
close p[0];  
buf = "texte a ecrire ";  
write(p[1], buf, 20);
```

⇒ C'est une demande d'écriture **de 20 caractères** dans le tube de **descripteur ouvert p[1]**. La séquence à écrire est prise dans le tableau de caractère **buf**. La valeur retour de **write()** est le nombre d'octets ainsi écrits. Là aussi, on a fermé le descripteur p[0] de lecture. **L'écriture dans un tube est atomique (tout est écrit ou rien n'est écrit)** et n'interfère pas avec d'autres écrivains éventuels. Les 20 caractères écrits ici seront consécutifs dans le tube. En d'autres termes, chaque séquence "tic" ou "tac" est écrite en bloc.



# La communication interprocessus

## L'utilisation des tubes :

### d) Comportement des Opérations `read()/write()`

Ces primitives ont un comportement particulier dû à la relation **producteur / consommateur** qui lie les **processus écrivains et lecteurs**. Quand on fait :

```
read(p[0], buf, 20);
```

Cela veut dire :

1) Si le tube désigné contient 20 caractères ou plus, alors 20 caractères seront lus et mis dans la zone désignée par **la variable buf**. S'il contient moins de 20 caractères, ils seront lus de la même façon. En tout cas, la valeur retour **de read()** est le nombre de caractères effectivement lus; 20 ou au moins.

2) Si le tube ne contient aucun caractères (**tube vide**), alors deux cas peuvent se produire:

- ✓ Il y a encore des écrivains susceptibles d'écrire dans le tube (**il existe un descripteur p[1] encore ouvert**); alors **read()** se met en attente qu'il y ait quelque chose d'écrite. **On dit lecture bloquante.**

- ✓ Aucun autre processus ne détient de descripteur en écriture sur ce tube, i.e. il n'y a aucun écrivain. Ce cas est considéré comme une fin de fichier et **read()** renvoi la valeur 0. Donc la seule façon de terminer les lectures dans un tube (e.g. dans une boucle while), c'est de fermer (**close(p[1])**) tous les descripteurs d'écriture dans ce tube .



# La communication interprocessus

## L'utilisation des tubes :

### d) Comportement des Opérations `read()/write()`



Donc la seule façon de terminer les lectures dans un tube (e.g. dans une boucle while), c'est de fermer (**`close(p[1])`**) tous les descripteurs d'écriture dans ce tube .

Il apparaît donc qu'une lecture dans un tube risque de faire attendre un processus jusqu'à ce qu'il y ait écriture dans ce tube, ou que tout écrivain disparaisse (fermeture des descripteurs en écritures). D'où, encore, la précaution, pour un lecteur de fermer tous les descripteurs en écriture.

Quand à la primitive d'écriture :

```
write(p[1], buf, 20);
```

elle se comporte comme suit:



- 1) Dans le cas où il n'y a aucun lecteur sur le tube (tous les descripteurs en lecture sont fermés), c'est une erreur fatale: le processus écrivain se termine par le signal **SIGPIPE**. C'est normal car l'écriture est inutile s'il n'y a pas de lecteurs. C'est le cas par exemple du message "**broken pipe**" sous shell.

# La communication interprocessus

## L'utilisation des tubes :

### d) Comportement des Opérations read()/write()

2) Dans le cas où il y a encore des lecteurs sur ce tube alors :

- S'il y a encore de la place dans le tube, alors les 20 caractères sont écrits et la valeur retour de **write()** est le nombre de caractères écrits. Sinon l'écriture est bloquée **en attente qu'il y ait de la place (qu'un lecteur lise)**.
- Si l'écriture est non bloquante (O\_NDELAY ou O\_NONBLOCK) et s'il n'y a pas assez de place pour écrire dans le tube, **write()** retourne avec 0 ou -1 selon l'indicateur fourni.
- Sinon (tout est bon) le message est écrit et la valeur retour de **write()** sera encore le nombre de caractères écrits.



**=> Mais dans le cas très particulier (toujours non bloquant) où le message à écrire est plus long que la limite PIPE\_BUF du tube, le comportement de write() et la valeur retour ne sont pas précisés. L'écriture dans un tube est donc plus délicate, car elle est abortive dans le cas aucun lecteur, et il faut veiller à ne pas dépasser la capacité du tube.**

Mais, avec les tubes, l'erreur à ne pas commettre est une situation d'inter-blocage illustrée, par exemple, par deux processus qui communiquent dans les deux sens à travers **deux tubes p et q**

```
PROCESSUS_1  
read(p[0], ch, n);  
...  
write(q[1], ch, n);
```

```
PROCESSUS_2  
read(q[0], ch, n);  
...  
write(p[1], ch, n);
```



# La communication interprocessus

## L'utilisation des tubes :

### ➤ Exemple :

```
#include <stdio.h>
char message[25] = « Bon courage les étudiants GI2 »;
main()
{
    /*
     * communication PERE --> FILS par pipe
     */
    int p[2];
    int pipe(int[2]);
    if (pipe(p) == -1) {
        fprintf(stderr, "erreur ouverture pipe\n");
        exit(1);
    }
}
```

```
if (fork() == 0) {      /* fils */
    char c;
    close(p[1]);
    while (read(p[0], &c, 1) != 0)
        printf("%c", c);
    close(p[0]);
    exit(0);
} else {                /* suite pere */
    close(p[0]);
    write(p[1], message, 24);
    close(p[1]);
    exit(0);
}
}
```



# La communication interprocessus

## L'échange des données :

- L'idée de ce mécanisme d'échange est de communiquer en utilisant le principe **des files, les processus voulant envoyer des informations les placent dans la file ; ceux voulant les recevoir les récupèrent dans cette même file.** Les opérations d'écriture et de lecture dans la file sont bloquantes et permettent donc la synchronisation.
- **Ce principe est utilisé par les files d'attente de message, par les tubes, nommés ou non, et par la transmission de messages par Internet ou les sockets Unix.**





# Les sockets

