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...)

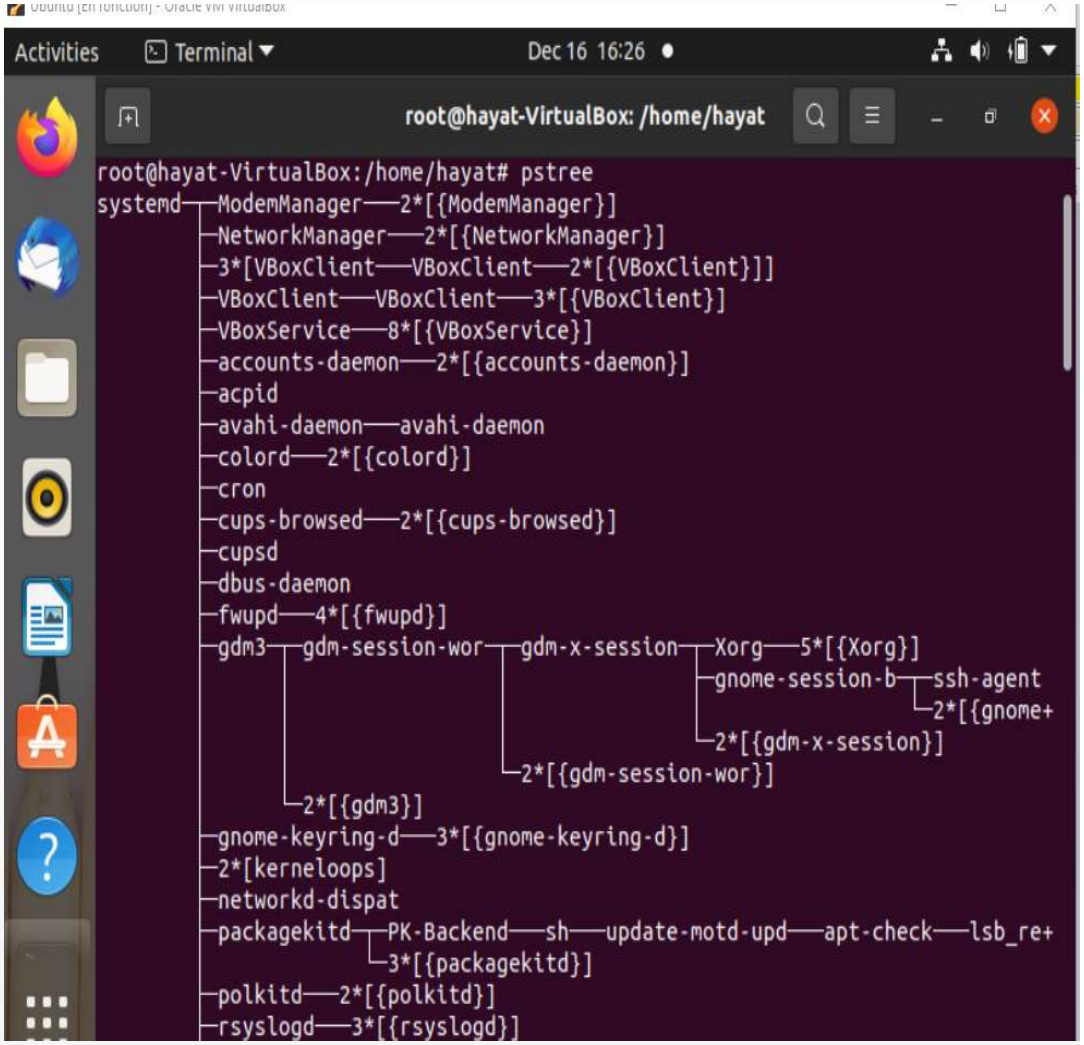
Caractéristiques Statiques
« Fixe au cours de sa vie »

- 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.



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 :**

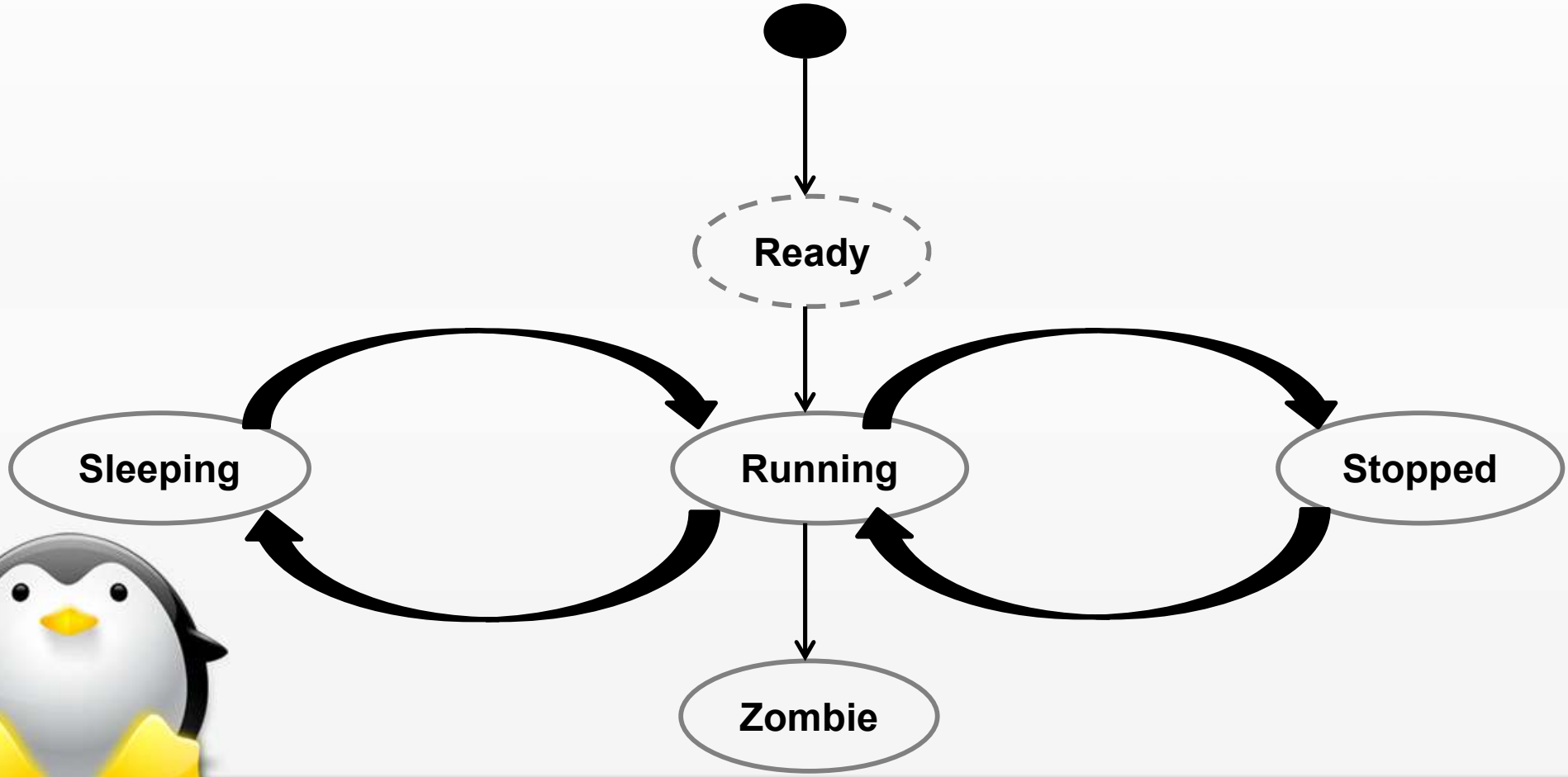
```
UID      PID  PPID  C STIME      TTY      TIME     COMMAND
root     1    0     0 Dec 6      ?        1:02     init
...
jean     319 300    0 10:30:30   ?        0:02     /usr/dt/bin/dtsession
olivier 321 319     0 10:30:34   ttyp1    0:02     csh
olivier 324 321     0 10:32:12   ttyp1    0:00     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 charger 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 créés 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
ps -ef	Faire un affichage détaillé de tous les processus
ps -u « user »	Voir les processus d' utilisateur « user »
ps -Al	Permettre d'avoir une sortie assez riche
man ps	Aide sur la commande ps
pstree	Visualiser l'arborescence des processus
top	Visualiser dynamiquement les caractéristiques et les états des processus exécutés en temps réels
kill –signal PID	Exemple : kill -9 25466 demande l'arrêt du processus de PID 25446
ps aux	Permettre d'afficher toutes les informations associées à tous les processus en cours d'exécution dans le système
nice <–nbre> <proc>	Définir la priorité avec un ordre nbre d'un processus proc à sonlancement.

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; parmis 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 entraine 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 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.

- ✓ **Le signal est un événement asynchrone** : il peut arriver n'importe quand il doit être traité pour dérouter 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 (<=> à un inode), mais d'un genre spécial **« invisible »,** car il n'existe dans aucun répertoire => 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: abondon 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 modifié, 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 associé, 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)	occurence d'un évènement attendu
30	SIGPWR	Power failure restart
31	SIGSYS	Erreur d'appel système
3Z	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 (<=> à un inode), mais d'un genre spécial **« invisible »,** car il n'existe dans aucun répertoire => 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:

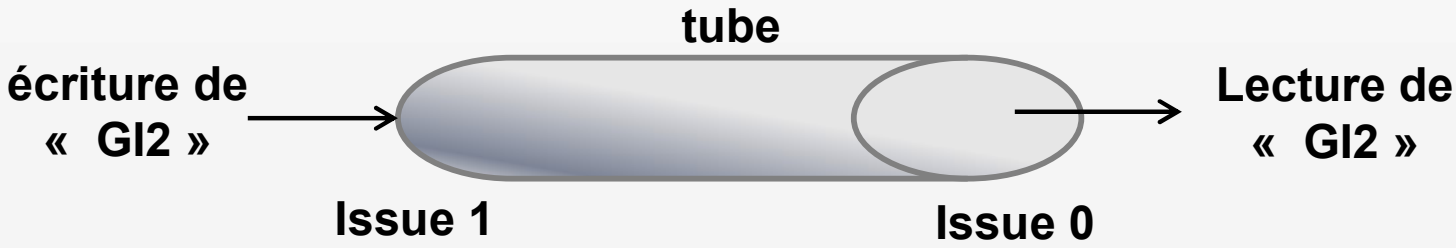
1. La communication est unidirectionnelle: on écrit à un bout et on lit à l'autre (d'où le nom de tube). Cela entraine qu'il faut au moins deux descripteurs pour manipuler un tube.

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 séquence, **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.



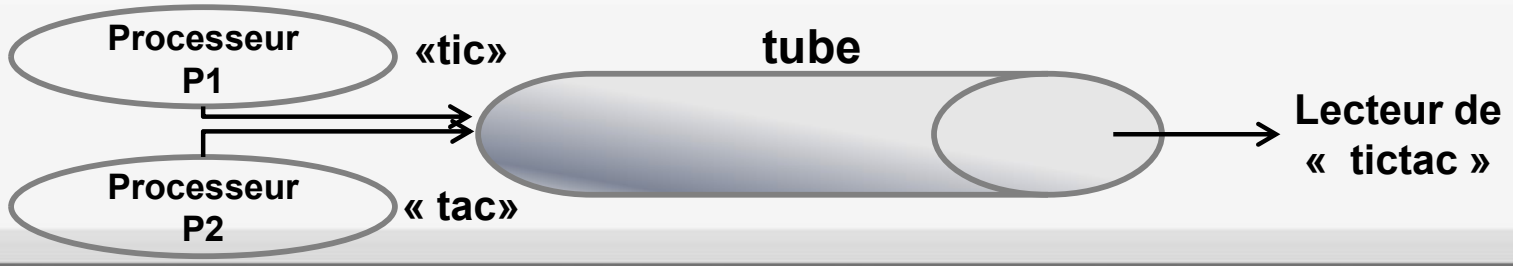
La communication interprocessus

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

C’est quoi un socket ?

- Un socket représente **une interface de communication logicielle** avec le système d’exploitation qui permet d’exploiter les services d’un protocole réseau,
- Via cette **communication logicielle** une application peut **envoyer/recevoir** des données.
- C’est donc un mécanisme de communication bidirectionnelle entre processus **(locaux et/ou distants).**

Interface de programmation (API)

- Un socket désigne aussi :

  => Un ensemble normalisé **des fonctions de communication (lancé par l'Université de Berkeley au début des années 1980) ⇔ API**

- Une interface de programmation qui est proposée dans tous les langages de programmation populaires (C, Java, C#, C++, ...) et répandue dans la plupart des systèmes d'exploitation (UNIX/Linux,Windows, ...).
- La notion de socket a été introduite dans les distributions de Berkeley (un fameux système de type UNIX, dont beaucoup de distributions actuelles utilisent des morceaux de code), c'est la raison pour laquelle on parle parfois de sockets **BSD (Berkeley Software Distribution).**

Pré-requis

La mise en œuvre de l'interface socket nécessite de connaître :

- ✓ **L'architecture client/serveur**
- ✓ **L'adressage IP et les numéros de port**
- ✓ **Notions d'API (appels systèmes sous Unix) et de programmation en langage C**
- ✓ **Les protocoles TCP et UDP, les modes connecté et non connecté**

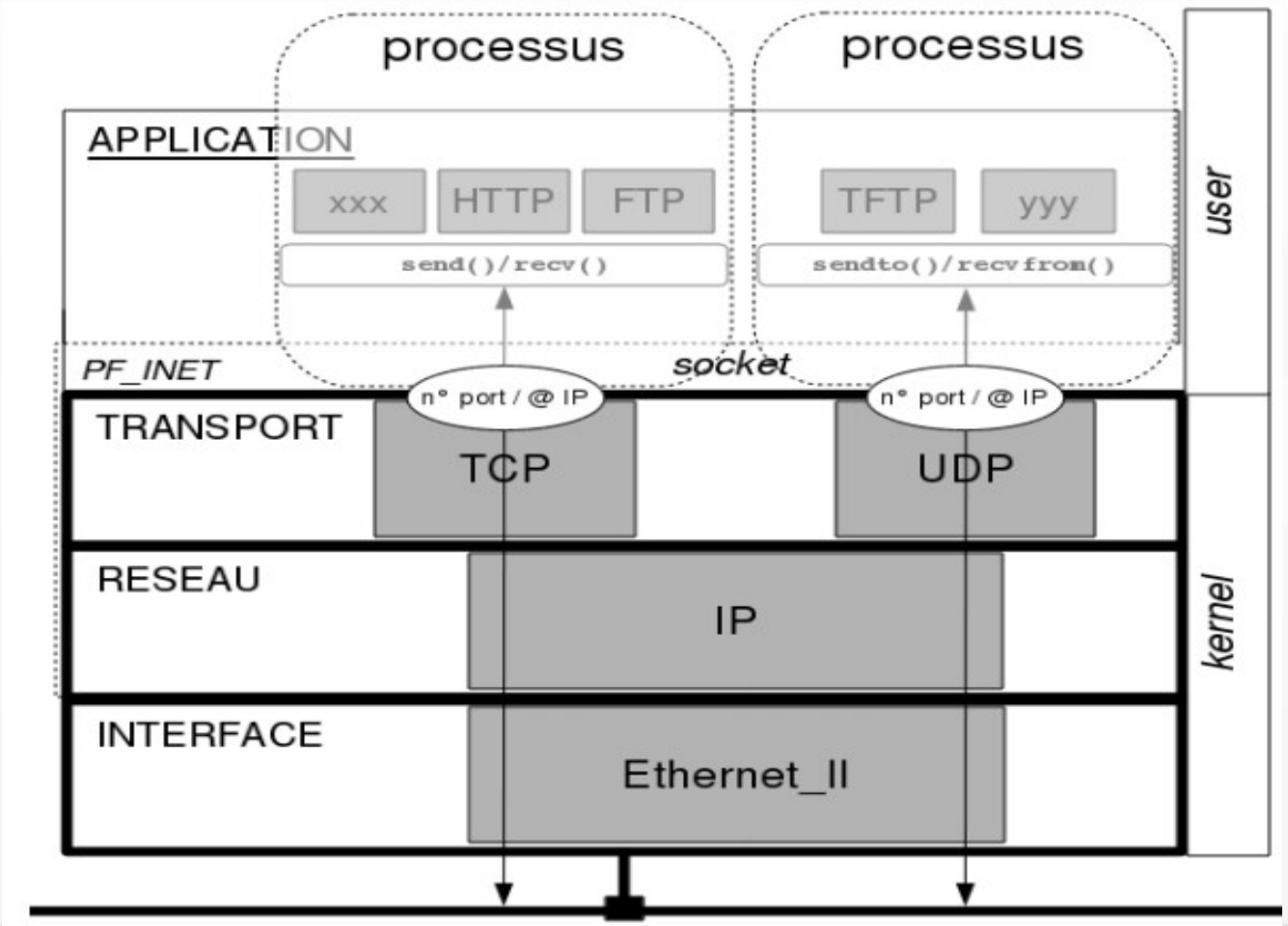
Les outils du programmeur I

- Le développeur utilisera donc concrètement une interface pour programmer **une application TCP/IP, UDP** grâce par exemple :
  - ✓ à l'API **Socket BSD** sous Unix/Linux ou
  - ✓ à l'API **WinSocket** sous Microsoft Windows
- Les pages **man** principales sous Unix/Linux concernant la programmation réseau contient les fonctions suivantes :

- **Socket :** interface de programmation des sockets
- **Packet :** interface par paquet au niveau périphérique
- **Raw :** sockets brutes (raw) IPv4 sous Linux
- **ip :** implémentation Linux du protocole IPv4
- **udp :** protocole UDP pour IPv4
- **tcp:** protocole TCP

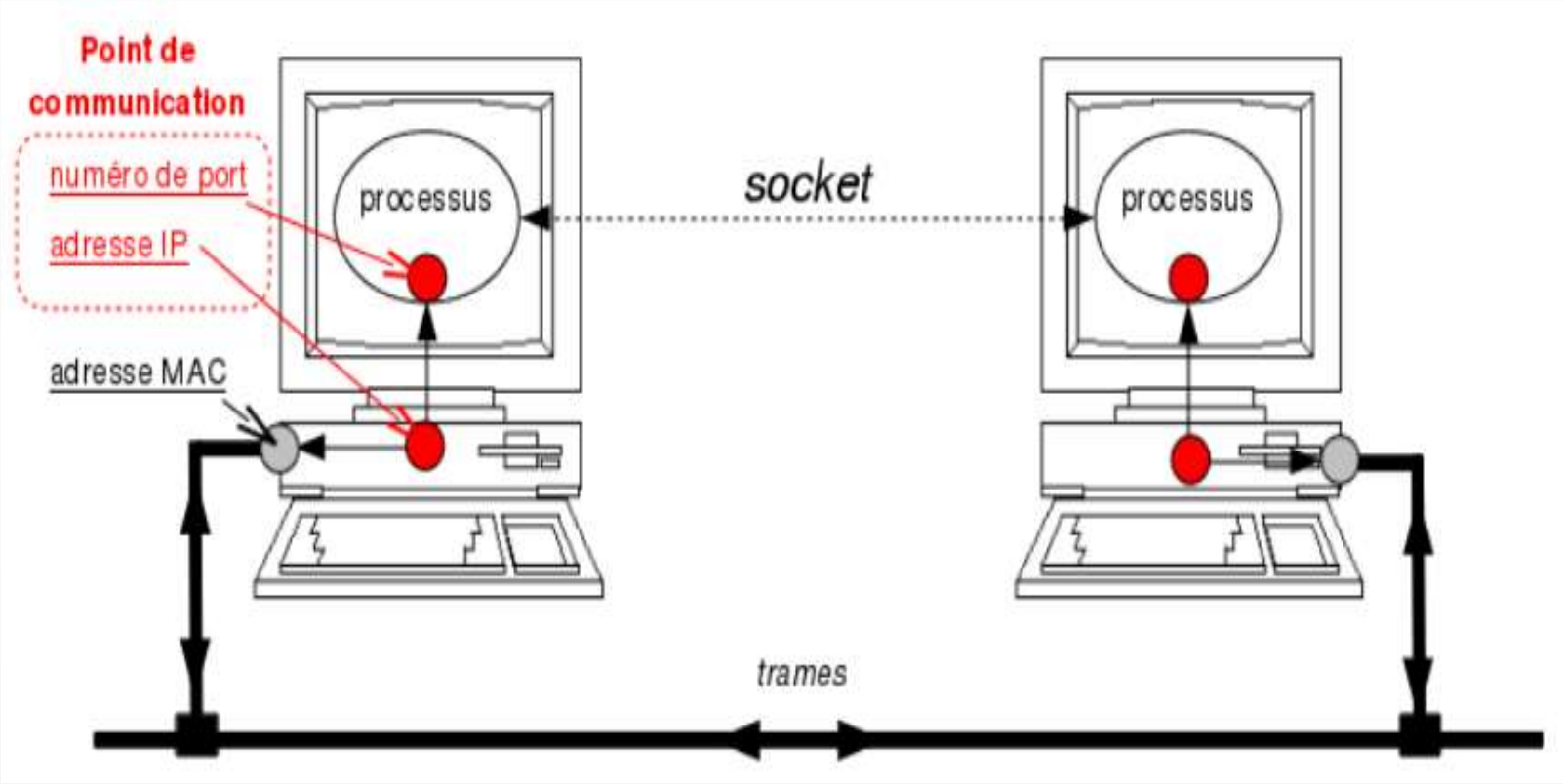
Modèle à couches I

- Un socket est communément représenté comme un point d'entrée initial au niveau de la couche TRANSPORT dans le modèle OSI

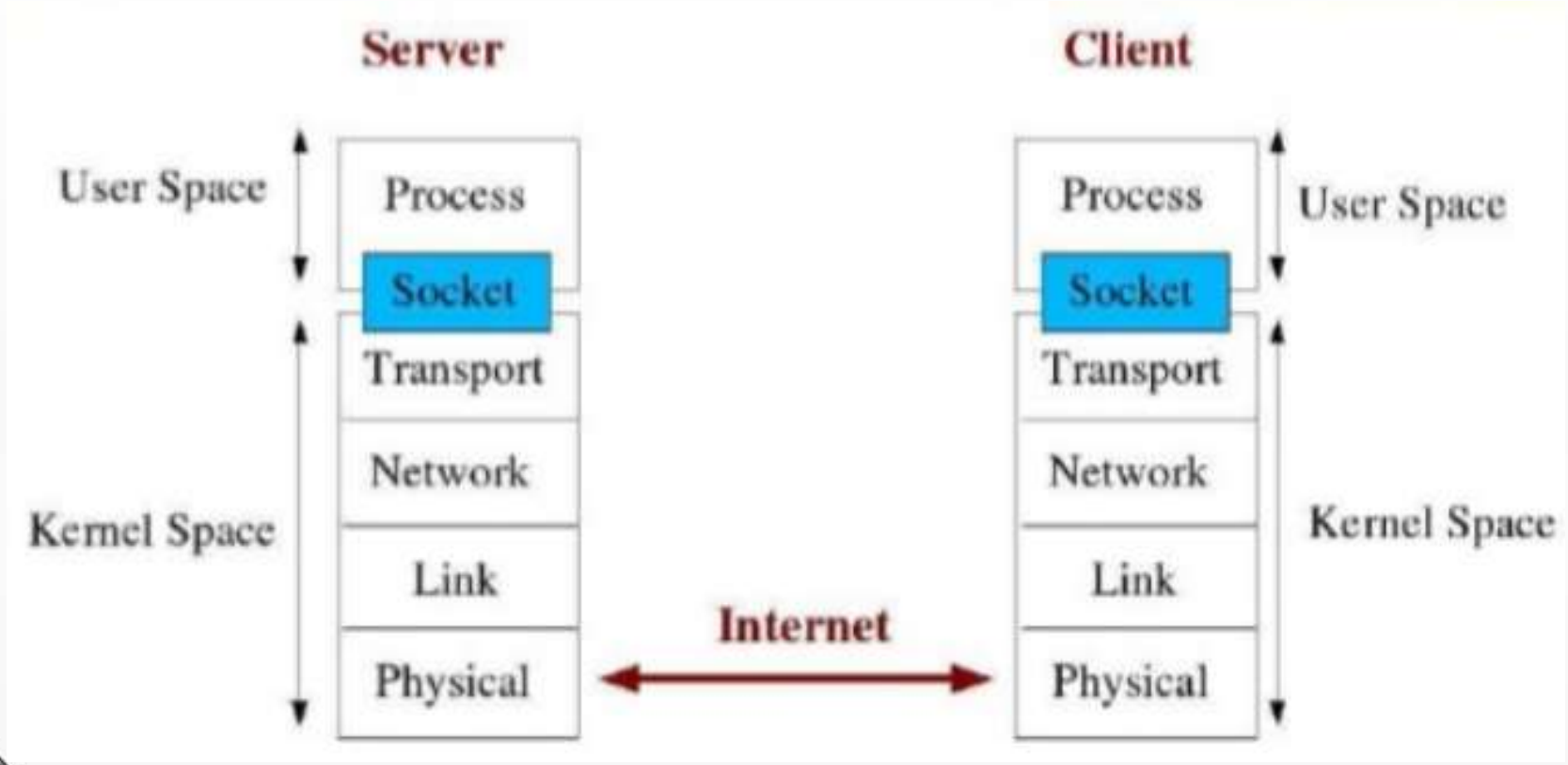


Modèle à couches II

- Un socket est une interface de communication identifiée par une adresse IP et un numéro de port pour le modèle TCP/IP



La description du socket



Création d'un socket

❖Pour dialoguer : chaque processus devra préalablement créer un **socket de communication** en utilisant l'appel de la fonction **socket()** et en indiquant :

a.le **domaine** de communication : ceci sélectionne la famille de protocole à employer.

b.le **type** de socket à utiliser pour le dialogue (mode connecté **TCP/IP**, non connecté **UDP….**).

c.le **protocole** à utiliser sur le socket en fonction du **type** et du **domaine de communication** sélectionnés.

=> La fonction **socket()** renvoie : soit un descripteur sur la socket créée **en cas de réussite**

soit -1 **en cas d'échec.**

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

int socket(int domain, int type, int protocol);
```

Caractéristiques d'un socket

- Le **domaine** sélectionne la famille de protocole à employer : Chaque famille possède son adressage. Par exemple : pour les protocoles => Internet IPv4, on utilisera le domaine **PF_INET** ou **AF_INET** et **AF_INET6** pour le protocole IPv6.
- le **type** de dialogue : Pour **PF_INET**, on aura par exemple le choix entre :

  SOCK_STREAM (qui correspond à un mode connecté donc TCP par défaut), **SOCK_DGRAM** (qui correspond à un mode non connecté donc UDP) ou **SOCK_RAW** (qui permet un accès direct aux protocoles de la couche Réseau comme IP, ICMP, ...).
- Le **numéro de protocole** dépendra du **domaine de communication** et du **type de socket.** Normalement, il n'y a qu'un seul protocole par type de socket pour une famille donnée : la valeur 0 désignera le protocole par défaut (**SOCK_STREAM => TCP et SOCK_DGRAM => UDP**). Il est nécessaire de le spécifier (c'est la cas pour **SOCK_RAW** où il faudra préciser le protocole à utiliser : **IPPROTO_IP, IPPROTO_ICMP**, ...).

Caractéristiques d'un socket

domain[2]	range	transport	address format	address C struct
AF_UNIX	same host	kernel	pathname	sockaddr_un
AF_INET	any host w/ IPv4 connectivity	IPv4 stack	32-bit IPv4 address + 16-bit port number	sockaddr_in
AF_INET6	any host w/ IPv6 connectivity	IPv6 stack	128-bit IPv6 address + 16-bit port number	sockaddr_in6

Caractéristiques d'un socket

- **Exemple :**

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

int socket_tcp, socket_udp, socket_icmp; //descripteurs de sockets

// Un socket en mode connecte
socket_tcp = socket(PF_INET, SOCK_STREAM, 0); //par defaut TCP

// Un socket en mode non connecte
socket_udp = socket(PF_INET, SOCK_DGRAM, 0); //par defaut UDP

// Un socket en mode raw
socket_icmp = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP); //on choisit ICMP
```

Adressage du point de communication

L'interface socket propose une structure d'adresse générique :

```
struct sockaddr
{
  unsigned short int sa_family; //au choix
  unsigned char sa_data[14];  //en fonction de la famille
};
```

⇒ Pour le domaine **PF_INET (IPv4),** l'adressage du socket sera réalisée par une structure **sockaddr_in**:

```
struct in_addr { unsigned int s_addr; }; // une adresse Ipv4 (32 bits)

struct sockaddr_in
{
  unsigned short int sin_family; // <- PF_INET (IPv4)
  unsigned short int sin_port; // <- numero de port
  struct in_addr sin_addr; // <- adresse IPv4
  unsigned char sin_zero[8];  // ajustement pour etre compatible avec sockaddr
};
```

Network byte order

- L'ordre des octets du réseau (network) est en fait big-endian. Il est donc possible qu'il soit différent de celui de la machine (host). Dans tous les cas, il est important de toujours convertir les informations à envoyer dans l'ordre du réseau et les lire dans l'ordre de la machine.

Pour les numéros de port sur 16 bits **(short int)** du champ **sin_port**, on utilisera :

- ✓ **htons() :** pour convertir le numéro de port depuis l'ordre des octets de l'hôte vers celui du réseau.
- ✓ **ntohs() :** pour convertir le numéro de port depuis l'ordre des octets du réseau vers celui de l'hôte.

Adresse IP

Dans l'en-tête d'un paquet IP, une adresse IPv4 est codée sur 32 bits **(unsigned int)** dans le champ **s_addr**. Comme on manipule souvent cette adresse avec chaîne de caractères en notation décimale pointée, il existe des fonctions de conversion :

- ✓ **inet_aton()** : convertit une adresse IPv4 en décimal pointé vers sa forme binaire 32 bits dans l'ordre d'octet du réseau.
- ✓ **inet_ntoa() :** convertit une adresse IPv4 sous sa forme binaire 32 bits dans l'ordre d'octet du réseau vers une chaîne de caractères en décimal pointé.

=> Voir aussi les fonctions de traduction d'adresses et de services réseau **getaddrinfo()** et **getnameinfo()** si on désire utiliser par exemple un résolveur DNS.

Adresse IP

- **Exemple :**

```
struct sockaddr_in adresseDistante;
socklen_t longueurAdresse;

// Obtient la longueur en octets de la structure sockaddr_in
longueurAdresse = sizeof(adresseDistante);

// Initialise la structure sockaddr_in
memset(&adresseDistante, 0x00, longueurAdresse);

// Renseigne la structure sockaddr_in avec les informations du serveur distant
adresseDistante.sin_family = PF_INET;

// On choisit le numero de port d'ecoute du serveur
adresseDistante.sin_port = htons(5000); // = IPPORT_USERRESERVED

// On choisit l'adresse IPv4 du serveur
inet_aton("192.168.52.2", &adresseDistante.sin_addr);
```

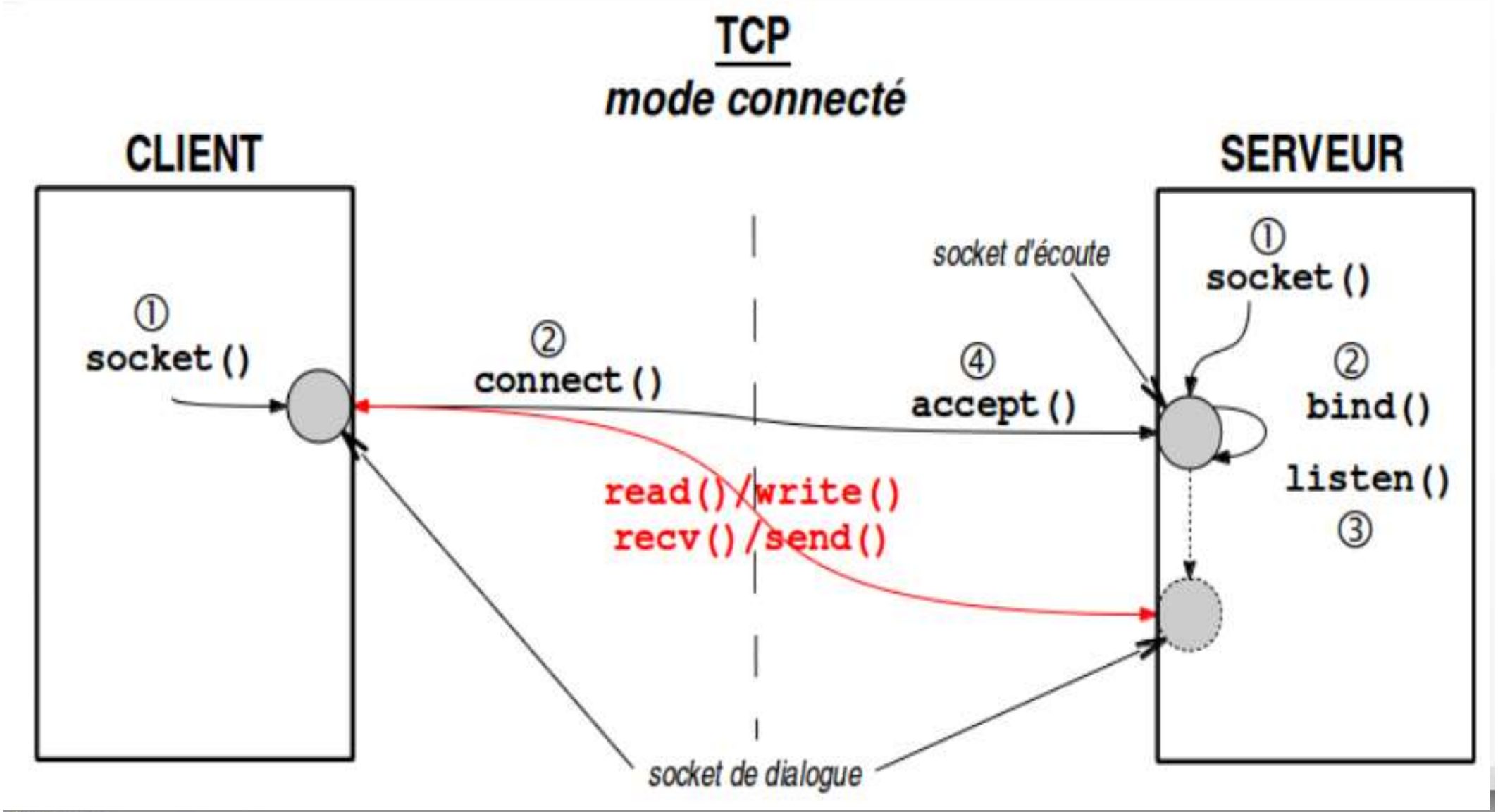
Communication TCP

Dans une communication TCP, une connexion doit être établie entre le client et le serveur.

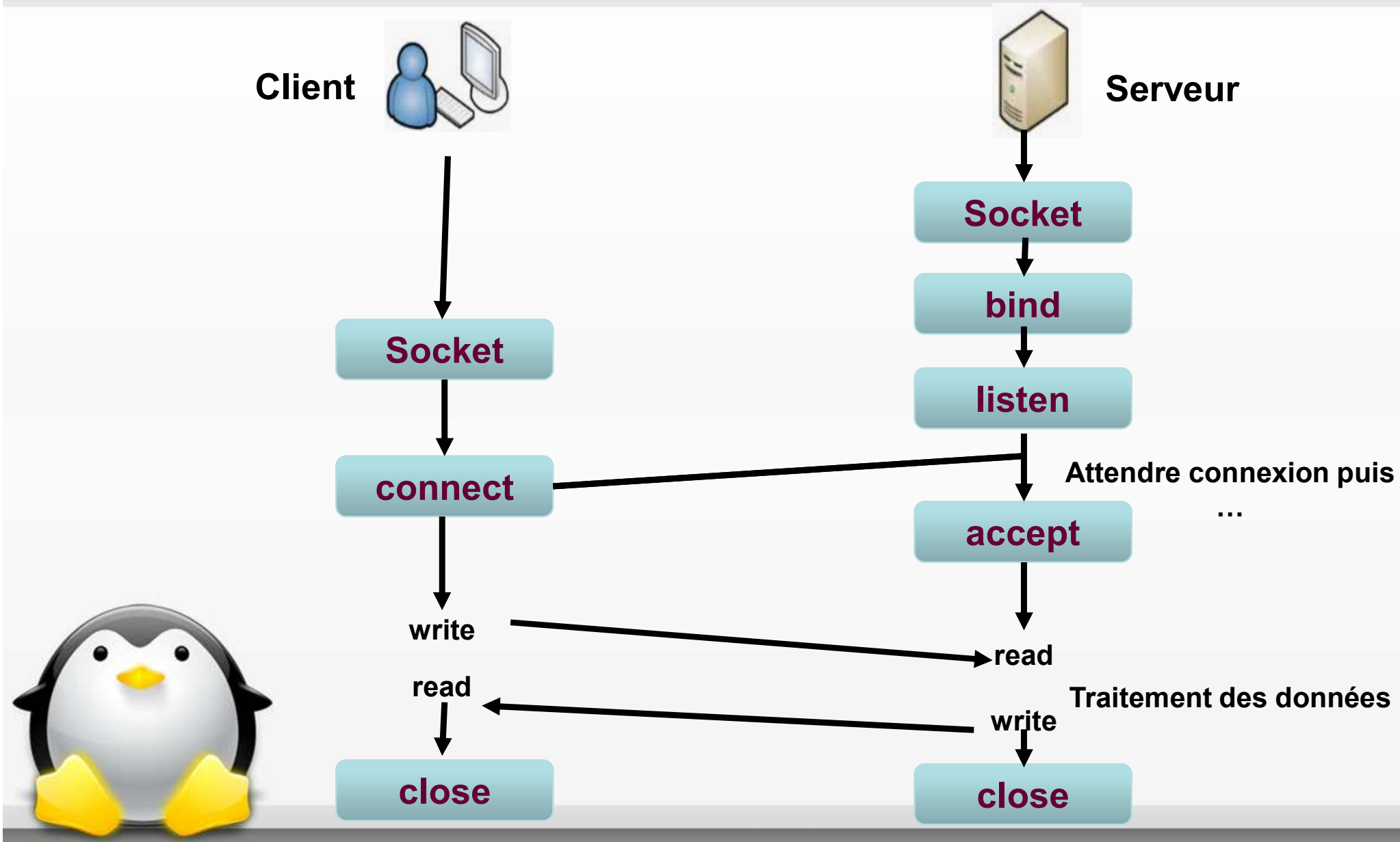
- ✓ La fonction **connect()** permet à un client de demander la connexion à un serveur.
- ✓ Le serveur utilisera préalablement : **la fonction bind() pour définir l'adresse de sa socket et la fonction listen() pour informer l'OS** qu'il est prêt à recevoir des demandes de connexion entrante.
- ✓ La fonction **accept()** permet à un serveur d'accepter une connexion.
- ✓ Une fois la connexion établie, les fonctions **send()** et **recv()** (ou tout simplement **write()** et **read()**) servent respectivement à envoyer et à recevoir des informations entre les deux processus.
- ✓ Une fonction **close()** (ou **shutdown()**) permet de terminer la connexion.

⇒ L'appel **accept()** du serveur crée et retourne une nouvelle socket pour le dialogue.

Communication TCP



Communication TCP

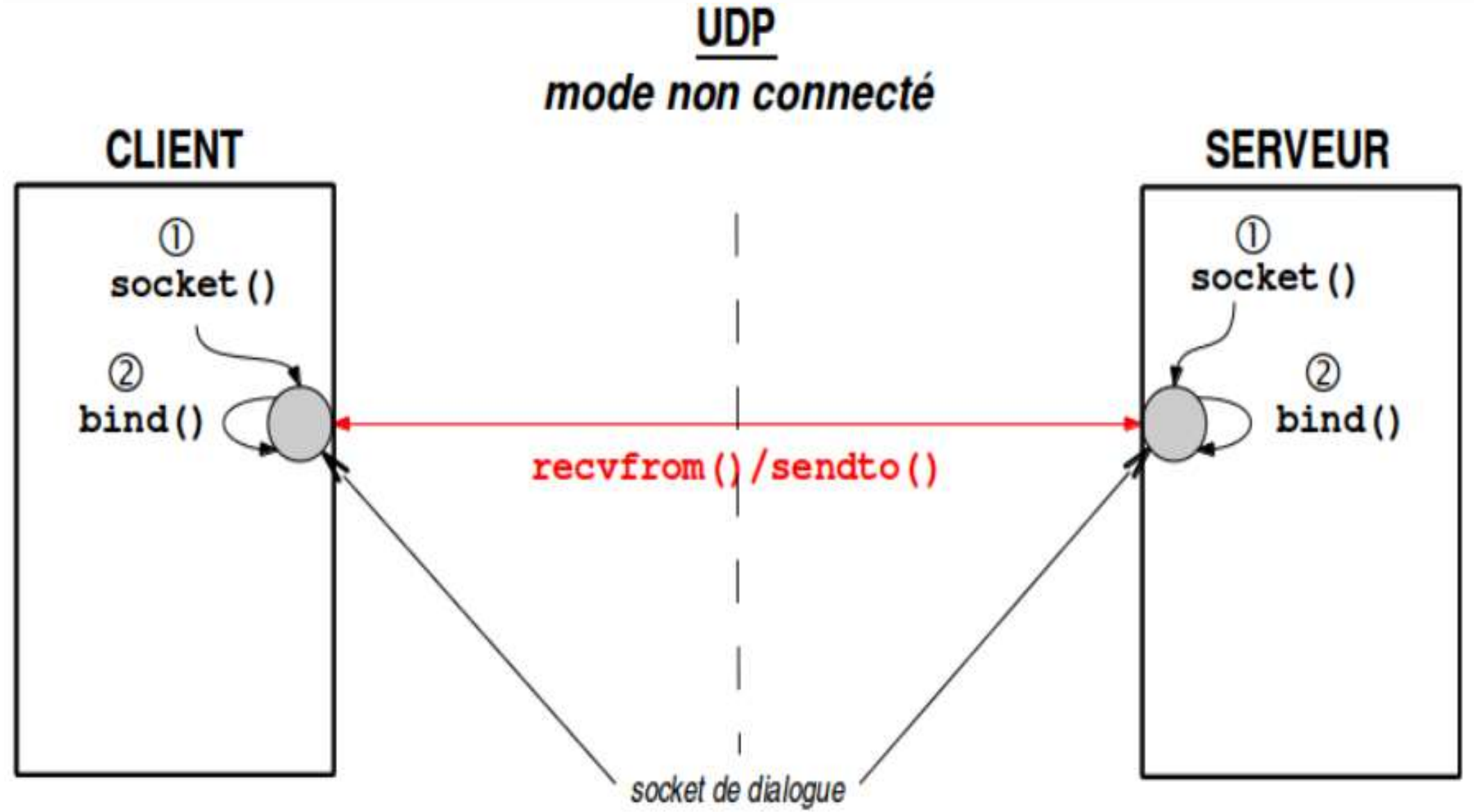


Communication UDP

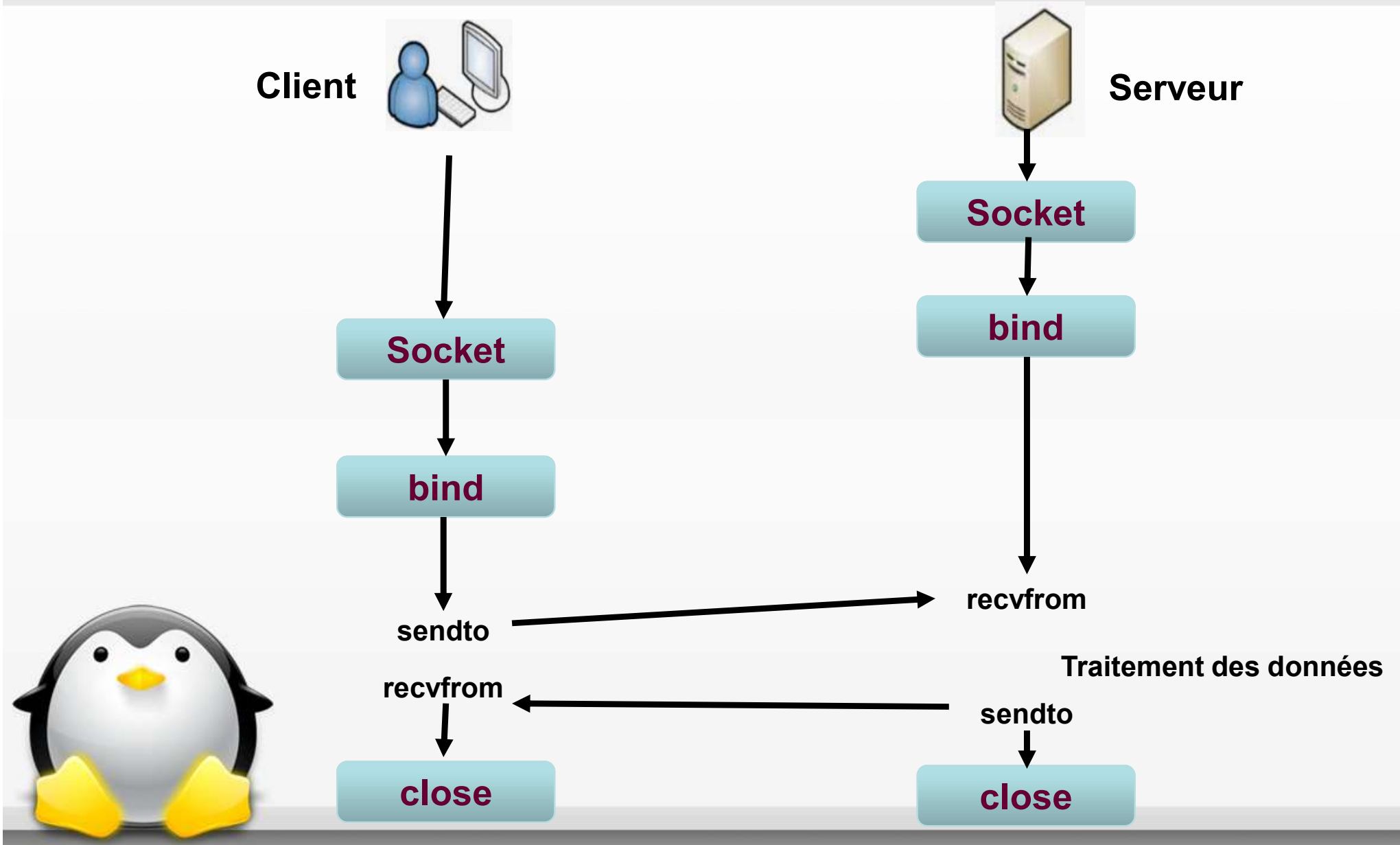
Dans une communication bidirectionnelle **UDP**, il n'y a pas de connexion entre le client et le serveur. Le client reste le processus à l'initiative de l'échange et de la demande de service.

- La fonction **bind()** permet de définir l'adresse locale de la socket.
- Les fonctions d'échanges de données sur une socket **UDP** sont **recvfrom()** et **sendto()** qui permettent respectivement la réception et l'envoi d'octets sur un descripteur de socket en mode non-connecté.
- La fonction **close()** permet ici de libérer la ressource socket localement.

Communication UDP



Communication UDP



Mode bloquant et non bloquant I

- Certains appels systèmes utilisés ici sont « **bloquants** » :

=> C'est-à-dire que le processus s'endort jusqu'à ce dont il a besoin soit disponible et le réveille.

=> C'est le cas des appels **accept()** et des fonctions de réception de données **read(), recv() et recvfrom().**

- C'est le choix par défaut fait par l'OS au moment de la création de la socket.
- Il est possible de placer le socket en mode non bloquant avec l'appel **fcntl().**
- En déclarant une socket **non bloquante**, il faudra alors l'interroger périodiquement. **Si vous essayez de lire sur une socket non bloquante et qu'il n'y a pas de donnée à lire, la fonction retournera -1.**

Cas du serveur multi-clients

- Un client se connecte sur la socket d'écoute.

⇒ **L'appel accept() va retourner une nouvelle socket connectée au client qui servira de socket de dialogue.**

⇒ **La socket d'écoute reste inchangée et peut donc servir à accepter des nouvelles connexions.**

- Le principe est simple mais un problème apparaît pour le serveur :

Comment dialoguer avec le client connecté et continuer à attendre des nouvelles connexions ?

- ✓ **Solution n°1 :** Le multiplexage synchrone d'E/S. On utilise les appels **select()** ou **poll()** pour surveiller et attendre un événement concernant une des sockets.
- ✓ **Solution n°2 :** Le multi-tâche. On crée un processus fils (avec l'appel fork()) qui traite lui-même le dialogue et le processus père qui continue l'attente des demandes de connexion. Si le temps de création des processus fils devient une charge importante pour le système, on pourra envisager les solutions suivantes : création anticipée des processus, utilisation des **(processus légers) threads**.

Programmation Shell

C’est quoi le Shell ?

- Supposons **un mini-langage de programmation intégré à Linux** qui n'est pas un langage aussi complet comme par exemple le C, le C++ ou le Java, mais cela permet d'automatiser la plupart de vos tâches : **sauvegarde des données, surveillance de la charge de votre machine, etc…..**

- Vous pouvez faire tout cela en créant un programme en C par exemple. **Le gros avantage du langage shell est d’être totalement intégré à Linux** : **il n'y a rien à installer, rien à compiler.** Et surtout : vous avez très peu de nouvelles choses à apprendre. En effet, toutes les commandes que l'on utilise dans les scripts shell sont des commandes du système que vous connaissez déjà : ls, cut, grep, sort…

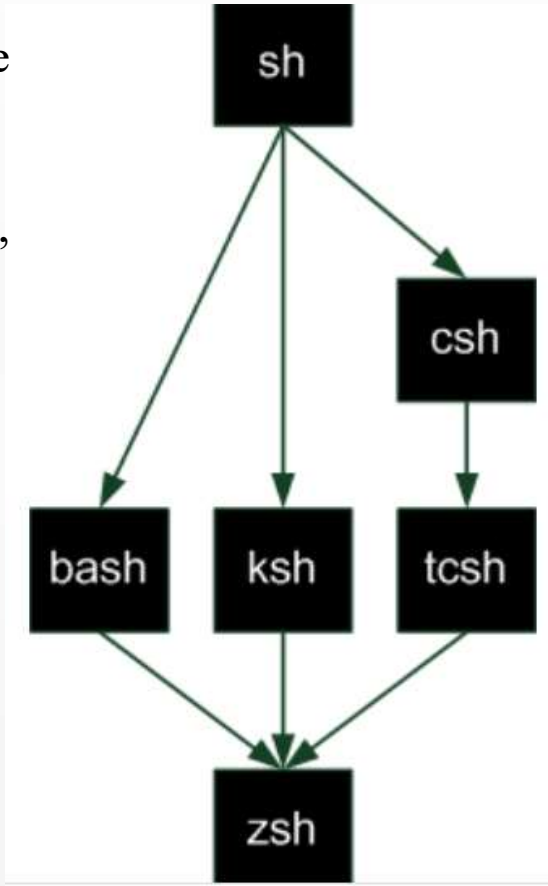
C’est quoi le Shell ?

- L’interprète de commandes **(shell)** permet d’interagir avec le système; et exécute des commandes **(modification / consultation de l’état du système)**
- Il y’a deux environnements très différents disponibles sous Linux :
  - ✓ **L'environnement console**
  - ✓ **L'environnement graphique.**
- La plupart du temps, sur sa machine, on a tendance à utiliser l'environnement graphique, qui est plus intuitif. Mais, la console est aussi un allié très puissant qui permet d'effectuer des actions habituellement difficiles à réaliser dans un environnement graphique.

Types de Shell

Voici les noms de quelques-uns des principaux **shells** qui existent :

- ✓ **sh :** Bourne Shell. L'ancêtre de tous les shells, et il est installé sur tous les OS basés sur Unix. Il est néanmoins pauvre en fonctionnalités par rapport aux autres shells.
- ✓ **bash :** Bourne Again Shell. Une amélioration du Bourne Shell, disponible par défaut sous Linux et Mac OS X.
- ✓ **ksh :** Korn Shell. Un shell puissant assez présent sur les Unix propriétaires, mais aussi disponible en version libre, compatible avec bash.
- ✓ **csh :** C Shell. Un shell utilisant une syntaxe proche du langage C.
- ✓ **tcsh :** Tenex C Shell. Amélioration du C Shell.
- ✓ **zsh :** Z Shell. Shell assez récent reprenant les meilleures idées de **bash, ksh et tcsh.**



Description de Shell

- Autocompléter une commande ou un nom de fichier lorsque vous appuyez sur Tab (figure suivante) ;
- Gérer les processus (envoi en arrière-plan, mise en pause avec Ctrl + Z…) ;
- Rediriger et chaîner les commandes (les fameux symboles >, <, |, etc.) ;
- Définir des alias (par exemple ll signifie chez moi ls -lArth).

⇒ Bref, le shell fournit toutes les fonctionnalités de base pour pouvoir lancer des commandes.

Programmation de Shell

➢ Installation de Shell :

Vous devriez avoir **sh** et **bash** installés sur votre système. Si vous voulez essayer **un autre shell**, comme **ksh** par exemple, vous pouvez le télécharger comme n'importe quel paquet :

```
# apt-get install ksh
```

Une fois installé, il faut demander à l'utiliser pour votre compte utilisateur. Pour cela, tapez :

```
$ chsh
```

=> **chsh signifie Change Shell.**

On vous demandera où se trouve le programme qui gère **le shell**. Vous devrez indiquer / **bin/ksh pour ksh, /bin/sh pour sh, /bin/bash pour bash. On va se concentrer plus sur le bash.**

Programmation de Shell

➢ Création de script Bash :

Nous allons commencer par étudier un premier script **bash** tout simple via lequel on pourra voir les bases de la création d'un script et comment celui-ci s'exécute, pour cela on crée un nouveau fichier pour notre script. Le plus simple est d'ouvrir **Vim** en lui donnant le nom du nouveau fichier à créer :

```
$ vim essai.sh
```

Si essai.sh n'existe pas, il sera créé (ce qui sera le cas ici). L'extension .**sh** indique le script **shell,** Certains scripts **shell** n'ont d'ailleurs pas d'extension du tout.

=> Le script essai doit contenir une syntaxe du programmation du shell **bash** qui est le plus répandu sous Linux et plus complet que **sh.** Nous indiquons où se trouve le programme **bash**

```
#!/bin/bash
```

La ligne du sha-bang #! permet donc de « charger » le bon shell avant l'exécution du script, vous devrez la mettre au tout début de chacun de vos scripts.

Programmation de Shell

➢ Exécution de commandes :

Après le sha-bang **#!**, vous pouvez commencer à coder et lancer les commandes dans le script **essai.sh**

```
#!/bin/bash

ls
```

Pour les commentaires, vous pouvez les ajouter dans le script. Ce sont des lignes qui ne seront pas exécutées mais qui permettent d'expliquer ce que fait votre script.

Tous les commentaires commencent par un #. Par exemple :

```
#!/bin/bash

# Affichage de la liste des fichiers
ls
```

Programmation de Shell

➢ Exécution du script bash :

Pour enregistrer votre fichier et fermez votre éditeur. Sous **Vim**, il suffit de taper **:wq** ou encore **:x. Le script s'exécute maintenant comme n'importe quel programme, en tapant « ./ » devant le nom du script :**

```
$ ./essai.sh
essai.sh
```

« Ce script fait juste un ls, il affiche donc la liste des fichiers présents dans le répertoire »

Pour exécuter un script, il faut que le fichier ait le droit « exécutable ». Le plus simple pour donner ce droit est d'écrire :

```
$ chmod +x essai.sh
```

Programmation de Shell

➢ Les variables en bash :

a) Déclaration d'une variable :

- ✓ La variable a pour **nom message** ;
- ✓ Et pour **valeur « Bonjour tout le monde » :**

```
$ ./essai.sh
essai.sh
```

- Ne mettez pas d'espaces autour du symbole égal « = »

- Si vous voulez insérer une apostrophe dans la valeur de la variable, il faut la faire précéder avec un antislash \.

```
message='Bonjour c\'est moi'
```

Programmation de Shell

➢ Les variables en bash :

Comme n'importe quel langage de programmation, on trouve en **bash** ce que l'on appelle **des variables**. Ces variables nous permettent **de stocker temporairement des informations en mémoire. Ce qui est la base de la programmation.**

a) Déclaration d'une variable :

On peut créer un nouveau script que nous appellerons variables.sh par l'instruction suivante

```
$ vim variables.sh
```

La première ligne de tous nos scripts doit indiquer quel shell est utilisé, et commence par cette phrase

```
#!/bin/bash
```

=> ce qui indique qu'on va programmer avec le bash

On définit une variable. Toute variable possède un nom et une valeur :

```
message='Bonjour tout le monde'
```

Programmation de Shell

➢ Les variables en bash :

a) Déclaration d'une variable :

✓ Si on reprend notre script. Il devrait à présent ressembler à ceci :

```
#!/bin/bash

message='Bonjour tout le monde'
```

- Si on l'exécute pour voir ce qui se passe (après avoir modifié les droits pour le rendre exécutable, bien sûr) => rien ne se passe :

```
$ ./variables.sh
$
```

Ce script met en mémoire le message « Bonjour tout le monde », et c'est tout ! Rien ne s'affiche à l'écran ! Pour afficher une variable, il va falloir utiliser la commande « echo »

Programmation de Shell

➢ Les variables en bash :

b) Affichage d'une variable :

✓ La commande « **echo** » affiche dans la console le message demandé. Un exemple :

```
$ echo Salut tout le monde
Salut tout le monde
```

- **la commande « echo » affiche dans la console tous les paramètres qu'elle reçoit. Ici, nous avons envoyé quatre paramètres : Salut ; tout ; le ; monde.**
- **Chacun des mots était considéré comme un paramètre affiché. Si vous mettez des guillemets autour de votre message, celui-ci sera considéré comme étant un seul et même paramètre (le résultat sera visuellement le même) :**

```
$ echo "Salut tout le monde"
Salut tout le monde
```

```
$ echo -e "Message\nAutre ligne"
Message
Autre ligne
```

Programmation de Shell

➢ Les variables en bash :

b) Affichage d'une variable :

✓Pour afficher une variable, on peut de nouveau utiliser son nom précédé du symbole dollar $:

```
#!/bin/bash

message='Bonjour tout le monde'
echo $message
```

- **Mais, supposons que l'on veuille afficher à la fois du texte et la variable. Nous serions tentés d'écrire :**

```
#!/bin/bash

message='Bonjour tout le monde'
echo 'Le message est : $message'
```

le résultat est :

```
Le message est : $message
```

```
message='Bonjour tout le monde'
echo "Le message est : $message"
Le message est : Bonjour tout le monde
```

Programmation de Shell

➢ Les variables en bash :

c) Demande d'une saisie :

✓ Vous pouvez demander à l'utilisateur de saisir du texte avec la commande **read** :

✓ Adaptons ce script qui nous demande de saisir un nom puis qu'il nous l'affiche :

```
#!/bin/bash

read nom
echo "Bonjour $nom !"
```

- **Lorsque vous lancez ce script, rien ne s'affiche, mais vous pouvez taper du texte (Mathieu, par exemple) :**

```
Mathieu
Bonjour Mathieu !
```