III LA COMMUNICATION INTER-PROCESSUS

3.1 Introduction

Les processus ne sont pas des entités indépendantes. Ils doivent partager les ressources de l'ordinateur. Dans certains cas, ils doivent communiquer entre eux pour se synchroniser ou pour communiquer de l'information.

La communication interprocessus (interprocess communication, IPC) consiste à transférer des données entre les processus. Par exemple, un navigateur Internet peut demander une page à un serveur, qui envoie alors les données HTML.

La communication interprocessus peut se faire de différentes manières : mémoire partagée, mémoire mappée, tubes, files et socket.

- La mémoire partagée permet aux processus de communiquer simplement en lisant ou écrivant dans un emplacement mémoire prédéfini.
- La mémoire mappée est similaire à la mémoire partagée, excepté qu'elle est associée à un fichier.
- Les tubes permettent une communication séquentielle d'un processus à l'autre.
- Les files FIFO sont similaires aux tubes excepté que des processus sans lien peuvent communiquer car le tube reçoit un nom dans le système de fichiers.
- Les signaux
- Les sockets permettent la communication entre des processus sans lien, pouvant se trouver sur des machines distinctes.

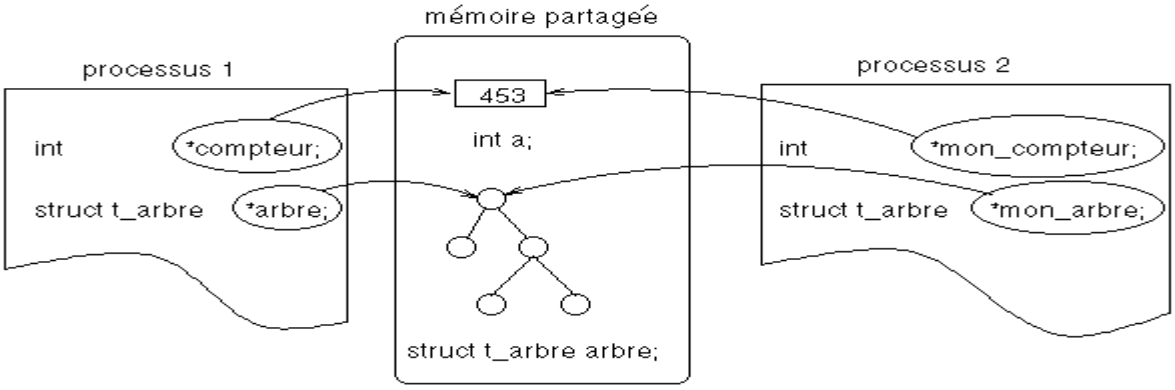
3.2 Mémoire Partagée

Une des méthodes de communication interprocessus les plus simples est d'utiliser la mémoire partagée. La mémoire partagée permet à deux processus ou plus d'accéder à la même zone mémoire comme s'ils avaient leurs pointeurs dirigés vers le même espace mémoire. Lorsqu'un processus modifie la mémoire, tous les autres processus voient la modification.

La mémoire partagée est la forme de communication interprocessus la plus rapide car tous les processus partagent la même mémoire. Elle évite également les copies de données inutiles.

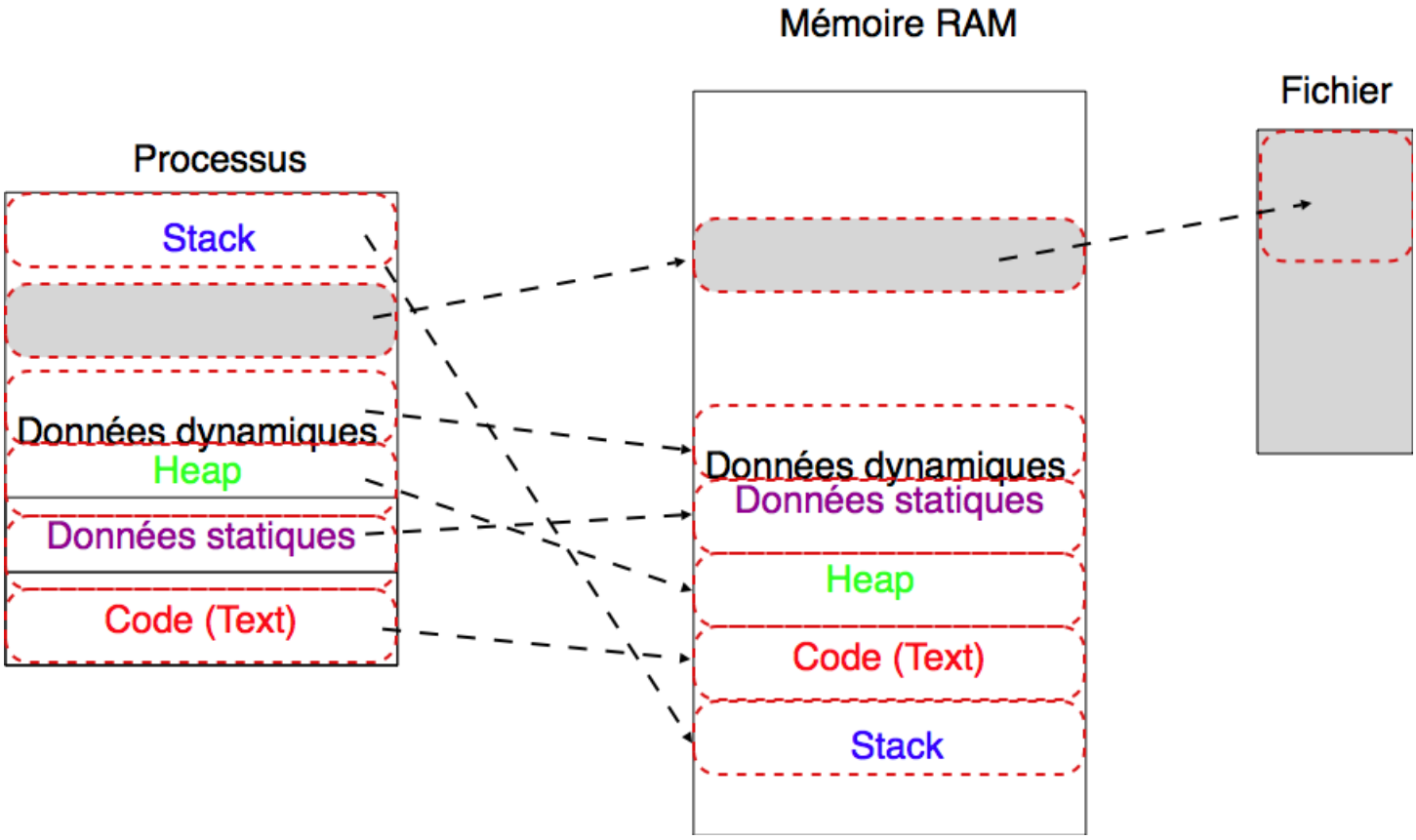
Pour utiliser un segment de mémoire partagée, un processus doit allouer le segment. Puis, chaque processus désirant accéder au segment doit l'attacher. Après avoir fini d'utiliser le segment, chaque processus le détache. À un moment ou à un autre, un processus doit libérer le segment.

Sous Unix, un processus alloue un segment de mémoire partagée en utilisant *shmget* (« SHared Memory GET », obtention de mémoire partagée). Son premier paramètre est une clé entière qui indique le segment à créer. Le second paramètre indique le nombre d'octets du segment. Le troisième paramètre est un ensemble d'indicateurs binaires décrivant les options demandées : s'agit-il d'un nouveau segment ou un segment existant qu'il faut attacher, permissions de lecture/écriture, … etc.



3.3 Mémoire Mappée

La mémoire mappée permet à différents processus de communiquer via un fichier partagé.
Lorsqu'un processus Unix veut lire ou écrire des données dans un fichier, il utilise en général les appels systèmes *open*, *read*, *write* et *close* directement. Ce n'est pas la seule façon pour accéder à des données sur un dispositif de stockage. Grâce à la mémoire virtuelle, il est possible de placer le contenu d'un fichier ou d'une partie de fichier dans une zone de la mémoire du processus. Cette opération peut être effectuée en utilisant l'appel système *mmap*. Cet appel système permet de rendre des pages d'un fichier accessibles à travers la table des pages du processus comme illustré dans la figure ci-dessous.



3.4 Tubes

Les tubes sans nom sont des liaisons unidirectionnelles de communication. La taille maximale d'un tube sans nom varie d'une version à une autre d'Unix, mais elle est approximativement égale à 4 Ko.
Les tubes sans nom peuvent être créés par le shell ou par l'appel système pipe().
Les tubes sans nom du shell sont créés par l'opérateur binaire « | » qui dirige la sortie standard d'un processus vers l'entrée standard d'un autre processus. Les tubes de communication du shell sont supportés par toutes les versions d'Unix.
Exemple : La commande shell suivante crée deux processus qui s'exécutent en parallèle et qui sont reliés par un tube de communication pipe. Elle détermine le nombre d'utilisateurs connectés au système en appelant *who*, puis en comptant les lignes avec *wc* :

who | wc -l

Le premier processus réalise la commande *who*. Le second processus exécute la commande *wc -l*. Les résultats récupérés sur la sortie standard du premier processus sont dirigés vers l'entrée standard du deuxième processus via le tube de communication qui les relie.

Le processus réalisant la commande *who* ajoute dans le tube une ligne d'information par utilisateur du système. Le processus réalisant la commande *wc -l* récupère ces lignes d'information pour en calculer le nombre total. Le résultat est affiché à l'écran. Les deux processus s'exécutent en parallèle, les sorties du premier processus sont stockées sur le tube de communication. Lorsque le tube devient plein, le premier processus est suspendu jusqu'à ce qu'il y ait libération de l'espace nécessaire pour stocker une ligne d'information. De façon similaire, lorsque le tube devient vide, le second processus est suspendu jusqu'à ce qu'il y ait au moins une ligne d'information sur le tube.

- **Création d'un tube sans nom**

Un tube de communication sans nom est créé par l'appel système *pipe*, auquel on passe un tableau de deux entiers :

int pipe(int descripteur[2]);

Au retour de l'appel système *pipe*(), un tube aura été créé, et les deux positions du tableau passé en paramètre contiennent deux descripteurs de fichiers.

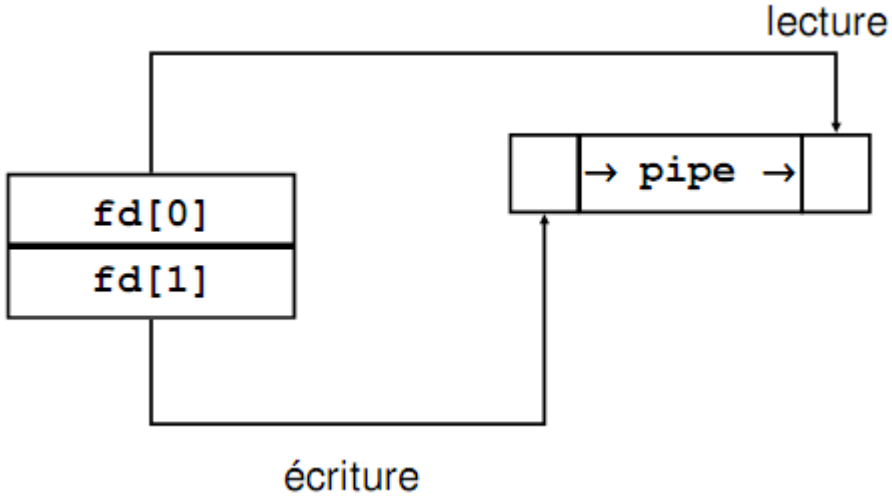
Un descripteur comme une valeur entière que le système d'exploitation utilise pour accéder à un fichier. Dans le cas du tube on a besoin de deux descripteurs : le descripteur pour les lectures du tube et le descripteur pour les écritures dans le tube.

L'accès au tube se fait via les descripteurs. Le descripteur de l'accès en lecture se retrouvera à la position 0 du tableau passé en paramètre, alors que le descripteur de l'accès en écriture se retrouvera à la position 1. Seul le processus créateur du tube et ses descendants (ses fils) peuvent accéder au tube. Si le système ne peut pas créer de tube pour manque d'espace, l'appel système pipe() retourne la valeur -1, sinon il retourne la valeur 0.

Exemple 1. Les déclarations suivantes créent le tube montré sur la figure suivante :

int fd[2];
pipe(fd);

Les tubes sans nom sont, en général, utilisés pour la communication entre un processus père et ses processus fils, avec un d'entre eux qui écrit sur le tube, appelé processus écrivain, et l'autre qui lit à partir du tube, appelé processus lecteur. La séquence d'événements pour une telle communication est comme suit :



- Le processus père crée un tube de communication sans nom en utilisant l'appel système pipe()
- Le processus père crée un ou plusieurs fils en utilisant l'appel système fork()
- Le processus écrivain ferme l'accès en lecture du tube
- De même, le processus lecteur ferme l'accès en écriture du tube
- Les processus communiquent en utilisant les appels système write() et read()
- Chaque processus ferme son accès au tube lorsqu'il veut mettre fin à la communication via le tube.

Exemple 2. Dans le programme suivant, le processus père crée un tube de communication pour communiquer avec son processus fils. La communication est unidirectionnelle du processus fils vers le processus père.

```
# include < sys/types.h> // types
# include < unistd.h> // fork , pipe , read , write , close
# include < stdio.h>
# define R 0
# define W 1
int main ( )
{
int fd [ 2 ] ;
char message [100] ; // pour récupérer un message
int nboctets ;
char phrase = "message envoye au pere par l e f i l s " ;
pipe (fd) ; // création d' un tube sans nom
if ( fork () = = 0 ) // creation d' un processus fils
{
// Le fils ferme le descripteur non utilise de lecture
close ( fd[R] ) ;
// dépôt dans le tube du message
write ( fd [W] , phrase , strlen( phrase ) + 1 ) ;
// fermeture du descripteur d ' ecriture
close ( fd [W] ) ;
}
else
{
// Le père ferme l e descripteur non utilise d'écriture
close ( fd [W] ) ;
// extraction du message du tube
nboctets = read ( fd [R] , message , 100 ) ;
print f ( " Lecture %d octets : % s\n" , nboctet s , message ) ;
// fermeture du descripteur de lecture
c lose ( fd [R] ) ;
}
return 0 ;
}
```

3.5 Les Files Fifo (Tubes nommés)

Un tube nommé est un tube qui a un nom dans le système de fichiers. Tout processus peut l'ouvrir à condition d'en connaître le nom et d'en avoir les droits d'accès. Un processus ayant ouvert un tube en écriture (resp.lecture) est suspendu tant qu'il n'existe pas de processus lecteur(resp.écrivain). Les processus n'ont pas forcément de lien de parenté (comme c'est le cas pour les tubes anonymes déjà rencontrés).

L'appel système *mkfifo(nom_type, mode)* crée un tube nommé, l'argument mode indique les permissions à accorder au tube .

Contrairement aux tubes anonymes, d'une part les tubes nommés possèdent un nom dans le système de fichier et d'autre part ils peuvent être utilisés comme moyen de communication entre processus qui n'ont pas de lien de parenté.

Exemple: Client-serveur avec des tubes nommés

On suppose qu'un serveur a accès à un fichier *notes.txt* contenant les notes d'étudiants. Chaque client envoie ses requêtes dans le tube nommé accesspoint : il écrit dans le tube par *write(accesspoint,"PID-210+Toto",12)*; signifiant qu'il souhaite connaître la note de l'étudiant Toto("210"est son numéro de PID).

Le serveur lit dans le tube *read(accesspoint, tamponreq,100);* (où tamponreq est un tableau de caractères).

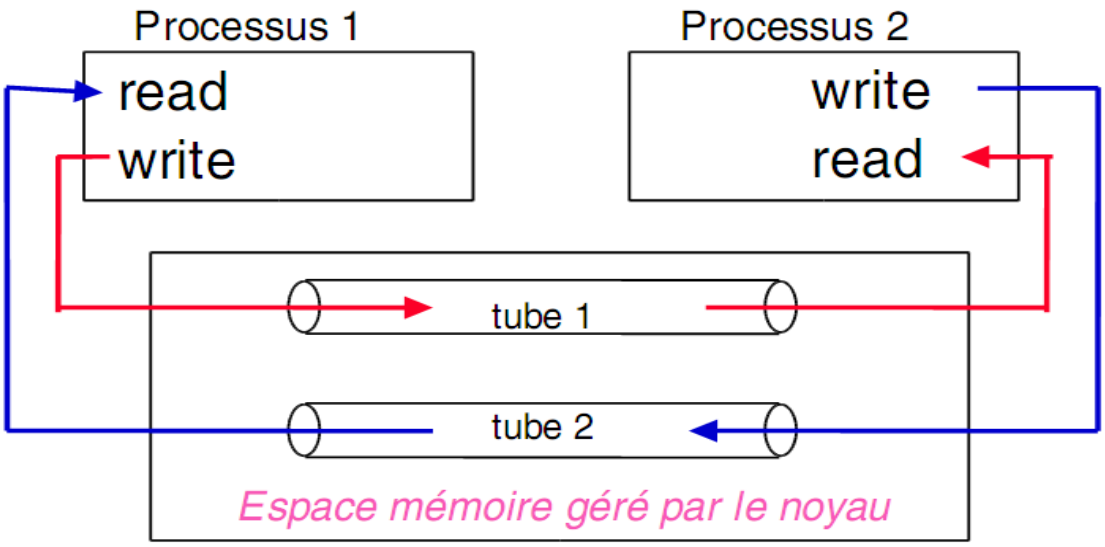
Il cherche la note correspondant à Toto dans notes.txt. Puis il crée le tube nommés rep210 et écrit par *write(rep210,"14",2);* la note de Toto(14 en l'occurrence).

Enfin, le client récupère la note par read(rep210,tableaunote,20); où tableaunote est un tableau de caractères.

Dans cet exemple on a utilisé le protocole de communication suivant:

Les clients envoient leur requête sur le tube accesspoint puis reçoivent les réponses sur le tube repPID (où PID est le numéro de pid du client).

Quand un client se termine, son tube nommé est détruit. L'envoi d'un message client(une requête) consiste en une écriture dans le tube nommé du serveur. La réception d'un message client consiste en la lecture dans le tube nommé du client.



3.6 Les signaux

Les signaux sont une autre forme de communication entre processus. Il sont utilisés pour rendre compte à un processus d'un événement ou d'une erreur. Il peuvent être générés à la suite d'un événement software (CTRL-C,violation de segment) ou hardware (erreur de bus,périphérique non prêt).
Il existe différents signaux pré-définis par le système lui-même qui provoquent la terminaison du processus si celui-ci ne prends fait rien à la réception d'un tel signal. Les comportements par défaut des signaux sont les suivants :

- Le signal est ignoré après avoir été reçu.
- Le processus est terminé après la réception.
- Un fichier *core* sera écrit puis le processus se terminera.
- Le processus se paralyse à la réception du signal.

La création d'un fichier *core* donnera toutes les informations nécessaires pour permettre d'étudier avec *gdb* le moment précis ou le processus à reçu ce signal. Il y a 32 signaux définis, certains peuvent êtres interceptés et pris en charge par le processus d'autres ne peuvent être intercepté ni ignorés. L'ensemble des signaux définis par le système linux se trouvent dans /usr/include/bits/*signum*.h.

La fonction qui permet d'envoyer un signal à un processus est kill*()* est l'équivalent de la commande kill sous Unix :

int kill(pid_t pid, int sig)

Si pid est positif, sig est envoyé au processus pid.
Si pid est nul, sig est envoyé à tous les processus appartenant au même groupe que le processus appelant.

Si pid vaut -1 le signal est envoyé à tous les processus sauf le premier (init).
La fonction renvoie 0 en cas de réussite, -1 en cas d'échec.

3.7 Les Sockets

Un socket est un dispositif de communication bidirectionnel pouvant être utilisé pour communiquer avec un autre processus sur la même machine ou avec un processus s'exécutant sur d'autres machines. Les programmes Internet comme Telnet, rlogin, FTP, talk et le World Wide Web utilisent des sockets.

Pour créer un socket, il faut indiquer trois paramètres: le style de communication, l'espace de nommage et le protocole.

Un style de communication contrôle la façon dont le socket traite les données transmises et définit le nombre d'interlocuteurs. Lorsque des données sont envoyées via le socket, elles sont découpées en morceaux appelés paquets. Le style de communication détermine comment sont gérés ces paquets et comment ils sont envoyés de l'émetteur vers le destinataire.

- Le style *connexion* garantit la remise de tous les paquets dans leur ordre d'émission. Si des paquets sont perdus ou mélangés à cause de problèmes dans le réseau, le destinataire demande
automatiquement leur retransmission à l'émetteur.

- Le style *datagramme* ne garantit pas la remise ou l'ordre d'arrivée des paquets. Des paquets peuvent être perdus ou mélangés à cause de problèmes dans le réseau.

L'espace de nommage d'un socket spécifie comment les adresses de socket sont écrites. Une adresse de socket identifie l'extrémité d'une connexion par socket. Par exemple, les adresses de socket dans « l'espace de nommage local » sont des noms de fichiers ordinaires. Dans « l'espace de nommage Internet », une adresse de socket est composée de l'adresse Internet (également appelée adresse IP) d'un hôte connecté au réseau et d'un numéro de port. Le numéro de port permet de faire la distinction entre plusieurs sockets sur le même hôte.

Un protocole spécifie comment les données sont transmises. Parmi ces protocoles, on peut citer TCP/IP; les deux protocoles principaux utilisés pour Internet, le protocole réseau AppleTalk; et le protocole de communication locale d'UNIX.