Travaux Pratiques 2 : Communication inter-processus

Objectif : Communication inter-processus via les pipes nommés et les signaux.

Exercice 1 : Les pipes nommés

Un pipe nommé a les mêmes caractéristiques qu'un pipe ordinaire. En plus, il a un nom sous Unix comme un fichier ordinaire. Il est créé de la même manière qu'un fichier spécial avec *mknod* ou *mkfifo*. Ensuite, il peut être ouvert avec la primitive *open()*. Il est accessible par les processus **n'ayant pas de lien de parenté**.

Il subit les mêmes règles qu'un fichier ordinaire. Donc, comme son nom l'indique, le pipe dispose d'un nom dans le système de fichier. Il suffit qu'un processus l'appelle par son nom, et il donne droit au processus appelant de lire ou écrire dans son contenu.

Quelques primitives et commandes utiles :

- *mknod(...)* (ou *mkfifo(...)*) : Création d'un pipe nommé
- *open(...) :* Ouverture d'un pipe nommé. Il existe deux façons pour ouvrir un pipe nommé:
  - Ouverture bloquante pour la lecture/écriture : *open (/uo/tcom/mypipe, O RDWR, 2);*
  - Ouverture non bloquante pour la lecture/écriture : *open (/uo/tcom/mypipe, O RDWR | O NDELAY, 2);*

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

char *path; /* chemin */
int mode;   /* type + permissions */
int mknod (path, mode)
```

- *read(...) :* Lecture d'un pipe nommé. Un exemple d'une lecture d'un FIFO: read(d, buf, 2000);
- *write(...)* : Écriture d'un pipe nommé. Un exemple d'une écriture de 5 caractères dans un FIFO de descripteur d: write(d,"ABCDE",5);
  - Si le pipe est **plein** et que l'ouverture soit **non bloquante** (le flag O NDELAY positionné) l'écriture retourne un 0.
  - Si le pipe est **plein** et l'ouverture est **bloquante** (le flag O NDELAY non positionné) le processus reste bloqué jusqu'à ce qu'il y ait de la place (par suite d'une lecture).
- *unlink* ou *rm* : Suppression d'un pipe nommé.

1. Créer le pipe "mypipe" à partir du Shell dans votre répertoire avec la commande : « *mknod mypipe p* ». Écrire un programme P1 qui ouvre de manière **non-bloquante** *mypipe*, et écrit dans un buffer des 'A'. Ensuite, écrire un programme P2 qui ouvre de manière **bloquante** *mypipe*, et lit *mypipe* et affiche son contenu dans le buffer.

2. Exécuter les tests suivants en utilisant 2 terminaux et observer ce qui se passe :

   A) Lancer en premier P1 et ensuite P2, est ce que la lecture et réalisée ?
   B) Lancer en premier P2 et ensuite P1 que se passe-t-il pour la lecture ?
   C) Lancer P1 ensuite P2 deux fois en lisant 2 000.
   D) Lancer P2 avec lecture de 4 000 et P1 avec écriture de 2 000.

Quelques primitives et commandes utiles:

- ***signal(****int sig, void (*action)(int))* : Permet de spécifier le comportement du processus à la réception d'un signal donné, il faut donner en paramètre à la fonction le numéro du signal *sig* que l'on veut détourner et soit une *routine* de traitement à réaliser à la réception du signal, ou SIG_DFL pour un comportement par défaut, ou SIG_IGN pour ignorer le signal.

  Remarque: les signaux SIGKILL, SIGSTOP ne peuvent pas être ignorés.

Travaux Pratiques 2 : Communication inter-processus

- ***kill****(int pid, int sig)* : Permet l'envoi de signaux. La valeur de *pid* indique le PID du processus auquel le signal est envoyé, **0** : tous les processus du groupe du processus réalisant l'appel kill, **1** : tous les processus du système sauf 0 et 1, un ***pid*** **positif** : le processus du PID indiqué, un ***pid*** **négatif** : tous les processus du groupe [*pid*].
- ***pause****(void)* : Fonction de mise en attente de réception d'un signal.
- ***alarm****(int nb_sec)* : Programme un réveil qui enverra un signal SIGALRM au processus appelant dans un délai en secondes spécifié dans le paramètre : *nb_sec*.
- ***system****("commande_shell")* : Permet de lancer une commande Shell à partir d'un programme.
- ***setjmp****(env)* : Assimilable à une étiquette. Elle permet de sauvegarder un point de reprise (pointeur programme et pointeur dans la pile d'exécution) dans le buffer *env*, et renvoie 0.
- ***longjmp****(env , 1)* : Permet de retourner au point de reprise.

Exercice 2 : Ignorer un signal – SIGCHLD

Le signal SIGCHLD (anciennement SIGCLD) est un signal utilisé pour réveiller un processus dont un des fils vient de mourir.

Écrivez un programme C où le processus-père créé un processus-fils. Affichez les PIDs des processus. Et, faites en sorte que le processus-père n'attend pas la mort du processus-fils en ignorant le signal SIGCHLD.

1. Vérifiez la table des processus à l'aide de la commande: ps -aux dans un autre terminal. Que remarquez-vous ?

Travail à faire : Utilisation de SIGUSR1

SIGUSR1 (et SIGUSR2) est un signal défini par le programmeur. Son action par défaut est la mise à mort du processus.

Écrivez un programme C où le processus-père créé deux processus-fils, et affiche son PID. Ensuite, il envoie le signal SIGUSR1 à ses fils en un seul appel : *kill(0, SIGUSR1)*. Le processus-père s'endort pendant 30 secondes afin de laisse aux fils le temps de traiter le signal. Les deux processus-fils affichent leurs PIDs, et attendent la réception du signal en exécutant une routine grâce à la primitive : *signal(SIGUSR1, nom_routine)*.

Exemple de sortie :

```
Processus-père: 15251
Le fils: 15252, reçoit le signal
Le fils: 15253, reçoit le signal
        Le signal, dont le numéro: 10, est reçu par le fils 15253
        Le signal, dont le numéro: 10, est reçu par le fils 15252
```

Dans le cadre d'un devoir individuel, le fichier code source de cet exercice doit être nommé « **NomPrenomEtuduant.c** » et soumis à travers la plateforme de **Google Classroom** et ceci avant le **1er Mai 2025**.