

ENSEIRB-MATMECA



**MISE EN ŒUVRE DES
COMMUNICATIONS
INTERPROCESSUS IPC SOUS *NIX**

Patrice KADIONIK
kadionik.enseirb-matmeca.fr

TABLE DES MATIERES

1	<i>But des travaux pratiques</i>	3
2	<i>Fork et exec</i>	3
3	<i>Signaux</i>	4
4	<i>Tubes et tubes nommés</i>	4
5	<i>IPC System V</i>	5
5.1	File de messages	5
5.2	Sémaphore	5
5.3	Mémoire partagée	6

1 BUT DES TRAVAUX PRATIQUES

Le but de ces TP est de balayer les différents mécanismes de communication interprocessus sous *NIX : `fork`, `exec`, tubes, signaux, IPC System V.

2 FORK ET EXEC

Dans son répertoire de travail, se créer un répertoire de travail `tp_ipc` et s'y placer :

```
% cd
% mkdir tp_ipc
% cd tp_ipc
```

Y recopier tous les fichiers depuis `~kadionik/pub` :

```
% cp ~kadionik/pub/* .
```

Exercice fork1 :

Créer le fichier `fork1.c`. En utilisant l'appel système `fork()`, créer un processus fils qui dort 10 secondes (appel système `sleep()`) puis meurt (`exit()`) alors que le processus père attend la fin d'exécution du processus fils (`wait()`) avant de se terminer.

On n'oubliera pas qu'à tout moment on peut avoir des informations en ligne sur un appel système via la commande *man*.

Exercice fork2 :

Créer le fichier `fork2.c`. En utilisant (l'appel système) `fork()`, créer un processus fils qui dort 10 secondes puis meurt alors que le processus père reste bloqué dans une boucle infinie.

A l'aide de la commande `ps` dans une autre fenêtre terminal, regarder l'évolution de l'état des processus et la création d'un processus zombi pour le fils au bout de 10 secondes. Quand disparaît le processus zombi ?

Exercice execl :

Créer le fichier `execl.c`. En utilisant `fork()`, créer un processus fils qui fait un recouvrement `execl()` avec la commande `/bin/ls` alors que le processus père meurt immédiatement.

3 SIGNAUX

Exercice sig1 :

Créer le fichier `sig1.c`. En utilisant `signal()`, traiter le signal `SIGALRM` en installant la routine de traitement `it()`. On utilisera ensuite `alarm()` pour générer le signal `SIGALRM` au bout de 5 secondes, le programme étant ensuite dans une boucle infinie. Vérifier le bon traitement du signal `SIGALRM` au bout de 5 secondes par la routine `it()`.

On peut vérifier la bonne valeur du numéro du signal avec la commande :

```
$ kill -l
```

Exercice sig2 :

Créer le fichier `sig2.c`. En utilisant `signal()`, traiter le signal `SIGINT` en installant la routine de traitement `fin()`. Le programme est ensuite dans une boucle infinie. Taper alors le caractère `CTRL C`. A quoi correspond ce caractère spécial ? Que se passe-t-il ?

Exercice sig3 :

Créer le fichier `sig3.c`. En utilisant `fork()`, créer un processus fils qui traite le signal `SIGUSR1` en installant la routine de traitement `it_fils()` et qui reste ensuite bloqué dans une boucle infinie. La routine de traitement `it_fils()` enverra alors le signal `SIGINT` au processus père en utilisant son `pid` (`getpid()`). Le processus père s'endort 5 secondes puis envoie le signal `SIGUSR1` au fils. Vérifier le bon fonctionnement du programme.

4 TUBES ET TUBES NOMMES

Exercice pipe1 :

Créer le fichier `pipe1.c`. Créer un tube avec `pipe()`. Ecrire 2 caractères ASCII 'A' et 'B' dans le tube puis on ferme le tube en écriture. On essaiera de lire 3 caractères du tube. Que se passe-t-il sur la tentative de lecture du 3^{ème} caractère ?

Exercice pipe2 :

Créer le fichier `pipe2.c`. Créer un tube avec `pipe()`. Installer avec `signal()` la routine `it()` de traitement du signal `SIGPIPE`. Fermer le tube en lecture. Ecrire le caractère ASCII 'A' dans le tube. Que se passe-t-il ?

Exercice pipe3 :

Créer le fichier `pipe3.c`. Créer un tube avec `pipe()`. En utilisant `fork()`, créer un processus fils qui lit le tube alors que le processus père écrit dans le tube le message «Hello fiston».

Exercice fifo1 :

Créer le fichier `fifo1.c`. Créer le tube nommé `ma_fifo` avec `mkfifo()`.

Exercice fifo2 :

Créer le fichier `fifo2.c`. Reprendre l'exercice `pipe3` mais en utilisant le tube nommé `ma_fifo` comme moyen de communication entre le processus père et le processus fils.

5 IPC SYSTEM V

5.1 File de messages

Exercice bourse :

Etudier le fichier `bourse.c` (qui a été expliqué en cours). Compiler et tester le programme.

On rappelle l'usage du programme `bourse` :

```
$ ./bourse client 100
$ ./bourse client 50
$ ./bourse agent
$ ./bourse agent
```

Exercice msg1 :

Créer le fichier `msg1.c`. En partant de l'exercice `bourse`, reprendre l'exercice `pipe3` mais en utilisant la file de message comme moyen de communication entre le processus père et le processus fils.

5.2 Sémaphore

Exercice rdv :

Etudier le fichier `dijkstra.h`. Quel type de sémaphore a-t-on implémenté ?

On désire mettre en place le rendez-vous entre un processus père et un processus fils créé avec `fork()`. A quelle valeur doit-être initialisé le sémaphore ?

Créer le fichier `rdv.c`. On inclura le fichier `dijkstra.h` dans `rdv.c` pour utiliser la bibliothèque « Dijkstra ». En utilisant `fork()`, créer un sémaphore convenablement initialisé et créer un processus fils qui libère le sémaphore au bout de 5 secondes alors que le processus père est en attente de libération du sémaphore.

Exercice partage1 :

Créer le fichier `partage1.c` à partir du fichier `rdv.c`. On désire maintenant utiliser un sémaphore pour gérer l'accès exclusif à une variable partagée entre le processus père et le processus fils.

La variable partagée est du type :

```
int partage ;
```

Quand le processus père ou le processus fils gagne l'accès à cette variable partagée, cette dernière sera incrémentée de 1. La valeur courante sera affichée à l'écran.

Le processus père et le processus fils implémentent une boucle infinie d'accès concurrent à la variable partagée.

On utilisera la bibliothèque « Dijkstra » pour le sémaphore.

Qu'observe-t-on sur l'évolution de la valeur courante de `partage`. Expliquer.

5.3 Mémoire partagée

Exercice shm1 :

Etudier le fichier `shm1.c`.

Le processus père et le processus fils vont « mapper » un même segment mémoire qu'ils vont partager. Le processus fils écrit la chaîne de caractères ASCII «Hello papa» dans le segment mémoire puis meurt au bout de 3 secondes. Le processus père attend la mort du processus fils (`wait()`) puis lit le contenu du segment mémoire et affiche finalement le message du processus fils.

Y-a-t-il ici un problème d'accès concurrent à la zone mémoire partagée ?

Exercice partage2 :

On reprend l'exercice précédent `partage1` en combinant les fichiers `shm1.c` et `partage1.c` pour créer le fichier `partage2.c`. Mettre en place un segment mémoire partagée dont l'accès est protégé par un sémaphore.

La variable partagée `partage` sera mappée dans ce segment mémoire.

Qu'observe-t-on sur l'évolution de la valeur courante de `partage`. A-t-on maintenant le résultat escompté ?