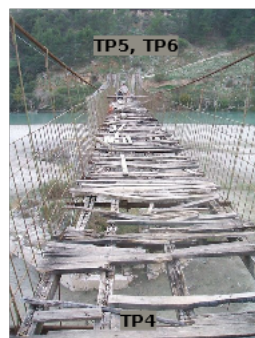


TP4 – IPC : Mémoire partagée et sémaphores

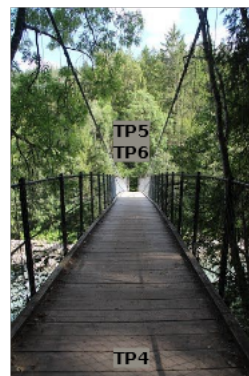
Lien avec le cours

- ✎ En utilisant un tube ou des files de messages, un message est créé dans le premier processus, recopié dans le tube puis une nouvelle fois stocké dans le deuxième processus : on a donc pris 3 fois la place mémoire d'un message. Pour éviter cette duplication, on va utiliser le mécanisme de segment de mémoire partagée proposé fourni par les IPC.
- ✎ Comme la mémoire est partagée, elle est donc commune à deux processus, il sera peut-être nécessaire de protéger l'accès à cette ressource critique. Nous utiliserons alors des sémaphores qui sont aussi un mécanisme IPC.

Durant les exercices suivants, vous aurez besoin de lire la partie concernant la mémoire partagée et les sémaphores du chapitre sur les IPC.



Avant le TP4



Après le TP4

Exercice n°1 : Mémoire partagée et sémaphore

Le but de cet exercice est de mettre en place une variable commune à deux processus et de prendre conscience qu'il faut donc mettre des sémaphores pour protéger l'accès.

- Créez un père et son fils (dans le même fichier).

- ▶ Avant le fork, le père créera un segment de mémoire partagée qui sera de la taille d'un entier `int` et que l'on appellera *pVariablePartagee* (c'est-à-dire un pointeur sur un entier). Il peut utiliser la clé `IPC_PRIVATE` (puisque le fils est dans le même programme, pas besoin de *ftok* pour générer une clé). Le père s'attachera puis initialisera la valeur à $(*pVariablePartagee) = 0$. Comme nous n'utilisons pas ici *execl*, le fils hérite directement de ce pointeur et sera donc automatiquement attaché à ce segment de mémoire partagée (sinon nous aurions dû le faire aussi).



Pas de *malloc*

Il ne faut pas utiliser la fonction *malloc* pour créer la variable mais bien un segment de mémoire partagée. La fonction *malloc* crée une variable^a dans l'espace d'adressage du père puis est dupliquée chez le fils, donc elle n'est pas commune !

a. En fait il s'agit d'une zone mémoire, qui sera utilisée avec un pointeur.

- le père (une fois qu'il aura créé le fils) incrémentera 100000 fois (¹) cet entier (*pVariablePartagee*) de 3 de la manière suivante :

100000 fois :

```
tmp = (*pVariablePartagee);
tmp = tmp + 3 ;
(*pVariablePartagee) = tmp;
```

tmp est une variable locale au père. Le fait de décomposer l'opération permet de comprendre plus facilement ce qu'il peut se passer en général.

- parallèlement, le fils incrémentera 100000 fois cet entier de 5 de la même manière.
- le père attendra la mort du fils pour afficher la valeur contenue dans la variable.

En lançant le programme plusieurs fois, vous remarquerez que la valeur est très rarement voire jamais égale à 800000 ($3 \times 100000 + 5 \times 100000$). Pourquoi ?

Réponse : la valeur est écrasée par l'autre processus pendant le calcul intermédiaire.

- ▶ Il faut bien sûr protéger toute cette section qui est donc critique. Nous allons utiliser un *ensemble de sémaphores IPC* (contenant 1 sémaphore d'exclusion mutuelle).


Vous créerez un deuxième fichier *mes_semaphores.c* dans lequel vous écrirez les 6 fonctions utilisées lorsque l'on manipule des sémaphores :

- **int sem_creation(int * semid, int nombre_semaphores)** : qui crée un ensemble de sémaphores (*semid*) de *nombre_semaphores*. Elle renvoie 0 si la création a bien fonctionné. Elle sera utilisée par le père.
- **int sem_initialisation(int semid, int num_semaphore, int nbr_jetons)** : initialise le sémaphore numéro *num_semaphore* de l'ensemble de sémaphore *semid* par la valeur *nbr_jetons*.

1. il se peut qu'il faille faire cela 10 millions de fois sur certaines machines pour bien voir ce qu'il se passe.

- **int P(int semid, int num_semaphore)** : Primitive P pour le sémaphore numéro *num_semaphore*. (pour cet exercice le numéro sera toujours 0 mais nous réutiliserons ces fonctions dans l'exercice suivant)
- **int V(int semid, int num_semaphore)** : Primitive V pour le sémaphore numéro *num_semaphore*.
- **int sem_destruction(int semid)** : qui détruit un ensemble de sémaphores (*semid*). Elle sera utilisée par le père.
- **int sem_recup(int * semid, int nb_semaphores)** : qui récupère le numéro de l'ensemble de sémaphores (*semid*) (généralement utilisé par le processus le fils, mais ici nous ne l'utiliserons pas puisque nous n'utilisons pas *execl*...mais faites-là quand même :) elle servira dans le prochain exercice). En entrée, il faut indiquer le nombre de sémaphore qu'il y a dans l'ensemble à récupérer. La fonction est identique à la création sauf pour l'option de *semget* qui doit être à 0.



en C

-  Il faut compiler vos fichiers en 2 temps : obtenir les fichiers .o d'abord puis faire l'édition de liens.
 - gcc -c monfichier.c -o monfichier.o (pour tous les fichiers)
 - gcc monfichier1.o monfichier2.o monfichier3.o -o monfichierexecutablefinal

- Le père générera l'ensemble de sémaphores (qui n'en contient qu'un) et l'initialisera avec 1 jeton (avant de créer le fils bien sûr !). N'oubliez pas de tester le retour de la fonction *semget()* car même si l'ensemble de sémaphores n'est pas créé le programme tournera quand même....
- C'est le père qui doit détruire l'ensemble des sémaphores.
- N'oubliez pas de redéfinir l'union *semun* (dans le fichier .c qui contient vos fonctions sur les sémaphores).


Et si tout fonctionne vous obtenez bien 800000 !

Vérifiez vos IPCs

-  Lors de réalisation du programme, vérifiez que la file a bien été détruite : taper *ipcs* (dans le shell) et effacer, si besoin est, la file de messages existante *ipcrm -q numéro_msqid*
-  Un petit script en bash (*Script_ipcs.bash*) est disponible su ARCHE. Il permet de lister et de générer automatiquement les commandes de destruction de tous les IPCs d'un même propriétaire (il suffit juste de copier-coller le résultat par exemple). Pour l'exercice 2, il peut vous faciliter la vie pour faire le ménage...



L'exo 1 est bien fait ? Vous l'avez compris ?

 **Il faut vous assurer que vos fonctions P, V, sem ... fonctionnent pour le prochain TP. Vous serez amené-e à les utiliser lors du TP5 et TP6 !**

Exercice n°2 : Demande de ressources

L'objectif de cet exercice est d'illustrer le problème d'interblocage dû à la demande de plusieurs ressources (*cf.* TD2 et TD3). Il y a 2 ressources *A* et *B*, chacune protégée par un sémaphore (mutex). Un processus père créera 6 fils. 3 fils demanderont la ressource *A* puis *B*, et les 3 autres la ressource *B* puis *A*. La demande des deux ressources se fait avant l'utilisation des deux. Chaque fils aura dans un premier temps l'algorithme suivant :

Algo du fils *i* demandant *X* et *Y* :

```
printf : "fils /pid/ : Je commence mon programme"  (+fflush)

/*demande*/
P(X)
utiliseX=i
printf : "fils /pid/ : Merci pour la ressource X"  (+fflush)
usleep(500000)
P(Y)
utiliseY=i
printf : "fils /pid/ : Merci pour la ressource Y" (+fflush)
usleep(500000)

/*utilisation*/
printf : "fils /pid/ : J'utilise X et Y" (+fflush)
usleep(1000000)

/*liberation*/
utiliseX=0
```

```

V(X)
utiliseY=0
V(Y)
printf : "fils /pid/ : J'ai libéré X et Y, au revoir" (+fflush)

```

X et Y sont à remplacer par A et B ou B et A suivant le cas. *utiliseA* et *utiliseB* sont des variables communes à tous les processus servant à connaître le possesseur de la ressource (i sera le pid du processus). (Notez que ces variables sont déjà protégées par P(A) et P(B)). Les ressources A et B ne sont pas matérialisées dans le programme, c'est au moment de la partie *utilisation* que l'on suppose que A et B sont en train d'être utilisées.

Vous ferez le code du processus fils dans un fichier différent du père. Ce fichier sera donc appelé 6 fois avec *execl* avec comme paramètres les ressources demandées (A, B ou B, A), l'ordre est important.

1. Algo à la demande ⇒ Interblocage

Réalisez ces demandes. Le processus père créera les sémaphores (utilisez le fichier *mes_semaphores.c* créé dans l'exercice précédent), les segments de mémoire partagée et les 6 fils. Puis le père attendra ces 6 fils (avec *wait*), et affichera un message de fin et détruira les sémaphores et les segments de mémoire partagée.

Le programme devrait tomber en interblocage...! Donc n'oubliez pas d'utiliser *ipcrm* pour faire le ménage.

2. Algo à la demande ⇒ Interblocage : affichage de l'interblocage

- Pour se rendre compte de l'interblocage, toutes les secondes le père se réveillera pour afficher le contenu des 2 variables communes (*utiliseA* et *utiliseB*). Utilisez la fonction *alarm(1)* (=le signal SIG_ALARM est envoyé au bout de 1 seconde) une première fois dans le programme du père. Puis ensuite dans le handler, affichez la valeur des variables puis relancez le décompte de l'alarme (*alarm(1)*).



Effet de bord de l'alarme



Le lancement de l'alarme annule les signaux d'attente (!). La fonction *wait/waitpid* sort de son attente même si le fils n'est pas fini. Il faut donc vérifier le retour du *wait* : si -1 (en cas de sortie prématurée, ici due à l'alarme) alors reboucler.

Comme le programme va tomber en interblocage, nous allons prévoir un délai après lequel l'alarme n'affichera pas les variables mais un message puis détruira les fils et les sémaphores et les segments de mémoire partagée. Ce délai sera de 30 secondes (en effet, sans interblocage les fils devraient avoir fini avant).

3. Algo de protection de la demande ⇒ Pas d'interblocage

Dans des nouveaux fichiers (père et fils) modifier votre programme pour éviter l'interblocage : utilisez la méthode de protection de la demande (avec un nouveau sémaphore). Et vérifiez qu'il n'y a plus d'interblocage.



FIN de séance

- ☞ Vérifiez qu'il ne reste pas de processus ou d'IPC en mémoire. Pour vérifier les IPC : taper `ipcs` (dans un shell) et pour effacer une file de messages existante `ipcrm -q numéro_msqid`.
- ☞ Utilisez le script en bash (`Script_ipcs.bash`) est disponible sur ARCHE.