TP C avancé n^o 7 Licence Informatique L3 - 2020-2021
G. Subrenat, Y. Bertrand, N. Courilleau, D. Meneveaux UFR SFA - Université de Poitiers

Communication IPC

Table des matières

1 Résumé et cadre de travail

Dans ce TP il s'agit de manipuler la communication entre plusieurs processus lourds (ou non d'ailleurs) via des sémaphores IPC.

Il y a également une initiation aux segments de mémoire partagée et aux files de messages.

Rappel d'une directive importante : pour tout appel système, (ouverture de fichier, lancement d'un thread, ...) vous devez tester la valeur de retour pour détecter toute anomalie ; on ne présuppose jamais qu'un appel système fonctionne. Un *assert* suffira amplement.

Le fichier exécutable *rmsem.sh* détruit tous les sémaphores qui ont les droits 0641 (cf. section 3).

2 Principe de fonctionnement des sémaphores IPC

Point de cours

Un sémaphore permet à plusieurs processus de se synchroniser. On note deux grands types de synchronisation :

- précédence : le code d'un processus doit impérativement s'exécuter avant le code d'un second
- barrière : en un point précis du code, les processus doivent s'attendre mutuellement avant de continuer leurs exécutions

Nous utiliserons obligatoirement les fonctions suivantes :

- *semget* : pour créer ou ouvrir un sémaphore
- *semop* : pour faire les actions courantes sur un sémaphore
- *semctl* : pour effectuer des opérations de "maintenance" (initialisation, destruction, ...)
- *ftok* : pour de générer facilement des clés uniques pour l'identification des sémaphores (ou autres objets IPC d'ailleurs).

Précision : *semget* crée en réalité un tableau de sémaphores, tableau dont la taille peut tout à fait être réduite à une case.

Pensez que les sémaphores IPC ne sont pas automatiquement détruits à la fin de l'exécution des processus. C'est donc bien aux processus de détruire explicitement les sémaphores inutiles. Si ce n'est pas fait, il faut effacer les sémaphores à la main dans la console de commandes (cf. commandes "*ipcs -s*", "*ipcrm -s*" et "*ipcrm -S*)".

3 Création et destruction de sémaphores

a) La commande

```
$ ipcs -s
```

permet d'avoir la liste des sémaphores existants. Notez les bien car lorsqu'on en effacera à la main (i.e. dans une console) il ne faudra pas toucher à cette liste initiale (ce qui autrement risquerait d'altérer le fonctionnement d'autres processus).

b) Créez un programme C (squelette *exo3b.c* fourni) qui crée un tableau de deux sémaphores. La clé sera *IPC_PRIVATE* et les droits seront :

- *rw-* pour l'utilisateur
- *r--* pour le groupe
- *--x* pour les autres

Ces droits farfelus (0641 donc) permettront d'identifier facilement vos sémaphores avec la commande "*ipcs -s*", et de les effacer automatiquement avec l'exécutable *rmsem.sh*.

Il est conseillé d'utiliser ces droits pour tout le reste du TP.
Ne détruisez pas le sémaphore dans votre programme et vérifiez qu'il est bien présent après l'exécution de ce dernier.
Enfin supprimez le sémaphore avec la commande "*ipcrm -s*".

c) Relancez le programme précédent trois fois et regardez que qu'affiche "*ipcs -s*". Supprimez tous les sémaphores inutiles.

d) Cette fois-ci le programme détruit le (tableau de) sémaphore(s). Le squelette *exo3d.c* est fourni.
Après l'exécution, vérifiez (avec "*ipcs -s*") que tout est correct

4 Utilisation basique d'un sémaphore

Le squelette *exo4.c* vous est fourni.

Le processus se duplique (*fork*) et chacun des deux processus fait un affichage. Mais il faut que celui du père se fasse avant celui du fils.
Il faut donc implanter une précédence pour assurer ce fonctionnement.

Le code fourni est commenté et vous indique quoi faire et à quels endroits.

5 Sémaphore et processus indépendants

Le but est le même que l'exercice précédent, mais les deux processus faisant une précédence ne sont pas issus directement d'un *fork*.

Il y a deux programmes *exo5_un.c* et *exo5_deux.c* dont les squelettes sont fournis.
exo5_un.c doit afficher son message en premier et donc *exo5_deux.c* en dernier.

Cela amène deux remarques :
- pour la clé il n'est pas possible de choisir *IPC_PRIVATE*,
- un des deux processus devra créer le sémaphore et l'autre uniquement le récupérer sans le créer.

Pour le second point, nous allons le régler violemment : c'est *exo5_un.c* qui le crée, et *exo5_deux.c* qui le récupère en faisant l'hypothèse qu'il est déjà créé.
Donc lors des tests, il faudra lancer *exo5_un* en premier.

Il faut que le premier paramètre de *semget* soit le même dans les deux programmes. Pour cette première version de l'exercice vous le choisissez et l'écrivez en dur dans le code (par exemple 78624).

6 Sémaphore et processus indépendants (version 2)

Point de cours

Lors de la création d'un sémaphore, il faut lui associer une clé. Et comme le sémaphore est *a priori* visible par tous les autres processus (en fonction des droits tout de même), tout autre processus désirant manipuler le même sémaphore doit tout simplement utiliser la même clé.

Le problème est double :
- comment choisir la valeur de la clé ?
- comment tous les processus manipulant le sémaphore connaissent la clé ?

Le seconde point n'est pas un problème, il suffit par exemple de mettre sa valeur dans un *.h* commun [a].

Le premier point est délicat car on ne peut pas choisir la valeur de manière arbitraire. En effet

souvenez-vous qu'un sémaphore n'est pas local à un utilisateur, mais global à la machine. Supposons que l'on choisisse la valeur 78623. Et bien on court le risque qu'un autre utilisateur de la machine choisisse la même clé [b].
C'est alors la catastrophe car notre programme (ou celui de l'autre utilisateur) pourrait ne pas fonctionner ; ou pire ils pourraient se perturber l'un l'autre.

Il existe une fonction (*ftok*) dont le but est de générer des clés uniques. Elle prend en entrée un nom de fichier présent sur le système et renvoie une clé différente pour chaque fichier [c].
La fonction est un peu plus souple. Elle prend un second paramètre qui permet de générer jusqu'à 255 clés différentes à partir du même fichier.

a. ou dégueux : mettre sa valeur en dur dans les programmes.
b. C'est certes peu probable, mais il n'est pas possible de prendre le risque.
c. La fonction utilise une partie de l'i-node du fichier, et donc les collisions sont en réalité possibles, cf. manuel.

Reprenez l'exercice précédent, mais avec une clé générée par *ftok*.

Les squelettes de *exo6_un.c* et *exo6_deux.c* sont fournis.

7 Sémaphore et barrière de synchronisation

Le but est ici que plusieurs processus s'attendent mutuellement au milieu de leur code avant de continuer.

Voici le pseudo-code des processus :

```
début
   je fais la 1re partie du code
   ...

   j'attends ici que tous les autres arrivent au même point

   // si je suis là, alors tous les autres aussi (ou plus loin)

   je fais la 2e partie du code
   ...
fin
```

Techniquement il s'agit d'utiliser l'opération 0 (zéro) de la fonction *semop*.

Le squelette des processus devant se synchroniser est fourni : *exo7_worker.c*.

Il reste toujours le problème de la création et de la destruction du sémaphore. Pour cela un programme à part se charge de ces opérations (squelette *exo7_master.c*). Il faut le lancer en premier en indiquant combien de *workers* vont être lancés ; et il faudra lui indiquer de détruire le sémaphore lorsque les *workers* auront terminé.
Il faudra une console pour le *master* et une console par *worker*.

8 Mémoire partagée

Point de cours

Deux (ou plus) processus lourds indépendants ou issus directement d'un *fork* ont par définition des espaces mémoire indépendants (au contraire des threads).

Il est possible malgré tout d'utiliser une zone partagée (ou encore segment partagé) via des primitives IPC.

La démarche est la suivante :

- création ou récupération de l'objet IPC : fonction *shmget*
- déclaration d'un pointeur qui pointe sur cette zone : fonction *shmat*
- travail avec la zone
- détachement du pointeur : fonction *shmdt*
- éventuellement destruction du segment : fonction *shmctl*

Pour avoir les liste des segments partagés présents sur la machine :

```
$ ipcs -m
```

Écrivez deux programmes.
Le premier :

- crée le segment
- écrit une chaîne de caractères
- détruit le segment

Le second :

- récupère le segment
- lit la chaîne
- ne détruit pas le segment puisque c'est le premier processus qui le fait

Attention à faire les opérations dans le bon ordre (synchronisation ?). Par exemple il ne faut pas que le second processus lise le segment avant que le premier n'ait écrit dedans.

9 File de messages

Point de cours

Le troisième type d'objet IPC est la file de messages. Par rapport à un tube, le fonctionnement est plus évolué ; notamment les messages sont typés et deux messages de types différents peuvent avoir des priorités différentes.

La démarche est la suivante :

- création ou récupération de l'objet IPC : fonction *msgget*
- envois de messages : fonction *msgsnd*
- et/ou réceptions de messages : fonction *msgrcv*
- éventuellement destruction de la file : fonction *msgctl*

Pour avoir les liste des files présentes sur la machine :

```
$ ipcs -q
```

Écrivez trois programmes.
Le premier :

- crée la file
- y envoie deux suites de valeurs :
  - une d'entiers terminée par -1
  - une de caractères terminée par '.'

  c'est le type du message qui indique à quelle suite appartient une valeur. Les deux suites s'entremêlent et ont des longueurs non connues par les deux processus récepteurs.
- détruit la file

Le second :

- récupère la file
- lit la suite d'entiers associés au premier type et en fait la somme

- ne détruit pas la file puisque c'est le premier processus qui le fait

Le troisième :
- récupère la file
- lit la suite de caractères associés au second type et les affiche
- ne détruit pas la file puisque c'est le premier processus qui le fait

Attention à faire les opérations dans le bon ordre (synchronisation ?). Par exemple il ne faut pas que le premier processus détruise la file si elle est encore utilisée par les récepteurs.

10 Paquetage de sémaphore

Comme l'utilisation des sémaphores est assez technique, il peut être intéressant de faire un module qui masque l'implémentation.

Programmez un tel module dont le squelette vous est fourni :
- *Semaphore.h* : à ne pas modifier
- *Semaphore.c* : à compléter

Reprenez l'exercice 7 en utilisant le module plutôt que de programmer directement les sémaphores.