Matière : PRS (Programmation Système)
Année : LA1
Année scolaire : 2019-2020
Enseignant : Isabelle Le Glaz

TP n°4 - Mémoire partagée, tubes et tubes nommés

Ce quatrième TP aborde l'utilisation de mémoire partagée entre processus, et des tubes pour échanger des données entre threads et entre processus.

Ce TP fera l'objet d'un compte-rendu contenant :

- Les réponses aux diverses questions présentes dans cet énoncé,
- Les codes sources commentés des programmes réalisés,
- Les jeux d'essais obtenus,
- Une conclusion sur le travail réalisé, les difficultés rencontrées, les connaissances acquises ou restant à acquérir, …

Il sera tenu compte de la qualité de vos pratiques de production de code :

- Lisibilité du code : emploi de commentaires, indentation, constantes symboliques
- Automatisation : Fichiers makefile avec macros …
- Robustesse : Vérification des valeurs de retours des appels systèmes

1 La mémoire partagée

Un des trois dispositifs IPC est la mémoire partagée: Ce dispositif permet l'échange de données entre plusieurs processus. Pour y parvenir, ces données sont placées dans un segment de mémoire que ces différents processus connaissent. Ce dispositif permet ainsi de réduire les appels au noyau (par opposition aux tubes ou encore aux files de messages).

Cependant, pour une bonne gestion de la mémoire partagée, celle-ci devra être couplée à un dispositif de verrouillage par sémaphore pour éviter les conflits d'accès. En effet, il faudra qu'un processus détermine s'il peut ou non accéder à la mémoire partagée avant de faire une mise à jour.

Pour pouvoir utiliser les fonctions dont les explications vont suivre, il faut inclure le fichier de définition suivant:

#include <sys/shm.h>

Sur certains OS (plus anciens), il faut aussi inclure :

#include <sys/types.h>

#include <sys/ipc.h>

1.1 Création et ouverture d'un segment de mémoire partagée

La fonction shmget() retourne un identifiant permettant de manipuler le segment ouvert ou créé.

int shmget (key_t key, int size , int shmflag);

En entrée:

key est la clef IPC obtenue avec ftok() (voir obtention d'une clé unique), et qui caractérise l'identifiant du segment. Vous passez par une clef unique d'identification, dans ce cas, la mémoire partagée ne sera créée que si IPC_CREAT est positionné dans mode et qu'il n'existe déjà pas une file de message pour cette clef. Ou par IPC_PRIVATE qui garantie la création d'un canal IPC avec une key unique.

size indique la taille du segment en octets.

shmflag indique les permissions **et** le mode d'ouverture: 0666 (en octal) avec IPC_CREAT et/ou IPC_EXCL (utilisez le | pour les combiner: 0666 | IPC_CREAT | IPC_EXCL).

En sortie:

Un retour -1 signale une erreur de type errno. Sinon, il s'agit d'un identifiant shmid de type int

NB : shmget permet d'ouvrir ou de créer un segment de mémoire partagée, mais ne permet pas d'y accéder physiquement. Il faudra utiliser la fonction char *shmat(int shmid , char *shmaddr , int shmflag) pour avoir un pointeur sur cette zone mémoire.

1.2 Attacher un segment de mémoire partagée

La fonction shmat() permet d'attacher le segment au processus courant:

char *shmat(int shmid , char *shmaddr , int shmflag);

En entrée:

shmid est l'identifiant du segment retourné par shmget().

shmaddr précise l'adresse du segment attaché, en pratique il doit être placé à NULL, dans ce cas le noyau allouera une adresse.

shmflag indique les conditions d'accès au segment, par défaut 0 autorise la lecture et l'écriture, SHM_RDONLY interdit l'écriture.

La fonction shmat retourne l'adresse du segment attaché en cas de réussite, - 1 sinon et le code erreur est dans errno.

NB : Comme souvent sur les systèmes, suite à l'allocation mémoire et la récupération d'un pointeur, veillez à vider la zone allouée à votre segment, celle-ci peut contenir des données aléatoires...

1.3 Détacher un segment de mémoire partagée

La fonction shmdt() permet de détacher un segment de mémoire du processus courant:

int shmdt (char *shmaddr);

En entrée:

shmaddr est l'adresse du segment à détacher.

En sortie

La fonction shmdt retourne 0 ou -1 en cas d'erreur et errno vaut EINVAL pour indiquer une adresse invalide.

NB : Cette fonction ne supprime pas un segment de mémoire partagée, elle se contente simplement de détacher le segment du processus exécutant le shmdt.

Pour effacer le segment, voir la fonction suivante **shmctl**

1.5 Contrôle, maj, suppression, vérrouillage/déverrouillage de la mémoire partagée

La fonction shmctl va permettre de récupérer et manipuler les informations liées à la mémoire partagée via son identifiant shmid

int shmctl (int shmid, int cmd, struct shmid_ds *buf);

shmid est l'identifiant du segment retourné par shmget().

cmd est l'une des commandes précisées ci-dessous.

buf est un pointeur sur une structure shmid_ds.

La valeur de cmd peut-être l'une des suivantes:

- **IPC_STAT** Permet de lire la structure shmid_ds.
- **IPC_SET** Permet de modifier la structure shmid_ds.
- **IPC_RMID** Permet de supprimer le segment de mémoire partagée, buf pourra être positionné à NULL dans ce cas. La suppression sera effective lorsque le membre shm_nattch sera passé à zéro. Seuls les créateur/propriétaire ou le super utilisateur peuvent lancer cette fonction.

Si le segment mémoire contient des données confidentielles, veuillez bien à effacer le contenu de cette zone avant de supprimer le segment ! En effet, lors de la prochaine allocation mémoire, une partie des données du segment mémoire détruit pourrait devenir accessible à d'autres programmes !

Sous Linux pour éviter ou autoriser le swapping:

SHM_LOCK Vérouillage en mémoire du segment (SU seulement).

SHM_UNLOCK Dévérouillage du segment (SU seulement).

La fonction shmctl() retourne 0 si pas d'erreurs, -1 sinon.

Une structure **shmid_ds** est utilisée pour chaque mémoire partagée.

1.7 Questions

1 - Ecrire un programme shm_e1.c qui crée un segment de mémoire partagée, dans lequel il stocke une chaîne de caractères de 100 caractères maximum, et affiche toutes les informations connues sur ce segment de mémoire partagée, puis détruit le segment de mémoire partagée.

2 - Ecrire un programme shm_e2.c qui complète le programme shm_e1.c en réalisant les opérations suivantes :

- Le programme crée un fils,le fils écrit dans la mémoire partagée et meurt.
- Le père attend la mort de son fils, lit dans la mémoire partagée et affiche le message du fils.

3 - Ecrire un programme shm_e3.c qui complète shm_e2.C en réalisant les opérations suivantes :

- Le programme crée à présent 2 fils.
- Chaque fils écrit un message différent dans la zone de mémoire partagée.Ce message est écrit caractère par caractère, (par exemple le premier fils écrit des caractères A, le second des caractères B), et fait une pause entre chaque caractère (par exemple avec sleep).
- Le père attend la mort de ses deux fils, lit dans la mémoire partagée et affiche le message

Que constatez vous ?

Exécutez plusieurs fois votre programme, Changez le temps de pause d'un des deux fils.Que constatez-vous ?

4 - Ecrire un programme shm_e4.c qui vous garantit que le dernier processus écrivant dans le segment de mémoire partagé a le dernier mot (mais on ne choisit pas qui aura le dernier mot)!

5 - Que se passe-t-il si le père n'attend pas la fin de ses fils pour lire le segment de mémoire partagée ? Comment résoudre ce problème simplement avec les informations à notre disposition sur le segment de mémoire partagée et les techniques déjà vues dans les TP précédents ? Ecrire un programme shm_e5.c permettant de résoudre ce problème.

6 - Testez et commentez la commande shell ipcs concernant les segments de mémoire partagée.

7 - Testez et commentez la commande shell ipcrm concernant les segments de mémoire partagée.

8 - Ecrivez deux programmes différents :

- Le premier s'appelle shm_srv.c; Il réalise les mêmes fonctionnalités que le programme shm_e1.c, mais écrit un message dans le segment de mémoire partagée et attend 30 secondes avant de se terminer.
- Le second s'appelle shm_cli.c; il a pour mission d'aller lire le message laissé par shm_srv et de l'afficher.

2 2.Les tubes anonymes

2.1 Contexte

Un tube anonyme de communication est un tuyau (en anglais **pipe**) dans lequel un processus peut écrire des données et un autre processus peut lire. Ce mécanisme inventé pour UNIX est principalement utilisé dans la communication inter-processus. Ouvrir un tube anonyme permet de créer un flux de donnée unidirectionnel FIFO entre un processus et un autre. Ces deux processus doivent se connaître, c'est à dire avoir un lien de parenté pour communiquer.Le tube anonyme disparaît quand le processus qui l'a créé meurt.

2.2 Création d'un tube

On crée un tube par un appel à la fonction **pipe**, déclarée dans unistd.h :

int pipe(int descripteur[2]);

La fonction renvoie 0 si elle réussit, et elle crée alors un nouveau tube. La fonction pipe remplit le tableau descripteur passé en paramètre, avec :

- descripteur[0] désigne la sortie du tube (dans laquelle on peut lire des données) ;
- descripteur[1] désigne l'entrée du tube (dans laquelle on peut écrire des données) ;

Le principe est qu'un processus va écrire dans descripteur[1] et qu'un autre processus va lire les mêmes données dans descripteur[0]. Le problème est qu'on ne crée le tube dans un seul processus, et un autre processus ne peut pas deviner les valeurs du tableau descripteur.

Pour faire communiquer plusieurs processus entre eux, il faut appeler la fonction pipe avant d'appeler la fonction fork. Ensuite, le processus père et le processus fils auront les mêmes descripteurs de tubes, et pourront donc communiquer entre eux. De plus, un tube ne permet de communiquer que dans un seul sens. Si l'on souhaite que les processus communiquent dans les deux sens, il faut créer deux pipes.

2.3 Lecture/ecriture dans un tube

Pour écrire dans un tube, on utilise la fonction **write** :

ssize_t write(int *descripteur1*, **const void** **buf*, **size_t** *count*);

Le descripteur doit correspondre à l'entrée d'un tube. La taille est le nombre d'octets qu'on souhaite écrire, et le bloc est un pointeur vers la mémoire contenant ces octets.

Pour lire dans un tube, on utilise la fonction **read** :

ssize_t read(int *descripteur0*, **void** **buf*, **size_t** *count*);

Le descripteur doit correspondre à la sortie d'un tube, le bloc pointe vers la mémoire destinée à recevoir les octets, et la taille donne le nombre d'octets qu'on souhaite lire. La fonction renvoie le nombre d'octets effectivement lus. Si cette valeur est inférieure à taille, c'est qu'une erreur s'est

produite en cours de lecture (par exemple la fermeture de l'entrée du tube suite à la terminaison du processus qui écrit).

Dans la pratique, on peut transmettre un buffer qui a une taille fixe. L'essentiel est qu'il y ait exactement le même nombre d'octets en lecture et en écriture de part et d'autre du pipe. La partie significative du buffer est terminée par un ' \0' comme pour n'importe quelle chaîne de caractère.

Il faut noter que les fonctions read et write permettent de transmettre uniquement des tableaux d'octets. Toute donnée (nombre ou texte) doit être convertie en tableau de caractère pour être transmise, et la taille des ces données doit être connue dans les deux processus communicants.

2.4 Fermer un tube

Lorsque nous utilisons un tube pour faire communiquer deux processus, il est important de fermer l'entrée du tube qui lit et la sortie du tube qui écrit. En effet, il faut que le noyau voie qu'il n'y a plus de processus disposant d'un descripteur sur l'entrée du tube. Ainsi, dès qu'un processus tentera de lire à nouveau, il lui enverra EOF (fin de fichier).

Pour fermer une entrée ou une sortie :

int close(int descripteur);

2.5 Questions

1 - Ecrire un programme tube_1.c qui crée un tube, puis crée un fils, attend la mort de son fils pour aller lire dans le tube.

2 - Faire évoluer tube_1.c en tube_2.c pour apporter la fonctionnalité suivante :

- Le fils récupère les messages à envoyer à son père via une saisie de l'utilisateur, et se termine quand le message envoyé est AU REVOIR.
- Le père ne se terminera que lorsque le fils lui aura envoyé le message AU REVOIR.

3 - faire évoluer tube_2.c en tube_3.c pour apporter la fonctionnalité suivante :

- Le père répond à son fils via un autre tube, en réaction au message reçu. Il envoie toujours la même réponse, après une attente de 2 secondes;il n'y a pas de saisie opérateur coté serveur.
- Le fils attend d'avoir reçu la réponse de son père pour envoyer le message suivant.

3 Les tubes nommés

Le tube nommé repose sur le même principe que le tube anonyme (tube de communication entre deux processus, dans un seul sens), mais permet à deux processus sans lien de parenté d'échanger. Pour cela, la technique des tubes nommés se repose sur le système de fichiers UNIX, en créant un fichier spécial appelé fifo.
Lié au système d'exploitation, un tube nommé doit être détruit explicitement (comme un fichier) quand le processus qui l'a créé meurt.

Pour créer un tube nommé, on utilise la fonction mkfifo de la bibliothèque sys/stat.h.

```
NAME
       mkfifo, mkfifoat - make a FIFO special file (a named pipe)

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

       int mkfifo(const char *pathname, mode_t mode);
```

La fonction **mkfifo**() crée un fichier spécial FIFO (tube nommé) à l'emplacement *pathname*.
mode indique les permissions d'accès. Ces permissions sont modifiées par la valeur d'**umask** du processus : les permissions d'accès effectivement adoptées sont **(***mode* **& ~umask)**.

Un fichier spécial FIFO est semblable à un tube (pipe), sauf qu'il est créé différemment. Plutôt qu'un canal de communication anonyme, un fichier FIFO est inséré dans le système de fichiers en appelant **mkfifo**().
Une fois qu'un fichier FIFO est créé, n'importe quel processus peut l'ouvrir en lecture ou écriture, comme tout fichier ordinaire. En fait, il faut ouvrir les deux extrémités simultanément avant de pouvoir effectuer une opération d'écriture ou de lecture. L'ouverture d'un FIFO en lecture est généralement bloquante, jusqu'à ce qu'un autre processus ouvre le même FIFO en écriture, et inversement.

Lorsque des processus échangent des données par le biais d'une file FIFO, le noyau transfère les informations de manière interne, sans passer par une écriture réelle dans le système de fichiers. Ainsi, le fichier spécial FIFO n'a pas de véritable contenu, c'est essentiellement un point de référence pour que les processus puissent accéder au tube en employant un nom dans le système de fichiers.
Un processus peut ouvrir une FIFO en mode non bloquant. Dans ce cas, l'ouverture en lecture seule réussira même si personne n'a encore ouvert le côté écriture. L'ouverture en écriture seule échouera avec l'erreur **ENXIO** (aucun périphérique ou adresse) si l'autre extrémité n'a pas encore été ouverte.

NB : Sous Linux, l'ouverture d'une file FIFO en lecture et écriture réussira aussi bien en mode bloquant que non bloquant. POSIX ne précise pas ce comportement. Ceci peut servir à ouvrir une FIFO en écriture, même si aucun lecteur n'est prêt. Un processus qui utilise les deux côtés d'une FIFO pour communiquer avec lui-même doit être très prudent pour éviter les situations de blocage.

Comme pour un fichier ordinaire, le fichier doit être ouvert (open), on peut y écrire (write), y lire (read), et le fermer (close).

La programmation avec les tubes nommés est donc semblable à celle avec les tubes ordinaires. La différence se situe au niveau création: on utilise mkfifo() -au lieu de pipe()- qui fixe aussi le mode de protection. En

outre, comme c'est un fichier "réel", un tube nommé doit être ouvert par open() qui fixera le type d'ouverture.
La commande

fd = open("canal", O_WRONLY);

ouvre le fifo canal en écriture pour retourner son descripteur.

3.1 Questions

1 Ecrire le programme tube_srv_n1.c qui réalise les opérations suivantes :

- Création d'un tube nommé
- Ecriture d'un message dans le tube nommé
- attente de quelques secondes (pour avoir le temps de lancer l'exécution du client)
- Fin du programme

2 Ecrire le programme tube_cli_n1.c qui réalise les opérations suivantes :

- Ouverture du tube nommé créé par tube_srv_n1.c
- Lecture du message présent dans le tube nommé
- Fin du programme

3 faire évoluer les 2 programmes (tub_srv_n2.c et tub_cli_n2.c) pour que

- le tube nommé soit dans l'autre sens (c'est le client qui écrit, le serveur qui lit)
- le message écrit par le client est issu d'une saisie utilisateur;le client s'arrête quand le message saisi par l'utilisateur est PAUSE
- le serveur s'arrête quand le message lut est PAUSE
- Exécutez une instance du serveur, 2 instances du client (dans 3 fenêtres terminal différentes).

4 Testez avec plusieur clients, et mettez fin prématurément au serveur (CTRL/C)… que remarquez-vous ? Pourquoi ?

5 Que faudrait-il mettre en place pour exploiter un véritable dialogue clients /serveur ?

5 Problème (séance 2 du TP N°4)

Il s'agit d'écrire une application client/serveur de "chat" de type broadcast.

Le serveur doit pouvoir répondre à des sollicitations de nombreux clients. Il doit recevoir des messages de la part d'un client, et re-transmettre ce message à tous les clients en ligne.

Le client peut envoyer des messages au serveur, saisi par l'utilisateur à la console.

Le serveur est lancé par un utilisateur qui lui communique une valeur de clef. Grâce à cette valeur de clef, le serveur va cacher quelque part les informations nécessaires aux clients pour se connecter, si ce client connaît lui-même la clef du jour, qui lui est passée en argument au lancement.

Exemple :
lancement du serveur :
./serveur 1234
Lancement d'un client :
./client 1234

1234 étant la "clef du jour".

Proposez une solution à cette problématique en vous appuyant sur les techniques suivantes :

- Les signaux
- les tubes nommés
- les threads

Avant de vous lancer dans le code, faites un schéma de votre solution, permettant de mettre en évidence les moyens d'échange entre chaque client et le serveur.

NB : les clients ne sont pas des fils du serveur.