R3.05 - TP 7
IPC - Sémaphores

Introduction aux IPC

Les **IPC**, qui signifient **I**nter **P**rocess **C**ommunication, sont un terme générique pour désigner des mécanismes permettant aux processus de communiquer entre eux.

Et on connaît déjà aussi d'autres mécanismes permettant d'échanger des informations : les tubes et les signaux. Les IPC viennent donc les compléter.

Il y a 3 types d'IPC : les **queues de messages**, les segments de **mémoire partagée** et les **sémaphores**.

Queues de messages

Les queues de messages (Message Queues, **`MSG`** en terminologie IPC Unix) sont un mécanisme qui s'apparente aux tubes mais qui apportent plus de souplesse en permettant de typer les messages pour créer des canaux multiples dans une même queue. Elles apportent aussi la possibilité d'avoir plusieurs écrivains et plusieurs lecteurs.

Nous n'étudierons pas les queues de message cette année.

Segments de mémoire partagée

Les segments de mémoire partagée (Shared Memories, **`SHM`** en terminologie IPC Unix) sont un mécanisme permettant à plusieurs processus de lire et d'écrire dans une même zone mémoire commune (partagée), ce qui n'est pas possible avec un **`malloc()`** par exemple.

Les SHM apportent aussi une persistance des données quand le processus s'arrête.

Nous n'étudierons pas les mémoires partagées cette année.

Sémaphores

Les sémaphores (**`SEM`** en terminologie IPC Unix) sont un mécanisme permettant d'aider les processus à accéder à des ressources partagées et à gérer l'exclusivité de l'accès en cas de concurrence.

Prenons quelques exemples :

- Un fichier doit être modifié par des processus s'exécutant en parallèle, mais il y a un risque que plusieurs d'entre eux tentent de le faire en même temps.

- Une imprimante doit imprimer des documents mais elle ne peut pas recevoir de nouvelle demande tant qu'une impression en cours n'est pas terminée.

Pour prendre un dernier exemple, non informatique cette fois-ci : les feux de signalisation routière sont aussi des sémaphores, ils indiquent aux véhicules s'ils peuvent passer ou pas. Il viennent généralement à plusieurs, combinés entre eux pour assurer les usagers que s'ils obtiennent un droit de passage, les autres qui croisent leur route seront bloqués. La route est une ressource commune et partagée, comme une imprimante ou un fichier.

Les sémaphores sont les seuls composants IPC que nous étudierons en détail cette année.

Principes communs aux IPC

Les IPC ne sont pas des fichiers (tout est résident en mémoire).

Les IPC sont des fonctionnalités de l'OS. A ce titre, ce sont des outils pour le développeur système.

Même si ce ne sont pas des fichiers, les IPC sont régis par les mêmes types de droits que ceux qui régissent les fichiers (**`rwx`** sur **`UGO`**). Ils sont aussi attachés à un propriétaire (celui qui crée l'IPC) et un groupe propriétaire. De ce fait, on peut les considérer comme des fichiers spéciaux résidant uniquement en mémoire.

Tout comme les fichiers, qui ont un identifiant unique[1] dans le FS, les IPC sont identifiés par une clé numérique unique (une par IPC créé).

Les IPC se manipulent (lecture et écriture) à l'aide de commandes qui nécessitent un **handler** d'IPC. On parle bien ici de commandes (Shell), mais on peut faire une analogie avec le handler qui permet de manipuler les données d'un fichier en C (dans le cas des fichiers en C, ce handler est ce que retourne la fonction **`open()`** par exemple).

Les IPC sont persistants en mémoire après l'arrêt des processus qui les ont créés. C'est une caractéristique très intéressante. Évidemment, un arrêt de l'OS fait disparaître les IPC, puisqu'ils n'existent qu'en mémoire.

Clé

Pour créer un IPC ou pour récupérer l'accès à un IPC déjà créé, il faut déterminer une **clé unique**.

On pourrait choisir une clé de façon aléatoire ou encore on pourrait s'accorder sur une clé choisie de manière absolue et arbitraire. Sans entrer dans les détails du pourquoi, ce n'est absolument pas une bonne pratique.

C'est pourquoi l'OS met à notre disposition un mécanisme pour calculer une clé unique. Le calcul d'une clé se fait à partir d'un fichier du FS. Ce calcul n'utilise pas le contenu du fichier mais s'appuie sur son numéro d'inode. Les avantages sont les suivants :

[1] L'inode.

- L'inode étant unique dans le FS, le calcul de la clé, qui s'appuie sur cette information unique, produit ainsi une valeur de clé qu'on s'assure être unique.
- L'inode étant une information publique que tout processus a le droit de connaître, on n'a même pas besoin d'avoir des droits particuliers (lecture ou écriture) sur le fichier utilisé pour calculer la clé.

On aurait pu utiliser directement l'inode comme clé mais la fonction de calcul de clé permet, à partir d'un seul inode, de créer une série de clés toutes différentes et uniques, comme on va le voir ci-dessous.

La fonction système qui calcule une clé s'appelle **`ftok()`** et signifie "**f**ile **to k**ey", autrement dit : crée une clé à partir d'un fichier, et plus précisément, comme on l'a vu, à partir de son inode. Après un petit **`man ftok`**, voici un exemple d'utilisation :

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

key_t cle;
cle = ftok(".", 0);
```

Le 1er paramètre passé à **`ftok()`** est le chemin vers un fichier ou un dossier du FS. Dans cet exemple, on a choisi simplement le dossier courant, celui où on sera quand on va lancer le programme. C'est pratique car on est sûr que ce "fichier" existe et il est commun à tous les programmes qui sont lancés depuis cet endroit. Ainsi, si on a plusieurs programmes qui doivent accéder aux mêmes IPC depuis ce dossier, et que chacun utilise cette technique et ce "fichier", qui leur est donc commun, comme support de création de clé, ils obtiendront tous la même clé !

Le 2ème paramètre est une valeur entière qui permet d'obtenir des clés différentes (en fonction de la valeur choisie pour cet entier) à partir du même fichier donné en 1er paramètre. On choisit souvent **`0`** en premier, puis **`1`**, puis **`2`** etc. si on a besoin de clés supplémentaires dans un même programme, chaque clé servant ensuite d'identifiant d'autant d'IPC distincts.

Ainsi, plusieurs programmes peuvent calculer la valeur de la clé (ou des clés si besoin de plusieurs) pour accéder aux IPC qu'ils ont en commun, et cela à partir d'un fichier connu d'eux : leur répertoire de travail commun. Pour le choix de ce "fichier" commun, il faut absolument éviter de choisir un fichier ou un dossier du FS qui n'a aucune relation avec le projet comme par exemple **`/usr`** ou **`/tmp`** ou encore un **`/etc/passwd`** car d'autres programmes pourraient faire ce même mauvais choix et vous risqueriez d'entrer en conflit de clé dans ce cas.

Obtention d'un handler

Calculer une clé ne crée rien du tout.

Il faut maintenant passer par une fonction de création ou de récupération d'un IPC existant pour pouvoir exploiter l'IPC.

Ca se fait au moyen d'une des fonctions suivantes :

- **shmget()** pour les mémoires partagées
- **msgget()** pour les queues de messages
- **semget()** pour les sémaphores

Nous n'allons pas travailler sur les queues de messages ni les mémoire partagées mais pour ne pas vous laisser dans l'ignorance, voici quand même la syntaxe pour ces deux types d'IPC :

```
#include <sys/shm.h>  // Pour shmget()
#include <sys/msg.h>  // Pour msgget()

int handler_shm, handler_msg;
handler_shm = shmget(cle, <taille>, <flags>);
handler_msg = msgget(cle, <flags>);
```

où **cle** est la clé calculée avec un **ftok()** et **<flags>** est un champ de bits combinant les droits **rwx** sur **UGO** de l'IPC créé ou à récupérer. À ces droits on peut combiner des flags complémentaires que vous trouverez dans le **man** de chacune de ces fonctions **???get()**. Un de ces flags est **IPC_CREAT** qui indique qu'on souhaite créer l'IPC. Attention, s'il existe déjà ça provoquera une erreur. Ce flag est donc à ajouter une seule fois, à la création. Exemple :

```
handler = msgget(cle, IPC_CREAT | 0640);
```

va créer[2] une queue de message qui sera la propriété de l'utilisateur qui exécute ce programme et (par défaut) celle de son groupe principal. Cette queue de message aura les droits **640** c'est-à-dire **rw** pour **U**ser, **r** pour **G**roup et rien pour **O**ther. Notez qu'un droit **x** n'a aucune signification dans le cas des IPC, ce n'est pas la peine d'essayer d'en mettre un (donc pas de **0750** par exemple).

Notez qu'une même valeur de **cle** peut servir à créer une mémoire partagée, une queue de message et un sémaphore en même temps et sans conflit.

Nous allons maintenant entrer dans le détail uniquement pour les sémaphores.

[2] Car présence du flag **IPC_CREAT**

Sémaphores

Nous nous intéressons uniquement aux sémaphores dans ce TP. Voici un exemple de code de création d'un sémaphore :

```
#include <sys/sem.h>  // Pour semget()

int handler;
handler = semget(cle, 1, IPC_CREAT | 0640);
```

Vous aurez noté que le second paramètre permet d'indiquer combien de sémaphores on souhaite créer (ici : **1**). En fait, on ne crée pas **un** sémaphore mais **un lot de** sémaphores. Le handler nous donne accès à ce lot et le développeur peut faire usage d'autant de sémaphores que le lot en contient. Chaque sémaphore est numéroté de **0** à **N-1** où **N** est la valeur du second paramètre de **semget()**.

Ensuite, pour récupérer[3] un handler sur un lot de sémaphores déjà créé, il faut simplement omettre le **IPC_CREAT** mais les droits **doivent obligatoirement être les mêmes** sinon ça provoquera une erreur !

Une fois le lot de sémaphores créé ou récupéré, on peut faire des actions sur un sémaphore du lot.

Mais avant de voir les actions possibles, revenons un peu sur les exemples qu'on a donnés en introduction des sémaphores : des accès concurrents à un fichier et des accès concurrents à une imprimante.

Dans ces deux situations, on voit bien qu'il s'agit d'accès exclusifs à une ressource, un seul processus peut avoir un droit d'accès à un instant T et quand il a terminé, il va rendre son autorisation d'accès pour qu'un autre processus, peut-être déjà en attente, puisse accéder à la ressource.

Il est possible, et même fréquent, d'avoir des ressources disponibles en quantité non binaire (**0** ou **1**, "j'ai le droit ou je ne l'ai pas"). Un exemple non informatique[4] est l'accès à un parking payant : il y a un certain nombre de places et les véhicules peuvent passer la barrière tant qu'il reste des places disponibles. Un véhicule "prend" une ressource (une place) en obtenant un sémaphore et libère ce sémaphore (sa place) quand il quitte le parking.

Les actions sur un sémaphore sont donc simplement l'acquisition ou la libération **d'une certaine quantité** de ressources gérées par ce sémaphore. Cette quantité peut être **1** mais peut éventuellement être différente de **1**. Dans notre exemple de parking, on

[3] Dans un autre programme ou lors d'un lancement ultérieur du programme créateur du sémaphore.
[4] Qui pourrait très bien être informatisé à l'aide de sémaphores IPC.

pourrait considérer qu'un minibus ou un camping-car occupe **2** places et consomme donc une quantité de **2** dans le sémaphore.

En fait, l'acquisition et la libération sont une seule et même action où la quantité est négative dans un cas (l'acquisition) et positive dans l'autre cas (la libération).

L'acquisition ne peut se faire **que s'il reste une quantité suffisante** dans le sémaphore.

La libération **peut toujours se faire** et même, chose qui peut être surprenante, elle peut libérer une quantité plus importante que celle qui avait été acquise. Il n'y a aucune contrainte dans le sens de la libération. De cette façon, on peut imaginer qu'un processus augmente, par son travail, la quantité d'une ressource donnée et donc libérée par lui.

Passons maintenant à la pratique.

Actions de consommation et de libération

La fonction qui permet de faire des actions sur la quantité de sémaphore est **`semop()`**[5].

La syntaxe est la suivante :

```
int ret;
struct sembuf sop;
sop.sem_num = <numéro>;          // Le numéro du sémaphore. Ex : 0
sop.sem_op = -1;                 // On va consommer une quantité de 1
sop.sem_flg = 0;
ret = semop(handler, &sop, 1);
```

Le paramètre **`<numéro>`** (qui est à compléter dans l'exemple précédent) est le numéro du sémaphore dans son lot, en commençant, comme toujours en C, à **`0`**. Donc ce sera **`0`** dans le cas d'un lot d'un seul sémaphore.

Pour consommer, on fait une opération négative (par exemple **`-1`**), pour libérer on fera une opération positive (par exemple **`+1`**).

Laissez le **`sem_flg`** à **`0`**.

En consommation, cette fonction est bloquante s'il n'y a pas une quantité suffisante et elle restera en attente jusqu'à ce que la quantité puisse être satisfaite. Ca signifie aussi une chose intéressante : si la quantité disponible dans le sémaphore ne satisfait pas à la demande mais satisfait à la demande d'un autre processus, ce dernier obtiendra sa quantité de sémaphore, même s'il est arrivé après l'autre processus en attente.

Deadlock

Un deadlock est une situation où plusieurs processus se bloquent mutuellement.

Par exemple, si deux processus **A** et **B** requièrent tous les deux un sémaphore **X** et un sémaphore **Y**, et que le processus **A** a déjà la main sur le sémaphore **X** mais que le

[5] **sem**aphore **op**eration

processus **B** a la main sur le sémaphore **Y**. Dans ce cas, les deux processus vont se retrouver inter-bloqués dans l'attente que l'autre libère le sémaphore qu'il détient déjà.

Pour éviter cette situation, il est possible d'effectuer des actions de consommation et de libération sur plusieurs sémaphores en un seul appel système qui se fera de façon atomique.

Un appel atomique signifie que les opérations sont faites uniquement quand elles peuvent être toutes effectuées, sinon l'appel reste bloqué tant que la situation n'est pas favorable. On ne peut pas se retrouver dans une situation où seules certaines opérations seraient effectuées et que l'appel système soit bloqué en attente que les autres opérations se réalisent. Ceci résout donc notre problème de deadlock.

Voici comment est construit le code dans ce cas (exemple avec **2** sémaphores) :

```
struct sembuf arr_sop[2];
arr_sop[0].sem_num = 0;
arr_sop[0].sem_op = -1;   // On consomme une qté de 1 sur sémaphore 0
arr_sop[0].sem_flg = 0;
arr_sop[1].sem_num = 1;
arr_sop[1].sem_op = -2;   // On consomme une qté de 2 sur sémaphore 1
arr_sop[1].sem_flg = 0;
ret = semop(handler, arr_sop, 2);
```

C'est le 3ème paramètre de **semop()** qui indique la taille du tableau **arr_sop**. Il vallait **1** dans le 1er exemple et vaut **2** pour les deux actions atomiques qu'on fait dans le second exemple ci-dessus.

Évidemment, un deadlock ne peut apparaître que sur les actions de consommation.

Autres actions

Il est possible de faire d'autres actions comme consulter la valeur d'un sémaphore, l'initialiser à une valeur donnée ou encore le supprimer. Il existe ainsi une quinzaine d'actions possibles.

Ces actions se font en utilisant la fonction **semctl()**[6]. Voici un exemple pour supprimer un lot de sémaphores :

```
int ret;
ret = semctl(handler, 0, IPC_RMID, 0);
```

[6] **sem**aphore **c**on**t**ro**l**

Certaines actions peuvent se faire sur un unique sémaphore et d'autres, comme cet **`IPC_RMID`**, agissent sur le lot entier. Consultez le **`man semctl`** pour du détail sur chaque action.

Initialisation, un cas particulier

Vous avez noté qu'on vient de parler d'initialisation d'un sémaphore en utilisant **`semctl()`**.

Le cas de l'initialisation est un cas particulier qui peut être réalisé de deux façons différentes. Par défaut, quand vous créez un sémaphore il est initialisé à une valeur inconnue, vous devez donc l'initialiser (pas le "consommer" ou le "libérer", c'est bien "initialiser") à la valeur qui vous convient.

Si c'est votre programme qui crée le sémaphore, en tant que créateur vous avez la possibilité de l'initialiser à **`0`** et ainsi considérer que vous disposez alors immédiatement du sémaphore pour vous, comme si vous l'aviez créé puis consommé. Tout autre processus qui essaie de prendre la main dessus se retrouve donc bloqué immédiatement. Ça peut être une sécurité pour éviter de vous faire couper l'herbe sous le pied ! Vous le libérerez alors (un **`+1`** avec un **`semop()`**) quand vous aurez terminé avec lui.

Mais vous pouvez aussi décider de l'initialiser à une valeur de départ, par exemple **`1`** ou une autre valeur supérieure en fonction du besoin et de la quantité de ressources que ce sémaphore "protège". Dans ce cas, un autre processus pourra prendre immédiatement la main sur le sémaphore, et potentiellement avant vous-même. C'est à vous de voir, en fonction du rôle de ce sémaphore, vous seul savez ce qui est acceptable et souhaitable.

Si vous voulez initialiser un sémaphore, voici le code pour mettre à **`10`** le 4ème sémaphore[7] d'un lot (d'au moins 4 sémaphores, évidemment) :

```
int ret;
ret = semctl(handler, 3, SETVAL, 10);
```

Commandes ipcs et ipcrm

La commande **`ipcs`** permet d'afficher des informations sur les IPC. Sans option, elle affiche tous les types d'IPC. Consultez le **`man`** pour voir les options, notamment pour cibler uniquement les sémaphores par exemple.

La commande **`ipcrm`**[8] permet de supprimer un IPC. Voici la syntaxe pour supprimer un lot de sémaphores :

```
ipcrm -s <ID>
```

[7] Le 4ème est le numéro **`3`** car on commence à zéro.
[8] Vous devinerez par vous même la signification 🙂

où **<ID>** peut être récupéré dans la liste affichée par la commande **ipcs**.

Certains IPC ne sont réellement supprimés que lorsque plus aucun processus ne les utilise (Queues de messages et Mémoires partagées). Pour les sémaphores, c'est immédiat.

A noter aussi que si vous recréez un lot de sémaphore en utilisant la même clé, l'**ID** changera et seul un redémarrage de l'OS permettra de revoir des **ID** déjà utilisés.

Lors de vos tests, dans les exercices suivants, vous utiliserez cette commande **ipcrm** pour supprimer un lot de sémaphores quand ce sera nécessaire.

Exercices

Pour les exercices suivants, retenez votre envie d'aller chercher sur Google une solution qui ne vous aidera pas à comprendre en faisant un simple copier-coller. Vous disposez du nécessaire dans les pages qui précèdent et, comme toujours, votre enseignant.e est votre "Google à domicile". On n'est pas payés pour ouvrir la porte et allumer la lumière, mais aussi pour vous aider à comprendre !

Exercice 1

Pour ce 1er exercice, on vous donne quelques indices sur les fonctions à utiliser.

Ecrivez un programme **exo1-crea.c** qui effectue les actions suivantes :

- Créer un (lot de **1**) sémaphore : **ftok()** + **semget()** avec **IPC_CREAT**.
- Initialise le sémaphore **0** à la valeur **1** : **semctl()** avec **SETVAL**.

Compilez, testez. Utilisez **ipcs** pour vérifier et **ipcrm** pour supprimer vos essais.

Pour passer à la partie suivante, ne supprimez pas votre sémaphore.

Ecrivez un programme **exo1-conso.c** qui effectue les actions suivantes :

- Ouvre un handler sur le sémaphore créé par **exo1-crea** : **ftok()** + **semget()** sans **IPC_CREAT**, mais avec les mêmes droits !
- Dans une boucle sans fin :
    - Afficher le texte **"Avant conso"**
    - Consommer une quantité de **1** sur le sémaphore
    - Afficher le texte **"Après conso"**
    - Afficher le texte **"Prêt à libérer ?"**
    - Lire une réponse au clavier (peu importe la réponse)
    - Afficher le texte **"Avant libération"**
    - Libérer une quantité de **1** sur le sémaphore
    - Afficher le texte **"Après libération"**

Compilez et exécutez ce programme deux fois en parallèle dans deux terminaux distincts. Vous devez observer un blocage de l'un tant que vous n'avez pas saisi de réponse à la question **`"Prêt à libérer ?"`** dans l'autre et réciproquement.

Si vous ne comprenez pas, faites vous aider par l'enseignant.e.

Exercice 2 - Synchronisation

Un processus **`master`** distribue du travail à un ensemble de processus **`workers`**.

Le temps requis à chaque **`worker`** pour réaliser sa tâche est aléatoire entre **`3`** et **`8`** secondes.

Le processus **`master`** ne peut distribuer sa tâche que si un **`worker`** est disponible.

Le **`master`** agit dans une boucle sans fin. Pour l'arrêter vous ferez un **`CTRL+C`**.

Vous avez besoin de deux sémaphores : **`A`** et **`B`**. C'est le **`master`** qui doit s'occuper de les créer et de les supprimer sur réception d'un **`SIGTERM`** ou d'un **`SIGINT`** (**`CTRL+C`**)

Le sémaphore **`A`** représente la quantité de **`workers`** disponibles à un instant T. Le **`master`** doit l'initialiser à **`0`** à la création.

Le sémaphore **`B`** représente la quantité de tâches soumises par le **`master`**. Le **`master`** doit l'initialiser à **`0`** à la création.

Quand le **`master`** veut distribuer une tâche il doit :

- Acquérir une quantité de **`1`** sur le sémaphore **`A`**
- Libérer une quantité de **`1`** sur le sémaphore **`B`**
- Boucler

Un **`worker`** qui se lance commence par incrémenter le sémaphore **`A`** pour indiquer qu'il se joint à la partie !

Puis il prend et traite des tâches en boucle de la façon suivante.

Pour qu'un **`worker`** puisse prendre une tâche il doit :

- Acquérir une quantité de **`1`** sur le sémaphore **`B`**
- Faire la tâche qui sera simulée par une pause entre **`3`** et **`8`** secondes. Utilisez **`srand()`** et **`rand()`** et appelez à l'aide si besoin !
- Libérer une quantité de **`1`** sur le sémaphore **`A`** pour signifier sa disponibilité pour une nouvelle tâche.

Si un **`worker`** reçoit un **`CTRL+C`**, il doit terminer sa tâche mais ne pas libérer le sémaphore **`A`** car il va arrêter de participer aux travaux. Pour rappel, il avait incrémenté **`A`** en entrant dans le jeu.

Vous devez mettre des affichages partout où c'est utile pour comprendre le déroulé des actions.

Exécutez votre **`master`** en 1er. Dans des terminaux séparés, lancez un premier **`worker`**, puis si tout se passe bien, lancez-en un second, puis un troisième. Plus il y a de **`workers`**, plus vous devez observer que le **`master`** distribue rapidement ses tâches.