**Les fonctionnalités IPC sous Unix**

**Avertissement**

Au regard de mes problèmes de santé quelque peu rémanents, il ne m'a pas été possible d'aller aussi loin que je l'aurais souhaité dans l'élaboration de ce cours. Je vous prie de bien vouloir m'en excuser. Vous pourrez trouver des renseignements complémentaires dans les [transparents du cours](http://brunogarcia.chez.com/Unix/DocsPwp/Processus/index.htm).

**Introduction aux fonctionnalités IPC**

Tout d'abord, le terme IPC signifie *“Inter processus communication”* donc, nous sommes confrontés à des fonctionnalités permettant à différents processus d'échanger des informations. Les IPC sont toutefois très différents des tubes :

1. Ils sont totalement détachés du système de fichier, ce qui est très rare dans le monde Unix. Ceci veut dire que les fonctions d'accès seront particulières aux IPC
2. Ils permettent de faire communiquer des processus *sans lien de parenté* mais *situés sur la même machine*. Deux fonctionnalités de communication sont fournies :
   1. Les segments de mémoire partagée, orientées, comme leur nom l'indique, vers le partage d'informations communes
   2. Des boîtes aux lettres, où les processus s'envoient des messages
3. Afin de sécuriser l'accès aux segments de mémoire partagée, mais également à toute ressource critique, un mécanisme de synchronisation orienté *sémaphores* est proposé.

**Particularités communes aux fonctionnalités IPC**

Toutes les fonctionnalités IPC sont créées à l'aide d'une clef numérique. Accéder à une IPC suppose, soit de connaître son identificateur, soit de disposer de la clef numérique qui permet de récupérer l'identificateur. Ce système n'est pas très heureux car le créateur est obligé de rendre publique d'une manière ou d'une autre les clefs qu'il désire partager.

La question est : comment les mettre à disposition ? l'un des moyens les plus utilisés consiste à utiliser une clef particulière, lui associer un segment de mémoire partagée avec une structure particulière où tout le monde ira chercher les informations nécessaires. Dans ce cas, il est toujours possible d'utiliser une clef qui serait souhaitée par une autre application. Il est également possible d'utiliser un fichier comme stockage de clefs.

L'autre problème, sur lequel nous reviendrons un peu plus tard, concerne le choix des clefs au moment de la création d'une facilité.

**Les commandes shell associées aux IPC**

Deux commandes shell particulièrement importantes sont associées aux IPC.

ipcs

Cette commande permet d'afficher la liste des fonctionnalités IPC présentes sur le système.

*ipcrm*

Utilisée pour supprimer une fonctionnalité qui a été oubliée par les programmes qui l'utilisaient

**Listage des IPC :** ipcs

Voici la syntaxe complète de la commande ipcs

ipcs [-s] [-m] [-q] [-u utilisateur]

Par défaut, la commande ipcs liste l'ensemble des fonctionnalités présentes sur un système. Les options ne servent qu'à limiter l'étendue des informations fournies.

* Par type de facilité :
  + -q pour afficher les boîtes aux lettres
  + -m pour afficher les segments de mémoire partagée
  + -s pour afficher les ensembles de sémaphore
* Par utilisateur (le créateur) avec l'option -u utilisateur. Bien que fort utile, cette option n'est pas disponible, loin de là, sur tous les systèmes. En particulier, certaines versions de AIX ne la proposent pas …

L'exemple suivant illustre le genre de listing fourni par ipcs :

bipro: ipcs

IPC status from /dev/mem as of Mon May 25 10:12:27 DFT 1998

T ID KEY MODE OWNER GROUP

Message Queues:

q 0 0x4107001c -Rrw-rw---- root printq

Shared Memory:

m 0 0x0d050132 --rw-rw-rw- root system

m 1 0x430530a2 --rw-rw-rw- ouorou labo

m 126978 0x6605d9da --rw-rw-rw- darmont labo

m 192515 0x0000029a --rw-rw-rw- bruno labo

Semaphores:

s 4096 0x4d09201d --ra-ra---- root system

s 1 0x620500eb --ra-r--r-- root system

s 2 0x0105007e --ra------- root system

s 57347 0x0000029a --ra-ra-ra- bruno labo

**Suppression d'IPC :** ipcrm

La commande ipcrm permet de supprimer une facilité dont on connait soit la clef, soit l'identificateur. Comme une même clef (ou un même identificateur) peut être associé aux trois variétés d'IPC, il sera nécessaire de spécifier le type d'IPC à détruire en paramètre.

La syntaxe complète est la suivante :

ipcrm [-q|-Q|-m|-M|-s|-S] nombre

La convention est simple, les options en lettres minuscules dénotent les identificateurs ; réciproquement, les options en lettres majuscules sont associées aux clefs. Donc :

|  |  |
| --- | --- |
| -q | Boîte aux lettres par id |
| -Q | Boîte aux lettres par clef |
| -m | Mémoire partagée par id |
| -M | Mémoire partagée par clef |
| -s | Sémaphores par id |
| -S | Sémaphores par clef |

Donc, si vous m'avez bien suivi, la commande ipcrm -q 12045 détruit la file de messages d'identificateur 12045 alors que ipcm -S 0x1234 va supprimer l'ensemble de sémaphores associé à la clef 0x1234.

Permettez moi deux petites remarques de rien du tout :

* Rien n'oblige à utiliser la représentation décimale pour les identificateurs et la base 16 pour les clefs. Toutefois, cela s'avère pratique car c'est le format de listage fourni par ipcs
* Un utilisateur ne peut détruire que les fonctionnalités qu'il a créées

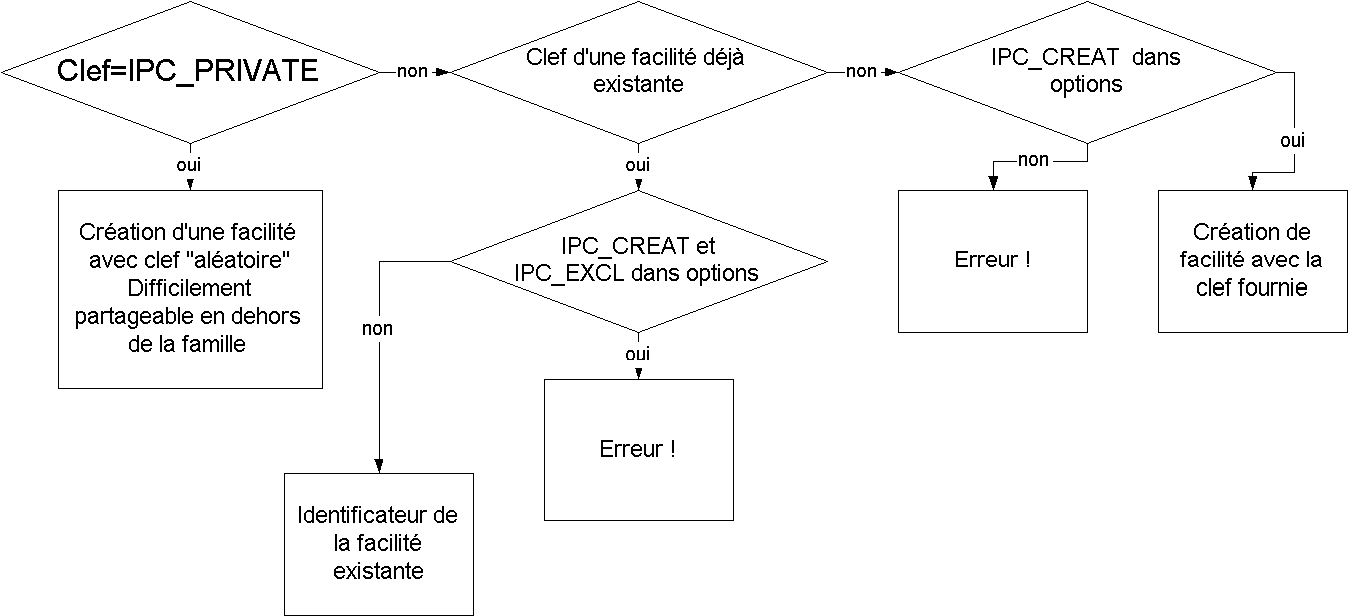
Lorsque l'on travaille avec les IPC, il est agréable de disposer d'un script automatique permettant de supprimer toutes les fonctionnalités IPC que l'on a créées et non encore détruites. Un tel script utilisant [awk](http://brunogarcia.chez.com/Unix/Docs/awk.html)vous est proposé en [annexe de la correction du TP #4](http://brunogarcia.chez.com/Unix/SolutionsTPS/solutiontp4.html#ipcclean).

**La programmation des IPC**

**Création des fonctionnalités**

C'est assurément l'un des points cruciaux de l'opération, car c'est là que l'on doit choisir, d'une part la clef à laquelle sera rattachée une fonctionnalité et d'autre part les options de création. En fait les fonctions permettant de créer une fonctionnalité à partir d'une clef et celle autorisant un utilisateur à accéder à une fonctionnalité connaissant son type et sa clef sont les mêmes.

La figure suivante résume ce processus de création.



Le prototype général de la méthode de création/accès à une ressource est le suivant :

int [msg|sem|shm]get(key\_t clef, ..., int options)

où les paramètres symbolisés par ... sont dépendants du type de fonctionnalité. Les options sont cruciales pour le bon déroulement de l'opération :

1. Si vous désirez créer une facilité, IPC\_CREAT doit apparaître dans les options. En outre, je vous conseille de lui adjoindre (par un "ou" binaire) la constante IPC\_EXCL qui assure que l'on ne chercher pas à recouvrir une fonctionnalité déjà existante. En outre, vous voudrez sans doute rajouter des droits d'accès. Le codage est similaire à celui des droits sur les fichiers, par exemple 0642 ouvre des droits en lecture et écriture pour le propriétaire, en lecture seulement pour les membres de son groupe et en écriture seulement pour le reste du monde (doit être un peu toqué lui:)).
2. Si vous souhaitez accéder à une facilité, vous ne mettrez probablement que les droits avec lesquels vous comptez accéder à l'IPC. Ces derniers sont codés comme pour l'accès à un fichier. Bien entendu, comme ils ne concernent qu'un processus particulier, vous ne spécifiez que les droits pour l'utilisateur courant. Par exemple, 0600 pour accéder en lecture et en écriture. Les droits d'accès sont, bien entendu, limités par les droits concédés au moment de la création.

Lorsque ce type de fonction échoue, la valeur retournée est -1, en cas de succès, la fonction renvoie l'identificateur de la fonctionnalité. Cet identificateur est très important car il sera requis par toutes les autres primivites.

**Les primitives de contrôle**

A l'instar des primitives de création/accès dont le fonctionnement est assez normalisé, il existe une famille de primitives de contrôle dont le but est d'effectuer des opérations de maintenance sur les IPC, telles que :

* Récupérer les informations sur le statut des IPC
* Changer le statut des IPC
* Supprimer une IPC

Le format général des fonctions de contrôle est le suivant :

int [msg|sem|shm]ctl(int identificateur, int operation, ... )

où operation est une constante définissant le type d'action à appliquer sur la fonctionnalité spécifiée par son identificateur. Il existe une grande variété d'actions et la plupart d'entre elles nécessitent des pointeurs vers des structures de renseignement struct msqid\_ds pour les boîtes aux lettres, struct semid\_ds pour les sémaphores ou struct shmid\_ds pour les segments de mémoire partagée.

Les actions les plus courament utilisées sont les suivantes :

|  |  |  |
| --- | --- | --- |
| **Constante** | **Action** | **Remarques** |
| IPC\_RMID | Suppression de fonctionnalité | Le pointeur de structure peut être nul, car il ne s'agit pas d'accèder ou de modifier les informations relatives à une fonctionnalité mais de la détruire |
| IPC\_STAT | Récupération d'informations sur la fonctionnalité |  |
| IPC\_SET | Mise en place d'informations sur la fonctionnalité | Certaines informations récupérées avec IPC\_STAT sont en lecture seulement et ne pourront donc être manipulées par IPC\_SET |

Il est très important de détruire les fonctionnalités inutilisées car elles survivent indépendament des processus. Si votre processus plante avant d'avoir effectué son ménage, il faut alors utiliser ipcs et ipcrm pour les détruire. Vous pouvez également utiliser un script tel que celui présenté dans [l'annexe de la solution du TP #4](http://brunogarcia.chez.com/Unix/SolutionsTps/solutiontp4.html#ipcclean).

**Segments de mémoire partagée**

L'utilisation des segments de mémoire partagée nécessite l'inclusion des fichiers standard <sys/ipc.h> et <sys/shm.h>. Quatre types d'opérations vont être disponibles sur un segment de mémoire partagée :

1. Création du segment ou accès au segment via sa clef
2. Attachement du segment
3. Détachement du segment
4. Contrôle du segment

Par rapport au cas général, les seules opérations spécifiques sont celles d'attachement et détachement.

**L'attachement d'un segment de mémoire partagée**

Par définition, les seules adresses accessibles à un programme sont celles comprises dans son espace d'adressage. L'opération d'attachement consiste à agrandir l'espace d'adressage d'un processus afin que celui-ci englobe le segment de mémoire partagée afin que les données qu'ils contiennent deviennent accessibles. Ce mécanisme est semblable (mais non similaire) à celui utilisé pour rendre accessibles les informations contenues dans une libraire dynamique partagée.

Résumons nous : l'opération d'attachement prend en paramètre l'identificateur d'un segment de mémoire partagée et fournit un pointeur vers le début de ce segment.

La fonction réalisant cette opération est la suivante :

void \*shmat(int identSHM, const void \*adresse, int options)

Détaillons maintenant les diverses composantes de cet appel :

*identSHM*

Il s'agit de l'identificateur du segment de mémoire partagée que l'on désire attacher

*adresse*

la fonction shmat vous permet de spécifier l'adresse à laquelle vous souhaiteriez voir attacher un segment de mémoire partagée. Cette opération est risquée et je ne vous conseille pas de l'utiliser. En outre, rien n'oblige le système à répondre favorablement à votre requête. En particulier, certains systèmes exigent que les adresses d'attachement des segments de mémoire partagée soient alignés sur des frontières de 4, 8 ou 16 octets.

Aussi, je vous conseille de toujours passer ici la constante 0 qui indique au système d'attacher le segment de mémoire partagée là où ça l'arrange !

*options*

Peu d'options sont rééellement disponibles, la plus utile étant SHM\_RDONLY qui indique que le segment est attaché en mode lecture seulement.

*Retour*

Si tout se passe bien, shmat renvoie un pointeur sur le premier octet du segment de mémoire partagée. En cas d'erreur le pointeur de valeur -1 est renvoyé. Attention ! il faut convertir cette valeur en int pour effectuer la comparaison sans risque !

Une fois le segment attaché, vous manipulez la mémoire partagée via le pointeur obtenu comme s'il s'agissait de mémoire propre à votre processus. Aucun garde fou ne garantit que, par exemple, plusieurs processus ne sont pas en train d'effectuer des écritures simultanées. Aussi, il est important de synchroniser l'utilisation de vos segments de mémoire partagée à l'aide, par exemple, de sémaphores.

Le système maintient un compteur du nombre d'attachements sur un segment de mémoire partagée. Tant que ce compteur est non nul, le segment ne peut être détruit. Aussi, est-il très important de rompre l'attachement d'un segment non utilisé. Cette opération connue sous le nom de *détachement* est décrite dans la section suivante.

Remarque : certains systèmes permettent d'attacher plusieurs fois le même segment dans le même processus et ce à des adresses différentes. Cette fonctionnalité, assez dangereuse au demeurant, tant à disparaître des versions les plus modernes des systèmes d'exploitation et ne doit pas être utilisée si la portabilité est importante, et elle l'est toujours !

**Le détachement d'un segment**

Lorsqu'un processus ne se sert plus d'un segment de mémoire partagée, il doit le détacher, c'est à dire le retirer de son espace d'adressage. En effet, tant qu'un processus attache un segment, celui-ci ne peut en aucun cas être détruit.

Remarque : le mort d'un processus détache automatiquement tous les segments qu'il pouvait avoir attacher.

La fonction permettant de détacher un segment est la suivante :

int shmdt(const void \*adresse)

où adresse est le pointeur obtenu lors de l'attachement. Notez bien que l'on utilise pas l'identificateur du segment mais l'adresse d'attachement ; rappelons en effet que certains systèmes autorisent l'attachement multiple d'un segment de mémoire partagée par le même processus.

Typiquement shmdt renvoie 0 si tout s'est bien passé, c'est à dire, virtuellement, tout le temps !

**La destruction d'un segment de mémoire partagée**

La destruction d'un segment se fait grâce à la primitive de contrôle shmctl associée à l'opération IPC\_RMID. La syntaxe complète de shmctl est la suivante :

int shmctl(int identSHM, int operation, struct shmid\_ds \*infos)

Cette fonction à usage général (voir la section consacrée aux fonctions de contrôle en général) n'utilise la structure d'informations de type struct shmid\_ds que si l'operation est IPC\_STAT (recupération d'informations sur un segment particulier) ou IPC\_SET (positionnement d'informations sur un segment particulier). Dans le cas de la destruction d'un segment avec IPC\_RMID, ce paramètre peut très bien être nul.

**Créer ou accéder à un segment de mémoire partagée**

La fonction permettant de créer ou accéder à un segment de mémoire partagée est shmget. Voici sa syntaxe complète :

int shmget(key\_t clef, size\_t taille, int options)

Comme dans le cas général, ce sont les options qui déterminent si l'on fait un accès ou bien une création. Le seul paramètre spécifique aux segments de mémoire partagée est la taille demandée en octets.

**Quelques commandes d'intérêt général**

* Création d'un segment de longueur longueuravec la clef clef tel que tout le monde puisse y lire et y écrire :

ident=shmget(clef, longueur, IPC\_CREAT | IPC\_EXCL | 0666);

* Accès à un segment de longueur longueur avec la clef clef et en accès total :

ident=shmget(clef,longueur,0600);

* Vérification du bon fonctionnement :

if (ident !=-1) /\* tout est ok \*/

* Attachement du segment d'identificateur ident à une adresse choisie par le système à placer dans le pointeur de type char \* pointeur :

pointeur=shmat(ident,0,0);

* Attachement du même segment mais en mode lecture seulement :

pointeur=shmat(ident,0,SHM\_RDONLY);

* Détachement d'un segment présent à l'adresse pointeur :

shmdt(pointeur);

* Destruction d'un segment d'identificateur ident :

shmctl(ident,IPC\_RMID,0);

**Boîtes à lettres**

Les boîtes à lettres implémentent le mécanisme d'une queue de messages où plusieurs processus peuvent écrire mais un seul lire.

**Création/Accès**

La primitive de création/accès est des plus simples, en effet son prototype est le suivant :

int msgget(key\_t clef, int options)

où les options sont celles standard de création de facilité. Il n'y a ni paramètre ni option spécifique aux boîtes aux lettres.

**Contrôle/Destruction**

La primitive de contrôle possède une syntaxe pour le moins alambiquée. Si vous n'utilisez que l'option IPC\_RMID permettant de détruire la boîte, la ligne de commande suivante seule vous sera utile :

msgctl(identificateur, IPC\_RMID,0)

**Structure des messages**

Nous entrons enfin dans le vif du sujet : les messages. Les messages IPC sont typés. Le type est codé sous la forme d'un long placé dans les premiers octets du message. Aussi, un message a la forme suivante :

struct \_TYPE\_MESSAGE

{

long typeMessage;

... toutes donnees necessaires;

};

typedef struct \_TYPE\_MESSAGE TypeStructMessage;

Vous pouvez placer n'importe quel type de données dans un message à l'exception des pointeurs extérieurs. En effet, il est évident qu'un message ne peut pas contenir de références vers des données extérieures car la valeur du pointeur n'a aucune signification pour le destinataire du message. Par exemple, si vous désirez envoyer une chaîne de caractères, il vous faudra envoyer la séquence des caractères et non pas un pointeur vers le premier caractère dans votre espace d'adressage local.

**Envoi de message**

La primitive d'envoi de message a pour prototype :

int msgsnd(int identMSG, const void \*adresseMessage, int longueurMessage, int options);

où identMSG est l'identificateur de la boîte du destinataire. Si un même message est destiné à plusieurs boîtes, il vous faudra répéter autant de fois l'opération qu'il y a de destinataires. Le contrôle du programme passe à l'instruction suivante dès que le message est posé dans la boîte ce qui signifie quasiment instantanément à moins que la boîte du destinataire ne soit pleine auquel cas le processus est bloqué jusqu'à ce que le boîte du destinataire soit suffisament dégagée pour recevoir le message ... à moins que l'option IPC\_NOWAIT ne soit activée.

La primitive msgsnd renvoie 0 en cas de succès, et -1 en cas d'échec.

Le paramètre options est mis à 0 dans la plupart des cas. Il est toutefois possible de lui mettre l'option IPC\_NOWAIT auquel cas msgsnd est non bloquante en cas de boîte aux lettres de destination pleine mais retourne un message d'erreur.

**Réception de message**

La primitive de réception de message a pour prototype :

int msgrcv(int identMSG, void \*adresseReception, int tailleMax, long typeM, int options);

C'est ici qu'intervient le type des messages ! En effet, il est possible de ne recevoir que les messages d'un certain type ! tout en gardant bien à l'esprit que le fonctionnement général est de type FIFO. Voici la signification exact de ce paramètre :

* Si typeM==0 le premier message de n'importe quel type est extrait
* Si typeM > 0 le premier message de type==typeM est extrait
* Si typeM < 0 le premier message de type < typeM est extrait

Comme l'on peut s'y attendre, tailleMax spécifie la taille maximale du message que l'on peut récupérer à l'adresse adresseReception. Il y a toutefois un détail particulièrement dérangeant : le type du message n'est pas compris dans tailleMax. Donc, supposons que l'emplacement mémoire pointé par adresseReception soit de 256 octets. Lorsque l'on récupère un message, le type est extrait avec les autres informations. Le paramètre tailleMax doit donc être fixé à 256 - sizeof(long) sous peine de surprise désagréable ! Mais, que se passe t'il si le message est trop long ? Par défaut, msgrcv retourne un code d'erreur, à moins que MSG\_NOERROR ne soit inclus dans les options, auquel cas, le message est tronqué pour rentrer dans l'espace prévu.

Deux petites remarques s'imposent :

* Un message ne peut être lu qu'une seule fois, même s'il est tronqué
* Un message ne contient pas d'informations identifiant son émetteur. Si l'émetteur désire être identifié, il devra placer lui même des informations dans le message, par exemple, une adresse de retour (clef ou identificateur de boîtes aux lettres)

**Sémaphores**

**Présentation des sémaphores et vocabulaire**

**Notion de sémaphore**

Les sémaphores sont des outils généraux de synchronisation qui servent, par exemple, à gérer l'accès en exclusion mutuelle à une ressource critique. Pour des informations complémentaires, je vous renvoie, une fois de plus j'en suis désolé aux [transparents du cours](http://brunogarcia.chez.com/Unix/DocsPwp/Processus/index.htm).

Rappelons qu'un sémaphore est représenté par une valeur entière. Prenons, par exemple, le cas d'une ressource critique nécessitée par plusieurs processus et dont la quantité est limitée. La quantité de ressource disponible initialement sera donnée par la valeur du sémaphore. Si nous disposons de *n* unités de ressources initialement, alors, la valeur initiale du sémaphore sera *n*.

**Les opérations sur les sémaphores**

Deux grandes opérations sont disponibles sur un sémaphore :

*Opération* ***P(m)****.*

Lorsqu'un processus nécessite *m* unités de ressource pour fonctionner, il va tenter de puiser dans la quantité disponible. La quantité de ressources va donc être décrémentée de *m*. Si après cette opération la quantité de ressources est négative, cela veut dire que le processus ne peut s'exécuter correctement : il est alors suspendu dans l'attente d'une quantité suffisante de ressources.

*Opération* ***V(m)***

Lorsqu'un processus libère *m* unités de ressource, les demandes des processus bloqués sont rééxaminées et certains sont débloqués.

Le tableau suivant résume le fonctionnement de ces deux opérations

|  |  |  |
| --- | --- | --- |
| **Type de l'opération** | **Effet** | **Signification** |
| Opération **P(*m*)** | si ((Valeur-m)<0) alors  Bloquer le processus  sinon Valeur=Valeur-m | Prise de *m* unités de ressource. Si les ressources ne sont pas disponibles, le processus est suspendu |
| Opération **V(*m*)** | Valeur=Valeur+m  si (Valeur >= 0) alors  Tous les processus bloqués  sont rééxaminés pour déblocage ! | Libération de *m* unités de ressource, certains processus suspendus pourront reprendre si les quantités de ressources nécessaires sont à nouveau disponibles |

L'implémentation des sémaphores disponible sous Unix propose bien entendu ces deux opérations ainsi qu'une troisième notée **Z**. Un processus qui effectue une opération **Z** est suspendu jusqu'à ce que la valeur du sémaphore associé soit égale à **Z**éro.

**Un point fondamental**

Il est absolument fondamental de remarquer que **toute opération sur les sémaphores est atomique** En d'autres termes, une fois que le processus entre dans une fonction traitant des sémaphores, le contrôle du processeur ne sera rendu qu'à la sortie de laditte fonction.

**Les ensembles de sémaphores**

Unix propose une extension des sémaphores simples : les ensembles de sémaphores. Quel est l'intérêt d'un tel ensemble ? et bien c'est de pouvoir considérer un ensemble d'opérations sur différents sémaphores de l'ensemble comme une seule opération, c'est à dire de manière atomique.

Par exemple, supposons que vous disposiez d'un ensemble de 2 sémaphores. Vous pourrez réaliser, par exemple, une opération P(1) sur l'un et une opération V(2) sur l'autre sans interruption.

**La programmation IPC des sémaphores**

Nous attaquons ici l'étude des primitives IPC permettant de manipuler les sémaphores. La primitive semop permettra d'effectuer les opérations P, V et Z sur les sémaphores

**Création/Accès à un ensemble de sémaphores**

La fonction, vous vous en doutiez, s'appelle semget, sa syntaxe est la suivante :

int semget(key\_t clef, int nbSemaphores, int options)

Par rapport au processus général de création de facilité, une seule option spécifique est présente : nbSemaphores qui permet de spécifier combien de sémaphores doivent être créés dans le groupe.

**Remarque très importante :** à sa création, un sémaphore a la valeur 0.

**Contrôle/Destruction de sémaphores**

La fonction semctl est un peu différente des autres car elle permet de ne traiter qu'un élément d'un ensemble de sémaphores. Toutefois, la destruction d'un sémaphore entraîne celle de tout l'ensemble !

La syntaxe de semctl est la suivante :

int semctl(int identSEM, int numeroSemaphore, int operation, ... autres parametres eventuels);

La valeur du paramètre operation détermine la signification de numeroSemaphore et des paramètres éventuels. Permettez moi de vous rappeler que la valeur de semnum n'est pas prise en compte si operation==IPC\_RMID. Vous pouvez donc mettre n'importe quoi (porte'nawak pour parler Tchoum:)) et ne vous en privez pas ! c'est pas si courant !

**Exercice :** à l'aide de man, découvrez comment obtenir la valeur d'un sémaphore.

**Opérations sur les sémaphores**

Nous attaquons ici le coeur de l'utilisation des sémaphores : la fonction permettant de réaliser des ensembles d'opérations *P*, *V* et *Z* sur les ensembles de sémaphores ! . La syntaxe générale est la suivante :

int semop(int identSEM, struct sembuf \* ensembleOps, int nbOps)

où chaque structure struct sembuf permet de définir une opération sur un sémaphore particulier de l'ensemble. Elle a la définition suivante :

struct sembuf

{

unsigned short int sem\_num; /\* Numero du semaphore sur lequel s'applique l'operation \*/

short sem\_op; /\* Operation P, V ou Z \*/

short sem\_flg; /\* Options sur l'operation \*/

};

Chaque opération est ainsi indépendante et peut même avoir des options particulières. Voici quelques explications concernant chacun des champs de la structure :

*sem\_num*

Numéro du sémaphore sur lequel va s'appliquer l'opération. Dans un ensemble de sémaphores, les numéros commencent à 0.

*sem\_op*

C'est le champ qui détermine l'opération, les valeurs possibles sont les suivantes :

*sem\_op=0*

Opération *Z*

*sem\_op > 0*

Opération *V(sem\_op)*

*sem\_op < 0*

Opération *P(sem\_op)*

*sem\_flg*

Options sur l'opération. La plus intéressante est SEM\_UNDO qui permet de fixer des valeurs d'ajustement sur les sémaphores. Son étude est hors du propos de cet exposé. Notez que l'on peut rendre toutes les opérations sur sémaphores non bloquantes en précisant IPC\_NOWAIT en option.

**Micro exemple de** semop

Le fragment de code suivant effectue une opération P(2) sur le premier sémaphore, un Z sur le second, et un V(1) sur le troisième après création d'un ensemble de 3 sémaphores. Afin de démontrer que l'ordre des opérations et celui des sémaphores n'est pas lié, les 2 dernières opérations ont été inversées.

int identSem;

struct sembuf operations[3];

/\* Ecriture/Lecture pour moi et les gens de mon groupe \*/

/\* Rien pour les autres ! \*/

identSem=semget(666,3,IPC\_CREAT | IPC\_EXCL | 0660);

/\* P(2) sur premier semaphore \*/

operations[0].sem\_num=0;

operations[0].sem\_op=-2;

operations[0].sem\_flg=0;

/\* V(1) sur troisieme semaphore \*/

operations[1].sem\_num=2;

operations[1].sem\_op=1;

operations[1].sem\_flg=0;

/\* Z sur second semaphore \*/

operations[2].sem\_num=1;

operations[2].sem\_op=0;

operations[2].sem\_flg=0;

/\* On effectue les 3 opérations \*/

semop(identSem,operations,3);

/\* destruction des semaphores \*/

semctl(identSem,56,IPC\_RMID)

**Exemple : deux programmes utilisant la mémoire partagée et les sémaphores**

Le programme ecrivain crée un segment de mémoire partagée et un ensemble de deux sémaphores. Il va écrire un message (une simple chaîne de caractères) à l'adresse du processus lecteur dans le segment de mémoire partagée.

1. Le premier sémaphore est destiné à contrôler la disponibilité d'une ressource. Sa valeur initiale est 1 afin de spécifier qu'il existe une unité de ressource : le segment de mémoire partagée.
2. Afin d'être sûr que le processus lecteur n'essaye pas de lire le message avant que celui-ci ne soit écrit, le processus ecrivain effectue une prise de ressource sur le sémaphore, soit **P(1)**.
3. Une fois cette opération terminée, ecrivain effectue une opération **V(1)** pour rendre la ressource. Réciproquement, le lecteur effectuera les mêmes opérations avant d'accéder au segment de mémoire partagée, garantissant ainsi un accès en exclusion mutuelle sur la ressource
4. Le deuxième sémaphore est quand à lui destiné à avertir le processus écrivain que le lecteur a terminé de lire son message. Pour ce faire, le processus ecrivain réalise une opération Z sur le sémaphore (de numéro 1) et le réveil s'effectue lorsque le processus lecteur réalise une opération P(1) sur ce sémaphore. Bien entendu, cela supposait que la valeur initiale du sémaphore était 1. Afin de démontrer l'utilisation de plusieurs opérations simultanées, nous effectuons les deux initialisations en une seule passe avec un tableau de 2 structures sembuf. Voici le code commenté des deux programmes ecrivain et lecteur

/\* Processus ecrivain \*/

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/sem.h>

#include <stdio.h>

#include <stdlib.h>

#define CLEF 666

#define LONGUEUR\_SEGMENT 512

int main(int argc,char \*argv[])

{

int semaphores;

int memoirePartagee;

struct sembuf manipSemaphores[2];

char \*attacheMoi;

if ((memoirePartagee=shmget(CLEF,LONGUEUR\_SEGMENT,IPC\_CREAT|0666)) == -1)

{

puts("Impossible de creer le segment de memoire partagee");

exit(1);

}

if ((semaphores=semget(CLEF,2,IPC\_CREAT|0666)) == -1)

{

puts("Impossible de creer les semaphores");

exit(1);

}

/\* Les deux operations d'initialisation des semaphores sont faites

en meme temps \*/

manipSemaphores[0].sem\_num=0;

manipSemaphores[0].sem\_op=1;

manipSemaphores[1].sem\_num=1;

manipSemaphores[1].sem\_op=1;

semop(semaphores,manipSemaphores,2);

/\* Operation P(1) sur le premier semaphore => prise de ressource sur la

memoire partagee \*/

manipSemaphores[0].sem\_num=0;

manipSemaphores[0].sem\_op=-1;

semop(semaphores,manipSemaphores,1);

puts("Je prends la ressource");fflush(stdout);

attacheMoi=shmat(memoirePartagee,NULL,0);

if ((int)(attacheMoi)==-1)

{

puts("Impossible d'attacher !");

exit(1);

}

else

{

printf("Adresse de l'attachement : %p \n",attacheMoi);

strcpy(attacheMoi,"Message a l'adresse de l'autre processus !");

shmdt(attacheMoi);

/\* Operation V(1) sur le premier semaphore => liberation de la ressource

memoire partagee \*/

manipSemaphores[0].sem\_num=0;

manipSemaphores[0].sem\_op=1;

semop(semaphores,manipSemaphores,1);

}

puts("Attente de l'autre processus");

fflush(stdout);

/\* Operation Z sur le second semaphore \*/

manipSemaphores[0].sem\_num=1;

manipSemaphores[0].sem\_op=0;

semop(semaphores,manipSemaphores,1);

puts("Debloque !");

fflush(stdout);

/\* Suppression des IPC \*/

semctl(semaphores,0,IPC\_RMID);

shmctl(memoirePartagee,IPC\_RMID,0);

return 0;

}

/\* Processus lecteur \*/

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/sem.h>

#include <stdio.h>

#include <stdlib.h>

#define CLEF 666

#define LONGUEUR\_SEGMENT 512

int main(int argc,char \*argv[])

{

int semaphores;

int memoirePartagee;

struct sembuf manipSemaphores;

char \*attacheMoi;

if ((memoirePartagee=shmget(CLEF,LONGUEUR\_SEGMENT,0600)) == -1)

{

puts("Impossible d'acceder au segment de memoire partagee");

exit(1);

}

if ((semaphores=semget(CLEF,2,0600)) == -1)

{

puts("Impossible d'acceder aux semaphores");

exit(1);

}

/\* Operation P(1) => on demande à accéder à la ressource \*/

manipSemaphores.sem\_num=0;

manipSemaphores.sem\_op=-1;

semop(semaphores,&manipSemaphores,1);

puts("Je demande a acceder a la ressource");fflush(stdout);

attacheMoi=shmat(memoirePartagee,NULL,0);

if ((int)(attacheMoi)==-1)

{

puts("Impossible d'attacher !");

exit(1);

}

else

{

printf("Adresse de l'attachement : %p \n",attacheMoi);

puts("Contenu de la memoire a l'attachement ");

puts(attacheMoi);

shmdt(attacheMoi);

manipSemaphores.sem\_num=0;

manipSemaphores.sem\_op=1;

semop(semaphores,&manipSemaphores,1);

}

puts("Deblocage de l'autre processus");

fflush(stdout);

/\* Debloquer le semaphore en attente se fait

a l'aide d'une operation prise de ressource P(1) car la valeur

courante de celui-ci est 1\*/

manipSemaphores.sem\_num=1;

manipSemaphores.sem\_op=-1;

semop(semaphores,&manipSemaphores,1);

return 0;

}