FAR
IPC : Communications inter-processus

Hinde Bouziane (bouziane@lirmm.fr)

Besoins

- On veut des moyens de communications entre processus s'exécutant sur la même machine (et même OS)
  - offrant d'autres possibilités que les tubes (et les sockets ?)
  - permettant l'échange de messages, le partage d'espace mémoire, la synchronisation.
- Les connus : communications dites **IPC - SV** :
  - **Files de messages** : envoi/réception de messages entre plusieurs processus ; l'unité transmise entre processus est un message.
  - **Ensembles de sémaphores** : outils et opérations évolués pour résoudre les conflits d'accès et la synchronisation.
  - **Mémoires partagées** : mémoire commune accessible à plusieurs processus, donc hors de l'espace d'adressage de chacun !
    - Ne pas confondre avec le partage d'un espace mémoire d'un seul processus par plusieurs threads.

Visualisation - exemple

La commande `ipcs` permet d'afficher (mais pas que) les objets IPC existants et leur état :

```
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 131072     jms        600        393216     2
0x00000000 163841     jms        600        393216     2          dest

------ Semaphore Arrays --------
key        semid      owner      perms      nsems      status
0xcbc384f8 0          jms        600        1
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x7a094087 32768      jms        666        160          20
```

Dans cet affichage, une file de messages dont l'identifiant est `32768` existe, avec le propriétaire et droits indiqués et elle contient actuellement 20 messages de longueur totale 160 octets.

Identification

L'identification d'un objet IPC se fait par mécanisme de **clef** qui permet d'obtenir un identifiant (descripteur de fichier) ou bien en obtenant directement l'identifiant.

Une clef est une suite binaire permettant **indirectement** d'obtenir l'identifiant d'un objet IPC.
La clef est calculée à partir de paramètres convenus à l'avance entre le créateur d'un objet IPC et l'ensemble des utilisateurs / processus.

La fonction dédiée :

```
key_t uneClef=ftok(const char * chemin, int entier)
```

Où `chemin` est le chemin d'un fichier existant, et `entier` est un entier quelconque (un caractère pour certains systèmes).

Identification - exemple

Tous les processus utilisateurs font :

```
key_t sesame = ftok("./readme.txt", 10)
```

ensuite ils peuvent obtenir l'identifiant de l'objet IPC via une fonction ayant la forme :

```
int id_obj = ? (sesame,....)
```

Files de messages - concepts

Une **file de messages** est une structure en mémoire permettant la communication entre processus, l'unité d'échange étant un **message**.

Un **message** est une structure de données quelconque portant une étiquette.

Actions possibles :

- créer une file et lui affecter des droits d'accès,
- utiliser une file existante (si possible),
- déposer un message,
- extraire un message de plusieurs façons : le premier disponible ou le premier portant une étiquette spécifique ; par exemple le premier message portant une étiquette rouge,
- consulter et gérer plusieurs paramètres de la file.

Remarques

- Un message extrait disparaît de la file.
- Il n'y a pas de notion d'ouverture/fermeture comme pour les tubes.
- La durée de vie d'une file va de sa création jusqu'à sa destruction : elle ne dépendent pas de la vie des processus accédant.
- La gestion des accès (lectures / écritures) concurrents à une file est prise en charge par le système.

Création et identification d'une file

Soit on crée une file et on obtient son identifiant, soit on obtient l'identifiant d'une file existante.

Syntaxe :

```
int        msgget   (key_t uneClef,        int droits)
 ⇑                    ⇑                     ⇑
identifiant          clef attachée         droits attachés à la file
                     à la file             ou accès demandé
```

Les droits s'énoncent comme pour la création de fichiers.

Exemples

```
int f_id = msgget(cle, IPC_CREAT|0666)
```

renvoie l'identifiant d'une file existante, sinon crée une nouvelle file avec les droits de lecture et d'écriture à tous les processus.

```
int f_id = msgget(cle, O_RDONLY)
```

est une demande d'accès en lecture seule à une file existante.

Structure d'un message

La structure du message est décrite ainsi dans le manuel :

```
struct msgbuf {
              long mtype;     /* message type, must be>0 */
              char mtext[1]; /* message data */
             };
```

Intreprétation : Une structure contenant l'étiquette comme première variable, suivie de variables dont la taille globale est > 0. Ces variables ne doivent pas contenir des pointeurs.

Exemple de message

```
struct strMonMsg {
      long monetiquette ;
      int num[10] ;
      char nom[30] ;
      } ;
...
struct strMonMsg monMsg;
```

Le contenu qui suit l'étiquette peut aussi être une `struct`.

Accès - extraction

Ce qu'on veut faire :

`extraire(idFile,tamponRécupération, uneEtiquette)`

Concrètement, l'appel système est **`msgrcv()`**, de syntaxe :

```
ssize_t    msgrcv(
           int identifiant,              ⇐ résultat de msgget()
           struct msgbuf *ptrmsg,        ⇐ pointeur tampon réception
           size_t lgmsg,                 ⇐ longueur max acceptée
           long étiquette,               ⇐ quelle étiquette
           int flags)                    ⇐ 0 pour l'instant
```

- Le résultat est le nombre d'octets lus hors étiquette.
- étiquette > 0 : lecture du premier message disponible avec l'étiquette e = étiquette.
- étiquette $= 0$: lecture du premier message disponible.
- étiquette < 0 : lecture premier message disponible avec la plus petite étiquette $e \leq |\text{étiquette}|$.

Exemple

```
struct sMsg {long etiq; char mot[12];} vmsg;

int ret = msgrcv(f_id, &vMsg,
     (size_t)sizeof(vMsg.mot), (long) monPid, 0);
```

demande à extraire de la file dont l'identifiant est `f_id`, le premier message portant l'étiquette de valeur `monPid`, et de copier ce message dans `vMsg`.

Rappel : le message va disparaître de la file.

Accès - dépôt

Ce qu'on veut faire :

`déposer(idFile, message, uneEtiquette (utile?))`

Concrètement, l'appel système est **`msgsnd()`**, de syntaxe :

```
int    msgsnd(
       int identifiant,          ⇐ résultat de msgget()
       struct msgbuf *ptrmsg,    ⇐ pointeur sur tampon à déposer
       size_t lgmsg,             ⇐ longueur du message
       int flags)                ⇐ 0 pour l'instant
```

- L'étiquette est absente de cet appel, car elle fait partie de la structure `msgbuf` et le message est déposé avec cette étiquette.
- Le résultat indique si l'opération a réussi ou échoué.
- Attention, une valeur négative ou nulle pour l'étiquette, est forcément une erreur ! Penser au lecteur pour s'en convaincre.

Suppression d'une file

La suppression d'une file peut se faire par la commande `ipcrm`, ou par l'appel système :

```
int msgctl(int f_id, int op,
           .../*struct msqid_ds *entreeTable*/).
```

L'appel système est en fait très général et permet de gérer tous les paramètres de la file. On se contente ici de donner la forme permettant la suppression seule :

```
int res =   msgctl(
            identifiant,    ⇐ résultat de msgget()
            IPC_RMID,       ⇐ constante pour la destruction
            NULL)           ⇐ pointeur si gestion de paramètres
```

Rappels

Un sémaphore est un mécanisme de synchronisation de processus.
Il s'agit d'une structure de données qui comprend :

- un entier non négatif donnant le nombre de ressources disponibles
- une file d'attente de processus

Et manipulée au travers de trois opérations :

- Init (sémaphore sem, int nombre_de_ressources)
- P(sémaphore sem, int nb_ressources) : bloque l'appelant si le nombre de ressources de sem demandées est supérieur au nombre de ressources disponibles, sinon décrémente le nombre de ressources de sem.
- V(sémaphore sem, int nb_ressources) : libère un nombre de ressources obtenues et débloque un ou des processus en attente s'il en existe.

Sémaphores SV

L'implantation SV des sémaphores permet de gérer un ensemble de sémaphores, de sorte à pouvoir contrôler

$$\begin{array}{lll} k_1 & \text{exemplaires de la ressource} & R_1 \\ \ldots & \ldots & \ldots \\ k_i & \text{exemplaires de la ressource} & R_i \end{array}$$

avec des opérations (pour chaque sémaphore de l'ensemble)

P_n qui bloque l'appelant si la valeur du sémaphore est inférieure à $|n|$

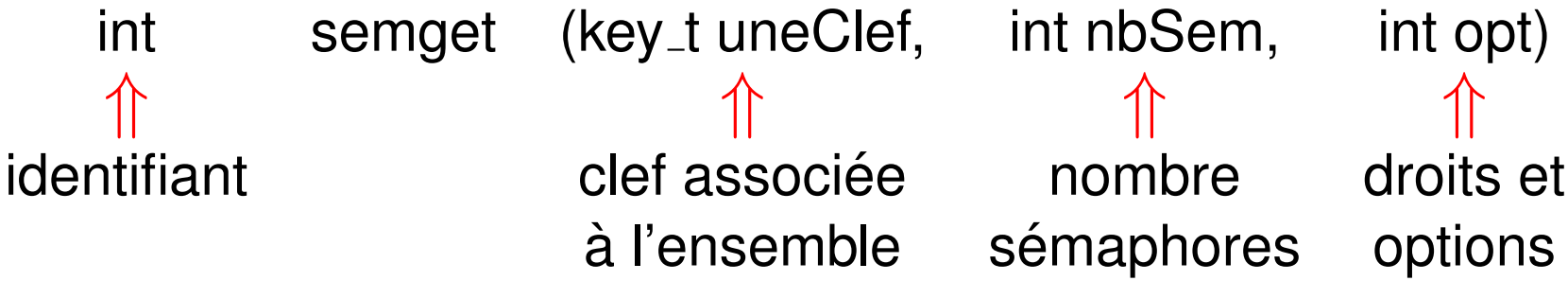
V_n qui incrémente la valeur du sémaphore de $|n|$ et débloque les attentes

Z qui attend que le sémaphore soit nul afin de réaliser des rendez-vous

et possibilité de réaliser plusieurs opérations P_n et V_n atomiquement.

Création

La création ressemble à celle des files de messages et mémoires partagées.



Permet de crée un **tableau** de *nbSem* sémaphores ou de récupérer l'identifiant d'un tableau existant. Attention, L'objet IPC ici est le tableau. Il s'agit d'un ≪ vrai ≫ tableau C.

Exemple :

```
int idSem = semget(cleSem, 1, IPC_CREAT|0666);
```

crée et/ou récupère l'identifiant d'un tableau à un seul sémaphore, associé à *cleSem*.

Opérations

On souhaite réaliser une combinaison d'opérations P, V et Z sur un (sous-)ensemble de sémaphores.

Concrètement, on utilise la fonction :

```
int semop(
                int idSem,              ⇐ résultat de semget()
                struct sembuf *tabOp,   ⇐ ensemble d'opérations
                                          à réaliser
                int nbOp)               ⇐ nombre d'opérations
                                          dans ce tableau
```

Le résultat est 0 (réussite) ou -1 (échec).

Où, toute opération (*P*, *V* ou *Z*) sur un sémaphore est décrite par une structure `sembuf` et est propre à ce sémaphore. L'**ensemble** des *nbOp* opérations sera réalisé de façon atomique.

Opérations - suite

```
struct sembuf {
  unsigned short  sem_num;   /* Numéro du sémaphore */
           short  sem_op;    /* Opération sur le sémaphore */
           short  sem_flg;   /* Options par exemple SEM_UNDO */
};
```

- Les numéros commencent à 0.
- La valeur n de sem_op détermine l'opération
  - si $n < 0$ l'opération est P avec comme valeur $|n|$: tentative de décrémenter le sémaphore numéro sem_num de $|n|$;
  - si $n > 0$ l'opération est V : incrémentation de n avec réveil des processus en attente ;
  - si $n = 0$ l'opération est Z : attente que la valeur du sémaphore soit 0 (voir rendez-vous).

Exemples

Pour un sémaphore unique (à la Dijkstra), on peut définir :

Une opération P :

```
struct sembuf opp;
opp.sem_num=0;
opp.sem_op=-1;
opp.sem_flg=0;
semop(idSem,&opp,1);
```

Une opération V :

```
struct sembuf opv;
opv.sem_num=0;
opv.sem_op=+1;
opv.sem_flg=0;
semop(idSem,&opv,1);
```

Ou encore :

```
struct sembuf op[]={
   {(u_short)0,(short)-1,0},
   {(u_short)0,(short)+1,0}  };
```

puis : `semop(idSem,op,1)` pour P,
et `semop(idSem,op+1,1)` pour V.

Initialisation

C'est la primitive système *semctl()* qui est utilisée pour l'initialisation d'un ensemble de sémaphores, toujours atomiquement.

Remarque : c'est cette même primitive qui sera utilisée pour détruire un ensemble de sémaphores ou obtenir des informations sur ces derniers.

Le prototype est défini ainsi :

```
int semctl(int semid, int semnum, int cmd, ...);
```

Interprétation : on veut faire telle commande sur le sémaphore numéro semnum, de l'ensemble semid. Pour les pointillés, le manuel dit ceci :

La fonction a trois ou quatre arguments, selon la valeur de cmd. Quand il y en a quatre, le quatrième est de type union semun. Le programme appelant doit définir cette union de la façon suivante :

Initialisation - suite du calvaire

```
union   semun {
        int val ;                       /* cmd = SETVAL */
        struct semid_ds *buf ;          /* cmd = IPC_STAT ou IPC_SET */
        unsigned short *array ;         /* cmd = GETALL ou SETALL */
        struct seminfo *__buf ;         /* cmd = IPC_INFO (sous Linux) */
    } ;
```

Conséquences :

- Il faut réapprendre ce qu'est une structure de type *union*
- Déduire qu'on peut certainement faire beaucoup d'opérations intéressantes.

Exemple - initialisation simple d'un sémaphore

En supposant qu'on a déclaré dans le programme une `union semun` comme celle décrite, on peut initialiser un sémaphore à 1 comme suit :

```
semun egCtrl;
egCtrl.val=1;
if(semctl(idSem, 0, SETVAL, egCtrl) == -1){
    perror(''problème init'');
    //suite
}
```

Revoir la structure `semun` : on peut initialiser de façon atomique un tableau de sémaphores (*semnum* devient le nombre d'éléments), obtenir des valeurs courantes ou encore gérer des caractéristiques relatives à l'ensemble de sémaphores.

Destruction

Enfin, la **destruction** d'un ensemble se fera classiquement avec l'appel :

```
semctl(idSem, 0, IPC_RMID)
```

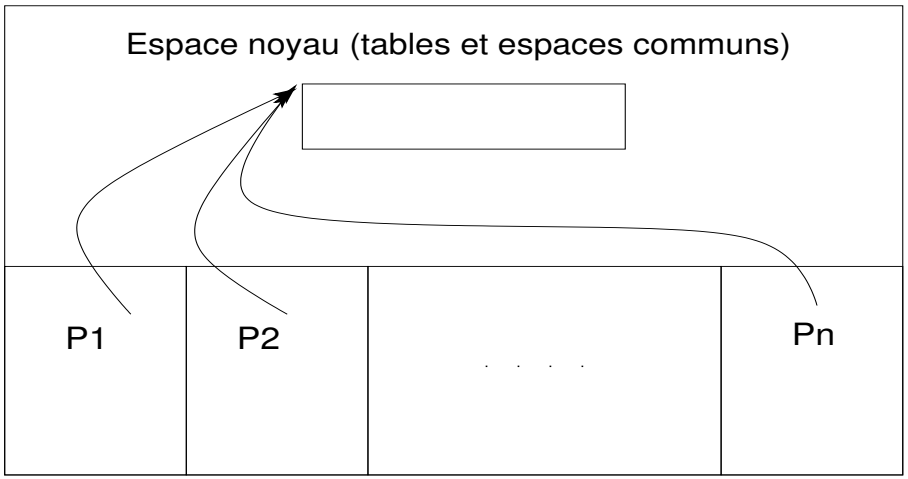
Elle réveillera tous les processus en attente, s'il en existe.

Principe

- La communication avec les tubes ou files de messages consiste à transférer (copier) des données, dans un espace géré et synchronisé par le système.
- Une **mémoire partagée** consiste à disposer d'un espace de mémoire, accessible à plusieurs processus.
- Chaque processus pourra y ≪ travailler ≫ comme sur toute donnée propre.
- Avec des restrictions possibles : certains processus pourront lire et écrire, d'autres ne pourront que lire ou n'auront aucun droit d'accès.

Caractéristiques

- Localisation dans l'espace alloué « au système ».



- L'espace alloué (on parlera de **segment**) sera persistant : son existence sera indépendante des processus qui y accèdent.

Utilisation

- Un espace, ou segment, de mémoire partagée sera créé par un processus.
- Chaque processus voulant y accéder demandera à s'**attacher** l'espace ; après vérification des droits, il disposera d'un pointeur vers cet espace.
- Les processus accédant devront gérer la synchronisation : exclusion et protection. Classiquement, ils utiliseront des *sémaphores*.
- La destruction de l'espace devra être faite par un processus ayant le droit de destruction. En cas d'arrêt du système, l'espace sera perdu : fonctionnement identique à celui des files de message.

Opérations sur un segment de mémoire partagée

Les opérations réalisables sur un espace mémoire partagée :

1. création d'un segment ;
2. demande d'attachement (obtention d'un pointeur) ;
3. détachement (abandon d'accès) ;
4. contrôle des paramètres dont suppression (comme pour les files de messages).

Remarque : l'accès en lecture/écriture se fait de manière classique, en utilisant un pointeur. Il n'y a donc pas de primitives dédiées.

Création et identification d'un segment

Comme pour les files de messages, le même appel système, *shmget()*, permet de créer un segment ou uniquement d'obtenir son identifiant.

Syntaxe :

```
int shmget (key_t uneClef, size_t taille, int optEtDroits)
```

- `int` ⇑ identifiant
- `uneClef` ⇑ clef associée au segment
- `taille` ⇑ taille demandée
- `optEtDroits` ⇑ droits et/ou type accès

- Le principe d'obtention de la clef est celui déjà vu avec `ftok()`.
- Les droits s'énoncent comme pour la création de fichiers.
- Lorsque le segment existe, on demande une taille inférieure ou égale à la taille du segment (0 est une bonne solution)

Exemple

```
struct uneChaine{  char c;
                   int x, y;
                   struct uneChaine *suiv;
                   };

int sh_id=shmget(  sesame,
                   size_t(30*sizeof(uneChaine)),
                   IPC_CREAT|0666);
```

permet de créer un segment avec les droits d'accès de lecture et écriture à tout processus.

Le segment contient une liste chaînée.

```
int sh_id=shmget(sesame,size_t(0),O_RDONLY);
```

est une demande d'accès en lecture seule, en supposant que le segment existe.

Demande d'accès : attachement

Pour accéder à un espace de mémoire partagé, un processus demande l'*attachement* de cet espace ; il consiste à obtenir un pointeur dans son espace propre, vers cet espace extérieur.

syntaxe :

```
void *      shmat    (int idMem,    const void * adrForce,    int options)
```



adresse ou (void $*$) -1 — identifiant obtenu — NULL sauf exception — type accès

- Le type d'accès par défaut est en lecture et écriture. `options` permet de le modifier, par exemple de demander l'accès en lecture seule avec `SHM_RDONLY`.

Abandon - détachement

On abandonne l'accès en détachant l'espace commun. **syntaxe** :

```
int        shmdt   (const void * adrAtt)
```

0 : réussite
−1 : échec

adresse d'attachement

- À la fin du processus, tous les segments préalablement attachés, dans ce processus, sont détachés.
- **Question** : Pourquoi faut-il donner une adresse d'attachement pour détacher et non l'identifiant ?

Exemples

Supposons un segment contenant un tableau d'entiers :

```
int * tab;
if((tab = (int *)shmat(idMem, NULL, 0))==(int *)-1){
     perror("shmat");
     //suite ...}
```

Ou la liste chaînée vue précédemment :

```
struct uneChaine * p_att;
p_att = (struct uneChaine *)shmat(idMem, NULL, 0);
if ((void *)p_att == (void *)-1){
     perror("shmat");
     //suite ...}
```

Détachement :

```
int dtres = shmdt((void *)p_att);
```

Suppression

La suppression est similaire à celle des files de messages :

```
int    shmctl(
       int identifiant,    ⇐ résultat de shmget()
       IPC_RMID,           ⇐ constante pour la destruction
       NULL)               ⇐ pointeur si gestion de paramètres
```

Mais encore :

On pourra regarder dans le manuel comment récupérer les caractéristiques courantes ou modifier celles qu'on peut modifier.

Bon à retenir

Faire la différence :

- Entre processus et threads.
- Partage de ressources inter-processus et inter-threads.
- Synchronisation entre processus et threads (exclusion mutuelle, attente d'un événement).

Conseils de programmation :

- bien initialiser les objets/variables utilisés,
- terminaison "propre" : libération de l'espace mémoire alloué, nettoyage des tables IPC, terminaison des processus etc,
- traitement des retours de fonction et gestion des erreurs,
- faire attention aux problèmes liés à la synchronisation, en particulier les situations d'interblocage. Exemple : ne jamais effectuer un blocage dans une section critique sans libérer la section critique.
- etc.