Communications inter-processus (IPC)

Hinde Bouziane (bouziane@lirmm.fr)

Besoins

- On veut des moyens de communication entre processus (lourds) s'exécutant sur une même machine (et même OS), qui n'ont pas forcément un lien de parenté (fork) ni le même utilisateur propriétaire.
  - permettant l'échange de messages, le partage d'espace mémoire, la synchronisation.
  - il existe plusieurs possibilités, tubes, IPC, sockets (?) ...
- Communications dites **IPC - SV** :
  - **Files de messages** : envoi/réception de messages entre plusieurs processus ;
  - **Ensembles de sémaphores** : outils et opérations évolués pour résoudre des conflits d'accès et des problèmes d'ordre d'exécution de sous-ensembles d'instructions (synchronisation)
  - **Mémoires partagées** : mémoire commune accessible à plusieurs processus (en dehors de l'espace d'adressage de chacun)

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.

Comment plusieurs processus accèdent à un même objet IPC

- Par mécanisme de **clef** qui permet d'obtenir un identifiant (un entier) ou bien en obtenant directement l'identifiant.
- Seule l'identification par clef est vue dans cette UE.
- Une clef, pour un utilisateur, est une paire (chemin d'un fichier existant, entier).
- Une clef, pour le système, est une suite binaire permettant d'obtenir l'identifiant d'un objet IPC.
- La même clef utilisée par différents processus et pour un type d'objet IPC, permet l'accès à un seul et unique objet IPC, donc de l'identifier.
- Donc, il faut obtenir la clef numérique à partir de la paire (chemin d'un fichier, entier) pour pouvoir ensuite obtenir l'identifiant.

En pratique

Pour obtenir la clef numérique :

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

Où `chemin` est le chemin d'un fichier existant, et `entier` est un entier quelconque.
Si tous les processus utilisateurs exécutent :

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

ils peuvent obtenir l'identifiant d'un objet IPC via une fonction ayant la forme :

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

Files de messages - introduction

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,
- détruire une file, consulter ou gérer ses paramètres.

Remarques

- A la création d'une file de messages, des droits de lecture/écriture sont donnés aux utilisateurs (même principe que pour les fichiers).
- La gestion des accès (lectures / écritures) concurrents à une file est prise en charge par le système (exclusion mutuelle).
- Un message extrait disparaît de la file.
- La durée de vie d'une file va de sa création jusqu'à sa destruction : elle ne dépend pas de la vie des processus accédant.

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 + options
```

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

Un message est une structure contenant une étiquette suivie du contenu du message (la donnée à envoyer ou à recevoir)

Dans le manuel, la structure du message est décrite ainsi :

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

Interprétation : 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 ayant l'identifiant `f_id`, le premier message portant l'étiquette de valeur `monPid`, et de copier ce message dans `vMsg`.

Remarque : le message disparaît de la file après son extraction.

Accès - dépôt

Ce qu'on veut faire :

`déposer(idFile, message avec son Etiquette)`

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

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

- L'étiquette n'est par un paramètre : elle fait partie de la structure `msgbuf` du message 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 ! Pourquoi ?

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

L'idée du sémaphore (Dijkstra - années 60) est d'utiliser un compteur de déblocage, un mécanisme de synchronisation de processus concurrents.
Il s'agit d'une structure de données qui comprend :

- un entier S non négatif qui désigne par exemple le nombre d'autorisations d'accès à une section critique ou un nombre de ressources partagées disponibles
- une file d'attente de processus

Et manipulée uniquement au travers de trois opérations :

- Init (sémaphore sem, int nb)
- P(sémaphore sem, int nb) : bloque l'appelant si nb $\leq$ la valeur de sem, sinon décrémente cette valeur de nb.
- V(sémaphore sem, int nb) : incrémente la valeur de sem de nb et peut provoquer le déblocage de processus en attente.

Sémaphores SV

Un sémaphore à la Dijkstra ne suffit pas pour résoudre efficacement (voir correctement) des problèmes tel que la demande d'un sous ensemble de ressources de différents types :

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

Efficacement, voir correctement, signifient : sans interblocage, sans famine, minimum/sans attente inutile...

Sémaphores SV

Le Système V propose la notion de tableau de sémaphores, permettant d'exécuter une combinaison d'opérations sur un sous ensemble des sémaphores du tableau. Cette combinaison s'exécute de manière atomique (sans interruption).

Les opérations possibles sur chaque sémaphore :

$P_n(S_i)$	qui bloque l'appelant si la valeur du sémaphore à l'indice i est inférieure à $\|n\|$
$V_n(S_i)$	qui incrémente la valeur du sémaphore à l'indice i de $\|n\|$ et débloque les attentes
$Z(S_i)$	qui attend que le sémaphore à l'indice i soit nul (exemple : pour réaliser des rendez-vous)

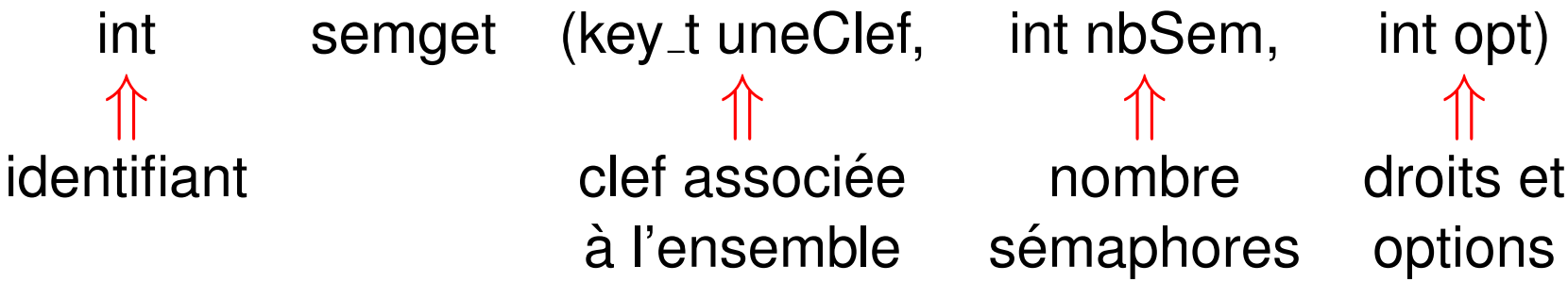
et possibilité de réaliser plusieurs opérations P_n et V_n atomiquement.

Sémaphores SV - actions sur les tableaux

- Créer un tableau / ensemble de sémaphores et lui affecter des droits d'accès,
- Utiliser un tableau existant,
- Initialiser les valeurs des sémaphores d'un tableau,
- Exécuter une combinaison d'opérations *P* et *V* sur les sémaphores,
- Détruire un tableau de sémaphore, consulter ou gérer ses paramètres.

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*.

Initialisation

La primitive système *semctl()* est utilisée pour l'initialisation d'un ensemble de sémaphores et pour d'autres actions. Elle est atomique.

L'initialisation doit absolument précéder toute utilisation (opérations) des sémaphores du tableau par tout processus concurrent.

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

```
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) */
    };
```

On en déduit qu'on peut faire beaucoup d'opérations intéressantes.

Exemple - initialisation 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.

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, avec de l'arithmétique des pointeurs :

```
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.

Destruction

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

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

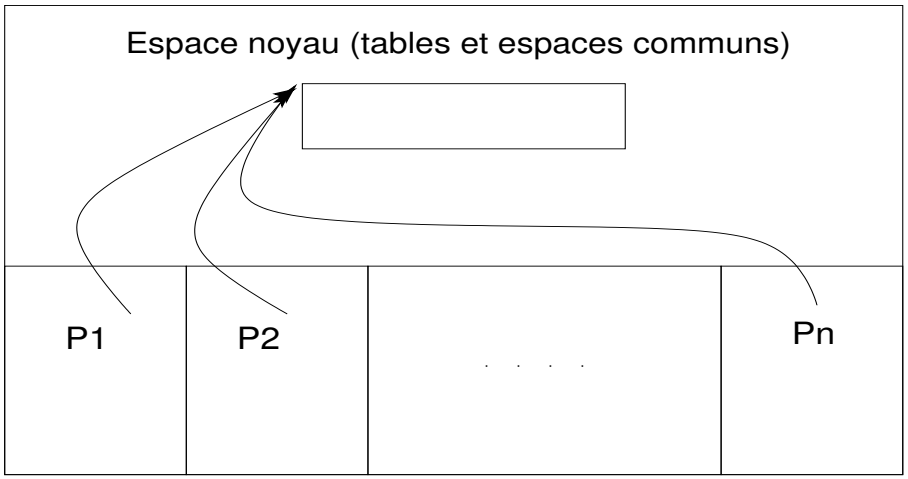
Elle réveillera tous les processus en attente, s'il en existe.

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.

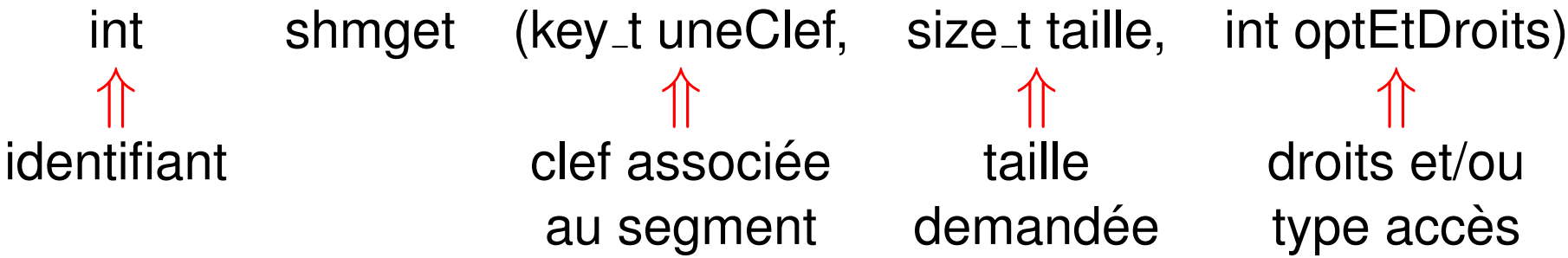
Actions sur un segment de mémoire partagée

- création d'un segment ;
- demande d'attachement (obtention d'un pointeur) ;
- détachement (abandon d'accès) ;
- 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

L'appel système *shmget()*, permet de créer un segment ou uniquement d'obtenir son identifiant.



- 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.

```
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 :

```
int          shmdt    (const void * adrAtt)
 ⇑                      ⇑
0 : réussite            adresse
−1 : échec              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.

Conseils pour la programmation :

- Veillez à initialiser les objets/variables avant de les utiliser.
- S'assurer qu'un tableau de sémaphores est initialisé avant de l'utiliser (un seul processus sera en charge de l'initialisation (logiquement le créateur de l'objet IPC).
- Il est possible d'identifier le processus responsable de la création d'un objet IPC via l'option IPC_EXCL.
- Terminaison "propre" : libération de l'espace mémoire alloué, nettoyage des tables IPC, terminaison des processus etc.
- Traitement des retours d'une 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.