La communication interprocessus : IPC Unix System V

7.1 Introduction

- Les processus concurrents sont créés pour participer à la réalisation d'une application commune.
- Ces processus se partagent des informations communes qu'ils peuvent consulter ou modifier au cours de leur exécution.
- Ces processus peuvent être crées dans un même programme(ils ont le même père) ou créés dans des programmes différents.
- La communication interprocessus(IPC : InterProcess Communication) permet à un ensemble de processus concurrents , s'exécutant sur une même machine, de communiquer entre eux en respectant certaines règles.

Types de communications IPC system V

Il existe trois types de communication interprocessus :

1. Les files de messages,
2. Les segments de mémoire partagée,
3. Les sémaphores.

- Ces mécanismes ont été introduits dans Unix system V en 1983.

- *Dans ce cours, nous nous limitons à la communication par segments de mémoire partagée et les sémaphores.*

7.2 Identification d'un IPC

- Une IPC (segment de mémoire ou sémaphore) est identifiée à l'aide d'une **clé unique connue par tous les processus coopérants** utilisant cette ressource partagée.

- Une IPC peut être **privée** ou **publique**:
  - ✓ **IPC privée :** Identifié par une clé spéciale : **IPC_PRIVATE**.

    Une IPC privée ne peut être utilisée que par le processus créateur et ses descendants ou processus créés dans le **même programme**. ➔

    Tous les processus fils peuvent accéder à cette IPC par héritage.

    Cet héritage est perdu lors d'un recouvrement de programme(chargement d'un nouveau programme à l'aide d'une instruction exec(execl, execlv, execve, …). ➔

    Tous les accès aux segments de mémoire partagée et aux sémaphores sont perdus.

- ✓ **IPC publique :** Utilisée par des processus créés dans des programmes différents(processus qui non pas de liens de parenté) ou par des processus créés dans le même programme.
  La clé est obtenue à partir d'**un chemin d'accès d'un fichier** ou **d'un répertoire existant** et d'**un numéro de projet**, grâce à la fonction suivante:

  `key_t ftok (char *chemin d'accès, char projet);`

  Remarque *: Les systèmes actuels acceptent un entier pour « projet »; L'octet de poids faible de cette valeur doit être* ***différent de zéro(0)****.*

Clé sur 32 bits

31 .. 24	23 .. 16	15 .. 0
Projet & 0xFF	N° Mineur périphérique & 0xFF	I-node & 0xFFFF

- Unix identifie chaque périphérique au moyen de 2 numéros , N° majeur : type de périphérique et N° mineur : Index dans le type.
- Un fichier ou répertoire est identifié à l'aide d'un nom et d'un d'i-node (index node ou indexe nœud)

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok (char *chemin, char projet);
```

Cette fonction renvoie une clé unique de 32 bits en combinant le numéro d'i-node du chemin(fichier ou répertoire) précisé avec le numéro de projet.

Exemple:

```
key_t  cle;
cle=ftok ("/home/." ,  'A');
If(cle==-1) {
            printf("\n erreur : la clé n'a pas été créée");
           exit(1);
}
```

Si Cle = -1 ➔ erreur: la clé n'a pas été créée.

7.3 Les segments de mémoire partagée

- Le principe de la mémoire partagée consiste à définir une zone(segment) en mémoire qui soit en dehors de la zone attribuée à chaque processus. Un segment de mémoire partagée est indépendant de tout processus.
- La mémoire partagée est la forme de communication interprocessus la plus rapide car tous les processus se partagent la même mémoire.
- L'accès à cette mémoire partagée ne nécessite pas d'appel système.
- Ce mécanisme n'implique pas de copie ou duplication des données : Tous les processus qui se partagent un segment de mémoire partagée peuvent lire et modifier les données se trouvant dans ce segment.
- La synchronisation des accès à la mémoire partagée est à la charge de l'utilisateur.
- Cette synchronisation est réalisée à l'aide des sémaphores.

- **Un segment de mémoire partagée doit être créé et détruit explicitement par un des processus concurrents.**
- **Remarque** : *Après la terminaison des tous les processus utilisant un segment de mémoire partagée, le système ne détruit pas ce segment.*

7.3.1 Utilisation d'un segment de mémoire partagée

Un processus désirant utiliser un segment partagé doit faire les actions suivantes :

1) Obtenir une clé(publique ou privée) qui permet d'identifier le segment,
2) Créer le segment de mémoire partagée,
3) Attacher le segment de mémoire partagée à son espace d'adressage,
4) Utiliser le segment de mémoire partagée,
5) Détacher le segment de mémoire partagée,
6) Détruire le segment(si dernier processus utilisant le segment et si c'est le processus créateur).

7.3.2 Fonctions de manipulation des segments de mémoire partagée

- **shmget()** : Permet à partir d'une clé d'obtenir l'identifiant d'un segment de mémoire partagée existant ou d'en créer un au besoin.
- **shmat()** : Permet d'attacher le segment de mémoire partagée dans l'espace d'adressage du processus.
- **shmdt()** : Permet de détacher le segment si on ne l'utilise plus.
- **shmctl()** : Permet de supprimer ou paramétrer un segment de mémoire partagée.

Les constantes et les prototypes de ces fonctions sont définis dans <sys/shm.h> ➔

```
#include < sys/shm.h >
```

1) Création d'un segment de mémoire partagée

int shmget(key_t cle, int taille, int options);

- ✓ **cle** : clé obtenue avec ftok() ou IPC_PRIVATE.
- ✓ **taille** : indique la taille en octets du segment de mémoire partagée.
  Si le segment existe déjà, la taille doit être inférieure ou égale à la taille effective du segment.
  La taille est arrondie au multiple supérieur de la taille des pages mémoire du système (4Ko sur un processeur Intel).
- ✓ **options** : OU (|) binaire entre IPC_CREAT, IPC_EXCL et les 9 bits d'autorisation.

  adr= shmget (cle, 4096, IPC_CREAT | **0**666);
  0 : pour indiquer que '666' est une valeur octale.

  - IPC_CREAT : créer un nouveau segment de mémoire partagée s'il y a pas de segment associé à la clé <cle>. S'il existe déjà un segment associé à la clé indiquée, La fonction renvoie l'identificateur de ce segment.

- Bits IPC_EXCL : toujours utilisé avec IPC_CREAT, permet de créer un nouveau segment. S'il existe déjà un segment associé à la clé indiquée, abandonner la fonction *shmget* (erreur : pas de création de segment).
- Bits d'autorisation : Propriètaire(rw-), membres du goupe(rw-), autres(rw-).

Remarques :

a) Le bit x(exécution) n'est pas utilisé(-): il n' y a pas d'exécution d'un segment de mémoire partagée(données).

b) Ne pas oublier de préciser les autorisations(permissions) d'accès lors de la création d'un segment de mémoire partagée. Sinon le segment aura la protection 000 (en octal). ➔ Aucun accès.

- La fonction retourne un identificateur du segment de mémoire partagée.
- Ce segment n'est pas encore utilisable, Le processus doit l'attacher à son espace d'adressage à l'aide la fonction shmat().

2) Attachement d'un segment de mémoire partagée à l'espace d'adressage d'un processus

```
char *shmat(int shmid, char *adr, int options);
```

- ✓ **shmid** : identificateur de segment de mémoire partagée créé avec shmget.
- ✓ **adr** : adresse désirée de l'attachement, elle est rarement utilisée. On met la valeur **NULL** ou **0** et c'est le système qui choisit un emplacement libre(une adresse) dans l'espace d'adressage du processus. Solution à utiliser dans les TP.
- ✓ **options** : SHM_RDONLY : l'attachement est réalisé en lecture seule,
  0 : l'attachement est réalisé en lecture et écriture.

- La fonction retourne l'adresse du premier octet du segment ou -1 si erreur.

Remarque : Pour les exercices et les TP, l'attachement **est obligatoire.** On ne tient pas compte de l'héritage de fork().

3) Détachement d'un segment de mémoire partagée

int shmdt(char *adrseg);

- Détachement du segment dont la demande d'attachement par shmat a été réalisée à l'adresse adrseg (adrseg contient l'adresse retournée par shmat).

4) Suppression d'un segment de mémoire partagée

int shmctl(int shmid, int opérations, struct shmid_ds *p_shm);

- Cette fonction réalise différentes opérations de contrôle : Les valeurs de « opérations » sont IPC_RMID, IPC_STAT, IPC_SET, SHM_LOCK et SHM_UNLOCK.
- Nous traitons que la valeur « IPC_RMID » qui permet de supprimer un segment de mémoire partagée. *Pour plus de détail voir les man's des IPC.*
  IPC_RMID : Le segment est marqué « prêt pour la suppression ».
  Un segment n'est effectivement détruit que s'il a été détaché par tous les processus qui l'utilisent.
- **Remarque** : La suppression des IPC est à la charge de l'utilisateur.

7.3.3 Exemple : Création et d'utilisation d'un segment de mémoire partagée

```
int main ()
{
  int taille= 4096;  /* taille d'une page mémoire sous linux */
 int adrmemp;   char  *mem;      cle key_t;

  /* Créer une clé. On suppose que le fichier fexemple.c existe */
  cle=ftok("./fexemple.c" ,'A');

  /* Créer un segment de mémoire partagée. */
  adrmemp = shmget (cle, taille, IPC_CREAT | 0666);
  if(adrmemp==-1){ printf(" \n erreur "); exit(0);}

  /* Attacher le segment de mémoire partagée. Adr=0 ➔
      c'est le système qui choisit l'adresse pour l'attachement*/
  mem = (char *) shmat (adrmemp, 0, 0);
  if(mem == NULL) { printf("\nErreur "); exit(13);}
```

```
 /* Utiliser la mémoire partagée : Initialiser toute la mémoire avec 'a' */
  for(int i=0; i<taille; i++) mem[i]='a';

  /* Détacher le segment de mémoire partagée. */
  shmdt (mem);

  /* Libèrer le segment de mémoire partagée.  */
  shmctl (adrmemp, IPC_RMID, 0);

  return 0;
}
```

Remarque : Dans cet exemple, on a pas traité le problème des conflits d'accès(synchronisation) à la mémoire par plusieurs processus.

7.4 Les sémaphores Unix system V

- Le partage de segment de mémoire partagée et la nécessite une coordination(synchronisation) des actions des processus qui l'utilise.
- Cette coordination(synchronisation) est assurée au moyen des sémaphores Unix SystemV.

7.4.1 Utilisation des sémaphores Unix system V

- Obtenir une clé(publique ou privée) qui permet d'identifier un ensemble de sémaphores(**un** ou **n** sémaphores),
  `key_t ftok (char *chemin d'accès, char projet);`
- Créer un ensemble de un ou n sémaphores: `semget()`,
- Initialiser ce ou ces sémaphores: `semctl()`,
- Utiliser les sémaphores pour synchroniser les actions des processus concurrents(opérations P et V ou autre opération) `semop()`,
- Détruire le ou les sémaphores(si dernier processus utilisant ce(s) sémaphore(s) et si c'est le processus créateur) : `semctl()`.

7.4.2 Création d'un ensemble de sémaphores

La création est réalisée à l'aide la fonction **semget()**.

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

int semget (key_t cle, int nb_sem, int option);
```

- ✓ **cle** : Clé identifiant l'ensemble de sémaphores.
  Il y a une clé par ensemble de sémaphores.
- ✓ **nb_sem** : Nombre de sémaphores de l'ensemble (**1 à 64**).
- ✓ **options** : OU (|) binaire entre IPC_CREAT, IPC_EXCL et les 9 bits d'autorisation.
  Adrs = semget (cle, 1 , IPC_CREAT | 0666);
  0 : pour indiquer que '666' est une valeur octale.

- **IPC_CREAT** : Créer un nouvel ensemble de sémaphores s'il n'y a pas d'ensemble associé à la clé <**cle**>.
  S'il existe déjà un ensemble de sémaphores associé à la clé indiquée, La fonction renvoie l'identificateur de cet ensemble.
- **IPC_EXCL** : toujours utilisé avec IPC_CREAT, permet de créer un nouveau segment. S'il existe déjà un segment associé à la clé indiquée, abandonner la fonction semget (erreur : pas de création de sémaphores).
- **Bits d'autorisation** : Propriètaire(rw-), membres du goupe(rw-), autres(rw-).

✓ La fonction retourne un identificateur de l'ensemble des sémaphores.

7.4.3 Initialisation d'un ensemble de sémaphores

- Il est possible de réaliser certaines opérations de contrôle sur un ensemble de sémaphores grâce à la primitive semctl() :

```
int semctl (int idf_sem, int num_sem, int commande, union semun arg);
```

✓ **idf_sem** : identifiant de l'ensemble de sémaphores. Cet identifiant est retourné par la primitive *semget*().

✓ **num_sem** : Numéro du sémaphore sur lequel il faut réaliser une opération.
Dans le cas où l'opération concerne l'ensemble de tous les sémaphores et non un sémaphore particulier, ce paramètre peut-être d'une valeur quelconque car celle-ci ne sera pas utilisée; mais il faut y mettre une valeur pour conserver l'ensemble des paramètres.

- **commande** : commande permettant de contrôler l'ensemble de sémaphores. Celle-ci doit être une des dix constantes suivantes :
  - **SETVAL** : affectation d'une valeur précise dans le sémaphore référencé par num_sem (ou dont le numéro se trouve dans num_sem)
  - **SETALL** : affectation d'une valeur précise dans tous les sémaphores de l'ensemble.
- **union semun {**

  int val;

  struct semid_ds *buf; /* voir le man */

  unsigned short *tab; /* voir le man */

  } arg;
  - **arg.val** : valeur du sémaphore (si commande = SETVAL ou SETALL).

Pour plus de détails voir le man de semctl().

7.4.4 Opérations sur un ensemble de sémaphores

La primitive semop() permet d'exécuter des opérations sur un groupe de de 1 à n sémaphores de l'ensemble. Ce groupe d'opérations sera atomique, c'est à dire qu'il sera exécuté sur tous les sémaphores du groupe ou sur aucun.

```
int semop (int sem_id, struct sembuf  *operation, unsigned int nb_op);
```

- ✓ **sem_id** : identifiant de l'ensemble de sémaphores.
- ✓ **Operation** : Pointeur sur une variable(ou un tableau) de type struct sembuf définissant l'opération à effectuer sur un sémaphore.

```
struct sembuf{
    short int  sem_num; /* Numéro du sémaphore dans l'ensemble */
    short int sem_op;    /* valeur à ajouter au sémaphore */
    short int sem_flg;   /* attribut pour l'opération */
}
```

- **opération** est définie par une valeur entière de sembuf.sem_op qui est interprétée comme suit :

Remarque : A chaque sémaphore est associée une structure:

```
struct sem{
unsigned short semval;  /* valeur du sémaphore */
short sempid;           /* PID du dernier processus ayant effectué
                           une opération sur ce sémaphore*/
unsigned short semncnt; /* nbre de processus en attente que
                           semval > valeur courante */
unsigned short semzcnt; /* nbre de processus en attente que
                           semval = 0 */
}
```

- **sembuf.sem_op > 0** : La valeur 'sembuf.sem_op' est ajoutée au compteur du sémaphore(sem.semval) et réveille les processus en attente.

❖ **sembuf.sem_op < 0** :

Si sem.semval **<** valeur absolue de sembuf.sem_op :

- Mettre le processus **en attente** jusqu'à ce que sem.semval **≥** valeur absolue de sembuf.sem.

  Lorsque cette condition sera vérifiée :

  - Ajouter la valeur sembuf.sem à sem.semval (décrémenter sem.semval).
  - Réveiller le processus.

❖ **sembuf.sem_op = 0** :

Si compteur du sémaphore(sem.semval) n'est pas égal à zéro,

- Mettre le processus **en attente** jusqu'à ce que le compteur du sémaphore soit égal à zéro(sem.semval=0).

- ✓ **Attribut Sem_flg** :
  - En général, on utilise la valeur 0(sem_flag=0);
  - **SEM_UNDO** : Restituer l'état initial après une fin anormale d'un processus. *(ne fonctionne pas bien sous linux : à vérifier.)*
    Le système sauvegarde toutes les opérations effectuées par chacun des processus sur le sémaphore et en cas de terminaison anormale d'un processus, le système exécute les opérations inverses de ce processus sur ce sémaphore.
  - **IPC_NOWAIT** : l'opération ne sera pas bloquante même si la valeur du champ sem_op est négattive ou nulle.

- ✓ **nb_op : nombre d'opérations à effectuer (1 à n).**

7.4.5 Destruction d'un sémaphore

- La fonction semctl() avec la 'commande IPC_RMID' permet de supprimer un ensemble de sémaphores :

  int semctl (int idf_sem, int num_sem, int commande, arg);

  - ✓ **idf_sem** :
  - ✓ **num_sem** : numéro du sémaphore que l'on veut supprimer. Dans le cas où l'opération concerne l'ensemble de tous les sémaphores et non un sémaphore particulier, ce paramètre peut-être est une valeur supérieure au égale au nombre de sémaphores de l'ensemble.
  - ✓ **Arg** : **0** , non utilisé.

7.4.6 Exemple : Sémaphores svide et splein initialisés à 0 et 10

```
#define   splein      0  /* Numéro du sémaphore splein */
#define   svide       1   /* Numéro du sémaphore svide */
/* Préparation des opérations P et V */
struct sembuf Psvide  = {svide, -1, 0};  /* P sur svide */
struct sembuf Vsvide  = {svide,  1, 0};  /* V sur svide */
struct sembuf Psplein = {splein, -1, 0}; /* P sur splein */
struct sembuf Vsplein = {splein,  1, 0}; /* V sur splein */

int sem;  /* Identificateur des sémaphores */
int main()
{
    union semun { int val;  struct semid_ds *buf;  unsigned short *array;
    } svideinit, spleininit;
    /* Création d'un tableau de 2 sémaphores splein et svide*/
    sem = semget(IPC_PRIVATE, 2, IPC_CREAT|0666);
    if(sem == -1) { printf("\nErreur de création des semaphore"); exit(1);}
```

```
/*  Initialisation du semaphore splein à 0 */
    spleininit.val = 0;  /* Valeur initiale de splein */
    semctl(sem, splein, SETVAL, spleininit);
/*  Initialisation du semaphore svide à 10 */
    svideinit.val = 10;   /* Valeur initiale de svide */
    semctl(sem, svide, SETVAL, svideinit);
    ...
/* Supprimer les sémaphores */
    semctl(sem, 2, IPC_RMID, 0);
}
```

Producteur()

```
...
/* p(svide)*/
   semop(sem, &Psvide, 1);
...

/* v(splein)*/
   semop(sem, &Vsplein, 1);
...
```

Consommateur()

```
...
/* p(splein)*/
   semop(sem, &Psplein, 1);
...

/* v(svide)*/
   semop(sem, &Vsvide, 1);
...
```

7.5 Gestion des IPC avec des commandes shell

Deux commandes : ipcs et ipcrm.

1. La commande shell ipcs permet de consulter les tables IPC du système.

- ipcs : consulter toutes les tables.
- ipcs –s : consulter la tables des sémaphores
- ipcs –m : consulter la table des segments de mémoire partagée.

2. La commande shell ipcrm permet de supprimer un segment de mémoire partagée ou un sémaphore.

   Le segment ou le sémaphore à supprimer peut être désignée soit
   - par la clé qui lui est associé (si clé différente de IPC_PRIVATE),
   - par son identificateur interne(identificateur affiché par ipcs).

- ipcrm {-m, -s } identificateur du segment ou du sémaphore
- ipcrm {-M, -S } clé du segment ou du sémaphore

Seul le propriétaire de l'objet peut demander sa suppression.

Pour plus de détails consulter le man de ipcs et ipcrm.