## Chapitre 4 : COMMUNICATION ENTRE PROCESSUS

La communication est un outil important qui permet de faire évoluer l'ensemble des processus dans un système.

Il existe plusieurs stratégies pour réaliser cette communication, nous citons :

- Variables partagées (modèles : producteur/ consommateur)
- Échange de messages (modèles : producteur/consommateur, client/serveur, Boite aux lettres)

**1. Communication par variables partagées (modèles : producteur/ consommateur) :**

Un modèle de communication entre processus avec partage de zone commune (tampon) est le ***modèle producteur-consommateur*.** Ce modèle a été décrit en détaille dans le chapitre 3 sur la synchronisation.

**2. Communication par échange de messages :**

La fonction du sous-système de messages est de permettre aux processus de communiquer sans avoir besoin de variables partagées. Il doit offrir deux opérations de base :

- Send(destination, &message)
- Receive (source , &message)

La liaison de communication peut être directe ou indirecte, unidirectionnelle ou bidirectionnelle.

**a. Communication directe :**

Tout processus qui désire communiquer avec ce mode, doit nommer explicitement le receveur ou l'expéditeur. Les primitives *Send* et *Receive* sont définis comme suit :

- Send(P, &message) : envoyer le « message » au processus P.
- Receive (Q , &message) : recevoir le « message » du processus Q.

**Exemple :**

Soit le problème producteur/ consommateur qui peut être décrit par le code suivant :

```
Producteur

        Répéter
                produire_objet (&objet)          /* produire un nouvel objet          */
                receive (consommateur , &m)      /* attendre un message vide          */
                faire_message (&m , objet)       /* construire un message à envoyer   */
                send (consommateur , &m)         /* envoyer le message                */
        Jusqu'à faux

consommateur
        send (producteur , &m)                   /* envoyer un messages vides  */
        Répéter
                receive (producteur , &m)        /* attendre un message        */
                retirer_objet (&m , &objet)      /* retirer l'objet du message */
                consommer_objet (objet)
                send (producteur , &m)           /* renvoyer une réponse vide  */
        Jusqu'à faux
```

**b. Communication indirecte :**

Les messages sont envoyés et reçus à travers des boites aux lettres (BAL). Ces dernières peuvent être vues comme des objets où l'on peut déposer et retirer des messages. Chaque boite aux lettres dispose d'un identificateur unique. Les processus peuvent alors communiquer à travers plusieurs boites aux lettres.

Les primitives *Send* et *Receive* sont définies comme suit :

- Send(A, &message) : envoyer le « message » à la BAL A avec ou sans blocage si pleine.
- Receive (A , &message) : recevoir le « message » de la BAL A avec ou sans blocage si vide.

Deux autres primitives sont nécessaires :

- BAL = CreationBAL(nom) : retourne le descripteur de la BAL crée.
- BAL = RelierBAL(nom) : recherche la BAL et retourne son descripteur.

On peut imaginer, pour le modèle producteur/ consommateur, une solution de type boîte aux lettres de capacité N messages, avec un producteur se bloquant si la boîte est pleine et un consommateur se bloquant si la boîte est vide.

## 3. Communication interprocessus sous UNIX :

Les moyens de communication interprocessus UNIX sont : Pipes (tubes), Signaux, Sockets, IPCs (Sémaphores, files de messages, mémoires partagées).

**a. Communication par Pipes (Tubes) :**

Les tubes (pipes) peuvent être considérés comme des fichiers temporaires. Ils permettent d'établir des liaisons unidirectionnelles de communication entre processus dépendants. En effet, Un tube de communication permet de mémoriser des informations et se comporte comme une file FIFO.

Un tube est caractérisé par :

- Deux descripteurs de fichiers (lecture et écriture).
- Sa taille limitée (d'où la notion de tube plein).
- L'opération de lecture dans un tube est destructrice : une information ne peut être lue qu'une seule fois dans un tube.

Il existe deux sortes de tube : les tubes ordinaires (anonymes) et les tubes nommés.

- **Tubes ordinaires :**

  Les tubes anonymes sont, en général, utilisés pour la communication entre un processus père et ses processus fils, avec un processus qui écrit sur le tube, appelé processus écrivain, et un autre qui lit à partir du tube, appelé processus lecteur.

  **Remarque** :

  Un tube anonyme n'a pas de référence dans le système de fichiers.

  La primitive de création d'un tube ordinaire est : pipe(p) ; avec int p[2] ;

  P[0] : correspond au descripteur en mode lecture

  P[1] : correspond au descripteur en mode écriture

Exemple d'utilisation des tubes ordinaires (*fichier pipe.c*):

```
# include <stdio.h>
# include <unistd.h>
int tube[2];
char buf[20];
main()
  {       pipe(tube);
           if (fork())
           { /* pere */
                    close (tube[0]);
                    write (tube[1], "bonjour", 8);
           }
          else
           { /* fils */
                    close (tube[1]);
                    read (tube[0], buf, 8);
                    printf ("%s bien recu \n", buf);
           }
}
```

L'exécution de ce programme ("pipe") donne le résultat suivant :
$ pipe
$ bonjour bien recu
$
***Question*** : Comment réaliser une communication bidirectionnelle avec les pipes ?
***Réponse*** : Utilisation de deux pipes entre deux processus.

- **Tubes nommés :**

Les tubes de communication nommés fonctionnent aussi comme des files de discipline FIFO (first in first out). Ils peuvent être utilisés par des processus indépendants ; à condition qu'ils s'exécutent sur une même machine.
Les tubes nommés sont créés par la commande « mkfifo» ou « mknod » ou par l'appel système mknod() ou mkfifo().

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *nomfichier, mode_t mode) ;
```

*nomfichier* : définit le chemin d'accès au tube et *mode* les droits d'accès des différents utilisateurs à cet objet.

**Exemple**

```
// programme writer.c envoie un message sur le tube mypipe
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
int main()
{  int fd;
   char message[100];
   sprintf(message, "bonjour du writer [%d]", getpid());
   fd = open("mypipe", O_WRONLY); //Ouverture du tube  mypipe en mode écriture
   printf("ici writer[%d] \n", getpid());
   if (fd!=-1)   write(fd, message, strlen(message)+1); // Dépot d'un message sur le tube
   else  printf( " désolé, le tube n' est pas disponible \n");
   close(fd);
   return 0;
}
```

```
// programme reader.c lit un message à partir du tube mypipe
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
int main()
{  int fd,n;
   char message[100];
   fd = open("mypipe", O_RDONLY);   // ouverture du tube mypipe en mode lecture
   printf("ici reader[%d] \n",getpid());
   if (fd!=-1)
    {  while ((n = read(fd,message,100))>0) // récupérer un message du tube, taille maximale est 100.
       printf("%s\n", message); // n est le nombre de caractères lus
   } else  printf( "désolé, le tube n' est pas disponible\n");
  close(fd);
  return 0;
}
```

**Remarque** :

- Si un processus ouvre un tube nommé en lecture (resp. écriture) alors qu'il n'y a aucun processus qui ait fait une ouverture en écriture (resp. lecture) alors, le processus sera bloqué jusqu'à ce qu'un processus effectue une ouverture en écriture (resp. en lecture).
- Attention au situation d'interblocage :

```
/* processus 1 */                    /* processus 2 */

int f1, f2;                          int f1, f2;
...                                  ....
f1 =open("fifo1", O_WRONLY);         f2=open("fifo2", O_WRONLY);
f2 =open("fifo2", O_RDONLY);         f1=open("fifo1", O_RDONLY);
...                                  ...
```

**b. Communication par *IPC* (Inter Process Communication) :**

Les IPC d'UNIX représentent trois outils de communication entre des processus situés sur une même machine.

- Les files de messages
- Les segments de mémoire partagée
- Les sémaphores

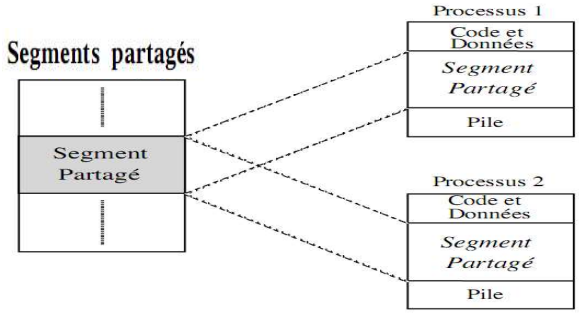
Les IPC ont un certain nombre de propriétés en commun :

- A chacun des trois mécanismes est affecté une table de taille prédéfinie.
- Dans chacune des tables toute entrée active est associée à une clé numérique choisie par l'utilisateur, et servant de nom d'identification global.
- Un appel système **'XXXget'** (*xxx → shm (mémoire partagée), msg (files de messages) ou sem (sémaphores)*) permet de créer une nouvelle entrée ou d'accéder une entrée existante.

**Remarque** : la commande `ipcs` *permet la consultation des tables d'IPC.*

```
$ipcs
IPC        ID     KEY          MODE          OWNER        GROUP
Messages Queues:
q          0      0x00000000   --rw-------   root         root
Shared Memory:
m          3      0x41442041   --rw-rw-rw    root         root
Semaphores:
s          1      0x4144314d   --ra-ra-ra-   root         root
```

1. **Les segments de mémoire partagée :** Les segments de mémoire partagée permettent à deux processus distincts de partager physiquement des données. Le partage est réalisé par l'espace d'adressage de chacun des processus, dont une zone pointe vers un segment de mémoire partagée. Lorsqu'un processus modifie la mémoire, tous les autres processus voient la modification.




Pour utiliser un segment de mémoire partagée, un processus doit allouer le segment. Puis, chaque processus désirant accéder au segment doit l'attacher. Après avoir fini d'utiliser le segment, chaque processus le détache.

**Remarque**

*Comme le noyau ne coordonne pas les accès à la mémoire partagée, vous devez mettre en place votre propre synchronisation. Par exemple, deux processus ne doivent pas écrire au même emplacement en même temps. Une stratégie courante pour éviter ces conditions de concurrence est d'utiliser des sémaphores.*

- Création d'un nouveau segment ou recherche de l'identifiant d'un segment déjà existant:

```
int segment_id = shmget (shm_key, getpagesize (), IPC_CREAT |
S_IRUSR | S_IWUSR);
```

- `shm_key`: spécifié la clé externe du segment de mémoire à créer. Il est utile de spécifier IPC_PRIVATE comme valeur de clé qui garantit qu'un nouveau segment mémoire est créé.
- `IPC_CREAT`: indicateur demande la création d'un nouveau segment.
- `S_IRUSR` et `S_IWUSR` spécifient des permissions de lecture et écriture pour le propriétaire du segment de mémoire partagée.

Si l'appel se passe bien, `shmget` renvoie un identifiant de segment. Si le segment de mémoire partagée existe déjà, les permissions d'accès sont vérifiées.

- Attachement d'un segment : Elle rend l'adresse à laquelle le segment a été attaché.

```
char*shared_memory= (char*)shmat(int shmid, void *adr, int
flags);
```

- `shmid`: identifiant du segment de mémoire partagée.
- `adr` : adresse d'attachement dans l'espace de processus. adr = 0 : e système choisit l'adresse d'attachement.

- Détachement d'un segment :

```
int shmdt(void *adr);
```

- `adr`: adresse du segment partagé attaché(adr renvoyée par shmat).

- rend 0 en cas de succès et -1 sinon.

♦ Suppression d'un segment :

```
int shmctl(int shmid, int cmd, shmid_ds *buf);
```

- `cmd` = IPC_RMID
- `buf`= NULL.

**Exemple** (fichier *shm.c*):

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
int main ()
{
     int segment_id; char* shared_memory; struct shmid_ds shmbuffer;
     int segment_size; const int shared_segment_size = 0x6400;

     /* Alloue le segment de mémoire partagée. */
     segment_id = shmget (IPC_PRIVATE, shared_segment_size, IPC_CREAT |
     S_IRUSR | S_IWUSR);
     /* Attache le segment de mémoire partagée. */
     shared_memory = (char*) shmat (segment_id, 0, 0);
     printf ("mémoire partagée attachée à l'adresse %p\n", shared_memory);

     /* Écrit une chaîne dans le segment de mémoire partagée. */
     sprintf (shared_memory, "Hello, world.");
     /* Détache le segment de mémoire partagée. */
     shmdt (shared_memory);

     /* Réattache le segment de mémoire partagée à une adresse différente.*/
     shared_memory = (char*) shmat (segment_id, (void*) 0x5000000, 0);
     printf ("mémoire partagée réattachée à l'adresse %p\n", shared_memory);
     /* Affiche la chaîne de la mémoire partagée. */
     printf ("%s\n", shared_memory);
     /* Détache le segment de mémoire partagée. */
     shmdt (shared_memory);
     /* Libère le segment de mémoire partagée. */
     shmctl (segment_id, IPC_RMID, 0);
     return 0;
}
```

**2. Les files de messages :**

C'est une implantation UNIX du concept de boîte aux lettres, qui permet la communication indirecte entre des processus. Les messages étant typés, chaque processus peut choisir les messages qu'il veut lire (extraire de la file).

Les informations sont stockées dans les files de messages en **DATAGRAM** (un message à la fois). Les files sont de type **FIFO** (**F**irst **I**n, **F**irst **O**ut).

Un processus peut émettre des messages vers plusieurs processus, par l'intermédiare d'une même file de message (*multiplexage*).

♦ Création ou recherche des files de messages :

```
int msgid = msgget(key_t cle, int flags);
```

**NB**: les arguments de la primitive `msgget` sont les mêmes que celle du `shmget`.

Si l'appel se passe bien, `msgget` renvoie un identifiant de file de message crée. Si la file de message existe déjà, les permissions d'accès sont vérifiées.

- Émission des messages :

```
int msgsnd(int msgid, struct msgbuf *msg,int taille,int flags);
```

où :

```
struct msgbuf { /* définie dans sys/msg.h */
                    long mtype;  /* type du message */
                    char mtext[1]; /* texte du message */
                };
```

Avec possibilité de redéfinir cette structure en fonction de ses besoins.

Cette primitive permet à un processus d'envoyer le message, `msg`, dans la file de message spécifié par l'id `msgid`. Elle retourne 0 en cas de succès et -1 sinon.

**Remarque** : *La primitive `msgsnd` est une fonction bloquante i.e. si la file de message est plein, le processus est suspendu jusqu'à extraction de messages de la file ou bien suppression du système de la file (retourne -1). Si* `IPC_NOWAIT → flags`, et que la file est pleine, la fonction devient non bloquante. Donc, le message ne sera pas envoyé et le processus peut reprendre la main immédiatement.

- Réception des messages :

```
int msgrcv(int msgid, struct msgbuf *msg, int taille, long
type, int flags);
```

La primitive `msgrcv` permet de retirer un message de la file en fonction de son type. Comme pour `msgsnd`, `IPC_NOWAIT` rend l'appel non bloquant.

- `type = 0` : le premier message est lu
- `type>0` : le premier message de type msgtype est lu
- `type<0` : le premier message dont le type est inférieur ou égal à |msgtyp| est lu

- Suppression des files de messages :

```
int msgctl(int msgid, int cmd, msqid_ds *buf);
```

- cmd = `IPC_RMID`.
- `Buf = NULL`.

**Exemple** (fichier *shm.c*):

```
#include <sys/ipc.h>
#include <sys/msg.h>
…
int msg_id; struct msqid_ds *buf;
struct message {
                long type;
                char texte[128];
               } msg;
…
msg_id = msgget (IPC_PRIVATE, IPC_CREAT);
…
msg.type = 1;
for ( ; ; ) {
printf ( "Entrer le texte a emettre \n");
scanf("%s",msg.texte);
msgsnd(msg_id , &msg , sizeof( struct message ) , 0);
…
}
```

**3. Les sémaphores :**

Tous les problèmes de synchronisation ne sont pas solubles avec les sémaphores simples de Djiskstra, tels qu'ils ont été décrits précédemment.

**Exemple** : la demande de **m** ressources différentes, dans un ensemble de **n** ressources (**m** <= **n**).

Soit les **n** sémaphores **S1, S2, ..., Sn** permettant de bloquer les n ressources. Si **P1** demande **R1, R2, R3** et **R4** et **P2** demande **R2, R3, R4** et **R5** alors on a :

```
P1 -> P(S1), P(S2), P(S3), P(S4) et
P2 -> P(S3), P(S4), P(S5), P(S2)
```

On a un risque d'**interblocage**.

**Solution** : Si les 4 opérations P étaient atomiques, on éviterait les **interblocages** (**deadlock**) :

```
P1 -> P(S1,S2,S3,S4) et
P2 -> P(S3,S4,S5,S2)
```

**Idée** : *ensemble de sémaphores*.

- Création ou recherche de l'identifiant d'un ens. de sémaphores :

```
int sem_id = semget(key_t cle, int nsems, int flags);
```

- `nsem`: nombre de sémaphores à créer dans le groupe.

Cette fonction renvoie l'id. de l'ens. des sémaphores ds la table en cas de succès (> 0) -1 sinon.

- Initialisation des sémaphores :

```
int semctl (int semid, int semnum, int cmd, union semun arg);
```

- `semnum` : numéro du sémaphore choisi dans le groupe (0 pour le premier).
- `cmd` : type d'opération à effectuer sur le sémaphore, par exemple SETVAL pour initialiser le sémaphore à la valeur contenue dans "arg".
- `arg` : sert à passer des arguments aux commandes exécutées par "cmd".

- Opérations sur les sémaphores :

```
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

structure d'une operation :

```
struct sembuf {
ushort sem_num; /* N° du semaphore*/
short sem_op; /* opération sur le sémaphore */
shrot sem_flag;  /* options */
};
```

- `sops` : tableau de structures qui contient la liste des opérations.

Cette fonction retourne 0 en cas de succès et -1 sinon.

L'opération semop est en principe bloquante, i.e. le processus est mis en sommeil si l'une des opérations de l'ensemble ne peut être effectuée. Dans ce cas, aucune opération de l'ensemble n'est réalisée.

Une opération élémentaire comporte un numéro de sémaphore et une opération qui peut avoir trois valeurs :

- ✓ une `sem_op` > 0, pour une opération V`sem_op`, (`semval = semval + sem_op`)
  Tous les processus en attente d'une augmentation du sémaphore sont reveilles.
- ✓ une `sem_op` = 0, teste si le sémaphore a la valeur 0. Si ce n'est pas le cas, le processus est mis en attente de la mise à zéro du sémaphore.
- ✓ une `sem_op` < 0, pour une opération P`sem_op` (`semval = semval - |sem_op|`)
  Si le résultat est :
  ***nul***: tous les processus en attente de cet événement sont réveillés
  ***négatif***: le processus est mis en attente d'une augmentation du sémaphore

**Exemple** (*semEM.c*) :

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

#define SEM_EXCL_MUT 0
#define NB_SEM 1
int FLAGS = IPC_CREAT; int  sem_id;
struct  sembuf  op;
main ( )
{
      sem_id = semget (IPC_PRIVATE, NB_SEM, FLAGS );
      semctl(sem_id, SEM_EXCL_MUT, SETVAL, 1);
      …
      P(SEM_EXCL_MUT);
      /* Section Critique */
      V(SEM_EXCL_MUT);
      …
      semctl(sem_id, SEM_EXCL_MUT, IPC_RMID, 0);
}

void P (int sem)              /* Primitive P() sur sémaphores */
{                             /* sem = Identifiant du sémaphore */
  op.sem_num = sem;           /* Identification du sémaphore impliqué */
  op.sem_op = -1;             /* Définition de l'opération à réaliser */
  op.sem_flg = SEM_UNDO;      /* Positionnement du bit SEM_UNDO */
  semop (sem_id, &op, 1);     /* Exécution de l'opération définie */
};

void V(int sem) /* Primitive V() sur sémaphores */
{
  op.sem_num = sem;
  op.sem_op = 1;
  op.sem_flg = SEM_UNDO;
  semop (sem_id, &op, 1);
};
```

**Exercice (*Rendez-vous de processus*)** :

Soit une application décomposée en **N** modules qui s'exécuteront de manière concurrente et ont la particularité d'être constitués de deux parties logiques **Ai:Bi**.

L'objectif est de faire en sorte que les **N** processus ne commencent l'exécution des séquences **Bi**, uniquement (point de rendez-vous) lorsque toutes les séquences **Ai** auront été complètement exécutées. Cela signifie que les processus doivent se synchroniser sur le processus **j** le plus lent à exécuter sa séquence **Aj** ; il est donc nécessaire de mettre en place entre les séquences **Ai** et **Bi** un mécanisme bloquant les processus quand ils terminent leur séquence **Ai**.

- Communication dans les langages évolués (CSP, ADA, JAVA..)

**Bibliographie :**

[1] J-L. Peterson, F. Silbershartz, P. B.Galivin « Operating systems concepts » Fourth Edition.

[2] Crocus, “Systèmes d’exploitation des ordinateurs” Dunod informatique 1975.

[3] J. Beauquier, B. Berard « Systèmes d’exploitation : concepts et algorithmes » McGraw Hill 1990

[4] A. Silberschatz, P. B. Galvin “ Principes des systèmes d’exploitation,“ 4e Edition, Addison Wessley

[5] Andrew S. Tanenbaum, “Moderrn operating systems,” Second Edition Prentice Hall.

[6] Maurice J.Bach, traduit par G.Feallah, “Conception du système UNIX, » Masson

et Prentice Hall 1990.