

SUPPORT DE COURS
SYSTEMES D'EXPLOITATION
COMMUNICATION INTER PROCESSUS

1. Introduction

Inter-Process Communication (IPC) : est l'un des aspects les plus importants (et aussi les plus délicats) de la programmation de système, il permet à des processus de communiquer entre eux. Pour voir comment l'IPC fonctionne en pratique, considérons un exemple simple, mais courant, un *spooler d'impression*. Lorsqu'un processus veut imprimer un fichier, il entre le nom du fichier dans un *répertoire* spécial du *spooler*. Un autre processus, le *démon d'impression*, vérifie périodiquement s'il y a des fichiers à imprimer, et s'il y en a, il les envoie à l'imprimante et supprime leurs noms du répertoire.

La communication inter-processus (IPC) est un mécanisme qui permet aux processus de communiquer entre eux et de synchroniser leurs actions. La communication entre ces processus peut être vue comme une méthode de coopération entre eux.

2. Les mécanismes de communication :

2.1 Messages : une file d'attente de messages, de taille finie, de type chaîne de caractère qui relie deux processus. Le processus émetteur crée un message avec la primitive : `msgget()` et envoie le message avec la primitive : `msgsnd()`. Le processus récepteur retire le message avec la primitive `msgrcv()` et c'est le SE qui assure l'accès exclusif à la file d'attente. La lecture d'un message est destructrice.

Création d'une file de messages – `msgget()`

Cet appel système accepte deux paramètres : une clé et des drapeaux (flags).

La clé peut être :

- `IPC_PRIVATE` : utilisée pour créer une file de messages privée.
- un entier positif : utilisé pour créer (ou accéder à) une file de messages accessible publiquement.

Le second paramètre contient des drapeaux qui régissent le traitement de l'appel système. Il peut inclure des drapeaux tels que `IPC_CREAT` (Crée un nouvel objet s'il n'existe pas) et/ou `IPC_EXCL` (provoque l'échec de `msgget()` si l'objet existe déjà), ce comportement est analogue à celui de `O_CREAT` et `O_EXCL` dans l'appel système `open()`, ainsi que des bits de droits d'accès. Les 9 bits de poids faible servent à définir les droits d'accès à la file, tout comme les 9 bits correspondants contrôlent l'accès aux fichiers. Ces bits sont répartis en trois groupes : propriétaire (user), groupe (group) et autres (others). Dans chaque groupe, le premier bit correspond au droit de lecture, le deuxième au droit d'écriture, et le troisième est ignoré (la notion de droit d'exécution ne s'applique pas aux files de messages).

Exemple :

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
/* Création file de messages privée, accessible uniquement au propriétaire. */
```

```
int queue_id = msgget(IPC_PRIVATE, 0600); /* nombre octal. */
if (queue_id == -1) {
    perror("msgget");
    exit(1);
}
```

1. L'appel système renvoie un identifiant entier du fichier d'attente créé. Par la suite, nous pourrions utiliser cette clé pour accéder à la file d'attente afin d'y lire et d'y écrire des messages.
2. Le fichier d'attente créé appartient à l'utilisateur dont le processus a effectué la création. Ainsi, les bits de permission étant fixés à « 0600 », seuls les processus s'exécutant pour le compte de cet utilisateur auront accès au fichier d'attente.

La structure du message - struct msgbuf

```
struct msgbuf {
    long mtype; /*nombre positif*/
    char mtext[1];
};
```

Exemple :

```
char* msg_text = "hello world";
/* Allouer suffisamment d'espace pour la longueur de la chaîne et un octet supplémentaire pour
le caractère nul de fin.*/
struct msgbuf* msg = (struct msgbuf*)malloc(sizeof(struct msgbuf) + strlen(msg_text));
msg->mtype = 1;
strcpy(msg->mtext, msg_text);
```

Écriture de messages dans une file – msgsnd()

Cet appel système copie la structure de message et la place en tant que dernier message de la file d'attente. Il accepte les paramètres suivants :

1. int msqid – identifiant de la file de messages, tel que renvoyé par l'appel msgget().
2. struct msgbuf* msg – pointeur vers une structure de message correctement initialisée, comme celle préparée dans la section précédente.
3. int msgsz – taille de la partie données (mtext) du message, en octets.
4. int msgflg – drapeaux spécifiant le mode d'envoi du message. IPC_NOWAIT : si le message ne peut pas être envoyé immédiatement sans bloquer le processus, renvoyer « -1 » et affecter EAGAIN à errno.

Pour ne spécifier aucun drapeau, on utilise la valeur « 0 ».

```
int rc = msgsnd(queue_id, msg, strlen(msg_text)+1, 0);
if (rc == -1) {
    perror("msgsnd");
    exit(1);
}
```

Lecture d'un message depuis la file - msgrcv()

Cet appel système accepte la liste de paramètres suivante :

1. int msqid : identifiant de la file d'attente, tel que renvoyé par msgget().
2. struct msgbuf* msg : un pointeur vers une structure msgbuf pré-allouée. Elle doit généralement être suffisamment grande pour contenir un message incluant des données arbitraires.
3. int msgsz : taille du texte du message le plus volumineux que l'on souhaite recevoir. Elle ne doit PAS dépasser l'espace alloué pour le texte du message dans 'msg'.

4. `int msgtyp` : Type de message que l'on souhaite lire. Il peut s'agir de l'une des valeurs suivantes :
- 0 : Le premier message de la file d'attente est renvoyé.
 - un entier positif : le premier message de la file d'attente dont le type (`mtype`) correspond à cet entier (sauf si un certain drapeau est défini dans `msgflg`, voir ci-dessous).
 - un entier négatif : le premier message de la file d'attente dont le type est inférieur ou égal à la valeur absolue de cet entier.
5. `int msgflg` : une combinaison par « OU » logique de l'un des drapeaux suivants :
- `IPC_NOWAIT` : si aucun message dans la file d'attente ne correspond à ce que l'on souhaite lire, renvoyer '-1' et affecter `ENOMSG` à `errno`.
 - `MSG_EXCEPT` : si le paramètre de type de message est un entier positif, renvoyer le premier message dont le type n'est PAS égal à l'entier donné.
 - `MSG_NOERROR` : Si un message dont la partie texte est plus grande que '`msgsz`' correspond à ce que l'on souhaite lire, tronquer le texte lors de la copie du message dans la structure `msgbuf`. Si ce drapeau n'est pas défini et que le texte du message est trop volumineux, l'appel système renvoie '-1' et `errno` est défini sur `E2BIG`.

Exemple :

```
struct msgbuf* recv_msg = (struct msgbuf*)malloc(sizeof(struct msgbuf)+strlen("hello world"));
int rc = msgrcv(queue_id, recv_msg, strlen("hello world")+1, 0, 0);
/* '0' pour type de message (any) et pas de flags (0). */
if (rc == -1) {
    perror("msgrcv");
    exit(1);
}
```

1. Si le message dans la file d'attente était plus grand que la taille de "hello world" (plus un), il y'aura erreur et `exit`.
2. S'il n'y avait aucun message dans la file d'attente, l'appel à `msgrcv()` aurait bloqué le processus jusqu'à ce que l'un des événements suivants se produise :
 - un message approprié a été placé dans la file d'attente.
 - la file d'attente a été supprimée (et alors `errno` serait défini sur `EIDRM`).
 - le processus a reçu un signal (et alors `errno` serait défini sur `EINTR`).

Exemple complet.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>
#include <string.h>

struct msgbuf {
    long mtype;
    char mtext[1];
};

int main(int argc, char* argv[]) {
    int queue_id = msgget(IPC_PRIVATE, 0600);
    struct msgbuf* msg;
    struct msgbuf* recv_msg;
```

```

int rc;

if (queue_id == -1) {
    perror("main: msgget");
    exit(1);
}
printf("IPC_PRIVATE=%d\n",IPC_PRIVATE);
printf("file de messages crée, file id '%d'.\n", queue_id);
msg = (struct msgbuf*)malloc(sizeof(struct msgbuf)+strlen("hello world"));
msg->mtype = 1;
strcpy(msg->mtext, "hello world");
rc = msgsnd(queue_id, msg, strlen(msg->mtext)+1, 0);
if (rc == -1) {
    perror("main: msgsnd");
    exit(1);
}
free(msg);
printf("message dans le file\n");
recv_msg = (struct msgbuf*)malloc(sizeof(struct msgbuf)+strlen("hello world"));
rc = msgrcv(queue_id, recv_msg, strlen("hello world")+1, 0, 0);
if (rc == -1) {
    perror("main: msgrcv");
    exit(1);
}

printf("msgrcv: message reçu: mtype '%d'; mtext '%s'\n", recv_msg->mtype, recv_msg->mtext);

return 0;
}

```

2.2 Mémoire partagée : l'outil permet de mettre en commun un segment mémoire entre plusieurs processus (ce qui est normalement interdit) et la synchronisation est à réaliser par le programmeur. Pour pouvoir partager de la mémoire, on doit utiliser les primitives suivantes :

1. shmget() (SHared Memory GET) construit le segment de mémoire partagée ;
2. shmctl() permet d'intervenir sur ce segment ;
3. shmat() permet à un processus de s'attacher ce segment ;
4. shmdt() permet à un processus de se détacher du segment.

Exemple d'utilisation de shmget() : Une fonction creeSegment() prenant comme arguments :

1. la taille de la mémoire à créer ;
2. une chaîne de caractères indiquant le nom du fichier associé à la fonction principale appelante ("main") ;
3. un entier qui associé au fichier permet de construire une clé identifiant le segment.

```

#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

```

```

int creerSegment(int size, char *name, int cle){

    int shmid ; /* l'identificateur de la memoire partagee */
    key_t clef ; /* la clef associee au segment */

    /* L'instruction ftok(name,(key_t) cle) permet de construire une cle identifiant le segment */
    clef = ftok(name,(key_t) cle) ;

    /*IPC_CREAT|IPC_EXCL|SHM_R|SHM_W permet d'indiquer les droits d'accès */

    shmid = shmget( clef, size, IPC_CREAT|IPC_EXCL|SHM_R|SHM_W ) ;
    if ( shmid== -1 ) {
        perror("La creation du segment de memoire partage a echouee") ;
        exit(1) ;
    }
    printf("l'identificateur du segment est %d \n",shmid) ;
    printf("ce segment est associe a la clef %d \n",clef) ;

    return shmid ;
}

int main (void){
    char *name ;
    name = "sharedMemory.c" ;
    creerSegment(100,name,1) ;
    return 1 ;
}

```

Remarque : Utiliser l'instruction shell `ipcs` pour avoir de l'information sur la mémoire partagée créée. Si on exécute le même code une seconde fois, l'opération échoue car ce segment de mémoire n'est pas détruit.

Contrôle du segment de mémoire partagée : `shmctl`

Elle est utilisée pour examiner et/ou modifier des informations sur le segment de mémoire partagée.

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

La structure de contrôle est du type `shmid_ds` définie comme suit :

```

struct shmid_ds {
    struct ipc_perm shm_perm;    /*structure des permissions*/
    size_t shm_segsz;           /* taille du segment en bytes */
    __time_t shm_atime;         /* date du dernier shmat() */
    unsigned long int __unused1;
    __time_t shm_dtime;         /* date du dernier shmdt() */
    unsigned long int __unused2;
    __time_t shm_ctime;         /*date de la dernière modification par shmctl() */
    unsigned long int __unused3;
    __pid_t shm_cpid;           /* pid du createur */
    __pid_t shm_lpid;           /* pid du dernier shmop */
    shmatt_t shm_nattch;        /* nombre d'attaches courant */
}

```

```

    unsigned long int __unused4;
    unsigned long int __unused5;
};
avec
struct ipc_perm
{
    __kernel_key_t key;
    __kernel_uid_t uid;
    __kernel_gid_t gid;
    __kernel_uid_t cuid;
    __kernel_gid_t cgid;
    __kernel_mode_t mode;
    unsigned short seq;
};

```

La primitive `shmctl()` permet de détruire un segment de mémoire partagée.

Les valeurs autorisées pour *cmd* sont (la liste est exhaustive) :

IPC_STAT : Copier les informations de la structure de données noyau associée à *shmid* dans la structure *shmid_ds* pointée par *buf*. L'appelant doit avoir la permission de lecture sur le segment de mémoire partagée.

IPC_RMID : Marquer le segment pour être détruit. Il ne sera détruit effectivement qu'après le dernier détachement (c'est-à-dire quand le membre *shm_nattch* de la structure *shmid_ds* associée vaudra zéro.)

Exemple : Les fonctions `infoSegment` et `détruitSegment` qui prennent en argument l'identificateur du segment sur lequel elles agissent.

```

#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

```

```

int creerSegment (int size, char *name, int cle){

```

```

    int shmid ;
    key_t clef ;

```

```

    clef = ftok(name,int projet-id) ; /* elle permet la creation d'une clé identifiant le segment. elle
    convertit le nom du fichier et un id du projet (huit bits ;peut etre 0) en une clé IPC*/

```

```

    shmid = shmget( clef, size, IPC_CREAT|IPC_EXCL|SHM_R|SHM_W ) ;
    if ( shmid== -1 )
    {
        perror("La creation du segment de memoire partage a echouee") ;
        exit(1) ;
    }

```

```

    printf("l'identificateur du segment est %d \n",shmid) ;

```

```

printf("ce segment est associe a la clef %d \n",clef) ;

return shmid ;
}

void infoSegment(int shmid) {
    struct shmid_ds buf ;

    if (shmctl(shmid,IPC_STAT,&buf) == -1)
    {
        perror("Erreur lors de l'utilisation de infoSegment") ;
        exit(1) ;
    }

    printf("Affichage des champs de controles\n") ;
    printf("identificateur du proprietaire : %d\n",buf.shm_perm.uid) ;
    printf("identificateur du groupe du proprietaire : %d\n",buf.shm_perm.gid) ;
    printf("identificateur du createur : %d\n",buf.shm_perm.cuid) ;
    printf("identificateur du groupe du createur : %d\n",buf.shm_perm.cgid) ;
    printf("mode d'accès : %d\n",buf.shm_perm.mode) ;
    printf("taille du segment : %d\n",(int) buf.shm_segsz) ;
}

void detruitSegment(int shmid) {
    if (shmctl(shmid,IPC_RMID,NULL) == -1)
    {
        perror("Erreur lors de la destruction") ;
        exit(1) ;
    }
}

int main(void){
    int shmid ;
    char *name ;

    name = "sharedMemory.c" ;
    shmid = creerSegment(100,name,1) ;
    infoSegment(shmid) ;
    detruitSegment(shmid) ;
    return 1 ;
}

```

Accès au segment de mémoire partagée : shmat et shmdt

Avant qu'un processus puisse utiliser un segment de mémoire partagée, ce dernier doit être *attaché* au processus grâce à la primitive `shmat()`. La primitive `shmdt()` permet au processus de se détacher du segment.

La fonction `shmat` retourne l'adresse du segment de mémoire partagée. Il n'y a pas copie de zone mémoire mais redirection de l'adressage vers le segment partagé.

Avec ces appels, on peut construire un processus père qui crée un segment de mémoire partagé puis crée un fils. Ce fils place une chaîne de caractères dans la mémoire partagée et meurt. Le père affiche cette chaîne et meurt.

Exemple complet : On regroupe les différentes fonctions construites précédemment sous la forme d'un fichier objet avec son entête sharedMemory.h et utiliser le code suivant :

```
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include "sharedMemory.h"
#include <string.h>

int main(void){

    int pid ;
    char *mem ;
    int shmid ; /* l'identificateur du segment de mémoire partagée */
    int flag = 0 ; /* un drapeau attaché au segment */
    char *name ;

    /* création du segment de mémoire partagée */

    name = "sharedMemory.c" ;
    shmid = creerSegment(100,name,1) ;

    switch(pid =fork()) {
        case -1 :
        {
            perror("impossible de creer un fils") ;
            return -1 ;
        }
        case 0 :
        {

            printf("Salut, je suis le fils %d\n",getpid()) ;
            printf("Je commence par m'attacher le segment de memoire\n") ;
            mem = shmat(shmid,0,flag) ;

            if (mem ==(char*)-1)
            {
                perror("attachement impossible") ;
                exit(1) ;
            }
            printf("je vais ecrire un message que le pere va afficher\n") ;
            strcpy(mem,"La vie est belle") ;

            if(shmdt(mem)==-1)
            {
                perror("detachment impossible") ;
            }
        }
    }
```

```

        exit(1) ;
    }

return 1 ;
}
default :
{
printf("Je suis le pere\n") ;
printf("Je commence par m'attacher le segment de memoire\n") ;
mem = shmat(shmid,0,flag) ;

if (mem ==(char*)-1)
{
    perror("attachement impossible") ;
    exit(1) ;
}

printf("je vais afficher un message que le fils a ecrit\n") ;
printf("%s\n",mem) ;

if(shmdt(mem)==-1)
{
    perror("detachment impossible") ;
    exit(1) ;
}
}

printf("Le pere attend la mort de son fils\n") ;
wait(0) ;
detruitSegment(shmid) ;
return 1 ;
}

```

Question: Comment faire pour que le fils s'exécute toujours avant le père ?

2.3 Les tubes de communication (Pipe) : Un tube est un tampon de données (fichier) dont le quel deux processus peuvent successivement lire et écrire en exclusion mutuelle.

exemple : pipe() de Linux(Unix).

Deux types de tubes : anonymes et nommés

Tubes anonymes :

Un tube est crée par cette primitive : `int pipe(int descripteur[2]);`

Elle renvoie une valeur de type entier , qui est de 0 si elle réussit, ou une autre valeur dans le cas contraire.

Elle prend en argument un tableau d'entiers, comprenant :

- descripteur[0]: désigne la sortie du tube ;
- descripteur[1]: désigne l'entrée du tube.

Comme on ne crée le tube que dans un seul processus, l'autre processus doit en connaître l'entrée ou la sortie ! En conséquence, il faut utiliser pipe avant d'utiliser fork. Ensuite, le père et le fils auront les mêmes valeurs dans leur tableau descripteur, et pourront donc communiquer.

Pour écrire dans un tube :

```
ssize_t write(int entreeTube, const void *elementAEcrire, size_t
nombreOctetsAEcrire);
```

La fonction prend en paramètre l'entrée du tube (descripteur[1]), un pointeur générique vers la mémoire contenant l'élément à écrire, ainsi que le nombre d'octets de cet élément. Elle renvoie une valeur de type ssize_t correspondant au nombre d'octets effectivement écrits (int).

Et pour lire dans un tube :

```
ssize_t read(int sortieTube, void *elementALire, size_t nombreOctetsALire);
```

La fonction prend en paramètre la sortie du tube (descripteur[0]), un pointeur vers la mémoire contenant l'élément à lire et le nombre d'octets de cet élément. Elle renvoie une valeur de type ssize_t qui correspond au nombre d'octets effectivement lus.

Remarques :

- la lecture est bloquante (si tube vide ou occupé)
- le signal SIGPIPE est envoyé au rédacteur si aucun tube n'est ouvert en lecture.

Fermeture du tube :

```
int close(int descripteur);
```

La valeur prise en paramètre correspond au descripteur de l'entrée ou de la sortie à fermer.

Exemple1 : un programme qui crée deux processus : le père écrit le message « Bonjour, fils. Je suis ton père ! ». Le fils le récupère, puis l'affiche.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

#define TAILLE_MESSAGE 256 /*la taille de la chaîne à lire et à écrire */

int main(void)
{
    pid_t pid_fils;
    int descripteurTube[2];
    unsigned char messageLire[TAILLE_MESSAGE], messageEcrire[TAILLE_MESSAGE];
    printf("Création du tube.\n");
    if(pipe(descripteurTube) != 0)
    {
        fprintf(stderr, "Erreur de création du tube.\n");
        return EXIT_FAILURE;
    }

    pid_fils = fork();
    if(pid_fils == -1)
    {
        fprintf(stderr, "Erreur de création du processus.\n");
        return 1;
    }

    if(pid_fils == 0)
    {
        printf("Fermeture de l'entrée dans le fils.\n\n");
        close(descripteurTube[1]);
        read(descripteurTube[0], messageLire, TAILLE_MESSAGE);
        printf("Nous sommes dans le fils (pid = %d).\nIl a reçu le message
suivant du père : \"%s\".\n\n", getpid(), messageLire);
```

```

    }

    else
    {
        printf("\nFermeture de la sortie dans le père.\n");
        close(descripteurTube[0]);
        sprintf(messageEcrire, "Bonjour, fils. Je suis ton père !");
        printf("Nous sommes dans le père (pid = %d).\n Il envoie le message
suivant au fils : \"%s\".\n\n\n", getpid(), messageEcrire);
        write(descripteurTube[1], messageEcrire, TAILLE_MESSAGE);
        wait(NULL);
    }
    return 0;
}

```

Tubes nommés :

L'inconvénient des tubes classiques et qu'il faut obligatoirement que les processus connaissent le processus qui a créé le tube. C'est pourquoi une "extension" des tubes a été créée, appelée "**tube nommé**" (*named pipe*). Le tube dispose d'un nom dans le système de fichier.

Pour créer un tube nommé : `int mkfifo(const char* nom, mode_t mode);`

Le premier argument est le nom du tube nommé. On donne généralement l'extension « .fifo » au nom du tube nommé, le deuxième paramètre sont des droits d'accès du tube (exemple : 0720).

Ensuite, ouvrir l'entrée/la sortie du tube:

```
int open (const char* cheminFichier, int options);
```

La fonction renvoie une valeur de type entier que l'on attribue à l'extrémité du tube en question. Le premier argument est le nom du fichier, le second argument indique si c'est l'entrée ou la sortie du tube. Il existe deux constantes pour cela, déclarées dans <fcntl.h>:

- O_WRONLY: pour l'entrée ;
- O_RDONLY: pour la sortie.

Ensuite, on peut écrire et lire avec write et read comme si c'était des tubes classiques.

- Ouverture en lecture : le processus est bloqué jusqu'à ce qu'un processus ait ouvert en écriture
- Ouverture en écriture : le processus est bloqué jusqu'à ce qu'un processus ait ouvert en lecture

Exemple2 :

Fichier emetteur :	Fichier recepteur
<pre> #include <stdio.h> #include <fcntl.h> int main() { int fd; char nomTube[] = "essai.fifo"; mkfifo(nomTube , 0644) fd=open(nomTube, O_WRONLY); write(fd, "un message", 10); printf("%d fin ecriture", getpid()); } </pre>	<pre> #include <stdio.h> #include <fcntl.h> #include <string.h> int main() { int fd; char buf[50]; fd=open("essai.fifo", O_RDONLY); read(fd, buf, sizeof(buf)); printf("%d>recu,%s", getpid(), buf); } </pre>

2.4 Communication par rendez-vous (RDV): proposé par Hoare en 1978 pour une synchronisation forte.

Soient P, Q : processus ; e : expression algébrique ; v : variable :

La commande d'envoi : P ! e

La commande de réception : Q ? v

Le processus P exécute l'envoi et la valeur de l'expression est mise dans v pour être reçue par le processus Q qui exécute la réception simultanément.

La commande d'envoi bloque le processus P jusqu'à réception et la commande de réception bloque le processus Q jusqu'à émission : donc un RDV est établi entre P et Q.

Les langages de programmation parallèle CSP, Modula-2 ou ADA proposent ce genre de synchronisation.

Le modèle ADA : Le langage ADA propose la notion de tâche au lieu d processus (task). Chaque service assuré par une tâche est un point d'entrée (entry) auquel on associe une file d'attente de tâches appelantes. L'ensemble des entrées d'une tâche constitue une interface et le traitement réalisé est implémenté par une unité appelée corps (body).

Une entrée d'une tâche peut être connectée à une interruption matérielle qui la rend une routine d'interruption.

```
task [type] <nom-tache>[is
    {entry<nom-entrée> [" («déclarations des paramètres formels »)"] ;}
end<nom-tache> ;
```

Les tâches s'exécutent en parallèle et utilisent une synchronisation par RDV.

La tâche appelante exécute l'appel à une entrée et reste bloqué jusqu'à satisfaction du service.

```
<tache-appelée>.<nom-entrée>[" ("paramètres effectifs")"] ;
```

La tâche appelée indique l'acceptation du RDV et reste bloqué jusqu'à avoir le RDV.

```
accept<nom-entrée>[" ("paramètres formels")"]
    [do<instructions> ;]
end<nom-entrée> ;
```

Exemple : soit une tâche mutex pour réaliser l'exclusion mutuelle, alors la tâche appelante va bloquer sur mutex.entrer et la tâche appelée va bloquer sur acceptentrer jusqu'au RDV.

2.5 Sockets :

Les sockets (interface socket) sont utilisées pour faire communiquer deux processus sur deux machines distantes. Le processus émetteur prépare un message dans son espace d'adressage puis exécute un appel système pour transmettre le message à un processus récepteur. La communication est bidirectionnelle et utilise soit le mode connecté (TCP) soit le mode non connecté (UDP).

A l'intérieur d'un processus, une socket sera identifiée par un descripteur de même nature que ceux identifiant les fichiers (c'est-à-dire dans le même ensemble). Cette propriété est essentielle car elle permet la redirection des fichiers d'entrée-sortie standard (descripteurs 0, 1 et 2) sur des sockets et donc l'utilisation d'applications standards sur le réseau.

A toute socket est associée dans le système un objet ayant la structure socket, définie dans le fichier standard <sys/socketvar.h> qui contient l'ensemble des caractéristiques de cet objet.