1 Les conditions

Lorsqu'un thread doit attendre jusqu'à ce qu'un événement survienne dans un autre thread, on emploie une technique appelée la **condition**.

Quand un thread est en attente d'une condition, il reste bloqué tant que celle-ci n'est pas réalisée par un autre thread. Un autre thread peut alors signaler la condition (c'est à dire déclarer la condition réalisée !).

1.1 Déclaration de la condition

On doit déclarer la condition , de cette manière :

```
pthread_cond_t condition;
```

1.2 Initialisation de la variable condition

La condition doit être initialisée avec la primitive **pthread_cond_init()** dont le prototype est :

```
int pthread_cond_init(pthread_cond_t* cond, const pthread_condattr_t* attr);
```

Paramètres :
- cond : pointeur vers la variable condition qu'on veux initialiser ;
- attr : pointeur vers une variable attribut de la condition : en pratique on met NULL

Valeur de retour : 0 si l'initialisation fonctionne, code d'erreur sinon.

1.3 Se mettre en attente d'une condition

Pour attendre une condition, il faut utiliser un mutex associé à la condition. la primitive qui attend la condition est :

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

Paramètres :
- cond : pointeur vers la variable condition dont on attend le signal ;
- mutex : pointeur vers le mutex associé

Valeur de retour : 0 si la condition est signalée et si le thread sort correctement du wait, code d'erreur sinon.

1.4 Pour signaler une condition

Pour signaler la condition un thread utilise la primitive pthread_cond_signal(). Le prototype est :

```
int pthread_cond_signal(pthread_cond_t *cond);
```

Le fait de signaler la condition ne débloque qu'un thread qui est en wait sur cette condition. Paramètre :
- cond : pointeur vers la variable condition qu'on désire signaler ;

Valeur de retour : 0 en cas de succès code d'erreur sinon.

Remarque : on peut aussi utiliser la primitive pthread_cond_broadcast() qui débloque tous les threads en attente sur la condition.

Voici un exemple des parties critiques.

Thread qui attend la condition :

```
pthread_mutex_lock(&(ma_data.mutex));
pthread_cond_wait(&(ma_data.condition),
&(ma_data.mutex));
pthread_mutex_unlock(&(ma_data.mutex));
```

Thread qui signale la condition :

```
pthread_mutex_lock(&(pdata->mutex));
pthread_cond_signal(&(pdata->condition));
pthread_mutex_unlock(&(pdata->mutex));
```

2 La mémoire partagée

2.1 Problématique et principe

On a vu que lorsqu'un processus crée un processus fils avec fork(), les deux processus se retrouvent avec des données clonées. Il n'existe pas de zône mémoire de données commune permettant de partager des variables. Pour deux processus complètement séparés (issus de deux exécutables différents) c'est la même chose.

Il existe un mécanisme permettant de mettre en place un bloc mémoire appelé **segment de mémoire partagée** en abrégé **shm** (shared memory) que les processus vont pouvoir s'attacher pour s'échanger des données via ce bloc.

2.2 Création du segment de mémoire partagé

Un processus alloue un segment de mémoire partagée en utilisant **shmget** (« SHared Memory GET », obtention de mémoire partagée). Son premier paramètre est une clé entière qui indique le segment à créer. Des processus sans lien peuvent accéder au même segment partagé en spécifiant la même valeur de clé.

Voici le prototype de shmget() :

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t clef, size_t size, int shmflg);
```

Paramètres :

- clef : variable entière (mais de type key_t) permettant d'accéder à la zône de mémoire partagée (shm)
- size : taille en nombre d'octets de la shm
- shmflg : options. Combinaison entre les droits (comme pour la création d'un fichier) et IPC_CREAT qui permet de créer la shm si elle n'existe pas. Par contre si elle existe shmget renvoie l'id de celle qui existe sans en recréer une nouvelle ou écraser l'ancienne.

Valeur retournée : Un identifiant de segment **shmid** valide est renvoyé en cas de réussite, sinon -1 est renvoyé et errno contient le code d'erreur.

2.3 Création de la clef

On crée la clef à l'aide de la fonction ftok() dont le prototype est :

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(char *pathname, int proj_id);
```

Paramètres :

- pathname : nom de fichier qui doit exister ;
- proj_id : entier non nul qui sera utilisé (8 bits de poids faible pour créer la clef).

Valeur retournée : si la création de la clef réussit ftok retourne la clef créée, sinon -1 et errno contient un code d'erreur.

2.4 Commandes du shell en rapport aux IPCs

La commande **ipcs** permet d'afficher les ressources IPC du système. Elle permet de lister les shm, les sémaphores et et les boites de message :

```
MacHubert:tp7 hub$ ipcs
IPC status from <running system> as of Sat Dec 15 09:28:52 CET 2018
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:

T     ID     KEY        MODE       OWNER    GROUP
Shared Memory:
m 1310721 0x1304c0ec --rw-rw-rw-      hub    staff

T     ID     KEY        MODE       OWNER    GROUP
Semaphores:

MacHubert:tp7 hub$
```

FIGURE 1 – La commande ipcs.

La commande **ipcrm** avec l'option -m permet de supprimer une shm :

```
T     ID     KEY        MODE       OWNER    GROUP
Shared Memory:
m 1310721 0x1304c0ec --rw-rw-rw-      hub    staff

T     ID     KEY        MODE       OWNER    GROUP
Semaphores:

MacHubert:tp7 hub$ ipcrm -m 1310721
MacHubert:tp7 hub$
```

FIGURE 2 – La commande ipcrm.

2.5 S'attacher un segment de mémoire partagé

Après l'allocation du segment on doit s'attacher le segment avec la primitive shmat() dont voici le prototype :

```c
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Paramètres :
- shmid : identifiant de la shm (valeur renvoyée par shmget) ;
- shmaddr : Adresse à laquelle va être attachée la shm. En général on met NULL et shmat attache le segment à une adresse libre du sytème.
- shmflg : options. Parmi les options possibles SHM_RDONLY permet d'attacher le segment uniquement en lecture.

Valeur retournée : l'adresse (void*) du segment attaché ou (void*)-1 si ça échoue.

2.6 Se détacher du segment de mémoire partagé

Quand on n'utilise plus le segment, il est conseillé de se détacher du segment avec la fonction shmdt() dont voici le prototype :

```c
#include <sys/types.h>
#include <sys/shm.h>
int shmdt(const void *shmaddr);
```

Paramètres :
- shmaddr : Adresse d'attachement de la shm.

Valeur retournée : 0 si de détachement réussit, -1 sinon (et errno contient le code d'erreur)

2.7 Supprimer un segment de mémoire partagé. La primitive shmctl().

La primitive shmctl() permet d'effectuer diverses opérations de contrôle sur la shm, dont la suppression. Voici son prototype :

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

```
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Paramètres :
- shmid : identifiant de la shm.
- cmd : commande à réaliser :
  - IPC_RMID : permet de préparer la destruction du shm (il ne sera détruit que lorsque le dernier processus se détache du shm.
  - IPC_STAT : permet d'obtenir des informations sur le shm shmctl remplit la structure shmid_ds passée en paramètre par adresse.
- buf : adresse d'une structure de type shmid_ds pour l'opération IPC_STAT

Valeur retournée : elle dépend de l'opération réalisée. 0 pour une suppression de shm qui réussit.

2.8 Exemple complet

Le premier programme va créer la shm, se l'attacher, puis écrit un message dedans. Pour finir il va se détacher de la shm.

Le second programme va allouer la shm (qui existe déjà) se l'attache, affiche le message contenu et se détache. Pour finir il supprime la shm.

Premier programme (les fichiers entête sont omis) :

```
int main()
{
  char* pshm;
  key_t clef1=ftok(".",19);
  printf("clef=%d\n",clef1);
int shmid=shmget(clef1,1000,0666|IPC_CREAT);
if(shmid== -1)
  {
  printf("Allocation de la shm en echec!\n");
  printf("%s\n",strerror(errno));
  exit(0);
  }
printf("shmid=%x\n",shmid);
pshm=shmat(shmid,NULL,0);
if(pshm!=(void*)(-1))
  {
  strcpy(pshm,"Message stocké dans la shm!");
  shmdt(pshm);
  }
}
```

Voici le second :

```
int main()
{
  char* pshm;
  key_t clef1=ftok(".",19);
  printf("clef=%d\n",clef1);
int shmid=shmget(clef1,100,0666|IPC_CREAT);
if(shmid== -1)
  {
  printf("Allocation de la shm en echec!\n");
  exit(0);
  }
printf("shmid=%x\n",shmid);
pshm=shmat(shmid,NULL,0);
if(pshm!=(void*)(-1))
  {
  printf("Contenu=%s\n",pshm);
  shmdt(pshm);
  }
shmctl(shmid,IPC_RMID,NULL);
}
```

3 Le sémaphore de processus.

3.1 Pincipe

Lorsqu'on utilise des ressources communes entre processus (par exemple une shm), il est souvent nécessaire de s'assurer d'avoir l'accès exclusif à la ressource pour une section critique du code. Pour ce la on utilise le sémaphore à accès exclusif.

Un sémaphore S est une variable à valeurs entières positives ou nulle, manipulable par l'intermédiaire de deux opérations P (proberen) et V (verhogen) :
- P (S) : si S $\leq$ 0, alors mettre le processus en attente ; puis : S <= S - 1
- V (S) : S <= S + 1 et réveil d'un processus en attente.

Pour l'opération P(S) si la valeur est supérieure à 0 S est simplement décrémenté. Par contre si la valeur est inférieure ou égale à 0 le processus va bloquer (en attente de sémaphore) Si un autre processus fait son opération V(S) ce processus pourra être réveillé et faire sa décrémentation de la valeur.

Si une ressource doit être protégée contre des accès concurrents il faut utiliser un sémaphore avec comme valeur initiale 1. Ensuite l'utilisation de la ressource doit être encadrée par P(S) avant et V(S) après.

3.2 Création du sémaphore

Les IPC d'Unix donnent la possibilité d'agir sur plusieurs sémaphores de manière atomique (non-interruptible), un IPC de type sémaphore gérant par conséquence non pas un seul sémaphore, mais un tableau de sémaphore, chacun étant identifié par son rang dans le tableau (0 pour le premier). Pour créer, ou acquérir un tableau de sémaphores, on dispose de la primitive semget() dont le prototype est :

```
int semget(key_t clef, int nsems, int semflag);
```

Cette fonction crée le tableau de sémaphore si celui-ci n'existe pas, ou renvoie un identificateur sur ce dernier s'il a déjà été créé par un autre processus.

Paramètres :
- clef : iconstante symbolique pour identifier le tableau de sémaphore à sa création (comme pour la shm)
- nsems : le nombre de sémaphore(s) souhaité(s) dans le tableau
- semflag : options. Représente les droits souhaités, généralement : IPC_CREAT | 0666

Valeur retournée : l'identificateur (semid) du sémaphore. si l'appel réussit, -1 sinon (et errno contient le code d'erreur)

3.3 Contrôle du sémaphore

la fonction semctl() permet d'effectuer (entre autres) l'initialisation de la valeur d'un sémaphore et la suppression du sémaphore. Son prototype est :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl (int semid, int semno, int cmd, ...);
```

semctl() effectue l'opération de contrôle indiquée par cmd sur le jeu de sémaphores (ou sur le semno-ième sémaphore du jeu) identifié par semid. (Les sémaphores sont numérotés à partir de zéro.) Cette fonction prend trois ou quatre arguments, selon la valeur de cmd. Lorsqu'il y en a quatre, le quatrième est de type : union semun.

Paramètres :
- semid : identifiant du jeu de sémaphores
- semno : numéro du sémaphore dans le tableau de sémaphores.
- cmd : commande à réaliser :
  - IPC_SET permet de positionner la valeur du sémaphore numéro semno avec la valeur passée comme 4e paramètre.
  - IPC_RMID supprime immédiatement le jeu de sémaphores en réveillant tous les processus bloqués dans l'appel semop() sur le jeu. Ils obtiendront un code d'erreur, et errno aura la valeur EIDRM.
  - D'autres opérations existent (voir le man)

Valeur de retour : renvoie -1 s'il échoue et errno contient le code d'erreur.

Voici comment initialiser un sémaphore unique (d'indice 0) avec la valeur 1 :

```
semctl(monsem, 0, SETVAL, 1);
```

3.4 Opérations sur un sémaphore

La primitive semop() permet de faire les opérations courantes sur un sémaphore. Son prototype est :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

Paramètres :
- semid : identifiant du jeu de sémaphores
- sops : adresse d'une structure sembuf

Valeur de retour : 0 si l'opération réussit. Sinon -1 et code d'erreur dans errno sinon. Voici comment coder les opérations simples p() et v() sur le sémaphore (on admet qu'il n'y a qu'un sémaphore dans le jeu de sémaphores).

Opération P :

```
void p(int semid)
{
int rep;
struct sembuf sb={0,-1,0}; //num sema, valeur à ajouter = -1, pas d'option
rep=semop(semid, &sb, 1); //1 = un seul sémaphore concerné
}
```

Opération V :

```
void v(int semid)
{
int rep;
struct sembuf sb={0,1,0}; //num sema, valeur à ajouter = +1, pas d'option
rep=semop(semid, &sb, 1); //1 = un seul sémaphore concerné
}
```

3.5 Exemple

Voici un exemple complet.

Le premier programme crée le sémaphore et l'initialise à 1 pour qu'on ait affaire à un sémaphore à accès exclusif. Ensuite il rentre dans une boucle où il verrouille le sémaphore jusqu'à ce qu'on tape quelque-chose au clavier, puis il déverrouille le sémaphore jusqu'au prochain essai.

le 2e programme s'attache le sémaphore sans l'initialiser, puis il fait une boucle dans laquelle il verrouille, puis déverrouille le sémaphore. En exécutant les deux programmes dans deux terminaux séparés on voit bien le verrouillage !

Premier programme :

```
int main()
{
  char tampon[10];
  key_t clef1=ftok(".",20);
  int monsem=semget(clef1,1,IPC_CREAT|0666);
  semctl(monsem, 0, SETVAL, 1);
  while(1)
    {
    if(monsem== (-1))
      {
        printf("Erreur de semget\n");
        exit(0);
      }
```

```
    printf("Voulez-vous continuer (O/N)?");
    fgets(tampon,10,stdin);
    if(tampon[0]=='O')
      {
      printf("Verrouillage du sémaphore jusqu'à ce que vous tapiez quelque chose:");
      p(monsem);
      fgets(tampon,10,stdin);
      v(monsem);
      }
    else
      break;
  }
}
```

Deuxième programme :

```
int main()
{
int compteur=0;
char tampon[10];
key_t clef1=ftok(".",20);
int monsem=semget(clef1,1,IPC_CREAT|0666);
if(monsem== (-1))
  {
  printf("Erreur de semget\n");
  exit(0);
  }
while(1)
  {
  p(monsem);
  printf("Section critique\nPassage : %d\n",compteur++);
  v(monsem);
  sleep(1);
  }
}
```