Programmation Systéme

K.Oukfif

Sémaphore

- Un sémaphore est un compteur entier qui désigne le nombre d'autorisations d'accès à une section critique. Il a donc un nom et une valeur initiale
- Les sémaphores sont manipulés au moyen des opérations P(wait) et V (signal).
- L'opération P(S) décrémente la valeur du sémaphore S si cette dernière est supérieure à 0. Sinon le processus appelant est mis en attente. Le test du sémaphore, le changement de sa valeur et la mise en attente éventuelle sont effectués en une seule opération atomique indivisible.
- L'opération V(S) incrémente la valeur du sémaphore S, si aucun processus n'est bloqué par l'opération P(S). Sinon, l'un d'entre eux sera choisi et redeviendra prêt. V est aussi une opération indivisible.

Sémaphore

- sémaphores réalisent des exclusions mtuelles

semaphore mutex=1 ;		
processus P1 : P(mutex) section critique de P1 ; V(mutex) ;		processus P2 ; P(mutex) section critique de P2 ; V(mutex) ;

Sous-Norme Posix pour les sémaphores

- Librairie: <semaphore.h>
- Le type sémaphore: le type sem_t
- int sem_init(sem_t *sem, int pshared, unsigned int valeur)
- int sem_wait(sem_t *sem) // P
- int sem_post(sem_t *sem) // V

Producteur/consommateur

Trois sémaphores:

- Plein: nombre d'emplacements occupés, initialisé à 0.
- Vide: nombre d'emplacements libres, initialisé à N (taille du tampon)
- Mutex: exclusion mutuelle pour l'accès au tampon, initialisé à 1

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>
# define taille 3
sem_t plein, vide, mutex ;
int tampon [taille];
pthread_t prod,cons;

void* producer(void *);
void* consommer(void *);

int main (int argc , char *argv[])
{
```

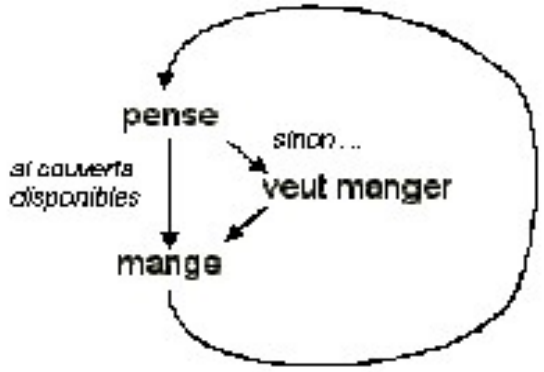
```
// initialiser le semaphore
sem_init(&plein , 0 , 0) ;
sem_init(&vide , 0 , taille) ;
sem_init(&mutex , 0 , 1) ;

// creation des threads
pthread_create(&prod,NULL,producer,NULL);
pthread_create(&cons,NULL,consommer,NULL);

// attendre la fin des threads

pthread_join(prod,NULL) ;
pthread_join(cons,NULL) ;
printf("fin des thread \n");
return 0;
}
```

```
void *producer (void *arg)  {
int ip=0, nbprod=0, objet=0;
do {
        sem_wait(&vide);
        sem_wait(&mutex);
        tampon[ip]=objet ;  //produire
        sem_post (&mutex ) ;
        sem_post (&plein) ;
printf( " ici producer : tampon[%d]= %d\n " ,ip ,objet) ;
objet ++;
nbprod ++;
ip =( ip +1)% taille ;
} while ( nbprod <= 5 ) ;
return ( NULL);
}
```

```
void *consommer (void *arg)  {
int ic=0, nbcons=0, objet;
do {
        sem_wait(&plein);    //p(plein)
        sem_wait(&mutex);
        objet = tampon[ic] ; //consommer
        sem_post (&mutex ) ;
        sem_post (&vide ) ;  //v(vide)
printf( " ici consommer : tampon[%d]= %d\n " ,ic ,objet) ;
 ic =(ic +1)% taille;
nbcons ++;
// sleep( 2 ) ;
} while ( nbcons <= 5 ) ;
return ( NULL);
}
```

Dîner des philosophes

- Combien de philosophes peuvent manger à un instant donné ?
- Quelque soit le nombre de philosophes, on ne peut jamais avoir deux philosophes mangeant cote à cote, pour de "conflit de fourchettes »
- Processus ←→Philosophe



Règles

- Un philosophe passe son temps à manger et à penser
- Pour manger, un philosophe a besoin de deux fourchettes : gauche et droite (qui sont de part et d'autre de son plat)
- Pour éviter l'interblocage, un philosophe ne prend jamais une seule fourchette, mais les deux à la fois
- fourchettes = objets partagés. L'accès et l'utilisation d'une fourchette doit se faire en exclusion mutuelle (sémaphore mutex)

Solution simple mais ….

```
Void philosophe()

{
      While(1)
      {
            Sleep();
            Prendre_fourchette_droite();
            Prendre_fourchette_gauche();
            Manger();
            Deposer_fourchette_droite();
            Deposer_fourchette_gauche();
      }
}
```

- Manger au même temps
- Prendre la fourchette gauche
- Bloqués sur la fourchette droite

Solution 1

Utiliser

- Un sémaphore par fourchette (dans notre cas, 5 fourchettes)
- Un mutex pour accéder aux fourchettes gauche et droite au même temps
- if (fourch[G] && fourch[i]) : si les deux fourchettes de gauche et de droite, ne sont pas libres, il faut libérer les fourchettes ainsi que mutex

```
#define N 5  // nombre de philosophe
#define G (( N+ i-1)%N) // philosophe de gauche de i
#define D (i) // philosophe de droite de i
#define libre 1
#define occupe 0

int fourch[N] = { libre, libre , libre , libre, libre } ;
sem_t mutex ;
void * philosophe ( void * ) ;
int main ( ) {
int NumPhi[N] = { 0 , 1 , 2 , 3 , 4 } ;  int i ; pthread_t ph[N] ;
sem_init(&mutex , 0 , 1 ) ;
for ( i=0 ; i<N ; i++) // creation des N philosophes
pthread_create(&ph[i], NULL, philosophe , & (NumPhi [i])) ;
i =0; // attendre la fin des threads
while ( i <N && ( pthread_join(ph [i++] ,NULL) == 0 )) ;
printf ( " fin des threads \n" ) ;
return 0 ;   }
```

```
void * philosophe ( void * num)  {
int i = * (int *) num ;
int  nb = 1 ;  // nombre de fois qu'un philosophe mange
while ( nb )  { // penser
//sleep ( 1 ) ;
// essayer de prendre les fourchettes pour manger
sem_wait(&mutex ) ;
if ( fourch[G] && fourch[i] ) {
fourch[G] = 0 ;  fourch[i] = 0 ;
printf( " philosophe [%d] mange \n" , i ) ;
sem_post (&mutex) ;
nb-- ;
// manger //sleep ( 1 ) ;
// liberer les fourchettes
sem_wait(&mutex ) ;
fourch [G] = 1 ;
fourch [i] = 1 ;
printf( " philosophe [%d ] a fini de manger\n " , i ) ;
sem_post (&mutex ) ;  }
else sem_post (&mutex ) ;  //ne  pas bloquer les autres
}   }
```

Problème de famine et d'équité

- Lors de l'exécution, c'est possible que certaines processus n'accèdent jamais aux fourchettes, donc ne mangent jamais.

→ C'est le problème de la famine

Solution 2

Pour éviter le problème de famine, il faut garantir que tout processus tentant d'entrer en section critique, il obtient satisfaction au bout d'un temps fini.

Utiliser

- Un sémaphore par philosophe, initialisé à 0
- Un tableau décrivant leurs états (penser, manger, faim)
- Un mutex pour tenter de manger

```
#define N 5  // nombre de philosophe
#define G (( N+ i-1)%N) // philosophe de gauche de i
#define D (i) // philosophe de droite de i
enum etat { penser , faim , manger } ;
enum etat Etat [N] = { penser , penser , penser , penser , penser } ;
sem_t mutex, S[N] ;
void * philosophe ( void * ) ;  void Test (int i ) ;
int main ( )  {
int NumPhi[N] = { 0 , 1 , 2 , 3 , 4 } ;  int i ;
pthread_t ph[N] ;
sem_init(&mutex , 0 , 1 ) ;
for ( i = 0 ; i <N ; i ++)     sem_init(&S[i] , 0 , 0 ) ;
for ( i=0 ; i<N ; i++)
pthread_create(&ph[i], NULL, philosophe , & (NumPhi [i])) ;
i =0;
while ( i <N && ( pthread_join(ph [i++] ,NULL) == 0 )) ;
printf ( " fin des threads \n" ) ;
return 0 ;
}
```

```
void * philosophe ( void * num)          {
int i = * (int *) num ;
int  nb = 1 ;  // nombre de fois qu'un philosophe mange
while ( nb )            { // penser
sleep ( 1 ) ;
// tenter de manger
sem_wait(&mutex ) ;
Etat[ i ]= faim ;
Test ( i ) ;
sem_post (&mutex ) ;
sem_wait(&S[ i ] ) ;
printf ( " philosophe [%d] mange\n " , i ) ;
sleep( 1 ) ; // manger
printf ( " philosophe [%d ] a fini de manger\n " , i ) ;
sem_wait(&mutex ) ;
Etat[i] = penser ;
// verifier si les vosins peuvent manger
Test(G ) ;  Test(D );
sem_post (&mutex ) ;            }            }
// procedure qui vrifie si le philosophe i peut manger
void Test ( int i )      {
if ( ( Etat [ i ] == faim ) && ( Etat [G] != manger )&& ( Etat[D] != manger ) ) {
Etat[i ] = manger ;
sem  post (&S [ i ] ) ;            }            }
```

Lecteurs/Redacteurs

Sémaphores:

- Mutex: exclusion mutuelle pour l'accès au tampon, initialisé à 1
- Redact : contrôle l'accès en écriture (en exclusion mutuelle) , initialisé à 1

Lecteur: Peut accéder si

- Il existe déjà d'autres lecteurs en cours de lecture (NbL > 0) (accès protégé par mutex)
- Aucun réducteur ni entrain d'écrire

Redacteur:

- Aucun réducteur ni entrain d'écrire (accès protégé par Redact)

```
#define NL 10 // definir le nombre de lecteur
#define NR 4   // definir le nombre de redacteur
int nbl=0,i;
sem_t mutex,redac;
void * lire(); void * ecrire();
int main()                    {
sem_init(&mutex,0,1); sem_init(&redac,0,1);
pthread_t thl[NL],thr[NR];
for(i=0;i<NL;i++)
pthread_create(&thl[i],NULL,lire,(void*) i);
for(i=0;i<NR;i++)
pthread_create(&thr[i],NULL,ecrire,(void*)i);
for(i=0;i<NL;i++) pthread_join(thl[i],NULL);
for(i=0;i<NR;i++) pthread_join(thr[i],NULL);
return 0;
}
```

```
void* lire(void* id)              {
int num =(int) id;
while(1)   {
sem_wait(&mutex);
if(nbl==0) {sem_wait(&redac); }
nbl++;
sem_post(&mutex);
sleep(1);
printf("je suis le lecteur Num=%d \n",num);
sem_wait(&mutex);
nbl--;
if(nbl==0) sem_post(&redac);
sem_post(&mutex);
sleep(1);  }
pthread_exit(NULL);              }
void *ecrire(void* id)           {
int num =(int) id;
while(1)                  {
sem_wait(&redac);
printf("je suis entrain d'ecrire mon num est  = %d\n ",num);
sleep(2);
printf("j'ai fini d'ecrire mon num est = %d\n ",num);
sem_post(&redac);
sleep(5);                 }
pthread  exit(NULL);             }
```