

### **TP N° 5 : Synchronisation des processus par des Moniteurs**

#### **Les objectifs spécifiques de TP système :**

Au terme de ce TP basé sur le Système Linux, l'étudiant doit être à la hauteur des compétences opérationnelles suivantes :

- ❖ maitrise La synchronisation des processus en utilisant les variables conditions,
- ❖ utiliser les deux appels système, `pthread_cond_wait()` et `pthread_cond_signal()` pour réalisé cette synchronisation.

#### **Synchronisation des threads par des mutex**

Tout comme les processus, les threads POSIX fonctionnent de manière asynchrone et peuvent donc être synchronisés à l'aide de mutex (semaphore binaire).

Les mutexs sont des sémaphores binaires (ne pouvant prendre que la valeur 0 ou 1) de la bibliothèque Pthreads. On les appelle également verrous.

Les threads partageant entre eux les variables globales et les descripteurs de fichiers du processus, une gestion de l'accès concurrentiel par mutex doit donc être introduite.

L'initialisation du mutex se fait avant la création des threads de la manière suivante:

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attributs);
```

Le premier paramètre (mutex) sera rempli à l'initialisation et sera utilisé comme identifiant pour ce mutex.

Le deuxième paramètre (attributs), permet de choisir un certain nombre d'attributs pour le mutex initialisé. Si les attributs par défaut doivent être utilisés, on placera cette valeur à NULL.

Lorsqu'un thread souhaite accéder à une section critique, il verrouille le mutex correspondant (opération de type down) de la manière suivante:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Si un autre thread se trouve dans la section critique, le thread appelant `pthread_mutex_lock` sera bloqué jusqu'à libération du mutex par la fonction suivante (opération de type up):

## ***TP N 5 : Synchronisation des processus par des Moniteurs***

---

**int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);**

Un « down non bloquant » est également possible ici à l'aide de la fonction `pthread_mutex_trylock` ce qui permet de tester l'accès à une section critique sans être bloqué si celle-ci est déjà utilisée par un autre thread.

Un mutex qui n'est plus utilisé doit être détruit de la manière suivante:

**int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);**

### **Résumé des mutexs**

La déclaration d'un mutex se fait de la façon suivante :

**pthread\_mutex\_t verrou;**

Pour initialiser un mutex, on procède comme suit :

**pthread\_mutex\_init (&verrou, NULL);**

Pour prendre un verrou (i.e. faire P sur le verrou), on fait :

**pthread\_mutex\_lock (&verrou);**

Et pour le libérer :

**pthread\_mutex\_unlock (&verrou);**

Par conséquent, une section critique pourra être faite de la façon suivante :

**pthread\_mutex\_lock (&verrou);**

**/\*section critique\*/**

**pthread\_mutex\_unlock(&verrou);**

### **Synchronisation par variable condition**

Synchronisation des threads par attente de condition outre les mutex, un autre moyen est mis à disposition pour synchroniser des threads: les variables de condition de type **pthread\_cond\_t**. De telles variables permettent de mettre en attente un ou plusieurs threads jusqu'à ce qu'un événement donné se produise.

L'initialisation de la condition se fait de la manière suivante:

**int pthread\_cond\_init(pthread\_cond\_t \* condition, pthread\_condattr\_t \* attributs);**

Le premier paramètre (condition) sera rempli à l'initialisation et sera utilisé comme identifiant pour la condition.

Le deuxième paramètre (attributs), permet de choisir un certain nombre d'attributs pour la condition initialisée. Si les attributs par défaut doivent être utilisés, on placera cette valeur à NULL.

## ***TP N 5 : Synchronisation des processus par des Moniteurs***

---

Lorsqu'un thread souhaite attendre une condition, il effectue cette série d'appel :

```
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_cond_wait(pthread_cond_t *condition, pthread_mutex_t *mutex);  
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

L'appel à **pthread\_cond\_wait** doit être protégé par un mutex. Ce même mutex doit être passé comme deuxième paramètre à la fonction **pthread\_cond\_wait** : ceci permet de déverrouiller le mutex et laisser le thread « notifiant » l'événement accéder à la variable de condition.

Lorsqu'un thread constate la réalisation de l'événement, une notification est alors envoyée à un ou à tous les threads en attente de l'événement:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_cond_signal(pthread_cond_t *condition); // Notification à 1 thread  
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_cond_broadcast(pthread_cond_t *condition); // Notification à tous les threads  
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Les threads qui étaient en attente de l'événement sont alors débloqués et peuvent poursuivre leur exécution.

Une variable de condition qui n'est plus utilisée doit être libérée par:

```
int pthread_cond_destroy(pthread_cond_t *condition);
```

### **Exercice N°1**

**Vous avez le programme suivant:**

#### **Exercice N°2**

```
#include<stdio.h>  
#include<stdlib.h>  
#include<pthread.h>  
#include<semaphore.h>
```

```
void *T1(void *arg) {  
    printf("TE");  
    printf(" T");  
    pthread_exit(NULL);  
}  
void *T2(void *arg) {  
    printf("S");
```

## ***TP N 5 : Synchronisation des processus par des Moniteurs***

---

```
printf("P");
pthread_exit(NULL);
}
void *T3(void *arg) {
printf("T");
printf(" N°2/n");

pthread_exit(NULL);
}
int main(){
pthread_t tid1, tid2;
pthread_create(&tid1,NULL,T1, NULL);
pthread_create(&tid2,NULL,T2, NULL);
pthread_create(&tid3,NULL,T3, NULL);

pthread_join(tid1,NULL);
pthread_join(tid2,NULL);
pthread_join(tid3,NULL);

exit(0);
}
```

**Q1** : complétez et synchronisez les 3 tache T1, T2 et T3 en utilisons moniteurs pour afficher le message suivante : **TEST TP N°2**

### **Exercice N°2**

#### **création de deux threads**

**On crée trois tâches ( threads ) pour exécuter chacune des trois fonctions : une affichera des étoiles '\*' des dièses '#'et arobase '@ .**

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
//Fonctions correspondant au corps d'un thread(tache)
void *etoile(void *inutilise);
void *diese(void *inutilise);

void *arobase(void *inutilise);

//Remarque:le prototype d'une tâche doit être :
void *(*start_routine)(void *)

int main(void)
{
pthread_t thrEtoile, thrDiese,thrarobase;
//les ID des de 3 thread
setbuf(stdout, NULL);
//pas de tampon sur stdout
printf("Je vais créer et lancer 3 threads");
pthread_create(&thrEtoile, NULL, etoile, NULL);
```

## ***TP N 5 : Synchronisation des processus par des Moniteurs***

---

```
pthread_create(&thrDiese, NULL, diese, NULL);

pthread_create(&thrarobase, NULL, arobase, NULL);

//printf("J'attends la fin des 3 threads\n");
pthread_join(thrEtoile, NULL);
pthread_join(thrDiese, NULL);
pthread_join(thrarobase, NULL);

printf("\nLes 3 threads se sont termines\n");
printf("Fin du thread principal\n");
pthread_exit(NULL);
return EXIT_SUCCESS;
}

void *etoile( void *inutilise)
{ int i;
  char c1 = '*';
  for(i=1;i<=200;i++)
  {
    write(1, &c1, 1);
    // écrit un caractère sur stdout(descripteur 1)
  }

  return NULL;
}

void *diese(void *inutilise)
int i;
char c1 = '#';
for(i=1;i<=200;i++)
{
  write(1, &c1, 1);
}
return NULL;
}

void *arobase(void *inutilise)
int i;
char c1 = '@';
for(i=1;i<=200;i++)
{
  write(1, &c1, 1);
}
return NULL;
}
```

**Q1 : compilez ce programme. Quel est le problème rencontré ?**

## ***TP N 5 : Synchronisation des processus par des Moniteurs***

---

**Q2 : synchronisez les trois threads de ce programme en utilisant les moniteurs pour avoir l'affichage suivant :**

-1) \*#\*#\*#\*#\*

-2)\*\*\*\*\*@ @ @\*\*\*\*\*@ @ @

3-)\*#\*#\*#\*#\*

### **Exercice N°3**

On a une piscine considérée comme une section critique entre les homes et les femmes. On suppose que la mixité est interdite au niveau de la piscine, c'est le premier qui arrive (femme ou home) peut accéder à la piscine si elle est libre.

Les personnes de sexe opposé devront attendre jusqu'à que la piscine sera libre.

On peut trouver plusieurs personnes de même sexe en même temps dans la piscine.

- Utilisez deux procédures entrer et sortie de la piscine pour chaque type de sexe (home ou femme).

Implémenter ce problème en utilisant les moniteurs entre 2 homes et 2 femmes selon les deux schémas suivantes :

#### **Schéma 1 :**

Entrer home 1

Entrer home 2

Sortie home 1

Sortie home 2

Entrer femme 1

Entrer femme 2

Sortie femme 1

Sortie femme 2

#### **Schéma 2 :**

Entrer home 1

Sortie home 1

Entrer femme 1

Sortie femme 1

Entrer home 2

Sortie home 2

Entrer femme 2

Sortie femme 2

## ***TP N 5 : Synchronisation des processus par des Moniteurs***

**Solution avec les moniteurs :**

### **Home :**

Si  $nf \neq 0$  alors wait (home)  
 $nh = nh + 1$   
Signal (home)

<< piscine sleep 4 s >>  
 $Nh = nh - 1$ ;  
Si  $nh = 0$  alors signal(femme)

### **femme:**

Si  $nh \neq 0$  alors wait (femme)  
 $nf = nf + 1$   
Signal (femme)

<< piscine sleep 4 s >>  
 $Nf = nf - 1$ ;  
Si  $nf = 0$  alors signal(g=home)

### **Exercice N°4**

**Vous le programme de producteur/consommateur**

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

/* définition du tampon */
#define N 10 /* Nb de cases du tampon */
#define NbMess 20 /* Nb de messages échangés */
int NbPleins=0;
int tete=1, queue=1;
int tampon[N];

/* définition des conditions et du mutex */
pthread_cond_t cons;
pthread_cond_t prod;
pthread_mutex_t mutex;
pthread_t tid[2];

void Deposer(int m){
    //xxxxxxxxxxxxxxxxxxxx
    if(NbPleins == N) //xxxxxxxxxxxx
        tampon[queue]=m;
    queue=(queue+1)%N;
    NbPleins++;
    //xxxxxxxxxxxxxxxxxxxx
    //xxxxxxxxxxxxxxxxxxxx
}

int Prelever(void){
    int m;
```

## TP N 5 : Synchronisation des processus par des Moniteurs

---

```

//xxxxxxxxxxxxxxxxxxxxxxx
if(NbPleins == 0) //xxxxxxx
m=tampon[tete];
tete=(tete+1)%N;
NbPleins--;
//xxxxxxxxxxxxxxxxxxxxxxx
//xxxxxxxxxxxxxxxxxxxxxxx
return m;
}

void * Producteur(void * k) /***** PRODUCTEUR
*/
{
    int i;
    int mess;
    for(i=1;i<=NbMess; i++){
        mess = i;
        Deposer(mess);
        printf("Mess depose:
%d\n",mess);
    }
}

void * Consommateur(void * k) /***** CONSOMMATEUR
*/
{
    int i;
    int mess;
    for(i=1;i<=NbMess; i++){
        mess=Prelever();
        printf("\tMess preleve:
%d\n",mess);
    }
}

void main() /* M A I N */
{
    int i, num;
    //xxxxxxxxxxxxxxxxxxxxxxx
    //xxxxxxxxxxxxxxxxxxxxxxx
    //xxxxxxxxxxxxxxxxxxxxxxx

    /* creation des threads */
    pthread_create(tid, 0, (void * (*)()) Producteur, NULL);
    pthread_create(tid, 0, (void * (*)()) Consommateur, NULL);

    // attente de la fin des threads
    pthread_join(tid[0],NULL);
}
```



## TP N 5 : Synchronisation des processus par des Moniteurs

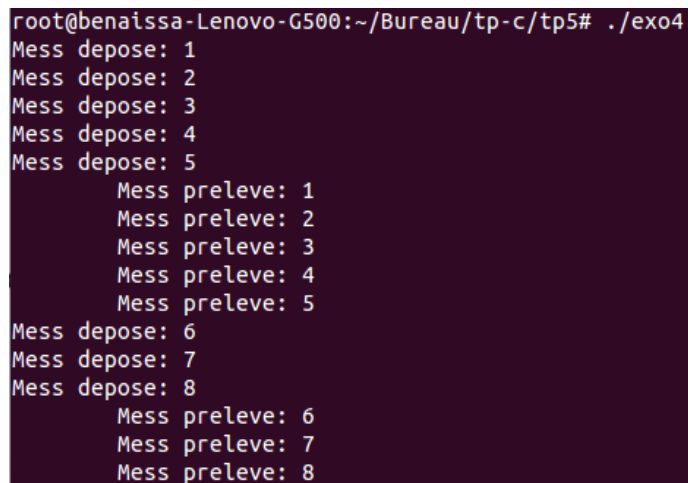
---

```
pthread_join(tid[1],NULL);

// libération des ressources
pthread_mutex_destroy(&mutex);
pthread_cond_destroy(&cons);
pthread_cond_destroy(&prod);
exit(0);
}
```

**Q1:** compléter ce programme en utilisant les moniteurs représenté par des variables conditions et des mutexs pour faire la synchronisation entre un thread producteur et un thread consommateur.

En doit obtenir le résultat suivant :



```
root@benaissa-Lenovo-G500:~/Bureau/tp-c/tp5# ./exo4
Mess depose: 1
Mess depose: 2
Mess depose: 3
Mess depose: 4
Mess depose: 5
    Mess preleve: 1
    Mess preleve: 2
    Mess preleve: 3
    Mess preleve: 4
    Mess preleve: 5
Mess depose: 6
Mess depose: 7
Mess depose: 8
    Mess preleve: 6
    Mess preleve: 7
    Mess preleve: 8
```

### Exercice N°5

**Vous le programme de lecteur/rédacteur**

```
#include <pthread.h>
#include <stdio.h>
#define N_REDACTEURS 5
#define N_LECTEURS 10
```

```
void debut_lecture(void);
void fin_lecture(void);
```

```
void debut_ecriture(void);
void fin_ecriture(void);
```

```
void fonction_lecture(void);
void fonction_ecriture(void);
```

```
int nbr_lecteur; //nombre de lecteur dans la section critique
(fichier)
```

## *TP N 5 : Synchronisation des processus par des Moniteurs*

---

```
int nbr_redacteur; //nombre de redacteur actif

pthread_cond_t red,lect;
pthread_mutex_t mutex;

int t = 0;

int main( void )
{
    int i, j;

    *****
    *****
    *****
    nbr_lecteur = 0;
        nbr_redacteur =0;

    pthread_t redacteurs[ N_REDACTEURS ];
    pthread_t lecteurs[ N_LECTEURS ];

    for( i = 0 ; i < N_REDACTEURS ; i++ )
    {
        pthread_create( &redacteurs[ i ], NULL,
(void*)&fonction_ecriture, NULL ); }

    for( j = 0 ; j < N_LECTEURS ; j++ )
    {
        pthread_create( &lecteurs[ j ], NULL,
(void*)&fonction_lecture, NULL );}
    for( i = N_REDACTEURS ; i > 0 ; i-- )
    {
        pthread_join( redacteurs[ i ], NULL ); }
    for( j = N_LECTEURS ; j > 0 ; j-- )
    {
        pthread_join( lecteurs[ j ], NULL );}
    return 0;
}

void debut_lecture( void )
{
    *****
    while( nbr_redacteur )
    {
        *****
    }

    nbr_lecteur++;
    printf( "\ndebut_lecture OK:  nbr-redacteur dans S-C =%2d ;
nbr-lecteur dans S-C=%2d \n", nbr_redacteur , nbr_lecteur );
    sleep(2);
}
```

## *TP N 5 : Synchronisation des processus par des Moniteurs*

---

```
*****
}
void fin_lecture( void )
{
    *****
    nbr_lecteur--;
    printf( "fin_lecture OK:    nbr-redacteur dans
S-C=%2d ; nbr-lecteur dans S-C=%2d \n", nbr_redacteur,
nbr_lecteur);
    sleep(2);
    if (nbr_lecteur ==0) *****

    *****
}
void debut_ecriture( void )
{
    *****
    while( nbr_redacteur || nbr_lecteur )
    {
        *****
    }
    nbr_redacteur = 1;
    printf( "\ndebut_ecriture OK: nbr-redacteur dans S-C=%2d ;
nbr-lecteur dans S-C=%2d \n", nbr_redacteur, nbr_lecteur );
    sleep(2);
    *****
}
void fin_ecriture( void )
{
    *****
    nbr_redacteur =0;
    *****
    *****
    printf( "fin_ecriture OK:    nbr-redacteur dans S-C=%2d ;
nbr-lecteur dans S-C=%2d \n", nbr_redacteur, nbr_lecteur );
    sleep(2);
    *****
}
void fonction_lecture( void )
{
    debut_lecture();
    printf( "  > Je lis %d\n", t );
    fin_lecture();
}
void fonction_ecriture( void )
{
    debut_ecriture();
    t++;
    printf( "  > J'ecris %d\n", t );
    fin_ecriture();
}
```

## ***TP N 5 : Synchronisation des processus par des Moniteurs***

---

**Q1:** compléter ce programme en utilisant les moniteurs représenté par des variables conditions et des mutexs pour faire la synchronisation entre un thread lecteur et un thread rédacteur.