Compléments des notes sur les outils de communication inter-processus

Master 1^{iere} année, Bruno Jacob

Novembre 2010

1 Introduction

Les programmes utilisant des sémaphores sont en général difficiles à mettre en oeuvre et à "débugguer" car les erreurs sont dues à des conditions de concurrence, des interblocages ou d'autres formes de comportements imprévisibles et/ou impossible à reproduire. Par ailleurs, les sémaphores sont des objets globaux devant être connus de tous les participants.
Reprenons l'exemple des producteurs/consommateurs avec des sémaphores :

```
int N ; /* Nb elements dans F */
File F[N] ;
semaphore Plein = N ;
          Vide  = 0 ;
          mutex = 1 ;
```

```
producteur()
{
  int e ;
  TQ vrai FRE
     e=produire();
     P(Plein);
     P(mutex);
     Enfiler(F,e) ;
     V(mutex);
     V(Vide);
  FTQ
}
```

```
consommateur()
{
  int e ;
  TQ vrai FRE
     P(Vide);
     P(mutex);
     Defiler(F,e);
     V(mutex);
     V(Plein);
     consommer(e);
  FTQ
}
```

C'est une programmation "de bas niveau" qui peut aboutir à un interblocage si l'on ne fait pas attention à l'ordre des primitives P et V.

Pour faciliter le développement de ces programmes, l'idée de proposer des mécanismes de plus haut niveau offrant de manière transparente des mécanismes d'exclusion mutuelle et de synchronisation a émergé. Hoare et Brinch Hansen ont mis au point en 1973 une primitive de synchronisation de haut niveau appelée *moniteur*.

Un moniteur s'inspire de mécanismes proposés pour la gestion de classes dans le langage Simula. Ce langage, dont la première version a été conçue en 1967, est une extension du langage Algol 60 et constitue de fait le premier langage ayant introduit les principaux concepts de la programmation objet.

2 Les moniteurs

2.1 Présentation

Un moniteur est une collection de procédures, de variables et de structures de données qui sont regroupées dans un module spécial. Les processus ne peuvent pas accéder directement aux structures de données internes du moniteur, il faut pour cela qu'ils utilisent les procédures déclarées à l'intérieur du moniteur.
Les moniteurs ont une propriété importante qui les rend intéressants pour faire de l'exclusion mutuelle : à un instant donné, un seul processus peut être actif dans le moniteur, c'est à dire que les procédures du moniteur ne peuvent être exécutées que par un seul processus à la fois.
Quand un processus P appelle une procédure du moniteur M, on dit qu'il cherche à entrer dans M. Inversement, quand P a fini d'exécuter la procédure on dit qu'il sort de M. Si un processus $P1$ cherche à entrer dans M alors qu'un autre processus $P2$ est déjà actif dans M, dans ce cas $P1$ est suspendu jusqu'à ce que $P2$ sorte du moniteur.
Les moniteurs sont une construction du langage de programmation. C'est donc le compilateur qui implémente l'exclusion mutuelle sur les entrées des processus dans le moniteur. Partant du principe que c'est le compilateur (et non le programmeur) qui prend en charge l'exclusion mutuelle, il y a moins de chance pour que les exécutions des processus se passent mal. Le compilateur traduira toutes les sections critiques en procédures de moniteur et ainsi deux processus ne pourront exécuter leurs sections critiques en même temps.

Dans notre exemple du Produteur/Consommateur un moniteur pourrait être :

```
int N ; /* Nb element dans F */
File F[N] ;

moniteur prodcons ;
{
  producteur() {.....}
  consommateur() {.....}
}
```

Ainsi, 2 processus ne pourront pas appeler en même temps les procédures `producteur` et `consommateur` ce qui revient à effectuer une exclusion mutuelle sur la `File F`;
Mais il faut de plus que les moniteurs ne se bloquent que quand ils ne peuvent pas poursuivre l'exécution de la procédure appelée.
Dans notre exemple, il faut :

- qu'un processus producteur soit bloqué quand la file est pleine
- qu'un processus consommateur soit bloqué quand la file est vide

Les codes des procédures seraient les suivants :

```
int N ;
File F[N] ;
moniteur prodcons
{
  producteur()
  {
    int e ;
    TQ vrai FRE
       e=produire();
       /* boucler en attendant que F ne soit plus pleine */
       TQ FilePleine(F) FRE rien FTQ
       Enfiler(F,e) ;
    FTQ
  }
  consommateur()
  {
    int e ;
    TQ vrai FRE
       /* boucler en attendant que F ne soit plus vide */
       TQ FileVide(F) FRE rien FTQ
       Defiler(F,e) ;
       consommer(e);
    FTQ
  }
}
```

Une telle solution n'est évidemment pas satisfaisante : un processus dans le moniteur attend (de manière active) qu'une condition soit réalisée sur la file `F`. Cette condition n'arrivera jamais. En effet aucun autre processus ne pourra entrer dans le moniteur du fait de cette attente active :

- un producteur qui boucle sur `FilePleine(F)` ne sera pas débloqué par un consommateur puisque celui ci ne pourra jamais entrer dans le moniteur
- un consommateur qui boucle sur `FileVide(F)` ne sera pas débloqué par un producteur pour la même raison

Les moniteurs doivent donc permettre à un processus qui s'y exécute de le relâcher pour permettre à un autre d'y pénétrer et, inversement, il doit être possible de réveiller le cas échéant un processus ayant relâché un moniteur.

Ceci est réalisé par des variables conditionnelles sur lesquelles on peut faire deux opérations habituellement appelées `wait` et `signal`. Si C est une variable conditionnelle alors on effectue

- un `wait` sur C, pour faire un relâchement du processus, quand une procédure du moniteur ne peut poursuivre. Le processus appelant est suspendu. Il sort donc du moniteur et permet ainsi à un autre d'y entrer.
- un `signal` sur C quand on veut réveiller les processus suspendus par un `wait` sur C

Afin d'éviter que tous les processus réveillés ne se retrouvent en même temps dans le moniteur, différentes règles on été établies pour définir se qui se passe à l'issue d'un signal.

- Règle de Hoare : ne laisser entrer dans le moniteur que le processus qui à été suspendu le moins longtemps
- Règle de Brinch Hansen : exiger du processus qui a fait `signal` de sortir immédiatement du moniteur. Il laisse ainsi la place à tous ceux qui étaient en attente. L'ordonnanceur en choisira un parmi ceux ci.
- 3^{ieme} règle : `signal` ne réveille qu'un seul processus choisi par l'ordonnanceur et laisse le processus qui a appelé `signal` se terminer

Si un `signal` est réalisé sur une variable conditionnelle et qu'aucun processus ne l'attend, ce signal est perdu.

```
int N ;
File F[N] ;
moniteur prodcons
{
  int cpt ;
  condition Pleine , Vide ;

  producteur()
```

```
  {
    int e ;
    TQ vrai FRE
       e=produire();
       SI( cpt == N ) ALORS wait(Pleine) FSI
       Enfiler(F,e) ;
       cpt = cpt + 1 ;
       SI cpt == 1 ALORS signal(Vide) FSI
    FTQ
  }
  consommateur()
  {
    int e ;
    TQ vrai FRE
       SI( cpt == 0 ) ALORS wait(Vide) FSI
       Defiler(F,e) ;
       cpt = cpt - 1  ;
       SI( cpt == N-1 ) ALORS signal(Pleine) FSI
       consommer(e);
    FTQ
  }
}
```

Pour permettre à deux processus de produire et de consommer en même temps :

```
int N ;
File F[N] ;
moniteur prodcons
{
  int cpt ;
  condition Pleine , Vide ;
  int tete = 0 ;
  int queue = 0 ;

  Enfiler(File F, int e)
  {
    SI( cpt == N ) ALORS wait(Pleine) FSI
    File[tete] = e ; tete = (tete+1)%N ;
    cpt = cpt + 1 ;
    SI cpt == 1 ALORS signal(Vide) FSI
  }
  Defiler(File F, int e)
  {
    int i =
    SI( cpt == 0 ) ALORS wait(Vide) FSI
    e = File[queue] ; queue= (queue+1)%N ;
    cpt = cpt - 1  ;
    SI( cpt == N-1 ) ALORS signal(Pleine) FSI
  }
```

```
}
producteur()
{
  int e ;
  TQ vrai FRE
      e=produire();
      Enfiler(F,e);
  FTQ
}
consommateur()
{
  int e ;
  TQ vrai FRE
      Defiler(F,e);
      consommer(e);
  FTQ
}
```

2.2 Réalisation Java

Java est un langage orienté objet qui prend en charge les threads au niveau utilisateur et qui autorise le regroupement de méthodes (procédures) en classes. En ajoutant le mot clé `synchronized` à une déclaration de méthode, Java garantit que, une fois qu'un thread a commencé à exécuter cette méthode, aucun autre thread ne pourra exécuter en même temps une autre méthode `synchronized` de cette classe.

L'implémentation d'un moniteur en Java est différente de celle d'un moniteur classique car Java ne dispose pas de variables conditionnelles. Java les remplace par deux procédures `wait` et `notify` qui sont les équivalents de `wait` et `signal`. Mais ces procédures peuvent être interrompues. On doit donc récupérer les exceptions Java correspondant aux interruptions pour gérer le `wait` tel qu'il est défini dans la présentation des moniteurs.

Exemple du producteur/Consommateur en Java :

```
import java.io.* ;
import java.lang.* ;

public class ProdCons
{
    // Nb emplacements dans F
     static final int N = 100 ;
    // instanciation d'un thread producteur
```

```
 static producteur p = new producteur() ;
// instanciation d'un thread consommateur
 static consommateur c = new consommateur() ;
// instanciation d'un nouveau moniteur
 static moniteur_file m = new moniteur_file() ;

 public static void main( String argv[] )
 {
       p.start() ; // demarre le thread produteur
       c.start() ; // demarre le thread consommateur
 }

 static class producteur extends Thread
 {
      public void run()
      {
           int e ;
           while(true)
           {
                e = produire() ;
                m.enfiler(e);
           }
      }
      private int produire()
      {
           return Math.round((int)(100 * Math.random())) ;
      }
 }

 static class consommateur extends Thread
 {
      public void run()
      {
           int e ;
           while(true)
           {

                e = m.defiler();
                consommer(e) ;
           }
      }
      private void consommer( int e )
      {
           System.out.println("Element = "+ Integer.toString(e) ) ;
      }
 }

 static class moniteur_file
 {
```

```
        private int file[] = new int[N] ;
        private int cpt  = 0 ;
        private int tete = 0 ;
        private int queue = 0 ;

        public synchronized void enfiler( int e )
        {
            if( cpt == N ) dormir() ;
            file[tete] = e ;
            tete = (tete+1)%N ;
            cpt = cpt + 1 ;
            if( cpt == 1 ) notify() ;
        }
        public synchronized int defiler()
        {
            int e ;

            if( cpt == 0 ) dormir() ;
            e = file[queue] ;
            queue = (queue+1)%N ;
            cpt = cpt - 1 ;
            if( cpt == N-1 ) notify() ;
            return e ;
        }

        private void dormir()
        {
            try{ wait() ; }
            catch( InterruptedException exc) {};
        }
    }
}
```

2.3 Réalisation C/POSIX

L'utilisation combinée de variables conditionnelles et de mutexes dans Posix fournit une interface correspondant au concept de moniteur. En voici les grandes lignes :

2.3.1 Procédures du moniteur

Pour garantir qu'une seule procédure du moniteur soit activée à un moment donné, il suffit de protéger l'exécution de toutes les procédures par le même sémaphore d'exclusion mutuelle.

```
static int enfiler(...)
{
```

```
    pthread_mutex_lock(&mutex);
    ...
    pthread_mutex_unlock(&mutex);
  }
  static int defiler(...)
  {
    pthread_mutex_lock(&mutex);
    ...
    pthread_mutex_unlock(&mutex);
  }
```

2.3.2 Variables conditionnelles

Une variable conditionnelle *var* de moniteur est l'association
— d'un sémaphore *var_mutex* de type `pthread_mutex_t`
— d'une variable de condition *var_cond* de type `pthread_cond_t`
var_mutex est utilisé pour assurer la protection des opérations sur la variable *var* et *var_cond* permet d'en transmettre les changements d'états.
L'initialisation d'une variable conditionnelle est réalisée par :

```
int pthread_cond_init( pthread_cond_t *cond,
                       pthread_cond_attr_t *attr)
```

qui initialise la variable conditionnelle `cond` : la zone correspondante doit avoir été allouée au préalable. La zone pointée par `attr` doit par ailleurs avoir été initialisée ou attr peut être NULL, auquel cas les attributs par défaut sont appliqués à la variable conditionnelle.
Une variable conditionnelle peut également être initialisée de manière statique lors de sa définition sous la forme :

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Les attributs d'une variable conditionnelle (la zone pointée par `attr` étant supposée allouée) peuvent etre initialisés à un ensemble de valeurs par défaut par

```
int pthread_condattr_init(pthread_condattr_t *attr)
```

Les fonctions suivantes rendent respectivement les attributs de variable conditionnelle et une variable conditionnelle inutilisables (la zone pointée doit être réinitialisée).

```
int pthread_cond_destroy(pthread_cond_t *cond)
int pthread_condattr_destroy(pthread_condattr_t *attr)
```

La primitive `wait` d'un moniteur est réalisée par :

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

les effets d'un appel à cette fonction sont :

- le mutex `mutex` est libéré
- l'activité est mise en attente sur la condition `cond`
- lorsque la condition est signalée par une autre activité, `mutex` est acquis de nouveau par l'activité et celle ci reprend son exécution. Etant donné que plusieurs activités peuvent modifier `cond`, au retour de la fonction, il faut revérifier que `cond` est toujours vraie.

La primitive `signal` d'un moniteur, permettant de signaler un évènement, est réalisée par :

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

Permet le réveil d'une seule activité attendant `cond`. Si aucune activité n'attend `cond` (c'est à dire si aucune thread n'a fait un `pthread_cond_wait`) alors la signalisation de cet évènement est perdue.

On peut réveiller toutes les activités attendant `cond` par un appel à la fonction

```
int pthread_cond_broadcast(pthread_cond_t *cond)
```

Il est ainsi possible d'implanter des barrières de synchronisation (voir §3).

Un code type d'utilisation des variables conditionnelles contiendrait les variables externes suivantes :

```
pthread_con_t var_cond ;
pthread_mutex_t var_mutex ;
type_t var ;
```

Du coté du processus qui attend la condition :

```
pthread_mutex_lock(&var_mutex);
while(! condition(var) ) pthread_cond_wait(&var_cond,&var_mutex);
pthread_mutex_unlock(&var_mutex) ;
```

Du coté du processus qui change l'état de la condition :

```
pthread_mutex_lock(&var_mutex);
var = nouvelle_valeur ;
if( condition(var) ) pthread_cond_signal(&var_cond);
pthread_mutex_unlock(&var_mutex) ;
```

2.3.3 Exemple en C/POSIX

Voici un exemple de réalisation d'un moniteur du producteur/Consommateur en C/POSIX :

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

#include <commun.h>

#define N 5         /* Nb emplacements dans F */
#define PROD  2     /* nombre de producteurs */
#define CONS  2     /* nombre de consommateurs */

/*---------- Fonctions locales ----------*/

static int produire()
{
     return((int)random()%100);
}

static void consommer( int e )
{
     printf("Element = %d\n" , e ) ;
}

/*---------- MONITEUR ----------*/

pthread_mutex_t mutex_moniteur ; /* le mutex controlant l'ensemble */
pthread_cond_t  cond_Vide      ; /* condition correspondant a la non vacuite */
pthread_cond_t  cond_Pleine    ; /* condition associee a la non-plenitude */

int file[N];
int cpt   = 0;
int tete  = 0;
int queue = 0;

extern void m_enfiler( int e )
{
     /* Entree dans le moniteur */
     pthread_mutex_lock(&mutex_moniteur);

     while( cpt == N ) pthread_cond_wait( &cond_Pleine , &mutex_moniteur );
     file[tete] = e ;
     tete = (tete+1)%N ;
```

```
        cpt = cpt + 1 ;
        if( cpt == 1 ) pthread_cond_signal(&cond_Vide) ;

        /* Sortie du moniteur */
        pthread_mutex_unlock(&mutex_moniteur);
}

extern int m_defiler()
{
        int e ;

        /* Entree dans lemoniteur */
        pthread_mutex_lock(&mutex_moniteur);

        while( cpt == 0 )pthread_cond_wait(&cond_Vide, &mutex_moniteur) ;
        e = file[queue] ;
        queue = (queue+1)%N ;
        cpt = cpt - 1 ;
        if( cpt == N-1 )  pthread_cond_signal(&cond_Pleine);

        /* Sortie du moniteur */
        pthread_mutex_unlock(&mutex_moniteur);
        return e ;
}

/*---------- THREADS ----------*/
/*
 * Fonction executee par les threads productrices
 */

static void producteur()
{
        int e ;

        while(TRUE)
        {
                e = produire() ;
                m_enfiler(e);
        }
}

/*
 * Fonction executee par les threads consommatrices
 */
static void consommateur()
{
        int e ;

        while(TRUE)
```

```
        {
                e = m_defiler() ;
                consommer(e);
        }
}

/*
 * Thread principale
 */

int
main(  int nb_arg , char * tab_arg[] )
{
  int i;
  pthread_t tid_prod[PROD]; /* identite des threads productrices */
  pthread_t tid_cons[CONS]; /* identite des threads consommatrices */
  /*------------*/

  srandom((unsigned int)getpid()) ;

  /* Initialisations du moniteur */
  /* --- du mutex */
  pthread_mutex_init(&mutex_moniteur, NULL);
  /* --- des variables de condition */
  pthread_cond_init(&cond_Vide, NULL);
  pthread_cond_init(&cond_Pleine, NULL);

  /* Creation des threads productrices */
  for(i = 0; i < PROD; i++)
        pthread_create(&tid_prod[i], NULL, (void *)producteur, (void *)NULL);
  /* creation des threads consommatrices */
  for(i = 0; i < CONS; i++)
        pthread_create(&tid_cons[i], NULL, (void *)consommateur, (void *)NULL);
  /* mettre toutes les threads en concurrence */
  pthread_setconcurrency(PROD+CONS);
  pause();
  exit(0);
}
```

3 Les barrières

Le mécanisme des barrières est utilisé quand les applications font intervenir des groupes de processus et qu'elles peuvent être décomposées en phases. Les barrières ont pour règle qu'aucun processus ne peut entrer dans la phase suivante avant que tous les processus ne soient prêts à y entrer. On place ainsi une barrière à la fin de chaque phase : lorsqu'un processus atteint une barrière, il est bloqué jusqu'à ce que tous les autres l'atteignent également. La FIG. 1 illustre ce fonctionnement.

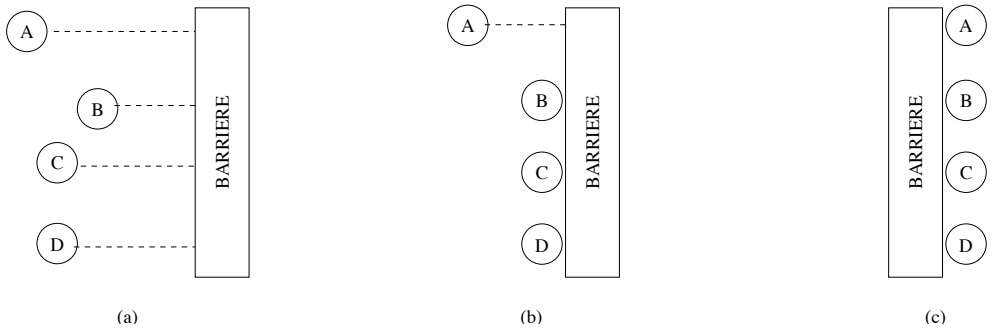

FIGURE 1 – Fonctionnement d'une barrière (a) Processus approchant de la barrière (b) On attend le dernier (c) Une fois le dernier arrivé, tous les processus peuvent passer