# Compléments des notes sur les outils de communication inter-processus

*Master 1$^{iere}$ année, Bruno Jacob*

Novembre 2005

## 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 comportement imprévisibles et/ou impossibles 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 semaphores :

```
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 à émergée. 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 peuvennt 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 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 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 : laisser le processus qui a fait `signal` se terminer puis réveiller tous ceux qui étaient en attente.

Nous retiendrons par la suite la règle de Brinch Hansen.

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
{
     static final int N = 100 ; // Nb emplacements dans F
     static producteur p = new producteur() ;// instancie un thread producteur
     static consommateur c = new consommateur() ; // instancie un thread consommateur
     static moniteur_file m = new moniteur_file() ; // instancie un nouveau moniteur

     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 meme 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)
```

Les primitives `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 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 §4).

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);
if(! 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 ;
pthread_mutex_unlock(&var_mutex) ;
if( condition(var) ) pthread_cond_signal(&var_cond);
```

### 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>

#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 lemoniteur */
     pthread_mutex_lock(&mutex_moniteur);

     if( 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);

     if( 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(B_TRUE)
      {
            e = produire() ;
            m_enfiler(e);
      }
}

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

      while(B_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 L'échange de messages

Les moniteurs, comme les sémaphores, sont des outils pour résoudre les problèmes d'accès à des sections critiques pour des processus ayant accès à une mémoire commune. Lorsque l'on passe sur un système distribué composé de plusieurs processeurs connectés en réseau, chacun disposant de sa propre mémoire, ces outils ne sont plus adaptés.

Une solution est l'échange de messages (*message passing*). Cette méthode de communication interprocessus emploie deux primitives `send` et `receive`. Ce sont des primitives système plutôt que des constructions du langage (donc pareilles aux sémaphores mais différentes des moniteurs).

On peut les voir comme les primitives `sendto(...,message,..., destination,...)` et `recvfrom(...,message,..., source,...)` de la communication entre processus distants par datagramme avec les sockets Unix ( voir cours n$^o$2 sur la Communication par sockets).

## 3.1 Inconvénients

Les systèmes d'échanges de messages ont le même problème que la gestion des pertes de paquets en datagramme, que l'on peut résoudre par un dispositif d'accusé de réception (*l'acknowledgment*) et de limite de temps d'attente de message (*le timeout*)(voir également le cours n$^o$2).

L'échange de message ne garantie pas que les messages arrivent au bon destinataire et non à un usurpateur.
S'ils offrent une solution pour des processus distants, leur utilisation pour des processus locaux est moins performante que les précédentes solutions car il est toujours plus coûteux de copier les messages d'un processus dans un autre plutot que d'appeler un sémaphore ou que d'entrer dans un moniteur.

## 3.2 Le problème du Producteur/Consommateur

Si le buffer en mémoire partagée contient $N$ emplacements, alors on créer $N$ messages. Le consommateur commence par émettre $N$ messages vides au producteur. Chaque fois que le producteur veut remplir un emplacement dans le buffer,

1. il prend un message vide
2. il renvoie un message plein au consommateur.

Ainsi, si le buffer est plein, le producteur est en attente de message vide.
Chaque fois que le consommateur veut extraire un emplacement du buffer,

1. il prend un message plein
2. il renvoie un message vide au producteur.

Ainsi, si le buffer est vide, le consommateur est en attente de message plein.
Il existe de nombreuses variantes d'échanges de messages. Parmis celles ci on peut citer :

- si les processus distants sont identifiés de manière unique, alors on envoie directement le message à un processus (`sendto` et `recvfrom` en datagramme)
- sinon on peut créer une structure "boîte aux lettres" dans laquelle les messages sont déposés. Les messages sont alors à destination d'une BAL et non d'une identification de processus (similaire aux files de messages Unix mais sur des machines distantes). Dans ce cas, le producteur et le consommateur créent chacun leur boîte aux lettres qui contiennent $N$ messages chacunes.
  Le producteur
  - dépose ses messages dans la BAL du consommateur
  - il sera en attente quand il voudra déposer un message et que la BAL consommateur sera pleine (tous les emplacements n'ont pas été extraits)

  Le consommateur
  - dépose ses messages dans la BAL du producteur
  - il sera en attente quand il voudra déposer un message et que la BAL producteur sera pleine (tous les emplacements n'ont pas été remplis)

L'échange de messages est souvent exploité dans les systèmes de programmation parallèle. Le MPI (*Message-Passing Interface*) est une réalisation très employée en informatique scientifique.

## 3.3 Exemple de réalisation en UDP

Le système Unix identifie les processus de manière unique. On peut donc envoyer un message directement à un processus en utilisant, par exemple, la communication par socket en datagrammes

### 3.3.1 Fonctions d'échange de messages

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <netinet/in.h>
#include <pwd.h>
#include <unistd.h>
#include <netdb.h>
#include <strings.h>
#include <signal.h>

#include <echange_messages.h>

extern
int send_message( const type_message_t type ,
                  const int sock_expediteur,
                  struct sockaddr_in adr_destinataire )
{
     message_t message ;

     /*----------*/


     /* Construction  du message a envoyer */
     message.type = type ;

     /* Envoi d'un paquet */
     sendto( sock_expediteur, &message , sizeof(message), 0,
             (struct sockaddr *)&adr_destinataire ,
             (int)sizeof(adr_destinataire));

     return(0);
}
```

```
extern
int receive_message( const int sock_destinataire ,
                     struct sockaddr_in * adr_expediteur )
{
     message_t  message;
     int lg_adr_expediteur ;

     /*----------*/

     /* Reception du message */
     lg_adr_expediteur = sizeof((*adr_expediteur)) ;
     recvfrom( sock_destinataire , (void *)&message , sizeof(message), 0,
               (struct sockaddr *)adr_expediteur ,
               (int *)&lg_adr_expediteur );

     return(0);
}
```

### 3.3.2 Producteur

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <netinet/in.h>
#include <unistd.h>
#include <netdb.h>
#include <strings.h>

#include <echange_messages.h>
#include <file_partagee.h>

int
main( int argc , char * argv[] )
{
     /* Variables Consommateur */
     struct sockaddr_in adr_cons ;

     /* Variables Producteur */
     int port_prod ;
     char nom_machine_prod[LONGUEUR_NOM_MACHINE];
     struct hostent  *infos_machine_prod ;

     int sock_prod ;
     struct sockaddr_in adr_prod;

     /* Variables de travail */
     int cpt ;
```

```
char msgerr[256];

/*-----*/

if( argc != 2 )
{
  fprintf( stderr, "Usage : %s <port producteur>\n", argv[0] );
  exit(-1);
}

if( gethostname( nom_machine_prod , LONGUEUR_NOM_MACHINE ) == -1 )
{
     perror("Pb gethostname");
     exit(-1);
}

sscanf( argv[1] , "%d" , &port_prod ) ;

/* Creations points d'entrees */
if( (sock_prod = socket(AF_INET, SOCK_DGRAM, 0 )) == -1 )
{
     perror("Pb socket consommateur");
     exit(-1);
}

/* Creations des adresses */
 /* -- producteur */
 adr_prod.sin_port = port_prod; ;
 adr_prod.sin_family = AF_INET ;
 if( (infos_machine_prod = gethostbyname(nom_machine_prod)) == NULL )
 {
      sprintf( msgerr , "Pb sur le nom machine %s", nom_machine_prod);
      perror(msgerr);
      exit(-2);
 }
 bcopy( infos_machine_prod->h_addr, &adr_prod.sin_addr, infos_machine_prod->h_leng
 bzero( adr_prod.sin_zero , sizeof(adr_prod.sin_zero) ) ;

 /* -- consommateur : inutile car c'est lui qui initie la communication */


 /* Attachements des points d'entrees aux adresses */
 if( bind(sock_prod,(struct  sockaddr *)&adr_prod,sizeof(adr_prod)) == -1 )
 {
      perror("Pb bind sur socket consommateur");
      exit(-3);
 }

 /* initialisation des pseudo-aleatoires */
```

```
        srandom((unsigned int)getpid());


        /* Consommation des elements */
        cpt = 0 ;
        while(B_TRUE)
        {
                /* Extraction message vide */
                receive_message( sock_prod ,
                                 &adr_cons );

                /* Production element */
                cpt++;
                printf("Production element %d\n" , cpt );

                /* Envoi message plein */
                send_message( PLEIN,
                              sock_prod,
                              adr_cons );

                /* Attente */
                sleep( random() % 3 ) ;
        }
}
```

### 3.3.3 Consommateur

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <netinet/in.h>
#include <unistd.h>
#include <netdb.h>
#include <strings.h>

#include <file_partagee.h>
#include <echange_messages.h>

int
main( int argc , char * argv[] )
{
        /* Variables Producteur */
        int port_prod ;
        struct sockaddr_in adr_prod ;
        char nom_machine_prod[LONGUEUR_NOM_MACHINE];
        struct hostent  *infos_machine_prod ;
```

```
/* Variables Consommateur */
int sock_cons ;
struct sockaddr_in adr_cons;
int lg_adr_cons ;

/* Variables de travail */
int i ;
int cpt ;
char msgerr[256] ;

/*-----*/

if( argc != 3 )
{
  fprintf( stderr, "Usage : %s <machine producteur> <port producteur>\n", argv[0
  exit(-1);
}

strcpy( nom_machine_prod , argv[1] ) ;
sscanf( argv[2] , "%d" , &port_prod ) ;

/* Creations points d'entrees */
if( (sock_cons = socket(AF_INET, SOCK_DGRAM, 0 )) == -1 )
{
     perror("Pb socket consommateur");
     exit(-1);
}

/* Creations des adresses */
/* -- consommateur */
adr_cons.sin_port = 0 ;
adr_cons.sin_family = AF_INET ;
adr_cons.sin_addr.s_addr = INADDR_ANY;
bzero( adr_cons.sin_zero , sizeof(adr_cons.sin_zero) ) ;

/* -- producteur */
adr_prod.sin_family = AF_INET ;
adr_prod.sin_port = htons(port_prod) ;
if( (infos_machine_prod = gethostbyname(nom_machine_prod)) == NULL )
{
     sprintf( msgerr , "Pb sur le nom machine %s", argv[1]);
     perror(msgerr);
     exit(-2);
}
bcopy( infos_machine_prod->h_addr, &adr_prod.sin_addr, infos_machine_prod->h_leng
bzero( adr_prod.sin_zero , sizeof(adr_prod.sin_zero) ) ;

/* Attachements des points d'entrees aux adresses */
if( bind(sock_cons,(struct  sockaddr *)&adr_cons,sizeof(adr_cons)) == -1 )
```

```
    {
        perror("Pb bind sur socket consommateur");
        exit(-3);
    }
    lg_adr_cons = sizeof(adr_cons);
    if(getsockname(sock_cons,
                   (struct sockaddr *)&adr_cons ,
                   (socklen_t *)&lg_adr_cons))
    {
        perror("Pb getsockname sur socket consommateur");
        exit(-4);
    }

    /* initialisation des pseudo-aleatoires */
    srandom((unsigned int)getpid());

    /* Envoi des N messages vides au producteur */
    for( i=0 ; i<N ; i++ )
    {
        send_message( VIDE,
                      sock_cons,
                      adr_prod );
    }

    /* Consommation des elements */
    cpt = 0 ;
    while(B_TRUE)
    {
        /* Extraction message plein */
        receive_message( sock_cons ,
                         &adr_prod );

        /* Consommation element */
        cpt++;
        printf("Consommation element %d\n" , cpt );

        /* Envoi message vide */
        send_message( VIDE,
                      sock_cons,
                      adr_prod );

        /* Attente */
        sleep( random() % 5 ) ;
    }
}
```

# 4 Les barrières

Ce mécanisme 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.

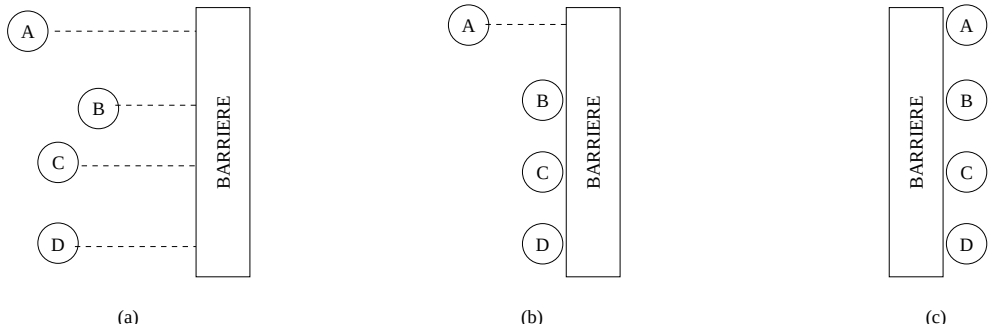


FIG. 1 – Fonctionnement d'une barrière (a) Processus approchant de la barrière (b) On attend le dernier (c) Une fois le dernier arrivé, tous les processus peuvent passer