HMIN105M : Principes de la programmation concurrente et répartie

Responsable : Hinde Bouziane (bouziane@lirmm.fr)
Intervenants : H. Bouziane et T. Lorieul

UM - LIRMM

Exemple conducteur

- On veut traiter une image, c'est-à -dire une matrice de points (pixels). Chaque point est une structure de données (couleur, profondeur, etc).
- On veut en particulier faire un traitement en parallèle sur les lignes impaires et les lignes paires.
- On défini deux fonctions *IImpair()* (pour le traitement des lignes impaires) et *IPair()* (resp. lignes paires).

- **Question 1** : Pourquoi paralléliser ?
- **Question 2** : Le compilateur (ou système) est-il capable de détecter des fonctions indépendantes ?
- **Exercice** : Avec les moyens que vous connaissez, proposez une solution.

Retour sur la notion de processus

Un processus constitue une seule **unité d'exécution** qui s'exécute séquentiellement sur un seul processeur, même si :

- dans un programme, il y a des parties indépendantes les unes des autres, qui pourraient s'exécuter en parallèle,
- l'ordinateur comporte plusieurs processeurs.

Un moyen pour faire du parallélisme consiste à créer plusieurs processus. Dans le cas de processus lourds :

- le changement de contexte peut être coûteux
- l'espace d'adressage du processus n'est pas partageable : obligation de passer par des outils de communication (tubes, files de messages, mémoires partagées et autres outils)
- outils de synchronisation entre processus difficiles.

Nous allons voir ...

- Comment paralléliser au sein d'un même processus
- Partager des données entre différentes parties parallèles
- Comment gérer la synchronisation (imposer un ordre, exclusion mutuelle, etc)

Notion de *thread*

Thread = fil en anglais. Un thread est un fil d'exécution

- Traductions : activité, tâche, fil d'exécution ou processus léger
- Les threads permettent de dérouler plusieurs suites d'instructions, en parallèle, à l'intérieur du même processus
- Un thread est une partie d'un processus ou un chemin d'exécution à l'intérieur d'un processus
- Concrètement, un thread exécute une fonction
  - chaque thread a sa propre pile et des variables locales
  - un thread peut partager des données en mémoire avec d'autres threads. Ainsi, la communication entre threads se fait via le partage de variables.
  - les threads offrent un mécanisme permettant à un processus de réaliser plusieurs unités d'exécution de façon *asynchrone*.

Notion de *thread* - suite

- Dans un processus, on aura un thread *principal*, celui exécutant la fonction `main()` et des threads *secondaires*.
- Sur une machine multi-processeur, chaque thread peut s'exécuter sur un processeur, indépendamment d'un autre.
- Le système gère la commutation de contexte entre threads.

Dans ce cours, l'interface de programmation normalisée POSIX est utilisée. Le mot *pthread* désignera les threads tels qu'ils sont vus dans cette interface.

Retour à l'exemple conducteur

Rappel : deux fonctions, *lImpair()* et *lPair()*, à exécuter en parallèle.

Un schéma possible de traitement de l'image avec des threads :

```
int main(){
  ...
  // définition de la matrice (image)
  ...
  pthread_create(lImpair,...);
  pthread_create(lpair,...);
  ...
}
```

Pour la suite

Tous les objets et fonctions manipulés ont la forme

`pthread_`objet`_t`
`pthread_`objet`_`opération()

Exemples : `pthread_t`, `pthread_mutex_t`, `pthread_create()`, `pthread_mutex_lock()`.

Ces objets et fonctions sont définis dans le fichier `pthread.h` à inclure.

`pthread_t` est le type *opaque* identifiant un thread, classiquement, un entier.

Remarque : l'option `-lpthread` peut être nécessaire à la compilation

Premières fonctions

Action	Fonction	Remarques
création	pthread_create()	activité créée à l'état prêt
fin	pthread_exit()	différent de exit() !
identification	pthread_self()	résultat de type `pthread_t`
égalité	pthread_equal()	portabilité : éviter $=$

Création

Prototype :

```
int    pthread_create(                  // résultat 0 si réussite, ≠ 0 sinon
       pthread_t * idThread,            // identité obtenue en résultat
       pthread_attr_t *attributs,       // NULL pour commencer
       void * (*fonction)(void *),      // fonction à démarrer
       void * param) ;                  // paramètre(s) à passer à la fonction
```

Cette fonction démarre l'exécution d'un nouveau thread, en parallèle avec celui qui l'a appelé (thread principal ou secondaire), mais dans le **même** processus !

Le dernier argument permet de passer un paramètre à la fonction, ou plusieurs regroupés dans une structure.

Abandon, identification, égalité

Prototype abandon/fin :

```
void pthread_exit(void * retour);
```

Le paramètre `retour` est un résultat, *(valeur de retour)*, pouvant être consulté par un autre thread du même processus, attendant la fin de celui qui vient d'abandonner.

Prototype identification :

```
pthread_t pthread_self(void);
```

renvoie l'identité du thread appelant.

Prototype égalité :

```
int pthread_equal(pthread_t idT1, pthread_t idT2);
```

renvoie une valeur non nulle si réussite, 0 si échec.

Exemple

```
using namespace std;
#include <pthread.h>
#include <iostream>
#include <sys/types.h>

void *monThread (void * par){
  pthread_t moi = pthread_self();
  cout<< "thread " << moi << ", proc. "<< getpid()<< endl;
  pthread_exit();
}

int main(){
  pthread_t idTh;
  if (pthread_create(&idTh, NULL, monThread, NULL) != 0)
     cout << "erreur creation" <<endl;
     //suite...en particulier attendre la fin du thread!
}
```

Exercice

- Ecrire le programme principal pour l'exemple conducteur
- L'image est passée en paramètre des fonctions *IImpair(...)* et *IPair(...)* (unique paramètre).
  - Donner la signature de ces fonctions
- L'image est définie par une structure *Image* que vous n'avez pas à définir

Remarques importantes

- Lorsqu'un thread, principal ou secondaire, fait `exit()`, il termine le processus ! Donc **tous** les threads seront arrêtés.
- Il vaut mieux attendre la fin des divers threads dans le thread principal.

- L'ordonnancement des threads, principal ou secondaires se fait en fonction de leurs priorités par défaut (modifiables).
- Sauf traitements particuliers, il vaut mieux utiliser des outils de synchronisation plutôt que de jouer sur les priorités.

Formes de synchronisation

Plusieurs outils de synchronisation de threads permettent :

- d'attendre la fin d'un thread,
- de créer des *verrous* binaires (à deux états, verrouillé ou non), utilisables par des threads appartenant au même (voir différents) processus,
- de définir des *variables conditionnelles* permettant d'attendre l'occurrence d'un événement.

Synchronisation : problème 1

```
...
void *monThread (void * par){
  pthread_t moi = pthread_self();
  cout<< "monThread " << moi<< ", proc. "<<getpid()<< endl;
  calculDureeSec(3); // calcul qui dure 3 sec.
  cout<< "monThread : fin"<< endl;
  pthread_exit();
}
int main(){
  pthread_t idTh;
  if (pthread_create(&idTh, NULL, monThread, NULL) != 0)
     cout << "erreur creation" <<endl;
  cout<< "principal : fin"<< endl;
}
```

Question : que se passe-t-il à l'exécution ?

Synchronisation : attente de la fin de l'exécution d'un thread

La forme la plus simple de synchronisation est l'attente de fin d'un thread, équivalente à `wait()` pour les processus *parents* attendant la terminaison de leurs *enfants*.
Ici, un thread quelconque, principal ou secondaire, peut attendre la fin d'un autre qu'il connaît :

Prototype

```
int pthread_join(pthread_t idT, void **retourCible);
```

Elle permet au thread appelant d'attendre la fin de celui issu du même processus, identifié par `idT`. Résultat 0 si réussite, $\neq 0$ sinon.

L'appelant est bloqué si le thread *idT* n'est pas terminé. Il sera débloqué lorsque le thread *idT* aura fait `pthread_exit()`.

Retour à l'exemple

```
...
void *monThread (void * par){
  pthread_t moi = pthread_self();
  cout<< "monThread " << moi<< ", proc. "<<getpid()<< endl;
  calculDureeSec(3); // calcul qui dure 3 sec.
  cout<< "monThread : fin"<< endl;
  pthread_exit();
}
int main(){
  pthread_t idTh;
  if (pthread_create(&idTh, NULL, monThread, NULL) != 0)
     cout << "erreur creation" <<endl;
  int res = pthread_join(idTh, NULL);
  cout<< "principal : fin"<< endl;
}
```

Synchronisation : problème 2 (partie 1)

```
...
void *T1 (void * par){
  int * cp = (int*)(par);
  for(int i=0; i < 1500; i++)  ++(*cp);
  pthread_exit(NULL);
}

void *T2 (void * par){
  int * cp = (int*)(par);
  for(int i=0; i < 3000; i++) ++(*cp);
  pthread_exit(NULL);
}
```

Synchronisation : problème 2 (partie 2)

```
int main(){
  pthread_t idT1, idT2;
  int counter = 0;

  if (pthread_create(&idT1, NULL, T1, &counter) != 0)
     cout << "erreur creation" <<endl;
  if (pthread_create(&idT2, NULL, T2, &counter) != 0)
    cout << "erreur creation" <<endl;

  int res = pthread_join(idT1, NULL);
  res = pthread_join(idT2, NULL);
  std::cout<<" Total = "<<counter<<std::endl;

  return 0;
}
```

Question : Quel est le résultat final ?

Synchronisation : verrous

- Un *verrou* est un sémaphore ayant deux états possibles : **libre** ou **occupé** (verrouillé).
- Lorsqu'un verrou est libre, un thread peut demander de le verrouiller. Un seul thread peut obtenir le verrouillage.
- Lorsqu'un thread a obtenu le verrouillage, un autre thread qui demande le verrouillage du même verrou sera bloqué (ou échouera dans un contexte non bloquant).
- Seul le thread qui a obtenu le verrouillage peut déverrouiller un verrou.
- Le verrouillage et déverrouillage sont des opérations atomiques.
- Un verrou est appelé `mutex`. Il est du type `pthread_mutex_t`.

Fonctions des verrous

Un mutex doit être déclaré dans un espace commun aux threads l'utilisant. Par exemple, dans le segment des données statiques.

Une fonction manipulant un mutex est de la forme
`pthread_mutex_fonction`

Fonction	Résultat
`pthread_mutex_init()`	verrou créé ; état libre
`pthread_mutex_lock()`	verrouillage
`pthread_mutex_trylock()`	verrouillage si état libre sinon, erreur sans blocage
`pthread_mutex_unlock()`	déverrouillage ; état libre
`pthread_mutex_destroy()`	destruction

Une initialisation plus simple :

```
pthread_mutex_t verrou = PHTREAD_MUTEX_INITIALIZER;
```

Retour à l'exemple

Exclusion mutuelle entre deux threads

```
//donnée statique
pthread_mutex_t verrou = PTHREAD_MUTEX_INITIALIZER;

void *monThread (void * par){
  pthread_t moi = pthread_self();
  cout << "monThread "<< moi << ", proc"<< getpid()<< endl;
  pthread_mutex_lock(&verrou);
  cout << "monThread : verrou obtenu"<< endl;
  calculDureeSec(5); // calcul qui dure 5 sec.
  cout<< "monThread : fin section critique" << endl;
  pthread_mutex_unlock(&verrou);
  cout << "monThread : verrou libéré "<< endl;
  pthread_exit();
}
```

Retour à l'exemple - suite

```
int main(){
  pthread_t idTh;
  if (pthread_create(&idTh, NULL, monThread, NULL) != 0)
     cout << "erreur creation" << endl;
  pthread_mutex_lock(&verrou);
  cout<< "principal : verrou obtenu" << endl;
  calculDureeSec(5); // calcul qui dure 5 sec.
  cout<< "principal : fin section critique" << endl;
  pthread_mutex_unlock(&verrou);
  cout<< "principal : verrou libéré" << endl;
  int retThr = pthread_join(idTh,NULL);
}
```

Question : Quelles sont les traces d'exécution possibles ?

Exercice

- Donner une solution au ”problème 2”

A retenir

Règle : Tout accès à une variable accessible en lecture par un thread et en écriture par un autre doit être protégé.

- Un seul mutex peut protéger plusieurs variables, mais pas l'inverse.
- Les opérations *...lock()* et *...unlock()* sont atomiques, mais pas la portion de code qui se trouve entre les deux.
- Cette portion de code est appelée **section critique**
- Si un thread est dans une section critique, il doit être garanti qu'aucun autre thread n'y soit simultanément
  - Nous parlons d'**exclusion mutuelle**
- C'est le cas des exemples précédents.

Synchronisation : autre problème

Acquis : Les verrous permettent de gérer facilement le partage de données communes accessibles en lecture et écriture.

Problème : Dans plusieurs situations, on a besoin de connaître l'état d'une donnée commune (sous-ensemble de valeurs parmi les valeurs possibles) afin de réaliser ou non des opérations en fonction de cet état.

Exemple Simple :
Deux threads T_1, T_2 travaillent sur un entier x commun, T_1 effectue des opérations uniquement si $x > seuil$, et seule T_2 peut engendrer une telle situation.

Exercice :

- montrer qu'on peut résoudre un tel problème avec un *mutex* ;
- montrer que cette solution est inefficace.

A éviter !

Partie Commune

```
int x, seuil;
pthread_mutex_t verrou;
...//initialisations diverses
```

Thread T_1

```
.... ;
pthread_mutex_lock(&verrou) ;
tant que (x ≤ seuil) faire {
    pthread_mutex_unlock(&verrou) ;
    pthread_mutex_lock(&verrou) ;
}
... // travail sur x;
pthread_mutex_unlock(&verrou) ;
...;
```

Thread T_2

```
.... ;
pthread_mutex_lock(&verrou) ;
... // travail sur x;
pthread_mutex_unlock(&verrou) ;
... ;
```

Solution (éventuellement) correcte mais inefficace !

Objectif

Thread 1

bloquer l'accès ;
si *prédicat non satisfait* **alors** {
    relâcher accès;
    attendre event (prédicat satisfait);
    bloquer l'accès ;
}

section critique

relâcher l'accès ;

Thread 2

bloque l'accès ;

section critique

signaler event(prédicat satisfait) ;
relâcher l'accès ;

Avec :

- libération du verrou et passage à l'état bloqué de façon atomique,
- réveil en retrouvant le verrou.

Remarque

Plusieurs threads en attente de satisfaction d'un même prédicat.

Thread 1

bloquer l'accès ;
tant que *prédicat non satisfait* **faire** {
    relâcher accès;
    attendre event (prédicat satisfait);
    bloquer l'accès ;
}
section critique

relâcher l'accès ;

Thread 2

bloque l'accès ;

section critique

signaler event(prédicat satisfait) ;
relâcher l'accès ;

Plusieurs threads attendent sur le même prédicat, un seul obtiendra le verrou.

Synchronisation : variable conditionnelle

Une *variable conditionnelle* est une donnée commune à plusieurs threads, fonctionnant comme un booléen et symbolisant l'occurrence d'un événement, autrement dit, la satisfaction d'un prédicat.

Exemple :

- Une variable conditionnelle *ilYaDuBoulot* est reliée à l'événement $x > seuil$ (le prédicat).
- Si le prédicat rend faux, T_1 doit patienter jusqu'à ce que *ilYaDuBoulot* soit annoncée par une tâche T_2. Ainsi, T_1 évitera de bloquer un verrou pour rien
- T_2 devra donc vérifier le résultat du prédicat. Si le résultat est vrai, T_2 devra l'annoncer à T_1, et par extension aux autres tâches intéressées.

Difficulté

- Ne pas confondre l'événement et la variable conditionnelle qui l'annonce.
- Au fonctionnement décrit, on doit associer un *verrou* qui permet de protéger l'accès à la donnée commune et l'*événement* attendu (le résultat du prédicat).
- Il faut obtenir un fonctionnement cohérent, combinant le verrou, le prédicat et la variable conditionnelle.

Schéma de fonctionnement

- Un thread ayant réussi le verrouillage d'un verrou, pourra le relâcher et passer à l'état bloqué pour attendre un évènement, ceci de façon **atomique**. Il demandera aussi à être réveillé en retrouvant l'état verrouillé.
- Le réveil sera réalisé par un autre thread qui constatera qu'il peut activer un ou plusieurs threads en attente par l'intermédiaire d'une variable conditionnelle.
- Si plusieurs threads sont réveillés, un seul obtiendra le verrou, les autres ne le retrouveront que lorsqu'il sera déverrouillé.

Schéma algorithmique

Partie Commune

```
int dCommune;
pthread_mutex_t verrou;
pthread_cond_t condi;
...// initialisations
```

Thread 1

.... ;
pthread_mutex_lock(&verrou) ;
tant que *(dCommune hors bornes)*
faire
    attendre(&condi, &verrou) ;
... // ici, je suis réveillé : travailler ;
pthread_mutex_unlock(&verrou) ;

Thread 2

.... ;
pthread_mutex_lock(verrou) ;
accèsEtModif(dCommune) ;
si *(dCommune dans bornes)* **alors**
  réveillerTâches(&condi) ;
pthread_mutex_unlock(verrou) ;

Actions possibles

Sur une variable conditionnelle `c` et un verrou `v`, on peut effectuer les actions suivantes :

Fonction	Action
`pthread_cond_init(&c)`	crée la variable conditionnelle `c`
`pthread_cond_wait(&c,&v)`	bloque l'appelant **et** rend le verrou de façon atomique
`pthread_cond_timedwait (&c,&v,&délai)`	..._wait() avec attente temporelle
`pthread_cond_broadcast(&c)`	libère **tous** les threads bloqués
`pthread_cond_signal(&c)`	libère un seul thread
`pthread_cond_destroy(&c)`	destruction

Retour : Toutes ces fonctions retournent 0 (zéro) en cas de succès et un résultat non-nul en cas d'erreur, accompagné d'un code d'erreur.

Création, destruction

Prototype Création

```
int pthread_cond_init (          // résultat 0 si réussite, ≠ 0 sinon
pthread_cond_t *cond ,           // variable conditionnelle à créer
pthread_condattr_t *attr) ;      // NULL par défaut
```

Permet de créer une variable conditionnelle et de l'initialiser.
Note : pour des raisons de portabilité, utiliser l'initialisation par défaut.

Une initialisation plus simple peut être effectuée par la déclaration :

```
pthread_cond_t uneCond = PHTREAD_COND_INITIALIZER;
```

Prototype destruction

```
int pthread_cond_destroy(pthread_cond_ *cond);
```

Détruit la variable conditionnelle pointée.

Attente non bornée

Prototype d'attente non bornée :

```
int pthread_cond_wait (          // résultat 0 si réussite, ≠ 0 sinon
pthread_cond_t *vcond,           // variable conditionnelle associée
                                 // à l'évennement attendu
pthread_mutex_t *verrou) ;       // verrou d'accès à la donnée commune
```

Cette primitive réalise un appel bloquant, qui **de façon atomique** déverrouille `verrou` **et** attend que la condition `vcond` soit annoncée, forcément par un autre thread (voir ci-après pour l'annonce).

Note : La primitive suppose que le thread appelant a obtenu précédemment le verrouillage de `verrou`.

Attention : plusieurs threads peuvent être débloqués. Il est donc utile de tester à nouveau le prédicat après réveil (à la sortie de l'attente !)

Exemple

On reprend l'exemple de la tâche T_1 attendant que le prédicat $x > seuil$ soit vrai pour continuer son travail.

Partie Commune

```
int x, seuil;
pthread_mutex_t verrou;
pthread_cond_t condi;

...//initialisations diverses
```

Thread T_1

.... ;
pthread_mutex_lock(&verrou) ;
tant que *(x ≤ seuil)* **faire** pthread_cond_wait(&condi, &verrou) ;
... // ici, je suis réveillé et le verrou est verrouillé ; travail... ;
pthread_mutex_unlock(&verrou) ;

Remarques importantes

- L'attente provoque le déverrouillage de `verrou`, donc un autre thread peut le verrouiller,
- Tous les threads attendant la même variable conditionnelle doivent spécifier le même verrou dans l'attente : une variable conditionnelle est associée à un et un seul verrou (mutex), mais un verrou peut être associé à un nombre quelconque de conditions,
- la variable conditionnelle `condi` est utilisée comme un avertisseur (un drapeau) : lorsqu'on est averti, il s'est passé un événement (le prédicat rend vrai), sur lequel on a demandé un réveil.

Annonces

Il y a deux façons permettant de réveiller des threads en attente sur une condition :

- réveiller tous les threads en attente,
- réveiller un seul thread, mais ce sera un parmi ceux qui attendent, sans pouvoir choisir.

Prototype 1

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

provoque le réveil d'un seul thread.

Attention : il n'y a aucun rapport entre `signal` dans la fonction de réveil de thread vu ici et le *signal* vu en cours de système comme une forme d'interruption logicielle.

Annonces - suite

Prototype 2

`int pthread_cond_broadcast(pthread_cond_t *cond);`
provoque le réveil de tous les threads attendant la variable conditionnelle `cond`.

Important : Lors du réveil, les threads réveillés vont obtenir tour à tour automatiquement le verrouillage du verrou.

Exemple

On reprend l'exemple du thread T_1 attendant que le prédicat $x > seuil$ rende vrai pour continuer son travail. Ici, le cas de T_2 qui réveille T_1.

Tâche T_2

```
.... ;
pthread_mutex_lock(&verrou) ;
.... ; //travail sur x et/ou seuil ;
si (x > seuil) alors pthread_cond_broadcast(&condi) ;
pthread_mutex_unlock(&verrou) ;
```

Question : que se passe-t-il si aucun thread n'attend ?
Réponse : le réveil est **perdu** ! C'est logique, mais suppose que tous les threads testent le prédicat **avant** d'attendre.

Discussions : le réveil - 1

Est-il préférable d'utiliser `broadcast` ou `signal` ?

Sur le plan de l'efficacité du fonctionnement du système, il serait préférable de ne réveiller qu'un thread, puisque de toute façon un seul obtiendra le verrou.

Examinons cette situation : des threads T_1, T_3, T_4, ... attendent une variable conditionnelle et T_2 se charge du réveil :

- **cas signal** : si un seul est réveillé par T_2, comment seront réveillés les autres ? Il faudra le prévoir.
- **cas broadcast** : du moment que tous les threads ont été réveillés, tous les threads obtiendront chacun à son tour le verrouillage du verrou, même si un seul l 'obtient à la fois. **Attention :** chacun devra re-tester le prédicat ; s'il reste vrai, il travaillera ; sinon il faudra refaire une attente.

Discussions : le réveil - 2

Recommandations pour le choix `signal` **ou** `broadcast` :

- Utiliser `signal` seulement si on est certain que **n'importe** quelle tâche en attente peut faire le travail requis **et** qu'il est indispensable de réveiller une seule tâche.
- Lorsqu'une variable conditionnelle est utilisée pour plusieurs prédicats, `signal` est à prohiber.
- En cas de doute, utiliser `broadcast`.

Discussions : le réveil - 3

Déverrouiller après ou avant l'annonce ? choix délicat.

- **Après** engendre qu'un thread réveillé ne pourra pas obtenir le verrouillage immédiatement car le verrou est toujours indisponible. Donc le thread réveillé devra se bloquer temporairement.
- **Avant** peut être plus efficace, mais il se peut aussi qu'un thread T_z non (encore) en attente obtienne le verrouillage. Il n'y a pas d'équité, alors que le thread réveillé T_a peut être plus prioritaire (T_z moins prioritaire a obtenu le verrouillage alors que T_a, en attente de l'annonce, ne pouvait l'obtenir).

Discussions : test et re-test du prédicat

Pourquoi avoir insisté pour que tout thread fasse le test du prédicat après le réveil ?

- Comme déjà vu, lorsque plusieurs threads sont réveillés, il est possible que le prédicat rende faux à nouveau après l'action d'un des thread activés.
- Un thread peut être réveillé pour une raison *différente* de celle pour laquelle il s'attend à être réveillé.
- Pour des raisons de robustesse.
- D'autres raison propres à chaque application (éviter toutes les formes de *c'est écrit partout...*y compris ce cours).

Exemple : producteur - consommateur

- Un producteur dépose des messages qui seront extraits par un consommateur.
- Les communications se font à travers une file (FIFO) circulaire commune de taille limitée.
- Le producteur ne peut déposer un message dans la file tant que la file est pleine.
- Le consommateur ne peut extraire un message de la file tant que la file est vide.

Exemple : déclarations et initialisations

```
#include <stdio.h>
#include <pthread.h>
#define BUFFER_SIZE 16

/* circular buffer of integers */
struct prodcons {
  int buffer [BUFFER_SIZE ]; /* the actual data */
  int read_pos, write_pos;   /* positions for read and write */
  pthread_mutex_t lock;      /* mutex ensuring exclusive */
                             /* access to buffer */
  pthread_cond_t notempty;   /* signaled when buffer is not empty */
  pthread_cond_t notfull;    /* signaled when buffer is not full */
};

/* Initialize a buffer */
void init(struct prodcons b){
  pthread_mutex_init(&b->lock, NULL);
  pthread_cond_init(&b->notempty, NULL);
  pthread_cond_init(&b->notfull, NULL);
  b->read_pos = 0;
  b->write_pos = 0;
}
```

Exemple : retrait d'un message

```
int get(struct prodcons *b){
  int data;
  pthread_mutex_lock(&b->lock);
  while (b->write_pos == b->read_pos) {
    /* Wait until buffer is not empty */
    pthread_cond_wait(&b->notempty, &b->lock);
  }
  data = b->buffer[b->read_pos];
  b->read_pos++;
  if (b->read_pos >= BUFFER_SIZE)
    b->read_pos = 0;

  // signal that the buffer is now not full
  pthread_cond_signal(&b->notfull);
  pthread_mutex_unlock(&b->lock);
  return data;
}
```

Exemple : dépôt d'un message

```
void put(struct prodcons *b, int data){
  pthread_mutex_lock(&b->lock);
  while ((b->write_pos + 1) % BUFFER_SIZE == b->read_pos){
    /* Wait until buffer is not empty */
    pthread_cond_wait(&b->notfull, &b->lock);
  }
  b->buffer[b->write_pos] = data;
  b->write_pos++;
  if (b->write_pos >= BUFFER_SIZE)
    b->write_pos = 0;

  // signal that the buffer is now not empty
  pthread_cond_signal(&b->notempty);
  pthread_mutex_unlock(&b->lock);
}
```

Exemple : producteur

```
#define OVER (-1)
void * producer (void * par){
  int n;
  for (int n = 0; n < 1000; n++){
    printf ("prod --> %d\n", n);
    put(&buffer, n);
  }

  put(&buffer, OVER);
  pthread_exit();
}
```

Exemple : consommateur

```
void * consumer (void * p){
  int d;
  while (1){
    d = get (buffer);
    if (d == OVER) break;
    printf ("cons --> %d\n", d);
  }
  pthread_exit();
}
```

Exemple : programme principal

```
struct prodcons buffer;
int main (){
  pthread_t th_p, th_c;
  void * retval;
  init (&buffer);
  /* Create the threads */
  pthread_create (&th_p, NULL, producer, 0);
  pthread_create (&th_c, NULL, consumer, 0);
  /* Wait until producer and consumer finish */
  pthread_join (th_p, &retval);
  pthread_join (th_c, &retval);
  return 0;
}
```