

Variables de condition

F. Gluck, *V. Pilloux*

Version 0.35

Introduction

- Une variable de condition est un mécanisme de **synchronisation** reposant sur le test d'une condition, basé sur le test d'une variable:
 - un (ou plusieurs) thread est bloqué tant que la condition n'est pas satisfaite ;
 - un (ou plusieurs) thread est réveillé lorsque la condition devient vraie.
- Le blocage d'un thread est une attente passive.

Si une **condition** peut être exprimée dans un programme, alors une **variable de condition** peut être utilisée !

Définition d'une VC

- Une **variable de condition (VC)** est une condition sur laquelle un thread peut :
 - **attendre** jusqu'à ce que la condition devienne vraie (ou a changé d'état).
 - **notifier** (signaler) un ou plusieurs threads que la condition est devenue vraie (ou a changé d'état).
⇒ mécanisme de signalisation entre threads.
- Trois opérations sont associées à une variable de condition :
 1. **wait** : bloque jusqu'à ce qu'un **autre** thread signale la VC.
 2. **signal** : réveille **un** thread en attente sur la VC.
 3. **broadcast** : réveille **tous** les threads en attente sur la VC.
- **Toutes** les opérations sur une VC doivent être effectuées **en section critique** ⇒ **avec un mutex verrouillé !**

Définition alternative

- Une **variable de condition** est une file d'attente de thread(s) en attente sur une condition à l'intérieur d'une section critique.
- **Idée principale** : bloquer (attente passive) à l'intérieur de la section critique, mais en **relâchant** le verrou de manière atomique au moment de la mise en attente.
- Opérations :
 - **cond_wait(&lock)** : relâche le verrou et bloque le thread, le tout **atomiquement**. Lorsque la condition est ensuite signalée, le thread reprend son exécution, mais en **re-verrouillant** le verrou avant de continuer.
 - **signal** : réveille **un** des threads en attente sur la condition.
 - **broadcast** : réveille **tous** les threads en attente sur la condition.

Pour toute opération sur une variable de condition, le verrou doit être préalablement verrouillé !

Interface de programmation

- La librairie pthread met à disposition:
 - le type `pthread_cond_t`
 - des fonctions et macros d'initialisation et destruction:
 - `pthread_cond_init`(pthread_cond_t *cond,
pthread_condattr_t *cond_attr)
 - pthread_cond_t cond = `PTHREAD_COND_INITIALIZER`
 - `pthread_cond_destroy`(pthread_cond_t *cond)
 - des fonctions de manipulation:
 - `pthread_cond_wait`(pthread_cond_t *cond,
pthread_mutex_t *mutex)
 - `pthread_cond_signal`(pthread_cond_t *cond)
 - `pthread_cond_broadcast`(pthread_cond_t *cond)

Initialisation et destruction

```
// Initialisation statique
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

// Initialisation dynamique
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *cond_attr);

int pthread_cond_destroy(pthread_cond_t *cond);
```

- Passer un attribut de condition NULL à `pthread_cond_init` spécifie les attributs par défaut.
- **L'implémentation Linux ne supporte pas d'attribut**, donc `cond_attr` est ignoré.
- Toutes les fonctions sur les variables de condition renvoient 0 en cas de succès et != 0 en cas d'erreur.

Attente

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

- L'exécution du thread appelant est suspendue (attente passive) jusqu'à ce que la condition **cond** soit signalée.



le **mutex** doit être **verrouillé** par le thread **avant** l'appel à **pthread_cond_wait**.

- **pthread_cond_wait** effectue ces opérations de manière **atomique** :
 - **relâche mutex**
 - attend que **cond** soit signalée.
- Au moment où **cond** est signalée, **pthread_cond_wait** **re-verrouille** automatiquement le **mutex**.

Signalisation

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

- `pthread_cond_signal` réveille un des threads en attente sur `cond` :
 - si aucun thread n'est en attente, la fonction n'a aucun effet: **il n'y a pas de mémoire de l'événement comme les sémaphores!**
 - si plusieurs threads sont en attente, **un seul** est réveillé (choisi par l'ordonnanceur).
- `pthread_cond_broadcast` réveille **tous** les threads en attente sur `cond` :
 - si aucun thread n'est en attente, la fonction n'a aucun effet
 - les threads réveillés continuent leur exécution **chacun à leur tour** car le mutex ne peut être repris que par un thread à la fois (comme toujours, l'ordre est **imprévisible** et dépend de l'ordonnanceur).

➤ **Si aucun thread n'est en attente, la signalisation n'a aucun effet!**

Exemple 1

```
void *child(void *arg) {  
    // Comment indiquer que le  
    // thread est terminé ?  
    return NULL;  
}  
  
int main() {  
    pthread_t t;  
    pthread_create(&t, NULL, child, NULL);  
    // Comment attendre la fin du  
    // thread child ?  
    return 0;  
}
```

- Comment bloquer passivement `main` tant que `child` ne s'est pas terminé, sans utiliser de sémaphore, de barrière de synchronisation, ni d'appel à `pthread_join` ?
- Autrement dit, comment implémenter l'équivalent de `pthread_join` à l'aide de variables de condition ?

Exemple 1

```
int child_done = 0;
pthread_mutex_t m =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
    PTHREAD_COND_INITIALIZER;

void *child(void *arg) {
    pthread_mutex_lock(&m);
    child_done = 1;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
    return NULL;
}

void thr_join() {
    pthread_mutex_lock(&m);
    if (!child_done)
        pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}

int main() {
    pthread_t t;
    pthread_create(&t, NULL, child, NULL);
    thr_join();
    return 0;
}
```

Deux scénarios possibles:

Exemple 1

```
int child_done = 0;
pthread_mutex_t m =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
    PTHREAD_COND_INITIALIZER;

void *child(void *arg) {
    pthread_mutex_lock(&m);
    child_done = 1;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
    return NULL;
}

void thr_join() {
    pthread_mutex_lock(&m);
    if (!child_done)
        pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}

int main() {
    pthread_t t;
    pthread_create(&t, NULL, child, NULL);
    thr_join();
    return 0;
}
```

1 child s'exécute avant thr_join

- Le thread child s'exécute avant que main n'arrive à thr_join:
 - verouille m;
 - met child_done à 1;
 - signale main, ce qui n'a aucun effet (car aucun thread bloqué sur c).
- Ensuite, le thread main:
 - exécute thr_join;
 - verouille m;
 - teste child_done à vrai;
 - relâche m;
 - se termine.

Exemple 1

```
int child_done = 0;
pthread_mutex_t m =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
    PTHREAD_COND_INITIALIZER;

void *child(void *arg) {
    pthread_mutex_lock(&m);
    child_done = 1;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
    return NULL;
}

void thr_join() {
    pthread_mutex_lock(&m);
    if (!child_done)
        pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}

int main() {
    pthread_t t;
    pthread_create(&t, NULL, child, NULL);
    thr_join();
    return 0;
}
```

2 thr_join s'exécute avant child

- Thread main crée le thread child et appelle thr_join avant que child ne s'exécute:
 - verrouille m;
 - teste child_done à faux;
 - bloque sur cond_wait (**donc relâche m**).
- Ensuite, le thread child:
 - verrouille m;
 - met child_done à 1;
 - réveille (signal) main.
- Ensuite, le thread main:
 - reprend depuis l'appel à cond_wait **avec m verrouillé**;
 - puis relâche m;

Question 1

```
int child_done = 0;
pthread_mutex_t m =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
    PTHREAD_COND_INITIALIZER;

void *child(void *arg) {
    pthread_mutex_lock(&m);
    child_done = 1;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
    return NULL;
}

void thr_join() {
    pthread_mutex_lock(&m);
    if (!child_done)
        pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}

int main() {
    pthread_t t;
    pthread_create(&t, NULL, child, NULL);
    thr_join();
    return 0;
}
```

- La variable `child_done` est-elle vraiment nécessaire ?

Question 1

```
int child_done = 0;
pthread_mutex_t m =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
    PTHREAD_COND_INITIALIZER;

void *child(void *arg) {
    pthread_mutex_lock(&m);
child_done = 1;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
    return NULL;
}

void thr_join() {
    pthread_mutex_lock(&m);
if (!child_done)
    pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}

int main() {
    pthread_t t;
    pthread_create(&t, NULL, child, NULL);
    thr_join();
    return 0;
}
```

- La variable `child_done` est-elle vraiment nécessaire ?

Question 1

```
int child_done = 0;
pthread_mutex_t m =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
    PTHREAD_COND_INITIALIZER;

void *child(void *arg) {
    pthread_mutex_lock(&m);
child_done = 1;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
    return NULL;
}

void thr_join() {
    pthread_mutex_lock(&m);
if (!child_done)
    pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}

int main() {
    pthread_t t;
    pthread_create(&t, NULL, child, NULL);
    thr_join();
    return 0;
}
```

- La variable `child_done` est-elle vraiment nécessaire ?
- Si le thread `child` est exécuté avant que `main` n'appelle `thr_join`:
 - la VC **c** est signalée, mais cela n'a aucun effet, car aucun thread n'est bloqué ;
 - `main` va ensuite bloquer sur `cond_wait` dans la fonction `thr_join...`

⇒ **deadlock !**

Question 2

```
int child_done = 0;
pthread_mutex_t m =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
    PTHREAD_COND_INITIALIZER;

void *child(void *arg) {
    pthread_mutex_lock(&m);
    child_done = 1;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
    return NULL;
}

void thr_join() {
    pthread_mutex_lock(&m);
    if (!child_done)
        pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}

int main() {
    pthread_t t;
    pthread_create(&t, NULL, child, NULL);
    thr_join();
    return 0;
}
```

- L'utilisation du mutex `m` est-elle vraiment nécessaire ?

Question 2

```
int child_done = 0;
pthread_mutex_t m =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
    PTHREAD_COND_INITIALIZER;

void *child(void *arg) {
    pthread_mutex_lock(&m);;
    child_done = 1;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);;
    return NULL;
}

void thr_join() {
    pthread_mutex_lock(&m);;
    if (!child_done)
        pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);;
}

int main() {
    pthread_t t;
    pthread_create(&t, NULL, child, NULL);
    thr_join();
    return 0;
}
```

- L'utilisation du mutex `m` est-elle vraiment nécessaire ?
- Si la fonction `thr_join` est exécutée avant le thread `child`:
 - `child_done` est testé à faux, mais le thread est préempté **avant** de faire le `pthread_cond_wait`;
 - `child` prend la main, met `child_done` à 1 et signale la VC;
 - `thr_join` reprend la main et bloque dans l'appel à `pthread_cond_wait`.

⇒ **deadlock !**

Première conclusion

```
int child_done = 0;
pthread_mutex_t m =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
    PTHREAD_COND_INITIALIZER;
```

➤ **Toujours** verrouiller le mutex **pendant la signalisation** d'une VC : verrouiller un mutex pendant l'attente sur une VC est imposé par la sémantique du wait. Sinon il se peut qu'un thread se trouve entre son test de condition et le cond_wait(): dans ce cas il ratera le signal.

➤ Un test sur une ou plusieurs variables est nécessaire pour l'utilisation d'une VC

```
int main() {
    pthread_t t;
    pthread_create(&t, NULL, child, NULL);
    thr_join();
    return 0;
}
```

- L'utilisation du mutex m est-elle vraiment nécessaire ?

- main reprend la main et bloque dans l'appel à wait.

⇒ **deadlock !**

Producteur-consommateur simple

- Considérons le modèle producteur-consommateur simple, dans le cas d'un buffer à un slot.
- Plusieurs producteurs et consommateurs sont autorisés.
- Comment synchroniser l'accès à l'aide de variables de condition ?

```
item_t buffer;  
int count = 0;  
  
void put_item(item_t item) {  
    buffer = item;  
    count = 1;  
}  
  
item_t get_item() {  
    item_t item = buffer;  
    count = 0;  
    return item;  
}
```

Solution 1

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    if (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    if (count == 0)
        cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Cette solution est-elle correcte ?

Solution 1

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    if (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    if (count == 0)
        cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Oui... mais pas dans le cas de plusieurs producteurs et plusieurs consommateurs !
- Pourquoi ?

Solution 1

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    if (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    if (count == 0)
        cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (P) et deux consommateurs (C1, C2) et le buffer vide (count \equiv 0).

Solution 1

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    if (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    C1 mutex_lock(&mutex);
    if (count == 0)
        cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (P) et deux consommateurs (C1, C2) et le buffer vide (count \equiv 0).
- C1 s'exécute, puis bloque sur le wait (à ce moment il relâche le mutex).

Solution 1

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    if (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    C1 if (count == 0)
        cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (P) et deux consommateurs (C1, C2) et le buffer vide (count \equiv 0).
- C1 s'exécute, puis bloque sur le wait (à ce moment il relâche le mutex).

Solution 1

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    if (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    if (count == 0)
        C1 cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (P) et deux consommateurs (C1, C2) et le buffer vide (count \equiv 0).
- C1 s'exécute, puis bloque sur le wait (à ce moment il relâche le mutex).

Solution 1

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
P  mutex_lock(&mutex);
    if (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    if (count == 0)
C1  cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (P) et deux consommateurs (C1, C2) et le buffer vide ($\text{count} \equiv 0$).
- C1 s'exécute, puis bloque sur le wait (à ce moment il relâche le mutex).
- P est ordonnancé, produit un item ($\text{count} \equiv 1$), signale la condition et relâche le mutex.

Solution 1

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    P if (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    if (count == 0)
        C1 cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (P) et deux consommateurs (C1, C2) et le buffer vide ($\text{count} \equiv 0$).
- C1 s'exécute, puis bloque sur le wait (à ce moment il relâche le mutex).
- P est ordonnancé, produit un item ($\text{count} \equiv 1$), signale la condition et relâche le mutex.

Solution 1

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    if (count == 1)
        cond_wait(&cond, &mutex);
    P buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    if (count == 0)
        C1 cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (P) et deux consommateurs (C1, C2) et le buffer vide ($\text{count} \equiv 0$).
- C1 s'exécute, puis bloque sur le wait (à ce moment il relâche le mutex).
- P est ordonnancé, produit un item ($\text{count} \equiv 1$), signale la condition et relâche le mutex.

Solution 1

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    if (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    P count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    if (count == 0)
        C1 cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (P) et deux consommateurs (C1, C2) et le buffer vide ($\text{count} \equiv 0$).
- C1 s'exécute, puis bloque sur le wait (à ce moment il relâche le mutex).
- P est ordonnancé, produit un item ($\text{count} \equiv 1$), signale la condition et relâche le mutex.

Solution 1

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    if (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    P cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    if (count == 0)
        C1 cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (P) et deux consommateurs (C1, C2) et le buffer vide ($\text{count} \equiv 0$).
- C1 s'exécute, puis bloque sur le wait (à ce moment il relâche le mutex).
- P est ordonnancé, produit un item ($\text{count} \equiv 1$), signale la condition et relâche le mutex.

Solution 1

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    if (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    P mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    if (count == 0)
        C1 cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (P) et deux consommateurs (C1, C2) et le buffer vide ($\text{count} \equiv 0$).
- C1 s'exécute, puis bloque sur le wait (à ce moment il relâche le mutex).
- P est ordonnancé, produit un item ($\text{count} \equiv 1$), signale la condition et relâche le mutex.

Solution 1

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    if (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    if (count == 0)
        C1 cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (P) et deux consommateurs (C1, C2) et le buffer vide ($\text{count} \equiv 0$).
- C1 s'exécute, puis bloque sur le wait (à ce moment il relâche le mutex).
- P est ordonnancé, produit un item ($\text{count} \equiv 1$), signale la condition et relâche le mutex.

Solution 1

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    if (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    C2 mutex_lock(&mutex);
    if (count == 0)
        C1 cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (**P**) et deux consommateurs (**C1**, **C2**) et le buffer vide ($\text{count} \equiv 0$).
- **C1** s'exécute, puis bloque sur le wait (à ce moment il relâche le mutex).
- **P** est ordonnancé, produit un item ($\text{count} \equiv 1$), signale la condition et relâche le mutex.
- Avant que **C1** ne soit réveillé, **C2** est ordonnancé et consomme l'item du buffer ($\text{count} \equiv 0$) et signale **cond**.

Solution 1

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    if (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    C2 if (count == 0)
    C1     cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (**P**) et deux consommateurs (**C1**, **C2**) et le buffer vide ($\text{count} \equiv 0$).
- **C1** s'exécute, puis bloque sur le wait (à ce moment il relâche le mutex).
- **P** est ordonnancé, produit un item ($\text{count} \equiv 1$), signale la condition et relâche le mutex.
- Avant que **C1** ne soit réveillé, **C2** est ordonnancé et consomme l'item du buffer ($\text{count} \equiv 0$) et signale **cond**.

Solution 1

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    if (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    if (count == 0)
        C1 cond_wait(&cond, &mutex);
    C2 item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (**P**) et deux consommateurs (**C1**, **C2**) et le buffer vide ($\text{count} \equiv 0$).
- **C1** s'exécute, puis bloque sur le wait (à ce moment il relâche le mutex).
- **P** est ordonnancé, produit un item ($\text{count} \equiv 1$), signale la condition et relâche le mutex.
- Avant que **C1** ne soit réveillé, **C2** est ordonnancé et consomme l'item du buffer ($\text{count} \equiv 0$) et signale **cond**.

Solution 1

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    if (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    if (count == 0)
        C1 cond_wait(&cond, &mutex);
    item_t item = buffer;
    C2 count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (**P**) et deux consommateurs (**C1**, **C2**) et le buffer vide ($\text{count} \equiv 0$).
- **C1** s'exécute, puis bloque sur le wait (à ce moment il relâche le mutex).
- **P** est ordonnancé, produit un item ($\text{count} \equiv 1$), signale la condition et relâche le mutex.
- Avant que **C1** ne soit réveillé, **C2** est ordonnancé et consomme l'item du buffer ($\text{count} \equiv 0$) et signale **cond**.

Solution 1

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    if (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    if (count == 0)
        C1 cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    C2 cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (**P**) et deux consommateurs (**C1**, **C2**) et le buffer vide ($\text{count} \equiv 0$).
- **C1** s'exécute, puis bloque sur le wait (à ce moment il relâche le mutex).
- **P** est ordonnancé, produit un item ($\text{count} \equiv 1$), signale la condition et relâche le mutex.
- Avant que **C1** ne soit réveillé, **C2** est ordonnancé et consomme l'item du buffer ($\text{count} \equiv 0$) et signale **cond**.

Solution 1

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    if (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    if (count == 0)
        C1 cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    C2 mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (**P**) et deux consommateurs (**C1**, **C2**) et le buffer vide ($\text{count} \equiv 0$).
- **C1** s'exécute, puis bloque sur le wait (à ce moment il relâche le mutex).
- **P** est ordonnancé, produit un item ($\text{count} \equiv 1$), signale la condition et relâche le mutex.
- Avant que **C1** ne soit réveillé, **C2** est ordonnancé et consomme l'item du buffer ($\text{count} \equiv 0$) et signale **cond**.

Solution 1

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    if (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    if (count == 0)
        C1 cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    C2 return item;
}
```

- Soit un producteur (**P**) et deux consommateurs (**C1**, **C2**) et le buffer vide ($\text{count} \equiv 0$).
- **C1** s'exécute, puis bloque sur le wait (à ce moment il relâche le mutex).
- **P** est ordonnancé, produit un item ($\text{count} \equiv 1$), signale la condition et relâche le mutex.
- Avant que **C1** ne soit réveillé, **C2** est ordonnancé et consomme l'item du buffer ($\text{count} \equiv 0$) et signale **cond**.

Solution 1

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    if (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    if (count == 0)
        C1 cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (P) et deux consommateurs (C1, C2) et le buffer vide ($\text{count} \equiv 0$).
- C1 s'exécute, puis bloque sur le wait (à ce moment il relâche le mutex).
- P est ordonnancé, produit un item ($\text{count} \equiv 1$), signale la condition et relâche le mutex.
- Avant que C1 ne soit réveillé, C2 est ordonnancé et consomme l'item du buffer ($\text{count} \equiv 0$) et signale cond.
- C1 est réveillé et retire un item du buffer alors que celui-ci est vide !

Solution 1

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
P  mutex_lock(&mutex);
    if (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    if (count == 0)
        cond_wait(&cond, &mutex);
C1 item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

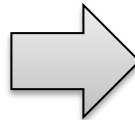
- Soit un producteur (P) et deux consommateurs (C1, C2) et le buffer vide ($\text{count} \equiv 0$).
- C1 s'exécute, puis bloque sur le wait (à ce moment il relâche le mutex).
- P est ordonnancé, produit un item ($\text{count} \equiv 1$), signale la condition et relâche le mutex.
- Avant que C1 ne soit réveillé, C2 est ordonnancé et consomme l'item du buffer ($\text{count} \equiv 0$) et signale cond.
- C1 est réveillé et retire un item du buffer alors que celui-ci est vide !



Solution 1

- Le problème précédent vient du fait que l'état du buffer a changé entre le moment où un thread est signalé et le moment où il s'exécute !
- Signaler un thread ne fait **que le réveiller** ; il n'y a aucune garantie que l'état soit toujours le même au moment où le thread est exécuté.
- Pour résoudre ce problème, il est nécessaire de **retester** la condition avant de poursuivre.
- Cela revient simplement à remplacer le `if` par un `while` dans le test de la condition :

```
if (count == 1)
    cond_wait(&cond, &mutex);
```



```
while (count == 1)
    cond_wait(&cond, &mutex);
```

Toujours tester chaque condition à l'intérieur d'une boucle !

Solution 2

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    if (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    if (count == 0)
        cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- En remplaçant les `if` par des `while`, nous obtenons le code suivant...

Solution 2

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    while (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    while (count == 0)
        cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- En remplaçant les `if` par des `while`, nous obtenons le code suivant... ce qui permet de retester l'état de la condition au réveil et de se rendormir si celle-ci a changé.

Solution 2

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    while (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    while (count == 0)
        cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- En remplaçant les `if` par des `while`, nous obtenons le code suivant... ce qui permet de retester l'état de la condition au réveil et de se rendormir si celle-ci a changé.
- Est-ce que ce code est exempt de bug ?

Solution 2

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    while (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    while (count == 0)
        cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (P) et deux consommateurs (C1, C2) et le buffer vide (count \equiv 0).

Solution 2

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    while (count == 1)
        cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    while (count == 0)
        C1 C2 cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (P) et deux consommateurs (C1, C2) et le buffer vide (count \equiv 0).
- C1 et C2 s'exécutent et bloquent tous les deux sur le wait.

Solution 2

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    while (count == 1)
        P      cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    while (count == 0)
        C1 C2  cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (P) et deux consommateurs (C1, C2) et le buffer vide (count \equiv 0).
- C1 et C2 s'exécutent et bloquent tous les deux sur le wait.
- P est ordonnancé, produit un item (count \equiv 1) et signale la condition; P veut encore produire, mais bloque sur le wait.

Solution 2

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    while (count == 1)
        P      cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    while (count == 0)
        C1 C2  cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (**P**) et deux consommateurs (**C1**, **C2**) et le buffer vide ($\text{count} \equiv 0$).
- **C1** et **C2** s'exécutent et bloquent tous les deux sur le wait.
- **P** est ordonnancé, produit un item ($\text{count} \equiv 1$) et signale la condition; **P** veut encore produire, mais bloque sur le wait.
- **C1** est ordonnancé, re-teste la condition, voit le buffer plein et consomme l'item; **C1** signale que le buffer est vide ($\text{count} \equiv 0$), veut encore consommer et se bloque sur le wait.

Solution 2

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    while (count == 1)
        P      cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    C1 while (count == 0)
        C2      cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (**P**) et deux consommateurs (**C1**, **C2**) et le buffer vide ($\text{count} \equiv 0$).
- **C1** et **C2** s'exécutent et bloquent tous les deux sur le wait.
- **P** est ordonnancé, produit un item ($\text{count} \equiv 1$) et signale la condition; **P** veut encore produire, mais bloque sur le wait.
- **C1** est ordonnancé, re-teste la condition, voit le buffer plein et consomme l'item; **C1** signale que le buffer est vide ($\text{count} \equiv 0$), veut encore consommer et se bloque sur le wait.

Solution 2

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    while (count == 1)
        P      cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    while (count == 0)
        C2      cond_wait(&cond, &mutex);
    C1 item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (**P**) et deux consommateurs (**C1**, **C2**) et le buffer vide ($\text{count} \equiv 0$).
- **C1** et **C2** s'exécutent et bloquent tous les deux sur le wait.
- **P** est ordonnancé, produit un item ($\text{count} \equiv 1$) et signale la condition; **P** veut encore produire, mais bloque sur le wait.
- **C1** est ordonnancé, re-teste la condition, voit le buffer plein et consomme l'item; **C1** signale que le buffer est vide ($\text{count} \equiv 0$), veut encore consommer et se bloque sur le wait.

Solution 2

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    while (count == 1)
        P      cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    while (count == 0)
        C2      cond_wait(&cond, &mutex);
    item_t item = buffer;
    C1      count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (P) et deux consommateurs (C1, C2) et le buffer vide (count \equiv 0).
- C1 et C2 s'exécutent et bloquent tous les deux sur le wait.
- P est ordonnancé, produit un item (count \equiv 1) et signale la condition; P veut encore produire, mais bloque sur le wait.
- C1 est ordonnancé, re-teste la condition, voit le buffer plein et consomme l'item; C1 signale que le buffer est vide (count \equiv 0), veut encore consommer et se bloque sur le wait.

Solution 2

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    while (count == 1)
        P      cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    while (count == 0)
        C2      cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    C1      cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (P) et deux consommateurs (C1, C2) et le buffer vide (count \equiv 0).
- C1 et C2 s'exécutent et bloquent tous les deux sur le wait.
- P est ordonnancé, produit un item (count \equiv 1) et signale la condition; P veut encore produire, mais bloque sur le wait.
- C1 est ordonnancé, re-teste la condition, voit le buffer plein et consomme l'item; C1 signale que le buffer est vide (count \equiv 0), veut encore consommer et se bloque sur le wait.

Solution 2

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    while (count == 1)
        P      cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    while (count == 0)
        C2     cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    C1 return item;
}
```

- Soit un producteur (P) et deux consommateurs (C1, C2) et le buffer vide (count \equiv 0).
- C1 et C2 s'exécutent et bloquent tous les deux sur le wait.
- P est ordonnancé, produit un item (count \equiv 1) et signale la condition; P veut encore produire, mais bloque sur le wait.
- C1 est ordonnancé, re-teste la condition, voit le buffer plein et consomme l'item; C1 signale que le buffer est vide (count \equiv 0), veut encore consommer et se bloque sur le wait.

Solution 2

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    while (count == 1)
        P      cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    C1 mutex_lock(&mutex);
    while (count == 0)
        C2      cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (**P**) et deux consommateurs (**C1**, **C2**) et le buffer vide ($\text{count} \equiv 0$).
- **C1** et **C2** s'exécutent et bloquent tous les deux sur le wait.
- **P** est ordonnancé, produit un item ($\text{count} \equiv 1$) et signale la condition; **P** veut encore produire, mais bloque sur le wait.
- **C1** est ordonnancé, re-teste la condition, voit le buffer plein et consomme l'item; **C1** signale que le buffer est vide ($\text{count} \equiv 0$), veut encore consommer et se bloque sur le wait.

Solution 2

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    while (count == 1)
        P      cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    C1 while (count == 0)
        C2      cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (P) et deux consommateurs (C1, C2) et le buffer vide (count \equiv 0).
- C1 et C2 s'exécutent et bloquent tous les deux sur le wait.
- P est ordonnancé, produit un item (count \equiv 1) et signale la condition; P veut encore produire, mais bloque sur le wait.
- C1 est ordonnancé, re-teste la condition, voit le buffer plein et consomme l'item; C1 signale que le buffer est vide (count \equiv 0), veut encore consommer et se bloque sur le wait.

Solution 2

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    while (count == 1)
        P      cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    while (count == 0)
        C1 C2  cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (P) et deux consommateurs (C1, C2) et le buffer vide (count \equiv 0).
- C1 et C2 s'exécutent et bloquent tous les deux sur le wait.
- P est ordonnancé, produit un item (count \equiv 1) et signale la condition; P veut encore produire, mais bloque sur le wait.
- C1 est ordonnancé, re-teste la condition, voit le buffer plein et consomme l'item; C1 signale que le buffer est vide (count \equiv 0), veut encore consommer et se bloque sur le wait.

Solution 2

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    while (count == 1)
        P      cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    while (count == 0)
        C1 C2  cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (P) et deux consommateurs (C1, C2) et le buffer vide (count \equiv 0).
- C1 et C2 s'exécutent et bloquent tous les deux sur le wait.
- P est ordonnancé, produit un item (count \equiv 1) et signale la condition; P veut encore produire, mais bloque sur le wait.
- C1 est ordonnancé, re-teste la condition, voit le buffer plein et consomme l'item; C1 signale que le buffer est vide (count \equiv 0), veut encore consommer et se bloque sur le wait.
- Quel thread devrait être ordonnancé ? On désire P, mais ce n'est pas garanti (dépend de l'ordonnanceur)...

Solution 2

```
item_t buffer;
int count = 0;
cond_t cond;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    while (count == 1)
        P      cond_wait(&cond, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&cond);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    while (count == 0)
        C1 C2  cond_wait(&cond, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&cond);
    mutex_unlock(&mutex);
    return item;
}
```

- Soit un producteur (P) et deux consommateurs (C1, C2) et le buffer vide (count \equiv 0).
- C1 et C2 s'exécutent et bloquent tous les deux sur le wait.
- P est ordonnancé, produit un item (count \equiv 1) et signale la condition; P veut encore produire, mais bloque sur le wait.
- C1 est ordonnancé, re-teste la condition, voit le buffer plein et consomme l'item; C1 signale que le buffer est vide (count \equiv 0), veut encore consommer et se bloque sur le wait.
- Quel thread devrait être ordonnancé ? On désire P, mais ce n'est pas garanti (dépend de l'ordonnanceur)...
- Si c'est C2, qui consomme le signal, que se passe-t-il ?

Solution 2

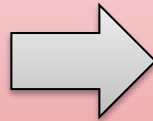
```
item_t buffer;  
int count = 0;  
cond_t cond;  
mutex_t mutex;
```

```
void put_item(item_t item) {  
    mutex_lock(&mutex);  
    while (count == 0) {  
        P      cond_wait(&cond, &mutex);  
        buffer = item;  
        count = 1;  
        cond_signal(&cond);  
        mutex_unlock(&mutex);  
    }
```

```
item_t get_item() {  
    mutex_lock(&mutex);  
    while (count == 0)
```

```
    C1 C2    cond_wait(&cond, &mutex);  
    item_t item = buffer;  
    count = 0;  
    cond_signal(&cond);  
    mutex_unlock(&mutex);  
    return item;  
}
```

Voyant que `count == 0`, **C2** va donc bloquer sur le `wait`.



Deadlock !

- Soit un producteur (**P**) et deux consommateurs (**C1**, **C2**) et le buffer vide (`count == 0`).
- **C1** et **C2** s'exécutent et bloquent tous les deux sur le `wait`.

...é, produit un item (`count == 1`); **P** veut consommer, mais bloque sur le `wait`.

...ncé, re-teste la condition; **P** voit que le buffer est plein et ne consomme pas; **C1** signale que le buffer est vide (`count == 0`), veut encore consommer et se bloque sur le `wait`.

- Quel thread devrait être ordonnancé ? On désire **P**, mais ce n'est pas garanti (dépend de l'ordonnanceur)...
- Si c'est **C2**, que se passe-t-il ?

Solution 2

- Le problème de la solution précédente vient du fait que **n'importe quel thread peut être réveillé** lors d'un signal et si ce n'est pas le bon, un deadlock peut se produire.
- Un thread producteur devrait seulement signaler un thread consommateur.
- Inversément, un thread consommateur devrait seulement signaler un thread producteur.
- Solution ?

Solution 2

- Le problème de la solution précédente vient du fait que **n'importe quel thread peut être réveillé** lors d'un signal et si ce n'est pas le bon, un deadlock peut se produire.
- Un thread producteur devrait seulement signaler un thread consommateur.
- Inversément, un thread consommateur devrait seulement signaler un thread producteur.
- Solution ?

➤ **Utiliser deux variables de condition au lieu d'une seule !**

Solution 3 (fonctionnelle!)

```
item_t buffer;
int count = 0;
cond_t empty, full;
mutex_t mutex;

void put_item(item_t item) {
    mutex_lock(&mutex);
    while (count == 1)
        cond_wait(&empty, &mutex);
    buffer = item;
    count = 1;
    cond_signal(&full);
    mutex_unlock(&mutex);
}

item_t get_item() {
    mutex_lock(&mutex);
    while (count == 0)
        cond_wait(&full, &mutex);
    item_t item = buffer;
    count = 0;
    cond_signal(&empty);
    mutex_unlock(&mutex);
    return item;
}
```

- Utilisation de 2 variables de condition différentes pour indiquer des actions différentes
- Boucles sur les tests des variables

Utilisation des VCs

- Le système du producteur-consommateur a été donné à titre d'exemple dans ce cours, mais comme nous l'avons vu, ce problème peut aussi être implémenté efficacement avec des sémaphores (et mutex).
- Les VCs sont pratiques lorsque plusieurs tâches ont besoin de tester une ou plusieurs variables partagées et qu'elles sont susceptibles de réagir à son/leurs changement(s).
- Ainsi lorsque la variable est écrite, la fonction `pthread_cond_broadcast()` peut être utilisée indiquant à toutes les tâches bloquées de se débloquenter et de re-tester la nouvelle valeur. Il n'y a pas de primitive équivalente avec des sémaphores.

```
int g1=0, g2=1;
```

```
void t1() {  
    mutex_lock;  
    g1++;  
    g2 ^= 1;  
    cond_signal_broadcast(&vc);  
    mutex_unlock;  
}
```

```
...  
while (g1>5 && g2)  
    cond_wait(&vc);  
...
```

■ ■ ■

```
...  
while (!g2)  
    cond_wait(&vc);  
...
```