

# Programmation Système

## Les processus légers

Emmanuel Bouthenot (emmanuel.bouthenot@u-bordeaux.fr)

Licence professionnelle ADSILLH - Université de Bordeaux

2018-2019

# Processus et processus légers

## Points communs :

- Ils permettent aux programmes de faire des tâches concurrentes (réception de données par le réseau pendant que l'interface graphique est mise à jour)
- Ils permettent de une exécution de tâches en parallèle si l'architecture le permet (système multiprocesseurs ou multicœurs)

## Différences :

- Les processus sont exécutés dans un espace mémoire qui leur est propre
- Les processus légers (**threads**) partage le même espace mémoire

# Avantages des processus légers

- Le partage d'informations entre processus requiert plus de travail (mise en place d'IPC).
- Créer et rendre un processus léger opérationnel par le noyau est de l'ordre d'un facteur 10
- Le partage d'informations entre processus légers existe de fait car il partage le même espace mémoire (variables globales et allouées sur le tas)

# Anatomie d'un processus en mémoire (schéma)

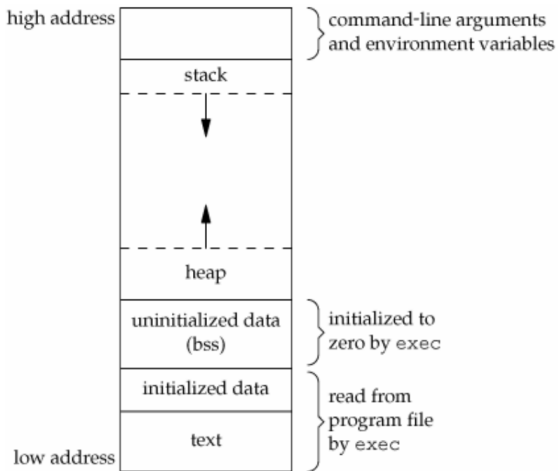


Figure: **APUE** 7.6

# Anatomie d'un processus léger en mémoire (schéma)

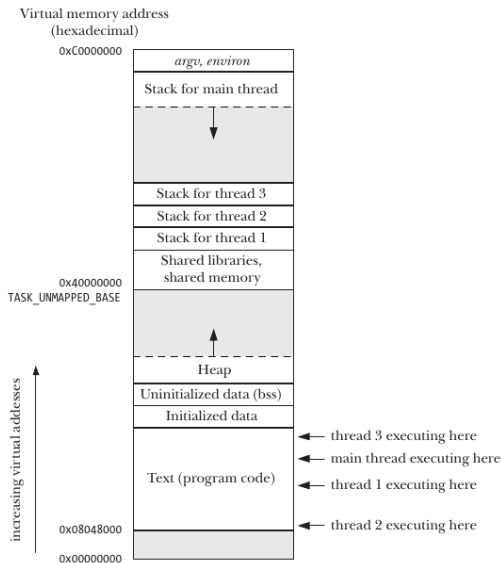


Figure: TLPI 29-1

Les processus légers partagent :

- Leur espace mémoire
- ID de processus (pid), processus parent (ppid), etc.
- La table des fichiers ouverts
- Les gestionnaires de signaux

Les processus légers ne partagent pas :

- Leur ID de processus léger (**Thread ID**)
- Leur pile (**stack**) (ie: pas de partage des variables locales lors d'appels de fonctions)
- **errno**
- La mémoire locale de processus léger (**Thread Local Storage (TLS)**)

# POSIX Thread API (Pthreads)

L'API Pthreads est issue de POSIX.1c (1995) et intégrée dans SUSv3. Elle est devenue l'API des processus légers standard sous UNIX.

Elle couvre :

- Les types de données
- La gestion des processus légers (création, annulation, attente, etc.)
- Exclusion mutuelle
- Variables conditionnelles
- Verrous, etc.

Cependant c'est une API fournie et complexe. (man 7 pthreads)

Convention habituelle :

- Succès: retourne 0 (ou un entier positif selon l'usage de l'appel)
- Échec: retourne -1 et errno est positionnée

Convention de l'API Pthreads :

- Succès: retourne 0
- Échec: errno est positionnée



## **-pthread** (ou -pthreads)

Add support for multithreading using the POSIX threads library. This option sets flags for both the preprocessor and linker. This option does not affect the thread safety of object code produced by the compiler or that of libraries supplied with it.

```
> gcc -Wall -pthread hello-threaded.c -o hello-threaded
```

# Création d'un processus léger

Par défaut, le processus est considéré comme le processus léger principal, on peut ensuite ajouter des processus légers supplémentaires :

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                  void *(*start_routine) (void *), void *arg);

// Retourne 0 en cas de succès.
// En cas d'erreur, errno est positionnée
```

- **start** pointe vers la fonction appelé au départ du processus léger
- **arg** sera passé en argument à **start** : `start(arg)`
- **thread** sera rempli par l'ID de processus léger (identifiant unique du thread)
- **attr** peut être utilisé pour définir les attributs du processus léger nouvellement créé

# Terminaison d'un processus léger

Un processus léger se termine quand :

- Sa fonction principale d'appel se termine
- Le processus léger est annulé (avec `pthread_cancel()`)
- L'un des processus légers appelle `exit` (ce qui signifie la terminaison de tous les processus légers)
- Le processus léger appelle `pthread_exit()`

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

```
void pthread_exit(void *retval);
```

```
// Ne retourne rien à l'appelant
```

```
// retval sera accessible aux autres
```

```
// threads en attente de sa terminaison
```

# Attente de la fin d'un autre processus léger

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);

// Retourne 0 en cas de succès
// En cas d'erreur, errno est positionnée
```

- Le processus léger appelant sera bloqué jusqu'à ce que le processus léger attendu se termine.
- **retval** sera initialisée à la valeur de retour du processus léger.
- N'importe quel processus léger peut attendre un autre processus léger.
- Il n'y a pas de possibilité d'attendre n'importe quel processus léger.

# Hello World - Exemple

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

void *tmain(void *arg) {
    char *msg = (char *) arg;
    printf("[T] %s", msg) ;
    pthread_exit((void *) strlen(msg));
}

int main(int argc, char **argv) {
    pthread_t t;
    int r;
    void *res;
    printf("Creating thread...\n");
    r = pthread_create(&t, NULL, tmain, (void *) "Hello World!\n");
    if (r != 0) {
        perror("Unable to create thread");
        exit(EXIT_FAILURE);
    }
    r = pthread_join(t, &res);
    if (r != 0) {
        perror("Unable to join thread");
    }
}
```

# Hello World - Exemple (suite)

```
        exit(EXIT_FAILURE);
    }
    printf("Thread return: %ld\n", (long) res);
    exit(EXIT_SUCCESS);
}
```

```
> gcc -Wall hello-pthread.c -o hello-pthread
> /tmp/ccIg0fDC.o: In function `main':
> hello-pthread.c:(.text+0x6e): undefined reference to `pthread_create'
> hello-pthread.c:(.text+0x9e): undefined reference to `pthread_join'
> collect2: error: ld returned 1 exit status
> gcc -pthread -Wall hello-pthread.c -o hello-pthread
> ./hello-pthread
> Creating thread...
> [T] Hello World!
> Thread return: 13
```

# Hello World - multi threaded - Exemple

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

void *tmain(void *arg) {
    long no = (long) arg;
    printf("[T%ld] Hello world!\n", no);
    return (void *) no;
}

int main(int argc, char **argv) {
    pthread_t *tlist;
    int count;
    void *res;

    if (argc != 2) {
        fprintf(stderr, "Wrong arguments\n");
        exit(EXIT_FAILURE);
    }
    count = atoi(argv[1]);
    if ((tlist = calloc(count, sizeof(pthread_t))) == NULL) {
        perror("Memory allocation failed");
        exit(EXIT_FAILURE);
    }
}
```

# Hello World - multi threaded - Exemple (suite)

```
}  
for(long i=0; i<count; i++) {  
    printf("Creating thread no %ld\n", i);  
    if (pthread_create(&tlist[i], NULL, tmain, (void *) i) != 0) {  
        perror("Unable to create thread");  
    }  
}  
for(long j=0; j<count; j++) {  
    printf("Trying to join thread no %ld\n", j);  
    if (pthread_join(tlist[j], &res) != 0) {  
        perror("Unable to join thread");  
    }  
    printf("Thread no %ld terminated with status: %ld\n",  
          j, (long) (long *) res);  
}  
free(tlist);  
exit(EXIT_SUCCESS);  
}
```



# Hello World - multi threaded - Démo

```
> gcc -pthread -Wall hello-pthreads.c -o hello-pthreads
> ./hello-pthreads 4
> Creating thread no 0
> Creating thread no 1
> [T0] Hello world!
> Creating thread no 2
> [T1] Hello world!
> Creating thread no 3
> [T2] Hello world!
> Trying to join thread no 0
> Thread no 0 terminated with status: 0
> Trying to join thread no 1
> Thread no 1 terminated with status: 1
> Trying to join thread no 2
> Thread no 2 terminated with status: 2
> Trying to join thread no 3
> [T3] Hello world!
> Thread no 3 terminated with status: 3
```

## Définition

A l'instar des processus "normaux", les processus léger dit *zombies* sont des processus léger qui ont terminés et dont aucun autre processus léger n'a pris connaissance.

Les processus légers zombies posent les même problème que le processus normaux en terme de consommation mémoire.

Sur certains UNIX, le nombre de processus légers par processus est limité à une valeur définie au niveau du système.

```
> cat /proc/sys/kernel/threads-max  
63399
```

# Détacher un processus léger

```
#include <pthread.h>

int pthread_detach(pthread_t thread);

// Retourne 0 en cas de succès.
// En cas d'erreur, errno est positionnée
```

Les processus légers détachés (non joignable) sont automatiquement pris en compte à leur terminaison.

En cas d'utilisation de `exit`, tous les processus légers (détachés ou non) sont terminés de la même façon.

# Identité d'un processus léger

```
#include <pthread.h>

pthread_t pthread_self(void);

// Retourne l'identifiant du thread appelant
// Cette fonction réussit toujours

int pthread_equal(pthread_t t1, pthread_t t2);

// Retourne une valeur non nulle si les 2
// identifiants sont égaux
// Retourne 0 dans le cas contraire
```

Les identifiants de processus légers doivent être considérés comme étant opaques. Il n'existe pas de façon portable de comparer directement deux valeurs de `pthread_t`.

## Section critique

Une section critique est un fragment de code qui accède à une ressource partagée et qui doit être exécutée de façon atomique au regard des autres entités (**threads**) qui souhaite y accéder.

Dans le cas des **threads** l'accès à une variable globale est une section critique.

# Sections critiques - exemple

On souhaite incrémenter une variable globale  
`static long glob = 0;` depuis deux (ou plus) threads qui  
s'exécute simultanément.

```
void *tmain(void *arg) {  
    long loc;  
  
    loc = glob;  
    loc++;  
    glob = loc;  
}
```

# Section critique - exemple (suite)

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

#define LOOP_COUNT 100000000
#define THREADS_COUNT 2

static long glob = 0;

void *tmain(void *arg) {
    long loc;
    for(long i = 0; i < LOOP_COUNT; i++) {
        loc = glob;
        loc++;
        glob = loc;
    }
    return NULL;
}

int main(int argc, char **argv) {
    pthread_t tlist[THREADS_COUNT];
    int r;
    void *res;
```

# Section critique - exemple (suite)

```
for(int i=0; i<THREADS_COUNT; i++) {
    printf("Creating thread %d...\n", i);
    r = pthread_create(&tlist[i], NULL, tmain, NULL);
    if (r != 0) {
        perror("Unable to create thread");
        exit(EXIT_FAILURE);
    }
    printf("Thread %d running...\n", i);
}
for(int i=0; i<THREADS_COUNT; i++) {
    printf("Joining thread %d...\n", i);
    r = pthread_join(tlist[i], &res);
    if (r != 0) {
        perror("Unable to join thread");
        exit(EXIT_FAILURE);
    }
    printf("Thread %d terminated...\n", i);
}
printf("%ld =? %ld\n", glob, (long) THREADS_COUNT * (long) LOOP_COUNT);
exit(EXIT_SUCCESS);
}
```



# Section critique - Démo

```
> gcc -Wall -pthread critical-section-pthread.c -o critical-section
> ./critical-section-pthread
Creating thread 0...
Thread 0 running...
Creating thread 1...
Thread 1 running...
Joining thread 0...
Thread 0 terminated...
Joining thread 1...
Thread 1 terminated...
100954120 =? 200000000
```

# Section critique - Schéma

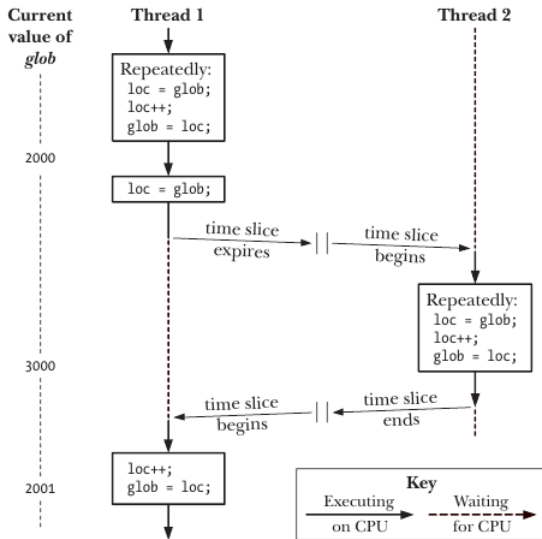


Figure: TLPI 30-1

# Mutex

Pour protéger les sections critiques, il faut un mécanisme d'exclusion mutuelle.

L'API des POSIX threads fournit les **mutex** comme mécanisme de de synchronisation entre threads pour protéger les sections critiques.

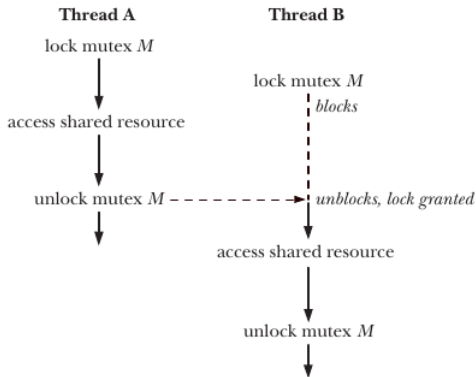


Figure: **TLPI 30-2**

# Création de mutex

La création de mutex peut de faire part une déclaration statique :

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  
  
// La création statique sous entend l'utilisation d'options  
// par défaut pour le mutex créé
```

ou dynamiquement, alloué sur le tas (malloc).

```
int pthread_mutex_init(pthread_mutex_t *mutex,  
                       const pthread_mutexattr_t *mutexattr);  
  
// Retourne 0 en cas de succès.  
// En cas d'erreur, errno est positionnée
```

**mutex** est ici une zone mémoire alloué sur le tas ou une zone mémoire allouée sur la pile.

# Destruction de mutex

Les mutex alloués dynamiquement doivent être détruits lorsqu'ils n'ont plus d'utilité.

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);  
  
// Retourne 0 en cas de succès.  
// En cas d'erreur, errno est positionnée
```

- Un mutex ne doit être détruit que lorsque qu'il n'est plus verrouillé
- Un mutex ne doit être détruit que lorsque qu'on est sûr qu'il ne sera plus utilisé

# Interblocages (deadlocks)

## Définition

L'interblocage (**deadlock**) se produit, par exemple, lorsqu'un thread T1 ayant déjà acquis la ressource R1 demande l'accès à une ressource R2, pendant que le thread T2, ayant déjà acquis la ressource R2, demande l'accès à la ressource R1.

Chacun des deux threads attend alors la libération de la ressource possédée par l'autre.

La situation est donc bloquée.

# Catégories de mutex

Les cas limites :

- ① Un thread qui tente de déverrouiller un mutex un seconde fois
- ② Un thread qui tente de déverrouiller un mutex qui n'est pas verrouillé
- ③ Un thread qui tente de déverrouiller un mutex qu'il n'a pas lui même verrouillé

Les catégories de mutex :

- PTHREAD\_MUTEX\_NORMAL : cas **1** = **deadlock**
- PTHREAD\_MUTEX\_ERRORCHECK : cas **1,2,3** = **error checking** sur toutes les opérations (plus lent)
- PTHREAD\_MUTEX\_RECURSIVE : **reference counting**, cas **1** = incrémentation du compteur, cas **2** = décrémentation du compteur (déverrouillage quand le compteur est à 0, erreur si inférieur), cas **3** = erreur

# Vérouillage / Dévérouillage de mutex

```
int pthread_mutex_lock(pthread_mutex_t *mutex);  
  
int pthread_mutex_unlock(pthread_mutex_t *mutex);  
  
int pthread_mutex_trylock(pthread_mutex_t *mutex);  
  
// Retourne 0 en cas de succès.  
// En cas d'erreur, errno est positionnée
```

`pthread_mutex_trylock` est la version non bloquante de `pthread_mutex_lock`

```
> man 3 pthread_mutex_lock  
> man 3 pthread_mutexattr_settype
```



# Section critique avec mutex - exemple

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

#define LOOP_COUNT 100000000
#define THREADS_COUNT 2

static long glob = 0;
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *tmain(void *arg) {
    long loc;
    for(long i = 0; i < LOOP_COUNT; i++) {
        if (pthread_mutex_lock(&mutex) != 0) {
            perror("Mutex lock failed");
            exit(EXIT_FAILURE);
        }
        loc = glob;
        loc++;
        glob = loc;
        if (pthread_mutex_unlock(&mutex) != 0) {
            perror("Mutex unlock failed");
            exit(EXIT_FAILURE);
        }
    }
}
```

## Section critique - exemple (suite)

```
    }  
  }  
  return NULL;  
}  
  
int main(int argc, char **argv) {  
  pthread_t tlist[THREADS_COUNT];  
  int r;  
  void *res;  
  
  for(int i=0; i<THREADS_COUNT; i++) {  
    printf("Creating thread %d...\n", i);  
    r = pthread_create(&tlist[i], NULL, tmain, NULL);  
    if (r != 0) {  
      perror("Unable to create thread");  
      exit(EXIT_FAILURE);  
    }  
    printf("Thread %d running...\n", i);  
  }  
  for(int i=0; i<THREADS_COUNT; i++) {  
    printf("Joining thread %d...\n", i);  
    r = pthread_join(tlist[i], &res);  
    if (r != 0) {  
      perror("Unable to join thread");  
    }  
  }  
}
```

# Section critique - exemple (fin), Démo

```
        exit(EXIT_FAILURE);
    }
    printf("Thread %d terminated...\n", i);
}
printf("%ld =? %ld\n", glob, (long) THREADS_COUNT * (long) LOOP_COUNT);
exit(EXIT_SUCCESS);
}
```

```
> gcc -Wall -pthread critical-section-pthread-mutex.c -o critical-s
> ./critical-section-pthread-mutex
Creating thread 0...
Thread 0 running...
Creating thread 1...
Thread 1 running...
Joining thread 0...
Thread 0 terminated...
Joining thread 1...
Thread 1 terminated...
200000000 =? 200000000
```

- Un **mutex** empêche l'accès à une section critique par de multiples threads en même temps
- Une **variable "condition"** permet a un thread d'informer ses pairs du changement d'une ressource partagée. Elle permet aux autres threads de se mettre en attente jusqu'à être informé d'un changement.
- Ce mécanisme contrecarre une utilisation excessive du CPU par l'utilisation de boucle d'attente de ressources à être traitées

# Les variables conditions (suite)

- Les **variables "conditions"** sont utilisées en conjonction avec des mutex
- 2 opérations sur les variables conditions :
  - **wait** : bloque jusqu'à ce qu'une notification arrive
  - **signal** : envoie une notification
- Les **variables "conditions"** sont sans conservations d'état (**stateless**) : si une notification est envoyé mais qu'aucun thread n'était en attente, elle est perdue.

# Création de variables conditions

La création de variables conditions peut de faire part une déclaration statique :

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER ;
```

ou dynamiquement, alloué sur le tas (malloc).

```
int pthread_cond_init(pthread_cond_t *cond,  
pthread_condattr_t *cond_attr);
```

```
// Retourne 0 en cas de succès.
```

```
// En cas d'erreur, errno est positionnée
```

# Action sur les variables conditions (1)

```
int pthread_cond_wait(pthread_cond_t *cond,  
                      pthread_mutex_t *mutex);  
  
// Retourne 0 en cas de succès.  
// En cas d'erreur, errno est positionnée
```

- **pthread\_cond\_wait()** déverrouille atomiquement le mutex et attend que la variable condition **cond** soit signalée
- L'exécution du thread est suspendue jusqu'à ce que la variable condition soit signalée
- Le mutex doit être verrouillé par le thread appelant à l'entrée de **pthread\_cond\_wait()**
- Avant de rendre la main au thread appelant, **pthread\_cond\_wait()** reverrouille mutex (comme **pthread\_lock\_mutex()**)

# Action sur les variables conditions (1)

```
int pthread_cond_signal(pthread_cond_t *cond);  
  
int pthread_cond_broadcast(pthread_cond_t *cond);  
  
// Retourne 0 en cas de succès.  
// En cas d'erreur, errno est positionnée
```

- **pthread\_cond\_signal()** relance l'un des threads attendant la variable condition **cond**. S'il n'existe aucun thread répondant à ce critère, rien ne se produit. Si plusieurs threads attendent sur **cond**, seul l'un d'entre eux sera relancé, mais il est impossible de savoir lequel
- **pthread\_cond\_broadcast()** relance tous les threads attendant sur la variable condition **cond**. Rien ne se passe s'il n'y a aucun thread attendant sur **cond**



# Détruire une variable condition

Les variables conditions allouées dynamiquement doivent être détruits lorsqu'ils n'ont plus d'utilité.

```
int pthread_cond_destroy(pthread_cond_t *cond);  
  
// Retourne 0 en cas de succès.  
// En cas d'erreur, errno est positionnée
```

- Une variable condition ne doit être détruite que lorsque aucun threads n'est en attente sur cette même condition

# Les variables conditions - exemple (1)

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

static long goods = 0;
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t avail = PTHREAD_COND_INITIALIZER;

void *producer(void *arg) {
    while(1) {
        sleep(1); // simulate time to produce
        if (pthread_mutex_lock(&mutex) != 0) {
            perror("Mutex lock failed");
            exit(EXIT_FAILURE);
        }
        goods++;
        if (pthread_mutex_unlock(&mutex) != 0) {
            perror("Mutex unlock failed");
            exit(EXIT_FAILURE);
        }
        if (pthread_cond_signal(&avail) != 0) {
            perror("Condition signal failed");
            exit(EXIT_FAILURE);
        }
    }
}
```

## Les variables conditions - exemple (2)

```
    }  
}  
return NULL;  
}  
  
void *consumer(void *arg) {  
    while(1) {  
        if (pthread_mutex_lock(&mutex) != 0) {  
            perror("Mutex lock failed");  
            exit(EXIT_FAILURE);  
        }  
        while(goods == 0) {  
            if (pthread_cond_wait(&avail, &mutex) != 0) {  
                perror("Condition wait failed");  
                exit(EXIT_FAILURE);  
            }  
        }  
        while(goods > 0) {  
            goods--;  
            printf("Consuming...\n");  
        }  
        if (pthread_mutex_unlock(&mutex) != 0) {  
            perror("Mutex unlock failed");  
            exit(EXIT_FAILURE);  
        }  
    }  
}
```

# Les variables conditions - exemple (3)

```
    }  
  }  
  return NULL;  
}  
  
int main(int argc, char **argv) {  
  pthread_t t_producer, t_consumer;  
  
  printf("Creating producer thread...\n");  
  if (pthread_create(&t_producer, NULL, producer, NULL) != 0) {  
    perror("Unable to create producer thread");  
    exit(EXIT_FAILURE);  
  }  
  printf("Creating consumer thread...\n");  
  if (pthread_create(&t_consumer, NULL, consumer, NULL) != 0) {  
    perror("Unable to create consumer thread");  
    exit(EXIT_FAILURE);  
  }  
  
  if (pthread_join(t_producer, NULL) != 0) {  
    perror("Unable to join producer thread");  
    exit(EXIT_FAILURE);  
  }  
}
```

# Les variables conditions - exemple (4)



```
if (pthread_join(t_consumer, NULL) != 0) {  
    perror("Unable to join consumer thread");  
    exit(EXIT_FAILURE);  
}  
  
exit(EXIT_SUCCESS);  
}
```

```
>./conditional-variable-pthread  
Creating producer thread...  
Creating consumer thread...  
Consuming...  
Consuming...  
Consuming...  
Consuming...  
Consuming...  
Consuming...  
Consuming...  
Consuming...  
Consuming...  
Consuming...  
Consuming...  
^C
```

## Définition

On dit d'une fonction qu'elle est **Thread Safe** lorsqu'elle est capable de fonctionner correctement lorsqu'elle est exécutée simultanément au sein du même espace d'adressage par plusieurs threads.

Les threads ne devraient pas invoquer de fonctions non **Thread Safe** sans explicitement mettre en place un mécanisme de synchronisation entre eux (par exemple en utilisant des **mutex**).

-  **[APUE]** Advanced Programming in the UNIX Environment.  
W. Richard Stevens and Stephen A. Rago.  
*Addison-Wesley Professional*, 2005.
-  **[TLPI]** The Linux Programming Interface.  
Michael Kerisk.  
*No Starch Press*, 2010.