Développement Système

# Les Threads Posix

# Développement Système: les Threads Posix

***Sommaire:***

# Développement Système: les Threads Posix

## *Introduction :*

- Le concept de processus se traduit par deux caractéristiques:

**1/La propriété de ses ressources :**

- Un processus a un espace d'adressage virtuel qui reflète les attributs du PCB (data, texte, PID, PC,…)
- L'OS protège cet espace de toutes interférences extérieures

**2/Ordonnancement / exécution :**

- L'exécution d'un processus suit un chemin d'exécution (une trace)
- Un processus a un état ( Ready, Running,…)
- Une priorité
- Est ordonnancé par l'OS

Ces deux caractéristiques sont totalement indépendantes

# Développement Système: les Threads Posix

## ***Introduction :*** *Le multithreading*

***Le Multithreading :*** La capacité d'un système d'exploitation à avoir simultanément des chemins d'exécution multiples au sein d'un processus unique.

- Le mot ***thread*** peut se traduire par "fil d'exécution", c'est-à-dire un déroulement particulier du code du programme qui se produit parallèlement à d'autres entités en cours de progression.
- Les threads sont généralement présentés en premier lieu comme des processus allégés ne réclamant que peu de ressources pour les changements de contextes.
- Les différents threads d'une application partagent un même espace d'adressage en ce qui concerne leurs données.
- Chaque thread dispose personnellement d'une pile et d'un contexte d'exécution contenant les registres du processeur et un compteur d'instruction.

# Développement Système: les Threads Posix

## ***Introduction :*** *Le multithreading*

- Les méthodes de communication entre les threads sont alors naturellement plus simples que les communications entre processus.
- En contrepartie, l'accès concurrentiel aux mêmes données nécessite une synchronisation pour éviter les interférences, ce qui complique certaines portions de code
- Les threads ne sont intéressants que dans les applications assurant plusieurs tâches en parallèle.
- Si chacune des opérations effectuées par un logiciel doit attendre la fin d'une opération précédente avant de pouvoir démarrer, il est totalement inutile d'essayer une approche multithread.

# Les Threads: définition

# Développement Système: les Threads Posix

## Définition :

***Un thread est l'unité de base de l'utilisation du CPU***,

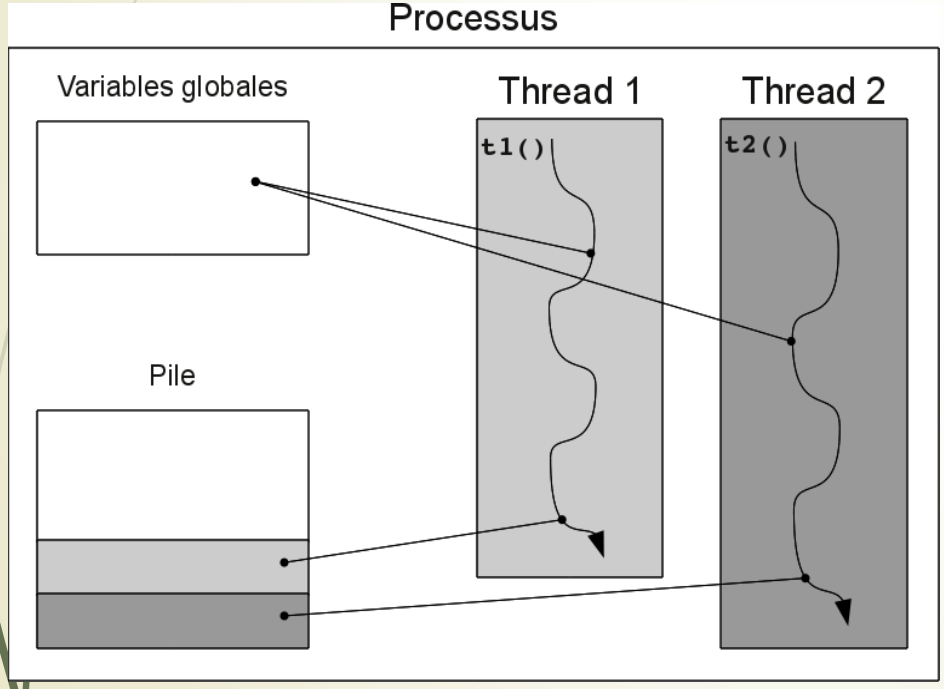
- il comporte un identifiant de thread,
- Un program counter
- Un ensemble de registres, et une pile.
- Il partage avec les autres threads appartenant au même processus
  - la section de code,
  - la section de données,
  - et d'autre ressources du système d'exploitation, ressources, telles que les fichiers ouverts et les signaux.
- Un processus traditionnel (poids lourd) a un seul thread. Si un processus a plusieurs threads, il peut effectuer plus d'une tâche à la fois

# Développement Système: les Threads Posix

## Définition : à retenir :

*Les threads permettent de dérouler plusieurs suites d'instructions en parallèle, à l'intérieur du même processus.*

*Un thread exécute une fonction.*




Un *thread* ressemble fortement à un processus fils classique à la différence qu'il est dans le même espace d'adressage et par conséquent partage beaucoup plus de données avec le processus qui l'a créé :

- ✓ Les variables globales,
- ✓ Les variables statiques locales,
- ✓ Les descripteurs de fichiers,

Mais il obtient sa propre pile et donc de ce fait ses propres variables locales.

# Développement Système: les Threads Posix

## *Définition :*

- **Avantages d'un thread :**
  - ***Multi-tâches moins couteux :*** il n'y a pas de changement de mémoire virtuelle, donc moins couteux que la commutation de contexte.
  - ***La communication entre threads est plus rapide et plus efficace :*** grâce au partage de certaines ressources entre threads, les IPC sont inutiles.
- **Inconvénients d'un thread :**
  - ***La programmation utilisant les threads est plus difficile à mettre en œuvre*** : obligation de mettre en place des mécanismes de synchronisation, risque élevé d'interblocage, d'endormissement …

# Les Threads: création

# Développement Système: les Threads Posix

## ***Les threads :*** *Création*

***int pthread_create(pthread_t * thread, const pthread_attr_t * attr,***

***void * (*start_routine)(void *), void * arg);***

- ***thread :*** mémorise l'identificateur du thread,
- ***attr :*** correspond aux attributs appliqués, *(NULL si attributs par défaut)*

*Les attributs permettent de spécifier la taille de la pile, la priorité, la politique de planification, etc. Il y a plusieurs formes de modification des attributs.*

- ***start_routine :*** fonction exécutée par le thread,
- ***arg :*** paramètres de la fonction start_routine

*La fonction **start_routine** est invoquée dès la création du thread et reçoit en argument le pointeur passé en dernière position dans **pthread_create**. Le type de l'argument étant **void ***, on pourra le transformer en n'importe quel type de pointeur pour passer un argument au thread.*

***La fonction pthread_create retourne 0 si le thread a été créé.***

# Développement Système: les Threads Posix

## ***Les threads :*** *Création*

- Lorsque la fonction principale d'un thread se termine, celui-ci est éliminé.
- Cette fonction doit renvoyer une valeur de type ***void** ** qui pourra être récupérée dans un autre fil d'exécution.
- Il est aussi possible d'invoquer la fonction ***pthread_exit( )***, qui met fin au thread courant tout en renvoyant le pointeur ***void** ** passé en argument.

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

- On ne doit naturellement pas invoquer ***exit( )***, qui mettrait fin à toute l'application et pas uniquement au thread appelant.
- Un processus qui se termine impose la terminaison de tous ses threads.

# Développement Système: les Threads Posix

## ***Les threads :*** *récupération de la valeur de retour*

- Pour récupérer la valeur de retour d'un thread terminé, on utilise la fonction ***pthread_join( )***

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

- *Cette fonction suspend l'exécution du thread appelant jusqu'à la terminaison du thread indiqué en argument.*
- *Elle remplit alors le pointeur passé en seconde position avec la valeur de retour du thread fini.*

# Développement Système: les Threads Posix

## ***Les threads :*** création, terminaison et récupération de valeur

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

void *fonction_thread(void *arg);

int main(int argc, char *argv[])
{
    pthread_t th1,th2;
    int retour;
    int valeur1,valeur2;
    valeur1 = 4;
    pthread_create(&th1,NULL,fonction_thread,(void *)&valeur1);
    valeur2 = 7;
    pthread_create(&th2,NULL,fonction_thread,(void *)&valeur2);
    pthread_join(th1,(void **)&retour);
    printf("Retour du thread1 : %d\n",retour);
    pthread_join(th2,(void **)&retour);
    printf("Retour du thread2 : %d\n",retour);
    return EXIT_SUCCESS;
}

void *fonction_thread(void *arg)
{
    int i;
    int max;
    max = *(int *)arg;
    for(i=0;i<max;i++)
    {
        printf("\t\tValeur de i : %d\n",i );
        sleep(1);
    }
    pthread_exit((void *)i);
}
```

- Exemple1:

```
nouzhatber@ServeurLinux:~/Documents/Threads$ ./Exemple1
                Valeur de i : 0
                Valeur de i : 0
                Valeur de i : 1
                Valeur de i : 1
                Valeur de i : 2
                Valeur de i : 2
                Valeur de i : 3
                Valeur de i : 3
                Valeur de i : 4
Retour du thread1 : 4
                Valeur de i : 5
                Valeur de i : 6
Retour du thread2 : 7
nouzhatber@ServeurLinux:~/Documents/Threads$
```

# Développement Système: les Threads Posix

## ***Les threads :*** création, terminaison et récupération de valeur

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
typedef struct {
    int val;
    int num;
}param;
void *fonction_thread(void *arg);
int main(int argc, char *argv[]){
    pthread_t th1,th2;
    int retour;
    param p1;
    param p2;
    p1.val=4;
    p1.num=1;
    printf("\n PID du processus créateur : %d \n",getpid());
    pthread_create(&th1,NULL,fonction_thread,(void *)&p1);
    p2.val=7;
    p2.num=2;
    pthread_create(&th2,NULL,fonction_thread,(void *)&p2);
    pthread_join(th1,(void **)&retour);
    printf("Retour du thread1 : %d\n",retour);
    pthread_join(th2,(void **)&retour);
    printf("Retour du thread2 : %d\n",retour);
    return EXIT_SUCCESS;
}
void *fonction_thread(void *arg){
    int i;
    int max;
    int n;
    param *Param = (param *)arg;
    n=Param->num;
    max=Param->val;
    printf("\n PID du thread%d : %d \n",n,getpid());
    for(i=0;i<max;i++)  {
        printf("\t\tThread %d: Valeur de i : %d\n",n,i );
        sleep(1);
    }
    pthread_exit((void *)i);
}
```

- Exemple2: pid de thread

```
pi@raspberrypi:~/Documents/ExemplesThreads $ ./ex2

 PID du processus créateur : 1754

 PID du thread1 : 1754
                Thread 1: Valeur de i : 0

 PID du thread2 : 1754
                Thread 2: Valeur de i : 0
                Thread 1: Valeur de i : 1
                Thread 2: Valeur de i : 1
                Thread 1: Valeur de i : 2
                Thread 2: Valeur de i : 2
                Thread 1: Valeur de i : 3
                Thread 2: Valeur de i : 3
                Thread 2: Valeur de i : 4
Retour du thread1 : 4
                Thread 2: Valeur de i : 5
                Thread 2: Valeur de i : 6
Retour du thread2 : 7
```

# Les Threads: détachement

# Développement Système: les Threads Posix

## ***Les threads :*** Détachement

- Quand on crée un thread avec des attributs par défaut, il est en mode "*JOINABLE*", ce mode oblige le thread créateur à attendre la fin du thread lancé. C'est l'appel de la fonction ***pthread_join*** qui permet de libérer les ressources mémoire du thread lancé.

- Si le processus créateur n'a pas besoin de la valeur de retour d'un thread lancé (ce qui évite donc l'appel à la fonction *pthread_join* qui est bloquante), il peut créer le thread en mode "*DETACHED*". Dans ce cas-là, la mémoire sera correctement libérée à la fin du thread lancé et le thread principal pourra travailler sans se soucier de la fin de ce thread.

# Développement Système: les Threads Posix

## ***Les threads :*** Détachement

- Pour cela il existe la fonction ***int pthread_attr_init(pthread_attr_t *attr)***, qui permet d'initialiser une structure ***pthread_attr_t.***

- On peut ainsi modifier le champ ***detachstate*** pour l'initialiser en mode ***"DETACHED"*** grâce à la fonction

  ***int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate)***.

- Il faut ensuite passer un pointeur sur cette structure à la fonction ***pthread_create***.

- Il est aussi possible de laisser l'initialisation telle qu'elle est (donc par défaut) et utiliser la fonction : ***int pthread_detach(pthread_t th)*** qui modifie la structure associée au thread qui a été lancé.

- Il est aussi possible de modifier le type d'ordonnancement, la priorité et la taille de la pile (voir le manuel de ***pthread_attr_init*** et attributs de création de thread).

# Développement Système: les Threads Posix

## *Les threads :* Détachement

```
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_create(&tid, &attr, ma_fonction, NULL);
pthread_attr_destroy(&attr);
```

# Développement Système: les Threads Posix

## *Les threads :* Détachement

- **Exemple3: Détachement**

```
void *fonction_thread(void *arg);

int main(int argc, char *argv[])
{
        pthread_t th1,th2;
        int valeur1, valeur2;
        valeur1 = 4;
        pthread_create(&th1,NULL,fonction_thread,(void
*)&valeur1);
        valeur2 = 9;
        pthread_create(&th2,NULL,fonction_thread,(void
*)&valeur2);
        pthread_detach(th1);
        pthread_detach(th2);
        if (argc > 1)
                sleep(atoi(argv[1]));
        printf("\n Fin du createur\n");
        return EXIT_SUCCESS;
}
```

# Développement Système: les Threads Posix

## ***Les threads :*** Détachement (exécutions )

```
pi@raspberrypi:~/Documents/ExemplesThreads $ ./ex3
                Valeur de i : 0

 Fin du createur
                Valeur de i : 0
```

```
pi@raspberrypi:~/Documents/ExemplesThreads $ ./ex3 5
                Valeur de i : 0
                Valeur de i : 0
                Valeur de i : 1
                Valeur de i : 1
                Valeur de i : 2
                Valeur de i : 2
                Valeur de i : 3
                Valeur de i : 3

FIN du thread (val = 4)
                Valeur de i : 4

 Fin du createur
```

```
pi@raspberrypi:~/Documents/ExemplesThreads $ ./ex3 10
                Valeur de i : 0
                Valeur de i : 0
                Valeur de i : 1
                Valeur de i : 1
                Valeur de i : 2
                Valeur de i : 2
                Valeur de i : 3
                Valeur de i : 3

FIN du thread (val = 4)
                Valeur de i : 4
                Valeur de i : 5
                Valeur de i : 6
                Valeur de i : 7
                Valeur de i : 8

FIN du thread (val = 9)

 Fin du createur
```

# Les Threads: Autres actions

# Développement Système: les Threads Posix

## ***Les threads :*** *Autres fonctions*

Pour connaître son propre identifiant, un thread invoque la fonction pthread_self( ), qui lui renvoie une valeur de type pthread_t :

- ***pthread_t pthread_self (void);***

il est possible de comparer à l'aide de la fonction ***pthread_equal()*** l'identité de deux threads

***int pthread_equal (pthread_t thread1, pthread_t thread2);***

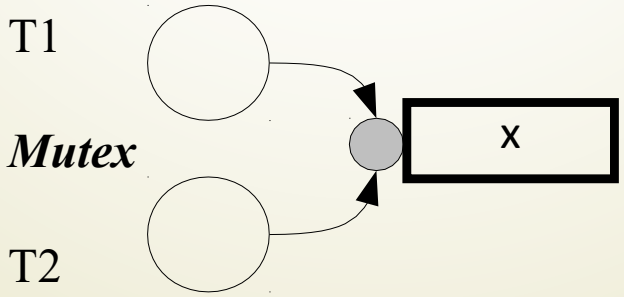
Cette fonction teste l'égalité entre deux identifiants de threads passés en argument. La fonction renvoie **0** si les deux threads sont différents, une valeur non nulle s'ils sont identiques.

# Développement Système: les Threads Posix

## ***Les threads :*** *Protection des données*

La cohérence des données ou des ressources partagées entre les processus légers est maintenue par des mécanismes de synchronisation. Il existe deux principaux mécanismes de synchronisation : ***mutex*** et variable ***condition.***

Un ***mutex*** (verrou d'exclusion mutuelle) possède deux états : verrouillé ou non verrouillé. Trois opérations sont associées à un mutex : **lock** pour verrouiller le mutex, **unlock** pour le déverrouiller et **trylock** (équivalent à lock , mais qui en cas d'échec ne bloque pas le thread).



# Les mutex

# Développement Système: les Threads Posix

## ***Les threads :*** *Protection des données****: les mutex***

- ***Création :***

***int pthread_mutex_init(pthread_mutex_t *mutex, const  pthread_mutexattr_t *mutexattr)***

- ***Destruction :***

***int pthread_mutex_destroy(pthread_mutex_t *mutex)***

- ***verrouiller*** :

***int pthread_mutex_lock(pthread_mutex_t *mutex)***

- ***déverrouiller :***

***int pthread_mutex_unlock(pthread_mutex_t *mutex)***

- ***Essayer de le verrouiller :***

***int pthread_mutex_trylock(pthread_mutex_t *mutex)***

- ***Le créer à la déclaration :***

***pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER***

# Développement Système: les Threads Posix

## ***Les threads :*** *Protection des données****: les mutex : Exemples***

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

#ifndef NUM_THREADS
#define NUM_THREADS 3
#endif

int valPartage = 0;

void* fThread(void* param) {
  printf("Incrementation de la valeur partagee...\n");
  for (int i = 0; i < 10000; ++i) {
    valPartage += 1;
  }
  return 0;
}

int main() {
  pthread_t threads[NUM_THREADS];

  for (int i = 0; i < NUM_THREADS; ++i) {
    pthread_create(&threads[i], NULL, fThread, NULL);
  }

  for (int i = 0; i < NUM_THREADS; ++i) {
    pthread_join(threads[i], NULL);
  }

  printf("Valeur partagee %d\n", valPartage);
  exit(EXIT_SUCCESS);
}
```

Dans cet exemple on crée 3 threads qui doivent incrémenter tous les 3 une valeur partagée. On ne protège pas l'accès à cette valeur. La valeur finale doit être 30000

***Exécution:***

```
pi@raspberrypi:~/Documents/ExemplesThreads $ ./ex4SansMutex
Incrementation de la valeur partagee...
Incrementation de la valeur partagee...
Incrementation de la valeur partagee...
Valeur partagee 23109
pi@raspberrypi:~/Documents/ExemplesThreads $ ./ex4SansMutex
Incrementation de la valeur partagee...
Incrementation de la valeur partagee...
Incrementation de la valeur partagee...
Valeur partagee 15288
pi@raspberrypi:~/Documents/ExemplesThreads $ ./ex4SansMutex
Incrementation de la valeur partagee...
Incrementation de la valeur partagee...
Incrementation de la valeur partagee...
Valeur partagee 23294
```

# Développement Système: les Threads Posix

## *Les threads : Protection des données: les mutex : Exemples*

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

#ifndef NUM_THREADS
#define NUM_THREADS 3
#endif

int valPartage = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* fThread(void* param) {
  pthread_mutex_lock(&mutex);
  printf("Incrementation de la valeur partagee...\n");
  for (int i = 0; i < 10000; ++i) {
    valPartage += 1;
  }
  pthread_mutex_unlock(&mutex);
  return 0;
}
int main() {
  pthread_t threads[NUM_THREADS];

  for (int i = 0; i < NUM_THREADS; ++i) {
    pthread_create(&threads[i], NULL, fThread, NULL);
  }

  for (int i = 0; i < NUM_THREADS; ++i) {
    pthread_join(threads[i], NULL);
  }

  printf("Valeur partagee %d\n", valPartage);
  exit(EXIT_SUCCESS);
}
```

Ici on protège l'accès à la valeur partagée par un mutex. La valeur finale doit être 30000

***Exécution:***

```
pi@raspberrypi:~/Documents/ExemplesThreads $ ./ex5AvecMutex
Incrementation de la valeur partagee...
Incrementation de la valeur partagee...
Incrementation de la valeur partagee...
Valeur partagee 30000
pi@raspberrypi:~/Documents/ExemplesThreads $ ./ex5AvecMutex
Incrementation de la valeur partagee...
Incrementation de la valeur partagee...
Incrementation de la valeur partagee...
Valeur partagee 30000
pi@raspberrypi:~/Documents/ExemplesThreads $ ./ex5AvecMutex
Incrementation de la valeur partagee...
Incrementation de la valeur partagee...
Incrementation de la valeur partagee...
Valeur partagee 30000
```

# Les sémaphores POSIX

# Développement Système: les Threads Posix

## ***Les threads :*** *Protection des données avec* ***les sémaphores POSIX***

***Les sémaphores POSIX permettent aux processus et aux threads de se synchroniser*** de la même manière que les sémaphores IPC.

Il est toutefois important de bien comprendre la différence entre les sémaphores POSIX qui sont utilisés dans les applications multithreading, et les sémaphores System V (ou IPC) qui, sont utilisés, pour les processus.

Un sémaphore est un entier dont la valeur ne peut jamais être négative.

Deux opérations peuvent être effectuées : incrémenter la valeur du sémaphore de 1, ou la décrémenter de 1.

Si la valeur courante est 0, le thread ou le processus est bloqué jusqu'à ce que la valeur devienne strictement positive.

Deux utilisations des sémaphores POSIX:

- La protection d'une ressource partagée (par exemple l'accès à une variable, une structure de donnée, une imprimante...). On parle de sémaphore d'exclusion mutuelle;
- La synchronisation de processus (ou threads) (un processus (ou threads) doit en attendre un autre pour continuer ou commencer son exécution).

# Développement Système: les Threads Posix

## *Les threads : utilisation des sémaphores POSIX*

- ***Création :***

***int sem_init(sem_t *sem, int pshared, unsigned int valeur)***

*si partagé entre threads d'un même processus : pshared = 0*

- ***Destruction :***

***int sem_destroy(sem_t * sem)***

- ***Prendre :***

***int sem_wait(sem_t * sem)***

- ***Vendre :***

***int sem_post(sem_t * sem)***

- ***Essayer de le prendre :***

***int sem_trywait(sem_t * sem)***

- ***Connaître sa valeur :***

***int sem_getvalue(sem_t * sem, int * sval)***

# Développement Système: les Threads Posix

## *Les threads : utilisation des sémaphores POSIX*

```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>

#ifndef THREADS
#define THREADS 4
#endif

int ValPartage = 0;
sem_t sem;
void *threadFonc(void *arg) {
    int loops = *((int *)arg);
    for (int j = 0; j < loops; j++) {
        if (sem_wait(&sem) == -1) perror("sem_wait");
        ValPartage++;
        if (sem_post(&sem) == -1) perror("sem_post");
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t t[THREADS];
    int s;
    int nloops;
    if (argc != 2) {
        fprintf(stderr, "Usage: %s nombre de tours\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    printf("\n valeur partagee avant  = %d\n", ValPartage);
    nloops = atoi(argv[1]);
    if (sem_init(&sem, 0, 1) == -1) perror("sem_init");
    for (int i = 0; i <= THREADS; ++i) {
        s = pthread_create(&t[i], NULL, threadFonc, &nloops);
        if (s != 0) perror("pthread_create");
    }
    printf("\n valeur partagee apres = %d\n", ValPartage);
    sem_destroy(&sem);
    exit(EXIT_SUCCESS);
}
```

Dans cet exemple on crée 4 threads qui doivent tous les 4 incrémenter N fois une valeur partagée protégée par sémaphore.

***Exécution:***

```
pi@raspberrypi:~/Documents/ExemplesThreads $ ./ex5sem 4

 valeur partagee avant  = 0

 valeur partagee apres = 16
pi@raspberrypi:~/Documents/ExemplesThreads $ ./ex5sem 10

 valeur partagee avant  = 0

 valeur partagee apres = 40
pi@raspberrypi:~/Documents/ExemplesThreads $ ./ex5sem 100

 valeur partagee avant  = 0

 valeur partagee apres = 400
```

# Développement Système: les Threads Posix

## ***Les threads :*** *Les conditions*

Une condition est un mécanisme de synchronisation permettant à un thread de suspendre son exécution jusqu'à ce qu'une certaine condition soit vérifiée. Une variable condition doit toujours être associée à un mutex, ceci évite qu'un thread se prépare à attendre une condition et qu'un autre signale la condition juste avant que le premier ne l'attende réellement.

**Pour créer une variable condition:**

***int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond attr)***

**Pour la détruire :**

***int pthread_cond_destroy(pthread_cond_t *cond)***

**Pour attendre une condition:**

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

Cette fonction déverrouille le mutex ***mutex*** et attend que la condition ***cond*** soit signalée. Pendant cette attente, ce thread ne consomme pas de temps processeur. Lorsque la condition est signalée, le mutex est reverrouillé. Il est donc nécessaire de verrouiller le mutex avant d'appeler cette fonction et de le déverrouiller après cette fonction.

# Développement Système: les Threads Posix

## ***Les threads :*** *Les conditions*

***int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime)***

Cette fonction réalise les mêmes choses que la fonction ***pthread_cond_wait*** mais ne reste bloquée que le temps précisé par ***abstime***. Il est donc possible d'attendre une condition avec un time out donné.

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

Cette fonction permet de signaler la condition ***cond*** et dans le cas où plusieurs threads attendent cette condition, seulement un thread sera libéré sans pouvoir savoir lequel.

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

Cette fonction permet, elle aussi, de signaler la condition *cond* mais libère tous les threads qui attendent cette condition.

Dans les deux cas, si aucun thread n'attend la condition le comportement est le même : il ne se passe rien.

# Développement Système: les Threads Posix

## ***Les threads : Les conditions*** : *Exemple*

```c
#include <math.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>

#define DUREE 5
typedef struct
{
    int val_max;
    pthread_mutex_t *pMutex;
    pthread_cond_t *pCondition;
    int n;
}param_thread;

void *fonction_thread(void *arg);
```

```c
void *fonction_thread(void *arg)
{
    param_thread *pParam;
    int i;
    int max;
    int n;
    pParam = (param_thread *)arg;
    max = pParam->val_max;
    n= pParam->n;
    printf("\tLancement du thread %d val_max =%d\n ",n,max );
    pthread_mutex_lock(pParam->pMutex);
    pthread_cond_wait(pParam->pCondition,pParam->pMutex);
    printf("\tthread %d débloqué\n",n);
    pthread_mutex_unlock(pParam->pMutex);
    for(i=0;i<max;i++)
    {
        printf("\t\tValeur de i : %d\n",i);sleep(1);
    }
    pthread_exit((void *)i);
}
```

# Développement Système: les Threads Posix

## ***Les threads : Les conditions*** *: Exemple*

```c
int main(int argc, char *argv[])
{
    pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
    pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
    param_thread param;
    pthread_t th1,th2;
    int retour;
    param.val_max = 6;
    param.n =1;
    param.pMutex = &mutex;
    param.pCondition = &cond;
    pthread_create(&th1,NULL,fonction_thread,(void *)&param);
    sleep(3);
    param.val_max = 2;
    param.n=2;
    pthread_create(&th2,NULL,fonction_thread,(void *)&param);
    printf("%d secondes de repos !\n", DUREE);
    sleep(DUREE);
    printf("Réveil !\n");
    pthread_cond_broadcast(&cond);
    printf("Condition signalée\n");
    pthread_join(th1,(void **)&retour);
    printf("Retour du thread1 : %d\n",retour);
    pthread_join(th2,(void **)&retour);
    printf("Retour du thread2 : %d\n",retour);
    return EXIT_SUCCESS;
}
```

# Développement Système: les Threads Posix

## ***Les threads : Les conditions*** *: Exemple*

***Exécution:***

```
pi@raspberrypi:~/Documents/ExemplesThreads $ ./ex6sync
        Lancement du thread 1 val_max =6
 5 secondes de repos !
        Lancement du thread 2 val_max =2
 Réveil !
Condition signalée
        thread 2 débloqué
                Valeur de i : 0
        thread 1 débloqué
                Valeur de i : 0
                Valeur de i : 1
                Valeur de i : 1
                Valeur de i : 2
                Valeur de i : 3
                Valeur de i : 4
                Valeur de i : 5
Retour du thread1 : 6
Retour du thread2 : 2
pi@raspberrypi:~/Documents/ExemplesThreads $
```

# Développement Système: les Threads Posix

## ***Les threads :*** *Les conditions*

### Exemple de synchronisation avec conditions:

- Afin de se bloquer et donc d'être synchronisés par le processus principal grâce à la condition, les threads commencent par prendre le mutex puis attendent la condition. La fonction ***pthread_cond_wait*** libère le mutex puis bloque le thread. Le second thread peut donc prendre le mutex sans être bloqué puis il attend lui aussi la condition ce qui libère le mutex. Lorsque la condition est signalée de manière générale (broadcast), tous les threads en attente sont donc libérés et la sortie de la fonction ***pthread_cond_wait*** reverrouille le mutex. C'est pour cela qu'il faut le déverrouiller (exécution de la fonction ***pthread_mutex_unlock***). Les threads ont donc été synchronisés par le processus principal et partent donc dans l'exécution de leur code.

- ***Remarque :*** le danger de cette synchronisation est de signaler la condition alors que tous les threads ne sont pas en attente. Ici le sommeil du processus principal évite ce problème mais ce n'est qu'un exemple permettant de montrer l'utilisation des conditions !