Cours "Système d'exploitation"

R3.05

2ème année BUT de Caen

Département d'informatique

*Développement système : **les threads***

Table des matières

Les Threads

I/ Introduction :

Imaginons un problème simple : une application avec une IHM où nous avons un objet qui se déplace (par exemple une *Progress barre*) et des boutons permettant de réaliser certaines actions. Pendant le déplacement de l'objet dans la fenêtre, l'IHM doit rester fonctionnelle. Le problème est que le déplacement de l'objet prend du temps, pendant ce temps toute action sur l'un des boutons est inopérante. La solution pour rendre l'interface active pendant le déplacement est de déléguer le déplacement de l'objet à une tâche.

LINUX qui est un système multi-tâches permet la création de processus fils à partir d'un processus existant en utilisant l'appel système *fork*, et donc permet de déléguer certaines tâches aux fils. Mais Les limites du *fork* apparaissent d'ores et déjà lorsqu'il s'agit de partager des variables entre un processus père et son fils. Une variable globale *i*, modifiée par le père a toujours l'ancienne valeur dans le fils. Ceci est le comportement normal du *fork* qui duplique le contexte courant lors de la création d'un processus fils. En plus d'empêcher le partage de variables, la création d'un nouveau contexte est pénalisante au niveau performances. Il en est de même pour le changement de contexte lors du passage d'un processus à un autre.

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

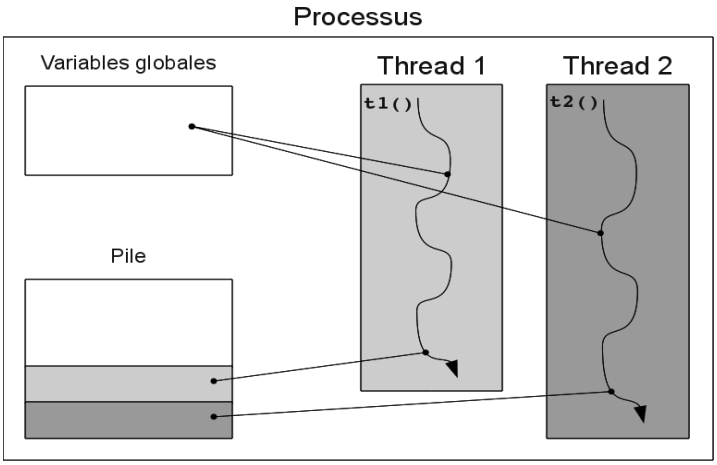
I-1/ Définition :

Un Thread (fil en anglais) est un fil d'exécution.

Plusieurs traductions sont apparues : *activité (tâche), lightweight process (processus léger)* par opposition au processus lourd créé par *fork.*

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

Un thread exécute une fonction.



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 …

Le Multithreading :

Le multithreading est la capacité d'un système d'exploitation à avoir simultanément des chemins d'exécution multiples au sein d'un processus unique.

En résumé :

- ✓ Chaque thread dispose d'une pile et d'un contexte d'exécution contenant les registres du processus et un compteur d'instructions.
- ✓ Les méthodes de communication entre threads sont naturellement plus simples que celles de la communication entre processus.
- ✓ En contrepartie, l'accès concurrentiel aux mêmes ressources nécessite une synchronisation pour éviter les interférences, ce qui complique les 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 totalement inutile d'essayer une approche multithread.

I-2/ Création d'un thread :

Il existe des appels-système qui permettent dans un contexte multithread de créer un nouveau thread, d'attendre la fin de son exécution, et de mettre fin au thread en cours.

- Un type ***pthread_t*** est utilisé pour distinguer les différents threads d'une application, à la manière des PID qui permettent d'identifier les processus.

La création d'un thread se fait à l'aide de la fonction

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

- Le premier argument est un pointeur qui sera initialisé par la routine avec l'identifiant du nouveau thread.
- Le second argument correspond aux attributs dont on désire doter le nouveau thread.Si ce pointeur est NULL, le thread aura des attributs par défaut.
- Le troisième argument est un pointeur représentant la fonction principale du nouveau thread. Cette fonction 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.

Ce nouveau thread s'exécute en concurrence du thread lanceur. Il va lancer la fonction ***start_routine*** qui reçoit l'argument ***arg.*** La fonction ***start_routine*** doit retourner un pointeur ***void ****.

Un thread peut être identifié par la fonction **pthread_t pthread_self(void).**

I-3/ Terminaison d'un thread :

Lorsqu'un processus créateur d'un (ou plusieurs) thread(s) se termine, ce (ou ces) thread(s) se termine(nt).

Le créateur doit donc attendre la fin d'un thread grâce à la fonction

int pthread_join(pthread_t th, void **thread_return)

Cette fonction suspend l'exécution du thread créateur et attend la fin du thread référencé par ***th***. Le créateur pourra obtenir la valeur retournée par le thread créé par l'intermédiaire de la variable ***thread_return***. Le thread créé pourra retourner une valeur au thread créateur grâce à la fonction ***void pthread_exit(void *retval).***

Il n'est pas possible d'attendre avec pthread_join la fin d'un thread donné parmi tous ceux qui sont lancé.

Exemple1 : Création et attente

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

void *ma_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,ma_fonction_thread,(void *)&valeur1);
    valeur2 = 7;
    pthread_create(&th2,NULL,ma_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 *ma_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);
}
```

L'exécution donne :

```
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$
```

On constate que les deux threads s'exécutent : ils comptent jusqu'à la valeur maximale qu'il leur a été transmise. Ils retournent cette valeur maximale (augmentée de 1) que le processus créateur récupère et affiche.

Exemple2 : pid d'un thread

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
typedef struct {
    int val;
    int num;
}param;
void *ma_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,ma_fonction_thread,(void *)&p1);
    p2.val=7;
    p2.num=2;
    pthread_create(&th2,NULL,ma_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 *ma_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\tValeur de i : %d\n",i );
        sleep(1);
    }
    pthread_exit((void *)i);
}
```

```
nouzhatber@ServeurLinux:~/Documents/Threads$ ./Exemple2

 PID du processus créateur : 678

 PID du thread2 : 678
                Valeur de i : 0

 PID du thread1 : 678
                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$
```

On constate que les 2 threads ont le même pid que le processus créateur.

II/Les attributs d'un thread

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.

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 aide *pthread_attr_init* et attributs de création de thread).

Exemple 3 : création d'un thread et détachement

Un processus un processus crée deux threads et les détache.

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

void *ma_fonction_thread(void *arg);

int main(int argc, char *argv[])
{
    pthread_t th1,th2;
    int valeur1, valeur2;
    valeur1 = 4;
    pthread_create(&th1,NULL,ma_fonction_thread,(void *)&valeur1);
    valeur2 = 9;
    pthread_create(&th2,NULL,ma_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;
}

void *ma_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);
    }
    printf("\nFIN du thread\n");
    pthread_exit((void *)i);
}
```

Exécution sans arguments de main :

```
nouzhatber@ServeurLinux:~/Documents/Threads$ ./Exemple3
                Valeur de i : 0
```

Exécution avec un argument de main à 5:

```
nouzhatber@ServeurLinux:~/Documents/Threads$ ./Exemple3 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
                Valeur de i : 4

FIN du thread

 Fin du createur
nouzhatber@ServeurLinux:~/Documents/Threads$
```

Exécution avec un argument de main à 10 :

```
nouzhatber@ServeurLinux:~/Documents/Threads$ ./Exemple3 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
                Valeur de i : 4

FIN du thread
                Valeur de i : 5
                Valeur de i : 6
                Valeur de i : 7
                Valeur de i : 8

FIN du thread

 Fin du createur
nouzhatber@ServeurLinux:~/Documents/Threads$
```

On constate que selon la valeur passé au *main* et donc à la fonction *sleep* les threads se terminent dès que le créateur est fini.

III/Protection de données partagées entre threads : les mutex

Les threads sont amenés à partager des données, il est donc nécessaire de protéger les accès à ces données. Ce problème peut être illustré par le problème typique de deux threads qui réalisent le même nombre de traitements sur une variable globale partagée. Un thread l'incrémente et l'autre la décrémente. La variable globale reviendra à sa valeur initiale à la fin puisqu'il y aura le même nombre d'incrémentations et de décrémentations. Chacun des deux threads doit attendre la fin de l'autre pour accéder à la valeur de cette variable. Pour cela, on peut utiliser des ***MUTEX*** ou des sémaphores afin de protéger cette variable.

Pour résoudre ce genre de problème, le système doit permettre au programmeur d'utiliser un verrou d'exclusion mutuelle, c'est-à-dire de pouvoir bloquer, en une seule instruction (atomique), toutes les tâches tentant d'accéder à cette donnée, puis, que ces tâches puissent y accéder lorsque la variable est libérée.

Remarque : une instruction atomique est une instruction qui ne peut être divisée (donc interrompue).

III-1/ Les MUTEX :

Un mutex est un objet d'exclusion mutuelle (MUTual EXclusion). Il est très pratique pour protéger des données partagées de modifications concurrentes et pour implémenter des sections critiques.

Un mutex peut être dans deux états : déverrouillé ou verrouillé (possédé par un thread).

Un mutex ne peut être pris que par un seul thread à la fois.

Un thread qui tente de verrouiller un mutex déjà verrouillé est suspendu jusqu'à ce que le mutex soit déverrouillé.

Les sémaphores d'exclusion mutuelle sont de type ***pthread_mutex_t***. Chaque sémaphore possède des attributs de type ***pthread_mutexattr_t***. La valeur prédéfinie pour les attributs de sémaphores est ***pthread_mutexattr_default*** (POSIX)

III-2/ Les fonctions de manipulation de mutex :

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

pthread_mutex_init initialise le mutex pointé par ***mutex*** selon les attributs de mutex spécifié par ***mutexattr***. Si ***mutexattr*** vaut NULL, les paramètres par défaut sont utilisés.

L'implémentation *LinuxThreads* ne supporte qu'un seul attribut, le type de *mutex*, qui peut être soit "rapide", "récursif" ou à "vérification d'erreur". Le type de mutex détermine s'il peut être verrouillé plusieurs fois par le même thread. Le type par défaut est "rapide".

Pour plus d'informations sur les attributs de mutex et leur utilisation, il faut lire la page "man" de pthread_mutexattr_init(3).

int pthread_mutex_lock(pthread_mutex_t *mutex);

pthread_mutex_lock verrouille le mutex. Si le mutex est déverrouillé, il devient verrouillé et est possédé par le thread appelant ; et *pthread_mutex_lock* rend la main immédiatement. Si le mutex est déjà verrouillé par un autre thread, *pthread_mutex_lock* suspend le thread appelant jusqu'à ce que le mutex soit déverrouillé.

int pthread_mutex_unlock(pthread_mutex_t *mutex);

pthread_mutex_unlock déverrouille le mutex.

int pthread_mutex_destroy(pthread_mutex_t *mutex);

pthread_mutex_destroy détruit un mutex, libérant les ressources qu'il détient. Le mutex doit être déverrouillé.

III-3/ Les sémaphores Posix

Les sémaphores POSIX ne doivent pas être confondus avec ceux des IPC.

Les fonctions d'utilisation de sémaphores POSIX :

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

et ***int sem_destroy(sem_t * sem);***

permettent de créer et de détruire un sémaphore POSIX.

La fonction ***int sem_wait(sem_t * sem);***

permet de prendre le sémaphore (ou de rester bloqué en attente de ce sémaphore).

La fonction ***int sem_trywait(sem_t * sem)*** ; essaie de prendre le sémaphore sans rester bloqué.

Et enfin la fonction ***int sem_post(sem_t * sem);*** libère le sémaphore.

La fonction ***int sem_getvalue(sem_t * sem, int * sval);*** permet d'avoir la valeur courante du sémaphore.

III-4/ Destruction d'un thread

Il existe un mécanisme dans lequel un thread peut en détruire un autre (à condition que ce dernier ait validé cette possibilité). C'est l'équivalent pour les threads de la fonction *kill()* des processus classiques. Plus précisément, un thread peut envoyer une requête d'annulation à un autre thread. Selon sa configuration, le thread ciblé peut soit ignorer la requête, soit l'honorer immédiatement, soit enfin retarder son application jusqu'à ce qu'un point d'annulation (***cancellation point***) soit atteint. Le comportement par défaut est de type synchrone (ou différé) et donc lorsqu'une requête de destruction est envoyée à un thread, elle n'est exécutée que lorsque ce thread passe par un "*cancellation point*".

La fonction : ***int pthread_cancel(pthread_t thread) ;*** détruit le thread d'identificateur *thread*. Celui-ci peut ignorer toute demande grâce à la fonction

int pthread_setcancelstate(int state, int *etat_pred);

Les deux états possibles sont :

- ✓ PTHREAD_CANCEL_ENABLE,
- ✓ PTHREAD_CANCEL_DISABLE,

etat_pred est un pointeur mémorisant l'état précédent. Si le thread a autorisé sa destruction (cas par défaut), cette destruction peut être immédiate ou différée et ceci est précisé par l'appel de la fonction

int pthread_setcanceltype(int mode, int *ancien_mode).

Les deux états possibles sont :

- ✓ PTHREAD_CANCEL_ASYNCHRONOUS (immédiatement)
- ✓ PTHREAD_CANCEL_DEFERRED (différé mode par défaut).

Lorsque le thread est en mode différé, il ne sera détruit que lorsqu'il passe par un "*cancellation point*". C'est à dire qu'il pourra tester par la fonction :

void pthread_testcancel(void); si un autre thread veut qu'il se détruise ou non.

Les autres points de destruction correspondent à l'exécution des fonctions : ***pthread_join*** (un thread qui attend la fin d'un autre thread provoquera donc sa destruction immédiate), ***pthread_cond_wait, pthread_cond_timedwait, sem_wait et sigwait.***

Tous les exemples permettant d'illustrer ces fonctionnements sont basés sur le même principe : un processus crée un mutex qu'il verrouille. Ensuite, il crée un thread et attend que

le thread ait modifié son comportement vis-à-vis de la destruction (test d'une variable booléenne).

Lorsque le processus sait que le thread a modifié son comportement, il tue ce thread et déverrouille le mutex. Il finit par analyser la fin du thread.

Le thread commence par modifier son comportement vis-à-vis de son éventuelle destruction, le signale (positionnement de la variable booléenne) puis prend le mutex. Ce dernier ayant déjà été pris par le processus, le thread se retrouve bloqué. Lorsque le processus libère le mutex, le thread va pouvoir poursuivre (s'il le peut !) son exécution.

(voir exemples fournis)

IV/ Conditions et synchronisation

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.

Les fonctions ***int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond attr)*** et ***int pthread_cond_destroy(pthread_cond_t *cond)*** initialise pour l'une et détruit pour l'autre une variable condition.

La fonction ***int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)*** 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.

La fonction ***int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime)*** réalise les mêmes choses mais ne reste bloquée que le temps précisé par ***abstime***. Il est donc possible d'attendre une condition avec un time out donné.

La fonction ***int pthread_cond_signal(pthread_cond_t *cond)*** 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.

La fonction ***int pthread_cond_broadcast(pthread_cond_t *cond)*** 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.

Exemple :

```
#include <stdlib.h>
#include <math.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>
#define DUREE 8

typedef struct
{
        int val_max;
        pthread_mutex_t *pMutex;
        pthread_cond_t *pCondition;
}param_thread;

void *ma_fonction_thread(void *arg);

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.pMutex = &mutex;
        param.pCondition = &cond;
        pthread_create(&th1,NULL,ma_fonction_thread,(void *)&param);
        sleep(3);
        param.val_max = 2;
        pthread_create(&th2,NULL,ma_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");
        return EXIT_SUCCESS;
}
```

```
void *ma_fonction_thread(void *arg)
{
	param_thread *pParam;
	int i;
	int max;
	pParam = (param_thread *)arg;
	max = pParam->val_max;
	printf("\tLancement d'un thread val_max =%d\n ",max );
	pthread_mutex_lock(pParam->pMutex);
	pthread_cond_wait(pParam->pCondition,pParam->pMutex);
	printf("\tthread débloqué\n");
	pthread_mutex_unlock(pParam->pMutex);
	for(i=0;i<max;i++)
	{
		printf("\t\tValeur de i : %d\n");sleep(1);
	}
	pthread_exit((void *)i);
}
```

Exécution :

```
nouzhatber@ServeurLinux:~/Documents/Threads$ ./SynchronisationThread
        Lancement d'un thread val_max =6
 8 secondes de repos !
        Lancement d'un thread val_max =2
 Réveil !
Condition signalée
        thread débloqué
                Valeur de i : 0
        thread 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
nouzhatber@ServeurLinux:~/Documents/Threads$
```

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 !