Chapitre 12

Les threads

1. Processus

- Chaque application (emacs, word, etc.) exécutée sur un ordinateur lui est associée un processus représentant l'activité de celle-ci.
- À ce processus est associé un ensemble de ressources personnalisées comme l'espace mémoire, le temps CPU etc.
- Ces ressources seront dédiées à l'exécution des instructions du programme associé à l'application.
- Les processus coûtent chers au lancement (l'espace mémoire à calculer, les variables locales au processus à définir etc.).

2. Multitâche

- C'est la possibilité que le système d'exploitation a pour faire fonctionner plusieurs programmes en même temps et cela, sans conflit.
- Par exemple: écouter de la musique et lire son courrier en même temps.
- Un système monoprocesseur (et non pas monoprocessus, ne pas confondre processus et processeur), simule le multitâche en attribuant un temps (un quantum ou temps déterminé à l'avance) de traitement pour chaque processus (description très simpliste du rôle du monoprocesseur).

- Un système multiprocesseur peut exécuter les différents processus sur les différents processeurs du système.

- Généralement, un programme est livré avec un tel système pour permettre de distribuer de manière efficace les processus sur les processeurs de la machine.

- Un programme est dit multitâche s'il est capable de lancer plusieurs parties de son code en même temps.

- À chaque partie du code sera associé un processus (sous-processus) pour permettre l'exécution en parallèle.

- La possibilité de lire l'aide en ligne et d'imprimer à la fois sous word. Ce sont deux sous-processus relevant d'un processus qui est associé à l'application word.

- Un serveur réseau (processus) et le traitement de requêtes (sous-processus associé à chaque requête).

- Nous aimerions que le traitement de ces requêtes soit fait de manière optimale de telle sorte à pouvoir servir toutes les requêtes dans un espace de temps raisonnable.

- Comme chaque processus coûte cher au lancement, sans oublier qu'il va avoir son propre espace mémoire, la commutation (passage d'un processus à un autre) ainsi que la communication inter-processus (échange d'informations) seront lourdes à gérer.

Question

- Peut-on avoir un sous-processus qui permettrait de lancer une partie du code d'une application sans qu'il soit onéreux?

- La réponse est venue avec les threads qui sont appelés aussi « processus légers ».

3. Threads

- Un thread est un sous-ensemble d'un processus, partageant son espace mémoire et ses variables.

- De ce fait, les coûts associés suite à son lancement sont réduits, donc il est plus rapide.

- En plus de cela, à chaque thread est associé des unités propres à lui: comme sa pile (pour gérer les instructions à exécuter par le thread), le masque de signaux (les signaux que le thread doit répondre), sa priorité d'exécution (dans la file d'attente), des données privées etc.

- Il faut ajouter à cela, les threads peuvent être exécutés en parallèle par un système multitâche.

- Cependant, le partage de la mémoire et les variables du processus entraînent un certain nombre de problèmes lorsqu'il y a un accès partagé à une ressource, par exemple.

- Deux agents (threads) voulant effectuer des réservations de places d'avion (ressource: nombre de places disponibles dans un avion).

- On protège l'accès à cette ressource dès qu'un thread est entrain de réaliser une écriture (réaliser une réservation) sur cette ressource.

4. C++ et les threads

- Le langage C++ ne dispose pas d'une librairie native pour manipuler les threads.
- Le comité en charge de standardiser le langage C++ est entrain de corriger cette carence. Une version normalisée va voir le jour dans le prochain standard « C++0x »[1], le standard actuel étant « C++98 » et a été mis à jour en 2003 « C++03 ».
- De ce fait plusieurs paquetages ont été rendus disponibles pour combler cette carence.
- Parmi ces paquetages, nous pouvons citer « POSIX threads », « Mach C-threads », « Solaris 2 UI-threads ». Et tout récemment, « wxThread », « Boost thread » etc.
- À noter que les des systèmes d'exploitation récents implémentent dans leur noyau la gestion des threads.
- Pour ce cours, nous allons utiliser le paquetage « Posix threads », connu aussi sous le nom de « Pthread ».

[1] http://en.wikipedia.org/wiki/C%2B%2B0x

5. Pthreads

5.1. Définition

- Les Pthreads font référence au standard « POSIX » qui définit une API pour la création et la synchronisation de threads.
- Ce standard a été normalisé en 1996 (IEEE, ANSI et ISO).
- Des librairies ont été développées pour implémenter les Pthreads définis par un tel standard. Ces librairies sont disponibles sur des systèmes comme Unix ou Solaris 2.
- En mars 1999 une première adaptation Pthreads a vu le jour pour le système Windows. La dernière version disponible d'une telle adaptation porte la référence « 2.8.0 » en date du « 22 décembre 2006 ».

5.2. Installation

- L'utilisation des « Pthreads » sous Linux se fait de manière transparente en ajoutant le flag « -lpthread » lors de l'édition de lien.
- Pour Windows, il faudra d'abord télécharger la dernière version « pthread-w32-v-v-v-release.exe » de ce site web :

http://sourceware.org/pthreads-win32/

Par la suite, il faudra copier les fichiers se trouvant dans le répertoire « Pre-built.2 » dans le répertoire « mingw » comme suit :

Fichiers	Destination
pthreadGCE.dll	mingw/bin
libpthreadGCE2.a	mingw/lib
pthread.h, sched.h et semaphore.h	mingw/include

La compilation d'un programme donné « explethread.cpp » se fera comme suit:

```
g++ -Wall -pedantic explethread.cpp -o explethread.exe -lpthreadGCE2
```

Il est possible de renommer « libpthreadGCE2.a » en « libpthread.a » et utiliser ainsi le même flag, que sur un système Linux, « -lpthread » lors de l'édition de liens sous Windows.

L'extension « GCE » signifie : « Gnu C avec la gestion des Exceptions C++ ».

Pour pouvoir compiler un programme, il faudra penser à inclure le fichier d'en-tête « pthread.h ».

5.3. Création et terminaison d'un thread

- On déclare d'abord une instance de pthread comme suit :

```
pthread_t ta; // ta une instance de pthread
```

- Pour créer un thread, il faudra utiliser la fonction « pthread_create ». La signature d'une telle fonction est comme suit :

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

- Cette fonction retourne « 0 » si le thread a été créé.

- Le premier argument « thread » est un pointeur vers une instance de « pthread ». Ce pointeur contient l'identificateur du thread créé. Le second argument « attr » contient les attributs du thread à créer. On passe la valeur « NULL » pour initialiser les attributs avec les valeurs par défaut. Le troisième argument est une fonction que le thread va exécuter. Cette fonction retourne un « void* » et accepte un argument du type « void* ». Le dernier argument « arg » est l'argument que nous allons passer à la fonction « start_routine ».

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

using namespace std;

const int N=10;

void * writer_thread (void * arg) {
    int i;
    for (i = 0; i < N; i++)
        cout << "   Thread enfant.\n";
    return NULL;
}
int main (int argc, char *argv[]) {
    int i;
    pthread_t writer_id;

    pthread_create (&writer_id, NULL, writer_thread, NULL);
    for(i = 0; i < N; i++)
        cout << "Thread Parent\n";

    pthread_exit (NULL);
    return 0;
}
```

- Dans cet exemple nous sommes en présence de 2 threads : le thread « parent » est représenté par la fonction « main » et le thread « enfant » par l'instance « writer_id ».

- Le programme va effectuer une série de 10 affichages en sortie : « Thread parent » un affichage provoqué dans la fonction « main » et « Thread enfant » un affichage provoqué par la fonction « writer_thread ». Cette fonction est exécutée par le thread « writer_id ».

- L'ordre dans lequel les tâches sont exécutées n'est pas défini. Voici deux affichages possibles en sortie :

Sortie -1-	Sortie -2-
Thread enfant.	Thread Parent
Thread enfant.	Thread Parent
Thread enfant.	Thread enfant.
Thread enfant.	Thread enfant.
Thread enfant.	Thread enfant.
Thread enfant.	Thread enfant.
Thread enfant.	Thread enfant.
Thread enfant.	Thread enfant.
Thread Parent	Thread enfant.
Thread Parent	Thread enfant.
Thread Parent	Thread enfant.
Thread Parent	Thread enfant.
Thread Parent	Thread Parent
Thread Parent	Thread Parent
Thread Parent	Thread Parent
Thread Parent	Thread Parent
Thread Parent	Thread Parent
Thread Parent	Thread Parent
Thread enfant.	Thread Parent
Thread enfant.	Thread Parent

- Nous avons utilisé dans le programme la fonction « pthread_exit(NULL) ». Cette fonction sert à terminer un thread.

- Il faudra utiliser cette fonction à la place de « exit » afin d'éviter d'arrêter tout le processus.

- L'appel à une telle fonction à la fin de la fonction « main » est utile pour éviter que le programme arrête les autres threads en cours.

- La signature d'une telle fonction est comme suit :

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

- « retval » est la valeur de retour du thread.

5.4. Mise en attente

- Si dans le précédent programme, nous avons ajouté deux instructions permettant d'afficher les positions de départ et de fin de la fonction « main » comme suit :

```
int main (int argc, char *argv[]) {

    cout << "*** Debut main ***\n";

    pthread_t writer_id;

    pthread_create (&writer_id, NULL, writer_thread, NULL);

    for(i = 0; i < N; i++)
        cout << "Thread Parent\n";

    cout << "*** Fin main ***\n";

    pthread_exit (NULL);

    return 0;
}
```

- Un affichage possible en sortie serait comme suit :

Sortie

```
*** Debut main ***
Thread Parent
Thread Parent
Thread Parent
Thread Parent
Thread Parent
Thread Parent
Thread Parent
Thread Parent
Thread Parent
Thread Parent
*** Fin main ***
   Thread enfant.
   Thread enfant.
   Thread enfant.
   Thread enfant.
   Thread enfant.
   Thread enfant.
   Thread enfant.
   Thread enfant.
   Thread enfant.
   Thread enfant.
```

- À noter dans le précédent affichage que la fonction « main » se termine avant la fin du thread « writer_id ».

- Nous utilisons la fonction « pthread_join » pour mettre en attente le programme tant que le thread « writer_id » n'a pas terminé de s'exécuter.

- La fonction « pthread_join » c'est une des formes de synchronisation entre les threads.

- Par défaut dans le standard POSIX, tous les threads sont créés dans l'état « joignable » par opposé à l'état « détaché ».

- La signature d'une telle fonction est comme suit :

```
int pthread_join(pthread_t th, void** thread_return);
```

- La fonction « pthread_join » attend que le thread « th » se termine.

- La valeur retournée par la fonction est « 0 » si tout est correct.

- Le pointeur « thread_return » pointe l'emplacement où est enregistré la valeur de retour du thread « th ».

- Cette valeur de retour est spécifiée à l'aide de la méthode « pthread_exit ».

```
int main (int argc, char *argv[]) {
    int i;

    cout << "***** Debut de la fonction main *****"<<endl;

    // Un thread, une instance de pthread_t
    pthread_t writer_id;

    // Création du thread "writer_id"
    pthread_create (&writer_id, NULL, writer_thread, NULL);
    for(i = 0; i < N; i++)
        cout << "Thread Parent\n";


    pthread_join(writer_id,NULL);

    cout << "***** Fin de la fonction main *****"<<endl;

    // Terminaison du thread
    pthread_exit (NULL);


    return 0;
}
```


- Deux affichages possibles en sortie seraient comme suit :

Sortie -1-	Sortie -2-
***** Debut de la fonction main ***** Thread Parent Thread Parent Thread Parent Thread Parent Thread Parent Thread enfant. Thread enfant. Thread enfant. Thread enfant. Thread enfant. Thread enfant. Thread enfant. Thread enfant. Thread enfant. Thread enfant. Thread Parent Thread Parent Thread Parent Thread Parent Thread Parent ***** Fin de la fonction main *****	***** Debut de la fonction main ***** Thread Parent Thread Parent Thread Parent Thread Parent Thread Parent Thread Parent Thread Parent Thread Parent Thread Parent Thread Parent Thread Parent Thread enfant. Thread enfant. Thread enfant. Thread enfant. Thread enfant. Thread enfant. Thread enfant. Thread enfant. Thread enfant. ***** Fin de la fonction main *****

- Le thread enfant quand il a la main, il ne la rend que lorsqu'il a fini son traitement.

- Nous utilisons la fonction « pthread_detach » pour détacher un thread. Quand ce thread prendra fin, l'espace mémoire occupé par celui-ci sera libéré. En le mettant dans un état détaché, nous n'avons pas besoin d'attendre qu'un autre thread fasse un appel à pthread_join.

```
int pthread_detach(pthread_t th);
```

5.5 La synchronisation

- Nous avons mentionné que les threads partagent les mêmes ressources (mémoire, variables du processus etc.).

- Dans certaines situations, il est nécessaire de synchroniser les threads pour obtenir un résultat cohérent.

- Prenez l'exemple d'un avion où il reste une seule place de disponible et deux clients se présentant à deux différents guichets. Si la place est proposée au premier client et que ce dernier prenne tout son temps pour réfléchir, nous n'allons pas attribuer cette place au second client qui en a fait la demande. De ce fait, nous synchronisons l'accès à la méthode de réservation de telle manière à ne pas attribuer la même place aux deux voyageurs ou bien au second voyageur avant que le premier ne prenne sa décision de la prendre ou pas.

Monitor (ou sémaphore)

- Le moniteur est utilisé pour synchroniser l'accès à une ressource partagée.

- Cette ressource peut-être un segment d'un code donné.

- Un thread accède à cette ressource par l'intermédiaire de ce moniteur.

- Ce moniteur est attribué à un seul thread à la fois (comme dans un relais 4x100m où un seul coureur tient le témoin dans sa main pour le passer au coureur suivant dès qu'il a terminé de courir sa distance).

- Pendant que le thread exécute la ressource partagée aucun autre thread ne peut y accéder.

- Le thread libère le monitor dès qu'il a terminé l'exécution du code synchronisé.

- Nous assurons ainsi que chaque accès aux données partagées est bien « mutuellement exclusif ».

- Nous allons utiliser pour cela les « mutex » une abréviation pour « Mutual Exclusion » ou « exlusion mutuelle ».

5.6 « Mutex »

- Un mutex est un verrou possédant deux états : déverrouillé (pour signifier qu'il est disponible) et verrouillé (pour signifier qu'il n'est pas disponible). Mutex est une variable d'exclusion mutuelle.

- Un mutex est du type « pthread_mutex_t ». Par exemple dans l'exemple ci-dessous, « mutex » est une instance de « pthread_mutex_t ».

```
pthread_mutex_t mutex;
```

- Avant de pouvoir utiliser un mutex, il faudra l'initialiser.

- Nous pouvons initialiser un mutex à l'aide d'une macro, dans ce cas il va avoir le comportement par défaut.

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- Nous pouvons initialiser aussi un mutex à l'aide de la fonction « pthread_mutex_init ».

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

- Le premier argument correspond au mutex à initialiser. Le second argument représente les attributs à utiliser lors de l'initialisation. Nous pouvons utiliser la valeur « NULL ».

- La fonction retourne toujours la valeur 0.

- Pour détruire un mutex, nous utilisons la fonction « pthread_mutex_destroy ».

```
int pthread_mutex_destroy (pthread_mutex_t* mutex);
```


Verrouillage / Déverrouillage

- Un thread peut verrouiller le mutex s'il a besoin d'un accès exclusif aux données protégées.

- Pour verrouiller un mutex, nous utilisons la fonction « pthread_mutex_lock ».

```
int pthread_mutex_lock (pthread_mutex_t* mutex);
```

- Dans ce cas de figure, la fonction retourne soit un message d'erreur pour signifier qu'une erreur s'est produite. 0 dans le cas contraire pour signifier que le mutex a été correctement verrouillé. Le thread en question est le seul à détenir un « verrou » sur le mutex.

- Si le mutex est déjà verrouillé par un autre thread, la fonction bloque tant que le verouillage sur le mutex n'est pas obtenu.

- Il existe une autre façon pour verrouiller un mutex.

- Nous pouvons utiliser aussi la fonction « pthread_mutex_trylock »

```
int pthread_mutex_trylock (pthread_mutex_t* mutex);
```

- Cette fonction ne bloque pas si le mutex est déjà verrouillé par un autre thread.

- Elle retourne 0 en cas de succès. Dans le cas contraire, le mutex est verrouillé, elle retourne « EBUSY ».

- Quand un thread a fini de travailler avec les données « protégées », il doit libérer le mutex. Pour cela, il doit le déverrouiller pour qu'un autre thread puisse l'utiliser. La fonction « pthread_mutex_unlock » est utilisée pour réaliser une telle opération.

```
int pthread_mutex_unlock (pthread_mutex_t* mutex);
```

- Le programme qui suit peut-être compilé de deux manières dépendamment de l'utilisation ou non de l'option « MUTEX ».

- Si l'utilisation des mutex n'est pas activée, il faudra utiliser cette commande de compilation :

« g++ -Wall –pedantic -Os exemple.cpp -o exemple.exe -lpthreadGCE2 »

- Si l'utilisation des mutex est activée, il faudra utiliser cette commande de compilation :

« g++ -Wall –pedantic –DMUTEX -Os exemple.cpp -o exemple.exe -lpthreadGCE2 »


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

using namespace std;

const int N_PROCESS=10;
// Pour éviter de faire un lock en trop
const int NM_PROCESS= N_PROCESS - 1;
int nParam;

#ifdef MUTEX
   pthread_mutex_t mutex;
#endif

void * Thread (void * arg) {
    int ParamLocal;

    ParamLocal = nParam;
    cout << "ParamLocal: " << ParamLocal << ", arg: " << (int)arg\
                        << ", " << ((ParamLocal == (int)arg) \
                        ? "ok" : "bug !!!!!") << "\n";

#ifdef MUTEX
    pthread_mutex_unlock(&mutex);
#endif
    pthread_detach(pthread_self());
    return NULL;
}
```

```
int main (int argc, char *argv[]) {
    int i;
    pthread_t ecrivain_id;

#ifdef MUTEX
    cout << "\nExemple de MUTEX et passage de parametres.\n\n";
#else
    cout << "\nExemple de bug-MUTEX et passage de parametres.\n\n";
#endif

#ifdef MUTEX
    pthread_mutex_init(&mutex, NULL);
    pthread_mutex_lock(&mutex);
#endif
    for (i = 0; i < N_PROCESS; i++) {
        nParam = i;
        pthread_create (&ecrivain_id, NULL, Thread, (void*)i);
#ifdef MUTEX
        if (i != NM_PROCESS) pthread_mutex_lock(&mutex);
#endif
    }
#ifdef MUTEX
    pthread_mutex_destroy(&mutex);
#endif
    pthread_exit (NULL);

    return 0;
}
```


sans l'option MUTEX	avec l'option MUTEX
Exemple de bug-MUTEX et passage de parametres. ParamLocal: ParamLocal: ParamLocal: ParamLocal: ParamLocal: ParamLocal: ParamLocal: ParamLocal: ParamLocal: ParamLocal: 5, arg: 1, bug !!!!! 5, arg: 2, bug !!!!! 5, arg: 3, bug !!!!! 5, arg: 4, bug !!!!! 9, arg: 5, bug !!!!! 9, arg: 6, bug !!!!! 9, arg: 7, bug !!!!! 9, arg: 8, bug !!!!! 9, arg: 9, ok 5, arg: 0, bug !!!!!	Exemple de MUTEX et passage de parametres. ParamLocal: 0, arg: 0, ok ParamLocal: 1, arg: 1, ok ParamLocal: 2, arg: 2, ok ParamLocal: 3, arg: 3, ok ParamLocal: 4, arg: 4, ok ParamLocal: 5, arg: 5, ok ParamLocal: 6, arg: 6, ok ParamLocal: 7, arg: 7, ok ParamLocal: 8, arg: 8, ok ParamLocal: 9, arg: 9, ok

5.7 Variables de condition

- Les variables offrent une autre manière aux threads pour se synchroniser.
- Au lieu de se synchroniser sur « l'accès à la donnée » comme dans le cas des mutex, les variables de condition permettent aux threads de synchroniser en fonction de la valeur de la donnée.
- À l'aide de ces variables de condition, un thread peut informer d'autres threads de l'état d'une donnée partagée.
- Une variable de condition est toujours utilisée conjointement avec « mutex_lock ».

Création et destruction d'une variable de condition

- Les opérations de création et de destruction d'une variable de condition sont similaires à celles en regard de la création ou la destruction de mutex.

- Une variable de condition est une instance de « pthread_cond_t ».

- Cette variable peut-être initialisée de manière statique :

```
pthread_cond_t condvar = PTHREAD_COND_INITIALIZER;
```

Ou bien à travers une fonction d'initialisation « pthread_cond_init » :

```
int pthread_cond_init(pthread_cond_t* pCond,
                      pthread_condattr_t* pCondattr);
```

- Pour libérer une variable de condition, nous allons utiliser « pthread_cond_destroy » :

```
int pthread_cond_destroy(pthread_cond_t* pCond);
```


Attente et notification

- Les variables de condition sont associées de facto à un mutex. Ce dernier va bloquer le thread jusqu'à qu'il y ait notification ou que le temps d'attente s'est écoulé.

- Nous utilisons les deux fonctions suivantes pour la mise en attente :

```
int pthread_cond_wait(pthread_cond_t* pCond,
                      pthread_mutex_t* pMutex);


int pthread_cond_timewait(pthread_cond_t* pCond,
                      pthread_mutex_t* pMutex,
                      struct timespec* tempsattente);
```

- La première fonction associe la condition au mutex. La seconde en plus de faire cette association ajoute une contrainte de temps comme 3^{e} argument.

- La première fonction ne rend la main que lorsque la notification est reçue ou qu'une erreur s'est produite. Alors que la seconde ajoute aussi un temps d'expiration. Si ce temps est écoulé, une valeur différente de 0 est retournée par la fonction.

- Si tout ce passe bien, les deux fonctions retournent la valeur 0. Dans le cas contraire la valeur retournée est non nulle.

- Elle retourne 0 en cas de succès. Dans le cas contraire, le mutex est verrouillé, elle retourne « EBUSY ».

- Ces fonctions bloquent le thread appelant tant que la condition spécifiée est signalée.

- Ces fonctions doivent être appelées quand un mutex est verrouillé. Elles vont automatiquement relâcher le mutex quand elles sont en mode attente.

- Après qu'un signal est reçu et que le thread est réveillé, le mutex sera automatiquement verrouillé pour être utilisé par le thread.

- Il faudra penser à relâcher le mutex quand le thread n'en a plus besoin.

- Pour réveiller un thread, nous utilisons la fonction « pthread_cond_signal » :

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

- Pour réveiller plus d'un thread qui sont dans en mode d'attente d'un signal, nous utilisons la fonction « pthread_cond_broadcast » :

```
int pthread_cond_broadcast(pthread_cond_t* pCond);
```

- Il faudra appeler « pthread_cond_wait » avant « pthread_cond_signal ». On ne réveille pas un thread avant de l'avoir mis en attente !


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

using namespace std;

const int NBRE_THREADS = 3;
const int TCOMPTEUR = 10;
const int COMPTEUR_LIMITE = 12;

int     compteur = 0;
int     thread_ids[3] = {0,1,2};
pthread_mutex_t compteur_mutex;

// mutex ajouté pour controller l'affichage en sortie

pthread_mutex_t affichage_mutex = PTHREAD_MUTEX_INITIALIZER;

pthread_cond_t compteur_seuil_cv;

void *inc_compteur(void *idp) {
    int j,i;
    double result=0.0;
    int *my_id = (int *) idp;

    for (i=0; i < TCOMPTEUR; i++) {
        pthread_mutex_lock(&compteur_mutex);
        compteur++;
```

```
        /*
        Vérifie la valeur de compteur. Si cette valeur est atteinte,
        envoyer un signal au thread en attente pour l'informer que
        la condition est satisfaite. À noter que cela se produit
        quand le mutex est verrouillé.
        */
        if (compteur == COMPTEUR_LIMITE) {
            pthread_cond_signal(&compteur_seuil_cv);
            pthread_mutex_lock(&affichage_mutex);
            cout << "inc_compteur(): thread " << *my_id << "\
                       compteur = " << compteur << " Seuil a ete\
                       atteint.\n";
            pthread_mutex_unlock(&affichage_mutex);
        }
        pthread_mutex_lock(&affichage_mutex);
        cout << "inc_compteur(): thread " << *my_id << \
                       " compteur = " << compteur << " ,\
                       deverouillage du mutex.\n";
        pthread_mutex_unlock(&affichage_mutex);

        pthread_mutex_unlock(&compteur_mutex);

        /* Faire quelque chose en sorte que les threads peuvent
                       alterner sur l'état du mutex */
        for (j=0; j < 1000; j++)
            result = result + (double)rand();
    }
    pthread_exit(NULL);
    return NULL;
}
```



```
void *observe_compteur(void *idp)  {
    int *my_id = (int *) idp;

    pthread_mutex_lock(&affichage_mutex);

    cout << "Debut de observe_compteur(): thread " << *my_id <<"\n";
    pthread_mutex_unlock(&affichage_mutex);


    /*
        Verrouiller le mutex et attendre le signal associé à la
        condition prédéfinie.
        À noter que la fonction pthread_cond_wait va automatiquement
        et « atomiquement » déverrouiller le mutex quand elle est en
        mode "attente".
        À noter aussi que si la valeur limite de "COMPTEUR_LIMITE"
        est atteinte avant que cette fonction n'est utilisée par un
        thread en attente, la boucle va être évitée afin de prévenir
        que la fonction pthread_cond_wait ne retourne jamais de sa
        condition.
    */

    pthread_mutex_lock(&compteur_mutex);
    // Vérifier la condition
    if (compteur < COMPTEUR_LIMITE) {

        /* On met le thread en attente tant que la condition n'est
           pas satisfaite. Le thread est en attente du signal pour
           sortir de son état "d'attente". Pour signaler que la
           condition est satisfaite, le thread en charge de la
           condition va envoyer un signal */
```

```
        pthread_cond_wait(&compteur_seuil_cv, &compteur_mutex);
        pthread_mutex_lock(&affichage_mutex);
        cout << "observe_compteur(): thread " << *my_id << ". La
                condition sur reception du signal a ete recue\n";
        pthread_mutex_unlock(&affichage_mutex);
    }
    pthread_mutex_unlock(&compteur_mutex);
    pthread_exit(NULL);
    return NULL;
}

int main(int argc, char *argv[]) {
    int i;
    pthread_t threads[3];
    pthread_attr_t attr;

    /* Initialiser les mutex et les variables de condition */
    pthread_mutex_init(&compteur_mutex, NULL);
    pthread_cond_init (&compteur_seuil_cv, NULL);

    /* Pour des raisons de portabilite, creer les threads de maniere
       explicite dans l'etat joignable */

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    pthread_create(&threads[0], &attr, inc_compteur, \
                        (void *)&thread_ids[0]);
    pthread_create(&threads[1], &attr, inc_compteur, \
                        (void *)&thread_ids[1]);
```



```
    pthread_create(&threads[2], &attr, observe_compteur, \
                        (void *)&thread_ids[2]);

    /* Attendre que tous les threads prennent fin */
    for (i = 0; i < NBRE_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_lock(&affichage_mutex);
    cout << "Main(): Le traitement des " << NBRE_THREADS << "\
                        threads a pris fin.\n";

    pthread_mutex_unlock(&affichage_mutex);

    /* Netoyage et quitter le programme */
    pthread_attr_destroy(&attr);
    pthread_mutex_destroy(&compteur_mutex);
    pthread_cond_destroy(&compteur_seuil_cv);

    pthread_exit (NULL);

    return 0;
}
```

- L'affichage obtenu en sortie est comme suit :

```
inc_compteur(): thread 0 compteur = 1 , deverouillage du mutex.

Debut de observe_compteur(): thread 2

inc_compteur(): thread 1 compteur = 2 , deverouillage du mutex.
inc_compteur(): thread 1 compteur = 3 , deverouillage du mutex.
inc_compteur(): thread 1 compteur = 4 , deverouillage du mutex.
inc_compteur(): thread 1 compteur = 5 , deverouillage du mutex.
inc_compteur(): thread 1 compteur = 6 , deverouillage du mutex.
inc_compteur(): thread 1 compteur = 7 , deverouillage du mutex.
inc_compteur(): thread 1 compteur = 8 , deverouillage du mutex.
inc_compteur(): thread 1 compteur = 9 , deverouillage du mutex.
inc_compteur(): thread 1 compteur = 10 , deverouillage du mutex.
inc_compteur(): thread 1 compteur = 11 , deverouillage du mutex.
                  thread 1 a fini ses 10 incrémentations
                   c'est le tour maintenant du thread 0
                   car thread 2 est en attente de la condition

inc_compteur(): thread 0 compteur = 12 Seuil a ete atteint.
                  envoi d'un signal pour réveiller thread 2

inc_compteur(): thread 0 compteur = 12 , deverouillage du mutex.

observe_compteur(): thread 2. La condition sur reception du signal a ete recue
                  thread 2 accuse reception
                  thread 2 a fini place au dernier thread qui reste
                  dans la file d'attente, i.e. thread 0
```



```
inc_compteur(): thread 0 compteur = 13 , deverouillage du mutex.
                  aucune condition d'arrêt, poursuite des opérations

inc_compteur(): thread 0 compteur = 14 , deverouillage du mutex.
inc_compteur(): thread 0 compteur = 15 , deverouillage du mutex.
inc_compteur(): thread 0 compteur = 16 , deverouillage du mutex.
inc_compteur(): thread 0 compteur = 17 , deverouillage du mutex.
inc_compteur(): thread 0 compteur = 18 , deverouillage du mutex.
inc_compteur(): thread 0 compteur = 19 , deverouillage du mutex.
inc_compteur(): thread 0 compteur = 20 , deverouillage du mutex.

Main(): Le traitement des 3 threads a pris fin.
                  fin des opérations. Tout est ok!
```

5.8 Fonctions supplémentaires

- La fonction « pthread_t pthread_self() » retourne l'identificateur du thread appelant.
- La fonction « int pthread_equal(pthread_t t1,pthread_t t2) » compare l'identificateur des deux threads t1 et t2. Elle retourne une valeur différente de 0 s'ils sont identiques.
- La fonction « int pthread_once(pthread_one_t* unefois, void (*init_routine) (void)) » exécute une seule fois la routine « init_routine » dans le process. Tout appel subséquent à cette routine n'aura aucun effet.
- La fonction « void pthread_yield() » force le thread appelant à renoncer son utilisation des ressources du processeur et se mettre dans la queue dans l'attente qu'il soit relancé.

5. Références

Livre : "Pthreads Programming", B. Nichols et al. O'Reilly and Associates.

De l'aide en ligne :

En français : http://mapage.noos.fr/emdel/pthreads.htm

En anglais : https://computing.llnl.gov/tutorials/pthreads/

La commande linux : « man pthread »