CHAPITRE 4 : Gestion des threads

1- Caractéristiques des threads

1.1- Concept de thread

Le concept de **thread** est apparu pour simplifier la mise en œuvre d'actions en **parallèle** dans un programme.
Dans un programme **séquentiel**, les instructions doivent être exécutées une après l'autre (jamais plusieurs simultanément) en suivant l'enchaînement défini par les structures de contrôle du programme.
Dans un programme **parallèle**, il existe une ou plusieurs parties du programme, constituées de blocs d'instructions qui doivent s'exécuter en parallèle.

Un processus initié à partir d'un programme **séquentiel**, possède à tout moment un seul **flux d'exécution**.
Un processus initié à partir d'un programme **parallèle**, possédera un seul **flux d'exécution** dans les parties séquentielles du programme (s'il y en a) mais plusieurs **flux d'exécution** dans les parties parallèles.

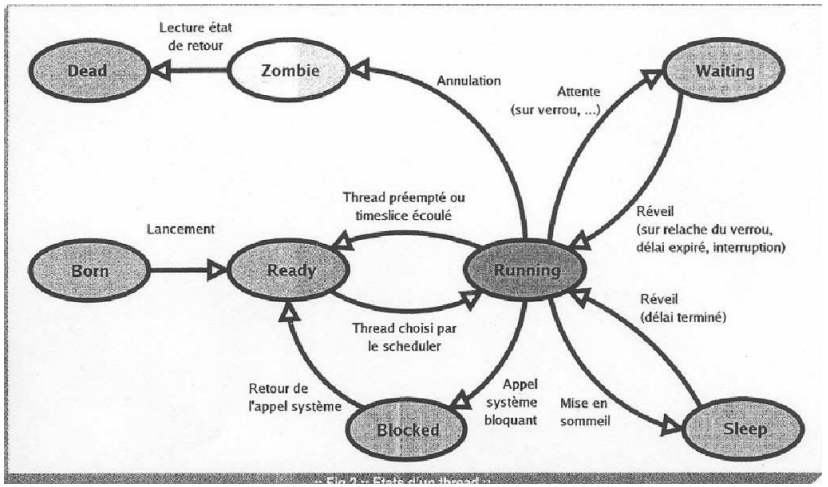
Ces flux d'exécution sont appelés des '**threads'**.

Les **threads** permettent donc de réaliser des sous tâches en parallèle à l'intérieur d'un processus et sont souvent qualifiés de sous-processus ou processus léger.
Un autre intérêt important des **threads** est que la zone mémoire privée contenant les données d'un processus est commune à tous les **threads** internes à ce processus.

Le noyau d'un système qui implémente les **threads** devra donc fournir des mécanismes de contrôle des **threads** similaires à ceux fournis pour les processus.

1.2- Etats d'un thread

La figure ci-dessous représente le diagramme d'état d'un thread.



1.3- Attributs d'un thread

Un thread est doté de différents attributs regroupés dans une structure de données de type **pthread_attr_t** .

Les attributs définis dans la norme Posix sont les suivants :

- **stackaddr et stacksize**

  ces attributs permettent de consulter et de modifier la taille et l'adresse du segment mémoire contenant la **pile système** et le **tas** privés du thread.

  Ce segment mémoire allouée au thread lors de sa création, a une taille fixe par défaut qui peut s'avérer insuffisante dans des traitements mettant en œuvre la récursivité ou l'allocation dynamique.

  Dans ce cas il faudra pouvoir augmenter la taille de ce segment mémoire et pour ce faire il sera souvent nécessaire de le reloger.

  La manipulation de ces attributs est délicate, c'est pourquoi pour modifier leur valeur par défaut, il faut que le propriétaire effectif du processus contenant le thread soit **root** (EUID = 0).

  Remarque :

  Pour l'instant, ces deux attributs ne sont ni consultables ni modifiables dans la bibliothèque LinuxThreads.

- **detachstate**

  cet attribut définit le comportement du thread lors de sa terminaison et peut prendre les valeurs suivantes :

constante symbolique	signification
PTHREAD_CREATE_JOIGNABLE	Lorsqu'il se termine, le thread génère une **valeur de retour** puis passe dans l'état **zombie** jusqu'à ce qu'un autre thread consulte cette valeur de retour. Après que cette valeur de retour ait été consultée, le système libère le segment mémoire contenant la **pile système** et le **tas** privés du thread et détruit le thread. C'est la valeur par **défaut** attribuée lors de la création d'un thread.
PTHREAD_CREATE_DETACHED	Si le thread est **indépendant** de tout les autres, il n'as pas besoin lorsqu'il se termine, de générer une **valeur de retour** et de passer dans l'état **zombie**. Le système peut libérer ses ressources privées et le détruire dès qu'il se termine.

- **schedpolicy**

cet attribut définit la méthode d'ordonnancement appliquée au thread et peut prendre les valeurs suivantes :

constante symbolique	signification
SCHED_OTHER	Ordonnancement classique (c'est pour Linux le principe du temps partagé qui est utilisé par le noyau pour ordonnancer les processus). C'est la valeur par **défaut** attribuée lors de la création d'un thread.
SCHED_RR	Séquencement **temps-réel** avec l'algorithme du tourniquet (Round Robin).
SCHED_FIFO	Ordonnancement **temps-réel** avec file d'attente (FIFO).

Remarques :

Les méthodes d' ordonnancement temps-réel (***SCHED_RR*** et ***SCHED_FIFO***) sont implémentées dans la bibliothèque LinuxThreads. Toutefois le noyau Linux actuel n'est pas un noyau **temps-réel strict**, il est du type **temps-réel large**. Donc l'ordonnancement d'un thread avec l'une ou l'autre de ces deux méthodes ne pourra assurer qu'un fonctionnement temps-réel approximatif.

Pour pouvoir modifier la valeur par défaut de cet attribut, il faut que le propriétaire effectif du processus contenant le thread soit **root** (EUID = 0).

- **schedparam**

cet attribut définit la priorité appliquée au thread et n'est significatif que pour les méthodes d'ordonnancement temps-réel (RR ou FIFO).

- **scope**

cet attribut n'est vraiment configurable que dans certains systèmes utilisant une implémentation hybride des threads reposant en partie sur un ordonnancement par le noyau et en partie sur une bibliothèque utilisateur indépendante du noyau.

constante symbolique	signification
PTHREAD_SCOPE_SYSTEM	Le thread est ordonnancé en concurrence avec tous les processus du système. Le séquenceur utilisé est donc celui du noyau.
PTHREAD_SCOPE_PROCESS	Le thread est ordonnancé en concurrence uniquement avec les autres threads du processus. Le séquenceur utilisé est dans ce cas une fonction fournie par la bibliothèque utilisateur des THREADS et qui est exécutée par le processus lui-même.

Remarque :

L'implémentation des threads de la bibliothèque LinuxThreads est basée uniquement sur un ordonnancement par le noyau et donc la valeur ***PTHREAD_SCOPE_PROCESS*** n'est pas **utilisable**.

- **inheritsched**

cet attribut indique si l'ordonnancement du thread doit être fait avec ses propres attributs **schedpolicy** et **schedparam** ou avec ceux du thread créateur.

constante symbolique	signification
PTHREAD_EXPLICIT_SCHED	Le thread est ordonnancé avec ses propres attributs. C'est la valeur par **défaut** attribuée lors de la création d'un thread.
PTHREAD_INHERIT_SCHED	Le thread est ordonnancé avec les attributs hérités du thread créateur.

Remarque :

Pour pouvoir modifier la valeur par défaut de cet attribut, il faut que le propriétaire effectif du processus contenant le thread soit **root** (EUID = 0).

2- Création et terminaison d'un thread

2.1- Identification d'un thread

Pour distinguer les différents threads créés à l'intérieur d'un processus, le système leur attribue un identifiant noté **TID** (thread identifier) de type **pthread_t** :

L'appel système permettant d'obtenir cet identifiant est le suivant:

```
# include <pthread.h>

    pthread_t  pthread_self ( void ) ;
```

cette fonction retourne l'identifiant du thread appelant (TID) ou le code d'erreur **-1** en cas d'erreurs.

Remarques :

Dans la bibliothèque LinuxThreads, le type **pthread_t** est défini comme un entier naturel :

```
    typedef  unsigned long int pthread_t ;
```

on peut donc manipuler le TID d'un thread comme un entier (affectation, comparaison, etc …)

Attention ! la représentation du type **pthread_t** n'est pas forcement la même dans les systèmes Unix.

2.2- Création d'un thread

Normalement un thread ne peut être créé que par un autre thread.
Lorsqu'un processus est crée, il faut donc qu'il existe déjà un thread pour pouvoir créer d'autres threads.
C'est pourquoi le processus lui-même est considéré comme un thread (**main thread**).

La création d'un thread (autre que le **main thread**) est réalisée par l'appel de la fonction **pthread_create( )** :

```
# include <pthread.h>

typedef void * ( * PtrFct )( void * );

    int  pthread_create( pthread_t * idThread, pthread_attr_t * attributs, PtrFct traitement, void * parametre);
```

attributs est un pointeur sur une variable contenant les attributs affectés au thread qui va être créé
(si ce pointeur vaut NULL, le thread est créé avec des valeurs d'attributs par défaut)

traitement est la fonction que devra **exécuter** le nouveau thread

parametre est un pointeur sur le **paramètre effectif** qui sera transmis à la fonction *traitement*

Si la création réussit, l'identifiant (TID) du thread est stocké dans la variable pointée par *idThread*
et la fonction retourne **0.** En cas d'erreur**,** la fonction retourne un code d'erreur non nul.

Remarques :

A la différence des processus, un thread ne possède pas dans ses attributs le TID de son parent et la bibliothèque LinuxThreads ne fournit pas de fonction similaire à **getppid( )**. La notion de groupe n'existe pas pour les threads.

Exemple : création d'un thread avec les attributs par défaut

```
…..
int err, nbre;
pthread_t  idThread;
…..
nbre = 1234 ;
err = pthread_create( & idThread, NULL, traitTH, (void *) & nbre);    /* le paramètre effectif nbre */
if  (err != 0 )                                                       /* est transmis par adresse    */
     /* la création du thread fils a échoué*/                         /*  il est donc du type int *  */
     { perror("echec pthread_create ") ; …. }                         /* il doit être transtypé  en void *  */

/* suite du traitement du parent */
……..
```

```
/* fonction définissant le traitement du thread */
void * traitTH( void * num )
{
int val ;
……..
val = *( (int *) num) ;     /* le paramètre formel num est un pointeur     */
.........                   /* du type void * , il faut d'abord le transtyper */
                            /* dans le type int * du paramètre effectif    */
                            /* avant de faire l'adressage indirect         */
……..
}
```


2.3- Initialisation des attributs d'un thread

Pour créer un thread avec un ou plusieurs attributs ayant des valeurs différentes des valeurs par défaut, il faut en premier lieu configurer une variable de type **pthread_attr_t** avec les valeurs des attributs choisies.
Ensuite on peut appeler la fonction **pthread_create** en lui passant en paramètre un pointeur sur cette variable.

La marche à suivre est la suivante :

► déclarer une variable de type **pthread_attr_t** :

```
    pthread_attr_t  attributs ;
```

► initialiser cette variable avec les valeurs par défaut des attributs :

```
    err = pthread_attr_init (  & attributs );
    if  (err != 0 )
         { perror("echec pthread_attr_init ") ; …. }
```

► consulter et/ou modifier la valeur de chacun des attributs qui le nécessitent à l'aide des fonctions suivantes :

```
    int valeurAttribut ;

    /* les valeurs possibles pour la variable valeurAttribut sont les constantes symboliques */
    /* figurant dans le tableau du § 1.3 correspondant à l'attribut invoqué */

    ..........................................
```



```
err = pthread_attr_getdetachstate (  & attributs , & valeurAttribut );
err = pthread_attr_setdetachstate (  & attributs , valeurAttribut );

err = pthread_attr_getschedpolicy (  & attributs , & valeurAttribut );
err = pthread_attr_setschedpolicy (  & attributs , valeurAttribut );

err = pthread_attr_getinheritsched (  & attributs , & valeurAttribut );
err = pthread_attr_setinheritsched (  & attributs , valeurAttribut );

err = pthread_attr_getscope (  & attributs , & valeurAttribut );
err = pthread_attr_setscope (  & attributs , valeurAttribut );

..........................................................

struct  sched_param priorite ;

/* le type sched_param est défini comme :  struct  sched_param  { int  sched_priority; }        */
/* la valeur de la priorité d'ordonnancement d'un thread n'est modifiable que pour les méthodes */
/* d'ordonnancement temps-réel (RR ou FIFO) et doit être comprise entre 1 et 99                 */
/* EX : pour affecter la priorité 35 */

priorite.sched_priority = 35 ;

err = pthread_attr_getschedparam (  & attributs , & priorite );
err = pthread_attr_setschedparam (  & attributs , priorite );

..........................................................
if  (err != 0 )
{ perror("echec pthread_attr_XXXXX ") ; …. }
```

► créer le thread :

```
…..
int err , nbre;
pthread_t  idThread;
pthread_attr_t  attributs ;
......

/* initialisation des attributs */

…..
nbre = 1234 ;
err = pthread_create( & idThread, & attributs, traitTH, (void *) & nbre);
if  (err != 0 )
      /* la création du thread fils a échoué */
      { perror("echec pthread_create ") ; …. }

/* suite du traitement du parent */
……..
```


2.4- Terminaison d'un thread

Un thread peut se terminer de 2 façons :

- **normalement** par l'appel d'une des fonctions **pthread_exit( )** ou **return( )** ou par la terminaison normale de son traitement.
- **anormalement** parce qu'il est annulé ou supprimé (**canceled**) par un autre thread.
  (le mécanisme d'annulation d'un thread sera abordé en détail dans la suite du cours).

```
# include <pthread.h>

        void  pthread_exit ( void * retVal ) ;

        void  return ( void * retVal ) ;
```

retVal est un pointeur sur une variable contenant les **informations retournées** par le thread ou le pointeur NULL si le thread ne retourne aucune information.

Dans le cas où le thread retourne une information, la variable peut être d'un type quelconque, à condition de **transtyper** le pointeur sur cette variable dans le type **void *** .

Rappels :

- **terminaison normale d'un thread "joignable"**

  l'attribut **detachstate** a la valeur ***PTHREAD_CREATE_JOIGNABLE***

  La fonction **pthread_exit** (ou **return**) termine l'exécution du thread, puis libère toutes les ressources privées du thread.
  Le thread passe ensuite dans l'état **zombie** jusqu'à ce qu'un autre thread récupère les informations qu'il a retournées.
  Après que les informations retournées ont été récupérées, le système libère le segment mémoire contenant la **pile système** et le **tas** privés du thread et détruit le thread.

  La fonction **pthread_join** qui permet à un thread de récupérer les informations retournées par un autre thread sera décrite ultérieurement.

- **terminaison normale d'un thread "détaché"**

  l'attribut **detachstate** a la valeur ***PTHREAD_CREATE_DETACHED***

  La fonction **pthread_exit** (ou **return**) termine l'exécution du thread, puis libère toutes les ressources privées du thread.

  Le thread ne passe pas dans l'état zombie, le système libère le segment mémoire contenant la **pile système** et le **tas** privés du thread et détruit le thread immédiatement.

Exemple : terminaison d'un thread "joignable"

```
/* fonction définissant le traitement du thread */
void * traitTH( void * num )
{
int val ;
static int  retVal ;        /* la variable retVal  doit être déclarée avec l'attribut static      */
                            /*  sinon elle sera allouée dans la pile système et elle disparaîtra  */
                            /* quand la fonction traitTH se terminera  */

……..
val = *( (int *) num) ;     /* le paramètre formel num est un pointeur du type void *              */
                            /* il faut d'abord le transtyper dans le type int * du paramètre effectif */
                            /* avant de faire l'adressage indirect                                 */
……..

retVal = val * 10 ;

……..

pthread_exit ( (void * )& retVal ) ;  /* & retVal est un pointeur du type int *      */
                                      /* il faut le transtyper dans le type void *   */
}
```

2.5- Attente de la terminaison d'un autre thread

Un thread **A** peut **suspendre** son exécution en attendant qu'un autre thread **B** se termine, en appelant la fonction **pthread_join ( )**.

```
# include <pthread.h>

    int  pthread_join ( pthread_t  idThread ,void ** pRapport ) ;
```

Si le thread **B** de TID = *idThread* est **"joignable"** (son attribut **detachstate** = ***PTHREAD_CREATE_JOIGNABLE***), le thread **A** va **suspendre** son exécution en attendant que le thread **B** se termine.

Si *pRapport* est différent du pointeur NULL, la valeur retournée par le thread **B** sera stockée dans la variable pointée par *pRapport* .

Si le thread **B** est **"détaché"** (son attribut **detachstate** = ***PTHREAD_CREATE_DETACHED***) ou s'il n'existe pas, la fonction retourne immédiatement un code d'erreur.

En cas de succès**,** la fonction retourne **0**, sinon elle retourne un code d'erreur non nul.

Remarques :

Un seul thread au plus, peut attendre la terminaison d'un autre thread.
Appeler la fonction **pthread_join** sur un thread dont un autre thread attend déjà la fin, renvoie une erreur.

Lorsqu'un thread **"joignable"** se termine, le système ne libère pas le segment mémoire contenant la **pile système** et le **tas** privés du thread tant qu'un autre thread ne le joint pas en appelant la fonction **pthread_join**.

Aussi la fonction **pthread_join** doit être appelée autant de fois que nécessaire pour contrôler la terminaison de tous les threads **"joignables"** afin d'éviter les "fuites" de mémoire.

Exemple : Attente de la terminaison d'un autre thread

```
/* fonction définissant le traitement du thread  A */
void * traitTH( void * num )
{
int err ;
pthread_t   idThB ;
void * ptrRetVal ;
int retVal ;
……..

err = pthread_join ( idThB, & ptrRetVal ) ;
if  (err != 0 )
     { perror("echec pthread_join ") ; …. }

retVal = * (( int * ) ptrRetVal ) ;     /* la valeur retournée par le thread B est un pointeur sur un int   */
                                        /* il faut transtyper le pointeur ptrRetVal dans le type int *       */

……..
}
```


2.6- Détachement d'un Thread

Un thread **"joignable"** peut pendant son exécution vouloir passer dans l'état **"détaché"**, en appelant la fonction **pthread_detach ( )**.

Contrairement aux processus, il n'y a pas de notion de hiérarchie ni de permissions entre threads, donc n'importe quel thread **A** peut invoquer cette fonction pour faire passer un autre thread **B** dans l'état **"détaché"**.

```
# include <pthread.h>

    int  pthread_detach ( pthread_t  idThread  ) ;
```

idThread est le TID du thread **B** destinataire

la fonction retourne en erreur si le thread destinataire n'existe pas ou s'il est déjà **"détaché".**

En cas de succès**,** la fonction retourne **0**, sinon elle retourne un code d'erreur non nul.

2.7- Annulation d'un Thread

L'annulation est un mécanisme par lequel un thread peut interrompre l'exécution d'un autre thread.
Plus précisément, un thread peut envoyer une **requête d'annulation** à un autre thread qui selon sa configuration peut soit **ignorer** la requête, soit la prendre en compte en **se terminant immédiatement**, soit enfin **différer** la prise en compte de la requête.

Ce dernier comportement signifie que le thread récepteur continuera son exécution jusqu'à atteindre un point précis du programme (appelé **point d'annulation**) où il prendra en compte la requête.

La notion de **point d'annulation** sera décrite ultérieurement.

Exemple de l'intérêt du mécanisme d'annulation :

On veut rechercher l'occurrence d'une valeur donnée dans un gros tableau.

Si on applique une méthode de recherche parallèle, on découpera le tableau en plusieurs tranches consécutives disjointes (2, 3, 4, …) et on lancera sur chaque tranche, un thread qui exécutera la recherche d'un élément dans un sous-tableau.

Chaque thread disposera de ses 2 variables privées **trouvé** et **rang** pour matérialiser le résultat de sa recherche.

Toutes ces variables seront **globales** afin q'un **thread superviseur** puisse les consulter en permanence.

Si le tableau contient au moins une occurrence de la valeur recherchée, le thread qui à trouvé le premier, va se terminer. A ce moment là, il est inutile que les autres threads continuent à chercher, le **thread superviseur** peut donc tous les annuler.

► **envoi d'une requête d'annulation**

Un thread **A** peut envoyer une **requête d'annulation** à un autre thread B en appelant la fonction **pthread_cancel ( ) :**

```
# include <pthread.h>

    int  pthread_cancel ( pthread_t  idThread ) ;
```

idThread est le TID du thread **B** destinataire

En cas de succès**,** la fonction retourne **0**, sinon elle retourne un code d'erreur non nul.

Remarque :

On rappelle qu'il n'y a pas de notion de hiérarchie ni de permissions entre threads, donc n'importe quel thread peut envoyer une **requête d'annulation** à un autre thread.

► **réception d'une requête d'annulation**

Lorsqu'un thread reçoit une requête d'annulation, il peut selon sa configuration soit :

- l'**ignorer**.
- l'**accepter** et dans ce cas soit :
  - se **terminer immédiatement.**
  - **différer** sa prise en compte et continuer son exécution jusqu'à atteindre un **point d'annulation** à l'endroit duquel il se terminera.

La fonction **pthread_setcancelstate( )** permet de configurer le comportement d'un thread lorsqu'il reçoit une requête d'annulation, à savoir s'il l'accepte ou s'il l'ignore.

```
# include <pthread.h>

    int  pthread_ setcancelstate ( int  etat_annulation ,int* ancien_etat ) ;
```

etat_annulation peut prendre les valeurs suivantes :

constante symbolique	signification
PTHREAD_CANCEL_ENABLE	Le thread acceptera les requêtes d'annulation. C'est la valeur par **défaut** attribuée lors de la création d'un thread.
PTHREAD_CANCEL_DISABLE	Le thread ne tiendra pas compte des requêtes d'annulation.

Si *ancien_etat* est différent du pointeur NULL, l'état précédent sera stocké dans la variable pointée par *ancien_etat* et pourra être restauré ultérieurement si nécessaire.

Remarque :

Si un thread reçoit une ou plusieurs requêtes d'annulation pendant qu'il est dans l'état ***PTHREAD_CANCEL_DISABLE*** , elles ne sont pas **mémorisées**, contrairement aux signaux par exemple.
De ce fait, si plus tard le thread repasse dans l'état ***PTHREAD_CANCEL_ENABLE*** , il ne se terminera pas.

La fonction **pthread_setcanceltype( )** permet de configurer le comportement d'un thread lorsqu'il a accepté une requête d'annulation, à savoir s'il se termine immédiatement ou s'il continue jusqu'à un **point d'annulation**.

```
# include <pthread.h>

    int  pthread_ setcanceltype ( int  type_annulation ,int* ancien_type ) ;
```

type_annulation peut prendre les valeurs suivantes :

constante symbolique	signification
PTHREAD_CANCEL_DEFERRED	Le thread ne se terminera qu'en atteignant un point d'annulation. C'est la valeur par **défaut** attribuée lors de la création d'un thread.
PTHREAD_CANCEL_ASYNCHRONOUS	Le thread se terminera dès réception de la requête.

Si *ancien_type* est différent du pointeur NULL, la valeur précédente sera stockée dans la variable pointée par *ancien_type* et pourra être restaurée ultérieurement si nécessaire.

Intérêt du mode "différé"

Sauf cas particulier, arrêter un thread brutalement va entraîner un dysfonctionnement grave ou même fatal de l'application (corruption de données, blocage de threads, etc …).

De même comme on l'a vu ci-dessus, ignorer une requête d'annulation peut faire que le thread ne s'arrêtera pas.

Il est donc nécessaire de disposer d'un mécanisme qui permet au thread de mémoriser la requête d'annulation puis de terminer les opérations en cours avant de se terminer sur un point d'annulation.

Les points d'annulation

La norme Posix propose 4 fonctions qui constituent des points d'annulation, c'est-à-dire des fonctions dans lesquelles un thread se terminera brutalement s'il est dans l'état ***PTHREAD_CANCEL_ENABLE*** avec le type d'annulation ***PTHREAD_CANCEL_DEFERRED*** et q'une requête d'annulation est en attente.

► **pthread_ join** ( ) qui a été présentée au § 2.5

► **pthread_ cond_wait** ( ) et **pthread_ cond_timedwait** ( ) qui seront présentées ultérieurement

► void **pthread_ testcancel** (void)

Cette dernière fonction teste si une requête d'annulation est en attente. Si c'est le cas elle termine le thread sinon elle ne fait rien et le thread continue son exécution.
On pourra donc répartir des appels à cette fonction aux endroits du programme du thread où on est sur qu'une annulation ne présente pas de danger.

CHAPITRE 5 : Synchronisation des threads

1- Introduction

Le développement d'applications **multi-threads** induit deux types de problématiques :

- celle de la décomposition de l'application en **tâches parallèles**, laquelle relève du domaine de l'algorithmique parallèle que nous n'aborderons pas ici.

- celle de la **synchronisation** et de la **communication** entre les différents threads mis en œuvre pour implémenter l'application.

Un des intérêts majeurs des **threads** est que tous les **threads** internes à un processus peuvent accéder aux **variables globales** de ce processus.

De ce fait, les mécanismes de communication utilisés pour les processus, tels que :

- les files d'attentes
- les segments de mémoire partagés
- les boites à lettres

dont la mise en œuvre est assez complexe, ne sont plus nécessaires puisque la communication entre threads peut se faire simplement par le biais de variables globales**.**

Par contre, les problèmes de synchronisation entre threads sont de même nature que ceux rencontrés dans les applications multi-processus :

- contrôler l'accès à un objet partagé ou l'utilisation d'une ressource commune pour garantir un fonctionnement cohérent de l'application
- assurer le séquençage des tâches qui sont liées par des relations de dépendances
- éviter les inter blocages entre threads

A titre d'exemple, voici quelques problèmes classiques de synchronisation :

Exemple1 : problème d'accès incontrôlé à une variable globale

A un instant donné, deux threads veulent "simultanément" décrémenter un compteur déclaré comme variable globale et donc accessible par les deux threads.
Le premier thread qui est dans l'état actif, lit le contenu N du compteur et le charge dans un registre du processeur puis décrémente ce registre qui prend la valeur N-1.
A ce moment là, l'ordonnanceur le suspend et réactive le second thread en lui allouant le processeur.
Le second thread lit le contenu du compteur qui vaut toujours N et le charge dans un registre du processeur puis décrémente ce registre qui prend la valeur N-1.
Ensuite il écrit le contenu du registre dans le compteur qui prend la valeur N-1.
A ce moment là, l'ordonnanceur le suspend et réactive le premier thread en lui allouant le processeur.
Le premier thread reprend son traitement là où il l'avait arrêté et écrit le contenu du registre qui vaut N-1 dans le compteur qui prend donc la valeur N-1.

Conclusion : le compteur a été décrémenté de 1 au lieu de 2.

Exemple2 : problème d'interblocage

Un thread veut accéder à une ressource partagée et il est bloqué en attendant qu'elle soit libérée.

Il n'existe aucun thread susceptible de libérer la ressource car ils sont tous terminés ou ils sont eux même bloqués en attente de la libération d'autres ressources.

Tous ces threads se bloquent entre eux et l'application est stoppée indéfiniment.

Exemple3 : problème de séquençage

Un thread A produit une suite de nombres qu'il doit communiquer à un thread B qui doit les traiter.
Une variable globale V sera utilisée pour que le thread A communique un nombre qu'il a produit au thread B qui le traitera.

Quand le thread A voudra écrire le nombre dans la variable V, il faudra qu'il soit sur que le nombre précédent a été lu par le thread B, sinon il y aura perte d'une donnée.

Quand le thread B voudra lire le nombre dans la variable V, il faudra qu'il soit sur que le thread A a bien écrit le nombre suivant dans la variable V, sinon il y aura traitement en double de la même donnée.

Notion de section critique

On appelle section critique une séquence d'instructions dans laquelle le thread peut rentrer en conflit avec d'autres threads pour accéder à une ressource partagée.

Si on ne contrôle pas l'entrée du thread dans cette section critique, il y aura sûrement un dysfonctionnement de l'application (données corrompues, interblocages, etc …).

Il est donc indispensable de bien synchroniser l'entrée d'un thread dans une section critique pour éviter ce type de problèmes.

Une solution au problème de la section critique doit satisfaire aux trois besoins suivants :

- **l'exclusion mutuelle** : si un thread exécute sa section critique aucun autre thread ne peut exécuter sa section critique.

- **le déroulement** : si des threads sont en attente de rentrer dans leur section critique, seuls les threads qui sortent de leur section critique peuvent décider desquels threads en attente seront autorisés à rentrer dans leur section critique. Cette décision ne peut pas être reportée indéfiniment.

- **l'attente limitée** : il faut limiter le nombre de fois q'un thread se voit refuser l'entrée dans sa section critique, avant que l'autorisation lui soit accordée, sinon il y aura un état de **famine** pour ce thread.

2- L'exclusion mutuelle

Pour mettre en œuvre l'exclusion mutuelle interthreads, la bibliothèque LinuxThreads fournit un type de variable appelée **mutex** (MUtual Exclusion) qui sert de **verrou** pour contrôler l'accès à une ressource partagée.

Une variable **mutex** (ou mutex en abrégé) ne peut être que dans 2 états :

- l'état **verrouillé** qui signifie que le mutex est détenu par un thread et qu'il interdit l'accès à la ressource à tous les autre threads
- l'état **déverrouillé** (ou disponible) qui signifie que le mutex n'est détenu par aucun thread et qu'il peut être verrouillé par le premier thread qui en fera la demande

Une variable **mutex** n'est pas une variable classique : un thread ne peut ni la tester ni la modifier.
Il ne peut y accéder que par le biais des deux fonctions suivantes:

- Une fonction qui **demande à verrouiller** le mutex et qui bloque le thread appelant jusqu'à ce qu'il l'obtienne.
- Une fonction qui **libère** le mutex.

Remarques :

- Un mutex ne peut être détenu que par un thread à la fois.
- Si un thread demande à verrouiller un mutex qui est dèjà verrouillé (donc détenu par un autre thread), il restera bloqué au moins jusqu'à ce que le mutex soit déverrouillé.
- Si plusieurs threads sont **bloqués** sur un mutex, un seul parmi eux sera autorisé à le verrouiller, les autres continueront à rester bloqués.

2.1- Déclaration et initialisation d'un mutex

Les mutex sont des variables du type **pthread_mutex_t**.
Avant toute utilisation, un mutex doit être initialisé dans l'état **déverrouillé.**
Cette initialisation peut être faite **statiquement** lors de la déclaration du mutex, avec la constante symbolique suivante :

pthread_mutex_t *variable_Mutex* = ***PTHREAD_MUTEX_INITIALIZER*** ;

Cette constante symbolique définit un **mutex 'normal'**. La bibliothèque LinuxThreads fournit deux autres constantes symboliques définissant deux types de mutex non standards, qui ne seront pas présentés ici et qui ne doivent pas être employés si on veut que le programme soit portable.

Cette initialisation peut aussi être faite en utilisant la fonction **pthread_mutex_init ( )** de la façon suivante :

pthread_mutex_init (*variable_Mutex, NULL***);**

Une variable mutex peut être allouée **dynamiquement** (avec la fonction **malloc**). Dans ce cas on ne peut l'initialiser qu'en utilisant la fonction **pthread_mutex_init ( ).**

Remarques :

- Étant donné que les mutex servent à synchroniser différents threads, il faut les déclarer comme des variables **globales** si on veut qu'ils soient utilisables par tous les threads du processus.
- On peut limiter l'usage d'un mutex à un thread et à sa descendance en déclarant ce mutex comme une variable **locale statique** de ce thread.

2.2- Verrouillage d'un mutex

Un thread peut **demander** à verrouiller un mutex en appelant la fonction **pthread_mutex_lock ( )** :

```
# include <pthread.h>

    int  pthread_mutex_lock ( pthread_mutex_t  * mutex ) ;
```

Si le mutex est disponible, il est immédiatement verrouillé et attribué au thread appelant.

Si le mutex est déjà verrouillé (donc détenu par un autre thread), le thread appelant restera bloqué au moins jusqu'à ce que le mutex soit déverrouillé.
En cas de succès**,** la fonction retourne **0**, sinon elle retourne un code d'erreur non nul.

Remarques :

- Le thread appelant peut rester bloqué **indéfiniment** si le thread qui détenait le mutex en dernier s'est terminé sans le déverrouiller.
- Si le thread appelant détient déjà le mutex, il s'**interbloque** lui même **indéfiniment** car il n'y a que lui qui pourrait déverrouiller le mutex.
- Si le thread appelant détient déjà le mutex, il s'**interbloque** lui même **indéfiniment** car il n'y a que lui qui pourrait déverrouiller le mutex.
- Pour éviter les situations de blocage, on peut utiliser la fonction **pthread_mutex_trylock ( )** qui fonctionne comme **pthread_mutex_lock** si le mutex est disponible, mais qui retourne un code d'erreur non nul si le mutex est déjà verrouillé, sans bloquer le thread appelant.

2.3- Libération d'un mutex

Un thread peut **libérer** un mutex qu'il détient en appelant la fonction **pthread_mutex_unlock ( )** :

```
# include <pthread.h>

    int  pthread_mutex_unlock ( pthread_mutex_t  * mutex ) ;
```

En cas de succès**,** la fonction retourne **0**, sinon elle retourne un code d'erreur non nul.

Remarque :

Pour les **mutex normaux,** la fonction ne contrôle pas si le thread appelant détient ou pas le mutex.

Donc un mutex normal peut être libéré par un thread quelconque.

Ce comportement n'est pas portable et ne doit pas être utilisé.

La fonction **pthread_mutex_unlock ( )** ne devra être appelée que par le thread qui détient le mutex

3- L'attente de conditions

La bibliothèque LinuxThreads fournit un type de variable appelée "**condition**" qui permet de mettre un ou plusieurs thread en attente d'un évènement produit par un autre thread.

Pour ce faire, on associe à l'évènement une variable **condition** qui restera dans l'état **faux** tant que l'évènement ne se sera pas produit et qui passera dans l'état **vrai** lorsque que l'évènement se sera produit.

Une variable **condition** n'est pas une variable classique : un thread ne peut ni la tester ni la modifier.
Il ne peut y accéder que par le biais des deux fonctions suivantes:

- Une fonction qui **met en attente** le thread appelant jusqu'à ce que la condition soit réalisée.
- Une fonction qui permet au thread qui à produit l'évènement de **signaler** à des threads qui sont en attente que la condition est réalisée.

Une variable **condition** étant accédée par plusieurs threads (au moins deux), il pourrait se produire la situation de blocage suivante :

- Un thread A est en train d'exécuter la fonction de **mise en attente.** Juste avant de s'endormir pour attendre que la condition soit réalisée, l'ordonnanceur le suspend et réactive un thread B en lui allouant le processeur.
- Le thread B termine d'exécuter la fonction qui **signale** que la condition est réalisée.
- Le thread A qui n'était pas encore endormi pour attendre que la condition soit réalisée, ne sera pas informé que la condition est réalisée.
- Lorsque l'ordonnanceur va réactiver le thread A, il va s'endormir pour attendre que la condition soit réalisée. Le thread B va croire que le thread A continue son traitement alors qu'il est bloqué.

Pour éviter ces éventuels problèmes de concurrence d'accès à une variable condition, on doit **obligatoirement** associer un **mutex** à une variable **condition.**

Le comportement des deux threads sera dans ce cas le suivant :

Thread attendant la réalisation de la condition	Thread signalant la réalisation de la condition
Il demande à verrouiller le mutex associé à la condition	
Dès qu'il a obtenu le mutex, il appelle la fonction qui le mettra en attente de la condition	
Il libère le mutex et s'endort	
	Il demande à verrouiller le mutex associé à la condition
	Dès qu'il a obtenu le mutex, il appelle la fonction qui signale que la condition est réalisée
Il se réveille et il demande à verrouiller le mutex	
	Il libère le mutex
Dès qu'il a obtenu le mutex, il termine la fonction	
Il libère le mutex	

3.1- Déclaration et initialisation d'une condition

Les **conditions** sont des variables du type **pthread_cond_t**.

Avant toute utilisation, une condition doit être initialisé dans l'état **faux.**
Cette initialisation peut être faite **statiquement** lors de la déclaration de la variable condition, avec la constante symbolique suivante :

pthread_cond_t *variable_Condition* = ***PTHREAD_COND_INITIALIZER*** ;

Cette initialisation peut aussi être faite en utilisant la fonction **pthread_cond_init ( )** de la façon suivante :

pthread_cond_init (*variable_Condition, NULL***);**

Une variable condition peut être allouée **dynamiquement** (avec la fonction **malloc**). Dans ce cas on ne peut l'initialiser qu'en utilisant la fonction **pthread_cond_init ( ).**

Remarques :

- Étant donné que les variables conditions servent à synchroniser différents threads, il faut les déclarer comme des variables **globales** si on veut qu'elles soient utilisables par tous les threads du processus.
- On peut limiter l'usage d'une variable condition à un thread et à sa descendance en déclarant cette variable condition comme une variable **locale statique** de ce thread.

3.2- Attente de la réalisation d'une condition

Un thread peut attendre la réalisation d'une condition en appelant la fonction **pthread_cond_wait ( )** :

```
# include <pthread.h>

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

Cette fonction **libère** le mutex *mutex* passé en paramètre et **endort** le thread.
Ces deux opérations sont réalisées de façon **atomique** (indivisible), ce qui fait que le mutex n'est libéré qu'après que le thread soit endormi.

Si le thread qui doit signaler la condition, **demande à verrouiller** le mutex avant d'appeler la fonction qui signale la condition, on est **assuré** que le thread qui attend est dèjà endormi quand la condition est signalée.

Si le thread est **seul en attente** de la condition, il sera réveillé dés que la condition *condition* sera réalisée.
S'il y a **plusieurs thread en attente**, il faudra en plus qu'il soit choisi par le noyau parmi les autres threads.
Il devra alors attendre une prochaine réalisation de la condition.

Dés que le thread est réveillé, la fonction reprend son cours et **demande à verrouiller** le mutex *mutex*.
Quand le thread a obtenu le mutex, la fonction se termine.

Cette fonction ne retourne jamais de code d'erreur, elle retourne toujours **0**.

Rappel :

la fonction **pthread_cond_wait ( )** est un point d'annulation.

Il existe une fonction d'attente temporisée **pthread_cond_timedwait ( )**, qui permet de limiter le délai d'attente de la réalisation d'une condition :

```
    int  pthread_cond_timedwait ( pthread_cond_t  * condition,  pthread_mutex_t  * mutex,
                                  const struct timespec * date ) ;
```

Attention *date* n'est pas la durée de l'attente, mais **la date limite** à laquelle l'attente prendra fin si la condition ne s'est pas réalisée entre temps.

La structure **timespec** est constituée d'un champ contenant le nombre de **secondes** écoulées depuis le 1er Janvier 1970, et un autre champ indiquant le complément en **nanosecondes**.

La date actuelle peut être obtenue avec les fonctions **time( )** ou **gettimeofday( ).**
La fonctions **time( )** fournit simplement le nombre de **secondes** écoulées depuis le 1er Janvier 1970, alors que la fonction **gettimeofday( )** est plus précise et fournit en plus le complément en **microsecondes**.

En cas de succès**,** la fonction retourne **0**, sinon elle retourne un code d'erreur non nul.

Rappel :

la fonction **pthread_cond_timedwait ( )** est un point d'annulation.

3.3- Signalement de la réalisation d'une condition

Un thread peut signaler la réalisation d'une condition aux threads qui sont en attente de cette condition en appelant la fonction **pthread_cond_signal ( )** :

```
# include <pthread.h>

    int  pthread_cond_signal ( pthread_cond_t  * condition ) ;
```

S'il y a plusieurs threads en attente, seul l'un d'entre eux est relancé mais on ne peut pas savoir lequel.

Si un thread veut relancer **tous** les threads en attente d'une condition, il doit dans ce cas appeler la fonction **pthread_cond_broadcast ( )** :

```
    int  pthread_cond_ broadcast ( pthread_cond_t  * condition ) ;
```

S'il n'y a aucun thread en attente, ces deux fonctions sont sans effet.

Ces deux fonctions ne retournent jamais de code d'erreur, elles retournent toujours **0**.

Consigne importante :

Pour être sur que tous les threads en attente de la condition sont bien endormis lorsque la condition est signalée, l'appel de ces fonctions devra **systématiquement** être encadré par une **demande de verrouillage** et par une **libération** du mutex associé à la condition.

Exemple : Un thread A contrôle la température fournie par une sonde et alerte un thread B chaque fois que la température devient inférieure à 19°. Le thread B est chargé de réguler un radiateur pour maintenir la température le plus proche possible de 19°.

```
/* déclarations globales de la variable condition et du mutex associé */
pthread_cond_t   condition_alarme = PTHREAD_COND_INITIALIZER ;
pthread_mutex_t   mutex_alarme = PTHREAD_MUTEX_INITIALIZER ;


/*  traitement du thread A : surveiller la température  */
void   threadTemperature ( void * rien )
{
int temperature ;
int delai = 300;          /*  le delai minimum entre deux mesures est de 5 minutes */
while ( 1 )
  {
    capterTemperature ( & temperature );          /*  cette procédure retourne la température en degrés */
    if ( temperature < 19 )                       /*  mesurée par la sonde  */
     {
        pthread_mutex_lock( & mutex_alarme );
        pthread_cond_signal( & condition_alarme );
        pthread_mutex_unlock( & mutex_alarme );
     }
    sleep ( delai );
  }
```

```
}
/*  traitement du thread B : réguler le radiateur */

void   threadReguler ( void * rien )
{
int duree =  300;
while ( 1 )
  {
     pthread_mutex_lock( & mutex_alarme );
     pthread_cond_wait( & condition_alarme , & mutex_alarme );
     pthread_mutex_unlock( & mutex_alarme );

     allumerRadiateur ( );
     sleep ( duree );
     arreterRadiateur ( );

  }
}
```


Consigne importante :

La norme Posix.1c autorise la fonction **pthread_cond_wait( )** à se terminer de façon impromptue, même si la condition correspondante n'est pas réalisée.

Par exemple, cela peut se produire si un thread **signale** une condition **sans avoir vérifié** qu'elle est réalisée.

Il faut donc **impérativement** placer l'appel de la fonction **pthread_cond_wait( )** dans une boucle qui rappelle cette fonction tant que la condition n'est pas réalisée.

Dans notre exemple ci-dessus, il faudrait d'abord déclarer ***temperature*** en variable **globale**.

Ensuite, il faudrait modifier l'appel de la fonction **pthread_cond_wait( )** de la façon suivante :

```
pthread_mutex_lock( & mutex_alarme );
while ( temperature >= 19 )
  {
     pthread_cond_wait( & condition_alarme , & mutex_alarme );
  }
pthread_mutex_unlock( & mutex_alarme );
```


4- Les sémaphores Posix.1b

La bibliothèque LinuxThreads fournit un mécanisme de synchronisation appelé **sémaphores** qui appartient en fait à la norme Posix.1b (temps-réel).

Un **sémaphore** est une variable d'un type particulier qui permet de limiter l'entrée dans une section critique de **plusieurs** threads ou de limiter l'accès à une ressource partageable par **plusieurs** threads.

On peut considérer un **sémaphore** comme un **distributeur de tickets** dont le fonctionnement est le suivant:

- quand un thread veut entrer dans sa section critique (ou accéder à la ressource partageable) il demande un ticket au sémaphore
- si le sémaphore possède au moins 1 ticket, il le donne au thread qui peut entrer dans sa section critique
- si le sémaphore n'a plus de tickets, le thread est bloqué et doit attendre q'un ou plusieurs autres threads sortent de leur section critique et rendent leur ticket au sémaphore.

Un sémaphore qui ne dispose que de 1 ticket au plus est équivalent à un mutex.

Une variable **sémaphore** n'est pas une variable classique : un thread ne peut ni la tester ni la modifier.
Il ne peut y accéder que par le biais des deux fonctions suivantes:

- Une fonction qui **demande un ticket** au **sémaphore** et qui bloque le thread appelant jusqu'à ce qu'il l'obtienne.
- Une fonction qui **restitue un ticket** au **sémaphore**.

4.1- Déclaration et initialisation d'un sémaphore

Les **sémaphores** sont des variables du type **sem_t**.
Avant toute utilisation, un **sémaphore** doit être initialisé avec le nombre **maximum de tickets** disponibles**.**
Cette initialisation est faite avec la fonction **sem_init ( ) :**

```
# include < semaphore.h>

    sem_t  semaphore ;

    int  sem_init ( sem_t  * semaphore, int  partage, unsigned int  max_tickets) ;
```

partage vaut **0** si le sémaphore est **interne** au processus et différent de 0 s'il est partageable par plusieurs processus. La bibliothèque LinuxThreads ne permet pas actuellement de partager un sémaphore entre des threads de différents processus. Donc *partage* doit toujours être égal à **0**.

max_tickets est le nombre **maximum de tickets** dont disposera le sémaphore. Il représente le nombre maximum de threads autorisés à entrer dans leur section critique (ou à accéder une ressource partageable).
En cas de succès**,** la fonction retourne **0**, sinon elle retourne un code d'erreur non nul.

Remarques :

- Etant donné que les sémaphores servent à synchroniser différents threads, il faut les déclarer comme des variables **globales** si on veut qu'elles soient utilisables par tous les threads du processus.
- On peut limiter l'usage d'un sémaphore à un thread et à sa descendance en déclarant cette variable comme une variable **locale statique** de ce thread.

4.2- Demande d'un ticket

Un thread qui veut entrer dans sa section critique (ou accéder une ressource partageable) doit **demander** un ticket au sémaphore en appelant la fonction **sem_wait ( )** :

include < semaphore.h>

int **sem_wait** (sem_t * *semaphore*) ;

Si le nombre de tickets disponibles est > 0, la fonction décrémente le compteur de tickets de 1 et se termine. Le thread appelant peut poursuivre son exécution.

Si le nombre de tickets disponibles est égal à 0, le thread appelant restera **bloqué** au moins jusqu'à ce que le le nombre de tickets disponibles redevienne > 0 et qu'en plus il soit choisi par le noyau s'il y a **plusieurs thread en attente**.

Cette fonction ne retourne jamais de code d'erreur, elle retourne toujours **0**.

Rappel :

la fonction **sem_wait ( )** est un point d'annulation.

La fonction **sem_trywait ( )** est une variante **non bloquante** de la fonction **sem_wait ( )**.

int **sem_trywait** (sem_t * *semaphore*) ;

Si le nombre de tickets disponibles est > 0, la fonction décrémente le compteur de tickets de 1 et se termine. Le thread appelant peut poursuivre son exécution.

Si le nombre de tickets disponibles est égal à 0, la fonction se termine immédiatement et positionne la variable **errno** à la valeur **EAGAIN**.

En cas de succès**,** la fonction retourne **0**, sinon elle retourne un code d'erreur non nul.

La fonction **sem_getvalue ( )** permet de connaître le nombre de tickets dont dispose un sémaphore à un instant donné :

int **sem_getvalue** (sem_t * *semaphore* , int * *tickets*) ;

le nombre courant de tickets du sémaphore, est stocké dans la variable pointée par *tickets* .

Cette fonction ne retourne jamais de code d'erreur, elle retourne toujours **0**.

4.3- Restitution d'un ticket

Un thread qui sort de sa section critique (ou qui libèrer une ressource partageable) doit **restituer** un ticket au sémaphore en appelant la fonction **sem_post ( )** :

include < semaphore.h>

int **sem_post** (sem_t * *semaphore*) ;

la fonction incrémente le compteur de tickets de 1 et se termine. Le thread appelant peut poursuivre son exécution.

En cas de succès**,** la fonction retourne **0**, sinon elle retourne un code d'erreur non nul.

Remarques :

- Le système ne fait aucun contrôle sur la détention de tickets par les threads. Seul un thread sait combien il détient de tickets de tel ou tel sémaphore.
- Par conséquent, un thread peut restituer un ticket qu'il n'a pas ou plus de ticket qu'il n'en a obtenu et dans ce cas la fonction **sem_post ( )** ne retournera pas d'erreur.
- La fonction **sem_post ( )** retournera une erreur uniquement quand le nombre de tickets restitués dépassera la valeur maximale autorisée pour un sémaphore (constante **SEM_VALUE_MAX**)**.**