Les PThreads

POSIX Threads

Michel Soto

Université Paris Descartes

Les Threads

Qu'est-ce qu'un thread ?

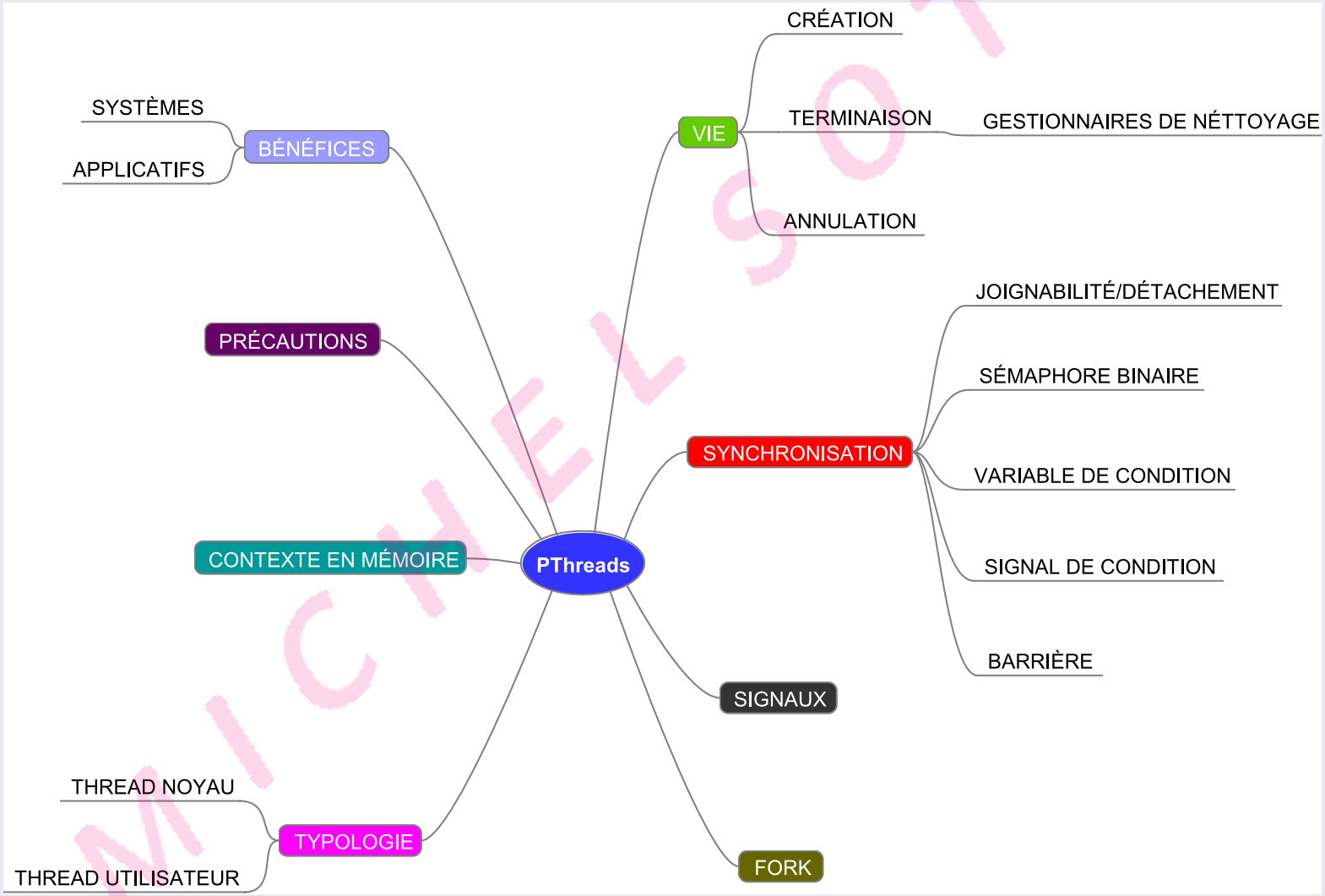
Definition

- Pour le système, il s'agit de flux (fil) d'instructions indépendants, *appartenant à un même processus*, et qui peuvent être ordonnancés pour le partage du processeur.
- Pour le développeur, ils s'agit de fonctions, au sein d'un même processus, qui peuvent s'exécuter simultanément (concurrence)

REMARQUE

Un processus "classique" peut donc être considéré comme n'ayant qu'un seul et unique thread

Carte conceptuelle



Bénéfices systèmes

- Les threads utilisent les ressources de leur processus.
  Seules sont dupliquées les ressources nécessaires à leurs fonctionnement et à leur ordonnancement.
- La gestion des processus par le système consomme une quantité non négligeable de ressources (overhead)
  - création plus rapide que pour un processus
  - ne sollicite pas le gestionnaire de mémoire
  - commutation rapide entre les threads
  - utilisation de plusieurs processeurs

Bénéfices applicatifs

Simplification de l'expression et de l'implémentation du parallélisme intrinsèque d'une application

- Simplification de la gestion des évènements asynchrones
  - Un thread peut être associé à chaque événement dont la gestion est alors codée de manière synchrone et donc plus simple.
- Facilite le partitionnement des programmes en tâches parallèles
  - Par exemple, les interfaces graphiques sont toujours multi threadées.
    Pendant que l'application travaille, l'utilisateur peut interagir avec son interface
- Permet de tirer partie des architectures multi processeurs

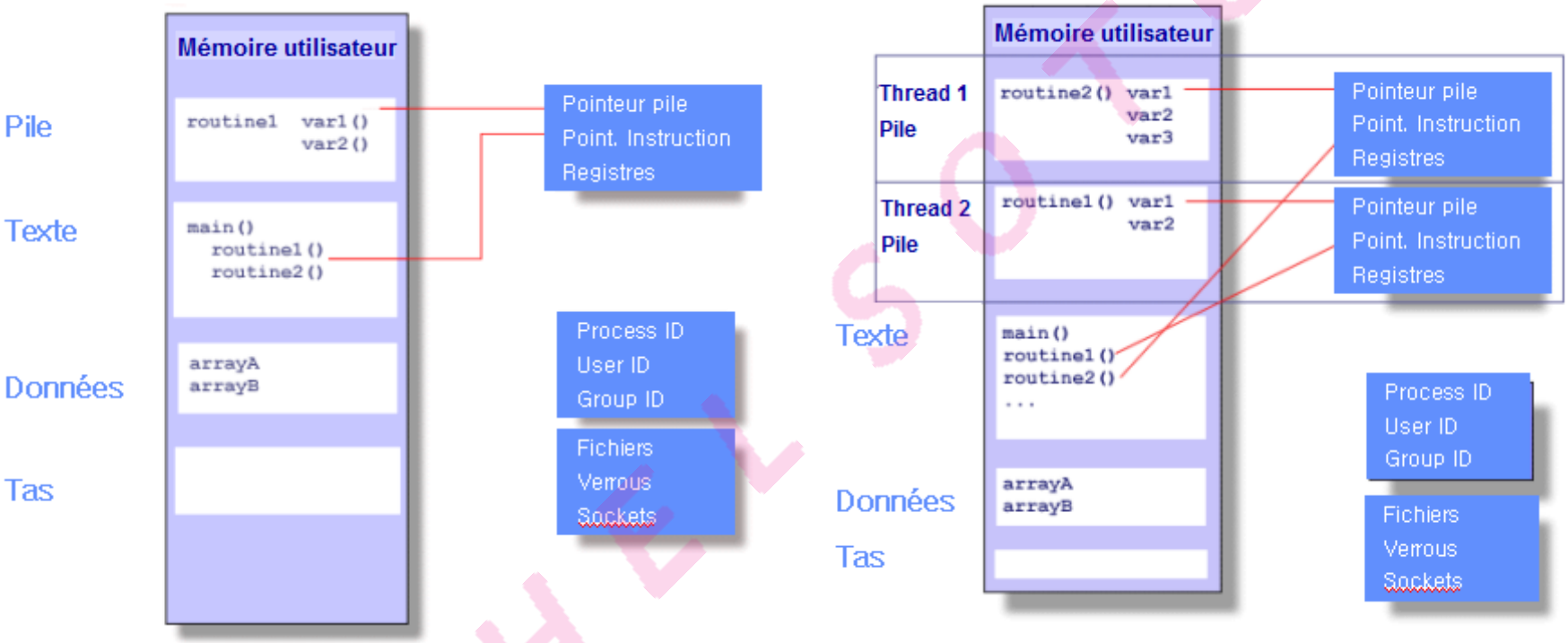
Précautions d'usage

- Le développement d'applications (écriture, test, maintenance) utilisant le parallélisme entre ses composants est toujours plus délicat.
  - Combinatoire importante des états
  - L'utilisation d'une même librairie (non réentrance) par plusieurs threads peut engendrer des corruptions de données
- Rigueur et méthode de développement sont plus que jamais nécessaires pour obtenir les gains potentiels.
- Portabilité
  - Certaines limites varient d'un système à l'autre
    - Nombre maximum de threads par processus
    - Taille maximale de la pile d'un thread

Contexte d'un thread

- Pointeur de pile
- Registres
- Propriétés d'ordonnancement (politique, priorité)
- Ensemble des signaux pendants ou bloqués
- Données propres (pile)
- Taille maximale de la pile d'un thread

Représentation en mémoire d'un thread (Fin)



Processus vs Thread

Propriétés d'un thread

- Utilise les ressources de son processus
- Partage les ressources de son processus avec les autres threads
- Tout changement effectué par un thread sur les ressources de son processus est visible par tous les autres threads
- Se termine si son processus se termine
- Partage le même espace virtuel de mémoire avec les autres threads
  - *Deux pointeurs ayant la même valeur pointent sur la même donnée*
- La lecture ou l'écriture d'une même adresse de mémoire par plusieurs threads est possible
  - *Requière donc un travail de synchronisation explicite* de la part du développeur
- Un thread peut créer d'autres threads
  - *Il n'existe ni hiérarchie ni dépendance entre les threads*

Identifiant

- Les identifiants ne peuvent être considérés uniques qu'entre les threads d'*un même processus*
- L'identifiant d'un thread est du type `pthread_t`
  - Ce type opaque autorise toutes les implémentations telles que un simple entier (Linux), un pointeur (Mac OS X, FreeBSD 8.0) ou une structure !!
    La fonction `pthread_equal` permet de comparer deux identifiants de threads

    ```
    #include <pthread.h>
    int pthread_equal(pthread_t tid1, pthread_t tid2);
    ```

  - Toute utilisation de ce type reposant sur la connaissance locale de son implémentation compromet la portabilité du code

Création

```
#include <pthread.h>
int pthread_create (pthread_t *thread, pthread_attr_t *attr,
                    void * (*start_routine)(void *), void *arg);
```

- `thread` : identifiant du nouveau thread retourné par la fonction
- `attr` : paramètres de fonctionnement du thread (`NULL` possible):
  Détaché ou joignable, héritage de la politique de scheduling, politique de scheduling (FIFO (first-in first-out), RR (round-robin) ou OTHER (déterminé par le système), paramètres de scheduling, taille de la pile, adresse de la pile, etc
- `start_routine` : fonction exécutée par le thread
- `arg` : paramètres éventuels (NULL possible) de `start_routine`. Passés par référence par un pointeur avec un cast de type void.

Retourne :

- 0 en cas de succès
- une valeur $\neq 0$ en cas d'erreur

Terminaison

```
#include <pthread.h>
void pthread_exit(void *value_ptr);
```

- `value_ptr` : Pointeur vers une variable contenant la valeur retournée par le thread qui se termine.
  Cette valeur pourra être consultée par tout thread effectuant un `pthread_join` avec le thread qui se termine.
  - Ne doit pas pointer vers une variable locale du thread qui se termine

Aucun retour de cette fonction

Fonctions `exit` ou `_exit`

- Un appel à l'une des fonctions `exit` a pour effet de terminer le processus auquel le thread appelant appartient
- La réception d'un signal par un thread peut avoir pour effet de terminer le processus auquel le thread appartient

Un exemple

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define  MAX_THREADS    3
// ------------------------------------------------
   void *Bonjour(void *thread_id){// Code du thread
// ------------------------------------------------
   long tid;
   tid = (long)thread_id;

   printf("Bonjour! Je suis le thread %ld!\n", tid);
   pthread_exit(NULL); // terminaison du thread
}// Bonjour
int main(int argc, char *argv[]){
   pthread_t threads[MAX_THREADS];
   int r;
   long t;
   for(t=0; t<MAX_THREADS; t++){
       printf("Main: création du thread %ld\n", t);
       r = pthread_create(&threads[t], NULL, Bonjour, (void *)t);
       if (r){printf("ERREUR; pthread_create() a retourné %d\n", r); exit(-1);}//if
   } // for
   pthread_exit(NULL); // terminaison du thread principal (main)
} // main
```

Un exemple: résultat

```
-bash-4.0$ ./bonjour
Main: création du thread 0
Main: création du thread 1
Bonjour! Je suis le thread 0!
Bonjour! Je suis le thread 1!
Main: création du thread 2
Bonjour! Je suis le thread 2!

-bash-4.0$ ./bonjour
Main: création du thread 0
Main: création du thread 1
Main: création du thread 2
Bonjour! Je suis le thread 1!
Bonjour! Je suis le thread 0!
Bonjour! Je suis le thread 2!

-bash-4.0$ ./bonjour
Main: création du thread 0
Main: création du thread 1
Bonjour! Je suis le thread 0!
Main: création du thread 2
Bonjour! Je suis le thread 1!
Bonjour! Je suis le thread 2!
```

Passage de paramètres

- Un seul et unique paramètre possible
  - passé par référence à la fonction `start_routine` au moment de la création du thread
- En cas de paramètres multiples:
  - création d'une structure contenant les divers paramètres
  - passage de l'adresse de cette structure via le paramètre `arg` de `start_routine`

Passage de plusieurs paramètres

Un exemple

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX_THREADS 9
struct thread_param{ // Paramétres des threads PrintMessage
        int num_message;  char *message;
};
struct thread_param t_thread_param[MAX_THREADS];
// ------------------------------------------------------
   void *PrintMessage(void *threadmess) {// Code du thread
// ------------------------------------------------------
   struct thread_param *mes_param;
   mes_param = (struct thread_param *) threadmess;
   printf ("message %d: %s\n", mes_param->num_message, mes_param->message);
   pthread_exit(NULL);
} // PrintMessage
int main (int argc, char *argv[]) {
    int i, r;
    pthread_t threads[MAX_THREADS];
    printf("DEBUT MAIN\n");
    for (i=1; i<argc && i<MAX_THREADS; i++){
        t_thread_param[i].num_message = i;
        t_thread_param[i].message = argv[i];

        r = pthread_create(&threads[i], NULL, PrintMessage, (void *) &t_thread_param[i]);
        if (r!=0) {printf("ERREUR; pthread_create() a retourné %d\n", r); exit(-1);}
    } // for
    sleep (2); // Les threads meurent avec le processus qui les a créés
               // Cet appel à sleep leur laisse le temps de s'exécuter
    printf("FIN MAIN\n");
} // main
```

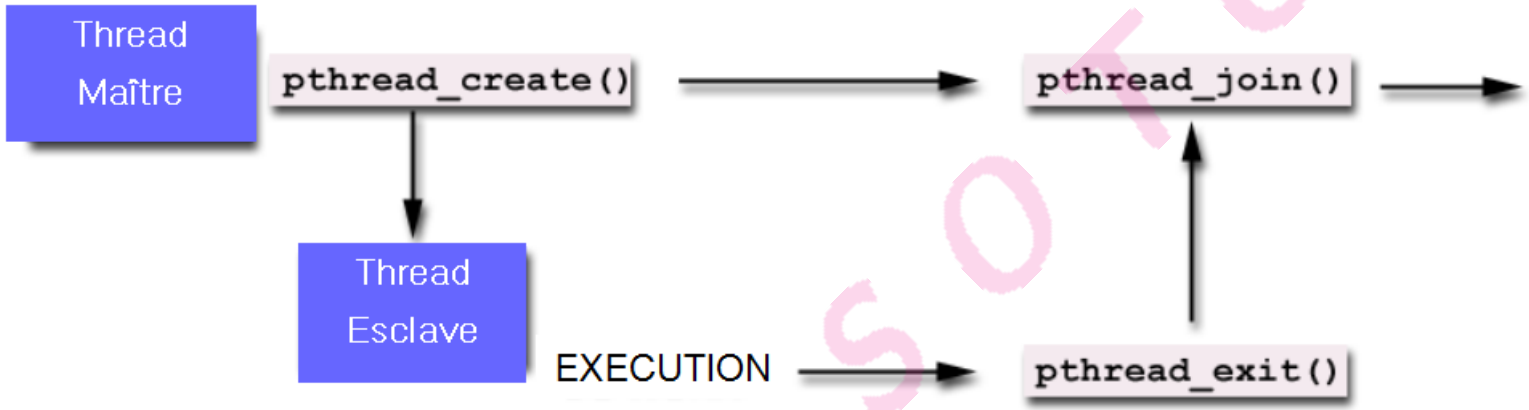
Un exemple: résultat

```
-bash-4.0$ ./bonjour_arg Timeo Danaos et dona ferentes
DEBUT MAIN
message 1: Timeo
message 5: ferentes
message 4: dona
message 2: Danaos
message 3: et
FIN MAIN

-bash-4.0$ ./bonjour_arg Timeo Danaos et dona ferentes
DEBUT MAIN
message 5: ferentes
message 2: Danaos
message 4: dona
message 3: et
message 1: Timeo
FIN MAIN

-bash-4.0$
```

Joignabilité



Principe

- Permet la synchronisation sur la terminaison des threads
- Un thread créé à l'état *joignable* peut être attendu par **n'importe quel autre thread**

REMARQUE

Analogue au mécanisme `wait` & `exit` entre processus mais sans la relation hiérarchique père/fils

Détachement

- Un thread crée à l'état *détaché* ne peut jamais être joint (attendu)
- Un thread créé à l'état *joignable* peut être *détaché*
  - il n'est alors plus joignable
  - il n'y a *pas de retour possible à l'état joignable*

Etat de joignabilité/détachement

```
#include <pthread.h>
int pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int detachstate);
int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                int *detachstate)
```

- `attr` : pointeur vers la variable d'état
- `detachstate` :
  - `PTHREAD_CREATE_JOINABLE`
  - `PTHREA_CREATE_DETACHED`

Retourne :0 en cas de succès ou une valeur $\neq 0$ en cas d'erreur

Portabilité

Par défaut, les threads sont joignables (dépend des implémentations !)

- Créer explicitement les threads à l'état joignable (portabilité)
- Créer les threads qui ne seront jamais joints à l'état détaché
  - Libération de ressources systèmes

Détachement d'un thread joignable

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
```

- `thread` : identifiant du thread à détacher

Retourne:

- 0 en cas de succès
- une valeur $\neq 0$ en cas d'erreur

Attente d'un thread joignable

```
#include <pthread.h>
int pthread_join(pthread_t th, void **thread_return);
```

- `th` : thread dont la fin est attendue
- `thread_return` : valeur renvoyée par le thread qui s'est terminé (NULL possible)
  - valeur de l'argument du `pthread_exit()`
  - `PTHREAD_CANCELED` si le thread attendu a été annulé

Retourne :0 en cas de succès ou une valeur $\neq 0$ en cas d'erreur

ATTENTION

- Suspend l'exécution du thread appelant jusqu'à ce que le thread identifié par `th` se termine, soit en appelant `pthread_exit()` soit après avoir été annulé.
- Au plus un seul thread peut attendre la mort d'un thread donné.
  - `pthread_join()` sur un thread `th` dont un autre thread attend déjà la fin renvoie une erreur.
- Les ressources mémoire (descripteur de thread et pile) d'un thread terminé ne sont pas désallouées jusqu'à ce qu'un autre thread le joigne en utilisant pthread_join().
  - `pthread_join()` doit être appelé une fois pour chaque thread joignable pour éviter des fuites de mémoire.

Joignabilité: mise en œuvre

- Utilisation du paramètre `attr` de `pthread_create()`
  1. Déclaration d'une variable de type `pthread_attr_t`
  2. Initialisation de cette variable avec `pthread_attr_init()`
  3. Positionnement de cette variable à l'état *joignable* avec `pthread_attr_setdetachstate()`
  4. Création des threads *joignables*
     `pthread create(..., &attr, ...)`
  5. Libération des ressources avec `pthread_attr_destroy()` une fois la création terminée
  6. Attente de la fin de chaque thread avec `pthread_join()`

Un exemple (1/3)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX_THREADS 10

struct thread_param{
        int tid;
        char *message;
        long retour;
};
struct thread_param t_thread_param[MAX_THREADS];
// ------------------------------------------------
   void *PrintMessage(void *threadmess) {// Code du thread
// ------------------------------------------------
      struct thread_param *mes_param;
      long retour;

      mes_param = (struct thread_param *) threadmess;
      printf ("\tThread %d: %s\n", mes_param->tid, mes_param->message);
      mes_param->retour=mes_param->tid*10;
      pthread_exit((void *)mes_param->retour);
} // PrintMessage
```

Un exemple (2/3)

```
   int main (int argc, char *argv[]) {
      int i, r;
      void *status;
// 1) Déclaration de la variable de type pthread_attr_t
      pthread_attr_t attr;
      pthread_t threads[MAX_THREADS];
      printf("DEBUT MAIN\n");
// 2) Initialisation de la variable de type pthread_attr_t
      pthread_attr_init(&attr);
// 3) Positionnement de l'état JOIGNABLE
      pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
      for (i=1; i<argc && i<MAX_THREADS; i++){
         t_thread_param[i].tid = i;
         t_thread_param[i].message = argv[i];
         t_thread_param[i].retour = 0;
// 4) Création du thread à l'état JOIGNABLE
         r = pthread_create(&threads[i], &attr, PrintMessage,
                            (void *) &t_thread_param[i]);
         if (r!=0){
             printf("ERREUR:pthread_create() a retourné %d\n",r);
             exit(-1);
         } // if
      } // for
```

Un exemple (3/3)

```
// 5) Libération des ressources
      pthread_attr_destroy(&attr);

// 6) Attente de la fin de chacun des threads avec pthread_join ()
      for (i=1; i<argc && i<MAX_THREADS; i++){

            r = pthread_join(threads[i], &status);
            if (r!=0) {
                printf("ERREUR:pthread_join() a retourné %d\n",r);
                exit(-1);
             } // if

             printf("MAIN: fin du thread %d qui a retourné la valeur %ld\n",
             t_thread_param[i].tid, (long)status);
      } // for

      printf("FIN MAIN\n");

} // main
```

Un exemple: résultat

```
-bash-4.0$ ./bonjour_join D amour belle Marquise vos yeux me font mourir
DEBUT MAIN
        Thread 1: D
        Thread 3: belle
        Thread 2: amour
        Thread 9: mourir
        Thread 6: yeux
        Thread 7: me
        Thread 8: font
MAIN: fin du thread 1 qui a retourné la valeur 10
MAIN: fin du thread 2 qui a retourné la valeur 20
MAIN: fin du thread 3 qui a retourné la valeur 30
        Thread 4: Marquise
        Thread 5: vos
MAIN: fin du thread 4 qui a retourné la valeur 40
MAIN: fin du thread 5 qui a retourné la valeur 50
MAIN: fin du thread 6 qui a retourné la valeur 60
MAIN: fin du thread 7 qui a retourné la valeur 70
MAIN: fin du thread 8 qui a retourné la valeur 80
MAIN: fin du thread 9 qui a retourné la valeur 90
FIN MAIN
```

Synchronisation entre threads

- Exclusion mutuelle
  - Mise en place de sections critiques pour accéder et modifier des données partagées entre plusieurs threads

- Variable de condition (événement)
  - Synchronisation de threads sur la valeur d'une variable globale
    - Évite les attentes actives
  - Toujours utilisée conjointement avec l'exclusion mutuelle

- Barrière
  - Coordination de multiples threads travaillant en parallèle
  - Permet à chacun des threads
    - *d'attendre* jusqu'à ce que TOUS les threads qui coopèrent aient atteint un certain point de leur exécution
    - *de continuer* , ensuite, son exécution
  - Toujours utilisée conjointement avec l'exclusion mutuelle

Types de sémaphore

- Sémaphore rapide
  `PTHREAD_MUTEX_INITIALIZER`
  - Si un thread verrouille un sémaphore qu'il possède déjà alors il est *bloqué définitivement*.
- Sémaphore récursif
  `PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP`
  - Si un thread verrouille un sémaphore qu'il possède déjà alors le compteur de verrouillage du sémaphore est incrémenté
  - Un nombre égal de déverrouillage par ce même processus doit être réalisé avant que le sémaphore retourne à l'état déverrouillé.
- Sémaphore à contrôle d'erreur
  `PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP`
  - Si un thread verrouille un sémaphore qu'il possède déjà alors `pthread_mutex_lock()` retourne immédiatement avec le code d'erreur `EDEADLK`

NP: Non Portable (hors POSIX)

Initialisation d'un sémaphore

```
#include <pthread.h>
int  pthread_mutex_init(pthread_mutex_t  *mutex,
     const pthread_mutex-attr_t *mutexattr);
```

- `mutex` : sémaphore binaire
- `mutexattr` :
  - `PTHREAD_MUTEX_INITIALIZER`
  - `PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP`
  - `PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP`
  - `NULL`

Retourne:

- 0 en cas de succès
- une valeur $\neq 0$ en cas d'erreur

Initialisation d'un sémaphore (suite)

Il est possible d'initialiser un sémaphore lors de sa déclaration

```
#include <pthread.h>
...
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;

ou
mpthread_mutex_t mutex=PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;

ou
pthread_mutex_t mutex=PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
```

Destruction d'un sémaphore

```
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- `mutex` : sémaphore à détruire

Retourne:

- 0 en cas de succès
- une valeur $\neq 0$ en cas d'erreur

- Détruit le sémaphore `mutex`
- Libère les ressources éventuelles associées à `mutex`

ATTENTION

- `mutex` doit être déverrouillé
- comportement indéfini en cas de tentative de destruction d'un sémaphore verrouillé

Verrouillage d'un sémaphore

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- `mutex` : sémaphore à verrouiller

Retourne:

- 0 en cas de succès
- une valeur $\neq 0$ en cas d'erreur
  - EINVAL : `mutex` n'a pas été initialisé.

- Verrouille `mutex` et retourne immédiatement si mutex n'est pas verrouillé
- Bloque le thread appelant si `mutex` est déjà verrouillé
- Débloque le thread appelant, *selon la politique de scheduling*, lorsque `mutex` est déverrouillé.
  - `mutex` demeure verrouillé

Verrouillage d'un sémaphore (suite)

```
#include <pthread.h>
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- `mutex` : sémaphore à verrouiller

Retourne:

- 0 en cas de succès
- une valeur $\neq 0$ en cas d'erreur
  - EINVAL : `mutex` n'a pas été initialisé.

- Verrouille `mutex` et retourne immédiatement si mutex n'est pas verrouillé
- *Retourne immédiatement* si `mutex` est déjà verrouillé
  - Code erreur: `EBUSY`

Déverrouillage d'un sémaphore

```
#include <pthread.h>
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- `mutex` : sémaphore à déverrouiller

Retourne:

- 0 en cas de succès
- une valeur $\neq 0$ en cas d'erreur
  - EINVAL : `mutex` n'a pas été initialisé.

- Choisit, selon la politique de scheduling, un thread parmi ceux bloqués sur `mutex` et le débloque
  - `mutex` *demeure verrouillé*
- Si aucun thread n'est bloqué sur `mutex` alors `mutex` est déverrouillé

Scénario type d'exclusion mutuelle

1. Déclaration et initialisation d'une variable (sémaphore) `mutex` de type `pthread_mutex_t`
2. Plusieurs threads tentent de verrouiller la variable `mutex`
3. Un seul thread parvient à verrouiller `mutex` et entre en section critique.
4. Ce thread accomplit les actions de sa section critique
5. Il sort de sa section critique en déverrouillant `mutex`
6. Un autre thread verrouille `mutex` et ainsi de suite (étape 4)
7. Finalement, libération (destruction) de `mutex`.

Un exemple (1/3)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define MAX_THREADS 1000
    struct thread_param {int tid; int nb_incr; long retour;};
    struct thread_param t_thread_param[MAX_THREADS];
// 1) Déclaration d'une variable (sémaphore) mutex} de type pthread_mutex_t
    pthread_mutex_t mutex_compteur;
    int compteur=0;
// -------------------------------------------------------
    void *IncrCompteur(void *threadparam) {// Code du thread
// -------------------------------------------------------
struct thread_param *mes_param;
long retour; int r;
mes_param = (struct thread_param *) threadparam;
for (; mes_param->nb_incr>0; mes_param->nb_incr--) {
// 2) Tentative de verrouillage de la variable mutex
        r=pthread_mutex_lock (&mutex_compteur);
        if (r!=0){perror("ERREUR pthread_mutex_lock()"); exit(EXIT_FAILURE);} // if
// 3) 6) Un seul thread parvient à verrouiller mutex et entre en section critique
// 4) Début de section critique
        compteur=compteur + 1;
// 5) Fin de section critique: dévérouillage de mutex
        r=pthread_mutex_unlock (&mutex_compteur);
        if (r!=0){perror("ERREUR pthread_mutex_unlock()"); exit(EXIT_FAILURE);} // if
    } // for
    mes_param->retour=mes_param->tid*10;
    pthread_exit((void *)mes_param->retour);
} // IncrCompteur
```

Un exemple (2/3)

```
int main (int argc, char *argv[]) {
int i, r, nb_threads, nb_incr;
void *status;
pthread_attr_t attr;
pthread_t threads[MAX_THREADS];

if (argc != 3){fprintf(stderr, "Erreur usage: %s nb_threads nb_incr\n", argv[0]); exit(EXIT_FAILURE);}
pthread_mutex_init (&mutex_compteur, NULL);
printf("DEBUT MAIN\n");
nb_threads=atoi(argv[1]);
nb_incr=atoi(argv[2]);
// 1) Initialisation du sémaphore
    r=pthread_mutex_init (&mutex_compteur, NULL);
    if (r!=0){perror("ERREUR pthread_mutex_init()"); exit(EXIT_FAILURE);} // if

//> Initialisation et positionnement de la joignabilité des threads
    r=pthread_attr_init(&attr);
    if (r!=0){perror("ERREUR pthread_attr_init()"); exit(EXIT_FAILURE);} // if
    r=pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
    if (r!=0){perror("ERREUR pthread_attr_setdetachstate()"); exit(EXIT_FAILURE);} // if

//> Création des threads
    for (i=1; i<=nb_threads && i<=MAX_THREADS; i++){
        t_thread_param[i].tid = i;
        t_thread_param[i].nb_incr = nb_incr;
        t_thread_param[i].retour = 0;

        r = pthread_create(&threads[i], &attr, IncrCompteur, (void *) &t_thread_param[i]);
        if (r!=0){perror("ERREUR pthread_create()"); exit(EXIT_FAILURE);} // if
    } // for
```

Un exemple (3/3)

```
//> Attente de la fin des threads
    for (i=1; i<=nb_threads && i<=MAX_THREADS; i++){

        r = pthread_join(threads[i], &status);
        if (r!=0) {perror("ERREUR pthread_join()");exit(EXIT_FAILURE);} // if

    } // for

// 7) Libération des ressources du sémaphore
    r=pthread_mutex_destroy (&mutex_compteur);
    if (r!=0) {perror("ERREUR pthread_mutex_destroy()");exit(EXIT_FAILURE);} // if

    printf(">>>> Valeur finale du compteur:   %d <<<<\n", compteur);
    printf(">>>> Valeur attendue du compteur: %d <<<<\n", nb_threads*nb_incr);

    printf("FIN MAIN\n");

} // main
```

Un exemple: résultats avec sémaphore

```
-bash-4.0$ ./mutex 3 1000
DEBUT MAIN
>>>> Valeur finale du compteur:   3000 <<<<
>>>> Valeur attendue du compteur: 3000 <<<<
FIN MAIN

-bash-4.0$ ./mutex 3 10000
DEBUT MAIN
>>>> Valeur finale du compteur:   30000 <<<<
>>>> Valeur attendue du compteur: 30000 <<<<
FIN MAIN

-bash-4.0$ ./mutex 15 1000
DEBUT MAIN
>>>> Valeur finale du compteur:   15000 <<<<
>>>> Valeur attendue du compteur: 15000 <<<<
FIN MAIN

-bash-4.0$ ./mutex 100 1000
DEBUT MAIN
>>>> Valeur finale du compteur:   100000 <<<<
>>>> Valeur attendue du compteur: 100000 <<<<
FIN MAIN

-bash-4.0$ ./mutex 500 1000
DEBUT MAIN
>>>> Valeur finale du compteur:   500000 <<<<
>>>> Valeur attendue du compteur: 500000 <<<<
FIN MAIN
```

Un exemple: résultats sans sémaphore

```
-bash-4.0$ ./no_mutex 3 1000
DEBUT MAIN
>>>> Valeur finale du compteur:   3000 <<<<
>>>> Valeur attendue du compteur: 3000 <<<<
FIN MAIN

-bash-4.0$ ./no_mutex 3 10000
DEBUT MAIN
>>>> Valeur finale du compteur:   28408 <<<<
>>>> Valeur attendue du compteur: 30000 <<<<
FIN MAIN

-bash-4.0$ ./no_mutex 15 1000
DEBUT MAIN
>>>> Valeur finale du compteur:   15000 <<<<
>>>> Valeur attendue du compteur: 15000 <<<<
FIN MAIN

-bash-4.0$ ./no_mutex 100 1000
DEBUT MAIN
>>>> Valeur finale du compteur:   100000 <<<<
>>>> Valeur attendue du compteur: 100000 <<<<
FIN MAIN

-bash-4.0$ ./no_mutex 500 1000
DEBUT MAIN
>>>> Valeur finale du compteur:   496687 <<<<
>>>> Valeur attendue du compteur: 500000 <<<<
FIN MAIN
```

Variable de condition: scénario type

	Thread principal (main)
1	Déclarer et initialiser la variable globale sur laquelle les threads se synchroniseront
2	Déclarer et initialiser une variable de condition
3	Déclarer et initialiser un sémaphore associé à la variable de condition
4	Créer les *threads 1 et 2*
5	. . .

	Thread 1
1	. . .
2	Entrer en section critique pour accéder la variable globale utilisée par d'autres threads (`pthread_mutex_lock ()`)
3	Appeler `pthread_cond_wait()` pour se mettre en attente bloquante jusqu'à la réception d'un signal de la part du *Thread 2* indiquant que la variable à atteint la valeur souhaitée. `pthread_cond_wait()` réalise de manière **ATOMIQUE** le déverrouillage du sémaphore ET la suspension de l'exécution du thread ▪ **Thread 2 n'est pas bloqué** ▪ **Aucun signal de condition ne peut être perdu**
4	Quand le signal du *Thread 2* est reçu, le sémaphore est verrouillé automatiquement et de manière atomique. L'attente est alors terminée
5	Déverrouiller le sémaphore
6	. . .

	Thread 2
1	. . .
2	Entrer en section critique pour accéder la variable globale utilisée par d'autres threads (`pthread_mutex_lock ()`)
3	Modifier la variable dont une valeur intéresse le *Thread 1* qui est éventuellement en attente.
4	Si cette variable globale a atteint la valeur qui interesse *Thread 1*, alors lui envoyer un signal avec `pthread_cond_signal()` afin de le débloquer.
5	Déverrouiller le sémaphore
6	. . .

	Thread principal (main)
6	Joindre ou pas *Thread 1* et *Thread 2*
7	. . .

Initialisation d'une variable de condition

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);
```

- `cond` : variable à initialiser
- `cond_attr` :
  - `NULL`
  - seul UNIX définit des paramètres pour les variables de condition

Retourne:

- 0 en cas de succès
- une valeur $\neq 0$ en cas d'erreur

Il est possible d'initialiser un variable condition lors de sa déclaration

```
#include <pthread.h>
...
pthread_cond_t cond=PTHREAD_COND_INITIALIZER;
```

Destruction d'une variable de condition

```
#include <pthread.h>
int pthread_cond_destroy(pthread_cond_t *cond);
```

- `cond` : variable à détruire
  - `cond` retourne à l'état *non initialisé*

Retourne:

- 0 en cas de succès
- une valeur $\neq 0$ en cas d'erreur

Attente de la réalisation d'une condition

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *cond,pthread_mutex_t *mutex);
```

- `cond` : variable de condition
- `mutex` : sémaphore associé et **verrouillé au préalable**

Retourne:

- 0 en cas de succès
- une valeur $\neq 0$ en cas d'erreur
  - EINVAL : `mutex` n'a pas été initialisé.

- **Bloque** le thread et **relâche atomiquement le sémaphore** si le signal correspondant à la variable condition n'est pas arrivé, non bloquant sinon.
- **Redemande le verrouillage du sémaphore** quand arrive le signal correspondant à la variable condition

Attente bornée de la réalisation d'une condition

```
#include <pthread.h>
int pthread_cond_timedwait(pthread_cond_t *cond,
    pthread_mutex_t *mutex, const struct timespec *abstime);
```

- `cond` : variable de condition
- `mutex` : sémaphore associé et **verrouillé au préalable**
- `abstime` : date de fin de l'attente du signal

Retourne: 0 en cas de succès ou une valeur $\neq 0$ en cas d'erreur

- EINVAL : `mutex` n'a pas été initialisé.

- **Bloque** le thread et **relâche atomiquement le sémaphore** si le signal correspondant à la variable condition n'est pas arrivé, non bloquant sinon.
- **Redemande le verrouillage du sémaphore** quand arrive le signal correspondant à la variable condition arrive avant le temps `abstime`.
- Met fin à l'attente si le signal n'est pas arrivé au temps `abstime` (erreur `ETIMEDOUT`) et redemande le verrouillage du sémaphore

Envoi d'un signal de condition

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
```

- `cond` : variable de condition

Retourne:

- 0 en cas de succès
- une valeur $\neq 0$ en cas d'erreur

- Envoie un signal pour la variable `cond`
- Débloque UN SEUL thread parmi ceux qui, éventuellement, attendent le signal
  - *Un thread débloqué ne reprend son exécution que lorsque le thread qui a envoyé le signal déverrouille le sémaphore associé à* `cond`.

Diffusion d'un signal de condition

```
#include <pthread.h>
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- `cond` : variable de condition

Retourne:

- 0 en cas de succès
- une valeur $\neq 0$ en cas d'erreur

- Diffuse un signal pour la variable `cond`
- Débloque TOUS les threads qui, éventuellement, attendent le signal
  - *les threads débloqués ne reprennent leur exécution que lorsque le thread qui a envoyé le signal déverrouille le sémaphore associé à* `cond`.
    - Ils sont alors en compétition (selon la politique de scheduling) pour verrouiller le sémaphore associé à `cond` et entrer dans leur section critique.

Variable de condition

Un exemple (1/)

```
#define MAX_THREADS     1000
struct thread_param{// paramétres des threads
              int tid; int nb_incr; long retour;};
struct thread_param t_thread_param[MAX_THREADS];
int compteur=0; // 1) Variable de synchronisation des threads
pthread_cond_t condition_compteur;// 2) Déclaration de la variable de condition
pthread_mutex_t mutex_compteur;// // 3) Sémaphore associé à la variable de condition
// -------------------------------------------------------------
   void *AttenteSeuil(void *threadparam) { // Code du thread 1
// -------------------------------------------------------------
    struct thread_param *mes_param;
    long retour; int r; int attente=0;
    mes_param = (struct thread_param *) threadparam;
// 2) Début de section critique
    r=pthread_mutex_lock(&mutex_compteur);
    while (compteur < mes_param->nb_incr) {//> Le seuil non atteint
          printf("Thread %d: <<< J'attends ....\n", mes_param->tid);
          attente=1;
// 3) Attente du signal
          r=pthread_cond_wait(&condition_compteur, &mutex_compteur);
    } // while car la la condition peut ne plus etre vraie aprés pthread_cond_wait () !!
    if(attente){printf("Thread %d: >>> J'ai reçu le signal: ", mes_param->tid);
              printf("le compteur a atteint %d\n", compteur);
    } else {printf("Thread %d: >>> Je n'attends pas: ", mes_param->tid);
             printf("le compteur vaut déja %d\n", compteur);
    }
// 4) Traitement
// 5) Fin de section critique
    r=pthread_mutex_unlock (&mutex_compteur);
    printf("Thread %d: Fin \n", mes_param->tid);
    mes_param->retour=mes_param->tid;
    pthread_exit((void *)mes_param->retour);
} // AttenteSeuil
```

Variable de condition

Un exemple (2/)

```
// ------------------------------------------------------------
   void *IncrCompteur(void *threadparam) { // Code du thread 2
// ------------------------------------------------------------
    struct thread_param *mes_param;
    long retour; int r;

    mes_param = (struct thread_param *) threadparam;
    printf("Thread %d: Debut\n", mes_param->tid);
    for (;;) {
// 2) Début de section critique
        r=pthread_mutex_lock (&mutex_compteur);
        if (compteur == mes_param->nb_incr){
// 4) Le seuil est atteint: envoi du signal
            r=pthread_cond_signal(&condition_compteur);
            printf("Thread %d: Fin \n", mes_param->tid);

        //> Fin de section critique
            r=pthread_mutex_unlock (&mutex_compteur);

        //> Terminaison du thread
            mes_param->retour=mes_param->tid;
            pthread_exit((void *)mes_param->retour);
        }
// 3) Le seuil n'est pas atteint: modifier la variable compteur
        else {compteur=compteur + 1;
        }  //if
// 5) Fin de section critique
        r=pthread_mutex_unlock (&mutex_compteur);
    } // for
} // IncrCompteur
```

Variable de condition

Un exemple (3/)

```
int main (int argc, char *argv[]) {
int i, r, nb_threads, nb_incr;
void *status;
pthread_attr_t attr;
pthread_t threads[MAX_THREADS];

if (argc != 3){fprintf(stderr, "Erreur usage: %s nb_threads nb_incr\n", argv[0]); exit(EXIT_FAILURE);}
printf("DEBUT MAIN\n");
nb_threads=atoi(argv[1]);
nb_incr=atoi(argv[2]);

//2) 3) Initialisation du sémaphore et de la variable de condition
r=pthread_mutex_init (&mutex_compteur, NULL);
r=pthread_cond_init (&condition_compteur, NULL);

//> Initialisation et positionnement de la joignabilité des threads
...
// 4) Création des threads d'incrémentation
...
//> Création du thread d'attente
...
//> Initialisation et positionnement de la joignabilité des threads
...
//6) Attente de la fin des threads de types 1 et 2
...
//> Libération des ressources du sémaphore
...

printf("FIN MAIN\n");
} // main
```

Un exemple: résultat

```
-bash-4.0$ ./var_cond 3 10000
DEBUT MAIN
Thread 1: Début
Thread 2: Début
Thread 2: Compteur = 10000, J'envoie le signal
Thread 2: Fin
Thread 1: Compteur = 10000, J'envoie le signal
Thread 1: Fin
Thread 3: >>> Je n'attends pas: le compteur vaut déjà 10000
Thread 3: Fin
FIN MAIN
-bash-4.0$ ./var_cond 3 10000
DEBUT MAIN
Thread 2: Début
Thread 3: <<< J'attends ...
Thread 1: Début
Thread 2: Compteur = 10000, J'envoie le signal
Thread 2: Fin
Thread 3: >>> J'ai reçu le signal: le compteur a atteint 10000
Thread 3: Fin
Thread 1: Compteur = 10000, J'envoie le signal
Thread 1: Fin
FIN MAIN
```

Variable de condition

Attention

- Recevoir un signal de condition *n'implique pas* que la condition attendue est encore vraie !!
  - rien ne garantie que le thread sera exécuté immédiatement après la réception du signal
    - *l'attente ne se termine qu'après la ré-acquisition du sémaphore*
- Démonstration: attente du Thread A sur la condition X=1
  1. Thread B: verrouillage du sémaphore associé à X.
  2. Thread A: attente sur la condition X=1.
  3. Thread B: **X=1, signal sur la variable condition associé à X**. Déverrouillage du sémaphore associé.
  4. Thread C: **verrouillage du sémaphore associé, X=0**, déverrouillage du sémaphore associé
  5. Thread A: fin de l'attente **mais X vaut 0**, *la condition est fausse*.
- Solution ? Toujours tester la condition après le réveil (voir l'exemple)

Initialisation d'une barrière

```
#include <pthread.h>
int pthread_barrier_init(pthread_barrier_t *restrict barrier,
                         const pthread_barrierattr_t *restrict attr,
                         unsigned int count);
```

- `barrier` : barrière à initialiser
- `attr` :
  - `NULL`
  - `process-shared`
- `count` : nombre de threads qui devront se synchroniser sur la barrière

Retourne:

- 0 en cas de succès
- une valeur $\neq 0$ en cas d'erreur

destruction d'une barrière

```
#include <pthread.h>
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

- `barrier` : barrière à détruire

Retourne:

- 0 en cas de succès
- une valeur $\neq 0$ en cas d'erreur

Arrêt (annulation) d'un thread

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

- `thread` : identifiant du thread à annuler

Retourne:

- 0 en cas de succès
- une valeur $\neq 0$ en cas d'erreur

- Des raisons applicatives peuvent nécessiter l'arrêt d'un thread
  - Équivalent du `kill()`pour les processus
- Quand le thread concerné répond à la requête d'annulation, il se comporte comme si il avait appelé `pthread_exit (PTHREAD_CANCELED)`

Arrêt d'un thread: comportement

- Un thread peut répondre à une requête d'annulation soit:
  - immédiatement
  - au moment où il atteint un point d'annulation
  - jamais
- Dépend de son *état d'annulation*

État et type d'annulation

- État
  Indique si un thread réagit à une requête d'annulation
  - `PTHREAD_CANCEL_ENABLE` : réagit aux requêtes d' annulation
  - `PTHREAD_CANCEL_DISABLE` : ignore les requêtes d' annulation
- Type
  Indique COMMENT un thread réagit à une requête d'annulation
  - `PTHREAD_CANCEL_ASYNCHRONOUS` : arrêt du thread dès réception d'une requête d'annulation
  - `PTHREAD_CANCEL_DEFERRED` : mise en attente la requête jusqu'à ce qu'un point d'annulation soit atteint.
- Par défaut un thread est créé avec
  - `PTHREAD_CANCEL_ENABLE` et
  - `PTHREAD_CANCEL_DEFERRED`

Points d'annulation

`pthread_join()`
`pthread_cond_wait()`
`pthread_cond_timedwait()`
`sem_wait()`
`sigwait()`

- Quand un thread exécute une de ces fonctions, il vérifie si des requêtes d'annulation lui ont été adressées
  - S'il existe une requête alors il exécute la séquence d'annulation et se termine
- En dehors de ces fonctions, `pthread_testcancel()` permet à un thread de tester si des requêtes d'annulation lui ont été adressées
  - S'il existe une requête alors il exécute la séquence d'annulation et se termine

Liste des points d'annulation

```
asctime()
basename()
catgets()
crypt()
ctermid() avec un paramétre non NULL
ctime()
dbm_clearerr()
dbm_close()
dbm_delete()
dbm_error()
dbm_fetch()
dbm_firstkey()
dbm_nextkey()
dbm_open()
dbm_store()
dirname()
dlerror()
drand48()
ecvt() [POSIX.1-2001 uniquement (fonction supprimée dans POSIX.1-2008)]
encrypt()
endgrent()
endpwent()
endutxent()
fcvt() [POSIX.1-2001 uniquement (fonction supprimée dans POSIX.1-2008)]
ftw()
gcvt() [POSIX.1-2001 uniquement (fonction supprimée dans POSIX.1-2008)]
getc_unlocked()
getchar_unlocked()
getdate()
getenv()
getgrent()
getgrgid()
getgrnam()
gethostbyaddr() [POSIX.1-2001 uniquement (fonction supprimée dans POSIX.1-2008)]
gethostbyname() [POSIX.1-2001 uniquement (fonction supprimée dans POSIX.1-2008)]
gethostent()
getlogin()
getnetbyaddr()
getnetbyname()
getnetent()
getopt()
getprotobyname()
getprotobynumber()
getprotoent()
getpwent()
getpwnam()
getpwuid()
getservbyname()
getservbyport()
getservent()
getutxent()
getutxid()
getutxline()
gmtime()
hcreate()
hdestroy()
hsearch()
inet_ntoa()
l64a()
lgamma()
lgammaf()
lgammal()
localeconv()
localtime()
lrand48()
mrand48()
nftw()
nl_langinfo()
ptsname()
putc_unlocked()
putchar_unlocked()
putenv()
pututxline()
rand()
readdir()
setenv()
setgrent()
setkey()
setpwent()
setutxent()
```

Liste des points d'annulation

```
strerror()
strsignal() [Ajoutée dans POSIX.1-2008]
strtok()
system() [Ajoutée dans POSIX.1-2008]
tmpnam() avec un paramétre non NULL
ttyname()
unsetenv()
wcrtomb() si son dernier paramétre est NULL
wcsrtombs() si son dernier paramétre est NULL
wcstombs()
wctomb()
```

Liste des points d'annulation

```
asctime()
accept()
aio_suspend()
clock_nanosleep()
close()
connect()
creat()
fcntl() F_SETLKW
fdatasync()
fsync()
getmsg()
getpmsg()
lockf() F_LOCK
mq_receive()
mq_send()
mq_timedreceive()
mq_timedsend()
msgrcv()
msgsnd()
msync()
nanosleep()
open()
openat() [Ajoutée dans POSIX.1-2008]
pause()
poll()
pread()
pselect()
pthread_cond_timedwait()
pthread_cond_wait()
pthread_join()
pthread_testcancel()
putmsg()
putpmsg()
pwrite()
read()
readv()
recv()
recvfrom()
recvmsg()
select()
sem_timedwait()
sem_wait()
send()
sendmsg()
sendto()
sigpause() [POSIX.1-2001 uniquement (dans la liste des
        fonctions pouvant être un point d'annulation dans
        POSIX.1-2008)]
sigsuspend()
sigtimedwait()
sigwait()
sigwaitinfo()
sleep()
system()
tcdrain()
usleep() [POSIX.1-2001 uniquement (fonction supprimée dans
        POSIX.1-2008)]
wait()
waitid()
waitpid()
write()
writev()
```

Gestion de l'état et du mode d'annulation

```
#include <pthread.h>
int pthread_setcancelstate (int state, int *etat_pred)
```

- `state`: état dans lequel le thread appelant se positionne
  - `PTHREAD_CANCEL_ENABLE`
  - `PTHREAD_CANCEL_DISABLE`
- `etat_pred`: ancien état dans lequel le thread appelant était positionné (`NULL` possible)

```
#include <pthread.h>
int pthread_setcanceltype (int mode, int *ancien_mode)
```

- `mode`: état dans lequel le thread appelant se positionne
  - `PTHREAD_CANCEL_ASYNCHRONOUS`
  - `PTHREAD_CANCEL_DEFERRED`
- `ancien_mode`: mode antérieur de réaction à une requête d'annulation du thread appelant (`NULL` possible)

Gestionnaires de "nettoyage"

```
#include <pthread.h>
void pthread_cleanup_push(void (*routine) (void *), void *arg);
```

- `routine`: fonction à exécuter
- `arg`: pointeur vers les paramètres de la fonction à exécuter

Aucune valeur retournée

- Installe un gestionnaire qui sera exécuté à la terminaison du thread
- Il s'agit en réalité d'une macro
- Les gestionnaires sont exécutés dans l'ordre inverse où ils ont été enregistrés

Gestionnaires de "nettoyage" (suite)

```
#include <pthread.h>
void pthread_cleanup_pop(int execute);
```

- `execute`
  - si 0 le DERNIER gestionnaire installé est *annulé*
  - si 1 le DERNIER gestionnaire installé est *exécuté et annulé*

Aucune valeur retournée

Attention

- Il s'agit en réalité d'une macro
- Les paires de `pthread_cleanup_push()` et `thread_cleanup_pop()` doivent être exécutées dans le même bloc lexical, au même niveau et ne peuvent pas être entrelacées avec une autre paire.
  En effet, les expansions de ces deux macros introduisent respectivement une accolade ouvrante "{ " et un accolade fermante " }"

Gestionnaires de "nettoyage"

Un exemple

```
pthread_cleanup_push(pthread_mutex_unlock, (void *) &mut);
pthread_mutex_lock(&mut);
                /* Traitement */
pthread_mutex_unlock(&mut);
pthread_cleanup_pop(0);
```

- Si le thread est annulé pendant sa section critique, le déverrouillage du sémaphore `mut` sera effectué avant qu'il se termine. Évitant ainsi qu'il reste verrouillé ad vitam aeternam
- Cette disposition est annulée par l'appel à `pthread_cleanup_pop(0)` à la sortie de sa section critique doivent

Les signaux

- Chaque thread possède:
  - Son propre masque de signaux
    - hérité du thread créateur
  - Son propre ensemble de signaux pendants
    - les signaux pendants ne sont pas hérités
- Les gestionnaires de signaux sont installés avec `sigaction`

Les signaux (suite)

```
#include <signal.h>
#include <pthread.h>
int pthread_sigmask(int how,const sigset_t *newmask, sigset_t  *oldmask);
```

- how
  - SIG_SETMASK : le masque de signal est positionné à newmask
  - SIG_BLOCK : les signaux contenus par newmask sont ajoutés au masque de signaux
  - SIG_UNBLOCK : les signaux contenus par newmask sont retirés du masque courant
- Si oldmask $\neq$ NULL, le précédent masque de signaux est sauvegardé à l'emplacement pointé par oldmask

Retourne:

- 0 en cas de succès
- une valeur $\neq$ 0 en cas d'erreur

Les signaux (suite)

```
#include <signal.h>
#include <pthread.h>
int pthread_kill(pthread_t thread, int signo);
```

- Envoie le signal numéro `signo` au thread `thread`

Retourne:

- 0 en cas de succès
- une valeur $\neq 0$ en cas d'erreur

Les signaux (suite)

```
#include <signal.h>
#include <pthread.h>
int sigwait(const sigset_t *set, int *sig);
```

- Suspend le thread appelant jusqu'à ce que l'un des signaux définis dans `set` soit envoyé au thread appelant
- Le numéro du signal reçu est sauvegardé à l'emplacement pointé par `sig`
- Les signaux définis dans `set` doivent être bloqués et non ignorés lorsqu'on entre dans `sigwait()`

Retourne:

- 0 en cas de succès
- une valeur $\neq 0$ en cas d'erreur

Thread utilisateur *vs* thread noyau

Thread utilisateur

- La gestion est faite en utilisant l'espace utilisateur
- Les opérations sont indépendantes du système
- Le noyau ne voit que le thread main()
- Un appel système bloquant d'un seul thread bloque le processus du thread et donc tous les autres threads également
- La commutation de contexte entre thread est réalisée par un bibliothèque légère
- Moins coûteux en ressources car pas d'appel systèmes
- Portable car indépendance vis-à-vis du noyau
- Les thread se partagent le temps CPU (quantum) de leur processus
- La priorité des thread est celle de leur processus
- Les threads s'exécutent sur le même processeur

Thread utilisateur *vs* thread noyau

Thread noyau

- Les threads sont des objets du système et il possède un descripteur pour chacun d'eux
- Les opérations sont dépendantes du système
- Le noyau ne voit chaque thread indépendamment de leur processus
- Un appel système bloquant ne bloque que le thread appelant
- La commutation de contexte entre threads est réalisée par le noyau
- Plus coûteux en ressources car appel systèmes
- Moins portable car dépendance vis-à-vis du noyau
- Les thread disposent chacun d'un temps CPU (quantum) comme les autres processus
- La priorité des thread est indépendante de celle de leur processus
- Les threads peuvent être répartis sur plusieurs processeurs

Thread utilisateur *vs* thread noyau

```
#include <pthread.h>
int pthread_attr_getscope(const pthread_attr_t *attr, int *scope);
```

- Permet de connaître le *contention scope* d'un thread
- `attr` : adresse de la variable attribut du thread concerné
- `scope` : adresse contenant le *contention scope* courant du thread

Retourne:

- 0 en cas de succès
- une valeur $\neq 0$ en cas d'erreur

Thread utilisateur *vs* thread noyau

```
#include <pthread.h>
int pthread_attr_setscope(const pthread_attr_t *attr, int scope);
```

- Permet de positionner le *contention scope* d'un thread
- `attr` : adresse de la variable attribut du thread concerné
- `scope` : adresse contenant le *contention scope* courant du thread
  - `PTHREAD_SCOPE_SYSTEM` : thread noyau
  - `PTHREAD_SCOPE_PROCESS` : thread utilisateur (non supporté par LINUX)

Retourne:

- 0 en cas de succès
- une valeur $\neq 0$ en cas d'erreur