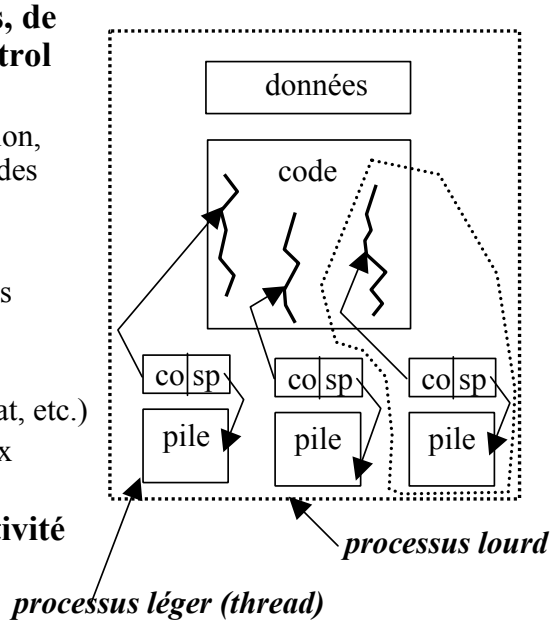
Cours 9 - PTHREADS

Partie 1

Luciana Arantes
Dec. 2005

Processus léger ou "Thread"

- **Partage les zones de code, de données, de tas + des zones du PCB (Process Control Block) :**
  - liste des fichiers ouverts, comptabilisation, répertoire de travail, userid et groupid, des handlers de signaux.
- **Chaque thread possède :**
  - un mini-PCB (son CO + quelques autres registres),
  - sa pile,
  - attributs d'ordonnancement (priorité, état, etc.)
  - structures pour le traitement des signaux (masque et signaux pendants).
- **Un processus léger avec une seule activité = un processus lourd.**



Caractéristiques des Threads

- **Avantages**
  - Création plus rapide
  - Partage des ressources
  - Communication entre les threads est plus simple que celle entre processus
    - communication via la mémoire : variables globales.
  - Solution élégante pour les applications client/serveur :
    - une thread de connexion + une thread par requête

- **Inconvénients**
  - Programmation plus difficile (mutex, interblocages)
  - Fonctions de librairie non ***multi-thread-safe***

Threads Noyau / Threads Utilisateur

- **Bibliothèque Pthreads:**
  - les threads définies par la norme **POSIX 1.c** sont indépendantes de leur implémentation.
- **Deux types d'implémentation :**
  - **Thread usager (pas connue du noyau):**
    - L'état est maintenu en espace utilisateur. Aucune ressource du noyau n'est allouée à une thread.
    - Des opérations peuvent être réalisées indépendamment du système.
    - Le noyau ne voit qu'une seule thread
      - Tout appel système bloquant une thread aura pour effet de bloquer son processus et par conséquent toutes les autres threads du même processus.
  - **Thread Noyau (connue du noyau):**
    - Les threads sont des entités du système (threads natives).
    - Le système possède un descripteur pour chaque thread.
    - Permet l'utilisation des différents processeurs dans le cas des machines multiprocesseurs.

Threads Noyau x Threads Utilisateur

Approche	Thread noyau	Thread utilisateur
Implémentation des fonctionnalités POSIX	Nécessite des appels systèmes spécifiques.	Portable sans modification du noyau.
Création d'une thread	Nécessite un appel système (ex. *clone*).	Pas d'appel système. Moins coûteuse en ressources.
Commutation entre deux threads	Faite par le noyau avec changement de contexte.	Assurée par la bibliothèque; plus légère.
Ordonnancement des threads	Une thread dispose de la CPU comme les autres processus.	CPU limitée au processus qui contient les threads.
Priorités des tâches	Chaque thread peut s'exécuter avec une prio. indépendante.	Priorité égale à celle du processus.
Parallélisme	Répartition des threads entre différents processeurs.	Threads doivent s'exécuter sur le même processeur.

Pthreads utilisant des threads Noyau

- **Trois différentes approches:**
  - **M-1 (many to one)**
    - Une même thread système est associée à toutes les *Pthreads* d'un processus.
      - Ordonnancement des threads est fait par le processus
        - Approche thread utilisateur.
  - **1-1 (one to one)**
    - A chaque *Pthread* correspond une thread noyau.
      - Les *Pthreads* sont traitées individuellement par le système.
  - **M-M (many to many)**
    - différentes *Pthreads* sont multiplexées sur un nombre inférieur ou égal de threads noyau.

Réentrance

- **Exécution de plusieurs activités concurrentes**
  - Une même fonction peut être appelée simultanément par plusieurs threads.
- **Fonction réentrante:**
  - fonction qui accepte un tel comportement.
    - pas de manipulation de variable globale
    - utilisation de mécanismes de synchronisation permettant de régler les conflits provoqués par des accès concurrents.
- **Terminologie**
  - Fonction **multithread-safe (MT-safe)** :
    - réentrant vis-à-vis du parallélisme
  - Fonction **async-safe** :
    - réentrant vis-à-vis des signaux

POSIX thread API

- **Orienté objet:**
  - ***pthread_t*** : identifiant d'une *thread*
  - ***pthread_attr_t*** : attribut d'une *thread*
  - ***pthread_mutex_t*** : *mutex* (exclusion mutuelle)
  - ***pthread_mutexattr_t*** : attribut d'un *mutex*
  - ***pthread_cond_t*** : variable de condition
  - ***pthread_condattr_t*** : attribut d'une variable de condition
  - ***pthread_key_t*** : clé pour accès à une donnée globale réservée
  - ***pthread_once_t*** : initialisation unique

POSIX thread API

- **Une Pthread est identifiée par un *ID* unique**
- **En général, en cas de succès une fonction renvoie 0 et une valeur différente de NULL en cas d'échec.**
- **Pthreads n'indiquent pas l'erreur dans *errno*.**
  - Possibilité d'utiliser *strerror*.
- **Fichier <*pthread.h*>**
  - Constantes et prototypes des fonctions.
- **Faire le lien avec la bibliothèque *libpthread.a***
  - gcc …. –l pthread
- **Directive**
  - #define _REENTRANT
  - gcc … -D _REENTRANT

Fonctions Pthreads

- **Préfixe**
  - Enlever le *_t* du type de l'objet auquel la fonction s'applique.
- **Suffixe (exemples)**
  - ***_init*** : initialiser un objet.
  - ***_destroy***: détruire un objet.
  - ***_create*** : créer un objet.
  - ***_get****attr* : obtenir l'attribut *attr* des attributs d'un objet.
  - ***_set****attr* : modifier l'attribut *attr* des attributs d'un objet.
- **Exemples :**
  - *pthread_create* : crée une thread (objet *pthread_t*).
  - *pthread_mutex_init* : initialise un objet du type *pthread_mutex_t*.

Gestion des Threads

- **Une *Pthread* :**
  - est identifiée par un *ID* unique.
  - exécute une fonction passée en paramètre lors de sa création.
  - possède des attributs.
  - peut se terminer (*pthread_exit*) ou être annulée par une autre thread (*pthread_cancel*).
  - peut attendre la fin d'une autre thread (*pthread_join*).
- **Une *Pthread* possède son propre masque de signaux et signaux pendants.**
- **La création d'un processus donne lieu à la création de la thread main.**
  - Retour de la fonction *main* entraîne la terminaison du processus et par conséquent de toutes les threads de celui-ci.

Gestion des Threads: attributs

- **Attributs passés au moment de la création de la thread :**
  Paramètre du type ***pthread_attr_t***
- **Initialisation d'une variable du type *pthread_attr_t* avec les valeurs par défaut :**
  int pthread_attr_init (pthread_attr_t *attrib) ;
- **Chaque attribut possède un *nom* utilisé pour construire les noms de deux types fonctions :**
  - *pthread_attr_get**nom** (pthread_attr_t *attr, ...)*
    - Extraire la valeur de l'attribut *nom* de la variable *attr*
  - *pthread_attr_set**nom** (pthread_attr_t *attr, ...)*
    - Modifier la valeur de l'attribut *nom* de la variable *attr*

Gestion des Threads: attributs (1)

- **Nom :**
  - **scope** (*int*) - thread native ou pas
    - PTHREAD_SCOPE_SYSTEM, PTHREAD_SCOPE_PROCESS
  - **stackaddr** (*void* *) - adresse de la pile
  - **stacksize** (*size_t*) - taille de la pile
  - **detachstate** (*int*) - thread joignable ou détachée
    - PTHREAD_CREATE_JOINABLE, PTHREAD_CREATE_DETACHED
  - **schedpolicy** (*int*) – type d'ordonnancement
    - SCHED_OTHER (unix) , SCHED_FIFO (temps-réel FIFO), SCHED_RR (temps-réel round-robin)
  - **schedparam** (*sched_param* *) - paramètres pour l'ordonnanceur
  - **inheritsched** (*int*) - ordonnancement hérité ou pas
    - PTHREAD_INHERIT_SCHED, PTHREAD_EXPLICIT_SCHED

Gestion des Threads: attributs (2)

- **Exemples de fonctions :**
  - **Obtenir/modifier l'état de détachement d'une thread**
    - PTHREAD_CREATE_JOINABLE, PTHREAD_CREATE_DETACHED

    int pthread_attr_get**detachstate** (const pthread_attr_t *attributs, int *valeur);

    int pthread_attr_set**detachstate** (const pthread_attr_t *attributs, int valeur);

  - **Obtenir/modifier la taille de la pile d'une thread**

    int pthread_attr_**getstacksize** (const pthread_attr_t *attributs, size_t *taille);

    int pthread_attr_set**stacksize** (const pthread_attr_t *attributs, size_t taille);

Gestion des Threads: attributs (3)

- **Exemples d'appels des fonctions :**
  - **Obtenir la taille de pile de la thread**

```
pthread_attr_t attr;   size_t taille;
pthread_attr_getstacksize(&attr, &taille);
```

  - **Détachement d'une thread**

```
pthread_attr_t attr;
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
```

  - **Modifier la politique d'ordonnancement (temps-réel)**

```
pthread_attr_t attr;
pthread_attr_setschedpolicy(&attr,SCHED_FIFO);
```


Création des Threads

- **Création d'une thread avec les attributs `attr` en exécutant `fonc` avec `arg` comme paramètre :**

```
int pthread_create(pthread_t *tid, pthread_attr_t *attr,
                   void * (*fonc) (void *), void *arg);
```

  - **attr :** si NULL, la thread est créée avec les attributs par défaut.
  - **code de renvoi** :
    - 0 en cas de succès.
    - En cas d'erreur une valeur non nulle indiquant l'erreur:
      - EAGAIN : manque de ressource.
      - EPERM : pas la permission pour le type d'ordonnancement demandé.
      - EINVAL : attributs spécifiés par *attr* ne sont pas valables.

Thread principale x Threads annexes

- **La création d'un processus donne lieu à la création de la *thread principale (thread main)*.**
  - Un retour à la fonction *main* entraîne la terminaison du processus et par conséquent la terminaison de toutes ses threads.
- **Une thread créée par la primitive *pthread_create* dans la fonction main est appelée une *thread annexe*.**
  - Terminaison :
    - Retour de la fonction correspondante à la thread ou appel à la fonction *pthread_exit.*
      - aucun effet sur l'existence du processus ou des autres threads.
  - L'appel à *exit* ou *_exit* par une thread annexe provoque la terminaison du processus et de toutes les autres threads.

Obtention et comparaison des identificateurs

- **Obtention de l'identité de la thread courante :**

```
pthread_t  pthread_self (void);
```

  - renvoie l'identificateur de la thread courante.

- **Comparaison entre deux identificateurs de threads**

```
pthread_t  pthread_equal(pthread_t t1,   pthread_t t2);
```

  - Test d'égalité : renvoie une valeur non nulle si *t1* et *t2* identifient la même thread.

Exemple 1 – Création d'une thread attributs standards

```
#define _POSIX_SOURCE 1
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void *test (void *arg) {
  int i;
  printf ("Argument reçu %s, tid: %d\n" ,
            (char*)arg, (int)pthread_self());

  for (i=0; i <10000000; i++);
  printf ("fin thread %d\n",
            (int)pthread_self());
  return NULL;
}
```

```
int main (int argc, char ** argv) {
  pthread_t tid;
  pthread_attr_t attr;

   if  (pthread_create (&tid, NULL,
            test, "BONJOUR") != 0) {
        perror("pthread_create \n");
        exit (1);
    }
    sleep (3);
    printf ("fin thread main \n" );
    return EXIT_SUCCESS;
}
```


Exemple 2 – Création d'une thread attributs standards

```
#define _POSIX_SOURCE 1
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void *test (void *arg) {
  int i;
  printf ("Argument reçu %s, tid: %d\n" ,
            (char*)arg, (int)pthread_self());

  for (i=0; i <10000000; i++);
  printf ("fin thread %d\n",
            (int)pthread_self());
  return NULL;
}
```

```
int main (int argc, char ** argv) {
  pthread_t tid;
  pthread_attr_t attr;

  if (pthread_attr_init (& attr) !=0) {
        perror("pthread init attibuts \n");
        exit (1);
    }
    if  (pthread_create (&tid, &attr,
            test, "BONJOUR") != 0) {
        perror("pthread_create \n");
        exit (1);
    }
    sleep (3);
    printf ("fin thread main \n" );
    return EXIT_SUCCESS;
}
```

Passage d'arguments lors de la création d'une thread

- **Passage d'arguments par référence (void *)**
  - ne pas passer en argument l'adresse d'une variable qui peut être modifiée par la thread *main* avant/pendant la création de la nouvelle thread.
    - **Exemple** :

```
/ * ne pas passer directement l'adresse de i */
int* pt_ind;

 for (i=0; i < NUM_THREADS; i++) {
     pt_ind = (int *) malloc (sizeof (i));
    *pt_ind =i;

   if  (pthread_create (&(tid[i]), NULL, func_thread, (void *)pt_ind ) != 0) {
      printf("pthread_create\n"); exit (1);
    }
  }
```


Terminaison d'une thread

- **Terminaison de la thread courante**

  void pthread_exit (void *etat);

  - Termine la thread courante avec une valeur de retour égale à *etat* (pointeur).
  - Valeur *etat* est accessible aux autres threads du même processus par l'intermédiaire de la fonction *pthread_join*.

Exemple 3 – Création/terminaison de threads

```
#define _POSIX_SOURCE 1
#include <stdio.h>              #include <pthread.h>
#include <stdlib.h>             #include <unistd.h>

#define NUM_THREADS 2
void *func_thread (void *arg) {
   printf ("Argument reçu : %s, thread_id: %d \n", (char*)arg, (int) pthread_self());
   pthread_exit ((void*)0); return NULL;
}
int main (int argc, char ** argv) {
    int i; pthread_t tid [NUM_THREADS];

      for (i=0; i < NUM_THREADS; i++) {
        if  (pthread_create (&(tid[i]), NULL, func_thread, argv[i+1]) != 0) {
            printf ("pthread_create \n"); exit (1);
        }
      }
      sleep (3);
      return EXIT_SUCCESS;
}
```


Relâchement de la CPU par une thread

- **Demande de relâchement du processeur :**

  int sched_yield (void);

  - **La thread appelante demande à libérer le processeur.**
  - **Thread est mise dans la file des *"threads prêtes"*.**
    - **La thread reprendra son exécution lorsque toutes les threads de priorité supérieure ou égale à la sienne se sont exécutées.**

Exemple 4 - relâchement de la CPU

```
#define _POSIX_SOURCE 1
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

#define  NUM_THREADS 3

void *test (void *arg) {
  int i,j;
  for (j=0; j<NUM_THREADS; j++) {
      for (i=0; i <1000; i++);
      printf ("thread %d %d \n",
            (int)pthread_self());
      sched_yield ();
  }
  return NULL;
}
```

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

  for (i=0; i < NUM_THREADS; i++)
    if  (pthread_create (&tid[i], NULL, test,
        NULL)  != 0)  {
     perror("pthread_create \n");
     exit (1);
    }

   sleep (3);
   printf ("fin thread main \n" );

   return EXIT_SUCCESS;
}
```


Types de thread

- **Deux types de thread :**
  - **Joignable (par défaut)**
    - Attribut : PTHREAD_CREATE_JOINABLE
    - En se terminant suite à un appel à *pthread_exit,* les valeurs de son identité et de retour sont conservées jusqu'à ce qu'une autre thread en prenne connaissance (appel à *pthread_join*). Les ressources sont alors libérées.
  - **Détachée**
    - Attribut : PTHREAD_CREATE_DETACHED
    - Lorsque la thread se termine toutes les ressources sont libérées.
    - Aucune autre thread ne peut les récupérer.

Détachement d'une thread

- **Passer une thread à l'état "détachée" (démon).**
- **Les ressources seront libérées dès le *pthread_exit*.**
  - Impossible à une autre thread d'attendre sa fin avec *pthread_join*.
- **Détachement : 2 façons**
  - **Fonction *pthread_detach*** :
    int pthread_detach(pthread_t tid);
  - **Lors de sa création** :
    - **Exemple:**

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


Attente de terminaison d'une thread joignable

- **Synchronisation :**

int pthread_join (pthread_t tid, void **thread_return);

  - Fonction qui attend la fin de la thread *tid*.
    - *thread tid* doit appartenir au même processus que la thread appelante.
    - Si la *thread tid* **n'est pas encore terminée**, la thread appelante sera **bloquée** jusqu'à ce que la *thread tid* se termine.
    - Si la *thread tid* est **déjà terminée**, la thread appelante **n'est pas bloquée**.
    - *Thread tid* doit être **joignable**.
      - Sinon la fonction renverra EINVAL.
    - Une seule thread réussit l'appel.
      - Pour les autres threads, la fonction renverra la valeur ESRCH.
      - Les ressources de la *thread* sont alors libérées.

Attente de terminaison d'une thread joignable (2)

- **Lors du retour de la fonction pthread_join**
  - La valeur de terminaison de la *thread tid* est reçue dans la variable *thread_return* (pointeur).
    - Valeur transmise lors de l'appel à *pthread_exit*
    - Si la thread a été annulée, *thread_return* prendra la valeur PTHREAD_ CANCEL.
- **code de renvoi** :
  - 0 en cas de succès.
  - valeur non nulle en cas d'échec:
    - ESRCH : thread n'existe pas.
    - EDEADLK : interblocage ou ID de la thread appelante.
    - EINVAL : thread n'est pas joignable.

Exemple 5 – attendre la fin des threads

```
#define _POSIX_SOURCE 1
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

#define NUM_THREADS 2

void *func_thread (void *arg) {
  printf ("Argument reçu %s, tid: %d\n"
,
        (char*)arg, (int)pthread_self());
  pthread_exit ((void*)0);
}
int main (int argc, char ** argv) {
  int i,status;
  pthread_t tid [NUM_THREADS];
```

```
for (i=0; i < NUM_THREADS; i++) {
    if  (pthread_create (&(tid[i]), NULL,
                func_thread, argv[i+1]) != 0) {
      printf("pthread_create \n"); exit (1);
    }
  }

 for (i=0; i < NUM_THREADS; i++) {
    if (pthread_join (tid[i], (void**) &status) !=0) {
       printf ("pthread_join"); exit (1);
     }
    else
      printf ("Thread %d fini avec status :%d\n",
          i, status);
  }
  return EXIT_SUCCESS;
}
```

Exemple 6 – transmission de la valeur de terminaison

```
int cod_ret =0;

void *func_thread (void *arg) {
…..
  pthread_exit ((void*)&cod_ret);
}

int main (int argc, char ** argv) {
pthread_t tid;
….
if (pthread_join (tid, (void**) &status) !=0)
            printf ("pthread_join");
        exit (1);
    }

printf ("Thread fini avec status :%d\n",
            *(int*)status);
…
}
```

```
void *func_thread (void *arg) {
int cod_ret =0;
…..
  pthread_exit ((void*)cod_ret);
}

int main (int argc, char ** argv) {
pthread_t tid;
….
if (pthread_join (tid, (void**) &status) !=0) {
            printf ("pthread_join");
        exit (1);
    }

printf ("Thread fini avec status :%d\n", status);
…
}
```


Exclusion Mutuelle –Mutex (1)

- **Mutex:**
  - Sémaphores binaires ayant deux états : *libre* et *verrouillé*
    - Seulement une thread peut obtenir le verrouillage.
      - Toute demande de verrouillage d'un mutex déjà verrouillé entraînera soit le blocage de la thread, soit l'échec de la demande.
  - Variable de type *pthread_mutex_t.*
    - Possède des attributs de type *pthread_mutexattr_t*
- **Utiliser pour:**
  - protéger l'accès aux variables globales/tas.
  - Gérer des synchronisations de threads.

Exclusion Mutuelle – Mutex (2)

- **Création/Initialisation (2 façons) :**
  - **Statique:**
    pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
  - **Dynamique:**
    int pthread_mutex_init(pthread_mutex_t *m, pthread_mutex_attr *attr);
    - Attributs :
      - initialisés par un appel à :
        int pthread_mutexattr_init(pthread_mutex_attr *attr);
    - NULL : attributs par défaut.
    - **Exemple :**
      pthread_mutex_t sem;
      /* attributs par défaut */
      pthread_mutex_init(&sem, NULL);

Exclusion Mutuelle (3)

- **Destruction :**
  int pthread_mutex_destroy (pthread_mutex_t *m);
- **Verrouillage :**
  int pthread_mutex_lock (pthread_mutex_t *m);
  - Bloquant si déjà verrouillé

  int pthread_mutex_trylock (pthread_mutex_t *m);
  - Renvoie EBUSY si déjà verrouillé
- **Déverrouillage:**
  int pthread_mutex_unlock (pthread_mutex_t *m);

Exemple 7 - exclusion mutuelle

```
#define _POSIX_SOURCE 1
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

pthread_mutex_t mutex =
PTHREAD_MUTEX_INITIALIZER;
int cont =0;

void *sum_thread (void *arg) {
  pthread_mutex_lock (&mutex);
  cont++;
  pthread_mutex_unlock (&mutex);

  pthread_exit ((void*)0);
}
```

```
int main (int argc, char ** argv) {
  pthread_t tid;

 if  (pthread_create (&tid, NULL, sum_thread,
                          NULL) != 0) {
     printf("pthread_create"); exit (1);
   }

    pthread_mutex_lock (&mutex);
    cont++;
    pthread_mutex_unlock (&mutex);

    pthread_join (tid, NULL);
    printf ("cont : %d\n", cont);

  return EXIT_SUCCESS;
}
```


Les conditions (1)

- **Utilisée par une thread quand elle veut attendre qu'un événement survienne.**
  - Une thread se met en attente d'une condition (opération bloquante). Lorsque la condition est réalisée par une autre thread, celle-ci signale à la thread en attente qui se réveillera.
- **Associer à une condition une variable du type *mutex* et une variable du type *condition*.**
  - *mutex* utilisé pour assurer la protection des opérations sur la variable *condition*

Les conditions : initialisation (2)

- **Création/Initialisation (2 façons) :**
  - **Statique:**

    pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
  - **Dynamique:**

    int pthread_cond_init(pthread_cond_t *cond,
    pthread_cond_attr *attr);
    - **Exemple :**

      pthread_cond_t cond_var;
      /* attributs par défaut */
      pthread_cond_init (&cond_var, NULL);

Conditions : attente (3)

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

- **Utilisation:**

  pthread_mutex_lock(&mut_var);
  pthread_cond_wait(&cond_var, &mut_var);
  …..
  pthread_mutex_unlock(&mut_var);

  - Une thread ayant obtenu un *mutex* peut se mettre en attente sur une variable condition associée à ce *mutex*.
  - **pthread_cond_wait:**
    - Le mutex spécifié est libéré.
    - La thread est mise en attente sur la variable de condition *cond.*
    - Lorsque la condition est signalée par une autre thread, le *mutex* est acquis de nouveau par la thread en attente qui reprend alors son exécution.

Conditions : notification (4)

- **Une thread peut signaler une condition par un appel aux fonctions :**

  int pthread_cond_signal(pthread_cond_t *cond);

  - réveil d'une thread en attente sur *cond*.

  int pthread_cond_braodcast(pthread_cond_t *cond);

  - réveil de toutes les threads en attente sur *cond*.

- Si aucune thread n'est en attente sur *cond* lors de la notification, cette notification sera perdue.

Exemple 8 - Conditions

```
#define _POSIX_SOURCE 1
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

pthread_mutex_t mutex_fin =
       PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond_fin =
     PTHREAD_COND_INITIALIZER;

void *func_thread (void *arg) {
  printf ("tid: %d\n", (int)pthread_self());

  pthread_mutex_lock (&mutex_fin);
  pthread_cond_signal (&cond_fin);
  pthread_mutex_unlock (&mutex_fin);
  pthread_exit ((void *)0);
}
```

```
int main (int argc, char ** argv) {
  pthread_t tid;

  pthread_mutex_lock (&mutex_fin);
if  (pthread_create (&tid , NULL, func_thread,
     NULL) != 0) {
       printf("pthread_create erreur\n"); exit (1);
    }
    if (pthread_detach (tid) !=0 ) {
      printf ("pthread_detach erreur"); exit (1);
    }

    pthread_cond_wait(&cond_fin,&mutex_fin);
    pthread_mutex_unlock (&mutex_fin);
    printf ("Fin thread \n");

  return EXIT_SUCCESS;
}
```

Les Conditions (5)

- **Tester toujours la condition associée à la variable contrôlée (*var*)**
  - Si plusieurs *Pthreads* sont en attente sur la condition, il se peut que la condition sur la variable contrôlée *var* ne soit plus satisfaite :

```
pthread_mutex_lock (&mutex);
while (! condition (var) ) {
   pthread_cond_wait(&cond,&mutex);
}
….
   pthread_mutex_unlock (&mutex);
```


Les Conditions (6)

- **Attente temporisée**

  int pthread_cond_timedwait (pthread_cond_t * cond, phtread_mutex_t* mutex, const struct timespec * abstime);
  - Fonction qui automatiquement déverrouille le *mutex* et attend la condition comme la fonction *pthread_cond_wait*. Cependant, le temps pour attendre la condition est borné.
    - spécifiée en temps absolu comme les fonctions *time ()* ou *gettimeofday()*.
    - Si la condition n'a pas été signalée jusqu'à *abstime*, le *mutex* est réacquis et la fonction se termine en renvoyant le code ETIMEDOUT.