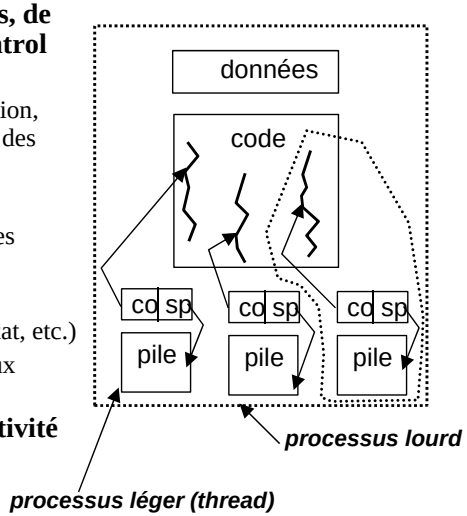
Cours 4 et 5
Processus légers - Threads

Cours 4 - PTHREADS

- **Définition**
- **Création et terminaison**
- **Synchronisation entre pthreads**
- **Attributs**
- **Annulation**
- **Intégration avec d'autres outils POSIX**
  - **Signaux**
  - **Sémaphores**
  - **Fork**

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 : variable

```
void *func_thread (void *arg) {
int  *pt = malloc (sizeof (…) );
…..
   pthread_exit ((void*)pt);
}

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

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

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_broadcast(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;
}
```


Exemple 9 - Conditions

```
#define _POSIX_SOURCE 1
#include <pthread.h>
…
int flag=0;
pthread_mutex_t m =
      PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond=
     PTHREAD_COND_INITIALIZER;

void *func_thread (void *arg) {
  pthread_mutex_lock (&m);
  while (! flag) {
    pthread_cond_wait (&cond,&m);
  }
  pthread_mutex_unlock (&m);
  pthread_exit ((void *)0);
}
```

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

if  ((pthread_create (&tid1 , NULL, func_thread,
     NULL) != 0) | |   (pthread_create (&tid2 ,
     NULL, func_thread, NULL) != 0))  {
       printf("pthread_create erreur\n"); exit (1);
    }

   sleep(1);
    pthread_mutex_lock (&m);
     flag=1;
    pthread_cond_broadcast(&cond,&m);
    pthread_mutex_unlock (&m;

  pthread_join (tid1, NULL);
  pthread_join (tid2, NULL);

  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.

Attributs des Threads (1)

- **Chaque thread possède un nombre d'attributs regroupé dans le type *pthread_attr_t*.**
  - Chaque attribut possède une valeur par défaut.
  - Attributs fixés lors de la création de la thread.
    - Paramètre du type *pthread_attr_t* de la fonction *pthread_create ()*.
      - NULL : attributs auront les valeurs par défaut.
    - Possibilité de changer dynamiquement les attributs.

Attributs des Threads (2)

- **Fonction pour créer une variable du type *pthread_attr_t :***

  int pthread_attr_init (pthread_attr_t * attributs);
  - Attributs initialisés avec les valeurs par défaut.
- **Fonction pour détruire une variable du type *pthread_attr_t :***

  int pthread_attr_destroy (pthread_attr_t * attributs);
- **Fonctions pour obtenir et modifier respectivement la valeur d'un attribut d'une variable du type *pthread_attr_t :***

  int pthread_attr_get**nom** (pthread_attr_t * attributs,...);

  int pthread_attr_set**nom** (pthread_attr_t * attributs,…);
  - **nom** : *nom* de l'attribut

Attributs des Threads (3)

- **Detachstate**
  - Thread joignable ou détachée :
    - PTHREAD_CREATE_JOINABLE (valeur par défaut)
    - PTHREAD_CREATE_DETACHED
  - Fonction pour obtenir l'état de détachement d'une thread :

pthread_attr_getdetachstate(const pthread_attr_t * attr, int * valeur);
  - Fonction pour modifier l'état de détachement d'une thread :

pthread_attr_setdetachstate(const pthread_attr_t * attr, int valeur);

Attributs des Threads (4)

- **Configuration de la pile :**
  - Obtenir et/ou modifier la taille et l'adresse de la pile
  - **Fonction pour obtenir la taille et l'adresse de la pile respectivement :**

    int pthread_attr_getstacksize (const pthread_attr_t * attr, size_t valeur);

    int pthread_attr_getstackaddr(const pthread_attr_t * attr, void ** valeur);
  - **Fonction pour modifier la taille et l'adresse de la pile respectivement :**

    int pthread_attr_setstacksize(const pthread_attr_t * attr, size_t valeur);

    int pthread_attr_setstackaddr(const pthread_attr_t * attr, void *valeur);
  - Peuvent entraîner des problèmes de portabilité

Attributs des Threads (5)

- **Configuration de la pile (cont.) :**
  - Valeurs de la taille et de l'adresse de la pile sont disponibles si les constantes suivantes on été définies dans le fichier <*unistd.h*> respectivement :
    - _POSIX_THREAD_ATTR_STACKSIZE
    - _POSIX_THREAD_ATTR_STACKADDR
  - Taille minimum d'une pile *(*<*unistd.h*>)
    - _PTHREAD_STACK_MIN

Attributs des Threads (6) - Exemple

```
….
void *thread_func (void *arg) {
….
  return NULL;
}
int main(int argc, char** argv) {
  int ret; size_t size; pthread_t tid;
  pthread_attr_t attr;

  if ((ret = pthread_attr_init (&attr)) !=0) {
    printf ("erreur  : %d\n", ret);  exit (1);
  }
  if ((ret = pthread_attr_getstacksize
            (&attr,&size)) !=0) {
     printf ("erreur : %d \n", ret);  exit (1);
}
```

```
else
  printf ("Taille: %d; taille min :%d\n",
            size, PTHREAD_STACK_MIN);

 if ((ret = pthread_attr_setstacksize
  (&attr, PTHREAD_STACK_MIN*2)) !=0) {
   printf ("erreur : %d \n", ret);   exit (1);
 }

   if ((pthread_create(&tid, &attr, thread_fun
        NULL)) !=0) {
       printf ("erreur: %d \n", ret);
       exit (1);
   }
…
}
```


Attributs des Threads (7) Ordonnancement

- **Quatre attributs associés à l'ordonnancement :**
  - inheritsched
  - scope
  - schedpolicy
  - schedparam
  - Disponibles si _POSIX_THREAD_PRIORITY_SCHEDULING est définie dans <*unistd.h*>.
- **Attribut inheritedsched**

  int pthread_attr_getinheritsched (const pthread_attr_t * attr, int* valeur);
  int pthread_attr_setinheritsched (const pthread_attr_t * attr, int valeur);
  - Valeurs possibles :
    - PTHREAD_EXPLICIT_SCHED : l'ordonnancement spécifié à la création de la thread.
    - PTHREAD_IMPLICIT_SCHED : l'ordonnancement (valeurs schedpolicy et schedparam) hérité de la thread appelante

Attributs des Threads (8) Ordonnancement

- **Attribut scope**
  - Pour les implémentations hybrides (ex. Solaris) :

    int pthread_attr_getscope (const pthread_attr_t * attr, int* valeur);
    int pthread_attr_setscope (const pthread_attr_t * attr, int valeur);
    - Valeurs possibles :
      - PTHREAD_SCOPE_SYSTEM : à chaque *Pthread* est associée une thread noyau.
      - PTHREAD_SCOPE_PROCESS : Les *Pthreads* d'un même processus sont prises en charge par un pool de threads noyau. Un nombre maximum de threads noyau existent pour un processus.
        - La taille du pool peut être consultée / modifiée par :

          int pthread_getconcurrency (void);
          int pthread_setconcurrency (int valeur);

Attributs des Threads (9) Ordonnancement

- **Attribut schedpolicy**

  int pthread_attr_getschedpolicy (const pthread_attr_t * attr, int* valeur);

  int pthread_attr_setschedpolicy (const pthread_attr_t * attr, int valeur);

  - Valeurs possibles :
    - SCHED_OTHER : ordonnancement classique temps partagé
    - SCHED_FIFO : Temps-réel. Politique *FIFO*.
      - Thread avec priorité fixe et ne peut être préemptée que par une autre thread ayant une priorité strictement supérieure.
    - SCHED_RR: Temps-réel. Politique *Round-Robin*.
      - Après un quantum, la CPU peut être affectée à une autre Thread de priorité au moins égale.

Attributs des Threads (10) Ordonnancement

- **Attribut schedparam**

  int pthread_attr_getschedparam (const pthread_attr_t * attr, struct schedparam* sched);

  int pthread_attr_setschedparam(const pthread_attr_t * attr, struct schedparam sched);

  - *struct schedparam* possède le champ:
    - *int sched_priority :* priorité du processus.
    - Valeur comprise entre :
      - *sched_get_priority_min (classe)* et *sched_get_priority_max (classe).*
        - Classe : SCHED_OTHER, SCHED_RR, SCHED_FIFO

Attributs des Threads (11) Ordonnancement

- **Modification dynamique des attributs d'ordonnancement :**
  - Attributs *schedpolicy* et *schedparam* d'une thread peuvent être consultés et/ou modifiés avec les fonctions:

  int pthread_getschedparam (pthread tid, int *classe, struct sched_param* sched);

  int pthread_setschedparam (const pthread_attr_t * attr, int class, struct sched_param sched);

Attributs des Threads (12) Ordonnancement – Exemple

```
void *thread_func (void *arg) {
    int ret; int policy; struct sched_param sched;
    ret = pthread_getschedparam (pthread_self(), &policy, &sched);
    if (ret)
        exit (1);
    else
        printf ("politique : %s, priorité :%d\n",
                (policy == SCHED_OTHER ? "SCHED_OTHER" :
                 (policy == SCHED_FIFO ? "SCHED_FIFO" :
                 (policy == SCHED_RR ? "SCHED_RR" : "inconnu"))),
                 sched.sched_priority);
    return NULL;
}
```

Annulation d'une thread (1)

- **Une thread peut vouloir annuler l'autre.**
  - Une thread envoie une demande d'annulation à une autre qui sera prise en compte ou non en fonction de la configuration de celle-ci.
    - La thread qui reçoit une demande d'annulation peut la refuser ou la repousser jusqu'à atteindre un ***point d'annulation***.
  - Lorsque la thread annulée se termine, elle exécute toutes les fonctions de terminaison programmées.

Annulation d'une thread (2)

- **Demande d'annulation:**

  int pthread_cancel (pthread_t tid);
  - **Code de renvoi**: 0 ou *ESRCH* (si la thread n'existe pas).
- **Annulation d'une thread peut entraîner des incohérences :**
  - Exemples :
    - accès à une variable globale, abandon d'un mutex verrouillé, etc.
- **Solution :**
  - interdire temporairement les demandes d'annulation dans certaines portions du code.

Annulation d'une thread (3)

- **Interdiction temporaire des demandes d'annulation :**

  int pthread_setcancelstate (int etat_annulation, int *ancien_etat);
  - Fonction qui permet de configurer le comportement d'une thread vis-à-vis d'une requête d'annulation. Permet aussi de récupérer l'ancien état.
    - Possibles valeurs pour *etat_annulation* :
      - PTHREAD_CANCEL_ENABLE
        - La thread acceptera les requêtes d'annulation (**par défaut**)
      - PTHREAD_CANCEL_DISABLE
        - La thread ne tiendra pas compte des requêtes d'annulation.

Annulation d'une thread (4)

- **Interdiction temporaire des demandes d'annulation (PTHREAD_CANCEL_DISABLE) :**
  - Les requêtes d'annulation ne restent pas pendantes, contrairement aux signaux masqués.
    - Une thread désactivant temporairement les requêtes pendant une section critique ne se terminera pas lorsqu'elle autorise de nouveau les annulations.
  - Solution :
    - Utiliser un mécanisme de synchronisation afin de retarder des annulations jusqu'à atteindre des points bien définis dans le processus.

Annulation d'une thread (5)

- **Type d'annulation :**

int pthread_setcanceltype (int type_annulation, int *ancien_type);

- Possibles valeurs pour *type_annulation* :
  - PTHREAD_CANCEL_DEFERRED
    - La thread ne terminera qu'en atteignant un point d'annulation (**par défaut**)
  - PTHREAD_CANCEL_ASYNCHRONOUS
    - L'annulation prendra effet dès la réception de la requête d'annulation.

Annulation d'une thread (6)

- **Fonctions qui constituent des points d'annulation :**
  - *pthread_cond_wait ()* et *pthread_cond_timed_wait ();*
  - *pthread_join ();*
  - *pthread_testcancel ();*
    - Fonction *void pthread_testcancel (void)* :
      - Permet à une thread de tester si une requête d'annulation lui a été adressée.
        - Dès qu'elle est invoquée, la thread peut se terminer si une demande d'annulation est en attente.
      - Répartir des appels à *pthread_testcancel ()* aux endroits du code où on est sûr qu'une annulation ne posera pas de problème.

Annulation d'une thread (7)

- **Ensemble des fonctions et appels système bloquants qui sont des points d'annulation :**
  - Exemples :
    - *open (); close (); create ();*
    - *fcntl(); fsync();*
    - *pause ();*
    - *read();*
    - *sem_wait();*
    - *sigsuspend();*
    - *sleep();*
    - *wait, waitpid ();*
    - *etc.*

Annulation d'une thread (8)

- **Résumé des configurations**
  - PTHREAD_CANCEL_DISABLE
    - Région critique où on n'accepte pas des annulations.
  - PTHREAD_CANCEL_ENABLE + PTHREAD_CANCEL_DEFERRED
    - Comportement par défaut.
  - PTHREAD_CANCEL_ENABLE + PTHREAD_CANCEL_ASYNCHRONOUS
    - Boucle de calcul sans appels système qui utilise beaucoup de CPU.

Annulation d'une thread (9)

- **Une thread peut être annulée à tout moment**
  - nécessité de libérer les ressources que la thread possède avant qu'elle ne se termine.
    - Fichiers ouverts, mutex verrouillé, mémoire allouée, etc.
- **Solution :**
  - Lorsqu'une thread alloue une ressource qui nécessite une libération ultérieure, elle enregistre le nom d'une routine de libération dans une pile spéciale en utilisant la fonction ***pthread_cleanup_push ()***.
  - Quand la thread désire libérer explicitement la ressource, elle appelle ***pthread_cleanup_pop ()***.

Annulation d'une thread (10)

- Enregistrer des routines de libération dans une "pile de nettoyage":

  void pthread_cleanup_push (void (*fonction)(void *), void *arg);
  - Lorsque la thread se termine, les fonctions sont dépilées, dans l'ordre inverse d'enregistrement et exécutées.

  void pthread_cleanup_pop (int exec_routine);
  - Retire la routine au sommet de la pile.
  - *exec_routine* :
    - si non nul la routine est invoquée.
    - si 0, la routine est retirée de la pile de nettoyage sans être exécutée.

Annulation d'une thread (11)

```
void *func_thread (void *arg) {
 char * buf;  FILE * fich;
 ….
 buf = malloc (TAILLE_BUF);
 if ( buf != NULL) {
    pthread_cleanup_push(free,buf);
    fich = fopen ("FICH","r");
     if (fich !=NULL) {
        pthread_cleanup_push(fclose,fich);
        …
        pthread_cleanup_pop(1);
  }
  …
  pthread_cleanup_pop(1);
 }
}
```

• **Observation :**
Les appels aux fonctions *pthread_cleanup_push( )* et *pthread_cleanup_pop( )* doivent se trouver dans la même fonction et au même niveau d'imbrication.

Annulation d'une thread (12)

- **Conditions**
  - L'annulation d'une thread doit laisser le *mutex* associé à la *condition* dans un état cohérent
    - *mutex* doit être libéré.

```
pthread_mutex_lock (&mutex);
pthread_cleanup_push(pthread_mutex_unlock, (void*) &mutex);
while (! condition (var) ){
   pthread_cond_wait(&cond,&mutex);
}
….
/* pthread_mutex_unlock (&mutex); */
pthread_cleanup_pop(1);
```

Pthreads et les signaux (1)

- **La gestion d'un signal est assurée pour l'ensemble de l'application en employant la fonction *sigaction ()*.**
- **Chaque thread possède son masque de signaux et son ensemble de signaux pendants.**
  - Le masque d'une thread est hérité à sa création du masque de la thread la créant.
  - Les signaux pendants ne sont pas hérités.

  int pthread_sigmask (int mode, sigset_t *pEns, sigset_t *pEnsAnc);
    - Permet de consulter ou modifier le masque de signaux de la thread appelante.

Pthreads et les signaux (2)

- **Envoi d'un signal à une thread**

  int pthread_kill (pthread_t tid, int signal);
  - Même comportement que la fonction *kill()*.
  - L'émission du signal au sein du même processus
  - Renvoie 0 en cas de succès ou ESRCH si la *thread tid* n'existe pas
- **Attente de signal**

  int sigwait (const sigset_t *ens, int *sig)
  - Extrait un signal de la liste de signaux pendants appartenant à *ens*. Le signal est récupéré dans *sig* et renvoyé comme valeur de retour de la fonction. S'il n'existe aucun signal pendant, celle-ci est bloquée.

Pthreads et les signaux (3)

- **Signal traité par une thread spécifique**
  - **synchrone**
    - événement lié à l'exécution de la thread active. Signal est délivré à la thread fautive.
      - SIGBUS, SIGSEGV, SIGPIPE
    - Signal envoyé par une autre thread en utilisant *pthread_kill*
- **Signal traité par une thread quelconque**
  - **asynchrone** – reçu par le processus.
    - Le signal sera pris en compte par une des threads du processus parmi celles qui ne masquent pas le signal en question.

Pthreads et les signaux (4)

- **Observation :**
  - Les fonctions POSIX qui permettent de manipuler les *Pthreads* ne sont pas nécessairement réentrantes. Par conséquent elles ne doivent pas être appelées depuis un gestionnaire de signaux.
- **Solution :**
  - Créer une thread dédiée à la réception des signaux, qui boucle en utilisant *sigwait ()*.

Pthreads et les signaux (5)

int sigwait (const sigset_t * masque, int * num_sig);

- Attente de l'un des signaux contenus dans le champ *masque* .
  - Si un signal est pendant, la fonction se termine en sauvegardant le signal reçu dans **num_sig*.
- Point d'annulation
- Possibilité d'utiliser les fonctions de la bibliothèque Pthreads.
- Toutes les autres threads doivent bloquer les signaux attendus.

Exemple 1 - signaux et Pthreads (6)

```
pthread_mutex_t mutex_sig = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond_cont = PTHREAD_COND_INITIALIZER;
int sig_cont;

void * thread_sig (void *arg){
  sigset_t ens;  int sig;
  sigemptyset (&ens); sigaddset (&ens,SIGINT);

  while (1) {
    sigwait (&ens,&sig);
    pthread_mutex_lock (&mutex_sig);
    sig_cont ++;
    pthread_cond_signal (&cond_cont);
    if (sig_cont == 5) {
      pthread_mutex_unlock (&mutex_sig);
      pthread_exit ((void *)0);
    }
    pthread_mutex_unlock (&mutex_sig);
  }
}
```


Exemple 1- signaux et Pthreads (cont) (7)

```
void *thread_cont (void *arg) {
sigset_t ens;
   sigfillset (&ens);
  pthread_sigmask (SIG_SETMASK, &ens,
                   NULL);
  while (1) {
    pthread_mutex_lock (&mutex_sig);

pthread_cond_wait(&cond_cont,&mutex_sig);
    printf ("cont: %d\n",sig_cont);
    if (sig_cont == 5) {
       pthread_mutex_unlock (&mutex_sig);
       pthread_exit ((void *)0);
    }
      pthread_mutex_unlock (&mutex_sig);
  }
}
int main (int argc, char ** argv) {
  pthread_t tid_sig, tid_cont;
  sigset_t ens;
```

```
sigfillset (&ens);
pthread_sigmask (SIG_SETMASK, &ens,NULL);

  if  ( (pthread_create (&tid_cont, NULL,
              thread_cont, NULL) != 0) ||
             (pthread_create (&tid_sig, NULL,
             thread_sig, NULL) != 0)) {
      printf ("pthread_create \n"); exit (1);
  }
  if (pthread_detach (tid_sig) !=0 ) {
      printf ("pthread_detach \n"); exit (1);
  }
    if (pthread_join (tid_cont, NULL) !=0) {
       printf ("pthread_join"); exit (1);
    }
       printf ("fin \n");
   return 0;
}
```


Pthreads et sémaphores POSIX (1)

- **Sémaphore**
  - Variable du type *sem_t* permettant de limiter l'accès à une section critique.
    - #include <semaphore.h>
      - Si la constante _POSIX_SEMAPHORE est définie dans <unistd.h>
- **Création / Destruction**

  int sem_init (sem_t *sem, int partage, unsigned int valeur);
  - *Partage* : si valeur nulle, le sémaphore n'est partagé que par les threads du même processus
  - *Valeur* : valeur initiale du sémaphore
    - Valeur inscrite dans un compteur qui est décrémenté à chaque fois qu'une thread rentre en section critique et incrémenté à chaque sortie.

  int sem_destroy (sem_t *sem);

Pthreads et sémaphores POSIX (2)

- **Entrée/Sortie en section critique**

  int sem_wait (sem_t *sem);
  - Entrée en SC. Fonction bloquante
    - Attendre que le compteur soit supérieur à zéro et le décrémenter avant de revenir.

  int sem_post (sem_t *sem);
  - Sortie de SC. Compteur incrémenté; une thread en attente est libérée.

  int sem_trywait (sem_t *sem);
  - Fonctionnement égal à *sem_wait* mais non bloquante.
- **Consultation compteur sémaphore**

  int sem_getvalue (sem_t *sem, int *valeur);
  - Renvoie la valeur du compteur du sémaphore *sem*. dans *valeur.

Exemple 2 - Sémaphore POSIX (3)

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

#define NUM_THREADS 4
sem_t sem;

void *func_thread (void *arg) {
 sem_wait (&sem);
 printf ("Thread %d est rentrée en SC \n",
          (int) pthread_self ());
  sleep ((int) ((float)3*rand()/ (RAND_MAX
+1.0)));
  printf ("Thread %d est sortie de la SC \n",
           (int) pthread_self ());
 sem_post(&sem);
 pthread_exit ( (void*)0);
}
```

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

sem_init (&sem,0,2);

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


Pthread et fork (1)

- **Lors du *fork***
  - Le processus est dupliqué, mais il n'y aura dans le processus fils que la thread qui a invoqué le *fork ()*.
- **Si une thread recouvre le code est les données par un appel à *exec*, toutes les autres threads du processus se terminent.**

Pthread et fork (2)

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

Pthread tid[2]; char *nom[2]={"T1", T2"};

void *func_thread (void *arg) {
  printf ("Thread %s  tid:%d,  pid %d avant fork
\n", arg, (int) pthread_self());
  if (pthread_self () == tid[0])
    if (fork () == 0)
      printf ( "Fils: %s tid: %d pid: %d \n", arg,
(int)
                pthread_self(), getpid());
   else
       printf ( "Père: %s  tid: %d pid: %d \n",
arg,
              (int)pthread_self(), getpid())
else
    print ("Fin %s  tid:%d, pid %d \n",
            argv, (int)pthread_self(), getpid());
}
```

```
int main (int argc, char ** argv) {
  int i;
for (i=0; i <2; i++)
   if  (pthread_create (&(tid[i]), NULL,
                  func_thread, arg[i]) != 0) {
     printf ("pthread_create"); exit (1);
   }
  for (i=0; i < NUM_THREADS; i++)
   if (pthread_join (tid[i], NULL) !=0) {
     printf ("pthread_join"); exit (1);
  }
 }
```

Thread T1 tid:34 pid 56 avant fork
Thread T2 tid:35 pid 56 avant fork
Fils: T1 tid: 34 pid: 57
Père: T1 tid: 34 pid: 56
Fin T2 tid:35, pid 56

Pthread et fork (3)

- **Problème :**
  - une autre thread du processus père a pris/verrouillé une ressource dont le fils aura besoin.
- **Exemple Problème :**
  - Thread1 et Thread2.
    - *Thread1* verrouille une ressource partagée en utilisant *mutex*
    - *Thread2* appelle fork *()*;
      - La seule thread du fils sera *Thread2*
    - *Thread2* du processus fils veut accéder à la ressource critique.
    - *Thread1* du processus père continue à exécuter et libère *mutex*
    - *Thread2 a*ttend la libération du verrou par *Thread1*, mais celle-ci n'existe pas dans le processus fils.
      - Processus fils **bloqué pour toujours**.

Pthread et fork (4)

- **Solution pour les ressources partagées** :
  - fonction *pthread_atfork* qui permet d'enregistrer les routines qui seront automatiquement invoquées si une thread appelle le *fork*.

```
int pthread_atfork (void (*avant) (void),
                    void (*dans_pere) (void), void (*dans_fils) (void));
```

    - *avant*: fonction appelée avant le *fork*.
    - *dans_pere* et *dans_fils* : fonctions appelées par le père et par le fils respectivement après le *fork* () au sein de la thread ayant invoqué le *fork*.

Pthread et fork (5)

- **Solution Problème Thread1 et Thread2:**
  - Installer avec pthread_atfork() les fonctions :
    - *avant() : pthread_mutex_lock ()*
    - *dans_père () : pthread_mutex_unlock ()*
    - *dans_fils () : pthread_mutex_unlock ()*
- **Exécution :**
  - *Thread1* verrouille *mutex* ;
  - *Thread2* appelle fork *()*;
    - *avant ()* est exécutée en bloquant *Thread2.*
  - *Thread1* libère *mutex*
    - *avant ()* se termine et le *fork()* a lieu.
      - *dans_père ()* et *dans fils ()* sont exécutées, libérant le verrou dans les deux processus.

Exécution unique de fonction (1)

- **Lorsque plusieurs threads appellent une même fonction, parfois il est souhaitable que cette fonction soit exécutée une seule fois**
  - Utiliser une variable statique de type *pthread_once_t* initialisée à *PTHREAD_ONCE_INIT*;
  - La fonction à n'exécuter qu'une seule fois est appelée en utilisant :

```
int pthread_once (pthread_once_t *once, void (*func));
```

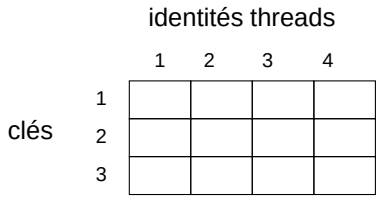
Exécution unique de fonction (2) Exemple

```
static pthread_once_t once = PTHREAD_ONCE_INIT;
 void foncInit (void) {
      .....
}

void * func_thread (void *arg) {
      …
      pthread_once (&once, foncInit);
 ….
}

int main (int argc, char* argv[]) {
I   int i; pthread_t tid[3];
     for (i=0; i<3; i++)
          pthread_create(&tid [i], NULL, func_thread,NULL);
….
}
```

FoncInit ne sera qu'appelée par la première thread qui exécutera *pthread_once*

Données privées (1)

- **Une thread peut posséder des données privées.**
  - Un ensemble de données statiques est réservé et réparti entre les threads d'un même processus.
    - Ensemble peut être vu comme une matrice :



Données privées (2)

- Fonction qui permet la création d'un nouvelle clé :

  int pthread_key_create (pthread_key_t *cle, (void*) destruc (void*));
  - Si *destruc* égal à NULL, l'emplacement créé n'est pas supprimé à la terminaison de la *thread*. Sinon, la fonction spécifiée dans *destruc* est appelée à la terminaison de la thread.
- Fonction qui permet de consulter une donnée privée :

  void * pthread_getspecific(pthread_key_t cle);
- Fonction qui permet modifier une donnée privée :

  int pthread_setspecific(pthread_key_t cle, void *valeur);
  - valeur est typiquement l'adresse d'une zone allouée dynamiquement.

Données privées (3) - Exemple

```
…..
pthread_key_t cle;
void *thread_func (void *arg) {
 int i;  int *pt, *pt2;
 i = *((int*) arg);
 if ((pt = (int*) malloc (sizeof (int))) == NULL)
   exit (1);
 else     *pt = i;
 if (pthread_setspecific (cle, pt) !=0)
      exit (1);
 else   *pt +=2;
 if ((pt2= pthread_getspecific (cle ))== NULL)
    exit (1);
 else
    printf ( "Thread %d - valeur : %d \n",i, *pt2);
 return NULL;
}
```

```
int main(int argc, char** argv) {
 pthread_t tid [2]; int* pt_ind; int i;

 if (pthread_key_create
       (&cle,NULL) !=0)
       exit (1);
  for (i=0; i < 2; i++) {
     pt_ind = (int *) malloc (sizeof (i));
    *pt_ind =i;
     if  (pthread_create (&(tid[i]),
                 NULL, thread_func,
                 (void *)pt_ind ) != 0)
          exit (1);
  }
/* join */
……
}
```

Thread 0 – valeur : 2
Thread 1 – valeur : 3