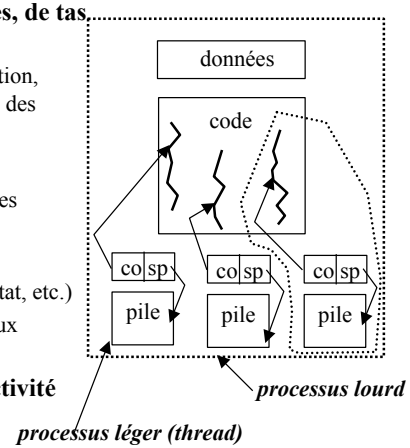
Master Informatique **M1** - Spécialité SAR
UE MI014 : Programmation Système Avancée

Cours 2 & 3

Processus Légers

Processus léger ou "Thread"

- **Partage les zones de code, de données, de tas + des zones du PCB :**
  - 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 de 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 aux applications client/serveur :
    - un thread de connexion + un thread par requête
- **Inconvénients**
  - Programmation plus difficile (mutex, interblocages)
  - Fonctions de librairie non ***multi-thread-safe***

Threads Noyau x Threads Utilisateur

- **Bibliothèque Pthreads:**
  - les threads définis par la norme POSIX 1.c sont indépendants de leur implémentation.
- **Deux types d'implémentation :**
  - **Thread usager (pas connu du noyau):**
    - L'état est maintenu en espace utilisateur. Aucune ressource du noyau n'est allouée à un thread.
    - Des opérations peuvent être réalisées indépendamment du système.
    - Le noyau ne voit qu'un seul thread
      - Tout appel système bloquant un thread aura pour effet de bloquer son processus et par conséquent tous les autres threads du même processus.
  - **Thread Noyau (connu du noyau):**
    - Les threads sont des entités du système (threads natifs).
    - 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'un thread	Nécessite un appel système (*eg.clone*)	Pas d'appel système - coûteuse en ressources
Commutation entre deux threads	Faite par le noyau avec changement de contexte	Assurée par la bibliothèque + 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 un prio. indépendante	Priorité égale à celle du processus
Parallélisme	Répartir les threads entre différents processeurs	Threads doivent s'exécuter sur le même processeur

Pthreads utilisant des threads Noyau

- **Trois différents approches:**
  - **M-1 (many to one)**
    - Tous les *Pthreads* d'un processus sont associés à un même thread système.
      - Ordonnancement des threads est fait par le processus
        - Approche thread utilisateur
  - **1-1 (one to one)**
    - A chaque *Pthread* correspond un thread noyau.
      - Les *Pthreads* sont traités individuellement par le système.
  - **M-M (many to many)**
    - différents *Pthreads* sont multiplexés 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)** :
    - réentrante vis-à-vis du parallélisme
  - Fonction **async-safe** :
    - réentrante vis-à-vis des signaux

POSIX thread API

Orienté objet:

- ***pthread_t*** : identifiant d'un thread
- ***pthread_attr_t*** : attribut d'un 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

- **Un Pthread est identifié par un *ID* unique**
- **En général, en cas de succès une fonction renvoie une valeur non nulle en cas d'échec, 0 sinon**
- **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 …. –lpthread

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* d'un attribut d'un objet.
  - ***_set****attr* : modifier l'attribut *attr* d'un attribut d'un objet.
- **Exemples :**
  - *pthread_create* : crée un thread (objet *pthread_t*).
  - *pthread_mutex_init* : initialise un objet du type *pthread_mutex_t*.

Gestion des Threads

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

Gestion de Threads: attributs

- **Attributs passés au moment de la création du 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_att_get****nom*** *(pthread_attr_t *attr, ...)*
    - Extraire la valeur de l'attribut ***nom*** de la variable *attr*
  - *pthread_att_set****nom*** *(pthread_attr_t *attr, ...)*
    - Modifier la valeur de l'attribut ***nom*** de la variable *attr*

Gestion de Threads: attributs

- **Nom :**
  - **scope** (*int*) - thread natif ou pas
    - PTHREAD_SCOPE_SYSTEM, PTHREAD_SCOPE_PROCESS (*default*)
  - **stackaddr** (*void* *) - adresse de la pile (*default : NULL*)
  - **stacksize** (*size_t*) - taille de la pille (*default : 1Mo*)
  - **detachstate** (*int*) - thread joignable ou détaché
    - PTHREAD_CREATE_JOINABLE (*default*), PTHREAD_CREATE_DETACHED
  - **schedpolicy** (*int*) – type d'ordonnancement
    - SCHED_OTHER (unix) (*default*) , 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 (*default*), PTHREAD_EXPLICIT_SCHED

Gestion de Threads: attributs

- **Exemples de fonctions :**
  - **Obtenir/modifier l'état de détachement d'un 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'un thread**

    int pthread_attr_get**stackaddr** (const pthread_attr_t *attributs, void ** valeur);

    int pthread_attr_set**stackaddr**(const pthread_attr_t *attributs, void* valeur);

Gestion des Threads - Création

- **Créer 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, le thread est créé 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.

Gestion des Threads – obtention et comparaison de identificateurs

- **Obtention de l'dentité du thread courant :**

  pthread_t pthread_self(void);
  - renvoie l'identificateur du thread courant.
- **Comparaison entre deux IDs 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 le même thread.

Gestion des Threads – terminaison

- **Terminaison du thread courant**

  void pthread_exit(void *etat);

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

Exemple – 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, pthread_self());
  pthread_exit ((void*)0);
}
int main (int argc, char ** argv) {
     int i;
     pthread_t tid [argc];

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


Gestion des Threads – Création

- **Passage d'arguments par référence (void *)**
  - ne pas passer en argument l'adresse d'une variable qui peut être modifiée par le thread *main* avant/pendant la création du nouveau 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);
    }
  }
```


Gestion de Threads - Annulation

- **Un thread peut annuler un autre thread :**

  int pthread_cancel (pthread_t tid);

  - **Code de renvoi** : 0 ou *ESRCH* (si le thread n'existe pas).
  - **Par défaut** la requête d'annulation est différée jusqu'à ce que le *Phtread* atteigne un point d'annulation :
    - Appel à des primitives bloquantes :
      pthread_cond_wait, phtread_join, open, read, pause, sem_wait, sigsuspend, ...
    - Explicitement en appelant la fonction :
      int pthread_testcancel ( );
      Permet à un thread de tester si une requête d'annulation lui a été adressée.

Gestion des Threads: type de threads

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

Gestion des Threads - Attente de Terminaison d'un thread

- **Attendre la fin d'un thread joignable**

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

  - Fonction bloquante : attendre la fin (*pthread_exit*) du thread *tid*
  - Valeur de terminaison reçue dans la variable *thread_return.*
  - Les ressources du *thread* sont alors libérées.
  - **code de renvoie:**
    - 0 en cas de succès
    - valeur non nulle en cas d'échec:
      - ESRCH : thread n'existe pas
      - EDEADLK : interblocage ou ID du thread appelant.
      - EINVAL : thread n'est pas joignable.

Exemple – attendre la fin d'un thread

```
#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 (0);
}
int main (int argc, char ** argv) {
  int i,status;
  pthread_t tid [NUM_THREADS];
```

```
for (i=1; i <= NUM_THREADS; i++) {
    if  (pthread_create (&(tid[i-1]), NULL,
                func_thread, argv[i]) != 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;
}
```


Détachement d'un thread

- **Passer un thread à l'état "détaché" (démon).**
- **Les ressources seront libérées dès le *pthread_exit.***
  - Impossible pour un 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);
```

Exclusion mutuelle

- **Outils de base pour :**
  - protéger l'accès aux variables globales/tas
  - Gérer des synchronisations entre threads
- **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);
    - **Exemple :**

      pthread_mutex_t sem;
      /* attributs par défaut */
      pthread_mutex_init(&sem, NULL)

Exclusion Mutuelle

- **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
- **Déverouillage:**

  int pthread_mutex_unlock (pthread_mutex_t *m);

Exemple - 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 (0);
}
```

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

  if  (pthread_create (&tid, NULL, sum_thread,
                       NULL) != 0) {
     perror ("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

- **Utilisées par les threads pour attendre des occurrences d'événements**
  - Un thread se met en attente d'une condition (opération bloquante).
  - Lorsque la condition est réalisée par un autre thread, celui-ci notifie le 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

- **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

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);
  - Un 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
    - Le thread est mis en attente sur la variable de condition *cond*
    - Lorsque la condition est signalée par un autre thread, le *mutex* est acquis de nouveau par le thread en attente qui reprend alors son exécution.

Conditions : notification

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

  int pthread_cond_signal(pthread_cond_t *cond);
  - réveil d'un thread en attente sur *cond*.

  int pthread_cond_broadcast(pthread_cond_t *cond);
  - réveil de tous les threads en attente sur *cond*.

- Si aucun thread n'est en attente sur *cond* lors de la notification, celle-ci sera perdue.

Conditions : exemple

```
#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;
}
```

Pthreads et les signaux

- **Chaque thread possède son masque de signaux et son ensemble de signaux pendants.**
  - Un thread hérite du masque de son créateur.
  - Les signaux pendants ne sont pas hérités.
- **Signal traité par un thread spécifique**
  - synchrone
    - Evénement lié à l'exécution de la thread active. Signal est délivré à la thread fautive.
      - SIGBUS, SIGSEGV, SIGPIPE
    - Signal envoyé par un autre thread en utilisant *pthread_kill*
- **Signal traité par un thread quelconque**
  - asynchrone – reçu par le processus.
    - Le signal sera pris en compte par un des threads du processus parmi ceux qui ne masquent pas le signal en question.

Pthreads et les signaux

- **Masque de signaux**

  int pthread_sigmask (int mode, sigset_t *pEns, sigset_t *pEnsAnc);
  - Permet de consulter ou modifier le masque de signaux du thread appelant.
- **Envoie d'un signal à une thread**

  int pthread_kill (pthread_t tid, int signal);
  - Renvoie 0 en cas de succès ou ESRCH si le thread *tid* n'existe pas
- **Attente de signal**

  int sigwait (const sigset_t *ens, int *sig)
  - Bloque le processus appelant tant qu'aucun des signaux de *ens* n'est pendant.
  - Dans le cas contraire, un des signaux pendants de *ens* est extrait, récupéré dans *sig* **et** renvoyé comme valeur de retour de la fonction.

Pthreads et les signaux

- **Déconseillé d'utiliser un gestionnaire de signaux**
  - Les fonctions POSIX qui permettent manipuler les *Pthreads* ne sont pas nécessairement réentrantes. Par conséquent elles ne doivent pas être appelées depuis un gestionnaire de signaux.
- **Solution :**
  - Avoir un thread dédié à la réception des signaux qui boucle en utilisant *sig_wait*.
    - Tous les autres threads doivent bloquer les signaux attendus.

Exemple signaux et Pthreads

```
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 signaux et Pthreads (cont)

```
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;
}
```


Sémaphore POSIX

- **Variable du type *sem_t***
- **Création /Déstruction**

  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

  int sem_destroy (sem_t *sem);
- **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.

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

Exemple - Sémaphore POSIX

```
#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

- **Lors du *fork* le processus est dupliqué, mais il n'y aura dans le processus fils que le thread qui a invoqué le fork.**
- **Ce mécanisme doit être restreint à l'utilisation de exec après le fork.**
- **Problème : un autre thread a verrouillé une ressource dont le fils aura besoin.**
  - Solution: Fonction *pthread_atfork* qui permet d'enregistrer des routines qui seront automatiquement invoquées si une thread appelle *fork*.

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

  - *avant*: avant le fork;
  - *dans_pere* et *dans_fils* : par le père et par le fils respectivement après le fork au sein du thread ayant invoqué le *fork*.

Gestion de Threads – Annulation (plus de détails)

- **Deux fonctions permettent de configurer le comportement d'un thread face à une requête d'annulation**

  int pthread_setcancelstate (int etat, int* ancien_etat) ;
  - PTHREAD_CANCEL_ENABLE
    - Le thread acceptera les requêtes d'annulation (par défaut).
  - PTHREAD_CANCEL_DISABLE
    - Le thread ne tiendra pas comptes des requêtes d'annulation

  int pthread_setcanceltype (int type_annul, int* ancien_type);
  - PTHREAD_CANCEL_DEFERRED
    - La thread ne se termine qu'en atteignant un point d'annulation (par défaut).
  - PTHREAD_CANCEL_ASYNCHRONOUS
    - L'annulation prendra effet dès la réception de la requête.

Gestion de Threads – Annulation (plus de détails)

- **Un thread peut être annulé à tout moment**
  - nécessité de libérer les ressources que le thread possède avant qu'il ne se termine.
    - Fichiers ouverts, mutex verrouillé, mémoire allouée, etc.
- **Solution :**
  - Enregistrer des routines de libération dans une "pile de nettoyage":

    void pthread_cleanup_push (void (*fonction) (void *arg), void *arg);
    - Lorsque le 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.
    - Si *exec_routine* est non nul la routine est invoquée.