Introduction à l'informatique temps réel

Sidi Mohammed SENOUCI

2011-2012

Motivations

- Du jour :
  - POSIX
- Comprendre les mécanismes systèmes mis en oeuvre dans un système temps réel

POSIX

- POSIX (Portable Operating System Interface)
  - Standard initialement normalisé en 1988 par l'IEEE sous le nom de P1003 et par l'ISO/IEC sous le nom ISO/IEC-9945.
  - Norme en perpétuelle évolution avec différents groupes de travail
    - Aspects du système d'exploitation Unix
    - Réseau au système de fichiers
    - Temps réel
    - Interfaçage avec différents langages de programmation, etc.
  - Les standards proposés concernent des interfaces et non pas des implémentations

POSIX

- Standards POSIX

Nom IEEE	Nom	Notes
1003.1	Interface système (*System Interface*)	Définit l'interface de programmation système d'un système d'exploitation POSIX – Dernière version 2004. Depuis 2001, intègre différents amendements donnés ci-après.
1003.1b	Extensions temps réel (*Realtime Extensions*)	Dernière version 1993, intégrée dans 1003.1 depuis 2001.
1003.1c	Tâches (*Threads*)	Dernière version 1995, intégrée dans 1003.1 depuis 2001.
1003.1d	Extensions temps réel additionnelles (*Additional Realtime Extensions*)	Dernière version 1999.
1003.1h	Tolérance aux fautes (*Fault Tolerance*)	Dernière version 2000 – Devenu 1003.25.
1003.1i	Corrections aux extensions temps réel (Fixes to 1003.1b)	Dernière version 1995, intégrée dans 1003.1 depuis 2001.
1003.1j	Extensions temps réel avancées (*Advanced Realtime extensions*)	Dernière version 2000.
1003.5	Interface Ada avec 1003.1 (*Ada Binding to 1003.1*)	Dernière version 1997.
1003.5a	Mise à jour Ada (*Ada Update*)	Abandonné en 1996.
1003.5b	Ada temps réel (*Ada Realtime*)	Dernière version 1996.
1003.13	Profils temps réel (*Realtime Application Environment Profile*)	Dernière version 1998.
1003.25	Tolérance aux fautes (*Fault Tolerance*)	Nouveau nom de 1003.1h. En développement depuis 1999.
2003.1	Méthodes de test pour mesurer la conformité à 1003.1 (*Test Methods for 1003.1*)	Dernière version 2000.
2003.1b	Méthodes de test pour mesurer la conformité à 1003.1b (*Test Methods for 1003.1b*)	Dernière version 2001.

Threads ou tâches POSIX

- Défini dans le standard IEEE POSIX 1003.1c et 1003.1j
- Les threads permettent d'écrire simplement des applications parallèles
- Plus légers à manipuler que les processus
  - Un thread est un "sous processus" dans le sens ou un thread fait parti d'un processus.
  - Les threads partagent le même espace mémoire global du processus
  - Chaque thread a sa propre pile et pointeur de pile et ses attributs particuliers (priorité, type d'ordonnancement, signaux…)

Threads ou tâches POSIX

- Caractéristiques
  - Création, gestion et destruction
    - pthread_create, etc.
  - Synchronisation et communication
    - mutex, sémaphore, rendez-vous, etc.

Threads POSIX en langage C

- Un thread est une portion de code (fonction) qui se déroule en parallèle au thread principal (aussi appelé main)
- Les threads permettent d'écrire des applications parallèles
  - Plusieurs fonctions s'exécutent simultanément et partagent les données de l'application -> Nécessité de synchronisation
- La bibliothèque Pthreads est portable (Linux et Windows)
  - Pour Windows : **http://sourceware.org/pthreads-win32/**
- Pour pouvoir compiler un projet
  - *Ajouter l'en-tête :#include <pthread.h>*
  - Pour Code::Blocks :
    - Project>Build Options>Linker Settings, rajouter –lpthread dans l'onglet "Other Linker Options"

Threads POSIX en langage C

- Nouveaux types

pthread_t *var*;	Identifiant d'un thread
pthread_attr_t *var*;	Attribut d'un thread (ordonnancement, priorité, scope, etc)
pthread_mutex_t *var*;	Sémaphore binaire d'exclusion mutuelle
pthread_mutexattr_t *var*;	Attribut d'un mutex. PTHREAD_PROCESS_PRIVATE (par défaut) PTHREAD_PROCESS_SHARED
pthread_cond_t *var*;	Variable de condition
pthread_condattr_t *var*;	Attribut d'une variable de condition PTHREAD_PROCESS_PRIVATE (par défaut) PTHREAD_PROCESS_SHARED

Threads POSIX en langage C

- Nommage des 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
    - **_getattr**: obtenir l'attribut attr des attributs d'un objet
    - **_setattr**: modifier l'attribut attr des attributs d'un objet.
  - Exemples :
    - **pthread_create**: crée un thread (objet pthread_t)
    - **pthread_mutex_init**: initialise un objet du type pthread_mutex_t

Threads POSIX en langage C

- 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 annulée par un autre thread (**pthread_cancel**)
  - Peut attendre la fin d'un autre thread (**pthread_join**)

Création d'un thread POSIX

Retourne une valeur non nulle si succès

int **pthread_create**(**pthread_t** **thr*, **const pthread_attr_t** **attr*, **void** ***(**start_routine*)(**void**), **void** **arg*)

pthread_t *thr	Identifiant du thread qui sera crée. Il est passsé par référence
const pthread_attr_t *attr	Attributs du thread crée. Utiliser NULL pour les attributs par défaut
void *(*start_routine)(void)	Cet argument permet de transmettre un pointeur sur la fonction que thread devra exécuter.
void *arg	Argument que l'on peut passer à la fonction que le thread doit exécuter

Les pointeurs de type **void** peuvent faire référence à **N'IMPORTE QUEL** type de données, mais ils **NE PEUVENT** être utilisés dans aucune opérations qui lit ou écrit ses données sans un **cast**

Création d'un thread POSIX

- Thread principal vs. Threads annexes
  - Un thread créée par la primitive **pthread_create** dans la fonction main est appelé thread **annexe**
  - La création d'un processus donne lieu à la création du thread main (thread principal)
  - Retour de la fonction main entraîne la terminaison du processus et par conséquent de tous les threads de celui-ci

Gestion des threads POSIX

void **pthread_exit**(void **arg*);

Termine l'éxécution du thread. Elle permet de renvoyer des informations au thread père via la l'argument *arg*

pthread_t **pthread_self**(*void*)

Cette fonction retourne l'identifiant du thread appelant.

void **pthread_cancel**(**pthread_t** *th*);

Annule un thread *th* à partir d'un autre. Renvoie 0 si elle réussie. L'activité cible de la demande d'abandon a un comportement dépendant de deux indicateurs (**anullabilité**, **immédiateté**)

Gestion des threads POSIX

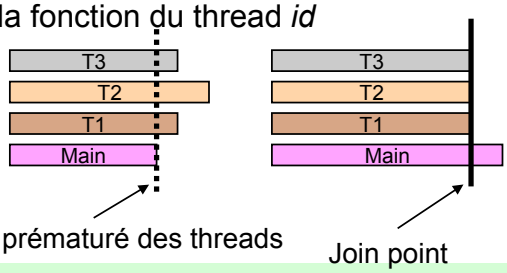
int **pthread_join**(**pthread_t** *id*, **void** ***tr*);

Suspend l'exécution du thread appelant jusqu'à ce que termine le thread *id*. Il retourne l'état de terminaison du thread *id*.
Le thread id doit être joignable (voir ci-après).

pthread_t id — Le thread à attendre

void **tr — La valeur de retour de la fonction du thread *id*

Retourne une valeur nulle si succès



Utilisation? *Lorsque nous créons des threads puis nous laissons continuer par exemple la fonction main, nous prenons le risque de terminer le programme complètement sans avoir pu exécuter les threads.*

Exemple 1

```
#include <stdio.h>
#include <pthread.h>
static void *task_a (){
    printf ("tidA: %d\n", (int)
    pthread_self());
    puts ("Hello world A");
    return NULL;
}
static void *task_b (){
    printf ("tidB: %d\n", (int)
    pthread_self());
    puts ("Hello world B");
    return NULL;
}
```

```
int main (void){
    pthread_t ta, tb;
    puts ("main init");
    pthread_create (&ta, NULL,
    task_a, NULL);
    pthread_create (&tb, NULL,
    task_b, NULL);
    pthread_join (ta, NULL);
    pthread_join (tb, NULL);
    puts ("main end");
    return 0;
}
```


Exemple 2

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
void *affiche(void *voidparam)
{
    int v= (int) voidparam;
    printf("  affiche %d\n", v);
    sleep(4);
    printf("termine %d\n", v);
    pthread_exit(NULL);
    return NULL;
}
```

```
int main(void)
{
    pthread_t tid[10];
    int code, i;
    // creation des threads
    for(i= 0; i < 10; i++)
    {
    code= pthread_create(&tid
    [i], NULL, affiche, (void *) i);
    if(code != 0) {
            printf("erreur creation");
            exit(1);
      }
    }
    // attente des threads
    for(i= 0; i < 10; i++)
    code= pthread_join(tid[i], NULL);
    return 0;
}
```

Exercice – Les matrices

- La manipulation des matrices et des vecteurs est un champ particulièrement propice pour les algorithmes parallèles. La multiplication de deux matrices a et b de m colonnes n lignes peut être parallélisé. Ainsi, chaque élément de la matrice résultat peut être un thread. Dans l'exemple suivant on aura 9 threads.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} =$$

$$\begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} & a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} & a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} & a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33} \end{pmatrix}$$

- Ecrire un programme qui calcule la somme de 2 matrices de taille NxN. La somme de chaque ligne est calculée avec des threads indépendants en parallèle. Les éléments des 2 matrices seront générées aléatoirement avec srand(). Essayer pour N=4.

Attributs d'un thread

pthread_attr_t : structure d'attributs passé au moment de la création du thread

int **pthread_attr_init**(**pthread_attr_t** **attr*);

Initialise la structure d'attributs du thread *attr* et la remplit avec les valeurs par défaut pour tous les attributs

- **Ordonnacement par défaut (Default Schedule)**: SCHED_OTHER
- **Portée de l'ordonnacement (Default Scope):** PTHREAD_SCOPE_SYSTEM
- **Etat par défaut (Default Join State):** PTHREAD_CREATE_JOINABLE

N.B. Chaque attribut *attrname* peut être individuellement modifié en utilisant la fonction **pthread_attr_setattrname** et récupéré à l'aide de la fonction **pthread_attr_getattrname** (voir ci-après)

Attributs d'un thread

- Noms des attributs d'un thread
  - **scope**(int) – thread natif 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, SCHED_FIFO, SCHED_RR
  - **schedparam**(sched_param*) – paramètres pour l'ordonnanceur
  - **inheritsched**(int) – ordonnancement hérité ou pas
    - PTHREAD_INHERIT_SCHED, PTHREAD_EXPLICIT_SCHED
  - **cancelstate**(int) – indicateur d'anullabilité (préemption)
    - PTHREAD_CANCEL_ENABLE, PTHREAD_CANCEL_DISABLE

Attributs d'un thread

- Exemples d'appel de fonctions de gestion des attributs d'un thread :
  - Obtenir la taille de pile du thread
    - pthread_attr_t attr;
    - size_t taille;
    - pthread_attr_getstacksize(&attr, &taille);
  - Détachement d'un thread
    - pthread_attr_t attr;
    - pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
  - Modifier la politique d'ordonnancement
    - pthread_attr_t attr;
    - pthread_attr_setschedpolicy(&attr, SCHED_FIFO);

Gestion des threads POSIX

int **pthread_attr_destroy**(**pthread_attr_t** *attr);

Détruit l'objet *attr* qui ne doit plus jamais être réutilisé jusqu'à ce qu'il soit réinitialisé. Il est conseillé de détruire l'objet attr après que le thread soit crée

Gestion des threads POSIX

int **pthread_setcancelstate** (int state, int * old_state);

Retourne une valeur nulle si succès

Positionne l'indicateur d'anullabilité (**préemption**) :
- PTHREAD_CANCEL_ENABLE (par défaut): Autorise les annulations pour le thread appelant.
- PTHREAD_CANCEL_DISABLE : Désactive les requêtes d'annulation.

old_state: Adresse vers l'état précédent (ou NULL) permettant ainsi sa restauration lors d'un prochain appel de la fonction.

int **pthread_setcanceltype** (int state, int * old_state);

Retourne une valeur nulle si succès

Positionne l'indicateur d'immédiateté (Valeur par défaut "Non activé"):
- PTHREAD_CANCEL_DEFERRED (par défaut) : Différé au premier point de contrôle rencontré
- PTHREAD_CANCEL_ASYNCHRONOUS : Effet immédiat

Gestion des threads POSIX

int **pthread_attr_setdetachstate**(**pthread_attr_t** **attr*, **int** *JOIN_STATE*);

Sert à établir l'état de terminaison d'un thread. *JOIN_STATE* peut prendre la valeur:

- **PTHREAD_CREATE_JOINABLE** (par défaut): Les threads créés avec attr seront dans un état joignables. Ils ne libéreront pas leurs ressources. Il sera donc nécessaire d'appeler phtread_join().
- **PTHREAD_CREATE_DETACHED**: Les threads créés avec attr seront dans un état détaché. Ils libéreront leurs ressources quand ils termineront.

Retourne une valeur nulle si succès

int **pthread_detach** (**pthread_t** *thr*, **void** ***value_ptr*)

Intervention sur un thread en cours.
Il place le thread *th* dans l'état détaché. Cela garantit que les ressources mémoire consommées par th seront immédiatement libérées lorsque l'exécution de th s'achèvera. Cependant, cela empêche les autres threads de se synchroniser sur la mort de th en utilisant pthread_join.

Exemple:

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

Gestion des threads POSIX

int **pthread_attr_setschedpolicy**(**pthread_attr_t** **attr*, **int** *policy*)

Sert à définir la politique d'ordonnancement du thread créé en utilisant l'objet d'attributs de thread attr.
Policy peut prendre la valeur :

- SCHED_OTHER (par défaut) → Thread normal (pas de priorité mais équitable)
- SCHED_RR → Round-robin
- SCHED_FIFO → First-in First-out

Retourne une valeur nulle si succès

int **pthread_attr_setschedparam**(**pthread_attr_t** **attr*, **const struct sched_param** **pr*)

Définit la priorité du thread créé en utilisant l'objet d'attributs de thread attr.
Default: 0

```
struct sched_param {
   int sched_priority;     /* Priorité d'ordonnancement */
};
```

La politique d'ordonnancement d'un thread peut être modifiée dynamiquement après sa création avec **pthread_setschedparam**

Gestion des threads POSIX

- SCHED_FIFO
  - Mode à tendance temps-réel
  - La tâche prête qui a la plus forte priorité au niveau global et qui est arrivée la première
  - Pas de préemption
  - Nécessite des privilèges (root)
- SCHED_RR
  - Mode à tendance temps-réel
  - Fonctionne comme SCHED_FIFO avec un quantum de temps appliqué par tourniquet (préemptif)
  - Nécessite des privilèges (root)
- SCHED_OTHER (par défaut)
  - Optimisé pour un temps de réponse et un rendement globale
  - Défini par l'implémentation (pas un processus au détriment des autres)
  - Priorité statique = 0 (le moins prioritaire)
  - Procédé spécifique à chaque système (quelquefois ≡ SCHED_RR avec la priorité statique 0)

Gestion des threads POSIX

int **pthread_setschedparam(pthread_t** *thr*, **int** *policy*, **const struct sched_param** ***p*);**

Retourne une valeur nulle si succès

Modifie **dynamiquement** la politique et les paramètres d'ordonnancement pour le thread thr (**après sa création**)
Policy peut prendre la valeur (SCHED_OTHER, SCHED_RR, SCHED_FIFO)
P change les paramètres d'ordonnancement pour les deux politiques temps réél

Exemple

```
void *thread_func (void *arg) {
    int ret; int policy; struct sched_param sched; char P[15];
    ret = pthread_getschedparam (pthread_self(), &policy, &sched);
    if (ret)
        exit (1);
    else
        {
            if (policy == SCHED_OTHER) strcpy(P, "SCHED_OTHER”);
            if (policy == SCHED_FIFO) strcpy(P, "SCHED_FIFO”);
            if (policy == SCHED_RR) strcpy(P, "inconnu”);
            printf ("politique : %s, priorité :%d\n", P, sched.sched_priority);
        }
    return NULL;
}
```


Gestion des threads POSIX

***int* pthread_attr_setinheritsched(pthread_attr_t **attr*, int *inherit*)**

Indique si la politique et les paramètres d'ordonnancement pour le nouveau thread sont déterminés par les valeurs des attributs schedpolicy et schedparam (valeur PTHREAD_EXPLICIT_SCHED) ou sont héritées du thread parent (valeur PTHREAD_INHERIT_SCHED).
Inherit prend la valeur :

- **PTHREAD_EXPLICIT_SCHED** (par défaut)
- **PTHREAD_INHERIT_SCHED**

***int* pthread_attr_setscope(pthread_attr_t **attr*, int *scope*)**

Définit les paramètres de contention et de portée

- **PTHREAD_SCOPE_SYSTEM** (par défaut) : tous les threads sont en compétition avec tous les processus en cours d'exécution pour le temps processeur
- **PTHREAD_SCOPE_PROCESS** : les threads ne sont en compétition qu'avec les autres threads du même processus

La synchronisation

pthreads.h — Terminaison

En utilisant la fonction **pthread_join**

pthreads.h — Mutex

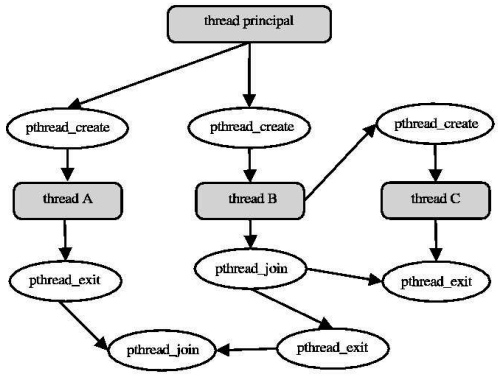
Sémpahore binaire d'exclusion mutuelle. Si un thread possède le verrou, seulement celui-ci peut lire et écrire sur les variables étant dans la portion de code protégée (aussi appelée zone critique). Lorsque le thread a terminé, il libère le verrou et un autre thread peut le prendre à son tour.

semaphore.h — Sémaphores

Sémaphores à compte

Exercice (TD)

- Synchronisation par Terminaison
  - Ecrire un programme dont le thread principal (main) crée deux threads A et B et attend la fin de leurs exécutions. Le thread B crée lui-même un thread C qu'il attendra avant de terminer. Choisir la fonction associée à chaque thread (une fonction d'affichage par exemple).



La synchronisation Mutex

- Sémaphore binaire ayant deux états : libre et verrouillé
  - Seulement un thread peut obtenir le verrouillage. Toute demande de verrouillage d'un mutex déjà verrouillé entraînera soit le blocage du thread, soit l'échec de la demande.
  - Variable de **type pthread_mutex_t**.
  - Possède des attributs de type **pthread_mutexattr_t**
  - Utilisé pour:
    - Protéger l'accès aux variables globales
    - Gérer des synchronisations de threads

La synchronisation Mutex

- Création/Initialisation (2 façons) :

1. Statique:
   - **pthread_mutex_t** m = PTHREAD_MUTEX_INITIALIZER;
2. 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);

La synchronisation Mutex

1 *int* **pthread_mutex_init**(**pthread_mutex_t** **m*, **const pthread_mutexattr_t** **ma*);

2 *int* **pthread_mutex_lock**(**pthread_mutex_t** *m);

3 *int* **pthread_mutex_unlock**(**pthread_mutex_t** **m*);

4 *int* **pthread_mutex_destroy**(**pthread_mutex_t** **m*);

1→ Initialiaation 2 → Verouillage (opération **P**)

3 → Déverouillage (opération **V**) 4 → Déstruction

pthread_mutexattr_t * ma = NULL pour les valeurs par défaut

La synchronisation Mutex

int **pthread_mutex_lock**(**pthread_mutex_t** *mutex);

Retourne 0 si succès

Verrouille le mutex. Si le mutex est déverrouillé, il devient verrouillé et est possédé par le thread appelant; et **pthread_mutex_lock** rend la main immédiatement. Si le mutex est déjà verrouillé par un autre thread, **pthread_mutex_lock** suspend le thread appellant jusqu'à ce que le mutex soit déverrouillé.

int **pthread_mutex_trylock**(**pthread_mutex_t** *mutex);

Retourne 0 si succès sinon EBUSY

Idem que **pthread_mutex_lock**, excepté qu'elle ne bloque pas le thread appelant si le mutex est déjà verrouillé par un autre thread. Au contraire, pthread_mutex_trylock rend la main immédiatement avec le code d'erreur EBUSY.

La synchronisation Mutex

int **pthread_mutex_unlock**(**pthread_mutex_t** *mutex);

Déverrouille le mutex.

Retourne 0 si succès

int **pthread_mutex_destroy**(**pthread_mutex_t** *mutex);

Détruit un mutex, libérant les ressources qu'il détient. Le mutex doit être déverrouillé. Dans quelques implémentations, aucune ressource ne peut être associé à un mutex, aussi **pthread_mutex_destroy** ne fait en fait rien si ce n'est vérifier que le mutex n'est pas verrouillé.

Retourne 0 si succès sinon erreur (EBUSY)

Exemple simple

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
pthread_mutex_t mutex =
    PTHREAD_MUTEX_INITIALIZ
    ER;
int cont =0;
void *sum_thread () {
pthread_mutex_lock (&mutex);
 cont++;
pthread_mutex_unlock
    (&mutex);
}
```

```
int main () {
pthread_t tid;
if (pthread_create (&tid, NULL,
    sum_thread, NULL) != 0) {
    printf("pthread_create
    error");       exit (1);
}
pthread_mutex_lock (&mutex);
cont++;
pthread_mutex_unlock
    (&mutex);
pthread_join (tid, NULL);
 printf ("cont : %d\n", cont);
return 0;
}
```

Synchronisation : sémaphores

- Un sémaphore Posix est un **sémaphore à compte**
  - Le compteur associé au sémaphore peut donc prendre des valeurs plus grande que 1 (contrairement à un mutex) pour bloquer ou non l'accès à plus d'une ressource.
- Il existe trois primitives de base pour gérer un sémaphore :
  - Initialisation du compteur au nombre de ressources disponibles
  - Prise d'une ressource -**P(S)**- qui décremente le compteur (si le compteur est négatif, le thread est bloqué)
  - Libération d'une ressource -**V(S)**- qui incrémente le compteur (si le compteur est nul ou négatif, un thread en attente est réactivé)
- Pour utiliser les sémaphores : **#include <semaphore.h>**

Synchronisation : sémaphores

int **sem_init** (**sem_t** *sem, int partage, unsigned int valeur);

La fonction sem_init initialise le sémaphore à compte pointé par **sem** à la **valeur** spécifiée en paramètre (positive ou nulle).
Le paramètre **partage** a le rôle suivant :

- s'il est non nul, le sémaphore peut être utilisé pour synchroniser des threads de processus différents ;
- s'il est nul, l'utilisation du sémaphore est limitée aux threads du processus appelant.

int **sem_wait** (**sem_t** *sem);

Pour effectuer la primitive P(s)

int **sem_trywait** (**sem_t** *sem);

Si on désire effectuer l'opération P(s) mais qu'elle ne soit pas bloquante. Si le compteur est nul, la fonction retourne avec le code d'erreur EAGAIN. Sinon elle décrémente le compteur et retourne 0.

Synchronisation : sémaphores

int **sem_post** (**sem_t** *sem);

Pour libérer la sémaphore V(s).

int **sem_getvalue (sem_t** *sem, int *valeur);

Permet de connaître la valeur (positive ou nulle) du compteur du sémaphore

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

Détruit le sémaphore sem (supposé avoir été initialisé par **sem_init**)

Exercice (TD)

- Résoudre le problème du « producteur-consommateur » en utilisant les sémaphores

La synchronisation : Les moniteurs

- L'implémentation des moniteurs sous POSIX se fait avec des mutex et des variables conditionnelles. Impossible de le définir autrement sous forme d'un objet surtout avec un langage non POO comme le C.
- Le mutex est utilisé pour assurer la protection des opérations sur la variable condition

La synchronisation : Les conditions

- Mécanisme de syncrhonisation permettant à un thread de suspendre son exécution juqu'à ce qu'une certaine condition (un prédicat) soit vérifiée.
- Les opérations fondamentales sur les conditions sont:
  - **Signaler** la condition (quand le prédicat devient vrai)
  - **Attendre** la condition jusqu'à ce qu'un autre thread signale la condition

La synchronisation : Les conditions

- 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);

La synchronisation : Les conditions

int **pthread_cond_init**(**pthread_cond_t** **cond*, **pthread_condattr_t** **attr*)

Initialise la variable condition *cond*, en utilisant les attributs de condition spécifiés par *cond_attr*, ou les attributs par défaut si *cond_attr* vaut NULL.
Les variables de type pthread_cond_t peuvent également être statiquement initialisées, en utilisant la constante PTHREAD_COND_INITIALIZER

int **pthread_cond_destroy**(**pthread_cond_t** **cond*)

Détruit une variable cond, libérant les ressources qu'elle possède. Aucun thread ne doit attendre sur la condition à l'entrée de **pthread_cond_destroy**. Dans certaines implémentations, aucune ressource ne peut être associée à une variable condition, aussi **pthread_cond_destroy** ne fait en fait rien d'autre que vérifier qu'aucun thread n'attend la condition.

La synchronisation : Les conditions

int **pthread_cond_wait**(**pthread_cond_t** **condition*, **pthread_mutex_t** **m*)

Si l'activité appelante a réalisé au préalable une opération P sur le mutex désigné, l'appel a pour effet de libérer le mutex, d'attendre un signal sur la condition désignée et de reprendre son exécution, en verrouillant à nouveau le mutex, après que la condition aura été signalée.

Utilisation:

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

Mutex Véroullé

Mutex Libéré -> pas d'interblocage

Mutex Véroullé

La synchronisation : Les conditions

int **pthread_cond_signal**(**pthread_cond_t** **cond*)

Relance l'un des threads attendant la variable condition *cond*. S'il n'existe aucun thread répondant à ce critière, rien ne se produit. Si plusieurs threads attendent sur *cond*, seul l'un d'entre eux sera relancé, mais il est impossible de savoir lequel.

int **pthread_cond_broadcast** (**pthread_cond** **cond*)

Relance tous les threads attendant sur la variable condition *cond*. Rien ne se passe s'il n'y a aucun thread attendant sur cond.

La synchronisation : Les conditions

Toutes les variables de condition ont besoin d'être associé à un mutex spécifique qui va bloquer le thread jusqu'à l'émission de la notification.

Thread attendant la condition	Thread signalant la condition
`pthread_mutex_lock`	
`pthread_cond_wait` déblocage du mutex et se mettre en file	
wait	
	`pthread_mutex_lock`
	`pthread_cond_signal` Réveil du thread
`pthread_cond_wait` Tentative de récupérer le mutex	
	`pthread_mutex_unlock`
Fin de `pthread_cond_wait`	
`pthread_mutex_unlock`	

La synchronisation : Les conditions

int **pthread_cond_timedwait**(**pthread_cond_t** **condition*, **pthread_mutex_t** **m,*
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 *abstime.* Ce temps spécifiée en temps absolu.
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.

La synchronisation : Les conditions

int **pthread_condattr_init** (**pthread_condattr_t** **attr*)

Initialise l'attribut de condition *attr* et le remplit avec les valeurs par défaut pour les attributs.

int **pthread_condattr_destroy** (**pthread_condattr_t** **attr*)

Détruit l'attribut de condition attr *attr*. Il est important de lé détruire après l'initialisation de la condition.

Exemple 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 () {
    printf ("tid: %d\n", (int)
    pthread_self());
    pthread_mutex_lock (&mutex_fin);
    pthread_cond_signal (&cond_fin);
    pthread_mutex_unlock
    (&mutex_fin);
    pthread_exit (NULL);
}
```

```
int main (){
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 0;
}
```

Exercice (TD)

- Soient deux threads exécutant chacun une fonction différente :
  - functionCount2(): incrémente un compteur count (initialisé à 0) seulement si count ∈ [4, 7]
  - functionCount1(): incrémente count seulement si count ∈ [1, 3] ∪ [8, 10]
- Ecrire un programme de test qui permet de synchroniser ces deux fonctions en utilisant les conditions

Autres fonctions utiles

void **pthread_yield** ()

Permet à un thread d'abandonner son temps d'éxécution au profit d'un autre thread (choisit par l'ordonnanceur).
Très utile lorsqu'un thread boucle sur une activité et bloque constamment un verrou pour réaliser sa tâche.

Exemple

```
int tab[10];
pthread_mutex_t verrou;
void *thread1(void *voidparam)
    {
       for(;;)
       {
          pthread_mutex_lock(&verrou);
          for(int i= 0; i< 10; i++)
             tab[i]= 0;
          pthread_mutex_unlock(&verrou);
          /* sans le yield, le thread va reprendre tout de suite le verrou sans
    laisser le temps aux autres threads de travailler sur le tableau   */
          pthread_yield();
       }
       pthread_exit(NULL);
       return NULL;
    }
```