GL4-RT5

Systèmes Temps-Réel

Chapitre 4 :

POSIX pour les Systèmes Temps-Réel

Olfa Mosbahi

olfamosbahi@gmail.com

POSIX

- **POSIX:** famille de normes techniques définie depuis 1988 par l'Institute of Electrical and Electronics Engineers (IEEE)
  - formellement désignée par IEEE 1003
  - Ces normes ont émergé d'un projet de standardisation des interfaces de programmation des logiciels destinés à fonctionner sur les variantes du système d'exploitation UNIX.

- **Le terme *POSIX*** a été suggéré par Richard Stallman, qui faisait partie du comité qui écrivit la première version de la norme. L'IEEE choisit de le retenir car il était facilement mémorisable.

- **POSIX** forment l'acronyme de ***P****ortable* ***O****perating* ***S****ystem* ***I****nterface* (interface portable de système d'exploitation), et le **X** exprime l'héritage UNI**X**.

Versions de POSIX

- **POSIX.1**, Services centraux (inclut le standard ANSI C) (IEEE Std 1003.1-1988) porte sur :
  - la création et le contrôle des processus
  - les gestions des signaux inter-processus
  - les exceptions des nombres flottants
  - les violations de segmentation
  - les instructions illégales
  - les erreurs de bus
  - les timers
  - les opérations sur les fichiers et les dossiers
  - les tubes
  - la bibliothèque standard de C
  - les entrées-sorties et le contrôle des ports
- **POSIX.1b**, extension pour le temps réel (IEEE Std 1003.1b-1993) :
  - l'ordonnancement
  - les signaux en temps réel
  - les horloges et les timers
  - les sémaphores
  - le passage de messages
  - la mémoire partagée
  - les entrées-sorties synchrones et les entrées-sorties asynchrones **(en)**
  - les outils de verrouillage de la mémoire
- **POSIX.1c**, extension sur les processus légers (les threads) (IEEE Std 1003.1c-1995) :
  - la création, le contrôle et la suppression des threads
  - l'ordonnancement des threads
  - la synchronisation des threads
  - l'interception des signaux (*Signal Handling*)

Introduction

- **Langage C** (par exemple la version standardisée en 1999) ne dispose pas d'instructions natives permettant d'écrire directement des **programmes multitâches**
- Bibliothèque ***pthread*** définie dans les normes *POSIX 1003.1* lui est associée pour atteindre cet objectif
- Fin 2011, une nouvelle version **instable** du langage C (**langage C11**) a été standardisée, lui permettant désormais de disposer des bibliothèques natives pour la programmation multitâche
- Dans ce cours,
  - nous ne nous intéressons pas à cette dernière version instable et **considérons le langage C dans sa version antérieure**
  - **Ce cours propose un double objectif :** la présentation d'une panoplie de fonctions des normes *POSIX 1003.1* et leurs mises en application dans le contexte du traitement des différentes problématiques liées au multitâche: création de tâche, périodicité de tâche, affectation d'une priorité et d'un type d'ordonnancement à une tâche, synchronisation entre tâches, communication entre tâches, etc.

Tâche sous POSIX

- Une ***tâche*** est une unité active d'une application.
  - **Un ensemble d'instructions séquentielles** correspondant souvent à une procédure ou à une fonction
  - l'équivalent d'un ***processus léger*** dans le jargon des systèmes d'exploitation
    - Contrairement à un ***processus lourd*** qui, lorsqu'il est créé, implique la réservation des ressources telles qu'un espace mémoire, une table de fichiers ouverts et une pile interne qui lui sont toutes dédiées, un ***processus léger***, lorsqu'il est créé, partage le même espace mémoire et la même table des fichiers ouverts que son processus père (son créateur), mais dispose de sa propre pile
    - L'avantage des processus légers sur les processus lourd est qu'ils sont très favorables à la programmation multitâche, car la création d'une tâche est moins coûteuse en termes de ressources du fait qu'elles ne sont pas toutes dupliquées. De plus, l'utilisation des tâches simplifie la gestion des problématiques liées à l'exécution concurrente (communication, synchronisation entre tâches...).
- Sous Posix, l'anglicisme utilisé pour dénommer un *processus léger* et donc une *tâche* est ***thread***

Création de Tâches sous POSIX

- Un programme C utilisant la bibliothèque ***pthread*** doit disposer de l'entête suivante : **# include <pthread.h>**
- Sous Posix, la création d'une tâche est réalisée avec la fonction suivante :

```
int    pthread_create  (pthread_t * thread,  pthread_attr_t * attr,
                        void* (*start_routine)(void*),   void* arg);
```

- ✓ ***thread*** **:** pointeur de type ***pthread_t*** contenant l'identificateur de la tâche qui vient d'être créée
- ✓ ***attr*** **:** variable de type ***pthread_attr_t***. Elle correspond à une sorte de conteneur qui va permettre d'indiquer les différentes ***propriétés*** de la tâche qui doit être exécutée (**son type d'ordonnancement, sa priorité, tâche joignable/détachable? ...)**
- ✓ ***start_routine*** **:** c'est la fonction C qui sera exécutée par la tâche qui est créée ;
- ✓ ***arg*** **:** pointeur qui correspond aux variables passées en paramètre à la fonction ***start_routine***. Il vaut **NULL** si aucun paramètre n'est passé à la fonction ;

Comme valeur de retour, la fonction *pthread_create* renvoie 0 si tout s'est bien passé et un nombre négatif sinon.

Création de Tâches sous POSIX

- Lorsqu'un processus crée une tâche sous Posix, s'il ne lui est pas explicitement indiqué d'attendre la fin d'exécution de cette tâche, alors à sa terminaison elle forcera l'arrêt de la tâche créée. Pour éviter cela, Posix propose la fonction ***pthread_join***. Le prototype de la fonction ***pthread_join*** est :

int pthread_join (pthread_t* thread, void thread_return)**

- ✓ Son premier paramètre ***thread*** représente l'identificateur de la tâche sur laquelle le processus ou la tâche créatrice doit attendre.

- ✓ Son deuxième paramètre **thread_return** représente une variable dans laquelle est stockée une éventuelle valeur de retour produite par la tâche. Ce dernier vaut **NULL** si la tâche ne renvoie aucune valeur.

Exemple de 2 Tâches sous POSIX

Considérons un programme où:

- Deux tâches doivent être créées par le processus exécutant la fonction ***main(void)*** et s'exécuter de manière concurrente.
- Chacune, lorsqu'elle devient active, doit s'identifier et effectuer un nombre quelconque d'itérations en l'écrivant à l'écran avant d'être préemptée.

Résultat d'exécution

```
#$ gcc -lpthread -o executionConcurrente executionConcurrente.c
#$ ./executionConcurrente
Tache 1 : 0 // Tache 2 : 0 // Tache 1 : 1 //
Tache 2 : 1 // Tache 2 : 2 // Tache 1 : 2 //
Tache 2 : 3 // Tache 1 : 3 // Tache 2 : 4 //
Tache 1 : 4 // Tache 2 : 5 // Tache 1 : 5 //
Tache 2 : 6 // Tache 1 : 6
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>


void* fonc(void* arg){
 int i;
 for(i=0;i<7;i++){
    printf("Tache %d : %d\n", (int) arg, i);
    usleep(1000000); //attendre 1 seconde
 } }
int main(void)
{
    pthread_t tache1, tache2;
//déclaration des deux tâches
    pthread_create(&tache1, NULL, fonc, (void*) 1);
//création effective de la tâche tache1
    pthread_create(&tache2, NULL, fonc, (void*) 2);
    pthread_join(tache1, NULL);
//la fonction principale main(void), doit attendre la fin de l'exécution de la tâche tache1
    pthread_join(tache2, NULL);
    return 0; }
```

Forcer la non-attente d'une Tâche

- Il peut arriver que dans une application multitâche, on n'ait pas besoin que le processus créateur d'une tâche attende la fin d'exécution de cette dernière avant de se terminer, Pour cela, on force la tâche à être détachée de son processus père. Ainsi, même si la fonction ***pthread_join(..)*** est appelée dans le code source du processus père, cela sera sans effet.

- Pour forcer une tâche à être détachée, il faut modifier la valeur de la variable de type ***pthread_attr_t*** de la tâche. Le bout de code suivant permet de réaliser cet objectif:

```
pthread_attr_t   attr;                  // déclaration de la variable contenant les propriétés de la tâche
pthread_attr_init(&attr);
//initialisation de attr aux valeurs par défaut. Obligatoire avant toute manipulation de attr
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
// affectation de la propriété détachable à attr
pthread_create(&tache1, &attr, fonc, 1);
//...
pthread_attr_destroy(&attr);            // détruire attr pour libérer la mémoire allouée
```

Priorité et ordonnancement des Tâches

- Dans une application multitâche, les tâches doivent s'exécuter selon leur **importance** et un **ordre** bien spécifique doit être choisi : c'est la problématique d'affectation des priorités aux tâches et du type de leur ordonnancement.
- Pour affecter une priorité et un type d'ordonnancement à une tâche, il faut modifier sa propriété ***attr*** de type ***pthread_attr_t***. Pour cela, nous aurons besoin des éléments suivants :

1. La déclaration de la structure de données Posix utilisée pour affecter une priorité à la tâche à créer : **struct sched_param param**
2. La fonction qui oblige le système d'exploitation à prendre en compte les différents paramètres (priorité et type d'ordonnancement) que va acquérir la tâche à créer :

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

Les différentes valeurs que peut prendre la variable ***inheritsched*** sont :

- ***PTHREAD_EXPLICIT_SCHED*** : la tâche à créer sera forcée d'utiliser les paramètres d'ordonnancement contenus dans sa propriété ***attr*** ;
- ***PTHREAD_INHERIT_SCHED*** : la tâche à créer héritera des propriétés d'ordonnancement de son processus créateur. Quels que soient les paramètres d'ordonnancement spécifiés dans ***attr***, elle les ignorera si cette option est utilisée.

Priorité et ordonnancement des Tâches

3. La fonction qui permet d'affecter à la propriété ***attr*** de la tâche à créer, sa priorité (qui est un nombre entier) contenue dans la variable ***param*** :

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

4. La fonction qui permet d'affecter à la propriété ***attr*** de la tâche à créer, son type d'ordonnancement noté ***policy*** :

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

Il existe plusieurs type d'ordonnancement correspondant sous Posix aux valeurs suivantes :

- ***SCHED_FIFO*** : il s'agit de l'ordonnancement préemptif à priorités fixes. Les tâches de même priorité sont ordonnancées en FIFO (c'est-à-dire, selon l'ordre de leurs activations).
- ***SCHED_RR*** : il s'agit de l'ordonnancement Round-Robin (c'est-à-dire, à tourniquet) à priorité préemptif. Une tâche utilise un quantum de temps puis est déplacée en queue de la file d'attente du niveau de sa priorité.
- ***SCHED_OTHER*** : il s'agit d'un ordonnancement à temps partagé entre tâches. Il pointe généralement sur SCHED_FIFO.

5. La fonction permettant d'affecter les paramètres d'ordonnancement à une tâche en cours d'exécution. Elle est utilisée pour affecter les paramètres d'ordonnancement du processus père qui crée toutes les autres tâches, ou directement dans le corps de la tâche :

int pthread_setschedparam(**pthread_t thread,int policy,const struct** sched_param ***param**)

Priorité et ordonnancement de Tâches

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void* fonc(void* arg){
 int i;
 for(i=0;i<7;i++){
    printf("Tache %d : %d /**/", (int) arg, i);
    usleep(1000000); }}
int main(void)
{
    pthread_t           tache1, tache2;
    pthread_attr_t      attr;
    struct sched_param    param;
    pthread_attr_init(&attr);
    param.sched_priority = 12;
    pthread_setschedparam  (pthread_self(),
                            SCHED_FIFO, &param);
//pthread_self() pointe sur le processus en cours
d'exécution, à l'occurrence la fonction main()
//le processus main() sera ordonnancé en
SHED_FIFO avec une priorité de 12
    pthread_attr_setinheritsched(&attr,
                        PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
    param.sched_priority = 10;
    pthread_attr_setschedparam(&attr, &param);
    pthread_create(&tache1, &attr, fonc, 1);
   // la tâche tache1 créée, sera ordonnancée en
   SHED_FIFO avec une priorité de 10
    param.sched_priority = 7;
    pthread_attr_setschedparam(&attr, &param);
    pthread_create(&tache2, &attr, fonc, 2);
   // la tâche tache2 créée, sera ordonnancée en
    SHED_FIFO avec une priorité de 7
    pthread_attr_destroy(&attr);
    pthread_join(tache1, NULL);
    pthread_join(tache2, NULL);
    return 0;
}
```

La sortie du programme précédent est la suivante :

```
#$ gcc -lpthread -o monprog monprog.c
#$ sudo ./monprog

Tache 1 : 0 /**/ Tache 2 : 0 /**/ Tache 2 : 1 /**/
Tache 1 : 1 /**/ Tache 1 : 2 /**/  Tache 2 : 2 /**/
Tache 1 : 3 /**/ Tache 2 : 3 /**/ Tache 1 : 4 /**/
Tache 2 : 4
Tache 1 : 5 /**/ Tache 2 : 5 /**/ Tache 1 : 6 /**/
Tache 2 : 600
```

Exclusion mutuelle en POSIX

1. La déclaration de votre donnée/ressource partagée
  – Peut être de tout type dans le langage C

2. La déclaration du ***mutex***. C'est le verrou qui va gérer l'accès à la donnée partagée. Elle fera en sorte qu'une seule tâche accède à la donnée à la fois :

pthread_mutex_t verrou

3. La fonction permettant **d'initialiser** le verrou. Obligatoire avant toute utilisation de ce verrou.

pthread_mutex_init (pthread_mutex_t ***verrou**, **const** pthread_mutextattr_t ***m_attr**)

4. La fonction permettant à une tâche de **prendre le verrou** :

pthread_mutex_lock (pthread_mutex_t ***verrou**)

5. La fonction permettant à une tâche de **libérer le verrou** après avoir utilisé la donnée partagée :

pthread_mutex_unlock (pthread_mutex_t ***verrou**)

Exemple d'Exclusion Mutuelle

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
typedef struct {
    float taille;
    float poids;
} type_donneePartagee;
pthread_mutex_t  verrou;
type_donneePartagee   donneePartagee;

void* tache1(void *arg){
type_donneePartagee ma_donneePartagee;
int i=0;
 while(i<10){
    pthread_mutex_lock(&verrou);
    ma_donneePartagee = donneePartagee;
    pthread_mutex_unlock(&verrou);
    printf("La tache %s vient de lire la donnée
          partagée\n", (char*) arg);
    usleep(1000000);
    i++; }}
```

```c
void* tache2(void *arg){
 int i=0;
 while(i<10){
    pthread_mutex_lock(&verrou);
    donneePartagee.taille = 100 + rand()%101;
    donneePartagee.poids = 10 + rand()%101;
    pthread_mutex_unlock(&verrou);
    printf("La tache %s vient de modifier la
          donnée partagée", (char*) arg);
    usleep(1000000);
    i++; } }
int main(void)
{
    pthread_t    th1, th2;
    pthread_mutex_init (&verrou, NULL);
    donneePartagee.taille = 100 + rand()%101;
    donneePartagee.poids = 10 + rand()%101;
    pthread_create (&th1, NULL, tache1, "1");
    pthread_create (&th2, NULL, tache2, "2");
    pthread_join (th1, NULL);
    pthread_join (th2, NULL);
    return 0; }
```

Exemple d'Exclusion Mutuelle

```
#$ gcc -lpthread -o partageDonnee
partageDonnee.c
#$ ./partageDonnee
La tache 1 vient de lire la donnee partagee
La tache 2 vient de modifier la donnee partagee
La tache 1 vient de lire la donnee partagee
La tache 2 vient de modifier la donnee partagee
La tache 2 vient de modifier la donnee partagee
La tache 1 vient de lire la donnee partagee
La tache 1 vient de lire la donnee partagee
La tache 2 vient de modifier la donnee partagee
La tache 2 vient de modifier la donnee partagee
La tache 1 vient de lire la donnee partagee
La tache 1 vient de lire la donnee partagee
La tache 2 vient de modifier la donnee partagee
La tache 2 vient de modifier la donnee partagee
La tache 1 vient de lire la donnee partagee
La tache 1 vient de lire la donnee partagee
La tache 2 vient de modifier la donnee partagee
La tache 1 vient de lire la donnee partagee
La tache 2 vient de modifier la donnee partagee
La tache 1 vient de lire la donnee partagee
La tache 2 vient de modifier la donnee partagee
```

Exclusion mutuelle & variable condition

- **Une condition est ajoutée sur une donnée partagée** par plusieurs tâches. Ainsi, suivant les besoins, une tâche accédant à la donnée peut être **endormie** si la condition n'est pas vérifiée. Elle ne sera **réveillée** que lorsqu'une autre tâche accédera à cette donnée et rendra la condition vraie.
- Pour notre exemple précédent, on peut considérer que la tâche1 ne lit la donnée partagée que si la *taille* et le *poids* écrits par la tâche 2 sont respectivement supérieurs à 120cm et 60kg. Pour ce faire, une *variable* ***condition*** doit être associée au sémaphore d'exclusion mutuelle pour répondre au problème. Les principaux éléments à connaître sur les variables conditions sont les suivants :

  1. La déclaration et l'initialisation de la variable **condition** :

  ```
  pthread_cond_t   cond = PTHREAD_COND_INITIALIZER;
  ```

  2. La fonction permettant d'endormir une tâche (possédant le *verrou* sur la donnée partagée) si la condition ***cond*** est fausse :

  ```
  int pthread_cond_wait (pthread_cond_t  *cond,  pthread_mutex_t  *verrou)
  ```

  3. La fonction permettant de rendre la condition *cond* vraie. Cela envoie un signal de réveil aux tâches qui ont été endormies sur cette condition :

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

- Si plusieurs tâches attendent sur une condition, l'utilisation de ***pthread_cond_signal ()*** ne réveille que l'une d'entre elles. Les autres restent malheureusement endormies. Pour réveiller toutes les tâches, on utilise la fonction suivante :

  ```
  int pthread_cond_broadcast  (pthread_cond_t  *cond)
  ```

Exclusion mutuelle & variable condition

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

typedef struct {
    float taille;
    float poids;
} type_donneePartagee;
thread_mutex_t   verrou;
pthread_cond_t   cond =
PTHREAD_COND_INITIALIZER;
type_donneePartagee  donneePartagee;
void* tache1(void *arg){ /
 type_donneePartagee ma_donneePartagee;
 int i=0;
 while(i<10){
    pthread_mutex_lock(&verrou);
    pthread_cond_wait(&cond, &verrou);
    ma_donneePartagee = donneePartagee;
    pthread_mutex_unlock(&verrou);
    printf("La tache %s vient de lire la donnee
     partagee\n", (char*) arg); usleep(1000000);
    i++; } }
```

```
void* tache2(void *arg){
int i=0;
 while(i<10){
    pthread_mutex_lock(&verrou);
    donneePartagee.taille = 100 + rand()%101;
    donneePartagee.poids = 10 + rand()%101;
    if(donneePartagee.taille >= 120 &&
          donneePartagee.poids >= 60){
       pthread_cond_signal(&cond);
    }
    pthread_mutex_unlock(&verrou);
    printf("La tache %s vient de modifier la donnee
             partagee\n", (char*) arg);
    usleep(1000000); i++; } }
int main(void)
{   pthread_t th1, th2;
    pthread_mutex_init(&verrou, NULL);
    donneePartagee.taille = 100 + rand()%101;
    donneePartagee.poids = 10 + rand()%101;
    pthread_create(&th1, NULL, tache1, "1");
    pthread_create(&th2, NULL, tache2, "2");
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    return 0; }
```

Exclusion mutuelle & variable condition

```
#$ gcc -lpthread -o partageDonnee
partageDonnee.c
#$ ./partageDonnee
La tache 2 vient de modifier la donnee partagee
La tache 1 vient de lire la donnee partagee
La tache 2 vient de modifier la donnee partagee
La tache 2 vient de modifier la donnee partagee
La tache 2 vient de modifier la donnee partagee
La tache 2 vient de modifier la donnee partagee
La tache 1 vient de lire la donnee partagee
La tache 2 vient de modifier la donnee partagee
La tache 2 vient de modifier la donnee partagee
La tache 1 vient de lire la donnee partagee
La tache 2 vient de modifier la donnee partagee
La tache 1 vient de lire la donnee partagee
La tache 2 vient de modifier la donnee partagee
La tache 2 vient de modifier la donnee partagee
```

Tâches Périodiques sous POSIX

- Certaines tâches peuvent s'exécuter de manière **périodique**
  - une tâche *th1* disposant d'une période de *p* unités de temps doit toujours débuter une nouvelle exécution après l'écoulement de ce temps
- Dans certaines applications, cette périodicité doit être très stricte et rigoureuse

- Une technique permettant de gérer la périodicité stricte des tâches :
  - La déclaration de la structure de données de gestion du temps (elle fait appel à la déclaration de la bibliothèque ***time.h***) : **struct timespec time**;
  - La fonction permettant de récupérer le temps de l'horloge du système :

int clock_gettime (clockid_t clk_id, struct timespec *time)

  - Les fonctions de manipulation de **mutex** et de la **variable condition** , plus la fonction suivante :

int pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *verrou, struct timespec *time)

Cette fonction utilise un décompteur (*timeout*) sur le temps ***time*** pour réveiller la tâche endormie sur l'attente de la variable condition ***cond*** qui ne sera jamais signalée.

Tâches Périodiques sous POSIX

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
void* tachePeridique(void* periode){
 pthread_cond_t    cond;
 pthread_mutex_t    verrou;
 struct timespec    time;
 pthread_cond_init (&cond, NULL);
 pthread_mutex_init (&verrou, NULL);

 int i=0;
 clock_gettime (CLOCK_REALTIME, &time);
 while(i<10){
   pthread_mutex_lock (&verrou);
   time.tv_sec = time.tv_sec + (int) periode;
   printf("La tache %s s'execute periodiquement à
   l'instant %d secondes\n", "t1", (int) time.tv_sec);
   //suite du code
   pthread_cond_timedwait(&cond, &verrou, &time);
   pthread_mutex_unlock(&verrou);
   i++;
 }
}
```

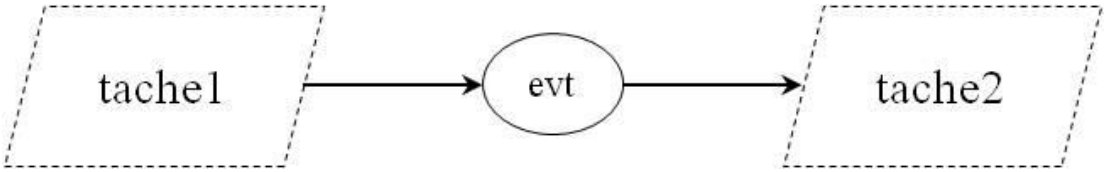
```c
int main(void)
{
   pthread_t  tache1;
   pthread_create (&tache1, NULL, tachePeridique,
                          (void*) 5);
  //la tache1 est périodique de periode 5s
   pthread_join( tache1, NULL);
   return 0;
}
```

```
#$ gcc -lpthread -o tachePeriodique tachePeriodique.c
#$ sudo ./tachePeriodique
[sudo] password for georges: .....

La tache t1 s'execute periodiquement Ã  l'instant 1389203510 secondes
La tache t1 s'execute periodiquement Ã  l'instant 1389203515 secondes
La tache t1 s'execute periodiquement Ã  l'instant 1389203520 secondes
La tache t1 s'execute periodiquement Ã  l'instant 1389203525 secondes
La tache t1 s'execute periodiquement Ã  l'instant 1389203530 secondes
La tache t1 s'execute periodiquement Ã  l'instant 1389203535 secondes
La tache t1 s'execute periodiquement Ã  l'instant 1389203540 secondes
La tache t1 s'execute periodiquement Ã  l'instant 1389203545 secondes
La tache t1 s'execute periodiquement Ã  l'instant 1389203550 secondes
La tache t1 s'execute periodiquement Ã  l'instant 1389203555 secondes
```

Synchronisation entre Tâches sous POSIX

- Les tâches soient soumises à **des contraintes de précédence**
  - mettre sur pied un mécanisme de ***synchronisation sur événement***



- Un ***sémaphore*** est la variable utilisée pour gérer la synchronisation sur événement entre deux tâches, il faut rajouter dans l'entête de votre programme la bibliothèque suivante : **#include <semaphore.h>**

1. La déclaration du ***sémaphore*** représentant ainsi l'événement à manipuler : **sem_t evt**
2. La fonction qui oblige une tâche à attendre sur l'événement *evt* : **int sem_wait (sem_t *evt)**
3. La fonction permettant à une tâche de signaler l'événement *evt* : **int sem_post (sem_t *evt)**
4. La fonction permettant d'initialiser le sémaphore

   int sem_init (sem_t *evt, int pshared, unsigned int valeur)
5. La fonction permettant de détruire un sémaphore. Une fois cette fonction appelée, aucune tâche ne doit plus être bloquée sur ce dernier : **int sem_destroy (sem_t *evt)**

Synchronisation entre Tâches sous POSIX

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
sem_t evt;
//déclaration du sémaphore représentant
l'événement de synchronisation

void* tache1(void *arg){
 int i=0;
 while(i<10){
   printf("La tache %s s'execute\n", (char*) arg);
   //suite du code
   sem_post(&evt);
  //la tâche 1 émet l'événement à la fin de son
   exécution
   i++;
 }
}
```

```c
void* tache2(void *arg){
 int i=0;
 while(i<10){
   sem_wait(&evt);
//la tâche 2 est bloquée en attente de l'émission de
l'événement qui lui permettra de poursuivre
  printf("La tache %s s'execute enfin\n", (char*) arg);
   //suite du code
   i++;
 } }

int main()
{
   pthread_t th1, th2;
   sem_init( &evt, 0, 0);
   // le sémaphore est local au processus issu de
    la fonction main() et a un compteur initialisé a 0
   pthread_create (&th1, NULL, tache1, "1");
   pthread_create (&th2, NULL, tache2, "2");
   pthread_join(th1, NULL);
   pthread_join(th2, NULL);
   return 0;
}
```

Exercice

Un pont supporte une charge maximale de 15 tonnes. Ce pont est traversé par des camions dont le poids est de 15 tonnes ainsi que par des voitures dont le poids est de 5 tonnes. On vous demande de gérer l'accès au pont de sorte que la charge maximale du pont soit respectée.

Questions :

1. Donnez un programme comportant un moniteur (une fonction d'acquisition et une fonction de libération du pont) qui simule les règles de partage du pont ci-dessus. Votre programme modélisera les camions et voitures sous la forme de threads.

2. Quand un véhicule quitte le pont, on souhaite donner la priorité aux camions : lorsqu'une voiture et un camion sont bloqués en attente d'obtenir l'accès au pont, le camion doit être réveillé en premier, sous réserve, bien sûr, que la capacité maximale du pont soit respectée.