Travaux Pratique
Systemes Temps-Réel

1. Introduction

Par défaut, le langage C (par exemple la version standardisée en 1999) ne dispose pas d'instructions natives permettant d'écrire directement des programmes multitâches. Ainsi, la bibliothèque *pthread* définie dans les normes *POSIX 1003.1c et 1003.1j* lui est associée pour atteindre cet objectif. Fin 2011, une nouvelle version du langage C (**langage C11**) a été standardisée, lui permettant désormais de disposer des bibliothèques natives pour la programmation multitâche. Mais dans ce TP, nous ne nous intéressons pas à cette dernière version et considérons le langage C dans sa version antérieure. Ce TP propose un double objectif : la présentation d'une panoplie de fonctions des normes *POSIX 1003.1c et 1003.1j* 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, inversion de priorité, etc.

2. Installation et Utilisation de Pthread

Dans ce TP, nous considérons que l'application multitâche est exécutée sur une plate-forme monoprocesseur (un seul processeur). L'exécution parallèle des tâches est alors virtuelle dans ce cas. Ainsi c'est donc au système d'exploitation installé sur l'architecture matérielle sur laquelle s'exécute l'application d'utiliser ses mécanismes internes pour gérer l'exécution des différentes tâches de l'application, afin de leurs permettre de respecter leurs différentes contraintes.

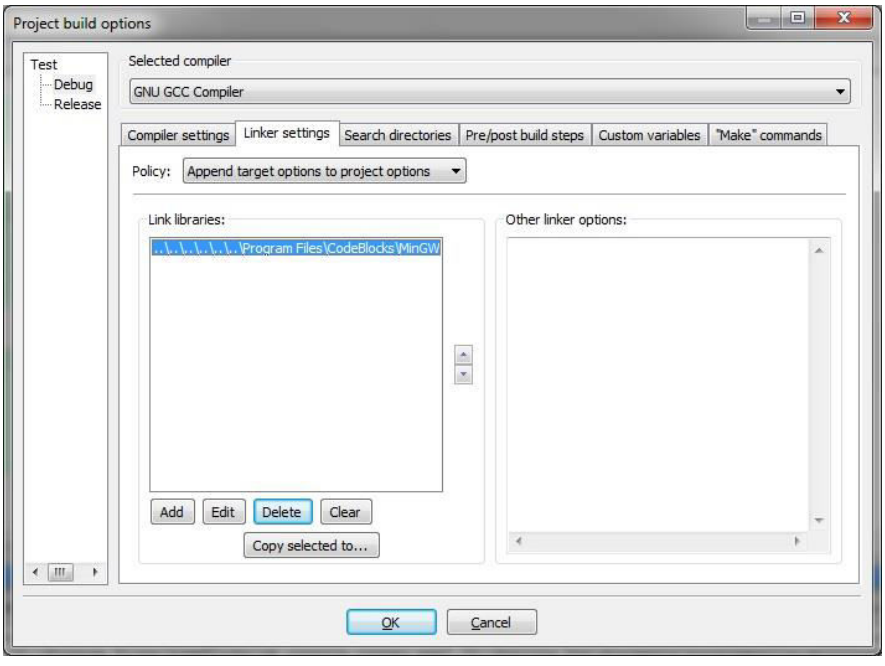
Il est possible d'écrire des programmes multitâches avec *pthread* aussi bien sur Windows que sur Linux. Linux par défaut dispose déjà d'une bibliothèque *pthread* qui lui est intégrée, tandis que Windows n'en dispose pas. Il est donc nécessaire de l'installer sur ce dernier. Par la suite, nous présenterons les différentes étapes d'installation de *pthread* sous Windows. Puis nous présenterons également les différents éléments nécessaires pour l'exécution d'un programme utilisant *pthread* sur Linux.

- **Sous windows:** La bibliothèque *pthread* est téléchargeable de ce lien (https://www.sourceware.org/pthreads-win32/). Pour l'instant, toutes les fonctions définies dans cette bibliothèque ne sont pas encore implémentées, ce qui limite sa portabilité. L'ensemble des éléments implémentés et non implémentés de *pthread* sont visibles à travers le lien suivant (https://www.sourceware.org/pthreads-win32/conformance.html). Sur la page de *pthread*, téléchargez le fichier exécutable le plus récent, actuellement :*pthreads-w32-2-8-0-*

release.exe. Exécutez-le pour dézipper son contenu dans un répertoire de votre choix. Si vous utilisez par exemple Code::Blocks, voici la procédure à suivre :

- répérez l'emplacement du compilateur C (*MinGW*) dans le répertoire d'installation de Code::Blocks sous votre machine (ex: C:\Program Files\CodeBlocks\MinGW) ;
- copiez les fichiers *.h* situés dans le répertoire *..\Pre-built.2\include* du dossier où vous avez dézippé la librairie *pthread* téléchargée précédemment. Collez ces fichiers dans le répertoire C:\Program Files\CodeBlocks\MinGW\include ;
- copiez les fichiers *.a* et *.lib* situés dans le répertoire *..\Pre-built.2\lib* du dossier où vous avez dézippé la librairie *pthread*. Collez ces fichiers dans le répertoire C:\Program Files\CodeBlocks\MinGW\lib ;
- copiez les fichiers *.dll* situés dans le répertoire *..\Pre-built.2\lib* du dossier où vous avez dézippé la librairie *pthread*. Collez ces fichiers dans le répertoire C:\Program Files\CodeBlocks\MinGW\bin.

Sous Code::Blocks, si cela n'est déjà pas automatiquement fait, ajoutez le chemin où est installé MinGW sur votre machine (ex: C:\Program Files\CodeBlocks\MinGW) dans l'onglet *Settings-->Compiler-->Toolchain executables-->compiler's installation directory*. Puis validez avec le bouton *ok*.
Maintenant, créez un projet C sous Code::Blocks dans lequel vous allez développer votre application. Cliquez droit sur ce projet et choisissez le menu *Build options...*, puis *Linker settings*. Cliquez sur le bouton *Add* et renseignez l'emplacement qui mène au fichier*libpthreadGC2.a* (avec ce fichier y compris) que vous aviez manuellement mis dans le répertoire *C:\Program Files\CodeBlocks\MinGW\lib*. Puis, Code::Blocks vous demandera s'il enregistre cet emplacement avec un *chemin* relatif. Acceptez cette proposition.



Note : Lorsque vous exécuterez votre programme en cliquant sur le bouton *run* de Code::Blocks, tout devrait fonctionner. Mais si vous voulez directement lancer votre programme en double cliquant sur l'exécutable (le *.exe*) situé dans le dossier projet sous votre

machine, vous devrez tout d'abord ajouter manuellement le fichier *pthreadGC2.dll* dans ce dossier près de l'exécutable.

- Sous Linux : Sous le système d'exploitation Linux, le compilateur C qui y est intégré dispose d'ores et déjà de la bibliothèque *pthread*. Il ne nécessite donc aucune installation. Pour écrire votre programme, créez un fichier d'extension *.c*, avec l'éditeur de texte *gedit* par exemple (pour de petits programmes, sinon vous pouvez installer un IDE dédié). Lors de la compilation et de l'exécution de votre programme (nommé par exemple *monprogramme.c*), sous la console, placez-vous dans le répertoire où il se trouve et exécutez respectivement les commandes suivantes :

- gcc -lpthread -o *monprogramme monprogramme*.c (pour la compilation) ;
- *./monprogramme* (pour l'exécution).

Lors de la compilation, l'option *-lpthread* indique au compilateur que le programme utilise la bibliothèque *pthread* (parfois il faut mettre cette option en fin de ligne si le compilateur ne reconnaît pas les primitives *pthread* de votre programme). L'option *-o* permet de générer le fichier exécutable *monprogramme* issu du fichier source *monprogramme.c*.

Par la suite, nous compilerons et exécuterons les programmes d'application de ce cours sur Linux, car ce système est mieux adapté pour le multitâche que Windows.

3. Notion d'une tâche

Une *tâche* est une unité active d'une application. Il s'agit d'un ensemble d'instructions séquentielles correspondant souvent à une procédure ou à une fonction.
Une tâche est 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***.

3.1. Création de tâches sous Posix

Un programme C utilisant la bibliothèque *pthread* doit disposer de l'entête suivant :

```
#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? ...). Nous développerons au fur et à mesure cette variable dans les prochains paragraphes ;
- ***start_routine*** : c'est la fonction C qui sera exécutée par la tâche qui est créée ;
- ***arg*** : pointeur 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.

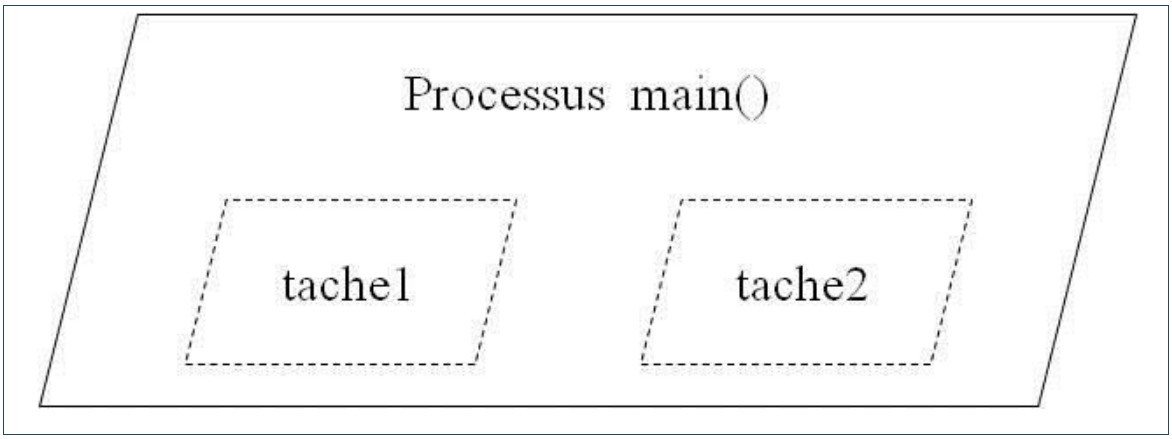
Lorsqu'un processus lourd crée une tâche sous Posix, s'il ne lui est pas explicitement indiqué d'attendre la fin d'exécution de cette tâche, alors à sa terminaison elle forcera l'arrêt de la tâche créée. Pour éviter cela, Posix propose la fonction *pthread_join*. Le prototype de la fonction *pthread_join*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

3.2. Petit programme à 2 tâches

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.



Le code source est le suivant :

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

```c
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;
}
```

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
```

Nous remarquons lors de l'exécution un ordre d'affichage arbitraire entre les tâches 1 et 2. Cet ordre d'affichage peut varier d'une exécution à une autre.

3.3. 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
```

Des informations supplémentaires à ce sujet sont trouvables (http://www.domaigne.com/blog/computing/joinable-and-detached-threads/).

4. Priorité et ordonnancement de tâches

Quelques fois dans une application multitâche, les tâches doivent s'exécuter selon leur importance et un ordre bien spécifique de les exécuter doit être choisi : c'est la problématique d'affectation des priorités aux tâches et du type de leur ordonnancement. Cette partie est une problématique généralement traitée dans le domaine des systèmes temps réel. 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.

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)
```

Le code source suivant présente le petit programme à deux tâches précédent avec prise en compte des paramètres d'ordonnancement (priorité et type d'ordonnancement).

```
#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\n", (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);
```

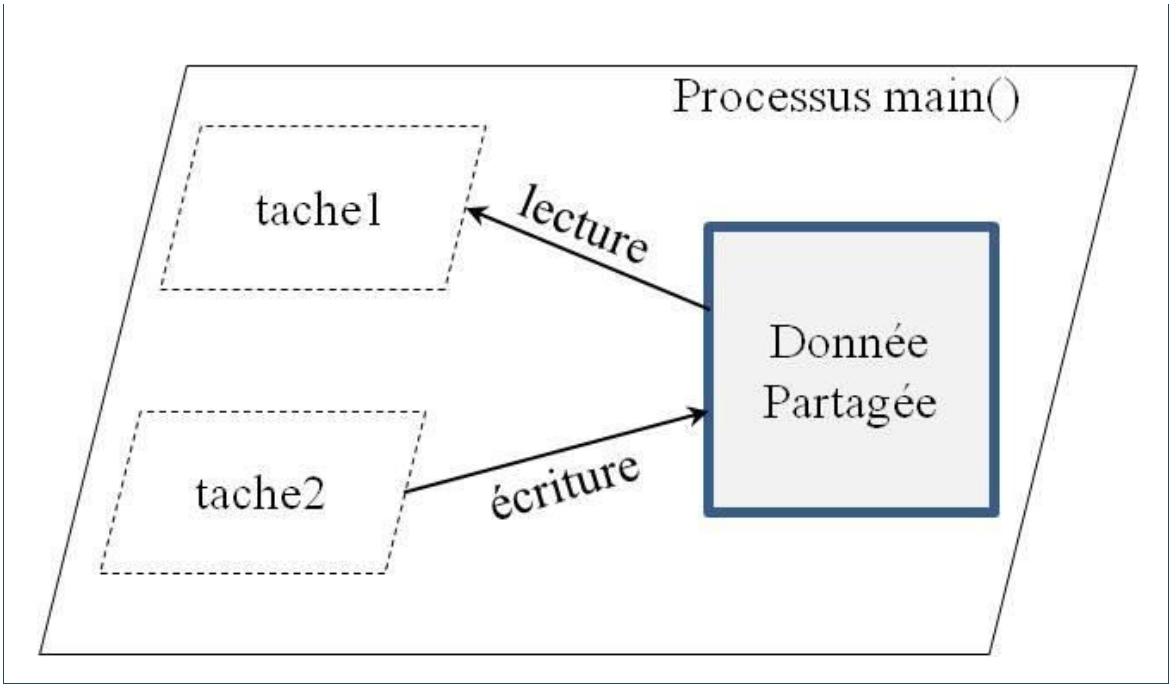
```
    pthread_create(&tache1, &attr, fonc, 1); // la tâche tache1 créée, sera ordonnancée en SHED_FIFO avec une priorité de 10

    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;
}
```

Un programme sur Linux utilisant des primitives temps réel tels que des paramètres d'ordonnancement (priorités, politiques d'ordonnancement...), doit être exécuté en mode *root* avec le mot clé *sudo*. La sortie du programme précédent est la suivante :

```
#$ gcc -lpthread -o monprog monprog.c
#$ sudo ./monprog
[sudo] password for georges: .....
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 : 6
```

5. Gérer le partage de données entre tâches

Plusieurs tâches lorsqu'elles s'exécutent peuvent accéder à une donnée partagée qui leur est commune (voir la figure) pour réaliser leurs besoins. Il s'agit du problème de la communication asynchrone entre tâches. Le terme *asynchrone* signifie que les instants d'accès à la donnée partagée par les tâches ne sont pas connus a priori.



L'une d'entre elles (la tâche 2) peut par exemple écrire sur la donnée partagée pendant que l'autre (la tâche 1) est chargée de la lire. Pour gérer le problème de la cohérence de la donnée partagée (c'est-à-dire éviter qu'elle soit corrompue au travers des accès aléatoires), il faut éviter par exemple qu'une tâche lise la donnée pendant qu'elle est entrain d'être modifiée, ou qu'une tâche modifie la donnée pendant qu'elle est entrain d'être lue. Pour résoudre ce problème, des mécanismes d'*exclusion mutuelle* doivent dont être mis sur pied pour protéger la donnée partagée. Nous allons montrer comment cela peut être fait sous Posix.

5.1. Exclusion mutuelle

L'exclusion mutuelle est le mécanisme qui permet qu'une et une seule tâche accède à une ressource partagée à la fois à un instant donné. Pour cela, on utilise une variable spéciale appelée *sémaphore d'exclusion mutuelle* qui joue le rôle de *verrou* pour accéder à la ressource. Sous Posix, elle est mise en place via les quatre éléments suivants :

1. La déclaration de votre donnée/ressource partagée. Elle 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.

Sélectionnez

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

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

Sélectionnez

```
pthread_mutex_lock(pthread_mutex_t *verrou)
```

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

Sélectionnez

```
pthread_mutex_unlock(pthread_mutex_t *verrou)
```

Le code source qui suit présente l'implémentation de l'exemple précédent à deux tâches, qui doivent se partager une donnée de type enregistrement, l'une chargée de la lire et l'autre chargée de la modifier.

Sélectionnez

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

typedef struct {    // déclaration du type de la donnée partagée
   float taille;
   float poids;
} type_donneePartagee;

pthread_mutex_t verrou; //déclaration du verrou
type_donneePartagee donneePartagee; //déclaration de la donnée partagée

void* tache1(void *arg){ // déclaration du corps de la tâche qui lit la donnée partagée. On considère qu'elle s'exécute indéfiniment
 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 donnee partagee\n", (char*) arg);
   //utilisation de ma_donneePartagee
   usleep(1000000);
   i++;
 }
}

void* tache2(void *arg){ // déclaration du corps de la tâche qui modifie la donnée partagée. On considère qu'elle s'exécute indéfiniment
 int i=0;
 while(i<10){
   pthread_mutex_lock(&verrou);
```

```
    donneePartagee.taille = 100 + rand()%101; //choisir une taille au hasard entre 100 et 200cm. // Dans un programme réel, les données à modifier peuvent provenir de capteurs et nécessitent un code un peu plus complexe
    donneePartagee.poids = 10 + rand()%101;
    pthread_mutex_unlock(&verrou);
    printf("La tache %s vient de modifier la donnee partagee\n", (char*) arg);
    usleep(1000000);
    i++;
  }
}

int main(void)
{
    srand(200);
    pthread_t th1, th2;
    pthread_mutex_init(&verrou, NULL);

    //initialisation de la donnée partagée
    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;
}
```

Résultat d'exécution

Sélectionnez

```
#$ 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
```

5.2. Exclusion mutuelle et variable condition

Il peut arriver qu'une condition soit placée 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 tâche 2 ne lit la donnée partagée que si la *taille* et le *poids* écrits par la tâche 1 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(pthread_cond_t *cond)* 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)
```

Le code ci-dessous reprend l'exemple du paragraphe V-A précédent, en intégrant une variable condition. La tâche 1 ne peut lire qu'un poids et une taille supérieurs respectivement à 60kg et 120cm, sinon, elle s'endort.

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


typedef struct {    //déclaration du type de la donnée partagée
   float taille;
```

```
    float poids;
} type_donneePartagee;

pthread_mutex_t verrou; //déclaration du verrou
pthread_cond_t cond = PTHREAD_COND_INITIALIZER; //initialisation de la variable condition

type_donneePartagee donneePartagee; //déclaration de la donnée partagée

void* tache1(void *arg){ // déclaration du corps de la tâche qui lit la donnée partagée. On considère qu'elle s'exécute indéfiniment
 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);
    //utilisation de ma_donneePartagee
    usleep(1000000);
    i++;
 }
}

void* tache2(void *arg){ // déclaration du corps de la tâche qui modifie la donnée partagée. On considère qu'elle s'exécute indéfiniment
 int i=0;
 while(i<10){
    pthread_mutex_lock(&verrou);
    donneePartagee.taille = 100 + rand()%101; //choisir une taille au hasard entre 100 et 200cm. // Dans un programme réel, les données à modifier peuvent provenir de capteurs et nécessitent un code un peu plus complexe
    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)
{
    srand(200);
    pthread_t th1, th2;
    pthread_mutex_init(&verrou, NULL);
```

```
    //initialisation de la donnée partagée
    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;
}
```

Résultat d'exécution

Sélectionnez

```
#$ 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
```

6. Le problème d'inversion de priorité

6.1. Description du problème

L'inversion de priorité est le problème qui exprime le fait qu'une tâche empêche l'exécution d'une autre tâche de priorité supérieure à elle. Elle est généralement une conséquence du problème de partage de donnée entre tâches présenté dans le paragraphe V précédent. Cette partie est une problématique généralement traitée dans le domaine des systèmes temps réel. Pour mieux illustrer ce problème considérons le scénario suivant :

Soient trois tâches *th1, th2, th3* et un mutex noté *verrou*. Supposons que ces tâches sont telles que *priorité(th3) > priorité(th2) > priorité(th1)*. Initialement, les tâches *th3* et *th2* sont endormies. La tâche *th1* commence son exécution en premier et s'empare du mutex *verrou*. Puis, la tâche *th3* plus prioritaire se réveille et préempte *th1*, s'exécute et cherche ensuite à prendre le mutex*verrou*. Mais elle se bloque et s'endort à nouveau, car *th1* a déjà pris le *verrou*. Suite à cela, la tâche *th1* se réveille et continue son exécution. Peu après, la tâche *th2* (qui n'utilise jamais le verrou) plus prioritaire que *th1*, se réveille aussi et préempte *th1*. Le résultat est dont le suivant :

- la tâche *th2* s'exécute ;

- la tâche *th1* est préemptée, mais détient le mutex *verrou* ;

- la tâche *th3* est bloquée et endormie en attente du mutex *verrou*.

Le fait que la tâche *th3* est bloquée alors que la tâche *th2* (moins prioritaire qu'elle) est entrain de s'exécuter, s'appelle le*problème d'inversion de priorité*.

6.2. Résolution du problème

Pour résoudre le problème de l'inversion de priorité, une priorité correspondant à celle de la tâche la plus prioritaire (*th3*) doit être affectée au mutex *verrou*. Il faut également associer aux tâches un protocole à héritage de la priorité du *verrou*. Ainsi, le scénario suivant sera celui qui évite l'inversion de priorité :

- La tâche *th1* commence son exécution, s'empare du *verrou* et hérite de la priorité de ce dernier ;
- La tâche *th3* se réveille, préempte *th1* (qui a hérité d'une priorité égale à la sienne) et commence son exécution, mais se bloque et se rendort sur le *verrou* déjà pris par *th1* ;
- La tâche *th1* reprend la main et continue son exécution dans sa section critique ; - La tâche *th2* se réveille à son tour mais ne peut débuter son exécution, car la tâche *th1* en cours d'exécution est la plus prioritaire (*th1* a hérité de la priorité du *verrou* qui est celle de la tâche la plus prioritaire du système) ;
- Lorsque *th1* libère le verrou, elle perd la priorité de ce verrou et reprend sa priorité initiale (la moins prioritaire dans notre exemple) ;
- La tâche *th2* déjà prête ne peut pas s'exécuter, car la tâche *th3* qui était bloquée sur le *verrou* est plus prioritaire qu'elle. C'est dont *th3* qui prend la main et finit son exécution ;
- Finalement, *th2* s'exécute et se termine. Si *th1* n'avait pas fini son exécution en libérant le *verrou*, elle reprend la main et se termine aussi.

En plus des fonctions déjà présentées dans les paragraphes IV et V, voici les éléments à connaître pour résoudre ce problème :

1. La déclaration de la propriété à affecter au mutex d'exclusion mutuelle (le *verrou*) :

```
pthread_mutexattr_t m_attr;
```

2. La fonction d'initialisation de la propriété du mutex. Obligatoire avant toute manipulation de l'attribut du mutex :

```
int pthread_mutexattr_init(pthread_mutexattr_t *m_attr)
```

3. La fonction permettant à un mutex d'être partagé par les tâches appartenant à n'importe quel processus. Pour cela, la variable*pshared* doit être égale à ***PTHREAD_PROCESS_SHARED***. Si le partage est interne au processus/tâche créateur du mutex, alors*pshared* doit prendre la valeur ***PTHREAD_PROCESS_PRIVATE*** :

```

```

Sélectionnez

```
int pthread_mutexattr_setpshared(pthread_mutexattr_t *m_attr, int pshared)
```

4. La fonction permettant de mettre dans la propriété *m_attr* qui sera affectée au mutex, sa priorité plafond *prioceiling* :

```
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *m_attr, int prioceiling)
```

5. La fonction permettant de mettre dans la propriété *m_attr* qui sera affectée au mutex, son protocole d'héritage *protocol*. Dans notre cas, puisque toute tâche s'emparant du mutex doit hériter de sa priorité, *protocol* vaudra ***PTHREAD_PRIO_PROTECT***. Il existe aussi d'autres valeurs pour la variable *protocol* que sont ***PTHREAD_PRIO_NONE*** et ***PTHREAD_PRIO_INHERIT*** :

```
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *m_attr, int protocol)
```

6. La fonction permettant à une tâche disposant de la prorpiété *attr* d'hériter d'une priorité. Pour ce faire, la variable *scope* vaudra***PTHREAD_SCOPE_SYSTEM***, si la tâche est en concurrence avec toute autre tâche de tout processus du système. La variable*scope* vaudra cependant ***PTHREAD_SCOPE_PROCESS***, si la tâche est uniquement en compétition avec les tâches internes au même processus :

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

Le code source ci-dessous présente l'implémentation du scénario de résolution du problème de l'inversion de priorité pour les trois tâches précédentes.

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


#define N 4  //déclaration d'une constante à utiliser dans la boucle des tâches

typedef struct {   // déclaration du type de la donnée partagée
   float taille;
   float poids;
} type_donneePartagee;

pthread_mutex_t verrou; //déclaration du verrou
type_donneePartagee donneePartagee; //déclaration de la donnée partagée

void* tache1(void *arg){ // déclaration du corps de la tâche 1
 type_donneePartagee ma_donneePartagee;
 int i = 0;
 while(i < N){
   pthread_mutex_lock(&verrou);
   ma_donneePartagee = donneePartagee;
   //suite du code avec l'utilisation de la donnée partagée lue
```

```
    pthread_mutex_unlock(&verrou);
    printf("La tache %s a lu la donnée partagée\n", (char*) arg);
    //reste du code
    i++;
  }
}

void* tache2(void *arg){ // déclaration du corps de la tâche 2
 int i = 0;
 while(i < N){
    usleep(5000000); //attendre 5 secondes
    printf("La tache %s s'execute\n", (char*) arg);
    //reste du code sans utilisation du mutex
    i++;
  }
}

void* tache3(void *arg){ // déclaration du corps de la tâche 3
 int i = 0;
 for(i = 0; i < N; i++){
    usleep(3000000); //attendre 3 secondes
    pthread_mutex_lock(&verrou);
    donneePartagee.taille = 100 + rand()%101;
    donneePartagee.poids = 10 + rand()%101;
    pthread_mutex_unlock(&verrou);
    printf("La tache %s a écrit sur la donnée partagée\n", (char*) arg);
    //reste du code
  }
}

int main(void)
{
    srand(200);
    pthread_t th1, th2, th3;
    pthread_attr_t attr;
    pthread_mutexattr_t m_attr;
    struct sched_param param;

    /*préparation de la propriété du verrou*/
    pthread_mutexattr_init(&m_attr);
    pthread_mutexattr_setpshared(&m_attr, PTHREAD_PROCESS_SHARED);
    pthread_mutexattr_setprioceiling(&m_attr, 15); //le mutex dispose d'une priorité de 15, qui
sera celle de la tâche la plus prioritaire
    pthread_mutexattr_setprotocol(&m_attr, PTHREAD_PRIO_PROTECT);
    pthread_mutex_init(&verrou, &m_attr);

    /*initialisation de la donnée partagée*/
    donneePartagee.taille = 100 + rand()%101;
    donneePartagee.poids = 10 + rand()%101;
```

```
    /*préparation de la propriété des tâches*/
    pthread_attr_init(&attr);
    pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    pthread_attr_setschedpolicy(&attr, SCHED_FIFO);

    param.sched_priority = 15;
    pthread_attr_setschedparam(&attr, &param);
    pthread_create(&th3, &attr, tache3, "tache3"); //création tâche 3

    param.sched_priority = 12;
    pthread_attr_setschedparam(&attr, &param);
    pthread_create(&th2, &attr, tache2, "tache2"); //création tâche 2

    param.sched_priority = 9;
    pthread_attr_setschedparam(&attr, &param);
    pthread_create(&th1, &attr, tache1, "tache1"); //création tâche1

    pthread_mutexattr_destroy(&m_attr);
    pthread_attr_destroy(&attr);

    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    pthread_join(th3, NULL);
    return 0;
}
```

La sortie du programme est donnée ci-dessous. Notons que puisque des primitives d'ordonnancement sont utilisées, il faut se mettre en mode *root* pour exécuter le programme.

Résultat d'exécution

```
#$ gcc -lpthread -o inversionPriorite inversionPriorite.c
#$ sudo ./inversionPriorite
[sudo] password for georges: .....
La tache tache1 a lu la donnÃ©e partagÃ©e
La tache tache1 a lu la donnÃ©e partagÃ©e
La tache tache1 a lu la donnÃ©e partagÃ©e
La tache tache1 a lu la donnÃ©e partagÃ©e
La tache tache3 a Ã©crit sur la donnÃ©e partagÃ©e
La tache tache2 s'execute
La tache tache3 a Ã©crit sur la donnÃ©e partagÃ©e
La tache tache3 a Ã©crit sur la donnÃ©e partagÃ©e
La tache tache2 s'execute
La tache tache3 a Ã©crit sur la donnÃ©e partagÃ©e
La tache tache2 s'execute
La tache tache2 s'execute
```

7. Gérer la périodicité d'une tâche

Dans une application multitâche, certaines tâches peuvent s'exécuter de manière périodiques. C'est-à-dire qu'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. Dans ce cas, l'application doit s'exécuter en utilisant directement le temps fourni par l'horloge système.
Il est donc intéressant de voir comment cela peut être traité sous Posix. Dans les exemples précédents, vous avez pu remarquer que la fonction *usleep(int p)* a été utilisée pour faire attendre une tâche pendant une période *p*. Mais pour des raisons de *dérive d'horloge* que nous ne détaillons pas ici, cette technique n'est pas rigoureuse dans la gestion de la périodicité de la tâche. Dans cette partie, sans tenir compte des solutions propriétaires Posix non généralement portables, nous présentons une technique permettant de gérer la périodicité stricte des tâches. Les éléments suivants sont à connaître :

1. 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;
```

2. La fonction permettant de récupérer le temps de l'horloge du système :

```
int clock_gettime(clockid_t clk_id, struct timespec *time)
```

La variable *clk_id* peut prendre les différentes valeurs suivantes : ***CLOCK_REALTIME***, ***CLOCK_MONOTONIC***,***CLOCK_PROCESS_CPUTIME_ID***, ***CLOCK_THREAD_CPUTIME_ID***. Attention, plusieurs de ces primitives ne fonctionnent pas sous Windows.

3. Les fonctions de manipulation de mutex et de variable condition que nous avons présenté dans le paragraphe V, 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.

Le code ci-dessous présente un exemple illustrant la mise en place d'une tâche périodique.

```
#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;
```

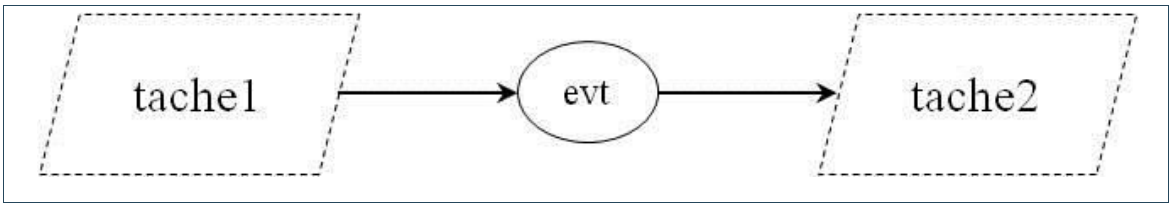
```
 pthread_cond_init(&cond, NULL);
 pthread_mutex_init(&verrou, NULL);

 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++;
 }
}

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
```

L'on remarque une activation stricte de la tâche toutes les 5 secondes : la différence en valeur absolue entre deux instants consécutifs d'activation est égale à 5.

8. Gérer la synchronisation entre tâches

Dans une application multitâche, il peut arriver que des tâches soient soumises à des contraintes de précédence. C'est-à-dire qu'une ou plusieurs tâches s'active(nt) après qu'une autre ait terminé son exécution. Pour résoudre un tel problème, il faut mettre sur pied un mécanisme de *synchronisation sur événement* (voir figure). Le scénario d'un tel mécanisme est simple : la tâche 2 au début de son exécution est bloquée sur l'attente d'un événement qui lui permettra de continuer son exécution. Pendant ce temps, la tâche 1 s'exécute et à la fin de

son exécution, elle émet l'événement attendu par la tâche 2 afin que cette dernière débute enfin son exécution.
Nous montrons comment ce problème peut être traité sous Posix avec les *sémaphores à compte*.



Le *sémaphore à compte* est la variable utilisée pour gérer la synchronisation sur événement entre deux tâches. Mais, elle n'est malheureusement pas définie dans les normes Posix *1003.1c* et *1003.1j*, mais plutôt dans la *1003.1b*. Cela dit, pour pouvoir l'utiliser, il faut rajouter dans l'entête de votre programme la bibliothèque suivante :

```
#include <semaphore.h>;
```

Puis, les principaux éléments à connaître sont les suivants :

1. La déclaration du *sémaphore à compte* 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 à compte. Initialement, ce sémaphore est fait pour gérer la synchronisation entre processus lourd. Mais, on peut le restreindre à gérer uniquement la synchronisation entre les tâches du processus courant (dans notre cas, le processus issu de la fonction main(void)), en fixant la variable *pshared* à **0**. Ce sémaphore est partagé entre plusieurs processus si la variable *pshared* est différente de 0. La variable *valeur* représente le compteur associé au sémaphore *evt*. C'est-à-dire le nombre maximum d'événements que la tâche 2 reçoit de la tâche 1 à un moment donné. Elle se doit alors de tous les considérer :

```
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)
```

Le code source ci-dessous présente un bref exemple de synchronisation entre les deux tâches présentées sur la figure précédente.

```
 #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++;
 }
}

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, 2); // le sémaphore est local au processus issu de la fonction main() et a un compteur initialisé a 2

   pthread_create(&th1, NULL, tache1, "1");
   pthread_create(&th2, NULL, tache2, "2");

   pthread_join(th1, NULL);
   pthread_join(th2, NULL);
   return 0;
}
```

Résultat d'exécution

```
#$ gcc -lpthread -o synchronisation synchronisation.c
#$ ./synchronisation
La tache 1 s'execute
La tache 1 s'execute
La tache 1 s'execute
```

```
La tache 1 s'execute
La tache 1 s'execute
La tache 1 s'execute
La tache 1 s'execute
La tache 1 s'execute
La tache 1 s'execute
La tache 1 s'execute
La tache 2 s'execute enfin
La tache 2 s'execute enfin
La tache 2 s'execute enfin
La tache 2 s'execute enfin
La tache 2 s'execute enfin
La tache 2 s'execute enfin
La tache 2 s'execute enfin
La tache 2 s'execute enfin
La tache 2 s'execute enfin
La tache 2 s'execute enfin
```