Sommaire

Séquence 1 : multi-tâche

On crée deux tâches (*threads*) : une affiche des étoiles '*' et l'autre des dièses '#' :

```
// threads.1.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

// Fonctions correspondant au corps d'un thread (tache)
void *etoile(void *inutilise);
void *diese(void *inutilise);

int main(void)
{
  pthread_t thrEtoile, thrDiese;

  setbuf(stdout, NULL);
  printf("Je vais creer et lancer 2 threads\n");

  pthread_create(&thrEtoile, NULL, etoile, NULL);
  pthread_create(&thrDiese, NULL, diese, NULL);

  //printf("J'attends la fin des 2 threads\n");
  pthread_join(thrEtoile, NULL);
  pthread_join(thrDiese, NULL);

  printf("\nLes 2 threads se sont termines\n");

  printf("Fin du thread principal\n");

  pthread_exit(NULL);

  return EXIT_SUCCESS;
}

void *etoile(void *inutilise)
{
  int i;
  char c1 = '*';

  for(i=1;i<=200;i++)
  {
     write(1, &c1, 1);// écrit un caractère sur stdout (descripteur 1)
  }

  return NULL;
}
```

```
void *diese(void *inutilise)
{
  int i;
  char c1 = '#';

  for(i=1;i<=200;i++)
  {
      write(1, &c1, 1);
  }

  return NULL;
}
```

Compilation :

```
$ gcc threads.1.c -o threads.1 -D_REENTRANT -pthread
```

Exemple d'exécution :

```
$ ./threads.1
Je vais creer et lancer 2 threads
*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#
*#*#*#*#*#*###########*#*#*#*****#*#*#*#*#*#*******
Les 2 threads se sont termines
Fin du thread principal
```

1) Compiler et tester le programme ci-dessous.

Séquence 2 : synchronisation de donnée

Dans la programmation concurrente, le terme de **synchronisation** se réfère à deux concepts distincts (mais liés) :

- la synchronisation de processus
- la synchronisation de données.

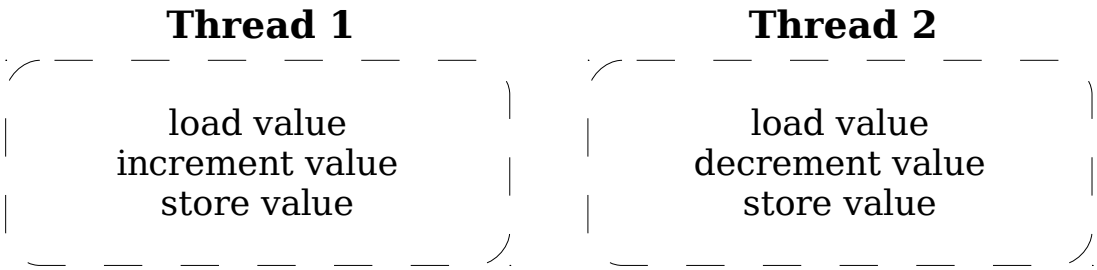
La **synchronisation de processus** est un mécanisme qui vise à bloquer l'exécution des différents processus à des points précis de leur programme de manière à ce que tous les processus passent les étapes bloquantes au moment prévu par le programmeur.

La **synchronisation de donnée** est un mécanisme qui vise à conserver la cohérence entre différentes données dans un environnement multitâche.

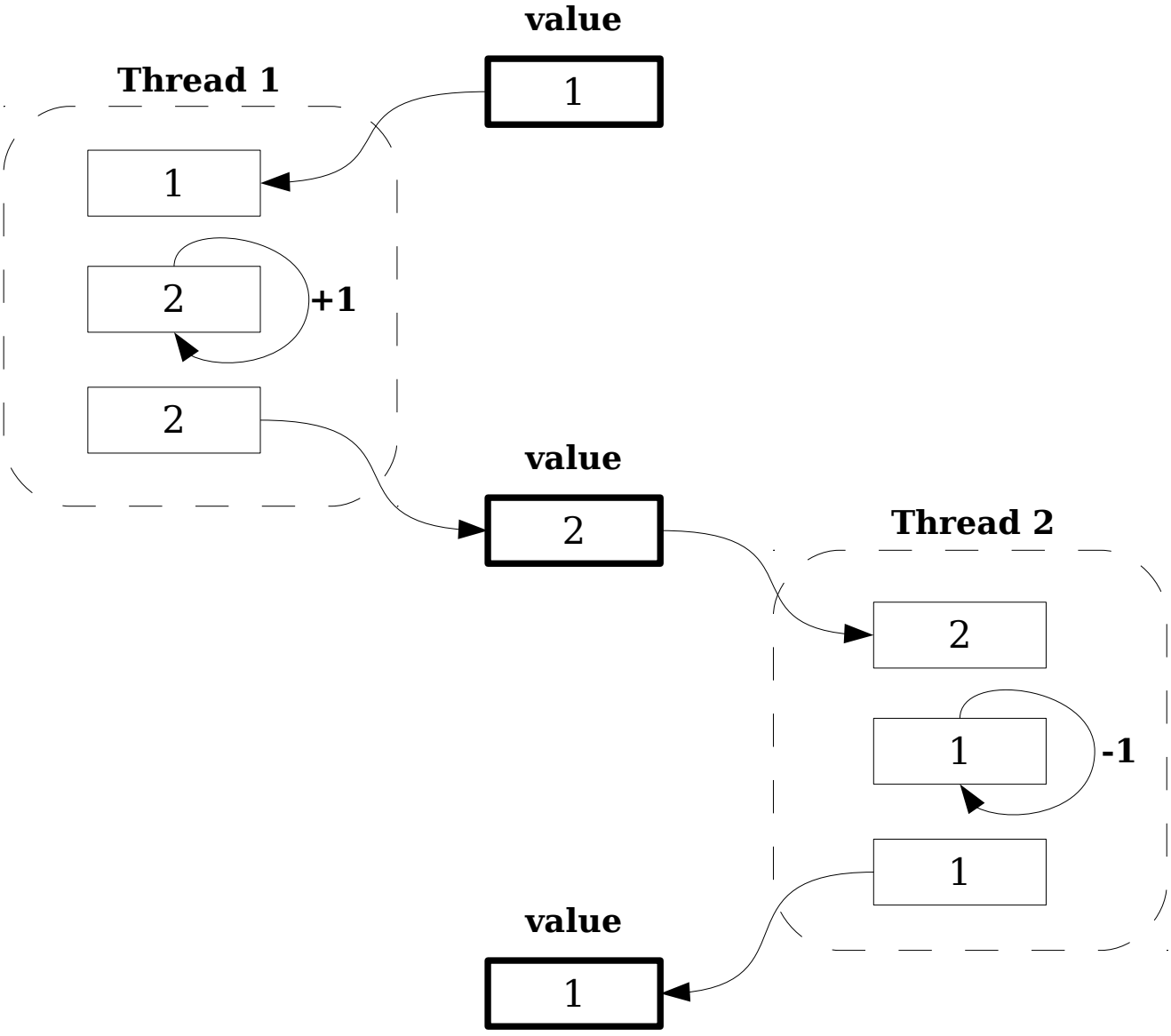
Les problèmes liés à la synchronisation rendent toujours la programmation plus difficile.

On va commencer par mettre en évidence le problème classique de la synchronisation de données.

On va créer deux tâches : une incrémente une variable partagée et l'autre la décrémente.



Un déroulement possible serait :



En réalité, le résultat n'est pas prévisible, du fait que vous ne pouvez savoir l'ordre d'exécution des instructions.

Le code à tester est le suivant :

```
// threads.2a.c

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

int value_globale = 1;

// Chaque thread (tache) va faire ses COUNT boucles
#define COUNT  5

// Fonctions correspondant au corps d'un thread (tache)
void *increment(void *inutilise);
void *decrement(void *inutilise);

int main(void)
{
  pthread_t thread1, thread2;

  printf("Avant les threads : value = %d\n", value_globale);
  pthread_create(&thread1, NULL, increment, NULL);
  pthread_create(&thread2, NULL, decrement, NULL);

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  printf("Apres les threads : value = %d\n", value_globale);
  printf("Fin du thread principal\n");

  pthread_exit(NULL);
  return EXIT_SUCCESS;
}

void *increment(void *inutilise)
{
   int value;
   int count = 0;

   while(1)
   {
      value = value_globale;
      printf("Thread1 : load value (value = %d) ", value);
      value += 1;
      printf("Thread1 : increment value (value = %d) ", value);
      value_globale = value;
      printf("Thread1 : store value (value = %d) ", value_globale);
      count++;

      if(count >= COUNT) {
         printf("Le thread1 a fait ses %d boucles\n", count);
         return(NULL);
      }
   }
   return NULL;
}
```

```
void *decrement(void *inutilise)
{
   int value;
   int count = 0;

   while(1)
   {
      value = value_globale;
      printf("Thread2 : load value (value = %d) ", value);
      value -= 1;
      printf("Thread2 : decrement value (value = %d) ", value);
      value_globale = value;
      printf("Thread2 : store value (value = %d) ", value_globale);
      count++;

      if(count >= COUNT)
      {
         printf("Le thread2 a fait ses %d boucles\n", count);
         return(NULL);
      }
   }
   return NULL;
}
```

Les deux tâches réalisent le même nombre de traitement (COUNT). On suppose donc que la variable globale (value_globale) doit revenir à sa valeur initiale (1) puisqu'il y aura le même nombre d'incrémentation et de décrémentation.

2) Tester plusieurs fois avec des valeurs différentes de COUNT. Commenter les résultats obtenus.

Pour résoudre ce genre de problème, le système doit permettre au programmeur d'utiliser un **verrou d'exclusion mutuelle**, c'est-à-dire de pouvoir bloquer, en une seule instruction (atomique), tous les tâches tentant d'accéder à cette donnée, puis, que ces tâches puissent y accéder lorsque la variable est libérée.

Remarque : une instruction atomique est une instruction qui ne peut être divisée (donc interrompue).

Un **mutex** est un objet d'exclusion mutuelle (*MUTual EXclusion*), et est très pratique pour protéger des données partagées de modifications concurrentes et pour implémenter des **sections critiques**.

Un mutex peut ête dans deux états : déverrouillé ou verrouillé (possédé par un thread). Un mutex ne peut être la pris que par un seul thread à la fois. Un thread qui tente de verrouiller un mutex déjà verrouillé est suspendu jusqu'à ce que le mutex soit déverrouillé.

Les sémaphores d'exclusion mutuelle sont de type **pthread_mutex_t**. Chaque sémaphore possède des attributs de type pthread_mutexattr_t. La valeur prédéfinie pour les attributs de sémaphores est pthread_mutexattr_default (POSIX).

pthread_mutex_init initialise le mutex pointé par mutex selon les attributs de mutex spécifié par mutexattr. Si mutexattr vaut NULL, les paramètres par défaut sont utilisés.

```
int pthread_mutex_init(pthread_mutex_t *mutex, const
pthread_mutexattr_t *mutexattr);
```

L'implémentation LinuxThreads ne supporte qu'un seul attribut, le type de mutex, qui peut être soit "rapide", "récursif" ou à "vérification d d'erreur". Le type de mutex détermine s'il peut être verrouillé plusieurs fois par le même thread. Le type par défaut est "rapide". Il faut lire la page man de pthread_mutexattr_init(3) pour plus d'informations sur les attributs de mutex et leur utilisation.

pthread_mutex_lock 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 appelant jusqu'à ce que le mutex soit déverrouillé.

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

pthread_mutex_unlock déverrouille le mutex.

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

pthread_mutex_destroy détruit un mutex, libérant les ressources qu'il détient. Le mutex doit être déverrouillé.

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

Exemple d'utilisation d'un mutex :

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

pthread_mutex_t globale_lock    = PTHREAD_MUTEX_INITIALIZER;

int value_globale = 1; // variable globale partagée

#define COUNT  5

…

pthread_mutex_lock(&globale_lock); // demande de verrouillage du mutex
…
pthread_mutex_unlock(&globale_lock); // déverrouillage du mutex
```

3) Écrire le programme threads.2b.c qui permet de corriger le problème de synchronisation de donnée en utilisant un mutex.

Séquence 3 : synchronisation de tâche

La synchronisation de tâche (ou de processus) cherche par exemple à empêcher des programmes d'exécuter la même portion de code en même temps, ou au contraire forcer l'exécution de deux parties de code en même temps. Dans la première hypothèse, le processus bloque l'accès au code en avant d'entrer dans la portion de code critique ou si cette section est en train d'être exécutée, se met en attente. Le processus libère l'accès en sortant de la partie du code. Ce mécanisme peut être implémenté de multiples manières.

Ces mécanismes sont par exemple la **barrière de synchronisation**.

L'implémentation des threads intègre la notion de variable de condition (de type **pthread_cond_t**), qui, associée à une variable normale et à un sémaphore, va permettre de synchroniser des activités sur les changements de valeur de la variable et les conditions satisfaites par la nouvelle valeur.

Une condition (abréviation pour variable-condition) est un mécanisme de synchronisation permettant à un thread de suspendre son exécution jusqu'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) et attendre la condition
- suspendre la condition jusqu'à ce qu'un autre thread signale la condition

Une variable de condition est associée à un ensemble d'attributs (de type pthread_condattr_t). La valeur prédéfinie pour les attributs de variable de condition est pthread_condattr_default. Les variables de type pthread_cond_t peuvent également être statiquement initialisées, en utilisant la constante PTHREAD_COND_INITIALIZER.

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

pthread_cond_signal relance l'un des threads attendant la variable condition cond. S'il n'existe aucun thread répondant à ce critère, 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_signal(pthread_cond_t *cond);
```

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

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

Déverrouiller le mutex et suspendre l'exécution sur la variable condition est effectué atomiquement. Donc, si tous les threads verrouillent le mutex avant de signaler la condition, il est garanti que la condition ne peut être signalée (et donc ignorée) entre le moment où un thread verrouille le mutex et le moment où il attend sur la variable condition.

pthread_cond_wait déverrouille atomiquement le mutex (comme pthread_unlock_mutex) et attend que la variable condition cond soit signalée. L'exécution du thread est suspendu et ne consomme pas de temps CPU jusqu'à ce que la variable condition soit signalée. Le mutex doit être verrouillé par le thread appelant à l'entrée de pthread_cond_wait. Avant de rendre la main au thread appelant, pthread_cond_wait reverrouille mutex (comme pthread_lock_mutex).

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

pthread_cond_timedwait déverrouille atomiquement mutex et attend sur cond, comme le fait pthread_cond_wait, cependant l'attente est bornée temporellement. Si cond n'a pas été signalée après la période spécifiée par abstime, le mutex mutex est reverrouillé et pthread_cond_timedwait rend la main avec l'erreur ETIMEDOUT.

```
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t
*mutex, const struct timespec *abstime);
```

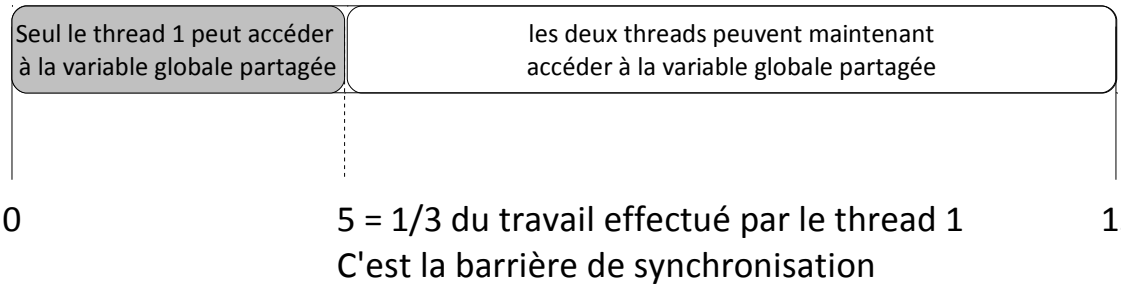
pthread_cond_destroy détruit une variable condition, libérant les ressources qu'elle possède. Aucun thread ne doit attendre sur la condition à l'entrée de pthread_cond_destroy.

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

Exemple d'utilisation d'une variable-condition :

```
pthread_mutex_t condition_lock  = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  condition_var   = PTHREAD_COND_INITIALIZER;
...
pthread_mutex_lock(&condition_lock);
pthread_cond_wait(&condition_var, &condition_lock);
pthread_mutex_unlock(&condition_lock);
...
pthread_mutex_lock(&condition_lock);
pthread_cond_signal(&condition_var);
pthread_mutex_unlock(&condition_lock);
```

En reprenant l'exemple précédent (increment/decrement), on va intégrer une variable-condition pour synchroniser les activités des deux tâches :

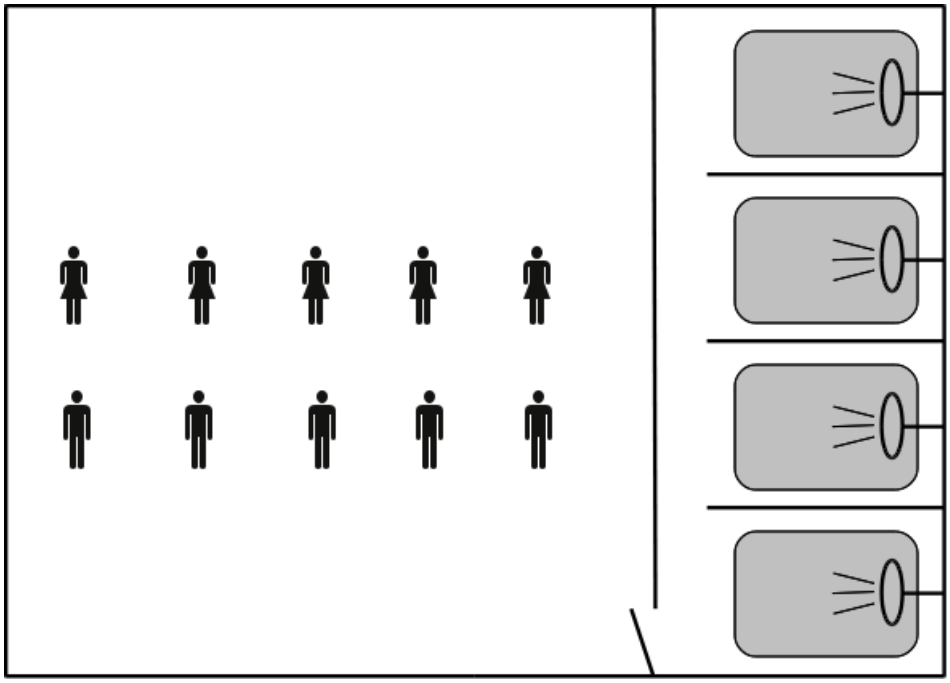


Le thread2 attendra donc avant de décrémenter la variable partagée que le thread1 ait réalisé au moins un tiers de ses boucles.

4) Écrire le programme threads.3.c qui permet de mettre en oeuvre une synchronisation de tâche (entre thread1 et thread 2) en utilisant une variable-condition.

Bonus : les douches

Les douches des garçons du gymnase sont en travaux. Pendant ce temps, les garçons et les filles utiliseront les douches des filles. Afin de ne choquer personne, un panneau est accroché à l'entrée des douches : libre, occupé par des filles ou occupé par des garçons. Seules des filles peuvent entrer dans les douches lorsque des filles s'y trouvent déjà et réciproquement pour les garçons.



Il faut créer un programme simulant un groupe de NB_PERSONNES composé de G garçons et de F filles attendant de prendre leur douche. Chaque personne sera simulée par un thread, et NB_CABINES cabines de douches seront disponibles.

On va tout d'abord définir les ressources à gérer :

```
// nombre de personnes
#define NB_PERSONNES 50

// nombre de cabines de douches
#define NB_CABINES 4

// etat pour la gestion d'une douche
#define ENTREE  1
#define SORTIE  2

// Les ressources pour gérer les douches
typedef struct
{
   int nb_filles;
   pthread_mutex_t filles_mutex;

   int nb_garcons;
   pthread_mutex_t garcons_mutex;

   int nb_cabines_libres;
   pthread_cond_t cabine_libre;
} RESSOURCES_DOUCHES;

// description d'une personne
typedef struct
{
   char prenom[16];
   char sexe;
   RESSOURCES_DOUCHES *res; // toutes les personnes partagent les
memes                                    ressources

   int heure_douche; // pour les calculs et statistiques
   int duree_douche;
   int duree_attente;
   int heure_fin_douche;
} PERSONNE;
```

Avant de commencer, il faudra initialiser ces ressources :

```
RESSOURCES_DOUCHES *initialiserRessources(RESSOURCES_DOUCHES *res)
{
   le mutex pour les filles
   le mutex pour les garcons
   la variable-condition pour les cabines
   le nombre de filles dans les douches
   le nombre de garcons dans les douches
   le nombre de cabines de libre
   retourner la structure initialisée
}
```

On va ensuite créer autant de threads qu'il y a de personnes. Mais on distinguera évidemment les garçons des filles.

Le code du thread d'un garçon sera le suivant :

```
void *garcon(void *param) {
   PERSONNE *p = (PERSONNE *)param;

   // il fait la queue
   printf("%s attend ...\n", p->prenom);
   doucher_garcon(p->res, ENTREE);

   // il prend sa douche
   p->duree_attente = time(NULL) - p->heure_douche;
   printf("  %s prend sa douche (%d secondes, attente %ds)\n", p-
>prenom, p->duree_douche, p->duree_attente);
   sleep(p->duree_douche);

   // il repart
   printf("    %s est propre !\n", p->prenom);
   doucher_garcon(p->res, SORTIE);

   printf("%s repart\n", p->prenom);
   p->heure_fin_douche = time(NULL);

   pthread_exit(NULL); return NULL;
}
```

Celui d'une fille sera identique :

```
void *fille(void *param);
```

La gestion d'accès aux douches est réalisée dans la fonction pour les garçons et pour les filles. Le pseudo-code pour les garçons est le suivant :

```
/* etat peut prendre soit la valeur ENTREE ou SORTIE */
void doucher_garcon(RESSOURCES_DOUCHES *res, int etat)
{
   SI un garçon entre pour se doucher ?
   ALORS
      on bloque les autres garçons
      TANT QUE il n'y a plus de cabines de libre ?
         attendre qu'une cabine soit disponible
      FINT TANT QUE
      un garçon de plus en train de se doucher
      SI il y a déjà un garçon dans les douches ?
      ALORS seul des garçons peuvent se doucher et on bloque les
filles
      FIN SI
      une cabine de libre en moins
      on libère les autres garçons
```

```
   SINON
      SI un garçon sort de la douche ?
      ALORS
         on bloque les autres garçons
         un garçon de moins en train de se doucher
         SI c'était le dernier garçon à se doucher ?
         ALORS on de-bloque les filles
         FIN SI
         une cabine de libre en plus
         on signale qu'une cabine est disponible
         on libère les autres garçons
      FIN SI
   FIN SI
}
```

Celui d'une fille peut être considéré comme symétrique :

```
void doucher_fille(RESSOURCES_DOUCHES *res, int etat);
```

Le programme principal est le suivant :

```
int main(void)
{
   PERSONNE p[NB_PERSONNES];
   pthread_t tid[NB_PERSONNES];
   RESSOURCES_DOUCHES res;
   int nbg = 0, nbf = 0;
   int heure_debut, heure_fin;
   int rcode;
   int i;

   srand(time(NULL));

   initialiserRessources(&res);

   printf("Gymnase : %d cabines de douches pour %d personnes\n",
NB_CABINES, NB_PERSONNES);

   heure_debut = time(NULL);
```

```
   // creation des filles et des garcons
   for(i = 0; i < NB_PERSONNES; i++)
   {
      // attendre, tout le monde ne se presente pas en meme temps
      sleep(rand() % 2);

      if(rand() < RAND_MAX/2)
      {
         p[i].res  = &res; // toutes les personnes partagent les memes
ressources
         p[i].sexe = 'G'; // je suis un garcon !
         sprintf(p[i].prenom, "ken%d", i);
         p[i].duree_douche = 3 + rand() % 5; // entre 3 et 7 secondes
pour se doucher
         p[i].heure_douche = time(NULL); // heure d'arrive pour
prendre la douche

         rcode = pthread_create(&tid[i], NULL, garcon, &p[i]);
         nbg++;
      }
      else
      {
         p[i].res  = &res; // toutes les personnes partagent les memes
ressources
         p[i].sexe = 'F'; // je suis une fille !
         sprintf(p[i].prenom, "barbie%d", i);
         p[i].duree_douche = 3 + rand() % 5; // entre 3 et 7 secondes
pour se doucher
         p[i].heure_douche = time(NULL); // heure d'arrive pour
prendre la douche

         rcode = pthread_create(&tid[i], NULL, fille, &p[i]);
         nbf++;
      }
   }

   // attendre
   for(i = 0; i < NB_PERSONNES; i++)
      rcode = pthread_join(tid[i], NULL);

   heure_fin = time(NULL);


   /* affiche quelques statistiques */
   /* … */
}
```

Exemple d'exécution :

```
$ ./threads_douche
...
Gymnase : 4 cabines de douches pour 50 personnes (30 garcons-20 filles
)
Duree totale : 68 secondes
Duree moyenne d'une douche : 4.72 s
Duree moyenne d'une douche pour les garcons : 5.00 s
Duree moyenne d'une douche pour les filles : 4.30 s
Duree d'attente moyenne : 18.48 s
Duree d'attente moyenne pour les garcons : 30.23 s
Duree d'attente moyenne pour les filles : 0.85 s
Premiere arrivee : h+0
Derniere arrivee : h+23
Plus courte attente : 0 s
Plus longue attente : 51 s
```

5) Compléter le code source du fichier threads_douche.c et valider son fonctionnement.