TP 8 : threads, processus,mémoire partagée

Exercice 1. Dentiste 1

Vous allez améliorer le fonctionnement du programme *dentiste* précédant en créant le programme **dentiste1** qui ajoutera la gestion d'une salle d'attente dont le **nombre de places est limité**.

Pour cela vous utiliserez une **variable globale** `en_attente` qui compte le nombre de personnes dans la salle et une constante `NB_PLACES` qui contient le nombre de sièges dans la salle. Initialement `NB_PLACES` aura la valeur **2**.

Chaque fois qu'un patient *arrive* dans la salle d'attente il consulte la variable `en_attente` et si elle a une valeur strictement inférieure à `NB_PLACES`, il affiche le nombre de places libres restantes (ex : "Patient : Il reste X place(s) libre(s)"), il incrémente `en_attente` et s'installe dans la salle.

Pour s'*installer* dans la salle, il effectue les mêmes opérations que dans *dentiste0* (c'est-à-dire : informe le dentiste qu'il arrive , attente du dentiste ,affichage d'un message d'arrivée).

Dans le cas où il n'y a plus de place libre il rentre chez lui en affichant "Patient : Plus de place, je rentre chez moi."

De son côté, le dentiste, chaque fois qu'il *reçoit* un patient ,décrémente `en_attente`. Le reste de son code reste inchangé.

Comme la variable `en_attente` est en accès concurrent (par le dentiste et les patients) il faut la placer en zone critique avec un troisième sémaphore de type **mutex**.

Dupliquer le programme *dentiste0.c* et renommez cette copie *dentiste1.c* puis complétez son code pour obtenir le comportement décrit ci-dessus.

Exemple d'exécution avec 2 places dans la salle d'attente :

```
$ ./dentiste1
Ouverture du cabinet.

> Arrivé d'un patient.
Patient : Il reste 1 place(s) libre(s).
Dentiste : je reçois un nouveau patient
Patient : Je rentre dans la piece et me fait soigner.

> Arrivé d'un patient.
Patient : Il reste 1 place(s) libre(s).

> Arrivé d'un patient.
Patient : Il reste 0 place(s) libre(s).

> Arrivé d'un patient.
```

```
Patient : Plus de place, je rentre chez moi.

etc
```

Exercice 2. Tri par fusion parallélisé

Voici un programme de tri par fusion (cf. *Wikipedia*) d'un tableau de 256 entiers générés aléatoirement (fonction `rand()`) en version séquentielle :

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>

#include <sys/types.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <time.h>
#include <semaphore.h>

int is_sorted(int n, int t[]) {
   int i;
   for (i=1; i < n; i++)
       if( t[i] < t[i-1] )
           return 1;
   return 0;
}


void merge(int n, int t1[], int t2[]) {
   int i=0, j=0, k=0;
   int t[2*n];

   while (i < n && j < n)
       t[k++] = (t1[i] <= t2[j])? t1[i++] : t2[j++];

   while (i < n)
       t[k++] = t1[i++];

   while (j < n)
       t[k++] = t2[j++];

   for (k=0; k < n; k++)
       t1[k] = t[k],
       t2[k] = t[k+n];
}

```

```c
void mergesort(int n, int t[]) {
   if (n == 1) return;

   mergesort(n/2, t);
   mergesort(n/2, t + n/2);
   merge(n/2, t, t+n/2);
}

int main() {

   int i;
   int n = 256;
    int t[n];

   for (i=0; i < n; i++)
       t[i] = rand()%1000;
   printf("Avant :\n");
   if (is_sorted(n, t))  printf("Pas trié !\n");
   else                  printf("Trié !\n");

   mergesort(n, t);
   printf("Aprés :\n");
   if (is_sorted(n, t)) printf("Pas trié!\n");
   else                  printf("Trié !\n");

}
```

a) Ecrivez une version **multi-threadée** synchronisée par des `phtread_join`.

Le principe sera le suivant :

La fonction threadée sera une version modifiée de `mergesort` dans laquelle les parties gauches et droites du tableau seront triées par appel récursif de cette même fonction. Les appels récursifs, pour les parties gauches et droites, seront effectués dans des **threads séparés**. Le passage de paramètres à la fonction `mergesort` se fera en utilisant une structure :

```c
typedef struct {
  int n; /* taille à trier */
  int * t; /* @ de debut */
} atrier;
```

La fusion des deux parties seront toujours effectuées par la fonction `merge` qui reste **identique à la version non threadée**.

b) Ecrivez une version parallélisée **multiprocessus**. Les processus seront générés par `fork()` et synchronisés par des **sémaphores anomymes**.

Un sémaphore anonyme doit être placé en mémoire partagée (dans **un segment de mémoire partagé anonyme**) pour pouvoir être utilisé par les processus père et fils. On utilisera la fonction `mmap` de la

façon suivante :

```
1  sem_t * sem = mmap(NULL, sizeof(sem_t), PROT_READ | PROT_WRITE,
       MAP_SHARED | MAP_ANONYMOUS, -1, 0);
```

Le tableau sera placé dans un **segment de mémoire partagée** qui utilisera les fonctions suivantes :

Includes

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
```

Fonctions

```
int shm_open(const char *name, int oflag, mode_t mode);
int shm_unlink(const char *name);
int ftruncate(int fildes, off_t length);
void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t
 off);
```

Dans la fonction **mergesort**, le processus *père* effectuera le tri de la partie droite et le *fils* fera le tri de la partie gauche ainsi que la fusion des deux parties par `merge` (qui reste **inchangée**). Cette fusion doit se faire uniquement quand les deux parties seront effectivement terminées : cette synchronisation se fera grâce à un **sémaphore anonyme**.

c) Réécrivez la version **multiprocessus** en utilisant un *segment de mémoire partagée* **anonyme** (donc sans `shm_open`).

Remarques :

- On peut voir la mémoire partagée par *ls -l /dev/shm*
- Options de compilation : *-lrt -lpthread*