

TP 5 : Sémaphores, Threads



Exercice 1. Dentiste

Vous allez écrire une série de programmes en C qui simulent le fonctionnement d'un cabinet de dentiste avec sa salle d'attente. Le dentiste est simulé par un **thread** et il soigne un patient à la fois. Chaque patient sera simulé par un thread (il y a donc autant de **threads** créés que de patients arrivant au cabinet dentaire).

Vous utiliserez comme base le squelette donné sur Moodle.

1. Programme dentiste0

Dans le main de ce programme, vous allez créer un thread pour le dentiste puis, toujours dans le main, dans une boucle « infinie » attendre l'appui sur une touche, chaque appui déclenche la création d'un patient arrivant au cabinet dentaire. C'est-à-dire que vous allez créer un thread simulant ce patient (voir **squelette_dentiste.c**)

Le dentiste ne traitant qu'un patient à la fois il sera associé à un sémaphore binaire « dentiste » : si la valeur est 1 le dentiste s'occupe d'un patient, 0 sinon.

Les patients sont associés à un sémaphore qui servira à la synchronisation avec le dentiste.

Le fonctionnement du *thread dentiste* est le suivant :

- Affichage d'un message "Ouverture du cabinet."
- Dans une boucle infinie :
 - attente d'un patient : opération P() sur le sémaphore patients
 - réception du patient : opération V() sur le sémaphore dentiste
 - affichage du message "Dentiste : je reçois un nouveau patient"
 - le dentiste soigne le patient. Ce sera simulé par un sleep de 2 secondes.
 - affichage du message de fin du soin : "Dentiste : Au suivant !"

Le fonctionnement d'un *thread patient* est le suivant :

- informe le dentiste qu'il arrive : opération V() sur le sémaphore patients
- attente du dentiste : opération P() sur le sémaphore dentiste
- Affichage d'un message d'arrivée : "Patient : Je rentre dans la piece et me fait soigner"

Réfléchissez aux valeurs initiales des sémaphores.

Exemple d'exécution avec 2 appuis sur la touche entrée :

```
1      $ ./dentiste0
2
3      Ouverture du cabinet.
4
5      > Arrivé d'un patient.
6
7      Dentiste : je reçois un nouveau patient
8      Patient  : Je rentre dans la piece et me fait soigner.
9
10     > Arrivé d'un patient.
11
12     Dentiste : Au suivant !
13     Dentiste : je reçois un nouveau patient
14     Patient  : Je rentre dans la piece et me fait soigner.
15     Dentiste : Au suivant !
```

Exercice 2. 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 :

```
1 $ ./dentiste1
2 Ouverture du cabinet.
3
4 > Arrivé d'un patient.
5 Patient : Il reste 1 place(s) libre(s).
6 Dentiste : je reçois un nouveau patient
7 Patient : Je rentre dans la piece et me fait soigner.
8
9 > Arrivé d'un patient.
10 Patient : Il reste 1 place(s) libre(s).
11
12 > Arrivé d'un patient.
13 Patient : Il reste 0 place(s) libre(s).
14
15 > Arrivé d'un patient.
16 Patient : Plus de place, je rentre chez moi.
17
18 etc
```

Exercice 3. 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 :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <fcntl.h>
6
7 #include <sys/types.h>
8 #include <sys/shm.h>
9 #include <sys/stat.h>
10 #include <sys/mman.h>
11 #include <time.h>
12 #include <semaphore.h>
13
14 int is_sorted(int n, int t[]) {
15     int i;
16     for (i=1; i < n; i++)
17         if( t[i] < t[i-1] )
18             return 1;
19     return 0;
20 }
21
22
23 void merge(int n, int t1[], int t2[]) {
```

```
24     int i=0, j=0, k=0;
25     int t[2*n];
26
27     while (i < n && j < n)
28         t[k++] = (t1[i] <= t2[j])? t1[i++] : t2[j++];
29
30     while (i < n)
31         t[k++] = t1[i++];
32
33     while (j < n)
34         t[k++] = t2[j++];
35
36     for (k=0; k < n; k++)
37         t1[k] = t[k],
38         t2[k] = t[k+n];
39 }
40
41 void mergesort(int n, int t[]) {
42     if (n == 1) return;
43
44     mergesort(n/2, t);
45     mergesort(n/2, t + n/2);
46     merge(n/2, t, t+n/2);
47 }
48
49 int main() {
50
51     int i;
52     int n = 256;
53     int t[n];
54
55     for (i=0; i < n; i++)
56         t[i] = rand()%1000;
57     printf("Avant :\n");
58     if (is_sorted(n, t)) printf("Pas trié !\n");
59     else printf("Trié !\n");
60
61     mergesort(n, t);
62     printf("Après :\n");
63     if (is_sorted(n, t)) printf("Pas trié!\n");
64     else printf("Trié !\n");
65
66 }
```

a) Ecrivez une version **multi-threadée** synchronisée par des `pthread_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 :

```
1 typedef struct {
2     int n; /* taille à trier */
3     int * t; /* @ de debut */
4 } 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 anonymes**.

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`