Architecture et Système 2023-2024

Département Informatique, ENS Paris Saclay

TP 12 : Threads and semaphores : Second Episode

1 Basics

How to compile :

```
gcc -pthread program.c -o program
```

How to declare, create and join threads :

```
pthread_t t1;
```

pthread_create takes as arguments a pointer to the thread ID, attributes to set the properties of thread, a function pointer to the function that thread will run in parallel on start (this function should accept a void * and return void * too) and arguments to be passed to the function. For instance :

```
pthread_create(&t1, NULL, &function, NULL);
```

Don't forget to join the thread afterward :

```
pthread_join(t1, NULL);
```

The second argument is a pointer to store the return value from the thread. pthread_create and pthread_join both return 0 if the thread has been created/joined successfully.

How to declare, initialize, destroy and use semaphores :

```
sem_t s1;
sem_init(&s1, 0, n);
sem_destroy(&s1);
```

sem_open takes as second argument 0 because the semaphore is shared between threads and not processes. The argument n means that sem_wait can be called n times until the semaphore is locked. For intance if $n = 0$ it means a sem_post must be used first. sem_open and sem_destroy both return 0 if the semaphore has been opened/destroyed successfully.

```
sem_wait(&s1);
\\ critical code section
sem_post(&s1);;
```

sem_post frees the semaphore (i.e., increases n), sem_wait waits for the semaphore to be free ($n > 0$) and decreases n.

See `sem.c` as an example.

How to declare, initialize, destroy and use mutexes :

```
pthread_mutex_t m1;
pthread_mutex_init(&m1, NULL);
pthread_mutex_destroy(&m1);
pthread_mutex_lock(&m1);
pthread_mutex_unlock(&m1);
```

2 Julia set by inverse iteration

We can approximate the Julia set of a complex function by interating its inverse. As an example, we can study the function :

$$Q_c = \begin{cases} \mathbb{C} & \to \mathbb{C} \\ z & \mapsto z^2 + c \end{cases}$$

With $c \in \mathbb{C}$.
We define the reverse orbit as $\{f_c^{-n}(z_0); n \in \mathbb{N}\}$. To compute it we can se that if $z^2+c = w$, then $z = \rho \exp(i\theta)$ with $\rho = \sqrt{|w-c|}$, and $\theta = \frac{\vartheta}{2} + \delta\pi$ with $\delta \in \{0, 1\}$, where

$$\vartheta = \arctan(\Im(w-c)/\Re(w-c)) + \begin{cases} 0 & \text{si } \Re(w-c) > 0 \\ \pi & \text{sinon} \end{cases}$$

The initial point w_0 does not matter. We want to create several threads computing orbits, each orbit having a different starting point.

We will write the points either on the standard output or in a file, with each line being of the form `x y`. If the point are in a file denoted `julia.dat`, we can output the graph with the command `gnuplot julia.p` assuming that `julia.p` is the following script :

```
set terminal png size 500,500
set output 'julia.png'
set title 'Julia set'
plot 'julia.dat'
```

Don't forget to include `math.h` and `complex.h` and to compile with `-lm` at *the end of the command line*. the following command should work `gcc julia.c -lpthread -lm`.

Try to draw the Julia set for the following values of c : $c \in \{-1, -0.4-0.6i, -1.5, -i, -0.8+0.4i, 0.5, 3, 1+i, 2\}$

3 Datetime Server : Simple mutual exclusion

We want to set-up a client-server architecture (We won't use th socket API, we will just do a thread for the server and some threads for the clients) in which the server is tasked to give the date each time a client requests it.

Therefore, the server runs continuously waiting for a request from the clients. Once it receives a resquest, it sends back a the current time and the date as a string to the client. After this, the server is ready to answer another request.

The client thread will run 50 times a function that :
— sends a request to the server
— recieves the date sent back by the server,
— displays it along with the request number (these should stay ordered).

a) Which information is shared between the threads ? Which means of communication will they use to transmit data ?
b) How to synchronize the various threads ?
c) Implement the solution with more than one client.

4 Peterson's algorithm

We will try to implement the Peterson's algoritm for mutual exclusion (You can find useful details on Wikipedia). The gobal idea is to use 3 variables `f0`, `f1` et `turn` to deal with the entrance of the « critical section ». The « critical section »being the section that contains the code where we want to avoid concurents access. Show that this algorithm ensures mutual exclusion.

a) By looking into the Wikipedia page : write a `process0` and `process1` function which do similar things, but use Peterson's algorithm to avoid concurrent execution of a certain part of their code.
b) Adapt the code of `process0` and `process1` to write a function `process` which allows the generic invocation of a thread (meaning when creating two different threads with it, they will enforce mutual axclusion).
c) Change your code to accomodate for 4 simultaneous processes.
d) Same questions for Dekker's algorithm.

5 Binary and Semaphores (Bonus)

We want to have n threads that each display the corresponding number from 0 to $n-1$. Howerever, they should synchronize to get them in order. In order to do this, they should use semaphores. During class, you saw a method which uses $n-1$ semaphores but we can do it in $\log_2(n)$ semaphores.
Hint : The semaphores should represent the binary digits of the thread whose turn it is
Hint 2 : Handle endianess with care