LSV / ENS de Cachan 2014/2015
Architecture et Système Feuille de TP n° 9

TP9

The course homepage is here:

http://www.lsv.ens-cachan.fr/~schwoon/enseignement/systemes/ws1415/.

You will find the slides and demonstration programs from the course and some other files for the exercise there.

Details of shell commands and C functions can be obtained by using the `man` command.

1 Semaphores

Remember that threads run in the same address space as the process creating them and in particular share globally defined variables. Now consider the following program (`counter.c`):

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

#define MAX 100000
int counter = 0;

void* count(void* data) {
  int i;
  for (i = 0; i < MAX; i++) counter++;
}

int main () {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, count, NULL);  // create first thread
  pthread_create(&t2, NULL, count, NULL);  // create second thread
  pthread_join(t1, NULL);  // wait for first thread
  pthread_join(t2, NULL);  // wait for second thread
  printf("Counter: %d\n", counter);
}
```

(a) What output do you expect from this code?

(b) What is the reason for unexpected values?

(c) What is the range of values that could potentially be output?

A concept to overcome those problems are *semaphores*. A semaphore is a data structure used for synchronization and dealing with limited ressources that are accessed by multiple threads. In `C`, the semaphore interface is defined in the header file `semaphore.h`. The most important functions that we will use in this class are the following:

- `int sem_init(sem_t *sem, int pshared, unsigned int value);`
  This function initializes `sem` with a given capacity representing the number of shared ressources available. Under Linux, `pshared` should always be `NULL`.
- `int sem_wait(sem_t *sem);`
  If the semaphore has currently zero capacity this function blocks. Otherwise, it decrements the capacity of the semaphore by one and returns.
- `int sem_post(sem_t *sem);`
  Increments the capacity of the semaphore and consequently unlocks it.

Read the `man` pages of the aforementioned functions in order to familiarize yourself with them.

(d) Use semaphores to fix the program `counter.c`.

Finally, consider the following program (`order.c`):

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

void* first(void* data)  { printf("First\n");  }
void* second(void* data) { printf("Second\n"); }
void* third(void* data)  { printf("Third\n");  }

int main () {
  pthread_t t1, t2, t3;

  pthread_create(&t3, NULL, third, NULL);
  pthread_create(&t2, NULL, second, NULL);
  pthread_create(&t1, NULL, first, NULL);

  // wait for all threads
  pthread_join(t1, NULL);  pthread_join(t2, NULL); pthread_join(t3, NULL);
}
```

Your task is make sure that after executing the compiled program it always outputs:

```
  First
  Second
  Third
```

(e) Write a C program that spawns two threads. The first thread outputs all even numbers up to 100, the second thread outputs all odd numbers. Use semapores to make sure that the numbers are printed in order.

2 Producer/consumer

A typical problem in concurrent programming is the producer/consumer problem. One process generates data, while another process consumes them. Consider the program `procon.c`. Here, the producer reads characters from a file and sends them to a common buffer shared with the consumer, which prints the characters. However, there is no synchronisation in place between the two threads.

Change the program so that it outputs the contents of the file correctly.

3 Mandelbrot

Let c be a complex number. We regard the associated series $z_0 := 0$ and $z_{n+1} := z_n^2 + c$. The Mandelbrot set is the set of values for c such that the series of z_n remains bounded. It turns out that this is the case iff z_n never exceeds a circle of radius 2 around the origin. If the series does exceed this radius, let $m(c)$ be the smallest index n for which z_n does so.

A popular application is to use $m(c)$ for drawing nice images. On the course webpage you will find such a program. Your task is to speed it up by making use of multiple threads that can run in parallel on a multi-core CPU (use `cat /proc/cpuinfo` to find more about the CPU on your machine). Each thread can be instructed to compute one slice of the picture.

Note: Compile the program using `make`.

Note (2): This exercise does not require semaphores.

Man pages: pthreads(7), pthread_create(3), pthread_join(3), see also the example programs from the course.