Lab Session 3: Threaded Programming
24296 - Sistemes Operatius

1 Introduction

The objective of this lab session is to understand threads and be able to program a multi-threaded application. You will need to deliver a compressed file with:

- A document answering the questions of the lab session and reporting benchmarking.
- The code file "mandelbrot_threaded.c". The code must compile, if not you will not be evaluated.
- The document is mandatory, is the first thing that will be evaluated after checking that the code compiles. If you don't submit a document your lab session will not be evaluated.

2 Mandelbrot

We have provided the file mandelbrot.c which generates the mandelbrot set image.tga. Compile it and test it: you should see the two files image.tga and image_scrambled.tga being created, which you can download (if using repl.it) and open with the file explorer. Now look at the code and understand carefully the following definitions and helper functions of the code:

```
(1)   void generate_mandelbrot(unsigned char *p, int width, int height);
(2)   int compute_iter(int i, int j, int width, int height);
(3)   void write_tga(char* fname, unsigned char *pixels, int width, int height)
(4)   int tga_write_header(int fd, int width, int height);
(5)   void interchange(int si, int sj, int ti, int tj, unsigned char *p, int width, int height);
(6)   #define R 4  // Constant indicating the image divisions RxR
(7)   enum { width=1024, height=1024 };   // image size constants width x height
```

1. Generates the mandelbrot color image and writes it to the already allocated memory pointer "p" of dimensions width*height*3. The 3 additional so-called channels, correspond to the RGB color code, having a byte for every color red, green), blue.

2. Computes the value (from 0..255) of the mandelbrot set at position i,j given the image size. For curiosity this value is linked to the convergence or divergence of a complex number being squared and summed a constant.

3. Creates and writes a tga image file with the indicated name, pixels pointer and size. The tga image format is a very basic and raw format which consists of a fixed header of 18 bytes followed by the pixels of the image as unsigned chars (bytes). If dealing with an RGB image we have width*height*3 bytes.

4. Writes a header of an image of size width x height to the given file descriptor (fd).

5. Interchanges two regions of an image (pointer p) each indicated by indices si, sj (region 1) and ti, tj (region 2). It assumes the image is divided in RxR regions (constant being defined in **6**), so, si for example can range between 0 and R-1.

6. The declaration defines the two constants for the size of the image as an "enum" type. Being constants, they can be used in the declaration of arrays.

3 Threaded Mandelbrot

Now we are going to make a threaded version of the previous program (copy file into mandelbrot_threaded.c that you will need to deliver). We have also provided the collaborative sum thread example seen in class so that you can compile it and test it as well. Follow the steps below one by one and compile often so that you don't accumulate errors with:

```
gcc mandelbrot_threaded.c myutils.c -pthread -o main
```

1. Create R*R threads. Remember R is the constant which divides the image in regions.
2. Make each thread compute its corresponding part of the mandelbrot set. Take inspiration of helper function (5) "interchange" to know the target memory region of each thread.
3. Wait for all threads to finish in the main thread and write the result as before. Benchmark the total time spent using the functions startTimer(0), endTimer(0) provided in myutils.c. Report results and the difference with the single thread version in the document.

4 Scrambled Mandelbrot

We are now going to make a threaded version of the scrambling of the image. We are going to perform a total of 1000 scrambles (interchanges of image regions). You will be asked to benchmark results and report them in the document. If you find hard to see improvements just increase the size of the image and the number of interchanges.

1. Create N threads each one is gonna do 1000 / N interchanges and try the result without synchronization. Benchmark the total time spent using the functions startTimer(0), endTimer(0) provided in myutils.c. Report problems in the document.
2. First implement the single lock solution to the previous problem. Benchmark the results and report them in the document.
3. Now implement a lock for each region. When doing an interchange each thread will need to lock both source and target regions. Be careful of deadlocks. To avoid problems, ask for the lock in the same order (using the order imposed by the indices). Benchmark the results and report them in the document.
4. Now we are going to implement a synchronization solution using monitors. We have provided in myutils.c helper functions to perform locks using monitors with the struct type monitor_lock. Create one monitor_lock per region and use the provided lock_monitors and unlock_monitors to synchronize access to resources.

Here follow the pthread declarations you will need. The last ones, regarding an implementation of locks using monitors will be explained in class and are provided in myutils.h and .c):

```
void* fthread(void* params) { return NULL; }  // thread code function definition

int pthread_create(pthread_t *thread, NULL, fthread, params);
int pthread_join(pthread_t thread, NULL);

pthread_mutex_t lock;
pthread_mutex_lock(&lock);
pthread_mutex_unlock(&lock);

typedef struct struct_monitor_lock {
 bool bInUse;                // the ressource is being used?
 pthread_mutex_t lock;       // lock to use/modify vars
 pthread_cond_t  cond_free;  // condition for waiters of the ressource
} monitor_lock;

void mon_lock_init(monitor_lock* ml);
void mon_lock(monitor_lock* ml);
void mon_unlock(monitor_lock* ml);
```