Lab Introduction to Concurrency

Out: November 26-27 2012

1 Introduction

Previously in your computer science career, you have written primarily *sequential* programs: programs that execute instructions one-by-one in a specified order. You have also written programs that perform distinct tasks simultaneously using `fork()`. The first of these programming paradigms does not permit parallel execution of distinct tasks, but maintains all program memory in the only thread of execution; the second of these paradigms does allow simultaneous execution of tasks, but does not enable memory to be shared between different threads of execution. But with *concurrent programming* techniques, a program can both execute distinct tasks in parallel as well as share memory between those threads of execution.

2 Fractals

As an exercise in concurrent programming, you'll be parallelizing a program which generates a representation of the Mandelbrot set and saves it in a *.png* image file. We have provided code that generates the fractal with a single line of execution; you will be editing the program to perform this computation concurrently.

The program you will be working with consists of code from the following files:

- *image.h* - the *image.h* file contains struct definitions and method headers used to generate fractal data.
  - `color_t`

    This struct is used to represent the color of each pixel in the fractal image. Image colors are typically represented in a three-dimensional color space, where the dimensions represent the amounts of red, green, and blue light that produce the color. A `color_t` represents those coordinates as `unsigned char` values, which range from 0 to 255.
  - `int save_image_data(char *file_name, color_t *image_data, int width, int height)`

    This function saves the color data stored in the `image_data` array into the file `file_name`.
  - `void generate_fractal_region(color_t *image_data, float width, float height, float start_y, unsigned int rows)`

    This function generates color data for a particular subset of the pixels of the fractal and stores that data in the `image_data` array. Out of `height` rows of pixels, this function generates data for a number of rows given by the value of `rows`, beginning with row `start_y`.
- *image.c* - the *image.c* file contains definitions for the functions declared in *image.h*. You should not need to be familiar with the contents of this file to complete this lab.

- *colors.h* - the *colors.h* file contains declarations of three functions that generate a color channel value (an integer between 0 and 255). These functions are used in *image.c* to generate color data for each pixel of the fractal.
- *colors.c* - the *colors.c* file contains definitions of the color generation functions declared in *colors.h*. The colors of the fractal generated by the program depend on these functions - change them to change program output.
- *fractal.h* - the *fractal.h* file contains a declaration for the `generate_fractal()` function.
- *fractal.c* - the *fractal.c* file contains the definition for the `generate_fractal()` function, generating the fractal using a single line of execution.
- *main.c* - the *main.c* file contains the program `main()` function. This function parses the `argv` array for program arguments and calls `generate_fractal()`, the function you will be filling in (see below).

You are additionally provided with the files *fractal_forked.c* and *fractal_threaded.c*, which provide empty definitions for `generate_fractal()`.

2.1 Running the Program

After you've compiled the program by running `make`, you can run the executable binary `fractal` with the following options:

- `-f <file>` and `--file <file>` change the file in which the fractal is saved.
- `-d <width> <height>` or `--dimensions <width> <height>` set the dimensions of the output image.

By default, `fractal` will store a 2000×2000 image in the file *out.png*. View this image using your favorite image viewer or editor; if you don't have one, run `okular out.png` to view the image.

2.2 Performance

In its present state, the `generate_fractal()` function of *fractal.c* does not utilize any form of concurrent execution. If you make the program and run `time ./fractal`, you'll notice that it takes a few seconds to execute. Performance of this program can be improved by distributing the task of fractal data generation amongst several *subprocesses* or *threads*.

Your task for this lab is to re-implement the `generate_fractal()` method of the fractals program in the files *fractal_forked.c* and *fractal_threaded.c* using concurrent programming techniques to improve program performance.

3 Shared Memory Using `fork` and `mmap`

- `pid_t fork()`

- `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)`
- `int munmap(void *addr, size_t length)`
- `int waitpid(pid_t pid, int *status, int options)`

Your first task is to parallelize the fractal program by using `fork()`, `mmap()`, and `munmap()`.

As you've seen previously, `fork()` is a system call which duplicates the calling process; the resulting child process receives a copy of its parent process' memory and stack, and begins execution from the point where `fork()` returns. Because memory is not ordinarily shared between the parent and child processes, filling in part of the `color_t` array in each child process will uselessly generate an incomplete image in each child process. To successfully generate the desired image, some mechanism by which the processes can share memory is needed.

Virtual memory provides such a mechanism with the system call `mmap()`. This system call creates a new *mapping* in a process' virtual address space - that is, it creates a new region of memory for use by the process. `mmap()` creates a new mapping of size `length` bytes and returns a pointer to the location of that new mapping, which is created at a virtual page boundary near `addr` (if `addr` is not `NULL`). The new mapping contains the contents of the open file given by the file descriptor `fd`[1], beginning from byte `offset` of that file. `prot` and `flags` are bit vectors which determine characteristics of the mapping. In particular, `prot` determines memory protection:

- `PROT_READ` indicates that the mapping may be read from.
- `PROT_WRITE` indicates that the mapping may be written to.
- `PROT_EXEC` indicates that the mapping may be executed.
- `PROT_NONE` indicates that the mapping may not be accessed.

`flags` determines behavior of the mapping. For example:

- `MAP_SHARED` indicates that updates to the mapping should be visible to other processes that map the same file. In addition, writes to the mapping will write back to that file.
- `MAP_PRIVATE` indicates that updates to the mapping should not be visible to other processes that map the same file, and that changes made in the mapping should not be written back to the file.

There are many other flags as well, but of particular interest is the flag `MAP_ANONYMOUS`. This flag indicates that the new mapping should not be backed by any file, causing the `fd` and `offset` arguments to be ignored. Instead the new mapping points to an anonymous file whose bytes are initialized to zero. If used in conjunction with the flag `MAP_SHARED`, `MAP_ANONYMOUS` sets up a convenient region of shared memory between processes; this is exactly what is needed here.

Your program needs to read to and write from an anonymous shared memory region, so you'll want to use protections `PROT_READ` and `PROT_WRITE` and the flags `MAP_ANONYMOUS` and `MAP_SHARED`. It doesn't matter where the mapping is located, so `addr` can be `NULL`.

[1] Unless you provide it with the flag `MAP_ANONYMOUS`, which you will be doing (see below).

Note that `mmap()` returns `(void *) -1` if an error occurs.

Once you've set up the shared memory region and `fork()`ed off each child process, the parent must wait until each child has completed its task before it saves the fractal data. To do this, you'll use the `waitpid()` system call to wait for each child process. To do so, you'll need to save the process ID of each child process.

Once the memory mapping is no longer needed, the system call `munmap()` cleans up and deletes the mapping. Call this function after you've saved the fractal data. You should also call this function in each child process before exiting — cleaning up before exiting is always good practice.

Task: use `fork()`, `mmap()`, and `munmap()` to edit *fractal_forked.c* so that work done to generate the fractal is divided amongst several child processes. This functionality should be abstracted to an arbitrary number of child processes; use the `workers` argument to `generate_fractal()`. By default, this argument is 4, but this value can be changed by passing the flag `-n <workers>` or `--workers <workers>` as an argument to the fractals program. You need not handle the case where the height of the generated image is not divisible by the number of child processes. You can build the fractals program with the command `make forked`, which includes *fractal_forked.c* instead of *fractal.c*. Doing so creates the executable *fractal_forked*.

4 Shared Memory Using Threads

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg)`
- `int pthread_join(pthread_t thread, void **retval)`

Using `mmap()` is certainly a valid way to share memory between a parent process and its child processes, but using it in this way requires some overhead. Any data generated outside the mapped region must be copied into the mapped region, for example, and there is also additional overhead from the operating system in providing each child with its own memory space.

The above issue is solved with *threads*. A thread is similar to a process in that it provides a unique line of execution; a thread is unlike a process in that it shares its virtual memory and state with its parent process. Because of this, threads are an invaluable part of concurrent programming.

The library you will be using for all of your threaded programming needs is the Posix thread library, *pthread.h*. Of primary concern are the functions `pthread_create()`, which creates and starts execution of a thread, and `pthread_join()`, which waits for a thread to finish executing and saves its return value in `retval`.

`pthread_create()` creates and executes a new thread, storing that thread in the argument `thread`. `attr` determines the attributes of this new thread; if `NULL` is passed to `attr`, the created thread is given the default thread attributes, which are sufficient for this lab. `start_routine()` is a pointer to the function which the new thread will execute; `arg` is the argument to that function. `pthread_create()` returns 0 if it is successful and an error number otherwise.

The signature of `start_routine()`, however, presents a problem: `generate_fractal_region()` requires several arguments, so it cannot be passed to `pthread_create()`. To work around this, you must define a `struct` which wraps each of the arguments to `generate_fractal_region()` into a single argument. Within *fractal_threaded.c* is the function definition `gen_fractal_region()`, which

has the signature required by `start_routine()`; edit this function so that it unpacks its argument struct and calls `generate_fractal_region()`.

Task: using threads, edit *fractal_threaded.c* so that work done to generate the fractal is divided amongst several threads. This functionality should be abstracted to an arbitrary number of threads; as with *fractal_forked.c*, the `workers` variable declared at the top of `main()` determines this value. You can also specify this value as an argument with the flag `-n <workers>` or `--workers <workers>`. You need not handle the case where the height of the generated image is not divisible by the number of threads. You can build the fractal program with the command `make threaded`, which includes *fractal_threaded.c* instead of *fractal.c*.

5 Comparison

Once you've finished implementing both concurrent programming paradigms, compare their performance by timing (using the command `time`) each paradigm with a different number of worker subprocesses or threads.[2] Does either paradigm seem to have a clear advantage in terms of performance?

6 Getting Checked Off

Once you've completed the lab, call a TA over and have them inspect your work. If you do not complete the lab during your lab session, you can get credit for it by completing it on your own time and bringing it to TA hours before next week's lab. Remember to read the course missive for information about course requirements and policies regarding labs and assignments.

7 Customizing the Fractal

Once you've finished the lab, you can feel free to customize the colors of the fractal generated by the program. *colors.c* contains the following functions:

- `int generate_red(float ratio)`
- `int generate_green(float ratio)`
- `int generate_blue(float ratio)`

These functions are used to determine the color of each pixel of the fractal image. Their argument, `ratio`, is a floating-point number between 0 and 1 inclusive. They return an integer between 0 and 255 inclusive. You can edit the bodies of these functions to customize the fractal program. Have fun!

[2] Make sure that the number of workers divides the height of the image so that the image is generated correctly. The divisors of 2000, the default height of the image, are 1, 2, 4, 5, 8, 10, 16, 20, 25, 40, 50, 80, 100, 125, 200, 250, 400, 500, 1000, and 2000.