Lab 11 - Concurrency 1

Due: December 3, 2017 at 4:00pm

1 Introduction

In this lab, you will be creating concurrent programs using threads. There are two parts to this assignment. The *fractals* directory will be used for the first section. The *linkedlist* directory will be used for the second part of the lab.

Install the stencil code for the lab by running

```
cs0330_install lab11
```

2 Fractals

The first part of this assignment is to repeat the fractals part of the virtual memory lab, but by using threads rather than **`fork()`** and **`mmap()`**.

Using **`mmap()`** is a valid way to share memory between different units of execution, but using it in this way requires some overhead. For example, any data generated outside the mapped region must be copied into the region. There is also additional overhead from the operating system in providing each child with its own memory space.

This can be 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*.

There are two functions of primary concern:

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

  This creates and executes a new thread, storing that thread in the argument **`thread`**.

- `pthread_join(pthread_t thread, void **retval)`

  This waits for the given thread to finish and saves its return value in **`retval`**.

For **`pthread_create()`**, **`attr`** determines the attributes of this new thread; if **`NULL`** is passed for **`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()`**.

2.1 Assignment

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 fractal_threaded`**, which includes *fractal_threaded.c* instead of *fractal.c*.

3 Concurrent Linked List

Running programs with multiple threads can serve two different purposes:

- *performance*: Using multiple threads can increase performance in a program by allowing other threads to run while one thread sleeps, which is often the case with programs that have to access the network or disk. On a computer with multiple processors, using multiple threads can help take advantage of that.

- *abstraction*: Using multiple threads in a program can also serve as a convenient abstraction. For example, a server that has to deal with multiple clients may want to have a different thread for each client. This way, each thread can operate independently, dealing with only one client at a time.

The issue with multithreaded programs, though, is that any data structures that are shared between the threads must be modified carefully, since each thread could be modifying the same element at the same time. As discussed in class, mutual exclusion is one way of solving this. In this assignment, we have provided a simple implementation of a linked list, along with a program that launches a number of threads that modify this list concurrently. The linked list implementation currently has no measures to provide thread-safety. Your task is to modify this code in order to make it safe for multithreading.

Two general approaches to making data structures thread-safe with mutual exclusion are *coarse-grained locking* and *fine-grained locking*. Coarse-grained locking involves locking the entire data structure with one lock, so that only one thread can access the data structure at any time. Fine-grained locking involves locking each component of the data structure separately. This approach allows multiple threads to access the data structure at the same time. In this lab, you will be implementing fine-grained locking, since implementing coarse-grained locking is fairly trivial in this case.

3.1 Assignment

In this part of the assignment, you will be creating a concurrent **sorted** linked list. There is an implementation of a sorted linked list in *linkedlist.c*, and you will be modifying this code to be thread-safe.

You will need to modify the **list_ele_t** struct to make the elements of your linked list thread-safe. In the stencil, the **list_ele_t** gives every element a value and a pointer to the next element in the list. Consider what you might add to the struct definition to be able to implement fine-grained locking! (Note: if you change the struct definition, you will also need to change the initialization of each element)

The functions in this file are:

- **`main()`**: Takes care of reading command line input and launching threads.

- **`randomListManip()`**: Randomly inserts or removes values from the linked list. The method's signature reflects the fact that it is being used in **`pthread_create`** to launch a new thread. The given *Makefile* generates a binary called *random* that uses this method.

- **`seqListManip()`**: Inserts values into the linked list. Running this method in multiple threads concurrently makes the effects of unsafe multithreaded programming quite apparent. Since each thread only adds elements to the list and never removes elements, anytime this method is run concurrently we can calculate the number of elements that should be in the list at the end of the program. The method's signature reflects the fact that it is being used in **`pthread_create`** to launch a new thread. The given *Makefile* generates a binary called *sequential* that uses this method.

- **`search()`**: Searches through the linked list for a node with the given value.

- **`insertList()`**: Inserts an element into the linked list if it is not already in the linked list.

- **`deleteList()`**: Deletes an element from the linked list if it exists in the linked list.

Running **`make`** using the provided *Makefile* creates two binaries: *sequential* and *random*. The former binary gives the **`seqListManip`** function as an argument to **`pthread_create()`**, and the latter does the same with **`randomListManip()`** instead. However, there is only one source file you will have to modify: *linkedlist.c*. The different binaries are generated by the macro surrounding the line that makes calls to **`pthread_create()`**, which simply swaps which function is being given to the new threads to run.

3.2 Implementation Details

If you were implementing coarse-grained locking, it would be enough to modify the **`seqListManip()`** or **`randomListManip()`** functions. However, since you are implementing fine-grained locking, you will have to modify the **`search()`**, **`insertList()`**, and **`deleteList()`** functions, as well as the definition of the list node struct. You will want to lock each node in the linked list separately, so that only one thread at a time can access the fields of any node in the linked list.

3.3 Tips: Using **`pthreads`**

You will probably want to use or learn more about the following pthread library functions in your implementation. Use the **`man`** command to find out more about them.

- pthread
    - `pthread_create()`

    - `pthread_join()`
    - `pthread_exit()`
- pthread_mutex
    - `pthread_mutex_init()`
    - `pthread_mutex_destroy()`
    - `pthread_mutex_lock()`
    - `pthread_mutex_unlock()`
- pthread_barrier
    - `pthread_barrier_init()`
    - `pthread_barrier_wait()`

In addition to that, it may be useful to look up the use of **`PTHREAD_MUTEX_INITIALIZER`** to statically initialize a mutex.

3.4 Testing

Start testing your program by using the *sequential* binary. This is a simple case where each thread adds a sequence of numbers to the list, though the sequences for each thread overlap. When running this on the stencil code (which is not thread-safe), the resulting linked list varies in size each time the program is run. You can get a count of the number of elements in the list at the end by running *./sequential <nthreads> <niterations> 2>/dev/null | wc -l*. Since this implementation is unsafe, some duplicates arise in the list, leading to lists that are larger than they should be in a thread-safe implementation.

As soon as you are sure that your program works with this simpler binary, move on to the *random* binary.

The program in the *random* binary adds and removes elements at random from the list, and will be a better test of whether your locking is valid. Do note that since the *sequential* binary is deterministic, it will not test your code fully, and you should move on to using the *random* binary before getting checked off.

3.5 Debugging

Concurrent programming occasionally leads to a common class of bugs called deadlocks. To solve deadlocks, it is often helpful to look at the state of threads while they are stuck to identify what they are waiting for. **`gdb`** offers a set of helpful debugging tools for multithreaded programs that allows just that. To start **`gdb`** with your program, type

```
gdb ./random
```

Once **`gdb`** is started up, use the **`run`** command followed by the program arguments to run your code.

While your program is running, it is possible to interrupt and pause it so you may inspect the state of each thread. You may interrupt **gdb** using **CTRL-Z**.

From here, you may print a list of threads by typing **info threads**. This command enumerates the threads. You will see a * next to the current thread you are inspecting. To inspect a different thread, type **thread NUMBER** where **NUMBER** corresponds to the numbers in the list you printed.

You will find it is most helpful to print a backtrace with the **bt** command. This will print the functions the current thread is executing.

If you want to run a command for all threads, you can run **thread apply all <command>**; for example, to print a stack trace of every thread, run **thread apply all bt**.

Use this information to discern where any deadlocks are coming from and how to fix your code.

4 Getting Checked Off

Once you've completed both sections of the lab, run **33lab_checkoff lab11**. You may run this command as many times as you want; we will use the most recent grade and handin time recorded by this program.