Asst2: Spooky Searching

Abstract

While Processes are the atomic unit of computing in most systems, they are not often used to implement parallelism. While it is possible to write multiprocess parallel code, threads are often preferred unless the extra services and capabilities that come along with making a new Process is necessary.

Introduction

When you fork(), a new Process is created. That Process is, as closely as can be determined, a copy of the Process that just invoked fork(). It has the same data and the same code. While it is common to exec() in a child Process and load different code, it is not necessary to do so. It is entirely possible to continue with two Processes running the same code. While running the same code in two Processes might seem rather redundant at first, there is no stipulation that both Processes have to do the same thing with the same code. In particular, one Process could, after determining it is a child Process, run some other function or jump to some other code while the parent Process continues on and runs the same code. Once you have two different contexts, they may do different things, even if they both have a copy of the same code.

Methodology

You'll write some code to compare multithreading to multiprocessing. The basic task will be to search a fairly long array for a value. You should write a common iterative search since you are testing how efficiently Processes run vs threads, not how efficiently you can write a search algorithm. Your code should split the list of numbers given in to groups of no more than 250 values and create another Process or thread to search each.

a. Random Number Generation

You will need to first generate a list of unique random numbers. While C has the facility to generate random numbers, it does not have a way to guarantee uniqueness. While it is unlikely that a number will repeat, it is not impossible. It makes no sense to generate a set of numbers and check their uniqueness by searching for each to see if it occurs in the list again, since the object is to use the list to do searching.

Rather than generating a random list and hoping they're all distinct, or checking them all, it is easier to generate a list in sequence and scramble it. First generate a list in sequence by allocating enough space for all your values, fill them in in order, then for each number in the list, generate two random numbers between zero and the max index and swap the two.

To generate a new random list for the next test, don't generate the entire thing over again. Generate only one new random index and swap the value at that index with the value at the index you searched for previously, then search for the same value again. This won't matter much for small lists, up to 100 or so, but once they start getting larger, generating all the random values would scale up the run time considerably. You do not want to have to generate a 20,000-element list more than once. As your lists get large, you may want to do less scrambling. Always generate at least three quarters as many random index pairs as the list is long (i.e. 15,000 pairs for a list of length 20,000).

b. Multiprocessing

>Be sure to set ulimit<

There are two issues with multiprocessing; indicating which Process should search which segment of the number list, and getting back information from it.

Given that you fork(), the data to be inspected will be in the child Process, however you need to control which part of it is searched by each Process. For this you can leave an indicator variable or flag in the parent; a loop variable, for instance. You need a way to deal with variable-sized arrays too. Not all of them will be the same length or divisible by 250.

You also need to find out whether the child Process actually found the value you are looking for and if so, which index it was at. For that you can use the system macros described in wait()'s man page to extract the lower 8 bits of the child's exit status. You can set the exit status to be the relative position of the value found, or a default value to represent 'not found' in the child. Be careful not to overload this value, though. It is only 8 bits long.

>Be sure to set ulimit<

c. Multithreading

Multithreading is a bit more direct, but requires some discipline in using memory and getting back data from the threads.

Given that each thread should only be reading from shared memory, you should have no issues with synchronization. Make sure you put your data to be shared between threads on the heap.

In order to get data back from the threads, make sure they invoke pthread_exit() and not return. You will need to pass in an exit pointer to capture the thread exit status value.

d. Variable make

Since you are testing two different searches you should strive to keep everything else the same; in particular you should write one external 'driver' program to invoke, run and test the fork-based search and the thread-based search the same way. How do you have the same code act differently? Set different library targets in your Makefile (as discussed in lecture).

As you recall from the previous project, you replaced all calls to malloc() with calls to your own code. The definition for the code existed in a library you compiled and loaded using your Makefile. There is no reason you can not do that again, but this time with a twist. In this case you want to invoke a the same program the same way, but in one case have it search using fork() and multiprocessing, and in the other case have it spawn threads and use multithreading. You can do this by having different compile targets.

Recall with your previous assignment, the actual code that was linked to the name 'mymalloc()' was in the library you compiled. The header only knew the function's name and prototype, not what it did (the implementation). You'll now be exploiting that fact. Your header should include a macro to redefine a dummy function name you use in your driver code and you should have two libraries with two different implementations of the redefined function name. Your Makefile should compile to the 'proc' and 'thread' targets. When compiled with 'proc', your Makefile should build the library that includes the fork-based implementation and compile your main code with the resulting object file. When compiled with 'thread', your Makefile should build the thead-based library and compile your main code with its object file. The 'all' target should exist, but emit an error requiring the choice of 'proc' or 'thread' targets. This will allow you to redefine the code that is loaded based on which build target you use.

Since the same dummy function name in the main source will be replaced, be careful to make sure the arguments it is called with will work for both your Process and thread code. Your tests should be set up in your main source so that a single invocation of your dummy function name performs a single test.

Results

Given that you have written the above, you can now test to see the general progression of amount of time to execute a search on a list of data of a given size vs whether Processes or threads are used. I have told you in class that threads are faster, but this is Computer Science and not Computer Lecture. You should verify the case for your self. Scientific conclusions must be not only repeatable but testable.

a. Testing and Composition

Be careful to use 'top' or 'ps aux' (or check the iLab status page) and select an iLab machine that seems not to be too busy. Lots of other people running lots of other stuff can alter your results. Given the number of students in this course alone, it is quite unlikely that the machine you choose will be entirely vacant. Do the best you can.

You should verify your own results(!). Run your test suite multiple times on different days to get different 'batches' of results. In one run of your test suite, be sure to run your individual tests multiple times (like memgrind). Report the min, max, average and standard deviation of the time each test below takes to run for; each batch, and over all batches. Be sure to graph the relations detailed below.

b. Test Suite

Similar to memgrid, write your own suite of tests to investigate the runtimes of Proceses and threads. You should compare Process and thread runtimes on different amounts of data. Make sure your steps in data size are significant based on what you want to test. You should determine:

- a general trend of: time vs. size of list to search for Processes as well as time vs. size of list to search for threads
- a tradeoff point for Processes vs threads
  i.e. how long a list would cause threads to perform at the same rate as a Process

e.g. perhaps, if you create a new thread/Proc for every 250 integers, then searching a list of 5000 integers using threads (requiring 20 threads) is as fast as searching a list of 250 integers using a single Process

- a tradeoff point for parallelism for Processes and threads
    - i.e. at what point does splitting the work over more Processes/threads make the task take longer than not doing so?
        - e.g. perhaps sorting a list of 250 elements, but splitting it up in to lists of size 10 (requiring 25 threads) is slower than splitting it up in to lists of size 11 (requiring 22 threads)

When run, your code should print out:

- which multi-mode it is in (-process or -thread)
- which test from your testplan document is being run
    - the parameters of the test
    - which iteration is being run
- the results (time) of each test
    - the aggregate results of a test batch once completed (min, max, average, standard deviation)

c. Extra

For additional credit, have tests that determine the following:

- average wall clock time to switch between Processes
- average wall clock time to switch between kernel threads

Conclusion

Submit the following:

In Asst2.tar.gz:

A 'Makefile' that will compile your search test to be multiprocessed when made with the 'proc' target and multithreaded when made with the 'thread' target.

(don't forget you need to compile with -lpthread)

A 'searchtest.c' that implements both your multiprocess and multithread searches based on how it is built

A 'multitest.h' header with a macro to replace your dummy search function in searchtest.c

A 'multitest_proc.c' library with a multiprocess implementation of your search function.

A 'multitest_thread.c' library with a multithread implementation of your search function.

A 'testplan.txt' or 'testplan.pdf' specifying how you will test the above.

A 'results.pdf' with your tests' data and graphs of the relations specified above.

Suggested implementation path:

Start with a simple single Process search and a single thread search.

Expand your code with a function that takes a pointer to test data and based on some parameters, forks() your single Process search code and reports when the result is found. Do the same for your thread code.

Make sure the the implementations have the same function prototype and move them to libraries.

Write your header and Makefile and check to make sure that you can correctly compile to different targets with some basic test functions/code.

Replace the test code with your libraries and test to make sure you can compile to them and they operate.

Start implementing your tests.