Lecture Notes on Operating Systems

Lab: Worker Thread Pool

Goal

In this lab, you will implement a simple master and worker pool, a pattern that occurs in many real life applications. You are given a program that spawns one master/producer thread. The master produces numbers continuously. You must complete this program in the following manner: you must store the produced numbers in a buffer, and spawn a pool of worker threads which will consume (i.e., print) these numbers to the screen.

This simple program is an example of a master-worker thread pool pattern that is commonly found in several real-life application servers. For example, in the commonly used multi-threaded architecture for web servers, the server has one master thread and a pool of worker threads. When a new connection arrives from a web client, the master hands the request over to one of the workers. The worker then reads the web request from the network socket and writes a response back to the client. Your simple master-worker program is similar in structure to such applications, albeit with much simpler request processing logic at the application layer.

Before you begin

Familiarize yourself with the `pthreads` or POSIX threads library, particularly with the synchronization primitives like mutexes and condition variables.

Part A: Thread pool with shared buffer array

You are given a skeleton program `master-worker-skeleton.c`. This program takes 3 command line arguments: how many numbers "produce" (M), the maximum size of the buffer in which the produced numbers should be stored (N), and the number of worker threads to spawn to consume these numbers (W). The skeleton code spawns a master thread, that produces the specified number of integers from 0 to $M - 1$ and exits. The main program waits for this thread to join and terminates.

You must copy this skeleton into your solution program `master-worker-array.c` and add the following functionality to it. You must initialize an integer array of N integers, which is shared between the master and worker threads. The master thread places produced integers into this array. The worker threads must remove integers from this array to consume and print to screen. The master thread must wait on a condition variable if the integer array is full, and the worker threads must similarly wait if the array is empty. The master and workers must coordinate to ensure that every request is processed exactly once and only once by some worker. While you need to ensure that all W workers are involved in consuming the integers, it is not necessary to ensure perfect load balancing between the workers. Once

all M integers (from 0 to $M-1$) have been produced and consumed, all threads must exit. The main thread must call `pthread_join` on the master and worker threads, and terminate itself once all threads have joined.

The master thread must call the function `print_produced` when it produces an integer into the buffer, and the worker threads must call the function `print_consumed` when it removes an integer from the array to consume. If your code is written correctly, every integer from 0 to $M-1$ will be printed exactly once by the master producer thread, and exactly once by the worker consumer threads. **Please do not modify these print functions, as their output will be parsed by our grading script.**

Part B: Thread pool with shared buffer linked list

For this part of the assignment, you must implement the shared buffer between the producer and consumer threads as a dynamically sized linked list, and not as a fixed size array. The linked list must start with zero elements. The producer thread must dynamically add elements to the list when producing integers, and the worker threads must delete elements from the list when consuming. Note that you must implement the linked list operations yourself, and must not use any library implementation of lists. You must dynamically allocate and free memory for the elements of the list. All insert/delete operations on the linked list must be implemented with proper locking to avoid race conditions. The maximum size of the linked list must still be bounded by the maximum buffer size N. All other functionality of your program remains the same as in part A.

Your code for part B should be written in a separate file `master-worker-ll.c` that should work independently from your code in part A.

Testing your code

To test correctness, you can run your code with small parameter values (say, a hundred numbers, a buffer of size 10, 4 workers), manually parse all the debug output generated, and convince yourself that your threads are running and terminating correctly. We have also provided you with some simple scripts, to easily verify the correctness of your code for a larger set of parameters. The script `check.awk` parses the output generated by your program, and checks that every number is printed exactly once by the producer thread and exactly once by the consumer thread. The script `test.sh` compiles your code, runs it, and invokes `check.awk` on the output produced by your code. The script `test.sh` takes the same command line arguments as your C program. You may use these scripts to test your program thoroughly before submission.

Submission instructions

- You must submit the files `master-worker-array.c` and `master-worker-ll.c`.
- Place these files and any other files you wish to submit in your submission directory, with the directory name being your roll number (say, 12345678).
- Tar and gzip the directory using the command `tar -zcvf 12345678.tar.gz 12345678` to produce a single compressed file of your submission directory. Submit this tar gzipped file on Moodle.

Grading

We will check correctness of your code using the test scripts provided. In addition to using the test scripts, we will also read through your code to ensure that you have adhered to the problem specification in your implementation.