ECE 650 Assignment 2: Fall 2016
Inter-Process Communication: Load Balancing
Due: 11:59 PM 22nd October 2016

Objective

This assignment will extend the work you did in Assignment 1, and allow you to to learn about, and gain practical experience, in load balancing with multi-producer/multi-consumer systems using (1) threads and shared memory (2) processes and message passing. In particular, you will use the pthreads, fork, mutex, semaphores, and message-queue facilities in a general Linux environment.

After this assignment, students will have a good understanding of, and ability to program with, one or more of: pthread_cond_xxx, pthread_mutex_xxx, sem_xxx system calls, and their use for controlling access to critical sections, as well as other system calls.

Requirements

Solve the multi-producer/multi-consumer problem with a bounded buffer using (1) multiple threads (P producers, C consumers) communicating via shared memory; (2) multiple processes (P producers, C consumers) communicating via message queues.

A typical software server receives requests, processes those requests, and generates replies. In this assignment, you should think of the producers as being entities that are receiving requests and consumers as entities that will process the request and generate a reply. In general, the ideal number of producers a system has will be a function of the request rate, which is a function of the number and performance of its Network Interface Cards (NICs) while the ideal number of consumers a system has is a function of the number of CPU cores, disk drives, etc., together with the complexity of the work required.

To emulate this, we will define a produce function, to be used by all producers, that issues a new request for the consumers after a random delay P_t. The size of the request to be transmitted to the consumers is R_s, likewise randomly distributed. Similarly, we will define a consume function, to be used by all consumers, that "completes" the task after a random time period. This time period, however, is more complex. Some work requests will be doable without IO, while others will require disk access and/or database access. The former will be relatively quick, while the latter will add an amount of time measured in milliseconds or tens of milliseconds. We therefore define parameter p_i as the probability that the work requires IO. There will then be two randomly distributed consumer time values, C_{t1} and C_{t2}, where the first is chosen with probability p_i and otherwise the second is chosen. We will ignore the cost of generating the reply, though this is problematic, as replies will be transferred via the system NICs and therefore the ideal number of consumers should factor that in, together with the size of a reply.

As with assignment 1, producers cannot continue to produce data for the consumers if there are more than a certain amount of data that has not yet been consumed. Define `B` as the number of integers that producers may send without consumers having consumed them before producers must stop producing. Unlike assignment 1, we are now thinking in terms of requests being sent to our producers. Every time a producer has a "request" (i.e., the random time is completed), but it cannot put the data into the shared space/message queue, we consider that producer to be blocked. This

is a bad thing. You should keep track of the number of times, and the amount of time, for which producers are blocked.

Whenever a consumer has removed a request from the shared memory/message queue, processed it (i.e., delayed for the requisite time (C_{t1} or C_{t2}, as the case may be)), that consitutes the completion of a request. You will want to keep track of the total number of requests completed. Also, just as a producer is blocked if the buffer/queue is full, consumers have nothing to do if the buffer/queue is empty. You should keep track of the number of times, and the amount of time, for which consumers are idle.

Requirements:

On a Linux platform, create a multi-producer/multi-consumer with a fixed-size shared buffer/message queue. The initial process will pthread_create()/fork() all other necessary processes.

```
./server <T> <B> <P> <C> <P_t parms> <R_s parms>
                  <C_t1 parms> <C_t2 parms> <p_i>
```

where <T> is the amount of time for which the system should execute, in sexonds, B is a parameter specifying the size (in bytes) of the shared memory space/shared message-queue size, <P> is the number of producers, <C> is the number of consumers, <P_t parms> is the parmeters related to the probabiity distribution for the random time P_t that the producers must wait between request productions, <R_s parms> is the parameters related to the probability distribution of the request size, <C_t1 parms> is the parmeters related to the probabiity distribution for the random time C_{t1} that the consumers take with probability p_i, <C_{t2} parms> is the parmeters related to the probabiity distribution for the random time C_{t2} that the consumers take with probability $1 - p_i$, and <p_i> is the probability p_i.

The command will execute per the above description and will then print out the time it took to execute, and how many requests were satisfied during that time period; the time for forking processes is fixed relative to the operation and therefore should be kept separate in your timing measurement. Further, steady-state operation is desirable information. Therefore you should measure the time before forking, and also periodically you should determine how many requests have been satisfied (*e.g.*, if the execution runs for a couple of minutes, it would be useful to determine how many requests are processed in every 10 second interval. You should also print out all producer blocking information.

Finally, for a given buffer size, P_t, etc. parameter, you should determine what is the ideal number of producers and consumers. The ideal will be that number for which the most number of requests are processed, with the fewest blocked producers. Accounts will be created within a week on servers with significantly more cores than a simple VM on a client box, which will allow you to approrpiately determine what the effects are of your design.

Deliverables

Submit the following items to a dropbox on Learn before the deadline. We will grade the last submission when there are multiple submissions of the same item.

1. Source code.
   Jar the entire source code together with other documents identified below and name the `.jar` file `assignment2.jar`.

2. An assignment report (pdf format, named "report2.pdf") which containing design choices and results of your two programs.

3. Run experiments multiple times, and compute averages and deviations.

References

[1] AD Marshall. Programming in c unix system calls and subroutines using c. *Available on-line at http://www.cs.cf.ac.uk/Dave/C/CE.html*, 1999.

[2] M. Mitchell, J. Oldham, and A. Samuel. Advanced linux programming. *Available on-line at http://advancedlinuxprogramming.com*, 2001.