Assingment 2: Multi Producer, Multi Consumer ECE 650 Fall 2016

Due on 30th October 2016

Murshed, Niyaz

nmurshed@uwaterloo.ca ID 20641996

1 Introduction

Interprocess communication (IPC) refers to the coordination of activities among cooperating processes. A common example of this need is managing access to a given system resource. To carry out IPC, some form of active or passive communication is required. [1]
The increasingly important role that distributed systems play in modern computing environments exacerbates the need for IPC. Systems for managing communication and synchronization between cooperating processes are essential to many modern software systems
. The most popular form of interprocess communication involves message passing. Processes communicate with each other by exchanging messages. A process may send information to a port, from which another process may receive information. The sending and receiving processes can be on the same or different computers connected via a communication medium. One reason for the popularity of message passing is its ability to support client-server interaction. A server is a process that offers a set of services to client processes. These services are invoked in response to messages from the clients and results are returned in messages to the client. Thus a process may act as a web search server by accepting messages that ask it to search the web for a string.[2]
The child process is initially running its parents program, with its parents virtual memory, file descriptors, and so on copied.The child process can modify its memory, close file descriptors, and the like without affecting its parent, and vice versa.[3]
As with processes, threads appear to run concurrently; the Linux kernel schedules them asynchronously, interrupting each thread from time to time to give others a chance to execute. Conceptually, a thread exists within a process.Threads are a finer-grained unit of execution than processes.When you invoke a program, Linux creates a new process and in that process creates a single thread, which runs the program sequentially.That thread can create additional threads; all these threads run the same program in the same process, but each thread may be executing a different part of the program at any given time.
When a program creates another thread, though, nothing is copied.The creating and the created thread share the same memory space, file descriptors, and other system resources as the original. If one thread changes the value of a variable, for instance, the other thread subsequently will see the modi- fied value. Similarly, if one thread closes a file descriptor, other threads may not read from or write to that file descriptor. Because a process and all its threads can be exe-cuting only one program at a time, if any thread inside a process calls one of the exec functions, all the other threads are ended (the new program may, of course, create new threads).[3]

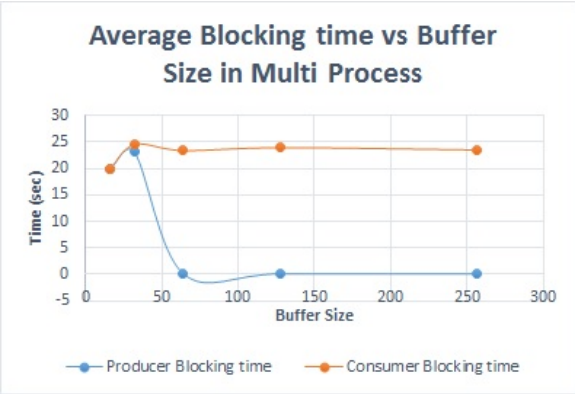
2 Program Description

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,a produce function is defined, to be used by all producers, that issues a new request for the consumers after a random delay Pt. The size of the request to be transmitted to the consumers is Rs, likewise randomly distributed. Similarly,a consume function is defined, 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

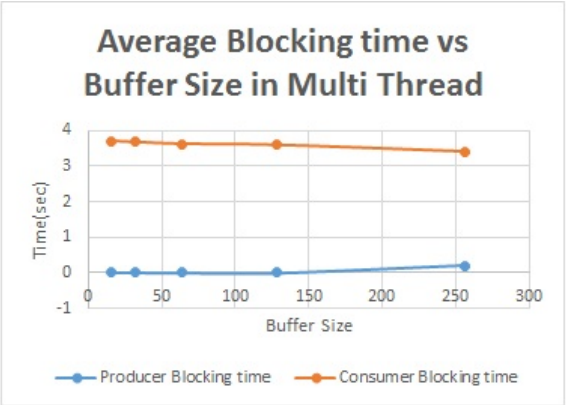
amount of time measured in milliseconds or tens of milliseconds. We therefore define parameter pi as the probability that the work requires IO. There will then be two randomly distributed consumer time values, Ct1 and Ct2, where the first is chosen with probability pi and otherwise the second is chosen. 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.B is the number of integers that producers may send without consumers having consumed them before producers must stop producing. 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. Whenever a consumer has removed a request from the shared memory/message queue, processed it (i.e., delayed for the requisite time (Ct1 or Ct2, as the case may be)), that constitutes the completion of a request.Just as a producer is blocked if the buffer/queue is full, consumers have nothing to do if the buffer/queue is empty.
For my simulation,I have Poisson distribution for arrival rate of request and processing time.User inputs the average number of request desired per second, average number of request processed per second in IO and average number of request processed in memory/disk access.
A recursive function is used to create multiple producers and consumers. Mutex and semaphores has been used to synchronize multi processes and threads.
An user can input varius combination of paramers as follows :
./m_thread 60 16 100 50 0.1 30 0.8 0.2 0.2
where ,
60 : execution time
16 : Buffer size
100: Number of producers
50 : Number of consumers
0.1: rate of requests
30 : average message size in bytes
0.8: rate of processing time in case if I/O
0.2: rate of processing time in case if memory access
0.2: Probability of request needing I/O or memory access

3 Test Results

For the above programs, I have ran the simulation for various sizes of buffer to analyze how buffer size effects the waiting time.I have kept the number of producers as 20 and number of consumers as 100 with random request and random processing time with Poisson distribution.
I have run the simulation for 90 seconds and have taken samples every 10 seconds.
Below figure shows the variation of blocking time for different buffer sizes.



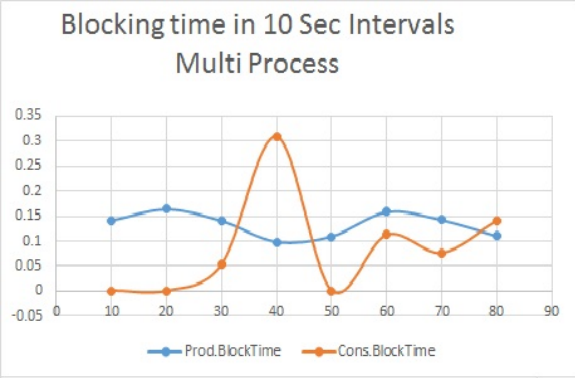
(a) Blocking Time vs Buffer Size



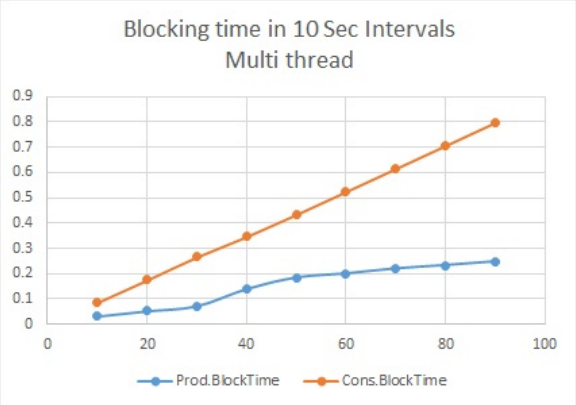
(b) Blocking Time vs Buffer Size

From above results, we can notice that in multi-process communication, both producer and consumer has delay at the beginning of execution, however, after a certain time both are stable. However, in multi-thread, both producer and consumer are stable from the beginning.This maybe due to the fact that process are spawned in a slower rate then threads, thus producers may have been spawned before all consumers are spawned resulting in blocking time in producers. Once all processes are spawned, blocking time is 0 for producers as expected.

For the next experiment,data was extracted every 10second interval to check blocking time varies with time.The execution time and all other parameters were kept constant.Below figures shows the change in blocking time with increasing time.



(c) Blocking Time in 10sec interval



(d) Blocking Time in 10sec interval

Above figures shows us that in terms of multi threading we have a linear progression of blocking time with increasing time however, in multi process the blocking delays are unstable and unpredictable.For a developer, knowing how the resources will be used and delay caused is of high importance.Multi threading gives this advatnage over multiprocesses.

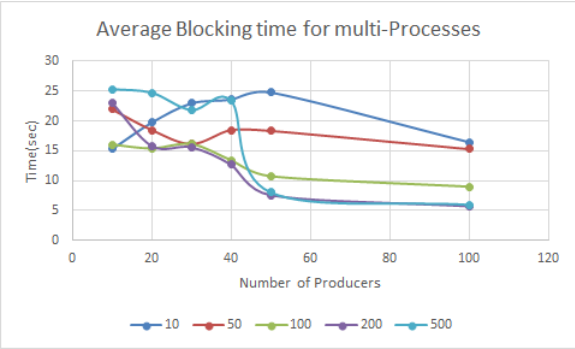
Finally, simulation was ran for multiple combination of producers and consumers to find an optimal producer consumer pair that would give the best results.The tables below shows the average blocking time in the multi process system with different number of producers and consumers.In the case of threads, it shows the total blocking time of all threads with different number of producers and consumers.

Table 1: Average Blocking Time : Multi Process

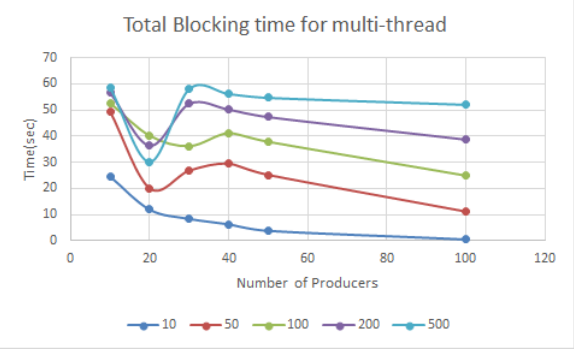
	10	**50**	**100**	**200**	**500**
10	15.28	21.95	16.06	23.04	25.33
20	19.74	18.49	15.43	15.81	24.73
30	22.95	16.15	16.15	15.6	21.85
40	23.57	18.49	13.44	12.68	23.36
50	24.77	18.34	10.77	7.58	8.03
100	16.46	15.29	9.01	5.72	5.97

Table 2: Blocking Time : Multi Thread

	10	**50**	**100**	**200**	**500**
10	24.55	49.42	52.7	56.7	58.64
20	11.89	20.05	40.21	36.38	30.17
30	8.36	26.856	36.22	52.38	58.04
40	6.29	29.54	41.22	50.2	56.177
50	3.76	25.21	37.89	47.33	54.69
100	0.48	11.18	24.92	38.69	52.005



(a) Blocking time : Multi Process



(b) Blocking time : Multi Thread

The simulation shows that, in terms of multi process communication, blocking time decreases with increasing number of consumers. A combination of 50-500 producer-consumer pair gives the lowest blocking time.
In the case of multi threads, total blocking time gets to a stable rate, and higher number of threads causes higher blocking delay.a combination of 10-100 producer-consumer threads gives the optimal performance.

4 Conclusion

From the results, we can notice that the threading module uses threads, the multiprocessing uses processes. The difference is that threads run in the same memory space, while processes have separate memory. This makes it a bit harder to share objects between processes with multiprocessing. Since threads use the same memory, precautions have to be taken or two threads will write to the same memory at the same time. This is what the global interpreter lock is for.
Spawning processes is a bit slower than spawning threads. Once they are running, there is not much difference.

In case of critical instructions in OS, it is recommended to use muti processes to reduce sharing of critial data that is done in muti processing.

References

[1] S. M. Lewandowski, "Interprocess communication in unix and windows nt," 1997.

[2] "Comp 242 class notes section 3: Interprocess communication." `http://www.cs.unc.edu/~dewan/242/s04/notes/ipc.PDF`. Accessed: 2016-10-06.

[3] M. Mitchell, J. Oldham, and A. Samuel, *Advanced linux programming*. New Riders, 2001.