ECE 650 Assignment 1: Fall 2016
Inter-process Communication: Single Producer, Single Consumer
Due: 11:59 PM 6th October 2016

Objective

This assignment is to learn about, and gain practical experience, in single producer/single consumer inpter-process communication using (1) threads and shared memory (2) processes and message passing. In particular, you will use the pthreads, fork, exec, and message-queue facilities in a general Linux environment.

After this assignment, students will have a good understanding of, and ability to program with, the following:

- the `fork()` and `exec()` system calls, and their use for creating a new child process on the Linux platform;
- the `wait()` family system calls, and their use to obtain a child-process' status-change information;
- the message-queue facility (`<sys/msg.h>`, `<mqueue.h>`, *etc.*) on the Linux platform;
- various `pthreads` system calls
- shared memory between two threads

Recommended Reading Before Starting

1. Read Section 3.2.2 about `fork()` and `exec()` and Sections 3.4.1 and 3.4.2 about `wait()` in [2].
2. Sections 4.1 (thread creation) and 4.4 (synchronization)
3. Read supplementary materials regarding IPC Message Queues and Threads in [1] (`http://www.cs.cf.ac.uk/Dave/C/node25.html` and `http://www.cs.cf.ac.uk/Dave/C/node29.html`).

Requirements

Solve the producer-consumer problem with a bounded buffer using (1) two threads (one producer, one consumer) communicating via shared memory; (2) two processes (one producer, one consumer) communicating via message queues.

This is a classic multi-tasking problem in which there are one or more tasks that create data (these tasks are referred to as "producers") and one or more tasks that use the data (these tasks are referred to as "consumers"). For the purpose of this assignment, we will use a single producer and single consumer. The producer will generate a fixed number, `N`, of random integers, one at a time. Each time a new integer is created, it is sent to the consumer using a shared memory space (or the message-passing facility). The consumer task reads the data from shared memory (or a receive queue) and prints out the integer it has read[1].

For the threaded, shared-memory approach, the shared buffer space is necessarily of finite size. As such, it is important to synchronize the producer and the consumer so that the producer does not overfill the buffer and so that the consumer does not read data that has not been placed in the buffer by the producer.

For the message passing solution, the two processes will not share any memory, and as such we do not need to worry about conflicting operations in shared-memory access: the operating system's message-passing facility takes care of the shared access within kernel space. However, kernel memory is finite, and thus there cannot be an unbounded number of messages outstanding; at some point the producer must stop generating messages and the consumer must consume them, otherwise the kernel's memory will be completely consumed with messages, blocking the sender from further progress. It is, in general, preferable for the producer to limit itself explicitly, rather than have the system block it because of resource exhaustion, since the latter case may well result in deadlock.

What is needed in both cases, therefore, is that the producer cannot continue to produce data for the consumer if there are more than a certain amount of data that has not yet been consumed. Define `B` as the number of integers that the producer may send without the consumer having consumed them before the producer must stop producing (in general `N > B`.). This is the total size of the shared memory buffer or the total number of messages that the operating system must be able to store, *e.g.,* within a message queue, depending on the particular solution.

If the producer must cease producing after at most `B` unconsumed integers, but needs to send `N > B` integers in total, then it must have some way of knowing that the consumer has consumed an integer. There are various ways to handle this issues, and it is left as a design choice for you, and you should study the alternatives to think about the tradeoffs implied by any particular choice. Just as the producer may block if it has sent `B` unconsumed integers, the consumer may block if there are currently no integers to consume but more are expected from the producer. The program terminates when the consumer has read and displayed all `N` integers from the producer.

Requirements:

- On the Linux platform, create a producer-consumer pair of processes with a fixed-size message queue in which the consumer is a child process of the producer (use the `fork()` system call). The system is called with the execution command

```
    ./produce <N> <B>
```

[1]If it helps, you can think of the producer as a keyboard device driver and the consumer as the application wishing to read keystrokes from the keyboard; in such a scenario the person typing at the keyboard may enter more data than the consuming program wants, or conversely, the consuming program may have to wait for the person to type in characters. This is, however, only one of many cases where producer/consumer scenarios occur, so do not get too tied to this particular usage scenario.

where `N` is a parameter specifying the number of integers the producer should produce and `B` is the number of integers the message queue can hold. Note that in general `N > B`. The command will execute per the above description and will then print out the time it took to execute; the time for forking is fixed relative to the operation and there fore should be kept separate in your timing measurement. Therefore you should measure the time before the fork, before the first integer is generated, and after the last integer is consumed and displayed. You should then print the time differences. Use `gettimeofday()` for to measure the time and the terminal screen for display. Your output should be:

```
Time to initialize system: <whatever the result is (in seconds)>
Time to transmit data: <whatever the result is (in seconds)>
```

- Repeat this exercise, but using threads and shared memory.

Deliverables

Submit the following items to the marmoset submission system 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 `assignment1.jar`.
2. An assignment report (pdf format, named "report1.pdf") which contains the following items.
   - Results of your two programs
     - Use tables or graphs (or both) to show how the average initialization time and average data-transmission time varies as (N, B) varies.
       Table 1 shows the (N, B) values you are required to present in the tables or graphs. You are free to present more timing data from (N, B) values not listed in Table 1 if you wish. If you want to use graphs, you may want to consider log-log plots.

		B				
		16	32	64	128	256
N	20	t_{11}	N/A	N/A	N/A	N/A
	40	t_{21}	t_{22}	N/A	N/A	N/A
	80	t_{31}	t_{32}	t_{33}	N/A	N/A
	160	t_{41}	t_{42}	t_{43}	t_{44}	N/A
	320	t_{51}	t_{52}	t_{53}	t_{54}	t_{55}
	640	t_{61}	t_{62}	t_{63}	t_{64}	t_{65}

Table 1: Timing measurement data table for given (N, B) values in 1000s

  - A table to show the standard deviation of the data transmission time for given (N, B) values in Table 1.
  - A histograms of the average data transmission time given $(N, B) = (640, 128)$.

- Comparison of your testing results using threads/shared memory vs. processes/message passing. Discuss the cost difference between these two approaches for solving the producer-consumer. Your conclusion should be based on comparing the performance difference (if any) by using the timing measurement data.

Precise details of the submission format will be e-mailed within the next week.

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.