Project 1: Interprocess Communication Using Shared Memory

100 points

Due date: Midnight (11:59pm) Fri. Feb. 20th, 2009

Motivation

One of the two most popular methods of interprocess communication is to share memory as the communication buffer space, or simply *shared memory*. Most large size applications consist of many programs, and, therefore, use shared memory internally to manage the sheer size of application. One of the common models to evaluate the communication method (shared memory) is to use the *bounded buffer* problem. The producer (writer) and consumer (reader) processes can be implemented either by a process or by a thread. Understanding of the fundamental issues of the bounded buffer problem together with process and thread issues will be a good basis to advance the knowledge of complicated operating systems and applications. In addition, the bounded buffer problem will be revisited later in this course.

Goal

By doing this project, students will learn 1) the methods to create and manage a child process and a thread, 2) shared memory as one of the two most popular interprocess communication methods, 3) the problem of bounded buffer, and, 4) the performance issues of processes and threads.

Bounded Buffer

This project is to investigate the performance issues of a *bounded buffer* on a single CPU, firstly using processes and secondly using threads. A bounded buffer (BB) typically runs this way.

1. *Syntax*: BB par1 par2 par3
    - *par1*: number of the total items to experiment
    - *par2*: size of the bounded buffer in number of items (length of the cycle in number of items)
    - *par3*: 1 (processes) or 2 (threads)
2. At the beginning, the main program creates a pair of processes (two children) as the producer and consumer.
3. The producer creates and attaches a shared memory of size *par2* items
4. The consumer attaches the shared memory, the producer created.
5. Depending on *par3*, both the producer and consumer create a writer process (or thread) and a reader process (thread) respectively.
6. The writer starts writing with 1 (integer) and ends writing with the number of total items such that each item has only one unique consecutive integer in an increasing order.

7. The reader starts reading with the very first item (1, integer) and keeps reading until the number of total items.
    - It reports whenever it finds either a sequence gap or a duplicate.
    - At the end, it reports the summary, the total number of items read, the total number of sequence gaps found, and the total number of duplicates found. If found, where those gaps and duplicates occurred.
8. After par2 number of writings (at the end of each cycle), the producer process terminates the writer, creates a new writer, and lets the new writer continue to produce exactly at the point where the predecessor left off (without sequence gap).
9. After par2 number of readings (at the end of each cycle), the consumer process terminates the reader, creates a new reader, and lets the new reader continue to consume exactly at the point where the predecessor left off (without sequence gap).
10. The exact time of terminating and recreating a writer can be different from that of a reader depending on the local CPU scheduling and the local workload.
11. At the end, after the reader reports the final summary, the consumer process terminates the reader and detaches the shared memory, the producer terminates the writer and detaches and removes the shared memory, and finally the main program terminates.

Analysis Report

- The first issue is the performance of the processes and thread. In order to investigate the overhead difference, create a graph which has two curves; one for processes and the other for threads. The curve will show the total amount of time the experiments took for both processes and threads. Use at least three comparison points for each curve. For instance, cycle of 100 items and the total items of 1,000, 10,000, and 100,000. Try to generalize these results to the situation where a server at "Amazon.com" could improve the performance by using threads instead of processes.
- The second issue is related to the general problem of "process synchronization", which will be discussed in detail in Chapter 6. First, state clearly if your implementation utilizes *par2* - 1 item slots or exact amount of *par2* and why. Note that you are not required to utilize the exact amount of *par2* but you are required to know what exactly you have implemented. Second, your implementation may or may not produce sequence gaps or duplicates. Explain why or why not your implementation does (not) produce those.

Implementation Techniques

- Use "fork" to create a process.
- Use POSIX pthread libraries to manage threads (not Windows thread libraries).
- Use Unix shared memory system calls to create and remove a shared memory (not Windows system calls).

- Use an appropriate data structure for the data item.
- Use an appropriate algorithm to implement a bounded buffer. The algorithms in the textbook can be a choice.

Testing

- The experimental results will be probabilistic not deterministic. Run a scenario many times (more than 10) to get a statistical value. State explicitly how many runs were used to get a value on the curve.
- After the experiments, there should be no dangling shared memory (not attached to any process) which was created by the producer.
- No specific output form is required on this project. Each project report should contain instructions to interpret the unique output. If needed, a command to show an interim status will be welcome.

Type

Either individual or a team of two people. No penalty on team work.

What/How to Turn In

You turn in source code, binary code, README, and a makefile if there is. Any instruction or information necessary to remake the binary code. Show clearly at the top of the program: author names, email addresses, student IDs, and the machine name in the CS lab where you ran your code. The grader will remake the binary code and will run the binary on the machine where you produced the results. A README file is expected in each submission which details the procedure to demonstrate and test. Turn in one .tar file. Name it as "firstletter-of-the-first-name-and-lastname-p109.tar". Mine would be "bchoi-p109.tar", for example. If you are a team of two, name it lastname1-lastname2-p108s.tar. Use "blackboard" to turn in your work.

Grading

All honest and correct work will get full points. Make sure that your implementation does not belong to any of the followings.

- Copy of someone else's work will get zero point.
- Zero point will be given if the grader is not able to regenerate the binary using the instructions submitted.
- Half points will be given if the binary fails to run while the grader tests it.
- There will be some deduction if the analysis is not compatible with the submitted results.
- There will be some deduction if your implementation leaves a dangling shared memory after testing.
- There will be some deduction if the binary does not run as it is supposed to.
- There will be some deduction if no README is submitted.