Lab 2 – Inter-Process Communication Engines

IUPUI CSCI 503 – Fall 2013
Assigned: September 9, 2013
Due time: 11:59PM, September 23, 2013

1. Objective

Lab 2 is created to help you understand how to use IPC mechanisms to support communications between two processes. In the class you have learned Pipe, RPC, Socket, Shared Memory, and Signal, this lab will only require you use two IPC methods: Unix Socket and Shm to enable processes to send/receive messages.

2. Background

This section introduces the context of the lab and gives you an idea of why and when you would need to do this type of task.

When system software engineers in industry build backend server systems (e.g., in big cloud or data centers), they often start and run *m* communication-driver processes and *n* application processes on the server, where *m* <= *n*. The driver process receives requests from a great number of clients, and then sends them to the application process to process. Note that here the driver process and the application process are on the same machine. To implement this step, you need to use an inter-process communication mechanism. In addition, if the application process needs to send replies back to clients, then you will use another communication channel directing from the application to the driver. To build such a kind of backend server, the key component is to implement a communication engine. In this lab, you will use two IPC methods to implement such a communication engine.

3. A Simplified Problem

In the real world, the driver itself and the application itself can be extremely complicated and vary a lot depending on the companies' needs. However, in Lab 2, we are going to simply the problem as much as possible to make it doable by a single student within two weeks. So here is our simplified communication engine problem:

- There is one driver process
- There is one application process (i.e., *m* = 1, *n* = 1)
- Messages are sent only from the driver to the application (i.e., one direction only)
- Messages are just integers (each message contains one integer)

Now, the problem sounds much simpler, right?

In the lab, you are required to implement two different versions of the communication engine:

1) Use Unix Socket to send messages from the driver process to the application process
2) Use Shm to pass messages from the driver process to the application process

The following two subsections describe how you should implement the two versions.

3.1 The Unix Socket Version

This one is straightforward. The driver process works as a Client, while the application process works as a Server. The driver process keeps sending messages (i.e., integers in our lab) to the application process through a **Unix Socket** (note: it is not TCP or UDP Internet Socket). Both processes run on the same machine.

Correctness

In your program, the driver process generates random integers and keeps track of the sum of the integers sent to the application process. On the receiving side, the application process also counts the sum of the received integers. After the driver is done, it will send a "special" message to the application and then exit. Upon the exit point, both processes should print the sum of the integers (either sent or received). If the two sums are equal, your program is "all right".

Performance

Moreover, you need to measure the throughput of your communication engine. That is, how many messages can be sent per second? For example, you can measure the time elapsed for a number of M million messages.

3.2 The Shm Version

The Shm version is a little bit trickier than the above socket version. You need an additional data structure to realize it: **Ring Buffer**. The ring buffer should be allocated in a shared memory region and accessed by both driver process and application process. The driver process keeps adding messages to the ring buffer. At the same time, the application process keeps removing messages from the ring buffer. Because the ring buffer is shared, two processes can produce and consume at the same time. Our lecture has covered both Ring Buffer and Shm.

To check your program's **correctness**, look at the above "Correctness" section.

To check your program's **performance**, look at the above "Performance" section.

4. Score Distribution

- Coding style: **10%** (as described in the "Labs Grading Policy")
- Unix Socket version: **30%** (25% correctness + 5% performance result)
- Shm version: **60%** (50% correctness + 10% performance result)

Again, one “Segmentation Fault” will lead to an automatic deduction of 20 points.

The total score is 100 points.

Hint: Use the “correctness” test to check if your code works correctly or not; use “fork” to create two processes.

5. Deliverable

There should be two subdirectories: one for each version.

In each subdirectory, there should be source code in C, a Makefile, and a README. You need to put your observed performance result (i.e., millions of message per second) in the README file.

Make a tar ball for the above two subdirectories and send it to the TA. The tar ball name may be “Lab2_your_name.tar”.