Oregon State University

Operating Systems II

CS444

Project 1 Writeup

Author:
Dylan Camus

Professor:
Dr. Kevin McGrath

April 14, 2016

Abstract

This project involved the exploring the uses of pthreads to solve concurrency problems. Specifically, the producer consumer problem was implemented using pthreads. Additionally, the Linux kernel was installed and run on the qemu virtual machine.

1 Commands Used

Command
git clone git://git.yoctoproject.org/linux-yocto-3.14
git checkout tags/v3.14.26
source /scratch/opt/environment-setup-i586-poky-linux.csh
cp /scratch/spring2016/files/bzImage-qemux86.bin /scratch/spring2016/cs444-014/linux-yocto-3.14
cp /scratch/spring2016/files/core-image-lsb-sdk-qemux86.ext3
cp /scratch/spring2016/files/config-3.14.26-yocto-qemu /scratch/spring2016/cs444-014/linux-yocto-3.14/.config
qemu-system-i386 -gdb tcp::5514 -S -nographic -kernel bzImage-qemux86.bin -drive file=core-image-lsb-sdk-qemux86.ext3, if=virtio -enable-kvm -net noe -usb -localtime –no-reboot –append "root=/dev/vda rw console=tty50 debug"
$GDB
remote target :5514
continue

Figure 1: This table includes every command used to get the kernel running

2 Qemu Flags

'-gdb tcp::5514'
This flag tells qemu to wait for a gdb connection from port 5514

'-S'
Do not start CPU on startup

'-nographic'
Disables SDL display through VGA output. This makes qemu a simple command line application

'-kernel bzImage-qemux86.bin'
Use bzImage-qemux86.bin as the kernel image

'-drive file=core-image-lsb-sdk-qemux86.ext3,if=virtio'
This tells qemu to use core-image-lsb-sdk-qemux86.ext3 as it's disk image. It also says that the drive is connected on a virtio interface.

'-enable-kvm'
Enable KVM full virtualization support.

'-net none'
This indicates that no network devices are to be configured

'-usb'
This enables the USB driver

'-localtime'
Sets localtime

'–no reboot'
Tells qemu to exit instead of rebooting

'–append "root=/dev/vda rw console=ttyS0 debug"'
Tells qemu to use root=/dev/vda as cmdline

3 Concurrency Writeup

The concurrency assignment is the classic consumer producer multi-process synchonization problem. In the problem, there are two processes, the producer and the consumer. Both the producer and the consumer share a common buffer. The buffer is implemented as a queue. The producer generates data and puts it into the buffer. At the same time, the consumer consumes this data, thus removing it from the buffer. In this specific implementation the producer and consumer take a randomly generated amount of time to produce and consume. Therefore, there is a possibility that the buffer may be either full or empty at any givin point. The solution to this problem is to have the producer block when the buffer is full until the consumer notifies it that it has consumed from the buffer and that the buffer is no longer full. Additionally, the consumer must block when the buffer is empty until the producer has notified it that it has produced an item.

There are a few considerations that need to be made for this problem. For example, it is possible for the producer to preempt the consumer after the consumer has checked if the buffer is empty, thus causing the consumer to not wake up. This is solved by using a mutex. The mutex enforces mutual exclusion of the buffer during these checks to prevent possible race conditions.

4 Concurrency Questions

4.1 What do you think the main point of this assignment is?

I believe that the main purpose of the assignment was to explore concurrency, which is a major concept within operating systems. Operating sytems are by design made to run many processes at the same time. Many of these processes need access to the same resources. This project allowed me to understand how the operating system allows multiple processes to be running at the same time and accessing the same resources without causing race conditions.

4.2 How did you personally approach the problem? Design decisions, algorithm, etc.

My approach was simple. First, I found an implementation for a queue, which I used for the buffer. Then, I created two functions, producer and consumer. The producer will check if the buffer is full. If it is, it blocks until the consumer signals to it that the buffer is no longer full and production can resume. The consumer acts in a similar way, except it blocks when the buffer is empty and waits for the producer to signal to it that there are items in the buffer ready to be consumed. In between checking if the buffer is full or empty and actually writing data to it, the buffer is locked through use of a Mutex. This prevents the buffer from changing states in between reading and writing to the buffer.

4.3 How did you ensure your solution was correct? Testing details, for instance.

I tested my solution by writing tests that ran the program with different inputs. For example, my tests would try running with a very small buffer, or a very large buffer (with smaller sleep times to make the test run faster). This was to help test the buffer full case. I also tested static sleep times for both the producer and the consumer. For example, I tested what would happen if the producer always took longer than the consumer, and vice versa.

4.4 What did you learn?

I learned how to use pthreads to create concurrency within a c program. I also learned how to debug multithreading based bugs. Finally, I learned that assignments in cs444 are more difficult than they appear, and I will be sure to utilize all the given time for my future assignments to avoid having to scramble at the last minute.

5 Version Control Log

Detail	**Author**	**Description**
f020a7f	Dylan Camus	testing pthreads
f5a0bc3	Dylan Camus	got rdrand and mt working
5680364	Dylan Camus	implemented producer consumer on both rdrand and mt
00e55f8	Dylan Camus	Finished concurrency assignment and started writeup.
31420ec	Dylan Camus	finished writeup, final commmit

Figure 2: Git commit log

6 Work Log

Date	Hours
4/9	2
4/10	2
4/11	5
4/12	5

Figure 3: Dates and number of hours spent working on project 1