Group 11-05: Daniel Schroeder and Brian Ozarowicz

CS 444 OPERATING SYSTEMS II
SPRING 2017

Assignment I: Getting Acquainted

Abstract

This document is a summary of Assignment 1 for CS 444 Operating Systems II at Oregon State University Spring 2017. This document includes a log of commands used to complete the kernel assignment, a description of all flags in the qemu command, a description of the concurrency assignment solution from Group 11-05, and the responses to the concurrency assignment follow up questions.

I. Kernel Assignment

Log of Commands

1) `cd /scratch/spring2017`
2) `mkdir 11-05`
3) `cd 11-05`
4) `git clone git://git.yoctoproject.org/linux-yocto-3.14`
5) `git checkout -b v3.14.26`
6) `source /scratch/opt/environment-setup-i586-poky-linux.csh`
7) `cd linux-yocto-3.14`
8) `cp /scratch/spring2017/files/config-3.14.26-yocto-qemu .config`
9) `make menuconfig`
10) `/`
11) `LOCALVERSION`
12) `1`
13) `-11-05-hw1`
14) `make -j4 all`
15) `cd ..`
16) `cp /scratch/spring2017/files/bzImage-qemux86.bin .`
17) `cp /scratch/spring2017/files/core-image-lsb-sdk-qemux86.ext3 .`
18) `qemu-system-i386 -gdb tcp::6605 -S -nographic -kernel bzImage-qemux86.bin -drive file=core-image-lsb-sdk-qemux86.ext3,if=virtio -enable-kvm -net none -usb -localtime --no-reboot --append "root=/dev/vda rw console=ttyS0 debug"`
19) `gdb`
20) `target remote :6605`
21) `continue`
22) `root`
23) `uname -a`
24) `reboot`
25) `qemu-system-i386 -gdb tcp::6605 -S -nographic -kernel linux-yocto-3.14/arch/x86/boot/bzImage -drive file=core-image-lsb-sdk-qemux86.ext3,if=virtio -enable-kvm -net none -usb -localtime --no-reboot --append "root=/dev/vda rw console=ttyS0 debug"`
26) `gdb`
27) `target remote :6605`
28) `continue`
29) `root`
30) `uname -a`
31) `reboot`

QEMU Flags

- `-gdb tcp::11-05` Waits for gdb connection on the specified port
- `-S` Does not start the CPU at startup
- `-nographic` Disables all graphical output so that the QEMU runs as a simple command line application
- `-kernel bzImage-qemux86.bin` Specifies the kernel image to use
- `-drive file=core-image-lsb-sdk-qemux86.ext3,if=virtio` Defines a new drive using the specified disk image (`file`) that is connected via virtio interface
- `-enable-kvm` Enables KVM (Kernel Virtual Machine) full virtualization support. This option is only available if KVM support is enabled when compiling.
- `-net none` Does not configure network devices
- `-usb` Enables the USB driver
- `-localtime` Sets the real time clock to the system's local time
- `--no-reboot` Exits instead of rebooting
- `--append "root=/dev/vda rw console=ttyS0 debug"` Adds additional options to the kernel command line which set the root filesystem as the first detected paravirtualized disk driver (`/dev/vda`), mounts the device read-write on boot (`rw`), and sets the output console to be the virtual console device on the first serial port with kernel debugging enabled

II. Concurrency Assignment

Write-up

Our concurrency solution implemented pthreads and multithreading to solve a producer/consumer style problem. The problem: with a buffer of size 32, have producers add items (structs with int value number and int value wait time) to the buffer. The producers must wait a random number of seconds (time to produce) and also wait if the buffer is full. The consumers take the items from the buffer, sleeping for the time given in the buffer item struct's wait value, and blocking if the buffer is empty. To do this we implemented mutexes, pthread conditions, and concurrent programming.
(Brian and Daniel) Created group directory at recitation April 11th at 11:00am (Brian) Ran script for folder permissions on Friday April 14th. Daniel and Brian met in OSU Library at 1:50 pm on Monday April 17th. Worked until 3:30pm, (Daniel) Committed the concurrency1.c file to github and pushed changes on Thursday April 20th. (Daniel and Brian) Tried to run concurrency1.c on os-class, found compile errors and rdrand error Thursday, April 20th. (Daniel and Brian) Implemented mt19937 function into concurrency1.c so that random numbers could be generated on os-class. Pushed to github. Thursday, April 20th. (Daniel and Brian) Worked on LaTex Makefile and formatting for Assignment1 repo (Brian) Transposed google doc file into LaTex format for submission Friday, April 21st. (Brian) Reconfigured Makefile to compile both LaTex submission file and concurrency1.c file Friday, April 21st. (Daniel) Wrote abstract and answered concurrency write-up questions for submission Friday, April 21st. (Brian) Reformatted concurrency1.c according to kernel submission format guide Friday, April 21st. (Brian) Provided class file for Tex styling Friday, April 21st. (Daniel and Brian) Pushed final changes to github for submission Friday, April 21st.

Our producers, once the thread is created, get a random number using either rdrand from x86 assembly instructions or the implemented mt19937 function and sleep for that amount of time (the time it takes to produce). When done sleeping the thread will lock the mutex which gives it exclusive access to the buffer. It then enters a loop to see if there are any available indexes in the buffer. If the buffer is full the producer thread waits for a consumer to throw the pthread condition for "full" meaning a consumer has opened up a space on the buffer. Once it is able to add its item to the buffer it throws a pthread condition "empty" to signal that the buffer is no longer empty, prints out the number, waits the time for the item it is adding, unlocks the mutex, and exits.

Similarly, our consumer threads lock the mutex and enter a loop checking for an item in the buffer to consume. If the buffer is empty it will wait for a producer thread to throw a pthread condition that the buffer is no longer empty. Once there is an available item in the buffer the consumer stores its number and wait time in local variables and sets both struct fields to -1 (implying it is now an empty space) and unlocks the mutex. The consumer then waits the given amount of time, prints a message that it has consumed the specified number, then exits.

Our main function mallocs enough size for the buffer and thread arrays, checks to see if rdrand is available to use, and initializes the mutex and pthread conditions. It then uses for loops to create and pthread_join all the threads and wait for them to exit. The pthread condition variables allow the threads to synchronize by locking and unlocking the mutex until the specified condition is satisfied.

The program accepts a command line parameter as the number of threads to produce. If the program is executed with `./concurrency1 100` it will create 100 producer and 100 consumer threads. On our own local machines this program works for N $>$1000 threads, but on os-class it segfaults after about 240 threads, which we assume to be due to limitations on server permissions to mediate the amount of threads a single user can produce.

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

This assignment was designed to introduce basic concurrency programming and get us familiar with pthreads, mutexes, and the basics of concurrent implementation. We were also introduced to inline assembly which we will use in our programs throughout the course.

How did you personally approach the problem?

We went about the producer/consumer problem in a fairly straightforward manner, implementing a function to produce an object and a function to consume an object. Each function was used as the start routine for the respective pthread and communicated via pthread conditions and mutexes for access over the global buffer and synchronization. We used basic

modular arithmetic to get a random number (within a specified range), from rdrand or the mt19937 algorithm, which we used as the wait-time struct field for consumer wait-time and producer sleep time.

We also used two pthread arrays to keep track of all the producer threads and consumer threads. The first for loop in main created N number of threads and added them to the pthread arrays upon creation. The second for loop in main iterates through each array and calls pthread_join to wait for thread termination.

The buffer item struct has two int fields, one for the number to consume and one for the wait-time the consumer thread must wait in order to consume the specified number. We initialized the the buffer to empty structs with number and wait-time values of -1 and the consumer sets the item struct back to this default format once it is consumed.

How did you ensure your solution was correct?

We used a series of printf's to debug. We printed out the values produced as well as the number of threads being run. We could follow along with the program logic through these print statements watching how long threads would sleep for, when production and consumption was performed, and when pthread conditions were thrown and received. We also used a print buffer function which would run a for loop and print out the values of each item in the buffer. We used this to see when items were added and removed from the buffer, and when the buffer was empty or full. This helped fix a bug we had where the buffer was not originally emptying properly.

What did you learn?

We had never used pthread condition variables before to communicate between threads and synchronized. This was very helpful in our solution to share the mutex during buffer full and buffer empty conditions. We also learned how to use inline assembly to generate random numbers which was a new concept for us in C programming.

III. Version Control Log

Detail	Author	Description
6fc0cad	DSchroederOSU	Initial commit
237aed0	DSchroederOSU	Added assignment 1 folder and starter file
2c475d7	DSchroederOSU	added concurrency 1 file
4126204	DSchroederOSU	working on concurrency1.c. made a skeleton of the project
d862ede	DSchroederOSU	progress, got rdrand to work
a134ada	DSchroederOSU	finished concurrency logic with two threads
7096e2f	DSchroederOSU	DONE I THINK
29c5440	DSchroederOSU	This works for n threads, there was a bug last time that would not clear the buffer space once consu
c6ed2db	ozarowib	recitation program
ce1d4cd	DSchroederOSU	added second random generator
870a6cf	DSchroederOSU	Merge branch 'master' of https://github.com/DSchroederOSU/CS444-Group_11_05
17fa4d7	Daniel Schroeder	works on os-class
8efe181	Daniel Schroeder	test
f860768	DSchroederOSU	fixed argc
63f1ea3	Daniel Schroeder	tex progress
e3916b0	Daniel Schroeder	idk
bafbb73	Daniel Schroeder	ahaha
d8ae437	ozarowib	conformed to class style guide
c7bf34a	ozarowib	IEEEtran working, updated Makefile

IV. Work Log

(Brian and Daniel) Created group directory at recitation April 11th at 11:00am (Brian) Ran script for folder permissions on Friday April 14th. Daniel and Brian met in OSU Library at 1:50 pm on Monday April 17th. Worked until 3:30pm. (Daniel) Committed the concurrency1.c file to github and pushed changes on Thursday April 20th. (Daniel and Brian) Tried to run concurrency1.c on os-class, found compile errors and rdrand error Thursday, April 20th. (Daniel and Brian) Implemented mt19937 function into concurrency1.c so that random numbers could be generated on os-class. Pushed to github. Thursday, April 20th. (Daniel and Brian) Worked on LaTex Makefile and formatting for Assignment1 repo (Brian) Transposed google doc file into LaTex format for submission Friday, April 21st. (Brian) Reconfigured Makefile to compile both LaTex submission file and concurrency1.c file Friday, April 21st. (Daniel) Wrote abstract and answered concurrency write-up questions for submission Friday, April 21st. (Brian) Reformatted concurrency1.c according to kernel submission format guide Friday, April 21st. (Brian) Provided class file for Tex styling Friday, April 21st. (Daniel and Brian) Pushed final changes to github for submission Friday, April 21st.