Writeup 1

Alex Hoffer

CS 444
Spring 2017

Abstract

This first write up describes important details from the completion of Project 1 for D. Kevin
McGrath's Operating Systems II class. Topics from Project 1 that are to be covered in this work
include the building of the Linux Yocto kernel on Oregon State's engineering server, usage of the
qemu virtual machine, and a solution of the Producer-Consumer concurrency problem using the C
programming language's POSIX threads execution model.

Contents

1 Log of Commands to Build Yocto Kernel

put code here

2 Log of Commands to Load Qemu

3 Flags in the listed Qemu command line

The listed Qemu command line is:

qemu−system−i386 −gdb tcp::???? −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".

The following list describes each flag:

- *-gdb* tells the system to wait to connect to gdb on the device tcp::????, this device being passed in as an argument.

- *-S* is a flag that tells the system to wait for a gdb connection.

- *-nographic* is a visualization option that turns off graphical output.

- *-kernel* is a flag that saves time and resources by granting the system a Linux kernel image, rather than requiring a full boot.

- *-drive* handles disk options and takes 1 to n number of arguments to specify these options, in the case of this command the file we passed to this flag is parsed according to the virtio interface.

- *-enable-kvm* is a flag that enables KVM virtualization support without any setting turned off.

- *-net* is a flag that sets network options.

- *-usb* is a flag that turns on the USB driver.

- *-localtime* is a flag that tells the system to use local time.

- *–no-reboot* tells the system to exit, not reboot, when rebooting is an option.

- *–append* is a flag that represents the command line necessary to interact with the kernel.

4 Concurrency

The following subsections answer the four questions as outlined on the Project 1 page on Kevin McGrath's course website.

4.1 Main point of assignment

There were several reasons for the assignment. One, I think, was to re-familiarize us with POSIX Threads, which we learned about in CS344 but likely forgot about before we got to this class. Second, I think that it was to teach us about the Producer-Consumer problem, which is a canonical example of concurrency. Third, it was to force us to use inline assembly language in our programs to help us learn how to implement ASM in C.

4.2   Personal approach to problem

My approach for the use of threads came from guidelines to the canonical Producer-Consumer problem that I found in the *Linux Programming Interface* text. For example, statements like pthread create, pthread join, pthread t, pthread mutex t, and pthread cond t were all found in that book and that book instructed me on how to use them. I used these static pthread statements rather than dynamic ones because frankly, I could not think of a reason why dynamic ones would be necessary for the purposes of this assignment. My approach for the buffer and the item to be held in the buffer was just to use a struct to represent the buffer and a struct to represent the item and have the buffer hold a statically allocated array of pointers to items and for each item to have two data fields: a random number and a random waiting period, both that had to be implemented in ASM. Given the statements necessary to control threads and mutexes and the structs necessary to store data, I solved the problem by creating a thread to correspond to a Producer, having that Producer thread lock all of the shared data using a mutex and then produce a random item. After unlocking the shared data within the Producer thread, I created a Consumer thread that locked the data again, consumed the first non-null index in the buffer, and then unlocked the data again. I contained these Producer and Consumer thread creation statements in a for loop that runs the number of times the user indicates in a command line argument so that when the program is run it creates the corresponding producer and consumer threads and produces and consumes random data.

4.3   Ensuring solution was correct

To ensure that the buffer was properly being added/removed to and to ensure that both threads were properly locking and unlocking, I added print statements in important functions such as when the global mutex was locked and unlocked in order to know whether these calls were actually being made. It is easy to verify that the threads are switching back and forth because both threads have different print statements that are only accessed within their specific thread is running. It is also easy to realize that the buffer is only sequentially being accessed by the two different threads in accordance with the mutex locking and unlocking because otherwise both threads would be accessing the same resource and either a memory read/write issue would occur or both threads would be depending on each other for an inordinate amount of time, resulting in *deadlock*. Obviously, my use of the pthread condition functionality allowed both threads to wait until resources were available, thus preventing deadlock. Finally, to ensure that the entire integrated software worked properly, I used a makefile to generate an executable of the code that accepts 1 argument that corresponds to the number of times a loop that creates the threads shall occur. To ensure the executable achieved what I wanted it to, I ran the software with command line arguments of 1, 2, 3, 4, and 5. The corresponding output for these arguments properly demonstrated that the different threads were called 1, 2, 3, 4, and 5 times, respectively, which is correct behavior.

4.4   What I learned

I learned more about POSIX thread functions, such as initializing them, the act of "joining" them, and locking resources down using a mutex to prevent threads from accessing the same data simultaneously. I also learned more about implementing in-line ASM into the C programming language, a skill that could pose useful for software projects that require maximized efficiency in the context of low-level programming. I found the in-line ASM to be complex and confusing, mainly because of a) the odd syntax and b) how it required my brain to be thinking in both ASM and C at the same time.

5   Version control log

History of versions

Version	Date	Author(s)	Changes
1	4.7.2017	Alex	Add latex template file
2	4.9.2017	Alex	Created structs to hold the data, inspect how to do the ASM randomization
3	4.9.2017	Alex	Add fake random functions to stand in place until ASM is understood
4	4.9.2017	Alex	add pseudocode for producer, consumer void functionswhich are passed to pthreads
5	4.10.2017	Alex	First pass at producer function done with mutexes, conditions, etc.
6	4.10.2017	Alex	Everything done except ASM randomization, currently prog loops forever, only terminating when hit with a signal
7	4.17.2017	Alex	change name of proj1 folder to project1
8	4.17.2017	Alex	Add part in latex about Qemu command line, flags explained
9	4.17.2017	Alex	Answer first two questions of the concurrency writeup
10	4.18.2017	Alex	Add loop counter accessible from command line
11	4.18.2017	Alex	Changed all typedefs of structs to just structs in accordance with Linus's coding style
12	4.18.2017	Alex	Add what I learned and how I ensured solution was correct sections to writeup
13	4.18.2017	Alex	Begin work on version control table

6   Work log