CS 444 - Operating Systems II- Assignment 1

Christian Mello, Daniel Stewart, Shengpei Yuan

Group 12-06

Oregon State University

Computer Science 444

Spring 2017

Abstract

This document outlines assignment 1 for Operating Systems II. There are two parts to the assignment, setting up a Qemu based Yocto environment, and a concurrency exercise using producers and consumers.

I. Concurrency

The main point of this assignment was to familiarize ourselves again with the concept of both concurrency and thinking in parallel. The problem calls for the creation of multiple threads, which all carry out one of two tasks, either producing or consuming. This means we have to think about what a consuming thread is doing when there is nothing to consume, and what threads are doing when another thread is either adding something or taking something away from the shared buffer.

We approached the problem by first dealing with the generation of random numbers. As the problem states, it needs to use both rdrand x86 ASM when supported, and Mersenne Twister when not. This is done with an if statement in the producer and consumer functions themselves. Each one calls another function, which returns a 1 if it can use ASM, and a 0 if not. I suppose this could have also been done off a global variable, but this way works. After this we implement the buffer to hold 32 objects at time, which are random integers, along with a lock and a condition for empty and full. We then only made 2 threads to start, one each for consume and produce. The contents of the buffer get printed out after each thread action, along with the time that they sleep for. The threads will also lock any other threads out of using the buffer while they are. A thread trying to use the locked buffer will just wait until it becomes available again. After making sure that 2 threads worked, implementing more wasn't hard, as it just required making an array to hold each thread. We also added a command line input variable for the number of threads desired, which can't be less than 2. It splits the desired number into consumer and producer threads using the mod operator.

There wasn't a lot that we thought needed to be done to make sure that our implementation was running the way it was intended to. It prints out the wait time for each thread, and the numbers in the buffer after each thread access it. So, running the program with different amounts of threads we can see what is going on. Specifically if the program is run with an odd number of threads, like 3. Since this creates 2 producer and 1 consumer thread, it tends to fill the buffer up fairly quick compared to only 2 threads, since 2 threads are adding new integers to it. We also ran it on our own machines, which should use rdrand, and on os-access, which won't be able to use it, and it works for both.

The things learned during this assignment were mostly ones forgotten from Operating Systems I, or that hadn't been used in a while, such as setting up pthreads and concurrency problems.