A Synchronization Mechanism of Prototype OS

Deverick Simpson & Yuekun Yang
Computer Science Department, University Of Nebraska-Lincoln

December 18th, 2014

1 Introduction

To continue our understanding of the OS kernel, the objective of this research project was to provide race-free execution for multiple producer/consumer threads through implementation of semaphores. Furthermore, the implementation will be embedded atop the Prototype OS that has been in development through the semester. The instructor solution for Prototype OS will be used to integrate the semaphore type. To avoid compatibility issues between machines, the programming environment will primarily be within the CSE server on campus.

2 Project Management

The completion of the projected consisted of the implementation of the semaphore type as well as this project report. It was agreed that implementation of the code for the semaphore type will be headed by Yuekun while the report will be completed by Deverick. The decision was based on previous assignments history, with Deverick completing the implementation of the program as well as the report. Deverick made himself available to discuss the theory of the project as well as debugging. This proved to be a risk as Yuekun was not familiar with running a program on the Nios II board. The development of the majority of the code was initially complete in the first few weeks

however no testing or attempt to run the program had been done on the program until the last week. Fortunately, most of the errors and bugs were resolved within a day.

Strategy for creating a solution included using research references such lecture powerpoint slides as well as speaking with classmates on a high-level on the difficulties and odd errors that arose during development. After careful review of the documentation, a few tasks were deemed important for the successful implementation of the semaphore type. Those are:

- **Create functions, mysem_create, mysem_up, mysem_delete, etc., outlined in documentation**
- **Recode previous thread initialization to create producer/consumer threads with specified criteria**
- **Add additional queue structs to contain blocked threads**
- **Display detailed information concerning whether the operation on buffer is successful or blocked unblocked on a semaphore**

3 Summary

To accomplish this, it was important to first understand the producer/consumer problem a bit further. After researching the problem, it was important to grasp the concept of a semaphore and its role within a multi producer/consumer thread application. Furthermore, it was important to understand the requirement develop the blocking queue and buffer to contain the current running process. Producer and consumer threads were created using the previously developed mythread_create of project 2. The mutex and semaphore locks occur within the producer/consumer thread created, allowing for complete atomicity of resources. A depiction of this can be seen in Figure 1. To enter the critical section, the consumer/producer must decrement its respective semaphore lock to

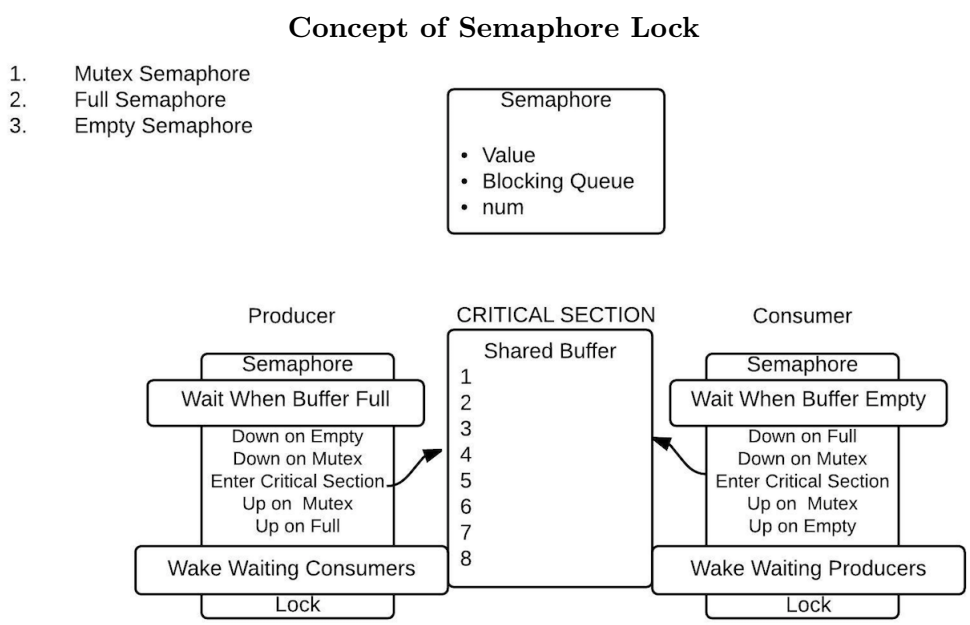

Figure 1: *To enter the critical section, a producer/consumer must obtain a lock on its respective semaphore. After leaving the critical section, the producer/consumer releases its lock.*

enter the critical section. Once within the critical section, the producer/consumer will complete its behavior of inserting/removing the char 'X'.

In this project, most implementation of the semaphore is complete. There are currently a few bugs preventing the threads from executing appropriately. At the moment, threads initialize and execute appropriately until reaching thread 5, after which data is corrupt. The debugging process involved reviewing code and setting breakpoints to analyze data at specific points in time.

Overall, the project is fairly clear and concise. The documentation provides most materials useful for the project. Outside of the given documentation , little research was required to complete the project. For future semaphore implementation projects, a similar documentation could be provided with little hesitation. The project provides great emphasis on the principles studied within the course by reenforcing the concepts of data race and mutual exclusion, often required in today's technologies. Furthermore,

the development of the race-free execution of producer/consumer threads is significant as it enables more efficient execution of threads than previously simulated models.

The implementation itself was fairly simple as most of the code had already been previously written. The fact that this project was a continuation of the last projects eased the development process as the material was fairly familiar. Often during courses, the mindset must swiftly switch from one concept for development to another. Coding practices continued on from the previous implementation by using pointers and abstracting data. Comments were also used to highlight important and/or confusing code that required a bit more elaboration. The overall organization of the project is far better than the previous assignment as the instructor source code was used as Prototype OS. This source consisted of .h header files as well as the .c files. Previously, our code was compiled into a few files. It was a lot easier to track the flow of the program with .c struct files separate from main function .c files.