COP 4610

Homework Assignment 3

Alejandro Vidal

PID 5913959

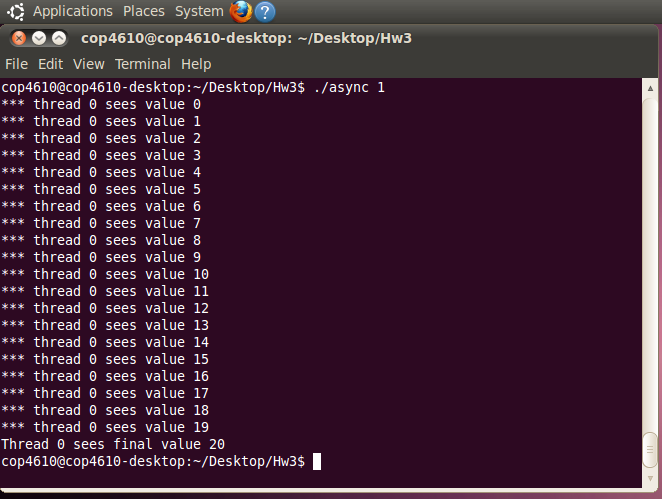
FIU

Fall 2017

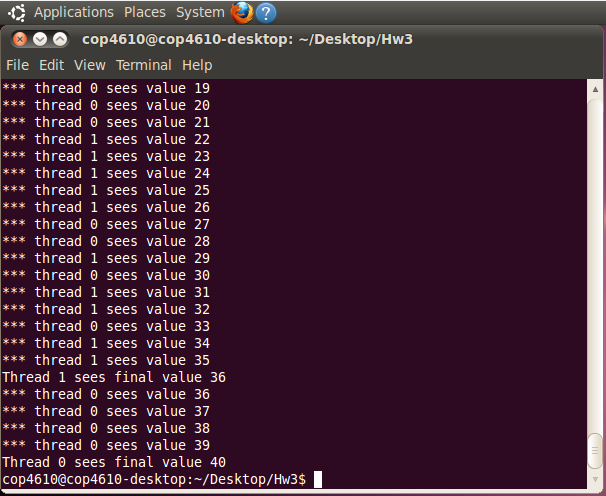
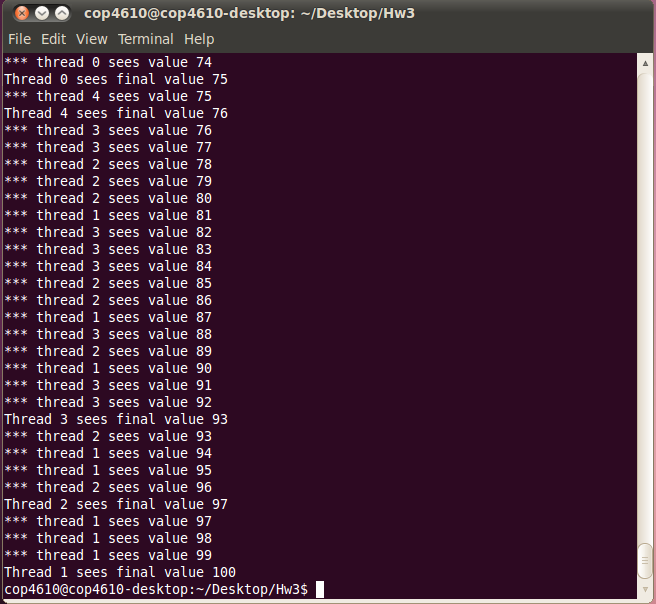
**Part 1:** **Simple Multithreaded Programming**

**1.1) Simple Multithreaded Programming without Synchronization**

When running the simple thread program passing 1 as a parameter it can be seen that the shared variable is modified successively by the created thread. In this case, there’s no concurrency issue since there’s only one thread modifying the global variable. The result of running the program with one thread can be seen below.



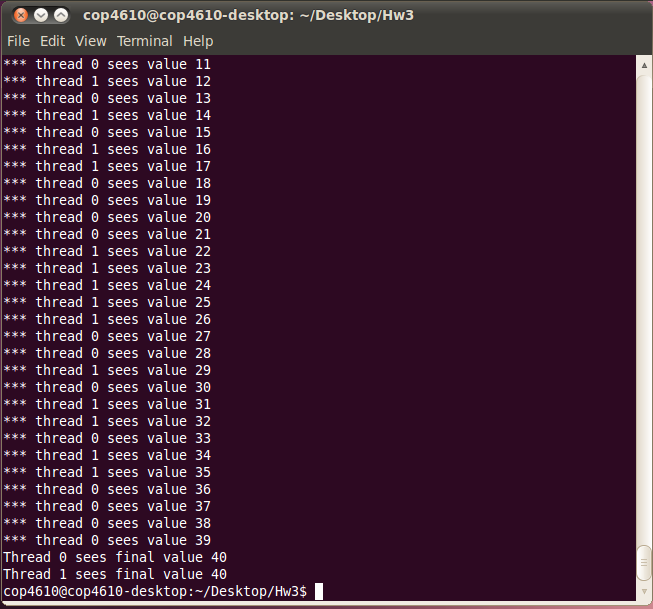
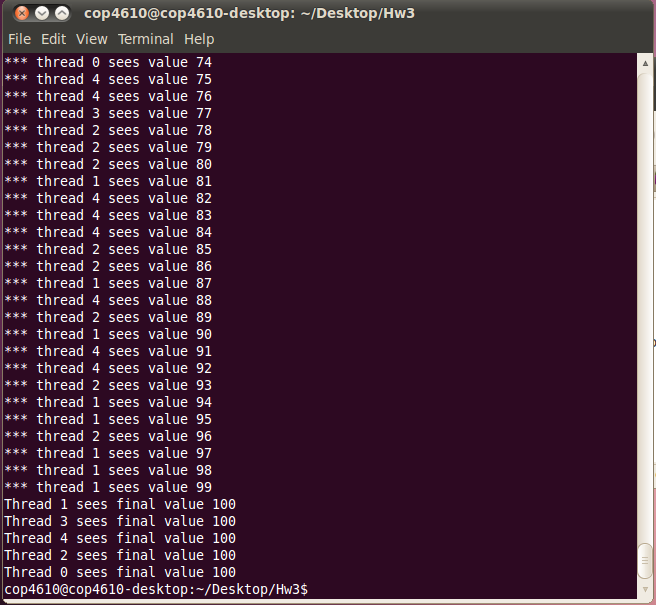
However, when running the program with a number of threads greater than 1, issues arise. Since the created threads are accessing the shared variable without any type of concurrency control, the may access and modify the shared variable concurrently, causing all the threads to have different final values for the shared variable. This happened when running the program with 2, 3,4, and 5 threads. The results of running the program with 2 and 5 threads can be seen below.



(2 Asynchronous Threads) (5 Asynchronous Threads)

**1.2) Simple Threads Programming with Proper Synchronization**

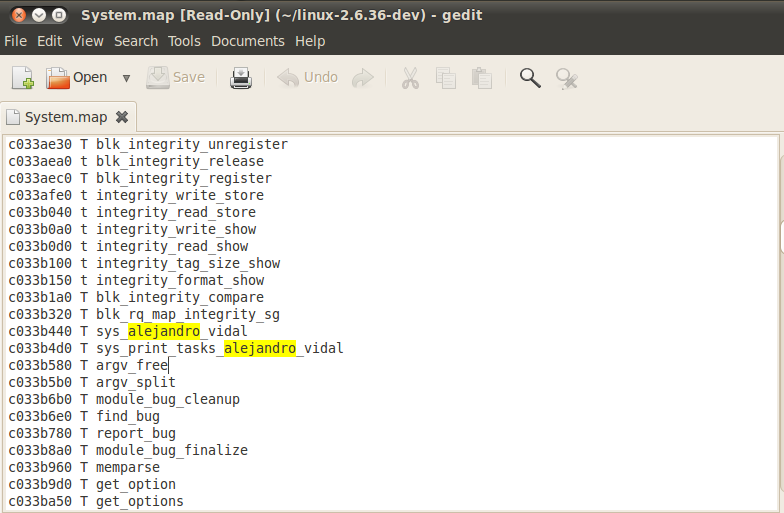
When adding a mutex variable to the program in task 1.1, the results are different. The mutex allows for synchronized access to shared data, and using one every time the shared variable is going to be modified guarantees that each iteration of the loop increments the shared variable by one, and that all threads see the same final value for the shared variable. Each thread, tries to acquire a lock every time it’s going to modify the shared variable, if some other thread has already acquired the lock, it waits. This was verified by running the program with 1, 2, 3, 4, and 5 threads. Results of running the program with 2, and 5 are shown below.



(2 Synchronic Threads) (5 Synchronic Threads)

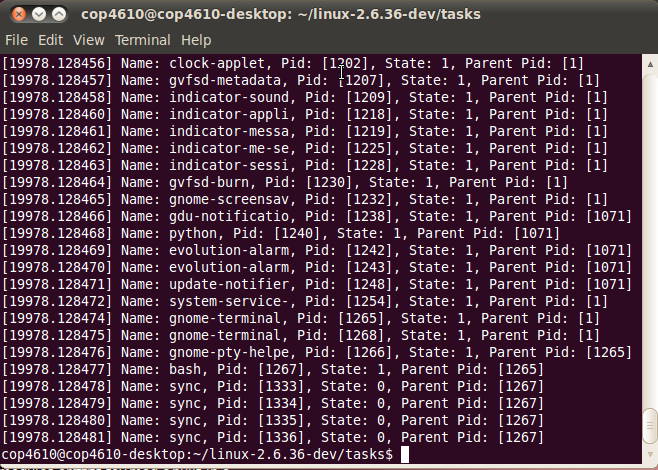
**Part 2: Inspect Tasks in Kernel**

Now I created a new system call (pint\_tasks\_alejandro\_vidal) and add it to the kernel. We can see it in the system.map file along with the previous assignment system call.



I created a new testmycall.c program to test my new system call. At user level the output of this test program is the same as the one used in Part 1.2 of this assignment including the returning status of the system call.

We can use **dmesg**  to see all the information regarding the processes and the threads.



We can see at the end, our **sync** process and all the threads information required.