**Operating System Part 2**

**Austin Blythe, Matthew Francis**

**Account: cs441102**

**20 February 2013**

**Program problem/statement:**  
For concurrency, there needs to be mutual exclusion and synchronization enforcement and deadlock and starvation avoidance. In order to develop and explore these operating system issues, we will need to modify the scheduler and dispatcher from part 1 to create part 2. Continue to work in teams of two or three.  
 **O/S** **Architecture:**

* 16-bit words
* 256-word memory (16 bits) word addressable
* 3 general purpose registers (1-3), 1 special accumulator (A; 0)
* 8-bit program counter (PC)
* condition code flags: = or zero(010), > or positive(001), < or negative (100)
* machine instruction cycle - fetch, decode, opfetch, execute, writeback

**Difficulties encountered:**

* We had some minor difficulties conceptualizing the idea of the semaphore and process queue, but we eventually came to understand it better and decided on an implementation that turned out successful.
* Because we did not implement certain organizational features from part 1, implementing part 2 was slow in the beginning.
* Compiling across different platforms (UNIX, Mac OSX, and Windows) occasionally caused different functionality. We attempted to normalize our compiling be using the same compiler consistently.

**Procedure:**  
 We began with a basic machine language interpreter from part 1. The first thing we did was add the scheduler functionality for multiple users (round robin). After we did this, we implemented the process “interrupts” after each user executes four “ticks”. The final, and most time consuming, implementation dealt with the memory locking feature and the process queue.  
**Additional Observations:**  
 We got a little bit of a late start on part 2, and we did not have the proper O/S organization in place from part 1. These things caused us to rush a bit towards the end and cut it a bit too close time-wise.

**Assumptions:**

We assumed that each user may only have one process in the queue at any given time. For example, if user 1 is running their program, and user 2 sends “run, user 2 gets a new process added to the queue. If user 2 tries to send “run” again, he will be blocked because he has a process waiting already.  
  
**Detailed table of work:**  
We spent about a week on this part; we met in the lab about 4 times.

|  |  |  |
| --- | --- | --- |
|  | **Austin** | **Matt** |
| **2/11/2013** | 2 hrs | 2 hrs |
| **2/12/2013** | .5 | 1.5 |
| **2/13/2013** | 2 | 2 |
| **2/14/2013** | 1 | 2 |
| **2/18/2013** | 2 | 2 |
| **2/19/2013** | 2 | 2.5 |