Adam Worthington
Lab 2

Section 3:

What happens to the victim process after myattackermalware() finishes, i.e., returns?

- The victim process does not get to continue execution due to there being no valid stack setup once the malware has finished execution. This causes Xinu to hang indefinitely after the malware has printed myvictimglobal = 1.

Section 4:

Modify XINU to enable monitoring CPU usage of processes. Describe your implementation in Lab2Answers.pdf.

- For my implementation I firstly required a couple new fields in the process table. I added prcpuused as instructed to track the total time a process has been active. I also added a prcpustart variable to track the timestamp every time a process becomes active for bookkeeping sake. Next over in resched I immediately take a timestamp before and context switching may occur. Afterwords I do a basic calculation on the amount of time the current “old” process has been used within its last iteration and update the prcpuused variable accordingly. To ensure the next “new” process has sufficient information for the calculations mentioned above, I record its prcpustart time immediately before it is context switched into.

Re-run the test cases of Problems 4.1 and 4.3 from lab1 with the kernel's monitoring facility. Compare the results of CPU usage from measurements to observations that you advanced in lab2. Discuss your findings in Lab2Answers.pdf.

- The results with respect to the ordering are the same as I had in lab one. Here is a copy:
  - 4.1: PAPBPCPDABCDABCDABCDABCD
  - 4.3: PAPBPCCCCCPDDDDDABABABAB
- I also ran the code to print out the time each process had taken and received the following results:

4.1 process time records		4.3 process time records	
Process	Time in ticks	Process	Time in ticks
A	1	A	1
B	1	B	1
C	1	C	1
D	1	C	11
A	11	C	21

B	11	C	31
C	11	C	41
D	11	D	1
A	21	D	11
B	21	D	21
C	21	D	31
D	21	D	41
A	31	A	11
B	31	B	11
C	31	A	21
D	31	B	21
A	41	A	31
B	41	B	31
C	41	A	41
D	41	B	41

- This output is expected to be the same as the output for lab 1 due to the fact that we have made no change to the scheduler yet. We have simply recorded each processes tot cpu usage which will not have an effect on priority.

Section 5:

DISCLAIMER: For the following results I tested be code with multiple values for memcpy loops and sleep times. The output I recorded below are a general conglomerate based on the overall patterns I have recognized by playing with the values. You may receive different results depending on the values.

Note, that XINU's *null process* should only run if there are no ready processes in the system. Implement a design choice that makes it so and describe your solution in Lab2Answers.pdf. In the write-up, discuss the output that you observe and whether they indicate fair sharing of CPU cycles by the 4 processes. You will need to experiment with different values of *LOOP1* and *LOOP2* to induce CPU sharing via context switching and resultant output that can be easily interpreted for gauging your implementation of TS scheduling performance.

- Firstly I handle the NULL process only running if the ready queue has no other processes by assigning its priority(prcpuused) to the maximum allowed value as defined by xinu. This ensures that it is always enqueued in the last possible position of the ready queue's new implementation.
- Secondly for 100% cpu intensive processes the new queue system is fair, assuming all of the processes are of equal importance initially. When executed the four processes take turns running in a cyclic order until the loops end. This is due to the queue system and can easily be explained visually:
    - Process 1 executed and now has run for 1x time while processes 2, 3, and 4 have ran for no time, or 1 time. This means that process one is now enqueued behind processes 2, 3, and 4 in the queue.
      1, 2, 3, 4, NULL -> 2, 3, 4, 1, NULL
      This cycle continues until all of them have executed once through and our queue is now back to its original state:
      1, 2, 3, 4, NULL
      Now all of the processes have ran for 1x time excluding NULL which has been initialized to the maximum allowed value. Now that they all have equal value again the cycle continues until the loops reach their terminating conditions.

When testing, use the same sleep time as argument to sleepms() for all 4 instances of I/O-intensive processes. In Lab2Answers.pdf, include a discussion of the results observed and your assessment of fairness by the dynamic priority scheduler.

- Again, for all intensive purposes I am assuming the processes are of equal importance here. In this case the current scheduler if fair for 100% I/O processes. The explanation follows every closely to that in which I provided for cpu intensive processes. The queue is initially filled and each process executed in turn for the first initial cycle. The difference is that each process runs for a significantly smaller amount of time than that of the cpu intensive processes. This is because whenever the processes sleep (simulate I/O), it is not in the ready state and thus does not have its time recorded/updated while it is sleeping. Regardless the queue if filled and emptied in the same cyclic fashion as before. Each I/O process runs for 1x time during execution and is then at a lower or equal to (non increasing) priority (higher prcpuused) than the other three (this is the nondecreasing queue implementation mentioned earlier).

In the first part of the evaluation, under this mixed workload of CPU- and I/O-intensive processes, determine if the 2 CPU-intensive processes --- among themselves --- achieve equal sharing of CPU cycles as indicated by the output. Do the same for the 2 I/O-intensive processes with their sleep time arguments to sleepms() fixed to the same value. Evaluate CPU sharing between the two groups of processes --- CPU- and I/O-bound --- and discuss your findings in Lab2Answers.pdf.

- With the mixed processes trying to share the CPU we can easily tell that our current implementation is not fair. It is biased towards the CPU intensive processes, however it favors executing I/O processes when available. Here is an examination of the execution.

- Firstly the processes are all enqueued with equal priority, thusly they are all run through one iteration each.
- The second iteration starts to explain why this system runs the cpu intensive so much more than the I/O bound processes. While the I/O are sleeping they are not in the ready state. Our queue system requires for each process to be ready in order for it to grab the CPU. This means that while the I/O processes are sleeping the CPU intensive processes will execute as many times as they can before a wake up by the I/O process. An example of this could be that our two CPU intensive processes have a prcpuused value of 10000 while our I/O processes have a prcpuused value of 10, however while the I/O are sleeping the CPU intensive processes will continue to execute, only to be interrupted briefly by the I/O bound process awaking and iterating its loops only to go back to sleep.
- The system gets a bit finicky with times here. If you have the I/O intensive processes sleep for too long, only the cpu intensive processes will print. If they sleep for too little they will be the only ones to print due to the fact that their execution time is so small that their priority remains lower than the cpu intensive ones throughout execution.
- Overall at surface level this system isn't all that bad. You want your cpu intensive process to be grinding away until an I/O interrupt, at which point our current system prioritises the I/O due to their drastically lower prcpuused values. The problem comes in when processes are not all initialized together. Our system will ignore any older processes in favor of newer processes due to the nature of our priority queue. This is a clear flaw as you would not want your hardware to become unresponsive a few seconds/minutes after you boot your system.

Bonus:

The above version of fair scheduling has an obvious flaw in that a newly created process, compared to existing long-running processes, will receive elevated priority for a prolonged period during which the long-running processes may starve. Our benchmark scenarios in Problem 5 do not encounter this issues since the concurrent test processes are created close together in time. What might be a solution that mitigates this problem? Describe your solution and argue its efficacy in Lab2Answers.pdf.

- My solution is based on information I have read on O(1) scheduler. Here is the primary source that I drew information from:
    - http://people.cis.ksu.edu/~gud/docs/ppt/scheduler.pdf
- In short to solve the current problem of process starvation you could simply implement a secondary priority queue that would ensure each process it given some ruin time while keeping the ability to prioritize certain processes over others. To implement this I enqueue elements to the secondary array once they have been dequeued from the primary one. Once the initial array has been depleted of elements the pointer would switch over to the new array and resume normal operation. This would eliminate complete starvation, but is not a perfect solution either. While you can have different

priorities you may experience problems with all of your I/O bound processes running at the beginning of your array and having to wait until all of your cpu intensive processes are finished executing before they get a chance to grab the CPU for I/O again. This means that there will be (QUANTUM * number of CPU intensive) milliseconds after all of the I/O processes finish before they can be called again.

- With respect to efficiency this implementation runs between O(1) and O(n) depending on the implementation. The reason being is that if there is no real comparisons in the queues, the enqueueing and changing of the pointers all run under the O(1) time restriction. You can run into some problems with time complexities if you perform comparison operation like we do with our non decreasing order implementation. Dequeuing and switching the array will run in O(1) time, however the comparisons in the enqueueing section will be under the time restriction of O(n) time due to the fact that they may be inserted at the end or beginning of your priority queue.