Problem 3

When myattackermalware() returns, it returns to where the function originally returned, since in myattacker the original function return value is set to myattackermalware(), and then myattackermalware() is set to return to the original return value. The victim process continues to execute as normal, and eventually returns to main as intended. If myattackermalware() return value is not set to anything in myattacker, then it has a return value of 0x0, and does nothing, causing XINU to break and become non-responsive. Myattackermalware() pid is 3, the same as the victim pid because it is being run by the victim process.

Problem 4

In my implementation of monitoring CPU usage, when I process is switched in, the value of clktimefine is stored in a global variable called clktimeswitch. When the process is switched out, the difference between clktimeswitch and clktimefine is added to prcpuused.

When rerunning the test cases for Problem 4.1 from lab1, each process that called printloop() took 28-29 milliseconds of cpu time, and completed in the order ABCD. For Problem 4.3 from lab1, each process took between 26-30 milliseconds, and completed in the order CDBA.

Problem 5

I implemented the dymanic priority scheduling by modifying the insert function to check if the queue key was less than or equal to key to be inserted, and then changing the 3rd argument in all calls to insert from the priority to the cpuused time of the process, which was initialized to 1. The Null process is only run when no other processes are ready, and this is done by initializing its cpu used time to a large value.

When running cpuintensive(), all processes used the same amount of cpu time (21 milliseconds), which shows the fairness in sharing of the 4 processes. When running iointensive(), the processes all were within 2 milliseconds of each other, a range of 13-15 milliseconds. However the processes did not stay in their initial order, due to the prcpuused not being set precisely on the beginning/end of a millisecond. Overall it is still a fair scheduler, scheduling the ones it perceived to have fun less ahead of those which have run longer.

For mixed, 2 cpuintensive and 2 iointensive processes, the dynamic scheduler was fair to both cpuintensive and iointensive individually, cpuintensive both taking up 23 milliseconds, but bouncing back and forth between the two, and iointensive taking up 13 and 14 milliseconds. Sharing between the 2 groups would be considered “fair” in that the cpu intense processes were given priority while the io intense process were “waiting” for a response, and thus finished in the most efficient and fair way.

Bonus

A solution that mitigates this problem would be to allow the newly created processes to run for a set period of time and then switching it to the next newly created process. This time would be long enough to prevent the long running processes from starving.