3.1:

First, in wrongturnIIa(), I check to make sure that the given process is in the suspended state and is not a newly created process. I do this by ensuring that the prstate process table field is equal to PR_SUSP. I ensure the given process is not newly created by checking that the element in the stack above ctxsw()'s return address is not equal to the userret() function pointer. I then increment the prstkptr process table field by 10 to make sure it is at the same location as the return address of ctxsw(). The return address of ctxsw() is saved in the global variable origretaddr, then overwritten to be equal to malwareIa. In malwareIa(), to implement the detour correctly, I preserve all the registers using inline assembly with the pushal instruction. After printing the PID of the process and "MalwareIa success", I change the value of the element in the stack directly above EBP (ESP) to the original return address of ctxsw(). To ensure the victim's stack is left in the same state as before the malware attack, I use the popal instruction to restore all the registers to their original values before finally returning.

4.4

In detourguide, I first preserved the stack by pushing EBP, moving EBP into ESP, pushing the eflags register, and pushing all the general-purpose registers onto the stack (all the usual CDECL requirements). Detourguide then calls the callback function which I stored in a global variable. Next, detourguide overwrites the address right above EBP to the original return address of clkdisp(). In doing so, detourguide will jump to the original return address of clkdisp() after executing ret. Lastly, detourguide restores the general-purpose registers, the eflags register, and changes where ESP points to by 4 bytes, so it is now pointing to the correct location on the stack. This work is very similar to the work done in 3.1 because in 3.1, the extended inline assembly also had to overwrite a specific address in the stack with a previously saved return address. Also, both functions required saving and restoring the stack after overwriting that specific address for the detour to function correctly. Even though detourguide required multiple more assembly instructions, the overall logic and work done between parts 3.1 and 4.4 were very similar.

4.5

To test and verify that my implementation worked correctly, I created and ran a test process that calls registerxcpu(). After calling registerxcpu(), the test process has a block of code containing a loop that runs for a specified amount of time. I made sure the while loop ran for longer than the specified cpu limit in the registerxcpu() call. I used a stopwatch to check when the callback function (in my case malwareI()) would execute its print statements. For example, if the cpu limit I set was 5000 (5 seconds), I made the loop run longer than 5 seconds. I then used a stopwatch when running my code to ensure that the callback function appeared around 5 seconds after XINU began running. I did this same process with many different cpu limit values, and the callback function always printed right after the cpu limit was reached. If the cpu limit was never reached, the callback function never executed or printed anything. I also had a test case where I called registerxcpu() a second time. When registerxcpu() was called a second time, the function returned SYSERR which is exactly what it was supposed to do in that situation.