3.1:

- We must use the pointers of ebp and esp (the frame pointer and stack pointer respectively) of the newly created process. We can't use the values of ebp and esp output by inspectnull(). By using the pointers of the newly created process, when context switching, ctxsw() will return to the new process rather than the old process. Without putting the correct values of ebp and esp in the registers, CDECL will cause the current process to return to resched() which will in turn return to the old process. Since we are context switching to a new process, we have to use the correct pointers corresponding to ebp and esp of the newly created process. As for all the other registers, if they are needed by the newly created process, they will be overwritten by the process, so we do not have to modify any registers aside from ebp and esp.

3.2:

- When userret() completes its work, the stack memory of the process will be effectively cleaned up, and the process will be freed meaning it will no longer run. When resched() returns, it will not return to userret(). Resched() will pick a different process to run while the original will not run again.

4:

- To test part 4, I first printed the CPU usage using xcpuutil(void). I then created many processes that call kprintf() and printed the usage again. By comparing the usage before and after calling kprintf(), I was able to confirm that the kernel modifications were working correctly. To start the NULL process's CPU usage measurement, changes must be made to sysinit() in initialize.c. When initializing the NULL process entry in the process table, the prcputicks field must be set equal to 0.

5:

- I did not deviate from the default values of IOSLEEP and CPUUPPER.

A. Cpubound: 4 1500 750 0
Cpubound: 5 1500 750 0
Cpubound: 6 1500 750 0
Cpubound: 7 1498 749 0
From left to right, the numbers are the PID of the process, the total CPU ticks, the number of time slices depleted, and the amount of times the CPU was voluntarily given up. These numbers all make sense because they are all similar between each process, there is a lot of total CPU ticks, the processes depleted their time slice every time, and the processes never gave up the CPU voluntarily, as CPU bound processes should behave.

B. Iobound: 4 80 0 20
Iobound: 5 80 0 20
Iobound: 6 80 10 20
Iobound: 7 89 9 20
From left to right, the numbers are the PID of the process, the total CPU ticks, the number of time slices depleted, and the amount of times the CPU was voluntarily given up. These numbers all make sense because they are all similar between each process, there is very little total CPU

ticks, and the processes rarely depleted their time slice, as IO bound processes should behave. The processes also voluntarily gave up the CPU 20 times each.

C.
```
cpubound: 5 1959 42 0
cpubound: 4 2258 42 0
iobound: 6 171 0 19
iobound: 7 171 0 19
```
D. From left to right, the numbers are the PID of the process, the total CPU ticks, the number of time slices depleted, and the amount of times the CPU was voluntarily given up. These numbers all make sense because they are all similar between the same types of processes. For CPU bound processes, the behavior is indicative of a CPU bound process (high CPU ticks, time slice depletion for every process, and never giving up the CPU voluntarily). For IO bound processes, the behavior is indicative of IO bound processes (low CPU ticks, no time slice depletion, and voluntarily giving up the CPU).

E.
```
cpubound: 4 1360 17 0
cpubound: 5 1362 18 0
cpubound: 6 1360 17 0
cpubound: 7 1280 16 0
```
The output is somewhat different than what I expected. I expected the total CPU ticks of the first two processes to be much higher than the total CPU ticks of the other two processes, but the total CPU ticks for all the processes are relatively similar. I also thought that the amount of times the time slice depleted would be greater for the first two processes, but they are about the same among all the processes.