Part 4 explanation

Testing:

For the test case, I relied on the one given in the lab handout. What this does is register a callback function childcb to main, and then initialize a child process to main. I named my child process do_nothing(). Since a child process executes the callback function registered to its parent, when do_nothing() completes it should run callback function childcb().

Implementation:

The callback function pointer is added as a field to the process struct named prcallback. Additionally, I created a global variable named callback_glbl in cbchildregister.c. cbchildregister() registers the callback function for a certain process, ie the callback function that the process's children will run. Since it can only be run once, first I check whether the process already has a process registered. If it does, I return SYSERR, otherwise I set the process's prcallback field to the passed in function pointer.

Because I only knew how to access global variables from assembly, in kill(), right before needing to execute the callback function for a killed process, I check the process's parent's prcallback and set callback_glbl to it. Then I change the assembly in clkdisp.S as is mentioned in the handout. The original order of the stack when clkdisp.S is run is: EFLAGS, CS, EIP, general registers. I modify it to be in the order: EIP, EFLAGS, CS, callback function, general registers. I use a helper global variable called eip_global to temporarily hold the value of EIP and simplify the work of changing the order of the stack values. When we call popal it restores the state of the registers I changed (only EAX) when moving around parts of the stack, and then when we call iret, it executes the callback function and returns to wherever EIP is pointing to.