Lab 6 Answers

Part 3

In order to handle the multiple signal issue I would implement a queue for each signal type so not only the order is kept, but if a signal is raised while one is currently being handled it will be enqueued rather than get immediately handled. The callback handler called by the kernel should check the queue after each signal of a type is handled, so if it is context switched in the middle of a callback execution, sent another signal, then context switched back in, it will still pick up the new signal.

In order to test a client-server model I made two applications that sent messages back and forth. When the "client" pinged the server, the server ran its callback, responded with the proper message, and was received by the client. All of my tests also worked with the XSIGCHL interrupt, as each child was received when the parent process was blocked on childwait(). For my XSIGXTM tests, I'm not sure how efficient it handles the signal as it does not instantaneously trigger the signal once time goes over. I don't think the functionality is that the process should immediately call the callback when the process goes over the time limit since there's no way to execute the callback handler without being in the kernel space at that time. However, all of my tests that tested the signal getting handled on being context switched in works fine.

Part 4

For my design, I essentially duplicated the free list implementation, but instead of adding memblk's to the list when they are free, I add them to a process' own used memory list. As memory is freed, I find the freed node and remove it from the list. Then, garbage collection becomes simple as all I have to do is check the process' used memory list inside kill() and call freemem() on those nodes.