Mark Yang

Lab 5: Signal Handling Subsystem and Memory Garbage Collection

Changes to XINU:
Added registercbsig.c, mysignal.h, alrmmywakeup.c
Cloned the queue again adding alrmmynewqueue.c, alrmmygetitem.c, alrmmyqueue.c, alrminsertd.c
Modified clkhandler.c, resched.c, kill.c, freemem.c, getmem.c
Added alrmPointer, alrmTime, xcpuPointer, xcpuTime, isALRMReady, and memoryBlocks to the process table
Added header files to xinu.h
Added appropriate functions to prototypes.h (at the very bottom)
Initialized appropriate fields from process table in create.c

1) As the lab mentioned, we are doing an extension/generalization of Problem 2 from the previous lab where all we have to do is create a registercbsig.c file that can handle 3 different signals: MYSIGRECV, MYSIGALRM, and MYSIGXCPU. MYSIGRECV stayed the same as the previous lab.

   For MYSIGXCPU, I handled this signal in clkhandler. Everytime clkhandler was called, I would check if the process had an xcpuPointer. If so, I would decrement the time until it reaches 0. When it reaches 0, I call the callback function.

   For MYSIGALRM, I created a new queue that would hold all the alarms. I used a delta list (alrminsertd) and would decrement the head of the delta list in clkhandler until it reached 0. Once it reaches 0, I would call alrmmywakeup which is like wakeup but for my alrmqueue. If the pid of the alrm that needs to be woken up is the currpid, then I can immediately run the alrm callback function. Otherwise, I would set the isALRMReady flag to true and unsleep and ready the process which will wait its turn to resched in. When resched context switches the process in, its alrm will be called very similar to how MYSIGRECV is called.

   The end results of my implementation were that MYSIGRECV the same as the last lab. MYSIGXCPU ran properly when cpu time was up. MYSIGALRM, if not the currpid, then it would have a very slight delay because we needed to resched and context switch the process that needed to run the alarm in.

2) For the garbage collection, I added a list to the process table to keep track of all of the blocks of memory that the list is using. To achieve that, I built a list of type memblk (which was predefined in memory.h) similar to a linked list in which every time the process calls getmem, I would also add that to the memoryBlocks list in the process table. I also added a gbheader to store the pointer to the next memory block in the list and also the length of the memory block.

   In freemem, I borrowed the code that iterates through a list to check for the block of memory that freemem is looking to free in the freelist inside of the memoryBlocks list of the process. If it exists, then I would remove that memblk from the memoryBlocks list of the process. However, if it's not found, then freemem is being called on the stack and therefore I would reset the block to the original block and also nbytes because I only touch the heap memory.

   I also modified kill.c because kill.c does not free all of the heap memory. I again iterated through the memoryBlocks list of the process and emptied out the entire list.

Bonus)

a) Kill the process because at this point, it would be the programmer's fault that they use so much memory
b) Set the priority very low so that it does not take over some other process'
c) Put the process to sleep until the process manually wakes itself up (force the programmer to call wakeup themselves to make sure their application is running)