Signal Handling Subsystem

When handling the XSIGRCV flag, I handled as I did in the previous lab. The only change I did was that I moved the callback to sleep, where resched returned, right after interrupts had been enabled. I made the mistake of putting the callback at the end of resched where interrupts were disabled. In the main, my create(resume()) calls a function above, where in that function, the callback is registered. To test that the callback was successful, I included a kprintf statement in my callback function in the main to signal when the process had entered it.

When handling the XSIGCHML flag, I created an array of pids for each process that holds the pids of its children and a counter that keeps track of how many children each process has. In create, I set the pid of the child in the parent array, where the array is initialized to -1 for each index. So, when I call childwait() in the parent once I register the parent callback, it checks immediately if there are any child processes in its array. If there isn't, I return, otherwise I find the first non -1 value in its children pid array. Once I find it, I do a while loop that waits until that pid in its array switches back to -1. This happens in kill; I make the pid of the child, in the parent's children array, to -1. Once the value switches to -1 when the child is killed, the parent returns the pid of the child that was terminated, and then calls the callback function. I verified my integration by including a kprintf statement in the callback function to visualize when the process has entered the callback function.

When handling the XSIGXTM flag, I check ed in resched() if the process had a callback, i.e. a nonzero value in prhascb2, and then I checked how long the process had been running. I did this by taking clktime – prptr->prstarttime (the value set to clktime when create is called) and checked if this was >= the value specified in the call back register function, tmarg that I stored in the proctable when the process registered its callback function. If the run time had exceeded, or was equal, I called the call back function specified by fptr2 in the process table. To verify my correctness of my implementation, I did a create(resume()) where it called my function. In this function, it registered the callback function and then slept for 3 seconds, where the timeout was sent in as 3 as well. I then added a kprintf statement in the callback to verify the process had entered such. I also tested the values of 2 and 1 in the sleep() call, and to my expectation, the callback function had not been called.

For the case of managing back to back invocations of the same signal, I would suggest a queue of sorts to hold future cases of the same invocation. The OS will hold the new process of the same signal until the current process returns to the handler or is killed, then it will switch in the first process from the queue.

Memory Garbage Collection

When dealing with garbage collection, Xinu tracks the overall memory allocated during the run time of the OS in memlist. To enhance this feature, I made every process keep track of its own memory allocated for itself. To do this, I attempted to define a new memblk struct within the procent table, but the compiler had an issue with this. To work around this, I copied the memblk struct into the process.h file and named it memblock with the same fields inside.

In getmem(), I added the same block of memory being allocated to the processes memblock struct. I also did the same procedure in freemem(), where I free the same requested block in the memblock struct within procent. This only handles when a block of memory was explicitly freed; to free the remaining memory, I cleared the memblock struct when the process's pid is sent to kill(). I looped through the memblock structure and sent each block to be freed. For structuring my code to add the block to my own linked list, I used the code provided by the default functionality for memlist. I also added the size of my struct on return to handle the new struct for the next pointer.

For testing my implementation, I called a function in my main that first printed the length of memlist first, allocated the follow: 100, 150, 200, 250, and 500, and then finished by printing the length of memlist once again. The total sent in was 1200, but with rounding, it totaled 1256. I then, in the main right after the call to the function that allocated the 1256 memory, I called sleep(5), and then printed the length of memlist. I saw that the 1256 amount had been freed, as well as a little bit more memory had been freed.

The following was the result of my latest test case for memory garbage collection:

```
Current memory before calls: 202248
Current memory after calls:  203504
Current memory after sleep: 201968
```

Between the first two kprintf statements, the difference is exactly 1256, which was the 1200 requested with rounding. Between the last two kprintf statements, the difference is 1536.