Kyung Jae Cha

Prof. Park

CS354

04/17/18

CS 354 - Spring 2018

Lab 6: Signal Handling Subsystem and Garbage Collection (340 pts)

Due: 04/17/2018 (Tue), 11:59PM

Problem 3. Signal handling subsystem

Implementation of XSIGRCV

XSIGRCV has not changed from cbreg(int (*fnp) (void)) of the lab5. Because it only serves the receiving signal and the third tmarg is ignored, it works okay as it is.

Implementation of XSIGCHL

Signal of SIGCHL makes signal handler to post the callback function inside of kill() function call. Parent process calling childwait will produce four different results. The normal case will return pid of the first child process that has ended. Second will be returning child process right away without blocking the parent process. For the third, if all the child processes ended childwait() returns the first process and subsequent childwait() call will return SYSERR. In order to achieve this, I tried to use the prnchld to keep the number of the child process for the parent process. Then the number of the children will be used inside of childwait to determine whether to return the pid or SYSERR.

Implementation of XSIGXTM

My implementation of XSIGXTM starts with adding additional field under process table called prwalltime. It is saved to tmarg times 1000 when signal callback function is registered with ssig of XSIGXTM. And inside of clkhandler(), it checks if the process signal and compares its lifetime to walltime. Whenever clktime(which is the total system time) is subtracted by the process's start time (starttime) and the value is bigger than or equal to 0, the callback happens.

Back-to-Back Invocation of the same signals

I would assume to use blocks to block signal handler, maybe have a queue structure like we made in Lab5. In this way, we could be able to handle one of the same signal at a time (FIFO – runs newly added signal after the first handler returns).

Problem 4. Memory garbage collection

Implementation on memory garbage collection

My implementation on memory garbage collection revolved around marking down the memory blocks that has been allocated per process. To achieve this, I've created another variable under procent in process.h in a type of struct memblk called prmemblk. This prmemblk is marked when getmem() function call allocates memory from free memory list. While the free list gets modified, this prmemblk also gets modified with the same information. For freeing the memory, I've modified kill() function so that it will check if prmemblk is null. If it is not, it will go traverse the prmemblk list and free all the memory associated with the process.