CIS722 Project 3

Due: 5:00pm on 12/11/2014 (Thursday)

Part 1: Signal Processing

Comment out unnecessary lines in between lines 529 and 540 in do_sigsend() in src/kernel/system/do_sigsend.c found in Course_Notes/signal_related.pdf in KSOL.

Note to comment out statements, use /*…. */, and do not use //……

In fact, "struct sigframe" defined in src/include/sys/sigcontext.h is referred to only from do_sigsend(). Therefore, in addition to deleting several lines in do_sigsend() and boot the OS, I deleted unnecessary lines in the structure and linked it to signal programs in Programs/Signal_Program.zip in KSOL. The programs run (printed the statements written by the printf statements right before the end of main() functions). However, the programs did not terminate or the control did not return to the shell. The OS also failed to shutdown. This is because there are many programs (shell, init, etc) that use signals. Those programs assume the original sigframe structure, but the OS used a modified sigframe structure.

In this (sub)project, I gave up to ask you to modify the sigframe structure and thus the project became too simple. Therefore, I added Part 2.

What to do:

1. Mount proj3.img. You will find two directories: proj3 and signal_proj. Copy signal_proj to your working directory and umount proj3.img (/dev/fd1)
2. In signal_proj, there are the top level Makefile and directory kernel. Directory kernel has two files, .depend and Makefile, and a directory system. Directory system has .depend, Makefile and do_sigsend.c. You work on do_sigsend.c. Comment out unnecessary lines in the file. Do not touch other files.
3. Go back to directory signal_project and issue "make fdboot" to create a boot image in /dev/fd0.
4. Makefile in signal_proj/kernel/system creates a lot of .o files in kernel/system and creates kernel/system.a by combining these .o files. Makefile in signal_proj/kernel creates "kernel" (the kernel image file) in the directory. Makefile in signal_proj creates "image" (boot image file) in the directory and installs it in /dev/fd0.
5. Shutdown the system and boot from fd0 (> boot fd0). If it boots, your implementation is working. In addition, compile and run several programs in

Signal_Programs. When compiling, do not forget to give –D_POSIX_SOURCE option.
6. When your implementation works, issue “make clean” in signal_proj and copy the directory back to /mnt (after mounting proj3.img). Also make a hardcopy of do_sigsend.c and highlight the lines you have removed. Turn in your hardcopy.

Part 2: Implementation of Semaphore System Calls

In this part of the project, you are to implement semaphore system calls. The semaphore sub-system supports the following five operations:

1. int init_sem(void)

    - Initialize the system data structures used in semaphore operations.
      This is to be called when errors in user processes have destroyed the semaphore system data structures (for example, when a process terminates without releasing semaphores).
    - Always return 0 (since no error is expected to occur). This is to maintain the compatibility with other semaphore operations (refer to "semlib.c").

2. int create_sem(int key, int initial_val)
    - The system should be able to create up to 5 semaphores
    - Create a semaphore with "key," initialize its semaphore value to "initial_val," and return the descriptor (a descriptor may be any int value (up to you) that uniquely identifies a semaphore).
    - "key" must be a positive integer and "initial_value" must be a non-negative integer.
    - If a semaphore with the same key exists, return its descriptor (in such a case, do not re-initialize the semaphore with "initial_val")
    - Return the descriptor if there is no error; otherwise return -1 and set global variable "errno" as follows:

      1 when "initial_val" is negative or "key" is not positive

      2 when no more semaphore is available (all 5 semaphores are used)

        - Hint: Review syscall.c on page 17 in the course notes
3. int p(int sem_desc)
    - Perform a "p" operation on semaphore represented by descriptor "sem_desc"

    - Return 0 if there is no error; -1 if "sem_desc" is either out of bounds or not in use
4. int v(int sem_desc)
    - Perform a "v" operation on semaphore represented by descriptor "sem_desc"
    - Return 0 if there is no error; -1 if "sem_desc" is either out of bounds or not in use
5. int delete_sem(int sem_desc)
    - Delete the semaphore represented by descriptor "sem_desc"
    - If there are blocked processes in the semaphore, release all these processes before deleting the semaphore
        - actually some of the blocked processes may no longer exist and trying to release such processes may be dangerous; however, in this project, you do not have to consider such complex situations
    - Return 0 if there is no error; -1 if "sem_desc" is either out of bounds or not in use

What to do

1. Copy /mnt/proj3 to your working directory and unmounts /dev/fd1. Directory proj3 has Makefile and three directories: servers, lib, and test
2. Directory servers has pm in which you find .depend, Makefile, and four .c/.h files you work on.
    - Files main.c and table.c are original files from /usr/src/servers/pm. In main.c, add a call to a function that initializes your semaphore data structure. In table.c add an entry to the samaphore calls. Both of the added functions must be implemented in semaphore.c (see below)
    - Files semaphore.c and semaphore.h are new files added for this project
        i. Implement your semaphore sub-system in these files.
        ii. Your semaphore implementation manages semaphore counters, queues, and keys, and blocks and unblocks processes.
        iii. Ownership or security features do not need to be implemented.
3. Use interface files semlib.h and semlib.c found in proj3/lib/
    - They define the semaphore system call entry functions. They also define the semaphore call number and message structures.
    - Create object file semlib.o by "cc -c semlib.c"
    - An application program using semaphores must link itself with semlib.o
4. Compile and execute semtest1.c, semtest2.c and semtest3.c in proj3/test
   Run initsem.c (which just calls init_sem()) when processes have destroyed the semaphore data structures.
    - Create the executable file of semtest.c by "cc -o semtest1 semtest1.c ../lib/semlib.o"
    - Read and understand each test program. Check your output based on your understanding of the code

    i. The output of semtest1 is found in semtest1_out.
    ii. Output of other test files cannot be captured because of buffering performed by the system

5. After completing your program, execute “make clean” in the proj3 directory and copy the directory back to /mnt (after mounting /dev/fd1)
6. Make a hard copy of semaphore.h and semaphore.c in proj3/servers/pm. *Also write explanation of your implementation. In particular, describe the data structure you used in your implementation. Figures will help.* Turn in the hard copy of your code and the description to me.
7. Change proj3.img to “YourLastName_YourFirstName.img” and submit it in file dropbox Project3 in KSOL. You do not have to zip the img file.

NOTE (VERY IMPORTANT)

The source code (version) of Minix3 installed in your virtual machine is slightly different from the code found in the textbook. Function get_work (18099 in the text) sets the process number (the index of the PCB array) of the sender in global variable “who_p”, instead of “who” found in the text. This value should be passed to setreply ((18116 in the text) as argument proc_nr. Another global variable “who_e” is also set in get_work. However, do not use “who_e” as argument proc_nr to setreply().