Gray Houston
CS 354
Dr. Park

Lab 5 Answers

Note: I am using 2 late days with this submission. Additionally, because of my commitment with the PMO Christmas Show, I was not able to complete part 2 of this lab.

1.)
The design approach that I used to implement the asynchronous message receive is the same as the standard implementation published in the lab notes. For the alarm signal handler, I implemented a time delayed queue (like the sleep queue) to keep track of all the processes who had registered alarms with the system. This alarm queue is updated by the clock handler, just like the sleep queue.

For the CPU usage signal handler, I altered the process table so that CPU usage was stored (in milliseconds). To test the effectiveness of this signal handler, I altered the behavior of the "ps" shell command so that it would also output the amount of CPU usage of the processes.

Here is an output of some simple test cases that are testing the effectiveness of the alarm and cpux signal handlers.

```
Welcome to Xinu!


xsh $ ps
Pid Name             State Prio Ppid Stack Base Stack Ptr  Stack Size CPU Use
--- ---------------- ----- ---- ---- ---------- ---------- ---------- -------
  0 prnull           ready    0    0 0x0EFDEFFC 0x0EFDEF40     8192       1
  1 rdsproc          wait   200    0 0x0EFDCFFC 0x0EFDCAAC    16384       1
  2 Main process     recv    20    0 0x0EFD8FFC 0x0EFD8F64    65536       7
  3 CPU Intense Pro  ready   20    2 0x0EFC8FFC 0x0EFC8EE0    65536    1871
  4 shell            recv    20    2 0x0EFB8FFC 0x0EFB8C6C     8192       1
  5 ps               curr    20    4 0x0EFB6FFC 0x0EFB6E28     8192       1
xsh $ Time remaining in alarm = 0       Signal Handler = 6 PID = 2
This is your alarm going off. Wake up, or you'll be late for Spiderman's awesome
party.

xsh $ psPID: 3, stop being a CPU hog!


Pid Name             State Prio Ppid Stack Base Stack Ptr  Stack Size CPU Use
--- ---------------- ----- ---- ---- ---------- ---------- ---------- -------
  0 prnull           ready    0    0 0x0EFDEFFC 0x0EFDEF40     8192       1
  1 rdsproc          wait   200    0 0x0EFDCFFC 0x0EFDCAAC    16384       1
  2 Main process     recv    20    0 0x0EFD8FFC 0x0EFD8F64    65536       7
  3 CPU Intense Pro  ready   20    2 0x0EFC8FFC 0x0EFC8EE0    65536    5299
```

```
   4 shell          recv    20    2 0x0EFB8FFC 0x0EFB8C6C     8192        1
   6 ps             curr    20    4 0x0EFB6FFC 0x0EFB6E28     8192        1
 xsh $ ^C
```

The use of the 2 calls to “ps” show how both the alarm and cpux signal handlers are working. The timed alarm is supposed to occur after 2 second. For the first call to “ps”, if you sum all of the milliseconds in CPU Usage column (the far right column), you get a total of 1882 milliseconds having occurred in the system. Therefore, it is logical that the alarm should be printed almost immediately after the call to “ps” (which it is). Therefore, the signal handler works.

To test the cpux signal handler, I have registered an alarm that should occur after a process uses 5000 milliseconds of time. If you look at the output of the 2nd call to “ps” (occurring immediately after the cpux callback function executes), you will see that the “CPU Intensive Pro” process has used just over 5000 milliseconds. Therefore, it can be logically inferred that the callback function was executed at the correct time.

Files Changed
----
process.h
- added callback function pointer and signal variable to the process table
- added entry so that cpu usage is stored in the process table
- added entry to store the amount of CPU usage which activates the MYSIGXCPU handler

mysignal.h
- created this file to hold constant signal values

xinu.h
- added include for mysignal.h and alarmQueue.h header files

prototypes.h
- added function prototypes for the registercb and registercbsig functions and all of the alarm queue functions

alarmQueue.h
- made separate queue for alarm signal handlers; based on the normal queue operations

clock.h
- added extern reference to alarmq (the variable for the alarm queue)

resched.c
- added functionality so that the proper callback function is executed for the correct signal if it exists immediately after process is context switched back on to the processor

registercb.c
- created the system call like in TA notes; it's super simple
- if it's the alarm signal, the process is placed into the alarm queue
- if it's the CPU usage signal, the appropiate table fields are initialized

registercbsig.c
- created this system call that implements the signal handling functionality as described in the lab question 1 specification

create.c
- initialize added process table entries for the alarm queue in the create function
- initialize CPU time to 0 in process table
- initialize usageLimit variable for each process table entry

initialize.c
- added print statement to ready loop of null process
- all entries in the process table, including null process, are initialized to have a null callback function, 0 signal handler value, 0 CPU time used, and 0 value for the usageLimit variable

alarmQueue.c
- implemented enqueue, dequeue, getfirst, getlast, getitem, and newqueue functions for the alarm queue

ainsertd.c
- added functionality to time delay insert items into the alarm queue

clkinit.c
- initialized the alarm queue right after the sleep queue

clkhandler.c
- update the alarm queue every millisecond when the clock handler is invoked
- updates the amount of time each process has spent on the CPU
- added execution of CPU usage signal handler

main.c
- testing my implementation

kprintf.c
- disabled interupts during function so that debugging output wouldn't be scrambled

xsh_ps.c
- added CPU usage information to the ps command