# CS 503 Lab 4 Answers

*Prashant Ravi — ravi18@purdue.edu*

April 3, 2016

## Files modified/added

### 1 Source Files

lab4q2.c : Added the three test cases as requested by problem specification.
resched.c: added handleCallback function that runs after context switch in each rescheduling policy.
send.c : added kprintf instruction so send command can be displayed on console with timestamp.
registercbsig.c : added support for the three signal types.
registercbsig.2 : added the man page
registercb.c : callback registration procedure for the regular callback on message arrival problem from part 1.
main.c: runs the initial round of test cases.
lab4q3.c : contains all the test cases for question 3 of the lab.
lab4q2.c: contains all the test cases for the question 2 of the lab.
initialize.c and create.c: account for initilization of new fields in procent structure.
clkinit.c : add new "addq" queue variable and initialize it.
clkhandler.c : account for xcpu signal countdown as well as waking up invocation of alarmq processes.
alarmwakeup.c ; waking up a process in alarmq. Involves checking if current process or not and then running the respective procedure.
alarmqueue.c ; everything for new alarm queue in one place.
alarm.c : system call similar to sleepms but for alarms. Will wake up process forcefully on end of alarm timer to execute its callback function.

### 2 Header Files

xinu.h : added mysignal.h and alarmqueue.h
prototypes.h: add the extern methods written in c files.
process.h : add additional fields in procent structure to support part 1 and part 2 of this lab.
mysignal.h: add the constants for signal types.
clock.h: introduce new alarmq

alarmqueue.h : trying to create new alarm queue make replica of queue.h and renamed the variables in this file.

## Note

TO RUN LAB 4: please make sure to change lab variable in main to equal to 2 so we can use the Fair scheduling policy provided by me, that allows us to use cputimemsec, which tells us how much time a process has held on to cpu. Useful for testing purposes.lab2flag =0;

## Question 4.2

For this part my solution is in registercb.c. I defined function called handleCallback in resched.c that executes after the context switch method cxstw. So basically as soon as process comes back from context switch first thing it runs is the handlCallback function. If there is valid callback function and the message buffer flag is set to true then we execute the callback function. Note, I also reset the flag after the call back function is executed as required in the TA notes. As for registering a call back it was as simple as making new field in procent that holds the callback function pointer and setting it upon registration, which we do my masking interrupts, registering and then unmaking interrupts, to prevent context switch during this registration process.

## Question 4.3

For the xcpu signal I basically implemented a countdown timer in the clkhandler.c. The xcputime is set in the registercbsig.c according to the optional argument which represents the additional cpu time that would be provided to this process before the xcpucallback function will be invoked and also define this associated callback function. In the clkhandler, as long as current process is running the xcputime alloted decrements. If it reaches zero, then we must call the callback function associated and some housekeeping to reset the callback function to prevent it from running again accidentally.

The alarm signal part is probably the most intricate to explain. First of all I make a new queue similar to one provided in xinu so far. I replicate everything provided and put it in alarmqueue.c. So now this queue is responsible for all the processes that called alarmms(), the system call that sets of the timer for the alarm signal for that process. In lay man's term once the timer goes off the callback function associated with the alarm in the process that's defined in the process entry table should be executed. So I follow the steps mentioned in specs and add in the decrementing, alarm induced process wakeup procedure in the clkhandler.c similar to the sleepq procedure in the same function. In the alarmwakeup.c procedure in similar to wakeup procedure we make ready sleeping processes who's alarm timer went off just so they execute the callback

function. I don't put such woken up processes to sleep later on. In the wakeup procedure if we are current process then it must be our lucky day, as we can directly just call the callback function, if not we just set the alarmTimeout variable to true so that when the process does got context switched in it will first go to the handleCallbacks routine where it will find that the alarm callback invocation requirements are fulfilled and hence will go onto execute the alarm callback. Finally, I would like to note that the insertion into the alarmqueue is done in the registercbsig.c or the alarmms system call where if a time and callback function is specified then we initialize both fields and insert this process into alarmq but if only the callback function is specified then I save function in the process entry table anticipating that the user will invoke a alarmms system call to actually define the waiting period for the timer to go off, then we insert the process into the alarmq in the alarmms routine. Mentioning the alarmms mandates I also make it clear that I support overwriting of alarms and for each process only one signal of each type could be active. That is for each process at most 3 unqiue signal types can be active, So overwriting is possible for all three signals.

### 1 Data structures used

To support MYSIGALARM I had to make a new queue and used the process entry data structure to support this functionality.

## Performance

The MYSIGRECV ran correctly. The MYSIGXCPU also ran correctly. I've printed out the amount of time that the process has spent on CPU. The amount of time at the registration time substracted by the amount of COU time during callback invocation is the amount of time that we set as the time argument while registering this signal. So, it was very accurate. However, the MYSIGALARM handler was invoked with some delay , deviation was a little from the configured timer, my understanding is that since we have the chance of having to wait until process gets context switched back in to run the alarm handler, it is possible to incur a delay.