CS4375-13948 Fall 2023
Ashley Magallanes
avmagallanes@miners.utep.edu

Homework Report
October 14, 2023

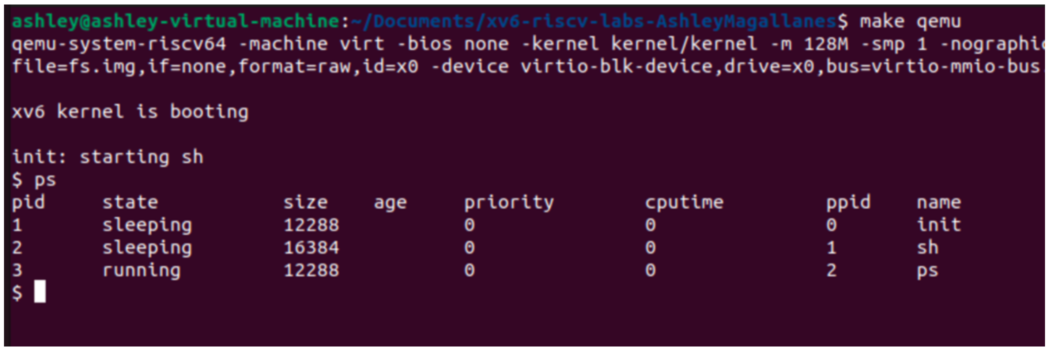
HW 3: Priority-based Scheduler for xv6

Task 1. Modify the provided ps command to print the priority of each process.

To retrieve and print the priority of each process I had to modify not only the ps.c command file but also the procinfo() function since it was being called within the getprocs() system call. The function procinfo() in proc.c copies the information belonging to each process provided the process address and then fills an array with the corresponding information. Therefore in order to allow the priority of the process to also be accessed I had to include this line: procinfo.priority = p->priority. However, since procinfo is a pstat structure the field 'priority' had to also be defined in the pstat structure in pstat.h. Therefore, the field priority exists in the proc structure and in the pstat structure as well.

Now that the priority can be accessed just like the state, size, process id and name. I followed the same format and added "priority" to the title printing line. Then inside the for loop that goes through all processes there is a printing statement that handles both retrieving and printing the process information, which includes the state, size, name, process id, and now also the priority. Since it can now be accessed through the pstat structure, the ps program already had a set pstat structure to access the rest of the process information so I just also added the priority field as: uproc[i].priority. Uproc is the pstat structure used in ps.c. Since the ps output sample file in the instructions also had the cputime I did the same in order to also print this field when the ps command was ran.

*These are the results of running the ps program from my qemu: I tested that the priority showed by just using the ps command.
(I had already done task two as well when I did this screenshot, but before it just didn't include the age field in the "table")



```
ashley@ashley-virtual-machine:~/Documents/xv6-riscv-labs-AshleyMagallanes$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 1 -nographi
file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus

xv6 kernel is booting

init: starting sh
$ ps
pid     state           size    age     priority        cputime         ppid    name
1       sleeping        12288           0               0               0       init
2       sleeping        16384           0               0               1       sh
3       running         12288           0               0               2       ps
$ 
```

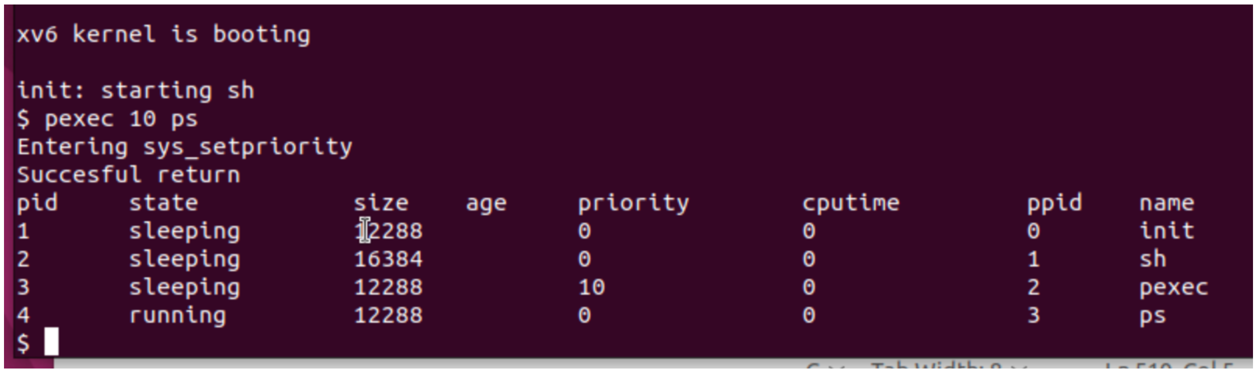
Task 2. Add a readytime field to struct proc, initialize it correctly, and modify ps to print a process's age.

To have the process's age I had to add a readytime field, which as mentioned was added in the proc structure. Following the same idea from task number one of being able to have access to this field for each specific process, I also added the readytime field on pstat and used its "copy" on procinfo as well. This was just like the procedure I had to do for the priority filed. Therefore, I included this line: procinfo.readytime = p->readytime. Again procinfo being a pstat structure and p being a pointer to a proc structure.

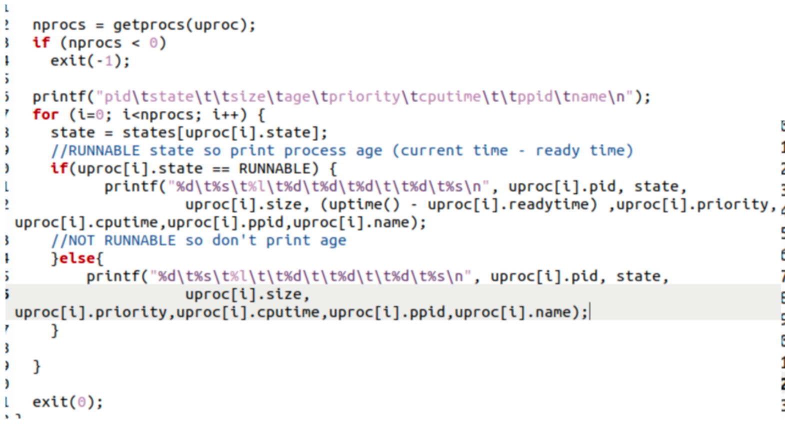
Since, the readytime field could now be accessed I need to make sure it worked properly and incremented, therefore it wasn't a set field just like the priority, name, process id. This field needs to be initialized every time the process state becomes RUNNABLE. Therefore, I went through all of the commands in proc.c and whenever the state changed to runnable, I would initialize the readytime filed to the current time. The current time can be accessed using the uptime() method but because this method is in the user and not in the kernel program, I had to do some modifications in order to be able to use it in the kernel. The uptime() has a corresponding sys_uptime() but this needs to be defined in defs.h, which is what I did, and then I could use the kernel's uptime system call in order to get the current time.

Now, I just had to check when the state was changed to RUNNABLE, which relied on p->state being set to RUNNABLE. Then the following line I would initialize the readytime to the current time by using: p->readytime = sys_uptime(). After doing the same procedure for all the methods that required a state to be set to runnable, I now modified my previous ps.c file to also include the age. The age of a process is the current time minus the readytime. So, I just added another "entry" on the title printing statement called age followed by a tab to create that "table" format when it prints. Then since not all processes are in runnable state I added an if statement to check this before hand, so: if(uproc[i].state == RUNNABLE). This would print the line including the age field, which is just: uptme() – uproc[i].readytime. If the state was not runnable then it would just not print anything as the age.

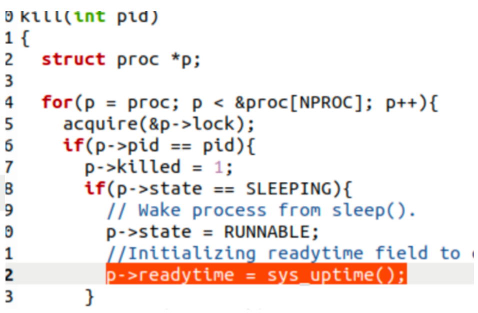
The results of my testing that age was also shown include the screenshot for task one along with these:


```
xv6 kernel is booting

init: starting sh
$ pexec 10 ps
Entering sys_setpriority
Succesful return
pid     state           size    age     priority        cputime         ppid    name
1       sleeping        12288           0               0               0       init
2       sleeping        16384           0               0               1       sh
3       sleeping        12288           10              0               2       pexec
4       running         12288           0               0               3       ps
$ 
```

These are just some portions of code and how I modified them for this task:



```
  nprocs = getprocs(uproc);
  if (nprocs < 0)
    exit(-1);

  printf("pid\tstate\t\tsize\tage\tpriority\tcputime\t\tppid\tname\n");
  for (i=0; i<nprocs; i++) {
    state = states[uproc[i].state];
    //RUNNABLE state so print process age (current time - ready time)
    if(uproc[i].state == RUNNABLE) {
          printf("%d\t%s\t%l\t%d\t%d\t%d\t\t%d\t%s\n", uproc[i].pid, state,
                       uproc[i].size, (uptime() - uproc[i].readytime) ,uproc[i].priority,
 uproc[i].cputime,uproc[i].ppid,uproc[i].name);
    //NOT RUNNABLE so don't print age
    }else{
        printf("%d\t%s\t%l\t\t%d\t\t%d\t\t%d\t%s\n", uproc[i].pid, state,
                       uproc[i].size,
 uproc[i].priority,uproc[i].cputime,uproc[i].ppid,uproc[i].name);
    }

  }

  exit(0);
```

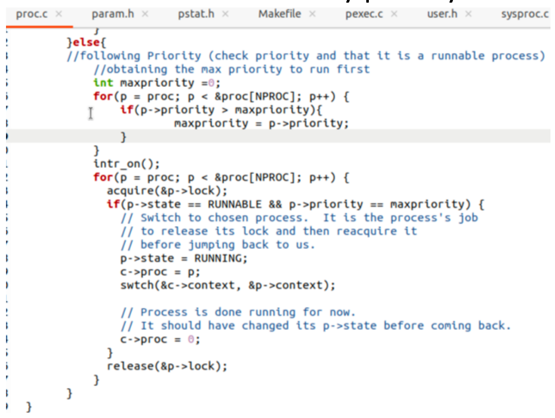


```
kill(int pid)
{
  struct proc *p;

  for(p = proc; p < &proc[NPROC]; p++){
    acquire(&p->lock);
    if(p->pid == pid){
      p->killed = 1;
      if(p->state == SLEEPING){
        // Wake process from sleep().
        p->state = RUNNABLE;
        //Initializing readytime field to
        p->readytime = sys_uptime();
      }
```

Task 3. Implement a priority-based scheduler.

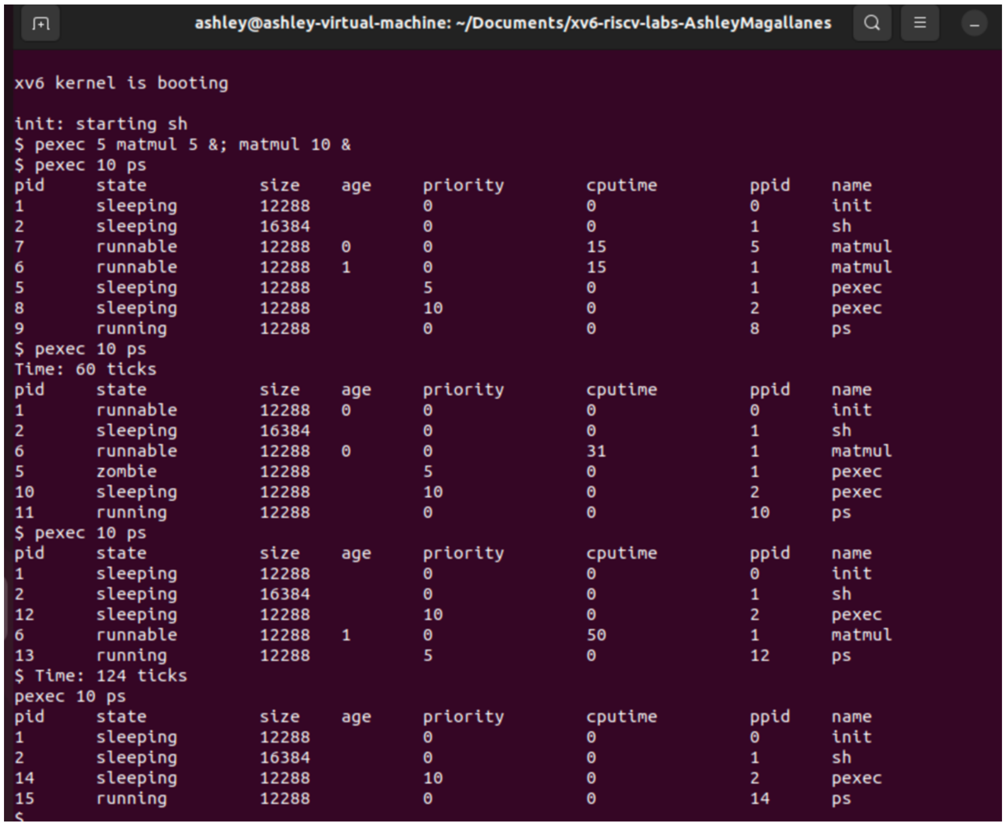
To implement the priority based scheduler I first added some constants on the param.h, these constants would allow me to chose between the scheduling polices. The constants I added were: #define RR 0 which is the constant that would run the round robin (already builtin/provided) scheduler, #define PRIORITY 1 which is the constant that would run the priority scheduler that I would include, lastly #define SCHEDULEPRIORITY PRIORITY which just sets the scheduler that would be using. Then in proc.c I added a schedulepolicy field, which would take care of which schedule would be used. Since, each schedule constant is just an int either 0 or 1 then the data type of this schedulepolicy field is also int. This will allow the program to distinguish which schedule to implement in the schedule().

Then I managed to implement the priority scheduler by just adding into the already provided scheduler() function. I used an if statement to check which schedule policy was being used, if the schedulepolicy was equal to zero then I left the round robin within it, else I implemented the priority. For the priority you don't only need to check that the state of the process is runnable but also check the priority of the process. The higher the priority the process should be run first. Therefore, I added a maxpriority field, and with the help of a for loop that traversed all the processes I could check which process had the highest priority and therefore when making the schedule it would consider the the process with the maxpriority first. This is the code for my priority scheduler:



```
        }else{
        //following Priority (check priority and that it is a runnable process)
            //obtaining the max priority to run first
            int maxpriority =0;
            for(p = proc; p < &proc[NPROC]; p++) {
                if(p->priority > maxpriority){
                        maxpriority = p->priority;
                }
            }
            intr_on();
            for(p = proc; p < &proc[NPROC]; p++) {
              acquire(&p->lock);
              if(p->state == RUNNABLE && p->priority == maxpriority) {
                // Switch to chosen process.  It is the process's job
                // to release its lock and then reacquire it
                // before jumping back to us.
                p->state = RUNNING;
                c->proc = p;
                swtch(&c->context, &p->context);

                // Process is done running for now.
                // It should have changed its p->state before coming back.
                c->proc = 0;
              }
              release(&p->lock);
            }
        }
  }
```

These are some results form my priority scheduler: (in the instructions it said that I could use the provided pexec program for my testing process)



ashley@ashley-virtual-machine: ~/Documents/xv6-riscv-labs-AshleyMagallanes

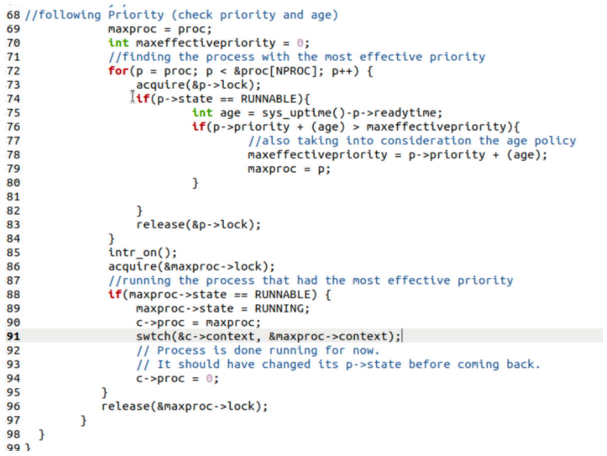
```
xv6 kernel is booting

init: starting sh
$ pexec 5 matmul 5 &; matmul 10 &
$ pexec 10 ps
pid     state           size    age     priority        cputime         ppid    name
1       sleeping        12288           0               0               0       init
2       sleeping        16384           0               0               1       sh
7       runnable        12288   0       0               15              5       matmul
6       runnable        12288   1       0               15              1       matmul
5       sleeping        12288           5               0               1       pexec
8       sleeping        12288           10              0               2       pexec
9       running         12288           0               0               8       ps
$ pexec 10 ps
Time: 60 ticks
pid     state           size    age     priority        cputime         ppid    name
1       runnable        12288   0       0               0               0       init
2       sleeping        16384           0               0               1       sh
6       runnable        12288   0       0               31              1       matmul
5       zombie          12288           5               0               1       pexec
10      sleeping        12288           10              0               2       pexec
11      running         12288           0               0               10      ps
$ pexec 10 ps
pid     state           size    age     priority        cputime         ppid    name
1       sleeping        12288           0               0               0       init
2       sleeping        16384           0               0               1       sh
12      sleeping        12288           10              0               2       pexec
6       runnable        12288   1       0               50              1       matmul
13      running         12288           5               0               12      ps
$ Time: 124 ticks
pexec 10 ps
pid     state           size    age     priority        cputime         ppid    name
1       sleeping        12288           0               0               0       init
2       sleeping        16384           0               0               1       sh
14      sleeping        12288           10              0               2       pexec
15      running         12288           0               0               14      ps
$
```

This task showed me how processes priority is important since round robin schedulers don't consider that piece of information. I also now better understand how the scheduler process works and how the state is important because to run the process it needs to be runnable which means that it is ready to be ran. I had a difficult time understanding the infinity loop that surrounds the scheduler process because I needed to find the maximum priority. However, Dr.Moore explained this to me and I was able of completing the task. Another thing that I struggled with was using pexec.c command to check the priority scheduling but after further debugging the AI helped me figure this out.

Task 4. Add aging to your priority based scheduler.

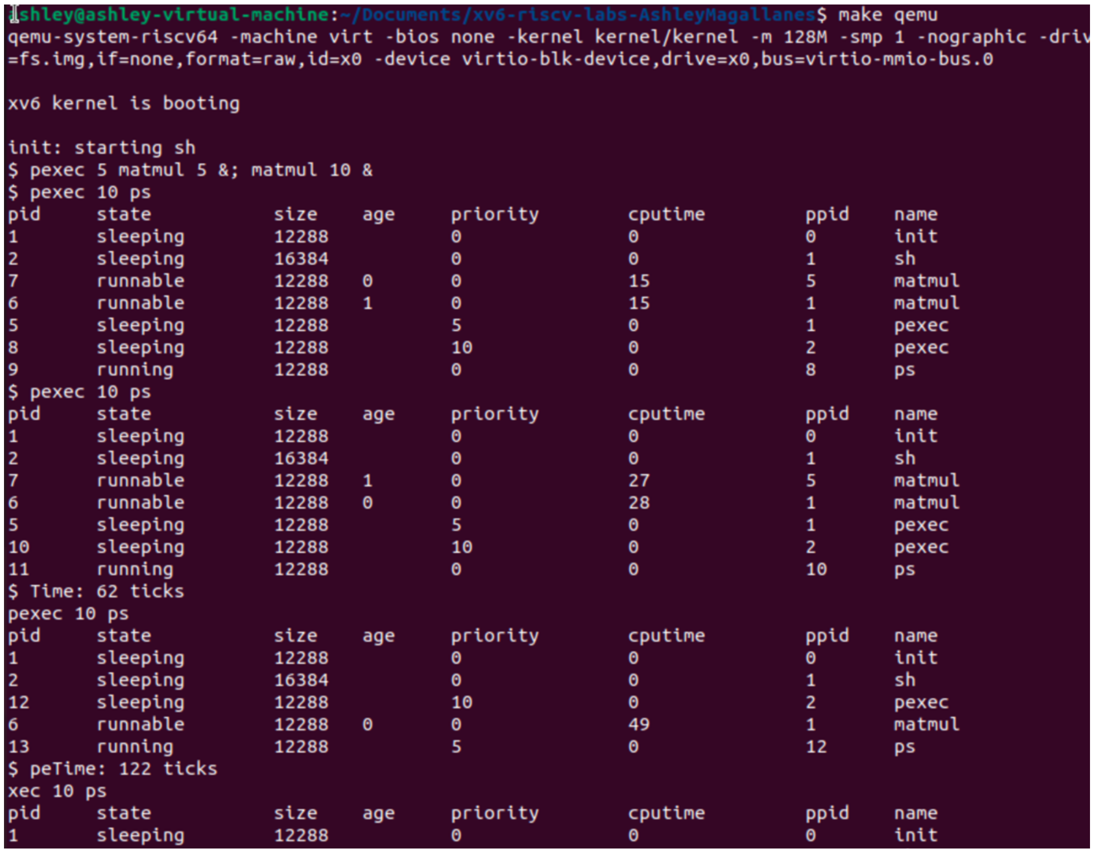
To add aging to the priority based scheduler that I had done in task number 3 I not only had to consider the priority but also the age, which as mentioned earlier is calculated my subtracting the current time and the readytime. The addition of both the maximum cpu and its corresponding age would provide the maximum effective priority which it belongs the process that has to be run first. Therefore instead of just finding the maximum priority I found the maximum effective priority and when I would traverse the array of process I would make sure that the process had the maximum effective priority and run that one. Since this task was a modification of the last one there isn't that much to explain other than what I changed.

These is my priority based scheduler following both the priority and the age of the processes which is the effective priority:



```
//following Priority (check priority and age)
            maxproc = proc;
            int maxeffectivepriority = 0;
            //finding the process with the most effective priority
            for(p = proc; p < &proc[NPROC]; p++) {
                acquire(&p->lock);
                if(p->state == RUNNABLE){
                        int age = sys_uptime()-p->readytime;
                        if(p->priority + (age) > maxeffectivepriority){
                                //also taking into consideration the age policy
                                maxeffectivepriority = p->priority + (age);
                                maxproc = p;
                        }

                }
                release(&p->lock);
            }
            intr_on();
            acquire(&maxproc->lock);
            //running the process that had the most effective priority
            if(maxproc->state == RUNNABLE) {
                maxproc->state = RUNNING;
                c->proc = maxproc;
                swtch(&c->context, &maxproc->context);
                // Process is done running for now.
                // It should have changed its p->state before coming back.
                c->proc = 0;
           }
           release(&maxproc->lock);
        }
  }
}
```

This are some results from running my most effective priority scheduler for task 4:



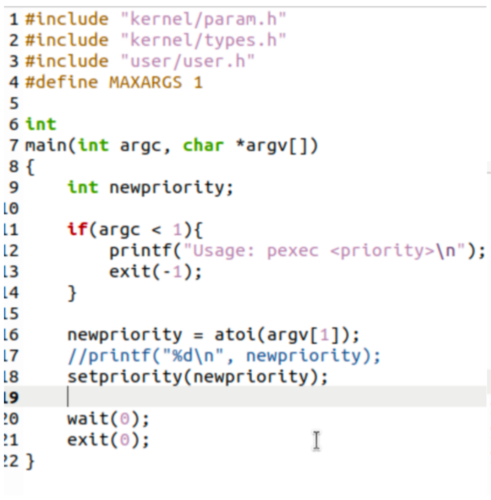
```
ashley@ashley-virtual-machine:~/Documents/xv6-riscv-labs-AshleyMagallanes$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 1 -nographic -driv
=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

init: starting sh
$ pexec 5 matmul 5 &; matmul 10 &
$ pexec 10 ps
pid     state           size    age     priority        cputime         ppid    name
1       sleeping        12288           0               0               0       init
2       sleeping        16384           0               0               1       sh
7       runnable        12288   0       0               15              5       matmul
6       runnable        12288   1       0               15              1       matmul
5       sleeping        12288           5               0               1       pexec
8       sleeping        12288           10              0               2       pexec
9       running         12288           0               0               8       ps
$ pexec 10 ps
pid     state           size    age     priority        cputime         ppid    name
1       sleeping        12288           0               0               0       init
2       sleeping        16384           0               0               1       sh
7       runnable        12288   1       0               27              5       matmul
6       runnable        12288   0       0               28              1       matmul
5       sleeping        12288           5               0               1       pexec
10      sleeping        12288           10              0               2       pexec
11      running         12288           0               0               10      ps
$ Time: 62 ticks
pexec 10 ps
pid     state           size    age     priority        cputime         ppid    name
1       sleeping        12288           0               0               0       init
2       sleeping        16384           0               0               1       sh
12      sleeping        12288           10              0               2       pexec
6       runnable        12288   0       0               49              1       matmul
13      running         12288           5               0               12      ps
$ peTime: 122 ticks
xec 10 ps
pid     state           size    age     priority        cputime         ppid    name
1       sleeping        12288           0               0               0       init
```

This task was a little more complicated than the rest because I got multiple infinity loops throughout the checking process. After further investigation I was able of showing some but not all results, therefore I had to do minor modifications in order to avoid this infinity loop. This task showed me how aging works, but it also demonstrated how a correct scheduling process should work.

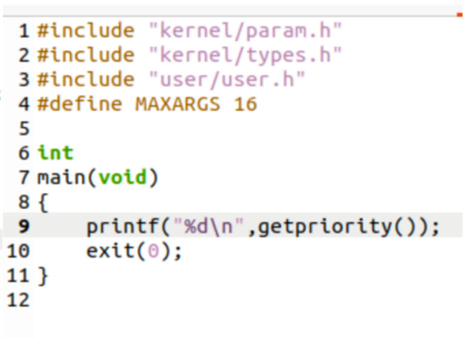
Extra Credit Task (10 points).

I added the getpriority and setpriority commands to qemu by making each one a .c file and then adding that to the makefile. I used the methods that I already had built in Task number one in order to complete this. I looked at the way that pexec was built as inspiration for the setpriority command. Lastly, these new commands will allow the response time of interactive jobs to be decreased!

The following images showcase some portions of the command code:

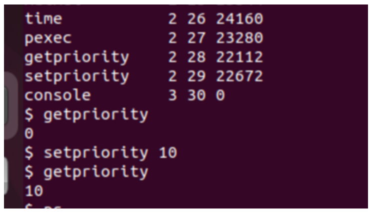


```c
#include "kernel/param.h"
#include "kernel/types.h"
#include "user/user.h"
#define MAXARGS 1

int
main(int argc, char *argv[])
{
    int newpriority;

    if(argc < 1){
        printf("Usage: pexec <priority>\n");
        exit(-1);
    }

    newpriority = atoi(argv[1]);
    //printf("%d\n", newpriority);
    setpriority(newpriority);

    wait(0);
    exit(0);
}
```



```c
#include "kernel/param.h"
#include "kernel/types.h"
#include "user/user.h"
#define MAXARGS 16

int
main(void)
{
    printf("%d\n",getpriority());
    exit(0);
}
```

This is some of the testing for these new commands:



```
time           2 26 24160
pexec          2 27 23280
getpriority    2 28 22112
setpriority    2 29 22672
console        3 30 0
$ getpriority
0
$ setpriority 10
$ getpriority
10
```