

ECE 478: Operating Systems

**Project 2 : <xv6 Scheduler – O(1) Scheduler>**

By

<Vlad Slyusar, Ben Dale>

Professor: <Dr. Shengquan Wang>



February 14<sup>th</sup>, 2015  
Winter 2015

Honor Code:

I have neither given nor received unauthorized assistance on this graded report.

Vladyslav Slyusar, Benjamin Dale

Department of Electrical and Computer Engineering  
College of Engineering and Computer Science  
The University of Michigan-Dearborn

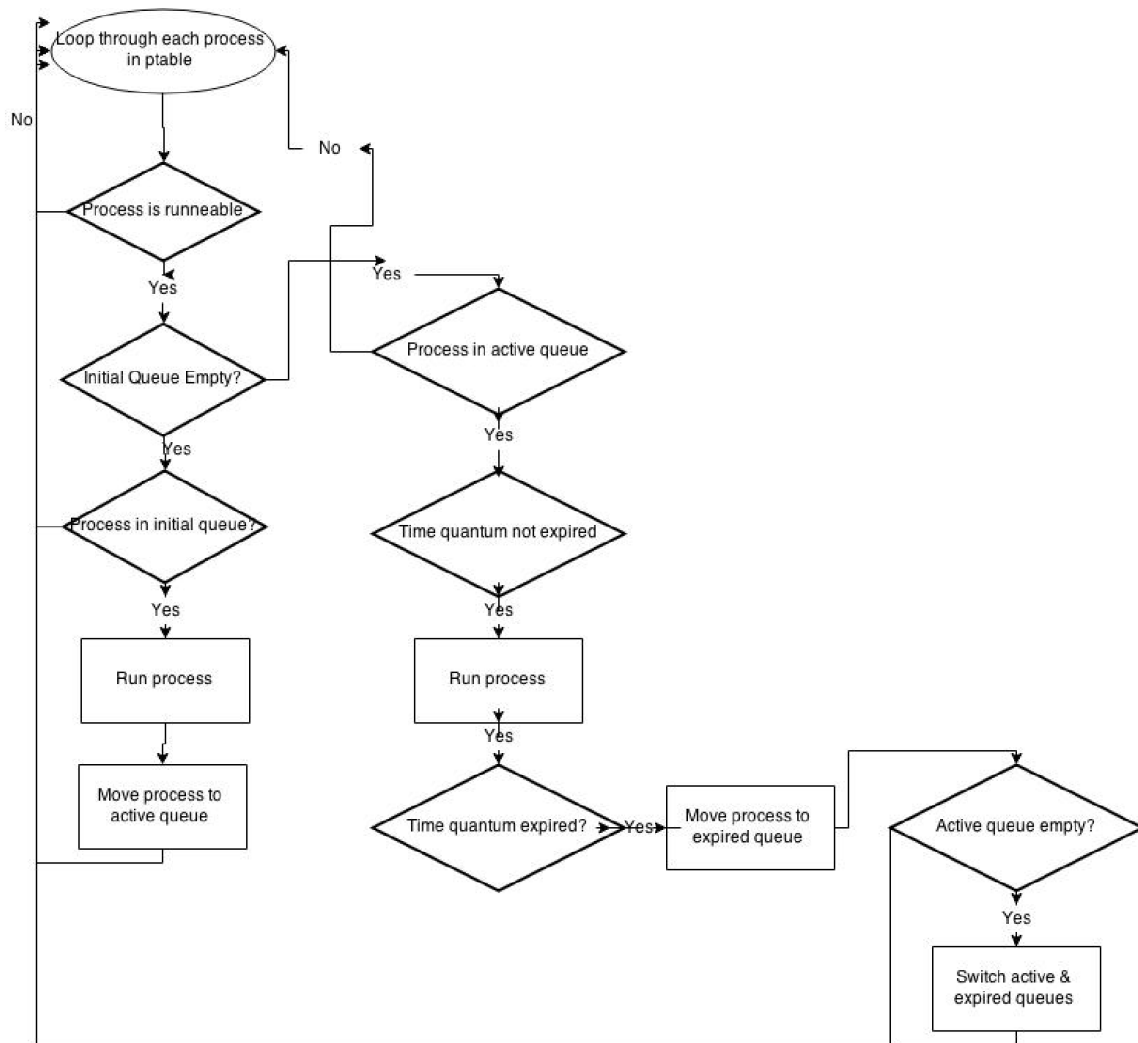
## Objective

The main objective of this project is to rewrite the scheduling algorithm used by the xv6 system kernel to make it an  $O(1)$  scheduler. The new scheduler must be implemented using multiple queues of different priority and different time quantum. When a new process is created, it must be placed into a high priority first-time queue for a single time quantum of 100ms. Once any process from the first-time queue has exhausted its time quantum, it must be moved to an active process queue with low priority and a time quantum of 400ms. Processes within individual queues are still selected using the round-robin algorithm originally implemented by xv6. When any process from the active queue has run for 400ms, it must be moved into an expired process queue which does not run at all until the active queue is completely empty. At that time, the expired queue must become the new active queue by switching the pointer variables for the active and expired queues.

## Implementation

The new scheduler implementation takes place almost completely within *proc.c*. Instead of maintaining an array for each of the queues, a simpler approach was used. Since *ptable* maintains an array of all processes and the *scheduler()* function loops through each process, queue assignments can be made as attributes of each process. This is done by adding a variable into the *proc* struct within *proc.h*. A variable such as *schedQ* will allow each process to be part of a different queue based on the value of *p->schedQ*. Since a total of 3 queues are required for the  $O(1)$  scheduler, *p->schedQ* for each new process was set to 0, or either 1 or 2 depending on which queue is currently the active queue. A similar approach was used to increase the time quantum for the active queue to 400ms instead of the standard 100ms round-robin implemented within xv6. Instead of modifying time interrupts for the active queue, a simple counter was used to make the process run 4 times ( $4 \times 100\text{ms}$ ) before moving it to the expired queue. A variable to keep track of the quantum was added to the *proc* struct which can be accessed as *p->quantum*. This variable gets decremented for the process every time it runs and reset back whenever

it is moved to the expired queue; therefore, when a queue switch takes place, all the processes are ready to go for 4 time quanta of 100ms. The diagram below illustrates a graphical representation of the algorithm used to implement the new scheduler for the xv6 kernel.



Scheduler Workflow Chart

When a new process is created, it gets assigned to the initial queue (q0) within the allocproc() function inside *proc.c*. Since process queues are not implemented as actual data structures, a global variable must be maintained for each queue to keep track of its

size. Furthermore, another global variable is used to maintain the pointer to which queue is currently the active queue. The value of the active queue variable is used as a condition within all parts of code that deal with processes from the active queue. This way, the members of each queue do not have to be individually changed to a different queue, simply the active queue designator points to the other queue.

For creating a custom process ps.c, a similar process was followed as the previous assignment with the exception of tracing a different system call(fstat). The file was added to the makefile the same way as cs.c in the past by simply inserting \_ps. The code for the psinfo process is provided below along with a screenshot of the output.

## Code Changes - Scheduler

### *Defs.h*

```
//Scheduler variables
extern int q0;
extern int q1;
extern int q2;
extern int activeQ;
```

### **Proc.h**

-----Modified proc structure-----

```
struct proc {
    uint sz;                // Size of process memory (bytes)
    pde_t* pgdir;          // Page table
    char *kstack;          // Bottom of kernel stack for this process
    enum procstate state;  // Process state
    int pid;               // Process ID
    struct proc *parent;   // Parent process
    struct trapframe *tf;  // Trap frame for current syscall
    struct context *context; // switch() here to run process
    void *chan;            // If non-zero, sleeping on chan
    int killed;            // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;     // Current directory
    char name[16];         // Process name (debugging)
    int schedQ;            // Process belongs to this queue(q0:0, q1:1, q2:2)
    int quantum;           // Remaining time quantum if process in active queue
};
```

Department of Electrical and Computer Engineering  
College of Engineering and Computer Science  
The University of Michigan-Dearborn

---

## Proc.c

-----Added at top of file-----

```
int q0 = 0; //number of items in new process queue
int q1 = 0; //number of items in active queue (initially)
int q2 = 0; //number of items in expired queue (initially)
int activeQ = 1; //sets active queue to either q1 or q2
```

---

-----Modified allocproc() function-----

```
static struct proc*
allocproc(void)
{
    struct proc *p;
    char *sp;
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if(p->state == UNUSED)
            goto found;
    release(&ptable.lock);
    return 0;

found:
    p->state = EMBRYO;
    p->pid = nextpid++;

    p->schedQ = 0; //place into queue 0, new process queue with highest priority
    p->quantum = 3; //time quantum 4*100ms
    q0+=1;
    cprintf("New Process, queues: %d, %d, %d, schedQ: %d, pid: %d schedQ: %d\n", q0, q1, q2, p->schedQ, p->pid, p->schedQ);

    release(&ptable.lock);

    // Allocate kernel stack.
    if((p->kstack = kalloc()) == 0){
        p->state = UNUSED;
        return 0;
    }
    sp = p->kstack + KSTACKSIZE;

    // Leave room for trap frame.
    sp -= sizeof *p->tf;
    p->tf = (struct trapframe*)sp;

    // Set up new context to start executing at forkret,
    // which returns to trapret.
    sp -= 4;
    *(uint*)sp = (uint)trapret;

    sp -= sizeof *p->context;
```

```

p->context = (struct context*)sp;
memset(p->context, 0, sizeof *p->context);
p->context->eip = (uint)forkret;
return p;
}

```

---

-----Modified exit() function-----

```

void
exit(void)
{
    struct proc *p;
    int fd;

    if(proc == initproc)
        panic("init exiting");

    // Close all open files.
    for(fd = 0; fd < NOFILE; fd++){
        if(proc->ofile[fd]){
            fileclose(proc->ofile[fd]);
            proc->ofile[fd] = 0;
        }
    }

    begin_op();
    iput(proc->cwd);
    end_op();
    proc->cwd = 0;

    acquire(&ptable.lock);

    // Parent might be sleeping in wait().
    wakeup1(proc->parent);

    // Pass abandoned children to init.
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->parent == proc){
            p->parent = initproc;
            if(p->state == ZOMBIE)
                wakeup1(initproc);
        }
    }

    // Jump into the scheduler, never to return.
    proc->state = ZOMBIE;

    if(proc->schedQ==0)
        q0--=1;
    else if(proc->schedQ==1)
        q1--=1;
    else if(proc->schedQ==2)
        q2--=1;

```

```

cprintf("Exit pid: %d, queueSizes: %d,%d,%d, schedQ: %d, activeQ: %d\n", proc-
>pid,q0,q1,q2,proc->schedQ,activeQ);

```

```

    sched();
    panic("zombie exit");
}

```

-----

-----Modified scheduler() function-----

```

void
scheduler(void)
{
    struct proc *p;
    int counter = 0; // Number of times process is skipped within active queue
    for(;;){
        // Enable interrupts on this processor.
        sti();

        // Loop over process table looking for process to run.
        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->state != RUNNABLE){ //check if process is runnable
                continue;
            }

            //Counter to prevent infinite loop, reset to 0 when any process is run
            counter+=1;

            //Switch queues when all processes in current active queue are
            doormant(waiting/sleeping)
            if((counter>=q1+1)&&(activeQ==1)){
                activeQ=2;
            }
            else if((counter>=q2+1)&&(activeQ==2)){
                activeQ=1;
            }

            //Run processes in initial queue (q0)
            else if(q0>0){
                if(p->schedQ == 0){
                    cprintf(" |queueSizes: %d,%d,%d, schedQ: %d, pid: %d, quantum: %d, active:
0|\n",q0,q1,q2,p->schedQ,p->pid,p->quantum);
                    goto runProcess;
                }
                else
                    continue;
            }

            //Run process in q1 (when q1 is the active queue)
            else if((activeQ==1)&&(q1>0)){
                if(p->schedQ != 1)
                    continue;
                else{

```

```

        cprintf(" |queueSizes: %d,%d,%d, schedQ: %d, pid: %d, quantum: %d,
active: %d|\n",q0,q1,q2,p->schedQ,p->pid,p->quantum, activeQ);
        goto runProcess;
    }
}

```

```

//Run process in q2 (when q2 is the active queue)
else if((activeQ==2)&&(q2>0)){
    if(p->schedQ != 2)
        continue;
    else{
        cprintf(" |queueSizes: %d,%d,%d, schedQ: %d, pid: %d, quantum: %d,
active: %d|\n",q0,q1,q2,p->schedQ,p->pid,p->quantum, activeQ);
        goto runProcess;
    }
}

```

```

else
    continue;

```

runProcess:

```

    counter=0;
    // Switch to chosen process. It is the process's job
    // to release ptable.lock and then reacquire it
    // before jumping back to us.
    proc = p;
    switchvm(p);
    p->state = RUNNING;
    swtch(&cpu->scheduler, proc->context);

```

//If process from initial queue, move to active queue

```

if((p->schedQ==0)&&(q0>0)){
    cprintf("SchedChange, run process pid: %d\n",p->pid);
    q0-=1;
    if(activeQ==1){
        q1+=1;//decrement new process queue counter
        p->schedQ = 1;
    }
    else if(activeQ==2){
        q2+=1;
        p->schedQ = 2;
    }
}
}

```

//If current process is in active queue (when Q1 is active queue)

```

else if((activeQ==1)&&(p->schedQ==1)&&(q1>0)){
    if(p->quantum>0){
        p->quantum-=1;//Decrement time quantum
    }
    else{//when process has run 4 times (400ms)
        p->schedQ = 2;//move process to expired queue
        p->quantum = 3;//reset time quantum
        q2+=1;//update queue counters
        q1-=1;
        cprintf(" [move pid:%d to Q2], queues: %d,%d,%d, activeQ: %d\n",p-
>pid,q0,q1,q2,activeQ);
        if(q1==0){//if last process in active queue, switch queues

```



```

        activeQ=2;
        cprintf("ActiveQ=2\n");
    }
}

//If current process is in active queue (when Q2 is active queue)
else if ((activeQ==2) && (p->schedQ==2) && (q2>0)) {
    if (p->quantum>0)
        p->quantum-=1;
    else{
        p->schedQ = 1;
        p->quantum = 3;
        q1+=1;
        q2-=1;
        cprintf(" [move pid:%d to Q1], queues: %d,%d,%d, activeQ: %d\n",p-
>pid,q0,q1,q2,activeQ);
        if (q2==0) {
            activeQ=1;
            cprintf("ActiveQ=1\n");
        }
    }
}

switchkvm();

//update cs_count
cs_count = cs_count+1;
proc = 0;

}
release(&ptable.lock);

}
}

```

---

## Ps.c

```

#include "types.h"
#include "user.h"
#include "proc.h"
int
main(int argc, char *argv[])
{
    struct pstat ps = psinfo(); // get pstat pointer
    printf(1, "PID %d\n", ps.pid[0]);
    exit();
}

```

## User.h

```

// system calls
int fork(void);
int exit(void) __attribute__((noreturn));

```

```

int wait(void);
int pipe(int*);
int write(int, void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(char*, int);
int mknod(char*, short, short);
int unlink(char*);
int fstat(int fd, struct stat*);
int psinfo(struct pstat*);
int link(char*, char*);
int mkdir(char*);
int chdir(char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int getcscount(void);

```

## **syscall.h**

```

// System call numbers
#define SYS_fork 1
#define SYS_exit 2
#define SYS_wait 3
#define SYS_pipe 4
#define SYS_read 5
#define SYS_kill 6
#define SYS_exec 7
#define SYS_fstat 8
#define SYS_chdir 9
#define SYS_dup 10
#define SYS_getpid 11
#define SYS_sbrk 12
#define SYS_sleep 13
#define SYS_uptime 14
#define SYS_open 15
#define SYS_write 16
#define SYS_mknod 17
#define SYS_unlink 18
#define SYS_link 19
#define SYS_mkdir 20
#define SYS_close 21
#define SYS_getcscount 22
#define SYS_psinfo 23

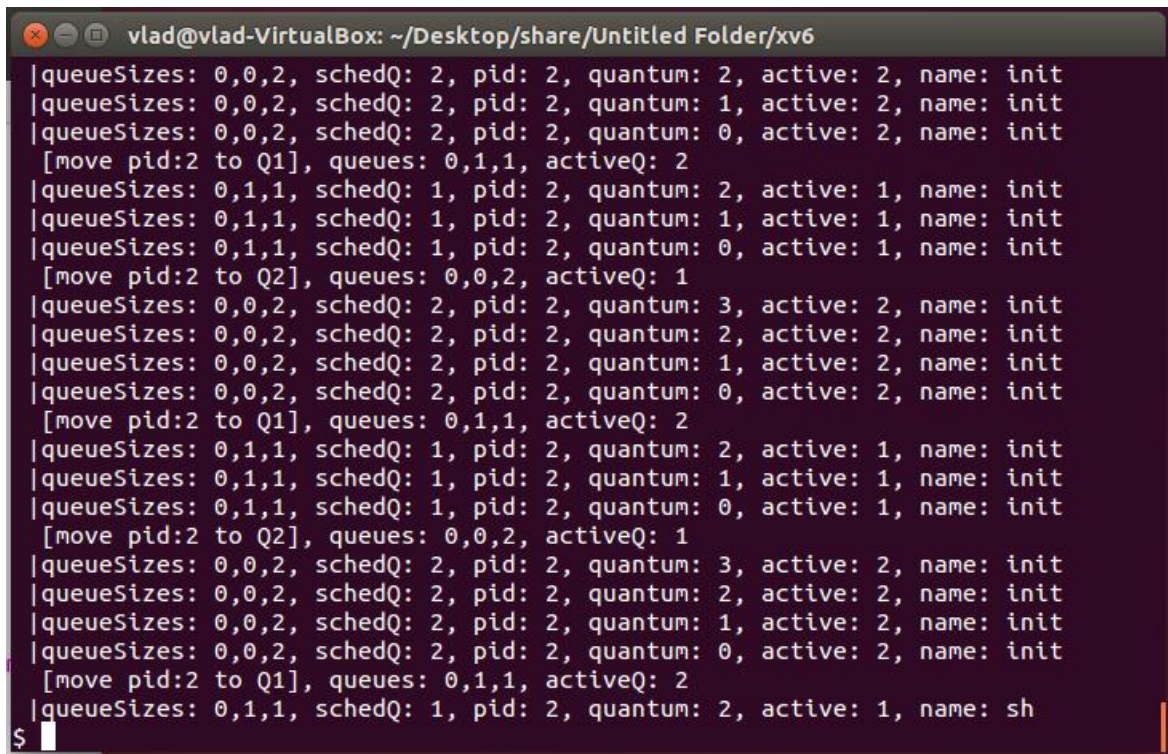
```

## Sysproc.c

```
int sys_psinfo(void)
{
    struct pstat *ps;

    if (argptr(1, (void*)&ps, sizeof(ps)) < 0)
        return -1;
    procfill(ps);
    return (int)ps;
}
```

## Experimental Results



```
vlad@vlad-VirtualBox: ~/Desktop/share/Untitled Folder/xv6
|queueSizes: 0,0,2, schedQ: 2, pid: 2, quantum: 2, active: 2, name: init
|queueSizes: 0,0,2, schedQ: 2, pid: 2, quantum: 1, active: 2, name: init
|queueSizes: 0,0,2, schedQ: 2, pid: 2, quantum: 0, active: 2, name: init
[move pid:2 to Q1], queues: 0,1,1, activeQ: 2
|queueSizes: 0,1,1, schedQ: 1, pid: 2, quantum: 2, active: 1, name: init
|queueSizes: 0,1,1, schedQ: 1, pid: 2, quantum: 1, active: 1, name: init
|queueSizes: 0,1,1, schedQ: 1, pid: 2, quantum: 0, active: 1, name: init
[move pid:2 to Q2], queues: 0,0,2, activeQ: 1
|queueSizes: 0,0,2, schedQ: 2, pid: 2, quantum: 3, active: 2, name: init
|queueSizes: 0,0,2, schedQ: 2, pid: 2, quantum: 2, active: 2, name: init
|queueSizes: 0,0,2, schedQ: 2, pid: 2, quantum: 1, active: 2, name: init
|queueSizes: 0,0,2, schedQ: 2, pid: 2, quantum: 0, active: 2, name: init
[move pid:2 to Q1], queues: 0,1,1, activeQ: 2
|queueSizes: 0,1,1, schedQ: 1, pid: 2, quantum: 2, active: 1, name: init
|queueSizes: 0,1,1, schedQ: 1, pid: 2, quantum: 1, active: 1, name: init
|queueSizes: 0,1,1, schedQ: 1, pid: 2, quantum: 0, active: 1, name: init
[move pid:2 to Q2], queues: 0,0,2, activeQ: 1
|queueSizes: 0,0,2, schedQ: 2, pid: 2, quantum: 3, active: 2, name: init
|queueSizes: 0,0,2, schedQ: 2, pid: 2, quantum: 2, active: 2, name: init
|queueSizes: 0,0,2, schedQ: 2, pid: 2, quantum: 1, active: 2, name: init
|queueSizes: 0,0,2, schedQ: 2, pid: 2, quantum: 0, active: 2, name: init
[move pid:2 to Q1], queues: 0,1,1, activeQ: 2
|queueSizes: 0,1,1, schedQ: 1, pid: 2, quantum: 2, active: 1, name: sh
$
```

Xv6 booting up

The screen above shows the output from the system kernel from xv6 initially booting up.

Since 2 processes are run, the boot up sequence of the operating system involves many

context switches. Since there are only 2 processes, active queue sizes change quite often. The code shows the decrementing of time quantum for each process(4 times) and moving the process to different queue once quantum=0. The queueSizes variables show the number of processes within each of the 3 queues at any given time and shows the scheduled queue of current process, as well as which queue is currently active(these numbers must match).

```
vlad@vlad-VirtualBox: ~/Desktop/share/Untitled Folder/xv6
spin 1000 & spin 1000 &
|queueSizes: 0,0,2, schedQ: 2, pid: 2, quantum: 1, active: 2, name: sh
Fork pid: 2
New Process, queues: 1, 0, 2, schedQ: 0, pid: 7 schedQ: 0|
Wait pid: 2
[move pid:2 to Q1], queues: 1,1,1, activeQ: 2
|queueSizes: 1,1,1, schedQ: 0, pid: 7, quantum: 3, active: 0|
SchedChange, run process pid: 7
|queueSizes: 0,1,2, schedQ: 2, pid: 7, quantum: 3, active: 2, name: sh
Fork pid: 7
New Process, queues: 1, 1, 2, schedQ: 0, pid: 8 schedQ: 0|
Wait pid: 7
|queueSizes: 1,1,2, schedQ: 0, pid: 8, quantum: 3, active: 0|
Fork pid: 8
New Process, queues: 2, 1, 2, schedQ: 0, pid: 9 schedQ: 0|
SchedChange, run process pid: 8
Exit pid: 8, queueSizes: 1,1,2, schedQ: 2, activeQ: 1
|queueSizes: 1,1,2, schedQ: 0, pid: 9, quantum: 3, active: 0|
SchedChange, run process pid: 9
|queueSizes: 0,2,2, schedQ: 1, pid: 9, quantum: 3, active: 1, name: sh
Fork pid: 7
New Process, queues: 1, 2, 2, schedQ: 0, pid: 10 schedQ: 0|
[move pid:7 to Q1], queues: 1,3,1, activeQ: 2
Exit pid: 9, queueSizes: 1,2,1, schedQ: 1, activeQ: 2
zombie!
Wait pid: 1
|queueSizes: 1,2,1, schedQ: 0, pid: 10, quantum: 3, active: 0|
Exit pid: 7, queueSizes: 1,1,1, schedQ: 1, activeQ: 1
$ SchedChange, run process pid: 10
|queueSizes: 0,2,1, schedQ: 1, pid: 10, quantum: 3, active: 1, name: sh
|queueSizes: 0,2,1, schedQ: 1, pid: 10, quantum: 2, active: 1, name: sh
|queueSizes: 0,2,1, schedQ: 1, pid: 10, quantum: 1, active: 1, name: sh
Exit pid: 10, queueSizes: 0,1,1, schedQ: 1, activeQ: 1
zombie!
Wait pid: 1
[move pid:1 to Q1], queues: 0,2,0, activeQ: 2
|queueSizes: 0,2,0, schedQ: 1, pid: 1, quantum: 3, active: 1, name: init
```

Running: Spin 1000 & spin 1000 & (Vlad Slyusar)

This screenshot illustrates the system behavior as it runs the provided test code. Printf statements were used to show when system calls are made for each process.



```
al
bendale@bendale-VirtualBox: ~/xv6
Wait pid: 2
|queueSizes: 1,0,2, schedQ: 0, pid: 10, quantum: 3, active: 0|
Exit pid: 10, queueSizes: 0,0,2, schedQ: 0, activeQ: 2
$spin 1000 & spin 2000 &
Fork pid: 2
New Process, queues: 1, 0, 2, schedQ: 0, pid: 11 schedQ: 0|
Wait pid: 2
|queueSizes: 1,0,2, schedQ: 0, pid: 11, quantum: 3, active: 0|
Fork pid: 11
SchedChange, run process pid: 11
|queueSizes: 0,0,3, schedQ: 2, pid: 11, quantum: 3, active: 2|
New Process, queues: 1, 0, 3, schedQ: 0, pid: 12 schedQ: 0|
Wait pid: 11
|queueSizes: 1,0,3, schedQ: 0, pid: 12, quantum: 3, active: 0|
SchedChange, run process pid: 12
|queueSizes: 0,0,4, schedQ: 2, pid: 12, quantum: 3, active: 2|
|queueSizes: 0,0,4, schedQ: 2, pid: 12, quantum: 2, active: 2|
|queueSizes: 0,0,4, schedQ: 2, pid: 12, quantum: 1, active: 2|
|queueSizes: 0,0,4, schedQ: 2, pid: 12, quantum: 0, active: 2|
Fork pid: 12
New Process, queues: 1, 0, 4, schedQ: 0, pid: 13 schedQ: 0|
[move pid:12 to Q1], queues: 1,1,3, activeQ: 2
|queueSizes: 1,1,3, schedQ: 0, pid: 13, quantum: 3, active: 0|
eExit pid: 12, queueSizes: 1,0,3, schedQ: 1, activeQ: 1
Fork pid: 11
SchedChange, run process pid: 13
|queueSizes: 0,0,4, schedQ: 2, pid: 13, quantum: 3, active: 2|
xecspin failed
Exit pid: 13, queueSizes: 0,0,3, schedQ: 2, activeQ: 2
|queueSizes: 0,0,3, schedQ: 2, pid: 1, quantum: 1, active: 2|
|queueSizes: 0,0,3, schedQ: 2, pid: 1, quantum: 0, active: 2|
[move pid:1 to Q1], queues: 0,1,2, activeQ: 2
zombie!
```

Running: Spin 1000 & spin 2000 & (*Ben Dale*)

```
vlad@vlad-VirtualBox: ~/Desktop/share/Untitled Folder/xv6
$ ps
|queueSizes: 0,1,1, schedQ: 1, pid: 2, quantum: 1, active: 1, name: sh
|queueSizes: 0,1,1, schedQ: 1, pid: 2, quantum: 0, active: 1, name: sh
Fork pid: 2
[move pid:2 to Q2], queues: 0,0,2, activeQ: 1
|queueSizes: 0,0,2, schedQ: 2, pid: 2, quantum: 3, active: 2, name: sh
New Process, queues: 1, 0, 2, schedQ: 0, pid: 3 schedQ: 0|
Wait pid: 2
|queueSizes: 1,0,2, schedQ: 0, pid: 3, quantum: 3, active: 0|
SchedChange, run process pid: 3
|queueSizes: 0,0,3, schedQ: 2, pid: 3, quantum: 3, active: 2, name: sh
|queueSizes: 0,0,3, schedQ: 2, pid: 3, quantum: 2, active: 2, name: sh
|queueSizes: 0,0,3, schedQ: 2, pid: 3, quantum: 1, active: 2, name: sh
|queueSizes: 0,0,3, schedQ: 2, pid: 3, quantum: 0, active: 2, name: sh
[move pid:3 to Q1], queues: 0,1,2, activeQ: 2
|queueSizes: 0,1,2, schedQ: 1, pid: 3, quantum: 2, active: 1, name: sh
|queueSizes: 0,1,2, schedQ: 1, pid: 3, quantum: 1, active: 1, name: sh
|queueSizes: 0,1,2, schedQ: 1, pid: 3, quantum: 0, active: 1, name: sh
[move pid:3 to Q2], queues: 0,0,3, activeQ: 1
|queueSizes: 0,0,3, schedQ: 2, pid: 3, quantum: 3, active: 2, name: sh
|queueSizes: 0,0,3, schedQ: 2, pid: 3, quantum: 2, active: 2, name: sh
|queueSizes: 0,0,3, schedQ: 2, pid: 3, quantum: 1, active: 2, name: sh
|queueSizes: 0,0,3, schedQ: 2, pid: 3, quantum: 0, active: 2, name: sh
[move pid:3 to Q1], queues: 0,1,2, activeQ: 2
|queueSizes: 0,1,2, schedQ: 1, pid: 3, quantum: 2, active: 1, name: sh
|queueSizes: 0,1,2, schedQ: 1, pid: 3, quantum: 1, active: 1, name: sh
|queueSizes: 0,1,2, schedQ: 1, pid: 3, quantum: 0, active: 1, name: sh
PID  PNAME  #QUEUE
2    sh     2
1    [move pid:3 to Q2], queues: 0,0,3, activeQ: 1
|queueSizes: 0,0,3, schedQ: 2, pid: 3, quantum: 3, active: 2, name: ps
init  1
|queueSizes: 0,0,3, schedQ: 2, pid: 3, quantum: 2, active: 2, name: ps
|queueSizes: 0,0,3, schedQ: 2, pid: 3, quantum: 1, active: 2, name: ps
3    ps     0
Exit pid: 3, queueSizes: 0,0,2, schedQ: 2, activeQ: 2
|queueSizes: 0,0,2, schedQ: 2, pid: 2, quantum: 1, active: 2, name: sh
$
```

Output from ps.c (*Step by step output from cprintf was kept*)

## Conclusion

This project introduced a great amount of experience with not only scheduling algorithms but writing physical operating system code itself. With the specifications and functionality defined, the scheduling algorithms was simple to implement on paper; however, when it came to making large changes to the system kernel many unexpected issues came up. The project was completed successfully only due to a slow approach

Department of Electrical and Computer Engineering  
College of Engineering and Computer Science  
The University of Michigan-Dearborn

making only slight modifications and compiling right away to check functionality. Once a structure was implemented to place processes into queues and switch active queues, it became evident that some processes exit completely and must be removed from their queue. Furthermore, an even larger problem to overcome was processes in wait or sleep state. Such a process within an active queue would not run and therefore queue switch would never take place to allow other processes to run. The infinite loop issue was overcome in a simple way using a counter variable and switching active queues when the current queue was not empty but no processes could run. (A more efficient implementation would have involved keeping track of waiting processes using separate variables, but since xv6 uses only a limited number of processes the counter approach does not degrade the scheduler performance and was much simpler to implement). As far as `ps.c`, the `psinfo` code was implemented following the guidelines for the project. The system calls throughout the scheduler provide adequate information for each process including the pid, and scheduled queue. The breakdown of processes in each queue is also provided, as well as the pointer designator of the current active queue. When calling `ps`, the system calls show the progression of processes while the PID table is also printed to screen. Multiple processes print to the screen so `ps.c` was interrupted by such output several times.

## **Work Breakdown**

The completion of this project required both partners (Vlad Slyusar, Ben Dale) to collaborate and solve the many issues that arose during this project. Scheduler code was written separately by both partners, but in the end various parts were combined to create the fully functional scheduler. (More time was spent on debugging and figuring out functionality than actually writing the code). Report collaboration was shared using a google document.