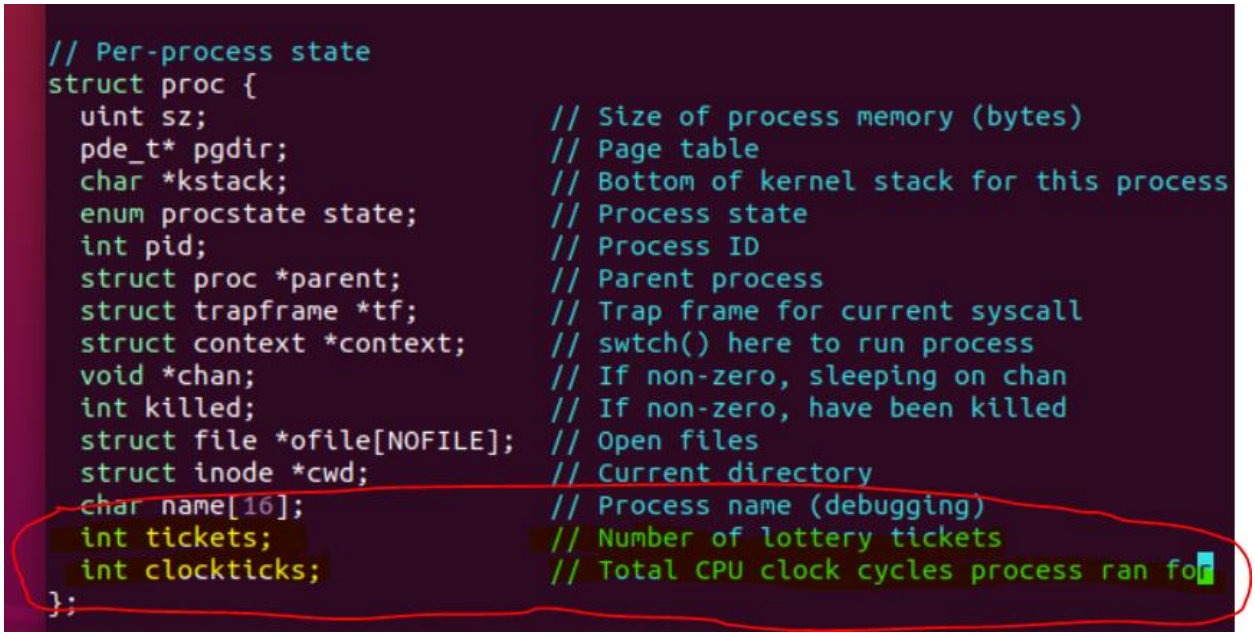
Student Name: Demetrius Johnson
Course: CIS-450
Professor: Dr. Jinhua Guo
Date: April 10, 2022
Due Date: April 6, 2022

Project 4 Report

Note: I uploaded my xv6 file as a tar file so that you can open it and run it yourself if you like.

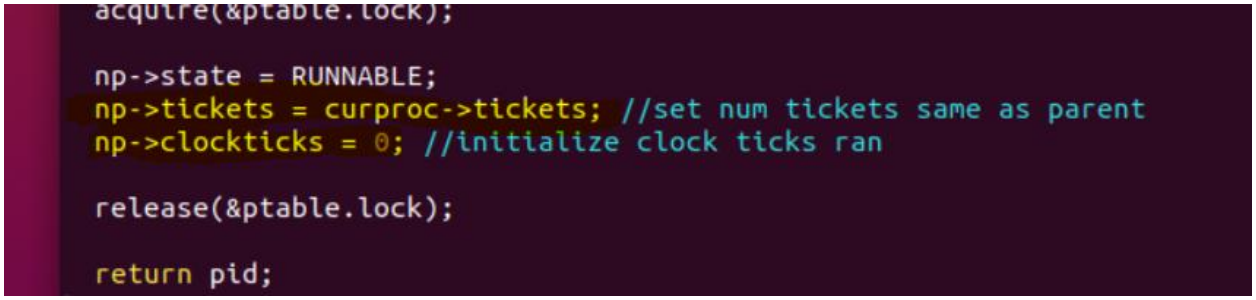
Lottery Scheduler in xv6:

Task 1: adding variables to the proc struct in proc.c



```
// Per-process state
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;     // swtch() 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 tickets;                 // Number of lottery tickets
  int clockticks;              // Total CPU clock cycles process ran for
};
```

Task 2: add necessary lines to fork() in proc.c



```
  acquire(&ptable.lock);

  np->state = RUNNABLE;
  np->tickets = curproc->tickets; //set num tickets same as parent
  np->clockticks = 0; //initialize clock ticks ran

  release(&ptable.lock);

  return pid;
```

Task 3: add necessary lines to userinit() in proc.c

```
  // because the assignment might not be atomic.
  acquire(&ptable.lock);

  p->state = RUNNABLE;
  p->tickets = 10;      //default number of lottery tickets = 10
  p->clockticks = 0;    //initialize num ticks ran to 0

  release(&ptable.lock);
}

// Grow current process's memory by n bytes.
"proc.c" 538L, 11954C written                                153,54
```

Task 4: adding the pstat.h file to xv6 directory

```
File  Edit  View  Search  Terminal  Help
#ifndef _PSTAT_H
#define _PSTAT_H_

#include "param.h"

struct pstat
{
  int num_processes;
  int pid[NPROC];     //the PID of each process
  int tickets[NPROC]; //the number of tickets each process has
  int ticks[NPROC];   //the number of ticks each process has accumulated
};

#endif //_PSTAT_H_
```

The pstat struct will be used by the provided ps.c function, which calls getpinfo() and passes a reference to the pstat struct it declares to that function → getpinfo(&pstatStructDeclared);

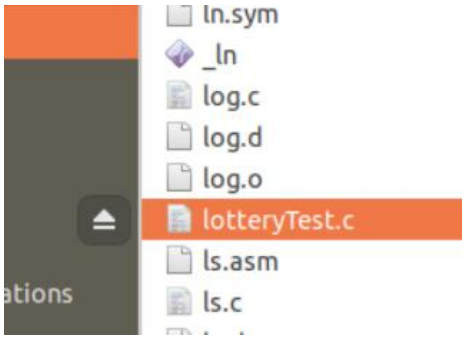
Task 5: adding the ps.c file to xv6 directory

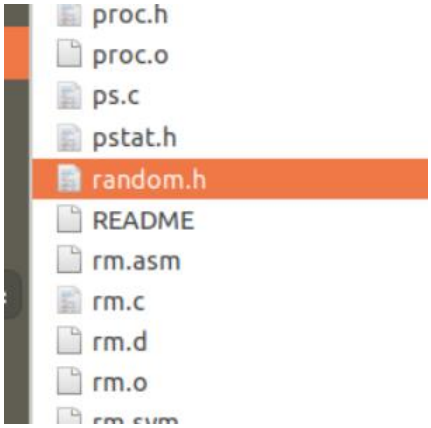
File Edit View Search Terminal Help

```
#include "types.h"
#include "mmu.h"
#include "param.h"
#include "proc.h"
#include "user.h"
#include "pstat.h"

int main(int argc, char *argv[])
{
    struct pstat info = {};

    getpinfo(&info);
    printf(1, "PID\tTICKETS\tTICKS\n");
    for (int i = 0; i < info.num_processes; ++i) {
      printf(1, "%d\t%d\t%d\n", info.pid[i], info.tickets[i], info.ticks[i]);
    }
    exit();
}
```

Task 6: adding random.h and lotteryTest.c file to xv6 directory

Because clipboard couldn't hold all of the text, I simply added both files to my xv6 directory via direct file copy from my local system to my virtual machine. For the other files provided that I needed to add

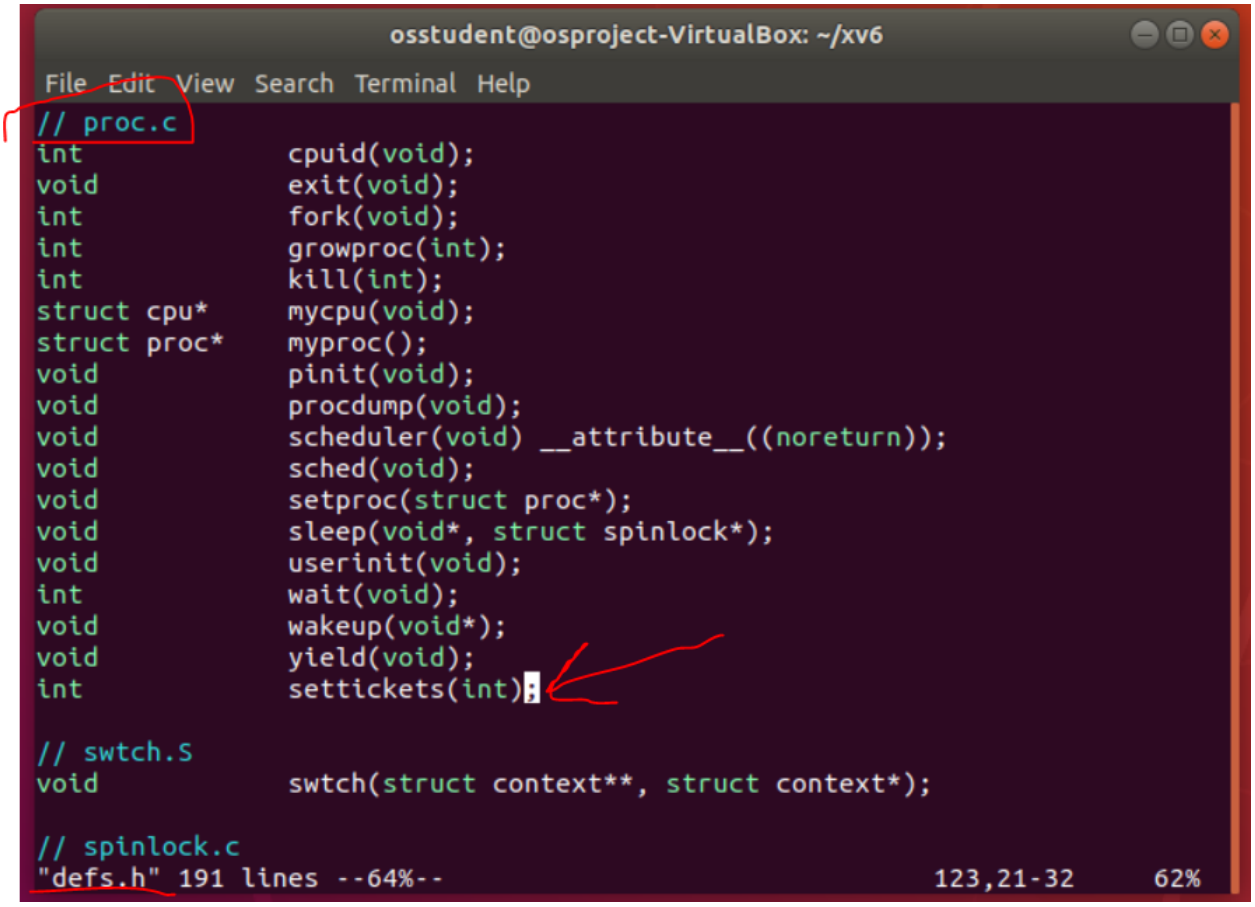


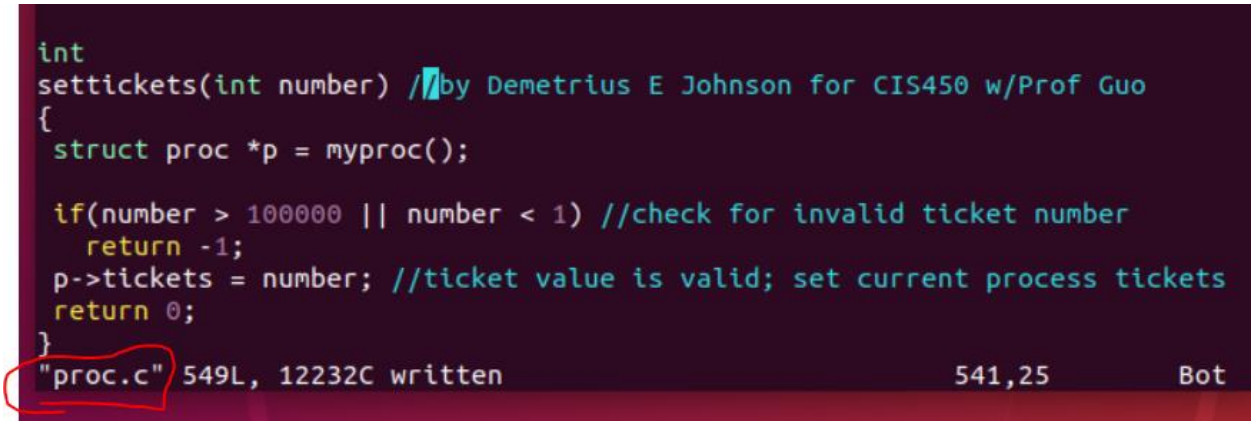


Notice ps.c and pstat.h are also added already from my previous tasks.

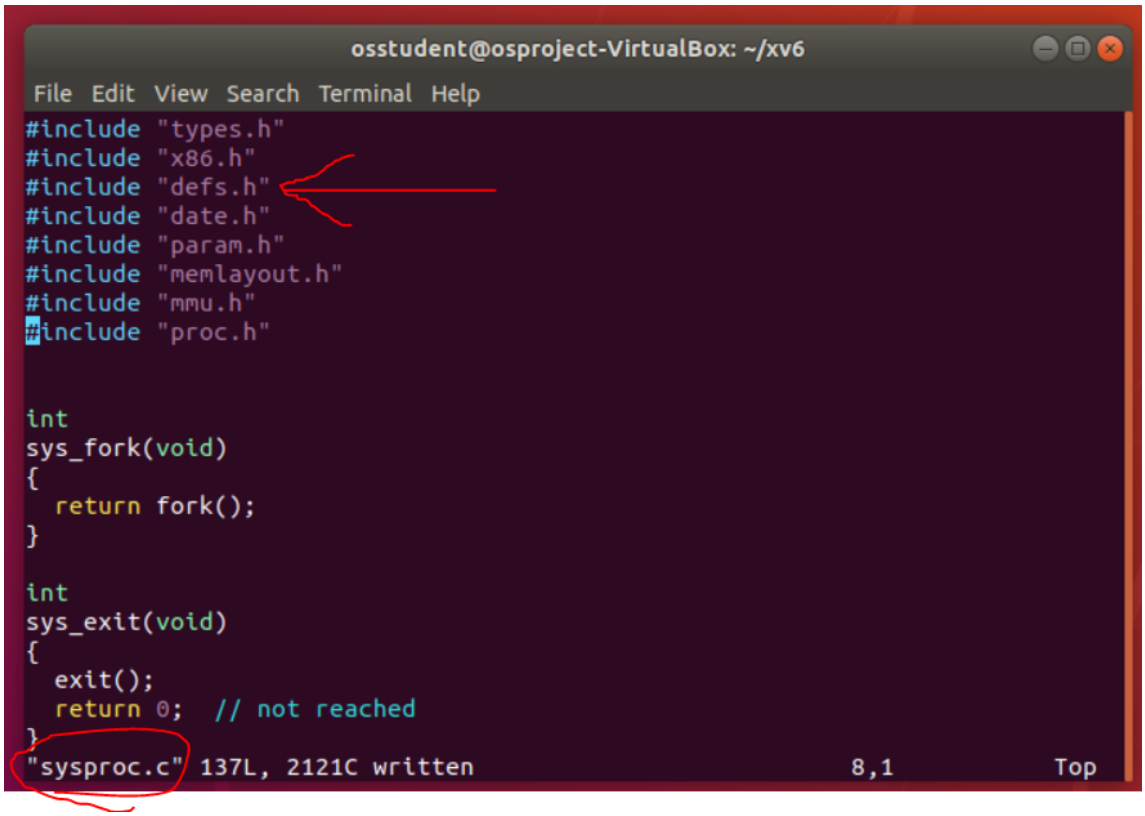
Task 7: writing the settickets() system call

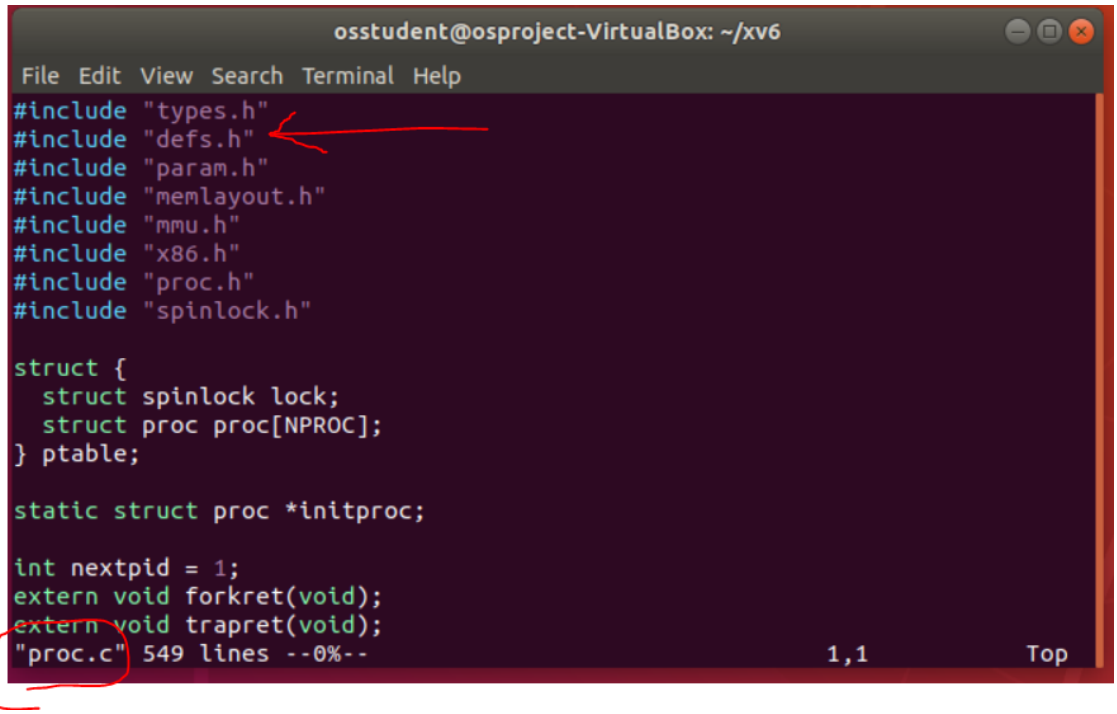
First, I added the implementation for the kernel-level function in the proc.c file, since proc.c contains all of the libraries/includes and data variables that I need to access for updating the process control block struct, and I want to follow the convention for the other system calls, which are all mostly wrapper functions. In order for the settickets() function written here in proc.c to be seen by the sys_call function I will write in the sysproc.c file, its declaration needs to be added into the defs.h file, which is the declaration file that is included in the main.c file and many other files that are linked with main.c and thus the preprocessor directive makes all function declarations that are defined later in various other files visible:



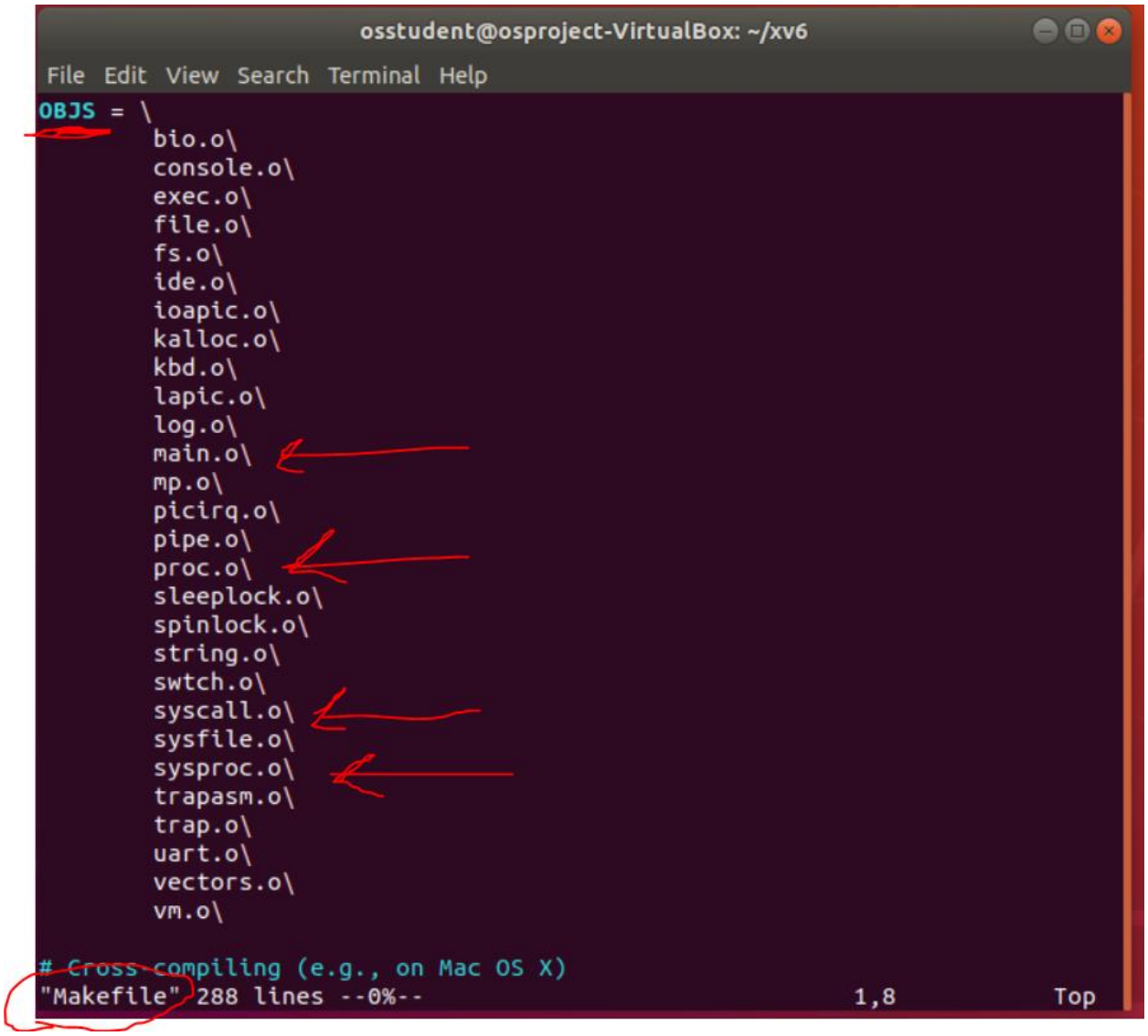


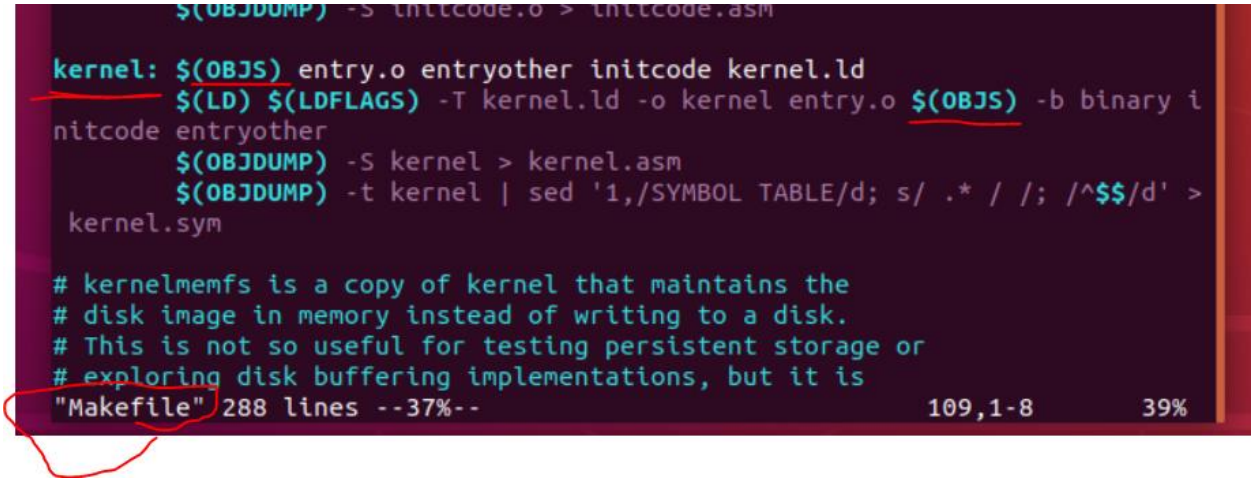
Note: defs.h is also included in the sysproc.c file, as well as the proc.c file. However, for our situation, we specifically need the defs.h included in the sysproc.c file, since this is where our sys_settickets() function resides and calls the settickets() function from proc.c:





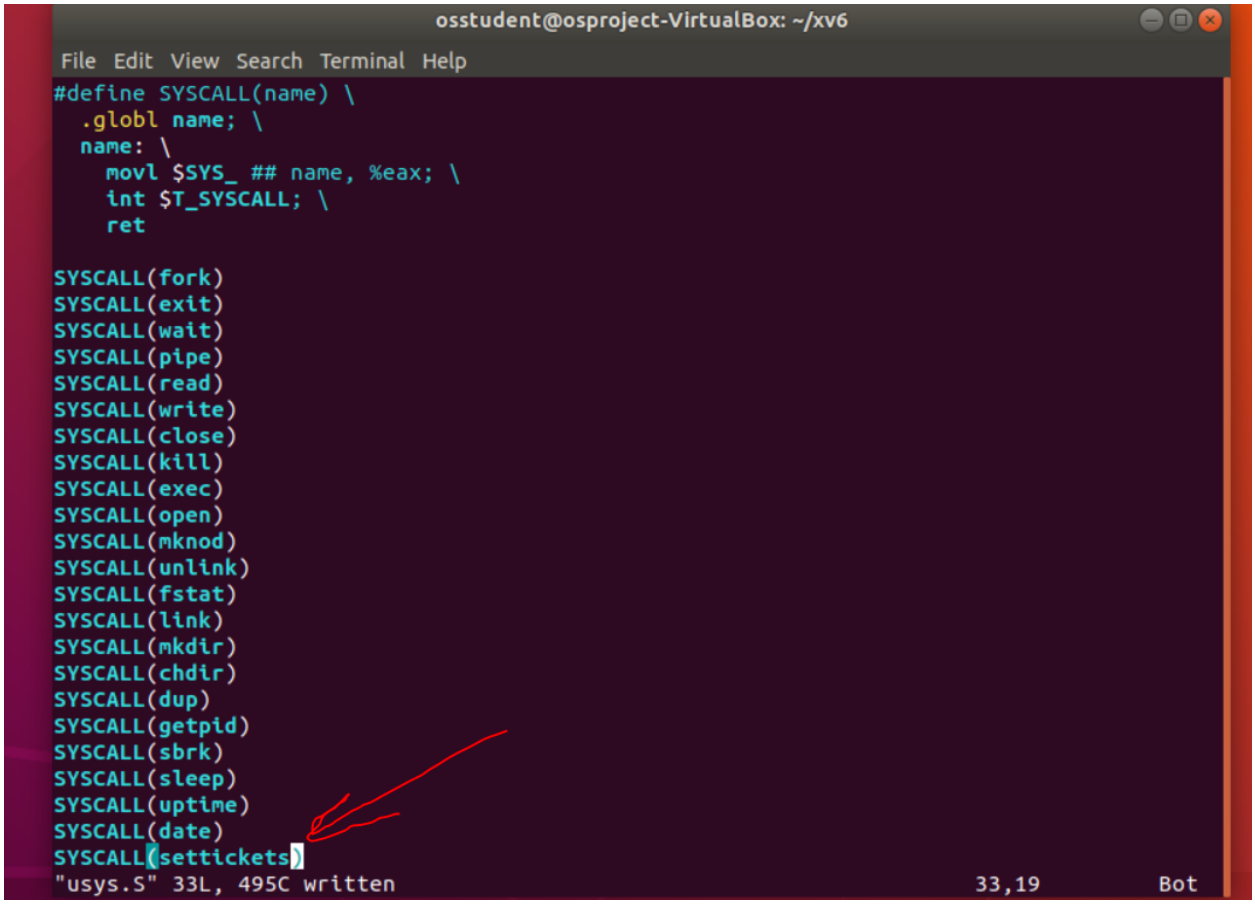
As another side note: here are all the files (stored in an array called OBJ) linked to the main.o (the main program) for xv6, specifically the kernel; "kernel" is the name given to the executable when all objects from OBJ array are linked:



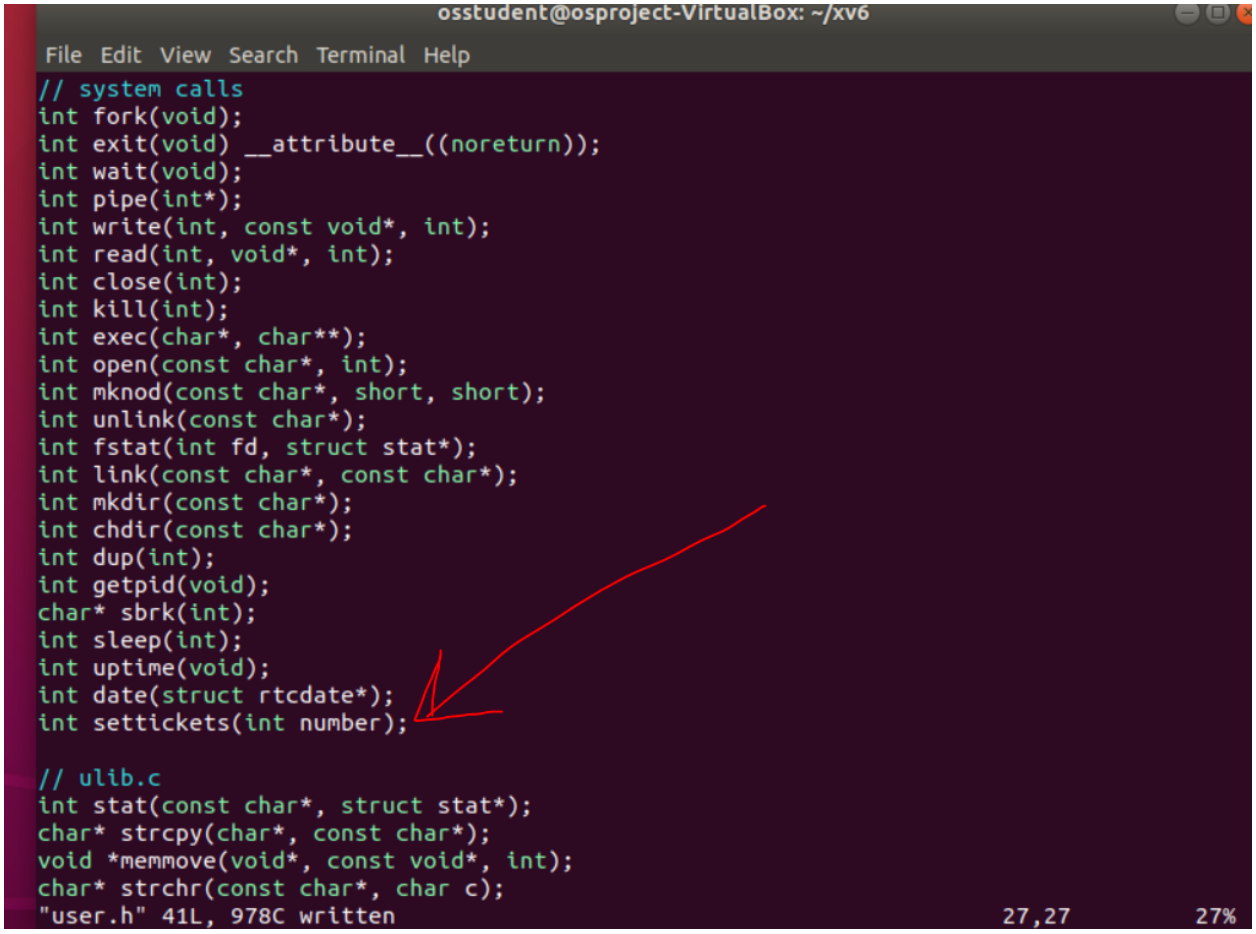

$(OBJDUMP) -S initcode.o > initcode.asm
kernel: $(OBJS) entry.o entryother initcode kernel.ld
$(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary i
nitcode entryother
$(OBJDUMP) -S kernel > kernel.asm
$(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' >
kernel.sym
kernelmemfs is a copy of kernel that maintains the
disk image in memory instead of writing to a disk.
This is not so useful for testing persistent storage or
exploring disk buffering implementations, but it is
"Makefile" 288 lines --37%--
109,1-8
39%


entryother
kalloc.o
picirq.d
string.c
vm.c
entryother.asm
kbd.c
picirq.o
string.d
vm.d
entryother.d
kbd.d
pipe.c
string.o
vm.o
entryother.o
kbd.h
pipe.d
swtch.o
wc.c
entryother.S
kbd.o
pipe.o
swtch.S
x86.h
entry.S
kernel
printf.c
syscall.c
xv6.img
exec.c
kernel.asm
printpcs
syscall.d
zombie.c
exec.d
kernel.ld
proc.c
syscall.h
osstudent@osproject-VirtualBox:~/xv6$

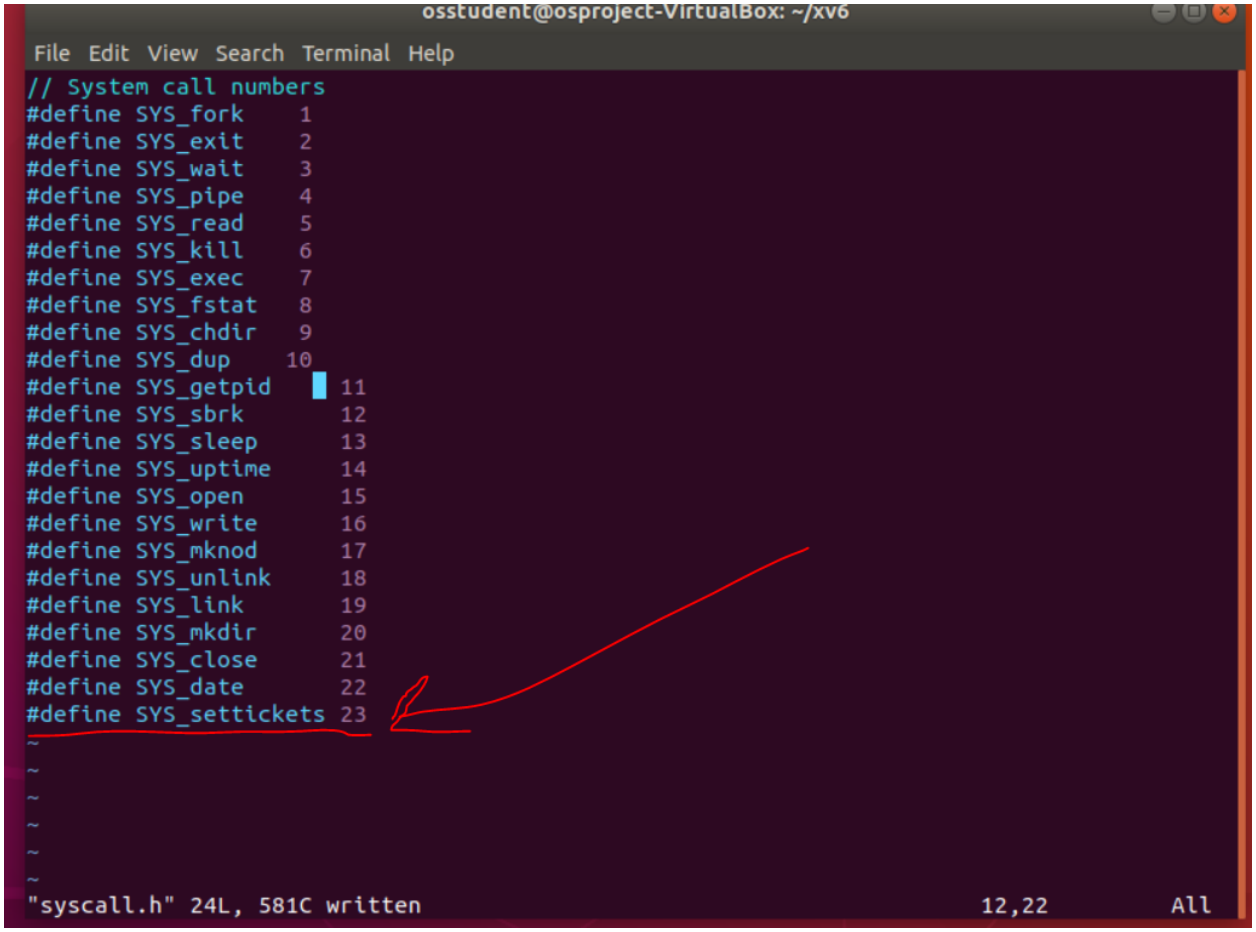
Now, I add the system call name to the usys.S file so that calling user function can interface with the system call (switching from the call in user.h to this call, which will lead to kernel mode and the kernel sys_call function in sysproc.c):



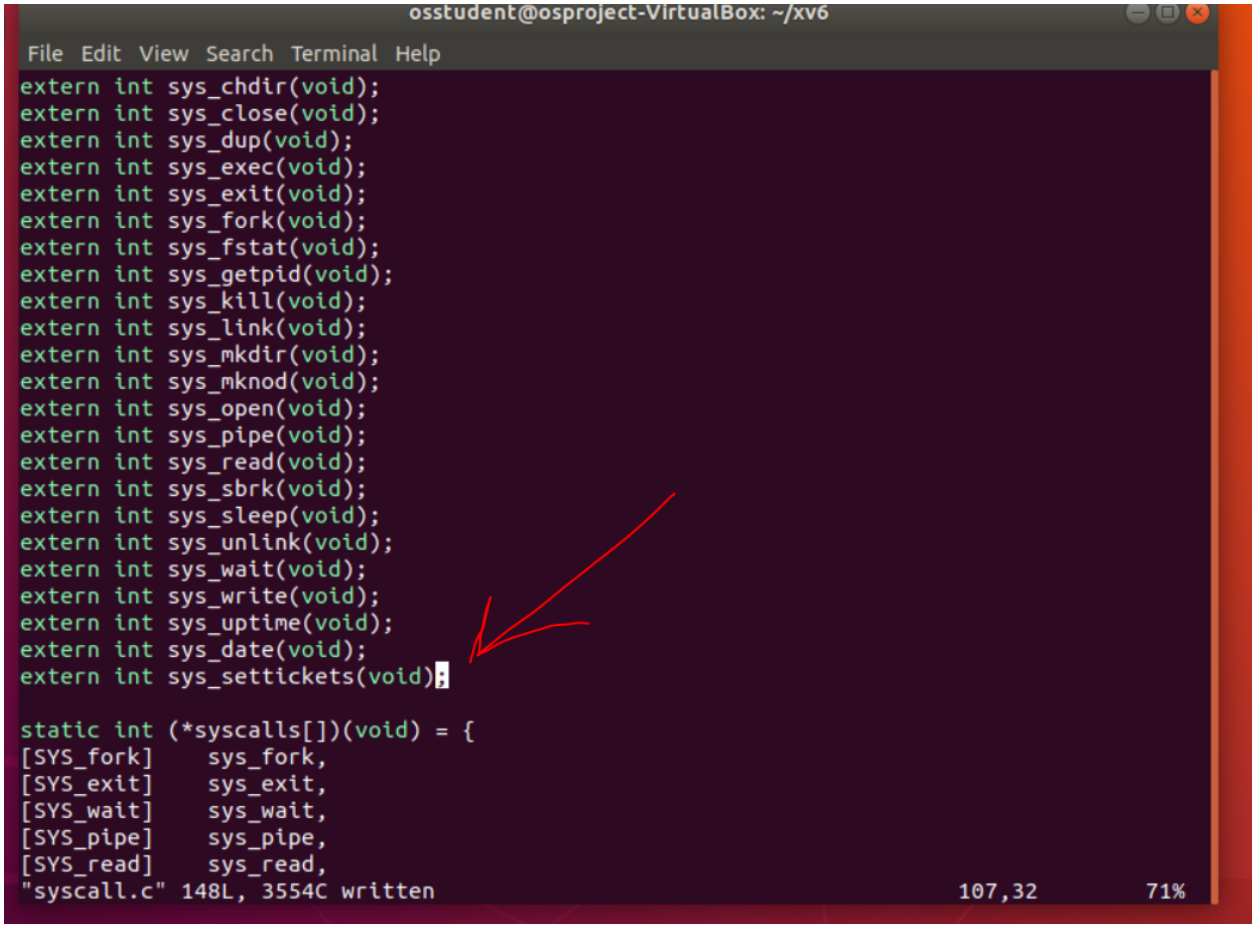
Now, add the system call to user.h file, which is not actually an implemented function, but rather transfers control from user mode to kernel mode and calls the associated system call function:

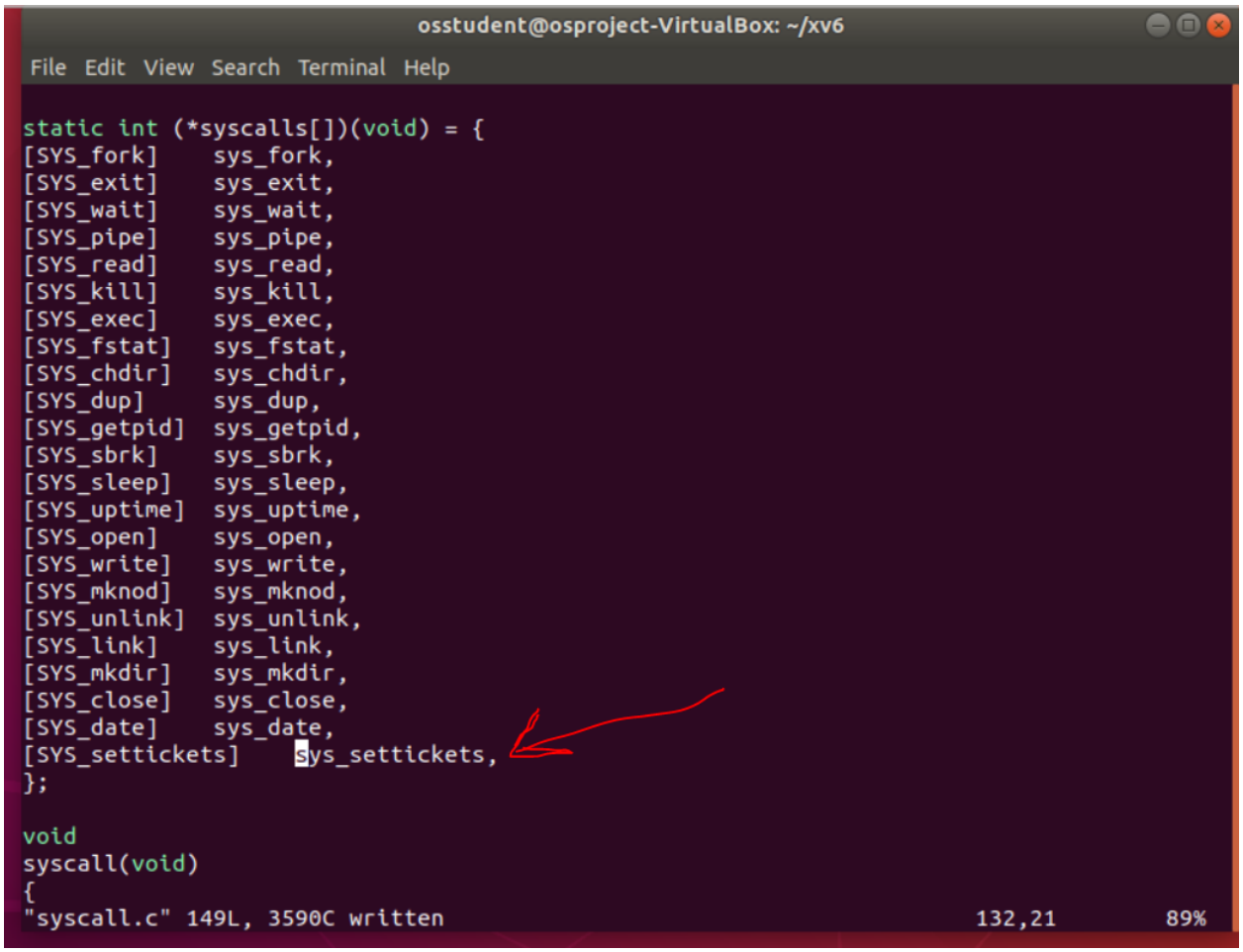


Now, define set tickets system call value in syscall.h file, which give the associated index value (in this case 23) for indexing into the function pointer array in syscall.c:

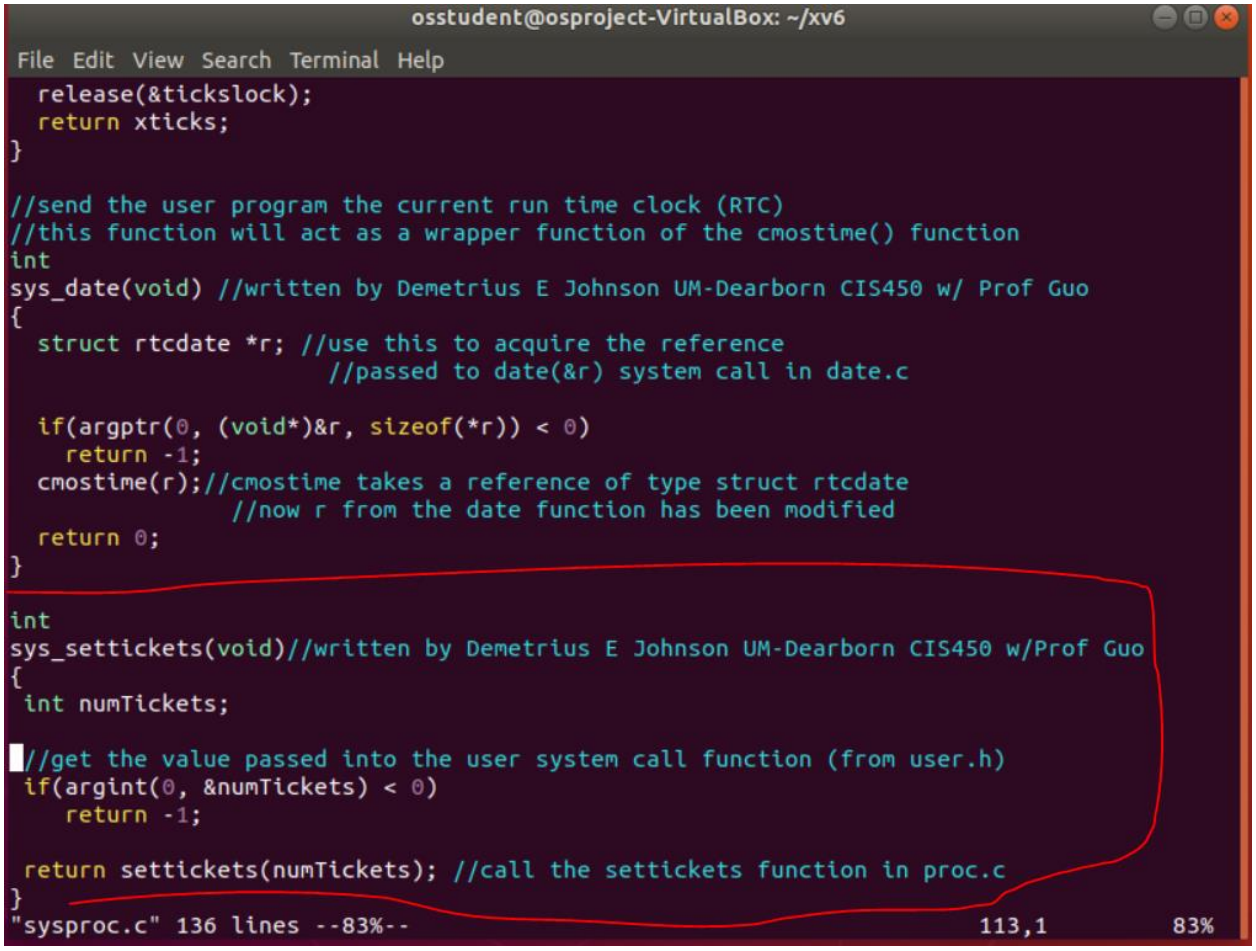


Then, define an external function and assign the value we added from the previous step (SYS_settickets == 23) to the syscalls[] array – both of these are done in syscall.c file:

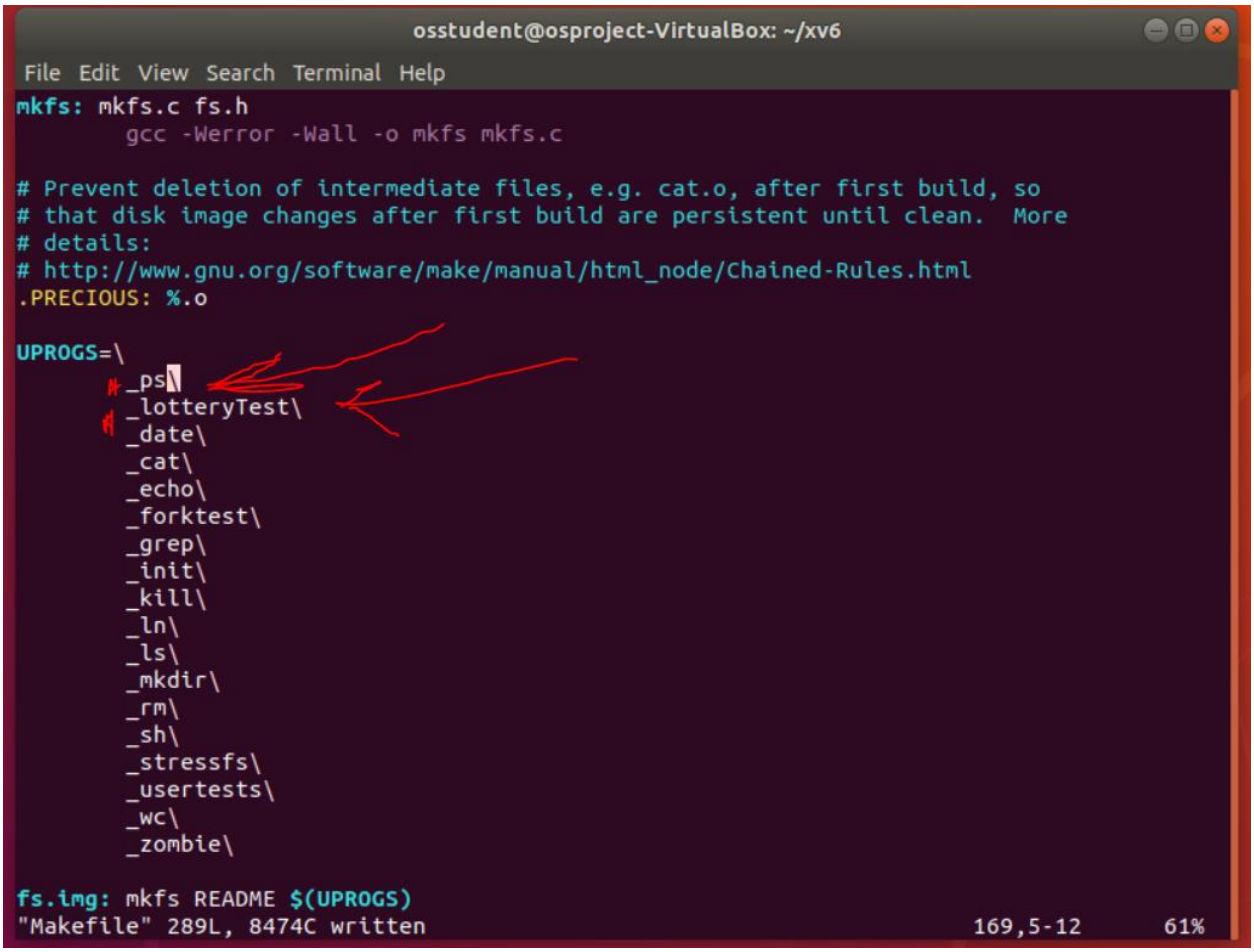



osstudent@osproject-VirtualBox: ~/xv6
File Edit View Search Terminal Help
static int (*syscalls[])(void) = {
[SYS_fork]   sys_fork,
[SYS_exit]   sys_exit,
[SYS_wait]   sys_wait,
[SYS_pipe]   sys_pipe,
[SYS_read]   sys_read,
[SYS_kill]   sys_kill,
[SYS_exec]   sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]   sys_dup,
[SYS_getpid] sys_getpid,
[SYS_sbrk]   sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime] sys_uptime,
[SYS_open]   sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink] sys_unlink,
[SYS_link]   sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_date]   sys_date,
[SYS_settickets]   sys_settickets,
};
void
syscall(void)
{
"syscall.c" 149L, 3590C written
132,21
89%

Now, I write the sys_settickets system call that the user function calls. For this function, I simply acquire the argument passed by the user function (from user.h function) using the argint() function, and then make this system call a wrapper function (following the convention of most of the other system calls) by calling the settickets() function from the proc.c file and returning its value. As a side note, when the user.h function is called that leads to the actual system call, the arguments passed to it are saved in registers so that they can be acquired by the sys_call function.



Task 8: adding entries to the Makefile so that lotteryTest.c and ps.c programs will be available as user programs



Task 9: writing the getpinfo() system call

Essentially, I follow the same steps as with adding the settickets() system call; first I add the declaration to defs.h:

```
// proc.c
int             cpuid(void);
void            exit(void);
int             fork(void);
int             growproc(int);
int             kill(int);
struct cpu*     mycpu(void);
struct proc*    myproc();
void            pinit(void);
void            procdump(void);
void            scheduler(void) __attribute__((noreturn));
void            sched(void);
void            setproc(struct proc*);
void            sleep(void*, struct spinlock*);
void            userinit(void);
int             wait(void);
void            wakeup(void*);
void            yield(void);
int             settickets(int);
int             getpinfo(struct pstat*);

// swtch.S
void            swtch(struct context**, struct context*);

// spinlock.c
"defs.h" 192L, 5592C written                                  124,29-40     61%
```

Add my implementation of getpinfo in proc.c. Again, my sys_call will just be a wrapper function; it will just pass the parameter to this function in proc.c:

```
int
getpinfo(struct pstat *pstats) //by Demetrius E Johnson for CIS450 w/Prof Guo
{
 struct proc *p;
 int pstatLoc = 0;

 if(pstats == 0) //make sure pstat struct pointer is valid
   return -1;

 acquire(&ptable.lock);//ACQUIRE PTABLE LOCK
 pstats->num_processes = 0; //initialize or reinitialize num processes
 //p iterates up to the last address in the proc array from the ptable struct:
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != UNUSED){
            pstats->num_processes++;
            pstats->pid[pstatLoc] = p->pid;
            pstats->tickets[pstatLoc] = p->tickets;
            pstats->ticks[pstatLoc] = p->clockticks;
            pstatLoc++; //prepare for next location to update in the pstat struct array


    }
  }
  release(&ptable.lock);//RELEASE PTABLE LOCK
  return 0;
}
"proc.c" 575 lines --100%--                                    575,1         Bot
```

Update usys.S

```
SYSCALL(date)
SYSCALL(settickets)
SYSCALL(getpinfo)
"usys.S" 34L, 513C written
```

Update user.h, adding the new system call function and giving pstat struct declaration so that the function knows about such data structure:

```
                    osstuden
File  Edit  View  Search  Terminal
struct stat;
struct rtcdate;
struct pstat;

// system calls
int fork(void);
int exit(void) __attribute_
"user.h" 43 lines --6%--
```

```
int sleep(int);
int uptime(void);
int date(struct rtcdate*);
int settickets(int);
int getpinfo(struct pstat*);
"user.h" 42 lines --14%--
```

Update syscall.h and syscall.c

```
#define SYS_date       22
#define SYS_settickets 23
#define SYS_getpinfo   24
~
~
~
~
~
"syscall.h" 25L, 607C written
```

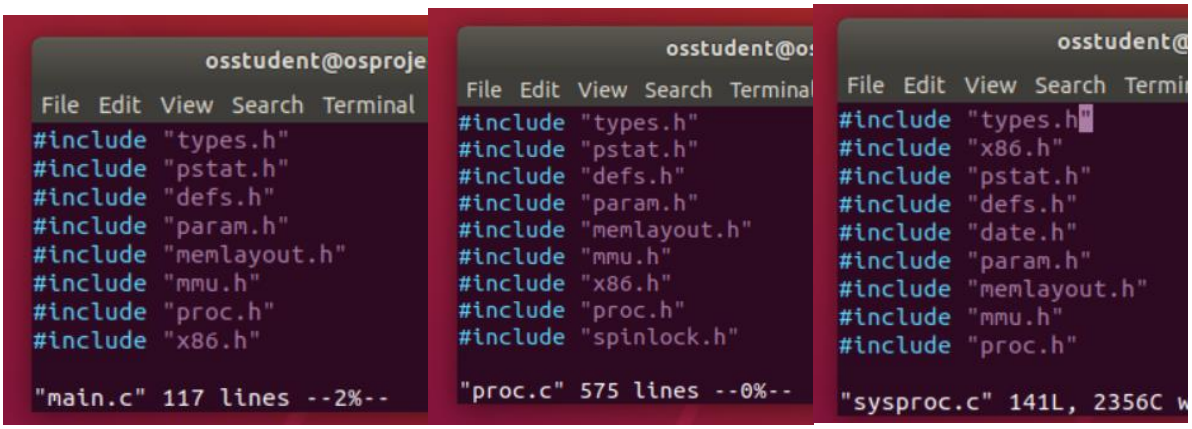
```
extern int sys_date(void);
extern int sys_settickets(void);
extern int sys_getpinfo(void);

static int (*syscalls[])(void) = {
"syscall.c" 150L, 3621C written
```

```
[SYS_date]    sys_date,
[SYS_settickets]    sys_settickets,
[SYS_getpinfo]      sys_getpinfo,
};

void
syscall(void)
{
  int num;
"syscall.c" 151L, 3655C written
```

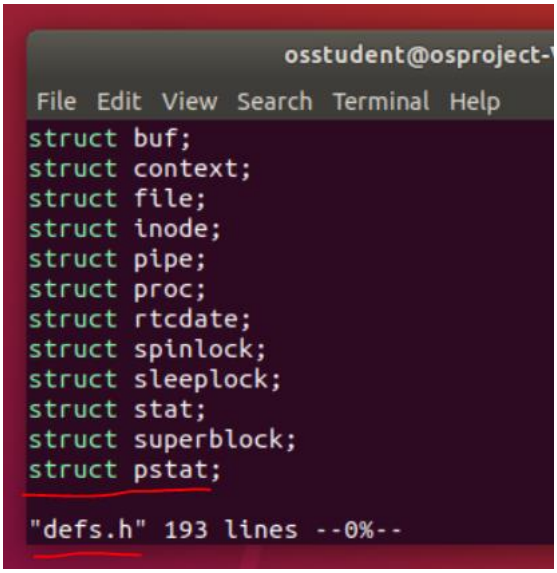
Update sysproc.c; write the sys_call function, which I made as just a wrapper function:

```
int
sys_getpinfo(void) //written by Demetrius E Johnson UM-Dearborn CIS450 w/Prof Guo
{
 struct pstat *pstats;

 if(argptr(0, (void*)&pstats, sizeof(*pstats)) < 0)
    return -1;//if no valid arguments, i.e. NULL pointer, return with error (-1)

 return getpinfo(pstats);//call getpinfo function in proc.c
}

"sysproc.c" 141 lines --90%--
```

Update sysproc.c, proc.c, and defs.h to include pstat.h file so that they can see the definition of the pstat struct that used in their respective file (and update main to include it too just in case I end up using the struct in main later!)



Update defs.h file, which needs the declaration of the struct that it uses:



Update ps.c to output more information (which will be useful to make sure it is working). I added some printf() lines:

```
#include "param.h"
#include "proc.h"
#include "user.h"
#include "pstat.h"

int main(int argc, char *argv[])
{
    struct pstat info = {};
    int test;

    test =  getpinfo(&info);
    if(test == -1)
      printf(1, "getpinfo failed\n");
    else
      printf(1, "getpinfo succeeded\n");
    printf(1, "CurrTot# non-UNUSED (used) processes = %d\n", info.num_processes);
    printf(1, "PID\tTICKETS\tTICKS\n");
    for (int i = 0; i < info.num_processes; ++i) {
      printf(1, "%d\t%d\t%d\n", info.pid[i], info.tickets[i], info.ticks[i]);
    }
    exit();
}
"ps.c" 25L, 597C written                                    18,19         66
```

Task 10: writing the yield() system call

Update user.h. (Note: this will be another wrapper function, and a very simple system call to add):

```
int settickets(int);
int getpinfo(struct pstat*);
int yield(void);

// ulib.c
int stat(const char*, struct stat*
char* strcpy(char*, const char*);
void *memmove(void*, const void*,
char* strchr(const char*, char c);
int strcmp(const char*, const char
void printf(int, const char*, ...)
"user.h" 44L, 1031C written
```

Update usys.S

```
SYSCALL(settickets)
SYSCALL(getpinfo)
SYSCALL(yield)
"usys.S" 35L, 528C written
```

Update syscall.h

```
#define SYS_getpinfo    24
#define SYS_yield       25
"syscall.h" 26L, 633C written
```

Update syscall.c

```
[SYS_settickets]    sys_settickets,
[SYS_getpinfo]      sys_getpinfo,
[SYS_yield]         sys_yield,
};

void
syscall(void)
{
"syscall.c" 152L, 3686C written
```

```
extern int sys_settickets(void);
extern int sys_getpinfo(void);
extern int sys_yield(void);

"syscall.c" 153 lines --61%--
```

Update sysproc.c – as you see, again, our sys_call function is just a wrapper function:

```
int
sys_yield(void) //written by Demetrius E Johnson UM-Dearborn CIS450 w/Prof Guo
{

 yield();
 return 0;
}
"sysproc.c" 141L, 2371C written                          140,0-1       Bot
```

Task 11: writing the schedule() function in proc.c to make it a lottery scheduler

First, include random.h and date.h in the proc.c file since I will be using the srand() and rand() functions. Seed value for srand is based on current system time, so I will use the rtcdate struct, found in date.h

```
File  Edit  View  Search  Termin
#include "types.h"
#include "pstat.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "x86.h"
#include "proc.h"
#include "spinlock.h"
#include "random.h"
#include "date.h"
"proc.c" 590 lines --0%--
```

Now, here are all of the screenshots for my lottery scheduler in proc.c. It contains a lot of commenting so that you can easily read it and follow its flow

```
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run
//  - swtch to start running that process
//  - eventually that process transfers control
//      via swtch back to the scheduler.
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  struct rtcdate systime;
  int seedtime;
  int winner;
  int curr_sum_tickets;
  int total_tickets;
  cmostime(&systime); //sum of system time fields = lottery scheduler seed
  seedtime = (    systime.second
                + systime.minute
                + systime.hour
                + systime.day
                + systime.month
                + systime.year
             );
  srand(seedtime);//generate seed based on system time for the random() function
  cprintf("srand() input seed set based on current system time.\n");
  cprintf("Input for srand() seed was: %d\n", seedtime);
  cprintf("srand() generated seed successfully...\nLottery scheduler ready...\n");

  c->proc = 0;//current process running on this processor is null (only scheduler is running)

  for(;;){
```

```
cprintf("srand() generated seed successfully...\nLottery scheduler ready...\n");

c->proc = 0;//current process running on this processor is null (only scheduler is running)

for(;;){
  // Enable interrupts on this processor.
  sti();

  //Lottery scheduler: winner of lottery has enough tickets to win.
  //Winner is generated using the rand() function.
  //Randomnly generated winner range must be within sum of tickets of
  //all runnable processes to guarantee a winner.
  //Each time winner not found, add to search total (curr_sum_tickets)
  //Need to acquire lock: only one CPU allowed to schedule user process at a time.
  acquire(&ptable.lock);
  //get current total tickets in system
  total_tickets = 0;
  curr_sum_tickets = 0;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
     if(p->state == RUNNABLE)
       total_tickets += p->tickets; //ticket sum only for RUNNABLE (ready) procs.
  //generate winning ticket
  winner = rand() % (total_tickets + 1); //+1 since initally total could = 0
  //find the winner if there exists a runnable process;
  //if for loop fully iterates, no winner found; start again.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
     if(total_tickets == 0)
       break; //no tickets means no proc to run; break to allow interrupts
     if(p->state != RUNNABLE)
       continue; //skip remaining stmnts in for loop - go to next iteration
     curr_sum_tickets += p->tickets;//found runnable process; add to curr sum
     if(curr_sum_tickets < winner)
       continue;//winner not found, trans contr to for loop for next iter

    //Winner found; continue rest of code:

    // Switch to chosen process.  It is the process's job
    // to release ptable.lock and then reacquire it
    // before jumping back to us.
    c->proc = p; //current CPU's running process is set to p
    switchuvm(p); //switch to user process virtual memory to p
    p->state = RUNNING;
    p->clockticks++; //defined as number of times a process was scheduled
```

```
    //get current total tickets in system
    total_tickets = 0;
    curr_sum_tickets = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
       if(p->state == RUNNABLE)
         total_tickets += p->tickets; //ticket sum only for RUNNABLE (ready) procs.
    //generate winning ticket
    winner = rand() % (total_tickets + 1); //+1 since initally total could = 0
    //find the winner if there exists a runnable process;
    //if for loop fully iterates, no winner found; start again.
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
       if(total_tickets == 0)
         break; //no tickets means no proc to run; break to allow interrupts
       if(p->state != RUNNABLE)
         continue; //skip remaining stmnts in for loop - go to next iteration
       curr_sum_tickets += p->tickets;//found runnable process; add to curr sum
       if(curr_sum_tickets < winner)
         continue;//winner not found, trans contr to for loop for next iter

      //Winner found; continue rest of code:

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p; //current CPU's running process is set to p
      switchuvm(p); //switch to user process virtual memory to p
      p->state = RUNNING;
      p->clockticks++; //defined as number of times a process was scheduled

      swtch(&(c->scheduler), p->context); //context switch; run p
      switchkvm(); //switch back to kernel page table when p returns

      // Process is done running for now.
      // It should have changed its p->state before coming back.

      c->proc = 0; //scheduler running again; no other processes running; set to null
      break; //winner found and ran; start infinite for-loop again to find new winner
    }
    release(&ptable.lock); //allow ptable changes and other CPU to schedule user proc

  }
}
```

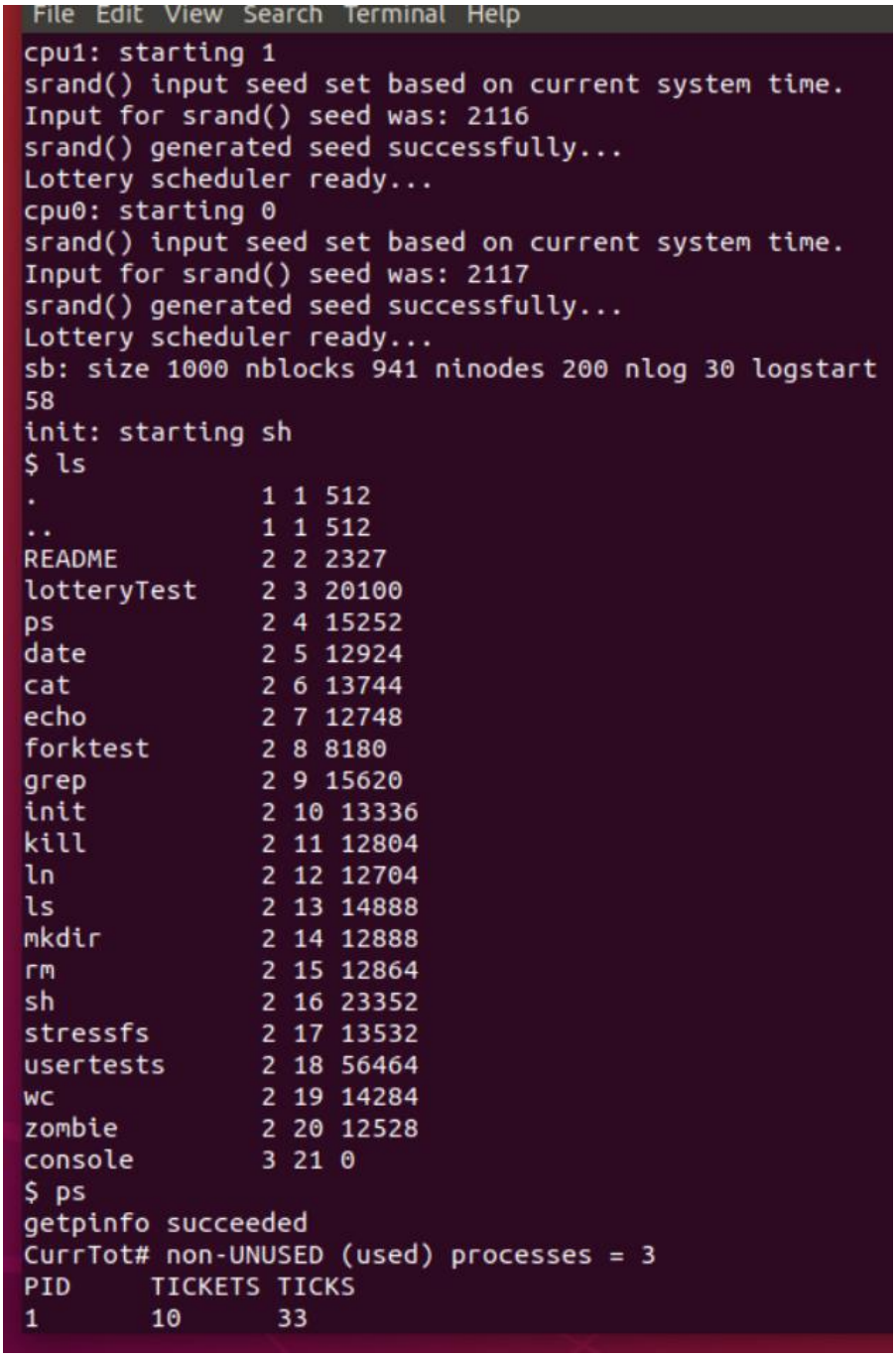
Task 12: project summary of my lottery implementation

For the lottery scheduler, for the most part I used the book's version of the implementation, but I did add my own changes. I used the rand() function to generate lottery tickets, but for initializing using that function, I had to use srand(). For the srand() function, I made a seed based on the current system time, so I used cmostime and the RTC date struct. Then, I summed up all of the fields of the RTC date struct and stored it in an int in order to get the seed. I commented my program very good for the scheduler function, so you should be able to peak through it and easily follow the details how I implemented it and how I was gathering an understanding while doing so.

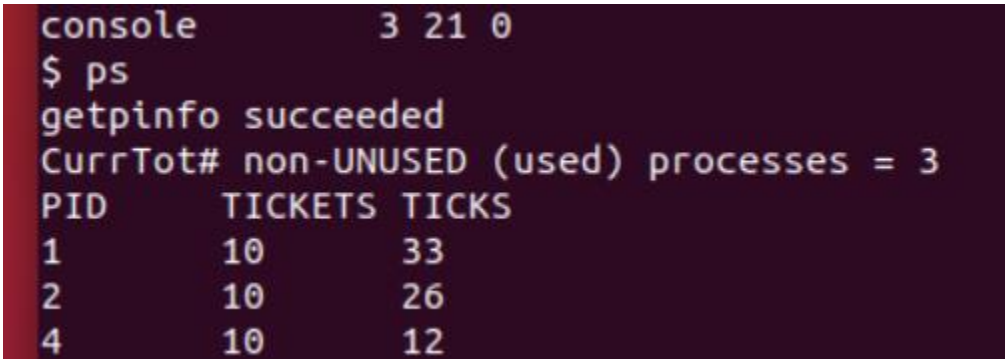
As one final note, I noticed from running all of my tests including the usertests program that for xv6 in particular, at least according to most of the programs I run, the round robin scheduler worked much better than the lottery scheduler – round robin appeared to run much faster, particularly when it came to screen output from prints. Also, when usertests ran, it took a bit longer for most of the tests to run to completion, but nonetheless, they all ran properly and so all tests were performed were a pass.

Task 13: test results with screenshots and explanation

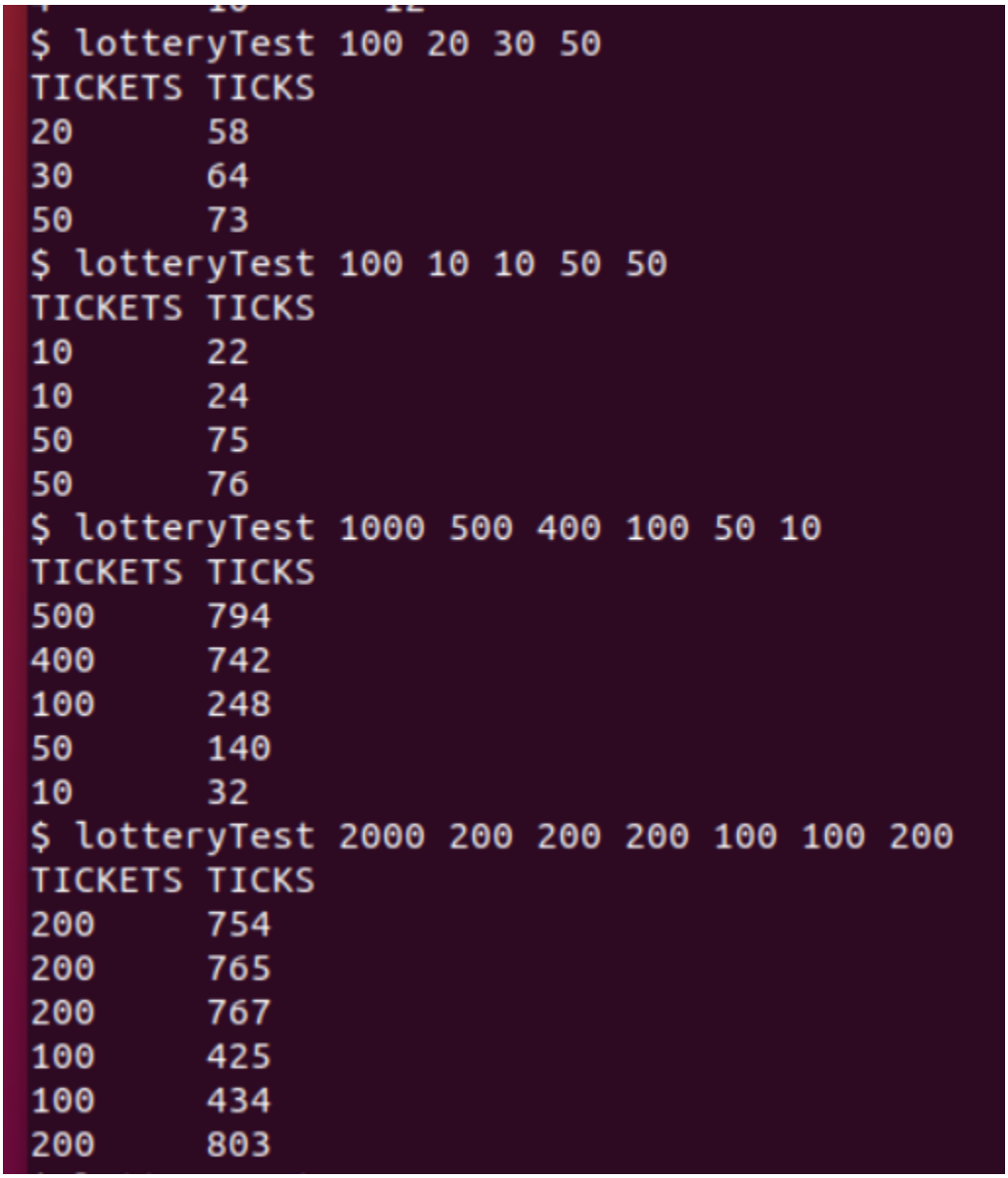
Showing xv6 starting with my lottery scheduler



Showing the ps function is working



Showing lottery tests using values from p4 video and for required tests. Notice: because 2 CPUs are running for xv6, input parameter for number of ticks reflects the number of ticks each scheduler will run, and thus since each CPU runs the scheduler function, we will have double the amount of total ticks ran. Also, notice that the number of tickets is roughly proportional to the total number of ticks ran compared to all other processes.

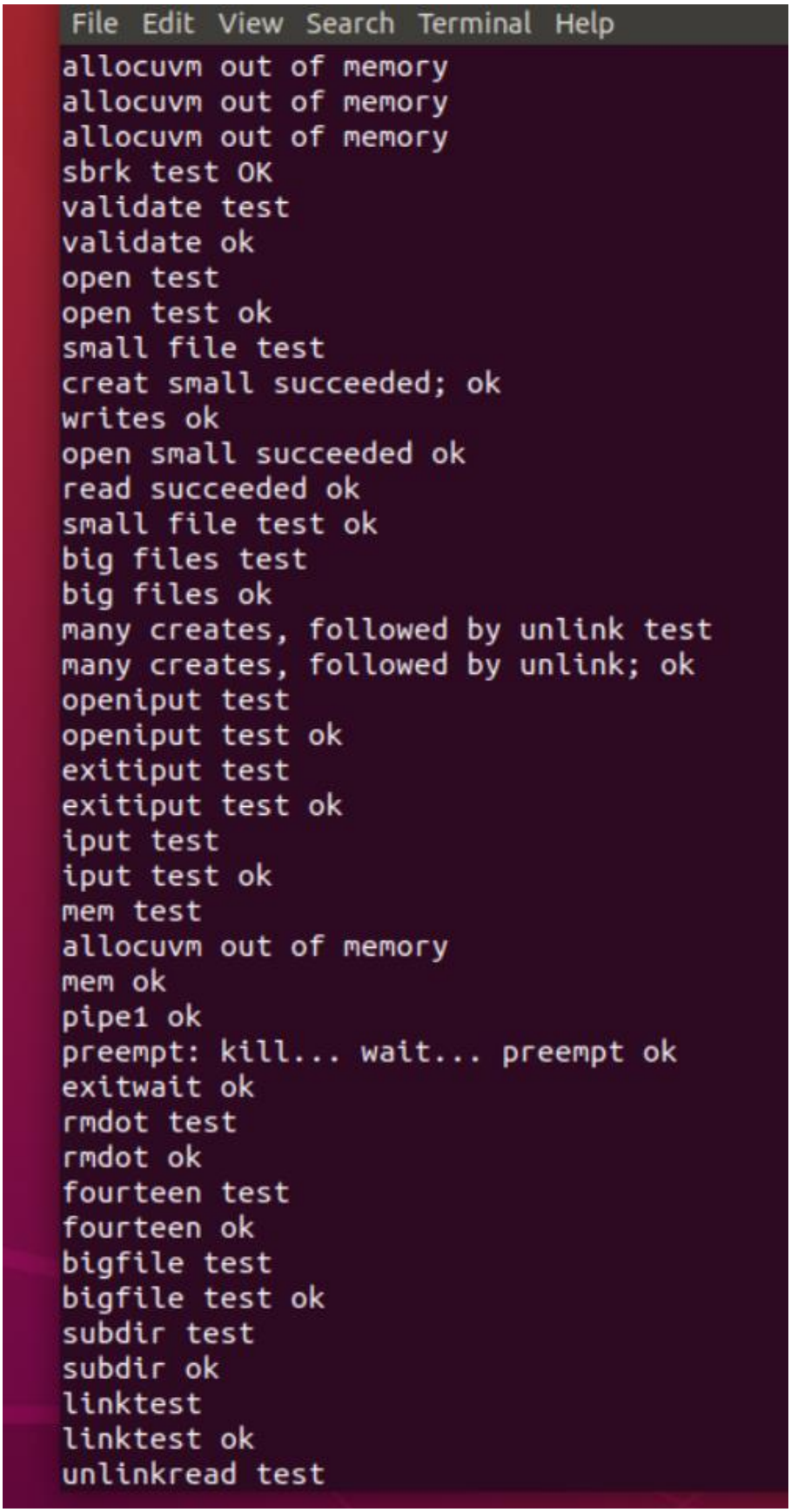


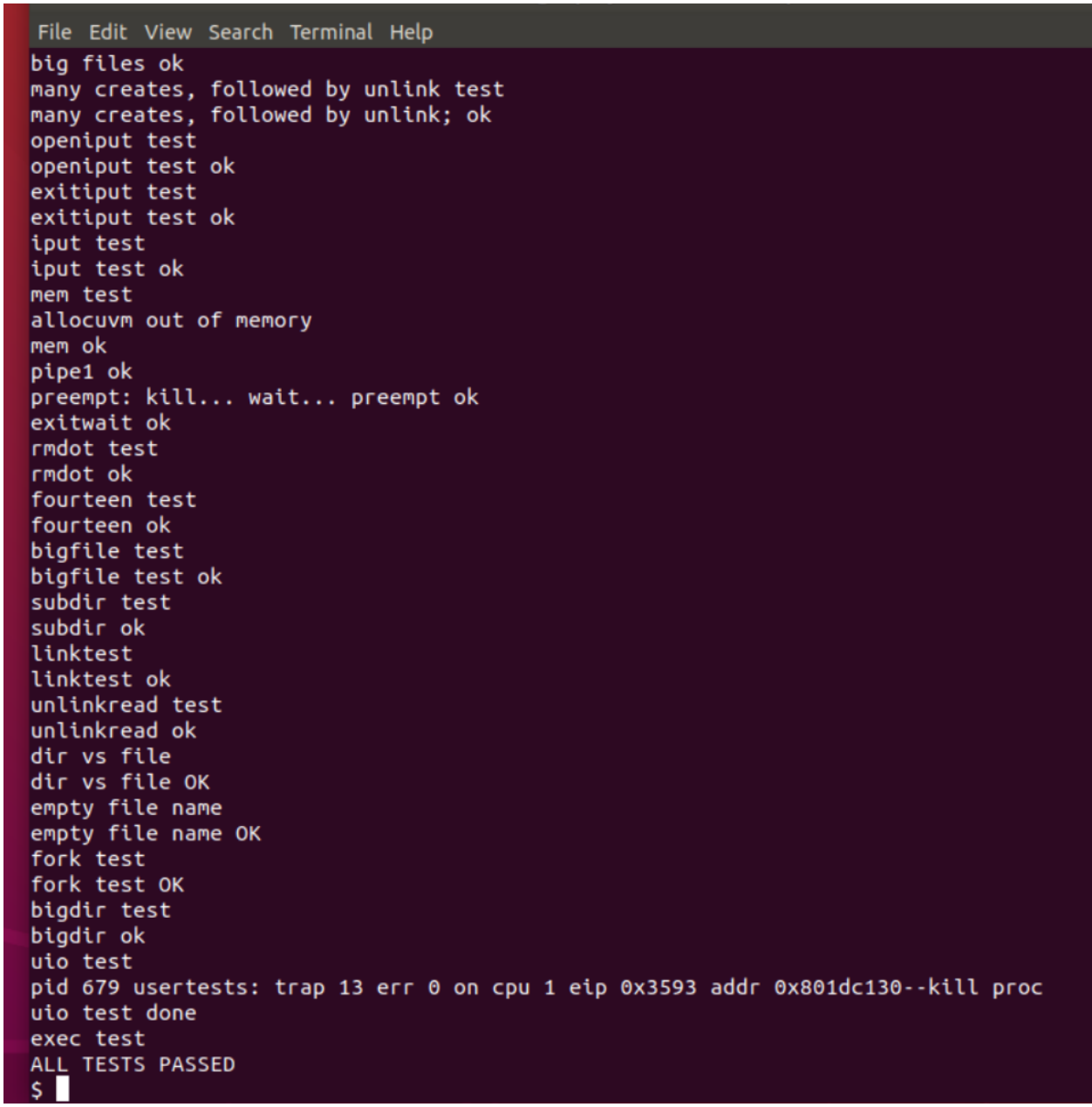
Showing the remaining lottery tests

```
$ lotteryTest 2000 10 20 50 100 200 300
TICKETS TICKS
10      99
20      167
50      381
100     731
200     1197
300     1395
```

Showing the usertests program being ran to test all xv6 functionality, and specifically the scheduler function that each CPU runs

```
$ usertests
usertests starting
arg test passed
createdelete test
createdelete ok
linkunlink test
linkunlink ok
concreate test
concreate ok
fourfiles test
fourfiles ok
sharedfd test
sharedfd ok
bigarg test
bigarg test ok
bigwrite test
bigwrite ok
bigarg test
bigarg test ok
bss test
bss test ok
sbrk test
pid 185 usertests: trap 14 err 5 on cpu 1 eip 0x304a addr 0x80000000--kill proc
pid 186 usertests: trap 14 err 5 on cpu 1 eip 0x304a addr 0x8000c350--kill proc
pid 187 usertests: trap 14 err 5 on cpu 1 eip 0x304a addr 0x800186a0--kill proc
pid 188 usertests: trap 14 err 5 on cpu 1 eip 0x304a addr 0x800249f0--kill proc
pid 189 usertests: trap 14 err 5 on cpu 1 eip 0x304a addr 0x80030d40--kill proc
pid 190 usertests: trap 14 err 5 on cpu 1 eip 0x304a addr 0x8003d090--kill proc
pid 191 usertests: trap 14 err 5 on cpu 1 eip 0x304a addr 0x800493e0--kill proc
pid 192 usertests: trap 14 err 5 on cpu 1 eip 0x304a addr 0x80055730--kill proc
pid 193 usertests: trap 14 err 5 on cpu 1 eip 0x304a addr 0x80061a80--kill proc
pid 194 usertests: trap 14 err 5 on cpu 1 eip 0x304a addr 0x8006ddd0--kill proc
pid 195 usertests: trap 14 err 5 on cpu 1 eip 0x304a addr 0x8007a120--kill proc
```


File Edit View Search Terminal Help
allocuvm out of memory
allocuvm out of memory
allocuvm out of memory
sbrk test OK
validate test
validate ok
open test
open test ok
small file test
creat small succeeded; ok
writes ok
open small succeeded ok
read succeeded ok
small file test ok
big files test
big files ok
many creates, followed by unlink test
many creates, followed by unlink; ok
openiput test
openiput test ok
exitiput test
exitiput test ok
iput test
iput test ok
mem test
allocuvm out of memory
mem ok
pipe1 ok
preempt: kill... wait... preempt ok
exitwait ok
rmdot test
rmdot ok
fourteen test
fourteen ok
bigfile test
bigfile test ok
subdir test
subdir ok
linktest
linktest ok
unlinkread test



Task 14: reflection on what I have learned

For this project, I really was able to understand the mechanism of a kernel, and how it runs the scheduler program in order to schedule all other programs. I also learned a lot about compiling and linking files, and about how make files work. I ran the usertests program that was already built into xv6 to test my program incrementally along with the ps function. I learned how to make wrapper functions and was able to understand how they work better – including making every single one of the system calls I added to xv6 for this project a wrapper function; the main implementation of all of my system calls were in the proc.c file. Initially it was causing lots of issues because I needed to understand better when and where and how to make the declaration of all of my functions from other files visible. I also figured out from the make file which of the .c files were linked with the xv6 main program. Before implementing and understanding the lottery scheduler, I first had to understand how the current round robin scheduler

already implemented worked, including how the kernel gives control from to a user program and transferring control from kernel to user mode.

Project Notes (including from P4 discussion video)

General

SUGGESTION – do the program incrementally:

Complete settickets function, TEST ITS FUNCTIONALITY.

Complete getpinfo, TEST ITS FUNCTIONALITY.

Then implement lottery scheduler algorithm.

GO TO PAGE 140 (CHAPTER 9, FIGURE 9.3) IN THE OSTEP (Operating Systems, Three Easy Pieces) book to see implementation of the lottery ticket CPU scheduling.

```
1  // counter: used to track if we've found the winner yet
2  int counter = 0;
3
4  // winner: use some call to a random number generator to
5  //         get a value, between 0 and the total # of tickets
6  int winner = getrandom(0, totaltickets);
7
8  // current: use this to walk through the list of jobs
9  node_t *current = head;
10 while (current) {
11     counter = counter + current->tickets;
12     if (counter > winner)
13         break; // found the winner
14     current = current->next;
15 }
16 // 'current' is the winner: schedule it...
```

Figure 9.1: **Lottery Scheduling Decision Code**

getrandom() generates a random number between 0 and total tickets. A random function is provided for us for this project.

Here is a demonstration of how the lottery scheduling works:

P1->tickets = 10

P2->tickets = 20

P3>tickets = 70

Total_tickets = 100

Winner = rand() % (total_tickets);

Winner < 10, P1 is the winner

10 <= Winter < 20, P2 is the winner

20 <= Winter < 100, P3 is the winner

(proc.h) and (proc.c) file:

```
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};

enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
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;     // swtch() 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)
};

// Process memory is laid out contiguously, low addresses first:
//   text
```

Above is in the proc.h file

It contains the proc (process) struct, which is essentially the process control block (PCB)

For process state enumerator: Sleeping == wait, Runnable == ready, running == running.

context struct saves the necessary registers so that a context switch can occur; saved registers will be reloaded into the proper registers when the process gets assigned to run on a CPU again:

```
File  Edit  View  Search  Terminal  Help
// x86 convention is that the caller has saved them.
// Contexts are stored at the bottom of the stack they
// describe; the stack pointer is the address of the context.
// The layout of the context matches the layout of the stack in swtch.S
// at the "Switch stacks" comment. Switch doesn't save eip explicitly,
// but it is on the stack and allocproc() manipulates it.
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};

enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
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
```

The following lines need to be added to the proc struct:

```
  uint ebp;
  uint eip;
};

enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
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;     // swtch() 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 ticks;
  int tickets;
};

// Process memory is laid out contiguously, low addresses first:
//   text
//   original data and bss
-- INSERT --
```

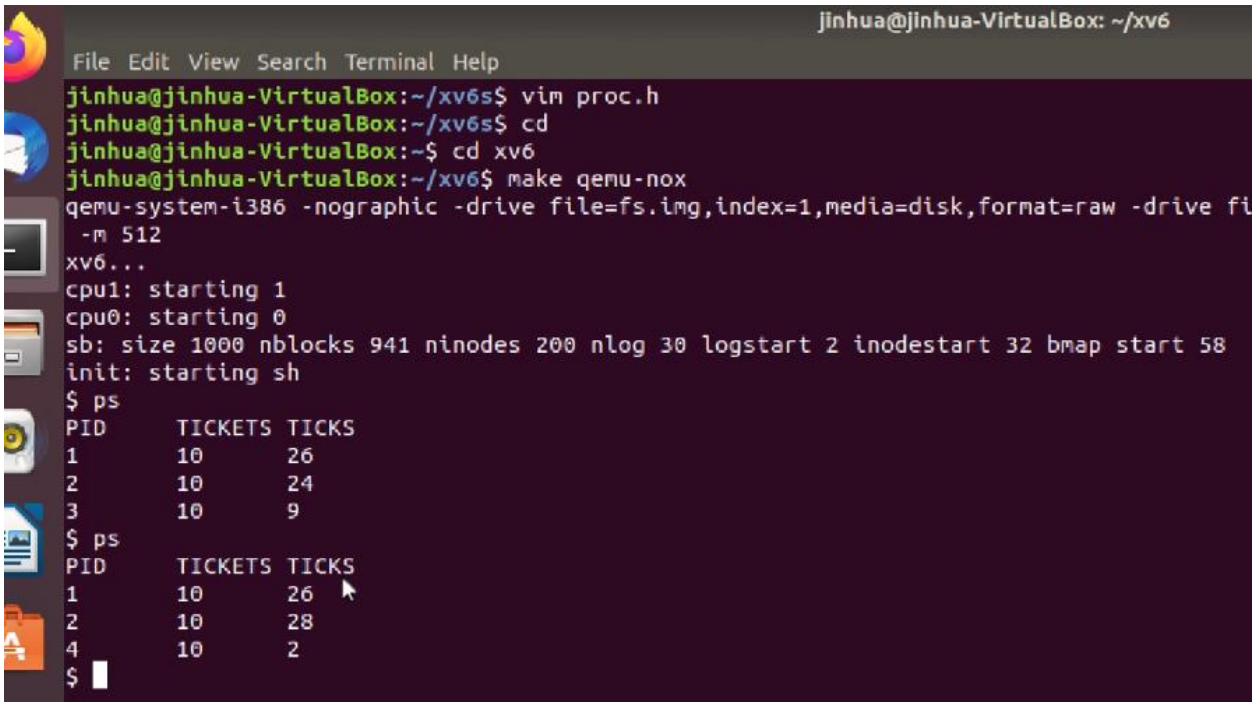
Ticks = number of times a process was scheduled to run on the CPU; each tick (time slice) is about 10ms. Tickets= num tickets assigned to a process.

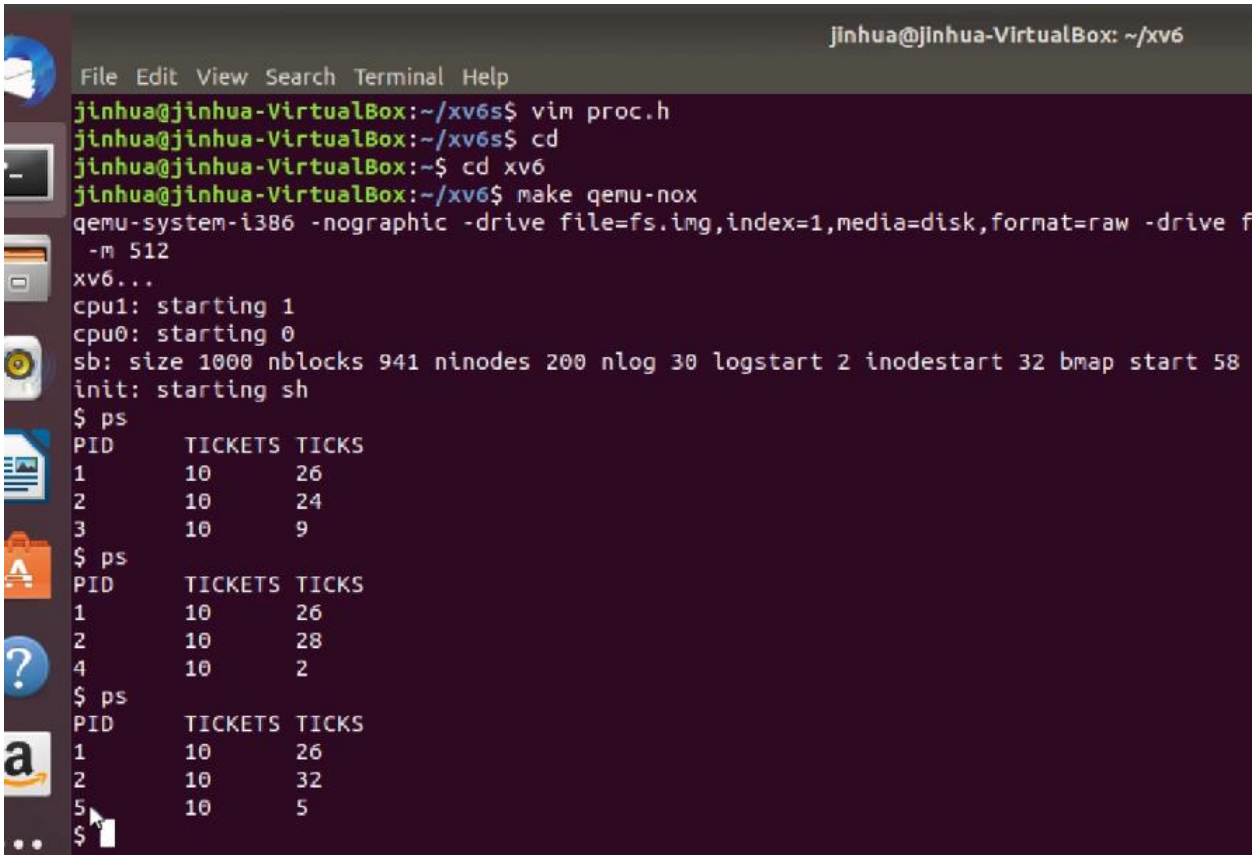
So for the settickets system call, you just need to update the two added variables in the struct.

USING THE PS FUNCTION (ps.c):

```c
#include "types.h"
#include "mmu.h"
#include "param.h"
#include "proc.h"
#include "user.h"
#include "pstat.h"

int main(int argc, char *argv[])
{
    struct pstat info = {};

    getpinfo(&info);
    printf(1, "PID\tTICKETS\tTICKS\n");
    for (int i = 0; i < info.num_processes; ++i) {
      printf(1, "%d\t%d\t%d\n", info.pid[i], info.tickets[i], info.ticks[i]);
    }
    exit();
}
```

ps() function will call the getpinfo() system call that we must write.





Using the above screenshots for reference, the below two processes will always be running in XV6:

PID 1 is the userinit() process; once it stops, it doesn't really run anymore (although it is not killed) as it is the initial user process and the parent process for the xv6 OS, so ticks (times it was schedule to CPU) will remain mostly static.

PID 2 is the share() process; this process will continue to be ran throughout the life of XV6 running.

Upon first call to ps when you first boot XV6, PID 3 is the ps() function itself; thus when you call ps() again immediately after the first call, you will notice processID 3 is no longer in the system as it has terminated, and now PID 4 is the new call to ps(); etc...

yield() system call

Need wrapper function for system call: int yield() this is because we simply need to call the already existent xv6 yield() implementation that is a kernel function which thusly cannot be accessed via user mode; hence, we need this simple wrapper system call so that we can switch from user mode to kernel mode and access the kernel level yield() function.

Given scheduler code is in the proc.c and proc.h files; it is round robin; thus we need to simply change it from round robin to lottery scheduling.

(main.c)

File Edit View Search Terminal Help

```
extern char end[]; // first address after kernel loaded from ELF file

// Bootstrap processor starts running C code here.
// Allocate a real stack and switch to it, first
// doing some setup required for memory allocator to work.
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // detect other processors
  lapicinit();     // interrupt controller
  seginit();       // segment descriptors
  picinit();       // disable pic
  ioapicinit();    // another interrupt controller
  consoleinit();   // console hardware
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  ideinit();       // disk
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  mpmain();        // finish this processor's setup
}
```

Notice: when we first start XV6, there are a lot of initializations (page tables, etc.).

startothers() function in main.c file for XV6 starts the other processers (if present/available; XV6 uses 2 by default); the lines above that function call are all ran on a single processor.

In main.c for XV6, the first user process ran is called userinit() (in proc.c).

So, first we run OS in kernel mode, then first user process called is userinit().

mpmain() in main.c of XV6 will call scheduler() so that the initial processor can start running processes.

```
}

// Common CPU setup code.
static void
mpmain(void)
{
  cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
  idtinit();       // load idt register
  xchg(&(mycpu()->started), 1); // tell startothers() we're up
  scheduler();     // start running processes
}
```

The startothers() function will also end up leading to a call to the scheduler() function so that all subsequent CPUs that start running with the initial CPU will also run the scheduler() function to run processes.

```
// Common CPU setup code.
static void
mpmain(void)
{
  cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
  idtinit();       // load idt register
  xchg(&(mycpu()->started), 1); // tell startothers() we're up
  scheduler();     // start running processes
}

pde_t entrypgdir[];  // For entry.S

// Start the non-boot (AP) processors.
static void
startothers(void)
{
  extern uchar _binary_entryother_start[], _binary_entryother_size[];
  uchar *code;
  struct cpu *c;
  char *stack;

  // Write entry code to unused memory at 0x7000.
  // The linker has placed the image of entryother.S in
  // _binary_entryother_start.
  code = P2V(0x7000);
  memmove(code, _binary_entryother_start, (uint)_binary_entryother_size);
```

```
  char *stack;

  // Write entry code to unused memory at 0x7000.
  // The linker has placed the image of entryother.S in
  // _binary_entryother_start.
  code = P2V(0x7000);
  memmove(code, _binary_entryother_start, (uint)_binary_entryother_size);

  for(c = cpus; c < cpus+ncpu; c++){
    if(c == mycpu())  // We've started already.
      continue;

    // Tell entryother.S what stack to use, where to enter, and what
    // pgdir to use. We cannot use kpgdir yet, because the AP processor
    // is running in low  memory, so we use entrypgdir for the APs too.
    stack = kalloc();
    *(void**)(code-4) = stack + KSTACKSIZE;
    *(void(**)(void))(code-8) = mpenter;
    *(int**)(code-12) = (void *) V2P(entrypgdir);

    lapicstartap(c->apicid, V2P(code));

    // wait for cpu to finish mpmain()
    while(c->started == 0)
      ;
  }
}
```

mpenter() will call mpmain():

```
// Other CPUs jump here from entryother.S.
static void
mpenter(void)
{
  switchkvm();
  seginit();
  lapicinit();
  mpmain();
}
```

And mpmain() will call scheduler()

So, both CPUs in XV6 will end up calling the scheduler() function to run user processes; after calling the userinit() function, they both will get to the scheduler.

Our job is to change the scheduler() function in the proc.c file from a round robin scheme to lottery.

In scheduler function from (proc.c) file:

Original scheduler function with some additional notes I added:

File Edit View Search Terminal Help

```
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run
//  - swtch to start running that process
//  - eventually that process transfers control
//      via swtch back to the scheduler.
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  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)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p; //current CPU's running process is set to p
      switchuvm(p); //switch to user process virtual memory to p
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context); //switch to user page table; run p
      switchkvm(); //switch back to kernel page table when p returns

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

  }
}
                                                              356,18
```

File Edit View Search Terminal Help

```
//  - swtch to start running that process
//  - eventually that process transfers control
//      via swtch back to the scheduler.
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  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)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      //cprintf("About to run: %s, [pid = %d]\n", p->name, p->pid);
```

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  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)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      //cprintf("About to run: %s, [pid = %d]\n", p->name, p->pid);

      swtch(&(c->scheduler), p->context);
      switchkvm();
```

Scheduler is an infinite loop so that any processes that will enter the system can then start to be ran by the CPU via the scheduler() function. So each CPU will stay in this scheduler loop (continue running it) forever until the system is shutdown/rebooted.

mycpu() function returns which CPU is accessing the scheduler function; returns an int (0 = cpu0, 1 = cpu1).

switchuvm() function switches the page table from kernel page table to the current user process page table – uvm perhaps stands for “user virtual memory”.

Then, process state changed from runnable to running.

Then, swtch() is called – it is written in assembly language directly, and is in the swtch.S file; which performs a context switch, which switches from the current kernel process that is running (c→scheduler()) to the chosen user process (p→context).

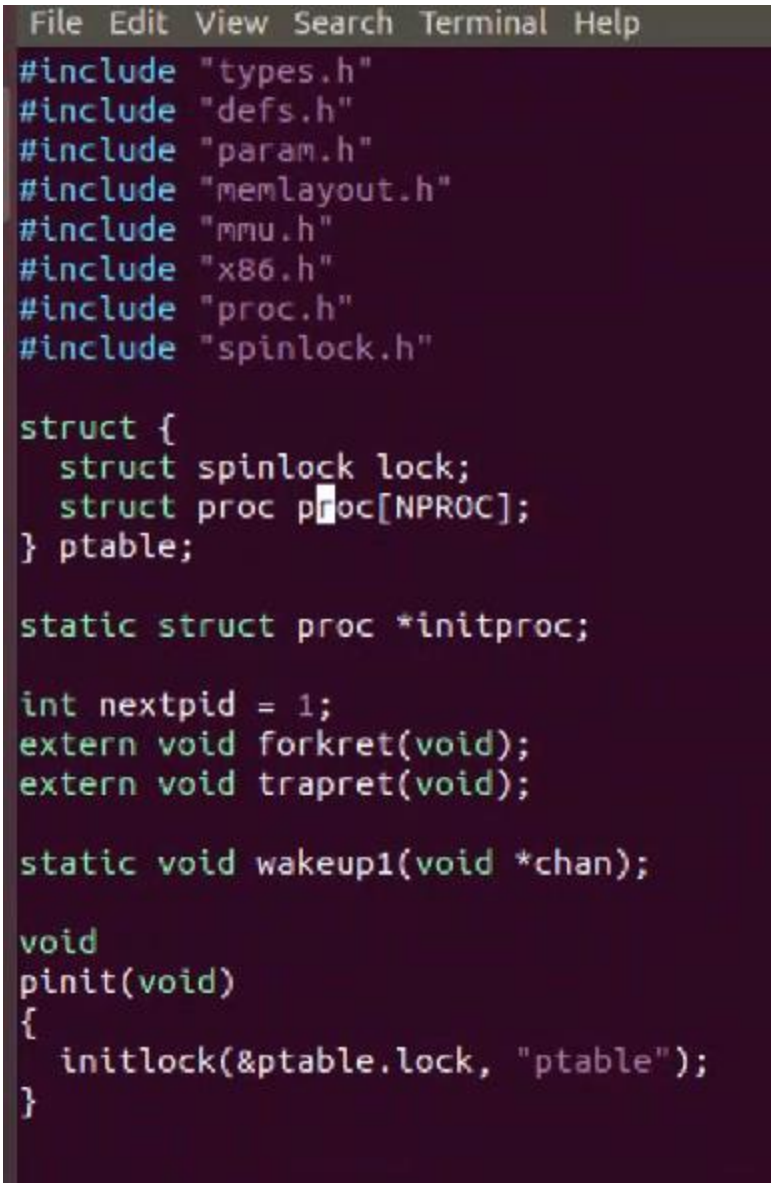
```
#
#   void swtch(struct context **old, struct context *new);
#
# Save the current registers on the stack, creating
# a struct context, and save its address in *old.
# Switch stacks to new and pop previously-saved registers.

.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-saved registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
"swtch.S" 29L, 542C
```

In xv6, we only need to save 5 registers, which are pushed onto the stack of the current process. Then the 5 registers pushed onto the stack of the process to be switched to are popped into the registers; thus we now have a context switch from one process to another.

Then when the user process returns (thus, after the switch statement returns), the swtichkvm() is called → switch from user page table back to the kernel page table (aka: switch from user virtual memory (uvm) to kernel virtual memory (kvm)).

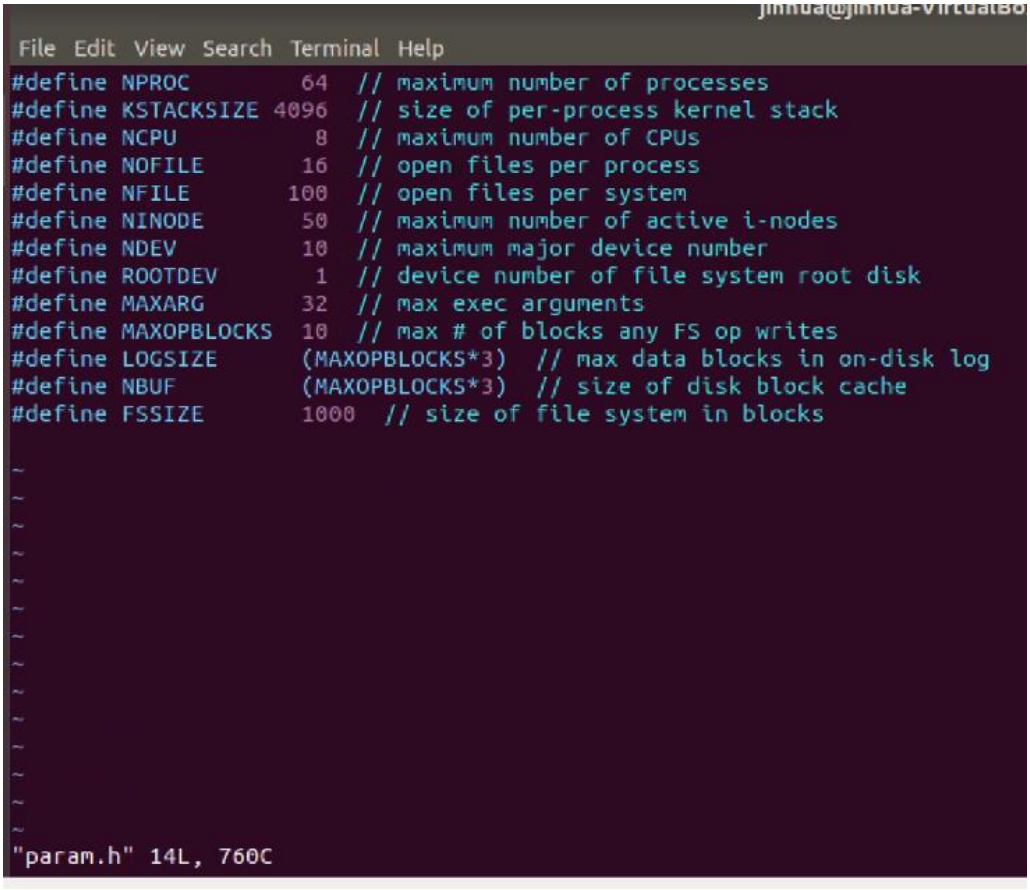
Then continue running (loop again) the scheduler() function's infinite loop to select and run another process, and reinitializing c→proc = 0.

ptable = process table. Here is part of the definition:



```
File  Edit  View  Search  Terminal  Help
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "x86.h"
#include "proc.h"
#include "spinlock.h"

struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;

static struct proc *initproc;

int nextpid = 1;
extern void forkret(void);
extern void trapret(void);

static void wakeup1(void *chan);

void
pinit(void)
{
  initlock(&ptable.lock, "ptable");
}
```

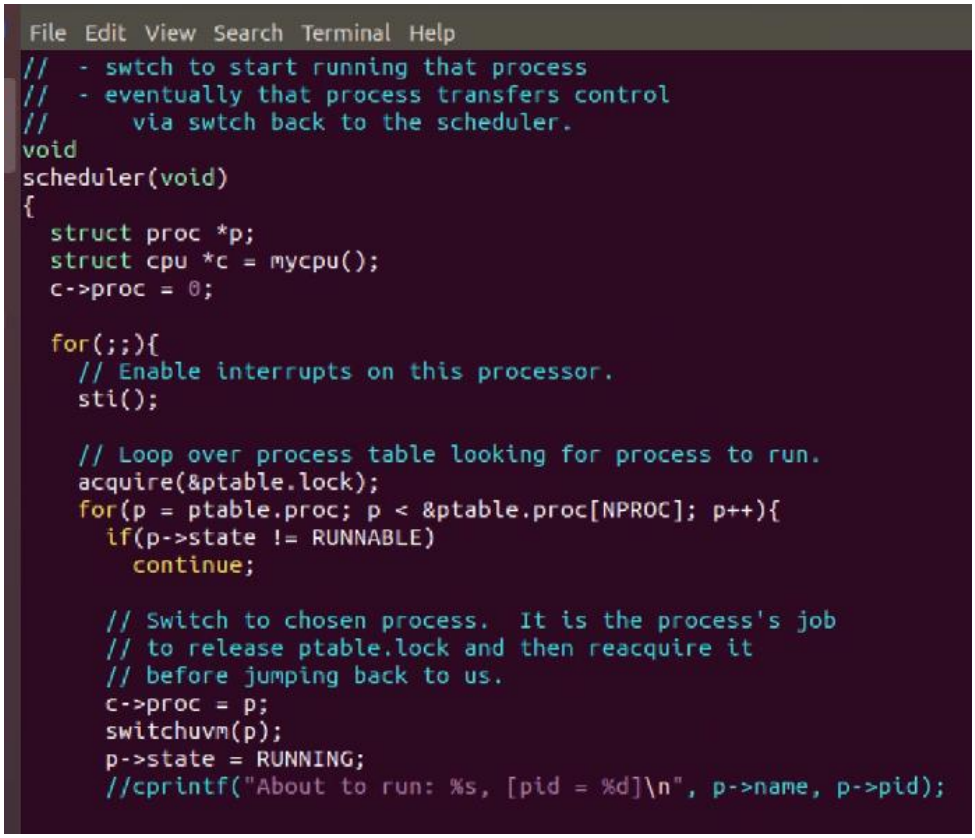
NPROC == 64; in XV6, you can only create a maximum of 64 processes. It is found in the param.h header file:



ptable struct has a lock variable that must be acquired before updating the fixed proc[NPROC] struct array.

In the proc[NPROC] struct array, it has all information about each process.

Our job is to change the scheduler() function in the proc.c file from a round robin scheme to lottery.



```
File  Edit  View  Search  Terminal  Help
//   - swtch to start running that process
//   - eventually that process transfers control
//       via swtch back to the scheduler.
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  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)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      //cprintf("About to run: %s, [pid = %d]\n", p->name, p->pid);
```

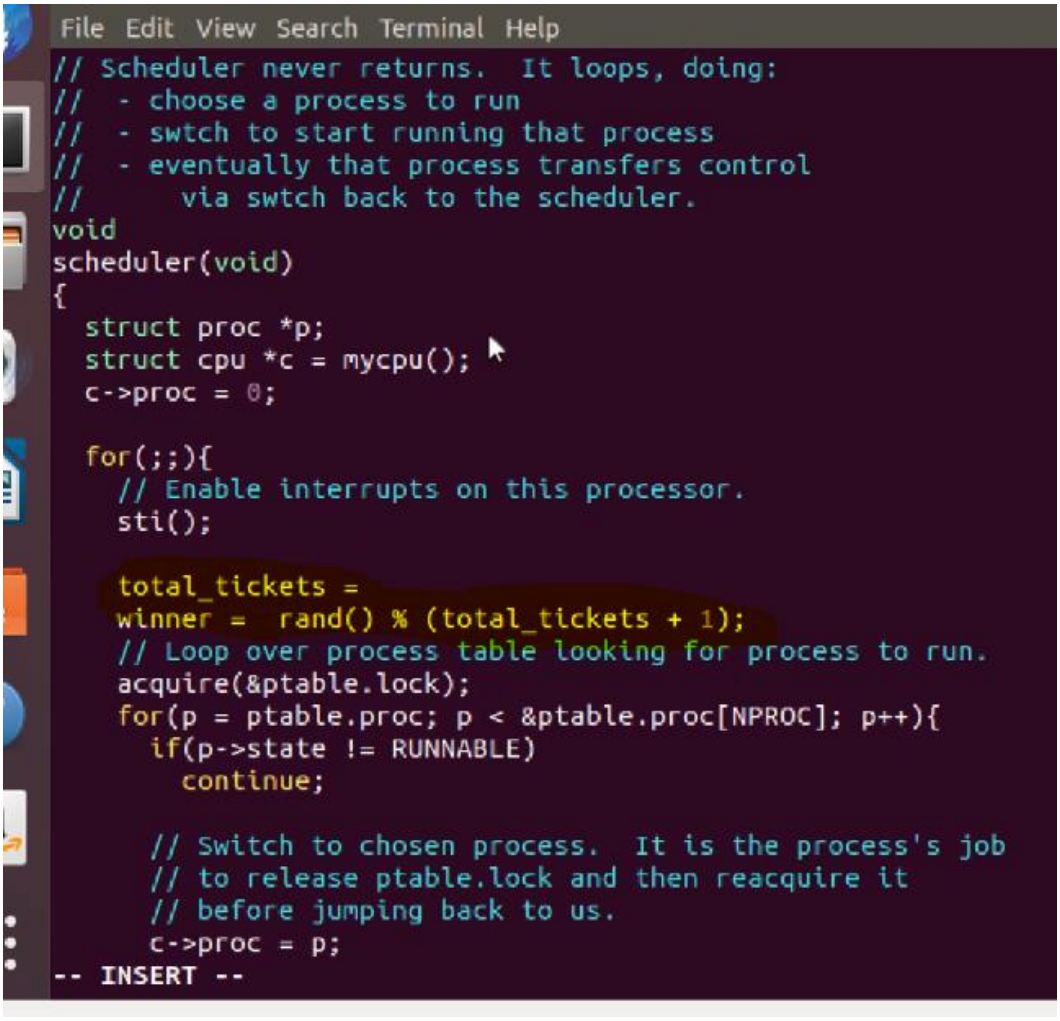
Notice above: we initialize c→ proc = 0 to indicate there are no user processes running.

Primary objective is to change the small code under "//Loop over process table looking for process to run", which is a round robin scheme, and change it to lottery scheduling. Everything after that (the context switching to the selected process, etc.) can stay as it is.

Make sure to leave the enable interrupts line in the code.

First, **outside** of the *second* for loop, iterate through entire process table (ptable) to count total number of tickets for processes in the RUNNABLE state. The outer for(; ;) should not have code written to it as it is what will infinitely loop to process user processes or wait for a user processes to process.

Second, below the line of code from the above first step, add the winner as a function of a random number generator to generate a number between 0 and total tickets (which was just acquired from the first step) by using a modulus operator. It is good to add +1 to the random number function, since sometimes the total tickets in the system can initially be or later be 0 and we don't want a divide by 0 error; so we can avoid that issue by adding 1 (since rand() % 0 is throws a divide by 0 software voluntary interrupt error.).



Then, following the logic how finding the winner, you implement that logic inside of the second for loop: continue to count all runnable processes (sum them) until you find a value where number of tickets a process has is >= winner value, as shown in the example:

```
P1->tickets = 10

P2->tickets = 20

P3>tickets = 70

Total_tickets = 100

Winner = rand() % (total_tickets);

Winner < 10, P1 is the winner

10 <= Winter < 20, P2 is the winner

20 <= Winter < 100, P3 is the winner
```

You can use cprintf statement to see which process gets scheduled and about to run, including each tick (cpu cycle) as it continues to be ran:

```
// before jumping back to us.
c->proc = p;
switchuvm(p);
p->state = RUNNING;
//cprintf("About to run: %s, [pid = %d]\n", p->name, p->pid);

swtch(&(c->scheduler), p->context);
```

```
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)
      continue;

    // Switch to chosen process.  It is the process's job
    // to release ptable.lock and then reacquire it
    // before jumping back to us.
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;
    cprintf("About to run: %s, [pid = %d]\n", p->name, p->pid);

    swtch(&(c->scheduler), p->context);
    switchkvm();

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

Example of output from the print command:

```
About to run: initcode, [pid = 1]
About to run: initcode, [pid = 1]
About to run: init, [pid = 1]
About to run: init, [pid = 1]
About to run: init, [pid = 1]
About to run: init, [pid = 1]
iAbout to run: init, [pid = 1]
nit: startiAbout to run: init, [pid = 1]
nAbout to run: init, [pid = 1]
g sh
About to run: init, [pid = 1]
About to run: init, [pid = 1]
About to run: init, [pid = 1]
About to run: init, [pid = 2]
About to run: init, [pid = 2]
About to run: init, [pid = 2]
```

Note, you can compile and run XV6 in the same terminal via this command (to quit xv6, press control+a x, that is press Ctrl+a together and then release. Then press the ‘x’ key):

```
jinhua@jinhua-VirtualBox:~/xv6s$ vim proc.c
jinhua@jinhua-VirtualBox:~/xv6s$ make qemu-nox
gcc -fno-pic -static -fno-builtin -fno-strict-aliasi
-fno-pie -no-pie   -c -o proc.o proc.c
```

fork() system call function (proc.c)

fork() system call creates a new process, and copies everything from its parent.

Allocate a new process, called np.

we copy the parent's process state to the new process (the child process).

```
File  Edit  View  Search  Terminal  Help
// Sets up stack to return as if from system call.
// Caller must set state of returned proc to RUNNABLE.
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;
```

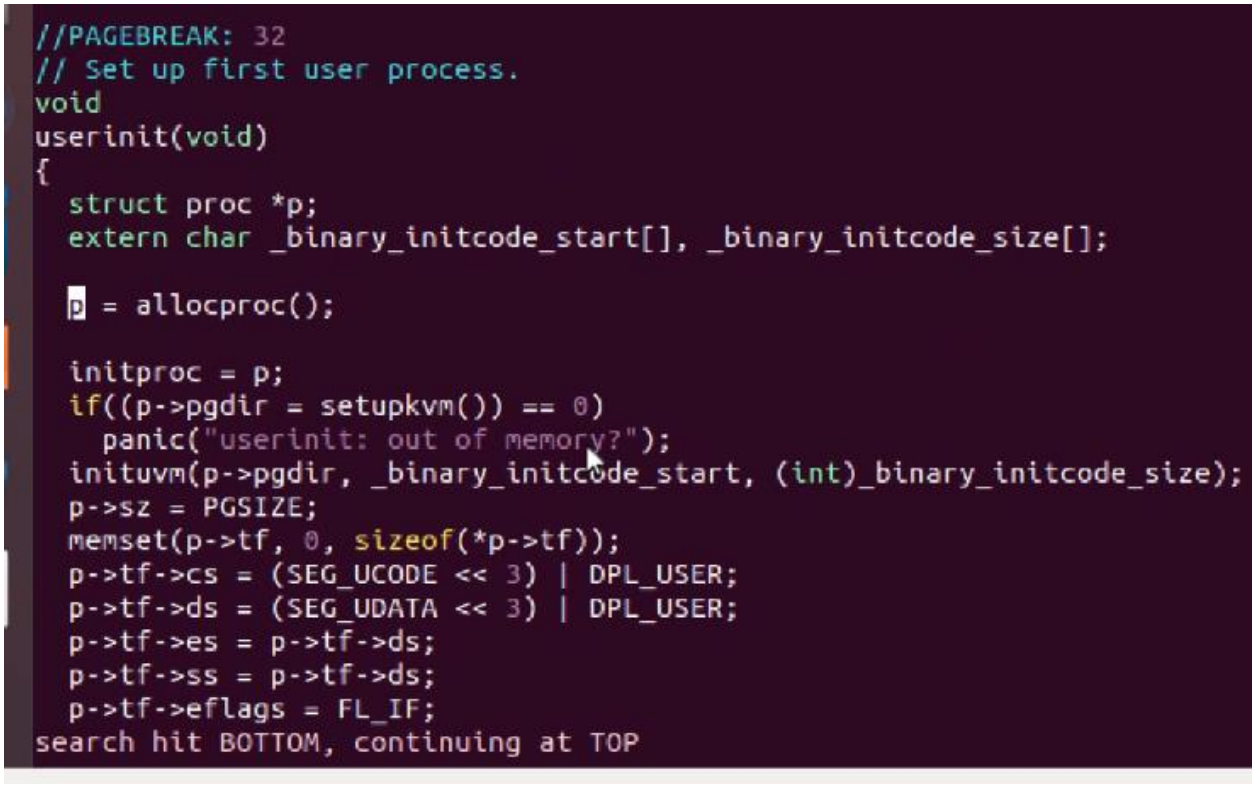
```
File  Edit  View  Search  Terminal  Help
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);

  safestrcpy(np->name, curproc->name, sizeof(curproc->name));

  pid = np->pid;

  acquire(&ptable.lock);

  np->state = RUNNABLE;

  release(&ptable.lock);

  return pid;
}
```

So remember for this project, we need to update 2 things: number of a tickets for a new ticket, where default is 10 tickets – but for a child process via fork, child tickets should = parents tickets; AND we need to update the new process numticks = 0:

```
  safestrcpy(np->name, curproc->name, sizeof(curproc->name));

  pid = np->pid;

  acquire(&ptable.lock);

  np->state = RUNNABLE;
  np->tickets = curproc->tickets;
  np->ticks = 0;

  release(&ptable.lock);

  return pid;
}

-- INSERT --
```

userinit() function (proc.c)

Need to make similar changes made to fork() function in the userinit() function.



```
//PAGEBREAK: 32
// Set up first user process.
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();

  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
search hit BOTTOM, continuing at TOP
```

```
  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  // this assignment to p->state lets other cores
  // run this process. the acquire forces the above
  // writes to be visible, and the lock is also needed
  // because the assignment might not be atomic.
  acquire(&ptable.lock);

  p->state = RUNNABLE;

  release(&ptable.lock);
}
```

Here, we need to assign number of tickets to the newly created process; but this time, set it to the default = 10, and set numticks = 0. Do it in the same location as with the fork() function (circled in red):

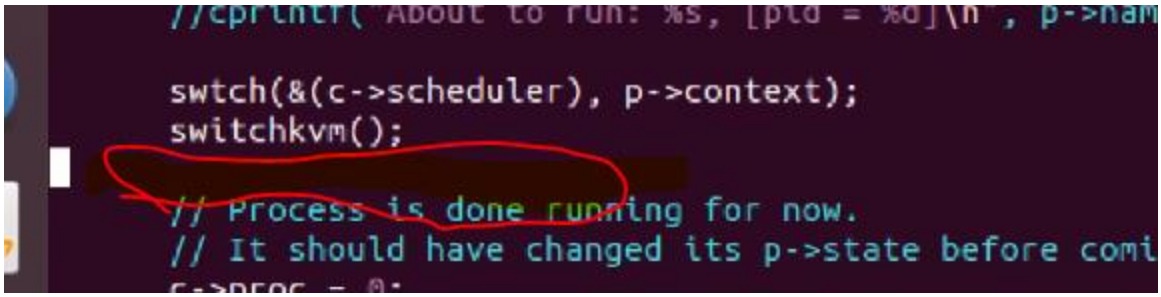
```
  // because the assignment might not be atomic.
  acquire(&ptable.lock);

  p->state = RUNNABLE;

  release(&ptable.lock);
}
```

getpinfo() function (sysproc.c)

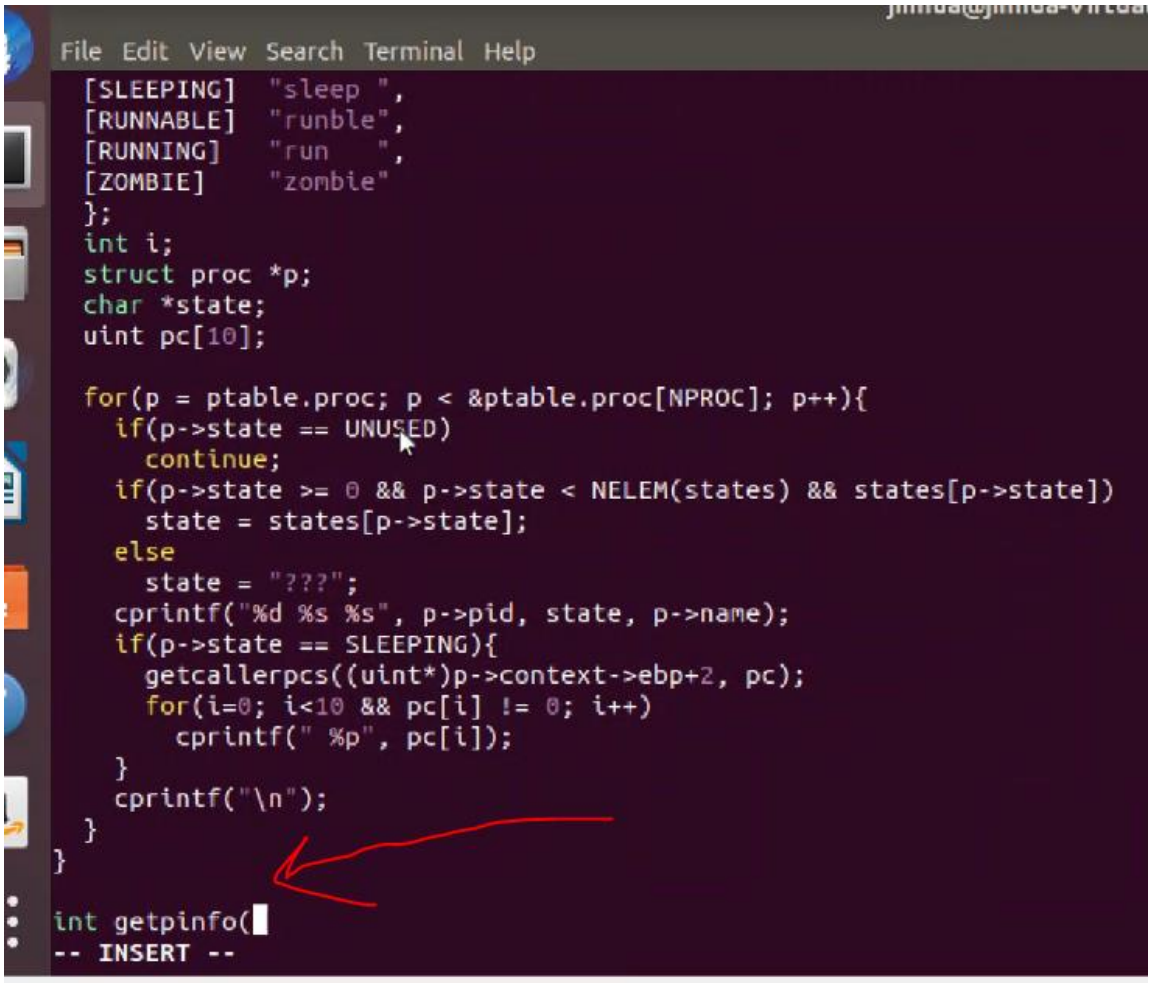
You need to add an entry for set tickets and yield and getpinfo functions in sysproc.c

getpinfo() needs to iterate through the process table to get information (state, id, numtickets, num ticks) about all processes that have a state that is **not** UNUSED. (so, runnable or running; get the information).

Call the getpinfo function at end of scheduler function:

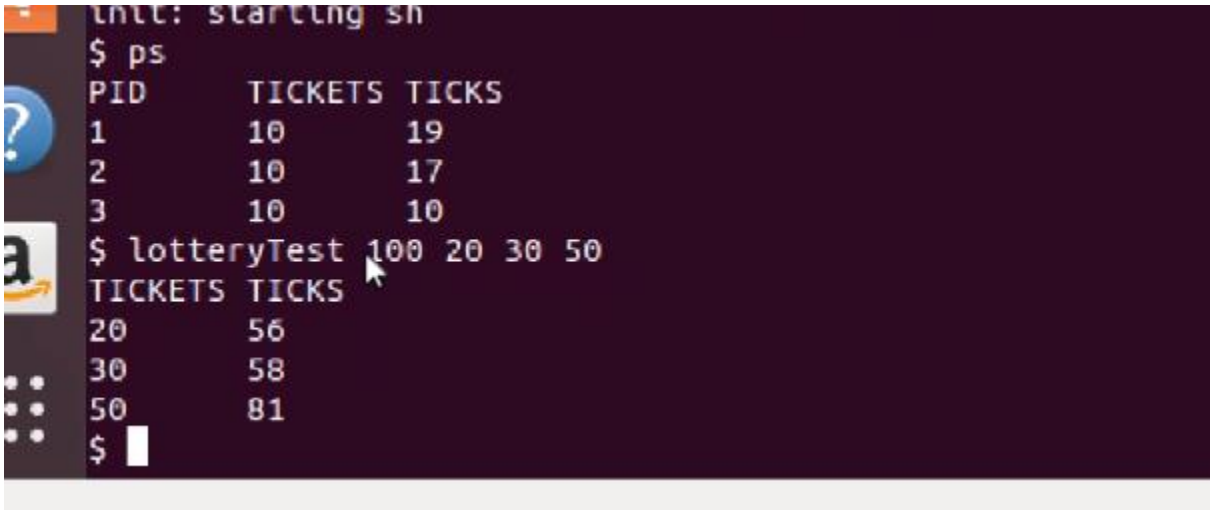


Add the implementation here:



Follow part of the logic from the above function which checks if a process is UNUSED; if not UNUSED, then get info.

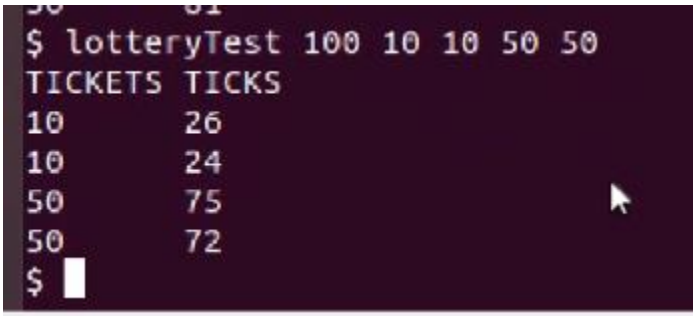
lotteryTest.c



Notice, number of ticks each process runs for is roughly proportional to the number of tickets each process has. Also, even though first parameter is number of ticks = 100, you notice that total number of ticks summed up for all processes is about 200 = double the input. This is because xv6 runs two CPUs.

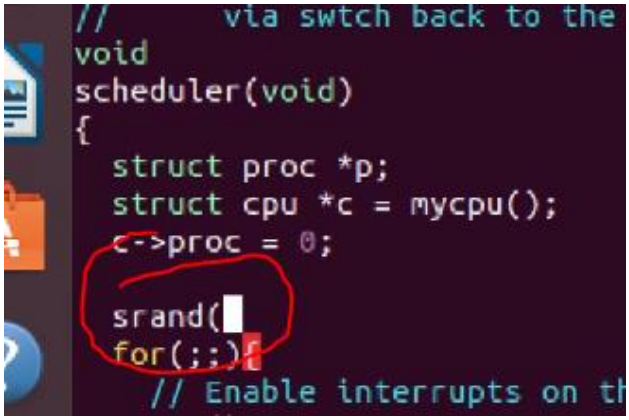
So if you input 200 for number of ticks to run a set of processes with set ticket values, the total ticks will be about 400 since **each** CPU runs about 200 ticks.

Proportionality with respect to number of tickets and total ticks will become more accurate if you input longer number of ticks and more the more processes you run in the system (need at least 4 processes):



random.h – concerning the scheduler() function

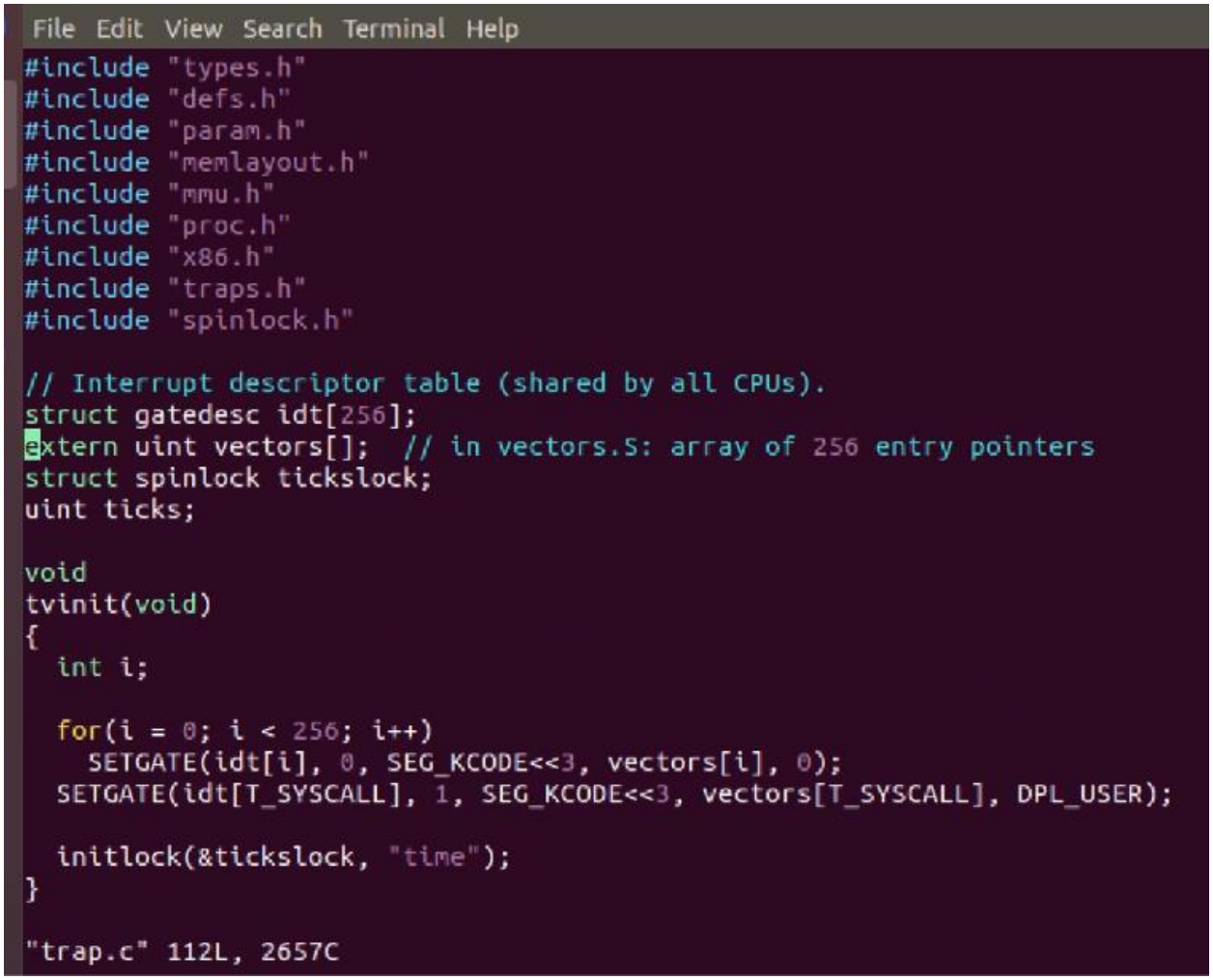
If we want more controlled but more random pseudo random, we may want to call srand() function which can take a parameter that determines the random value generated; the passed in parameter can be based on current time of day or both date and time (use cmos time):



Otherwise, you can pass in a value to purposely generate the same pseudo-random values for more controlled initial testing, or just call the rand() function which uses current state of the system to generate pseudo random values.

(trap.c)

Question: when we context switch to a user process and change mode to user mode and run the user process, how do we switch back to the kernel again (on yield, interrupt, system call, etc.)???



There are all the different kinds of interrupts in trap.c

For example, software interrupts, such as system calls.

But normally when we switch back to kernel from user mode (and a user process) there is a yield due to a timer expiring and the process has not finished yet, so that kernel can run kernel code again (namely, the scheduler/schedule function) and context switch to another process to be ran.

```
    }
    // In user space, assume process misbehaved.
    cprintf("pid %d %s: trap %d err %d on cpu %d "
            "eip 0x%x addr 0x%x--kill proc\n",
            myproc()->pid, myproc()->name, tf->trapno,
            tf->err, cpuid(), tf->eip, rcr2());
    myproc()->killed = 1;
  }

  // Force process exit if it has been killed and is in user space.
  // (If it is still executing in the kernel, let it keep running
  // until it gets to the regular system call return.)
  if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
    exit();

  // Force process to give up CPU on clock tick.
  // If interrupts were on while locks held, would need to check nlock
  if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();

  // Check if the process has been killed since we yielded
  if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
    exit();
}
:/yi
```

Now lets see the yield() function, which is in (proc.c)

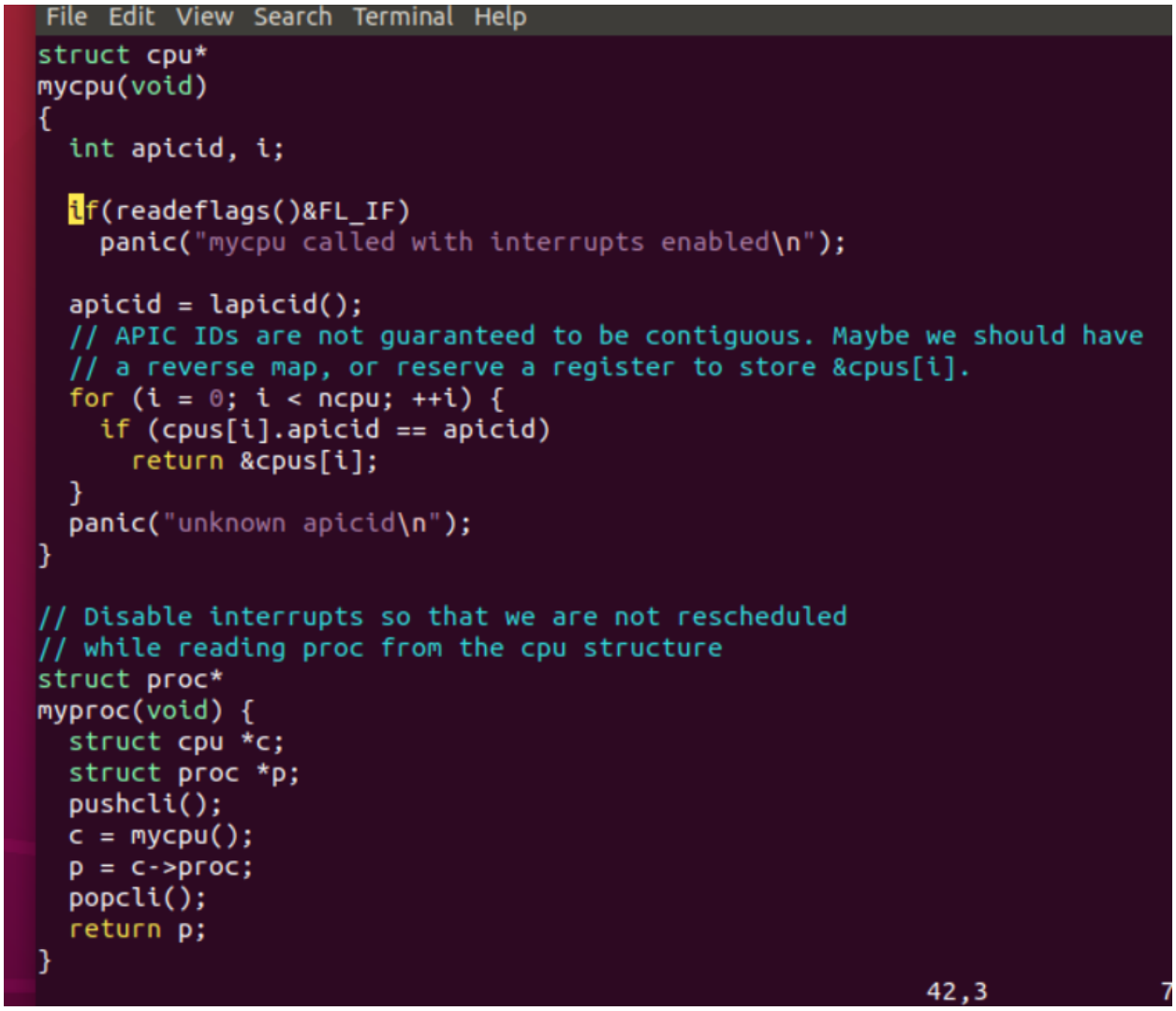
```
}

// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

Process will be changed from running state to runnable (ready).

Also from the sched() function (also in proc.c) it will end up calling swtch() function to do a context switch from current running user process (or whatever type of process calls yield) to the cpu scheduler (a kernel-level function; thus this is a context switch from user mode to the kernel). And then as noted in other places in this document, the kernel scheduler() function will perform a context switch to another user process:

```
// kernel thread, not this CPU. It should
// be proc->intena and proc->ncli, but that would
// break in the few places where a lock is held but
// there's no process.
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}

// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
```

mycpu() and myproc() functions (proc.c):



The mycpu() function gets the current information stored in a CPU registers for the currently running process and returns a cpu data type reference where the data/information is stored in memory. Then, myproc() function uses mycpu() function to acquire that reference, and then acquire specifically the proc struct (which is in the reference to the cpu struct), which is the PCB = Process Control Block data, and return a reference to the proc struct that is in the cpu struct. So both functions simply return references to global variables declared in the proc.c program, namely:

the global cpu struct variable (in proc.h):

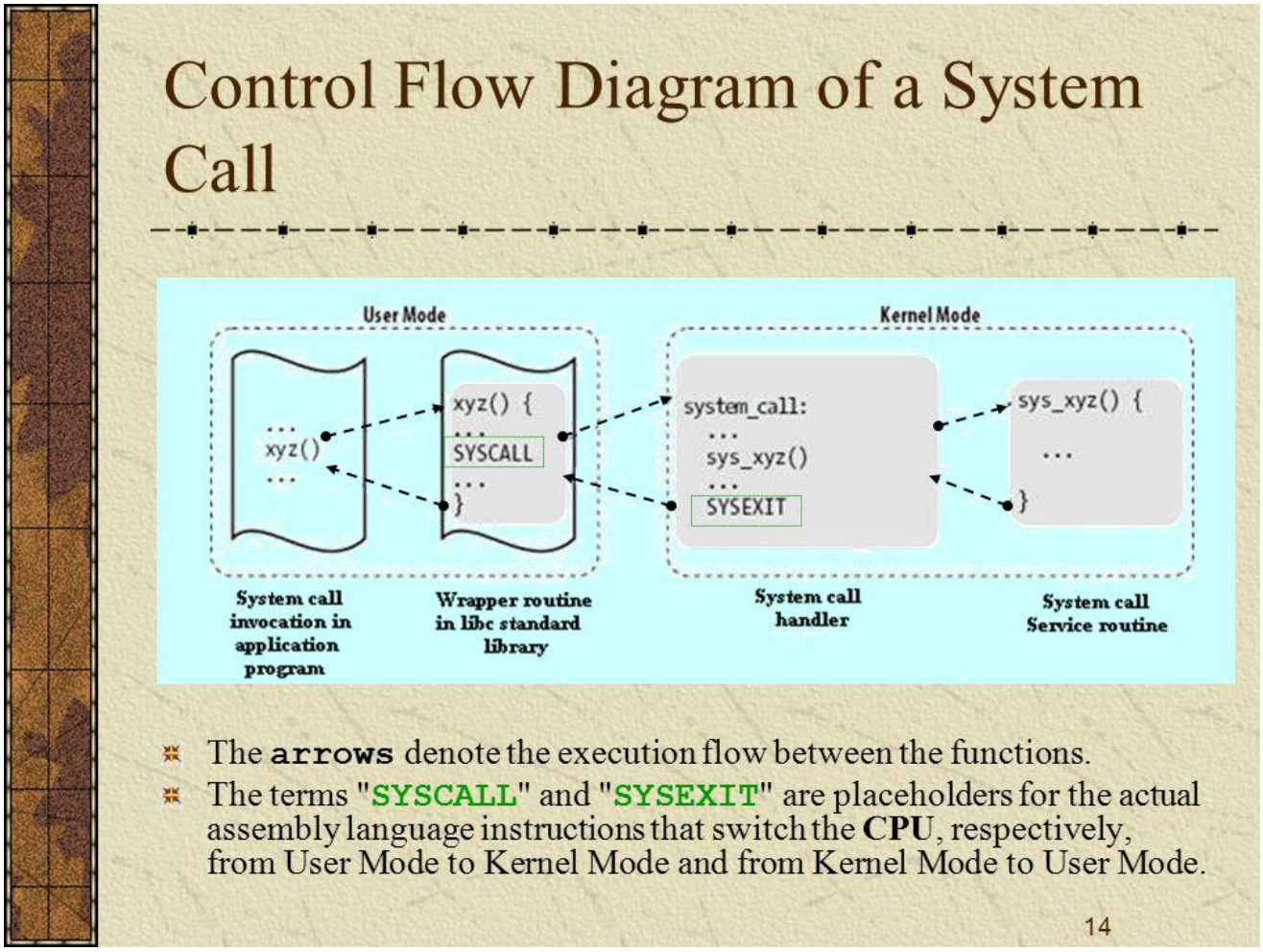
```
// Per-CPU state
struct cpu {
  uchar apicid;                // Local APIC ID
  struct context *scheduler;   // swtch() here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];   // x86 global descriptor table
  volatile uint started;       // Has the CPU started?
  int ncli;                    // Depth of pushcli nesting.
  int intena;                  // Were interrupts enabled before pushcli?
  struct proc *proc;           // The process running on this cpu or null
};

extern struct cpu cpus[NCPU];
extern int ncpu;
```

and the global proc struct variable (in ptable struct from proc.c):

```
File  Edit  View  Search  Terminal  Help
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "x86.h"
#include "proc.h"
#include "spinlock.h"

struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;

static struct proc *initproc;

int nextpid = 1;
extern void forkret(void);
extern void trapret(void);

static void wakeup1(void *chan);
```

This means I can use myproc() and mycpu() to acquire the global variable references in order to use the data from a CPU and processes running in the system.

System call flow diagrams



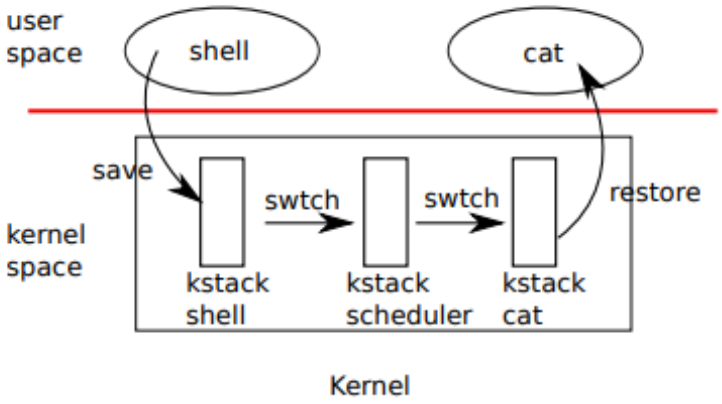


Figure 5-1. Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

Referencing my date user and system call function from project 2:

```
File  Edit  View  Search  Terminal  Help
#include "types.h"
#include "user.h"
#include "date.h"

int
main(int argc, char *argv[])
{
        //NOW IN USER MODE IN A USER FUNCTION

  struct rtcdate r;

  //call the date system call function
  if (date(&r)) { //SWITCH TO KERNEL MODE BY CALLING SYSTEM CALL FUNCTION
    printf(2, "date failed\n");
    exit();
  }

       //RETURN TO USER MODE AND USER FUNCTION

  // print the time as a formatted string
  printf(1, "%d-%d-%d %d:%d:%d\n",
           r.month, r.day, r.year, r.hour, r.minute, r.second);

          //END USER FUNCTION BY CALLING A
  exit(); //SYSTEM CALL TO KILL THE PROCESS; SWITCH TO KERNEL MODE AGAIN
}

~
~
~
"date.c" 27L, 626C written                                    19,0-1        All
```

Using vim editor

Searching a file

When you open a file (for example: vim proc.c), type the following in order to search for a word or phrase in the file:

:/mystring (this will search the file for “mystring”

If the first match is not what you are looking for, simple type :/ to go to the next instance.

Switch file

To edit another file with vim, simple type :e nameOfFile. This will simply exit the current file and enter to the specified file, but the file must be in the same directory as the current file.