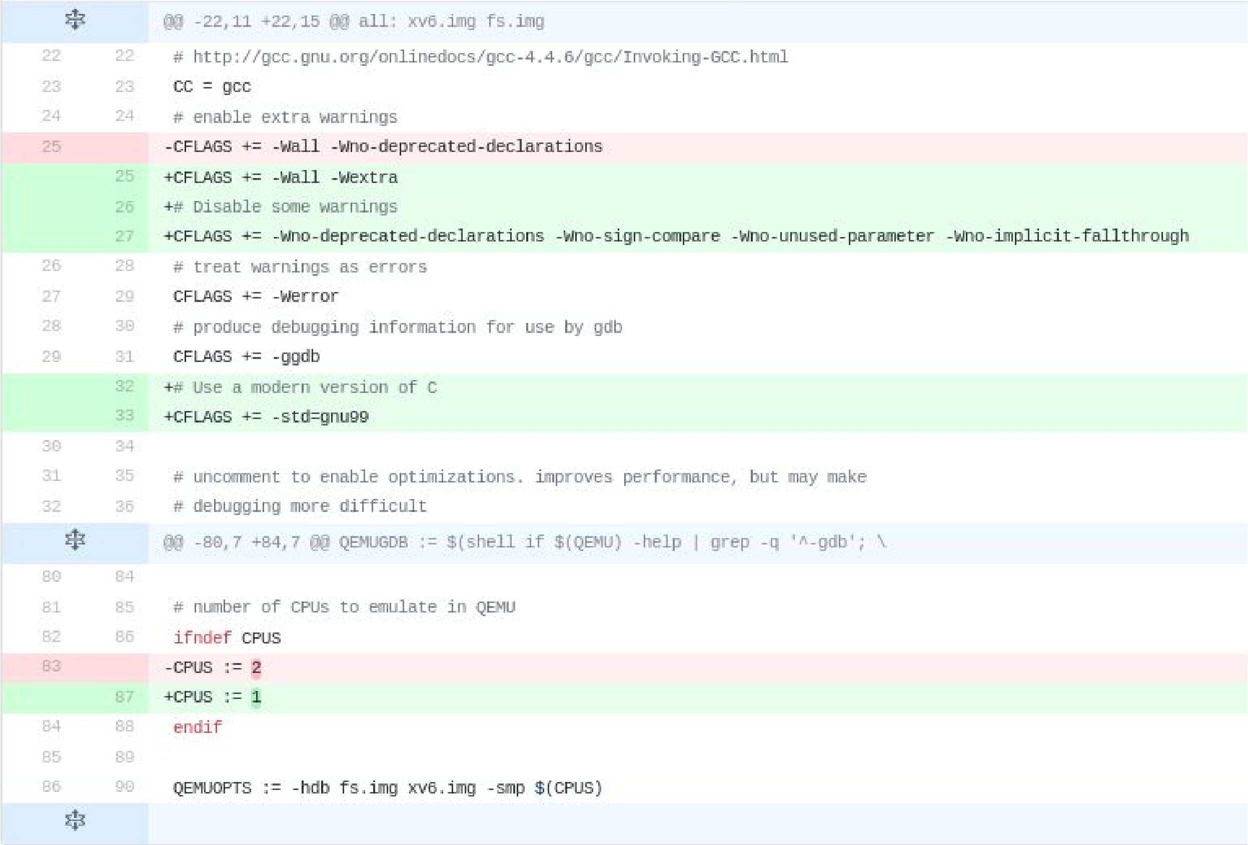
XV6 Lottery Scheduler

Team Members
Dagan Martinez

Source Added and Changed

xv6 root

Makefile.c

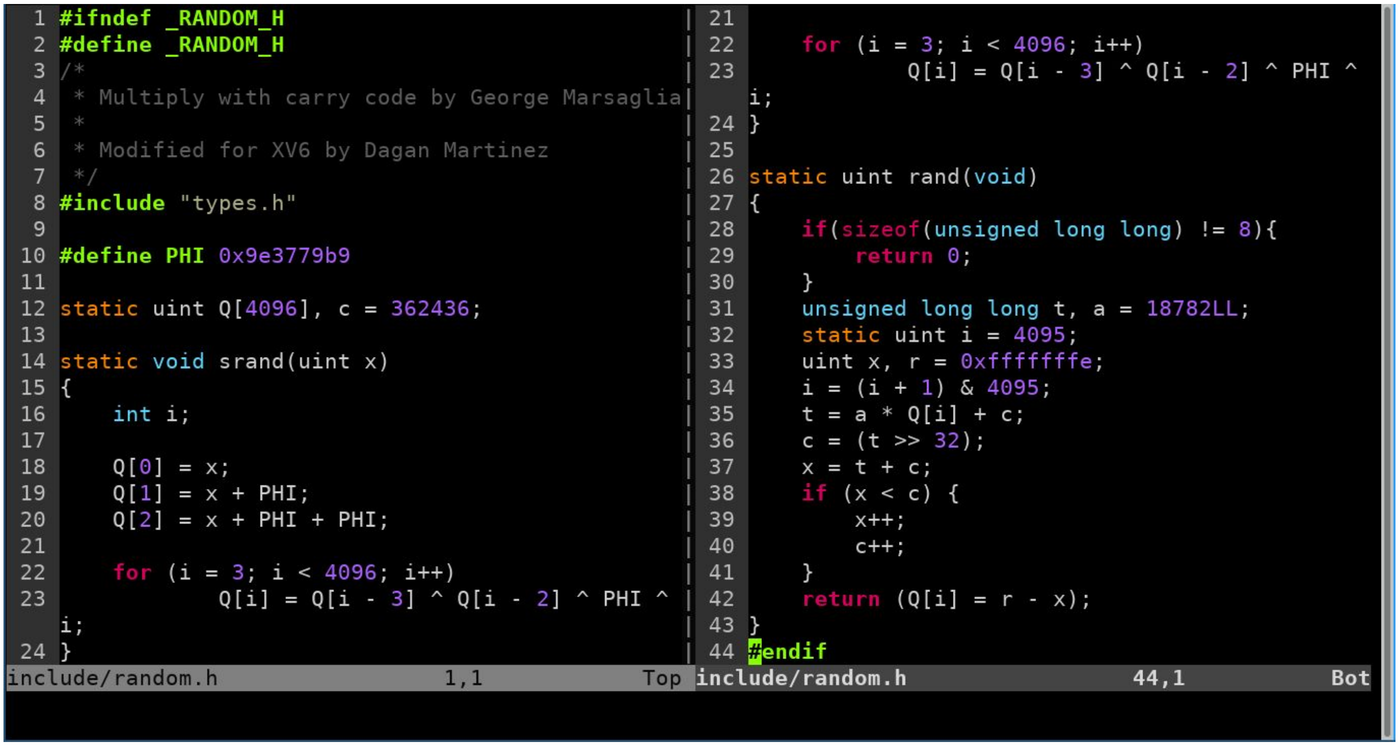


```
@@ -22,11 +22,15 @@ all: xv6.img fs.img
 # http://gcc.gnu.org/onlinedocs/gcc-4.4.6/gcc/Invoking-GCC.html
 CC = gcc
 # enable extra warnings
-CFLAGS += -Wall -Wno-deprecated-declarations
+CFLAGS += -Wall -Wextra
+# Disable some warnings
+CFLAGS += -Wno-deprecated-declarations -Wno-sign-compare -Wno-unused-parameter -Wno-implicit-fallthrough
 # treat warnings as errors
 CFLAGS += -Werror
 # produce debugging information for use by gdb
 CFLAGS += -ggdb
+# Use a modern version of C
+CFLAGS += -std=gnu99

 # uncomment to enable optimizations. improves performance, but may make
 # debugging more difficult
@@ -80,7 +84,7 @@ QEMUGDB := $(shell if $(QEMU) -help | grep -q '^-gdb'; \

 # number of CPUs to emulate in QEMU
 ifndef CPUS
-CPUS := 2
+CPUS := 1
 endif

 QEMUOPTS := -hdb fs.img xv6.img -smp $(CPUS)
```

I changed the number of CPUs to be run from 2 to 1, to better show lottery scheduling with two processes. I also added and suppressed some errors to taste, and fixed the C version instead of falling back on the default choice of that version of GCC.

xv6/include

Random.h



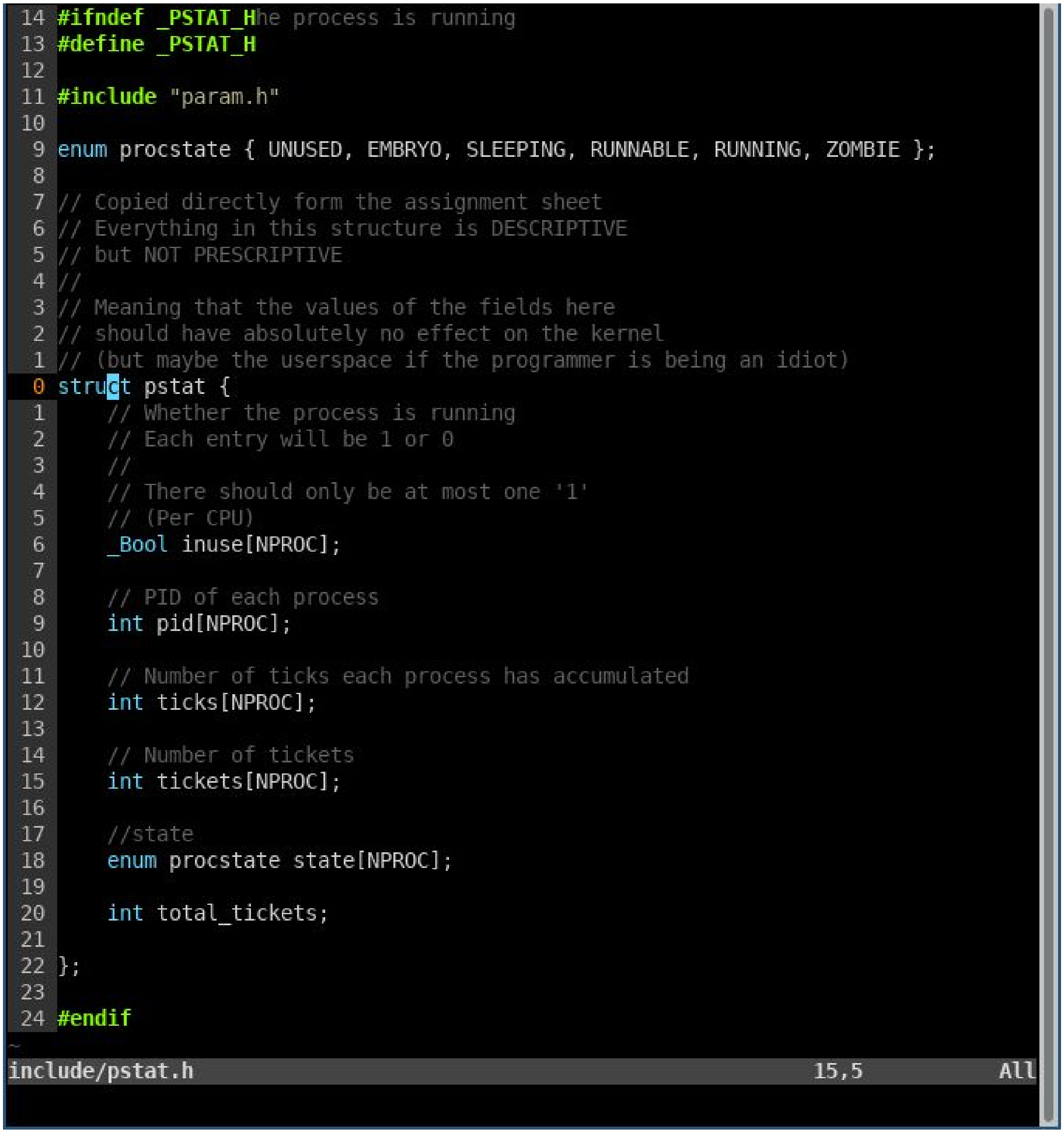
```c
#ifndef _RANDOM_H
#define _RANDOM_H
/*
 * Multiply with carry code by George Marsaglia
 *
 * Modified for XV6 by Dagan Martinez
 */
#include "types.h"

#define PHI 0x9e3779b9

static uint Q[4096], c = 362436;

static void srand(uint x)
{
    int i;

    Q[0] = x;
    Q[1] = x + PHI;
    Q[2] = x + PHI + PHI;

    for (i = 3; i < 4096; i++)
            Q[i] = Q[i - 3] ^ Q[i - 2] ^ PHI ^ i;
}
```

```c

    for (i = 3; i < 4096; i++)
            Q[i] = Q[i - 3] ^ Q[i - 2] ^ PHI ^ i;
}
```

```c

static uint rand(void)
{
    if(sizeof(unsigned long long) != 8){
        return 0;
    }
    unsigned long long t, a = 18782LL;
    static uint i = 4095;
    uint x, r = 0xfffffffe;
    i = (i + 1) & 4095;
    t = a * Q[i] + c;
    c = (t >> 32);
    x = t + c;
    if (x < c) {
        x++;
        c++;
    }
    return (Q[i] = r - x);
}
#endif
```

Most of this code was taken from Wikipedia, with some slight modification. I put static functions in a header file so the same code would be available to both the kernel and user(for easier testing).

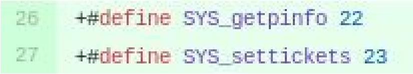
Pstat.h



```
#ifndef _PSTAT_Hhe process is running
#define _PSTAT_H

#include "param.h"

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

// Copied directly form the assignment sheet
// Everything in this structure is DESCRIPTIVE
// but NOT PRESCRIPTIVE
//
// Meaning that the values of the fields here
// should have absolutely no effect on the kernel
// (but maybe the userspace if the programmer is being an idiot)
struct pstat {
    // Whether the process is running
    // Each entry will be 1 or 0
    //
    // There should only be at most one '1'
    // (Per CPU)
    _Bool inuse[NPROC];

    // PID of each process
    int pid[NPROC];

    // Number of ticks each process has accumulated
    int ticks[NPROC];

    // Number of tickets
    int tickets[NPROC];

    //state
    enum procstate state[NPROC];

    int total_tickets;

};

#endif
~
include/pstat.h                                          15,5          All
```

This structure is purely descriptive. In addition to the three fields given in Miniproject2.pdf, I added the tickets and state fields, as well as a non-array field: total_tickets, which holds the amount of ticks held by all processes combined(not counting sleeping processes)

Syscall.h

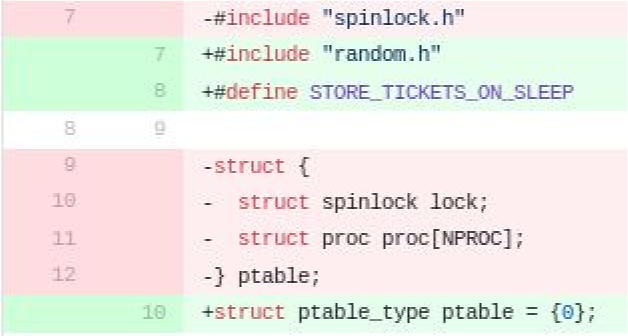


```
+#define SYS_getpinfo 22
+#define SYS_settickets 23
```

I had to add the syscall numbers for my two new syscalls

xv6/kernel

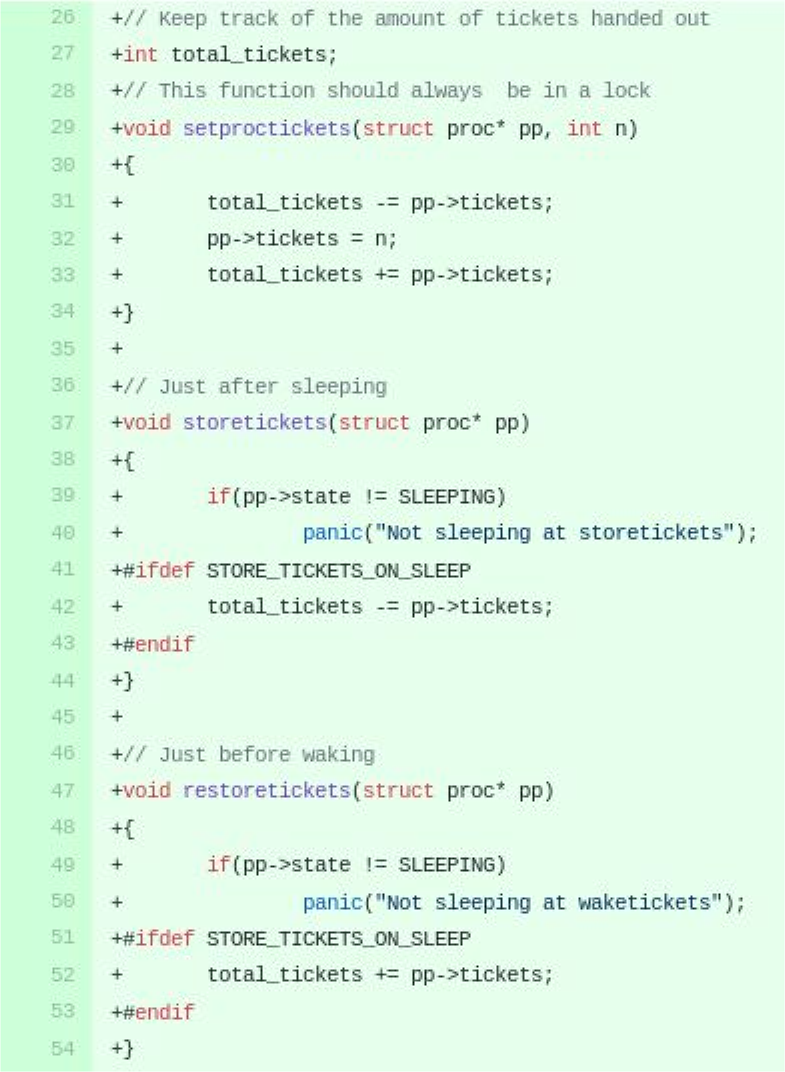
proc.c



```
-#include "spinlock.h"
+#include "random.h"
+#define STORE_TICKETS_ON_SLEEP

-struct {
-  struct spinlock lock;
-  struct proc proc[NPROC];
-} ptable;
+struct ptable_type ptable = {0};
```

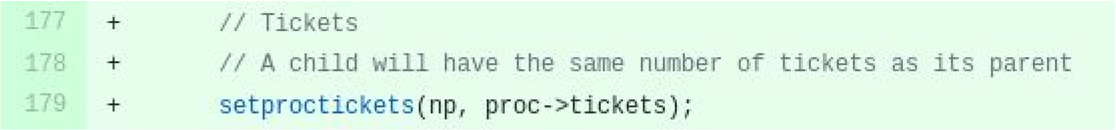
I had to move the struct definition of ptable to proc.c, so I could access ptable in sysproc.c (for my getpinfo syscall)

STORE_TICKETS_ON_SLEEP is an option to not count sleeping processes in the lottery



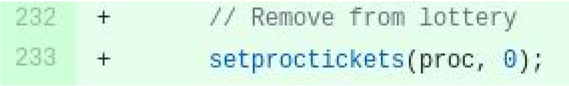
```
+// Keep track of the amount of tickets handed out
+int total_tickets;
+// This function should always  be in a lock
+void setproctickets(struct proc* pp, int n)
+{
+       total_tickets -= pp->tickets;
+       pp->tickets = n;
+       total_tickets += pp->tickets;
+}
+
+// Just after sleeping
+void storetickets(struct proc* pp)
+{
+       if(pp->state != SLEEPING)
+               panic("Not sleeping at storetickets");
+#ifdef STORE_TICKETS_ON_SLEEP
+       total_tickets -= pp->tickets;
+#endif
+}
+
+// Just before waking
+void restoretickets(struct proc* pp)
+{
+       if(pp->state != SLEEPING)
+               panic("Not sleeping at waketickets");
+#ifdef STORE_TICKETS_ON_SLEEP
+       total_tickets += pp->tickets;
+#endif
+}
```

These are functions used to set, store, and restore the tickets of a process



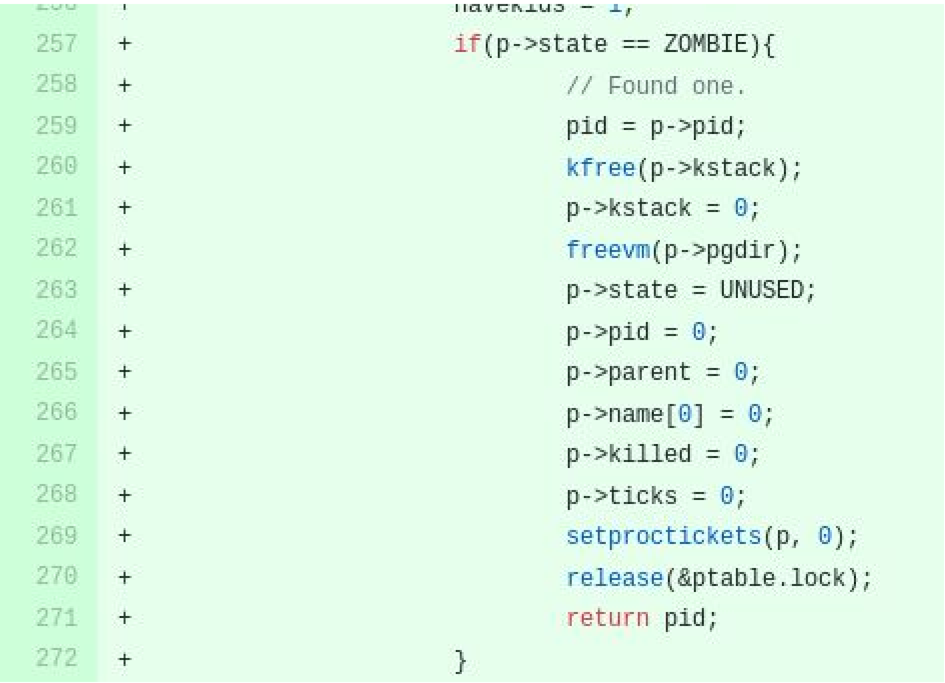
```
+       // Tickets
+       // A child will have the same number of tickets as its parent
+       setproctickets(np, proc->tickets);
```

When a process fork()s, its child should have the same number of tickets



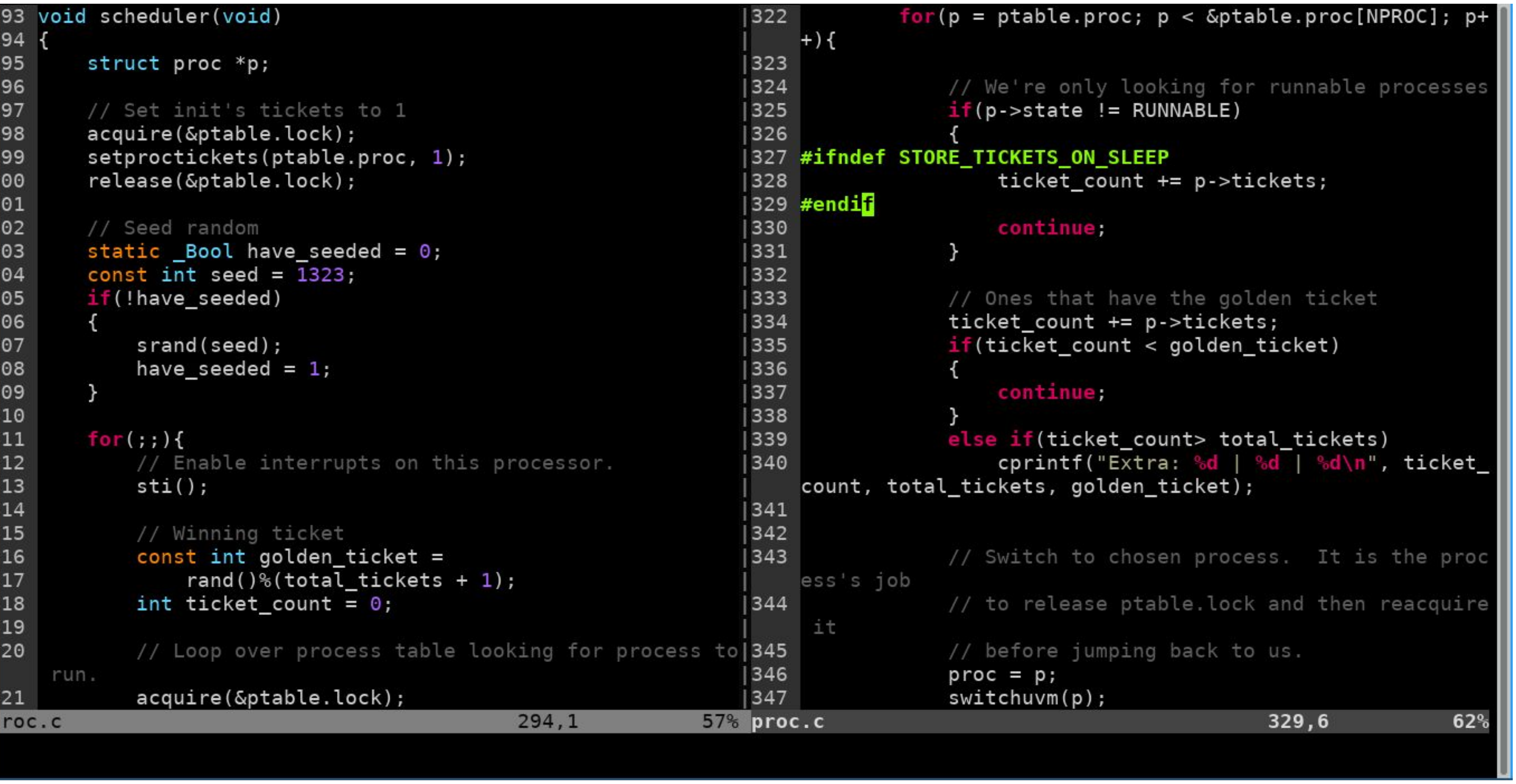
```
+       // Remove from lottery
+       setproctickets(proc, 0);
```

In exit(), we have to remove the process from the lottery



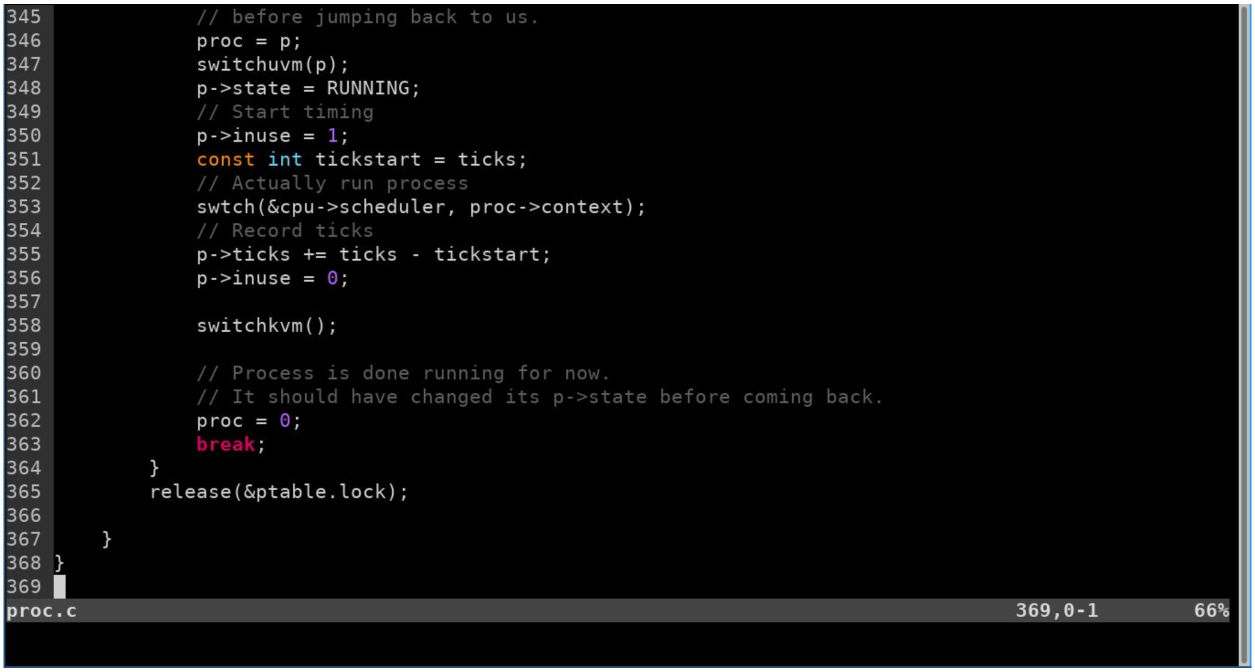
```
+                       havekids = 1;
+                       if(p->state == ZOMBIE){
+                               // Found one.
+                               pid = p->pid;
+                               kfree(p->kstack);
+                               p->kstack = 0;
+                               freevm(p->pgdir);
+                               p->state = UNUSED;
+                               p->pid = 0;
+                               p->parent = 0;
+                               p->name[0] = 0;
+                               p->killed = 0;
+                               p->ticks = 0;
+                               setproctickets(p, 0);
+                               release(&ptable.lock);
+                               return pid;
+                       }
```

When we’re cleaning up zombie processes, we should also remove them from the lottery (in wait()) and clear the ticks



```
void scheduler(void)
{
    struct proc *p;

    // Set init's tickets to 1
    acquire(&ptable.lock);
    setproctickets(ptable.proc, 1);
    release(&ptable.lock);

    // Seed random
    static _Bool have_seeded = 0;
    const int seed = 1323;
    if(!have_seeded)
    {
        srand(seed);
        have_seeded = 1;
    }

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

        // Winning ticket
        const int golden_ticket =
            rand()%(total_tickets + 1);
        int ticket_count = 0;

        // Loop over process table looking for process to run.
        acquire(&ptable.lock);
roc.c                                         294,1          57%
```

```
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

            // We're only looking for runnable processes
            if(p->state != RUNNABLE)
            {
#ifndef STORE_TICKETS_ON_SLEEP
                ticket_count += p->tickets;
#endif
                continue;
            }

            // Ones that have the golden ticket
            ticket_count += p->tickets;
            if(ticket_count < golden_ticket)
            {
                continue;
            }
            else if(ticket_count> total_tickets)
                cprintf("Extra: %d | %d | %d\n", ticket_count, total_tickets, golden_ticket);


            // 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;
            switchuvm(p);
proc.c                                   329,6          62%
```



```
            // before jumping back to us.
            proc = p;
            switchuvm(p);
            p->state = RUNNING;
            // Start timing
            p->inuse = 1;
            const int tickstart = ticks;
            // Actually run process
            swtch(&cpu->scheduler, proc->context);
            // Record ticks
            p->ticks += ticks - tickstart;
            p->inuse = 0;

            switchkvm();

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

    }
}

proc.c                                                    369,0-1       66%
```

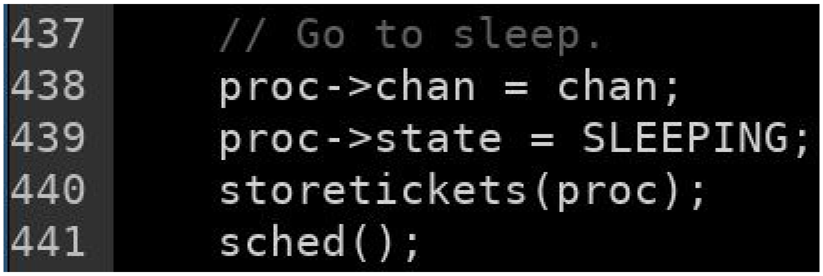
The scheduler function had the most changes. First, I had to set the first process's tickets to 1(which will set all children's processes to 1 via fork()). Next I had to seed the random function with an appropriate value.

In each iteration of the forever loop, I had to create a new winning ticket, and set the “ticket count” to zero.

When I’m foreaching through the process table, I have to decide whether or not I want to count sleeping processes in the lottery. As I defined STORE_TICKETS_ON_SLEEP previously, I do NOT add tickets from non-runnable processes.

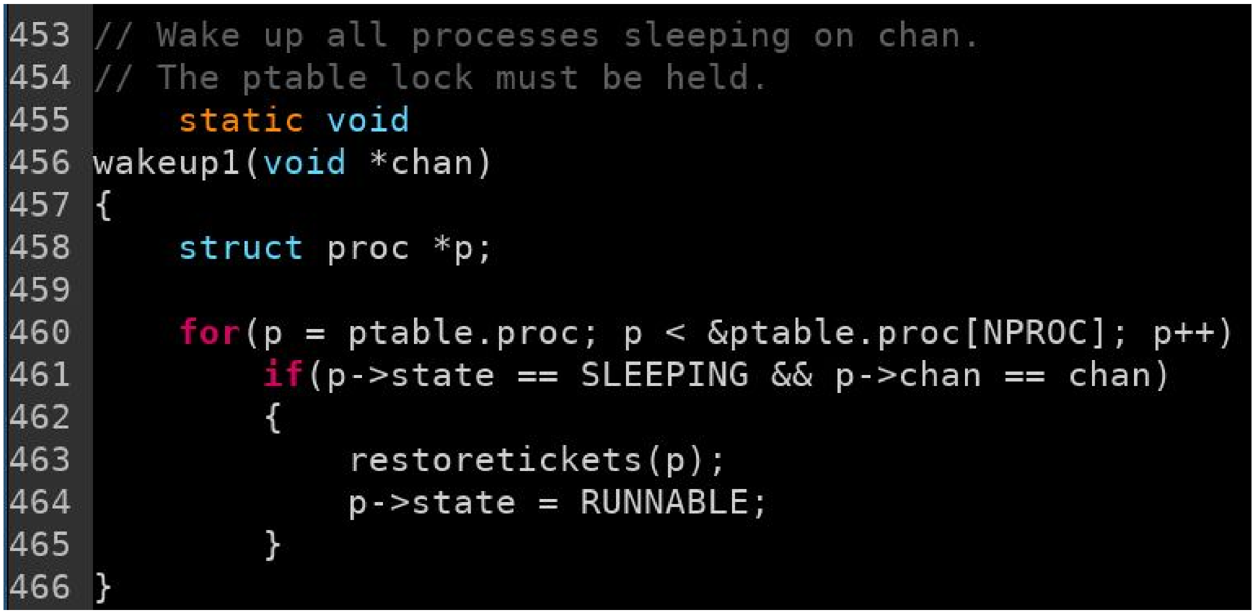
Now, I have to filter any process that does not have the winning ticket to go back to the beginning of the loop. Note that only one process will win, after which the loop is restarted.

Before the processes is run, the timer must be resumed. After the process has run, the timer is stopped again.



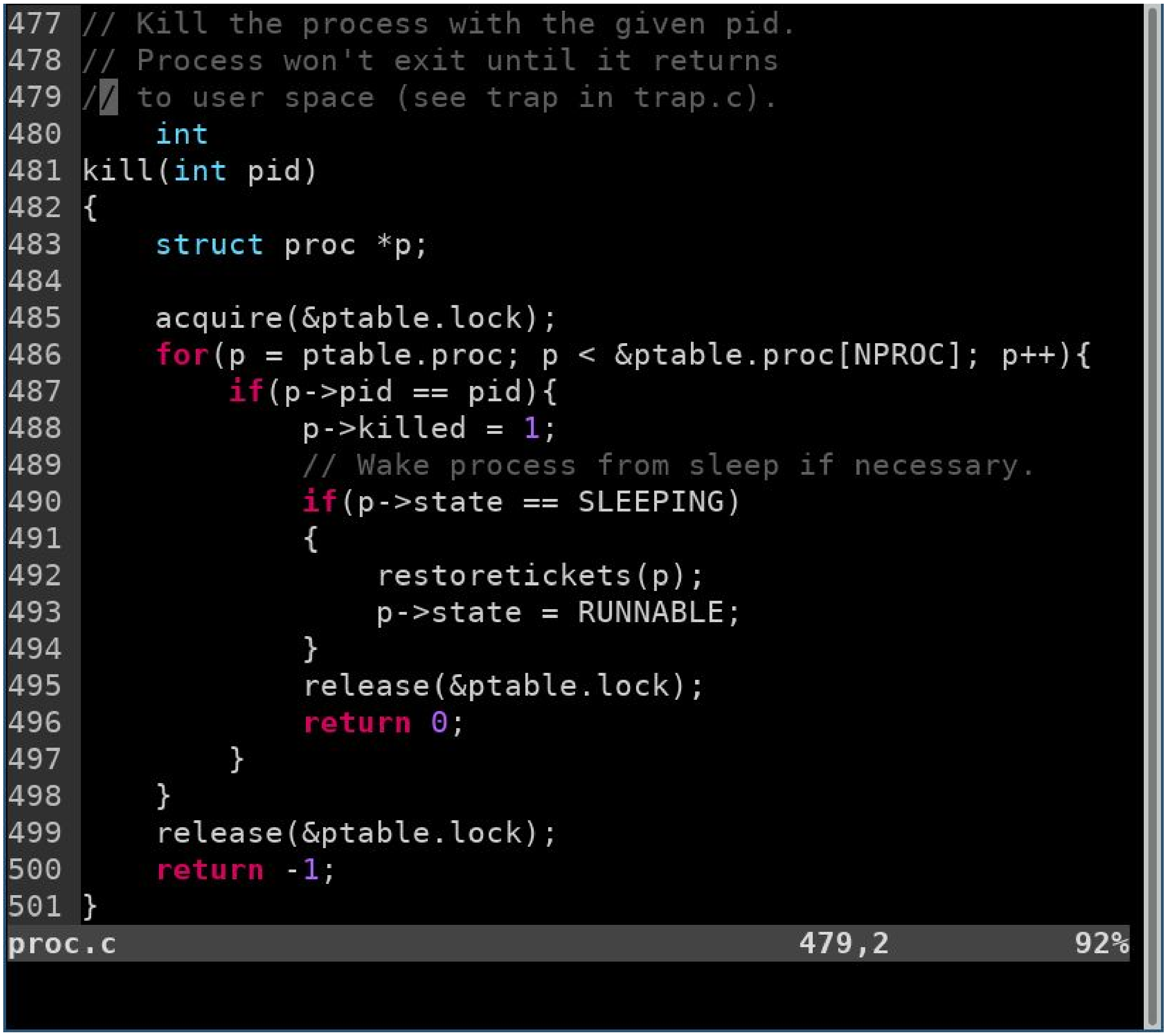
```
    // Go to sleep.
    proc->chan = chan;
    proc->state = SLEEPING;
    storetickets(proc);
    sched();
```

In sleep(), I have to store the tickets as previously mentioned. This temporarily removes them from the lottery.



```
// Wake up all processes sleeping on chan.
// The ptable lock must be held.
    static void
wakeup1(void *chan)
{
    struct proc *p;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if(p->state == SLEEPING && p->chan == chan)
        {
            restoretickets(p);
            p->state = RUNNABLE;
        }
}
```

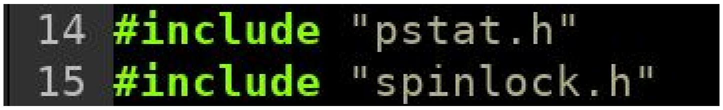
Inside wakeup1, I restore those tickets



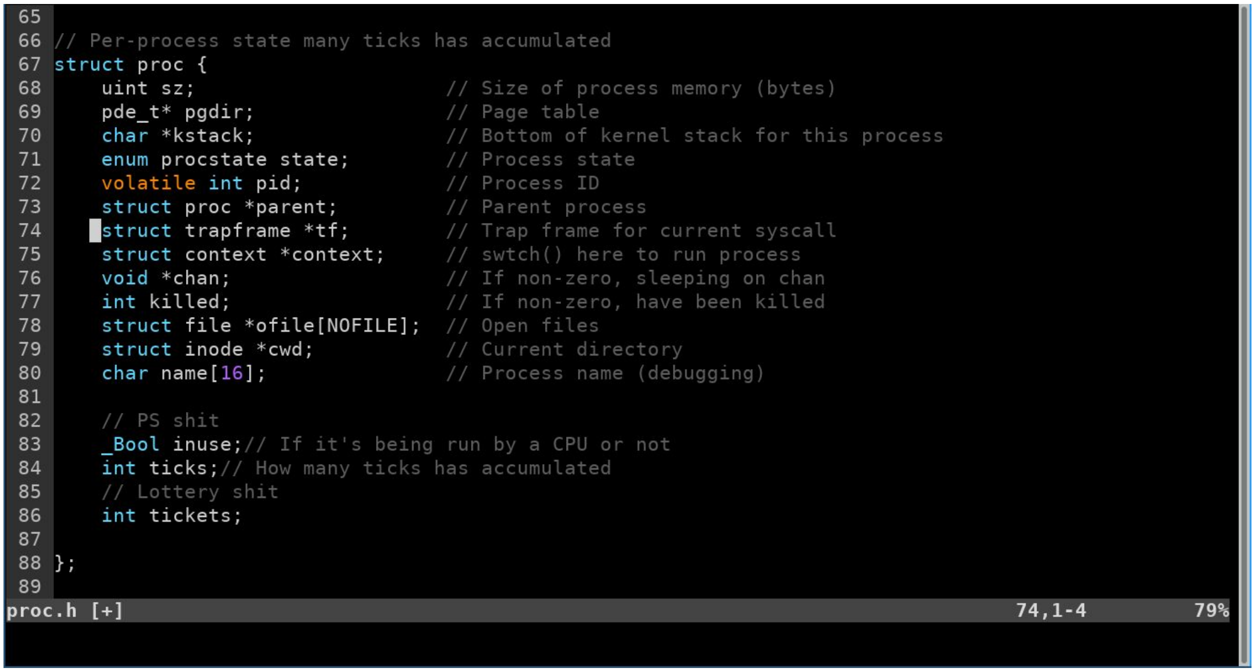
```
// Kill the process with the given pid.
// Process won't exit until it returns
// to user space (see trap in trap.c).
    int
kill(int pid)
{
    struct proc *p;

    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid == pid){
            p->killed = 1;
            // Wake process from sleep if necessary.
            if(p->state == SLEEPING)
            {
                restoretickets(p);
                p->state = RUNNABLE;
            }
            release(&ptable.lock);
            return 0;
        }
    }
    release(&ptable.lock);
    return -1;
}
proc.c                                             479,2          92%
```

Kill does not actually remove the process from the ptable, so I do restore tickets if the process is sleeping(which will then almost immediately be removed again)

Proc.h

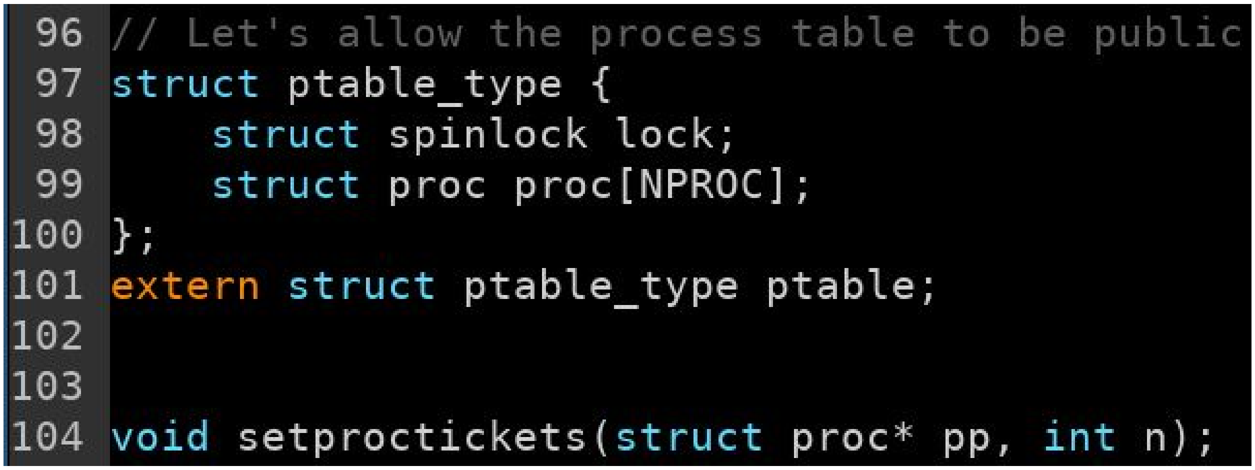


```
#include "pstat.h"
#include "spinlock.h"
```



```
// Per-process state many ticks has accumulated
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
    volatile 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)

    // PS shit
    _Bool inuse;// If it's being run by a CPU or not
    int ticks;// How many ticks has accumulated
    // Lottery shit
    int tickets;

};

proc.h [+]                                                  74,1-4         79%
```

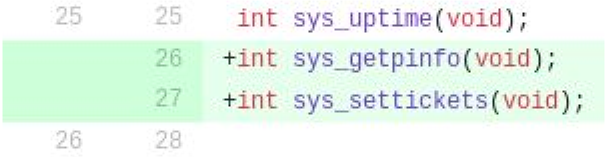
I store some of the information from pstat in proc, to make the code cleaner. Inuse is effectively (state==RUNNING)



```
// Let's allow the process table to be public
struct ptable_type {
    struct spinlock lock;
    struct proc proc[NPROC];
};
extern struct ptable_type ptable;


void setproctickets(struct proc* pp, int n);
```

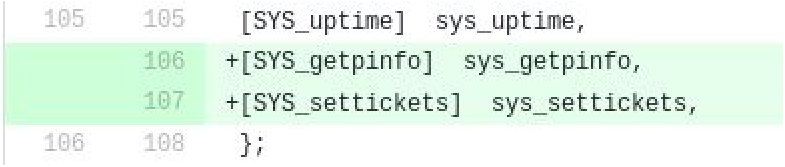
I mentioned previously that I moved the definition of ptable’s struct to proc.h so I could access ptable from other files

Sysfunc.h



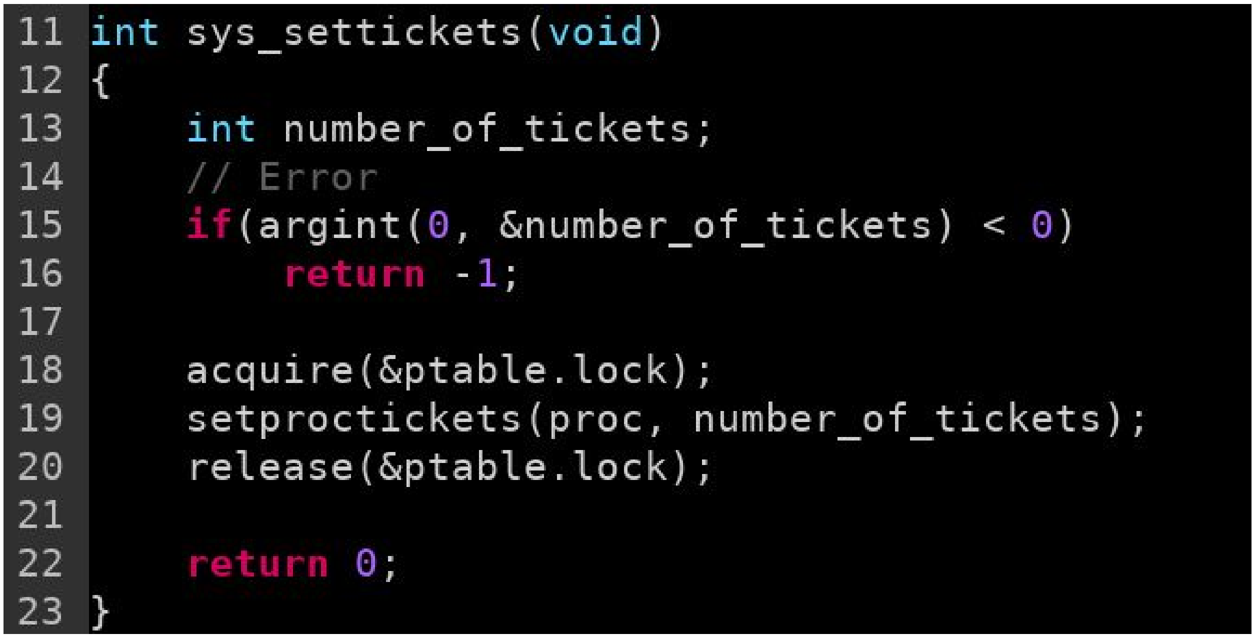
```
 int sys_uptime(void);
+int sys_getpinfo(void);
+int sys_settickets(void);

```

syscall.c



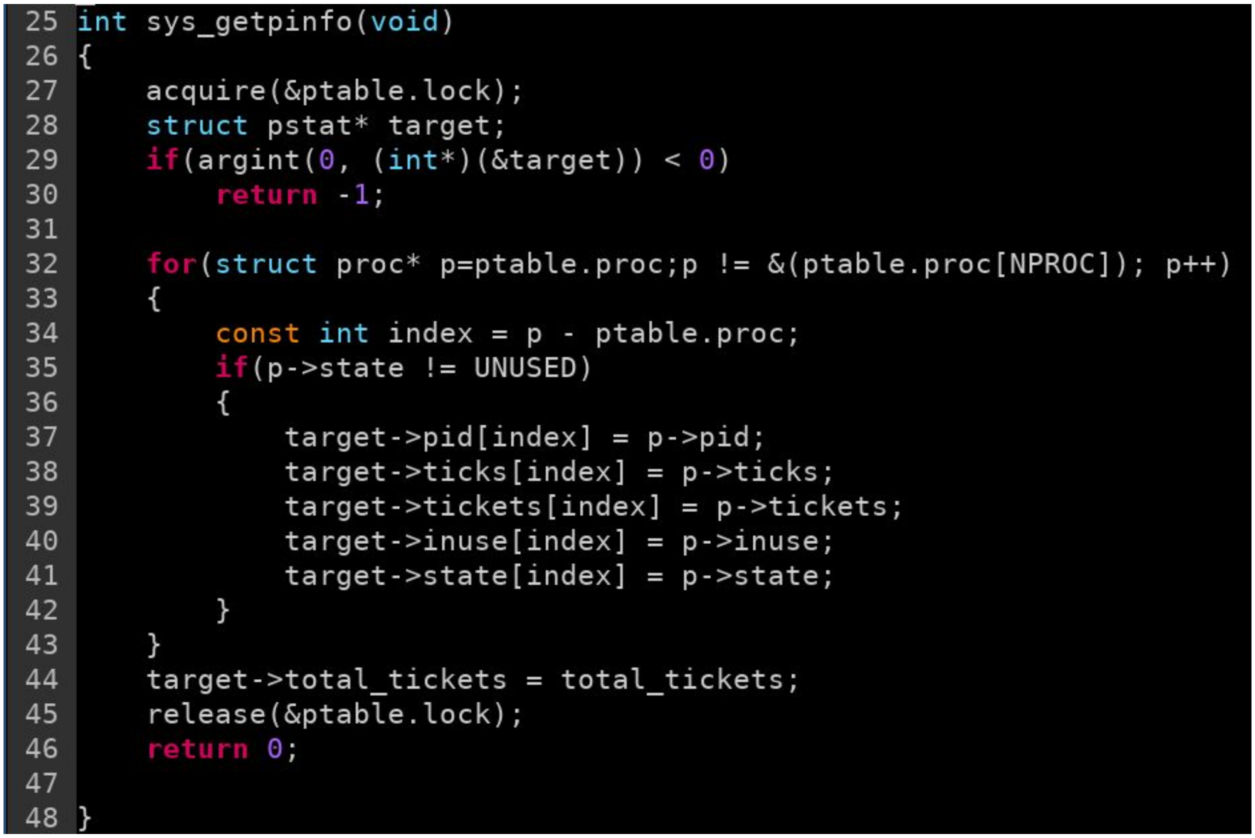
```
 [SYS_uptime]   sys_uptime,
+[SYS_getpinfo]   sys_getpinfo,
+[SYS_settickets]   sys_settickets,
 };
```

Sysproc.c



```
int sys_settickets(void)
{
    int number_of_tickets;
    // Error
    if(argint(0, &number_of_tickets) < 0)
        return -1;

    acquire(&ptable.lock);
    setproctickets(proc, number_of_tickets);
    release(&ptable.lock);

    return 0;
}
```

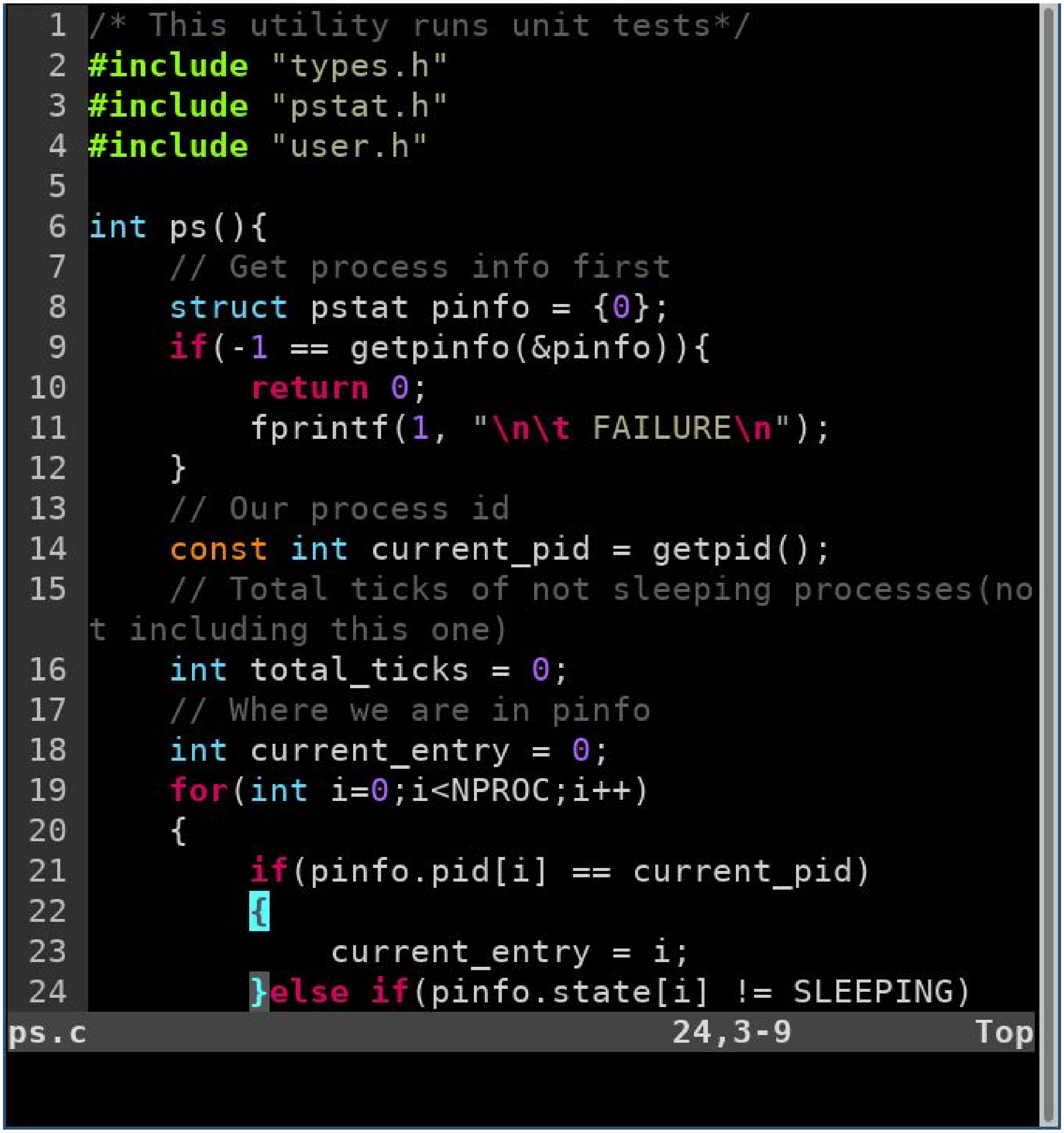
Sys_settickets changes the number of tickets of a process, and returns 0 on success and -1 on error.



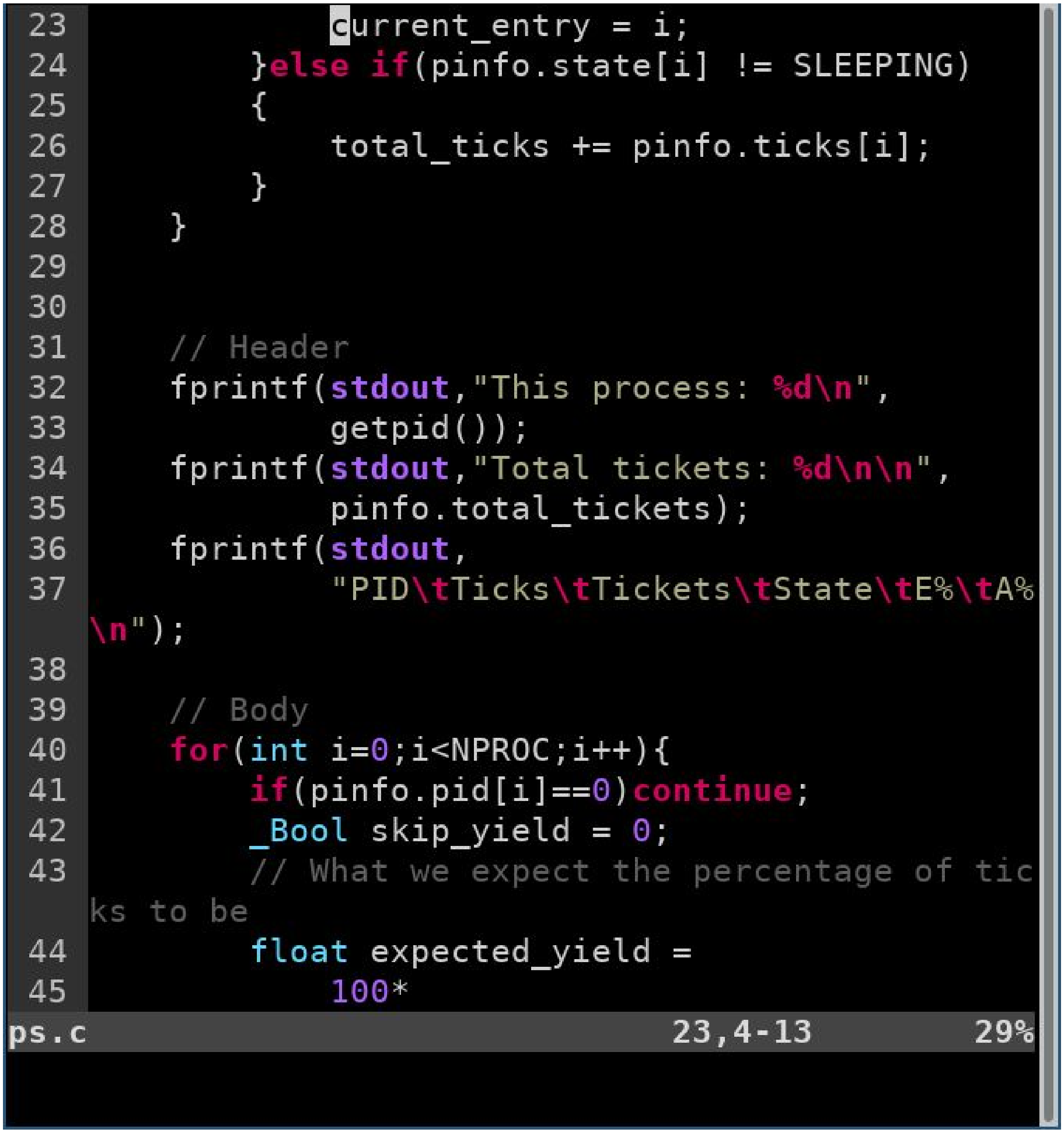
```
int sys_getpinfo(void)
{
    acquire(&ptable.lock);
    struct pstat* target;
    if(argint(0, (int*)(&target)) < 0)
        return -1;

    for(struct proc* p=ptable.proc;p != &(ptable.proc[NPROC]); p++)
    {
        const int index = p - ptable.proc;
        if(p->state != UNUSED)
        {
            target->pid[index] = p->pid;
            target->ticks[index] = p->ticks;
            target->tickets[index] = p->tickets;
            target->inuse[index] = p->inuse;
            target->state[index] = p->state;
        }
    }
    target->total_tickets = total_tickets;
    release(&ptable.lock);
    return 0;

}
```

Sys_getpinfo fills a pstat struct with data from the ptable. I have to do this all within a lock for the case of multiple CPUs xv6/user

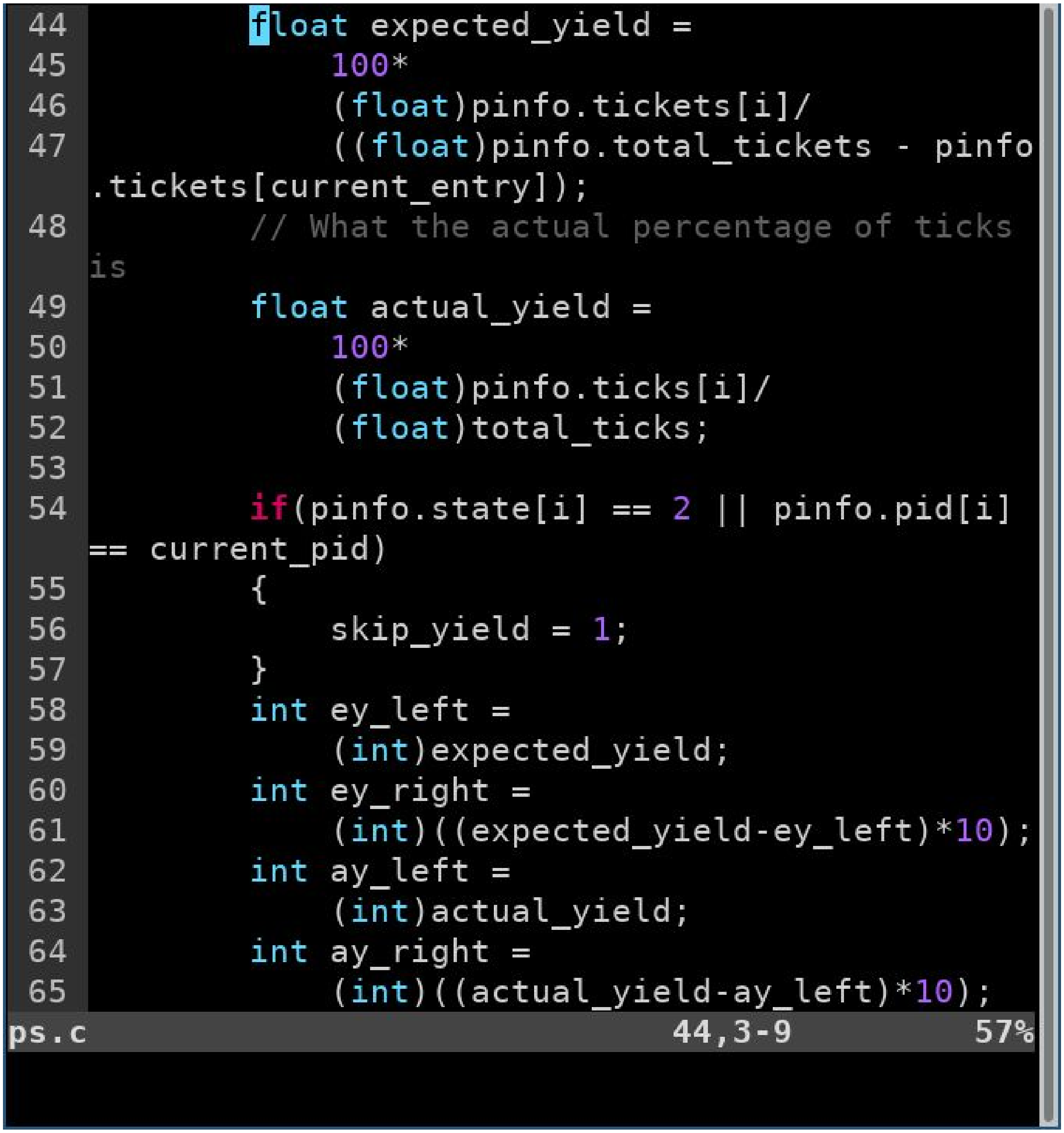
Ps.c



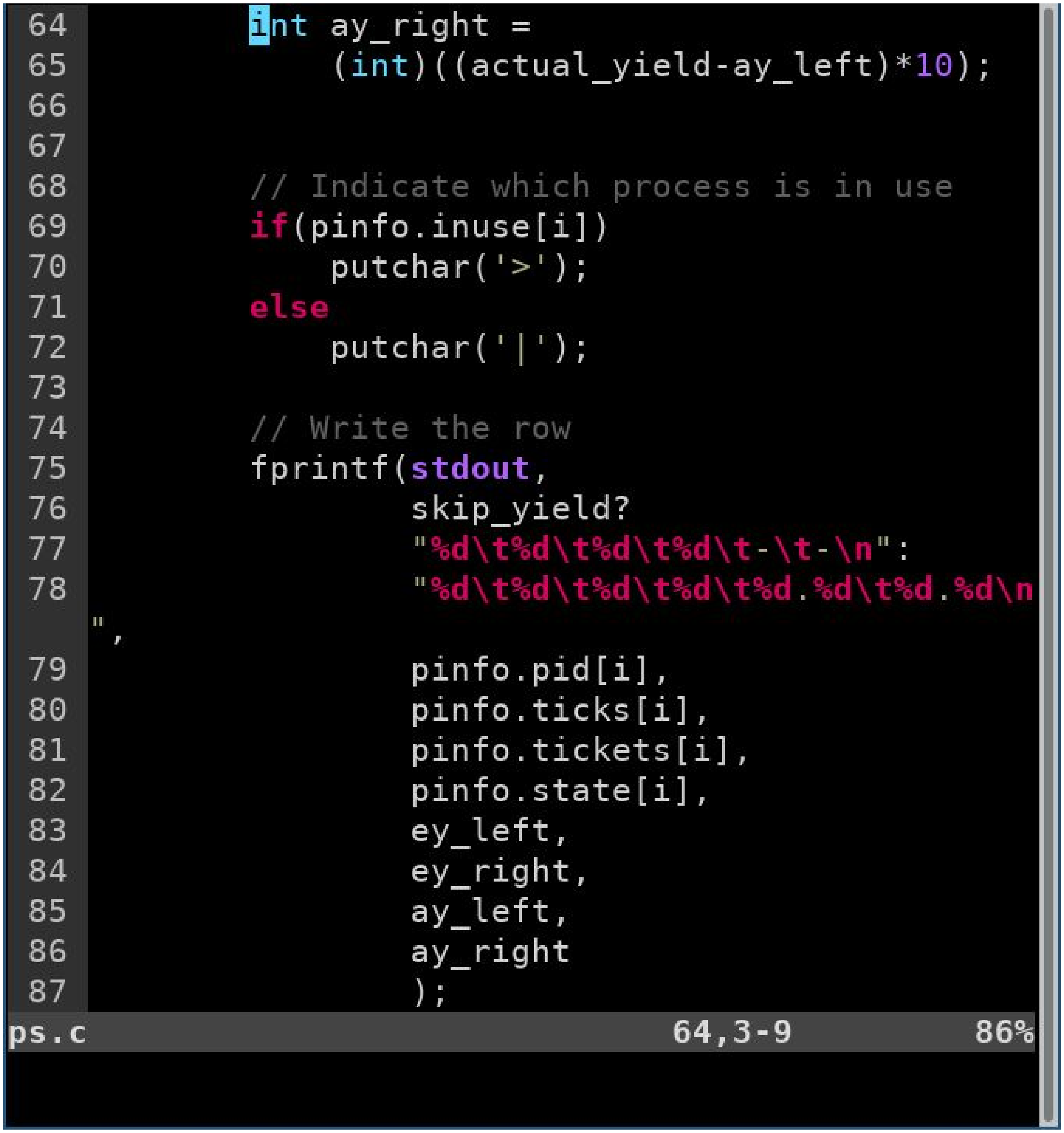
```
/* This utility runs unit tests*/
#include "types.h"
#include "pstat.h"
#include "user.h"

int ps(){
    // Get process info first
    struct pstat pinfo = {0};
    if(-1 == getpinfo(&pinfo)){
        return 0;
        fprintf(1, "\n\t FAILURE\n");
    }
    // Our process id
    const int current_pid = getpid();
    // Total ticks of not sleeping processes(not including this one)
    int total_ticks = 0;
    // Where we are in pinfo
    int current_entry = 0;
    for(int i=0;i<NPROC;i++)
    {
        if(pinfo.pid[i] == current_pid)
        {
            current_entry = i;
        }else if(pinfo.state[i] != SLEEPING)
```



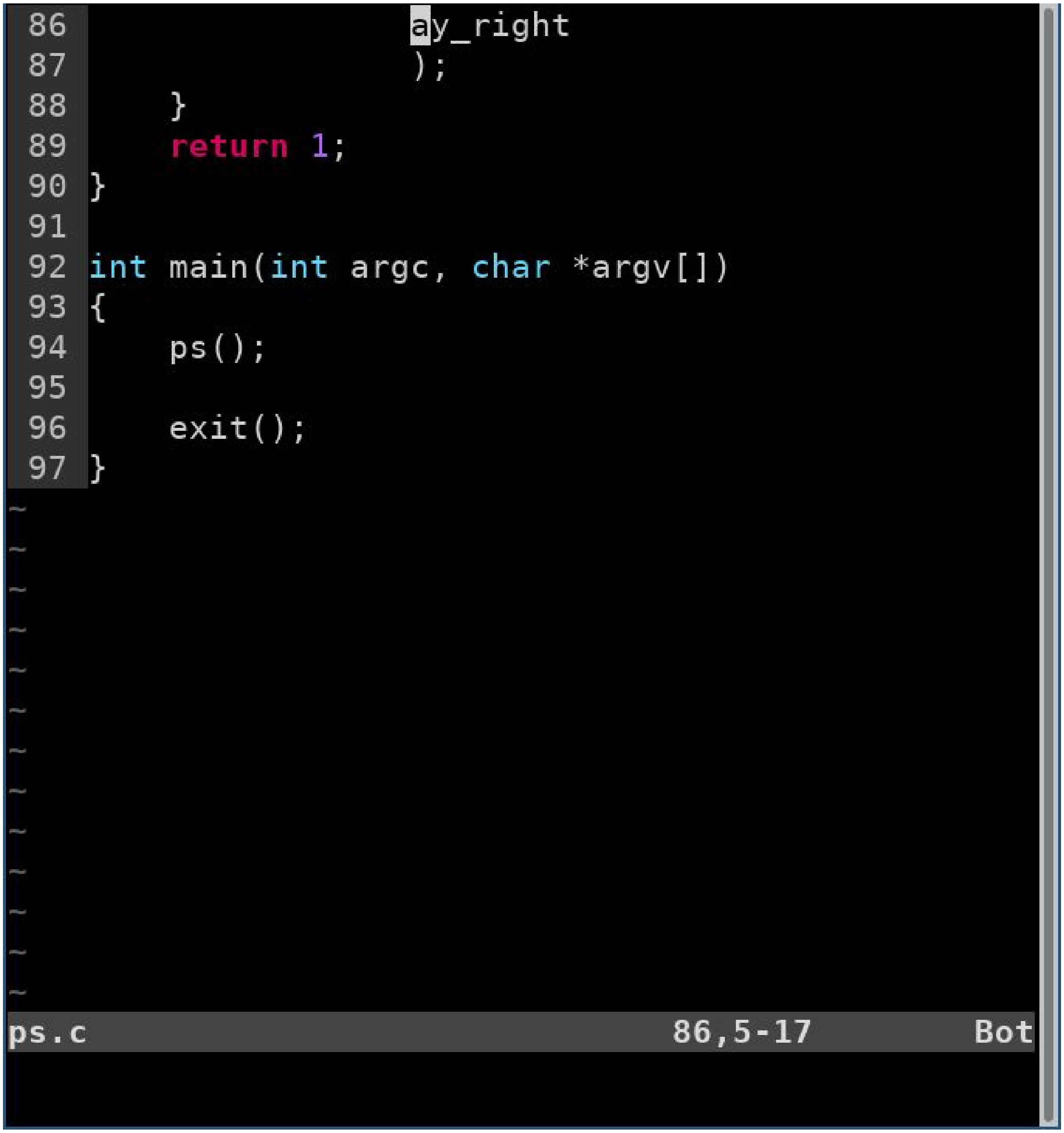
```
            current_entry = i;
        }else if(pinfo.state[i] != SLEEPING)
        {
            total_ticks += pinfo.ticks[i];
        }
    }


    // Header
    fprintf(stdout,"This process: %d\n",
            getpid());
    fprintf(stdout,"Total tickets: %d\n\n",
            pinfo.total_tickets);
    fprintf(stdout,
            "PID\tTicks\tTickets\tState\tE%\tA%\n");

    // Body
    for(int i=0;i<NPROC;i++){
        if(pinfo.pid[i]==0)continue;
        _Bool skip_yield = 0;
        // What we expect the percentage of ticks to be
        float expected_yield =
            100*
ps.c                                    23,4-13       29%
```



```
        float expected_yield =
            100*
            (float)pinfo.tickets[i]/
            ((float)pinfo.total_tickets - pinfo.tickets[current_entry]);
        // What the actual percentage of ticks is
        float actual_yield =
            100*
            (float)pinfo.ticks[i]/
            (float)total_ticks;

        if(pinfo.state[i] == 2 || pinfo.pid[i] == current_pid)
        {
            skip_yield = 1;
        }
        int ey_left =
            (int)expected_yield;
        int ey_right =
            (int)((expected_yield-ey_left)*10);
        int ay_left =
            (int)actual_yield;
        int ay_right =
            (int)((actual_yield-ay_left)*10);
ps.c                                     44,3-9         57%
```



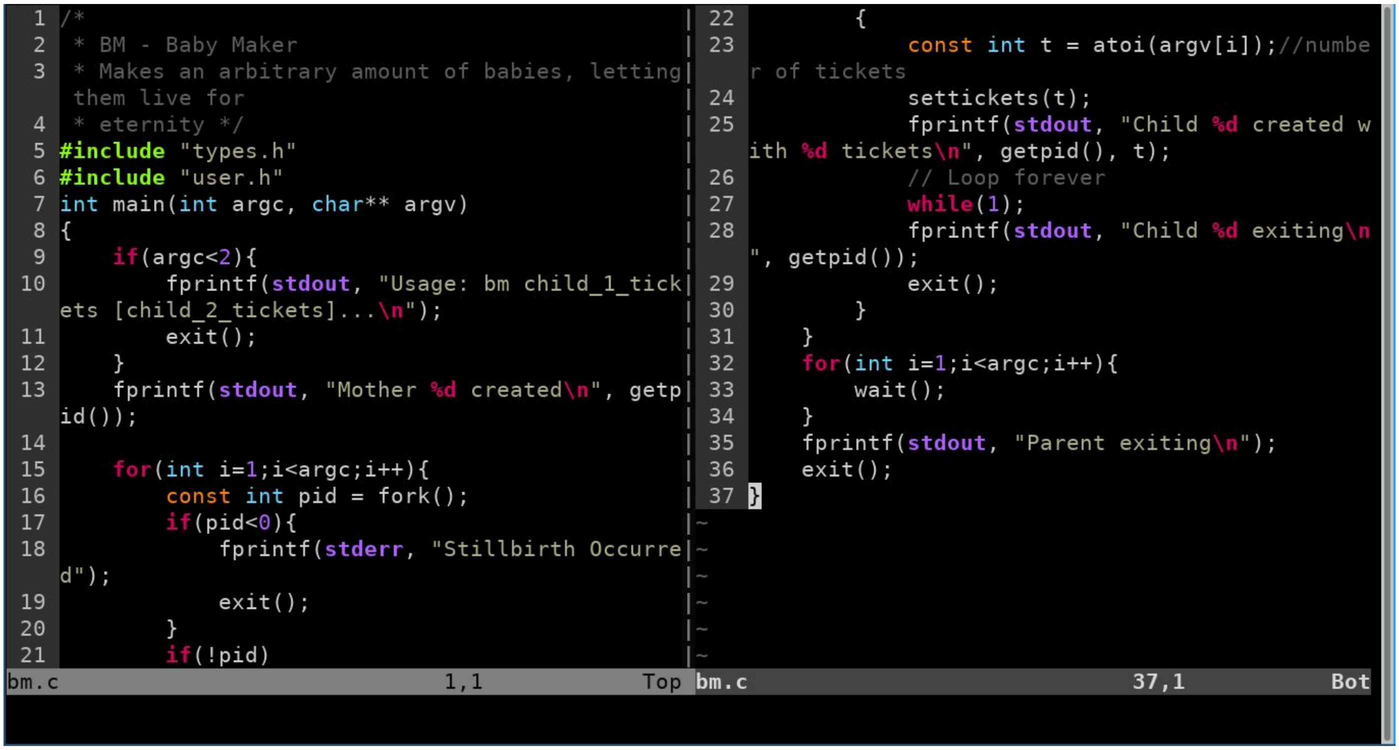
```
        int ay_right =
            (int)((actual_yield-ay_left)*10);


        // Indicate which process is in use
        if(pinfo.inuse[i])
            putchar('>');
        else
            putchar('|');

        // Write the row
        fprintf(stdout,
                skip_yield?
                "%d\t%d\t%d\t%d\t-\t-\n":
                "%d\t%d\t%d\t%d\t%d.%d\t%d.%d\n
",
                pinfo.pid[i],
                pinfo.ticks[i],
                pinfo.tickets[i],
                pinfo.state[i],
                ey_left,
                ey_right,
                ay_left,
                ay_right
                );
ps.c                             64,3-9         86%
```



```
                ay_right
                );
    }
    return 1;
}

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

    exit();
}
```

This is the user program “PS” that displays the PID, ticks, tickets, state, expected ticks %, and actual ticks %, of each process, as well as whether the process is in use by displaing “|”(not in use) or “>”(in use)

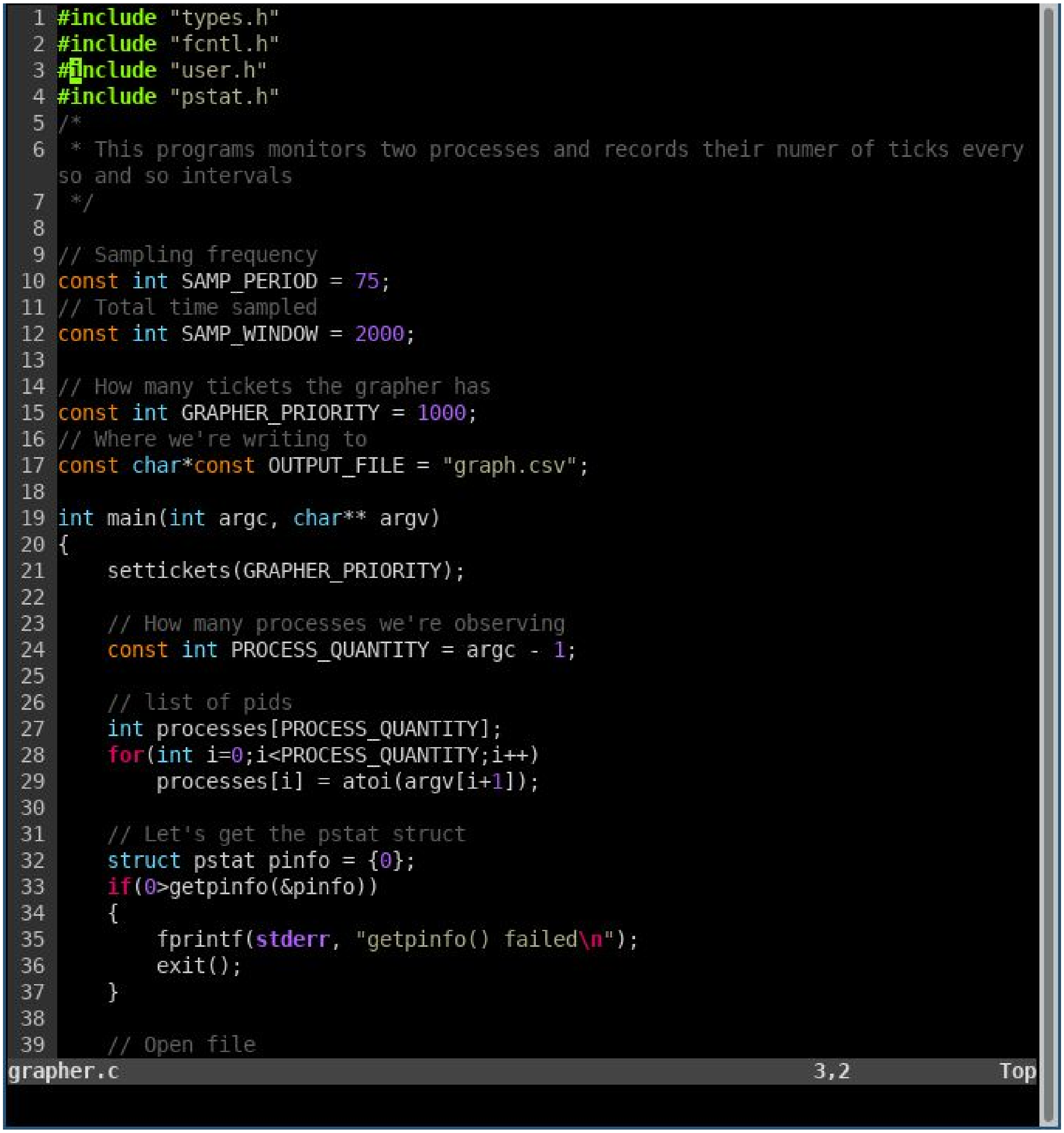
Bm.c



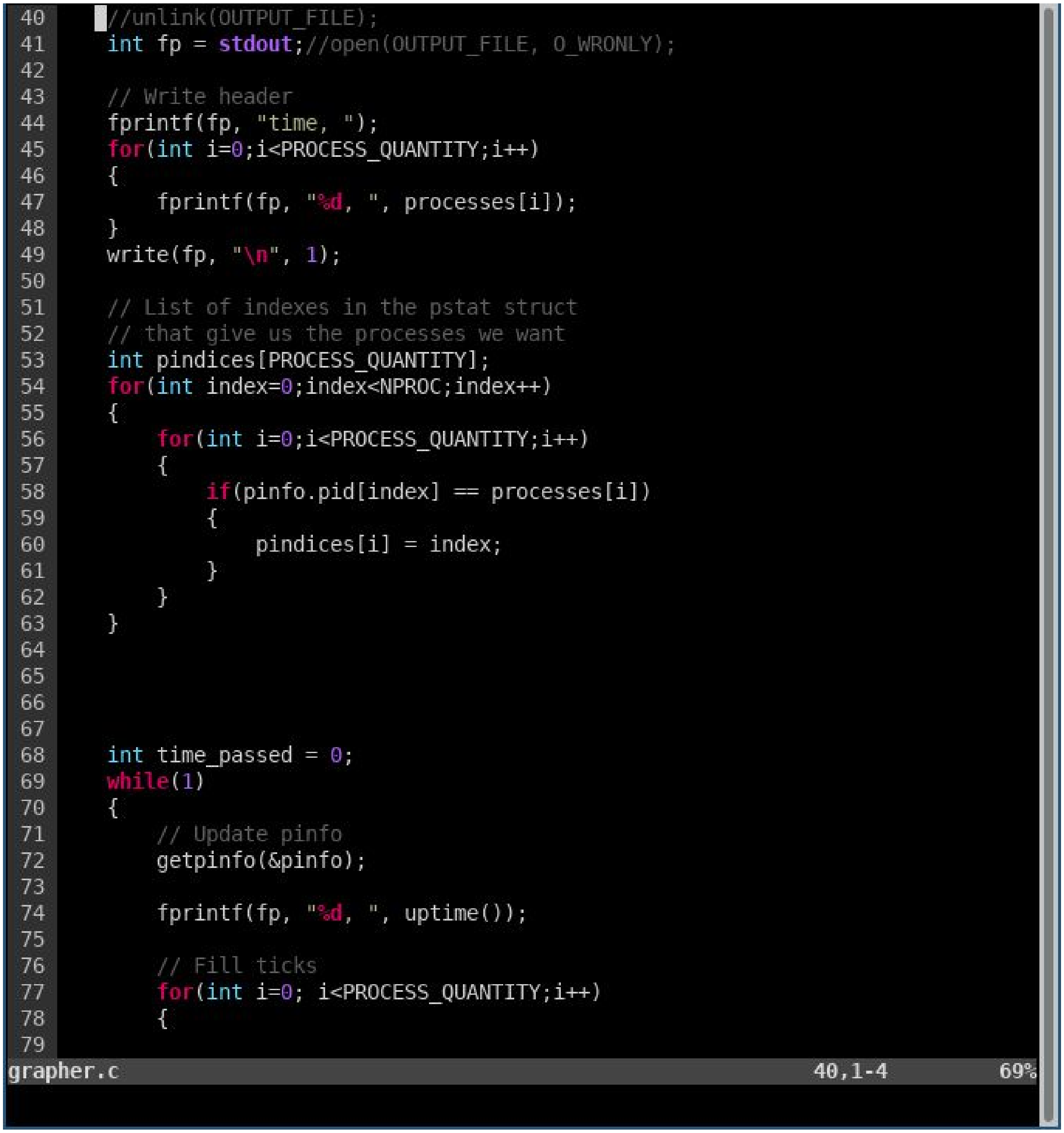
```c
/*
 * BM - Baby Maker
 * Makes an arbitrary amount of babies, letting them live for
 * eternity */
#include "types.h"
#include "user.h"
int main(int argc, char** argv)
{
    if(argc<2){
        fprintf(stdout, "Usage: bm child_1_tickets [child_2_tickets]...\n");
        exit();
    }
    fprintf(stdout, "Mother %d created\n", getpid());

    for(int i=1;i<argc;i++){
        const int pid = fork();
        if(pid<0){
            fprintf(stderr, "Stillbirth Occurred");
            exit();
        }
        if(!pid)
        {
            const int t = atoi(argv[i]);//number of tickets
            settickets(t);
            fprintf(stdout, "Child %d created with %d tickets\n", getpid(), t);
            // Loop forever
            while(1);
            fprintf(stdout, "Child %d exiting\n", getpid());
            exit();
        }
    }
    for(int i=1;i<argc;i++){
        wait();
    }
    fprintf(stdout, "Parent exiting\n");
    exit();
}
```

This program creates an arbitrary number of child processes of a specified number of tickets

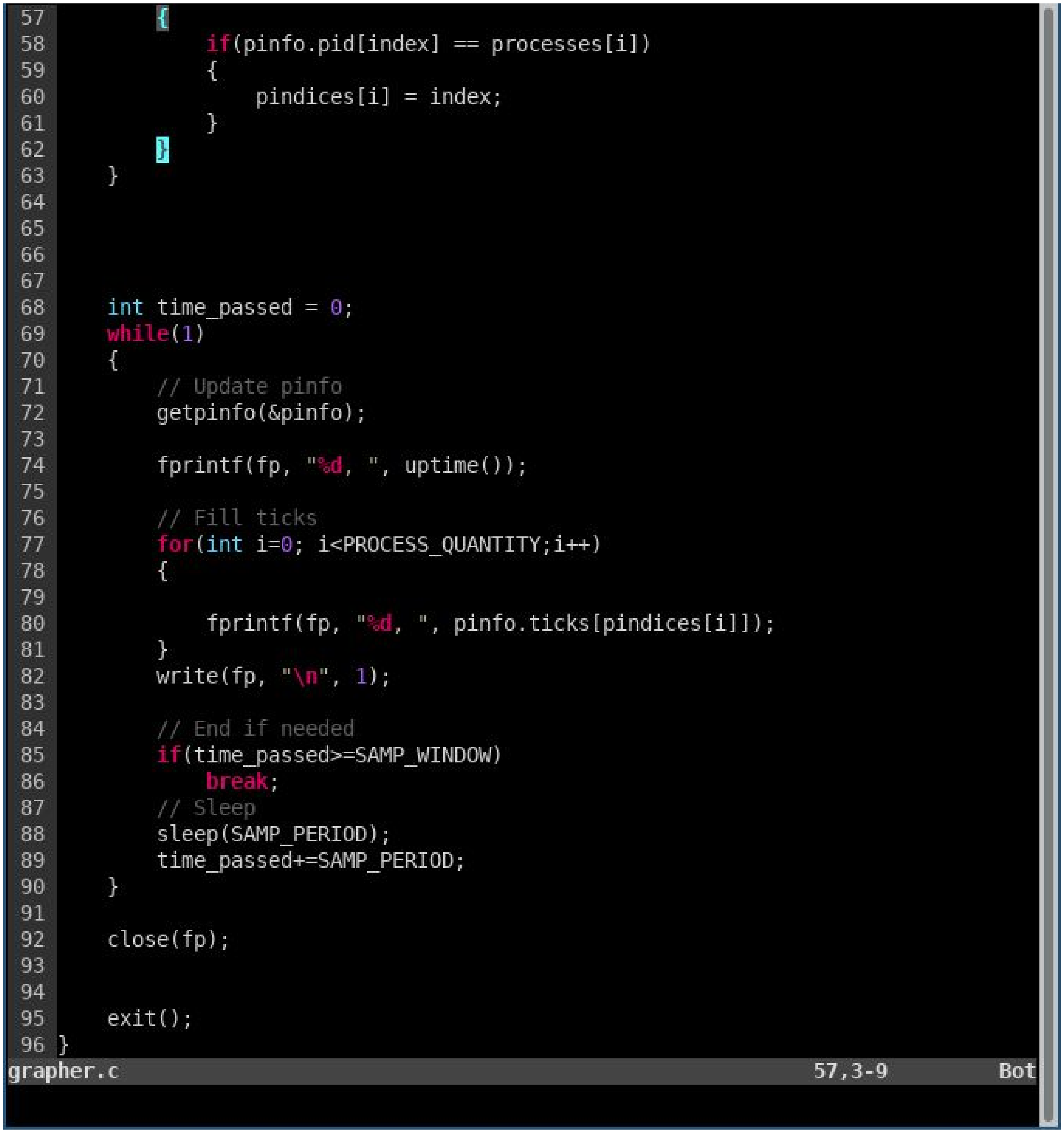
Grapher.c



```c
#include "types.h"
#include "fcntl.h"
#include "user.h"
#include "pstat.h"
/*
 * This programs monitors two processes and records their numer of ticks every so and so intervals
 */

// Sampling frequency
const int SAMP_PERIOD = 75;
// Total time sampled
const int SAMP_WINDOW = 2000;

// How many tickets the grapher has
const int GRAPHER_PRIORITY = 1000;
// Where we're writing to
const char*const OUTPUT_FILE = "graph.csv";

int main(int argc, char** argv)
{
    settickets(GRAPHER_PRIORITY);

    // How many processes we're observing
    const int PROCESS_QUANTITY = argc - 1;

    // list of pids
    int processes[PROCESS_QUANTITY];
    for(int i=0;i<PROCESS_QUANTITY;i++)
        processes[i] = atoi(argv[i+1]);

    // Let's get the pstat struct
    struct pstat pinfo = {0};
    if(0>getpinfo(&pinfo))
    {
        fprintf(stderr, "getpinfo() failed\n");
        exit();
    }

    // Open file
```



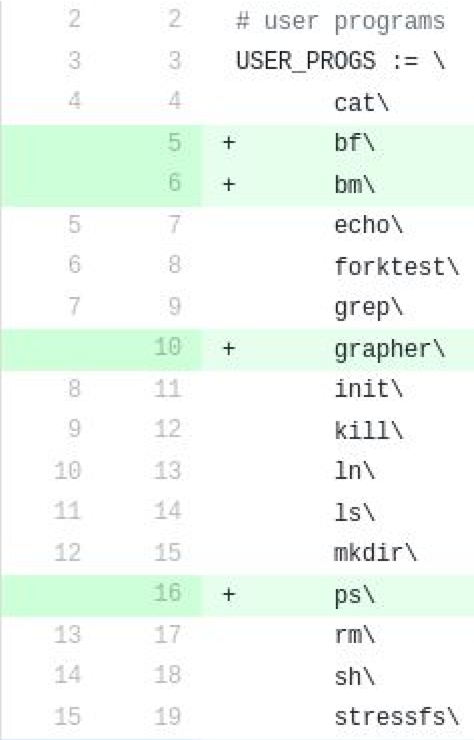
```
    //unlink(OUTPUT_FILE);
    int fp = stdout;//open(OUTPUT_FILE, O_WRONLY);

    // Write header
    fprintf(fp, "time, ");
    for(int i=0;i<PROCESS_QUANTITY;i++)
    {
        fprintf(fp, "%d, ", processes[i]);
    }
    write(fp, "\n", 1);

    // List of indexes in the pstat struct
    // that give us the processes we want
    int pindices[PROCESS_QUANTITY];
    for(int index=0;index<NPROC;index++)
    {
        for(int i=0;i<PROCESS_QUANTITY;i++)
        {
            if(pinfo.pid[index] == processes[i])
            {
                pindices[i] = index;
            }
        }
    }




    int time_passed = 0;
    while(1)
    {
        // Update pinfo
        getpinfo(&pinfo);

        fprintf(fp, "%d, ", uptime());

        // Fill ticks
        for(int i=0; i<PROCESS_QUANTITY;i++)
        {

grapher.c                                                   40,1-4        69%
```



```
        {
            if(pinfo.pid[index] == processes[i])
            {
                pindices[i] = index;
            }
        }
    }



    int time_passed = 0;
    while(1)
    {
        // Update pinfo
        getpinfo(&pinfo);

        fprintf(fp, "%d, ", uptime());

        // Fill ticks
        for(int i=0; i<PROCESS_QUANTITY;i++)
        {

            fprintf(fp, "%d, ", pinfo.ticks[pindices[i]]);
        }
        write(fp, "\n", 1);

        // End if needed
        if(time_passed>=SAMP_WINDOW)
            break;
        // Sleep
        sleep(SAMP_PERIOD);
        time_passed+=SAMP_PERIOD;
    }

    close(fp);


    exit();
}
```

This is a program that creates a graph plot in the form of a CSV file which is then fed into Octave to create the graph proper.

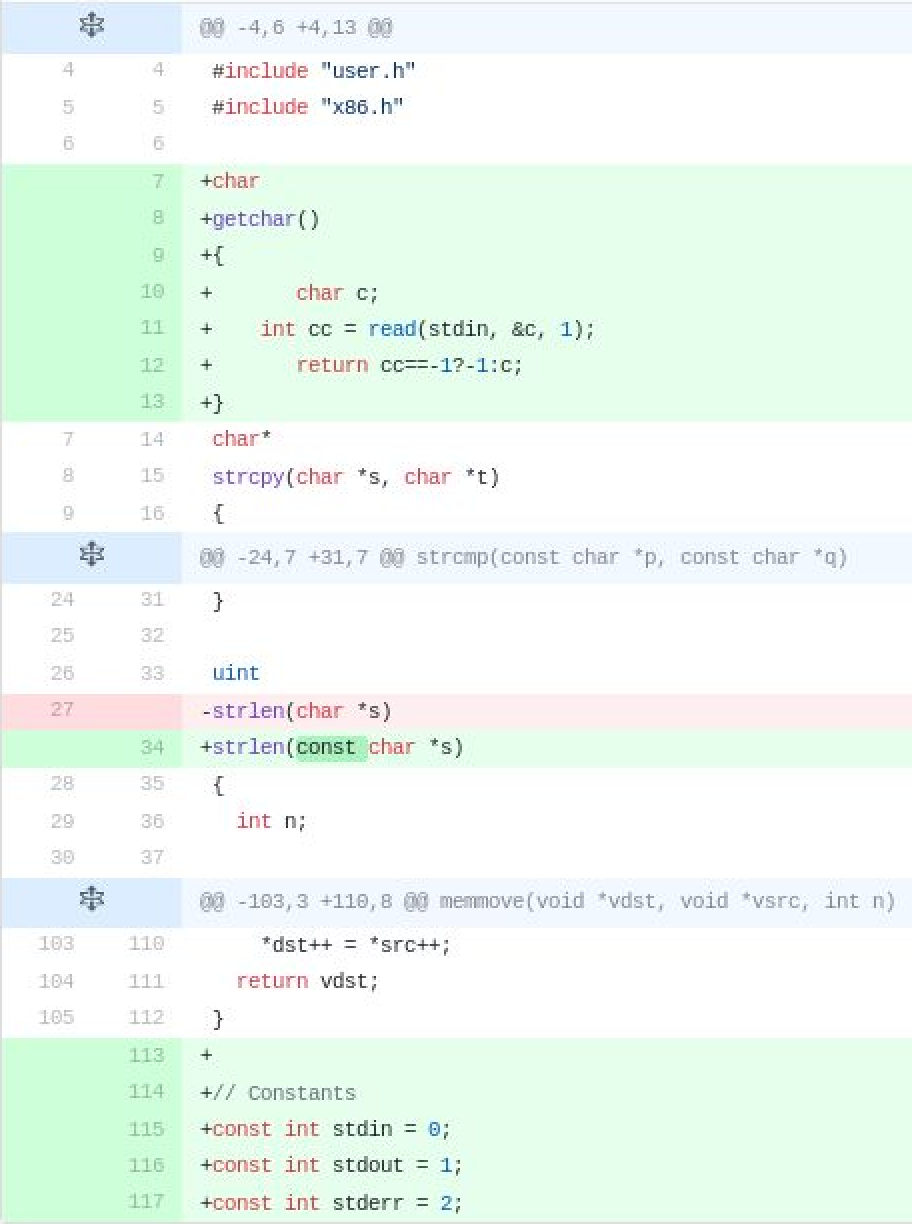
Makefile.mk



```
 # user programs
 USER_PROGS := \
        cat\
+       bf\
+       bm\
        echo\
        forktest\
        grep\
+       grapher\
        init\
        kill\
        ln\
        ls\
        mkdir\
+       ps\
        rm\
        sh\
        stressfs\
```

Included here are the three programs I mentioned, plus a Brainf*ck interpreter (for fun)

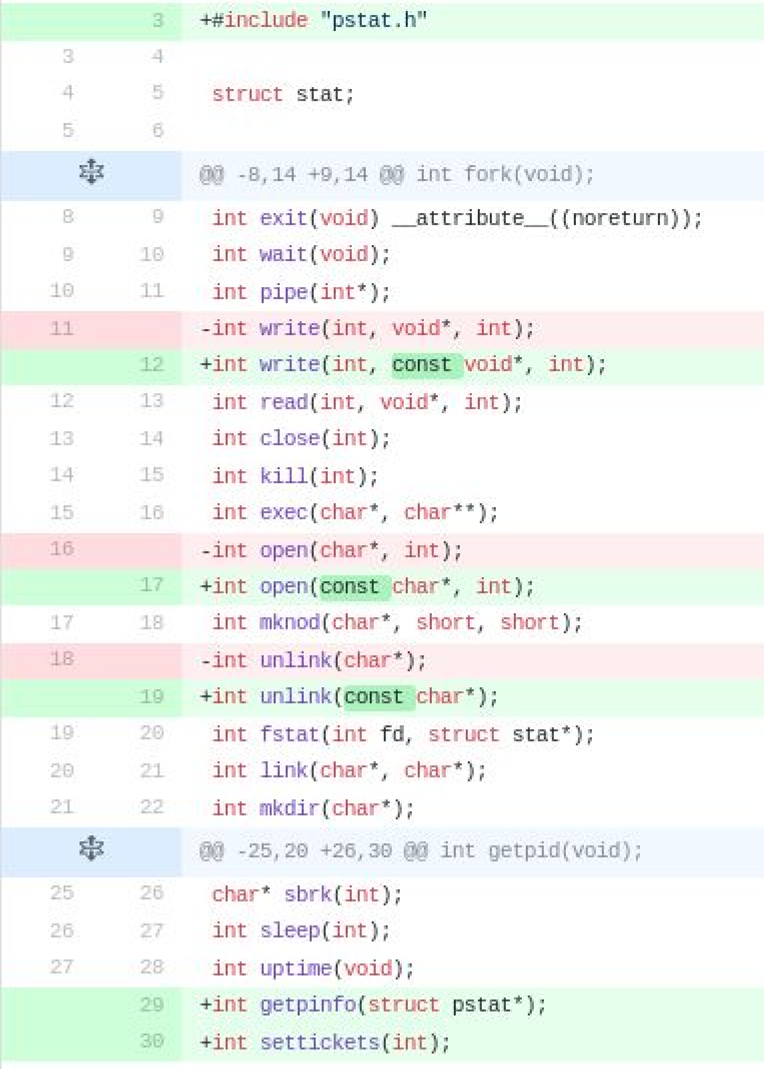
Ulib.c



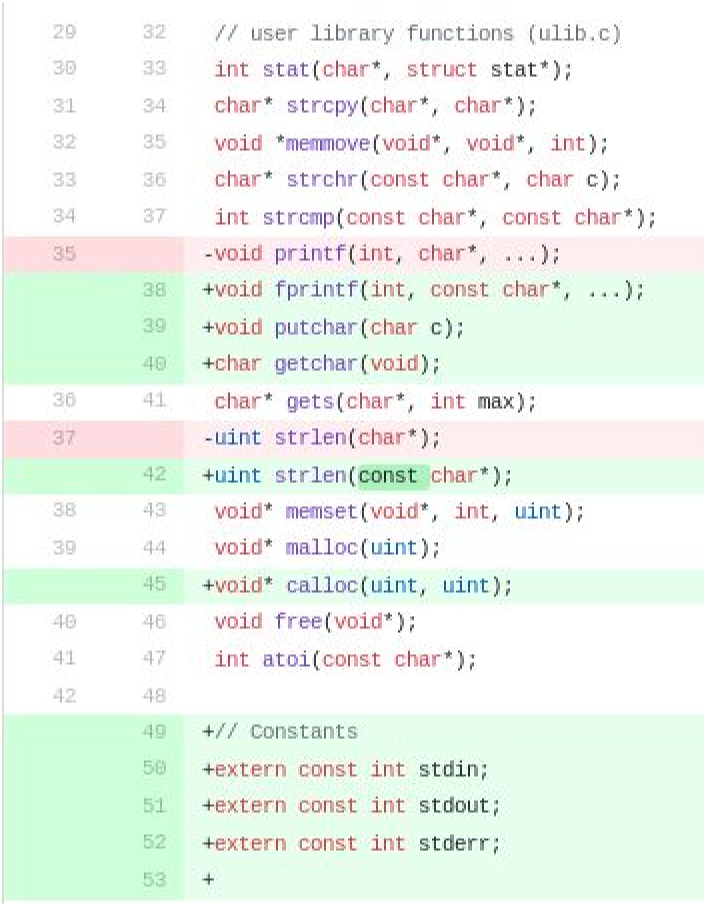
```
@@ -4,6 +4,13 @@
 #include "user.h"
 #include "x86.h"
 
+char
+getchar()
+{
+       char c;
+    int cc = read(stdin, &c, 1);
+       return cc==-1?-1:c;
+}
 char*
 strcpy(char *s, char *t)
 {
@@ -24,7 +31,7 @@ strcmp(const char *p, const char *q)
 }
 
 uint
-strlen(char *s)
+strlen(const char *s)
 {
   int n;
 
@@ -103,3 +110,8 @@ memmove(void *vdst, void *vsrc, int n)
     *dst++ = *src++;
   return vdst;
 }
+
+// Constants
+const int stdin = 0;
+const int stdout = 1;
+const int stderr = 2;
```

Added the function “getchar” and made a few function accept constant pointers

User.h



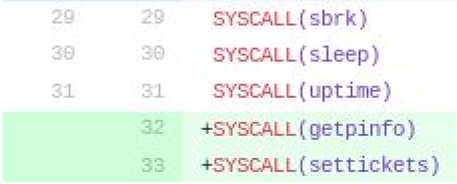
```
+#include "pstat.h"
 
 struct stat;
 
@@ -8,14 +9,14 @@ int fork(void);
 int exit(void) __attribute__((noreturn));
 int wait(void);
 int pipe(int*);
-int write(int, void*, int);
+int write(int, const void*, int);
 int read(int, void*, int);
 int close(int);
 int kill(int);
 int exec(char*, char**);
-int open(char*, int);
+int open(const char*, int);
 int mknod(char*, short, short);
-int unlink(char*);
+int unlink(const char*);
 int fstat(int fd, struct stat*);
 int link(char*, char*);
 int mkdir(char*);
@@ -25,20 +26,30 @@ int getpid(void);
 char* sbrk(int);
 int sleep(int);
 int uptime(void);
+int getpinfo(struct pstat*);
+int settickets(int);
```



```
 // user library functions (ulib.c)
 int stat(char*, struct stat*);
 char* strcpy(char*, char*);
 void *memmove(void*, void*, int);
 char* strchr(const char*, char c);
 int strcmp(const char*, const char*);
-void printf(int, char*, ...);
+void fprintf(int, const char*, ...);
+void putchar(char c);
+char getchar(void);
 char* gets(char*, int max);
-uint strlen(char*);
+uint strlen(const char*);
 void* memset(void*, int, uint);
 void* malloc(uint);
+void* calloc(uint, uint);
 void free(void*);
 int atoi(const char*);

+// Constants
+extern const int stdin;
+extern const int stdout;
+extern const int stderr;
+
```

Here, I added my two syscalls as well as some personalization

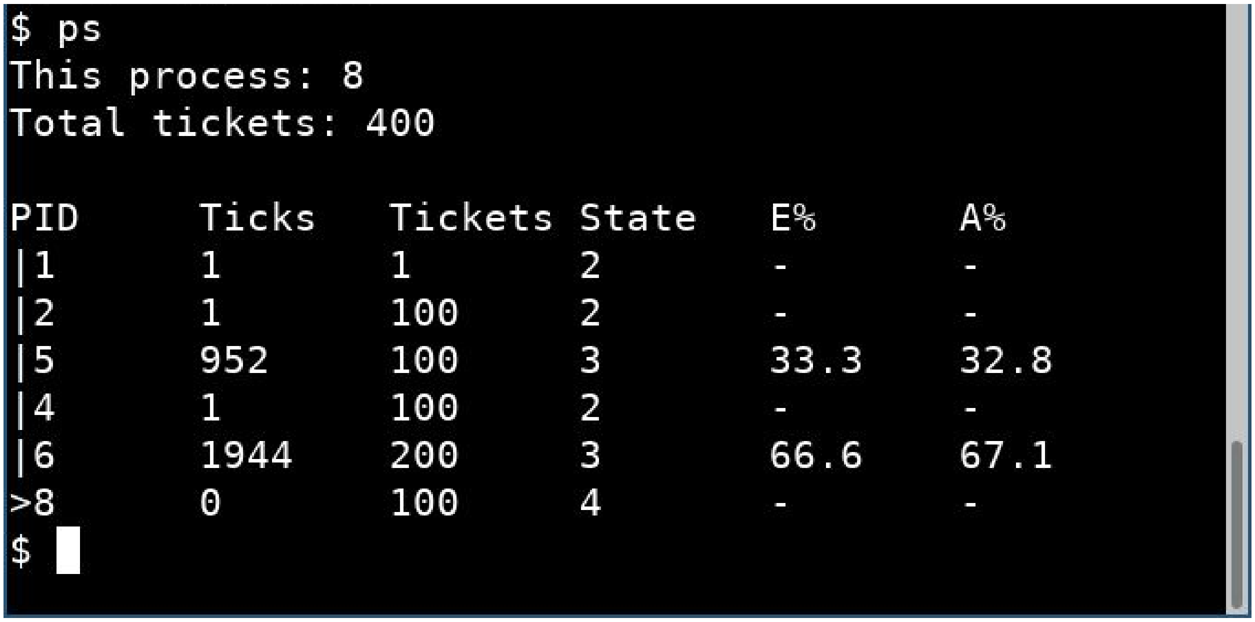
usys.S



```
 SYSCALL(sbrk)
 SYSCALL(sleep)
 SYSCALL(uptime)
+SYSCALL(getpinfo)
+SYSCALL(settickets)
```

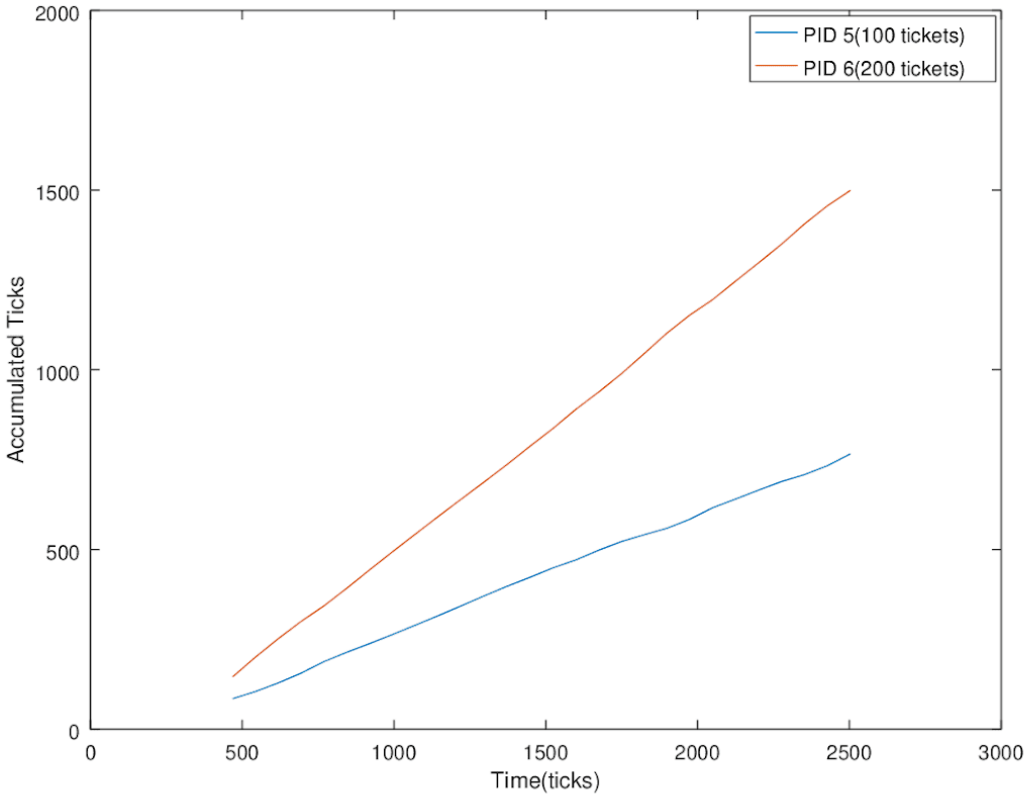
Added syscalls to assembly file

Results



```
$ ps
This process: 8
Total tickets: 400

PID     Ticks   Tickets State   E%      A%
|1      1       1       2       -       -
|2      1       100     2       -       -
|5      952     100     3       33.3    32.8
|4      1       100     2       -       -
|6      1944    200     3       66.6    67.1
>8      0       100     4       -       -
$ 
```

The result of running `ps` sometime after `bm 100 200`
Note that while the default for any process spawning from init(PID 1) is 1 ticket, but the default for any process spawning from the shell(PID 2) is 100, because the shell’s ticket is manually set in the user program `sh`.



This graph shows the accumulated ticks of two processes PID5 and PID6 with 100 and 200 tickets, respectively.