Lab 1: OS structure and Scheduler

-Professor Nael B. Abu-Ghazaleh

Name: Prince Choudhary
SID: 862254827

Part A: System call and xv6 basic structure

Implementation

1) To count the number of processes running in the system we iterate through table "ptable" and count the number of running processes.
2) To count the total number of system calls that the current process has made so far, we add a counter to the function syscall(void) in file syscall.c. Whenever a system call is made this function is called and counter is updated providing us with the number of system calls.
3) To calculate the number of memory pages the current process is using we can divide the process size "sz" by the "PGSIZE" (4096 bytes).

Code Changes:

1) **proc.c-** Here we implement the logic

Under allocproc(void) we initialize variable sys

```
//  Prince Lab 1
p->sysCallsCount = 0;
```

Created new function info(int param)

```
//Lab 1 Part A
int
info(int param)
{
    struct proc *curproc = myproc();
    if (param == 1) {
        struct  proc *p;
        int count = 0;
        acquire(&ptable.lock);
        for(p = ptable.proc; p<&ptable.proc[NPROC]; p++){
            if(p->state != ZOMBIE)
                count++;
        }
        release(&ptable.lock);
        return count;
    }
    else if (param == 2) {
        return curproc->sysCallsCount;
    }
    else if (param == 3){
        int size = 0;
        if((curproc->sz) % PGSIZE == 0 )
            size = (curproc->sz)/PGSIZE;
        else
            size = ((curproc->sz)/PGSIZE) +1;
        return size;
    }
return 0;
}
```

2) **syscall.c**

In function syscall(void) we add counter for system calls

```
curproc->sysCallsCount++;
```

We also declare system call “info” and add to syscalls array here

3) **syscall.h**

We add system call number here.

4) **sysproc.c**

Function prototype is added here with argument check.

```
sys_info(void)
{
    int p;
    if (argint(0, &p) < 0)
        return -1;
    return info(p);
}
```

5) **proc.h**

We add variable to count syscall to structure proc

```
int sysCallsCount;
```

6) **usys.S**

An interface for the system call is defined here

7) **defs.h**

Add our info function under proc.c list

8) **user.h**

The function called by the user is defined here.

9) **info.c**

New file is created to run the application

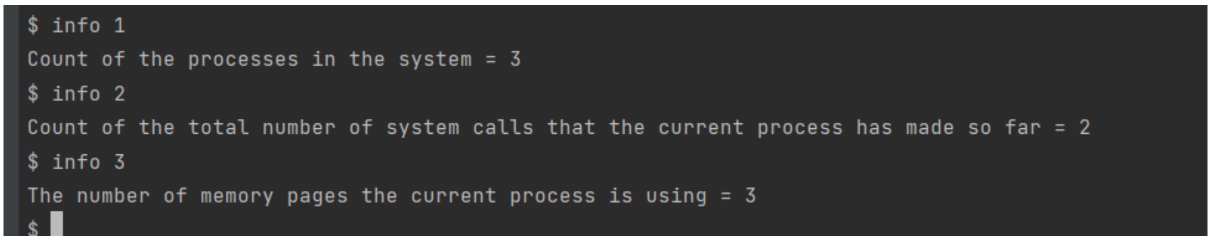
```
#include "types.h"
#include "stat.h"
#include "user.h"

int main(int argc, char *argv[]){
    int c = argv[1][0] - '0';
        switch(c){
    case 1:
        printf(1, "Count of the processes in the system = %d\n", info(1));
        break;
    case 2:
        printf(1, "Count of the total number of system calls that the
current process has made so far = %d\n", info(2));
        break;
    case 3:
        printf(1, "The number of memory pages the current process is using
= %d\n", info(3));
        break;
        default:
        printf(1,"Wrong choice, select 1,2 or 3");
        break;
    }
        exit();
}
```

10) **Makefile**

Entry to newly created file is added here.

Output:

We start by command: make qemu-nox
We call our application info passing argument at the same time to get the result.



```
$ info 1
Count of the processes in the system = 3
$ info 2
Count of the total number of system calls that the current process has made so far = 2
$ info 3
The number of memory pages the current process is using = 3
$ 
```

Part B: Scheduling

In this section we are implementing lottery and stride schedulers. In lottery scheduler tickets are given to processes and through random function (lottery) a winning ticket is selected. The process having this winning ticket gets the resource next. After the process consumes the allocated resource, it again goes back to runnable state and is allotted ticket(s) and this continues. In stride scheduling the processes are given strides which is inversely proportional to their chances of acquiring the resource. More tickets are provided to processes implies that their chances of getting the resource increases.

Most of the code for the two schedulers are common (i.e. they share the same code). However, the main scheduler logic which is present in proc.c differs. To run a particular scheduler, we need to uncomment that scheduler function “scheduler(void)” and the other scheduler should be commented. (appropriate comment is provided to recognize them)

Common code:

syscall.h

```
#define SYS_settickets 23
#define SYS_getticks 24
```

Syscall.c

```
extern int sys_settickets(void);
extern int sys_getticks(void);
```
and

```
[SYS_settickets]  sys_settickets,
[SYS_getticks] sys_getticks,
```

proc.c

settickets is used to allot desired number of tickets to processes.

```
int
settickets(int num)
{
    struct proc *p = myproc();
    acquire(&ptable.lock);
    p->tickets = num;
    p->pass = 10000/(p->tickets); //useful when stride scheduler is running
    release(&ptable.lock);
    return p->tickets;
}
```

under allocproc(void)

```
p->tickets = 5;
p->ticks = 0;
p->stride = 0;
```

defs.h

under proc.c list we add

```
int                settickets(int);
int                getticks(void);
```

usys.S

```
SYSCALL(settickets)
SYSCALL(getticks)
```

sysproc.c

```
int
sys_getticks(void)
{
    return myproc()->ticks;
}
int
sys_settickets(void)
{
   int tickets;
    argint(0,&tickets);
    if(tickets < 0)
        return -1;
    settickets(tickets);
    return 0;
}
```

user.h

```
int settickets(int);
int getticks(void);
```

Test files prog1, prog2 and prog3 with settickets 30, 20 and 10 respectively

```
#include "types.h"
#include "stat.h"
#include "user.h"
int main(int argc, char *argv[])
{
    settickets(30); // write your own function here
    int i,k;
    const int loop=43000;
    for(i=0;i<loop;i++) {
        asm("nop"); //in order to prevent the compiler from optimizing the
for loop
        for(k=0;k<loop;k++) {
            asm("nop");
        }
    }
    printf(1, "prog1 %d\n", getticks());
    exit();
}
```

Makefile

Add files prog1, prog2 and prog3.

Lottery Scheduling

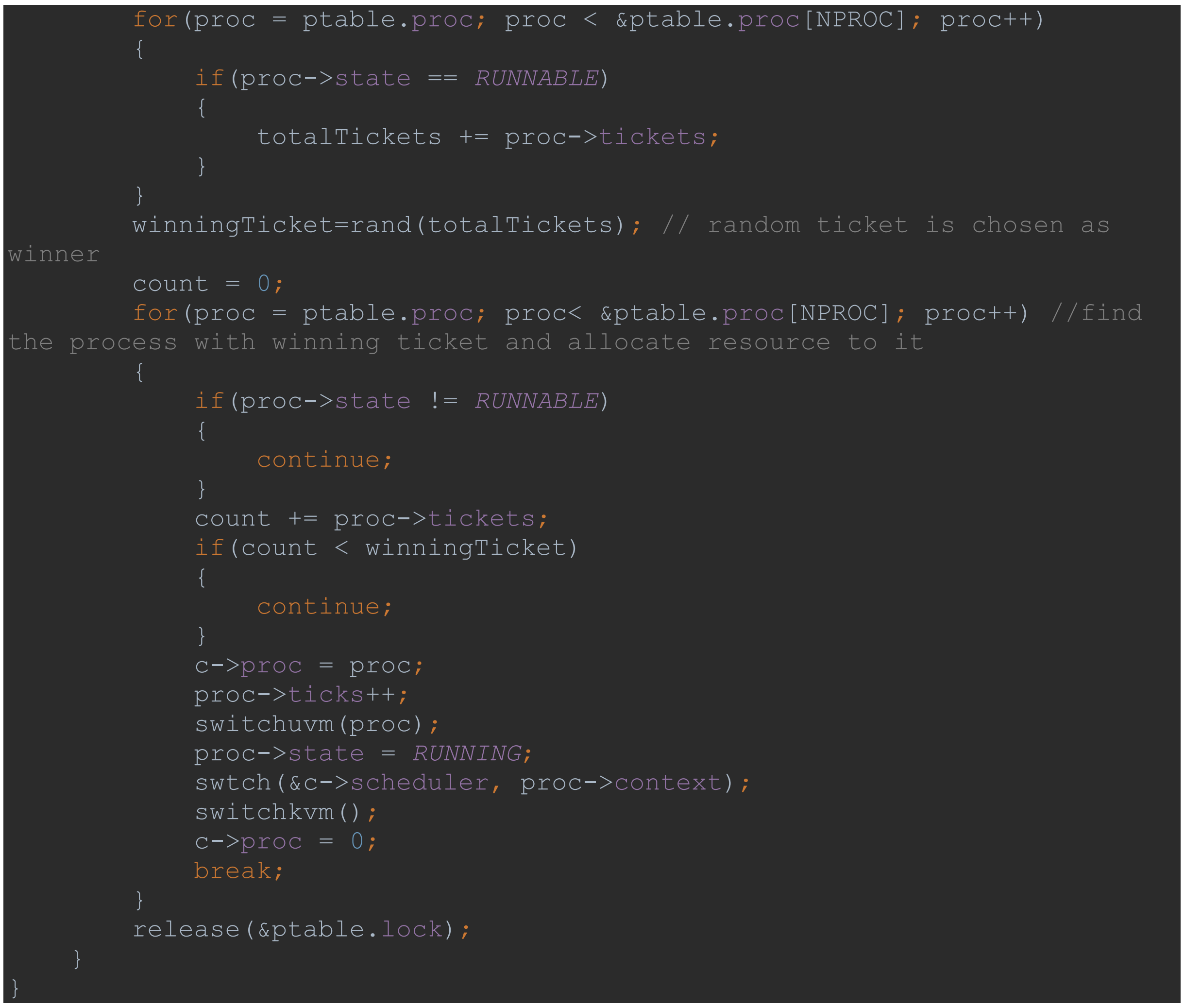
proc.c

Add rand() function, used to select winning ticket

```
int rand()
{
    return (rseed = (rseed * 214013 + 2531011) & RAND_MAX_32) >> 16;
}
```

Lottery scheduler implementation

```
void
scheduler(void)
{
    struct proc *proc;
    int totalTickets = 0;
    int winningTicket = 0;
    struct cpu *c = mycpu();
    c->proc = 0;
    int count = 0;
    for(;;){
        sti();
        acquire(&ptable.lock);
```



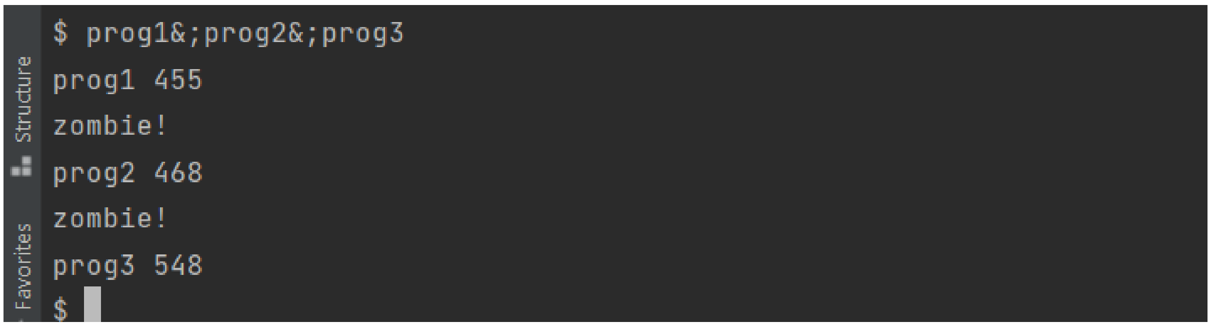
```
        for(proc = ptable.proc; proc < &ptable.proc[NPROC]; proc++)
        {
            if(proc->state == RUNNABLE)
            {
                totalTickets += proc->tickets;
            }
        }
        winningTicket=rand(totalTickets); // random ticket is chosen as
winner
        count = 0;
        for(proc = ptable.proc; proc< &ptable.proc[NPROC]; proc++) //find
the process with winning ticket and allocate resource to it
        {
            if(proc->state != RUNNABLE)
            {
                continue;
            }
            count += proc->tickets;
            if(count < winningTicket)
            {
                continue;
            }
            c->proc = proc;
            proc->ticks++;
            switchuvm(proc);
            proc->state = RUNNING;
            swtch(&c->scheduler, proc->context);
            switchkvm();
            c->proc = 0;
            break;
        }
        release(&ptable.lock);
    }
}
```

Execution and output

Start qemu using : make qemu-nox

Run test files: prog1&;prog2&;prog3

Output:



```
$ prog1&;prog2&;prog3
prog1 455
zombie!
prog2 468
zombie!
prog3 548
$ 
```

Stride Scheduler

proc.c

Stride scheduler implemetation

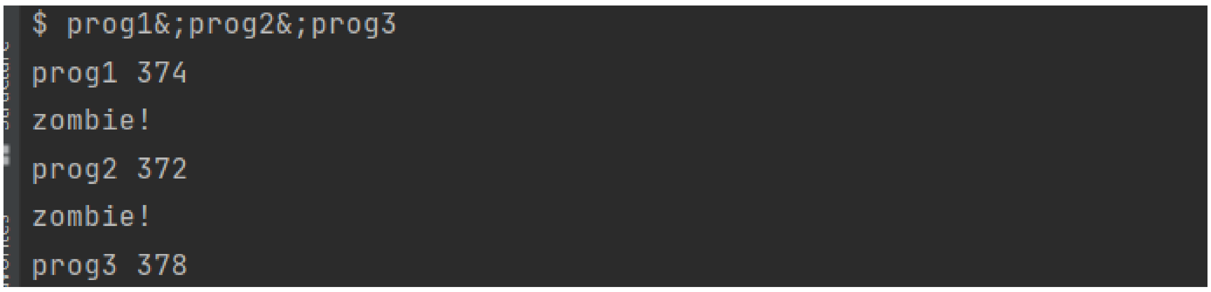
```
void
scheduler(void)
{
  struct proc *proc;
  struct cpu *c = mycpu();
  float minPassValue;
  c->proc = 0;
  struct proc *minproc = myproc();
  for (;;)
  {
    sti();
    acquire(&ptable.lock);
    minPassValue = 9999999999;
    for (proc = ptable.proc; proc < &ptable.proc[NPROC]; proc++) //find min
stride
    {
      if (proc->state != RUNNABLE)
      {
            continue;
    }
    if (proc->stride < minPassValue)
    {
        minPassValue = proc->stride;
       minproc = proc;
    }
  }
      c->proc = minproc;
      minproc->stride += minproc->pass;     //increment the stride value so
as to allow other processes to get resource
      minproc->ticks++;
      switchuvm(minproc);
      minproc->state = RUNNING;
      swtch(&(c->scheduler),minproc->context);
      switchkvm();
      c->proc = 0;
   release(&ptable.lock);
 }
}
```

Execution and output

Start qemu using: make qemu-nox

Run test files: prog1&;prog2&;prog3

Output:



```
$ prog1&;prog2&;prog3
prog1 374
zombie!
prog2 372
zombie!
prog3 378
```

Graphical analysis and comparison between Lottery and Stride Scheduler

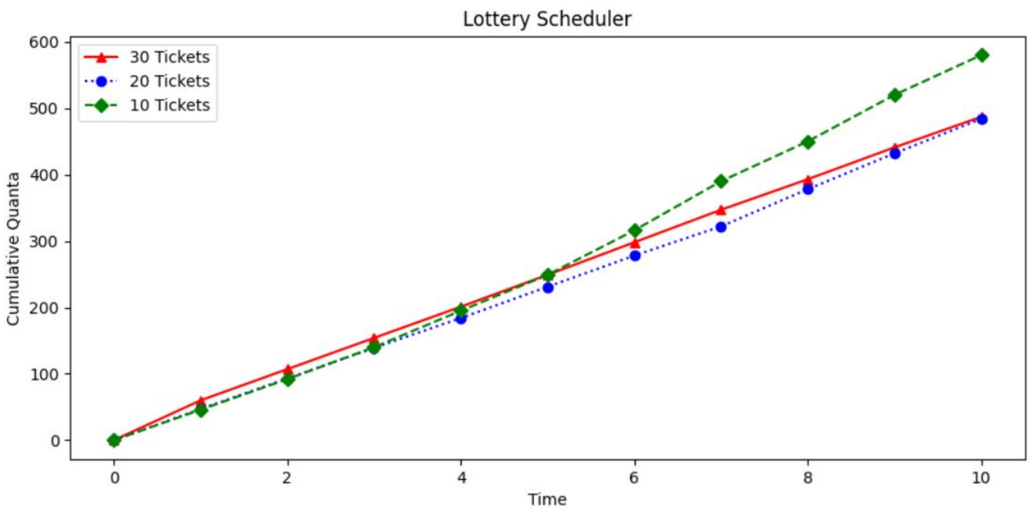
To plot graph, we need intermittent execution values, which we can easily obtain by adding a printf statement in our test file, for example, I added it after the outer loop so as to get exact 10 points for each process.

```
#include "types.h"
#include "stat.h"
#include "user.h"
int main(int argc, char *argv[])
{
    settickets(30); // write your own function here
    int i,k;
    const int loop=43000;
    for(i=0;i<loop;i++) {
        asm("nop"); //in order to prevent the compiler from optimizing the
for loop
        for(k=0;k<loop;k++) {
            asm("nop");
        }
        if(i%4300 == 0)
            printf(1, "prog1 %d %d\n",(i/4300), getticks());
    }
    printf(1, "prog1 %d\n", getticks());
    exit();
}
```

Once we get the values, we can plot the graph using the values on Cumulative Quanta Vs Time and Mean Error Vs Time

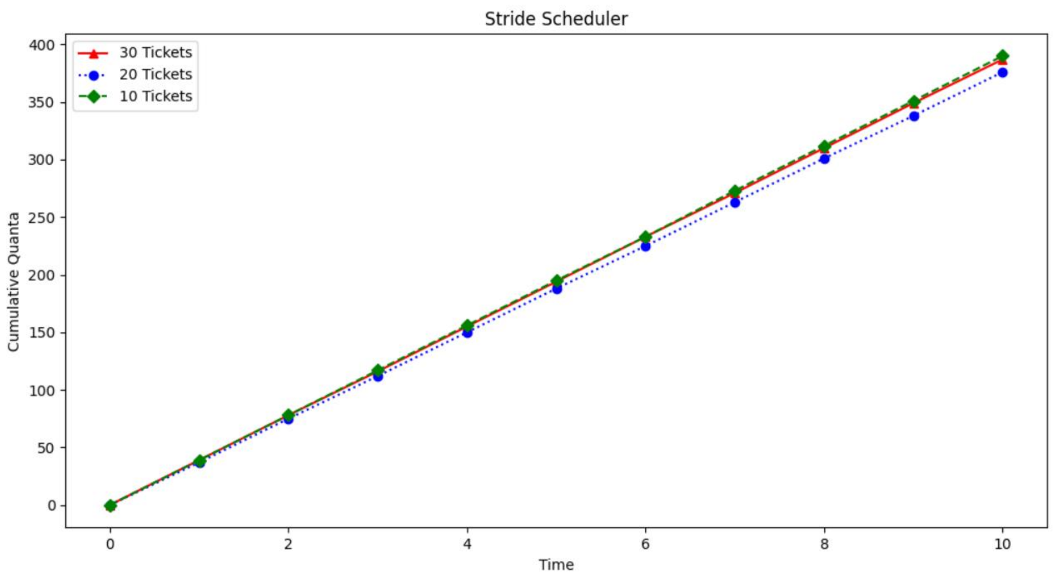
Though there are multiple ways to plot graph once we obtain the values, I used matplotlib in python to plot the graph.

```
import numpy as np
import matplotlib.pyplot as plt

x1 = np.array([0,1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
y1 = np.array([0,60,107,154,201,249,298,347,393,441,487])
y2 = np.array([0,47,93,139,184,231,278,322,378,432,484])
y3 = np.array([0,46,92,140,195,249,316,390,450,520,580])

plt.plot(x1, y1, marker = '^', color = 'r', ls = 'solid',label='30
Tickets')
plt.plot(x1, y2, marker = 'o', color = 'b', ls = 'dotted',label='20
Tickets')
plt.plot(x1, y3, marker = 'D', color = 'g', ls = 'dashed', label='10
Tickets')
plt.title("Lottery Scheduler")
plt.xlabel("Time")
plt.ylabel("Cumulative Quanta")
plt.legend()
plt.show()
```



Lottery scheduler data in tabular form

Iterations(Time)	Prog1(30 Tickets)	Prog2(20 Tickets)	Prog3(10 Tickets)
0	0	0	0
1	60	47	46
2	107	93	92
3	154	139	140
4	201	184	195
5	249	231	249
6	298	278	316
7	347	322	390
8	393	378	450
9	441	432	520
10	487	484	580



Stride scheduler data in tabular form

Iterations(Time)	Prog1(30 Tickets)	Prog2(20 Tickets)	Prog3(10 Tickets)
0	0	0	0
1	39	37	39
2	78	75	78
3	116	112	117
4	155	150	156
5	194	188	195
6	233	225	233
7	271	263	273
8	310	301	312
9	349	338	351
10	387	376	390