Operating System CS 202

Lab Assignment Report -II

Group Member: Madhusudhan Tungamitta(862325125), Akhilesh Reddy Gali(862258042), Srinivasa Biradavolu(862318458)

Summary:

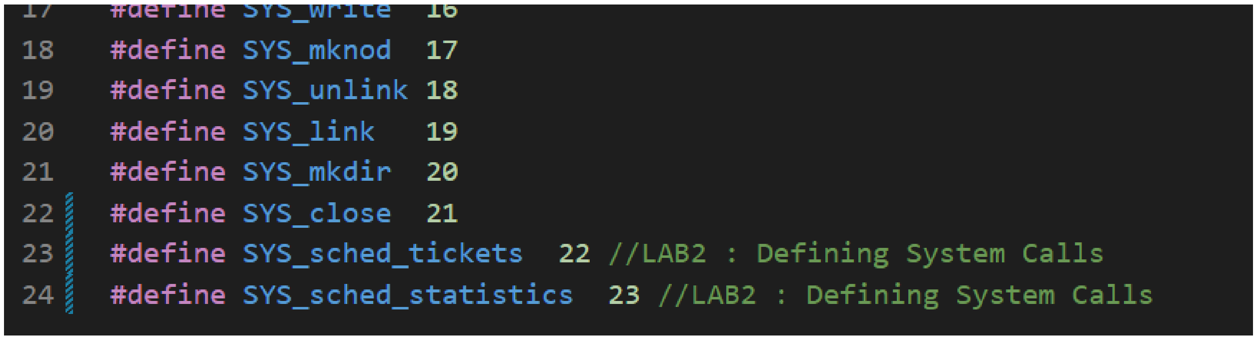
1. List of Files Modified:

We have created a new file “lab2.c” in the user folder and below are the files we made changes

Folder Name	Files
Kernel	syscall.h
Kernel	syscall.c
Kernel	sysproc.c
Kernel	proc.c
Kernel	proc.h
Kernel	defs.h
user	usys.pl
user	user.h
user	lab2.c
	MakeFile

Kernel\Syscall.h:

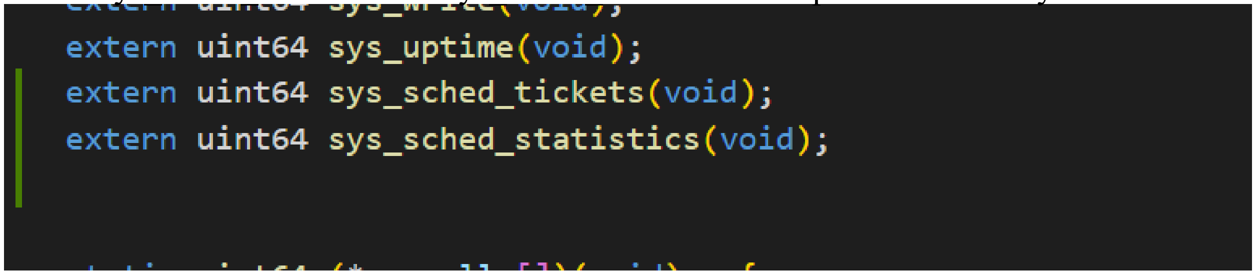
By adding sys_sched_tickets in line 23 and sys_sched_statistics in line 24 we are reserving a place for our own system call.



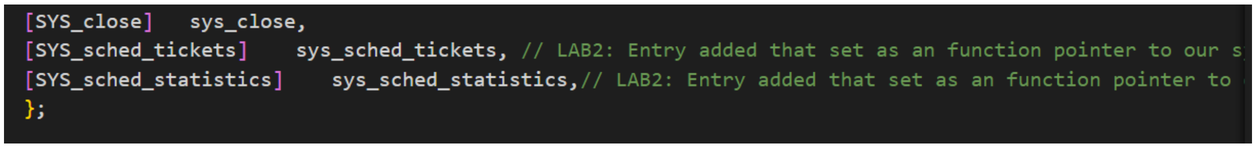
```
17   #define SYS_write  16
18   #define SYS_mknod  17
19   #define SYS_unlink 18
20   #define SYS_link   19
21   #define SYS_mkdir  20
22   #define SYS_close  21
23   #define SYS_sched_tickets  22 //LAB2 : Defining System Calls
24   #define SYS_sched_statistics  23 //LAB2 : Defining System Calls
```

Kernel\Syscall.c

We create an entry for the system calls "SYS_sched_statistics", "SYS_sched_tickets" and their functions "sys_sched_statistics", "sys_sched_tickets" respectively in the system call table. Here we are adding that acts as a pointer to the system call where this file contains an array of function pointers to the system calls which are defined in different locations, So the sys_sched_statistics and sys_sched_tickets act as a pointer for our system call



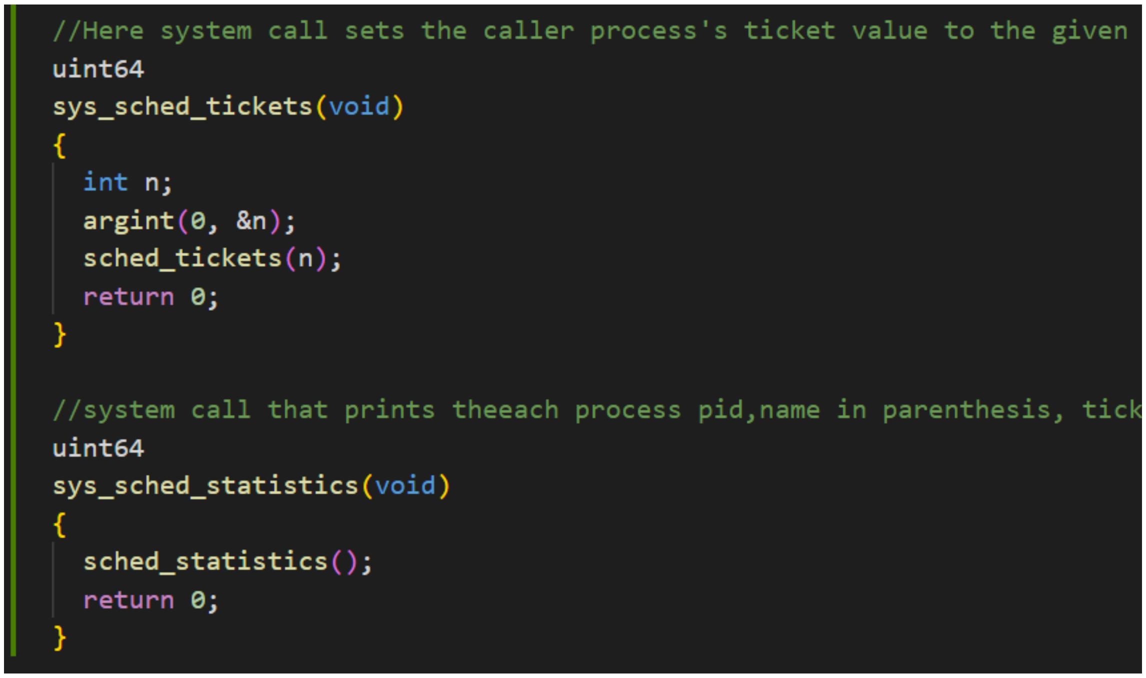
```
extern uint64 sys_uptime(void);
extern uint64 sys_sched_tickets(void);
extern uint64 sys_sched_statistics(void);
```



```
[SYS_close]    sys_close,
[SYS_sched_tickets]    sys_sched_tickets, // LAB2: Entry added that set as an function pointer to our s
[SYS_sched_statistics]    sys_sched_statistics,// LAB2: Entry added that set as an function pointer to
};
```

Kernel\Sysproc.c

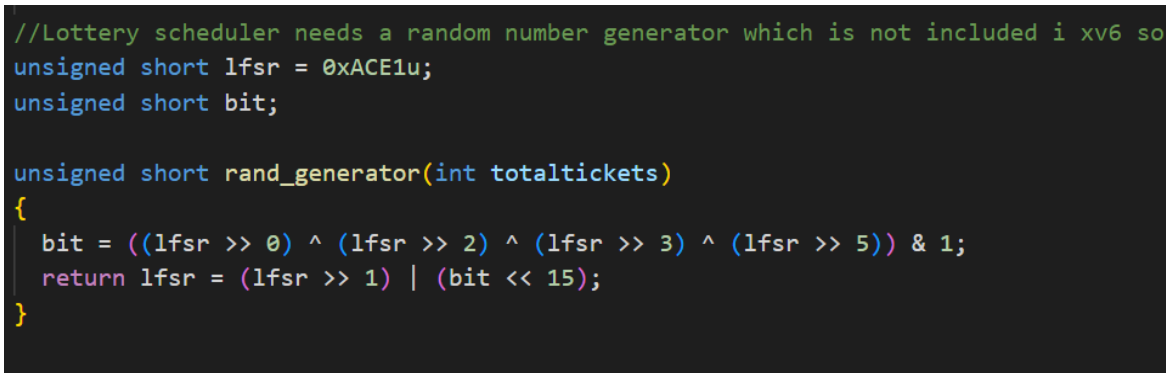
As we created new functions sys_sched_tickets and sys_sched_statistics in the system call table we defined previously in the sysproc.c file, we defined the sys_sched_tickets and sys_sched_statistics functions. The clear need for writing sysproc is that when the user wants to pass arguments to kernel-level functions from the user level function it is not directly possible in xv6.
//So Xv6 developers came up with their inbuilt function "argint"() that helps in passing //arguments from user to kernel level. Then that integer is passed into the print_info //function using argint().



```
//Here system call sets the caller process's ticket value to the given
uint64
sys_sched_tickets(void)
{
  int n;
  argint(0, &n);
  sched_tickets(n);
  return 0;
}

//system call that prints theeach process pid,name in parenthesis, tick
uint64
sys_sched_statistics(void)
{
  sched_statistics();
  return 0;
}
```

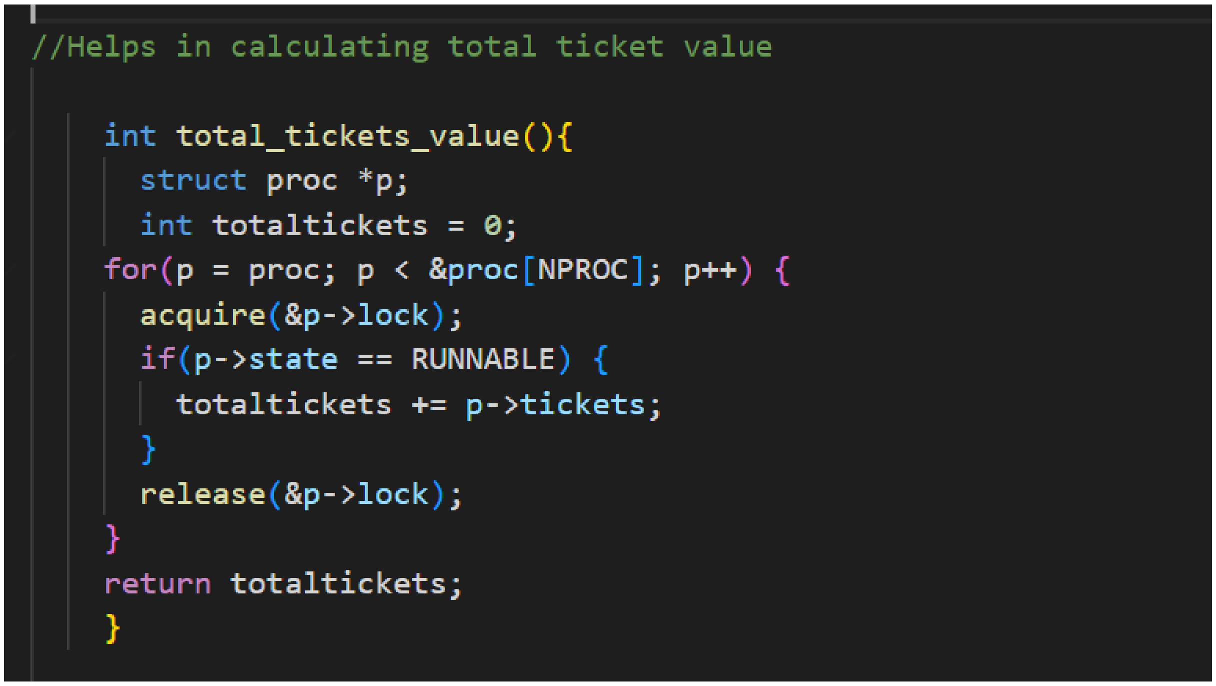
Kernel\Proc.c:

Lottery Scheduling:
Functions Used: Random Generator, Total Tickets.
Random Generator:
Lottery scheduling uses Random scheduling Hence we are using the code that is provided in the lab2 document for the generation of the random values and we pass the total number of the tickets to find the random values.


```
//Lottery scheduler needs a random number generator which is not included i xv6 so
unsigned short lfsr = 0xACE1u;
unsigned short bit;

unsigned short rand_generator(int totaltickets)
{
  bit = ((lfsr >> 0) ^ (lfsr >> 2) ^ (lfsr >> 3) ^ (lfsr >> 5)) & 1;
  return lfsr = (lfsr >> 1) | (bit << 15);
}
```

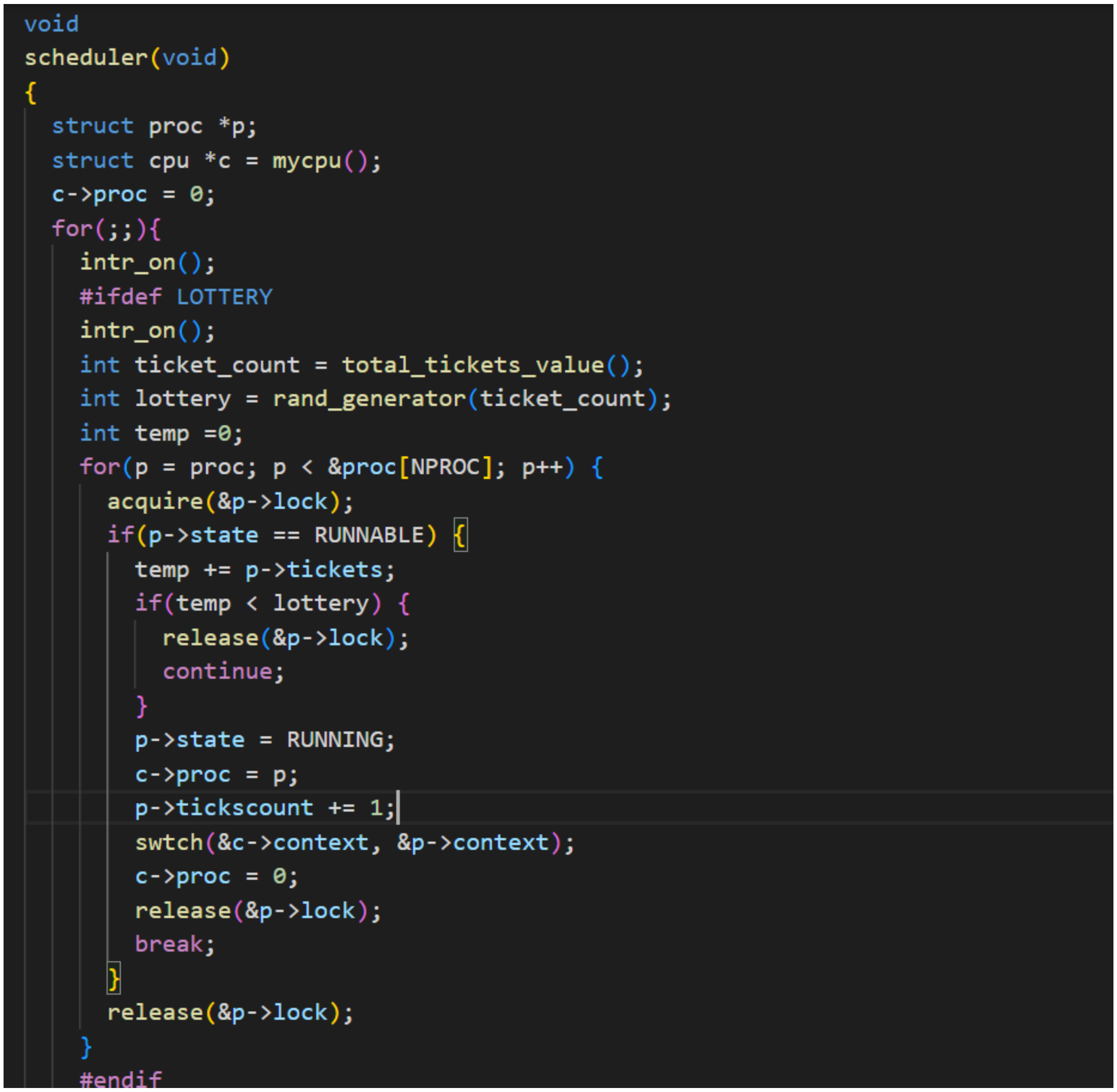
Counting Total Tickets:

This function helps in calculating the total number of tickets that can be used her we run a loop if the process is in running state we increment the total tickets counter and displays the results.


```
//Helps in calculating total ticket value

    int total_tickets_value(){
      struct proc *p;
      int totaltickets = 0;
    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        totaltickets += p->tickets;
      }
      release(&p->lock);
    }
    return totaltickets;
    }
```

Lottery scheduling Logic:

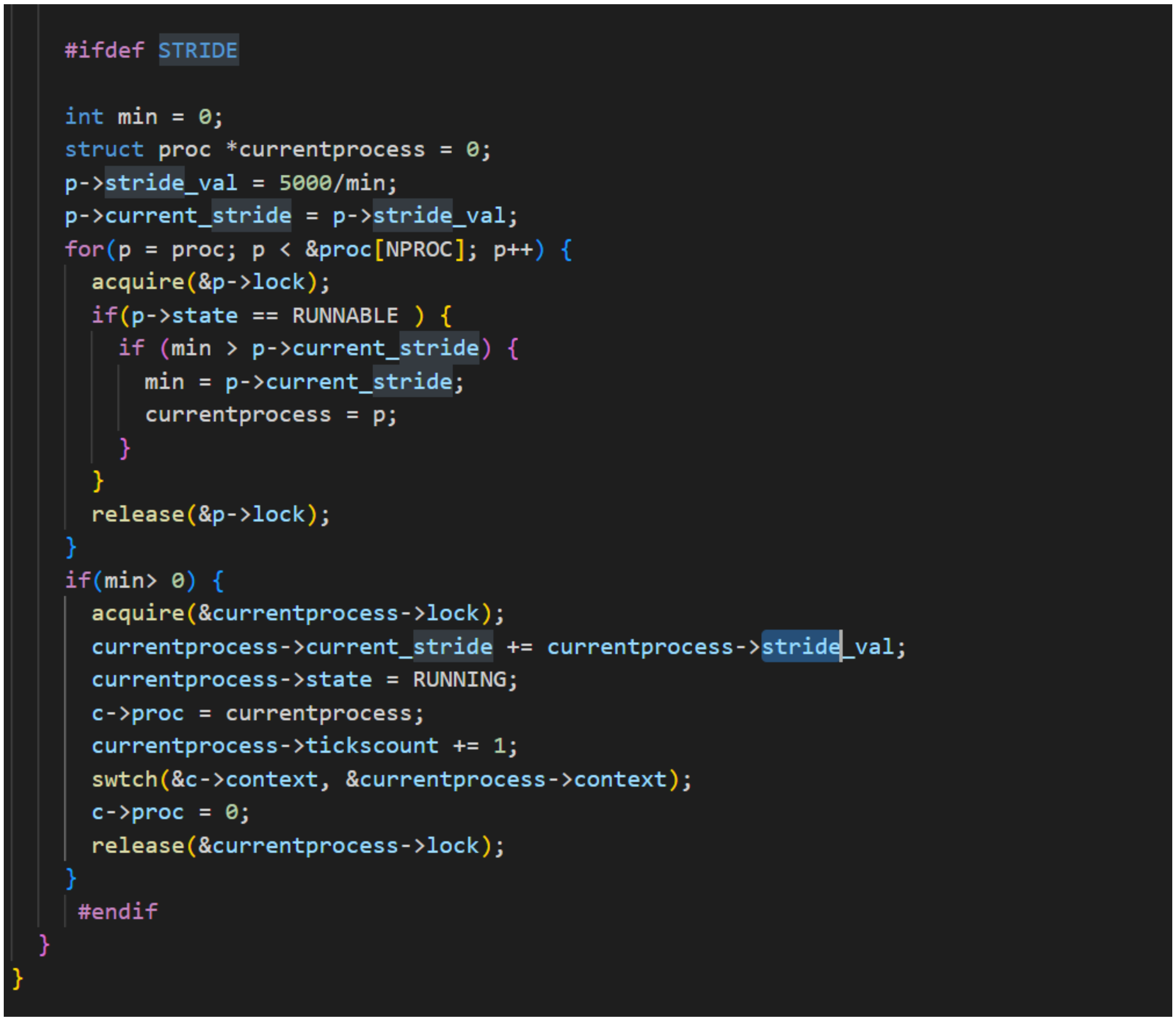
In Lottery scheduling, we are finding the lottery number that we use to schedule from the total number of tickets generated, we will check the current tickets value iteratively by sending in the for loop if the value of the lottery is less than the tickets than we will add a counter and keep the process in running states and this will helps us in finding the number of tickets it has been scheduled to run.



```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;
  for(;;){
    intr_on();
    #ifdef LOTTERY
    intr_on();
    int ticket_count = total_tickets_value();
    int lottery = rand_generator(ticket_count);
    int temp =0;
    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        temp += p->tickets;
        if(temp < lottery) {
          release(&p->lock);
          continue;
        }
        p->state = RUNNING;
        c->proc = p;
        p->tickscount += 1;
        swtch(&c->context, &p->context);
        c->proc = 0;
        release(&p->lock);
        break;
      }
      release(&p->lock);
    }
    #endif
```

Stride Scheduling:

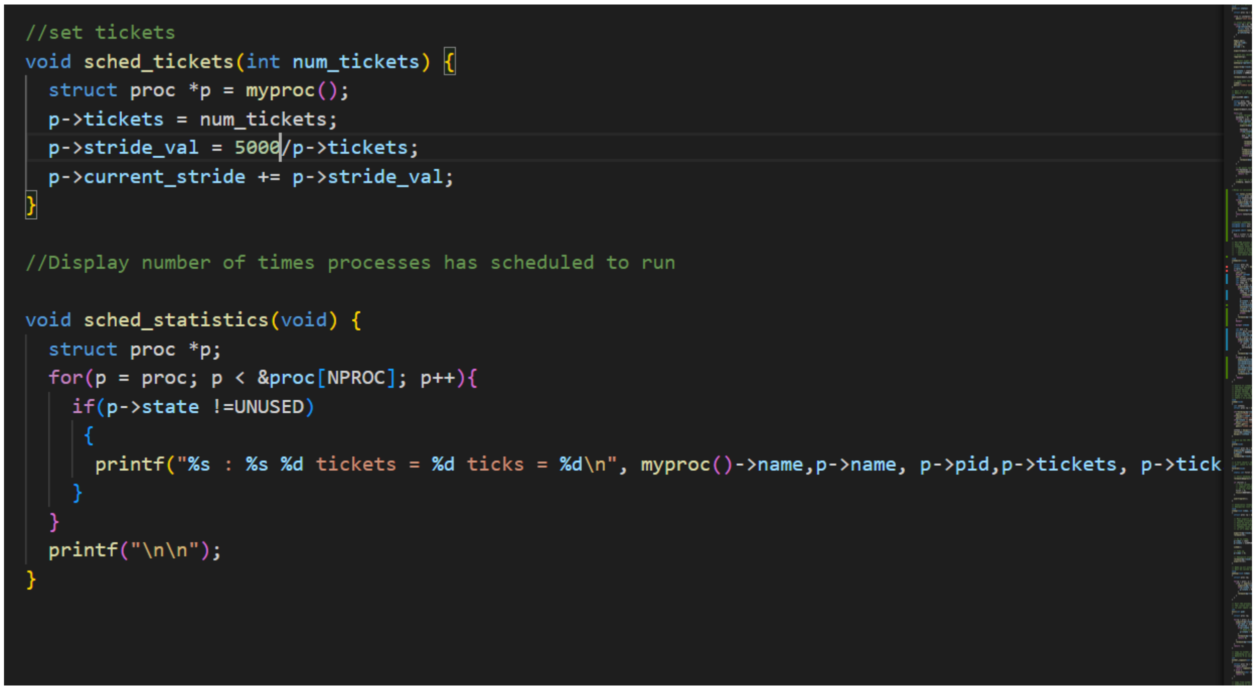
In stride scheduling the largest assigned value we are using is 5000 from the lab instructions and we are calculating the current stride value and also initialize the minimum value which will helps us for stride scheduling, By comparing the minimum value with current stride we are able to find which one to scheduled in further and if min is greater than zero we are incrementing the current stride value(Basically doubling the value) and it keeps the process in running state and we can increment the ticks count which helps in finding the no of times it has been scheduled to run



```
    #ifdef STRIDE

    int min = 0;
    struct proc *currentprocess = 0;
    p->stride_val = 5000/min;
    p->current_stride = p->stride_val;
    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE ) {
        if (min > p->current_stride) {
          min = p->current_stride;
          currentprocess = p;
        }
      }
      release(&p->lock);
    }
    if(min> 0) {
      acquire(&currentprocess->lock);
      currentprocess->current_stride += currentprocess->stride_val;
      currentprocess->state = RUNNING;
      c->proc = currentprocess;
      currentprocess->tickscount += 1;
      swtch(&c->context, &currentprocess->context);
      c->proc = 0;
      release(&currentprocess->lock);
    }
     #endif
  }
}
```

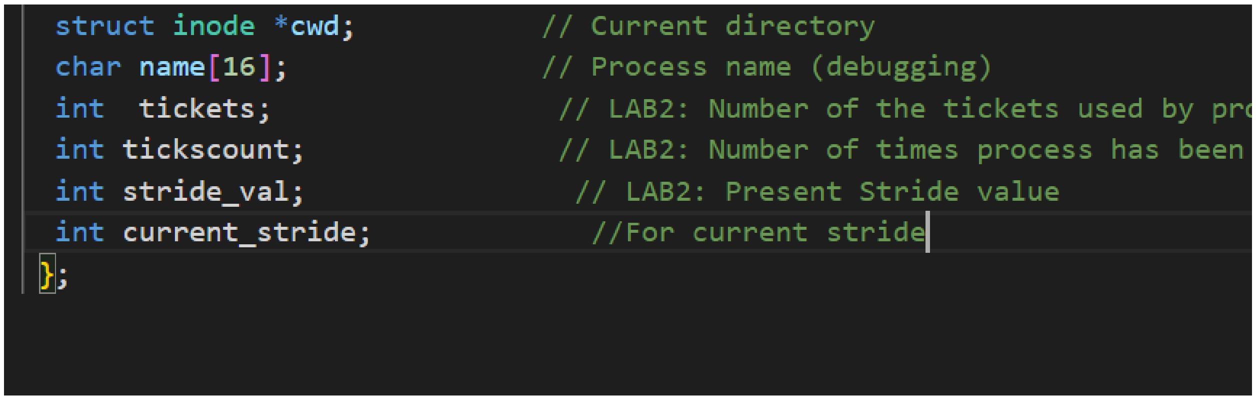
Sched_tickets in proc.c
Here sched tickets function sets as the caller value, When we give the input it acts as an entry for the caller process.

Sched statistics:
In this function, it prints the system call for the process, ticket value and number of times of it has been attempted to schedule.



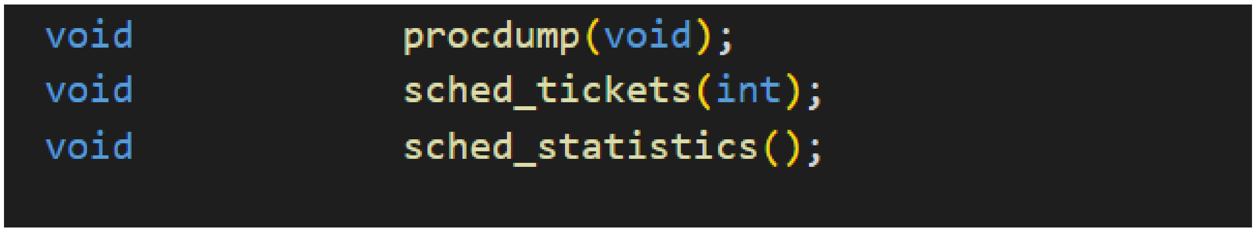
```
//set tickets
void sched_tickets(int num_tickets) {
  struct proc *p = myproc();
  p->tickets = num_tickets;
  p->stride_val = 5000/p->tickets;
  p->current_stride += p->stride_val;
}

//Display number of times processes has scheduled to run

void sched_statistics(void) {
  struct proc *p;
  for(p = proc; p < &proc[NPROC]; p++){
    if(p->state !=UNUSED)
     {
      printf("%s : %s %d tickets = %d ticks = %d\n", myproc()->name,p->name, p->pid,p->tickets, p->tick
    }
  }
  printf("\n\n");
}
```

<u>Kernel\proc.h:</u>
This is the header files for the process and here we are declaring the counter variable of type integer which is used to track the number of system calls that a current process has made so far.



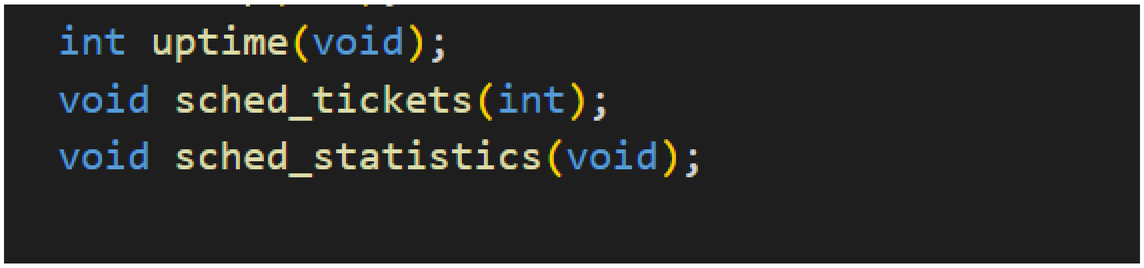
```
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int  tickets;                 // LAB2: Number of the tickets used by pro
  int tickscount;               // LAB2: Number of times process has been
  int stride_val;                // LAB2: Present Stride value
  int current_stride;             //For current stride
};
```

Kernel\defs.h



```
void            procdump(void);
void            sched_tickets(int);
void            sched_statistics();
```

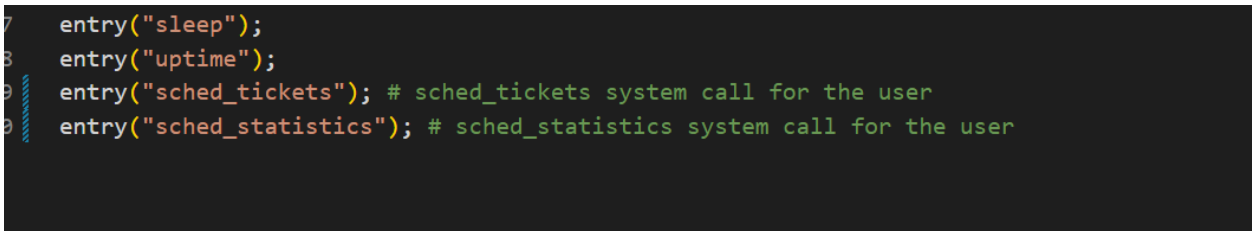
User\user.h:



```
int uptime(void);
void sched_tickets(int);
void sched_statistics(void);
```

User\usys.pl

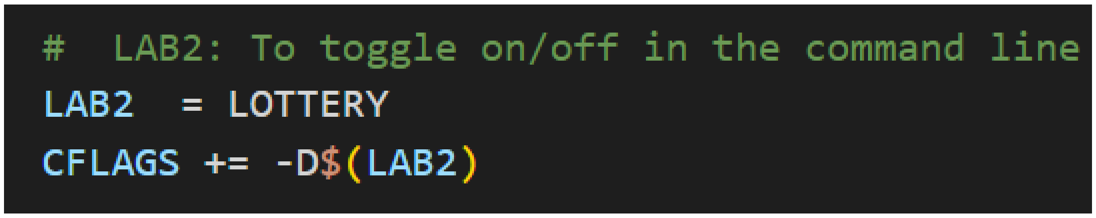
This is the entry point for the info



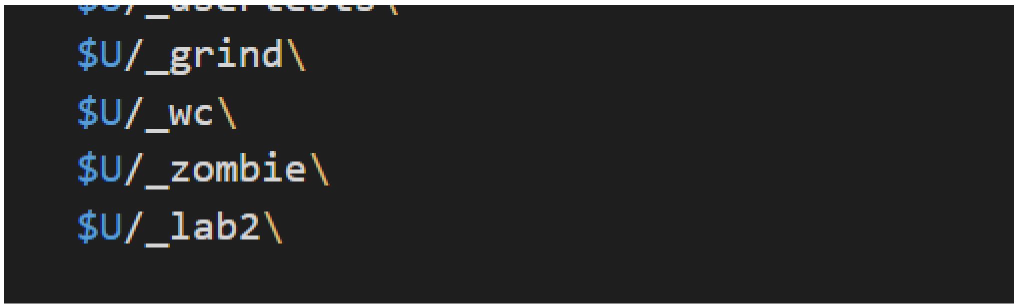
```
entry("sleep");
entry("uptime");
entry("sched_tickets"); # sched_tickets system call for the user
entry("sched_statistics"); # sched_statistics system call for the user
```

Make File:

Changing the lines 72 and 73 for controlling the command line



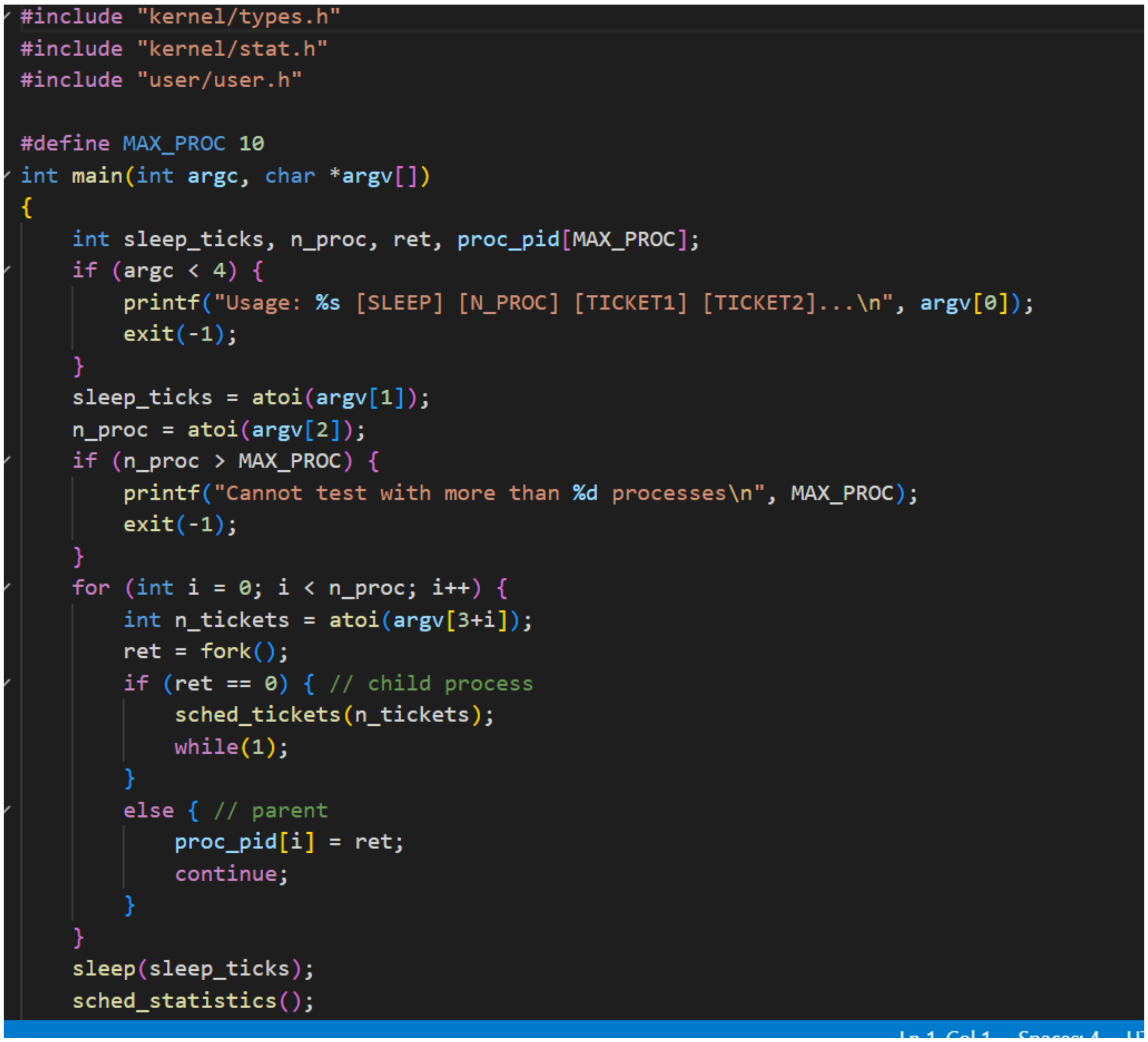
```
#  LAB2: To toggle on/off in the command line
LAB2   = LOTTERY
CFLAGS += -D$(LAB2)
```



```
    $U/_grind\
    $U/_wc\
    $U/_zombie\
    $U/_lab2\
```

Lab2.c:

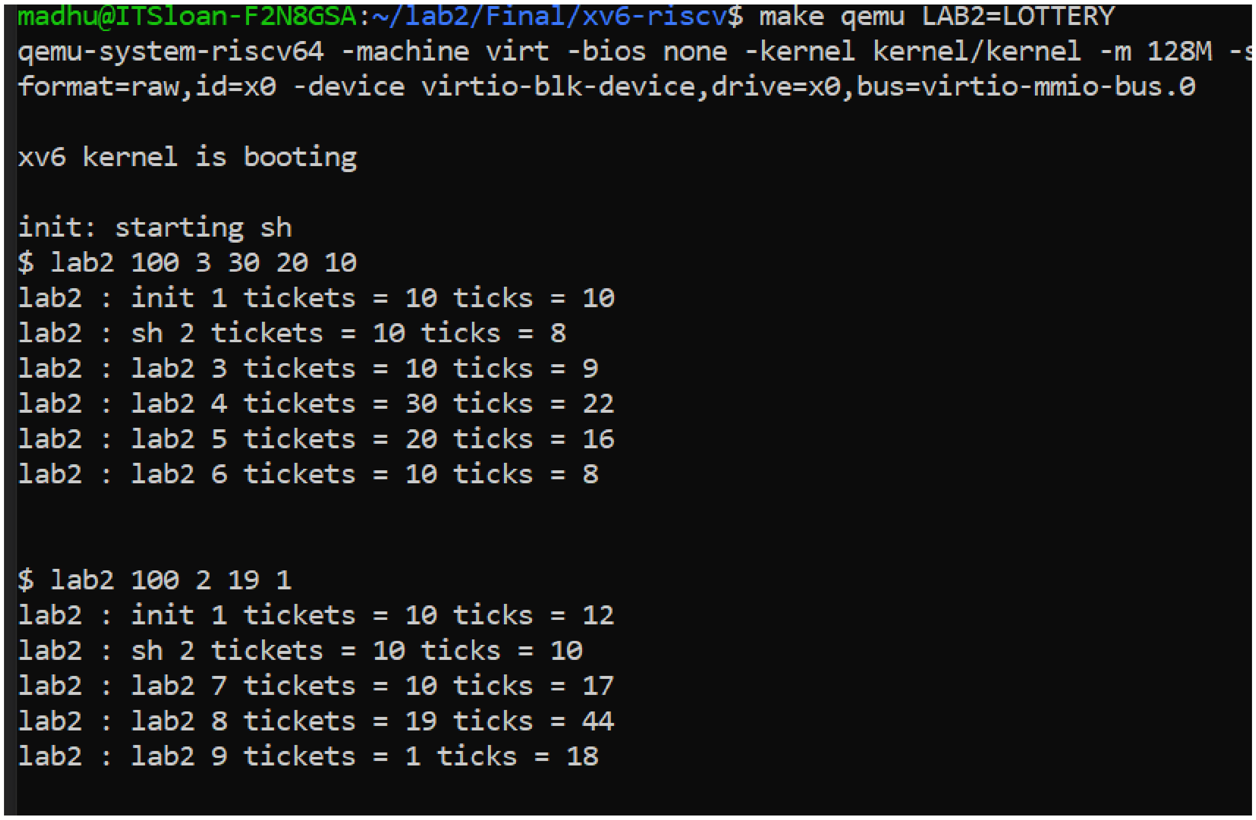
This is the user program function that is added to test the functionality and it will be used to call the system call, when we run the user program n qemu on the terminal it will display the results



```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

#define MAX_PROC 10
int main(int argc, char *argv[])
{
    int sleep_ticks, n_proc, ret, proc_pid[MAX_PROC];
    if (argc < 4) {
        printf("Usage: %s [SLEEP] [N_PROC] [TICKET1] [TICKET2]...\n", argv[0]);
        exit(-1);
    }
    sleep_ticks = atoi(argv[1]);
    n_proc = atoi(argv[2]);
    if (n_proc > MAX_PROC) {
        printf("Cannot test with more than %d processes\n", MAX_PROC);
        exit(-1);
    }
    for (int i = 0; i < n_proc; i++) {
        int n_tickets = atoi(argv[3+i]);
        ret = fork();
        if (ret == 0) { // child process
            sched_tickets(n_tickets);
            while(1);
        }
        else { // parent
            proc_pid[i] = ret;
            continue;
        }
    }
    sleep(sleep_ticks);
    sched_statistics();
```

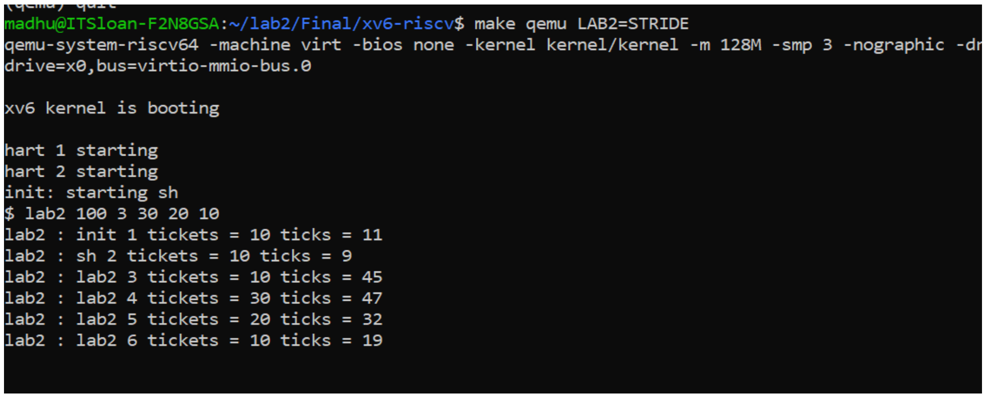
Output

Lottery: We have shown this in the demo also



```
madhu@ITSloan-F2N8GSA:~/lab2/Final/xv6-riscv$ make qemu LAB2=LOTTERY
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -s
format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

init: starting sh
$ lab2 100 3 30 20 10
lab2 : init 1 tickets = 10 ticks = 10
lab2 : sh 2 tickets = 10 ticks = 8
lab2 : lab2 3 tickets = 10 ticks = 9
lab2 : lab2 4 tickets = 30 ticks = 22
lab2 : lab2 5 tickets = 20 ticks = 16
lab2 : lab2 6 tickets = 10 ticks = 8


$ lab2 100 2 19 1
lab2 : init 1 tickets = 10 ticks = 12
lab2 : sh 2 tickets = 10 ticks = 10
lab2 : lab2 7 tickets = 10 ticks = 17
lab2 : lab2 8 tickets = 19 ticks = 44
lab2 : lab2 9 tickets = 1 ticks = 18
```

Stride: In the demo, We have explained for 3 processes and for 2 processes as well

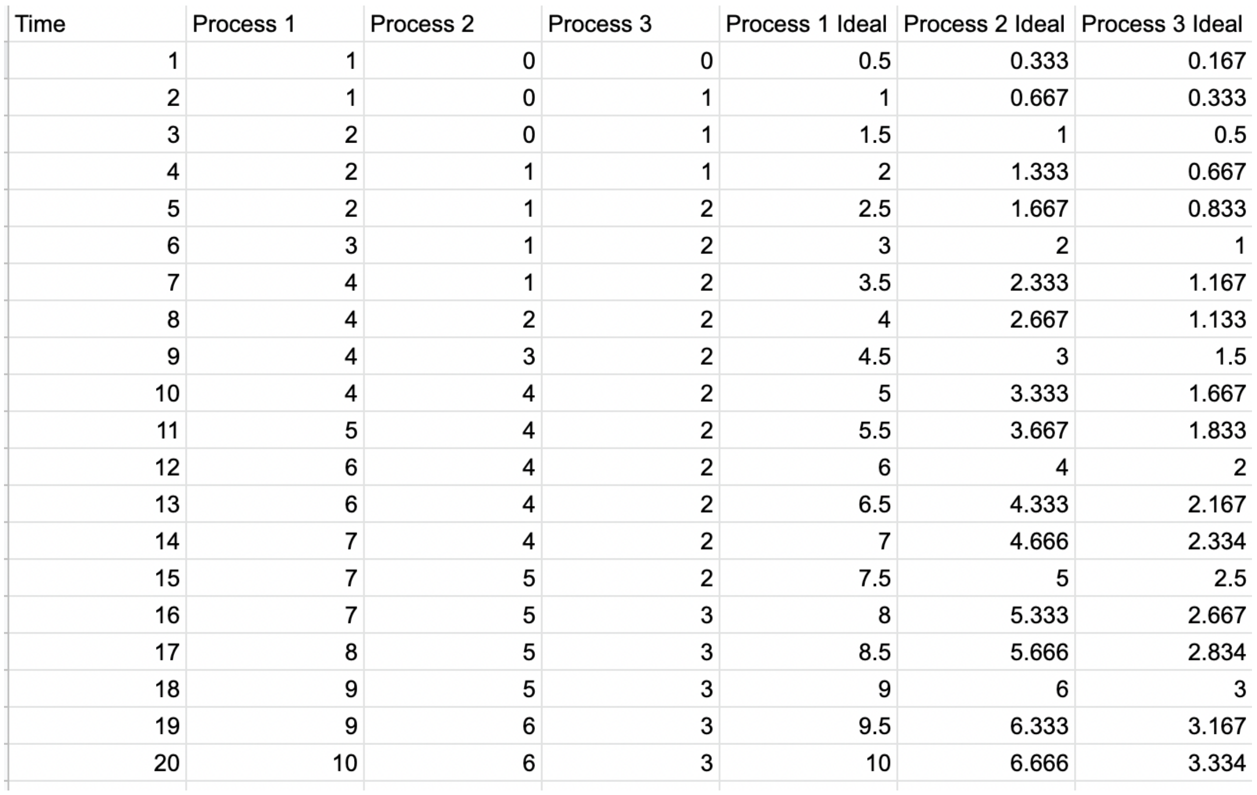


```
madhu@ITSloan-F2N8GSA:~/lab2/Final/xv6-riscv$ make qemu LAB2=STRIDE
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -dr
drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ lab2 100 3 30 20 10
lab2 : init 1 tickets = 10 ticks = 11
lab2 : sh 2 tickets = 10 ticks = 9
lab2 : lab2 3 tickets = 10 ticks = 45
lab2 : lab2 4 tickets = 30 ticks = 47
lab2 : lab2 5 tickets = 20 ticks = 32
lab2 : lab2 6 tickets = 10 ticks = 19
```

4. Graphs

STRIDE Scheduling:

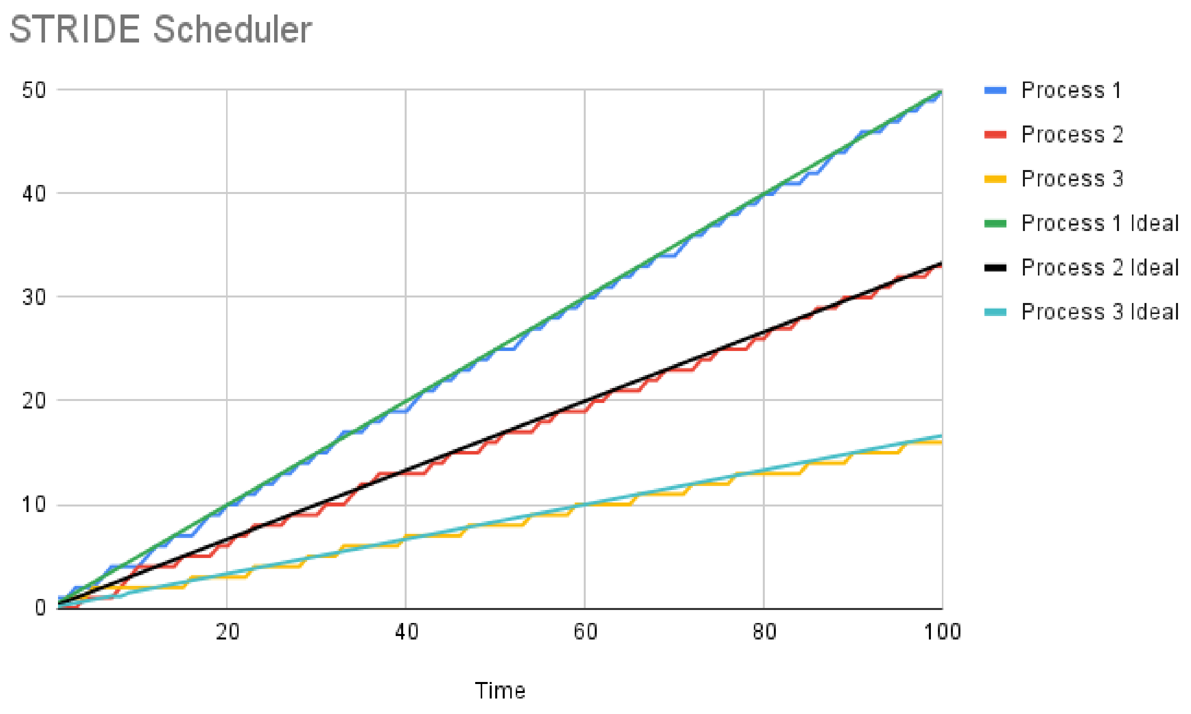
First we generate a table for the inputs for 3 processors with 30, 20, 10 tickets and 2 processors with 19, 1 respectively, then we use the data we got from the output and compare that with the ideal process (here the ideal process is the one with corresponding ticket ratio 3:2:1 split, 19:1 and drawing lines to them from 0).

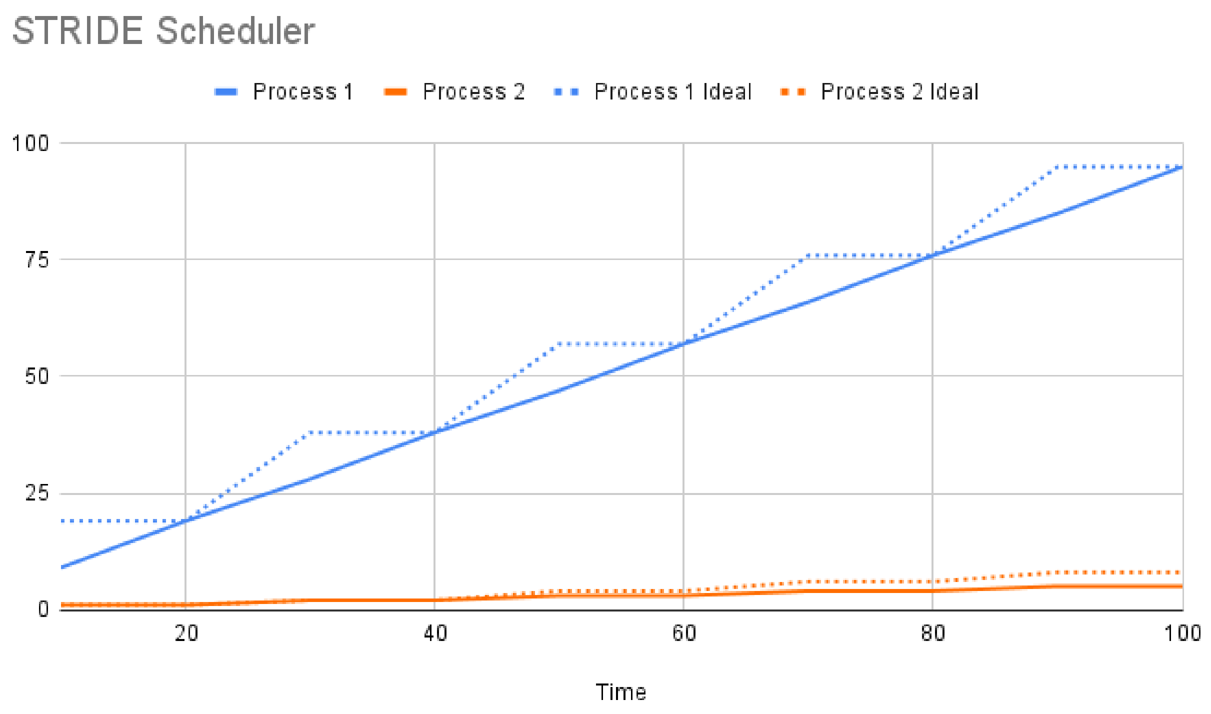


Time	Process 1	Process 2	Process 3	Process 1 Ideal	Process 2 Ideal	Process 3 Ideal
1	1	0	0	0.5	0.333	0.167
2	1	0	1	1	0.667	0.333
3	2	0	1	1.5	1	0.5
4	2	1	1	2	1.333	0.667
5	2	1	2	2.5	1.667	0.833
6	3	1	2	3	2	1
7	4	1	2	3.5	2.333	1.167
8	4	2	2	4	2.667	1.133
9	4	3	2	4.5	3	1.5
10	4	4	2	5	3.333	1.667
11	5	4	2	5.5	3.667	1.833
12	6	4	2	6	4	2
13	6	4	2	6.5	4.333	2.167
14	7	4	2	7	4.666	2.334
15	7	5	2	7.5	5	2.5
16	7	5	3	8	5.333	2.667
17	8	5	3	8.5	5.666	2.834
18	9	5	3	9	6	3
19	9	6	3	9.5	6.333	3.167
20	10	6	3	10	6.666	3.334

Time	Process 1	Process 2	Process 1 Ideal	Process 2 Ideal
10	9	1	19	1
20	19	1	19	1
30	28	2	38	2
40	38	2	38	2
50	47	3	57	4
60	57	3	57	4
70	66	4	76	6
80	76	4	76	6
90	85	5	95	8
100	95	5	95	8

We generate graphs from the above tables and we can see that the ideal and the actual graphs are very similar to each other as STRIDE Scheduler is a deterministic fair scheduler version of lottery scheduler. Unlike lottery, STRIDE provides consistent results with the ideal values.





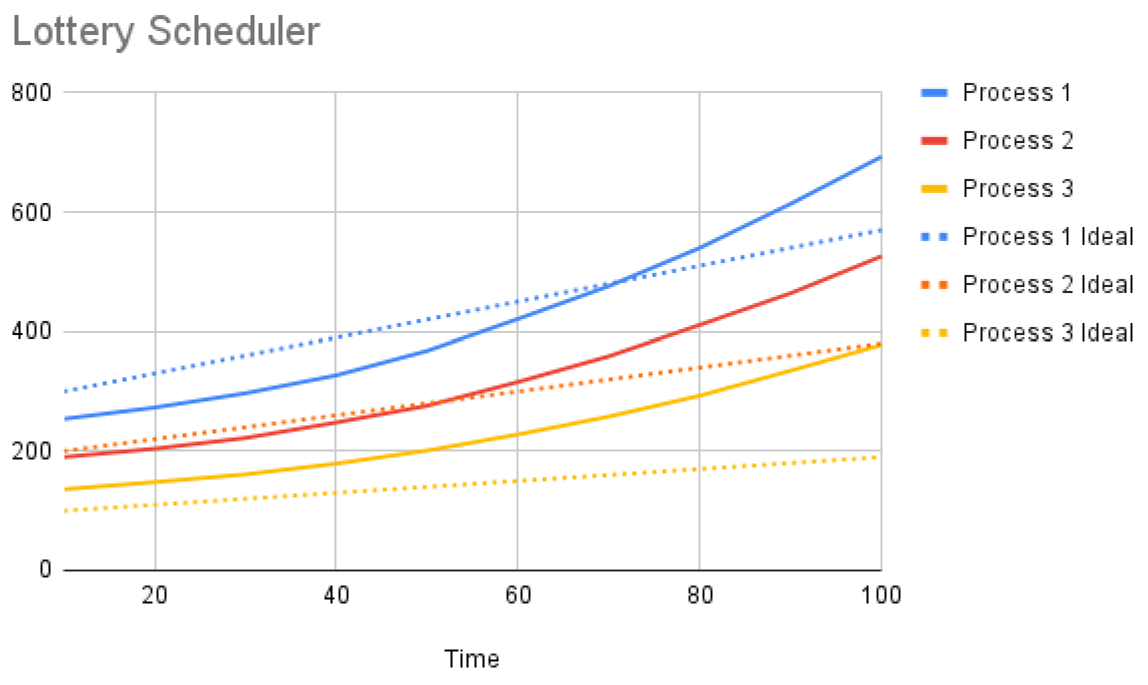
Lottery Scheduling:

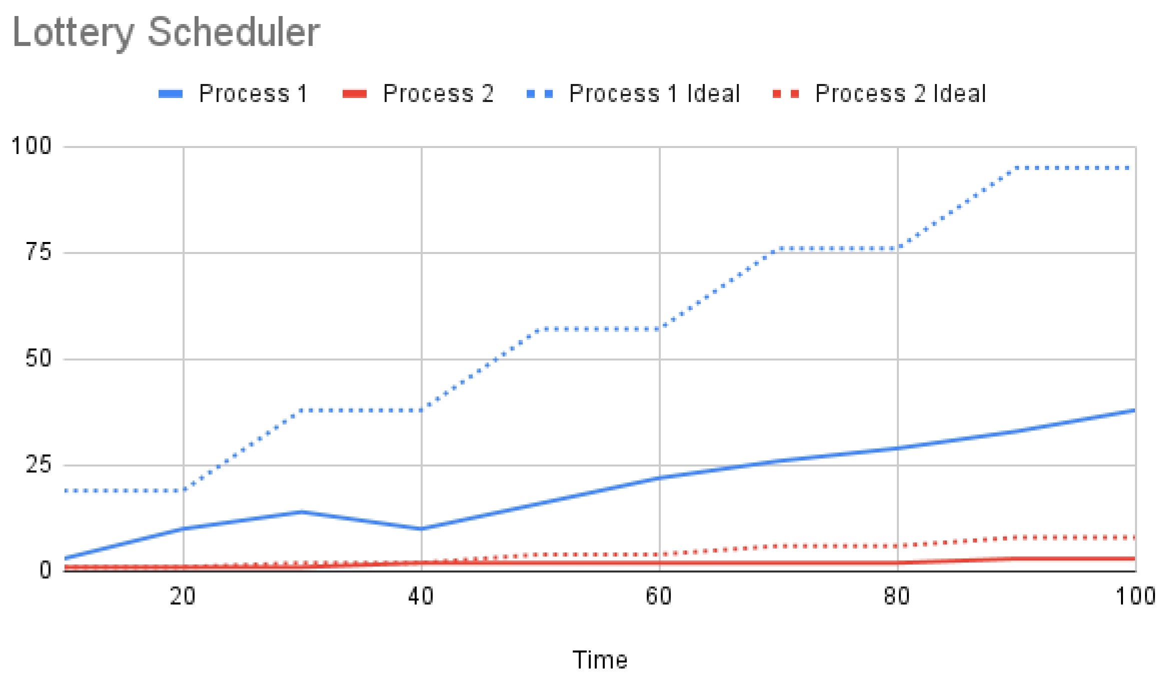
We generate a table for the inputs as we did in STRIDE for 3 processors with 30, 20, 10 tickets and 2 processors with 19, 1 respectively, then we use the data we got from the output and compare that with the ideal process (here the ideal process is the one with corresponding ticket ratio 3:2:1 split, 19:1 and drawing lines to them from 0).

Time	Process 1	Process 2	Process 3	Process 1 Ideal	Process 2 Ideal	Process 3 Ideal
10	254	190	136	300	200	100
20	273	204	148	330	220	110
30	297	222	161	360	240	120
40	327	248	179	390	260	130
50	368	276	201	420	280	140
60	421	316	228	450	300	150
70	476	359	258	480	320	160
80	540	411	293	510	340	170
90	614	464	335	540	360	180
100	693	526	378	570	380	190

Time	Process 1	Process 2	Process 1 Ideal	Process 2 Ideal
10	3	1	19	1
20	10	1	19	1
30	14	1	38	2
40	10	2	38	2
50	16	2	57	4
60	22	2	57	4
70	26	2	76	6
80	29	2	76	6
90	33	3	95	8
100	38	3	95	8

We generate graphs from the above tables and we can see that the ideal and the actual graphs vary vastly with each other as LOTTERY Scheduler is a probabilistic fair scheduler. Unlike STRIDE, LOTTERY scheduler's ideal and actual process vary a lot as randomness does not guarantee fairness.





6. Running Commands:

Running on Windows subsystem:

1. Install ubuntu from Microsoft store
2. Run the below commands

$ sudo apt-get update && sudo apt-get upgrade

$ sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu

3. Copy our project and got to the folder “cd xv6-riscv”

Running Lottery:

4. Run the command “make qemu LAB2=LOTTERY”
   a. Check with the following command “lab2 100 3 30 20 10” and “lab2 100 2 19 1”

Running Stride:

5. Run the command “make qemu LAB2=STRIDE”
   b. Check with the following command “lab2 100 3 30 20 10” and “lab2 100 2 19 1”

7.Demo Link:

https://drive.google.com/drive/folders/1aZi0fdHxhMk8i0LSMEXxetZ6kjutb8wD?usp=sharing