LAB - 2 Report

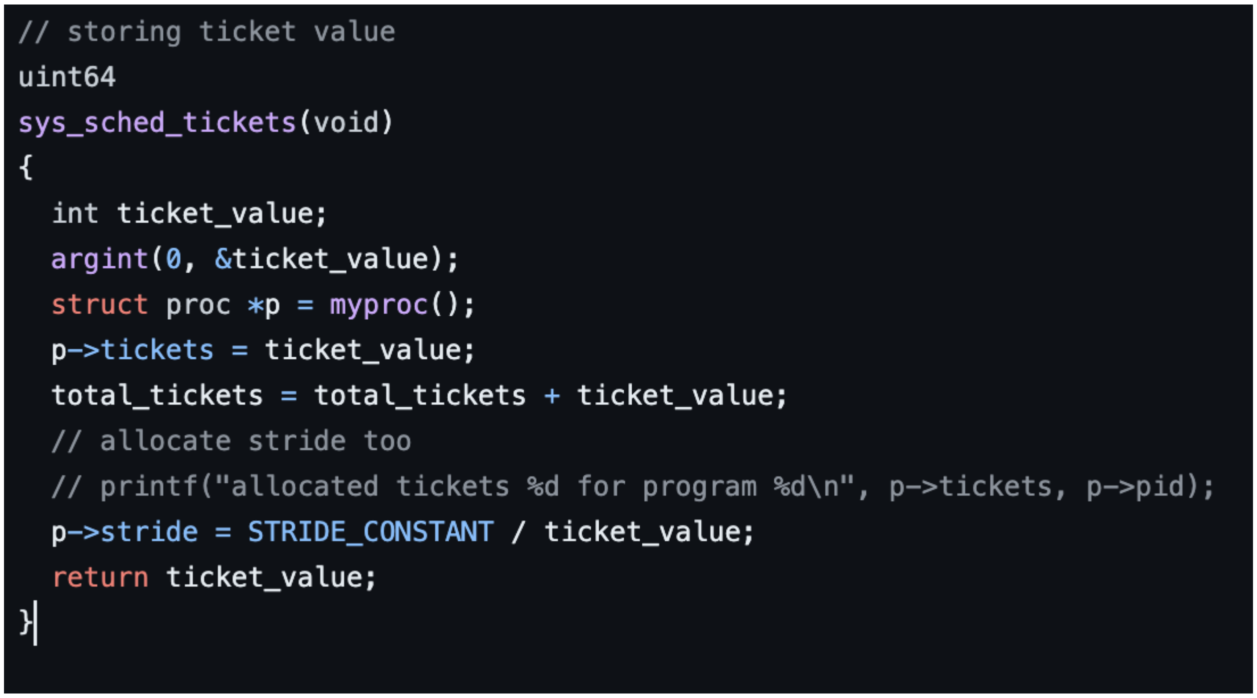
Aaryan Bhagat, Yash Bhalgat

Demo Video Link

Implementation details

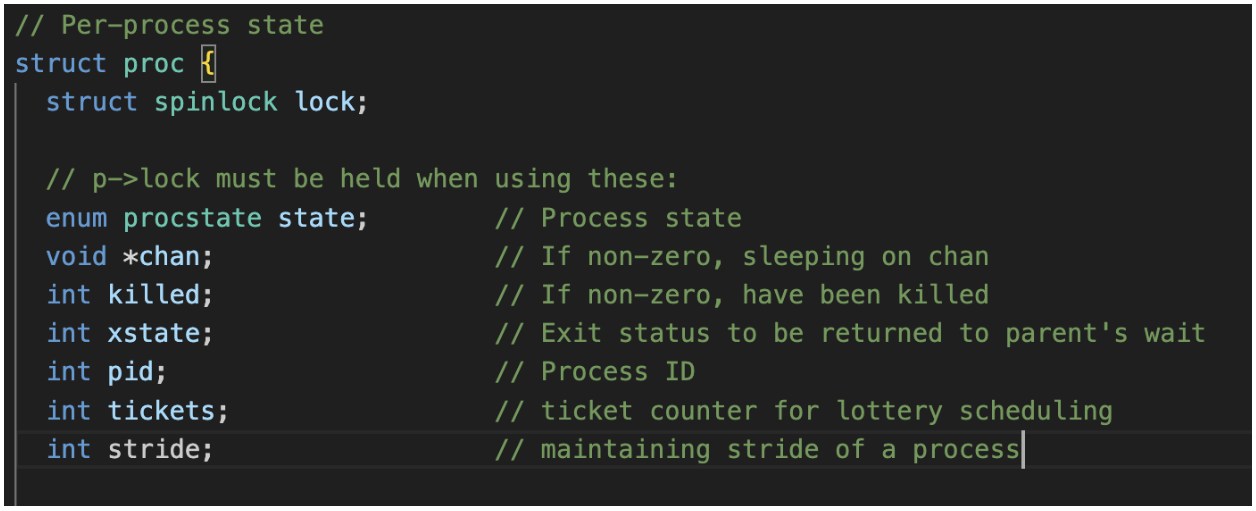
I) System calls (with explanation)

Sys_sched_tickets:
This system call helps to store the ticket value obtained from input into the 'ticket' property of a process



```
// storing ticket value
uint64
sys_sched_tickets(void)
{
  int ticket_value;
  argint(0, &ticket_value);
  struct proc *p = myproc();
  p->tickets = ticket_value;
  total_tickets = total_tickets + ticket_value;
  // allocate stride too
  // printf("allocated tickets %d for program %d\n", p->tickets, p->pid);
  p->stride = STRIDE_CONSTANT / ticket_value;
  return ticket_value;
}
```

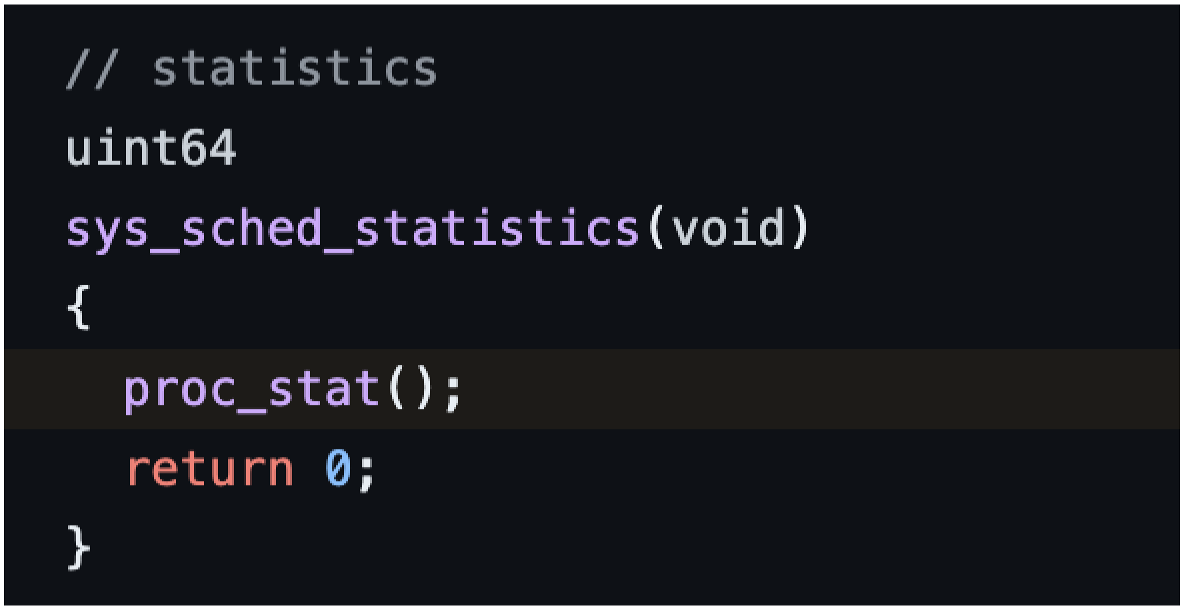
This system call also helps to initialise the stride value for the stride scheduling part of the assignment. A counter of total_tickets is maintained which helps to locate the process that falls in the randomly chosen ticket in lottery scheduling.



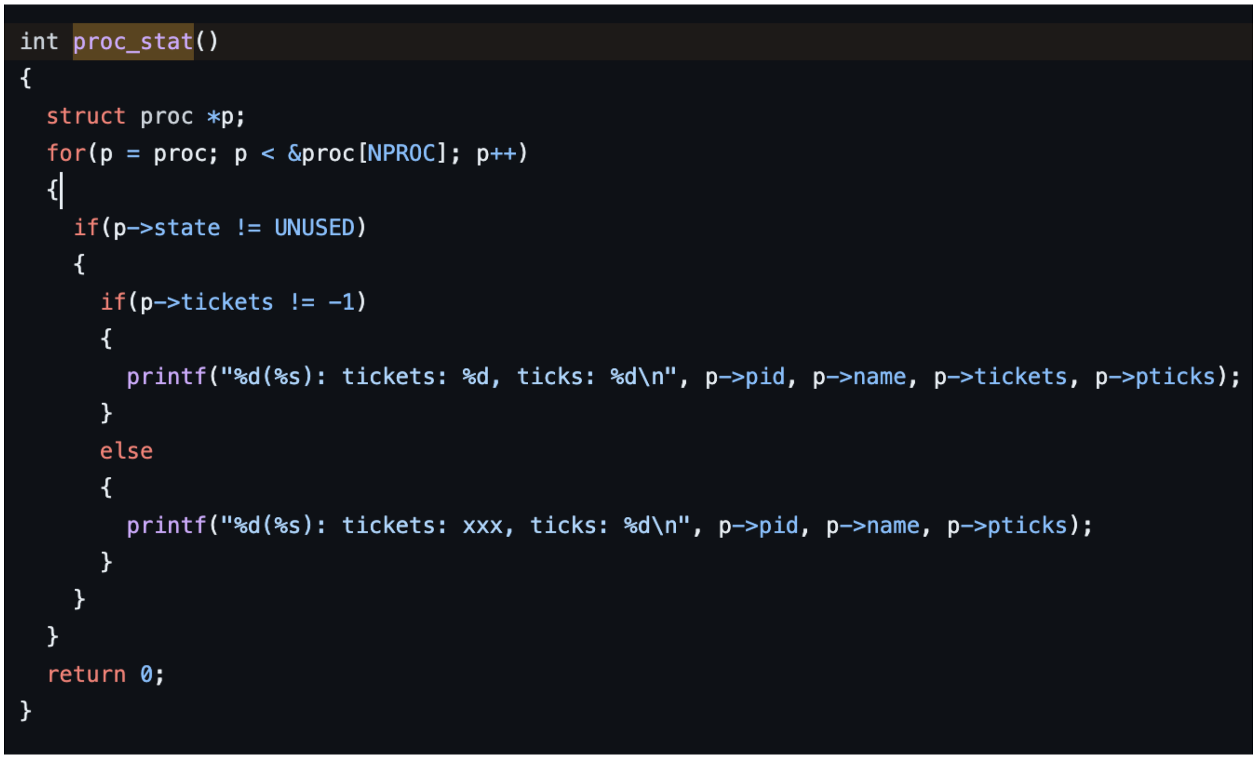
```
// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID
  int tickets;                 // ticket counter for lottery scheduling
  int stride;                  // maintaining stride of a process
```

sys_sched_statistics:

This system call prints, for each process, 1) PID, 2) name in a parenthesis, 3) the ticket value, and 4) the number of times it has been scheduled to run (i.e., a rough estimation of the number of ticks used by each process).



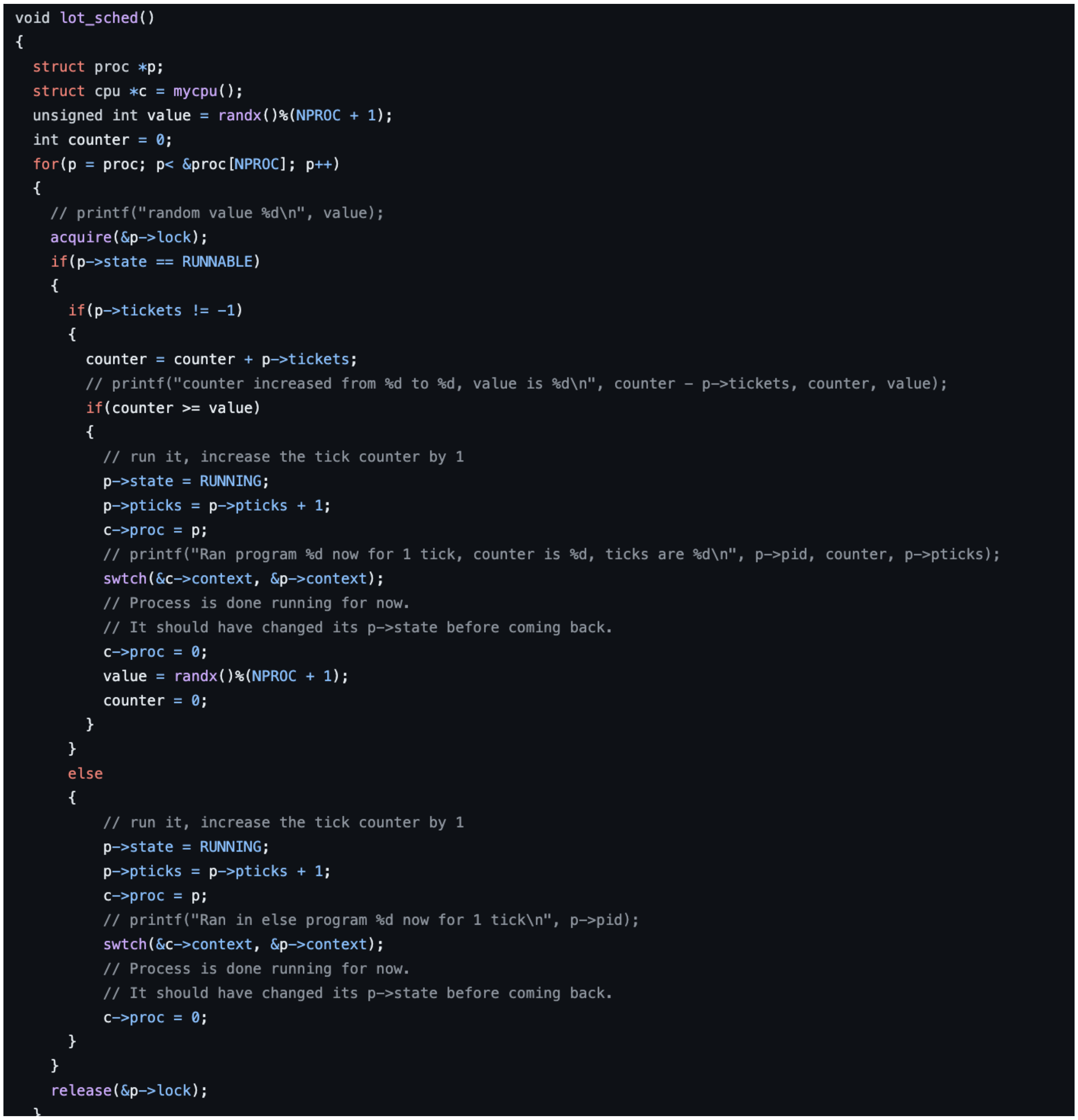
```
// statistics
uint64
sys_sched_statistics(void)
{
  proc_stat();
  return 0;
}
```



```c
int proc_stat()
{
  struct proc *p;
  for(p = proc; p < &proc[NPROC]; p++)
  {
    if(p->state != UNUSED)
    {
      if(p->tickets != -1)
      {
        printf("%d(%s): tickets: %d, ticks: %d\n", p->pid, p->name, p->tickets, p->pticks);
      }
      else
      {
        printf("%d(%s): tickets: xxx, ticks: %d\n", p->pid, p->name, p->pticks);
      }
    }
  }
  return 0;
}
```

II) Lottery Scheduler:

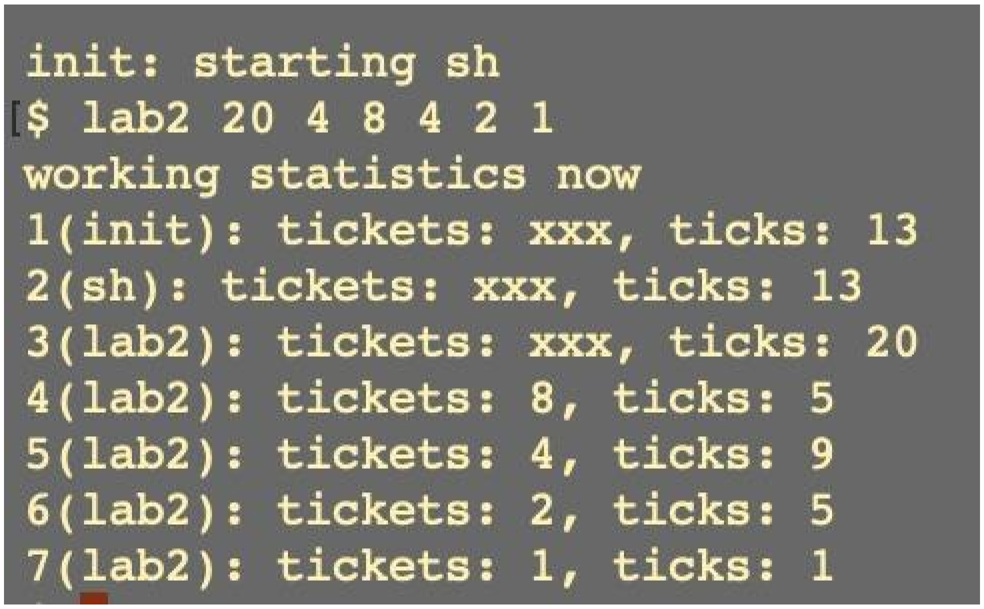
After making a system call which allows you to set the tickets for a process, In the scheduler function count the total number of tickets for all processes that are runnable. Generate a random number between 0 and the total tickets calculated above. When looping through the processes keep the counter of the total number of tickets passed. Just when the counter becomes greater the random value we got, run the process.



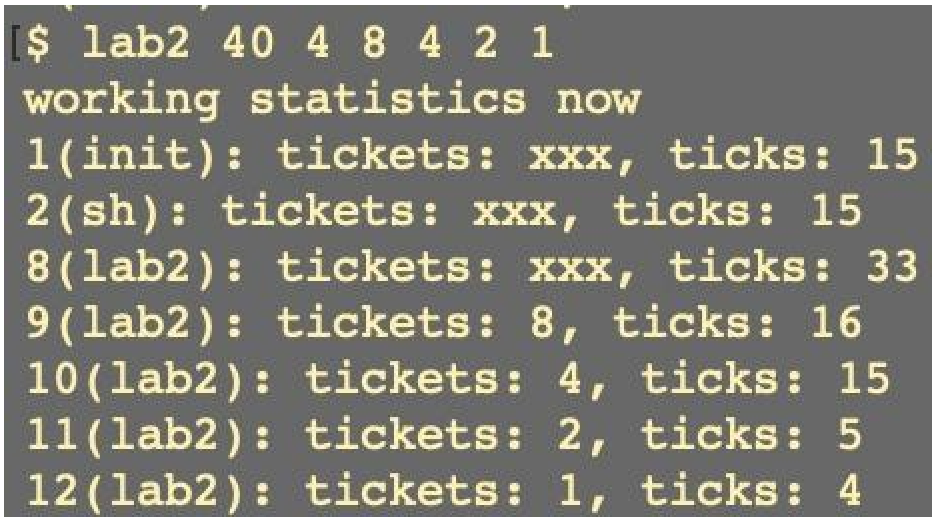
```c
void lot_sched()
{
  struct proc *p;
  struct cpu *c = mycpu();
  unsigned int value = randx()%(NPROC + 1);
  int counter = 0;
  for(p = proc; p< &proc[NPROC]; p++)
  {
    // printf("random value %d\n", value);
    acquire(&p->lock);
    if(p->state == RUNNABLE)
    {
      if(p->tickets != -1)
      {
        counter = counter + p->tickets;
        // printf("counter increased from %d to %d, value is %d\n", counter - p->tickets, counter, value);
        if(counter >= value)
        {
          // run it, increase the tick counter by 1
          p->state = RUNNING;
          p->pticks = p->pticks + 1;
          c->proc = p;
          // printf("Ran program %d now for 1 tick, counter is %d, ticks are %d\n", p->pid, counter, p->pticks);
          swtch(&c->context, &p->context);
          // Process is done running for now.
          // It should have changed its p->state before coming back.
          c->proc = 0;
          value = randx()%(NPROC + 1);
          counter = 0;
        }
      }
      else
      {
          // run it, increase the tick counter by 1
          p->state = RUNNING;
          p->pticks = p->pticks + 1;
          c->proc = p;
          // printf("Ran in else program %d now for 1 tick\n", p->pid);
          swtch(&c->context, &p->context);
          // Process is done running for now.
          // It should have changed its p->state before coming back.
          c->proc = 0;
      }
    }
    release(&p->lock);
  }
```

Output:

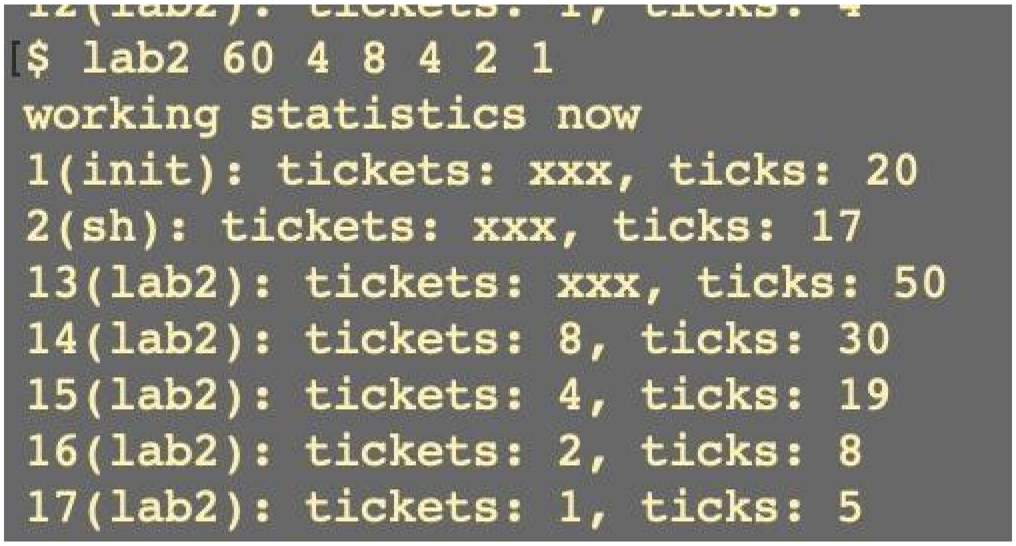
For 20, 40, 60, 80 time ticks, four processes with 8, 4, 2, 1 tickets for each



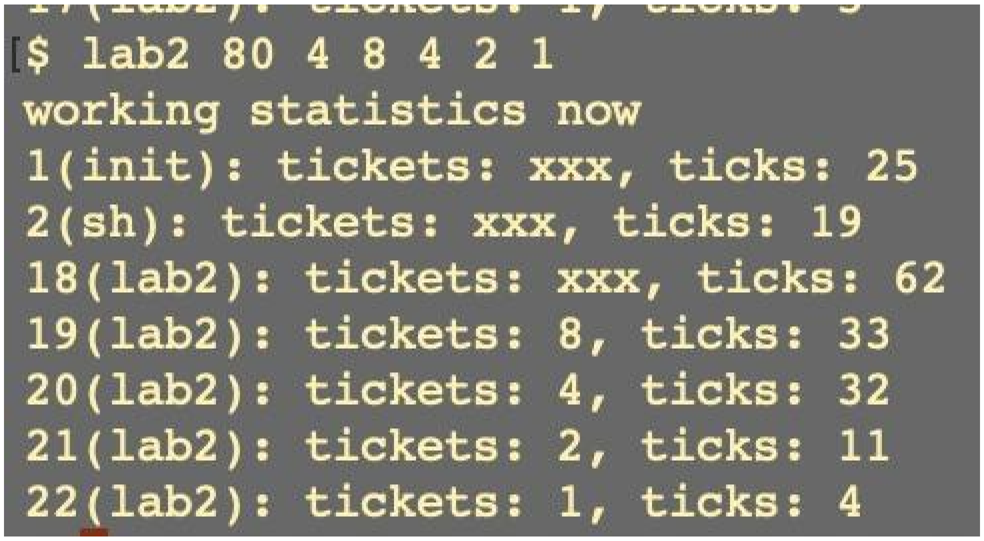
```
init: starting sh
$ lab2 20 4 8 4 2 1
working statistics now
1(init): tickets: xxx, ticks: 13
2(sh): tickets: xxx, ticks: 13
3(lab2): tickets: xxx, ticks: 20
4(lab2): tickets: 8, ticks: 5
5(lab2): tickets: 4, ticks: 9
6(lab2): tickets: 2, ticks: 5
7(lab2): tickets: 1, ticks: 1
```



```
$ lab2 40 4 8 4 2 1
working statistics now
1(init): tickets: xxx, ticks: 15
2(sh): tickets: xxx, ticks: 15
8(lab2): tickets: xxx, ticks: 33
9(lab2): tickets: 8, ticks: 16
10(lab2): tickets: 4, ticks: 15
11(lab2): tickets: 2, ticks: 5
12(lab2): tickets: 1, ticks: 4
```



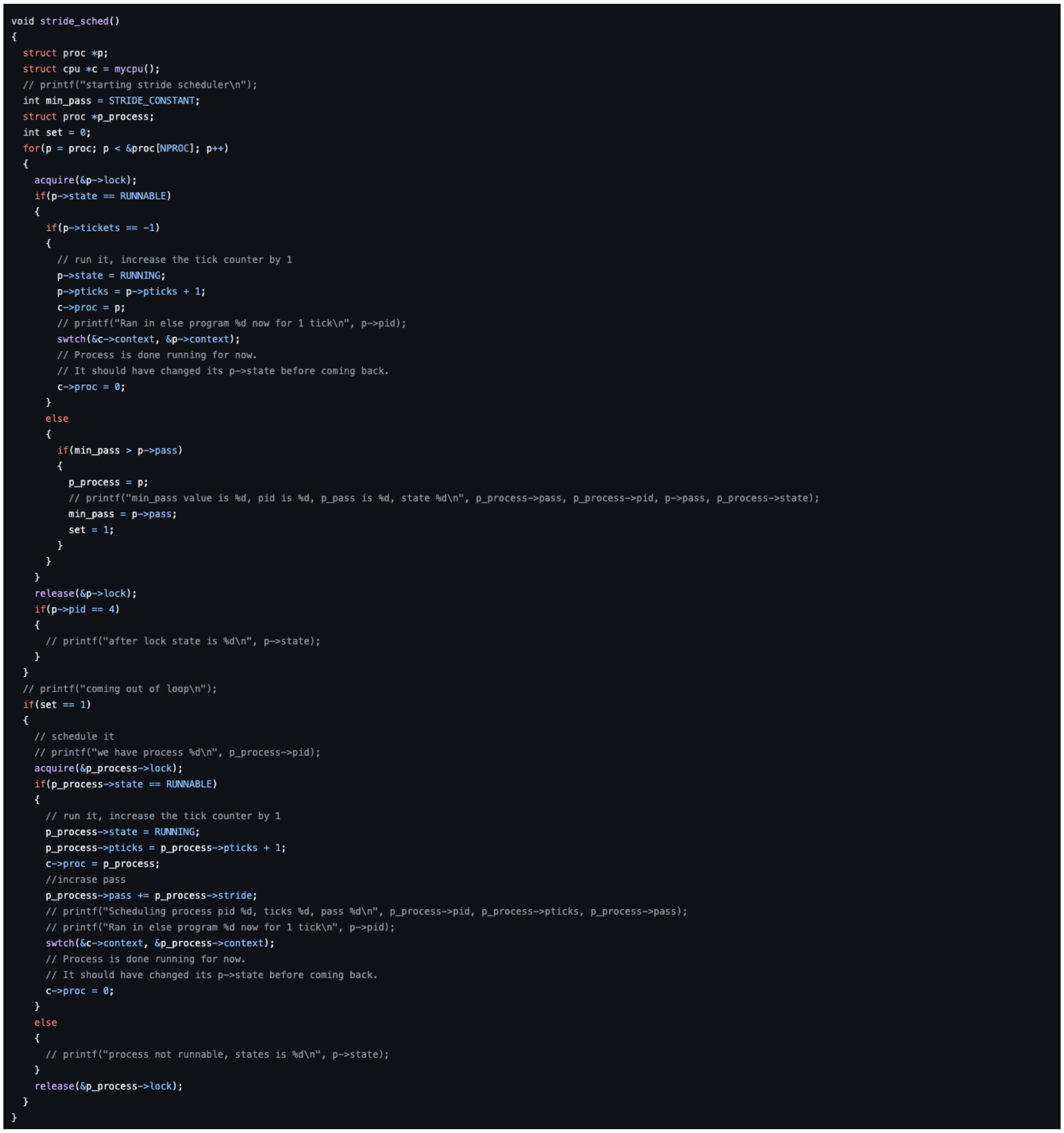
```
$ lab2 60 4 8 4 2 1
working statistics now
1(init): tickets: xxx, ticks: 20
2(sh): tickets: xxx, ticks: 17
13(lab2): tickets: xxx, ticks: 50
14(lab2): tickets: 8, ticks: 30
15(lab2): tickets: 4, ticks: 19
16(lab2): tickets: 2, ticks: 8
17(lab2): tickets: 1, ticks: 5
```



```
[$ lab2 80 4 8 4 2 1
working statistics now
1(init): tickets: xxx, ticks: 25
2(sh): tickets: xxx, ticks: 19
18(lab2): tickets: xxx, ticks: 62
19(lab2): tickets: 8, ticks: 33
20(lab2): tickets: 4, ticks: 32
21(lab2): tickets: 2, ticks: 11
22(lab2): tickets: 1, ticks: 4
```

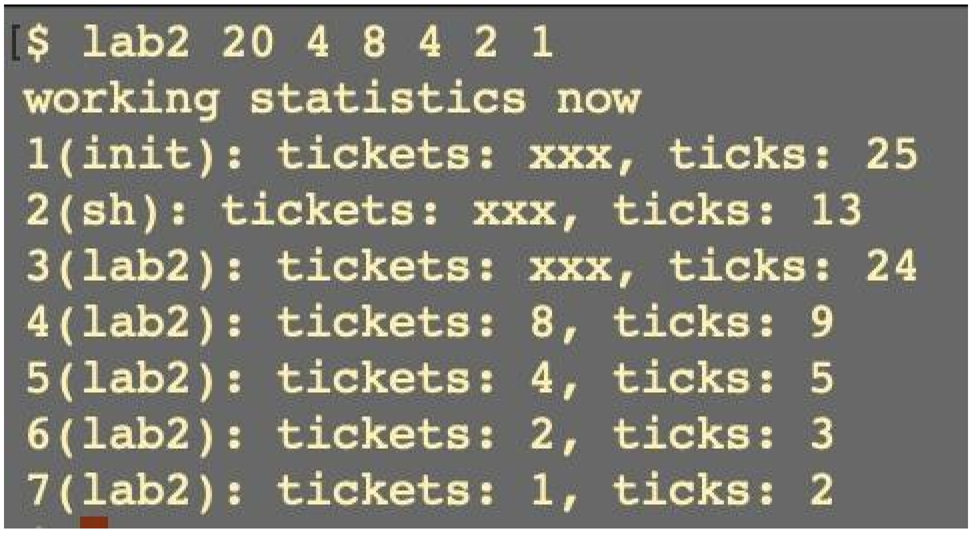
<u>III) Stride Scheduling:</u>

In stride scheduling the task with minimum pass will be selected.
We initialise the min_stride to some constant high value and then loop throught the proc table. If that process is not runnable then we continue further else we see if the stride value of the current process is less than the current minimum stride value then we update the minimum stride and make the minimum process to the current process. So finally once the loop ends we would have the process with the minimum pass value. We can now update its stride with the initial stride constant value and make the process ‘RUNNING’

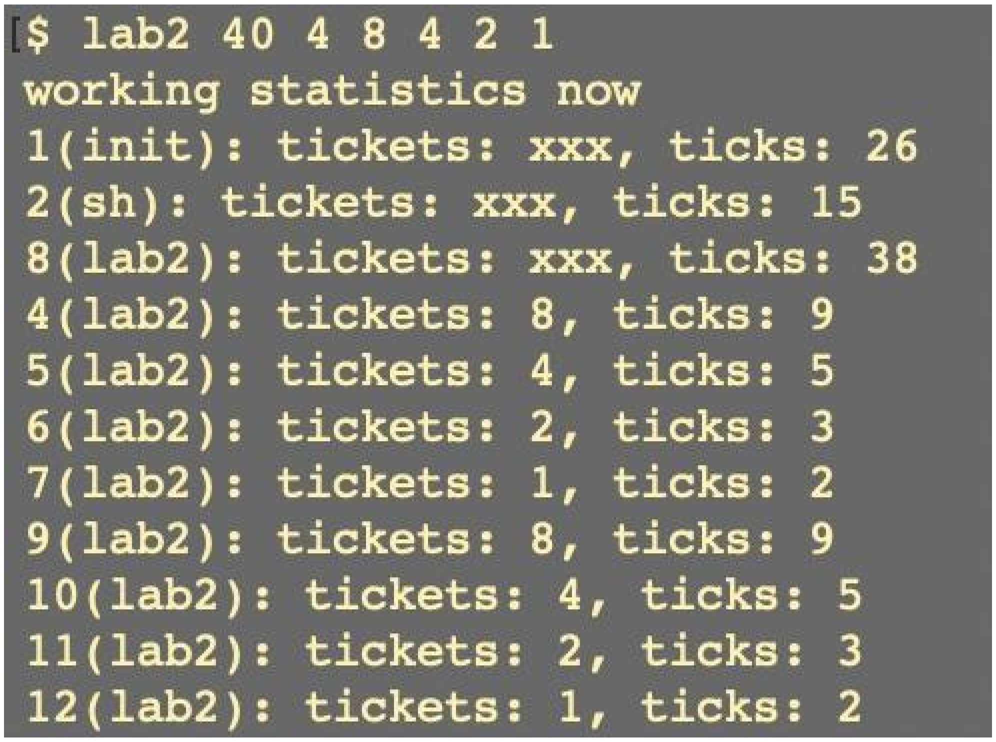


```c
void stride_sched()
{
  struct proc *p;
  struct cpu *c = mycpu();
  // printf("starting stride scheduler\n");
  int min_pass = STRIDE_CONSTANT;
  struct proc *p_process;
  int set = 0;
  for(p = proc; p < &proc[NPROC]; p++)
  {
    acquire(&p->lock);
    if(p->state == RUNNABLE)
    {
      if(p->tickets == -1)
      {
        // run it, increase the tick counter by 1
        p->state = RUNNING;
        p->pticks = p->pticks + 1;
        c->proc = p;
        // printf("Ran in else program %d now for 1 tick\n", p->pid);
        swtch(&c->context, &p->context);
        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;
      }
      else
      {
        if(min_pass > p->pass)
        {
          p_process = p;
          // printf("min_pass value is %d, pid is %d, p_pass is %d, state %d\n", p_process->pass, p_process->pid, p->pass, p_process->state);
          min_pass = p->pass;
          set = 1;
        }
      }
    }
    release(&p->lock);
    if(p->pid == 4)
    {
      // printf("after lock state is %d\n", p->state);
    }
  }
  // printf("coming out of loop\n");
  if(set == 1)
  {
    // schedule it
    // printf("we have process %d\n", p_process->pid);
    acquire(&p_process->lock);
    if(p_process->state == RUNNABLE)
    {
      // run it, increase the tick counter by 1
      p_process->state = RUNNING;
      p_process->pticks = p_process->pticks + 1;
      c->proc = p_process;
      //incrase pass
      p_process->pass += p_process->stride;
      // printf("Scheduling process pid %d, ticks %d, pass %d\n", p_process->pid, p_process->pticks, p_process->pass);
      // printf("Ran in else program %d now for 1 tick\n", p->pid);
      swtch(&c->context, &p_process->context);
      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    else
    {
      // printf("process not runnable, states is %d\n", p->state);
    }
    release(&p_process->lock);
  }
}
```

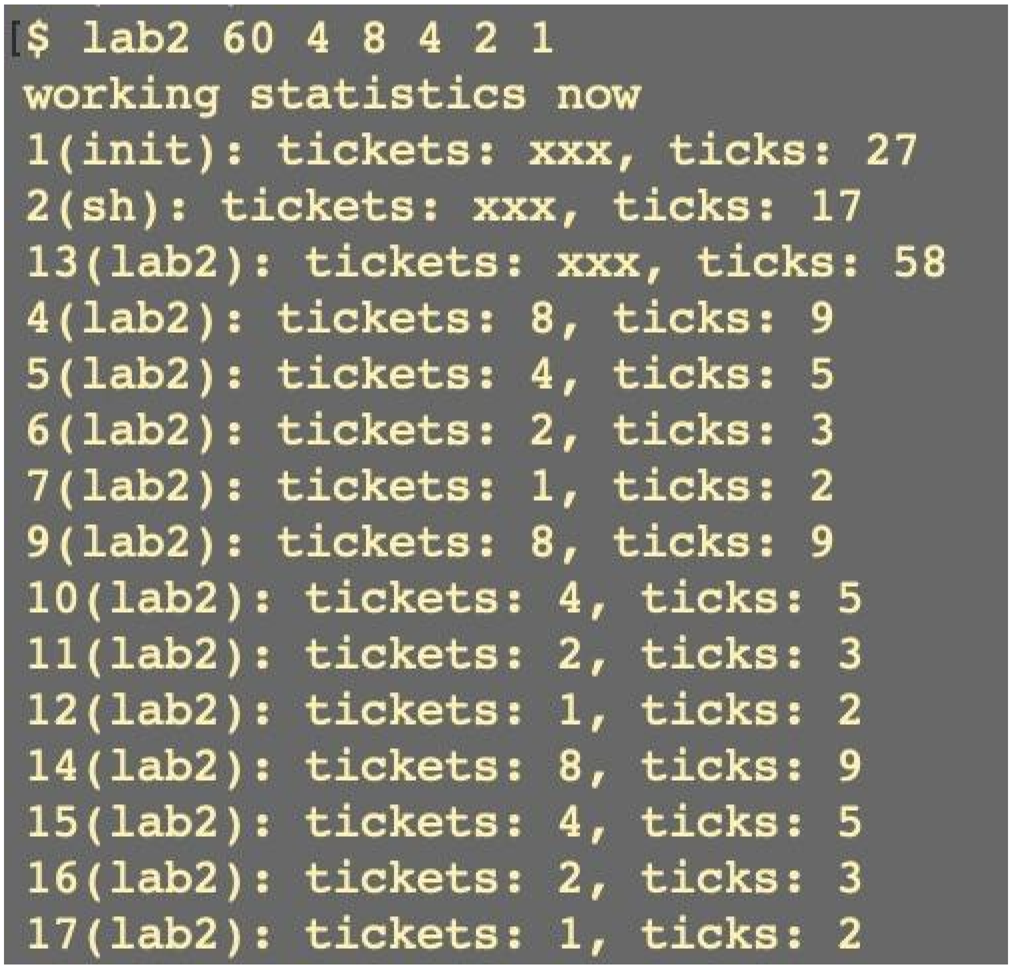
Output:



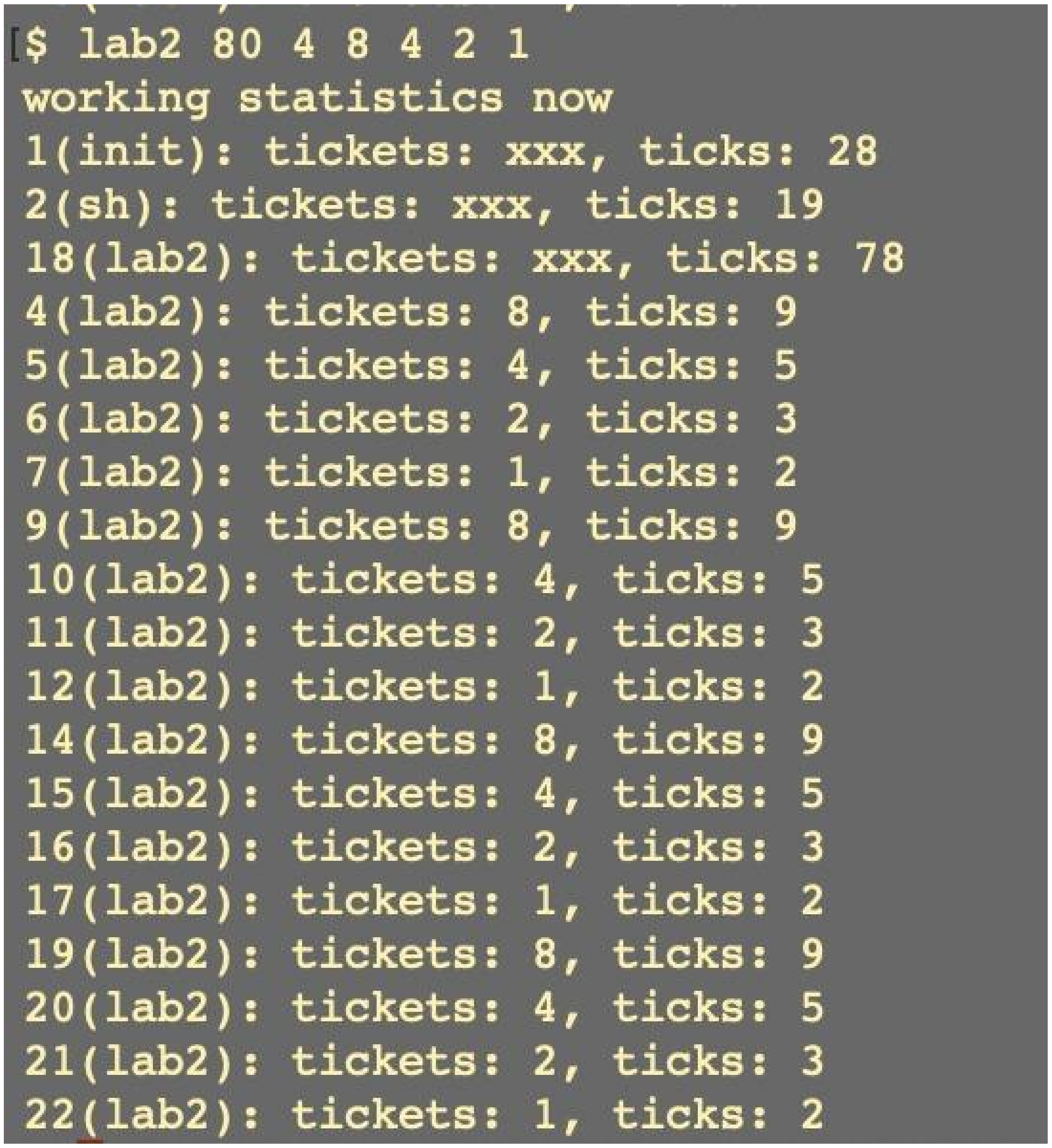
```
$ lab2 20 4 8 4 2 1
working statistics now
1(init): tickets: xxx, ticks: 25
2(sh): tickets: xxx, ticks: 13
3(lab2): tickets: xxx, ticks: 24
4(lab2): tickets: 8, ticks: 9
5(lab2): tickets: 4, ticks: 5
6(lab2): tickets: 2, ticks: 3
7(lab2): tickets: 1, ticks: 2
```



```
$ lab2 40 4 8 4 2 1
working statistics now
1(init): tickets: xxx, ticks: 26
2(sh): tickets: xxx, ticks: 15
8(lab2): tickets: xxx, ticks: 38
4(lab2): tickets: 8, ticks: 9
5(lab2): tickets: 4, ticks: 5
6(lab2): tickets: 2, ticks: 3
7(lab2): tickets: 1, ticks: 2
9(lab2): tickets: 8, ticks: 9
10(lab2): tickets: 4, ticks: 5
11(lab2): tickets: 2, ticks: 3
12(lab2): tickets: 1, ticks: 2
```

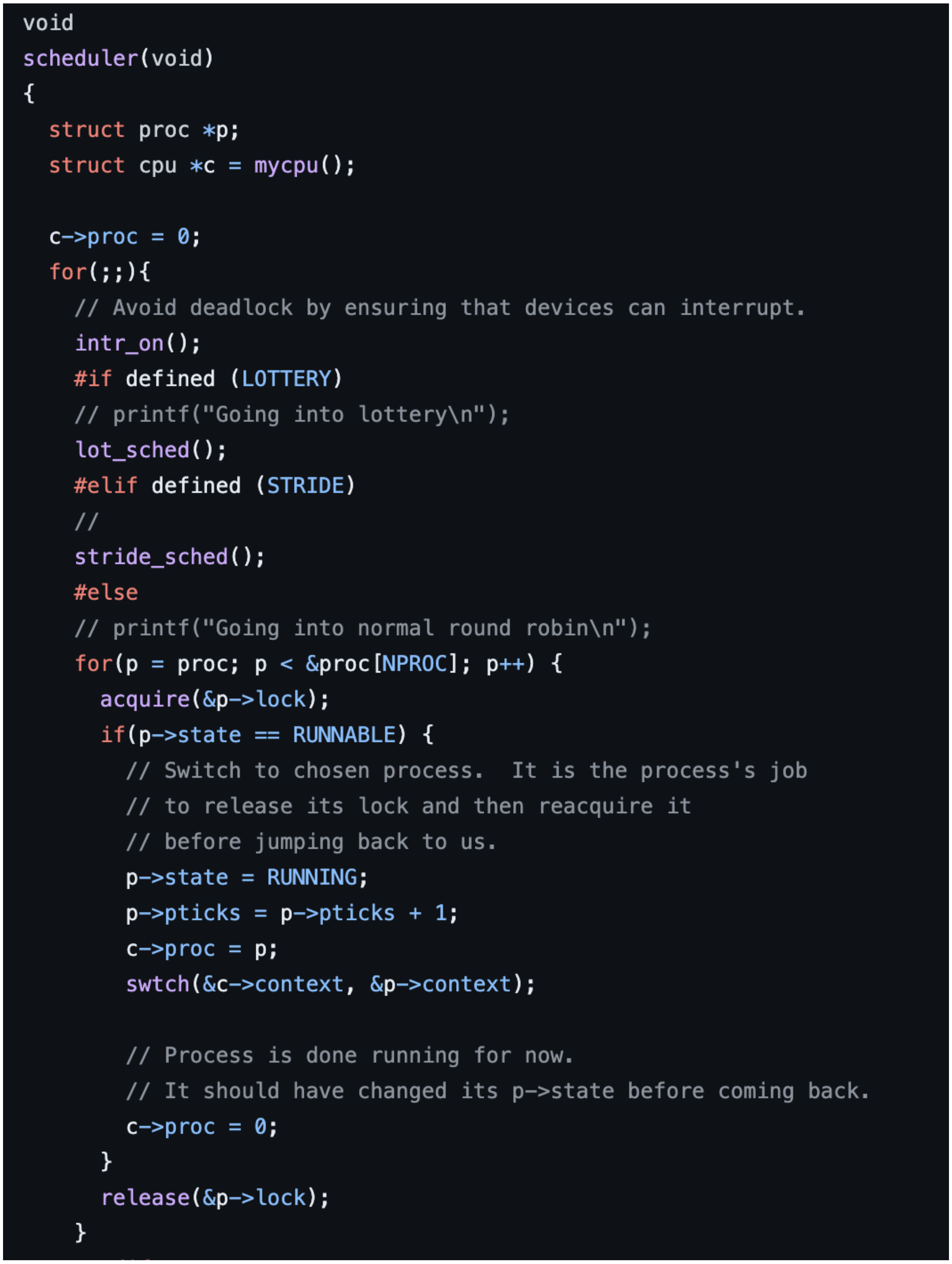


```
[$ lab2 60 4 8 4 2 1
working statistics now
1(init): tickets: xxx, ticks: 27
2(sh): tickets: xxx, ticks: 17
13(lab2): tickets: xxx, ticks: 58
4(lab2): tickets: 8, ticks: 9
5(lab2): tickets: 4, ticks: 5
6(lab2): tickets: 2, ticks: 3
7(lab2): tickets: 1, ticks: 2
9(lab2): tickets: 8, ticks: 9
10(lab2): tickets: 4, ticks: 5
11(lab2): tickets: 2, ticks: 3
12(lab2): tickets: 1, ticks: 2
14(lab2): tickets: 8, ticks: 9
15(lab2): tickets: 4, ticks: 5
16(lab2): tickets: 2, ticks: 3
17(lab2): tickets: 1, ticks: 2
```



```
[$ lab2 80 4 8 4 2 1
working statistics now
1(init): tickets: xxx, ticks: 28
2(sh): tickets: xxx, ticks: 19
18(lab2): tickets: xxx, ticks: 78
4(lab2): tickets: 8, ticks: 9
5(lab2): tickets: 4, ticks: 5
6(lab2): tickets: 2, ticks: 3
7(lab2): tickets: 1, ticks: 2
9(lab2): tickets: 8, ticks: 9
10(lab2): tickets: 4, ticks: 5
11(lab2): tickets: 2, ticks: 3
12(lab2): tickets: 1, ticks: 2
14(lab2): tickets: 8, ticks: 9
15(lab2): tickets: 4, ticks: 5
16(lab2): tickets: 2, ticks: 3
17(lab2): tickets: 1, ticks: 2
19(lab2): tickets: 8, ticks: 9
20(lab2): tickets: 4, ticks: 5
21(lab2): tickets: 2, ticks: 3
22(lab2): tickets: 1, ticks: 2
```

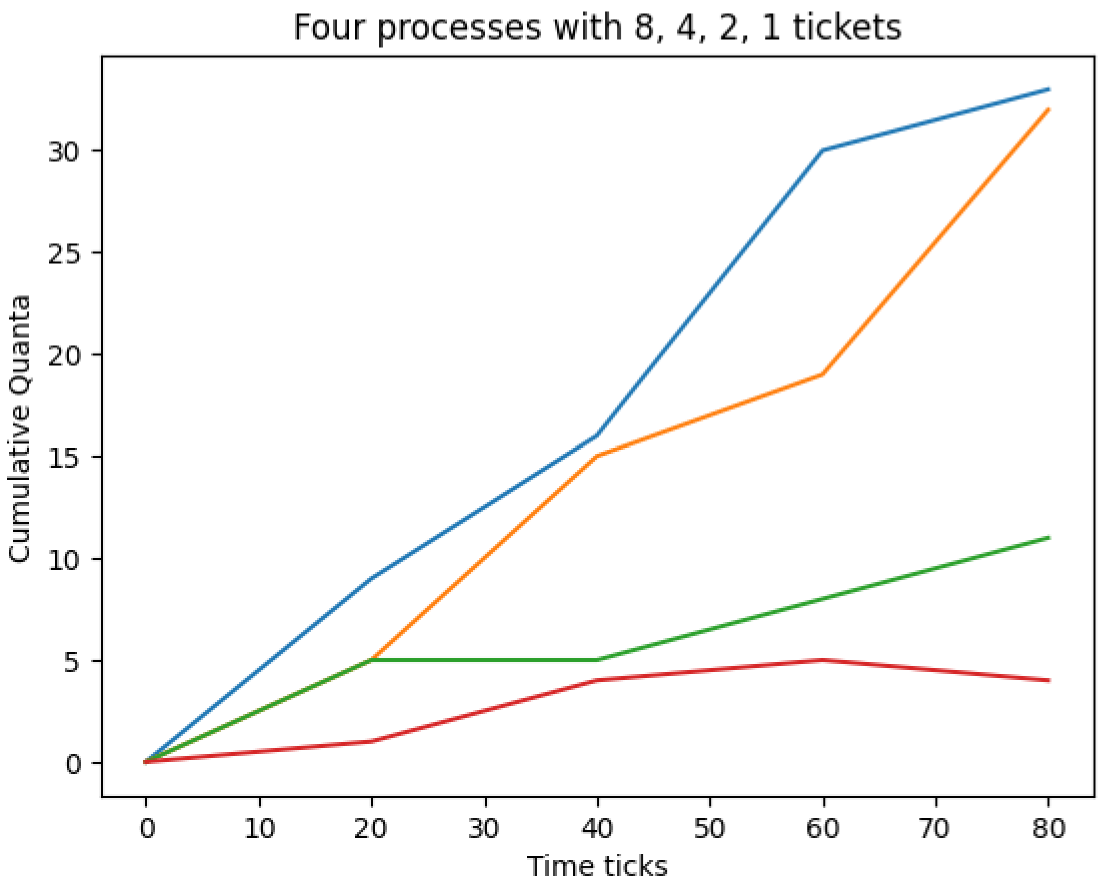
Below is the part of scheduler code which allows us to run lottery/stride/round robin based on the input provided from command prompt.

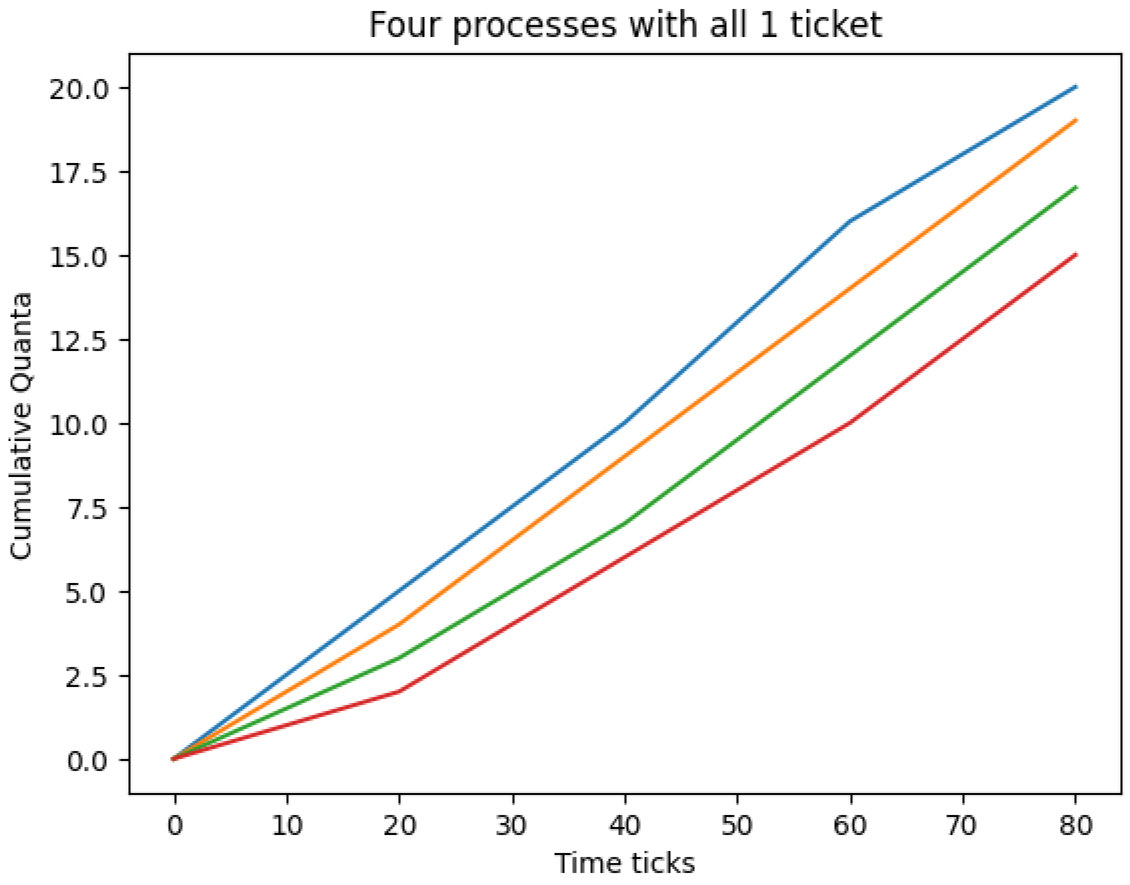


```c
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();

  c->proc = 0;
  for(;;){
    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();
    #if defined (LOTTERY)
    // printf("Going into lottery\n");
    lot_sched();
    #elif defined (STRIDE)
    //
    stride_sched();
    #else
    // printf("Going into normal round robin\n");
    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        // Switch to chosen process.  It is the process's job
        // to release its lock and then reacquire it
        // before jumping back to us.
        p->state = RUNNING;
        p->pticks = p->pticks + 1;
        c->proc = p;
        swtch(&c->context, &p->context);

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

Experiment figures:

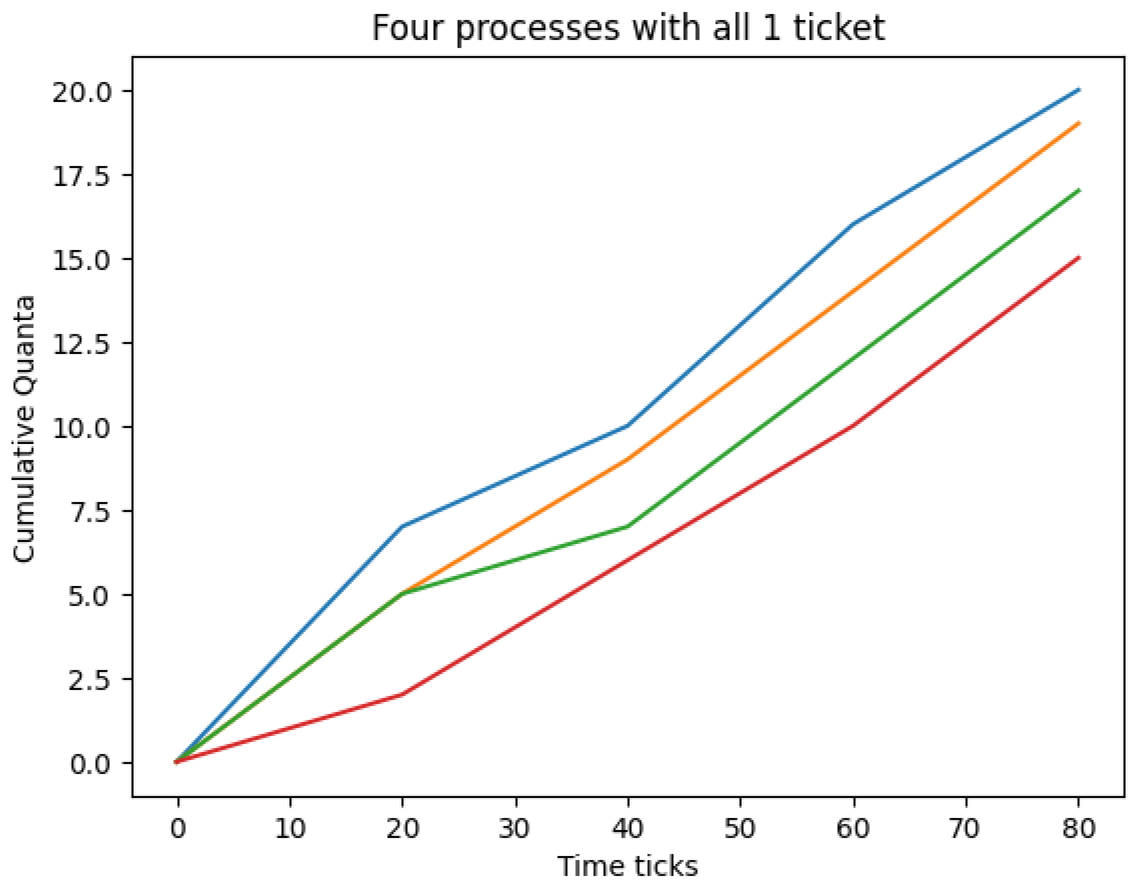
Lottery scheduling plots for 8:4:2:1 allocation and 1:1:1:1 allocation
Allocation by randomized lottery scheduler shows significant variabilit

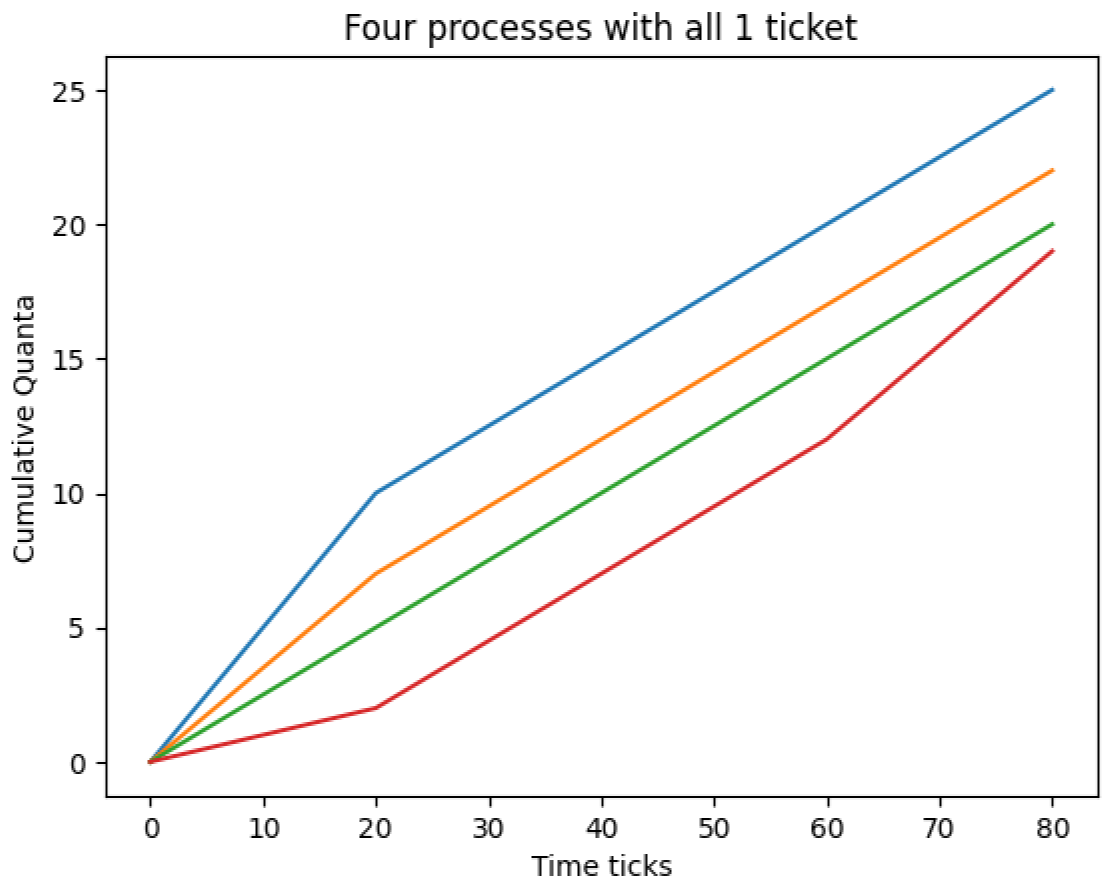




Stride scheduling:

Stride scheduling plots for 8:4:2:1 allocation and 1:1:1:1 allocation
Allocation by deterministic stride scheduler exhibits close to precise periodic behavior


Four processes with all 1 ticket
Cumulative Quanta
Time ticks
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
0
10
20
30
40
50
60
70
80


Four processes with all 1 ticket
Cumulative Quanta
Time ticks
0
5
10
15
20
25
0
10
20
30
40
50
60
70
80