Homework #3

Scheduling

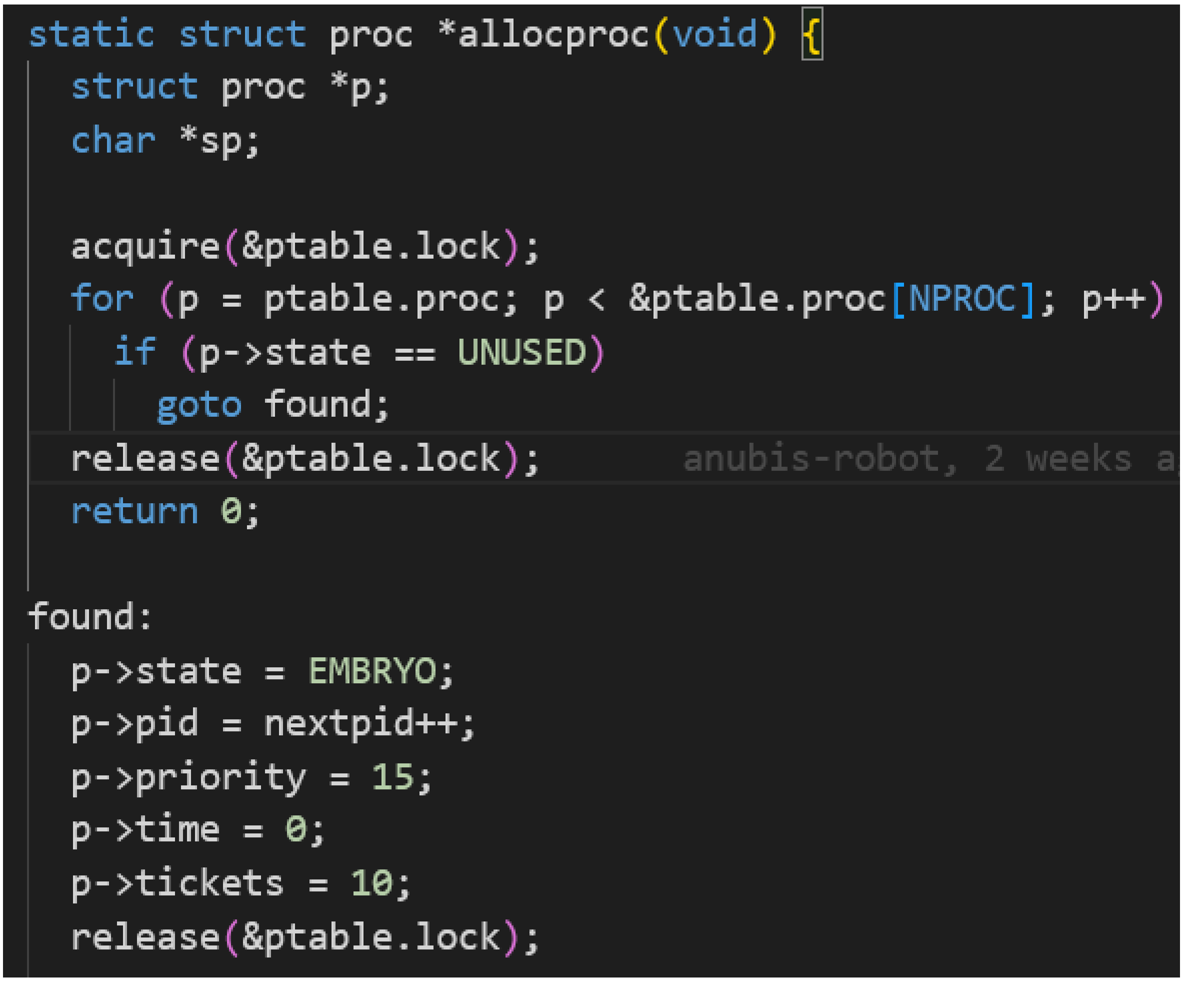
Part 1 - Nice (20 Points)

In order to implement the nice system call we have to make changes to the following files:

> 1. syscall.h
> 2. syscall.c
> 3. user.h
> 4. usys.S
> 5. sysproc.c
> 6. defs.h
> 7. proc.h
> 8. proc.c

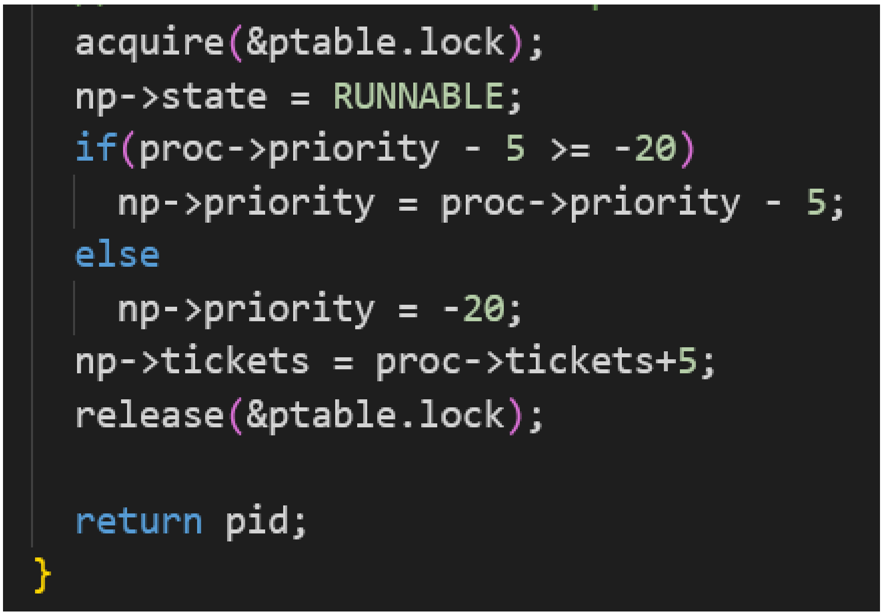
The syscall.h file maintains a table of all the system calls associated by a number. We first define our call in that file. We then add our call to the syscall.c file. We then give it a prototype in user.h file. Next we add our call in the usys.S file. This file interacts with the hardware of the system. We then define our function inside sysproc.c which calls our function defined in proc.c . In order to define our function inside proc.c we have to declare it in defs.h . Lastly, we change the process structure inside proc.h and add a new attribute called priority to every process.

In the allocproc function in proc.c file, I am initializing the priority of every process to 10. Inside the fork function I am assigning the priority of child process as priority_of_parent - 5.



```
static struct proc *allocproc(void) {
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if (p->state == UNUSED)
      goto found;
  release(&ptable.lock);       anubis-robot, 2 weeks a
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->priority = 15;
  p->time = 0;
  p->tickets = 10;
  release(&ptable.lock);
```

Below is the screenshot from fork function. I am also checking if subtracting 5 from the parent priority goes below -20. If the does then I set the priority to -20.

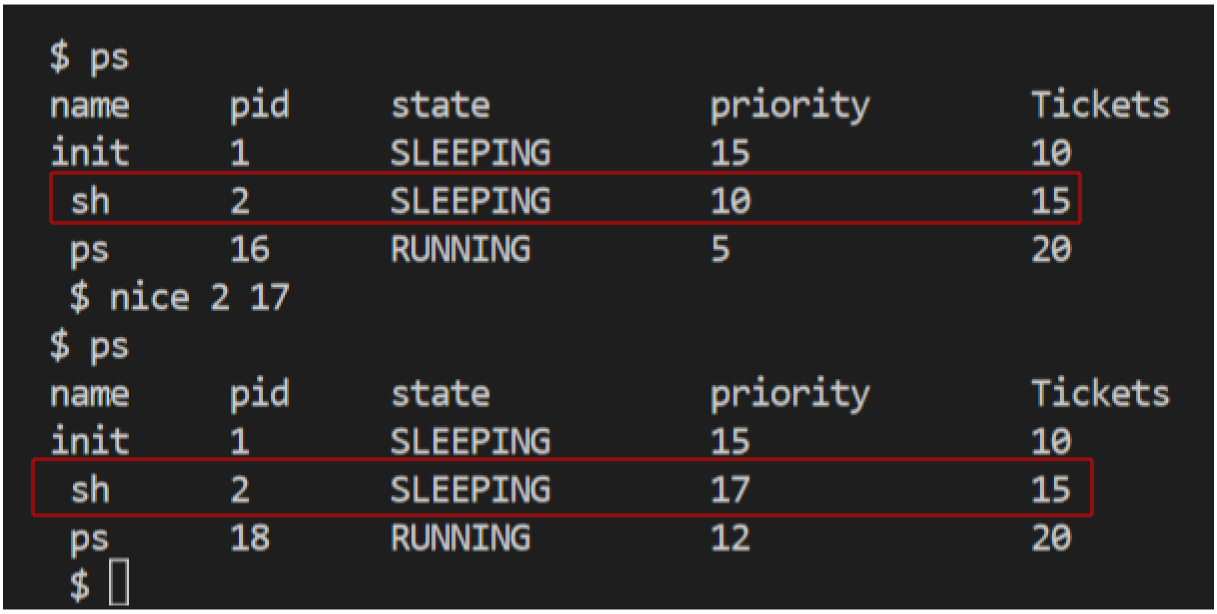


```
  acquire(&ptable.lock);
  np->state = RUNNABLE;
  if(proc->priority - 5 >= -20)
    np->priority = proc->priority - 5;
  else
    np->priority = -20;
  np->tickets = proc->tickets+5;
  release(&ptable.lock);

  return pid;
}
```

In order to verify the function of my nice system call, I have also implemented ps system call.

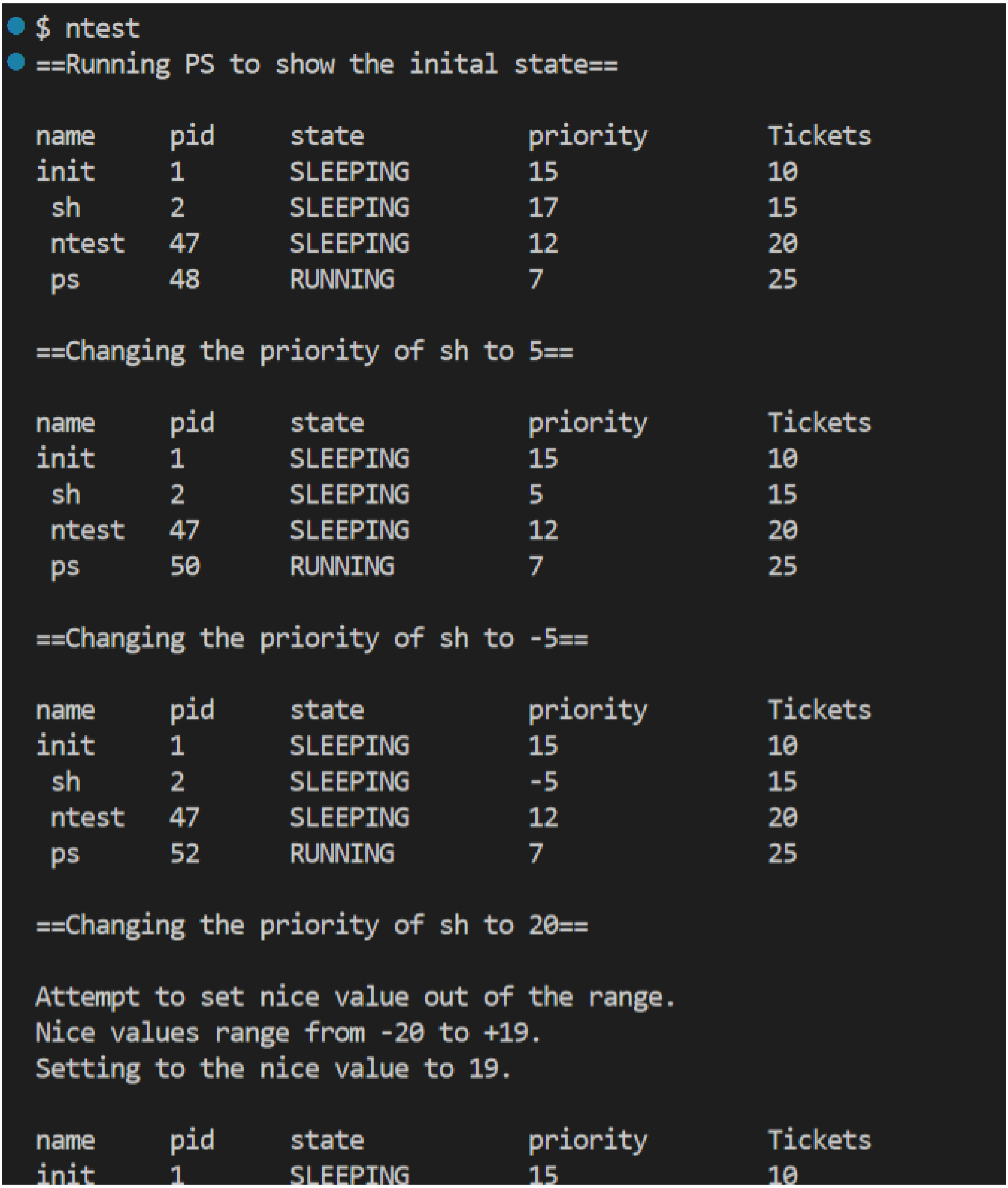
Below screenshot shows the working of the nice command. In the output of 1st ps command we can see that since sh is a child process of init, it has a priority of 15-5=10 and since ps is the child of sh it has priority of 5.

After we run the nice command we can see the priority of sh was changed from 10 to 17.



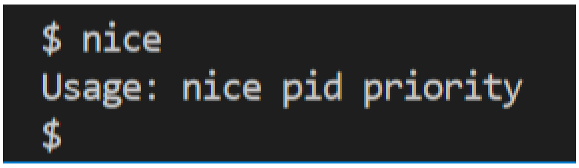
```
$ ps
name     pid     state          priority       Tickets
init     1       SLEEPING       15             10
 sh      2       SLEEPING       10             15
 ps      16      RUNNING        5              20
 $ nice 2 17
$ ps
name     pid     state          priority       Tickets
init     1       SLEEPING       15             10
 sh      2       SLEEPING       17             15
 ps      18      RUNNING        12             20
 $ 
```

In order to further test the nice system call, you can run the command ntest (code link: here) from the command line. It is a user program that has all the test cases for nice.



```
$ ntest
==Running PS to show the inital state==

name     pid     state           priority        Tickets
init     1       SLEEPING        15              10
 sh      2       SLEEPING        17              15
 ntest   47      SLEEPING        12              20
 ps      48      RUNNING         7               25

==Changing the priority of sh to 5==

name     pid     state           priority        Tickets
init     1       SLEEPING        15              10
 sh      2       SLEEPING        5               15
 ntest   47      SLEEPING        12              20
 ps      50      RUNNING         7               25

==Changing the priority of sh to -5==

name     pid     state           priority        Tickets
init     1       SLEEPING        15              10
 sh      2       SLEEPING        -5              15
 ntest   47      SLEEPING        12              20
 ps      52      RUNNING         7               25

==Changing the priority of sh to 20==

Attempt to set nice value out of the range.
Nice values range from -20 to +19.
Setting to the nice value to 19.

name     pid     state           priority        Tickets
init     1       SLEEPING        15              10
```

If you wish to test manually, we can always use the nice command from the command line (code link: here).



```
$ nice
Usage: nice pid priority
$
```

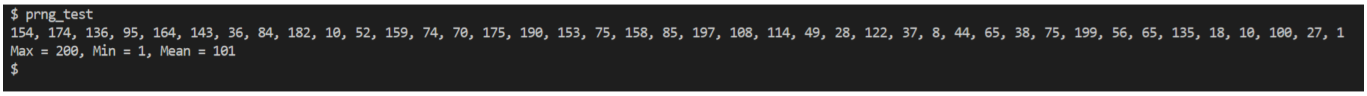
Part 2 - Random Number Generator (20 Points)

In order to implement the prng I have chosen to use the XORSHIFT64s. The source code for the prng can be found on here.

In order to then limit the values up to a range, I paired the prng with a division based rejection technique. This will always give us an answer between 1 and the max value that the program takes as in input.

The prng is written in prng.c file (code link: here) and I also created a header file called prng.h (code link: here)

In order to test the prng, I wrote a user program that run prng in a loop for 900 times and calculated the min, max and mean of all the values. The test code is inside prng_test.c file (code link: here).



```
$ prng_test
154, 174, 136, 95, 164, 143, 36, 84, 182, 10, 52, 159, 74, 70, 175, 190, 153, 75, 158, 85, 197, 108, 114, 49, 28, 122, 37, 8, 44, 65, 38, 75, 199, 56, 65, 135, 18, 10, 100, 27, 1
Max = 200, Min = 1, Mean = 101
$
```

Part 3 - Scheduler (60 Points)

Since lottery scheduling depends on the number of tickets each process has, I again changed the process structure inside proc.h and added two new attributes - tickets, time.

The tickets attribute determines the number of tickets each process has and the time attribute determines the number of times a process wins the lottery.

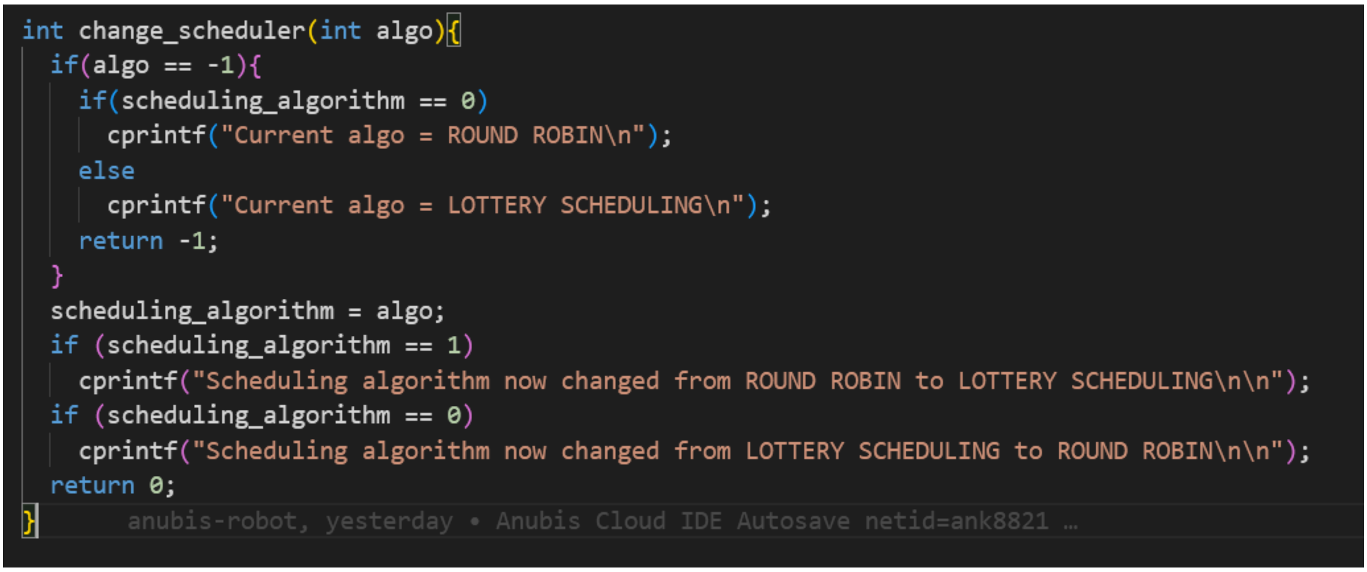
Again, in the allocproc function inside proc.c I have initialized the tickets for each process to 10 and the time for each process to 0. Inside the fork function, I have initialized the tickets for a child process as tickets_of_parent + 5.

In order to implement lottery scheduling, a major change was required inside the scheduler function in proc.c. Once the outer loop begins execution, we acquire the ptable and using a for loop count the total number of tickets of all the RUNNABLE processes. We then choose the lottery ticket using our prng and pass the max value as the total number of tickets. We then loop through all the processes in RUNNABLE state and store the sum of tickets for each process in a count variable. When the count becomes greater than the lottery ticket we execute the process our pointer is pointing at.

I then increment the time attribute by one. This will count the number of times the process won the lottery i.e., the number of times the process was in a RUNNING state.

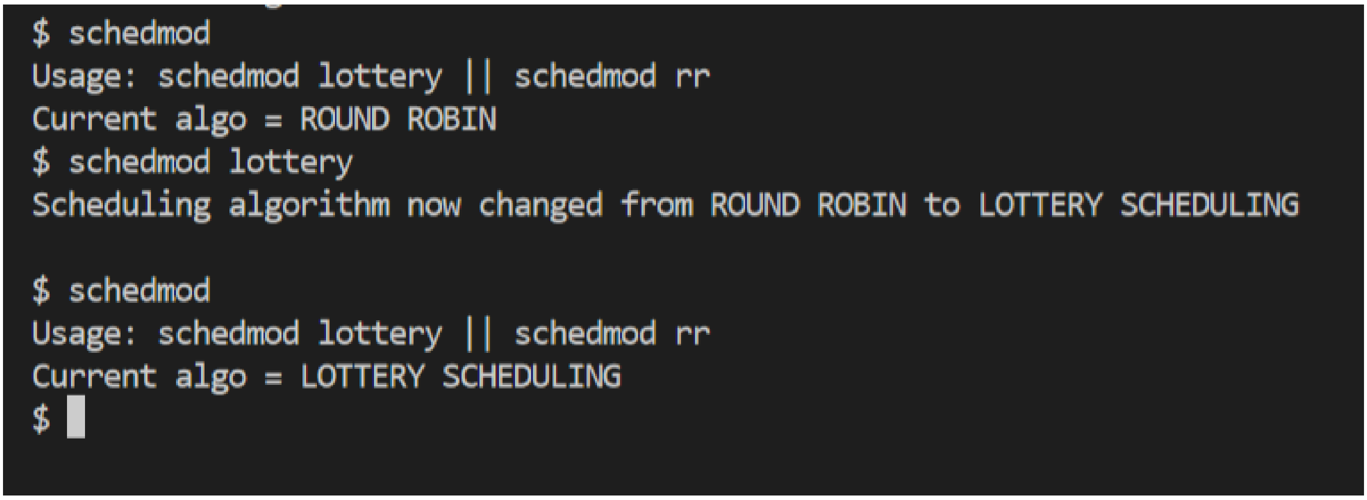
In order to implement both Round Robin and Lottery Scheduling, I have implement a system call called change_scheduler. It will set/unset a variable in proc.c that will track the scheduling policy.

Below is the code for the system call.



```c
int change_scheduler(int algo){
  if(algo == -1){
    if(scheduling_algorithm == 0)
      cprintf("Current algo = ROUND ROBIN\n");
    else
      cprintf("Current algo = LOTTERY SCHEDULING\n");
    return -1;
  }
  scheduling_algorithm = algo;
  if (scheduling_algorithm == 1)
    cprintf("Scheduling algorithm now changed from ROUND ROBIN to LOTTERY SCHEDULING\n\n");
  if (scheduling_algorithm == 0)
    cprintf("Scheduling algorithm now changed from LOTTERY SCHEDULING to ROUND ROBIN\n\n");
  return 0;
}       anubis-robot, yesterday • Anubis Cloud IDE Autosave netid=ank8821 …
```

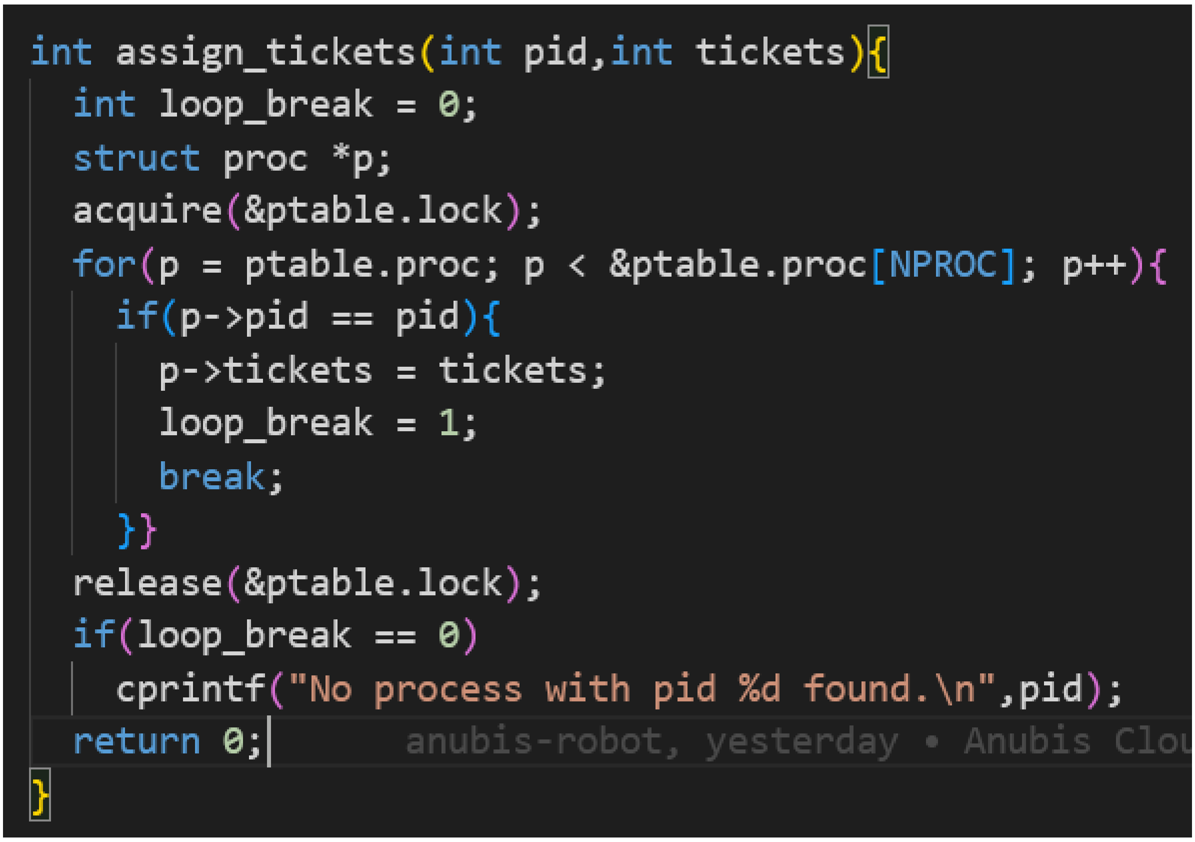
I have also written a user program called schedmod (code link: here) that will call the system call and change the policy.



```
$ schedmod
Usage: schedmod lottery || schedmod rr
Current algo = ROUND ROBIN
$ schedmod lottery
Scheduling algorithm now changed from ROUND ROBIN to LOTTERY SCHEDULING

$ schedmod
Usage: schedmod lottery || schedmod rr
Current algo = LOTTERY SCHEDULING
$ 
```

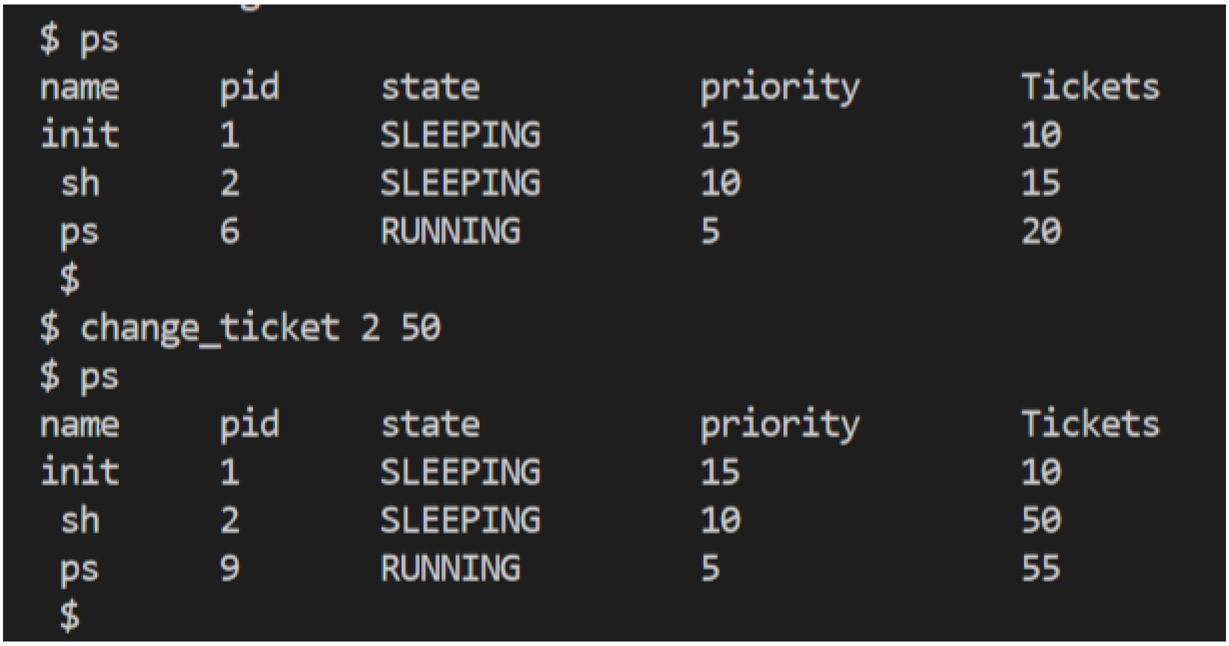
NOTE: Round Robin the default scheduling policy. You will have to run schedmod lottery after make qemu to change the policy.

I have also written a system call to change the number of tickets assigned to a process.



```c
int assign_tickets(int pid,int tickets){
  int loop_break = 0;
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->tickets = tickets;
      loop_break = 1;
      break;
    }}
  release(&ptable.lock);
  if(loop_break == 0)
    cprintf("No process with pid %d found.\n",pid);
  return 0;      anubis-robot, yesterday • Anubis Clou
}
```

The user can use the program change_ticket (code link: here) to change the tickets of a process. It takes two inputs, pid and the number of tickets.



```
$ ps
name     pid     state          priority        Tickets
init     1       SLEEPING       15              10
 sh      2       SLEEPING       10              15
 ps      6       RUNNING        5               20
 $
$ change_ticket 2 50
$ ps
name     pid     state          priority        Tickets
init     1       SLEEPING       15              10
 sh      2       SLEEPING       10              50
 ps      9       RUNNING        5               55
 $
```

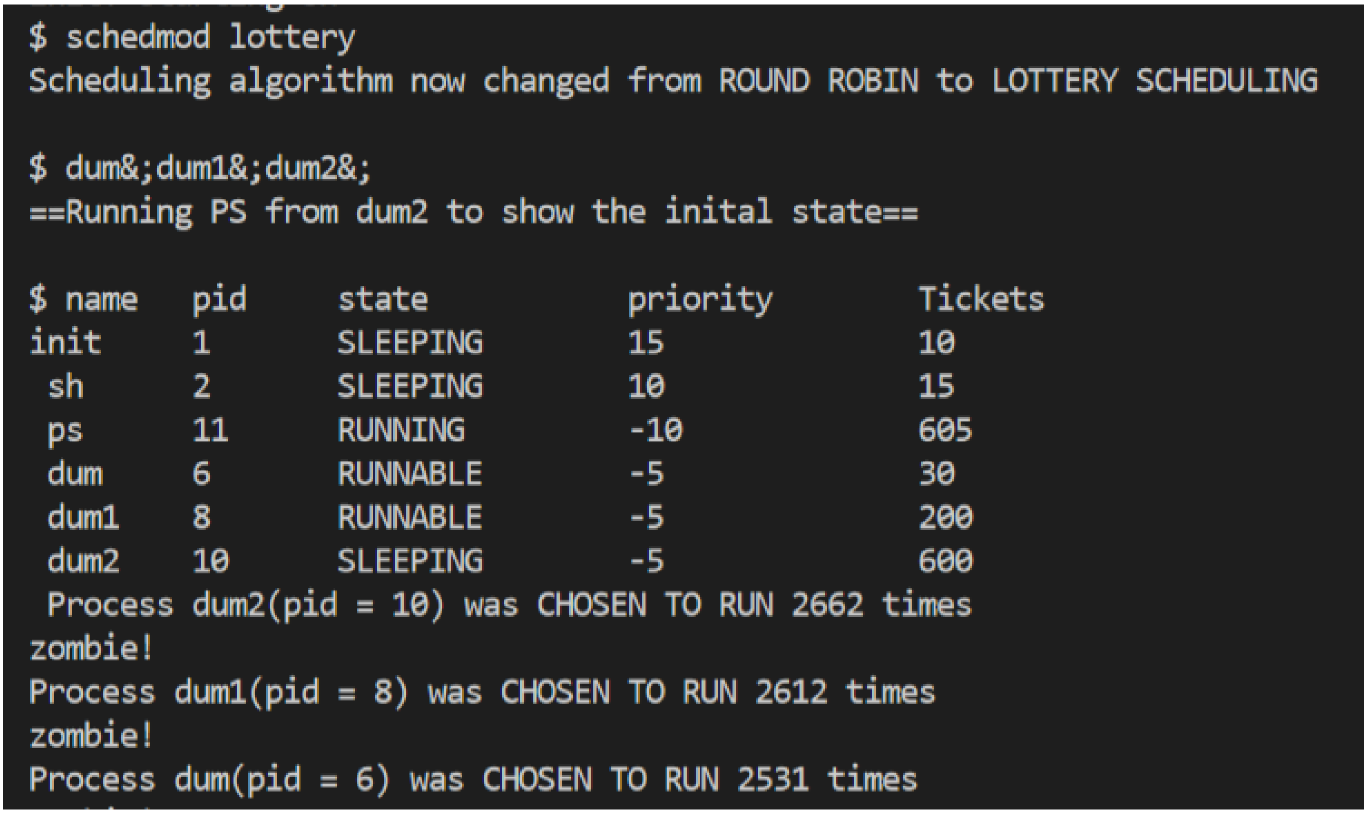
In the image above we can see that the number of tickets of sh changed from 15 to 50.

In order to test the scheduling, I have written 3 user programs that do normal mathematical computation. The user programs are dum (code link: here), dum1 (code link: here) and dum2 (code link: here). Running dum/dum1/dum2 -h will show the usage.

TEST CASE 1 - Processes have different tickets

Commands to run:

1. schedmod lottery
2. dum&;dum1&;dum2&;



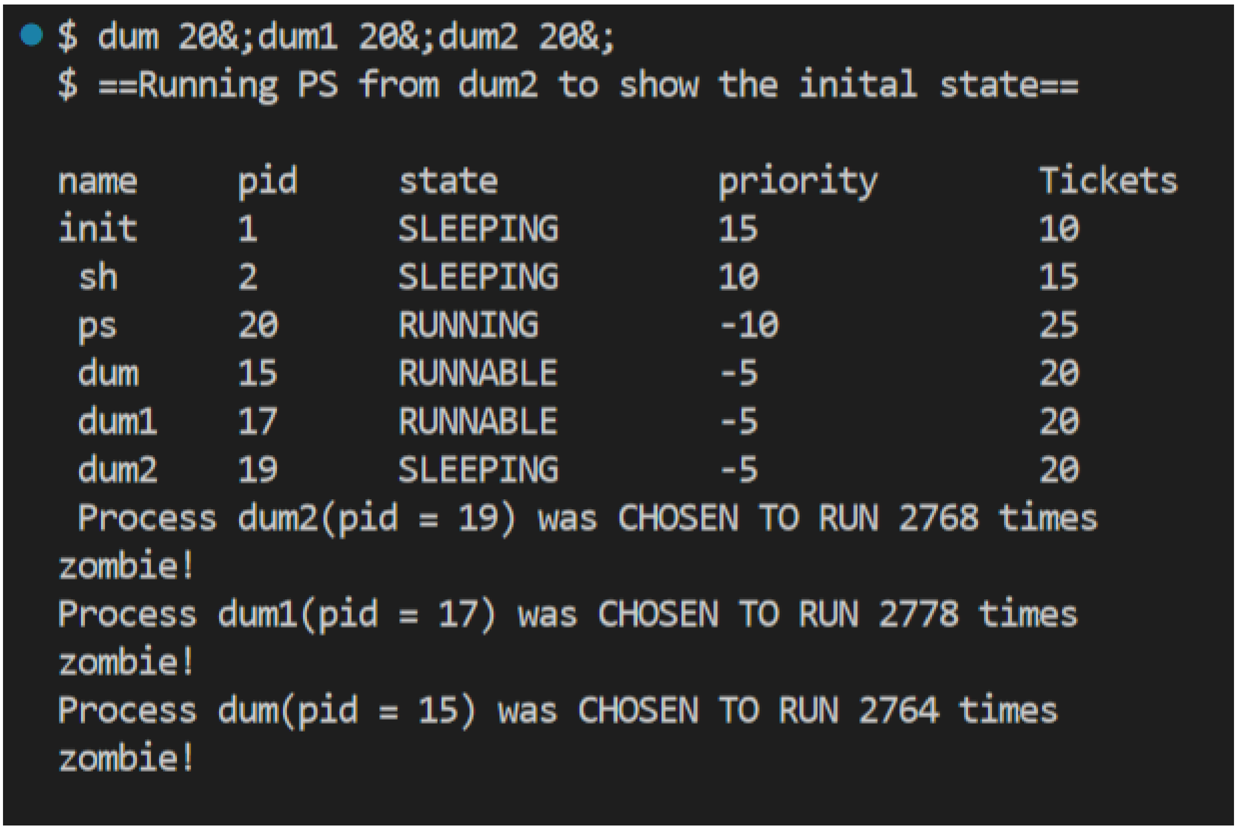
```
$ schedmod lottery
Scheduling algorithm now changed from ROUND ROBIN to LOTTERY SCHEDULING

$ dum&;dum1&;dum2&;
==Running PS from dum2 to show the inital state==

$ name   pid     state          priority        Tickets
init     1       SLEEPING       15              10
 sh      2       SLEEPING       10              15
 ps      11      RUNNING        -10             605
 dum     6       RUNNABLE       -5              30
 dum1    8       RUNNABLE       -5              200
 dum2    10      SLEEPING       -5              600
 Process dum2(pid = 10) was CHOSEN TO RUN 2662 times
zombie!
Process dum1(pid = 8) was CHOSEN TO RUN 2612 times
zombie!
Process dum(pid = 6) was CHOSEN TO RUN 2531 times
```

We can see that the process with more tickets was chosen more number of times.

TEST CASE 2 - Processes have same tickets

Commands to run:

1. dum 20&;dum1 20&;dum2 20&;



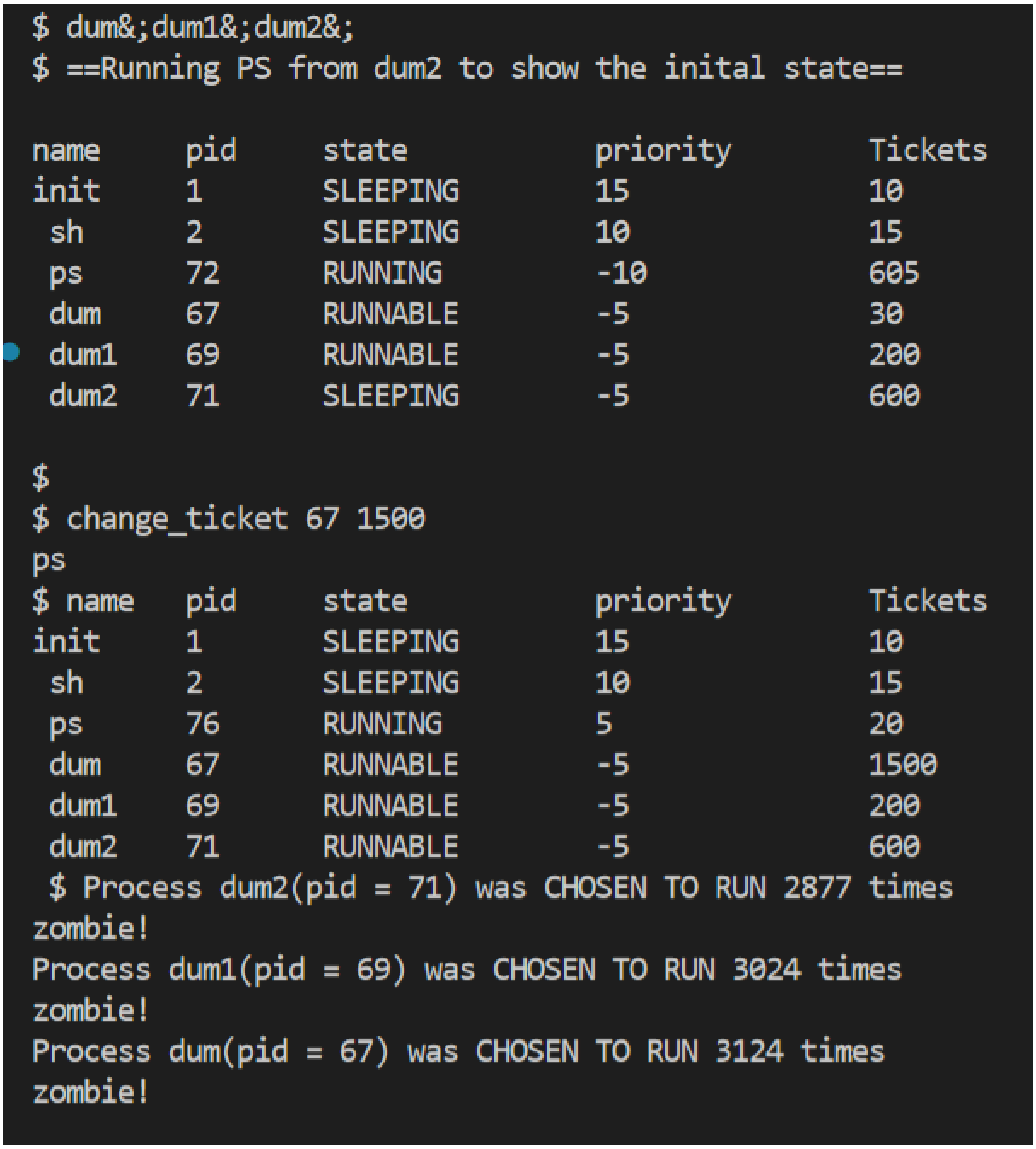
```
$ dum 20&;dum1 20&;dum2 20&;
$ ==Running PS from dum2 to show the inital state==

name     pid      state           priority        Tickets
init     1        SLEEPING        15              10
 sh      2        SLEEPING        10              15
 ps      20       RUNNING         -10             25
 dum     15       RUNNABLE        -5              20
 dum1    17       RUNNABLE        -5              20
 dum2    19       SLEEPING        -5              20
 Process dum2(pid = 19) was CHOSEN TO RUN 2768 times
zombie!
Process dum1(pid = 17) was CHOSEN TO RUN 2778 times
zombie!
Process dum(pid = 15) was CHOSEN TO RUN 2764 times
zombie!
```

Here, since the process have same number of tickets they were all chosen to run for a similar number of times.

TEST CASE 3 - Changing the tickets of a process manually

Commands to run:

1. dum&;dum1&;dum2&;
2. change_ticket 1500



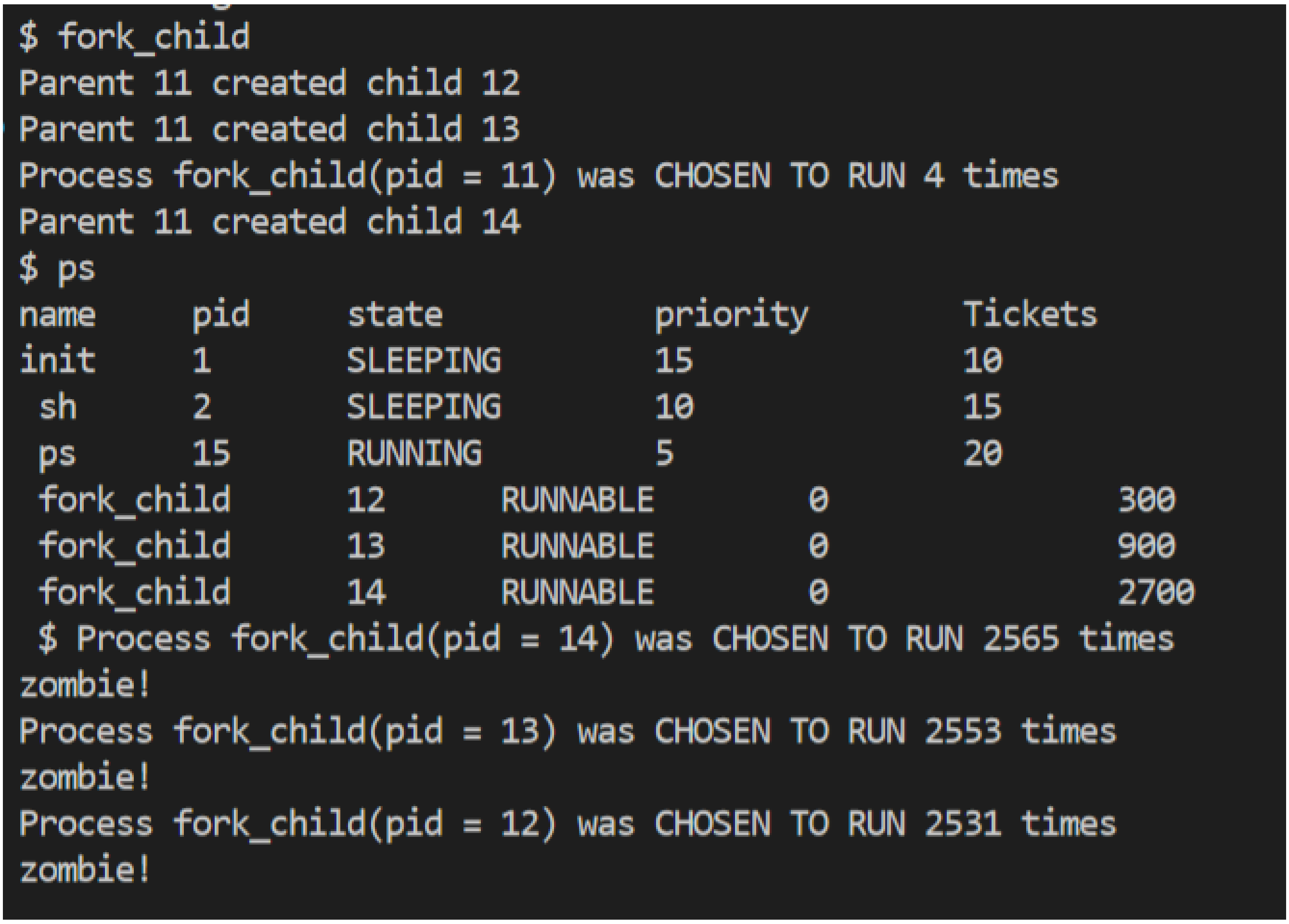
```
$ dum&;dum1&;dum2&;
$ ==Running PS from dum2 to show the inital state==

name     pid      state           priority        Tickets
init     1        SLEEPING        15              10
 sh      2        SLEEPING        10              15
 ps      72       RUNNING         -10             605
 dum     67       RUNNABLE        -5              30
 dum1    69       RUNNABLE        -5              200
 dum2    71       SLEEPING        -5              600

$
$ change_ticket 67 1500
ps
$ name   pid      state           priority        Tickets
init     1        SLEEPING        15              10
 sh      2        SLEEPING        10              15
 ps      76       RUNNING         5               20
 dum     67       RUNNABLE        -5              1500
 dum1    69       RUNNABLE        -5              200
 dum2    71       RUNNABLE        -5              600
 $ Process dum2(pid = 71) was CHOSEN TO RUN 2877 times
zombie!
Process dum1(pid = 69) was CHOSEN TO RUN 3024 times
zombie!
Process dum(pid = 67) was CHOSEN TO RUN 3124 times
zombie!
```

In the first ps output we can see that dum had 30 tickets. Once we changed the number of tickets for dum to 1500 as seen in the second ps output we can see that dum was chosen to run more number of times than dum1 and dum2.

TEST CASE 4 - Forking multiple children with different tickets

Commands to run:
1. fork_child 2. ps

This program will fork 3 children with tickets increasing by a factor of 3.(code link: here)

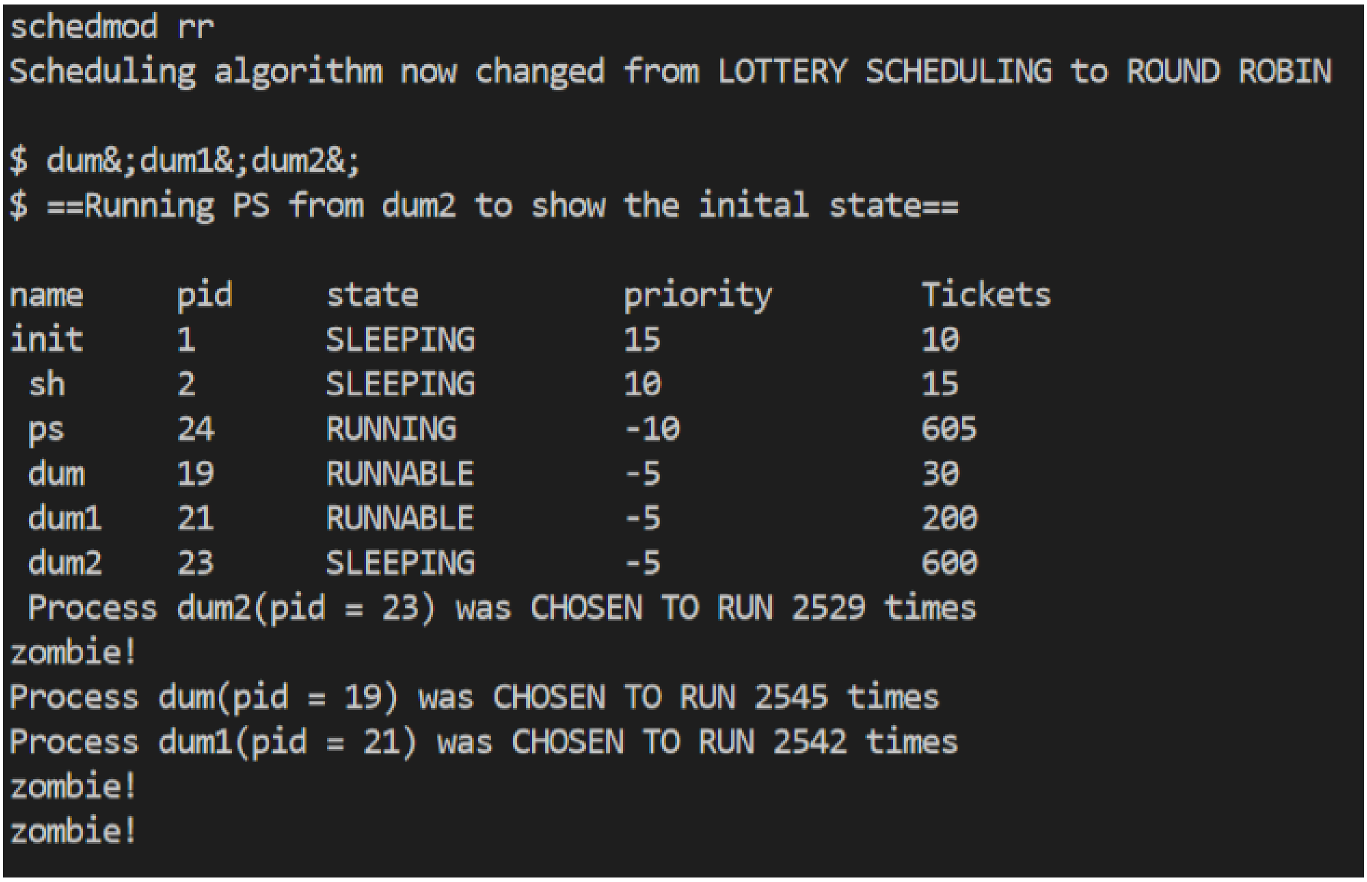


```
$ fork_child
Parent 11 created child 12
Parent 11 created child 13
Process fork_child(pid = 11) was CHOSEN TO RUN 4 times
Parent 11 created child 14
$ ps
name     pid     state           priority        Tickets
init     1       SLEEPING        15              10
 sh      2       SLEEPING        10              15
 ps      15      RUNNING         5               20
 fork_child      12      RUNNABLE        0               300
 fork_child      13      RUNNABLE        0               900
 fork_child      14      RUNNABLE        0               2700
 $ Process fork_child(pid = 14) was CHOSEN TO RUN 2565 times
zombie!
Process fork_child(pid = 13) was CHOSEN TO RUN 2553 times
zombie!
Process fork_child(pid = 12) was CHOSEN TO RUN 2531 times
zombie!
```

TEST CASE 5 - Running program with Round robin to see the difference

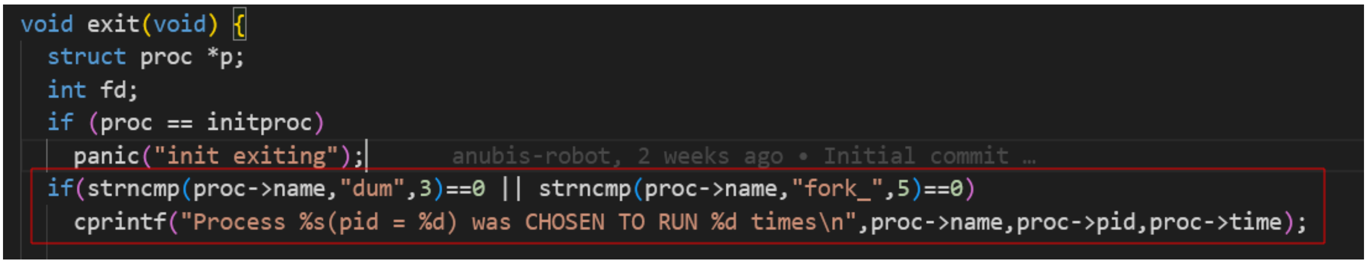
Commands to run:
1. schedmod rr
2. dum&;dum1&;dum2&;

Here, the number of tickets a process has should not matter. The number of times each process get selected to run is almost the same.



```
schedmod rr
Scheduling algorithm now changed from LOTTERY SCHEDULING to ROUND ROBIN

$ dum&;dum1&;dum2&;
$ ==Running PS from dum2 to show the inital state==

name     pid     state          priority       Tickets
init     1       SLEEPING       15             10
 sh      2       SLEEPING       10             15
 ps      24      RUNNING        -10            605
 dum     19      RUNNABLE       -5             30
 dum1    21      RUNNABLE       -5             200
 dum2    23      SLEEPING       -5             600
 Process dum2(pid = 23) was CHOSEN TO RUN 2529 times
zombie!
Process dum(pid = 19) was CHOSEN TO RUN 2545 times
Process dum1(pid = 21) was CHOSEN TO RUN 2542 times
zombie!
zombie!
```

Lastly, in order to print the information about the number of times the process was chosen, inside the exit function of proc.c I added 2 lines to show only the out for the test programs and not all the programs.



```
void exit(void) {
  struct proc *p;
  int fd;
  if (proc == initproc)
    panic("init exiting");        anubis-robot, 2 weeks ago • Initial commit …
  if(strncmp(proc->name,"dum",3)==0 || strncmp(proc->name,"fork_",5)==0)
    cprintf("Process %s(pid = %d) was CHOSEN TO RUN %d times\n",proc->name,proc->pid,proc->time);
```

If you wish to see the output for every program, comment output the if statement and leave the cprintf statement as it is.