

Explanations

Part-1 Implementing Nice

Implementation Steps:

1. User Space (*nice.c*): In the user space, we have created a program named ``nice``. This program takes two command-line arguments: a process ID (``pid``) and a priority value. It first validates the provided priority value to ensure it falls within the acceptable range (0-20). If the priority is out of range, the program displays an error message and exits. Otherwise, it proceeds to set the priority of the specified process using the ``chpr`` function.

(The above implementation code is included in `nice.c`).

2. Kernel Space (*proc.c*): We have implemented ``chpr`` function in the kernel space to facilitate the modification of process priorities. To ensure the integrity of the process table, it acquires the process table lock. It then iterates through the process table to locate the process with the specified ``pid``. Once the process is found, the function updates its priority to the new value. Finally, the process table lock is released, and the function returns the ``pid``.

((The above implementation code is included at the end of `proc.c`).

3. System Call (*sysproc.c*): To create a bridge between user space and kernel space, we have defined a system call named ``sys_chpr``. This system call retrieves the process ID and priority from user space using the ``argint`` function. If any of these steps fail, it returns an error code. However, if the process ID and priority are successfully retrieved, it invokes the ``chpr`` function in the kernel space. The result of the ``chpr`` function is returned to the user space.

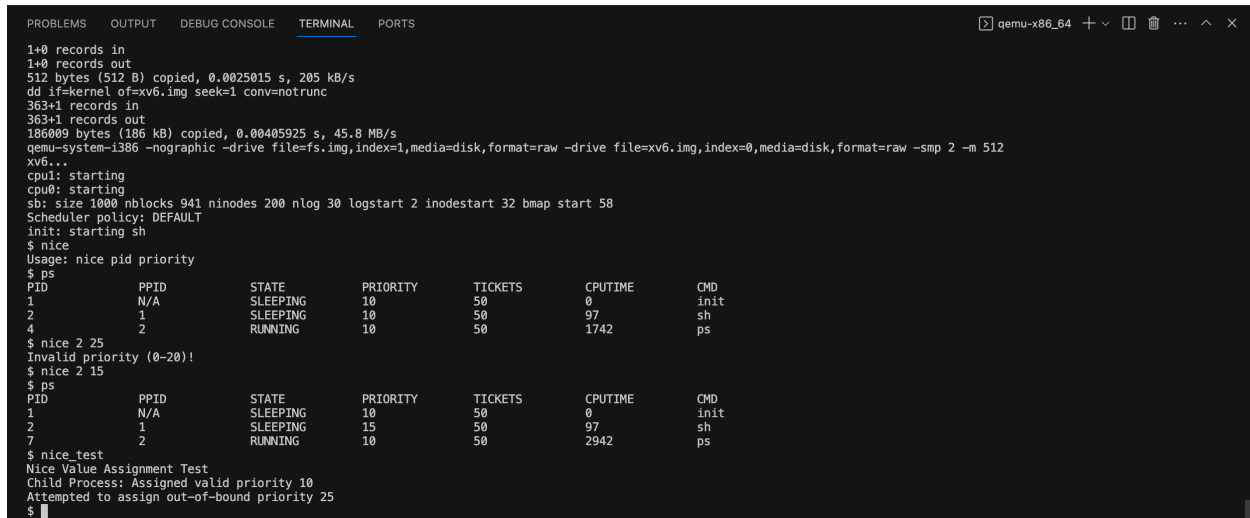
Data Structures:

To support this implementation, we have introduced an integer variable named ``priority`` in the ``struct proc`` in the ``proc.h`` header file. This variable store and manages the priority of each process in xv6.

System Call Declaration:

For the system call to be recognized and utilized by user-level programs, we have declared the necessary changes in various header and source files, including `defs.h`, `user.h`, `syscall.h`, `syscall.c`, `Usys.s`, and others.

Test cases and screenshots:



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
qemu-x86_64 + - - - - -

1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.0025015 s, 205 kB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
363+1 records in
363+1 records out
186009 bytes (186 kB) copied, 0.00405925 s, 45.8 MB/s
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
xv6...
cpu1: starting
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
Scheduler policy: DEFAULT
init: starting sh
$ nice
Usage: nice pid priority
$ ps
PID          PPID        STATE        PRIORITY    TICKETS     CPUTIME     CMD
1             N/A         SLEEPING     10          50          0           init
2             1           SLEEPING     10          50          97          sh
4             2           RUNNING      10          50          1742        ps
$ nice 2 25
Invalid priority (0-20)!
$ nice 2 15
$ ps
PID          PPID        STATE        PRIORITY    TICKETS     CPUTIME     CMD
1             N/A         SLEEPING     10          50          0           init
2             1           SLEEPING     15          50          97          sh
7             2           RUNNING      10          50          2942        ps
$ nice_test
Nice Value Assignment Test
Child Process: Assigned valid priority 10
Attempted to assign out-of-bound priority 25
$
```

Explanation of testcases and screenshot:

1. First we tried to use by just pressing nice and enter.
OUTPUT: Usage nice pid priority (states we need to use nice call in the way it stated)
2. We tried to change the priority of the process 2 with value 25
OUTPUT: Invalid priority (as we have defined its range from 0-20)
3. We tried to assign the process 2 with priority 15
OUTPUT: successfully changed the priority of the process 2.
4. Sample testcase where we assign nice values dynamically in the nice_test program.

Part-2: Implementing PRNG

XORShift-Based PRNG:(included this in proc.c)

In our system, we have introduced a Pseudo-Random Number Generator (PRNG) based on the XORShift algorithm to enhance the randomness and unpredictability in the lottery scheduler. This PRNG is a crucial addition to our scheduler, as it helps distribute CPU time among processes in a more unpredictable and equitable manner, ultimately improving the fairness of process scheduling.

xorshift_seed() Function:

We have implemented the **xorshift_seed()** function, which is responsible for initializing the state of the XORShift PRNG. To create a new seed, we combine the current system uptime with a counter. This new seed is used to set the initial values of the **xorshift_state** array.

Usage of uptime() to generate seed value to increase randomness:

In our system, we have leveraged the **uptime()** function to generate seed values for our XORShift-based Pseudo-Random Number Generator (PRNG). By incorporating the current system uptime as part of the seed generation process, we introduce a level of unpredictability and real-world entropy into our PRNG. This approach ensures that each time the PRNG is initialized, a unique seed is generated by combining the uptime with a counter. This unique seed enhances the fairness and equity of our lottery scheduler by producing distinct sequences of pseudo-random numbers in each PRNG session, effectively preventing any patterns or biases in process selection. This use of **uptime()** underscores our commitment to a more dynamic and responsive process scheduler, capable of adapting to real-world conditions and delivering fair and equitable process scheduling outcomes.

xorshift() Function:

The core of our PRNG is the **xorshift()** function, which leverages the XORShift algorithm. This algorithm applies bitwise operations to the state variables to generate relatively unpredictable pseudo-random values. By utilizing this algorithm, we have introduced a level of randomness that contributes to the equitable distribution of CPU time among processes.

***random()* Function:**

To make our PRNG more versatile, we have introduced the **random()** function. This function accepts a maximum value (**max**) as an argument and generates a random number between 0 and **max - 1**. It begins by calling **xorshift_seed()** to refresh the PRNG's seed, ensuring that each call to **random()** results in a different sequence of random numbers. The generated random value is then constrained to the range [0, **max - 1**].

Through the introduction of this XORShift-based PRNG, we have improved the overall fairness and unpredictability of our lottery scheduler, benefiting the scheduling of processes in our system.

SCREENSHOTS AND TESTCASES:

First testcase (random.c):

```
$ random 100
79
$ random 100
54
$ random 100
45
$ random 100
52
$ random 100
95
$
```

We have used XOR Based PRNG and we have declared function as `random(int max)` where `max` is the maximum value you can give in a range (0,max). This function generates random number within the range(0,100) when you execute **random 100** command.

OUTPUT: We can see that each time the random number it gives is different and not repetitive.

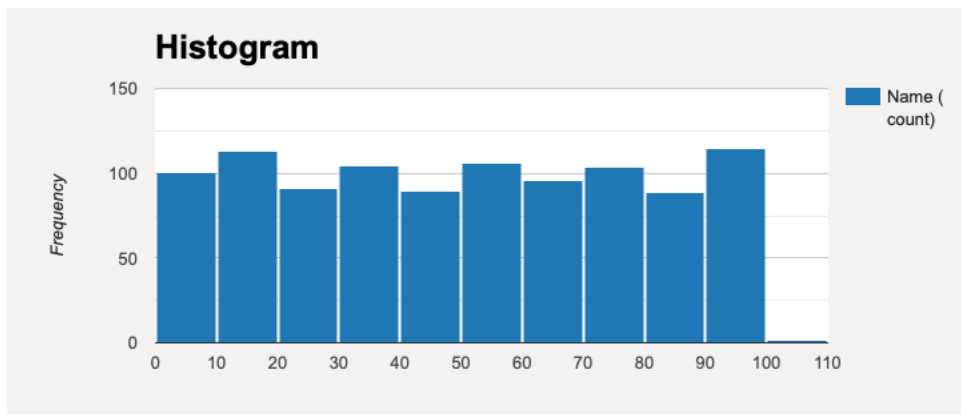
Random usage :

```
$ random_test
Usage: random_test <max>
$ random
Usage: random <max>
$
```

Second testcase(random_test.c): Running it for 1000 times between(0,100)

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
$ random_test 100
77
57
97
17
57
69
9
93
57
97
9
49
77
9
21
5
45
57
97
5
89
57
85
25
65
77
17
45
85
97
61
73
13
41
81
13
25
65
93
77
89
9
37
5
33
73
85
61
45
57
97
25
```

I have taken these values and used these values to create a histogram:



I have created a histogram for each 10 interval (0-10,10-20...etc)
Thus, I can say that the numbers are evenly distributed and not skewed.

Part-3: Lottery Scheduler Implementation

Implementation of the custom lottery scheduler and the use of a XORShift-based PRNG in your operating system:

Lottery Scheduler Implementation:

In our operating system, we have introduced a lottery scheduler to enhance process scheduling fairness and unpredictability.

1.xscheduler Variable Significance: The **xscheduler** variable plays a crucial role in our operating system, acting as a toggle between two distinct scheduling algorithms. When **xscheduler** is set to 1, the custom lottery scheduler takes charge. In contrast, when **xscheduler** is set to 0, the default round-robin scheduler comes into play, ensuring equitable process execution through a time-sliced mechanism.

2.Scheduler Function (``scheduler``):

- The core of our custom scheduler is the ``scheduler`` function in ``proc.c``, responsible for selecting the process to run.
- This function periodically selects a winner among runnable processes using a lottery-based approach.
- The key element of this scheduler is the allocation of "tickets" to each runnable process, where the number of tickets a process has is influenced by its "nice" value.

3. Ticket Allocation:

- The number of tickets allocated to a process is determined based on its "nice" value, which represents its priority.
- Processes with higher "nice" values receive fewer tickets, while those with lower "nice" values get more tickets, reflecting their relative priority.

4. Randomness and Unpredictability:

- To ensure unpredictability in process selection, we use a Pseudo-Random Number Generator (PRNG) based on the XORShift algorithm.
- The PRNG introduces randomness into the lottery, making it less predictable and more equitable.
- The PRNG is initialized with a unique seed, combining the current system uptime with a counter, ensuring different sequences of random numbers for each lottery session.

5. Total Tickets Calculation (``totalTickets``):

- The ``totalTickets`` function calculates the total number of tickets allocated to all runnable processes. This total is used to ensure that the lottery is conducted fairly.

6. User Influence via ``chtickets`` System Call:

- Users can influence their process's likelihood of winning the lottery through the ``chtickets`` system call.
- This system call allows users to set the number of tickets for a specific process, essentially adjusting their process's priority in the scheduling lottery.

XORShift-Based PRNG Implementation:

1. Seed Generation (``xorshift_seed``):

- The XORShift-based PRNG relies on a seed value to produce pseudo-random numbers.
- Our ``xorshift_seed`` function initializes the PRNG by combining the current system uptime with a counter.
- The unique seed ensures that each PRNG session produces different sequences of random numbers.

2. Random Number Generation (``xorshift``):

- The ``xorshift`` function is at the core of our PRNG, implementing the XORShift algorithm.
- It applies bitwise operations to the PRNG's state variables to generate relatively unpredictable pseudo-random values.

3. ``random`` Utility Function:

- To make our PRNG more versatile, we have implemented the ``random`` utility function.
- It takes a maximum value as an argument and generates a random number between 0 and ``max - 1``, enhancing its usability.

User-Level Ticket Management Utility:

1. ``ticket`` User-Level Utility (``tickets.c``):

- To empower users with control over process scheduling, we have developed the ``ticket`` user-level utility.
- Users can adjust the number of tickets for specific processes using this utility, thereby influencing their process's priority in the scheduling lottery.

The combination of a custom lottery scheduler with XORShift-based PRNG introduces fairness and unpredictability into our operating system's process scheduling. This approach accommodates varying process priorities, ensures equitable distribution of CPU time, and allows users to influence their process's scheduling behavior. The use of the system uptime in seed generation and a versatile PRNG utility enhances the system's adaptability to real-world conditions, contributing to the fairness and efficiency of process scheduling. Overall, our custom lottery scheduler serves as a valuable addition to our operating system, delivering a fair, dynamic, and responsive process scheduling mechanism.

SCREENSHOTS AND TESTCASES:

Assign tickets:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

$ tickets
Usage: ticket pid numtickets
$ ps
PID          PPID      STATE      PRIORITY    TICKETS    CPUTIME    CMD
1            N/A       SLEEPING   10          50         0          init
2            1         SLEEPING   10          50         38         sh
6            2         RUNNING    10          50         15426      ps
$ tickets 2 200
Tickets must be greater than zero and less than 101!
$ tickets 2 90
$ ps
PID          PPID      STATE      PRIORITY    TICKETS    CPUTIME    CMD
1            N/A       SLEEPING   10          50         0          init
2            1         SLEEPING   10          90         38         sh
9            2         RUNNING    10          50         16702      ps
$
```

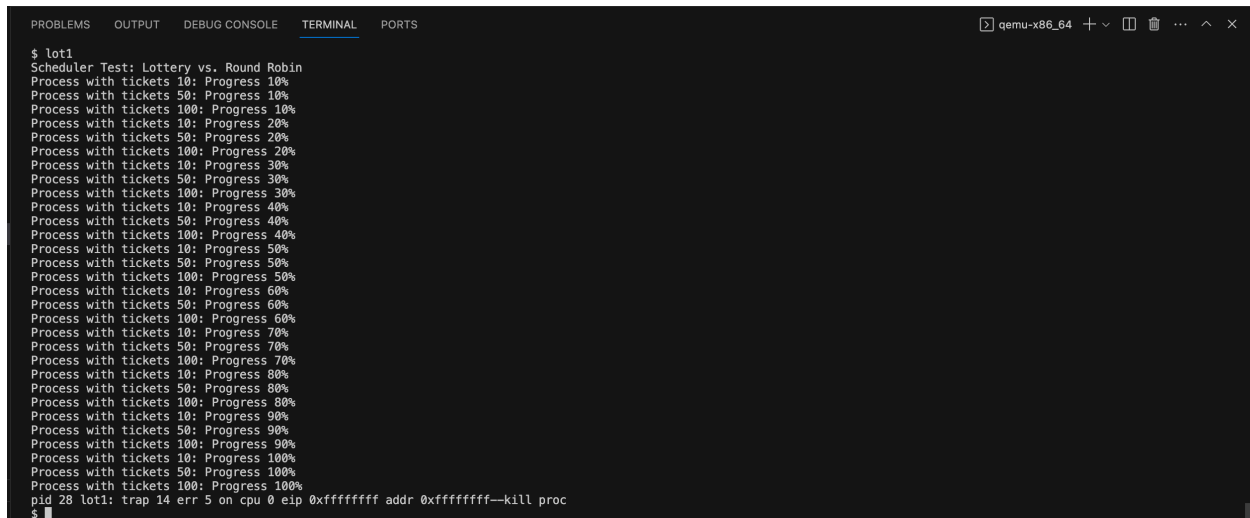
We have covered all the basic testcases for tickets as showed above.

Running testcase1(lot1.c) with default round robin scheduler:

Here we have assigned tickets to 3 different processes. Regardless of the tickets it shares the CPU time equally with each process. Thus, you can see CPU time is sliced equally and all process finished at the same time.

Value in param.h

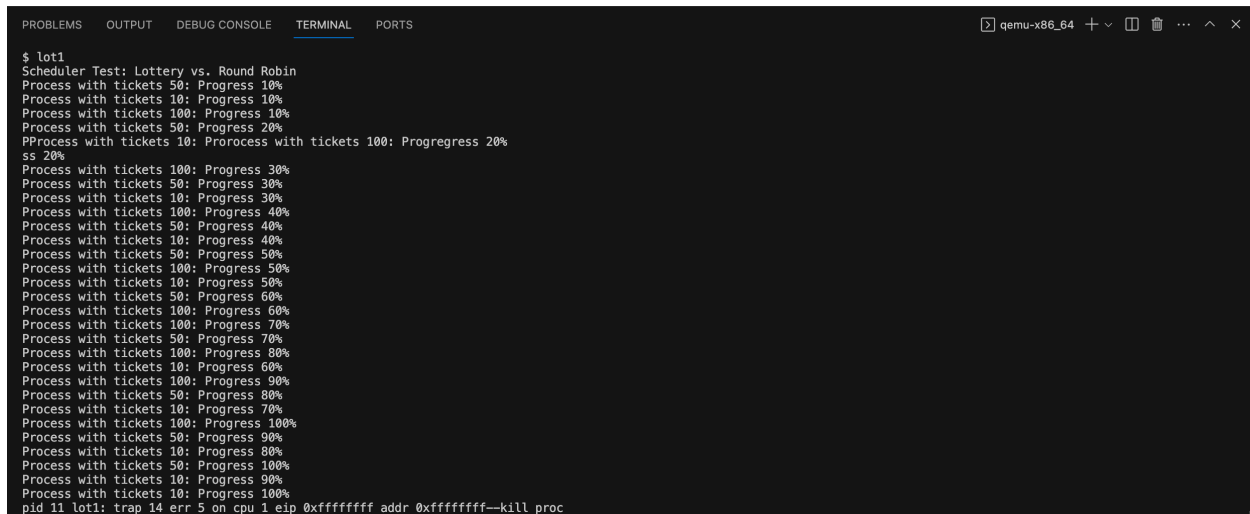
```
#define xscheduler 0
```



```
$ lot1
Scheduler Test: Lottery vs. Round Robin
Process with tickets 10: Progress 10%
Process with tickets 50: Progress 10%
Process with tickets 100: Progress 10%
Process with tickets 10: Progress 20%
Process with tickets 50: Progress 20%
Process with tickets 100: Progress 20%
Process with tickets 10: Progress 30%
Process with tickets 50: Progress 30%
Process with tickets 100: Progress 30%
Process with tickets 10: Progress 40%
Process with tickets 50: Progress 40%
Process with tickets 100: Progress 40%
Process with tickets 10: Progress 50%
Process with tickets 50: Progress 50%
Process with tickets 100: Progress 50%
Process with tickets 10: Progress 60%
Process with tickets 50: Progress 60%
Process with tickets 100: Progress 60%
Process with tickets 10: Progress 70%
Process with tickets 50: Progress 70%
Process with tickets 100: Progress 70%
Process with tickets 10: Progress 80%
Process with tickets 50: Progress 80%
Process with tickets 100: Progress 80%
Process with tickets 10: Progress 90%
Process with tickets 50: Progress 90%
Process with tickets 100: Progress 90%
Process with tickets 10: Progress 100%
Process with tickets 50: Progress 100%
Process with tickets 100: Progress 100%
pid 28 lot1: trap 14 err 5 on cpu 0 eip 0xffffffff addr 0xffffffff--kill proc
$
```

Run testcase1 with lottery scheduler:

```
#define xscheduler 1
```



```
$ lot1
Scheduler Test: Lottery vs. Round Robin
Process with tickets 50: Progress 10%
Process with tickets 10: Progress 10%
Process with tickets 100: Progress 10%
Process with tickets 50: Progress 20%
Process with tickets 10: Progress 20%
Process with tickets 100: Progress 20%
Process with tickets 100: Progress 30%
Process with tickets 50: Progress 30%
Process with tickets 10: Progress 30%
Process with tickets 100: Progress 40%
Process with tickets 50: Progress 40%
Process with tickets 10: Progress 40%
Process with tickets 50: Progress 50%
Process with tickets 100: Progress 50%
Process with tickets 10: Progress 50%
Process with tickets 50: Progress 60%
Process with tickets 100: Progress 60%
Process with tickets 100: Progress 70%
Process with tickets 50: Progress 70%
Process with tickets 100: Progress 80%
Process with tickets 10: Progress 60%
Process with tickets 100: Progress 90%
Process with tickets 50: Progress 80%
Process with tickets 10: Progress 70%
Process with tickets 100: Progress 100%
Process with tickets 50: Progress 90%
Process with tickets 10: Progress 80%
Process with tickets 50: Progress 100%
Process with tickets 10: Progress 90%
Process with tickets 10: Progress 100%
pid 11 lot1: trap 14 err 5 on cpu 1 eip 0xffffffff addr 0xffffffff--kill proc
$
```

We can see that process with more tickets has completed first and with 50 as second and 10 tickets in the last.

Test case 2: Dynamically allocating the tickets.(lot2)

In lottery scheduler:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
$ lot2
Lottery Scheduler Test: Dynamic Ticket Adjustment
Process 24 with 50 tickets: Progress 10%
Process 25 with 50 tickets: Progress 10%
Process 24 with 50 tickets: Progress 20%
Process 26 with 50 tickets: Progress 10%
Process 25 with 50 tickets: Progress 20%
Process 26 with 50 tickets: Progress 20%
Process 25 with 50 tickets: Progress 30%
Process 25: Increased tickets to 80
Process 26 with 50 tickets: Progress 30%
Process 26: Increased tickets to 80
Process 24 with 50 tickets: Progress 40%
Process 25 with 80 tickets: Progress 30%
Process 24: Increased tickets to 80
Process 26 with 80 tickets: Progress 40%
Process 25 with 80 tickets: Progress 50%
Process 26 with 80 tickets: Progress 50%
Process 24 with 80 tickets: Progress 40%
Process 25 with 80 tickets: Progress 60%
Process 25: Decreased tickets to 20
Process 26 with 80 tickets: Progress 60%
Process 26: Decreased tickets to 20
Process 25 with 20 tickets: Progress 70%
Process 24 with 20 tickets: Progress 70%
Process 26 with 20 tickets: Progress 50%
Process 25 with 20 tickets: Progress 80%
Process 26 with 20 tickets: Progress 80%
Process 24 with 20 tickets: Progress 90%
Process 25 with 20 tickets: Progress 60%
Process 24: Decreased tickets to 20
Process 26 with 20 tickets: Progress 90%
Process 26 with 20 tickets: Progress 100%
Process 25 with 20 tickets: Progress 100%
Process 24 with 20 tickets: Progress 70%
Process 24 with 20 tickets: Progress 80%
Process 24 with 20 tickets: Progress 90%
Process 24 with 20 tickets: Progress 100%
pid 23 lot2: trap 14 err 5 on cpu 1 eip 0xffffffff addr 0xffffffff--kill proc
$
```

We can see that once we change or assigned 80 tickets to each process dynamically when it completes 30% of the entire process, Probability of appearing processes with 80 tickets has appeared more and we wantedly decreased tickets to 20 once each process reaches to 60%.

No change in round robin Default scheduler regardless of tickets:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
$ lot2
Lottery Scheduler Test: Dynamic Ticket Adjustment
Process 37 with 50 tickets: Progress 10%
Process 38 with 50 tickets: Progress 10%
Process 39 with 50 tickets: Progress 10%
Process 37 with 50 tickets: Progress 20%
Process 38 with 50 tickets: Progress 20%
Process 39 with 50 tickets: Progress 20%
Process 37 with 50 tickets: Progress 30%
Process 37: Increased tickets to 80
Process 38 with 50 tickets: Progress 30%
Process 38: Increased tickets to 80
Process 39 with 50 tickets: Progress 30%
Process 39: Increased tickets to 80
Process 37 with 80 tickets: Progress 40%
Process 38 with 80 tickets: Progress 40%
Process 39 with 80 tickets: Progress 40%
Process 37 with 80 tickets: Progress 50%
Process 38 with 80 tickets: Progress 50%
Process 39 with 80 tickets: Progress 50%
Process 37: Decreased tickets to 20
Process 38 with 80 tickets: Progress 60%
Process 38: Decreased tickets to 20
Process 39 with 80 tickets: Progress 60%
Process 39: Decreased tickets to 20
Process 37 with 20 tickets: Progress 70%
Process 38 with 20 tickets: Progress 70%
Process 39 with 20 tickets: Progress 70%
Process 37 with 20 tickets: Progress 80%
Process 38 with 20 tickets: Progress 80%
Process 39 with 20 tickets: Progress 80%
Process 37 with 20 tickets: Progress 90%
Process 38 with 20 tickets: Progress 90%
Process 39 with 20 tickets: Progress 90%
Process 37 with 20 tickets: Progress 100%
Process 38 with 20 tickets: Progress 100%
Process 39 with 20 tickets: Progress 100%
pid 36 lot2: trap 14 err 5 on cpu 0 eip 0xffffffff addr 0xffffffff--kill proc
$
```

Testcase 3: Unfair distribution of tickets (lot3)

Three processes with tickets {10,10,90}

In lottery:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

$ lot3
Lottery Scheduler Test: Unfair Distribution of Tickets
Process with tickets 90 (PID 30): Progress 10%
Process with tickets 10 (PID 29): Progress 10%
Process with tickets 10 (PID 28): Progress 10%
Process with tickets 90 (PID 30): Progress 20%
Process with tickets 10 (PID 29): Progress 20%
Process with tickets 10 (PID 28): Progress 20%
Process with tickets 90 (PID 30): Progress 30%
Process with tickets 10 (PID 29): Progress 30%
Process with tickets 90 (PID 30): Progress 40%
Process with tickets 10 (PID 28): Progress 30%
Process with tickets 10 (PID 29): Progress 40%
Process with tickets 90 (PID 30): Progress 50%
Process with tickets 10 (PID 29): Progress 50%
Process with tickets 10 (PID 28): Progress 40%
Process with tickets 90 (PID 30): Progress 60%
Process with tickets 10 (PID 29): Progress 60%
Process with tickets 90 (PID 30): Progress 70%
Process with tickets 10 (PID 28): Progress 50%
Process with tickets 90 (PID 30): Progress 80%
Process with tickets 10 (PID 29): Progress 70%
Process with tickets 90 (PID 30): Progress 90%
Process with tickets 10 (PID 28): Progress 60%
Process with tickets 10 (PID 29): Progress 80%
Process with tickets 90 (PID 30): Progress 100%
Process with tickets 10 (PID 28): Progress 70%
Process with tickets 10 (PID 29): Progress 90%
Process with tickets 10 (PID 28): Progress 80%
Process with tickets 10 (PID 29): Progress 100%
Process with tickets 10 (PID 28): Progress 90%
Process with tickets 10 (PID 28): Progress 100%
```

Process with 90 tickets has completed way earlier than other processes.

In round robin:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

$ lot3
Lottery Scheduler Test: Unfair Distribution of Tickets
Process with tickets 10 (PID 41): Progress 10%
Process with tickets 10 (PID 42): Progress 10%
Process with tickets 90 (PID 43): Progress 10%
Process with tickets 10 (PID 41): Progress 20%
Process with tickets 10 (PID 42): Progress 20%
Process with tickets 90 (PID 43): Progress 20%
Process with tickets 10 (PID 41): Progress 30%
Process with tickets 10 (PID 42): Progress 30%
Process with tickets 90 (PID 43): Progress 30%
Process with tickets 10 (PID 41): Progress 40%
Process with tickets 10 (PID 42): Progress 40%
Process with tickets 90 (PID 43): Progress 40%
Process with tickets 10 (PID 41): Progress 50%
Process with tickets 90 (PID 43): Progress 50%
Process with tickets 10 (PID 42): Progress 50%
Process with tickets 10 (PID 41): Progress 60%
Process with tickets 90 (PID 43): Progress 60%
Process with tickets 10 (PID 42): Progress 60%
Process with tickets 10 (PID 41): Progress 70%
Process with tickets 90 (PID 43): Progress 70%
Process with tickets 10 (PID 42): Progress 70%
Process with tickets 10 (PID 41): Progress 80%
Process with tickets 90 (PID 43): Progress 80%
Process with tickets 10 (PID 42): Progress 80%
Process with tickets 10 (PID 41): Progress 90%
Process with tickets 90 (PID 43): Progress 90%
Process with tickets 10 (PID 42): Progress 90%
Process with tickets 10 (PID 41): Progress 100%
Process with tickets 90 (PID 43): Progress 100%
Process with tickets 10 (PID 42): Progress 100%
pid 40 lot3: trap 14 err 5 on cpu 1 eip 0xffffffff addr 0xffffffff--kill proc
$
```

Thus we can say that We have successfully implemented the lottery scheduler and has kept round-robin default scheduler as an option.

BY HARI KISHAN REDDY ABBASANI(ha2755)
BHARANI KUMAR REDDY(bb3722)

