Brandon Vo

Task 1: Nice (20 marks)

Having policies is the first step in implementing a non-trivial scheduler. For the sake of simplicity, you can implement the nice UNIX API as an extra system call. A higher nice value means lower priority. For more information type – man 2 nice on Linux system.

You should be able to store the nice value in the process structure. Xv6 stores the PCB in struct proc in the file proc.h

Note: You are not expected to enforce specific permissions on nice. On Linux systems, typically only root can increase priorities. But you can skip this part in the assignment. - Implement the nice system call in xv6. To verify that the nice value can be set and read back correctly you would need to write a simple test case. Also, explain your approach in the PDF.

Add the function definitions of GetNice and ChangeNice in defs.h

Changed proc.h to include a Nice value

Changed Proc.c to include the functions of GetNice and ChangeNice.

The following files were modified to add an additional system call for nice.

- **sysproc.c**
- **syscall.h**
- **user.h**
- **syscall.c**
- **syscall.h**
- **usys.S**
- **def.h**
- **sysfunc.h**

In order for the nice system call to be accessible, it has to be written into the xv6 kernel and be defined as a system call. In order to make nice functions accessible, it has to be defined in user.h to be callable through the shell. It has to be defined in def.h to locate the system call. It has to be written into sysfunc.h and syscall.c so that the system call handler can identify the nice function and be callable by the kernel.

The variable of nice itself is written into proc.c and defined in proc.h. Nice is written as part of the PCB block to identify priority.

- The nice variable serves as a means to establish priority in the system. Since we're using the lottery scheduler, the nice value will determine the weight of the tickets held by the processes.
- The nice value of a process can be changed by the nice.c program.
    - Format: nice [pid] [Nice Value]
        - nice 2 20 -> process with pid of 2 will get a nice value of 20
- If the nice value reported by the debug scheduler doesn't match the changed value of nice specified, that's because the scheduler is set to automatically lower the nice value if a process has been ignored too many times, so a process might its priority changed again after the forced nice value change.
    - In addition, because nice is being used to determine the amount of tickets given out, 0 isn't being used for nice due to division and modulation being used. Any process with a nice value of 0 will have its nice value randomly assigned from a range of 1-20

Brandon Vo

Task 2: Random Number Generator (20 marks)

Lottery scheduling is a randomized heuristic. Hence, we need to design a pseudo-random number generator (PRNG). Since xv6 does not include a PNRG, you need to design one.

Note: This does not require strong randomness such as in cryptographic algorithms but a decent PRNG will do.

We recommend using the Xorshift family, although you can use anything else.

If you do tend to use the Xorshift values, care should be taken to pick non-zero seed values.

- Implement an in-kernel PNRG. Write a simple test case to verify that the values returned look evenly distributed. Mention how to run this test in the PDF document.

The Random Number Generator was designed using a basic XORShift algorithm. It takes a base seed from the system time. The system time counts the ticks from the moment the Xv6 operating system, so it will always be changing every second and will never start at 0. The function will also autocorrect any negative numbers to their absolute value.

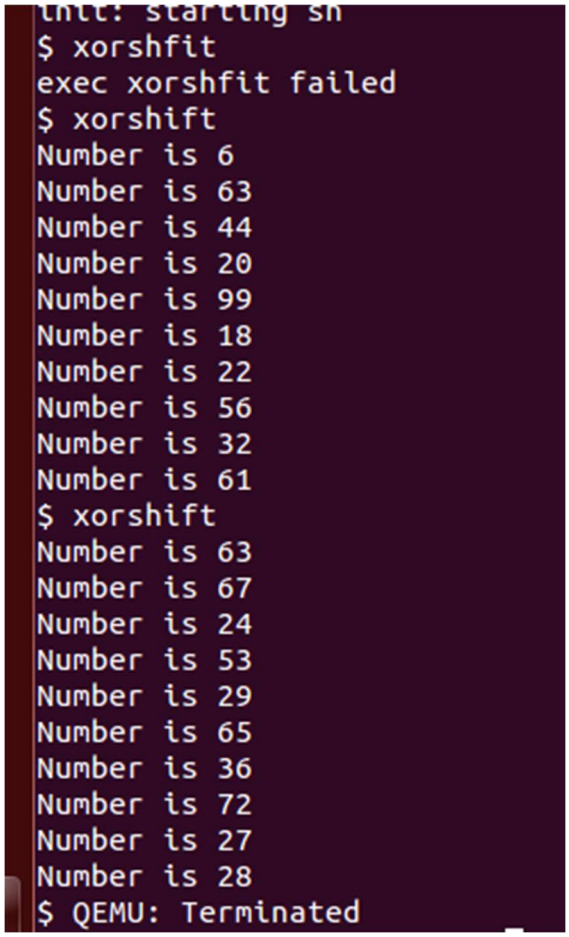
It will take the argv[1] of the xorshift command to use as x % mod to keep numbers within a specified range.

Possible input format:

- xorshift [maximum number] [seed]: Will use the specified seed and maximum number given
- xorshift [maximum number]: Will use the system time as the seed
- xorshift: Will use 100 as the default maximum number and the system time as the seed.

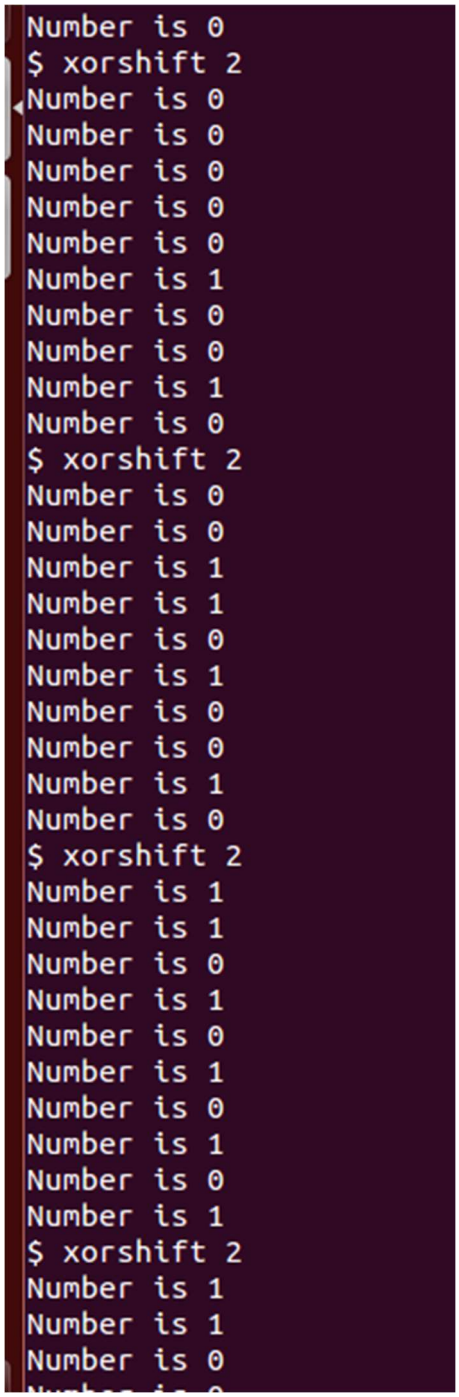
Brandon Vo

Xorshift using default settings (system time as a seed and max value of 100)



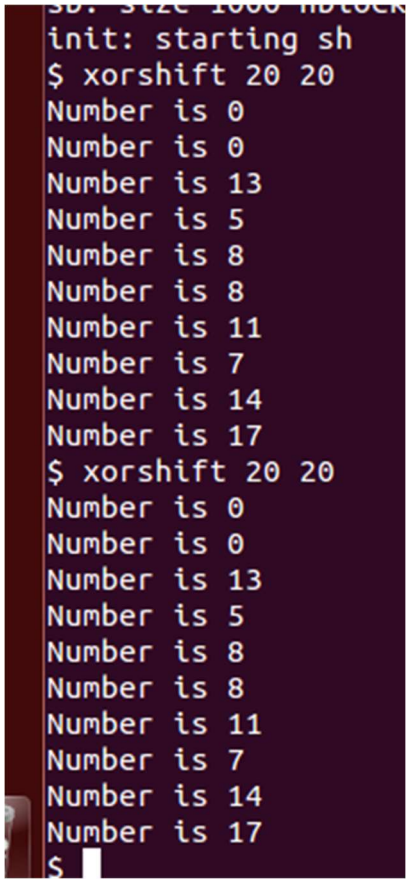
Brandon Vo

Xorshift using only a specified maximum number



Brandon Vo

Xorshift using the same seed.



Brandon Vo

Task 3: Scheduler (60 marks)

Mentioned in link, lottery scheduling works by assigning tickets to each of the process in the system. Then, in each time slice it randomly picks a 'winning' ticket. It is up to you to decide how many more tickets to assign less nice processed and vice versa.

The design problem is more open-ended. After reading the text mentioned in the link above, some of the design factors you might want to consider:

- Which data structure makes sense?

- In what way would you handle a process leaving or entering a system all while maintaining a policy?

- How would you map the winning number back to the runnable process?

Note: You should keep both the schedulers as an option. It is fine to select the scheduler (Round Robin vs Lottery) to be a compile time option.

- Implement lottery scheduling in xv6. As mentioned, the particulars of the data structure and the policies to map the number of tickets to nice values is a design choice which is somewhat open-ended. You have to convince us that your implementation handles edge cases. Be concrete while documenting your approach in the PDF. (Questions like 'what if?' should be mentioned and handled as well) - 50 marks

Come up with at least 3 test cases for your lottery scheduler - 10 marks

Hints:

- You can fork several children doing the same task with different priorities. Each of the child can print the progress in incremental points.

- You may change the priority of a process dynamically and check if its progress accelerates or slows down appropriately.

Brandon Vo

Variables being used:

- Nice: Priority value. Determines the weight of the tickets that a process holds.
- Tickets: Used to determine which processor gets chosen
- AllTickets: Counts every ticket being added. Used as a seed for the random number generator.
- MaxTicket: Not an actual variable, but a stand-in to calculate all the **runnable** tickets in the processor. Not accurate read out in the debugger most of the time. Used to calculate the range for the Winning_Ticket.
- Winning_Ticket: Determines which processor gets a chance to use the CPU.
- Ignored: Identifies how long a processor has been ignored in the lottery scheduler. If it gets too high, then the nice priority value is decremented, increasing priority.
- Ticket_counter: Keeps track of which process is checking their winning tickets. Used to help map the Winning_Ticket to the wining process.

Changes made:

The lottery ticket scheduler relies on a system where each process is given a set of tickets which will be chosen based on what the winning ticket is generated by the scheduler. When the scheduler is initiated in xv6, it will look through the process table for any runnable processes. This is because only these processes should be allowed in the scheduler. The ticket distribution starts off entirely random. Each process is given a random amount of tickets and a random nice value to determine priority. This is because there isn't a clear method to identify which processes need more computation time, so the necessary computation time is random.

When all tickets have been distributed, the maximum number of tickets held by every runnable process is calculated to know the maximum number of tickets. This variable is labelled as MAXTICKETS. MAXTICKETS is used to determine what range of numbers in which the winning ticket will be generated. When the winning ticket is generated, it will go through the list of processes from the proc table and check which one has a ticket value greater than the winning ticket. The winning ticket is calculated based on the total number of tickets of all runnable processes in the scheduler.

When checking which process has the winning ticket, it uses a variable called ticket_counter on top of using the process's ticket number to check if the process is the winning process. Ticket_Counter is used as the means to define the data structure of this lottery scheduler. If the process has tickets greater than the winning ticket, that means the process is chosen to enter the CPU. Otherwise, the scheduler check the next process, and the scheduler will increment the ticket counter using the unchosen process's tickets. In addition to using the ticket counter and the process's tickest to determine if the process has enough tickets, the total amount will be divided by the nice value of the process. This is to give a form of priority for the processes. Lower nice values means the tickets have more weight while higher nice values mean the tickets have less weight. This ensures it so that a process with a high nice value and a lot of tickets may have lower priority compared to a process with a low nice value and few tickets. With the ticket counter, it guarantees that atleast one process will be chosen for the lottery scheduler. Every time the scheduler is called, it will repeat this process.

Each time a process is killed, enters zombie state, or goes to sleep, it will relinquish their spot and let go of their tickets. In addition to being marked as killed, the ticket counter will deduct the killed process's tickets to account for the change. If a new process is put into the scheduler and is marked runnable, then that process will be given its own set of tickets and the global ticket counter will be incremented to

account for the new process. When the scheduler is utilized, it will go through the list of runnable processes and calculate all the tickets of the runnable processes to find the MAXTICKETS.

To help account for fairness, the nice value determines the weight of a process's tickets. On top of the nice value, every process holds an ignored counter identifying how many times a process has been ignored. If a process has been ignored too many times, its nice value will decrease, increasing the weight of the process's tickets. This would increase the priority of the process.

- Inside proc.c, there is a DEBUG macro. Turning the macro from 0 to 1 will have the scheduler print out information regarding the tickets found, a process's nice value, and the winning ticket.
- Forked processes will also inherit the same nice value from their parents
    - This excludes the ticket value because every forked process comes from a root process running the xv6 OS. This would cause every process to share the same ticket value with the main OS.
- Because the lottery scheduler is focused on runnable processes, any process that sleeps or is killed must have their tickets taken out from the scheduler. The global ticket counter must be decremented based on the number of tickets lost by the process exiting.
    - If a process wakes up, its tickets are added back into the counter as it's now a part of the scheduler.

Test Cases:

- Case1.c is a test case that generates multiple forks at once, modifies the child processor to be low priority and the parent processor to have high priority. This creates multiple zombie processes because this makes it more likely for the parent process to finish first before the child process. It also shows the lottery scheduler's random nature because it will always show different processes with different PIDs announcing that they've been chosen each time.
    - Input: case1
    - The number of zombies created will be random due to the lottery scheduler randomly choosing a parent process to end faster than the child process.
    - Mainly just used to observe the random pattern of the lottery scheduler by looking at rate of zombie processes being created.

```
init: starting sh
$ case1
This is the parent process, my process ID is 3 and my child is 4
Hello, I'm in the child, my process ID is 4
This is the paThis is the parent process, my process ID is 3 and my child is 5
Hrent process, my process ID is 4 and my child is 6
Hello, I'm in the child, my process ID is 5
This is the parent process, my procThis is the parent process, my process ID is 3 and my child is 7
ello, I'm in the child, my process ID is 6
This isHess ID is 4 and my child is 8
This is the parent process, my process ID is 5 and my child is 9
Hello, I'm in the child, my process ID is 8
 the parent process, my process ID is 3 and my child is 10
This is the parent process, my process ID is 6 and my child is 11
ello, I'm in the child, my process ID is 10
Hello, I'm in the child, my process ID is 9
This is Hello, I'm in the child, my process ID is 18
This is the parent process, my process ID is 6 and my cThis is the parent process, my process ID is 4 and my child is 12
Hello, I'm in the child, my process ID is hthe parent process, my process ID is 9 and my child is 18
This is the parent process, my process ID is 10 and my child is 17
Hello, I'm in the child, my process ID is 17
This is the parent process, my process ID is 18 and my child is 19
This is the parent process, my process ID is 8 and my child is 14
1Hello, I'm in the child, my process ID is 14
Hello, I'm in the child, my process ID is 16
This is the parent process, my process ID is 3 and my child is 15
ild is 16
Hello, I'm in the child, my process ID is 15
This is the parent process, my process ID is 4 and my child is 20
This is the parent process, my process ID isH2
Hello, I'm in the child, my process ID is 24
This is the parent process, my process ID is 8 and my child is 22
ello, I'm in the child, my process ID is 7
 9 and my child is 21
This is the parent process, my process ID is 12 and my child is 26
Hello, I'm in the child, my process ID is 20
This is the parent process, my process ID is 5 and my child is 13
This is the pareThis is the parent process, my process ID is 16 and my child is 24
Hello, I'm in the child, my process ID is 26
This is the parent process, my process ID is 6 and my child is 25
Hello, I'm in the child, my process ID is 27
```

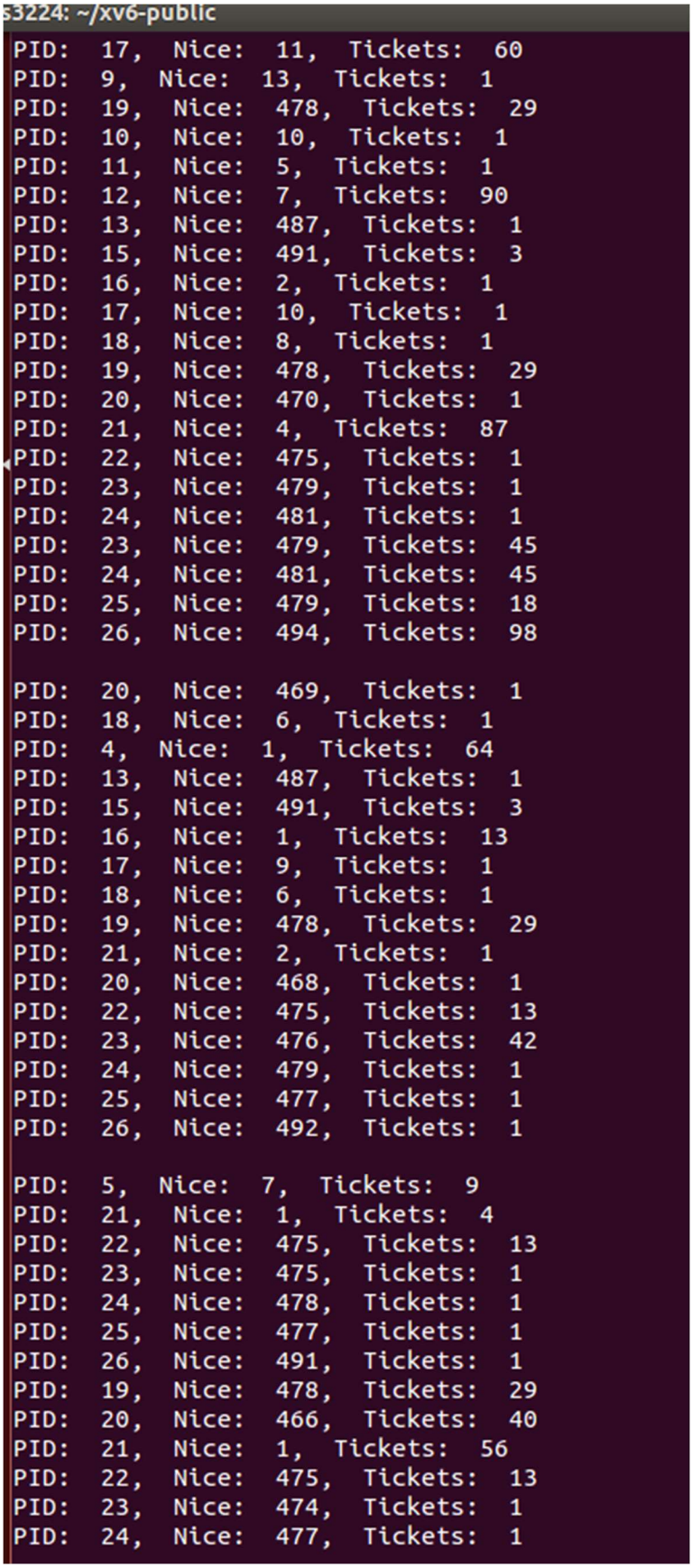
```
Hello, I'm in the child, my process ID is 23
zombie!
Hzombie!
zombie!
zombie!
Hello, I'm in the child, my process ID is 21
ello, I'm in the child, my process ID is 25
t process, my process ID is 5 and my child is 27
Hello, I'm in the child, my process ID is 13
HThis is the parent process, my process ID is 7 and my child is 2ello, I'm in the child, my process ID is 22
Hello, I'm in the child, my process ID is 19
8
Hello, I'm in the child, my process ID is 11
This is the parent pThis is the parent process, my process ID is 7 and my child is 30
zHello, I'm in the child, my process ID is 28
Hello, I'm in the child, my process ID is 30
rocess, my process ID is 11 and my child is 31
ombie!
zombie!
zombie!
zombie!
zombie!
zombie!
zombie!
Hello, This is the parent process, my process ID is 11 and my child is This is the parent process, my pr33
ocess ID is 28 and my child is 32
Hello, I'm in the child, my process ID is 33
zombie!
IHello, I'm in the child, my process ID is 32
'm in the child, my process ID is 31
This is the parHzombie!
ent process, my process ID is 13 and my child is 29
ello, I'm in the child, my process ID is 29
This is the parent process, my process ID is 31 and my child is 34
Hello, I'm in the child, my process ID is 34
zombie!
zombie!
```

- Case2.c is used to monitor the change in nice and ticket values of a process that has an extremely low priority value compared with a child process with the highest nice priority value. It will call a function and continuously fork that processes as it runs a loop to show the change in nice values

and tickets overtime. The parent gets the highest priority value while the children higher priority values.

- Input: case2 [Nice value]
- Will affect only the child process's nice values as the parent processes will always have the highest priority value.
- The screenshots below show some of the high priority processes such as process 24, process 25, and process 26 which have a nice value over 400 being decreased by the next update due to being ignored too many times.

Brandon Vo


s3224: ~/xv6-public
PID: 17, Nice: 11, Tickets: 60
PID: 9, Nice: 13, Tickets: 1
PID: 19, Nice: 478, Tickets: 29
PID: 10, Nice: 10, Tickets: 1
PID: 11, Nice: 5, Tickets: 1
PID: 12, Nice: 7, Tickets: 90
PID: 13, Nice: 487, Tickets: 1
PID: 15, Nice: 491, Tickets: 3
PID: 16, Nice: 2, Tickets: 1
PID: 17, Nice: 10, Tickets: 1
PID: 18, Nice: 8, Tickets: 1
PID: 19, Nice: 478, Tickets: 29
PID: 20, Nice: 470, Tickets: 1
PID: 21, Nice: 4, Tickets: 87
PID: 22, Nice: 475, Tickets: 1
PID: 23, Nice: 479, Tickets: 1
PID: 24, Nice: 481, Tickets: 1
PID: 23, Nice: 479, Tickets: 45
PID: 24, Nice: 481, Tickets: 45
PID: 25, Nice: 479, Tickets: 18
PID: 26, Nice: 494, Tickets: 98
PID: 20, Nice: 469, Tickets: 1
PID: 18, Nice: 6, Tickets: 1
PID: 4, Nice: 1, Tickets: 64
PID: 13, Nice: 487, Tickets: 1
PID: 15, Nice: 491, Tickets: 3
PID: 16, Nice: 1, Tickets: 13
PID: 17, Nice: 9, Tickets: 1
PID: 18, Nice: 6, Tickets: 1
PID: 19, Nice: 478, Tickets: 29
PID: 21, Nice: 2, Tickets: 1
PID: 20, Nice: 468, Tickets: 1
PID: 22, Nice: 475, Tickets: 13
PID: 23, Nice: 476, Tickets: 42
PID: 24, Nice: 479, Tickets: 1
PID: 25, Nice: 477, Tickets: 1
PID: 26, Nice: 492, Tickets: 1
PID: 5, Nice: 7, Tickets: 9
PID: 21, Nice: 1, Tickets: 4
PID: 22, Nice: 475, Tickets: 13
PID: 23, Nice: 475, Tickets: 1
PID: 24, Nice: 478, Tickets: 1
PID: 25, Nice: 477, Tickets: 1
PID: 26, Nice: 491, Tickets: 1
PID: 19, Nice: 478, Tickets: 29
PID: 20, Nice: 466, Tickets: 40
PID: 21, Nice: 1, Tickets: 56
PID: 22, Nice: 475, Tickets: 13
PID: 23, Nice: 474, Tickets: 1
PID: 24, Nice: 477, Tickets: 1

- Case3.c is used as a way to check the chances of two processes ending first. It creates one process and forks it. These two processes will share the same nice value and the same ticket value. Both processes will wait a certain amount of time before ending their program, declaring which one finished first.
    - Because the forked process shares the same ticket value and nice value, ideally, they both should have a 50/50 chance on average for one process to end first.
    - Command to execute is just case3.

```
xv6...
cpu1: starting
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ case3
Child finished
Parent finished
$ case3
Parent finished
Child finished
$ case3
Parent finished
Child finished
$ case3
Child finished
Parent finished
$ case3
Parent finished
Child finished
$ case3
Parent finished
Child finished
$ case3
Parent finished
Child finished
$
```

Problems:

- Because fork copies the PID's nice value and tickets to the child processor, this includes the PID running xv6.
    - Every process is a fork off the root, meaning every process gets a copy of the main process's nice and ticket value.
    - Many processes may spawn with the same number of tickets or nice values due to this additional fork procedure.
- During xv6's runtime tests, most of the Winning Tickets generated were small ranging from 0 – 10. There could be multiple factors for this problem.
    - The Winning_Ticket range is determined from the random number generator under the range of the maximum number of tickets found by every runnable process
    - There could be just too few runnable processes to create a large enough range of Winning Tickets to generate
    - It could be a problem with the random number generator because uptime() cannot be used in proc.c due to conflicting system calls. The pseudo-random number generator in xv6 is using a set seed in random.h.

    - It could be that the mod MaxTickets leads to a bias towards low numbers due to not being truly random.
- Similarly to, or because of, the above issue, the tickets granted to each process is biased towards 1 ticket. This may be due to the same issues that the Winning_Ticket generation has.
- The nice values were decremented until it reaches 1 to indicate the process reached the highest priority.
    - This leads to a conflict with forking processes because every process is a fork off the Xv6 root process.
    - When the root process's nice value was decremented to one, that meant every forked process after that will share a nice value of 1, giving the maximum priority to every process.
- A process might generate with 1 ticket, giving it the lowest priority regardless of its nice value.
    - Attempting to change a process's ticket value while the scheduler risks creating a panic error
- AllTickets was initially used as a global ticket counter to constantly keep track of every ticket being used in the process. However, there leftover tickets because decrementing the counter when a process went to sleep would cause a panic error due to AllTickets going into the negative integers.
    - At the cost of efficiency, all tickets are calculated whenever the scheduler runs. This requires going through the process table once, reducing efficiency.
    - The maximum number of tickets is not calculated accurately in the debugger because it calculates for running processes while the debugger reads from all processes.
    - Because AllTickets keeps getting incremented, running the scheduler too long will eventually cause AllTickets to have an integer overflow into the negatives.
- When running the test cases, the debugger shows many of the wining tickets being generated to be very low. This is because there are very few runnable processes running in the scheduler, meaning the range of winning tickets to create is low.

Crashes:

- Whenever using the lottery scheduler, there is a risk of a memory or page fault error in xv6. This may be the result of the process pointer going out of bounds, leading to a triple fault.
    - Uncertain if this problem has been fixed or not after recent changes and fixes to allow the winning ticket to be 0
    - Uncertain if DEBUG setting affects this as well
- Turning on DEBUG settings in proc.c uses cprintf. These commands create processes within the scheduler, which risk giving misleading info as to what processes are in the scheduler.
    - Because these are extra processes being spawned, it can create issues regarding who gets the scheduler lock. As a result, turning on debug runs a risk of running into an xv6 panic: acquire issue.

Uncertainties:

- Not certain if the lottery scheduler is being fair because the ticket generation and priority generation is random.
    - Because it is difficult to determine what processes need how many resources or what priority they require, the nice values are randomly generated for every process

        - The nice values are decremented as they go on, but never incremented. This means every process that persists will have their priority increase overtime.
    - The lottery scheduler looks through the table and selects the first process that has the winning ticket. The ones in the back have a lower chance of being chosen.
- A process getting a ticket of 1 gives it the lowest priority regardless of its nice value.
    - Tickets are added to the process if it gets ignored for too long, but the tickets granted to the process are also randomized.