**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:

A screenshot of a computer

Description automatically generated

**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):

A black background with white text

Description automatically generated

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 with in 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 :

A black screen with white text

Description automatically generated

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

A screenshot of a computer

Description automatically generated

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

A graph of blue bars

Description automatically generated with medium confidence

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:**

**A screenshot of a computer screen

Description automatically generated**

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

**A black screen with white text

Description automatically generated**

Run testcase1 with lottery scheduler:

#define xscheduler 1

**A screen shot of a computer

Description automatically generated**

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:**

A screenshot of a computer

Description automatically generated

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:**

**A screenshot of a computer

Description automatically generated**

**Testcase 3: Unfair distribution of tickets (lot3)**

Three processes with tickets {10,10,90}

In lottery:

A screen shot of a computer

Description automatically generated

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

In round robin:

**A screenshot of a computer

Description automatically generated**

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)