Nice System call and Priority Scheduler Implementation

Nice System Call

The goal is to add a nice system call to xv6, similar to the UNIX nice command. This system call adjusts the scheduling priority of a process, with higher nice values indicating lower priority. This is particularly useful in environments where certain processes should yield CPU time to others, thus allowing control over process scheduling behaviour.

Overview of Key Components and Modifications -

The implementation of nice requires changes across multiple files in xv6 to ensure that:

- The nice value can be stored and accessed in each process's control block.
- The kernel has a mechanism to modify this nice value through a system call.
- The scheduler considers the nice value when selecting processes to run.

nice.c - User-Level Command Implementation -

This file contains the implementation of the nice command that users will run to change the priority of a process based on its nice value. The nice value is directly proportional to priority, with a higher nice value indicating a lower priority.

1) **Arguments Parsing and Usage Check**:

   The argc (argument count) variable is checked to determine if the user provided one or two arguments.

   - Case 1: If the user provides two arguments, it assumes the arguments are pid and value, in that order.
   - Case 2: If the user provides only one argument, the program assumes it is the value and applies it to the current process (pid is obtained using getpid()).
   - Error Case: If the arguments don’t match either case, the program prints the correct usage format.

2) **Calling the nice System Call:**

   - The nice system call is invoked with pid and value. The function returns the previous nice value if successful; otherwise, it returns a negative value, indicating an error.

   - If the nice call fails (returns a negative value), an error message is printed. Otherwise, it prints the pid and the old nice value.

sys_nice - System Call Implementation in sysproc.c

This function implements the nice system call in the kernel. It changes the nice value (priority level) of a process based on its pid.

1) **Fetching Arguments:**

   - argint(0, &pid) and argint(1, &value) retrieve the arguments from the user call: pid and value.
   - It checks if the value is within a predefined range (0 <= value <= MAX_NICE). If not, it returns an error (-1).

2) **Process Table Locking:**

   To avoid race conditions, the process table is locked using acquire(&ptable.lock). This ensures only one process can modify or access the table at a time.

3) **Finding and Updating the Process:**

   - The function iterates over all processes (ptable.proc). If it finds a process with the specified PID, it updates its nice value to value and stores the old nice value.

   - The function releases the lock and returns the old nice value if successful, or -1 if no process with the specified pid was found.

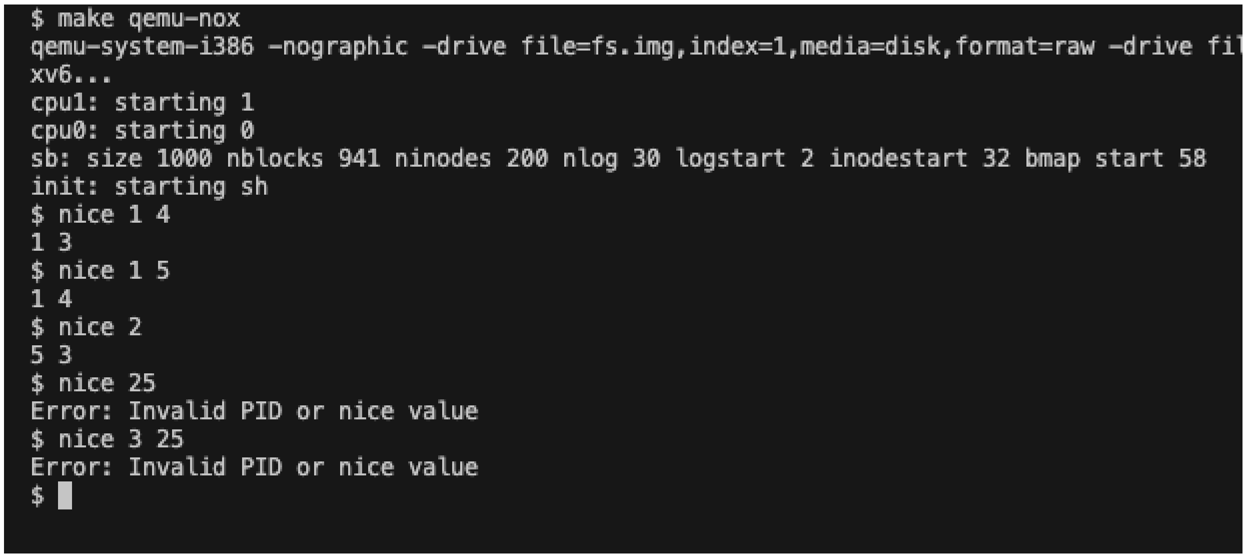
These are the main implementation points. Apart from this, the following actions are performed to add the system call.

- A new integer field ***nice*** is added to the proc structure, representing the process's priority. Lower values indicate higher priority, with a default nice value typically set to zero or a medium priority at process creation.
- MAX_NICE is defined to specify the maximum allowable nice value, thus setting bounds on how much the priority can be adjusted.
- A new system call number (SYS_nice) is defined. This unique identifier allows the system to recognize the nice system call in user programs.
- sys_nice is declared as an external function and added to the syscalls table, mapping SYS_nice to the sys_nice function. This mapping allows xv6 to call sys_nice when a user program invokes the nice system call.

Edge Case Considerations

- **Invalid pid or value:** The sys_nice function verifies both pid and nice values to ensure they are valid. Invalid inputs result in a return value of -1.
- **Bounds Checking:** The value is checked to ensure it lies within 0 and MAX_NICE.
- **Concurrency and Process Table Locking:** By acquiring and releasing the ptable.lock, sys_nice ensures thread-safe access to the process table, avoiding race conditions when multiple processes attempt to change nice values**.**

Screenshot of Result -



```
$ make qemu-nox
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive fil
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ nice 1 4
1 3
$ nice 1 5
1 4
$ nice 2
5 3
$ nice 25
Error: Invalid PID or nice value
$ nice 3 25
Error: Invalid PID or nice value
$ 
```

In the above screenshot, we can see our various test cases in action.

- When we run nice 1 4, it returns the PID (1) and the old nice value (3). Now if we run nice 1 5, it returns the PID and the old nice value which was set previously (4)
- When we run nice 2, it returns the PID of the current process and the old nice value.
- When we run nice with a nice value of 25, which is more than our MAX_NICE, we get an error as expected.

Priority Scheduler

The goal is to implement a Priority-Based Scheduler in xv6 OS that schedules processes based on their priority levels. Each process is assigned a nice value, representing its priority, where a lower nice value indicates a higher priority. The scheduler is configured to choose processes with higher priorities (lower nice values) before those with lower priorities. If no high-priority process is available, it defaults to lower-priority processes.

Key Files Modified/Created -

- proc.c - Contains the modified scheduler function with priority-based scheduling logic.
- proc.h - Defines process structure fields, including the nice value for priority.
- test1.c, test2.c, test3.c - Test files to validate the priority scheduler with different process workloads.
- sysproc.c - Defines the sys_nice function, which allows users to set or modify a process's nice value.
- syscall.c, syscall.h - Include the nice system call, allowing for interaction with the scheduler.
- Scheduler Logic and Implementation
- Our scheduler is implemented to support both Round Robin (RR) and Priority-Based Scheduling. This flexibility is controlled at compile-time using the macro PRIORITY_SCHEDULER. When enabled, the priority-based logic is used, and processes are scheduled based on their nice values.

The scheduler implementation is in proc.c, specifically in the scheduler() function. Here's a breakdown of the process:

Selecting the Scheduler Type:

```
#define PRIORITY_SCHEDULER 1
#if PRIORITY_SCHEDULER
   // Priority-based scheduling logic
#else
   // Round Robin scheduling logic
#endif
```

This directive allows selecting between the original round-robin and priority scheduling at compile-time.

Priority Scheduling Logic:

In the priority scheduler, processes with the lowest nice value (highest priority) are selected for execution first. Here's a step-by-step explanation of how this logic works:

Step 1: Initialize Selection Criteria:

```
struct proc *selected_proc = 0;
int min_nice = MAX_NICE + 1;
```

Before starting the search for the highest-priority runnable process, we initialize a selected_proc pointer to 0, representing no process selected yet, and set min_nice to MAX_NICE + 1. The min_nice variable will keep track of the lowest nice value found among runnable processes, which indicates the highest priority. Setting it initially to MAX_NICE + 1 ensures that any runnable process with a nice value within bounds will update this minimum.

Step 2: Find the Highest-Priority Runnable Process:

```
acquire(&ptable.lock);  // Lock the process table

// Loop to find the highest priority (lowest nice value) among runnable processes
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if (p->state == RUNNABLE && p->nice < min_nice) {
        min_nice = p->nice;
    }
}
```

With the process table lock acquired, we loop through all processes in ptable.proc. For each process, we check if its state is RUNNABLE and if its nice value is lower than min_nice. If both conditions are met, we update min_nice to this process's nice value. This loop ensures we find the lowest nice value among all runnable processes, representing the highest priority.

Step 3: Select a Process with the Minimum Nice Value:

```
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if (p->state == RUNNABLE && p->nice == min_nice) {
        selected_proc = p;
        break;
    }
}
```

After identifying the minimum nice value, we loop again through the processes to find a process with this specific min_nice value that is also in the RUNNABLE state. The first matching process becomes the selected_proc. This process will be the highest-priority process available to run at this point. Using break after finding a match ensures that we select only one process without searching further.

Step 4: Run the Selected Process:

Once the highest-priority runnable process is identified, it is set to RUNNING, and control is switched to this process:

```
if (selected_proc) {
   p = selected_proc;
   c->proc = p;
   switchuvm(p);
   p->state = RUNNING;

   swtch(&(c->scheduler), p->context);
   switchkvm();

   c->proc = 0;
}
```

Step 5: Unlock Process Table:

Finally, we release the lock on the process table to allow other operations.

```
release(&ptable.lock);
```

Test Cases -

We developed three test cases (test1.c, test2.c, and test3.c) to evaluate the priority scheduler. Each test case runs multiple child processes executing a prime number calculation task. Each process is assigned a unique nice value to observe how the scheduler prioritizes processes with different nice values.

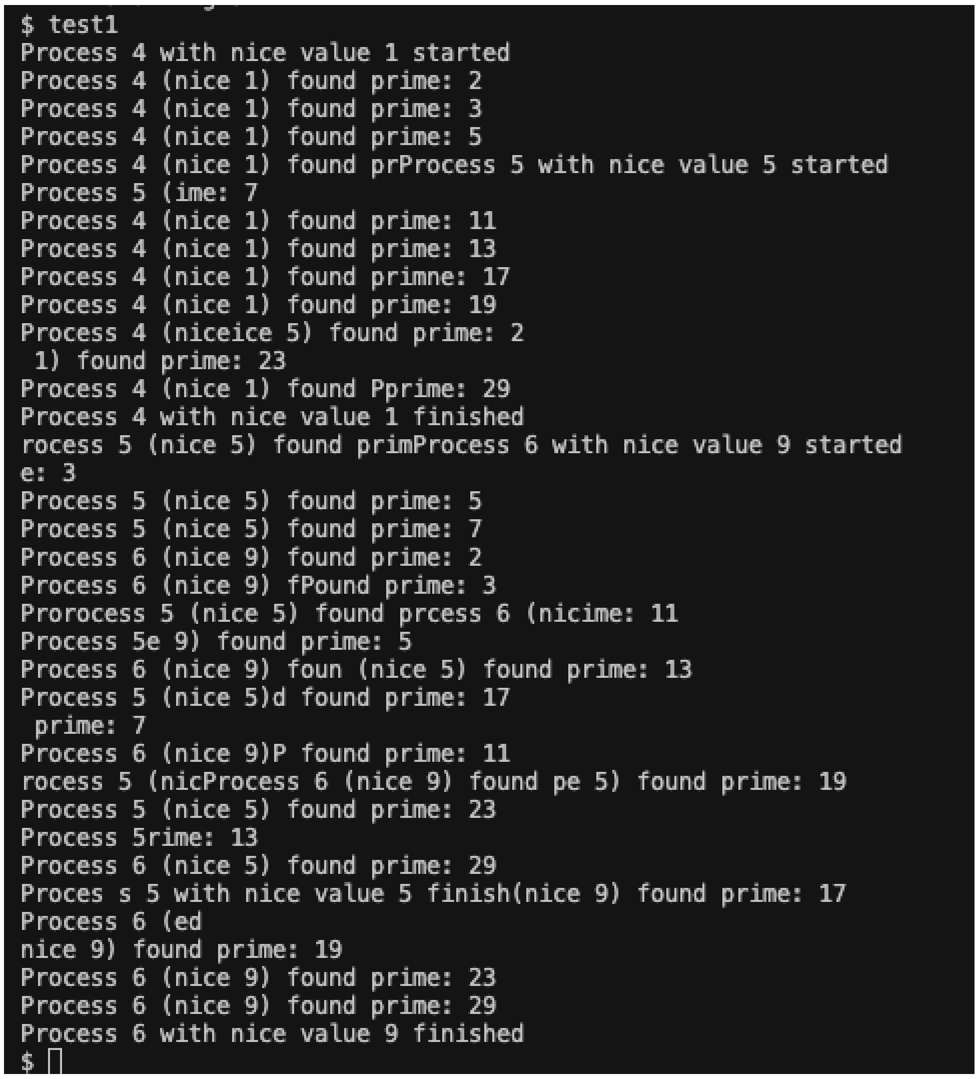
Test Case 1: Basic Priority Test - test1.c

This test case tests the priority scheduler in xv6 by creating three child processes with different priority levels, each running a function to compute and print prime numbers up to a limit. The function takes in a process ID and a nice_value (priority) and calculates prime numbers.

In the main function, three processes are created:

- The first child has a high priority (nice value of 1), so it should receive more CPU time.
- The second child has medium priority (nice value of 5).
- The third child has low priority (nice value of 9), meaning it is expected to be preempted more often by higher-priority processes.

Below is a screenshot of the result. In this case, the first, second and third child are process numbers 4,5 and 6 respectively.



```
$ test1
Process 4 with nice value 1 started
Process 4 (nice 1) found prime: 2
Process 4 (nice 1) found prime: 3
Process 4 (nice 1) found prime: 5
Process 4 (nice 1) found prProcess 5 with nice value 5 started
Process 5 (ime: 7
Process 4 (nice 1) found prime: 11
Process 4 (nice 1) found prime: 13
Process 4 (nice 1) found primne: 17
Process 4 (nice 1) found prime: 19
Process 4 (niceice 5) found prime: 2
 1) found prime: 23
Process 4 (nice 1) found Pprime: 29
Process 4 with nice value 1 finished
rocess 5 (nice 5) found primProcess 6 with nice value 9 started
e: 3
Process 5 (nice 5) found prime: 5
Process 5 (nice 5) found prime: 7
Process 6 (nice 9) found prime: 2
Process 6 (nice 9) fPound prime: 3
Prorocess 5 (nice 5) found prcess 6 (nicime: 11
Process 5e 9) found prime: 5
Process 6 (nice 9) foun (nice 5) found prime: 13
Process 5 (nice 5)d found prime: 17
 prime: 7
Process 6 (nice 9)P found prime: 11
rocess 5 (nicProcess 6 (nice 9) found pe 5) found prime: 19
Process 5 (nice 5) found prime: 23
Process 5rime: 13
Process 6 (nice 5) found prime: 29
Proces s 5 with nice value 5 finish(nice 9) found prime: 17
Process 6 (ed
nice 9) found prime: 19
Process 6 (nice 9) found prime: 23
Process 6 (nice 9) found prime: 29
Process 6 with nice value 9 finished
$ 
```

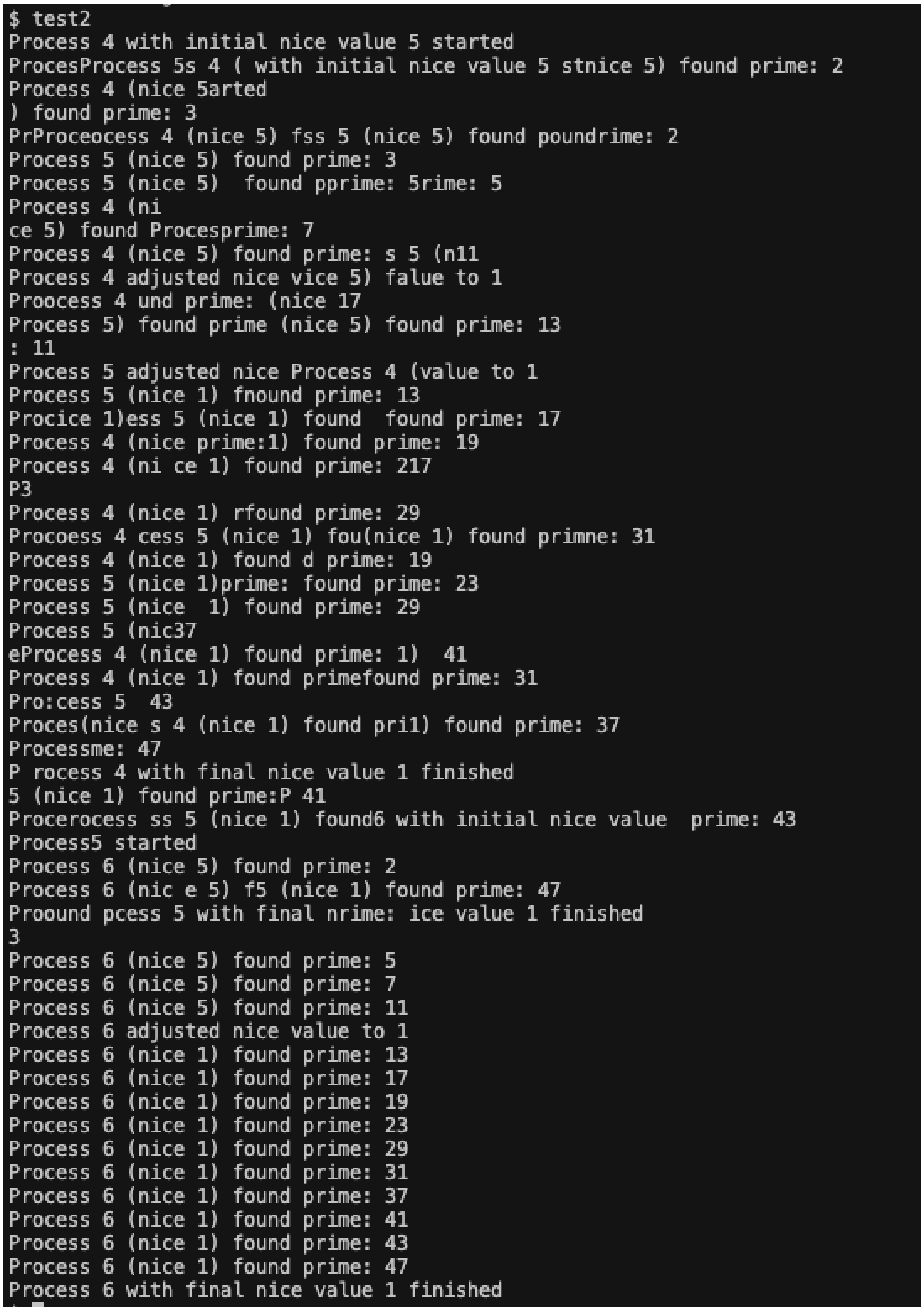
In this output, Process 4, with the highest priority (nice value of 1), starts and completes its task almost without interruption, finding all primes and finishing before the others. Process 5 (medium priority, nice value 5) starts shortly after but is allowed fewer CPU cycles, resulting in a slower pace of prime finding. Process 6, with the lowest priority (nice value 9), starts even later and has the least CPU time, as indicated by its sporadic prime findings mixed with outputs from Process 5. Although there is some output interleaving, the scheduler works as expected.

This behaviour demonstrates that the priority scheduler in xv6 is working as intended: higher-priority processes (lower nice values) receive more CPU time, allowing them to complete tasks faster, while lower-priority processes are preempted more frequently.

Test Case 2: Dynamic Priority Adjustment - test2.c

This test case evaluates dynamic priority adjustment in the xv6 priority scheduler. It starts with three processes, all initially assigned a medium priority (nice value of 5). Each process runs a CPU-intensive task, calculating a series of prime numbers and printing each result. After each process finds its first five primes, its priorities are dynamically changed: two processes are adjusted to a higher priority (nice value of 1), while the third process remains unchanged. This setup allows us to observe if the scheduler dynamically reallocates CPU time based on new priority assignments.

Below is a screenshot of the result. In this case, the first, second and third child are process numbers 4,5 and 6 respectively.



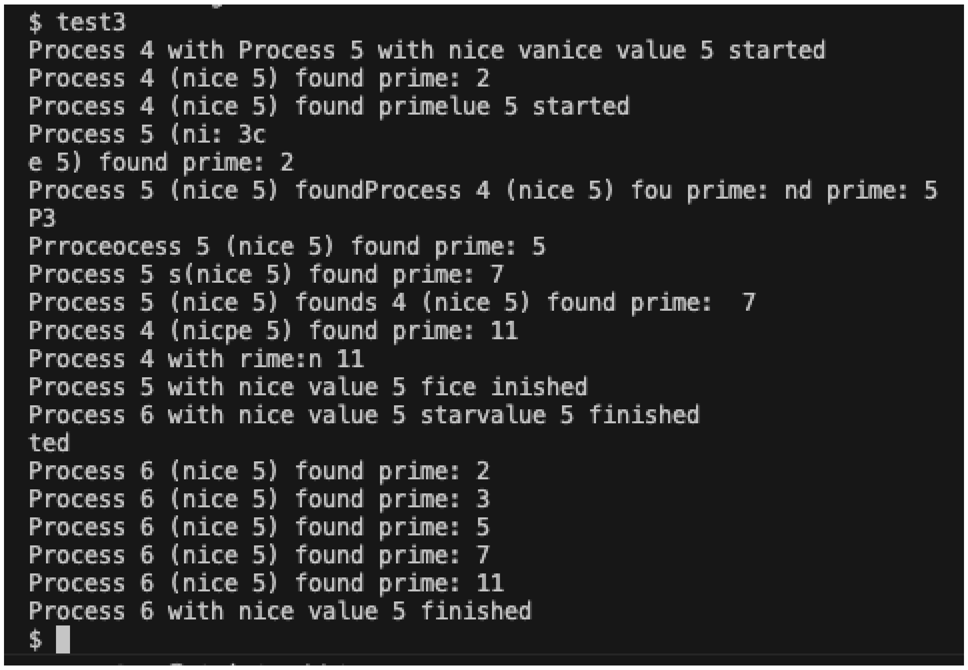
```
$ test2
Process 4 with initial nice value 5 started
ProcesProcess 5s 4 ( with initial nice value 5 stnice 5) found prime: 2
Process 4 (nice 5arted
) found prime: 3
PrProceocess 4 (nice 5) fss 5 (nice 5) found poundrime: 2
Process 5 (nice 5) found prime: 3
Process 5 (nice 5)  found pprime: 5rime: 5
Process 4 (ni
ce 5) found Procesprime: 7
Process 4 (nice 5) found prime: s 5 (n11
Process 4 adjusted nice vice 5) falue to 1
Proocess 4 und prime: (nice 17
Process 5) found prime (nice 5) found prime: 13
: 11
Process 5 adjusted nice Process 4 (value to 1
Process 5 (nice 1) fnound prime: 13
Procice 1)ess 5 (nice 1) found  found prime: 17
Process 4 (nice prime:1) found prime: 19
Process 4 (ni ce 1) found prime: 217
P3
Process 4 (nice 1) rfound prime: 29
Procoess 4 cess 5 (nice 1) fou(nice 1) found primne: 31
Process 4 (nice 1) found d prime: 19
Process 5 (nice 1)prime: found prime: 23
Process 5 (nice  1) found prime: 29
Process 5 (nic37
eProcess 4 (nice 1) found prime: 1)  41
Process 4 (nice 1) found primefound prime: 31
Pro:cess 5  43
Proces(nice s 4 (nice 1) found pri1) found prime: 37
Processme: 47
P rocess 4 with final nice value 1 finished
5 (nice 1) found prime:P 41
Procerocess ss 5 (nice 1) found6 with initial nice value  prime: 43
Process5 started
Process 6 (nice 5) found prime: 2
Process 6 (nic e 5) f5 (nice 1) found prime: 47
Proound pcess 5 with final nrime: ice value 1 finished
3
Process 6 (nice 5) found prime: 5
Process 6 (nice 5) found prime: 7
Process 6 (nice 5) found prime: 11
Process 6 adjusted nice value to 1
Process 6 (nice 1) found prime: 13
Process 6 (nice 1) found prime: 17
Process 6 (nice 1) found prime: 19
Process 6 (nice 1) found prime: 23
Process 6 (nice 1) found prime: 29
Process 6 (nice 1) found prime: 31
Process 6 (nice 1) found prime: 37
Process 6 (nice 1) found prime: 41
Process 6 (nice 1) found prime: 43
Process 6 (nice 1) found prime: 47
Process 6 with final nice value 1 finished
```

Initially, all three processes—Process 4, Process 5, and Process 6—begin with the same priority (nice value of 5), so they receive equal CPU time from the scheduler, resulting in a balanced, interleaved output where each process progresses at a similar pace. For example, each process finds and prints primes in the same range, such as 2, 3, 5, and 7, demonstrating that the scheduler treats them equally. After the initial phase, priorities are adjusted: Process 4 and Process 5 both change to a nice value of 1, giving them the highest priority, while Process 6 remains at a nice value of 5. This adjustment significantly impacts the scheduler's CPU time allocation. Processes 4 and 5, now with higher priority, dominate the CPU time, progressing through their remaining prime calculations with minimal interruptions, while Process 6, with a lower priority, slows down and appears less frequently in the output. As a result, Processes 4 and 5 reach the end of their tasks first, printing their "finished" messages earlier, while Process 6, deprived of CPU time due to its lower priority, completes last. This output confirms that the scheduler dynamically reallocates CPU time based on priority changes, allowing higher-priority processes to complete faster when their nice values are lowered.

Test Case 3: Equal Priorities - test3.c

This program tests the priority scheduler in xv6 by creating three child processes, each with an identical nice value of 5, meaning they share the same medium priority. Each child runs the find_primes function, which calculates and prints the first five prime numbers. The objective is to verify that processes with identical nice values are handled fairly by the scheduler.

Below is a screenshot of the result. In this case, the first, second and third child are process numbers 4,5 and 6 respectively.



```
$ test3
Process 4 with Process 5 with nice vanice value 5 started
Process 4 (nice 5) found prime: 2
Process 4 (nice 5) found primelue 5 started
Process 5 (ni: 3c
e 5) found prime: 2
Process 5 (nice 5) foundProcess 4 (nice 5) fou prime: nd prime: 5
P3
Prroceocess 5 (nice 5) found prime: 5
Process 5 s(nice 5) found prime: 7
Process 5 (nice 5) founds 4 (nice 5) found prime:  7
Process 4 (nicpe 5) found prime: 11
Process 4 with rime:n 11
Process 5 with nice value 5 fice inished
Process 6 with nice value 5 starvalue 5 finished
ted
Process 6 (nice 5) found prime: 2
Process 6 (nice 5) found prime: 3
Process 6 (nice 5) found prime: 5
Process 6 (nice 5) found prime: 7
Process 6 (nice 5) found prime: 11
Process 6 with nice value 5 finished
$ 
```

In this test output, we observe three processes—Process 4, Process 5, and Process 6—each with the same nice value of 5, meaning they share equal priority. Process 4 and Process 5 start nearly simultaneously, resulting in interleaved output as they find primes (2, 3, 5, 7, and 11) at a similar pace, reflecting the scheduler's fair CPU allocation to processes with equal priority. Both Process 4 and Process 5 complete their tasks almost in sync, as indicated by their "finished" messages appearing close together. After they finish, Process 6 starts and runs without interruption, quickly finding its set of primes and completing its task. This sequence demonstrates that with equal priority, the scheduler distributes CPU time fairly among processes, allowing each to progress at a similar rate, and once some processes are complete, the remaining processes finish uninterrupted.