AOS ASSIGNMENT 3
XV6 -RISCV REPORT

- ***Introduction***

xv6 is a modern reimplementation of Sixth Edition Unix in ANSI C for multiprocessor x86 and RISC-V systems.

- ***Requirement 1 : System Call -trace***

1. ***Description :***
   When you use `strace` with a command, it will execute the given command until it finishes. During this execution, `strace` will capture and log the system calls made by the process. To specify which system calls to trace, you provide an integer value as an argument called `mask`. Each bit in this mask represents a specific system call to trace.

2. Implementation & Modifications:

a. In `kernel/sysproc.c`, the `sys_trace()` function retrieves the trace mask argument from the trapframe and then calls the `trace()` system call to store this mask in the `proc` structure.

b. The `fork()` function has been enhanced to include the copying of the trace mask from the parent process to the child process.

c. The `fork()` function was adjusted to ensure that the trace mask is correctly propagated from the parent to the child process during process creation.

d. Alterations were introduced in the `syscall()` function located in `kernel/syscall.c` to facilitate the printing of trace output when a traced system call is executed.

e. A user program named `strace.c` was developed in the `user` directory. This program handles user arguments parsing and initiates the relevant system call based on user input.

f. Within the Makefile, the inclusion of `$U/_strace` was appended to the `UPROGS` list, indicating that it should be included in the build process.

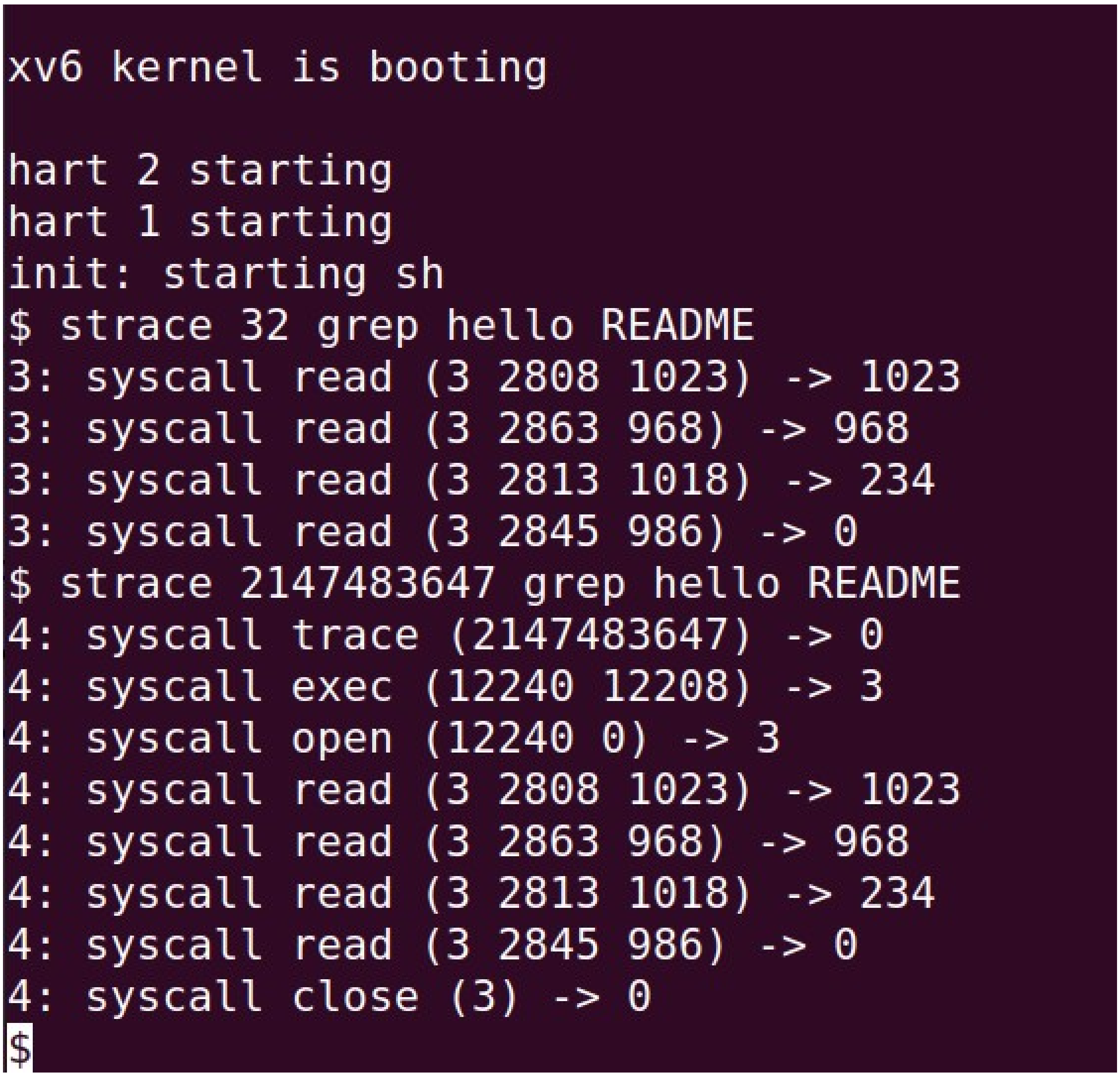
g. To accommodate the new system call, a stub was added to the `user/usys.pl` file, and a corresponding prototype for the system call was introduced in `user/user.h`.

2.

3. ***Command :***

```
strace mask [command] [args]
make clean
make qemu
$ strace 32 grep hello README
$ strace 2147483647 grep hello README
```

4. *Screenshot :*



```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ strace 32 grep hello README
3: syscall read (3 2808 1023) -> 1023
3: syscall read (3 2863 968) -> 968
3: syscall read (3 2813 1018) -> 234
3: syscall read (3 2845 986) -> 0
$ strace 2147483647 grep hello README
4: syscall trace (2147483647) -> 0
4: syscall exec (12240 12208) -> 3
4: syscall open (12240 0) -> 3
4: syscall read (3 2808 1023) -> 1023
4: syscall read (3 2863 968) -> 968
4: syscall read (3 2813 1018) -> 234
4: syscall read (3 2845 986) -> 0
4: syscall close (3) -> 0
$
```

- ***Requirement 2 : Scheduling Algorithms***

1. ***Description :***
   The xv6 operating system primarily employs a round-robin-based scheduler as its default scheduling policy. In addition to this default policy, xv6 also incorporates three additional scheduling strategies.

1.1. FCFS(First Come First Serve) Scheduling

his is a non-preemptive policy that selects the process with the lowest creation time (creation time refers to the tick number when the process was created). The process will run until it no longer needs CPU time.

2. Implementation & Modifications:

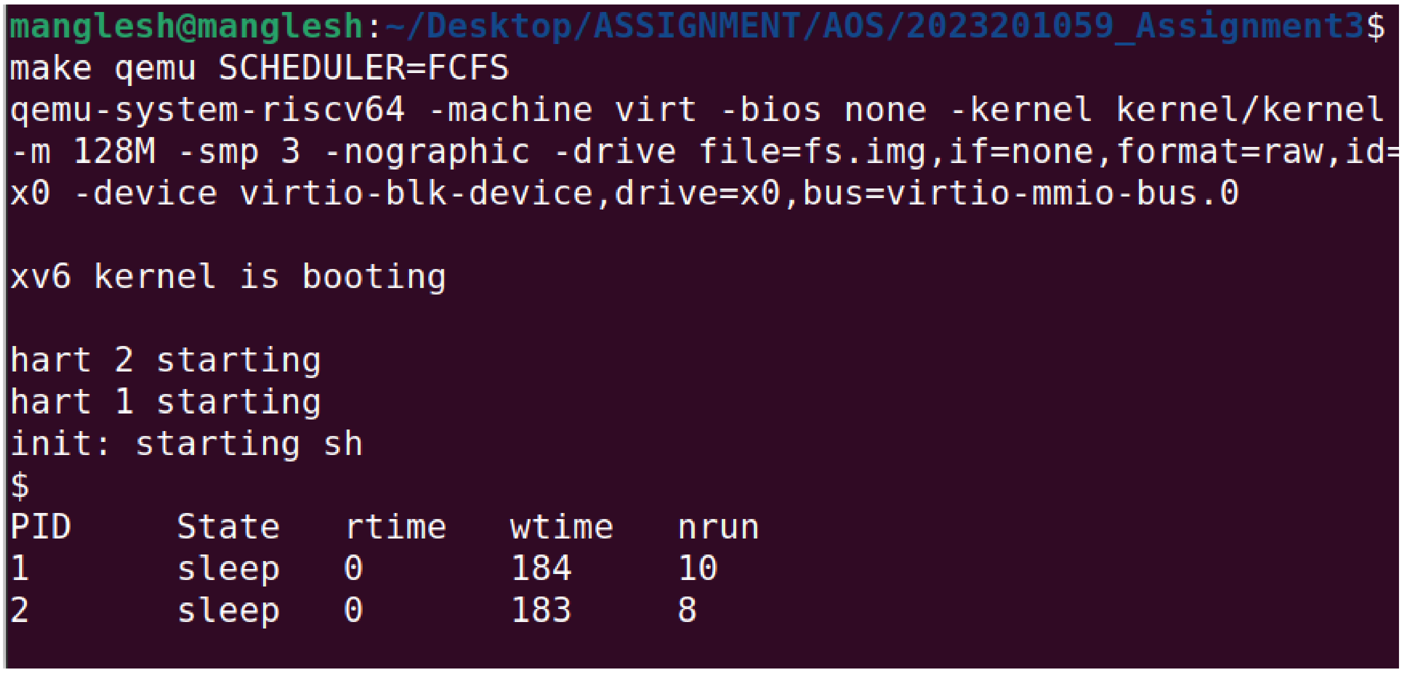
The only modification made was in `kernel/proc.c`. We run a for loop to search for the process with the lowest process creation time (`struct proc::ctime`, which stores the number of ticks when the process is allocated and initialized).

- To disable preemption, the call to `yield()` in `usertrap()` and `kerneltrap()` in `kernel/trap.c` was disabled conditionally, depending on the scheduler chosen. For FCFS, it has been disabled.

3. ***Command :***

```
 make clean
$ make qemu SCHEDULER=FCFS
$ scheduler test
```

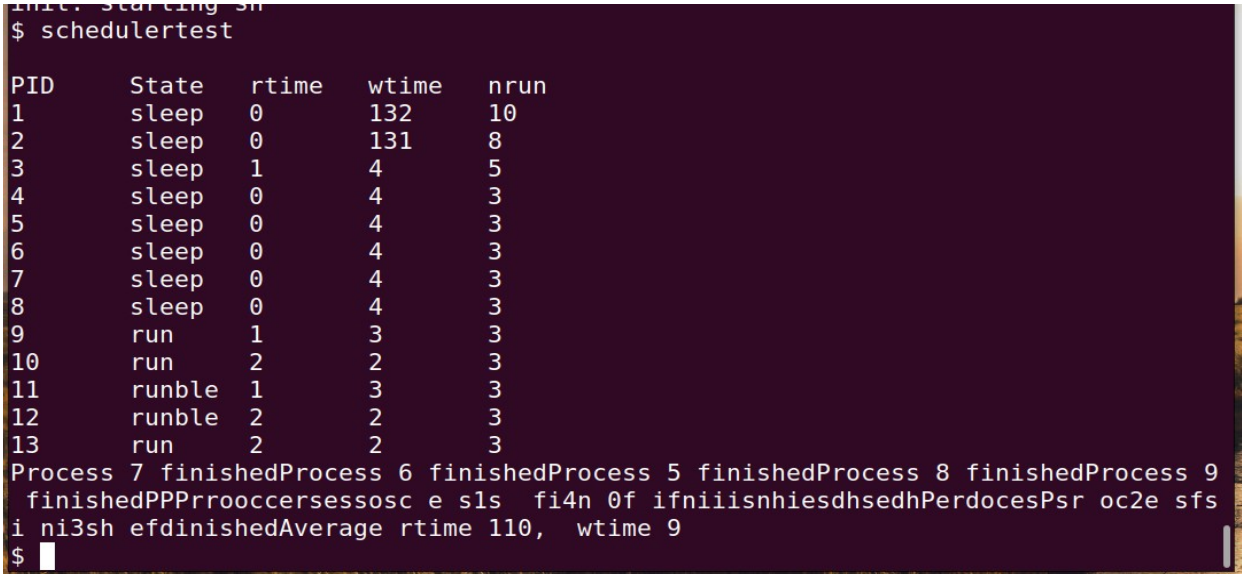
4. ***Screenshot :***



```
manglesh@manglesh:~/Desktop/ASSIGNMENT/AOS/2023201059_Assignment3$
make qemu SCHEDULER=FCFS
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel
-m 128M -smp 3 -nographic -drive file=fs.img,if=none,format=raw,id=
x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$
PID     State   rtime   wtime   nrun
1       sleep   0       184     10
2       sleep   0       183     8
```



```
init: starting sh
$ schedulertest

PID     State   rtime   wtime   nrun
1       sleep   0       132     10
2       sleep   0       131     8
3       sleep   1       4       5
4       sleep   0       4       3
5       sleep   0       4       3
6       sleep   0       4       3
7       sleep   0       4       3
8       sleep   0       4       3
9       run     1       3       3
10      run     2       2       3
11      runble  1       3       3
12      runble  2       2       3
13      run     2       2       3
Process 7 finishedProcess 6 finishedProcess 5 finishedProcess 8 finishedProcess 9
 finishedPPPrrooccersessosc e s1s  fi4n 0f ifniiisnhiesdhsedhPerdocesPsr oc2e sfs
i ni3sh efdinishedAverage rtime 110,  wtime 9
$ 
```

Average Running Time 110 ticks
Average Waiting Time : 9 ticks

1.2. a. PBS(Priority Based) Scheduling

This scheduling policy is a non-preemptive system that prioritizes processes based on their assigned priority levels. It selects the process with the highest priority to execute. When multiple processes share the same priority, we use the number of times a process has been previously scheduled to resolve the tie. If the tie persists, we resort to the process's start time to break it, with processes having lower start times given priority.

In this context, two types of priorities are at play: static priority and dynamic priority. Dynamic priority adjusts as a process runs and sleeps, ultimately influencing the scheduling decision. Static priority, on the other hand, serves as the basis for computing dynamic priority.

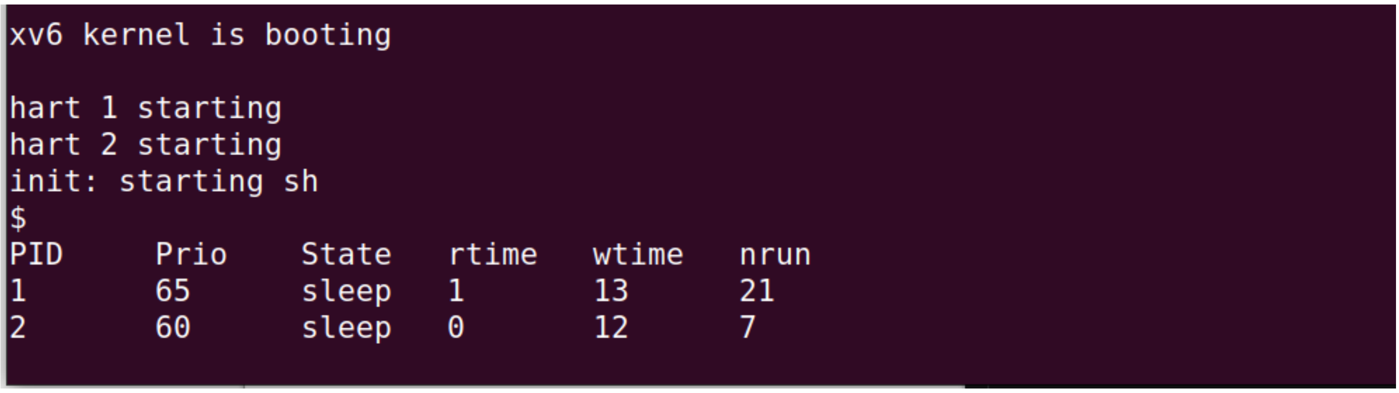
b. Implementation & Modifications:

a. We run a loop to search for the process with the highest priority (lowest dynamic priority).
b. To measure the duration of sleep, when a process enters a sleep state through the `sleep()` function in `kernel/proc.c`, the system records the current tick count in `struct proc::s_start_time`. Subsequently, upon invoking `wakeup()` in `kernel/proc.c`, the system calculates the difference between the current tick count and the previously stored time, storing it as the sleep duration in `struct proc::stime`.
c. Within `struct proc`, only the static priority is retained, typically set at 60 by default. The niceness level and dynamic priority are computed during the loop when the process to be scheduled is being selected.
d. Similar to the First-Come, First-Serve (FCFS) policy, the invocation of `yield()` has been conditionally disabled for Priority-Based Scheduling (PBS).
e. The `set_priority()` system call provides the capability to alter a process's static priority. Its implementation follows the same approach outlined in specification 1, and a corresponding user program has been developed as well.

3. ***Command :***

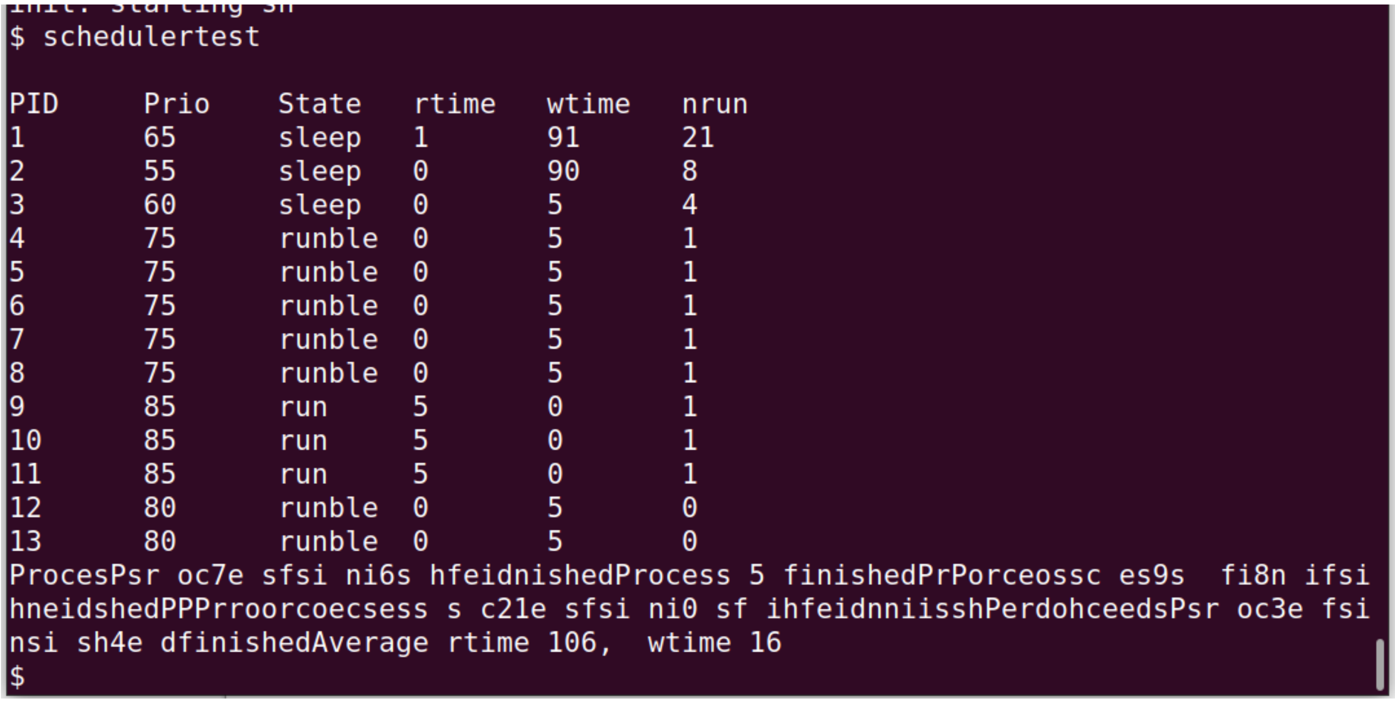
```
make clean
$ make qemu SCHEDULER=PBS
$ scheduler test
```

4. ***Screenshot :***



```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$
PID     Prio    State   rtime   wtime   nrun
1       65      sleep   1       13      21
2       60      sleep   0       12      7
```



```
init: starting sh
$ schedulertest

PID     Prio    State   rtime   wtime   nrun
1       65      sleep   1       91      21
2       55      sleep   0       90      8
3       60      sleep   0       5       4
4       75      runble  0       5       1
5       75      runble  0       5       1
6       75      runble  0       5       1
7       75      runble  0       5       1
8       75      runble  0       5       1
9       85      run     5       0       1
10      85      run     5       0       1
11      85      run     5       0       1
12      80      runble  0       5       0
13      80      runble  0       5       0
ProcesPsr oc7e sfsi ni6s hfeidnishedProcess 5 finishedPrPorceossc es9s  fi8n ifsi
hneidshedPPPrroorcoecsess s c21e sfsi ni0 sf ihfeidnniisshPerdohceedsPsr oc3e fsi
nsi sh4e dfinishedAverage rtime 106,  wtime 16
$
```

Average Running Time 106 ticks
Average Waiting Time : 16 ticks

1.3. *MLFQ(Multi Level Feedback Queue) Scheduling*

This is a streamlined preemptive scheduling approach that permits processes to transition across various priority queues contingent on their execution characteristics and CPU usage patterns.

a. In cases where a process consumes excessive CPU time, it is demoted to a lower-priority queue, thereby preserving higher-priority queues for I/O-bound and interactive processes.

b. In order to avert prolonged process neglect, an aging mechanism has been incorporated.

2. *Implementation:*

a. For priority 0 : 1 timer tick

b. For priority 1 : 2 timer ticks

c. For priority 2 : 4 timer ticks

d. For priority 3 : 8 timer ticks

e. For priority 4 : 16 timer ticks

f. An aging mechanism has been incorporated, right before the scheduling process. This involves a simple for loop that iterates through the runnable processes. If the difference between the current number of ticks and the entry time in the current queue exceeds 16 ticks, the processes are promoted to a higher-priority queue.

g. The demotion of processes after their time slice has been exhausted is managed within `kernel/trap.c`, triggered by a timer interrupt. When the time spent in the current queue surpasses 2 raised to the power of the current queue number, a demotion occurs (incrementing the `current_queue`).

h. A process's position in the queue is determined by its `struct proc::entry_time`, which keeps track of the entry time into the current queue. This entry time is reset to the current time whenever the process is scheduled, ensuring that the wait time in the queue starts at 0.

i. If a process voluntarily releases the CPU, its entry time is once again reset to the current tick count.

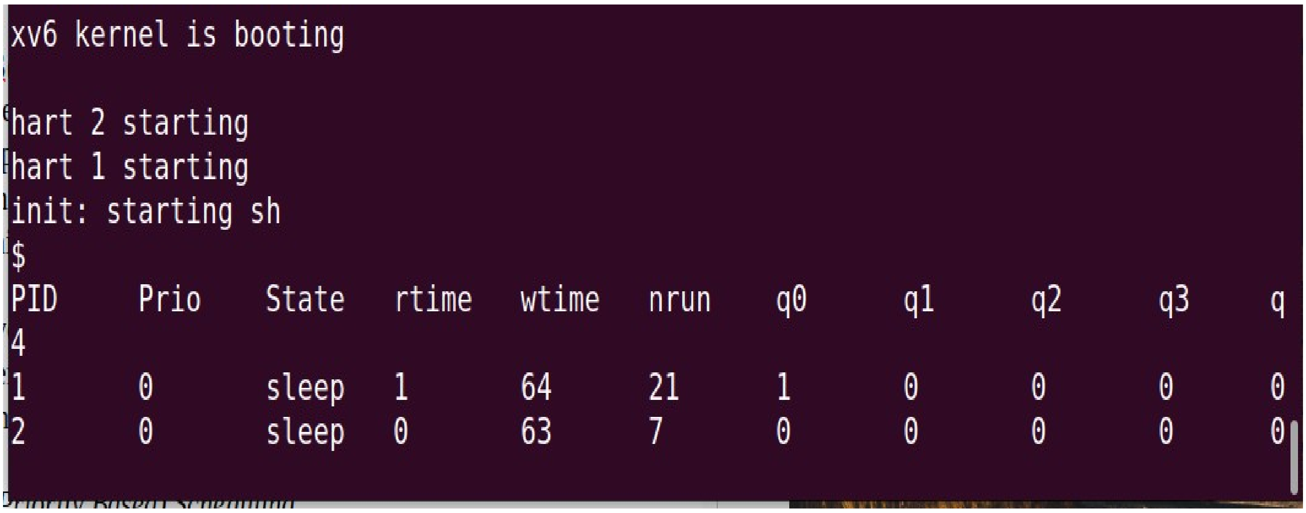
3. *Bash Command :*

make clean

a. $ make qemu SCHEDULER=MLFQ

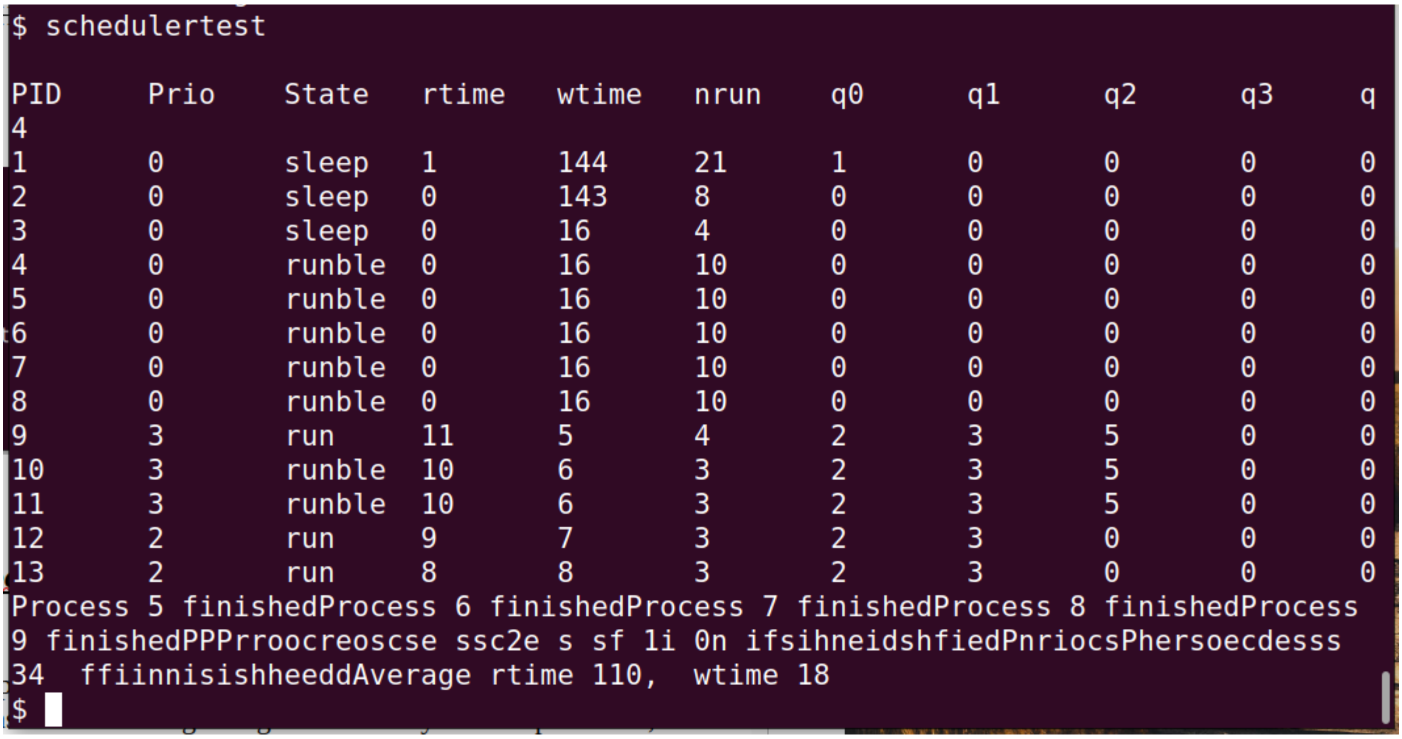
b. $ schedulertest

4. Screenshot :



```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$
PID     Prio    State   rtime   wtime   nrun    q0      q1      q2      q3      q
4
1       0       sleep   1       64      21      1       0       0       0       0
2       0       sleep   0       63      7       0       0       0       0       0
```



```
$ schedulertest

PID     Prio    State   rtime   wtime   nrun    q0      q1      q2      q3      q
4
1       0       sleep   1       144     21      1       0       0       0       0
2       0       sleep   0       143     8       0       0       0       0       0
3       0       sleep   0       16      4       0       0       0       0       0
4       0       runble  0       16      10      0       0       0       0       0
5       0       runble  0       16      10      0       0       0       0       0
6       0       runble  0       16      10      0       0       0       0       0
7       0       runble  0       16      10      0       0       0       0       0
8       0       runble  0       16      10      0       0       0       0       0
9       3       run     11      5       4       2       3       5       0       0
10      3       runble  10      6       3       2       3       5       0       0
11      3       runble  10      6       3       2       3       5       0       0
12      2       run     9       7       3       2       3       0       0       0
13      2       run     8       8       3       2       3       0       0       0
Process 5 finishedProcess 6 finishedProcess 7 finishedProcess 8 finishedProcess
9 finishedPPPrroocreoscse ssc2e s sf 1i 0n ifsihneidshfiedPnriocsPhersoecdesss
34  ffiinnisishheeddAverage rtime 110,  wtime 18
$ 
```

Average Running Time : 110 ticks
Average Waiting Time : 18 ticks

Requirement 3 : procdump

1. ***Description :***

   In this particular implementation, I've expanded the capabilities of this function to provide more extensive details regarding all currently active processes, which encompass the following information:

   ➩ Process ID
   ➩ Priority (specifically applicable to PBS and MLFQ scheduling)
   ➩ Current state
   ➩ Running time (captured in `struct proc::rtime`)
   ➩ Total waiting time (current tick count or `struct proc::etime` - creation time – running time)
   ➩ Number of times scheduled (stored in `struct proc::no_of_times_scheduled`)
   ➩ queue_ticks[5](MLFQ Scheduling Only) : Number of ticks done in each queue.

2. ***Implementation :***

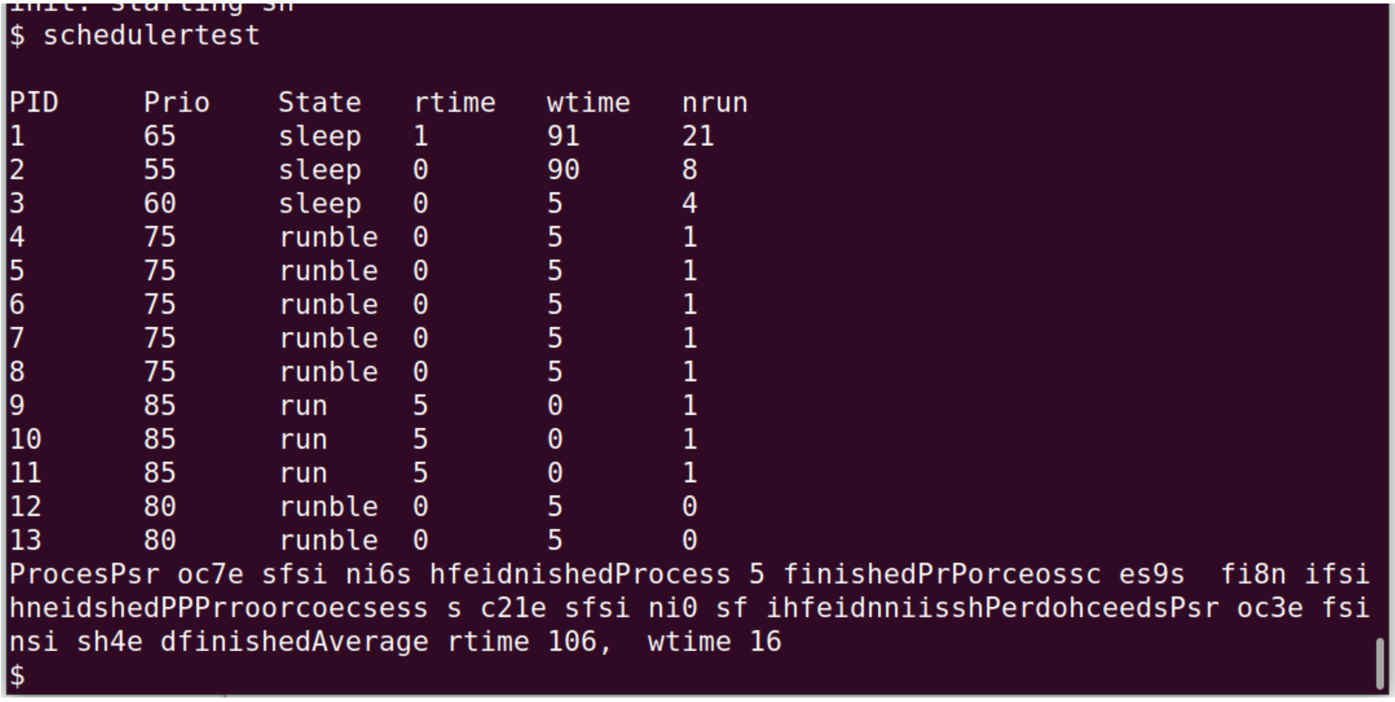
   The `procdump()` function in `kernel/proc.c` serves the purpose of displaying a comprehensive list of processes on the console when a user inputs Ctrl+P. So modifications done on it as per the requirements.

3. ***Command :***

   make clean
   $ make qemu SCHEDULER=MLFQ

   a. schedulertest
   b. Ctrl + P

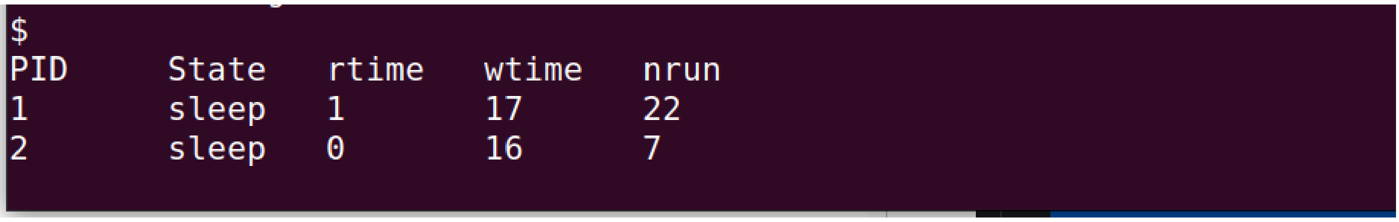
4. *Screenshots :*



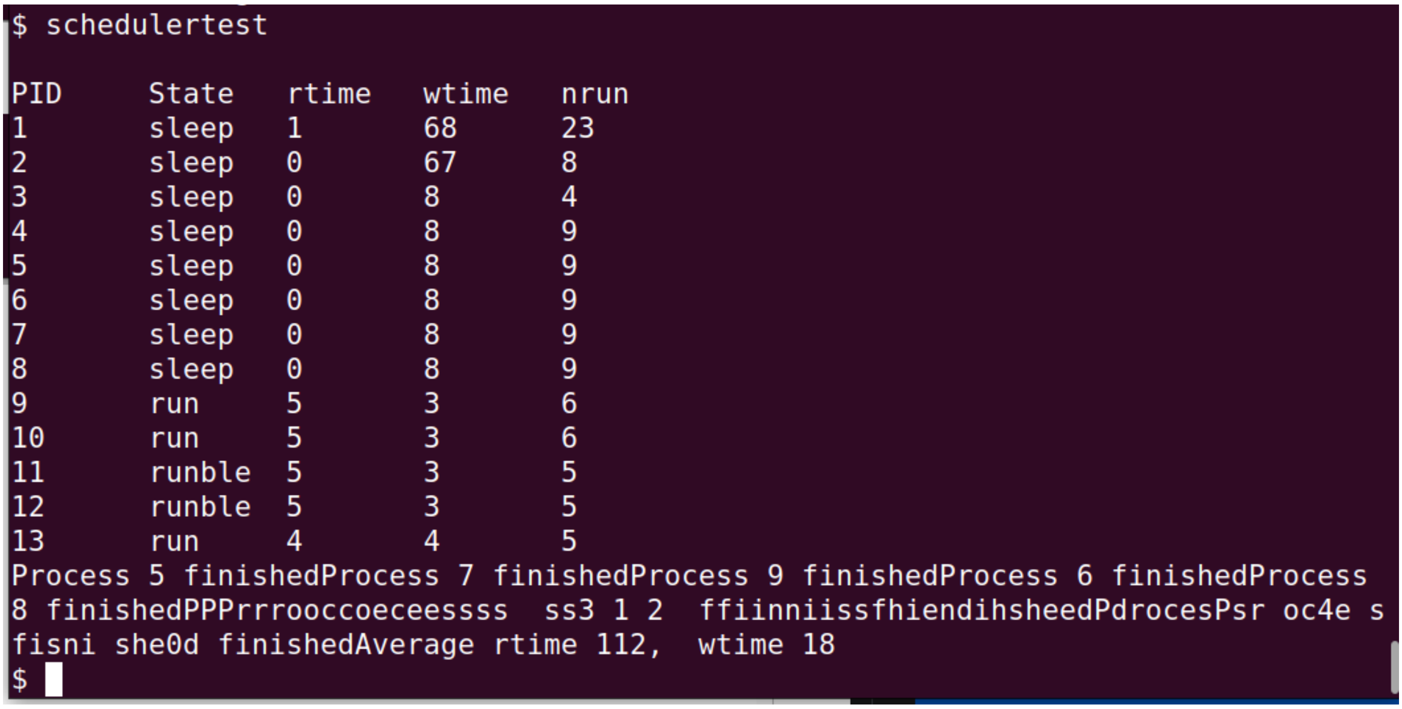
```
init: starting sh
$ schedulertest

PID     Prio    State   rtime   wtime   nrun
1       65      sleep   1       91      21
2       55      sleep   0       90      8
3       60      sleep   0       5       4
4       75      runble  0       5       1
5       75      runble  0       5       1
6       75      runble  0       5       1
7       75      runble  0       5       1
8       75      runble  0       5       1
9       85      run     5       0       1
10      85      run     5       0       1
11      85      run     5       0       1
12      80      runble  0       5       0
13      80      runble  0       5       0
ProcesPsr oc7e sfsi ni6s hfeidnishedProcess 5 finishedPrPorceossc es9s  fi8n ifsi
hneidshedPPPrroorcoecsess s c21e sfsi ni0 sf ihfeidnniisshPerdohceedsPsr oc3e fsi
nsi sh4e dfinishedAverage rtime 106,  wtime 16
$
```

4. Round -Robin (default)



```
$
PID      State    rtime    wtime    nrun
1        sleep    1        17       22
2        sleep    0        16       7
```



```
$ schedulertest

PID      State    rtime    wtime    nrun
1        sleep    1        68       23
2        sleep    0        67       8
3        sleep    0        8        4
4        sleep    0        8        9
5        sleep    0        8        9
6        sleep    0        8        9
7        sleep    0        8        9
8        sleep    0        8        9
9        run      5        3        6
10       run      5        3        6
11       runble   5        3        5
12       runble   5        3        5
13       run      4        4        5
Process 5 finishedProcess 7 finishedProcess 9 finishedProcess 6 finishedProcess
8 finishedPPPrrrooccoeceesssss  ss3 1 2  ffiinniissfhiendihsheedPdrocesPsr oc4e s
fisni she0d finishedAverage rtime 112,  wtime 18
$
```

Avgerage Running Time : 112 ticks
Avgerage Waiting Time : 18 ticks