# Modified xv6-riscv

---

# Specifications:

---

## 1. Syscall Tracing

- Added “Strace” system call in existing XV6 code.
- strace runs the specified command until it exits
- It intercepts and records the system calls which are called by a process during its Execution.
- It should take one argument, an integer mask, whose bits specify which system calls to trace

***List of files where changes are made:***

1. Makefile.mk
2. kernel/proc.c
3. kernel/proc.h
4. kernel/sysproc.c
5. kernel/syscall.h
6. user/strace.c
7. user/usys.S
8. user/usys.pl

***Steps to add this functionality:***

1. **Makefile Enhancement:**
    - ❖ Added the target `$U/_strace` to the UPROGS section in the Makefile.

2. **Kernel Modification (Header File and Define):**
   - ❖ Introduced a new variable `tracemask` in `kernel/proc.h` to store the trace mask provided by the user via a command-line argument.
   - ❖ Defined a constant `SYS_strace` with a value of 22 in `insyscal.h` to represent the new system call.

3. **Parent-Child Trace Mask Propagation:**
   - ❖ In the `fork.c` function of `kernel/proc.c`, assigned the trace mask from the parent process to the child process using the statement: `np->tracemask = p->mask;`

4. **New System Call Implementation:**
   - ❖ Implemented a `sys_trace()` function in `kernel/sysproc.c` to handle the new system call.

5. **Syscall Function Enhancement:**
   - ❖ Enhanced the existing `syscall()` function in `kernel/syscall().c` to include trace output display and utilization of the `syscall_names` and `syscall_num` arrays for additional context.

6. **User Program Creation:**
   - ❖ Developed a user program located in the user folder, specifically in `user/strace.c`, to serve as the main component of the strace program.
   - ❖ Implemented logic to handle various command-line argument cases.

7. **Integration with Makefile:**
   - ❖ Included a stub in `user/usys.pl` to instruct the Makefile to execute the Perl script `user/usys.pl`.
   - ❖ This execution results in the generation of `user/usys.S` and the addition of a syscall number to `kernel/syscall.h`.

These steps collectively enhance the system with trace mask functionality and provide a more comprehensive strace program for users.

To execute the above command sequence effectively, follow these steps:

1. **Start the Kernel:**
   - ❖ Begin by launching the kernel.

2. **Acquire the Strace Program:**
   ❖ Ensure that you have the strace program available within your system. This program is essential for tracing system calls.

3. **Execute Strace with a Sample Example:**
   ❖ Utilize the strace program by executing it with a sample command. Here's an example using the `grep` command to search for the word "hello" in a file named "README":

```shell
strace 32 grep hello README
```

This command will initiate the strace program to trace the system calls made by the `grep` command while searching for "hello" in the "README" file. The number "32" in this example represents the process ID (PID) of the `grep` command, which you may need to adapt according to your specific scenario.

***Result:***

```
spartan@pop-os:~/Downloads/folder$ make qemu SHEDULER=PBS
qemu-system-riscv64 -machine virt -bios none -kernel kernel/ker
-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

init: starting sh
$ ls
.              1 1 1024
..             1 1 1024
README         2 2 2305
cat            2 3 33000
echo           2 4 31848
forktest       2 5 16008
grep           2 6 36376
init           2 7 32344
kill           2 8 31808
ln             2 9 31632
ls             2 10 34944
mkdir          2 11 31872
rm             2 12 31856
sh             2 13 54304
stressfs       2 14 32744
usertests      2 15 180640
grind          2 16 47696
wc             2 17 33952
zombie         2 18 31224
strace         2 19 32048
setpriority    2 20 32176
schedulertest  2 21 32464
console        3 22 0
$ strace 32 grep hello README
4: syscall read (0 4112 1023) -> 1023
4: syscall read (0 4174 961) -> 961
4: syscall read (0 4151 984) -> 321
4: syscall read (0 4112 1023) -> 0
$
```

---

# 2. Scheduling

## ***Steps to add this functionality:***

- ***<u>FCFS</u>***

**1. Default Scheduler Configuration:**

❖ In the absence of user-defined preferences, a default scheduling algorithm (Round Robin) has been established to ensure smooth kernel operation.

**2. FCFS Scheduling Implementation:**

❖ FCFS selects the process with the least creation time, determined by the tick number corresponding to when the process was created. This selected process continues execution until it is terminated.

**3. Makefile Modification:**

❖ The Makefile has been modified to support the 'SCHEDULER' macro for compiling the specified scheduling algorithm.

❖ The code snippet below demonstrates the Makefile changes:

```make
ifndef SCHEDULER

SCHEDULER := RR

endif

CFLAGS += "-D$(SCHEDULER)"
```

` ` `

**4. Addition of 'timeOfCreation' Variable:**

- ❖ A 'timeOfCreation' variable has been introduced to the 'struct proc' in 'kernel/proc.h.'

**5. Initialization of 'timeOfCreation':**

- ❖ The 'timeOfCreation' variable is initialized to 0 within the 'allocproc()' function located in 'kernel/proc.c.'

**6. Scheduling Functionality Implementation:**

- ❖ The scheduling functionality has been implemented in the 'scheduler()' function within 'kernel/proc.c.' This function selects the process with the least 'timeOfCreation' value from all available processes.

**7. Disabling 'yield()' in FCFS:**

- ❖ To prevent the preemption of processes after clock interrupts in FCFS, the 'yield()' function in 'kernel/trap.c' has been disabled. This ensures that a process continues to execute until it completes its task.

- ***<u>Priority based scheduling</u>***

**1. Priority-Based Scheduler (PBS) Overview:**

- ❖ PBS is a non-preemptive Priority-Based scheduler that selects the process with the highest priority for execution.
- ❖ When multiple processes share the same priority, the number of times a process has been scheduled is used to break the tie.
- ❖ If a tie persists, the start-time of the processes is considered, with those having a lower start-time receiving a higher priority.

**2. Integration of Variables in 'struct proc':**

- To support PBS, the following variables were added to the 'struct proc' in 'kernel/proc.h':
  - static_priority
  - rtime
  - stime
  - no_of_times_scheduled
  - stime
- These variables were initialized with default values within the 'allocproc()' function in 'kernel/proc.c.'

**3. Implementation of PBS Scheduling:**

- The scheduling functionality for PBS was introduced, which calculates the dynamic priority of processes based on their static priority and 'niceness' value.
- 'niceness' is computed as:
- `int value = (p->static_priority - niceness + 5 < 100 ? p->static_priority - niceness + 5 : 100);`
- The dynamic priority is determined as:

  `max(0, min(Static Priority - niceness + 5, 100))`

**4. Addition of 'set_priority()' Function:**

- A 'set_priority()' function was added in 'kernel/proc.c' to facilitate the adjustment of process priorities.

**5. User Program for Priority Adjustment:**

- A user program named 'user/setpriority.c' was created to allow users to set process priorities.

**6. Introduction of 'sys_set_priority()' System Call:**

- To interact with the 'set_priority()' function, a 'sys_set_priority()' system call was added in 'kernel/sysproc.c.' This system call enables users to modify process priorities as needed.

- ***<u>MLFQ scheduling</u>***

**Files Modified:**

1. `kernel/proc.h`: Added fields to `struct proc` for MLFQ-related data.
2. `kernel/proc.c`: Modified `allocproc()` for process initialization and implemented the `scheduler()` function for scheduling logic.
3. `kernel/sysproc.c`: Added system calls if needed.
4. User programs and their corresponding source files

**1. Struct Changes in proc.h:**

- ❖ Add fields to the `struct proc` in `kernel/proc.h` to store information needed for MLFQ scheduling, such as queue index and statistics for each queue. For example:
    - `int queue;` to store the queue index.
    - `int ticks[QUEUES];` to maintain the ticks spent in each queue.
    - We are considering number of queues = 5
    - `int age;` to keep track of the time since the process was last dequeued.

**2. Initialize MLFQ Parameters:**

- ❖ Initialized the queue-related fields and other necessary variables during process creation in the `allocproc()` function in `kernel/proc.c`.

**3. Scheduler Function in proc.c:**

- ❖ Implemented the scheduling logic in the `scheduler()` function in `kernel/proc.c`. This function will decide which process to run based on MLFQ rules.
- ❖ It should consider queue priorities , demotion/promotion of processes, and queue rotation.

**4. Queue Management:**

- ❖ Implemented queue management functions to move processes between queues based on their behavior
    - ➢ aging
    - ➢ priority adjustments

➢ demotion.

5. **Time Quantum:**

- ❖ Implemented a time quantum management, where each queue has a different time quantum. After a process exhausts its time quantum, it is moved to a lower-priority queue (as per given in the question time quantum in each Queue i = 2 * i).

**6. Add System Calls:**

- ❖ Implemented system calls that allow users to interact with the scheduler.
- ❖ For instance, you may add system calls for setting process priorities or querying queue-related statistics.

**7. User Programs**

- ❖ **Create User Programs:** Created user programs to demonstrate and interact with the MLFQ scheduler. This may include programs that set process priorities, monitor queue statistics, or perform other actions related to MLFQ.

---

## 3. Procdump function enhancement

- This function is useful for debugging scheduling algorithms that we have implemented in step 2.
- There are various parameters that were computed in the process of scheduling implementation that have been added in the output of procdump output.

***List of files where changes are made:***

Makefile.mk
kernel/proc.c
kernel/proc.h

## *Steps to add this functionality:*

**1. Integration of 'user/schedulertest' Program:**

- ❖ The 'user/schedulertest' program has been seamlessly incorporated into the 'UPROGS' directory.
- ❖ A configuration flag ('CFLAG') has been introduced, allowing users to specify their preferred scheduler for kernel execution.

**2. Default Scheduler Configuration:**

- ❖ In the absence of user-defined preferences, a default scheduling algorithm (Round Robin) has been established to ensure smooth kernel operation.

**3. Enhanced 'procdump' Functionality:**

- ❖ To enhance the 'procdump' function's capabilities, two crucial variables have been introduced: 'runtime' (as 'rtime') and 'endtime' (as 'etime').
- ❖ The 'endtime' variable is now initialized within the 'exit' function, which resides in 'proc.c,' and triggers when a process transitions into the 'zombie' state.

**4. Refinement of the 'procdump' Function:**

- ❖ The 'procdump' function has undergone further refinement to improve its overall performance and functionality.

**5. Display of Scheduler Performance Metrics:**

- ❖ Detailed performance metrics of the scheduler are now presented, offering valuable insights into its operational efficiency.

## *Schedulers and its output:*

a. FCFS :
   i. PID (process ID)
   ii. State (state of process)
   iii. Rtime (Run time)
   iv. Wtime (waiting time)
   v. Nrun (The number of times the process has been scheduled)

```
xv6 kernel is booting

init: starting sh
$
PID     State   rtime   wtime   nrun
1       sleep   2       141     26
2       sleep   0       139     12
```

```
$ schedulertest
exec schedulertest failed
$ schedulertest

PID     State   rtime   wtime   nrun
1       sleep   0       362     13
2       sleep   0       360     15
4       sleep   0       29      6
5       run     29      0       1
6       runble  0       29      0
7       runble  0       29      0
8       runble  0       29      0
9       runble  0       29      0
10      runble  0       29      0
11      runble  0       29      0
12      runble  0       29      0
13      runble  0       29      0
14      runble  0       29      0

Process 0 finished

Process 1 finished

Process 2 finished

Process 3 finished

Process 4 finished

Process 5 finished

Process 6 finished

Process 7 finished

Process 8 finished

Process 9 finished

Average Running time: 256,  Average waiting time 56
$ 
```

b. Priority Based scheduler:
   i. PID (process ID)
   ii. Prio (priority of the current process in the range 0 to 100)
   iii. State (state of process)
   iv. Rtime (run time time)
   v. Wtime
   vi. Nrun (The number of times the process has been scheduled)

```
xv6 kernel is booting

init: starting sh
$ schedulertest

PID     Prio    State   rtime   wtime   nrun
1       60      sleep   0       356     24
2       65      sleep   1       352     13
3       65      sleep   1       14      6
4       75      runble  0       14      1
5       75      runble  0       14      1
6       75      runble  0       14      1
7       75      runble  0       14      1
8       75      runble  0       14      1
9       85      run     14      0       1
10      80      runble  0       14      0
11      80      runble  0       14      0
12      80      runble  0       14      0
13      80      runble  0       14      0
```

```
init: starting sh
$ schedulertest

PID     Prio    State   rtime   wtime   nrun
1       65      sleep   1       218     24
2       55      sleep   0       217     13
3       60      sleep   0       20      6
4       75      runble  0       20      1
5       75      runble  0       20      1
6       75      runble  0       20      1
7       75      runble  0       20      1
8       75      runble  0       20      1
9       85      run     20      0       1
10      80      runble  0       20      0
11      80      runble  0       20      0
12      80      runble  0       20      0
13      80      runble  0       20      0

Process 5 finished

Process 6 finished

Process 7 finished

Process 8 finished

Process 4 finished

Process 0 finished

Process 1 finished

Process 2 finished

Process 3 finished

Process 9 finished

Average Running time: 173,  Average waiting time 28
$ 
```

c. Multilevel Feedback Queue Scheduling:
   i. PID (Process ID)
   ii. Prio (priority)
   iii. State (state of the process)
   iv. Rtime (run time)
   v. Wtime (waiting time)
   vi. Nrun (The number of times the process has been scheduled)
   vii. Q0 (Number of ticks done in queue)
   viii. Q1 (Number of ticks done in queue)
   ix. Q2 (Number of ticks done in queue)
   x. Q3 (Number of ticks done in queue)
   xi. Q4 (Number of ticks done in queue)

```
xv6 kernel is booting

init: starting sh
$
PID     Prio    State   rtime   wtime   nrun    q0      q1      q2      q3      q4
1       0       sleep   2       64      24      2       0       0       0       0
2       0       sleep   0       62      12      0       0       0       0       0

$ schedulertest

PID     Prio    State   rtime   wtime   nrun    q0      q1      q2      q3      q4
1       0       sleep   2       182     24      2       0       0       0       0
2       0       sleep   0       180     15      0       0       0       0       0
4       0       sleep   0       14      6       0       0       0       0       0
5       0       sleep   0       13      4       0       0       0       0       0
6       0       sleep   0       13      4       0       0       0       0       0
7       0       sleep   0       13      4       0       0       0       0       0
8       0       sleep   0       13      4       0       0       0       0       0
9       0       sleep   0       13      4       0       0       0       0       0
10      2       runble  5       8       2       2       3       0       0       0
11      1       run     2       11      2       2       0       0       0       0
12      1       runble  2       11      1       2       0       0       0       0
13      1       runble  2       11      1       2       0       0       0       0
14      1       runble  2       11      1       2       0       0       0       0
```

```
Process 0 finished

Process 1 finished

Process 2 finished

Process 3 finished

Process 4 finished

PID     Prio    State   rtime   wtime   nrun    q0      q1      q2      q3      q4
1       0       sleep   2       381     24      2       0       0       0       0
2       0       sleep   0       379     15      0       0       0       0       0
4       0       sleep   0       213     8       0       0       0       0       0
10      3       runble  45      167     10      2       3       40      141     0
11      2       run     41      171     10      2       3       35      141     0
12      3       runble  40      172     9       2       3       35      121     0
13      3       runble  43      169     10      2       6       53      120     0
14      3       runble  43      169     10      2       6       53      120     0

Process 8 finished

Process 5 finished

Process 6 finished

Process 7 finished

Process 9 finished

Average Running time: 203,  Average waiting time 26
$ 
```

---

# 4. Benchmark Testing

A new file created in the `user/schedulertest.c` was added to test the implemented schedulers.

- **Round Robin**
    - Average run time: 219
    - Average waiting time 30

- **First Come First Serve**
  - Average run time: 256
  - Average waiting time 56

- **Priority Based Scheduling**
  - Average run time: 173
  - Average waiting time 28

- **Multilevel feedback queue**
  - Average run time: 203
  - Average waiting time 26

**Conclusion:**

Certainly, when analyzing the performance of different scheduling algorithms, it becomes evident that FCFS (First-Come-First-Serve) tends to deliver the least favorable results. This is primarily due to the possibility of extended waiting times for other processes if a CPU-bound task with an extended execution duration is prioritized first.

In comparison, MLFQ (Multi-Level Feedback Queue) stands out as the top-performing scheduling algorithm, with PBS (Priority-Based Scheduling) coming in as a strong runner-up. Meanwhile, Round Robin trails behind in terms of performance.

These conclusions were drawn based on an evaluation conducted using the benchmark program found in user/schedulertest.c.