Assignment 4 : Enhancing XV-6

Operating Systems and Networks, Monsoon 2022

Deadline: 12th October , 5 PM

---

xv6 is a simplified operating system developed at MIT. Its main purpose is to explain the main concepts of the operating system by studying an example kernel. xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix version 6 (v6). xv6 loosely follows the structure and style of v6, but is implemented for a modern RISC-V multiprocessor using ANSI C. In this assignment you will be tweaking the Xv6 operating system as a part of this assignment. You can download the xv6 source code from here. You can see the install instructions here. You might find resource here to be helpful.

Specification 1: System Calls [15 marks]

System calls provide an interface to the user programs to communicate requests to the operating systems. In this specification, you're tasked to implement the following syscalls:

System Call 1 : `trace` [5 marks]

- Add the system call `trace` and an accompanying user program `strace` . The command will be executed as follows :

```
strace mask command [args]
```

- `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.
- say for example: to trace the ith system call, a program calls strace 1<<i, where i is the syscall number (look in kernel/syscall.h).

- You have to modify the xv6 kernel to print out a line when each system call is about to return if the system call's number is set in the mask
- Following things should be kept in mind while implementing the strace syscall :
  - The line should contain:

    1. The process id
    2. The name of the system call
    3. The decimal value of the arguments(xv6 passes arguments via registers).

    NOTE: You must always interpret the register value as an integer. (see kernel/syscall.c)

    4. The return value of the syscall.

- **NOTE**: The trace system call should enable tracing for the process that calls it and any children that it subsequently forks, but should not affect other processes.
- Example :

```
$ strace 32 grep hello README
6: syscall read (3 2736 1023) -> 1023
6: syscall read (3 2793 966) -> 966
6: syscall read (3 2764 995) -> 70
6: syscall read (3 2736 1023) -> 0
$ strace 2147483647 grep hello README
3: syscall trace (2147483647) -> 0
3: syscall exec (12240 11872) -> 3
3: syscall open (12240 0) -> 3
3: syscall read (3 2736 1023) -> 1023
3: syscall read (3 2793 966) -> 966
3: syscall read (3 2764 995) -> 70
3: syscall read (3 2736 1023) -> 0
3: syscall close (3) -> 0
```

System call 2 : `sigalarm` and `sigreturn` [10 marks]

In this specification you'll add a feature to xv6 that periodically alerts a process as it uses CPU time. This might be useful for compute-bound processes that want to limit how much CPU time they chew up, or for processes that want to compute but also want

to take some periodic action. More generally, you'll be implementing a primitive form of user-level interrupt/fault handlers like the `SIGCHILD` handler in the previous assignment.

You should add a new `sigalarm(interval, handler)` system call. If an application calls `alarm(n, fn)` , then after every n "ticks" of CPU time that the program consumes, the kernel will cause application function `fn` to be called. When `fn` returns, the application will resume where it left off.

Add another system call `sigreturn()`, to reset the process state to before the `handler` was called. This system call needs to be made at the end of the handler so the process can resume where it left off.

You can find an example usage of this code at https://gist.github.com/AbhijnanVegi/0df4ec53cea38618b36ceb28c0ae573b

---

Specification 2: Scheduling [60 marks]

The default scheduler of xv6 is round-robin-based. In this task, you'll implement three other scheduling policies and incorporate them in xv6. The kernel shall only use one scheduling policy which will be declared at compile time, with default being round robin in case none is specified.

Modify the `makefile` to support the SCHEDULER macro to compile the specified scheduling algorithm. Use the flags for compilation:-

- First Come First Serve = `FCFS`
- Priority Based = `PBS`
- Multilevel Feedback Queue = `MLFQ`

a. FCFS (First Come First Serve) (5)

Implement a 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.

HINTS:

1. Edit the struct proc (used for storing per-process information) in kernel/proc.h to store extra information about the process.
2. Modify the allocproc() function to set up values when the process starts. (see kernel/proc.h)
3. Use pre-processor directives to declare the alternate scheduling policy in scheduler() in kernel/proc.h.
4. Disable the preemption of the process after the clock interrupts in kernel/trap.c

b. Lottery Based Scheduler (10)

Implement a preemptive scheduler that assigns a time slice to the process randomly in proportion to the number of tickets it owns. That is the probability that the process runs in a given time slice is proportional to the number of tickets owned by it.

Implement a system call `int settickets(int number)` , which sets the number of tickets of the calling process. By default, each process should get one ticket; calling this routine makes it such that a process can raise the number of tickets it receives, and thus receive a higher proportion of CPU cycles.

Note: You'll need to assign tickets to a process when it is created. Specifically, you'll need to make sure a child process *inherits* the same number of tickets as its parents.

c. PBS (Priority Based Scheduler) (15)

Implement a non-preemptive priority-based scheduler that selects the process with the highest priority for execution. In case two or more processes have the same priority, we use the number of times the process has been scheduled to break the tie. If the tie remains, use the start-time of the process to break the tie(processes with lower start times should be scheduled further).

There are two types of priorities.

- The Static Priority of a process (SP) can be in the range [0,100], the smaller value will represent higher priority. Set the default priority of a process as 60. The lower the value the higher the priority.
- Dynamic Priority (DP) is calculated from static priority and niceness.
- The niceness is an integer in the range [0, 10] that measures what percentage of the time the process was sleeping (see sleep() in kernel/proc.c. xv6 allows a

process to give up the CPU and sleep while waiting for an event, and allows another process to wake the first process up)

- The meaning of niceness values are:
  - 5 is neutral
  - 10 helps priority by 5
  - 0 hurts priority by 5
- To calculate the niceness:
  - Record for how many ticks the process was sleeping and running from the last time it was scheduled by the kernel. (see sleep() & wakeup() in kernel/proc.c)
  - New processes start with niceness equal to 5. After scheduling the process, compute the niceness as follows:

$$niceness = Int\left(\frac{Ticks\ spent\ in\ (sleeping)\ state}{Tick\ spent\ in\ (running + sleeping)\ state} * 10\right)$$

- Use Dynamic Priority to schedule processes which is given as:

$$DP = max(0, min(SP - niceness + 5, 100))$$

- To change the Static Priority add a new system call set_priority(). This resets the niceness to 5 as well.

  int set_priority(int new_priority, int pid)

  The system call returns the old Static Priority of the process. In case the priority of the process increases(the value is lower than before), then rescheduling should be done.

  Also make sure to implement a user program setpriority, which uses the above system call to change the priority. And takes the syscall arguments as command-line arguments.

```
setpriority priority pid
```

- **Notes:**
  - To implement set_priority() system call refer to the hints of specification 1 (Trace system call)
  - Don’t forget to update the niceness of the process

c. Multi Level Feedback Queue (MLFQ) (25)

Implement a simplified preemptive MLFQ scheduler that allows processes to move between different priority queues based on their behavior and CPU bursts.

- If a process uses too much CPU time, it is pushed to a lower priority queue, leaving I/O bound and interactive processes in the higher priority queues.
- To prevent starvation, implement aging.

Details:

1. Create five priority queues, giving the highest priority to queue number 0 and lowest priority to queue number 4
2. The time-slice are as follows:
   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

   NOTE: Here tick refers to the clock interrupt timer. (see kernel/trap.c)

Synopsis for the scheduler:-

1. On the initiation of a process, push it to the end of the highest priority queue.
2. You should always run the processes that are in the highest priority queue that is not empty.

Example:

Initial Condition: A process is running in queue number 2 and there are no processes in both queues 1 and 0.

Now if another process enters in queue 0, then the current running process (residing in queue number 2) must be preempted and the process in queue 0 should be allocated the CPU.

3. When the process completes, it leaves the system.
4. If the process uses the complete time slice assigned for its current priority queue, it is preempted and inserted at the end of the next lower level queue.
5. If a process voluntarily relinquishes control of the CPU(eg. For doing I/O), it leaves the queuing network, and when the process becomes ready again after the I/O, it is inserted at the tail of the same queue, from which it is relinquished earlier (Q: Explain in the README how could this be exploited by a process(see specification 4 below)).
6. A round-robin scheduler should be used for processes at the lowest priority queue.
7. To prevent starvation, implement aging of the processes:
    a. If the wait time of a process in a priority queue (other than priority 0) exceeds a given limit (assign a suitable limit to prevent starvation), their priority is increased and they are pushed to the next higher priority queue.
    b. The wait time is reset to 0 whenever a process gets selected by the scheduler or if a change in the queue takes place (because of aging).

Specification 3: Copy-on-write fork (15)

In xv6 the `fork` system call creates a duplicate process of the parent process and also copies the memory content of the parent process into the child process. This results in inefficient usage of memory since the child may only read from memory.

The idea behind a copy-on-write is that when a parent process creates a child process then both of these processes initially will share the same pages in memory and these shared pages will be marked as copy-on-write which means that if any of these processes will try to modify the shared pages then only a copy of these pages will be

created and the modifications will be done on the copy of pages by that process and thus not affecting the other process.

The basic plan in COW fork is for the parent and child to initially share all physical pages, but
to map them read-only. Thus, when the child or parent executes a store instruction, the RISC-V
CPU raises a page-fault exception. In response to this exception, the kernel makes a copy of the
page that contains the faulted address. It maps one copy read/write in the child's address space and the other copy read/write in the parent's address space. After updating the page tables, the kernel resumes the faulting process at the instruction that caused the fault. Because the kernel has updated the relevant PTE to allow writes, the faulting instruction will now execute without a fault.

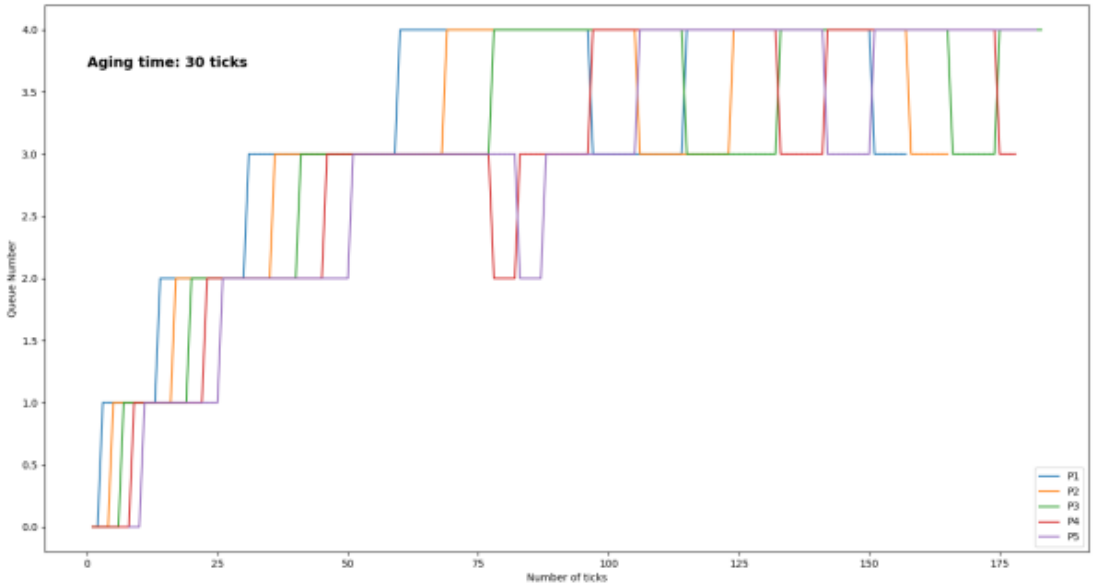
Implement copy-on-write (COW) fork in xv6.

Specification 4 : Report [10 marks]

- Include a well-written description of how you put the other four standards into practice.
- The report must contain explanation about the implementation of the various implemented scheduling algorithms.
- Include the performance comparison between the default and 3 implemented policies in the README by showing the average waiting and running times for processes

MLFQ scheduling analysis [5 marks]

- Create timeline graphs for processes that are being managed by MLFQ Scheduler. 'Variate the length of time that each process consumes the CPU before willingly quitting using the benchmark from Specification 2 (Hint: use sleep()). The graph should be a timeline/scatter plot between queue_id(y-axis) and time elapsed(x-axis) from start with color-coded processes.
- Example :



Guidelines:

1. Submission format: <RollNo>_Assignment4.tar.gz. **Please make a note of the naming and the compression format, any discrepancies in the above, will result in penalties.**
2. Submission by email to TAs will not be accepted, please adhere to the deadline.
3. We will use more than 2 CPUs to test for FCFS and PBS. But for the MLFQ scheduler, we will only use 1 CPU.
4. Make sure you write a README that contains the report components of the assignment. Including a README file is NECESSARY
5. Whenever you add new files do not forget to add them to the Makefile so that they get included in the build.
6. Make sure to include a detailed **report, describing the implementation of the scheduling algorithms [specification 2] , failing which will result in direct 0 marking for those questions.**

Do NOT take codes from seniors or your batch mates, by any chance. We will extensively evaluate cheating scenarios along with the previous few year’s submissions.

Viva will be conducted during the evaluations, related to your code and also the logic/concept involved. If you're unable to answer them, you'll get no marks for that feature/topic that you've implemented.