CS3.304 - Advanced Operating Systems
Assignment 3: Enhancing XV-6 OS
Deadline: 12th September 2023, 11:59:00 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. 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 installation instructions here.

Requirement 1: System Call:

- 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 printed while executing the strace program the strace syscall:
  - The process id.
  - The name of the system call.
  - 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)
  - 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.
  To implement a system call, refer here

- 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
```

Requirement 2: Scheduling:

The default scheduler of xv6 is round-robin-based. In this task, you'll implement 3 other scheduling policies and incorporate them in xv6. The kernel shall only use one scheduling policy which will be declared at compilation time.
Modify the Makefile to support SCHEDULER - a macro for the compilation of the specified scheduling algorithm.

$ make qemu SCHEDULER=MLFQ

Use the flags for compilation:
- First Come First Serve = FCFS
- Priority Based = PBS
- Multilevel Feedback Queue = MLFQ

1. **First come First server (FCFS):** 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 tim
   Hint:
   - Edit the struct proc (used for storing per-process information) in kernel/proc.h to store extra information about the process.

- Modify the allocproc() function to set up values when the process starts. (see kernel/proc.h)
- Use preprocessor directives to declare the alternate scheduling policy in scheduler() in kernel/proc.h.
- Disable the preemption of the process after the clock interrupts in kernel/trap.c

2. **Priority Based Scheduler (PBS):** 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 waiting for an event, and allows another process to wake the first process up)
   - The meaning of niceness values is:
     - 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:

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

- 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

Hint:

- To implement a system call, refer here.
- Don't forget to update the niceness of the process.

3. **Multilevel Feedback queue scheduling (MLFQ):** 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:

- Create five priority queues, giving the highest priority to queue number0 and lowest priority to queue number 4.

- 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

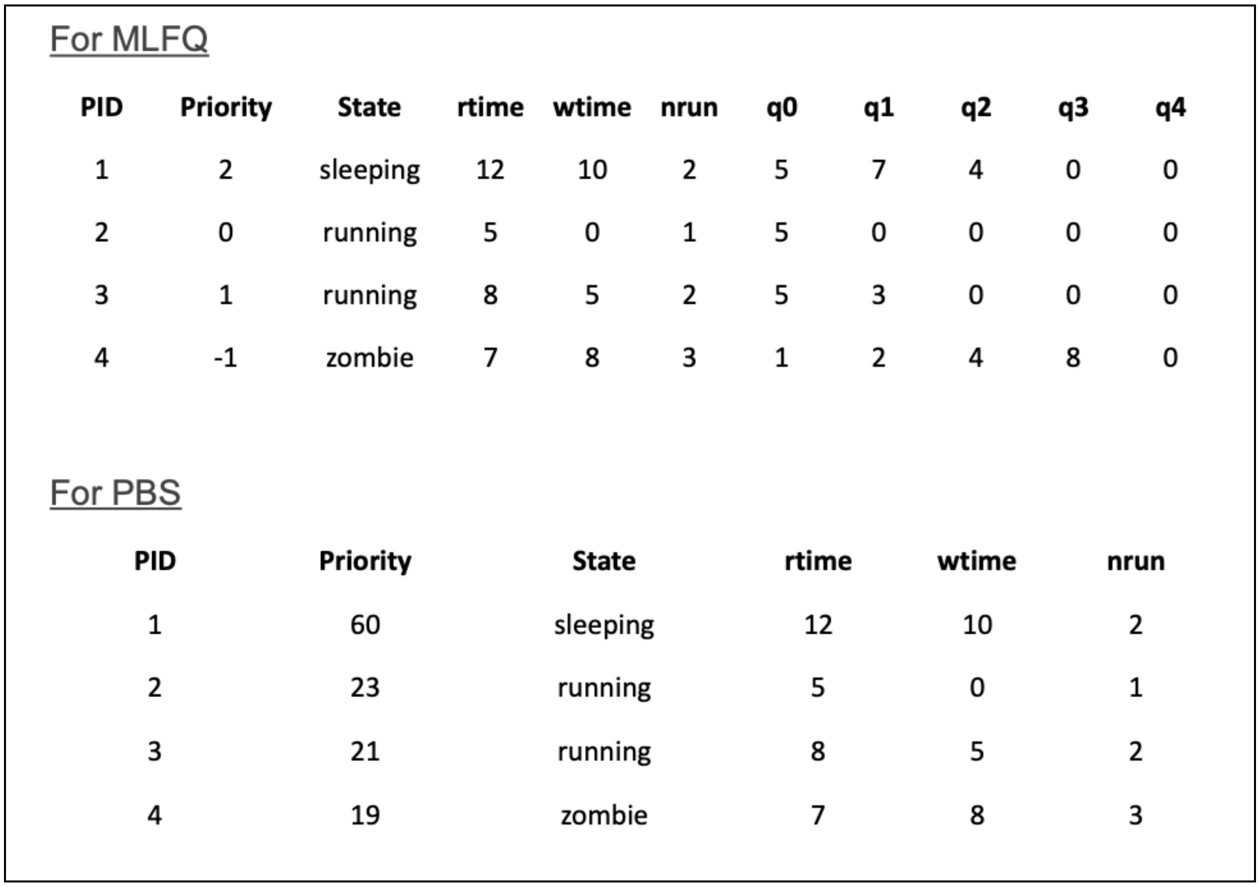
  NOTE: Here tick refers to the clock interrupt timer. (see kernel/trap.c)

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.FordoingI/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:
   - If the wait time of a process in a Priority queue (other than priority 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.
   - The wait time is reset to 0 whenever process gets selected by the scheduler or if a change in the queue takes place (because of aging).

Requirement 3:

procdump is a function that is useful for debugging (see kernel/proc.c). It prints a list of processes to the console when a user types *ctrl-p* on the console. In this task, you have to extend this function to print more information about all the active processes. Check the sample output given below to know what all information needs to be displayed.



For MLFQ

PID	Priority	State	rtime	wtime	nrun	q0	q1	q2	q3	q4
1	2	sleeping	12	10	2	5	7	4	0	0
2	0	running	5	0	1	5	0	0	0	0
3	1	running	8	5	2	5	3	0	0	0
4	-1	zombie	7	8	3	1	2	4	8	0

For PBS

PID	Priority	State	rtime	wtime	nrun
1	60	sleeping	12	10	2
2	23	running	5	0	1
3	21	running	8	5	2
4	19	zombie	7	8	3

NOTE:

1. **The above shown image is just an example output of what is expected.**
2. **priority** [PBS, MLFQ only]:
    - PBS: Current dynamic-priority of the process.It will range from [0, 100].
    - MLFQ: This corresponds to the queue number of the process. Put -1 in case the process is currently not queued in any of the queues.
3. **state**: The current state of the process. (see enum procstate in kernel/proc.h)
4. **rtime**: Total ticks for which the process ran on the CPU till now.
5. **wtime**: This corresponds to the total waiting time for the process. However, In the case of the MLFQ scheduler, this is the wait time in the current queue.
6. **nrun**: Number of times the process was picked by the scheduler.

7. **q_i** [MLFQ only]: Number of ticks the process has spent in each of the 5 queues. Sum of q_i = wait time of the process + run time of the process. (Runtime of one execution instance is to be added to the q_i for the queue from which it was picked).

Requirement 4:

● Include well-written documentation of how you implemented the above 3 requirements. The report document should contain all the steps and procedures followed to accomplish the activity. List of all the files where changes have been made, list of all the files that had been added new and what all changes have been made are to be well-documented.

● Include the performance comparison between the default and 3 implemented scheduling policies in the document by showing the average waiting and running times for processes. (You may find the use of waitx() syscall to be quite helpful).

Guidelines:

- Make sure you write a README that contains the report components of the assignment. Including a README file is NECESSARY.

- Whenever you add, new files do not forget to add them to the Makefile so that they get included in the build.

- Make sure to include a detailed report, describing the implementation of the scheduling algorithms, failing which will result in direct 0 marking for those questions.

Submission Format:

- The XV-6 OS folder will be originally as “xv6-riscv-riscv”.
- Rename the folder to <Roll_Number>_Assignment3.
- Add the Report named as <Roll_Number>_Report.pdf in the same folder.
- Finally zip the folder and the final submission file should be as <Roll_Number>_Assignment3.zip