Assignment 5
Enhancing xv6 OS
Operating Systems and Networks
Monsoon 2020

Deadline: 29th October, 11:55 PM.

Xv6 is a simplified operating system developed at MIT. You will be tweaking the Xv6 operating system as a part of this assignment.

Task 1

waitx sys call

As a part of this task, you're expected to extend the current proc structure and add new fields ctime, etime and rtime for creation time, end-time, and total time respectively of a process. When a new process gets created the kernel code should update the process creation time. The run-time should get updated after every clock tick for the process. To extract this information from the kernel add a new system call which extends wait system call. The new call will be:

int waitx(int* wtime, int* rtime)

The two arguments are pointers to integers to which waitx will assign the total number of clock ticks during which process was waiting and the total number of clock ticks when the process was running. The return values for waitx should be the same as that of the wait system-call. Create a test program which utilizes the waitx system call by creating a '**time**' like command for the same.

Note: This can be used to implement your scheduler functions.

ps (user program)

You will need to create a new user program: **ps.** This user program returns some basic information about all the active processes. Check the sample output given below to know what all information needs to be displayed.

For keeping a track of all the required information for all the processes, you will need to modify the existing processor structure.

Note that **ps** must be a user program (we should be able to type **ps** in the terminal, and output should be displayed. You are free to create whatever system call you need to get the desired information about all the processes from the kernel space. Ideally, one system call to print all the information of the process table should be enough)

Sample output:

PID	Priority	State	r_time	w_time	n_run	cur_q	q0	q1	q2	q3	q4
1	60	sleeping	12	10	2	3	5	7	0	0	0
2	23	running	5	0	1	1	5	0	0	0	0
3	21	running	8	5	2	1	5	3	0	0	0
4	19	zombie	7	10	2	-1	5	2	0	0	0

- **Priority** - Current priority of the process (defined as per the need of schedulers below)
- **State** - Current state of the process
- **r-time** - Total time for which the process ran on CPU till now (use a suitable unit of time)
- **w-time** - Time for which the process has been waiting (reset this to 0 whenever the process gets to run on CPU or if a change in the queue takes place (in the case of MLFQ scheduler))
- **n_run** - Number of times the process was picked by the scheduler
- **cur_q** - Current queue (check task 2 part C)
- **q{i}** - Number of ticks the process has received at each of the 5 queues

Set appropriate default values for fields which do not make sense (like cur_q does not make sense if the scheduler is FCFS).
Please make sure that you implement this particular user program, as we will use this to evaluate your code and it will also help you in debugging your code.

Task 2

The default scheduler of xv6 is a round-robin based scheduler. In this task, you'll implement 3 other scheduling policies and incorporate them in Xv6.

(a) First come - First Served (FCFS)

Implement a non-preemptive policy that selects the process with the lowest creation time. The process runs until it no longer needs CPU time.

(b) Priority Based Scheduler

A priority-based scheduler selects the process with the highest priority for execution. In case two or more processes have the same priority, we choose them in a round-robin fashion. The priority of a process can be in the range [0,100], **the smaller value will represent higher priority**. Set the default priority of a process as 60. To change the default priority add a new system call **set_priority** which can change the priority of a process.

int set_priority(int new_priority, int pid)

The system-call returns the old-priority value 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.

(it will be run in the following way: **setPriority new_priority pid**)

(c) Multi-level Feedback queue scheduling

MLFQ scheduler 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 for higher priority queues. Also, to prevent starvation, it implements aging.

Keeping all these benefits in mind, implement a simplified multi-level feedback queue scheduler.

Scheduler Details:-

1. Create five priority queues, with the highest priority being number as 0 and the bottom queue with the lowest priority as 4.
2. Assign a suitable value for 1 tick of CPU timer.
3. The time-slice for priority 0 should be 1 timer tick. The times-slice for priority 1 is 2 timer ticks; for priority 2, it is 4 timer ticks; for priority 3, it is 8 timer ticks; for priority 4, it is 16 timer ticks.

Procedure:-

1. On the initiation of a process, push it to the end of the highest priority queue.

2. The highest priority queue should be running always, if not empty.

3. If 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, 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 **(Explain in**

the report how could this be exploited by a process).

6. A round-robin scheduler should be used for processes at the lowest priority queue.

7. To prevent starvation, implement the aging phenomenon:-

a. If the wait time of a process in lower priority queues 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).

Modify the Makefile to support SCHEDULER - a macro for the compilation of the specified scheduling algorithm. Use the flags for compilation:-

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

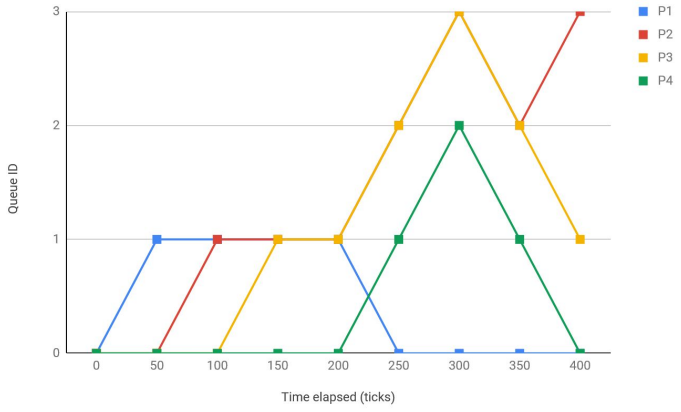
Example:- make qemu SCHEDULER=MLFQ

Write a sample benchmark program (make sure it runs for at least 20 secs on your laptop) which can be used to compare the performances of the scheduling algorithms and demonstrate using that program during the evals. **Also, include the performance comparison between the default and 3 implemented policies in the report**

Bonus:-

Plot timeline graphs for processes running with MLFQ Scheduler. Use the benchmark/workload from Task 2 to vary how long each process uses the CPU before relinquishing voluntarily (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. Add to the report the observations recorded for different types of processes. Example:-



Note:- Plotting of the graph can be done in the language of your choice.

Guidelines

1. Submission format: RollNo_Assignment5.tar.gz.
2. Submission by email to TAs will not be accepted.
3. Any copy cases found will lead to serious consequences.
4. Make sure you write a readme which briefly describes the implementation (This might carry weightage).
5. Whenever you add new files do not forget to add them to the Makefile so that they get included in the build.
6. The xv6 OS base code can be downloaded from

   https://github.com/mit-pdos/xv6-public