CS 333: Operating Systems Lab
Autumn 2022
Lab6: New Scheduling Policies in xv6

The scope of this lab is to understand the current CPU scheduler of xv6 and to design and implement a new scheduling algorithm in xv6. In this lab, we will modify the xv6 scheduler to allocate a greater quanta of CPU time to some processes .

Part 0: Overview

- Setting up xv6 (for details follow this link)
- In your Makefile, replace CPUS := 2 with CPUS := 1 . The old setting runs xv6 with two CPU cores, but you only need to do the scheduling for a single core in this lab.
- Download, read and use as reference the xv6 source code companion book.
    - https://pdos.csail.mit.edu/6.828/2017/xv6/book-rev10.pdf
      (Page numbers - 61 to 74 explain in detail about scheduling and related code in xv6)
- The xv6 OS book is here
    - https://pdos.csail.mit.edu/6.828/2017/xv6/xv6-rev10.pdf
- The *current xv6 scheduler* loops through all the processes which are available to run (in the **RUNNABLE** state) and allocates the CPU to each one of them (schedules processes) one at a time.
  The file **proc.c** contains most of the logic for the scheduler in the xv6, and the associated header file, **proc.h** is also quite useful to examine. The scheduler function is called by the *mpmain* function in **main.c** as the last step of initialization. This function will never return. It loops forever to schedule the next available process for execution. If you are curious about how it works, read Chapter 5 of the xv6 book.

Part 1: Priority based scheduling

In this scheme, each process is associated with a "priority" value. The scheduler ensures the use of process priority in picking the next process to schedule. The idea is as follows: assign each running process a priority, which is a non-negative integer number (the higher the number, higher will be its priority). Look up this video to understand how to add a system call to add or change priority in xv6.

Each time the scheduler runs, the scheduler should run processes that have a high priority. If there are two or more processes that have the same high priority, the scheduler should loop over the processes with high priority and choose the first which is available. A low-priority process does NOT run as long as there are higher priority processes waiting to run. Look up this video to understand how to implement priority based scheduling in xv6.

No submission is required for this part of the lab. It's a self study assignment. However, it's essential to go through this process and understand how to modify the scheduler. You will need this knowledge for the next part of the lab.

Part 2: Wait2 System Call

Implement a new system call **`wait2`** similar to **`wait`**, but with more functionalities, in order to check the performances of xv6 scheduling algorithms. Specifically, it will have the following interface:

```
int wait2(int *wtime, int *rutime);
```

The call takes two arguments, pointers to variables that denote the amount of time the process spent waiting for the CPU and the time spent executing on the CPU. It waits for a child process to exit, fills waiting time and run time (both are in terms of ticks counted in the trap handler. Note that xv6 is configured to generate 100 ticks per second which corresponds to a tick every 10ms. You may however report just the tick count.) in **`wtime`** and and **`rutime`** buffer respectively for the process that is exiting and return its **pid**. Return **-1** if the calling process has no children.

The **waiting time** of a process is defined as the time spent by the process in the RUNNABLE state (ready to run and waiting for CPU) and **run time** is the time spent by the process spent on the CPU. Note that

The user programs provided (test1.c & test2.c) have a simple implementation to check the wait2() system call.

The **`wait2`** call will be useful to understand how a scheduling policy affects the times of every process.

Hints:

A. Logging of durations/timings will need tracking down all events where the state of the process changes between WAITING, RUNNABLE, RUNNING etc. and appropriate duration updates via variables in the PCB entry for the process..

B. An implementation of the **`wait`** system call exists in xv6 and can be the starting point for the **`wait2`** implementation – a copy of the code of wait is a good starter for the implementation of **`wait2`**.

C. You can change the proc struct to include **`wtime`** and **`rutime`**, and update these in the trap handler.

D. You can initialise **`wtime`** and **`rutime`** in the allocproc function in the proc.c file.

Note: It is important to keep in mind that the process table struct ptable is protected by a lock. You must acquire the lock before accessing this structure for reading or writing and must release the lock after you are done.

Sample Output:

In part2.txt

Measure the waiting and running times of the test cases provided with the default (built-in) scheduler and log these times. You will need to present these with the corresponding times for the new scheduler you are about to build.

Part 3: The lets-get-even policy!

Implement a new scheduler policy in xv6 for this part of the lab.

In the default xv6 scheduler implementation, each process runs for a time quanta and upon this expiring, a timer interrupt yields the CPU, and xv6 schedules another runnable process. This is the process if the process state has not changed already via another system call.

When a process yields the CPU on a timer interrupt, the **lets-get-even** policy does the following:

- If the process whose time quanta ran out, has an odd numbered PID, the scheduler picks the next process in turn to schedule. Hence, processes with odd PID will run for 1 quanta.
- However, if the process whose time quanta ran out has an even PID it will be scheduled for another quanta before being context switched out.

Length (duration) of a quanta is what xv6 uses and need not be set/managed. Make sure the timer interrupts pause a process with an odd PID twice as frequently as that with an even PID, i.e., timer interrupts should make an odd PID yield the CPU if it has just completed one quanta while in case of an even PID, it shouldn't be running continuously for more than 2 quantas.

You must maintain a per process variable to help the trap handler in deciding whether to make the process give up CPU or not.

The same user test programs (test1.c & test2.c) forks 2 processes, prints the same stats. The waiting time and running time should help you check if your implementation is correct or not.

Some helpful hints:

- Look up the scheduler function in **proc.c** to know how xv6 does the process scheduling.
- Look up the trap handler to understand how timer interrupts help in making the processes run concurrently.

Note: Processes can be scheduled off the CPU for various reasons other than timer interrupts, like a blocking operation, some system call, etc. In case the process is scheduled for any reason other than the timer interrupt, it should be scheduled again the next round since a partially completed quanta does NOT count.

Sample Output:

```
In part3.txt
```

Note: You may get different waiting and running time. For correct implementation, the waiting time of even pid process should be less than waiting time of odd pid process.

Once you have built your new scheduler, measure the waiting and running times of the test cases provided with the new scheduler. Create a table to compare measurements of the default with the new scheduler and submit along with the code. Store this in a file named `measurements.txt`.

Update:

- ***New test programs (test1.c and test2.c) are added***
- ***You need to run these for both of the parts to check your implementation***
- ***MakeFile is updated***
- ***Outputs are added in the part2.txt and part3.txt files***
- ***Some other details are in updates.txt***
- ***For the purpose of the lets-get-even policy, we will only consider time accounting on a timer interrupt.***
- ***time spent in cases of partial execution in a system call + switch to another process is not to be accounted for the policy.***

Submission Instructions

- All submissions are to be done on moodle only.
- Name your submission as <rollnumber>_lab6.tar.gz (e.g 190050096_lab6.tar.gz)
- The tar should contain the following files in the following directory structure:
  <rollnumber>_lab6/

    |_ _< all modified files in xv6 such as

        syscall.c, syscall.h, sysproc.c, user.h, usys.S, proc.c, trap.c, defs.h, proc.h, . . . >

    |_ _Makefile

    |_ _measurements.txt

  Please adhere to it strictly.
- Your modified code/added code should be well commented on and readable.
- `tar -czvf <rollnumber>_lab6.tar.gz <rollnumber>_lab6`

Deadline: Friday, 30th September 2022, 11:55 PM via moodle.

Department of Computer Science and Engineering
Indian Institute of Technology Bombay