OS-344 Assignment-2

Instructions

- **Assignment has to be done by a group of 4 members.**
- **Group members should ensure that one member submits the completed assignment within the deadline.**
- **Each group should submit a report, with relevant screenshots/ necessary data and findings related to the assignments.**
- **We expect a sincere and fair effort from your side. All submissions will be checked for plagiarism through a software and plagiarised submissions will be penalised heavily, irrespective of the source and copy.**
- **There will be a viva associated with assignment. Attendance of all group members is mandatory.**
- **Assignment code, report and viva all will be considered for grading.**
- **Early start is recommended, any extension to complete the assignment will not be given.**

The goal of this lab is to understand process management and scheduling in xv6.

Before you begin

- Download, install, and run the original xv6 OS code.
- After you install the original code, copy the files from the xv6 patch provided to you into the original xv6 code folder. This patch contains the files modified for this lab.
- For this lab, you will need to understand and modify following files: proc.c, proc.h, syscall.c, syscall.h, sysproc.c, user.h, and usys.S.

Below are some details on these files.

- ✘ user.h contains the system call definitions in xv6. You will need to add code here for your new system calls.
- ✘ usys.S contains a list of system calls exported by the kernel.
- ✘ syscall.h contains a mapping from system call name to system call number. You must add to these mappings for your new system calls.
- ✘ syscall.c contains helper functions to parse system call arguments, and pointers to the actual system call implementations.
- ✘ sysproc.c contains the implementations of process related system calls. You will add your system call code here.
- ✘ proc.h contains the struct proc structure. You may need to make changes to this structure to track any extra information about a process.

- ✘ proc.c contains the function scheduler which performs scheduling and context switching between processes.

- You had already learnt, how to add a new system call in xv6. Some system calls do not take any arguments and return just an integer value (e.g., uptime in sysproc.c). Some other system calls take in multiple arguments like strings and integers (e.g., open system call in sysfile.c), and return a simple integer value. Further, more complex system calls return a lot of information back to the user program in a user-defined structure. As an example of how to pass a structure of information across system calls, you can see the code of the **ls** userspace program and the **fstat** system call in xv6. The fstat system call fills in a structure struct stat with information about a file, and this structure is fetched via the system call and printed out by the ls program.
- Understand how scheduling and context switching works in xv6. xv6 uses a simple round-robin scheduling policy, as you can see in the scheduler function in proc.c.

Part A: You will implement the following new system calls in xv6.

1. You will implement system calls to get information about currently active processes, much like the ps and top commands in Linux. Implement the system call **getNumProc()**, to return the total number of active processes in the system (either in embryo, running, runnable, sleeping, or zombie states). Also implement the system call **getMaxPid()** that returns the maximum PID amongst the PIDs of all currently active (i.e., occupying a slot in the process table) processes in the system.
2. Implement the system call **getProcInfo(pid, &processInfo)**. This system call takes as arguments an integer PID and a pointer to a **structure processInfo**. This structure is used for passing information between user and kernel mode. We have already implemented this structure in the xv6 patch provided to you, within the file processInfo.h. You may want to include this structure in user.h, so that it is available to userspace programs. You may also want to include this header file in proc.c to fill in the fields suitably. You must write code to fill in the fields of this structure and print it out. The information about the process that must be returned includes the parent PID, the number of times the process was context switched in by the scheduler, and the process size in bytes. Note that while some of this information is already available as part of the **struct proc** of a process, you will have to add new fields to keep track of some other extra information. If a process with the specified PID is not present, this system call must return -1 as the error code.

3. In the next part, you will change the xv6 scheduler to take **approximate process burst/running time** into account. To that end, add new system calls to xv6 to set/get process burst time. When a process calls **set_burst_time(n)**, the burst time of the process should be set to the specified value (in seconds say from 1 to 20). The burst time can be any positive integer value, with higher values denoting more time to execute. Also, add a system call **get_burst_time()** to read back the burst time just set, in order to verify that it has worked. For now, you do not have to do anything with these burst times, except storing and retrieving them in the process structure.

For all system calls that do not have an explicit return value mentioned above (e.g., getProcInfo), you must return 0 on success and a negative value on failure. Also, add appropriate programs in OS image to test all the system calls.

Note: It is important to keep in mind that the process table structure **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. Please ensure good locking discipline to avoid subtle bugs in your code.

Part B: The current scheduler in xv6 is an unweighted round robin scheduler. In this exercise, you will modify the scheduler to take into account user-defined process burst time and implement a shortest job first scheduler. Please begin this part only after completing the previous part, where you would have added system calls to get/set burst times.

- Modify the xv6 scheduler to use this burst time in picking the next process to schedule. Interpretation of the burst time and using it to make scheduling decisions is a design problem that is entirely left to you. For example, you can use the burst time to do strict shortest job first scheduling and then you may implement hybrid of round robin and shortest job first algorithm mentioned later in the assignment (For bonus marks). The only requirement is that a process with shortest burst time should be completed first by the CPU. Your report must clearly describe the design and implementation of your scheduler, and any new kernel data structures you may have created for your implementation. Also describe the runtime complexity of your scheduling algorithm. For example, the current round robin algorithm has a complexity of O(1).
- Make sure you handle all corner cases correctly in your scheduler implementation. Also make sure your code is safe when run over multiple CPU cores by using locks when accessing the kernel data structures.
- Now, you will need to write test cases to test your scheduler. You must write a separate user-space test program **test_scheduler** that runs all your tests. Your test program must fork several processes (at-least 5), set different approximate burst times within the forked processes, and

show that the burst times cause the child processes to behave differently. You must come up with at least two test cases with different programs, where your scheduler causes a different execution behaviour as compared to the default round robin scheduler, and you must quantify and explain this difference in the execution time of processes. Note that your test cases must use both CPU-bound and I/O bound processes, to show that your scheduler works correctly even when processes block.

Here is a simple test case to give you an example of what is expected. You can create two identical child processes that execute a CPU-intensive workload, and show that one finishes execution much faster than the other when its burst time is decreased.

Further, when you implement a hybrid scheduler and set the time slice as process with lower burst times, you should show that it finishes execution in the appropriate sequence. You should come up with such test cases that showcase your scheduling algorithm both qualitatively and quantitatively.

Submission instructions

- For this lab, you will need to modify the following files: proc.c, proc.h, syscall.c, syscall.h, sysproc.c, user.h, and usys.S. You may also need to modify defs.h depending on your implementation.
- Place all the files you modified in a zip file, with the file name being your group number (say, G10.zip).
- report.pdf should contain a detailed description of your new scheduler: the scheduling policy, design of any new data structures, implementation details, and how you handle various corner cases.
- Further, you must describe your test cases in some detail, and the observations you made from them.
- A patch of your code. Your patch must include all modifications to the source code as well as to the Makefile. Do not forget to include your test program. We will patch your code onto our codebase and run your test program. Please make sure that the output of your test program is clean enough to understand the results of the tests.

***** For hybrid scheduling algorithm you can choose to read this paper. Easy explanation is given below:

I) Create a ready queue RQ where the processes get submitted.

II) Set up the processes in RQ in the increasing order of burst -time of each process.

III) Fix the time slice as execution time of the first process lying in the Ready Queue RQ

IV) DO steps 5 to 6 UNTIL ready queue RQ gets vacant.

V) Select the primary process in RQ and allocate CPU to it for unit time quantum.

VI) The process in ready queue RQ is to be removed IF running process' remaining burst time becomes zero. ELSE move the method which is executing to the termination of the Ready Queue RQ.

- Note that this is a generic algorithm and you may need to change the code according to the actual implementation in OS.

Example based on hybrid scheduling algorithm:

Four processes P1, P2, P3, and P4 are considered. The processes come to the Ready Queue at 0-time interval. Assume, the burst time for the processes P1, P2, P3, P4 are 10, 12, 8 and 5 respectively. The processes are set up in Ready Queue in increasing order of their respective burst time as P4, P3, P1 and P2. The quantum of time for scheduling is fixed as the burst time of the primary process in the Ready Queue as 5-time units. The processor is allocated to every process for a time period of 5-time units.

In the first cycle, the CPU is allocated to the processes and for the processes P4, P3, P1 and P2, the outstanding burst times are 0, 3, 5 and 7 respectively. The process P4 is eliminated from Ready Queue as its lasting burst time is zero. Since the processes are already in increasing order of their respective outstanding burst times, they will be allocated CPU in the same order. The time quantum remains always constant i.e. 5-time units. Hence, after the second cycle, the remaining burst time for remaining processes in ready queue i.e. P3, P1 and P2 are 0, 0 and 2 respectively. The processes P3, P1 and P2 have zero outstanding burst time so these processes are eliminated from Ready Queue. In the third cycle, the process P2 is only left in the Ready Queue. After the third cycle, the outstanding burst time of each process becomes zero and Ready Queue becomes empty.

This algorithm has some advantages over the basic shortest job first or round robin scheduling Firstly, it does not let the bigger processes to starve while always waiting for the smaller processes to complete. Second, it decreases the average waiting time and average turn-around time.

--End of Assignment-2--