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:

1- Add support for tracking the number of "tickets" in a process:

- adding a system call called settickets which sets the number of "tickets" for a process with the following prototype: int settickets(int number). By default, processes should have 10 tickets. You can assume that the maximum number of tickets per process is 100000. The number of tickets should be inherited by children created via fork.

2- Add a system call called getprocessesinfo with the following prototype

```
int getprocessesinfo(struct processes_info *p);
```

where struct processses_info is defined as:

```
struct processes_info {
  int num_processes;
  int pids[NPROC];
  int ticks[NPROC];  // ticks = number of times process has been scheduled
  int tickets[NPROC]; // tickets = number of tickets set by settickets()
};
```

The system call should fill in the struct pointed to by its arguments by:

- setting num_processes to the total number of non-UNUSED processes in the process table
- for each i from 0 to num_processes, setting:
  - pids[i] to the pid of ith non-UNUSED process in the process table;
  - ticks[i] to the number of times this proces has been scheduled since it was created (don't adjust for whether or not the process used its whole timeslice);
  - tickets[i] to the number of tickets assigned to each process.

3- Add support for a lottery scheduler to xv6, by:

- changing the scheduler in proc.c to use the number of tickets to randomly choose a process to run based on the number of tickets it is assigned;

Some Supplied Test Programs

We have supplied some testing programs to aid you in figuring out whether your implementation is correct. Note that these programs are not complete tests. For example, they do not test that you set the correct default ticket count, that ticket counts are inherited by child processes correctly, and they may miss combinations of ticket counts some implementation techniques have problems with.

processlist.c

We have supplied `processlist.c` which is a test program which runs getprocessinfo and outputs the information in the struct.

You can add this test program the same way you added a test program for the xv6intro assignment.

timewithtickets.c

We have supplied timewithtickets.c which is a test program which:

- takes as command line arguments an amount of time to run for followed by a list of numbers of tickets to assign to each subprocess. Each subprocess runs an infinite loop and is killed after running of the designated amount of time. By default, the process does nothing during the infinite loop, but you can change #undef USE_YIELD to #define USE_YIELD to have the process instead call the system call yield() repeatedly in the loop.
- outputs a table of the programs in order, with the number of tickets assigned to each and the number of ticks it ran, as reported by getprocessesinfo
- reports an error if getprocessesinfo
  - the num_processes field returned is negative or exceeds NPROC
  - indicated that a child process had the wrong number of tickets;
  - was missing a child process

lotterytest.c

If you get an error about __divdi3 not being defined when trying to use this, try running sudo apt install gcc-multilib and recompiling.

We have supplied an automated test program `lotterytest.c` which essentially runs several cases of timewithtickets.c and checks the number of ticks reported. It also has a couple of test cases to make sure that programs that are not runnable don't confuse your scheduler.

It has some rather complicated code that attempts to perform a statistical test. We've set the parameters of the test so it should almost never indicate your code is wrong because of "bad luck". However, this test might be sensitive enough to detect if you don't use unbiased randomness (particularly if you use a pseudorandom number generator with a small range). I'm hoping this code is

correct, but it's very tricky and not easy to test if it's too sensitive or not sensitive enough. See also the **note on bias** below.

- If you get errors about tests not running for a non-trivial number of ticks, your schedule might be scheduling processes that aren't part of the test or not scheduling any process when it should schedule one or your scheduler might not be tracking ticks correclty or your system may just be too slow (especially if it spends a lot of time running `cprintf`s you added). You can try to make the test wait longer by increasing the `#define` for `SLEEP_TIME` to see if it's just the system being too slow.

- When your scheduler appears to have the wrong ratio of ticks, lotterytest runs some additional tests to diagnose this (checking if the ticks are in the right ballpark even if they are definitely statistically wrong). This means that the total tests run count will vary.

- If you see a tick count of -99999, this usually means that `lotterytest` didn't find the process at all in the `getprocessesinfo` output. (There is usual a test failure before this tick count indicating this.)

Hints

Reading on xv6's scheduler

1. Read Chapter 5 of the xv6 book for documentation on xv6's existing scheduler.

2. You can review Chapter 3 of the book to add system call and utility functions like `argptr`.

Reading on Lottery Scheduling

1. For an alternate explanation to the lecture slides, see Chapter 9 of Arpaci-Dusseau.

Suggested order of operations

1. Implement `settickets`, but don't actually use ticket counts for anything.

2. Implement `getprocessesinfo`. Use the processlist.c or other programs (e.g. ones you write) based on it to verify that it works.

3. Add tracking of the number of ticks a process runs. Use the timewithtickets.c or other programs based on it to verify that it works. Note that with round-robin scheduling, if multiple processes are CPU-bound, then each process should run for approximately the same amount of time.

4. Implement the lottery scheduing algorithm. Use the timewithtickets.c to test it. Then use lotterytest.c to further test it.

Tracking the number of ticks a process has been running

1. `proc.h` contains xv6's process control block, to which you can add a member variable to track the number of ticks a process has used.

2. You may need to modify `fork` (in `proc.c`) or related functions to make sure the tick count variable is initialized correctly.

Adding settickets

1. You can use `argint` to retrieve the integer argument to your system call. (Making `sys_settickets` take an argument will not work.)

2.Like for tracking the number of ticks a process has been runing, you will need to edit the process control block in `proc.h`.

3.Follow the example of similar system calls in `sysproc.c`.

Adding getprocessesinfo

1.You can use the `argptr` to retrieve the pointer argument in your system call handler.

2.You should iterate through the process list `ptable`, skipping over UNUSED processes.

3.Look at the code for `kill` in `proc.c` for an example of how to search through the list of processes by `pid`.

4.Before and after accessing the process table (`ptable`), you should acquire `ptable.lock` and afterwards you should release it. You can see an example of this in `kill` in `proc.c`. This will keep you from running into problems if a process is removed while you are iterating through the process table.

Adding the lottery scheduling algorithm

1.You will need to add a psuedorandom number generator to the kernel. We've supplied a suitable version of the Park-Miller psuedo-random number generator (use the `next_random` function to get a random number between 1 and MAX_RANDOM) from Wikipedia's page on Lehmer random number generators. You may use psuedorandom number generator code from elsewhere, so long as you clearly cite where obtained the code from.

2.It is okay if your pseudorandom number generator uses a fixed seed. But if you don't want to do this xv6 has a function `cmostime` that reads the current time of day.

3.The logic in `schedule` implements a round-robin scheduling algorithm, by iterating through all processes, scheduling each one as it iterates though them. You most likely will modify it to instead, iterate through all processes (perhaps more than once) each time a new process needs to be scheduled to choose which one to run.

4.xv6 does not support floating point, so you will need to do your random selection without `float`s or `double`s.

5.`getprocessesinfo` provides you the information you need to test that your lottery scheduler behaves correctly. You will probably not use it in the implementation of the scheduler itself.

6.When there are no runnable processes, your schduler should release the process table lock to give interrupts (like for keypresses) a chance to make programs runnable, then reacquire that lock and iterate through the process table again.

Identifying panics

1.If xv6 prints a message containing something like `panic: acquire`, this means that something called `panic("acquire");`. The `panic()` function stops the OS, printing out an error message. Generally, these panics are caused by assertions in the xv6 code, checking the current state is consistent.

Most xv6 panic messages include the name of the function that called panic, so you can often search for that function name and see when it called panic.

In general, you can get grep the xv6 code to find out what exactly the cause was.

For example, panic("acquire"); appears in acquire() in spinlock.c. It is called if a thread tries to acquire a spinlock that the current thread already holds.

2. There is a panic in the trap function that triggers when an unexpected trap occurs in kernel mode. You can infer more about those from the table of of trap numbers in traps.h and the message which is printed out before the panic.

One of the possible traps is a page fault, which means you accessed a memory location that was invalid. This can be caused by using an invalid pointer, going out of bounds of an array, or using more space on the stack than is allocated.

Note on random bias

1. A common source of bias is off-by-one errors in using ticket counts. Looking at what process is chosen with small ticket counts (e.g. 1, 2, 3) is helpful for this.

2. If you take a random number x in the range, say, 0 to 1023, and use it to choose a random number from 0 to 99 using x % 100, then you will choose numbers between 0 and 23 too often. One way to avoid this is to check if x is greater than 999, and, if so, select another random number x between 0 and 1023 (and keep doing this until you get one between 0 and 999 inclusive).

3. If you are experiencing large deviation from the expected ratios, the cause is probably something else, like double-counting a process's tickets when deciding how to map a random number to a process to run.

Part B: 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 C: 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.

Submission instructions

- Have the code ready to be verified during evaluation.

- Create a patch of your modified files.

- The report.pdf should contain a detailed description of all of your implementation including screenshots of code and output. This is to be uploaded the day before verification within the deadline.

- Further, you must describe your test cases in some detail, and the observations you made from them.

--End of Assignment-2--