Lecture Notes on Operating Systems

Lab: Processes and Scheduling in xv6

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. You can use your regular desktop/laptop to run xv6; it runs on an x86 emulator called QEMU that emulates x86 hardware on your local machine. In the xv6 folder, you will need to run `make`, followed by `make qemu`.
- After you install the original code, copy the files from the xv6 patch provided to you ito 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.
- Learn how to add a new system call in xv6. You can follow the implementation of an existing system call to understand how to add a new one. 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.

- Learn how to write your own user programs in xv6. For example, if you add a new system call, you may want to write a simple C program that calls the new function. There are several user programs as part of the xv6 source code, from which you can get an idea. We have also provided a simple test program `testcase.c` as part of our patch. This test program is compiled by our patched `Makefile` and you can run it on the xv6 shell by typing `testcase` at the command prompt, just like any in-built command. You must be able to write other such test programs to test your code. Note that the xv6 OS itself does not have any text editor or compiler support, so you must write and compile the code in your host machine, and then run the executable in the xv6 QEMU emulator.
- 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: New system calls in xv6

You will implement the following new system calls in xv6.

1. Implement a system call, called `hello()`, which prints Hello to the console. You can use cprintf for printing in kernel mode.
2. Implement a system call, called `helloYou(`*name*`)`, which prints a string *name* to the console. You can use cprintf for printing in kernel mode.
3. Next, we will implement system calls to get information about currently active processes, much like the `ps` and `top` commands in Linux do. 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.
4. 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.
5. In the next part, you will change the xv6 scheduler to take process priorities into account. To that end, add new system calls to xv6 to set/get process priorities. When a process calls `setprio(n)`, the priority of the process should be set to the specified value. The priority can be any positive integer value, with higher values denoting more priority. Also, add a system call `getprio()` to

read back the priority set, in order to verify that it has worked. For now, you do not have to do anything with these priorities, 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.

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: Implementing weighted scheduling algorithm in xv6

The current scheduler in xv6 is an unweighted round robin scheduler. In this lab, you will modify the scheduler to take into account user-defined process priorities and implement a weighted scheduler. Please begin this part only after completing the previous part, where you would have added system calls to get/set priorities. Modify the xv6 scheduler to use this priority in picking the next process to schedule, by using the priorities as weights to implement a weighted round robin scheduler. We would like you to achieve two things: (a) a higher numerical priority should translate into more CPU time for the process, so that higher priority processes finish faster than lower priority ones, and (b) lower priority processes should not be excessively starved, and should get some CPU time even in the presence of higher priority processes.

Make sure you handle all corner cases correctly in your scheduler implementation. For example, what is the default priority of a process before its priority is set? Also make sure your code is safe when run over multiple CPU cores by using locks when accessing the kernel data structures. You must think about how you will test the correctness of your scheduler. Come up with testcases that showcase your new scheduling policy.

We have provided you with several testcases and the expected output when you run these test cases. You can replace the simple `testcase.c` with each of these testcases, compile xv6, invoke the `testcase` command from the shell, and observe the output. You can compare with the expected output to verify correctness. You can also add all these testcases to the `Makefile` to run them all at once. It is important to note that the output from some test cases is not deterministic. For example, you will get different answers for the number of context switches of a process in different runs of the testcase. Similarly, the scheduler-related testcases also may not produce the same output given to you always. So, please do not panic if your output differs from the expected output in minor ways. Also, the test cases provided by us are not exhaustive in any way, and you are encouraged to write your own testcases, beyond those provided, to test your code.

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 directory, with the directory name being your roll number (say, 12345678).
- You need not submit any test cases you may have run. We will use our own `Makefile` with our testcases during testing.

- Tar and gzip the directory using the command tar -zcvf 12345678.tar.gz 12345678 to produce a single compressed file of your submission directory. Submit this tar gzipped file on Moodle.

Grading

We will run several testcases (beyond those provided) to test the correctness of your code. We will also read your code to ensure that you have adhered to the problem specification in your implementation.