Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Lab: Processes management xv6

The goal of this lab is to understand process management 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, run `make`, followed by `make qemu` or `make-qemu-nox`, to boot xv6 and open a shell.
- We have modified some xv6 files for this lab, and these patched files are provided as part of this lab's code. Before you begin the lab, copy the patched files into the main xv6 code directory.
- For this lab, you will need to understand and modify following files: `proc.c`, `proc.h`, `syscall.c`, `syscall.h`, `sysproc.c`, `defs.h`, `user.h`, and `usys.S`. Below are some details on these files.
  - `user.h` contains the system call definitions in xv6.
  - `usys.S` contains a list of system calls exported by the kernel, and the corresponding invocation of the trap instruction.
  - `syscall.h` contains a mapping from system call name to system call number. Every system call must have a number assigned here.
  - `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.
  - `defs.h` is a header file with function definitions in the kernel.
  - `proc.h` contains the `struct proc` structure.
  - `proc.c` contains implementations of various process related system calls, and the scheduler function. This file also contains the definition of `ptable`, so any code that must traverse the process list in xv6 must be written here.
- 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 system call. There are several user programs as part of the xv6 source code, from which you can learn. We have also provided a simple test program `testcase.c` as part of our patched code. 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. We have also provided several test programs to test the xv6 code you write in this lab. Feel free to test your code with these, as well as with other test cases you write. Remember that any test program you write should be included in the Makefile for it to be compiled and executed from the xv6 shell. 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.

Part A: New system calls in xv6

You will implement the following new system calls in xv6.

- Implement a system call, called `hello()`, which prints Hello to the console. You can use cprintf for printing in kernel mode. You can use `testcase-hello.c` provided to you to test your implementation.

- Implement a system call, called `helloYou(`*name*`)`, which prints a string *name* to the console. You can use cprintf for printing in kernel mode. You can use `testcase-helloyou.c` provided to you to test your implementation.

- Next, we will implement system calls to get information about currently active processes in the system. 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 processes in the system.

  Next, 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 defined this structure in the xv6 patched files folder provided to you, in 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. 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. Note that the parent PID of the init process (PID=1) is set to 0 by convention.

With all of these system calls put together, it is possible to iterate over all active processes in the system, and print their information to screen, just like the `ps` command does in Linux. The test program `testcase-procinfo.c` does this for you.

For all system calls that do not have an explicit return value mentioned above, 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 bugs in your code.

Part B: Weighted round robin scheduler

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.

First, 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 the system call `getprio()` to read back the priority set, in order to verify that it has worked. You may assume that a maximum value of priority (say, 1000) to simplify your implementation. Use `testcase-prio.c` to test your implementation of setting priorities.

Next, 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, you may have to set a default priority for new processes. 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 test-cases that showcase your new scheduling policy. We have provided one sample test case for you in `testcase-sched.c`, in which a parent spawns multiple children with increasing priorities, and makes them perform a CPU-intensive task. If your scheduler is working correctly, you will see that the higher priority processes will complete the task before the lower priority ones, because they will get more time to run from the weighted round robin scheduler.

welcome done should execute later the trapret and till then hold the tf->eip 's actual value somewhere. Also the welcome function runs in user space so beware of that.

Part C: A “welcoming” fork

In this part, we will implement a simple variant of the fork system call, to help you understand how processes return from trap. The default behavior of the fork system call is that a forked child starts execution from the same point in the code that the parent returns to after the fork system call. In this part, you will modify the behavior of fork to enable a child process to resume execution, first in a “welcome” function set by the parent, and then return to the code after fork. This functionality will be implemented via the following two new system calls.

- A parent process that wishes for a child to begin execution in a different function should invoke the system call `welcomeFunction`. This system call takes the address of the welcome function

as an argument. If this welcome function is not set, child processes should begin execution right after fork, as usual. If a welcome function address is set using this system call, new processes should begin execution in this welcome function instead.

- A child that begins execution in a welcome function set by the parent must invoke the system call `welcomeDone` to go back to executing code after the fork system call, like regular child processes do. This system call takes no arguments, and will be invoked by the child at the end of its execution of the welcome function.

The implementation of this functionality will require you to modify the trap frame of the child process suitably, to alter the EIP to point to the welcome function initially, and to restore it back to its original value when the child completes the welcome function execution. We have provided a simple test program `testcase-fork-welcome.c` that can be used to test your implementation. In the test program, a parent spawns two children, one before it sets the welcome function pointer, and one after. The first child will return to the code after fork as usual, while the second child will first run the welcome function and then resume execution in the code after fork. You can come up with other such test programs to test your implementation.

Submission instructions

- For this lab, you will need to modify the following files: `proc.c`, `proc.h`, `defs.h`, `syscall.c`, `syscall.h`, `sysproc.c`, `user.h`, and `usys.S`. You may also write new test cases, and modify the `Makefile` to compile additional test cases.
- Place all the files you modified in a directory, with the directory name being your roll number (say, 12345678).
- 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.