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 xv6 OS code given to you. 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.
- 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 `mytest.c` as part of the custom xv6 tarball for this course. This test program is compiled by our modified `Makefile` and you can run it on the xv6 shell by typing `mytest` 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: Adding new system calls in xv6

You will implement the following new system calls in xv6.

1. Implement the system call `void hello()`, which prints the string "Hello" to the console. You can use the testcase file `hello.c` provided to you to test your implementation.

2. Implement the system call `void helloYou(`*`name`*`)`, which prints a string *name* to the console. You can use `helloyou.c` provided to you to test your implementation.

3. Implement the system call `void getChildren()`, which prints PIDs of all the children of a calling process, and also prints the total number of children. A sample execution of the testcase is shown below. Note that a few parts of the output like PIDs may be different in different executions.

```
$ getchildren
Children PID's are:
4
5
6
No. of Children: 3
```

4. Implement the system call `void getSibling()`, which prints PIDs of all the siblings (i.e., other processes which have the same parent) of the calling process, and also prints the total number of siblings. A sample execution is below.

```
$ getsibling
Sibling PID's are:
8
9
10
No. of Siblings: 3
```

5. Implement the system call `void pstree()`, which prints the pstree (process tree) starting from the calling process, i.e., print the PIDs of all descendents of the current process, along with any identifier of the process (say, a name). Further, the print output should be indented differently for the children in different generations, and the number of leading whitespaces in the output should be equal to the depth of the process in the tree. To simplify your code, you can assume that the depth of the family tree is at most 2, i.e., you will only need to worry about printing the children and grandchildren. So, you can use two for-loops to find all descendents, instead of performing a depth-first-search. Once you finish this implementation, you can try to implement the more general case of printing children of arbitrary depth in the tree. A sample execution is shown below.

```
$ pstree
12 [pstree]
 13 [pstree]
  15 [pstree]
 14 [pstree]
```

Note: You can use cprintf for printing in kernel mode. Also, 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: Implementing ps

In this part, your goal is to get details about the various fields of a process in xv6. The system calls you must implement are as follows:

1. `int is_proc_valid(int pid)` takes pid as an argument, and returns 1 if the process's state belongs to the set {`SLEEPING, RUNNING, RUNNABLE`}, and 0 otherwise. So, basically this function tells whether the pid is in use by a valid process or not.

2. `int get_proc_state(int pid, char *buf, int size)` takes pid, (char*) buffer and the size of the buffer as arguments. It returns 1 if the process's pid exists (corresponding process found in the ptable), and fills the buffer with the state of the process, else it returns 0. For example, if the process's state is `SLEEPING`, it fills "SLEEPING" into the buffer. Check `proc.c` for the possible states of a process.

3. `int fill_proc_name(int pid, char* name)` takes pid and (char*) name as arguments. This syscall is used to give a name to a process with the given pid. It returns 1 if the process's pid exists, and fills the name into a new field in `struct proc`. It returns 0 if the process's pid does not exist.

4. `int get_proc_name(int pid, char* buf, int size)` takes pid, (char*) buffer and the size of the buffer as arguments. It returns 1 if the process's pid exists, and fills the name of the process (which is stored in the new field you added) into this buffer, else it returns 0. You can assume that the previous syscall `fill_proc_name` has been called before this syscall.

5. `int get_num_syscall(int pid)` takes a pid as an argument. It returns the number of system calls the function has received so far. You must add a field in `struct proc` to keep track of this. You need to increment this counter every time a syscall is made by a process (this includes all the syscalls included in syscall.h).

6. `int get_num_timer_interrupts(int pid)` takes a pid as an argument. It returns the number of timer interrupts the corresponding process has received so far. Again, you must add a fiield in `struct proc` for this. You need to increment this counter every time the process calls `yield()` because of a timer interrupt. You can check `trap.c` to understand how timer interrupts are handled.

Hint: Check `string.c` for the function `safestrcpy` for copying strings. Also, assume that all strings involved in the testcases are less than 16 characters long.

The output of the various testcases given to you should look like this (approximately) after you correctly implement all system calls.

```
$ test_valid
_______TESTING is_proc_valid(pid)_______
is_proc_valid(1): 1
is_proc_valid(2): 1
is_proc_valid(3): 1
is_proc_valid(10000): 0

$ test_state
_______TESTING get proc state(pid)_______
Process with pid (1) has state: SLEEPING
Process with pid (2) has state: SLEEPING
Process with pid (3) has state: RUNNING
Process not found

$ test_proc_name
_______TESTING fill_proc_name(pid) and get_proc_name(pid)_______
fill_proc_name(5): hello world! (Status: 1)
fill_proc_name(10000): hello world! (Status: 0)
Process with pid (5) has name: hello world!
Process with pid (10000) was not found

$ test_num_s
_______TESTING get_num_syscall(pid)_______
get_num_syscall(6): 4
get_num_syscall(6) [after sleep syscall]: 6

$ test_num_t
_______TESTING get_num_timer_interrupts(pid)_______
get_num_timer_interrupts(7): 1
get_num_timer_interrupts(7) [after long for loop]: 58
```

You might have guessed where we are heading. We have given you a template for a file `ps.c`, which is a simple program that will use the syscalls you have implemented above to print the details of all processes in the system, much like the `ps` command in Linux. The command should work as follows.

- Without any flag: When "ps" is run on the qemu terminal, it should print the details of the current process (which is "ps" itself). The details should include the pid, name of the process (which you need to set to "ps" through one of the syscalls mentioned above),state, number of system calls made by the process and number of timer interrupts received by the process.

  ```
  $ ps
  PID     NAME    STATE   SYS     INT
  3       ps      RUNNING 63      1
  ```

  To get the above output, make sure you separate the columns with a tab ($\backslash t$).

- With -e flag: When "ps -e" is run on the qemu terminal, it should print the details of all the valid processes (now you already know how to check validity of a process). The details should include the pid, state, number of system calls made by the process and number of timer interrupts received by the process (you do not need to print the name of the process in this case).

  ```
  $ ps -e
  PID     STATE           SYS     INT
  1       SLEEPING        26      1
  2       SLEEPING        19      1
  4       RUNNING         69      1
  ```

  As mentioned before, make sure you separate the columns with a tab ($\backslash t$). Also, you can assume that the maximum pid (of any process) is 64.

Part C: 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 the given test case 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 testcases that showcase your new scheduling policy. We have provided one sample test case for you, 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.

Part D: 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 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.

Part E: Implementing signals in xv6

The goal of this question is to implement signals and signal handlers inside xv6. To simplify, we shall assume that there is only one type of signal that can be sent or handled. For this question, you will need to implement the following two system calls.

- `int signal(void* handler)` takes as argument a pointer to a signal handler function, which must be executed when the process receives the signal.
- `int sigsend(int pid)` takes as argument the PID of the process to which the signal is to be sent. This signal must be handled by the target (receiving) process just before it returns from a trap the next time. You may assume that each process can track at most one signal at a time, so if a new signal arrives while there is already a pending signal, then the newer signal is discarded.

Note that the trap may be the result of an interrupt or syscall or any such event. You will simply need to invoke the signal handler before returning from a trap. (You can run this code on 1 CPU if you do not want to worry about parallel process execution.)

Hint: You may create new fields inside `struct proc` in `proc.h` to keep track of the signal handler function pointer (note that the value of the address of the handler function may be 0), received signals and so on.

There are two different ways in which the signal handler can be invoked by the process that receives the signal. When returning from trap, the process can either execute the signal handler in the kernel space itself, or after it returns to userspace. These two different implementations will be explored in the two parts of the question below. You are given separate test cases for the two parts below, and you must write your solution code separately for the two parts.

1. **Kernel Mode Execution**: In this part, you will execute the signal handler function directly in the kernel space just before returning from trap into userspace. Of course, if the process has not received a signal or has not registered a signal handler, then it should return from trap as usual. Remember to reset the signal status once you have handled it.

   Hint: If you have a `void func() {// does something;}` and you want to execute it given a `void* func_ptr;`, then you can invoke the function as follows:
   `((void (*)(void))func_ptr)();`

2. **User Mode Function Execution**: Executing user-defined functions in kernel mode is never a good idea for obvious reasons. However it is not so straightforward to implement userspace signal handler function execution. You will need to change your privilege level to user level, execute the signal handler function, then trap back to kernel mode, restore the old trapframe and go back to the code that was being executed before the trap. Note that it is important to keep track of the original trap frame before returning from the kernel space to execute the signal handler in user space. This is because, once you trap into the kernel space after finishing execution of the signal handler, the trap frame will be overwritten.

   To simplify the signal handler, let us assume that there is a system call, called `sigreturn()` that must be called at the end of the signal handler function. That is, it is the user's responsibility to place this system call at the end of their signal handling function. This syscall will allow us to return to kernel mode at the end of the execution of the signal handling function. So, signal handlers in this part of the question would look something like this:

```
void signal_handler(){
---signal handler code---
sigreturn();
}
```

With this new system call, we get the opportunity to restore the old trapframe inside the kernel during `sigreturn()`, and return the process to the code that was being executed before the trap, thereby enabling us to execute the signal handler function in user mode.

Hint: Instead of directly calling the signal handling function, save the trapframe of the process somewhere. Perhaps you could define a `struct trapframe* old_tf` inside of `struct proc`, and allocate memory for it in `allocproc` using
`p->old_tf = (struct trapframe*)kalloc();`
Then use `memmove(char *dest, char *src, int nbytes)` to copy the trapframe. Then, set the eip of the trapframe to signal handler function (you may need to typecast). This old trapframe can then be restored during the `sigreturn()` syscall.

We have given you three test cases each for the two parts. If you are making your own test cases, then note that you should not be using any system calls inside the signal handler function when executing the handler in kernel mode, as they will not work.

A sample execution of the test cases for the first part (kernel mode execution) is as shown below.

```
$ test-simple
x: 1
$ test-fork
X: 0
x: 1
$ test-syscall
x: 1
```

A sample execution of the test cases for the second part (user mode execution) is shown below. The PIDs printed may differ across executions. You can assume that `esp` always points to the top of the working stack.

```
$ test-fork2
X: 0
Hey, I exist!
x: 1
$ test-siblings
Hey, I exist!
I am 6, I have a misbehaving child 7
I am 7. I am now exitting
$ test-sigreturn
x: 1
val: 3
```