Lab 1: Fun with processes

Handed out Monday, Oct. 9, 2017
Design document due Monday, October 23, 2017
Due Friday Nov. 3, 2017

Introduction

There are two parts to this assignment: (1) Design, add and use a new system call to xv6, and (2) Change the scheduler to implement priority scheduling and priority inheritence.

Part 1: Add a new system call

In this part of the assignment, you need to understand the system call implementation and add a new system call. Repeating from the optional lab:

Open two terminal windows. In one, enter `make qemu-gdb` (or `make qemu-nox-gdb`). This starts up QEMU, but QEMU stops just before the processor executes the first instruction and waits for a debugging connection from GDB. In the second terminal, from the same directory you ran `make`, run `gdb`. (Briefly, `gdb -q -iex "set auto-load safe-path /home/csprofs/nael/xv6-master/"` . Change the last part to your path to the xv6 directory. You should see something like this,

```
sledge% gdb
GNU gdb (GDB) 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
+ target remote localhost:26000
The target architecture is assumed to be i8086
[f000:fff0] 0xffff0:ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb)
```

Now set a breakpoint on exec() by typing `break exec` in the gdb window type `continue` You should see something like:

```
(gdb) cont
Continuing.
[New Thread 2]
[Switching to Thread 2]
The target architecture is assumed to be i386
=> 0x80100af8 :push   %ebp

Breakpoint 1, exec (path=0x1c "/init", argv=0x8dfffe98) at exec.c:12
12{
(gdb)
```

Here we stop execution after the OS is initialized at the stage where it is starting the first process (init). If you type continue again, you will break again as follows:

```
gdb) cont
Continuing.
[Switching to Thread 1]
```

```
=> 0x80100af8 :push   %ebp

Breakpoint 1, exec (path=0x8c3 "sh", argv=0x8dffee98) at exec.c:12
12{
```

As you can see, at this stage, init started a shell process which is the xv6 shell we get when the OS boots. If you continue again, gdb will not return since it is waiting for a command to be started in the shell. Switch to the other window and try typing a command (for example, `cat README`) at which time you will get another break as the shell forks then execs the cat program. Feel free to look around at the program when it breaks to see how we reach the system call which should give you ideas about how to add one.

Assignment for this part

First, extend the current xv6 process implementation to maintain an exit status. To get this done, add a field to the process structure (see proc.h) in order to save an exit status of the terminated process. Next, you have to change all system calls affected by this change (e.g., exit, wait etc.).

a) Change the exit system call signature to void exit(int status). The exit system call must act as previously defined (i.e., terminate the current process) but it must also store the exit status of the terminated process in the corresponding structure. In order to make the changes in exit system call you must update the following files: user.h, defs.h, sysproc.c, proc.c and all the user space programs that uses exit system call. Note, from now on, all user space programs must supply an exit status when terminated.

Hassle: one hassle that this change (and the one in part b below) introduces is that all existing places that used exit(), including ones that are in test programs have now to be changed to use the new prototype. You can either do that yourself (e.g., use grep to find all locations of this call and change them), or create a new exit call to match the new prototype.

Goals of this part of the assignment: Get familiar with system call arguments and how arguments are passed given the presence of two stacks (user mode and kernel mode). Understand the backward compatibility hassles that come from modifying the system call prototype. Carry out a gentle modification to an existing system call and to the Process Control Block (PCB), which will be needed by the next part of the Lab.

b) Update the wait system call signature to int wait(int *status). The wait system call must prevent the current process from execution until any of its child processes is terminated (if any exists) and return the terminated child exit status through the status argument. The system call must return the process id of the child that was terminated or -1 if no child exists (or unexpected error occurred). Note that the wait system call can receive NULL as an argument. In this case the child' s exit status must be discarded.

Goal of this part of the assignment: Continue to get familiar with system call arguments, in this case with how to return a value.

c) Add a waitpid system call: int waitpid(int pid, int *status, int options). This system call must act like wait system call with the following additional properties: The system call must wait for a process (not necessary a child process) with a pid that equals to one provided by the pid argument. The return value must be the process id of the process that was

terminated or -1 if this process does not exist or if an unexpected error occurred. We are required only to implement a nonblocking waitpid where the kernel prevents the current process from execution until a process with the given pid terminates.

Write an example program to illustrate that your waitpid works. You have to modify the makefile to add your example program so that it can be executed from inside the shell once xv6 boots.

Goals of this part of the assignment: (1) Add a new system call; (2) Think about a non-trivial implementation of an operation that requires modifications to the PCB and manipulation of the process queues.

Part 2: Implement priority scheduling

In this part of the assignment, you will change the scheduler from a simple round-robin to a priority scheduler. Add a priority value to each process (lets say taking a range between 0 to 63). When scheduling from the ready list you will always schedule the highest priority thread/process first.

Add a system call to change the priority of a process. A process can change its priority at any time. If the priority becomes lower than any process on the ready list, you must switch to that process.

To get started, look at the file proc.c

Explain how you would implement priority donation/priority inheritence, or for 5% bonus implement it. (Bonus 1)

Add fields to track the scheduling performance of each process. These values should allow you to compute the turnaround time and wait time for each process. Add a system call to extract these values or alternatively print them out when the process exits. 5\% bonus (Bonus 2)

Goals of Part 2: Understand how the scheduler works. Understand how to implement a scheduling policy and characterize its impact on performance. Understand priority inversion and a possible solution for it.

Deliverables

There are two deliverables for this project, a walk-through/design document and the final program with two separate due dates outlined at the top of the assignment. The walk-through/design document is a text document answering the following questions:

- Explain how you would add your own hello world program so that you are able to execute it from the xv6 shell.
- Track the operation of the wait system call using the debugger. Explain what happens when the process has no children, and when it has a child process. You should be clear, showing all major steps in implementing the operation fo the system call.
- Explain how the current scheduler works, including interaction with the timer.