Lab 1: Fun with system calls

Handed out Friday Jan 11, 2020
Due Friday Jan. 31st, 2020

Introduction

In this assignment, you will design a new system call for xv6, and use it in some fork synchronization problems. The goals of this assignment are:

- Understand the system call interface
- Understand how user programs send parameters to the kernel, and receive values back
- See how the event handling in the Operating System works
- Understand the process structure and modify it

The deliverables for the assignment are the code that includes the required changes (the new system call, and modifications to exit), as well as the user test program(s). You will be graded by interview and expected to demonstrate that you understand what you did and why you did it that way.

Getting started

Follow the instructions on the class lab page to set up xv6 either on sledge or on your own machine using the vagrant option. Repeating from the optional Lab 0:

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)
```

From the qemu-nox-gdb terminal

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{
```

On the qemu-nox-gdb terminal, we can see "init: starting sh"

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

We actually don't need to maintain exit status in xv6 in reality. This assignment is just for our understanding.

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. We will need this for implementing wait. 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.

Change in: usys.S, syscall.h, syscall.c, sysproc.c, proc.c, proc.h, defs.h, user.h

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.

Change in: sysproc.c, proc.c, defs.h, user.h

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 blocking waitpid where the kernel prevents the current process from execution until a process with the given pid terminates. In other words, you do not need to worry about the options field for now.

Change in: usys.S, syscall.h, syscall.c, sysproc.c, proc.c, defs.h, user.h

d) 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.

e) (2% credit + Bonus 5%; you can get 98% the lab credit without implementing this part): Check out the waitpid option WNOHANG, for example as specified in this link. Implement WNOHANG and create a version of CELEBW02 example on the same page that checks of a child process is still running (it has to be simplified to work with xv6, for example, avoiding the use of time). You can also make assumptions on what is returned in status and implement only an exited status (i.e., enough to run something like the CELEBW02 example).

Add one else if block in proc.c and add argint() for options in sysproc.c

Help, hints, questions to explore, etc...

We will be extending this section based on how I see progress (for example, to help you understand the code base, or provide ideas to make you more effective in working with the code base).

- **Investing in learning gdb will really make your life easier this quarter!** There are only a few commands that you need to learn to get started.
  - `break` places a break point which causes the program to stop when it reaches that point. There are many ways to set a break point; I like to use `break function` to cause a stop at the function that I want to look at.
  - `continue` causes gdb to continue execution until the next break point. If you want to execute only one line of the code use `next` or `step`. The difference between the two is if the next line contains a function call, next will execute the full call before stopping while step steps into the function and stops at the first line there.
  - `print` allows you to examine a variable. You can use some of the C semantics here, for example to derference a pointer and see the memory location its point to (e.g., print *p), to print a specific format such as string or hex (e.g., print %x p). `display` is like print but it repeats the print every time gdb stops (e.g., after a break point or after a step or next)
  - `list` shows the code around the area you are.

    You can find more gdb help in the gdb resources linked on the class page under resources.

- Another really useful tool is `grep`. You can grep for a keyword (or if you are adventureous, a regular expression) in a file or group of files to find if/where it exists. So, lets say I am interested in a specific function or variable and want to see where it is used. In the xv6 directory, you can type: `grep wait *.c` to search for the string in all files ending with .c

  There are a couple of other useful options for grep. The `-i` option (e.g., `grep -i wait *.c` will make grep ignore case. The `-v` option excludes a pattern. So, lets say you want to search for wait, but not waitpid. One way you can do it is `grep wait *.c | grep -v waitpid`

- Background/deeper understanding: When any event in xv6 occurs, the hardware switches mode to the operating system, and jumps to the code in trapasm.S (starting at label alltraps). This code builds the trap frame (first segment), and then sets up the memory segments for the kernel (second segment--unimportant for now).

  The trap frame is a data structure built on the stack that is used to pass some important arguments to the trap handler including pointers to the user stack to enable getting arguments to system calls. Note that the user stack and the kernel stack for a process are separate. We discussed why in class. So, when we push arguments for a system call on the user stack, we need to give a pointer to the kernel to be able to access them.

You can see the trapframe structure in x86.h -- its ok to abstract it away since we will not be working with it in detail.

After filling parts of the trapframe, it calls trap() with the trap frame as argument, which takes us to trap in trap.c

Here you find a big switch statment based on the trapno that caused the trap (this is stored in the trap frame). Each case represents an event and implements its trap handler. Only a few events are currently supported such as system calls and the timer interrupt which are necessary for the barebones xv6 to run.

If you look under the system call case (the first case), you see some sanity checks (OS code is typically paranoid to avoid kernel panics), you see that we eventually call syscall() which is the top level handler for system calls. Lets follow syscall which is located at the bottom of syscall.c

We get the system call number from the eax register in the user code (saved on the trapframe) and use it to index into the system call table and pick up the appropriate handler to call. The return value from the system call handler is stored in the register eax in the trapframe (which is used by convention to store return values in Linux/x86) to provide the return value back to the user.

The handlers are all of the same function type (they are all called sys_xyz where xyz is the system call they handle) which enables us to use this trap handler table and call any one of them as appropriate.

Lets look at the implementation of one of these, lets say sys_kill() which passes a signal to a process, often to kill it.

sys_kill() and several other handlers are implemented in sysproc.c because they have to do with processes. Other handlers are implemented in the file system code, or memory code as appropriate to their operation. Use grep to find them if you can't figure out where they are.

You will note that sys_kill eventually calls kill where the real implementation of the system call is. This enables us to pass different parameters to each system call to get around the fact that the sys_ handler are all the same type.

Note also that sys_kill had to get those parameters fro the user stack. To help with this low level read, there are a number of accessor functions (argint, argstr, etc...) defined to get access to the arguments based by the user by using information in the trapframe.