CS 416 - Operating Systems Design

Homework 1
Adding Signals to xv6

September 27, 2016
Due October 14, 2016

1 Introduction

As you know from class (and the previous project), a signal, aka software interrupt/exception, is a notification to a process that an event has occurred. Just like hardware interrupts, signals interrupt the normal flow of execution of a program. A signal can be sent by different sources:

- The kernel
- Some process in the system
- A process sending a signal to itself

In this homework, we are going to focus on signals sent by the kernel to the process. Different types of events can cause the kernel to send a signal to a process:

- A hardware exception: For example executing malformed instructions, dividing by 0, or referencing inaccessible memory locations.
- Typing the terminal's special characters: For example the interrupt character (usually Ctrl-C) and the suspend character (usually Ctrl-Z).
- A software event: For example a timer going off or a child of the process terminating.

There are different options when handling a signal: the default action occurs (for example terminating the process upon receiving **SIGINT**), the signal is ignored, or a user specified signal handler is executed.

2 Signaling on xv6

You need to extend the xv6 kernel to add support for signals. For this part, you must modify the xv6 kernel to allow support for handling signals generated as a result of a hardware exception.

To simplify the implementation you should implement support for only one new signal at this time: **SIGFPE**[1], which should occur when the processor encounters a division by zero.

At a high level, the flow of a signal delivery should look like the following:

- An event that occurs that requires that a signal be delivered (Example: division by zero).
- An exception occurs, which causes the OS trap/exception handler to be invoked. (See `trap.c`)
- Code in the trap handler determines what type of exception has occured, and that a signal may need to be delivered.
- Code in the trap handler checks to see if the user has registered a signal handler for the type of event that has occurred.
- If a handler has been registered, call a function to set up a signal for the process.
- This function pushes the address of the excepting instruction, the volatile registers (**eax**, **ecx**, and **edx** on x86), as well as some other critical information onto the process's stack..
- This function also changes the instruction pointer **eip** to point to the signal handler function.
- Function returns, allowing the trap to return control to the process that caused it.
- Process resumes, with instruction pointer now moved to the signal handler, so the signal handler executes.
- Signal handler returns, but return address on the stack has been set to the address of a **signal trampoline**.
- Signal trampoline executes. This function is very simple. It consists of only a call to the `sigreturn()` system call.
- `sigreturn()` executes, calling an internal kernel function that cleans up the stack.
- This function recovers the saved volatile registers from the stack, as well as the saved excepting instruction address. It restores the volatile registers and sets the **eip** back to the excepting instruction, as well as moves the stack pointer to where it was before the exception occurred.
- When this function returns, everything should be as it was before the exception occurs. If nothing was changed in the handler, the excepting instruction restarts, and the exception occurs again.

[1] **SIGFPE** stands for "floating point exception", which is a historical misnomer, because it can also occur as a result of integer arithmetic.

3 Detailed Description

3.1 Summary

We will provide a considerable framework to help you get started on this new signal facility. In particular, you will need to implement code to do the following:

- Add a new case to the xv6 trap handler `trap()` to determine when a trap was caused by a division by zero, and call the `signal_deliver()` function.
- Implement the `signal_deliver()` function to construct a signal frame on the user stack and change the instruction pointer (**eip**) to point to the signal handler.
- Implement the `signal_return()` function to remove all traces of the signal frame from the user stack and change the instruction pointer (**eip**) back to the faulting instruction.

More details will follow in the subsequent sections.

3.2 Starting Point

To start on this project, use the following commands to check out a copy of xv6 with the skeleton code already in place:

```
git clone http://github.com/wkatsak/xv6
cd xv6
git checkout fa16-hw1-start
```

To compile xv6, type `make`. To start the compiled OS in QEMU, type `make qemu` (for a graphical terminal) or `make qemu-nox` if you are connecting via SSH.

3.3 Trap Handler

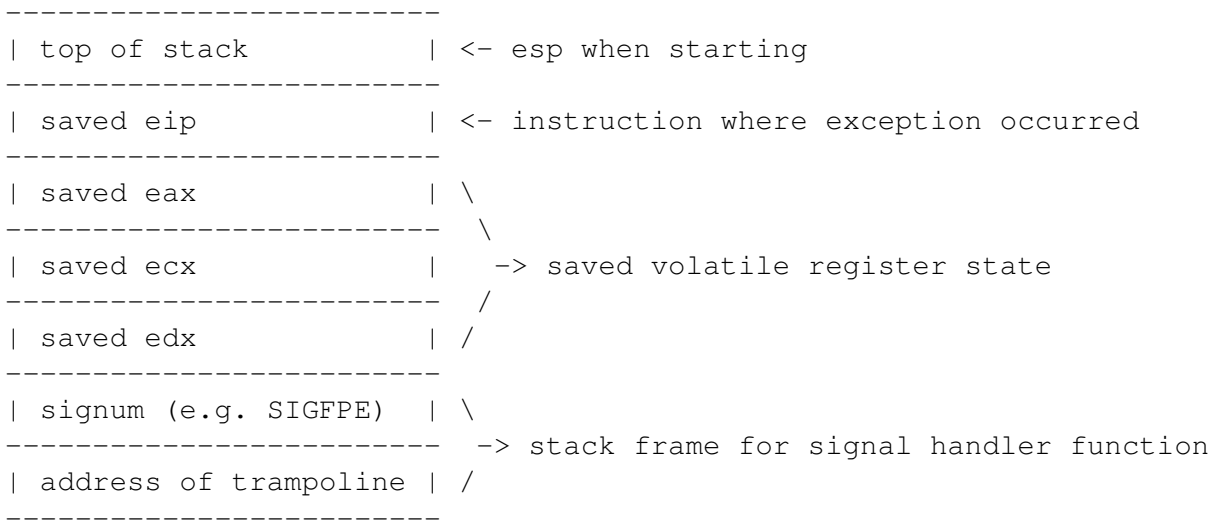
In xv6, the trap/exception handler is implemented in `trap.c` as `trap()`. This function includes a large `switch()` statement that switches between different cases (type of exceptions). You will add a case for `T_DIVIDE` that checks for a signal handler for `SIGFPE`, and if this exists, calls `signal_deliver()` to set up a signal. We have added a field to the `struct proc` structure (each process has one of these) that holds pointers to any signal handlers that the user has registered (See `proc.h`). The proc struct of the current process is always accessible in the kernel using the variable name `proc`. For example, you can check for a handler for `SIGFPE` by checking `proc->signal_handlers[SIGFPE]`.

3.4 Deliver Signal

We have provided a function stub called `signal_deliver()` in `proc.c`. This function must construct the signal frame on the process' call stack, change the instruction pointer to the signal handler, and return control to the process.

Whenever an exception occurs, all of the excepting process' state (registers) is saved into a structure called the **trapframe**. These registers are accessible via `proc->tf->register`, e.g. `proc->tf->eip`. Any change made in the kernel to these registers will be applied when the process resumes execution.

Knowing this, and using `proc->tf->esp` (stack pointer) as reference point, you should make the user stack look like the following:



Keep in mind, that the stack grows *down* in memory on x86, so the bottom of this diagram represents the top of the stack.

After you have put all of this into memory, you should update the **esp** (on the trapframe) to point to the address of the trampoline (the original **esp** - 24 bytes). You should also update the **eip** (also on the trapframe) to point to the address of the signal handler.

In our framework, you can find the address of the trampoline in `proc->signal_trampoline`.

3.5 Restore Stack After Signal

We have also provided a function stub called `signal_restore()` in `proc.c`. This is the function that executes when the system call `sigrestore()` is called. This function must essentially reverse what happened in `signal_deliver()`, restoring the volatile registers (to the trapframe), and setting the **eip** (also in the trapframe) to the excepting instruction. All of these register values must be recovered from where you placed them on the stack.

After this is all done, you must also change the **esp** on the trapframe back to its original location.

4 Source Control

Please ensure that you `git commit` often while you are working. We recommend that you commit each time you have made a complete, significant change, and use a descriptive commit message. Typically in systems development, one feature may be implemented over dozens of individual commits.

To submit, you will tar/gzip the entire repo, but source control is still useful to track your development.

5 Testing

5.1 Test Programs

The starting branch mentioned above will contain a test program called `signal_test`. This test should only pass if your signal handler successfully executes 100,000 times, the stack is correctly maintained, and the volatile registers are restored. We have also included a program called `signal_skel` which is a very

basic program that registers a handler and causes a SIGFPE. This might be useful as a starting point for debugging.

5.2 Executing in xv6

If you follow the setup instructions on the iLabs, everything should just work. Just type `make qemu` or `make qemu-nox` to build your modified code and you can run the test programs inside xv6.

6 Submission

To submit your work, please tar/gzip the entire git repo (xv6 directory) and submit this file on Sakai.

Only ONE student from each group should commit, but you NEED to make sure you give the netids of both partners in the submission notes and report.

7 Requirements

- The code has to be written in **C** language. You should discuss on Sakai if you think inline **asm** is necessary to accomplish something.
- Do not copy the solution from other students. Use Sakai to discuss ideas of how to implement it. Do not post your solution there. Use Sakai to submit it.
- Submit a report in PDF form on **Sakai** detailing what you accomplished for this project, including issues you encountered.