CS 416 - Operating Systems Design

Homework 1
Adding Signals to xv6

February 25, 2016
Due March 11th, 2016

1 Introduction

As you know from class, 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 either as a result of a hardware exception or a user-defined action. To simplify the implementation you should implement support for only two new signals at this time: **SIGFPE**[1], which should occur when the processor encounters an aritmetic error, e.g. a division by zero, and **SIGALRM**, which occurs when a user-defined timer expires. You can check the standard Linux man page to see the details of how each of these signals are supposed to work.

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

3 Project Stages

We will break down the project info several parts.

- **Stage 1** - Implement an incomplete signal facility that supports handling **SIGFPE** and **SIGALRM**, but does not properly handle volatile registers (see description for details).
- **Stage 2** - Implement a correct signal facility that properly saves/restores volatile registers.
- **Stage 3** - Use the new signal facility to measure the time cost of a exception/trap in xv6.

3.1 Starting Point

To start on this project, use the following commands in your xv6 git repository (assuming you have checked it out as instructed in the document uploaded to Sakai):

```
git fetch origin
git checkout sp16-hw1-start
```

4 Stage 1 - Basic (Incomplete) Signal Facility

To complete this stage, you will need to add support for **SIGFPE** and **SIGALRM** to xv6. Each process should be able to register its own signal handler for each of these signals. You need to add a new system call: *int register_signal_handler(int signum, sighandler_t handler)* to facilitate this registration. You also need to add a very simple function to the user library (ulib.c): *int signal(int signum, sighandler_t handler)*. At this time, this function should simply wrap another call to **register_signal_handler(..)**. The reason for this wrapper will become apparent in Stage 2.

You also need to add two symbolic constants (SIGFPE and SIGALRM) that are accessible to user programs, to be used as a signum argument to **signal(..)**. These constants should be added to **signal.h**.

You will also need to add another system call: *int alarm(int seconds)*. This system call will notify the kernel that the process requires an alarm after *seconds* has passed.

The **signal(..)** call takes the signal number (*signum*) and a pointer to the signal handler function (*handler*) as arguments. It returns the previous value of the signal handler on success or -1 on failure. The parameter *handler* is of type sighandler_t, which you will have to define as *typedef void (sighandler_t)(siginfo_t)*. Type siginfo_t should be defined as a struct, that includes at a minimum, a *signum* field to indicate the type of the signal that is being handled. Add these definitions to **signal.h**. When called by a process, the **signal(..)** system call should register *handler* as the signal handler for that process.

To make everything work, you need to extend the current xv6 exception/trap handling code to recognize when a signal is required, and if this is the case, modify the stack of the currently running program so that when execution resumes, the instruction pointer points to the start of the registered signal handler function. The *siginfo_t* argument to the signal handler should have the proper field populated with the *signum* that was caught.

Since you need to support two signals, you need to possibly set up a signal under two possible trap/exception scenarios:

- The cause of the exception is a division by zero.

- The cause of the exception is a timer interrupt **AND** an alarm is registered **AND** the required time has passed.

4.1 Source Control

We recommend that you create a separate branch in git to implement the each stage. This is because we would like to be able to evaluate your solution to each stage independently, and able you to make modifications without interfering with the solution(s) to the previous stages.

For this stage, execute the following in your repo directory (after checking out the start branch as specified above):

```
git branch hw1-stage1
git checkout hw1-stage1
```

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.

4.2 Testing

The starting branch mentioned above will contain two test programs called **stage1_sigfpe** and **stage1_sigalrm**, respectively, that will test your new signal facility. Your implementation must be able to pass the tests performed by this code. *These programs are commented out of the Makefile by default, so make sure that you uncomment them to add them to the compile process.*

5 Stage 2 - Dealing With Volatile Registers

While the signal handler implemented in Stage 1 works for simple handlers, it's not a correct implementation. For this part of the homework, you have to produce a complete and correct implementation. But first, you need to understand why what you did for Stage 1 is not complete.

When a function is called in C, there is a certain convention on register use. Certain registers (the so called the volatile ones) have no useful content across a function call, hence the callee can use them without any concern. This is because if they had useful content in the caller, it would be the responsibility of the caller to save their value somewhere (on the stack) before issuing the call. And there are other registers (the non volatile ones), which the callee must save before using because the assumption is that the caller can leave data in them when making a call.

Now, in your implementation of the signal handler, when an exception occurs, the kernel imitates a call of the signal handler by pushing on the stack the corresponding function address (as well as the parameter *siginfo_t*). For all purposes, the handler follows the calling assumptions we discussed above, namely it can use the volatile registers without any precaution. However, in this case, the caller does not exist as such, since a faulty instruction caused the signal handler to be invoked and not a function call (which the compiler could have identified in order to save the volatile registers). As a result, if in your implementation of the signal handler, the volatile registers are used (say by a function like printf), their useful content will be messed up.

To address this issue, you will have to add two things to your implementation. First, the kernel must be the one to push on the user stack the volatile registers (found on the trapframe) before building the call of the signal handler. Second, it must ensure that someone pops off these registers after the return of the handler

and before returning to the instruction that caused the exception. *The kernel itself cannot do this since the return from the handler does not involve the kernel.*

To solve this problem, you will have to add another function to be called just after the handler for the sole purpose of popping the volatile registers from the stack. The kernel will have to add the address of this function on the stack below the handler address.

You need to write this function and present its address to the kernel when calling the **register_signal_handler(..)** system call (along with the signal handler function). You should modify your **register_signal_handler(..)** to take this additional argument, but to avoid exposing the user to this additional complexity, you should implement this special function (called a wrapper or signal trampoline) inside the C library, and simply pass it to the kernel inside your userspace **signal(..)** library function (remember that we said it would become clear why we needed this level of indirection?). Note that this approximately 3-line function must be written in assembly to avoid the complications caused by the C procedure call conventions. You *must* implement this function inside ulib.c.

The following should help with some of the terminology. By convention, there are two types of registers. One type is callee-saved (non-volatile) and the other type is caller-saved (volatile). The caller-saved register are temporary registers and it's the resposibility of the caller to save them on the stack if it modifies them. The callee-saved registers should be preserved across all calls. The convention on x86 is that out of the 8 general purpose registers, **esp**, **ebp**, **ebx**, **esi**, and **edi** are callee-saved, while **eax**, **ecx**, and **edx** are caller-saved registers.

5.1 Source Control

For this stage, execute the following in your repo directory (with your hw1-stage1 branch checked out):

```
git branch hw1-stage2
git checkout hw1-stage2
```

Please note that all of the previous stage's changes must have been committed before you branch.

5.2 Testing

The starting branch mentioned above will contain a test program called **stage2** that will test your corrected signal facility. Your implementation must pass the test done by this program. *This program is commented out of the Makefile by default, so make sure that you uncomment it to add it to the compile process.*

6 Stage 3 - Measuring the Cost of an Exception/Trap

To complete this stage, you will use your newly implemented signal facility to measure the cost of an exception/trap on xv6.

You should realize by now that a division by zero causes a processor exception which vectors execution into the operating system, when then activates your new signal facility. You must leverage this fact to measure the cost of an exception/trap. Due to imprecision of timing, it is impossible to measure the cost of a *single* exception. You must instead measuring the time taken to handle a large number of exceptions, and infer the cost of a single exception by calculating the average time over this larger set of exceptions.

Build a program based inside the provided **stage3.c** file to perform this timing. Remember that after a signal handler executes, the instruction that caused the trap is restarted. You can leverage this fact to measure the trap time. Ordinarily, if the signal handler does not exit the program, the signal will occur over and over again. Use a **static variable** inside the signal handler to keep track of how many times the exception occurs, and when the exception has occured enough times to get an accurate timing, *modify the return address on the stack* to skip the instruction that causes the exception.

Use the **uptime()** system call to measure the timing (xv6 does not offer any other timing facility).

6.1 Source Control

For this stage, execute the following in your repo directory (with your hw1-stage2 checked out):

```
git branch hw1-stage3
git checkout hw1-stage3
```

Please note that all of the changes from the previous stages must have been committed before you branch.

6.2 Testing

The starting branch mentioned above will contain a skeleton test program called **stage3**. You must modify the **stage3** program as specified. Your program MUST output results in EXACTLY the following format:

```
Traps Performed: XXXX
Total Elapsed Time: XXXX ms
Average Time Per Trap: XXXXX ms
```

This skeleton program is commented out of the Makefile by default, so make sure that you uncomment it to add it to the compile process.

7 Submission

To submit your work, please follow the following instructions:
For each branch, type the following (for each netids field, use both partner's netids, hyphenated like **NETID1-NETID2**):

```
git checkout hw1-stageN
make submit netids=<NETIDS> name=stageN base=hw1-start
```

After this is complete, you should have three .tar.gz files (one for each stage), containing an each containing individual patch file(s) for each commit. Ensure that the tar files actually contain the patches, then submit these tarballs on Sakai.

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

8 Hints

This section is a list of hints for some of the more difficult portions of this part of the assignment. This does not cover every aspect of the code changes that must be made.

- The process structure should be extended (in **proc.h**) to include an array of pointers to store pointers to the user's signal handler functions. The entry in the array should be initialized to -1 (the default action) whenever a new process is created. The default action for a signal should be **SIGKILL**, which already exists in the xv6 kernel.
- You may also wish to extend the process structure to add support for a list of pending alarms.
- To register a custom signal handler for a process in the kernel, you must first read the arguments off of the calling process' stack to get the *handler* parameter. Then you will need to set the handler in the calling process' signal entry. Finally, you should return the return value back to the calling process.
- To deliver the signal to a process: If no signal handler has been set for the process, terminate it (This is the current default action in the code). If there is a custom signal handler set, you must place the address contained in the target process' eip at the top of PID's user stack (set target process: $esp - 4 = eip$). Then decrement its esp by 4 and change PID's eip to point to the address of PID's custom signal handler for *signum* (set target process': $eip =$ *custom signal handler address*). In this way, when the target process is scheduled next, it will return to execute the custom signal handler first. Returning from the custom signal handler, the process will continue with the instruction where the trap/interrupt occurred. A process' eip and esp can be found in its trap frame.

9 Requirements

- The code has to be written in **C** language. You should discuss on Piazza if you think inline **asm** is necessary to accomplish something.
- Do not copy the solution from other students. Use Piazza to discuss ideas of how to implement it. Do not post your solution there. Use Sakai to submit it.
- Create a new branch or tag in your local git for each step of your solution to make sure you can isolate each step later in case you need to refer back.
- Also keep a separate branch for each stage, so you can easily turn them in separately.
- Submit a report in PDF form on **Sakai** detailing what you accomplished for this project, including issues you encountered.