CS 416 - Operating Systems Design

Homework 02
Having fun with Interrupts, Exceptions and Traps

March 1, 2014
Parts [1, 2] due by March 9th, 2014
Parts [3, 4, 5] due by March 16th, 2014

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. There are a lot of details about signals, which you can look up online.

2 Part 1
What is the cost of an exception?

[25 points] In this part, you will write a program to measure the cost of a software interrupt. Since you do not have access to the kernel, what you can measure from the user space is the cost of a signal handler (which most likely will include the cost of exception handling in the kernel plus the preparation of the signal stack for the signal handler call and the cost of executing the handler itself).

To achieve this, you will have to set a handler for a signal (say for the divide by zero exception). If the handler does nothing to fix the cause of the exception, the process will return to the faulty instruction after the execution of the signal. This causes the same exception and another call to the handler. The running time of a single exception handler is too small to be measured accurately; therefore, you have to measure the time to execute a large number of exception handlings and calculate an average. To accomplish this, the handler should count the number of times it's called. After the handler is executed the desired number of times, the handler must force the process to skip the faulty C instruction (note that a C instruction is most likely translated into more than one assembly instruction). Skipping over the faulty instruction can be achieved by changing the value of the return address on the signal frame. The return address sets the value of the IP (Instruction Pointer) after the execution of the handler. Without modification the return address' value is the address of the faulty instruction. The return address is located at a certain offset from top of the stack, which can be determined by using the signal frame structure (found on some header file) or, empirically, by inspecting the handler stack during the handler's execution.

For this part send your **program source code** with a **Makefile**. Also include a short **report** about your results and the program's description.

3 Part 2
A system call on xv6

[25 points] A system call is an interface provided by the Operating System to user applications allowing them execute its services. It provides generality and isolation since all the hardware and implementation details are underneath the system call API layer. Another interesting characteristic of a system call is that user applications voluntarily calls them releasing the execution to the Operating System, which can decides to schedule other processes, terminate the user application, etc.

This part of the homework, you will have to implement a system call (**icount**) on xv6 operating system. In addition to that, you will implement an user application (**hw02**) that will use the system call previously implemented. The system call will return the number of **keyboard interruptions** since the Operating System starts its operation.

To implement the counter, you will need to see how interrupts, exceptions and traps are dealt in xv6, on file **traps.c**.

The user application (**hw02**) has to print on the screen the returned value from the **icount** system call. When you call your application again, the returned value has to be different from the previous one, since you had to type the application name again on the shell command prompt.

To create a new user space program in xv6, you first create the program file (**hw02.c**) with the program source code. You also need to add a line in the **Makefile**, so that your user space program is compiled with the rest (see lines starting right after "UPROGS=" in the **Makefile**). You can use kill.c, ls.c, echo.c, etc. as a template for your user space program. Make sure all programs exit by explicit calls to the exit() syscall, otherwise you will get an error on program exit.

As a **hint**, you should look at how the **uptime** syscall is implemented in xv6. It may help you with the **icount** system call, since there is a certainly similarity between them.

4 Part 3
Signaling on xv6

(Some parts of this section have been summarized from the Linux signal(7) man page)
[20 points] A signal is an event which can be sent by one process to another process or by the kernel to a process. Each signal has a current disposition, which determines how the process behaves when it is delivered the signal.

The following default actions are available:

- **Term** - terminate the process.
- **Ign** - ignore the signal.
- **Core** - terminate the process and create a core dump file.
- **Stop** - stop the process.
- **Cont** - continue the process if it is currently stopped.

There are a number of standard signals that are supported by most UNIX-like OSs, for example **SIGHUP**, **SIGINT**, **SIGQUIT**, **SIGKILL**, **SIGSEGV**, **SIGPIPE**, **SIGALRM**, **SIGTERM**, **SIGUSR1**, and **SIGUSR2**. You can take a look at the signals(7) man page for more details on these signals. There are others listed on the man page as well. Signals can be synchronous or asynchronous, queued or not. Conceptually signals are intuitive, but the full kernel implementation is quite complex.

A process can change the action of a signal using the **signal(..)** or preferably **sigaction(..)** system call. Using this system call, a process can elect one of the following behaviors to occur on delivery of the signal: perform the default action; ignore the signal; or catch the signal with a signal handler, a programmer-defined function that is automatically invoked when the signal is delivered.

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 trap. To simplify the implementation you should implement support for only one signal: **SIGSEGV**. Additionally, you must implement the support for a custom signal handler provided by the process.

Here are the steps you will need to complete for this part of the assignment:

1. Add support for the signal **SIGSEGV**. Each process should be able to register its own signal handler for this signal. Add a new system call: *int signal(signum, sighandler_t handler)*. The **signal(..)** call takes the signal number (signum) an a pointer to the signal handler (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)(void)* at appropriate places in the source code. When called by a process, the **signal(..)** system call should register "handler" as the signal handler for that process.
2. Create a user level test program called **signal_test** that creates a signal handler for each of the new signal. After registering the handler the signal is invoked. The handler for the signal should not terminate the program flow and after handling the signal should continue the program.

3. **SIGSEGV** stands for **SIG**nal **SEG**mentation **V**iolation, the signal has to be sent to a process when it makes an invalid virtual memory reference, or segmentation fault. In your **signal_test** application, try to access an address that is not part of the process's address space to raise this exception.

4.1 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, though.

- The process structure should be extended (in **proc.h**) to include an array of pointers to store pointers to customer signal handler functions. This should be large enough for 2 signals (0 and 1) and initialized to -1 (the default action) for each signal in (**proc.c**) whenever a new process is created. The default action for a signal should be **SIGKILL**, which already exists in the xv6 kernel.

- 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 pointed to by 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.

- To create a new user space program in xv6, you first create the program file (foo.c or whatever you wish to call it) with the program source code. You also need to add a line in the Makefile, so that your user space program is compiled with the rest (see lines starting right after "UPROGS=" in the **Makefile**). You can use kill.c, ls.c, echo.c, etc. as a template for your user space program. Make sure all programs exit by explicit calls to the **exit** syscall, otherwise you will get an error on program exit.

5 Part 4
What is the cost of an exception on xv6?

[20 points] In this part, you will write a program to measure the cost of a software interrupt. Consider that you do not have access to the kernel, what you can measure from the user space is the cost of a signal handler (which most likely will include the cost of exception handling in the kernel plus the preparation of the signal stack for the signal handler call and the cost of executing the handler itself).

To achieve this, you will have to set a handler for the signal that you implemented in Part 3. If the handler does nothing to fix the cause of the exception, the process will return to the faulty instruction after the execution of the signal. This causes the same exception and another call to the handler. The running time of a single exception handler is too small to be measured accurately; therefore, you have to measure the time to execute a large number of exception handlings and calculate an average. To accomplish this, the handler should count the number of times it's called. After the handler is executed the desired number of times, the handler must force the process to skip the faulty C instruction (note that a C instruction is most likely translated into more than one assembly instruction). Skipping over the faulty instruction can be achieved

by changing the value of the return address on the signal frame. The return address sets the value of the IP (Instruction Pointer) after the execution of the handler. Without modification the return address' value is the address of the faulty instruction. The return address is located at a certain offset from top of the stack, which can be determined by using the signal frame structure (found on some header file) or, empirically, by inspecting the handler stack during the handler's execution.

As xv6 doesn't have implemented a library function such as $gettimeofday(..)$, you can use **uptime** system call to estimate the time spent on the OS to handle the signal. One way to do that is to execute the signal handler several times and get the average cost. The granularity of **uptime** is 100ms, you need to estimate the number of times the exception needs to happen to get a meaningful result.

For this part send your **program source code** with a **Makefile**. Also include a short **report** about your results and the program's description.

6 Part 5
Compare the cost of a signal on Linux vs. xv6

[10 points] Compare the results of **Part 1** and **Part 4**.

7 Notes

- The code has to be written in **C** language, without using inline **asm**.
- You need to use an IA-32 architecture machine, Linux OS machine for Part 1.
- The stack frame of a procedure call is different of a signal frame.
- Global variables are initialized with $zero$.
- Submit a tarfile of the xv6 code with your modifications. (note: write your program in XV6 revision 6)
- 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.