Background on UNIX signals

(Some parts of this section have been summarized from the Linux signal(7) man page)

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 an 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.

Problem statement

Extend the xv6 kernel to add support for signals. For this homework assignment, 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 two signals: SIGFPE and SIGSEGV. Additionally, you must implement the support for custom signal handlers provided by the process.

Here are the steps you will need to complete for this assignment:
1. Add support for 2 signals, SIGFPE and SIGSEGV. Each process should be able to register its own signal handler for these signals. 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. (70 pts)

2. Create a user-level test program called `signal_test` that creates a different signal handler for each of the 2 signals. After registering the handler each signal is invoked. The handler for the first invoked signal should not terminate the program flow and after handling the signal should continue the program. The second signal handler should print an arbitrary message and then terminate the program. (20 pts)

**Hints**

This section is a list of hints for some of the more difficult portions 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 add a new system call, you must extend the syscall vector, in syscall.c, and add a new syscall number in syscall.h. Your code for the syscalls should be placed in sysproc.c and proc.c. You should look in syscall.c to see how arguments are passed on the stack, between user space and the kernel. Finally, you must create a stub for the new syscall in usys.S similar to those found in usys.S.

- 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 "signal" (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.

Submission Instructions

Readme.txt (10pts): This file should be strictly in plain text format. Should contain the following information in this order:

- Your Name

- Parts of the project you have done in the following format:
  
  For example:
  
  Part1: Done
  Part 2: Not Done [first signal implemented-Second signal not implemented]

- Describe the changes you have made to implement the syscalls and signal handling and why they were necessary. Please keep the explanation short and meaningful.

Code:

- Submit a tarfile of the xv6 code with your modifications. (note: write your program in XV6 revision 6)