Project 2: Signals

Consult the submit server for deadline date and time

You will implement the Signal() and Kill() system calls, preserving basic functions of pipe (to permit SIGPIPE) and fork (to permit SIGCHLD).

The primary goal of this assignment is to develop an understanding of the behavior of signal handlers and the interactions between signals and processes. This assignment also reinforces register state manipulation from the fork and exec assigment, adding changes to the user stack.

1 Signals

A signal is an inter-process communication mechanism by which one process causes another process (the target) to execute one of a small array of functions (the signal handlers). Each process has a table of signal handlers (function pointers) indexed by signal number. A process can use the "signal" system call to manipulate its table, thereby assigning a signal handler function for a signal number. A process can send a signal to another process by using the "kill" system call; the arguments are the pid of the target process and the signal number. The kernel will then arrange for the target process to execute the signal handler function for that signal number. This is your main task.

Signal handlers should be executed in user mode. Furthermore, a process should have at most one signal handler executing at any time. This implies that when the signal handler returns, control must go back to the kernel (e.g., to check whether another signal is pending) before the target process is resumed from wherever it was. Achieving this is the tricky part of the project. It involves defining a user-level "trampoline" function and a "return signal" system call; the former calls the latter. Before the kernel diverts the process to execute the signal handler, the kernel should set up the user and kernel stacks so that when the process returns from the signal handler it will enter the trampoline function. From there, the process will enter the kernel (via the return-signal syscall), at which point, the kernel should set up the user and kernel stacks so that the process will return to the code it was about to execute when it was diverted to the signal handler.

2 System Calls and the Application Interface

The Signal() system call has two arguments: a signal handler function and a signal number. It registers the function as the handler for the signal. the signal number. The Signal() call can set the handler argument to a behavior, for example, ignore a signal or return to default behavior.

Registered signal handlers are preserved across Fork(), and discarded across Exec() for reasons that should be obvious.

The Kill() system call has two arguments: a signal number and a process PID. It delivers the signal to process with the given PID. Signal delivery (i.e., execution of the signal handler) need not take place synchronously; rather, a signal may be queued for later delivery. This is comparable to how an interrupt might arrive while the processor has interrupts disabled: the interrupt will be delivered once interrupts are enabled. In the signals case, the signal may be delivered just as the process is about to regain the processor.

Other actions generate signals, including the death of a child that is not being Wait()ed for (SIGCHLD), a write to a pipe that has no readers (SIGPIPE), or (not for this assignment) a countdown timer alarm (SIGALARM).

3 Getting Started

Implement the following system calls.

Sys_Signal: This system call registers a signal handler for a signal number. The signal handler is a function that takes the signal number as an argument (it may not be useful to it), processes the signal in some way, then returns nothing (void). If called with SIGKILL, return an error (EINVALID). The handler may be set as the pre-defined "SIG_DFL" or "SIG_IGN" handlers. SIG_IGN tells the kernel that the process wants to ignore the signal (it need not be delivered). SIG_DFL tells the kernel to revert to its default behavior, which is to terminate the process on KILL, PIPE, USR1, and USR2, and to discard (ignore) SIGCHLD. A process may need to set SIG_DFL after setting the handler to something else.

Sys_RegDeliver: This system call registers the "trampoline" function. This function does only one thing: invoke the system call Sys_ReturnSignal (see below). The trampoline function is executed at the conclusion of signal handler. The RegDeliver system call is invoked by Sig_Init when called by the _Entry function in src/libc/entry.c; i.e., this function is invoked prior to running the user program's main().

Sys_Kill: This system call sends a signal to a process. Its arguments are the PID of the target process and the signal number. It should set a flag in the target process, so that when the target process is about to start executing in user space again, rather than returning to where it left off, it will execute the appropriate signal handler instead.

Sys_ReturnSignal: This system call is not invoked by user-space programs directly, but rather is invoked by the trampoline function. The latter is executed when the signal handler returns.

Sys_WaitNoPID: The Sys_Wait system call takes as its argument the PID of the child process to wait for, and returns when that process dies. The Sys_WaitNoPID call, in contrast, takes a pointer to an integer as its argument, and reaps any child process that happens to be a zombie. It places the exit status of the child process in the memory location the argument points to and returns the pid of the zombie. If there are no dead child process, then the system call should return ENOZOMBIES.

3.1 The default handler

If the target process has the default handler for a signal that terminates the process (e.g., SIGKILL), `Print("Terminated %d.\n", g_currentThread->pid);` and invoke Exit with status value 256 + the signal number.

3.2 Reentrancy and Preemption

Sending a signal should appear as if setting a flag in the PCB about the pending signal; the signal handler need not be executed immediately. In particular, if the process is executing a signal handler, do not start executing another signal handler. Further, multiple invocations of kill() to send the same signal to the same process before it begins handling even one will have the same effect as just one invocation of kill(). For example, if two children finish while another handler is executing (and blocked), the SIGCHLD handler will be called only once. However, if one child finishes while the parent's SIGCHLD handler is executing, another SIGCHLD handler should be called. See the sigaction() man page if in doubt about reentrancy. The delivery order of pending signals is not specified. (They need not be delivered in the order received.)

```
/*
 * This struct reflects the contents of the stack when
 * a C interrupt handler function is called.
 * It must be kept up to date with the code in "lowlevel.asm".
 */
struct Interrupt_State {
    /*
     * The register contents at the time of the exception.
     * We save these explicitly.
     */
    uint_t gs;
    uint_t fs;
    uint_t es;
    uint_t ds;
    uint_t ebp;
    uint_t edi;
    uint_t esi;
    uint_t edx;
    uint_t ecx;
    uint_t ebx;
    uint_t eax;

    /*
     * We explicitly push the interrupt number.
     * This makes it easy for the handler function to determine
     * which interrupt occurred.
     */
    uint_t intNum;

    /*
     * This may be pushed by the processor; if not, we push
     * a dummy error code, so the stack layout is the same
     * for every type of interrupt.
     */
    uint_t errorCode;

    /* These are always pushed on the stack by the processor. */
    uint_t eip;
    uint_t cs;
    uint_t eflags;
};

/*
 * An interrupt that occurred in user mode.
 * If Is_User_Interrupt(state) returns true, then the
 * Interrupt_State object may be cast to this kind of struct.
 */
struct User_Interrupt_State {
    struct Interrupt_State state;
    uint_t espUser;
    uint_t ssUser;
};
```

Figure 1: User Interrupt State

4 Helpers in signal.c

To implement signal delivery, you will need to implement (at least) three routines in src/geekos/signal.c:

Check_Pending_Signal: This is called by code in lowlevel.asm when a kernel thread is about to be dispatched. It returns true if the following THREE conditions hold:

1. A signal is pending for that process process.
2. The process is about to start executing in user space. This can be determined by checking the Interrupt_State's CS register: if it is not the kernel's CS register (see include/geekos/defs.h), then the process is about to return to user space.
3. The process is not currently handling another signal (recall that signal handling is non-reentrant).

Setup_Frame: This is called when Check_Pending_Signal returns true for a process. It sets up the process's user stack and kernel stack so that when the process resumes execution, it starts executing the correct signal handler, and when that handler completes, the process will invoke the trampoline function (which issues Sys_ReturnSignal system call). IF instead the process is relying on SIG_IGN or SIG_DFL, handle the signal within the kernel. IF the process has defined a signal handler for this signal, Setup_Frame has to do the following:

1. Choose the correct handler to invoke.
2. Acquire the pointer to the top of the user stack. This pointer is below the saved interrupt state stored on the kernel stack (as shown in the figure above).
3. Push onto the user stack a snapshot of the interrupt state that is currently stored at the top of the kernel stack. The interrupt state is the topmost portion of the kernel stack, defined in include/geekos/int.h in struct Interrupt_State, shown above.
4. Push onto the user stack the number of the signal being delivered.
5. Push onto the user stack the address of the "trampoline" (which was registered by the Sys_RegDeliver system call, mentioned above).
6. Change the current kernel stack such that (notice that you already saved a copy in the user stack)
   (a) The user stack pointer is updated to reflect the changes made in steps 3–5.
   (b) The saved program counter (eip) points to the signal handler.

Complete_Handler: This routine should be called when the Sys_ReturnSignal call is invoked (when a signal handler has completed). It needs to restore back on the top of the kernel stack the snapshot of the interrupt state currently on the top of the user stack.

5 Hints

Remember that the "call" assembly instruction does two things: it pushes the address of the next instruction on the stack as the return address, and it sets the processor's instruction pointer to the top of the called routine. To invoke a function in assembly (using x86 conventions) requires:

1. saving any caller-save registers (not necessary for us),
2. pushing the arguments right-to-left onto the stack.
3. calling the function,
4. popping the arguments off (or, equivalently, incrementing the stack pointer above the arguments),
5. restoring any saved caller-save registers (not needed for us).

You'll probably forget to push or pop something, creating an off-by-something error on a stack pointer that will lead to an exception. You should be able to tell which direction you're off by looking for values that are in the wrong place (for example, finding a segment number in the base pointer field).

If you would like to blow your mind, read `https://cseweb.ucsd.edu/~hovav/dist/rop.pdf` or maybe a summary `http://en.wikipedia.org/wiki/Return-oriented_programming`. We use this sort of technique (point the return address to a function) for good, but it could be powerful evil.