

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 assignment, adding changes to the user stack.

1 Signals

A signal is an inter-process communication mechanism that involves causing one of a small array of functions to be invoked in another process. Each process can use the “signal” system call to manipulate a table that lists signal handlers (function pointers) to be invoked at the request of another process. To send a signal, a process calls the “kill” system call with the pid of the target and the signal number to send. The signal number represents the index into the table of signal handlers in the target process. The kernel will then arrange for the signal handler function to be called within the process. This is your main task.

The tricky part is to ensure that when the signal handler returns, control is passed back to the kernel and the process can resume from wherever it was. To accomplish this task, we define a “return signal” system call that will be invoked as the signal handler returns. In implementing signals, you will need to arrange for the process to have both a context when executing the signal handler and a saved context that the signal handler will return to after the signal handler completes.

2 System Calls and the Application Interface

The `Signal()` system call will register a handler, which will be a function that takes an integer argument representing the signal number. The `Signal()` call can also take a behavior, to, 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 will deliver a signal to a given process. Signal delivery 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`).

2.1 Repository

```
cd geekos-project && git pull
```

Using “git”, occasionally you will want to invoke “git pull” to grab any changes we may have made, e.g., to a `.submit` file or the `projects.h` file.

The `projects.h` file includes flags that cause different `TODO()` macros in the distribution to fire.

3 Getting Started

Implement the following system calls.

Sys_Signal: This system call registers a signal handler for a given 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 (EINVAL). 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: The signal handling infrastructure requires a special “trampoline” function to be implemented. This “trampoline” invokes the system call Sys_ReturnSignal (see below) 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: In this project, Kill will be used to send a signal to a certain process. So in addition to the PID, Sys_Kill will take a signal number as defined above. It should be implemented as setting a flag in the process to which the signal is to be delivered, so that when the given 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 executed by some stub code at the completion of a signal handler. That is, Sys_Kill/Send.Signal sends process P a signal, which causes it to run its signal handler. When this handler completes, we will have set up its stack so that it will “return” to the trampoline registered by Sys_RegDeliver. This trampoline will invoke Sys_ReturnSignal to indicate that signal handling is complete.

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 zombie child process that happens to have terminated. It places the exit status 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 default handler is invoked for a signal that will terminate the process (e.g., SIGKILL), `Print("Terminated %d.\n", g_currentThread->pid);` and invoke `Exit`.

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.)

4 Helpers in signal.c

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

```

/*
 * 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

Send_Signal: takes as its arguments a pointer to the kernel thread to which to deliver the signal, and the signal number to deliver. This should set a flag in the given thread to indicate that a signal is pending. This flag is used by `Check_Pending_Signal`, described next.

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

1. A signal is pending for that user 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).

Set_Handler: use this routine to register a signal handler provided by the `Sys_Signal` system call.

Setup_Frame: this routine is called when `Check_Pending_Signal` returns true, to set up a user process's user stack and kernel stack so that when it starts executing, it will execute the correct signal handler, and when that handler completes, the process will invoke the `Sys_ReturnSignal` system call to go back to what it was doing. 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, this function will have to do the following:

1. Choose the correct handler to invoke.
2. Acquire the pointer to the top of the user stack. This 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 "signal trampoline" that invokes the `Sys_ReturnSignal` system call, and 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 step 3 - 5.
 - (b) The saved program counter (`eip`) points to the signal handler.

Complete_Handler: this routine should be called (by your code) when the `Sys_ReturnSignal` call is invoked, to indicate 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 overwrites the 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 onto the stack right-to-left,
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.