

Project 2: Signals

Consult the submit server for deadline date and time

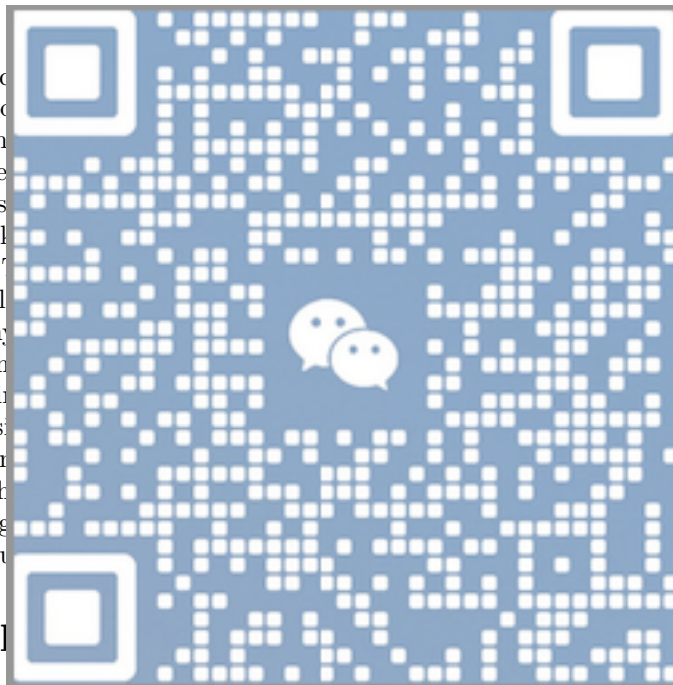
You will implement the `Signal()` and `Kill()` system calls, preserving basic functions of pipe (to permit `SIGPIPE`) and fork (to permit `SIGCHLD`).

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 (the target) to execute one of a set of signal handlers (function pointers) that manipulate its table, the signal handler table. A process can send a signal to another process by specifying the signal number. The kernel then looks up the signal number for that signal number.

Signal handlers should be able to execute at any time, even when the process is in the kernel (e.g., to change the process state). Achieving this requires a “return signal” function and a “return signal” function to execute the signal handler. When the process returns from the signal handler, the kernel (via the return-signal function) that the process will return to the user stack.



Each process has a table of signal handlers. A process can send a signal to another process by specifying the signal number. The kernel then looks up the signal number for that signal number.

Each process has at most one signal handler for each signal. When a signal is received, control must go back to the user stack. The process is resumed from the user stack. A user-level “trampoline” function is used to divert the process to the kernel. When the process returns from the signal handler, the kernel (via the return-signal function) that the process will return to the user stack.

2 System Calls

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()` 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`).

Implement the following system calls.

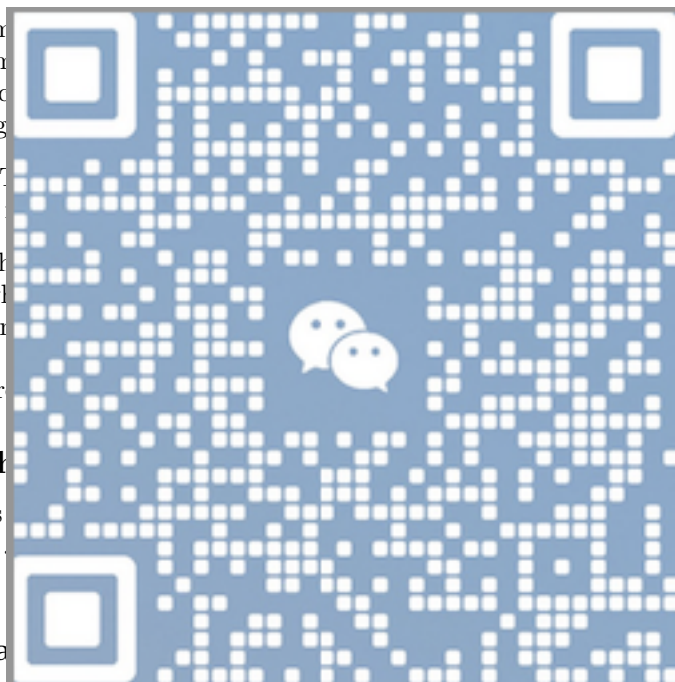
Sys_RegDeliver: This system call registers the “trampoline” function. This function does only one thing: invoke the system call Sys_ReturnSignal (see below). The trampoline function is executed at the conclusion of signal handler. The RegDeliver system call is invoked by Sig_Init when called by the _Entry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

Sys_ReturnSignal: `return` is not strictly speaking a signal, but rather is invoked by the trampoline and returns.

Sys_WaitNoPID: The child process to wait for, and returns when the child process terminates. It takes an integer as its argument, which is the pid of the child. It returns the status of the child process. If the child process is a zombie, it returns the status of the zombie. If there are no child processes, it returns -1 and sets errno to ECHILD.

If the target process has `SIGKILL` as (e.g., `SIGKILL`), `Print("Terminated %d", sigval)` thus value 256 + the signal number.

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 onto the stack.
     * This makes it easy to find out which interrupt occurred.
     */
    uint_t intNum;

    /*
     * This may be pushed onto the stack as
     * a dummy error code for every type of interrupt.
     */
    uint_t errorCode;

    /* These are always pushed onto the stack */
    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.
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 user stack to use.
2. Acquire the process's saved interrupt state.
3. Push onto the kernel stack the interrupt state currently stored at the top of the kernel stack, defined in `include/geekos/defs.h`.
4. Push onto the kernel stack the address of the signal handler.
5. Push onto the kernel stack the address of the `Sys_RegDeliver` system call, which will be invoked when the signal handler has completed.
6. Change the current user stack to the user stack defined in `include/geekos/defs.h`.
 - (a) The user stack.
 - (b) The saved interrupt state.

Complete_Handler: This is called when the signal handler has completed. It pushes the interrupt state back onto the kernel stack and invokes the trampoline function.

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.