CS 214 / 2021-03-24
==========

Assignment 3 preview:

You will need to
    - fork a child process
    - use execl or execv to run another program
    - hard mode:
        - redirect the program's standard output to a pipe
        - that is, the parent must read the output of the child

Simple way to send input from one fd to another
    int infd, outfd, bytes;
    char buf[SIZE];

    while ((bytes = read(infd, buf, SIZE)) > 0) {
        write(outfd, buf, bytes);
    }

---

recap:
    traps / hardware interrupts
        - a mechanism that allows the CPU to react to something immediately
        - suspends the current process and switches to some designated OS code
        - when the trap handler completes, OS can resume the suspended process
        - not something that user programs deal with directly

related idea: signals
- mechanism for communicating with a running process
- signals are sent to a process (from OS or other processes or the same process)
- they start out as "pending"
- normally, after some short period, they are "delivered" to the process

- for each signal, we declare a "disposition"
    - block the signal (leaves it pending; may get delivered later if signal is unblock)
    - ignore the signal
    - terminate process
    - execute a signal handler

a signal handler is a function that will be called when the signal is delivered

We can declare our own signal handlers using signal

        #include <signal.h>

        typedef void (*sighandler_t)(int);

        sighandler_t signal(int signum, sighandler_t handler);

manual for signal function: man 2 signal
manual for signals in general: man 7 signal

signal registers a signal handler
    - There is a table somewhere that holds our process's disposition for each
      signal
    - When the OS delivers a signal, it calls the appropriate signal handler

signal(int signal_number, sighandler_t signal_handler)

    sighandler_t is usually a void function that takes an int
    or it could be:
        SIG_DFL - the default handler for this signal
        SIG_IGN - ignore this signal

When our process starts, it has a default disposition for every signal
    - what that will do depends on the specific signal
        - some terminate the process
        - some terminate the process and create a core file
        - some stop the process (but it can be resumed later)
        - some are ignored

    man 7 signal lists the default behaviors

To register a signal handler:

    sighandler_t prev = signal(SIGINT, interrupt_handler);

    prev is the previous signal handler, or SIG_ERR

    -> any time you register a signal, you should check for SIG_ERR

To ignore ^C

    if (signal(SIGINT, SIG_IGN) == SIG_ERR) {
        // didn't work
    }

    // now we are immune to ^C
    // not usually a good idea (makes it harder for user to stop the program)
    // if we block SIGINT, then users will have to use SIGKILL to stop our program
    //   SIGKILL is what kill -9 or kill -KILL sends
    // we cannot set a handler for SIGKILL, so we can't do any cleanup if we
    //   receive it

-> common technique is to intercept SIGINT, set a global variable, and then
    shut down cleanly

notes on signal handlers
-> signal handlers are functions that may be called from anywhere
    - normally, the current function is interrupted and the signal handler
      is added to the call stack as though it had been called
    - it is a function call that could happen at any time

-> it generally isn't a good idea to do a lot of work in a signal handler
    - not all library functions are safe to call from a signal handler
        - e.g., you might get the signal in the middle of calling printf
          calling printf again may cause problems
    - you may get another signal while you are executing the signal handler
    - normally, a signal handler will not be interrupted by itself

signal does not behave consistently across Posix implementations
    - the main difference has to do with what happens if you receive the same
      signal while the signal handler is running
      possibility 1: use the default handler for that signal
      possibility 2: block the signal until the handler completes
    - portable code cannot assume which of these is being used
    - GCC on the iLab uses #2

For portable code, use sigaction
    - Posix (may not be available on non-Posix systems)
    - more powerful and flexible
    - more work to use

GCC Manual discussing signals
    https://www.gnu.org/software/libc/manual/html_node/Signal-Handling.html

Blocking signals
- in addition to signal handlers, our process has a "signal mask"
    - for each signal, the mask says that signal is blocked
    - a blocked signal stays pending; it does not get delivered
    - we can change the mask while we are running
    - if we unblock a signal, and there is a pending message for that signal,
      we receive it at that time

Waiting for signals
    int pause(void);

    pause suspends the current program until a signal is received
    it returns the signal that was received, or -1 on error

    e.g., if we didn't have sleep, we could implement our own using
        alarm(some_time); // set an alarm -> receive SIGALRM when it is up
        pause();   // wait until a signal is received

    -> be sure to override the default behavior for SIGALRM first!

Termination signals
Different signals are used to terminate our process in different circumstances
SIGHUP - "hang up"; sent if the shell that started our process ends
           (e.g., we closed the window or logged out)
SIGINT - "interrupt from keyboard" - user typed ^C
SIGTERM - "terminate", default signal for kill
SIGQUIT - "terminate and dump core"; used for debugging (type ^\)
SIGKILL - "kill", terminate without cleanup
           processes cannot handle or ignore SIGKILL; it always terminates

Stop and continue
SIGSTOP - stop signal; cannot be handled or ignored
SIGTSTP - "typed stop"; sent when user types ^Z; can be handled
SIGCONT - continue signal; sent when resuming a stopped process

Many other signals terminate by default
- usually because our program hit an error and can't safely proceed
    - division by zero
    - illegal memory access
    - malformed instruction
    - others (e.g., arithmetic errors)

- be careful when writing a handler for error condition
    - the error condition will still be there if the handler resumes
    - the only safe thing you can do is terminate the process or jump to somewhere
      else in the program (siglongjmp)

Aside: stopping and restarting processes in the shell

    Use ^Z to stop the current process
        Usually prints a message like: [1] Stopped your_program
        1 is the "job number"
            use jobs to get the current list of jobs
            processes started by the shell have job numbers
            all processes have PIDs
        Stopped is the current state of the process

    To restart the process, use fg or bg
    fg [job number]
        resumes process in the foreground
    bg [job number]
        resumes process in the background (concurrent with shell)

    Either give job number, or leave it out
        defaults to the most recent job

    To start a process in the background, put & at the end of a command

        $ ./long_process_to_finish &
        [1] Running ./long_process_to_finish
        $

    kill sends a signal to a process

        kill PID   - send SIGTERM to process PID
        kill -KILL PID - send SIGKILL to process PID

        Use %N to get the PID of job N

            $ ./long_process &
            [1] 10234
            $ kill %1
            $
            [1]+ Done   ./long_process

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <signal.h>

volatile int signo = 0;
    // marked volatile because it may change asynchronously
    // that is, signals may be received at any time


// very simple signal handler
// note that it returns normally, so we can only use it with non-error signals
void handler(int signum)
{
    signo = signum;
}

// a very simple exit handler
void make_a_note(void)
{
    puts("We are in the exit handler");
}

int main(int argc, char **argv)
{
    // register an exit handler: make_a_note will be called after main returns
    atexit(make_a_note);

    // register some signal handlers
    // we can reuse handler because it will receive the signal
    signal(SIGHUP, handler);
    signal(SIGINT, handler);
    signal(SIGTERM, handler);
    signal(SIGCONT, handler);

    pause();  // stops the process until a signal is received
    // alternative:
    // while (signo == 0) { puts("Waiting"); sleep(1); }

    if (signo > 0) psignal(signo, "caught signal");

    return EXIT_SUCCESS;
}
```