Lecture 6, Jan 19, 2024

Basic IPC

- IPC is any method to transfer bytes between two or more processes
- Reading and writing files is a form of IPC
- The `ssize_t read(int fd, void buf[.count], size_t count);` syscall reads from a file and returns number of bytes read
  - 0 is returned on end of file or a closed file descriptor
  - We should check for errors!
- The `ssize_t write(int fd, const void buf[.count], size_t count);` syscall writes to a file and returns number of bytes written
- Linux always uses the lowest available file descriptor for new opened files, so we could close file descriptor 0 and open a new file, and this will replace stdin
  - Similarly we can also replace stdout, stderr, etc
  - This can be done in a shell using a redirect, e.g. `./program < input_file > output_file`
  - Without changing the code the program can work with any type of input/output stream

Signals

- *Signals* are a form of IPC that interrupts the program
  - The kernel sends a number to the program indicating the type of signal
  - Note signals can interrupt syscalls like `read()` and `write()`, resulting in an `EINTR`
- Using Ctrl+C sends a `SIGINT` (interrupt) to the program, which is a signal
- We can write handlers to handle these signals
  - The default handler behaviour is to exit the program with an exit code of 128 + signal number
  - We can write handlers to ignore the signals so the program won't exit immediately
- Using the `sigaction()` syscall allows us to define our own signal handlers
  - Signal handlers return nothing and takes an int argument, which is the signal number
  - Some standard signal numbers:
    - 2: `SIGINT` (interrupt from keyboard)
    - 9: `SIGKILL` (terminate immediately)
    - 11: `SIGSEGV` (segmentation fault)
    - 15: `SIGTERM` (terminate)
      - Can be ignored
- Using the `int kill(pid_t pid, int sig);` syscall we can send signals manually
  - We can use the `kill` command to send signals (by default sends `SIGTERM`); use `pidof` to get a process's PID

Interrupts

- Most operations are non-blocking, i.e. returning immediately and we check later if something occurs
- `read()`, `write()`, `wait()` are blocking by default, but they have nonblocking variants/arguments
- To react to changes to a non-blocking call, we can use a *poll* or *interrupt*
- Polling continuously checks for changes
  - Very simple to setup and we don't have to worry about things getting interrupted
  - This is inefficient and if we don't poll fast enough our response can be delayed
- Using interrupts, we can register a signal handler to check for e.g. `SIGCHLD` when children exit, to get notified immediately