Lecture 4

Kernel Mode Transfers

- Syscall
  - Problem requests a system service, such as exit
  - Like a function call, but **outside** the process
  - Doesn't have the address of the system function to call
  - Like a Remote Procedure Call (RPC) -- arguments must then be passed in
    - Triggered by the user process
  - Marshall the syscall id and args in registers and exec syscall
- Interrupt
  - External async event triggers context switch
    - e.g. timer, i/o device.
  - Independent of user process
- Trap/Exception
  - Synchronous event in process triggers context switch
  - Most likely some error

Interrupts

- Interrupt processing not visible to the user process
  - Occurs between instructions, restarted transparently
- Interrupt controller / masking
  - Interrupts invoked by interrupt lines on devices
  - Controller chooses which interrupt requests to honor
  - The mask enables/disables the interrupt
  - The priority encoder picks the highest enabled interrupt
  - Software Interrupt will be set and cleared by the software
  - CPU can disable all interrupts w/ an internal flag
  - Non-Maskable Interrupt (NMI) line **can't** be disabled.

- Interrupt vector
    - Limited number of entry points into kernel
- Kernel interrupt stack
    - Cannot share stacks with the user.
    - By separating the stack, handler works regardless of state of user code
- Atomic transfer of control
    - "Single instruction" - like to change:
        - program counter
        - stack pointer
        - mem protection
        - kernel/user mode

During a syscall / interrupt, we will save stack from user memory onto the kernel exception stack

Managing Processes

- How to manage state of process?
    - Create
    - Exit
- Everything outside of the kernel is running in a process
- Processes are created and managed by processes

Bootstrapping: First process (init, pid 0) is started by the kernel, after which all processes will be created by other processes

API:

- `exit` - terminate a process (`pthread_exit`)
- `fork` - copy current process (`pthread_create`)
- `exec` - change program being run by current proc
- `wait` - wait for proc to finish (`pthread_join`)
- `kill` - send a signal (interrupt-like notif) to another proc

- `sigaction` - set handlers for signals

Forking

- `pid_t fork()` -- copy the current process
  - new process has different pid
  - new process contains one thread
- return value from `fork()` : `pid` (int)
  - when > 0:
    - running in original (parent) process
    - return value is `pid` of new child
  - when = 0:
    - running in new child process
  - when < 0:
    - error, must be handled somehow
    - running in original process
- fork race
  - prints things in a random order
  - sleeping each process doesn't help

Sigaction

Needs a signal handler

Common POSIX signals:

- `SIGINT` : ctrl-C
- `SIGTERM` : `kill` shell command
- `SIGSTOP` : ctrl-Z
- `SIGKILL` , `SIGSTOP` : terminate/stop program. can't be changed with sigaction

Files

Unix/POSIX idea: **everything** is a file. There are identical interfaces for:

- files on disk

- devices (terminals, printers, etc)
- regular files on disk
- networking sockets

Based on the system calls `open`, `read`, `write`, and `closed`. Additionally, `ioctl()` for custom configuration that doesn't quite fit the above.

- was radical when proposed, but hasn't been changed since.
- so they prolly did something right.

Every process has a current working directory (cwd). Absolute paths ignore it, and relative paths are relative to it. `.` is the cwd itself, and `..` is the parent dir.

- can be set with a syscall

File API: Operates on "streams" -- unformatted sequences of bytes with a position.

- Open streams are represented by a pointer to a `FILE` ds
- Error is reported with a null response
- Three predefined streams, `stdin`, `stdout`, and `stderr`, are opened implicitly when prog is executed
- All three can be redirected