Outline

- UNIX Signals: kill, signal system calls
- Kernel-level Threads
- User-level Threads
- Intro to Concurrency and Locks

UNIX Signals

- Signals are a limited form of inter-process communication
- When a signal is sent, the operating system interrupts the target process's normal flow of execution to deliver a signal.
- If the process has previously registered a signal handler, that routine is executed. Otherwise, the default signal handler is executed.
- Examples:
    - Ctrl-C sends SIGINT; by default, this causes the process to terminate
    - Ctrl-Z sends SIGTSTP; by default, this causes the process to suspend execution
    - SIGCHLD signal is sent to a process when a child process terminates.
    - SIGFPE is the floating point exception (for example, on divide by zero)
    - SIGSEGV on segmentation fault
    - SIGUSR1 and SIGUSR2 are user-defined signals
- `kill(pid, signum)`: send signal SIG to PID.
- `signal(signum, void (*handler)(int))`: associates HANDLER with signal SIGNUM.

Shell Backgrounding

- How do you create a background job?

```
    $ compute &
```

- How does the shell implement "&", backgrounding? (Don't call wait immediately).
- Q: What if a background process exits while sh waits for a foreground process?
- *Think about* how a shell implements the following functionalities.
    - Lists of commands, separated by ";" (e.g., touch a; ls)
    - Sub shells implemented using "(" and ")". For example, a subshell can be used to setup a "dedicated environment" for a command group:

```
COMMAND1;
COMMAND2;
(
  PATH=/bin
  COMMAND3
  COMMAND4
);
COMMAND5
```

Interesting uses of open/read/write/close

- Linux has a nice representation of a process and its FDs, under /proc/PID/
    - maps: VA range, perms (p=private, s=shared), offset, dev, inode, pathname
    - fd: symlinks to files pointed to by each fd. (what's missing in this representation?)
    - can do fd manipulation in shell and see it reflected in /proc/$$/fd
    - can read or write kernel parameters using read() and write() syscalls!

Threads

Figure on process address space: code, static data, stack, heap. On fork, the whole address space gets replicated. On thread create, the created thread has a different program counter, registers, and stack (through stack pointer). Everything else is shared between threads.

Kernel-level threads are just processes minus separate address spaces. Discuss the kernel scheduler which is invoked at every timer interrupt. Each thread is an independent entity for the kernel.

Write the `cswitch` function for processes and threads. Notice that switching among threads requires no privileged operations. Switching the stack can be done by switching the `sp` register. A cswitch needs to be fast (typically a few 100 microseconds).

Advantages of threads over processes

- *Much* more lightweight than processes. Faster creation, deletion, switching.
- Much faster communication among threads: allow shared data structures to be maintained.

User-level threads can be implemented inside a process by writing `scheduler()` and `cswitch()` functions. The scheduler can be called periodically using SIGALRM signal.

Pros of user-level threads:

- Lighter-weight
- Faster cswitch
- Do not need kernel's permission

Cons of user-level threads:

- Will not be scheduled on different cores, because they look like one process to the kernel.
- If one thread blocks (e.g., on I/O), all threads block.

Threading models (slide)