CSE 451: Operating Systems
Spring 2011

Module 3: Processes

John Zahorjan
zahorjan@cs.washington.edu
Allen Center 534

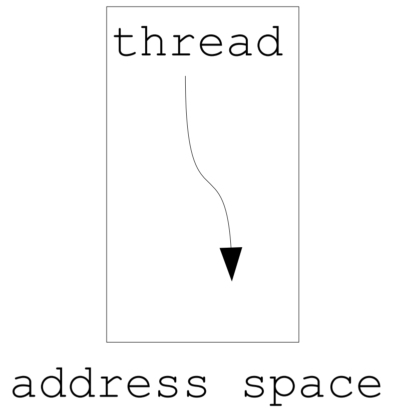
Process management

- This module begins a series of topics on processes, threads, and synchronization
  - this is the most important part of the class
  - there definitely will be several questions on these topics on the midterm

- Today: processes and process management
  1. What is a “process”?
  2. What's the process namespace?
  3. How are processes represented inside the OS?
  4. The execution states of a process?
  5. How are they created?
  6. Making creation fast(er)
  7. Shells
  8. An example of process-process communication: signals

1. What is a process?

- The process is the OS's abstraction for execution
  - A process is a program in execution
- It's the OS-provided higher level abstraction for the hardware CPU and main memory resources
  - E.g., notions of real time are simplified to sequential execution of successive instructions
- The simplest (classic) case is the sequential process
  - An address space (abstraction of memory)
  - A single thread (abstraction of the CPU)
- A sequential process is:
  - the unit of execution
  - the unit of scheduling
  - the dynamic (active) execution context
    - compared with program: static, just a bunch of bytes

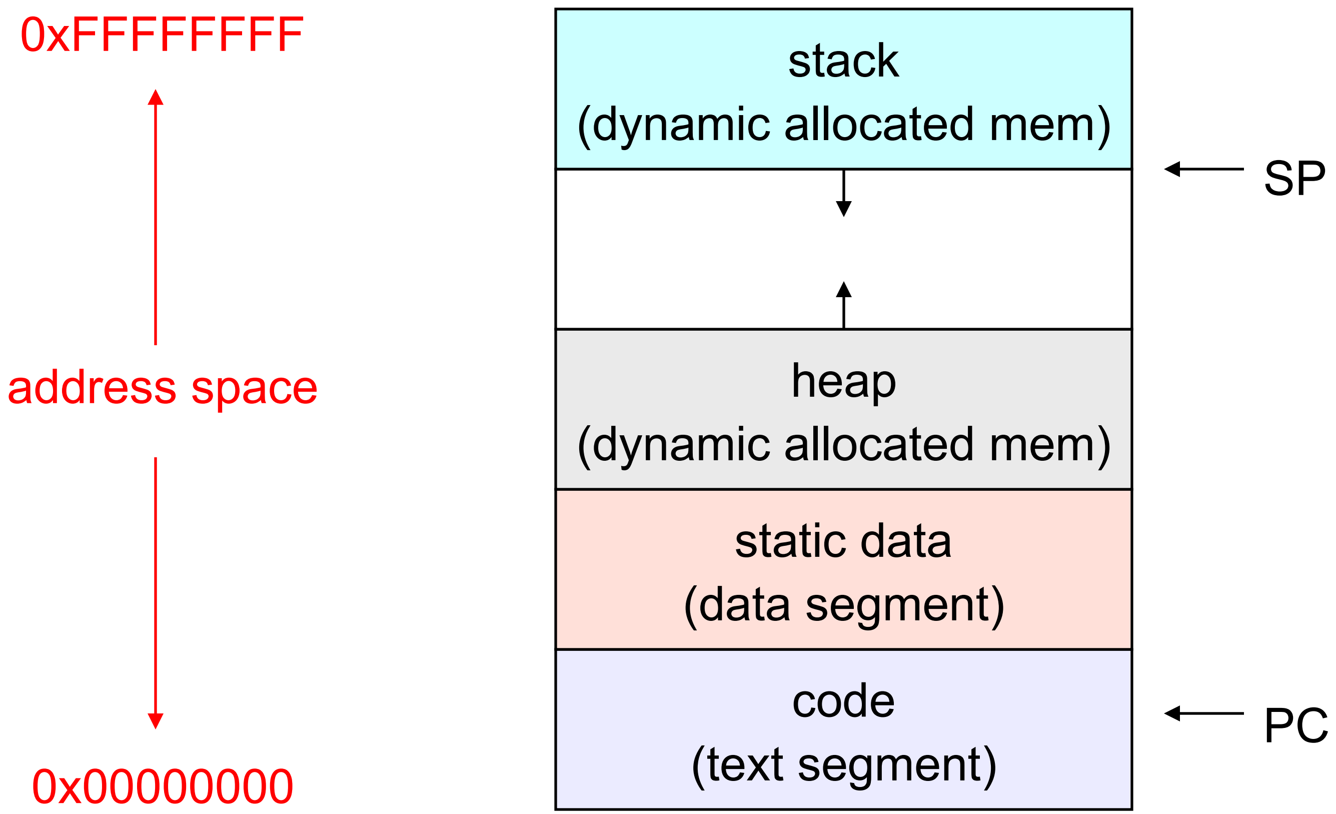


What's in a process?

- A process consists of (at least):
  - an <u>address space</u>, containing
    - the code (instructions) for the running program
    - the data for the running program
  - <u>thread state</u>, consisting of:
    - the program counter (PC), indicating the next instruction
    - the stack pointer register (implying the stack it points to)
    - Other general purpose register values
  - a set of <u>OS resources</u>
    - open files, network connections, sound channels, …
  - other <u>process metadata</u>
    - e.g., signal handlers

- In other words, it's all the stuff you need to run the program
  - or to re-start it, if it's interrupted at some point

Reminder from CSE 378: A process's address space (idealized)



2. The process namespace

- (Like most everything, the particulars depend on the particular OS)

- The name for a process is called a process ID (PID)
  - An integer

- The PID namespace is global to the system
  - Only one process at a time has a particular PID

- Operations that create processes return a PID
  - e.g., fork(), clone(), exec()

- Operations on processes take PIDs as an argument
  - e.g., kill(), wait(), nice()

3. Processes in the OS

- The kernel maintains a data structure to keep track of process state
  - Called the <u>process control block</u> (PCB)
- OS keeps all of a process's hardware execution state in the PCB when the process isn't running
  - PC, SP, registers, etc.
  - when a process is unscheduled, the state is transferred out of the hardware into the PCB
  - (when a process is running, its state is spread between the PCB and the CPU)
- Note: It's natural to think that there must be some esoteric techniques being used
  - fancy data structures that'd you'd never think of yourself

Wrong! It's pretty much just what you'd think of!

The PCB

- The PCB is a data structure with many, many fields:
  - process ID (PID)
  - parent process ID
  - execution state
  - program counter, stack pointer, registers
  - address space info
  - user id (uid)
  - group id (gid)
  - scheduling priority
  - accounting info
  - pointers for use in state queues
- In Linux:
  - defined in **`task_struct`** (**`include/linux/sched.h`**)
  - over 95 fields!!!

PCBs and hardware state

- When a process is running, its hardware state is loaded on a CPU
  - PC, SP, registers
  - CPU contains current values
- When a process is transitioned to the waiting state, the OS saves the register values in the PCB
  - when the OS returns the process to the running state, it loads the hardware registers from the values in that process's PCB
- The act of switching a CPU from one process to another is called a <u>context switch</u>
  - timesharing systems may do 100s or 1000s of switches/sec.
  - takes about 5 microseconds on today's hardware
- Choosing which process to run next is called <u>scheduling</u>

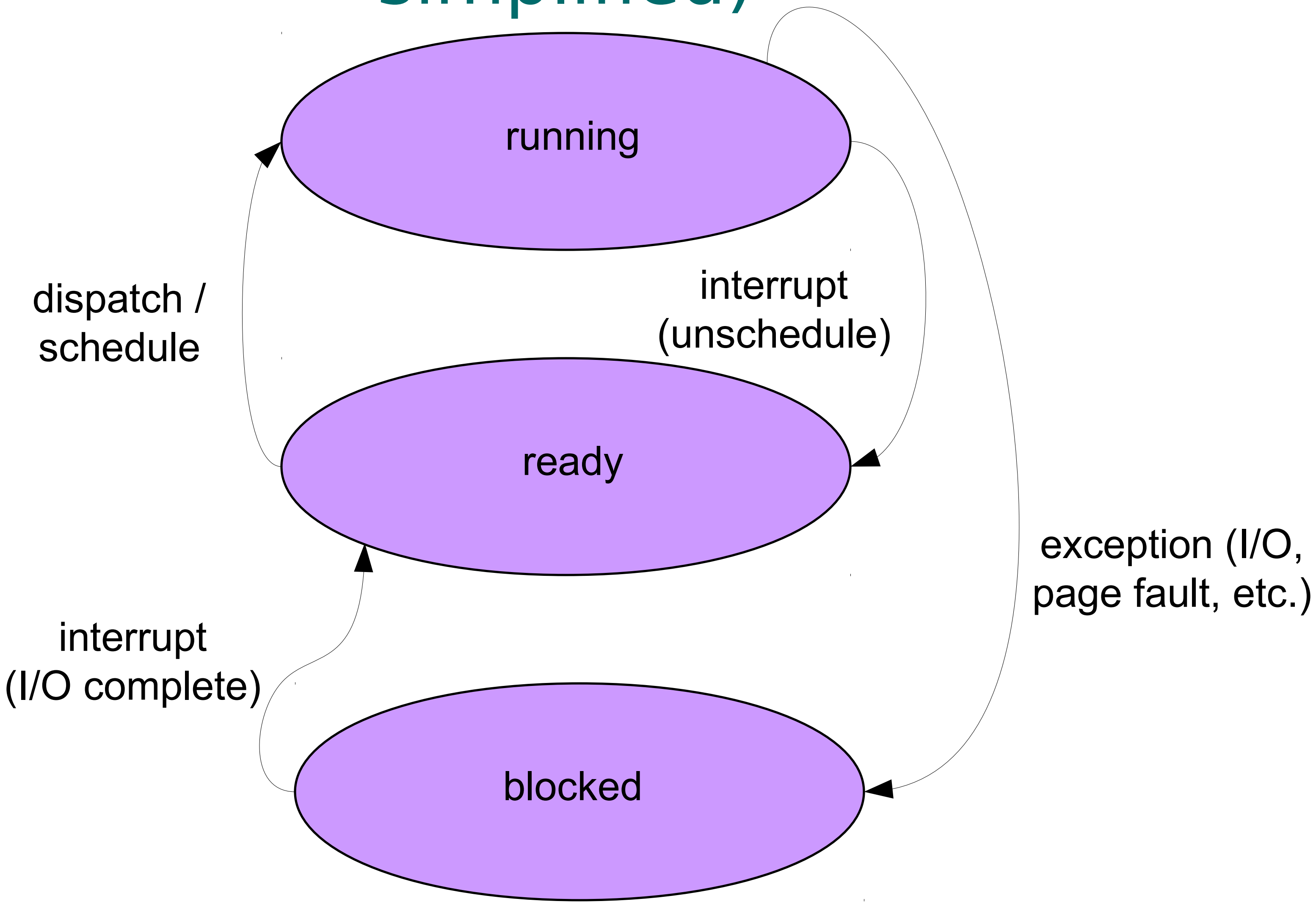
Process ID
Pointer to parent
List of children
Process state
Pointer to address space descriptor
Program counter stack pointer (all) register values
uid (user id) gid (group id) euid (effective user id)
Open file list
Scheduling priority
Accounting info
Pointers for state queues
Exit (“return”) code value

This is (a simplification of) what each of those PCBs looks like inside!

4. Process execution states

- Each process has an execution state, which indicates what it is currently doing
  - ready: waiting to be assigned to a CPU
    - could run, but another process has the CPU
  - running: executing on a CPU
    - is the process that currently controls the CPU
    - pop quiz: how many processes can be running simultaneously?
  - waiting (aka "blocked"): waiting for an event, e.g., I/O completion
    - cannot make progress until event happens
- As a process executes, it moves from state to state
  - UNIX: run **`ps`**, STAT column shows current state
  - which state is a process in most of the time?

States of a process (slightly simplified)

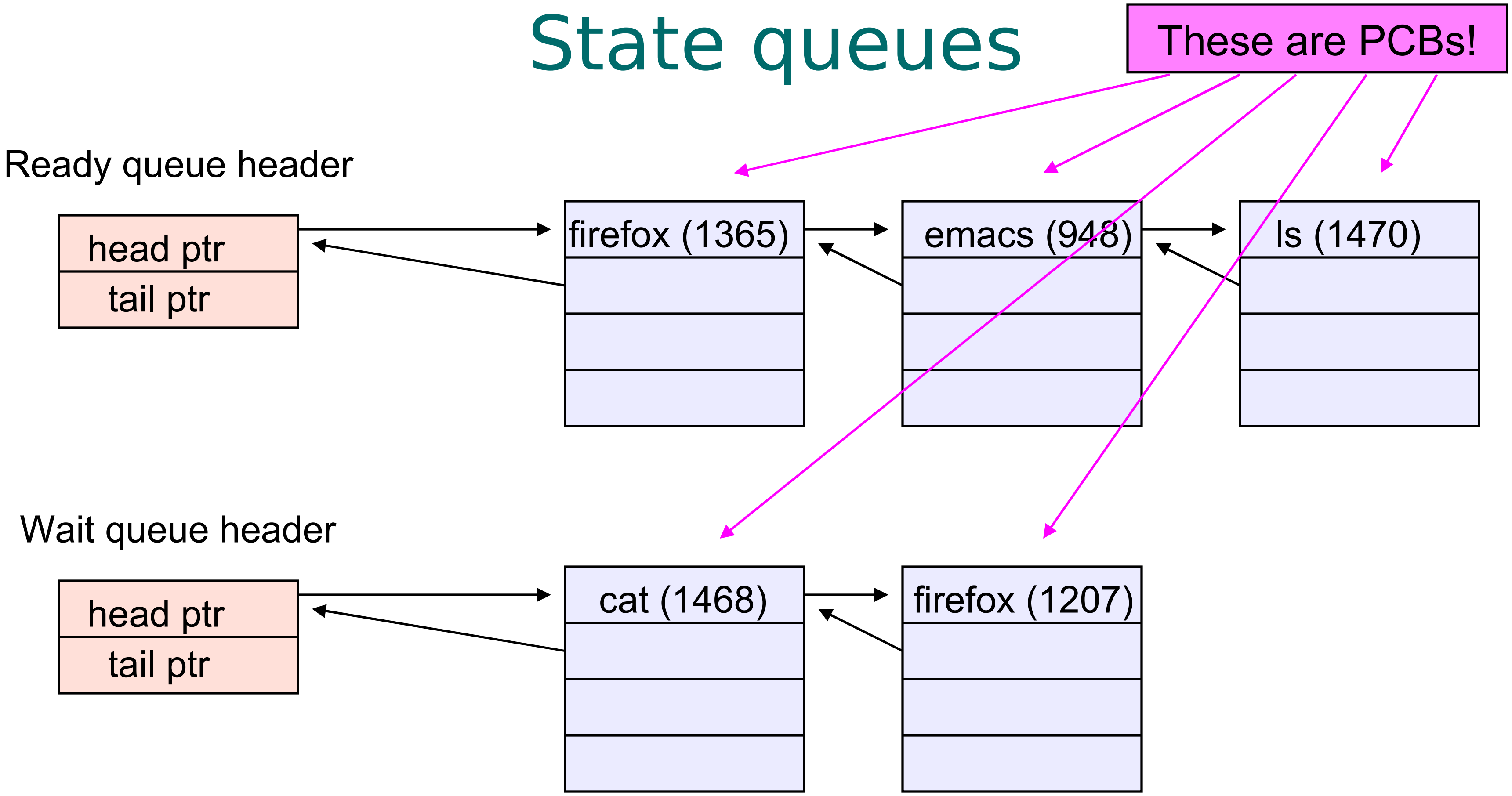


State queues

- The OS maintains a collection of queues that represent the state of all processes in the system
  - typically one queue for each state
    - e.g., ready, waiting, …
  - each PCB is queued onto a state queue according to the current state of the process it represents
  - as a process changes state, its PCB is unlinked from one queue, and linked onto another

- Once again, *this is just as straightforward as it sounds!* The PCBs are moved among queues, which are represented as linked lists. *There is no magic!*

State queues



- There may be many wait queues, one for each type of wait (particular device, timer, message, ...)

PCBs and state queues

- PCBs are data structures
  - dynamically allocated inside OS memory
- When a process is created:
  - OS allocates a PCB for it
  - OS initializes PCB
  - OS puts PCB on the correct queue
- As a process computes:
  - OS moves its PCB from queue to queue
- When a process is terminated:
  - PCB may hang around for a while (exit code…)
    - What is the process state?
  - eventually, OS deallocates its PCB

5. Process creation

- New processes are created by existing processes
  - creator is called the parent
  - created process is called the child
    - UNIX: do **`ps`**, look for PPID field
  - *what creates the first process, and when?*


init
dbus-launch
dbus-daemon [x2]
rtkit-daemon
rpcbind
upowerd
polkitd
sendmail:Queue
sshthrottle
gvfsd
gpm
hald
hald-runner
hald-addon-acpi
hald-addon-input:Listening
SCREEN
bash
gdm-binary
gdm-simple-slave
gnome-session
polkit-gnome-authentication-agent-1
gdm-simple-greeter
at-spi-registryd
metacity
Xorg
pam:gdm-password
sshd
sshd:aryan
sshd:aryan@pts/6
-bash
emacs
sshd:aryan
sshd:aryan@pts/11
-bash
emacs
sshd:zahorjan
sshd:zahorjan@pts/0
-bash
emacs
perl
ps
sshd:aryan
sshd:aryan@pts/12
-bash
emacs
java [x3]
udisks-daemon
udisks-daemon:polling
SCREEN
bash
gnome-settings-daemon
automount
crond
rpc.idmapd
SCREEN
bash [x2]
system-setup-keyboard
rpc.statd

Process Creation Semantics

- (Depending on the OS) child processes inherit certain attributes of the parent

- Examples:
  - pid/gid: implies authorization of child
  - Open file table: implies stdin/stdout/stderr
  - Environment variables
  - … other metadata

  - On some systems, resource allocation to parent may be divided among children
    - Hierarchical resource allocation limits impact of your activity on mine

UNIX process creation details

- UNIX process creation through **`fork()`** system call
  - creates and initializes a new PCB
  - creates a new address space
  - initializes new address space with a copy of the entire contents of the address space of the parent
  - initializes kernel resources of new process with resources of parent (e.g., open files)
  - places new PCB on the ready queue

- the **`fork()`** system call “returns twice”
  - once into the parent, and once into the child
  - returns the child’s PID to the parent
  - returns 0 to the child

- **`fork()`** = “clone me”

- (We'll see why in a minute...)

testparent – use of fork( )

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  char *name = argv[0];
  int pid = fork();
  if (pid == 0) {
    printf("Child of %s is %d\n", name, pid);
    return 0;
  } else {
    printf("My child is %d\n", pid);
    return 0;
  }
}
```

testparent output

```
spinlock% gcc -o testparent testparent.c
spinlock% ./testparent
My child is 486
Child of testparent is 0
spinlock% ./testparent
Child of testparent is 0
My child is 571
```

fork() … exec()

- Q: So how do we start a new program, instead of just forking the old program?
  - A: first fork, then exec

- **`int exec(char * prog, char * argv[])`**
  - **(actually, there are many flavors of exec)**

  - stops the current process
  - loads program 'prog' into the address space
    - i.e., overwrites existing process image
  - initializes hardware context, args for new program
  - places PCB onto ready queue
  - note: does not create a new process!

- To run a new program:
  - fork()
  - Child process does an exec()
  - (parent either waits for child to complete, or not)

6. Making Creation Fast(er)

- The semantics of fork() say the child's address space is a copy of the parent's

- Implementing fork() that way is slow:
  - Have to allocate physical memory for the new address space
  - Have to copy parent's address space contents into child's address space
  - Have to set up child's page tables to map new address space

- We can speed this up...

Method 1: vfork()

- vfork() is the older of the two approaches talked about here

- It's (once again) an instance of changing the problem definition into something we can implement efficiently

- Instead of “child address space is a copy of parent's,” the semantics are “child address space is the parent's”
  - With a “promise” that the child won't modify the address space before doing an exec()
    - This is unenforced. You use vfork() at your own peril.
  - When exec() is called, a new address space is created, new page tables set up for it, and it's loaded with the new executable
  - This saves the wasted effort of duplicating the parent's address space (setting up page tables and copying contents) when the child is just going to exec() anyway (which is common)

Method 2: copy-on-write

- This approach retains the original semantics, but copies “only what is necessary,” rather than the entire address space

- On fork():
  - Create a new address space
  - Initialize its page tables to the same mappings as the parent's (i.e., they both point to the same physical memory)
    - No copying of address space contents have occurred to this point
  - Set both parent and child page tables to make all pages read-only
  - If either the parent or child writes to memory, a protection fault occurs
  - When the fault occurs:
    - Allocate a new physical frame for the child, and point its page table entry at it
    - Copy the current contents of the parent address space to that frame
    - Mark the entries in both the parent's and child's address space writable for that page
    - Restart the process doing the write, re-executing the write instruction

- The result: only pages *modified* by the parent or child ever end up being copied

7. UNIX shells

```
$ ./myprog
```

```
int main(int argc, char **argv)
{
  while (1) {
    printf("$ ");
    char *cmd = get_next_command();
    int pid = fork();
    if (pid == 0) {
       exec(cmd);
       panic("exec failed!");
    } else {
       wait(pid);
    }
  }
}
```

Input/Output Redirection

- $ ./myprog <input.txt >output.txt   # UNIX
  - each process has an open file table
  - by (universal) convention:
    - 0: stdin
    - 1: stdout
    - 2: stderr
  - a child process inherits the parent's open file table
  - Redirection: open files before executing child process code

UNIX shells: input/output redirection

```
$ foo myFile.txt <input.txt >output.txt
```

```
int main(int argc, char **argv)
{
  while (1) {
    printf("$ ");
    char *cmd = get_next_command();
    int pid = fork();
    if (pid == 0) {
       manipulate stdin/stdout/stderr
       exec(cmd);
       panic("exec failed!");
    } else {
       wait(pid);
    }
  }
}
```

More…

- Note that redirection is completely transparent to the child process
- What about
  - $ ./myprog >>output.txt
  - $ ./myprog >output.txt 2>&1
  - $ ./myprog | less
  - $ ./myprog &

8. Process-process communcation via signals

- Processes can register event handlers
  - Feels a lot like event handlers in Java, which…
  - Feel sort of like catch blocks in Java programs
- When the event occurs, process asynchronously jumps to event handler routine
- Used to catch exceptions
- Also used for process-process communcation:
  - a process can trigger an event in another one using signal

Signals

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25		Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

Example Use

- You're implementing Apache, a web server
- Apache reads a configuration file when it is launched
  - Controls things like what the root directory of the web files is, what permissions there are on pieces of it, etc.
- Suppose you want to change the configuration while Apache is running
  - If you restart the currently running Apache, you drop some unknown number of user connections
- Solution: send the running Apache process a signal
  - It has registered an signal handler that gracefully re-reads the configuration file