Processes

* A process is an instance of a running program
  + One of the most profound ideas in CS
* Processes provide each program with two key abstractions
  + Logical control flow
  + Private address space

Creating and terminating processes

* Process is in one of three states
  + Running
  + Stopped
  + Terminated
* Creating processes
  + Parent process creates a new running child process by calling fork
  + Int fork(void)
    - Returns 0 to the child process, child’s PID to parent process
    - Child is almost identical to parent
      * Gets an identical (but separate) copy of the parent’s virtual address space
      * Gets identical copies of the parent’s open file descriptors
      * Has a different PID than the parent
  + Fork is interesting because it is called once but returns twice
    - It returns once for the child process (a copy of the executable code) and the parent process (the original executable code where fork was called)

System call error handling

* On error, linux system-level functions typically return -1 and set global variable errno to indicate the cause
* Hard and fast rule:
  + You must check the return status of every system-level function
  + The only exception is the handful of functions that return void
* Error-reporting functions
  + Create some wrappers for error handling to make your code more readable
    - Stevens-style error-handling wrappers: **ex on slide 8**

Example on slide 11

* Fork() returns the PID. So in the child process, fork() == 0 because the child’s PID is 0, but in the parent process fork() != 0 bc the parent has a PID. So the parent process only prints out 9

Modeling fork with process graphs

* A process graph is a useful tool for capturing the partial ordering of statements in a concurrent program:
  + Each vertex is the execution of a statement
  + A -> b means a happens before b
  + Edges can be labeled with current value of variables
  + Printf vertices can be labeled with output
  + Each graph begins with a vertex with no in-edges
* Any **topological sort** of the graph corresponds to a feasible ordering
  + Ordering of vertices where all edges point from left to right
* Process graph **example: slide 13 – 16**

Reaping child processes

* Idea
  + When a process terminates, it still consumes system resources
  + Called a zombie
    - Living corpse, half alive and half dead
* Reaping
  + Performed by parent on terminated child (using wait or waitpid)
  + Parent is given exit status information
  + Kernel then deletes zombie child process
* What if the parent doesn’t reap?
  + If any parent terminate without reaping a child, then the orphaned child will be reaped by init process (pid == 1)
  + So only need explicit reaping in long-running processes
    - Ex: shells and servers

Wait: synchronizing with children

* Parent reaps a child by calling the wait function
* Int wait(int \*child\_status)
  + Suspends current process until one of its children terminates
  + Return value is the pid of the child process that terminated
  + If child\_status != NULL, then the integer it points to will be set to a value that indicates reason the child terminated and the exit status:
    - Checked using macros defined in wait.h
      * WIFEXITED, WEXITSTATUS
* If multiple children are completed, it will take an arbitrary order
* Macros WIFEXITED and WEXITSTATUS provides information about the exit status

Execve: loading and running programs

* Int execve(char \*filename, char \*argv[], char \*envp[])
* Loads and runs in the current process:
  + Executable file filename
    - Can be an object file or script file beginning with #!interpreter (ex: #!/bin/bash)
  + With argument list argv
    - By convention argv[0] == filename
  + And environment variable list envp
* Overwrites code, data, and stack
  + Retians PID, open files and signal context
* Called once and never returns except if there is and error

Process summary

* At any given time, the system has multiple active processes
* Only one can execute at a time on a single core
* Each process appears to have total control of processor + private memory space
* Spawning processes
  + Call fork
  + One call, two returns
* Reaping and waiting for processes
  + Call wait or waitpid
* Process completion
  + Call exit
  + One call, no return
* Loading and running programs
  + Call execve or variant
  + One call, no return

Signals

* A signal is a small message that notifies a process that an event of some type has occurred in the system
  + Akin to exceptions and interrupts
  + Sent from the kernel to a process
  + Signal type is identified by small integer ID’s
  + Only information in a signal is its ID and the fact that it arrived
  + **Check slide 8 of lecture 16 for example list** of signals

Sending a signal

* Kernel sends (delivers) a signal to a destination process by updating some state in the context of the destination process
* Kernel send a signal for one of the following reasons:
  + Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD)
  + Another process has invoked the kill system call to explicitly request the kernel to send a signal to the destination process

Receiving a signal

* A destination process receives a signal when it is forced by the kernel to react in some way to the delivery of the signal
* Some possible ways to react:
  + Ignore the signal
  + Terminate the process
  + Catch the signal by executing a user-level function called signal handler
  + Check lecture 16 slide 11 for code

Default actions

* Each signal type has a predefined default action, which is one of:
  + The process terminates
  + The process stops until restarted by a SIGCONT signal
  + The process ignores the signal

Installing signal handlers

* The signal function modifies the default action associated with the receipt of signal signum:
  + Handler\_t \*signal(int signum, handler\_t \*handler)
* Different values for handler:
  + SIG\_IGN: ignore signals of type signum
  + SIG\_DFL: revert to the default action on receipt of signals of type signum
  + Otherwise, handler is the address of a user-level signal handler
  + Called when process receives signal of type signum
    - Referred to as installing the handler
    - Executing handler is called catching or handling the signal
  + When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal
* Nested signal handlers
  + Handlers can be interrupted by other handlers