Last Updated: 1-24-23

Programs & Processes

# More File I/O

* “Everything in Unix [and Posix] is a file”
* Ex: A binary file stores an array of people\_t structs- how would we read every fourth struct in the file?
  + Any open file keeps track of a position- starting point for the next read/write operation
  + Read/writes implicity advance this position by number of bytes consumed/produced
  + We can explicitly ask to adjust the current position

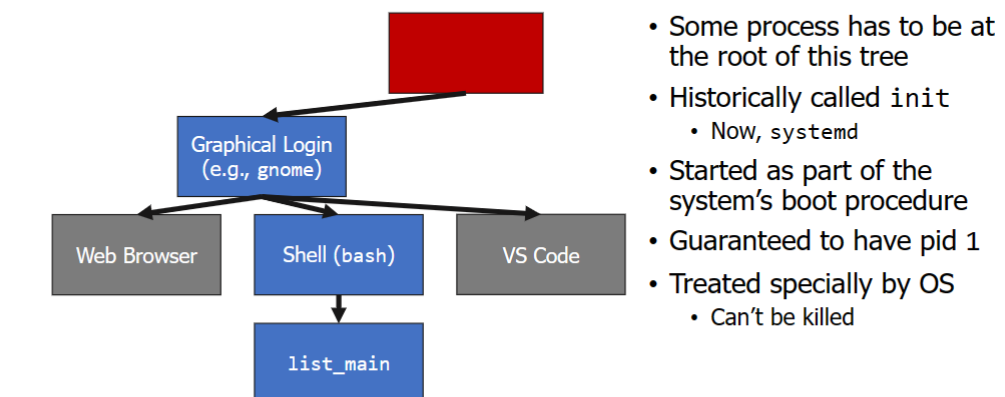
int fseek(FILE \*f, long offset, int whence)

* + - * Offset argument can be positive or negative
      * Whence determines starting point for the seek
        + SEEK\_SET: offset is relative to the beginning of the file
        + SEEK\_CUR: offset is relative to current position in file
        + SEEK\_END: offset is relative to the end of file
  + We can also ask what the current position is

ftell()

# Processes

* Program to Process
  + What does hardware give you?
    - Execution of a sequence of instructions
    - Storage in RAM
    - Other devices for input
  + What does the OS give you?
    - Multiple running programs taking turns on the CPU
    - An isolated address space through VM
    - Files for persistent storage
  + What does runtime give you?
    - Higher-level memory concepts like stack and heap
* What is a process
  + A process is an actively executing program and a live set of resources needed to support this execution
    - A unit of work
    - A unit of resource allocation and accounting
    - A boundary respecting and protecting running code
  + A process contains…
    - Its own address space
    - An execution state
    - A view of the files system
* Possible process states
  + New- being created
  + Ready- eligible for execution but waiting for a turn
  + Running- in the middle of its turn on the CPU
  + Blocked- waiting for some event
  + Done- finished execution, no longer eligible for execution
* Working w/ Processes
  + Each process has a unique numerical process ID (pid)
  + Lots of tools for managing system are based around these IDs
    - Terminal commands
      * Ps: show all running processes associated with this terminal session
      * Ps a: show all running processes on the machine
      * Ps u: show all processes for me (current user)
    - Commands for signaling
      * Kill 1234: Send process with ID 1234 the TERM signal
      * Kill -9 1234: Send process 1234 the KILL signal
      * <Ctrl-C>: send the current shell process the INT (interrupt) signal
* Our first System Calls
  + Retrieving info about current process
    - Get working directory: char \*getcwd(char \*buf, size\_t size)
    - Get process ID: pid\_t getpid()
    - Get parent’s process ID: pid\_t getppid()
  + Pid\_t: basically a typedef for some numerical type
  + Any process can create new processes and become a parent
* Process Family Tree:
  + Say we run ./list\_main from a graphical terminal window



* Process Creating: fork
  + System call to create a new processL pid\_t fork()
  + Creates a complete copy of the calling process that inherits:
    - Parent’s address space contents (stack, heap, code, etc.) and register state
    - Parent’s current working directory
    - Parent’s open files
  + One important exception: the return value of fork
    - In parentL return value is the process ID of new child
    - In child: return value is 0
    - Negative return value: error occurred, no child created
  + Weird way to think about this: fork is called once, but returns twice
    - Both parent and child continue of their way, starting at return from fork call+