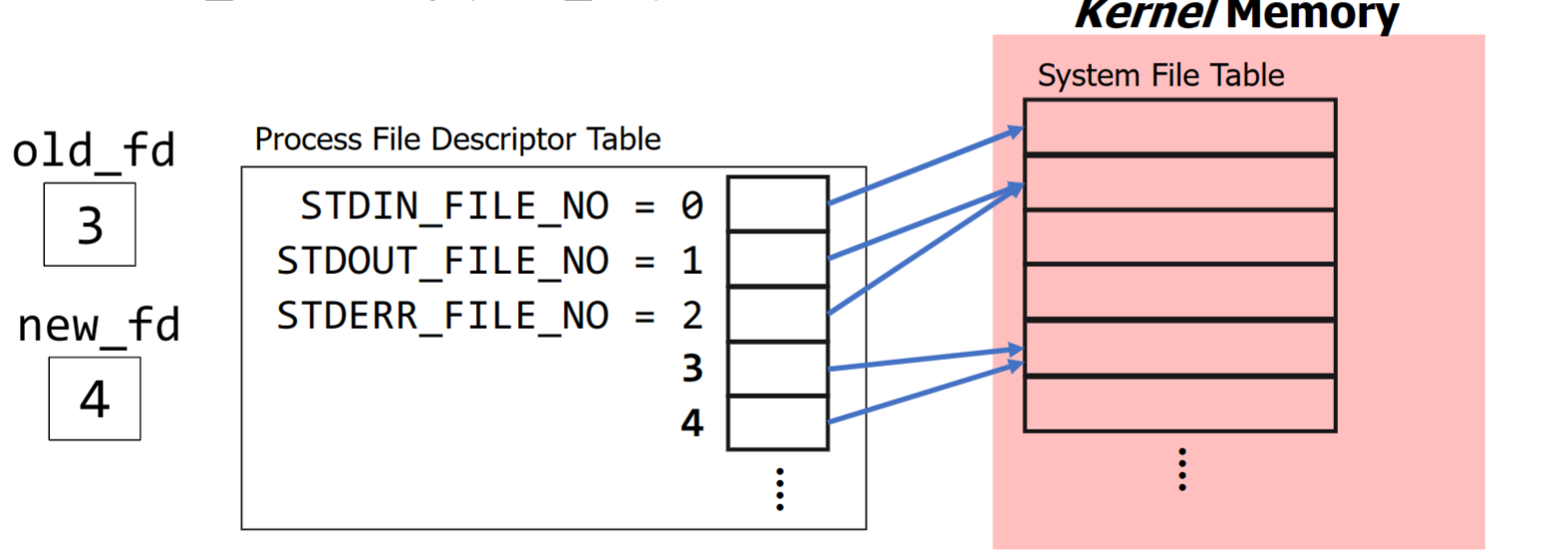
Last Updated:

File Systems

# You’ve been Dup’d!

* Duplicate a file descriptor: dup()

Int new\_fd = dup(old\_fd);



* A “targeted” version of dup() - dup2() - changes new\_fd into old\_fd

Int dup2(int old\_fd, int new\_fd)

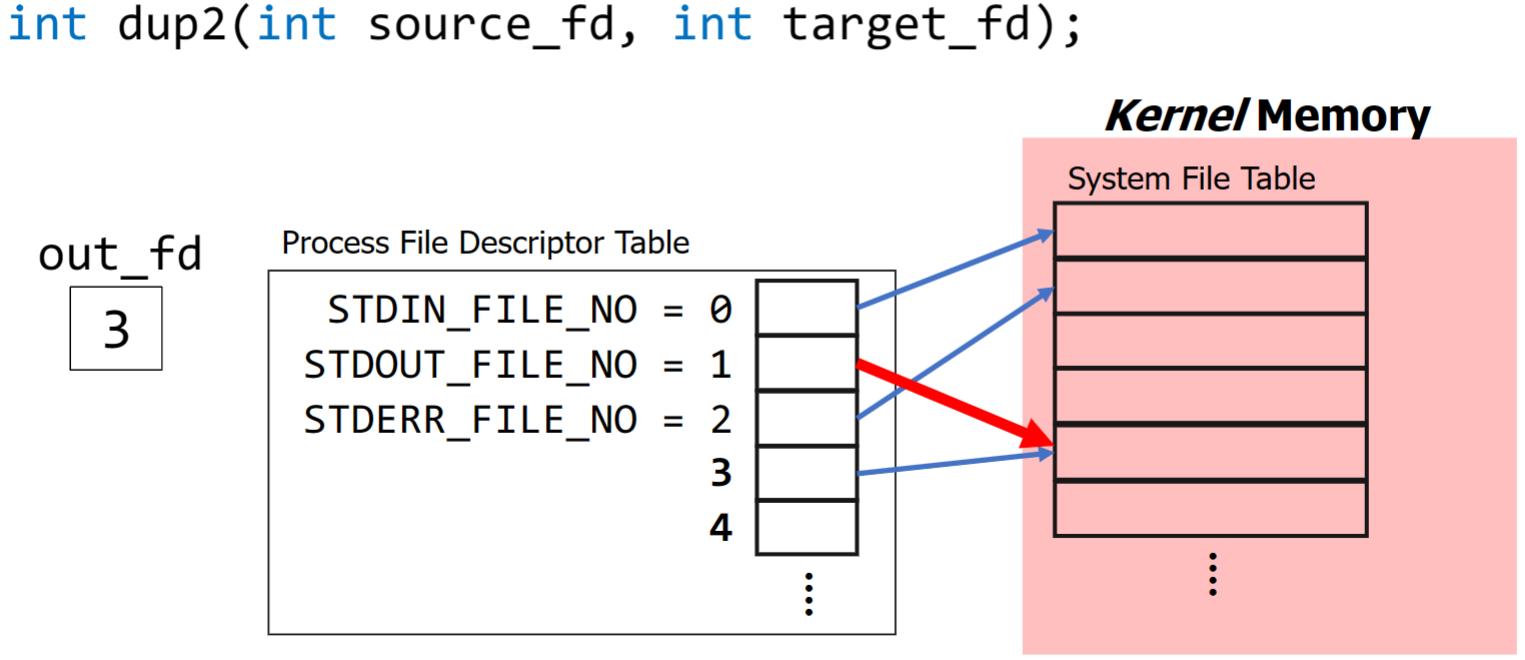
* + Automatically executes close(new\_fd) if it corresponds to an open file
  + Very useful in combination with fork()
    - And file descriptors survive a call to exec()

## Output Redirection

* + Example - write code that simulates behavior of the shell when redirecting output:

1. Run ls -l in a child process
2. Ensure that the output of ls -l goes to the file ls\_out.txt
3. The parent should print ‘Command Done’ once child is finished

* + Built-in file descriptors do not get special treatment- we use them in calls to dup() and dup2() like anything else
    - Ex: dup2(regular\_old\_fd, STDOUT\_FILE\_NO) redirects stdout to open file from regular\_old\_fd
      * The process believes that it is printing to the screen, really just appending to the file- both are essentially a write() system call to different destinations

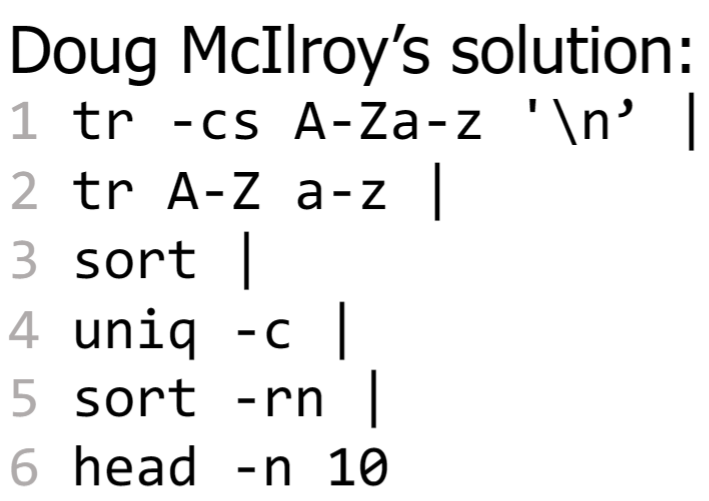


* + With this, we can change the destination of a program’s output **without making changes to our code**
    - One of the reasons why file descriptors survive an exec()
  + Taking this further:
    - Command line “pipes” get their name from the pipe OS feature (for interprocess communication) mentioned earlier
    - Example: ls -l | sort
      * Output (normally stdout) of ls -l is pipelined as input (normally stdin) to the sort command
    - Philosophy: combining small, sharply focused tools to achieve big things
    - How does this work? Simply put, the shell process forks children to run these commands, using pipes and dup2(), it allows ls program to communicate with sort program via printing

# All-Powerful Command Line

* Knuth’s problem: Read in a text file and print out the 10 most frequently used words in that file, sorted from most to least frequent
* Ilroy’s Explanation:

1. Put each word on its own line by converting the complements of the alphabet (-c) into newlines, squeezing repeated newlines (-s) into one
2. Convert uppercase to lowercase
3. Sort lines to bring identical words together
4. Replace each run of duplicate words with a single word and include the count (-c)
5. Sort in reverse (-r) numeric (-n) order
6. Print out the first 10 lines



## Useful Recipes & Combinations

* + - Figure out which subdirectories are taking up the most space (e.g., to free up room for your quota)

du -d -h . | sort -h -r

* + - Revert all files which have been modified since the last git commit:

git status | grep modified | awk ‘{print 2}’ | xargs git checkout

* + - Of course, their are easier ways to perform these actions, but this demonstrates the power that the command line holds

# Storage Devices

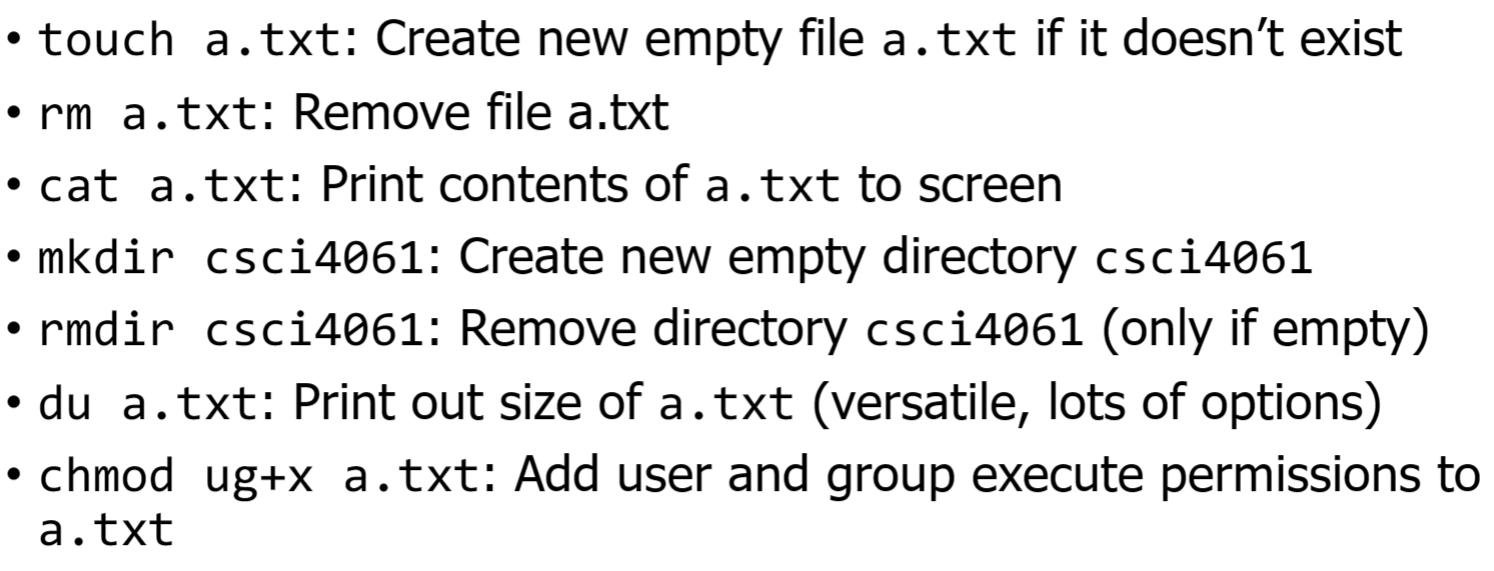
* This uniform interface is made possible by **device drivers**: software modules within the OS that are responsible for interacting with a specific hardware component
  + Example- Samsung SSD containing a microcontroller for reading/writing data; the OS has a device driver for this device that knows how to “talk to” the microcontroller to execute operations
* All storage device drivers offer the same interface to the rest of the OS
  + Read/Write data at block number i

## File Systems:

* + A **file system** is the OS component responsible for managing the contents of persistent storage (e.g., a hard drive or SSD) to offer convenient features to users
  + Reading a text file w/o a file system:
    - Have to remember the index of each disk blocks containing file’s contents
    - Read one block, jump to next block when end of first is hit, repeat
    - File blocks are not necessarily contiguous

### File systems are hierarchical

* + - As users, we never see disk blocks directly
    - We think about files, directories, etc.
    - Files on computers are organized within a hierarchy of directories
    - A directory may contain multiple files or additional nested directories
    - A **path** in this hierarchy identifies a unique file or directory
      * Nodes on a path are separated by a /
  + Commands for working with files



# File Paths & Directories

* Two ways to identify a specific file in a system:
  + **Absolute Path**: specifies how to reach a file from the root directory
    - Always starts with a leading ‘/’
  + **Relative Path**: specifies how to reach the file from the current location
    - Current location?
    - Every process has a working directory that determines the *starting* point for all relative path file lookups
    - The working directory of a child is inherited from the parent
    - Any path not specified with a leading ‘/’ is treated as a relative path

#### Two relevant system calls:

* + - * Get current directory:

char \*getcwd(char \*buf, size\_t size)

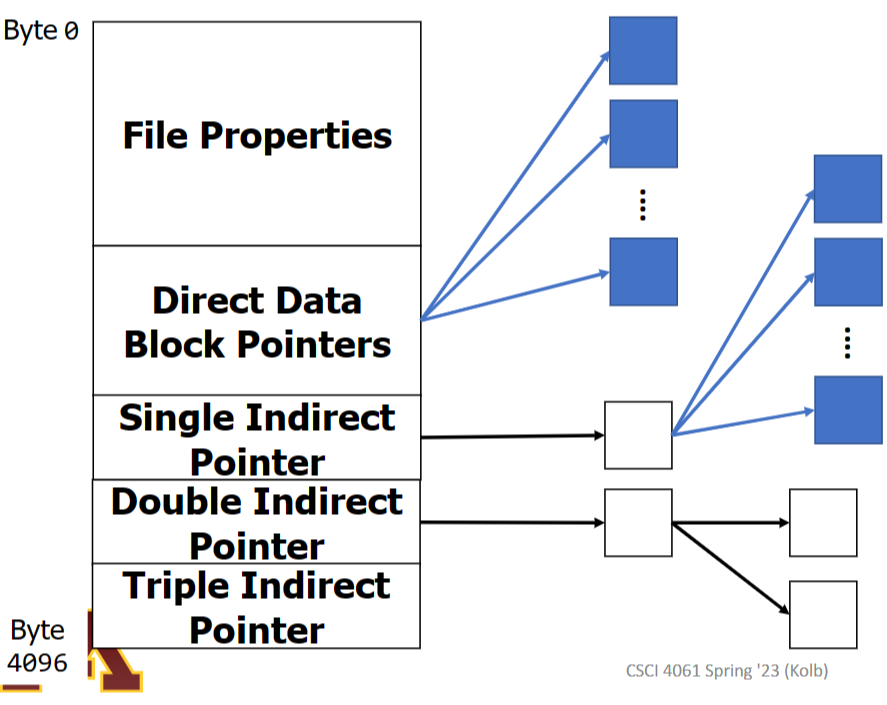
* + - * Change directory:

int chdir(char \*path)

* Home Directories
  + In a shell, we can use a ‘~’ as shorthand- this is specific to the shell and not C
  + In C, the home directory is stored in the HOME environment variable
    - Use getenv(“HOME”) in a C program to retrieve it

# Inodes & Metadata

* An **inode** (index node) is a data structure that stores metadata about a file
  + Metadata includes file size, access permissions, owner and group, as well as where the file lives (i.e. the disk block indexes that holds its contents)
* Structure:
  + “Pointer” here is a disk block index
  + Organized as a tree, disk blocks with file contents are leaves; as the file gets larger, more levels are added to tree to enable more leave nodes



* Accessing File Metadata
  + Use stat() to access info based on path (relative or absolute) or fstat() to access info based on a file descriptor

int stat(char \*path, struct stat \*stat\_buff)

int fstat(int fd, struct stat \*stat\_buff)

* + The file metadata will be placed into the specified struct