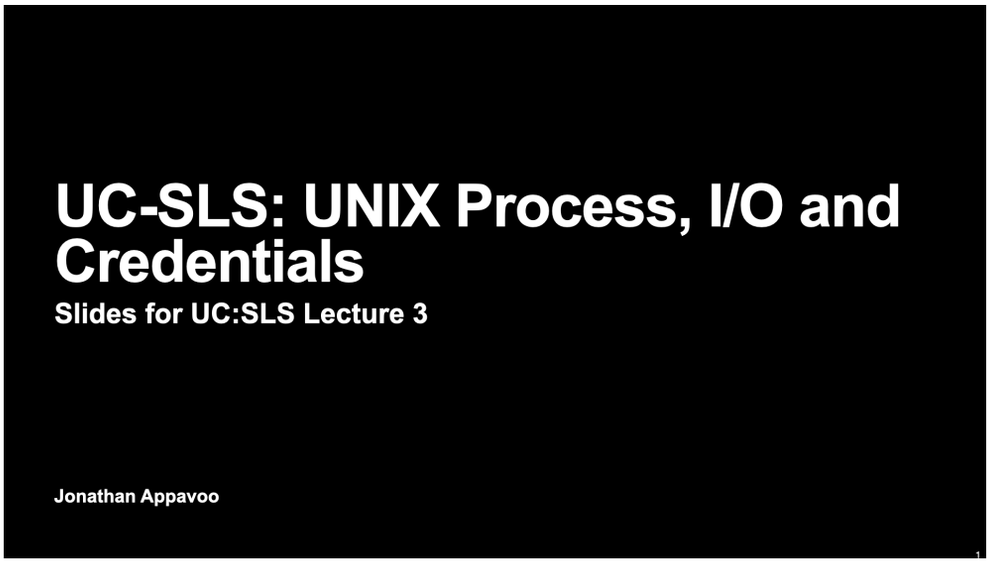
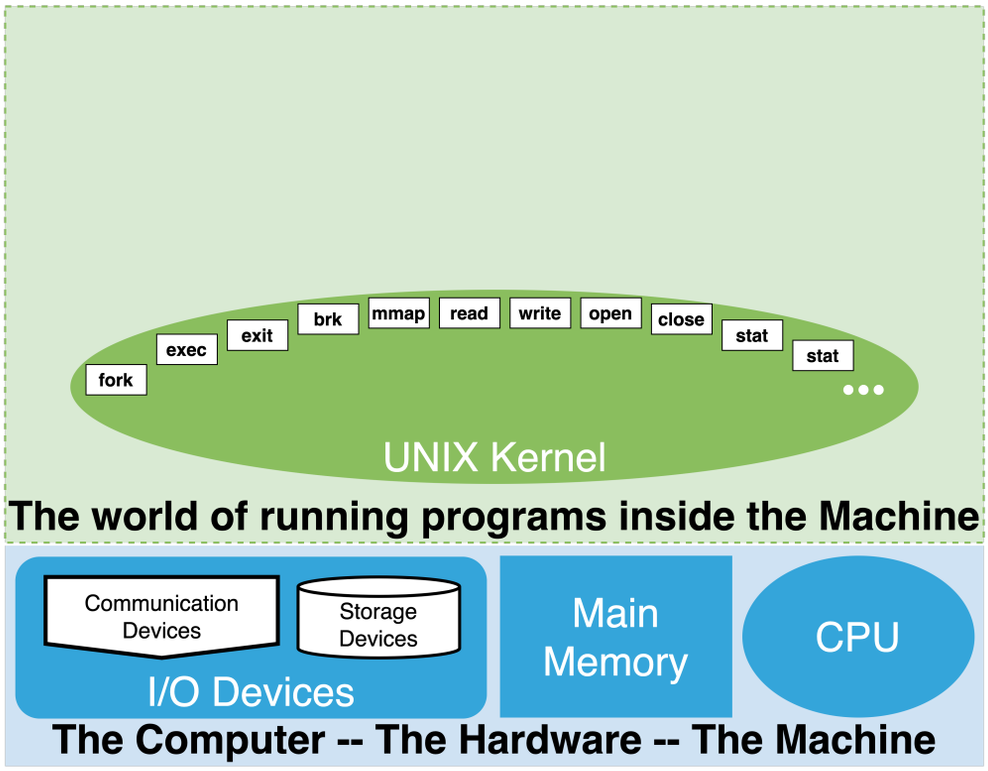
UNIX Process, I/O and Credentials

Contents

- 3.1. Processes, Files and Streams
- 3.2. Process management
- 3.3. Credentials and file permissions





3.1. Processes, Files and Streams

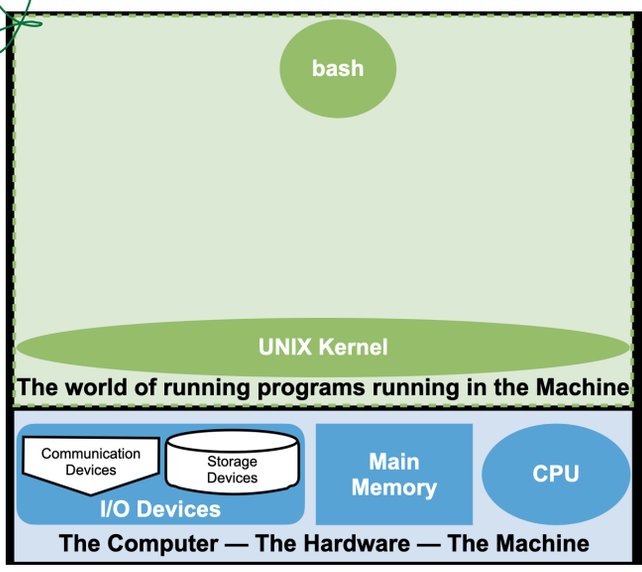
3.1.1. What is a process in more detail and what can it

do

- a running program
  - a process can launch other processes
    - bash is implemented to start commands as processes (when needed)
    - passes command line arguments and environment variables – a set of key value pairs Eg.
      - x=cool
      - HOME=/home/jovyan

Processes

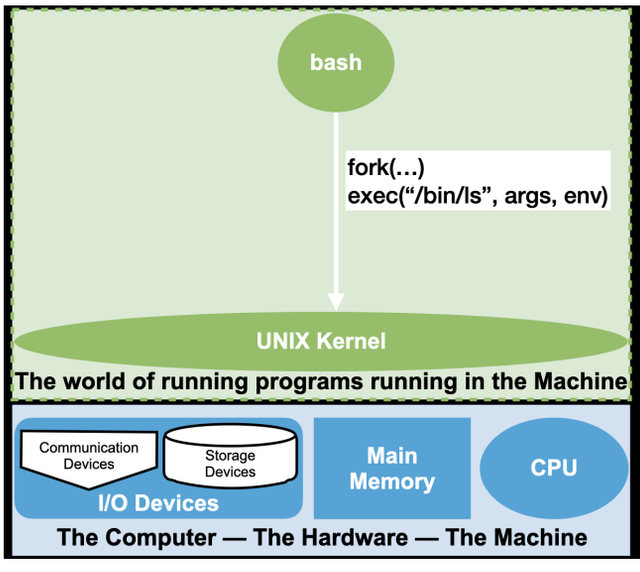
2



Processes

- A Running Executable
  - **Executable** : A file that is marked as executable and conforms to kernels loader format (Linux ELF file)
    - Or an ASCII file who's first line says what executable to run and pass the file as "**script**" of commands. eg. `#!/bin/bash`

3

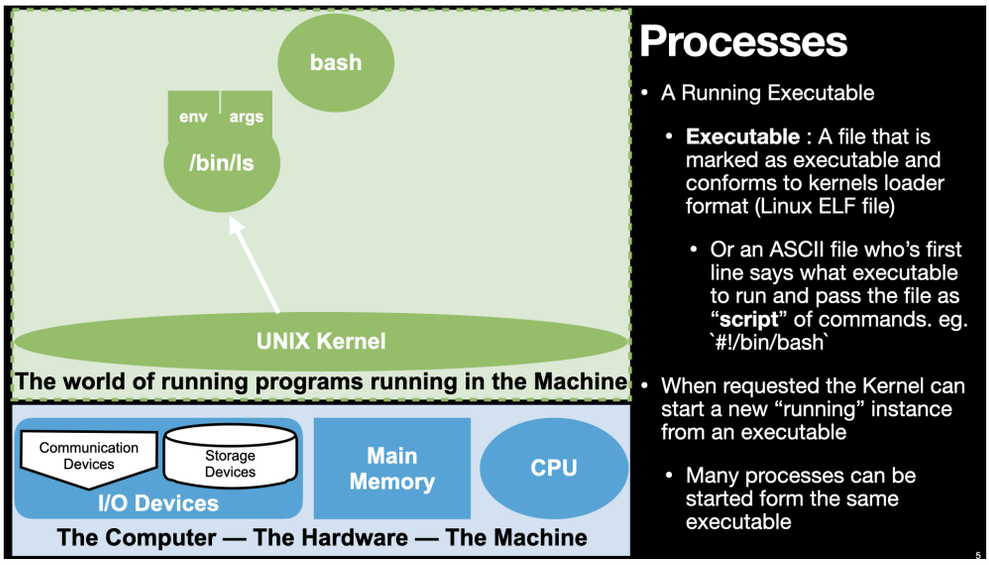


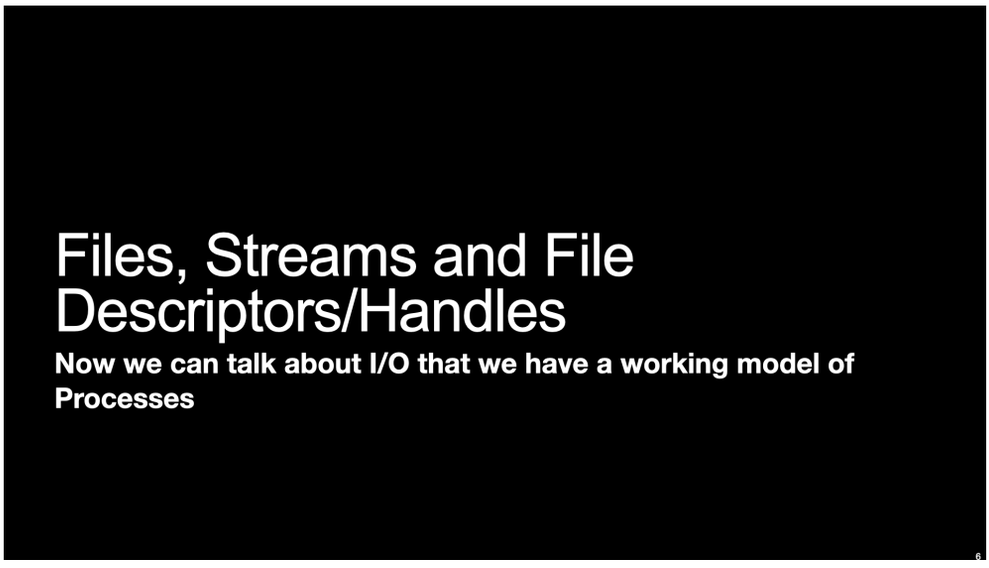
Processes

- A Running Executable
  - **Executable** : A file that is marked as executable and conforms to kernels loader format (Linux ELF file)
    - Or an ASCII file who's first line says what executable to run and pass the file as "**script**" of commands. eg. `#!/bin/bash`
- When requested the Kernel can start a new "running" instance from an executable
  - Many processes can be started form the same executable

4

ls -a → all of the file in my home directory.





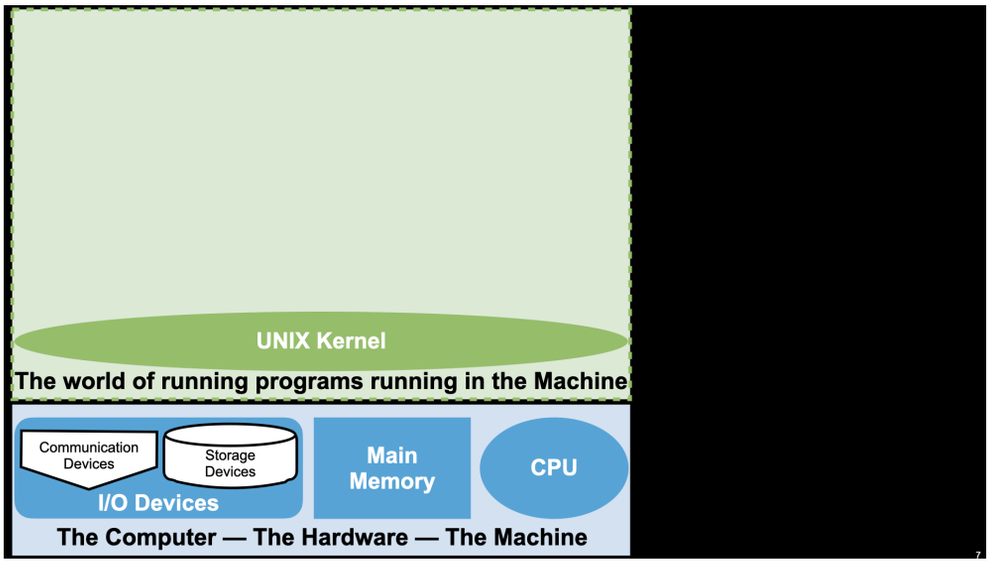
ls /sys/dev
↓
IO devices

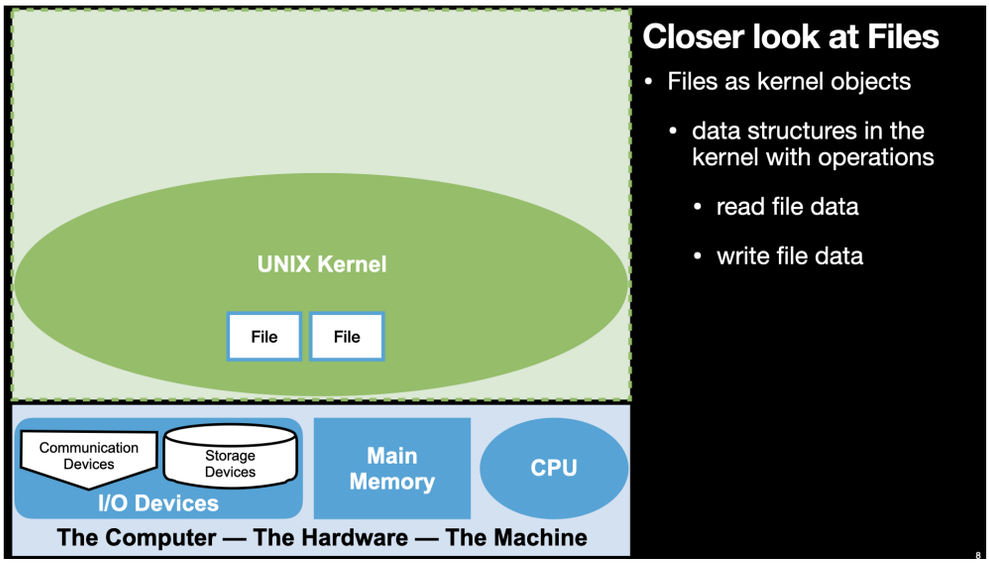
ls /dev
↳
device recognize ls file

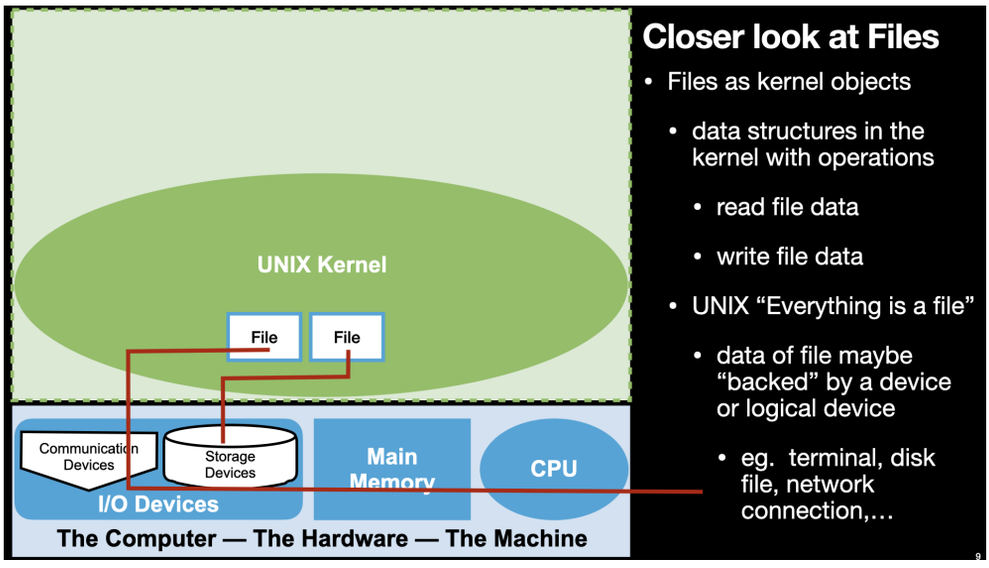
3.1.2. Processes, Files and channel/stream model of I/O

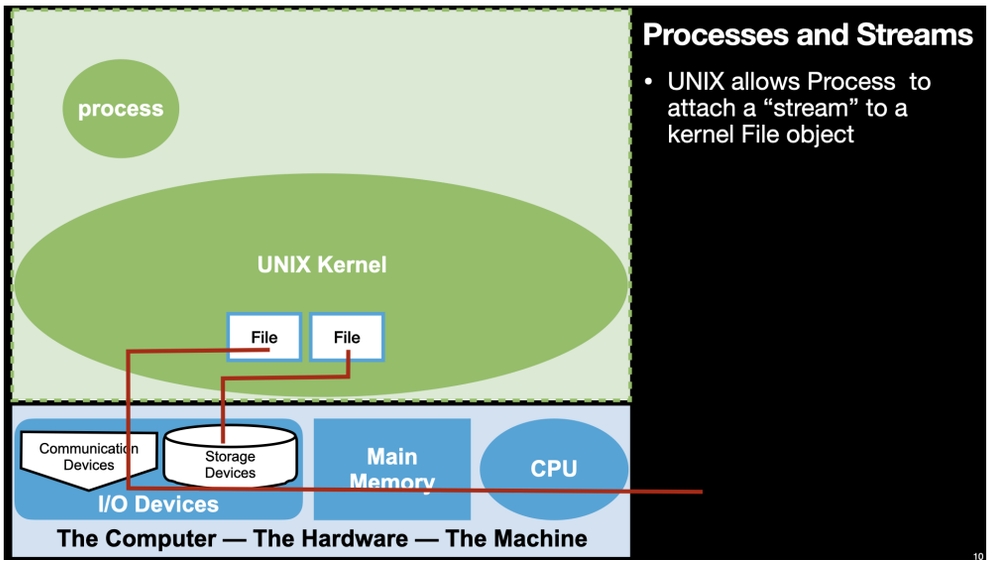
3.1.2.1. Files and Kernel objects

- read, write
- everything is a file





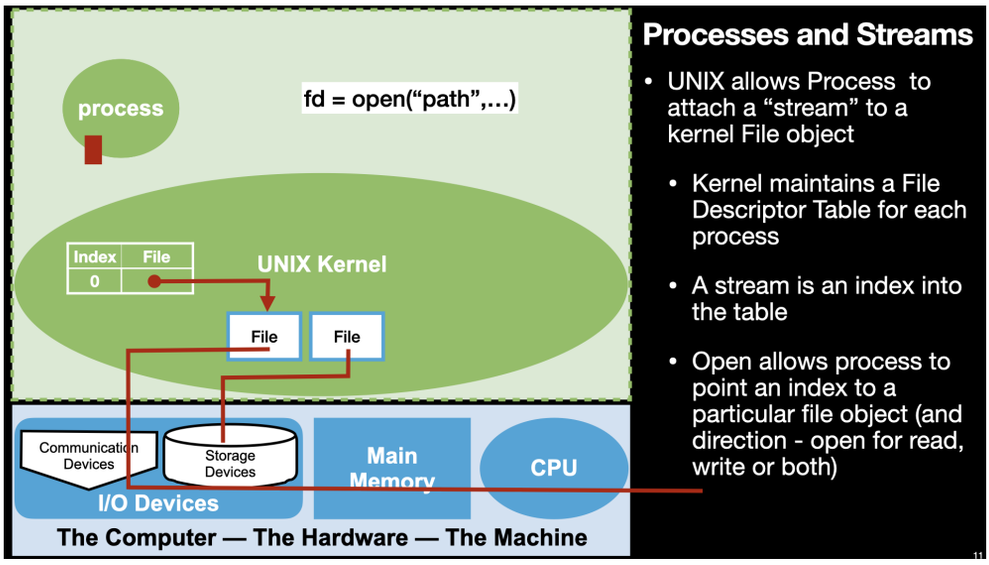




3.1.3. Streams: Processes and Files

- open : attach a file as a stream
- file descriptors/handles : stream
  - read and write bytes to a stream
  - close
  - dup
- standard input, standard output, standard error

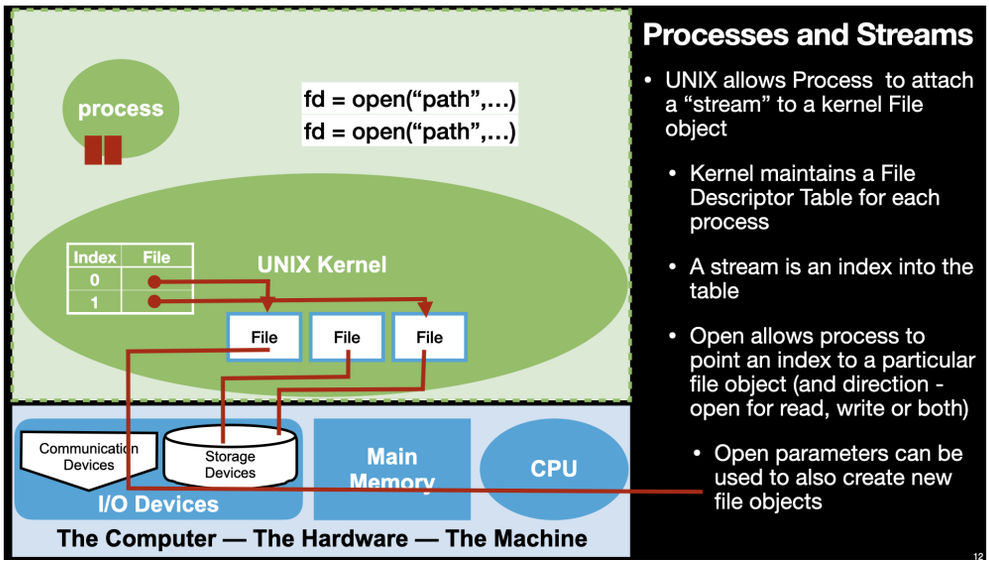
↓ ↳
output
normally terminal



Processes and Streams

- UNIX allows Process to attach a "stream" to a kernel File object
  - Kernel maintains a File Descriptor Table for each process
  - A stream is an index into the table
  - Open allows process to point an index to a particular file object (and direction - open for read, write or both)

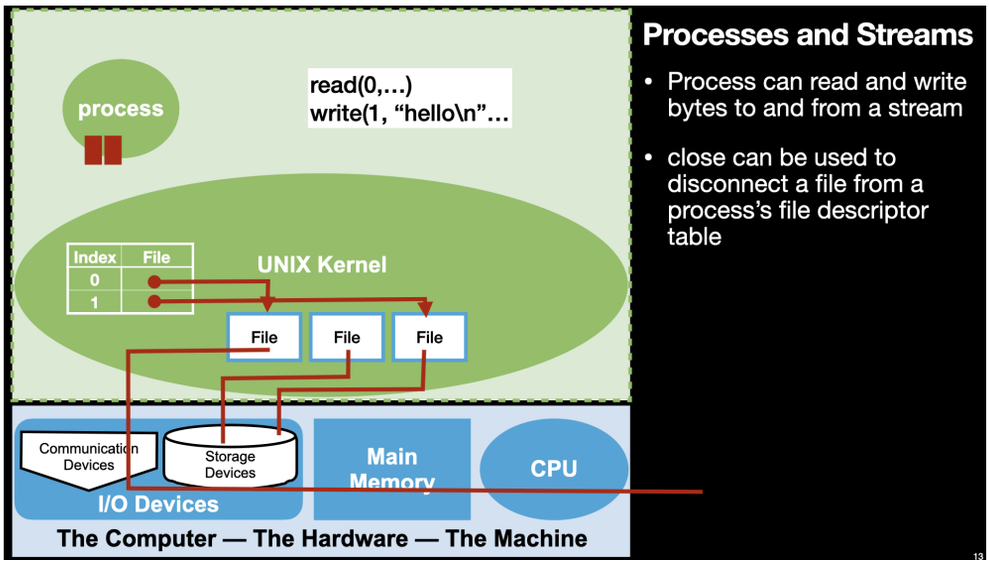
11



Processes and Streams

- UNIX allows Process to attach a "stream" to a kernel File object
  - Kernel maintains a File Descriptor Table for each process
  - A stream is an index into the table
  - Open allows process to point an index to a particular file object (and direction - open for read, write or both)
    - Open parameters can be used to also create new file objects

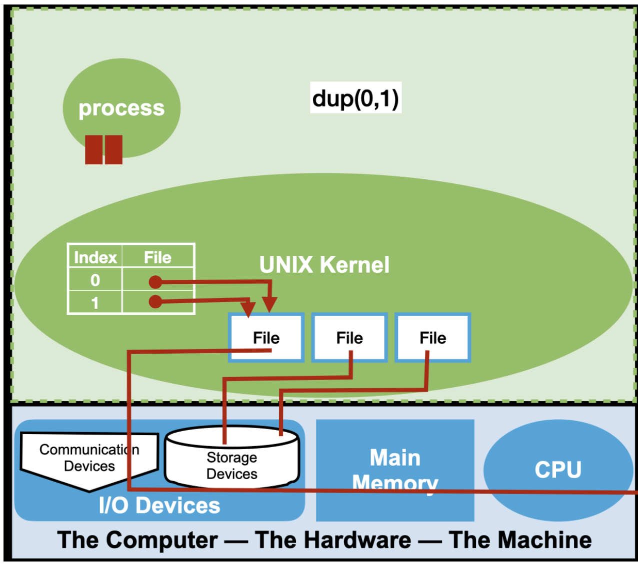
12



Processes and Streams

- Process can read and write bytes to and from a stream
- close can be used to disconnect a file from a process's file descriptor table

13



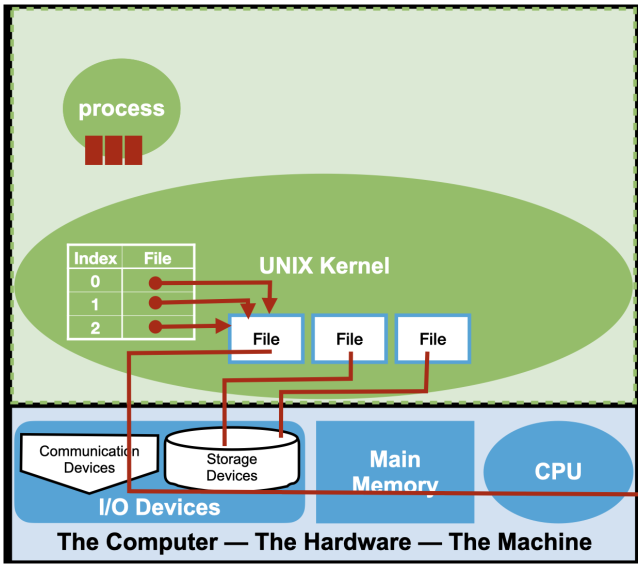
Processes and Streams

- Process can read and write bytes to and from a stream
- close can be used to disconnect a file from a process's file descriptor table
- dup and dup2 can be used to duplicate a one index to another
  - now both fd indexes point to the same file object

14

read/write

ls-la



UNIX Standard Streams

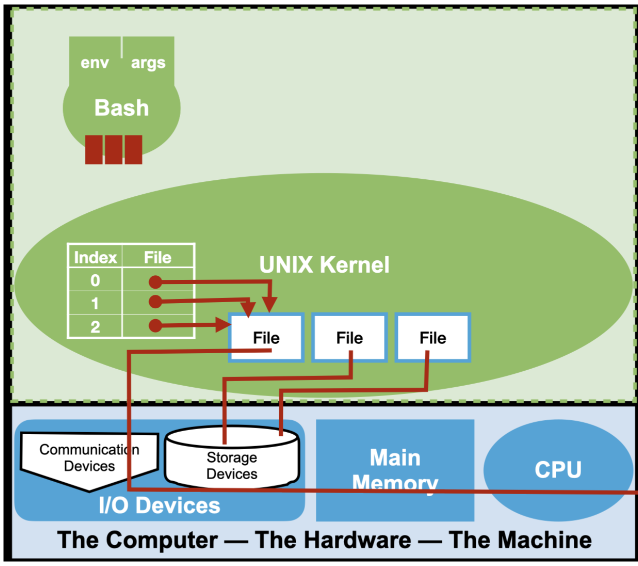
- UNIX has three special streams
  - STanDard IN - 0
  - STD OUT - 1
  - STD ERRor - 2
- By convention standard routines for reading input, writing output and writing errors use these
- Therefore what 0,1, and 2 point too, directs where a process's standard input, output and error will come from and go to

15

ls -la >> home-files

standard out → home-files

standard in으로 들어간다.



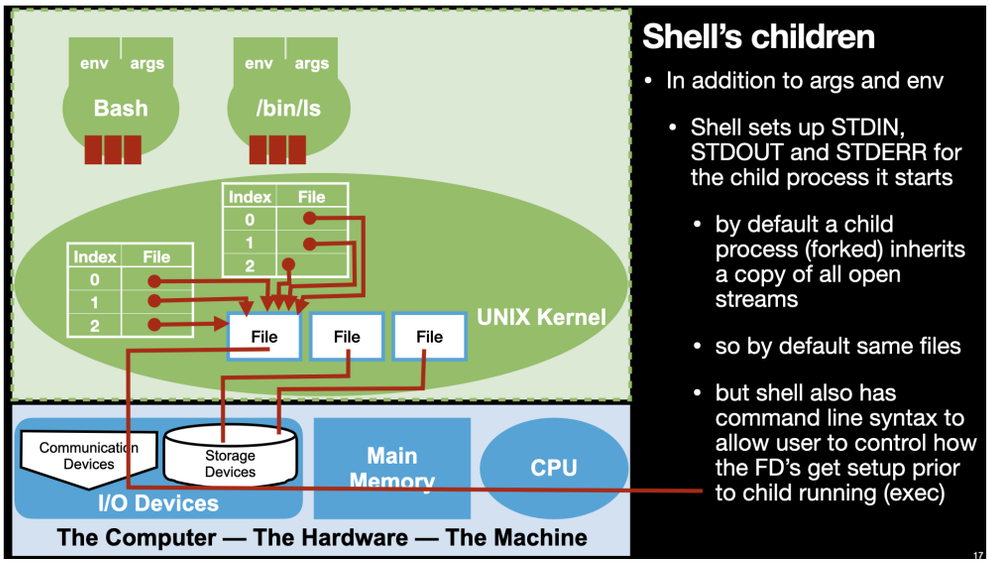
Interactive Shell Process

- UNIX when it starts a shell process for a terminal connection it
  - setups the initial environment variables
  - passes the arguments,
  - setups up the STDIN, STDOUT and STDERR pointing to a file that represents the terminal connection
    - `tty` command tells you if STDIN is a tty what its tty file is
    - `/dev/tty` is alias to make file easier

16

tty

ps → shows process running on the operator.



Shell I/O redirection - Syntax

Shell provides syntax to make use of many kernel features without having to write code or know the details

18

3.1.4. Shell Syntax:

- `echo 'Hello world' > hello`
- `cat hello`
- `cat < ./hello`

We will stick to the basics

Lets run through some examples

- `> path` `# redirect fd 1 stdout`
- `> path1 2> path2` `# redirect fd 1 stdout and fd 2 stderr independently`
- `> path 2>&1` `# redirect fd 1 stdout and then dup fd 1 to fd 2 stderr`
- `>> path` `# redirect fd 1 but open for append`
- `< path` `# redirect fd 0 stdin`

But there is syntax for much more — open, close, dup and "special" files, here documents,…

Remember because every thing is a file redirection is very powerful (eg redirecting to a tty file)

19

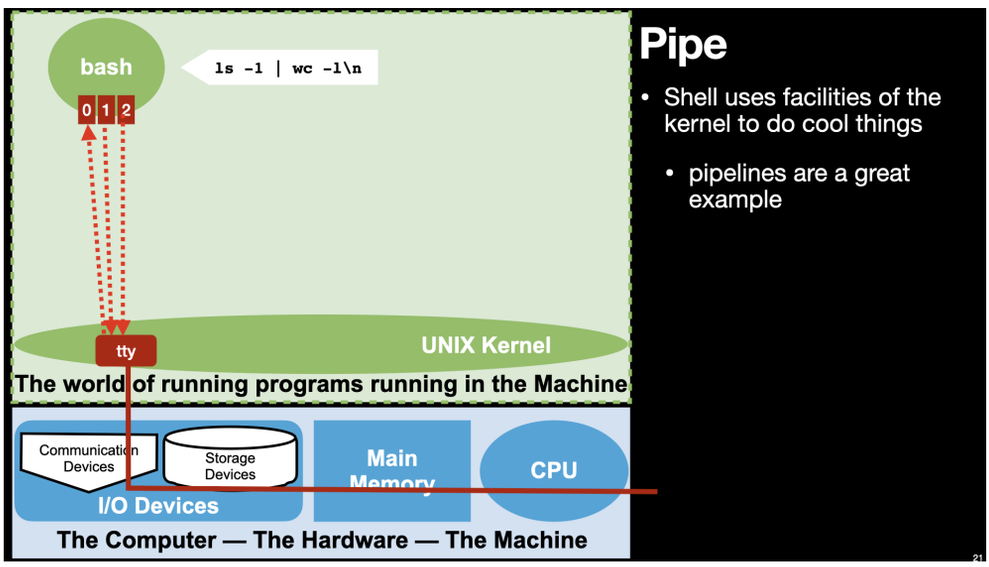
3.1.5. We now can understand what a pipe is

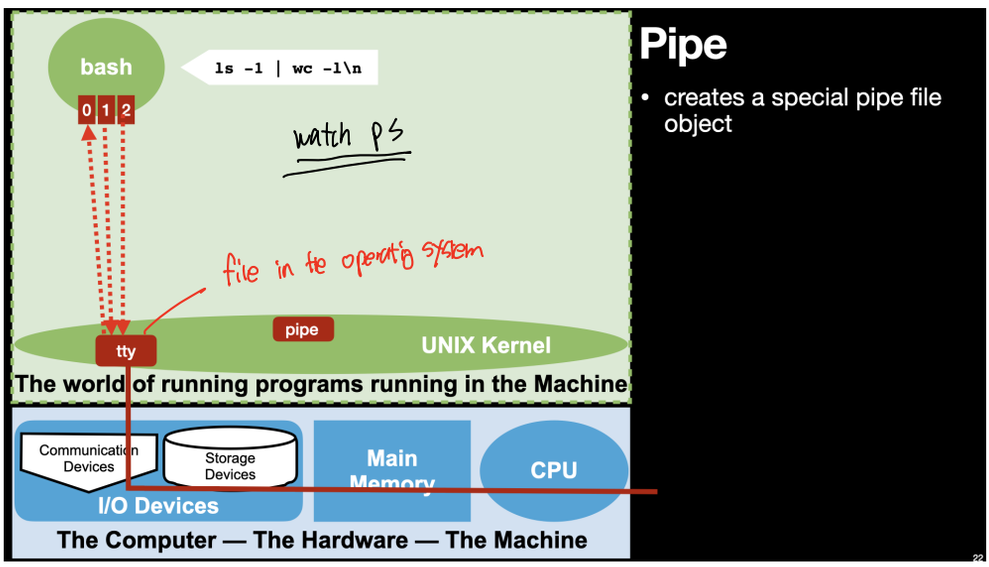
- pipe file object
  - Process 1 stdout into pipe and Process 2 stdin from pipe

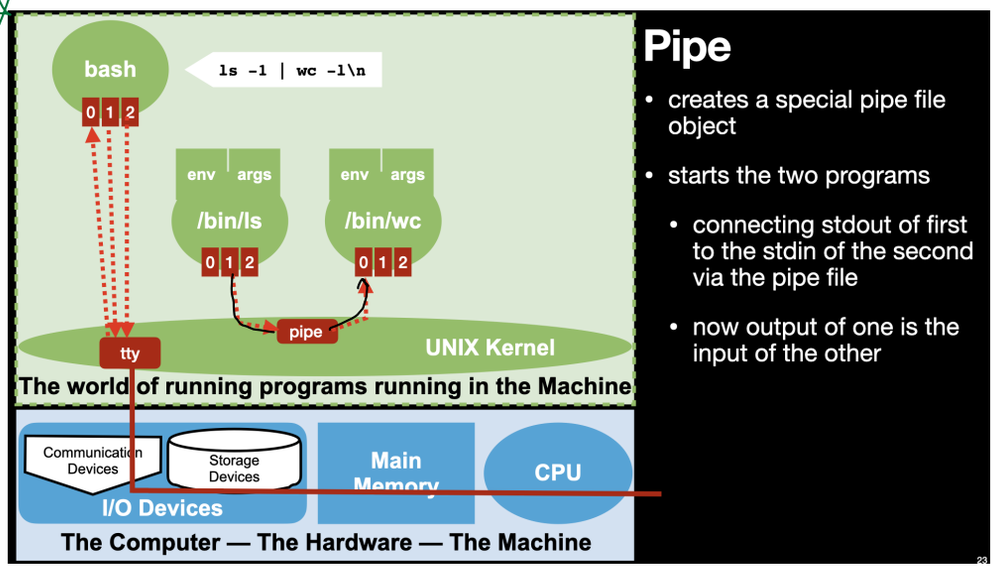
3.1.6. Shell Syntax:

- `ls -1 | wc -l`
- `ls -1 | grep '^l*' | wc -l`
- `mknod mypipe p` and `mkfifo mypipe`

mknod : make pipe

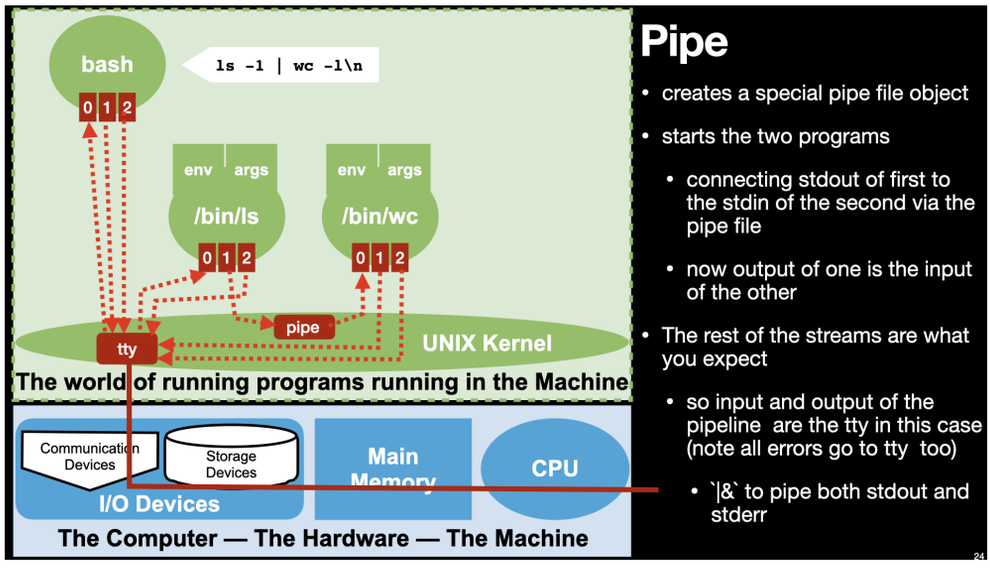






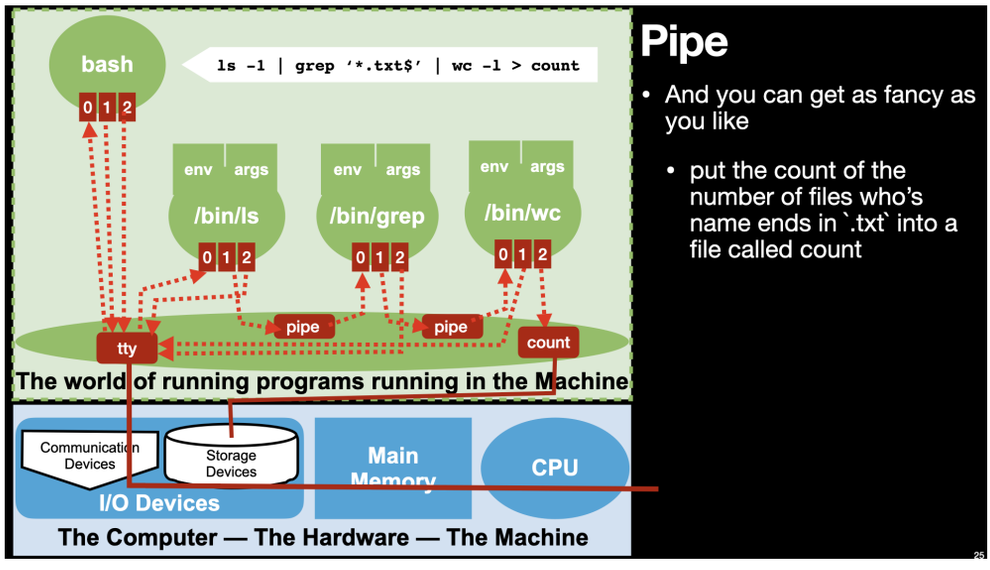
Pipe

- creates a special pipe file object
- starts the two programs
  - connecting stdout of first to the stdin of the second via the pipe file
  - now output of one is the input of the other



Pipe

- creates a special pipe file object
- starts the two programs
  - connecting stdout of first to the stdin of the second via the pipe file
  - now output of one is the input of the other
- The rest of the streams are what you expect
  - so input and output of the pipeline are the tty in this case (note all errors go to tty too)
    - `|&` to pipe both stdout and stderr



Pipe

- And you can get as fancy as you like
  - put the count of the number of files who's name ends in `.txt` into a file called count

Named Pipes

Super useful idiom when automating things

- `mkfifio <path>` or `mknod <path> p` - creates a pipe file object that is in the file system with name <path>
- You can now connect arbitrary processes to each other via the named pipe
  - eg. from ledSrv

```
#!/bin/bash
set -x
INPIPE=~/.ledsin
if [[ ! -p $INPIPE ]]; then
  [[ -e $INPIPE ]] && rm $INPIPE
  if ! mkfifo $INPIPE; then
    echo ERROR could not mkfifo $INPIPE
    exit -1
  fi
fi

tail -f $INPIPE | ./flashy

rm $INPIPE
```

26

Process Management

27

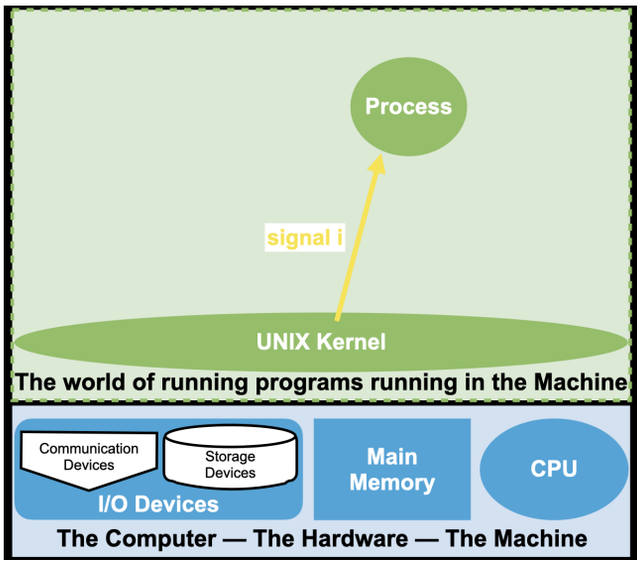
3.2. Process management

- `ps` - look at all processes
- The shell and its children
  - `&` : foreground and background
  - `ctrl-z`
  - `jobs`
- so we know how to start process, list them how about stopping
  - `kill` kill -9
  - signals watch command
    - `ctrl-c`
  - or without prejudice

PS - Process Status

- The text equivalent to our running process diagrams
  - `'man ps'` Lots and lots of options
  - '`ps`' — defaults to process with same tty and user
    - pid, tty, cmd
  - '`ps auxgww`' (bsd options to all process) https://en.wikipedia.org/wiki/The_Cuckoo%27s_Egg_(book)
    - USER what user is associated with the process (more later)
    - information about processes usage of the computer: %CPU, %MEM, VSZ, RSS
    - State the process is in STAT (including zombie)
    - times: START: when the process was started TIME: cumulative used
  - Processes form a tree of parents and children: PPID, `f` will "draw" as tree in ASCII
  - Note we can get to see both the arguments and using 'e' even the environment that was passed in

28

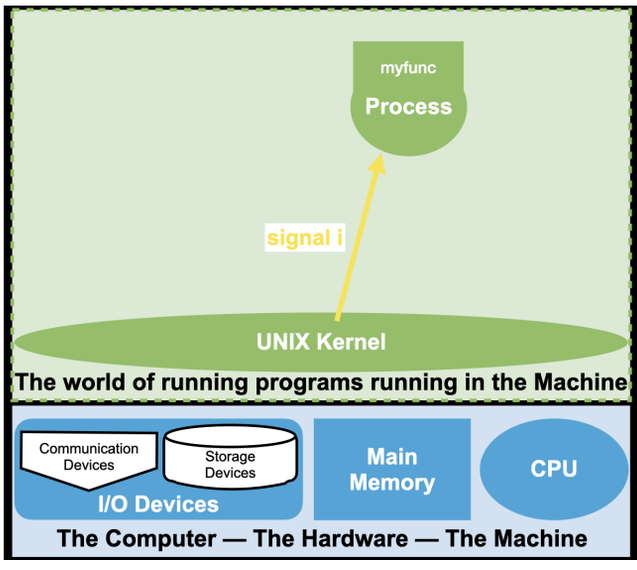


Signals

- Kernel provides the ability to send a "signal" to process
- There are several types of signals — each having unique number

pid: process number.

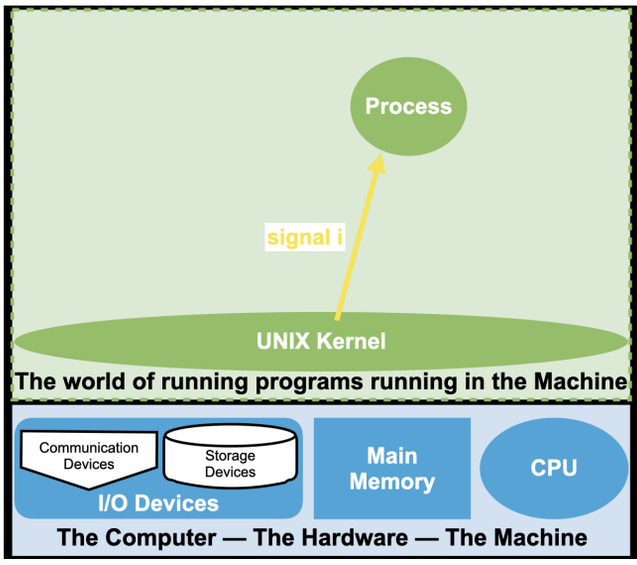
29



Signals

- Kernel provides the ability to send a "signal" to process
- There are several types of signals — each having unique number
- Programs can setup functions within themselves to "handle" a particular signal if it is sent to them
  - Eg. when I get signal i invoked `myfunc`
- there is default behavior defined for each signal
  - eg. terminate, stop, ignore, continue

30



Signals

- Kernel provides the ability to send a “signal” to process
- There are several types of signals — each having unique number
- Programs can setup functions within themselves to “handle” a particular signal if it is sent to them
  - Eg. when I get signal i invoked `myfunc`
- there is default behavior defined for each signal
  - eg. terminate, stop, ignore, continue

31

3 common uses of signals

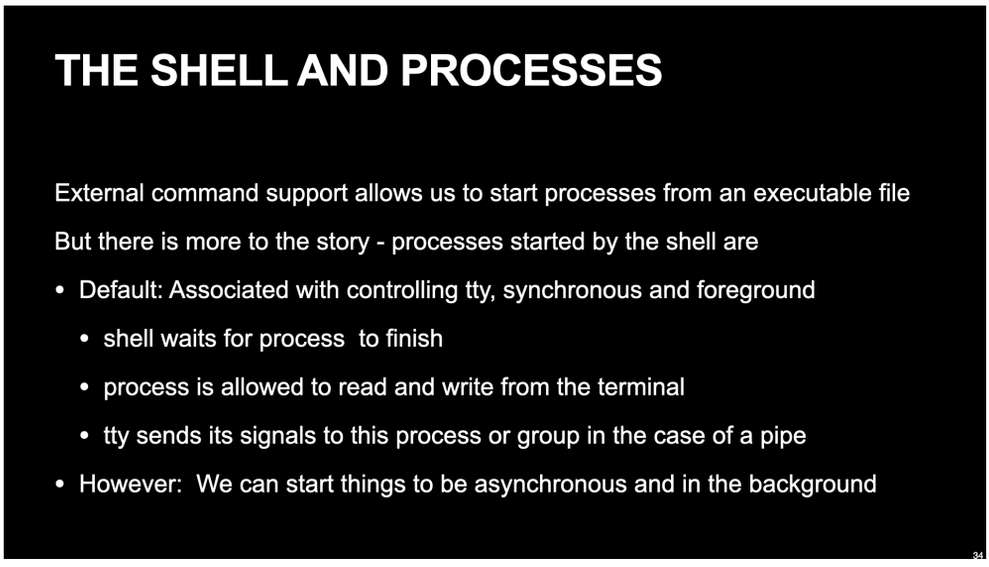
1. Kernel wants to inform process of some event signal
   1. A child process of has terminated
   2. attempted to access invalid memory
   3. divide by zero
2. Have terminal generate signals when a certain key-sequence it pressed
   1. ctrl-c — send the interrupt signal (typical default behavior is to terminate)
   2. ctrl-z — send a please stop executing (pause)
3. One process sends another process a signal
   1. `kill` command allows user to send signals to another process: `man kill`
      1. kill -L
      2. default is the nice kind of kill SIGTERM - TERMINATE (allows process to catch this and cleanup — save files etc)
      3. however can also use it to send the NOT nice SIGKILL — Kernel handles this one and SIGSTOP

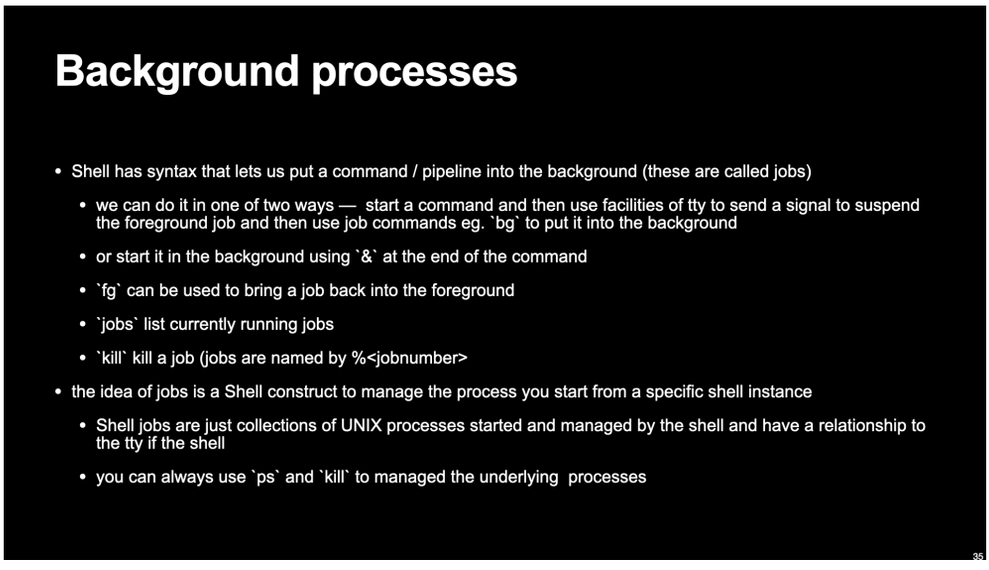
32

lets play with signals

- `bash -c ‘for ((i=0; i<100; i++)); do echo “$$: $i”; sleep 2; done’`
  - What does this do?
  - default starts

33

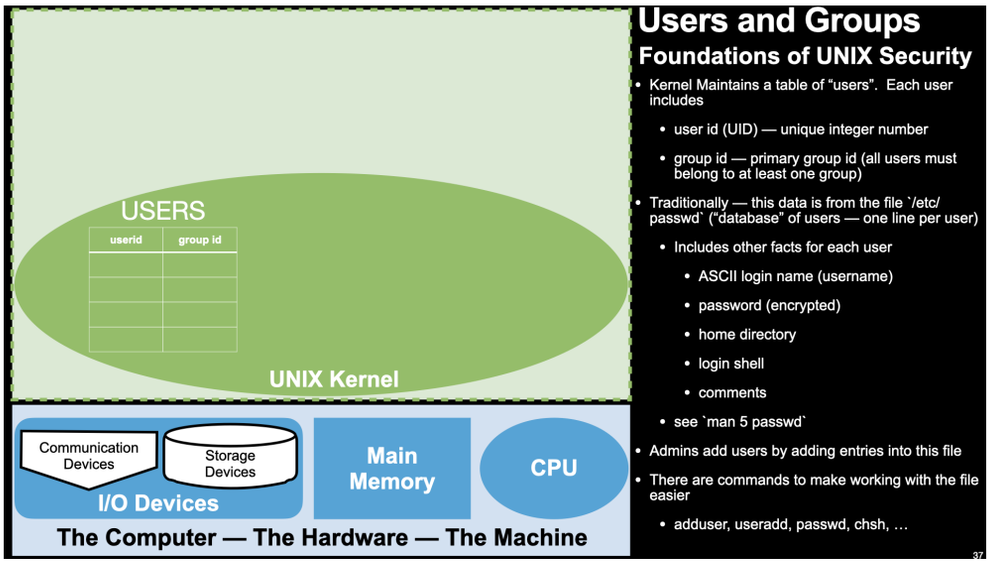


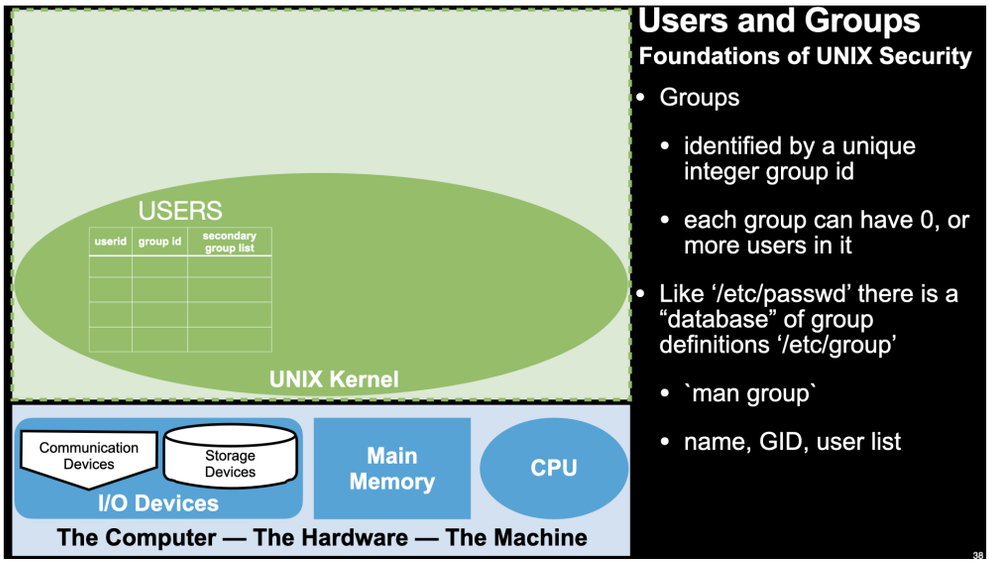


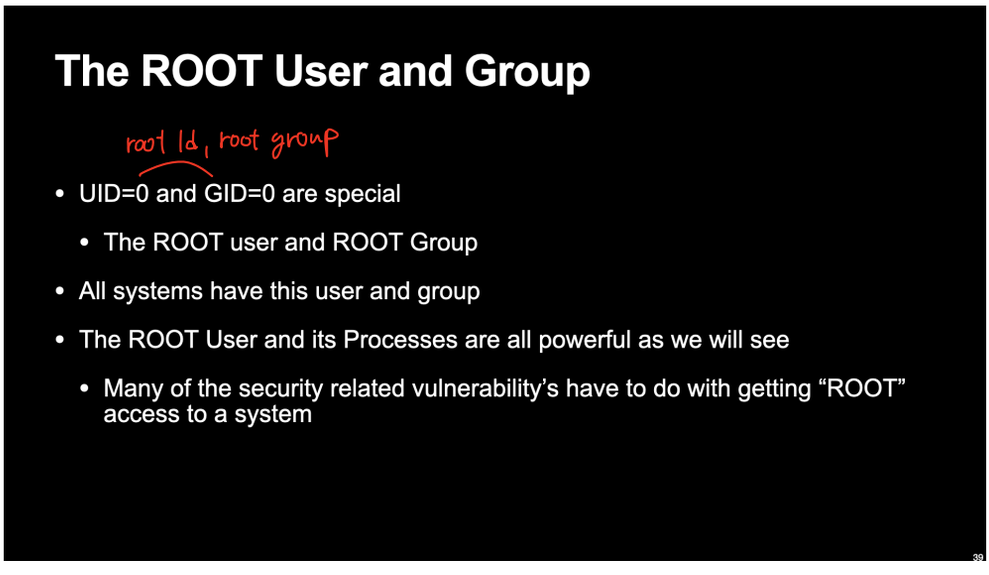
3.3. Credentials and file permissions

- Process have id's associated with them
  - a single user id : `id`
    - a single number that maps to a string user name (`/etc/passwd`)
  - set of group ids
    - user has a primary group but can be in many secondary groups
    - each has a number that maps to a name (`/etc/group`)
    - each group can have many users

- `ps auxgww`
  - process's inherit their ids from their parent





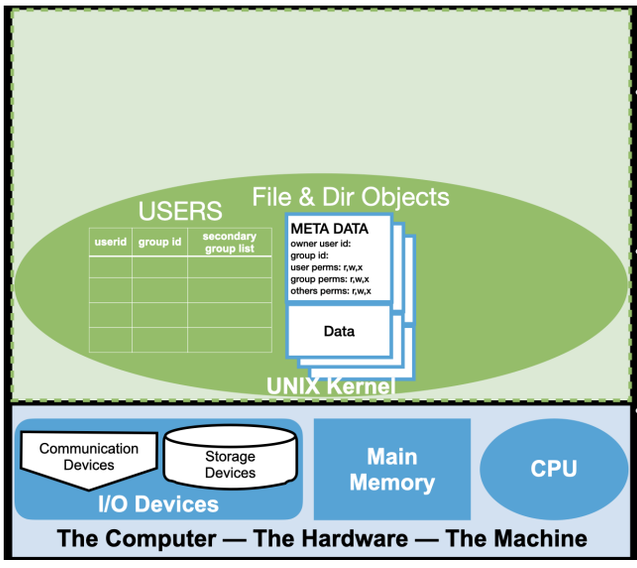


Users and Groups

Shell commands

- There are commands so that you can see facts about a user and group
  - `man id`
    - `id -a` lists a users ids numeric and names
  - `man groups`
    - lists a users group names
- There are standard Shell Variables too:
  - echo $HOME # path of home directory
  - echo $USER # user login name

40



Files/Directories, ids and Permissions

- Every file/dir object has two ids as part of its Meta Data
  - Owning User : UID
    - The user who owns the file by default the user that created it
  - Group ID : GID
- Every File/Dir object has THREE sets of permissions
  - USER: What the user is allowed to do
  - GROUP: other users within the group of the
  - OTHERS: what users other than the owner or those in the group can do
- Basic permissions are a vector of 3 bits: Read, Write, eXecute (there are some others)
  - Read: allowed to open for read
  - Write: allowed to open for write
  - Execute: allowed to start a process from the file

41

- Files have id's and permissions `ls -l`, `chmod`, `chown`, and `chgrp`
  - user, group, other -> read, write, execute
  - kernel ensures that process id's and requested operations match permissions

File/Directory ids and permission

Shell commands

- `ls -l` : long listing… displays ids and permissions
- `chmod` : change mode bits of the file or directory
  - `man chmod`
- `chown`: change ownership of the file or directory
- `chgrp`: change group of a file

42

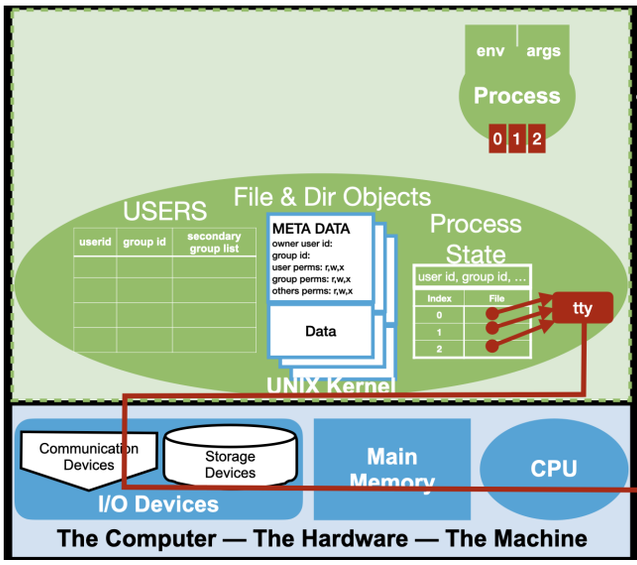
File vs Directory meaning of Permissions

How the permission bits are interpreted differs for Files vs Directories

- For files it is obvious — Read contents, write contents, can ask kernel to execute
  - note you can have execute permissions without read or write for a binary
  - script must have read and execute
- For directories things are more subtle :
  - Read - can read directory. Eg. `ls`
  - Write - add or modify entries : eg. `touch`, `mv`, `rm`
  - Execute - does not mean execute — sometimes called search… you are allowed to access entries of the directory if you know them

The above permissions and the user, group and other categories can be used in very creative ways. In addition to R,W,X bits there are 3 more set-uid, set-gid, sticky — see chmod for more details.

43



Processes have id's too!

- Process State: Like the File Descriptor Table the Kernel maintain other facts/state for each process
  - This includes an owning UID and GID
  - When a process attempts an operation such as opening a file for read the process ids are checked against the target Files permissions

44

Process state

A slightly more expanded view

- IDS — inherited from parent — so shell passes along to children (fork)
- CURRENT WORKING DIRECTORY — inherited from parent - so shell passes along to children (fork)
  - implications — if you `cd` in the shell you and then run a external command it's working directory will be the one you lasted set
- ARGS and ENV — set by shell when it asks kernel to run the new binary (exec)
  - Args parsed form the command line
  - ENV will be the set of current environment variable and their values at the time of launch by the shell
- More advanced is controlling tty, session leader, group leader but shell is responsible for setting them up

Much of what we do in the shell and its syntax is built on a basic understanding of wha constitutes a process and its state

45

Summary

CORE POINTS:

Processes: have ids that include a UID, GID and set of supplementary group ids. Ids are inherit from parent

Files: have two ids (UID, GID) and 3 sets of permission bits (User:RWX, Group:RWX and Other: RWX)

Checking: Broken into two parts 1) Validate Path and 2) Validate Operation (open for read, write or execute) on the path

Part 1. Kernel walks the path that is being operated on and confirms that the necessary components of the path exist (confirming that the process has the necessary permission to search each directory along the path)

Part 2. Kernel verifies that a process has the necessary rights to perform the requested operby examining the ids of the process against the permissions of the path as follows:

1.if File UID == process UID then check request against file user permissions if ok then do operation and done

2.else if File GID == Process GID or if File GID is in the Process set of supplementary group ids then check request agains file group permissions if ok then do operation and done

3.else check request against the file other permissions if ok then do operation and done

4.fail operation

46

We have covered the basics of Credentials

- We have looked at the basics that should let you have an understanding for what is going on.
  - Process id's are a bit more complicated as there are various other ids that allow finer gain control and the ability for a process to switch or impersonate another user or group
  - for the details see manual pages at the end of the lecture

47

The Story has evolved

Foundations of UNIX Security Model

- What we have covered is actually the traditional UNIX security primitives and model
- Things have evolved and there are newer more complex finer grain security mechanisms in modern UNIX systems. Including
  - Capabilities — see `man capabilities`
  - Access Control Lists — see `man acl`
- But what we have covered is still very much the foundation of what is used and enough for us to understand the basic ideas.

48

Relevant man pages

That are now hopefully easier to understand

- `man credentials`
- `man path_resolution`

49

- Lets put all this knowledge to use:
  - find out what programs a particular user is running
- Write a string to a different terminal window
- Do an operation on every file in this directory
- Do an operation on every file in this and all subdirectories
- Search for a string in all files

50

By Jonathan Appavoo