

2.1 Command Line: Shell

INFO1112

Bob Kummerfeld

User Interface - the Command Line

Originally, systems only had simple character input and display - no windows, no mouse. To use the system you typed characters that were interpreted by a special program called a command interpreter or (in Unix) **the shell**.

The shell analysed the character string and ran the program you specified. Output was characters only on a screen.

Note that the **shell** is a program that anyone could write.

1990's



1970's



The Unix Shell

Since it is relatively easy to write a shell command line interpreter, many have been developed over the years.

The original was simply called "sh" and still available, but in this course we will use one called "bash".

The name comes from "Bourne Again Shell" since bash is a clone of a shell written at Bell Labs by Steve Bourne.

There are many others: csh (get it...), ksh, zsh,

Here are some the shell commands for working with files that you saw in first semester:

cd	Change directory
mkdir	Make directories
ls	List directory contents
rmdir	Remove directory
rm	Remove
mv	Move (also used to rename files)
cp	Copy
pwd	Print Working Directory

And here are some special names for elements of the name space (file system):

..	Parent Directory	cd ..
.	Current Directory	./program
-	Previous directory	cd -
~	Home directory	cd
/	Root directory	cd /

More Advanced Shell

The shell commands you learnt so far allow you to move around the file system, look at files, rename and delete files. But the shell, in combination with simple application programs, is far more.

The shell is a sophisticated programming language with control structures and variables, just like Python.

When combined with simple text processing applications/commands it can do very powerful tasks.

You should consider using a shell program rather than write a python program for any simple text processing task.

Shell Variables

The shell has variables like conventional programming languages. To give a variable a value we use an assignment statement:

```
$ NAME=bob  
$ Note: no spaces around the = sign!
```

We retrieve the value of a variable by putting \$ at the beginning:

```
$ echo $NAME  
bob  
$ echo is a command that prints its arguments
```

Shell variables can only hold strings of characters but this still makes them very useful. For example, a variable could contain a file name:

```
$ FILE=fred
```

```
$ cat $FILE
```

```
some text in file called fred
```

```
$
```

The way this works is that the shell replaces the variable name `$NAME` by the string it contains. In the example, `$FILE` is replaced by the value "fred".

Shell variables can *only* have string values, ie a sequence of characters.

Even though shell variables are strings, we can still use them for numbers by using an application program called `expr`.

```
$ A=1
```

```
$ expr $A + 1
```

```
2
```

```
$
```

`expr` looks at its arguments and evaluates them as an arithmetic expression.

But how do we get the result back into A if we want $A=A+1$?

Enclosing a command in backquotes (`) means "replace the command with its output"

So the command:

```
$ `echo cat myfile.txt`
```

is replaced with "cat myfile.txt" and this command is then interpreted by the shell and copies the file "myfile.txt" to output

What does this do?

```
$ `echo echo hello`
```

What does this do?

```
$ `echo echo hello`
```

- the first echo prints its arguments, ie "echo hello"
- the output ("echo hello") replaces the line
- the shell then executes the command "echo hello"
- and the output is:
 - hello

We can get the standard output of "expr" and use it as an argument:

```
$ A=1
```

```
$ A=`expr $A + 1`
```

```
$ echo $A
```

```
2
```

```
$
```

Shell Control structures

In future segments we will introduce shell control structures that include the **if** statement, **for** statement, **switch** statement.

When combined with shell variables and **pipes** the shell becomes a powerful programming language.

We can store shell commands in a file and run them as if they were another command. These are known as "shell scripts".

Shell Scripts

Shell commands can be turned into a program, usually called a "shell script".

Here is a very simple shell script:

```
#!/bin/bash  
echo $1 is your name
```

If we store those lines in a file called "myname", we can run it using:

```
$ bash myname Bob  
Bob is your name
```

Shell Scripts

What does it mean?

specifies what command to use to interpret the script



```
#!/bin/bash
```

```
echo $1 is your name
```

Shell Scripts

What does it mean?

```
#!/bin/bash
echo $1 is your name
```



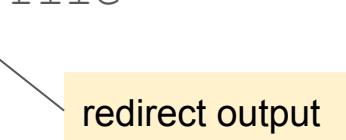
special variable that specifies the first argument for the script

Pipes and redirection

You can redirect the output of a command so it goes to a file instead of the screen:

```
$ echo hello >myfile  
$ cat myfile
```

hello



You can also take the output of one command and send it to the input of another using a *pipe*:

```
$ cat myfile | wc
```



You can use pipes to combine commands in many useful ways.

Unix has a lot of simple commands that process text: count lines/words/chars, select lines based on a pattern, cut out sections of lines, paste sections to make lines, calculate, etc

For example the following commands process plain text:

wc	line/word/character count
cut	remove a portion of each line (eg cut out columns)
paste	paste several files together vertically
sed	stream editor
tr	transliterate one class of character to another
sort	sort a file on one or more fields
grep	search for matching lines
uniq	remove duplicate lines
fmt	format the text

In Unix based systems (eg Linux) the documentation for all the basic commands is stored in files. You can display the relevant file using the "man" command.

For example, to get the manual entry for the man command itself:

```
$ man man
```

```
MAN(1)                                Manual pager utils                               MAN(1)

NAME
    man - an interface to the on-line reference manuals

SYNOPSIS
    man [-C file] [-d] [-D] [--warnings[=warnings]] [-R encoding] [-L locale] [-m system[,...]] [-M path] [-S list]
    [-e extension] [-i|-I] [--regex|--wildcard] [--names-only] [-a] [-u] [--no-subpages] [-P pager] [-r prompt] [-7]
    [-E encoding] [--no-hyphenation] [--no-justification] [-p string] [-t] [-T[device]] [-H[browser]] [-X[dpi]] [-Z]
    [[section] page[.section] ...] ...
    man -k [apropos options] regexp ...
    man -K [-w|-W] [-S list] [-i|-I] [--regex] [section] term ...
    man -f [whatis options] page ...
    man -l [-C file] [-d] [-D] [--warnings[=warnings]] [-R encoding] [-L locale] [-P pager] [-r prompt] [-7] [-E en-
    coding] [-p string] [-t] [-T[device]] [-H[browser]] [-X[dpi]] [-Z] file ...
    man -w|-W [-C file] [-d] [-D] page ...
    man -c [-C file] [-d] [-D] page ...
    man [-?V]
```

DESCRIPTION

man is the system's manual pager. Each page argument given to man is normally the name of a program, utility or function. The manual page associated with each of these arguments is then found and displayed. A section, if provided, will direct man to look only in that section of the manual. The default action is to search in all of the available sections following a pre-defined order ("1 n 1 8 3 2 3posix 3pm 3perl 3am 5 4 9 6 7" by default, unless overridden by the SECTION directive in /etc/manpath.config), and to show only the first page found, even if page exists in several sections.

The table below shows the section numbers of the manual followed by the types of pages they contain.

Summary

In this segment we introduced:

command line concepts

Unix shell

shell variables

shell control structures

shell scripts

pipes and redirection

common shell commands (programs)



2.2 Running Programs: Processes

INFO1112

Bob Kummerfeld

Processes and the Unix Kernel

In normal operation a Unix system has many programs stored in memory at the same time. If it is a single CPU system only one program can run at a time. The Unix kernel is responsible for switching between programmes.

This depends on a feature of CPUs called an *interrupt*. This is where a currently running program is interrupted, all of its "state" (memory, cpu registers etc) is saved, and the kernel is entered. Later the kernel can restore the state and restart the program. The program will continue as if nothing had happened.

Programs in memory have a standard layout which we will discuss later.

The combination of the memory occupied by the program and all of its state (which also includes details of what files it currently has open) is called a ***process***

A Unix system will usually have a large number of processes in the memory at any one time. On a single CPU machine only one process will be executing at a time (or possibly none if the kernel is running).

A typical small system will have more than 150 processes!

Processes

We can find out what processes are running with the `ps` command.

`ps` allows you to get information on all processes or a subset:

```
$ ps
```

PID	TTY	TIME	CMD
6438	pts/0	00:00:00	bash
17380	pts/0	00:00:00	ps

Here we only see processes belonging to the logged in user (User Ids will be discussed soon).

Processes

The ps command can display more information about running processes

```
$ ps al |more
```

F	UID	PID	PPID	PRI	NI	VSZ	RSS	WCHAN	STAT	TTY	TIME	COMMAND
0	1003	19208	19207	20	0	21208	4964	wait	Ss	pts/0	0:00	-bash
0	1003	24486	19208	20	0	27640	1452	-	R+	pts/0	0:00	ps al
0	1003	24487	19208	20	0	8236	748	pipe_w	S+	pts/0	0:00	more



Documentation - the man command

All Unix systems have documentation about commands that can be accessed using the "man" command.

```
$ man ps
PS(1)           User Commands           PS(1)
NAME
    ps - report a snapshot of the current processes.
SYNOPSIS
    ps [options]
DESCRIPTION
    ps displays information about a selection of the active processes. ....
```

User IDs

When a process is running it has an associated User Identification number or UID.

This is used in the file system to indicate the owner of a file or directory.

We map the UID onto a name using the ***password file*** stored in /etc/passwd

Entries in the password file look like this:

```
info1112:x:1003:1003:Bob Kummerfeld,,,:/home/info1112:/bin/bash
```

Each entry is a line of text. In this one 1003 is the UID, /home/info1112 is the home directory and /bin/bash is the shell program for this user.

File Ownership

The owner of a file is specified by a UID:

```
$ ls -l
```

```
total 4
```

```
-rw-rw-r-- 1 info1112 info1112 6 Jul 26 12:46 myfile
```

owner



The ls command looks up the password file and shows the name associated with the UID.

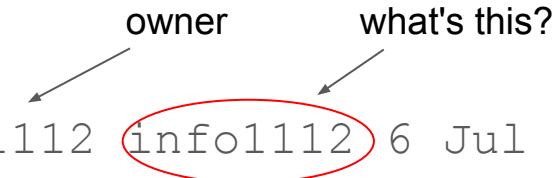
File Ownership

The owner of a file is specified by a UID and associated name:

```
$ ls -l
```

```
total 4
```

```
-rw-rw-r-- 1 info1112 info1112 6 Jul 26 12:46 myfile
```



The `ls` command looks up the password file and shows the name associated with the UID.

The second field is the "group owner" of the file.

Groups

As well as User Ids, unix has "Group Ids" or gid.

The /etc/group file contains the mapping of gid onto group name.

info1112:x:1003:

When an account is created it also gets a group of the same name.

What are groups used for?

Groups

Files have an owner but also a group. Permissions (read,write,execute/search) are given for the owner, for the group and for anyone else (other).

The /etc/group entry can contain multiple user ids for a group.

For example:

```
adm:x:4:syslog,info1112
```

that entry shows that "syslog" and "info1112" are members of group "adm".

Superuser ID

The uid zero is very special. A process running with owner uid 0 is privileged and can access any file and run any program. This means it can access any device (since devices are files in the namespace). The user with uid 0 is called the ***superuser***

System parameters are stored in files owned by uid 0 so the superuser can reconfigure the system.

The superuser is given the user name "root" since the first process that is executed by the system is the root of a tree of processes.

The superuser can also kill any running process.

The superuser is **all powerful** and therefore very dangerous - beware!

Finding the UID

To find out the UID of the running shell we can use the command:

```
$ id
```

```
uid=1003(info1112) gid=1003(info1112) groups=1003(info1112)
```

This shows the user id, group id and what groups the user is a member of.

When a program is started by the shell (eg the id command) it is run with the uid of the shell. This means that the commands you run can access files owned by you or you have group permission for.

File Access

Files have several associated attributes: name, owner, group, access permissions, position in the name space, location in mass storage. These attributes are also called the *metadata* for the file.

A program (or process) can access a file using the open system call. Your program will be suspended briefly while the operating system kernel takes the path name you specify, finds it in the file system, and sets up a data structure containing the metadata. The data structure also shows where your program is up to when reading the file sequentially.

In a running process, this data structure is referred to by a **file descriptor** which is a small integer.

Open Files

When a program is started, the process is automatically given three open files and associated file descriptors:

- 0 standard input
- 1 standard output
- 2 standard error output

If you are running a program from the command line these correspond to the keyboard, the screen and the screen.

Open Files at the shell level

When you run a command from the shell the standard input/output/error files can be redirected using the notation:

- < for standard input file
- > for standard output file
- 2> for standard error file

And more generally, N> will redirect output for file descriptor N

Example: echo Hello >myfile

this will write the string "Hello" to standard output, but standard output is redirected to the file 'myfile'

Pipes

One of the great inventions in Unix was the ability to create a special "file" called a **pipe** at run time that had the property that whatever you wrote to it was available to read (for a modest amount of data) without it being stored on mass storage.

If this "pipe" was shared by two processes, one process could write data into it and the other process would receive the data when reading.

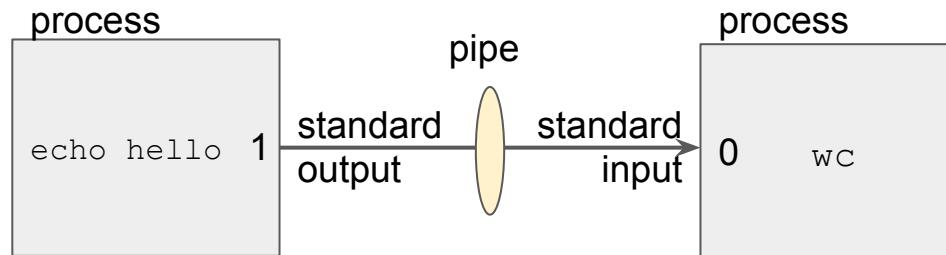
The shell has a special syntax that uses the character '|' to create a pipe between two programs. For example:

```
$ echo hello | wc  
1      1      6
```

Pipes and the shell

When a shell runs the command line "echo hello | wc" it creates a pipe, then starts "echo hello" with standard output replaced by the pipe file descriptor, then starts "wc" with standard input replaced by the pipe file descriptor.

In this way two processes can be made to send data from output of one to input of the other.



Devices in the namespace

Another invention of Unix was to make access to devices the same as any other file: it has a name in the namespace and the standard open/close/read/write/delete system calls do roughly the same thing for devices as they do for normal files.

For example, on a typical Linux system `/dev/sda` is a pathname that refers to a disk drive. By opening and reading from that file you can access the whole device starting at the beginning.

```
$ ls -l /dev/sda
```

```
brw-rw---- 1 root disk 8, 0 Jun 23 13:04 /dev/sda
```

Device

In Unix all devices are accessed this way. Other systems (eg Windows) have special names (eg C:) for devices.

When a device file is opened, the operating system looks at the file metadata and sees that this is not an ordinary file and passes it to the appropriate **device driver**. This is a part of the operating system that manages the particular type of device. For example, a disk drive or a network interface. The device type can be seen in an "ls -l":

```
brw-rw---- 1 root disk 8, 0 Jun 23 13:04 /dev/sda  
      ↑           ↑           ↙  
"block"device   "major" number, "minor" number
```

Other Special files

As well as device files, the namespace can specify pseudo files that, when opened and read, return information about the system.

For example:

```
$ cat /proc/uptime  
3713661.60 29657491.61
```

This shows the total up time and idle time for the system in seconds.

Files in /proc are not device files but are served by a special file system driver.

Text processing with shell commands

Unix has a lot of simple commands that process text: count lines/words/chars, select lines based on a pattern, cut out sections of lines, paste sections to make lines, calculate, etc

wc	line/word/character count
cut	remove a portion of each line (eg cut out columns)
paste	paste several files together vertically
sed	stream editor
tr	transliterate one class of character to another
sort	sort a file on one or more fields
grep	search for matching lines
uniq	remove duplicate lines
fmt	format the text

Text processing with shell commands

By using these commands in pipelines we can do useful things.

The `grep` command is very powerful. It reads lines of text and displays lines that match a pattern or *regular expression*, regular expressions are a topic for another segment.



Summary

In this segment we covered:

processes and the unix kernel

User ids and file ownership

file access

pipes

devices in the namespace

special "files"

Before we begin the proceedings, I would like to acknowledge and pay respect to the traditional owners of the land: the Gadigal people of the Eora Nation. It is upon their ancestral lands that the University of Sydney is built.

As we share our own knowledge, teaching, learning and research practices within this university may we also pay respect to the knowledge embedded forever within the Aboriginal Custodianship of Country.

INFO1112

Nazanin Borhan

2022

Welcome to INFO1112

- introductions
- course resources
- assessment
- reference book
- course content
- Demo



Photo by [Morgan Richardson](#) on Unsplash



Nazanin Borhan

- PhD in Computer Science
- Specialized in Cyber Security and Data Science
- Taught a large range of courses before: programming, Cyber Security, Data Science, networks, etc.

Teaching Team

Our team of tutors is listed on the Canvas page for the course and is led by **Tiancheng (Micheal) Mai.**

The course will be taught with a mixture of online and face-to-face classes.

All lecture materials will be online, and most labs will be face-to-face, but some labs are online.

Each week the course will consist of:

a small number of recorded online presentations
covering some aspect of the course.

You are expected to view these and do associated quizzes (not for marks).

Each week the course will consist of:

A small number of **recorded online presentations** covering some aspect of the course.

You are expected to view these and do associated quizzes (not for marks).

Your journey with quizzes starts today in:

[Pre-Semester Survey](#)

Each week the course will consist of:

A **two hour online session** with the lecturer that will consist of examples, demonstrations and questions.

These sessions will be presented on Zoom each week of semester Monday 12 to 2pm. You will have the opportunity to ask questions during that time.

The lectures will be recorded and available via Canvas.

Each week the course will consist of:

A 2 hour lab/tutorial with a tutor.

There will be homework due at the tutorial **every second week** starting week 2.

Online Resources

Canvas will have links to the presentation videos, lecture notes, lab notes, announcements and self test quizzes.

Edstem will be used for questions/answers and discussions. We hope to answer questions posted there within 24 hours (but hopefully sooner).

Gradescope will be used for mid- and end-semester quizzes.

Git will be used for code management.

Assessment

- weekly homework ($6 \times 5\% = 30\%$)
 - due in your lab in weeks 2,4,6,8,10,12
- assignment 1 (10%). Week 4
- assignment 2 (20%). Week 8
- assignment 3 (20%). Week 13
- mid semester quiz done in lecture (10%). Week 7
- end semester quiz done in lecture (10%). Week 13
- **NO exam** in exam period!

Reference Book

This course introduces a wide range of subjects within Computer Science and there is **no required textbook** available that covers this range.

However, an important skill covered by the course is the ability to use command line tools and understand systems and networks. The book:

Small, Sharp Software Tools: Harness the Combinatoric Power of Command-Line Tools and Utilities

Brian P Hogan

The Pragmatic Programmers

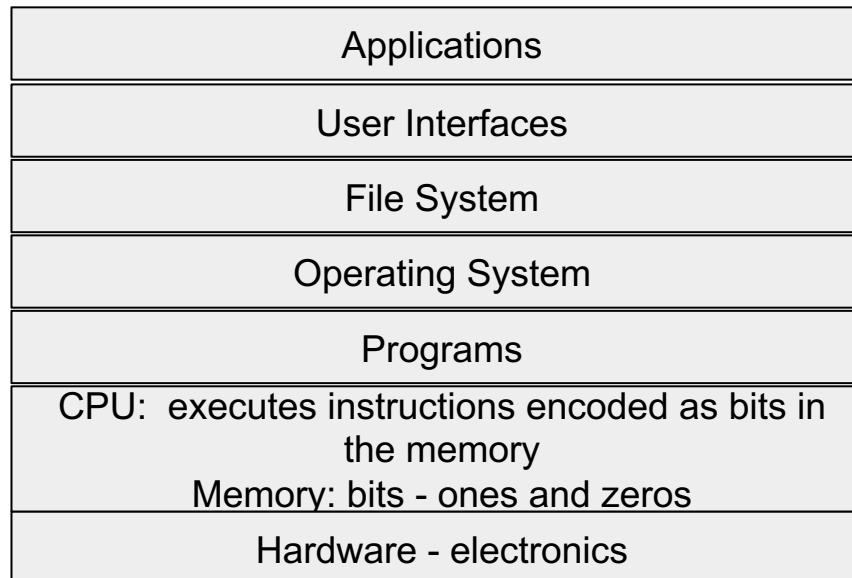
<https://pragprog.com/book/bhcldev/small-sharp-software-tools>

is recommended for this aspect of the course. It is available in print and ebook.

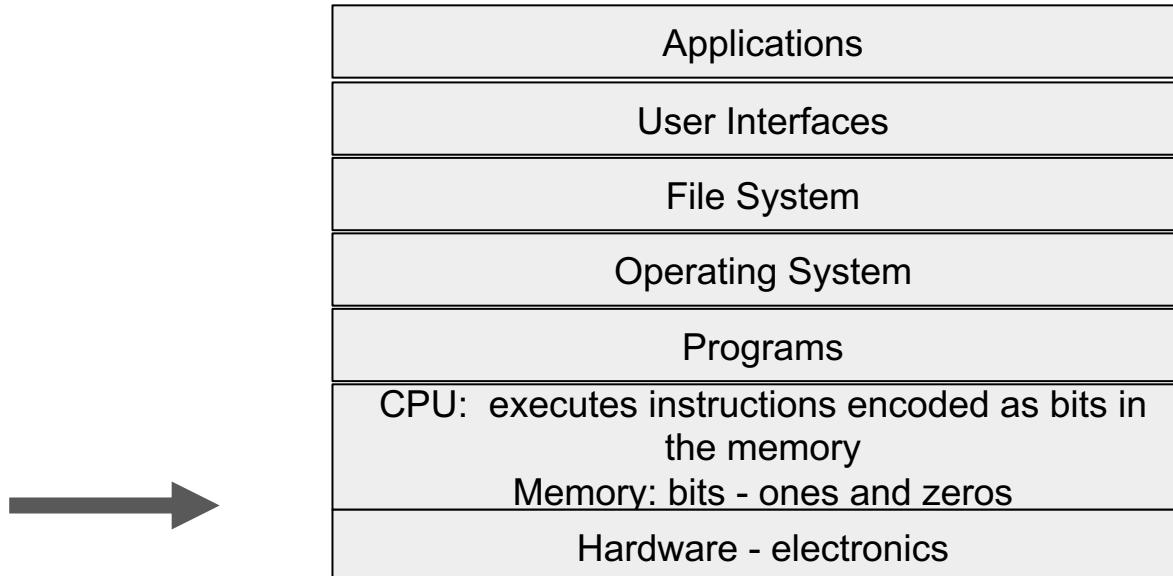
The Course

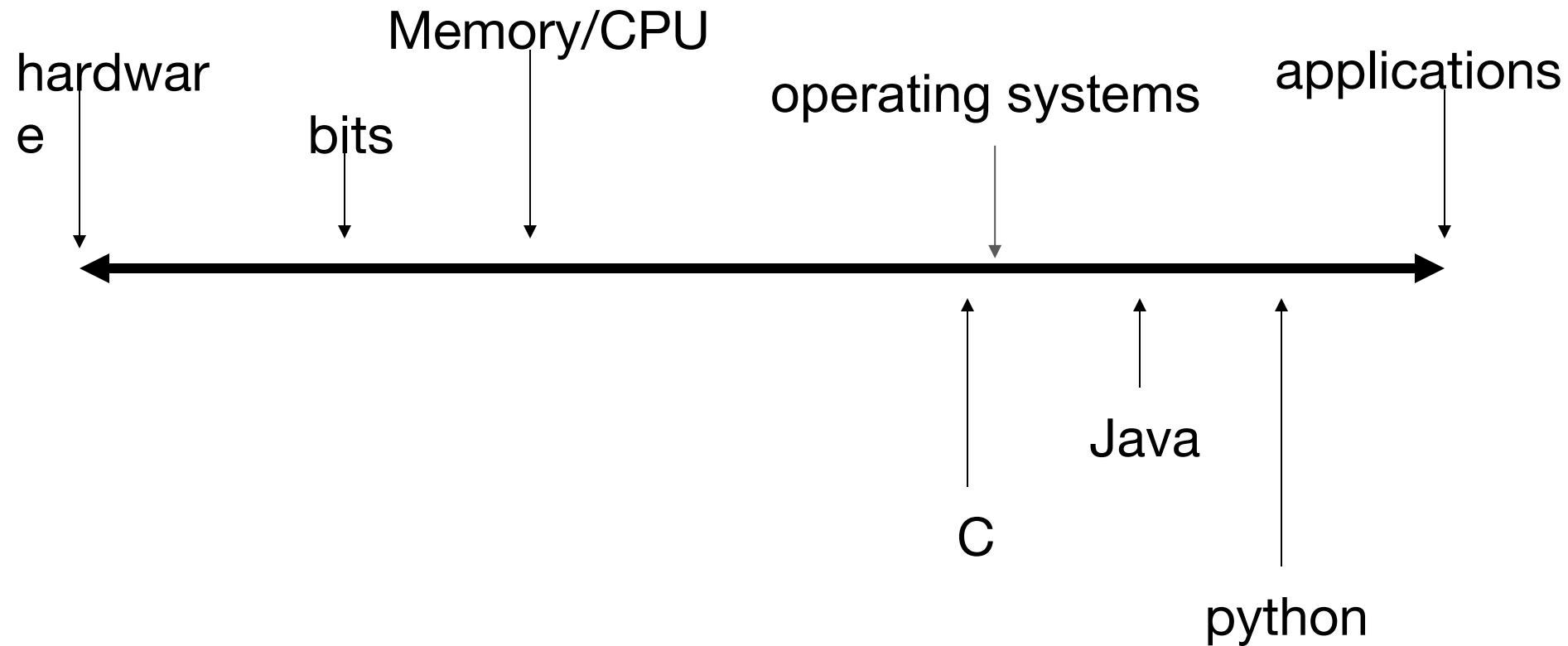
- "bits to applications"
- How does it work?
- a spiral approach - almost all the topics covered in the course will be covered in much greater depth in other courses
- INFO1112 is designed to introduce you to the underlying technical ideas
- ... with a bit of history as well.

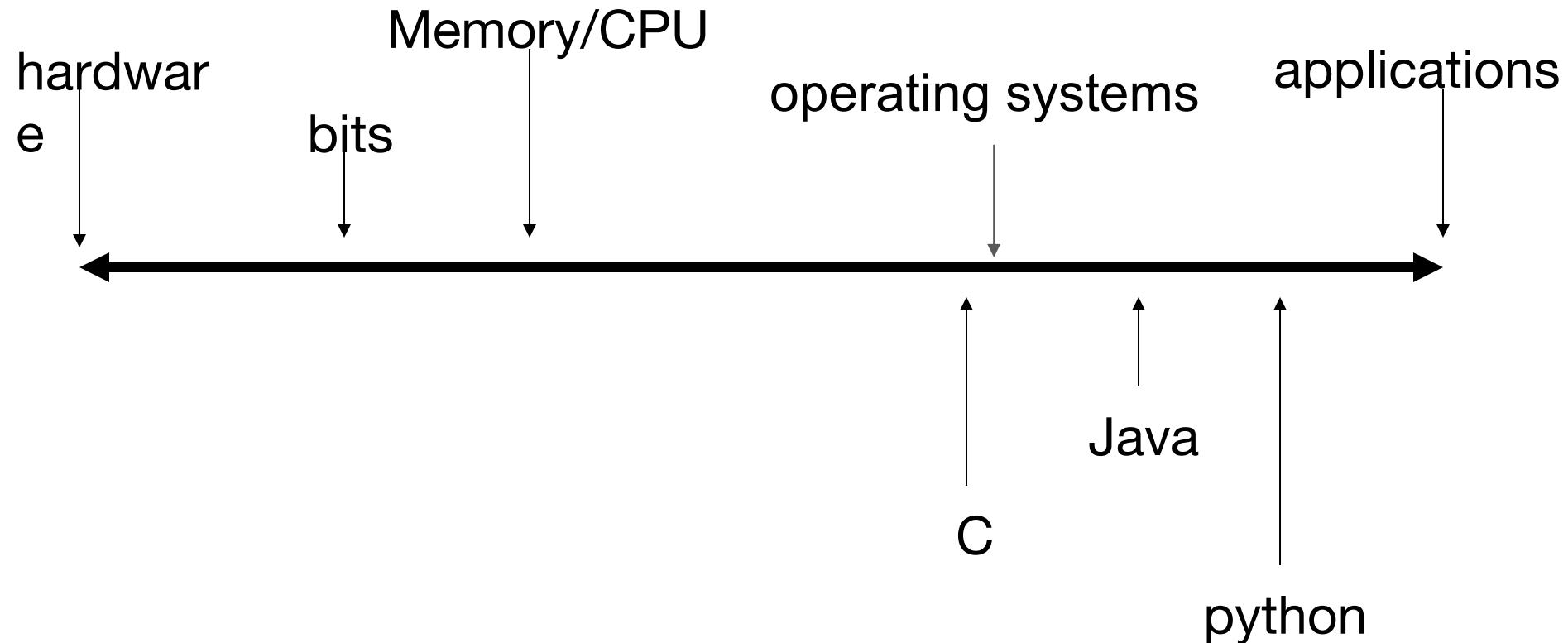
The computer system that you interact with is built in a series of *layers* where each layer builds on the one below



Starting with the bits.....







INFO1112 Weekly Outline

The topics for each week in 2022 will be:

- W1: representations (binary, numbers, characters, instructions), operating system introduction
- W2: processes, command line
- W3: file system, name space
- W4: Memory, the boot sequence
- W5: virtual machines
- W6: networks
- W7: internet protocol: TCP/IP
- mid-semester break
- W8: domain name service, application layer
- W9: computer and network security
- W10: cloud computing
- W11: window systems and Android
- W12: Micro services and Internet based APIs
- W13: course review

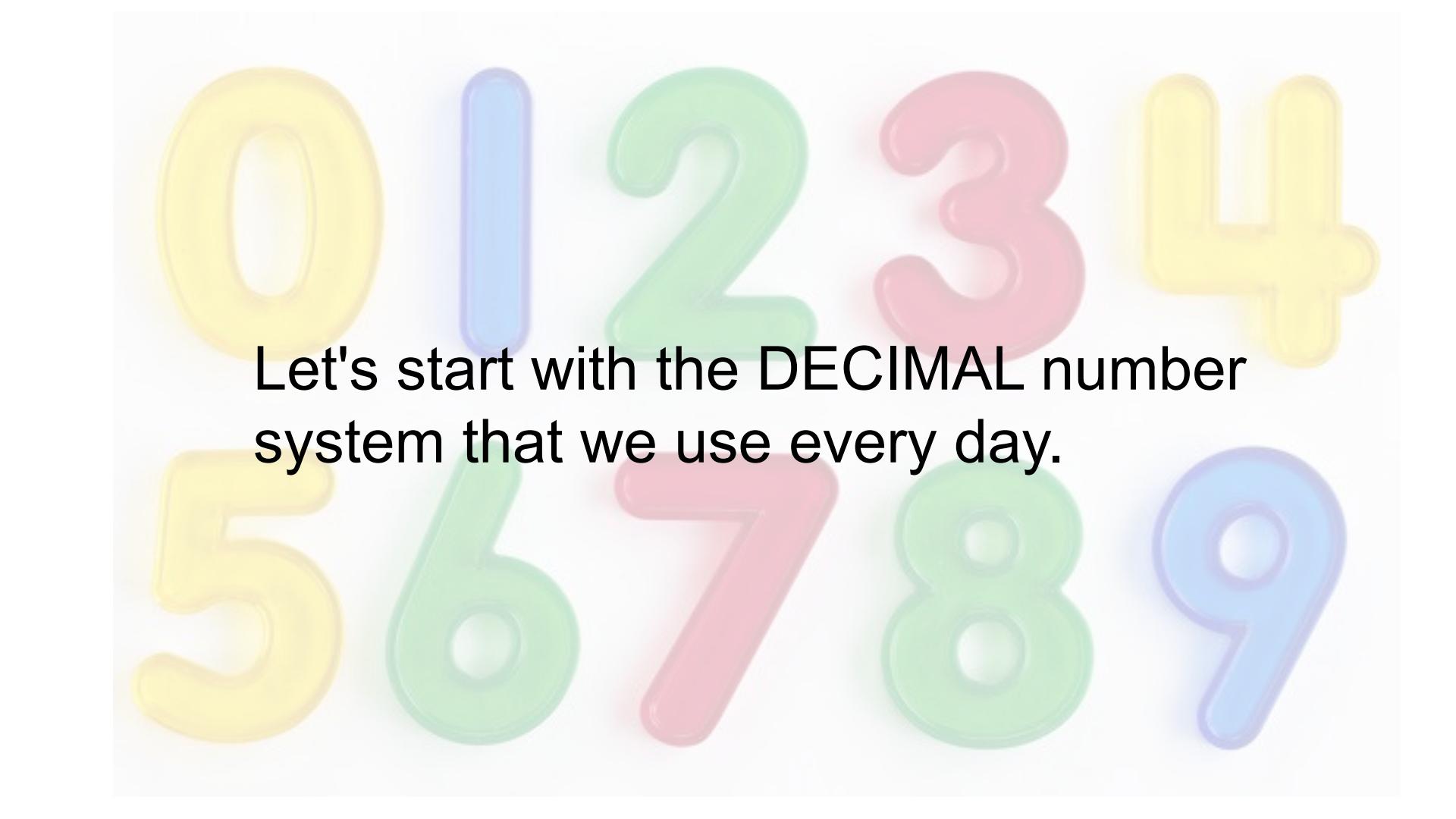
- › Python is used throughout the course, assignments etc
 - › We assume a basic knowledge of Python!
-
- › BUT some students are only learning Python now. Please refer to FAQ section of Canvas site.
 - › Although we will provide extra Python help for students that are new to the language or need revision, but knowledge of python is required for doing the assessments of this course.

end

1.1 Binary Numbers

INFO1112

Bob Kummerfeld



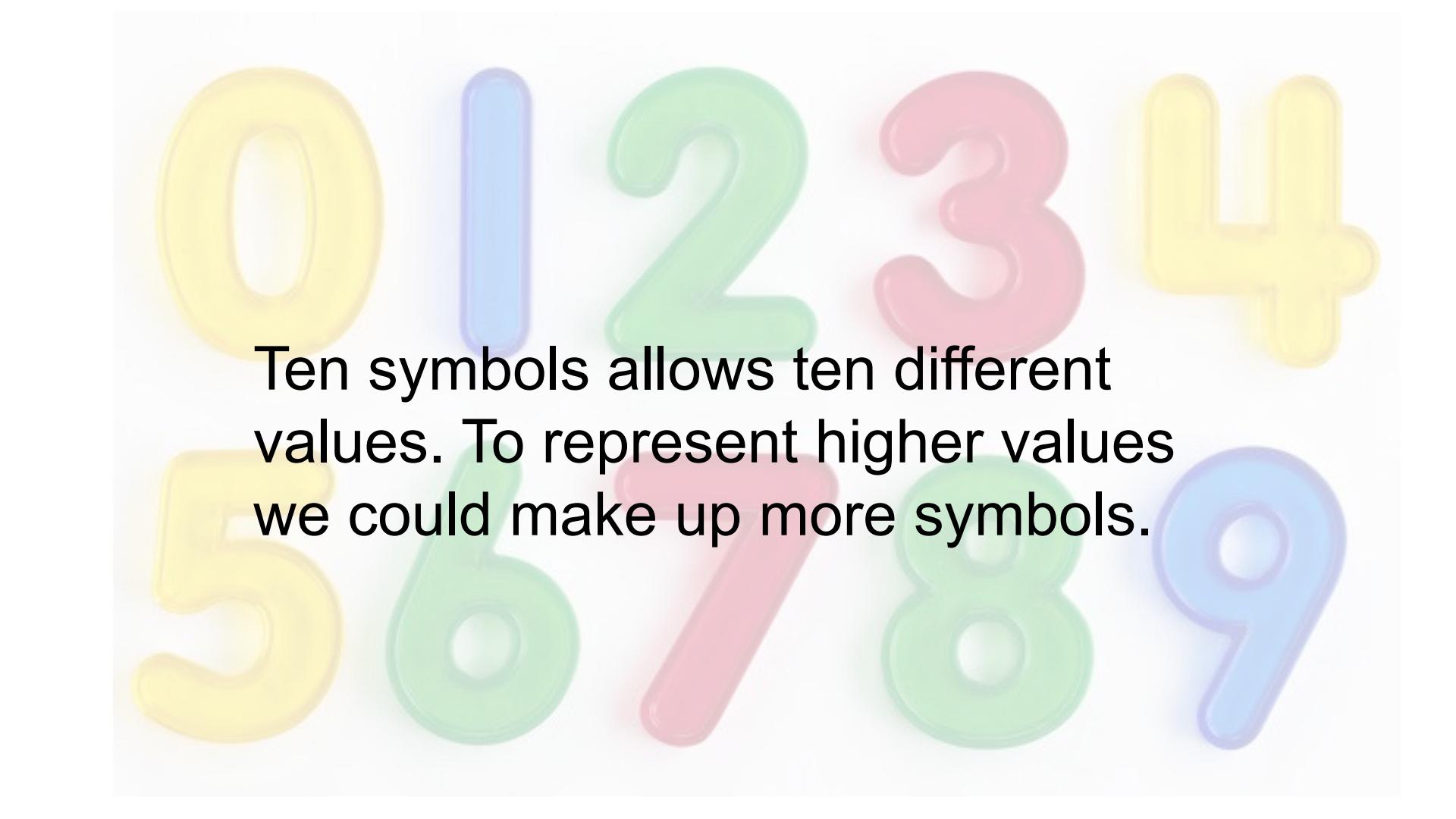
Let's start with the DECIMAL number system that we use every day.

A row of colorful, 3D plastic numbers from 0 to 4. The numbers are arranged horizontally and have a translucent, slightly rounded appearance. The colors used are yellow for 0 and 4, blue for 1, green for 2, and red for 3.

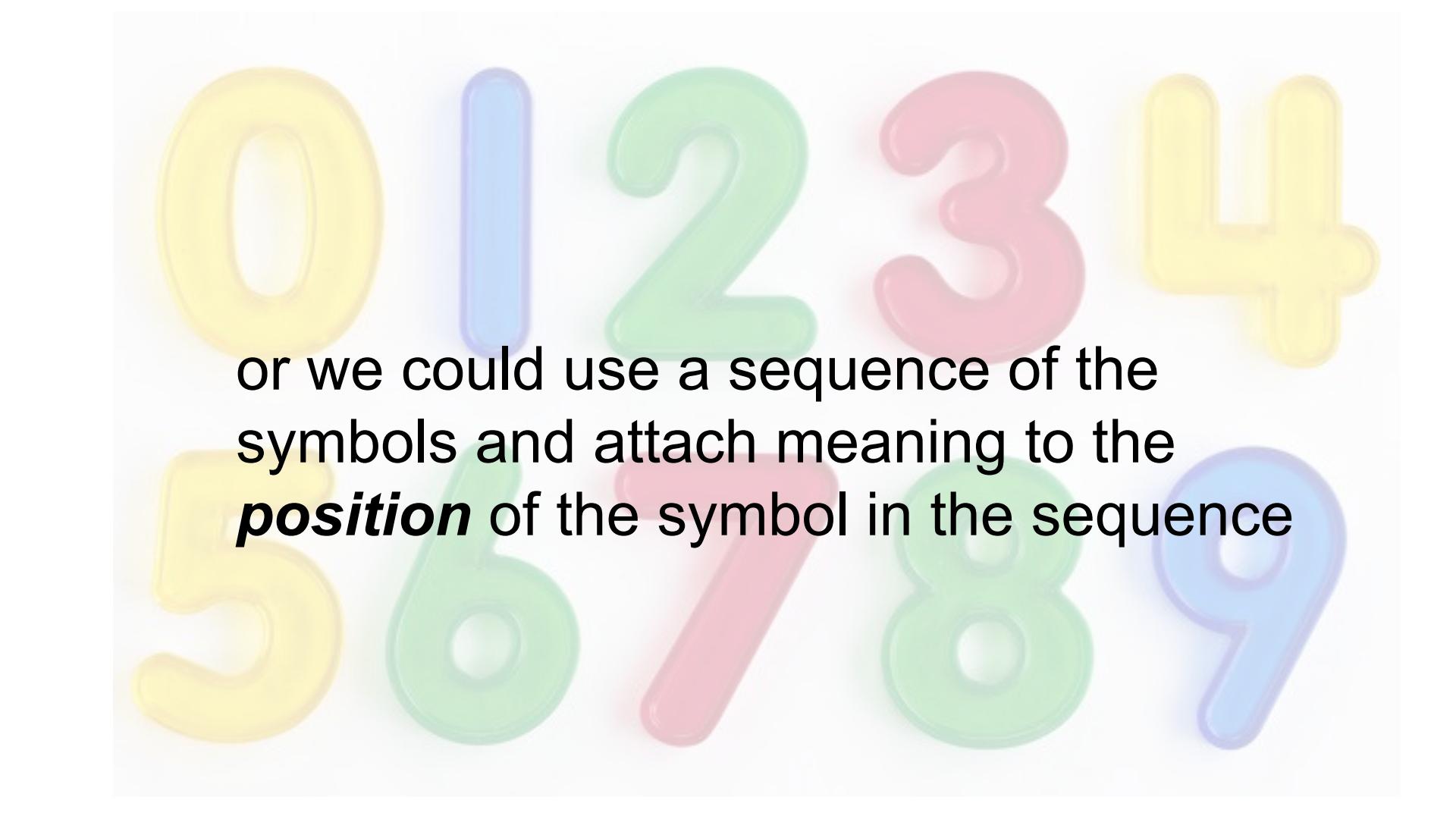
Decimal uses 10 different symbols:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

A row of colorful, 3D plastic numbers from 5 to 9. The numbers are arranged horizontally and have a translucent, slightly rounded appearance. The colors used are yellow for 5, green for 6, red for 7, green for 8, and blue for 9.

A background of large, semi-transparent, colorful numbers (0-9) in various colors like yellow, green, red, blue, and pink, arranged in a scattered pattern.

Ten symbols allows ten different values. To represent higher values we could make up more symbols.



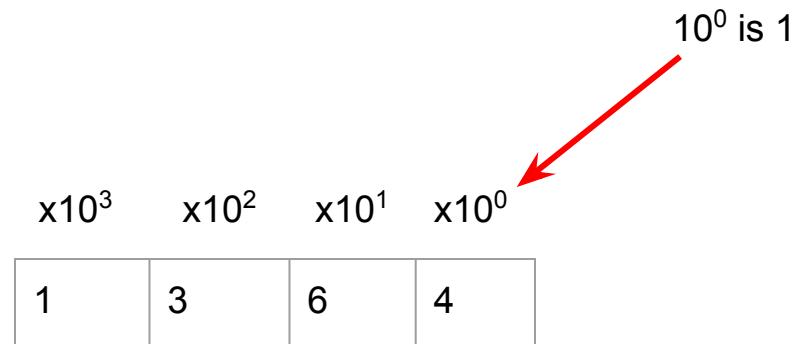
or we could use a sequence of the symbols and attach meaning to the ***position*** of the symbol in the sequence

Example

x1000	x100	x10	x1
1	3	6	4

$$1 \times 1000 + 3 \times 100 + 6 \times 10 + 4 \times 1 = 1364$$

Each position is a power of ten



$$1 \times 10^3 + 3 \times 10^2 + 6 \times 10^1 + 4 \times 10^0 = 1364$$

Each position in the number has a weight.

The rightmost has a weight of one.

The second to the right has a weight of a hundred.

The third to the right has a weight of a thousand.

...and so on.

The weights are a power of 10.

The BINARY number system only uses two symbols:

0 and 1

For BINARY the positions represent powers of 2 rather than powers of 10 used in decimal.

Example

x8	x4	x2	x1
1	0	1	1

$$1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 = 11$$

Each position is a power of two

$x2^3$	$x2^2$	$x2^1$	$x2^0$
1	0	1	1

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 11$$

Binary numbers take up more space than decimal!

For example, the number 9384 in decimal is 10010010101000 in binary.

One way to save space is to use another number base, other than 2 or 10, to represent the number.

If we choose a number for the base that is also a power of two it makes it easy to convert each "digit" into binary and so convert the whole number.

The most common base used in computing is base 16, usually called HEXADECIMAL.

The word comes from the Greek "hex" for six and Latin "deci" for ten.

Originally the British called base 16 "Sexadecimal" from the Latin word "sex" for six. It didn't catch on.

For hexadecimal or base 16 we need 16 different symbols, so we use 0 to 9 and A to F.

0	0	8	8
1	1	9	9
2	2	A	10
3	3	B	11
4	4	C	12
5	5	D	13
6	6	E	14
7	7	F	15

For example, the number:
10010010101000 in binary can be written

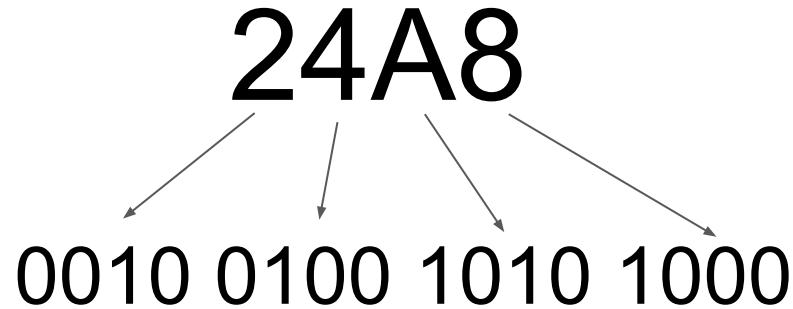
24A8

much shorter!

It is easy to convert to and from binary because each digit in the hexadecimal number represents a number in the range 0-15. If we write the digit in binary form it is:

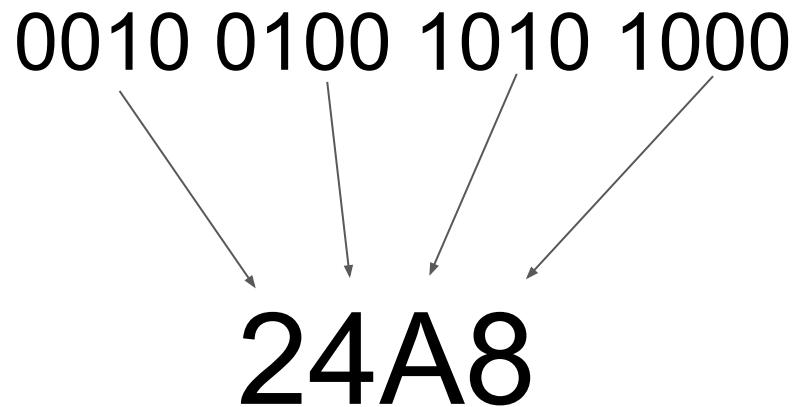
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

If we take each digit in the hex number and convert it we get:



and that is our original number.

We can also do the reverse - convert a binary number to Hexadecimal easily:



You soon get used to converting 4 binary digits into the single hex digit.

Computer memory is created from memory elements that are either **ON** or **OFF**, representing 1 or zero.

These binary digits, or **BITS** are usually arranged in groups of 8 called **BYTES**.

Since bytes consist of 2 groups of 4 bits, it is very easy to write the value as 2 hex digits.

*There are 10 sorts of people
that understand binary: those
who do and those who don't*

1.2 Integer and Real Numbers

INFO1112

Bob Kummerfeld

Why BINARY?

› Why 0/1?

- easy to build a circuit that is on or off and use it to represent physically the mathematical entities 0/1
- A simple switch is off (0) or on (1)
- A transistor is a simple switch: in modern computers the binary digits (0/1 or ***bits***) are represented by the state of transistor switches
 - there are many ways to represent 0/1 physically

We can represent (almost) any piece of information as a sequence of 0's and 1's

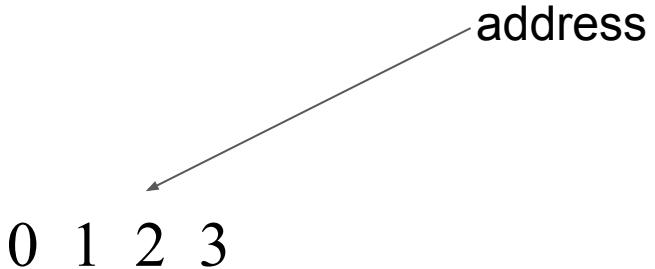
Memory

- › The memory of a computer is made up of binary digits or **bits**
 - But not just a simple stream of bits
- › Memory is divided into 8 bit pieces called **bytes**
- › These pieces are each given a number from 0 to the number of bytes in the memory
- › Each of these numbers is called a memory **address**

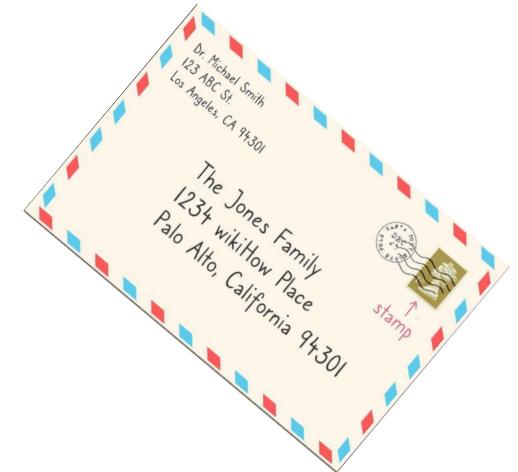
0 1 2 3

.....

Memory



byte



We represent **everything** in the computer using patterns of binary digits.

Integers, fractions, characters, images, sound, video....

Computers can store numbers (integers)

1 = 1

2 = 10

3 = 11

4 = 100

5 = 101

...etc...

We can store these bit patterns in bytes.



But each byte is only 8 bits long – this puts a limit on the range of values that can be represented. (How big?)

Numbers in the computer are not the same as integers in mathematics!

Binary representation of numbers

- › What does a decimal number 100010 mean?
- › What does a binary number 100010 mean

We can also represent fractions:

- › What does a decimal number 100.010 mean?
- › What does a binary number 100.010 mean?

Computers don't have infinite memory!

- › Number representations are finite: we choose a certain number of bits to represent a number
 - › Representations matter: we agree on what the bits mean
- › Memory is always finite

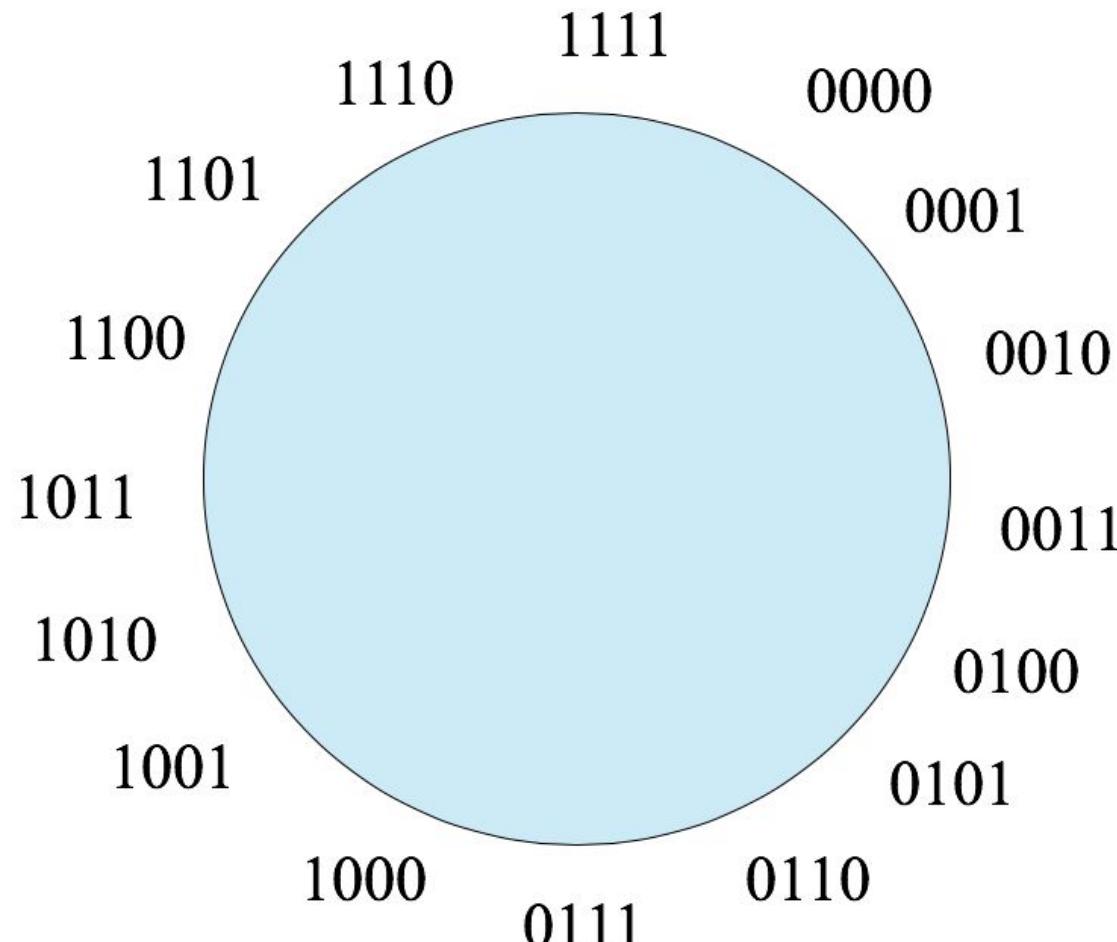
Positive and Negative numbers

To represent negative numbers we need to choose another pattern of binary digits other than the pattern used for positive numbers. Usually this is some variation on the pattern used to represent the positive value.

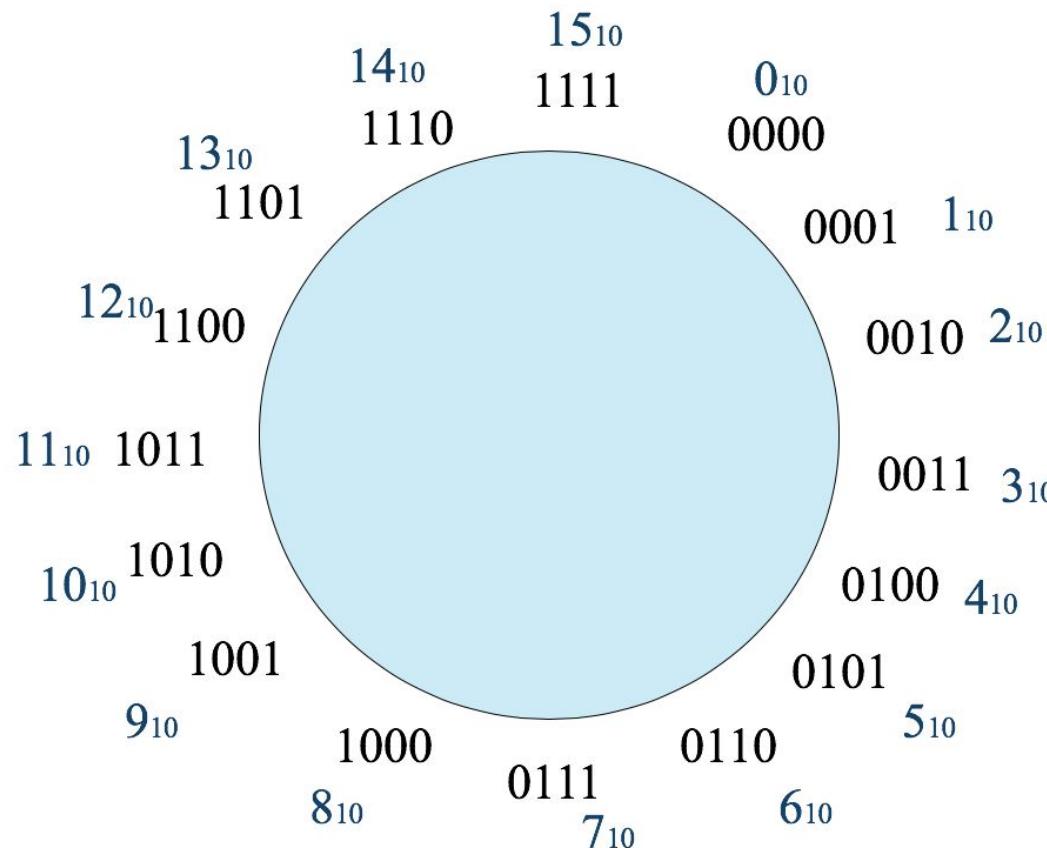
One approach is to set aside one of the bits to represent the sign. For example, zero for positive, one for negative.

But there are other ways...

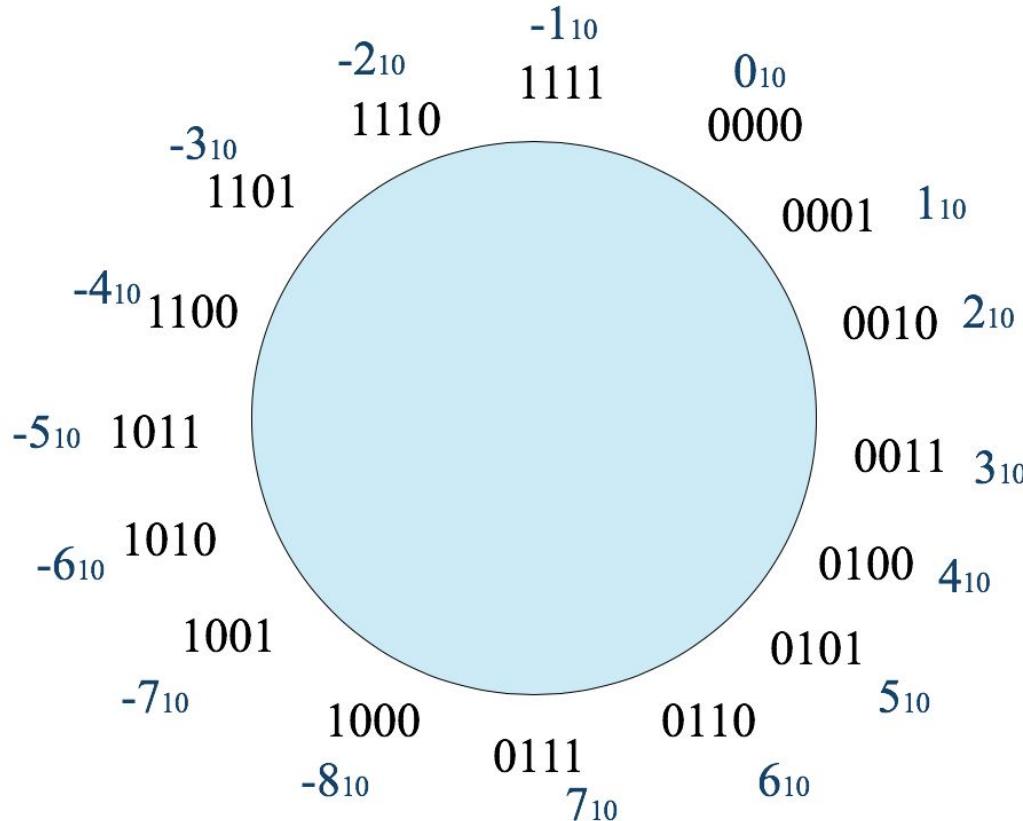
Number Circle for 4 bit numbers



Number Circle for 4 bit numbers - Unsigned Interpretation



Number Circle for 4 bit numbers - 2s Complement Interpretation



Using our 4 bit 2's complement numbers:

if $a > 0$ and $b > 0$, is $a + b > 0$?

if $a < 0$ and $b < 0$, is $a + b < 0$?

is X squared ≥ 0 for all integers X ?

What about $5+6$? try it.... (use the number circle)

What about $-2 + -3$? try it....

Using our 4 bit 2's complement numbers:

if $a > 0$ and $b > 0$, is $a + b > 0$?

if $a < 0$ and $b < 0$, is $a + b < 0$?

is $x^2 \geq 0$ for all int x ?

Number representations are finite!

How big can an integer be?

- › Four bits would only allow a very small range of numbers to be represented. In practice numbers are usually represented with more than 4 bits. Integers are often represented in 4 bytes or more.
 - how many bits in 4 bytes?
 - what is the maximum size of an integer that can be represented in 4 bytes?
 - what about 2s complement representation in 4 bytes?
- › A 4 byte integer is known as "single precision". Some computers allow 8 byte integers or "double precision".

Real Numbers

(ie not integers)

- Potentially infinite precision as well as infinite size
- Need to impose limits in a real machine
- Many schemes for representing real numbers

For example, in a “fixed point” number:

101010.100001

we have a fixed number of bits before and after the binary point.

What is the decimal value of that number?

In fixed point representation, how do we represent $\frac{1}{3}$ in binary?

The “floating point” approach

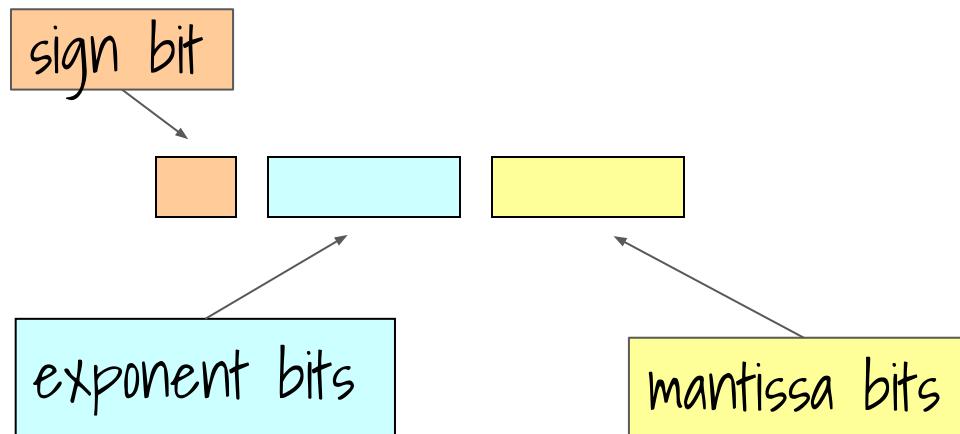
This is the approach used in “**scientific notation**”. Thus, in decimal, **315.82** could be expressed as

$$3.1582 \times 10^2$$

mantissa = 3.1582

exponent = 2

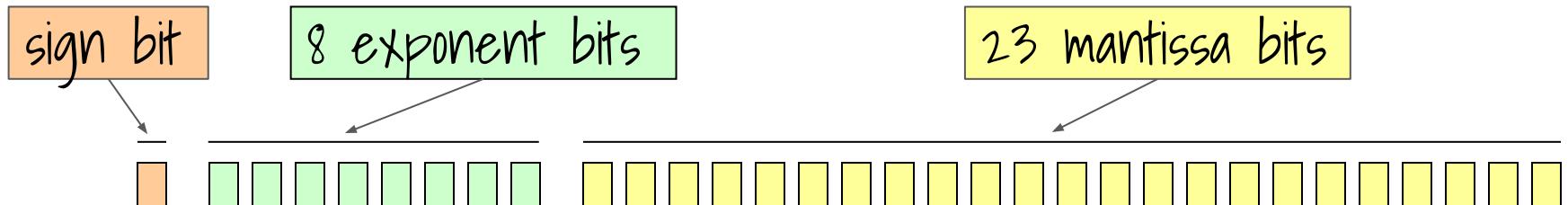
In binary floating point the bits are arranged like this:



General structure

Floating point widths are commonly referred to as (IEEE-754):

- single-precision 32b
- double-precision 64b
- extended-precision 80b



Problems with floating point numbers

- › It is not possible to represent many numbers perfectly in floating point
- › problems arise when doing some sorts of arithmetic that involves very large numbers (big positive exponent) and very small numbers (big negative exponent).
- › repeated operations can lose precision (eg add 0.1 to itself a million times)
- › the order of operations can change a result
- › because of these problems you should be careful testing results for equality

Representing Numbers with Binary Patterns

I have described the main methods of representing numbers with patterns of binary digits. But there are other approaches.

For example:

- › arbitrary precision numbers: represented with a variable string of bits
- › rational numbers: represented with a pair of integers

1.3 Characters

INFO1112

Bob Kummerfeld

Computers can also store characters

We can choose some patterns of bits to represent characters.

Characters are often represented in a single byte, but this restricts the range of characters (how many?).

Characters can also be represented with multiple bytes, either a fixed number of bytes (eg 2) or a variable number of bytes (how do we know how many?).

Character Representation

Example: We can represent characters using a code that associates an 7-bit number with a character

A = 1000001 = 65 (decimal)

B = 1000010 = 66

...

a = 1100001

b = 1100010

space = 0100000

etc

American Standard Code for Information Interchange

These are examples in the ASCII character code

Maximum code value is 127 (7 bits) or 1111111

What is the implication?

Character Codes

Only 0-127 (128) characters can be represented in ASCII
Fine for English, what about other languages? Chinese? Or even French?
There are many character codes in use.

After many years, and many proposed standards, a character code called "**Unicode**" was developed by a consortium of companies and in cooperation with the International Organisation for Standardisation and first released in 1991.

Character Codes

Unicode uses 32 bits/character (4 bytes) and so could theoretically represent over 4 billion characters!

Currently (2020) there are over 140,000 characters represented in unicode that cover more than 150 languages.

There are also characters that are no longer used except in historical documents represented in Unicode. And there are "emojis" 🙌😊👍

😊 has code 1F600 in hexadecimal

😷 has code 1F637 in hexadecimal



Character Codes

Note that the characters are based on the symbol or "grapheme" and shouldn't be confused with the font. For example, "A" and "À" have the same Unicode.

More Examples of characters:



Unicode Encoding

Four byte characters could be a big waste of space if you only needed the ASCII characters. It would multiply the size of documents by 4. For this reason, several encodings or compressions were invented that will squeeze the 32 bit character to a smaller size in most cases. In other words, encode the encoding!

The most widely used encoding of Unicode is called UTF-8 or Unicode Transformation Format 8. It encodes all the ASCII characters into one byte with values 0-127. Values larger than 127 indicate other characters which might involve one or more extra bytes. All Unicode characters can be encoded in UTF-8 and it is used by more than 90% of web pages





Unicode Example

The letter

A

has unicode value (shown as a 32bit number in hexadecimal)

00000041

One common encoding (UTF8) represents it as (in Hex)

41 note that this is the same value as the ASCII code

Unicode and programming

Most programming languages now have support for Unicode.

For example, Python uses unicode for all character strings.

Sometimes it is difficult to enter unicode characters and you may have to have special support for keyboards that include the character you want. Alternatively you could use the HEX code for the character.



Characters

We have now seen that almost any character (actually the "grapheme") can be represented with a string of bits.

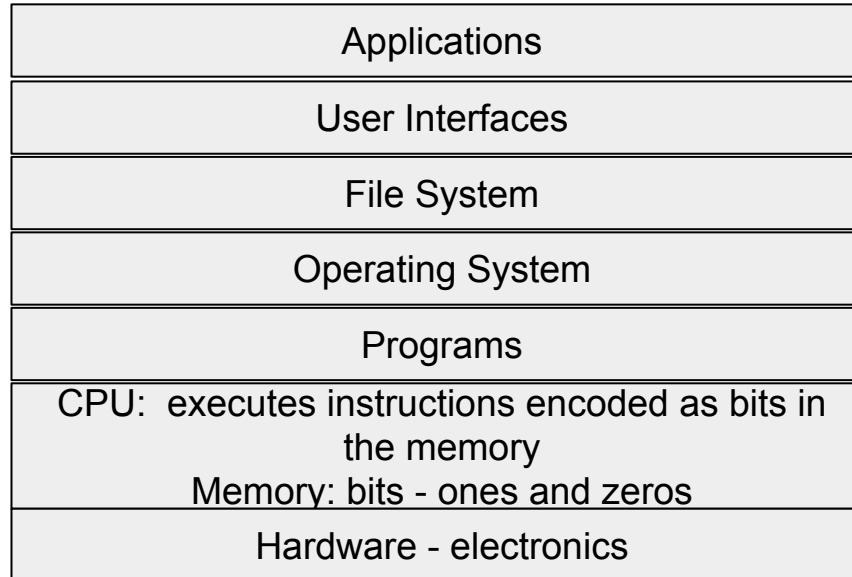
Unicode is the most widely used standard for this representation.

1.4 Instructions

INFO1112

Bob Kummerfeld

Machine Instructions

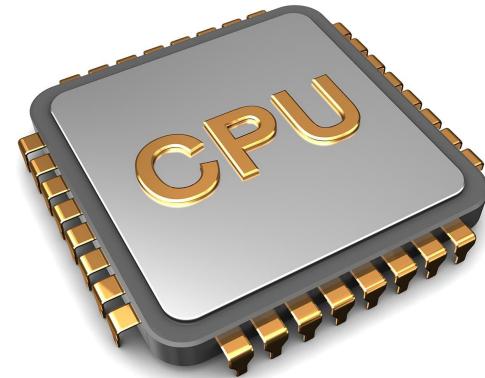


Instructions or "Code"

Computer programs are a set of instructions executed by the central processing unit.



- › As well as data (numbers, characters etc), we also store the program instructions (or **code**) in the memory of the computer
- › A program written in a language like C is translated into machine instructions that are loaded into the memory and executed by the central processing unit (CPU)
- › Machine instructions are also patterns of bits
- › Programs can create machine instructions, store them in the memory and then get the CPU to execute them



Central Processing Unit - CPU

- › Instructions are represented in bits and stored in a sequence of bytes, eg:

01100100 10001100

This might (in some hypothetical machine) mean “add the integer in address 4 to the integer in address 8 and store the result at address 12”

- › the idea to store instructions in the same memory as values was **the** major breakthrough in the development of computers
 - “stored program computers”

Instructions

The example might be broken down as:

0110 code for “add” instruction

0100 address 4

1000 address 8

1100 address 12

Instructions

There are many different types of CPU, each with their own set of instructions. The bit pattern for an "add" instruction in one type of CPU may be very different from another type.

Some CPUs have very simple sets of instructions while others have many more instructions and more complex instructions. They all store the instructions as patterns of bits in the memory.

Registers

- As well as instructions, real machines usually have special storage in the CPU called *registers*
- These are usually the size of an integer and are used for intermediate results eg:

load R1, 4 //load contents of address 4 into R1

add R1, 8 // add contents of addr 8 to R1

store R1, 12// store R1 into address 12

indicates register 1

Assembly Language

- › Instructions like:

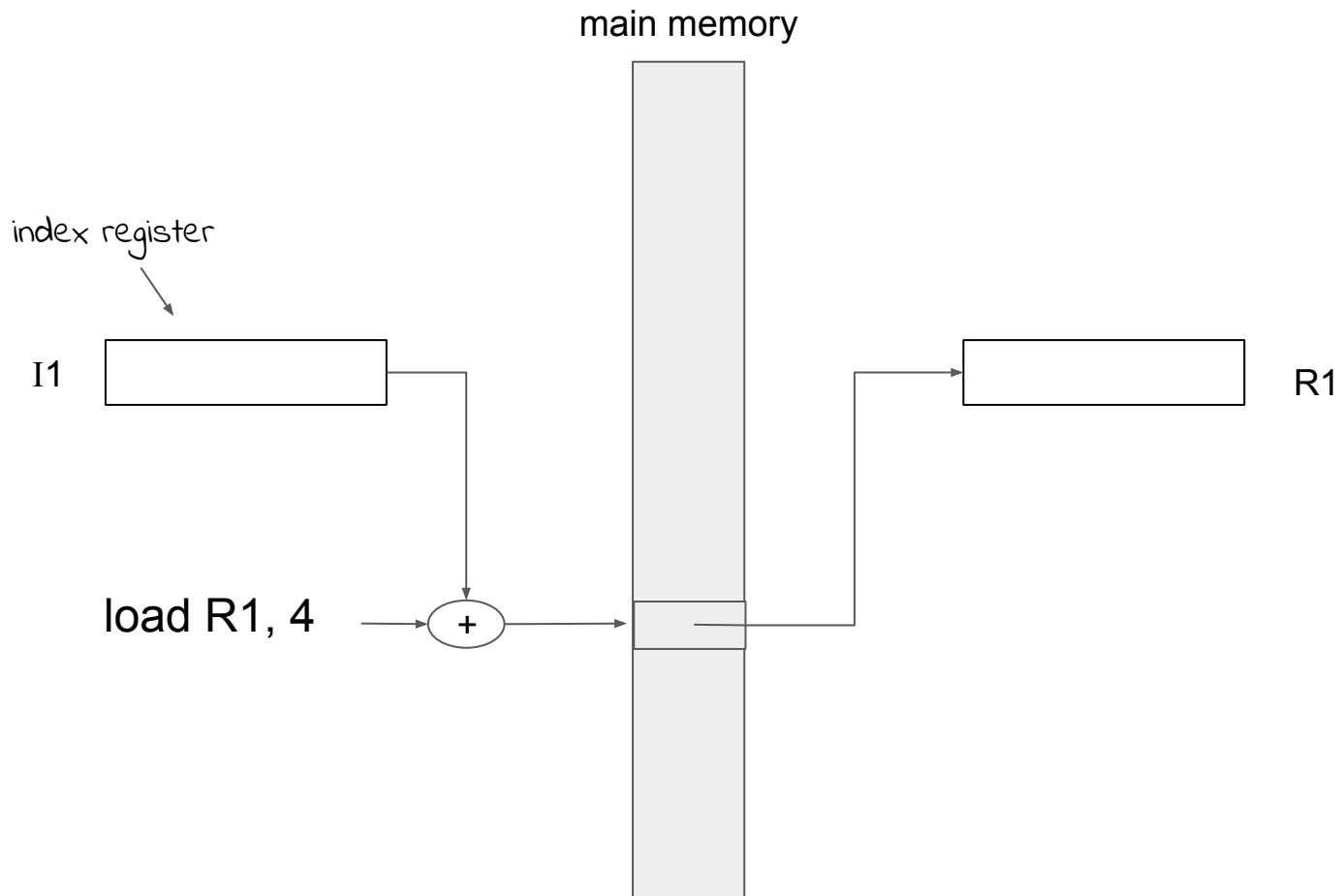
load R1, 4 //load contents of address 4 into R1

are a human readable way of writing the machine instruction

- › This form is called ***assembly language***
- › Assembly language is translated (by a program called an *assembler*) into the binary presentation

"Index" Registers

- › As well as general purpose registers for intermediate results, the cpu might have special registers that contain addresses or offsets used when fetching a value from memory
- › This makes moving through strings and lists of things easier



Intel 386 assembler

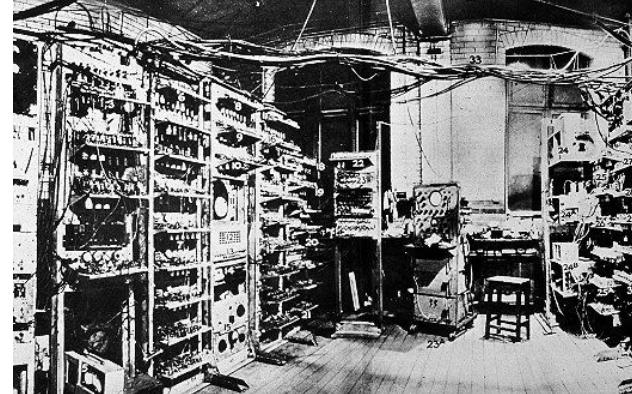
```
.text
.global _start
_start:

    mov $4, %eax /* write system call */
    mov $1, %ebx /* stdout */
    mov $msg, %ecx
    mov $msgend-msg, %edx
    int $0x80
```

```
    mov $1, %eax /* _exit system call */
    mov $0, %ebx /* EXIT_SUCCESS */
    int $0x80
```

```
.data
msg: .ascii "Hello, world\n"
msgend:
```

Some History



- › The first stored program computer to run a program was (probably) the "**Manchester Baby**" (also known as the **Small-Scale Experimental Machine**) developed at the University of Manchester in the UK in 1948.
- › It was designed as a testbed to try out ideas for a larger computer, and not as a practical computer and was never used for real problems.
- › It had 1024 bits of memory in 32 bit groups called 'words'. There were 32 words.
- › There were 8 instruction codes, encoded in 3 bits.
- › The first program to run (in June 1948) had 17 instructions.



Instructions or "Code"

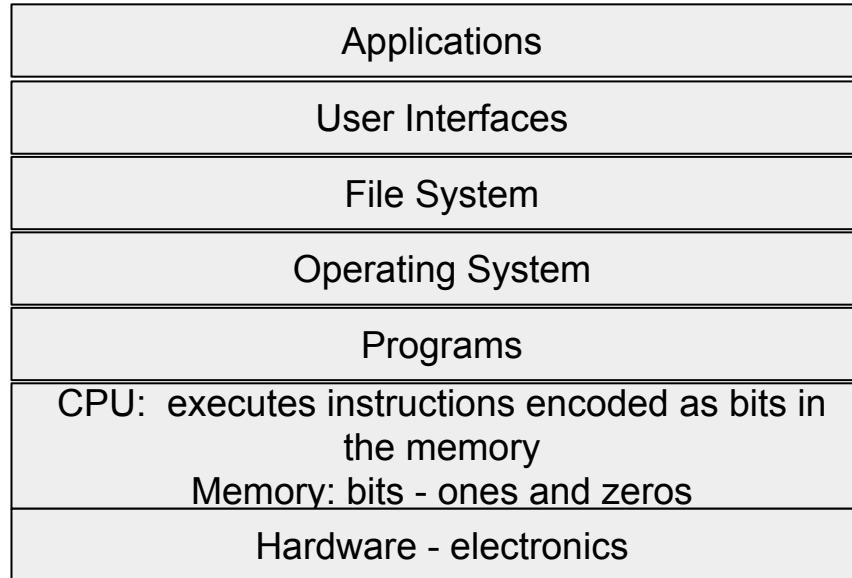
- program instructions are stored in the memory of the computer as patterns of bits
- CPUs usually have small amounts of special fast memory called registers that are used for intermediate results
- instructions are usually written by a programmer in a more readable form called assembly language

1.5 Operating Systems

INFO1112

Bob Kummerfeld

Operating Systems



A "simple" system consists of memory+CPU+input/output devices.

Program instructions are stored in memory and executed by the CPU and may read/write data from external devices such as keyboard and screen.

(you will develop programs like this in ELEC1601)

But systems such as MS Windows or Apple MacOS provide far more:

- › sophisticated graphical user interface
- › other I/O devices such as a mouse
- › a file system
- › network connections
- › concurrency (running more than one program at a time)
- › security



These services are generally provided by an *Operating System*

The Unix Operating System

Invented long long ago (1969) in a place called Bell Laboratories in the USA by Dennis Ritchie and Ken Thompson. Many others contributed.

Released to Universities in source code form in 1975.

Many versions of the original and many clones (eg Linux) have been produced since then. All are recognisable as a Unix based system.

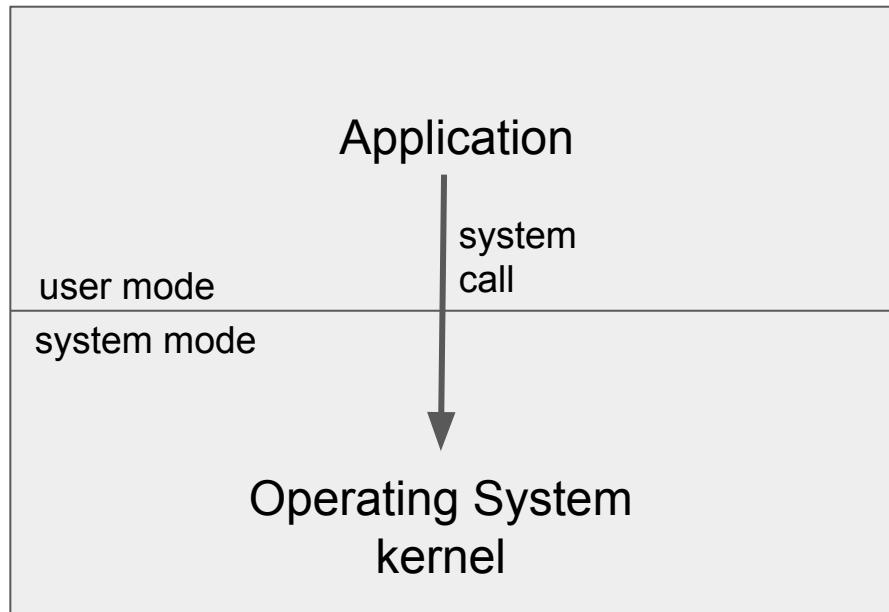
We will use Linux for all our examples and lab exercises.

Linux is a Unix clone developed originally by Linus Torvalds, a Finnish programmer. Linux is now probably the most widely used operating system. Unix based systems are the basis for Android, Apple (MacOS, IOS, ipadOS, watchOS etc) and many others.

An Operating System is a special program that is *privileged* (can access and control all of the hardware) and provides all of the basic services. For other services such as the graphical user interface, the operating system is supplemented by application programs.

To enforce the distinction between the operating system and user programs the CPU usually has (at least) two modes of operation: system and user.

To switch from user mode to system mode (eg when an application program requests a service such as reading a file) and special CPU instruction is executed - system call. This will cause execution to continue in the operating system program in system mode.



The File System

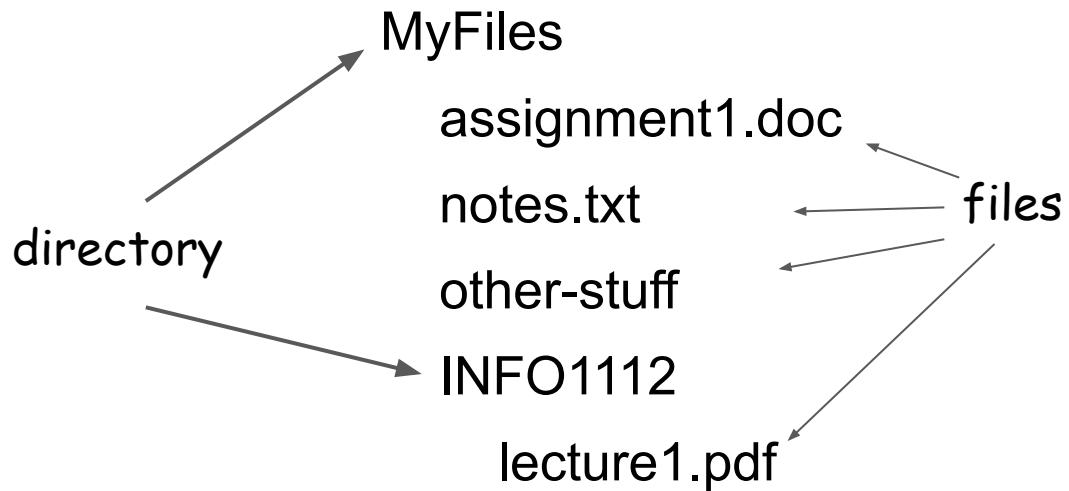
To store "persistent" data that doesn't disappear when we turn off the computer we need storage other than main memory. This can be on a separate device such as a disk drive connected to your computer or on another computer connected to the network.

"disk" drives are implemented with various technologies but all provide **persistent** storage.

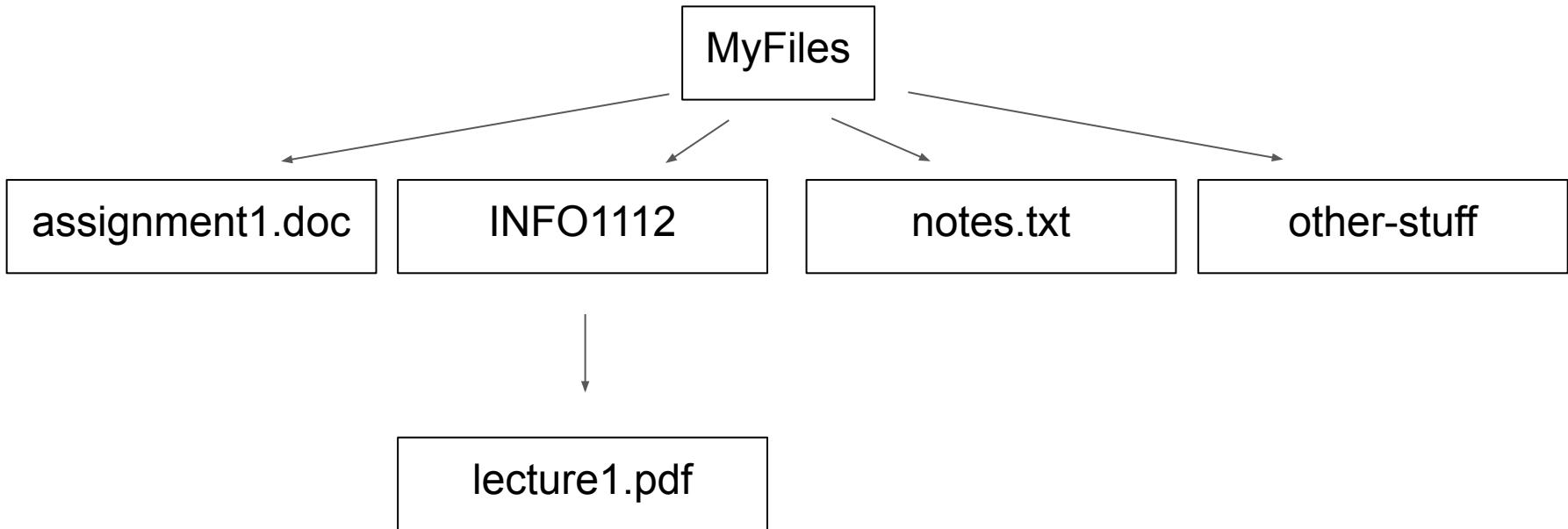
The raw storage provided by these mass storage devices is not so convenient to use, so in nearly all cases we structure the storage into **files** and **directories**.

This is what we call the **file system**.

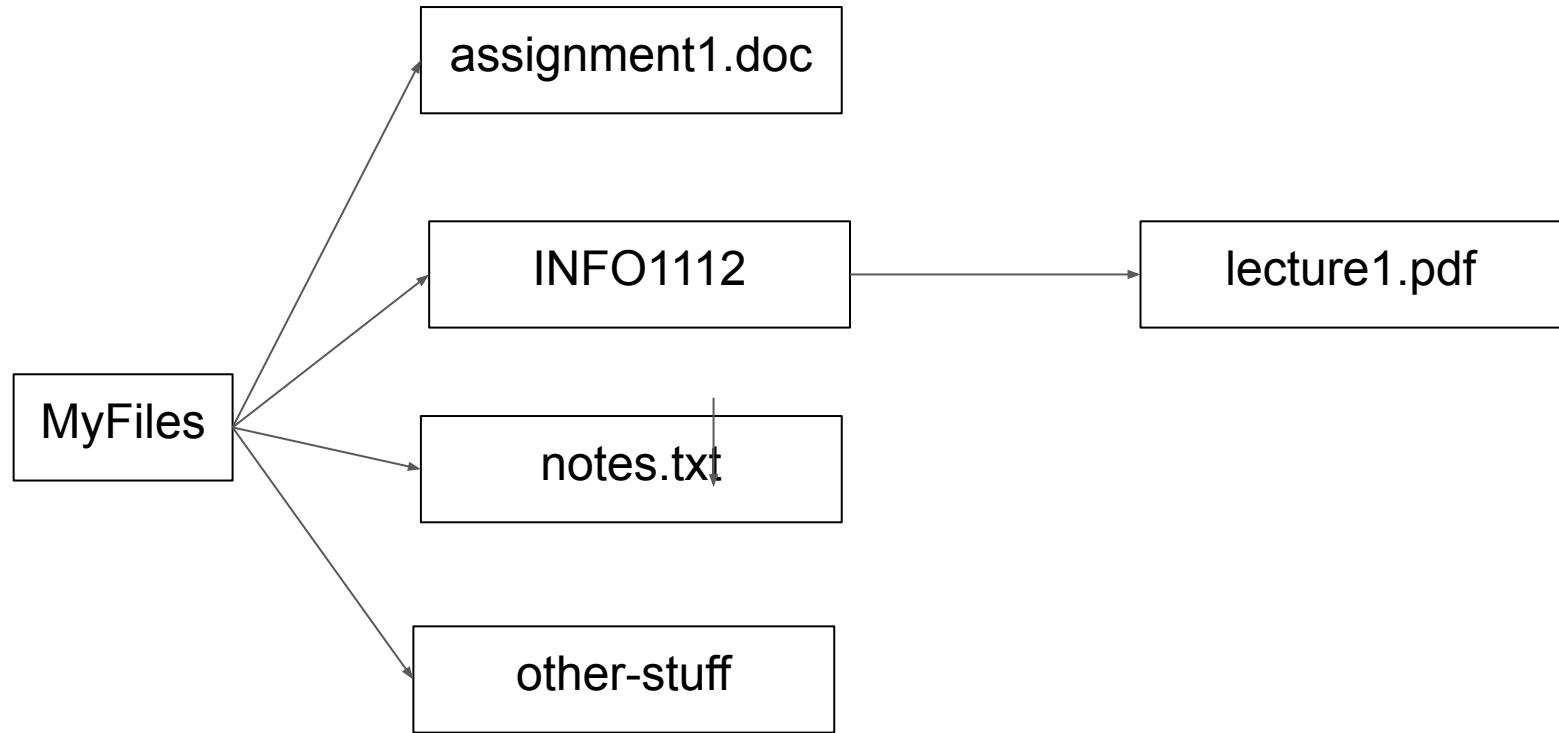
The file system (in Unix) consists of a set of directories (or folders) that can contain files or sub-directories.



Another way to look at this is:



Or this:



This is called a *tree* data structure

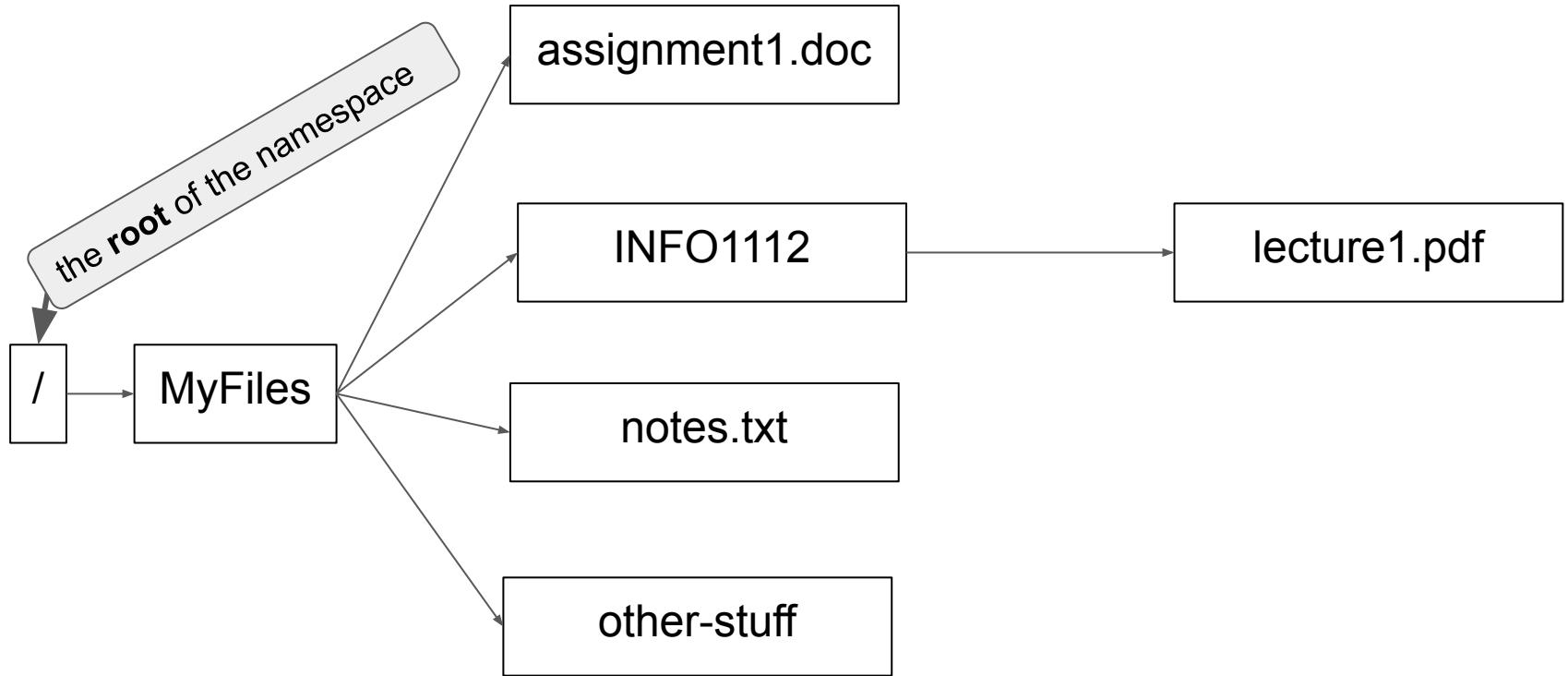
The names and the structure of the file system are called the ***name space***

To specify a given file or directory in the Unix file system *name space* we write the ***path name*** of the file or directory.

For example: to specify the lecture1.pdf file we might write:

/MyFiles/INFO1112/lecture1.pdf

This tells us that "MyFiles" is a directory at the ***root*** of the file system and that "assignment1.doc" is contained in "MyFiles".



Operating Systems

There is a lot more to say about operating systems and file systems.

The "kernel" of the operating system is supplemented with a large number of other programs that provide various services. For example the window manager.

There are also internal modules in the kernel that manage different devices, file systems and networking. These are sometimes called "drivers".

Finally the operating system is also supported by many application programs that provide services such as email or wordprocessing.

Many of these will be covered in this course.

We acknowledge the tradition of custodianship and law of
the Country on which the University of Sydney campuses stand.
We pay our respects to those who have cared and continue to
care for Country.



INFO1112

Week 2 Whole Class Session

Dr Nazanin Borhan



Video Segments of this week

- > 2.1 Command line: the shell
shell variables, shell control structures, shell scripts, pipes
and redirection, common shell commands (programs)

- > 2.2 Processes
processes, user ids and file ownership, file access,
pipes, devices in the namespace, special "files"

Command line Shell

Giving value to variables:

NAME=Nazanin

Call the variable later with \$:

echo \$NAME

Expressions:

A=1

expr \$A + 1

A= `expr \$A + 1`

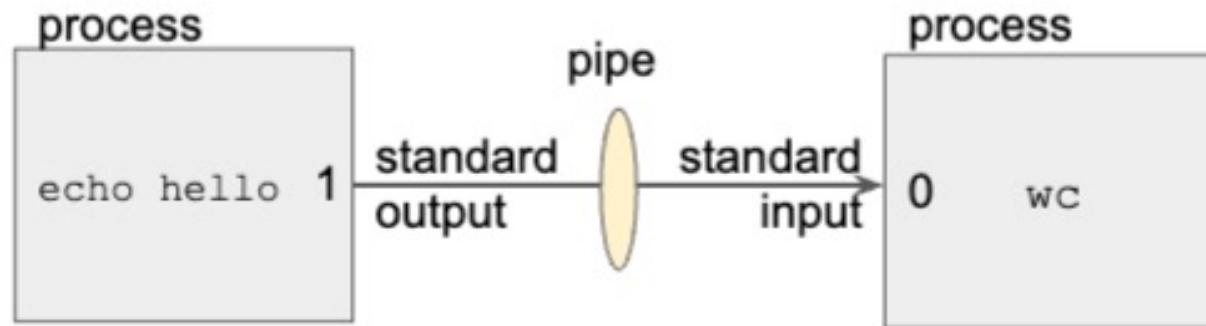
echo \$A

Pipe in shell

Pipe can take the output of one process and feed it as an input to the other process:

- Example:

```
echo hello | wc
```



Command Line Shell

Shell Scripts:

Specify what command to use to interpret the script



```
#!/bin/bash
```

```
echo HI
```

Some commands in Shell

- wc : how many lines/words/chars?
- cut : selecting parts of lines, columns, fields
- sort : on some field, column etc
- uniq : find unique lines
- grep : find lines matching a pattern
- sed : edit each line in a stream of lines
- diff, sdiff, comm : compare text files

Processes

```
$ ps al|more
F  UID   PID  PPID PRI  NI   VSZ   RSS WCHAN  STAT TTY      TIME COMMAND
0 1003 19208 19207  20   0 21208  4964 wait    Ss  pts/0      0:00 -bash
0 1003 24486 19208  20   0 27640  1452 -       R+  pts/0      0:00 ps al
0 1003 24487 19208  20   0  8236   748 pipe_w S+  pts/0      0:00 more
          ↑           ↑           ↑           ↑
          ↓           ↓           ↓           ↓
          process id  parent process id priority owner user id
          ↑           ↑           ↑           ↑
          ↓           ↓           ↓           ↓
          command
```

The ps command display information about processes

File Ownership

Different types of file Ownerships levels:

- User
- Group
- Others

```
$ ls -l
```

```
total 4
-rw-rw-r-- 1 info 1112 info 1112 6 Jul 26 12:46 myfile
```

Permissions

owner

Group

File Name

Demo



HOLLY
PRODUCT
DIRECTOR

Regular Expressions

- Regular expressions are a very important aspect of computing.
- A regular expression is a text pattern that is matched with a line of text.
- There are many variations on regular expressions but they all involve interpreting a pattern.

Regular expressions

abc	matches the string abc in the text
a.b	the dot means <i>any</i> character will match
^abc	match lines that start with abc
abc\$	match lines that end with abc
a[xy]c	[xy] match <i>either</i> x or y
ab*c	b* means zero or more occurrences of b

Demo



BREAK



Assignment 1



Assignment 1

- This assignment involves developing a program to check the access and permissions to files and change their permissions.
- Your program will read a text file that specifies what file paths to check. Then check the access for each file path and test their existence, readability, and executability of them and then change the permissions of the files

Structure of the program

Your program needs to contain these files:

- pcontrol.py
- filelist.txt

And create this file:

- output.txt

When you submit you will have these files:

- pcontrol.py
- tests/filelist.txt
- tests/exp_output.txt
- tests/test.sh. (optional, a shell script that helps run your tests)

Example of output of the program

- You have a file called file.ext in your current path
- You specify the name of this file in filelist.txt
- Your program check the attributes of the file and change the “Group Executable” permission on the file and the output this line:

file.ext Group Readable: True, Group Executable: True, Size: 10, Owner: NazaninBorhan,
Group: staff, last modified date: Jul 17 2022, last access date: Jul 17 2022

Changing permission

Changing Permissions with ‘chmod’

The syntax is expressed in a regular expression form:

[ugoa]*([-+=]([rwxXst]*|[ugo]))+|[-+=][0-7]+

For assigning permissions:

- chmod <groupName>+<permissionName> <fileName>
- E.g: chmod u+x file.ext

for removing permissions:

- chmod <groupName>-<permissionName> <fileName>
- E.g: chmod g-x file.ext

Demo



Working with Bash

- Create Bash Scripts
- Create Variables
- Use Variables
- Changing permissions in Bash

A cluster of five large, stylized question marks and exclamation points, cut from paper in various colors (brown, pink, white) and arranged in a loose, overlapping group. They are set against a solid brown background.

Questions?

T H E
E N D

We acknowledge the tradition of
custodianship and law of the Country on which
the University of Sydney campuses stand.
We pay our respects to those who have cared
and continue to care for Country.



THE UNIVERSITY OF
SYDNEY

INFO1112

Week 3 Whole Class Session

Dr Nazanin Borhan



Video Segments this week

- ›

3.1 Processes

processes as programs, command line arguments and python,
systems calls to start and manage a process: fork, exec, wait

- ›

3.2 File systems

the namespace, file systems on mass storage,
API for file system access: open/close/read/write/create/delete
mounting a file system or service in the namespace,
system services, user file systems

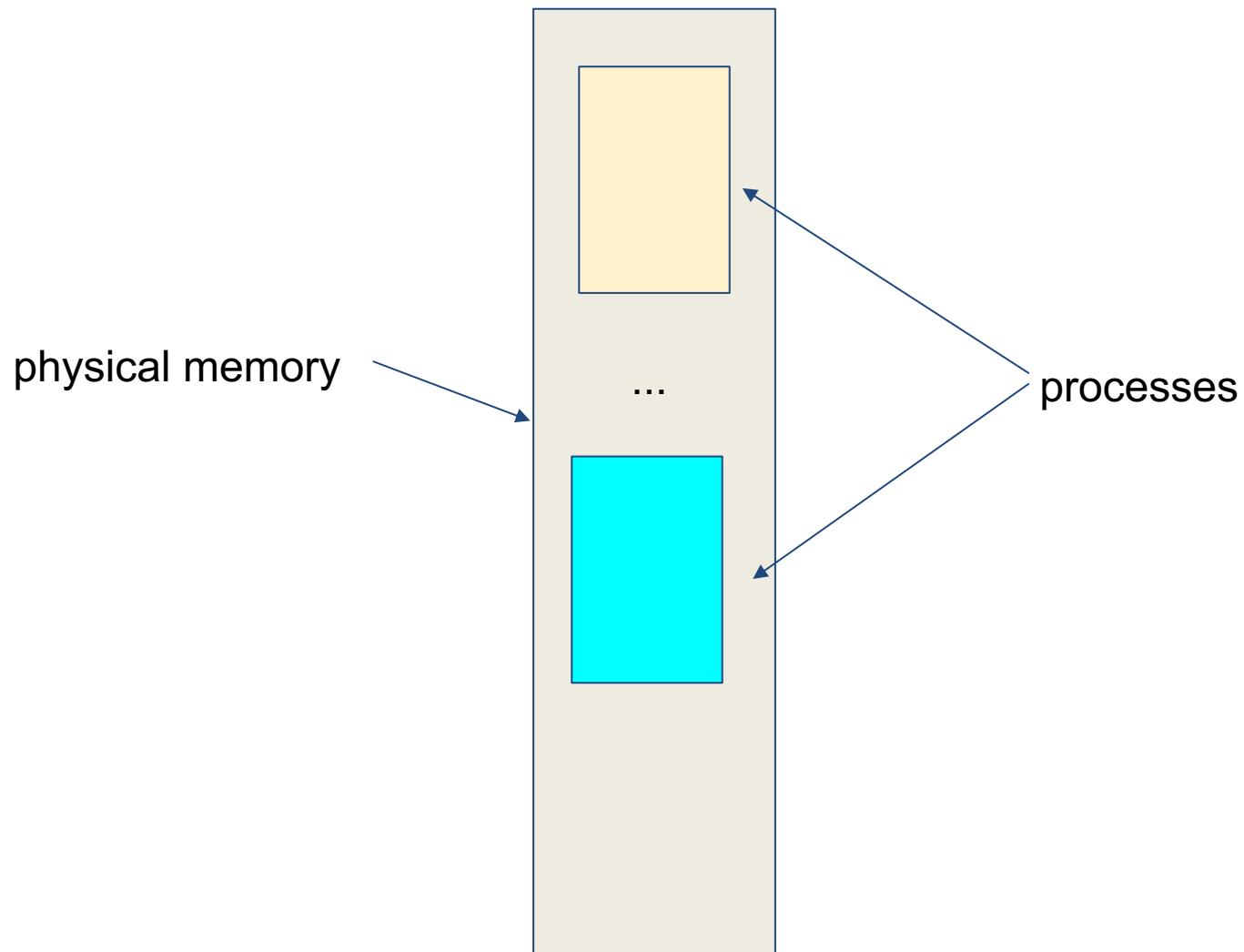
Processes

In this week we see how to start and stop a process and communicate between processes.

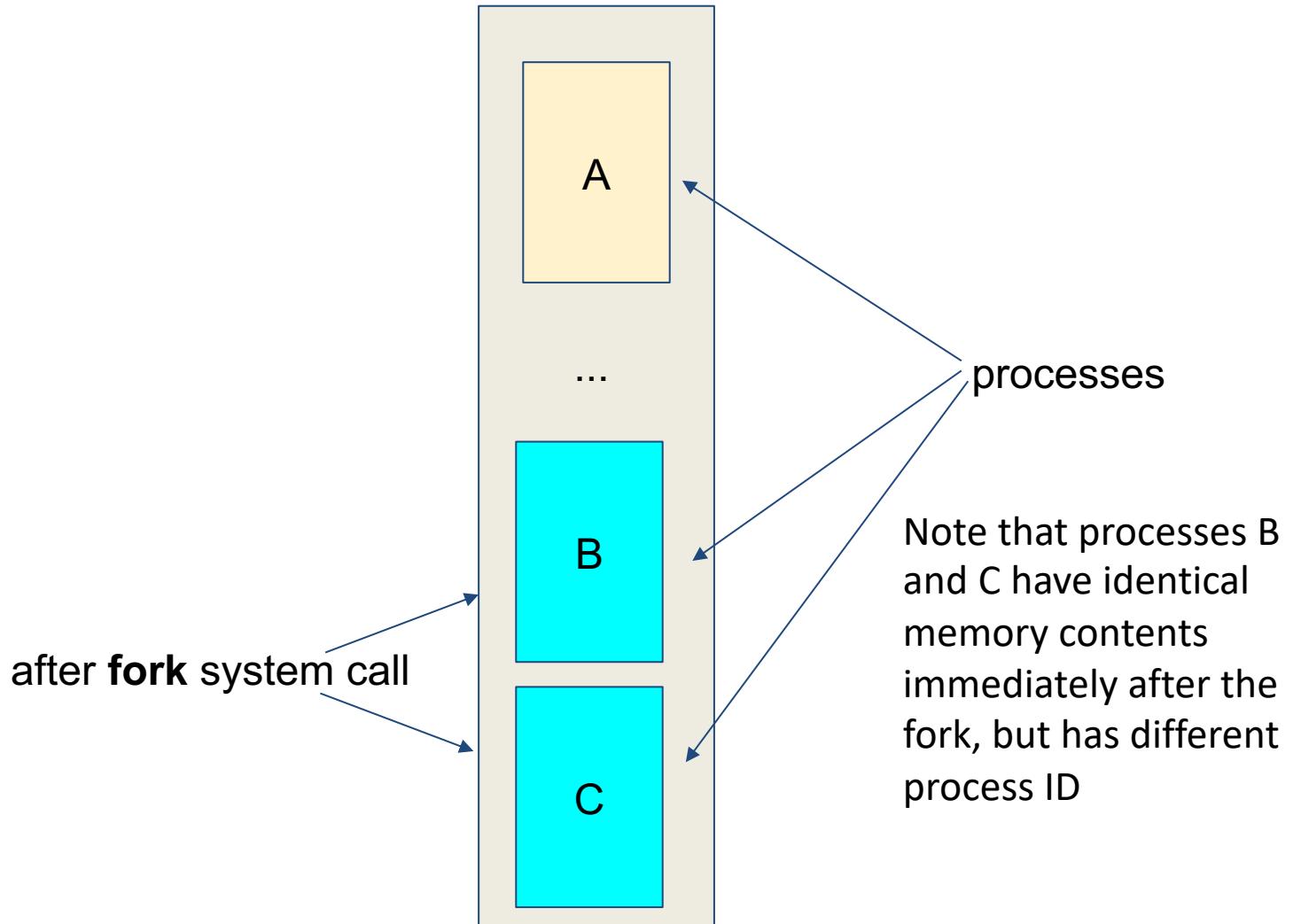
Unix **system calls** manages the processes, and the main system calls for managing processes are:

- *fork*
- *exec* (has many variations)
- *wait*
- *kill*

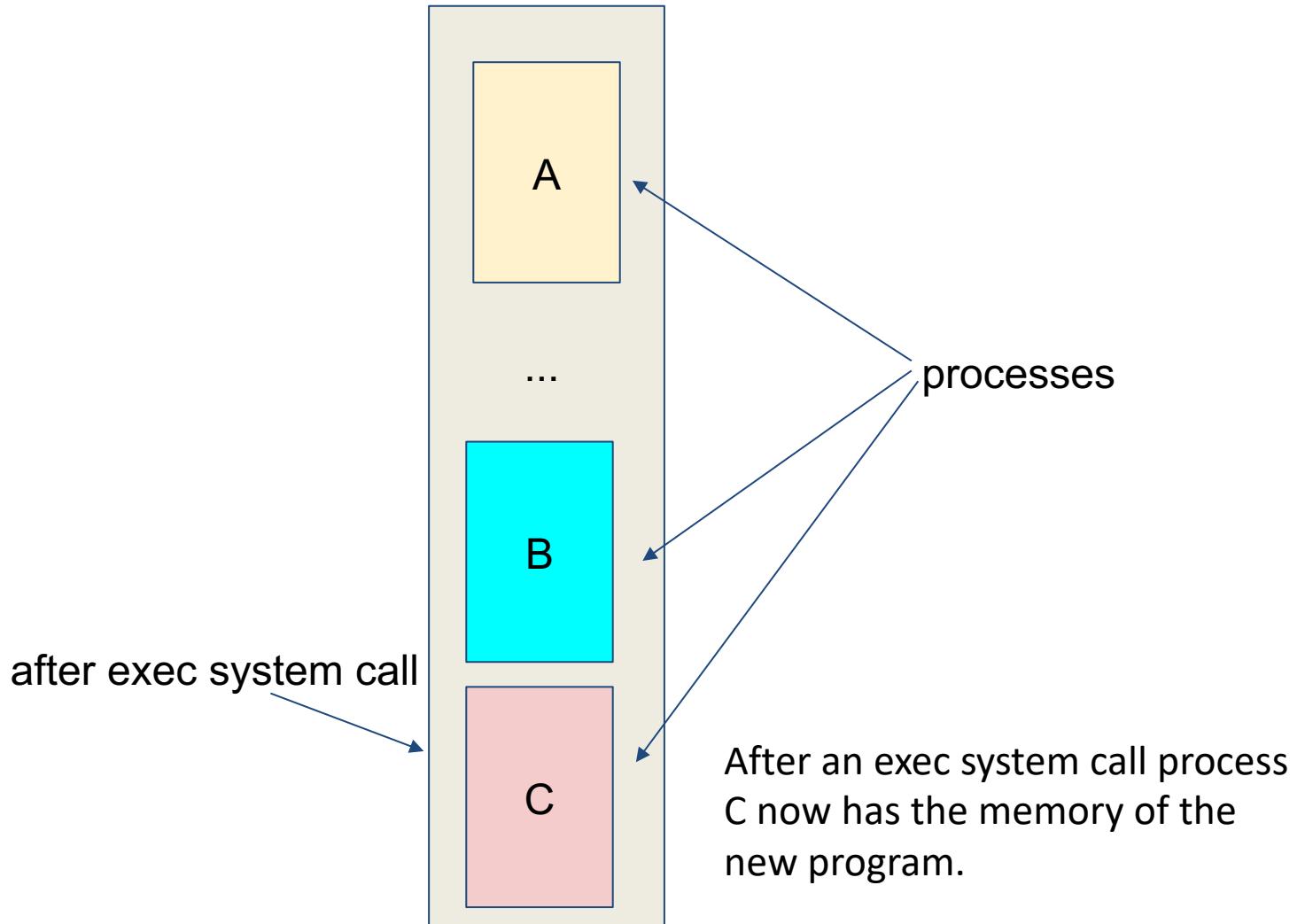
Processes in memory



fork system call



exec system call



Python fork/exec

```
pid = os.fork()
if pid == 0:
    print ("\tHi! I'm the child process.")
    print ("\tsleeping for 5 seconds....")
    time.sleep(5)
    print ("\t... exit with status 99.")
    sys.exit(99)
elif pid == -1:
    print ("yikes! fork failed!")
    sys.exit(1)
else:
    print ("I'm the parent process")
    print ("waiting for child with PID", pid)
    wval = os.wait()
    print ("wait over! process ID was ")
    print (wval[0], "exit status",wval[1]>>8)
```

Python fork/exec/wait

parent process

```
pid = os.fork()
if pid == 0:
    print ("\tHi! I'm the child process.")
    print ("\tsleeping for 5 seconds....")
    time.sleep(5)
    print ("\t... exit with status 99.")
    sys.exit(99)
elif pid == -1:
    print ("yikes! fork failed!")
    sys.exit(1)
else: ←
    print ("I'm the parent process")
    print ("waiting for child with PID", pid)
    wval = os.wait()
    print ("wait over! process ID was ")
    print (wval[0], "exit status",wval[1]>>8)
```

child process

```
pid = os.fork()
if pid == 0: ←
    print ("\tHi! I'm the child process.")
    print ("\tsleeping for 5 seconds....")
    time.sleep(5)
    print ("\t... exit with status 99.")
    sys.exit(99)
elif pid == -1:
    print ("yikes! fork failed!")
    sys.exit(1)
else:
    print ("I'm the parent process")
    print ("waiting for child with PID", pid)
    wval = os.wait()
    print ("wait over! process ID was ")
    print (wval[0], "exit status",wval[1]>>8)
```

Code to try from segment 1

```
pid = os.fork()
if pid == 0:
    print ("\tHi! I'm the child process.")
    print ("\tl'm going to sleep for 5 seconds....")
    time.sleep(5)
    print ("""\t... and now start the command "echo 'This is the child""""")
    os.execl("/bin/echo", "echo", "This is the child")
    print ("oops! It looks like exec failed")
    sys.exit(1)
elif pid == -1:
    print ("yikes! fork failed!")
    sys.exit(1)
else:
    print ("I'm the parent process and I'm waiting for the child with process ID ", pid)
    wval = os.wait()
    print ("wait over! Child process ID: ", wval[0], " exit status:", wval[1]>>8)
    print("The parent process ID is", (os.getpid()))
print ("finished.")
```

Demo

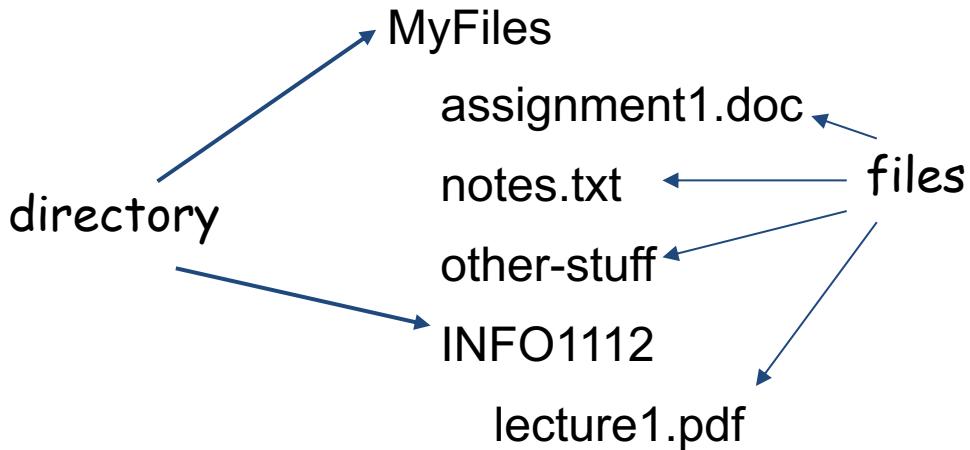


Name-space: files, devices and services

- What is the "name-space"?
- How does it relate to files, devices and services?
- How does this work?

Name Space

A name space is a set of identifiers (names) arranged in a hierarchy that designate or identify an object (file) or a service. A file system is a set of file objects and names in the name space refer to the files and directories (a directory if a special file that contains file names)



For file-systems stored on mass storage (disk), the meta-data is stored on the disk.

The meta-data includes information about the owner, group and permissions.

When you open a file with a pathname in the namespace (eg /MyFiles/INFO1112/ lecture1.pdf) the operating system traverses the path, works out which file system is responsible and hands the path to that file system to handle the open call.

The mount command

There is also a data-structure that says what system driver (program) is responsible for providing the file data. We can investigate this with the mount command.

```
$ mount
```

```
mass storage device containing the file system
```

place in the namespace to mount the file system

type of the file system

parameters of the file system

```
/dev/sda1 on /boot type ext2 (rw, relatime, barrier, user_xattr,  
acl,stripe=4)
```

Shell if statement

```
if test-commands; then  
    consequent-commands;
```

General form:

```
[elif more-test-commands; then  
    more-consequents;  
[else alternate-consequents;  
fi
```

```
if [ $NAME == Bob ]  
then  
    echo name is ok  
elif [ $NAME == Alice ]  
then  
    echo name is good  
else  
    echo Wrong name  
fi
```

Example of if statement in Shell

```
if [ $UID -eq 0 ]
then
    echo 'You are root!'
elif [ $UID -eq 504 ]
then
    echo 'You are user, Welcome!'
else
    echo 'You are not Welcome!'
fi
```

AND and OR in if statement with: && , ||

```
if [[ $UID -eq 504 && $name == bob ]]
Then
    echo you are welcome here
fi
```

Break



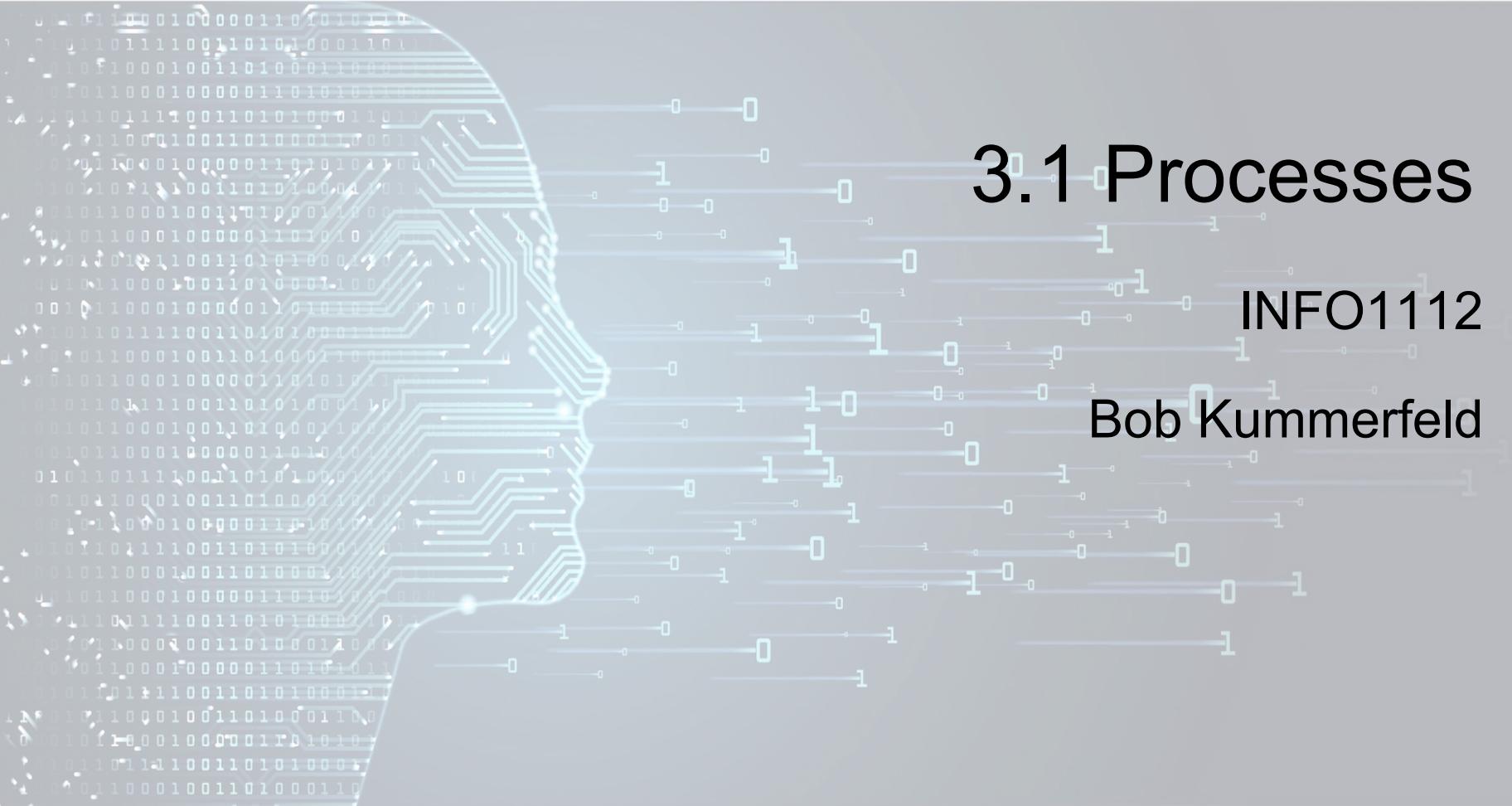
Demo With Python



Questions?



T1 H4 E1
E1 N1 D2



3.1 Processes

INFO1112

Bob Kummerfeld

Processes

So far you know that a process is a program with its code and data stored in the memory along with some *metadata* that shows who the owner is (UID), what open files it has (file descriptors) etc

A process may be executing, waiting for some data from a device or pipe, or sleeping for some other reason.

Now we look at how we start and stop a process, as well as how we communicate between processes.

From programs to processes

A program is stored on the persistent storage (eg disk) in a file. When a program is started, the operating system reads the file and generates a memory image that contains code and data. While doing that it will also read and incorporate any functions you use from the library. The image is read into some free memory and the associated metadata structures are created (eg standard input/out/error file descriptors etc). Lastly the operating system starts the process executing.

Starting a parallel process with the shell

When you type a command to the shell it is executed and the shell waits for it to finish. This is the normal case.

The shell also has a syntax for running a process in parallel.

```
$ echo hello &  
[1] 32466  
$ hello
```

this indicates you want the command
run in parallel with the shell

process ID of child

printed by the echo

System calls for processes

Unix has a number of ***system calls*** that manage processes. Remember that a system call is like a function call to the operating system, except the privilege level switches from the user to the superuser and execution continues in the operating system kernel code.

The main system calls for managing processes are:

`fork`

`exec` (has many variations)

`wait`

The `fork` system call

`fork` creates a new process (the ***child***) where the memory image and the metadata are an exact copy of the process that called `fork` (the ***parent***), except for:

- the child process has a new process ID
- the child process has the parent process ID of the process that called `fork`
- various values of resource metadata (eg cpu time used) are reset

Note that this means the child **shares the same files** initially.

After the `fork` call, both the child and the parent continue execution as if the `fork` function had returned, except that in the child `fork` returns 0 and in the parent `fork` returns the process ID of the child.

exec system call

After the fork system call we have a copy of the parent process running as the child. This is ok but it is more useful to be able to run another program as the child. We can do this with the `exec` system call in the child.

`exec` comes in several versions. They all specify what program to execute (pathname etc), some specify program arguments and some specify the shell ***environment*** variables.

`exec` is a simple transfer of control. The calling process is completely **replaced** by the new program.

The wait system call

`wait` suspends execution of the process that calls the `wait` function until one of its child processes exits.

`wait` returns the process ID of the child that has exited

If a child process terminates and the parent is not "waiting" for it, then it enters the "zombie" state until the parent process waits for it.

Starting a Python process

When a Python program is started from the shell, the arguments are passed in the list `argv` available in the `sys` module.

```
#!/usr/bin/python

import sys

print ("first argument is ", sys.argv[1])
```

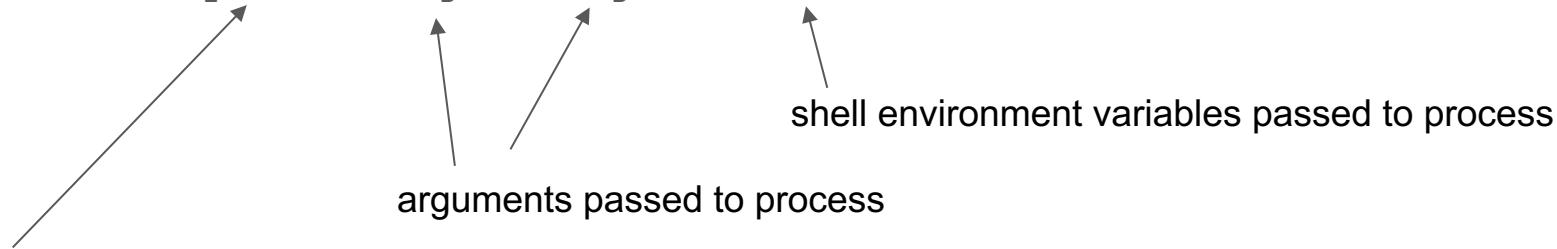
Starting a process *from* Python

Python has an interface to the system calls fork/exec/wait so we can create new processes within python, start another program and wait for it to terminate.

The `os` module contains methods (functions) for many of the Unix system calls.

`os.fork()`

`os.execle(path, arg0, arg1, ...env)`



path to file containing the program

arguments passed to process

shell environment variables passed to process

Python fork/exec



```
pid = os.fork()
if pid == 0:
    print ("\tHi! I'm the child
process.")
    print ("\tsleeping for 5
seconds....")
    time.sleep(5)
    print ("\t... exit with status 99.")
    sys.exit(99)
elif pid == -1:
    print ("yikes! fork failed!")
    sys.exit(1)
else:
    print ("I'm the parent process")
    print ("waiting for child with PID", pid)
    wval = os.wait()
    print ("wait over! process ID was ")
    print (wval[0], "exit status", wval[1]>>8)
```

parent process

```
pid = os.fork() ←
if pid == 0:
    print ("\tHi! I'm the child
process.")
    print ("\tsleeping for 5
seconds....")
    time.sleep(5)
    print ("\t... exit with status 99.")
    sys.exit(99)
elif pid == -1:
    print ("yikes! fork failed!")
    sys.exit(1)
else:
    print ("I'm the parent process")
    print ("waiting for child with PID", pid)
    wval = os.wait()
    print ("wait over! process ID was ")
    print (wval[0], "exit status", wval[1]>>8)
```

child process

```
pid = os.fork() ←
if pid == 0:
    print ("\tHi! I'm the child
process.")
    print ("\tsleeping for 5
seconds....")
    time.sleep(5)
    print ("\t... exit with status 99.")
    sys.exit(99)
elif pid == -1:
    print ("yikes! fork failed!")
    sys.exit(1)
else:
    print ("I'm the parent process")
    print ("waiting for child with PID", pid)
    wval = os.wait()
    print ("wait over! process ID was ")
    print (wval[0], "exit status", wval[1]>>8)
```

parent process

```
pid = os.fork()
if pid == 0:
    print ("\tHi! I'm the child
process.")
    print ("\tsleeping for 5
seconds....")
    time.sleep(5)
    print ("\t... exit with status 99.")
    sys.exit(99)
elif pid == -1:
    print ("yikes! fork failed!")
    sys.exit(1)
else:
    print ("I'm the parent process")
    print ("waiting for child with PID", pid)
    wval = os.wait()
    print ("wait over! process ID was ")
    print (wval[0], "exit status",wval[1]>>8)
```

child process

```
pid = os.fork()
if pid == 0: ←
    print ("\tHi! I'm the child
process.")
    print ("\tsleeping for 5
seconds....")
    time.sleep(5)
    print ("\t... exit with status 99.")
    sys.exit(99)
elif pid == -1:
    print ("yikes! fork failed!")
    sys.exit(1)
else:
    print ("I'm the parent process")
    print ("waiting for child with PID", pid)
    wval = os.wait()
    print ("wait over! process ID was ")
    print (wval[0], "exit status",wval[1]>>8)
```

parent process

```
pid = os.fork()
if pid == 0:
    print ("\tHi! I'm the child
process.")
    print ("\tsleeping for 5
seconds....")
    time.sleep(5)
    print ("\t... exit with status 99.")
    sys.exit(99)
elif pid == -1:
    print ("yikes! fork failed!")
    sys.exit(1)
else:
    print ("I'm the parent process")
    print ("waiting for child with PID", pid)
    wval = os.wait()
    print ("wait over! process ID was ")
    print (wval[0], "exit status", wval[1]>>8)
```

child process

```
pid = os.fork()
if pid == 0:
    print ("\tHi! I'm the child
process.")
    print ("\tsleeping for 5
seconds....")
    time.sleep(5) ↑
    print ("\t... exit with status 99.")
    sys.exit(99)
elif pid == -1:
    print ("yikes! fork failed!")
    sys.exit(1)
else:
    print ("I'm the parent process")
    print ("waiting for child with PID", pid)
    wval = os.wait()
    print ("wait over! process ID was ")
    print (wval[0], "exit status", wval[1]>>8)
```

```
pid = os.fork()
if pid == 0:
    print ("\tHi! I'm the child process.")
    print ("\tI'm going to sleep for 5 seconds....")
    time.sleep(5)
    print ("""\t... and now start the command "echo 'This is the child'""")
    os.execl("/bin/echo", "echo", "This is the child")
    print ("oops! It looks like exec failed")
    sys.exit(1)
elif pid == -1:
    print ("yikes! fork failed!")
    sys.exit(1)
else:
    print ("I'm the parent process and I'm waiting for the child with
process ID ", pid)
    wval = os.wait()
    print ("wait over! process ID: ", wval[0], " exit status:", wval[1]>>8)
print ("finished.")
```

Stopping a process

Processes can exit voluntarily or they can be terminated by another process using the *kill* system call.

Kill operates by sending a *signal* to the process (more about signals in the next lecture) with the default signal being TERM to terminate.

Signals can only be sent to another process if both the sender process and the receiver process have the same UID or if the sender is the superuser.

There is a `kill` command that can be used from the shell.

Summary

In this segment we have covered:

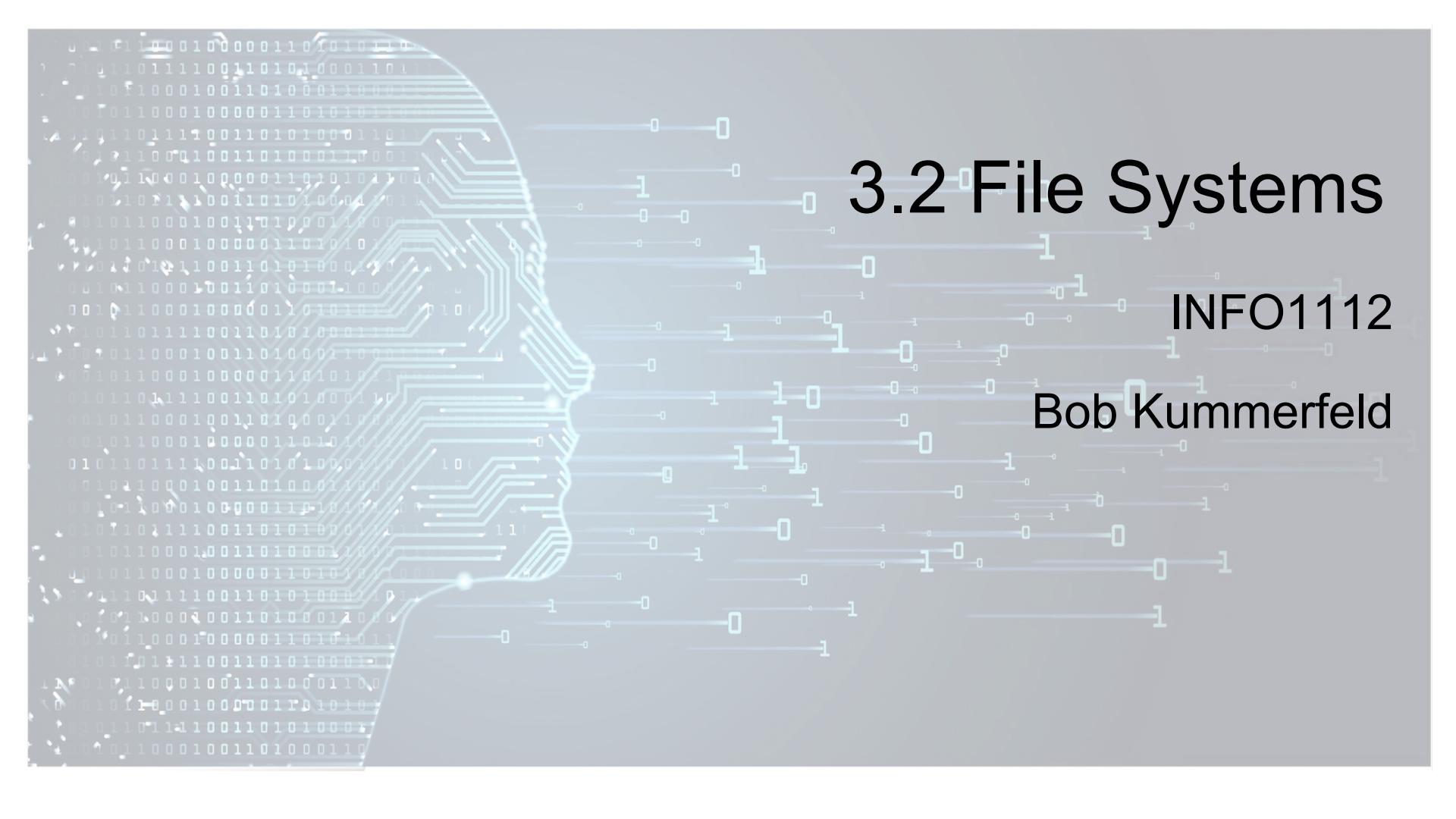
processes as programs

command line arguments and python

starting a process to run in parallel in the shell

systems calls to start and manage a process: fork, exec, wait

Python example of starting a process



3.2 File Systems

INFO1112

Bob Kummerfeld

File System (or namespace)

As you've seen, Unix has a tree structured **namespace** that allows us to access files stored on mass storage, devices and system status details.

The file system is a way of implementing access to objects in this namespace. The system status we saw in another segment is provided by a file system module in the kernel. More conventional file systems manage mass storage to store data files.

File Systems on mass storage

There are many ways to lay out the file data on mass storage. Both the file data and the file metadata (name, permissions etc) must be stored so the file is "persistent".

This means there are many implementations of file systems, optimised for different features such as speed, reliability/redundancy etc

For example: FAT16, FAT32, exFAT in the Windows operating system, ext2, ext4, Btrfs, ZFS etc for the Linux operating system.

Linux distributions have a default file system.

The important point is that all the Unix file systems provide the same interface.

File systems interface

A file system module for the operating system provides a standard interface to the namespace (creat/delete/open/close/read/write....) and as long as the module implements that it can provide any type of data in response.

If you are accessing a normal file on mass storage this usually means the data from the file. But a file system module can return other information and that's how the system information is provided via a file system "mounted" on /proc

File system mounting

Unix has a system call "mount" that grafts a file system into the namespace tree at some point. There is also a mount command that runs the mount system call so you can mount new file systems from the command line.

File system mounting

Unix has a system call "mount" that grafts a file system into the namespace tree at some point. There is also a mount command that runs the mount system call so you can mount new file systems from the command line.

Typing `mount` by itself will give you a list of mounted file systems. This can be long since many services are implemented this way.

```
$ mount
/dev/sda1 on /boot type ext2 (rw, relatime, barrier, user_xattr, acl,stripe=4)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
```

mass storage device containing the file system

place in the namespace to mount the file system

type of the file system

parameters of the file system

```
/dev/sda1 on /boot type ext2 (rw, relatime, barrier, user_xattr, acl,stripe=4)
```

When you open a file with a pathname in the namespace (eg /home/info1112/SomeFiles) the operating system traverses the path, works out which file system is responsible and hands the path to that file system to handle the open call.

User File Systems

The approach of implementing file systems with standard interfaces and then grafting them into the name space gives great flexibility.

There is even a file system module that allows the file system implementation to run in user space called "fuse"

This means normal users/developers can easily implement a new service and put it into the namespace.

Summary

In this segment we covered:

the namespace

file systems on mass storage

the basic API for file system access: open/close/read/write/creat/delete

attaching a new subtree of the namespace: "mounting"

system services in the namespace

user file systems

We acknowledge the tradition of
custodianship and law of the Country on which
the University of Sydney campuses stand.
We pay our respects to those who have cared
and continue to care for Country.



THE UNIVERSITY OF
SYDNEY

INFO1112

Week 4 Whole Class Session

Dr Nazanin Borhan



Video Segments this week:

- › **4.1 Memory Management**
memory layout of processes, virtual memory, paging and page tables
- › **4.2 Scheduling Processes**
deciding what process to execute, process priorities
- › **4.3 Boot Sequence**
the sequence of steps that occur when the system starts

Lab session this week:

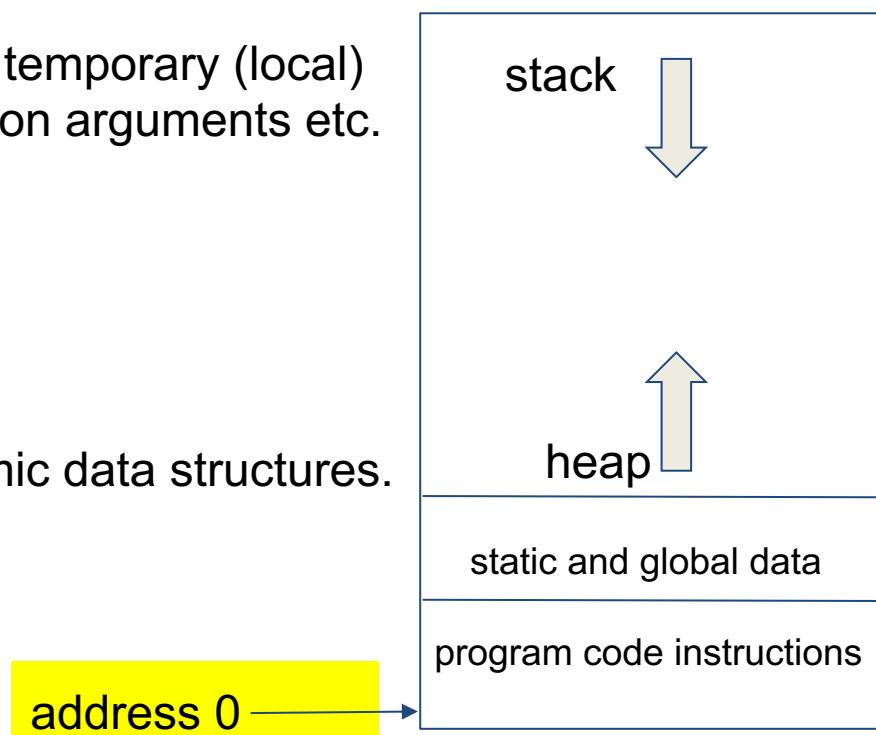
- Investigating process memory (ps)
- Process priority (nice)
- The boot sequence

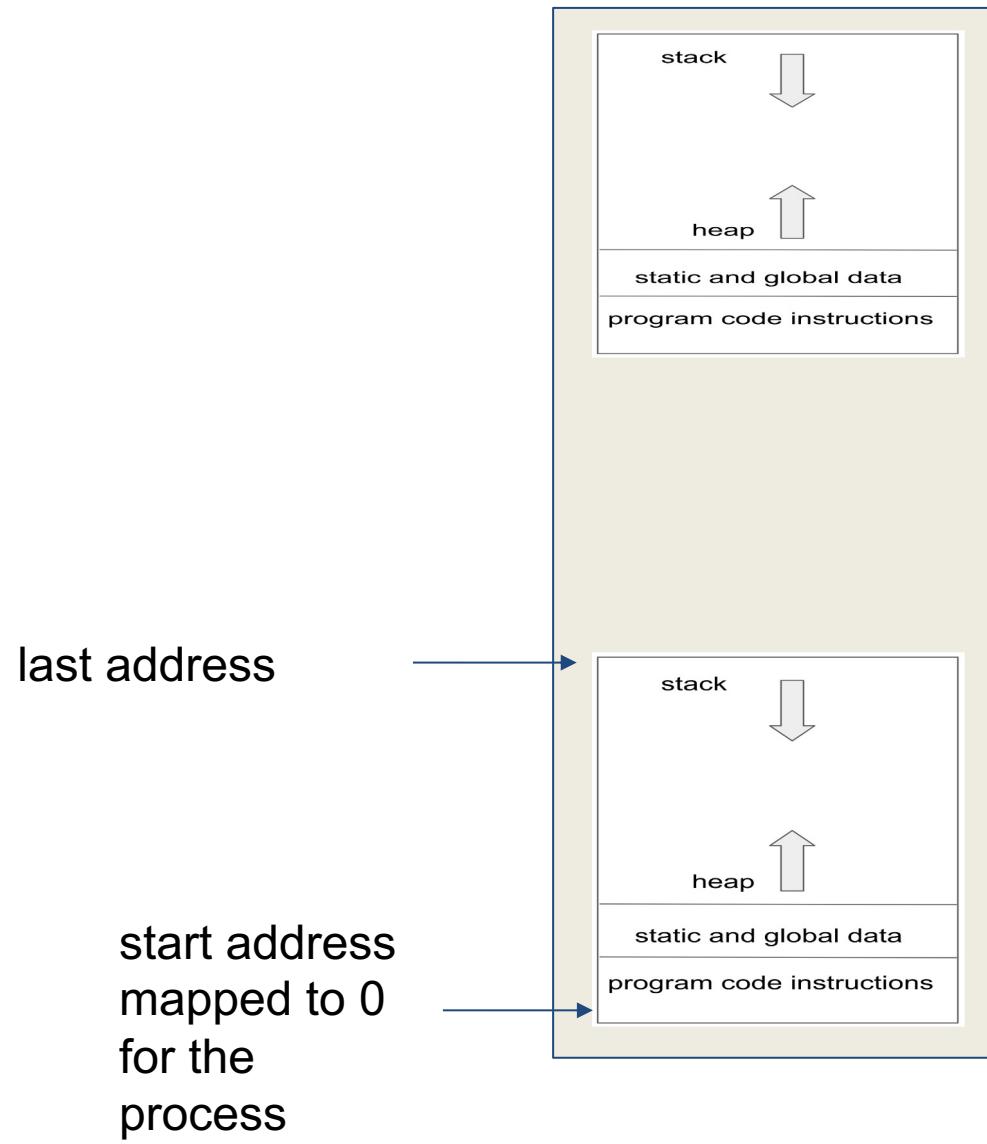
Programs in memory

- When a program is ready to run as a process it is stored in memory in a particular layout:

The stack is for temporary (local) variables, function arguments etc.

The heap is for dynamic data structures.





ps and nice

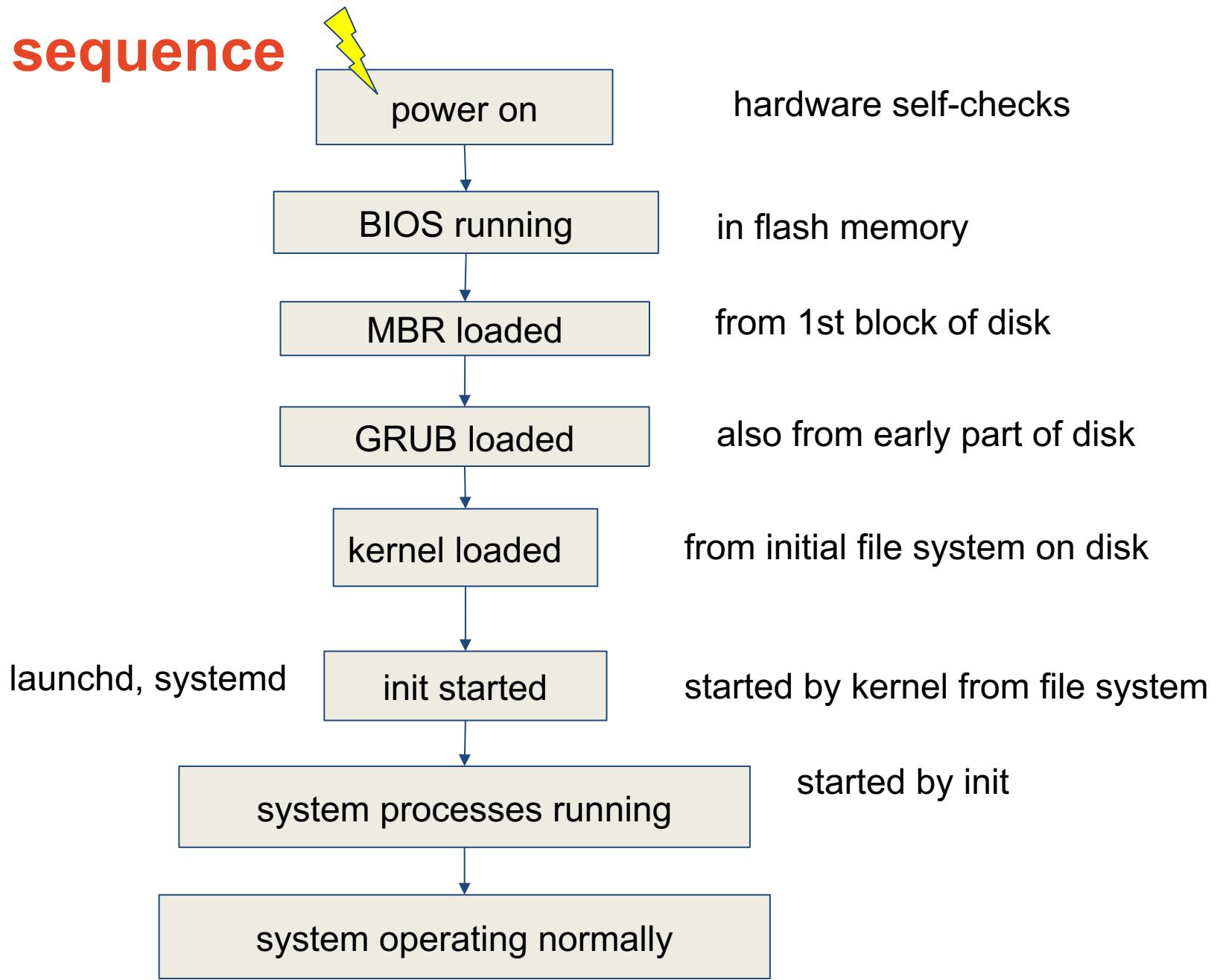
```
$ ps 1
F   UID   PID   PPID  PRI  NI    VSZ     RSS WCHAN   STAT TTY      TIME COMMAND
0 1001  3577  3574   20    0 21288  5104 wait    Ss    pts/1  0:00 -bash
0 1001 12728  3577   20    0 27640  1532 -       R+    pts/1  0:00 ps 1
0 1001 29861 29860   20    0 21288  5092 wait_w  Ss+   pts/0  0:00 -bash
```

```
$ nice ps 1
F   UID   PID   PPID  PRI  NI    VSZ     RSS WCHAN   STAT TTY      TIME COMMAND
0 1001  3577  3574   20    0 21288  5104 wait    Ss    pts/1  0:00 -bash
0 1001 12729  3577   30   10 27640  1436 -       RN+   pts/1  0:00 ps 1
0 1001 29861 29860   20    0 21288  5092 wait_w  Ss+   pts/0  0:00 -bash
$
```

Demo



Boot sequence



Break



Git

Poll

**Suggested tutorial:
<https://www.atlassian.com/git/tutorials>**

Git

- **Git is a distributed source code management (revision) system.**
- **It is distributed.**
- **Git maintains a database of changes to your source code text.**
- **Git branches, Pull, Push, Merge.**
- **Git hub is the most popular centralized git, and university of Sydney has its own instances of github and you can store your source code on university of Sydney's git.**

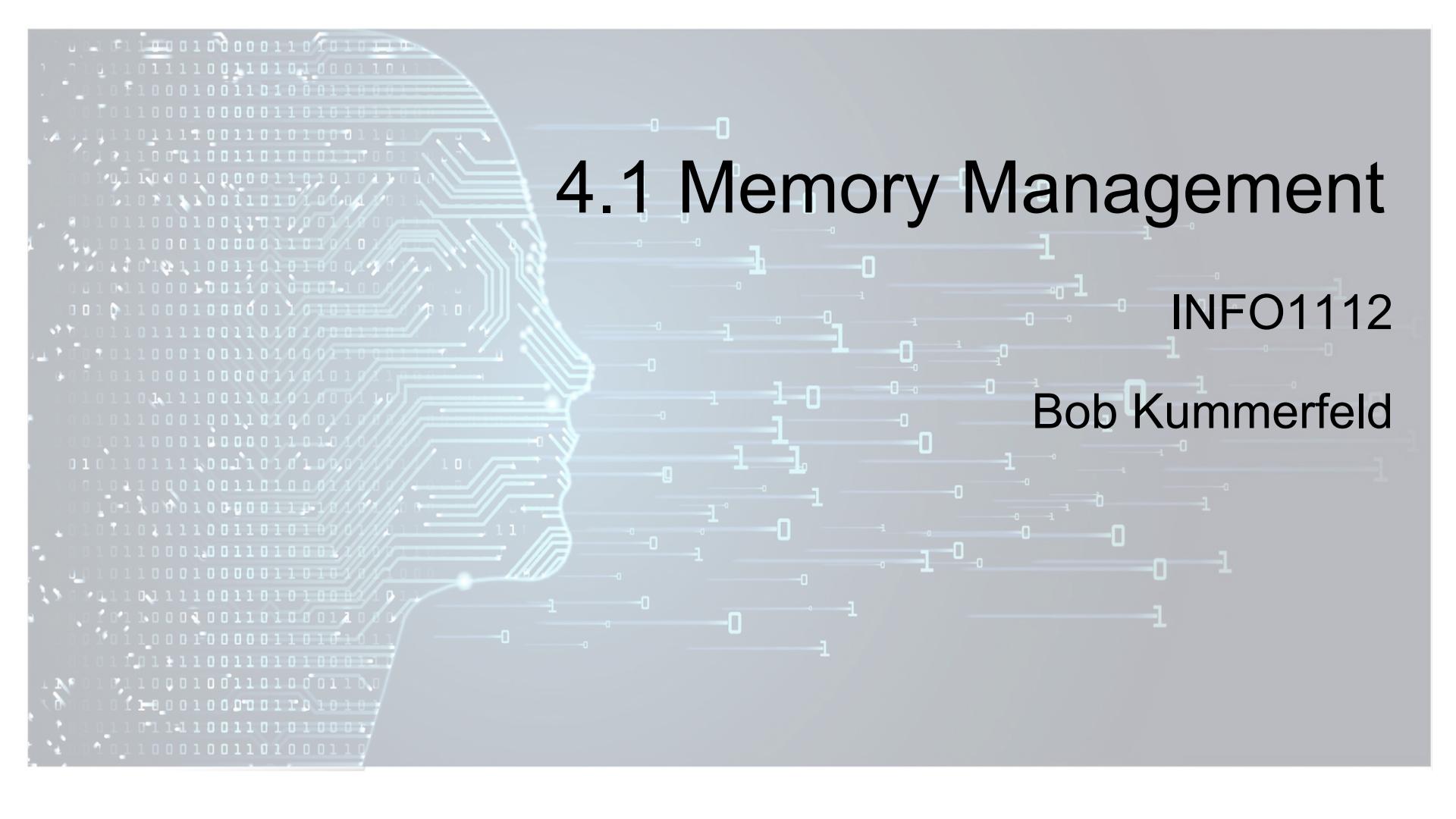
Demo
"Git"



Questions?



T1 H4 E1
E1 N1 D2



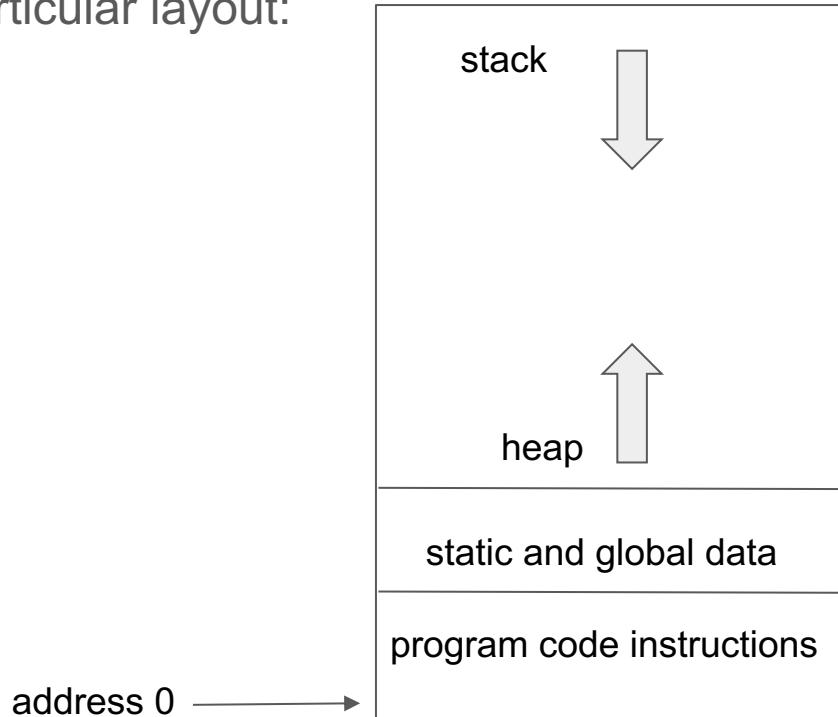
4.1 Memory Management

INFO1112

Bob Kummerfeld

Programs in memory

When a program is ready to run as a process it is stored in memory in a particular layout:

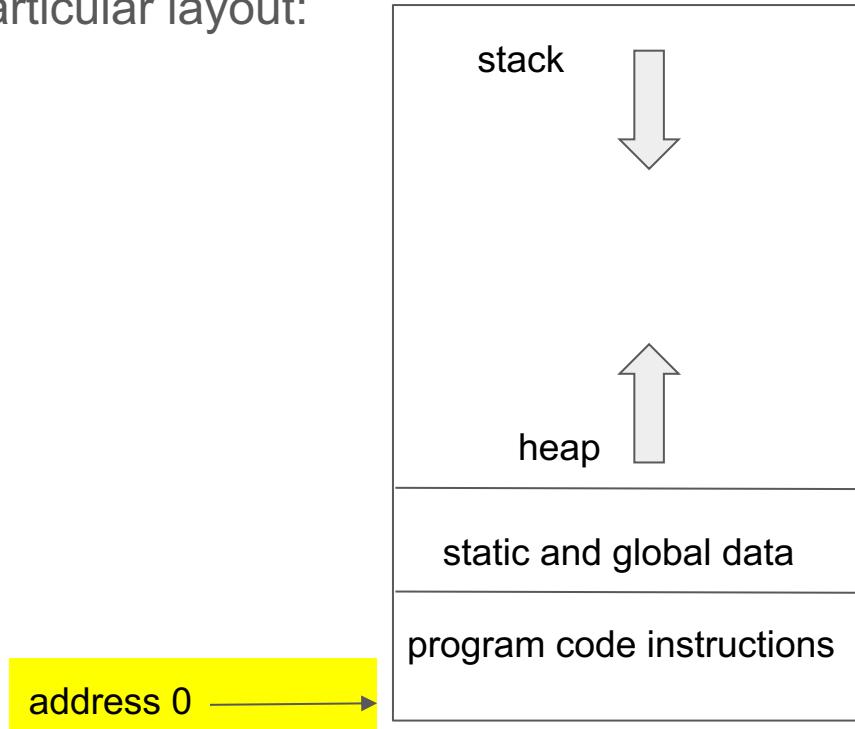


The stack is for temporary variables.

The heap is for data structures.

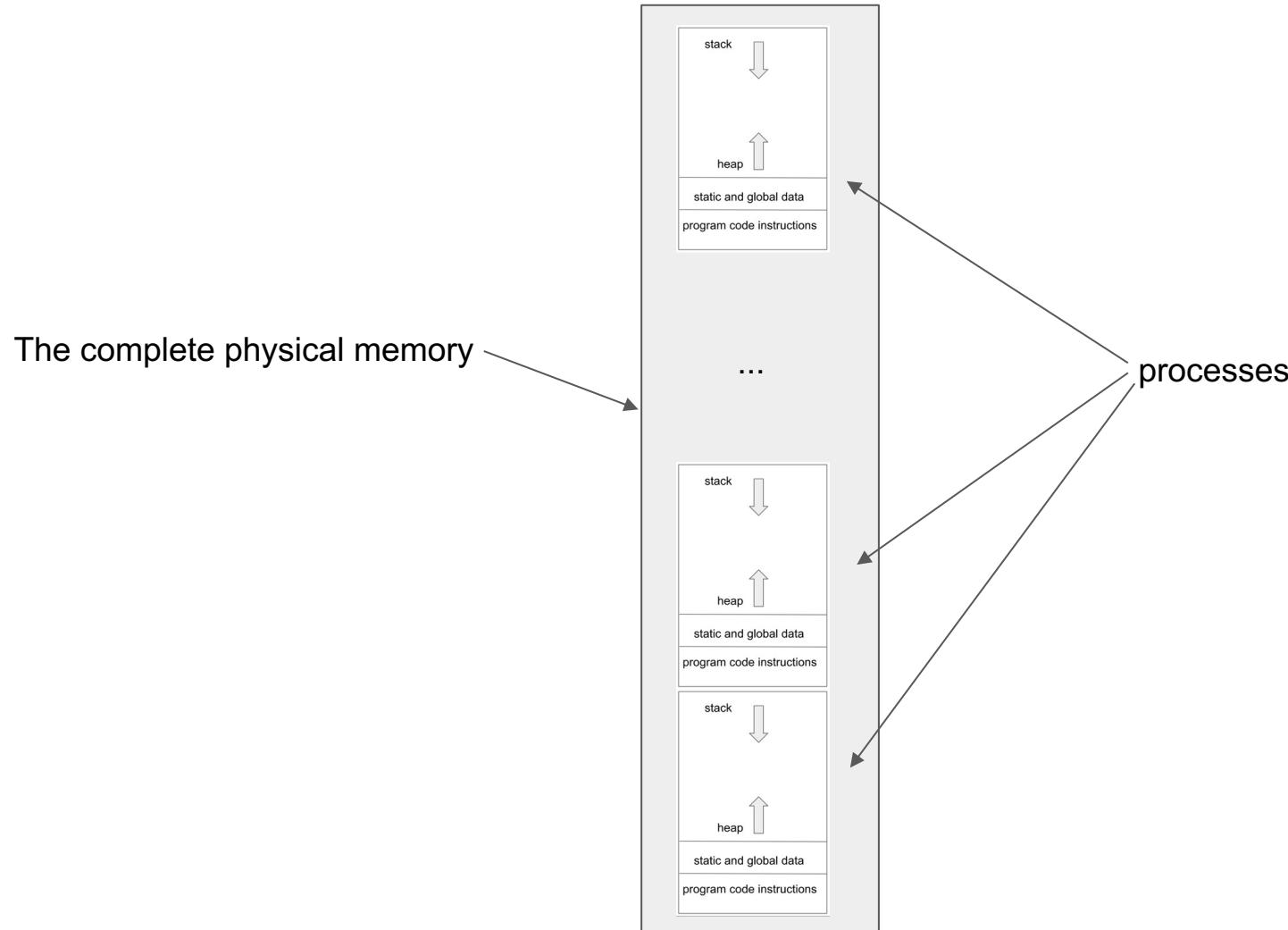
Programs in memory

When a program is ready to run as a process it is stored in memory in a particular layout:



The stack is for temporary variables.

The heap is for data structures.



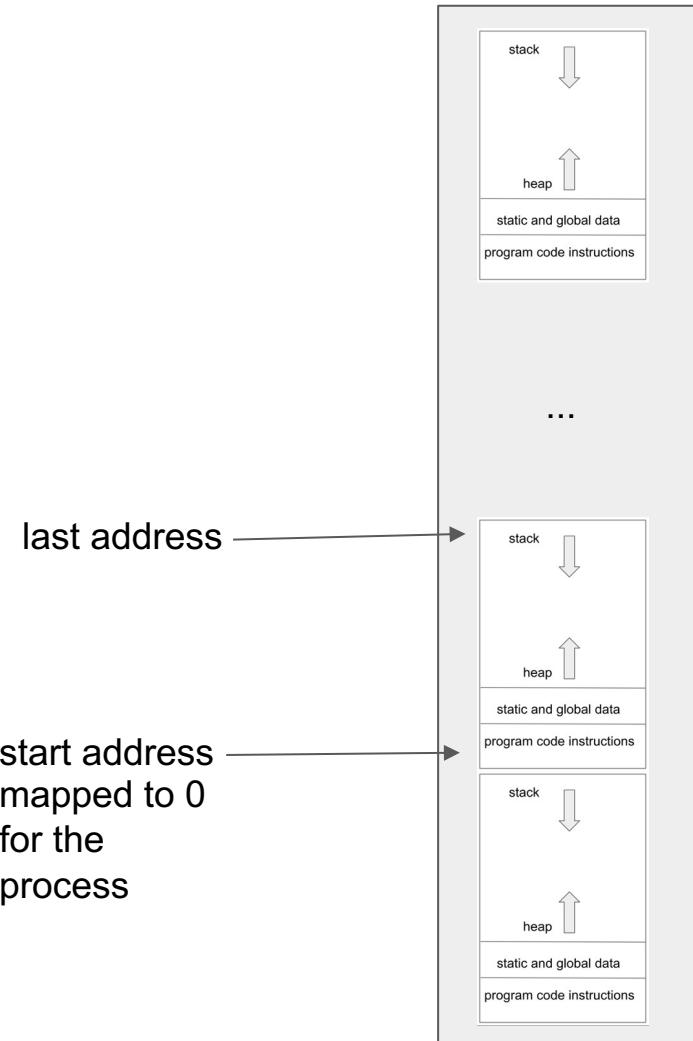
Memory Mapping

The problem with this is that there are many processes in the memory at the same time. How can they **all** be located at address zero?

This is achieved by hardware in the CPU that **remaps** the address space for each process when it is running.

For example, if a process memory image is stored starting at real address 1000, then the kernel of the operating system can use privileged instructions to set a special register that says "all references to addresses are offset by 1000" that is, a reference to address 0 will actually refer to address 1000.

In addition, there is a register that puts an upper limit on the addresses the process can use. This prevents the process code from accessing beyond its allotted memory area.



Virtual Memory

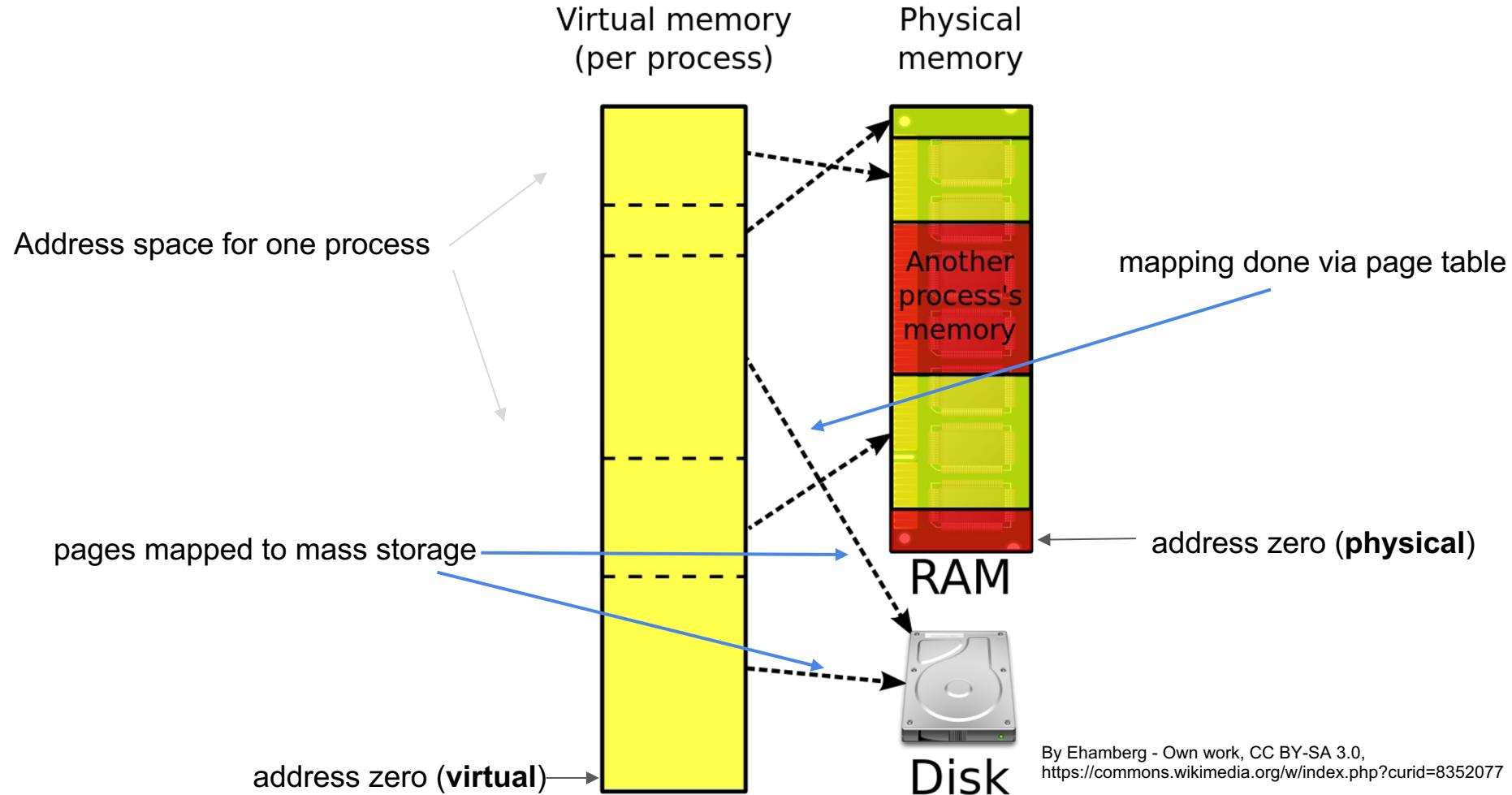
This simple memory mapping scheme was the early approach to memory management and protection. Now we use a more sophisticated (and complex) scheme that involves remapping smaller blocks of memory called "pages".

There is a table in the system that contains an entry for every page of the process. This table maps the virtual address of the page used by the process to the physical address in the main memory or an indication that the page is currently on mass storage. This is known as the ***page table***.

The kernel manages this table and moves pages to and from mass storage.

Page Table Example

Process ID	Process (Virtual) Address	Physical Memory Address	Mass Storage Address
1456	100000	12000000	0
1456	120000	0	32100
...
...



Virtual Memory

What happens if a program attempts to use memory that the page table has mapped to mass storage?

In that case, the access will cause an *interrupt* that will transfer control to the kernel. An interrupt is like a forced function call to the kernel that occurs between two instructions in such a way that the CPU state is saved and can be restored when the kernel restarts the process. In this way the process has no indication that it was interrupted.

When the kernel gains control, it will find the page on disk (stored in the *swap* area) and read it in to a free part of main memory. Then setup up the page table to point to the new physical location and return to the process.

Virtual Memory

If the kernel needs to read in a page from disk and discovers there is no free physical memory available, it has to first write an existing block out to the swap area to free up the memory.

If this happens too frequently, the system will become very slow and inefficient. This is because the mass storage is much much slower than RAM and it takes a long time to position a disk and read the page into memory.

Virtual Memory

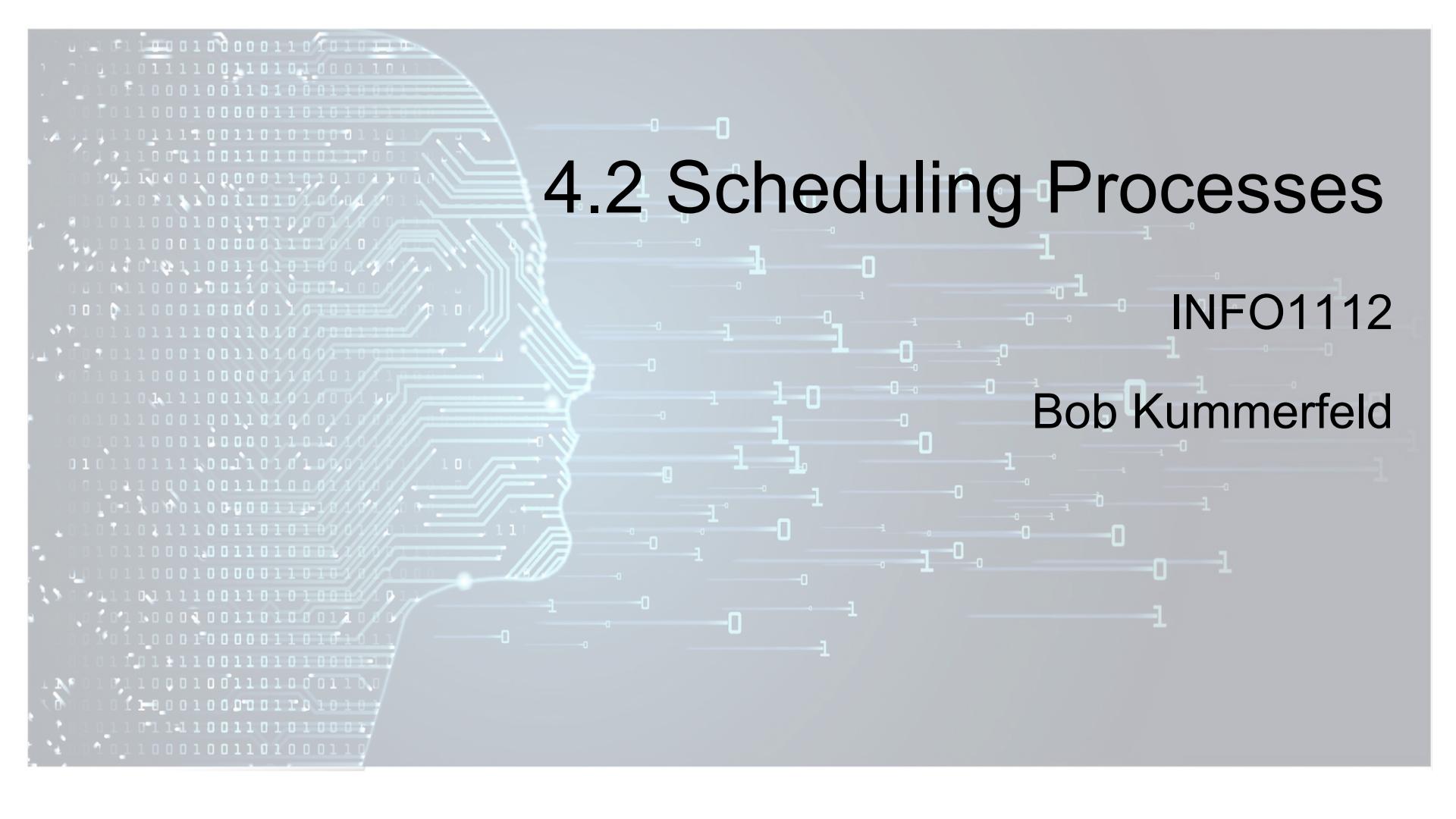
Virtual memory has a number of neat optimisations.

- "shared memory" means that some processes have pages that are mapped to the same physical memory. In this way they can either share readonly data, or readonly code such as library functions. Or share writable memory for sharing data (careful!).
- "copy on write" means that when a process forks, it shares physical memory pages between the processes until one or other wants to change (write) a page. At that point a writable copy is made.

Summary

In this segment we covered:

- memory layout of processes
- simple memory mapping
- virtual memory
- paging and page tables
- paging/swapping to mass storage



4.2 Scheduling Processes

INFO1112

Bob Kummerfeld

Scheduling

As we saw earlier, on a single CPU system, only one process can execute at a time. Other processes in the memory are sleeping - waiting for some resource to become available or waiting for their turn to use the CPU.

Usually the resource they are waiting for is usually some form of I/O: waiting for the data to be delivered from a peripheral such as a disk drive or a keyboard etc.

Sometimes the process can be waiting for the virtual memory system to bring in a page.

Eventually each process will need to execute and so need a turn using the CPU.

Deciding which process gets to run next is called ***scheduling***.

Scheduler

The basic idea is that the scheduler maintains an ordered queue of processes waiting to run. The ordering is decided with some measure of priority.

The simplest scheduling algorithm is called "round robin" scheduling because it give each process an equal slice of the cpu time and just circulates around processes.

More sophisticated schedulers take into account a priority value and other factors.

A lot of research has been done on schedulers and there is a very large number of algorithms. Some of the earliest and important work was carried out here at USyd by Prof Judy Kay and Piers Lauder who developed an entire class of schedulers called "fair share" schedulers.

Schedulers and Linux

Processes in Linux have a priority value that can be seen on a "ps al" listing:

F	UID	PID	PPID	PRI	NI	VSZ	RSS	WCHAN	STAT	TTY	TIME	COMMAND
4	0			1		0	20	0	185248	5792 -	Ss ?	1:07

↑
priority

Scheduler and Linux

The priorities range from negative integers (the **highest** priority) to positive numbers (the **lowest** priority). A normal user process starts with priority 20.

You can set the priority of one of the processes you create using the "nice" command.

```
sudo nice -n N command
```

This will add N to the priority of the command when it runs.

From the manual entry:

NAME

nice -- execute a utility with an altered scheduling priority

SYNOPSIS

```
 nice [-n increment] utility [argument ...]
```

DESCRIPTION

nice runs utility at an altered scheduling priority. If an increment is given, it is used; otherwise an increment of 10 is assumed. The super-user can run utilities with priorities higher than normal by using a negative increment. The priority can be adjusted over a range of -20 (the highest) to 20 (the lowest).

Scheduler and Linux

```
$ ps l
F   UID   PID   PPID  PRI  NI   VSZ    RSS WCHAN   STAT TTY      TIME COMMAND
0  1001  3577  3574  20   0  21288  5104 wait    Ss    pts/1          0:00 -bash
0  1001 12728  3577  20   0  27640  1532 -       R+    pts/1          0:00 ps l
0  1001 29861 29860  20   0  21288  5092 wait_w  Ss+   pts/0          0:00 -bash
```

```
$ nice ps l
F   UID   PID   PPID  PRI  NI   VSZ    RSS WCHAN   STAT TTY      TIME COMMAND
0  1001  3577  3574  20   0  21288  5104 wait    Ss    pts/1          0:00 -bash
0  1001 12729  3577  30   10 27640  1436 -       RN+   pts/1          0:00 ps l
0  1001 29861 29860  20   0  21288  5092 wait_w  Ss+   pts/0          0:00 -bash
$
```

Example that will run "mycommand" at a lower priority:

```
$ nice -n 10 ./mycommand
```

Example that will attempt to run "mycommand" at a higher priority:

```
$ nice -n -10 ./mycommand
nice: setpriority: Permission denied
```

This fails because the user is not the superuser.

What if you want to change the priority of a process that is already running?

Then we can use the `renice` command:

NAME

`renice` -- alter priority of running processes

SYNOPSIS

`renice priority [[-p] pid ...] [[-g] pgrp ...] [[-u] user ...]`

`renice -n increment [[-p] pid ...] [[-g] pgrp ...] [[-u] user
...]`

DESCRIPTION

The `renice` utility alters the scheduling priority of one or more running processes. The following `who` parameters are interpreted as process ID's, process group ID's, user ID's or user names. The `renice'ing` of a process group causes all processes in the process group to have their scheduling priority altered. The `renice'ing` of a user causes all processes owned by the user to have their scheduling priority altered. By default, the processes to be affected are specified by their process ID's.

For example if we want to **lower** the priority of a process with PID 1234:

```
$ renice 10 -p 1234
```

This will increase the priority number by 10 effectively **lowering** the priority.

If we are the superuser we can **increase** the priority using the command:

```
$ renice -10 -p 1234
```

Most shell command interpreters allow you to manage processes by starting and stopping them and moving them into the background.

If you start a process and forget to use & to put it into the background, you can stop the process with `ctrl-Z` and put it into the background the `bg` command. Within the shell processes are managed as jobs. Here is an example:

```
$ sleep 60
^Z
[7]+  Stopped                 sleep 60
$ bg 7
[7]+ sleep 60 &
$
```

forgot the &

type control-Z

job number

sleep command now
running in background

There is also a `jobs` command to list all the jobs currently under management by the shell.

Also, there is an `fg` command to bring a job from the background back to the foreground.

```
$ sleep 600
^Z
[1]+  Stopped                  sleep 600
$ bg 1
[1]+ sleep 600 &
$ jobs
[1]+  Running                  sleep 600 &
$ fg 1
sleep 600
^C
```

Summary:

- a scheduler is a part of the kernel that decides what process to run
- processes have a priority number
- we can use the nice and renice commands to manage priority
- the shell can move processes to and from the background



4.3 Boot Sequence

INFO1112

Bob Kummerfeld

Starting the system: Bootstrapping

Starting your computer begins when you turn on the power.

From that point until you have a running system is the subject of this segment.

In the beginning....there was hardware.



The computer has several sorts of main memory: RAM (Random Access Memory), ROM (Read Only Memory) and flash memory. The RAM is volatile - the contents are lost when the power is turned off. The ROM is persistent - the contents are maintained even with the power off but ROM cannot be changed after initial manufacture. Flash memory is also persistent but can be changed by a program. The ROM contains the initial instructions that are executed when the power is first turned on for the first time.

When you first turn on the computer the *hardware* does some test operations. This is before any program is executed.

In the second stage....



When you turn on a microcomputer such as Arduino based systems, the startup is very simple. The first instruction to be executed is at a particular memory location in the address space. This initial address is in persistent memory (flash or ROM) and is usually address zero.

At this initial address there is a **jump instruction** that transfers control to program code for the next stage of the boot sequence. The program code is also stored in persistent memory. This stage does more hardware checks, works out what devices are connected (disk drives etc).

In the third stage...



The program stored in the persistent memory (also called the **firmware**) then works out where to find the next stage of the boot program, loads it into memory and jumps to it. Typically this is done by looking at a fixed list of places that is stored in the flash memory. You can look at this list in older PCs using the "BIOS" (Basic I/O System) program. The locations are devices like hard drive, USB mem, CD ROM drive etc.

On disk, the program it loads is stored in the Master Boot Record (MBR), normally the first block on the device.

Master Boot Record

The MBR is very restricted in size. It is stored on the first block of a disk (or other mass storage) and is less than 512 bytes long.

It contains a program that reads in the next stage of boot program (in linux one option is called GRUB - GRand Unified Boot loader) as well as a table of device configuration for the boot device.



In the fourth stage...

The program (GRUB) that is read from mass storage can be much larger and it will typically have a driver for a file system on mass storage. This is because the kernel of the operating system is stored on a file in a file system.

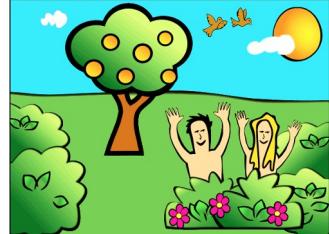
This part of the boot program will navigate the namespace of the file system on mass storage and find the file containing the kernel (often stored in the `/boot` directory). It will then load it into memory and start it running. There may be several versions of the kernel in the file system and GRUB works out which one to load.



In the fifth stage...

The kernel is now running. It performs final, high-level hardware checks and then loads device drivers and kernel modules as required. This is decided by configuration files stored in the initial file system.

For example, file systems that are to be mounted are specified in the `/etc/fstab` file (File System Table). The kernel will then mount all the file systems.



In the sixth stage...

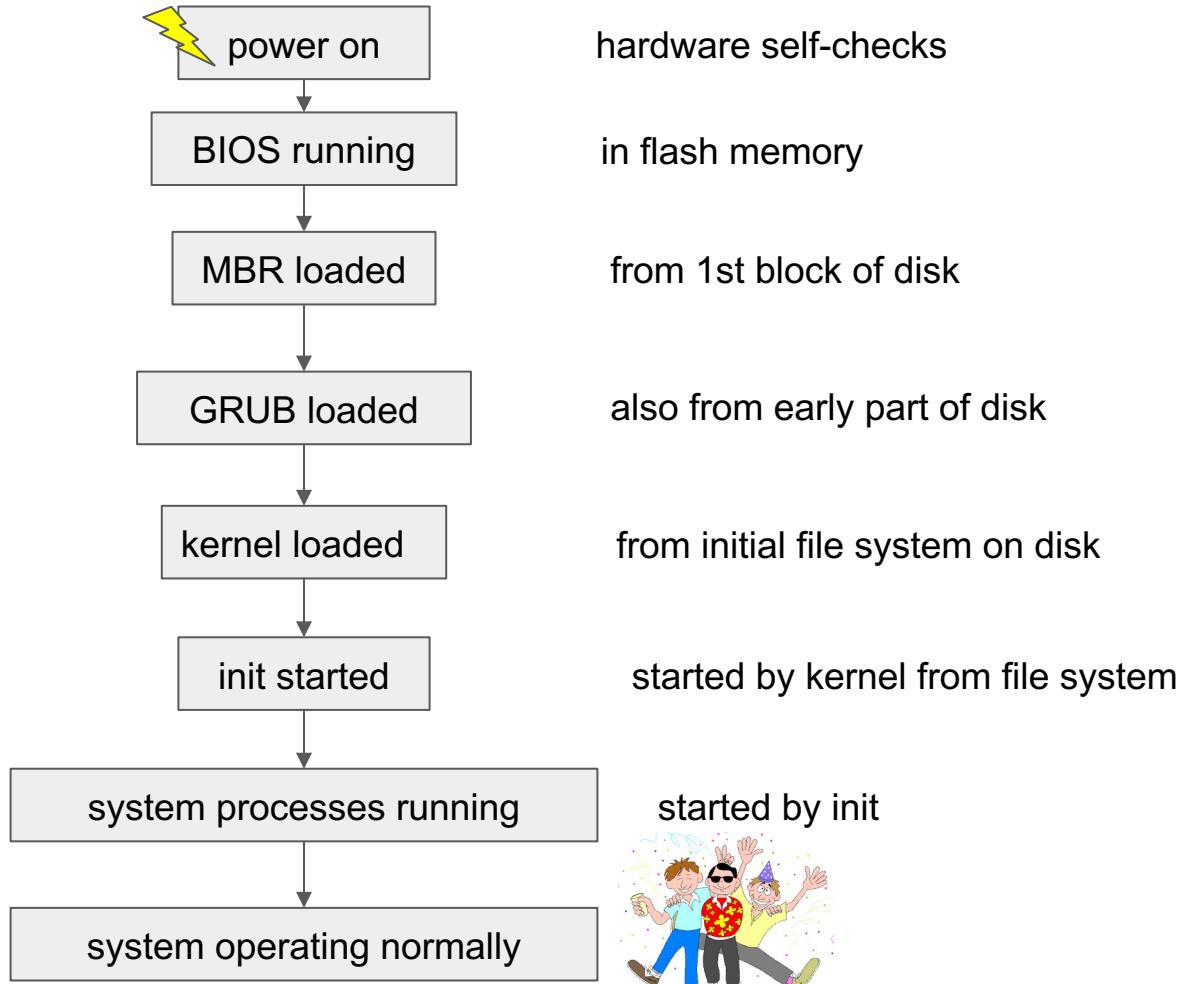
The kernel is now ready to start real processes. The first process in a Unix system is called "`init`" (in Linux there are similar programs, eg `systemd`). This initial process is started by the kernel and this is the only process that the kernel starts. From then on processes are only started by other processes and "`init`" is the first. All processes can trace their ancestry back to `init`. It starts system processes (daemons) and initializes all active subsystems.

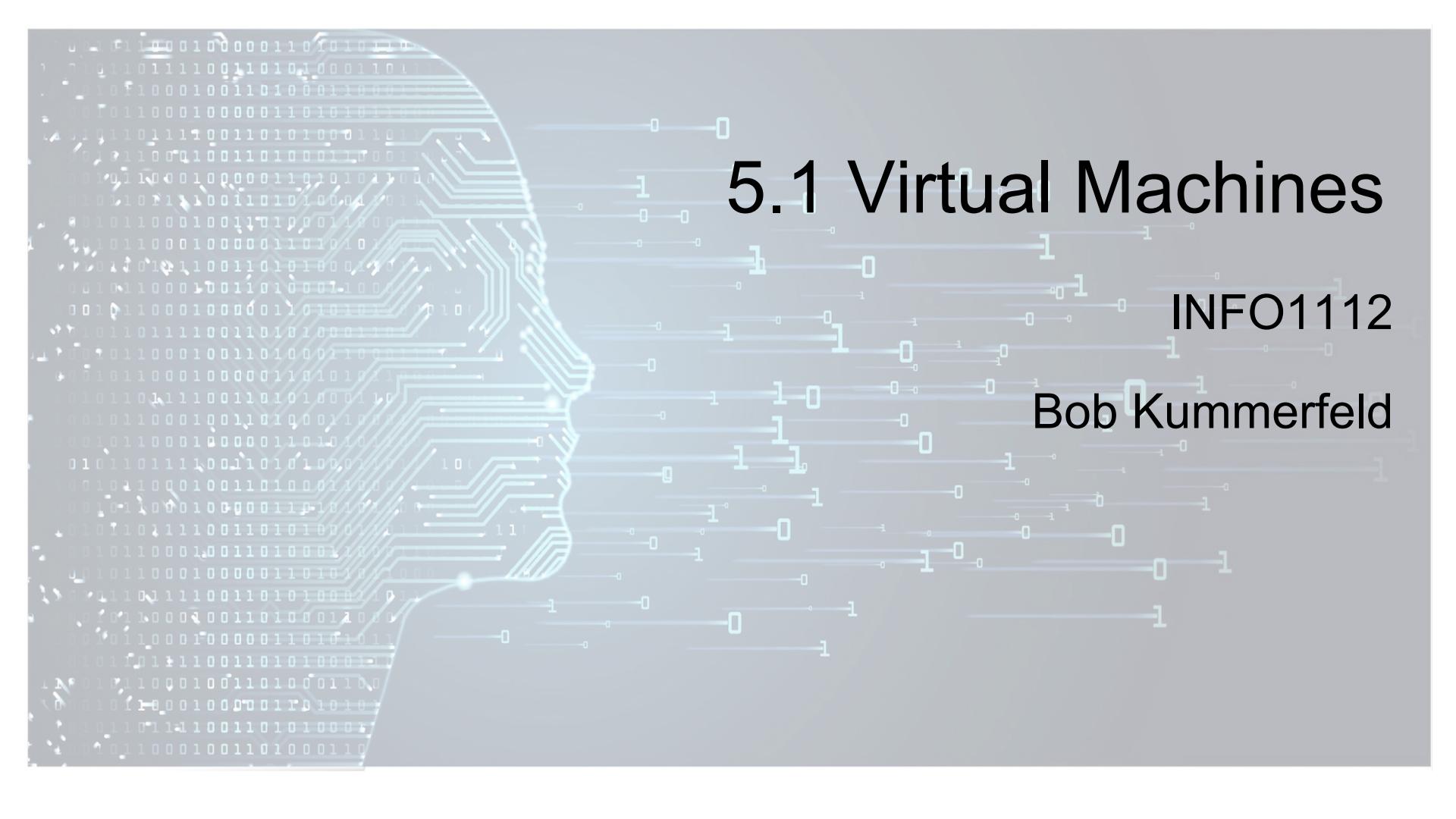
In a server only system (no graphical display) it will start a "login" program. For a system with a Graphical User Interface (GUI), `init` will start a window manager, which in turn runs the login process.

On the seventh stage the system is operating...

At this point everything is loaded, logins are accepted, background tasks are running.

```
[ 0.000000] Booting Linux on physical CPU 0x0
[ 0.000000] Linux version 4.19.75-v7l+ (dom@buildbot) (gcc version 4.9.3 (crosstool-NG crosstool-ng-1.22.0-88-g8460611))
#1270 SMP Tue Sep 24 18:51:41 BST 2019
[ 0.000000] CPU: ARMv7 Processor [410fd083] revision 3 (ARMv7), cr=30c5383d
[ 0.000000] CPU: div instructions available: patching division code
[ 0.000000] CPU: PIPT / VIPT nonaliasing data cache, PIPT instruction cache
[ 0.000000] OF: fdt: Machine model: Raspberry Pi 4 Model B Rev 1.1
[ 0.000000] Memory policy: Data cache writealloc
[ 0.000000] cma: Reserved 256 MiB at 0x000000001ec00000
[ 0.000000] On node 0 totalpages: 504832
[ 0.000000]   DMA zone: 1728 pages used for memmap
[ 0.000000]   DMA zone: 0 pages reserved
[ 0.000000]   DMA zone: 196608 pages, LIFO batch:63
[ 0.000000]   HighMem zone: 308224 pages, LIFO batch:63
[ 0.000000] random: get_random_bytes called from start_kernel+0xc0/0x4e8 with crng_init=0
[ 0.000000] percpu: Embedded 17 pages/cpu s39488 r8192 d21952 u69632
[ 0.000000] pcpu-alloc: s39488 r8192 d21952 u69632 alloc=17*4096
[ 0.000000] pcpu-alloc: [0] 0 [0] 1 [0] 2 [0] 3
[ 0.000000] Built 1 zonelists, mobility grouping on. Total pages: 503104
[ 0.000000] Kernel command line: coherent_pool=1M 8250.nr_uarts=0 cma=64M cma=256M video=HDMI-A-1:640x480M@60,margin_left=0,margin_right=0,margin_top=0,margin_bottom=0 smsc95xx.macaddr=DC:A6:32:3C:B3:83 vc_mem.mem_base=0x3ec00000 vc_mem.mem_size=0x40000000 dwc_otg.lpm_enable=0 console=ttyS0,115200 console=tty1 root=/dev/mmcblk0p7 rootfstype=ext4 elevator=deadline fsck.repair=yes rootwait quiet splash plymouth.ignore-serial-consoles
[ 0.000000] Dentry cache hash table entries: 131072 (order: 7, 524288 bytes)
[ 0.000000] Inode-cache hash table entries: 65536 (order: 6, 262144 bytes)
[ 0.000000] Memory: 1721832K/2019328K available (8192K kernel code, 661K rwdta, 2352K rodata, 2048K init, 850K bss, 353
52K reserved, 262144K cma-reserved, 1232896K highmem)
[ 0.000000] Virtual kernel memory layout:
              vector : 0xfffff0000 - 0xfffff1000  ( 4 kB)
```





5.1 Virtual Machines

INFO1112

Bob Kummerfeld

Revisiting low level machine instructions

In the first week we looked at the representation of data objects using binary numbers - bits.

In particular we showed that a key part of modern computers is that the bits can represent **instructions** to the Central Processing Unit.

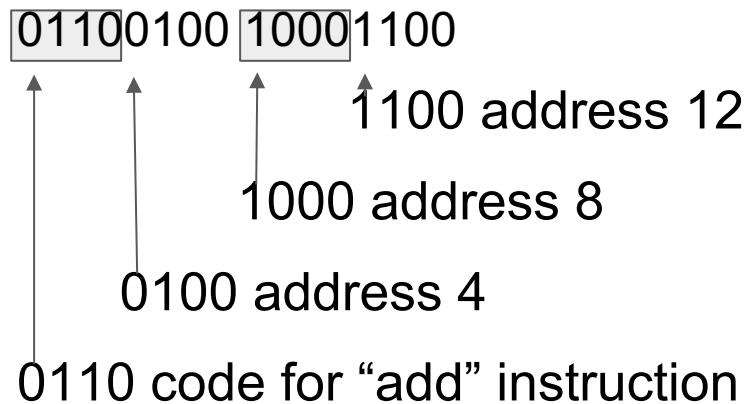
CPU

- › The CPU reads instructions from memory and interprets or **executes** them
- › Instructions are represented in bits and stored in a sequence of bytes, eg:
01100100 10001100

This might (in some hypothetical machine) mean “add the integer in address 4 to the integer in address 8 and store the result at address 12”

Instructions

In some hypothetical CPU this might be broken down as:



Assembly Language

Rather than write 01100100 10001100 when writing programs, we use a human readable shorthand called "assembly language". So our hypothetical instruction might be written as:

ADD 4, 8, 12

We use a program called an **Assembler** to read instructions in assembly language and translate them into the binary numbers that represent machine instructions.

"Hello, world" in 386 assembler

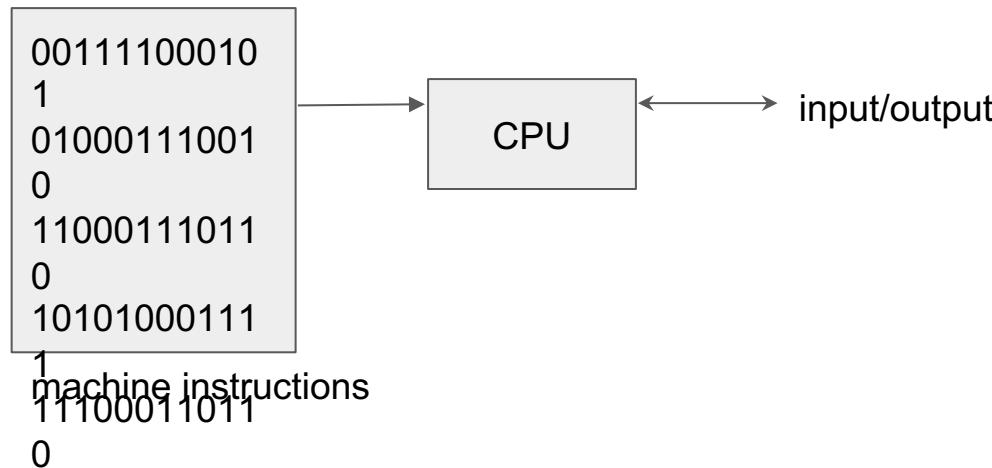
```
mov $4, %eax /* write system call */  
mov $1, %ebx /* stdout */  
mov $msg, %ecx  
mov $msgend-msg, %edx  
int $0x80  
mov $1, %eax /* _exit system call */  
mov $0, %ebx /* EXIT_SUCCESS */  
int $0x80  
msg: .ascii "Hello, world\n"  
msgend:
```

Assembler Program

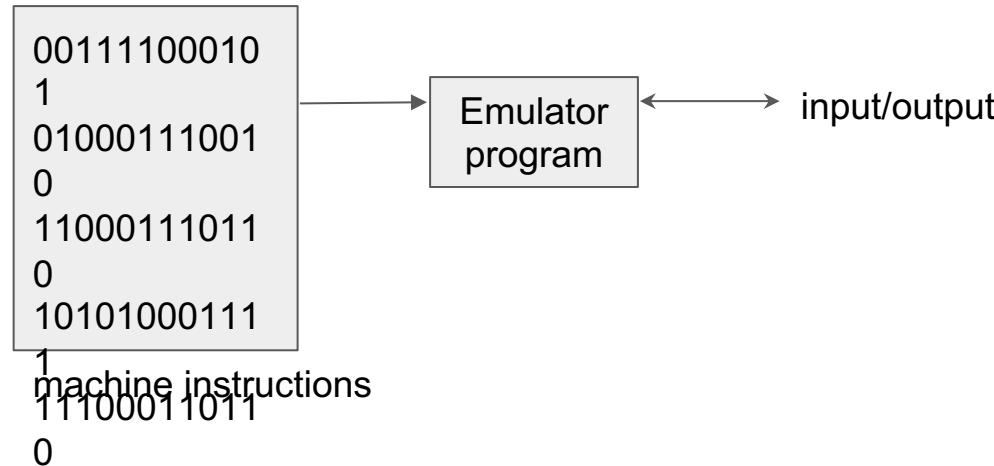
00111100010
1
01000111001
0
11000111011
0
10101000111

1
machine instructions
11100011011
0
.....

CPU interprets instructions

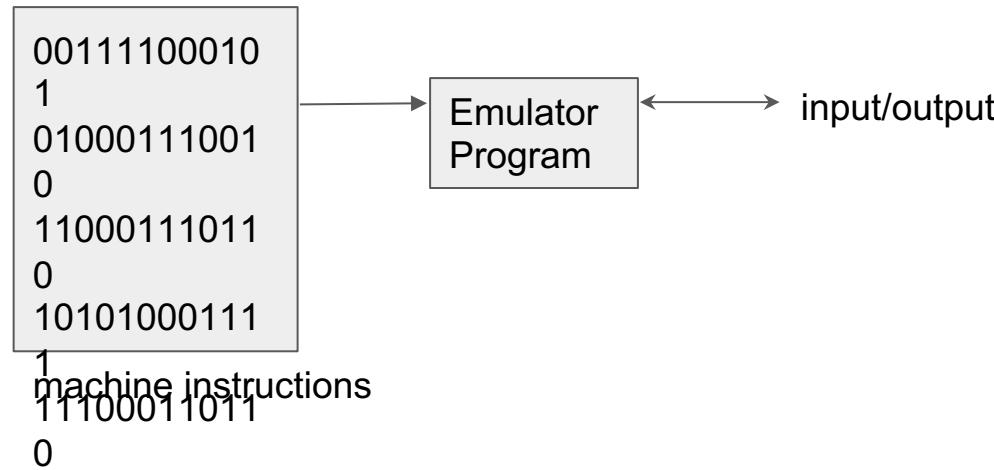


**Virtual machine emulator
program interprets instructions**



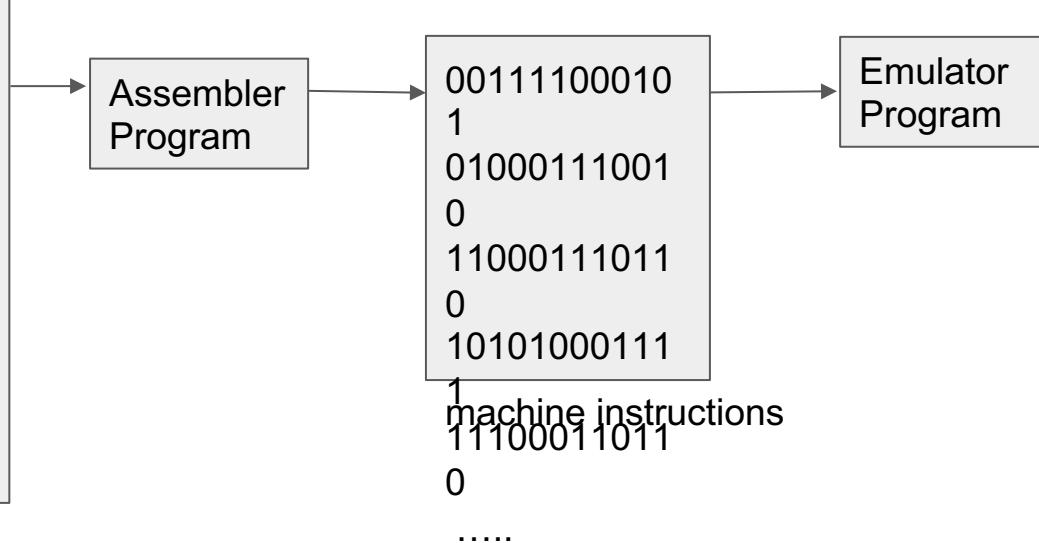
Emulator programs

As well as using a program to translate Assembly language into binary machine code, we can also write a program that reads the binary instructions, works out what they mean and then carries out the instruction. This type of program is called an ***Emulator*** or ***Virtual Machine***, because it emulates the instructions, pretending to be a real CPU.

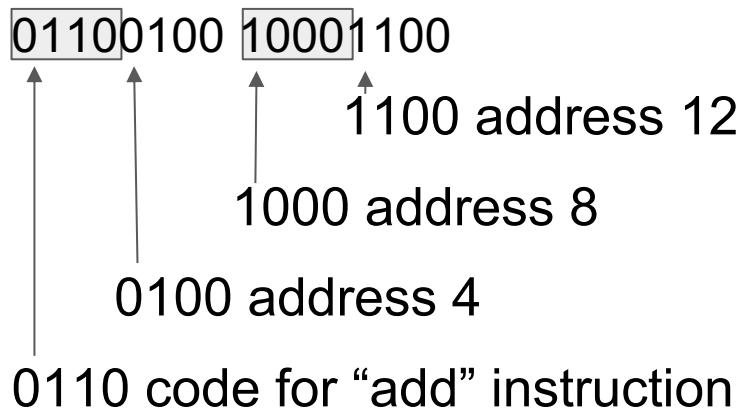


"Hello, world" in 386 assembler

```
mov $4, %eax /* write system call */  
mov $1, %ebx /* stdout */  
mov $msg, %ecx  
mov $msgend-msg, %edx  
int $0x80  
mov $1, %eax /* _exit system call */  
mov $0, %ebx /* EXIT_SUCCESS */  
int $0x80  
msg: .ascii "Hello, world\n"  
msgend:
```



Example Instruction breakdown



So we can determine the operation code using the shift operation.

01100100 10001100 >> 12



number of bits to shift

shift RIGHT operation in Python

The result is:

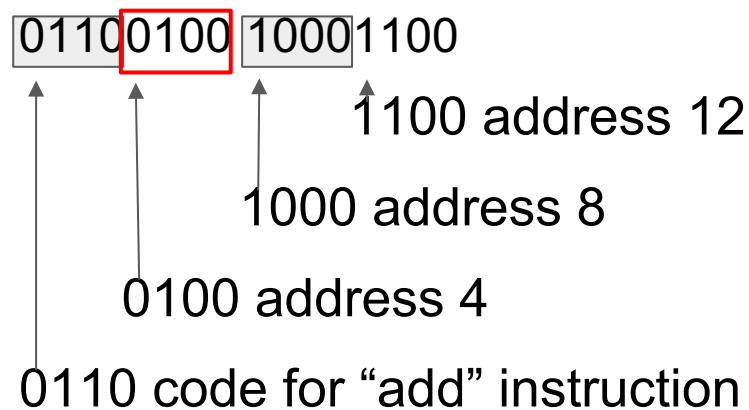
0000000000000110



instruction operation code

zero bits added from the left

What if we wanted to find the first address operand in the instruction?



If we shift right by 8 we will have that address field at the right of the number, but the operation code will still be in the number.

We can extract the code using a shift operation + the AND logical operation.

Start with the shift:

01100100 10001100 >> 8

↑
↑
number of bits to shift

shift RIGHT operation in Python

The result is:

00000000001100100

still have the operation code part
↑
address operand

zero bits added from the left

We can extract the code using a shift operation + the AND logical operation.

0000000001100100 & 0b0000000000001111

↑
↑
bit mask

logical AND operation in Python

The result is:

0000000000000100

↑
first address field

zero bits added from the left

How does this work?

Logical and is a logic operation. On a single bit it works like this:

$$0 \& 0 = 0$$

$$0 \& 1 = 0$$

$$1 \& 0 = 0$$

$$1 \& 1 = 1$$

So if we have two binary numbers, the value at any bit position in the first number will be the same if the second number has a 1 in that same position.

eg:

$$\begin{array}{r} 10101010 \\ \& \underline{00001111} \\ 00001010 \end{array}$$

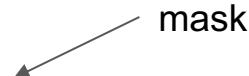
This is also called "masking"

The second number can be thought of as a "mask" that allows bits through wherever the mask has a one.

eg:

$$\begin{array}{r} 10101010 \\ \& \underline{00001111} \\ \hline 00001010 \end{array}$$

mask



Looking again at the AND operation:

0000000001100100 & 0b00000000000001111

↑
↑
bit mask

logical AND operation in Python

The result is:

0000000000000100

↑
first address field

zero bits added from the left

We can write an emulator program in any language to emulate ANY existing instruction set and so run programs written for any other computer on our computer. BUT our emulated programs may run very slow.

It is possible to speed up the emulator code by writing in a low level language and optimising how we do things.

<https://www.cambus.net/emulators-written-in-javascript/>

To run very old software (eg games) we can use an emulator or VM. The old machines were much slower and had less memory than modern computers. An emulator will often run faster than the original machine!

We can even write emulators in javascript in a browser:

<https://www.cambus.net/emulators-written-in-javascript/>

Emulators generally emulate all the instructions and registers of a CPU but if they also emulate all the system aspects such as memory management and I/O then we usually call them

Virtual Machines.

VMs are very complete emulations and are capable of running complete operating systems in the emulator. This means you can run another operating system (eg Linux) on your Windows machine by running it in a virtual machine.

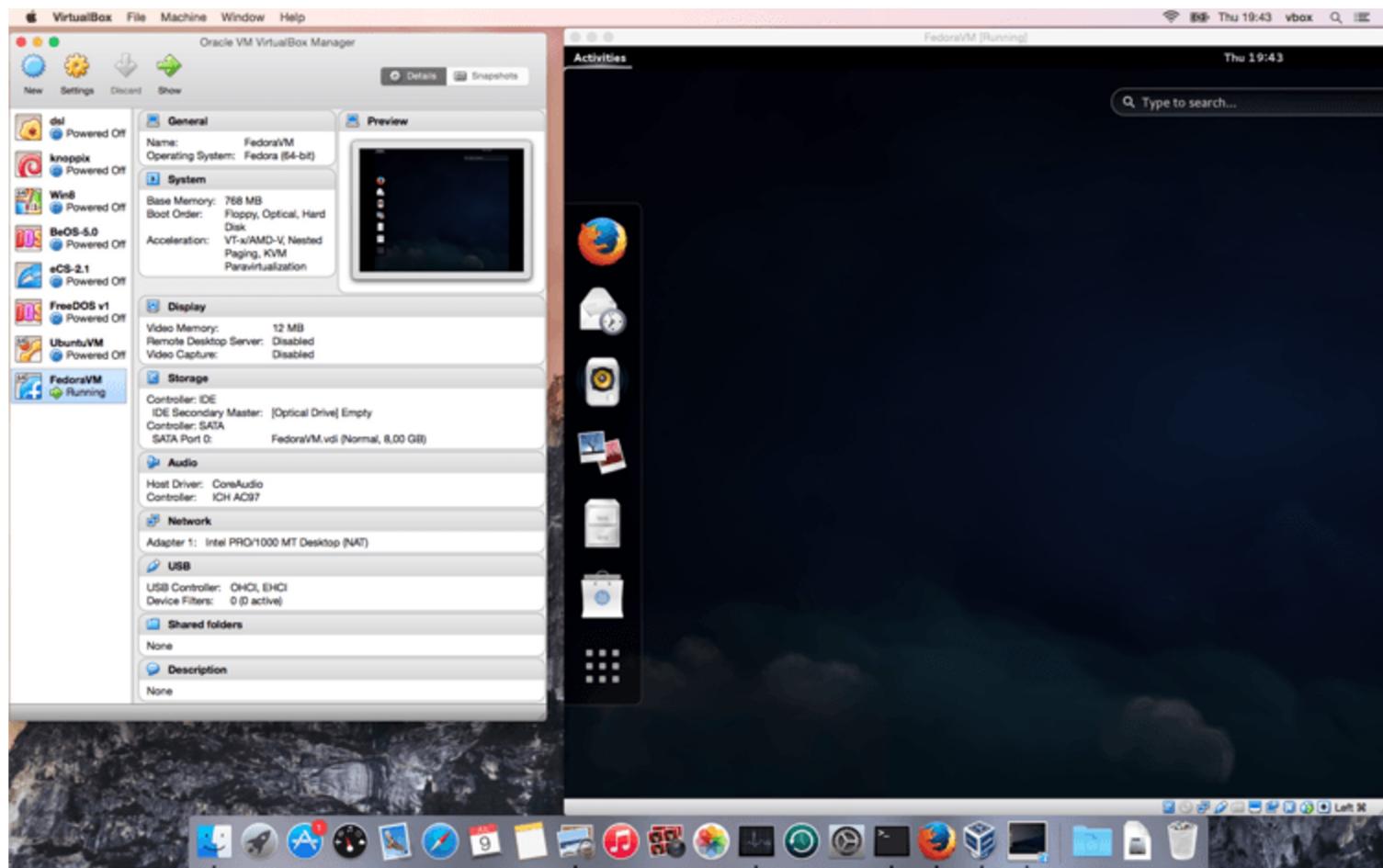
Virtual Machines

There are a number of implementations of virtual machines for particular computer architectures. By far the most popular are VMs for the Intel 8086 architecture found in Windows PCs.

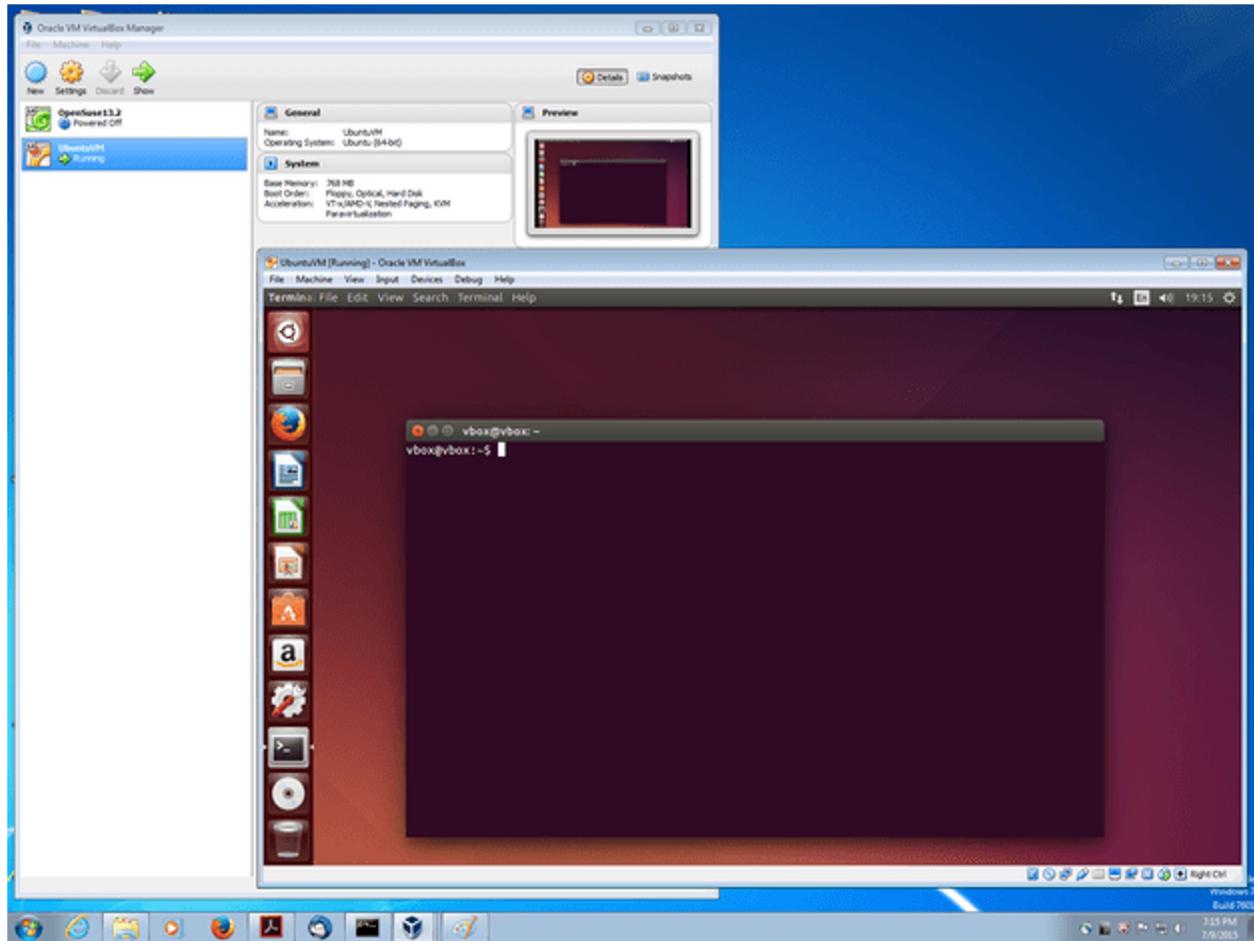
A popular and freely available (Gnu GPLv2) VM is VirtualBox by Oracle.

<https://www.virtualbox.org/wiki/Screenshots>

Fedora Linux
running inside
VirtualBox
running on
MacOS



Ubuntu Linux running inside VirtualBox running on Windows



VMs allow you to run a different operating system or a different version of an operating system

VMs allow you to run older software

VMs provides isolation (mostly) from bugs or malicious code

VMs allow you to test software in a different environment

VMs allow you to develop and test software on many different target environments from the same machine

VMs allow many small systems to run simultaneously on a single large machine (cloud providers use VMs extensively)

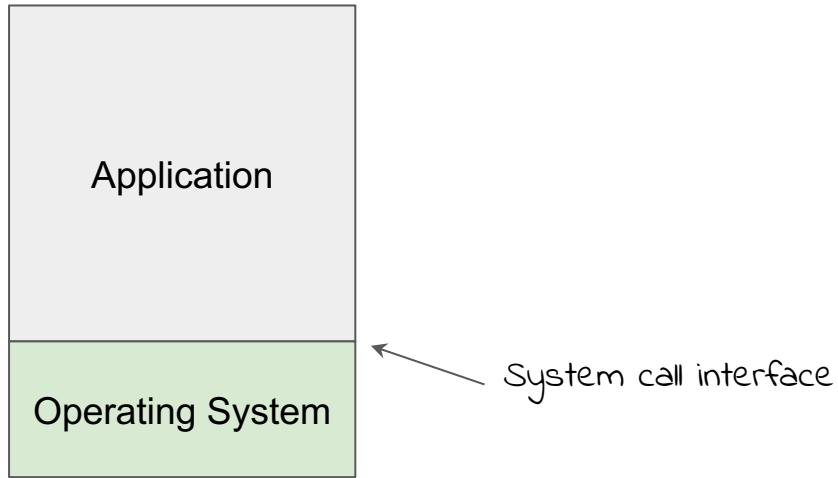


5.2 Containers

INFO1112

Bob Kummerfeld

Application Environment



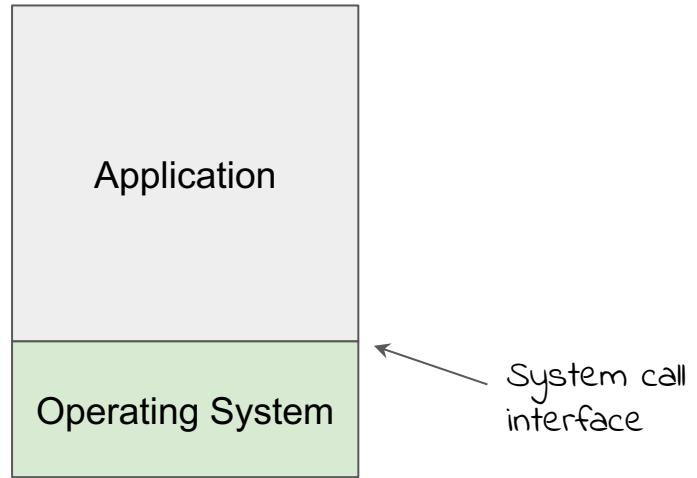
The application can **only** access the rest of the system using system calls. All file I/O (and therefore access to devices, network etc) is carried out using system calls.

Application Environment

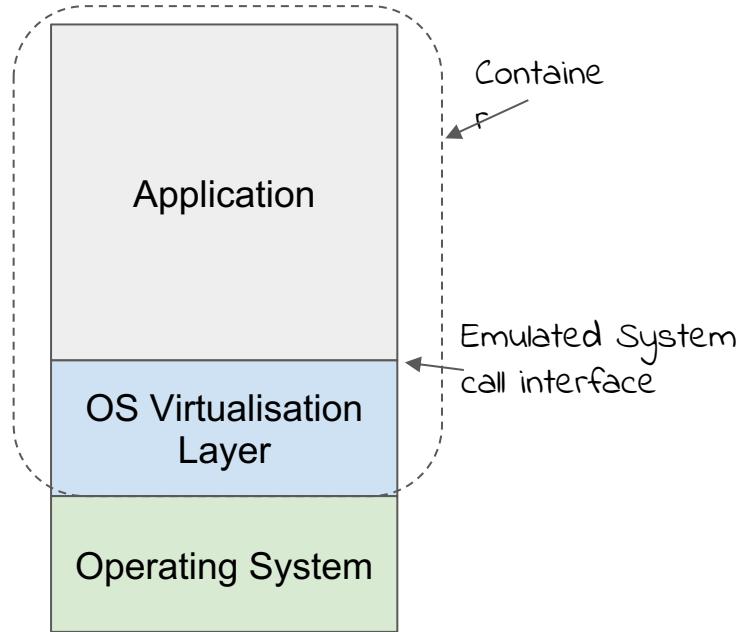
The application view of its environment is the system call API and the namespace.

Since the namespace (files, devices etc) are all accessed via the **system call interface** (open, close, read, write, fork, exec.....) then if all system calls from an application could be captured and *emulated*, the application could be tricked into thinking it was running in any required environment or system.

If we add a layer between the application and the operating system kernel that intercepts the system calls we can provide a completely new environment.



Container



Containers

If we add this extra "Operating System virtualisation" layer we can trick the application into thinking it is running in a new environment.

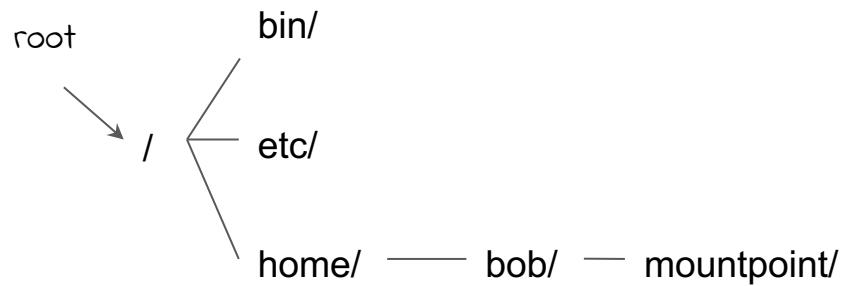
Emulating the Namespace

The application view of the namespace starts at the root (/) and includes directories for standard command line utilities (/bin, /usr/bin etc), configuration files (/etc), libraries (/lib, /usr/lib...) and other application specific locations.

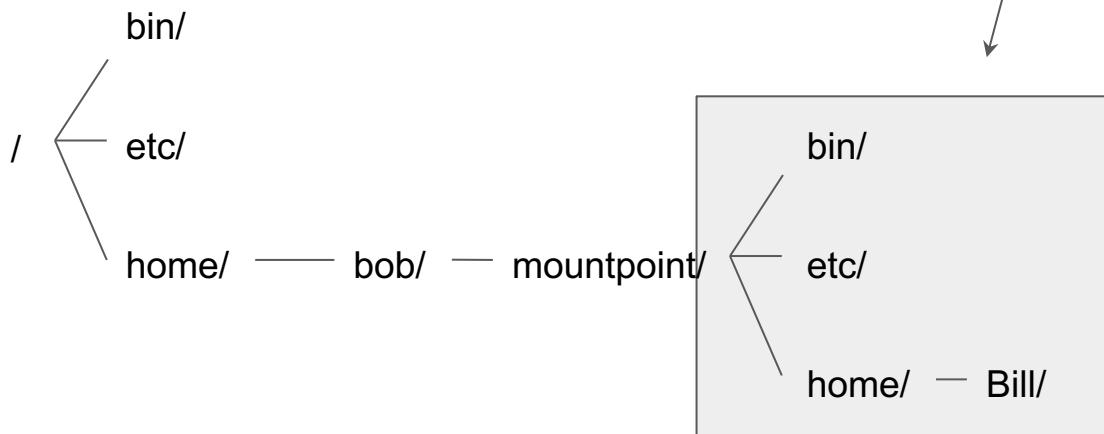
If we could trick an application into viewing the namespace root as starting at some other point in the real name space, then we can run the application in a completely different environment.

Unix has a system call "chroot" that changes the root of the namespace to some other existing point in the namespace.

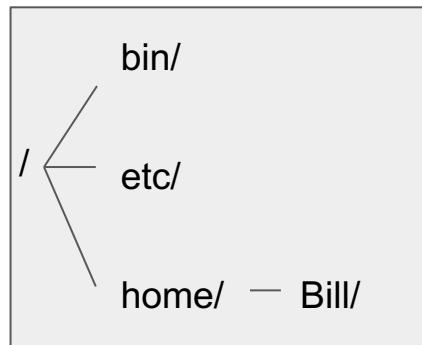
Typical namespace



Sub tree in the namespace that
emulates a typical namespace

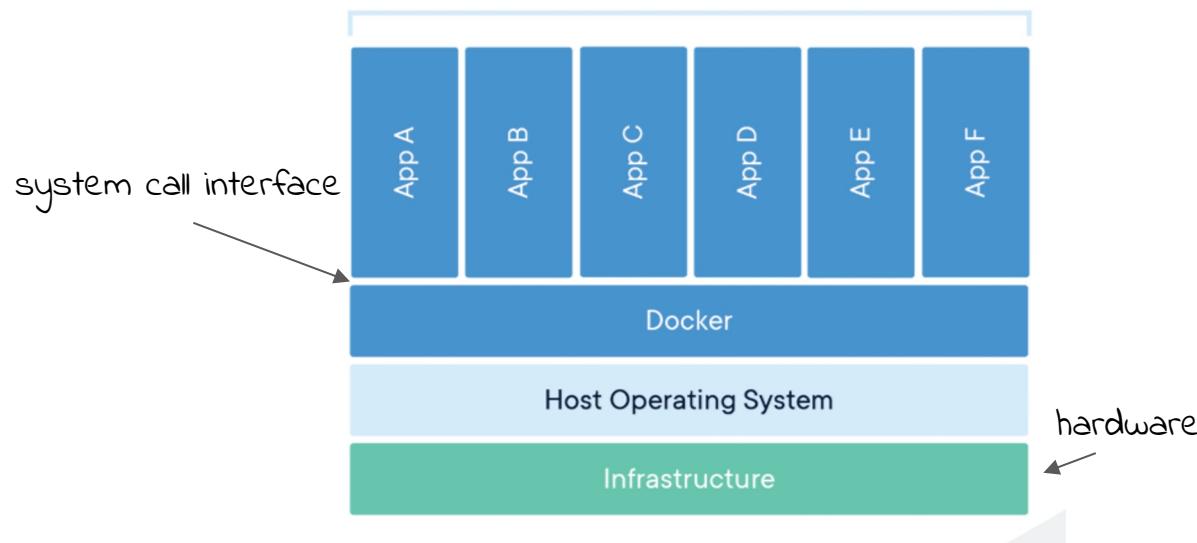


Sub tree in the namespace that
emulates a typical namespace



A program may have just this view of the namespace

Docker



Containers: docker

There is a free version of docker available (limited support, non-commercial etc).

Containers are much lighter weight than full Virtual Machines and are widely supported on cloud systems (remotely accessible computer servers sold as a service).

They have many of the advantages of VMs: isolation etc.

We can start and stop containers easily in the virtualisation layer.

We can also package up the container environment and transport it to other systems.

Containers

Containers are very useful for providing a standard environment to an application.

This feature means it is easy to move the application to a new machine by moving the container. (Assuming you have the virtualisation layer installed.)

For example, an application may make use of a particular version of a library. If this library is not installed on the target machine this make it more complex to install the application. If the application is containerised, it carries the library in the container environment.

Docker is free for individuals. Go to
docker.com, create an account then →
Explore → Docker → install the version for
your machine

Docker Example

run bash shell in the
ubuntu linux environment

```
$ docker run -it ubuntu bash
```

Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu

ubuntu container
downloaded from library

a4a2a29f9ba4: Pull complete
127c9761dcba: Pull complete
d13bf203e905: Pull complete
4039240d2e0b: Pull complete

Digest:

sha256:35c4a2c15539c6c1e4e5fa4e554dac323ad0107d8eb5c582d6ff386b383b7dce

Status: Downloaded newer image for ubuntu:latest

root@23c9be4ae564:/# ls

bin boot dev etc home lib lib32 lib64 libx32 media mnt opt
proc root run sbin srv sys tmp usr var

root@23c9be4ae564:/# exit

exit

\$

shell command

namespace inside the
container

Summary

- emulating the system call interface and the name space allows applications to be run in other environments
- we can do this with an OS Virtualisation layer
- this approach is called "Containers"
- docker is a widely used container implementation

Containers are a great way to run applications in the cloud (ie on remote machines).

Cloud services provide

- bare machines only (BYO system etc)
- machine with operating system of your choice
- virtual machine only
- virtual machine with OS
- container with OS

You add your own application(s).

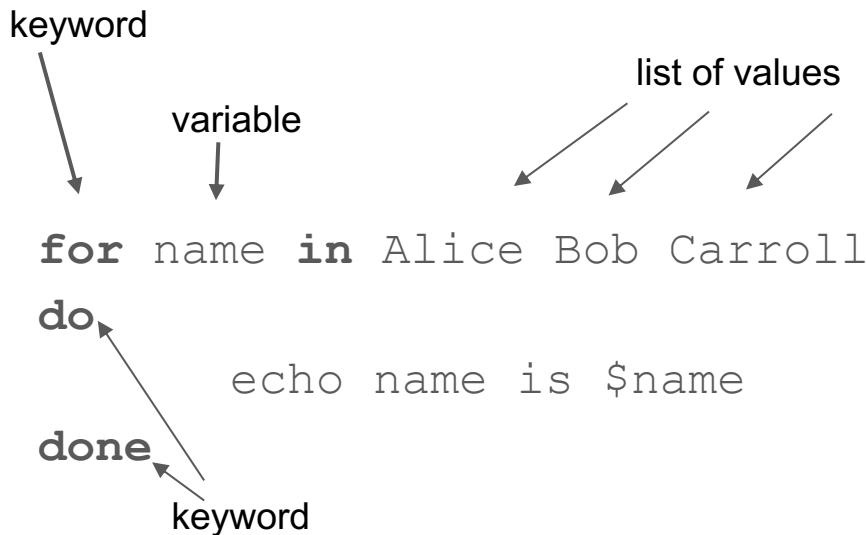
VM and Container systems have sophisticated configuration options that allow access to mass storage, file systems, devices, networks on the host machine



The Shell
for statement

Shell **for** statement

The shell has a "for" statement like conventional programming languages. The statement repeats a loop for each value in a list of values:



General form is: **for** variable **in** list ; **do** statement list **done**

Shell **for** statement

Don't forget that the list of values could be generated by a shell wildcard or the output of a program (backquotes):

```
variable           list of values: all the files in the directory
↓
for file in *
do
    echo file name is $file
done
```



#31932011

We acknowledge the tradition of
custodianship and law of the Country on which
the University of Sydney campuses stand.
We pay our respects to those who have cared
and continue to care for Country.



THE UNIVERSITY OF
SYDNEY

INFO1112

Week 6 Whole Class Session

Dr Nazanin Borhan



Video Segments of the week

- **Introduction to Network**

Circuit switching, packet switching, LAN, Ethernet, IP, TCP/IP

- **The shell while statement**

While statement in bash

- **Lab exercise for this week:**

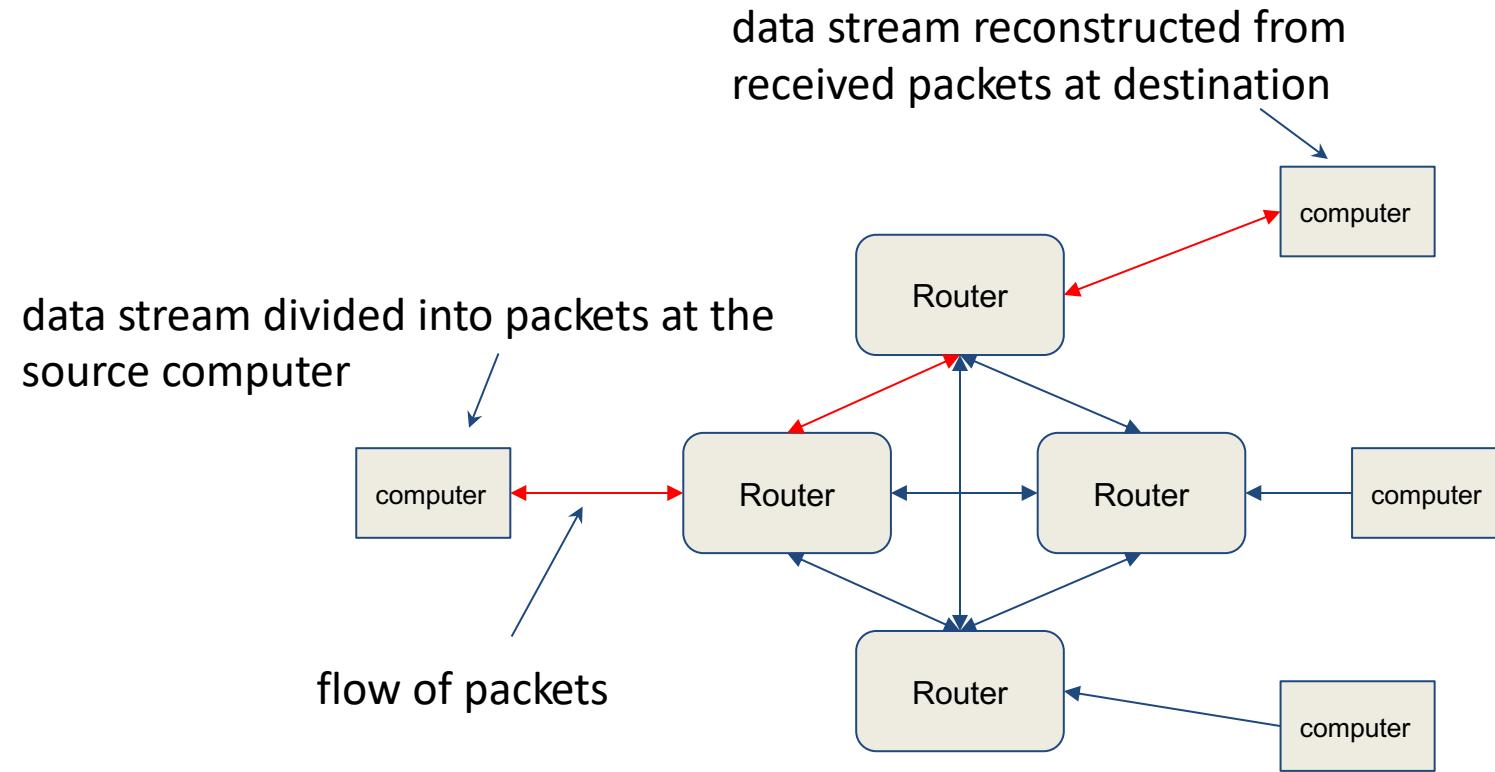
Starting your own VM to launch a webserver

Packet Switching

Instead of sending data as a continuous stream over a fixed circuit, packet switching involves breaking the stream up into small chunks or *packets*, sending them independently through the network and reassembling them in the correct order at the destination.

The packets don't necessarily all follow the same path or *route* through the network.

Packet Switching

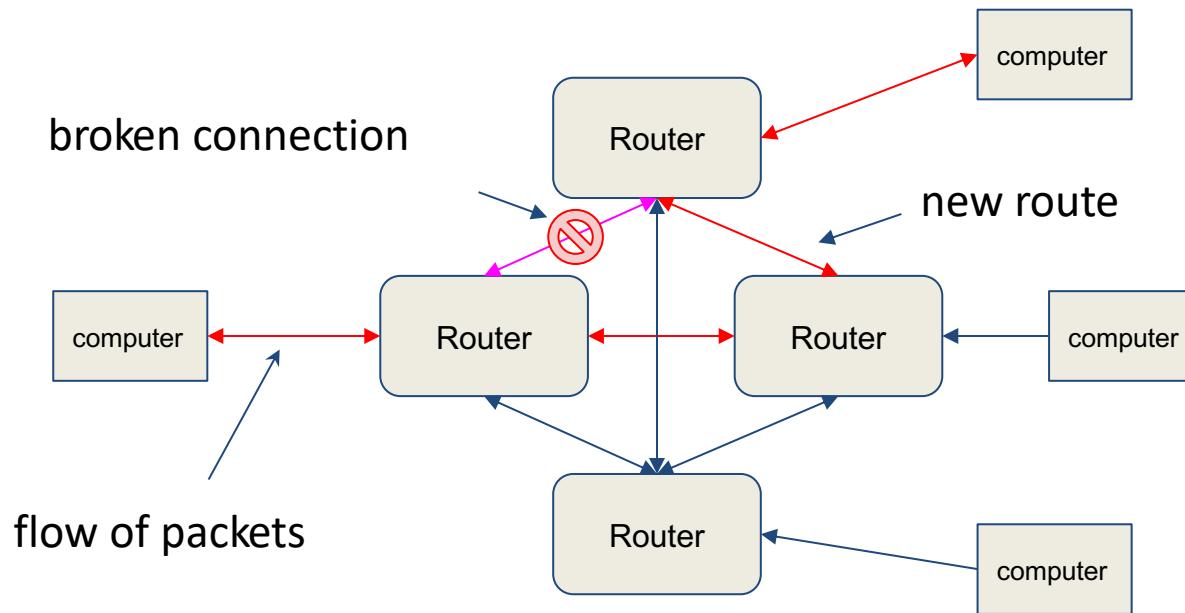


Packet Switching

Packets can follow different routes through the network, so they can be rerouted around breaks or congestion in the network.

This is a huge advantage. It gives the network resilience in the face of faults and overloading.

Routing



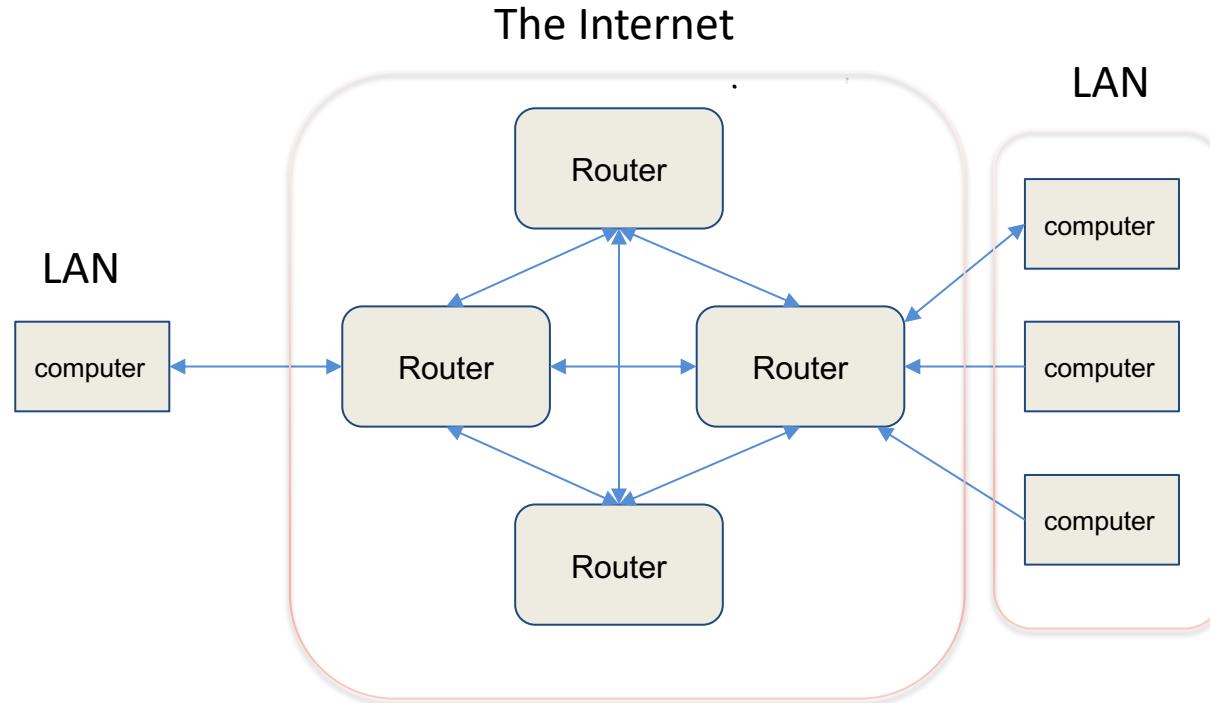
Packet Switching

Another major advantage is that the data from different connections can share a channel.

The most widely used networking standards are the Internet standards. Even phone calls are now transmitted using Internet technology and not old-style circuit switching.

Internet

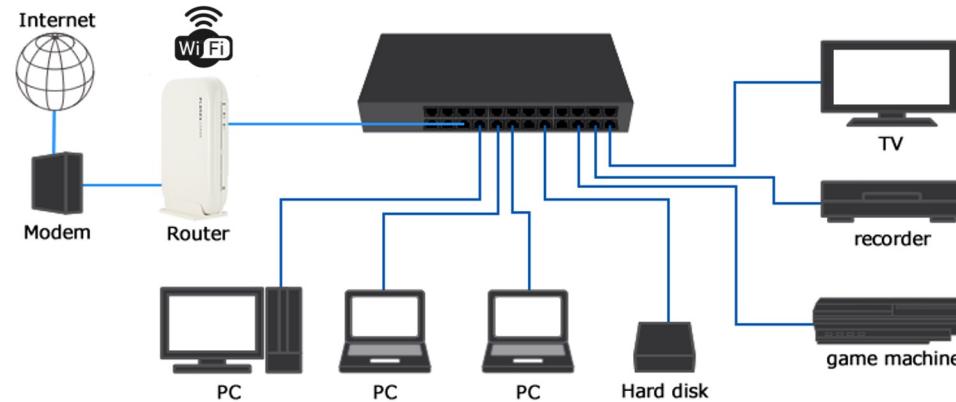
The term "internet" comes from "interconnected networks". Packets are sent from one network to another towards the destination.



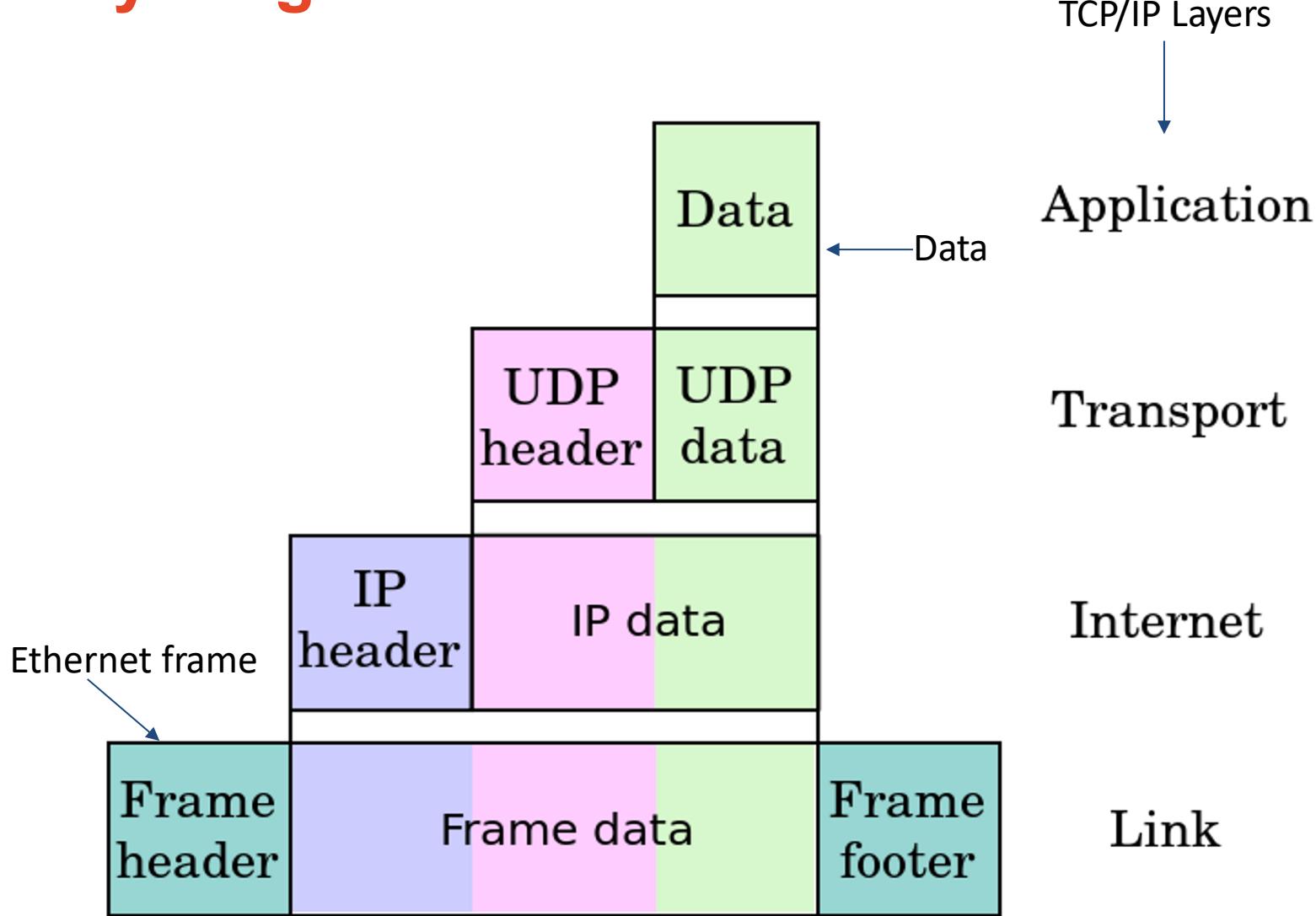
Local Area Network

A local area network (LAN) is a **collection of devices connected together in one physical location, such as a building, office, or home.** The computers in a LAN connect to each other via TCP/IP ethernet or Wi-Fi.

In a wired and wireless ethernet network the computers are connected to each other using twisted pair wiring via a router or switch. Packets are carried in form called an ethernet *frame* which is sort of packet. So we essentially have a packet (data) inside another packet (ethernet).

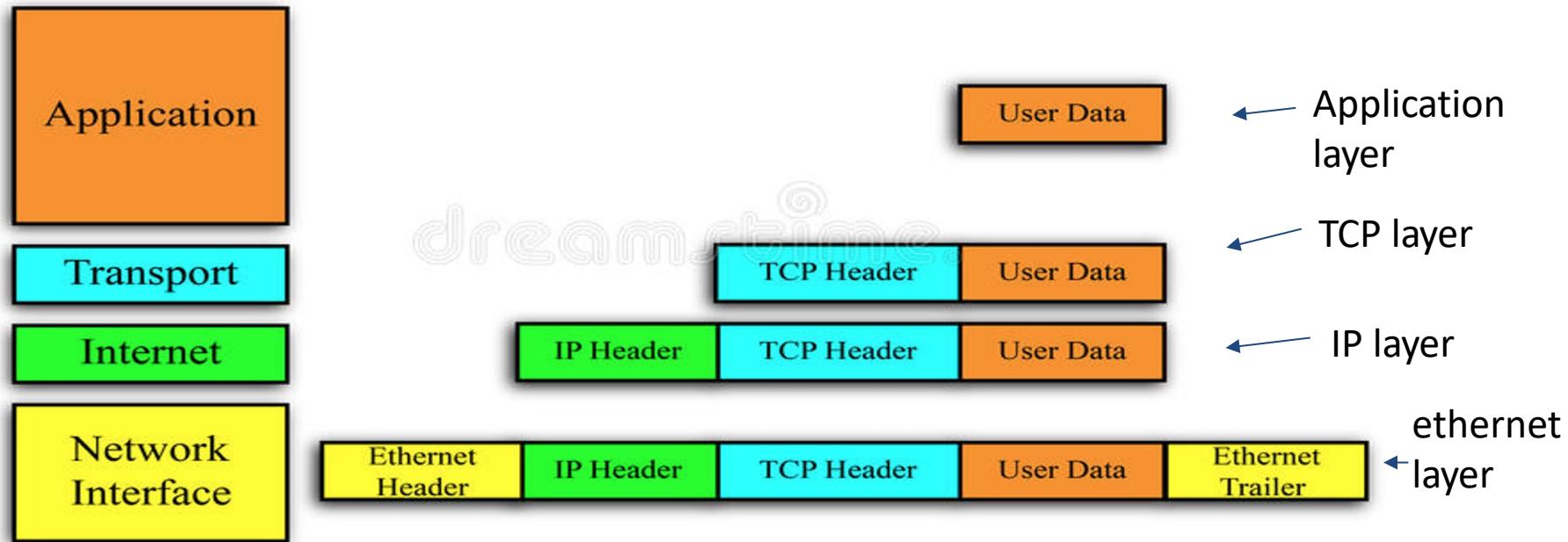


Layering



TCP/IP network layering

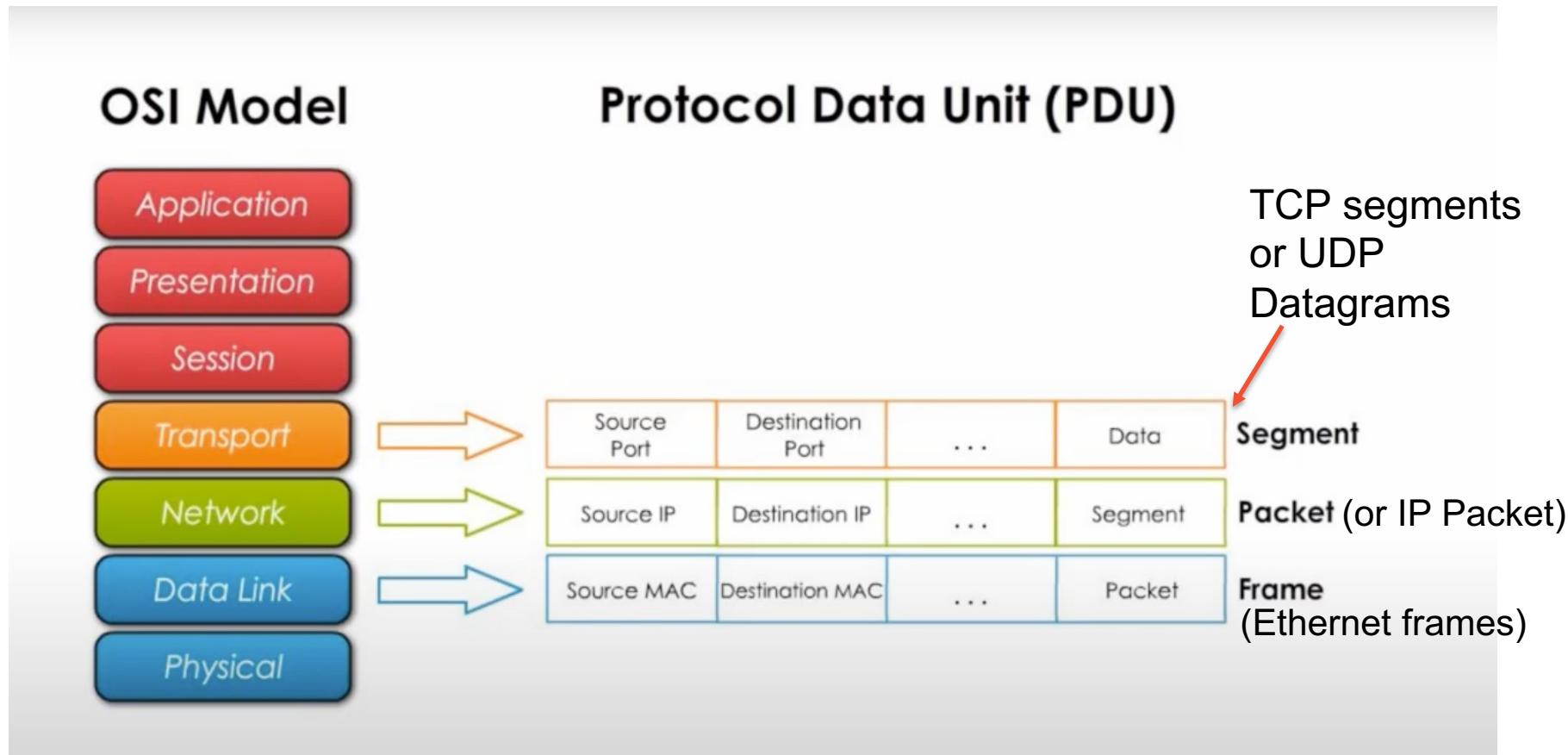
TCP/IP Network Model Encapsulation



The OSI 7 Layers

7	Application Layer	Human-computer interaction layer, where applications can access the network services
6	Presentation Layer	Ensures that data is in a usable format and is where data encryption occurs
5	Session Layer	Maintains connections and is responsible for controlling ports and sessions
4	Transport Layer	Transmits data using transmission protocols including TCP and UDP
3	Network Layer	Decides which physical path the data will take
2	Data Link Layer	Defines the format of data on the network
1	Physical Layer	Transmits raw bit stream over the physical medium

Protocol Data Unit (frames) in OSI model



OSI vs. TCP/IP Model

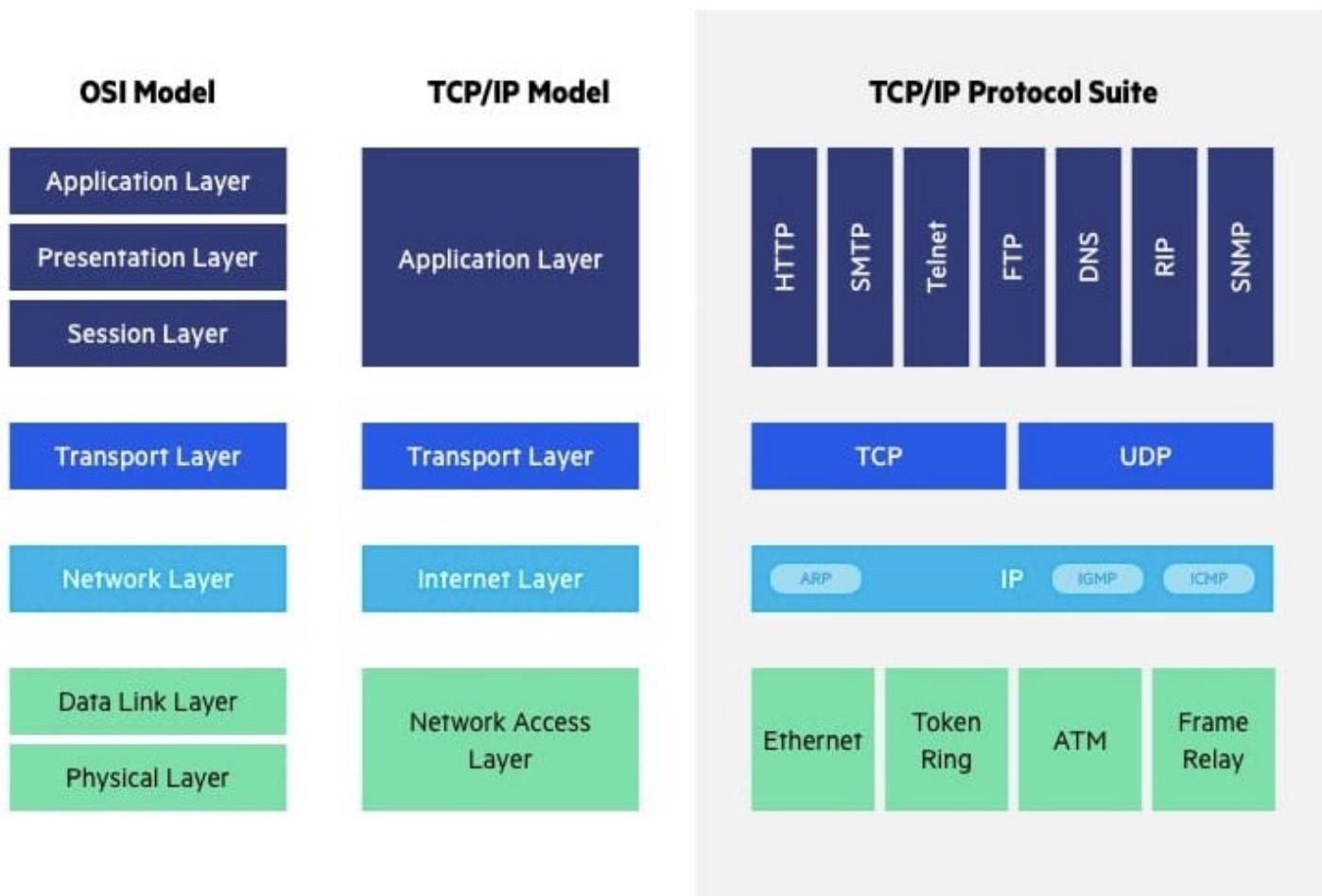


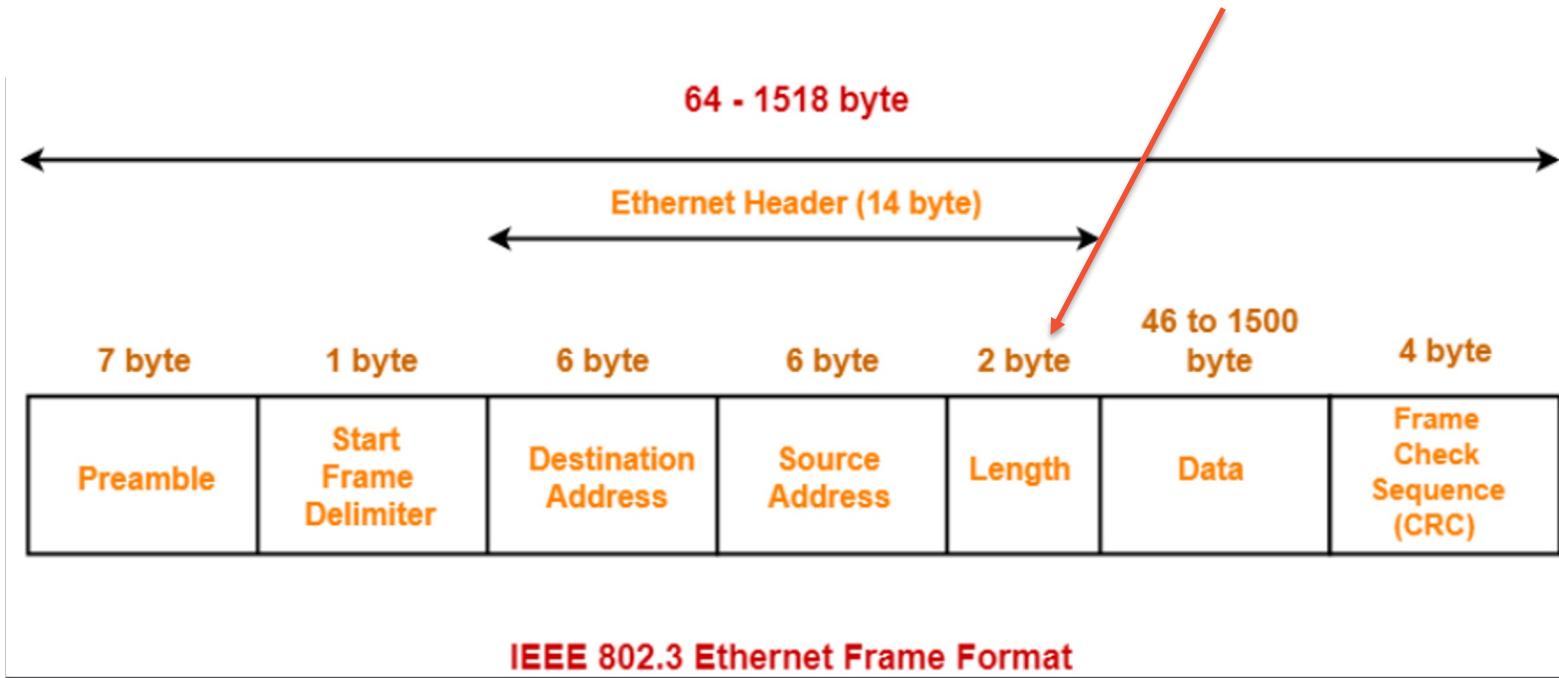
Photo source: <https://www.imperva.com/learn/application-security/osi-model/>

Break

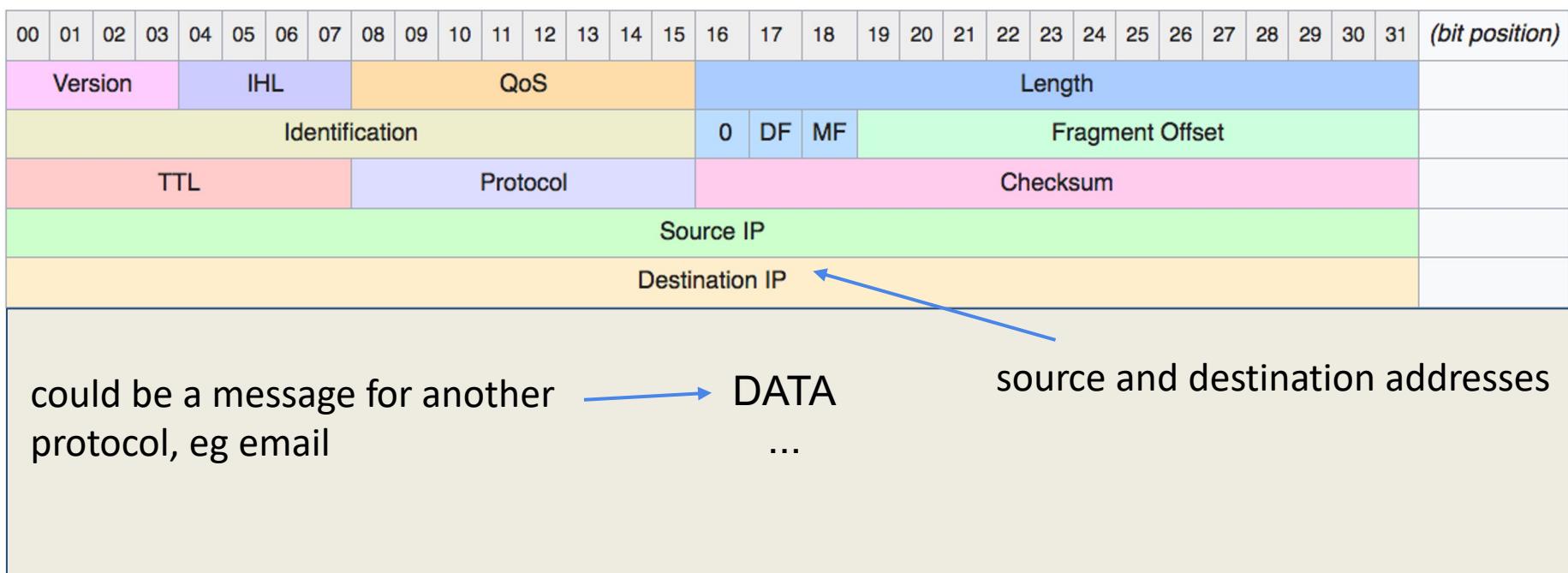


Ethernet Frame Format

In Ethernet II frame format this field is 'Type'



Internet Packet Format



Internet Packet Addressing

The source and destination address are 32 bits or 4 bytes each.

The convention is that an internet address is written as four decimal numbers each in the range 0-255 separated by dots. This is a lot easier to write than a 32 bit binary or hexadecimal number.

Example: 129.78.8.1 81.4E.8.1 (HEX)

IP V6: next generation Internet Protocol (IP) address standard

IPv4

Deployed 1981

32-bit IP address

4.3 billion addresses

Addresses must be reused and masked

Numeric dot-decimal notation

192.168.5.18

DHCP or manual configuration

IPv6

Deployed 1998

128-bit IP address

7.9×10^{28} addresses

Every device can have a unique address

Alphanumeric hexadecimal notation

50b2:6400:0000:0000:6c3a:b17d:0000:10a9

(Simplified - 50b2:6400::6c3a:b17d:0:10a9)

Supports autoconfiguration

Comparing IPv4 and IPv6 Header

IPv6 Header

Version	Traffic Class	Flow Label	
Payload Length		Next Header	Hop Limit
Source Address			
Destination Address			

IPv4 Header

Version	IHL	Type of Service	Total Length	
Identification			Flags	Fragment Offset
TTL	Protocol	Header Checksum		
Source Address				
Destination Address				
Options			Padding	

Legend



- Fields **kept** in IPv6
- Fields **kept** in IPv6, but name and position changed
- Fields **not kept** in IPv6
- Fields that are **new** in IPv6

Demo



Questions?



T1 H4 E1
E1 N1 D2

6.1 Introduction to Networks

INFO1112

Bob Kummerfeld

History

The internet of today has its origins in the 1960's with the development of "packet switched networks". Up until that time computers were connected using telephone network technology ("circuit switching").

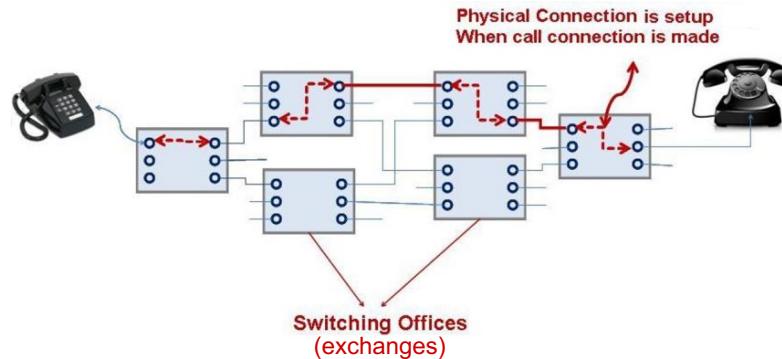
Circuit switching involves creating a complete circuit from the source to the destination when a call is requested and shutting the circuit down when the data transfer is complete.

Circuit Switching

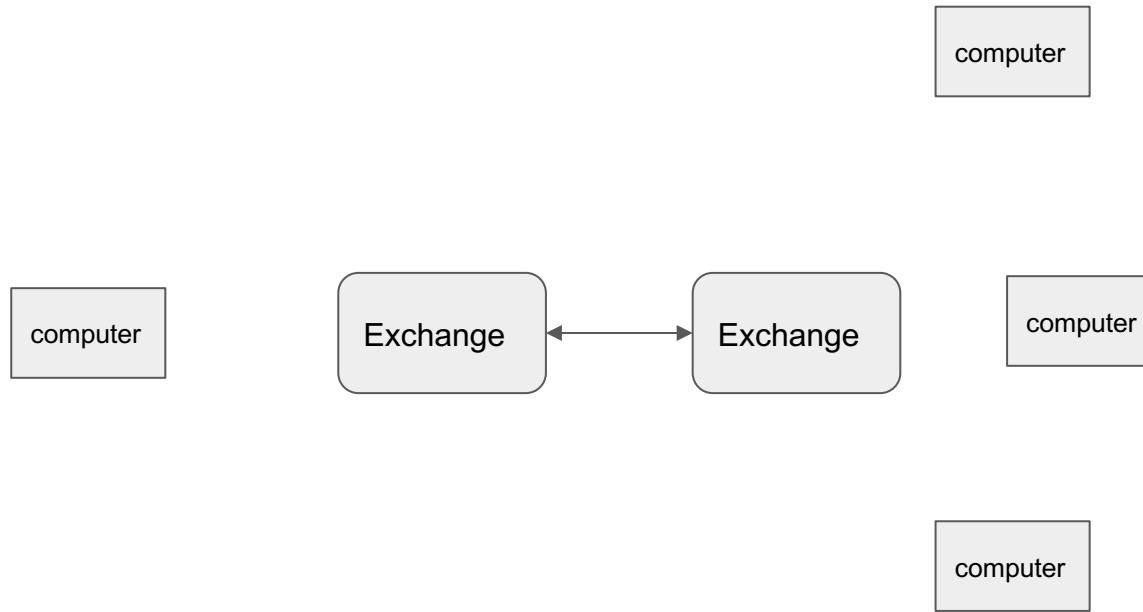
With circuit switching the data is sent in a continuous stream over the same circuit for the duration of the call.

The intermediate systems are usually called exchanges and there were relatively few of them.

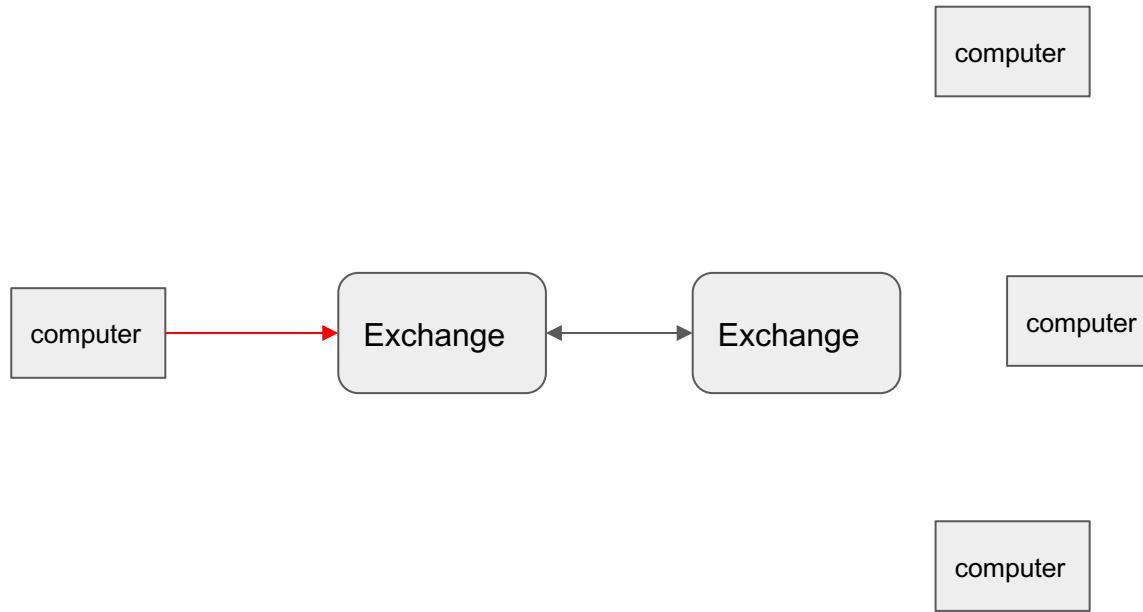
Circuit switching for process to process communication is now rare.



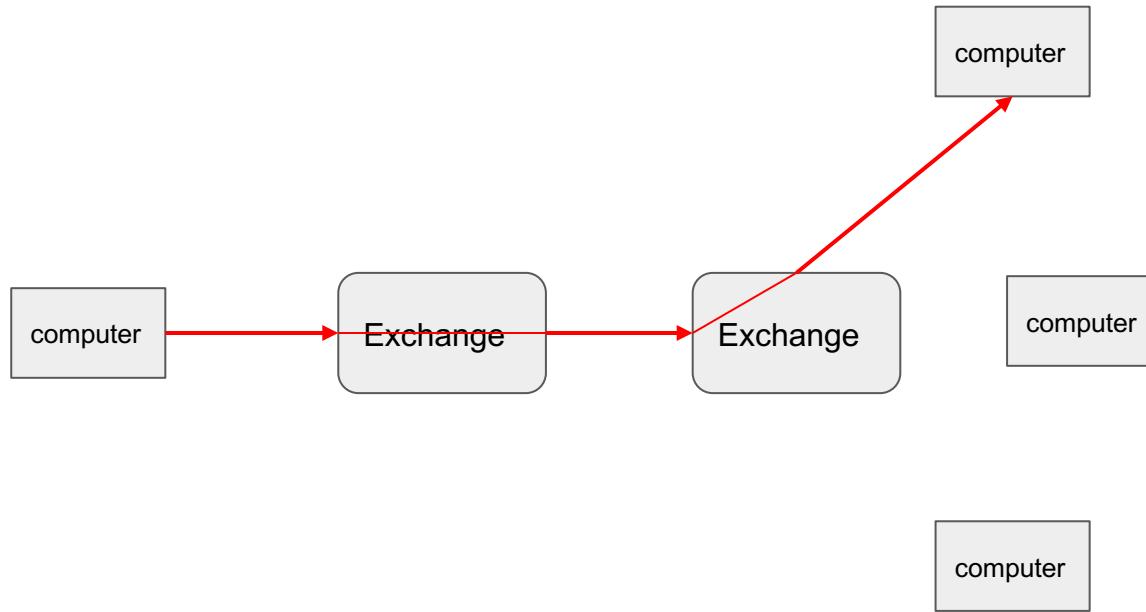
Circuit Switching



Circuit Switching



Circuit Switching



Circuit Switching

Advantage:

- each data connection has reserved bandwidth and so constant delay for data flowing over the connection

Problems:

- reserved bandwidth is wasted if no data is being transmitted
- an interruption to the circuit cannot be recovered from without making a new call

The technique of ***packet switching*** was invented to fix these problems, but at the cost of giving up guaranteed bandwidth.

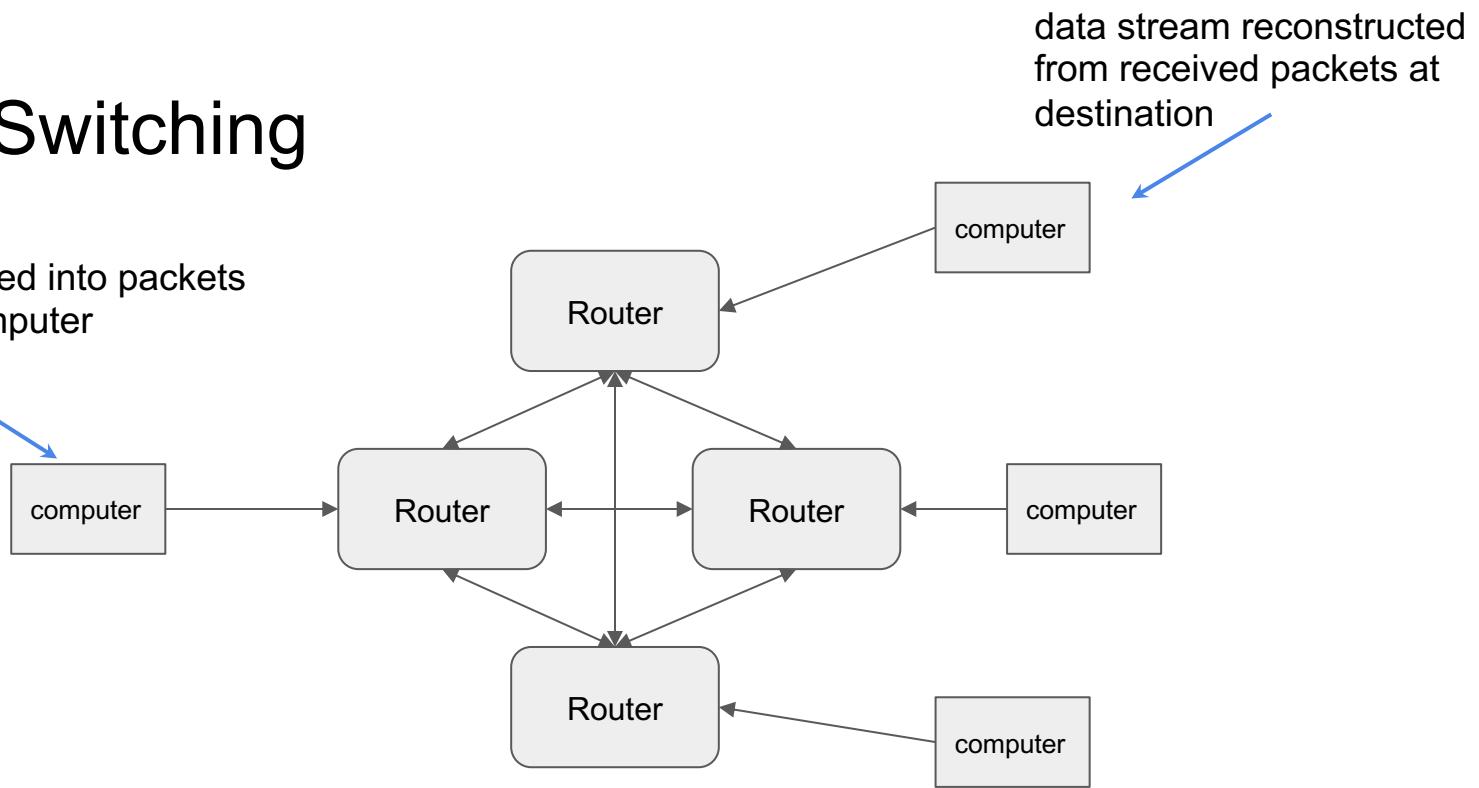
Packet Switching

Instead of sending data as a continuous stream over a fixed circuit, packet switching involves breaking the stream up into small chunks or ***packets***, sending them independently through the network and reassembling them in the correct order at the destination.

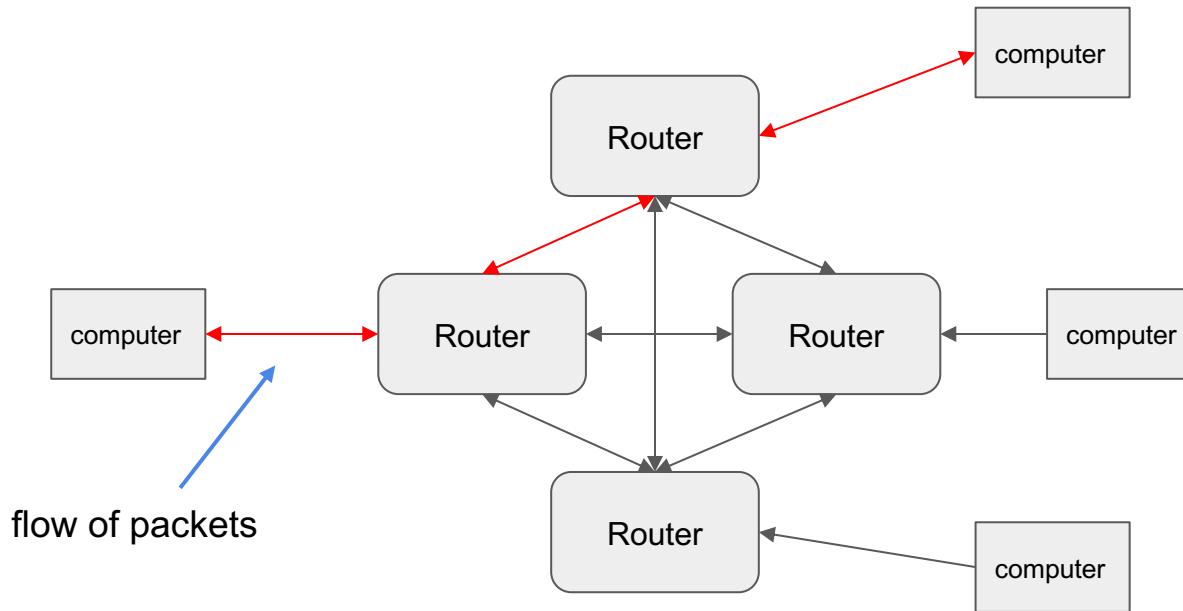
The packets don't necessarily all follow the same path or ***route*** through the network.

Packet Switching

data stream divided into packets
at the source computer



Packet Switching

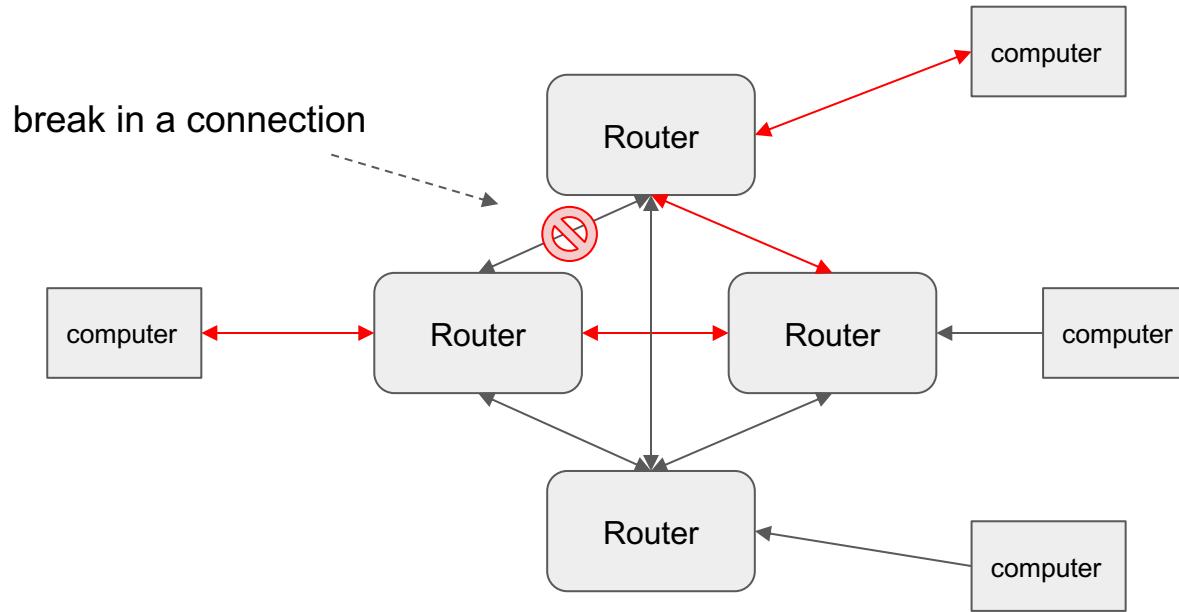


Packet Switching

Packets can follow different routes through the network, so they can be rerouted around breaks or congestion in the network.

This is a huge advantage. It gives the network resilience in the face of faults and overloading.

Packet Switching



Packet Switching

Another major advantage is that the data from different connections can share a channel.

The most common network technology today involves packet switching and by far the most widely used standards are the Internet standards. Even phone calls are now transmitted using Internet technology and not old-style circuit switching.

LANs and WANs

Connections between computers fall into two broad classes: local area networks (LAN) and wide area networks (WAN).

A LAN can be as small as your home and as large as a campus. A WAN is global and consists of interconnected LANs.

In fact the term "**inter**connected **net**works" is where we get the term *internet*.

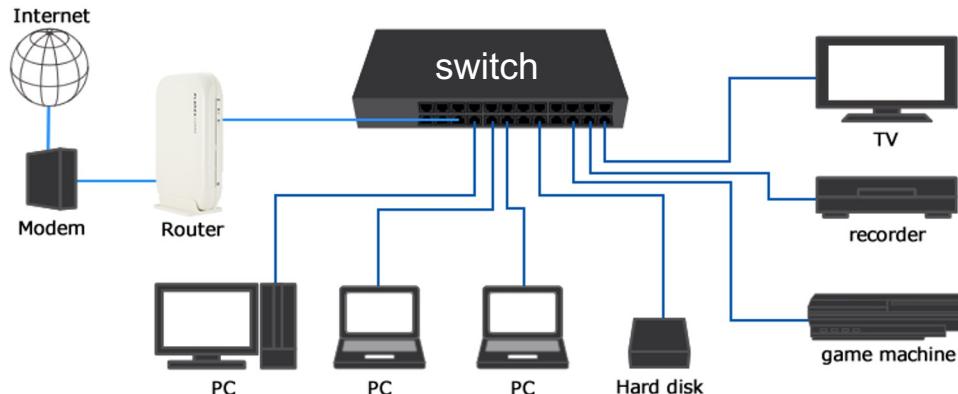
Both LANs and WANs use the packet switching technique to transfer data.

Local Area Networks

The most common form of LAN uses **ethernet** technology for the interconnection.

Ethernets come in many forms.

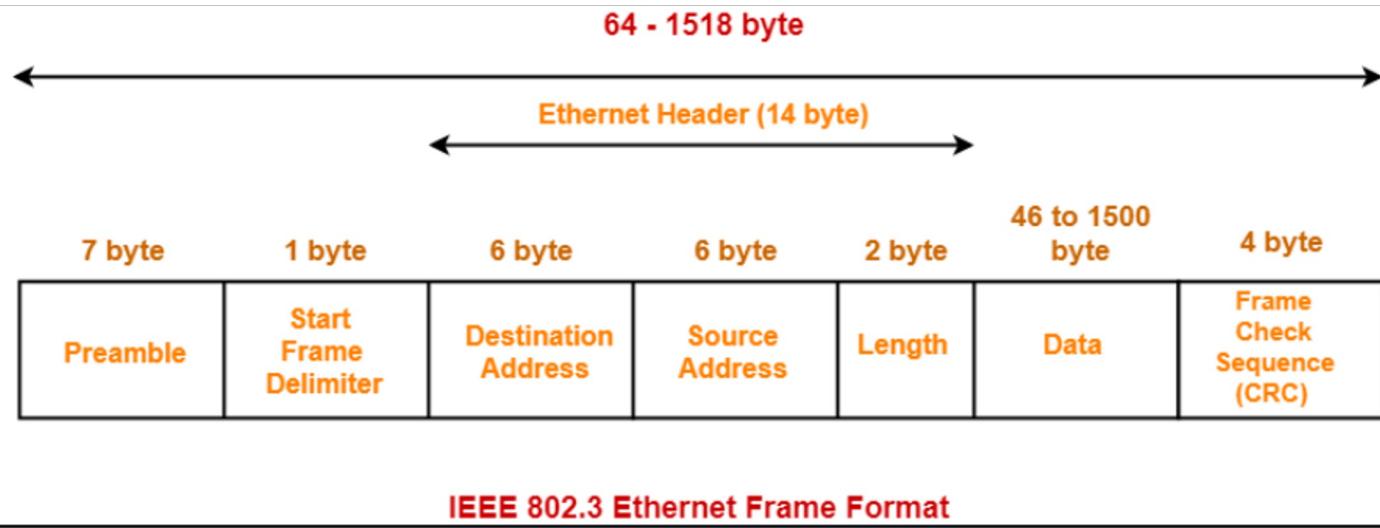
In a wired ethernet network the computers are connected to each other using twisted pair wiring via a router or switch.



Local Area Networks

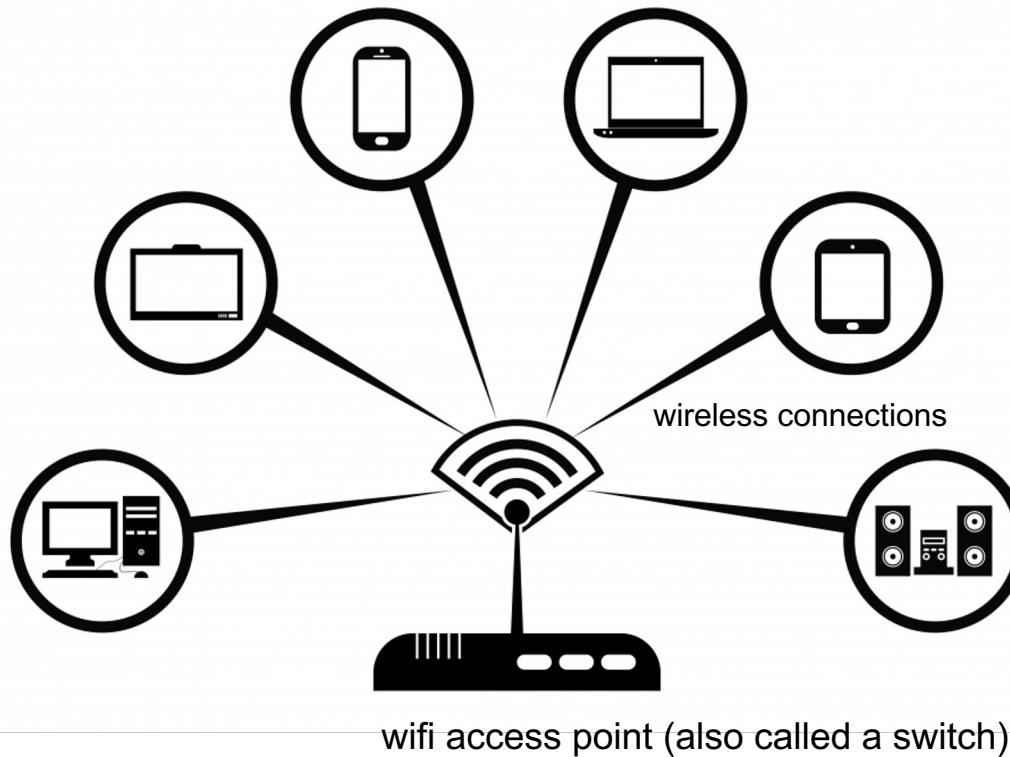
A wifi network in your home carries the packets using wireless connections. This consists of a wifi access point or switch that each device (computer/phone/tablet/etc) connects to using wireless transmission.

In wired and wireless ethernet networks the data packet is carried in an ethernet **frame** which is another sort of packet. So we essentially have a packet (data) inside another packet (ethernet).



Source and Destination addresses are 6 bytes (48 bits) long and are unique.

WiFi Network



Ethernet

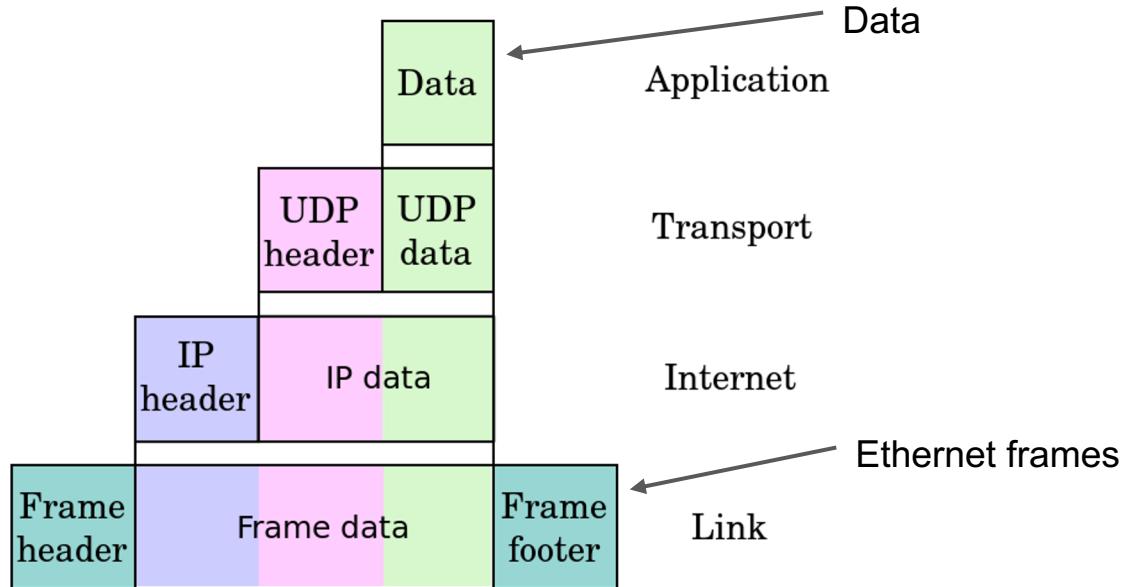
Ethernet packets or frames are carried point to point only. From a device to the access point or switch. Whereas the data packets are forwarded on to other switches and eventually reach their destination.

The data packets may make many hops along the route to the destination. For each hop they will be carried by a different lower level network message.

This illustrates a very important concept in computer networking called ***layering***.

Layering

Network services often have many layers where a message is carried inside another message which is carried inside another and so on.



Internet History (continued)

The early internet packet switching technique was developed by both the British and Americans at about the same time. The first production network was in Britain but the system that developed into today's internet came from the USA.

The Advanced Research Project Agency (ARPA) in the USA funded a major research project on packet switched networks in the 1960's at the height of the cold war since they gave the ability to withstand major disruption in a nuclear war. This first network was called ARPANET and started in 1969. It linked together Universities and research centres across the USA. In the late 1970's after 10 years of development, the basic protocols were revised and at the beginning of 1983 the network switched over to the system we use today.

The first internet link to Australia was established in 1989 by a team from the University of Sydney, University of Melbourne and NASA.

Internet Packet - IP

The data stream is divided into chunks called "internet packets" or IP.

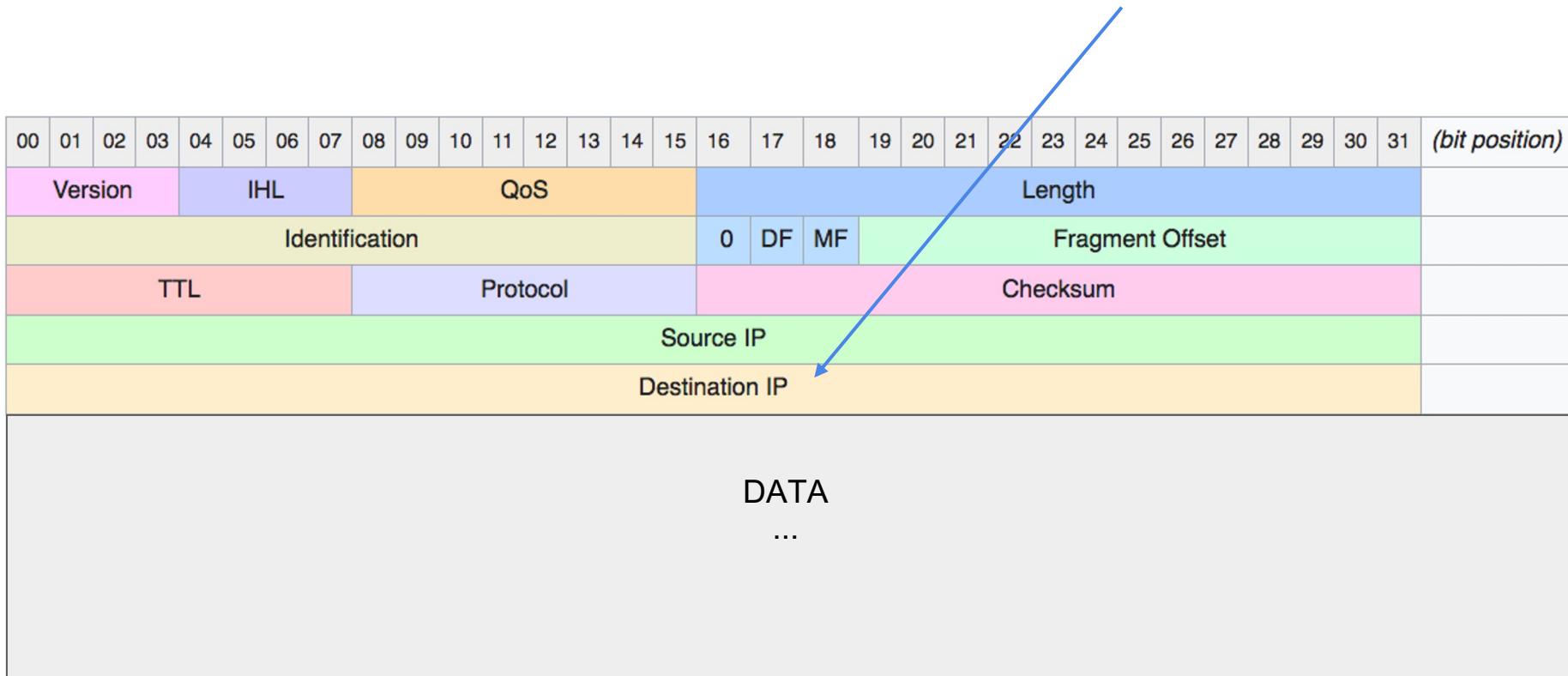
Internet packets can be up to 645536 (64x1024 or 64K) bytes long but are typically much smaller eg 1500 bytes.

The packet consists of a header containing the length and other information about the packet including the destination address and source address. This is followed by the actual data. The header is 20 bytes long.

An important part of the packet are the address fields.

Internet Packet Format

source and destination addresses



Internet Packet Addressing

The source and destination address are 32 bits or 4 bytes each.

The convention is that an internet address is written as four decimal numbers each in the range 0-255 separated by dots. This is a lot easier to write than a 32 bit binary or hexadecimal number.

For example, a machine in the School of IT has the address 129.78.8.1

An IP address is used to identify each computer and also to determine the **route** to a computer.

Internet Protocol

A network *protocol* is the set of rules for controlling the sequence of packets that are exchanged between computers.

The network protocols have to be implemented by all the computers on the network. The Internet has a set of these protocols that govern all the layers of the network.

History: Internet Protocol Documentation

As the internet protocols and procedures were developed they were documented in a set of documents called "Request For Comments" or RFC. The name comes from the fact that they were used originally for proposals that eventually became internet standards.

These documents are all online at <https://www.rfc-editor.org/> and available in various formats but the traditional standard form is plain text.

The internet tradition is that protocols are developed by committees that anyone can join through an organisation called the Internet Engineering Task Force or IETF.

Internet protocol

A common requirement in computer to computer communication is to send a large message or a stream of data. We can use packet switching to break up the data stream and send the packets independently but this alone can have problems.

If a packet is lost or corrupted due to a hardware failure (electrical interference ...), or congestion is detected and the packet travels via a longer route, this can cause packets to fail to arrive or to arrive out of order. We can detect most data errors at the Internet Packet level using a *checksum*.

How can we recover when an error is detected?

Transmission Control Protocol - TCP

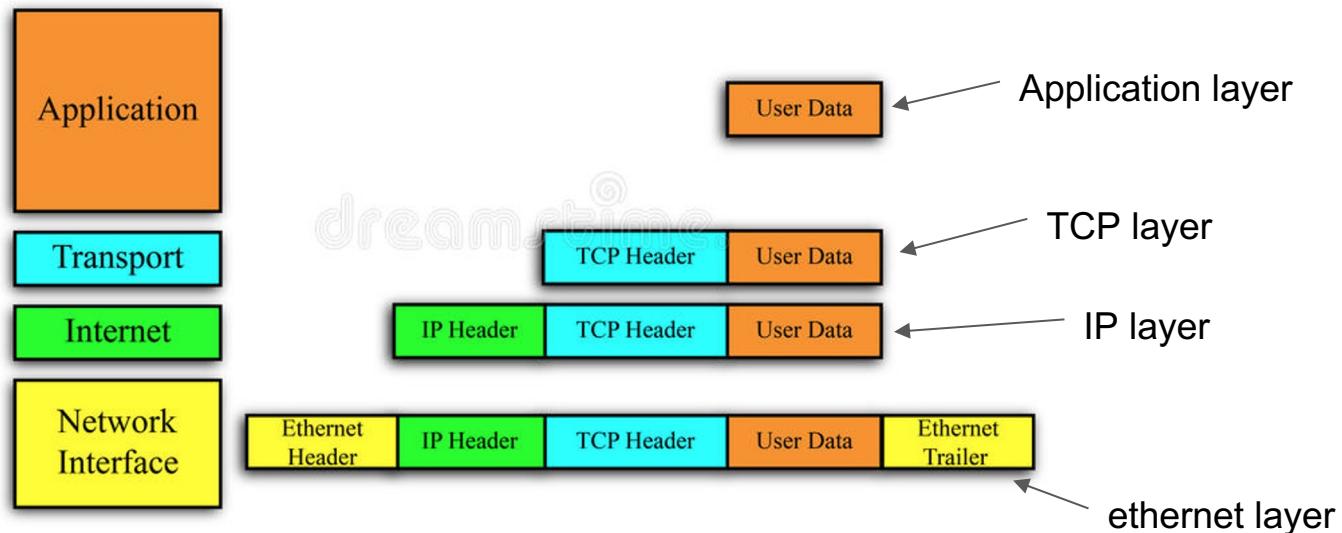
The solution is to create a protocol layered on top of Internet Packets, that carries the data packets - layering!

This protocol maintains a sequence number in each packet so out-of-order packets can be reassembled in the correct order, a way of requesting retransmission of packets with errors or packets that don't arrive in a reasonable time, and a way of acknowledging packets received correctly.

The protocol is called the Transmission Control Protocol or TCP and is commonly called "TCP/IP".

TCP/IP Network Layering

TCP/IP Network Model Encapsulation



Application Layer

What I have been calling "User Data" is actually more layers since it is usually data for an application. For example a web page or a video stream etc.

These applications will usually have their own protocols with packet formats and rules for exchanging messages.

Still to come ...

- Routing of packets
 - in the LAN
 - in the WAN
- Domain Name Service (DNS)
- Application layer protocols
 - HTTP
 - SMTP
 - MAIL - MIME



The Shell if statement

Shell If statement

The shell has an "if" statement like conventional programming languages. The statement tests the return value of a program:

if the program returns 0 it means TRUE, non-zero means FALSE

```
if test $NAME == Bob
then
    echo name is ok
fi
```

Shell If statement

The shell has an "if" statement like conventional programming languages. The statement tests the return value of a program:

if the program returns 0 it means TRUE, non-zero means FALSE

keyword

```
if test $NAME == Bob
```

then

```
    echo name is ok
```

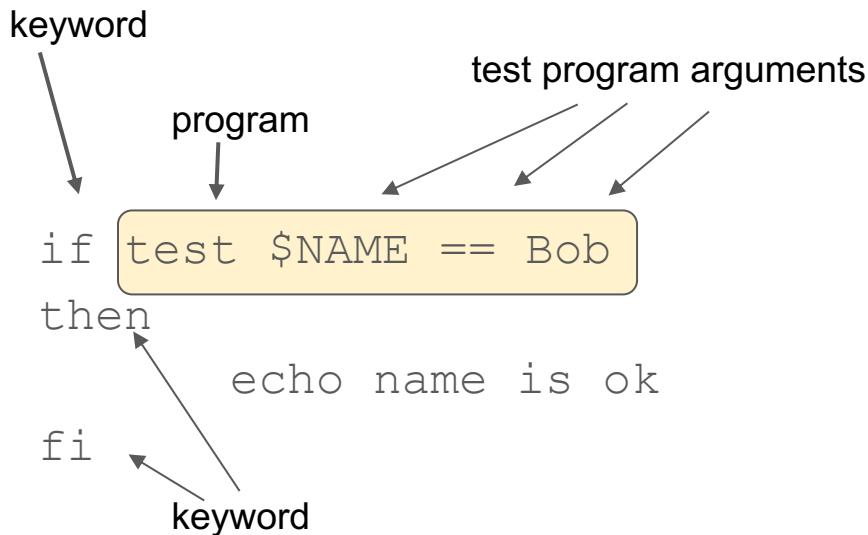
fi

keyword

Shell If statement

The shell has an "if" statement like conventional programming languages. The statement is testing the return value of a program:

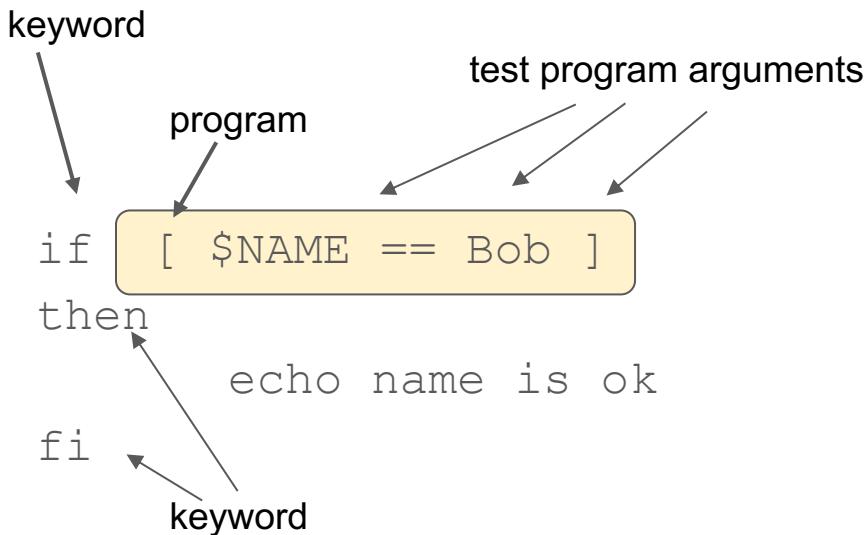
0 means TRUE, non-zero means FALSE



Shell If statement

The shell has an "if" statement like conventional programming languages. The statement is testing the return value of a program:

0 means TRUE, non-zero means FALSE



Note: The character "[" is a synonym for "test". The character "]" is ignored by "test"

Shell If statement

In addition to a `then` part, the `if` statement can also have an `else` part like most programming languages. Also, rather use nested if statements, it can have an `elif` part.

Here is an example:

```
if      [ $NAME == Bob ]
then
        echo name is ok
elif    [ $NAME == Alice ]
        echo name is good
else
        echo Wrong name
fi
```

Shell If statement

Here is the general form:

```
if test-commands; then
    consequent-commands;
[elif more-test-commands; then
    more-consequents; ]
[else alternate-consequents; ]
fi
```



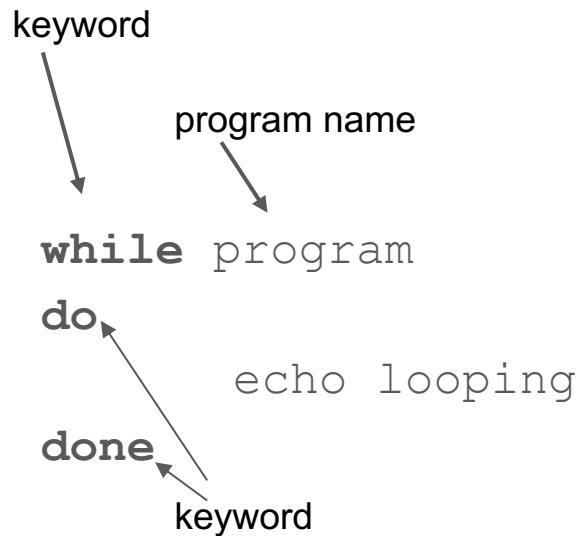
#31932011



The Shell
While

Shell **while** statement

The shell has a "while" statement like conventional programming languages. The statement repeats a loop while some program returns True/success:



General form is: **while** program; **do** statement list **done**

Before we begin the proceedings, I would like to acknowledge and pay respect to the traditional owners of the land: the Gadigal people of the Eora Nation. It is upon their ancestral lands that the University of Sydney is built.

As we share our own knowledge, teaching, learning and research practices within this university may we also pay respect to the knowledge embedded forever within the Aboriginal Custodianship of Country.

INFO1112

Week 1 Whole Class Session

Dr Nazanin Borhan





Welcome to INFO1112

- introductions
- Administrative info
 - Covid University policies
 - timetable
 - assessment
 - special consideration
- Intro to the course
- Python demo

Dr Nazanin Borhan

PhD in Computer Science

**Current research in "Security"
and "Data Science"**

**Taught a large range of
courses: programming,
security, data analytics,
networks, etc**



Teaching Team

Our team of tutors is listed on the Canvas page of the course and is led by the Teaching Assistant **Tiancheng (Michael) Mai.**



THE UNIVERSITY OF
SYDNEY



Follow University's COVID safety precautions



Stay home if you are sick



Wash hands regularly



Avoid physical greetings



Cough or sneeze into your elbow or tissue



Keep 1.5m away from others where possible



Avoid crowding entrances and exits

sydney.edu.au/covid-19



Feeling unwell?

- **Stay at home**
 - if you are feeling unwell with any COVID-19 symptoms
 - If you have been directed to self-isolate
- **Get tested**
 - If you are feeling unwell with COVID-19 symptoms, please get tested as soon as possible
- **Did you test positive?**

Yes? If you have visited campus within the last 72 hours you must advise the University via:

 - email covid19.taskforce@sydney.edu.au, or
 - call +61 2 9351 2000 (select option 1).
- **Stay informed**
 - Monitor [the list of confirmed COVID case locations on campus page](#) to check for potential exposure and [follow NSW Health isolation and testing requirements.](#)

COVID-19 support and care

- Most large lectures will be delivered online, and accommodations will be made for international students who have not yet returned to Australia.
- If you become infected with COVID-19 during the semester, or need to isolate, please notify your unit of study coordinator, as with any unexpected absence.
- If COVID-19 isolation or illness impacts assessment, use the usual mechanisms including simple extensions and special consideration to arrange reasonable adjustments. Visit <https://www.sydney.edu.au/covid-19/students/study-information/test-exams-assessment.html#consideration>
- Further information on student support can be found on the University website at <https://www.sydney.edu.au/covid-19/students/support-wellbeing.html>
- Other helpful study information can be found on the website at <https://www.sydney.edu.au/covid-19/students/study-information.html>

INFO1112: Places

- Whole class session: Monday 12-2pm online via Zoom
 - see Canvas for zoom URL
- Lab: depends on your timetable
 - Go to the lab you are scheduled for
 - If for some reason you miss it, you can attend a later lab session *if there is space and the tutor agrees*, but ask the tutor before taking a "seat"
 - zoom URLs for online lab classes is on Canvas
- Do not miss class, except for illness, emergencies, etc
 - BUT if you feel even slightly ill (headache, runny nose...) **DO NOT ATTEND A FACE TO FACE LAB**

Tips for online Lab activities

- Remember that you are still in a (virtual) space with other students.
- Mute your microphone when not speaking.
- Use headphones - you'll disturb others less.
- If you are speaking to the camera, make eye contact with the camera (and therefore your classmates and teacher).
- Try not to talk over someone else.
- Ask questions on the Edstem discussion.
- please open INFO1112 on Canvas during the tutorial session

Expectations

- Students attend scheduled classes, and devote an extra 6-9 hrs per week
 - doing homework and assignments
 - preparing for classes, watching pre-record video before class
 - revising and integrating the ideas
 - practice and self-assess
- Students are responsible learners
 - Participate in classes, constructively
 - Respect for one another (criticize ideas, not people)
 - Humility: none of us knows it all; each of us knows valuable things
 - Check Canvas site regularly!
 - Notify me or your tutor if there are difficulties - use the Ed forum or email if urgent

Activity – Padlet Meet and Greet

What would you like to share about yourself with others in this class? [Padlet Page](#)

sydney.padlet.org/nazaninborhan1/malyshgzxhroq1ec

Nazanin Borhan • 1m
INFO1112 - S2 Week 1 - Meet and greet

Nazanin Borhan
What would you like to share about yourself with others in this class?

To make a post, please click on the + button on the page and write your name in the Title section, then share your information at the bottom.
You can also comment on others' posts and like their posts.

0 Add comment

Nazanin Borhan
I am the lecturer for this course. I started teaching in USYD this

INFO1112 Assessment

Assessment Information in Canvas

Description	Marks	Type	Due
6 homework exercises	30 (5x6)	shell/Ed	every second week starting wk 2
assignment 1	10	intro python/Ed	week 4
assignment 2	20	python/Ed	week 8
assignment 3	20	python/Ed	week 13
mid semester quiz	10	gradescope	lecture of wk 7
end of semester quiz	10	gradescope	lecture of wk 13
Total	100		

BREAK



Late submissions in INFO1112

- **There is not late submission accepted for Homeworks.**
- Late submissions for assignments: Suppose you hand in assignments after the deadline:
 - If you have not been granted special consideration a penalty of 5% of the maximum marks will be taken per day (or part) late. After ten days, you will be awarded a mark of zero.
 - *e.g. If an assignment is worth 40% of the final mark and you are **one hour** late submitting, then the maximum marks possible would be 38%.*
 - *e.g. If an assignment is worth 40% of the final mark and you are 28 hours late submitting, then the maximum marks possible marks would be 36%.*
- Warning: submission systems get very slow near deadlines
- Submit early; you can resubmit if there is time before the deadline

Special Consideration (University policy)

- If your performance on assessments is affected by illness or misadventure
- Follow proper bureaucratic procedures
 - Have professional practitioner sign special USyd form
 - Submit application for special consideration online
 - Note you have only a quite short deadline for applying
 - http://sydney.edu.au/current_students/special_consideration/
- Also, notify me by email as soon as *anything begins to go wrong*

Academic Integrity (University policy)

- The University of Sydney is unequivocally opposed to, and intolerant of, plagiarism and academic dishonesty.
 - Academic dishonesty means seeking to obtain, or obtaining academic advantage for oneself or for others (including in the assessment or publication of work) by dishonest or unfair means.
 - Plagiarism means presenting another person's work as one's own work by presenting, copying or reproducing it without appropriate acknowledgement of the source. [from site below]
- <http://sydney.edu.au/elearning/student/EI/index.shtml>
- Submitted work is compared against other work (from students, the internet, etc)
 - **Ed will do this for all assessments in INFO1112**
- Penalties for academic dishonesty or plagiarism can be severe
- You must complete the self-education module AHEM1001 (required to pass INFO1112)

Pre semester Survey

Statistics



THE UNIVERSITY OF
SYDNEY



Course

I N F O

The Course (finally!)

- "bits to applications"
- How does it work?
- a *spiral* approach - almost all the topics covered in the course will be covered in much greater depth in other courses
- INFO1112 is designed to introduce you to the underlying technical ideas
- ... with a bit of history as well.

Whole class Survey

In Canvas

A bit about you and your technology

4 minutes survey



THE UNIVERSITY OF
SYDNEY



The Course Components

- a set of videos to watch each week
- a whole-class session every Monday 12-2pm for demos and Q&A
- a 2 hour lab session each week (see your timetable)
 - see your timetable if you are in a face-to-face lab
 - you should now have the zoom URL if online
- homework due in the lab sessions in weeks 2 to 12
- three assignments during semester
- mid and end-semester quizzes (done on Gradescope)

INFO1112 Weekly Outline

The topics for each week in 2022 will be:

- W1: representations (binary, numbers, characters, instructions), operating system introduction
- W2: processes
- W3: file system, name space
- W4: the boot sequence
- W5: virtual machines
- W6: networks
- W7: internet protocol
- mid-semester break
- W8: domain name service, application layer
- W9: computer and network security
- W10: cloud computing
- W11: window systems and Android
- W12: Micro-services
- W13: course review



Study Research Engage with us About us News & opinion Q

Current students

[Units](#) / [INFO1112](#) / Semester 2 2021 [Normal day]

Unit of study...

INFO1112: Computing 1B OS and Network Platforms

Overview

The unit introduces principles and concepts of modern computer systems, including mobile computers and the Internet, to provide students with fundamental knowledge of the environments in which modern, networked applications operate. Students will have basic knowledge to understand how computers work and are aware of principles and concepts they are likely to encounter in their career. The unit covers: Principles of operating systems and the way applications interact with the OS, including the particularities of modern operating systems for mobile devices Principles of computer networking, including mobile networking Writing applications that use facilities of the OS and networking, including understanding the challenges that are common in distributed systems

[Details](#) [Enrolment rules](#) [Teaching staff and contact details](#)

On this page

- [Overview](#)
- [Assessment](#)
- [Weekly schedule](#)
- [Learning outcomes](#)
- [Closing the loop](#)
- [Additional information](#)

Useful links

- [myuni](#) →
- [Proposed timetables for 2022](#) →
- [Library](#) →

Week 1

Short Segments this week:

- › 1.1 binary numbers
- › 1.2 number representation
- › 1.3 character representation
- › 1.4 computer instructions
- › 1.5 operating systems

Lab session **this week**: revision of shell commands.

No homework this week but there is homework due in your lab session in Week 2.

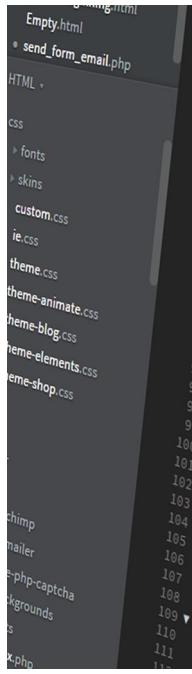
See Canvas for details and links to all material.

A cluster of colorful question marks and exclamation points on a brown background.

Questions?

Homeworks in Edstem

Late submissions are not accepted



```
Empty.html
send_form_email.php
HTML
css
> fonts
> skins
custom.css
ie.css
theme.css
theme-animate.css
theme-blog.css
theme-elements.css
theme-shop.css
.
.
.
chimp
mailer
e-php-captcha
backgrounds
s
x.php

78     ' . ltrim(preg_replace('/\\\\\\\\', '/', $image_src), '/');
79     $SESSION['_CAPTCHA']['config'] = serialize($captcha_config);
80   )
81   return array(
82     'code' => $captcha_config['code'],
83     'image_src' => $image_src,
84   );
85 }
86
87
88 if( !function_exists('hex2rgb') ) {
89   function hex2rgb($hex_str, $return_string = false, $separator = ',') {
90     $hex_str = preg_replace("/[^0-9A-Fa-f]/", '', $hex_str); // Gets a proper hex string
91     $rgb_array = array();
92     if( strlen($hex_str) == 6 ) {
93       $color_val = hexdec($hex_str);
94       $rgb_array['r'] = 0xFF & ($color_val >> 0x10);
95       $rgb_array['g'] = 0xFF & ($color_val >> 0x8);
96       $rgb_array['b'] = 0xFF & $color_val;
97     } elseif( strlen($hex_str) == 3 ) {
98       $rgb_array['r'] = hexdec(str_repeat(substr($hex_str, 0, 1), 2));
99       $rgb_array['g'] = hexdec(str_repeat(substr($hex_str, 1, 1), 2));
100      $rgb_array['b'] = hexdec(str_repeat(substr($hex_str, 2, 1), 2));
101    } else {
102      return false;
103    }
104  }
105  return $return_string ? implode($separator, $rgb_array) : '';
106 }
107 // Draw the image
108 if( isset($_GET['c']) )
109   .
110   .
111   .
112   .
113   .
114   .
115   .
116   .
117   .
118   .
119   .
120   .
121   .
122   .
123   .
124   .
125   .
126   .
127   .
128   .
129   .
130   .
131   .
132   .
133   .
134   .
135   .
136   .
137   .
138   .
139   .
140   .
141   .
142   .
143   .
144   .
145   .
146   .
147   .
148   .
149   .
150   .
151   .
152   .
153   .
154   .
155   .
156   .
157   .
158   .
159   .
160   .
161   .
162   .
163   .
164   .
165   .
166   .
167   .
168   .
169   .
170   .
171   .
172   .
173   .
174   .
175   .
176   .
177   .
178   .
179   .
180   .
181   .
182   .
183   .
184   .
185   .
186   .
187   .
188   .
189   .
190   .
191   .
192   .
193   .
194   .
195   .
196   .
197   .
198   .
199   .
200   .
201   .
202   .
203   .
204   .
205   .
206   .
207   .
208   .
209   .
210   .
211   .
212   .
213   .
214   .
215   .
216   .
217   .
218   .
219   .
220   .
221   .
222   .
223   .
224   .
225   .
226   .
227   .
228   .
229   .
230   .
231   .
232   .
233   .
234   .
235   .
236   .
237   .
238   .
239   .
240   .
241   .
242   .
243   .
244   .
245   .
246   .
247   .
248   .
249   .
250   .
251   .
252   .
253   .
254   .
255   .
256   .
257   .
258   .
259   .
260   .
261   .
262   .
263   .
264   .
265   .
266   .
267   .
268   .
269   .
270   .
271   .
272   .
273   .
274   .
275   .
276   .
277   .
278   .
279   .
280   .
281   .
282   .
283   .
284   .
285   .
286   .
287   .
288   .
289   .
290   .
291   .
292   .
293   .
294   .
295   .
296   .
297   .
298   .
299   .
299   .
300   .
301   .
302   .
303   .
304   .
305   .
306   .
307   .
308   .
309   .
309   .
310   .
311   .
312   .
313   .
314   .
315   .
316   .
317   .
318   .
319   .
319   .
320   .
321   .
322   .
323   .
324   .
325   .
326   .
327   .
328   .
329   .
329   .
330   .
331   .
332   .
333   .
334   .
335   .
336   .
337   .
338   .
339   .
339   .
340   .
341   .
342   .
343   .
344   .
345   .
346   .
347   .
348   .
349   .
349   .
350   .
351   .
352   .
353   .
354   .
355   .
356   .
357   .
358   .
359   .
359   .
360   .
361   .
362   .
363   .
364   .
365   .
366   .
367   .
368   .
369   .
369   .
370   .
371   .
372   .
373   .
374   .
375   .
376   .
377   .
378   .
379   .
379   .
380   .
381   .
382   .
383   .
384   .
385   .
386   .
387   .
388   .
389   .
389   .
390   .
391   .
392   .
393   .
394   .
395   .
396   .
397   .
398   .
399   .
399   .
400   .
401   .
402   .
403   .
404   .
405   .
406   .
407   .
408   .
409   .
409   .
410   .
411   .
412   .
413   .
414   .
415   .
416   .
417   .
418   .
419   .
419   .
420   .
421   .
422   .
423   .
424   .
425   .
426   .
427   .
428   .
429   .
429   .
430   .
431   .
432   .
433   .
434   .
435   .
436   .
437   .
438   .
439   .
439   .
440   .
441   .
442   .
443   .
444   .
445   .
446   .
447   .
448   .
449   .
449   .
450   .
451   .
452   .
453   .
454   .
455   .
456   .
457   .
458   .
459   .
459   .
460   .
461   .
462   .
463   .
464   .
465   .
466   .
467   .
468   .
469   .
469   .
470   .
471   .
472   .
473   .
474   .
475   .
476   .
477   .
478   .
479   .
479   .
480   .
481   .
482   .
483   .
484   .
485   .
486   .
487   .
488   .
489   .
489   .
490   .
491   .
492   .
493   .
494   .
495   .
496   .
497   .
498   .
499   .
499   .
500   .
501   .
502   .
503   .
504   .
505   .
506   .
507   .
508   .
509   .
509   .
510   .
511   .
512   .
513   .
514   .
515   .
516   .
517   .
518   .
519   .
519   .
520   .
521   .
522   .
523   .
524   .
525   .
526   .
527   .
528   .
529   .
529   .
530   .
531   .
532   .
533   .
534   .
535   .
536   .
537   .
538   .
539   .
539   .
540   .
541   .
542   .
543   .
544   .
545   .
546   .
547   .
548   .
549   .
549   .
550   .
551   .
552   .
553   .
554   .
555   .
556   .
557   .
558   .
559   .
559   .
560   .
561   .
562   .
563   .
564   .
565   .
566   .
567   .
568   .
569   .
569   .
570   .
571   .
572   .
573   .
574   .
575   .
576   .
577   .
578   .
579   .
579   .
580   .
581   .
582   .
583   .
584   .
585   .
586   .
587   .
588   .
589   .
589   .
590   .
591   .
592   .
593   .
594   .
595   .
596   .
597   .
598   .
599   .
599   .
600   .
601   .
602   .
603   .
604   .
605   .
606   .
607   .
608   .
609   .
609   .
610   .
611   .
612   .
613   .
614   .
615   .
616   .
617   .
618   .
619   .
619   .
620   .
621   .
622   .
623   .
624   .
625   .
626   .
627   .
628   .
629   .
629   .
630   .
631   .
632   .
633   .
634   .
635   .
636   .
637   .
638   .
639   .
639   .
640   .
641   .
642   .
643   .
644   .
645   .
646   .
647   .
648   .
649   .
649   .
650   .
651   .
652   .
653   .
654   .
655   .
656   .
657   .
658   .
659   .
659   .
660   .
661   .
662   .
663   .
664   .
665   .
666   .
667   .
668   .
669   .
669   .
670   .
671   .
672   .
673   .
674   .
675   .
676   .
677   .
678   .
679   .
679   .
680   .
681   .
682   .
683   .
684   .
685   .
686   .
687   .
688   .
689   .
689   .
690   .
691   .
692   .
693   .
694   .
695   .
696   .
697   .
698   .
699   .
699   .
700   .
701   .
702   .
703   .
704   .
705   .
706   .
707   .
708   .
709   .
709   .
710   .
711   .
712   .
713   .
714   .
715   .
716   .
717   .
718   .
719   .
719   .
720   .
721   .
722   .
723   .
724   .
725   .
726   .
727   .
728   .
729   .
729   .
730   .
731   .
732   .
733   .
734   .
735   .
736   .
737   .
738   .
739   .
739   .
740   .
741   .
742   .
743   .
744   .
745   .
746   .
747   .
748   .
749   .
749   .
750   .
751   .
752   .
753   .
754   .
755   .
756   .
757   .
758   .
759   .
759   .
760   .
761   .
762   .
763   .
764   .
765   .
766   .
767   .
768   .
769   .
769   .
770   .
771   .
772   .
773   .
774   .
775   .
776   .
777   .
778   .
779   .
779   .
780   .
781   .
782   .
783   .
784   .
785   .
786   .
787   .
788   .
789   .
789   .
790   .
791   .
792   .
793   .
794   .
795   .
796   .
797   .
798   .
799   .
799   .
800   .
801   .
802   .
803   .
804   .
805   .
806   .
807   .
808   .
809   .
809   .
810   .
811   .
812   .
813   .
814   .
815   .
816   .
817   .
818   .
819   .
819   .
820   .
821   .
822   .
823   .
824   .
825   .
826   .
827   .
828   .
829   .
829   .
830   .
831   .
832   .
833   .
834   .
835   .
836   .
837   .
838   .
839   .
839   .
840   .
841   .
842   .
843   .
844   .
845   .
846   .
847   .
848   .
849   .
849   .
850   .
851   .
852   .
853   .
854   .
855   .
856   .
857   .
858   .
859   .
859   .
860   .
861   .
862   .
863   .
864   .
865   .
866   .
867   .
868   .
869   .
869   .
870   .
871   .
872   .
873   .
874   .
875   .
876   .
877   .
878   .
878   .
879   .
880   .
881   .
882   .
883   .
884   .
885   .
886   .
887   .
888   .
888   .
889   .
889   .
890   .
891   .
892   .
893   .
894   .
895   .
896   .
897   .
898   .
898   .
899   .
900   .
901   .
902   .
903   .
904   .
905   .
906   .
907   .
908   .
909   .
909   .
910   .
911   .
912   .
913   .
914   .
915   .
916   .
917   .
918   .
919   .
919   .
920   .
921   .
922   .
923   .
924   .
925   .
926   .
927   .
928   .
929   .
929   .
930   .
931   .
932   .
933   .
934   .
935   .
936   .
937   .
938   .
939   .
939   .
940   .
941   .
942   .
943   .
944   .
945   .
946   .
947   .
948   .
949   .
949   .
950   .
951   .
952   .
953   .
954   .
955   .
956   .
957   .
958   .
959   .
959   .
960   .
961   .
962   .
963   .
964   .
965   .
966   .
967   .
968   .
969   .
969   .
970   .
971   .
972   .
973   .
974   .
975   .
976   .
977   .
978   .
978   .
979   .
980   .
981   .
982   .
983   .
984   .
985   .
986   .
987   .
988   .
988   .
989   .
989   .
990   .
991   .
992   .
993   .
994   .
995   .
996   .
997   .
998   .
999   .
999   .
1000  .
1001  .
1002  .
1003  .
1004  .
1005  .
1006  .
1007  .
1008  .
1009  .
1009  .
1010  .
1011  .
1012  .
1013  .
1014  .
1015  .
1016  .
1017  .
1018  .
1019  .
1019  .
1020  .
1021  .
1022  .
1023  .
1024  .
1025  .
1026  .
1027  .
1028  .
1029  .
1029  .
1030  .
1031  .
1032  .
1033  .
1034  .
1035  .
1036  .
1037  .
1038  .
1039  .
1039  .
1040  .
1041  .
1042  .
1043  .
1044  .
1045  .
1046  .
1047  .
1048  .
1049  .
1049  .
1050  .
1051  .
1052  .
1053  .
1054  .
1055  .
1056  .
1057  .
1058  .
1059  .
1059  .
1060  .
1061  .
1062  .
1063  .
1064  .
1065  .
1066  .
1067  .
1068  .
1069  .
1069  .
1070  .
1071  .
1072  .
1073  .
1074  .
1075  .
1076  .
1077  .
1078  .
1078  .
1079  .
1080  .
1081  .
1082  .
1083  .
1084  .
1085  .
1086  .
1087  .
1088  .
1088  .
1089  .
1090  .
1091  .
1092  .
1093  .
1094  .
1095  .
1096  .
1097  .
1097  .
1098  .
1099  .
1100  .
1101  .
1102  .
1103  .
1104  .
1105  .
1106  .
1107  .
1108  .
1109  .
1109  .
1110  .
1111  .
1112  .
1113  .
1114  .
1115  .
1116  .
1117  .
1118  .
1119  .
1119  .
1120  .
1121  .
1122  .
1123  .
1124  .
1125  .
1126  .
1127  .
1128  .
1129  .
1129  .
1130  .
1131  .
1132  .
1133  .
1134  .
1135  .
1136  .
1137  .
1138  .
1139  .
1139  .
1140  .
1141  .
1142  .
1143  .
1144  .
1145  .
1146  .
1147  .
1148  .
1148  .
1149  .
1150  .
1151  .
1152  .
1153  .
1154  .
1155  .
1156  .
1157  .
1158  .
1159  .
1159  .
1160  .
1161  .
1162  .
1163  .
1164  .
1165  .
1166  .
1167  .
1168  .
1169  .
1169  .
1170  .
1171  .
1172  .
1173  .
1174  .
1175  .
1176  .
1177  .
1178  .
1178  .
1179  .
1180  .
1181  .
1182  .
1183  .
1184  .
1185  .
1186  .
1187  .
1188  .
1188  .
1189  .
1190  .
1191  .
1192  .
1193  .
1194  .
1195  .
1196  .
1197  .
1197  .
1198  .
1199  .
1200  .
1201  .
1202  .
1203  .
1204  .
1205  .
1206  .
1207  .
1208  .
1209  .
1209  .
1210  .
1211  .
1212  .
1213  .
1214  .
1215  .
1216  .
1217  .
1218  .
1219  .
1219  .
1220  .
1221  .
1222  .
1223  .
1224  .
1225  .
1226  .
1227  .
1228  .
1229  .
1229  .
1230  .
1231  .
1232  .
1233  .
1234  .
1235  .
1236  .
1237  .
1238  .
1239  .
1239  .
1240  .
1241  .
1242  .
1243  .
1244  .
1245  .
1246  .
1247  .
1248  .
1249  .
1249  .
1250  .
1251  .
1252  .
1253  .
1254  .
1255  .
1256  .
1257  .
1258  .
1259  .
1259  .
1260  .
1261  .
1262  .
1263  .
1264  .
1265  .
1266  .
1267  .
1268  .
1269  .
1269  .
1270  .
1271  .
1272  .
1273  .
1274  .
1275  .
1276  .
1277  .
1278  .
1278  .
1279  .
1280  .
1281  .
1282  .
1283  .
1284  .
1285  .
1286  .
1287  .
1287  .
1288  .
1289  .
1290  .
1291  .
1292  .
1293  .
1294  .
1295  .
1296  .
1297  .
1297  .
1298  .
1299  .
1300  .
1301  .
1302  .
1303  .
1304  .
1305  .
1306  .
1307  .
1308  .
1309  .
1309  .
1310  .
1311  .
1312  .
1313  .
1314  .
1315  .
1316  .
1317  .
1318  .
1319  .
1319  .
1320  .
1321  .
1322  .
1323  .
1324  .
1325  .
1326  .
1327  .
1328  .
1329  .
1329  .
1330  .
1331  .
1332  .
1333  .
1334  .
1335  .
1336  .
1337  .
1338  .
1339  .
1339  .
1340  .
1341  .
1342  .
1343  .
1344  .
1345  .
1346  .
1347  .
1348  .
1349  .
1349  .
1350  .
1351  .
1352  .
1353  .
1354  .
1355  .
1356  .
1357  .
1358  .
1359  .
1359  .
1360  .
1361  .
1362  .
1363  .
1364  .
1365  .
1366  .
1367  .
1368  .
1369  .
1369  .
1370  .
1371  .
1372  .
1373  .
1374  .
1375  .
1376  .
1377  .
1378  .
1378  .
1379  .
1380  .
1381  .
1382  .
1383  .
1384  .
1385  .
1386  .
1387  .
1388  .
1388  .
1389  .
1390  .
1391  .
1392  .
1393  .
1394  .
1395  .
1396  .
1397  .
1398  .
1398  .
1399  .
1400  .
1401  .
1402  .
1403  .
1404  .
1405  .
1406  .
1407  .
1408  .
1409  .
1409  .
1410  .
1411  .
1412  .
1413  .
1414  .
1415  .
1416  .
1417  .
1418  .
1419  .
1419  .
1420  .
1421  .
1422  .
1423  .
1424  .
1425  .
1426  .
1427  .
1428  .
1429  .
1429  .
1430  .
1431  .
1432  .
1433  .
1434  .
1435  .
1436  .
1437  .
1438  .
1439  .
1439  .
1440  .
1441  .
1442  .
1443  .
1444  .
1445  .
1446  .
1447  .
1448  .
1449  .
1449  .
1450  .
1451  .
1452  .
1453  .
1454  .
1455  .
1456  .
1457  .
1458  .
1459  .
1459  .
1460  .
1461  .
1462  .
1463  .
1464  .
1465  .
1466  .
1467  .
1468  .
1469  .
1469  .
1470  .
1471  .
1472  .
1473  .
1474  .
1475  .
1476  .
1477  .
1478  .
1478  .
1479  .
1480  .
1481  .
1482  .
1483  .
1484  .
1485  .
1486  .
1487  .
1488  .
1488  .
1489  .
1490  .
1491  .
1492  .
1493  .
1494  .
1495  .
1496  .
1497  .
1498  .
1498  .
1499  .
1500  .
1501  .
1502  .
1503  .
1504  .
1505  .
1506  .
1507  .
1508  .
1509  .
1509  .
1510  .
1511  .
1512  .
1513  .
1514  .
1515  .
1516  .
1517  .
1518  .
1519  .
1519  .
1520  .
1521  .
1522  .
1523  .
1524  .
1525  .
1526  .
1527  .
1528  .
1529  .
1529  .
1530  .
1531  .
1532  .
1533  .
1534  .
1535  .
1536  .
1537  .
1538  .
1539  .
1539  .
1540  .
1541  .
1542  .
1543  .
1544  .
1545  .
1546  .
1547  .
1548  .
1549  .
1549  .
1550  .
1551  .
1552  .
1553  .
1554  .
1555  .
1556  .
1557  .
1558  .
1559  .
1559  .
1560  .
1561  .
1562  .
1563  .
1564  .
1565  .
1566  .
1567  .
1568  .
1569  .
1569  .
1570  .
1571  .
1572  .
1573  .
1574  .
1575  .
1576  .
1577  .
1578  .
1578  .
1579  .
1580  .
1581  .
1582  .
1583  .
1584  .
1585  .
1586  .
1587  .
1588  .
1588  .
1589  .
1590  .
1591  .
1592  .
1593  .
1594  .
1595  .
1596  .
1597  .
1598  .
1598  .
1599  .
1600  .
1601  .
1602  .
1603  .
1604  .
1605  .
1606  .
1607  .
1608  .
1609  .
1609  .
1610  .
1611  .
1612  .
1613  .
1614  .
1615  .
1616  .
1617  .
1618  .
1619  .
1619  .
1620  .
1621  .
1622  .
1623  .
1624  .
1625  .
1626  .
1627  .
1628  .
1629  .
1629  .
1630  .
1631  .
1632  .
1633  .
1634  .
1635  .
1636  .
1637  .
1638  .
1639  .
1639  .
1640  .
1641  .
1642  .
1643  .
1644  .
1645  .
1646  .
1647  .
1648  .
1649  .
1649  .
1650  .
1651  .
1652  .
1653  .
1654  .
1655  .
1656  .
1657  .
1658  .
1659  .
1659  .
1660  .
1661  .
1662  .
1663  .
1664  .
1665  .
1666  .
1667  .
1668  .
1669  .
1669  .
1670  .
1671  .
1672  .
1673  .
1674  .
1675  .
1676  .
1677  .
1678  .
1678  .
1679  .
1680  .
1681  .
1682  .
1683  .
1684  .
1685  .
1686  .
1687  .
1688  .
1688  .
1689  .
1690  .
1691  .
1692  .
1693  .
1694  .
1695  .
1696  .
1697  .
1698  .
1698  .
1699  .
1700  .
1701  .
1702  .
1703  .
1704  .
1705  .
1706  .
1707  .
1708  .
1709  .
1709  .
1710  .
1711  .
1712  .
1713  .
1714  .
1715  .
1716  .
1717  .
1718  .
1719  .
1719  .
1720  .
1721  .
1722  .
1723  .
1724  .
1725  .
1726  .
1727  .
1728  .
1729  .
1729  .
1730  .
1731  .
1732  .
1733  .
1734  .
1735  .
1736  .
1737  .
1738  .
1739  .
1739  .
1740  .
1741  .
1742  .
1743  .
1744  .
1745  .
1746  .
1747  .
1748  .
1749  .
1749  .
1750  .
1751  .
1752  .
1753  .
1754  .
1755  .
1756  .
1757  .
1758  .
1759  .
1759  .
1760  .
1761  .
1762  .
1763  .
1764  .
1765  .
1766  .
1767  .
1768  .
1769  .
1769  .
1770  .
1771  .
1772  .
1773  .
1774  .
1775  .
1776  .
1777  .
1778  .
1778  .
1779  .
1780  .
1781  .
1782  .
1783  .
1784  .
1785  .
1786  .
1787  .
1788  .
1788  .
1789  .
1790  .
1791  .
1792  .
1793  .
1794  .
1795  .
1796  .
1797  .
1798  .
1798  .
1799  .
1800  .
1801  .
1802  .
1803  .
1804  .
1805  .
1806  .
1807  .
1808  .
1809  .
1809  .
1810  .
1811  .
1812  .
1813  .
1814  .
1815  .
1816  .
1817  .
1818  .
1819  .
1819  .
1820  .
1821  .
1822  .
1823  .
1824  .
1825  .
1826  .
1827  .
1828  .
1829  .
1829  .
1830  .
1831  .
1832  .
1833  .
1834  .
1835  .
1836  .
1837  .
1838  .
1839  .
1839  .
1840  .
1841  .
1842  .
1843  .
1844  .
1845  .
1846  .
1847  .
1848  .
1849  .
1849  .
1850  .
1851  .
1852  .
1853  .
1854  .
1855  .
1856  .
1857  .
1858  .
1859  .
1859  .
1860  .
1861  .
1862  .
1863  .
1864  .
1865  .
1866  .
1867  .
1868  .
1869  .
1869  .
1870  .
1871  .
1872  .
1873  .
1874  .
1875  .
1876  .
1877  .
1878  .
1878  .
1879  .
1880  .
1881  .
1882  .
1883  .
1884  .
1885  .
1886  .
1887  .
1888  .
1888  .
1889  .
1890  .
1891  .
1892  .
1893  .
1894  .
1895  .
1896  .
1897  .
1898  .
1898  .
1899  .
1900  .
1901  .
1902  .
1903  .
1904  .
1905  .
1906  .
1907  .
1908  .
1909  .
1909  .
1910  .
1911  .
1912  .
1913  .
1914  .
1915  .
1916  .
1917  .
1918  .
1919  .
1919  .
1920  .
1921  .
1922  .
1923  .
1924  .
1925  .
1926  .
1927  .
1928  .
1929  .
1929  .
1930  .
1931  .
1932  .
1933  .
1934  .
1935  .
1936  .
1937  .
1938  .
1939  .
1939  .
1940  .
1941  .
1942  .
1943  .
1944  .
1945  .
1946  .
1947  .
1948  .
1949  .
1949  .
1950  .
1951  .
1952  .
1953  .
1954  .
1955  .
1956  .
1957  .
1958  .
1959  .
1959  .
1960  .
1961  .
1962  .
1963  .
1964  .
1965  .
1966  .
1967  .
1968  .
1969  .
1969  .
1970  .
1971  .
1972  .
1973  .
1974  .
1975  .
1976  .
1977  .
1978  .
1978  .
1979  .
1980  .
1981  .
1982  .
1983  .
1984  .
1985  .
1986  .
1987  .
1988  .
1988  .
1989  .
1990  .
1991  .
1992  .
1993  .
1994  .
1995  .
1996  .
1997  .
1998  .
1998  .
1999  .
2000  .
2001  .
2002  .
2003  .
2004  .
2005  .
2006  .
2007  .
2008  .
2009  .
2009  .
2010  .
2011  .
2012  .
2013  .
2014  .
2015  .
2016  .
2017  .
2018  .
2019  .
2019  .
2020  .
2021  .
2022  .
2023  .
2024  .
2025  .
2026  .
2027  .
2028  .
2029  .
2029  .
2030  .
2031  .
2032  .
2033  .
2034  .
2035  .
2036  .
2037  .
2038  .
2039  .
2039  .
2040  .
2041  .
2042  .
2043  .
2044  .
2045  .
2046  .
2047
```

Assignment 1

Description will be released soon in Ed

The assignment is Python based

Scaffold will be available in ed.

Due date: Week 4 Thursday 11:59 PM.



BREAK

Part 1 finished





python

```
from watson.framework import events
from watson.http.messages import Response, Request
from watson.common.imports import get_qualified_name
from watson.common.contextmanagers import suppress

ACCEPTABLE_RETURN_TYPES = (str, int, float, bool)

class Base(ContainerAware, metaclass=abc.ABCMeta):
    """The base class for all controllers.

    Attributes:
        __action__ (string): The last action that was called on the controller.
    """
    def execute(self, **kwargs):
        method = self.get_execute_method(**kwargs)
        self.__action__ = method
        return method(**kwargs) or {}

    @abc.abstractmethod
    def set_execute_method(self, **kwargs):
        raise NotImplementedError(
            'Implement get_execute_method()')
    
```

Python

Knowledge of Python is assumed for this course.

However, I know that there are a few students learning python in parallel with INFO1112.
Mostly doing INFO1110.

If you don't have knowledge of Python, you will find the practical work for INFO1112 hard.

If you want a fast start to Python, I strongly recommend the free course OLET1306.

In this second hour of the whole class session, I will spend some time some weeks working through Python features, examples and demonstrations how we write programs in Python.

Python

Python Developed in early 1990s by Guido van Rossum

Name comes from the British TV comedy “Monty Python’s Flying Circus”
(early 1970's)

Today a large number of people contribute to the development and
maintenance of Python

Python is *open source* meaning it can be downloaded and used for free and
modified in any way

Python

Clean and simple syntax, easy to read and write.

Encourages good programming habits

Has a LOT of power... but doesn't get in the way of learnability

Has a large standard library (“comes with batteries included”)

Python scripts are portable to anywhere the interpreter runs.

Python runs almost everywhere, even tiny machines such as arduino or the micro:bit

Python is available (for free) for: Windows, Apple, Unix/Linux

On a piece of paper write a Python function that takes an integer argument and prints a triangle of that many Xs?

eg given 3, print this:

```
X  
XX  
XXX
```



Demo Example code



Acknowledgement

This course has been designed by A/P Robert Kummerfeld to deliver in online mode. He recorded the video segments of this course, and we are going to re-use them this semester. Shout out to him and his great contribution to INFO1112.

Thank you to the TA and all Tutors of this course who are taking a big responsibility this semester.

T H A N K
Y O U

T H E
E N D

INFO1112 Weeks 1 Lab : Shell and Unix

1.1 Introduction

In the lab, we provide dual-boot machines, which can be used as Windows or Linux. Typically, you won't be doing the activities under Windows; instead you will need to reboot the machines into Linux.

Because of the setup involved, in providing the environment for you in the lab, the login can be very slow (especially the first time you login at a particular machine). So, as soon as you enter the lab for your weekly session, start your login. This uses your UniKey username and password (the same as you use for eLearning, Sydney Student and other facilities; for most students, the username consists of four letters and four digits).

You are welcome to use your own laptop and connect to the University's wireless network instead of a lab machine, as long as it is running a version of Unix (eg MacOS or the Linux subsystem for Windows 10). **Please note: in that case any admin support or troubleshooting is up to you.**

This (and many other labs) have a lot of information; they are intended to be kept as a reference as well as instructions during the session. Most labs end with a capability checklist, that points you to the crucial aspects that you really should practice and remember.

To test your knowledge we have included exercises in this document.

1.2 What is Unix?

Unix is an operating system, by which we broadly mean the suite of software that makes the computer work. Specifically, the operating system provides services for the programs they run. Unix (and its cousin Linux). The Unix operating system is made up of three parts: the kernel; the shell; and the programs.

1.2.1 The Kernel

The kernel of Unix is the hub of the operating system: it allocates time and memory to programs and handles the file store and communications in response to system calls (requests from programs for the kernel to do things on behalf of the program, e.g. print characters on the screen, write data to disk, etc).

1.2.2 The Shell

The shell acts as an interface between the user and the kernel. When a user logs in, the login program checks the username and password, and then starts another program called the shell. The shell is a

command line interpreter (CLI), just like the Python interpreter. It interprets the commands the user types in and arranges for them to be carried out. The commands themselves are usually programs: when they terminate, the shell gives the user another prompt to indicate it is ready to accept more commands.

There are many different shells, one of the earliest (simply called "the shell" or /bin/sh) is available on all Unix distributions. We will be showing you "bash" (Bourne Again SHell) (/bin/bash). Bash is an sh-compatible shell that incorporates useful features from other more modern shells.

1.3 Running examples

The form of the examples is as follows:

\$ COMMAND

To execute the command above be sure to only type COMMAND, then enter. Don't type the \$ character, this is only a prompt to indicate you can type a command..

1.4 Files and Processes

Everything in Unix is either a file or a process.

A process is an executing program identified by a unique *process identifier* or PID.

A file is a collection of data. Files can be created by users using text editors, running programs, or copying existing files from elsewhere on the system, or across the network etc. Notice that the idea of a file does not talk specifically about being on disk, it could be in memory, on disk, being transmitted across the network or being generated on the fly by another program or the operating system.

For an example of this last case, try running the following commands on a Linux machine:

```
$ cat /proc/uptime  
37931.26 37812.78  
$ cat /proc/uptime  
37932.72 37814.24
```

Notice, every time you run the program the numbers are changing. To give you an idea of what the file contains, try running the uptime command, which tells you how long the machine has been running (or up) for (in this case about 10.5 hours)

```
$ uptime  
00:27:44 up 10:33, 2 users, load average: 0.00, 0.00, 0.00
```

1.5 Working with Files and Directories

Just like in the Windows environment, files are stored in a hierarchical directory structure. The location of a particular file (or subdirectory) is described using a path. Under Windows, the hierarchy starts from a drive letter (for example C:\) but in Unix the hierarchy starts from a single slash (/) called the root directory. An absolute path is one that starts from the top of the hierarchy (i.e. begins with a slash). Alternatively, a relative path starts from the current directory.

An absolute path points to the same place regardless of what the current directory is. A relative path is just that, relative to your current location.

Don't directories break the everything is a file or process rule? No: directories are in fact special files which link filenames and other information about a file, with the file data itself.

1.5.1 pwd

Each process (a running program) has a working directory (also called the current working directory) from which all relative paths are interpreted.

The `pwd` command prints the current path and thus enables you to work out where you are in the filesystem. For example, if you run `pwd` when you first log in, you will find you are in your home directory:

```
$ pwd  
/home/bgat5227  
$
```

On linux, user accounts are usually given home directories of the form: /home/<login> where <login> is your UniKey login name.

1.5.2 ls

When you first login, your current working directory is your home directory. Your home directory has the same name as your username, for example, `bgat5227` for the lecturer's account, and it is where your personal files and subdirectories are saved.

To find out what is in your home directory, type:

```
$ ls  
Downloads Favorites MyMusic MyPictures MyShapes MyVideos  
$RECYCLE.BIN  
$
```

The `ls` command lists the contents of your current working directory.

If the home directory is empty no files will be shown (and you will just get a prompt back on the next line), but yours won't be. There will be some directories inserted by the System Administrator when your account was created.

The University environment is set up so that your home directory is also available (one says that the directory is mounted) on the School's lab machines, both under Windows and under Linux. If you are working on a School lab machine under Windows, you can find this as U:\

`ls` does not, in fact, cause all the files in your home directory to be listed, but only those ones whose name does not begin with a dot (.) Files beginning with a dot are known as hidden files and usually contain important program configuration information. They are hidden because you should not change them unless you are very familiar with Unix!

To list all files in your home directory including those whose names begin with a dot, type:

```
$ ls -a  
... .bash_history Downloads Favorites MyMusic MyPictures MyShapes  
MyVideos $RECYCLE.BIN
```

`ls` is an example of a command which can take options: `-a` is an example of an option. The options change the behaviour of the command. There are online manual pages that tell you which options a particular command can take, and how each option modifies the behaviour of the command. (See later in these lab notes)

1.5.3 mkdir

We will now make a subdirectory in your home directory to hold the files you will be creating and using in this lab. To make a subdirectory called `unixstuff` in your current working directory type:

```
$ mkdir unixstuff
```

To see the directory you have just created, type:

```
$ ls  
Downloads Favorites MyMusic MyPictures MyShapes MyVideos  
$RECYCLE.BIN unixstuff
```

1.5.4 cd

The change directory command `cd <directory>`, changes the current working directory to `<directory>`. The current working directory may be thought of as the directory you are "in", i.e. your

current position in the file-system tree.

To change to the directory you have just made, type:

```
$ cd unixstuff
```

Type `ls` to see the contents (which should be empty) and `pwd` to see the absolute path.

```
$ ls  
$ pwd  
/home/bgat5227/unixstuff
```

1.5.5 The Special directories . and ..

Using `ls -a` will always display the two directories `.` and `..`. They represent the current directory and its parent directory respectively. Clearly, these special directories are relative paths because they depend (by definition) on the current working directory.

These can be used with `cd` to navigate the directories:

```
$ pwd  
/home/bgat5227/unixstuff  
$ cd .  
$ pwd  
  
/home/bgat5227/unixstuff  
$ cd ..  
$ pwd  
/home/bgat5227
```

As you can see, `cd ..` takes you up to the parent directory.

When we type `cd .` we do not move as we are asking to move to the directory we are already in. It might not seem very useful yet, but there are lots of places later on where we will need to refer to the current directory.

1.5.6 More about directories and pathnames

```
$ pwd  
/home/bgat5227  
$ mkdir foo/bar  
mkdir: cannot create directory 'foo/bar': No such file or directory  
$ mkdir unixstuff/backups  
$ ls unixstuff  
backups  
$ mkdir -p foo/bar  
$ ls foo  
bar
```

We cannot create nested directories (directories within other directories) without first creating the necessary parent directories. The `-p` option can be used to force the building of parent directories first. Now we have two directories within our home directory (`unixstuff` and `foo`). The `unixstuff` directory has a `backups` subdirectory, and `foo` has a `bar` subdirectory.

```
$ ls backups  
ls: backups: no such file or directory  
$
```

We can't see inside the `backups` directory directly from our home directory, because it is only accessible from inside the `unixstuff` directory. However, we explicitly put parent directories using slashes (a relative path) like:

```
$ cd unixstuff/backups  
$ pwd  
/home/bgat5227/unixstuff/backups  
$ ls ..  
backups  
$ ls ../../..  
Downloads Favorites foo MyMusic MyPictures MyShapes MyVideos  
$RECYCLE.BIN unixstuff  
$ ls ../../../foo
```

bar

Notice we can also use the `..` directory with the slashes to get access to the grandparent directory, and then other directories from there.

`~` (the tilde) is another special directory that represents your home directory.

1.6 Copy, Moving and Deleting Files and Directories

1.6.1 cp

This is used to copy a file or directory to a new location:

```
$ cd ~  
$ touch afile  
$ ls  
afile Downloads Favorites foo MyMusic MyPictures MyShapes MyVideos  
$RECYCLE.BIN unixstuff  
$ cp afile unixstuff  
$ ls unixstuff  
afile backups  
$
```

`touch` is used to create an empty file (containing no characters), but you could create a file like this just as easily with a text editor (e.g. try using nano: `nano afile`, it's good for quick and easy text editing, but not for longer more complex editing sessions).

Here we have copied the file `afile` into the `unixstuff` directory. So now a copy of `afile` exists in your home directory and also in the `unixstuff` directory.

The file can also be copied to different filenames. Here we make a copy of `afile` called `anotherfile`:

```
$ cp afile anotherfile  
$ ls  
afile Downloads foo MyPictures MyVideos unixstuff  
anotherfile Favorites MyMusic MyShapes $RECYCLE.BIN
```

As well as specifying a named directory, you can also copy files to the current directory using dot:

```
$ cd unixstuff  
$ cp ../anotherfile .  
$ ls  
afile anotherfile backups
```

If we want to copy directories and all of their subdirectories recursively, the **-r** flag should be used:

```
$ cd  
$ cp -r unixstuff moreunixstuff  
$ ls moreunixstuff  
afile anotherfile backups  
$
```

Notice that the **cd** command without a directory argument returns you to your home directory (just like **cd ~**). We have then created a complete copy of **unixstuff** and its subdirectory **backups**.

1.6.2 mv

This is used to move (or rename) files and directories. Note, unlike MSDOS and the Windows command shell (**cmd.exe**) there is no separate rename command:

```
$ ls  
afile Downloads foo MyMusic MyShapes $RECYCLE.BIN  
anotherfile Favorites moreunixstuff MyPictures MyVideos unixstuff  
$ mv afile foo  
$ mv anotherfile yetanotherfile  
$ mv moreunixstuff lessunixstuff  
$ ls  
  
Downloads foo MyMusic MyShapes $RECYCLE.BIN yetanotherfile  
Favorites lessunixstuff MyPictures MyVideos unixstuff  
$ ls foo  
afile bar  
$
```

Here we have renamed the file **anotherfile** to **yetanotherfile** and **moreunixstuff** to **lessunixstuff**. We have also moved **afile** from our current directory into the **foo** directory.

While we are here, notice that the `ls` command tries to format the filenames in the most readable manner. This will change depending on the size of the shell window, and the length and number of filenames in the directory being listed.

1.6.3 rm

This is used to delete files.

```
$ ls
Downloads foo MyMusic MyShapes $RECYCLE.BIN yetanotherfile
Favorites lessunixstuff MyPictures MyVideos unixstuff
$ rm yetanotherfile
$ ls
Downloads foo MyMusic MyShapes $RECYCLE.BIN
Favorites lessunixstuff MyPictures MyVideos unixstuff
$
```

Here we have deleted the file `yetanotherfile`. **NB: unlike Windows there is no trash can under Unix, once a file is deleted it is gone forever**. The only way of getting back a file is to have a backup copy of it somewhere else on the system.

```
$ rm lessunixstuff
rm: lessunixstuff is a directory
$ rm -r lessunixstuff
$ ls
Downloads Favorites foo MyMusic MyPictures MyShapes MyVideos
$RECYCLE.BIN unixstuff
$
```

We cannot use this to delete directories unless we use the `-r` flag. Be careful using this as it will delete everything in that directory, i.e. all of the subdirectories and the files they contain.

1.6.4 rmdir

This is used to delete directories. It will only work when they are empty.

```
$ rmdir foo
rmdir: directory "foo": Directory not empty
$ rm foo/afile
```

```
$ rmdir foo/bar
$ ls foo
$ rmdir foo
$ ls
Downloads Favorites MyMusic MyPictures MyShapes MyVideos
$RECYCLE.BIN unixstuff
$
```

This series of commands deleted the file `afile` and the directory bar from within the `foo` directory, so we could then delete the `foo` directory using `rmdir`. You can see why using `rm -r` is tempting, but be careful!

1.7 Investigating Files

1.7.1 clear

This command will clear the screen and is sometimes useful when viewing files or listing lots of directories. It doesn't clear the entire history, only the current screenful. Commands prior to that will still be visible if you scroll the terminal window up.

1.7.2 cat

This will print the given file(s) to the standard output (screen).

```
$ cat /usr/share/dict/words
1080
10-point
...
zzt
zzz
$
```

You should have seen a lot of words fly by on the screen with this command. That's because `/usr/share/dict/words` contains a list of standard English words.

Note that `/usr/share/dict/words` is sometimes found at `/usr/dict/words`, particularly on older Linux installations, and they can contain very different sets of words. Or if you are using Ubuntu, you may need to install the `wamerican` or `wbritish` package to get access to the words file using `sudo apt-get install wamerican wbritish`.

If multiple files are given, they will be printed one after the other in order.

1.7.3 less

`less` is a more advanced text viewer. It allows the navigation a file or data read from standard input (normally the keyboard). It is a more advanced version of another text viewer, called `more` (which didn't have the capability of scrolling backwards through a file).

Here are some of the supported commands:

Arrow keys, Enter	up or down one line at a time
PageUp, PageDown	up or down one screen-worth at a time
b, SpaceBar	up or down one screen-worth at a time
g	beginning of the file
G	end of the file
/<pattern>	find a given pattern <pattern>
n	find the next matching instance of the pattern
q	quit out of less
h	get help on the available commands

`less` can be used as a direct substitute for `cat`:

```
$ less /usr/share/dict/words
1080
10-point
10th
... [to end of page]
$
```

But it can also be used in the following way:

```
$ cat /usr/share/dict/words | less  
1080  
10-point  
10th  
... [to end of page]  
$
```

Both these have the same effect. The second idiom can be used to view the output of any program that produces multiple screenfuls of output. Here the vertical bar (|) is a pipe, indicating that the output from the first program should not be output to the screen, but rather should be used as input to the second program (instead of the keyboard). This means `cat` is providing input to `less` rather than `less` directly reading it from the file. We will see why this is useful shortly.

Answer the following questions.

1. What is a relative file path? How is it different to an absolute file path?
2. What is an advantage of using an absolute file path with the `cd` command?
3. What defines whether or not a file is hidden?

1.7.4 grep

`grep` prints lines of the input matching a given expression:

```
$ grep 'uu' /usr/share/dict/words  
ahuula  
Anschwitz  
anschauung  
antiquum  
bestuur  
busuuti  
Carduus  
...
```

```

$ grep '^x' /usr/share/dict/words
x
xalostockite

xanth-
xanthaline
...
$ grep -i '^x' /usr/share/dict/words
x
x
x25
xA
xalostockite
...
$
```

grep is case sensitive by default, so we have to use the `-i` flag to set case insensitivity. Other useful flags are `-v` (line that do not match) and `-c` (produce a count only). If multiple files are given then the results are given by file.

More complex expressions can be used for the pattern, including wildcard characters. These include:

- `*` (0 or more of the preceding character),
- `.` (any character),
- `[abc]` (any 1 of the included characters),
- `[^abc]` (any 1 character, except those included),
- `^` (if put at the start of a pattern, specify that the string must start with the match)
- `$` (if put at the end of a pattern, specify that the string must end with the match)

When used (or if a space is used), the expression must be enclosed in single quotes.

These complex patterns are called regular expressions and we will spend quite a bit of time in this course learning how to use this powerful way of describing strings.

```

$ grep '^Z.us' /usr/share/dict/words
Zauschneria
Zeus
Ziusudra
$
```

This searches for any line with a string starting with ‘Z’ followed by any character, followed by a ‘u’ character, and finished with an ‘s’. We will come back to these in later labs.

1.7.5 wc

wc is used to count characters, words and lines in a file:

```
$ wc /usr/share/dict/words
479828 479828 4953680 /usr/share/dict/words
$ wc -l /usr/share/dict/words
479828 /usr/share/dict/words
$ wc -c /usr/share/dict/words
4953680 /usr/share/dict/words
$
```

Here we print the listing of lines, words and bytes, then the lines only and then the bytes (or characters) only for `/usr/share/dict/words`. Because this file only contains one word per line the number of each is the same for this file.

1.8 Redirection

Most processes initiated by Unix commands write to the standard output (or `stdout`) i.e. they write to the terminal window, and many take their input from the standard input (or `stdin`), i.e. they read it from the keyboard. There is also the standard error (or `stderr`), where processes write their error messages, which by default, is also to the terminal window.

1.8.1 More on cat

Here we see the other use of **cat**: taking input from the standard input (keyboard) and printing it to the standard output (the screen). You need to type in the bits in italic text and then press `<Ctrl-D>` at the end:

```
$ cat
foo
foo
bar
bar
<Ctrl-D>
$
```

Redirection can be used to change the standard input, output and error of a program to a file instead.

Running `cat` again, and redirecting standard output to a file looks like this:

```
$ cat > list1
```

apple

pear

<Ctrl-D>

```
$ cat list1
```

apple

pear

\$

Here we have redirected the standard output to the file `list1` using the `>` operator. This will create or overwrite the file specified after the redirection operator. We can use the `>>` operator to append the output. If the file does not exist it acts the same as the `>` operator.

```
$ cat >> list1
```

banana

orange

<Ctrl-D>

```
$ cat list1
```

apple

pear

banana

orange

```
$ cat >> list2
```

foo

bar

<Ctrl-D>

```
$ cat list2
```

foo

bar

```
$ cat > list2
```

beef

lamb

<Ctrl-D>

```
$ cat list2
```

```
beef
```

```
lamb
```

```
$
```

You can see in this example above that failing to use `>>` for the second lot of output for `list2` causes the original contents (foo and bar) to be deleted.

We can also use `cat` to join (or concatenate) files:

```
$ cat list1 list2 > biglist
```

```
$ cat biglist
```

```
apple
```

```
pear
```

```
banana
```

```
orange
```

```
beef
```

```
lamb
```

```
$
```

1.8.2 sort

`sort` can be used to sort the contents of a file:

```
$ sort biglist
```

```
apple
```

```
banana
```

```
beef
```

```
lamb
```

```
orange
```

```
pear
```

```
$ sort < biglist > sortedlist
```

```
$ cat sortedlist
```

```
apple
```

```
banana
```

```
beef
```

```
lamb
```

```
orange
```

```
pear
```

```
$
```

Sort by default takes input from the standard input and prints the sorted contents to standard output. However, if you specify a filename, here `biglist`, it will sort that file rather than standard input. Alternatively, we can use the `<` operator to redirect standard input to be the file `biglist`. Here we use both operators to take input from a file and send the output to a file. Note that `sort` can take a series of files as input and sort these.

1.8.3 pipes

Suppose we want to run some input through several programs, such as when counting the results of `grep` using `wc`. If we use redirection, it requires the creation of temporary files.

```
$ grep 'b' biglist > filteredlist
$ cat filteredlist
banana
beef
lamb
$ wc filteredlist
3 3 17 filteredlist
$
```

This requires us to then cleanup the file when complete. Instead we can use the `|` to create a pipe between the two programs.

```
$ grep 'b' biglist | wc
3 3 17
$
```

This pipe makes the output of the first program the input of the second program. These can then be chained together in arbitrarily long sequences.

```
$ grep 'b' biglist | wc | wc
1 3 24
$
```

1.9 Wildcards

Wildcards can be used on the command line to identify files.

* will match zero or more of any character. ? will find exactly one of any character. Square brackets can be used to create sets of characters to match from. For example, [abc] will match either the ‘a’, ‘b’ or ‘c’ in a filename.

```
$ touch alist
$ ls list*
list1
list2
$ ls *list
alist
biglist
filteredlist
$ ls *list*
alist
biglist
filteredlist
list1
list2
$ ls ?list
alist
$
```

1.10 Getting Help

1.10.1 man

This will produce the manual page for given program if it exists. Most standard Unix command have a man page. These will list a description of the command and a set of options, flags and other usage information. The manual will usually be displayed using `more` or `less`.

```
$ man ls
...man information...
$
```

If you ran the above code, you will notice that the entry is for `ls(1)`. This means `ls` is in section 1 of

the Unix manual pages. Typically, command line programs are in section 1, operating system function calls are in section 2, and standard library functions are in section 3.

Some entries occur in several sections (such as when the name is reused, say for a command line program and a C function), so you may not get the one you are looking for just by typing the name after man. In these cases, put the section number before the program name. Other section where the name occurs are often listed at the end of the manual entry.

```
$ man getopt  
User Commands getopt(1)
```

NAME

getopt - parse command options

...

SEE ALSO

intro(1), shell_builtins(1), sh(1), getopt(3C), attributes(5)
\$ man 3 getopt
...man information for C/C++ getopt...

\$

1.10.2 whatis

This produces one line descriptions of man entries containing the given keyword.

```
$ whatis getopt  
getopt getopt (1) - parse command options  
getopt getopt (3c) - get option letter from argument vector  
$
```

1.11 Filesystem Access

1.11.1 ls

Using the -l flag for ls, we can generate a long listing of a file or directory:

```
$ ls -l  
total 5120  
-rw-r--r-- 1 bgat5227 linuxusers 35 Mar 3 12:07 biglist  
drwxr-xr-x 3 bgat5227 linuxusers 0 Mar 3 09:16 Downloads
```

```
drwxr-xr-x 4 bgat5227 linuxusers 0 Mar 3 09:16 Favorites
-rw-r--r-- 1 bgat5227 linuxusers 17 Mar 3 12:08 filteredlist
-rw-r--r-- 1 bgat5227 linuxusers 25 Mar 3 12:06 list1
-rw-r--r-- 1 bgat5227 linuxusers 10 Mar 3 12:06 list2
drwxr-xr-x 3 bgat5227 linuxusers 0 Mar 3 09:16 My Music
drwxr-xr-x 3 bgat5227 linuxusers 0 Mar 3 09:16 My Pictures
drwxr-xr-x 2 bgat5227 linuxusers 0 Mar 3 09:16 My Shapes
drwxr-xr-x 3 bgat5227 linuxusers 0 Mar 3 09:16 My Videos
drwxr-xr-x 2 bgat5227 linuxusers 0 Mar 3 09:16 $RECYCLE.BIN
-rw-r--r-- 1 bgat5227 linuxusers 35 Mar 3 12:07 sortedlist
drwxr-xr-x 3 bgat5227 linuxusers 0 Mar 3 11:44 unixstuff
```

\$

Reading left to right, the properties listed are type and permissions, number of links, owner (username), owner (group), file size, date last modified and name. The type is the first character of the listing. These have the value - for files, d for directories and l for symbolic links. The next nine characters are the permissions. They are in three character blocks for user, group and world (or other) permissions. The each block is made up of three flags: r, w, and x. They have slightly different meanings for files and directories. If the permission is not granted a - (hyphen) appears.

1.11.2 Access Rights for Files

r indicates read permission (or otherwise), that is, the presence or absence of permission to read and copy the file;

w indicates write permission (or otherwise), that is, the permission (or otherwise) to change a file, including to delete it;

x indicates execution permission (or otherwise), that is, the permission to execute a file, where appropriate.

1.11.3 Access Rights for Directories

r allows users to list files in the directory;

w means that users may delete files from the directory or move files into it;

x means the right to access files in the directory. This implies that you may read files in the directory provided you have read permission on the individual files.

For the above, everyone can read all the files listed, but only user bgat5227 can write (or delete) them. Everyone can see into the directory unixstuff and access files within it (to which they have permission), but, again, only bgat5227 can modify it.

1.11.4 chmod

The owner of a file can change the permissions on a file using chmod. This allows them to hide or reveal files to other users on the computer. `chmod` takes options as follows:

u	Owner
g	Group
o	Other (world)
a	All
+	Add permission
-	Remove permission
=	Set permissions
r	Read
w	Write and delete
x	Execute and access directories

It is used as follows:

```
$ ls -l biglist
-rw-r--r-- 1 bgat5227 linuxusers 35 Mar 3 12:07 biglist
$ chmod g+rwx biglist
$ ls -l biglist
-rw-rwrxr-- 1 bgat5227 linuxusers 35 Mar 3 12:07 biglist
$ chmod go-rwx biglist
$ ls -l biglist
-rw----- 1 bgat5227 linuxusers 35 Mar 3 12:07 biglist
$ chmod a+r,a-w biglist
$ ls -l biglist
-r--r--r-- 1 bgat5227 linuxusers 35 Mar 3 12:07 biglist
$ cat > biglist
-bash: biglist: Permission denied
```

\$

Let us consider the following myscript.sh file.

```
-rwxr-xr-x 6 archie users 4096 Jul 5 17:37 myscript.sh
```

What will be the result of executing the following commands?

1. chmod g-wx myscript.sh
2. chmod a+w myscript.sh
3. chmod o=r myscript.sh
4. chmod o=rx myscript.sh
5. chmod o= myscript.sh

Answer the following questions.

1. What is the difference between a redirection and a pipe?
2. What happens when we redirect using ‘>’ to a file which already exists?
3. What happens when we redirect using ‘>>’ to a file which does not exist?

Consider the words file: /usr/share/dict/words. Use Unix commands to answer the following questions.

1. How many lines are the words file?
2. What word immediately comes before and after the word 'compute' in the words file?
3. Output the words file in reverse alphabetical order.
4. Find all words containing 'uu'.
5. Find all words which start with 'za'.
6. Find all words which end with 'space'.
7. Save all words which end with 'space' to a new file called 'space.txt'.
8. Find all words which do not contain the letter 'e'.
9. How many words do not contain the letter 'e'?
10. Find all words which do not have any vowels.
11. How many words do not have any vowels?
12. Find all the word which you can type with only the top row of the keyboard.
13. Find all words which start with and end with a vowel.

1.12 Processes

1.12.1 ps

Each process executed on the system is given a unique identifier -- a process ID, or PID for short. The

ps command can be used to list your (and others') running processes. To see this we can use the **sleep** command, which will waits (or sleeps) for the given number of seconds. To be able to see this working, we also use the background modifier & (the ampersand). Any process executed with this at the end of its command will be run in the background. This allows the user to continue to type commands into the shell while the program is running.

When a process is run in the background, the shell will tell you the jobspec (in brackets) and the PID of the process so it can later be brought to the foreground using the **fg** command (using the last backgrounded process or taking the jobspec as its argument).

```
$ sleep 10 &
[1] 20746
$ ps
PID TTY TIME CMD
18116 pts/4 00:00:00 bash
20746 pts/4 00:00:00 sleep
20747 pts/4 00:00:00 ps
$
```

Notice, there are three of your processes running (from this shell window). One of them is the shell itself (**bash**), another is the **ps** program itself and the third is the **sleep** command.

1.12.2 bg and fg

A currently running process can be stopped without destroying it by using pressing <**Ctrl-Z**>. This sends the process to sleep and prints the jobspec to screen. **bg** can then be used to start the process again in the background. It will then run as if it had been run from the command line using the & command. Using <**Ctrl-Z**> is a good trick when you forgot to put an ampersand on the end of your command line. **fg** can be used to run the process in the foreground.

Here is an example of these tricks in action:

```
$ sleep 30
<Ctrl-Z>
[1]+ Stopped sleep 30
$ bg
[1]+ sleep 30 &
$ fg 1
sleep 30
```

\$

If you actually want to kill a program on the other hand, press <Ctrl-C>. Many people confuse the two and run <Ctrl-Z> and so rather than killing their programs, they just put them to sleep, so they end up with lots of sleeping processes hanging around the system.

1.12.3 jobs

This is used to list the current sleeping and backgrounded jobs:

```
$ sleep 30 &
[1] 21119
$ sleep 30 &
[2] 21120
$ sleep 30
<Ctrl-Z>
[3]+ Stopped sleep 30
$ jobs
[1] Running sleep 30 &
[2]- Running sleep 30 &
[3]+ Stopped sleep 30
$
```

1.12.4 kill

This is used to send signals to programs to stop, halt or terminate them. It takes a PID or jobspec (preceded by a %). <Ctrl-C> is the equivalent to killing on a running process.

```
$ sleep 30
<Ctrl-C>
$ sleep 30 &
[1] 21158
$ jobs
[1]+ Running sleep 30 &
$ kill %1
$ jobs
[1]+ Terminated sleep 30
```

```
$ sleep 30 &
[1] 21165
$ kill 21165
$ jobs
[1]+ Terminated sleep 30
$
```

Sometimes a specific signal sent by `kill` must be specified. The most common of these is `-9`, which will cause a program to exit immediately. The normal kill signal gives the program the opportunity to cleanup after itself. This does not. `-9` is most often used when a program refuses to stop.

It is also possible to kill all of your processes, in all of your shell windows at once using `kill -9 -1`. This kills everything at once, including your shell processes, so it logs you out as well.

1.13 Other Useful Commands

1.13.1 `exit`

This ends the current interactive shell session and so causes you to be logged out. If you still have processes running, it will hang there until the processes have finished.

1.13.2 `passwd`

This allows you to change your password (note the missing ‘or’ in the command name). `passwd` will first ask for the current password, and then ask you twice for the new one (to make sure you don’t mistype it). If you get the original password wrong, it won’t change the password to the new one. It doesn’t do this check until after you have given it a new password:

```
$ passwd
Old password:
New password:
Re-enter new password:
passwd: Incorrect Password.
```

In this case, the password has not been changed. This catches people out all of the time.

Also, the password program can be rather choosy about passwords. Please make sure that your password isn’t easy to guess (far too many people use “password” or “password1” or their birthday)

1.13.3 `gzip` and `gunzip`

`gzip` and `gunzip` are the GNU zipping programs for compressing and decompressing files. Notice that they remove the original and compressed version of the files after they have run:

```
$ gzip biglist
$ ls biglist*
biglist.gz
$ gunzip biglist.gz
$ ls biglist*
biglist
$
```

1.13.4 tar

gzip is designed to compress a single file, so how do you compress and transfer multiple files around the system?

The `tar` archiving program collates many files and directories into a single file. The name comes from “tape archive” because these archives were often used to store lots of old files on a tape for the long-term, in case of a need to recover to an old state. The `c` flag specifies create an archive file, the `x` flag specifies extract an archive and the `t` flag specifies list the contents of an archive without extracting files. The `v` flag specifies verbose output and the `f` flag indicates that the following argument is the output file.

The `z` flag can be added to use `gzip` compression and decompression, which is the same as running `gzip/gunzip` after/before running the `tar` command.

```
$ tar cvf lists.tar list1 list2
list1
list2
$ tar xvf lists.tar
list1
list2
$ tar cvzf lists.tar.gz list1 list2
list1
list2
$ ls -l lists.tar*
-rw-r--r-- 1 bgat5227 linuxusers 10240 Mar 3 12:19 lists.tar
-rw-r--r-- 1 bgat5227 linuxusers 185 Mar 3 12:19 lists.tar.gz
$ tar tzvf
```

1.14 Tab completion

The tab key can be used to autocomplete shell strings on the command line. In its simplest form it allows for completion of filenames. Later versions of Bash also allow completion of arguments and restrict irrelevant file types. Try typing a single character and then pressing <tab>. If there is only one possible completion, the shell will automatically complete it for you. If there is more than one completion, you will hear a beep instead. Press <tab> again to see the list of possible completions.

1.15 Shell Variables

Variables are a way of passing information from the shell to programs when you run them. Programs look “in the environment” for particular variables and if they are found will use the values stored in them. Some are set by the system, others by you, yet others by the shell, or any program that loads another program.

Standard Unix variables are split into two categories: environment variables and shell variables. In broad terms, shell variables apply only to the current instance of the shell and are used to set short-term working conditions; environment variables have a farther reaching significance, and those set at login are valid for the duration of the session. By convention, environment variables have UPPER CASE and shell variables have lower case names.

1.15.1 echo

The `echo` command writes its arguments to standard output and can therefore be used to display the contents of variables. It will repeat to standard output whatever followed it on the command line. Environment and shell variables can be accessed by using the \$ symbol before their name.

```
$ echo $USER  
bgat5227  
$
```

1.15.2 Common Variables

OSTYPE	the operating system type
USER	your login name
HOME	the path name of your home directory
HOSTNAME	the name of the computer you are using
PATH	the directories the shell should search to find a command

1.15.3 printenv

`printenv` can be used to display all the currently set environment variables.

1.15.4 Resource Files

These are used by the shell at startup to set variables and run programs. These live in the user's home directory and can be modified to change the startup environment for a user. `.profile` and `.bash_profile` are used by Bash. `.cshrc` is used by C shell.

The commands in these files are a series of command line statements that are executed in order. These are an example of shell scripts. Most shells also support programming structures such as if statements and loops.

1.16 Acknowledgements

Thanks to James Gorman for compiling the first version of these lab notes. They are largely derived from the notes written by M Stonebank from the University of Surrey. Material was also taken from the University of Utah's Unix command summary at <http://www.math.utah.edu/computing/unix/unix-commands.html>. James Curran and Tara Murphy created the previous version of the notes used in INFO1903 until 2017. Bob Kummerfeld and Jonathan Du produced the 2018 version. The document was improved by Kelly Stewart and Tyson Thomas in 2019, and further improved by Bob Kummerfeld in 2020.

1.17 Capability checklist

When you've finished this lab, check that you know how to . . .

1. Move around the file system using `cd`, `ls`, `pwd`
2. Copy, move and delete files using `cp`, `mv`, `rm`, `rmdir`, `mkdir`
3. Investigate files using `cat`, `less`, `wc`, `grep`, `sort`
4. Use redirection and pipes to combine Unix commands
5. Use wildcards in Unix commands

INFO1112 Week 2 Tutorial

Congratulations on making it through the first week of semester!

We will be continuing our adventure of shell scripting by looking at a few more concepts such as:

- Shortcuts and productivity
- Permissions
- Redirection
- Sorting
- Regular Expressions
- Processes

1: Some shortcuts

To make your life a little easier while interacting with the terminal, these are some common shortcuts.

- CTRL + A, moves the cursor to the start of the command string
- CTRL + E, moves the cursor to the end of the command string
- CTRL + K, deletes from the cursor to the end of the command string
- CTRL + U, deletes from the cursor to the start of the command string
- CTRL + LEFT_ARROW , moves the cursor to the beginning of a word
- CTRL + RIGHT_ARROW , moves the cursor to the end of a word
- CTRL + W, cuts from the cursor to the start of a word
- CTRL + Y, pastes from the cursor to the end of a word
- UP_ARROW, retrieves previous command that has been executed
- DOWN_ARROW, retrieves next command that has been executed
- CTRL + R, search for a previously executed command, uses grep, you can continue to press the shortcut to get the next option with the criteria you have set.
- TAB, attempts to autocomplete the word, pressing it multiple times will provide options
- CTRL + L, clears the output from terminal

chmod

The owner of a file can change the permissions on a file using chmod. This allows them to hide or reveal files to other users on the computer. chmod takes options as follows:

Access rights for files

- r indicates read permission (or otherwise), that is, the presence or absence of permission to read and copy the file
- w indicates write permission (or otherwise), that is, the permission (or otherwise) to change a file, including to delete it
- x indicates execution permission (or otherwise), that is, the permission to execute a file, where appropriate.

Access rights for directories

- r allows users to list files in the directory
- w means that users may delete files from the directory or move files into it
- x means the right to access files in the directory. This implies that you may read files in the directory provided you have read permission on the individual files

```
$ ls -l biglist
-rw-r--r-- 1 bgat5227 linuxusers 35 Mar 3 12:07 biglist
```

In this example, everyone can read all the files listed, but only user bgat5227 can write (or delete) them.

```
$ chmod g+rwx biglist
$ ls -l biglist
-rw-rwxr-- 1 bgat5227 linuxusers 35 Mar 3 12:07 biglist
```

After changing permission with chmod, all linuxusers group members have read, write and execute permissions.

Here are some more examples of changing permissions on biglist; for removing read, wright and execute for group and other we can do:

```
$ chmod go-rwx biglist.      #removing rwx for group and other
$ ls -l biglist
-rw----- 1 bgat5227 linuxusers 35 Mar 3 12:07 biglist
```

To grant read permission to everyone and remove write permissions for everyone we can do:

```
$ chmod a+r,a-w biglist
$ ls -l biglist
-r--r--r-- 1 bgat5227 linuxusers 35 Mar 3 12:07 biglist
$ cat > biglist
-bash: biglist: Permission denied
```

At the end, write and execute permissions for everyone has been denied, even for bgat5227 user.

2: Permissions

From last week, you have discovered how to list information within a directory using the ls command.

As security conscious individuals, we want to prevent unwanted access to files we own while also specifying permissions to files we want to share with other users on our systems.

Create a file called `myscript.sh`

Now execute the following commands, each followed by

```
ls -l myscript.sh
```

- `chmod g-wx myscript.sh`
- `chmod a+w myscript.sh`
- `chmod o=r myscript.sh`
- `chmod o=rx myscript.sh`
- `chmod o= myscript.sh`
- `chmod o=w myscript.sh`

3. Redirection

Most processes initiated by Unix commands write to the standard output (or `stdout`) i.e. they write to the terminal window, and many take their input from the standard input (or `stdin`), i.e. they read it from the keyboard. There is also the standard error (or `stderr`), where processes write their error messages, which by default, is also to the terminal window.

Here we see the use of the `cat` command to take input from the standard input (keyboard) and printing it to the standard output (the screen). You need to type in the bits in italic text and then press **CTRL + D** at the end:

```
$ cat  
foo  
foo  
bar  
bar
```

```
<Ctrl-D>
$
```

Redirection can be used to change the standard input, output and error of a program to a file instead. Running cat again, and redirecting standard output to a file looks like this:

```
$ cat > list1
apple
pear
<Ctrl-D>
$ cat list1
apple
pear
$
```

Here we have redirected the standard output to the file list1 using the `>` operator. This will create or overwrite the file specified after the redirection operator. We can use the `»` operator to append the output. If the file does not exist it acts the same as the `>` operator.

```
$ cat >> list1
banana
orange
<Ctrl-D>
$ cat list1
apple
pear
banana
orange
$ cat >> list2
foo
bar
<Ctrl-D>
$ cat list2
foo
bar
$ cat > list2
beef
lamb
<Ctrl-D>
$ cat list2
beef
```

```
lamb
$
```

You can see in this example above that failing to use » for the second lot of output for list2 causes the original contents (foo and bar) to be deleted.

We can also use cat to join (or concatenate) files:

```
$ cat list1 list2 > biglist
$ cat biglist
apple
pear
banana
orange
beef
lamb
$
```

4: Cat questions

`cat` is a very versatile command that allows you to easily read from text files. We are able to concatenate files and produce a file with the merged result.

- Create a file that contains a schedule for the day
- Use the shopping list file from last tutorial and concatenate the contents with the newly created schedule
- Use the head and tail command on the shopping list, observe the output without specifying a flag
- After you have observed the output using the head and tail, use the flag -n with the value 2

5: More Redirection

We will utilise redirection in different ways within bash. We may want to redirect standard output or standard error, depending on the problem domain. Answer the following questions:

- What is the difference between a redirection and a pipe?
- What happens when we redirect using `>` to a file which already exists?
- What happens when we redirect using `»` to a file which does not exist?
- What happens when we redirect using `<` to a command
- What happens when we redirect using `2>` to a file?

Try the command:

```
cat /usr/share/dict/words
```

You should have seen a lot of words fly by on the screen with this command. That's because `/usr/share/dict/words` contains a list of standard English words.

Note that `/usr/share/dict/words` is sometimes found at `/usr/share/dict/words`, particularly on older Linux installations, and they can contain very different sets of words. Or if you are using Ubuntu, you may need to install the `wamerican` or `wbritish` package to get access to the words file using

```
sudo apt-get install wamerican wbritish
```

6: Regular Expressions

`grep` is a command that searches for text patterns using regular expressions.

```
grep pattern file1 file2
```

That command will print lines from `file1` and `file2` that match the pattern.

`grep` is case sensitive by default, so we have to use the `-i` flag to set case insensitivity. Other useful flags are `-v` (lines that do not match) and `-c` (produce a count only). If multiple files are given then the results are given by file.

More complex expressions can be used for the pattern, including wildcard characters.

These include:

- * (0 or more of the preceding character)
- . (any character),
- [abc] (any 1 of the included characters)
- [^abc] (any 1 character, except those included)
- ^ (if put at the start of a pattern, specify that the string must start with the match)
- \$ (if put at the end of a pattern, specify that the string must end with the match)

When used (or if a space is used), the expression must be enclosed in single quotes.

These complex patterns are called regular expressions and we will spend quite a bit of time in this course learning how to use this powerful way of describing strings.

```
$ grep '^Z.us' /usr/share/dict/words
Zauschneria
Zeus
Ziusudra
```

This searches for any line with a string starting with 'Z' followed by any character, followed by a 'u' character, and finished with an 's'. We will come back to these in later labs.

7: Sorting

Sort by default takes input from the standard input and prints the sorted contents to standard output. However, if you specify a filename, here biglist, it will sort that file rather than standard input. Alternatively, we can use the < operator to redirect standard input to be the file biglist. Here we use both operators to take input from a file and send the output to a file. Note that sort can take a series of files as input and sort these.

```
$ sort biglist
apple
banana
beef
lamb
orange
```

```
pear
$ sort < biglist > sortedlist
$ cat sortedlist
apple
banana
beef
lamb
orange
pear
$
```

8: Grep, Sort and Counting!

Consider the words file: /usr/share/dict/words. Use Unix commands to answer the following questions.

- How many lines are the words file?
- What word immediately comes before and after the word 'compute' in the words file?
- Output the words file in reverse alphabetical order.
- Find all words containing 'uu'.
- Find all words which start with 'za'.
- Find all words which end with 'space'.
- Save all words which end with 'space' to a new file called 'space.txt'.
- Find all words which do not contain the letter 'e'.
- How many words do not contain the letter 'e'?
- Find all words which do not have any vowels.
- How many words do not have any vowels
- Find all the word which you can type with only the top row of the keyboard
- Find all words which start with and end with a vowel.

Given the following file, use the command sort, and sort the values, under the name units.txt.

giga
kilo
mega
peta
tera

and powers.txt

3
6
9
12
15

Answer the following questions:

- Sort the units.txt file
- Use the paste command with units.txt and powers.txt and sort the list • Sort the combined list using the data from powers.txt
- Consider how we can sort the list using its numeric value

9. Processes

Each process executed on the system is given a unique identifier – a process ID, or PID for short. The ps command can be used to list your (and others') running processes. To see this we can use the sleep command, which waits (or sleeps) for the given number of seconds.

To be able to see this working, we also use the background modifier & (the ampersand). Any process executed with this at the end of its command will be run in the background. This allows the user to continue to type commands into the shell while the program is running.

When a process is run in the background, the shell will tell you the jobspec (in brackets) and the PID of the process so it can later be brought to the foreground using the fg command (using the last backgrounded process or taking the jobspec as its argument).

```
$ sleep 10 &
[1] 20746
$ ps
PID TTY TIME CMD
18116 pts/4 00:00:00 bash
20746 pts/4 00:00:00 sleep
20747 pts/4 00:00:00 ps
...
$
```

Notice, there are three of your processes running (from this shell window). One of them is the shell itself (bash), another is the ps program itself and the third is the sleep command.

bg and fg

A currently running process can be stopped without destroying it by using pressing <Ctrl-Z>. This sends the process to sleep and prints the jobspec to screen. `bg` can then be used to start the process again in the background. It will then run as if it had been run from the command line using the & command. Using <Ctrl-Z> is a good trick when you forgot to put an ampersand on the end of your command line. `fg` can be used to run the process in the foreground. Here is an example of these tricks in action:

```
$ sleep 30
<Ctrl-Z>
[1]+ Stopped sleep 30
$ bg
[1]+ sleep 30 &
$ fg 1
sleep 30
```

If you actually want to kill a program on the other hand, press <Ctrl-C> . Many people confuse the two and run <Ctrl-Z> and so rather than killing their programs, they just put them to sleep, so they end up with lots of sleeping processes hanging around the system.

Kill

This is used to send signals to programs to stop, halt or terminate them. It takes a PID or jobspec (preceded by a %). <Ctrl-C> is the equivalent to killing on a running process.

```
$ sleep 30
<Ctrl-C>
$ sleep 30 &
[1] 21158
$ jobs
[1]+ Running sleep 30 &
$ kill %1
$ jobs
```

```
[1]+ Terminated sleep 30
$ sleep 30 &
[1] 21165
$ kill 21165
$ jobs
[1]+ Terminated sleep 30
$
```

Sometimes a specific signal sent by kill must be specified. The most common of these is -9 , which will cause a program to exit immediately. The normal kill signal gives the program the opportunity to cleanup after itself. This does not. -9 is most often used when a program refuses to stop.

10: Wrangling with processes

It is also possible to kill all of your processes, in all of your shell windows at once using kill -9 -1 . This kills everything at once, including your shell processes, so it logs you out as well.

Try the following scenarios:

- Launch a program (such as cat and attempt to kill it from a separate terminal, use the kill command to send a signal to interrupt the process.
- Launch the program yes and stop the process using SIGSTOP shortcut on your keyboard.

INFO1112 Week 3 Tutorial

Processes and Files

When a program is running it is called a process and a Unix system will have over a hundred processes at any one time. Only a few will be directly started by users, the rest are system processes.

1: Process List Command

We can discover what processes are running at any one time using the `ps` command. From last week we dabbled with the process command but we will revisit the command. Each process executed on the system is given a unique identifier - a process identifier, or PID for short.

The `ps` command can be used to list your (and others) running processes. To see this we can use the `sleep` command, which will wait (or sleeps) for the given number of seconds.

- Use the `man` command to read about the `ps` command
- Use `ps` to list the users processes
 - what programs are shown?
 - what are their process ids?
 - what are the parent process ids?
 - explain why a process has a parent process?
- Use `ps` to list all processes running in the system
- Write a shell command pipeline to count the number of processes in the system

2: Parent and child processes

In a unix system, processes start other processes using the two system calls "fork" and "exec" (several variations). We can access these system calls from Python using the "os" module. This was demonstrated in the lecture. Python documentation is found at python.org - look at the Library documentation for "os" and "time".

- Write two python programs: first.py and second.py, where first.py uses the functions in the os module to start second.py running. Verify they are running with suitable printed messages.
- Modify second.py so that it uses the sleep function of the "time" module to sleep for 60 seconds before exiting. Start the first.py program asynchronously from the shell (use &) and then use ps to view the status of first.py and second.py. What is the status?

In Unix, when a process completes execution, it will return an integer result called an exit code which communicates to the shell whether the process has completed successfully, or has failed. The bash shell has an if statement that lets us test the exit code of a program. For example, the grep command will return 0 ("True") if one or more lines match the pattern and 1 ("False") if no lines match. As usual, see the manual entry for grep for more detail.

- Write a shell script that uses an if statement to test if second.py is sleeping (use ps and grep in a pipeline) and print the message "Child is sleeping" if it is, "Child is not sleeping" if not.

3: Zombie Processes

We can encounter a situation where a process finishes execution but has not been removed from the process table.

You can execute the following code and observe the process table once you have started executing.

```
import os
import time
pid = os.fork();
if pid > 0:
    time.sleep(40)
else:
    exit(0)
```

- What did you observe in the process table?
- What could we do to deal with this process?

4: Kill a process

You may encounter a situation when a pesky process wants to continue to execute even though there may be a logical error. Typically the SIGINT signal is used to simply interrupt the process and shut it down gracefully, however, processes can handle a slew of signals within the system with the exception of SIGKILL.

With the following program, launch it and send signals to the process using the kill command.

```
$ kill <signal number> <pid>
```

You can use `kill -l` to get a list of signals available on your system.

Unix-like operating systems provide an implementation of posix (posix is a unix standard) signals. These signals allow the kernel and other processes to communicate to a process. Within the context of the shell, they can be used for job control.

5: Process Control

We can actually control our processes using signals by using SIGSTOP and SIGCONT. These allow us to stop execution of a process and resume it at a later time.

Construct a shell script that will toggle the execution state of a process. If the process is currently running, this will send a signal to stop the process from executing (stopped state), if the process has been stopped, it will resume execution.

Use bash command line arguments (\$0, \$1, ...) to receive the process id from the user.

```
$ ./toggle_proc.sh 1112
pid 1112 has been stopped
$ ./toggle_proc.sh 1112
pid 1112 has been resumed
```

6: Inspecting the filesystem

A running Unix system will have several file systems mounted. These file systems are either real data stores with conventional files that contain data, or services where the file system driver is providing information or performing services in response to file system access.

We explored the files and directories in the /proc directory in lectures as an example of this behaviour. Example using the script:

```
ls /proc
```

We explored the files and directories in the /proc directory in lectures as an example of this behaviour.

- Use the mount command to find out how many file systems are mounted on your system.
- How many of those are connected to mass storage devices?
- What are the different types of file system?

INFO1112 Week 4 Tutorial

PIPE

In Linux, the pipe command lets you send the output of one command to another. Piping, as the term suggests, can redirect the standard output, input, or error of one process to another for further processing.

The syntax for pipe and unnamed pipe command is the | character between any two commands: Command-1 | Command-2 | ...| Command-N Here, the pipe cannot be accessed via another session; it is created temporarily to accommodate the execution of Command-1 and redirect the standard output. It is deleted after successful execution.

Now try this command and explain the results:

```
echo hello | wc
```

Now create a file called “sample.txt” and add some texts in a couple of lines to it. Then try this command and interpret the results:

```
cat sample.txt | grep -v a | sort -r
```

Memory, Scheduler, Boot Sequence

In week 3 we introduced the idea of a process, a running program, and investigated the ps command to gather detail about processes. First complete any left over activities.

All processes have a parent process, this is the process that did the fork system call to create the new process. Each parent has its parent and so on. The very first process in a running system has process ID of 1 and it is the init or launchd or systemd program (different versions of Linux have different initial programs).

We can view the ancestry of all the processes using the "pstree" command.

- Experiment with the pstree command to see what processes you have running.
- Trace the ancestry of your processes back to the initial process. This week the lecture covers memory management, the scheduler and the boot sequence of the Linux operating system.

Memory

- Read the manual entry for ps and work out how to determine the virtual memory size for a process
- What is the virtual memory size of your bash shell?
- Also from the ps manual: how do you determine the amount of physical memory currently in use by the process?
- What is the amount of physical memory being used by your bash shell?

Scheduler

- The ps command will also display the priority of processes. What is the column heading for the priority?
- What is the priority of your bash shell? Enter the following python program, nicer.py

```
import time
count = 0
while True:
    count = count+1
    time.sleep(1)
```

- run nicer.py from the shell asynchronously and use ps to find its priority
- increase the nice value (ie lower the priority) of nicer.py by 10 and look at the new priority with ps

Boot Sequence

The linux boot sequence described in lectures involves GRUB loading the kernel from a file system, usually /boot. There are usually several versions of the kernel in this location.

Find the location of the kernel file for your system and see how many versions are available.

Reboot your lab system and watch the sequence

INFO1112 Week 5 Tutorial

1: Emulators

A (not very practical) hypothetical computer CPU has instructions that are 16 bits and 256 bytes of memory.

- how many bits are required to represent a memory address?
- how many bytes are used for the memory address?
- how many bytes are used for an instruction?

The CPU has 4 registers.

- how many bits are needed to specify a register?

The register that will be involved in an operation is specified by the following bits of the operation code.

- define a mask that would extract the register number from an instruction
- define a mask and shift operation that would extract the operation code from an instruction.

Each instruction consumes 2 bytes and starts with an Instruction Code (6 bits), followed by a Register (2 bits) and lastly a Memory Address (8 bits). The instructions of the cpu and their binary codes are:

Instruction Name	Instruction Code (6)	Register (2)	Memory Address (8)
clear register	0	reg. address	
load register	1	reg. address	memory address
store register	2	reg. address	memory address
add to reg.	3	reg. address	memory address
subtract from reg.	4	reg. address	memory address
jump	5		memory address
jump if zero	6	reg. address	memory address
save PC in reg.	7	reg. address	

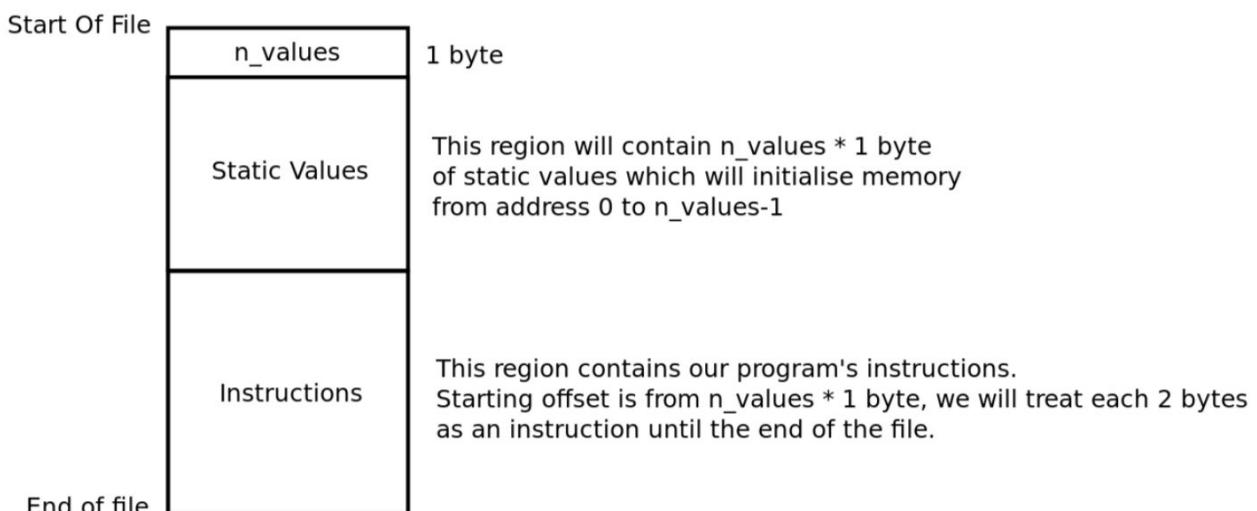
Take a copy of the file `emulate.py` from the Canvas page for this week and complete the code as marked. Start by assuming all instructions are fixed-width.

Extension: For a CLEAR instruction, there is no need to encode 8 bits for a memory address. How can we extend our emulator to handle variable-width instructions?

2: Programs for our computer

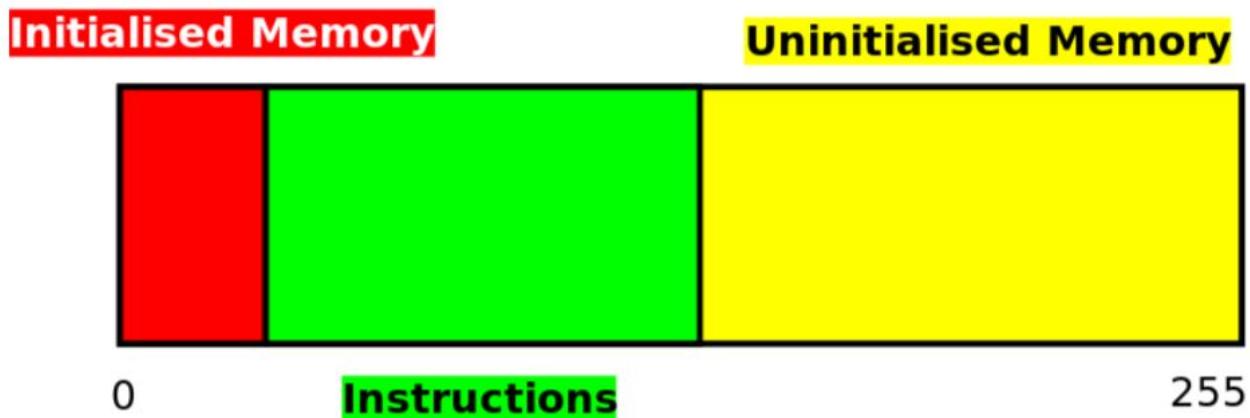
Our computer currently has no programs to run. When writing any kind of emulator (or software) we will need to ensure that our software is working. For this task, we will write a simple program loader that will load a programs. You can retrieve an example program from canvas.

The following layout will inform us of the program container.



The start of the file will specify how many values will be loaded into memory. After we have retrieved this value, we will load these values into main memory until we hit the instructions section. This section will also be loaded into main memory and the program will start executing from the first instruction in main memory.

After loading our program and initialising memory, our memory will have the following regions (size will vary based on the properties of the program).



To read all the data from a binary file, open the file in "rb" mode and read all data into a bytearray.

The following program reads all data from file into a byte array.

```
with open('prog.emu', 'rb') as f:
    data = bytearray(f.read())
```

3: Making programs

Your emulator should be able to load program files. However, we want to be able to write our own programs and use them within our application. Using python, write a python program that will produce a program for our emulator. Remember to use bytearray and binary literals to make it easy to encode instructions and write them to a file.

You can always inspect a binary file using the command line utility xxd (using the -b flag will output each byte in its binary representation).

INFO1112 Week 6 Tutorial

1: Downloading and starting your VM

You may get started by downloading and installing Alpine Linux. Visit <https://alpinelinux.org/downloads/> and select Virtual,x86_64image. The lab machines contain Virtual Box, when setting up the virtual machine, please use the following settings.

- Click new, create a linux virtual machine (Other 64-bit)
- 1024MB of RAM
- 2GB of Disk Space, use VDI

Afterwards you can launch your virtual machine and the image as a bootdisk.

- Keyboard layout and variant is us
- Keep your hostname as localhost
- use eth0 and dhcp retrieving your ip address
- Specify a password for yourself, must include letters and number
- Timezone is Australia/Sydney
- NTP client is chrony
- Use aarnet.edu.au mirror
- Use openssh for ssh service
- Use sda for disk and use sys installation

Alternatively, we have provided a prebuilt virtual machine. This virtual machine will be used to assist with simulating a small network. Import the virtual machine (tau) into virtual box.

2: Updating password for root

Before exposing our virtual machine to others, let's change the password to our root account. Use the passwd command while currently logged in to change the password of the user.

```
$ passwd
Changing password for root
New password:
Retype password:
passwd: password for root changed by root
Please make sure you remember this password, this linux distro requires the use of letters and numbers for user passwords.
```

3: Adding a user and group

You will be adding a couple of users to your VM. Right now, we will be adding a main user, which you will use for various tasks within this tutorial and for further coursework.

Use the `adduser` command to create and add a new user to your system.

```
$ adduser pi  
New password:  
Retype password:  
passwd: password for pi changed by root
```

Attempt to reboot and login with this new user.

4: Secure Shell

SSH is an application to log into a remote device. This allows us to execute any command we normally have access to through our own shell. We are limited to the tools the remote machine has access to however.

We have been utilising a local shell on our current machine, we going to attempt to connect to ucpu2 using our host machine.

```
ssh abcd1234@ucpu2.ug.it.usyd.edu.au
```

Our unikey maps to an existing user on ucpu2. This user has a home drive already set up and you are able to access your files that have been stored on your U drive.

The `ssh` command is available on most modern Unix systems. On Windows, you can use the PuTTY terminal emulator or WSL. You can access the undergraduate servers using `ssh` while connected to the university network.

- Connect to ucpu2 and check your current id on your system
- Inspect what groups you are associated with by using the `groups` command

5: Logging in!

After logging in to ucpr2, you should try and make your virtual machine accessible via SSH. However, let's first inspect the `known_hosts` file within the `.ssh` directory. From a quick inspection, we would see something similar to the following entry.

```
ucpr2.ug.it.usyd.edu.au,172.16.65.242 ecdsa-sha2-nistp256
AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAIBmlzdHAyNTYAAABBCMXGDNFLZMVyw
nrP4ncjnW/FdNQC5/d33C+1CjENzBaoz093ROhSb4keXXRTKo//NtvNJ10Eq43AhSG
o3Djpds=
```

Everytime we connect to a new server over `ssh`, the client will record the fingerprint of the device in `.ssh/known_hosts`.

Before you continue, you just need to perform an adjustment on your VM, you will need to forward the SSH port to the host machine. Execute the following command and substitute the `<VM_NAME>` with the name of your virtual machine.

```
VBoxManage modifyvm <VM_NAME> --natpf1 "guestssh,tcp,,3001,,22"
```

Let's get started with logging into the virtual machine. Make sure openssh has been started and will autostart when you log in. Use rc-update and specify sshd to be added to default run-level. You can retrieve the current ipaddress of the host machine by using ifconfig or ip addr show, specifically look out for the eth0 network controller.

We should be able to log into the device using our user credentials. By specifying the username, we will be prompted for the password which is mapped to the user.

6: SSH Authentication!

We will be adding an additional layer of authentication to our virtual machine. Using `ssh-keygen`, we will be creating a key-pair. It will generate a public key and a private key.

Generate a key on your host machine and specify to use the rsa encryption algorithm. You will be met with a few prompts asking for a passphrase to your key-pair and path.

```

Generating public/private rsa key pair.
Enter file in which to save the key (/home/pak/.ssh/id_rsa): Enter
passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in id_rsa.
Your public key has been saved in id_rsa.pub.
The key fingerprint is:
SHA256:8HbBJWk9wvaCKwpSeCUjvMiBOxyvbbx1z2YiXg1Y7Sk pak@rover The
key's randomart image is:
+---[RSA 3072]----+
|o ..o.          |
|o+o. ..*oo      |
|+++= .. =oo .    |
|=+o. ooo o...   |
|++ .ES+..       |
|...+ ..o=.      |
| ...o...oo.     |
|.....=         |
|....+          |
+---[SHA256]-----+

```

Once you have generated a key, you are free to distribute your public key to anyone but make sure you never reveal your private key. Afterwards, associate your public key with your virtual machine user.

- Check to see if /.ssh directory exists and authorized_keys file, if not, create both the directory and file
- Copy the public key to the intended user (user created at the beginning of the lab) by using either scp or ssh-copy-id
- When copying your public key, you can overwrite or append to the authorized_keys file (ssh-copy-id will do this for you)

Once you have copied this file over, you can then attempt to log in to your VM using your key-pair, it will add an additional layer of authentication. It will not prompt you for a password when attempting to login.

7: Sharing with friends

Similar to the previous exercise, ask a friend to provide you their public key and attempt to set up a user for them to access.

- Create a user for your friend on your virtual machine
- Ask for your friend's public key, this will be used to authenticate their usage on your system
- Check to see if they can log in to your virtual machine

You may want to use the command scp to copy files from the host to the virtual machine.

8: Hello init

Part of the boot process during the linux kernel, starts services/daemons for the operating system to boot. We will write a simple bash script that will output "Hello init!" during the boot sequence.

Our virtual machine init system uses OpenRC, you are able to place scripts within /etc/init.d. Ensure that your program has execute permissions.

```
#!/sbin/openrc-run
start() {
    echo "Hello Init!"
}
```

Make sure you set the correct permissions on the init script so it is able to be executed.

- Reboot your virtual machine and check to see if your script was executed "Hello Init!" should show up during the boot sequence
- If your script did not execute, check rc-status, it shows what scripts have started.
- In the event your script is not on the list, manually start your script using rc-service
- You can then ensure your script is autostarted by using
 - rc-update add <script_name> default

9: Launching a webserver

We have built a simple web server application in python. However, we will want this server to be launched during the boot process. Download the `server.py` file from canvas. You can simply launch this python script by using `python3` command within.

If the `python3` command does not exist, you can add the package using

```
apk add python3
```

- What do you observe when you run the `server.py` as the root user?
- What do you observe when you run the `server.py` as your newly created user?
- Attempt to connect to your webserver using the `curl` command, use the appropriate port

10: Running your VM in headless mode

If you are using VirtualBox you are able to select the VM to run in headless mode. This removes the GUI to your virtual machine and if you have set up your users and openssh, you are able to connect to your VM.