LOGIC CIRCUITS

CIRCUITS

Combinational circuit

- The input values explicitly determine the output

Sequential circuit

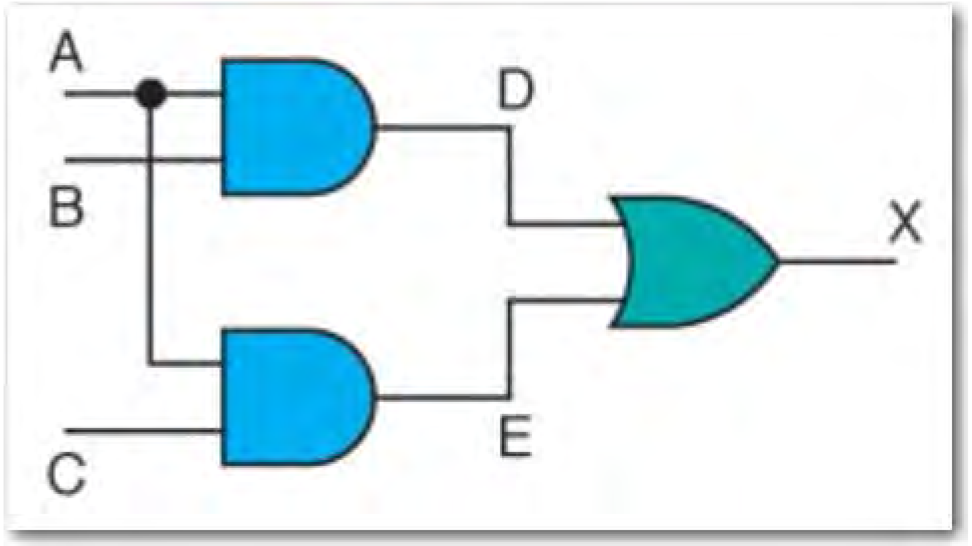
- The output is a function of the input values and the existing state of the circuit

Circuit operation described using

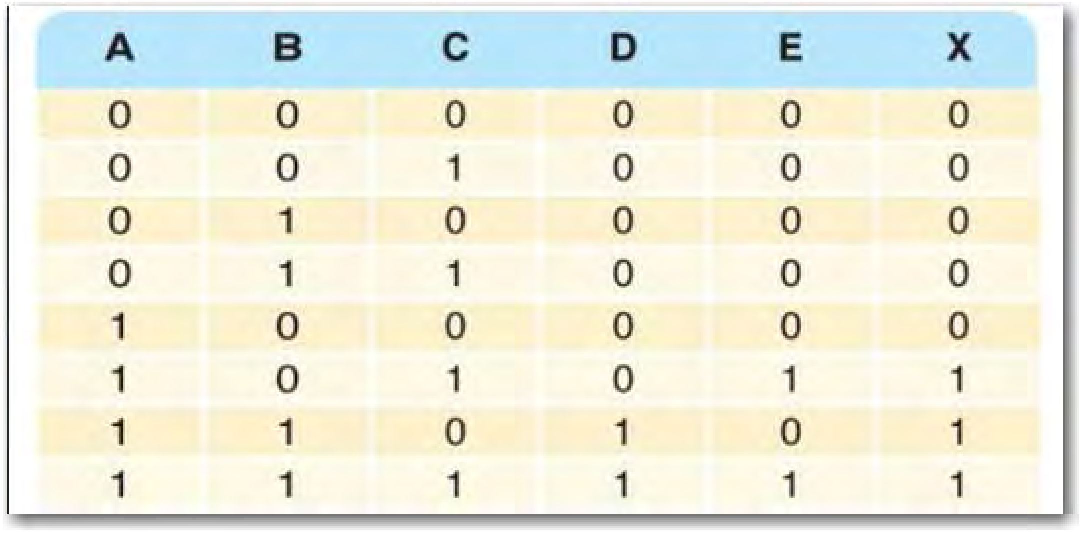
- Boolean expressions
- Logic diagrams
- Truth tables

COMBINATIONAL CIRCUITS

Gates are combined into circuits by using the output of one gate as the input for another



COMBINATIONAL CIRCUITS



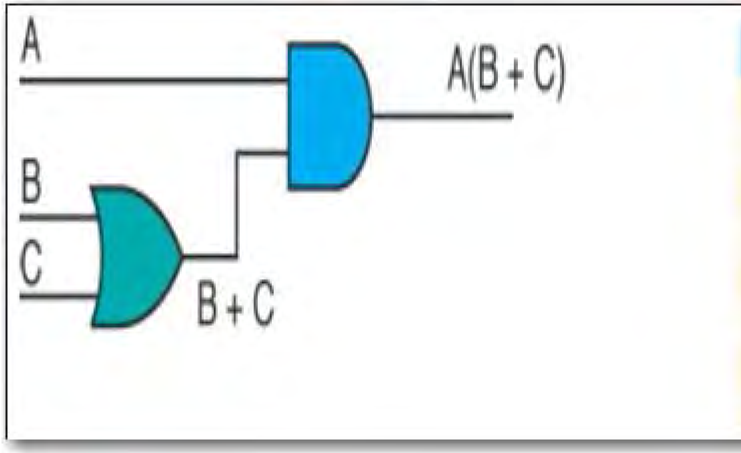
A	B	C	D	E	X
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	1	1

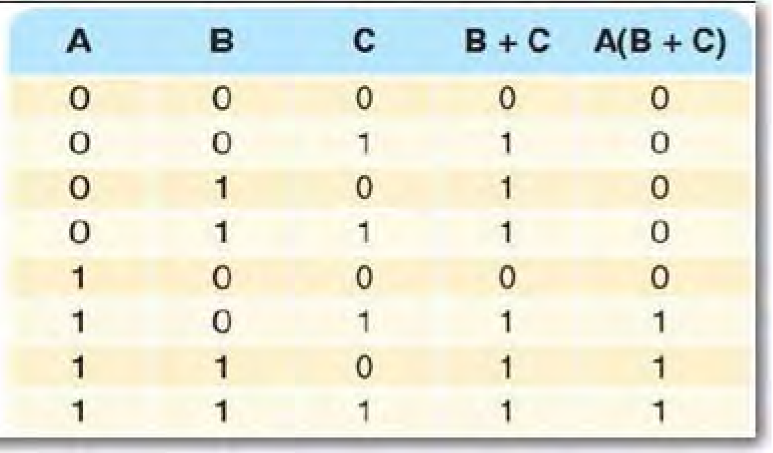
Three inputs require eight rows to describe all possible input combinations

This same circuit using a Boolean expression is (AB + AC)

COMBINATIONAL CIRCUITS

Consider the following Boolean expression A(B + C)





A	B	C	B + C	A(B + C)
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

Does this truth table look familiar?

Compare it with previous table

COMBINATIONAL CIRCUITS

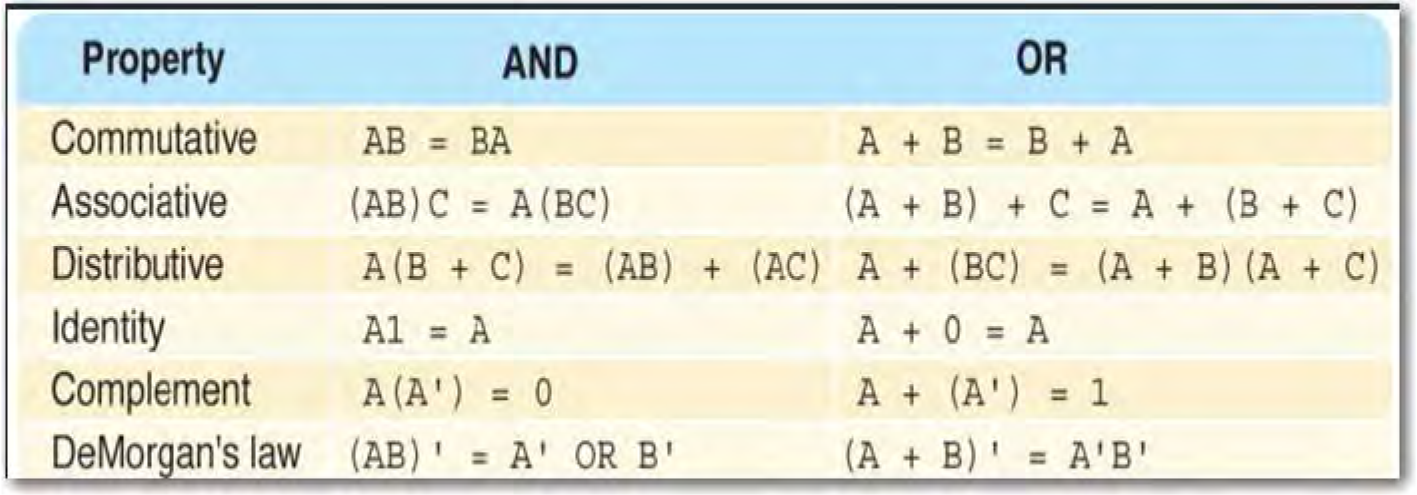
Circuit equivalence

Two circuits that produce the same output for identical input

Boolean algebra allows us to apply provable mathematical principles to help design circuits

$A(B + C) = AB + AC$ (distributive law) so circuits must be equivalent

PROPERTIES OF BOOLEAN ALGEBRA



Property	AND	OR
Commutative	AB = BA	A + B = B + A
Associative	(AB)C = A(BC)	(A + B) + C = A + (B + C)
Distributive	A(B + C) = (AB) + (AC)	A + (BC) = (A + B)(A + C)
Identity	A1 = A	A + 0 = A
Complement	A(A') = 0	A + (A') = 1
DeMorgan's law	(AB)' = A' OR B'	(A + B)' = A'B'

ADDERS

At the digital logic level, addition is performed in binary

Addition operations are carried out by special circuits called “adders”

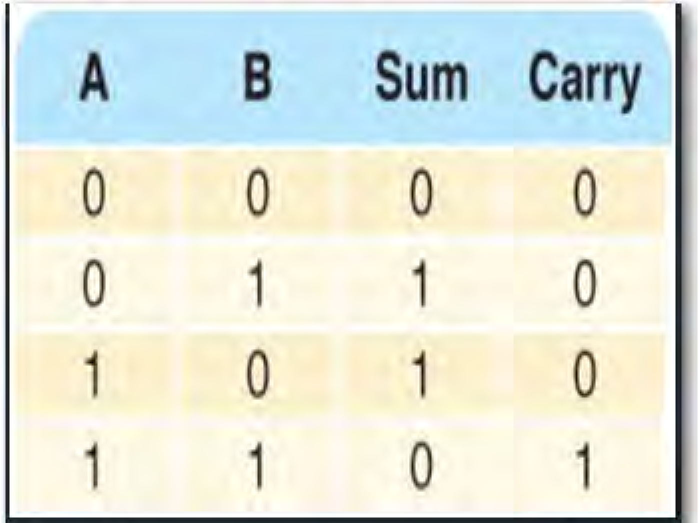
HALF ADDER

The result of adding two binary digits could produce a *carry value*

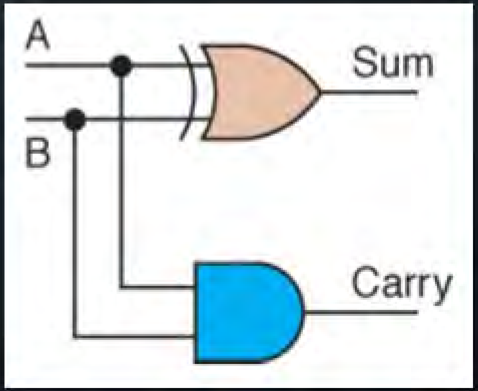
Recall that 1 + 1 = 10 in base two

- Half adderA circuit that computes the sum of two bits and produces the correct carry bit

Circuit diagram and boolean expressions:



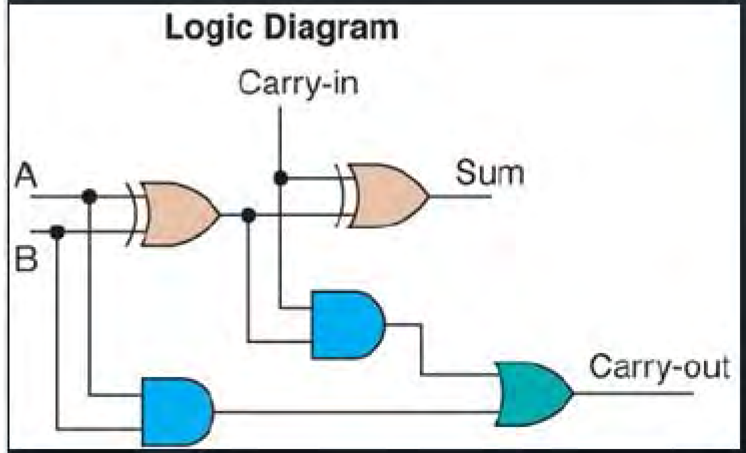
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

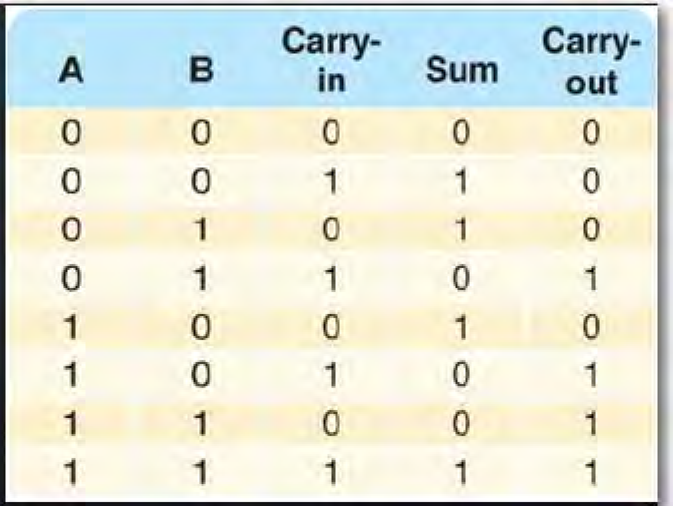


$$\text{sum} = A \oplus B$$
$$\text{carry} = AB$$

FULL ADDER

A circuit that takes the carry-in value into account





A	B	Carry-in	Sum	Carry-out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

CIRCUITS AS MEMORY

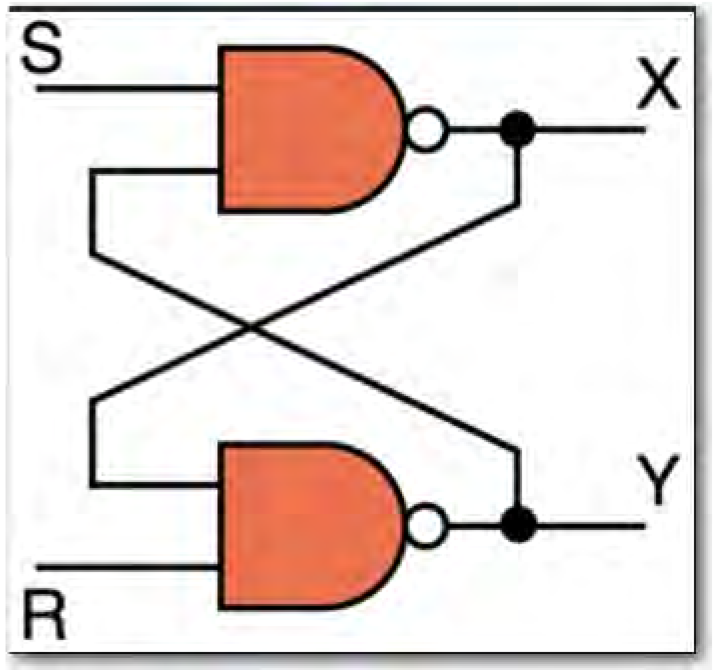
Digital circuits can be used to store information

These circuits form a sequential circuit, because the output of the circuit is also used as input to the circuit

S-R LATCH – MEMORY CIRCUIT

An S-R latch stores a single binary digit (1 or 0)

There are several ways an S-R latch circuit can be designed using various kinds of gates (generally NAND and NOR gates are used)

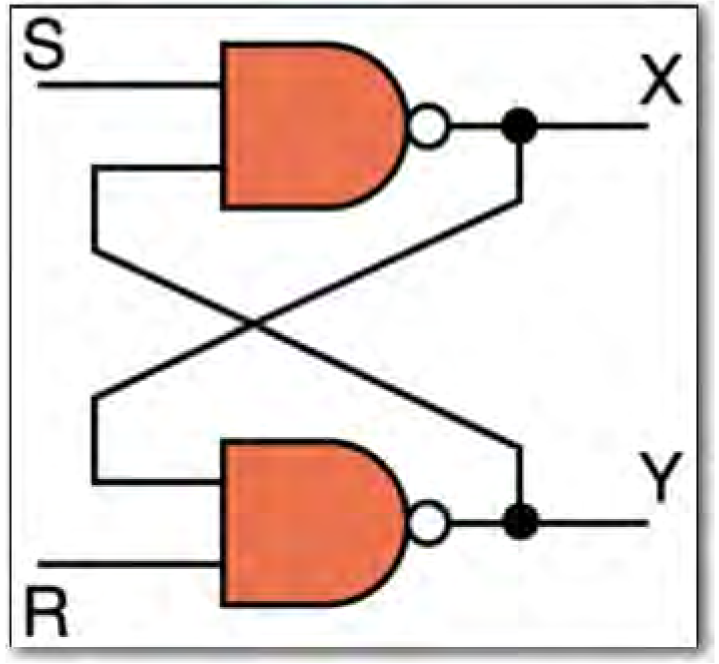


S-R LATCH – MEMORY CIRCUIT

The design of this circuit guarantees that the two outputs X and Y are always complements of each other

The value of X at any point in time is considered to be the current state of the circuit

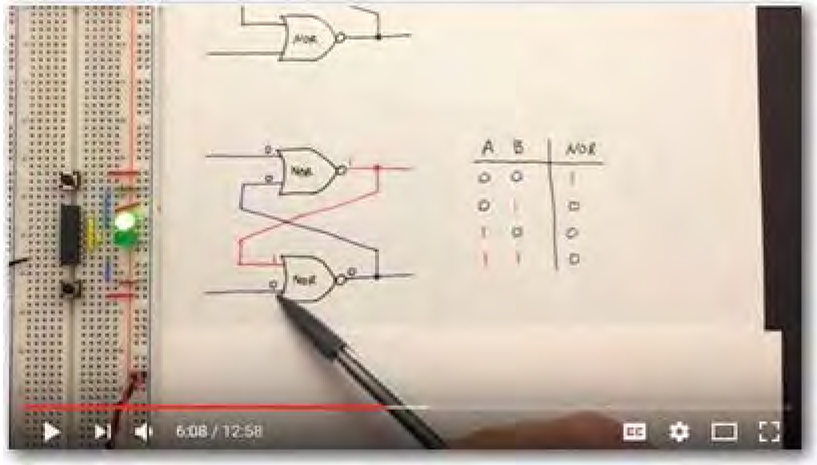
Therefore, if X is 1, the circuit is storing a 1; if X is 0, the circuit is storing a 0

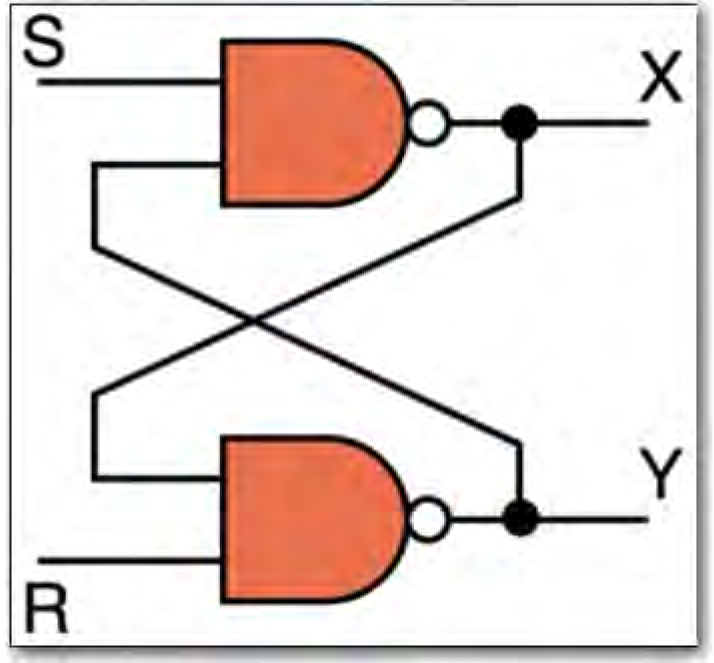


S-R LATCH – MEMORY CIRCUIT

Good video:

https://www.youtube.com/watch?v=KM0DdEaY5sY





INTEGRATED CIRCUITS

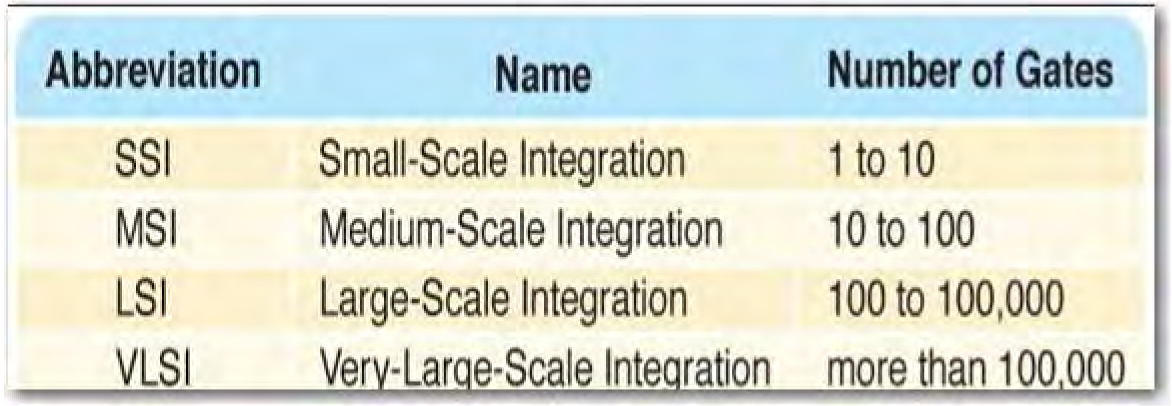
Integrated circuit (also called a *chip*)

A piece of silicon on which multiple gates (transistors) have been embedded

Silicon pieces are mounted on a plastic or ceramic package with pins along the edges that can be soldered onto circuit boards or inserted into appropriate sockets

INTEGRATED CIRCUITS

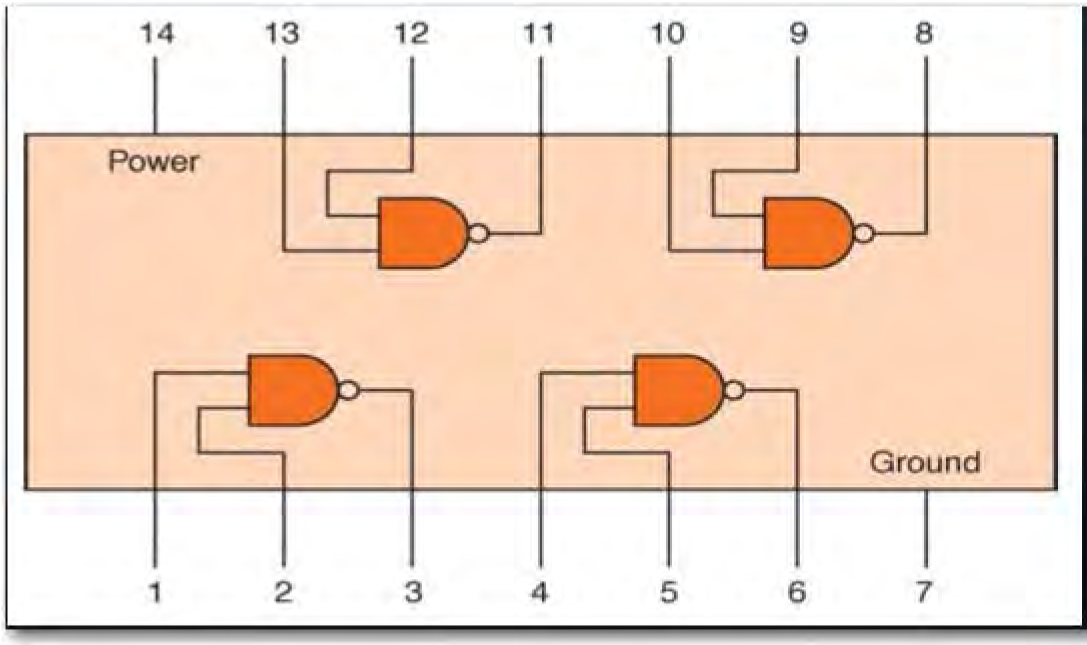
Integrated circuits (IC) are classified by the number of gates contained in them



Abbreviation	Name	Number of Gates
SSI	Small-Scale Integration	1 to 10
MSI	Medium-Scale Integration	10 to 100
LSI	Large-Scale Integration	100 to 100,000
VLSI	Very-Large-Scale Integration	more than 100,000

INTEGRATED CIRCUITS

E.g. An SSI chip contains independent NAND gates



GMIT

INSTITIUID TEICNEOLAIOCHTA NA GAILLIMHE-MAIGH EO

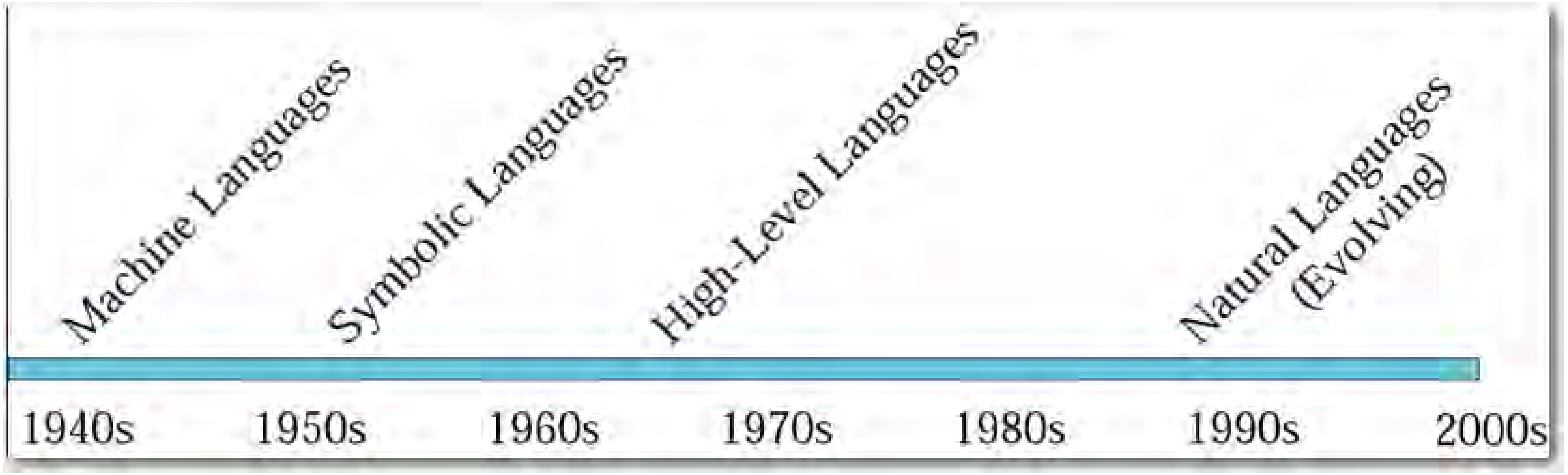
GALWAY- MAYO INSTITUTE OF TECHNOLOGY

COMPUTER LANGUAGE EVOLUTION

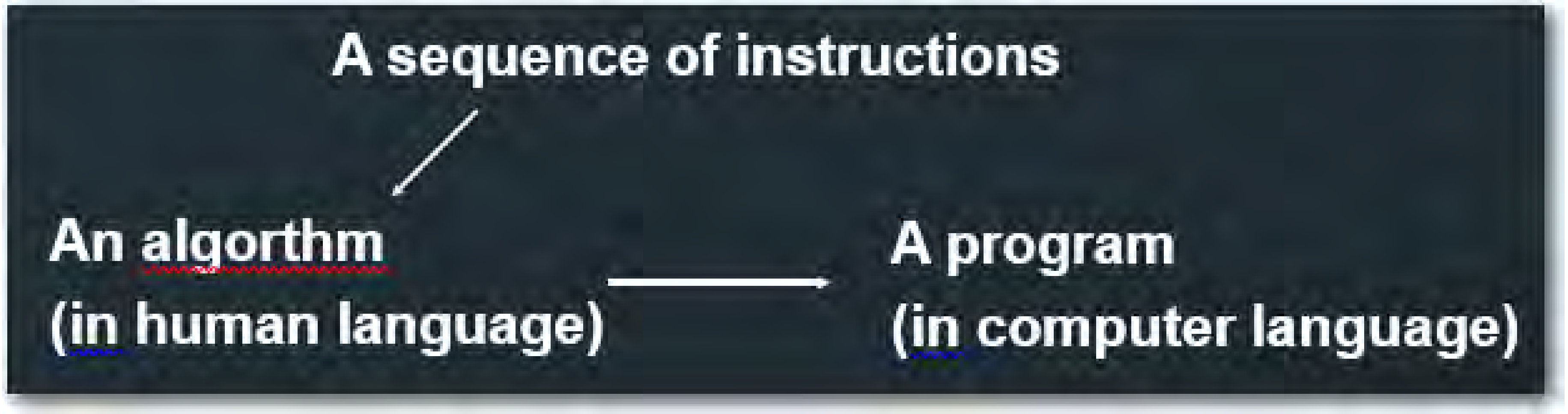
Lecture will cover

- Computer language evolution
- Distinguish between machine, assembly, and high-level languages
- Understand the process of program creation and execution

EVOLUTION OF COMPUTER LANGUAGES



SOFTWARE DEVELOPMENT
WHAT IS A (PROGRAMMING) LANGUAGE?



A computer program needs to be written in a language

There are many programming languages

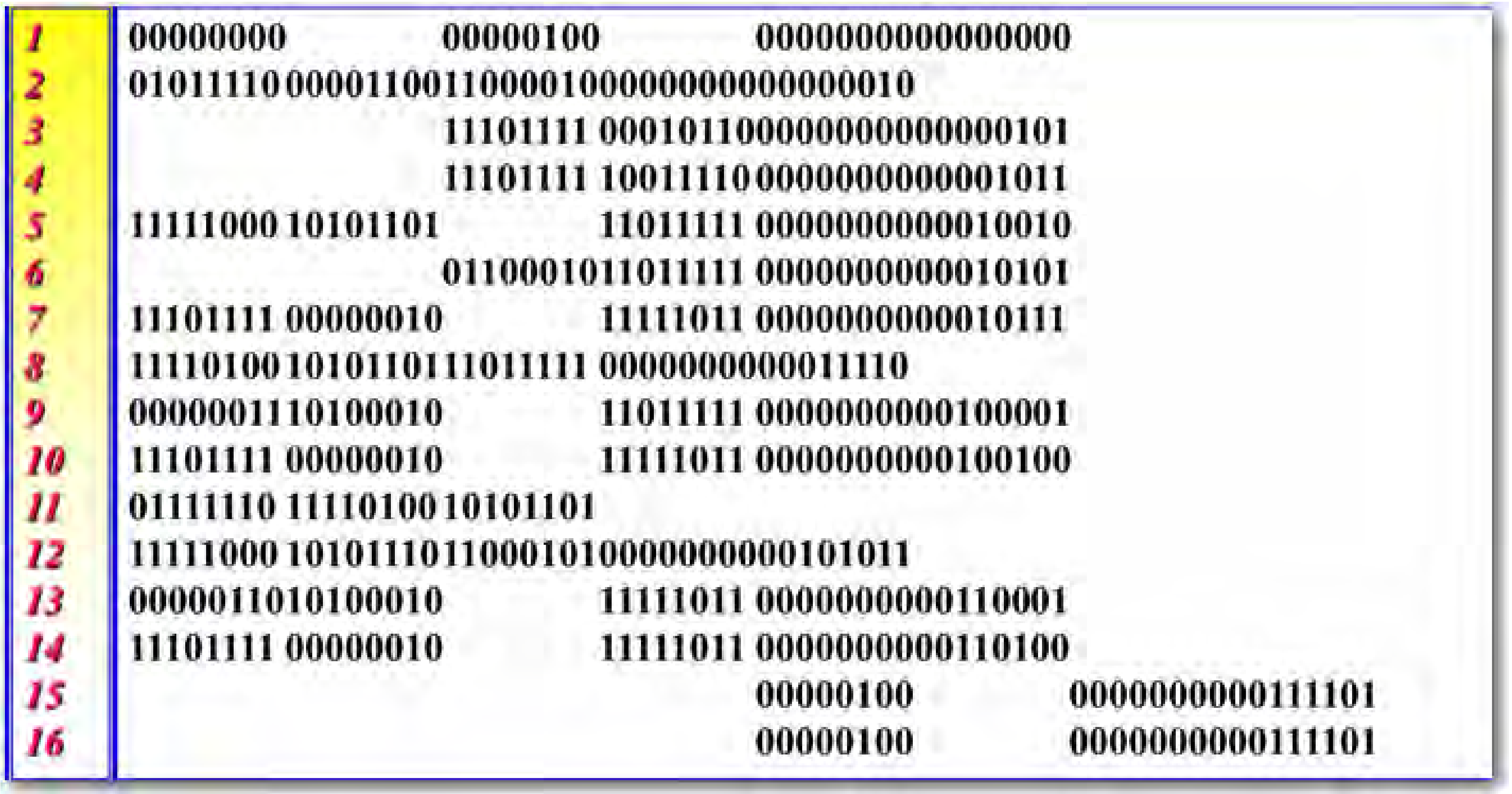
- Low-level, understandable by a computer (binary)
- High-level, need a translator! (C++/Java)

1 ST GENERATION LANGUAGE

Machine Language

- Binary Representation of the task
- Only programming language that the computer can understand directly without translation
- Each processor requires its own machine language - machine-dependent
- Tedious, difficult and time consuming method of programming
- Fast execution speeds and efficient use of primary memory

MACHINE LANGUAGE PROGRAM

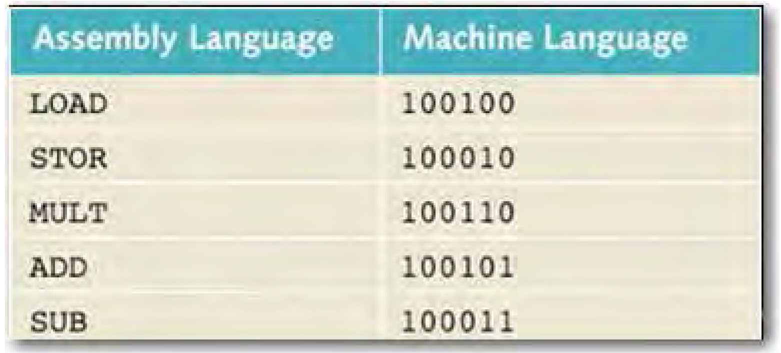


```
00000000                 00000100                 0000000000000000
01011110 00001100 11000010000000000000000010
                         11101111 0001011000000000000000101
                         11101111 10011110 0000000000001011
11111000 10101101                 11011111 0000000000010010
                         0110001011011111 0000000000010101
11101111 00000010                 11111011 0000000000010111
11110100 1010110111011111 0000000000011110
0000001110100010                  11011111 0000000000100001
11101111 00000010                 11111011 0000000000100100
01111110 11110100 10101101
11111000 1010111011000101 0000000000101011
0000011010100010                  11111011 0000000000110001
11101111 00000010                 11111011 0000000000110100
                                           00000100                 0000000000111101
                                           00000100                 0000000000111101
```

2ND GENERATION LANGUAGE

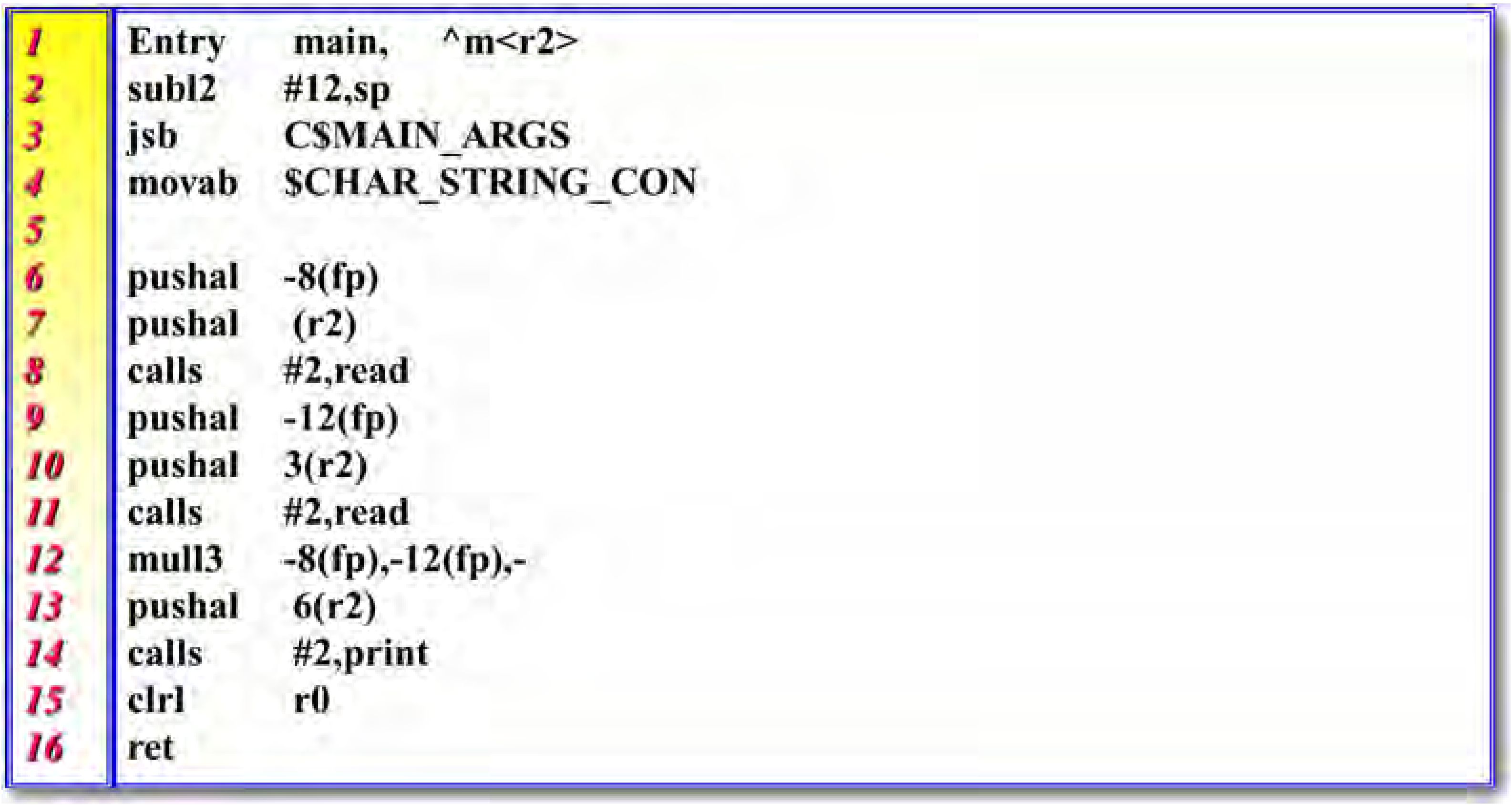
Assembly Language

- Symbolic Representation of the task - used mnemonic codes
- Codes translated into machine language by a program called the "assembler“
- Detailed knowledge of hardware still required.
- More efficient, use less storage, and execute much faster than programs designed using high-level languages



Assembly Language	Machine Language
LOAD	100100
STOR	100010
MULT	100110
ADD	100101
SUB	100011

ASSEMBLY PROGRAM



```
Entry    main,   ^m<r2>
subl2    #12,sp
jsb      C$MAIN_ARGS
movab    $CHAR_STRING_CON

pushal   -8(fp)
pushal    (r2)
calls    #2,read
pushal   -12(fp)
pushal   3(r2)
calls    #2,read
mull3    -8(fp),-12(fp),-
pushal    6(r2)
calls     #2,print
clrl      r0
ret
```

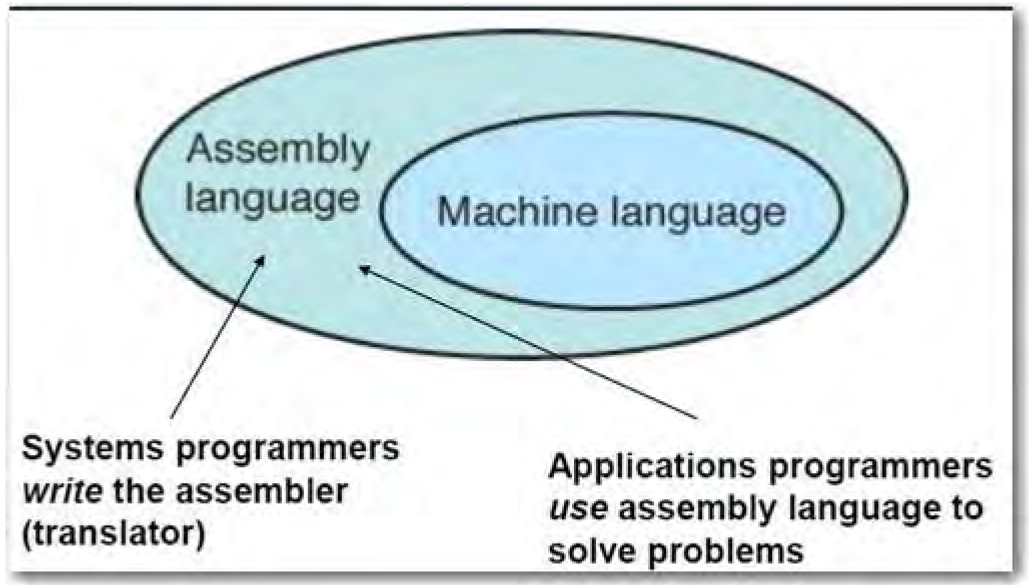
PROGRAM EXECUTION

- Von Neumann's Fetch, Decode, Execute cycle
  - 'Program counter' iterates through instructions one line at a time
    - Fetch the next instruction from memory
    - Decode the instruction and figure out what data is needed from memory
    - Execute the instruction and store the result if necessary
- Fetch Execute Decode viewing
  - How a CPU works: https://www.youtube.com/watch?v=jFDMZpkUWCw

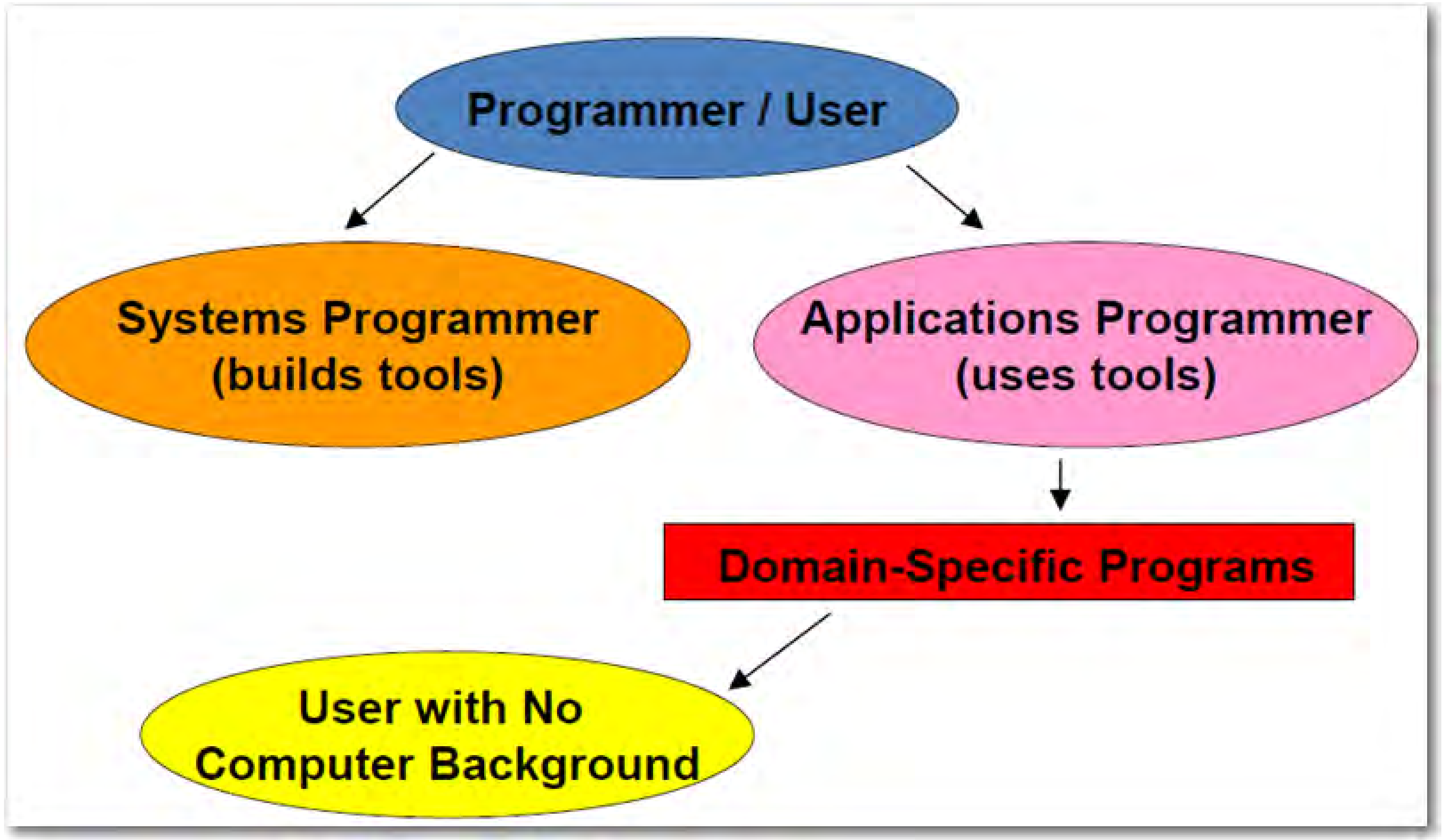
PROGRAMMER ROLES - CHANGES

When the Assembly – Machine language translation process developed, programmers became divided into 2 categories

Application programmers and systems programmers


Assembly language
Machine language
Systems programmers *write* the assembler (translator)
Applications programmers *use* assembly language to solve problems

COMPUTING AS A TOOL


Programmer / User
Systems Programmer (builds tools)
Applications Programmer (uses tools)
Domain-Specific Programs
User with No Computer Background

COMPUTING AS A DISCIPLINE

How to solve problems?

What can be (efficiently) automated?

Four Necessary Skills

- Algorithmic Thinking/Logic
- Representation
- Programming
- Design

What do you think?

- Is Computer Science a mathematical, scientific, or engineering discipline?

EXAMPLES OF SYSTEMS AREAS

Algorithms and Data Structures

Programming Languages

Architecture

Operating Systems

Software Engineering

Human-Computer Interaction

EXAMPLES OF APPLICATION AREAS

Numerical Computation

Databases and Information Retrieval

Graphics and Visual Computing

Net-Centric Computing

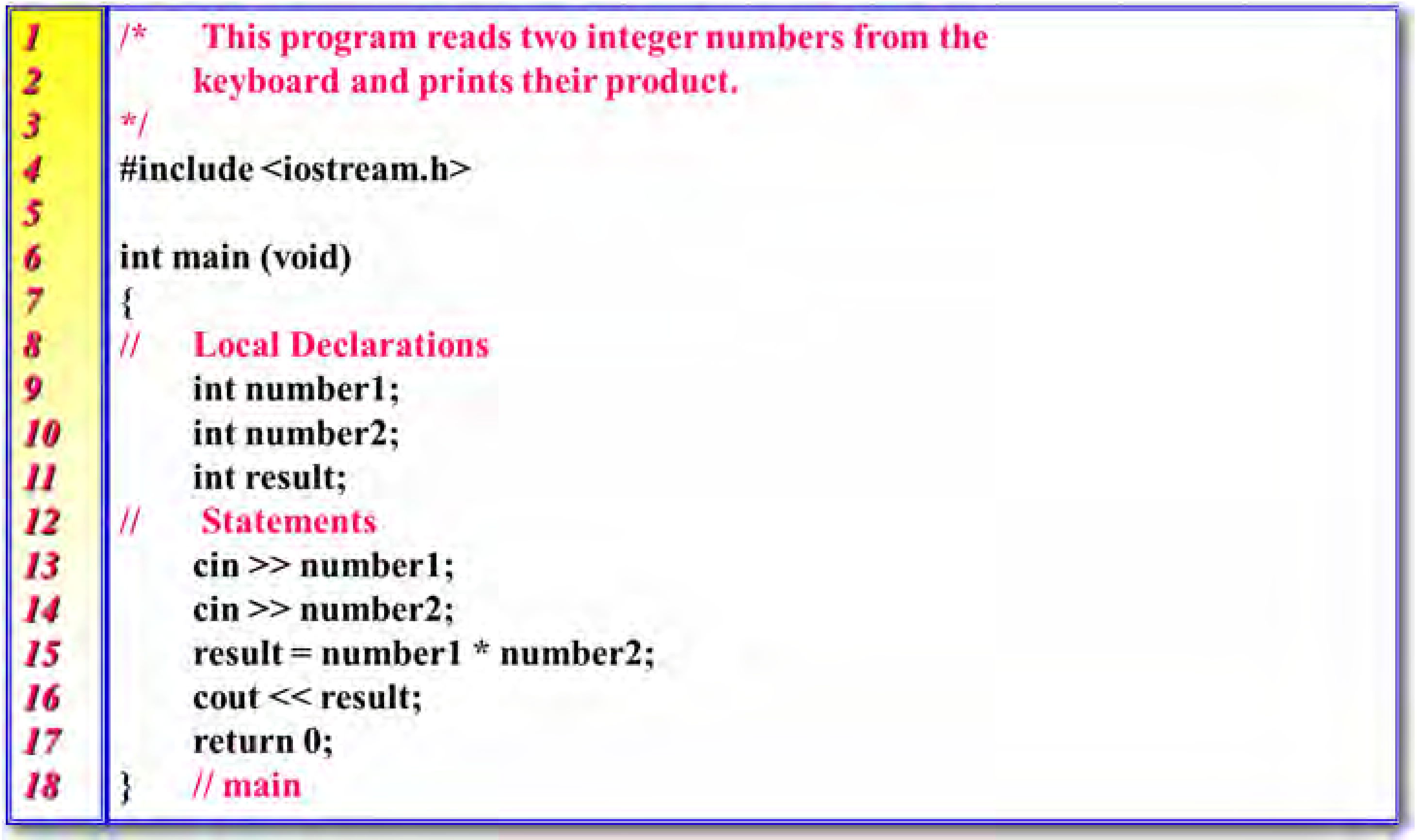
Computational Science

Artificial Intelligence and Machine Learning

3RD GENERATION LANGUAGE

- Third generation languages - also known as high-level languages
  - Much like everyday text and mathematical formulas in appearance.
  - Relieve the programmer of the detailed and tedious task of writing programs in machine language and assembly languages.
    - More time to focus on designing software to meet the user's needs.
  - Designed to run on a number of different computers with few or no changes.
  - A language translator is required to convert a high-level language program into machine language.
    - Two types of language translators are used with high level languages: compilers and interpreters.
  - Examples: C, C++, C#, Java, JavaScript

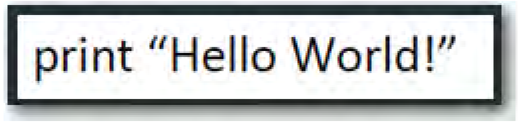
C++ PROGRAM



```
/*     This program reads two integer numbers from the
       keyboard and prints their product.
*/
#include <iostream.h>

int main (void)
{
//     Local Declarations
       int number1;
       int number2;
       int result;
//     Statements
       cin >> number1;
       cin >> number2;
       result = number1 * number2;
       cout << result;
       return 0;
}      // main
```

4TH GENERATION LANGUAGE



```
print "Hello World!"
```

Fourth generation languages - also known as very high level languages

- Allow programmers specify what the computer is supposed to do without having to specify how it's supposed to do it.
- Need approximately one tenth the number of statements that a high level languages needs to achieve the same results.
- Increases the speed of developing programs.
  - System programmers write translators for high-level languages
  - Application programmers use high-level languages to solve problems
- Examples: Python, Ruby, SQL

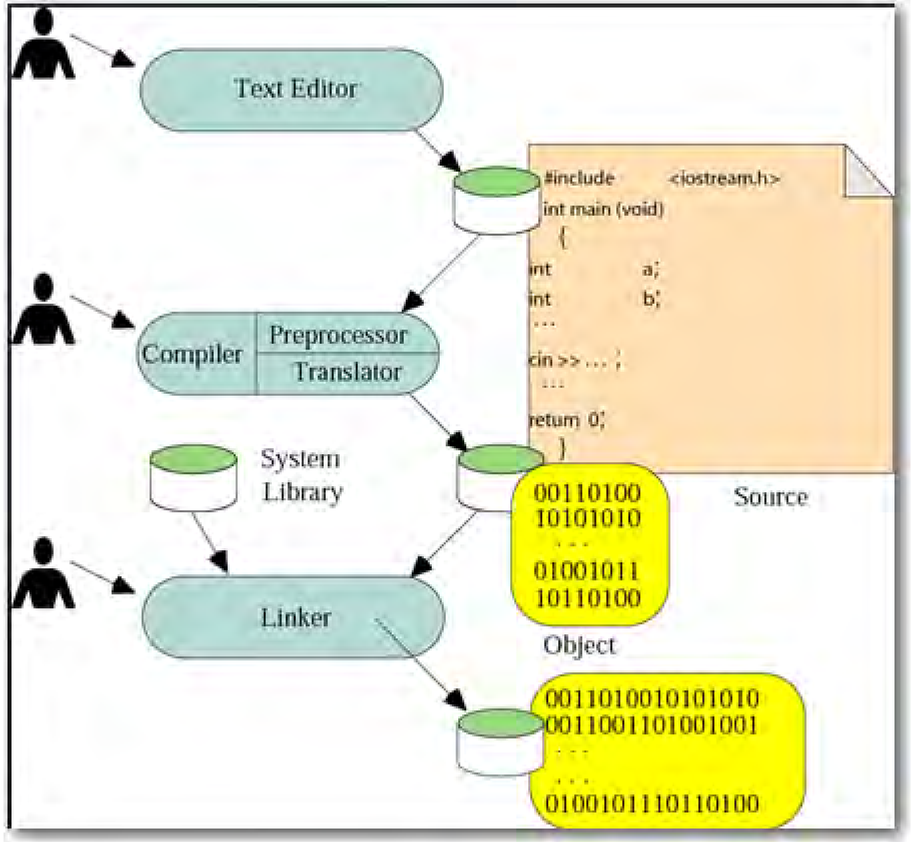
GMIT

INSTITIUID TEICNEOLAIOCHTA NA GAILLIMHE-MAIGH EO

GALWAY- MAYO INSTITUTE OF TECHNOLOGY

COMPUTER PROGRAM TRANSLATION

BUILDING AND EXECUTING A PROGRAM



```
#include <iostream.h>
int main (void)
{
int a;
int b;
...
cin >> ... ;
...
return 0;
}
```

```
00110100
10101010
...
01001011
10110100
```

```
0011010010101010
0011001101001001
...
...
0100101110110100
```

TRANSLATION SYSTEMS

Set of programs used to develop software

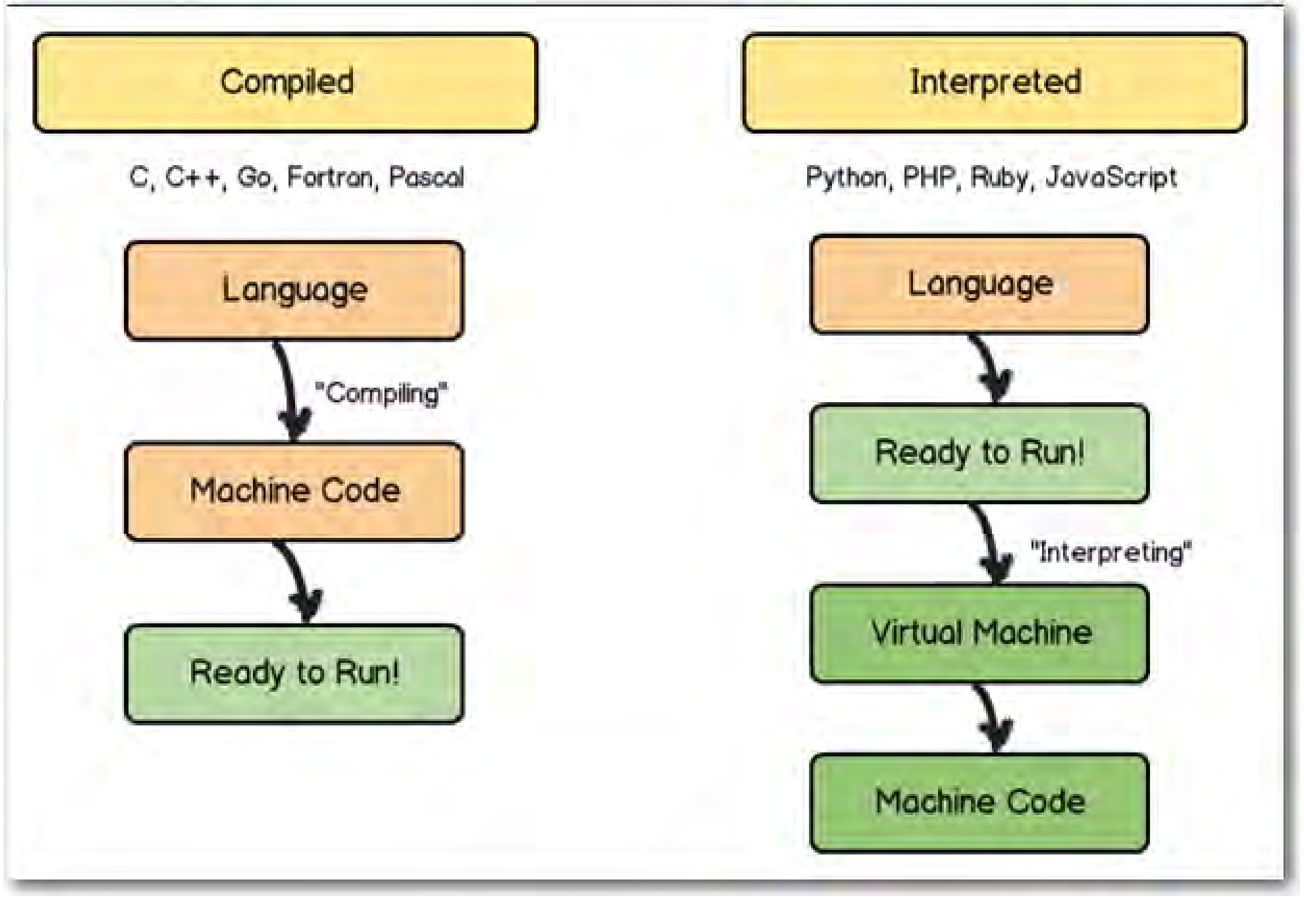
Types of translators:

- Interpreter
- Compiler
- Linker

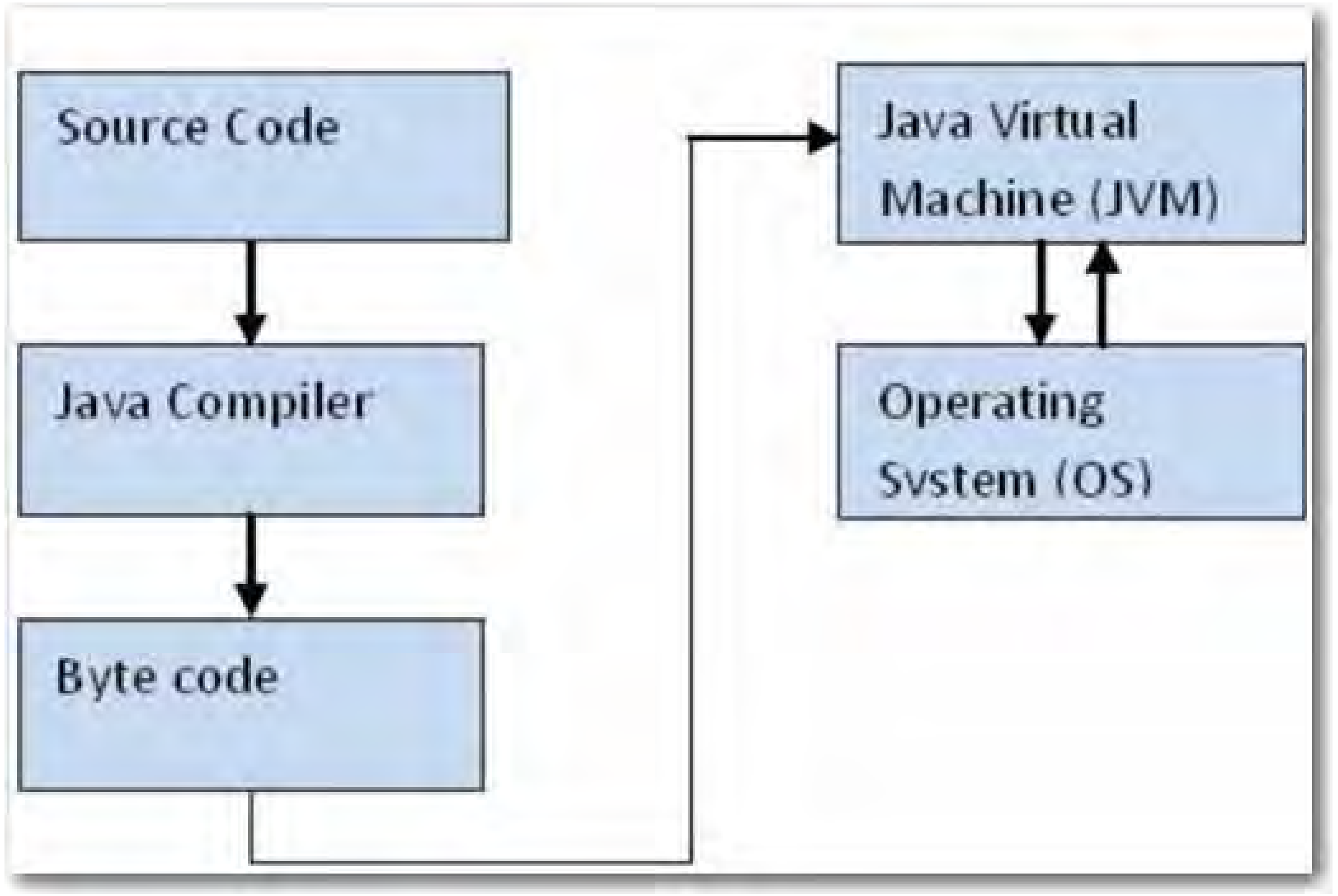
Examples

Microsoft Visual C++, javac, python interpreter

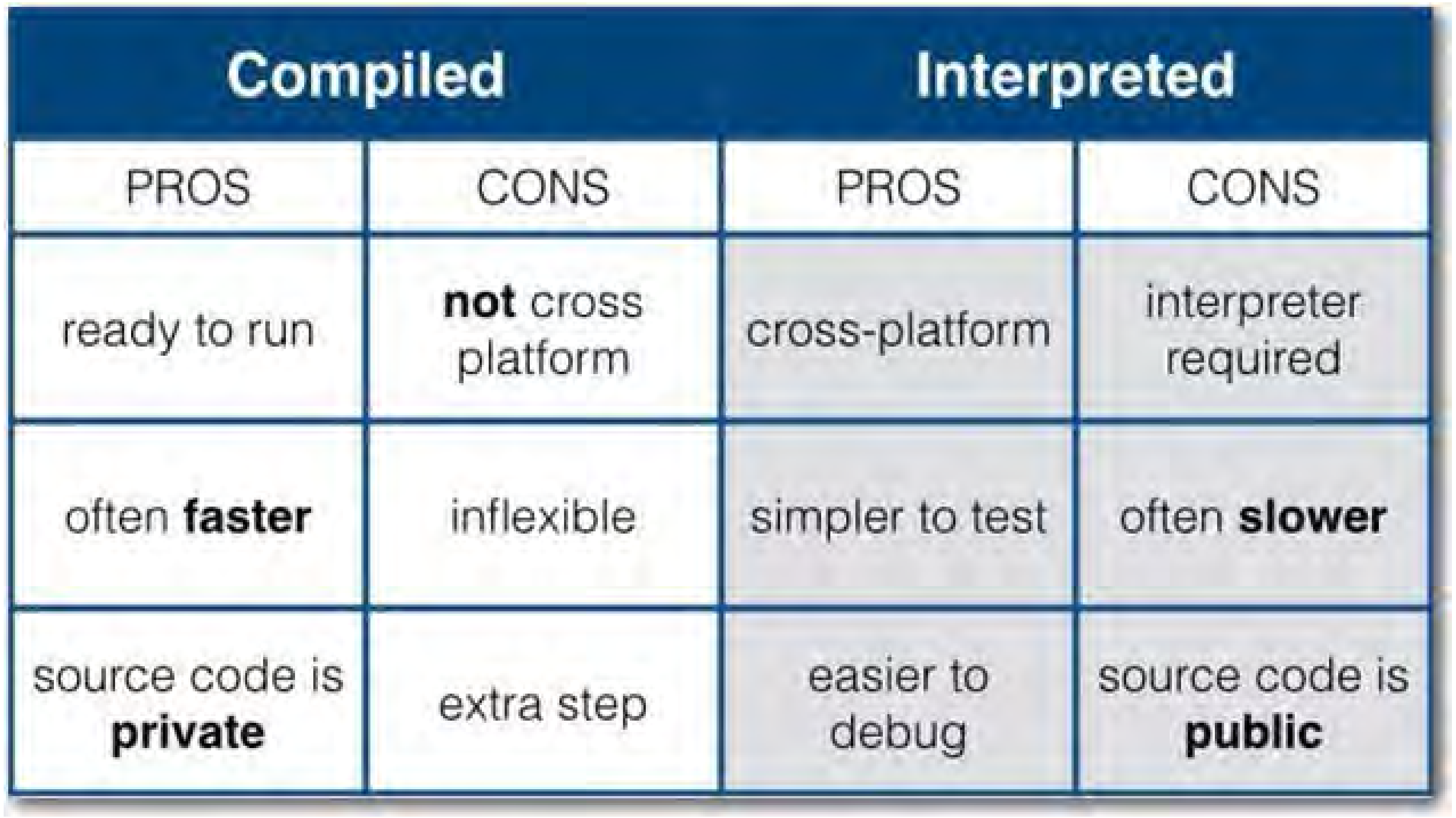
INTERPRETED VS COMPILED LANGUAGES



INTERPRETED LANGUAGES



INTERPRETED VS COMPILED LANGUAGES



Compiled		Interpreted	
PROS	CONS	PROS	CONS
ready to run	**not** cross platform	cross-platform	interpreter required
often **faster**	inflexible	simpler to test	often **slower**
source code is **private**	extra step	easier to debug	source code is **public**

IF SLOWER, WHY USE INTERPRETED LANGUAGES?

- Biggest advantage: cross platform
  - Mobile devices
  - Internet
- Java
- Python
- Matlab
- Javascript (View source in browser)

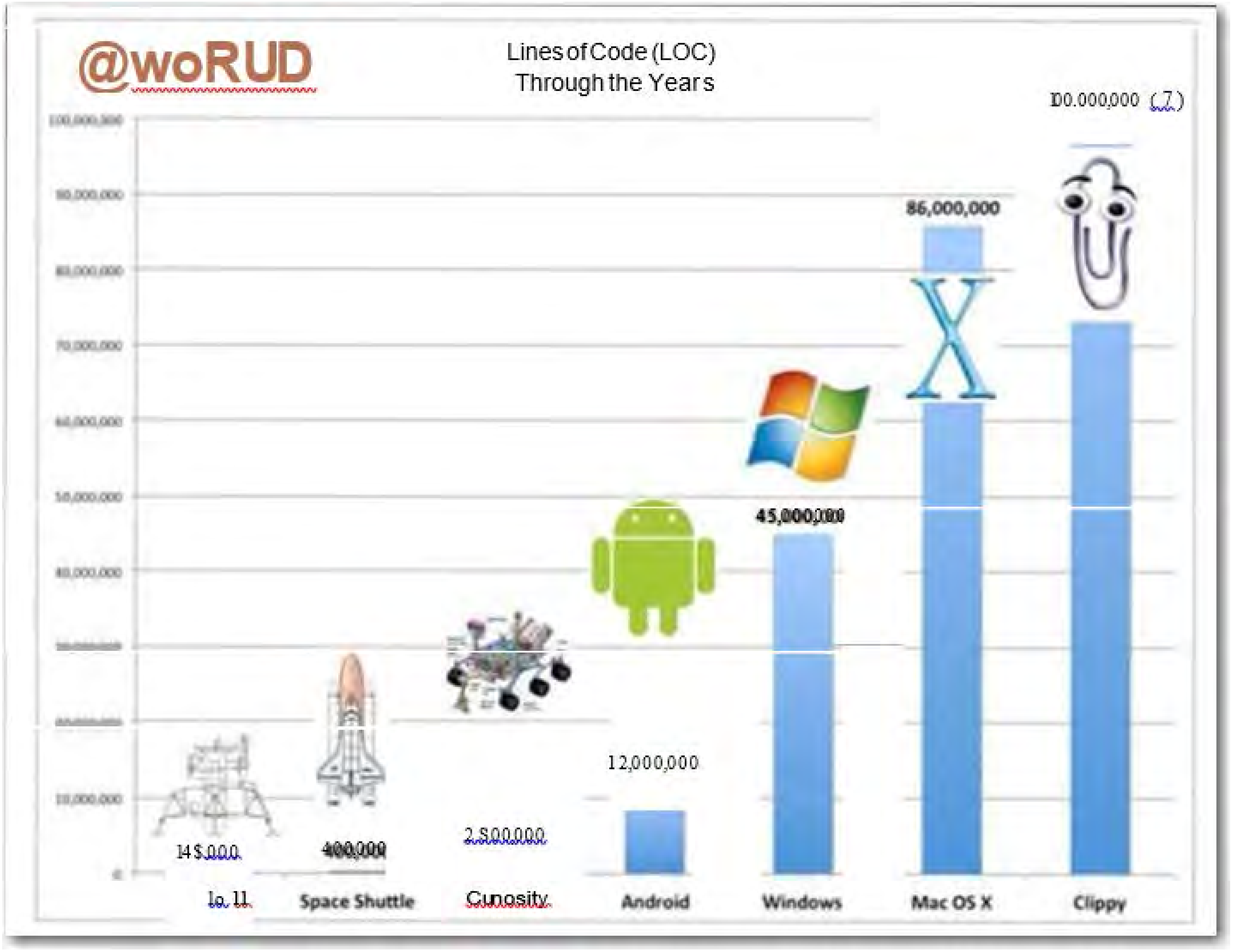
THE ERA BEFORE AUTOMATIC TRANSLATORS

Margaret Hamilton

Lead software engineer of the Apollo Project standing next to the code she wrote by hand that was used to take humanity to the moon - 1969

Invented the term software-engineering

45,000,000


@woRUD
Lines of Code (LOC)
Through the Years
100,000,000
90,000,000
80,000,000
70,000,000
60,000,000
50,000,000
40,000,000
30,000,000
20,000,000
10,000,000
0
145,000
400,000
2,500,000
12,000,000
45,000,000
86,000,000
100,000,000 (?)
Apollo 11
Space Shuttle
Curiosity
Android
Windows
Mac OS X
Clippy

GMIT

INSTITIUID TEICNEOLAIOCHTA NA GAILLIMHE-MAIGH EO

GALWAY- MAYO INSTITUTE OF TECHNOLOGY

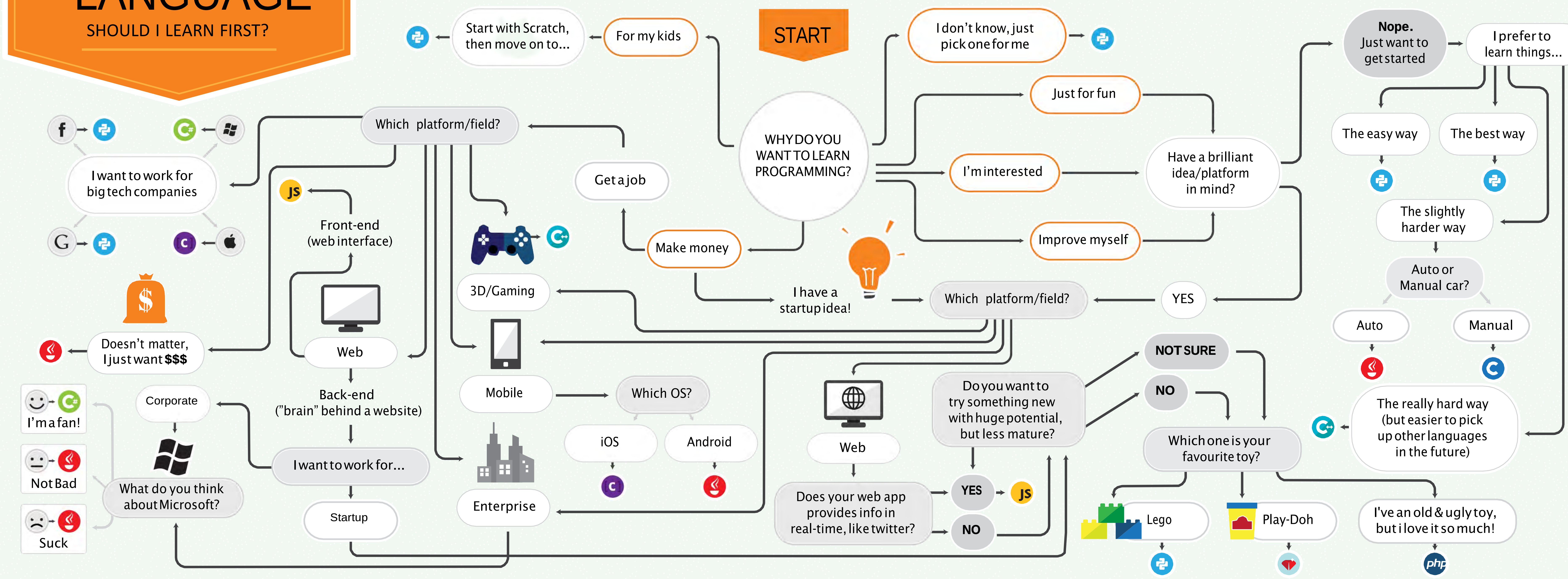
WHICH PROGRAMMING LANGUAGE SHOULD I LEARN FIRST?

WHAT IS PROGRAMMING?

Writing very specific instructions to a very dumb, yet obedient machine.

LANGUAGES

PYTHON · JAVA · C · PHP · C++
JAVASCRIPT · C# · RUBY · OBJECTIVE-C


START
WHY DO YOU WANT TO LEARN PROGRAMMING?
For my kids
Start with Scratch, then move on to...
I don't know, just pick one for me
Just for fun
I'm interested
Improve myself
Have a brilliant idea/platform in mind?
Nope. Just want to get started
I prefer to learn things...
The easy way
The best way
The slightly harder way
Auto or Manual car?
Auto
Manual
The really hard way (but easier to pick up other languages in the future)
YES
Which platform/field?
NOT SURE
NO
Do you want to try something new with huge potential, but less mature?
Which one is your favourite toy?
Lego
Play-Doh
I've an old & ugly toy, but i love it so much!
Make money
Get a job
I have a startup idea!
Which platform/field?
I want to work for big tech companies
Doesn't matter, I just want $$$
Front-end (web interface)
Web
Back-end ("brain" behind a website)
I want to work for...
Corporate
Startup
What do you think about Microsoft?
I'm a fan!
Not Bad
Suck
3D/Gaming
Mobile
Which OS?
iOS
Android
Enterprise
Web
Does your web app provides info in real-time, like twitter?
YES
NO

THE LORD OF THE RINGS ANALOGY TO PROGRAMMING LANGUAGES

Python
The Ent

DIFFICULTY ★☆☆☆☆

Help little Hobbits (beginners) to understand programming concepts

Help Wizards (computer scientists) to conduct researches

Widely regarded as the best programming language for beginners

Easiest to learn

Widely used in scientific, technical & academic field, i.e. Artificial Intelligence

You can build website using Django, a popular Python web framework

POPULARITY ★★★★½

USED TO BUILD YouTube, Instagram, Spotify

AVG. SALARY $107,000 🌐 🖥

Java
Gandalf

DIFFICULTY ★★★☆☆

Wants peace & works with everyone (portable)

Very popular on all platforms, OS, and devices due to its portability

One of the most in demand & highest paying programming languages

Slogan: write once, work everywhere

POPULARITY ★★★★★

USED TO BUILD Gmail, Minecraft, Most Android Apps, Enterprise applica-tions

AVG. SALARY $102,000 🌐 📱 🖥

C
One Ring

DIFFICULTY ★★★☆☆

The power of C is known to them all

Everyone wants to get its Power

Lingua franca of programming language

One of the oldest and most widely used language in the world

Popular language for system and hardware programming

A subset of C++ except the little details

POPULARITY ★★★★★

USED TO BUILD Operating systems and hardware

AVG. SALARY $102,000 📱 🖥

C++
Saruman

DIFFICULTY ★★★★☆

Everyone thinks that he is the good guy

But once you get to know him, you will realize he wants the power, not good deeds

Complex version of C with a lot more features

Widely used for developing games, industrial and performance-critical applications

Learning C++ is like learning how to manufacture, assemble, and drive a car

Recommended only if you have a mentor to guide you

POPULARITY ★★★★½

USED TO BUILD Operating systems, hardware, and browsers

AVG. SALARY $104,000 📱 🖥

JavaScript
Hobbit

DIFFICULTY ★★☆☆☆

Frequently underestimated (powerful)

Well-known for the slow, gentle life of the Shire (web browsers)

"Java and Javascript are similar like Car and Carpet are similar" - Greg Hewgill

Most popular clients-side web scripting language

A must learn for front-end web developer (HTML and CSS as well)

One of the hottest programming language now, due to its increasing popularity as server-side language (node.js)

POPULARITY ★★★★☆

USED TO BUILD Paypal, front-end of majority websites

AVG. SALARY $99,000 🌐 📱

C#
Elf

DIFFICULTY ★★★☆☆

Beautiful creature (language), used to stay in their land, Rivendell (Microsoft Platform), but recently started to open up to their neighbours (open source)

A popular choice for enterprise to create websites and Windows application using .NET framework

Can be used to build website with ASP.NET, a web framework from Microsoft

Similar to Java in basic syntax and some features

POPULARITY ★★★★½

USED TO BUILD Enterprise and Windows applica-tions

AVG. SALARY $94,000 🌐 📱 🖥

Ruby
Man (Middle Earth)

DIFFICULTY ★★☆☆☆

Very emotional creature

They (some Ruby developers) feel they are superior & need to rule the Middle Earth

Mostly known for its popular web framework, Ruby on Rails

Focuses on getting things done

Designed for fun and productive coding

Best for fun and personal projects, startups, and rapid development

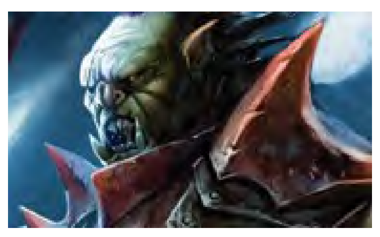
POPULARITY ★★★½☆

USED TO BUILD Hulu, Groupon, Slideshare

AVG. SALARY $107,000 🌐 🖥

PHP
Orc

DIFFICULTY ★★☆☆☆



Ugly guy (language) and doesn't respect the rules (inconsistent and unpredictable)

Big headache to those (developers) to manage them (codes)

Yet still dominates the Middle-earth (most popular web scripting language)

Suitable for building small and simple sites within a short time frame

Supported by almost every web hosting services with lower price

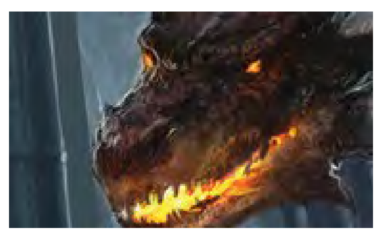
POPULARITY ★★★★☆

USED TO BUILD Wordpress, Wikipe-dia, Flickr

AVG. SALARY $89,000 🌐

Objective-C
Smaug

DIFFICULTY ★★★☆☆



Lonely and loves gold

Primary language used by Apple for Mac OS X & iOS

Choose this if you want to focus on developing iOS or OS X apps only

Consider to learn Swift (newly introduced by Apple in 2014) as your next language

POPULARITY ★★★☆☆

USED TO BUILD Most iOS Apps and part of Mac OS X

AVG. SALARY $107,000 📱 🖥

ACTUALLY...
IT DOESN'T REALLY MATTER HOW YOU START.

You need to know at least few languages to understand the underlying concepts. Just get your feet wet!

TO GET STARTED, CHECK OUT THE FULL LIST OF BEST TUTORIALS AND TOOLS FOR EACH PROGRAMMING LANGUAGE AT:

CARLCHEO.COM/STARTCODING

SPECIAL THANKS TO
Prithviraj Udaya for his awesome The Lord of the Rings analogy on Quora http://www.quora.com/If-there-was-a-war-of-programming-languages-which-would-you-support-and-why

SOURCES
Salary data from Indeed.com | http://stackoverflow.com/questions/245062/whats-the-difference-between-javascript-and-java | http://spectrum.ieee.org/static/interactive-the-top-programming-languages
http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext | http://www.itworld.com/article/2693638/big-data/the-most-in-demand--and-valuable--programming-languages.html

IMAGES
http://nightfurylive.com/media/2013/12/Smaug-fan-art-under-the-lonely-mountain.jpg | http://blog.elope.com/five-things-you-should-know-about-the-one-ring/

PRESENTED BY
CarlCheo.com

GMIT

INSTITIUID TEICNEOLAIOCHTA NA GAILLIMHE-MAIGH EO

GALWAY- MAYO INSTITUTE OF TECHNOLOGY

RECAP & ABSTRACTION

ABSTRACTION

A technique for separation of complexity within computer systems.

Establish a level of complexity on which a user/programmer interacts with the system, suppressing the more complex details below the current level.

Programmer works with well defined interfaces – allows addition of levels of functionality that would otherwise be too complex to handle.

E.g. When writing code that involves numerical operations - not interested in the way numbers are represented in the underlying hardware

ABSTRACTION

Control abstraction – abstraction of actions

  use of subroutines/functions/methods and control flow abstractions

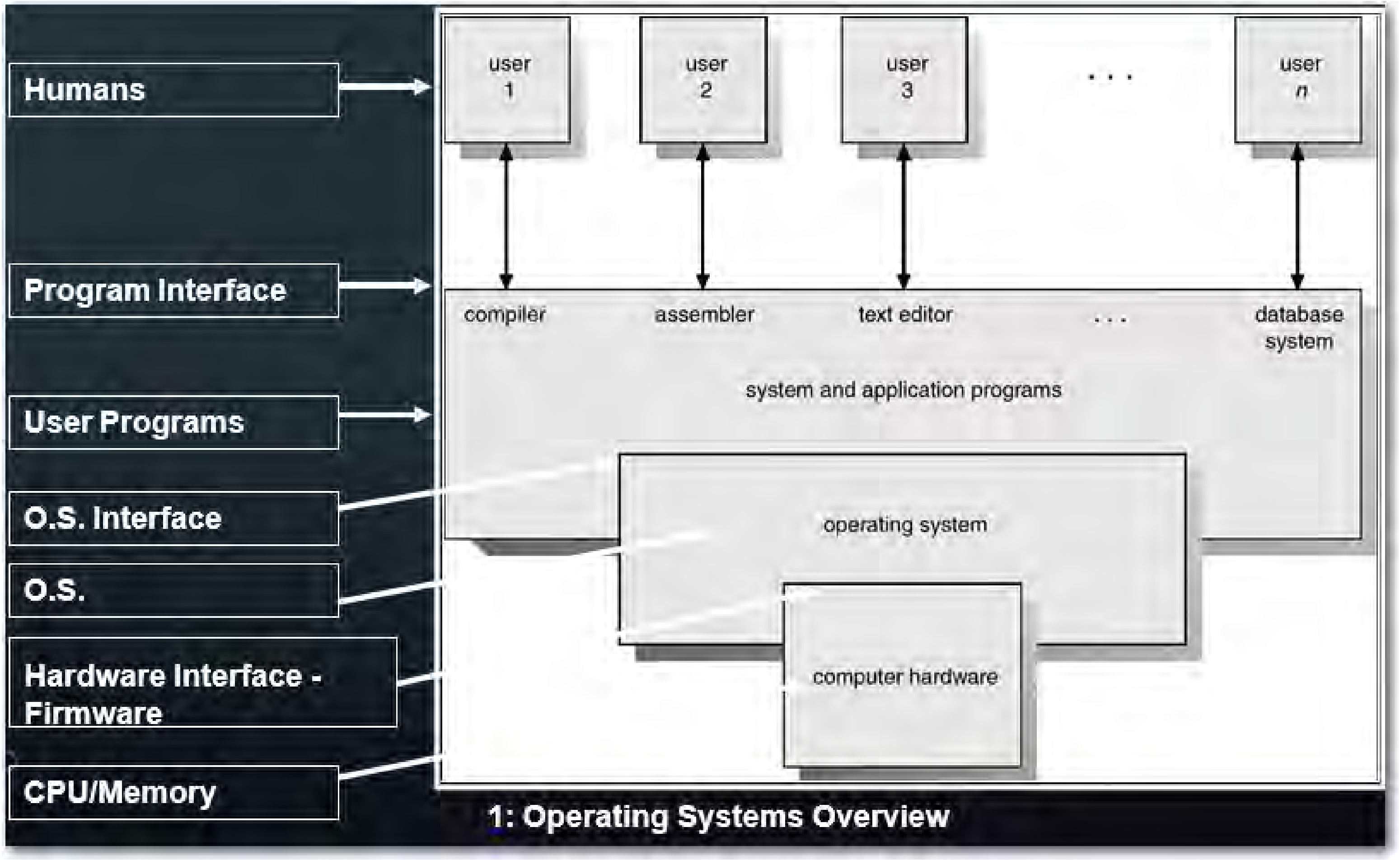
Data abstraction

  Use of data structures and data types - allows handling pieces of data in meaningful ways

OO programming employs both types – incorporates into objects

  3 tenets of encapsulation, inheritance and polymorphism

LAYERS (AND INTERFACES) WITHIN A COMPUTER SYSTEM



GMIT

INSTITIUID TEICNEOLAIOCHTA NA GAILLIMHE-MAIGH EO

GALWAY- MAYO INSTITUTE OF TECHNOLOGY

FIRMWARE

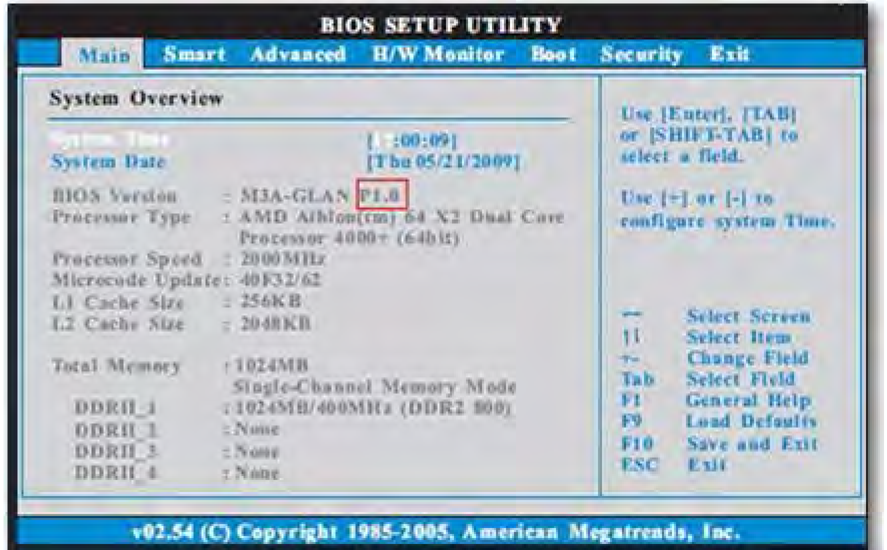
FIRMWARE

- Permanent software programmed/configured into a read-only (non-volatile) memory.
  - ROM/Flash memory (can now be updated e.g. update phone antenna to 4G)
- Does not run on a CPU (hence not software)
- Software runs on a reprogrammable device (Turing machine)
- Firmware is a type of software that provides control, monitoring and data manipulation of engineered products and systems.
  - E.g. traffic lights, consumer appliances, computers, mobile phones, and digital camras, ABS on vehicles
- Lots of hacking involves reprogramming firmware

BIOS (BASIC INPUT/OUTPUT SYSTEM)

FUNCTIONS:

- BIOS Drivers
  - Access and set-up computer system at most basic level
- Power-on Self Test (POST)
  - checks RAM, keyboard, other basic devices
- Determines boot device
  - Locates and loads OS into RAM from non-volatile memory
  - By reading the Master Boot Record
- BIOS resides in Flash memory
  - Sits on the motherboard – often motherboard specific
  - Can configure time, date and password



```
System Overview

System Time                       [  :00:09]
System Date                       [Thu 05/21/2009]

BIOS Version      : M3A-GLAN P1.0
Processor Type    : AMD Athlon(tm) 64 X2 Dual Core
                    Processor 4000+ (64bit)
Processor Speed   : 2000MHz
Microcode Update: 40F32/62
L1 Cache Size     : 256KB
L2 Cache Size     : 2048KB

Total Memory      : 1024MB
                    Single-Channel Memory Mode
  DDRII_1         : 1024MB/400MHz (DDR2 800)
  DDRII_2         : None
  DDRII_3         : None
  DDRII_4         : None

Use [Enter], [TAB]
or [SHIFT-TAB] to
select a field.

Use [+] or [-] to
configure system Time.

←→      Select Screen
↑↓      Select Item
+-      Change Field
Tab     Select Field
F1      General Help
F9      Load Defaults
F10     Save and Exit
ESC     Exit

v02.54 (C) Copyright 1985-2005, American Megatrends, Inc.
```

UEFI (UNIFIED EXTENSIBLE FIRMWARE INTERFACE)

Successor to BIOS since 2014

- No Master Boot Record
- Instead -> GUID Partition Table (GUID – Globally Unique Identifiers)

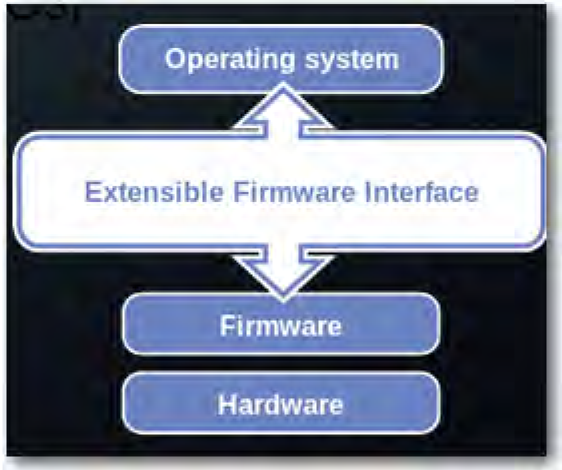
Spec that defines interface between Operating System (OS) and platform firmware

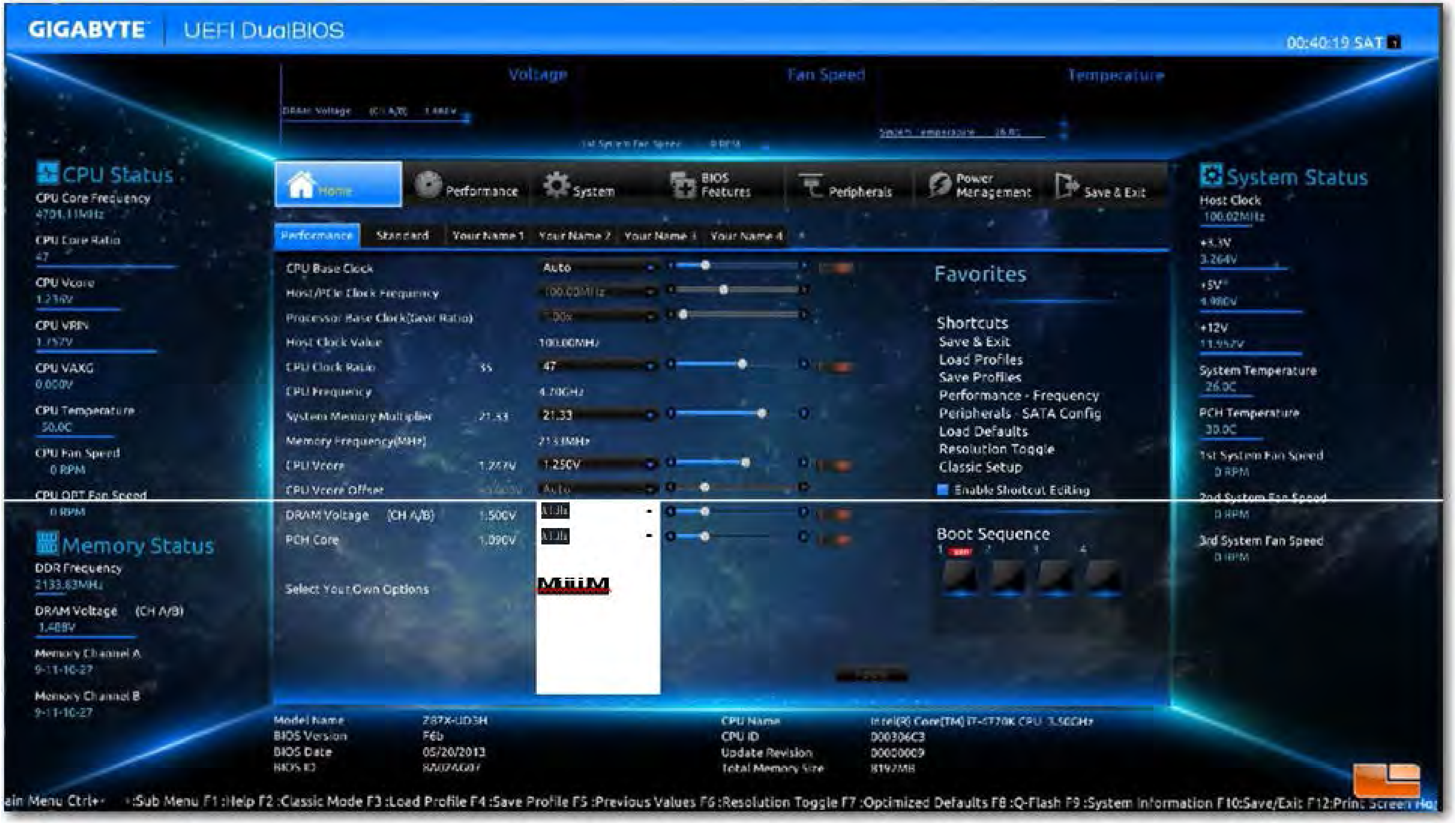
Support remote diagnostic and computer repair (even with no OS)

- Pre-OS network capability

Can boot from large (>2TB) disks

CPU independent drivers





CPU Status
CPU Core Frequency
4701.11MHz
CPU Core Ratio
47
CPU Vcore
1.236V
CPU VRIN
1.757V
CPU VAXG
0.000V
CPU Temperature
50.0C
CPU Fan Speed
0 RPM
CPU OPT Fan Speed
0 RPM

Memory Status
DDR Frequency
2133.83MHz
DRAM Voltage (CH A/B)
1.488V
Memory Channel A
9-11-10-27
Memory Channel B
9-11-10-27

CPU Base Clock		Auto
Host/PCIe Clock Frequency		100.00MHz
Processor Base Clock(Gear Ratio)		1.00x
Host Clock Value		100.00MHz
CPU Clock Ratio	35	47
CPU Frequency		4.70GHz
System Memory Multiplier	21.33	21.33
Memory Frequency(MHz)		2133MHz
CPU Vcore	1.247V	1.250V
CPU Vcore Offset	[illegible]	Auto
DRAM Voltage (CH A/B)	1.500V	Auto
PCH Core	1.090V	Auto
Select Your Own Options		MiiiM

System Status
Host Clock
100.02MHz
+3.3V
3.264V
+5V
4.980V
+12V
11.952V
System Temperature
26.0C
PCH Temperature
30.0C
1st System Fan Speed
0 RPM
2nd System Fan Speed
0 RPM
3rd System Fan Speed
0 RPM

Model Name	Z87X-UD3H	CPU Name	Intel(R) Core(TM) i7-4770K CPU 3.50GHz
BIOS Version	F6b	CPU ID	000306C3
BIOS Date	05/20/2013	Update Revision	00000009
BIOS ID	8A07AG07	Total Memory Size	8192MB

ain Menu Ctrl+ :Sub Menu F1 :Help F2 :Classic Mode F3 :Load Profile F4 :Save Profile F5 :Previous Values F6 :Resolution Toggle F7 :Optimized Defaults F8 :Q-Flash F9 :System Information F10:Save/Exit F12:Print Screen Ho

THE OPERATING SYSTEM

OPERATING SYSTEM (OS)

- **What is an OS and what 6 functions does it provide?**
- An interface between users and hardware - an environment "architecture"

  Performs basic computer tasks – E.g. managing the various peripheral devices (mouse, keyboard)

  Security - provides information protection

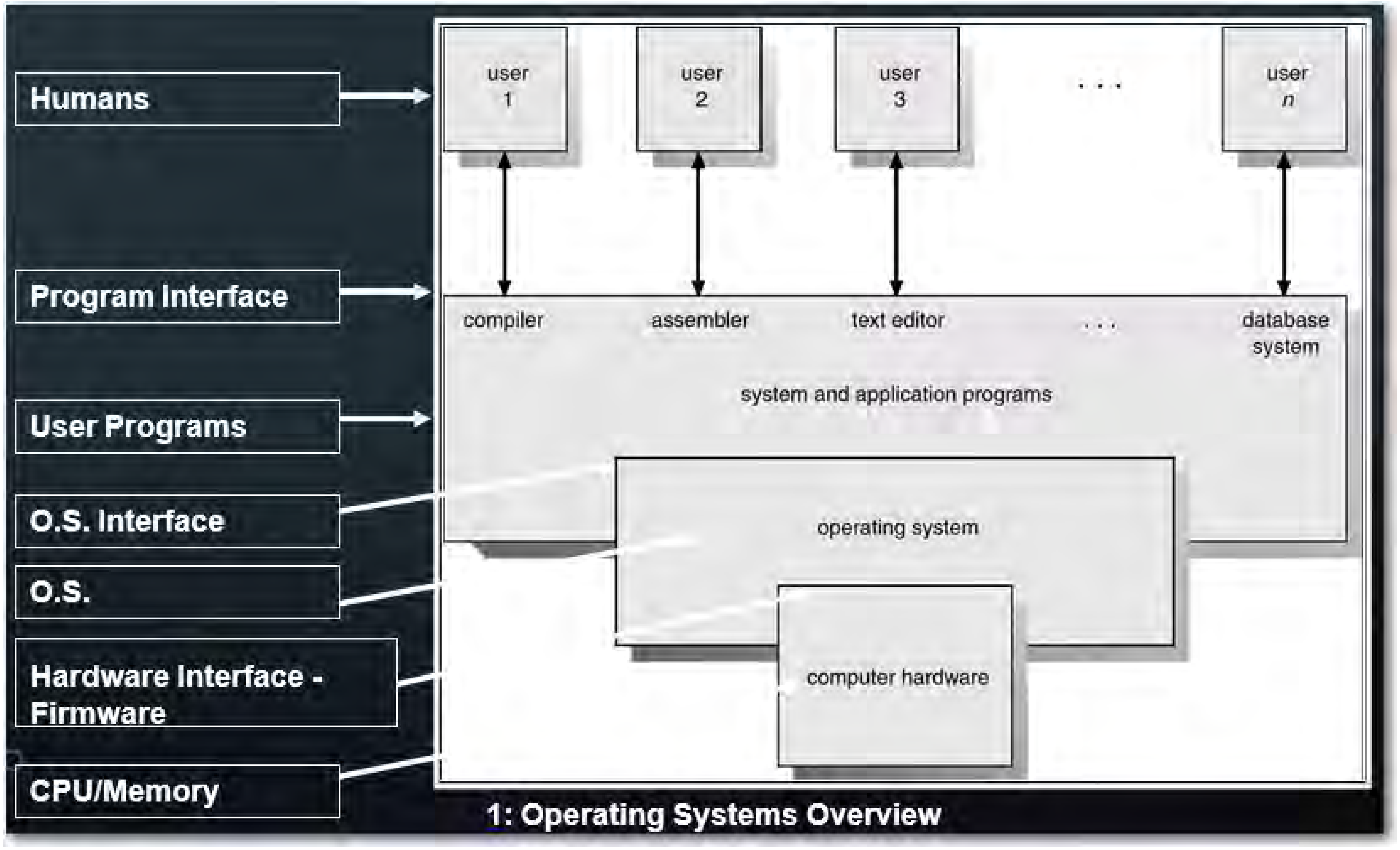
  Time Multiplexing - parallel activity, avoids wasted CPU cycles, gives each application/user a **slice** of the resources

  Booting the computer

  User Interface – e.g. command line, graphical user interface (GUI)

  File and Memory Management – how data is saved and retrieved

LAYERS (AND INTERFACES) WITHIN A COMPUTER SYSTEM



OPERATING SYSTEM SERVICES

- Program execution/Process management - load program into memory and run it

  Time Multiplexing -

  Achieved using OS Interrupts

  Enables resource scheduling

  Offers deadlock protection

  Memory Management

  Achieved through caching

  Memory protection

  File System Manipulation - read, write, create, delete files

OPERATING SYSTEM SERVICES

Protection and Security

I/O Operations – manages mouse, keyboard, etc.

Communications/Networking - interprocess and intersystem

Error Detection - in hardware, I/O devices, user programs

System Generation

Services for providing efficient system operation

- Accounting - usage statistics
- Protection - ensure access to system resources is controlled

GMIT

INSTITIUID TEICNEOLAIOCHTA NA GAILLIMHE-MAIGH EO

GALWAY- MAYO INSTITUTE OF TECHNOLOGY

TIME MULTIPLEXING / SCHEDULING

TIME MULTIPLEXING

Schedules jobs or processes.

- Scheduling can be as simple as running the next process, or it can use relatively complex rules to pick a running process.

Simultaneous CPU execution and IO handling.

- Processing is going on even as IO is occurring in preparation for future CPU work.

CPU is wasted if a job waits for I/O. This leads to:

- **Multiprogramming** (dynamic switching). While one job waits for a resource, the CPU can find another job to run. It means that several jobs are ready to run on the CPU

TIME MULTIPLEXING - OS INTERRUPTS

Interrupts

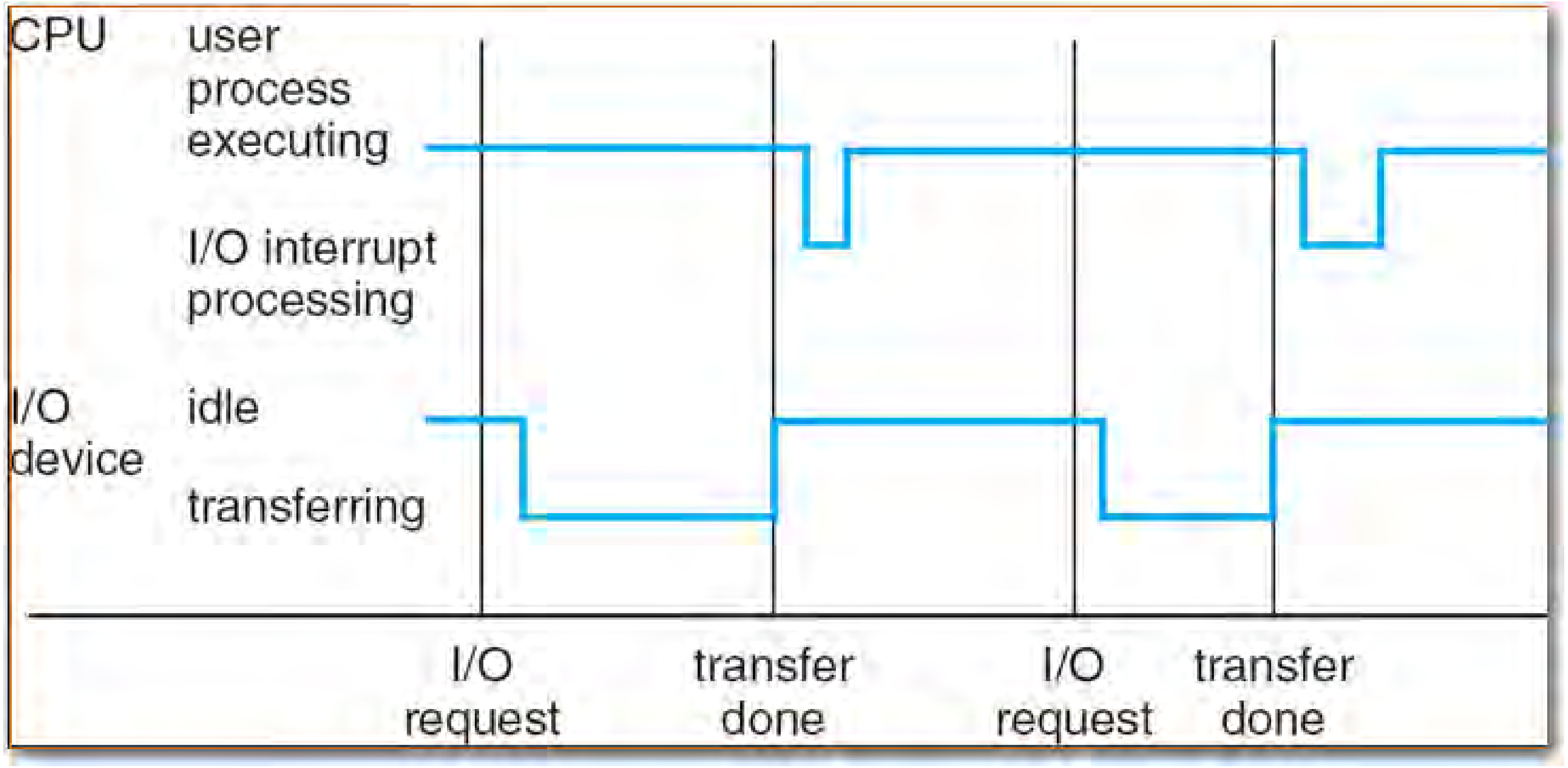
OS transfers control from one process to another

Interrupt architecture must save the address of the interrupted instruction so that it can resume at the same location.

Incoming interrupts are disabled while another interrupt is being processed to prevent a lost interrupt. (e.g. keypresses/mouse clicks can be disabled)

An operating system is interrupt driven.

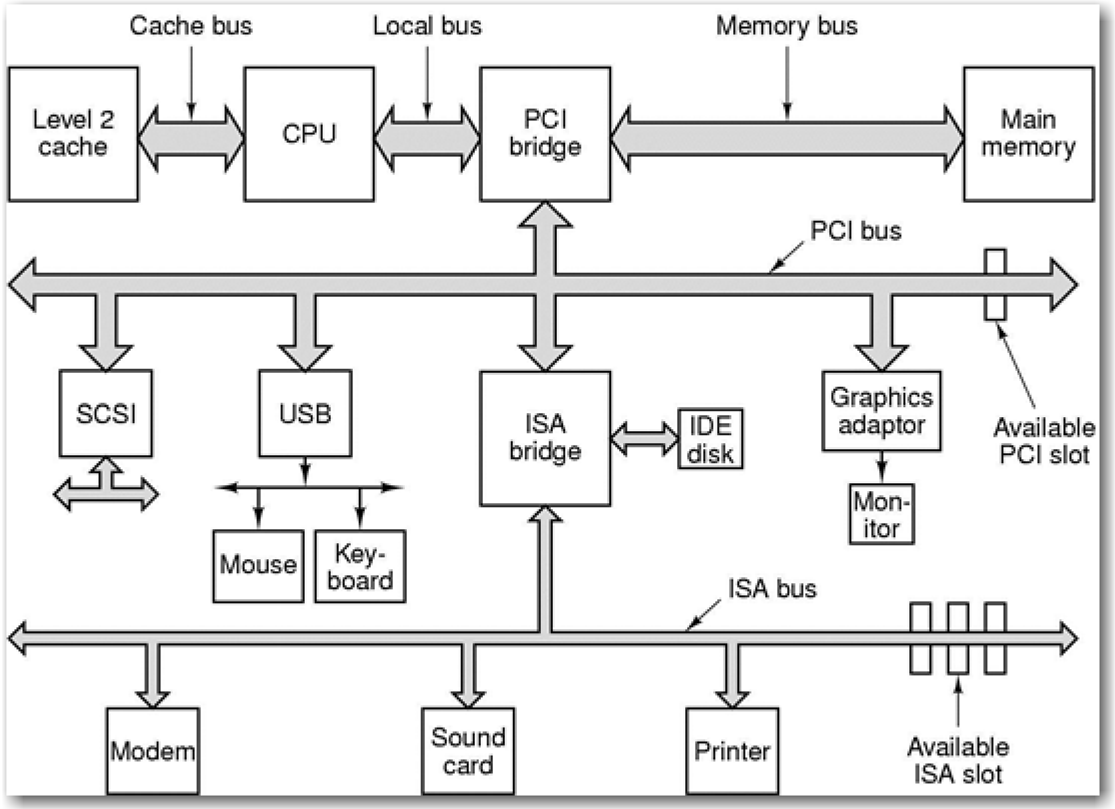
TIME MULTIPLEXING - OS INTERRUPTS



TIME MULTIPLEXING - OS INTERRUPTS

These are the devices that make up a typical system.

Any of these devices can cause an electrical interrupt that grabs the attention of the CPU.



TIME MULTIPLEXING - TIMER INTERRUPTS - DESCRIPTION AND USES

Timer - interrupts computer after specified period to ensure that OS maintains control.

- Timer is decremented every clock tick.
- When timer reaches a value of 0, an interrupt occurs.

Timer is commonly used to implement time sharing.

Timer is also used to compute the current time.

Load timer is a privileged instruction.

SCHEDULING - LINK

https://www.cs.rutgers.edu/~pxk/416/notes/07-scheduling.html

GMIT

INSTITIUID TEICNEOLAIOCHTA NA GAILLIMHE-MAIGH EO

GALWAY- MAYO INSTITUTE OF TECHNOLOGY

Process Scheduling

Who gets to run next?

By Paul Krzyzanowski
February 18, 2015

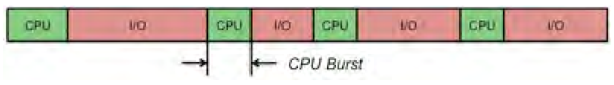
The finest eloquence is that which gets things done; the worst is that which delays them.
— David Lloyd George (speech, Jan. 1919)

Introduction

Introduction

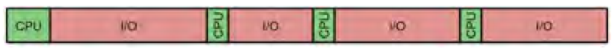
If you look at any process, you'll notice that it spends some time executing instructions (computing) and then makes some I/O request, for example, to read or write data to a file or to get input from a user. After that, it executes more instructions and then, again, waits on I/O. The period of computation between I/O requests is called the CPU burst.

Interactive processes spend more time waiting for I/O and generally experience short CPU bursts. A text editor is an example of an interactive process with short CPU bursts.



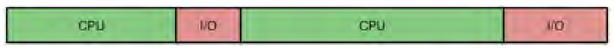
CPU bursts

Compute-intensive processes, conversely, spend more time running instructions and less time on I/O. They exhibit long CPU bursts. A video transcoder is an example of a process with long CPU bursts. Even though it reads and writes data, it spends most of its time processing that data.



Interactive bursts

Most interactive processes, in fact, spend the vast bulk of their existence doing nothing but waiting on data. As I write this on my Mac, I have 44 processes running just under my user account. This includes a few browser windows, a word processor, spreadsheet, several shell windows, Photoshop, iTunes, and various monitors and utilities. Most of the time, all these processes collectively are using less than 3% of the CPU. This is not surprising since most of these programs are waiting for user input, a network message, or sleeping and waking up periodically to check some state.



Compute bursts

Consider a 2.4 GHz processor. It executes approximately 2,400 million instructions per second (and this does not even count multi-core or hyperthreaded processors). It can run 24 billion instructions in the ten seconds it might take you to skim a web page before you click on a link — or 1.2 billion instructions in the half second it might take you to hit the next key as you're typing. The big idea in increasing overall system throughput was the realization that we could keep several programs in memory and switch the processor to run another process when a process has to wait on an I/O operation. This is multiprogramming. The next big idea was realizing that you could preempt a process and let another process run and do this quickly enough to give the illusion that many processes are running at the same time. This is multitasking.

Scheduler

Most systems have a large number of processes with short CPU bursts interspersed between I/O requests and a small number of processes with long CPU bursts. To provide good time-sharing performance, we may preempt a running process to let another one run. The ready list, also known as a run queue, in the operating system keeps a list of all processes that are ready to run and not blocked on input/output or another blocking system request, such as a semaphore. The entries in this list are pointers to a process control block, which stores all information and state about a process.

When an I/O request for a process is complete, the process moves from the waiting state to the ready state and gets placed on the run queue.

The process scheduler is the component of the operating system that is responsible for deciding whether the currently running process should continue running and, if not, which process should run next. There are four events that may occur where the scheduler needs to step in and make this decision:

1. The current process goes from the running to the waiting state because it issues an I/O request or some operating system request that cannot be satisfied immediately.
2. The current process terminates.
3. A timer interrupt causes the scheduler to run and decide that a process has run for its allotted interval of time and it is time to move it from the running to the ready state.
4. An I/O operation is complete for a process that requested it and the process now moves from the waiting to the ready state. The scheduler may then decide to preempt the currently-running process and move this newly-ready process into the running state.

A scheduler is a preemptive scheduler if it has the ability to get invoked by an interrupt and move a process out of a running state to let another process run. The last two events in the above list may cause this to happen. If a scheduler cannot take the CPU away from a process then it is a cooperative, or non-preemptive scheduler. Old operating systems such as Microsoft Windows 3.1 or Apple MacOS prior to OS X are examples of cooperative schedulers. Older batch processing systems had run-to-completion schedulers where a process ran to termination before any other process would be allowed to run.

The decisions that the scheduler makes concerning the sequence and length of time that processes may run is called the scheduling algorithm (or scheduling policy). These decisions are not easy ones, as the scheduler has only a limited amount of information about the processes that are ready to run. A good scheduling algorithm should:

- Be fair – give each process a fair share of the CPU, allow each process to run in a reasonable amount of time.
- Be efficient – keep the CPU busy all the time.
- Maximize throughput – service the largest possible number of jobs in a given amount of time; minimize the amount of time users must wait for their results.
- Minimize response time – interactive users should see good performance.
- Be predictable – a given job should take about the same amount of time to run when run multiple times. This keeps users sane.
- Minimize overhead – don't waste too many resources. Keep scheduling time and context switch time at a minimum.
- Maximize resource use – favor processes that will use underutilized resources. There are two motives for this. Most devices are slow compared to CPU operations. We'll achieve better system throughput by keeping devices busy as often as possible. The second reason is that a process may be holding a key resource and other, possibly more

important, processes cannot use it until it is released. Giving the process more CPU time may free up the resource quicker.

- Avoid indefinite postponement – every process should get a chance to run eventually.
- Enforce priorities – if the scheduler allows a process to be assigned a priority, it should be meaningful and enforced.
- Degrade gracefully – as the system becomes more heavily loaded, performance should deteriorate gradually, not abruptly.

It is clear that some of these goals are contradictory. For example, minimizing overhead means that jobs should run longer, thus hurting interactive performance. Enforcing priorities means that high-priority processes will always be favored over low-priority ones, causing indefinite postponement. These factors make scheduling a task for which there can be no perfect algorithm.

To make matters even more complex, the scheduler does not know much about the behavior of each process and certainly has no idea of what the process will do in the future. As we saw earlier, some processes perform a lot of input/output operations but use little CPU time (examples are web browsers, shells and editors). They spend much of their time in the blocked state in between little bursts of computation. The overall performance of these I/O bound processes is constrained by the speed of the I/O devices. CPU-bound processes and spend most of their time computing (examples are ray-tracing programs and circuit simulators). Their execution time is largely determined by the speed of the CPU and the amount of CPU time they can get.

To help the scheduler monitor processes and the amount of CPU time that they use, a programmable interval timer interrupts the processor periodically (typically 100 times per second). This timer is programmed when the operating system initializes itself. At each interrupt, the operating system's scheduler gets to run and decide whether the currently running process should be allowed to continue running or whether it should be suspended and another ready process allowed to run. This is the mechanism that enables preemptive.

Preemptive scheduling allows the scheduler to control response times by taking the CPU away from a process that it decided has been running too long in order to let another process run. It incurs more overhead than nonpreemptive scheduling since it has to deal with the overhead of context switching processes instead of allowing a process to run to completion or run until the next I/O operation or other system call. However, it allows for higher degrees of concurrency and better interactive performance.

The scheduling algorithm has the task of figuring out whether a process should be switched out for another process and which process should get to run next. The dispatcher is the component of the scheduler that handles the mechanism of actually getting that process to run on the processor. This requires loading the saved context of the selected process, which is stored in the process control block and comprises the set of registers, stack pointer, flags (status word), and a pointer to the memory mapping (typically a pointer to the page table). Once this context is loaded, the dispatcher switches to user mode via a return from interrupt operation that causes the process to execute from the location that was saved on the stack at the time that the program stopped running — either via an interrupt or a system call.

In the following sections, we will explore a few scheduling algorithms.

Let's first introduce some terms.

Turnaround time

Turnaround time is the elapsed time between the time the job arrives (e.g., you type a command) and the time that it terminates. This includes the delay of waiting for the scheduler to start the job because some other process is still running and others may be queued ahead.

Start time

Also known as release time, the start time is the time when the task is scheduled to run and actually gets to start running on the CPU.

If we look a process as a series of CPU bursts the start time applies to each CPU burst. It is the time when each CPU burst starts to run.

Response time

This is the delay between submitting a process and it being scheduled to run (its start time). Again, if we look a process as a series of CPU bursts the response time applies to each CPU burst. It is the delay between a task being ready to run and actually running.

Completion time

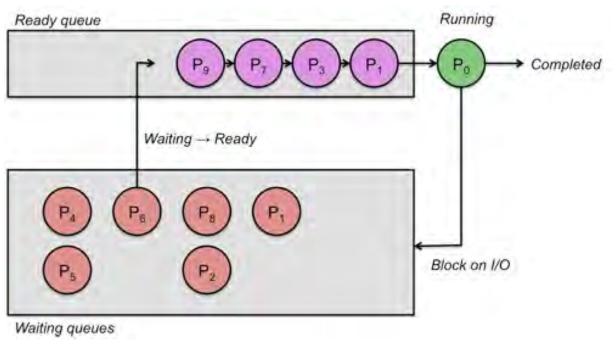
This is the time when the process terminates.

Throughput

Throughput refers to the number of processes that complete over some unit of time. By comparing throughput on several schedulers, we can get a feel of whether one scheduler is able to get more processes to complete than another over some period of time. This can be due to several factors: keeping the CPU busy, scheduling I/O as early as possible to keep disks an other slow devices busy, and the amount of overhead spent doing all of this.

First-Come, First-Served Scheduling

Possibly the most straightforward approach to scheduling processes is to maintain a FIFO (first-in, first-out) run queue. New processes go to the end of the queue. When the scheduler needs to run a process, it picks the process that is at the head of the queue. This scheduler is non-preemptive. If the process has to block on I/O, it enters the waiting state and the scheduler picks the process from the head of the queue. When I/O is complete and that waiting (blocked) process is ready to run again, it gets put at the end of the queue.



First Come - First Served

With first-come, first-served scheduling, a process with a long CPU burst will hold up other processes, increasing their turnaround time. Moreover, it can hurt overall throughput since I/O on processes in the waiting state may complete while the CPU bound process is still running. Now devices are not being used effectively. To increase throughput, it would have been great if the scheduler instead could have briefly run some I/O bound process so that could run briefly, request some I/O and then wait for that I/O to complete. Because CPU bound processes don't get preempted, they hurt interactive performance because the interactive process won't get scheduled until the CPU bound one has completed.

Advantage: FIFO scheduling is simple to implement. It is also intuitively fair (the first one in line gets to run first).

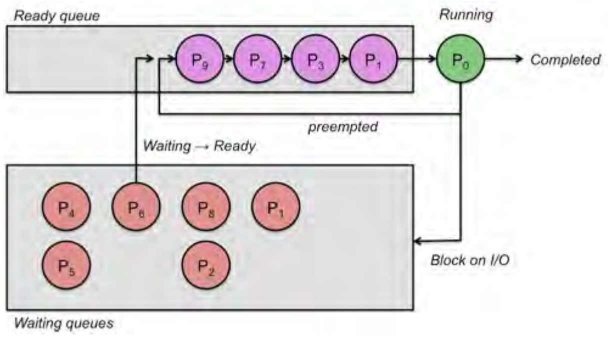
Disadvantage: The greatest drawback of first-come, first-served scheduling is that it is not preemptive. Because of this, it is not suitable for interactive jobs. Another drawback is that a long-running process will delay all jobs behind it.

Round robin scheduling

Round robin scheduling is a preemptive version of first-come, first-served scheduling. Processes are dispatched in a first-in-first-out sequence but each process is allowed to run for only a limited amount of time. This time interval is known as a time-slice or quantum. If a process does not complete or get blocked because of an I/O operation within the time slice, the time slice expires and the process is preempted. This preempted process is placed at the back of the run queue where it must wait for all the processes that were already in the queue to cycle through the CPU.

If a process gets blocked due to an I/O operation before its time slice expires, it is, of course, enters a blocked because of that I/O operation. Once that operation completes, it is placed on the end of the run queue and waits its turn.

A big advantage of round robin scheduling over non-preemptive schedulers is that it dramatically improves average response times. By limiting each task to a certain amount of time, the operating system can ensure that it can cycle through all ready tasks, giving each one a chance to run.



Round Robin Scheduling

With round robin scheduling, interactive performance depends on the length of the quantum and the number of processes in the run queue. A very long quantum makes the algorithm behave very much like first come, first served scheduling since it's very likely that a process with block or complete before the time slice is up. A small quantum lets the system cycle through processes quickly. This is wonderful for interactive processes. Unfortunately, there is an overhead to context switching and having to do so frequently increases the percentage of system time that is used on context switching rather than real work.

Advantage: Round robin scheduling is fair in that every process gets an equal share of the CPU. It is easy to implement and, if we know the number of processes on the run queue, we can know the worst-case response time for a process.

Disadvantage: Giving every process an equal share of the CPU is not always a good idea. For instance, highly interactive processes will get scheduled no more frequently than CPU-bound processes.

Setting the quantum size

What should the length of a quantum be to get "good" performance? A short quantum is good because it allows many processes to circulate through the processor quickly, each getting a brief chance to run. This way, highly interactive jobs that usually do not use up their quantum will not have to wait as long before they get the CPU again, hence improving interactive performance. On the other hand, a short quantum is bad because the operating system must perform a context switch whenever a process gets preempted. This is overhead: anything that the CPU does other than executing user code is essentially overhead. A short quantum implies many such context switches per unit time, taking the CPU away from performing useful work (i.e., work on behalf of a process).

The overhead associated with a context switch can be expressed as:

$$\text{context switch overhead} = C / (Q+C)$$

where Q is the length of the time-slice and C is the context switch time. An increase in Q increases efficiency but reduces average response time. As an example, suppose that there are ten processes ready to run, Q = 100 ms, and C = 5 ms. Process 0 (at the head of the run queue, the list of processes that are in the ready state) gets to run immediately. Process 1 can run only after Process 0's quantum expires (100 ms) and the context switch takes place (5 ms), so it starts to run at 105 ms. Likewise, process 2 can run only after another 105 ms. We can compute the amount of time that each process will be delayed and compare the delays between a small quantum (10 ms) and a long quantum (100 ms.):

	Q = 100ms	Q = 10ms
Proc #	delay (ms)	delay (ms)
0	0	0
1	105	15
2	210	30
3	315	45
4	420	60
5	525	75
6	630	90
7	735	105
8	840	120
9	945	135

We can see that with a quantum of 100 ms and ten processes, a process at the end of the queue will have to wait almost a second before it gets a chance to run. This is much too slow for interactive tasks. When the quantum is reduced to 10 ms, the last process has to wait less than 1/7 second before it gets the CPU. The downside of this is that with a quantum that small, the context switch overhead (5/(10+5)) is 33⅓%. This means that we are wasting over a third of the CPU just switching processes! With a quantum of 100 ms, the context switch overhead is just 4%.

Shortest remaining time first scheduling

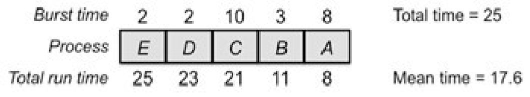
The shortest remaining time first (SRTF) scheduling algorithm is a preemptive version of an older non-preemptive algorithm known as shortest job first (SJF) scheduling. Shortest job first scheduling runs a process to completion before running the next one. The queue of jobs is sorted by estimated job length so that short programs get to run first and not be held up by long ones. This minimizes average response time.

Here's an extreme example. It's the 1950s and three users submit jobs (a deck of punched cards) to an operator. Two of the jobs are estimated to run for 3 minutes each while the third job while the third user estimates that it will take about 48 hours to run the program. With a shortest job first approach, the operator will run the three-minute jobs first and then let the computer spend time on the 48-hour job.

With the shortest remaining time first algorithm, we take into account the fact that a process runs as a series of CPU bursts: processes may leave the running state because they need to wait on I/O or because their quantum expired. The algorithm sorts the run queue by the the process' anticipated CPU burst time, picking the shortest burst time. Doing so will optimize the average response time of processes.

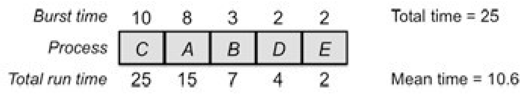
Let's consider an example of five processes in the run queue. If we process them in a FIFO manner, we see that all the CPU bursts add up to 25 (pick your favorite time unit; this is just an example). The mean run time for a process, however, is the mean of all the run times, where the run time is the time spent waiting to run + the CPU burst time of the process. In this example, our mean run time is (8 + 11 + 21 + 23 + 25)/5, or 17.6.

If we reorder the processes in the queue by the estimated CPU burst time, we still have the same overall total (the processes take the same amount of time to run) but the mean run time changes. It is now (2 + 4 + 7 + 15 + 25), or 10.6. We reduced the average run time for our processes by 40%!



Shortest Remaining Time First

Estimating future CPU burst time



Shortest Remaining Time First (sorted)

The biggest problem with sorting processes this way is that we're trying to optimize our schedule using data that we don't even have! We don't know what the CPU burst time will be for a process when it's next run. It might immediately request I/O or it might continue running for minutes (or until the expiration of its time slice).

The best that we can do is guess and try to predict the next CPU burst time by assuming that it will be related to past CPU bursts for that process. All interactive processes follow the following sequence of operations:

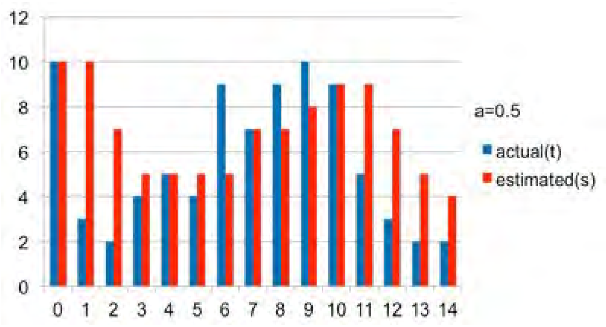
compute — I/O — compute — I/O — compute — I/O

Suppose that a CPU burst (compute) period is measured as T_0. The next compute period is measured as T_1, and so on. The common approach to estimate the length of the next CPU burst is by using a time-decayed exponential average of previous CPU bursts for the process. We will examine one such function, although there are variations on the theme. Let T_n be the measured time of the nth burst; s_n be the predicted size of the nth CPU burst; and a be a weighing factor, $0 \leq a \leq 1$. Define s_0 as some default system average burst time. The estimate of the next CPU burst period is:

$$s_{n+1} = aT_{n} + (1 - a)s_n$$

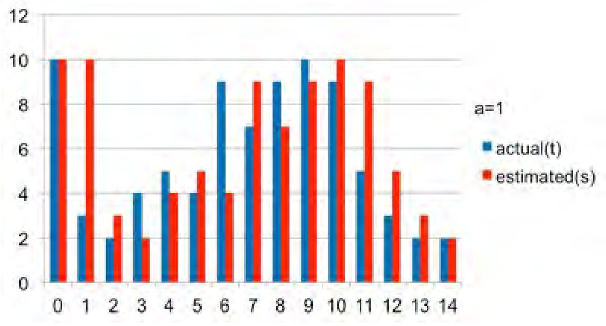
The weighing factor, a, can be adjusted how much to weigh past history versus considering the last observation. If a = 1, then only the last observation of the CPU burst period counts. If a = ½, then the last observation has as much weight as the historical weight. As a gets smaller than ½, the historical weight counts more than the recent weight.

Here is an example with a = 0.5. The blue bars represent the actual CPU burst over time. The red bars represent the estimated value. With a weighting value of ½, we can see how the red bars are strongly influenced by the previous actual value but factor in the older values.



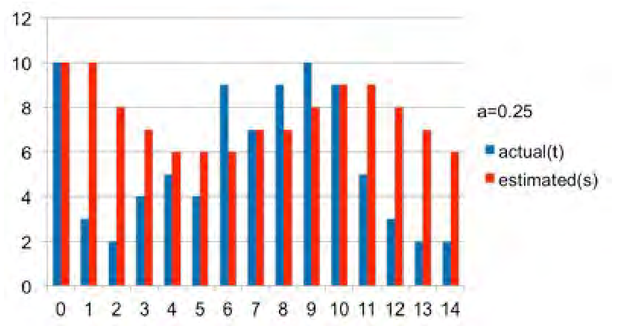
Exponential Average (a=0.5)

Now let's see what happens when we set a = 1. This ignores history and only looks at the previous CPU burst. We can see that each red bar (current estimate) has exactly the same value as the previous blue bar (latest actual CPU burst).



Exponential Average (a=1)

For a final example, let's set a= 0.25. Here, the last measured value only counts for 25% of the estimated CPU burst, with 75% being dictated by history. We can see how immediate changes in CPU burst have less impact on the estimate when compared with the above graph of a = 0.5. Note how the estimates at 2, 3, 13, and 14 still remain relatively high despite the rapid plunge of actual CPU burst values.



Exponential Average (a=0.25)

Advantage of shortest remaining time first scheduling: This scheduling is optimal in that it always produces the lowest mean response time. Processes with short CPU bursts are given priority and hence run quickly (are scheduled frequently).

Disadvantages: Long-burst (CPU-intensive) processes are hurt with a long mean waiting time. In fact, if short-burst processes are always available to run, the long-burst ones may never get scheduled. The situation where a process never gets scheduled to run is called starvation. Another problem with the algorithm is that the effectiveness of meeting the scheduling criteria relies on our ability to estimate the length of the next CPU burst.

Priority scheduling

Round robin scheduling assumes that all processes are equally important. This generally is not the case. We would sometimes like to see long CPU-intensive (non-interactive) processes get a lower priority than interactive processes. These processes, in turn, should get a lower priority than jobs that are critical to the operating system.

In addition, different users may have different status. A system administrator's processes may rank above those of a student's.

These goals led to the introduction of priority scheduling. The idea here is that each process is assigned a priority (just a number). Of all processes ready to run, the one with the highest priority gets to run next (there is no general agreement across operating systems whether a high number represents a high or low priority; UNIX-derived systems tend to use smaller numbers for high priorities while Microsoft systems tend to use higher numbers for high priorities).

With a priority scheduler, the scheduler simply picks the highest priority process to run. If the system uses preemptive scheduling, a process is preempted whenever a higher priority process is available in the run queue.

Priorities may be internal or external. Internal priorities are determined by the system using factors such as time limits, a process' memory requirements, its anticipated I/O to CPU ratio, and any other system-related factors. External priorities are assigned by administrators.

Priorities may also be static or dynamic. A process with a static priority keeps that priority for the entire life of the process.

A process with a dynamic priority will have that priority changed by the scheduler during its course of execution. The scheduler would do this to achieve its scheduling goals. For example, the scheduler may decide to decrease a process' priority to give a chance for lower-priority job to run. If a process is I/O bound (spending most if its time waiting on I/O), the scheduler may give it a higher priority so that it can get off the run queue quickly and schedule another I/O operation.

Static and dynamic priorities can coexist. A scheduler would know that a process with a static priority cannot have its priority adjusted throughout the course of its execution.

Ignoring dynamic priorities, the priority scheduling algorithm is straightforward: each process has a priority number assigned to it and the scheduler simply picks the process with the highest priority.

Advantage: priority scheduling provides a good mechanism where the relative importance of each process may be precisely defined.

Disadvantage: If high priority processes use up a lot of CPU time, lower priority processes may starve and be postponed indefinitely, leading to starvation. Another problem with priority scheduling is deciding which process gets which priority level assigned to it.

Dealing with starvation

One approach to the problem of indefinite postponement is to use dynamic priorities. At the expiration of each quantum, the scheduler can decrease the priority of the current running process (thereby penalizing it for taking that much CPU time). Eventually its priority will fall below that of the next highest process and that process will be allowed to run.

Another approach is to have the scheduler keep track of low priority processes that do not get a chance to run and increase their priority so that eventually the priority will be high enough so that the processes will get scheduled to run. Once it runs for its quantum, the priority can be brought back to the previous low level.

This periodic boosting of a process' priority to ensure it gets a chance to run is called process aging. A simple way to implement aging is to simply increase every process' priority and then have them get readjusted as they execute.

Multilevel queues

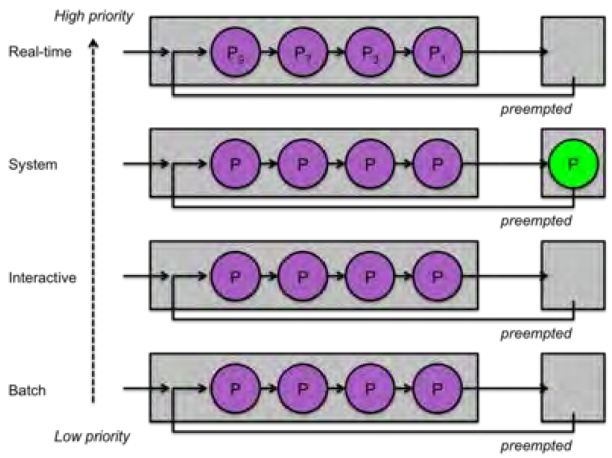
What happens if several processes get assigned the same priority? This is a realistic possibility since picking a unique priority level for each of possibly hundreds or thousands of processes on a system may not be feasible.

We can group processes into priority classes and assign a separate run queue for each class. This allows us to categorize and separate system processes, interactive processes, low-priority interactive processes, and background non-interactive processes. The scheduler picks the highest-priority queue (class) that has at least one process in it. In this sense, it behaves like a priority scheduler.

Each queue may use a different scheduling algorithm, if desired. Round-robin scheduling per priority level is the most common. As long as processes are ready in a high priority queue, the scheduler will let each of run for their time slice. Only when no processes are available to run at that priority level will the scheduler look at lower levels. Alternatively, some very high priority levels might implement the non-preemptive first-come, first-served scheduling approach to ensure that a critical real-time task gets all the processing it needs.

The scheduler may also choose a different quantum for each priority level. For example, it is common to give low-priority non-interactive processes a longer quantum. They won't get to run as often since they are in a low priority queue but, when they do, the scheduler will let them run longer. Linux, on the other hand, does the opposite. It gives a longer quantum to high-priority processes on the assumption that they are important and that they are likely to be interactive so they will usually block long before using up their time slice.

One problem with multilevel queues is that the process needs to be assigned to the most suitable priority queue a priori. If a CPU-bound process is assigned to a short-quantum, high-priority queue, that's not optimal for either the process nor for overall throughput.



Multilevel Queue

Multi-level queues are generally used as a top-level scheduling discipline to separate broad classes of processes, such as real-time, kernel threads, interactive, and background processes. Specific schedulers within each class determine which process gets to run within that class. Most operating systems, including Windows, Linux, and OS X support a form of multilevel queues and scheduling classes.

Multilevel feedback queues

A variation on multilevel queues is to allow the scheduler to adjust the priority (that is, use dynamic priorities) of a process during execution in order to move it from one queue to another based on the recent behavior of the process.

The goal of multilevel feedback queues is to automatically place processes into priority levels based on their CPU burst behavior. I/O-intensive processes will end up on higher priority queues and CPU-intensive processes will end up on low priority queues.

A multilevel feedback queue uses two basic rules:

1. A new process gets placed in the highest priority queue.
2. If a process does not finish its quantum (that is, it blocks on I/O) then it will stay at the same priority level (round robin) otherwise it moves to the next lower priority level

With this approach, a process with long CPU bursts will use its entire time slice, get preempted and get placed in a lower-priority queue. A highly interactive process will not use up its quantum and will remain at a high priority level.

Although not strictly necessary for the algorithm, each successive lower-priority queue may be given a longer quantum. This allows a process to remain in the queue that corresponds to its longest CPU burst.

Process aging

One problem with multilevel feedback queues is starvation. If there are a lot of interactive processes or if new processes are frequently created, there is always a task available in a high-priority queue and the CPU-bound processes in a low-priority queue will never get scheduled.

A related problem is that an interactive process may end up at a low priority level. If a process ever has a period where it becomes CPU-intensive, it trickles down to a low priority level and is forever doomed to remain there. An example is a game that needs to spend considerable CPU time initializing itself but then becomes interactive, spending most of its time waiting for user input.

Both these problems can be solved with process aging. As we saw earlier, we periodically increase the priority of a process to ensure that it will be scheduled to run. A simple approach is to periodically bring all processes to the highest priority queue.

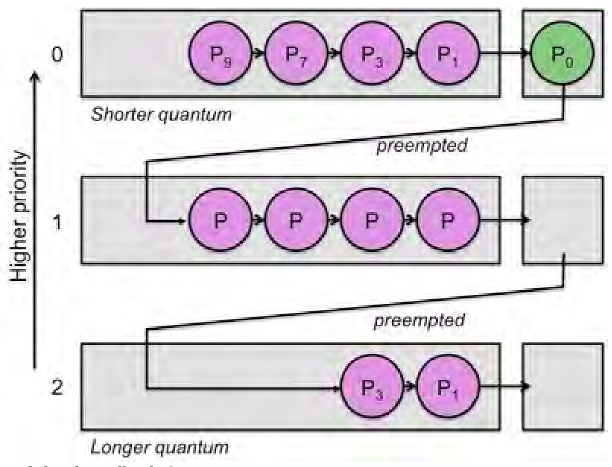
An advantage of a multilevel feedback queue is that the algorithm is designed to adjust the priority of a process whenever it runs, so a CPU-bound process will quickly trickle back down to a low priority level while an interactive process will remain at a high level.

Gaming the system

If a programmer knows how the scheduler works and wants to write software that will ensure that the process remains at a high priority level, she can write code that will force the system to block on some low-latency I/O operation (e.g., sleep for a few milliseconds) just before the quantum expires. That way, the process will be rewarded for not using up its quantum even if it repeatedly uses up a large chunk of it.

A solution to this approach is to modify the way the scheduler decides to demote the priority of a process. Instead of simply checking whether a process uses up its time slice, it keeps track of the total time that the process spent running over a larger time interval (several time slices). Each priority queue will have a maximum CPU time allotment associated with it. If a process uses up that allotment over that multi-time-slice interval the process will be punished by being moved to a lower priority level.
Advantages: Multi-level feedback queues are good for separating processes into categories based on their need for a CPU. They favor I/O bound processes by letting them run often. Versions of this scheduling policy that increase the quantum at lower priority levels also favor

CPU bound processes by giving them a larger chunk of CPU time when they are allowed to run.



Multilevel Feedback Queue

Disadvantages: The priority scheme here is one that is controlled by the system rather than by the administrator or users. A process is deemed important not because it necessarily is important, but because it happens to do a lot of I/O.

This scheduler also has the drawback that I/O-bound processes that become CPU bound or CPU-bound processes that become I/O-bound will not get scheduled well. Moreover, CPU-bound processes have the danger of starving on a system with many interactive processes. Both these problems can be dealt with by applying process aging.

Another drawback is the ability of a programmer to keep the priority of a process high by performing bogus I/O operations periodically. Again, we have a solution for this by measuring CPU use over a larger, multi-quantum time interval and punishing processes that use more of the CPU.

Lottery scheduling (fair share)

With lottery scheduling (also known as fair share scheduling), the goal is to allow a process to be granted a proportional share of the CPU - a specific percentage. Conceptually, lottery scheduling works by allocating a specific number of "tickets" to each process. The more tickets a process has, the higher its chance of being scheduled.

For example, suppose that we have 100 tickets in total and three processes to run: A, B, and C. We would like to schedule process A twice as frequently as processes B and C. To do this, we assign A twice as many tickets. With tickets numbered in the range 0…99, we might assign

```
Process A: 50 tickets (0...49)
Process B: 25 tickets (50...74)
Process C: 25 tickets (75...99)
```

The scheduler then picks a random number in the range 0…100. That number becomes the "winning ticket" and the process holding that ticket gets to run. When its time slice is up, or if it blocks, the sheduler picks another ticket and that process gets to run. Over time, processes will run with a random distribution but one that is weighted by the per-process ticket allocation.

The benefit of the algorithm is that each process is given a proportional share of the CPU. The difficulty is determining ticket distribution, particularly in an environment where processes come and go and get blocked. This isn't a useful algorithm for general-purpose scheduling but is more useful for environments with long-running processes that may need to be allocated shares of CPUs, such as running multiple virtual machines on a server.

Multiprocessors

Our discussions thus far assumed an environment where a single process gets to run at a time. Other ready processes wait until the scheduler switches their context in and gives them a chance to run. With multiple CPUs, multiple cores on one CPU, hyperthreaded processors, more than once thread of execution can be scheduled at a time. The same scheduling algorithms apply; the scheduler simply allows more than one process to be in the running state at one time.

The environment we will consider here is the common symmetric multiprocessing (SMP) one, where each processor has access to the same memory and devices. Each processor is running its own process and may, at any time, invoke a system call, terminate, or be interrupted with a timer interrupt. The scheduler executes on that processor and decides which process should be context switched to run. It is common for the operating system to maintain one run queue per processor. This allows one processor to manipulate the queue (e.g., when context switching) without having to worry about the delay of having the queue locked by another processor.

Processors are designed with cache memory that holds frequently-used regions of memory that processes accessed. This avoids the time delay of going out to the external memory bus to access memory and provides a huge boost to performance. As we will see in our forthcoming discussion on memory management, processors also contain a translation lookaside buffer, or TLB, that stores recent virtual to physical address translations. This also speeds up memory access dramatically.

When a scheduler reschedules a process onto the same processor, there is a chance that some of the cached memory and TLB lines are still present. This allows the process to run faster since it will make less references to main memory. If a scheduler reschedules a process onto a different processor then no part of the process will be present in that processor's cache and the program will start slowly as it populates its cache.

Processor affinity is the aspect of scheduling on a multiprocessor system where the scheduler keeps track of what processor the process ran on previously and attempts to reschedule the process onto that same processor. There are two forms of processor affinity. Hard affinity ensures that a process always gets scheduled onto the same processor. Soft affinity is a best-effort approach. A scheduler will attempt to schedule a process onto the same CPU but in some cases may move the process onto a different processor. The reason for doing this is that, even though there may be an initial performance penalty to start a process on another CPU, it's probably better than having the CPU sit idle with no process to run. The scheduler tries to load balance the CPUs to make sure that they have a sufficient number of tasks in their run queue. There are two approaches to load balancing among processors:

1. Push migration is where the operating system checks the load (number of processes in the run queue) on each processor periodically. If there's an imbalance, some processes will be moved from one processor onto another.

2. Pull migration is where a scheduler finds that there are no more processes in the run queue for the processor. In this case, it raids another processor's run queue and transfers a process onto its own queue so it will have something to run.

It is common to combine both push and pull migration (Linux does it, for example).

Scheduling domains

The real world is not as simple as deciding whether to run a task on the same processor or not. Many systems have multiple processors and some are preferable to others when rescheduling a ready task. The categories of process include:

Virtual CPUs in a hyperthreaded core

> Many intel CPUs support hyperthreading (HT technology). A single processor core presents itself as two virtual CPUs to the operating system. The processor core has a separate set of registers for each virtual CPU and multitple instructions can execute in parallel as long as they don't compete for the same section of the processor. Although one execution thread may hold up the performance of another one, the threads share acces to all processor caches.

Multiple cores in a processor

> Many processors, particularly those on laptops, desktops, and servers, contain several processor cores (often 2, 4, 6, or 8) within a single chip. In this case, the TLB and fastest instruction and data caches are not shared across cores. However, all the cores share access

to a common memory cache. This cache is slower than the high-speed per-core cache but still much faster than accessing main memory.

Multiple processors in an SMP architecture

Multiple processors in one computer system share common access to memory in the system. However, they do not share any caches: one processor does not have access to cached data on another processor.

Multiple processors in an NUMA architecture

NUMA, Non-Uniform Memory Architecture is a multiprocessor computer architecture, designed for large numbers of processors where each processor or group of processors has access to a portion of memory via a high-speed memory interface (e.g., on the same circuit board) while other regions of memory are slower to access since they reside on other processors and are accessed via a secondary, slower, interface.

What we have now is the realization that if a task is to be migrated to another processor, migrating it to some processors is preferable to others. For example, scheduling a task on a different core on the same chip is preferable to scheduling it onto a core on a different chip.

Linux introduces the concept of scheduling domains to handle this. Scheduling domains allow the system to organize processors into a hierarchy, where processors are grouped from the most desirable migration groups at the lowest layer (e.g., hyperthreaded CPUs on the same core) through to the least desirable migration groups at the highest layers of the hierarchy (e.g., migrating across processors on different circuit boards in a NUMA system).

A scheduling domain constains one or more CPU groups. Each CPU group is treated as one entity by the domain. A higher-level domain treats lower-level domains as a group. For example, two hyperthreaded CPUs sharing the same core will be placed in one group that has two subgroups, one for each CPU. All the per-core groups will make up a higher-level domain that represents the entire processor.

Each CPU has a runqueue structure associated with it. In addition to the structures needed to keep track of ready processes (e.g., a balanced tree or a set of queues), this per-cpu structure keeps track of scheduling data, including statistics about CPU load. Each scheduling domain has a balancing policy associated with it that defines the balancing rules for that specific level of the hierarchy. This policy answers questions such as:

– How often should attempts be made to balance load across groups in the domain?

– How far can the loads in the domain get unbalanced before balancing across groups is needed?

– How long can a group in the domain sit idle?

Linux performs active load balancing periodically. The scheduler moves up the scheduling domain hierarchy and checks each groups along the way. If any group is deemed to be out of balance based on the policy rules, tasks will be moved from one CPU's run queue to another's to rebalance the domain.

Scheduling classes

Linux supports a modular scheduling system that can accommodate different schedulers. A scheduling class defines a common set of functions that define the behavior of that scheduler (e.g., add a task, remove a task, choose the next task tor run). Multiple schedulers can run concurrently. Each task in the system belongs to one scheduling class. A task will belong to one of two scheduling classes:

1. sched_fair: implements the CFS (completely fair share) scheduler, a general purpose multilevel queue scheduler that dynamically adjusts priority levels based on how much "virtual runtime" each task used over a period of time. Tasks that spend more time running (using the CPU) are given a lower priority over those that spend spend less time running.
2. sched_rt: implements a simple multilevel priority-based round-robin scheduler for real-time tasks.

To pick a task to run, the scheduler iterates through the list of scheduling classes to find the class with the highest priority that has a runnable task.

References

- Volker Seeker, Process Scheduling in Linux, University of Edinburgh, May 12, 2013.
- Inside the Linux scheduler, M. Tim Jones, IBM developerWorks Technical Library, June 30, 2006
- Understanding the Linux Kernel, Daniel P. Bovet & Marco Cesati, October 2000, Chapter 10: Process Scheduling
- Inside the Linux 2.6 Completely Fair Scheduler: Providing fair access to CPUs since 2.6.23, M. Tim Jones, IBM developerWorks, December 15, 2009
- Completely Fair Scheduler, Wikipedia article

This document is updated from its original version of September 27, 2010.

GMIT

INSTITIUID TEICNEOLAIOCHTA NA GAILLIMHE-MAIGH EO

GALWAY- MAYO INSTITUTE OF TECHNOLOGY

MEMORY MANAGEMENT

MEMORY MANAGEMENT - CACHING

Caching refers to temporarily storing data in high-speed memory for fast access

Very fast storage is very expensive.

- OS manages a hierarchy of storage devices in order to make the best use of resources.

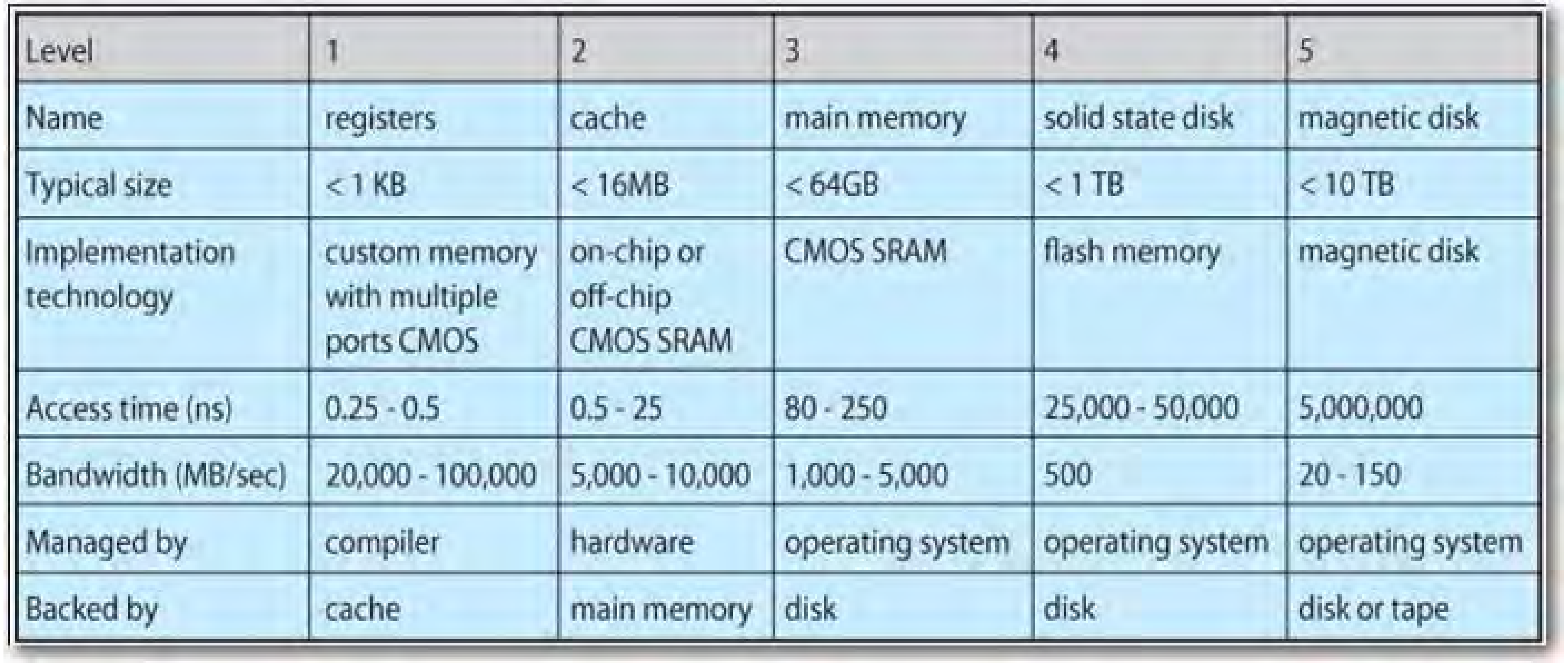
Caching performed at many levels in a computer (in hardware, operating system, software)

Information in use copied from slower to faster storage temporarily

Cache checked first to determine if information is there

- If it is, information used directly from the cache (fast)
- If not, data copied to cache and used there

OS MEMORY MANAGEMENT HIERARCHY



Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

OS MEMORY PROTECTION

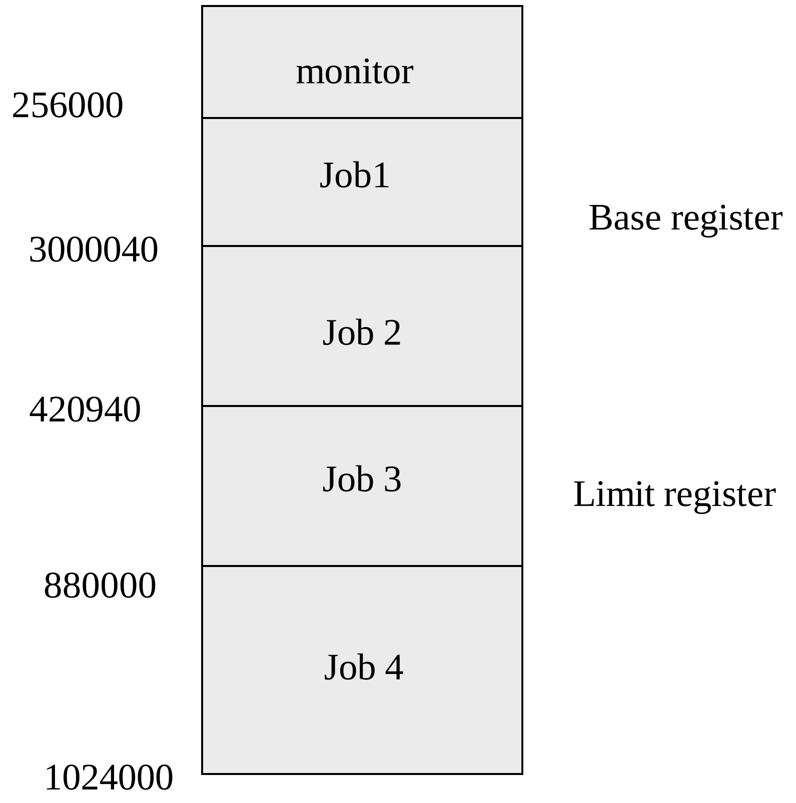
To provide memory protection

- Two registers determine the range of legal addresses a program may access.
  - Base Register - holds smallest legal physical memory address.
  - Limit register - contains the size of the range.

Memory outside the defined range is protected.

Protection against viruses and other malicious programs

Limits specified by OS for each program



OS ADDRESS SPACE (32 VS. 64 BIT)

Managing a program's address space:

Address space ⇒ the set of accessible addresses + state associated with them:

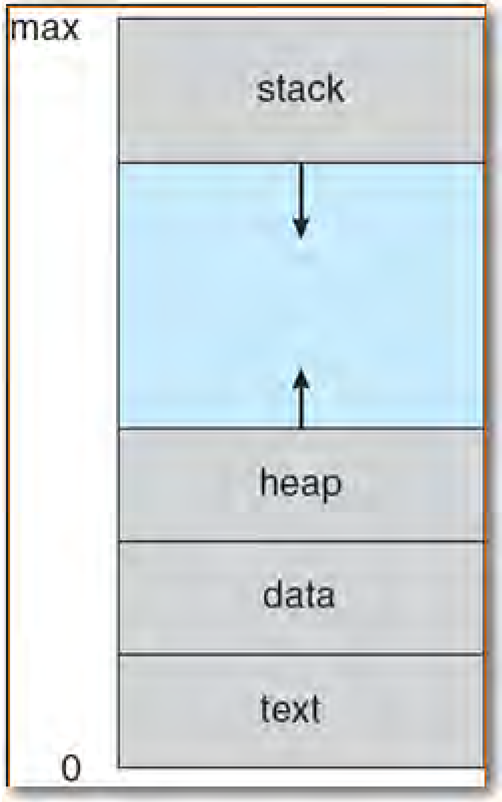
For a 32-bit processor there are $2^{32} = 4$ billion addresses

Each bit in the 32-bit address space represents 1 **byte** in a byte-addressable system.

Only supports 4GB of storage

64-bit avoids this problem – allows for 2^{24} Terabytes of addressable memory

A program's address space provides the illusion of separate address spaces

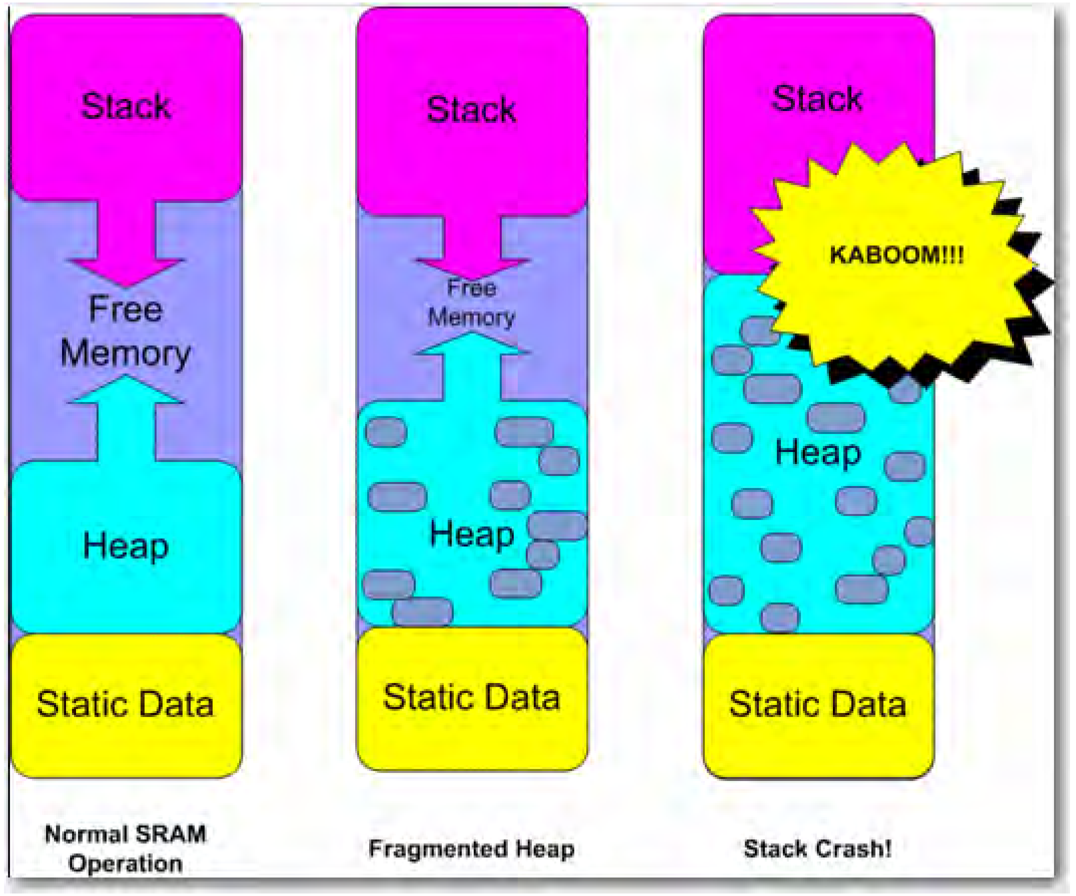


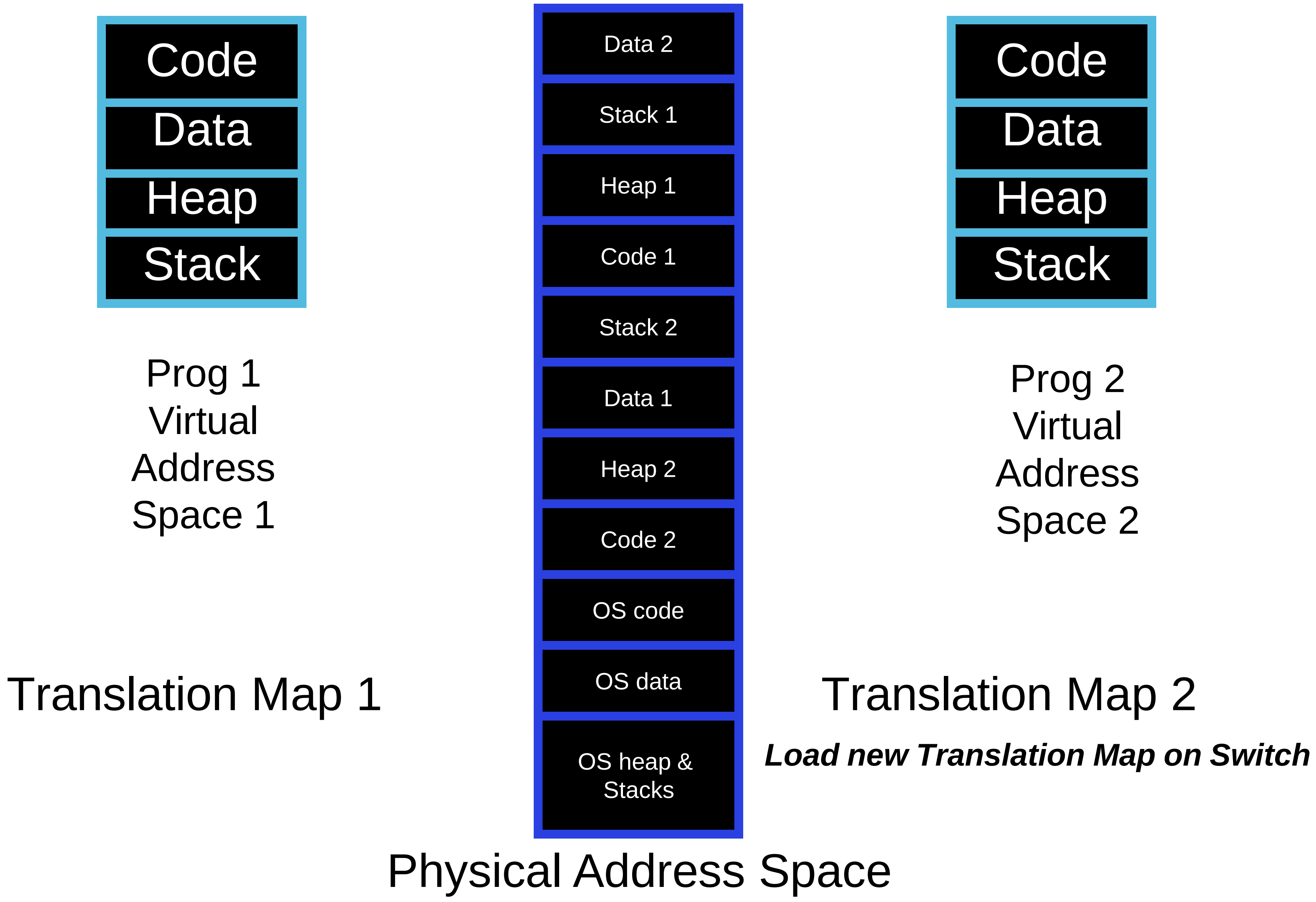
DIFFERENCE BETWEEN HEAP AND STACK

- Stack is used for static memory allocation
  - When function called, fixed space reserved for local variables – cleared after
  - Space allocated using stack – pop on, pop off – model
  - Allocation is dealt with when the program is compiled.
- Heap for dynamic memory allocation
  - Memory allocated and freed on ad-hoc basis, as needed
  - More complex to keep track of which parts are allocated or free at any given tme;
- Both under control of OS and stored in the computer's RAM.
- Access/Write speeds
  - Stack is faster because storage pattern makes it trivial to allocate and deallocate memory from it (a pointer/integer is simply incremented or decremented). Also, each byte in the stack tends to be reused frequently which means it gets mapped to the processor's cache, making it very fast.

Heap requires more complex bookkeeping for allocation or deallocation (slower).

DIFFERENCE BETWEEN HEAP AND STACK




Code
Data
Heap
Stack
Prog 1 Virtual Address Space 1
Data 2
Stack 1
Heap 1
Code 1
Stack 2
Data 1
Heap 2
Code 2
OS code
OS data
OS heap & Stacks
Physical Address Space
Code
Data
Heap
Stack
Prog 2 Virtual Address Space 2
Translation Map 1
Translation Map 2
Load new Translation Map on Switch

GMIT

INSTITIUID TEICNEOLAIOCHTA NA GAILLIMHE-MAIGH EO

GALWAY-MAYO INSTITUTE OF TECHNOLOGY

OS: OTHER SERVICES

OS SERVICES: PROCESS MANAGEMENT

Process - fundamental concept in OS

- Process is a program in execution.
- Process needs resources - CPU time, memory, files/data and I/O devices.

OS is responsible for the following process management activities.

- Process creation and deletion
- Process suspension and resumption
- Process synchronization and interprocess communication
- Process interactions - deadlock detection, avoidance and correction

OS SERVICES: PROTECTION AND SECURITY

Protection mechanisms control access of programs and processes to user and system resources.

- Protect user from themselves, user from other users, system from users.

Protection mechanisms must:

- Distinguish between authorized and unauthorized use.
- Specify access controls to be imposed on use.
- Provide mechanisms for enforcement of access control.
- Security mechanisms provide trust in system and privacy
  - Authentication, certification, encryption etc.

OS SERVICES: NETWORKING

Connecting processors in a distributed system

Distributed System is a collection of processors that do not share memory or a clock.

Processors are connected via a communication network.

Advantages:

- Allows users and system to exchange information
- Provide computational speedup
- Increased reliability and availability of information

OS SERVICES: SYSTEM GENERATION

OS written for a class of machines, must be configured for each specific site.

- SYSGEN (Windows) program obtains info about specific hardware configuration and creates version of OS for hardware

Booting

Bootstrap program - loader program loads kernel, kernel loads rest of OS.

- Bootstrap program stored in ROM/FLASH

SYSTEM PROGRAMS

OS- convenient environment for program development and execution.

User view of OS is defined by system programs, not system calls.

- Command Interpreter (Unix shell) - parses/executes other system programs
- File manipulation - copy (cp), print (lpr), compare(cmp, diff)
- File modification - editing (vim, emacs, notepad++)
- Application programs - send mail (mail), read news (rn)
- Programming language support (gcc - compiler)

OS COMMAND INTERPRETER SYSTEM (CIS) - DESCRIPTION AND ROLE

Part of an OS that understands and executes commands that are entered interactively by a human user or from another program.

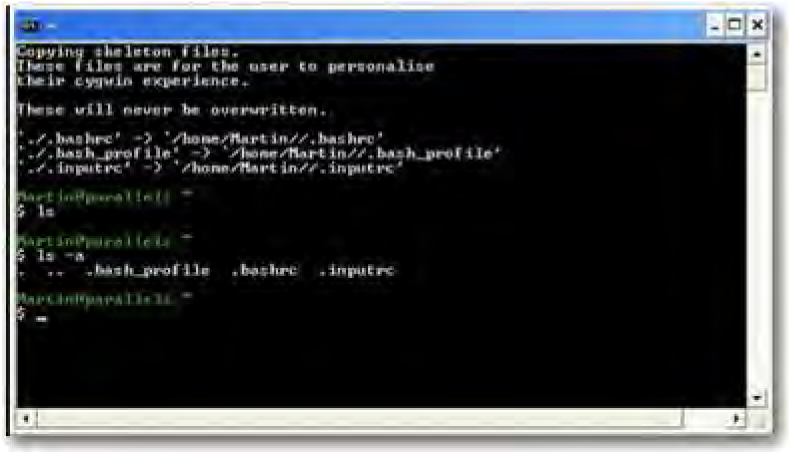
Commands given to OS via command statements that execute

- Process creation and deletion, I/O handling, Secondary Storage Management, Main Memory Management, File System Access, Protection, Networking, etc.

Obtains the next command and executes it.

Also know as -

- Command-line interpreter (cmd), shell (in UNIX/Linux – e.g. bash)



```
Copying skeleton files.
These files are for the user to personalise
their cygwin experience.

These will never be overwritten.

`./.bashrc' -> `/home/Martin//.bashrc'
`./.bash_profile' -> `/home/Martin//.bash_profile'
`./.inputrc' -> `/home/Martin//.inputrc'

Martin@parallel1 ~
$ ls

Martin@parallel1 ~
$ ls -a
.  ..  .bash_profile  .bashrc  .inputrc

Martin@parallel1 ~
$ _
```

GMIT

INSTITIUID TEICNEOLAIOCHTA NA GAILLIMHE-MAIGH EO

GALWAY- MAYO INSTITUTE OF TECHNOLOGY

OPERATING SYSTEMS: HOW ARE THEY ORGANIZED?

Simple

Only one or two levels of code

Layered

Lower levels independent of upper levels

Microkernel

OS built from many user-level processes

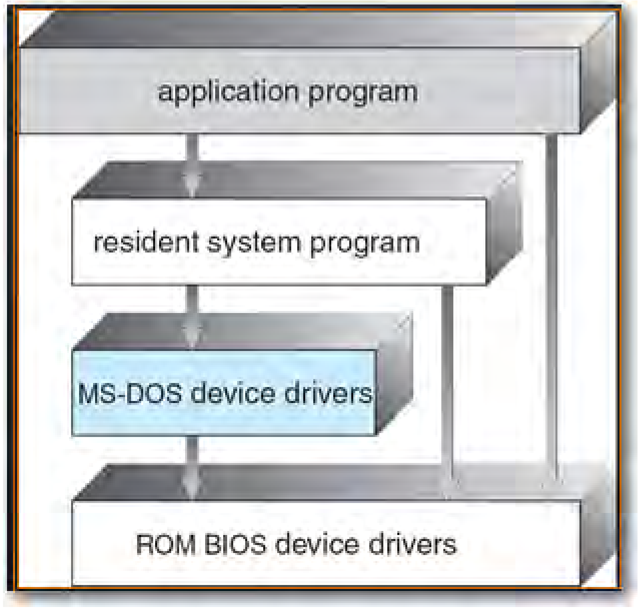
Modular

Core kernel with Dynamically loadable modules

OS STRUCTURE - SIMPLE APPROACH

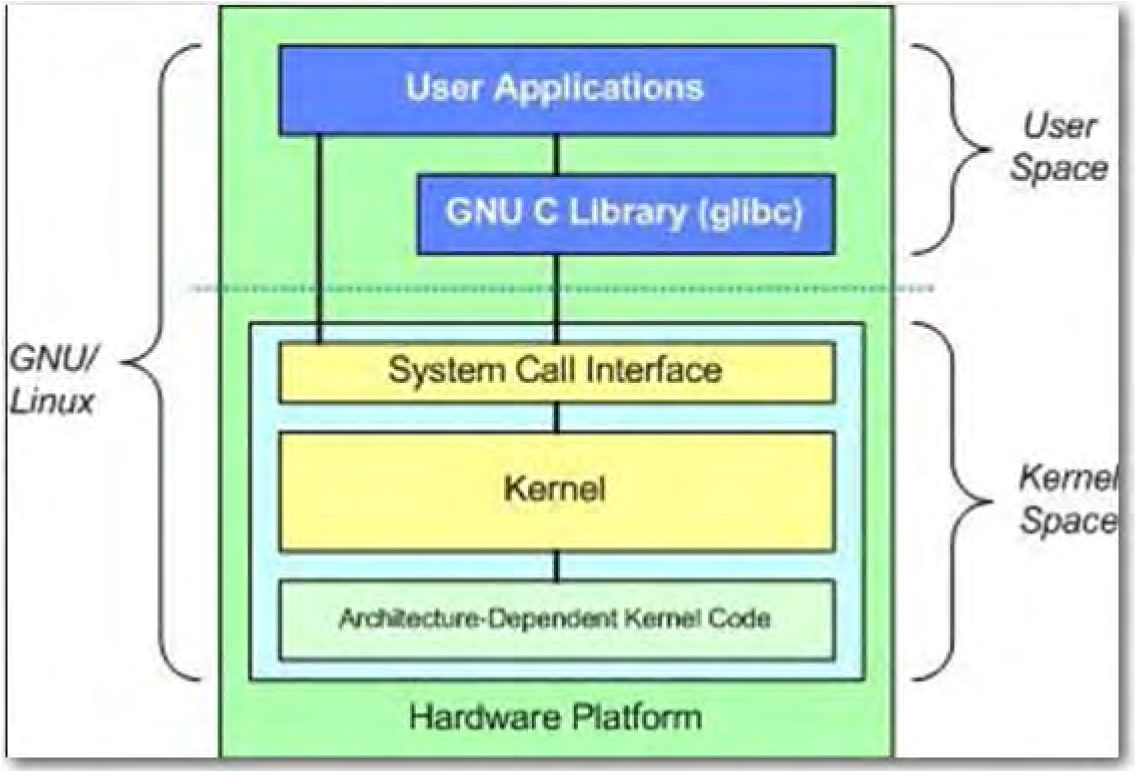
MS-DOS - provides a lot of functionality in little space.

Not divided into modules, Interfaces and levels of functionality are not well separated

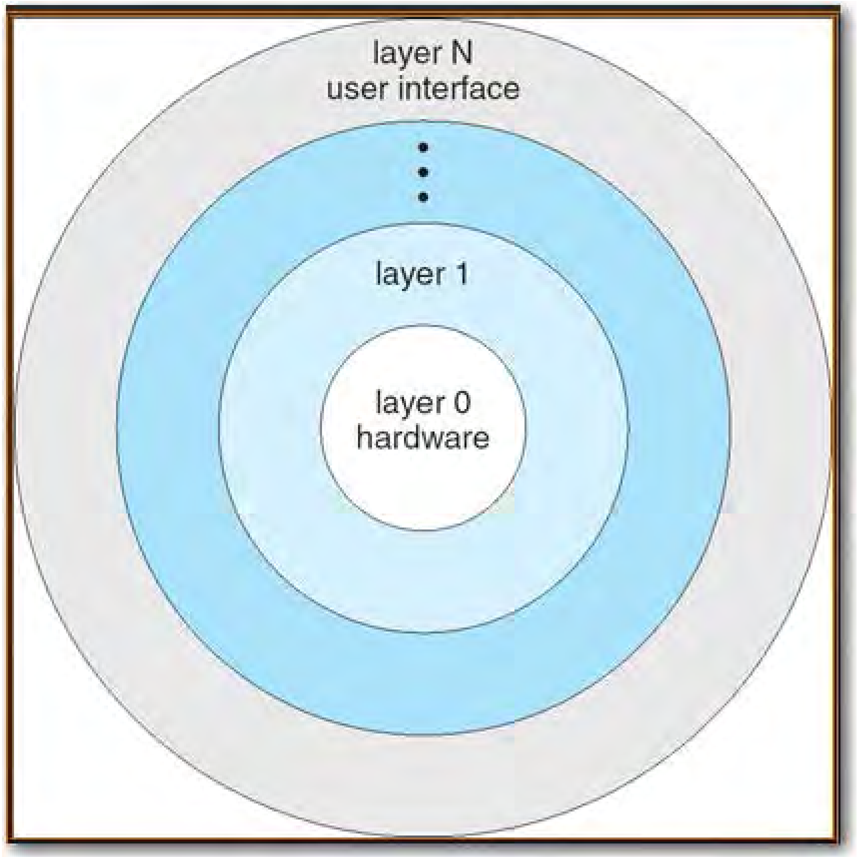


UNIX SYSTEM STRUCTURE

- UNIX - limited structuring, has 2 separable parts
  - Systems programs
  - Kernel
    - everything below system call interface and above physical hardware.
    - Filesystem, CPU scheduling, memory management



LAYERED OPERATING SYSTEM



MICROKERNEL STRUCTURE

Moves as much from the kernel into "*user*" space

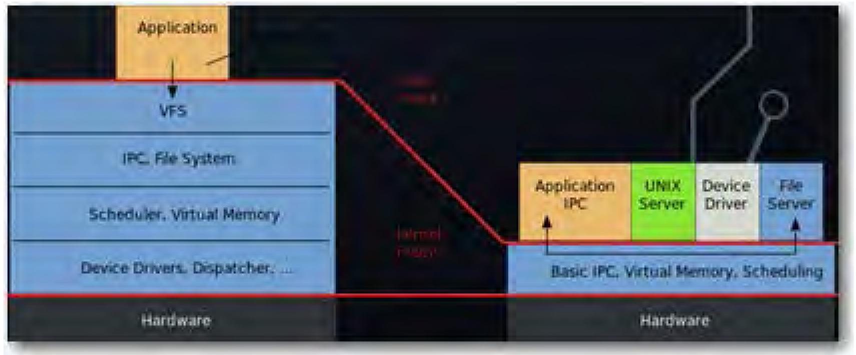
- Small core OS running at kernel level
- OS Services built from many independent user-level processes

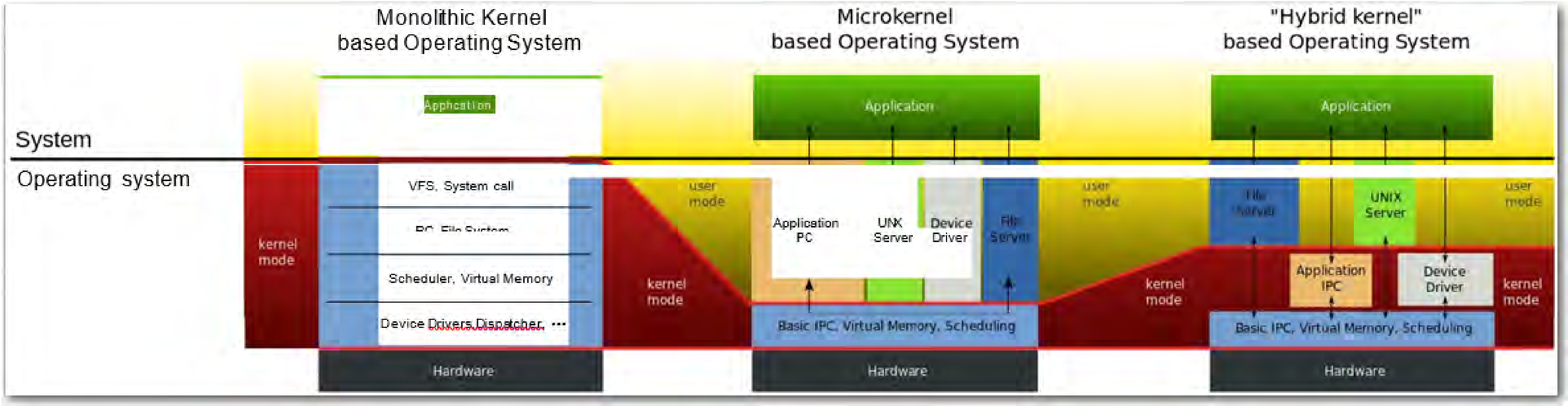
Communication between modules with message passing

Benefits:

- Easier to extend a microkernel
- Easier to port OS to new architectures
- More reliable (less code is running in kernel mode) - Servers
- Fault Isolation (parts of kernel protected from other parts)
- More secure
- No Graphics necessary

< 10,000 lines of code e.g. Symbian, Mac OSX




Monolithic Kernel based Operating System
Microkernel based Operating System
"Hybrid kernel" based Operating System
System
Operating system
Application
VFS, System call
IPC, File System
Scheduler, Virtual Memory
Device Drivers Dispatcher, ...
kernel mode
Hardware
Application
user mode
Application IPC
UNIX Server
Device Driver
File Server
Basic IPC, Virtual Memory, Scheduling
kernel mode
Hardware
Application
user mode
File Server
UNIX Server
Application IPC
Device Driver
Basic IPC, Virtual Memory, Scheduling
kernel mode
Hardware

VIRTUAL MACHINE (VM)

A **virtual machine** (**VM**) is an operating system OS or application environment that is installed on software which imitates dedicated hardware.

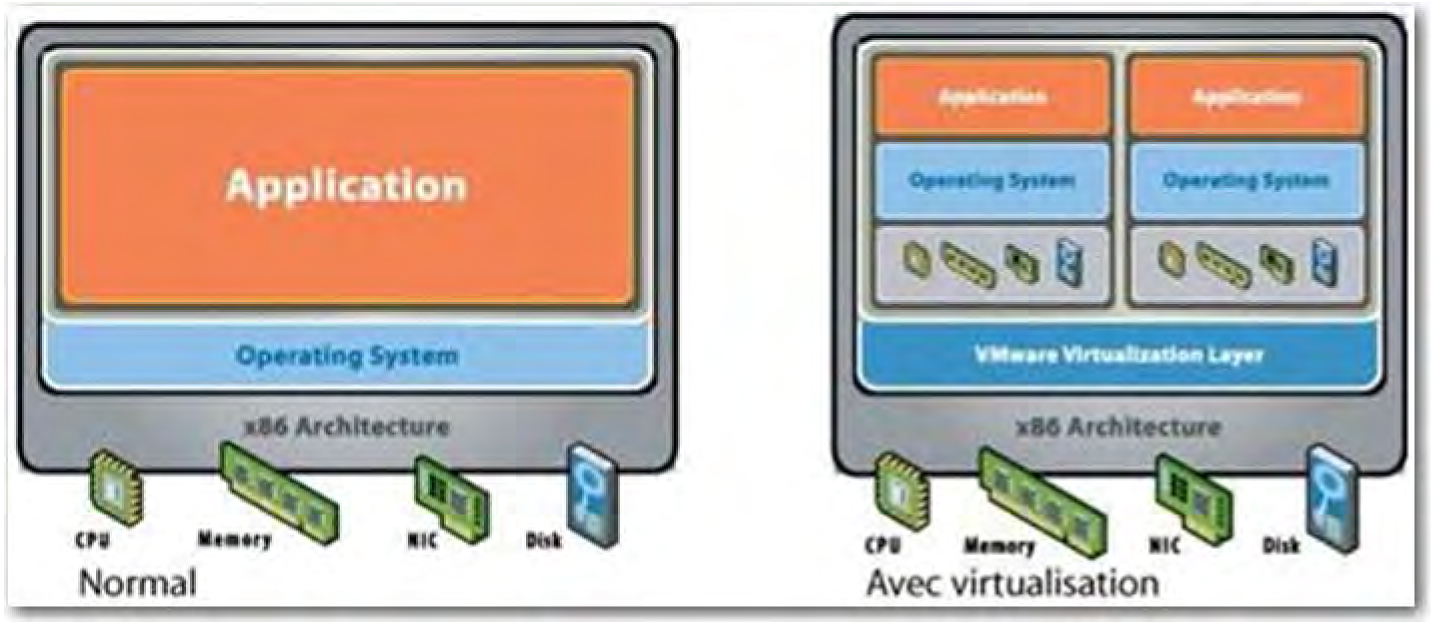
Provides a complete system platform which supports the execution of a complete operating system (OS).

E.g. Ubuntu can be installed as virtual machine (using VMWare software)

Big players:

- https://cloud.google.com/compute/
- https://azure.microsoft.com/
- https://aws.amazon.com/ec2/

VIRTUAL MACHINE (VM)



GMIT

INSTITIUID TEICNEOLAIOCHTA NA GAILLIMHE-MAIGH EO

GALWAY- MAYO INSTITUTE OF TECHNOLOGY

INTRODUCTION TO THE LINUX SYSTEM

REFERENCES

Online

The Linux Documentation Project (LDP): http://www.tldp.org/

Linux books (library)

UNIX/LINUX OPERATING SYSTEM

Introduction to Unix

History of UNIX

What is LINUX

LINUX Distributions

Unix OS Structure

Unix File System

Unix Directories, Files and Inodes

Users, Groups and Permissions

Those who do not understand UNIX are condemned to reinvent it, poorly.

— Henry Spencer

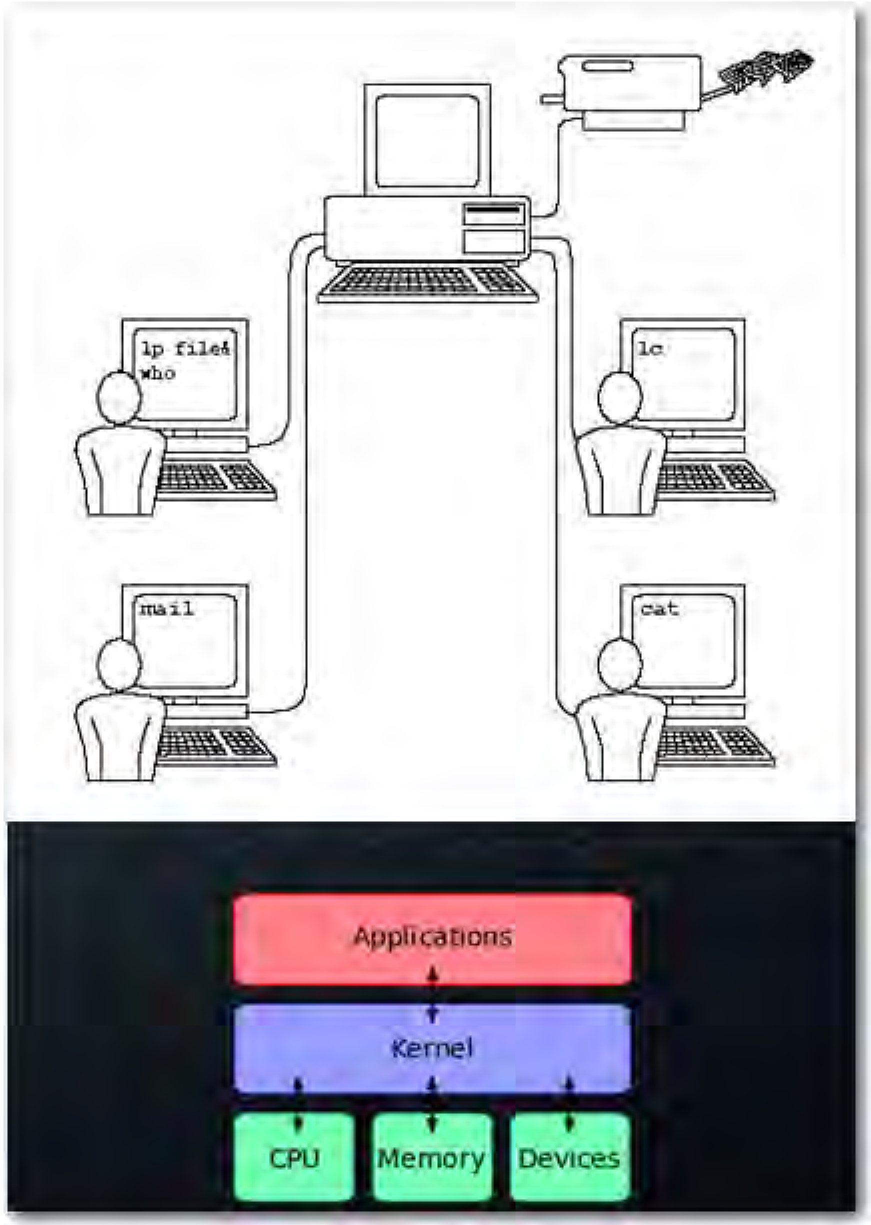
UNIX

Unix is a multi-user, multi-tasking operating system.

You can have many users logged into a system simultaneously, each running many programs.

Kernel's job to:

- Keep each process and user separate
- Regulate access to system hardware, including CPU, memory, disk and other I/O devices.



HISTORY OF UNIX (BELL LABS)

First Version was created in Bell Labs in 1969.

Founded by Alexander Graham Bell

Bell labs inventions:

- Radio astronomy
- Digital Signal Processor – Mobile Phone Tech.
- Solar Cells
- Data Networking
- Communication Satellites
- The transistor
- The laser
- The charge-coupled device (CCD)
- Information theory
- The UNIX operating system
- The programming languages C, C++
- Transatlantic Telephone Cable

8 Nobel prizes

HISTORY OF UNIX

Some of the Bell Labs programmers who had worked on this project, Ken Thompson, Dennis Ritchie, Rudd Canaday, and Doug McIlroy designed and implemented the first version of the Unix File System on a PDP-7 along with a few utilities.

Given the name UNIX by Brian Kernighan.

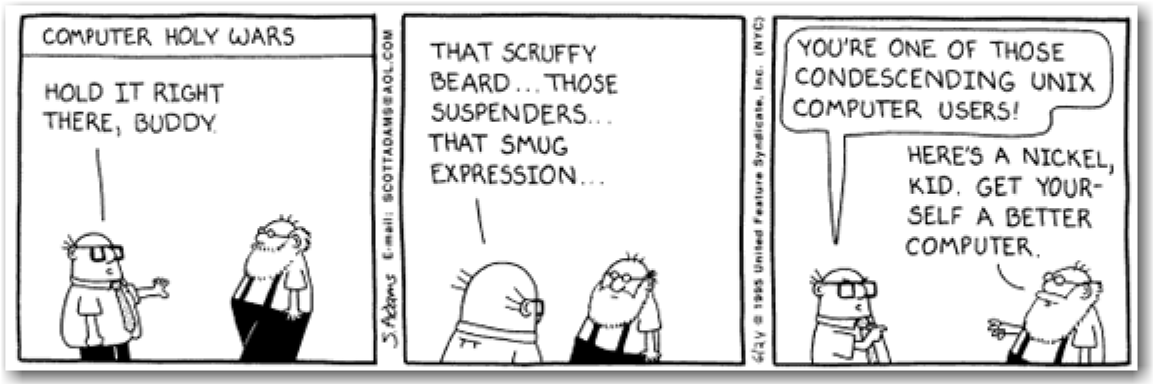
00:00:00 Hours, Jan 1, 1970 is time zero for UNIX. It is also called as epoch.

HISTORY OF UNIX

1973 Unix is re-written mostly in C, a new language developed by Dennis Ritchie.

Being written in this high-level language greatly decreased the effort needed to port it to new machines.

HISTORY OF UNIX

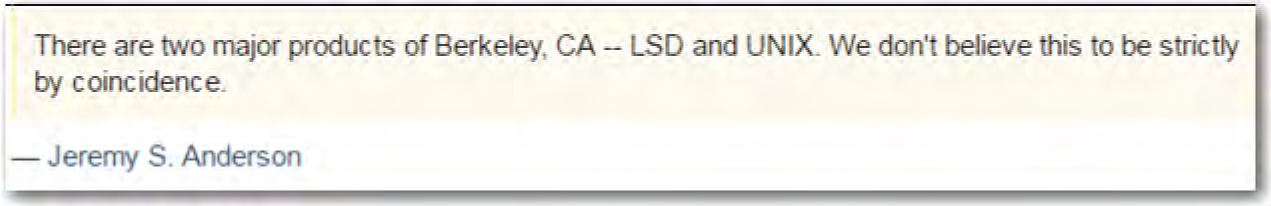

COMPUTER HOLY WARS
HOLD IT RIGHT THERE, BUDDY.
THAT SCRUFFY BEARD... THOSE SUSPENDERS... THAT SMUG EXPRESSION...
YOU'RE ONE OF THOSE CONDESCENDING UNIX COMPUTER USERS!
HERE'S A NICKEL, KID. GET YOURSELF A BETTER COMPUTER.

1977 There were about 500 Unix sites world-wide.

1980 BSD 4.1 (Berkeley Software Distribution)

1983 SunOS, BSD 4.2, System V

1991 Linux was originated.



There are two major products of Berkeley, CA -- LSD and UNIX. We don't believe this to be strictly by coincidence.

— Jeremy S. Anderson

WHAT IS LINUX

Linux is a free Unix-type OS originally created by Linus Torvalds with the assistance of developers around the world.

It originated in 1991 as a personal project of Linus Torvalds, a Finnish graduate student.

The Kernel version 1.0 was released in 1994 and today the most recent stable version is 4.4.1

Developed under the GNU General Public License , the source code for Linux is freely available to everyone.

LINUX DISTRIBUTIONS

Mandrake: http://www.mandrakesoft.com/

RedHat: http://www.redhat.com/

Fedora: http://fedora.redhat.com/

SuSE/Novell: http://www.suse.com/

Debian: http://www.debian.org/

Ubuntu: http://www.ubuntu.com

Red Hat Enterprise Linux is a Enterprise targeted Operating System. It based on mature Open Source technology and available at a cost with one year Red Hat Network subscription for upgrade and support contract

LINUX SHELL

LINUX DESKTOP

Text Mode and Graphical Mode

Multiple Non-GUI (Text Mode) logins are possible through Virtual Consoles.

There are by default 6 Virtual Text Mode Consoles available through CTRL-ALT-F[1-6]. CTRL-ALT-F7 will bring back the GUI mode.

In GUI Mode, there are two Desktop Environments

- GNOME
- KDE

GNOME is the default Desktop

UNIX/LINUX COMMANDS

A command is a program which interacts with the kernel to perform the functions called for by the user.

A command can be: a built-in shell command; an executable shell file (shell script); or source compiled code.

The shell is a command line interpreter. The user interacts with the kernel through the shell.

- Can write ASCII (text) scripts to be acted upon by a shell.

UNIX/LINUX SHELL – ROLE AND DESCRIPTION

- Command Interpreter
  - The shell sits between you and the operating system, acting as a command interpreter – shell scripting: Turing complete
  - It reads terminal input and translates the commands into OS actions.
    - Analogous to cmd in DOS.
- When the shell starts up it reads its startup files and may set environment variables, command search paths, and command aliases, and executes any commands specified in these files.
  - etc.init.d

SELECTION OF UNIX/LINUX SHELLS

When you log into the system you are given a default shell.

The original shell was the Bourne shell – '*sh*'

Every Unix platform will either have the Bourne shell, or a Bourne compatible shell available.

The default prompt for the Bourne shell is $ (or #, for the root user).

Another popular shell is C Shell. The default prompt for the C shell is %.

"echo $0" will list what shell you are using

SELECTION OF UNIX/LINUX SHELLS

- Numerous other shells are available from the network.
  - Almost all of them are based on either sh or csh with extensions to:
    - Allow in-line editing of commands
    - Page through previously executed commands (history),
    - Provide command name completion and custom prompt, etc.
- Some of the more well known of these may be on your Linux system:
  - Korn shell, ksh, by David Korn
  - Bourne Again Shell - bash, from the Free Software Foundations GNU project,
    - both Korn and bash based on sh,
  - T-C shell: tcsh and the extended C shell: cshe, both based on csh.

File Commands

ls - directory listing
ls -al - formatted llsting with hidden files
cd *dir* .change directory *todir*
cd - change to home
pwd - show current directory
llkdir *dir* - create a directory *dir*
nt *file* - delete *file*
nt -r *dir* - delete directory *dlr*
nt -f *file* - force remove *file*
nt -rf *dir* - force remove directory *dlr* •
cp *filel file2* - copy *filel* to *file2*
cp -r *dirl dir2* - copy *dirl* to *dir2;* create *dlr2* if it doesn't exist
av *filel file2* - reoame or *movefilel tofile2*
if *file2* is an existing directory, moves *filel* into directory *file2*
ln _,.. *file link* - ,.IP. mhnlit: link *llttJt* tn *filP*
touch *file* - create or update *file*
cat > *file* - places standard input Into *fi•e*
110re *file* - output the contents of *file*
head *file* - output the first 10 lines of *file*
tail *file* - output the last 10 lines of *file*
tail -f *file* - output the contents of *file* as il grows, starting with the last 1Olines

Process Mana ement

pc; - cii"fll.Ay ynn r f":n n-P.ntJy .Ar:tivA J'UYU":ASAA
top - display all runcing processes
kill *pid* - kill process id *pld*
killall *proc* - kill llprocesses named *proc* •
bg - llsts stopped or 3ackground jobs; resume a stopped job in the background
fg - brings the most recent job to foregrocnd
fg *n* - bringsjob *n* to the foreground

File Permissions

ehiand net-al file -t:hA n!JA thA j"IArmisslnn.s *nfj11P* to *octnl*, which can ha found separately for user, group, and world by adding:

- 4 - read (r)
- 2 - write (w)
- 1- execute (x)

m

ch11<>d 777 - read, write, execute for all
ch11<>d 755 - cwx for owner, rx for group and world

fːor nıbre optioos·see aan Cl:aod.

ssh user@llost - connect *to host* as *user*
ssh -p port *user@host* - connect *to hosl* on port port as *user*
ssh-copy-id *user@host* - add your key *to host* for *user* to enable a keyed or passwonlless login

Searchin

grep *pattern files* - search for *pattern mfiles*
grep -r *pattern dir* - search recursively for *pattern* in *dir*
c.,...nd | grep *pattern* - search for *pattern* lo the output of *command*
locate *file* - find all instances of f11e

S stemhfo

date - show the current date and time
cat - shOIV this month's calendar
uptiae - show current uptime
w - display who is online
whoaai - who you are logged in as
finger *user* - display information about *user*
unaae -a - show kernel information
cat /proc/cpuinto - cpu mtormabon
cat /proc/aeainfo - memory information
un c nd - show the manual for *command*
df - show disk usage
du - show directory space usage
free - show memory and swap usage
whereis •*PP* - show possible locations of *app*
which app - show which *app* *w>ll* be run by default

Com ression

tar cf file.*tar files* - create a tar named *file.tor* CO>taining *files*
tar xf file.*tar* - extract the files from ji1e.tor
tar czf file.*tar*.gz *files* - create a tar with Gzip compression
tar xzf *file*.tar.gz - extract a tar using Gzip
tar cjf *file.tar.bz2* - create a tar with Bzip2 compression
tar xjf *file.tar*.bz2 - extract a tar using Bzip2
gzip *file* compro33CS *file* and renames it to *file.gz*
gzip d *file.gz* - decompresses *file.gz* back to *file*

Network

ping *host* - ping *host* and output results
whois *d0111ain* - get whois information for *domain*
dig *domaln* - get DNS information for *domain*
dig -x *host* - reverse lookup *ast*
wget *tile* - download *file*
wget C *file* - continue a stopped download

hstallation

Install from source:
./configu re
H
uke install
dpkg •i *pkg.deb* _ install a package (Debian)
•ı rpa Uvh *pkg.* r,,. - install a package (RPM)

Shortcuts

Ctrt+C - baits the current command
Ctrt+Z - stops the current command, resume with fg lo the foreground or bg in the background
Ctrl+D - bg out or current seion, similar *to* exit
Ctrt+ll - erases one word in the current line
Ctrl+U - erases the whole line
Ctrl+R - type to bring up a recent command
11 - repeats the last command
exit - log out of current sessio'

• use \\lith extreme caution.

l@ © @j

LINUX COMMAND HIGH-LEVEL FUNCTIONS

File Management and Viewing

Filesystem Mangement

Help, Job and Process Management

Network Management

System Management

User Management

Printing and Programming

Document Preparation

Miscellaneous

COMMAND STRUCTURE

Command <Options> <Arguments>

Multiple commands separated by ; can be executed one after the other

HELP FACILITIES FOR COMMANDS

To understand the working of the command and possible options use (man command)

Using the GNU Info System (info, info command)

Listing a Description of a Program (whatis command)

Many tools have a long–style option, `––help', that outputs usage information about the tool, including the options and arguments the tool takes.

Ex: whoami --help

GMIT

INSTITIUID TEICNEOLAIOCHTA NA GAILLIMHE-MAIGH EO

GALWAY- MAYO INSTITUTE OF TECHNOLOGY