# FIT1047 - Week 6

## Operating Systems

An Operating System (OS) is a piece of software (a program) that manages the resources in a computer and provides a convenient interface between application programs and the hardware.

## Important people

Ken Thompson and Dennis Ritchie, creators of the original UNIX operating system. UNIX was written in the C programming language, which was also developed by Ritchie (together with Brian Kernighan).

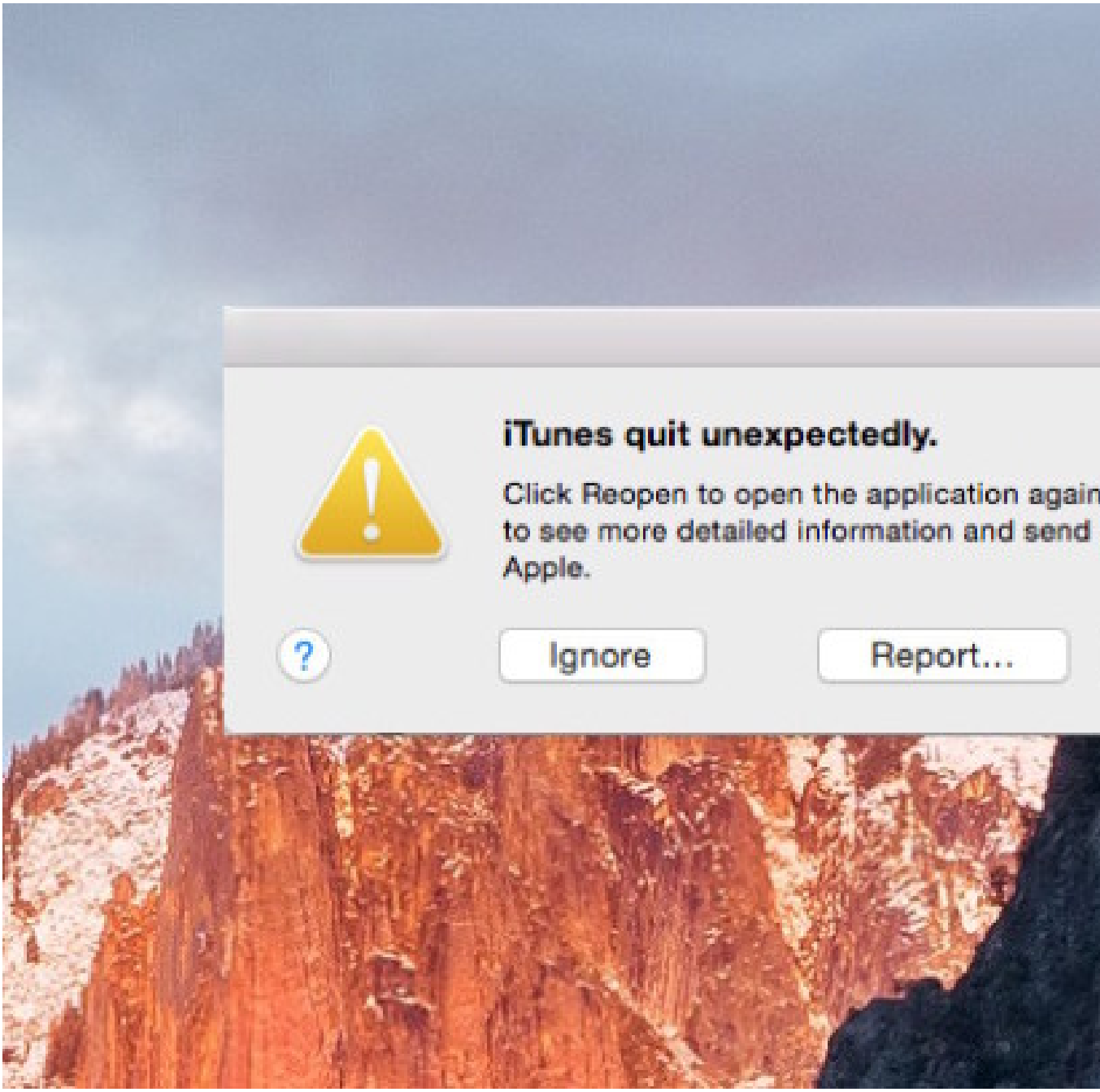
Bill Gates, founder of Microsoft, which sold the original MS-DOS operating system for IBM-compatible PCs, and later the line of Windows operating systems.

Steve Jobs and Steve Wozniak, founders of Apple.

Linus Torvalds, developed the Linux operating system in 1991.

## What does an OS do?


Files
Wikipedia - Mozilla Firefox
Wikipedia
https://www.wikipedia.org
Search
WIKIPEDIA
English
The Free Encyclopedia
4 993 000+ articles
Español
La enciclopedia libre
1 208 000+ artículos
Deutsch
Die freie Enzyklopädie
1 867 000+ Artikel
Русский
Свободная энциклопедия
1 262 000+ статей
中文
自由的百科全書
845 000+ 條目
Português
A enciclopédia livre
892 000+ artigos
English
Home
Recent
Home
Desktop
Documents
Downloads
Music
Pictures
Videos
Trash
Computer



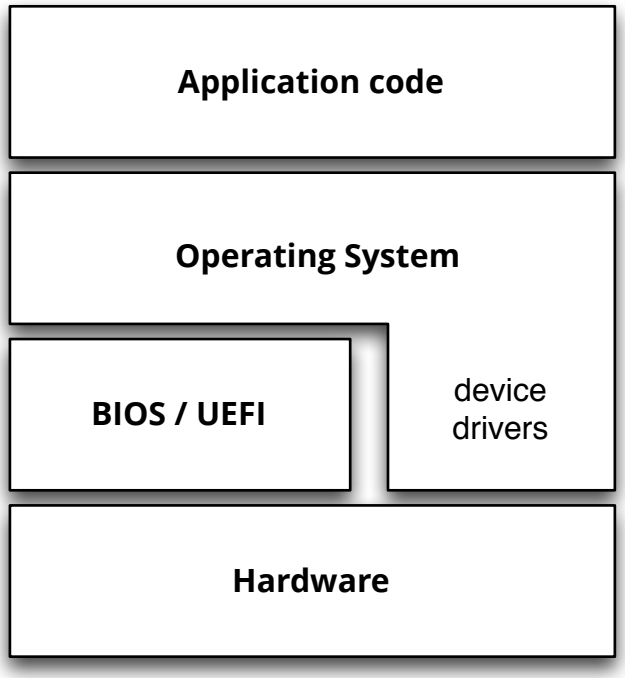

Most people think the OS is things like the Windows menu, maybe the web browser, maybe the file system explorer.

Many people think the OS provides the graphical user interface (GUI).

**But the core functionality is actually much more fundamental: multiple processes are running in parallel,** the OS provides file system, network access, crash reports.

## What does an OS do?

An OS is a **level of abstraction** between hardware and software.




## Overview

What does an OS do?

- Process management (a process is a running program)
- Memory management
- I/O

(it does more but that's what we'll cover)

The notion of *process* is quite important, and you need to know what the difference is between a process and a program.

- A program is the code that you write, the sequence of instructions (and possibly data).
- A process is an *instance* of the program that is currently being executed by a computer.

An important difference is that there can be multiple processes executing the same program, e.g. some web browsers start a process for each open window.

## Free textbook

There's a free textbook, available from http://www.ostep.org/

Don't worry about the level of detail, we'll only use small parts:

- Section 2.6: History
- Chapters 3, 4.1, 4.4, 6: Processes
- Chapters 12, 13, 15: Memory

This book assumes a working knowledge of the C programming language for some of the topics, and it uses example code written in C. For the purpose of this unit, just skip those parts - we are only interested in the fundamental concepts, not in the implementation details.

## A bit of history

First OSs:

- just a library
- single program running at a time
- no protection (e.g. any program could read/write entire disk!)

Multiprogramming means running multiple programs at the same time.

When writing a program for one of the first computers, you could use a library of existing code for useful functions such as input/output. The computer would then be loaded with your program (which included the library code), usually by entering punched cards or paper tapes, and execute it until it finished (or your allocated time was up).

## A bit of history

Multiprogramming:

- OS can run multiple processes "simultaneously"
- improves CPU utilisation
- important when computers were very expensive

Required protection:

- protect memory of other processes
- protect files of other users

Gave rise to UNIX family of OSs

## A bit of history

Modern OSs:

- hundreds of processes running at any point in time
- provide access to diverse hardware
- full network support built-in
- are somewhat related to UNIX

Many early operating systems for personal computers and home computers did not come with any networking support. Special OS versions existed that included network drivers (and had to be paid for!).

Mac OS and Linux are both closely related to UNIX. Windows is less closely related but has used many of the same principles since Windows NT, and in particular since Windows 2000.

## Goal: ease of use.

- For end users:
  - provide consistent *user interface*
  - manage *multiple applications* simultaneously
  - *protect* users from malicious or buggy code
- For programmers:
  - provide *programming interface*
  - enable access to hardware and I/O
  - manage system resources (memory, disk, network)

Introduce a **level of abstraction** between the computer and the application.

## Abstraction

The most important concept in IT!

- hide **complexity** from users
- provide clean, well-defined **interface** to functionality

Simple example: your multiplication subroutine

- hides details of implementation behind simple interface

## Abstraction in OSs

Virtualisation:

- Provide *virtual* form of each *physical resource* for each process.

This means you can code as if your program

- has the entire CPU to itself
- has a large, contiguous memory just for itself
- can use system resources through library functions (e.g. keyboard, graphics, disk, network)

Example:

On a normal day, let's say you leave the house at 7:30am, drive to work, park your car at 8:10am, leave work at 5pm, pick up your car from the car park and drive home, arriving at 5:45pm, where the car sits in the garage until the next morning.

In a car sharing scenario, your car could be somewhere else while you're not using it, e.g. someone could use it during the day to visit clients, and someone else working night shifts could have it from 6pm till 7am.

Let's assume we can make sure that whenever you and the other two car users need the car, it is right where you and they expect it to be. Then we have turned the one *physical* car into three *virtual* cars.

Operating systems do the same, sharing the scarce physical resources among multiple processes.

But it goes further: imagine everyone could leave some stuff in the glove compartment, but they always only see their own stuff! And the fuel gauge is always at the level where you left the car. Operating systems isolate processes against each other, preventing them from accessing each others' resources, such as memory and files on disk.

## The OS kernel

Modern OSs have many different functions.

We're only looking at the *core* functionality.

The part of the OS that implements this is called the **kernel**.

## Virtualising the CPU

How to run many processes on a single CPU?

Timesharing!

- OS kernel switches between processes
- If switching is fast and occurs often, creates illusion of *concurrency*
- Illusion for **both user and programmer**!

For the user, e.g. playing some music in the background while scrolling through a web page should sound and look convincing (as if the two processes are actually running simultaneously).

For the programmer it should look exactly like having exclusive access to the CPU. The programmer should not have to change their program depending on what other programs are running.

## Managing processes

Mechanisms: *how* to virtualise the CPU

- rest of today's lecture

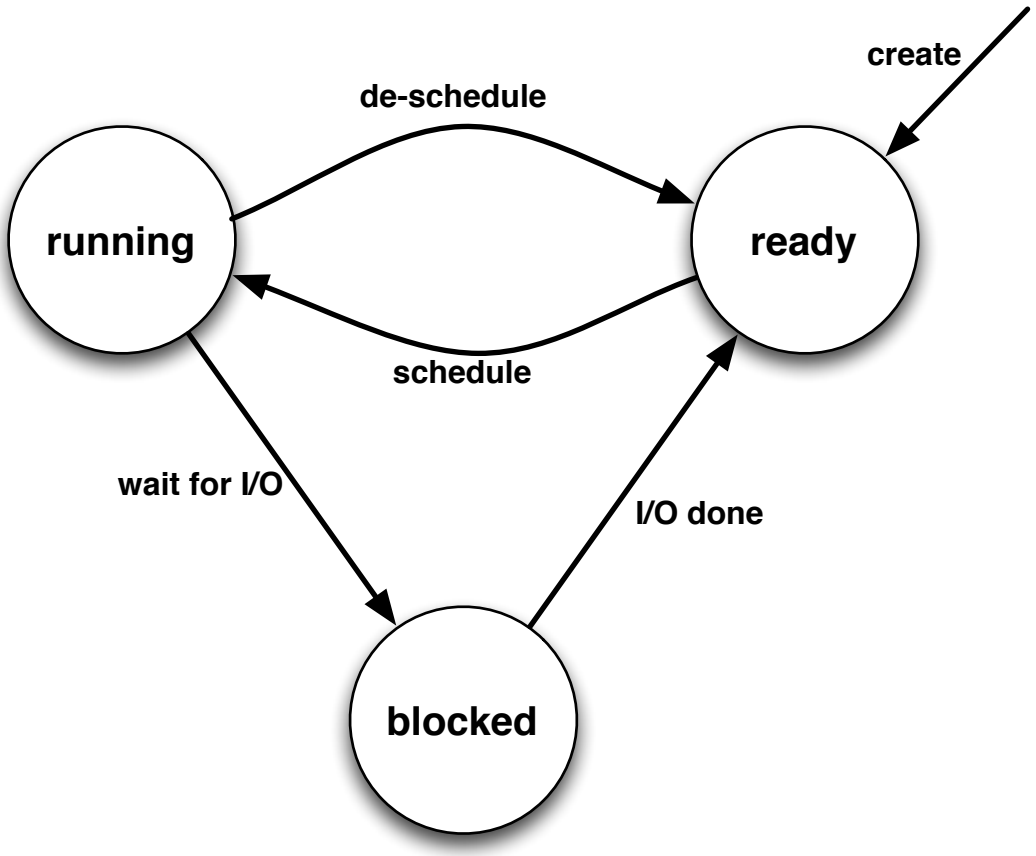
Policies: *when* to switch

- next part tomorrow

## Processes

## Processes

- Created by loading code into memory
- Can be in one of three states:




## Process states

| Time | Media player (MP) | Web browser (WB) | description |
|---|---|---|---|
| 1 | Running | Ready | |

... continued on next page

| Time | Media player (MP) | Web browser (WB) | description |
|---|---|---|---|
| 2 | Running | Ready | |
| 3 | Running | Ready | MP initiates I/O |
| 4 | Blocked | Running | switch to WB |
| 5 | Blocked | Running | |
| 6 | Ready | Running | I/O done |
| 7 | Ready | Running | switch back to MP |
| 8 | Running | Ready | |
| 9 | Running | Ready | |
| 10 | Ready | Running | switch back to WB |

## Challenges

Performance:

- CPU virtualisation should not create huge overhead

Control:

- OS must stay in control
- Enable fair scheduling
- Protect against malicious or buggy code

Requires hardware support!

## Limited Direct Execution

Application code runs directly on CPU.

So clearly, as long as it runs, the OS doesn't run!

Two problems:

1. How to restrict what the program can do without affecting efficiency?
2. How to stop a process and switch to another process?

## Restricting what programs can do

## Restricted Operations

Solution: CPUs have different *modes!*

**Kernel mode:** Code is run without any restrictions. The OS runs in kernel mode. Interrupts trigger switch to kernel mode.

**User mode:** Only a limited subset of instructions can be used. E.g. no I/O instructions. Normal applications run in user mode.

Without I/O instructions, a process cannot simply access arbitrary parts of the hard disk, or send data over the network, or draw to the screen, or record sound from the microphone.

Kernel mode is also sometimes called *supervisor* mode.

**Problem:** how can a user mode process do I/O?

## System calls

Special CPU instructions that expose OS functionality to user programs, e.g.

- perform file I/O
- access the network interface
- communicate with other processes
- allocate memory (we'll talk about that later)

## System calls

OS sets up a **table** of system call handlers.

When process makes a system call number *n*,

- save process context in memory
- switch CPU to kernel mode
- jump to call handler *n* and execute
- restore process context
- switch CPU to user mode

- return to calling process

Looks familiar?

## Software Interrupts

Recap on interrupts:

- Hardware triggers flag in CPU
- CPU jumps to special code and then returns to running program
- Context switch makes sure program can continue as if no interrupt had happened

To implement system calls:

Add an **instruction** that causes a special interrupt!

Also called *software interrupts*.

## Summary: Limited Direct Execution

- Application code runs directly on CPU **but we need to restrict what it can do**
- CPU has *modes*:
  - kernel mode (no restrictions)
  - user mode (some instructions are not allowed)
- *System calls* let user mode code call OS functions

## Process switching

## Process Switching

Remember that the problem is that the user process is now running on the CPU. But to create the illusion of multiple processes running simultaneously, the OS needs to switch between them in regular intervals.

Since the OS is not running right now, how can it do the switching?

How can the OS regain control over a running process?

Solution: **interrupts** switch from user code to kernel.

## Context switching

Recap from week 4:

- Save all process state to memory
  - mainly CPU registers
- Execute interrupt handler
- Restore process state from memory
- Return to process

Only difference with process switching: **don't return to the same process!**

## Process Switching

How can the OS regain control over a running process?

Solution: **interrupts** switch from user code to kernel.

Two variants:

**Cooperative:** wait for a **system call**.

**Preemptive:** wait for a **timer interrupt**.

## Cooperative timesharing

OS regains control when user mode process makes a **system call**.

- OS then checks whether to switch from process A to B
  - if no, just handle system call and return to process A
  - if yes, put process A into *Ready* state and switch process B into *Running* state.
- Easy to implement!
- But what if processes don't cooperate?
  - E.g. infinite loop without system calls

## Preemptive timesharing

OS sets up **timer interrupt**.

- Timer is implemented in the hardware.
- "Fires" e.g. every 10 ms
- Interrupt switches to kernel mode and calls interrupt handler in the OS
- OS can then switch processes just like in the cooperative case.

Now OS always regains control in regular intervals!

This is important, because it enables the user to kill a process! Think about the Windows Task Manager, if a program doesn't react, you can simply kill it. In a cooperative timesharing system, you couldn't open the task manager, because that's another process.

## Summary

- OS makes hardware easy to use
- Virtualisation:
  - looks like each process has exclusive access to hardware
  - system calls give acces to OS (file system, network, memory)
  - cooperative vs. preemptive multitasking

## Next lecture

- When to switch (process scheduling)
- Virtual memory