Operating Systems

Dr. Vivek Nallur (vivek.nallur@ucd.ie)

First things, first – enrol on Moodle

- Register yourself [ucdconnect account] on Moodle:

https://csmoodle.ucd.ie/moodle/login/index.php

- Search for: COMP 2006J
- Find a module entitled: COMP2006J Operating Systems 2017-2018

Group Number	Enrollment Key
153711	COMP2006JVN153711
153712	COMP2006JVN153712
153713	COMP2006JVN153713

Group Number	Enrollment Key
163721	COMP2006JVN163721
163722	COMP2006JVN163722
163723	COMP2006JVN163723

Plagiarism & UCD Computer Science

- **Plagiarism is a serious academic offence**
  - [Student Code, sections 6.2 & 6.3] or [UCD Registry Plagiarism Policy] or [CS Plagiarism policy and procedures]
- Our staff and demonstrators are **proactive** in looking for possible plagiarism in all submitted work
- Suspected plagiarism is reported to the CS Plagiarism subcommittee for investigation
  - Usually includes an interview with student(s) involved
  - 1st offence: **usually** 0 or NG in the affected components
  - 2nd offence: may be referred to the **University disciplinary committee**
- Student who enables plagiarism is equally responsible

http://www.ucd.ie/registry/academicsecretariat/docs/plagiarism_po.pdf

http://www.ucd.ie/registry/academicsecretariat/docs/student_code.pdf

http://libguides.ucd.ie/academicintegrity

Operating Systems Structure

Operating System Architecture

Operating System Structure

- In order to understand any operating system structure, we must consider its **components** and their organisation

- What are the essential components?

- How do they relate to each other?

Operating System Structure

- It is important to remember that
- The concepts we study will exist in some form in every operating system
  - But they will be implemented in different ways
- Divisions between components are not always clearly defined

Operating System Components

- **Kernel**: software containing the core OS components; it may typically include:

- Memory Manager
  - Provides efficient memory allocation and deallocation of memory

- I/O manager
  - Handles input and output requests from and to hardware devices (through device drivers)

Operating System Components

- Inter-process communication (IPC) manager
  - Provides communication between different processes (programs in execution)
- Process Manager (scheduler)
  - Handles what is executed when and where (if more than one CPU)

Operating System Components

- A OS kernel may consist of many more components:
  - System service routines
  - File System (FS) manager
  - Error handling systems
  - Accounting systems
  - System programs
  - And many more

Operating System Interface

- Original OS interfaces were very simple and called **Command Line Interface** (CLI) or **Command Interpreter** (CI)
- This is the like the command prompt on windows and terminal or shell on Mac/Linux
- The user types a command and the CI executes it

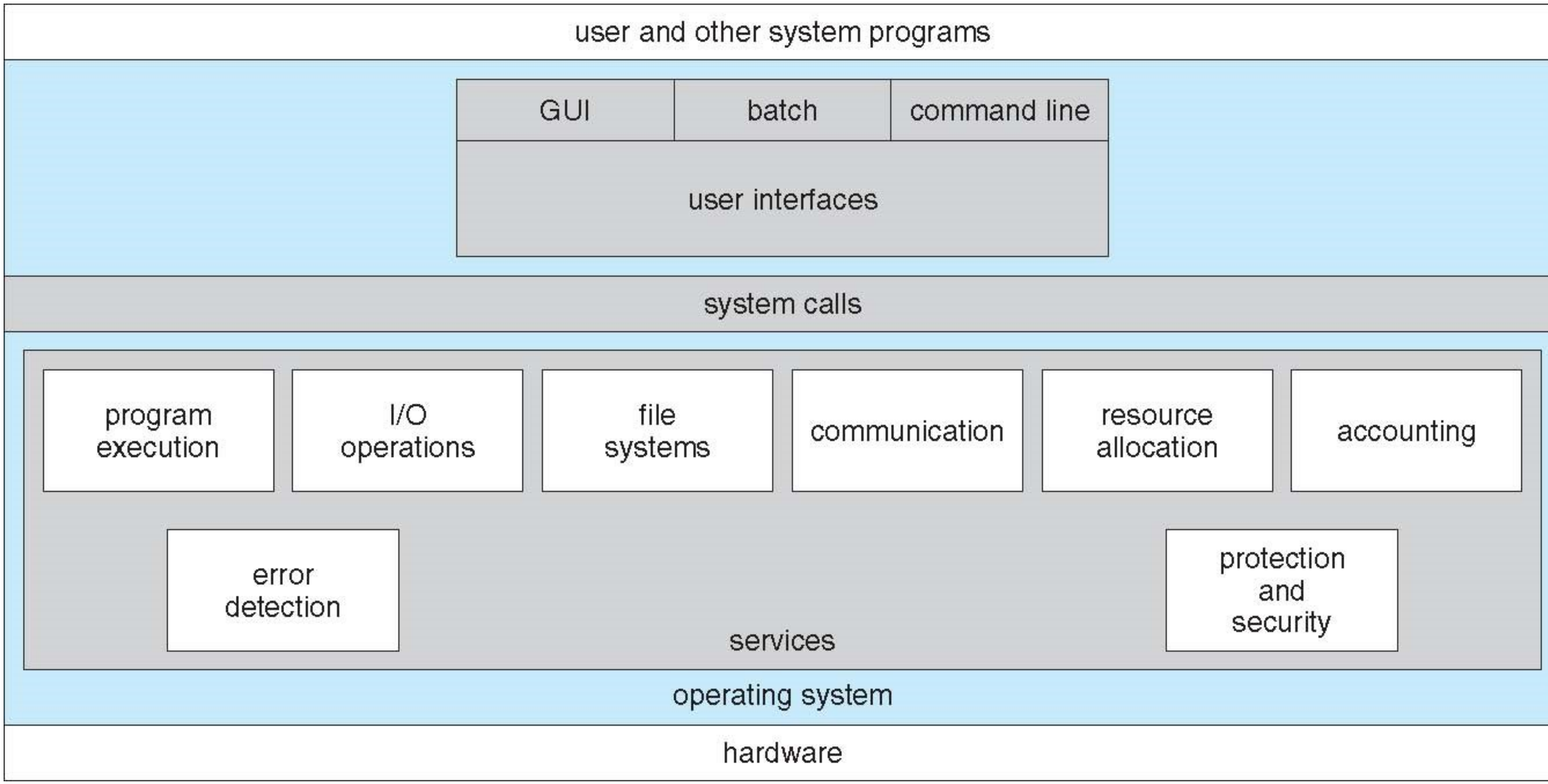
Operating System Interface

- Later systems used a more user firendly Graphical User Interface (GUI)

- Based on Desktop idea
  - Usually mouse, keyboard, and monitor
  - WIMP, (windows, icons, menus, pointing)
  - Icons represent files, programs, actions, etc

- Used on almost all systems today

Example system



OS Structure Issues:

- How are all of these components organised?
- What are the entities involved and where do they exist?
- how do these entities cooperate?

Operating System Goals

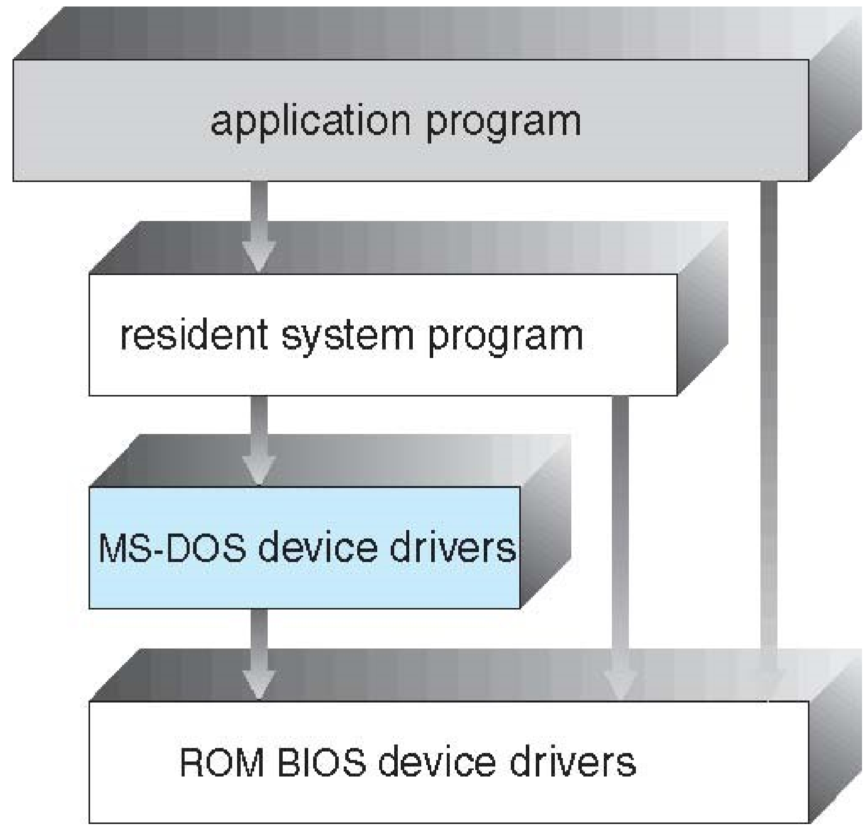
- When we design an operating system we want it to be:
  - Efficient – (High **throughput**)
  - Interactive
  - **Robust** – (Fault tolerant & reliable)
  - Secure
  - Scalable
  - Extensible
  - Portable

Monolithic Architecture

- Traditionally, systems were built around **monolithic** kernels
  - every OS component is contained in the kernel
  - any component can directly communicate with any other (by means of function calls)
  - due to this they tend to be highly efficient (performance)

Example : MS DOS

- Written to provide the most functionality in the least space
- Not divided into modules
- Interfaces and levels of functionality are not well separated



Problems with Monolithic Structure

- Because they are unstructured they are hard to understand modify and maintain

- Susceptible to damage from errant or malicious code, as all kernel code runs with unrestricted access to the system

Layered Structure

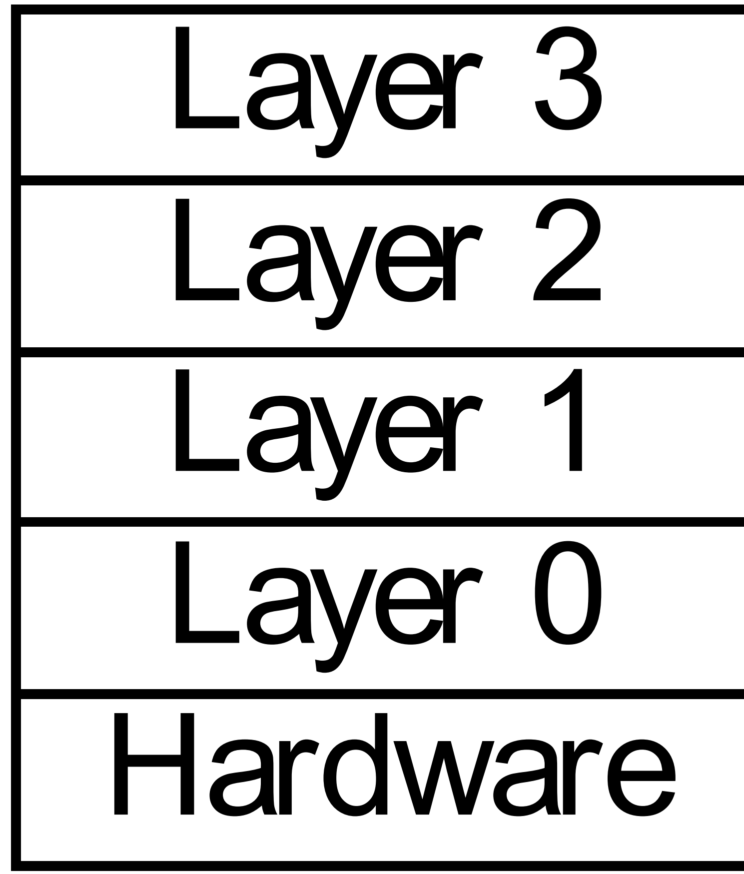
- To try and solve the problems with the monolithic structure, the layered structure was developed
- Components are grouped into layers that perform similar functions

Advantages of Layered Structure

- Designing the system as a number of modules gives the system structure and consistency
- This allows easier debugging, modification and reuse

Layered Architecture

- Each layer communicates only with layers immediately above and below it
  - each layer is a virtual machine to the layer above
  - a higher layer provides a higher-level virtual machine



First Layers-based OS

- Layering was first used in Dijkstra's THE OS (1968)

- Each layer "sees" a logical machine provided by lower layers

The Layers

layer 4	User Programs
layer 3	I/O Management
layer 2	Console Device (commands), IPC
layer 1	Memory Management
layer 0	CPU Scheduling (multiprogramming)
	Hardware

First Layers-based OS

- Each layer "sees" a logical machine provided by lower layers
  - layer 4 (user space) sees virtual I/O drivers
  - layer 3 sees virtual console
  - layer 2 sees virtual memory
  - layer 1 sees virtual processors
- Based on a static set of cooperating processes
- Each process can be tested and verified independently

Problems with layering

- Appropriate definition of layers is difficult
  - A layer is implemented using only those operations provided by lower-lever layers
  - A real system structure is often **more complex** than the strict hierarchy required by layering

Problems with layering

- The secondary memory (disk) driver would normally placed **above** the CPU scheduler (because an I/O wait may trigger a CPU rescheduling operation)
- However, in a large system the CPU scheduler may need more memory than can fit in memory: parts of the memory can be swapped to disk (virtual memory), and then the secondary memory driver should be **below** the CPU scheduler
- Both things cannot be achieved at the same time; therefore the layered approach is not flexible

Problems with Layering

- Performance issues
  - Processes' requests might pass through many layers before completion (layer crossing)
  - System throughput can be lower than in monolithic kernels

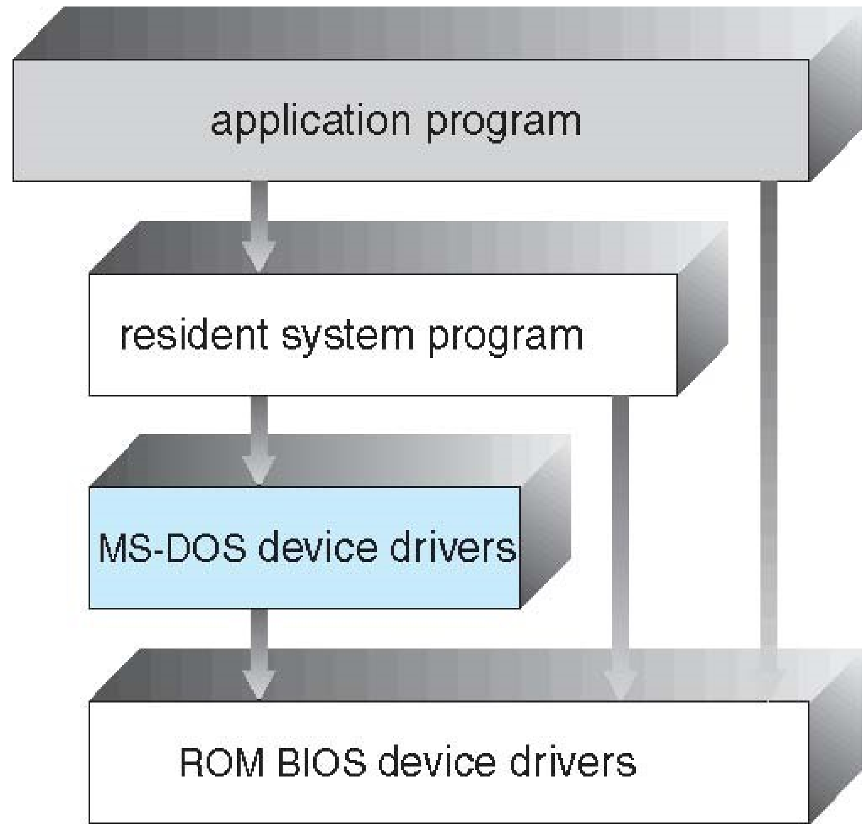
Problems with Layering

- Still susceptible to malicious/errant code if all layers can have unrestricted access to the system
  - As we will see later, this can only be avoided through hardware

- As a consequence, (imperfect) layers are often used for convenience in system design, but any OS implementation cannot be purely layered

Example: MS DOS

- Structure is somewhat layered
- Interfaces and levels of functionality are not well separated



Microkernel Architecture

- The **microkernel** (μ-kernel) architecture was designed to minimise the services offered by the kernel
- This was an attempt to keep it small and scalable

Microkernel Architecture

- There is no agreement about minimal set of services inside the microkernel
- At least: minimal process and memory management capabilities, plus inter-process communications
- Services such as networking and file system tend to run nonprivileged at the user process level

Benefits of Microkernel

- Modularity
- It promotes uniform interfaces
- Distributed systems support:
  - modules communicate through the microkernel, even through a network

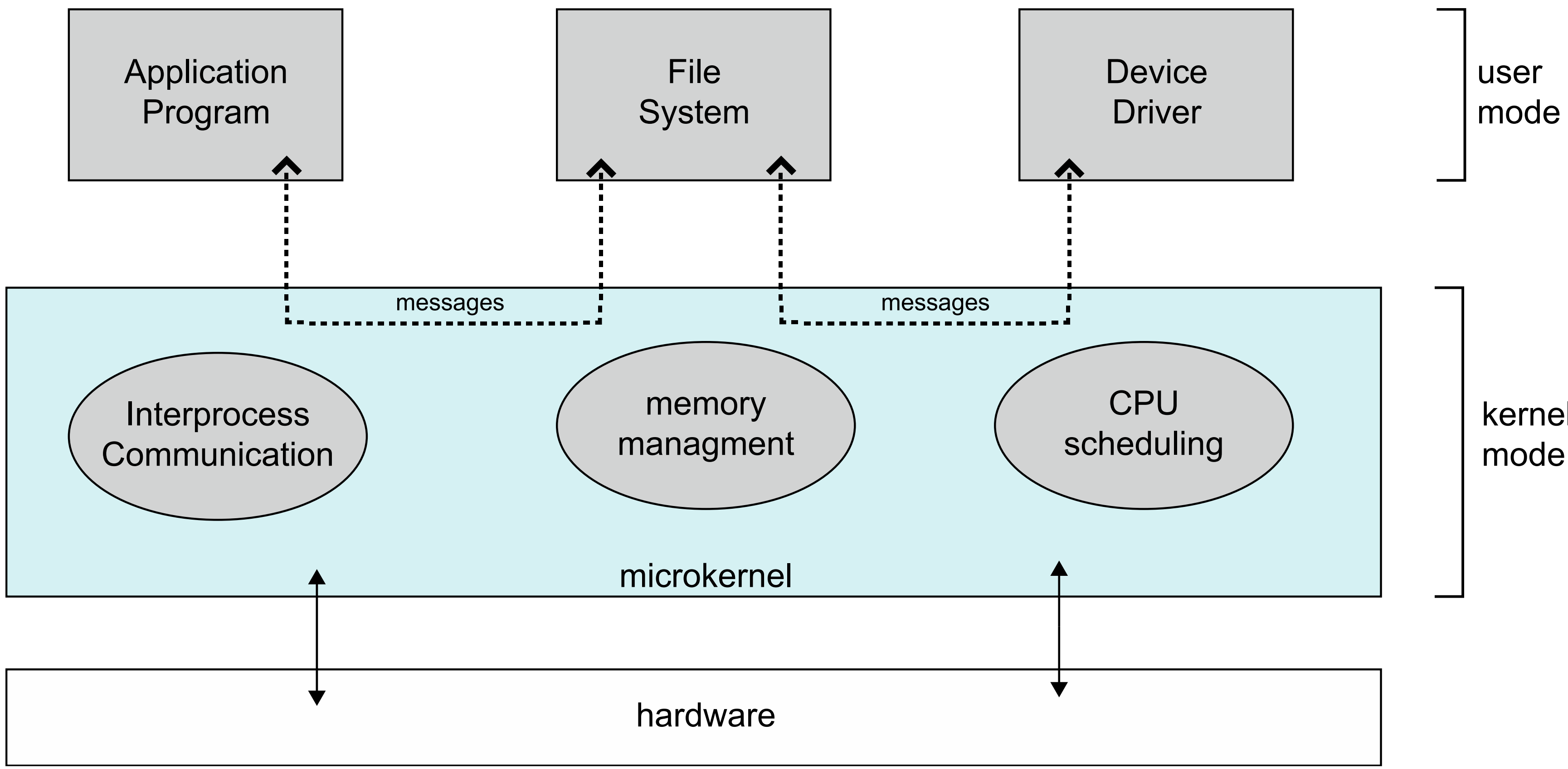
Benefits of Microkernel

- Reliability
- Scalability
- Portability
- Easy to extend and customise

Disadvantages of Microkernel

- System performance can be worse than in monolithic kernels, especially if kernel minimisation is taken too far

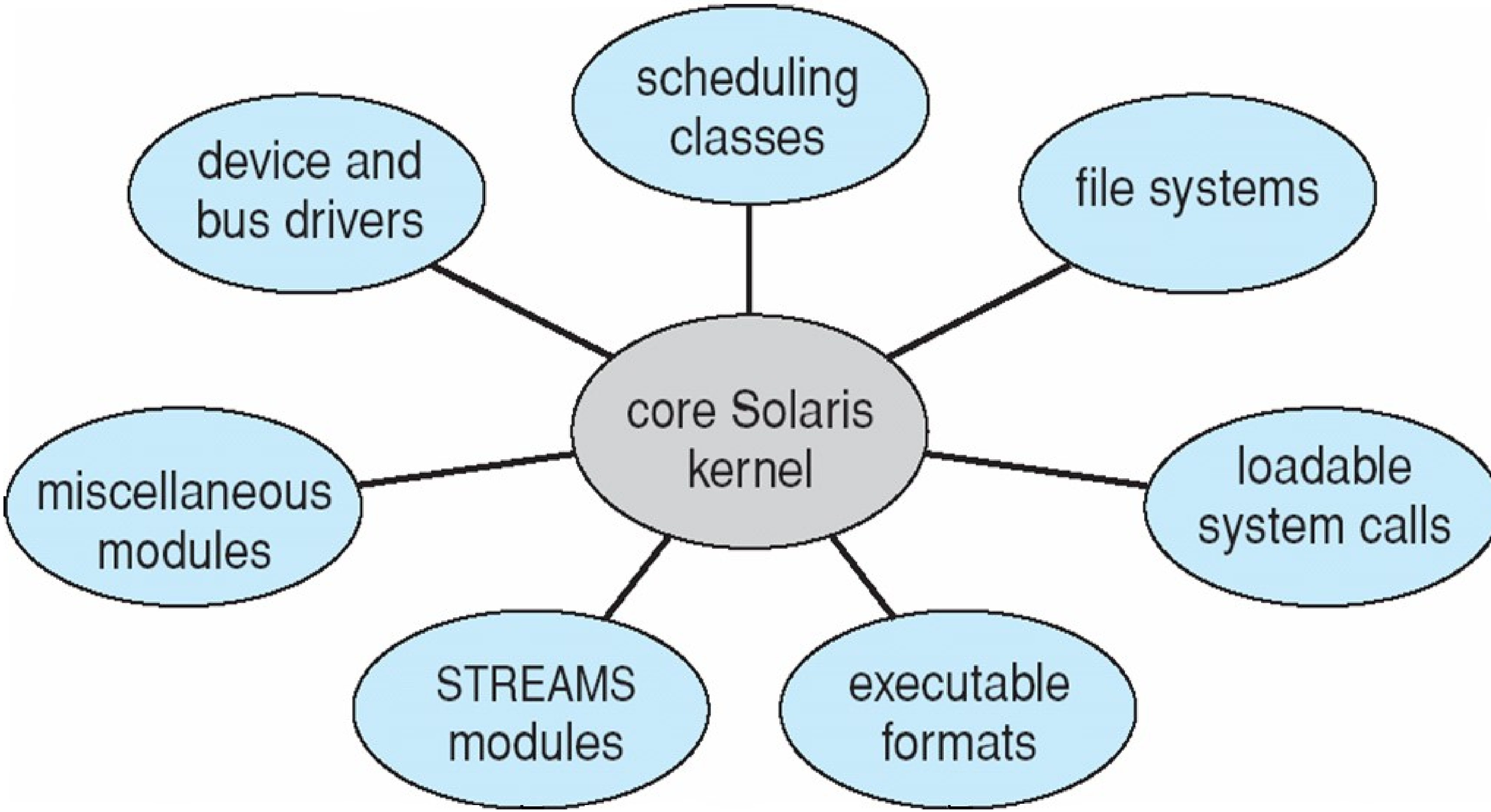
Microkernel System Structure



Modules

- Many modern operating systems implement loadable kernel modules
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexibility
  - Linux, Solaris, etc

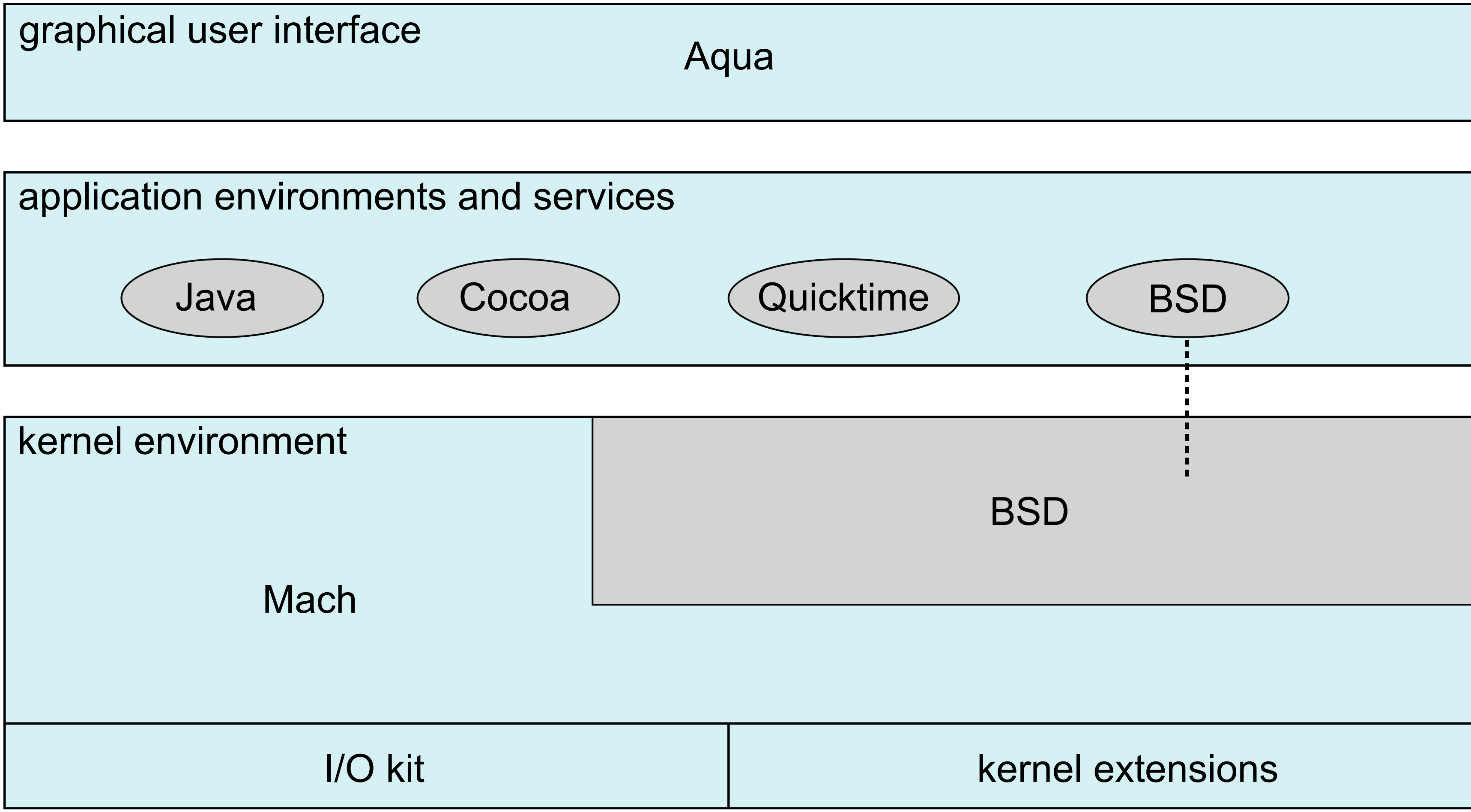
Solaris Modular Approach



Hybrid Systems

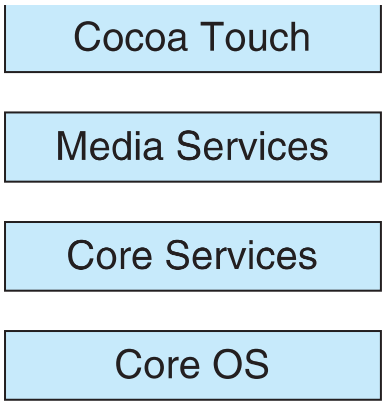
- Most modern operating systems are actually not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels are in kernel address space, so they are monolithic, but also modular for dynamic loading of functionality
  - Apple Mac OS X combines a layered approach with a kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules

Mac OS X Structure



iOS

- Based on Mac OS X
  - Cocoa Touch Objective-C API for developing apps
  - Media services layer for graphics, audio, video
  - Core services provides cloud computing, databases
  - Core operating system, based on Mac OS X kernel



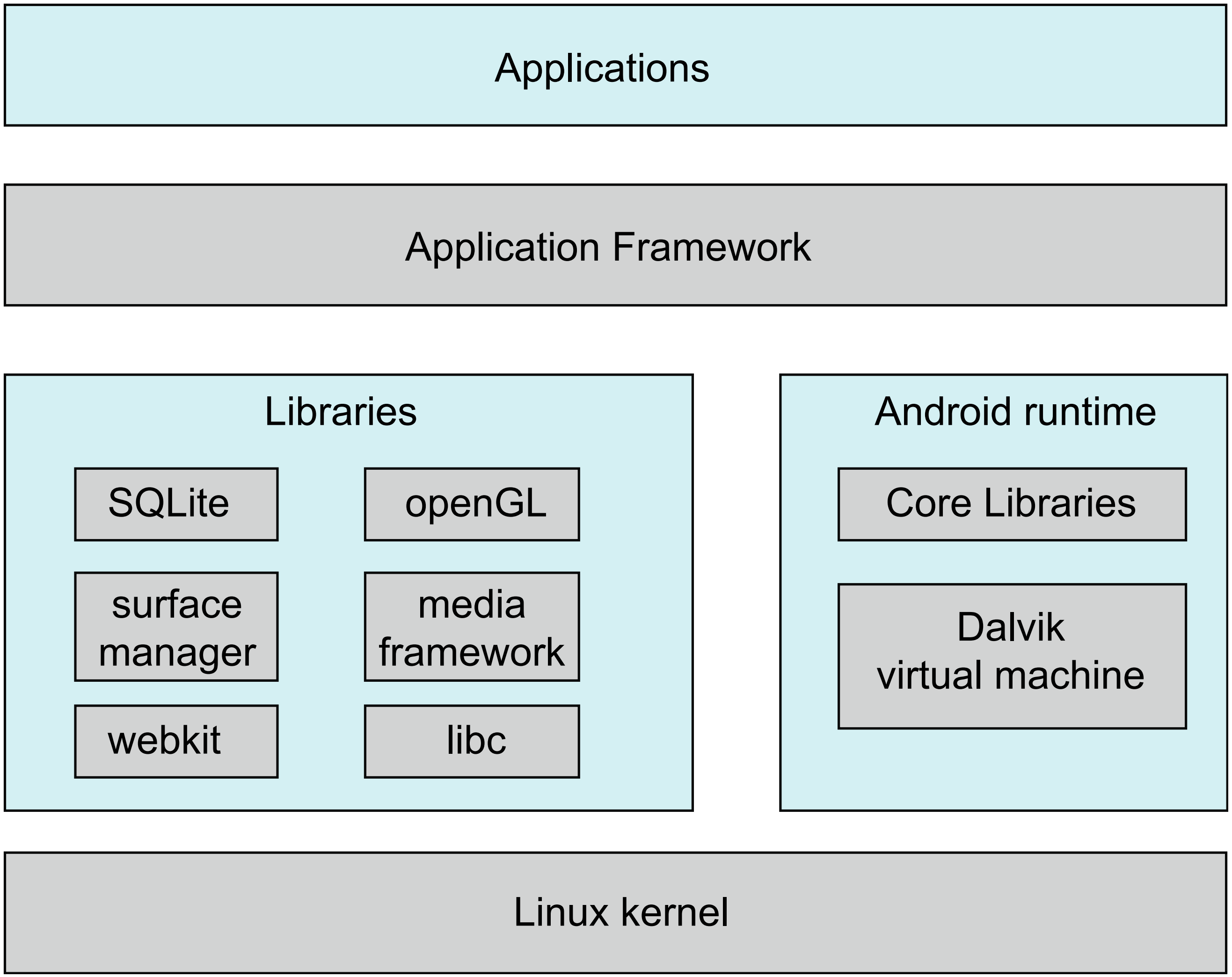
Andriod

- Open Source OS Developed by Open Handset Alliance (mostly Google)
- Similar stack to IOS
- Based on Linux kernel but modified
  - Provides process, memory, device-driver management
  - Adds power management

Andriod

- Runtime environment includes core set of libraries and Dalvik/ART(2015) virtual machine
  - Apps developed in Java plus Android API
  - Java class files compiled to Java bytecode then translated to executable than runs in Dalvik VM

Android Architecture



Operating Systems Structure

Operating System and Hardware
Features

Operating System and Hardware

- An OS is partly dedicated to a specific hardware architecture
- Good hardware support can greatly simplify its design
- Hardware features are many times motivated by OS needs

Protected Instructions

- In modern systems some instructions are typically restricted to the OS (**protected or privileged instructions**)

- This is to prevent a faulty or malicious user program from affecting the whole system

Protected Instructions

- Typically, users are not allowed to
  - Directly access I/O (disk, printer,. . .)
  - Directly manage memory
  - Execute CPU halt instructions

- These operations are always handled through privileged instructions or memory mapping

Dual Mode Operation

- The implementation of protected instructions requires some type of hardware mechanism

- The HW must support - at least - two operation modes
  - kernel mode: access to all the CPU instruction set
    - also called monitor/system/privileged mode
  - user mode: access restricted to a subset of the instruction set

Dual Mode Operation

- The mode is indicated by a **status bit** (mode bit) in a protected processor register
  - OS programs & protected instructions executed in kernel mode
  - user programs executed in user mode
- Examples of protection in older and newer systems:
  - MS-DOS (based on Intel 8088): no protection modes
  - Windows 2000/XP, OS/2, Linux (based on Intel x86 systems): protection modes

Crossing Protection Boundries

- User-mode programs cannot execute privileged instructions, but they still need kernel-mode services (I/O operations, memory management, etc)

- To execute a privileged instruction, a user must call an OS procedure: **system call**

Crossing Protection Boundries

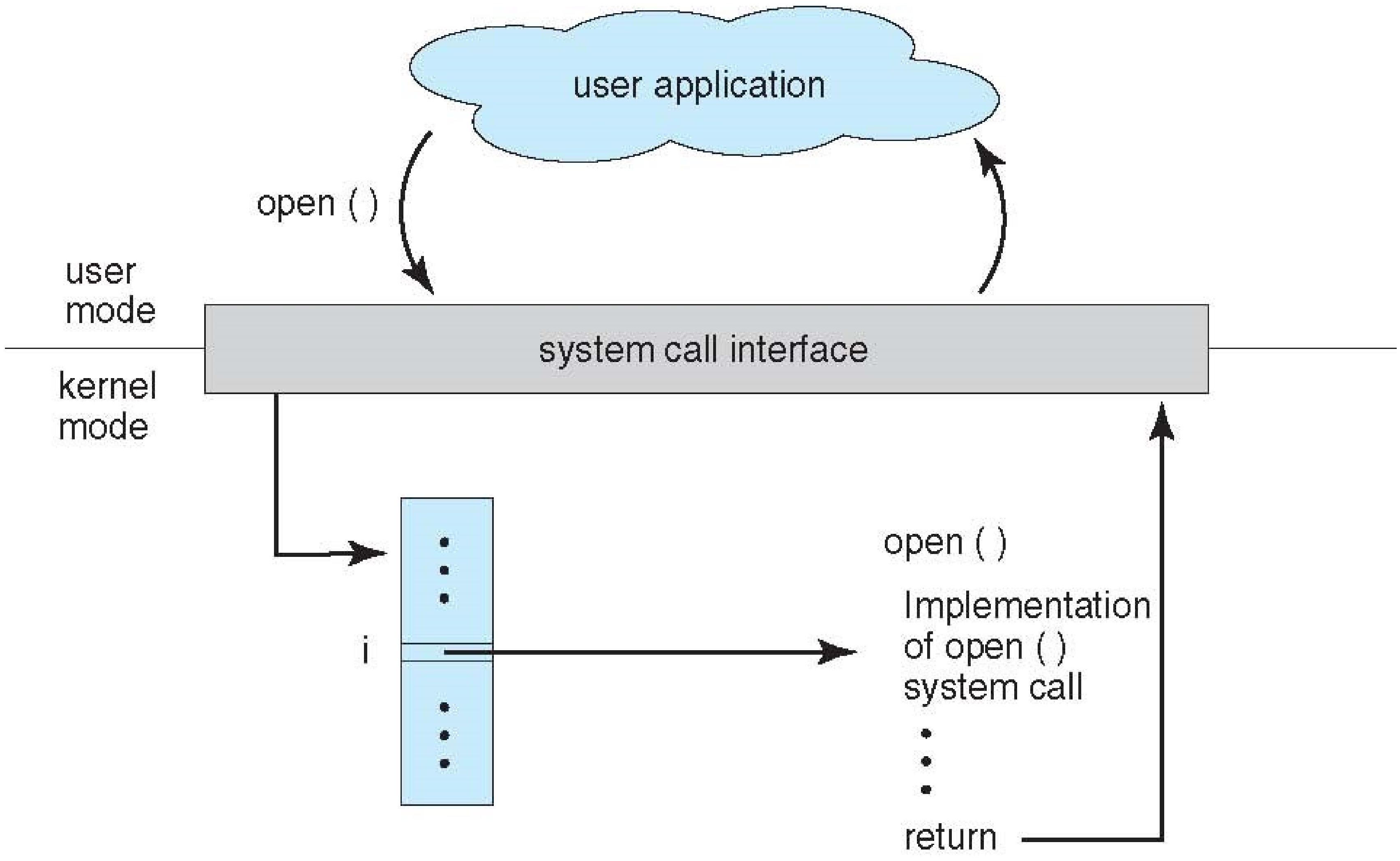
- A system call causes a **trap**, which jumps to the trap handler in the kernel
- When called the trap handler
  - uses call parameters to determine which system routine to run
  - saves caller's state: program counter (PC), mode bit, . . .

Crossing Protection Boundries

- After this the hardware must
  - Implement caller's parameters verification (e.g. memory pointers should only be allowed within user's section)
  - Return to user-mode when trap system call finished

- The trap is treated by the hardware as a **software-initiated interrupt**

The Open System Call



Exceptions

- **Exception** (hardware-initiated interrupt): basically the same as a trap
- Exceptions are automatically triggered by an error or a particular situation rather than on purpose (like in a system call)

Exceptions

- Exceptions also transfer control to a handler within the OS
  - system status can be saved on exceptions (memory dump), so that faulty processes can be later debugged
- Exceptions decrease performance
  - exception conditions could be detected by inserting extra instructions in the code, but at a high performance cost

Typical Exceptions

- memory access out of user space
- overflow, underflow
- trace traps (debugging)
- illegal use of privileged instructions
- virtual memory (paging): page faults, write to read-only page

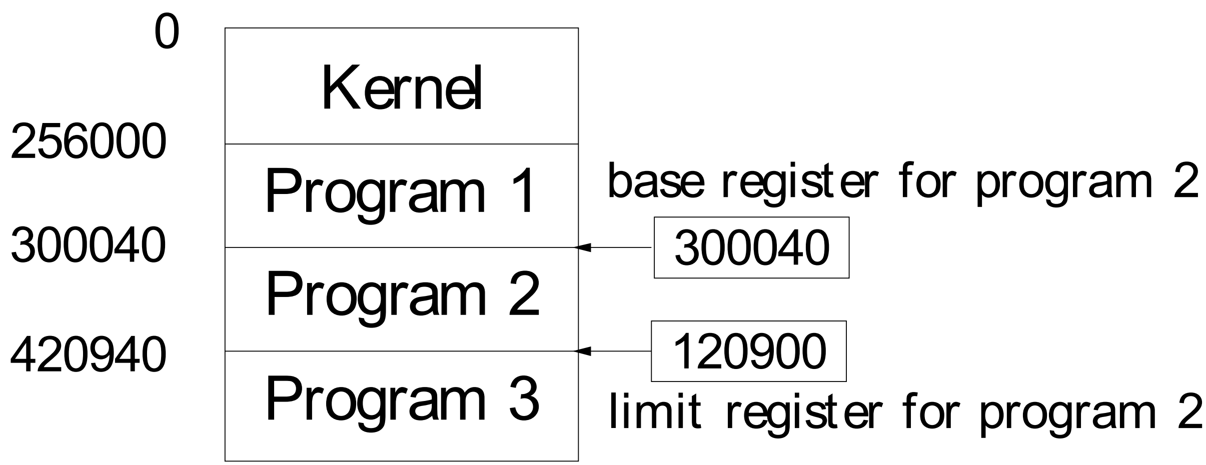
Memory Protection

- A memory protection mechanism must protect
  - user programs from each other
  - OS (kernel) from user programs

- Simplest scheme is to use **base** and **limit** registers

Base and Limit Registers

- Base and limit registers are loaded by the OS before starting the execution of any program in user mode



- base ≤ address < base + limit; otherwise exception raised

Memory Protection

- Currently memory protection is more complex than this
- However, the simple base and limit scheme is the basis of modern virtual memory when a strategy called "segmentation" is used

I/O Control

- All I/O instructions are privileged
  - This is because a program could disrupt the whole system by issuing illegal I/O instructions
- Two situations must be considered:
  - I/O start: handled by system calls
  - I/O completion and I/O events: handled by interrupts

I/O Control

- Interrupts are the basis for **asynchronous** I/O
  - I/O devices have small processors that allow them to run autonomously (i.e. asynchronously with respect to the CPU)
  - I/O devices send interrupt signals when done with an operation; CPU switches to address corresponding to interrupt
    - an interrupt vector table contains the list of kernel routine addresses that handle different events

CPU Protection

- Apart from protecting memory and I/O, we must ensure that the OS always maintains control

- A user program might get stuck into an infinite loop and never return control to the OS

- **Timer**: it generates an interrupt after a fixed or variable amount of execution time

CPU Protection

- When an interrupt is generated by the Timer the OS may choose to treat the interrupt as a fatal error (and stop program execution) or allocate more execution time

- note: in time-sharing systems a timer interrupt is periodically generated after a fixed period of time (for scheduling a new program)

Summary

- An OS provides a number of services
- These relate to managing
  - The hardware
  - The processes
  - The users
  - Communication between all of them
- Multiple ways of structuring the kernel, the system programs and user programs
- Each has advantages and disadvantages

Summary - II

OS Requirement	Hardware Feature
Dual kernel/user modes	Protected instructions
System calls	Trap instructions and vectors
Exceptions, signals	Interrupt vectors
Memory protection	Base and limit registers
I/O control	Interrupts
CPU protection, scheduling	Timer (clock)

That's all, folks!

- Questions?