Introduction

Chapters 1.1, 1.2, 1.3.1, 1.3.2, 1.4.2, 1.4.1, 1.5, 1.6, 1.7, 1.8. 1.10

# What Operating Systems Do

* A computer system is divided into 4 parts:
  + Hardware: CPU, memory, IO devices
  + Operating System
  + Application programs: Word processors, spreadsheets, compilers, web browsers etc.
  + User: you!
* The operating system controls the hardware, and coordinates its use among the various application programs for the various users.
* A computer system can also be seen as a government, providing an environment within which other programs can operate.

## User view

* Diagram

  Description automatically generatedDepends on interface.
* Goal: maximise the work the user is performing.
* In most cases, the operating system is designed for **ease of use**.
* Some attention is paid to performance and security, and **none** to **resource utilisation**.

## System View

* The operating system is also a **resource allocator**.
* Many resources may be required to solve a problem:
  + CPU time
  + Memory space
  + I/O devices
* These must be allocated in such a way that the system can run efficiently
* The operating system is also a **control program**, managing the execution of user programs to prevent errors and misuse of the computer.
* This specifically relates to I/O devices

## Graphical user interface, text, application, chat or text message Description automatically generatedDefining Operating Systems

* The fundamental goal of an operating system is to execute programs and make solving user problems easier.
* A common definition is that the operating system is the one program running on the computer at all times: the **kernel**.
* There are two other types of programs:
  + System programs,
  + Application programs: all programs not associated with the operation of the computer system
* More recently operating systems feature **middleware**:
* These can be systems that support databases, multimedia and graphics.
* Operating system includes:
* **kernel** (runs at all times)
* **Middleware frameworks** (ease application development) a set of software frameworks that provide additional services to application developers.
* **System Programs** associated with the operating system but not part of the kernel. They aid managing system while it is running.

# Computer Organisation

* Diagram

  Description automatically generatedA computer usually consists of a set of CPU(s) and **device controllers**, connected through a **bus** that provides access to components and shared memory.
* The **device controller** is responsible for moving the data between the peripherals it controls and the **local buffer** storage.
* Most OS’s also have a **device driver** for each controller. This provides the OS with a uniform interface to the device, such that the OS can understand the device controller.
* Device controllers and the CPU can execute in **parallel**, so to ensure orderly access to memory, a **memory controller** synchronises access to memory.

## Interrupts

* The device controller can receive commands via the device driver.
* To inform the driver that it has finished the task (e.g. data transfer for read), it uses **interrupts**.

### Overview

* Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually via system bus.
* Interrupts have several other uses: they are key to hardware-OS interaction.
* When the CPU is interrupted, the CPU stops its current job and begins execution from a fixed location, which usually contains a starting address where the service routine for the interrupt is located.

Diagram

Description automatically generated

* An interrupt must transfer to the appropriate service routine.
* Interrupts must also be processed quickly, as many can be received at one given time.
* To meet this criteria, we use a generic **routine**, a method executed by the CPU to examine the interrupt information.
* The routine in turn calls the interrupt specific handler.
* To find the interrupt routine, an **interrupt vector** is used, where the routine is called indirectly through the table.
* This vector contains all the addresses of the interrupt service handlers.
* The interrupt vector of addresses is indexed by a unique number, given by the interrupt request, to provide the address fork the interrupt service routine for the interrupting device.
* The interrupt architecture must also **save the state information** of whatever was interrupted, in order for the CPU to return to its previous task successfully.

### Implementation

* The interrupt mechanism works as follows:
* After executing every instruction, the CPU senses the **interrupt-request line**, which is a wire connected to it.
* A controller **raises** and interrupt by asserting a signal onto the interrupt-request line. The CPU detects the signal which contains the interrupt number.
* The CPU **catches** the interrupt and **dispatches** to the **interrupt-handler** routine, using the interrupt number as an index into the index vector.
* Execution begins at the address associated with the given index:
* Any state information altered during the interrupt operation is saved.
* The CPU finds the cause of the interrupt.
* The CPU then performs the necessary processing.
* The previous state is restored
* A return\_from\_interrupt instruction is used to return the CPU to the execution state prior to the interrupt.
* The interrupt has been **cleared** and the CPU continues.
* This is a basic interrupt mechanism, but we need more sophisticated interrupt-handling features for modern OS’s:

1. We need the ability to **defer** interrupt handling during **critical processing**.
2. We need an efficient way to dispatch the proper interrupt handler for a device.
3. We need **multilevel interrupts**, such the OS can prioritise interrupts, and respond with the appropriate degree of urgency.

* These three features are provided by the CPU and the **interrupt-controller hardware**.

Diagram

Description automatically generated

* Most CPU’s have **2** interrupt lines:
* **Nonmaskable Interrupt**: reserved for events such as unrecoverable memory errors
* **Maskable Interrupt**: It can be turned off by the CPU to allow critical instructions to be completed first.
* In practice, computers have more devices therefore more interrupt handlers than they have address elements in the interrupt vector.
* To solve this problem, **interrupt chaining** is used, whereby each element in the interrupt vector points to the head of a list of interrupt handlers, similarly to a heap-stack structure.
* When an interrupt is raised, the handlers on the corresponding list are called one by one, until the one that can service the request is found.
* Table

  Description automatically generatedBelow is the Intel processor event-vector table:
* The events 0-31 are **nonmaskable** i.e. they are used for error conditions
* The events 32-255 are **maskable**, so they are used for purposes such as device-generated interrupts.
* The interrupt system also implements a system of **interrupt** **priority levels**, enabling the CPU to defer low-priority interrupts without masking all of them, and order interrupts appropriately.
* To summarise, interrupts are used throughout modern OSs to handle asynchronous events and other purposes.
* Device controllers and hardware faults raise interrupts.
* Modern computers use a system of interrupt priorities to order interrupts and account for urgency.
* Efficient interrupt is impertinent to a good system because they are used so heavily used.

## Storage Structure

### EEPROM and RAM

* The CPU can load instructions from memory only, so any required programs must be present in memory first.
* This is called **RAM** or **main memory**, and is rewriteable using semi-conductor technology (Dynamic Random-Access Memory or **DRAM**)
* RAM is **volatile** i.e. temporary, so it is cleared when the computer is turned off.
* There are other forms of memory, such as the **bootstrap** **program**, which is the very first program to run on start-up before loading in the OS.
* This is held on electrically erasable programmable read-only memory (EEPROM) and other forms of **firmware**- storage that is infrequently written to and non-volatile.
* EEPROM can be changed, but not frequently.
* It is also low speed and so it mainly consists of static programs and data that is not frequently used.
* E.g. iPhones used EEPROM to store serial numbers and hardware information about the device.
* All forms of memory provide an array of **bytes**, each bye having its own address.
* The CPU uses load and store commands to interact with memory:
* load instructions move a byte or word to main memory from an **internal register** within the CPU (Fetch-decode-execute cycle)
* store instructions moves the content of a register into main memory
* Aside from explicit load and store memory, the CPU automatically loads instructions from main memory for execution from the location stored in the program counter
* Typical instruction-execution follow these steps:
  1. Fetch instruction from memory and store in **instruction register**.
  2. Instruction is then decoded and may cause operands (for operation) to be fetched from memory and stored in internal registers.
  3. CPU executes instruction
  4. Result is stored in memory if needed.
* Memory units see only a stream of memory addressed, so we can ignore how a memory address is generated by a program.
* We are only interested in the **sequence of memory addresses** generated by the running program.

### Secondary Storage

* Ideally the programs and data used would reside in main memory permanently; however, this is not the case as :

1. Main memory is usually to small to store **all** needed programs and data permanently.
2. Main memory is **volatile**

* Therefore, most computer systems require a **secondary storage**, extending main memory, which need to hold large quantities of data.
* This can be a hard-drive, SSD, or any **non-volatile memory** (NVM) devices, providing storage for both programs and data.
* Many programs use secondary storage as both the source and destination of processing.
* It is slower than main memory due to size and distance from the CPU, so its management is important.

### Hierarchy of Storage Devices

* There is also **tertiary** storage, which is high capacity but extremely slow form of memory.
* They all form a sort of hierarchy as follows:
* Diagram

  Description automatically generatedVolatile storage is referred to as **memory**, and non-volatile storage is referred to as **NVS**.
* NVS can be of two types:
* **Mechanical**: HDDs, Optical disks, holographic storage etc.
* **Electrical**: FRAM, NRAM, SSD etc. Referred to as **NVM**

## I/O Structure

* Diagram

  Description automatically generatedInterrupt driven I/O is fine for moving small amounts of data but can produce high overhead when used for bulk data movement.
* We solve this by using **direct memory access** (DMA).
* Buffers, pointers, and counters are set up for the I/O device, and then entire blocks of data can be transferred directly to or from the device and main memory, with no CPU intervention.
* Only one interrupt is needed for each block: to notify the CPU that the operation is completed.
* This means the CPU can continue other work while the device controller performs data transfer.
* Some high-end systems use a switch instead of bus architecture.
* This allows multiple components to communicate with each other concurrently, instead of competing for cycles on the shared bus.
* This makes DMA even more effective.

# Computer-System Architecture

## Single Processor Systems

* Most computer systems previously used a single processor, containing one core.
* The **core** executes instructions and registers for storing data locally.
* The CPU then executes the instruction set, including instructions from processes.
* Devices can have **device-specific** processors as well, running a limited instruction set without running processes.
* These can be managed by the OS e.g. disk-controller microprocessor receives a sequence of requests from the main CPU core and implements its own disk queue and scheduling algorithm.
* As a result, the CPU does not have to deal with the overhead of disk management.
* In other situations, microprocessors are low-level components built into the hardware, and cannot communicate with the OS, completing their jobs autonomously.

## Multiprocessor Systems

* These have two or more processors, each with a single core CPU (most systems are like this)
* Note that the speed up ratio of processors is less than , because there is some overhead for organising and operating each unit when they cooperate on a task.
* The expected gain of a processor lowers as increases.

### SMP (Symmetric Multiprocessing)

* Diagram

  Description automatically generatedA lot of multiprocessor systems use **Symmetric Multiprocessing** (SMP), where each CPU processor performs all tasks.
* Each processor has its own CPU has shown above, as well as its own registers and (private or local) cache.
* All processors however share physical memory via system bus.
* This model allows several processes to run simultaneously i.e. processes run with CPUs, without large performance disruption.
* It can however result in inefficiencies, whereby one CPU is idle, and the other is overloaded.
* These inefficiencies can however be avoided by sharing certain data structures, but must be written carefully

### Diagram Description automatically generatedMulticore Systems

* This is when multiple computing cores reside on a single chip.
* These can be more efficient than multiple single-core chips, as on-chip communication is much faster than inter-chip communication.
* One chip also uses a lot less power than multiple single-core chips.
* The OS interprets an core system as CPUs, so OSs must make efficient use of said cores.

### Diagram Description automatically generatedNUMA (Non-uniform memory access)

* As CPU number increases, overhead increases as contention for the system bus becomes a bottleneck, and performance begins to degrade.
* The alternative approach to accommodate more CPUs is **NUMA**.
* With this system, each CPU has a local memory, and is then connected by a **shared system interconnect**, so all CPU’s share one physical address space.
* This means that when a CPU accesses local memory, it is fast and there is no contention over the system interconnect, allowing NUMA systems to scale more effectively as CPUs are added.
* It can however be slower if a CPU wants to access another CPUs local memory. This can be minimised through careful CPUI scheduling and memory management.

### Blade Servers

* **Blade Servers** are systems where multiple processor, I/O and networking boards are places in the same chassis.
* The difference is that each blade processor board boots independently and runs its own OS.
* Some boards are multi-processors as well, meaning these servers consist of multiple independent microprocessor systems.

## Key Terminology

|  |  |
| --- | --- |
| Component | Description |
| CPU | hardware that executes instructions |
| Processor | Physical chip containing one or more CPUs |
| Core | Basic Computational unit of CPU |
| Multicore | Including multiple cores of the same CPU |
| Multiprocessor | Including multiple processors |

## Clustered System

* Multiple multicore systems connected together.
* Uses **fault tolerance**: monitored machine can be restarted by monitoring machine.
* **Graceful Degradation**: providing service proportional to level of surviving hardware
* Can be **symmetric** or **asymmetric**, and allows for parallelization to run cores in parallel.
* **Distributed Lock Managers** (**DLMs**) assure access to database results in no conflicting errors.
* **Storage Area Networks** (**SANs**) allow many systems to attach to a pool of storage, so any host can be assigned with running an application stored on a SAN.

# Operating-System Operations

## Dual-Mode and Multimode Operation

* An OS must ensure that an incorrect or malicious program does not disrupt the OS and other programs.
* To distinguish between execution of OS code and user-defined code, we have **modes** of operation: **User mode** or **kernel mode** (also called supervisor, system or privileged mode)
* A **mode bit** is added to the computer’s hardware to indicate the current mode:
* Kernel: 0
* User: 1
* Diagram

  Description automatically generatedThis allows us to distinguish between OS and user tasks.
* The above diagram is the process followed when **traps**, **system calls** or **interrupts** occur.
* Any instructions that can potentially cause harm are **privileged instructions**.
* When these occur, they can only be executed in kernel mode. If in user mode, the hardware will not execute the instruction but treat it as illegal, and trap it to the operating system.
* Processors can have several modes, e.g. Intel has 4 separate **protection rings**, with 0 being kernel and 3 being user.
* CPUs supporting virtualisation have separate modes to indicate if the **virtual machine manager** (**VMM**)is controlling the system.
* VMM has more privileges than a user, but fewer than kernel, because it needs the level of privilege to create and manage VMs and change the CPU state.
* **System calls** are used to request the OS to perform kernel mode tasks on behalf of the user’s program i.e. it requests action by the OS.
* Usually these take the form of a **trap** (an exception or fault), though some systems of specific syscalls instead.
* They follow the following procedure:

1. System call is executed and treated by hardware as a software interrupt
2. Control passes through interrupt vector to service routine in OS, setting mode bit to kernel mode i.e. 0 (the system-call service routine is part of the OS)
3. Kernel examines interrupting instruction to determine what system call has occurred via a parameter, that indicates what kind of service the user program is requesting.
4. Additional info needed for request is passed into registers if needed on stack or in memory (with pointers to memory locations passed into registers i.e. stack-heap).
5. Kernel verifies if parameters are correct and legal
6. Request is executed
7. Control is returned to the instruction following the system call.

* Hardware protection detects errors that violate modes.
* In the case of an error, the OS handles it, as the hardware traps to the Operating System.
* Similarly to interrupts, trap transfers control to the interrupt vector, and when a program error occurs, the OS terminates the program abnormally.
* The same code is used for user-requested abnormal terminations.
* An appropriate error message is given and memory the program may be dumped. This might be written to a file to allow the user to examine the contents and fix the error.

## Multiprogramming and Multitasking

* **Multiprogramming** increases CPI utilisation. as well as keeping users satisfied, by organising programs so that the CPU always has one to execute. These programs are called **processes.**
* It operates in such a way that processes are kept in memory simultaneously, and when one process has to wait for a task e.g. I/O operation, the OS switches to and executes another process.
* When the external task is completed, the CPU returns to its original task. This can operate on multiple levels, such that as long as one process needs to execute, the CPU is never idle.
* **Multitasking** is a logical extension of multiprogramming, whereby the CPI executes multiple processes by switching among them, except it is done more frequently, resulting in a fast **response time**.
* Take for example a user interacting with the computer, operating at “user speeds”
* Instead of idly waiting for the user to provide the input, the CPU rapidly switches to another process.
* Managing several processes involves memory management, and **CPU Scheduling** makes the decisions as to what process it should run if several are available.
* The OS must also ensure that processes can run concurrently with limited effects on one another. This relates to process scheduling, disk storage and memory management.
* **Virtual Memory** allows the execution of processes that are not completely in memory.
* This enables users to run programs larger than actual physical memory, and **abstracts** main memory into a large, uniform array of storage, separating **logical memory** as viewed by the user from physical memory.
* This frees programmers from concern over memory-storage limitations
* Multiprogramming and multitasking also demand a file system, residing on secondary storage.
* A system must also protect resources from inappropriate use.
* To ensure orderly execution, mechanisms processing synchronisation and communication are also needed, and the OS must ensure that processes to not get stuck in deadlock.

# Resource Management

## Process Management

* **Process**: instance of a program **in execution**
* Processes need **resources** such as CPU tome, memory, files, and I/o devices to accomplish its task. These are typically allocated to a process while it is running.
* Initialisation data (input parameters) may also be needed for a process to run e.g. URL.
* Programs: **passive**, processes (running programs): **active**
* Single-threaded processes have one **program counter**.
* Execution of a process Is sequential and one at a time
* This means that although two separate processes may be associated with the same program, they have separate execution sequences.
* **Multithreaded processes**  have multiple program counters instead.
* A process can be looked at as a “unit of work” in a system. A process can be OS or user based.
* All these processes can be executed concurrently using multiplexing on a single CPU core, or in parallel across multiple CPU cores.
* The OS is responsible for the following activities regarding process management:
  + Creating and deleting system and user processes
  + Scheduling processes and threads on the CPUs
  + Suspending and resuming processes.
  + Providing mechanism for process synchronisation
  + Providing mechanism for process communication

## Memory Management

* Main memory is a large array of bytes, each with its own address, such that it is quickly accessible to from the CPU and I/0 devices.
* The CPU can read (instruction fetch) data and read & write data (data-fetch) from main memory.
* Instructions must be in main memory in order to execute them.
* The OS is following for the following activities:
  + Keeping track of which parts of memory are currently being used and which process is using them
  + Allocating and deallocating memory space and needed
  + Deciding which processes (or parts of processes) and data to move into and out of memory.

## File Management

* OS abstracts physical properties of storage to define logical storage units: **files**.
* Files are collection of related information, usually representing programs or data.
* The operating system is responsible for the following activities in connection with file management:
  + Creating and deleting files
  + Creating and deleting directories to organise files
  + Supporting primitives for manipulating files and directories
  + Mapping files onto mass storage
  + Backing up files on stable (non-volatile) storage media

## Mass-Storage Management

* The OS is responsible for the following activities regarding secondary storage management:
  + Mounting and unmounting
  + Free-space management
  + Storage allocation
  + Disk Scheduling
  + Partitioning
  + Protection

## Cache Management

* First place program checks for information before checking main memory.
* Shape, arrow

  Description automatically generatedMemory can exist in different states while being passed between storage devices.
* Must be carefully managed in multitasking system, as several CPU’s can have different copies in their caches.
* **Cache Coherency**: ensuring that updates to a file to one cache are reflected in all other caches where the file exists. Usually handled by hardware.

## I/0 System Management

* I/O subsystem consists of:
  + Memory-Management for buffering, caching and spooling
  + General Device-drive interface
  + Drivers for specific hardware devices that handles specifics of assigned device.

# Summary

* An OS is a software that manages computer hardware and provides an environment application for programs to fun.
* Interrupts are a key way in which hardware interacts with an OS. A hardware device can trigger an interrupt by sending a signal to the CPU, alerting it that some event requires attention, and the interrupt is managed by the interrupt handler
* Programs must be in main memory for a computer to execute it

# Security and Protection

* **Protection**: any mechanism for controlling the access of processes or users to the resources defined by a computer system, whereby controls can be specified and imposed.
* Prevention of certain system attacks are part of OS functionality.
* OS’s maintain list of **usernames** and **user identifiers** e.g. Windows have Security ID (SID)
* **Group identifiers** can split users into groups, providing specified functionality to sets of users.
* To **escalate privileges** some systems run a program with a different User ID which you enter upon program activation.

# Virtualisation

* **Definition**: technology allowing the abstraction of a single computer into several different execution environments.
* Using a **virtual machine** (**VM**), a user can simulate various OS’s.
* **Diagram

  Description automatically generatedEmulation** simulates computer hardware using software. Used when source CPU type is different from target CPU type.
* Virtualisationcan be used to runmultiple VM’s as shown, with a **VMM** (**Virtual Machine Manager**) managing resource use and protecting the guest from one another.

# Distributed Systems

* A collection of physical separate, possibly heterogenous computer systems, networked to provide users with access to the system’s resources.
* **Network**: communication path between two or more systems.
* Vary based on:
  + Protocols used (most common is **TCP/IP** (**Transmission Control Protocol/Internet Protocol Suite**).
  + Distance between nodes
  + Transport Media
* Different types of networks include
  + **LAN**: Local-area Network (usually connected via ethernet or Wi-fi)
  + **WAN:** Wide-area Network (between buildings, cities and countries)
  + **MAN:** Metropolitan -area Network (specifically buildings in a city)
  + **PAN:** Personal-area Network (Bluetooth between phone and headset)
* **Network Operating System**: OS that provides features like file sharing and communication between computers on network.

# Kernel Data Structures

* Include:
  + (Linked) Lists, Stacks and Queues
  + Trees
  + Hash functions and maps
  + Bitmaps (string of binary digits to represent items)

# Computing Environments

* Consists of the following types
  + **Traditional**: typical PC computing
  + **Mobile**: computing on smartphones and tablets.
  + **Client-Server**: **server** **systems** satisfy requests from **client systems**. Two different types:
    - **Compute**-**server**: Provides interface for client requests.
    - **File-server**: clients can create read, update and delete files e.g. web servers for clients with web browsers

## Peer-to-Peer (P2P):

* All nodes in system can be either servers or clients.
* There are two ways a node can determine what services are available:
  + Centralised lookup system to determine which node provides the desired service.
  + Diagram

    Description automatically generatedNodes broadcast a request and the node(s) that can provide the request respond. This approach uses a **discovery protocol**. Works as shown below

## Cloud computing

* Delivers computing storage and applications **as a service** across a network
* Several types include:
  + **Public cloud**: Available to anyone that pays for the services
  + **Private cloud**: Run by company for individual use
  + **Hybrid cloud**: includes both public and private cloud components
  + **SaaS** (Software as a Service): one ore more applications available via internet
  + **PaaS** (Platform as a Service): software stack via Internet (e.g. database server)
  + **Diagram

    Description automatically generatedIaaS** (Infrastructure as a Service): servers or storage available over Internet (e.g. backup storage)

## Real-Time Embedded Systems

* Embedded systems include car engines, manufacturing robots, optical drives, microwave ovens etc.
* Almost always run **real-time operating systems**, which is used when rigid time requirements are place on the operation or a processor of the flow of data, making it an effective control device for a dedicated application.
* Real-time systems also have defined constraints.

# Summary

* Three main purposes of an OS:
  + Provide an environment for a computer to execute programs on computer hardware in a convenient and efficient manner
  + Efficiently and fairly allocate separate resource as needed on a computer system to perform the required tasks.
  + As a control program to:
    - Supervise user programs to prevent errors and improper use of the computer
    - Manage operation and control of I/O devices
* When is it appropriate to “waste resources” and why is this not really the case?
  + Single-user systems should maximise the use of the system for the user. A GUI might “waste” CPU cycles but it optimises the user interaction with the system.
* The main difficulties for writing an OS for a real-time environment:
  + keeping the OS within the fixed time constrains of a real-time system.