# UNIVERSITY OF MINES AND TECHNOLOGY


## FACULTY OF ENGINEERING

## Computer Science and Engineering Department

# OPERATING SYSTEMS
# CE 375

Course Lecturer
**Vincent M. Nofong, Ph.D.**

November 2021

Compiled from various reference materials by:

Vincent M. Nofong, PhD.

Lecture notes for the course CE 375 (Operating Systems) for the Computer Science and Engineering Department at the University of Mines and Technology, Tarkwa - Ghana.

## COURSE ASSESSMENT

Students will be assessed in this course as follows:

1. **Continuous Assessment (40%):** Which will comprise of the following components:
   (a) **Attendance (10%):** Attendance and participation in all class and practical activities.
   (b) **Theoretical Quizzes (20%):** There will be three (3) written quizzes. Dates of quizzes will be announced in class.
   (c) **Practical Quizzes and Assignments (10%):** There will be a number of practical quizzes and one take home assignment. Dates of quizzes will be announced in class.
2. **End of Semester Examinations (60%):** This will be a written examination.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Operating Systems

An **Operating System** (**OS**) is a program that controls the execution of application programs and acts as an interface between applications and the computer hardware. An OS can also be referred to as a large complex set of system programs that control various operations of a computer system and provide a collection of services to other programs (users). Various types of operating systems however differ in the control and provision of services. For instance, while mainframe operating systems are designed primarily to optimize utilisation of hardware, mobile phone operating systems are designed to provide an environment in which users can easily interface with the computer and execute programs.

The services provided by an operating system are implemented as a large set of system functions. Typical of such system functions include scheduling of programs, memory management, device management, file management, network management, and advanced services related to protection and security.

Though various types of operating systems differ in the control and provision of services, the purpose of an operating system involves three main objectives:

- **Convenience**: An OS should provide convenient, easy-to-use, and powerful set of services to the users and the application programs in the computer system.
- **Efficiency**: An OS should manage and allow the computer resources to be used in the most efficient manner.
- **Ability to evolve**: An OS should be designed to permit effective development, testing, and introduction of new system functions without interfering with services.

A simple overview of the main computer components is shown in Figure 1.1. The computer hardware which is at the bottom consists of chips, boards, disks, a keyboard, a monitor, and similar physical objects. On top of the hardware is the software which consists of the operating system, application programs, and other system programs.

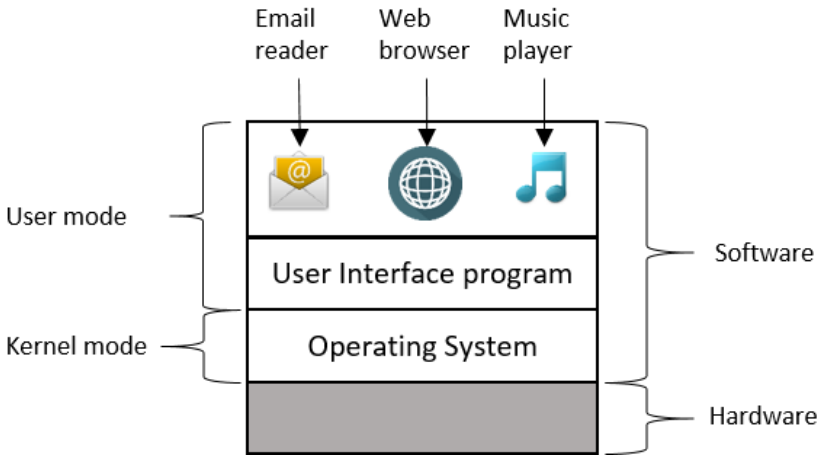



**Figure 1.1:** An External View of a Computer System

As shown in Figure 1.1, the operating system runs on the bare hardware and provides the base for all the other software. The user interface program, shell or GUI (Graphical User Interface), is the lowest level of user-mode software which allows the user to start other programs, such as a Web browser, email reader, or music player.

Most general-purpose operating systems support two general modes of operation: a.) *user mode*, and b.) *kernel mode*. A user process will normally execute in user mode while instructions such as, low-level I/O functions and memory access to special areas where the OS maintains its data structures, can execute only in kernel mode.

### 1.1.1 Operating System Interfaces

An operating system provides three general levels of interfaces through which users and application programmers can communicate with the operating system. These interfaces as depicted in Figure 1.2 serve the following purposes:

- **Graphical user interface (GUI)**: The graphical user interface allows Input/Output (I/O) interaction with a user and the operating system in a relatively easy and convenient manner through intuitive icons, menus, and other graphical objects. For example, using the click of a mouse to select an option or command.

- **Command line interpreter**: The command line interpreter (also called the shell) is a text oriented interface which allows advanced users and application programmers to communicate directly with the operating system.

- **System-call interface**: The system-call interface is used by the application programs to request the various services provided by the operating system by invoking or calling system functions.

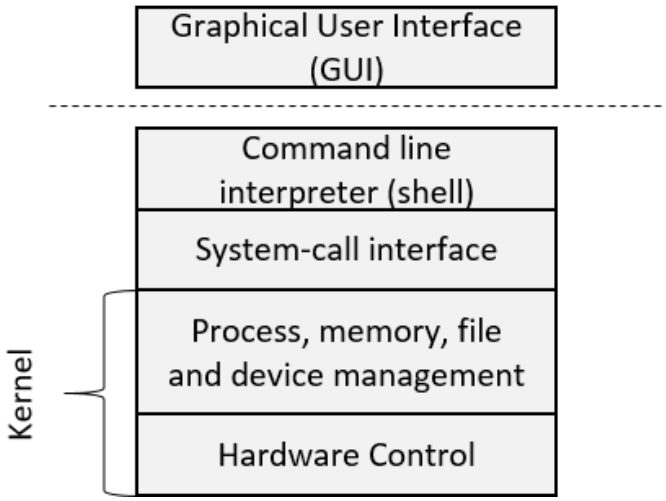



**Figure 1.2:** Basic Structure of an Operating System

## 1.2 Brief History of Operating Systems

Operating systems have evolved through a number of distinct phases or generations. The evolutions of operating systems can be summarized as follows:

1. The early computers had no operating systems. Some were binary, some used vacuum tubes and some programmable, but all were very primitive. All programming was done in absolute machine language or by wiring up electrical circuits connecting thousands of cables to plugboards to control the machine's basic functions.

2. The first types of operating systems were the simple batch operating systems in which users had to submit their jobs on punched cards or tapes. The computer operator grouped all the jobs in batches (of cards or tapes) and stacked them on the main input device (a fast card reader or tape reader). One of the most important concepts for these systems was automatic job sequencing.

3. The next types of operating systems developed were batch systems with multiprogramming. With the incorporation of multiprogramming, these systems were capable of handling several programs active in memory at any time and required more advanced memory management. When a program stopped to wait for I/O in these systems, the OS was able to switch very rapidly from the currently executing program to the next.

4. Time-sharing operating systems were the next important generation of systems developed. The most significant advantage of these systems was the capability to provide interactive computing to the users connected to the system. The basic technique employed on these systems was that the processor's time was uniformly shared by the user programs. The operating system provided CPU service during a small and fixed interval to a program and then switched to the next program.

5. Variations of multiprogramming operating systems were developed with more advanced techniques. These included improved hardware support for interrupt mechanisms and allowed implementation of better scheduling techniques based on priorities.

6. Advances in hardware and memory management techniques allowed development of operating systems with newer and more powerful features, such as paging and virtual memory, multi-level cache, and others.

In recent times, development of modern operating systems focus on networking, distribution, reliability, protection, and security. Some of the widely used operating systems available today include:

- Microsoft Windows: 98, Me, CE, 2000, XP, Vista, Windows 7, and others.
- Linux: Ubuntu, Fedora, Debian, Linux Mint, and others
- MacOS X (Apple)
- Solaris (Sun Microsystems)
- OS/2 (IBM)

## 1.3 Computer Hardware Overview

A simple personal computer can be abstracted to a model resembling that of Figure 1.3. The CPU, memory, and I/O devices are all connected by a system bus and communicate with one another over it.

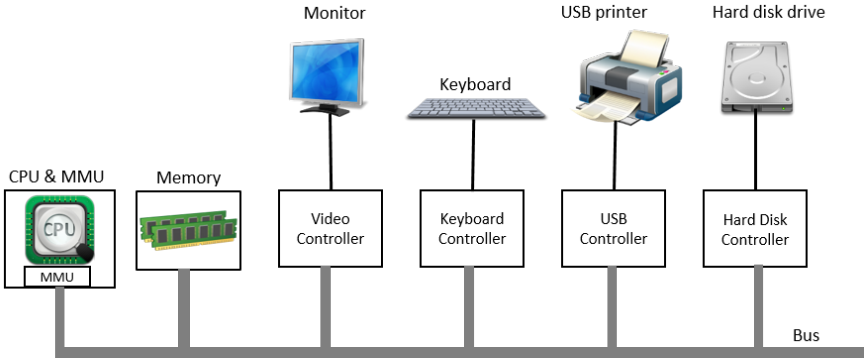



**Figure 1.3:** Abstract View of the Components of a Simple Personal Computer.

As mentioned previously, an operating system runs on and is tied to the hardware of the computer. The operating system extends the computer's instruction set and manages its resources. Hence, it must know about the hardware, at least about how the hardware appears to the programmer. As such, this section briefly reviews the components of the computer hardware as found in modern personal computers and examine some of the hardware issues that are of concern to operating system designers.

### 1.3.1 Processors

The processors control the operation of the computer and performs its data processing functions. When there is only one processor, it is often referred to as the central processing unit (CPU). The

main function of the CPU is to fetch and execute instructions from the memory. For any CPU, its basic cycle is to fetch the first instruction from memory, decode it to determine its type and operands, execute it, and then fetch, decode, and execute subsequent instructions. The cycle is repeated until the program finishes.

All CPUs contain some registers inside to hold key variables and temporary results. In addition to the general registers used to hold variables and temporary results, most computers have several special registers that are visible to the programmer. Typical of such registers include:

- **The program counter**: This contains the memory address of the next instruction to be fetched. After that instruction has been fetched, the program counter is updated to point to its successor.
- **The stack pointer**: This points to the top of the current stack in memory. The stack contains one frame for each procedure that has been entered but not yet exited. A procedure's stack frame holds those input parameters, local variables, and temporary variables that are not kept in registers.
- **The Program Status Word (PSW)**: This register contains the condition code bits, which are set by comparison instructions, the CPU priority, the mode (user or kernel), and various other control bits. User programs may normally read the entire PSW but typically may write only some of its fields.

Each CPU has specific set of instructions that it can execute. As such an x86 processor cannot execute ARM (Advanced RISC[1] Machines) programs and an ARM processor cannot execute x86 programs.

It is worth noting that, many modern CPUs have facilities for executing more than one instruction at the same time. For example, a CPU might have separate fetch, decode, and execute units, so that while it is executing instruction $n$, it could also be decoding instruction $n+1$ and fetching instruction $n+2$.

### 1.3.2 Memory

The memory, for storing data and programs, is one of the major components in any computer. Ideally, a memory should be extremely fast, of large capacity and relatively cheap in relation to other components of the computer. However, no current technology satisfies all of these goals. The design of a memory thus involve a trade-off among the three key characteristics of memory: *capacity*, *access time*, and *cost*.

The way out of this dilemma is to not rely on a single memory component or technology, but to employ a memory hierarchy. A typical memory hierarchy is illustrated in Figure 1.4.

As one goes down the hierarchy, the following occur:

1. Decreasing cost per bit,

[1] reduced instruction set computer

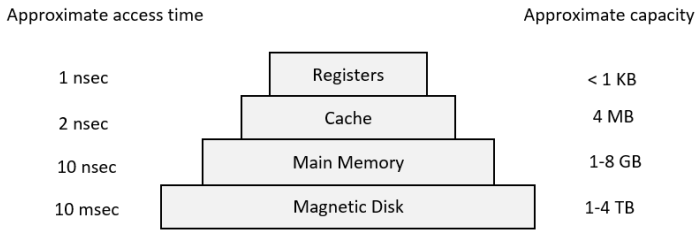



**Figure 1.4:** A Typical Memory Hierarchy - numbers are rough approximations

2. Increasing capacity,

3. Increasing access time, and

4. Decreasing frequency of access to the memory by the processor.

The top layer of the memory hierarchy consists of the registers, internal to the CPU. They are made of the same material as the CPU and are thus just as fast as the CPU. Consequently, there is no delay in accessing them. The storage capacity available in them is typically 32 × 32 bits on a 32-bit CPU and 64 × 64 bits on a 64-bit CPU. Less than 1 KB in both cases.

Next comes the cache memory, which is mostly controlled by the hardware. Cache memory is divided up into cache lines, typically 64 bytes, with addresses 0 to 63 in cache line 0, 64 to 127 in cache line 1, and so on. The most heavily used cache lines are kept in a high-speed cache located inside or very close to the CPU. When the program needs to read a memory word, the cache hardware checks to see if the line needed is in the cache. If it is, called a cache hit, the request is satisfied from the cache and no memory request is sent over the bus to the main memory. Cache hits normally take about two clock cycles while cache misses have to go to memory, with a substantial time penalty. Cache memory is limited in size due to its high cost. Some machines have two or even three levels of cache, each one slower and bigger than the one before it.

Caching plays a major role in many areas of computer science, not just caching lines of RAM. Whenever a resource can be divided into pieces, some of which are used much more heavily than others, caching is often used to improve performance. Operating systems use it all the time. For example, most operating systems keep (pieces of) heavily used files in main memory to avoid having to fetch them from the disk repeatedly.

Main memory comes next in the hierarchy of Figure 1.4. This is the workhorse of the memory system. Main memory is usually called RAM (Random Access Memory). All CPU requests that cannot be satisfied out of the cache go to main memory. In addition to the main memory, many computers have a small amount of non-volatile random-access memory. Unlike RAM, non-volatile memory does not lose its contents when the power is switched off. ROM (Read Only Memory) is programmed at the factory and cannot be changed afterwards. Often fast and inexpensive, the ROM on some computers contains the bootstrap loader used to start computers.

### 1.3.3 Disks

Disk storage is two orders of magnitude cheaper than RAM per bit and often two orders of magnitude larger as well. The only problem is that the time to randomly access data on it is close to three orders of magnitude slower. The reason is that a disk is a mechanical device, as shown in Figure 1.5.

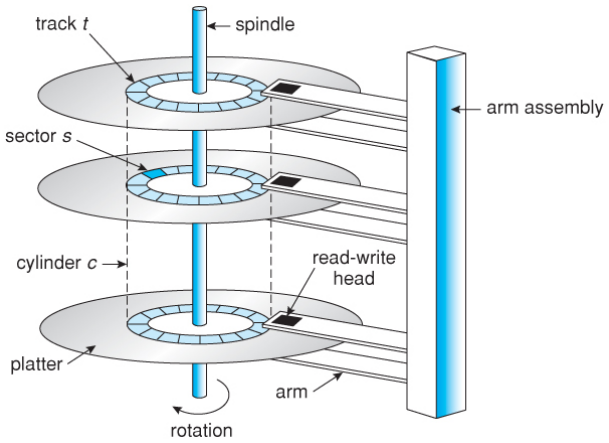



**Figure 1.5:** Structure of a Disk Drive

A disk consists of one or more metal platters that rotate at 5400, 7200, 10,800 RPM or more. A mechanical arm pivots over the platters from the corner, similar to the pickup arm on an old 33-RPM phonograph for playing vinyl records.

Information is written onto the disk in a series of concentric circles. At any given arm position, each of the heads can read an annular region called a **track**. Together, all the tracks for a given arm position form a **cylinder**. Each track is divided into some number of sectors, typically 512 bytes per sector. On modern disks, the outer cylinders contain more sectors than the inner ones. Moving the arm from one cylinder to the next takes about 1 msec. Moving it to a random cylinder typically takes 5 to 10 msec, depending on the drive.

Some disks like the Solid State Disks (SSDs) are really not disks at all. SSDs do not have moving parts, do not contain platters in the shape of disks, and store data in (Flash) memory. The only ways in which they resemble disks is that they also store a lot of data which is not lost when the power is off.

### 1.3.4 Input/Output Devices

Input/Output (I/O) devices interact heavily with the operating system. As shown in Figure 1.3, I/O devices generally consist of two parts: a controller and the device itself. The controller is a chip or a set of chips that physically control the device. It accepts commands from the operating system (for example, to read data from the device) and carries them out.

Because each type of controller is different, different software is needed to control each one. The software that talks to a controller, giving it commands and accepting responses, is called a **device driver**. Each controller manufacturer has to supply a driver for each operating system it supports.

Thus a scanner may come with drivers for OS X, Windows 8, Windows 10, and Linux, for example.

To be used, the driver has to be put into the operating system so it can run in kernel mode. Drivers can actually run outside the kernel, and operating systems like Linux and Windows nowadays do offer some support for doing so. The vast majority of the drivers still run below the kernel boundary.

There are three ways new drivers can be put into the kernel:

- One way is to re-link the kernel with the new driver and then reboot the system. Many older UNIX systems work like this.
- Another way is to make an entry in an operating system file telling it that it needs the driver and then reboot the system. At boot time, the operating system goes and finds the drivers it needs and loads them. Windows works this way.
- The final way is for the operating system to be able to accept new drivers while running and install them on the fly without the need to reboot. This way used to be rare but is becoming much more common now.

Every controller has a small number of registers that are used to communicate with it. To activate the controller, the driver gets a command from the operating system, then translates it into the appropriate values to write into the device registers. The collection of all the device registers forms the I/O port space.

Input and output can be done in three different ways:

- In the first method, a user program issues a system call, which the kernel then translates into a procedure call to the appropriate driver. The driver then starts the I/O and sits in a tight loop continuously polling the device to see if it is done. When the I/O has completed, the driver puts the data (if any) where they are needed and returns. The operating system then returns control to the caller. This method, called the busy waiting has the disadvantage of tying up the CPU polling the device until it is finished.
- The second method is for the driver to start the device and ask it to give an interrupt when it is finished. At that point the driver returns. The operating system then blocks the caller if need be and looks for other work to do. When the controller detects the end of the transfer, it generates an interrupt to signal completion.
- The third method for doing I/O makes use of special hardware: a DMA (Direct Memory Access) chip that can control the flow of bits between memory and some controller without constant CPU intervention. The CPU sets up the DMA chip, telling it how many bytes to transfer, the device and memory addresses involved, and the direction, and lets it go. When the DMA chip is done, it causes an interrupt, which is handled as described above.

### 1.3.5 Buses

A bus, in computing, is a set of physical connections (cables or printed circuits) used to connect computer components for communication or data transfer between the connected components. The main

purpose of buses is to reduce the number of pathways needed for communication between components by carrying out all commmunications over a single data channel.

Buses are often characterised by the amount of information that they can transmit at once. This amount, expressed in bits, corresponds to the number of physical lines over which data is sent simultaneously. The bus speed which is defined by its frequency (expressed in Hertz), is the number of data packets sent or received per second.

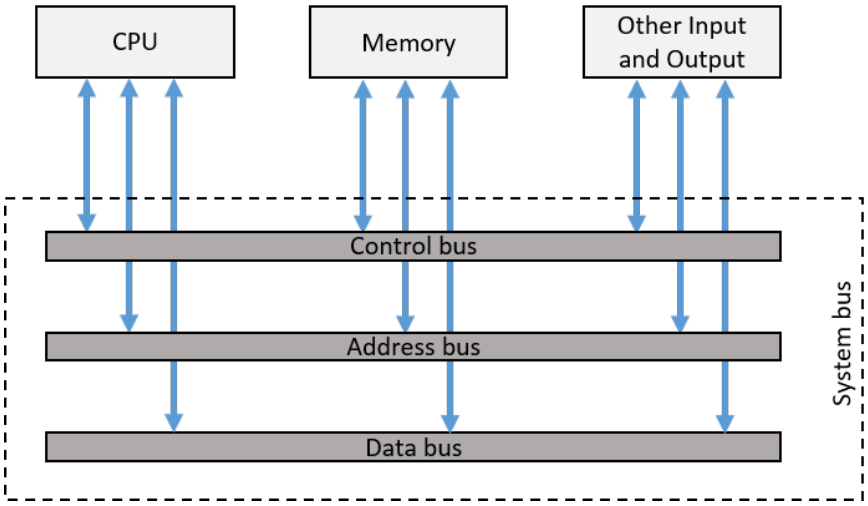



**Figure 1.6:** Typical Computer Bus Architecture

Figure 1.6 illustrates a simple architecture of the computer bus. Each bus as shown in Figure 1.6 is generally constituted of 50 to 100 distinct physical lines, divided into three sub-assemblies:

1. **The address bus**: This is a unidirectional bus for transporting memory addresses which the processor wants to access in order to read or write data. The address bus is sometimes referred to as the memory bus.
2. **The data bus**: This is a bidirectional bus which transfers instructions coming from or going to the processor.
3. **The control bus (or command bus)**: This is a bidirectional bus which transports orders and synchronisation signals coming from the control unit and travelling to all other hardware components.

There are generally two buses within a computer:

1. **The internal bus**: This allows the processor to communicate with the system's central memory (the RAM). The internal bus is sometimes called the front-side bus.
2. **The expansion bus** (sometimes called the input/output bus): This allows various motherboard components (such as USB, serial, and parallel ports, cards inserted in PCI connectors, hard drives, CD-ROM and CD-RW drives, etc.) to communicate with one another. However, it is mainly used to add new devices using what are called expansion slots connected to the input/output bus.

## 1.4 Categories of Operating Systems

Operating systems have been in existence from the first computer generation and they keep evolving with time. This section discusses some of the important types of operating systems which are most commonly used.

### 1.4.1 Batch Operating Systems

The batch operating systems process routine jobs without interacting with users. Users prepare their jobs on off-line devices like punch cards and submit the punch cards to the computer operator. The operator then sorts the programs with similar requirement into batches and stacked these on the main input device, which are routinely processed by the computer. Claims processing in an insurance company or sales reporting for a chain of stores are typically done in batch mode.

The major problems with the batch systems are as follows:

- There is no interaction between the user and the job.
- The CPU is often idle as the speed of the mechanical I/O devices are slower than the CPU.
- It is difficult to provide the desired priority.

To maximize the processor use in the batch systems, the **multiprogrammed batch operating systems** were introduced. In multiprogrammed batch systems, some jobs are selected and loaded in memory since memory is much smaller than the job pool. The CPU then picks a job from the memory and executes it instead of waiting for the job to be loaded by the mechanical device. Though this reduces the CPU idle time, several issues have to be addressed. Such issues include: which job should sit in memory, which job should the CPU pick up, how to manage memory, what happens to the job which is suspended, and so on.

### 1.4.2 Time-Shared Operating Systems

In time-shared operating systems, many people located at various terminals are able to use a particular computer system at the same time. Basically time-shared operating systems allow a processor's time to be shared among multiple users simultaneously. Multiple jobs from users in time-shared systems are executed by the CPU by switching between them, which could occur so frequently. As such each user on a time-shared system can receive an immediate response.

Effective sharing of the processor's time among multiple users in time-sharing systems is achieved by using CPU scheduling and multiprogramming to provide each user with a small portion of a time.

Some of the main advantages of time-shared operating systems include:

- A reduction in CPU idle time.
- Advantage of a quick response to jobs.

- Avoids duplication of software.

Some disadvantages of the time-shared operating systems include the problem of reliability and the question of security and integrity of user programs and data.

### 1.4.3 Distributed Operating Systems

A distributed operating system is an operating system which manages a group of independent computers making them appear as a single computer. In distributed systems, multiple processors are often used to serve multiple real-time applications and multiple users. The processors communicate with one another via various communication lines such as high-speed buses. The processors in distributed systems (which are often referred to as sites, nodes or computers) may vary in size and function.

Some of the advantages of distributed systems include:

- Resource sharing: It enables a user at one site to be able to use the resources available at another.
- Reliability: If one site fails in a distributed system, the remaining sites can potentially continue operating.
- A reduction of the load on the host computer.
- Computation speed up: It reduces the delays in data processing.

Some disadvantages of the distributed operating systems include security issues due to sharing, and losing messages in the network system.

### 1.4.4 Mainframe Operating Systems

At the high end are the operating systems for mainframes, those room-sized computers still found in major corporate data centers. These computers differ from personal computers in terms of their I/O capacity. A mainframe with 1000 disks and millions of gigabytes of data is not unusual; a personal computer with these specifications would be the envy of its friends.

The operating systems for mainframes are heavily oriented toward processing many jobs at once, most of which need prodigious amounts of I/O. They typically offer three kinds of services: batch, transaction processing, and time-sharing.

Some of the advantages of mainframe computing systems include:

- They have high-level computing powers.
- They are more reliable as they can detect, report, and self-recover from system problems.
- They have long lasting performance.

- They are more secure in storing and protecting data where confidentiality of data is critical.

Some disadvantages of mainframe computing systems include:

- They are costly as most hardware and software are expensive.
- Dedicated staff are needed to run the system.
- They can take up a lot of space and require dedicated environmental management, for instance, cooling systems.

An example mainframe operating system is OS/390, a descendant of OS/360.

### 1.4.5 Network Operating Systems

Network operating systems or server operating systems run on servers and provide the server the capability to manage users, data, groups, security and other networking functions. The primary purpose of network operating systems is to serve multiple users over a network and allow users share hardware and software resources among multiple computers in a network.

Some of the advantages of network operating systems include:

- Easy integration of upgrades to new technologies and hardware into the system.
- Ability to access servers remotely from different locations.
- Security is server managed.

Some disadvantages of network operating systems include:

- Requires regular maintenance and updates.
- Dependency on a central location for most operations.
- High cost of buying and running a server.

Examples of network operating systems include Microsoft Windows Server (2003, 2008, 2012, 2016), UNIX, Linux, Mac OS X, Novell NetWare, and BSD.

### 1.4.6 Single-User Operating Systems

A single-user operating system is a type of operating system that is developed and intended for use on a computer or similar machine that will only have a single user at any given time. This is the most common type of OS used on home computers, computers in offices and other work environments. There are two general types of single-user systems:

- **Single-user, single task operating systems**: These operating systems are designed to manage the computer so that only one user can effectively run one user application at a time. Such operating systems are specifically designed to run one application efficiently for one user. An example of an OS that is single user single task is the Palm OS which was used on mobile phones (not smart phones) and personal digital assistants (PDAs).
- **Single-user, multi-tasking operating systems**: These operating systems are designed mainly with a single user in mind, but they can deal with many applications running at the same time. These are the type of operating systems found on most personal desktop and laptop computers. Unlike the single-user, single task operating systems, these operating systems must be able to handle many different applications all running at the same time. Typical examples of single-user, multi-tasking operating systems are Windows, Linux and Mac OS

### 1.4.7 Embedded Operating Systems

Embedded operating systems run on the computers that control devices that are not generally thought of as computers and which do not accept user-installed software. Typical examples are microwave ovens, TV sets, DVD recorders, traditional phones, and MP3 players. The main property which distinguishes embedded systems from handhelds is the certainty that no untrusted software will ever run on it. You cannot download new applications to your microwave oven as all the software is in ROM. This means that there is no need for protection between applications, leading to design simplification. Systems such as Embedded Linux and QNX are popular in this domain.

### 1.4.8 Sensor-Node Operating Systems

Networks of tiny sensor nodes are being deployed for numerous purposes. These nodes are tiny computers that communicate with each other and with a base station using wireless communication. Sensor networks are used to protect the perimeters of buildings, guard national borders, detect fires in forests, measure temperature and precipitation for weather forecasting, glean information about enemy movements on battlefields, and much more.

Each sensor node is a real computer, with a CPU, RAM, ROM, and one or more environmental sensors. It runs a small, but real operating system, usually one that is event driven, responding to external events or making measurements periodically based on an internal clock. The operating system has to be small and simple because the nodes have little RAM and battery lifetime is a major issue. Also, as with embedded systems, all the programs are loaded in advance; users do not suddenly start programs they downloaded from the Internet, which makes the design much simpler. TinyOS is a well-known operating system for a sensor node.

### 1.4.9 Real-Time Operating Systems

These systems are characterized by having time as a key parameter. They are often used as a control devices in dedicated applications or when there are rigid time requirements on the operation of a processor or the flow of data. Real time operating systems must have well-defined, fixed time constraints, otherwise the system will fail.

There are two types of real-time operating systems.

- **Hard real-time systems**: Hard real-time systems must provide absolute guarantees that a certain action will occur by a certain time. In hard real-time systems, secondary storage is limited or missing and the data is stored in ROM. Many of these are found in industrial process control, avionics, military, and similar application areas.
- **Soft real-time systems**: A soft real-time systems is one where missing an occasional deadline, while not desirable, is acceptable and does not cause any permanent damage. Unlike hard real-time systems, soft real-time systems are lest restrictive. Many of these are found in digital audio or multimedia systems.

### 1.4.10 Smart Card Operating Systems

The smallest operating systems run on smart cards, which are credit-card-sized devices containing a CPU chip. They have very severe processing power and memory constraints. Some are powered by contacts in the reader into which they are inserted, but contactless smart cards are inductively powered, which greatly limits what they can do. Some of them can handle only a single function, such as electronic payments, but others can handle multiple functions.

Some smart cards are Java oriented. This means that the ROM on the smart card holds an interpreter for the Java Virtual Machine (JVM). Java applets are downloaded to the card and are interpreted by the JVM interpreter. Some of these cards can handle multiple Java applets at the same time, leading to multiprogramming and the need to schedule them. Resource management and protection also become an issue when two or more applets are present at the same time. These issues must be handled by the (usually extremely primitive) operating system present on the card.

## 1.5 Booting a Computer

Every PC contains a parentboard/motherboard. On the parentboard is a program called the system BIOS (Basic Input Output System) which contains low-level I/O software, including procedures to read the keyboard, write to the screen, and do disk I/O, among other things. Nowadays, it is held in a flash RAM, which is nonvolatile but which can be updated by the operating system when bugs are found in the BIOS. The boot process of a computer is as follows.

When the computer is booted, the BIOS is started. It first checks to see how much RAM is installed and whether the keyboard and other basic devices are installed and responding correctly. It starts out by scanning the PCIe and PCI buses to detect all the devices attached to them. If the devices present are different from when the system was last booted, the new devices are configured.

The BIOS then determines the boot device by trying a list of devices stored in the CMOS (complementary metal-oxide semiconductor) memory. The user can change this list by entering a BIOS configuration program just after booting. Typically, an attempt is made to boot from a CD-ROM (or sometimes USB) drive, if one is present. If that fails, the system boots from the hard disk. The first sector from the boot device is read into memory and executed. This sector contains a

program that normally examines the partition table at the end of the boot sector to determine which partition is active. Then a secondary boot loader is read in from that partition. This loader reads in the operating system from the active partition and starts it.

The operating system then queries the BIOS to get the configuration information. For each device, it checks to see if it has the device driver. If not, it asks the user to insert a CD-ROM containing the driver (supplied by the device's manufacturer) or to download it from the Internet. Once it has all the device drivers, the operating system loads them into the kernel. Then it initializes its tables, creates whatever background processes are needed, and starts up a login program or GUI.

# Chapter 2

# Processes and Threads

## 2.1 Processes

A process is a program in execution, ready to execute, or one that has executed partially and is waiting for other services in the computer system. In simpler terms, a process is an instantiation of a program, or an execution instance of a program. A process is a dynamic entity in the system because it exhibits behaviour (state changes), and is capable of carrying out some (computational) activity, whereas a program is a static entity.

Two basic types of processes are the system processes and the user processes. The system processes execute operating system services while the user processes execute application programs.

In most operating systems, several processes are often stored in memory at the same time and the operating system manages the sharing of one or more processors (CPUs) and other resources among the various processes. As such, process management (creating processes and controlling their execution) is one of the major functions of the operating system. To enable operating systems manage several processes at the same time, most operating systems have the multiprogramming [1] technique implemented in them.

One of the requirements of multiprogramming is that the operating system must allocate the CPUs and other system devices to the various processes in such a way that the CPUs and other devices are maintained busy for the longest total time interval possible, thus minimizing the idle time of these devices. If there is only one CPU in the computer system, then only one process can be in execution at any given time.

### 2.1.1 Process Creation

Operating systems need some way to create processes. In very simple systems, or in systems designed for running only a single application (e.g., the controller in a microwave oven), it may be possible to have all the processes that will ever be needed be present when the system comes up. In general-

[1] This is a form of parallel processing in which several programs are run at the same time on a processor

purpose systems, however, some way is needed to create and terminate processes during operation.

Four principal events which cause processes to be created are:

1. System initialization.
2. Execution of a process-creation system call by a running process.
3. A user request to create a new process.
4. Initiation of a batch job.

When an operating system is booted, typically numerous processes are created. Some of these are foreground processes, that is, processes that interact with (human) users and perform work for them. Others run in the background and are not associated with particular users, but instead have some specific function. For example, one background process may be designed to accept incoming email, sleeping most of the day but suddenly springing to life when an email arrives.

In addition to the processes created at boot time, new processes can be created afterwards as well. Often, a running process will issue system calls to create one or more new processes to help it do its job. Creating new processes is particularly useful when the work to be done can easily be formulated in terms of several related, but otherwise independent interacting processes. On a multiprocessor, allowing each process to run on a different CPU may also make the job go faster.

In interactive systems, users can start a program by typing a command or (double) clicking on an icon. Taking either of these actions starts a new process and runs the selected program in it. In command-based UNIX systems running X, the new process takes over the window in which it was started. In Windows, when a process is started it does not have a window, but it can create one (or more) and most do. In both systems, users may have multiple windows opened at once, each running some process. Using the mouse, the user can select a window and interact with the process, for example, providing input when needed.

The last situation in which processes are created applies only to the batch systems found on large mainframes. Here users can submit batch jobs to the system (possibly remotely). When the operating system decides that it has the resources to run another job, it creates a new process and runs the next job from the input queue in it.

Technically, in all these cases, a new process is created by having an existing process execute a process creation system call. That process may be a running user process, a system process invoked from the keyboard or mouse, or a batch-manager process. What that process does is execute a system call to create the new process. This system call tells the operating system to create a new process and indicates, directly or indirectly, which program to run in it.

In UNIX, there is only one system call to create a new process: **fork()**. This call creates an exact clone of the calling process. After the **fork()**, the two processes, the parent and the child, have the same memory image, the same environment strings, and the same open files. The child process then executes **execve()** or a similar system call to change its memory image and run a new program.

In Windows, in contrast, a single Win32 function call, **CreateProcess()**, handles both process creation and loading the correct program into the new process. This call has 10 parameters, which

include the program to be executed, the command-line parameters to feed that program, various security attributes, bits that control whether open files are inherited, priority information, a specification of the window to be created for the process (if any), and a pointer to a structure in which information about the newly created process is returned to the caller. In addition to **CreateProcess()**, Win32 has about 100 other functions for managing and synchronizing processes and related topics.

In both UNIX and Windows systems, after a process is created, the parent and child have their own distinct address spaces. If either process changes a word in its address space, the change is not visible to the other process. In UNIX, the child's initial address space is a copy of the parent's, but there are definitely two distinct address spaces involved; no writeable memory is shared.

### 2.1.2 Process Termination

After a process has been created, it starts running and does whatever its job is. However, nothing lasts forever, not even processes. Sooner or later the new process will terminate, usually due to one of the following conditions:

1. Normal exit (voluntary).
2. Error exit (voluntary).
3. Fatal error (involuntary).
4. Killed by another process (involuntary).

Most processes terminate because they have done their work. When a compiler has compiled the program given to it, the compiler executes a system call to tell the operating system that it is finished. This call is **exit()** in UNIX and **ExitProcess()** in Windows. Screen-oriented programs also support voluntary termination. Word processors, Internet browsers, and similar programs always have an icon or menu item that the user can click to tell the process to remove any temporary files it has open and then terminate.

The second reason for termination is that the process discovers a fatal error. For example, if a user types the command *javac HelloWorld.java* to compile the program *HelloWorld.java* and no such file exists, the compiler simply announces this fact and exits. Screen-oriented interactive processes generally do not exit when given bad parameters. Instead they pop-up a dialog box and ask the user to try again.

The third reason for termination is an error caused by the process, often due to a program bug. Examples include executing an illegal instruction, referencing nonexistent memory, or dividing by zero. In some systems (e.g., UNIX), a process can tell the operating system that it wishes to handle certain errors itself, in which case the process is signalled (interrupted) instead of terminated when one of the errors occurs.

The fourth reason a process might terminate is that the process executes a system call telling the operating system to kill some other process. In UNIX this call is **kill()**. The corresponding Win32 function is **TerminateProcess()**. In both cases, the killer must have the necessary authorization

to do in the process to be killed. In some systems, when a process terminates, either voluntarily or otherwise, all processes it created are immediately killed as well.

### 2.1.3 Process Hierarchies

In some systems, when a process creates another process, the parent process and child process continue to be associated in certain ways. The child process can itself create more processes, forming a process hierarchy.

In UNIX, a process and all of its children and further descendants together form a process group. When a user sends a signal from the keyboard, the signal is delivered to all members of the process group currently associated with the keyboard (usually all active processes that were created in the current window). Individually, each process can catch the signal, ignore the signal, or take the default action.

In contrast, Windows has no concept of a process hierarchy. All processes are equal. The only hint of a process hierarchy is that when a process is created, the parent is given a special token (called a handle) that it can use to control the child. However, it is free to pass this token to some other process, thus invalidating the hierarchy. Processes in UNIX cannot disinherit their children.

### 2.1.4 Process States

The process manager controls the execution of the various processes in the system. Processes change from one state to another during their lifetime in the system. From a high level of abstraction, processes exhibit their behaviour by changing from one state to the next. The state changes are really controlled by the operating system. For example, when the process manager blocks a process because it needs a resource that is not yet available, the process changes from the running state to the waiting for resource state. When a process is waiting for service from the CPU, it is placed in the ready queue and is in the ready state. In a similar manner, when a process is waiting for I/O service, it is placed in one of several I/O queues and it changes to a wait state.

In general, a process can have one of the following five states at a time:

1. **Start State**: This is the initial state when a process is first started/created.

2. **Ready**: The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after Start state or while running it gets interrupted by the scheduler to assign CPU to some other process.

3. **Running**: Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.

4. **Waiting**: Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.

5. **Terminated** or **Exit**: Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.

A state diagram represents the various states of a process and the possible transitions in the behaviour of a process. Figure 2.1 shows the state diagram of a typical process. Each state is indicated by an ellipse and the arrows connecting the states represent transitions from one state to the next. Normally, a process spends a finite time in any of its states.

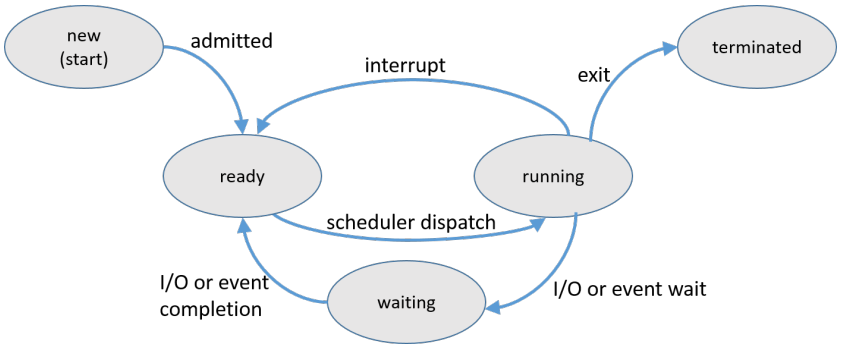



**Figure 2.1:** Diagram of Process States

When a job arrives, and if there are sufficient resources (such as memory) available, a corresponding process is created. In a time-sharing system, when a user logs in, a new process is created. After being created, the process becomes ready and waits for CPU service. It eventually receives this service, then waits for I/O service, and at some later time receives I/O service. The process then goes back to the ready state to wait for more CPU service. The state in which a process is waiting for I/O service is normally called the blocked state. After satisfying all service requests, the process terminates.

Another possible wait state is defined when the process is swapped out to disk if there is not sufficient memory. This means that the operating system can move a process from memory to disk and have the process wait in a suspended state; at some other time the process is moved back to memory.

### 2.1.5 Process Control Block (PCB)

A Process Control Block (sometimes referred to as process table) is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). For each process, its PCB keeps all the information needed to keep track of that process. Typical information kept in the PCB include:

- Process State: The current state of the process i.e., whether it is ready, running, waiting, or whatever.
- Process privileges: This is required to allow/disallow access to system resources.
- Process ID: Unique identification for each of the processes in the operating system.

- Pointer: A pointer to parent process.
- Program Counter: Program Counter is a pointer to the address of the next instruction to be executed for this process.
- CPU registers: Various CPU registers where the process need to be stored for execution.
- CPU Scheduling Information: Process priority and other scheduling information which is required to schedule the process.
- Memory management information: This includes the information memory limits to be used by the operating system.
- Accounting information: This includes the amount of CPU used for process execution, time limits, execution ID etc.
- I/O status information: This includes a list of I/O devices allocated to the process.

The PCB of each process is maintained throughout its lifetime, and is deleted once the process terminates. Figure 2.2 illustrates the structure of the process control block.

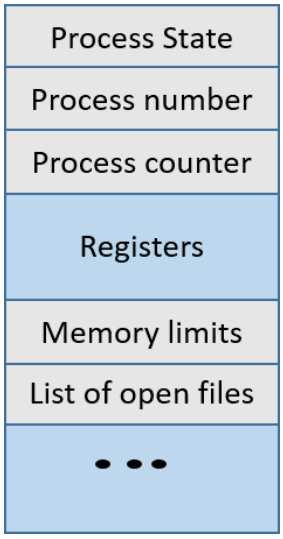



**Figure 2.2:** Process Control Block (PCB)

To implement the process model, the operating system maintains a table (an array of structures), called the process table, with one entry per process. Some authors call these entries process control blocks. This entry contains important information about the process' state, including its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information, and everything else about the process that must be saved when the process is switched from running to ready or blocked state so that it can be restarted later as if it had never been stopped.

## 2.2 Threads

A thread often called a lightweight process or mini-process, is a dynamic component of a process. Several threads are usually created within a single process where each thread belongs to exactly one

process and no thread can exist outside a process. These threads share part of the program code and the resources that have been allocated to the process.

Each thread has its own attributes such as: *execution state, context* (the program counter within the process), *execution stack, local memory block, and, a reference to the parent process* (to access the shared resources allocated to the process). For any created thread, the operating system uses a data structure (thread descriptor) to store all the relevant data of a thread. A typical thread descriptor contains the following fields: *thread identifier, execution state of the thread, process owner of the thread, list of related threads, execution stack, thread priority, and, thread specific resources.*

### 2.2.1 Thread Usage

There are several reasons for having threads, some of which include:

- Threads which are lighter weight than processes, are easier (faster) to create and destroy than processes. This is because threads share the code and resources of the process, there is no need to change the code and resources from one thread to another thread in the same process.
- Threads yield no performance gain when all of them are CPU bound, but when there is substantial computing and also substantial I/O, having threads allows these activities to overlap, thus speeding up the application.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.
- Use of threads provides concurrency within a process.

There are two general types of threads: user-level threads and kernel-level threads.

### 2.2.2 User-Level Threads

User-level threads (ULT) are found in the user space. Thread management is carried out at the level of the application without kernel intervention. As far as the kernel is concerned, it is managing ordinary, single-threaded processes. When threads are managed in user space, each process needs its own private thread table to keep track of the threads in that process.

Figure 2.3 illustrates a user-level threads in an operating system.

Some of the advantages of user-level threads include:

- User level threads are fast to create and manage.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- Thread switching does not require Kernel mode privileges.

Some disadvantages of the user-level threads include:

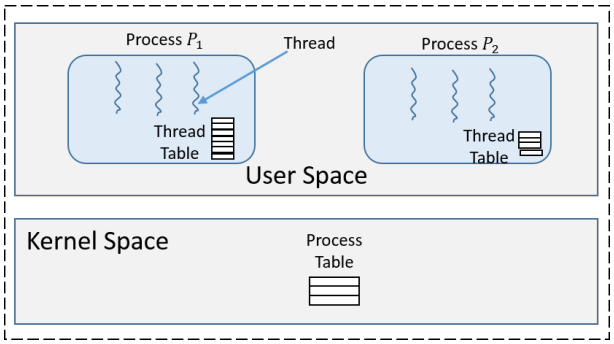



**Figure 2.3:** User-Level Threads

- When a thread starts running, no other thread in that process will ever run unless the first thread voluntarily gives up the CPU.
- Multithreaded application cannot take advantage of multiprocessing.

### 2.2.3 Kernel-Level Threads

With kernel-level threads (KLT), the thread management tasks are carried out by the kernel. A process that needs thread handling services has to use the system call interface of the kernel thread facility. The kernel maintains all the information for the process and its threads.

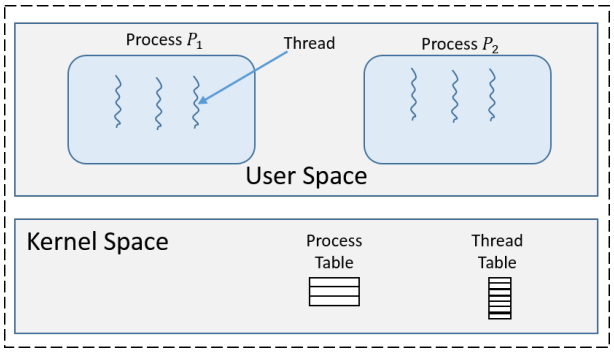



**Figure 2.4:** Kernel-Level Threads

Some advantages of the kernel-level threads include:

1. The kernel can simultaneously schedule multiple threads from the same process on multiple processors.
2. If one thread in a process is blocked, the Kernel can schedule another thread of the same process.

Some disadvantages of the kernel-level threads include:

1. Kernel threads are generally slower to create and manage than the user threads.

2. Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

### 2.2.4 Hybrid Level Threads

Various ways have been investigated to try to combine the advantages of user-level threads with kernel-level threads. Typical examples of such combinations are as described below.

#### 2.2.4.1 Many to Many Model

The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads as shown in Figure 2.5.

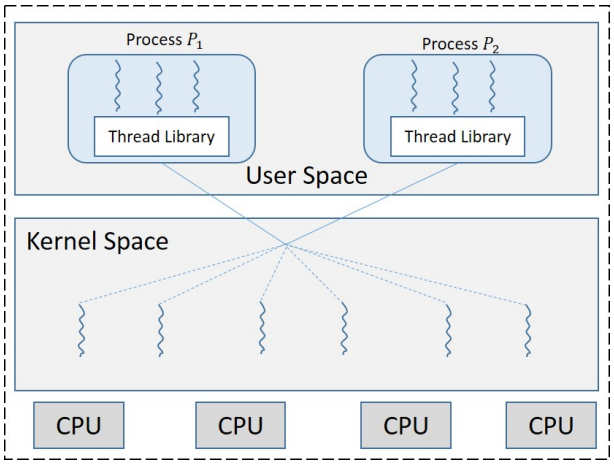



**Figure 2.5:** Many to Many Thread Model

In the many-to-many model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model provides the best accuracy on concurrency. With the many to many model, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

#### 2.2.4.2 Many to One Model

In the many-to-one model, many user-level threads are mapped to one Kernel-level thread as shown in Figure 2.6.

Thread management is done in user space and unlike the many-to-many model, when thread makes a blocking system call, the entire process will be blocked. In this model, only one thread can access the Kernel at a time and multiple threads are unable to run in parallel on multiprocessors.

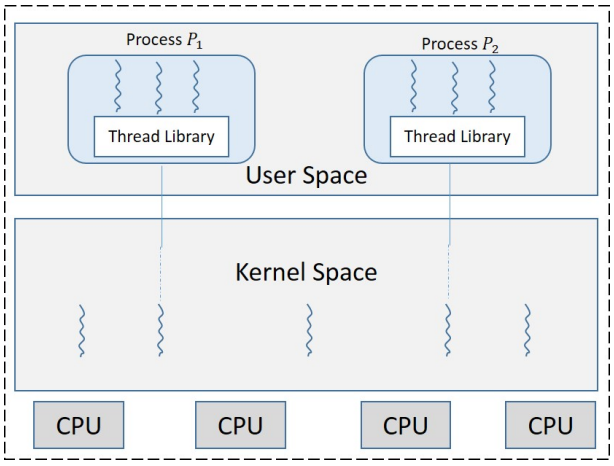



**Figure 2.6:** Many to One Thread Model

#### 2.2.4.3 One to One Model

This model provides more accuracy for concurrency. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.

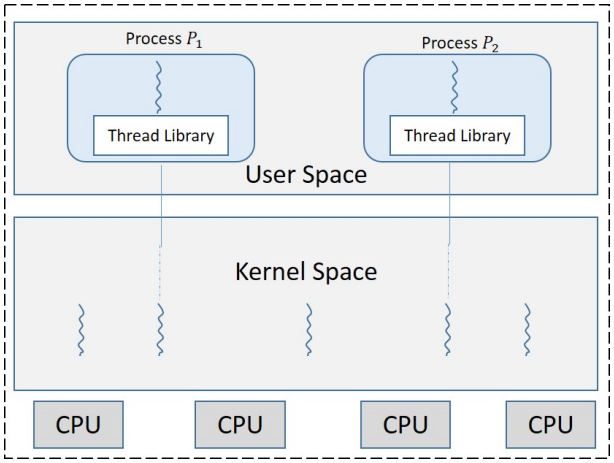



**Figure 2.7:** One to One Thread Model

The disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, Windows NT and Windows 2000 use one to one relationship model.

### 2.2.5 Pthreads

Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization. This is a specification for how a thread behaves to ensure they can be used across different platforms. Systems which implement the Pthreads specification are mostly the UNIX-type systems such as: Linux, Mac OS X, and Solaris. Windows doesn't support Pthreads but some third-party implementations for Windows are available.

### 2.2.6 Interprocess Communication

The processes executing in any operating system may be either **independent processes** or **cooperating processes**. Independent processes are processes that do not share data with other processes, and, cannot affect or be affected by the other processes executing in the system. Cooperating processes on the other hand are processes that share data with other processes, and can affect or be affected by the other processes executing in the system.

There are several reasons for providing an environment that allows process cooperation:

- Computation speed-up: To speed-up a task (process), it can be broken into subtasks to enable them run in parallel (possible on computers with multiple processing cores). Such a speed-up will be difficult without interprocess communication.
- Convenience: To allow an individual user work on many tasks at the same time. For instance, editing a word document, listening to music, and compiling in parallel.
- Information sharing: To allow several users (or processes) interested in a shared information (e.g. a shared file) have concurrent access to to such information.
- Modularity: To ensure that systems can be constructed in a modular fashion by dividing the system functions into separate processes or threads.

Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication: shared memory and message passing, as discussed below.

#### 2.2.6.1 Shared Memory Systems

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space.

Since the operating system tries to prevent one process from accessing another process's memory. Shared memory systems requires that two or more processes agree to remove this restriction allowing the exchange information between the processes involved (by reading and writing data in the shared areas). The form of the data and the location are determined by these processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

#### 2.2.6.2 Message-Passing Systems

Message passing systems provide a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. Message passing systems are particularly useful in distributed environments, where the communicating processes may reside on different computers

connected by a network. Messages sent by a process can be either fixed or variable in size. A message-passing facility provides at least two operations: *send(message)* and *receive(message)*.

**Shared Memory Systems vs Message Passing Systems**

- Message passing is useful for exchanging smaller amounts of data
- Message passing models are easier to implement in a distributed system than shared memory models.
- Shared memory can be faster than message passing as message passing requires kernel intervention (more time consuming)

# Chapter 3

# Synchronization and Deadlocks

## 3.1 Process Synchronization

We learnt in the previous chapter that a *cooperating process* is one that can affect or be affected by other processes executing in the system. Cooperating processes as discussed previously can either directly share a logical address space or be allowed to share data only through files or messages. Concurrent access to shared data by cooperating processes may however result in data inconsistency. This following subsections discuss the various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

### 3.1.1 The Race Condition

Given that cooperating processes are allowed to concurrently manipulate the same data, the outcome of the execution depends on the particular order in which the access takes place. A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a *race condition*. To guard against the race condition above, there is the need to ensure that only one process at a time can be manipulating the in its *critical section* manipulating the shared data. To make such a guarantee, it thus become a requirement for processed to be synchronized in some way.

### 3.1.2 The Critical-Section Problem

The critical section of a process is a section of code in which a process accesses shared resources. The *critical-section problem* is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section. The general structure of a typical process is shown in Figure 3.1.

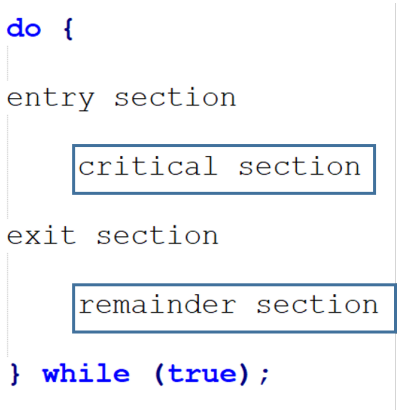



**Figure 3.1:** General Structure of a Typical Process

#### 3.1.2.1 Solutions to the Critical Section Problem

A solution to the critical-section problem must satisfy the following three requirements:

1. *Mutual exclusion*: If process $P_i$ is executing in its critical section, then no other processes can be executing in their critical sections. Under mutual exclusion, the critical section protocol should be capable of blocking processes that wish to enter the critical section once $P_i$ is in its critical section. Additionally the entry protocol must be able to know when $P_i$ exits the critical section and allows a waiting process to enter.

2. *Progress*: If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. *Bounded waiting*: There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted. The bounded waiting will ensure that processes blocked by other waiting processes, will not be waiting forever.

### 3.1.3 Approaches for Implementing Synchronization

There are several approaches for implementing solutions to the critical section problem (synchronization) which can be grouped into two categories: *hardware* and *software* approaches.

#### 3.1.3.1 Hardware Approaches

The hardware approaches for implementing synchronization are based on the premise of *locking* — that is, protecting critical regions through the use of *locks*. Some of the hardware approaches of implementing synchronizations are discussed below.

- *Disabling and Re-enabling Interrupts*: With this approach, a process disables all interrupts just after entering critical section and re-enables them just before leaving critical section. With interrupts disabled, process can examine and update shared global resource without any other process interfering. Disabling and re-enabling the interrupts will ensure no interference when a process is in its critical section as the CPU only switches from one process to another as a result of clock or other interrupts - which would have been disabled. Unfortunately, this solution which is feasible in single processor environment is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decrease. Additionally, with this approach, a process waiting to enter its critical section could suffer from starvation.

### 3.1.4 Software Approaches

The hardware-based solutions to the critical-section problem presented are complicated and often not inaccessible to application programmers. As such, operating-systems designers build software tools to solve the critical-section problem. Some of the software approaches for solving the critical-section problem are discussed below.

- *Mutexes*: A mutex[1] is a shared variable that can be in one of two states: *locked* and *unlocked*. The mutex lock is used to protect critical regions and thus prevent race conditions. That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section. The *acquire()* function acquires the lock, and the *release()* function releases the lock, as illustrated in Figure 3.2.

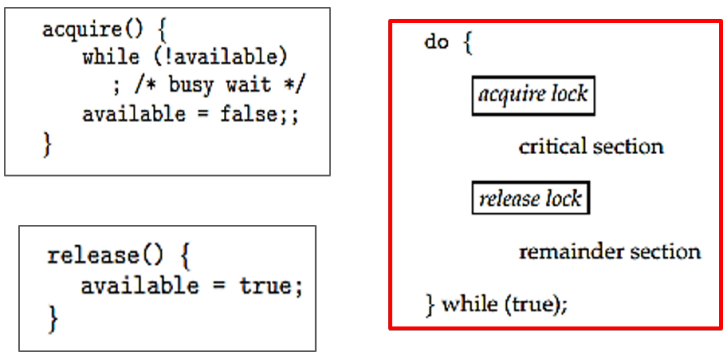



**Figure 3.2:** Mutex Functions and Solution to the Critical-Section Problem using Mutex Locks

Mutexes are good only for managing *mutual exclusion* to some shared resource or piece of code. They are easy and efficient to implement, hence useful in thread packages that are implemented entirely in user space. The main disadvantage of using mutexes is that they require busy waiting. That is, while a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to *acquire()*.

- *Semaphores*: A semaphore $S$ is an integer variable that is accessed only through two standard atomic operations[2] *wait()* and *signal()*. As shown in Figure 3.4, the *wait*() tests the value of its

[1] The term mutex is short for mutual exclusion.)
[2] Appears to the rest of the system to occur instantaneously.

argument, $S$ and decrement its value while the *signal()* increments the value of its argument, $S$ as an individual operation.

```
public void wait(){

    if S > 0 then
    S--;
    else
        place process into waiting queue;
        sleep();
}
```

```
public void signal(){
    S++;
    if queue is non-empty then
    remove a process from waiting queue;
    wakeup process;
}
```

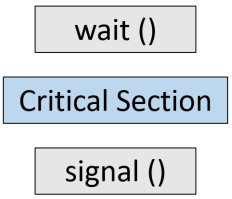



**Figure 3.3:** Semaphore Functions and Solution to Critical-Section Problem using Semaphores

To implement a solution to the critical section problem, the critical section in every process is enclosed between the *wait()* and the *signal()* operations as shown in Figure 3.4.

Every process follows this sequence of instructions:

1. Invoke the *wait* operation, which tests the value of its integer attribute $S$.
   (a) If the value of $S$ is greater than 0, it is decremented and the process is allowed to proceed to enter its critical section.
   (b) If the value of $S$ is zero, the wait operation suspends the process and places it in the semaphore queue.
2. Execute the critical section of the process.
3. Invoke the signal operation, which increments the value of the attribute $S$ and re-activates the process waiting at the head of the semaphore queue.
4. Continue executing the normal sequence of instructions; the reactivated process will become the current process.

Generally there are two types of semaphores:

1. The *binary semaphore*: whose integer attribute S can take only two values, 0 or 1. These are mostly used to implement mutual exclusion.
2. The *counting semaphore*: whose integer attribute S can take any non-negative integer value. These are mostly used to implement bounded concurrency.

Semaphores are simple and work with many processes. Additionally, they can permit multiple processes into the critical section at once, if desirable. However, with improper use of semaphores, a process may block indefinitely. Another challenge with using semaphores is priority inversion.

- *Monitors*: A monitor is a programming language construct (syntax) that controls access to shared data. These are synchronization codes added by compiler and enforced at runtime to protect data from unstructured access. Monitors guarantee that processes/threads accessing its data through its procedures interact only in legitimate ways. They have an important property

that make them useful for achieving mutual exclusion "only one process can be active in a monitor at any instant".

Monitors support synchronization by the use of condition variable defined with two operations: *wait* and *signal*. A process that invokes the:

- wait operation on condition variable $x$, releases mutual exclusion, places itself on the condition queue for $x$, and suspends itself.
- signal operation on condition variable $x$, and re-activates one waiting process from the condition queue of $x$; the current process eventually releases mutual exclusion and exits the monitor. The reactivated process re-evaluates the condition to continue execution.

Monitors are simpler to use than semaphores because they handle all of the details of lock acquisition and release. Additionally, unlike semaphores, monitors automatically acquire the necessary locks.

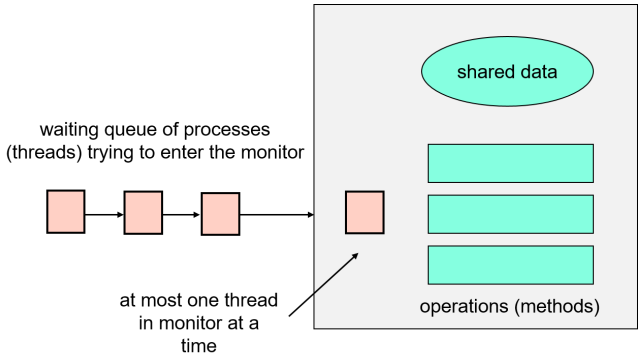



**Figure 3.4:** Solution to Critical-Section Problem using Monitors

## 3.2 Deadlocks

#### 3.2.0.1 Resources

A resource can be a hardware device (e.g., a DVD-drive) or a piece of information (e.g., a record in a database). A computer will normally have many different resources that a process can acquire. For some resources, several identical instances may be available, such as three DVD-drives. When several copies of a resource are available, any one of them can be used to satisfy any request for the resource. In short, a resource is anything that must be acquired, used, and released over the course of time. Resources can be categorised as either:

1. Preemptable resource: A resource that can be taken away from the process owning it with no ill effects. - E.g memory
2. Non-preemptable resource: A resource that cannot be taken away from its current owner without potentially causing failure - E.g. DVD-drive

The abstract sequence of events required to use a resource is given below:

1. Request the resource
2. Use the resource
3. Release the resource

If the resource is not available when it is requested, the requesting process is forced to wait. In some operating systems, the process is automatically blocked when a resource request fails, and awakened when it becomes available. In other systems, the request fails with an error code, and it is up to the calling process to wait a little while and try again. A process whose resource request has just been denied will normally sit in a tight loop requesting the resource, then sleeping, then trying again. Although this process is not blocked, for all intents and purposes it is as good as blocked, because it cannot do any useful work.

### 3.2.1 Basic Principles of Deadlock

A deadlock is the state of indefinite waiting that processes may reach when competing for system resources or when attempting to communicate. Deadlock can also be defined as the permanent blocking of a set of processes that either compete for system resources or communicate with each other.

Deadlock is possible if the following four conditions are present simultaneously:

1. Mutual exclusion: Only one process can access a resource at a time and the process has exclusive use of the resource.
2. Hold and wait: A process has acquired one or more resources and is requesting other resources.
3. Circular wait: There must be a circular list of two or more processes, each of which is waiting for a resource held by the next member of the chain.
4. No preemption: A process cannot be interrupted and forced to release resources.

The presence of these four conditions does not imply there is a deadlock. These conditions are necessary but not sufficient for deadlock to occur. That is, even if the four conditions are present, deadlock may not actually occur. The presence of the four conditions however imply that there is **potential** for a deadlock.

#### 3.2.1.1 Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**. This graph consists of a set of vertices $V$ and a set of edges $E$. The set of vertices $V$ is partitioned into two different types of nodes: $P = P_1, P_2, \ldots, P_n$, the set consisting of all the active processes in the system, and $R = R_1, R_2, ..., R_m$, the set consisting of all resource types in the system.

A directed edge from process $P_i$ to resource type $R_j$ is denoted by $P_i \rightarrow R_j$; it signifies that process $P_i$ has requested an instance of resource type $R_j$ and is currently waiting for that resource. A

directed edge from resource type $R_j$ to process $P_i$ is denoted by $R_j \rightarrow P_i$; it signifies that an instance of resource type $R_j$ has been allocated to process $P_i$ . A directed edge $P_i \rightarrow R_j$ is called a **request edge**; a directed edge $R_j \rightarrow P_i$ is called an **assignment edge**. Pictorially, we can represent each process $P_i$ as a ellipse and each resource type $R_j$ as a rectangle as shown in Figure 3.5 below.

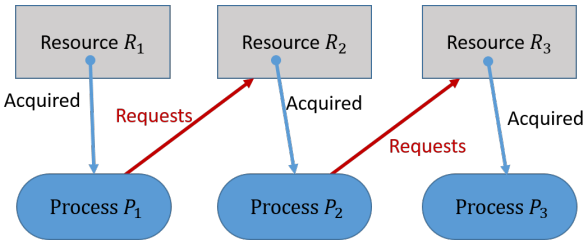



**Figure 3.5:** Resource Allocation Graph

### 3.2.2 Methods for Handling Deadlock

Generally deadlocks can be dealt with in one of four ways:

1. By ignoring the problem altogether and pretend that deadlocks never occur in the system.
    - Often referred to as the Ostrich Algorithm, this is the simplest solution for handling deadlocks. This approach is often used when the cost of the solution is not justified by the magnitude of the problem to be fixed. For instance if deadlocks occur on the average once every five years, but system crashes due to hardware failures and operating system bugs occur once a week, most engineers would not be willing to pay a large penalty in performance or convenience to eliminate deadlocks.
2. By allowing the system to enter a deadlocked state, detect it, and recover.
    - This is often used in systems which do not employ deadlock prevention or avoidance algorithms. With this approach, the system is allowed to enter into a state of deadlock and algorithms are used to recover from the deadlock.
3. By dynamically avoiding deadlocks through careful resource allocation.
    - This often employs the use of algorithms to get additional information concerning the resources a process will request during its lifetime. Based on this the operating system can decide for each request whether or not the process should wait, hence avoiding a possible deadlock.
4. By structurally negating one of the deadlock conditions to prevent its occurrence.
    - Since there are four conditions which if present can lead to a possible deadlock, this method simply ensures at least one of the necessary conditions does to occur by constraining how request for resources can be made, hence preventing a potential deadlock.

The below subsections give brief elaborations on the prevention, avoidance and recovery from deadlocks.

### 3.2.3 Deadlock Prevention

As mentioned in Section 3.2.1, for a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, will thus prevent the occurrence of a deadlock. We elaborate on this approach by examining each of the four necessary conditions separately.

- **Mutual Exclusion**: The mutual exclusion condition must hold. That is, at least one resource must be non-shareable. Shareable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a shareable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a shareable resource. In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-shareable. For example, a mutex lock cannot be simultaneously shared by several processes.

- **Hold and Wait**: To ensure that the hold-and-wait condition never occurs in the system, two techniques can be used to achieve this: (a) a process must request and acquire all the resources it needs before starting to use acquired resources; and (b) a process must release all resources it has acquired before requesting any new resources.

  To illustrate the difference between these two protocols, consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end. The second method allows the process to request initially only the DVD drive and disk file. It copies from the DVD drive to the disk and then releases both the DVD drive and the disk file. The process must then request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates. Both these protocols have two main disadvantages. First, resource utilization may be low, since resources may be allocated but unused for a long period. Second, starvation is possible as a process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

- **Circular Wait**: A circular wait condition exists in a collection of processes $P_l$ to $P_n$: If process $P_l$ is waiting for a resource held by $P_2$, process $P_2$ is waiting for a resource held by $P_3$, process $P_3$ is waiting for a resource held by $P_n$, and process $P_n$ is waiting for a resource held by $P_l$. Figure 3.6 illustrates a typical circular wait condition.

  To disallow the circular wait condition, a total ordering is assigned to all the resource types in the system where each resource type can be assigned a unique integer number, or an index number. With the ordering, a process is allowed to acquire a resource, $R_k$, if $R_k > R_j$ for some $R_j$ that the process already holds. If process $P_i$ requests $R_j$, while holding $R_k$ such that $R_k > R_j$, the system would not allow the allocation of resources in this order. Process $P_i$ must release $R_k$ to acquire $R_j$ and before acquiring $R_k$.

  Using Figure 3.7 to illustrate the above, process $P_1$ in Scenario A cannot acquire resource $R_1$ since it has a lower order than resource $R_3$ (which is currently held by process $P_1$), however, process $P_1$ in Scenario B can acquire resource $R_3$ since it has a higher order than resource $R_1$ (which is currently held by process $P_1$).

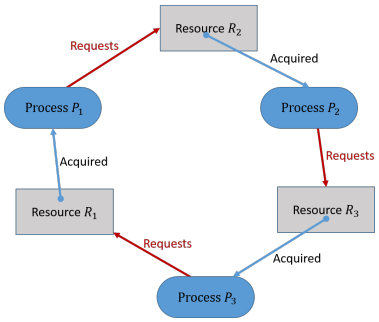



**Figure 3.6:** Typical Circular Wait Condition

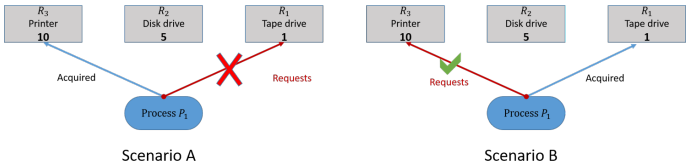



**Figure 3.7:** Disallowing Circular Wait Condition

- **No Preemption**: To ensure that this condition does not hold, the following protocol can be used. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted. That is, these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

  Alternatively, if a process requests some resources, they must first be checked whether they are available. If they are, they are allocated to the process. If they are not, they are checked to see whether they are allocated to some other process that is waiting for additional resources. If so, the desired resources from the waiting process are preempted and allocated to the requesting process. If the resources are neither available nor held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be preempted, but only if another process requests them. A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting. This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space. It cannot generally be applied to such resources as mutex locks and semaphores.

### 3.2.4 Deadlock Avoidance

Deadlock avoidance techniques allow a system to change state by allocating resources only when it is certain that deadlock will not occur by subsequent resource allocations. The goal of deadlock avoidance techniques is to determine if there is some sequence of resource requests, allocations, and deallocations that allow every process to eventually complete. For example, in a system with one tape drive and one printer, the system might need to know that process $P_1$ will request first the tape drive and then the printer before releasing both resources, whereas process $P_2$ will request first the printer and then the tape drive.

If there is no sequence of resource requests, then the resource allocation state is said to be an unsafe state - a potential for deadlock to occur. To avoid deadlock, the system is required to allocate resources in some order to the requesting processes, not simultaneously meet all maximum claims (the total number of resource instances that a process will ever request) of the processes.

### 3.2.5 Deadlock Detection and Recovery

When a detection algorithm determines that a deadlock exists, several alternatives are available. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually. Another possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock. One is simply to abort one or more processes to break the circular wait. The other is to preempt some resources from one or more of the deadlocked processes.

#### 3.2.5.1 Process Termination

To eliminate deadlocks by aborting a process, one of two methods can be used. In both methods, the system reclaims all resources allocated to the terminated processes.

- Abort all deadlocked processes: This method clearly will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
- Abort one process at a time until the deadlock cycle is eliminated: This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job.

#### 3.2.5.2 Resource Preemption

To eliminate deadlocks using resource preemption, some resources are successively preempted from processes and given to other processes until the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. Selecting a victim: Which resources and which processes are to be preempted? As in process termination, the order of preemption to minimize the cost must be determined. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.
2. Roolback: If a resource is preempted from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. The

process must roll back to some safe state and restart it from that state. Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.

3. Starvation: How do to ensure that starvation will not occur? That is, how can to guarantee that resources will not always be preempted from the same process? In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation any practical system must address. Clearly, the system must ensure that a process can be picked as a victim only a (small) finite number of times.

### 3.2.6 The Dining Philosophers

Reading Assignment

# Chapter 4

# Processor Scheduling

## 4.1 Basic Concepts

In a single-processor system, only one process can run at a time. Other processes must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The idea is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle with no useful work is accomplished during the waiting time. Multiprogramming however tries to use this waiting time productively. With several processes are kept in memory at one time, when one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. Every time one process has to wait, another process can take over use of the CPU. Scheduling of this kind is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.

### 4.1.1 Process Behaviour

Every process that executes in the computer consists of a cycle of CPU execution and I/O wait. The complete execution of a process alternate between these two state. Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution as shown in Figure 4.1.

Based on the duration of the CPU and I/O bursts, processes can be categorized into two as either:

- **CPU-bound Processes**: Typically have long CPU bursts and infrequent I/O bursts (waits).
- **I/O-bound Processes**: Typically have short CPU bursts and frequent I/O bursts.

As shown in Figure 4.2, the key factor to this categorization is the length of the CPU burst, not the length of the I/O burst.

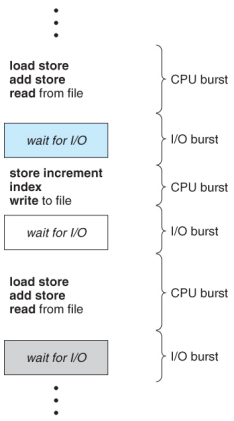



**Figure 4.1:** Alternating Sequence of CPU and I/O Bursts

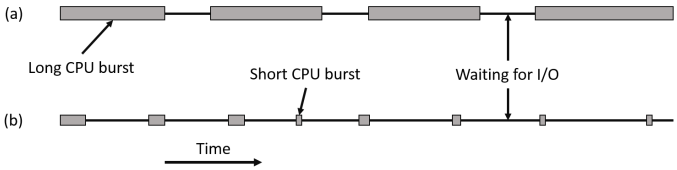



**Figure 4.2:** (a) CPU-bound Process. (b) I/O-bound Process

### 4.1.2 CPU Scheduler and Scheduling

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the CPU *scheduler*. The scheduler basically is the part of the operating system that makes the choice on which process in the ready queue to run first - using a scheduling algorithm. The scheduler will always schedule a process when any of the following four conditions occurs:

1. When a new process is created and a decision needs to be made whether to run the parent process or the child process
2. When a process exits.
3. When a process blocks on I/O or for some other reason.
4. When an I/O interrupt occurs

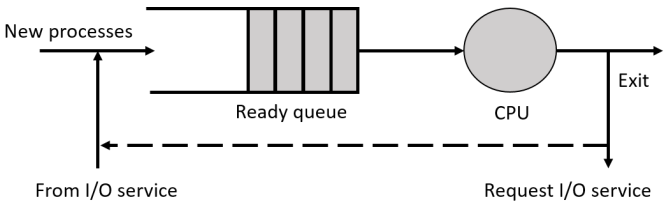



**Figure 4.3:** A simple CPU Scheduling Model

Figure 4.3 shows a simple CPU scheduling model and the necessary conditions for scheduling.

CPU scheduling basically is the process of making full use of a CPU by allowing one process to use the CPU while the execution of another process is on hold due to unavailability of any resource like I/O. The main aim of scheduling as mentioned previously is to make the system *efficient, fast and fair*. Generally there are two categories of CPU scheduling policies:

1. **Nonpreemptive scheduling**: In nonpreemptive scheduling, a process that is executing will continue until completion of its CPU burst. The process will then change to its wait state for I/O service, or terminate and exit the system.
2. **Preemptive scheduling**: In preemptive scheduling, the process that is executing may be interrupted before completion of its current CPU burst and moved back to the ready queue. A process may be interrupted due to another process with a higher priority arriving at the ready queue.

## 4.2 Scheduling Criteria

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favour one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms. Many criteria have been suggested for comparing CPU-scheduling algorithms, some of which include the following:

- **CPU utilization**: The proportion of time that the CPU spends executing processes. CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system).
- **Throughput**: The total number of processes that are executed and completed during some observation period.
- **Process average waiting time**: The average of the waiting intervals of all processes that are completed.
- **Average turnaround time**: The average of the intervals from arrival until completion, for all processes.
- **Average response time**: The average of the intervals from the time a process sends a command or request to the operating system until a response is received, for all processes.
- **Fairness**: A metric that indicates if all processes are treated in a similar manner.

For all systems, the general goals in scheduling are to ensure; there is **fairness** (giving each process a fair share of the CPU), **policy enforcement** (seeing that stated policy is carried out), and **balance** (keeping all parts of the system busy). The goals of scheduling in different systems however differ as follows:

- **Batch Systems**: *throughput* (maximizing jobs per hour), *turnaround time* (minimizing time between submission and termination of processes), and *CPU utilization* (keeping the CPU busy all the time).

- **Interactive Systems**: *response time* (responding quickly to requests) and *proportionality* (meeting users' expectations).
- **Real Time Systems**: *meeting deadlines* (avoiding losing of data) and *predictability* (avoiding quality degradation).

The goals of scheduling in different environments may differ

## 4.3 Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU-scheduling algorithms, some of which are discussed below.

### 4.3.1 First-Come, First-Served (FCFS) Scheduling

This is the most basic and simplest of all scheduling algorithms. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a *First-In, First-Out* (FIFO) queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand.

On the negative side, the average waiting time under the FCFS algorithm is generally not minimal and may vary substantially if the processes‘ CPU burst times vary greatly.

#### 4.3.1.1 Simple Analysis with FCFS

The following examples illustrate the concepts and techniques for the basic analysis of this scheduling policy and how to calculate some of the performance metrics. Let five processes arrive to the ready queue in the order $P_l$, $P_2$, $P_3$, $P_4$, and $P_5$ at time zero with their respective CPU burst as shown in Table 4.5.

**Table 4.1:** CPU Service Demand for Five Processes

| **Process** | $P_l$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|
| **CPU-burst (ms)** | 135 | 100 | 36 | 200 | 150 |

The Gantt chart (or execution diagram) in Figure 4.4 shows the starting times, execution intervals, and completion times for the five processes. This chart is used to build simple deterministic models that represent the scheduling of processes.

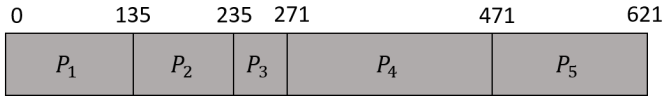



**Figure 4.4:** Gantt Chart of FCFS Scheduling for Five Processes

For this problem, process $P_1$ starts immediately, so its waiting time is zero. Process $P_2$ waits until $P_1$ completes, process $P_3$ waits until $P_2$ completes, process $P_4$ waits until $P_3$ completes, and process $P_5$ waits until $P_4$ completes.

Table 4.2 shows the starting times, completion times, and wait intervals for every process. From this data, the average turnaround time can be calculated using the completion times from the table. The average turnaround time for the five processes is 346.6 microseconds. In a similar manner, the average wait time can be calculated using the wait time for every process, and this value is 222.4 microseconds. The throughput is 5 because all five processes completed execution, and the CPU utilization is 100 percent because the CPU was never idle.

**Table 4.2:** Data for Five Processes Using FCFS Scheduling

| Process | Start Time | Completion Time | Wait Time | Turnaround Time | Normalized Turnaround Time |
|---|---|---|---|---|---|
| $P_l$ | 0 | 135 | 0 | 135 | 1.000 |
| $P_2$ | 135 | 235 | 135 | 235 | 2.350 |
| $P_3$ | 235 | 271 | 235 | 271 | 7.528 |
| $P_4$ | 271 | 471 | 271 | 471 | 2.355 |
| $P_5$ | 471 | 621 | 471 | 621 | 4.140 |

**Note**:

- Start Time: this is the time a process starts its execution when given the the CPU
- Completion Time: This is the time a process completes its entire CPU burst and exists the CPU.
- Wait Time: This is the time the process waits before it is given the CPU to execute.
- Turnaround Time: This is the time from process arrival to its finish time
- Normalized Turnaround Time: This is the ratio of the Turnaround Time of a process to its CPU burst. That is, Normalized Turnaround Time $= \frac{\text{Turnaround Time}}{\text{CPU-burst}}$
- CPU Utilisation $= \frac{\text{Final Completion Time} - \text{Total Idle Time}}{\text{Final Completion Time}} \times 100\%$

Consider the same five processes arrive to the ready queue in the order $P_l$, $P_2$, $P_3$, and $P_4$ at time zero while $P_5$ arrives at time 200 ms with their respective CPU-bursts as shown in Table 4.3

Table 4.4 shows the starting times, completion times, and wait intervals for every process in the second scenario.

The average turnaround time for the five processes in this case is 306.6 microseconds while the average wait time is 182.4 microseconds. As can be observed, the average turnaround and wait time have been significantly reduced because process $P_5$ arrives at 200ms.

**Table 4.3:** CPU Service Demand for Five Processes

| **Process** | $P_l$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|
| **CPU-burst (ms)** | 135 | 100 | 36 | 200 | 150 |

**Table 4.4:** Data for Five Processes Using FCFS Scheduling

| Process | Start Time | Completion Time | Wait Time | Turnaround Time | Normalized Turnaround Time |
|---|---|---|---|---|---|
| $P_l$ | 0 | 135 | 0 | 135 | 1.000 |
| $P_2$ | 135 | 235 | 135 | 235 | 2.350 |
| $P_3$ | 235 | 271 | 235 | 271 | 7.528 |
| $P_4$ | 271 | 471 | 271 | 471 | 2.355 |
| $P_5$ | 471 | 621 | 271 | 421 | 2.807 |

**Take Home Assignment**

- Let five processes arrive to the ready queue with their respective CPU-bursts and arrival times as shown in Tables 4.5, 4.7 and 4.6. Using the FCFS, compute the average: wait, turnaround, normalized turnaround and CPU utilisation for each scenario.

**Table 4.5:** CPU Service Demand for Five Processes: Scenario A

| **Process** | $P_l$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|
| **CPU-burst (ms)** | 135 | 100 | 36 | 200 | 150 |
| **Arrival Time (ms)** | 0 | 130 | 200 | 200 | 250 |

**Table 4.6:** CPU Service Demand for Five Processes: Scenario B

| **Process** | $P_l$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|
| **CPU-burst (ms)** | 135 | 100 | 36 | 200 | 150 |
| **Arrival Time (ms)** | 0 | 130 | 210 | 200 | 250 |

**Table 4.7:** CPU Service Demand for Five Processes: Scenario C

| **Process** | $P_l$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|
| **CPU-burst (ms)** | 135 | 100 | 36 | 200 | 150 |
| **Arrival Time (ms)** | 0 | 150 | 200 | 200 | 350 |

### 4.3.2 Shortest-Job-First (SJF) Scheduling

The SJF algorithm associates with each process the length of the process's next CPU burst. The operating system assigns a higher priority to the group of processes with the shortest CPU bursts. With the scheduler giving preference to the processes with the shortest CPU burst, when the CPU is available, it is assigned to the process that has the smallest CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. The SJF scheduling is provably optimal since it often results in the minimum wait time for processes.

On the negative side, the SJF scheduling is not fair compared to FCFS scheduling. For instance, if processes with shorter CPU-burst continue arriving at the ready queue, processes with longer CPU-burst may be left waiting indefinitely.

#### 4.3.2.1 Simple Analysis with SJF

The following examples illustrate the concepts and techniques for the basic analysis of this scheduling policy and how to calculate some of the performance metrics based on the SJF.

Let five processes arrive to the ready queue with their respective CPU-bursts and arrival times as shown in Table 4.8. Using the SJF scheduling, show in a tabular form the start, wait, completion, turnaround and normalised turnaround times for the five processes, and compute the CPU utilisation.

**Table 4.8:** CPU Service Demand for Processes

| **Process** | $P_l$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|
| **CPU-burst (ms)** | 10 | 15 | 5 | 11 | 9 |
| **Arrival Time (ms)** | 0 | 15 | 25 | 25 | 25 |

The Gantt chart in Figure 4.5 shows the starting times, execution intervals, and completion times for the five processes.

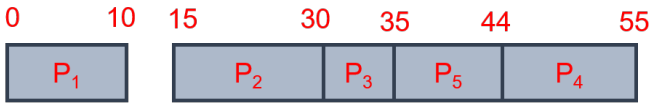



**Figure 4.5:** Gantt Chart of SJF Scheduling for Five Processes

For this problem, Process $P_1$ starts immediately it arrives since it is the only process in the ready queue. Process $P_1$ finishes executing its CPU burst at time 10ms and the CPU is idle till $P_2$ arrives at time 15ms which also starts executing without any wait time. At time 25ms before $P_2$ finishes its execution, $P_3$, $P_4$ and $P_5$ all arrive together. Using the SJF principle, $P_3$ is assigned the CPU after $P_2$ finishes since $P_3$ has the smallest CPU burst compared to $P_4$ and $P_5$. Subsequently $P_5$ is assigned the CPU next after $P_3$ finishes its execution. $P_4$ is finally assigned the CPU after $P_5$ finishes executing its CPU burst.

Table 4.9 shows the starting times, completion times, and wait times and normalized turnaround times for the five processes.

**Table 4.9:** Data for Five Processes Using SJF Scheduling

| Process | Start Time | Wait Time | Completion Time | Turnaround Time | Normalized Turnaround Time |
|---|---|---|---|---|---|
| $P_l$ | 0 | 0 | 10 | 10 | 1.00 |
| $P_2$ | 15 | 0 | 30 | 15 | 1.00 |
| $P_3$ | 30 | 5 | 35 | 10 | 2.00 |
| $P_5$ | 35 | 10 | 44 | 19 | 2.11 |
| $P_4$ | 44 | 19 | 55 | 30 | 2.73 |

From Table 4.9 and Figure 4.5, the CPU utilisation can be evaluated as: CPU utilisation = $\frac{55-5}{55} \times 100\% = 90.91\%$.

**Take Home Assignment**

- Let five processes arrive to the ready queue with their respective CPU-bursts and arrival times as shown in Tables 4.10, 4.11, 4.12 and 4.10. Using the SJF, compute the average: wait, turnaround, normalized turnaround and CPU utilisation for each scenario.

**Table 4.10:** CPU Service Demand for Processes: SJF A

| **Process** | $P_l$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|
| **CPU-burst (ms)** | 135 | 100 | 36 | 200 | 150 |
| **Arrival Time (ms)** | 0 | 0 | 0 | 0 | 0 |

**Table 4.11:** CPU Service Demand for Processes: SJF B

| **Process** | $P_l$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|
| **CPU-burst (ms)** | 100 | 100 | 36 | 200 | 15 |
| **Arrival Time (ms)** | 0 | 0 | 0 | 0 | 0 |

**Table 4.12:** CPU Service Demand for Processes: SJF C

| **Process** | $P_l$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|
| **CPU-burst (ms)** | 135 | 100 | 36 | 200 | 10 |
| **Arrival Time (ms)** | 0 | 50 | 100 | 200 | 250 |

**Table 4.13:** CPU Service Demand for Processes: SJF D

| **Process** | $P_l$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|
| **CPU-burst (ms)** | 135 | 100 | 60 | 200 | 10 |
| **Arrival Time (ms)** | 0 | 150 | 140 | 200 | 200 |

### 4.3.3 Shortest Remaining Time Next (SRTN) Scheduling

The SRTN scheduling algorithm can be seen as a preemptive version of the SJF. The operating system assigns higher priority to processes with the smallest remaining run time. The scheduler always chooses the process whose remaining run time is shortest and assigns the CPU to it. In the case the CPU is already assigned to a process which is still in execution and a process with a shorter remaining run time arrives at the ready queue, the CPU will be preempted from the already executing process and given to the one with the shortest remaining run time. This means the run time of each process has to be known in advance. When a new process arrives, its total time is compared to the current process's remaining time. If the new process needs less time to finish than the current process, the current process is suspended and the new process started.

#### 4.3.3.1 Simple Analysis with SRTN

The following examples illustrate the concepts and techniques for the basic analysis of this scheduling policy and how to calculate some of the performance metrics based on the SRTN.

Let five processes arrive to the ready queue with their respective CPU-bursts and arrival times as shown in Table 4.14. Using the SRTN scheduling, show in a tabular form the start, wait, completion, turnaround and normalised turnaround times for the five processes, and compute the CPU utilisation.

**Table 4.14:** CPU Service Demand for Processes

| **Process** | $P_l$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|
| **CPU-burst (ms)** | 10 | 2 | 3 | 9 | 2 |
| **Arrival Time (ms)** | 0 | 5 | 6 | 7 | 7 |

The Gantt chart in Figure 4.6 shows the starting times, execution intervals, and completion times for the five processes.

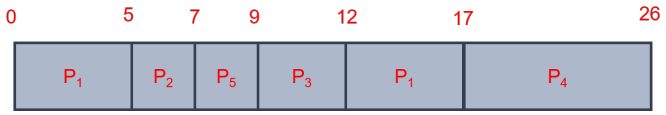



**Figure 4.6:** Gantt Chart of SRTN Scheduling for Five Processes

For this problem, process $P_l$ is assigned the CPU immediately it arrives. At time 5ms, $P_2$ arrives at the ready queue. Using the shortest remaining time next principle, the CPU is preempted from $P_l$ and assigned to $P_2$. $P_2$ begins execution while $P_l$ is placed back in the ready queue with 5ms of CPU burst remaining. At time 6ms, $P_2$ arrives at the ready queue with a remaining CPU burst of 3ms. Since $P_2$ has 1ms left to execute at time 6ms, it continues its execution till it finishes at 7ms - at which point there $P_4$ and $P_5$ arrive at the ready queue. At time 7ms, the CPU is assigned to $P_5$ since it has the shortest remaining time among the processes in the ready queue. When P5 finishes executing, the CPU is assigned to $P_3$ and subsequently to $P_l$ before $P_4$, all based on the SRTN principle.

Table 4.15 shows the starting times, completion times, and wait times and normalized turnaround times for the five processes.

**Table 4.15:** Data for Five Processes Using SJF Scheduling

| Process | Start Time | Wait Time | Completion Time | Turnaround Time | Normalized Turnaround Time |
|---|---|---|---|---|---|
| $P_l$ | 0 | 7 | 17 | 17 | 1.70 |
| $P_2$ | 5 | 0 | 7 | 2 | 1.00 |
| $P_5$ | 7 | 0 | 9 | 2 | 1.00 |
| $P_3$ | 9 | 3 | 12 | 6 | 2.00 |
| $P_4$ | 17 | 10 | 26 | 19 | 2.11 |

From Table 4.15 and Figure 4.6, the CPU utilisation can be evaluated as: CPU utilisation $= \frac{26-0}{26} \times 100\% = 100\%$.

**Take Home Assignment**

- Let five processes arrive to the ready queue with their respective CPU-bursts and arrival times as shown in Table 4.16. Using the SJF, compute the average: wait, turnaround, normalized turnaround and CPU utilisation.

**Table 4.16:** CPU Service Demand for Five Processes

| **Process** | $P_l$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|
| **CPU-burst (ms)** | 160 | 100 | 36 | 25 | 15 |
| **Arrival Time (ms)** | 0 | 50 | 100 | 200 | 250 |

### 4.3.4 Round-Robin (RR) Scheduling

The round-robin (RR) scheduling algorithm is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a time **quantum**, is defined. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

To implement RR scheduling, the ready queue is treated as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process. One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The main advantage that this scheduling approach provides to users is interactive computing. It is worth noting that the performance using this approach will be affected significantly.

#### 4.3.4.1 Simple Analysis with RR

The following examples illustrate the concepts and techniques for the basic analysis of this scheduling policy and how to calculate some of the performance metrics based on the RR.

Let five processes arrive to the ready queue with their respective CPU-bursts and arrival times as shown in Table 4.17. Using the RR scheduling with a quantum of 40ms, show in a tabular form the start, wait, completion, turnaround and normalised turnaround times for the five processes, and compute the CPU utilisation.

**Table 4.17:** CPU Service Demand for Five Processes

| **Process** | $P_l$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|
| **CPU-burst (ms)** | 135 | 100 | 36 | 200 | 150 |

The Gantt chart in Figure 4.7 shows the starting times, execution intervals, and completion times for the five processes.

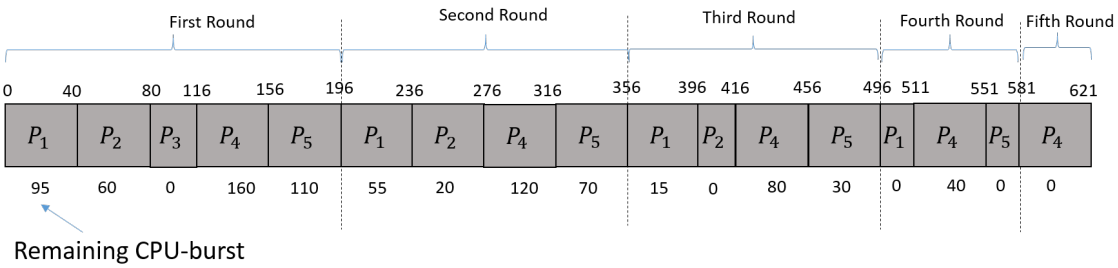



**Figure 4.7:** Gantt Chart of RR Scheduling for Five Processes

For this problem, as shown in Figure 4.7, process P1 is assigned the CPU first for a time quantum of 40ms. Based on the RR principle, the CPU is preempted and assigned to P2 while P1 is placed back in the ready queue. P2 executes also for the same time quantum before the CPU is preempted, assigned to P3 and P2 placed in the ready queue. P3 however executes for 36ms and releases the CPU to P4. P4 executes for the allotted quantum time before the CPU is preempted from it and assigned to P5. This process continues till all the processes have executed their CPU bursts. Table 4.18 shows the starting times, completion times, and wait times and normalized turnaround times for the five processes.

**Table 4.18:** Data for Five Processes Using RR Scheduling

| Process | Start Time | Completion Time | Wait Time | Turnaround Time | Normalized Turnaround Time |
|---|---|---|---|---|---|
| $P_l$ | 0 | 511 | 376 | 511 | 3.785 |
| $P_2$ | 40 | 416 | 316 | 416 | 4.160 |
| $P_3$ | 80 | 116 | 80 | 116 | 3.222 |
| $P_4$ | 116 | 621 | 461 | 621 | 3.105 |
| $P_5$ | 156 | 518 | 431 | 518 | 3.453 |

From Table 4.18 and Figure 4.7, the CPU utilisation can be evaluated as: CPU utilisation $= \frac{621-0}{621} \times 100\% = 100\%$.

### Take Home Assignment

- Let five processes arrive to the ready queue with their respective CPU-bursts and arrival times as shown in Table 4.19. Using the RR scheduling with a quantum of 30ms, show in a tabular form the start, wait, completion, turnaround and normalised turnaround times for the five processes, and compute the CPU utilisation.

**Table 4.19:** CPU Service Demand for Five Processes

| **Process** | $P_l$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|
| **CPU-burst (ms)** | 160 | 100 | 36 | 25 | 15 |
| **Arrival Time (ms)** | 0 | 20 | 100 | 200 | 250 |

### 4.3.5 Priority Scheduling

With this scheduling apprach, a priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. Processes with larger CPU burst are often assigned lower priority, and vice versa.

Priority scheduling can be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling algorithms is indefinite blocking, or starvation. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

#### Take Home Assignment

- Let five processes arrive to the ready queue in the order $P_l$, $P_2$, $P_3$, $P_4$ and $P_5$ at time zero with their respective CPU-bursts and priorities as shown in Table 4.20 below. Using the nonpreemptive priority scheduling technique, show in a tabular form the start, wait, completion, turnaround and normalised turnaround times for the five processes, and compute the CPU utilisation.

**Table 4.20:** CPU Service Demand for Five Processes

| **Process** | $P_l$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|
| **CPU-burst (ms)** | 135 | 180 | 36 | 20 | 150 |
| **Priority** | 2 | 3 | 1 | 1 | 2 |

- Let five processes arrive to the ready queue with their respective CPU-bursts, arrival times and priorities as shown in Table 4.21.

**Table 4.21:** CPU Service Demand for Five Processes

| **Process** | $P_l$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|
| **CPU-burst (ms)** | 105 | 150 | 56 | 40 | 90 |
| **Arrival Time (ms)** | 0 | 20 | 70 | 150 | 200 |
| **Priority** | 2 | 3 | 1 | 1 | 2 |

### 4.3.6 Multilevel Queue Scheduling

The multilevel queue scheduling algorithm was created for situations in which processes are easily classified into different groups. For example, a common division is made between foreground (interactive) processes and background (batch) processes. These two types of processes have different response-time

requirements and so may have different scheduling needs. In addition, foreground processes may have priority (externally defined) over background processes.

A multilevel queue scheduling algorithm partitions the ready queue into several separate queues (see Figure 4.8). The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.

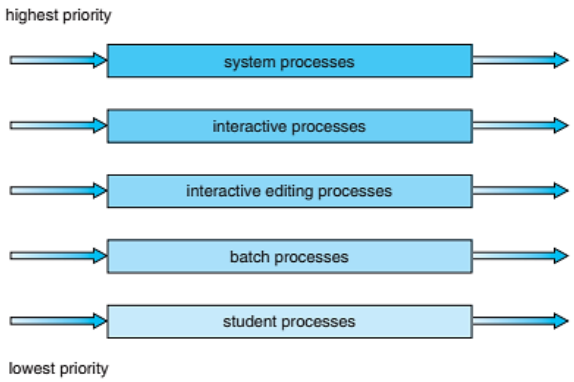



**Figure 4.8:** Multilevel Queue Scheduling

### 4.3.7 Multilevel Feedback Queue Scheduling

Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but it is inflexible.

The multilevel feedback queue scheduling algorithm, in contrast, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This prevents starvation.

# Chapter 5

# Memory Management

Main memory (RAM) is an important resource that must be very carefully managed. In the ideal world, the ideal memory is expected to be infinitely large, infinitely fast, non-volatile and very affordable. Though achieving these properties on a single unit is not possible currently, the closest solution getting the ideal memory properties is through memory hierarchy. With the concept of the memory hierarchy, computers have a few megabytes of very fast, expensive, volatile cache memory, a few gigabytes of medium-speed, medium-priced, volatile main memory, and a few terabytes of slow, cheap, nonvolatile magnetic or solid-state disk storage. It is the job of the operating system to abstract this hierarchy into a useful model and then manage the abstraction.

The part of the operating system that manages (part of) the memory hierarchy is called the memory manager. Its job is to efficiently manage memory: keep track of which parts of memory are in use, allocate memory to processes when they need it, and deallocate it when they are done.

This chapter will discuss several different memory management models, ranging from very simple to highly sophisticated. Since managing the lowest level of cache memory is normally done by the hardware, the focus of this chapter will be on the programmer‘s model of main memory and how it can be managed.

## 5.1 No Memory Abstraction

The simplest memory abstraction is to have no abstraction at all. Early mainframe computers (before 1960), early minicomputers (before 1970), and early personal computers (before 1980) had no memory abstraction. Every program simply saw the physical memory. Thus, the model of memory presented to the programmer was simply physical memory, a set of addresses from 0 to some maximum, each address corresponding to a cell containing some number of bits, commonly eight. Under these conditions, it was not possible to have two running programs in memory at the same time. If the first program wrote a new value to, say, location 2000, this would erase whatever value the second program was storing there. Nothing would work and both programs would crash almost immediately.

With the no memory abstraction, several options are possible. Three variations are shown in Figure 5.1. The operating system may be at the bottom of memory in RAM (Random Access Memory),

as shown in Figure 5.1(a), or it may be in ROM (Read-Only Memory) at the top of memory, as shown in Figure 5.1(b), or the device drivers may be at the top of memory in a ROM and the rest of the system in RAM down below, as shown in Figure 5.1(c). Models (a) and (c) have the disadvantage that a bug in the user program can wipe out the operating system, possibly with disastrous results.

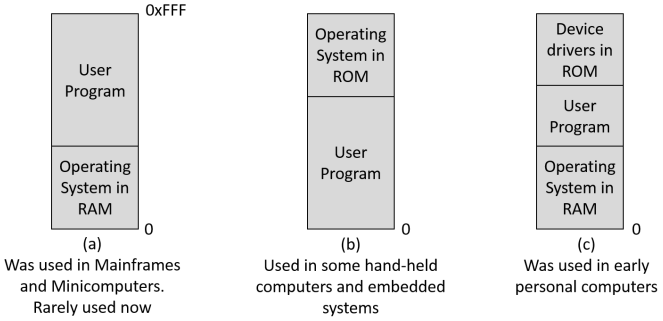



**Figure 5.1:** Some Variations of Arranging the OS in No Memory Abstraction

### 5.1.1 Running Multiple Programs Without a Memory Abstraction

When the system is organized in this way, generally only one process at a time can be running. As soon as the user types a command, the operating system copies the requested program from disk to memory and executes it. When the process finishes, the operating system displays a prompt character and waits for a user new command. When the operating system receives the command, it loads a new program into memory, overwriting the first one.

One way to get some parallelism in a system with no memory abstraction is to program with multiple threads. Since all threads in a process are supposed to see the same memory image, the fact that they are forced to is not a problem. While this idea works, it is of limited use since what people often want is unrelated programs to be running at the same time, something the threads abstraction does not provide.

To actually runt multiple programs with the no memory abstraction, the operating system has to save the entire contents of memory to a disk file, then bring in and run the next program. As long as there is only one program at a time in memory, there are no conflicts. This concept (swapping) will be discussed in Section 5.2.2. Additionally, with the addition of some special hardware, it is possible to run multiple programs concurrently, even without swapping.

## 5.2 Memory Abstraction: Address Spaces

Exposing physical memory to processes as mentioned previously has several major drawbacks. First, if user programs can address every byte of memory, they can easily trash the operating system, intentionally or by accident, bringing the system to a grinding halt. This problem exists even if only one user program (application) is running. Second, with this model, it is difficult to have multiple programs running at once (taking turns, if there is only one CPU). On personal computers, it is common to have several programs open at once (a word processor, an email program, a Web browser),

one of them having the current focus, but the others being reactivated at the click of a mouse. Since this situation is difficult to achieve when there is no abstraction from physical memory, something had to be done.

### 5.2.1 The Notion of An Address Space

To allow multiple applications to be in the memory at the same time without interfering with each other, two problems have to be solved: protection and relocation of memory.

One way to solving these problems is to invent a new abstraction for memory: the address space. The address space creates a kind of abstract memory for programs to live in. An *address space* is the set of addresses that a process can use to address memory. Each process has its own address space, independent of those belonging to other processes (except in some special circumstances where processes want to share their address spaces).

Address spaces do not have to be numeric. The set of *.com* Internet domains is also an address space. Somewhat harder is how to give each program its own address space, so address 28 in one program means a different physical location than address 28 in another program.

If the physical memory of the computer is large enough to hold all the processes, the schemes described so far will more or less do. But in practice, the total amount of RAM needed by all the processes is often much more than can fit in memory. Serious user application programs nowadays, like Photoshop, can easily require 500 MB just to boot and many gigabytes once they start processing data. Consequently, keeping all processes in memory all the time requires a huge amount of memory and cannot be done if there is insufficient memory. To address this issue, two general approaches (swapping and virtual memory) were developed over the years to deal with the memory overload.

### 5.2.2 Swapping

The is the simplest strategy to deal with memory overload. *Swapping* basically consists of bringing in each process in its entirety, running it for a while, then putting it back on the disk. Idle processes are mostly stored on disk, so they do not take up any memory when they are not running (although some of them wake up periodically to do their work, then go to sleep again).

The operation of a swapping system is illustrated in Figure 5.2. Initially, only process A is in memory. Then processes B and C are created or swapped in from disk. In Figure 5.2(d) A is swapped out to disk. Then D comes in and B goes out. Finally A comes in again. Since A is now at a different location, addresses contained in it must be relocated, either by software when it is swapped in or (more likely) by hardware during program execution.

When swapping creates multiple holes in memory, it is possible to combine them all into one big one by moving all the processes downward as far as possible. This technique is known as *memory compaction*. It is usually not done because it requires a lot of CPU time. For example, on a 16-GB machine that can copy 8 bytes in 8 nsec, it would take about 16 sec to compact all of memory.

A point that is worth making concerns how much memory should be allocated for a process when

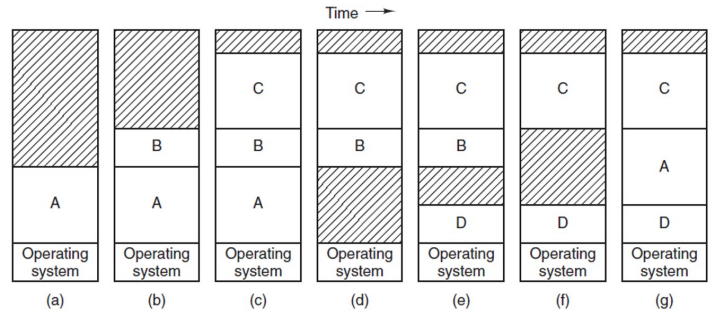



**Figure 5.2:** Typical Swapping Operation

it is created or swapped in:

1. If processes are created with a fixed size that never changes, then the allocation is simple:
   - The operating system allocates exactly what is needed, no more and no less.
2. If, however, processes' data segments can grow, for example, by dynamically allocating memory from a heap.
   - If a hole is adjacent to the process, it can be allocated and the process allowed to grow into the hole. On the other hand, if the process is adjacent to another process, the growing process will either have to be moved to a hole in memory large enough for it, or one or more processes will have to be swapped out to create a large enough hole.
   - If a process cannot grow in memory and the swap area on the disk is full, the process will have to suspended until some space is freed up (or it can be killed).
3. If it is expected that most processes will grow as they run.
   - Allocate a little extra memory whenever a process is swapped in or moved.

Figure 5.3 illustrate some typical memory allocation for growing processes during swapping.

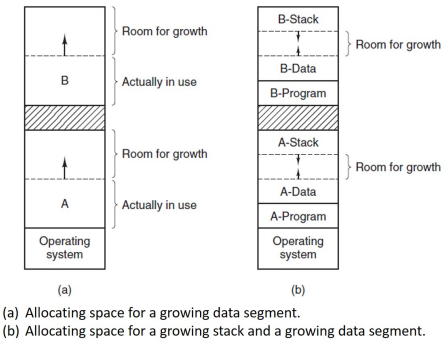



**Figure 5.3:** Swapping: Typical Meomry Allocation for Growing Processes

### 5.2.3 Managing Free Memory

When memory is assigned dynamically, the operating system must manage it. In general terms, there are two ways to keep track of memory usage: *bitmaps* and *linked lists* which are discussed below.

#### 5.2.3.1 Memory Management with Bitmaps

With a bitmap, memory is divided into allocation units as small as a few words and as large as several kilobytes. Corresponding to each allocation unit is a bit in the bitmap, which is 0 if the unit is free and 1 if it is occupied (or vice versa). Figure 5.4 shows part of memory and the corresponding bitmap.

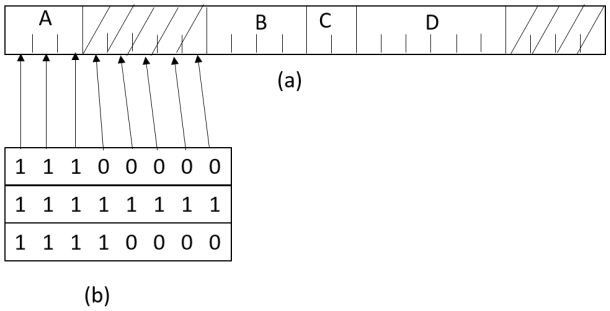



**Figure 5.4:** Memory Management with Bitmaps: (a) Four Processes with two Holes (b) Corresponding Bitmap

A bitmap provides a simple way to keep track of memory words in a fixed amount of memory because the size of the bitmap depends only on the size of memory and the size of the allocation unit. The main problem is that when it has been decided to bring a $k$-unit process into memory, the memory manager must search the bitmap to find a run of $k$ consecutive 0 bits in the map. Searching a bitmap for a run of a given length is a slow operation (because the run may straddle word boundaries in the map); this is an argument against bitmaps.

#### 5.2.3.2 Memory Management with Linked Lists

Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment either contains a process or is an empty hole between two processes. The memory of Figure 5.4 is represented in Figure 5.5 as a linked list of segments. Each entry in the list specifies a hole (H) or process (P), the address at which it starts, the length, and a pointer to the next item.

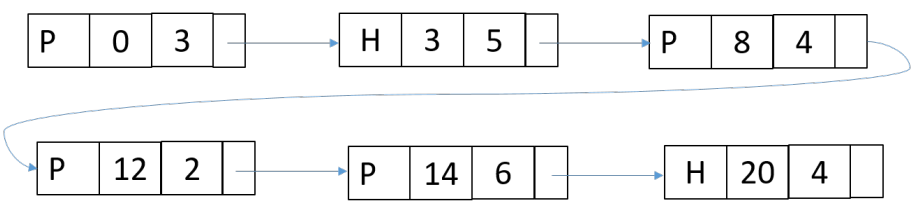



**Figure 5.5:** The Four Processes with two Holes in Figure 5.4 in a Linked list

With the linked list approach, the segment list is kept sorted by address. Sorting this way has the advantage that when a process terminates or is swapped out, updating the list is straightforward. When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a created process (or an existing process being swapped in from disk). Typical of such algorithms include:

1. First Fit: With the first fit algorithm, the memory manager scans along the list of segments until it finds a hole that is big enough. The hole is then broken up into two pieces, one for the process and one for the unused memory, except in the statistically unlikely case of an exact fit. First fit is a fast algorithm because it searches as little as possible.

2. Next Fit: This is a minor variation of first fit. It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole. The next time it is called to find a hole, it starts searching the list from the place where it left off last time, instead of always at the beginning, as first fit does.

3. Best Fit: The best fit algorithm searches the entire list, from beginning to end, and takes the smallest hole that is adequate. Rather than breaking up a big hole that might be needed later, best fit tries to find a hole that is close to the actual size needed, to best match the request and the available holes. Best fit is slower than first fit because it must search the entire list every time it is called.

4. Worst Fit: Unlike the best fit, this searches for and always take the largest available hole, so that the new hole will be big enough to be useful.

5. Quick Fit: Reading assignment.

## 5.3 Virtual Memory

Though recently computers are equipped with memory sizes that are increasing rapidly, software sizes are increasing much faster. puts even more demands on memory. As a consequence of these developments, there is a need to run programs that are too large to fit in memory, and there is certainly a need to have systems that can support multiple programs running simultaneously, each of which fits in memory but all of which collectively exceed memory. Though swapping can be employed to solve this issue, it is not an attractive option, since a typical SATA disk has a peak transfer rate of several hundreds of MB/sec, which means it takes seconds to swap out a 1-GB program and the same to swap in a 1-GB program.

A solution adopted in the 1960s was to split programs into little pieces, called *overlays*. When a program started, all that was loaded into memory was the overlay manager, which immediately loaded and ran overlay 0. When it was done, it would tell the overlay manager to load overlay 1, either above overlay 0 in memory (if there was space for it) or on top of overlay 0 (if there was no space). This method has come to be known as *virtual memory*.

**Virtual memory** is a space where large programs can store themselves in form of pages (portions) while their execution and only the required portions of processes are loaded into the main memory. This space is a section of the hard disk that is set up to emulate the computer's RAM.

The basic idea behind virtual memory is that each program has its own address space, which is broken up into chunks called pages. Each page is a contiguous range of addresses. These pages are mapped onto physical memory, but not all pages have to be in physical memory at the same time to run the program. When the program references a part of its address space that is in physical memory, the hardware performs the necessary mapping on the fly. When the program references a part of its address space that is not in physical memory, the operating system is alerted to go get the missing piece and re-execute the instruction that failed.

The reason behind virtual memory and paging is because there are several situations when the entire program does not need to be loaded into memory. Some typical situations when entire program is not required to be loaded fully in main memory include:

- Handling error codes: error handling routines are used only when an error occurred in the data or computation.
- When certain features and options of some programs are rarely used.
- When the ability to execute a program that is only partially in memory would counter many benefits.

There are many benefits with the use of virtual memory, typical of such include:

- More programs could be run at the same time: increasing CPU utilization and throughput with no increase in turnaround time.
- Large programs can be written: the virtual space available is huge compared to physical memory.
- Programs can be larger than physical memory as they would no longer be constrained to the amount of physical memory
- Availability of more physical memory: programs are stored on virtual memory, as such they occupy less space on actual physical memory.
- Faster and easy swapping of processes: as less I/O will be required to load or swap user programs into memory.

### 5.3.1 Paging

Despite the many benefits of virtual memory, improper use or management will not result in these benefits. Most virtual memory systems use a technique called *paging* in virtual memory management. *Paging* is basically a memory management scheme for controlling how a computer's virtual memory resources are accessed and/or shared. It is an important part of virtual memory implementations in modern operating systems with two main policies:

1. Fetch Policies: For resolving how pages are loaded into the memory in which two policies involved: *Demand Paging* and *Prepaging*
2. Replacement Policies: For resolving how pages are replaced in the memory

#### 5.3.1.1 Prepaging

Prepaging is a virtual memory management policy in which currently used pages (portions) of the process and other pages (not demanded) of the process are loaded into memory. Though prepaging saves time when large contiguous structures are used and reduces a large number of page faults that occurs at process startup, it wastes memory and time when the additional pages loaded are not demanded immediately.

#### 5.3.1.2 Demand Paging

In demand paging, portions of the processes currently being referenced are loaded in the main memory while portions (pages) of the processes that are not being referenced are loaded into the secondary memory (virtual memory). If an executing program references a page which is not available in the main memory, the processor treats this invalid memory reference as a *page fault*. When a page fault is detected, the program transfers control to the operating system to demand the page in the memory. The operating system then fetches (loads) the demanded page from the virtual memory into the main memory. After a new page is loaded, the process continues its execution or may start execution from the beginning.

The following are the advantages of demand paging:

- More efficient use of memory.
- Allows more number of processes into the memory at the same time
- Large virtual memory.
- There is no limit on degree of multiprogramming.

One main disadvantage of demand paging is that, after a new page is loaded, the process may start execution from the beginning.

## 5.4 Page Replacement Algorithms

These are the techniques which the operating system uses to decide which memory pages to swap out or write to disk when a page of memory needs to be allocated. There are two groups of paging algorithms used in virtual memory management:

- Based on static frame[1] allocation and static paging algorithms
- Based on dynamic frame allocation and dynamic paging algorithms

The main difference between the two groups of algorithms is on how memory frames are allocated to the processes.

[1] A small fixed-sized block of (physical) memory

#### 5.4.0.1 Basic Page Replacement

When a page fault occurs, the operating system has to choose a page to evict (remove from memory) to make room for the incoming page. If the page to be removed has been modified while in memory, it must be rewritten to the disk to bring the disk copy up to date. If, however, the page has not been changed (e.g., it contains program text), the disk copy is already up to date, so no rewrite is needed. The page to be read in just overwrites the page being evicted.

The basic page replacement when a page fault occurs is as follows:

1. Find the location of the page requested by ongoing process on the disk.
2. Find a free frame. If there is a free frame, use it.
3. If there is no free frame, use a page-replacement algorithm to select any existing frame to be replaced - victim frame.
4. Write the victim frame to disk and change all related page tables to indicate that this page is no longer in memory.
5. Move the required page and store it in the frame and adjust all related page and frame tables to indicate the change.
6. Restart the process that was waiting for this page.

Some of the common page replacement algorithms are discussed below.

### 5.4.1 First-In, First-Out Replacement (FIFO) Algorithm

The simplest for page replacement algorithm. When a page fault occurs and there are no empty frames for the process, the page selected to be replaced is the one that has been in memory the longest time, that is, the oldest page. In practice, FIFO has the worst performance compared to the other algorithms. The only advantage of this algorithm is its simplicity and ease of implementation.

### 5.4.2 The Optimal Page Replacement Algorithm

The best possible page replacement algorithm is easy to describe but impossible to actually implement. This algorithm requires knowledge of the entire page reference stream in advance. When a page fault occurs and there are no empty frames for the process, the algorithm looks ahead in the page reference stream to find out about future references to the pages currently in physical memory. Based on this "future" knowledge, the optimal page to be removed from memory is chosen. The approach used is to replace the page in memory that will not be used for the longest period. Though this algorithm is not normally realizable in practice, it serves as a reference for comparison with the other algorithms.

### 5.4.3 Least Recently Used (LRU) Replacement

This algorithm replaces the page that has not been used for the longest period. It takes past knowledge from the page reference stream, instead of future knowledge as used in the optimal algorithm. The assumption taken in the algorithm is that recent page references give a good estimation of page references in the near future. When a page fault occurs and there are no empty frames, the algorithm selects the least recently referenced page for replacement.

### 5.4.4 The Second-Chance Page Replacement Algorithm

Reading assignment

## 5.5 Segmentation

Reading Assignment

## 5.6 Things Worth Noting

### 5.6.1 Thrashing

Thrashing is a condition or state in the system in which all the time a process spends is dedicated for swapping pages, thus no computation is carried out by the process. If the number of frames allocated to a process is too low, the execution of the process will generate an excessive number of page faults, and eventually the process will make no progress in its execution as it will spend practically all of its time paging. One approach used to prevent thrashing is to determine or estimate, for every active process, the sizes of the localities of the process in order to decide on the appropriate number of frames to allocate to that process.

# Chapter 6

# File Systems

## 6.1 Introduction

The file system is the most visible aspect of an operating system to most users. It provides the mechanism for on-line storage of and access to both data and programs of the operating system and all the users of the computer system. The file system consists of two distinct parts: a collection of files, each storing related data, and a directory structure, which organizes and provides information about all the files in the system.

## 6.2 File Concept

Generally, a file is a named collection of related information that is recorded on secondary storage. More specifically, a *file* is simply a sequence of bytes that have been stored in some device on the computer. Those bytes will contain whatever data we would like to store in the file - for example, a text file containing only the characters that we are interested in, a word processing document file that also contains data about how to format the text, and a database file that contains data organized in multiple tables.

A file has a certain defined structure, which depends on its type. For example, while **text file** is a sequence of characters organized into lines (and possibly pages), **source file** is a sequence of functions, each of which is further organized as declarations followed by executable statements, and **an executable file** is a series of code sections that the loader can bring into memory and execute.

### 6.2.1 File Attributes

A file's attributes vary from one operating system to another but typically consist of these:

- **Name:** The symbolic file name is the only information kept in human readable form.
- **Identifier:** This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- **Type:** This information is needed for systems that support different types of files.
- **Location:** This information is a pointer to a device and to the location of the file on that device.

- **Size:** The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- **Protection:** Access-control information determines who can do reading, writing, executing, and so on.
- **Time, date, and user identification:** This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

### 6.2.2 File Operations

A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files. This section discusses what the operating system must do to perform each of these six basic file operations.

- **Creating a file:** Two steps are necessary to create a file. First, space in the file system must be found for the file. Second, an entry for the new file must be made in the directory.
- **Writing a file:** To write a file, the operating system makes a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location. The system must keep a write pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.
- **Reading a file:** To read from a file, the operating system uses a system call that specifies the name of the file and where (in memory) the next block of the file should be put. Again, the directory is searched for the associated entry, and the system needs to keep a read pointer to the location in the file where the next read is to take place. Once the read has taken place, the read pointer is updated.
- **Repositioning within a file:** The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as a file seek.
- **Deleting a file:** To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase the directory entry.
- **Truncating a file:** The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged — except for file length — but lets the file be reset to length zero and its file space released.

### 6.2.3 File Types

If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways. A common technique for implementing file types is to include the type as part of the file name. The

name is split into two parts — a name and an extension, usually separated by a period (see Figure 6.1). In this way, the user and the operating system can tell from the name alone what the type of a file is. Most operating systems allow users to specify a file name as a sequence of characters followed by a period and terminated by an extension made up of additional characters. Examples include assignment.docx, server.c, and ReaderThread.cpp.

| file type | usual extension | function |
|---|---|---|
| executable | exe, com, bin or none | ready-to-run machine-language program |
| object | obj, o | compiled, machine language, not linked |
| source code | c, cc, java, perl, asm | source code in various languages |
| batch | bat, sh | commands to the command interpreter |
| markup | xml, html, tex | textual data, documents |
| word processor | xml, rtf, docx | various word-processor formats |
| library | lib, a, so, dll | libraries of routines for programmers |
| print or view | gif, pdf, jpg | ASCII or binary file in a format for printing or viewing |
| archive | rar, zip, tar | related files grouped into one file, sometimes compressed, for archiving or storage |
| multimedia | mpeg, mov, mp3, mp4, avi | binary file containing audio or A/V information |

**Figure 6.1:** Some Common File Types

## 6.3 Access Methods

An access method describes the manner and mechanisms by which a process accesses the data in a file. There are two common access methods:

- Sequential: The simplest access method where information in the file is processed in order, one record after the other. The file is read or written sequentially, starting at the beginning of the file and moving to the end. This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion.

- Random (or direct): In the random access method, the application may choose to access any part of the file at any time. There are no restrictions on the order of reading or writing for a direct-access file. Direct-access files are of great use for immediate access to large amounts of information.

## 6.4 Directory Functions

There are a number of operations or functions that can typically be specified with folders (directories):

- Create file: When a process opens a file that does not already exist, it must be created. In this situation, the open system call will use the create file function to allocate space for the file and create a directory entry for it.
- Delete file: Deletes space allocated to the file and its directory entry.
- Rename file: Changes the name recorded in the directory entry for a file.
- File exist?: Searches a folder to see if there is a directory entry whose name matches that of the desired file.
- Directory list: Returns a list of the directory entries in a folder.
- Get and set attributes: Returns various attributes from a directory entry or changes those attributes.

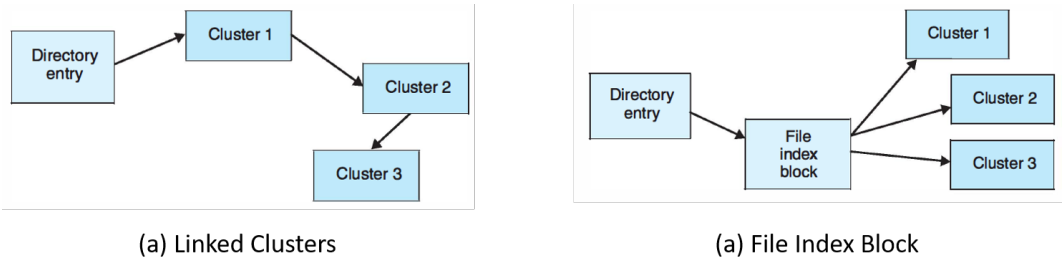
## 6.5 File Space Allocation

One of the major tasks for the file management subsystem is to allocate space for files. A simple mechanism for doing this is *contiguous allocation*. In this scheme, a contiguous section of the disk is allocated for a file. This scheme actually achieves optimal performance for the application that is reading/writing the file. Unfortunately, the contiguous allocation scheme has two major deficiencies:

- It is necessary to know how much space will be required for the file prior to creating the file. This usually is not possible.
- Because files are continually being created and deleted, after a system has run for some time, there may not be the necessary space in a contiguous chunk. The space will be typically available in smaller chunks scattered around the disk.

### 6.5.1 Cluster Allocation

To resolve these two problems in the *contiguous allocation*, systems will commonly use *cluster allocation* - a scheme in which chunks of space are allocated to a file as a process that is writing the file needs them. A system that is using cluster allocation needs to remember where all of the clusters for a file are located. There are two approaches for carrying this out. The first uses *linked clusters*, a system in which clusters are linked together in a chain in such a manner that each cluster points to the next. The second approach uses indexed clusters, a system in which the directory entry points to a file index block, which has an array of pointers to clusters. These two approaches are shown in Figure 6.2




(a) Linked Clusters (a) File Index Block

**Figure 6.2:** Types of Cluster Allocation

## 6.6 Real-World Systems

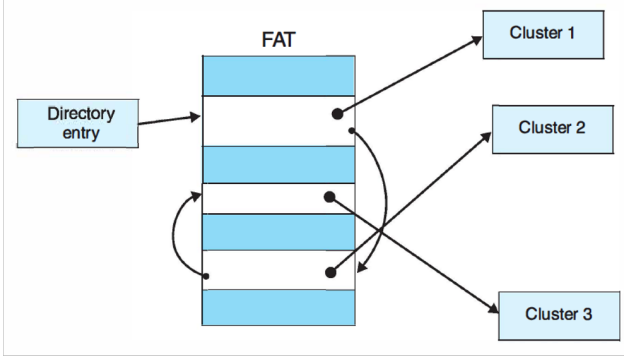
A disk drive in a real-world system is typically organized in a specific manner. There is a *boot block*, which contains the code to do the initial boot of the system. There is typically a *partition control block*, which describes the partitions that follow. Then, there will be one or more partitions. One of these partitions is designated the active partition, and it will contain the operating system that is to be booted. It is also possible to have each partition controlled by a different file system, which will thus use different strategies and provide different features. We now examine several real-world file systems:

- **Microsoft FAT System**: The Microsoft Fat System was originally created for the MS-DOS operating system. It had a limitation of a 2GB maximum disk size. While this seemed gigantic when it was invented in the early 1980s, disk drive manufacturers surpassed this size in the mid-1990s. Thus, Microsoft created a new version, FAT-32, for Windows 98. Windows NT, 2000, XP, Vista, and 2003 Server also support FAT-32.

  The FAT file system has a large table on disk called the file allocation table (FAT). The directory entry for a file points to an entry in the FAT. That entry points to the first cluster. If there is a second cluster, the first FAT entry points to a second FAT entry, which points to the second cluster, and so on. Thus, the FAT file system uses a linked clusters approach where the links go through the FAT entries rather than the actual clusters. Figure 6.3 shows the characteristics and structure of a FAT-32 system.

| | |
|---|---|
| Cluster size | 1K to 32K |
| Maximum partition size | 2 terabytes |
| Maximum file size | 4 gigabytes |
| Filenames | 8 bit ASCII characters |
| File permissions supported? | No |
| Transaction log? | No |

(a) Microsoft FAT File System Characteristics



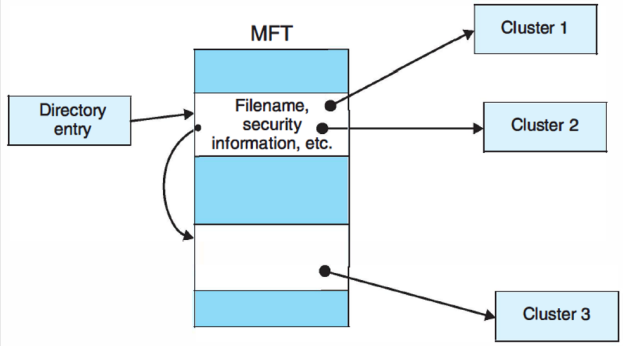

(a) Microsoft FAT File System Structure

**Figure 6.3:** Microsoft FAT System

- **Microsoft NTFS System**: The Microsoft New Technology File System (NTFS) was created as part of the Windows NT development effort to create a reliable system that could be used as a server by businesses. NTFS has a table on disk called the Master File Table (MFT). Each directory entry points to an entry in the MFT. The MFT entry will contain several data fields about the file, including security information and a list of pointers to allocated clusters. The MFT entries are a fixed size. Once the MFT is filled with pointers to clusters, another MFT entry will be allocated to hold additional cluster pointers. Thus, this system uses indexed clusters, with the MFT entry acting as the file index block. Figure 6.4 shows the characteristics and structure of the NTFS MFT.

| | |
|---|---|
| Cluster size | 512 to 64K |
| Maximum partition size | 256 terabytes |
| Maximum file size | 16 terabytes |
| Filenames | Unicode characters |
| File permissions supported? | Yes |
| Transaction log? | Yes |

(a) Microsoft NTFS File System Characteristics




(a) Microsoft NTFS MFT Structure

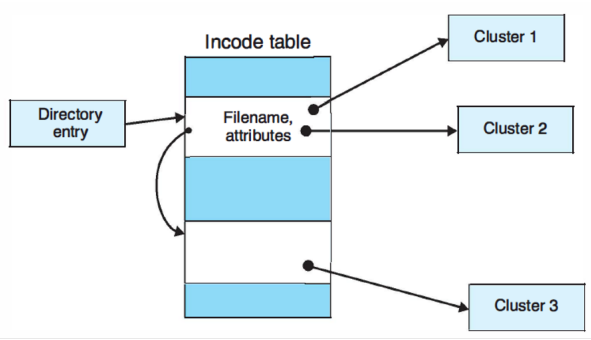
**Figure 6.4:** Microsoft NTFS System

- **Linux Ext2 and Ext3 Systems**: The Linux Ext2 File System uses indexed clusters, similar to the Microsoft NTFS file system. Some important differences between the Ext2 and NTFS systems include the following:
  - Ext2 does not support transaction logging (journaling). Ext3 is an extension of the Ext2 file system and does have a journaling capability.
  - While Ext2 supports file permissions, it does not provide as much control over the permissions as NTFS.

  Figure 6.5 shows the characteristics and structure of the Linux Ext2 file system.

| | |
|---|---|
| Cluster size | 1 to 4K |
| Maximum partition size | 4 terabytes |
| Maximum file size | 4 terabytes |
| Filenames | 8 bit ASCII characters |
| File permissions supported? | Yes |
| Transaction log? | Ext2 : No |

(a) Linux Ext2 File System Characteristics




(a) Linux Ext2 File System Structure

**Figure 6.5:** Linux Ext2 File System System

- **Mac OS X HFS+ Systems**: HFS+ is the standard filesystem on Mac OS X and supports transaction logging (journaling), long filenames, symbolic links, and other features. Figure 6.6 shows the characteristics of the Mac OS X HFS+ file system.

| | |
|---|---|
| Cluster size | 512 bytes or larger |
| Maximum partition size | 2 terabytes |
| Maximum file size | |
| Filenames | Unicode characters |
| File permissions supported? | Yes |
| Transaction log? | Yes |

**Figure 6.6:** MAC OS X HFS+ File System Characteristics

- **Other file systems**: There are many other types of file systems. Examples include the following:
  - *CD-ROMs*: The data on CD-ROMs is formatted with a file system scheme called ISO 9660, which allocates contiguous space for each file on the disk. Figure 6.7 shows the characteristics of the ISO 9660 file system.

| | |
|---|---|
| Cluster size | 512 bytes or larger |
| Maximum partition size | 2 terabytes |
| Maximum file size | |
| Filenames | Unicode characters |
| File permissions supported? | Yes |
| Transaction log? | Yes |

**Figure 6.7:** ISO 9660 File System Characteristics

  - *Network File System (NFS)*: A Network File System (and other similar systems) allows a process on one computer to access a file that physically resides on another computer across the network. This file system does not implement any disk space allocation, but provides the mechanisms for the various file and directory access methods to communicate across the network.

# Chapter 7

# Practicals

### 7.0.1 Booting a Windows Computer in Safe Mode

Why Safe Mode?

- A special way for Windows to load when there is a system-critical problem that interferes with the normal operation of Windows.
- Allow you to troubleshoot Windows and try to determine what is causing it to not function correctly.
- Once you have corrected the problem, then you can reboot and Windows will load normally.

**Booting Windows 7 in Safe Mode**

1. Start with computer turned off or restart it
2. As the computer starts up, repeatedly tap the F8 key
3. Use the arrow keys to highlight Safe Mode, and press ENTER
4. To leave Safe Mode, restart the computer normally

**Booting Windows 10 in Safe Mode: Way 1**

1. Open the Run window - by pressing Windows key and R key together
2. Type msconfig in the Run window and press enter
3. On the Boot tab, check the Safe boot
4. Select Minimal, Apply settings and click Ok.
5. Restart system to effect changes and boot into safe mode.

### 7.0.2 Configuring Boot Order

1. Reboot the computer
2. Watch the start-up screen for information about accessing the BIOS.
3. Press the appropriate key. Often, this is either DEL or F2, F12, or another Function key.
4. Work through the tabs or category options to locate information about the boot order.
5. Configure the first boot device, second boot device, and third boot device, as applicable.
6. Save your changes and exit.

### 7.0.3 Detecting the Number of Processors on a Computer

**Windows 7 Computer**

1. Left- click the Windows Start button.
2. Right-click Computer
3. Left-click properties
4. Review the processor information offered.

**Windows 10 Computer - Way 1**

1. On any opened Folder
2. Right click on "This PC"
3. Left-click properties
4. Review the processor information offered.

**Windows 10 Computer - Way 2**

1. Right click on Taskbar
2. Left-click on Task Manager
3. Click the "Performance" tab
4. Select CPU under performance
5. Review the processor information offered.

**Windows 10 Computer - Way 3**

1. Type WMIC CPU Get /Format:List in the command prompt
2. Press Enter
3. Review the processor information offered.

**Linux Computer**

1. Click System Settings.
2. Click Details.
3. Select Overview.
4. Review the processor information offered.

### 7.0.4 Allocating Processor Resources

**Windows 7**

1. Left-click the Windows start button.
2. Right-click Computer.
3. Left-click properties.
4. Click Advanced System settings (on the left).
5. Under performance, click settings.
6. Click the Advanced tab.
7. From the Visual Effects tab, you can also increase performance by disabling specific visual featured.

NOTE: Performance increase is often not noticeable.

**Windows 10**

1. On any opened Folder,
2. Right-click "This PC".
3. Left-click properties.
4. Click Advanced System settings (on the left).
5. Under performance, click settings.
6. Click the Advanced tab.
7. From the Visual Effects tab, you can also increase performance by disabling specific visual featured.

NOTE: Performance increase is often not noticeable.

### 7.0.5 Opening the Command Prompt - Windows PC

**Using The Run Window**

1. Press $Win$+R (windows logo key and "R") on your keyboard to open the Run Window.
2. Type cmd or cmd.exe and press Enter or click/tap OK

**From the Start Menu**

1. Open the Start Menu
2. For Windows 7: Go to All Programs → Accessories
3. For Windows 10: Go to All Programs → Windows System
4. Click on the Command Prompt short-cut.

**By Search**

1. Open the start Menu
2. Type cmd in the "Search bar"
3. Click on the Command Prompt

**Keyboard and Mouse Short-cut**

1. Without selecting any item in a folder or on the Desktop
2. Hold the shift key and right click
3. Select the Open Command Window Here

### 7.0.6 Opening the Command Prompt - Linux PC

**By Search**

1. Open the Dash by clicking the Ubuntu icon in the upper-left
2. Type terminal
3. Select the Terminal application from the results

**Keyboard Short-cut**

1. Keyboard shortcut: Ctrl + Alt + T

### 7.0.7 Starting Applications from Command Prompt

**Windows PC**

1. Type "start" and the application name in the command prompt.
   - Example : start firefox
   - This will open firefox if it is installed
   - This can also be used in opening folders. E.g: start downloads

**Linux PC**

1. Type the application name in the terminal (command prompt).
   - Example 1: firefox
   - Example 2: firefox &
     - This frees the command prompt to enable you start other applications

### 7.0.8 Closing Applications from Command Prompt

**Windows PC - Using process name**

1. Type taskkill /im + process name.
   - Example : taskkill /im firefox.exe
   - im → Image name
   - To see the process name, type tasklist and press the enter key in command prompt

**Windows PC - Using Process ID**

1. Type taskkill /PID + Process ID
   NOTE: To force kill a process, add /F after process name or ID

- Example: Taskkill /im firefox.exe /F

**Linux PC - Using process name**

1. Type pkill + process name.
   - Example : pkill + firefox
   - To see the process name and ID, type ps - A and enter in terminal

**Linux PC - Using Process ID**

1. Type kill + Process ID

### 7.0.9 Some Relevant Command Line Commands

**Directory Commands**

| Action | Unix Command | Windows Command |
|---|---|---|
| Check current Print Working Directory | pwd | cd |
| Return to user's home folder | cd | cd/ |
| Up one folder | cd .. | cd .. |
| Make directory | mkdir | md |
| Remove directory | rmdir | rd |
| List directory | dir | dir |
| List directory tree | ls | tree |

**File Commands**

**Command Prompt Commands**

## 7.1 Simple Operating System Design

### 7.1.1 Materials Needed

1. Microsoft Visual Studio 2019 or above (any edition)

| Action | Unix Command | Windows Command |
|---|---|---|
| Copy old.file to new.file | cp old.file new.file | copy old.file new.file |
| Remove file (new.txt) | rm new.txt | del new.txt |
| View a file | vi file.txt | file.txt |
| Edit a file | pico file.txt | file.txt |
| Sort file content | sort file.txt | sort file.txt |
| Sort file content to new file | sort file.txt > new file | sort file.txt > new file |

| Action | Unix Command | Windows Command |
|---|---|---|
| Display line of text | echo text | echo text |
| Clear Screen | clear | cls |
| Exit | exit | exit |

2. COSMOS[1] User Development Kit
3. A Virtualization Software (E.g VMware Player)
    - To run the Operating System

**Prerequisite Knowledge**

- Basics of C#, C++, Java .net

### 7.1.2 COSMOS - (C# Open Source Managed Operating System)

COSMOS is an operating system development kit which uses Visual Studio as its development environment. Despite C# in the name, any .NET based language can be used including VB.NET, Fortran, Delphi Prism, IronPython, F# and more. COSMOS is not an operating system in the traditional sense, it is an "Operating System Kit" that lets you create operating systems just as Visual Studio and C# normally let you create applications.

#### 7.1.2.1 Creating a Simple Operating System

1. Open visual studio and navigate to File > New Project
2. In the New Project dialog box, select COSMOS and then COSMOS "C#, VB.NET or F# Operating System" in the templates box.
3. Name the project and click OK
4. Click on the Kernel.cs to load it.
5. Edit the code in "void BeforeRun()" to display your preferred text when the Operating System loads
6. Edit the code in "void Run()" to perform the function for which your Operating System was designed.
7. Run your application.

[1] C# Open Source Managed Operating System