Department of Computer Science, University of Benin, Benin City
Lecture Note on Operating Systems (CSC411)

Course Outline

1. The meaning and Essence of Operating Systems
2. The Position and Roles of Operating Systems in Computer Systems
3. Types of Computer Systems Architecture and Environments
4. Overview of Operating Systems
5. Operating System as Computer Resource Manager: Process and Data
6. Virtualization and Operating Systems Components
7. Network and Distributed Operating Systems
8. Case Studies: Linux (Ubuntu) and Windows 10 Operating Systems

N.B: Proficiency in Pascal, C/C++ and Java Programming Languages is assumed

Recommended Texts:

1. Silberschatz A, Galvin PB and Gagne G (2018). Operating System Concepts, 10th Edition. USA: John Wiley and Sons
2. Stallings W. (2012). Operating Systems: Internals and Design Principles, 7th Edition. USA: Prentice Hall
3. Tanenbaum, AS (2009). Modern Operating Systems, 3rd Edition. USA: Pearson Education International

Grading Eligibility and Rules

1. To be eligible for course examination and grading, the regular students MUST meet 50% Attendance while the Carry-over students MUST meet 40% Attendance. The University rule is 75%. However, the Course Lecturers compassionately permitted the lowering of the 75% attendance as the examination/grading eligibility requirement, as above, only for this session as a palliative measure for the students to accommodate the hardship that may result from the recent fuel subsidy removal policy of the Federal Government of Nigeria. To this end, Lecture Notes on each topic will be made available via the University Learning Management System at the end of each lecture.
2. Continuous Assessment Test shall account for 25% of the Examination Score (10% for Percentage Attendance; 10% for Tests/Quizzes/Laboratory Assignments on Linux; 5% for Tests/Quizzes/Laboratory Assignments on Windows 10).
3. The Final Examination shall account for 75% of the Examination Score
4. A student may be asked out of the lecture hall for improper behavior and dressing and have his/her attendance for that day cancelled
5. Late coming and leaving the lecture hall without permission will not be tolerated
6. In case of ill health, a note to that effect from the UNIBEN health center MUST reach the course lecturer before the end of each lecture the student may miss due to ill health.
7. Students that may be away on University approved competitions must submit letter of release, stating for how long, by the University Management **before departure**.

Week One – Day 1

The Meaning and Essence of Operating Systems

An operating system (OS) is a computer program or software environment that enables users (humans, machines, or programs) conveniently, efficiently and reliably exploit the computer resources. These resources include the information stored in the system (both data and code), as well as the processor (CPU), memory (primary storage), secondary storage, tertiary storage, communications equipment, terminals, and computer busses: system bus (address bus, data bus, and control bus) which could be internal bus, external bus, or expansion bus; and directly linked to main memory (front-side bus) or via cache (backside bus). However, considering the diverse, evolving and increasing services of OS, the diverse computer systems architecture and environments, technological advancements, increasing and changing human requirements and expectations, no completely adequate or universally accepted definition of an OS exist.

For the purpose of this course, an OS consists of a kernel, middleware frameworks that ease application development and seamlessly provide features, and systems programs that aid in managing the system and ensuring the continuous and smooth running of the computer systems. Thus, we can say, generally, that an OS is a computer program that controls and coordinates the execution of application programs and serves as a mediator between the application programs and the computer hardware (as shown in Figure 1) ensuring the seamless utilization of the computer resources. It should have the following objectives: (i) Convenience, (ii) Efficiency, (iii) Extensibility, (iv) Availability, and (v) Reliability.

Typically, the OS provides services in at least the following areas:

(i) **User interface**: Almost all operating systems have a user interface (UI). This interface can take several forms. Most commonly, a graphical user interface (GUI) is used. Here, the interface is a window system with a mouse that serves as a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text. Mobile systems such as phones and tablets provide a touch-screen interface, enabling users to slide their fingers across the screen or press buttons on the screen to select choices. Another option is a command-line interface (CLI), which uses text commands and a method for entering them (say, a keyboard for typing in commands in a specific format with specific options). Some systems provide two or all three of these variations.

(ii) **Program development:** The OS provides a variety of facilities and services, such as editors and debuggers, to assist the programmer in creating programs. Typically, these services are in the form of utility programs that, while not strictly part of the core of the OS, are supplied with the OS and are referred to as application program development tools.

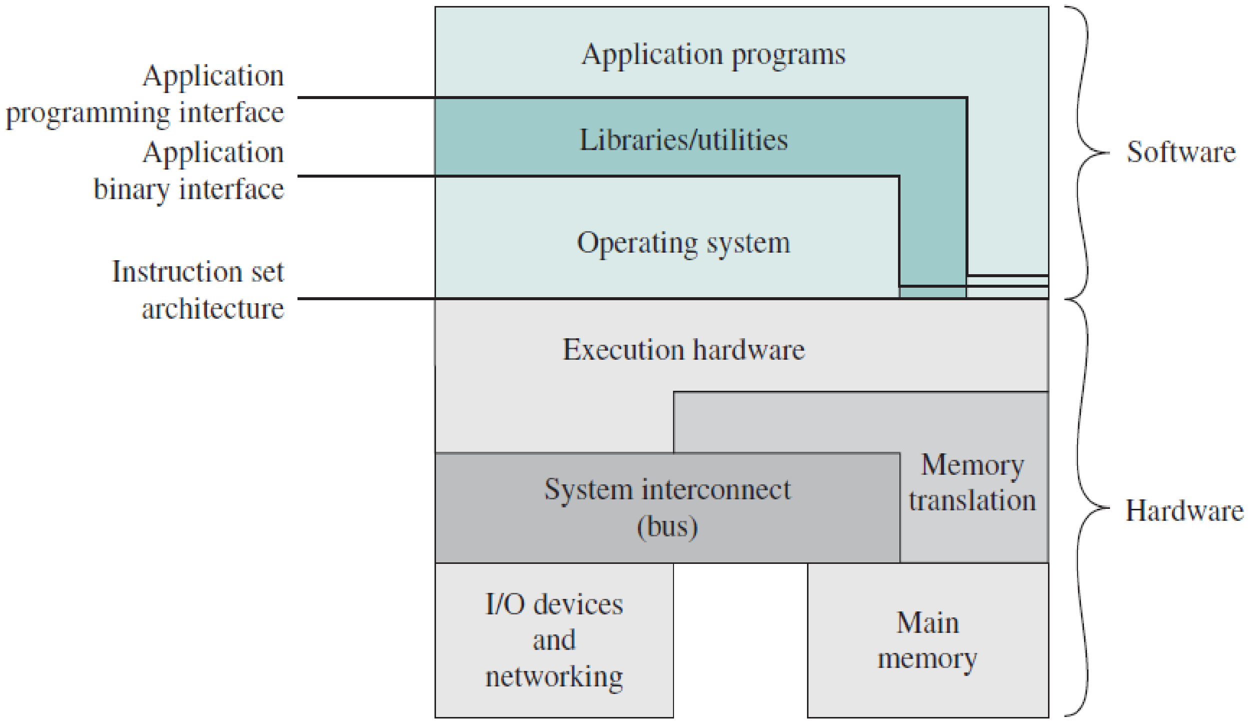

Application programs
Libraries/utilities
Operating system
Execution hardware
System interconnect (bus)
Memory translation
I/O devices and networking
Main memory
Application programming interface
Application binary interface
Instruction set architecture
Software
Hardware

Figure 1: The Computer Hardware and Software Structure (Stallings, 2012)

(iii) **Program execution:** A number of steps need to be performed to execute a program. Instructions and data must be loaded into main memory, I/O devices and files must be initialized, and other resources must be prepared. The OS handles these scheduling duties for the user.

(iv) **Resource allocation**: When there are multiple processes running at the same time, resources must be allocated to each of them. The operating system manages many different types of resources. Some (such as CPU cycles, main memory, and file storage) may have special allocation code, whereas others (such as I/O devices) may have much more general request and release code. For instance, in determining how best to use the CPU, operating systems have CPU-scheduling routines that take into account the speed of the CPU, the process that must be executed, the number of processing cores on the CPU, and other factors. There may also be routines to allocate printers, USB storage drives, and other peripheral devices.

(v) **Access to I/O devices:** Each I/O device requires its own peculiar set of instructions or control signals for operation. The OS provides a uniform interface that hides these details so that programmers can access such devices using simple reads and writes.

(vi) **Controlled access to files:** For file access, the OS must reflect a detailed understanding of not only the nature of the I/O device (disk drive, tape drive) but also the structure of the data contained in the files on the storage medium. In the case of a system with multiple users, the OS may provide protection mechanisms to control access to the files.

(vii) **System access:** For shared or public systems, the OS controls access to the system as a whole and to specific system resources. The access function must provide protection of resources and data from unauthorized users and must resolve conflicts for resource contention.

(viii) **Communications**: There are many circumstances in which one process needs to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a network. Communications may be implemented via shared memory, in which two or more processes read and write to a shared section of memory, or message passing, in which packets of information in predefined formats are moved between processes by the operating system.

(ix) **Error detection and response:** A variety of errors can occur while a computer system is running. These include internal and external hardware errors, such as a memory error, or a device failure or malfunction; and various software errors, such as division by zero, attempt to access forbidden memory location, and inability of the OS to grant the request of an application. In each case, the OS must provide a response that clears the error condition with the least impact on running applications. The response may range from ending the program that caused the error, to retrying the operation, to simply reporting the error to the application.

(x) **Accounting:** A good OS will collect usage statistics for various resources and monitor performance parameters such as response time. On any system, this information is useful in anticipating the need for future enhancements and in tuning the system to improve performance. On a multiuser system, the information can be used for billing purposes.

Figure 1 also indicates three key interfaces in a typical computer system:

(xi) **Instruction set architecture (ISA)**: The ISA defines the repertoire of machine language instructions that a computer can follow. This interface is the boundary between hardware and software. Note that both application programs and utilities may access the ISA directly. For these programs, a subset of the instruction repertoire is available (user ISA). The OS has access to additional machine language instructions that deal with managing system resources (system ISA).

(xii) **Application binary interface (ABI)**: The ABI defines a standard for binary portability across programs. The ABI defines the system call interface to the operating system and the hardware resources and services available in a system through the user ISA.

(xiii) **Application programming interface (API)**: The API gives a program access to the hardware resources and services available in a system through the user ISA supplemented with high-level language (HLL) library calls. Any system calls are usually performed through libraries. Using an API enables application software to be ported easily, through recompilation, to other systems that support the same API.

The above service areas, as highlighted, sufficiently explain why an OS is a critical component of the computer system that should be well understood by any computing professional. Obviously, as almost all code runs on top of an operating system, knowledge of how operating systems work is crucial to proper, efficient, effective, and secure programming. Understanding the fundamentals of operating systems, how they drive computer hardware, and what they provide to applications is not only essential to those who program them but also highly useful to those who write programs on them and use them.

Week One – Day 2

The Position and Roles of Operating Systems in Computer Systems

It is now evident that OS covers many roles and functions in the computer. Computers are present in diverse form, architecture and organization within toasters, cars, spacecraft, homes, and business. They are the basis for game machines, cable TV tuners, smart factories and cities, and industrial control systems. This diversity notwithstanding, OS plays the unique role of a resource manager in the computer. The resources include CPU time, memory space, storage space, and I/O devices. As a resource manager, it takes up the roles of (i) process management, (ii) memory management, (iii) File-system management, (iv) Mass storage management, (v) Cache management, (vi) I/O system management. The OS also provides the environment within which programs are executed. As programs execution container, the OS is capable of performing the following operations: (vii) multiprogramming and multitasking, (viii) Dual-mode and multimode, (ix) virtualization, and (x) timing. More so, the OS serves as the Chief Security Officer of the computer system (CS) where it plays the roles of: (xi) privacy protection, and (xii) security. Finally, OS can now position itself as lobbyist negotiating and connecting physically separate, possibly heterogeneous computer systems, and enabling the different computer systems to communicate with and use others resources (network OS) or exploit the resources of other computer systems transparently as their internal resources (distributed OS). Hence, the additional roles of: (xiii) networking, and (xiv) migration – data migration, computation migration, and process migration, on OS. The following subsections discuss these varying positions and roles of OS in CS.

2.1. Position 1 – Computer Resource Manager

The dominant position of OS in a CS is as the manager of CS resource: CPU, memory space, file-storage space, and I/O devices. Generally, a manager is an entity that plans, coordinates, controls, and accounts for the exploitation of systems resources for the efficient, effective, reliable, and sustainable execution of the system's service in strong alignment with the system's objectives for optimum (excellent) system's performance and stakeholders (users) experience. This is exactly what OS is to a CS. As a manager of CS, OS performs several roles as earlier hinted and discussed as follows:

(i) **Process Management:** A program can do nothing unless its instruction is executed by the CPU. A process is simply a program or set of programs in execution. Typically, a process or service execution is a unit of work (task) in a computer system. Each process consists of a nonempty set of threads – sovereign executable process units or blocks of instructions. A computer system, online, consists of collection of processes which on a single core, executes concurrently by multiplexing, and on multiple cores executes in parallel. The OS performs the following process management tasks:

a. Creating and deleting both user and system processes
b. Scheduling processes and threads on the CPUs
c. Suspending and resuming processes
d. Providing mechanisms for process synchronization
e. Providing mechanisms for process communication

(ii) **Memory Management:** Instructions and data from the CPU and I/O devices are only executable or consumable only if they are mapped to absolute addresses and loaded to memory. The main memory is generally the only large storage device the CPU is able to address and access directly. Similarly, instructions must be in memory for the CPU executes them. Eventually, when the program terminates, its memory including those of associated data is declared available, and the next program can then loaded with its associated data and executed. Thus, the operating system is responsible for the following activities in connection with memory management:

- **a.** Keeping track of which parts of memory are currently being used and which process is using them
- **b.** Allocating and de-allocating memory space as needed
- **c.** Deciding which processes (or parts of processes) and data to move into and out of memory

(iii) **File-System Management:** File is simply a logical storage unit of data/information. The OS implements this abstract concept of data organization called file, or its collections (called directory), by managing mass storage media and the devices that control them. Finally, when multiple users have access to files, it may be desirable to control which user may access a file and how that user may access it (for example, read, write, append). To this end, the operating system is responsible for the following activities in connection with file management:

- **a.** Creating and deleting files
- **b.** Creating and deleting directories to organize files
- **c.** Supporting primitives for manipulating files and directories
- **d.** Mapping files onto mass storage
- **e.** Backing up files on stable (nonvolatile) storage media

(iv) **Mass-Storage Management:** As we have already seen, the computer system must provide secondary storage to back up main memory. Most modern computer systems use HDDs and NVM devices as the principal on-line storage media for both programs and data. Most programs—including compilers, web browsers, word processors, and games—are stored on these devices until loaded into memory. The programs then use the devices as both the source and the destination of their processing. Hence, the proper management of secondary storage is of central importance to a computer system. The operating system is responsible for the following activities in connection with secondary storage management:

a. Mounting and unmounting
b. Free-space management
c. Storage allocation
d. Disk scheduling
e. Partitioning
f. Protection

Because secondary storage is used frequently and extensively, it must be used efficiently. The entire speed of operation of a computer may hinge on the speeds of the secondary storage subsystem and the algorithms that manipulate that subsystem. At the same time, there are many uses for storage that is slower and lower in cost (and sometimes higher in capacity) than secondary storage. Backups of disk data, storage of seldom-used data, and long-term archival storage are some examples. Magnetic tape drives and their tapes and CD DVD and Blu-ray drives and platters are typical tertiary storage devices (see Figure 2).

Tertiary storage is not crucial to system performance, but it still must be managed. Some operating systems take on this task, while others leave tertiary-storage management to application programs. Some of the functions that operating systems can provide include mounting and unmounting media in devices, allocating and freeing the devices for exclusive use by processes, and migrating data from secondary to tertiary storage.

(v) **Cache Management:** Caching is an important migration principle that employs a very fast but limited storage system called cache to temporarily hold data or instructions in demand, or migrate back to memory stale data or instruction, for efficiency. The caching principles also involve high speed registers. Obviously, the caching process usually results in duplication of data and instruction. The OS can help manage this migration and guarantee cache coherency or replica consistency. This may be explicit or implicit depending on hardware design and the controlling OS.

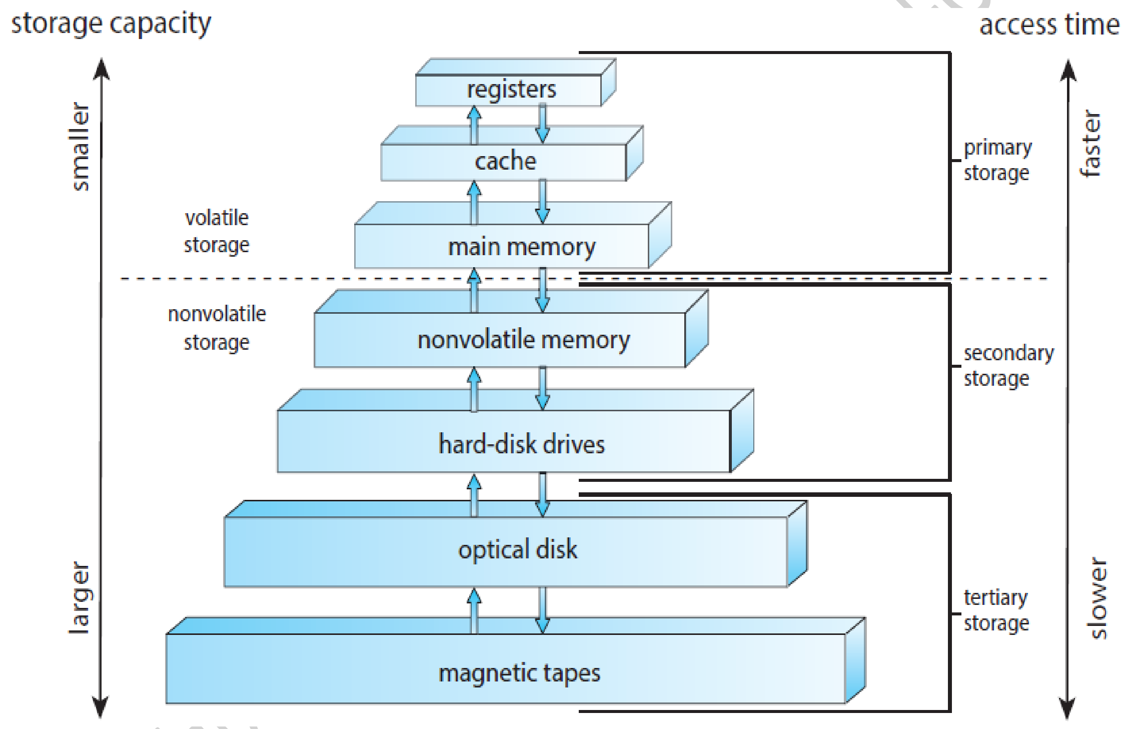


Figure 2: Storage Device Hierarchy

(vi) **I/O System Management:** One of the roles of OS as CS manager is to hide the peculiarities of specific hardware device from the user. Only the device driver knows the peculiarities of its device to which it is assigned. Thus, I/O system management is concerned with the management of:

a. Buffering, caching, and spooling components of I/O devices

b. Device-driver interface

c. Drivers for specific hardware devices

2.2. Position 2 – Program Execution Environment

An OS besides being the CS manager also provides the environment within which programs are executed. Once the CS is rebooted or powered up, the OS kernel or nucleus is loaded to memory by the bootstrap program in firmware. Once the kernel is loaded and executing, it then serves as the execution environment for all other programs, coordinating their resource utilization by performing the operations as earlier hinted and discussed as follows:

(vii) **Multiprogramming and Multitasking:** One of the roles of OS is to enable the simultaneous utilization of the CS resources multiple programs, allowing several programs to be in memory and execute simultaneously. This enables optimum utilization of CS resources, improved CS efficiency and user experience. While multiprogramming switch CPU due to idleness, multitasking switch CPU based on time slice. Both execution type will require scheduling (akin to giving appointment) and synchronization (akin to ordering) to guarantee process execution fairness.

(viii) **Dual-Mode and Multimode Operation:** An OS though housing a program is on its own a program too. So, in attempts to distinguish program executions and their privileges, the OS need to execute in at least two mode of operations – user mode (for application programs) and kernel mode for itself. The dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another. We accomplish this protection by designating some of the machine instructions that may cause harm as privileged instructions. The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.

The instruction to switch to kernel mode is an example of a privileged instruction. Some other examples include I/O control, timer management, and interrupt management. The concept of modes can be extended beyond two modes. For example, Intel processors have four separate protection rings, where ring 0 is kernel mode and ring 3 is user mode. (Although rings 1 and 2 could be used for various operating-system services, in practice they are rarely used). ARMv8 systems have seven modes. CPUs that support virtualization frequently have a separate mode to indicate when the virtual machine manager (VMM) is in control of the system. In this mode, the VMM has more privileges than user processes but fewer than the kernel. It needs that level of privilege so it can create and manage virtual machines, changing the CPU state to do so.

(ix) **Virtualization:** virtualization enables multiple presentation of same computer via the eye of OS as multiple execution environments or computer systems. Virtualization make the OS assume the place of the hardware and even enable OS to run within another OS. One notable advantage of virtualization is portability – an OS, say, can be made to run on a machine it was not natively designed for.

(x) **Timing:** as an execution environment that accommodates multiple programs and processes, the timing role is critical to guarantee fair resource allocation. The timer ensures that no process can gain control of the CPU without eventually relinquishing control.

2.3. Position 3 – Chief Security Officer of the Computer System

The OS ensures that access to CS resources is regulated and all processes execute without any hitch. Thus, the OS help regulate access and ensure a seamless execution of programs and protection of CS resources including data. This is discussed as follows:

(xi) **Privacy Protection:** Since a computer system accommodates multiple users and allows the concurrent execution of multiple processes, then access to data must be regulated. For that purpose, mechanisms ensure that files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system. For example, memory-addressing hardware ensures that a process can execute only within its own address space. Device-control registers are not accessible to users, so the integrity of the various peripheral devices is protected.

Privacy protection, then, is any mechanism for controlling the access of processes or users to the resources defined by a computer system. This mechanism must provide means to specify the controls to be imposed and to enforce the controls. Privacy protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by another subsystem that is malfunctioning. Furthermore, an unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user. A protection-oriented system provides a means to distinguish between authorized and unauthorized usage.

(xii) Security: A system can have adequate protection but still be prone to failure and allow inappropriate access. Consider a user whose authentication information (her means of identifying herself to the system) is stolen. Her data could be copied or deleted, even though file and memory protection are working. It is the job of security to defend a system from external and internal attacks. Such attacks spread across a huge range and include viruses and worms, denial-of-service attacks (which use all of a system's resources and so keep legitimate users out of the system), identity theft, and theft of service (unauthorized use of a system). Prevention of some of these attacks is considered an operating system function on some systems, while other systems leave it to policy or additional software. Due to the alarming rise in security incidents, operating system security features are a fast-growing area of research and implementation.

Privacy protection and security require the system to be able to distinguish among all its users. Most operating systems maintain a list of user names and associated user identifier (user IDs). In Windows parlance, this is a security ID (SID). These numerical IDs are unique, one per user. When a user logs in to the system, the authentication stage determines the appropriate user ID for the user. That user ID is associated with all of the user's processes and threads. When an ID needs to be readable by a user, it is translated back to the user name via the user name list.

In some circumstances, we wish to distinguish among sets of users rather than individual users. For example, the owner of a file on a UNIX system may be allowed to issue all operations on that file, whereas a selected set of users may be allowed only to read the file. To accomplish this, we need to define a group name and the set of users belonging to that group. Group functionality can be implemented as a system-wide list of group names and group identifier.

A user can be in one or more groups, depending on operating-system design decisions. The user's group IDs are also included in every associated process and thread. In the course of normal system use, the user ID and group ID for a user are sufficient. However, a user sometimes needs to escalate privileges to gain extra permissions for an activity. The user may need access to a device that is restricted, for example. Operating systems provide various methods to allow privilege escalation. On UNIX, for instance, the *setuid* attribute on a program causes that program to run with the user ID of the owner of the file, rather than the current user's ID. The process runs with this effective UID until it turns off the extra privileges or terminates.

2.4. Position 4 – Computer System's Lobbyist

OS as CS lobbyist enables multiple CS to work as one or like one unlike virtualization the makes a CS to work as many. This brings us to the networking and cooperation (migration) roles of OS discussed as follow:

(xiii) **Networking:** A network operating system provides an environment in which users can access remote resources (implementing resource sharing) by either logging in to the appropriate remote machine or transferring data from the remote machine to their own machines. Currently, all general-purpose operating systems, and even embedded operating systems such as Android and iOS, are network operating systems. An important function of a network operating system is to allow users to log in remotely.

Another major function of a network operating system is to provide a mechanism for remote file transfer from one machine to another. In such an environment, each computer maintains its own local file system. If a user at computer A wants to access a file owned by another located on another computer B, then the file must be copied explicitly from computer B to the user computer A. The communication is one-directional and individual. The Internet provides a mechanism for such a transfer with the file transfer protocol (FTP) and the more private secure file transfer protocol (SFTP).

Basic cloud-based storage applications allow users to also transfer files much as with FTP. Users can upload files to a cloud server, download files to the local computer, and share files with other cloud-service users via a web link or other sharing mechanism through a graphical interface. Common examples include Dropbox and Google Drive.

An important point about SSH (protocol for remote access), FTP, and cloud-based storage applications is that they require the user to change paradigms. FTP, for example, requires the user to know a command set entirely different from the normal operating system commands. With SSH, the user must know appropriate commands on the remote system. With cloud-based storage applications, users may have to log into the cloud service (usually through a web browser) or native application and then use a series of graphical commands to upload, download, or share files. Obviously, users would find it more convenient not to be required to use a different set of commands. Distributed operating systems are designed to address this problem.

(xiv) **Migration:** In a distributed operating system, users access remote resources in the same way they access local resources. Data and process migration from one site to another is under the control of the distributed operating system. Depending on the goals of the system, it can implement data migration, computation migration, process migration, or any combination thereof.

Suppose a user on site A wants to access data (such as a file) that reside at site B. The system can transfer the data by one of two basic methods. One approach to data migration is to transfer the entire file to site A. From that point on, all access to the file is local. When the user no longer needs access to the file, a copy of the file (if it has been modified) is sent back to site B. Even if only a modest change has been made to a large file, all the data must be transferred. This mechanism can be thought of as an automated FTP system. This approach was used in the Andrew file system, but it was found to be too inefficient.

The other approach is to transfer to site A only those portions of the file that are actually *necessary* for the immediate task. If another portion is required later, another transfer will take place. When the user no longer wants to access the file, any part of it that has been modified must be sent back to site B. (Note the similarity to demand paging.) Most modern distributed systems use this approach. Whichever method is used, data migration includes more than the mere transfer of data from one site to another. The system must also perform various data translations if the two sites involved are not directly compatible (for instance, if they use different character-code representations or represent integers with a different number or order of bits).

In some circumstances, we may want to transfer the computation, rather than the data, across the system; this process is called computation migration. For example, consider a job that needs to access various large files that reside at different sites, to obtain a summary of those files. It would be more efficient to access the files at the sites where they reside and return the desired results to the site that initiated the computation. Generally, if the time to transfer the data is longer than the time to execute the remote command, the remote command should be used. Such a computation can be carried out in different ways. Suppose that process P wants to access a file at site A. Access to the file is carried out at site A and could be initiated by an RPC. An RPC uses network protocols to execute a routine on a remote system. Process P invokes a predefined procedure at site A. The procedure executes appropriately and then returns the results to P. Alternatively, process P can send a message to site A. The operating system at site A then creates a new process Q whose function is to carry out the designated task. When process Q completes its execution, it sends the needed result back to P via the message system. In this scheme, process P may execute concurrently with process Q. In fact, it may have several processes running concurrently on several sites.

Either method could be used to access several files (or chunks of files) residing at various sites. One RPC might result in the invocation of another RPC or even in the transfer of messages to another site. Similarly, process Q could, during the course of its execution, send amessage to another site, which in turn would create another process. This process might either send a message back to Q or repeat the cycle.

A logical extension of computation migration is process migration. When a process is submitted for execution, it is not always executed at the site at which it is initiated. The entire process, or parts of it, may be executed at different sites. This scheme may be used for several reasons:

a. Load balancing. The processes (or subprocesses) may be distributed across the sites to even the workload.
b. Computation speedup. If a single process can be divided into a number of subprocesses that can run concurrently on different sites or nodes, then the total process turnaround time can be reduced.
c. Hardware preference. The process may have characteristics that make it more suitable for execution on some specialized processor (such as matrix inversion on a GPU) than on a microprocessor.
d. Software preference. The process may require software that is available at only a particular site, and either the software cannot be moved, or it is less expensive to move the process.
e. Data access. Just as in computation migration, if the data being used in the computation are numerous, it may be more efficient to have a process run remotely (say, on a server that hosts a large database) than to transfer all the data and run the process locally.

We use two complementary techniques to move processes in a computer network. In the first, the system can attempt to hide the fact that the process has migrated from the client. The client then need not code her program explicitly to accomplish the migration. This method is usually employed for achieving load balancing and computation speedup among homogeneous systems, as they do not need user input to help them execute programs remotely. The other approach is to allow (or require) the user to specify explicitly how the process should migrate. This method is usually employed when the process must be moved to satisfy a hardware or software preference.

You have probably realized that the World Wide Web has many aspects of a distributed computing environment. Certainly it provides data migration (between a web server and a web client). It also provides computation migration. For instance, a web client could trigger a database operation on a web server. Finally, with Java, Javascript, and similar languages, it provides a form of process migration: Java applets and Javascript scripts are sent from the server to the client, where they are executed. A network operating system provides most of these features, but a distributed operating system makes them seamless and easily accessible. The result is a powerful and easy-to-use facility—one of the reasons for the huge growth of the World Wide Web.

Week Two – Day 1

Types of Computer Systems Architecture and Environments

A computer system can be categorized based on the number of general-purpose processors used and the computing paradigm it supports. These CS architectures and computing paradigms influence to a large extent the type/nature of OS that manage them.

Classification of CS by its number of processors makes the following CS evident: (i) Single-Processor System (SPS), (ii) Multi-Processor System (MPS), and (iii) Clustered Computer System (CCS).

(i) Single-Processor System (SPS): CS with a single processor containing one CPU with a single processing core.

(ii) Multi-Processor System (MPS): CS with two or more processors, each with a single-core CPU. The processors share the computer bus and sometimes the clock, memory, and peripheral devices. The primary advantage of multiprocessor systems is increased throughput. The speed-up ratio with *N* processors is not *N,* however; it is less than *N.* MPS could be symmetric (multiple single core chips), multi-core (one chip with multiple core), non-uniform memory access, or NUMA, (multiple independent core chips), or blade servers (multiple independent CS on board).

(iii) Clustered Computer System (CCS): multiple independent CS on separate board but with shared storage or storage network. It provides high availability, scalability, fault tolerant, and guaranteed responsiveness. CCS can be asymmetric (one CS active at a time, others watching in stand-by), symmetric (multiple CS simultaneously active), parallel clusters (simultaneous access to shared storage, Clustering over WAN (CoWAN), or database cluster (DC).

The possible computing paradigms that influence the nature/type of OS in CS include (iv) Traditional Computing (TC), (v) Mobile Computing (MC), (vi) Client-Server Computing (CSC), (vii) Peer-to-Peer Computing (PPC), (viii) Cloud Computing (CC), and (ix) Real-time/Embedded Computing (REC).

(iv) Traditional Computer (TC): Here, all computing activities are carried within a CS with one or more terminals usually in fixed location. They are associated with conventional PCs, Mini-computers, and Super-computers with characteristic features of heavy OS, high processor capability, large memory capacity, and limited mobility. Supporting OS include Windows and UNIX.

(v) Mobile Computing (MC): In MC the CS has limited processor capability and memory capacity, unlimited mobility with light weight OS. MC typically supports the unique features of mobile devices such as global positioning systems, GPS (location tracking), accelerometers (ability to estimate and respond to vibration and linear change in motion), gyroscopes (responds to angular change or relative orientation to the earth surface), and cellular data networks. Supporting OS include Apple iOS and Google Android. Examples of mobile CS include smartphones, and tablet PCs.

(vi) Client-Server-Computing (CSC): CSC enable a set of dependent and usually light weight computers (clients) simultaneously completes their computational tasks via networks in a sophisticated independent heavy weight computer (server). They should be capable of carrying out computation or CRUD operations in the server.

(vii) Peer-to-Peer Computing (PPC): In PPC, two or more fairly equally sophisticated computers completes their computational tasks using their respective computer resources via a network in a mutually beneficial form. Any computer in PCC can be a client at one time, and a server another time or both simultaneously depending on the active computational task.

(viii) Cloud Computing (CC): Cloud computing is an extension of CSC with virtualization and subscription capabilities. CC systems are ultra-fast with ultra-large memory. CC supports service orientation philosophy where computational resources, entities and systems are enabled as service. Its climax is an interesting world where you do not need to own anything to use it – just pay as you use or use as you can pay. Already, we have various types of CC service: SaaS, PaaS, and IaaS which can be consumed on public cloud, private cloud, hybrid cloud, or multi-cloud.

(ix) Real-Time Embedded Computing (REC): Embedded computers are the most prevalent form of computers in existence. By embedded we mean hidden or transparent to human. REC usually runs real-time OS and thus usually used for control, monitoring, and other autonomous computing tasks. REC is the future of computing, already with us though, where cyber-physical systems will control, coordinate, monitor, and perform both the physical and computational tasks/entities with the risk of leading humanity into singularity. Examples of REC include Internet of Things (IoTs), smart buildings, smart factory or Industry 4.0, robots, and smart vehicles (e.g. "drones: they are unmanned – space, aerial, surface, underwater – vehicles remotely or autonomously operated for precision, indeterminate, impregnable, hazardous, safety and time critical tasks" (Ekuobase, 202X, Chapter 3)).

Week Three

Overview of Operating Systems (OS)

The objectives and functions of OS have been identified and discussed. This week, we shall examine from a bird's eye view the fundamental structure and capabilities of the two dominant OS – Windows, and Linux.

3.1. Overview of Windows OS

Windows, which began with a very different OS, developed by Microsoft for the first IBM personal computer and referred to as MS-DOS, is now a sophisticated multitasking OS, designed to manage the complexity of the modern computing environment, provide a rich platform for application developers, and support a rich set of experiences for users. Like Solaris, Windows is designed to have the features that enterprises need, while at the same time Windows, like MacOS, provides the simplicity and ease-of-use that consumers require. Windows, as with virtually all OS, consists of the user mode components (application-oriented software) and the kernel mode components (core OS software); as shown in Figure 3. Windows 10 in particular can also use its Hyper-V hypervisor to provide orthogonal security model through Virtual Trust Levels (VTLs): VTL 0 (Normal world) and VTL 1 (secure world) in a third mode – Virtual Secure Mode (VSM). In both worlds are the user mode and the kernel mode. The latest version of Windows OS is Windows 11.

The kernel-mode components of Windows are the following:

(i) Executive: Contains the core OS services, such as memory management, process and thread management, security, I/O, and interprocess communication.

(ii) Kernel: Controls execution of the processors. The Kernel manages thread scheduling, process switching, exception and interrupt handling, and multiprocessor synchronization. Unlike the rest of the Executive and the user level, the Kernel's own code does not run in threads.

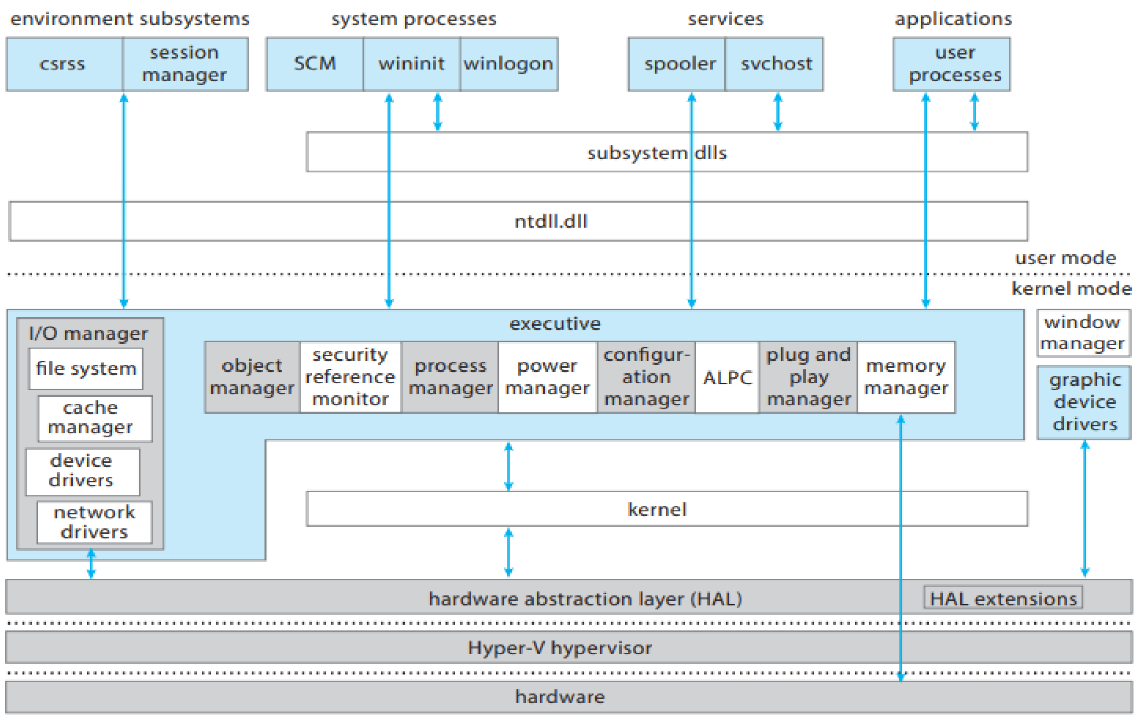


Figure 3: Windows Block Diagram

(iii) Hardware abstraction layer (HAL): Maps between generic hardware commands and responses and those unique to a specific platform. It isolates the OS from platform-specific hardware differences. The HAL makes each computer's system bus, direct memory access (DMA) controller, interrupt controller, system timers, and memory controller look the same to the Executive and Kernel components. It also delivers the support needed for SMP, explained subsequently.

(iv) Device drivers: Dynamic libraries that extend the functionality of the Executive. These include hardware device drivers that translate user I/O function calls into specific hardware device I/O requests and software components for implementing file systems, network protocols, and any other system extensions that need to run in kernel mode.

(v) Windowing and graphics system: Implements the GUI functions, such as dealing with windows, user interface controls, and drawing.

Four basic types of user-mode processes are supported by Windows:

(i) Special system processes: User-mode services needed to manage the system, such as the session manager, the authentication subsystem, the service manager, and the logon process.

(ii) Service processes: The printer spooler, the event logger, user-mode components that cooperate with device drivers, various network services, and many, many others. Services are used by both Microsoft and external software developers to extend system functionality as they are the only way to run background user-mode activity on a Windows system.

(iii) Environment subsystems: Provide different OS personalities (environments). The supported subsystems are Win32 and POSIX. Each environment subsystem includes a subsystem process shared among all applications using the subsystem and dynamic link libraries (DLLs) that convert the user application calls to ALPC calls on the subsystem process, and/or native Windows calls.

(iv) User applications: Executables (EXEs) and DLLs that provide the functionality users run to make use of the system. EXEs and DLLs are generally targeted at a specific environment subsystem; although some of the programs that are provided as part of the OS use the native system interfaces (NT API). There is also support for running 32-bit programs on 64-bit systems.

3.2. Overview of Linux OS

Linux is a variant of UNIX that has gained popularity over the last several decades, powering devices as small as mobile phones and as large as room filling supercomputers.

Linux looks and feels much like any other UNIX system; indeed, UNIX compatibility has been a major design goal of the Linux project. Linux development began in 1991, when a Finnish university student, Linus Torvalds, began creating a small but self-contained kernel for the 80386 processor – a 32-bit processor in Intel's range of PC-compatible CPUs.

The Linux system is composed of three main bodies of code, in line with most traditional UNIX implementations:

1. Kernel. The kernel is responsible for maintaining all the important abstractions of the operating system, including such things as virtual memory and processes.
2. System libraries. The system libraries define a standard set of functions through which applications can interact with the kernel. These functions implement much of the operating-system functionality that does not need the full privileges of kernel code. The most important system library is the C library, known as libc. In addition to providing the standard C library, libc implements the user mode side of the Linux system call interface, as well as other critical system-level interfaces.
3. System utilities. The system utilities are programs that perform individual, specialized management tasks. Some system utilities are invoked just once to initialize and configure some aspect of the system. Others— known as daemons in UNIX terminology—run permanently, handling such tasks as responding to incoming network connections, accepting logon requests from terminals, and updating log files.

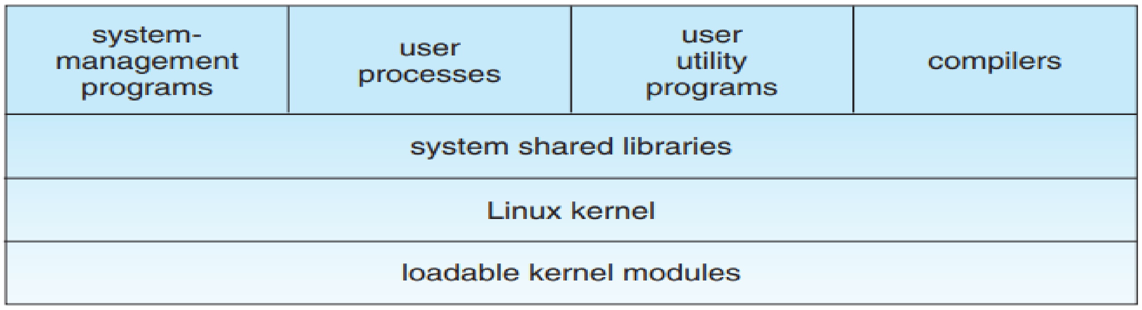


Figure 4: Components of the Linux System

Week Four – Day One

Process Description

A process is simply a program in execution, or a computational entity that can be assigned to and executed on a processor, to perform a given task or job. A program – executable code – becomes a process when loaded to memory. Basically, a process consists of two parts – code and associated set of data. The data can be static, heap dynamic or stack dynamic and are unique to individual process. Two or more processes can have same code but not same data. Also, a process can assume one of the following states depending on its relationship with the processor core and memory: New, Running, Waiting/Blocked, Suspend, Ready, or Terminated as depicted in Figures 5 and 6.

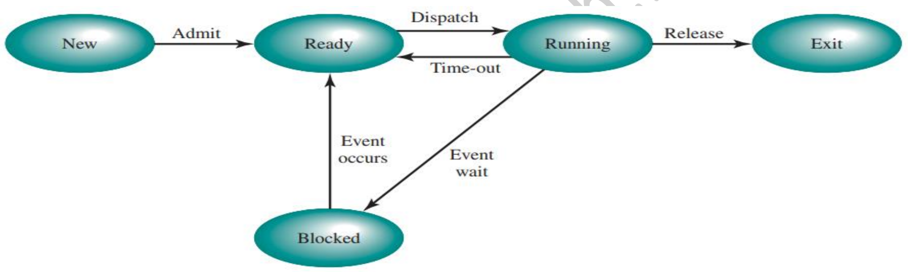


Figure 5: A Process in Memory

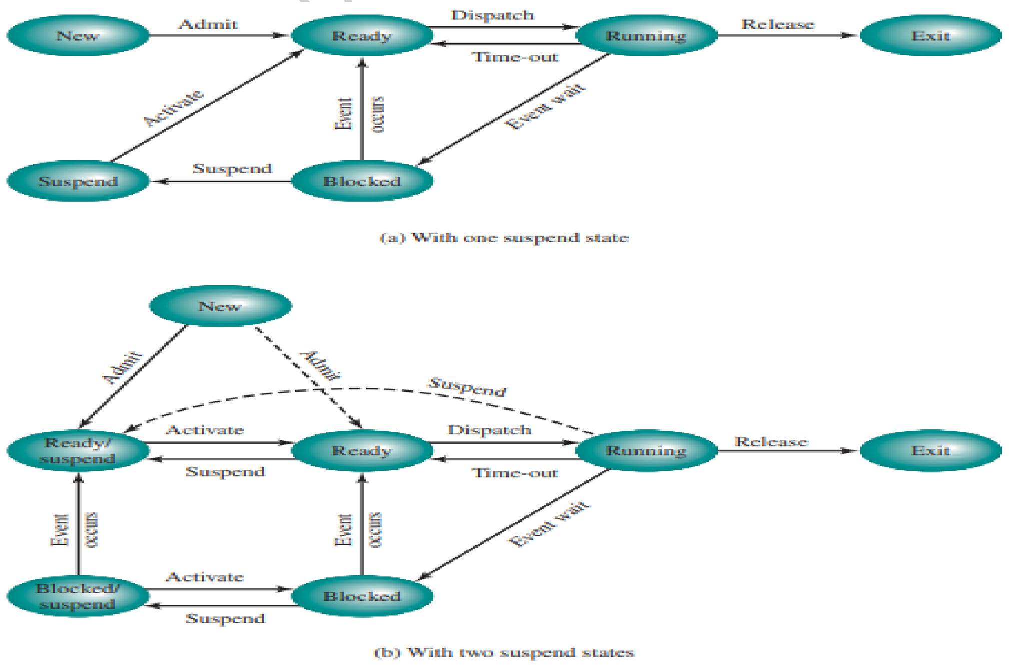
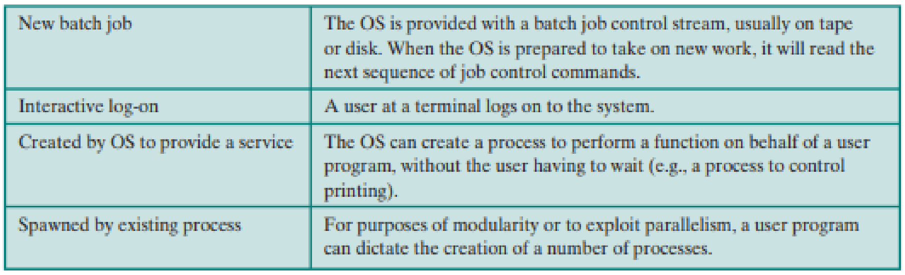


Figure 6: A Process on Disk

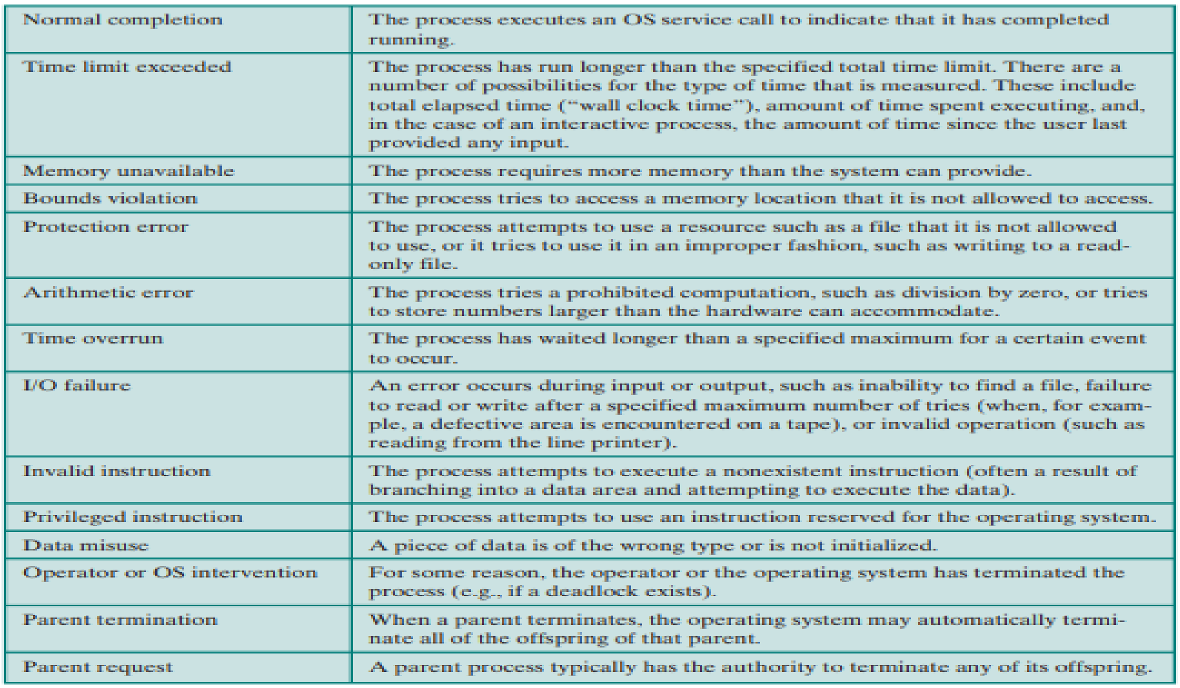
A process that is not in main memory is a suspended process. This process may or may not be waiting for an event and can only be removed from this state by the explicit orders only of the agents that ordered the process suspension. Tables 1 to 3 summarize reasons for process creation, termination, and suspension in OS.

Table 1: Reasons for Process Creation



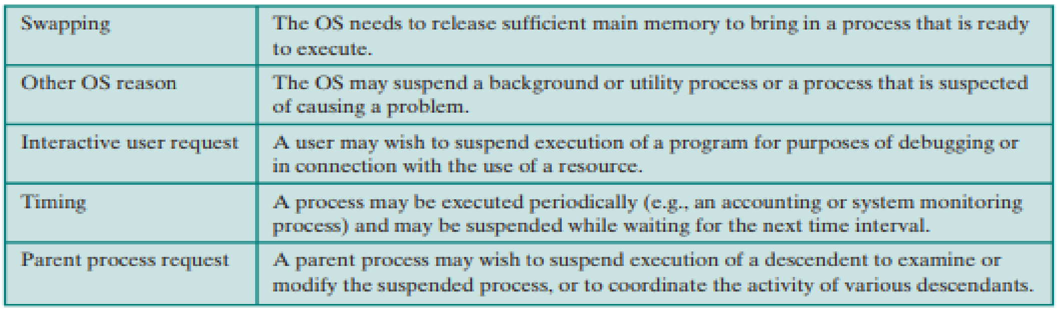
New batch job	The OS is provided with a batch job control stream, usually on tape or disk. When the OS is prepared to take on new work, it will read the next sequence of job control commands.
Interactive log-on	A user at a terminal logs on to the system.
Created by OS to provide a service	The OS can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).
Spawned by existing process	For purposes of modularity or to exploit parallelism, a user program can dictate the creation of a number of processes.

Table 2: Reasons for Process Termination



Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.
Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (e.g., if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

Table 3: Reasons for Process Suspension



Swapping	The OS needs to release sufficient main memory to bring in a process that is ready to execute.
Other OS reason	The OS may suspend a background or utility process or a process that is suspected of causing a problem.
Interactive user request	A user may wish to suspend execution of a program for purposes of debugging or in connection with the use of a resource.
Timing	A process may be executed periodically (e.g., an accounting or system monitoring process) and may be suspended while waiting for the next time interval.
Parent process request	A parent process may wish to suspend execution of a descendent to examine or modify the suspended process, or to coordinate the activity of various descendants.

State is not the only attribute of a process. Other attributes includes:

(i) **Identifier**: A unique identifier associated with this process, to distinguish it from all other processes.

(ii) **Priority**: Priority level relative to other processes.

(iii) **Program counter**: The address of the next instruction in the program to be executed.

(iv) **Memory pointers**: Includes pointers to the program code and data associated with this process, plus any memory blocks shared with other processes.

(v) **Context data**: These are data that are present in registers in the processor while the process is executing.

(vi) **I/O status information**: Includes outstanding I/O requests, I/O devices (e.g., disk drives) assigned to this process, a list of files in use by the process, and so on.

(vii) **Accounting information**: May include the amount of processor time and clock time used, time limits, account numbers, and so on.

This process information is collectively called process/task control block (PCB or TCB). A PCB serves as the repository for all data needed to start, or restart, a process along with some accounting data. However, on system that supports multiple threads, the PCB is expanded to include information for each thread. A thread is a lightweight process, a dispatch able unit of work with a processor context (which includes the program counter and stack pointer) and its own data area for a stack (to enable subroutine branching). A thread executes sequentially, and is interruptible so that the processor can turn to another thread. Several threads of same process can execute concurrently in one or more processors – multithreading. The benefits of multithreading are: responsiveness, resource sharing, economy, and scalability. However, the following challenges are evident: task and data splitting, load balancing, data dependency, and testing and debugging. **N.B**: Amdahl's Law: speedup $\leq \frac{1}{S+\frac{(1-S)}{N}}$; as N >> ∞, speedup >> 2.

Week Four – Day Two

Process Management

The central themes of operating system design are all concerned with the management of processes and threads which are either competing or cooperating:

(i) **Multiprogramming**: The management of multiple processes within a uniprocessor system

(ii) **Multiprocessing** : The management of multiple processes within a multiprocessor

(iii) **Distributed processing**: The management of multiple processes executing on multiple, distributed computer systems. The recent proliferation of clusters is a prime example of this type of system.

Fundamental to all of these areas, and fundamental to OS design, is concurrency. Concurrency encompasses a host of design issues, including communication among processes, sharing of and competing for resources (such as memory, files, and I/O access), synchronization of the activities of multiple processes, and allocation of processor time to processes (CPU scheduling). We shall see that these issues arise not just in multiprocessing and distributed processing environments but even in single-processor multiprogramming systems. Concurrency arises in three different contexts:

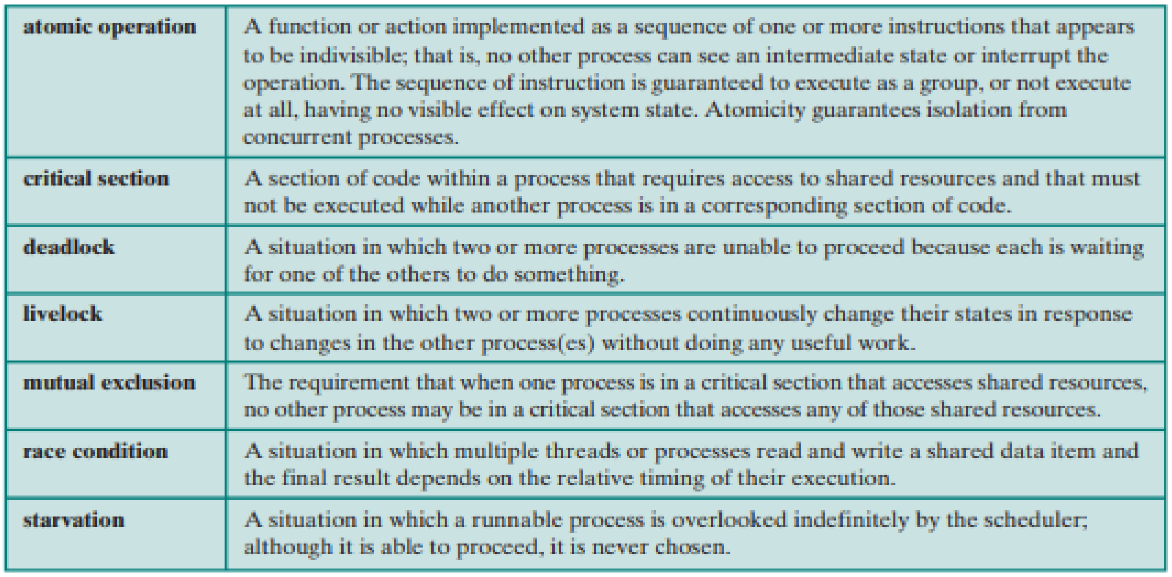
(i) Multiple applications: Multiprogramming was invented to allow processing time to be dynamically shared among a number of active applications.

(ii) Structured applications: As an extension of the principles of modular design and structured programming, some applications can be effectively programmed as a set of concurrent processes.

(iii) Operating system structure: Operating systems are themselves often implemented as a set of processes or threads.

In this course of this lecture, we shall be exposed to the following concepts as highlighted in Table 4.

Table 4: Concepts Associated with Concurrency



atomic operation	A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.
critical section	A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
deadlock	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
livelock	A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.
mutual exclusion	The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
race condition	A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.
starvation	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

It is evident that process scheduling as captured in Figure 7 underpins multiprogramming, multiprocessing and timesharing which constitutes the basis of concurrency in OS.

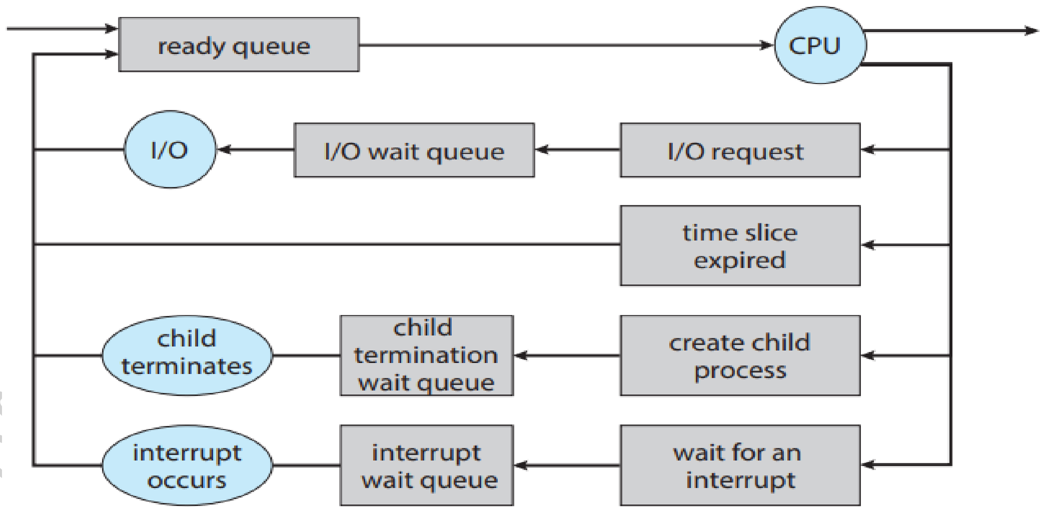


Figure 7: Queueing Diagram Representation of Process Scheduling

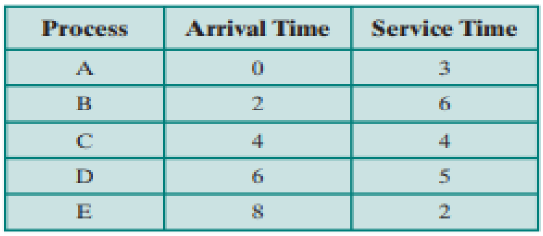
Scheduling can be long-term (determines the degree of multiprogramming: process creation & termination), medium term (context switching), or short-term (dispatcher). Scheduling is meant to optimize CPU utilization, throughput, and turnaround, waiting and response times.

Week Five – Day One

Process Management: Scheduling Algorithms

We have several scheduling algorithms for uniprocessor processes: FCFS, SJFS, SRTFS, RRS, PS, PS-A, PS-RRS, MLQS, & MLFQS. For example problem, see Table 5 and Figure 8.

Table 5: Process Scheduling



Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

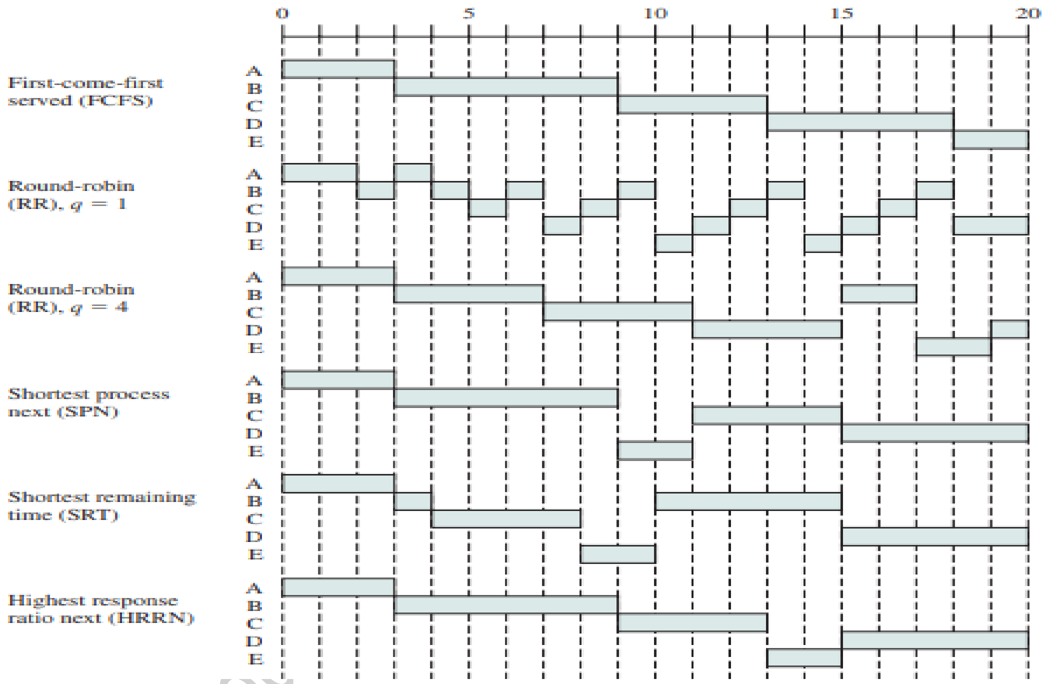

0
5
10
15
20
First-come-first served (FCFS)
Round-robin (RR), $q = 1$
Round-robin (RR), $q = 4$
Shortest process next (SPN)
Shortest remaining time (SRT)
Highest response ratio next (HRRN)
A
B
C
D
E

Figure 8: A Comparison of Some Scheduling Policies

Also see pages 212 to 214 of recommended text. For multiprocessors (Multicore CPUs, Multithreaded cores, NUMA systems, and Heterogeneous multiprocessing) we shall handle them in advanced (MSc) class.

Week Five – Day Two

Process Management: Cooperating Processes Synchronization

A cooperating process is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through shared memory or message passing. Concurrent access to shared data may result in data inconsistency, however. In this class, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

Critical Section Problem: Consider a system consisting of n processes {P0, P1, ..., Pn−1}. Each process has a segment of code, called a critical section, in which the process may be accessing — and updating — data that is shared with at least one other process. The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The critical-section problem is to design a protocol that the processes can use to synchronize their activity so as to cooperatively share data. Each process must request permission to enter its critical section.

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

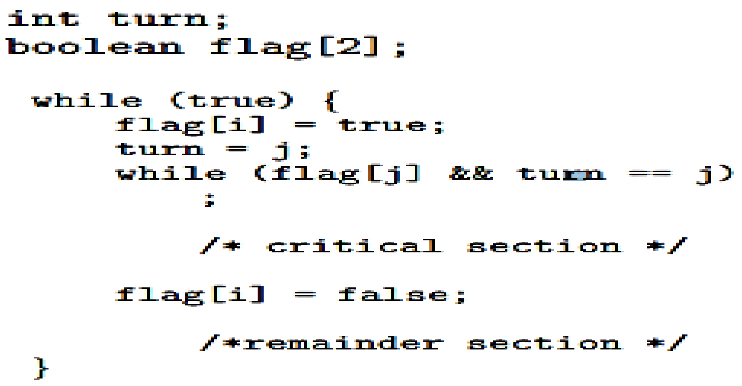
(i) **Mutual exclusion**. If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.

(ii) **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.

(iii) **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.

Possible solutions to this problem include Peterson solution as well as the use of Mutex locks, Semaphores, and Monitors. Each of these solutions has their unique features, strengths and weaknesses discussed as follows:

Peterson's Solution:



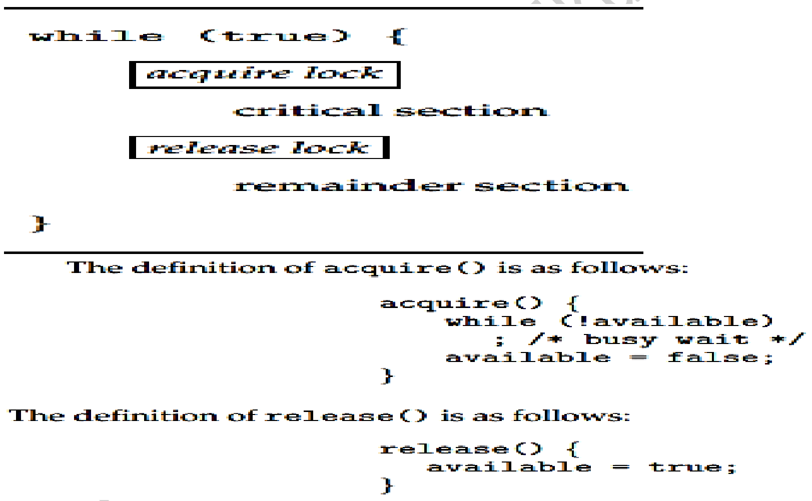
```
int turn;
boolean flag[2];

  while (true) {
       flag[i] = true;
       turn = j;
       while (flag[j] && turn == j)
            ;
            /* critical section */

       flag[i] = false;

            /*remainder section */
  }
```

Figure 9: Peterson's Solution



```
 while (true) {
        acquire lock
              critical section
        release lock
              remainder section
 }
```

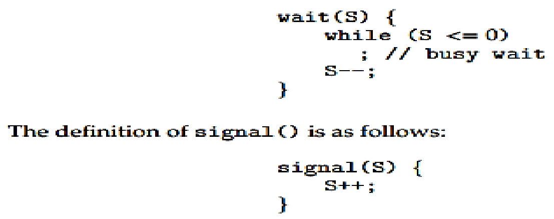
The definition of acquire() is as follows:

```
acquire() {
     while (!available)
          ; /* busy wait */
     available = false;
}
```

The definition of release() is as follows:

```
release() {
   available = true;
}
```

Figure 10: Mutex Lock Solution

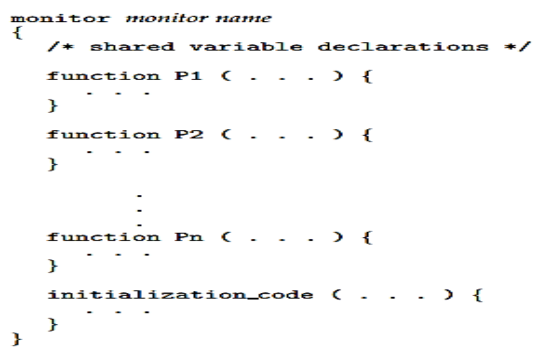


```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

The definition of signal() is as follows:

```
signal(S) {
    S++;
}
```

Figure 11: The Semaphore Solution



```
monitor monitor name
{
   /* shared variable declarations */
   function P1 ( . . . ) {
      . . .
   }
   function P2 ( . . . ) {
      . . .
   }
          .
          .
          .
   function Pn ( . . . ) {
      . . .
   }
   initialization_code ( . . . ) {
      . . .
   }
}
```

Figure 12: Monitor Solution

Monitor Problems

Common problems with monitor (and Semophores) are as follows:

(i) A process might access a resource without first gaining access permission to the resource.

(ii) A process might never release a resource once it has been granted access to the resource.

(iii) A process might attempt to release a resource that it never requested.

(iv) A process might request the same resource twice (without first releasing the resource).

The Concept of Liveness

Liveness refers to a set of properties that a system must satisfy to ensure that processes make progress during their execution life cycle. Failure to satisfy the liveness property results in liveness failure.

Week Six – Day One

Process Management: Competing Processes Synchronization

Irrespective of whether a process is cooperating or competing, it can be exposed to liveness failure – starvation, priority inversion, livelock and deadlock.

Deadlock: Two or more processes or threads are said to be deadlocked iff the following four conditions hold simultaneously in a system:

(i) **Mutual exclusion**. At least one resource must be held in a nonsharable mode; that is, only one process or thread at a time can use the resource. If another process or thread requests that resource, the requesting process or thread must be delayed until the resource has been released.

(ii) **Hold and wait**. A process or thread must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes or threads.

(iii) **No preemption**. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process or thread holding it, after that process or thread has completed its task.

(iv) **Circular wait**. A set {T0, T1, ..., Tn} of waiting threads must exist such that T0 is waiting for a resource held by T1, T1 is waiting for a resource held by T2, ..., Tn−1 is waiting for a resource held by Tn, and Tn is waiting for a resource held by T0.

Concept of Resource Allocation Graph (RAG)

A RAG is directed graph of two set of vertices P & R and two set of edges: request edge of the form $P_i \rightarrow R_j$ and assignment edge of the form $R_j \rightarrow P_i$ where processes or threads (Ps) are denoted by oval and resources are denoted by rectangle.

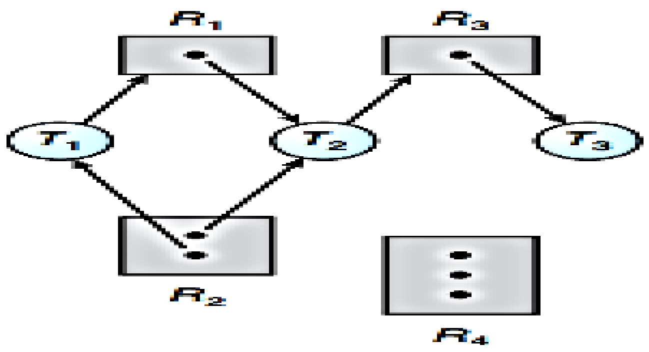


Figure 13: A Resource Allocation Graph

Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no thread in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process or thread involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

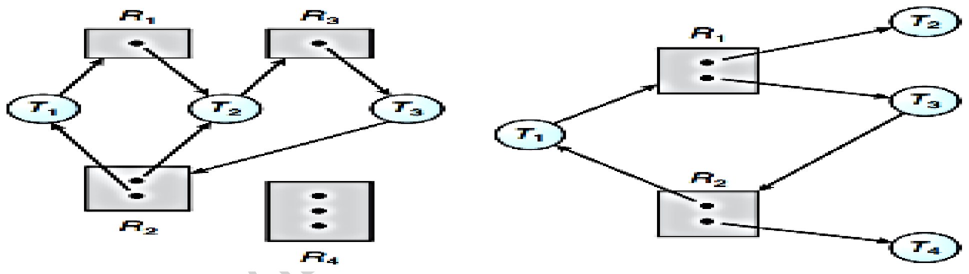


Figure 14: More Examples of RAG

Deadlock solutions: Ignore, prevent, avoid, detect, and recover.

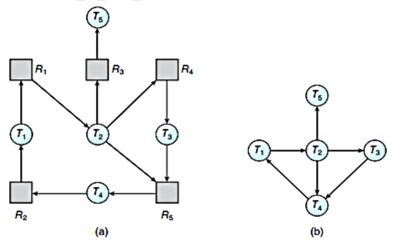


Figure 15: RAG and its equivalent wait-for Graph

Time the only resource mortals cannot feel, touch or control but instead controls mortals beckons defining their living, rise, reign and fall. As usual, Master Time – the greatest of all resources – I remain loyal and I will not struggle as I have never struggled with you. Time says stop, who am I to do otherwise. Loyalty don't question, hence I cannot be loyal to any mortal. I am loyal to only I am who only can control time and assign it to chance.

I wish all the favour of "I am"!