Reading

Operating systems

An **operating system (OS)** is a collection of programs which interfaces between the user or application programs and the computer hardware itself in order to control and manage the operation of the computer. It essentially performs three types of functions: a) booting the computer operation, b) interfacing with users, and c) managing resources, while ensuring the protection of the system from unauthorized access.

Some established operating systems include: **CP/M** (1974) was the first mass-market operating system developed for personal computers. It was designed and implemented by Gary Kildall, who was one of the first people to recognize the necessity for operating systems. **AppleDOS** (1978) and **Apple ProDOS** were two operating systems that were used for all the Apple II family of computers. **DOS** (1981) was the operating system that was used for IBM personal computers. In 1984, Apple released the Macintosh, which was the first to include a graphical user interface-based OS. In 1985, **Microsoft's Windows 1.0** was launched as an interface to operate with DOS (disk operating system). Numerous versions of it have been released by Microsoft, as it seems to be the most widely-used desktop OS today.

UNIX was developed by the Bell Laboratories for use on minicomputers, but it was subsequently modified to run on personal computers. It is a highly portable system as it is written mainly in the C language and thus can be moved across various platforms with no significant changes. **Linux** is an open-source OS that was developed by Finnish programmer Linus Torvalds and was based on UNIX. Linux was originally aimed for computer enthusiasts but nowadays it is widely used by popular commercial platforms, and Linux distributions are dominant in the server and supercomputing sectors. In mobile devices (smartphones and tablets), the dominant OSs are **Apple's iOS** (2007) and **Google's Android** (2008).

Evolution of Operating Systems

The computers of the 1940s and 1950s occupied entire rooms. Program execution required a diligent preparation of equipment, which involved placing punched cards in card readers, and setting switches. The execution of each program, called a *job*, was a separate and isolated procedure.

As soon as the program was executed, all the punched cards had to be removed and retrieved again for the purposes of the next program preparation. When several users needed to share a machine, they would have to sign up special sheets in order to reserve the machine for a specific time slot. The session began with program setup and continued with short periods of program execution. In this context, operating systems were introduced as a means of simplifying program setup and for streamlining the transition between jobs.

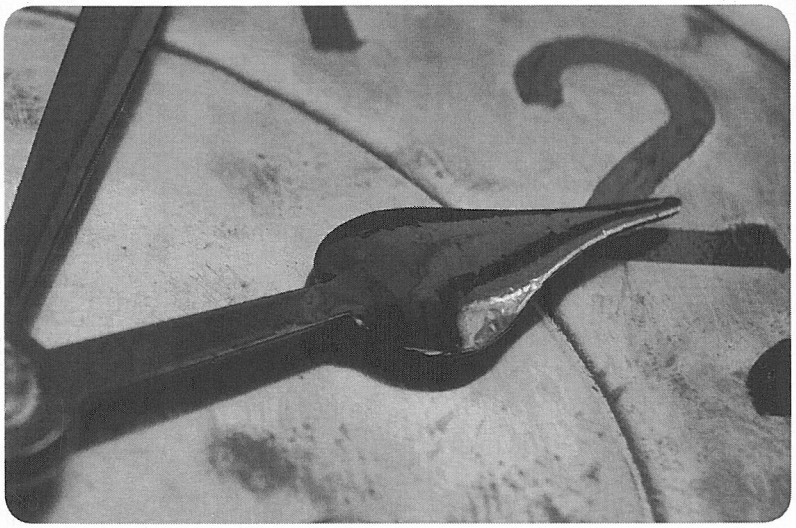
One early development was the hiring of a computer operator to operate the machine. The programmer would submit the program, accompanied by all relevant data and special instructions, and the operator would load these materials into the machine's mass storage and later return with the results. With this approach, called **batch processing**, jobs were collected in a single batch and then executed without any further interference by the user.

In the mid-1960s, **multiprogramming** (also known as **multitasking** or **concurrent programming**) was introduced so as to enhance the operating system's efficiency. The concept concerns the *interleaved* execution of two or more different and independent programs by the same computer. An important strategy emerged known as **timesharing**. This technique involves splitting time into slices and then restricting the execution of a job to only one slice at a time. Resources can be shared between different jobs, and each job receives only a small portion of time on the CPU. Another main concept that evolved during that era was that of the **interrupt**. Essentially, an interrupt is a signal issued by the program indicating

there is an event that needs immediate attention. The processor is interrupting the code that is currently executed. As soon as the requested function has been handled, the interrupt finishes and the processor resumes normal activities.

Operating systems can be broadly classified as **single-user** and **multi-user**. Single-user systems have no facilities to distinguish users, but may allow multiple programs to run. A multi-user OS extends the basic concept of multitasking and can identify processes and resources that belong to multiple users; these users can also interact with the OS at the same time.

Further advances in operating systems resulted in the creation of **distributed systems**. A job that used to be accomplished on one computer could now be shared across various computers. The idea involved one program which was run partially on one computer and partially on another computer, as long as the two computers were connected through a communication network.



Real-time operating systems have become invaluable in real-world scenarios where responsiveness is critical. The real-time operation is given priority, and the operating system can interrupt operations so as to handle real-time enquiries or file updates. **Embedded** systems

are found inside cars, airplanes, thermostats, and other devices we use every day. Their role is to supervise the execution of the programs written for their microprocessors and operate with a limited number of resources. Therefore, they are extremely compact and efficient by design.

Components

As with any complex organization, operating systems need to manage different parts in the computer system while ensuring that the coordination of the various activities is accomplished in an efficient, successful, and secure way. A modern operating system has four main components: *memory manager*, *process manager*, *device manager*, and *file manager*. Additionally, the user interface or shell is a component which is not directly regulated by a specific manager, but rather acts as a command interface responsible for communication outside the operating system.

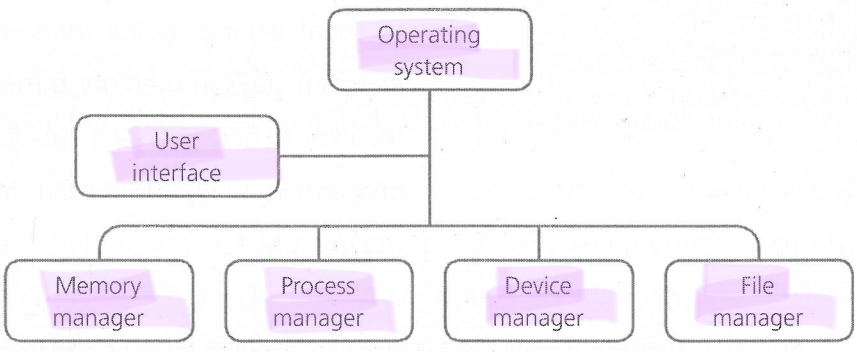


Figure 4.1 Components of operating systems.

Memory Manager

Memory allocation must be managed wisely so that the memory size of modern computers can suffice for more complex programs and greater amounts of data. In the past, most of the memory capacity was dedicated to a single program. In **monoprogramming**, the

whole program was in memory for execution, and only a small part was allotted to the operating system. The program would run until it was replaced by another program. In later years, **multiprogramming** would solve this inefficiency by allowing multiple programs to be in memory at the same time, thus enabling the concurrent execution of all the residing programs.

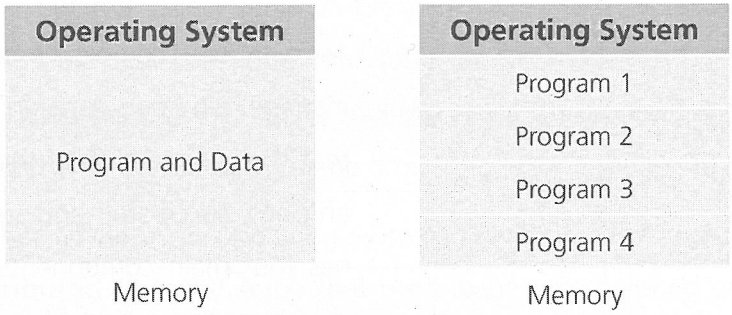


Figure 4.2 Monoprogramming vs. Multiprogramming.

Multiprogramming can be divided into four main techniques. Two of them belong to the *nonswapping* category, meaning that the program stays in memory throughout its execution. The other two techniques belong to the *swapping* category, which means that the program is swapped between memory and disk for the duration of the execution.

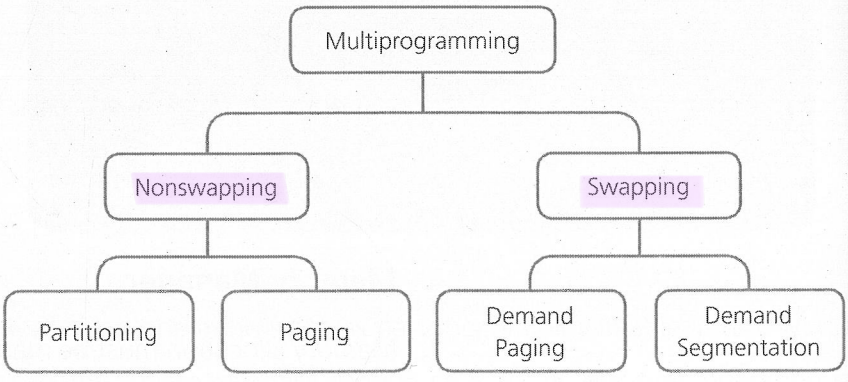


Figure 4.3 Categories of multiprogramming.

The first technique used in multiprogramming is **partitioning**. In this memory allocation scheme, main memory is divided into *partitions* (sections). Each section contains information for a specific job to be executed. The CPU switches between programs. It executes part of the instructions from one program before another input/output operation is requested. Then, the CPU 'unallocates' the specific job, and moves on to allocate a partition to another job, from another program. The cycle of steps is repeated until all of the programs have been served. With this technique, all programs reside in memory and occupy adjacent areas.

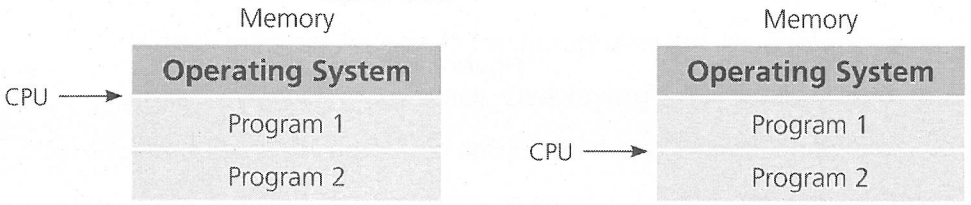


Figure 4.4 Partitioning.

Some of the issues that arise from using this technique are:

- The memory manager needs to specify the size of the partitions beforehand. If partition sizes are small, some programs may not fit into the assigned memory location. If partition sizes are large, *holes* (unused locations) might be found in memory.
- Even if partitions are well-structured at the beginning of the process, holes may occur when new programs take the place the previous ones.
- In the event of many holes, the memory manager can rearrange the partitions so as to remove the holes. However, the creation of new partitions will impose an added overhead on the system.

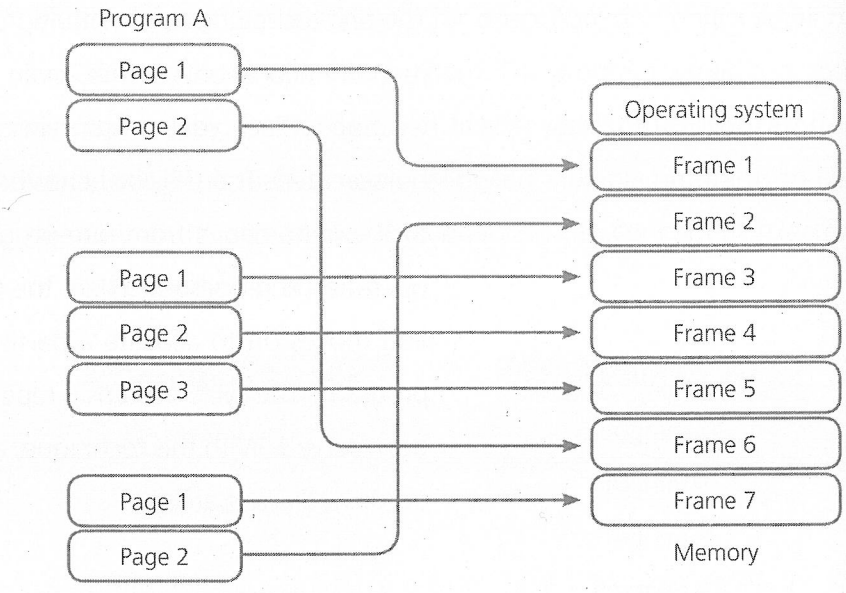


Figure 4.5 Paging.

Paging seeks to improve the partitioning process. In this allocation scheme, memory is divided in fixed-size units called *frames*. Furthermore, the program's virtual address space is divided into same-sized blocks called *pages*. The memory management component is responsible for *mapping* pages to frames. With this technique, the physical memory is organized on a page basis and the address space seems to allocate contiguous locations. Although efficiency is improved with paging, the whole program still needs to be in memory before being executed.

Virtual Memory

In **virtual storage systems**, there are segments of the program which are not immediately required for processing, therefore they are not stored in main memory, but on disk. Through control of the operating system and the memory management software, the portion of the program stored on disk is called into main memory only when required.

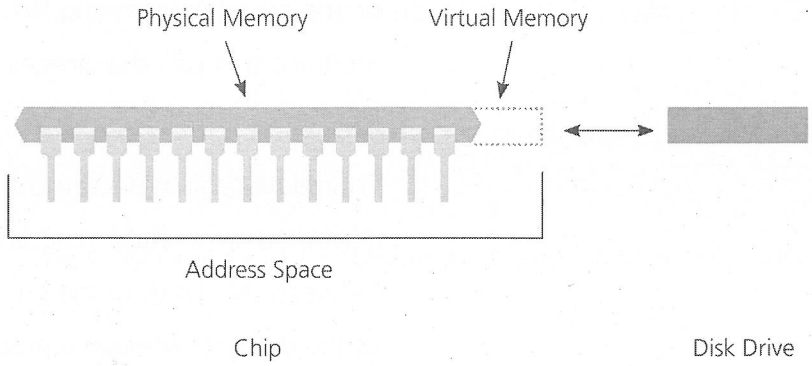


Figure 4.6 Virtual memory.

Virtual memory facilitates multiprogramming by applying a more efficient use of storage space. **Overlaying** is a programming method through which programs are allowed to be larger than the actual main memory. The relevant overlay is loaded into its destination region only when it is needed. As a result, the storage capacity of physical memory is extended, and the illusion of a very large main memory is created. The benefits of virtual memory are essentially used in the paging technique.

Process Manager

Three concepts that are prevalent in modern operating systems are those of a *program*, a *job*, and a *process*. They need to be clearly distinguished from each other since they are all used to refer to a given set of instructions. The **program** involves the set of instructions which were written by a programmer and stored on disk prior to their being selected for execution. As soon as the program is selected for execution, it becomes a **job**. It may stay on disk while waiting to be loaded to memory or it may stay in memory while waiting to be executed by the CPU. When the job finishes running, it goes back to becoming a program that resides on disk and remains out of control

of the relevant operating system. For as long as the program is in memory, it is called a **process**. It is either executing or waiting for CPU time.

The relationship between a program, a job, and a process can be illustrated in a state diagram. The **state diagram** distinguishes between the states of the entities involved in the procedure, as well as the borders between a program, a job, and a process.

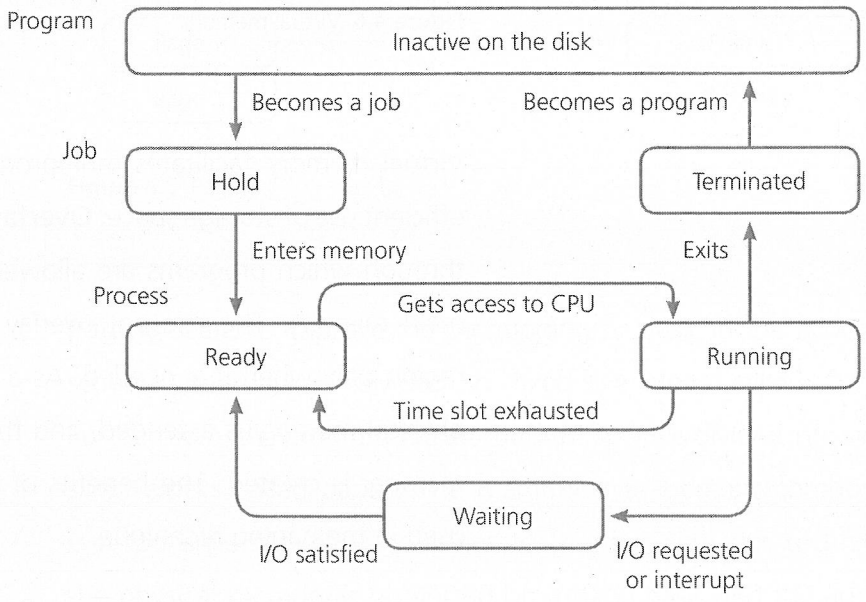


Figure 4.7 State diagram.

A program becomes a job when it is selected by the operating system and brought to the **hold state**. It remains in this state until it can be loaded into memory. As soon as memory, for the whole program or part of it, becomes available, the job is transferred to the **ready state**, and it becomes a process. It stays in memory and in this state until the CPU can execute it. It has now moved to the **running state**. One of three things can happen while the process is in this state:

- The process goes to the **waiting state** and waits until I/O is complete.
- The process has exhausted its allocated time slice and goes directly to the ready state.
- The process goes to the **terminated state** and is no longer a process.

Before reaching the terminated state, the process can move between the ready, running, and waiting states several times. The state diagram is bound to be more complicated if the system uses virtual memory, in which case programs will be swapped in and out of main memory multiple times.

Process management essentially involves the **synchronization** of different processes with different resources. When a process is waiting for another process to release a requested resource, which in turn is waiting for another resource, resulting in a state where no progress can be made, the situation is known as a **deadlock**. Deadlock is a common problem in concurrent systems, where resources have to be shared, and control over them has to be arbitrated. However, in communication systems, deadlocks result from lost or corrupt signals rather than competition for resources.

A deadlock may arise if and only if all of the following conditions take place simultaneously in a system:

- *Mutual exclusion* (only one process can use a resource; at least one resource must be held in a non-shareable mode, so that the processes are deterred from using it)
- *Resource holding* (or, *hold and wait*: a process holds a resource while requesting additional resources held by other resources)
- *No preemption* (the operating system cannot temporarily reallocate a resource; the resource would have to be released by the process holding it)

- *Circular waiting* (or, *set of waiting processes*: all processes and resources involved form a loop; each process is waiting for a resource held by another process, which in turn is waiting for the first process to release the resource)

When you prevent any of the above conditions from happening, you can avoid deadlock.

Conversely, resource **starvation** can take place when a process is perpetually denied necessary resources in order to perform its work. A classic starvation problem was introduced by Edsger W. Dijkstra. Five philosophers are sitting at a round table. Each philosopher needs two chopsticks to eat a bowl of rice. However, one or both chopsticks could be used by a neighbor. A philosopher could 'starve' if two chopsticks are unavailable at the same time. Every multi-tasking system is supposed to employ scheduling of tasks. Special scheduling algorithms have been designed to allocate resources fairly; a high-priority process would need to run before low priority processes.

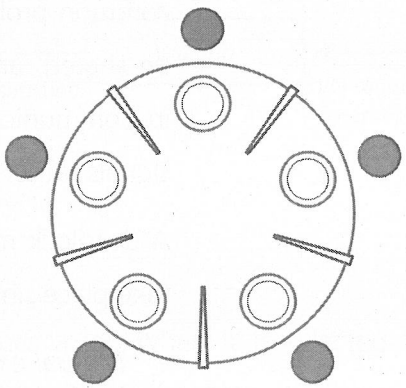


Figure 4.8 Dining philosophers.

Device Manager

The **device manager** handles access to input/output devices. It allows users to monitor the quality of hardware attached to the

computer. Some types of problems that typically arise, may involve devices being disabled either manually or by some error, hardware not working properly, or hardware not being recognized. The device manager has to guarantee the efficient use of input/output devices.

File Manager

The **file manager** controls access to files. Common operations applied to files and folders include creating, opening, renaming, copying, deleting, searching for other files, archiving, preserving backups, and modifying the file attributes and file permissions. File managers used to be called *directory editors*. The prevalent class of file managers is known as *orthodox file managers*. They present the user with a two-panel directory with a command line below.

Some file managers provide network connectivity via known communication protocols, such as FTP (File Transfer Protocol). The user is allowed access to the server's file system like a local file system. However, the need for non-technical website moderators to manage media in their websites has led to the development of *web-based file managers*. Files can be located and edited without the need for some special protocol. The user is then provided with customizable interface.

User Interface

The **user interface** is a program inside the operating system that is responsible for accepting requests from users (processes) and interpreting them for the rest of the operating system. Generally, there are two types of user interfaces: a) *command-line interface (CLI)*, in which the user issues commands directly to the OS. The CLI was the primary means of interaction with most computers until the late 1980s. b) *graphical user interface (GUI)*, in which users interact with the OS through graphical icons and visual indicators.