Operating Systems

Elvis C. Foster

Lecture 01: Introduction to Operating Systems

Welcome to your course in operating systems. This course will carry you through various design concepts of operating systems. Consider it if you will as an advanced data structures course, or an applied software engineering course. You will learn about the design considerations for the most complex software system that most people will ever use — an operating system.

This first lecture explains what an operating system is, and provides you with an overview of issues that will be revisited in more detail as the course progresses. The topics covered here are as follows:

- Definition and Concepts
- Historical Development of Operating Systems
- Requirements of an Operating System
- Multiprogramming, Multitasking, and Multithreading
- Time Sharing Systems and Real-time Systems
- Multiprocessor Systems
- Design and Structure of an Operating System
- Operating System Development Issues
- Summary and Concluding Remarks

1.1 Definitions and Concepts

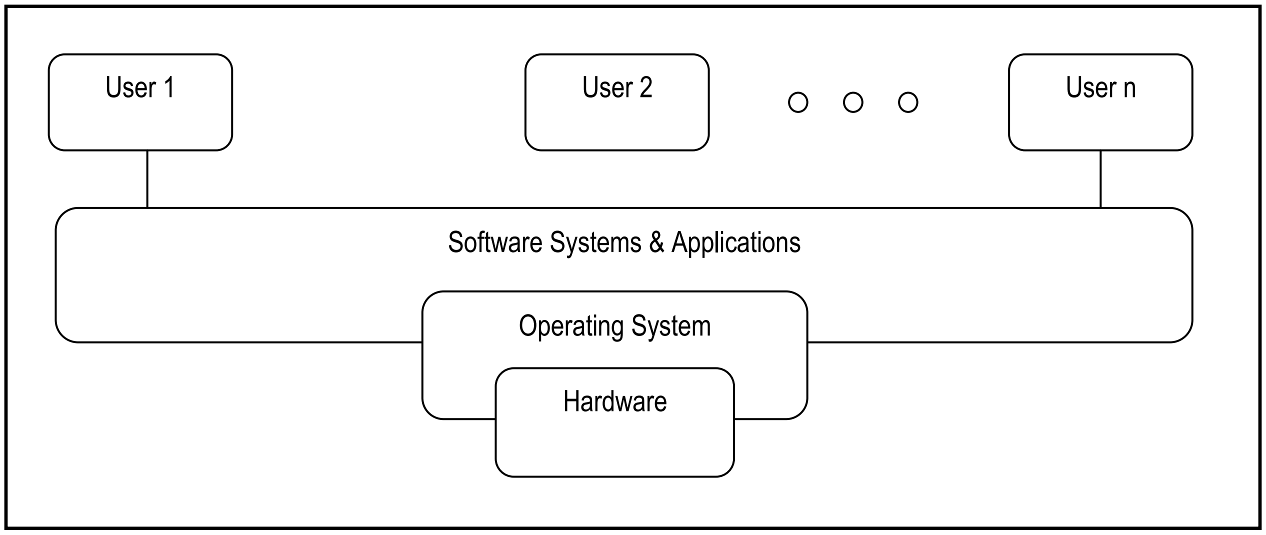
An operating system (OS) is a software system that provides certain desirable and necessary features for users of a computer. These features include the following:

- Helpful and comfortable working environment
- Isolation from hardware
- Facilities for storing, organizing and retrieving data
- Utilities for copying data (files & records)
- Multi - user access
- Efficient use of equipment
- I/O Systems
- Protection, Security and backup
- Connectivity to other operating systems
- Job Scheduling and resource allocation
- User-user Communication

The primary goal of the OS is to make the computer system convenient to use. A secondary goal is the efficient use of the computer hardware. As you will soon see, these two objectives are not always congruent, but often conflict with each other. In many cases, tradeoffs have to be made.

The OS may be viewed as a resource arbitrator, linking the hardware, the application programs and the users. Figure 1.1 illustrates.

Figure 1.1: Abstract View of the Role of the Operating System



1.2 Historical Development of Operating Systems

Early Systems:

Early computers were (physically) very large machines run from the console. The programmer would write a program (of binary codes) and load it into the machine via the console. The program would be run, debugged and corrected from the console.

Single-User Environment: Users had to book times for use of the computer. Possible problems with this approach:

- Machine could be idle if the job was completed ahead of anticipated time.
- User might not finish task in the booked time.

Compiling Systems:

Compilers (for Fortran, Algol, etc.) were developed. The task of programming became easier, but operation of the computer was more complex.

Running a program meant observing the following procedure:

a) Load compiler from tape.
b) Read program from card reader & produce another tape.
c) Compiler produced assembly code.
d) Load & run the assembler (which produced machine code).
e) Execute program.

Problems associated with the compiler approach were:

- Considerable setup time could be involved.
- Each task consisted of several steps. If a mistake was made the whole process had to be restarted.
- Too much human intervention.
- CPU sat idle during setup time.

Special Operators:

Special Operators were hired separate and distinct from programmers. This improved the situation in that:

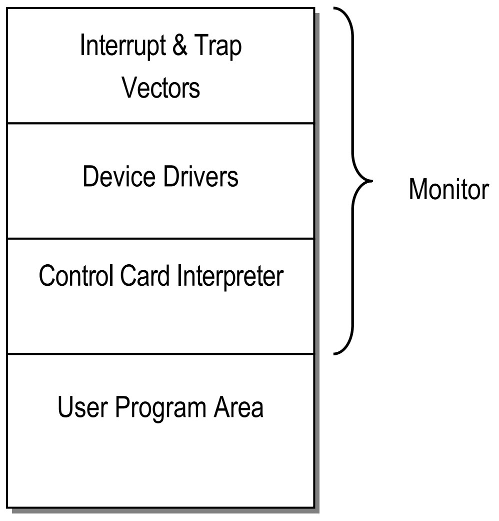
- Computer operators were skilled; hence setup time was significantly reduced.
- Similar jobs were batched and run as a group.

This improved utilization considerably. However, job scheduling was still human directed.

Resident Monitor:

To overcome the problem of idle time, automatic job sequencing was introduced via a program called the resident monitor (because it always resides in memory). Figure 1.2 illustrates:

Figure 1.2: Memory Layout of the Resident Monitor



Contemporary Systems:

In recent times, we have been blessed with the development of more sophisticated operating systems that facilitate multiprogramming, resource scheduling, security etc.

Contemporary systems are *high sophisticated operating systems which facilitate multiprogramming, multitasking, multithreading, resource scheduling, security*, etc.

1.3 Requirements of an Operating System

Among the important requirements of an operating system are the following:

- When a job starts, it should be isolated (protected) from other competing jobs
- When a job finishes, we want to return control to the OS so that another job can start
- When a job completes abnormally, clean up and start a new job
- Facilitate inputs and storage of data
- Facilitate outputs
- Protection of one job from other jobs
- Internal clock to recognize time
- Protection of the OS from users' jobs
- Prevention of unauthorized access
- Facilitation of multi-user access and multitasking
- Efficient resource management
- Efficient memory management
- Utmost ease of usage (user friendliness and technology hiding)
- Functionality — users must be able to effectively use the system
- Provision of desirable utilities of copying, interrupt handling etc.
- Connectivity to other Operating Systems

1.4 Multiprogramming, Multitasking and Multithreading

Multiprogramming is the ability (of the OS) to (apparently) service a number of jobs concurrently.
A job begins when a user signs on to the system.

Multi-programmed operating systems are very sophisticated and imply the need for efficient:
- Memory management;
- CPU scheduling;
- Concurrency control;
- Security and protection mechanisms

These will be discussed in subsequent lectures.

Multitasking is the facility for each user to engage in multiple tasks while signed on to the system in a single session.

Multithreading is a principle operating system designers use to allow the operating system to split a single process into concurrent threads, which may run simultaneously.

1.5 Time Sharing Systems and Real-Time Systems

A time sharing system is opposite to a batch system where jobs are put in batches and processed sequentially.

A time sharing system is an interactive, hands-on system which provides online communication between the user and the system. The user gives instruction directly and receives a direct response.

A real-time system is a system which operates in real time; operating systems are typically real-time systems. Activities must be performed within specific time quanta, otherwise the system fails. With advancement in robotics and mechanization in our era, the need for real time system has significantly increased.

1.6 Multiprocessor Systems

Attempts have been made at creating multiprocessor systems. Historically, this was an objective that had been pursued for several years. More commonly one of the following three approaches is used:
- A number of processors work together, with each having a specific task. Collectively, they are controlled by a main processor.
- A number of processors work together, controlled by events (example interrupts).
- A computer network.

We have seen the implementing of each approach in the current era, This will be further discussed later in the course (lecture 10).

1.7 Design and Structure of an Operating System

An operating system is easily among the most complex software that many people will ever encounter. It must therefore, be carefully and meticulously engineered. A common software engineering approach used in the design and construction of complex software is to break down the problem into smaller, more manageable components. This approach is widely used in operating system design and construction.

1.7.1 Design Goals

The first step in the design of an OS is to define the goals and specifications of the system. Here are some guidelines:

- The supporting hardware is of paramount importance. The wider the range of hardware support, the better.
- Decision must also be taken on the type of system to be designed: batch, time-shared, single-user, multi-user, multitasking, distributed, real-time or general purpose.
- Users want a system that is convenient to use, easy to learn, easy to use, reliable, safe, and fast.
- The system must comprehensive in its coverage of the problem domain, must have self-maintaining features, and easy for end users to conceptualize. It should also be flexible, reliable, efficient, and error-free.
- Mechanics must be separated from policies.

1.7.2 Structure

Older systems such as DOS and Unix essentially evolved (from humble initiation) into widely used systems. They were therefore lacking in their flexibility and ease of maintenance.

Contemporary systems are typically layered, where each layer is built on top of the layer immediately preceding it. Typically, a layer communicates only with two other layers: the layer immediately preceding it and the layer immediately succeeding it.

Figure 1.3a shows the essence of the layered approach, while figure 1.3b provides an example using the Linux operating system (consisting of seven layers).

Figure 1.3a: Layered Architecture

Level	**Main Focus**
Level o	The hardware
Level 1	Instruction Interpretation
Level 2	CPU Scheduling
Level 3	I\O Management
Level 4	Memory Management
Level 5	Device Drivers and Schedulers
Level 6	User Interface

1.7.2 Structure (continued)

Figure 1.3b: Linux Layered Architecture

Level	**Features**
Application	Compilers, DBMS Suites, Desktop Applications, etc.
Linux Shells	Z Shell, Basic Shell, Bourn Shell, Bourn Again Shell, C Shell, TC Shell.
Language Libraries	C, C++, Java, Fortran, etc.
System Call Interface	
Kernel	File Management, Inter-Process Communications (IPC), Memory Management, CPU Scheduler
Device Drivers	Mouse Driver, Printer Driver, CD-ROM Driver, DVD Driver, Hard Disk Driver, etc.
Hardware	Wires, ICChips, CPU, RAM, Monitor, CD-ROM, CD-RW. Etc.

The main advantages of the layered approach are:
- Simplification of the design and construction
- Aid in documentation of the system
- Easier Understanding
- Easier Maintenance

The main drawback with the layered approach is that efficiency is sacrificed. The more layers introduced, the less efficient the system is likely to be. This is so because cross-layer communication is not allowed. An application which needs to store data to disk will therefore have to make a system call, which in turn will trigger other system calls until the desired layer is reached. The problem of reduced efficiency is checked by using fewer layers with more functionality.

1.8 Operating System Development Issues

Old systems were written in assembly language and micro code. Modern systems are written in powerful high level languages. Here are some examples:
- Multics, developed by MIT, was written in PL/1 mainly.
- Unix was written mainly in C.
- OS\2, Windows and other modern operating systems were written in C/C++.

Figure 1.4 provides a comparison of the development alternatives, showing advantages and disadvantages in each case. Note also that with improved processor speeds due to modern technology, these drawbacks are hardly of concern to software engineers as they were historically.

Figure 1.4: Development Alternatives

Development in Assembly or Machine Code	
Advantages	**Disadvantages**
Increased processing speed	Difficult maintainability
Reduces overhead of compilation, interpretation etc.	Inflexibility
Development in High Level Language	
Advantages	**Disadvantages**
Flexibility	Slower speed of processing
Maintainability	Increased overhead (compilation, interpretation etc.)
Faster development; more compact coding	

1.9 Summary and Concluding Remarks

Here is a summary of what has been covered in this lecture:

- An operating system (OS) is a software system that provides certain desirable and necessary feat users for users of a computer.
- Operating system design has been through various stages, starting from bare machines to contemporary systems.
- An OS should fulfill a minimum set of requirements in fulfilling the objectives of convenience and efficiency. These goals are often contradictory.
- The recommended contemporary approach to OS design is the layered approach.

1.11 Recommended Readings

[Bacon & Harris 2003] Bacon, Jean S. & Tim Harris. 2003. *Operating Systems: Concurrent and Distributed Software Design*. Boston, MA: Addison-Wesley. See chapters 1 – 3.

[Davis & Rajkumar 2005] Davis, William S. & T. M. Rajkumar. 2005. *Operating Systems: A Systematic View*, 6th ed. Boston, MA: Addison-Wesley. See chapters 1 – 3.

[Nutt 2004] Nutt, Gary. 2004. *Operating Systems* 3ed. Boston, MA: Addison-Wesley. See chapters 1 – 4.

[Silberschatz, et.al. 2012] Silberschatz, Abraham, Peter B. Galvin & Greg Gagne. 2012. *Operating Systems Concepts*, 9th ed. Hoboken, NJ: John Wiley & Sons. See chapter 1-2.

[Stallings 2005] Stallings, William. 2005. *Operating Systems*, 5th ed. Upper Saddle River, NJ: Prentice Hall. See chapters 1 – 2.

[Tanenbaum & Woodhull 1997] Tanenbaum, Andrew S. & Albert S. Woodhull. 1997. *Operating Systems: Design and Implementation* 2ed. Upper Saddle River, NJ: Prentice Hall. See chapter 1.