CSCI 120 Introduction to Computation
Operating System (draft)

Saad Mneimneh
Visiting Professor
Hunter College of CUNY

1 Introduction

So far, we have studied computation from different perspectives, starting from its abstract nature and the design of algorithms all the way to the physical issues involving the electronic representation (bits) and the computer architecture required to carry out the computation. We identified essential components of such an architecture including the CPU, the control unit, the ALU, the registers, the instruction set, main memory, cache memory, the bus, I/O controllers, I/O ports, input and output devices, and mass storage devices. We have also experienced how to write small programs that perform useful operations using the native instruction set, and we have seen some of the programming constructs that a high level programming language would offer to simplify the task of programming, such as functions, loops, conditionals, recursion, etc...

Now we ask, what makes everything work together? Who is responsible for coordinating the overall operation of a computer? The answer is the **operating system**. The operating system is a software program that is loaded into main memory when the computer first starts. This program provides means by which a user can store and retrieve files, provides the interface by which a user can request the execution of other programs, and provides the environment necessary to execute these programs. Examples of operating systems are Windows, MacOS, Unix, and Linux.

2 History of operating systems

In the 1940s and 1950s, computers were not very flexible or efficient. The machines occupied entire rooms, and the program execution required significant preparation of equipments, including the following:

- mounting and connecting magnetic tapes and mass storage devices
- placing punched cards to input the program
- setting switches depending on the program requirements

For this reason, the execution of each program, also called a job, was handled as an isolated activity. Therefore, computers of that time performed only one job at a time. All the tapes, punched cards, and printed results are then retrieved.

Because multiple users needed to share the machine, sign up sheets were provided so that users can reserve the machine for blocks of time. However, when allocated to a user, the machine was under the total control of that user (no one else can do anything during that time).

In such a single user environment, the operating system started as an effort to simplify program set up and hence ease the transition between users.

2.1 Batch processing

The first step was to separate the users from the equipments and to eliminate the physical transfer of people in and out of the computer room. This was achieved by hiring a computer operator to operate the machine. Users submit their jobs to the operator with the required data and some direction about the program requirements, then return for their results. The operator in turn loads the programs into the machine's mass storage device, where a program called the operating system could read and execute them one at a time. This is called **batch processing**. In batch processing, the jobs residing in mass storage wait for executing in a queue. A queue is a structure in which objects are ordered First In First Out FIFO (compare to stack, LIFO). That is, objects (in this case jobs) are served in the order in which they are received.

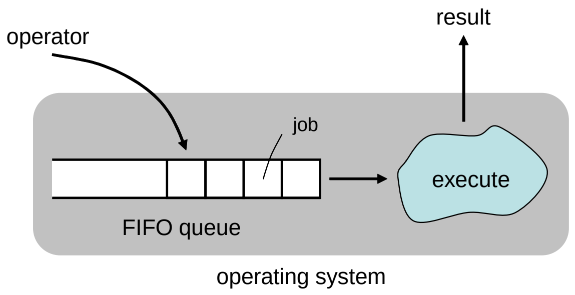

Figure 1: Batch processing

In practice, the queue is not necessarily FIFO. For instance, the queue may be a priority queue. In a priority queue, each job has a priority. Therefore, when a high priority job is submitted by the operator, it bumps few jobs behind it and takes its place in the queue.

Each job was accompanied by some instructions explaining the steps needed to prepare the machine to execute the job. These instructions were written in a special language called Job Control Language JCL. The operating system retrieves the job at the head of the queue, reads the JCL instructions, and prints what needs to be set up on a printer. The operator reads the information from the printer, makes the required set up to the machine, and then instructs the operating system to start executing the job.

Funny observation: To illustrate the operation of the machine, we sometimes use a tiny person that goes back and forth between different components, carrying bits, etc... This was really the case in the early days. A person used to be inside the computer!

2.2 Interactive processing

One major drawback of batch processing (and the presence of an operator) is that users have no interaction with their jobs once submitted to the operator. While this may be acceptable in some applications (e.g. offline computation), it is not acceptable when the user needs to interact with the program during execution (e.g. word processing, computer game, ...). For instance, the user might need to enter some information to guide the computation based on the result obtained so far. Such need lead to the development of operating systems that provide **interactive processing** (as opposed to batch processing).

With interactive processing, the operating system allow users to interact with their programs through remote terminals (also called workstations, which is a term still used today) that are connected to the computer by links (probably the first form of computer networks). One of the problems of interactive processing is that all users seek interactive service in *real-time*; hence the emergence of the term **real-time processing**. If an interactive operating system had been required to serve one user only, then providing timely response, would have been not a problem. But computers in the 1960s were still expensive so each machine had to serve several users. Therefore, providing timely response to multiple users was a challenge. For instance, if one user is exclusively interacting with the machine, all others users have to wait. This problem was solved with the technique known as time sharing.

2.3 Real-time processing and time sharing

In light of the problem mentioned above, it is obvious that the operating system cannot execute one job at a time; otherwise, only the user of that job will receive real-time service. A solution to the problem of real-time processing was to design the operating system to rotate various jobs in and out of execution. The solution, called time-sharing, was to divide time into intervals. Then the operating system restricts the execution of a job to only one interval at a time. At the end of each interval, the current job is interrupted and set aside, and another job resumes (or starts) execution during the next interval. Jobs are rapidly shuffled in this way creating an illusion of several jobs being executed simultaneously. Adequate performance was achieved in those days with 30 users simultaneously, each working on a separate terminal connected to the central computer. The following figure illustrates time-sharing for three jobs.

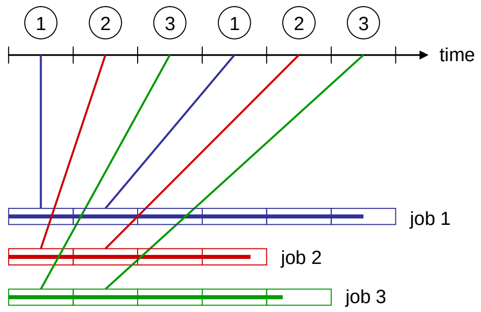

Figure 2: Time-sharing

With multi-user time-sharing operating systems, a computer installation was configured as a large central computer connected to a number of workstations. Commonly used programs were stored on the machine's mass storage device, and the operating system designed to execute these programs as requested by workstations. The role of operator faded.

2.4 Today

Today time-sharing is still used in multi-user systems and provides adequate performance for hundreds and even thousands of users, and also in single-user systems (personal computer) where it is usually called multitasking. The operator is now simply the system administrator whose responsibility is to maintain the machine, install programs, provide accounts to access the machine, etc... Operating systems have grown from simple programs that retrieve and execute programs to complex systems that coordinate time-sharing and maintain programs and data. With multiprocessor machines, operating systems perform another form of multitasking by assigning different task to different processors. The following issues have become major design issues in such operating systems:

- Load balancing: distributing tasks among processors
- scaling: braking a task into smaller independent tasks to ensure high level of parallelism (running subtasks in parallel)
- cache coherence: making sure all processors have valid data in their cache (since multiple processors are updating memory now)

3 Operating system and other software

How does the operating system fit among all other software applications? What is part of the operating system and what is not? To understand this, we attempt to classify software into categories. Although this categorization is not exact, and in fact is sometimes vague, especially with the dynamic subject of software applications and software development, it should provide us with a "big picture".

We begin by dividing software into two broad categories: **application software** and **system software**.

Application software consists of programs that perform tasks particular to the machine utilization, i.e. these programs typically depend on the user. Examples of such programs include word processing, drawing tools, etc... For instance, different users prefer different word processing applications. While some people use Microsoft Office, others use Latex (The software producing this document). Moreover, a machine used by an artist is likely to contain different application software from that used by an electrical engineer. Usually, application software is installed by the user.

In contrast, system software consists of programs that perform tasks common to the computer system in general. Obviously, the operating system is part of the system software. System software is further divided into two categories: **utility software** and the **operating system software** itself.

Utility software consists of programs that perform activities fundamental to the computer system but not included in the operating system itself. These are software units that either extend the operating system with capabilities or customize it. Examples of utility software may include disk format or copy programs, file compression programs, etc... This separation between utility software and the operating systems allows customization more easily than if it were part of the operating system. For instance, one could update the disk copy utility to become more efficient, without having to change the core of the operating system.

The distinction between utility software and application software is vague. For instance, is word processing application or utility? From a user's point of view, the difference is whether it comes with the installation or not. Sometimes an application becomes so fundamental and a standard used by everyone (for instance, like a text editor, e.g. emacs under Unix), that it eventually becomes part of the utility software.

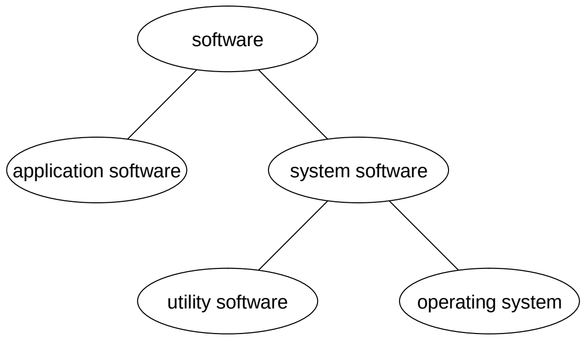

Figure 3: Software

4 The operating system

In this section, we study the main components of an operating system. As a crude description, we divide the operating system into the **shell** and the **kernel**. In order to perform actions requested by the user, the operating system must be able to communicate with the user. The portion of operating system responsible for this communication is called the shell. The rest is the kernel.

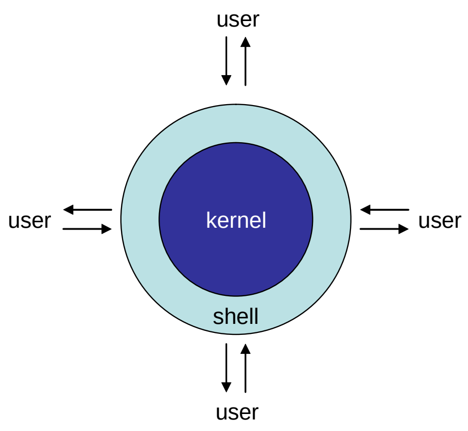

Figure 4: Shell and Kernel

A shell can be text based (interaction with text commands) or a graphical user interface GUI (pronounced Goo-ee). In a GUI, objects to be manipulated such as files and programs are represented pictorially on the screen, e.g. icons. The user issues commands by pointing to these icons using a mouse.

Although an operating system's shell plays an important role in establishing the machine's functionality, it is merely an interface between the user and the heart of the operating system. In fact, most operating systems allow the user to change or choose a shell to customize the type of interaction desired. Example shells: cshell, tcshell, and bourne shell in Unix. Windows itself is a shell, the real operating system is MSDOS (Microsoft Disk Operating System).

GUI shells have an important component known as the window manager, which allocates rectangular blocks of space on the screen, called *windows*, and keeps track of which application is running in a particular window. When an application needs to display something, it notifies the window manager which displays the desired graphics in the window associated with the application. Similarly, when a mouse on a window is clicked, the window manager notifies the application and sends to it the mouse coordinates. A Unix based operating system allows to customize or even choose among a number of window managers. Each has its own look and feel.

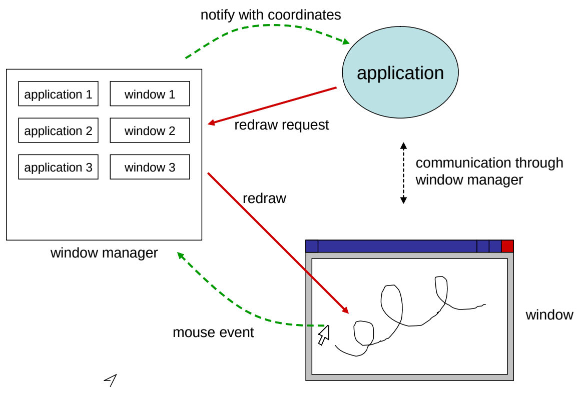

Figure 5: Window manager in GUI shell

The kernel is the heart of the operating system and contains components that perform the very basic functions required by the computer. We have five main components in the kernel:

- File manager
- Device drivers
- Memory manager
- Scheduler
- Dispatcher

4.1 File manager

The file manager coordinates the use of the machine's mass storage devices. It contains records of all files stored in mass storage including:

- where each file is physically located on disk
- which users are allowed to access a file
- which portions on disk are available

For convenience, most file managers allow files to be grouped in directories or folders. This approach makes it easy for users to organize their data. For instance, if you open Windows Explorer (which is the GUI interface of the file manager), you can access c:\windows\system32\notepad.exe. This chain of directories is known as path. Therefore, the path for the file notepad.exe is c:\windows\system32. The file notepad.exe is the executable for the text editor in Windows. The same concept of directories and paths apply in other operating systems. For instance, in Unix, the file manager uses / instead of \.

The file manager is the one who grants access to files requested by other modules. Once approved, the file is opened and a "file descriptor" is stored in memory. Any further operations on the file (reading, writing, etc...) are performed through the file descriptor.

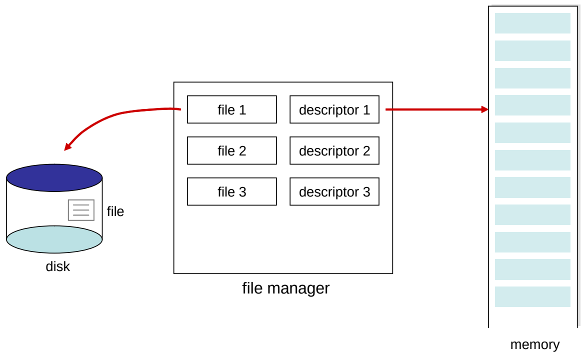

Figure 6: File manager and file descriptors

4.2 Device drivers

Device drivers are software units that communicate with the I/O controllers. Basically, a device driver translates the user requests to the more technical steps needed by the device I/O controller. Although initiated by the user, these requests are typically issued by application software (or other components of the operating system), such as a word processing application requesting to print. Therefore, device drivers are important because they hide the complication of talking directly to the device. For example, the word processing application need only talk to the driver through a certain interface. If the printer is changed, or the driver is updated, the word processing application need not change (otherwise, it would be really inconvenient!).

Every device (such as printer, scanner, etc...) comes with its own device driver. The driver may be part of the operating system; for instance, most HP printers have drivers already installed in Windows.

4.3 Memory manager

The memory manager coordinates the use of the machine's main memory. This task may be trivial in an environment in which a computer is required to perform only one task at a time. In such a case, the program performing the task is put in a predetermined location in memory and the execution starts from there (by setting the program counter accordingly). However, with multi-users and multi-tasking, many programs and blocks of data must reside in memory. The memory manager must:

- find and assign memory space for all these programs and data blocks
- ensure that the actions of each program are restricted to its space (otherwise, a program may override data belonging to another program)
- keep track of available blocks of memory as different activities come and go

The task of the memory manager is even more complicated when the total amount of memory needed by all programs and data exceeds the amount of memory available to the machine. In this case, the memory manager creates illusion of additional memory space by rotating programs and data back and forth between memory and disk. This technique is called **paging**. In paging, programs and data are divided into small chunks called pages that are shuffled back and forth between memory and disk. The additional memory thus obtained is called **virtual memory**. You may have experienced times when Windows becomes very slow and a messages pops up saying "Windows is running low on virtual memory". This is because too many programs are running and the memory manager is swapping a lot of activities between memory and disk, using almost all of the disk space allocated for virtual memory.

4.4 Scheduler and dispatcher

The scheduler and the dispatcher work together to coordinate the execution of the programs. The scheduler determines which programs are to be considered for execution, and the dispatcher controls the allocation of times for these programs. To understand how the scheduler and dispatcher work together, we must look at one of the most fundamental concepts in modern time-sharing operating systems, which is the distinction between the program itself and the activity of executing the program.

- program: this is a static set of instructions to perform a task, and it does not change over time
- execution of program: in a time-sharing system, this is a dynamic activity whose properties change with time. A program can have infinite number of executions in a time-sharing system depending on how it is brought in and out of execution (see Figure 2).

The execution of a program is called a **process**. Therefore, associated with a process is the current status of the activity, called **process state**. A process states includes:

- current position of program being executed (program counter)
- contents of the CPU registers
- contents of the memory locations associated with the program

In other words, a process state is a snapshot of the machine at a particular time of execution.

The main job of the scheduler can be summarized as follows:

1. It maintains record of all processes present in the system, called process table.
2. It adds new processes to this table upon execution requests from the user.
3. It removes completed processes from the table.

The process table is stored in memory. Each process in the system has an entry in this table. Each entry contains: (1) a process ID, (2) the area of memory assigned to this process (obtained from memory manager), (3) the priority of the process, and (4) the status such as Ready for execution or Waiting until some external event occurs.

The dispatcher ensures the scheduled processes (those that are in the process table) are actually executed. This is generally accomplished by time-sharing. Time is divided in intervals of certain length, e.g. 50 milliseconds, and the processes are rotated in and out of execution as explained before. The procedure of changing from one process to another is called **process switch** or **context switch**. Each time the dispatcher allocates a time interval for a process, it initiates a timer circuit that will indicate the end of the interval. When this happens, the timer sends an *interrupt* signal to the CPU.

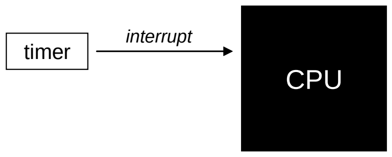

Figure 7: Interrupt

How can we justify this design? In other words, why is it the CPU that should be interrupted? We explore the answer to this question in class, but for short, it is the CPU who is actually running the process. Moreover, the dispatcher, which is another program, is idle (the CPU can run one program at a time). Only the CPU can give back control to the dispatcher.

Now imagine this: what happens if you are reading a book and you are interrupted by someone? You probably save the page number and the location you reached so far on the page, so that you can restart your reading activity later on. You are also likely to finish the current sentence before you actually respond to the interrupt. The CPU does exactly the same thing:

1. CPU receives an interrupt signal
2. CPU completes the current machine cycle
3. CPU saves its position in the current process (program counter)
4. CPU saves contents of registers and vital information
5. CPU begins executing a program called "interrupt handler" which is part of the dispatcher and is stored in a predetermined memory location

Although it can be handled by the dispatcher, typically the CPU is equipped with control instructions to perform the above procedure. The interrupt handler (i.e. the dispatcher) performs the following:

1. Dispatcher allows the scheduler to update priorities, e.g. lower the priority of the current process and raise the priority for others.

2. Dispatcher selects a process with a highest priority among the ready ones in the process table, restarts the timer circuit, and allows the selected process to begin execution (setting the program counter).

5 Booting and BIOS

The operating system provides the infrastructure required by other programs, but we have not considered how the operating system itself gets started! This process is called boot strapping often shortened to booting, which is performed every time the computer is turned on.

Let's think how this can be done. Obviously we want to avoid the possibility of introducing another kind of operating system to start the operating system! We may design the CPU to always start with a predetermined value for the program counter. We can store the operating system program starting at this predetermined location in memory. When the computer is turned on, the CPU starts fetching instructions from that location and the operating system starts execution. However, this idea does not work. The reason is the following: Main memory, RAM, is volatile. When the computer is turned off, everything in RAM (including the operating system) is gone.

Therefore, another idea is to store the operating system program in a portion of memory that is not volatile, e.g. Read Only Memory ROM. However, it would be expensive to store the whole operating system in ROM because this would require a large ROM. Moreover, it becomes hard to update the operating system (need to change the ROM!).

The solution is the following: we store a small program, called the bootstrap in ROM. This is the one that first executes when the computer is turned on. It's task is to direct the CPU to transfer the operating system from a predetermined location on disk (the boot sector) into memory starting at a predetermined location also. Then the bootstrap directs the CPU to execute a jump instruction to that memory location.

ROM also contains a collection of software routines for performing fundamental input/output activities such as receiving information from keyboard, displaying messages on screen, and reading data from disk. These will be used by the bootstrap program to perform I/O activities before the operating system becomes functional in memory. Collectively, they form the Basic Input Output System BIOS. Therefore, BIOS is only a portion of ROM, although BIOS is often used (incorrectly) to refer to the ROM itself.

6 Resource Allocation

An important task of the operating system is to allocate the machine's resources to the different processes. In a broad sense, a resource can be any feature in the machine. As a concrete example, a resource could be a printer. It is important to coordinate the use of the printer among different processes. For instance, if two processes are simultaneously printing, then the printer receives data from both, and the produced document is worthless.

To control access to a printer, the operating system must keep track of whether the printer has been allocated. This can be simply achieved by a single bit, called a flag. If the bit is 0 (flag is clear), then the printer is free. If the bit is 1 (flag is set), then the printer is allocated. Therefore, before printing, every process can execute a small program that is part of the operating system to check the flag and see whether it can grab the printer or not. If the flag is clear, the request to use the printer is granted. If the flag is set, the request is denied. In this case, the process waits (it can repeatedly check the flag).

Although this sounds to be a good solution, unfortunately it does not work! Here's a scenario where this scheme would fail:

1. process 1 starts execution
2. process 1 checks the flag
3. process 1 determines that the flag is clear
4. process 1 is interrupted
5. process 2 start execution
6. process 2 checks the flag
7. process 2 determines that the flag is clear
8. process 2 sets the flag
9. process 2 grabs the printer and start printing
10. process 2 is interrupted
11. process 1 resumes execution (last time it determined that flag is clear)
12. process 1 sets the flag (it is already set by process 2 anyway)
13. process 1 grabs the printer and start printing

A solution to this problem is to disallow interrupt during a flag testing routing. Another solution is to provide an instruction **test-and-set** that reads the value of the flag, notes that value, and sets the flag, all in one machine cycle (so it cannot be interrupted). This is not a trivial instruction (register has to be read and written in one machine cycle). Such instruction is called a semaphore. Most modern instruction sets provide semaphores.

Although semaphores guarantee correctness of resource allocation, one still has to worry about resource allocation in the presence of multiple resources. Here's an example: Assume two resources A and B are required to perform a certain task. Process 1 succeeds in grabbing resource A and is waiting for resource B. Process 2 succeeds in grabbing resource B and is waiting for resource A. Although nothing wrong has occurred in terms of two processes using the same resource simultaneously, but no progress can be made. This is called **deadlock**. Of course if nothing is happening, then nothing can go wrong!

Here's a classical example that illustrate resource allocation and the problem of deadlock, known as the Dining Philosophers.

Philosophers (processes) are seated on a round table, usually thinking. Between each pair of philosophers is a single fork (resource). From time to time, any philosopher might become hungry and attempt to eat. To eat, the philosopher needs exclusive use of the two adjacent forks. After eating, the philosopher puts the two forks back on the table and resumes thinking.

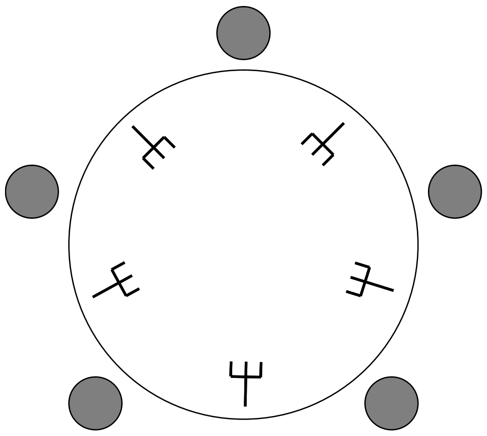
Figure 8: Dining philosophers

If all philosophers become hungry at the same time, and they all grab their left fork first, then they will be all waiting for their right fork to become available. But no more forks will be available because all of the forks have been grabbed. This is a situation of deadlock. No philosopher will be able to eat. Each philosopher waits indefinitely for the right fork. Eventually, they may give up and put back their left fork on the table. But if they do that at the same time, and the all grab their left fork (or right fork) again, the deadlock situation is reached again.

Randomization will help, i.e. each philosopher waits for a random time then attempts to grab a fork. But here's a strategy that is guaranteed to work (i.e. avoid deadlock). The philosophers are numbered (either clockwise or counterclockwise). The even numbered philosophers attempt to grab their left fork first, so they wait until it becomes available and grab it. Once they grab the left fork, they wait for their right fork to become available. The odd numbered philosophers do the opposite. They attempt to grab their right fork first, then wait for their left fork.