Operating Systems – An Introduction

This article provides an introduction to one of the most important items of software, the operating system. We begin with a very brief description of the operating system's role, its history, and then a description of some of the actions it performs.

The operating system is rather paradoxical. An operating system may consume a considerable fraction of the cost of a personal computer if it is Windows. On the other hand, it may cost little or nothing if it is Linux or Android based. Similarly, one author might describe the operating system as being like the perfect government – *it facilitates the running of the state but does not perform actions itself*. On the other hand, some modern operating systems are not like perfect bureaucrats operating in the background, but provide many services to the users. Here, we look at some of these paradoxes.

Role of the Operating System

A simple working definition of an operating system is that it provides an interface between a computer's hardware and software whose function is to control the operation of the whole system. If you consider an orchestra as being composed of instruments, players, and a conductor, the conductor is anagoges to the operating system. A conductor does not make music; he or she enables the musicians and instruments to work together harmoniously to achieve a desired goal. Similarly, the operating system integrates all the various parts of the computer (processor, memory, peripherals, input/output devices) and ensures that they work efficiently with the various programs being executed.

The operating system almost entirely frees the user from having to know anything about the details of the computer. When you with the use a word processor, do you have to go and look for it somewhere on a disk drive, figure out how to get it into the computer's memory (and where to put it)? Do you have to think about how you can route information form the Internet into your computer? No. All these tasks are carried out automatically by the operating system. All you have to do is to click a mouse on an icon, and that it.

But it wasn't always like that.

Boats and Planes and Trains

Some might be tempted to think that operating systems belong only to conventional computers like PCs.

Operating systems exist in embedded systems ranging from cameras to aircraft automatic landing systems to engine controllers in automobiles. Consider the modern digital camera. This has input and output devices (image sensor, buttons and joysticks, touch-sensitive screen), a processor, and storage. Moreover, the processor may be performing many complex tasks such as image processing and even video encoding. Such a complex system requires an operating every bit as much as a personal computer.

History of Operating Systems

The first electronic computers appeared in the 1940 and 1950s. These were large devices constructed using vacuum tubes but with very limited input and output devices. Typically, data was input and output by means of teleprinters (a now obsolete electromechanical devices with a keyboard and a mechanical printer) and many control functions were performed by means of switches. These computers were designed to perform scientific and engineering calculations

Before the invention of operating systems, you had to control the computer directly. In the very early days of the computer, the computer was programmed directly in machine code by using switches on a front panel to input strings of 1s and 0s. Life was made a little easier in the 1950s when data could be entered by means of punched cards or paper tape. However, in those days a program was loaded into the system and run until it was completed before the next program could be run.

The first operating systems developed in the 1950s were called *batch systems* because they allowed a group of jobs (i.e., programs) to be submitted to a computer—normally in the form of large decks of punched cards. Jobs were executed in the order in which they were submitted. The batch operating system allowed programs to be run one after another and control cards to be used to select memory devices and I/O hardware.

The next generation of operating systems employed *multitasking* that permitted computers to run several jobs at the same time. Of course, programs were not executed at exactly the same time; they were divided into slices and interleaved which gave the impression that they were running concurrently. The operating system was responsible for sequencing the execution of these jobs and ensuring that each job got the memory and resources it required. Multitasking operating systems were developed for mainframe computers by companies like IBM in the 1960s.

A close relative of a multitasking operating system is the *timesharing* or *multiuser* system. From the late 1960s to the 1970s (i.e., pre-microprocessor) computers were still very expensive and it was not possible to give every used their own computer. The timesharing operating system was developed to allow several users to access a computer at the same time via on-line terminals. Each user sat at a terminal and was able to interact with the computer in real time. You could edit a program, compile it, and then run it without getting out of your seat. Although entirely commonplace today, that was considered a great step forward.

Another relative of multitasking is the *real-time* operating system. The real-time operating system was originally developed for embedded computers controlling complex systems in industry such a chemical plants and oil refineries. A real-time operating system is designed to ensure that all the various programs running a process can respond to external events (e.g., the detection of a dangerously high-temperature) within a guaranteed time. Today real-time embedded systems are used in safety-critical systems in automobiles.

Operating Systems Curriculum

It is always instructive to look at what others regard as the curriculum for a course in operating systems. The IEEE/ACM joint committee's latest report (2013) lists the following as the key components of an OS course.

- OS principles
- Concurrency
- Scheduling and dispatch
- Memory management
- Security and Protection
- Virtual machines
- Device management
- File systems
- Real time and embedded systems
- Fault tolerance
- OS performance evaluation

The BCS Certificate in IT Computer & Network Technology paper is equivalent to the first year of a UK degree in computing, and its OS syllabus is:

- The human interface
- Multitasking, interrupts, concurrency, scheduling, memory management
- System software: Internet browsers, email systems, security products (e.g. anti virus software)
- System performance and its evaluation: definition, measurement and benchmark

The low-cost of personal computers and the creation of computer networks led to the development of networked operating systems in the mid-1980s. A networked operating system gives you control of your own system and makes other systems on the network appear as an extension of your computer. In a networked system you can access the resources of other computers.

One of the most important developments in operating system technology took place in the early 1980s at the Xerox Palo Alto Research Center, where the *graphical user interface*, GUI, was developed. Users employ a pointing device such as a mouse to select options on a menu displayed on the screen. The graphical user interface was employed by both Apple in their Macintosh range of computers and by Microsoft in their Windows operating system. Today, GUI interfaces have been extended to almost all devices with screens: smartphones, tablets, and cameras.

To a considerable extent, you could say that operating systems have broadened the access to computers. First-generation computers were programmed by people who had to be programmers, systems designers and engineers—they had to understand the computer at the level of electronic circuits. The batch and timesharing operating systems allowed programmers, students and scientists to use the computer without having an intimate knowledge of the computer.

The graphical operating system environments of the 1990s took the expansion of computing one step further by enabling the non-specialist to use the computer as a tool with almost no knowledge of the underlying computer science.

The growth of interconnectivity, first with the Ethernet and then with the Internet, has led to the growth of the *client-server model of computing*. Processes running on one of the terminals are called client processes and are able to make requests to the server. Thus, the operating system is distributed between the client and the server. A client on one host is able to use the resources of a server on another host (the programmer may not even be aware of the location of the server). Today, servers are also known by the type of service they provide; for example, web server or file server.

Driving Forces behind Operating Systems

Mainstream Operating Systems

Between the 1960s and 1980s at the peak of the mainframe and minicomputer era, there were many operating systems. Often, each new manufacturer created a special-purpose operating system for their own range of computers; for example, IBM's OS/360 that was designed in the mid-1960s for IBM's mainframes.

Two events led to a rationalization of operating systems. The Unix operating systems was developed at AT&T's Bell Labs in the USA and adopted by the then popular PDP-11 minicomputer. However, Unix was rapidly adopted by the academic world and has become a de facto standard operating system. In 1991 Linus Torvalds released a free software (open) version of Unix called Linux. Since then, Unix has become a standard operating system for all those who do not use Windows.

Unix is a command-line operating system; that is, it uses a language (rather like a programming language) to perform operations. However, many modern versions of Unix/Linux now use graphical front ends in order to make it easier for non-computer experts to used computers whose underlying operating system is Unix, for example, Apples OS X is built in Unix.

Windows

The most popular language running on PCs is Microsoft's Windows. When IBM introduced its personal computer in the 1980s, Microsoft was invited to write a small test-based operating system to control the PC. MS-DOS was a very basic operating system that provides simple file-handling facilities; that is, the ability to create and delete files and to execute files.

In 1985 Microsoft designed their first GUI interface called Windows. This operating system allows users to have multiple windows open at the same time on the monitor. Some traditionally non-OS functions such as a text editor, calendar, and calculator were also included. This was the beginning of the expansion of operating systems into the area of applications.

By 2013 Microsoft has released version 8.1 of Windows (early versions were numbered by the year, or by a name, and later by version number). Windows 8.1 represents Microsoft's attempt to reconcile a traditional computer operating system with a tablet and smartphone environment that uses a touchscreen as a prime input.

Android

Android is an operating system that was developed by Android Inc., a company that was taken over by Google in 2005. Like Linux, Android is an open source product (even though it was originally developed commercially). Android is a GUI-based operating system built on Unix and is specifically designed for mobile devices with touchscreens such as smartphones and tablets. By 2010, Android had become the world's most popular mobile operating system and by 2013 it had taken an 80% share of the smartphone market. In late 2013 the most most recent version of Android was 4.4, also known as KitKat.

Android is not the only *mobile* operating system. Apple has designed its own iOS (currently version iOS 7 in 2013). Like Android, iOS is built on Unix, but unlike Android iOS is not an open system and it runs only on Apple hardware. Consequently, Apple applications are not generally available on non-Apple hardware.

The Operating System as an Interface

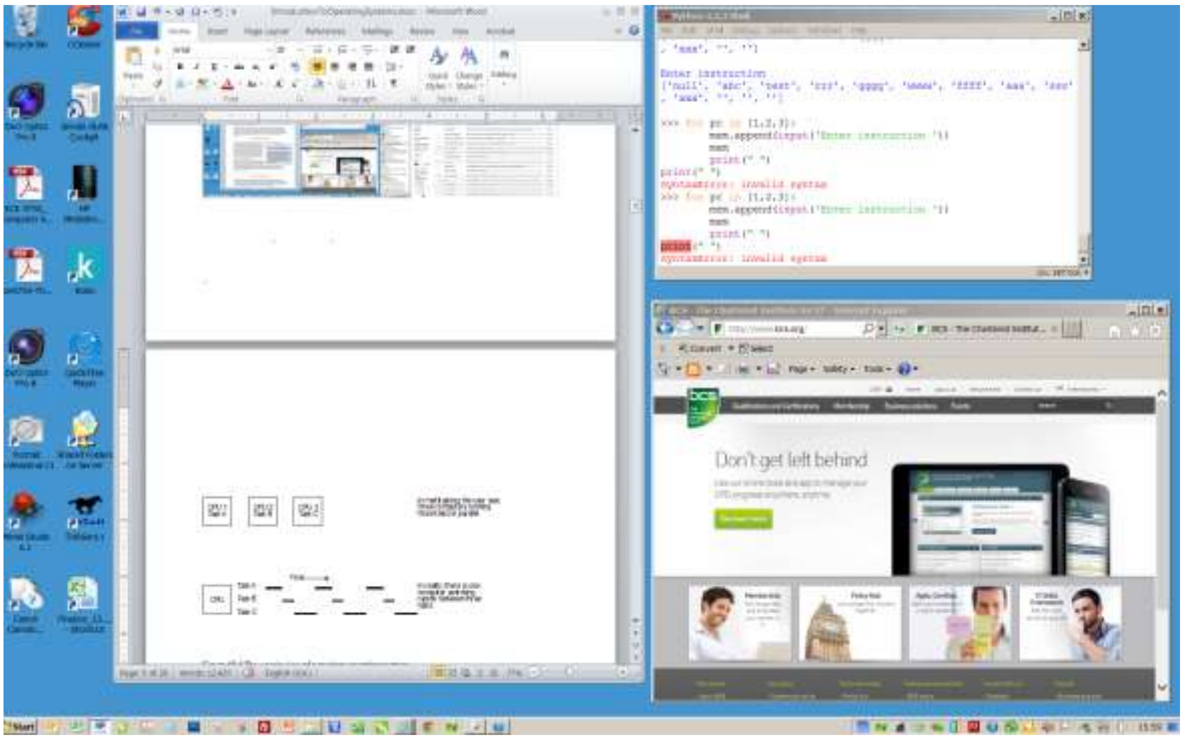
The first role of the operating system is to act as an interface between the user and the system hardware and software. A user wants to tell the computer *what* is to be done and is not concerned with *how* it is done.

The first operating systems used job control languages, JCLs, to control the way in which a computer loaded software, run the software and routed information to I/O devices such as tape drives and printers. In order to use a JCL which is not unlike many other computer languages, you have to be a programmer with a high level of knowledge. The GUI made life easier for the professional and made it possible for non-professionals to use computers. Note that the JCL is not dead. It is still used by some computers in professional applications and many Unix/Linux users still prefer to employ a non-GUI interface. Using a JCL is not a mental aberration; you can do things with a JCL that are not generally possible with a GUI, for example, repetitive and conditional actions.

The GUI represents operations by icons on a screen and lets the user make selections by clicking objects. Moreover, by allowing left and right clicks (or clicking with control keys pressed) you can give the user a range of options. The use of a GUI is largely intuitive and the learning curve is not steep. In other words, you can begin to use a GUI-based system in far less time than one with a JCL.

GUIs continue to evolve as the underlying technology is developed. Low-resolution screens permit only chunky icons which limits the number of choices that the user can make. Today's high-resolution screens allow a far greater range of icons (and text) that can be selected. Touchscreen monitors often increase the apparent resolution by providing multiple screens that can be reached by swiping a finger across the screen. Indeed, multiple-fingered gestures are becoming popular (particularly with tablets) and these allow special operations such as resizing windows and documents.

Figure 1 The GUI



```
, 'aaa', '', '']

Enter instruction
['null', 'abc', 'test', 'rrr', 'gggg', 'wwww', 'ffff', 'aaa', 'rrr'
, 'aaa', '', '', '']

>>> for pc in [1,2,3]:
        mem.append(input('Enter instruction '))
        mem
        print(" ")
print(" ")
SyntaxError: invalid syntax
>>> for pc in [1,2,3]:
        mem.append(input('Enter instruction '))
        mem
        print(" ")
print(" ")
SyntaxError: invalid syntax
```

Two new technologies are beginning to emerge. Direct speech input can be used to interface with digital systems; for example, the iPhone's Siri input uses speech recognition and natural language processing to recognize voice commands and questions. Another input uses a camera to recognize *gestures*. You can regard a gesture as a form of 3D manoeuvre that takes place in space rather than on a screen or tablet. Human sign language is a form of gesture-based communication used to communicate with those having hearing difficulties. Even facial expressions can be used as the basis of human-computer communication.

GUIs are not always the best solution to every computer communications problem. Those with mobility problems or hearing or visual impairments may find it harder to use a GUI than a JCL.

Communication with an operating system using a GUI is a two-way process. An effective system should also be able to provide feedback to the user to deal with error messages, warnings, advice and so on.

Multiprocessing

One of the first operating systems that could be used on a variety of different computers was UNIX, which was designed by Ken Thompson and Dennis Richie at Bell Labs. UNIX was written in C; a systems programming language designed by Richie. Originally intended to run on DEC's primitive PDP-7 minicomputer, UNIX was later rewritten for the popular PDP-11. This proved to be an important move, because, in the 1970's, most university computer science departments used PDP-11s. Bell Labs was able to license UNIX for a nominal fee, and, therefore, UNIX rapidly became the standard operating system in the academic world.

UNIX is a very powerful and flexible, interactive, timesharing operating system that was designed by programmers for programmers. What does that mean? If I said that laws are written for lawyers, I think that a picture might be forming in your mind. UNIX is a user friendly language like a brick is an aerodynamically efficient structure. However, UNIX is probably the most widely used operating system in many universities—this is what Andrew Tannenbaum had to say on the subject

"While this kind of user interface [a user friendly system] may be suitable for novices, it tends to annoy skilled programmers. What they want is a servant, not a nanny."

UNIX is a powerful and popular operating system because it operates in a consistent and elegant way. When a user logs in to a UNIX system, a program called the shell interprets the user's commands. The commands of UNIX take a little getting used to, because they are heavily abbreviated and the abbreviations are not always what you might expect; for example, if you want help you type man which is short for "manual". Similarly, the MS-DOS command type that lists the contents of a text file is given the more obscure name cat (short for catalogue) in UNIX. To be fair, UNIX facilities like the wildcard character "*" make it very easy to carry out powerful operations; for example, the command rm *.tmp allows you to delete any file that has the extension tmp.

Because of UNIX's immense popularity in the academic world, it influenced the thinking of a generation of programmers and systems designers.

One trend we can expect to see in the design of operating systems is the growth of *configuration* and *anticipation* mechanisms. Configuration describes the ability of an interface to be structured to suit its user; for example, each member of a family might tailor the GUI to their own uses in terms of resources and permissions. Anticipation indicates the ability of an operating system to guess what the user might wish to do next and to present selections and alternatives in advance. We already see this in look-ahead typing mechanisms that attempt to reduce the number of keystrokes by guessing what the user is going to type next.

Multitasking

Computers are capable of running several programs simultaneously. You can use a word processor while downloading a movie over the Internet. The ability of a computer to run two or more programs simultaneously is called *multitasking*. Because a CPU has a single program counter that steps through a program instruction by instruction, such multitasking is apparently impossible. However, the human time frame is different to the computer's time frame; a second is a fleeting moment to a human, but to a computer it is 1,000,000,000 nanoseconds or over 10,000,000 instructions. If the operating system switches between programs A, B, and C rapidly (i.e., the programs are executed in the sequence ABCABCABCABC,...), the computer still executes programs sequentially but it appears to a human as if it were executing A, B, and C in parallel. Figure 2 illustrates this concept. Both television and the cinema rely on the same phenomenon; they show a rapid sequence of still images but the viewer perceives a moving image.

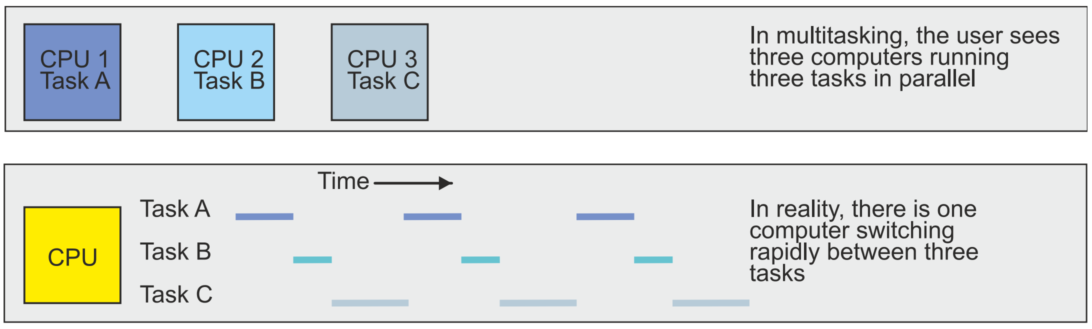
Multiprocessing

This section assumes that the computer has a single processor and that tasks can be split up into slices and run sequentially. Modern computers have multiple processors and tasks can be allocated to individual processors and run in parallel. That is, multitasking divides tasks into slices in *time* whereas multiprocessing divides tasks into slices in *space*.

Multitasking and multiprocessing involve similar operations, the decomposition of tasks into time slices. However, multitasking is controlled by the operating system and used to give the appearance than several processers are running at the same time.

We are going to continue our description of the operating system by looking at its heart—the part that holds everything else together, the kernel.

Figure 2 Multitasking by switching between three tasks


CPU 1
Task A
CPU 2
Task B
CPU 3
Task C
In multitasking, the user sees three computers running three tasks in parallel
Time
CPU
Task A
Task B
Task C
In reality, there is one computer switching rapidly between three tasks

The Kernel

One of the most important components of an operating system is its *kernel* or *nucleus* or *first-level interrupt handler*. This component deals with interrupts from all sources and is responsible for switching tasks (i.e., transferring control from one task to another). The kernel is important because its performance directly determines the efficiency of the operating system. If the operating system switches between tasks rapidly, it is vital that as little time as possible be devoted to the task switching process itself—the more time spent in switching tasks, the less time is available to the running of the tasks themselves.

An essential part of the kernel is the interrupt mechanism that facilitates the task switching process. Although we have already introduced interrupts when we described how the computer performs input/output operations, we provide a summary here.

- An interrupt is a request to the processor for attention
- The interrupt may be a hardware interrupt that is received from an external device
- The interrupt may be a software interrupt that is generated internally by means of a special instruction called a trap
- The interrupt may be a software interrupt that is generated by certain types of error
- The interrupt is dealt with automatically by calling an interrupt handler
- Interrupt handlers form part of the operating system
- An interrupt is transparent to the program that was interrupted—the interrupt handler preserves all working registers

Figure 3 illustrates the simplified structure of the software in a typical general-purpose computer. The software components have been divided into two groups: those that belong to applications programs such as word processors or compilers (i.e., user 1 and user 2), and those that belong to the operating system (in the group that is shaded). The kernel (marked scheduler in figure 3) is responsible for switching between tasks.

Figure 3 Software components in a computer

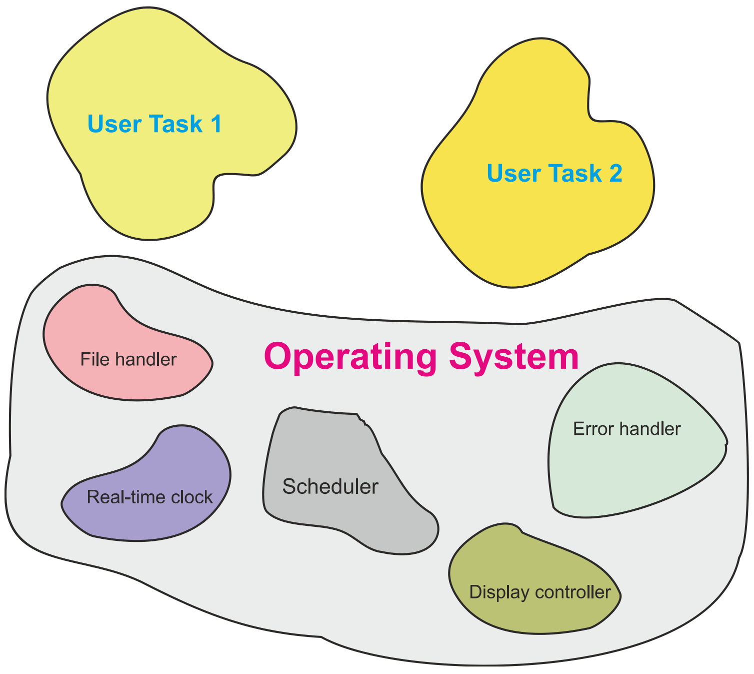
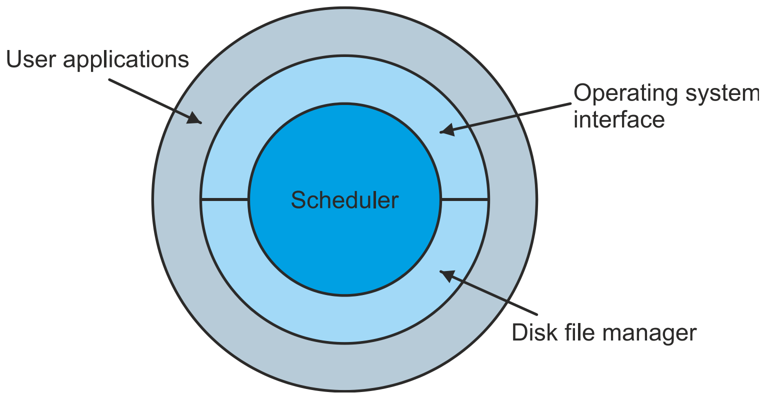


Figure 4 demonstrates how the software components of a computer systems can be regarded as a hierarchical structure with the scheduler at the center surrounded by the other components of the operating system. The uppermost layer consists of the user tasks that employ the services of the operating system. There are two reasons for using a hierarchical model of the operating system. The first is that the outer layers request services provided by the inner layers (this model shares some of the features of the ISO model for open systems interconnection). The second is that the kernel at the heart of the operating system determines the fundamental characteristics of the operating system.

Figure 4 Hierarchical model of an operating system

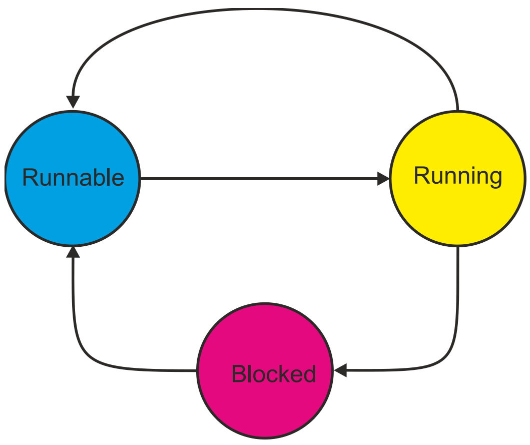


What is a Task?

A task or process is a piece of executable code that can be executed by the processor (i.e., CPU). Each task runs in an environment that is made up of the contents of the processor’s registers, its program counter, and its status register that contains the flag bits. The environment defines the current state of the task and tells the computer where it’s up to in the execution of a task.

At any instant a task can be in one of three states: *running*, *runnable*, or *blocked*. Figure 5 provides a state diagram for a task in a multitasking system. When a task is created, it is in a runnable state—it is waiting its turn for execution. When the scheduler passes control to the task, it is running (i.e., being executed). If the task has to wait for a system resource such as a printer before it can continue, it enters the blocked state. The difference between runnable and blocked is simple—a runnable task can be executed when its turn comes; a blocked task cannot enter the runnable state until the resources it requires become free.

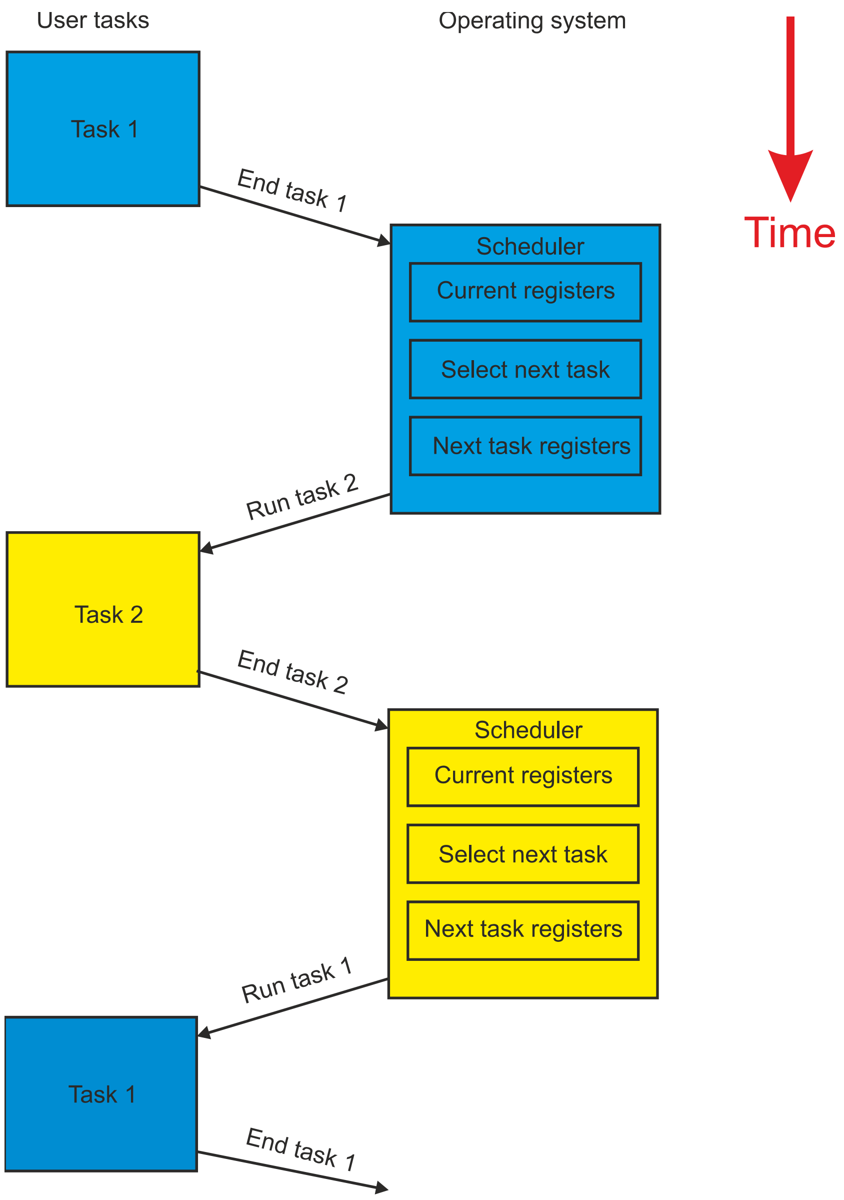
Figure 5 State diagram of a task in a multitasking system



Suppose that the buffer program is currently being executed, and that a timer circuit generates a periodic hardware interrupt to invoke the scheduler every *t* seconds. This operation is called preemptive task switching, because a task is suspended by the active intervention of the scheduler whether or not the task has been completed. A non-preemptive scheduler is called by a task itself when the task has run to completion or requires an operating system service. When the scheduler in the operating system is called, it must ensure that the task that was just halted is left in such a state that it can later be resumed as if nothing had happened. The scheduler then locates the next task to run and passes control to it.

Figure 6 illustrates how task switching is carried out in a system with two tasks. Initially, *Task 1* shown in blue at the top of figure 6, is running. *Task 1* is interrupted by the scheduler in the operating system. The arrow from*Task 1* to the scheduler shows the flow of control from the *Task 1* to the operating system. The scheduler stores the current values of the current task's registers, program counter and status in memory. As we have already said, these registers make up the environment of *Task* 1 and completely define its state.

Figure 6 Switching tasks



The scheduler then loads the processor's registers with the environment of the new task, *Task 2*, causing the new task to be executed. In figure 6 you can see that control is passed back from the scheduler in the operating system to the new task. At a later time the scheduler interrupts *Task 2*, saves its environment, and loads the processor with the registers saved from the *Task 1*. When this happens, the *Task 1* continues from the point at which it was last suspended suspended. We can represent the action of the scheduler by the following pseudocode.

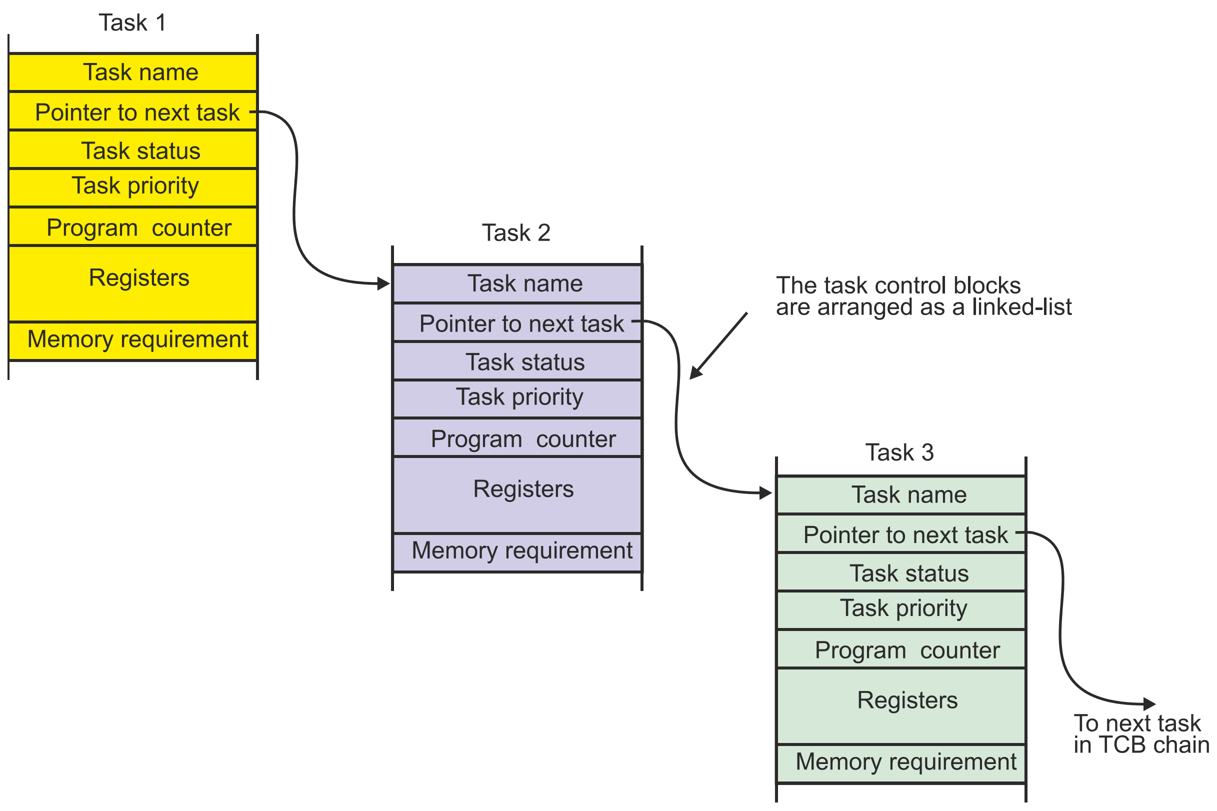
```
Save the environment of the current task
REPEAT
  Find next runnable task
UNTIL a runnable task is located
Load the environment of the new task
Execute the new task
```

The scheduler not only switches tasks, but controls the order in which they are run. Some schedulers allocate the amount of time each task can be run before being interrupted. Some schedulers prioritize tasks and ensure that a high priority task (e.g., a request for service from a disk drive) always runs before tasks with a lower priority.

The way in which an operating system handles task switching depends on the nature of the operating system—some operating systems are specifically designed to switch tasks efficiently. An operating system maintains a table of task descriptors called *task control blocks*, TCBs, that describe the nature of each task and its status.

Figure 7 describes the conceptual structure of a possible task control block (each real operating system has its own particular TCB structure). In addition to the task's environment, a TCB contains a pointer to the next TCB in the chain of TCBs; that is, the TCBs are arranged as a linked list. A new task can be created simply by adding its TCB to the linked list. The TCB contains additional information about the task such as its priority, status (runnable or blocked), and memory requirements.

Figure 7 The task control block



SCHEDULING

Criteria
• CPU utilization – keep the CPU as busy as possible"
• Throughput – # of processes that complete their execution per time unit"
• Turnaround time – amount of time to execute a particular process"
• Waiting time – amount of time a process has been waiting in the ready queue"
• Response time – amount of time it takes from when a request was submitted until the first response is produced, not output (for timesharing environment)"

Task Scheduling

Task scheduling, that is, determining which task is to be executed next is an important part of an operating system and, to some extent, determines its characteristics. There are several ways of selecting the next task to execute. Some of these are:

First-Come, First-Served Scheduling

First-come, first-served scheduling, FCFS, is a simple non-preemptive algorithm. The linked list of task control blocks is arranged as first-in, first-out queue. Each new task created is added to the end (i.e., tail) of the linked list, and each task that is executed is taken from the front (i.e., head) of the list. Because this algorithm is non-preemptive, each task is executed to completion (or until it requests operating system intervention) before the next task is executed.

The FCFS algorithm is easy to implement, and is efficient because little time is lost in searching for the next task to run. An important figure of merit for a multitasking system is the average task waiting time each task has to wait before it is executed. Unfortunately, the FCFS algorithm can lead to a long average waiting time. Moreover, the sequence in which the tasks are received radically affects the average waiting time. Suppose four tasks, t1, t2, t3, and t4, are entered into the task list with requirements of 3, 5, 2, and 10 ms, respectively. The waiting time for each of these tasks is 0, 3, 8, and 10 ms, respectively. The first task has a zero waiting time because it is executed immediately, and then each task is executed after all the tasks before it have been executed. In this example, the average waiting time is (0 + 3 + 8 +10)/4 = 5.25 ms.

If the same four tasks arrived in a different order, say, t4, t1, t3, and t2, the waiting times would be 0, 10, 13, and 14 ms, respectively, corresponding to an average waiting time of (0 + 10 + 13 + 15)/4 = 9.5 ms. Changing the order of the tasks has had a considerable effect on the average waiting time. Note that the total execution time does not change: that is 3 + 5 + 2 + 10 = 20 ms.

Shortest Task Next Scheduling

Another non-preemptive scheduling algorithm is called the shortest task next, STN, algorithm. Have you ever stood in line at a bank when everyone in front of you seems to require an endlessly complex transaction and all you want is to pay a bill? You feel like shouting, "Let me in—It'll only take a few seconds". The shortest task next algorithm behaves just like this and executes the task with the shortest time requirement. This algorithm requires each task to have a parameter that indicates the time required to execute the task. When a new task is first created, it is inserted into the linked list of TCBs at the appropriate point. The shortest task next algorithm discriminates against long tasks, although it ensures that short tasks are executed with very little waiting.

If we apply this algorithm to the same four tasks described previously (t1, t2, t3, t4, with times of 3, 5, 2, and 10 ms), the shortest task next algorithm provides the sequence t3, t1, t2, t4, and an average waiting time of (0, 2, 5, 10)/4 = 4.25 ms. The STN algorithm provides the minimum average waiting time for any sequence of tasks. In practice, this algorithm cannot be applied accurately, because the operating system does not always know how long each task is to take.

Priority Scheduling

Priority scheduling arranges the task in order of their priority (i.e., importance). This scheduling strategy makes sense because some tasks are more important than others—the processor needs to read data from a high-speed disk more urgently than it need to read data from a keyboard. No task may be executed until all tasks with a higher priority have been executed. When a new task is created, the linked list of task control blocks is searched and the priority field of each task read. The new task inserted in the position so that tasks in front of it have a higher equal priority, and tasks behind it have a lower priority.

Although priority scheduling is conceptually reasonable and behaves the way we do, it has a serious limitation. Under certain circumstances, low priority tasks may never be executed because higher priority tasks are always arriving. Such low priority tasks are indefinitely blocked. One way of dealing with this type of blocking is to gradually increase the priority of old, low priority tasks in the queue. Eventually a low priority task will work its way up the queue.

Round Robin Scheduling

A simple method of dealing with the task queue is called the *round robin algorithm*—each task gets a fixed amount of time, called a time quantum or time slice, before it is suspended. Round robin scheduling is preemptive, because a new task is run even though the current task has not yet been completed. The TCBs are arranged as a circular queue, so that the last entry in the queue points to the first. When all tasks have taken a turn, the first task in the queue is run, and so on. The round robin algorithm is called fair because each task gets an equal chance of being executed. Round robin scheduling is useful in time-sharing systems where multiple users are accessing a computer.

The performance of the round robin algorithm depends on the time slice allocated to each task. If the time slice is made very large and each task executes to completion, this algorithm becomes the same as the first-come first-served algorithm. If the time slice is very short, each task in a p-task system appears to have a processor working at 1/p the speed of the actual processor (this calculation neglects the time lost to task switching). With a short time slice, an operating system using the round robin task scheduling algorithm behaves like a time sharing system.

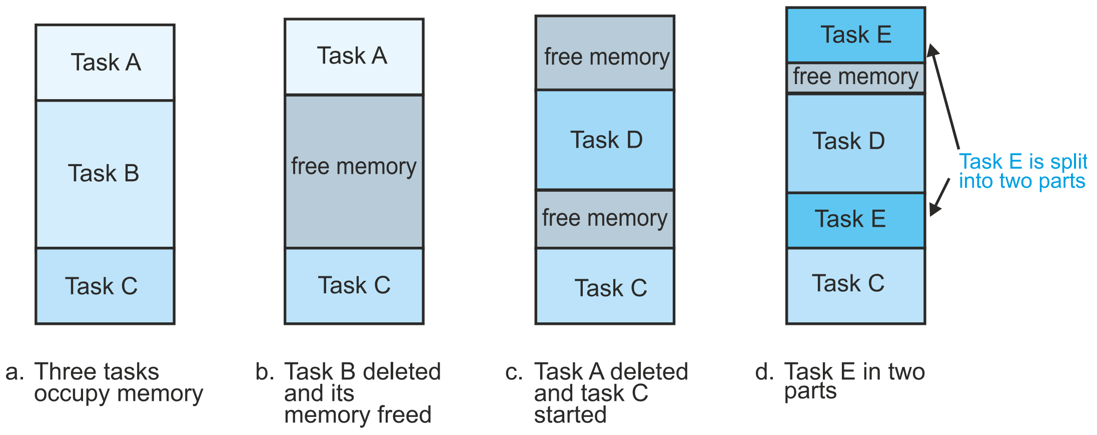
Here we have looked at some of the fundamental algorithms used to schedule tasks in a multitasking system. In practice there are many more algorithms that can be employed, each of which have their own advantages and disadvantages.

Memory Management

If all computers had an infinite amount of random access memory, life would be easy for the operating system designer. When a new program is loaded from disk, you can place it immediately after the last program you fetched into memory. Moreover, with an infinitely large memory you never have to worry about loading programs that are too large for the available memory. In practice, real computers often have too little memory. We now demonstrate how the operating system manages the available memory.

Figure 8a demonstrates a multitasking system in which three programs are initially loaded into memory—task A, task B, and task C. In figure 8b task B has been executed to completion and deleted to leave a hole in the memory. In figure 8c task A had been completed, and a new process, task D, is loaded in part of the free memory left by task B. Finally, in figure 8d a new process, task E, is loaded in memory in two parts because it can't fit in any single free block of memory space.

Figure 8 Memory fragmentation in a multitasking environment


Task A
Task B
Task C
a. Three tasks occupy memory
Task A
free memory
Task C
b. Task B deleted and its memory freed
free memory
Task D
free memory
Task C
c. Task A deleted and task C started
Task E
free memory
Task D
Task E
Task C
Task E is split into two parts
d. Task E in two parts

A multitasking system runs into the memory allocation and memory fragmentation problems described by figure 8. We need a simple but fast means of loading data into memory without having to worry about where it is. In other words we need a system that will map addresses generated by the computer onto the actual location of the data in memory.

Operating systems solve the problem of matching data accessed by the CPU onto its actual address by means of **memory management** that maps the computer's programs onto the available memory space.

Figure 9 describes the arrangement of a memory management unit, MMU. Whenever the CPU generates the address of an operand or an instruction, it places the address on its address bus. This address is called a *logical address*—it's the address that the programmer sees. The MMU translates the logical address into the location or *physical address* of the operand in memory.

The logical address consists of two parts, a word address and a page address. Figure 10 illustrates the relationship between page address and word address for a very simple system with four pages of eight words (i.e., 4 x 8 = 32 locations).

The address from the CPU in figure 10 consists of a 2-bit page address that selects one of 2^2 = 4 pages, and a 3-bit word address that provides an offset (or index) into the currently selected page. A 3-bit offset can access 2^3 = 8 words within a page. If, for example, the CPU generates the address 10101, location 5 on logical page 2 is accessed.

The 3-bit word address from the CPU goes directly to the memory, but the 4-bit page address is sent to the memory management unit. The logical page address from the CPU selects an entry in a table of pages in the MMU as figure 10 demonstrates. Suppose the processor accesses logical page 2, the corresponding page table entry contains the value 3. This value (i.e., 3) corresponds to the physical page address of the location being accessed in memory; that is the MMU has translate logical page 2 into physical page 3. The physical address corresponds to the location of the actual operand in memory. The MMU translates a logical address P on page Q into physical address P on page R.

Figure 9 The memory management unit

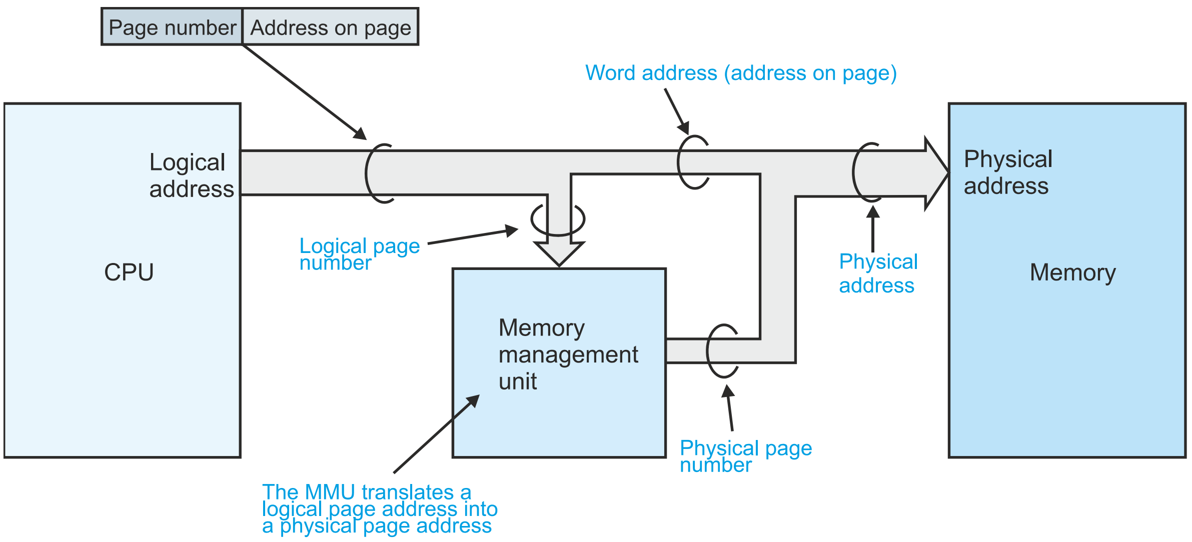
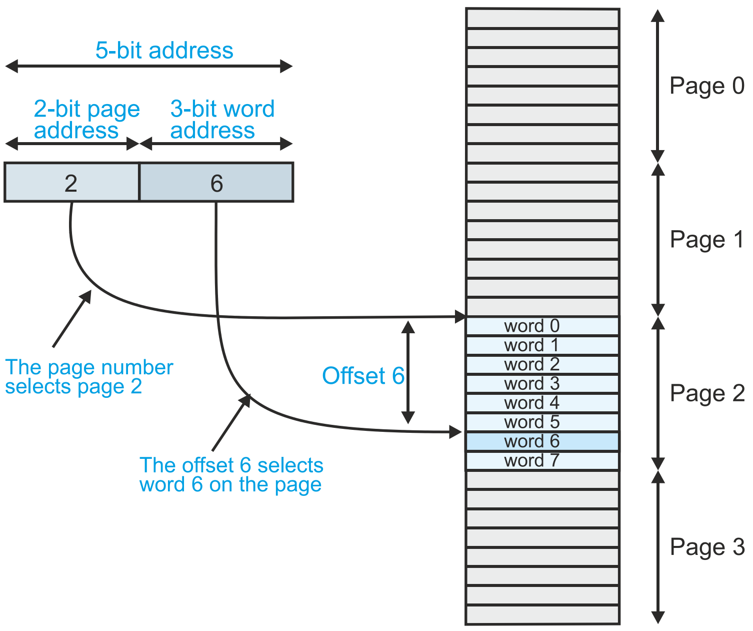


Figure 10 The structure of paged memory

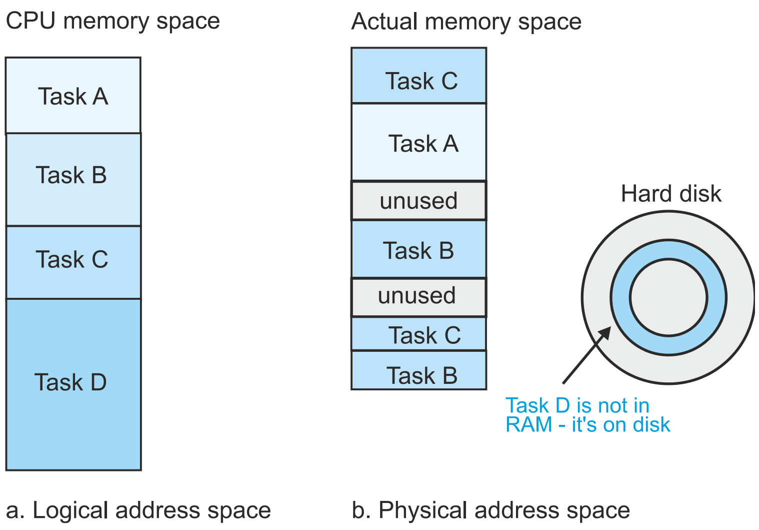


Why should the operating system go to the trouble of taking an address generated by the processor and then using an MMU to convert it into a new address in order to access memory? To answer this question we have to look at how programs are arranged in memory. Figure 11 shows the structure of both logical memory and physical memory at some point during the execution of tasks A, B, C, and D. As far as the processor is concerned, the tasks all occupy single blocks of address space that are located consecutively in logical memory—figure 11a.

If you examine the physical memory, figure 11b, the actual tasks are distributed in real memory in an almost random fashion. Both tasks B and C are split into non-consecutive regions, and there are two regions of physical memory that are currently unallocated. Note also that the logical address space seen by the processor is larger than the physical address space—task D is currently located on the hard disk and is not in the computer's RAM.

A processor's logical address space is composed of all the addresses that the processor can specify. If the processor has a 32-bit address, its logical address space consists of 2^{32} bytes. The physical address space is composed of the actual memory and its size depends on how much memory the computer user can afford. We will soon see how the operating system deals with situations in which the processor wishes to run programs that are larger than the available physical address space. The function of the MMU is to map the addresses generated by the CPU onto the actual memory and to keep track of where data is stored as new tasks are created and old ones removed. With an MMU, the processor doesn't have to worry about where programs and data are actually located.

Figure 11 Logical and physical address space



Consider a system with 4 Kbyte logical and physical pages, and suppose the processor generates the logical address 881234_{16}. This 24-bit address is made up of a 12-bit logical page address 881_{16} and a 12-bit word address 234_{16}. The 12 low-order bits, 234_{16}, define the same relative location within both logical and physical address pages. The logical page address is sent to the MMU which looks up the corresponding physical page address in entry number 881 in the page table. The physical page address found in this location is passed to memory.

Let's look at the way in which the MMU carries out the mapping process. Figure 12 demonstrates how the pages or frames of logical address space are mapped onto the frames of physical address space. The corresponding address mapping table is given by table 1. Notice that logical page 3 and logical page 8 are both mapped onto physical page 6. This situation might arise when two programs share a common resource (e.g., a compiler or an editor). Although each program thinks that it has a unique copy of the resource, both programs access a shared copy of the resource.

Figure 12 Mapping logical address space onto physical address space

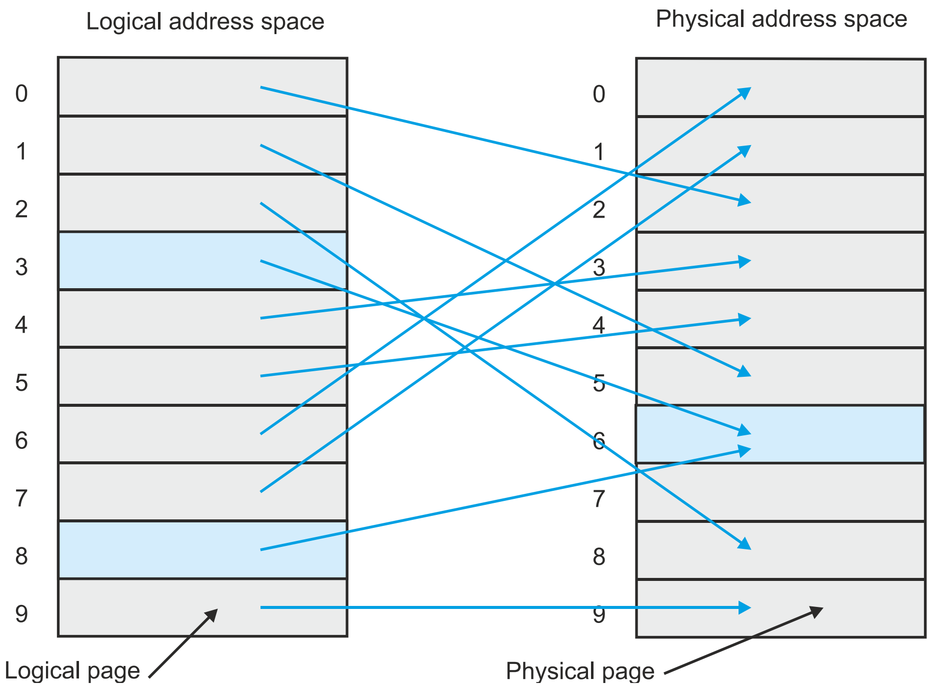


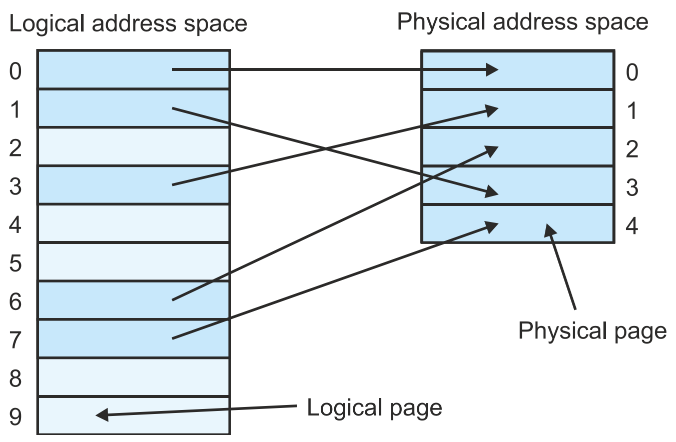
Table 1 Logical to physical address mapping table corresponding to figure 12

Logical page	Physical page
0	2
1	5
2	8
3	6
4	3
5	4
6	0
7	1
8	6
9	9

Virtual memory

We’ve already said that a computer can execute programs larger than its physical memory. The means of accomplishing such an apparently impossible task is called Virtual memory and was first used in the Atlas computer at the University of Manchester, England, in 1960. Figure 10.18 illustrates a system with ten logical address pages but only five physical address pages. Consequently, only 50% of the logical address space can be mapped onto physical address space. Table 2 provides a logical page to physical page mapping table for this situation. Each entry in the logical address page table has two entries: one is the present bit that indicates whether the corresponding page is available in physical memory; the other is the logical page to physical page mapping.

Figure 13 A system with a smaller physical address space than a logical address space



Because it’s impossible to fit all the data required by the processor in main memory at any instant, part of the data must remain on disk. When the processor generates a logical address, the memory management unit reads the mapping table to get the corresponding physical page address. If the page is present, a logical to physical address translation takes place and the operand is accessed. If the logical page is currently not in memory, an address translation cannot take place. In this case, the MMU sends a special type of interrupt to the processor called a page-fault.

Table 2 Logical to physical address mapping table corresponding to figure 10.18

Logical page	Present bit	Physical page
0	1	0
1	1	3
2	0	
3	1	1
4	0	
5	0	
6	1	2
7	1	4
8	0	
9	0	

When the processor detects a page-fault from the MMU, the operating system intervenes and copies a page of memory from the disk to the random access memory. Finally, the operating system updates the page mapping table in the MMU, and reruns the faulted memory access. This arrangement is called virtual memory because the processor appears to have a physical memory as large as its logical address space.

Virtual memory works effectively only if, for most of the time, the data being accessed is in physical memory. Fortunately, accesses to programs and their data are highly clustered. Operating systems designers speak of the 80:20 rule—for 80% of the time the processor accesses only 20% of a program. Note that the principles governing the operation of virtual memory are, essentially, the same as those governing the operation of cache memory.

When a page-fault is detected, the operating system transfers a new page from disk to physical memory, and overwrites a page in physical memory. So, which page gets the chop when a new page is loaded in memory? The most sensible way of selecting an old page for removal is to take the page that is not going to be required in the near future. Unfortunately, this algorithm is impossible to implement.

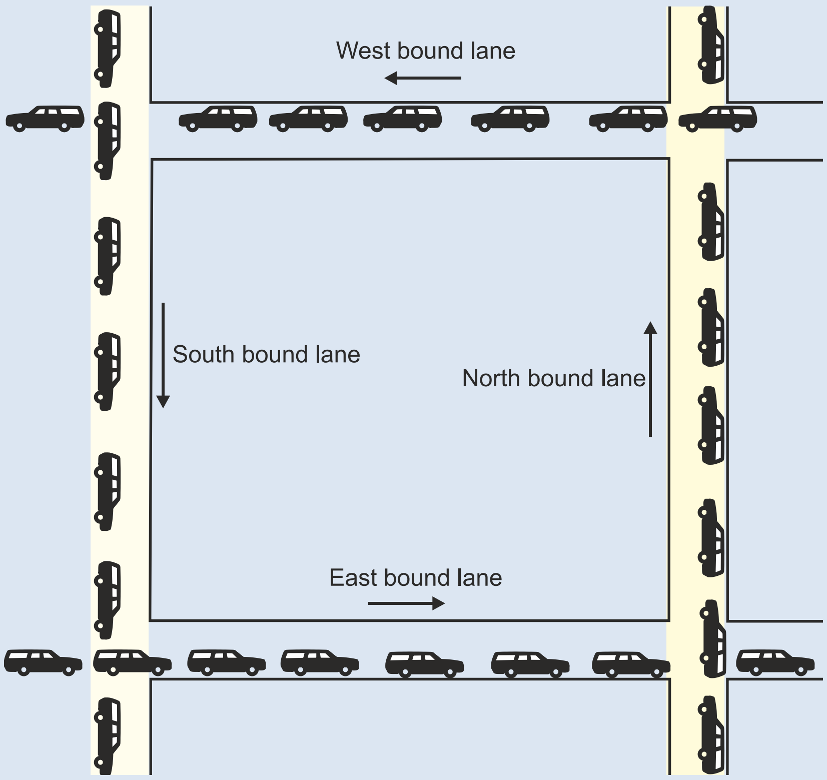
A simple page replacement algorithm is called the not-recently-used algorithm, NRU. When a page is created, it is marked with a time bit. The time bit is changed periodically (e.g., every 50 ms), and therefore, some pages are marked with a 1 and some with a 0. Suppose that the current time bit is 1, and a new page is marked with a 1. If a page is to be removed, the operating system selects a page at random that has a time bit set to 0. In this way, you ensure that the page you are overwriting was created in a previous time slot. The NRU algorithm is not optimum, but it is very easy to implement.

When an old page is replaced by a new page, any data in the old page frame that has been modified since it was created must be written back to disk. A typical virtual memory system clears a dirty bit in the page table when the page is first created. Whenever the processor performs a write operation to an operand on this page, the dirty bit is set. When this page is swapped out (i.e., overwritten by a new page), the operating system looks at its dirty bit. If this bit is clear, nothing need be done; if it is set, the page must be copied to disk.

Virtual memory allows the programmer to write programs without having to know anything about the characteristics or real memory and where the program is to be located. We are now going to look at another of life’s nasty realities that the operating system has to deal with—the computer equivalent of the gridlock.

Operating Systems and Deadlock

In a multitasking system operating system several tasks may be running concurrently. These tasks require various resources during the course of their execution; for example the CPU, the disk, access to other files, the printer, the mouse, the display, memory, and so on. One of the problems an operating system has to contend with is called *deadlock* that occurs when the operating system is blocked and cannot continue. A type of deadlock that is found in everyday big city life is the *gridlock* illustrated below.



The roads are full and none of the lanes can move because each lane blocks its neighbor. The west bound lane is blocked by traffic in the south bound lane. But traffic in the south bound lane is blocked by traffic in the east bound lane. Traffic in the east bound lane is blocked by traffic in the north bound lane. Traffic in the north bound lane is blocked by traffic in the west bound lane, and so on.

Consider a situation in which process A in a computer has all the resources it needs to run, apart from the disk drive, which has been assigned to process B. Process A can run as soon as process B has used the disk drive and freed it. Process B has all the resources it needs to run, apart from the printer, which is currently assigned to process A. As soon as process A has released the printer, process B can run. This is an example of a deadlock because both processes have some of the resources they need, but neither process can run because the other has a resource it needs.

An operating system can deal with deadlock in several ways. The operating system can apply deadlock detection and periodically examine the way in which resources are currently allocated to processes. If a deadlock situation is detected, the operating system must take back resources that have been assigned to processes and then re-assign them. In terms of the gridlock analogy avobe, the operating system is acting as a traffic cop that intervene and sorts out problems. This strategy can be expensive, because processing time is lost each time the operating system goes looking for deadlock. Moreover, it is not a completely effective strategy, because deadlocked processes are not dealt with until after they have occurred.

An alternative approach is to employ deadlock prevention and ensure that processes never become deadlocked. Essentially, the way in which resources are allocated to processes is designed to prevent deadlock. For example, if a process is required to specify all the resources it needs before it runs, the operating system can determine whether a deadlock will occur.

If the operating system permits a process to run only when all the resources it requires are free, deadlock cannot occur. Unfortunately, this strategy is sometimes very inefficient because resources may lie idle for a long time. Another solution is to force blocked processors (i.e., processes waiting for resources) to release resources in favor of a currently running process.

- Under what circumstances is an operating system unnecessary?
- Why are operating systems with a graphical user interface proving so popular?
- Why do you think that some users don’t like graphical user interfaces?
- A floppy disk can hold 1.4Mbytes of data and a punched card can hold 80 characters (bytes). Assume that the average punched card contains 40 characters and weighs 1 gm. What is the total weight of punched cards that can be carried on a floppy disk?
- A multitasking system has 10 tasks. The overhead in switching between two tasks is 200µs. Suppose you wish to implement a time sharing system using the round robin scheduling algorithm and switch tasks as rapidly as possible. If the maximum overhead you are willing to allocate to task switching is 20% of the available processor time, what is the shortest time slice you can allow a process?
- Six tasks, t1, t2, t3, t4, t5, and t6, have durations of 10, 20, 1, 5, 5, 9ms, respectively. What is the average waiting time if a first-come first-served scheduling algorithm is used?
- For the same data as question 6, what would the average waiting time be if the smallest task first algorithm were used?
- Write a program to emulate the task scheduling kernel of an operating system. To do this you will have to design your own task control block, and select a suitable task scheduling algorithm.
- What is the difference between a logical address and a physical address?
- Under what circumstances can a system’s logical address space be larger than the system’s physical address space? If you designed a system whose physical address space was larger than its logical address space, what do you think the consequences would be?
- What is a hierarchical file structure and why is it so important? Can you think of any other way of organizing a file structure?
- Tannenbaum prefers operating systems for programmers to operating systems for non-computer specialists. To what extent do you think that his feelings are justified? Suggest ways in which an operating system could be constructed to get the best of both worlds (i.e., a level of terseness that isn’t an insult to a professional programmer and a degree of user friendliness that enables even users with little or no training to cope)?
- The PATH statement that can be used in an MS-DOS AUTOEXEC.BAT file tells the operating system which pathways to search for files whose locations are not specified fully. What are the advantages and disadvantages of the PATH statement?
- Some professional programmers and computer scientists regard UNIX as a good operating system and MS-DOS as a bad operating system? Why do you think that they have formed this view? If MS-DOS is less than optimum, what factors do you think contributed?
- Design a better, more flexible, and more powerful language than that used by the MS-DOS interface. For example, consider way of allowing more than one instruction to be written on the same line.
- How do you think that the Windows environment could be improved to overcome the criticisms of those who prefer traditional command line operating systems like MS-DOS and UNIX.