CHAPTER 1- Introduction

What is an operating system?

Computers are equipped with a layer of software called the operating system, whose job is to provide user programs with a better, simpler, cleaner, model of the computer and to handle managing all the resources. Operating systems perform two essentially unrelated functions: providing application programmers (and application programs, naturally) a clean abstract set of resources instead of the messy hardware ones and managing these hardware resources.

An **Operating System (OS)** is a software that acts as an interface between computer hardware components and the user. Every computer system must have at least one operating system to run other programs. Applications like Browsers, MS Office, Notepad Games, etc., need some environment to run and perform its tasks.

The OS helps you to communicate with the computer without knowing how to speak the computer’s language. It is not possible for the user to use any computer or mobile device without having an operating system.

OS Hardware:

1. Processors: The ‘‘brain’’ of the computer is the CPU. It fetches instructions from memory and executes them. The basic cycle of every CPU is to fetch the first instruction from memory, decode it to determine its type and operands, execute it, and then fetch, decode, and execute subsequent instructions. The cycle is repeated until the program finishes. In this way, programs are carried out.
2. Memory: Ideally, a memory should be extremely fast (faster than executing an instruction so that the CPU is not held up by the memory), abundantly large, and dirt cheap. No current technology satisfies all of these goals, so a different approach is taken. The memory system is constructed as a hierarchy of layers. The top layers have higher speed, smaller capacity, and greater cost per bit than the lower ones, often by factors of a billion or more.
3. Disk: Disk storage is two orders of magnitude cheaper than RAM per bit and often two orders of magnitude larger as well. The only problem is that the time to randomly access data on it is close to three orders of magnitude slower. The reason is that a disk is a mechanical device
4. I/O devices:
   1. controller + device
      1. driver: software which talks to a controller◦needs to be put into the operating system so it can be run in kernel mode.◦busy waiting◦program issues system call -> kernel sends procedure to driver -> driver starts I/O -> polls device -> driver puts data then returns◦interrupt◦driver starts device -> receives interrupt -> driver returns
   2. I/O devices generally consist of two parts: a controller and the device itself. The controller is a chip or a set of chips that physically controls the device. It accepts commands from the operating system, for example, to read data from the device, and carries them out. In many cases, the actual control of the device is complicated and detailed, so it is the job of the controller to present a simpler (but still very complex) interface to the operating system.
5. Buses: as processors and memories got faster, the ability of a single bus (and certainly the IBM PC bus) to handle all the traffic was strained to the breaking point. Something had to give. As a result, additional buses were added, both for faster I/O devices and for CPU-to-memory traffic.

Thread: a kind of lightweight process which is running a program

The BIOS contains low-level I/O software, including procedures to read the keyboard, write to the screen, and do disk I/O, among other things. Nowadays, it is held in a flash RAM, which is nonvolatile but which can be updated by the operating system when bugs are found in the BIOS.

Real-time OS

For example, if a car is moving down an assembly line, certain actions must take place at certain instants of time. If, for example, a welding robot welds too early or too late, the car will be ruined. If the action absolutely must occur at a certain moment (or within a certain range), we have a hard real-time system. Many of these are found in industrial process control, avionics, military, and similar application areas. These systems must provide absolute guarantees that a certain action will occur by a certain time.

A soft real-time system is one where missing an occasional deadline, while not desirable, is acceptable and does not cause any permanent damage. Digital audio or multimedia systems fall in this category. Smartphones are also soft real time systems.

OS concepts

* Processes: A process is basically a program in execution. Associated with each process is its address space, a list of memory locations from 0 to some maximum, which the process can read and write. The address space contains the executable program, the program’s data, and its stack. Also associated with each process is a set of resources, commonly including registers (including the program counter and stack pointer), a list of open files, outstanding alarms, lists of related processes, and all the other information needed to run the program. A process is fundamentally a container that holds all the information needed to run a program.
* Address Spaces (Not mentioned in the slide though): Every computer has some main memory that it uses to hold executing programs. In a very simple operating system, only one program at a time is in memory. To run a second program, the first one has to be removed and the second one  
  placed in memory.  
  More sophisticated operating systems allow multiple programs to be in memory at the same time. To keep them from interfering with one another (and with the  
  operating system), some kind of protection mechanism is needed. While this mechanism has to be in the hardware, it is controlled by the operating system.
* Files: a major function of the operating system is to hide the  
  peculiarities of the disks and other I/O devices and present the programmer with anice, clean abstract model of device-independent files. System calls are obviously  
  needed to create files, remove files, read files, and write files. Before a file can be  
  read, it must be located on the disk and opened, and after being read it should be  
  closed, so calls are provided to do these things.  
  To provide a place to keep files, most PC operating systems have the concept  
  of a **directory** as a way of grouping files together. The process and file hierarchies both are organized as trees, but the similarity  
  stops there. Process hierarchies usually are not very deep (more than three levels is  
  unusual), whereas file hierarchies are commonly four, fiv e, or even more levels  
  deep. Every file within the directory hierarchy can be specified by giving its **path  
  name** from the top of the directory hierarchy, the **root directory**. Such absolute  
  path names consist of the list of directories that must be traversed from the root directory to get to the file, with slashes separating the components.
* Input/Output: All computers have physical devices for acquiring input and producing output. Many kinds of input  
  and output devices exist, including keyboards, monitors, printers, and so on. It is  
  up to the operating system to manage these devices. Consequently, every operating system has an I/O subsystem for managing its  
  I/O devices. Some of the I/O software is device independent, that is, applies to  
  many or all I/O devices equally well. Other parts of it, such as device drivers, are  
  specific to particular I/O devices.
* Protection: Computers contain large amounts of information that users often want to protect and keep confidential. This information may include email, business plans, tax  
  returns, and much more. It is up to the operating system to manage the system security so that files, for example, are accessible only to authorized users. In addition to file protection, there are many other security issues. Protecting  
  the system from unwanted intruders, both human and nonhuman (e.g., viruses) is  
  one of them.
* The shell: The operating system is the code that carries out the system calls. Editors,  
  compilers, assemblers, linkers, utility programs, and command interpreters definitely are not part of the operating system, even though they are important and useful. It is also the main interface between a user sitting at his terminal and the operating system, unless the user is  
  using a graphical user interface. Many shells exist, including *sh*, *csh*, *ksh*, and *bash*. When any user logs in, a shell is started up. The shell has the terminal as standard input and standard output. It starts out by typing the **prompt**, a character  
  such as a dollar sign, which tells the user that the shell is waiting to accept a command. Most personal computers these days use a GUI. In fact, the GUI is just a program running on top of the operating system, like a shell.

The Process Model

In this model, all the runnable software on the computer, sometimes including  
the operating system, is organized into a number of **sequential processes**, or just  
**processes** for short. A process is just an instance of an executing program, including the current values of the program counter, registers, and variables. Conceptually, each process has its own virtual CPU. In reality, of course, the real CPU  
switches back and forth from process to process, but to understand the system, it is  
much easier to think about a collection of processes running in (pseudo) parallel  
than to try to keep track of how the CPU switches from program to program. This  
rapid switching back and forth is called **multiprogramming.** With the CPU switching back and forth among the processes, the rate at which  
a process performs its computation will not be uniform and probably not even  
reproducible if the same processes are run again. Thus, processes must not be programmed with built-in assumptions about timing.

Process has a program, input, output, and a state. A single processor may be shared among several  
processes, with some scheduling algorithm being accustomed to determine when to  
stop work on one process and service a different one. In contrast, a program is  
something that may be stored on disk, not doing anything.

THREAD MODEL

One way of looking at a process is that it is a way to group related resources  
together. A process has an address space containing program text and data, as well  
as other resources. These resources may include open files, child processes, pending alarms, signal handlers, accounting information, and more. By putting them  
together in the form of a process, they can be managed more easily.

The other concept a process has is a thread of execution, usually shortened to  
just **thread**. The thread has a program counter that keeps track of which instruction to execute next. It has registers, which hold its current working variables. It  
has a stack, which contains the execution history, with one frame for each procedure called but not yet returned from. Although a thread must execute in some  
process, the thread and its process are different concepts and can be treated separately. Processes are used to group resources together; threads are the entities  
scheduled for execution on the CPU.

When a multithreaded process is run on a single-CPU system, the threads take  
turns running.

The CPU switches rapidly back and forth among the threads, providing the  
illusion that the threads are running in parallel, albeit on a slower CPU than the  
real one. With three compute-bound threads in a process, the threads would appear  
to be running in parallel, each one on a CPU with one-third the speed of the real  
CPU. Different threads in a process are not as independent as different processes. All  
threads have exactly the same address space, which means that they also share the same global variables. Since every thread can access every memory address within  
the process’ address space, one thread can read, write, or even wipe out another  
thread’s stack. There is no protection between threads because (1) it is impossible,  
and (2) it should not be necessary.

Text

Description automatically generated with medium confidence

When a thread has finished its work, it can exit by calling a library procedure,  
say, *thread exit*. It then vanishes and is no longer schedulable. In some thread  
systems, one thread can wait for a (specific) thread to exit by calling a procedure,  
for example, *thread join*. This procedure blocks the calling thread until a (specific) thread has exited. In this regard, thread creation and termination is very much  
like process creation and termination, with approximately the same options as well.

**Modeling Multiprogramming**When multiprogramming is used, the CPU utilization can be improved

A better model is to look at CPU usage from a probabilistic viewpoint. Suppose that a process spends a fraction *p* of its time waiting for I/O to complete. With  
*n* processes in memory at once, the probability that all *n* processes are waiting for  
I/O (in which case the CPU will be idle) is *pn*. The CPU utilization is then given  
by the formula  
CPU utilization = 1 - *pn*

Diagram

Description automatically generated

From the figure it is clear that if processes spend 80% of their time waiting for  
I/O, at least 10 processes must be in memory at once to get the CPU waste below  
10%. When you realize that an interactive process waiting for a user to type something at a terminal (or click on an icon) is in I/O wait state, it should be clear that  
I/O wait times of 80% and more are not unusual. But even on servers, processes  
doing a lot of disk I/O will often have this percentage or more.

INTERPROCESS COMMUNICATION

Processes need to share information with each other.Processes (or threads) should not compete with each other.Processes (or threads) may be dependent on a certain sequence.

Processes frequently need to communicate with other processes. For example,  
in a shell pipeline, the output of the first process must be passed to the second  
process, and so on down the line. Thus there is a need for communication between  
processes, preferably in a well-structured way not using interrupts.

Interprocess communication is the mechanism provided by the operating system that allows processes to communicate with each other. This communication could involve a process letting another process know that some event has occurred or transferring of data from one process to another.

Race Conditions

Situations like this, where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are  
called **race conditions**.

Although this requirement avoids race conditions, it is not sufficient for having  
parallel processes cooperate correctly and efficiently using shared data. We need  
four conditions to hold to have a good solution:  
1. No two processes may be simultaneously inside their critical regions.  
2. No assumptions may be made about speeds or the number of CPUs.  
3. No process running outside its critical region may block any process.  
4. No process should have to wait forever to enter its critical region.

A semaphore could have the value 0, indicating that no wakeups were saved, or some  
positive value if one or more wakeups were pending.

SLEEP AND WAKEUP

Two  
processes share a common, fixed-size buffer. One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out. (It is also possible to generalize the problem to have *m* producers and *n* consumers, but we will  
consider only the case of one producer and one consumer because this assumption  
simplifies the solutions.)  
Trouble arises when the producer wants to put a new item in the buffer, but it is  
already full. The solution is for the producer to go to sleep, to be awakened when  
the consumer has removed one or more items. Similarly, if the consumer wants to  
remove an item from the buffer and sees that the buffer is empty, it goes to sleep  
until the producer puts something in the buffer and wakes it up.  
This approach sounds simple enough, but it leads to the same kinds of race  
conditions we saw earlier with the spooler directory. To keep track of the number  
of items in the buffer, we will need a variable, *count*. If the maximum number of  
items the buffer can hold is *N*, the producer’s code will first test to see if *count* is *N*.  
If it is, the producer will go to sleep; if it is not, the producer will add an item and  
increment *count*.

Semaphores

The down operation on a semaphore checks to see if the value is greater than 0. If so, it decrements the value (i.e., uses up one stored wakeup) and just continues. If the value is  
0, the process is put to sleep without completing the down for the moment. Checking the value, changing it, and possibly going to sleep, are all done as a single,  
indivisible **atomic action**. It is guaranteed that once a semaphore operation has  
started, no other process can access the semaphore until the operation has completed or blocked. This atomicity is absolutely essential to solving synchronization  
problems and avoiding race conditions. Atomic actions, in which a group of related  
operations are either all performed without interruption or not performed at all, are  
extremely important in many other areas of computer science as well.  
The up operation increments the value of the semaphore addressed. If one or  
more processes were sleeping on that semaphore, unable to complete an earlier  
down operation, one of them is chosen by the system (e.g., at random) and is allowed to complete its down. Thus, after an up on a semaphore with processes  
sleeping on it, the semaphore will still be 0, but there will be one fewer process  
sleeping on it. The operation of incrementing the semaphore and waking up one  
process is also indivisible. No process ever blocks doing an up, just as no process  
ev er blocks doing a wakeup in the earlier model

The producer and consumer model uses three semaphores: one called *full* for counting the number of  
slots that are full, one called *empty* for counting the number of slots that are empty,  
and one called *mutex* to make sure the producer and consumer do not access the  
buffer at the same time. *ull* is initially 0, *empty* is initially equal to the number of  
slots in the buffer, and *mutex* is initially 1. Semaphores that are initialized to 1 and  
used by two or more processes to ensure that only one of them can enter its critical  
region at the same time are called **binary semaphores**. If each process does a  
down just before entering its critical region and an up just after leaving it, mutual  
exclusion is guaranteed.

In a system using semaphores, the natural way to hide interrupts is to have a semaphore,  
initially set to 0, associated with each I/O device. Just after starting an I/O device,  
the managing process does a down on the associated semaphore, thus blocking immediately. When the interrupt comes in, the interrupt handler then does an up on  
the associated semaphore, which makes the relevant process ready to run again.

MUTEX

When the semaphore’s ability to count is not needed, a simplified version of  
the semaphore, called a mutex, is sometimes used. Mutexes are good only for managing mutual exclusion to some shared resource or piece of code. They are easy  
and efficient to implement, which makes them especially useful in thread packages  
that are implemented entirely in user space.  
A **mutex** is a shared variable that can be in one of two states: unlocked or  
locked. Consequently, only 1 bit is required to represent it, but in practice an integer often is used, with 0 meaning unlocked and all other values meaning locked.

When the semaphore’s ability to count is not needed, a simplified version of  
the semaphore, called a mutex, is sometimes used. Mutexes are good only for managing mutual exclusion to some shared resource or piece of code. They are easy  
and efficient to implement, which makes them especially useful in thread packages  
that are implemented entirely in user space.  
A **mutex** is a shared variable that can be in one of two states: unlocked or  
locked. Consequently, only 1 bit is required to represent it, but in practice an integer often is used, with 0 meaning unlocked and all other values meaning locked.

MONITORS

A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind  
of module or package. Processes may call the procedures in a monitor whenever  
they want to, but they cannot directly access the monitor’s internal data structures  
from procedures declared outside the monitor.

Monitors have an important property that makes them useful for achieving  
mutual exclusion: only one process can be active in a monitor at any instant. Monitors are a programming-language construct, so the compiler knows they are special  
and can handle calls to monitor procedures differently from other procedure calls.  
Typically, when a process calls a monitor procedure, the first few instructions of  
the procedure will check to see if any other process is currently active within the  
monitor. If so, the calling process will be suspended until the other process has left  
the monitor. If no other process is using the monitor, the calling process may enter.

SCHEDULING

When to Schedule

1.New process created2.Process exits3.Process blocks on I/O◦semaphore4.I/O interrupt◦from I/O device

Categories of Scheduling Algorithms

1.Batch.2.Interactive.3.Real time.

Scheduling in Batch Systems

•First-Come First-Served

•Shortest Job First

•Shortest Remaining Time Next

**First-Come, First-Served**Probably the simplest of all scheduling algorithms ever devised is nonpreemptive **first-come, first-served**. With this algorithm, processes are assigned the CPU  
in the order they request it. Basically, there is a single queue of ready processes.  
When the first job enters the system from the outside in the morning, it is started  
immediately and allowed to run as long as it wants to. It is not interrupted because  
it has run too long. As other jobs come in, they are put onto the end of the queue.  
When the running process blocks, the first process on the queue is run next. When  
a blocked process becomes ready, like a newly arrived job, it is put on the end of  
the queue, behind all waiting processes.

**Shortest Job First**Now let us look at another nonpreemptive batch algorithm that assumes the run  
times are known in advance. In an insurance company, for example, people can  
predict quite accurately how long it will take to run a batch of 1000 claims, since  
similar work is done every day. When several equally important jobs are sitting in  
the input queue waiting to be started, the scheduler picks the **shortest job first**.

**Shortest Remaining Time Next**A preemptive version of shortest job first is **shortest remaining time next**.  
With this algorithm, the scheduler always chooses the process whose remaining  
run time is the shortest. Again here, the run time has to be known in advance.  
When a new job arrives, its total time is compared to the current process’ remaining time. If the new job needs less time to finish than the current process, the current process is suspended and the new job started. This scheme allows new short  
jobs to get good service.

SCHEDULING in interactive system

**Round-Robin Scheduling**

Each process is assigned a time interval, called its **quantum**, during which  
it is allowed to run. If the process is still running at the end of the quantum, the  
CPU is preempted and given to another process. If the process has blocked or finished before the quantum has elapsed, the CPU switching is done when the process  
blocks, of course. Round robin is easy to implement. All the scheduler needs to do  
is maintain a list of runnable processes, as shown in Fig. 2-42(a). When the process uses up its quantum, it is put on the end of the list

The only really interesting issue with round robin is the length of the quantum.  
Switching from one process to another requires a certain amount of time for doing  
all the administration—saving and loading registers and memory maps,

**Priority Scheduling**

The need to take external factors into account leads to **priority scheduling**. The basic idea is straightforward: each process is assigned a priority, and the runnable process with the highest priority is allowed to run.  
Even on a PC with a single owner, there may be multiple processes, some of  
them more important than others. For example, a daemon process sending electronic mail in the background should be assigned a lower priority than a process  
displaying a video film on the screen in real time.  
To prevent high-priority processes from running indefinitely, the scheduler  
may decrease the priority of the currently running process at each clock tick (i.e.,  
at each clock interrupt). If this action causes its priority to drop below that of the  
next highest process, a process switch occurs. Alternatively, each process may be  
assigned a maximum time quantum that it is allowed to run. When this quantum is  
used up, the next-highest-priority process is given a chance to run.

**Multiple Queues**

processes in the  
highest class were run for one quantum. Processes in the next-highest class were  
run for two quanta. Processes in the next one were run for four quanta, etc. Whenev er a process used up all the quanta allocated to it, it was moved down one class.

**Shortest Process Next**Because shortest job first always produces the minimum average response time  
for batch systems, it would be nice if it could be used for interactive processes as  
well. To a certain extent, it can be. Interactive processes generally follow the pattern of wait for command, execute command, wait for command, execute command, etc. If we regard the execution of each command as a separate ‘‘job,’’ then  
we can minimize overall response time by running the shortest one first. The problem is figuring out which of the currently runnable processes is the shortest one.

**Guaranteed Scheduling**A completely different approach to scheduling is to make real promises to the  
users about performance and then live up to those promises. One promise that is  
realistic to make and easy to live up to is this: If *n* users are logged in while you are  
working, you will receive about 1/*n* of the CPU power. Similarly, on a single-user  
system with *n* processes running, all things being equal, each one should get 1/*n* of  
the CPU cycles.

**Lottery Scheduling**

The basic idea is to give processes lottery tickets for various system resources,  
such as CPU time. Whenever a scheduling decision has to be made, a lottery ticket  
is chosen at random, and the process holding that ticket gets the resource. When  
applied to CPU scheduling, the system might hold a lottery 50 times a second, with  
each winner getting 20 msec of CPU time as a prize.

**Fair-Share Scheduling**

s a result, if user 1 starts up nine processes and user 2  
starts up one process, with round robin or equal priorities, user 1 will get 90% of  
the CPU and user 2 only 10% of it.  
To prevent this situation, some systems take into account which user owns a  
process before scheduling it. In this model, each user is allocated some fraction of

the CPU and the scheduler picks processes in such a way as to enforce it. Thus if  
two users have each been promised 50% of the CPU, they will each get that, no  
matter how many processes they hav e in existence.

**File Systems**

**Essential requirements for long-term information storage:1.It must be possible to store a very large amount of information.2.Information must survive termination of process using it.3.Multiple processes must be able to access information concurrently.**

**Files: logical units of information created by processes**

**◦Persistent: not affected by process creation and termination**

**◦File system: part of operating system dealing with files**

**Files** are logical units of information created by processes. A disk will usually  
contain thousands or even millions of them, each one independent of the others. In  
fact, if you think of each file as a kind of address space, you are not that far off, except that they are used to model the disk instead of modeling the RAM.

A file is an abstraction mechanism. It provides a way to store information on  
the disk and read it back later. This must be done in such a way as to shield the  
user from the details of how and where the information is stored, and how the disks  
actually work.  
Probably the most important characteristic of any abstraction mechanism is the  
way the objects being managed are named, so we will start our examination of file  
systems with the subject of file naming. When a process creates a file, it gives the  
file a name. When the process terminates, the file continues to exist and can be accessed by other processes using its name.

File Naming

Many operating systems support two-part file names, with the two parts separated by a period, as in *prog.c*. The part following the period is called the **file  
extension** and usually indicates something about the file.

FILE STRUCTURE

iles can be structured in any of sev eral ways. Three common possibilities are  
depicted in Fig. 4-2. The file in Fig. 4-2(a) is an unstructured sequence of bytes.  
In effect, the operating system does not know or care what is in the file. All it sees  
are bytes. Any meaning must be imposed by user-level programs. Both UNIX and  
Windows use this approach.

Three kinds of files. (a) Byte sequence. (b) Record sequence.  
(c) Tree.

Tree: In this organization, a  
file consists of a tree of records, not necessarily all the same length, each containing a **key** field in a fixed position in the record. The tree is sorted on the key  
field, to allow rapid searching for a particular key.

FILE TYPES

Many operating systems support several types of files. UNIX (again, including  
OS X) and Windows, for example, have regular files and directories. UNIX also  
has character and block special files. **Regular files** are the ones that contain user  
information. All the files of Fig. 4-2 are regular files. **Directories** are system files  
for maintaining the structure of the file system. We will study directories below.  
**Character special files** are related to input/output and used to model serial I/O devices, such as terminals, printers, and networks. **Block special files** are used to  
model disks. In this chapter we will be primarily interested in regular files.  
Regular files are generally either ASCII files or binary files. ASCII files consist of lines of text. In some systems each line is terminated by a carriage return  
character. In others, the line feed character is used. Some systems (e.g., Windows)  
use both. Lines need not all be of the same length.

Other files are binary, which just means that they are not ASCII files. Listing  
them on the printer gives an incomprehensible listing full of random junk. Usually,  
they hav e some internal structure known to programs that use them.

Every operating system must recognize at least one file type: its own executable file; some recognize more.

**DIRECTORIES**To keep track of files, file systems normally have **directories** or **folders**, which  
are themselves files. In this section we will discuss directories, their organization,  
their properties, and the operations that can be performed on them

Single-Level Directory Systems

The simplest form of directory system is having one directory containing all  
the files. Sometimes it is called the **root directory**, but since it is the only one, the  
name does not matter much. On early personal computers, this system was common, in part because there was only one user. Interestingly enough, the world’s  
first supercomputer, the CDC 6600, also had only a single directory for all files,  
ev en though it was used by many users at once. This decision was no doubt made  
to keep the software design simple.  
An example of a system with one directory is given in Fig. 4-6. Here the directory contains four files. The advantages of this scheme are its simplicity and the  
ability to locate files quickly—there is only one place to look, after all. It is sometimes still used on simple embedded devices such as digital cameras and some  
portable music players.

**Hierarchical Directory Systems**The single level is adequate for very simple dedicated applications (and was  
ev en used on the first personal computers), but for modern users with thousands of  
files, it would be impossible to find anything if all files were in a single directory.

PATH NAMES

When the file system is organized as a directory tree, some way is needed for  
specifying file names. Two different methods are commonly used. In the first  
method, each file is given an **absolute path name** consisting of the path from the root directory to the file. As an example, the path */usr/ast/mailbox* means that the  
root directory contains a subdirectory *usr*, which in turn contains a subdirectory  
*ast*, which contains the file *mailbox*. Absolute path names always start at the root  
directory and are unique. In UNIX the components of the path are separated by /.  
In Windows the separator is \ . In MULTICS it was >. Thus, the same path name  
would be written as follows in these three systems:

|  |  |
| --- | --- |
| Windows  UNIX  MULTICS | \usr\ast\mailbox /usr/ast/mailbox >usr>ast>mailbox |

No matter which character is used, if the first character of the path name is the separator, then the path is absolute.  
The other kind of name is the **relative path name**. This is used in conjunction  
with the concept of the **working directory** (also called the **current directory**). A  
user can designate one directory as the current working directory, in which case all  
path names not beginning at the root directory are taken relative to the working directory. For example, if the current working directory is */usr/ast*, then the file  
whose absolute path is */usr/ast/mailbox* can be referenced simply as *mailbox*. In  
other words, the UNIX command

**4.3.1 File-System Layout**File systems are stored on disks. Most disks can be divided up into one or  
more partitions, with independent file systems on each partition. Sector 0 of the  
disk is called the **MBR** (**Master Boot Record**) and is used to boot the computer.  
The end of the MBR contains the partition table. This table gives the starting and  
ending addresses of each partition. One of the partitions in the table is marked as  
active. When the computer is booted, the BIOS reads in and executes the MBR.  
The first thing the MBR program does is locate the active partition, read in its first  
block, which is called the **boot block**, and execute it.

Contagious blocks

The simplest allocation scheme is to store each file as a contiguous run of disk  
blocks. Thus on a disk with 1-KB blocks, a 50-KB file would be allocated 50 consecutive blocks. With 2-KB blocks, it would be allocated 25 consecutive blocks.

Contiguous disk-space allocation has two significant advantages. First, it is  
simple to implement because keeping track of where a file’s blocks are is reduced  
to remembering two numbers: the disk address of the first block and the number of  
blocks in the file. Given the number of the first block, the number of any other  
block can be found by a simple addition.  
Second, the read performance is excellent because the entire file can be read  
from the disk in a single operation. Only one seek is needed (to the first block).  
After that, no more seeks or rotational delays are needed, so data come in at the  
full bandwidth of the disk. Thus contiguous allocation is simple to implement and  
has high performance.  
Unfortunately, contiguous allocation also has a very serious drawback: over the  
course of time, the disk becomes fragmented. To see how this comes about, examine

I-nodes

**I-nodes**Our last method for keeping track of which blocks belong to which file is to  
associate with each file a data structure called an **i-node** (**index-node**), which lists  
the attributes and disk addresses of the file’s blocks.

The big advantage of this scheme over linked files using an in-memory table is that  
the i-node need be in memory only when the corresponding file is open. If each inode occupies *n* bytes and a maximum of *k* files may be open at once, the total  
memory occupied by the array holding the i-nodes for the open files is only *kn*bytes. Only this much space need be reserved in advance.

**Disk-Space Management**Files are normally stored on disk, so management of disk space is a major concern to file-system designers. Two general strategies are possible for storing an *n*byte file: *n* consecutive bytes of disk space are allocated, or the file is split up into  
a number of (not necessarily) contiguous blocks. The same trade-off is present in  
memory-management systems between pure segmentation and paging. The same  
problem holds for segments in memory, except that moving a segment in memory  
is a relatively fast operation compared to moving a file from one disk position to  
another. For this reason, nearly all file systems chop files up into fixed-size blocks  
that need not be adjacent.

**File-System Backups**Destruction of a file system is often a far greater disaster than destruction of a  
computer.

Most people do not think making backups of their files is worth the time and  
effort—until one fine day their disk abruptly dies, at which time most of them  
undergo a deathbed conversion. Companies, however, (usually) well understand the  
value of their data and generally do a backup at least once a day, often to tape.  
Modern tapes hold hundreds of gigabytes and cost pennies per gigabyte. Nevertheless, making backups is not quite as trivial as it sounds, so we will examine some  
of the related issues below.  
Backups to tape are generally made to handle one of two potential problems:  
1. Recover from disaster.  
2. Recover from stupidity

Making a backup takes a long time and occupies a large amount of space, so  
doing it efficiently and conveniently is important. These considerations raise the  
following issues. First, should the entire file system be backed up or only part of  
it? At many installations, the executable (binary) programs are kept in a limited  
part of the file-system tree.

PERFORMANCE

Access to disk is much slower than access to memory. Reading a 32-bit memory word might take 10 nsec. Reading from a hard disk might proceed at 100  
MB/sec,

Memory management

Address spaces

◦set of addresses that process can use to access memory

◦each process has its own space

◦two problems to solve: protection and relocation

The basic idea behind virtual memory is that each program has  
its own address space, which is broken up into chunks called **pages**. Each page is  
a contiguous range of addresses. These pages are mapped onto physical memory,  
but not all pages have to be in physical memory at the same time to run the program. When the program references a part of its address space that is in physical memory, the hardware performs the necessary mapping on the fly. When the program references a part of its address space that is *not* in physical memory, the operating system is alerted to go get the missing piece and re-execute the instruction  
that failed.

Paging

When virtual memory is used, the virtual addresses do not go directly to the memory bus. Instead, they go to an **MMU** (**Memory Management Unit**) that maps the virtual addresses onto the physical memory  
addresses

The virtual page number is used as an index into the page table to find the  
entry for that virtual page. From the page table entry, the page frame number (if  
any) is found. The page frame number is attached to the high-order end of the offset, replacing the virtual page number, to form a physical address that can be sent  
to the memory.

Thus, the purpose of the page table is to map virtual pages onto page frames.  
Mathematically speaking, the page table is a function, with the virtual page number as argument and the physical frame number as result. Using the result of this

function, the virtual page field in a virtual address can be replaced by a page frame  
field, thus forming a physical memory address.

The solution that has been devised is to equip computers with a small hardware  
device for mapping virtual addresses to physical addresses without going through  
the page table. The device, called a **TLB** (**Translation Lookaside Buffer**) or  
sometimes an **associative memory**, is illustrated in Fig. 3-12. It is usually inside  
the MMU and consists of a small number of entries, eight in this example, but  
rarely more than 256.

When a page fault occurs, the operating system has to choose a page to evict  
(remove from memory) to make room for the incoming page. If the page to be removed has been modified while in memory, it must be rewritten to the disk to bring  
the disk copy up to date. If, however, the page has not been changed (e.g., it contains program text), the disk copy is already up to date, so no rewrite is needed.  
The page to be read in just overwrites the page being evicted.

The best possible page replacement algorithm is easy to describe but impossible to actually implement. It goes like this. At the moment that a page fault occurs, some set of pages is in memory.

The optimal page replacement algorithm says that the page with the highest  
label should be removed. I

FIFO

The operating  
system maintains a list of all pages currently in memory, with the most recent arrival at the tail and the least recent arrival at the head. On a page fault, the page at  
the head is removed and the new page added to the tail of the list. When applied to  
stores, FIFO might remove mustache wax, but it might also remove flour, salt, or  
butter. When applied to computers the same problem arises: the oldest page may  
still be useful. For this reason, FIFO in its pure form is rarely used.

A simple modification to FIFO that avoids the problem of throwing out a heavily used page is to inspect the *R* bit of the oldest page. If it is 0, the page is both  
old and unused, so it is replaced immediately. If the *R* bit is 1, the bit is cleared,  
the page is put onto the end of the list of pages, and its load time is updated as  
though it had just arrived in memory. Then the search continues.  
The operation of this algorithm, called **second chance**,

**3.5.3 Page Size**The page size is a parameter that can be chosen by the operating system. Even  
if the hardware has been designed with, for example, 4096-byte pages, the operating system can easily regard page pairs 0 and 1, 2 and 3, 4 and 5, and so on, as  
8-KB pages by always allocating two consecutive 8192-byte page frames for them.

Determining the best page size requires balancing several competing factors.  
As a result, there is no overall optimum. To start with, two factors argue for a  
small page size. A randomly chosen text, data, or stack segment will not fill an  
integral number of pages. On the average, half of the final page will be empty.  
The extra space in that page is wasted. This wastage is called **internal fragmentation**. With *n* segments in memory and a page size of *p* bytes, *np*/2 bytes will be  
wasted on internal fragmentation. This reasoning argues for a small page size.

write will be applied.  
In modern systems, there are many large libraries used by many processes, for  
example, multiple I/O and graphics libraries. Statically binding all these libraries to  
ev ery executable program on the disk would make them even more bloated than  
they already are.  
Instead, a common technique is to use **shared libraries** (which are called  
**DLLs** or **Dynamic Link Libraries** on Windows). To make the idea of a shared

Segmentation

* + Memory management approach where the process is divided into the variable size segments and loaded to the logical memory address space
  + The logical address space is the collection of variable size segments. Each segment has its name and length.
  + For the execution, the segments from logical memory space are loaded to the physical memory space.

FIG 3-29

Once the process starts running, it may get a page fault. The fault handler figures out which virtual page is needed and sends a message to the external pager,  
telling it the problem. The external pager then reads the needed page in from the  
disk and copies it to a portion of its own address space. Then it tells the fault handler where the page is. The fault handler then unmaps the page from the external  
pager’s address space and asks the MMU handler to put it into the user’s address  
space at the right place. Then the user process can be restarted