**<<<<<<<<<<<<<<<<<<<- Solution Set ->>>>>>>>>>>>>>>>>>>**

**1. Attempt *any three* of the following:**

**a.** What is Operating System? Explain the role of OS as extended machine.

The **architecture** (instruction set, memory organization, I/O, and bus structure) of most computers at the machine-language level is primitive and awkward to program, especially for input/output. To make this point more concrete, consider modern **SATA** (**Serial ATA**) hard disks used on most computers. A book (Anderson, 2007) describing an early version of the interface to the disk—what a programmer would have to know to use the disk—ran over 450 pages. Since then, the interface has been revised multiple times and is more complicated than it was in 2007. Clearly, no sane programmer would want to deal with this disk at the hardware level. Instead, a piece of software, called a **disk driver**, deals with the hardware and provides an interface to read and write disk blocks, without getting into the details. Operating systems contain many drivers for controlling I/O devices. But even this level is much too low for most applications. For this reason, all operating systems provide yet another layer of abstraction for using disks: files. Using this abstraction, programs can create, write, and read files, without having to deal with the messy details of how the hardware actually works. This abstraction is the key to managing all this complexity. Good abstractions turn a nearly impossible task into two manageable ones. The first is defining and implementing the abstractions. The second is using these abstractions to solve the problem at hand. One abstraction that almost every computer user understands is the file, as mentioned above. It is a useful piece of information, such as a digital photo, saved email message, song, or Web page. It is much easier to deal with photos, emails, songs, and Web pages than with the details of SATA (or other) disks.

The job of the operating system is to create good abstractions and then implement and manage the abstract objects thus created. In this book, we will talk a lot about abstractions. They are one of the keys to understanding operating systems. This point is so important that it is worth repeating in different words. With all due respect to the industrial engineers who so carefully designed the Macintosh, hardware is ugly. Real processors, memories, disks, and other devices are very complicated and present difficult, awkward, idiosyncratic, and inconsistent interfaces to the people who have to write software to use them. Sometimes this is due to the need for backward compatibility with older hardware. Other times it is an attempt to save money.

Application programs
Beautiful interface
Operating system
Ugly interface
Hardware

Often, however, the hardware designers do not realize (or care) how much trouble they are causing for the software. One of the major tasks of the operating system is to hide the hardware and present programs (and their programmers) with nice, clean, elegant, consistent, abstractions to work with instead.

Operating systems turn the ugly into the beautiful, as shown in Fig

**b.** Write a short note on Fifth Generation of Operating System.

**The Fifth Generation (1990–Present): Mobile Computers**

The first real mobile phone appeared in 1946 and weighed some 40 kilos. You could take it wherever you went as long as you had a car in which to carry it. The first true handheld phone appeared in the 1970s and, at roughly one kilogram, was positively featherweight. It was affectionately known as ‘‘the brick.’’ Pretty soon everybody wanted one. Today, mobile phone penetration is close to 90% of the global population. We can make calls not just with our portable phones and wrist watches, but soon with eyeglasses and other wearable items. Moreover, the phone part is no longer that interesting. We receive email, surf the Web, text our friends, play games, navigate around heavy traffic—and do not even think twice about it.

While the idea of combining telephony and computing in a phone-like device has been around since the 1970s also, the first real smartphone did not appear until the mid-1990s when Nokia released the N9000, which literally combined two, mostly separate devices: a phone and a **PDA** (Personal Digital Assistant). In 1997, Ericsson coined the term *smartphone* for its GS88 ‘‘Penelope.’’

Now that smartphones have become ubiquitous, the competition between the various operating systems is fierce and the outcome is even less clear than in the PC world. At the time of writing, Google’s Android is the dominant operating system with Apple’s iOS a clear second, but this was not always the case and all may be different again in just a few years. If anything is clear in the world of smartphones, it is that it is not easy to stay king of the mountain for long.

After all, most smartphones in the first decade after their inception were running **Symbian** OS. It was the operating system of choice for popular brands like Samsung, Sony Ericsson, Motorola, and especially Nokia. However, other operating systems like **RIM's** Blackberry OS (introduced for smartphones in 2002) and Apple's iOS (released for the first **iPhone** in 2007) started eating into Symbian's market share. Many expected that RIM would dominate the business market, while iOS would be the king of the consumer devices. Symbian's market share plummeted.
In 2011, Nokia ditched Symbian and announced it would focus on Windows Phone as its primary platform. For some time, Apple and RIM were the toastof the town (although not nearly as dominant as Symbian had been), but it did not take very long for Android, a Linux-based operating system released by Google in 2008, to overtake all its rivals.
For phone manufacturers, Android had the advantage that it was open source and available under a permissive license. As a result, they could tinker with it and adapt it to their own hardware with ease. Also, it has a huge community of developers writing apps, mostly in the familiar Java programming language. Even so, the past years have shown that the dominance may not last, and Android's competitors are eager to claw back some of its market share.

c. Explain multithreaded and multi-core chips.

Moore's law states that the number of transistors on a chip doubles every 18 months. This ''law'' is not some kind of law of physics, like conservation of momentum, but is an observation by Intel cofounder Gordon Moore of how fast process engineers at the semiconductor companies are able to shrink their transistors. Moore's law has held for over three decades now and is expected to hold for at least one more. After that, the number of atoms per transistor will become too small and quantum mechanics will start to play a big role, preventing further shrinkage of transistor sizes.
The abundance of transistors is leading to a problem: what to do with all of them? We saw one approach above: superscalar architectures, with multiple functional units. But as the number of transistors increases, even more is possible. One obvious thing to do is put bigger caches on the CPU chip. That is definitely happening, but eventually the point of diminishing returns will be reached. The obvious next step is to replicate not only the functional units, but also some of the control logic. The Intel Pentium 4 introduced this property, called **multithreading** or **hyperthreading** (Intel's name for it), to the x86 processor, and several other CPU chips also have it—including the SPARC, the Power5, the Intel Xeon, and the Intel Core family. To a first approximation, what it does is allow the CPU to hold the state of two different threads and then switch back and forth on a nanosecond time scale. (A thread is a kind of lightweight process, which, in turn, is a running program; we will get into the details in Chap. 2.) For example, if one of the processes needs to read a word from memory (which takes many clock cycles), a multithreaded CPU can just switch to another thread. Multithreading does not offer true parallelism. Only one process at a time is running, but thread-switching time is reduced to the order of a nanosecond. Multithreading has implications for the operating system because each thread appears to the operating system as a separate CPU. Consider a system with two actual CPUs, each with two threads. The operating system will see this as four CPUs. If there is only enough work to keep two CPUs busy at a certain point inBeyond multithreading, many CPU chips now hav e four, eight, or more complete processors or **cores** on them. The multicore chips of Fig. 1-8 effectively carry four minichips on them, each with its own independent CPU. (The caches will be explained below.) Some processors, like Intel Xeon Phi and the Tilera TilePro, already sport more than 60 cores on a single chip. Making use of such a multicore chip will definitely require a multiprocessor operating system.
Incidentally, in terms of sheer numbers, nothing beats a modern **GPU** (**Graphics Processing Unit**). A GPU is a processor with, literally, thousands of tiny cores.
They are very good for many small computations done in parallel, like rendering polygons in graphics applications. They are not so good at serial tasks. They are also hard to program. While GPUs can be useful for operating systems (e.g., encryption or processing of network traffic), it is not likely that much of the operating system itself will run on the GPUs.

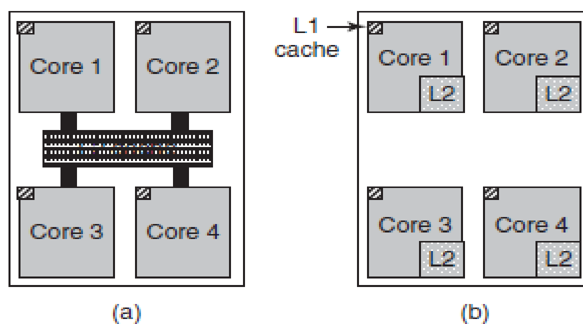



**Figure 1-8.** (a) A quad-core chip with a shared L2 cache. (b) A quad-core chip with separate L2 caches.

d. Using suitable diagram explain the structure of disk drive.

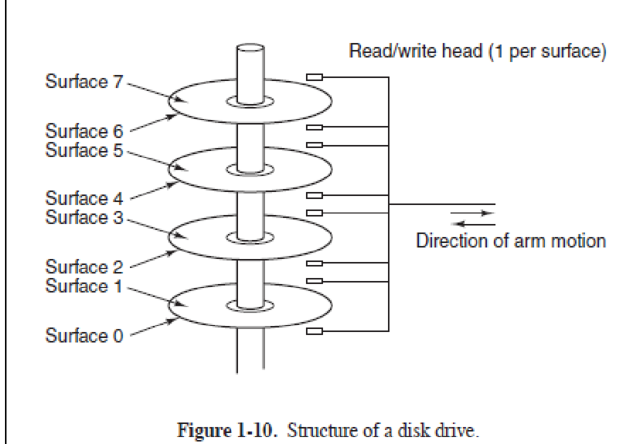



**Figure 1-10.** Structure of a disk drive.

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, as shown in Fig.
A disk consists of one or more metal platters that rotate at 5400, 7200, 10,800 RPM or more. A mechanical arm pivots over the platters from the corner, similar to the pickup arm on an old 33-RPM phonograph for playing vinyl records.

Information is written onto the disk in a series of concentric circles. At any given arm position, each of the heads can read an annular region called a **track**. Together, all the tracks for a given arm position form a **cylinder**. Each track is divided into some number of sectors, typically 512 bytes per sector. On modern disks, the outer cylinders contain more sectors than the inner ones.
Moving the arm from one cylinder to the next takes about 1 msec. Moving it to a random cylinder typically takes 5 to 10 msec, depending on the drive. Once the arm is on the correct track, the drive must wait for the needed sector to rotate under the head, an additional delay of 5 msec to 10 msec, depending on the drive's RPM. Once the sector is under the head, reading or writing occurs at a rate of 50 MB/sec on low-end disks to 160 MB/sec on faster ones.

e. Write a short note on 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**

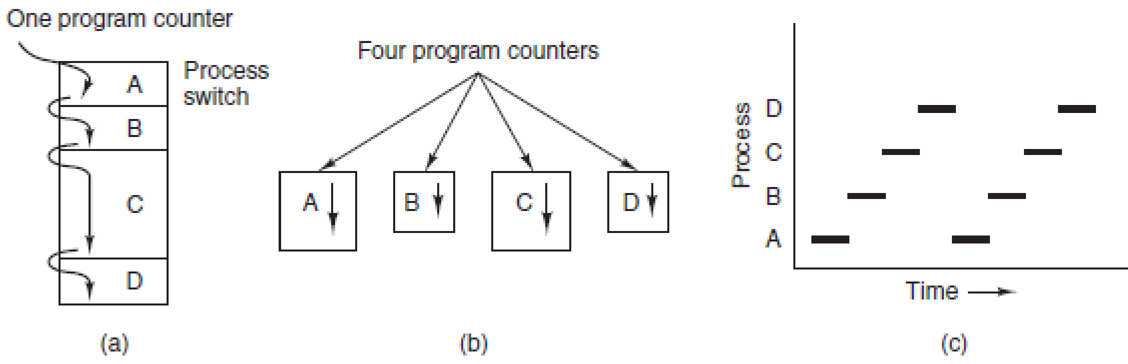
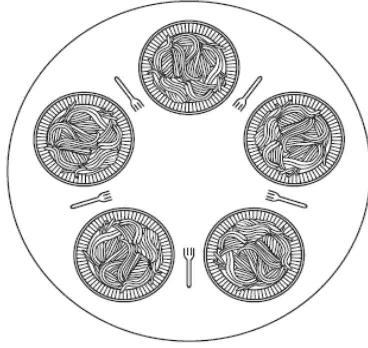



**Figure 2-1.** (a) Multiprogramming four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

we will assume there is only one CPU. Increasingly, however, that assumption is not true, since new chips are often multicore, with two, four, or more cores. We will look at multicore chips and multiprocessors, but for the time being, it is simpler just to think of one CPU at a time. So when we say that a CPU can really run only one process at a time, if there are two cores (or CPUs) each of them can run only one process at a time. 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. Consider, for example, an audio process that plays music to accompany a high-quality video run by another device. Because the audio should start a little later than the video, it signals the video server to start playing, and then runs an idle loop 10,000 times before playing back the audio. All goes well, if the loop is a reliable timer, but if the CPU decides to switch to another process during the idle loop, the audio process may not run again until the corresponding video frames have already come and gone, and the video and audio will be annoyingly out of sync. When a process has critical real-time requirements like this, that is, particular events *must* occur within a specified number of milliseconds, special measures must be taken to ensure that they do occur. Normally, however, most processes are not affected by the underlying multiprogramming of the CPU or the relative speeds of different processes.

**f.** Explain the dining philosophers problem.



Five philosophers are seated around a circular table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates is one fork. The layout of the table is illustrated in figure.
The life of a philosopher consists of alternating periods of eating and thinking. (This is something of an abstraction, even for philosophers, but the other activities are irrelevant here.)

When a philosopher gets sufficiently hungry, she tries to acquire her left and right forks, one at a time, in either order. If successful in acquiring two forks, she eats for a while, then puts down the forks, and continues to think.

**2. Attempt *any three* of the following:**

**a.** Explain the concept of running multiple programs without memory abstraction.

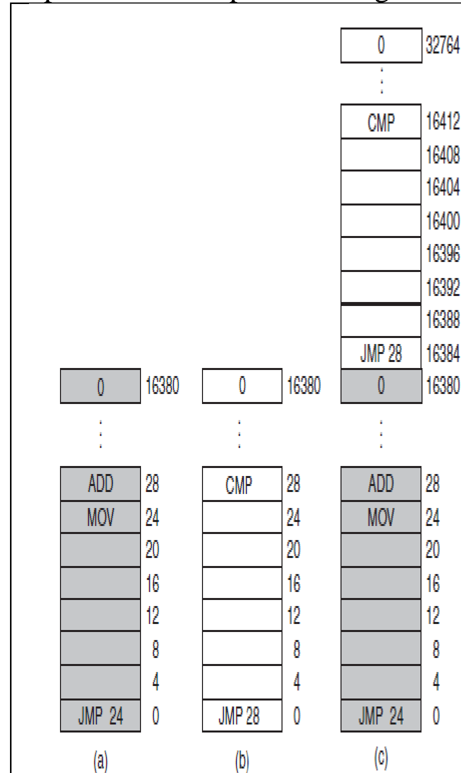



Figure 3-2. Illustration of the relocation problem. (a) A 16-KB program. (b) Another 16-KB program. (c) The two programs loaded consecutively into memory.

it is possible to run multiple programs concurrently, even without swapping. The early models of the IBM 360 solved the problem as follows. Memory was divided into 2-KB blocks and each was assigned a 4-bit protection key held in special registers inside the CPU. A machine with a 1-MB memory needed only 512 of these 4-bit registers for a total of 256 bytes of key storage. The PSW (Program Status Word) also contained a 4-bit key. The 360 hardware trapped any attempt by a running process to access memory with a protection code different from the PSW key. Since only the operating system could change the protection keys, user processes were prevented from interfering with one another and with the operating system itself. Nevertheless, this solution had a major drawback, depicted in Fig. 3-2. Here we have two programs, each 16 KB in size, as shown in Fig. 3-2(a) and (b). The former is shaded to indicate that it has a different memory key than the latter. The first program starts out by jumping to address 24, which contains a MOV instruction.

The second program starts out by jumping to address 28, which contains a CMP instruction. The instructions that are not relevant to this discussion are not shown. When the two programs are loaded consecutively in memory starting at address 0, we have the situation of Fig. 3-2(c). For this example, we assume the operating system is in high memory and thus not shown.

**b.** How swapping helps to hold large programs in RAM? Explain Using suitable diagram.

If the physical memory of the computer is large enough to hold all the processes, the schemes described so far will more or less do. But in practice, the total amount of RAM needed by all the processes is often much more than can fit in memory. On a typical Windows, OS X, or Linux system, something like 50–100 processes or more may be started up as soon as the computer is booted. For example, when a Windows application is installed, it often issues commands so that
on subsequent system boots, a process will be started that does nothing except check for updates to the application. Such a process can easily occupy 5–10 MB of memory. Other background processes check for incoming mail, incoming network connections, and many other things. And all this is before the first user program is started. Serious user application programs nowadays, like Photoshop, can easily require 500 MB just to boot and many gigabytes once they start processing data.
Consequently, keeping all processes in memory all the time requires a huge\ amount of memory and cannot be done if there is insufficient memory.

Two general approaches to dealing with memory overload have been developed over the years. The simplest strategy, called **swapping**, consists of bringing in each process in its entirety, running it for a while, then putting it back on the disk. Idle processes are mostly stored on disk, so they do not take up any memory when they are not running (although some of them wake up periodically to do their work, then go to sleep again). The other strategy, called **virtual memory**, allows programs to run even when they are only partially in main memory.

The operation of a swapping system is illustrated in Fig. 3-4. Initially, only process $A$ is in memory. Then processes $B$ and $C$ are created or swapped in from disk. In Fig. 3-4(d) $A$ is swapped out to disk. Then $D$ comes in and $B$ goes out. Finally $A$ comes in again. Since $A$ is now at a different location, addresses contained in it must be relocated, either by software when it is swapped in or (more likely) by hardware during program execution. For example, base and limit registers would work fine here.

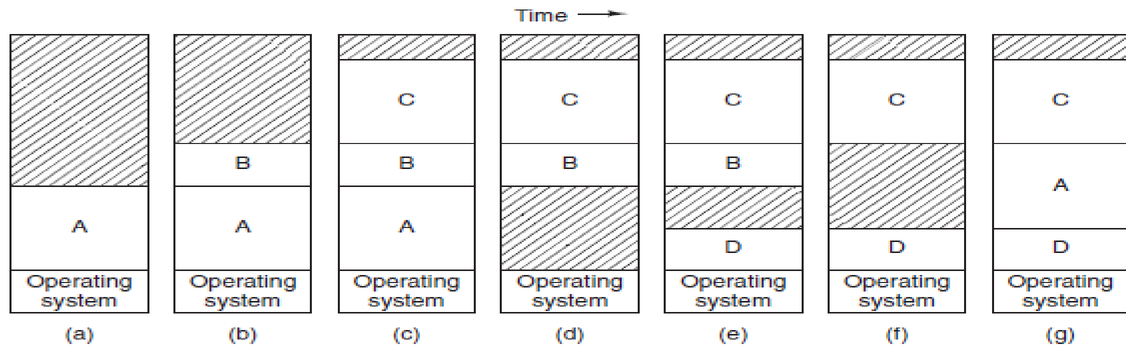



**Figure 3-4.** Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

c. Explain Clock page replacement algorithm, using suitable example.

A better approach
is to keep all the page frames on a circular list in the form of a clock, as shown in
Fig. The hand points to the oldest page.

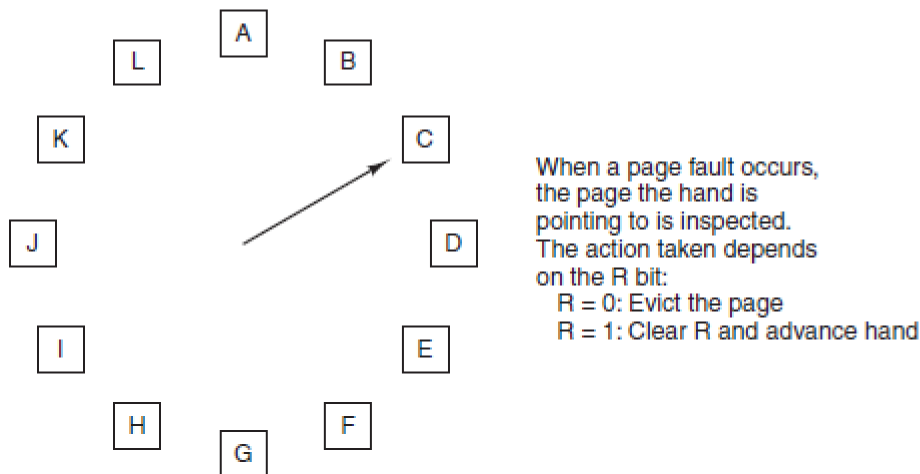



**Figure 3-16.** The clock page replacement algorithm.

When a page fault occurs, the page being pointed to by the hand is inspected. If its $R$ bit is 0, the page is evicted, the new page is inserted into the clock in its place, and the hand is advanced one position. If $R$ is 1, it is cleared and the hand is advanced to the next page. This process is repeated until a page is found with $R = 0$. Not surprisingly, this algorithm is called **clock**.

d. List and explain any five operations performed on Files.

Files exist to store information and allow it to be retrieved later. Different systems provide different operations to allow storage and retrieval.
The most common system calls relating to files.

1. Create. The file is created with no data. The purpose of the call is to announce that the file is coming and to set some of the attributes.

2. Delete. When the file is no longer needed, it has to be deleted to free up disk space. There is always a system call for this purpose.

3. Open. Before using a file, a process must open it. The purpose of the open call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls.

4. Close. When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal table space. Many systems encourage this by imposing a

5. Read. Data are read from file. Usually, the bytes come from the current position. The caller must specify how many data are needed and must also provide a buffer to put them in.

6. Write. Data are written to the file again, usually at the current position. If the current position is the end of the file, the file's size increases. If the current position is in the middle of the file, existing data are overwritten and lost forever.

7. Append. This call is a restricted form of write. It can add data only to the end of the file. Systems that provide a minimal set of system calls rarely have append, but many systems provide multiple ways of doing the same thing, and these systems sometimes have append.

8. Seek. For random-access files, a method is needed to specify from where to take the data. One common approach is a system call, seek, that repositions the file pointer to a specific place in the file. After this call has completed, data can be read from, or written to, that position.

9. Get attributes. Processes often need to read file attributes to do their work. For example, the UNIX *make* program is commonly used to manage software development projects consisting of many source files. When *make* is called, it examines the modification times of all the source and object files and arranges for the minimum number of compilations required to bring everything up to date. To do its job, it must look at the attributes, namely, the modification times.

10. Set attributes. Some of the attributes are user settable and can be changed after the file has been created. This system call makes that possible. The protection-mode information is an obvious example. Most of the flags also fall in this category.

11. Rename. It frequently happens that a user needs to change the name of an existing file. This system call makes that possible.

e. Explain the Unix V 7 File system.

The file system is in the form of a tree starting at the root directory, with the addition of links, forming a directed acyclic graph. File names can be up to 14 characters and can contain any ASCII characters except / (because that is the separator between components in a path) and NUL (because that is used to pad out names shorter than 14 characters). NUL has the numerical value of 0. A UNIX directory entry contains one entry for each file in that directory. Each entry is extremely simple because UNIX uses the i-node scheme. A directory entry contains only two fields: the file name (14 bytes) and the number of the i-node for that file (2 bytes). These parameters limit the number of files per file system to 64K.

The UNIX i-node contains some attributes. The attributes contain the file size, three times (creation, last access, and last modification), owner, group, protection information, and a count of the number of directory entries that point to the i-node. The latter field is needed due to links. Whenever a new link is made to an i-node, the count in the i-node is increased. When a link is removed, the count is decremented. When it gets to 0, the i-node is reclaimed and the disk blocks are put back in the free list. Keeping track of disk blocks is done using a generalization of Fig. 4-13 in order to handle very large files. The first 10 disk addresses are stored in the i-node itself, so for small files, all the necessary information is right in the i-node, which is fetched from disk to main memory when the file is opened. For somewhat larger files, one of the addresses in the i-node is the address of a disk block called a **single indirect block**. This block contains additional disk addresses. If this still is
not enough, another address in the i-node, called a **double indirect block**, contains the address of a block that contains a list of single indirect blocks. Each of these single indirect blocks points to a few hundred data blocks. If even this is not enough, a **triple indirect block** can also be used. The complete picture is given in Fig

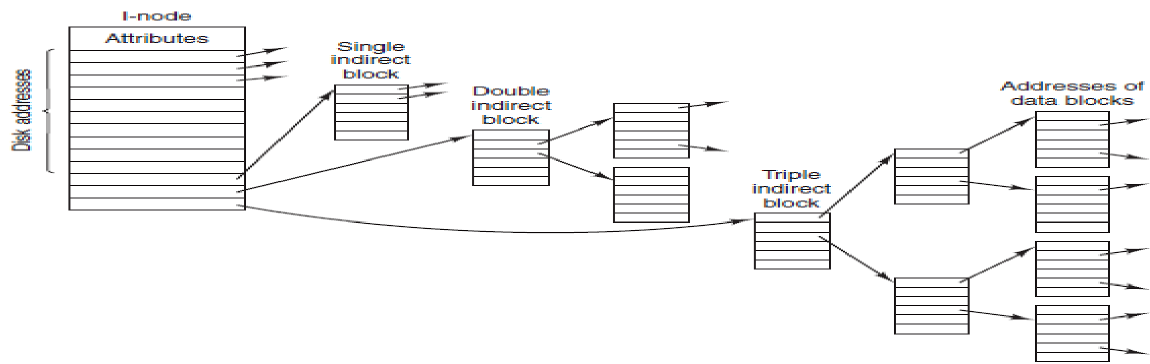


Figure 4-33. A UNIX i-node.

| | |
|---|---|
| f. | List and explain any five operations performed on Directories.<br>1. Create. A directory is created. It is empty except for dot and dotdot, which are put there automatically by the system (or in a few cases, by the *mkdir* program).<br><br>2. Delete. A directory is deleted. Only an empty directory can be deleted.<br>A directory containing only dot and dotdot is considered empty as these cannot be deleted.<br><br>3. Opendir. Directories can be read. For example, to list all the files in a directory, a listing program opens the directory to read out the names of all the files it contains. Before a directory can be read, it must be opened, analogous to opening and reading a file.<br><br>4. Closedir. When a directory has been read, it should be closed to free up internal table space.<br><br>5. Readdir. This call returns the next entry in an open directory. Formerly, it was possible to read directories using the usual read system call, but that approach has the disadvantage of forcing the programmer to know and deal with the internal structure of directories. In contrast, readdir always returns one entry in a standard format, no matter which of the possible directory structures is being used.<br><br>6. Rename. In many respects, directories are just like files and can be renamed the same way files can be.<br><br>7. Link. Linking is a technique that allows a file to appear in more than one directory.<br><br>8. Unlink. A directory entry is removed. If the file being unlinked is only present in one directory (the normal case), it is removed from the file system. |
| | |
| **3.** | **Attempt *any three* of the following:** |
| a. | What are block devices and character devices? Explain.<br>I/O devices can be roughly divided into two categories: **block devices** and **character devices**. A block device is one that stores information in fixed-size blocks, each one with its own address. Common block sizes range from 512 to 65,536 bytes. All transfers are in units of one or more entire (consecutive) blocks. The essential property of a block device is that it is possible to read or write each block independently of all the other ones. Hard disks, Blu-ray discs, and USB sticks are common block devices. If you look very closely, the boundary between devices that are block addressable and those that are not is not well defined. Everyone agrees that a disk is a block addressable device because no matter where the arm currently is, it is always possible to seek to another cylinder and then wait for the required block to rotate under the head. Now consider an old-fashioned tape drive still used, sometimes, for making disk backups (because tapes are cheap). Tapes contain a sequence of blocks. If the tape drive is giv en a command to read block *N*, it can always rewind the tape and go forward until it comes to block *N*. This operation is analogous to a disk doing a seek, except that it takes much longer. Also, it may or may not be possible to rewrite one block in the middle of a tape. Even if it were possible to use tapes as random access block devices, that is stretching the point somewhat: they are normally not used that way.<br><br>The other type of I/O device is the character device. A character device delivers or accepts a stream of characters, without regard to any block structure. It is not addressable and does not have any seek operation. Printers, network interfaces, mice (for pointing), rats (for psychology lab experiments), and most other devices that are not disk-like can be seen as character devices. |

b. Write a short note on Memory Mapped IO.

In addition to the control registers, many devices have a data buffer that the operating system can read and write. For example, a common way for computers to display pixels on the screen is to have a video RAM, which is basically just a data buffer, available for programs or the operating system to write into. The issue thus arises of how the CPU communicates with the control registers and also with the device data buffers. Two alternatives exist. In the first approach, each control register is assigned an **I/O port** number, an 8- or 16-bit integer. The set of all the I/O ports form the **I/O port space**, which is protected so that ordinary user programs cannot access it (only the operating system can). Using a special I/O instruction such as

```
IN REG,PORT,
```

the CPU can read in control register PORT and store the result in CPU register REG. Similarly, using

```
OUT PORT,REG
```

the CPU can write the contents of REG to a control register.

In this scheme, the address spaces for memory and I/O are different, as shown in Fig. 5-2(a). The instructions

```
IN R0,4
```

and

```
MOV R0,4
```

are completely different in this design. The former reads the contents of I/O port 4 and puts it in R0 whereas the latter reads the contents of memory word 4 and puts it in R0. The 4s in these examples refer to different and unrelated address spaces.

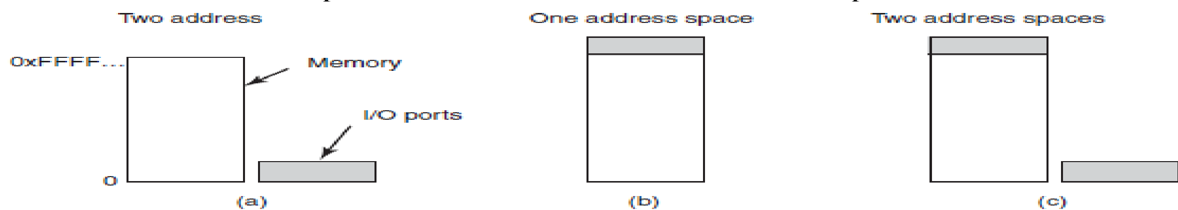



**Figure 5-2.** (a) Separate I/O and memory space. (b) Memory-mapped I/O. (c) Hybrid.

c. Explain Direct Memory Access using suitable diagram.

No matter whether a CPU does or does not have memory-mapped I/O, it needs to address the device controllers to exchange data with them. The CPU can request data from an I/O controller one byte at a time, but doing so wastes the CPU's time, so a different scheme, called **DMA** (**Direct Memory Access**) is often used. To simplify the explanation, we assume that the CPU accesses all devices and memory via a single system bus that connects the CPU, the memory, and the I/O devices, as shown in Fig. 5-4. We already know that the real organization in modern systems is more complicated, but all the principles are the same. The operating system can use only DMA if the hardware has a DMA controller, which most systems do. Sometimes this controller is integrated into disk controllers and other controllers, but such a design requires a separate DMA controller for each device. More commonly, a single DMA controller is available (e.g., on the parentboard) for regulating transfers to multiple devices, often concurrently.

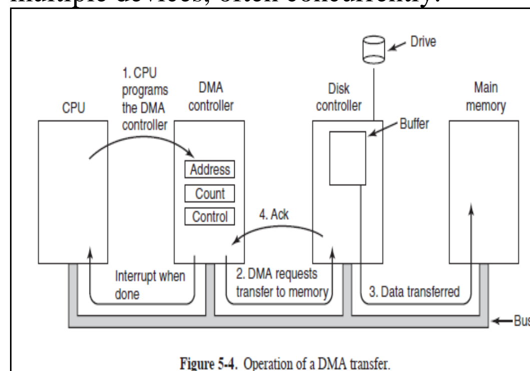



**Figure 5-4.** Operation of a DMA transfer.

No matter where it is physically located, the DMA controller has access to the system bus independent of the CPU, as shown in Fig. 5-4. It contains several registers that can be written and read by the CPU. These include a memory address register, a byte count register, and one or more control registers. The control registers specify the I/O port to use, the direction of the transfer (reading from the I/O device or writing to the I/O device), the transfer unit (byte at a time or word at a time), and the number of bytes to transfer in one burst.

d. Explain preemptable and non-preemptable resources.

Resources come in two types: preemptable and nonpreemptable. A **preemptable resource** is one that can be taken away from the process owning it with no ill effects. Memory is an example of a preemptable resource. Consider, for example, a system with 1 GB of user memory, one printer, and two 1-GB processes that each want to print something. Process *A* requests and gets the printer, then starts to compute the values to print. Before it has finished the computation, it exceeds its time quantum and is swapped out to disk.

Process *B* now runs and tries, unsuccessfully as it turns out, to acquire the printer. Potenially, we now hav e a deadlock situation, because *A* has the printer and *B* has the memory, and neither one can

| | |
|---|---|
| | proceed without the resource held by the other. Fortunately, it is possible to preempt (take away) the memory from *B* by swapping it out and swapping *A* in. Now *A* can run, do its printing, and then release the printer. No deadlock occurs.<br>A **nonpreemptable resource**, in contrast, is one that cannot be taken away from its current owner without potentially causing failure. If a process has begun to burn a Blu-ray, suddenly taking the Blu-ray recorder away from it and giving it to another process will result in a garbled Blu-ray. Blu-ray recorders are not preemptable at an arbitrary moment.<br>Whether a resource is preemptible depends on the context. On a standard PC, memory is preemptible because pages can always be swapped out to disk to recover it. However, on a smartphone that does not support swapping or paging, deadlocks cannot be avoided by just swapping out a memory hog.<br>In general, deadlocks involve nonpreemptable resources. Potential deadlocks that involve preemptable resources can usually be resolved by reallocating resources from one process to another. Thus, our treatment will focus on nonpreemptable resources.<br>The abstract sequence of events required to use a resource is given below.<br>1. Request the resource.<br>2. Use the resource.<br>3. Release the resource.<br>If the resource is not available when it is requested, the requesting process is forced to wait. In some operating systems, the process is automatically blocked when a resource request fails, and awakened when it becomes available. In other systems, the request fails with an error code, and it is up to the calling process to wait a little while and try again.<br>A process whose resource request has just been denied will normally sit in a tight loop requesting the resource, then sleeping, then trying again. Although this process is not blocked, for all intents and purposes it is as good as blocked, because it cannot do any useful work. In our further treatment, we will assume that when a process is denied a resource request, it is put to sleep. |
| e. | List Coffman's four conditions that must hold for to be a resource deadlock.<br>Coffman et al. (1971) showed that four conditions must hold for there to be a (resource) deadlock:<br>1. Mutual exclusion condition. Each resource is either currently assigned to exactly one process or is available.<br>2. Hold-and-wait condition. Processes currently holding resources that were granted earlier can request new resources.<br>3. No-preemption condition. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.<br>4. Circular wait condition. There must be a circular list of two or more processes, each of which is waiting for a resource held by the next member of the chain.<br>All four of these conditions must be present for a resource deadlock to occur. If one of them is absent, no resource deadlock is possible.<br>It is worth noting that each condition relates to a policy that a system can have or not have. Can a given resource be assigned to more than one process at once? Can a process hold a resource and ask for another? Can resources be preempted? Can circular waits exist? Later on we will see how deadlocks can be attacked by trying to negate some of these conditions. |
| f. | Explain the process of Deadlock Detection with One Resource of Each Type.<br>As an example of a system, consider a system with seven processes, *A* though *G*, and six resources, *R* through *W*. The state of which resources are currently owned and which ones are currently being requested is as follows:<br>1. Process *A* holds *R* and wants *S*.<br>2. Process *B* holds nothing but wants *T*.<br>3. Process *C* holds nothing but wants *S*.<br>4. Process *D* holds *U* and wants *S* and *T*.<br>5. Process *E* holds *T* and wants *V*.<br>6. Process *F* holds *W* and wants *S*.<br>7. Process *G* holds *V* and wants *U*. |

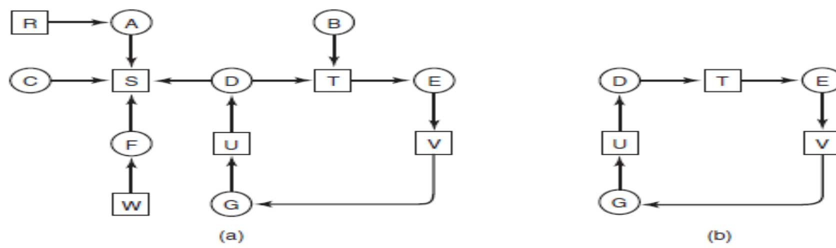



**Figure 6-5.** (a) A resource graph. (b) A cycle extracted from (a).

uses one dynamic data structure, *L*, a list of nodes, as well as a list of arcs. During the algorithm, to prevent repeated inspections, arcs will be marked to indicate that they have already been inspected.

The algorithm operates by carrying out the following steps as specified:

1. For each node, *N*, in the graph, perform the following five steps with *N* as the starting node.
2. Initialize *L* to the empty list, and designate all the arcs as unmarked.
3. Add the current node to the end of *L* and check to see if the node now appears in *L* two times. If it does, the graph contains a cycle (listed in *L*) and the algorithm terminates.
4. From the given node, see if there are any unmarked outgoing arcs. If so, go to step 5; if not, go to step 6.
5. Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and go to step 3.
6. If this node is the initial node, the graph does not contain any cycles and the algorithm terminates. Otherwise, we have now reached a dead end. Remove it and go back to the previous node, that is, the one that was current just before this one, make that one the current node, and go to step 3.

**4.** **Attempt *any three* of the following:**

a Write a note on Type-1 and Type-2 Hypervisor.

**type 1 hypervisor** is like an operating system, since it is the only program running in the most privileged mode. Its job is to support multiple copies of the actual hardware, called **virtual machines**, similar to the processes a normal operating system runs.
In contrast, a **type 2 hypervisor** is a different kind. It is a program that relies on, say, Windows or Linux to allocate and schedule resources, very much like a regular process. Of course, the type 2 hypervisor still pretends to be a full computer with a CPU and various devices. Both types of hypervisor must execute the machine's instruction set in a safe manner. For instance, an operating system running on top of the hypervisor may change and even mess up its own page tables, but not those of others. The operating system running on top of the hypervisor in both cases is called the **guest operating system**. For a type 2 hypervisor, the operating system running on the hardware is called the **host operating system**. The first type 2 hypervisor on the x86 market was **VMware Workstation**
Type 2 hypervisors, sometimes referred to as **hosted hypervisors**, depend for much of their functionality on a host operating system such as Windows, Linux, or OS X. When it starts for the first time, it acts like a newly booted computer and expects to find a DVD, USB drive, or CD-ROM containing an operating system in the drive. This time, however, the drive could be a virtual device. For instance, it is possible to store the image as an ISO file on the hard drive of the host and have the hypervisor pretend it is reading from a proper DVD drive. It then installs the operating system to its **virtual disk** (again really just a Windows, Linux, or OS X file) by running the installation program found on the DVD. Once the guest operating system is installed on the virtual disk, it can be booted and run..

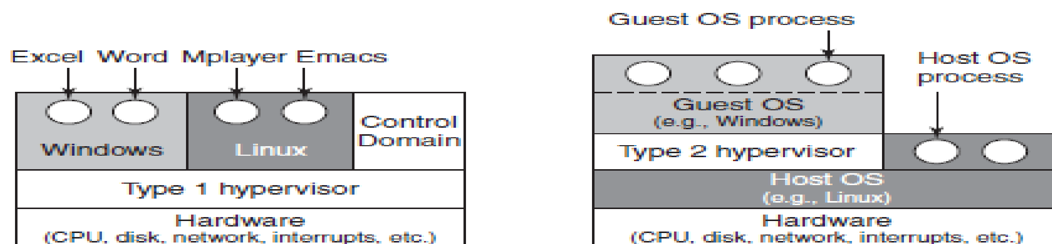



**Figure 7-1.** Location of type 1 and type 2 hypervisors.

b Explain any five advantages of virtualization.

1. a failure in one virtual machine does not bring down any others. On a virtualized system, different servers can run on different virtual machines, thus maintaining the

partial-failure model that a multicomputer has, but at a lower cost and with easier maintainability. Moreover, we can now run multiple different operating systems on the same hardware, benefit

| | |
|---|---|
| | from virtual machine isolation in the face of attacks, and enjoy other good stuff.<br><br>2. Other is that having fewer physical machines saves money on hardware and electricity and takes up less rack space. For a company such as Amazon or Microsoft, which may have hundreds of thousands of servers doing a huge variety of different tasks at each data center, reducing the physical demands on their data centers represents a huge cost savings. In fact, server companies frequently locate their data centers in the middle of nowhere—just to be close to, say, hydroelectric dams (and cheap energy).<br><br>3. Virtualization also helps in trying out new ideas. Typically, in large companies, individual departments or groups think of an interesting idea and then go out and buy a server to implement it. If the idea catches on and hundreds or thousands of servers are needed, the corporate data center expands. It is often hard to move the software to existing machines because each application often needs a different version of the operating system, its own libraries, configuration files, and more. With virtual machines, each application can take its own environment with it.<br><br>4. Another advantage of virtual machines is that checkpointing and migrating virtual machines (e.g., for load balancing across multiple servers) is much easier than migrating processes running on a normal operating system. In the latter case, a fair amount of critical state information about every process is kept in operating system tables, including information relating to open files, alarms, signal handlers, and more.<br><br>5. run legacy applications on operating systems (or operating system versions) no longer supported or which do not work on current hardware. These can run at the same time and on the same hardware as current applications.<br><br>6. Yet another important advantage of virtual machines is for software development. |
| c | List and explain five essential characteristics of Cloud.<br>lists five essential characteristics:<br>1. **On-demand self-service.** Users should be able to provision resources automatically, without requiring human interaction.<br>2. **Broad network access.** All these resources should be available over the network via standard mechanisms so that heterogeneous devices can make use of them.<br>3. **Resource pooling.** The computing resource owned by the provider should be pooled to serve multiple users and with the ability to assign and reassign resources dynamically. The users generally do not even know the exact location of ‘‘their’’ resources or even which country they are located in.<br>4. **Rapid elasticity.** It should be possible to acquire and release resources elastically, perhaps even automatically, to scale immediately with the users’ demands.<br>5. **Measured service.** The cloud provider meters the resources used in a way that matches the type of service agreed upon. |
| d | Write a note on Virtual Machine Migration.<br>Virtualization technology not only allows IAAS clouds to run multiple different operating systems on the same hardware at the same time, it also permits clever management. We hav e already discussed the ability to overcommit resources, especially in combination with deduplication. Now we will look at another management issue: what if a machine needs servicing (or even replacement) while it is running lots of important machines? Probably, clients will not be happy if their systems go down because the cloud provider wants to replace a disk drive. Hypervisors decouple the virtual machine from the physical hardware. In other words, it does not really matter to the virtual machine if it runs on this machine or that machine. Thus, the administrator could simply shut down all the virtual machines and restart them again on a shiny new machine.<br>A slightly better approach might be to pause the virtual machine, rather than shut it down. During the pause, we copy over the memory pages used by the virtual machine to the new hardware as quickly as possible, configure things correctly in the new hypervisor and then resume execution. Besides memory, we also need to transfer storage and network connectivity, but if the machines are close, this can be relatively fast. We could make the file system network-based to begin with (like NFS, the network file system), so that it does not matter whether your virtual machine is running on hardware in server rack 1 or 3. Likewise, the IP address can simply be switched to the new location. Nevertheless, we still need to pause the machine for a noticeable amount of time. Less time perhaps, but still noticeable.<br>Instead, what modern virtualization solutions offer is something known as **live migration**. In other words, they move the virtual machine while it is still operational. For instance, they employ techniques like **pre-copy memory migration**. |

This means that they copy memory pages while the machine is still serving requests. Most memory pages are not written much, so copying them over is safe. Remember, the virtual machine is still running, so a page may be modified after it has already been copied. When memory pages are modified, we have to make sure that the latest version is copied to the destination, so we mark them as dirty. They will be recopied later. When most memory pages have been copied, we are left with a small number of dirty pages. We now pause very briefly to copy the remaining pages and resume the virtual machine at the new location. While there is still a pause, it is so brief that applications typically are not affected. When the downtime is not noticeable, it is known as a **seamless live migration**.

e What is Master-Slave Multiprocessors Operating System?

**Master-Slave Multiprocessors**

A second model is shown in Fig. 8-8. Here, one copy of the operating system and its tables is present on CPU 1 and not on any of the others. All system calls are redirected to CPU 1 for processing there. CPU 1 may also run user processes if there is CPU time left over. This model is called **master-slave** since CPU 1 is the master and all the others are slaves.

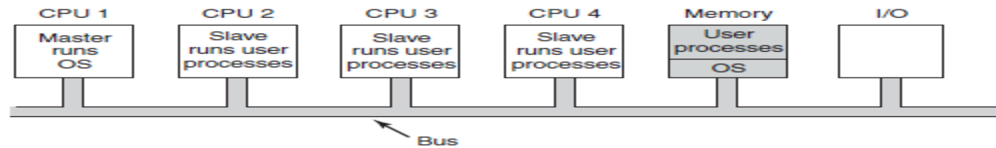



**Figure 8-8.** A master-slave multiprocessor model.

The master-slave model solves most of the problems of the first model. There is a single data structure (e.g., one list or a set of prioritized lists) that keeps track of ready processes. When a CPU goes idle, it asks the operating system on CPU 1 for a process to run and is assigned one. Thus it can never happen that one CPU is idle while another is overloaded. Similarly, pages can be allocated among all the processes dynamically and there is only one buffer cache, so inconsistencies never occur.

The problem with this model is that with many CPUs, the master will become a bottleneck. After all, it must handle all system calls from all CPUs. If, say, 10% of all time is spent handling system calls, then 10 CPUs will pretty much saturate the master, and with 20 CPUs it will be completely overloaded. Thus this model is simple and workable for small multiprocessors, but for large ones it fails.

f List the different Multicomputer Interconnection Technologies. Explain any two.

**Interconnection Technology**

Each node has a network interface card with one or two cables (or fibers) coming out of it. These cables connect either to other nodes or to switches. In a small system, there may be one switch to which all the nodes are connected in the star topology of Fig. 8-16(a). Modern switched Ethernets use this topology.

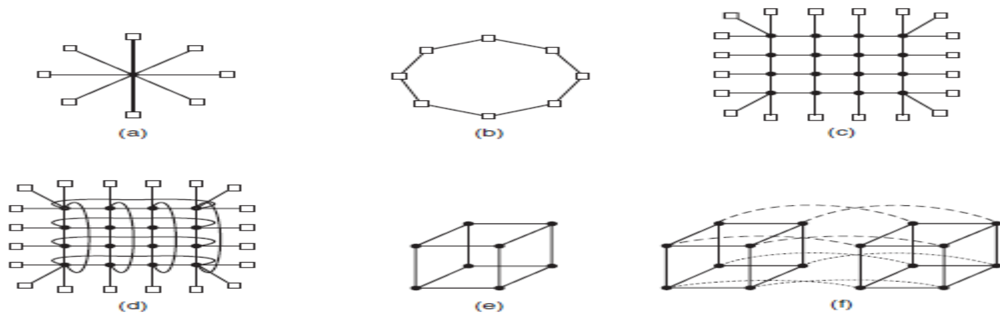



**Figure 8-16.** Various interconnect topologies. (a) A single switch. (b) A ring. (c) A grid. (d) A double torus. (e) A cube. (f) A 4D hypercube.

**5. Attempt *any three* of the following:**

a. Explain the kernel structure of Linux.

The kernel sits directly on the hardware and enables interactions with I/O devices and the memory management unit and controls CPU access to them. At the lowest level, as shown in Fig. 10-3 it contains interrupt handlers, which are the primary way for interacting with devices, and the low-level dispatching mechanism. This dispatching occurs when an interrupt happens. The low-level code here stops the running process, saves its state in the kernel process structures, and starts the appropriate driver. Process dispatching also happens when the kernel completes some operations and it is time to start up a user process again. The dispatching code is in assembler and is quite distinct from scheduling. Next, we divide the various kernel subsystems into three main components. The I/O component in Fig. 10-3 contains all kernel pieces responsible for interacting with devices and performing network and storage I/O operations. At the highest level, the I/O operations are all integrated under a **VFS** (**Virtual File System**) layer. That is, at the top level, performing a read operation on a file, whether it is in memory or on disk, is the same as performing a read operation to retrieve a character from a terminal input. At the lowest level, all I/O operations pass through some device driver.

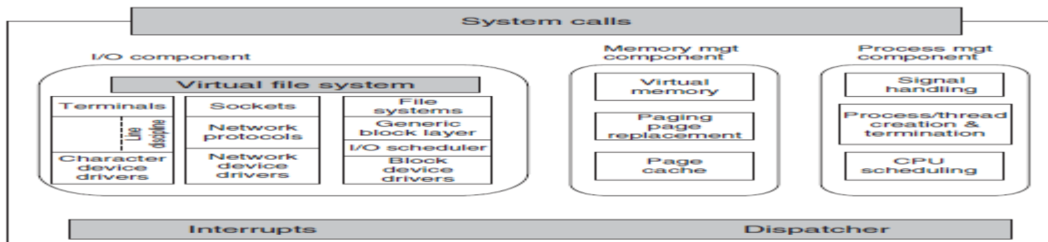



**Figure 10-3.** Structure of the Linux kernel

b. List and explain any five file-system related system calls in Linux.

| System call | Description |
|---|---|
| fd = creat(name, mode) | One way to create a new file |
| fd = open(file, how, ...) | Open a file for reading, writing, or both |
| s = close(fd) | Close an open file |
| n = read(fd, buffer, nbytes) | Read data from a file into a buffer |
| n = write(fd, buffer, nbytes) | Write data from a buffer into a file |
| position = lseek(fd, offset, whence) | Move the file pointer |
| s = stat(name, &buf) | Get a file's status information |
| s = fstat(fd, &buf) | Get a file's status information |
| s = pipe(&fd[0]) | Create a pipe |
| s = fcntl(fd, cmd, ...) | File locking and other operations |

c. Using suitable diagram explain the architecture of Android Operating System.

### 10.8.4 Android Architecture

Android is built on top of the standard Linux kernel, with only a few significant extensions to the kernel itself that will be discussed later. Once in user space, however, its implementation is quite different from a traditional Linux distribution and uses many of the Linux features you already understand in very different ways.

As in a traditional Linux system, Android's first user-space process is *init*, which is the root of all other processes. The daemons Android's *init* process starts are different, however, focused more on low-level details (managing file systems and hardware access) rather than higher-level user facilities like scheduling cron jobs. Android also has an additional layer of processes, those running Dalvik's Java language environment, which are responsible for executing all parts of the system implemented in Java.

Figure 10-39 illustrates the basic process structure of Android. First is the *init* process, which spawns a number of low-level daemon processes. One of these is *zygote*, which is the root of the higher-level Java language processes.

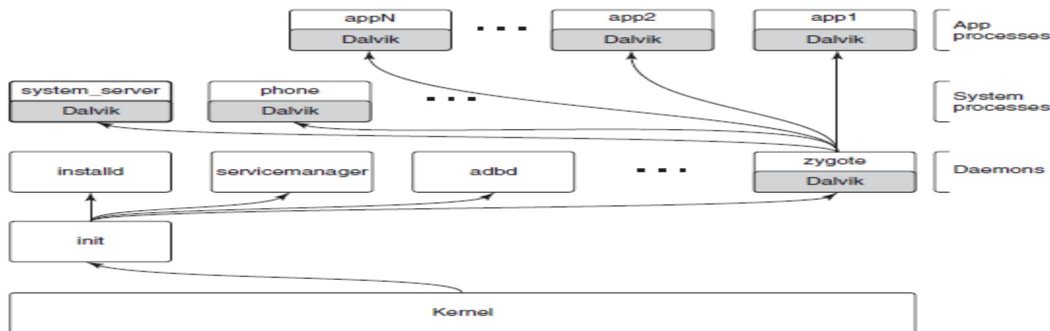



**Figure 10-39.** Android process hierarchy.

d. Explain the programming layers in modern windows operating System.

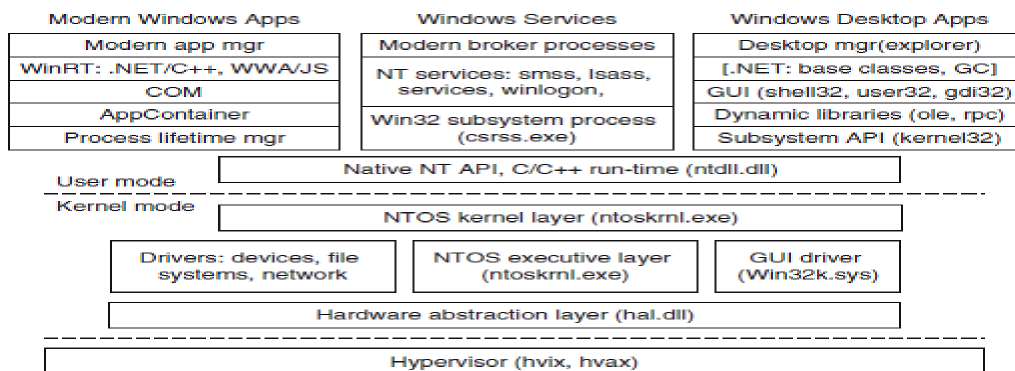



**Figure 11-4.** The programming layers in Modern Windows.

e. Explain the booting process of windows OS.

When a computer is
turned on, the first processor is initialized by the hardware, and then set to start executing a program in memory. The only available code is in some form of nonvolatile CMOS memory that is initialized by the computer manufacturer (and sometimes updated by the user, in a process called **flashing**). Because the software persists in memory, and is only rarely updated, it is referred to as **firmware**. The firmware is loaded on PCs by the manufacturer of either the parentboard or the computer system. Historically PC firmware was a program called BIOS (Basic Input/Output System), but most new computers use **UEFI** (**Unified Extensible Firmware Interface**). UEFI improves over BIOS by supporting modern hardware, providing a more modular CPU-independent architecture, and supporting

| | |
|---|---|
| | an extension model which simplifies booting over networks, provisioning new machines, and running diagnostics.<br>The main purpose of any firmware is to bring up the operating system by first loading small bootstrap programs found at the beginning of the disk-drive partitions. The Windows bootstrap programs know how to read enough information off a file-system volume or network to find the stand-alone Windows *BootMgr* program. *BootMgr* determines if the system had previously been hibernated or was in stand-by mode (special power-saving modes that allow the system to turn back on without restarting from the beginning of the bootstrap process). If so, *BootMgr* loads and executes *WinResume.exe*. Otherwise it loads and executes *WinLoad.exe* to perform a fresh boot. *WinLoad* loads the boot components of the system into memory: the kernel/executive (normally *ntoskrnl.exe*), the HAL (*hal.dll*), the file containing the SYSTEM hive, the *Win32k.sys* driver containing the kernel-mode parts of the Win32 subsystem, as well as images of any other drivers that are listed in the SYSTEM hive as **boot drivers**—meaning they are needed when the system first boots. If the system has Hyper-V enabled, WinLoad also loads and starts the hypervisor program.<br>Once the Windows boot components have been loaded into memory, control is handed over to the low-level code in NTOS which proceeds to initialize the HAL, kernel, and executive layers, link in the driver images, and access/update configuration data in the SYSTEM hive. After all the kernel-mode components are initialized, the first user-mode process is created using for running the *smss.exe* program. |
| f. | Briefly explain windows power management.<br>The **power manager** rides herd on power usage throughout the system. Historically management of power consumption consisted of shutting off the monitor display and stopping the disk drives from spinning. But the issue is rapidly becoming more complicated due to requirements for extending how long notebooks can run on batteries, and energy-conservation concerns related to desktop computers being left on all the time and the high cost of supplying power to the huge server farms that exist today. Newer power-management facilities include reducing the power consumption of components when the system is not in use by switching individual devices to standby states, or even powering them off completely using *soft* power switches. Multiprocessors shut down individual CPUs when they are not needed, and even the clock rates of the running CPUs can be adjusted downward to reduce power consumption. When a processor is idle, its power consumption is also reduced since it needs to do nothing except wait for an interrupt to occur. Windows supports a special shut down mode called **hibernation**, which copies all of physical memory to disk and then reduces power consumption to a small trickle (notebooks can run weeks in a hibernated state) with little battery drain.<br>Because all the memory state is written to disk, you can even replace the battery on a notebook while it is hibernated. When the system resumes after hibernation it restores the saved memory state (and reinitializes the I/O devices). This brings the computer back into the same state it was before hibernation, without having to login again and start up all the applications and services that were running.<br>An alternative to hibernation is **standby mode** where the power manager reduces the entire system to the lowest power state possible, using just enough power to the refresh the dynamic RAM. Because memory does not need to be copied to disk, this is somewhat faster than hibernation on some systems. |
| | |