(2½ Hours) **[Total Marks: 75]**

N. B.: (1) **All** questions are **compulsory**.
(2) Make **suitable assumptions** wherever necessary and **state the assumptions** made.
(3) Answers to the **same question** must be **written together**.
(4) Numbers to the **right** indicate **marks**.
(5) Draw **neat labeled diagrams** wherever **necessary**.
(6) Use of **Non-programmable** calculators is **allowed**.

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

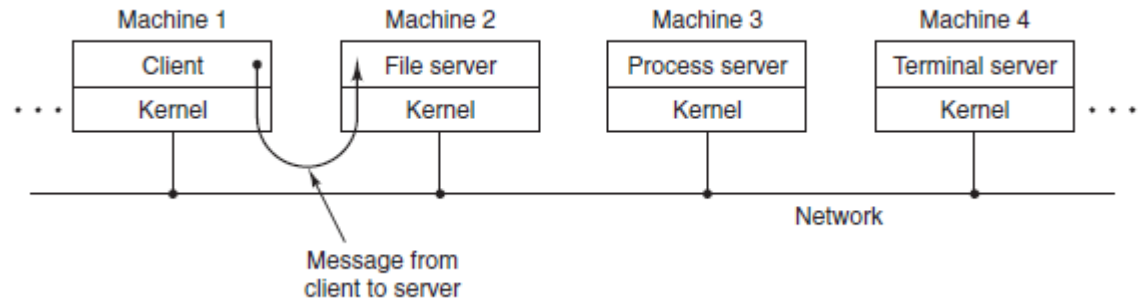
a. Explain third generation operating systems.
Explain 5M
.

b. List and explain system calls for process management.
Listing 1 Mark
Explanation of four system calls 1 Mark each

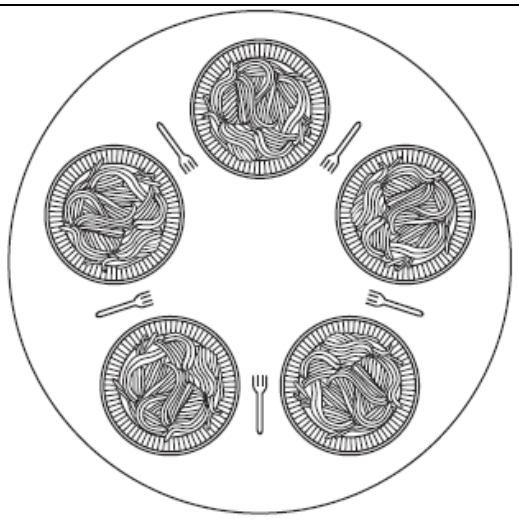
Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

c. Explain client-server model
Explanation 4 Marks
Diagram 1 Mark
A slight variation of the microkernel idea is to distinguish two classes of processes, the **servers**, each of which provides some service, and the **clients**, which use these services. This model is known as the **client-server** model. Often Communication between clients and servers is often by message passing. To obtain a service, a client process constructs a message saying what it wants and
sends it to the appropriate service. The service then does the work and sends back the answer. If the client and server happen to run on the same machine, certain optimizations are possible, but conceptually, we are still talking about message passing here.



d. Explain the dining philosophers problem.
Explanation 5 Marks
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



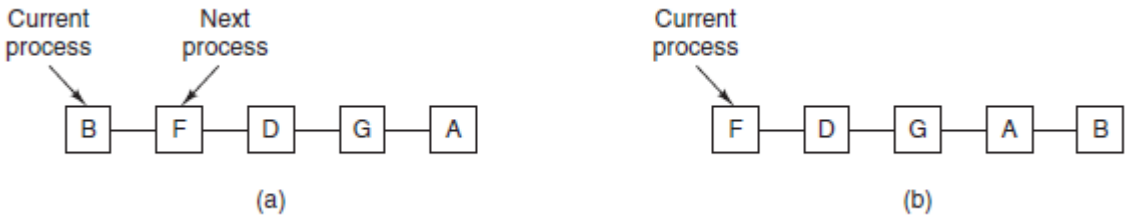
The life of a philosopher consists of alternating periods of eating and thinking. (This is something of an abstraction, even for philosophers, but the other activities are irrelevant here.) When a philosopher gets sufficiently hungry, she tries to acquire her left and right forks, one at a time, in either order. If successful in acquiring two forks, she eats for a while, then puts down the forks, and continues to think.

e. Explain round robin scheduling. Give proper example.

Explanation 4 Marks

Example 1 Mark

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.(a). When the process uses up its quantum, it is put on the end of the list, as shown in Fig. (b). 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, updating various tables and lists, flushing and reloading the memory cache, and so on. Suppose that this **process switch** o r **context switch**, as it is sometimes called, takes 1msec, including switching memory maps, flushing and reloading the cache, etc. Also suppose that the quantum is set at 4 msec. With these parameters, after doing 4 msec of useful work, the CPU will have to spend (i.e., waste) 1 msec on process switching. Thus 20% of the CPU time will be thrown away on administrative overhead.

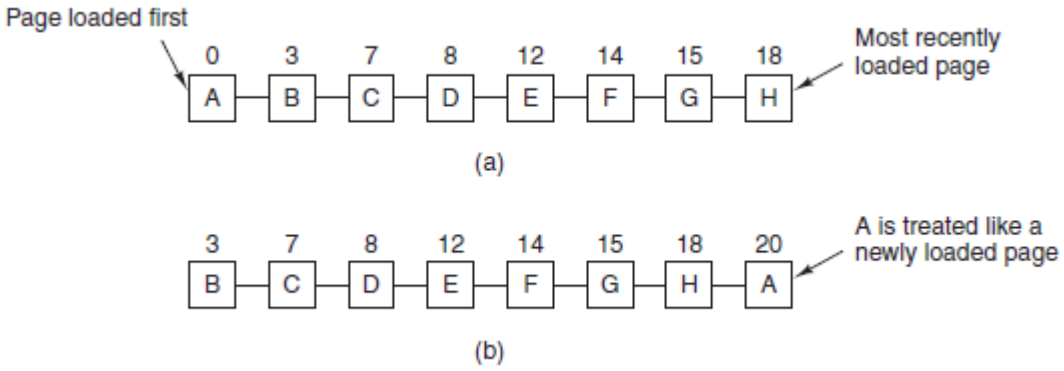


f. Write a short note on semaphores.

Explanation 5 Marks

This was the situation in 1965, when E. W. Dijkstra (1965) suggested using an integer variable to count the number of wakeups saved for future use. In his proposal, a new variable type, which he called a **semaphore**, was introduced. A semaphore could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were

	pending. Dijkstra proposed having two operations on semaphores, now usually called down and up (generalizations of sleep and wakeup, respectively). 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 ever blocks doing a wakeup in the earlier model.	
2.	**Attempt *<u>any three</u>* of the following:**	**15**
a.	Explain with example the second chance page replacement algorithm. Explanation 5 Marks 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**, is shown in figure. In Fig. (a) we see pages *A* through *H* kept on a linked list and sorted by the time they arrived in memory. Suppose that a page fault occurs at time 20. The oldest page is *A*, which arrived at time 0, when the process started. If *A* has the *R* bit cleared, it is evicted from memory, either by being written to the disk (if it is dirty), or just abandoned (if it is clean). On the other hand, if the *R* bit is set, *A* is put onto the end of the list and its ‘‘load time’’ is reset to the current time (20). The *R* bit is also cleared. The search for a suitable page continues with *B*. What second chance is looking for is an old page that has not been referenced in the most recent clock interval. If all the pages have been referenced, second chance degenerates into pure FIFO. Specifically, imagine that all the pages in Fig.(a) have their *R* bits set. One by one, the operating system moves the pages to the end of the list, clearing the *R* bit each time it appends a page to the end of the list. Eventually, it comes back to page *A*, which now has its *R* bit cleared. At this point *A* is evicted. Thus the algorithm always terminates.	



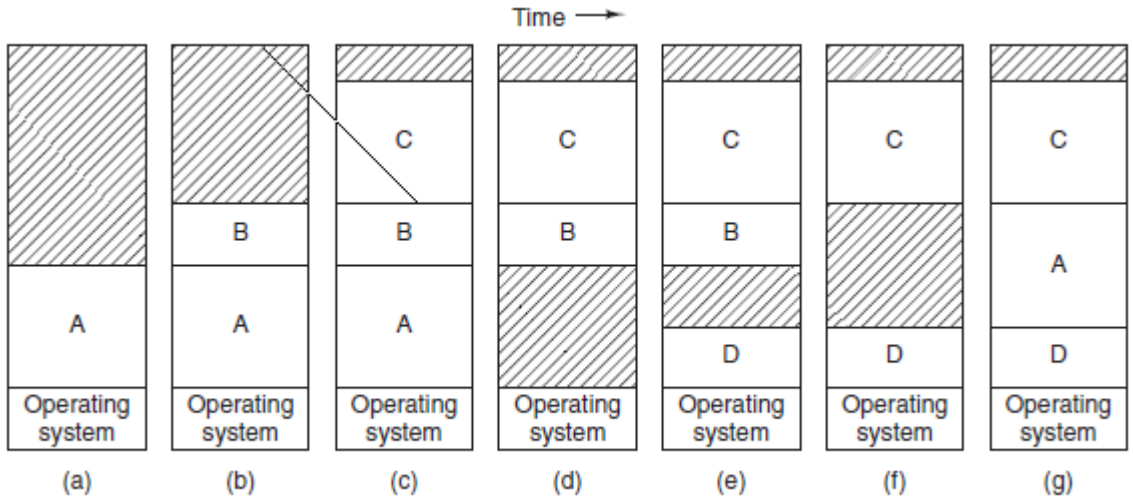
b. Explain swapping.

Explanation 4 Marks

Diagram 1 Mark

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.

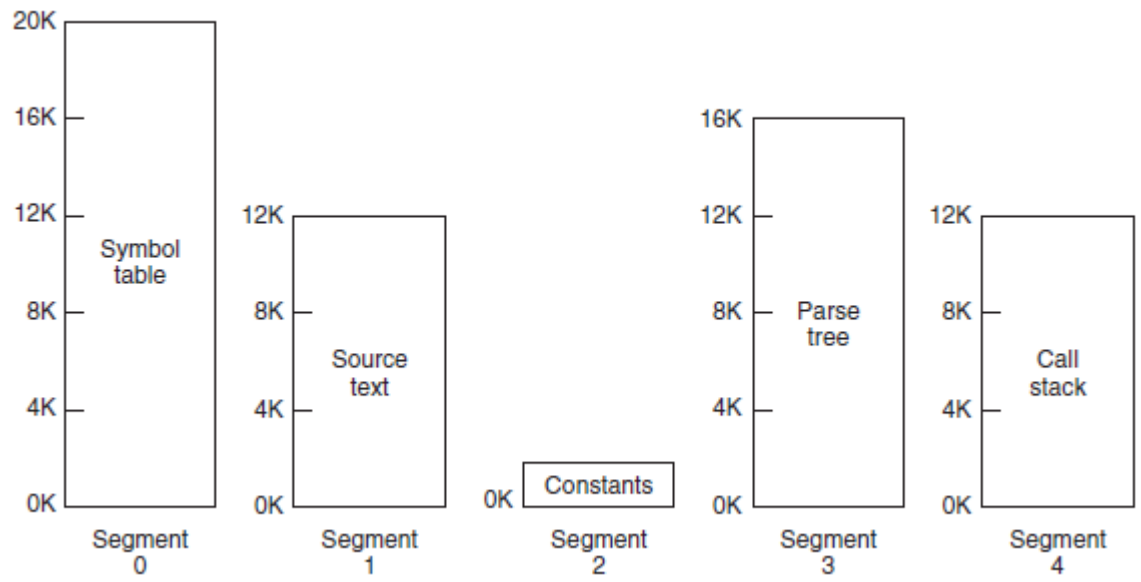
The operation of a swapping system is illustrated in Figure Initially, only process *A* is in memory. Then processes *B* and *C* are created or swapped in from disk. In Fig.(d) *A* is swapped out to disk. Then *D* comes in and *B* goes out. Finally *A* comes in again. Since *A* is now at a different 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.



c. Write a short note on segmentation.

Explanation 5 Marks

A straightforward and quite general solution is to provide the machine with many completely independent address spaces, which are called **segments**. Each segment consists of a linear sequence of addresses, starting at 0 and going up to some maximum value. The length of each segment may be anything from 0 to the maximum address allowed. Different segments may, and usually do, have different lengths. Moreover, segment lengths may change during execution. The length of a

	stack segment may be increased whenever something is pushed onto the stack and decreased whenever something is popped off the stack. Because each segment constitutes a separate address space, different segmentscan grow or shrink independently without affecting each other. If a stack in a certain segment needs more address space to grow, it can have it, because there is nothing else in its address space to bump into. Of course, a segment can fill up, but segments are usually very large, so this occurrence is rare. To specify an address in this segmented or two-dimensional memory, the program must supply a two-part address, a segment number, and an address within the segment. Figure illustrates a segmented memory being used for the compiler tables discussed earlier. Five independent segments are shown.  	
d.	List and explain different file attributes. Listing 1 Mark Explanation of any four 1 Mark each	

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

e. List various ways of implementing files. Explain any one in detail.

Listing 1 Mark

Explanation of any one 4 Marks

various ways of implementing files are

1. Contiguous Allocation

2. Linked-List Allocation

3. I-nodes

f. Explain disk quotas.

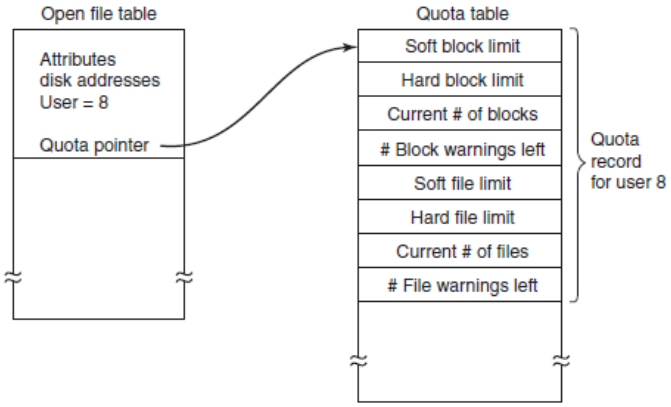
Explanation 4 Marks

Diagram 1 Mark

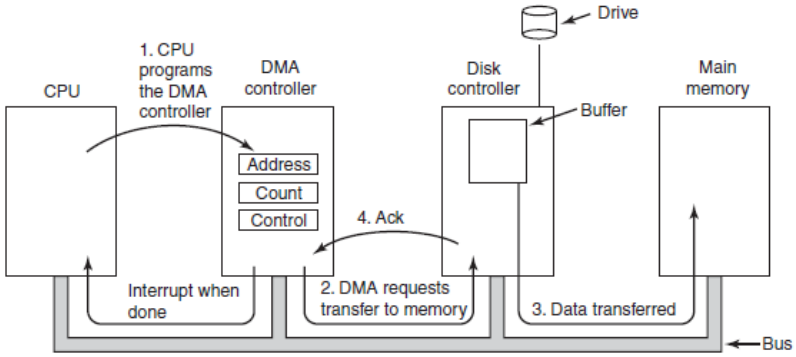
To prevent people from hogging too much disk space, multiuser operating systems often provide a mechanism for enforcing disk quotas. The idea is that the system administrator assigns each user a maximum allotment of files and blocks, and the operating system makes sure that the users do not exceed their quotas. When a user opens a file, the attributes and disk addresses are located and put into an open-file table in main memory. Among the attributes is an entry telling who the owner is. Any increases in the file's size will be charged to the owner's quota.

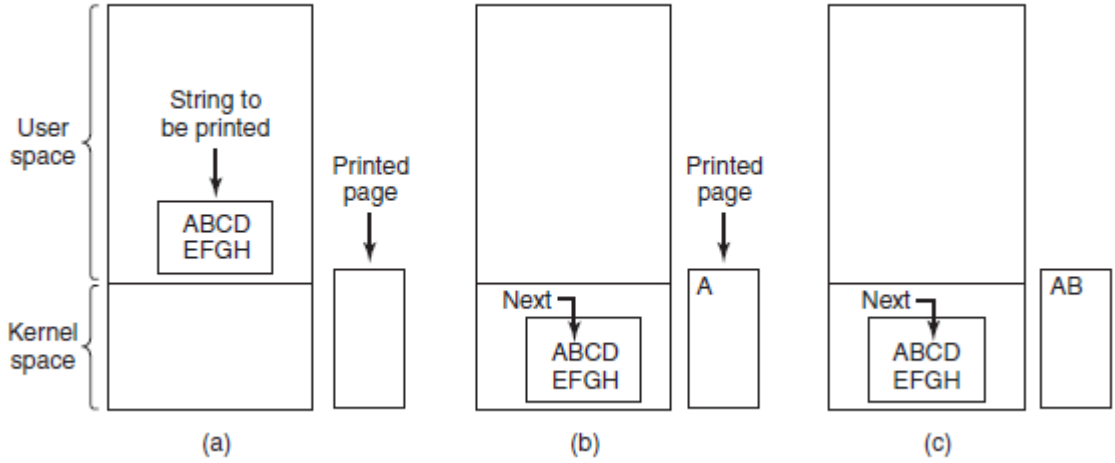
A second table contains the quota record for every user with a currently open file, even if the file was opened by someone else. This table is shown in Fig. It is an extract from a quota file on disk for the users whose files are currently open. When all the files are closed, the record is written back to the quota file. When a new entry is made in the open-file table, a pointer to the owner's quota record is entered into it, to make it easy to find the various limits. Every time a block is added to a file, the total number of blocks charged to the owner is incremented, and a check is made against both the hard and soft limits. The soft limit may be

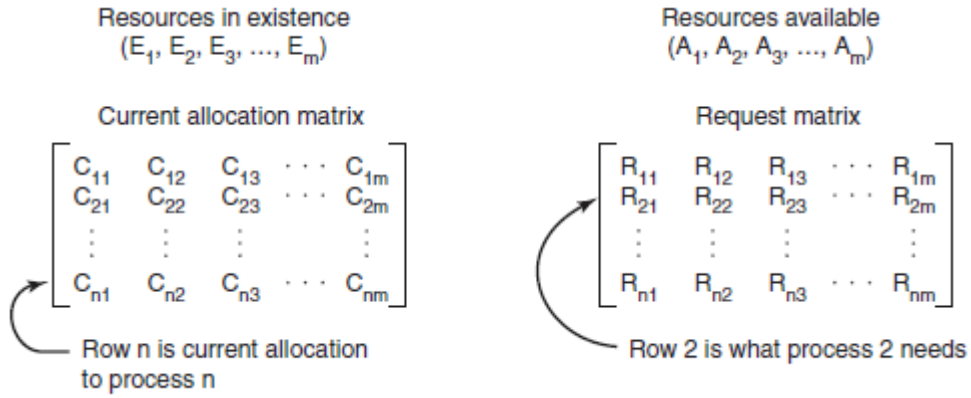
exceeded, but the hard limit may not. An attempt to append to a file when the hard block limit has been reached will result in an error. Analogous checks also exist for the number of files to prevent a user from hogging all the i-nodes. When a user attempts to log in, the system examines the quota file to see if the user has exceeded the soft limit for either number of files or number of disk blocks If either limit has been violated, a warning is displayed, and the count of warnings remaining is reduced by one. If the count ever gets to zero, the user has ignored the warning one time too many, and is not permitted to log in. Getting permission to log in again will require some discussion with the system administrator.



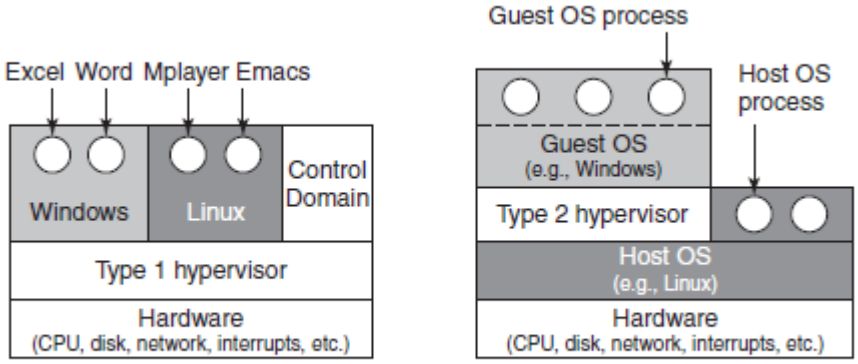
3.	**Attempt *any three* of the following:**	**15**
a.	Write a short note on direct memory access. Explanation 4 Marks Diagram 1 Mark The CPU can request data from an I/O controller one byte at a time, but doing so wastes the CPU's time, so a different scheme, called **DMA** (**Direct Memory Access**) is often used. let us first look at how disk reads occur when DMA is not used. First the disk controller reads the block (one or more sectors) from the drive serially, bit by bit, until the entire block is in the controller's internal buffer. Next, it computes the checksum to verify that no read errors have occurred. Then the controller causes an interrupt. When the operating system starts running, it can read the disk block from the controller's buffer a byte or a word at a time by executing a loop, with each iteration reading one byte or word from a controller device register and storing it in main memory. When DMA is used, the procedure is different. First the CPU programs the DMA controller by setting its registers so it knows what to transfer where (step 1 in Fig.). It also issues a command to the disk controller telling it to read data from the disk into its internal buffer and verify the checksum. When valid data are in the disk controller's buffer, DMA can begin. The DMA controller initiates the transfer by issuing a read request over the bus to the disk controller (step 2). This read request looks like any other read request, and the disk controller does not know (or care) whether it came from the CPU or from a DMA controller. Typically, the memory address to write to is on the bus' address lines, so when the disk controller fetches the next word from its internal buffer, it knows where to write it. The write to memory is another standard bus cycle (step 3). When the write is complete, the disk controller sends an acknowledgement signal to the DMA controller, also over the bus (step 4). The DMA controller then increments the memory address to use and decrements the byte count. If the byte count is still greater than 0, steps 2 through 4 are repeated until the count reaches 0. At that time, the DMA controller interrupts the CPU to let it know that the transfer is now complete. When the operating system starts up, it does not have to copy the	

	disk block to memory; it is already there.  	
b.	Explain programmed I/O. Explanation 5 Marks There are three fundamentally different ways that I/O can be performed. In this section we will look at the first one (programmed I/O). In the next two sections we will examine the others (interrupt-driven I/O and I/O using DMA). The simplest form of I/O is to have the CPU do all the work. This method is called **programmed I/O** Consider a user process that wants to print the eight-character string ‘‘ABCDEFGH’’ on the printer via a serial interface. Displays on small embedded systems sometimes work this way. The software first assembles the string in a buffer in user space, as shown in Fig. (a). The user process then acquires the printer for writing by making a system call to open it. If the printer is currently in use by another process, this call will fail and return an error code or will block until the printer is available, depending on the operating system and the parameters of the call. Once it has the printer, the user process makes a system call telling the operating system to print the string on the printer. The operating system then (usually) copies the buffer with the string to an array, say, *p*, in kernel space, where it is more easily accessed (because the kernel may have to change the memory map to get at user space). It then checks to see if the printer is currently available. If not, it waits until it is. As soon as the printer is available, the operating system copies the first character to the printer’s data register, in this example using memory-mapped I/O. This action activates the printer. The character may not appear yet because some printers buffer a line or a page before printing anything. In Fig. (b), however, we see that the first character has been printed and that the system has marked the ‘‘B’’ as the next character to be printed. As soon as it has copied the first character to the printer, the operating system checks to see if the printer is ready to accept another one. Generally, the printer has a second register, which gives its status. The act of writing to the data register causes the status to become not ready. When the printer controller has processed the current character, it indicates its availability by setting some bit in its status register or putting some value in it. At this point the operating system waits for the printer to become ready again. When that happens, it prints the next character, as shown in Fig. (c). This loop continues until the entire string has been printed. Then control returns to the user process.	

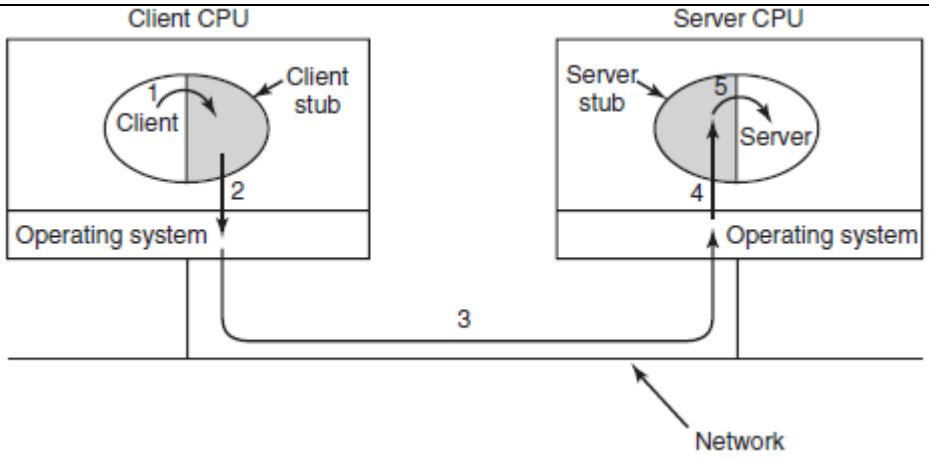
<table>
<tr><td></td><td> 
</td><td></td></tr>
<tr><td>c.</td><td>List different I/O software layers. Explain any two of them.
List 1 Mark
Explanation of any two 2 Marks each
Different I/O software layers are:
1. User-level I/O software
2. Device-independent operating system software
3. Device drivers
4. Interrupt handlers
5. Hardware</td><td></td></tr>
<tr><td>d.</td><td>What is deadlock? List and explain conditions that are necessary for a resource deadlock to occur.
Definition 1 Mark
List 1 Mark
Explanation 3 Marks
A set of processes is deadlocked if each process in the s et is waiting for an event that only another process in the set can cause.
Coffman et al. (1971) showed that four conditions must hold for there to be a (resource) deadlock:
1. Mutual exclusion condition. Each resource is either currently assigned to exactly one process or is available.
2. Hold-and-wait condition. Processes currently holding resources that were granted earlier can request new resources.
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.
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.</td><td></td></tr>
<tr><td>e.</td><td>Explain deadlock detection algorithm to detect deadlock when multiple resources of each type are available.
Explanation 5 Marks
When multiple copies of some of the resources exist, a different approach is needed to detect deadlocks. We will now present a matrix-based algorithm for detecting deadlock among n processes, P1 through Pn. Let the number of resource classes be m, with E1 resources of class 1, E2 resources of class 2, and generally, Ei resources of class i ($1 \leq i \leq m$). E is the existing resource vector. It gives the total number of instances of each resource in existence. For example, if class 1 is</td><td></td></tr>
</table>

	tape drives, then $E1 = 2$ means the system has two tape drives. At any instant, some of the resources are assigned and are not available. Let A be the **av ailable resource vector**, with Ai giving the number of instances of resource i that are currently available (i.e., unassigned). If both of our two tape drives are assigned, $A1$ will be 0. Now we need two arrays, C, the **current allocation matrix**, and R, the **request matrix**. The ith row of C tells how many instances of each resource class Pi currently holds. Thus, Cij is the number of instances of resource j that are held by process i. Similarly, Rij is the number of instances of resource j that Pi wants. These four data structures are shown in Fig.  An important invariant holds for these four data structures. In particular, every resource is either allocated or is available. This observation means that $\sum_{i=1}^{n} C_{ij} + A_j = E_j$ The deadlock detection algorithm is based on comparing vectors. Let us define the relation $A \leq B$ on two vectors A and B to mean that each element of A is less than or equal to the corresponding element of B. Mathematically, $A \leq B$ holds if and only if $Ai \leq Bi$ for $1 \leq i \leq m$. Each process is initially said to be unmarked. As the algorithm progresses, processes will be marked, indicating that they are able to complete and are thus not deadlocked. When the algorithm terminates, any unmarked processes are known to be deadlocked. This algorithm assumes a worst-case scenario: all processes keep all acquired resources until they exit. The deadlock detection algorithm can now be given as follows. 1. Look for an unmarked process, Pi , for which the ith row of R is less than or equal to A. 2. If such a process is found, add the ith row of C to A, mark the process, and go back to step 1. 3. If no such process exists, the algorithm terminates.	
f.	Explain how deadlocks are prevented? Explanation 5 Marks If we can ensure that at least one of these conditions is never satisfied, then deadlocks will be structurally impossible Attacking the Mutual-Exclusion Condition Attacking the Hold-and-Wait ConditionZ Attacking the No-Preemption Condition Attacking the Circular Wait Condition	
4.	**Attempt *any three* of the following:**	**15**
a.	Write the advantages of virtualization. Any five advantages 5 Marks 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	

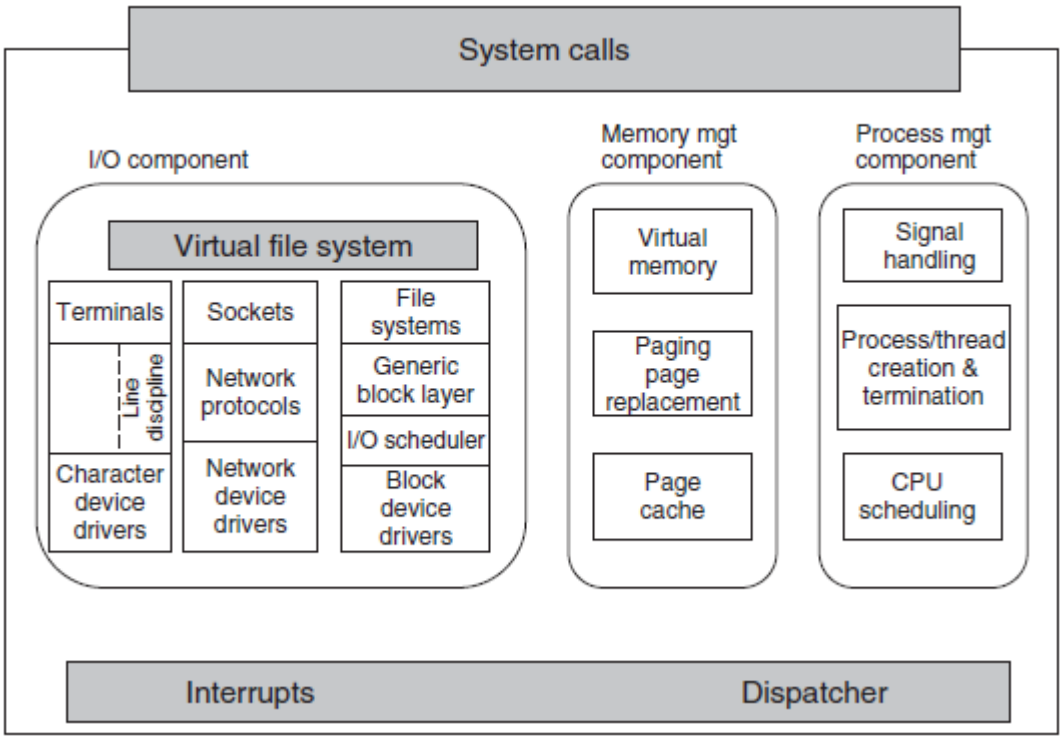
	partial-failure model that a multicomputer has, but at a lower cost and with easier maintainability. Moreover, we can now run multiple different operating systems on the same hardware, benefit from virtual machine isolation in the face of attacks, and enjoy other good stuff. 2. Other is that having fewer 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). 3. Virtualization also helps in trying out new ideas. Typically, in large companies, individual departments or groups think of aninteresting 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. 4. Another advantage of virtual machines is that checkpointing and migrating virtualmachines (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 systemtables, including information relating to open files, alarms, signal handlers, and more. 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. 6. Yet another important advantage of virtual machines is for software development.	
b.	With neat diagram explain type 1 and type 2 hypervisor. Explanation 4 Marks Diagrams 1 Mark Goldberg (1972) distinguished between two approaches to virtualization. One kind of hypervisor, dubbed a **type 1 hypervisor** is illustrated in Fig. 7-1(a). Technically, it 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**, shown in Fig. (b), is a different kind of animal. 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** (Bugnion et al., 2012). In this section, we introduce the general idea. 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.	
c.	What is cloud? Write the essential characteristics of cloud. Definition 1 Mark Any four characteristics 4 Marks The key idea of a cloud is straightforward: outsource your computation or storage needs to a well-managed data centerrun by a company specializing in this and staffed by experts in the area. Five essential characteristics: 1. **On-demand self-service.** Users should be able to provision resources automatically, without requiring human interaction. 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. 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. 4. **Rapid elasticity.** It should be possible to acquire and release resources elastically, perhaps even automatically, to scale immediately with the users’ demands. 5. **Measured service.** The cloud provider meters the resources used in a way that matches the type of service agreed upon	
d.	List different types of multiprocessor operating systems. Explain any two. List 1 Mark Explanation of any two 2 marks each List **Each CPU Has Its Own Operating System** **Master-Slave Multiprocessors** **Symmetric Multiprocessors**	
e.	With neat diagram explain various interconnection technologies used in multicomputer. Explanation 4 Marks Diagram 1 Mark **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.(a). Modern switched Ethernets use this topology.	

	As an alternative to the single-switch design, the nodes may form a ring, with two wires coming out the network interface card, one into the node on the left and one going into the node on the right, as shown in Fig. (b). In this topology, no switches are needed and none are shown. The **grid** or **mesh** of Fig. (c) is a two-dimensional design that has been used in many commercial systems. It is highly regular and easy to scale up to large sizes. It has a **diameter**, which is the longest path between any two nodes, and which increases only as the square root of the number of nodes. A variant on the grid is the **double torus** of Fig.(d), which is a grid with the edges connected. Not only is it more fault tolerant than the grid, but the diameter is also less because the opposite corners can now communicate in only two hops. The **cube** of Fig. (e) is a regular three-dimensional topology. We hav e illustrated a $2 \times 2 \times 2$ cube, but in the most general case it could be a $k \times k \times k$ cube. In Fig. (f) we have a four-dimensional cube built from two three-dimensional cubes with the corresponding nodes connected. We could make a five dimensional cube by cloning the structure of Fig. (f) and connecting the corresponding nodes to form a block of four cubes. To go to six dimensions, we could replicate the block of four cubes and interconnect the corresponding nodes, and so on. An *n*-dimensional cube formed this way is called a **hypercube**.	
f.	Write a short note on remote procedure call. Explanation 5 Marks Although the message-passing model provides a convenient way to structure a multicomputer operating system, it suffers from one incurable flaw: the basic paradigm around which all communication is built is input/output. The procedures *send* and *receive* are fundamentally engaged in doing I/O, and many people believe that I/O is the wrong programming model. The solution was allowing programs to call procedures located on other CPUs. When a process on machine 1 calls a procedure on machine 2, the calling process on 1 is suspended, and execution of the called procedure takes place on 2. Information can be transported from the called to the callee in the parameters and can come back in the procedure result. No message passing or I/O at all is visible to the programmer. This technique is known as **RPC** (**Remote Procedure Call**) In the simplest form, to call a remote procedure, the client program must be bound with a small library procedure called the **client stub** that represents the server procedure in the client's address space. Similarly, the server is bound with a procedure called the **server stub**. These procedures hide the fact that the procedure call from the client to the server is not local. The actual steps in making an RPC are shown in Fig. Step 1 is the client calling the client stub. This call is a local procedure call, with the parameters pushed onto the stack in the normal way. Step 2 is the client stub packing the parameters into a message and making a system call to send the message. Packing the parameters is called **marshalling**. Step 3 is the kernel sending the message from the client machine to the server machine. Step 4 is the kernel passing the incoming packet to the server stub (which would normally have called *receive* earlier). Finally, step 5 is the server stub calling the server procedure. The reply traces the same path in the other direction.	



5.	**Attempt *any three* of the following:**	**15**
a.	Explain the kernel structure of Linux. Diagram 1 Mark Explanation 4 Marks	

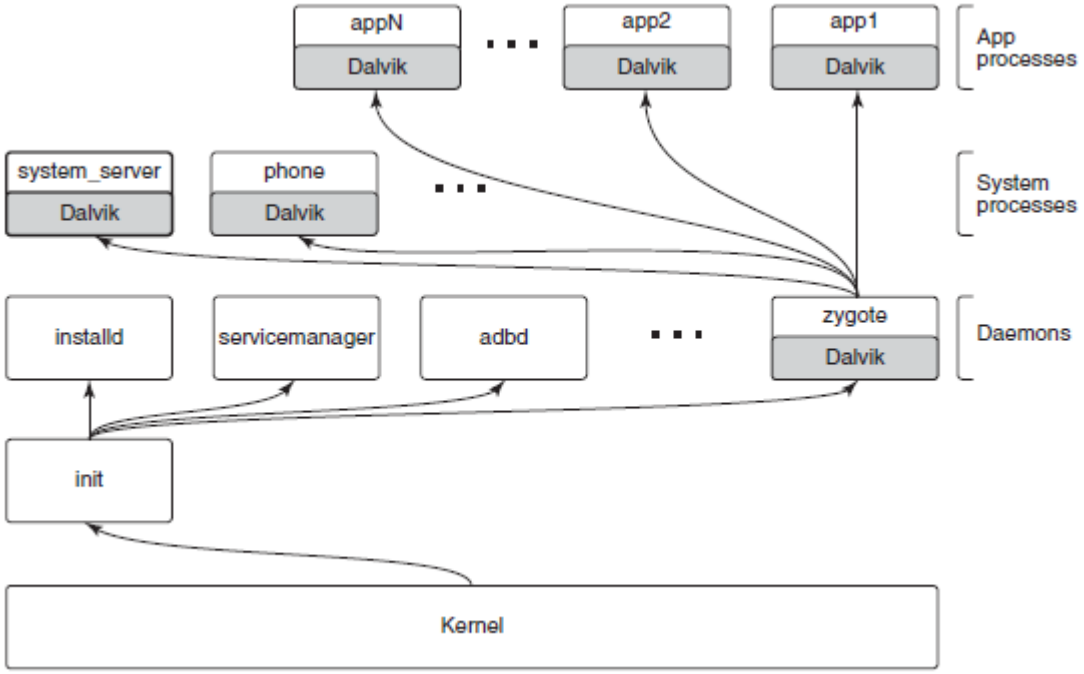


The kernel sits directly on the hardware and enables interactions with I/O devices and the memory management unit and controls CPU access to them. At the lowest level, as shown in Fig. it contains interrupt handlers, which are the primary way for interacting with devices, and the low-level dispatching 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. 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. All Linux drivers are classified as either character-device drivers or block-device drivers, the main difference being that seeks and random accesses are allowed on block devices and not on character devices. Technically, network devices are really character devices, but they are handled somewhat differently, so that it is probably clearer to separate them, as has been done in the figure. Above the device-driver lev el, the kernel code is different for each device type. Character devices may be used in two different ways. Some programs, such as visual editors like *vi* and *emacs*, want every keystroke as it is hit. Raw terminal (tty) I/O makes this possible. Other software, such as the shell, is line oriented, allowing users to edit the whole line before hitting ENTER to send it to the program. In this case the character stream from the terminal device is passed through a socalled line discipline, and appropriate formatting is applied. Networking software is often modular, with different devices and protocols supported. The layer above the network drivers handles a kind of routing function, making sure that the right packet goes to the right device or protocol handler. Most Linux systems contain the full functionality of a hardware router within the kernel, although the performance is less than that of a hardware router. Above the router code is the actual protocol stack, including IP and TCP, but also many additional protocols. Overlaying all the network is the socket interface, which allows programs to create sockets for particular networks and protocols, getting back a file descriptor for each socket to use later. On top of the disk drivers is the I/O scheduler, which is responsible for ordering and issuing disk-operation requests in a way that tries to conserve wasteful disk head movement or to meet some other system policy. At the very top of the block-device column are the file systems. Linux may, and in fact does, have multiple file systems coexisting concurrently. In order to hide the gruesome architectural differences of various hardware devices from the file system implementation, a generic block-device layer provides an abstraction used by all file systems. To the right in Fig. are the other two key components of the Linux kernel. These are responsible for the memory and process management tasks. Memory- management tasks include maintaining the virtual to physical-memory mappings, maintaining a cache of recently accessed pages and implementing a good page-replacement policy, and on-demand bringing in new pages of needed code and data into memory. The key responsibility of the process-management component is the creation and termination of processes. It also includes the process scheduler, which chooses which process or, rather, thread to run next. As we shall see in the next section, the Linux kernel treats both processes and threads simply as executable entities, and will schedule them based on a global scheduling policy. Finally, code for signal handling also belongs to this component.	
b.	Explain Android architecture. Explanation 5 Marks 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 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. Android's *init* does not run a shell in the traditional way, since a typical
Android device does not have a local console for shell access. Instead, the daemon process *adbd* listens for remote connections (such as over USB) that request shell access, forking shell processes for them as needed. Since most of Android is written in the Java language, the *zygote* daemon and
processes it starts are central to the system. The first process *zygote* always starts is called *system server*, which contains all of the core operating system services.
Key parts of this are the power manager, package manager, window manager, and activity manager.
Other processes will be created from *zygote* as needed. Some of these are ''persistent'' processes that are part of the basic operating system, such as the telephony stack in the phone process, which must remain always running. Additional application processes will be created and stopped as needed while the system is running.
Applications interact with the operating system through calls to libraries provided by it, which together compose the **Android framework**. Some of these libraries can perform their work within that process, but many will need to perform interprocess communication with other processes, often services in the *system server* process.
The package manager provides a framework API for applications to call in their local process, here the *PackageManager* class. Internally, this class must get a connection to the corresponding
service in the *system server*. To accomplish this, at boot time the *system server* publishes each service under a well-defined name in the *service manager*, a daemon started by *init*. The *PackageManager* in the application process retrieves a connection from the *service manager* to its system service using that same name.
Once the *PackageManager* has connected with its system service, it can make calls on it. Most application calls to *PackageManager* are implemented as interprocess communication using Android's *Binder* IPC mechanism, in this case making calls to the *PackageManagerService* implementation in the *system server*. The implementation of *PackageManagerService* arbitrates interactions across all client applications and maintains state that will be needed by multiple applications.



c. List and explain file-system system calls in Linux.

List 1 Mark

Explanation 4 Marks

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

d. Write down I/O and object manager steps for creating/opening a file and getting back a file handle.

1. When an executive component, such as the I/O manager implementing the native system call NtCreateFile, calls ObOpenObjectBy- Name in the object manager, it passes a Unicode path name for the NT namespace, say \ *??* \ *C:* \ *foo* \ *bar*.
2. The object manager searches through directories and symbolic links and ultimately finds that \ *??* \ *C:* refers to a device object (a type defined by the I/O manager). The device object is a leaf node in the part of the NT namespace that the object manager manages.
3. The object manager then calls the *Parse* procedure for this object type, which happens to be IopParseDevice implemented by the I/O manager. It passes not only a pointer to the device object it found (for *C:*), but also the remaining string \ *foo* \ *bar*.
4. The I/O manager will create an **IRP** (**I/O Request Packet**), allocate a file object, and send the request to the stack of I/O devices determined by the device object found by the object manager.

5. The IRP is passed down the I/O stack until it reaches a device object representing the file-system instance for *C:*. At each stage, control is passed to an entry point into the driver object associated with the device object at that level. The entry point used here is for CREATE operations, since the request is to create or open a file named \ *foo* \ *bar* on the volume.
6. The device objects encountered as the IRP heads toward the file system represent file-system filter drivers, which may modify the I/O operation before it reaches the file-system device object. Typically these intermediate devices represent system extensions like antivirus filters.
7. The file-system device object has a link to the file-system driver object, say NTFS. So, the driver object contains the address of the CREATE operation within NTFS.
8. NTFS will fill in the file object and return it to the I/O manager, which returns back up through all the devices on the stack until Iop- ParseDevice returns to the object manager.
9. The object manager is finished with its namespace lookup. It received back an initialized object from the *Parse* routine (which happens to be a file object—not the original device object it found). So the object manager creates a handle for the file object in the handle table of the current process, and returns the handle to its caller.
10. The final step is to return back to the user-mode caller, which in this example is the Win32 API CreateFile, which will return the handle to the application.

e. List Win32 calls for managing processes, threads and fibers.

Win32 API Function	Description
CreateProcess	Create a new process
CreateThread	Create a new thread in an existing process
CreateFiber	Create a new fiber
ExitProcess	Terminate current process and all its threads
ExitThread	Terminate this thread
ExitFiber	Terminate this fiber
SwitchToFiber	Run a different fiber on the current thread
SetPriorityClass	Set the priority class for a process
SetThreadPriority	Set the priority for one thread
CreateSemaphore	Create a new semaphore
CreateMutex	Create a new mutex
OpenSemaphore	Open an existing semaphore
OpenMutex	Open an existing mutex
WaitForSingleObject	Block on a single semaphore, mutex, etc.
WaitForMultipleObjects	Block on a set of objects whose handles are given
PulseEvent	Set an event to signaled, then to nonsignaled
ReleaseMutex	Release a mutex to allow another thread to acquire it
ReleaseSemaphore	Increase the semaphore count by 1
EnterCriticalSection	Acquire the lock on a critical section
LeaveCriticalSection	Release the lock on a critical section
WaitOnAddress	Block until the memory is changed at the specified address
WakeByAddressSingle	Wake the first thread that is waiting on this address
WakeByAddressAll	Wake all threads that are waiting on this address
InitOnceExecuteOnce	Ensure that an initialize routine executes only once

f. List and explain attributes used in MFT records.

Attribute	Description
Standard information	Flag bits, timestamps, etc.
File name	File name in Unicode; may be repeated for MS-DOS name
Security descriptor	Obsolete. Security information is now in $Extend$Secure
Attribute list	Location of additional MFT records, if needed
Object ID	64-bit file identifier unique to this volume
Reparse point	Used for mounting and symbolic links
Volume name	Name of this volume (used only in $Volume)
Volume information	Volume version (used only in $Volume)
Index root	Used for directories
Index allocation	Used for very large directories
Bitmap	Used for very large directories
Logged utility stream	Controls logging to $LogFile
Data	Stream data; may be repeated