USN | | | | | | | | | | |

Internal Assessment Test 3 – May. 2022

Sub:	Operating System					Sub Code:	18 EC 641	Branch:	ECE	
Date:	09-07-22	Duration:	90 min's	Max Marks:	50	Sem / Sec:	6 – A B C D		OBE	

	Answer any FIVE FULL Questions	MARKS	CO	RBT
1.	With a diagram interpret the interface between file system and IOCS.	[10]	CO3	L2
2.	Interpret directory structure with relevant diagram and its fields.	[10]	CO3	L2
3.	Relate the disk space allocation method with conventional method.	[10]	CO3	L3
4.	Illustrate the file organization and access methods in detail	[10]	CO3	L3
5.	With a neat diagram interpret message passing.	[10]	CO5	L3
6.	Explain mail box with relevant diagram.	[10]	CO5	L2
7.	Define deadlock and explain deadlock prevention mechanism.	[10]	CO5	L2

USN | | | | | | | | | | |

Internal Assessment Test 3 – May. 2022

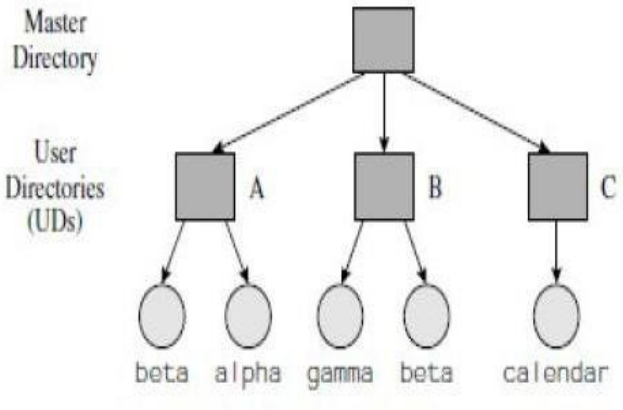
Sub:	Operating System					Sub Code:	18 EC 641	Branch:	ECE	
Date:	09-07-22	Duration:	90 min's	Max Marks:	50	Sem / Sec:	6 – A B C D		OBE	

	Answer any FIVE FULL Questions	MARKS	CO	RBT
1.	With a diagram interpret the interface between file system and IOCS.	[10]	CO3	L2
2.	Interpret directory structure with relevant diagram and its fields.	[10]	CO3	L2
3.	Relate the disk space allocation method with conventional method.	[10]	CO3	L3
4.	Illustrate the file organization and access methods in detail	[10]	CO3	L3
5.	With a neat diagram interpret message passing.	[10]	CO5	L3
6.	Explain mail box with relevant diagram.	[10]	CO5	L2
7.	Define deadlock and explain deadlock prevention mechanism.	[10]	CO5	L2

Internal Assessment Test 3– July. 2022

Sub:	Operating System					Sub Code:	**18 EC 641**	Branch:	ECE
Date:	**09-07-22**	Duration:	90 Minutes	Max Marks:	50	Sem / Sec:	6/A,B,C,D		OBE

	Scheme and Solution	MARKS	CO	RBT
1	The file system uses the IOCS to perform I/O operations and the IOCS implements them rough kernel calls. The interface between the file system and the IOCS consists of three data structures the *file map table* (FMT), the *file control block* (FCB), and the *open files table* (OFT) and functions that perform I/O operations. Use of these data structures avoids repeated processing of file attributes by the file system, and provides a convenient method of tracking the status of ongoing file processing activities. The file system allocates disk space to a file and stores information about the allocated disk space in the *file map table* (FMT). The FMT is typically held in memory during the processing of a file. A file control block (FCB) contains all information concerning an ongoing file processing activity. This information can be classified into the three categories shown in Table 6.2.Information in the file organization category is either simply extracted from the file declaration statement in an application program, or inferred from it by the compiler, e.g., information such as the size of a record and number of buffers is extracted from a file declaration, while the name of the access method is inferred from the type and organization of a file. The compiler puts this information as parameters in the open call. When the call is made during execution of the program, the file system puts this information in the FCB. Directory information is copied into the FCB through joint actions of the file system and the IOCS when a new file is created. Information concerning the current state of processing is written into the FCB by the IOCS. This information is continually updated during the processing of a file. The open files table (OFT) holds the FCBs of all open files. The OFT resides in the kernel address space so that user processes cannot tamper with it. When a file is opened, the file system stores its FCB in a new entry of the OFT. The offset of this entry in the OFT is called the internal id of the file. The internal id is passed back to the process, which uses it as a parameter in all future file system calls. **Fields in FCB** **Category** — **Fields** File organization — File name; File type, organization, and access method; Device type and address; Size of a record; Size of a block; Number of buffers; Name of access method Directory information — Information about the file's directory entry; Address of parent directory's FCB; Address of the file map table (FMT) (or the file map table itself); Protection information Current state of processing — Address of the next record to be processed; Addresses of buffers	06 04	CO3	L2
2	A directory contains information about a group of files. Each entry in a directory contains the attributes of one file, such as its type, organization, size, location, and the manner in which it may be accessed by various users in the system. Figure shows the fields of a typical directory entry. The *open count* and *lock* fields are used when several processes open a file concurrently. The *open count* indicates the number of such processes. As long as this count is non zero, the file system keeps some of the metadata concerning the file in memory to speedup accesses to the data in the file. The *lock* field is used when a process desires exclusive access to a file. A user can create a file to hold data or to act as a directory. When a distinction between the two is important, we will call these files respectively *data files* and *directory files*, or simply	06	CO3	L2

	directories. In a **directory tree**, each file except the root directory has exactly one parent directory. This provides total separation of different users files and complete file naming freedom. However,it makes file sharing rather cumbersome  **Field** — **Description** File name — Name of the file. If this field has a fixed size, long file names beyond a certain length will be truncated. Type and size — The file's type and size. In many file systems, the type of file is implicit in its extension; e.g., a file with extension .c is a byte stream file containing a C program, and a file with extension .obj is an object program file, which is often a structured file. Location info — Information about the file's location on a disk. This information is typically in the form of a table or a linked list containing addresses of disk blocks allocated to a file. Protection info — Information about which users are permitted to access this file, and in what manner. Open count — Number of processes currently accessing the file. Lock — Indicates whether a process is currently accessing the file in an exclusive manner. Flags — Information about the nature of the file—whether the file is a directory, a link, or a mounted file system. Misc info — Miscellaneous information like id of owner, date and time of creation, last use, and last modification.  	04		
3	A disk may contain many file systems, each in its own partition of the disk. The file system knows which partition a file belongs to, but the IOCS does not. Hence disk space allocationis performed by the file system. Early file systems adapted the contiguous memory allocation model by allocating a single contiguous disk area to a file when it was created. This model was simple to implement. It also provided data access efficiency by reducing disk head movement during sequential access to data in a file. However, contiguous allocation of disk space led to external fragmentation. Interestingly, it also suffered from internal fragmentation because the file system found it prudent to allocate some extra disk space to allow for expansion of a file. **Linked Allocation** A file is represented by a linked list of disk blocks. Each disk block has two fields in it *data* and *metadata.* The *data* field contains the data written into the file, while the *metadata* fieldis the link field, which contains the address of the next disk block allocated to the file. Figure6.10 illustrates linked allocation. The *location info* field of the directory entry of file alpha points to the first disk block of the file. Other blocks are accessed by following the pointers in the list of disk blocks. The last disk block contains null information in itsmetadata field. Thus, file alpha consists of disk blocks 3 and 2, while file beta consists of blocks 4, 5,and 7.	[04]	CO3	L3

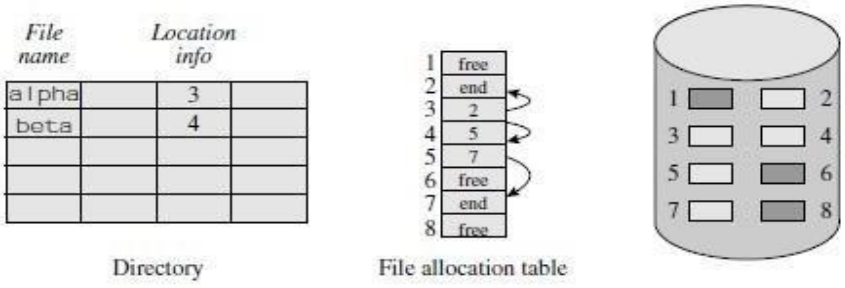


Fig :linked allocation of disk space

[03]

File Allocation Table (FAT)

MS-DOS uses a variant of linked allocation that stores the metadata separately from the file data. A **file allocation table (FAT)** of a disk is an array that has one element corresponding to every disk block in the disk. For a disk block that is allocated to a file, the corresponding FAT element contains the address of the next disk block. Thus the disk block and its FAT element together form a pair that contains the same information as the disk block in a classical linked allocation scheme.

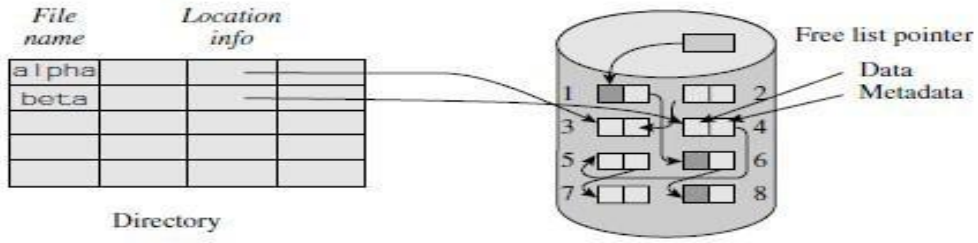


Fig: File Allocation Table

[03]

Indexed Allocation

In indexed allocation, an index called the *file map table* (FMT) is maintained to note the addresses of disk blocks allocated to a file. In its simplest form, an FMT can be an array containing disk block addresses. Each disk block contains a single field the data field. The field of a file's directory entry points *location info* to the FMT for the file

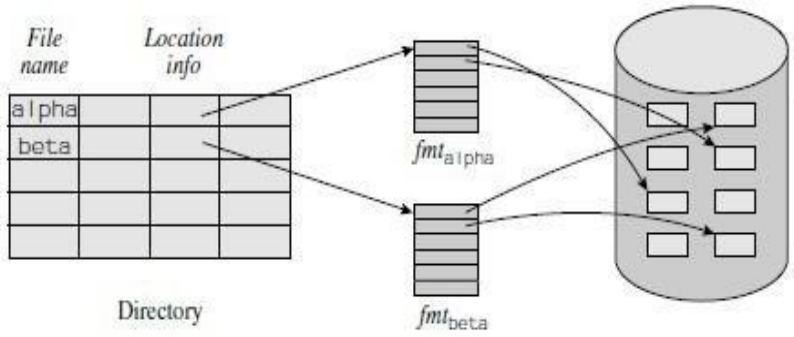
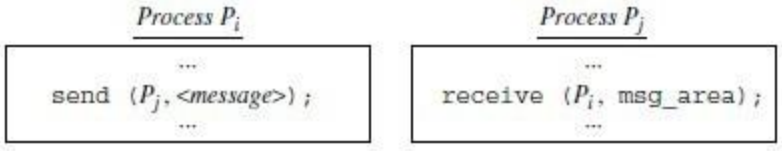
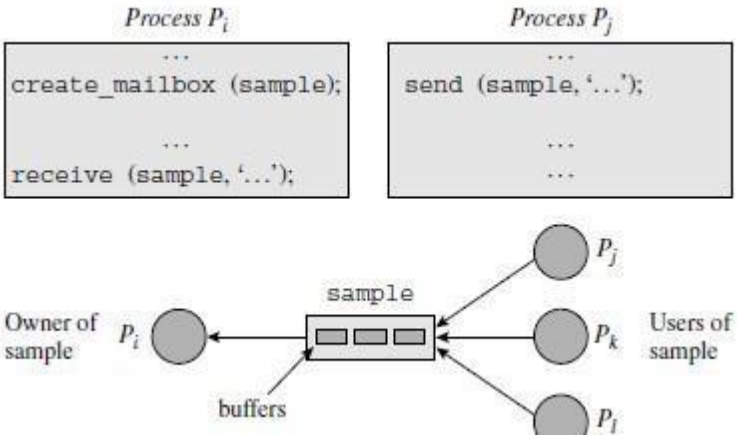


Figure:Indexed allocation of disk space.

4	A **file organization** is a combination of two features a method of arranging records in a file and a procedure for accessing them. A file organization is designed to exploit the characteristics of an I/O device for providing efficient record access for a specific record access pattern. A file system supports several file organizations so that a process can employ the one that best suits its file processing requirements and the I/O device in use. This section describes three fundamental file organizations sequential file organization, direct file organization and index sequential file organization. Other file organizations used in practice are either variants of these fundamental ones or are special-purpose organizations that exploit less commonly used I/O devices. Accesses to files governed by a specific file organization are implemented by an IOCS module called an *access method.* An access method is a policy module of the IOCS. In **sequential file organization,** records are stored in an ascending or descending sequence according to the key field; the record access pattern of an application is expected to follow suit. Hence sequential file organization supports two kinds of operations: read the next (or previous) record, and skip the next (or previous) record. A sequential-access file is used in an application if its data can be conveniently pre-sorted into an ascending or descending order. The sequential	[03]	CO3	L3

	file organization is also used for byte stream files. The **direct file organization** provides convenience and efficiency of file processing when records are accessed in a random order. To access a record, a read/write command needs to mention the value in its key field. We refer to such files as direct access files. A direct-access file is implemented as follows: When a process provides the key value of a record to be accessed, the access method module for the direct file organization applies a transformation to the key value that generates the address of the record in the storage medium. If the file is Organized on a disk, the transformation generates a (track no, record no) address. The disk heads are now positioned on the track track no before a read or write command is issued on the record record no. The **index sequential file** organization is a hybrid organization that combines elements of the indexed and the sequential file organizations. To locate a desired record, the access method module for this organization searches an index to identify a section of the disk that may contain the record, and searches the records in this section of the disk sequentially to find the record. The search succeeds if the record is present in the file; otherwise, it results in a failure. This arrangement requires a much smaller index than does a pure indexed file because the index contains entries for only some of the key values. It also provides better access efficiency than the sequential file organization while ensuring comparably efficient use of I/O media.	[03] [04]		
5	Message passing suits diverse situations where exchange of information between processes plays a key role. One of its prominent uses is in the *client server* paradigm, wherein a *server* process offers a service, and other processes, called its *clients*, send messages to it to use its service. This paradigm is used widely a microkernel- based OS structures functionalities suchas scheduling in the form of servers, a conventional OS offer services such as printing through servers, and, on the Internet, a variety of services are offered by Web servers. Another prominent use of message passing is in higher-level protocols for exchange of electronic mails and communication between tasks in parallel or distributed programs. Here, message passing is used to exchange information, while other parts of the protocol are employed to ensure reliability. The key issues in message passing are how the processes that send and receive messages identify each other, and how the kernel performs various actions related to delivery of messages how it stores and delivers messages and whether it blocks a process that sends a message until its message is delivered. These features are operating system specific. We describe different message passing arrangements employed in operating systems and discuss their significance for user processes and for the kernel. We also describe message passing in UNIX and in Windows operating systems. The four ways in which processes interact with one another *data sharing*, *message passing*, *synchronization*, and *signals*. Data sharing provides means to access values of shared data ina mutually exclusive manner. Process synchronization is performed by blocking a process until other processes have performed certain specific actions. Capabilities of message passing overlap those of data sharing and synchronization; however, each form of process interaction has its own niche application area..Figure 8.1 shows an example of message passing. Process *Pi* sends a message to process *Pj* by executing the statement send (*Pj*, <*message*>). The compiled code of the send statement invokes the library module send. Send makes a system call *send*, with *Pj* and the message as parameters. Execution of the statement receives (*Pi*, msg_area), where msg_area is an area in *Pj* space, results in a system call *receive*.  Figure:Message passing.	[10]	CO5	L3

	The semantics of message passing are as follows: At a *send* call by *Pi* , the kernel checks whether process *Pj* is blocked on a *receive* call for receiving a message from process *Pi* . If so, it copies the message into msg_area and activates *Pj*. If process *Pj* has not already made a *receive* call, the kernel arranges to deliver the message to it when *Pj* eventually makes a *receive* call. When process *Pj* receives the message, it interprets the message and takes an appropriate action. Messages may be passed between processes that exist in the same computer or in different computers connected to a network. Also, the processes participating in message passing may decide on what a specific message means and what actions the receiver process should perform on receiving it. Because of this flexibility, message passing is used in the following applications: Message passing is employed in the *client server* paradigm, which is used to communicate between components of a microkernel-based operating system and user processes, to provide services such as the print service to processes within an OS, or to provide Web-based services to client processes located in other computers. Message passing is used as the backbone of higher-level protocols employed for communicating between computers or for providing the electronic mail facility. Message passing is used to implement communication between tasks in a parallel or distributed program. In principle, message passing can be performed by using shared variables. For example, msg_area in Figure. Two important issues in message passing are: **Naming of processes***:* Whether names of sender and receiver processes are explicitly indicated in send and receive statements, or whether their identities are deduced by the kernel in some other manner. **Delivery of messages***:* Whether a sender process is blocked until the message sent by it is delivered, what the order is in which messages are delivered to the receiver process, and how exceptional conditions are handled.			
6	A mailbox is a repository for inter process messages. It has a unique name. The owner of a mailbox is typically the process that created it. Only the owner process can receive messages from a mailbox. Any process that knows the name of a mailbox can send messages to it. Thus, sender and receiver processes use the name of a mailbox, rather than each other‘s names, in send and receive statements; it is an instance of *indirect naming*. Figure 8.6llustrates message passing using a mailbox named sample. Process *Pi* creates the mailbox, using the statement create_mailbox. Process *Pj* sends a message to the mailbox, using the mailbox name in its send statement. If *Pi* has not already executed a receive statement, the kernel would store the message in a buffer. The kernel may associate a fixed set of buffers with each mailbox, or it may allocate buffers from a common pool of buffers when a message is sent. Both create_mailbox and send statements return with condition codes.The kernel may provide a fixed set of mailbox names, or it may permit user processes to assign mailbox names of their choice. In the former case, confidentiality of communication between a pair of processes cannot be guaranteed because any process can use a mailbox. Confidentiality greatly improves when processes can assign mailbox names of their own choice. To exercise control over creation and destruction of mailboxes, the kernel may require a process for explicitly connect to a mailbox before using it and to disconnect when it finishes using it.	[07] [03]	CO5	L2

	 Figure : Creation and use of mailbox sample. **Use of a mailbox has following advantages:** *Anonymity of receiver:* A process sending a message to request a service may have no interest in the identity of the receiver process, as long as the receiver process can perform the needed function. A mailbox relieves the sender process of the need to know the identity of the receiver. Additionally, if the OS permits the ownership of a mailbox to be changed dynamically, one process can readily take over the service of another. *Classification of messages:* A process may create several mailboxes, and use each mailbox to receive messages of a specific kind. This arrangement permits easy classification of messages.			
7	A deadlock is a situation concerning a set of processes in which each process in the set waits for an event that must be caused by another process in the set. **Deadlock Prevention** For a deadlock to occur each of the four necessary conditions must hold. If at least one of the there condition does not hold then we can prevent occurrence of deadlock. • Non – Shareable Resource : This holds for non-sharable resources. Eg:-A printer can be used by only one process at a time. Mutual exclusion is not possible in sharable resources and thus they cannot be involved in deadlock. Read-only files are good examples for sharable resources. A process never waits for accessing a sharable resource. So we cannot prevent deadlock by denying the mutual exclusion condition in non-sharable resources • Hold and Wait: This condition can be eliminated by forcing a process to release all its resources held by it when it request a resource i.e., not available.x One protocol can be used is that each process is allocated with all of its resources before its start execution. Eg:-consider a process that copies the data from a tape drive to the disk, sorts the file and then prints the results to a printer. If all the resources are allocated at the beginning then the tape drive, disk files and printer are assigned to the process. The main problem with this is it leads to low resource utilization because it requires printer at the last and is allocated with it from the beginning so that no other process can use it. x Another protocol that can be used is to allow a process to request a resource when the process has none. i.e., the process is allocated with tape drive and disk file. It performs the required operation and releases both. Then the process once again request for disk file and the printer and the problem and with this is starvation is possible. • No Preemption:To ensure that this condition never occurs the resources must be preempted. The following protocol can be used. x If a process is holding some resource and request another resource that cannot be immediately	[02]+[08]	CO5	L2

	allocated to it, then all the resources currently held by the requesting process are preempted and added to the list of resources for which other processes may be waiting. The process will be restarted only when it regains the old resources and the new resources that it is requesting. x When a process request resources, we check whether they are available or not. If they are available we allocate them else we check that whether they are allocated to some other waiting process. If so we preempt the resources from the waiting process and allocate them to the requesting process. The requesting process must wait. • Circular Wait:-The fourth and the final condition for deadlock is the circular wait condition. One way to ensure that this condition never, is to impose ordering on all resource types and each process requests resource in an increasing order. Let R={R1,R2,.....Rn} be the set of resource types. We assign each resource type with a unique integer value. This will allows us to compare two resources and determine whether one precedes the other in ordering. Eg:-we can define a one to one function -F(disk drive)=5 F(printer)=12 F(tape drive)=1 Deadlock can be prevented by using the following protocol:x Each process can request the resource in increasing order. A process can request any number of instances of resource type say Ri and it can request instances of resource type Rj only F(Rj) > F(Ri). x Alternatively when a process requests an instance of resource type Rj, it has released any resource Ri such that F(Ri) >= F(Rj). If these two protocol are used then the circular wait can't hold.			