Solution

Internal Assessment Test 3 – November 2019

Sub:	Operating system							**Code:**	17EC553
Date:	19/11/2019	Duration:	90mins	Max Marks:	50	**Sem:**	V	**Branch:**	ECE

Note: Answer Any Five Questions

1. **What are the facilities provided by file-system and IOCS? Write the logical organization of file system and explain.** 10M

Scheme- Facilities provided by file-system and IOCS File system. 5M

Logical organization of file system 5M

Solution- Facilities provided by file-system and IOCS File system layer are

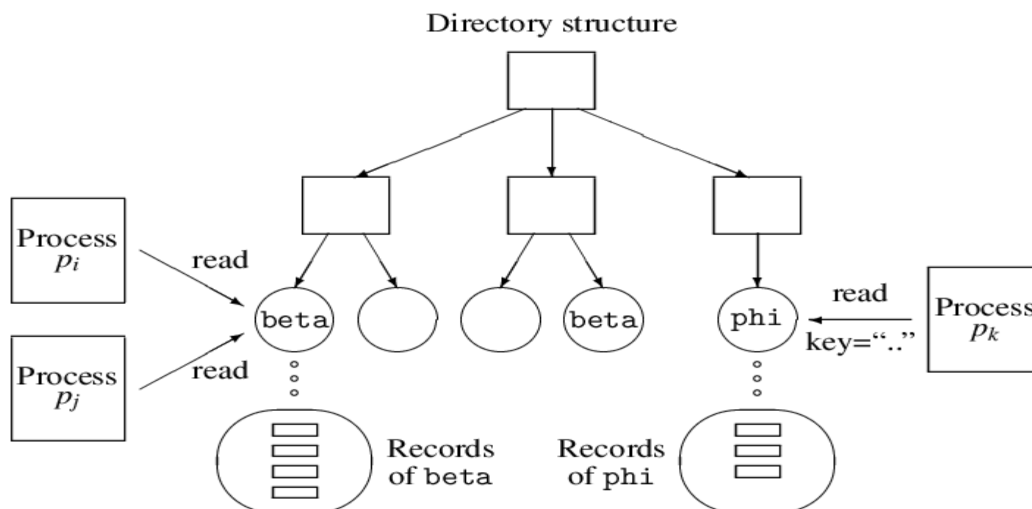
File System

- Directory structures for convenient grouping of files
- Protection of files against illegal accesses
- File sharing semantics for concurrent accesses to a file
- Reliable storage of files

Input-Output Control System (IOCS)

- Efficient operation of I/O devices
- Efficient access to data in a file

Logical organization of file system



Two files named beta exist. The file system must open the correct one when a process executes open (beta, ..)
Records may be organized differently in different files.
Two files named beta exist in the file system. Thus users enjoy file naming freedom Processes P_i and P_j share one of these files. The rules of sharing are determined by *file sharing semantics*
Files beta and phi have different organizations and are accessed differently. File beta is a sequential file; its records are read in a sequence. File phi is a direct file; its records can be read in random order

2. Explain a) File types b) File attributes c) File operations.

Solution-

File Types
A file system houses and organizes different types of files, e.g., data files, executable programs, object modules, textual information, documents, spreadsheets, photos, and video clips. Each of these file types has its own format for recording the data. These file types can be grouped into two classes:
- Structured files
- Byte stream files

A *structured file* is a collection of records, where a record is a meaningful unit for processing of data.A*record* is a collection of fields, and a *field* contains a single data item. Each record in a file is assumed to contain a *key* field. The value in the key field of a record is unique in a file; i.e., no two records contain an identical key.
File types used by standard system software like compilers and linkers have a structure determined by the OS designer, while file types of user files depend on the applications or programs that create them.
A *byte stream file* is "flat." There are no records and fields in it; it is looked upon as a sequence of bytes by the processes that use it.

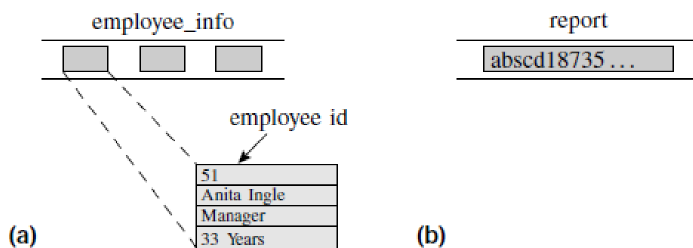


Fig. illustration of structured and byte stream files.

File Attributes
A file attribute is a characteristic of a file that is important either to its users or to the file system, or both. Commonly used attributes of a file are: type, organization, size, location on disk, access control information, which indicates the manner in which different users can access the file; owner name, time of creation, and time of last use. The file system stores the attributes of a file in its directory

entry. During a file processing activity, the file system uses the attributes of a file to locate it, and to ensure that each operation being performed on it is consistent with its attributes. At the end of the file processing activity, the file system stores changed values of the file's attributes, if any, in the file's directory entry.

File Operations

Operations such as open, close, rename, and delete are performed by file system modules. Actual access of files, i.e., reading or writing of records, is implemented by the IOCS modules.

Operation	Description
Opening a file	The file system finds the directory entry of the file and checks whether the user whose process is trying to open the file has the necessary access privileges for the file. It then performs some housekeeping actions to initiate processing of the file.
Reading or writing a record	The file system considers the organization of the file (see Section 13.3) and implements the read/write operation in an appropriate manner.
Closing a file	The file size information in the file's directory entry is updated.
Making a copy of a file	A copy of the file is made, a new directory entry is created for the copy and its name, size, location, and protection information is recorded in the entry.
File deletion	The directory entry of the file is deleted and the disk area occupied by it is freed.
File renaming	The new name is recorded in the directory entry of the file.
Specifying access privileges	The protection information in the file's directory entry is updated.

3. Explain the fundamental file organizations

Solution

The two fundamental record access patterns are *sequential access*, in which records are accessed in the order in which they fall in a file (or in the reverse of that order), and *random access*, in which records may be accessed in any order. The file processing actions of a process will execute efficiently only if the process's record access pattern can be implemented efficiently in the file system. The characteristics of an I/O device make it suitable for a specific record access pattern. For example, a tape drive can access only the record that is placed immediately before or after the current position of its read/write head. Hence it is suitable for sequential access to records. A disk

drive can directly access any record given its address. Hence it can efficiently implement both the sequential and random record access patterns.

A ***file organization*** is a combination of two features—a method of arranging records in a file and a procedure for accessing them.Afile 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. While compiling a program, the compiler infers the file organization governing a file from the file's declaration statement (or from the rules for default, if the program does not contain a file declaration statement), and identifies the correct access method to invoke for operations on the file.We describe the functions of access methods after discussing the fundamental file organizations

Sequential File Organization

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 presorted into an ascending or descending order. The sequential file organization is also used for byte stream files.

Direct File Organization

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*. Consider a file of employee information organized as a direct-access file. Let p records be written on one track of the disk. Assuming the employee numbers and the track and record numbers of the file to start from 1, the address of the record for employee number n is (*track number* (*tn*), *record number* (*rn*))

$$t_n = \left\lceil \frac{n}{p} \right\rceil$$

$$r_n = n - (t_n - 1) \times p$$

Direct file organization provides access efficiency when records are processed randomly. However, it has three drawbacks compared to sequential file organization:

• Record address calculation consumes CPU time.

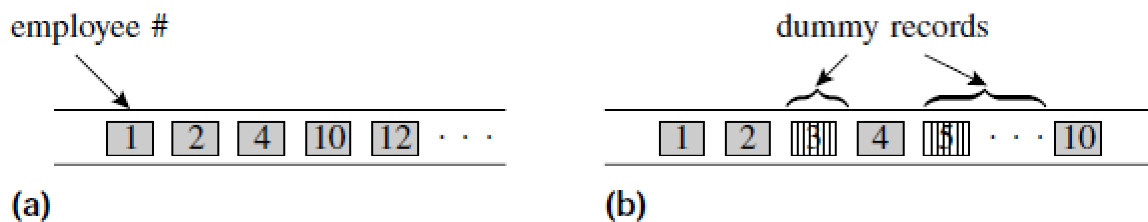
• Disks can store much more data along the outermost track than along the innermost track.However, the direct file organization stores an equal amount of data along each track. Hence some recording capacity is wasted.

• The address calculation formulas work correctly only if a record exists for every possible value of the key, so dummy records have to exist for keys that are not in use. This requirement leads to poor utilization of the I/O medium.

Hence sequential processing of records in a direct-access file is less efficient than processing of records in a sequential-access file. Another practical problem is that characteristics of an I/O device are explicitly assumed and used by the address calculation formulas which make the file organization device dependent. Rewriting the file on another device with different characteristics, e.g., different track capacity, will imply modifying the address calculation formulas. This requirement affects the portability of programs.

Index Sequential File Organization

An *index* helps to determine the location of a record from its key value. In a pure indexed file organization, the index of a file contains an index entry with

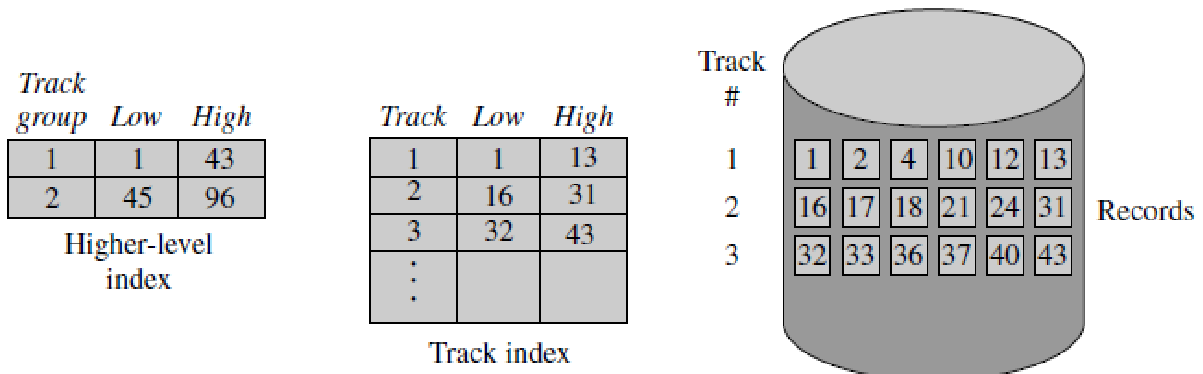


Records in (a) sequential file; (b) direct-access file.

the format (key value, disk address) for each key value existing in the file. To access a record with key *k*, the index entry containing *k* is found by searching the index, and the disk address mentioned in the entry is used to access the record. If an index is smaller than a file, this arrangement provides high access efficiency because a search in the index is more efficient than a search in the file.

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.

For a large file the index would still contain a large number of entries, and so the time required to search through the index would be large. A higher-level index can be used to reduce the search time. An entry in the higher-level index points to a section of the index. This section of the index is searched to find the section of the disk that may contain a desired record, and this section of the disk is searched sequentially for the desired record. The next example illustrates this arrangement.



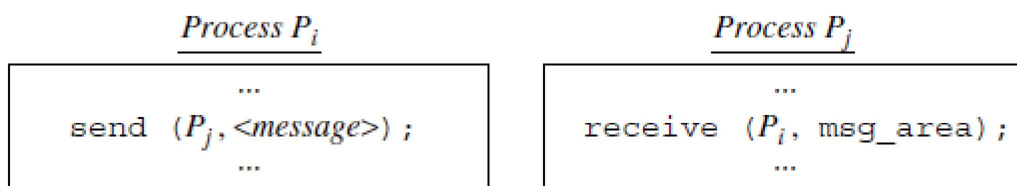
Track index and higher-level index in an index sequential file.

4) Define message passing. Illustrate the implementations of message passing.

Solution

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 such as scheduling in the form of servers, a conventional OS offers 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.



Message passing.

Four ways in which processes interact with one another—*data sharing, message passing, synchronization*, and *signals*.

Because of this flexibility, message passing is used in the following applications:

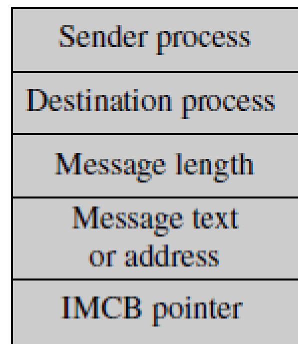
• Message passing is employed in the *client–server* paradigm, which is used to communicate between components of a microkernel-based operating system and user processes, to provide services such as the print service to processes within an OS, or to provideWeb-based services to client processes located 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.

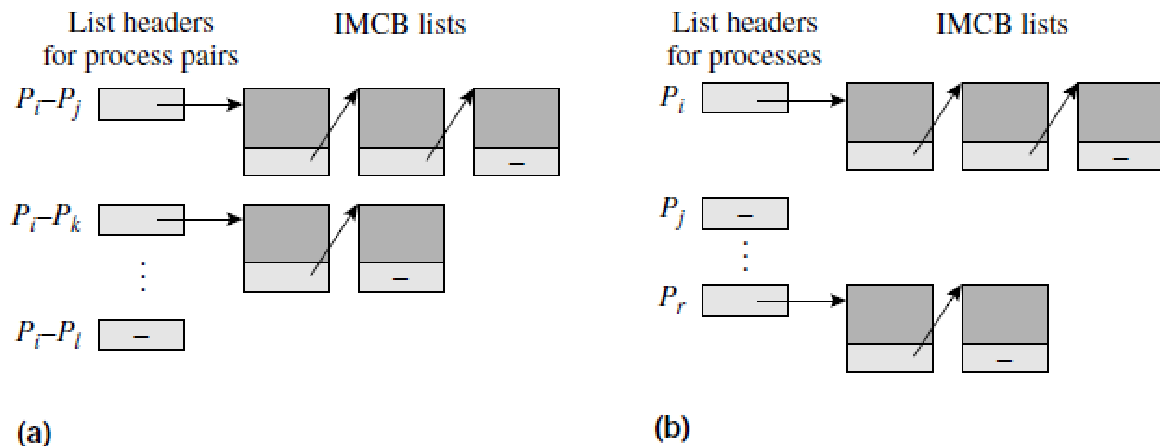
IMPLEMENTING MESSAGE PASSING

Buffering of Interprocess Messages When a process *Pi* sends a message to some process *Pj* by using a nonblocking *send*, the kernel builds an *interprocess message control block* (IMCB) to store all information needed to deliver the message. The control block contains names of the sender and destination processes, the length of the message, and the text of the message. The control block is allocated a buffer in the kernel area. When process *Pj* makes a *receive* call, the kernel copies the message from the appropriate IMCB into the message area provided by *Pj*. The pointer fields of IMCBs are used to form IMCB lists to simplify message delivery. Figure shows the organization of IMCB lists when blocking *send*s and FCFS message delivery are used. In symmetric naming, a separate list is used for every pair of communicating processes. When a process *Pi* performs a *receive* call to receive a message from process *Pj* , the IMCB list for the pair *Pi*–*Pj* is used to deliver the message. In asymmetric naming, a single IMCB list can be maintained per recipient process. When a process performs a *receive,* the first IMCB in its list is processed to deliver a message.



Interprocess message control block (IMCB).

If blocking *send*s are used, at most one message sent by a process can be undelivered at any point in time. The process is blocked until the message is delivered. Hence it is not necessary to copy the message into an IMCB. The



Lists of IMCBs for blocking *send*s in (a) symmetric naming; (b) asymmetric naming.

kernel can simply note the address of the message text in the sender's memory area, and use this information while delivering the message. This arrangement saves one copy operation on the message. However, it faces difficulties if the sender is swapped out before the message is delivered, so it may be preferable to use an IMCB. Fewer IMCBs would be needed than when *send*s are nonblocking, because at most one message sent by each process can be in an IMCB at any time. The kernel may have to reserve a considerable amount of memory for interprocess messages, particularly if nonblocking sends are used. In such cases, it may save message texts on the disk. An IMCB would then contain the address of the disk block where the message is stored, rather than the message text itself.

5. Define mailbox. Explain message passing using a mailbox with necessary sketches. Also mention the advantages of using mailboxes.

A mailbox is a repository for interprocess 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 shows 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 to explicitly "connect" to a mailbox before starting to use it, and to "disconnect" when it finishes using it. This way it can destroy a mailbox, If no process is connected to it.

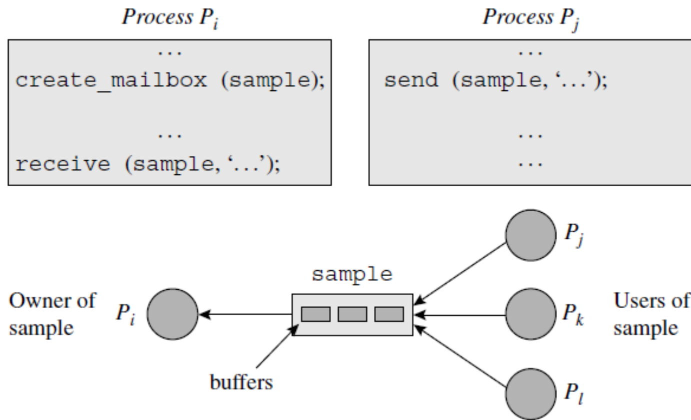


Figure: Creation and use of mailbox sample.

Alternatively, it may permit the owner of a mailbox to destroy it. In that case, it has the responsibility of informing all processes that have "connected" to the mailbox. The kernel may permit the owner of a mailbox to transfer the ownership to another process. 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.

Anonymity of a receiver process, as we just saw, can offer the opportunity to transfer a function from one process to another. Consider an OS whose kernel is structured in the form of multiple processes communicating through messages. Interrupts relevant to the process scheduling function can be modeled as messages sent to a mailbox named *scheduling*. If the OS wishes to use different process scheduling criteria during different periods of the day, it may implement several schedulers as processes and pass ownership of the *scheduling* mailbox among these processes. This way, the process scheduler that currently owns *scheduling* can receive all scheduling-related messages. Functionalities of OS servers can be similarly transferred. For example, all print requests can be directed to a laser printer instead of a dot matrix printer by simply changing the ownership of a *print* mailbox.

Although a process can also remain anonymous when sending a message to a mailbox, the identity of the sender often has to be known. For example, a server may be programmed to return status information for each request. It can be achieved by passing the sender's id along with the text of the message. The sender of the message, on the other hand, might not know the identity of the server; then, it would have to receive the server's reply through an asymmetric *receive*. As an alternative, the compiler can implement the *send* call as a blocking call requiring a reply containing the status information; so, return of status information would be a kernel responsibility.

6. with necessary sketches, explain the different deadlock prevention approaches.

<u>DEADLOCK PREVENTION:</u>

The four conditions must hold simultaneously for a resource deadlock to arise in a system. To prevent deadlocks, the kernel must use a resource allocation policy that ensures that one of these conditions cannot arise. In this section, we first discuss different approaches to deadlock prevention and then present some resource allocation policies that employ these approaches.

Nonshareable Resources Wait-for relations will not exist in the system if all resources could be made shareable. This way paths in an RRAG would contain only allocation edges, so circular waits could not arise. Figure shows the effect of employing this approach: the request edge *(Pi ,Rl)* would be replaced by an allocation edge *(Rl , Pi)* because the resource unit of class *Rl* is shareable.

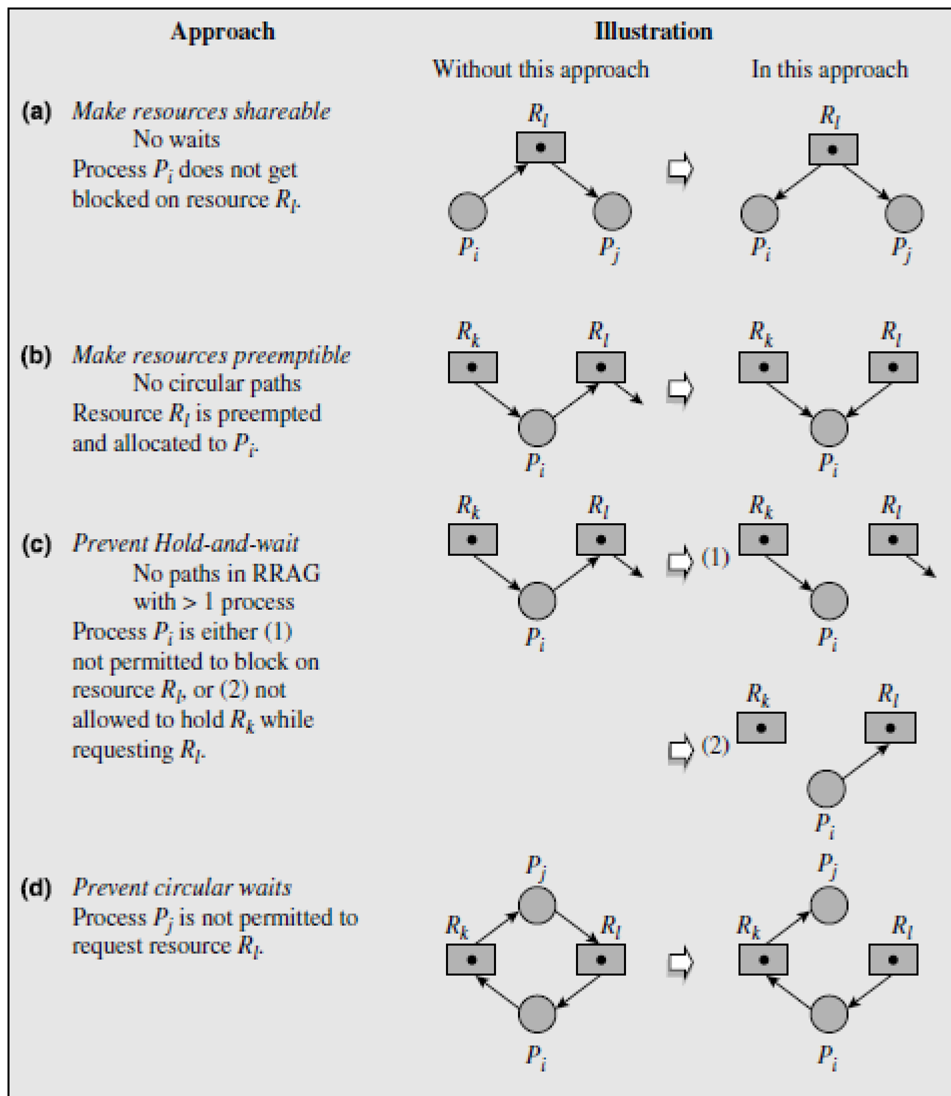


Figure: Approaches to deadlock prevention.

However, some resources such as printers are inherently non-shareable, so how can they be made shareable? OSs uses some innovative techniques to solve this problem. An example is found in the THE multiprogramming system of the 1960s. It contained only one printer, so it buffered the output produced by different processes, formatted it to produce “page images,” and used the printer to print one page image at a time. This arrangement mixed up the printed pages produced by different processes, and so the output of different processes had to be separated manually. The

nonshareability of a device can also be circumvented by creating virtual devices e.g., virtual printers can be created and allocated to processes. However, this approach cannot work for software resources like shared files, which should be modified in a mutually exclusive manner to avoid race conditions.

Preemption of Resources If resources are made preemptible, the kernel can ensure that some processes have all the resources they need, which would prevent circular paths in RRAG. For example, resource *Rl* can be preempted from its current holder and allocated to process *Pi* . However, nonpreemptibility of resources can be circumvented only selectively. The page formatting approach of the THE system can be used to make printers preemptible, but, in general, sequential I/O devices cannot be preempted.

Hold-and-Wait To prevent the hold-and-wait condition, either a process that holds resources should not be permitted to make resource requests, or a process that gets blocked on a resource request should not be permitted to hold any resources. Thus, in Figure c, either edge *(Pi ,Rl)* would not arise, or edge *(Rk, Pl)* would not exist if *(Pi ,Rl)* arises. In either case, RRAG paths involving more than one process could not arise, and so circular paths could not exist. A simple policy for implementing this approach is to allow a process to make only one resource request in its lifetime in which it asks for all the resources it needs.

Circular Wait A circularwait can result fromthe hold-and-wait condition, which is a consequence of the non-shareability and non-preemptibility conditions, so it does not arise if either of these conditions does not arise. Circular waits can be separately prevented by not allowing some processes to wait for some resources; e.g., process *Pj* in Figure (d) may not be allowed to wait for resource *Rl* . It can be achieved by applying a *validity constraint* to each resource request. The validity constraint is a boolean function of the allocation state. It takes the value *false* if the request may lead to a circular wait in the system, so such a request is rejected right away. If the validity constraint has the value *true*, the resource is allocated if it is available; otherwise, the process is blocked for the resource.

8. Write the algorithm for deadlock detection and use it for the following example system to identify if a deadlock exists in it or not.

R1	R2	R3
2	1	0

Free Resources

	R1	R2	R3
P1	2	1	0
P2	1	3	1
P3	1	1	1
P4	1	2	2

Allocated Resources

	R1	R2	R3
P1	2	1	3
P2	1	4	0
P3	0	0	0
P4	1	0	2

Requested Resources

(a) Initial state

Allocated resources

	R_1	R_2	R_3
P_1	2	1	0
P_2	1	3	1
P_3	1	1	1
P_4	1	2	2

Requested resources

	R_1	R_2	R_3
P_1	2	1	3
P_2	1	4	0
P_3			
P_4	1	0	2

Free resources

R_1	R_2	R_3
0	0	1

(b) After simulating allocation of resources to P_4 when process P_3 completes

Allocated resources

P_1	2	1	0
P_2	1	3	1
P_3	0	0	0
P_4	2	2	4

Requested resources

P_1	2	1	3
P_2	1	4	0
P_3			
P_4			

Free resources

0	1	0

(c) After simulating allocation of resources to P_1 when process P_4 completes

Allocated resources

P_1	4	2	3
P_2	1	3	1
P_3	0	0	0
P_4	0	0	0

Requested resources

P_1			
P_2	1	4	0
P_3			
P_4			

Free resources

0	2	1

(d) After simulating allocation of resources to P_2 when process P_1 completes

Allocated resources

P_1	0	0	0
P_2	2	7	1
P_3	0	0	0
P_4	0	0	0

Requested resources

P_1			
P_2			
P_3			
P_4			

Free resources

3	0	4