Virtual memory

When the operating system uses either contiguous allocation or paging or segmentation technique to load programs into main memory, then the operating system has to load the entire program into main memory in order to start the execution of program.

When the size of the program is greater than the size of main memory, then it is not possible to load the entire program into main memory and start the execution of the program.

With virtual memory concept, the operating system can load part of the program into main memory and can start the execution of the program.

Virtual memory supports execution of programs whose size is greater than the size of main memory.

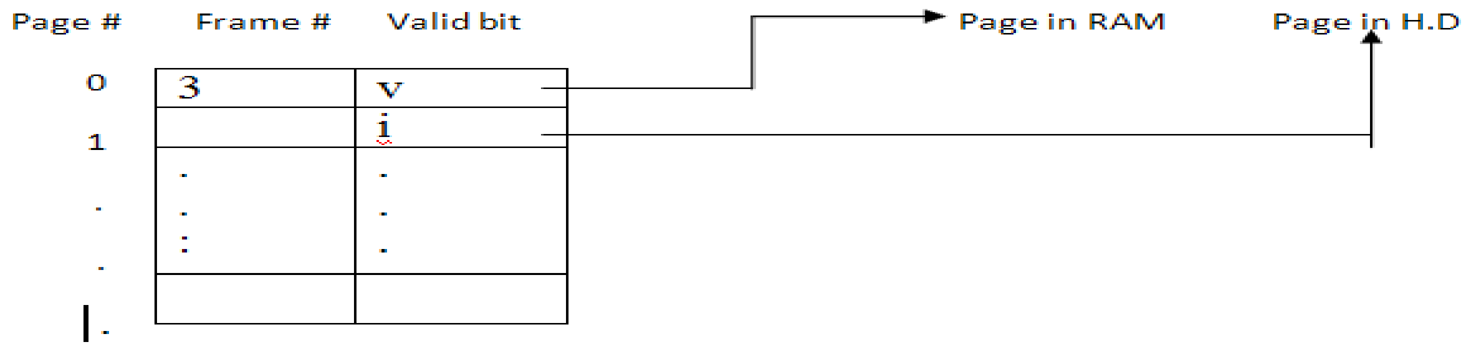
Demand Paging Technique

The space in RAM is divided into a number of equal size frames before loading any program into the RAM.

To load a program into the RAM, the operating system divides the program into a number of equal size pages.

Depending on the number of free frames available in the RAM, the operating system loads either all or few pages of the program into the RAM.

Then, creates a page table for the program as shown below.



The Number of entries in the page table is equal to the number of pages in the program.

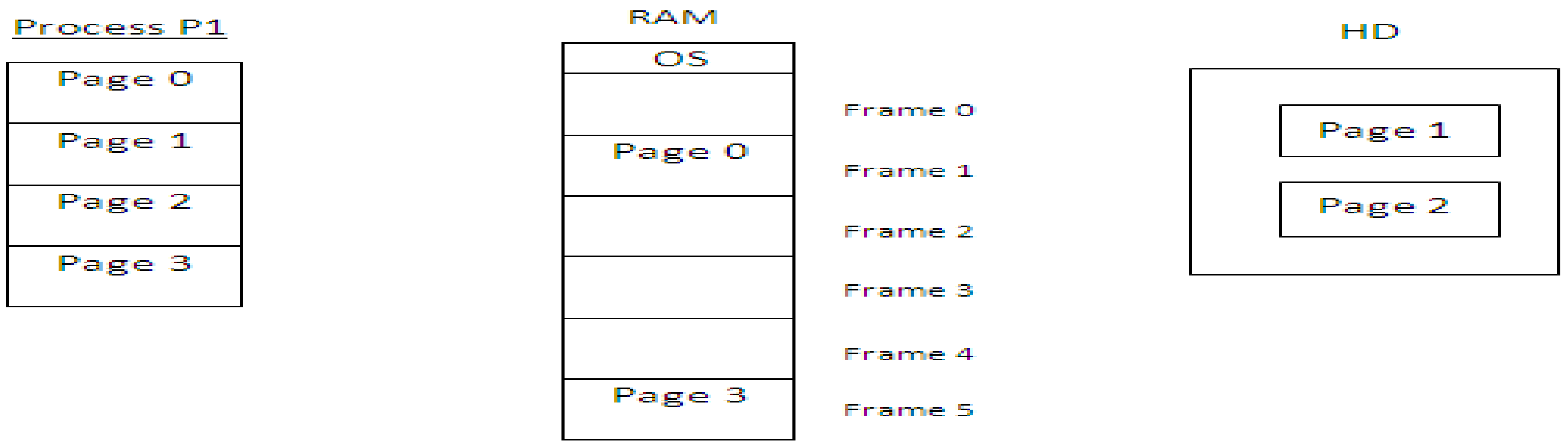
If the page is currently loaded into RAM, then the corresponding entry in the page table is filled with the frame number and ‘v’.

Otherwise, the frame number field is empty and the valid bit field is filled with ‘i’.

In the following figure, program P1 is divided into 4 pages. Two pages are loaded into RAM and remaining two pages are in the HD.

The page table of program P1 contains four entries.

Two entries of the page table contain the corresponding frame numbers and the remaining two entries are empty.



Page table of P1

Page #	Frame #	Valid bit
0	1	v
1		i
2		i
3	5	v

To execute a statement of the program, the CPU generates the logical address of the statement.

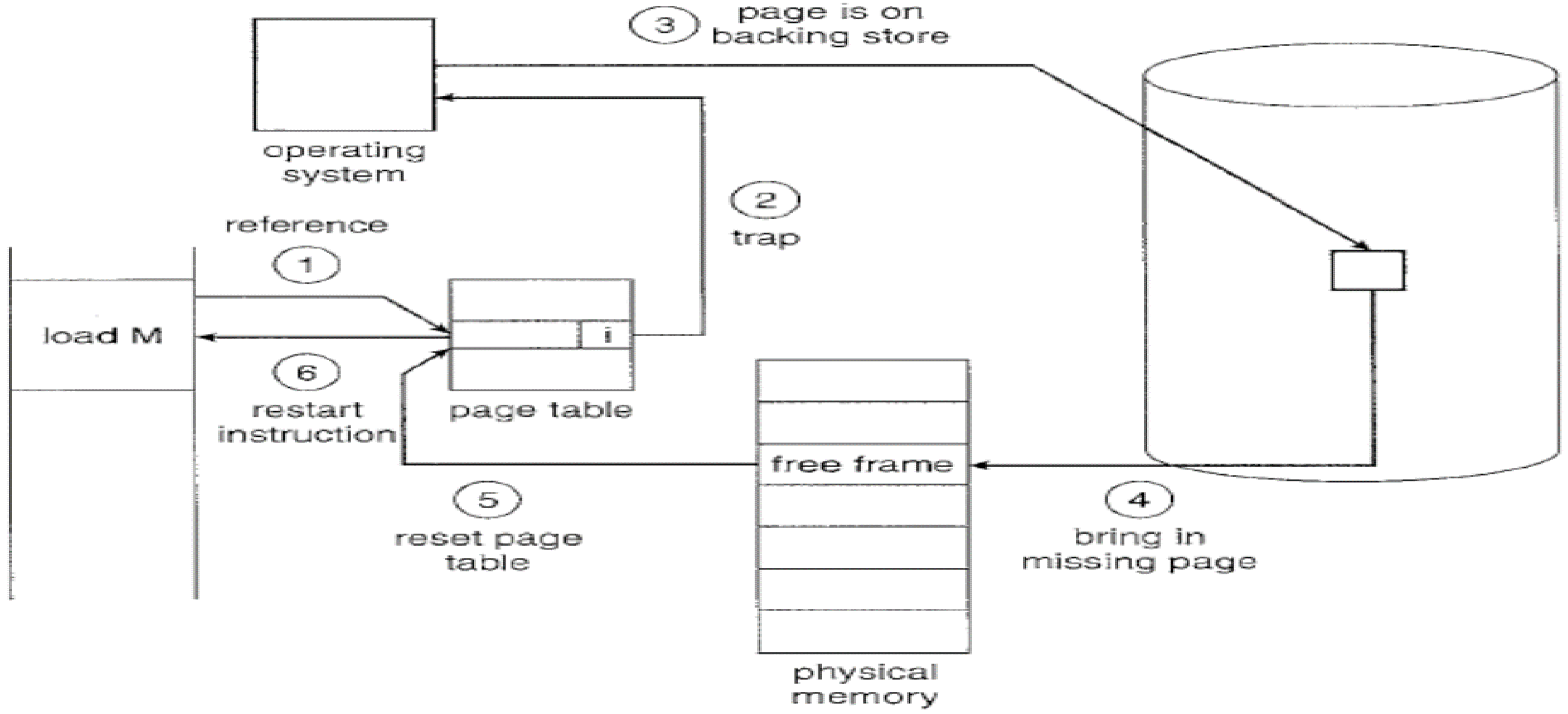
The page number of logical address is used as an index into the page table of the program.

If the entry of page table contains a frame number, then that frame number is appended with the offset of logical address to generate the physical address.

If the entry of page table does not contain any frame number, then that situation is called a page fault.

Operating system performs the following activities when a page fault occurs.

1. Checks whether the RAM contains a free frame or not.
2. If the RAM contains a free frame, then the operating system loads the required page into the free frame and updates the page table of program.
3. Otherwise, the operating system replaces one of the pages in the RAM with the required page and updates the page table of program. Operating System uses a **page replacement algorithm** for replacing one of the pages in the RAM.
4. Instructs the CPU to generate the same logical address again.



3 page is on backing store
operating system
2 trap
reference
1
load M
i
6 restart instruction
page table
free frame
5 reset page table
4 bring in missing page
physical memory

Page Replacement

If the RAM is currently full and if the operating system wants to load a page of a process into RAM, then the operating system moves one of the pages in the RAM to Hard disk and loads the new page into the free frame.

To select a page for replacement, the operating system uses one of the following page replacement algorithms.

1. First in first out (FIFO) page replacement algorithm
2. Optimal page replacement algorithm
3. Least Recently Used (LRU) page replacement algorithm

Reference string

Reference string indicates the order in which the pages of a process are executed or referenced.

First in first out page replacement algorithm

The page that was loaded for the first time into RAM is replaced by the requested page.

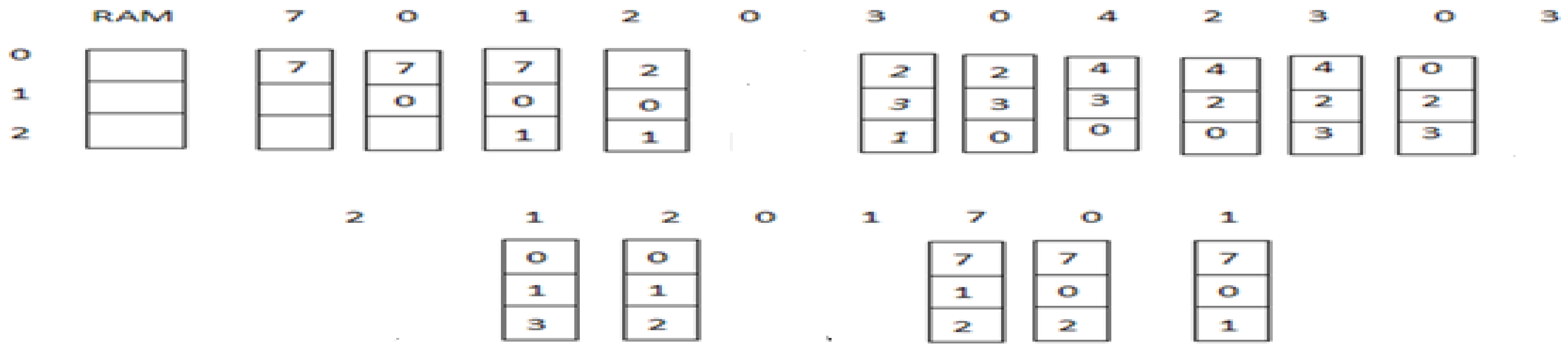
Ex:-

Number of pages in the process=8 (0 to 7)

Number of frames in the RAM=3

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

The replacement of pages in the RAM is shown in below figure



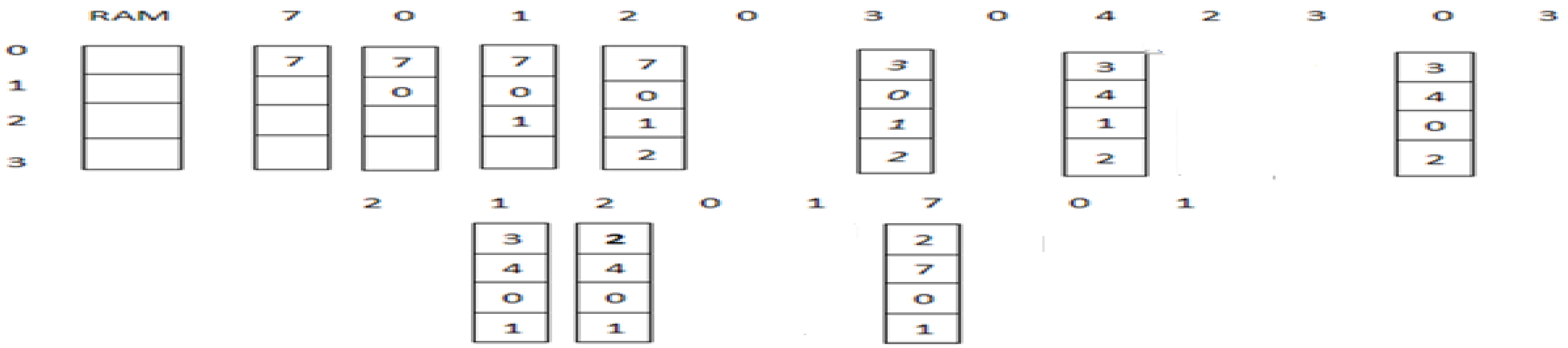
Number of page faults=15

Ex2:-

Number of pages in the process=8

Number of frames in the RAM=4

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1



Number of page faults=10

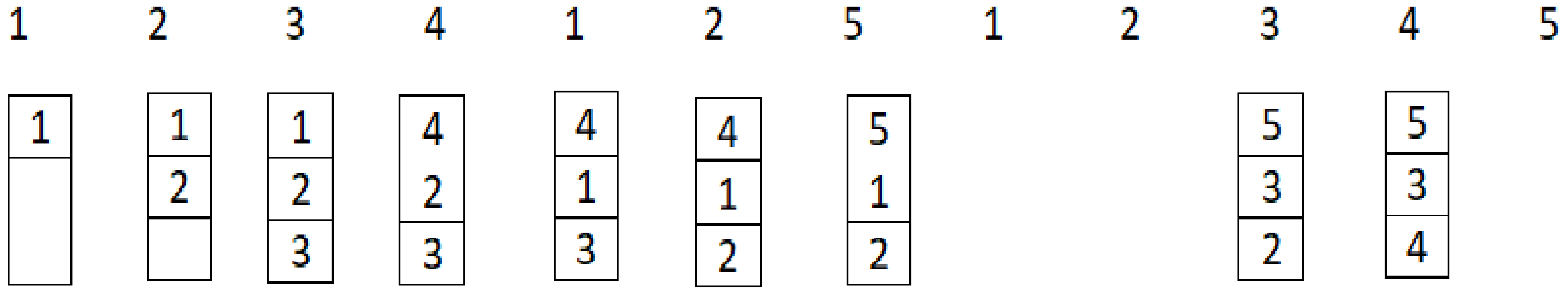
Number of page faults can be reduced by increasing the number of frames in RAM.

Ex3:

Number of pages in the process=5(1 to 5)

Number of frames in the RAM=3

Reference String: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



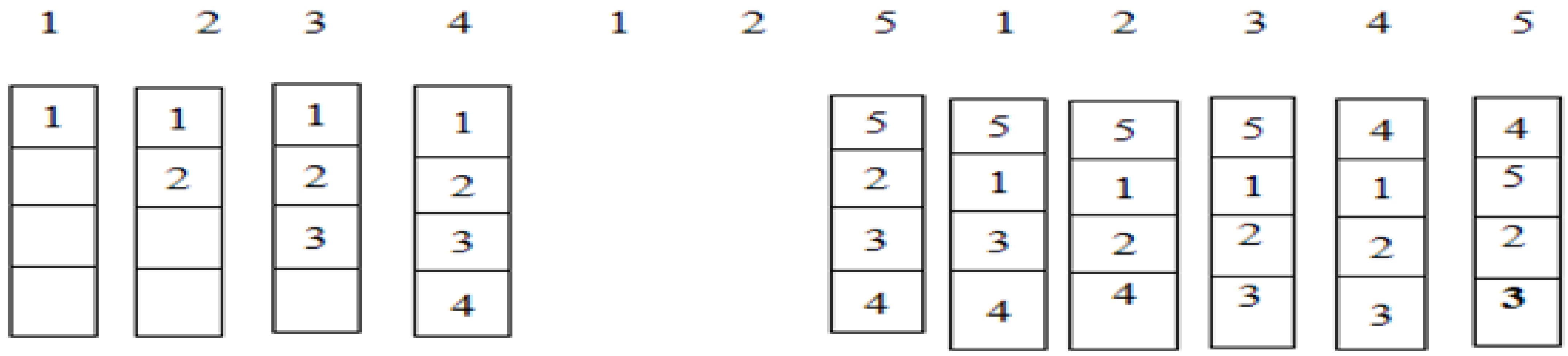
Number of page faults=9

Ex4:

Number of pages in the process=5(1 to 5)

Number of frames in the RAM=4

Reference String: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



Number of page faults=10

Generally, the number of page faults will be decreased by increasing the number of frames in the RAM. With FIFO algorithm, in some cases, the number of page faults increases when the number of frames in the RAM increases.

This situation is called **“Belady’s Anamoly”.**

This is a major drawback with FIFO.

To avoid Belady’s Anamoly, the optimal page replacement algorithm is used.

<u>Optimal page replacement algorithm</u>

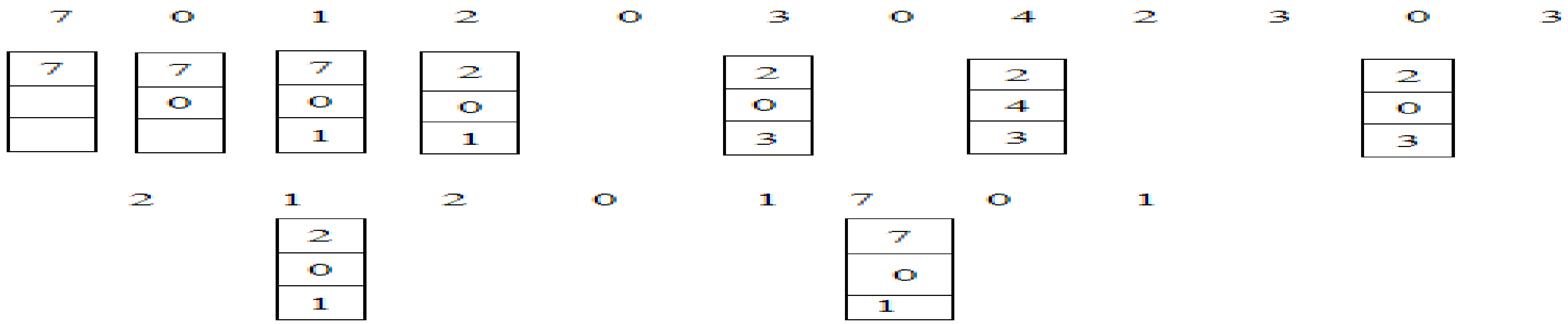
The page that will not be used for the longest period of time will be replaced by the requested page.

Ex1:

Number of pages in the process=8 (0 to 7)

Number of frames in the RAM=3

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1



Number of page Faults=9

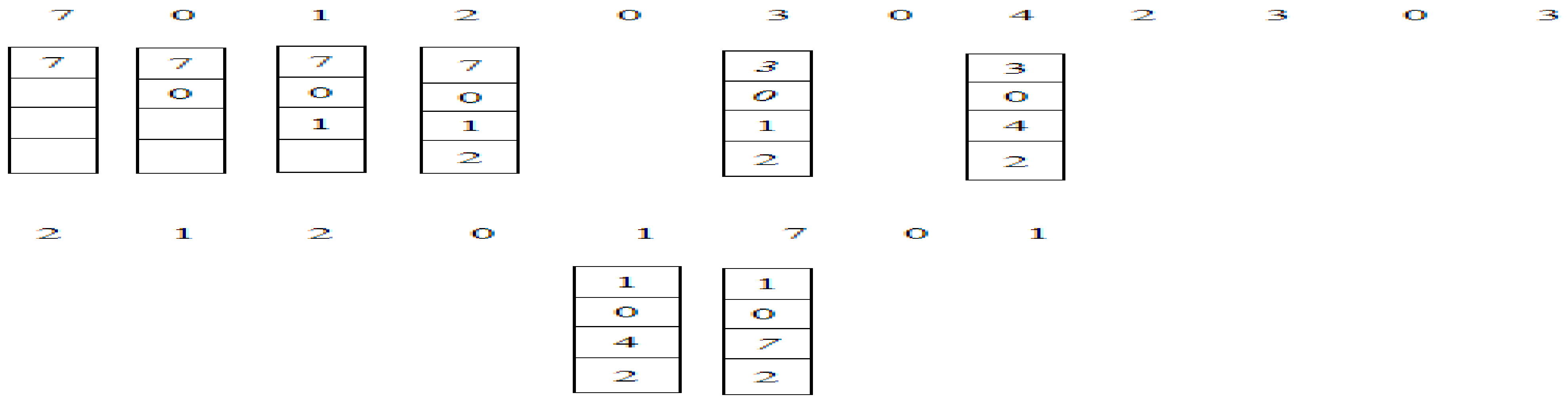
Ex2:

Number of pages in the process=8

Number of frames in the RAM=4

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Number of frames=4



Number of page faults=8

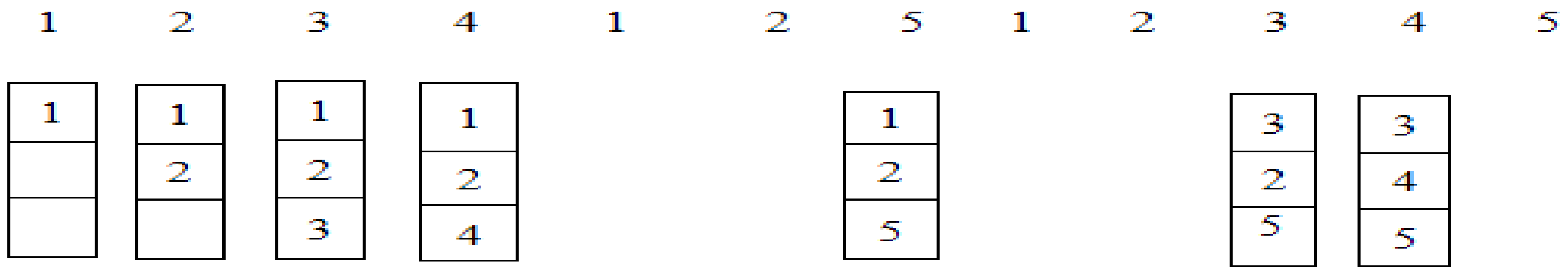
Ex3:

Number of pages in the process=5(1 to 5)

Number of frames in the RAM=3

Reference String: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Number of frames=3



Number of page Faults=7

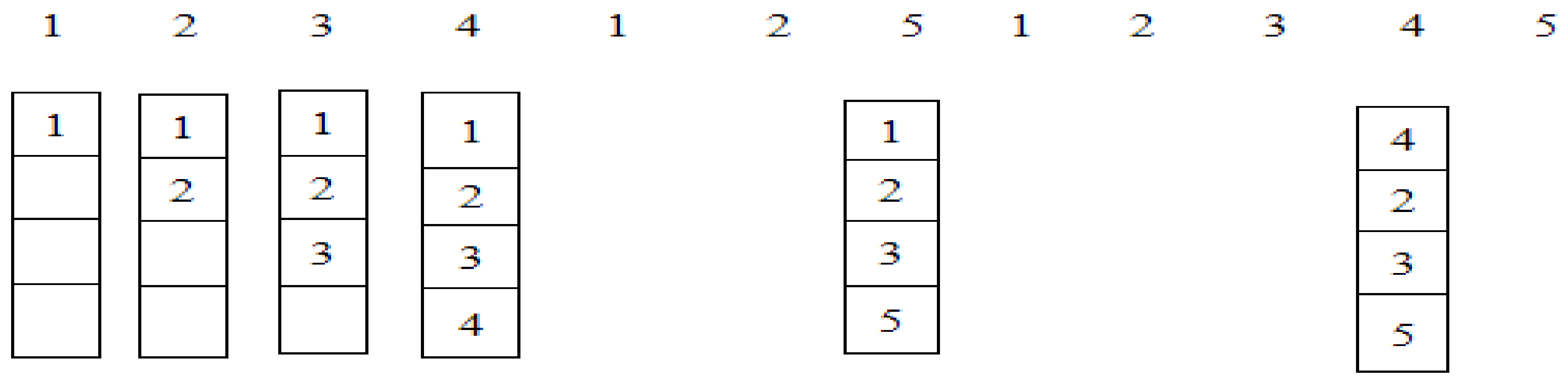
Ex4:

Number of pages in the process=5(1 to 5)

Number of frames in the RAM=4

Reference String: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Number of frames=4



Number of page faults=6

Advantage:

Less number of page faults compared to FIFO.

Disadvantage:

This algorithm works based on the future references of pages.

If the operating system doesn’t know the order in which the pages will be referred, then it is not possible to calculate future references for the pages and not possible to use this algorithm.

Least Recently Used Page Replacement Algorithm

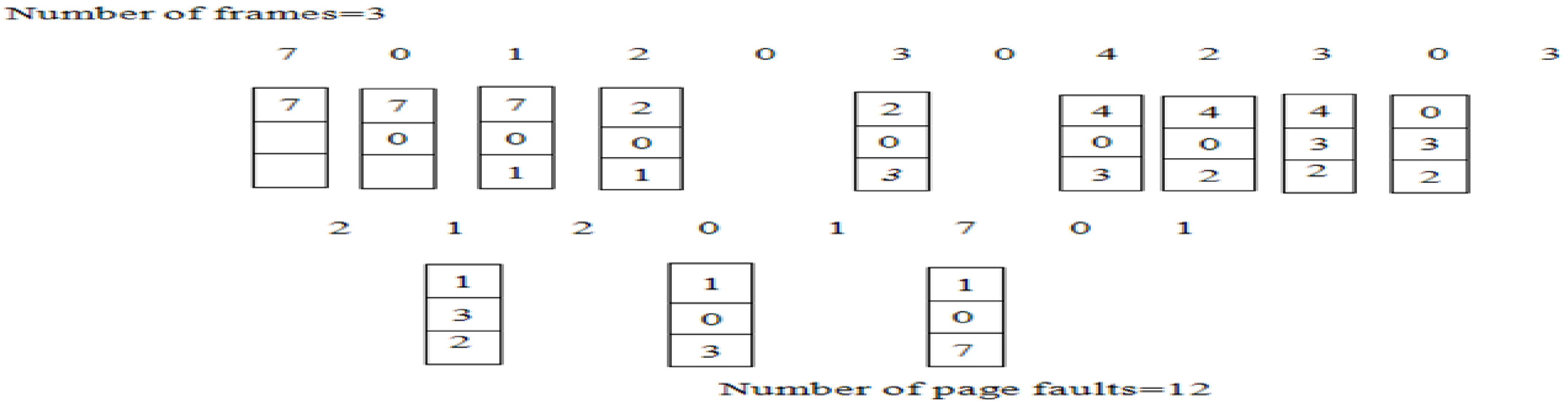
The page that has not been used for the longest period of time is replaced by the requested page.

Ex1:

Number of pages in the process=8

Number of frames in the RAM=3

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

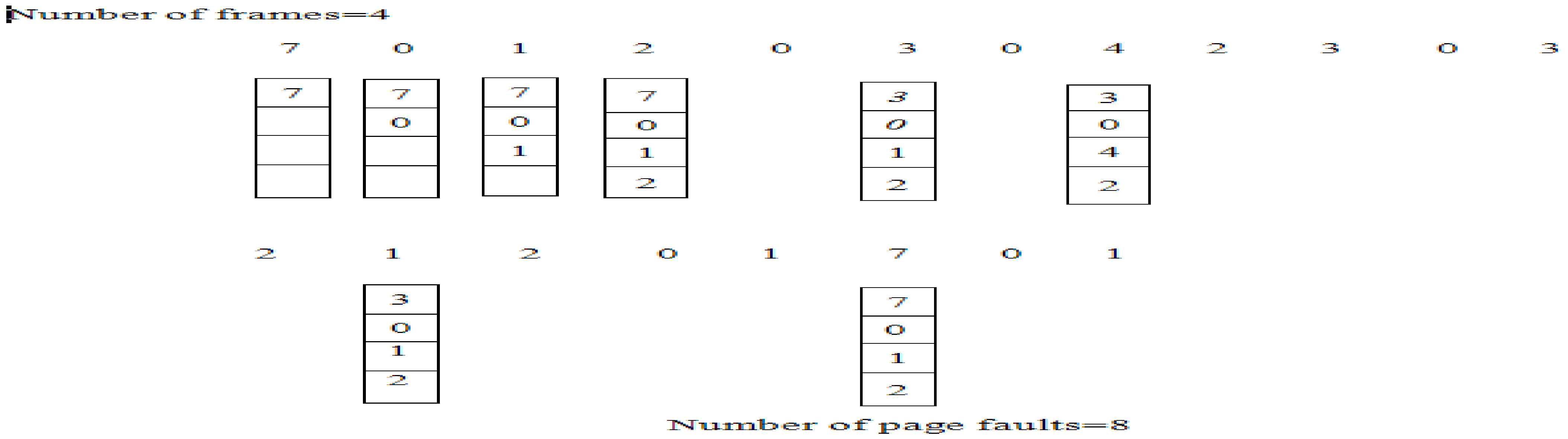


Ex2:

Number of pages in the process=8

Number of frames in the RAM=4

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1



Advantages:

1) Less number of page faults compared to FIFO.

2) Its implementation is easy as it depends only on previous references of pages.

Variants of LRU page replacement algorithm / LRU approximation algorithms

1. Second-Chance algorithm
2. Enhanced Second-Chance algorithm

Second-Chance algorithm

In this algorithm, a reference bit is maintained with each page loaded in the RAM.

The reference bit value of a page is ‘0’ if that page is not referenced. Otherwise, reference bit value is ‘1’.

When a page fault occurs, the reference bit of each page loaded in the RAM is checked.

This algorithm moves over the pages with reference bit value ‘1’.

While moving, sets the reference bit value of pages to ‘0’.

When a page with reference bit value ‘0’ is found then that page is replaced by the required page.

Ex: Number of pages in the program = 10 (0 to 9)

Number of frames in the RAM = 7 (0 to 6)

RAM

	OS	
RB = 1	Page 2	Frame 0
RB = 1	Page 4	Frame 1
RB = 1	Page 3	Frame 2
RB = 0	Page 0	Frame 3
RB = 1	Page 5	Frame 4
RB = 0	Page 1	Frame 5
RB = 0	Page 7	Frame 6

<u>Enhanced Second-Chance algorithm</u>

In this algorithm, a reference bit and a modify bit are maintained with each page loaded in the RAM. Four situations are possible:

(0, 0) - neither recently used nor modified - best page to replace.

(0, 1) - not recently used, but modified - not quite as good, because the page will need to be written out before replacement.

(1, 0) - recently used but clean - probably will be used again soon.

(1, 1) - probably will be used again, will need to write out before replacement.

When a page fault occurs, the page with reference bit value ‘0’ and modify bit value ‘0’ is replaced with the required page.

	MODIFY BIT	REFERENCE BIT	MEANING
CASE - 1	0	0	Page was neither recently used nor modified—best page to replace
CASE – 2	0	1	Page was recently referenced but not modified – hence a clean page
CASE – 3	1	0	not recently used but modified—needs to be written to secondary memory before replacement
CASE - 4	1	1	recently used and modified—probably will be used again soon, and needs to be written out to secondary storage before it can be replaced

Counting based page replacement algorithm

In this algorithm, a reference count is maintained with each page loaded in the RAM. Reference count indicates the number of times a page is referenced. If a page is not referenced after loading it into the RAM then its reference count is ‘0’. Two variants of counting based page replacement are

1. Least Frequently Used (LFU) page replacement algorithm
2. Most Frequently Used (MFU) page replacement algorithm

Least Frequently Used (LFU) page replacement algorithm

When a page fault occurs, the page with least reference count is replaced with the required page.

Most Frequently Used (LFU) page replacement algorithm

When a page fault occurs, the page with highest reference count is replaced with the required page.

<u>Allocation of frames</u>

To increase performance of the computer system, CPU of the computer system should be kept busy at all times.

To keep the CPU busy, number of programs needs to be loaded into RAM.

To load more programs into the RAM, frames in the RAM needs to be allocated to the programs.

The following points need to be considered to allocate frames of RAM to the programs.

1. Minimum number of frames allocated to each program
2. Allocation algorithms
3. Global versus Local Allocation
4. Non-Uniform Memory Access (NUMA)

Minimum number of frames

For each program to be loaded into the RAM, operating system should allocate frames in the RAM.

Operating system should allocate minimum two frames for each program.

One frame for loading at least one page of the program and another frame for loading the page table of the program.

Allocation Algorithms

Operating system uses two types of algorithms to allocate frames in the RAM to the programs.

1. Equal allocation
2. Proportional allocation

<u>Equal allocation</u>

With equal allocation, equal number of frames is allocated to all the programs to be loaded into the RAM.

If there are ‘m’ number of frames in the RAM and ‘n’ number of programs to be loaded into the RAM then ‘m/n’ number of frames is allocated to each program.

Two disadvantages with equal allocation are:

1. Some frames are unused when the number of pages in the program is less than the allocated ‘m/n’ number of frames. For example, if a program requires 4 frames and if 10 frames are allocated then 6 frames are unused.
2. More page faults will occur when the number of pages in the program is far more than the allocated ‘m/n’ number of frames. For example, if a program requires 20 frames and if only 3 frames are allocated then more number of page faults will occur.

<u>Proportional allocation</u>

Different number of frames is allocated to different programs depending on the size or priority or both of programs.

More number of frames is allocated to large programs and less number of frames is allocated to small programs.

Advantages:

1. Less number of page faults occur as adequate number of frames is allocated to each program.
2. No wastage of frames.

<u>Global versus Local Allocation</u>

<u>Local allocation</u>

While executing the program, if a page fault occurs then one of the frames allocated to the same program is selected to load the required page.

Global allocation

While executing the program, if a page fault occurs then one of the frames allocated to other programs is selected to load the required page.

Global allocation is the commonly used algorithm.

Non-Uniform Memory Access (NUMA)

In a computer system, processors and RAM components are placed either on the same system board or on different system boards.

If the RAM component in which the program is loaded and the processor to which the program is assigned for execution are on the same system board then the processor can access the program in less time.

Otherwise, more time is required to access the program by the processor.

To execute a program in less time, operating system has to load the program in the RAM component placed on system board on which the processor is placed.

<u>Thrashing</u>

To increase the performance of computer system, number of programs is loaded into the RAM at a time.

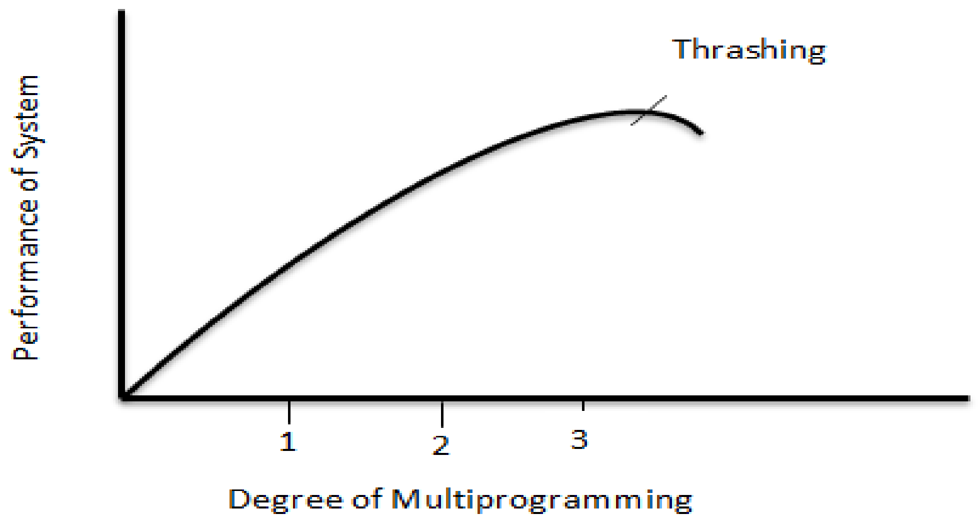
Degree of multiprogramming indicates the number of programs loaded at a time into the RAM.

Increasing the degree of multiprogramming also increases the number of page faults.

When the number of page faults is increased then most of the time will be spent on moving the pages from Hard Disk to RAM instead of executing the pages.

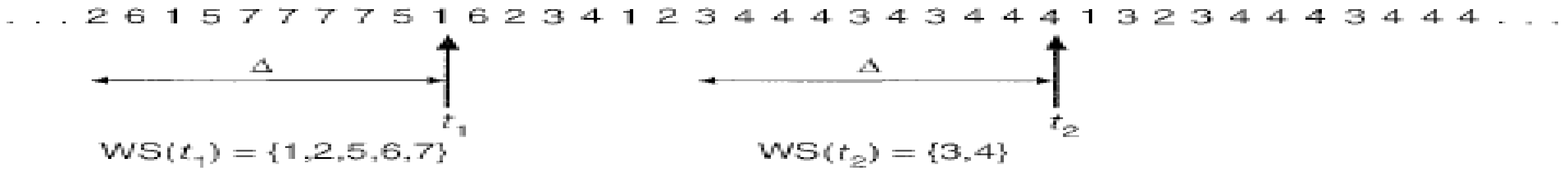
When the number of page faults is increased to a maximum level then the performance of computer system is suddenly dropped.

The sudden drop in the performance of computer system is called thrashing.



Working-set model

In the working-set model, a working-set window is maintained. The working-set window moves on the set of page references or reference string as shown below.



Size of working-set window is determined by the operating system.

In above diagram, the size of working-set window is taken as 10. i.e. the working-set window at any particular time covers 10 pages in the reference string.

At any particular time, the active pages covered by the working-set window is indicated through a set called working-set (WS).

The above diagram shows the pages in the working-set at time t1 and at time t2.

The number of pages in the working-set determines the minimum number of frames required by the program at any particular time.

In above diagram, the program requires 5 frames at time t1 and 2 frames at time t2.

Based on the working-set, the operating system from time to time determines the number of frames required by the program and allocates the required number of frames to the program to reduce the number of page faults.

FILE MANAGEMENT

File

A file is a container of information that is stored in the secondary storage device (Hard disk).

Data cannot be written to disk unless they are within a file.

Files are broadly categorised into program files and data files.

Data files may contain numeric, alphabetic, alphanumeric or binary information.

<u>File Attributes</u>

A file is associated with a set of attributes.

A file’s attributes vary from one operating system to another.

The general attributes associated with a file are

1) Name: a file has a name and is in human readable form.

2) Identifier: it is a unique number assigned by operating system. The operating system identifies a file by its identifier.

3) Type: indicates the type of information stored in the file.

4) Location: indicates the location or address of the file in the disk.

5) Size: indicates the current size of the file in bytes or words or blocks.

6) Protection: indicates the access permissions for different users.

7) Time, date and user identification: indicates the owner of file and the date and time on which the file is created and last modified.

File Operations

The six basic operations performed on a file are

1) Creating a file: to create a new file, the operating system has to allocate space for the file and create an entry for the file in the directory.

2) Writing a file: to write to a file, name of the file and the information to be written should be specified. The operating system identifies the file and writes the information into the file at the position specified by file pointer and moves the file pointer.

3) Reading a file: to read from a file, name of the file should be specified. The operating system identifies the file and reads the data from the position pointed by the file pointer and moves the file pointer.

4) Repositioning within a file: repositions the file pointer.

5) Deleting a file: the operating system releases the file space and erase the directory entry of the file.

6) Truncating a file: the contents of the file are deleted but the space allocated to the file is not released.

Other operations that can be performed on a file are

1) appending new information to the end of an existing file

2) renaming an existing file

3) create a copy of a file

4) copy the file to another i/o device

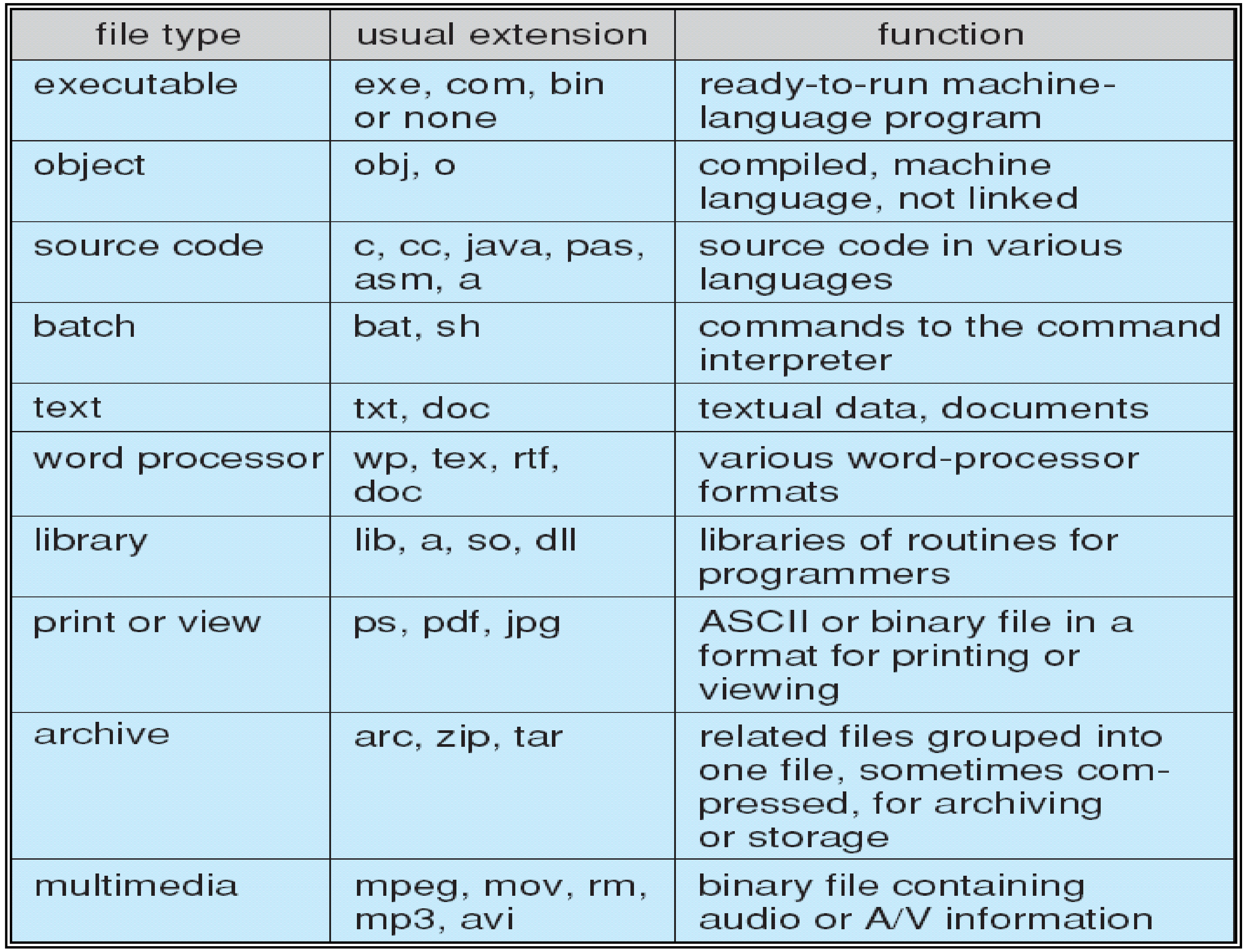
5) determining the status of a file

<u>File Types</u>

File type is indicated through the name of file.

The file name is split into two parts: name and extension, separated by a period character.

Based on the file type, the operating system decides the operations that can be performed on the file.



file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes com-pressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

Access Methods

Files store information.

The information in the file can be accessed in several ways.

Some systems provide only one access method for files.

Some systems support many access methods.

The different access methods are
1) Sequential Access
2) Direct Access
3) Indexed Access

Sequential Access

Information in the file is accessed in sequential order i.e. one record after the other.

Compilers and editors access files in this fashion.

Most frequently performed operations on a file are read and write.

The syntax of read operation is

```
read next
```

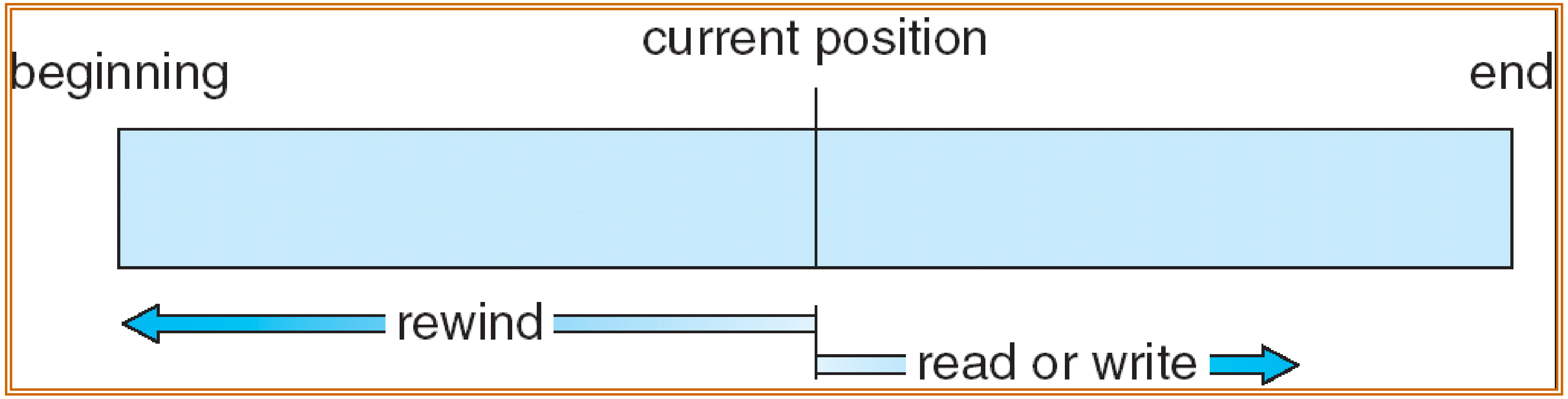
reads the data pointed by file pointer and then advances the file pointer.

The syntax of write operation is

```
write next
```

writes the data at the position pointed by file pointer and then advances the file pointer.

Sequential access is depicted in the following figure.



<u>Direct Access</u>

Another name of direct access is relative access.

With direct access, any record or block of the file can be accessed directly.

For example, we may read block 14, then read block 53, and then write block 8.

Data from databases is generally accessed using direct access method.

The syntax of read and write operations is

```
read n
write n
```

where ‘n’ is the block number.

The block number ‘n’ is a relative block number. Some systems start the relative block number from ‘0’; others start at ‘1’.

Not all operating systems support both sequential and direct access for files.

Some systems allow only sequential file access; others allow only direct access.

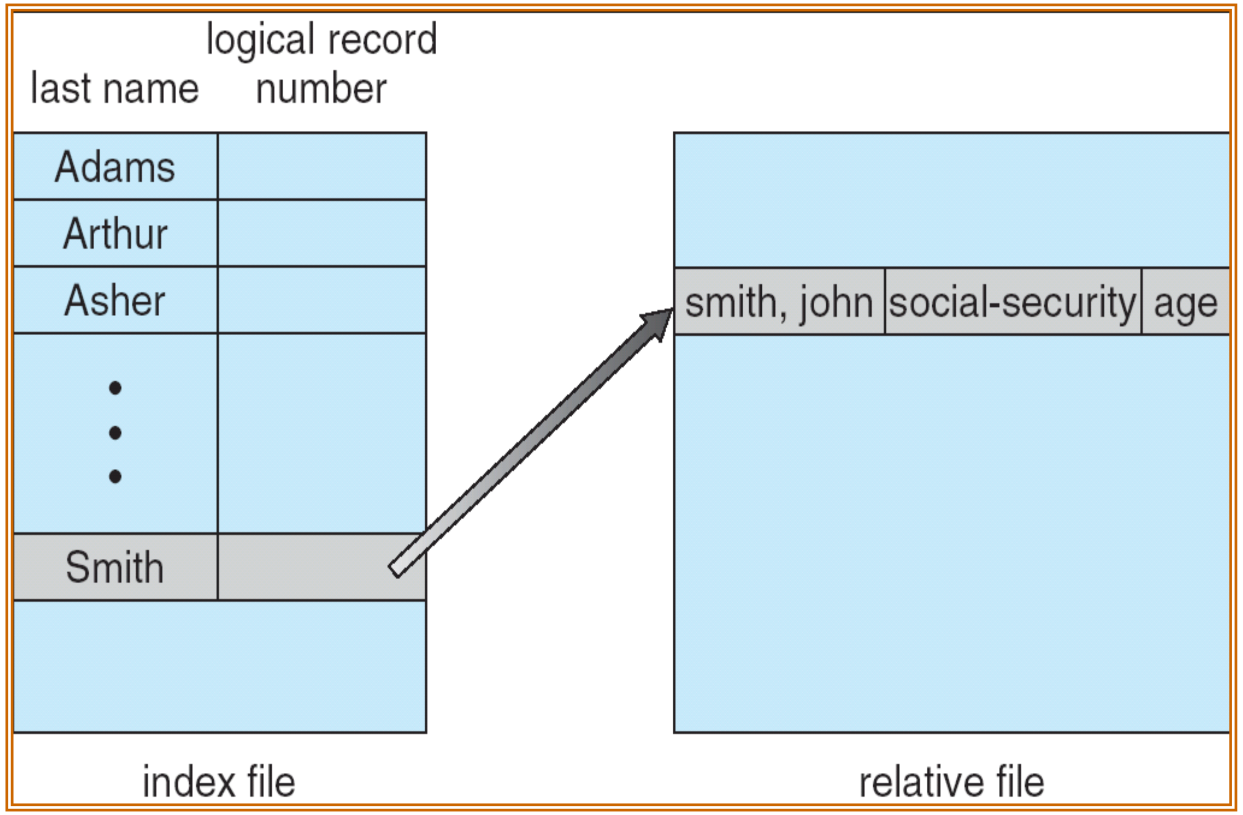
Indexed Access

Indexed access method is developed based on direct access method.

An index is created for the file. The index contains pointers to blocks of the file.

To access a block of the file, the index is searched first and then the pointer in the index is used to access the block.

The following figure shows how the data in the file is accessed using the indexed access method.


logical record number
last name
Adams
Arthur
Asher
Smith
smith, john
social-security
age
index file
relative file

The index is searched using binary search method.

When the file is large then the index file becomes large.

In this case, the index file is maintained in two levels.

The primary index file contains pointers to secondary index files.

The secondary index files point to the actual data items.

Storage Structure

Files are stored on random-access storage devices, including hard disks, optical disks, and solid state disks.

A storage device can be used in its whole or can be divided into parts.

For example, a disk can be partitioned into quarters (drives).

Storage devices can also be collected together into RAID sets that provide protection from the failure of a single disk.

Partitions are also known as slices or minidisks.

A file system (FAT, GFS, HFS, NTFS, UDF) can be created on each of these parts of the disk.

Any entity containing a file system is generally known as a volume.

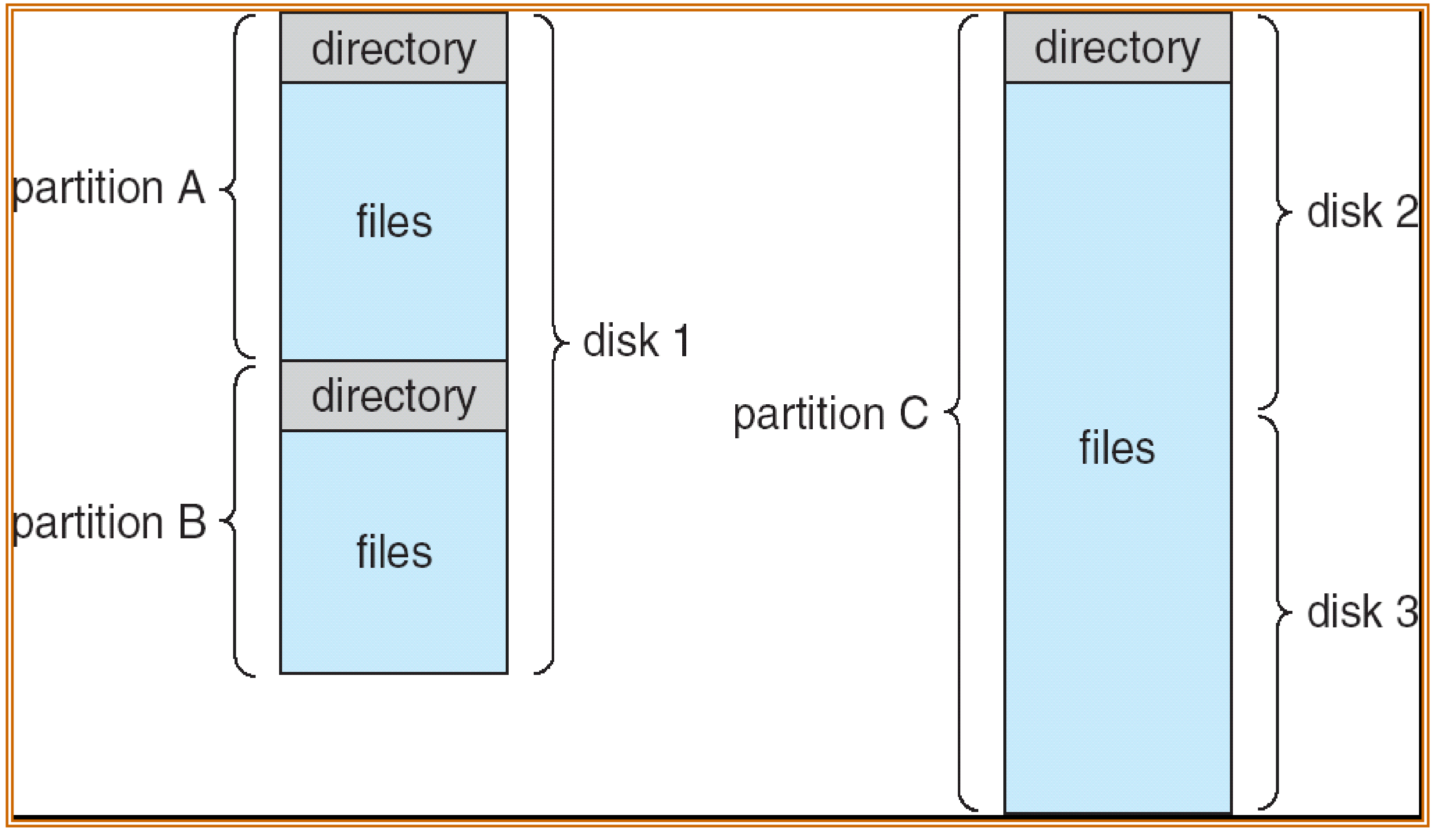
Volumes can also store multiple operating systems, allowing a system to boot and run more than one operating system.

Each volume that contains a file system must also contain information about the files in the system.

This information is kept in the entries in a directory.

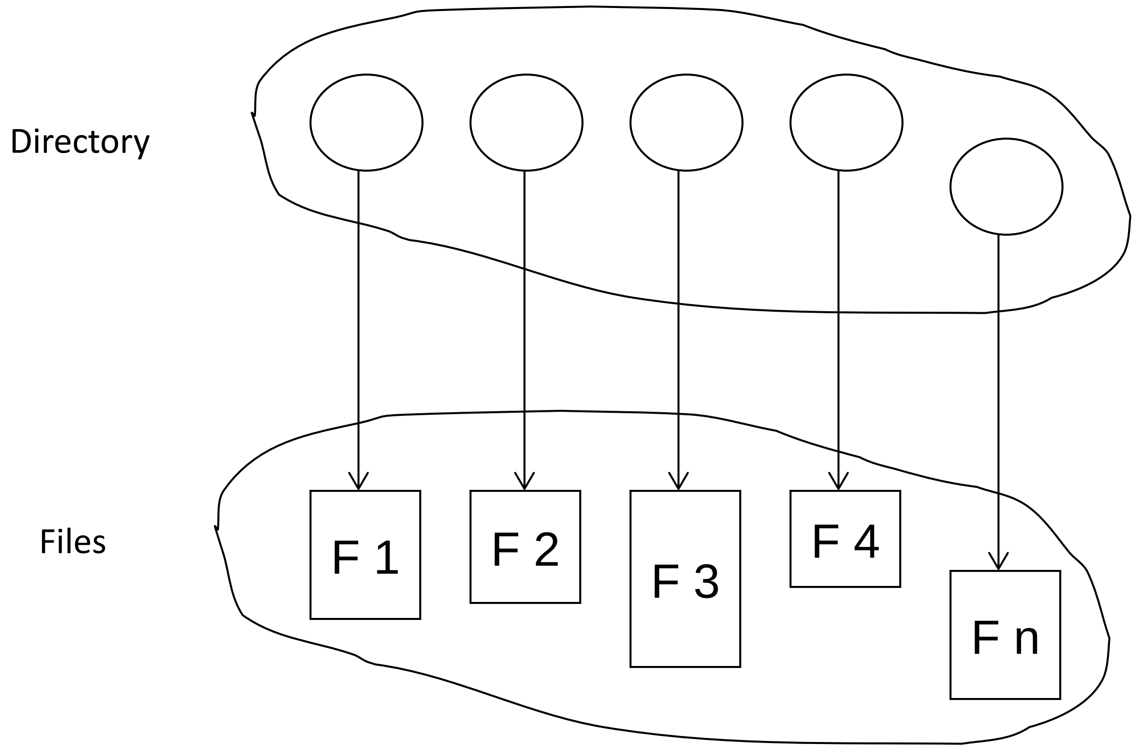
The directory simply records information such as name, location, size, and type for all files on that volume.

Following figure shows a typical file-system organization.



Directory Overview/Structure

A directory is collection of nodes or entries containing information about all files in that directory.



There are different structures for directories. When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:

1) Search for a file: We need to be able to search a directory structure to find the entry for a particular file.

2) Create a file: New files need to be created and added to the directory.

3) Delete a file: When a file is no longer needed, we want to be able to remove it from the directory.

4) List a directory: We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.

5) Rename a file: Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes.

6) Traverse the file system: We may wish to access every directory and every file within a directory structure.

<u>Different Structures of Directory</u>

1) Single-Level Directory

2) Two-Level Directory

3) Tree-Structured Directory

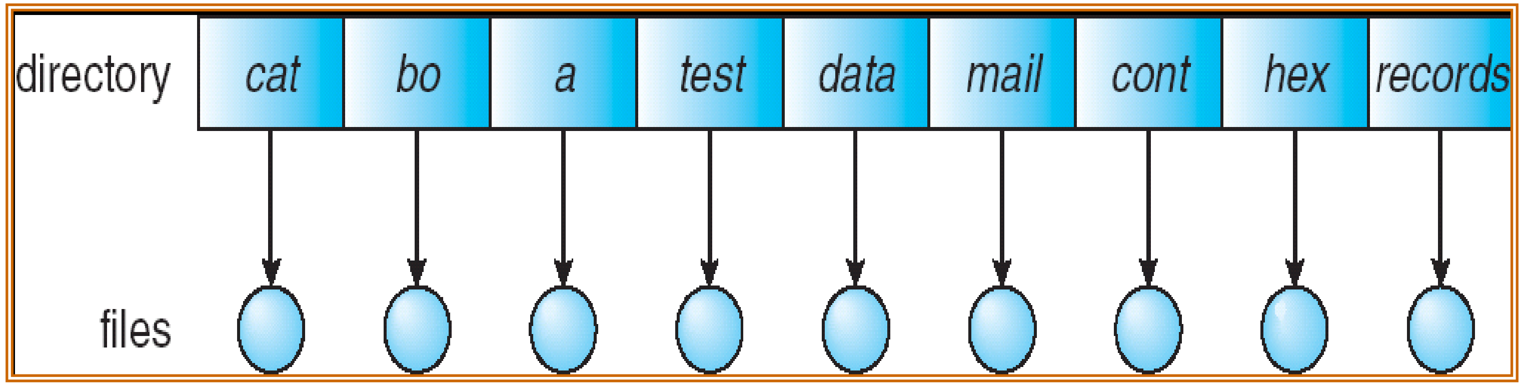
4) Acyclic-Graph Directory

5) General-Graph Directory

Single-Level Directory

The simplest directory structure is the single-level directory.

All files are contained in the same directory as shown in following figure.



Limitations with Single-Level directory:

1) Since all files are in the same directory, they must have unique names. If two users call their data file *test,* then the unique-name rule is violated.

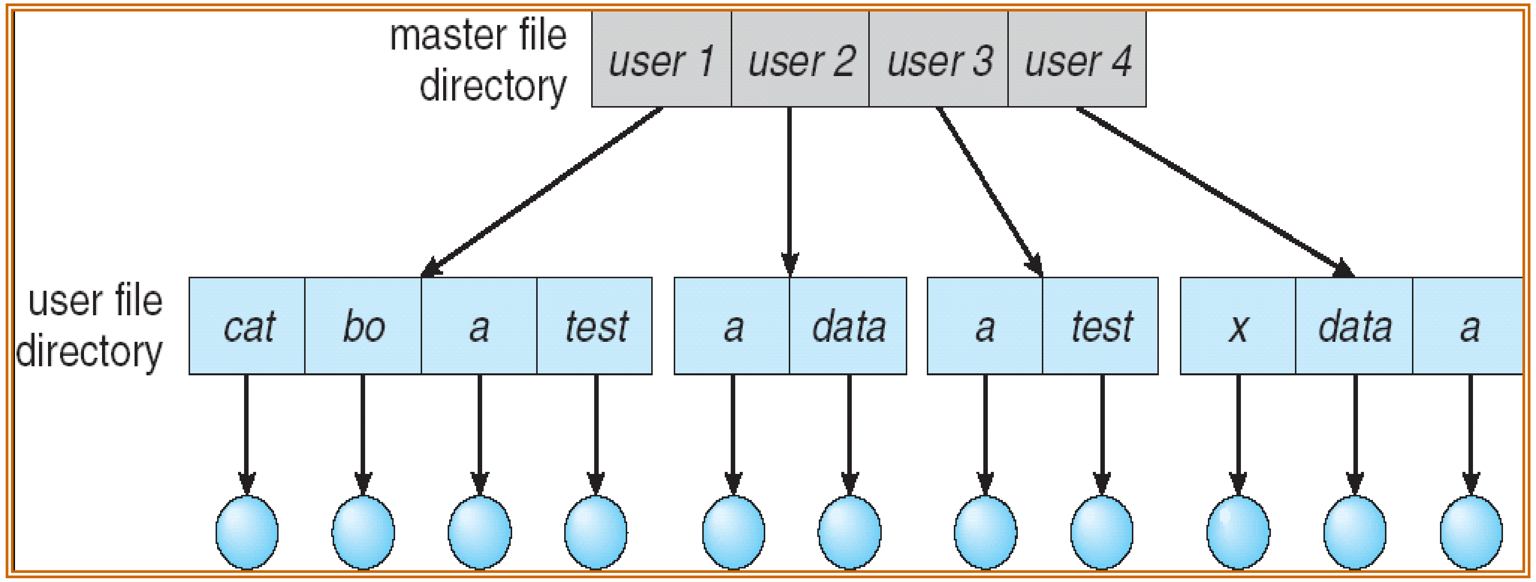
2) Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases.

Two-Level Directory

In the two-level directory structure, each user has his own User File Directory (UFD).

The UFDs have similar structure, but each lists only the files of a single user.

The MFD is indexed by user name or account number, and each entry points to the UFD for that user as shown in below figure.



Different users may have files with the same name, as long as all the file names within each UFD are unique.

To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists.

To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

Advantage:
solves the name-collision problem

Disadvantages:
This structure isolates one user from another. Isolation is an advantage when the users are completely independent but is a disadvantage when the users *want* to cooperate on some task and to access one another's files.

A two-level directory can be thought of as a tree of height 2.

The root of the tree is the MFD.

Its direct descendants are the UFDs.

The descendants of the UFDs are the files themselves.

The files are the leaves of the tree.

To name a particular file uniquely in a two-level directory, we must give both the user name and the file name.

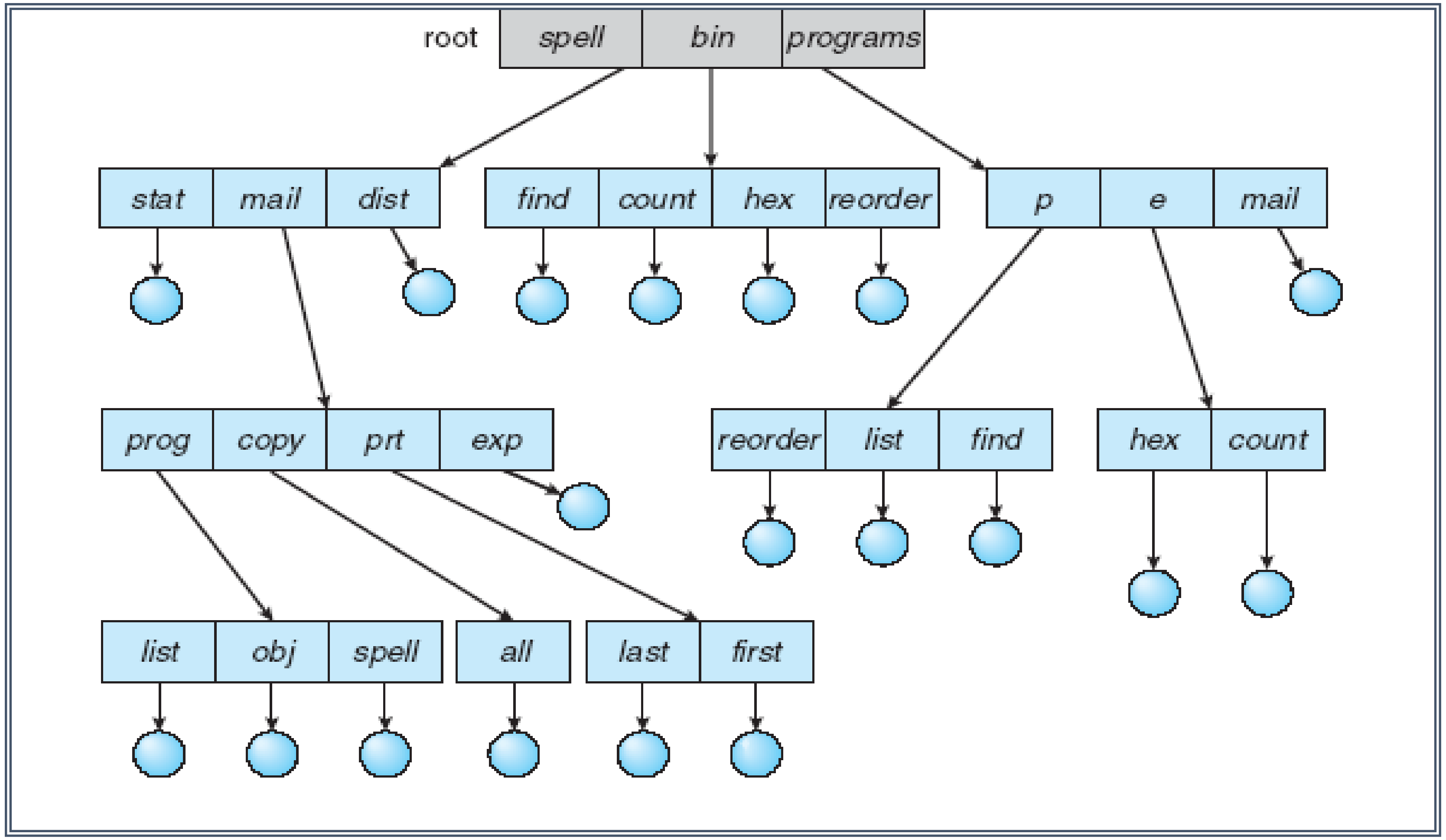
Thus, a user name and a file name define a *path name.*

Tree-Structured Directory

A tree is the most common directory structure.

Tree structure allows users to create their own subdirectories and to organize their files accordingly.

The tree has a root directory, and every file in the system has a unique path name.


root
spell
bin
programs
stat
mail
dist
find
count
hex
reorder
p
e
mail
prog
copy
prt
exp
reorder
list
find
hex
count
list
obj
spell
all
last
first

A directory (or subdirectory) contains a set of files or subdirectories.

All directories have the same internal format.

One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1).

Path names can be of two types: *absolute* and *relative.*

An absolute path name begins at the root and follows a path down to the specified file, giving the directory names on the path.

A relative path name defines a path from the current directory.

For example, in the tree-structured file system of above figure, if the current directory is *root/spell/mail,* then the relative path name *prt/first* refers to the same file as does the absolute path name *root/spell/mail/prt/first.*

An interesting policy decision in a tree-structured directory concerns how to handle the deletion of a directory.

If a directory is empty, its entry in the directory that contains it can simply be deleted.

However, suppose the directory to be deleted is not empty but contains several files or subdirectories. One of two approaches can be taken.

Some systems, such as MS-DOS, will not delete a directory unless it is empty.

Thus, to delete a directory, the user must first delete all the files in that directory.

If any subdirectories exist this procedure must be applied recursively to them, so that they can be deleted also.

This approach can result in a substantial amount of work.

An alternative approach, such as that taken by the UNIX rm command, is to provide an option: when a request is made to delete a directory, all that directory's files and subdirectories are also to be deleted.

The latter policy is more convenient, but it is also more dangerous, because an entire directory structure can be removed with one command.

If that command is issued in error, a large number of files and directories will need to be restored (assuming a backup exists).

With a tree-structured directory system, users can be allowed to access, in addition to their files, the files of other users.

For example, user B can access a file of user A by specifying its path name.

User B can specify either an absolute or a relative path name.

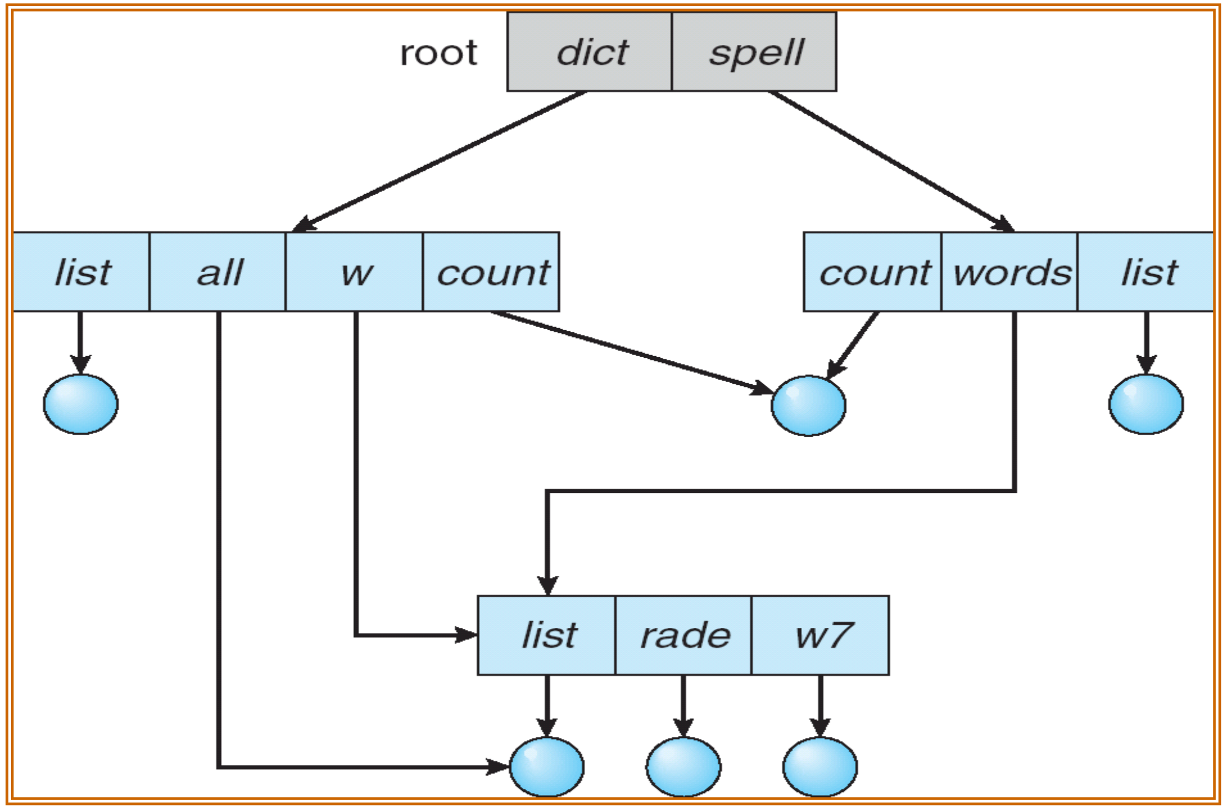
Acyclic-Graph Directory

When two or more programmers are working on a joint project then the files associated with that project can be stored in a subdirectory.

The common subdirectory should be *shared* by all programmers*.*

A tree structure does not allow the sharing of files or directories.

An acyclic graph allows directories to share subdirectories and files as shown in below figure.


root
dict
spell
list
all
w
count
count
words
list
list
rade
w7

The *same* file or subdirectory may be in two different directories.

A shared file (or directory) is not the same as two copies of the file.

With a shared file, only *one* actual file exists, so any changes made by one person are immediately visible to the other.

When a new file is created by one person then that file will automatically appear in all the shared subdirectories.

Disadvantages

A file may have multiple absolute path names.

Consequently, distinct file names may refer to the same file.

When we traverse the file system then the same file is traversed more than once.

Another problem involves deletion. When can the space allocated to a shared file be deallocated and reused?

One possibility is to remove the file whenever anyone deletes it, but this action may leave dangling pointers to the now-nonexistent file.

Another approach to deletion is to preserve the file until all references to it are deleted.

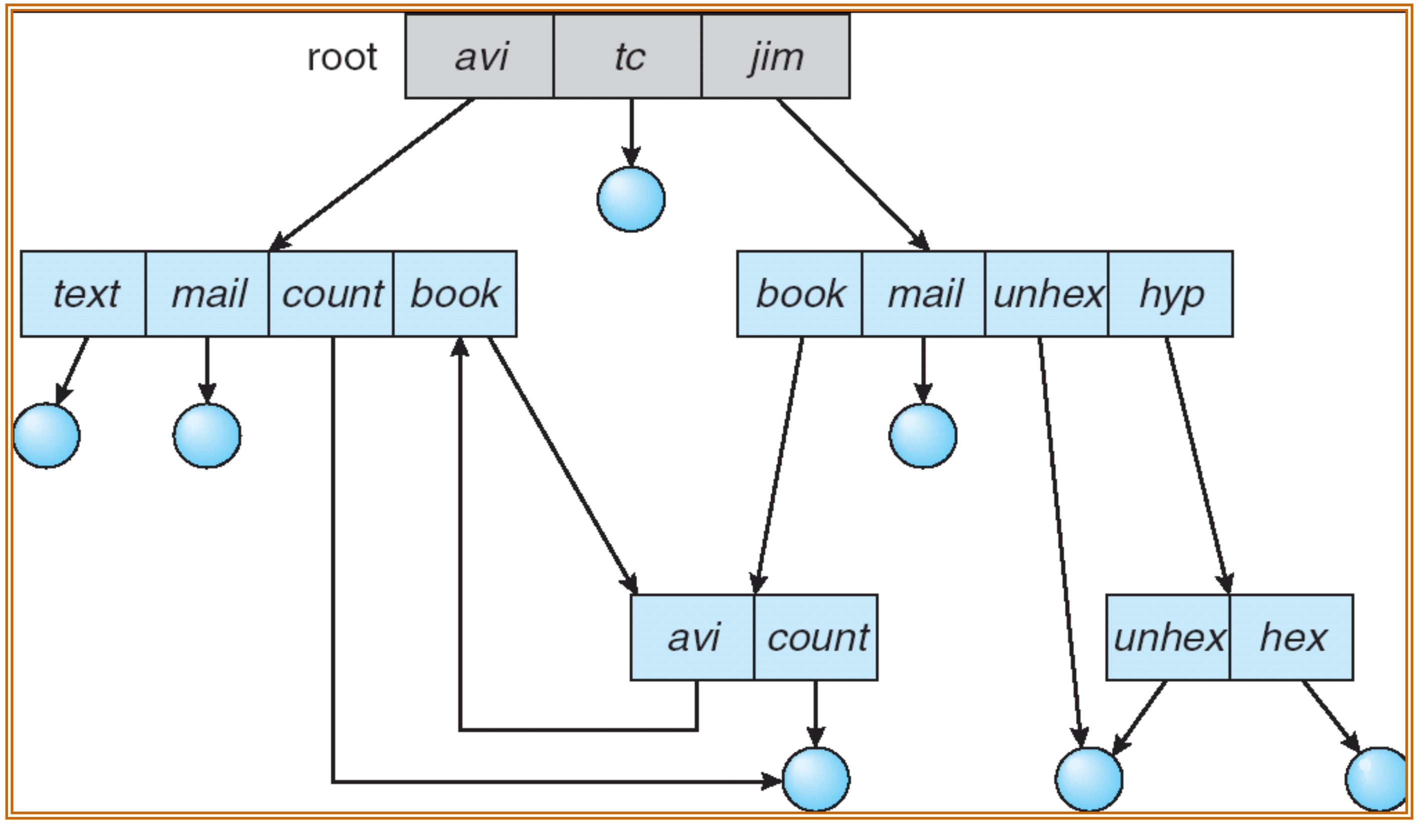
To implement this approach, we need to keep a count of the *number* of references.

General Graph Directory

If we start with a two-level directory and create subdirectories then a tree-structured directory is created.

When we add new files and subdirectories to an existing tree-structured directory the tree-structure is preserved.

But when we add links then the tree structure is destroyed and results in a simple graph structure.


root
avi
tc
jim
text
mail
count
book
book
mail
unhex
hyp
avi
count
unhex
hex

One problem with graph structure is that the traversing program may enter into an infinite loop.

Another problem is how to decide on when a file can be deleted.

With acyclic-graph directory structures, a value of 0 in the reference count means that there are no more references to the file or directory, and the file can be deleted.

However, when cycles exist, the reference count may not be 0 even when it is no longer possible to refer to a directory or file.

In this case, we generally need to use a garbage-collection scheme to determine when the last reference has been deleted and the disk space can be reallocated.

Garbage collection involves traversing the entire file system, marking everything that can be accessed.

Then, a second pass collects everything that is not marked onto a list of free space.

Key Differences:

Feature	Acyclic Graph Directory	General Graph Directory
Cycles	No cycles (graph is acyclic)	Cycles are allowed (graph can be cyclic)
Parent-Child Relationships	A file can have multiple parent directories, but no circular references	A file can have multiple parents, and cycles are allowed
Use case	Used when flexibility is needed, but without loops	Used when more complex relationships or cycles are needed
Complexity	Easier to manage due to no cycles	More complex due to the possibility of cycles
Traversal	Traversal avoids loops and is more predictable	Can lead to infinite loops if not managed correctly

File System Structure

The file system provides the mechanism for storage and access to file contents.

The file system resides permanently on *secondary storage,* which is designed to hold a large amount of data permanently.

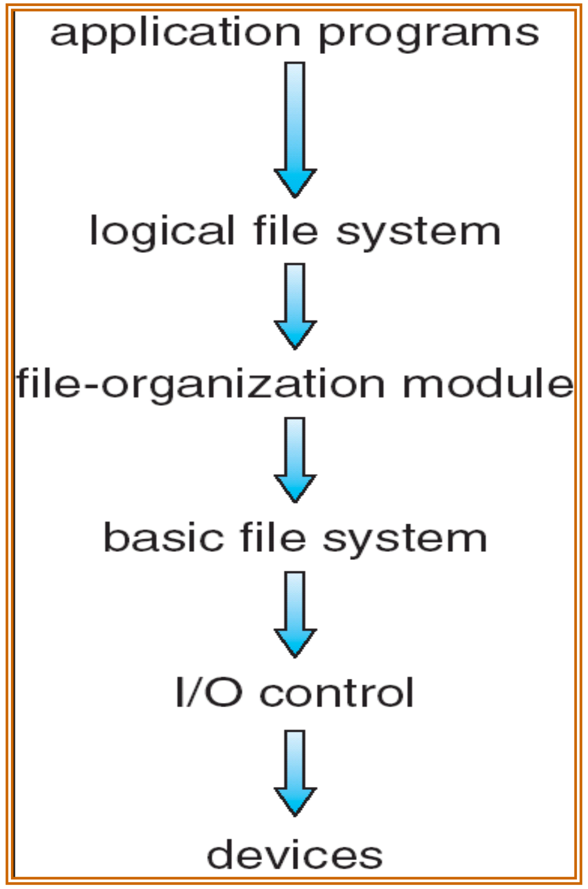
Disks provide the bulk of secondary storage on which a file system is maintained.

File Systems provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily.

The file system is composed of many different levels.

The structure shown in below figure is an example of a layered design.

Each level uses the features of lower levels to create new features for use by higher levels.


application programs
logical file system
file-organization module
basic file system
I/O control
devices

The lowest level, **the *I/O control*,** consists of device drivers to transfer information between the main memory and the disk system.

Input to device driver is a high-level command like "retrieve block 123."

The device driver passes low level, hardware-specific instructions to the hardware controller.

The hardware controller interacts with the device and performs the operation.

The ***basic file system*** issues generic commands to the device driver to read and write physical blocks on the disk.

basic file system layer also manages the memory buffers.

A block in the buffer is allocated before the transfer of a disk block can occur.

When the buffer is full, the buffer manager must free up buffer space to allow a requested I/O to complete.

The ***file-organization module*** knows about files and their logical blocks, as well as physical blocks.

The file-organization module translates logical block addresses to physical block addresses for the basic file system to transfer.

Each file's logical blocks are numbered from 0 (or 1) through *N.*

Since the physical blocks containing the data usually do not match the logical numbers, a translation is needed to locate each block.

The file-organization module also includes the ***free-space manager***, which tracks unallocated blocks and provides these blocks to the file-organization module when requested.

Finally, the ***logical file system*** manages metadata information.

Metadata includes all of the file-system structure except the actual *data* (or contents of the files).

The *logical file system* manages the directory structure. It maintains file structure via file-control blocks.

A file control block (FCB) contains information about the file, including ownership, permissions, and location of the file contents.

The *logical file system* is also responsible for protection and security.

When a layered structure is used for file-system implementation, duplication of code is minimized.

The *I/O control* and sometimes the *basic file-system* code can be used by multiple file systems.

Each file system can then have its own *logical file-system* and *file-organization modules*.

Unfortunately, layering can introduce more operating system overhead, which may result in decreased performance.

Many file systems are in use today. Most operating systems support more than one.

Each operating system has one or more disk based file systems. UNIX uses the *UNIX File System (UFS)*.

Windows NT, 2000, and XP support disk file-system formats of FAT, FAT32, and NTFS (or Windows NT File System).

Although Linux supports over forty different file systems, the standard Linux file system is known as the *extended file system* with the most common versions being *ext2* and *ext3*.

Google created its own file system to meet the company's specific storage and retrieval needs.

Allocation Methods

Many files are stored on the same disk.

To allocate space to these files three major methods are in wide use: contiguous, linked, and indexed.

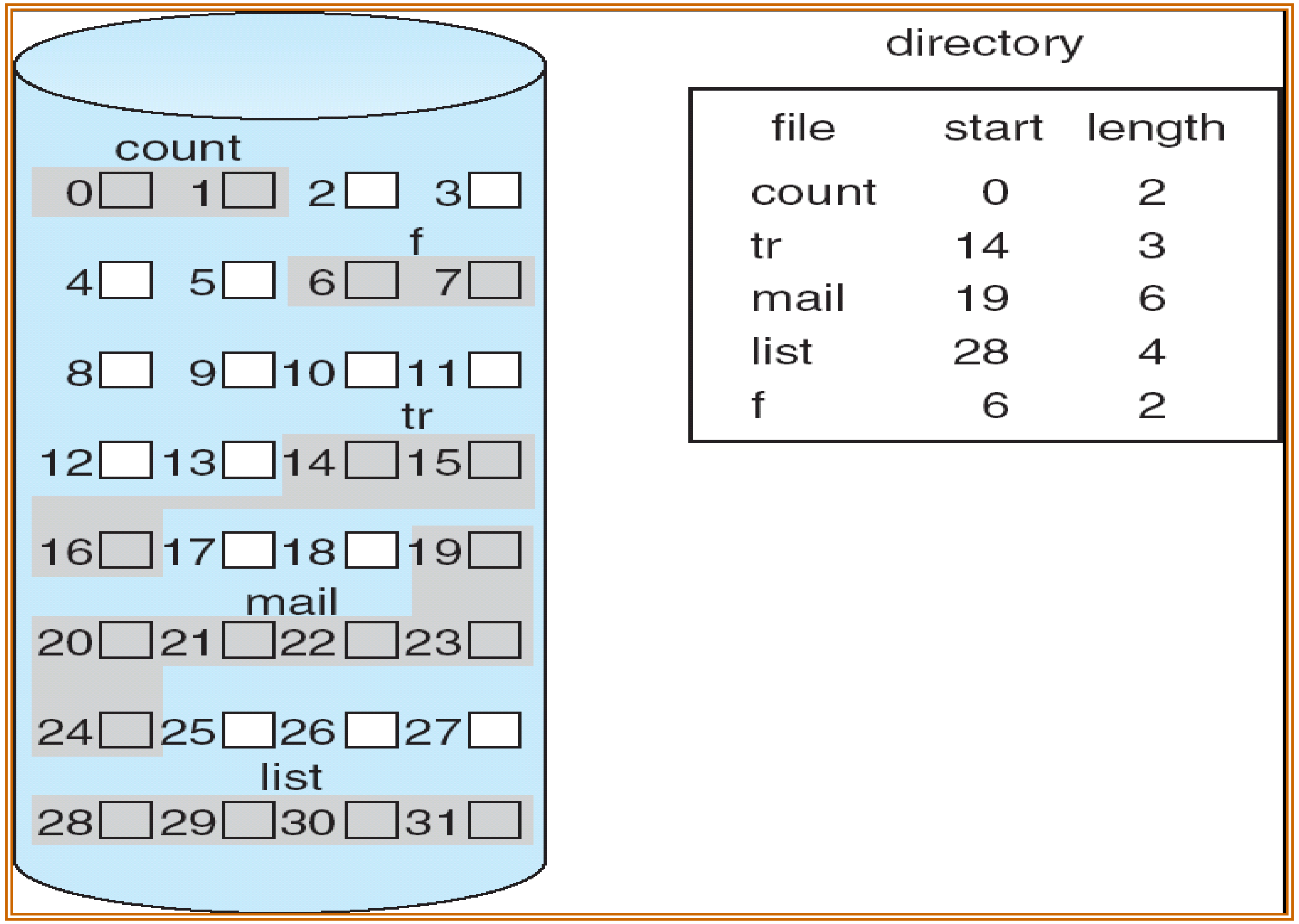
Each method has advantages and disadvantages.

Some operating systems support all three.

An operating system uses one method for all files within a file-system type.

Contiguous Allocation

A set of contiguous blocks is allocated to each file.



file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

If the file is n blocks long and starts at location b, then it occupies blocks b, b+1, b+2, ... ,b+n-1.

The directory entry for each file indicates the address of the starting block and the number of blocks allocated for that file.

<u>Advantage</u>

Supports both sequential and direct access.

<u>Disadvantages</u>

1) External fragmentation

As files are allocated and deleted, the free disk space is broken into pieces.

External fragmentation exists whenever free space is broken into chunks and when the largest contiguous chunk is insufficient for a request.

Compaction technique can be used to solve the external fragmentation problem.

Compaction technique compact all free space into one contiguous space.

But, this compaction process requires lot of time.

2) Another problem with contiguous allocation is determining how much space is needed for a file.

If too little space is allocated to a file then the file cannot be extended.

If more space is allocated then some space may be wasted (internal fragmentation).

To minimize these drawbacks, some operating systems use a modified contiguous-allocation scheme.

In this scheme, a contiguous chunk of space is allocated initially; then, if that space is not enough, another chunk *of* contiguous space, known as an *extent* is added.

The directory entry of the file now contains address of the starting block, block count, plus address of first block of the next extent.

Linked Allocation

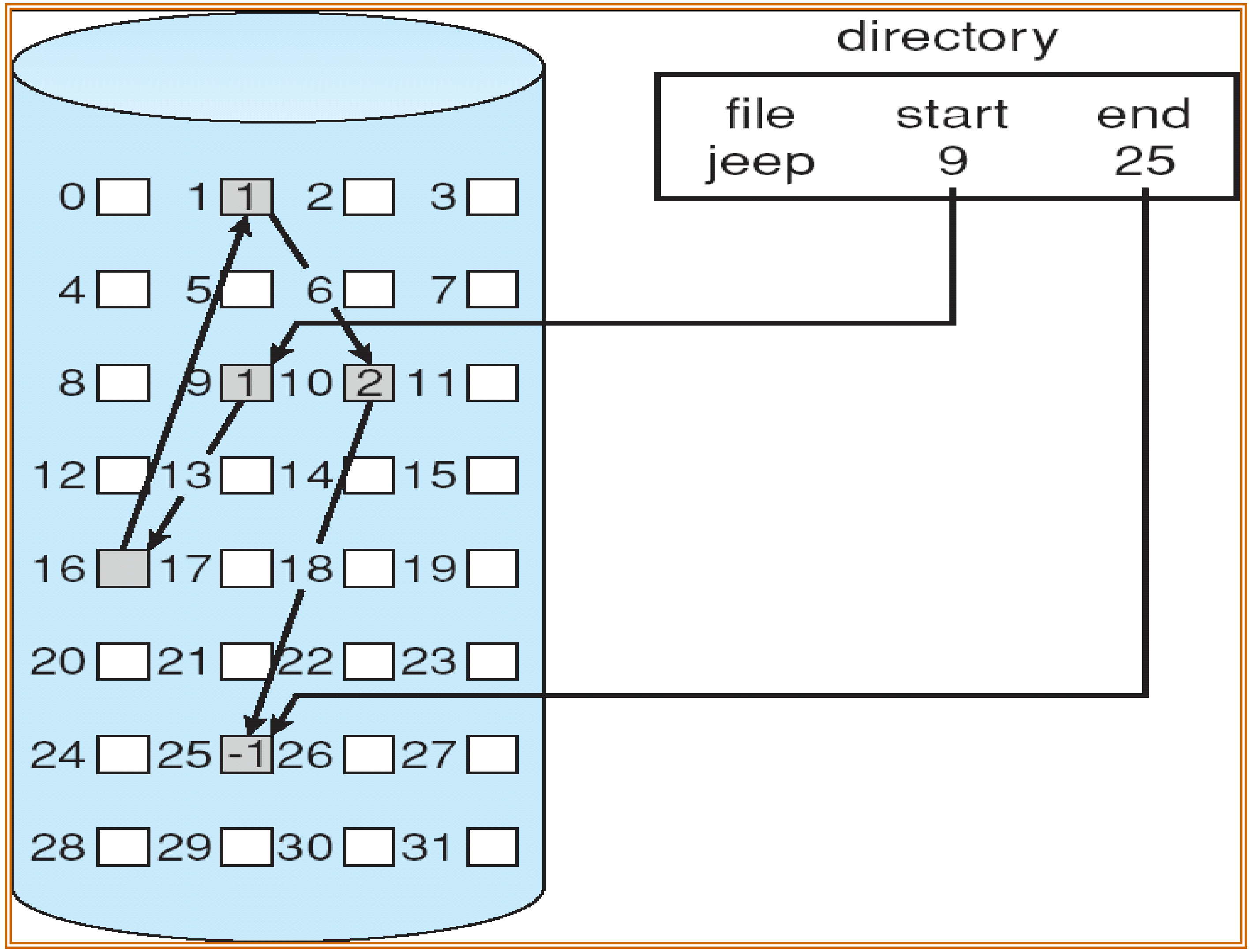
Linked allocation solves all problems of contiguous allocation.

With linked allocation, the blocks at any position of the disk can be allocated to a file.

For example, a file of five blocks may start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25.

Each block allocated to the file contains a pointer to the next block allocated to the file.

The directory entry of a file contains a pointer to the first and last blocks of the file.


directory
file
jeep
start
9
end
25
0
1
1
2
3
4
5
6
7
8
9
1
10
2
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
-1
26
27
28
29
30
31

Advantages

1) There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request.

2) The size of a file need not be declared when that file is created. A file can continue to grow as long as free blocks are available.

Disadvantages

1) Does not support direct access. To find the ith block of a file, we must start at the beginning of that file and follow the pointers until we get to the ith block.

2) Another disadvantage is the space required for the pointers.

One solution to this problem is to collect blocks into multiples, called *clusters* and to allocate clusters rather than blocks.

For example, a cluster is defined as four blocks. Pointers then use a much smaller percentage of the file's disk space.

Cluster mechanism improves disk throughput and decreases the space needed for free-list management.

But, this approach increases internal fragmentation, because more space is wasted when a cluster is partially full than when a block is partially full.

3) Another problem with linked allocation is reliability. If a block allocated to a file is corrupted then it is not possible to access the remaining blocks of the file.

Indexed Allocation

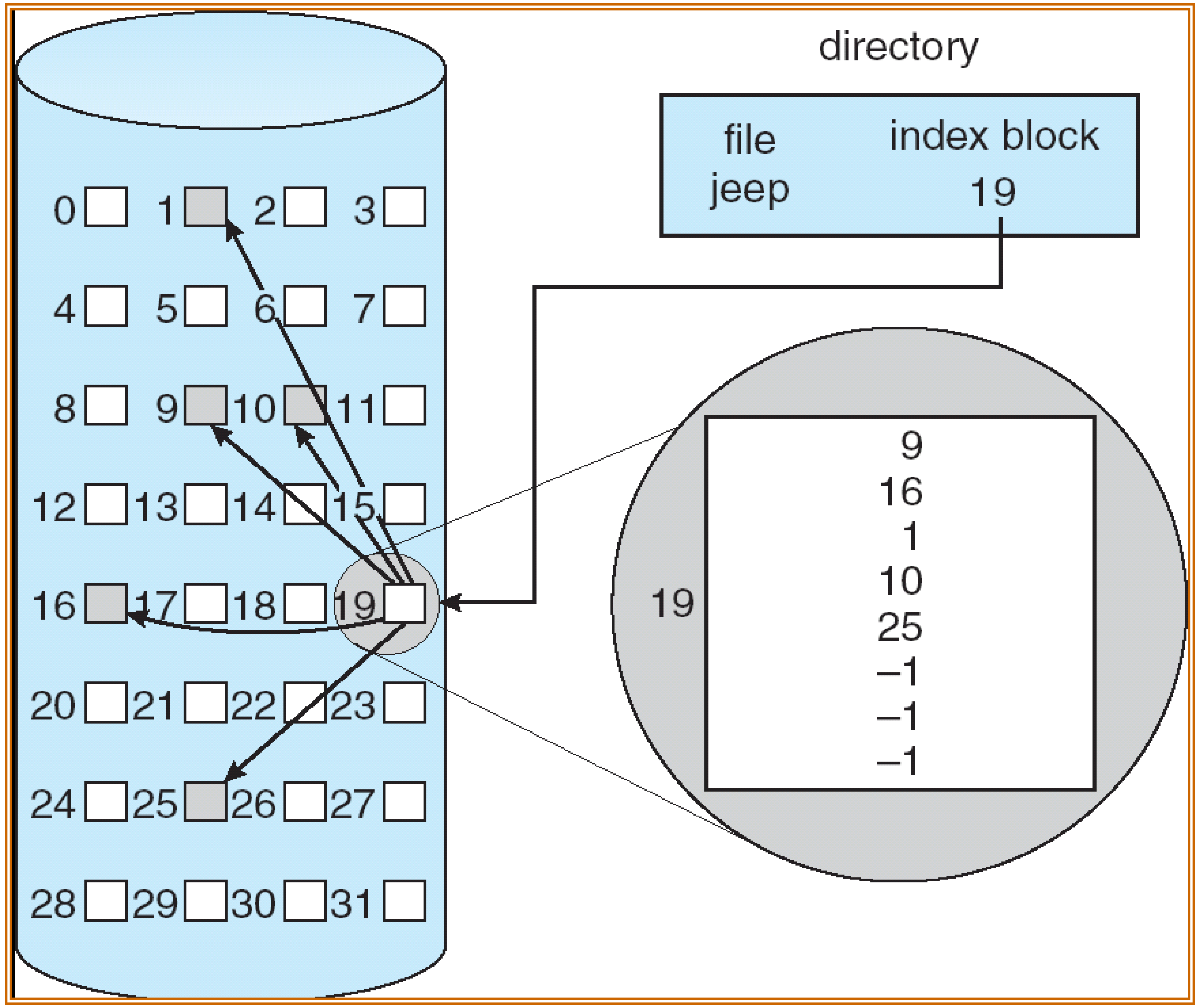
Blocks at any position of the disk can be allocated to a file as in linked allocation method.

The addresses of these blocks are stored into another block called *index block.*

Each file has its own index block, which is an array of disk-block addresses.

The *ith* entry in the index block points to the *ith* block of the file.

The directory entry of the file contains the address of the *index block*.


directory
file
index block
jeep
19
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
19
9
16
1
10
25
−1
−1
−1

To find and read the *ith block*, we use the pointer in the *ith* entry of *index block*.

When the file is created, all pointers in the index block are set to *nil.*

When the *ith* block is first written, a block is obtained from the free-space manager and its address is put in the *ith* entry of *index block*.

Advantages

1) No external fragmentation
2) Files can grow
3) Supports direct access
4) Reliability is more

Disadvantages

1) Wastage of space in index block

Consider a file which occupies only one or two blocks. With linked allocation, we lose the space of only one pointer per block.

With indexed allocation, an entire index block must be allocated, even if only one or two pointers will be non-nil.

2) Size of file is limited to no of pointers in the index block.

The following mechanisms are used to overcome this drawback

1. Linked scheme
2. Multilevel index

Linked scheme

If one index block is not enough to store the addresses of blocks allocated to a file then number of index blocks are allocated to the file and they are linked together.

The last entry in first index block contains the address of second index block.

The last entry in second index block contains the address of third index block and so on.

Multilevel index

This scheme uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks.

To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block.

This approach could be continued to a third or fourth level, depending on the file size.

Combined scheme

This scheme is used in the UFS. In this scheme, the first 15 pointers of the index block are stored in the file's *inode*.

The first 12 of these pointers point to *direct blocks*; that is, they contain addresses of blocks that contain data of the file.

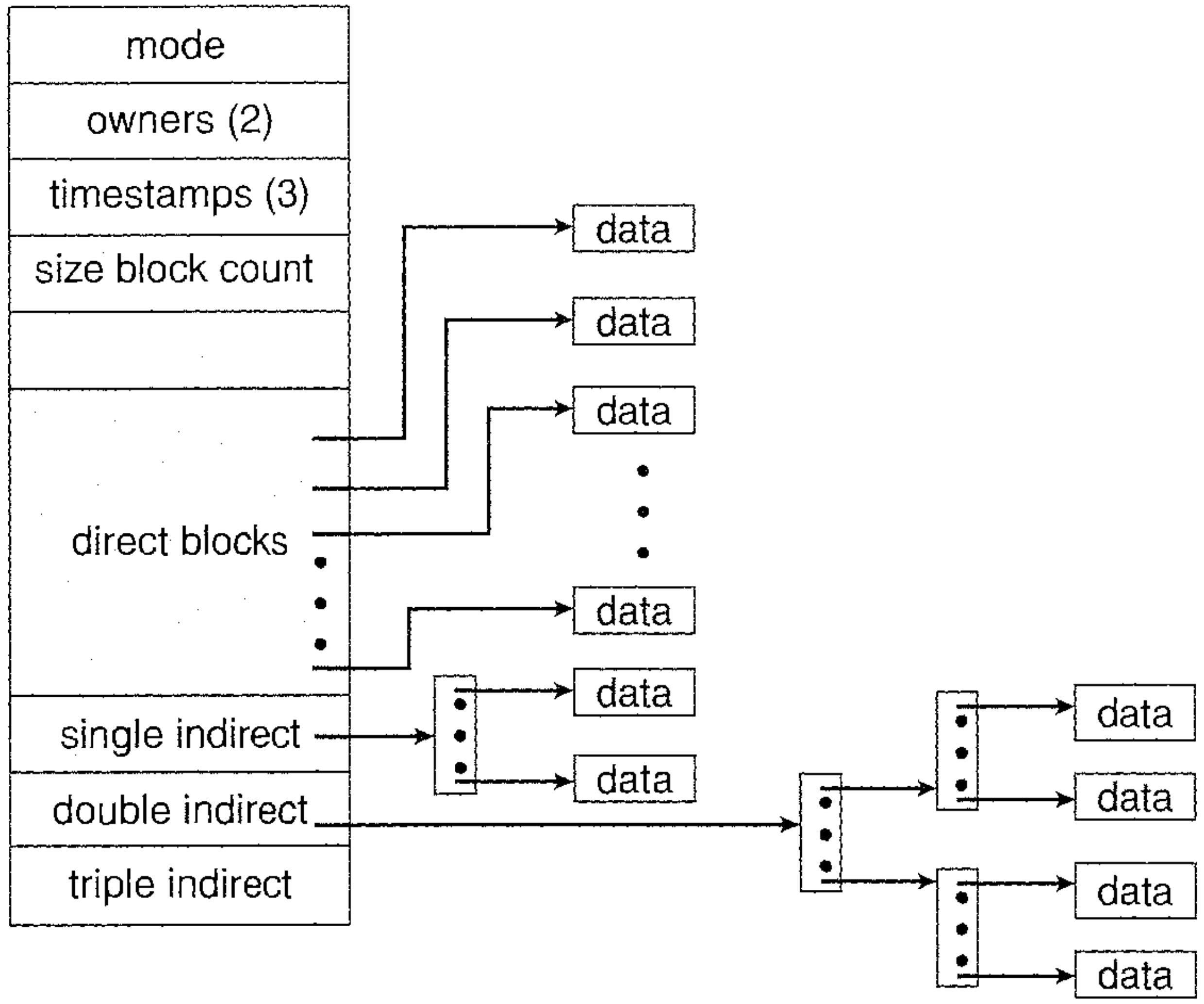
Thus, the data for small files (of not more than 12 blocks) do not need a separate index block.

The next three pointers point to *indirect blocks*.

The first points to a *single indirect block*, which is an index block containing not data but the addresses of blocks that do contain data.

The second pointer points to a *double indirect block*, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks.

The last pointer contains the address of a *triple indirect block*.


mode
owners (2)
timestamps (3)
size block count
direct blocks
single indirect
double indirect
triple indirect
data
data
data
data
data
data
data
data
data
data

inode

Free Space Management

The operating system maintains a *free-space list* to keep track of free blocks in the disk.

The *free-space list contains* the addresses or numbers of free blocks in the disk.

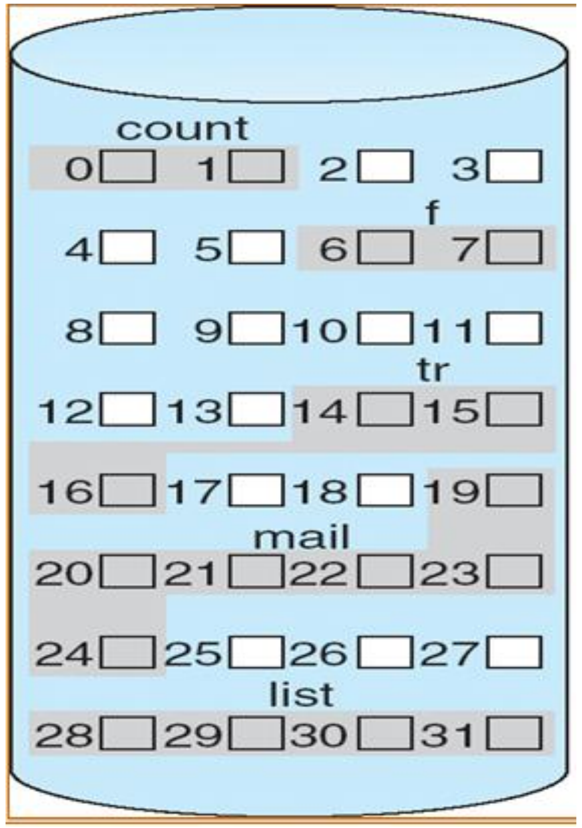
To allocate blocks to a file, the operating system searches the *free-space list* and identifies the required number of free blocks and allocates that blocks to the file.

The allocated blocks are then removed from the *free-space list.*

When a file is deleted, the blocks allocated to that file are added to the *free-space list.*

Methods for maintaining the Free Space List

1) Bit Vector
2) Linked List
3) Grouping
4) Counting



Bit Vector

The free-space list is implemented as a *Bit Map or Bit Vector*.

Each block is represented by one bit. If the block is free then the bit is 0; if the block is allocated then the bit is 1.

For example, consider a disk with 32 blocks (0 to 31) where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated.

The free-space *Bit Vector* is

11000011000000111001111110001111

Advantages

1) Simple

2) Easy to find first free block or n consecutive free blocks

Disadvantage

To store the *Bit Vector*, more space is required when the size of disk is large.

For example, if the size of disk is 1-GB (2^{30} bytes) and size of each block in the disk is 1-KB (2^{12} bytes) then the size of *Bit Vector* is

$$2^{30}/2^{12} = 2^{18} \text{ bits (or 32K bytes)}$$

32 KB space is required to store the *Bit Vector*.

A 1-TB disk with 4-KB blocks requires 32 MB to store the *Bit Vector*.

Linked List

All free blocks in the disk are linked together.

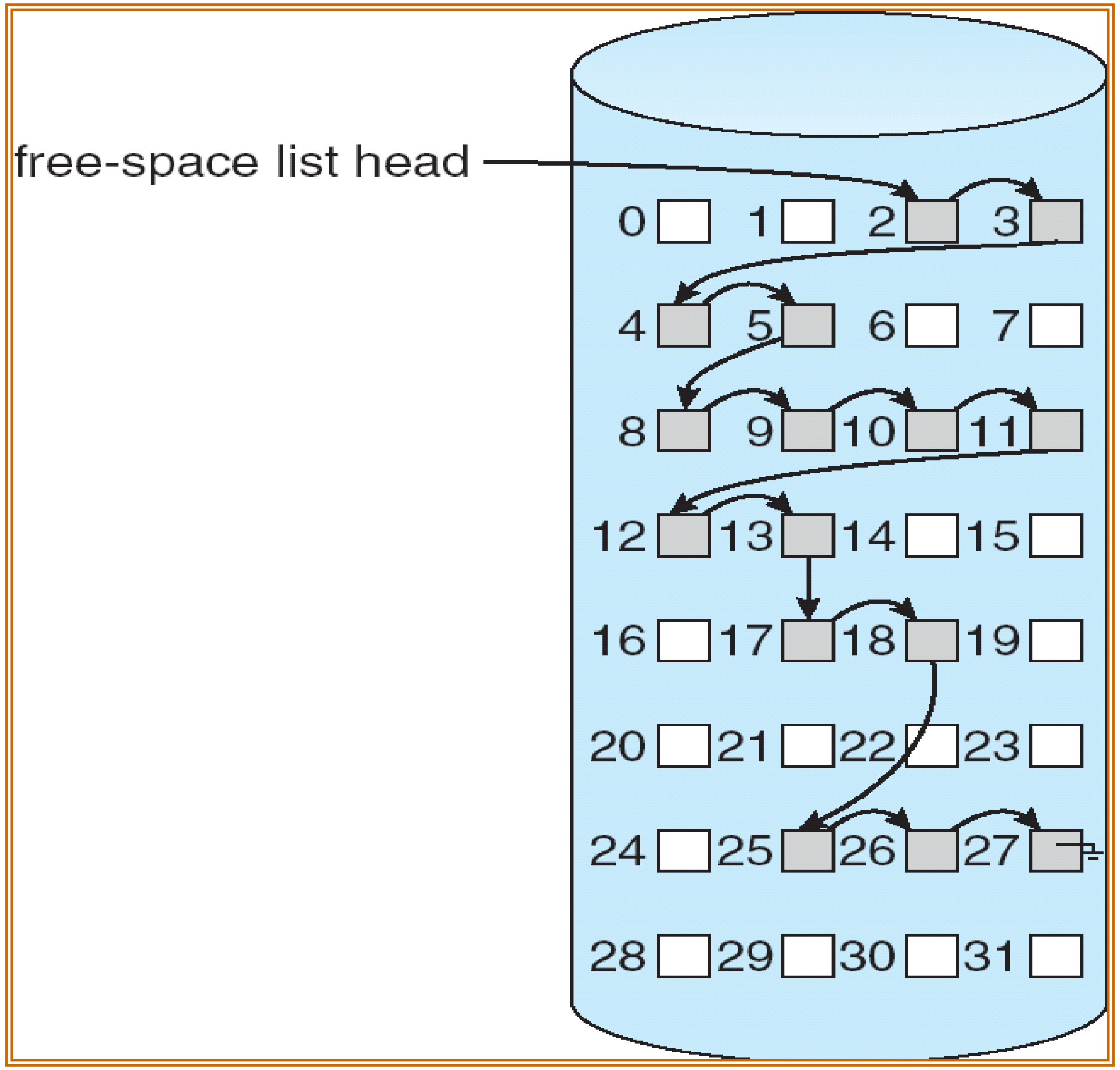
The address of first free block is stored in a special location in the disk.

The first free block contains a pointer to the next free block, and so on.

For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated.

In this situation, the address of block 2 is stored in the special location in the disk.

Block 2 will contain a pointer to block 3, which will point to block 4, which will point to block 5, which will point to block 8, and so on.


free-space list head
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31

<u>Advantage</u>

No wastage of space. i.e. no need to store the addresses of free blocks separately.

<u>Disadvantage</u>

Difficult to get contiguous free blocks

Grouping

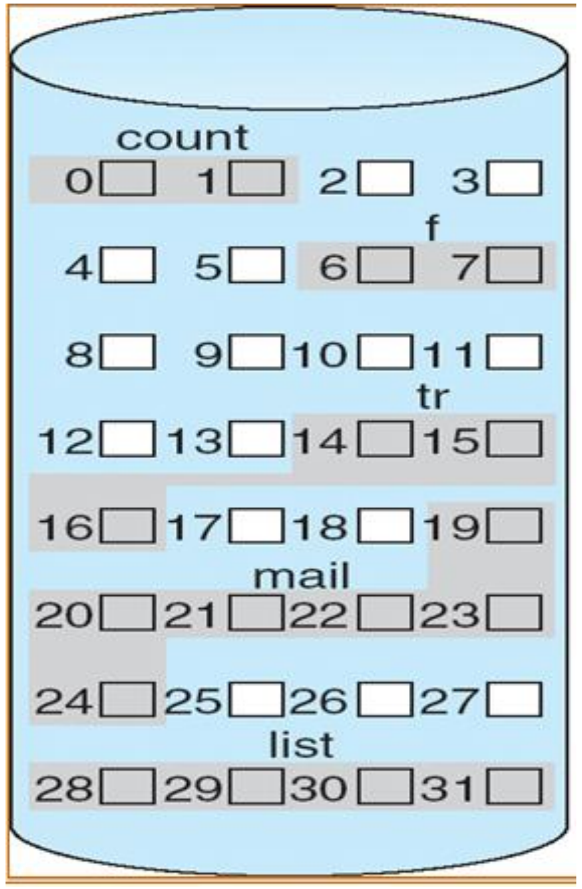
The addresses of *n* free blocks are stored in the first free block.

The first n-1 of these blocks are actually free.

The last block contains the addresses of another *n* free blocks, and so on.

Advantage

The addresses of a large number of free blocks can be found quickly.



Assume the size of each block is 6 pointers.

Block number 2 contains the addresses of blocks 3, 4, 5, 8, 9, 10.

Block number 10 contains the addresses of blocks 11, 12, 13, 17, 18, 25.

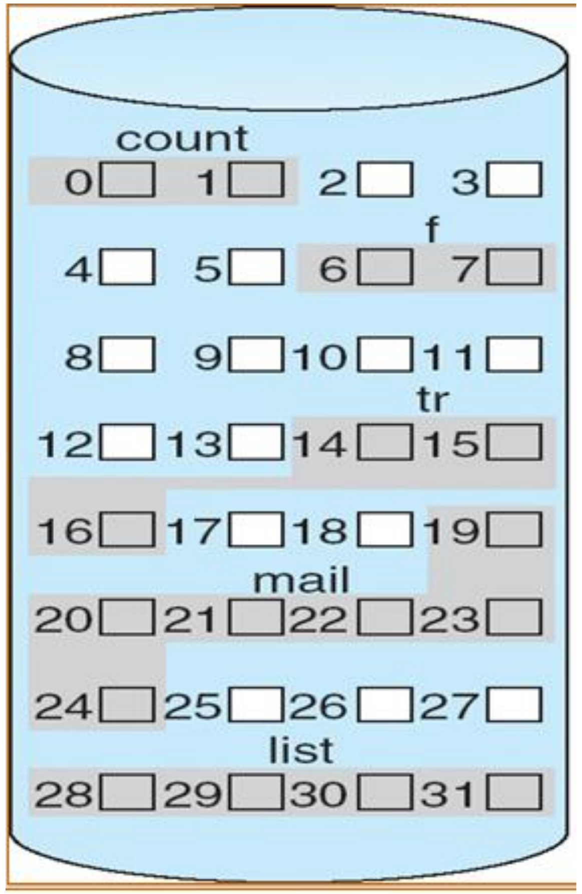
Block number 25 contains the addresses of blocks 26, 27.

Counting

When the space in disk is allocated with the contiguous allocation algorithm then several contiguous blocks may be allocated or freed simultaneously.

Instead of keeping a list of *n* free block addresses, we can keep the address of the first free block and the number (n) of contiguous free blocks that follow the first free block.

Each entry in the free-space list then consists of a disk address and a count.



Free Space List

(2, 4), (8, 6), (17, 2), (25, 3)