Module 5.1

Virtual Memory

Virtual Memory

In module 4, we discussed various memory-management strategies (contiguous allocation, paging, segmentation) used in computer systems for loading programs into main memory.

All these strategies require that an entire program be in memory before it can execute.

Virtual memory is a technique that allows the execution of programs that are not completely in memory.

One major advantage of this scheme is that programs can be larger than physical memory.

Virtual memory allows to load more number of programs into main memory, thereby increases performance of computer system.

Virtual memory also allows processes to share files and libraries, and to implement shared memory.

Demand Paging

Demand paging is commonly used in virtual memory systems.

With demand-paged virtual memory, pages are loaded only when they are ***demanded*** during program execution.

Pages that are never accessed are thus never loaded into physical memory.

The procedure used in demand paging technique to load programs into main memory is as follows:

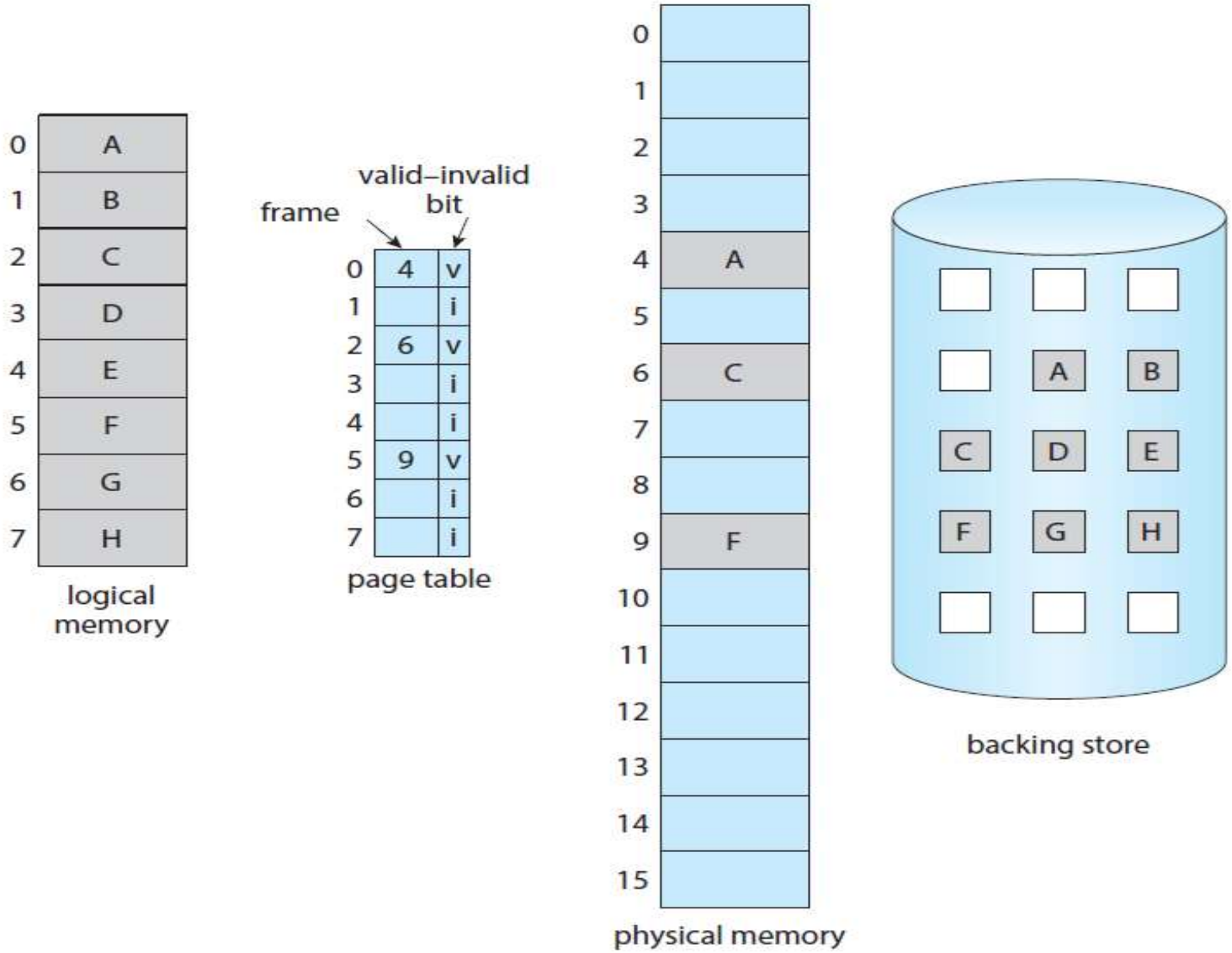
Before loading any program into main memory, the space in main memory is divided into a number of equal size frames.

To load a program into the main memory, the program is divided into a number of equal size pages.

Loads the required pages of the program into the free frames of main memory.

Then, creates a page table for the program.

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


0
A
1
B
2
C
3
D
4
E
5
F
6
G
7
H
logical memory
frame
valid–invalid bit
0 4 v
1 i
2 6 v
3 i
4 i
5 9 v
6 i
7 i
page table
0
1
2
3
4 A
5
6 C
7
8
9 F
10
11
12
13
14
15
physical memory
A B
C D E
F G H
backing store

A valid-invalid bit is stored in each entry of the page table to indicate whether the page has been loaded into main memory or not.

If the page is loaded into main memory, then the corresponding frame number is stored in the page table entry and the valid-invalid bit is set to ‘v’.

If the page is not loaded into main memory, then the frame number field is empty and the valid-invalid bit is set to ‘i’.

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

The logical address has two parts: page number and offset.

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**.

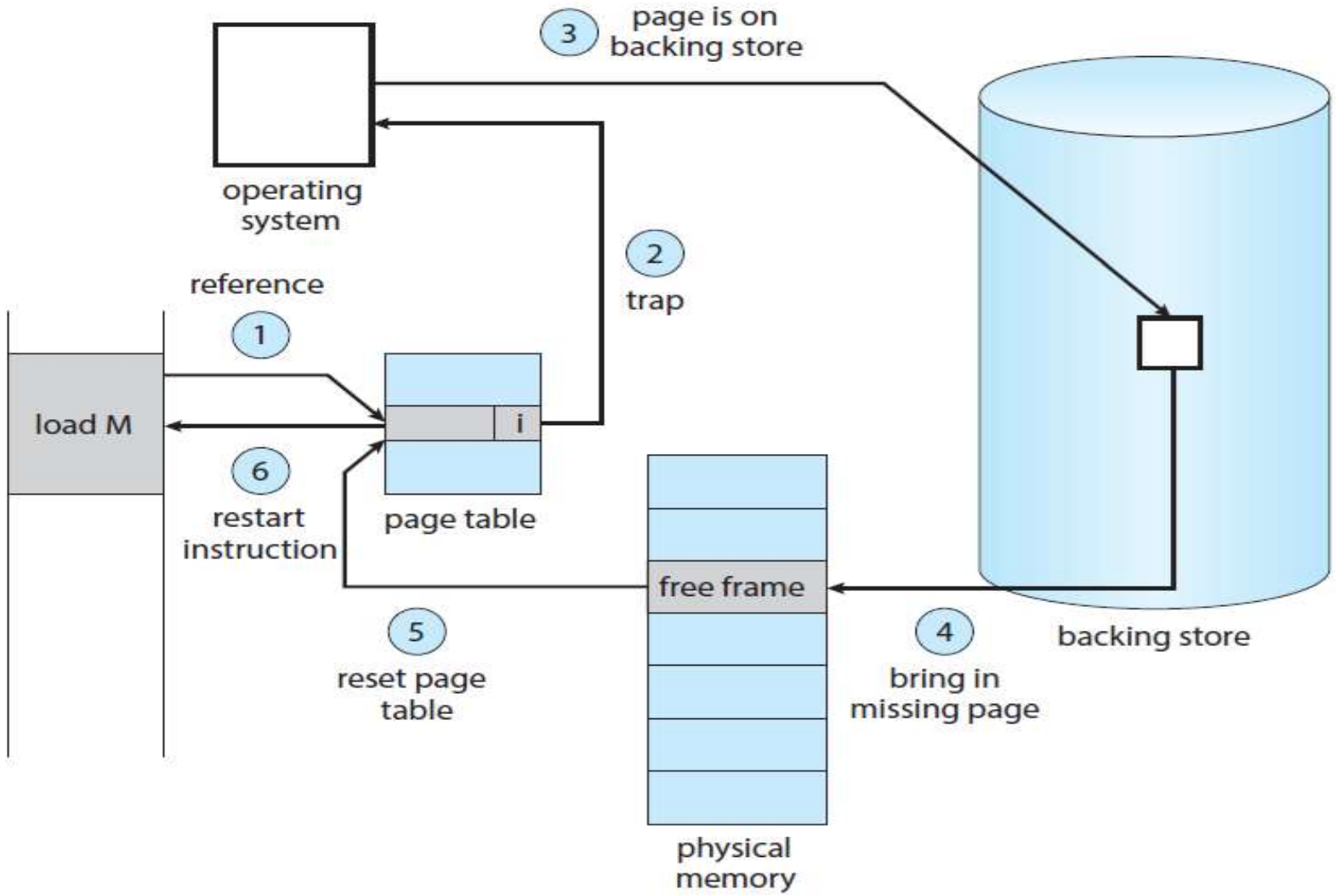
A page fault occurs when the required page is not present in the main memory.

When a page fault occurs, the operating system has to load the required page into the main memory.

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

1. Verifies whether any free frame is available in the main memory.
2. If any free frame is available, then loads the required page into the free frame and updates the page table of program.
3. Otherwise, replaces one of the pages in the main memory 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 main memory.
4. Instructs the CPU to generate the same logical address again.

The following diagram depicts the activities performed by operating system to handle the page fault.


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
backing store
physical
memory

Free-Frame List

When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.

To resolve page faults, most operating systems maintain a **free-frame list**, a pool of free frames for satisfying such requests.

Operating system allocate free frames using a technique known as **zero-fill-on-demand**.

Zero-fill-on-demand frames are “zeroed-out” before being allocated, thus erasing their previous contents.

Performance of Demand Paging

Demand paging can affect the performance of a computer system.

Effective access time is calculated for a demand-paged memory as follows:

Effective access time = $(1 - p) \times$ memory access time + $p \times$ page fault time.

p is the probability of a page fault ($0 \leq p \leq 1$).

Memory access time indicates the time required to access the statement from main memory.

Page fault time is the time required to service the page fault i.e. the time required to load the required page into main memory from secondary storage.

Ex: if the probability of occurring age fault is 0.2, memory access time is 100 nano seconds and the page fault time is 2000 nano seconds then the effective access time is

Effective access time=(1 - 0.2) X 100 + 0.2 X 2000 = 0.8 X 100 + 400 = 80 + 400 = 480 nano seconds

Page Replacement

During the execution of a program, if a page fault occurs then the operating system has to load the required page into the main memory.

If the main memory is not having a free frame then the operating system has to replace one of the pages in the main memory with the required page.

To select a page in main memory for replacement, the operating system uses different page replacement algorithms.

Page replacement algorithms

The following are different page replacement algorithms used by operating systems:

1. First in first out (FIFO) page replacement algorithm
2. Optimal page replacement algorithm
3. Least Recently Used (LRU) page replacement algorithm
4. Least Frequently Used (LFU) page replacement algorithm
5. Most Frequently Used (MFU) page replacement algorithm

Reference string

In a program the pages will be executed in any order and each page may be executed for any number of times.

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

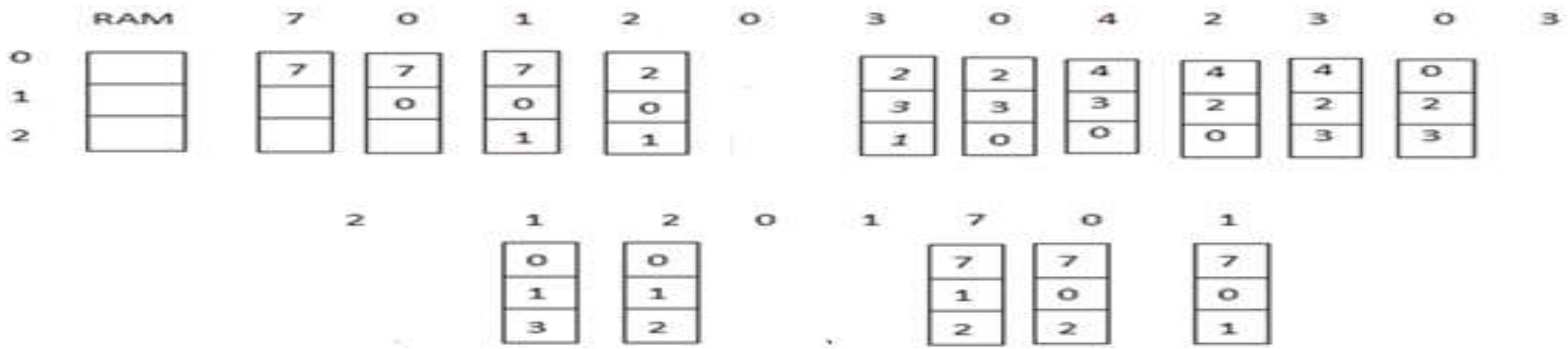
First in first out page replacement algorithm

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

Ex1:-

Number of pages in the program = 8 (0 to 7)
Number of frames in the main memory = 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 main memory is shown in below figure:

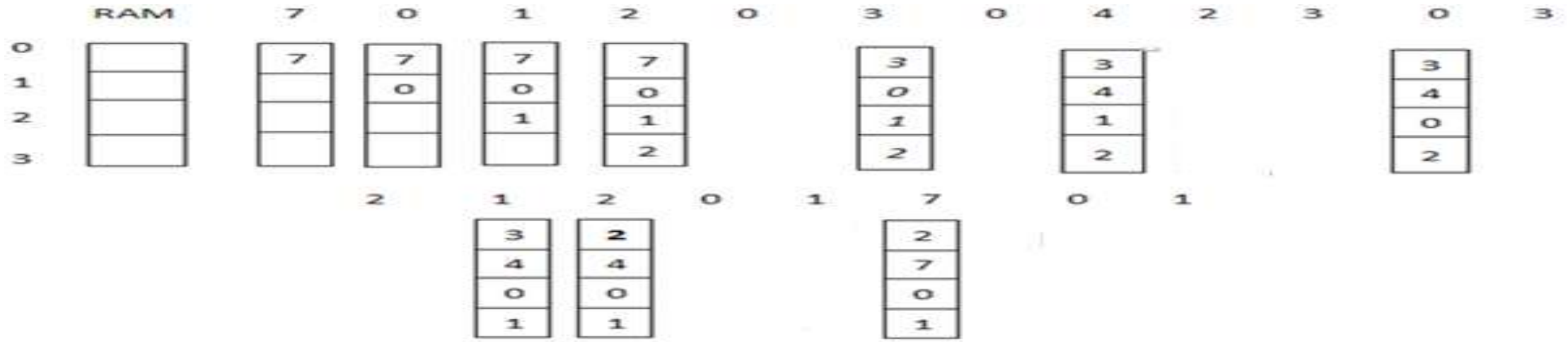


Number of page faults=15

Ex2:-

Number of pages in the program = 8
Number of frames in the main memory = 4
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 main memory is shown in below figure:



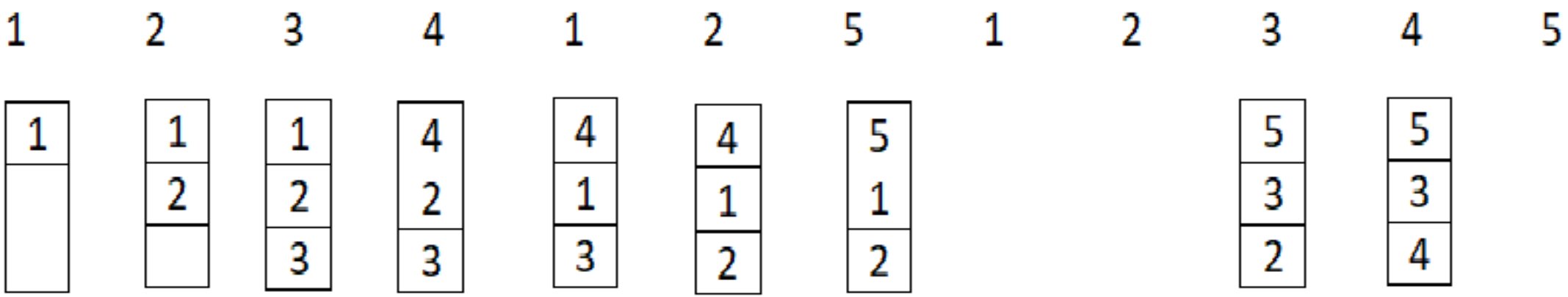
Number of page faults=10

Number of page faults will be decreased by increasing the number of frames in main memory.

Ex3:

Number of pages in the program = 5 (1 to 5)
Number of frames in the main memory = 3
Reference String: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

The replacement of pages in the main memory is shown in below figure:

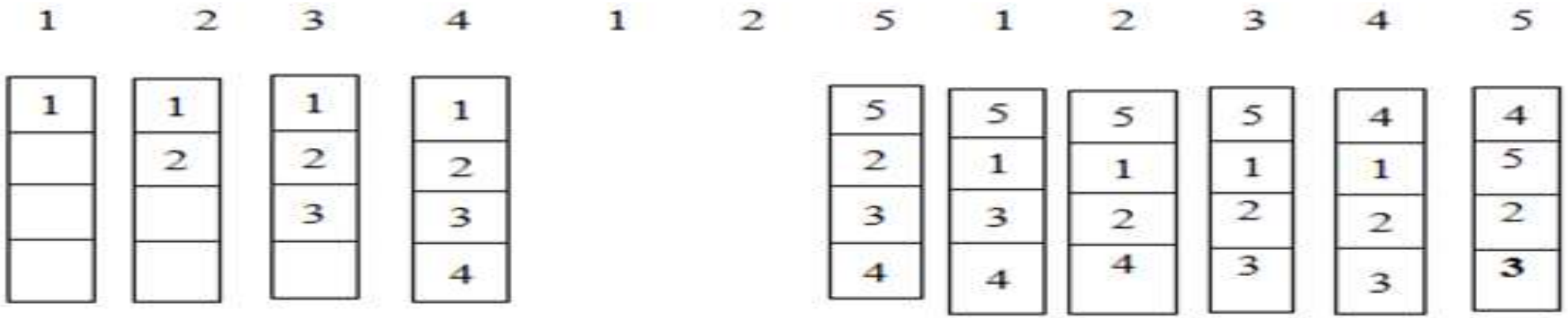


Number of page faults=9

Ex4:

Number of pages in the program = 5 (1 to 5)
Number of frames in the main memory = 4
Reference String: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

The replacement of pages in the main memory is shown in below figure:



Generally, the number of page faults will be decreased by increasing the number of frames in the main memory.

With FIFO algorithm, in some cases, the number of page faults increases when the number of frames in the main memory are increased.

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

Advantage:

Easy to implement when compared with other algorithms.

Disadvantages:

1) More number of page faults
2) Possibility of occurring Belady’s Anamoly.

To overcome from these drawbacks, Optimal page replacement algorithm is used.

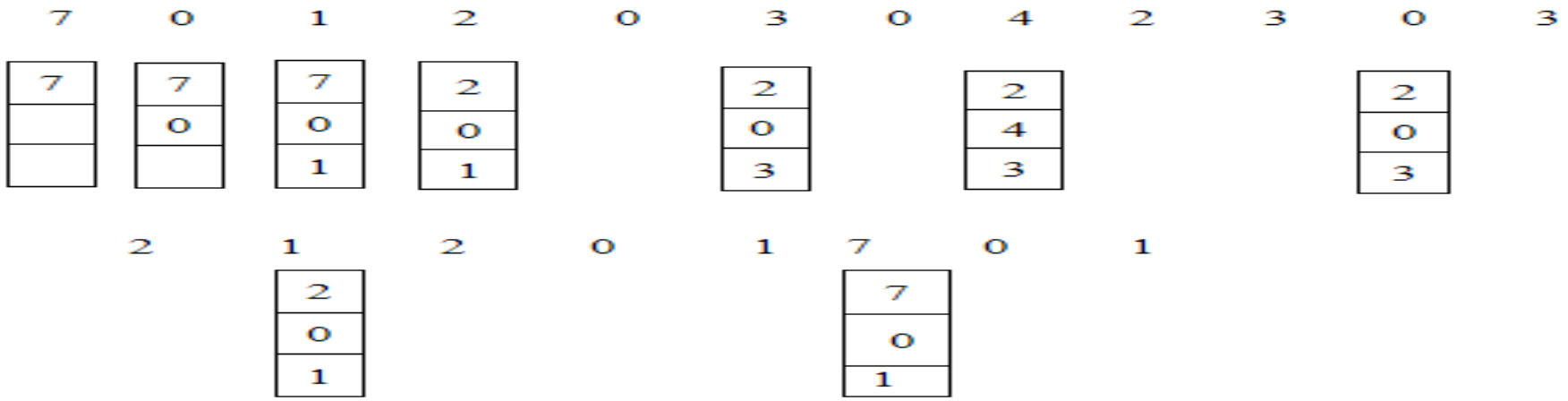
Optimal page replacement algorithm

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 program = 8 (0 to 7)
Number of frames in the main memory = 3
Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

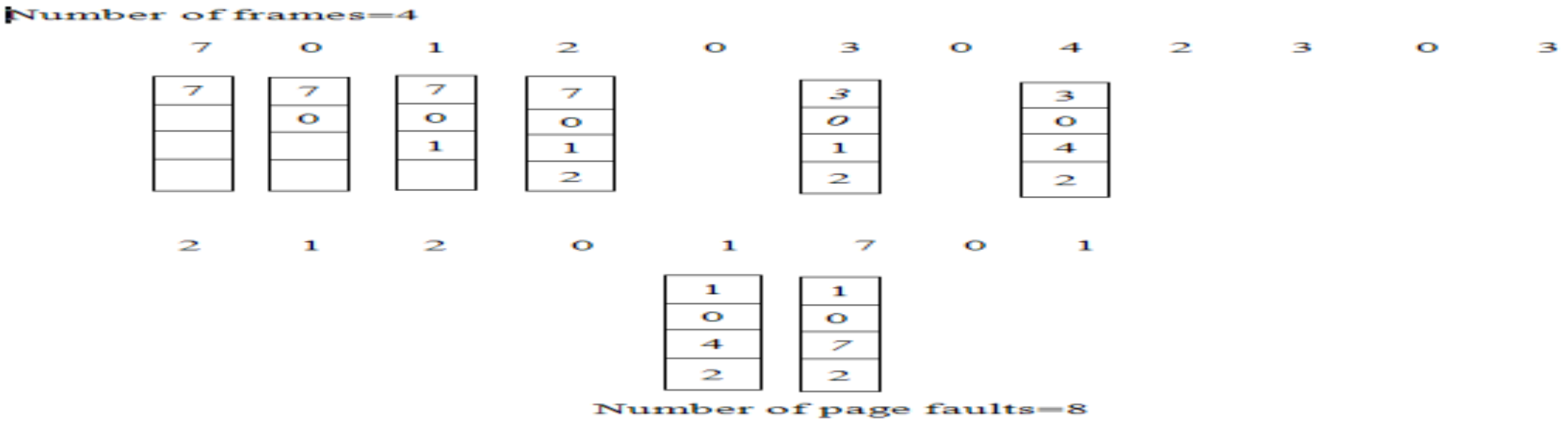
The replacement of pages in the main memory is shown in below figure:



Ex2:

Number of pages in the program = 8
Number of frames in the main memory = 4
Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

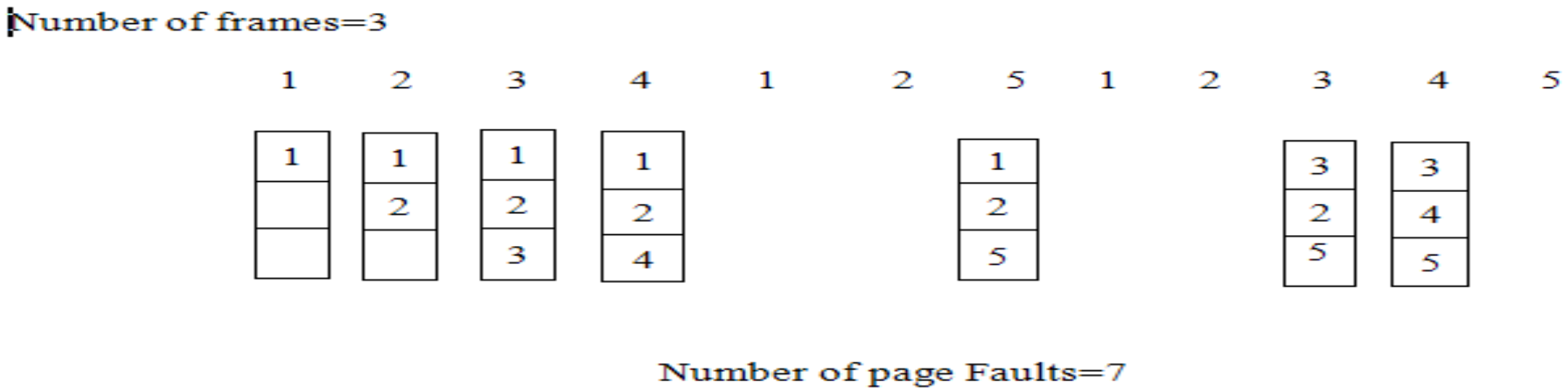
The replacement of pages in the main memory is shown in below figure:



Ex3:

Number of pages in the program = 5 (1 to 5)
Number of frames in the main memory = 3
Reference String: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

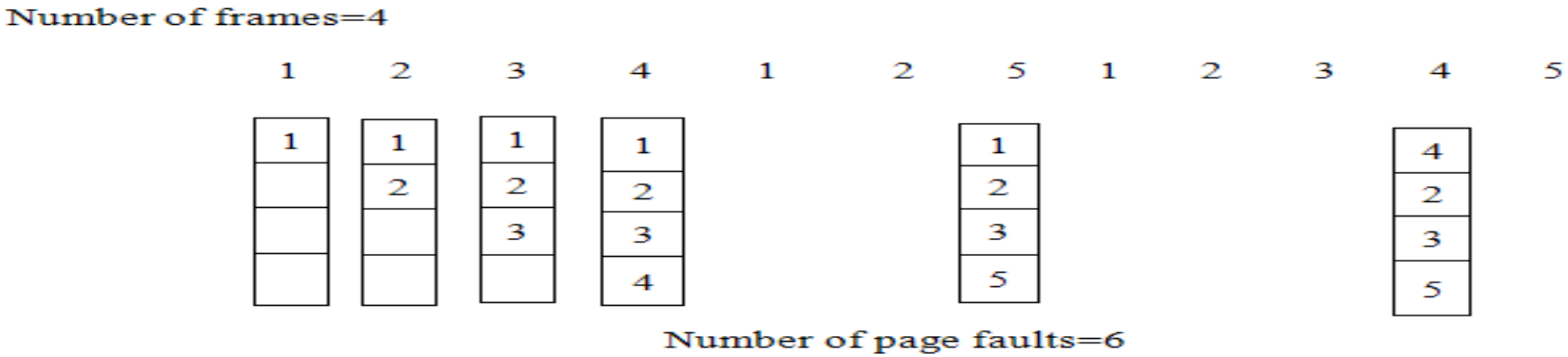
The replacement of pages in the main memory is shown in below figure:



Ex4:

Number of pages in the program = 5 (1 to 5)
Number of frames in the main memory = 4
Reference String: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

The replacement of pages in the main memory is shown in below figure:



Advantages:

1) Less number of page faults compared to FIFO algorithm.
2) No chance of occurring Belady's Anamoly.

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.

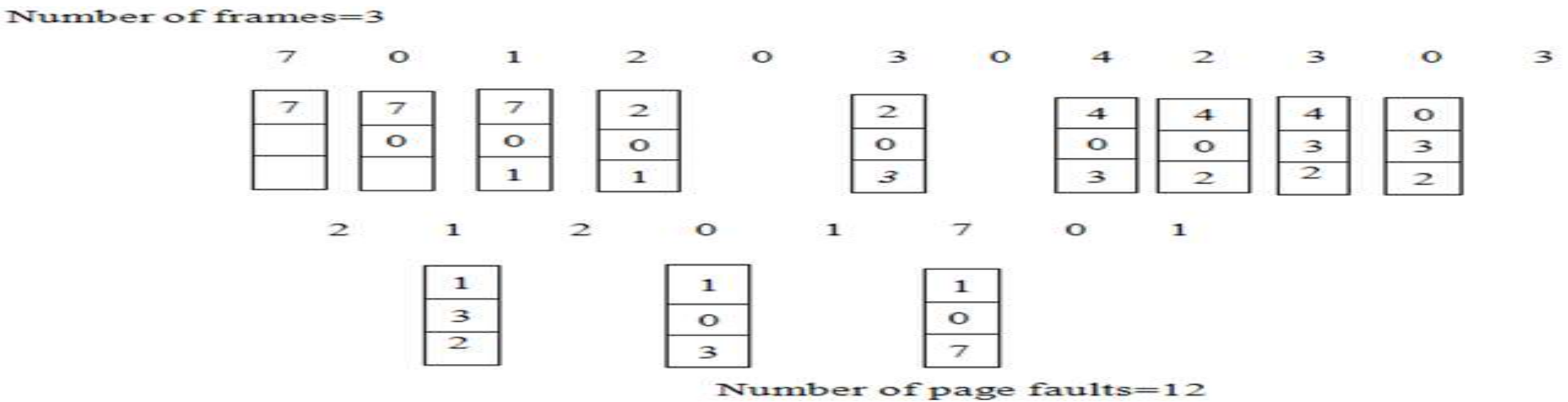
Least Recently Used Page Replacement Algorithm

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 program = 8
Number of frames in the main memory = 3
Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

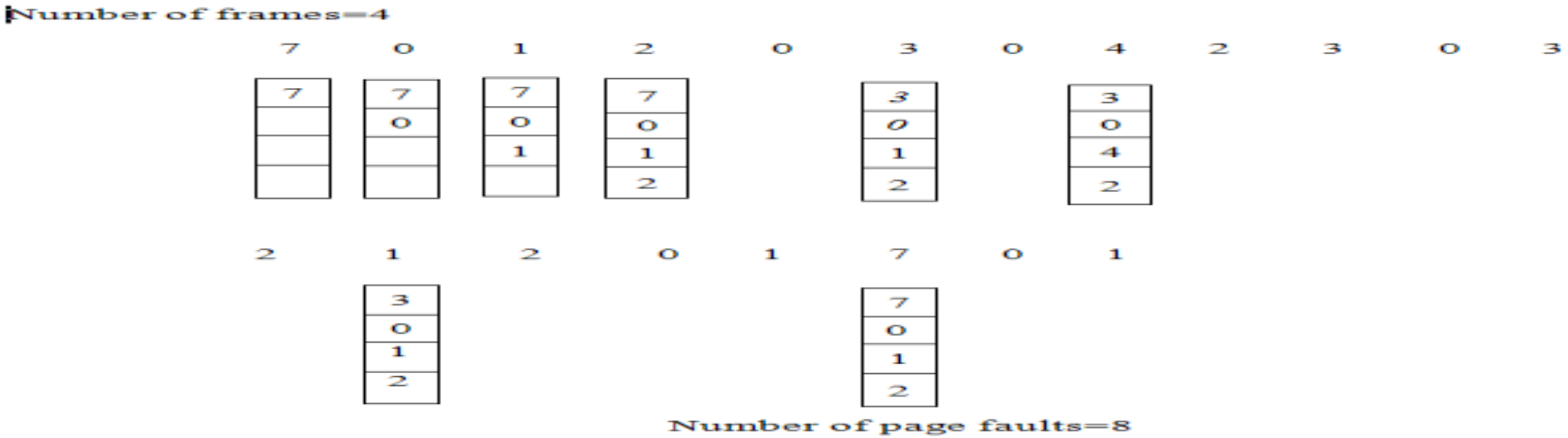
The replacement of pages in the main memory is shown in below figure:



Ex2:

Number of pages in the program = 8
Number of frames in the main memory = 4
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 main memory is shown in below figure:



Advantages:

1) Less number of page faults compared to FIFO algorithm.
2) No chance of occurring Belady's Anamoly.
3) Implementation is easy as it works based on the pages that have been referred or executed already.

Counting-Based Page Replacement

A reference count is maintained with each page loaded in the main memory.

Reference count of a page indicates the number of times that page is referenced or executed.

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

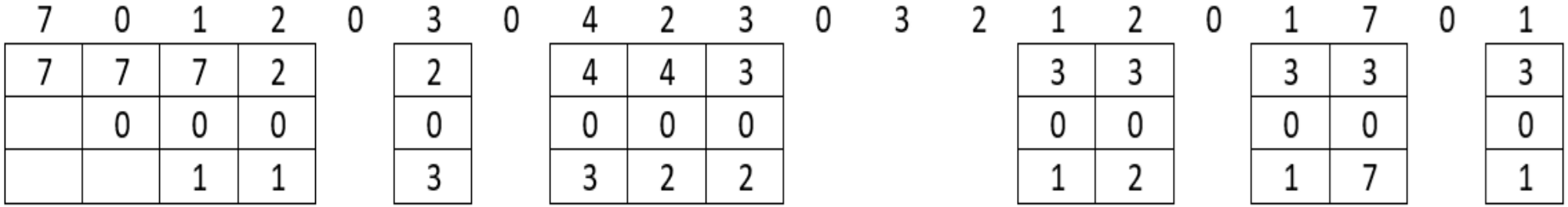
Replaces the page with least reference count.

If two or more pages have the same least reference count, then out of the pages with least reference count the page that was loaded first is replaced.

Ex1:

Number of pages in the program = 8
Number of frames in the main memory = 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 main memory is shown in below figure:



Number of page faults = 13

Most Frequently Used (MFU) page replacement algorithm

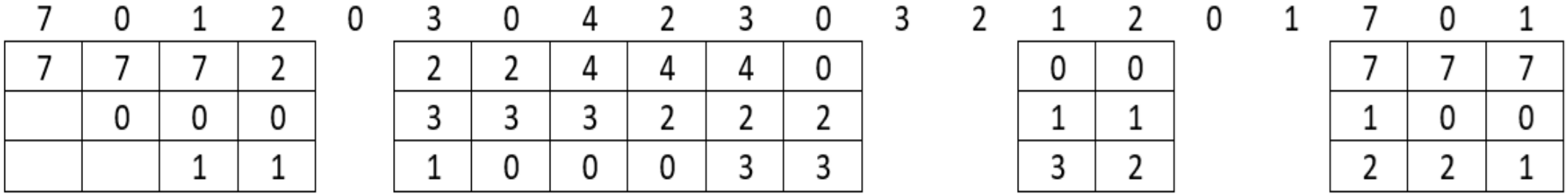
Replaces the page with highest reference count.

If two or more pages have the same highest reference count, then out of the pages with highest reference count the page that was loaded first is replaced.

Ex1:

Number of pages in the program = 8
Number of frames in the main memory = 3
Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

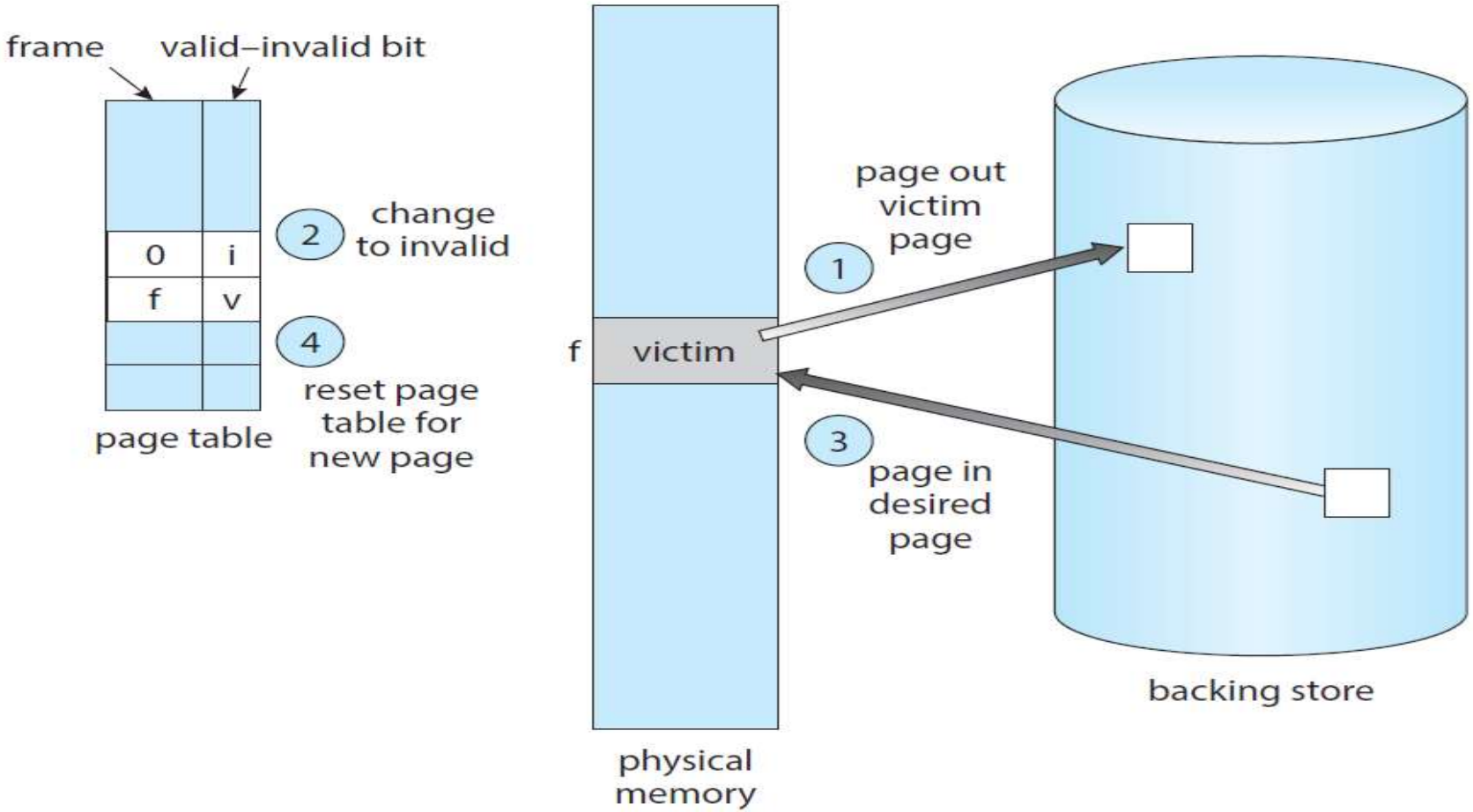
The replacement of pages in the main memory is shown in below figure:



Number of page faults = 15

Modify bit or dirty bit

When a page fault occurs, if no frames are free, ***two*** page transfers (one for the page-out and one for the page-in) are required as shown in the below figure.



This situation doubles the page-fault service time and increases the effective access time accordingly.

This overhead can be reduced by using a **modify bit** (or **dirty bit**).

When this scheme is used, each page has a modify bit associated with it.

The modify bit for a page is set whenever any byte in the page is written into, indicating that the page has been modified.

When a page is selected for replacement, its modify bit is examined.

If the bit is set, then the page must be written to the disk.

If the modify bit is not set, then the page is not required to be written to the disk. The page can be discarded.

This scheme reduces the time required to service a page fault.

Module 5.2

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
1) program files
2) data files

Program files contain programs written in different programming languages.

Data files may contain numeric or text data, photos, music, video, and so on..

In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user.

A file is named, for the convenience of its users, and is referred to by its name.

A name is usually a string of characters, such as example.c.

File Attributes

A file has an associated set of attributes.

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

The general attributes of 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.

Directory

The information about all files is kept in the directory structure, which resides on the same device as the files themselves.

The directory has an entry for each file.

A directory entry consists of the file's name and its unique identifier.

The identifier in turn locates the other file attributes.

File Operations

The seven 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) Opening a file: a file has to be opened to perform any operations on the file. Upon successful opening of the file, a file handle is returned using which operations are performed on the file.

3) Writing a file: to write to a file, file handle 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 after completing the write operation.

4) Reading a file: to read from a file, file handle should be specified. The operating system identifies the file and then reads the data from the position pointed by the file pointer and moves the file pointer after completing the read operation.

5) Repositioning within a file: repositions the file pointer. This operation is also called as file seek.

6) Deleting a file: the operating system search for the file, releases the file space and erase the directory entry of the file.

7) 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

File Types

File type helps the operating system to decide what operations can be performed on the file.

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.

Examples include resume.docx, server.c, and ReaderThread.cpp.

The following table depict some common file types, extensions and the content in that files.

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 compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

Allocation Methods

Many files are stored on the same disk.

The main problem is how to allocate space to these files so that storage space is utilized effectively and files can be accessed quickly.

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.

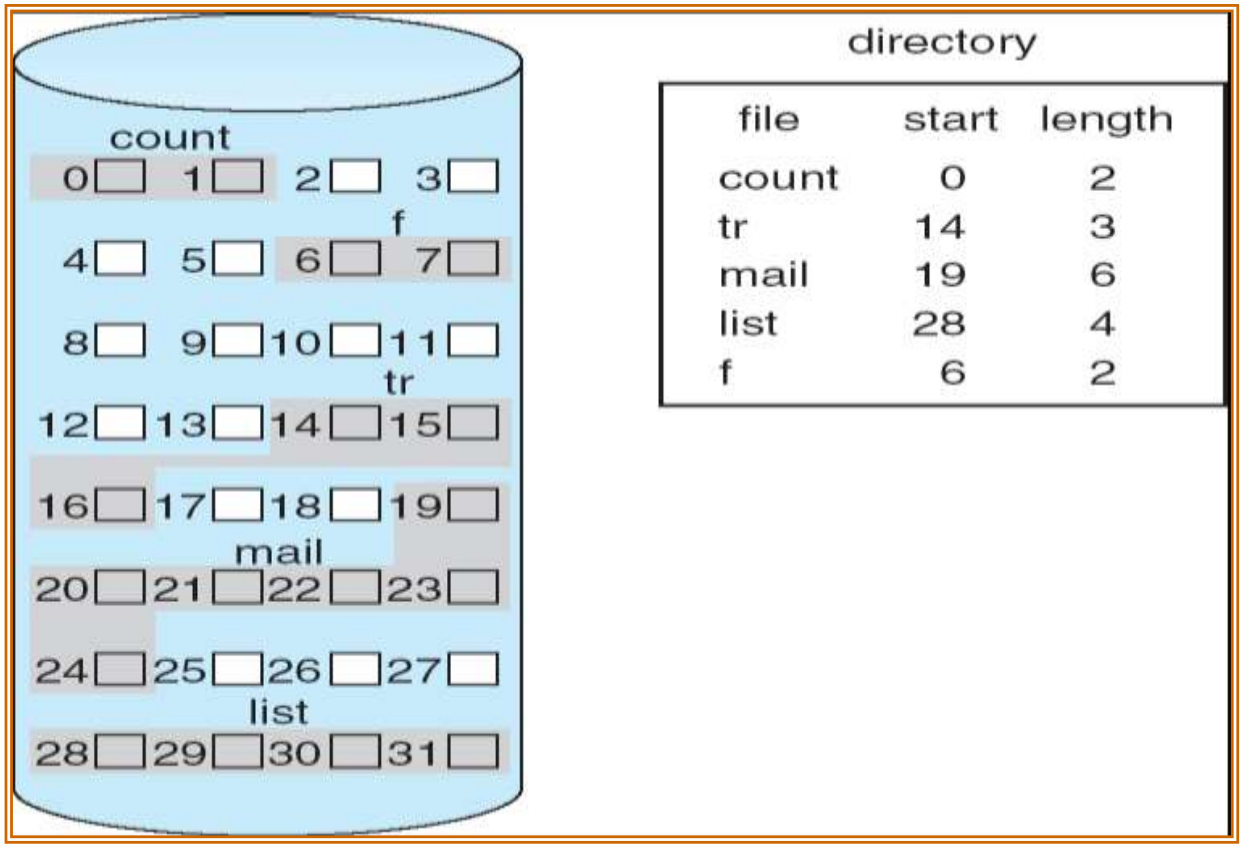
An operating system uses one method for all files within a file-system type.

Contiguous Allocation

A set of contiguous blocks is allocated to each file.

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.



Advantages:

1) Easy to implement

2) Supports both sequential and direct access to the blocks of file (For direct access to block *i* of a file that starts at block *b,* we can immediately access block *b* + *i).*

Disadvantages:

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.

When the file is created, the total amount of space it will need must be found and allocated.

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

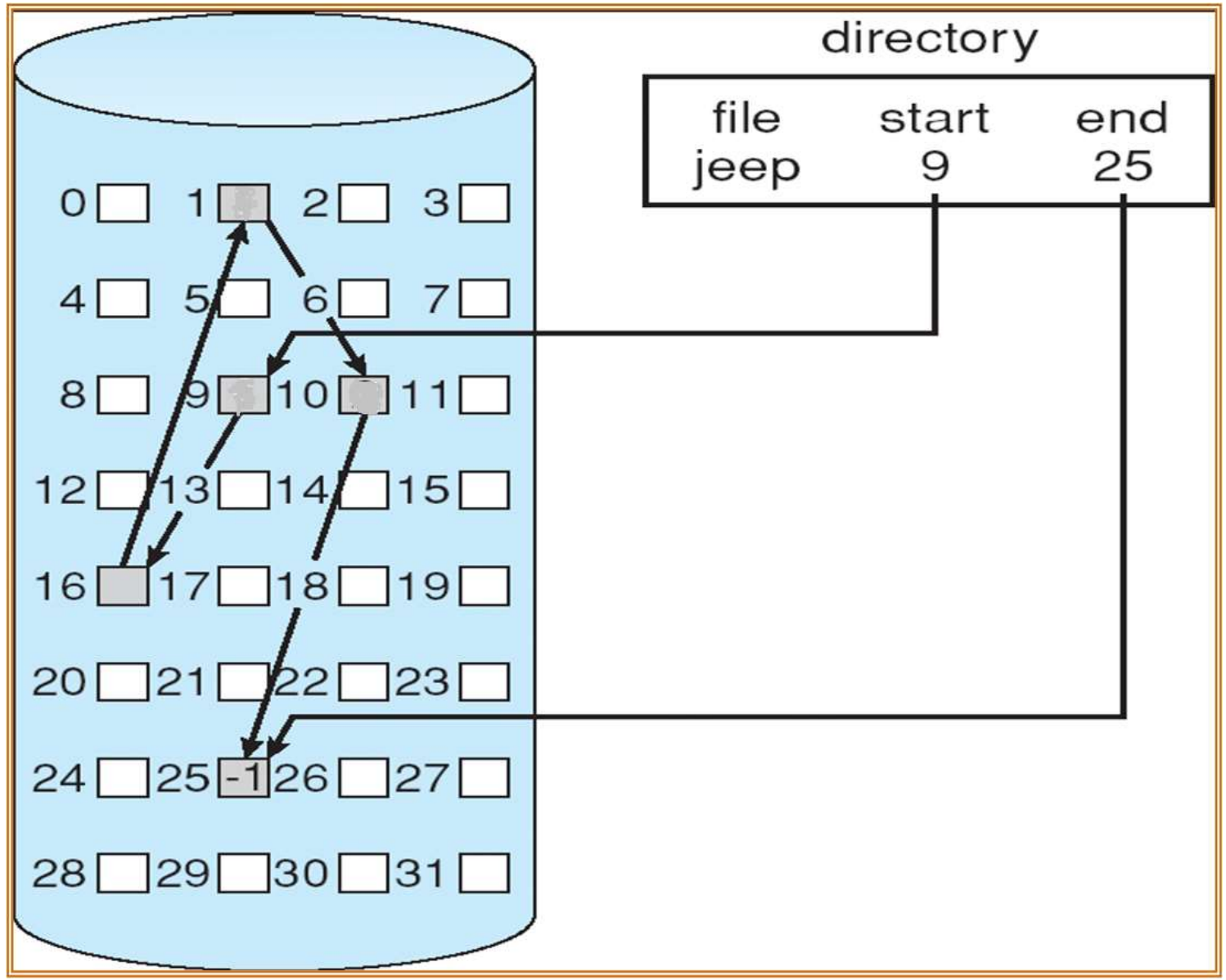
Linked allocation solves all problems of contiguous allocation.

With linked allocation, free 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
2
3
4
5
6
7
8
9
10
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 i^{th} block of a file, we must start at the beginning of that file and follow the pointers until we get to the i^{th} 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 block allocation and 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.

File Allocation Table (FAT)

FAT is a variant of Linked allocation technique.

This simple but efficient method of disk-space allocation was used by the MS-DOS operating system.

A section of storage at the beginning of the disk is reserved for the FAT.

The FAT has one entry for each block in the disk and is indexed by block number.

Initially, all entries in FAT are filled with 0 to indicate that all blocks in the disk are initially free.

To store a file in the disk, free blocks at any position of the disk can be allocated to the file as in linked allocation technique.

In the directory entry, name of the file and number of the first block allocated to the file is stored.

In the FAT, the entry indexed by the first block number that is allocated to the file contains the number of the next block allocated to the file.

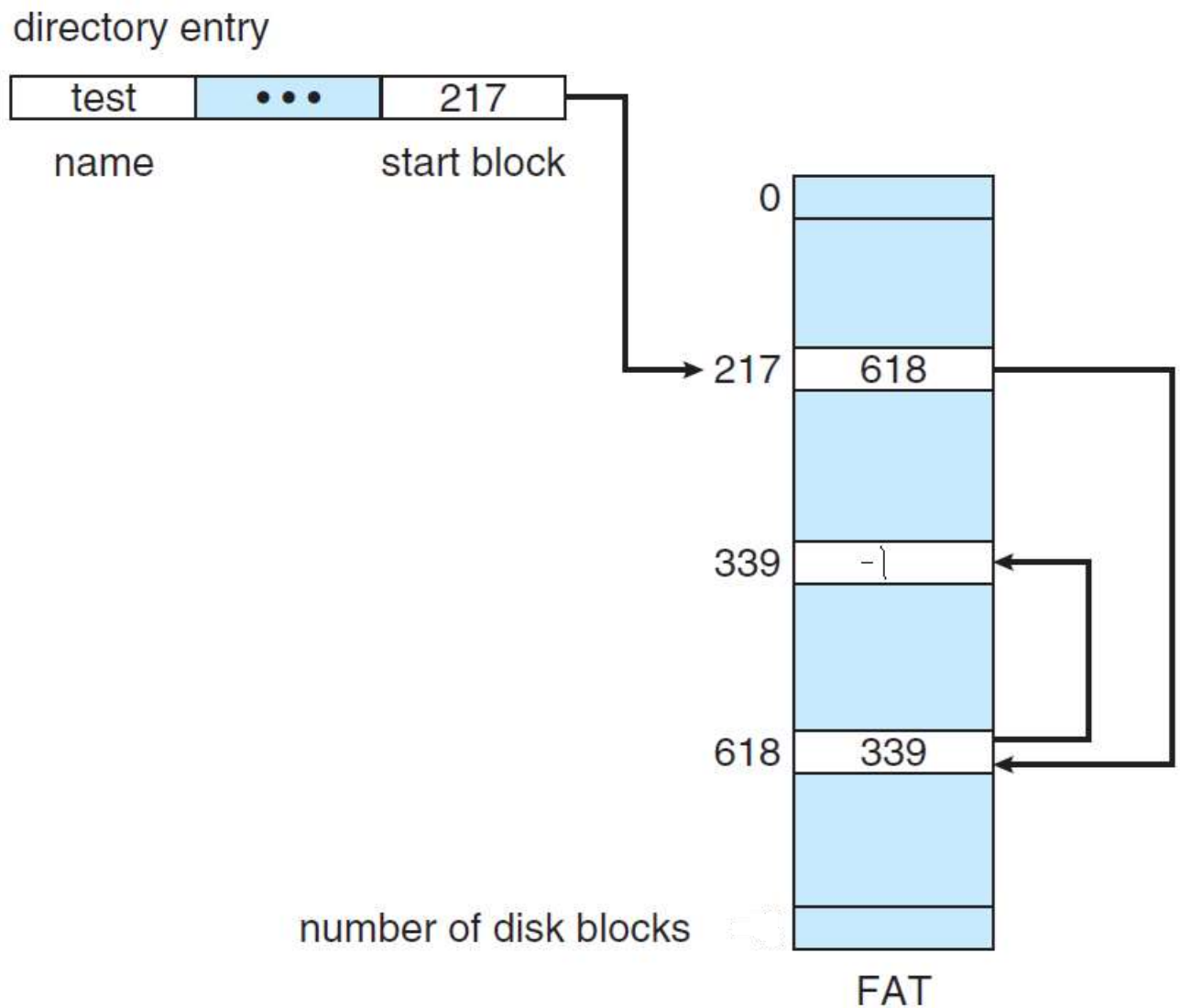
This chain continues until it reaches the last block, which has a special end-of-file value as the table entry.

Consider a file occupying the blocks 217, 618, and 339.

Following figure shows how the blocks allocated to the file are linked with FAT.

Allocating a new block to a file is a simple matter of finding the first 0-valued table entry and replacing the previous end-of-file value with the address of the new block.

The 0 is then replaced with the end-of-file value.


directory entry
test
• • •
217
name
start block
0
217
618
339
-1
618
339
number of disk blocks
FAT

Indexed Allocation

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

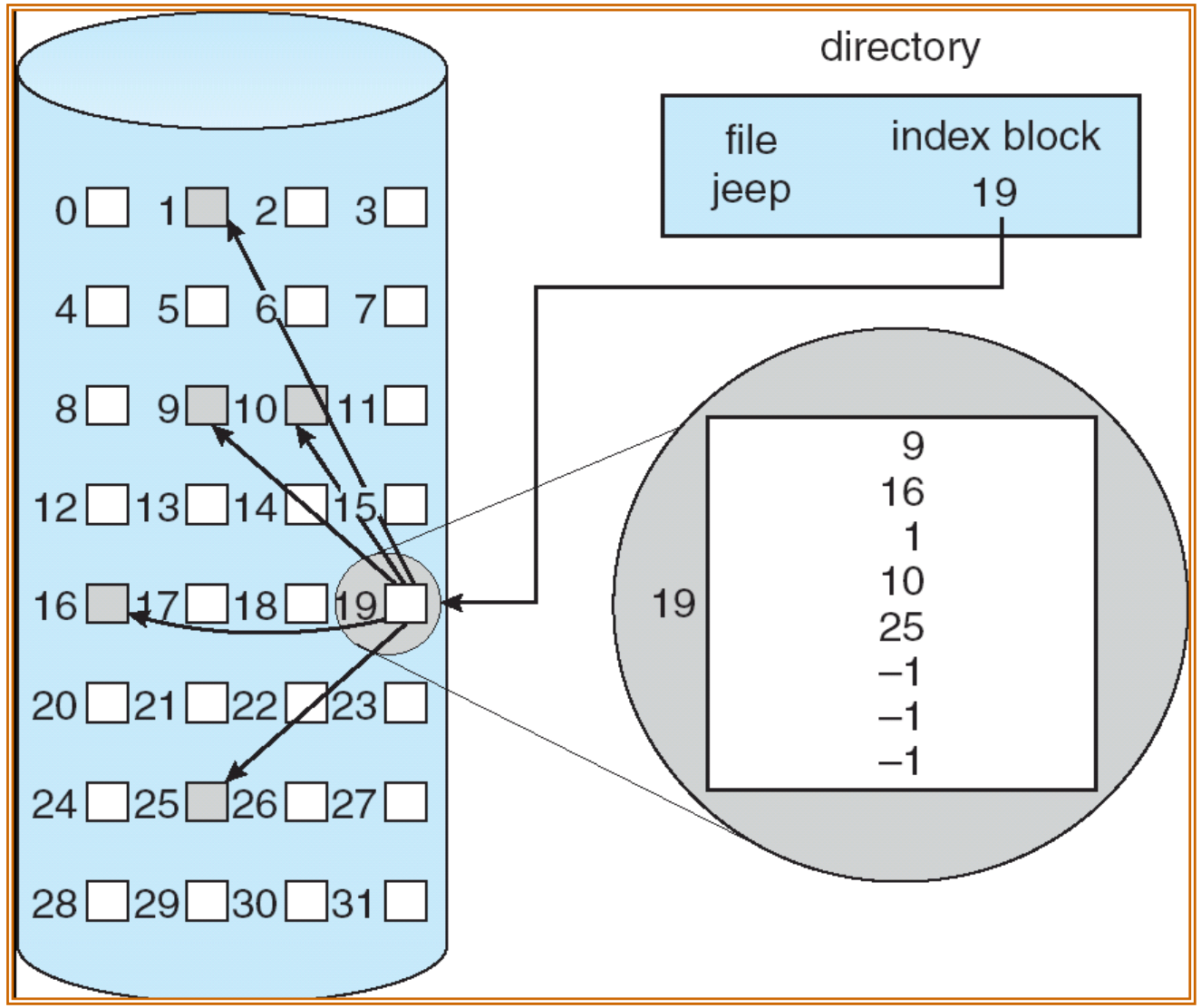
The addresses of blocks allocated to the file are stored into another block called *index block.*

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

The i^{th} entry in the index block points to the i^{th} block of the file.

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

To access the i^{th} *block*, we use the pointer in the i^{th} entry of *index block*.


directory
file
jeep
index block
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

Advantages:

1) There is no external fragmentation with indexed 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.

3) Blocks of the file can be accessed directly. i^{th} block of the file can be accessed directly using the pointer in the i^{th} entry of *index block*.

4) If one of the blocks allocated to the file is corrupted, still the remaining blocks of the file can be accessed.

Disadvantages:

1) Wastage of space in the index block of the file.

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-null.

2) Size of file is limited to number of pointers in the index block.

The following mechanisms are used to overcome this drawback:

1. Linked scheme
2. Multilevel index
3. Combined scheme

Linked scheme

If one index block is not enough to store the addresses of blocks allocated to a file then a 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.

This chain is repeated till the last index block of the file.

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 Unix File System (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.

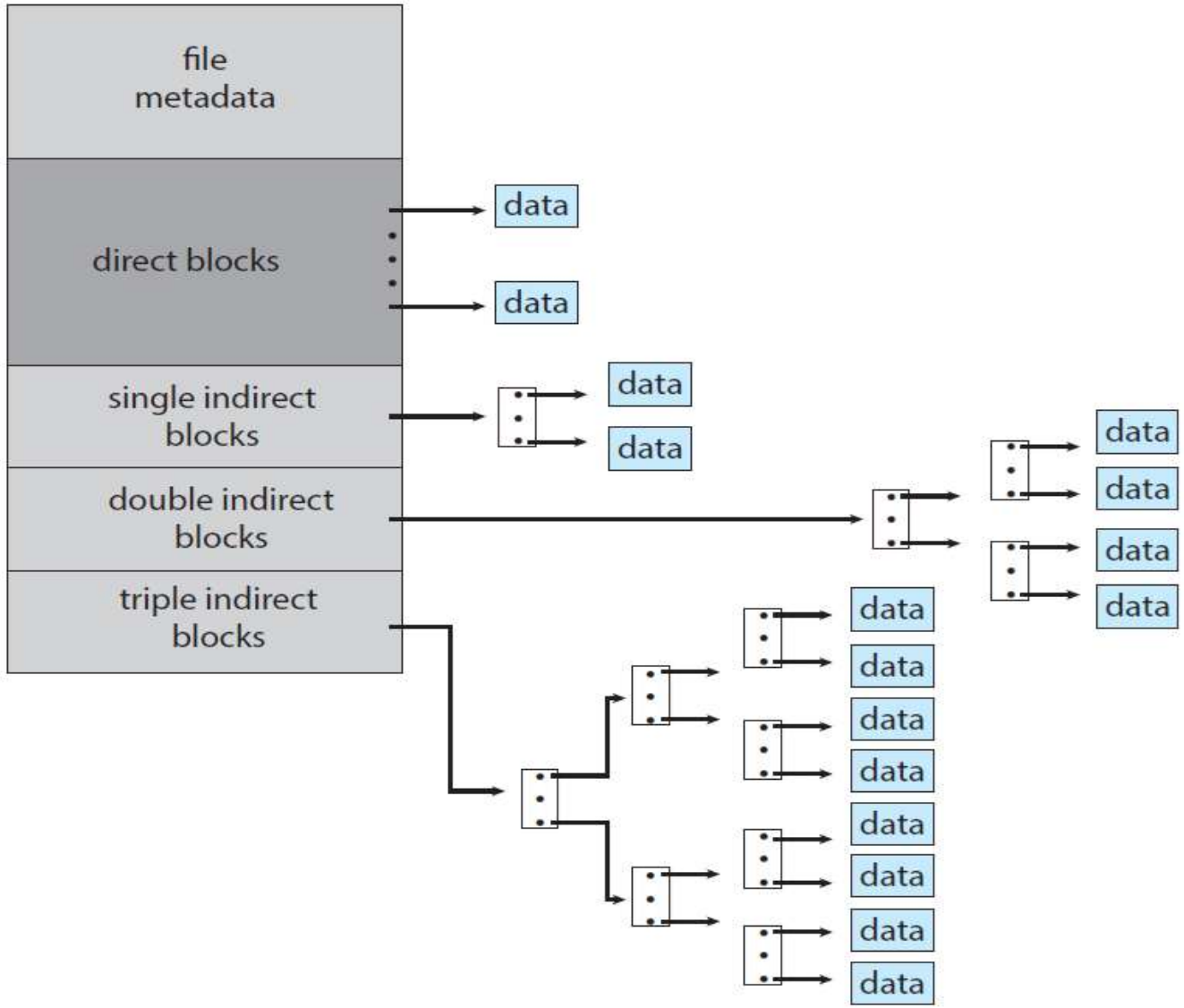
Thus, the data for small files (of not more than 12 blocks) do not need a separate index block.

The next three pointers point to *indirect blocks*.

The first pointer 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*.



The UNIX 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 implementing Free Space List

1) Bit Vector
2) Linked List
3) Grouping

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.

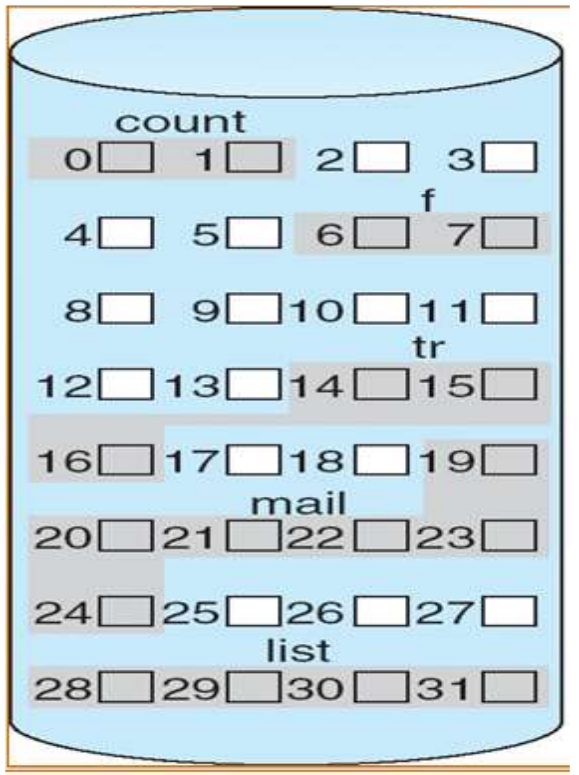
For example, consider a disk with 32 blocks (0 to 31) where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated.

The free-space *Bit Vector* is

11000011000000111001111110001111

Advantage:

1) 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-TB (2^{40} bytes) and size of each block in the disk is 4-KB (2^{12} bytes) then the size of *Bit Vector* is

$$2^{40}/2^{12} = 2^{28} \text{ bits} = 2^{25} \text{ bytes} = 2^5 \text{ MB (32 MB)}$$

32 MB space is required 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.

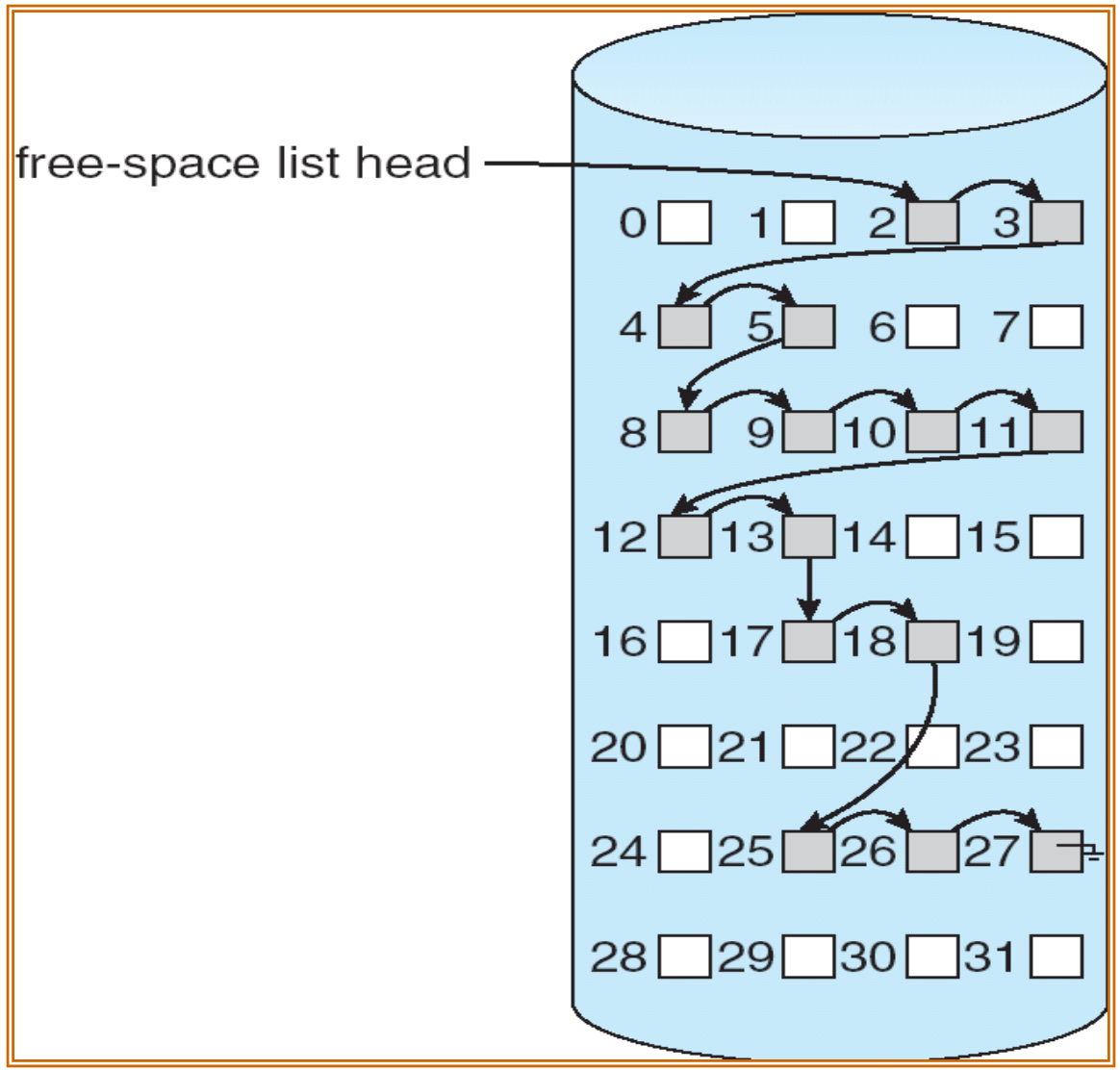
Block 2 will contain a pointer to block 3, which will point to block 4, which will point to block 5, which will point to block 8, and so on.

Advantage:

No need to store the addresses of free blocks separately.

Disadvantage:

Difficult to get contiguous free blocks.


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

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.

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.

Advantage:

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