Module 4: Memory and Storage management

Contiguous and Non-Contiguous Memory Allocation – Paging - Structure of the Page Table - Swapping – Demand Paging – Page Replacement – Segmentation – Allocation of Frames – Thrashing. Disk Structure - Disk Management - DISK Scheduling: FCFS, SSTF, SCAN, C-SCAN, LOOK. File System: Concept, Structure, Access Methods and allocation methods.

Background

- Program must be brought into memory (from disk) and placed within a process for it to be run
- CPU only has direct access to main memory and registers
    - Machine instructions take memory addresses as arguments
    - None take disk addresses as arguments

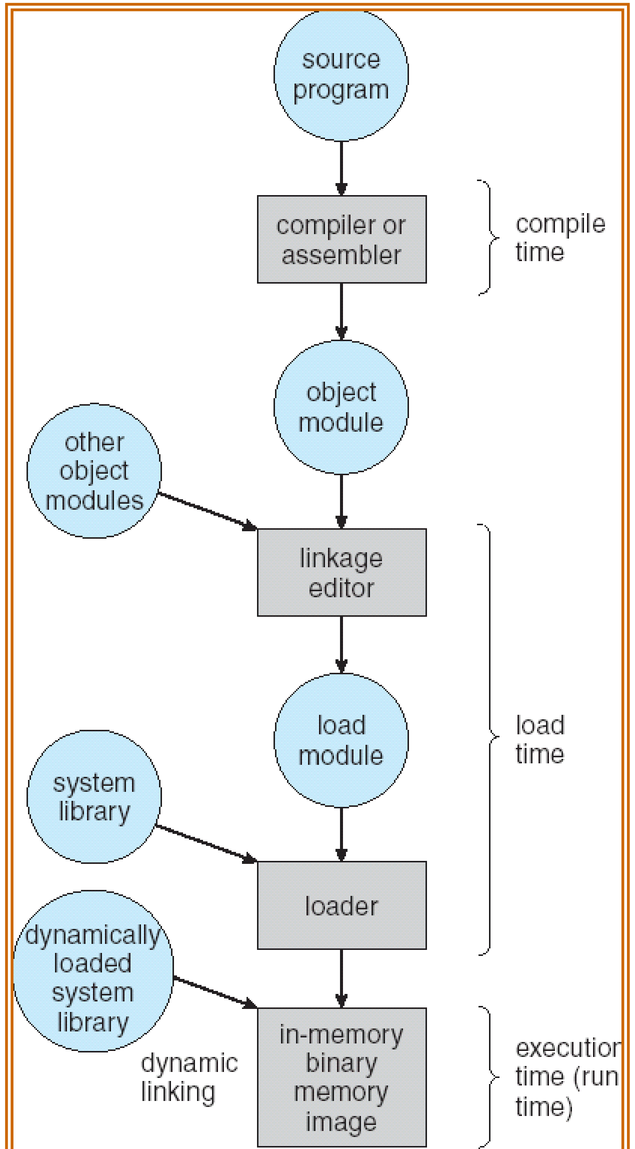
- Registers can be accessed in a single CPU clock cycle
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
    - Acts as buffer to accommodate access speed differentials

- **Protection** of memory required to ensure correct operation

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time; actual addresses are calculated at load time from the base load address.
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers)

Multistep Processing of a User Program



Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address**
    - Generated by the CPU
    - Also referred to as *virtual address*
  - **Physical address**
    - Address seen by the memory unit

- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme

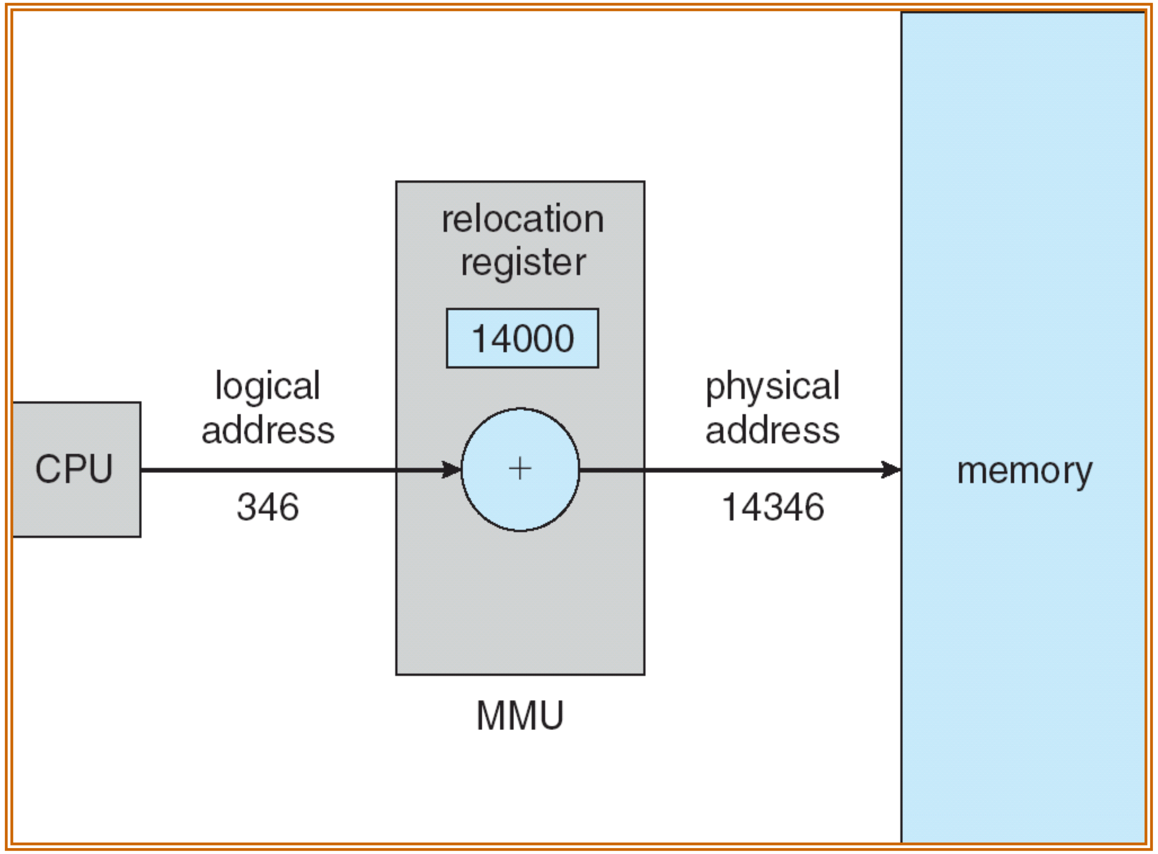
Memory-Management Unit (MMU)

- Hardware device that maps logical/virtual to physical address

- In MMU scheme, the value in the base register (relocation register) is added to every address generated by a user process at the time it is sent to memory

- The user program deals with *logical* addresses; it never sees the *real* physical addresses

Dynamic relocation using a relocation register



Dynamic Loading

- Routine is not loaded until it is called
- Better memory-space utilization
  - Unused routines are never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases (such as exception handling)
- No special support from the operating system is required; implemented through program design
- Previously, we computed “overlays” by hand

Dynamic Linking

- Linking postponed until execution time
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system needed to check if routine is in processes’ memory address
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**

Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution

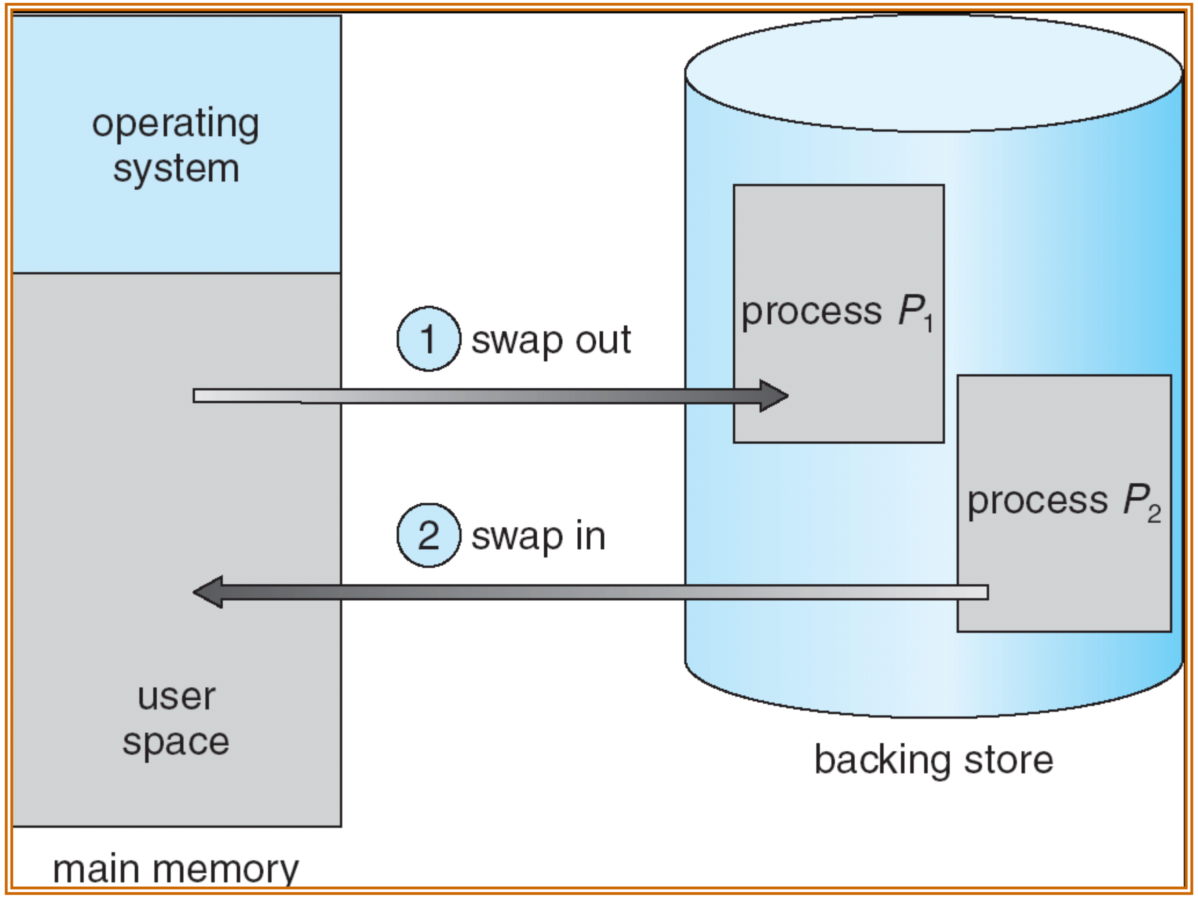
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images

- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed

- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped

- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Schematic View of Swapping

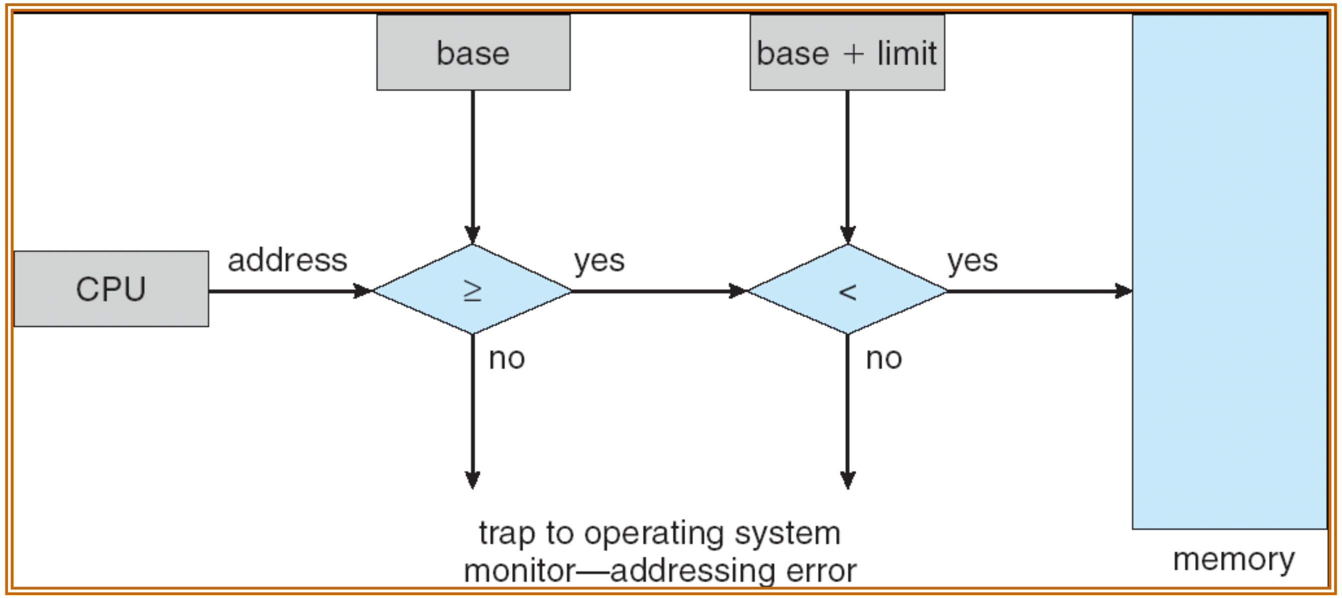


Contiguous Allocation

- Main memory usually into two partitions:
    - Resident operating system usually held in low memory
        - Typically because the interrupt vector is there too!
    - User processes usually held in high memory

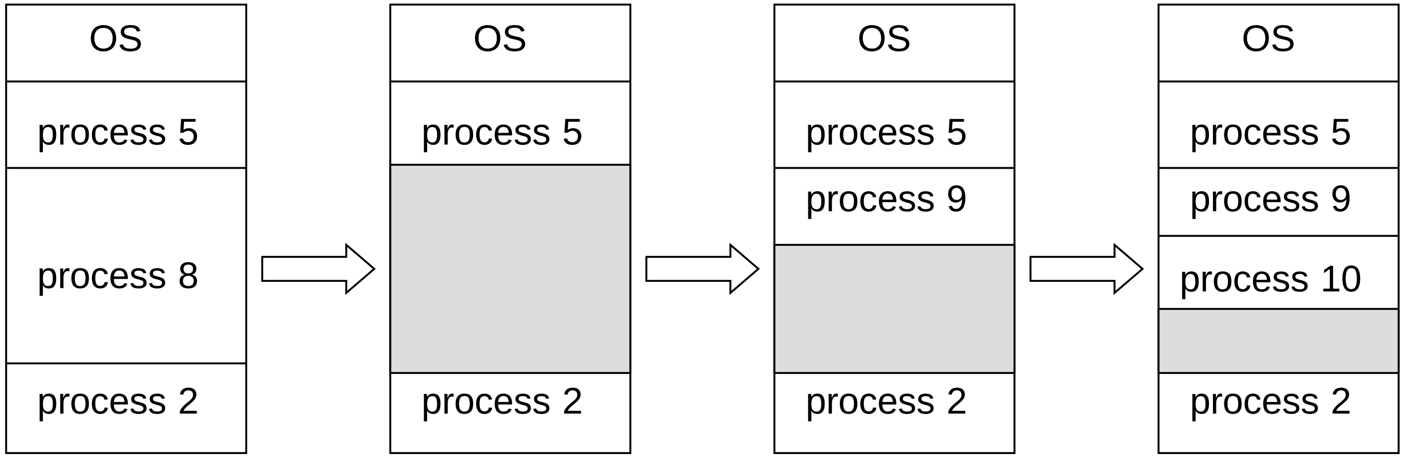
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
    - Base register contains value of smallest physical address
    - Limit register contains range of logical addresses – each logical address must be less than the limit register
    - MMU maps logical address *dynamically*

HW address protection with base and limit registers



Contiguous Allocation (Cont.)

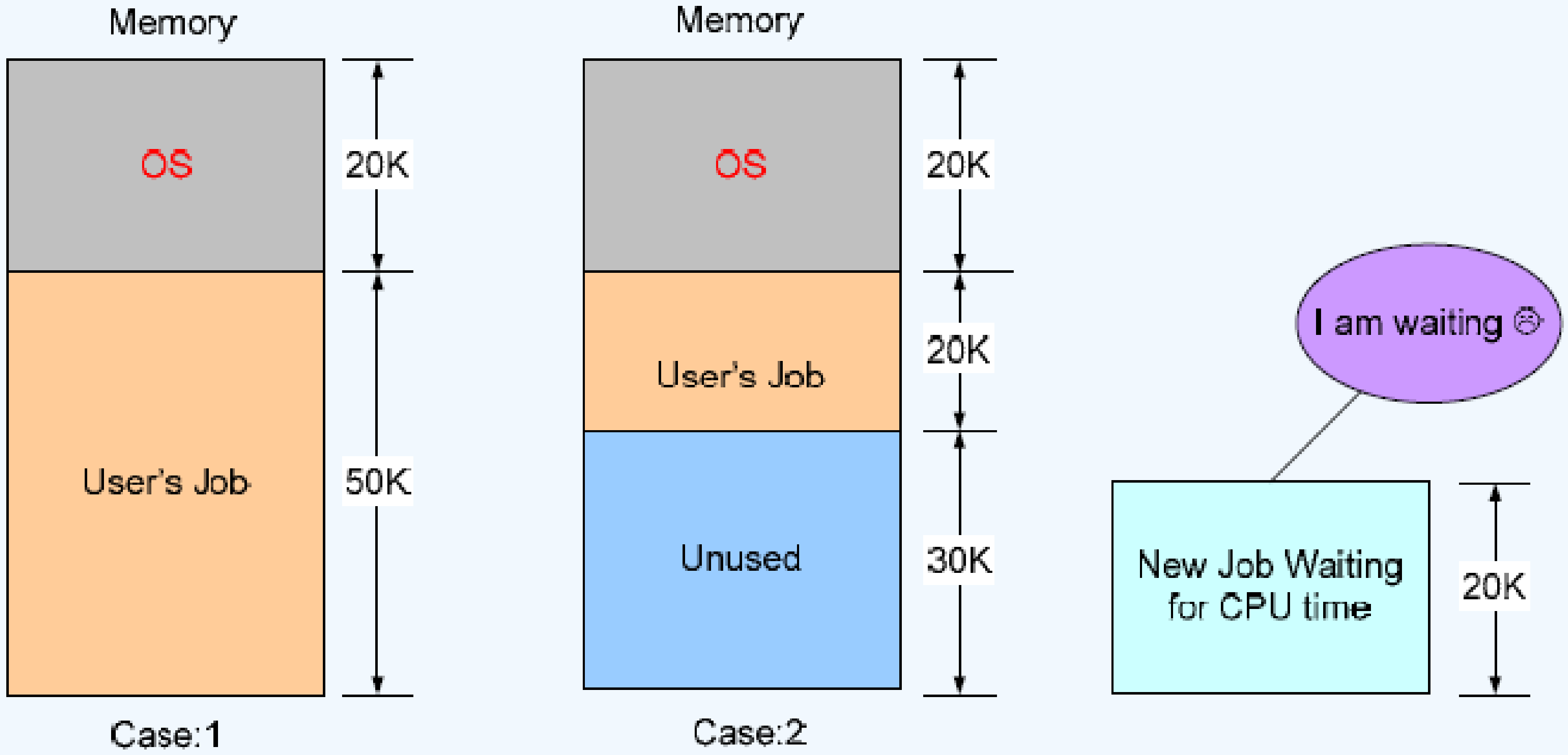
- Multiple-partition allocation
  - Hole – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Operating system maintains information about:
    a) allocated partitions   b) free partitions (holes)



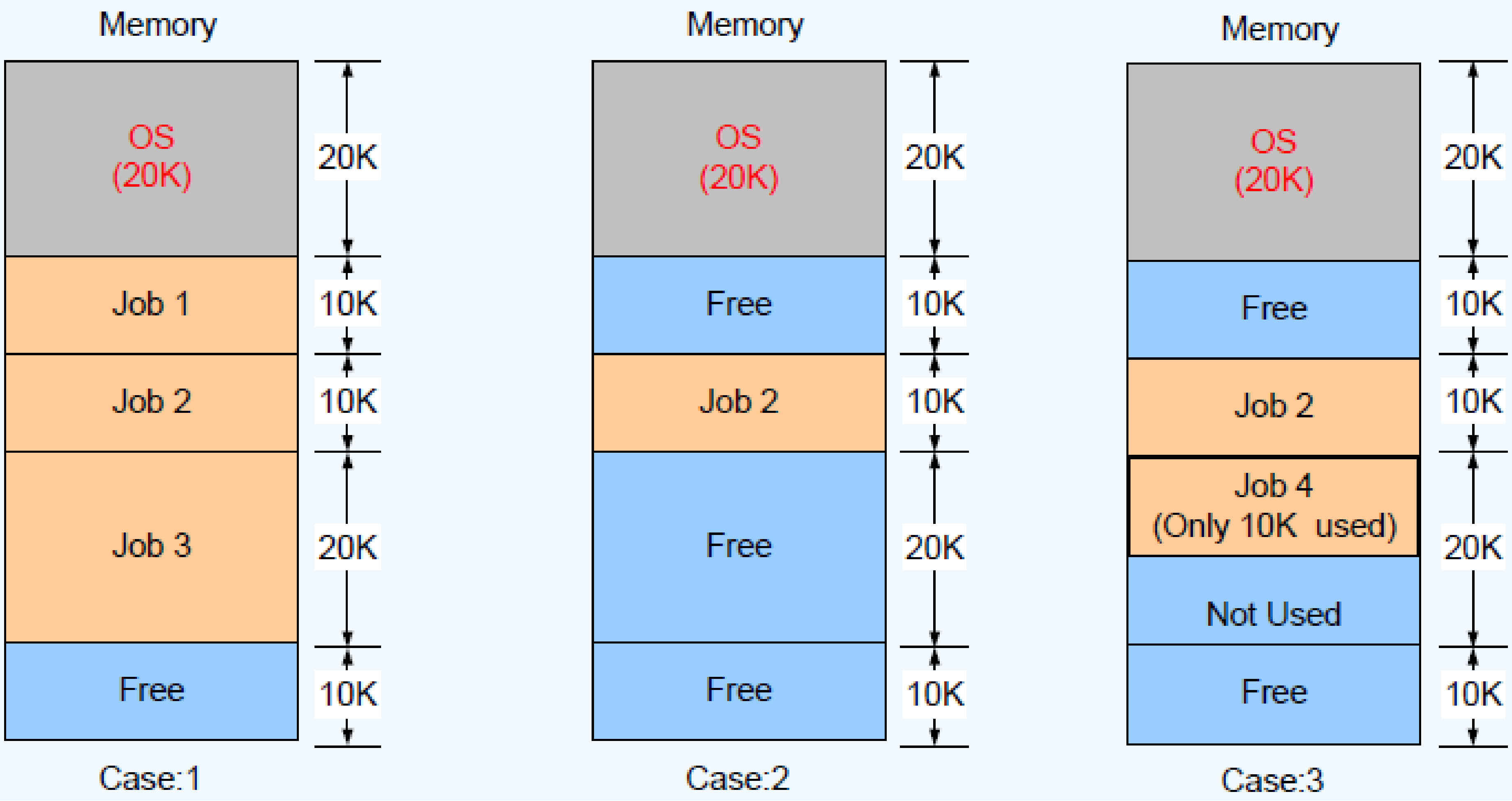
Memory management Schemes

- Single Contiguous allocation
- Partitioned allocation
- Relocatable Partitioned allocation
- Simple Paged allocation
- Demand Paging
- Segmentation

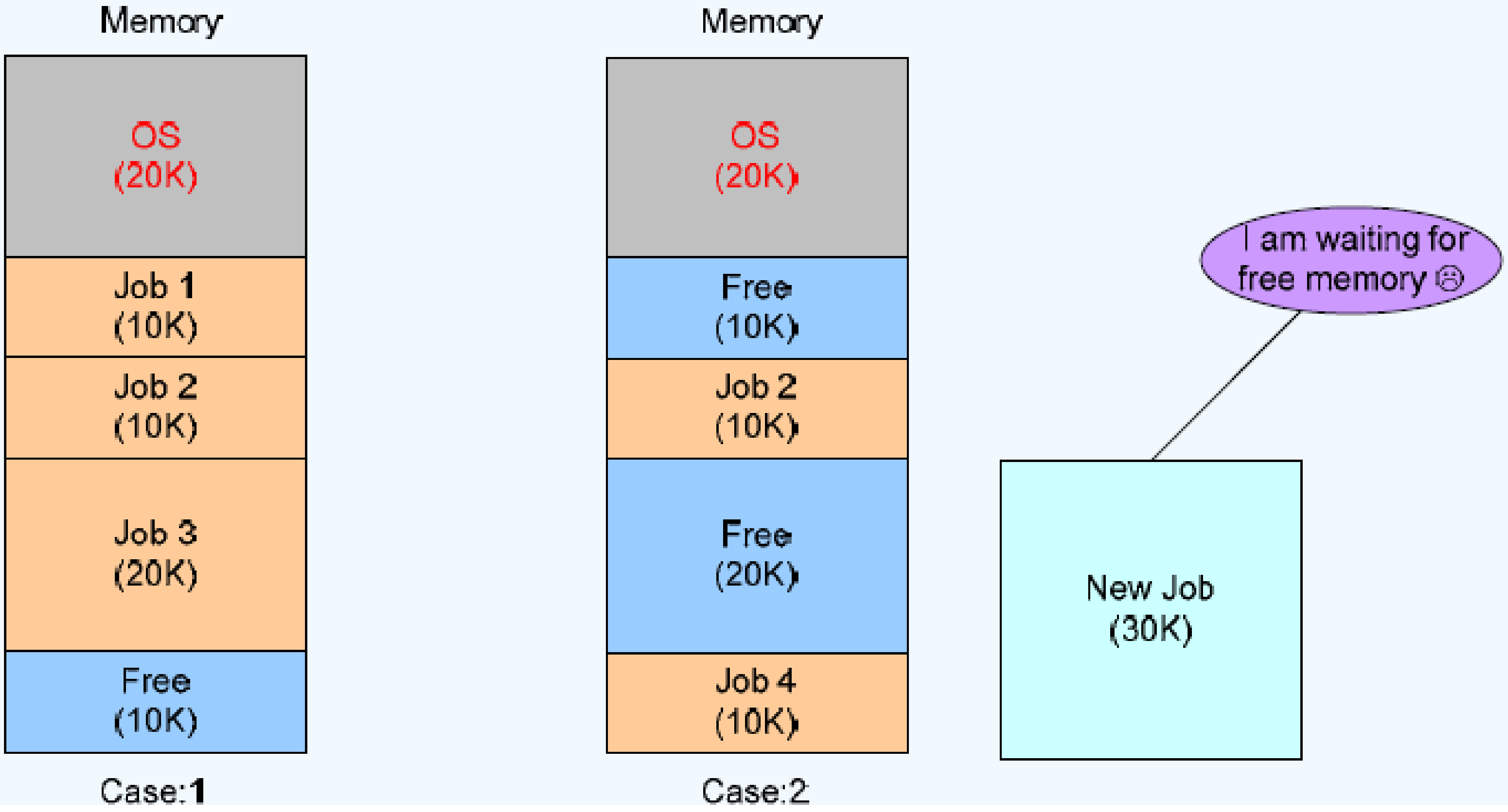
Single Contiguous Allocation



Fixed Partitioned Allocation



Variable Partitioned Allocation



Memory

OS
(20K)

Job 1
(10K)

Job 2
(10K)

Job 3
(20K)

Free
(10K)

Case:1

Memory

OS
(20K)

Free
(10K)

Job 2
(10K)

Free
(20K)

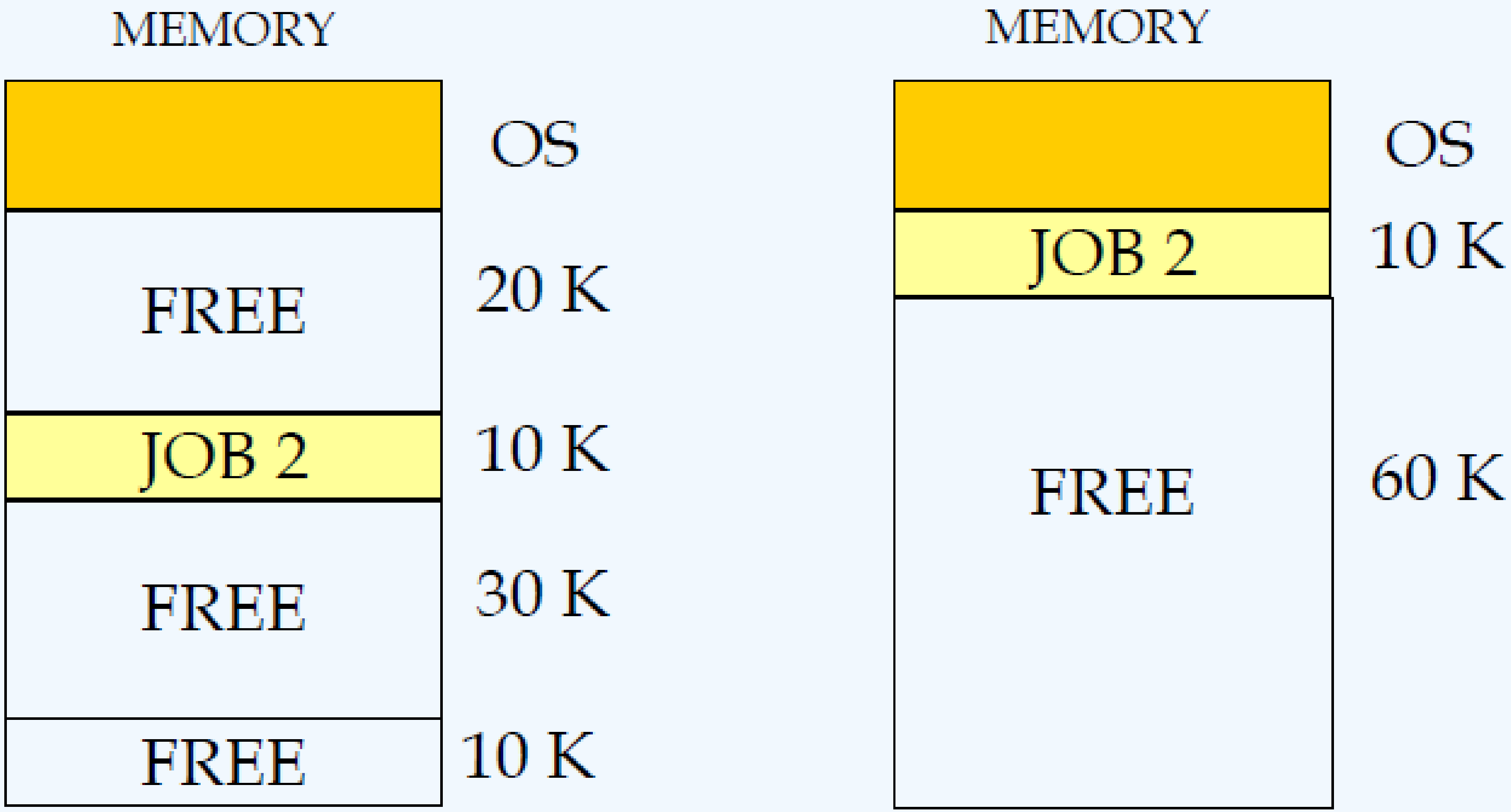
Job 4
(10K)

Case:2

New Job
(30K)

I am waiting for free memory ☹

Relocatable partitioned allocation



MEMORY

OS

FREE 20 K

JOB 2 10 K

FREE 30 K

FREE 10 K

MEMORY

OS

JOB 2 10 K

FREE 60 K

Dynamic Storage-Allocation Problem

How to satisfy a request of size *n* from a list of free holes

First Fit

In the first fit approach is to allocate the first free partition or hole large enough which can accommodate the process. It finishes after finding the first suitable free partition.

Advantage

Fastest algorithm because it searches as little as possible.

Disadvantage

The remaining unused memory areas left after allocation become waste if it is too smaller. Thus request for larger memory requirement cannot be accomplished.

Best Fit

The best fit deals with allocating the smallest free partition which meets the requirement of the requesting process. This algorithm first searches the entire list of free partitions and considers the smallest hole that is adequate. It then tries to find a hole which is close to actual process size needed.

Advantage

Memory utilization is much better than first fit as it searches the smallest free partition first available.

Disadvantage

It is slower and may even tend to fill up memory with tiny useless holes.

Worst fit

In worst fit approach is to locate largest available free portion so that the portion left will be big enough to be useful. It is the reverse of best fit.

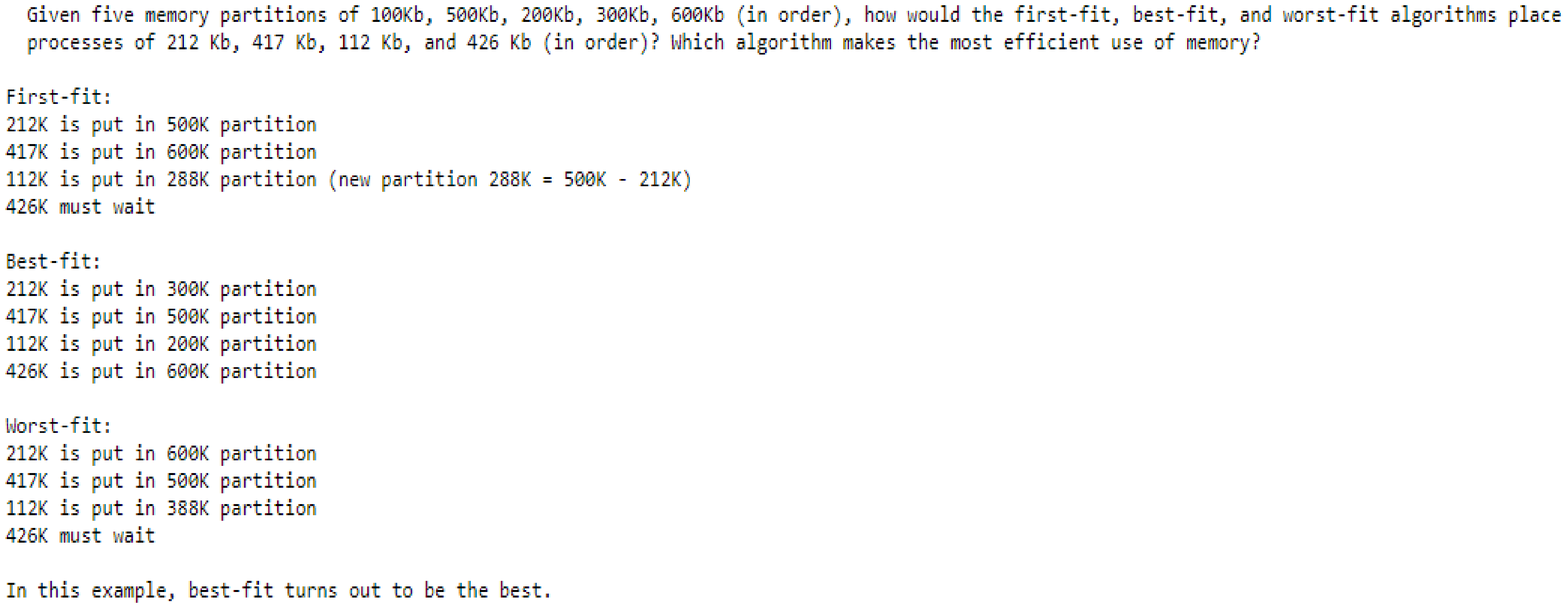
Advantage

Reduces the rate of production of small gaps.

Disadvantage

If a process requiring larger memory arrives at a later stage then it cannot be accommodated as the largest hole is already split and occupied.

Example



Given five memory partitions of 100Kb, 500Kb, 200Kb, 300Kb, 600Kb (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of 212 Kb, 417 Kb, 112 Kb, and 426 Kb (in order)? Which algorithm makes the most efficient use of memory?

First-fit:
212K is put in 500K partition
417K is put in 600K partition
112K is put in 288K partition (new partition 288K = 500K - 212K)
426K must wait

Best-fit:
212K is put in 300K partition
417K is put in 500K partition
112K is put in 200K partition
426K is put in 600K partition

Worst-fit:
212K is put in 600K partition
417K is put in 500K partition
112K is put in 388K partition
426K must wait

In this example, best-fit turns out to be the best.

Fragmentation

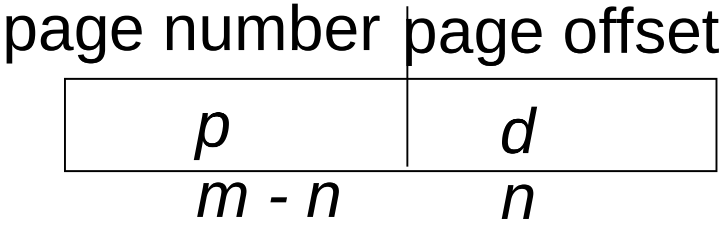
- **External Fragmentation** – total free memory space exists to satisfy a request, but it is not contiguous (the holes are too small)
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers

Paging

- Logical address space can be physically non-contiguous. Process is allocated physical memory wherever the latter is available
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size ***n*** pages, need to find *n* free frames and load program
- Set up a page table to translate logical to physical addresses
    - Page table maps each logical page to a physical frame
- Internal fragmentation

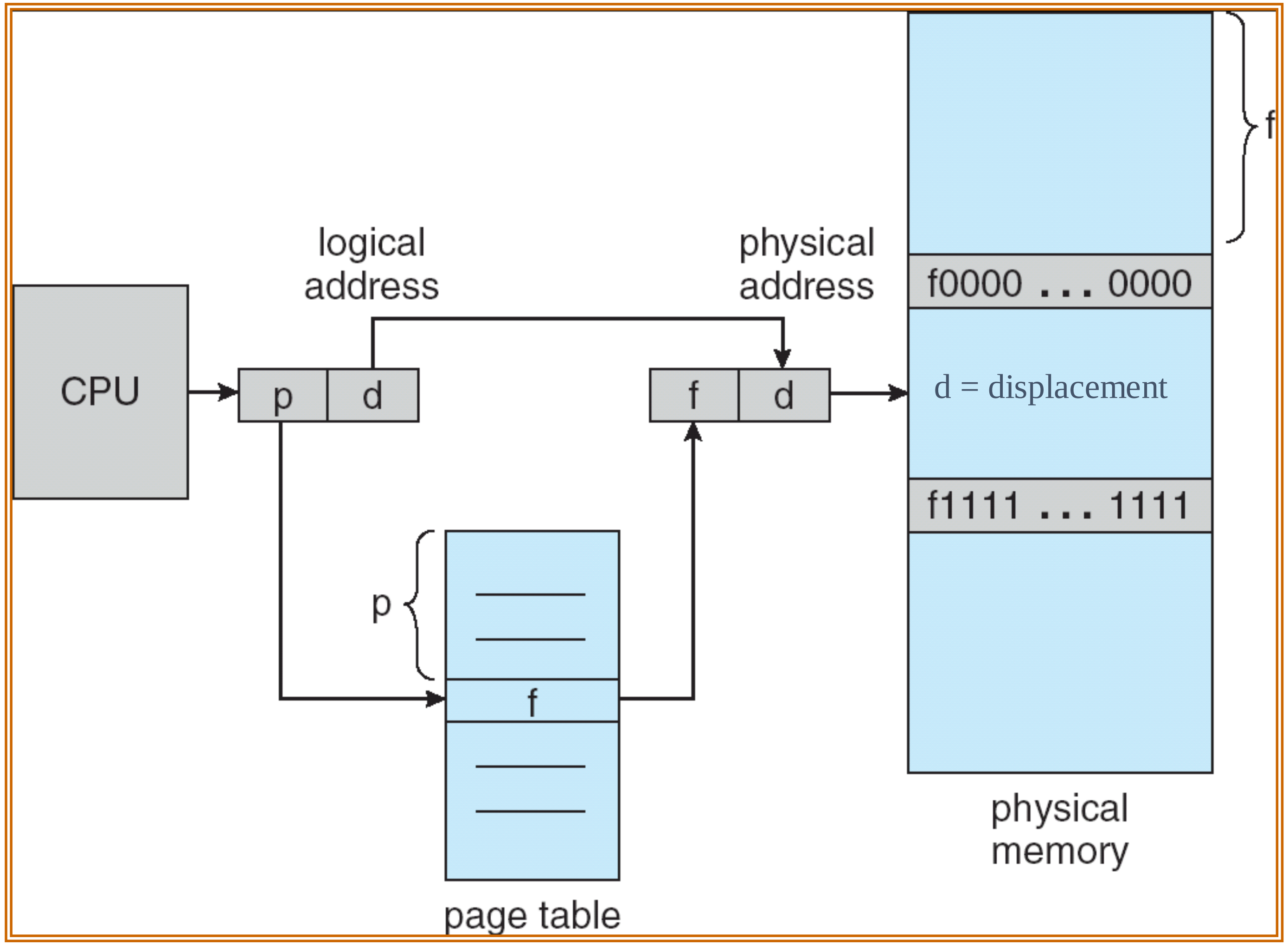
Address Translation Scheme

- Logical address generated by CPU is divided into:
  - **Page number (*p*)** – used as an index into a *page table* which contains base address of each frame in physical memory
  - **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit.
    - Offset "d" denotes the displacement in the frame

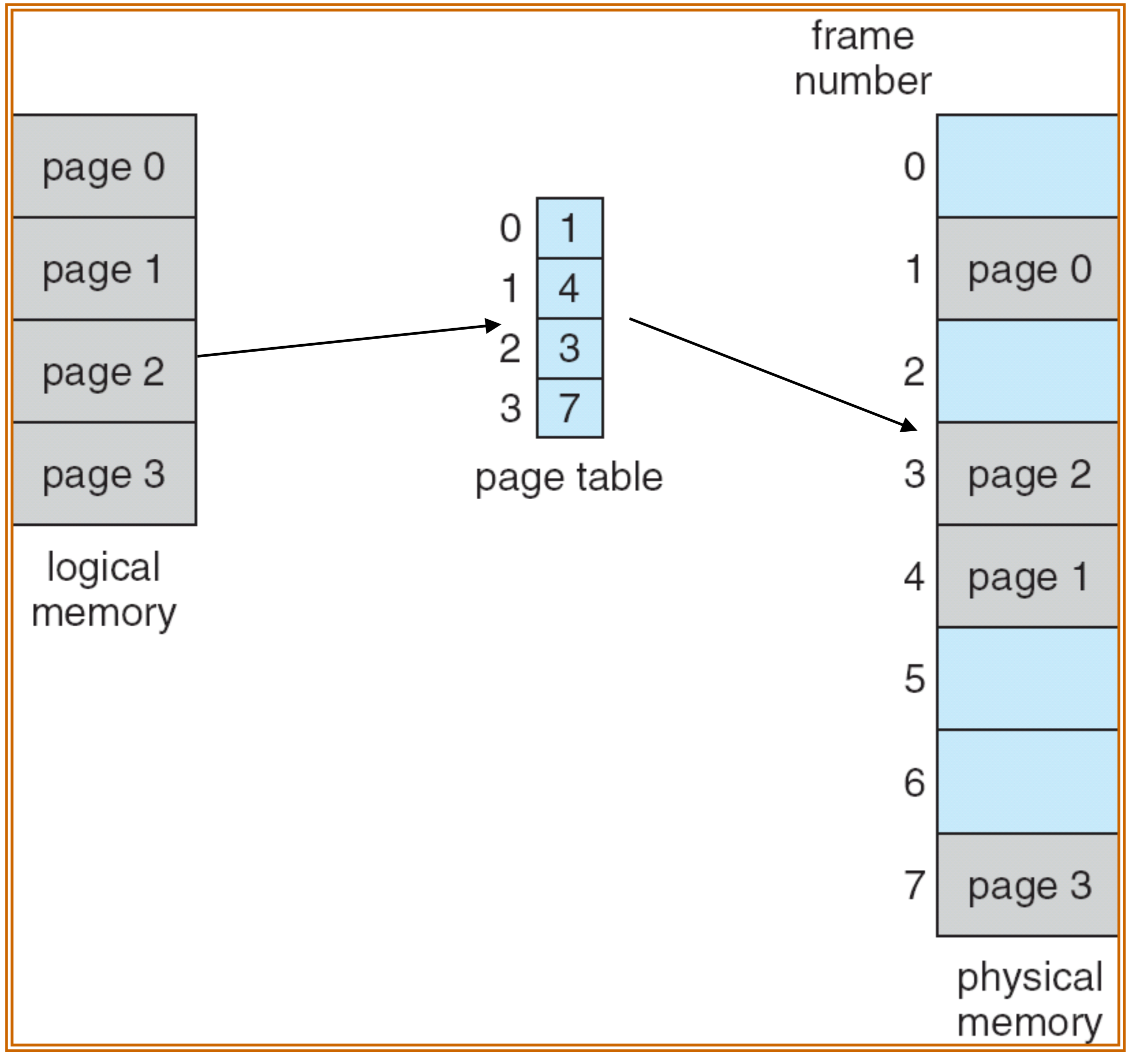


- For given logical address space 2^m *and page size* 2^n

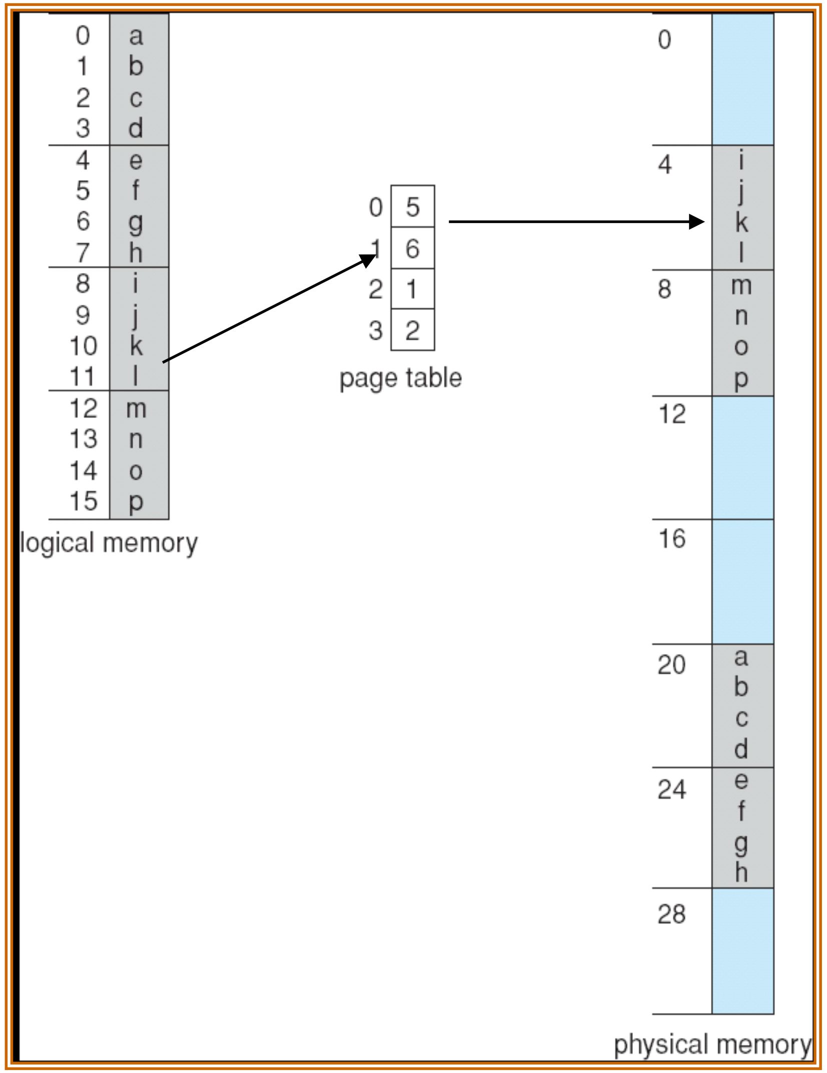
Paging Hardware



Paging Model of Logical and Physical Memory



Paging Example



Byte addressable:
8 pages total with
4-byte per page.

Page index = 3 bits
Page offset = 2 bits

Example:
Logical Address 13
Binary rep = 01101

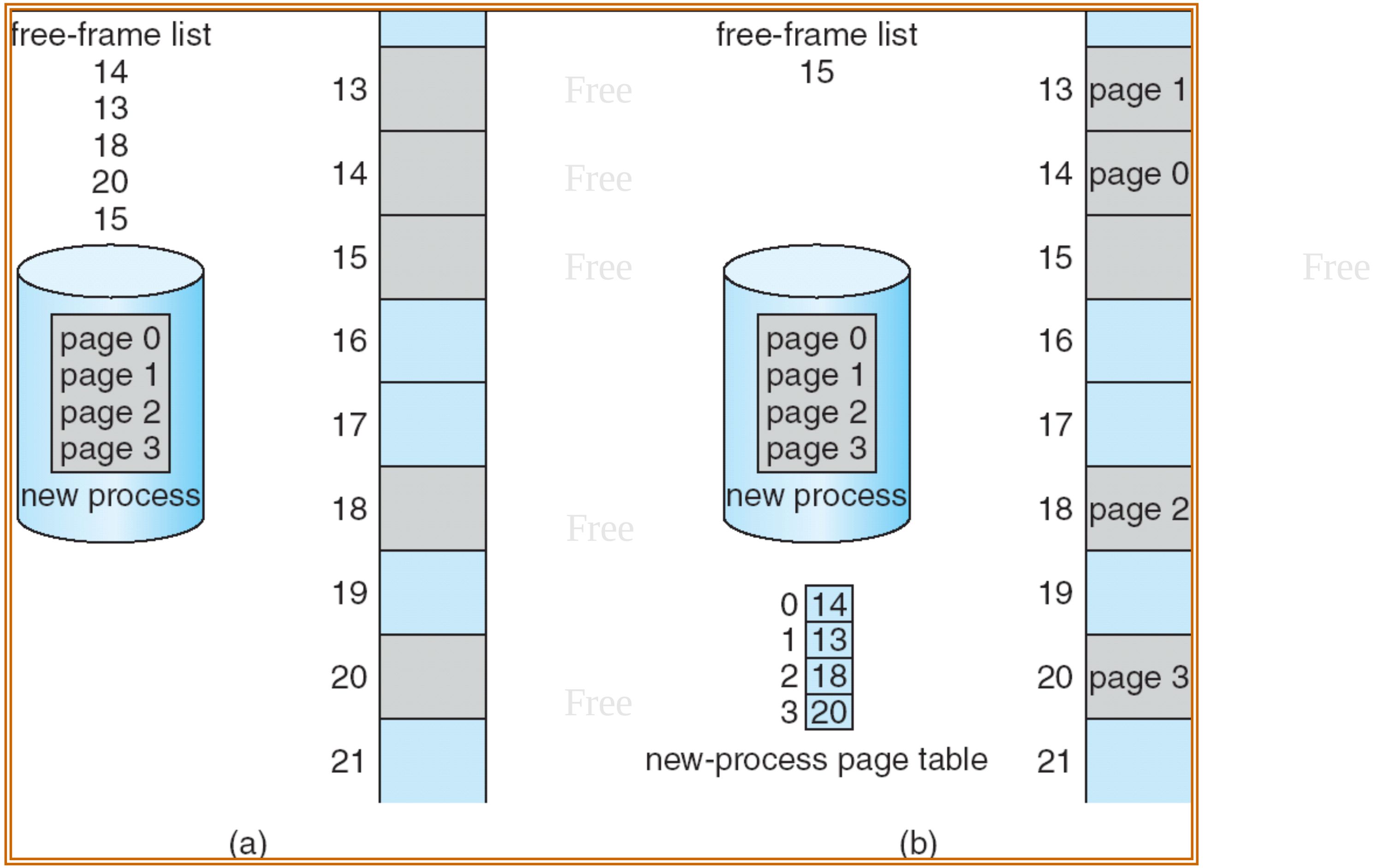
Page index = 011 = 3
Page offset = 01 = 1

Page[3] = Frame 2

Physical Address =
2*|Frame|+Offset = 9

32-byte memory and 4-byte pages

Free Frames – Managed by Frame Table



Before allocation

After allocation

Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PRLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

Associative Memory

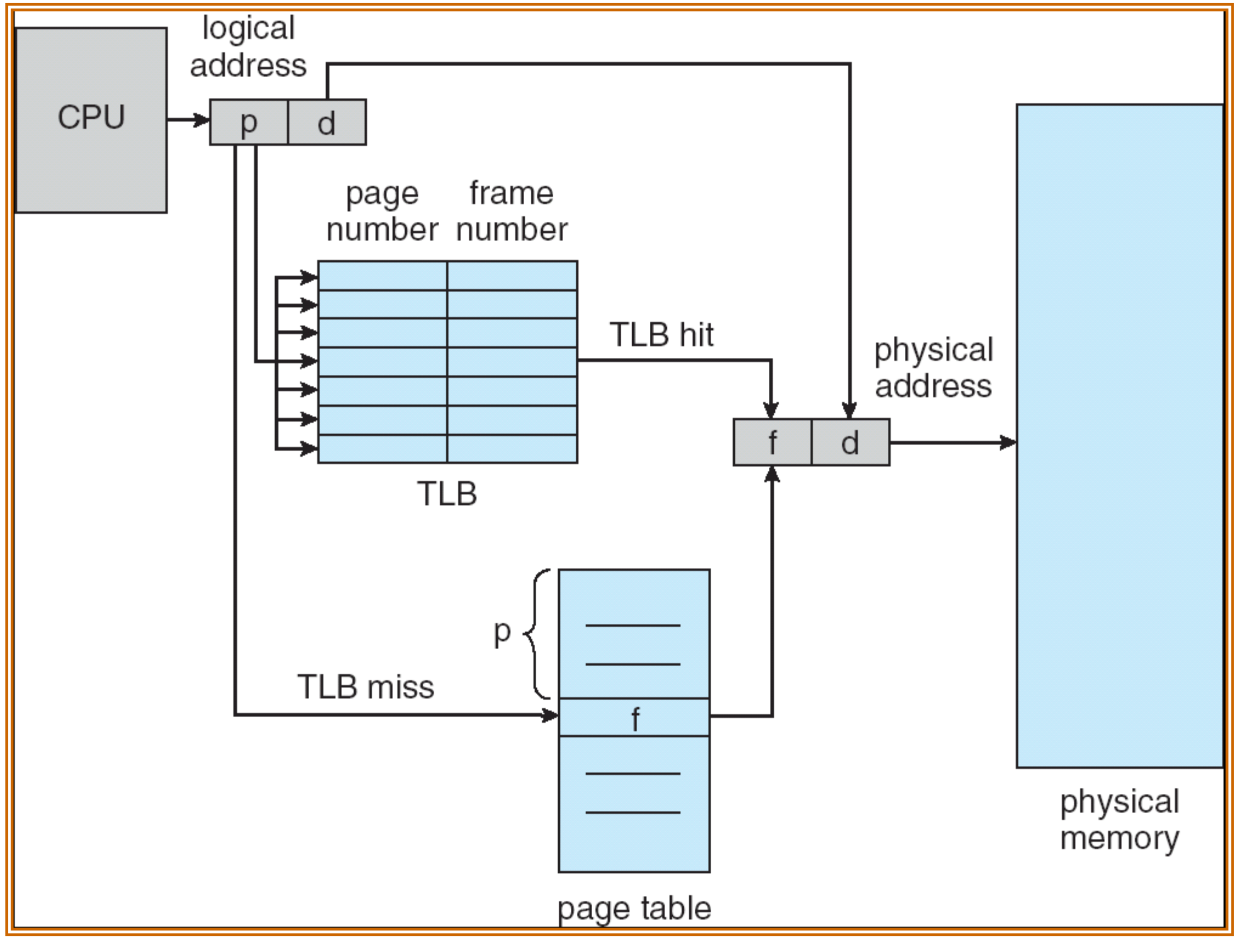
- Associative memory – parallel search

Page #	Frame #

Address translation (p, d)

- If p is in associative register, get frame # out
- Otherwise get frame # from page table in memory

Paging Hardware With TLB



Effective Access Time

- Associative Lookup = ε time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio = α
  - Percentage of times that a page is found in the TLB
  - Ratio related to number of associative registers
- **Effective Access Time** (EAT)

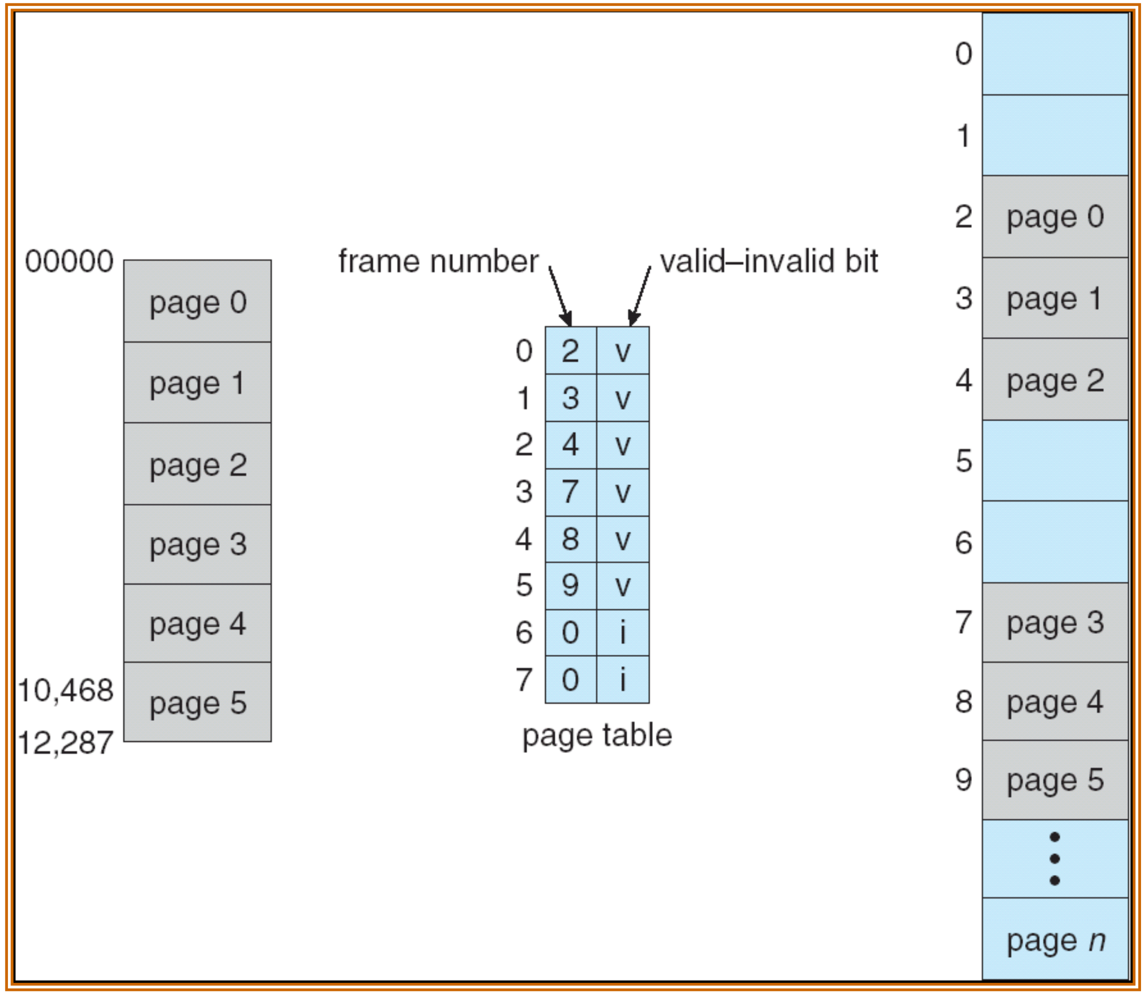
$$\begin{aligned} \text{EAT} &= (1 + \varepsilon)\ \alpha + (2 + \varepsilon)(1 - \alpha) \\ &= 2 + \varepsilon - \alpha \end{aligned}$$

Memory Protection

- Memory protection implemented by associating protection bit with each frame

- **Valid-invalid** bit attached to each entry in the page table:
  - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
  - “invalid” indicates that the page is not in the process’ logical address space

Valid (v) or Invalid (i) Bit In A Page Table



	frame number	valid–invalid bit
0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i

Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

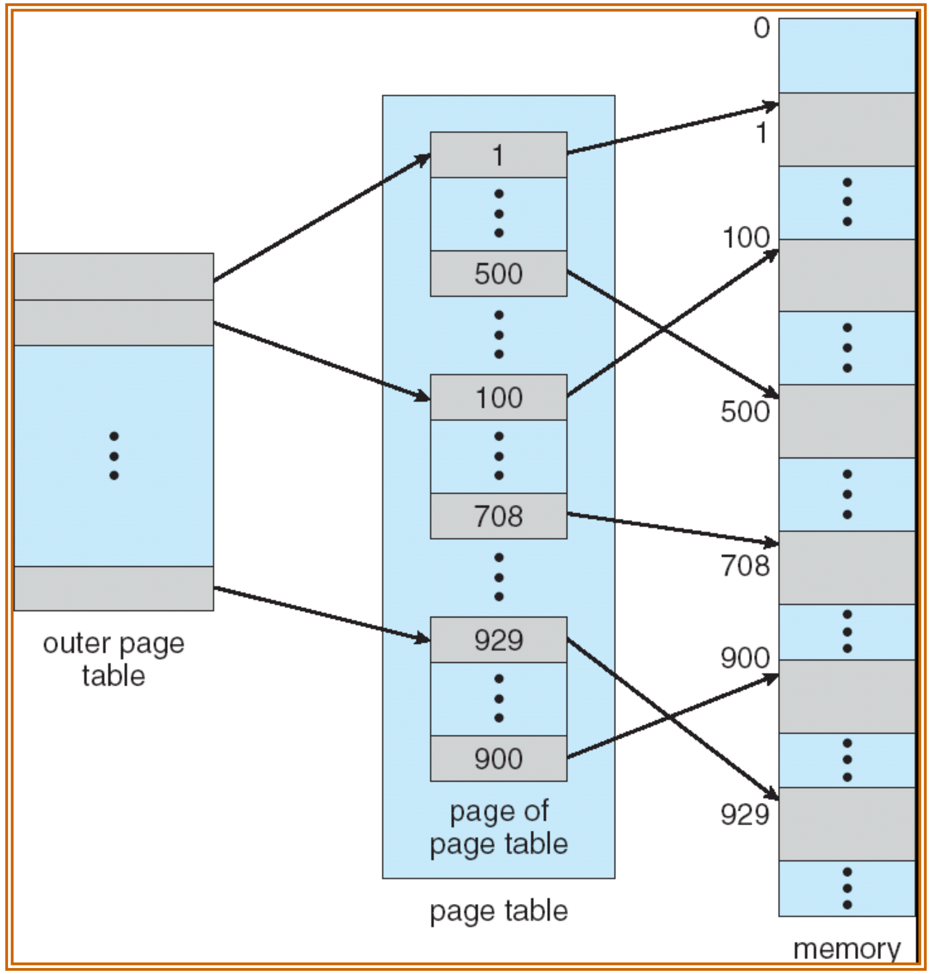
Hierarchical Page Tables

- Modern computer systems support large logical address spaces
  - Consider a system with a 32-bit logical address space
  - Suppose the page size is 4KB (2^12 bits)
  - Page table may contain 1 million entries (32-12 = 2^20
  - If each entry (word) is 4 bytes, then the page table alone will require 4MB of physical address space

- Should this be allocated contiguously?
  - Certainly possible, but leads to external fragmentation
  - Instead, break logical address space into multiple page tables

- A simple technique is a two-level page table

Two-Level Page-Table Scheme



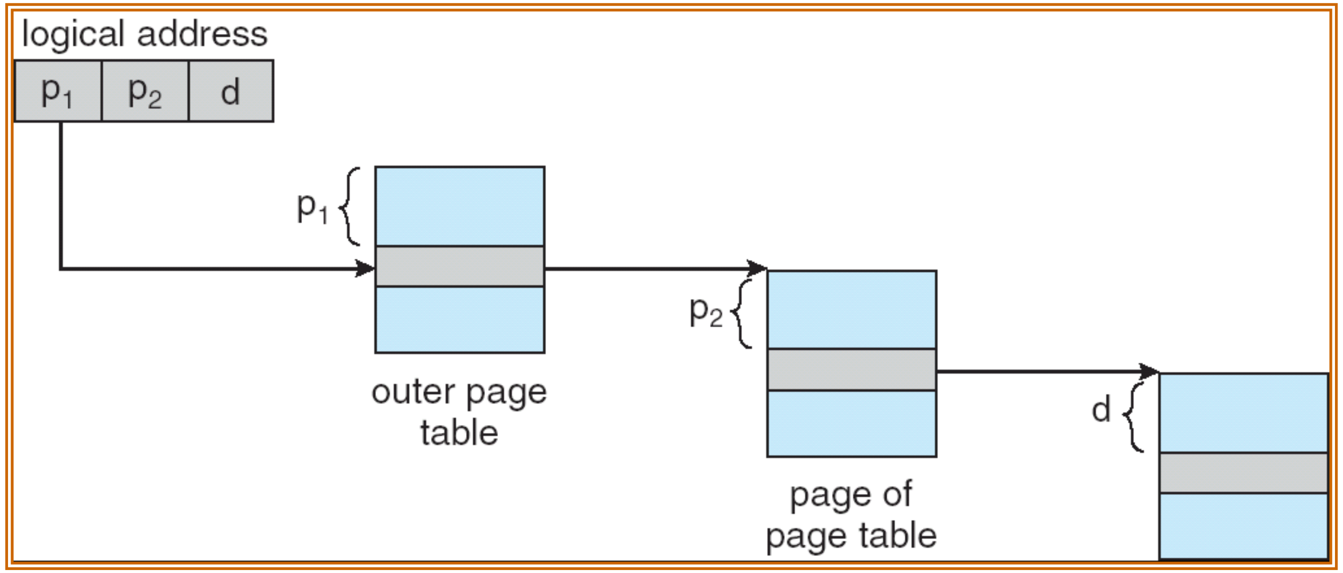
Two-Level Paging Example

- **A logical address (on 32-bit machine with 1K page size) is divided into:**
  - **a page number consisting of 22 bits**
  - **a page offset consisting of 10 bits**
- **Since the page table is paged, the page number is further divided into:**
  - **a 12-bit page number**
  - **a 10-bit page offset**
- **Thus, a logical address is as follows:**

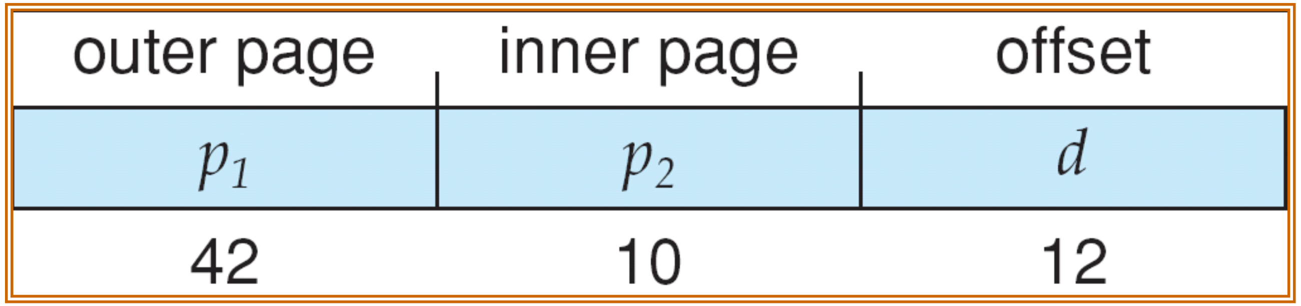
page number		page offset
p_i	p_2	d
12	10	10

where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

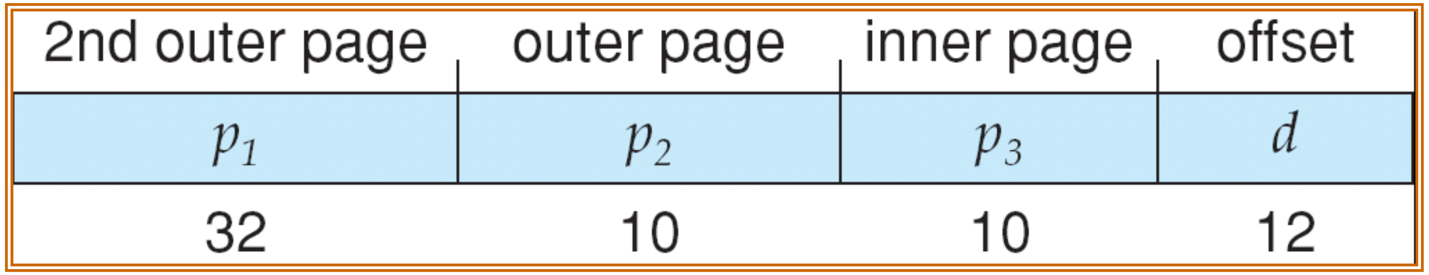
Address-Translation Scheme



Three-level Paging Scheme



outer page	inner page	offset
p_1	p_2	d
42	10	12



2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

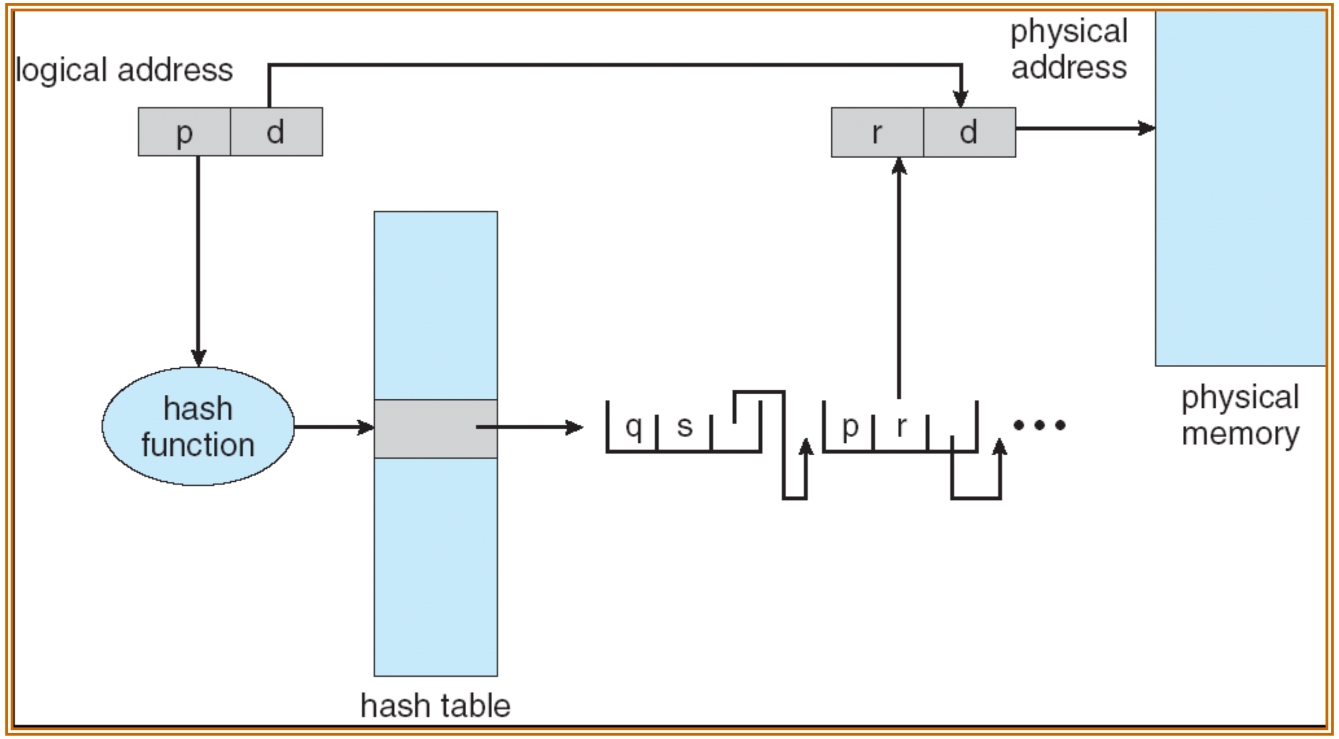
Hashed Page Tables

- Common in address spaces > 32 bits

- The virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location.

- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.

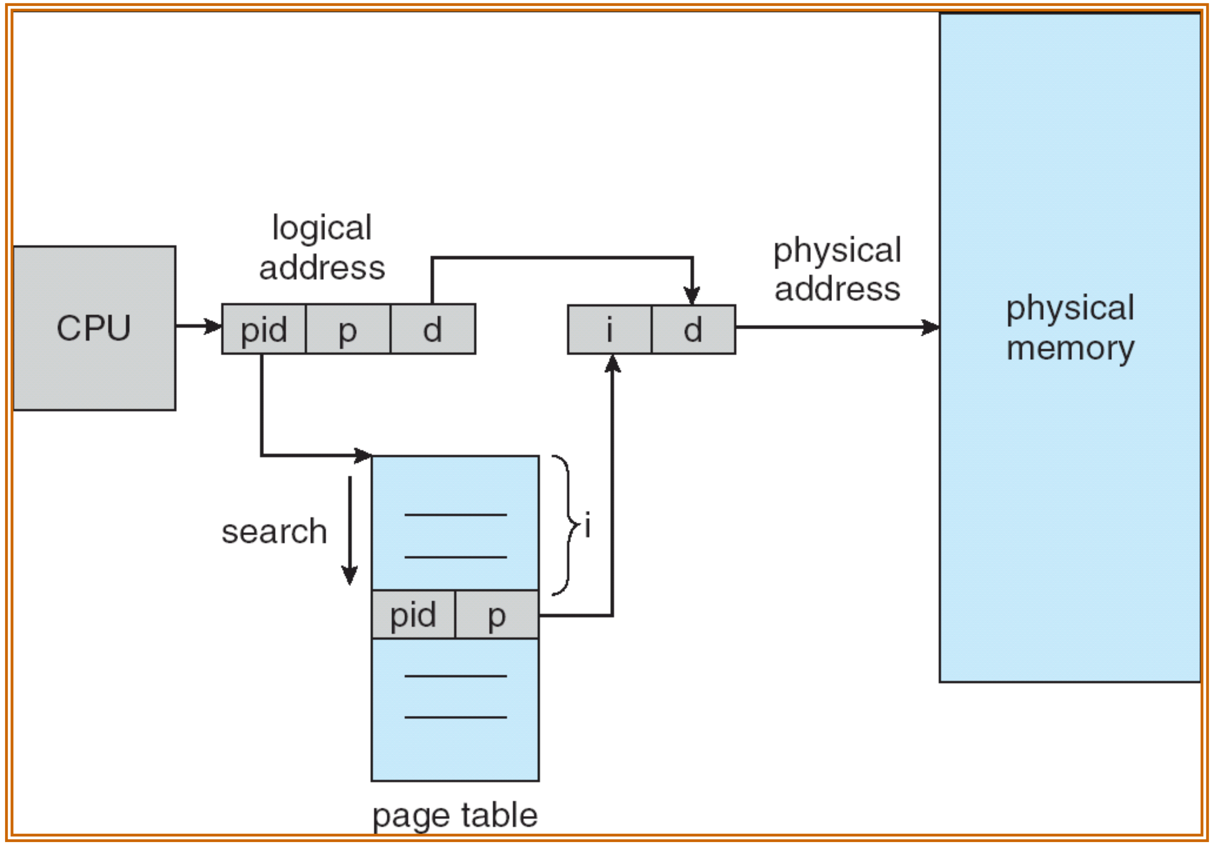
Hashed Page Table



Inverted Page Table

- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries

Inverted Page Table Architecture



Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments. A segment is a logical unit such as:

main program,

procedure,

function,

method,

object,

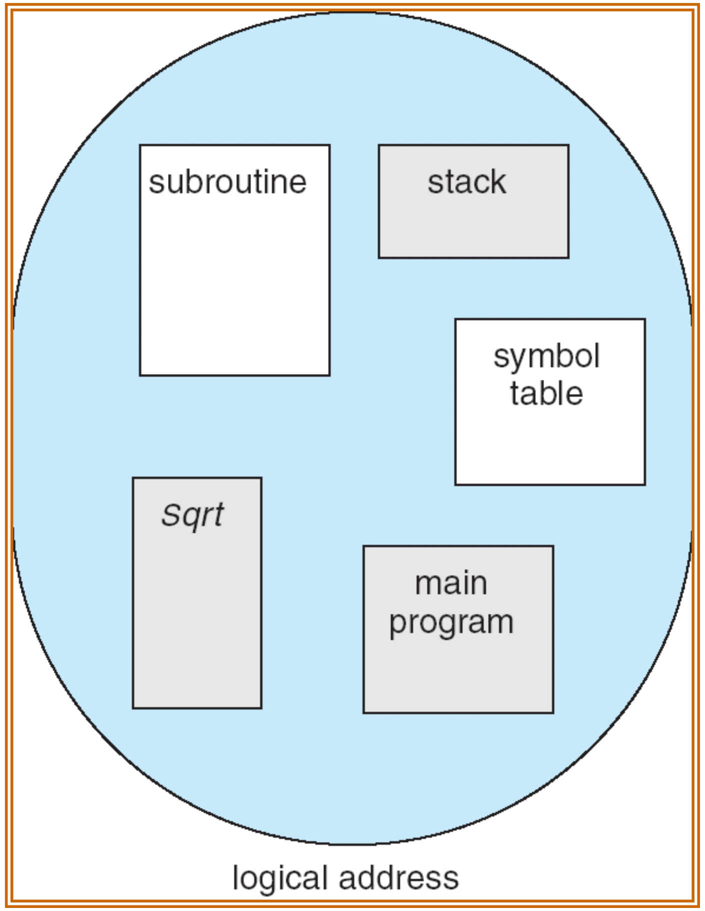
local variables, global variables,

common block,

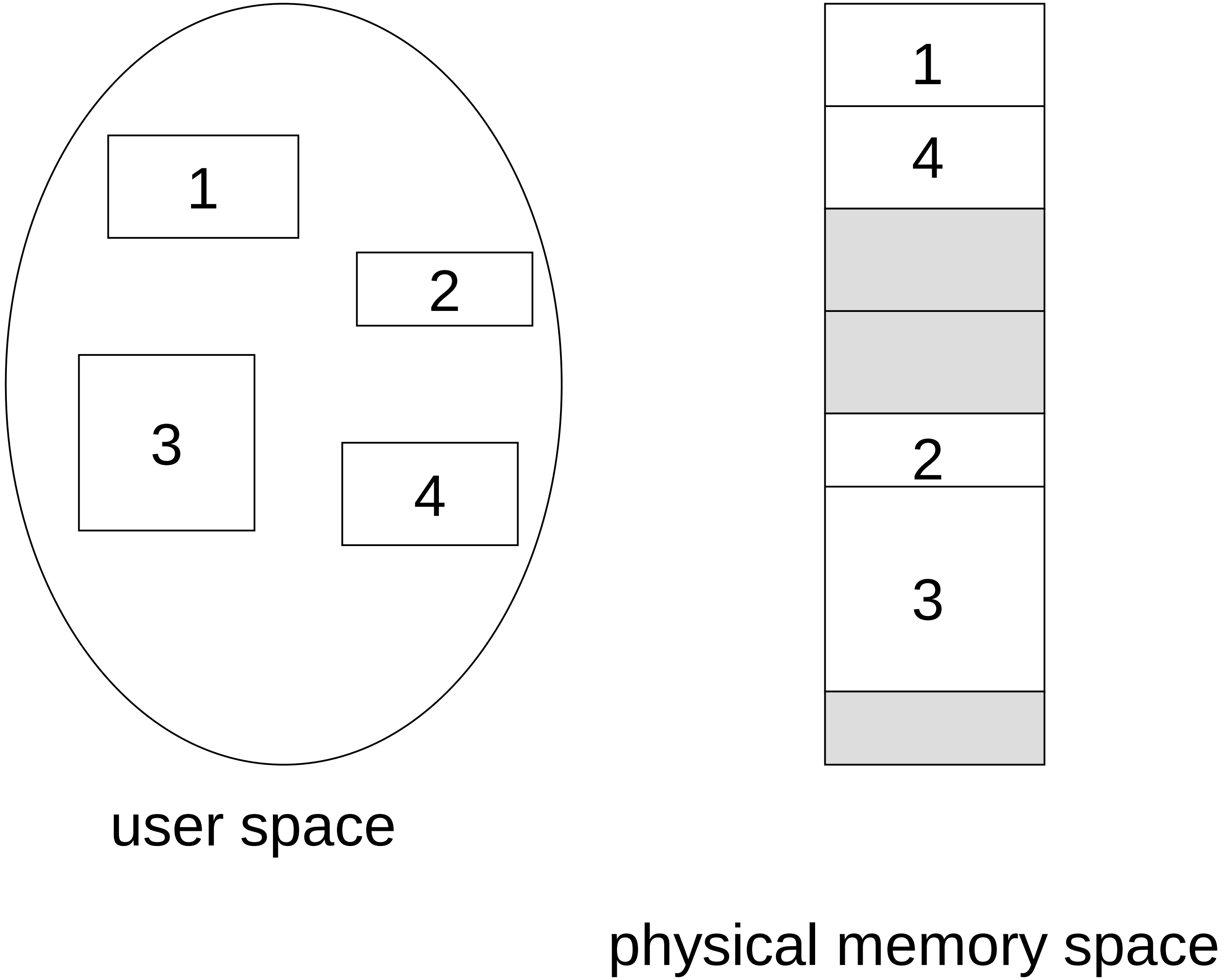
stack,

symbol table, arrays

User’s View of a Program



Logical View of Segmentation



Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>,

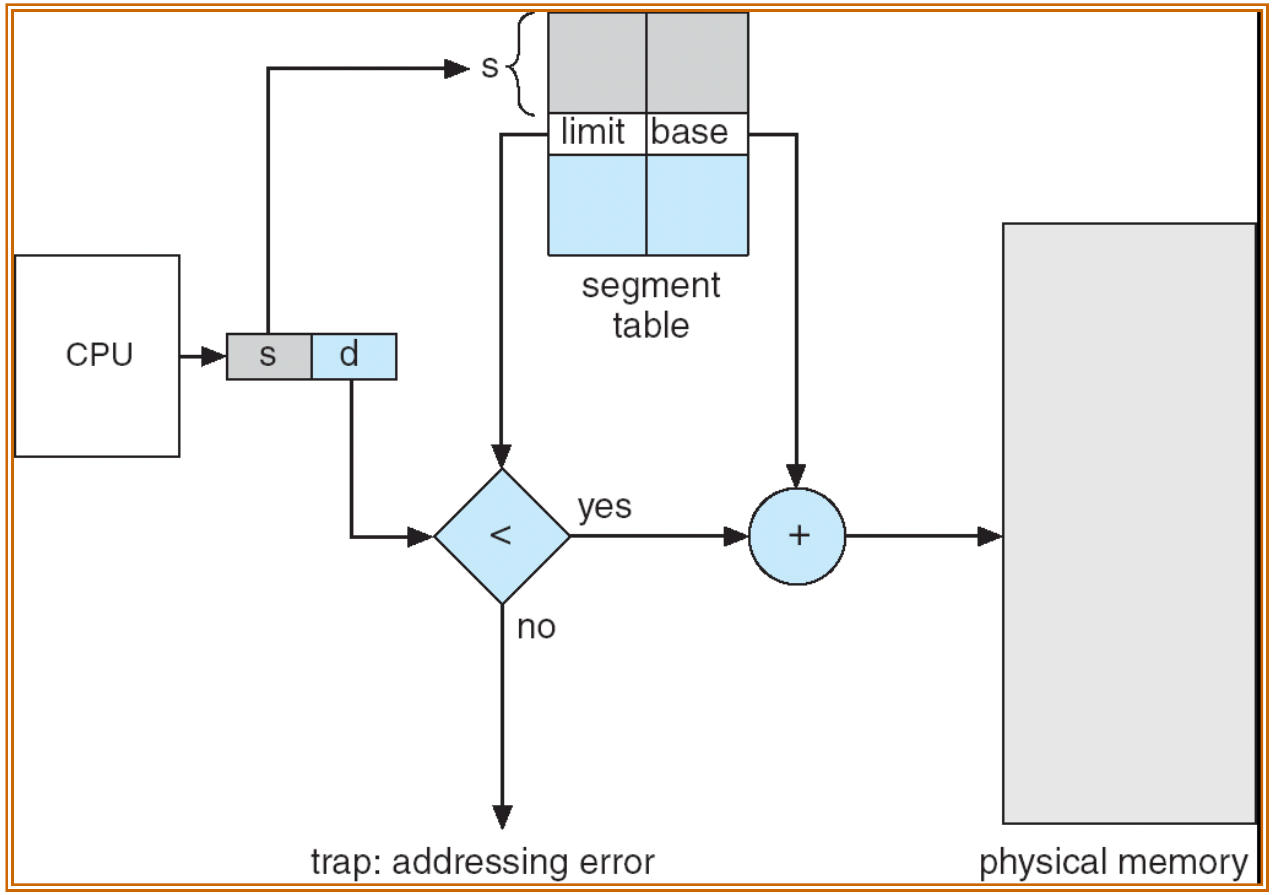
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;

  segment number ***s*** is legal if ***s*** < **STLR**

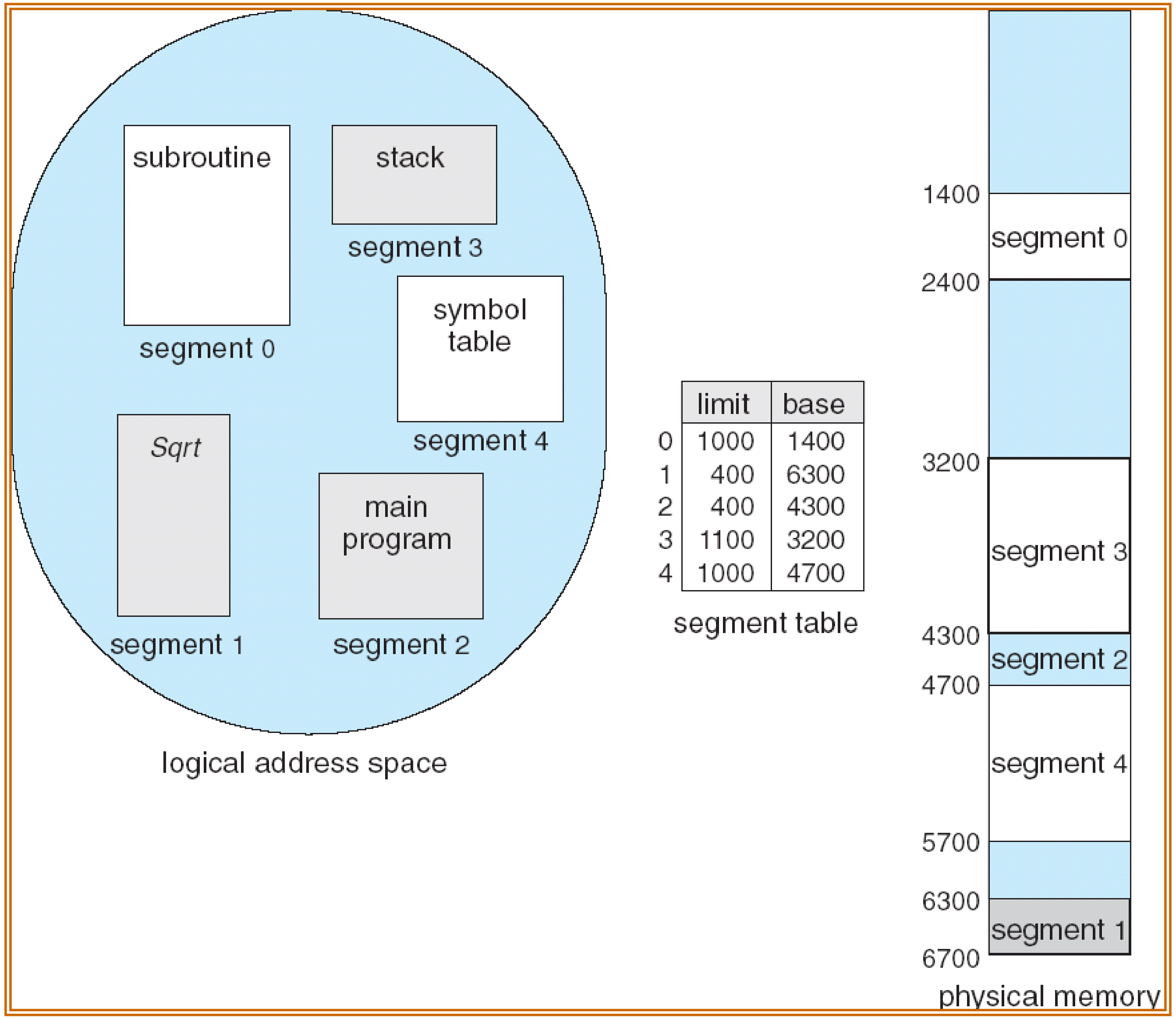
Segmentation Architecture (Cont.)

- Protection
    - With each entry in segment table associate:
        - (validation bit = 0) $\Rightarrow$ illegal segment
        - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

Segmentation Hardware



Example of Segmentation



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table

Virtual Memory

- Background
- Demand Paging
- Page Replacement
- Allocation of Frames
- Thrashing

Objectives

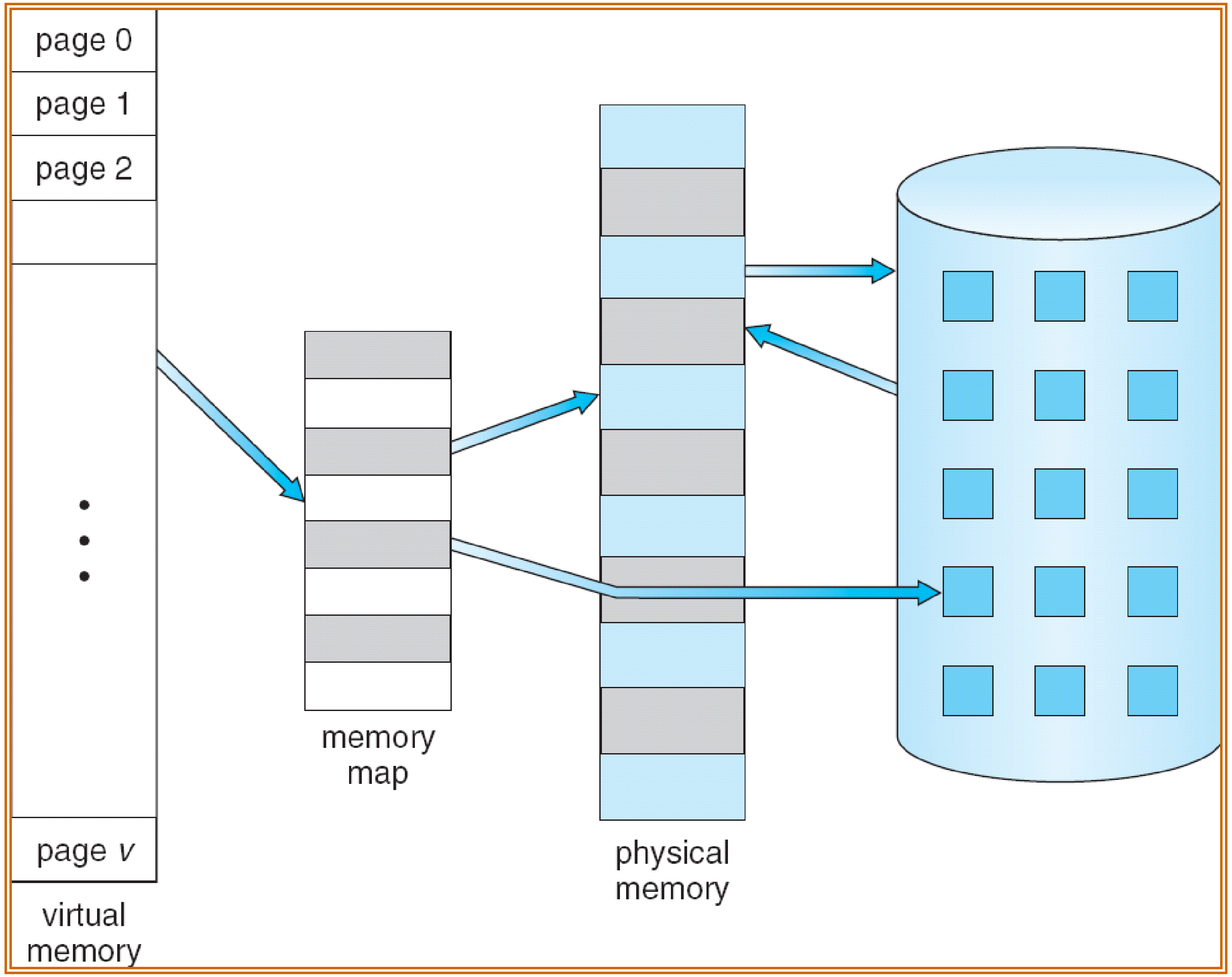
- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- To discuss the principle of the working-set model

Background

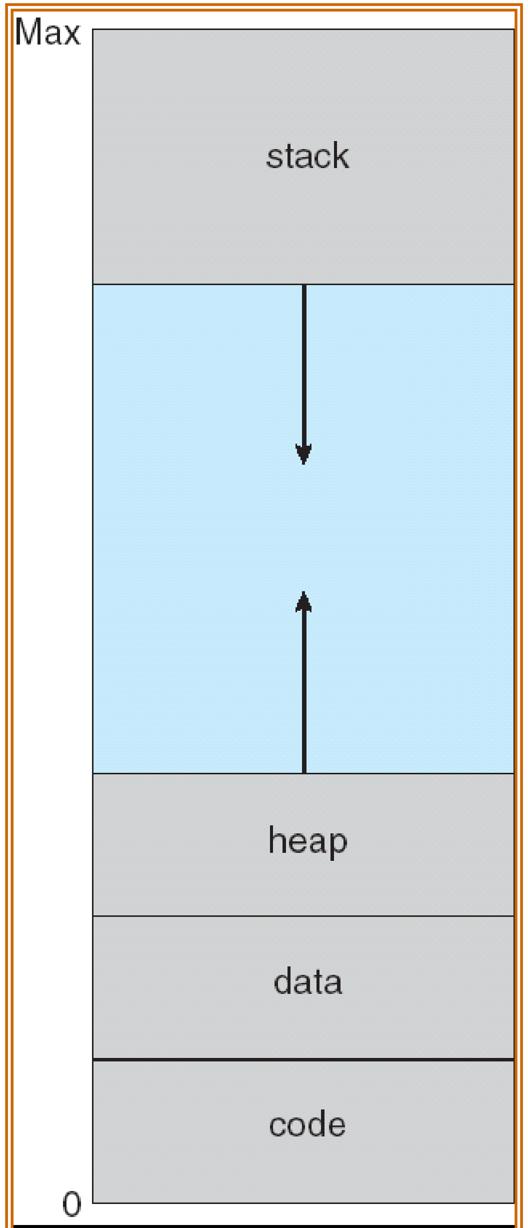
- **Virtual memory** – separation of user logical memory from physical memory.
    - Only part of the program needs to be in memory for execution
    - Logical address space can therefore be much larger than physical address space
    - Allows address spaces to be shared by several processes
    - Allows for more efficient process creation

- Virtual memory can be implemented via:
    - Demand paging
    - Demand segmentation

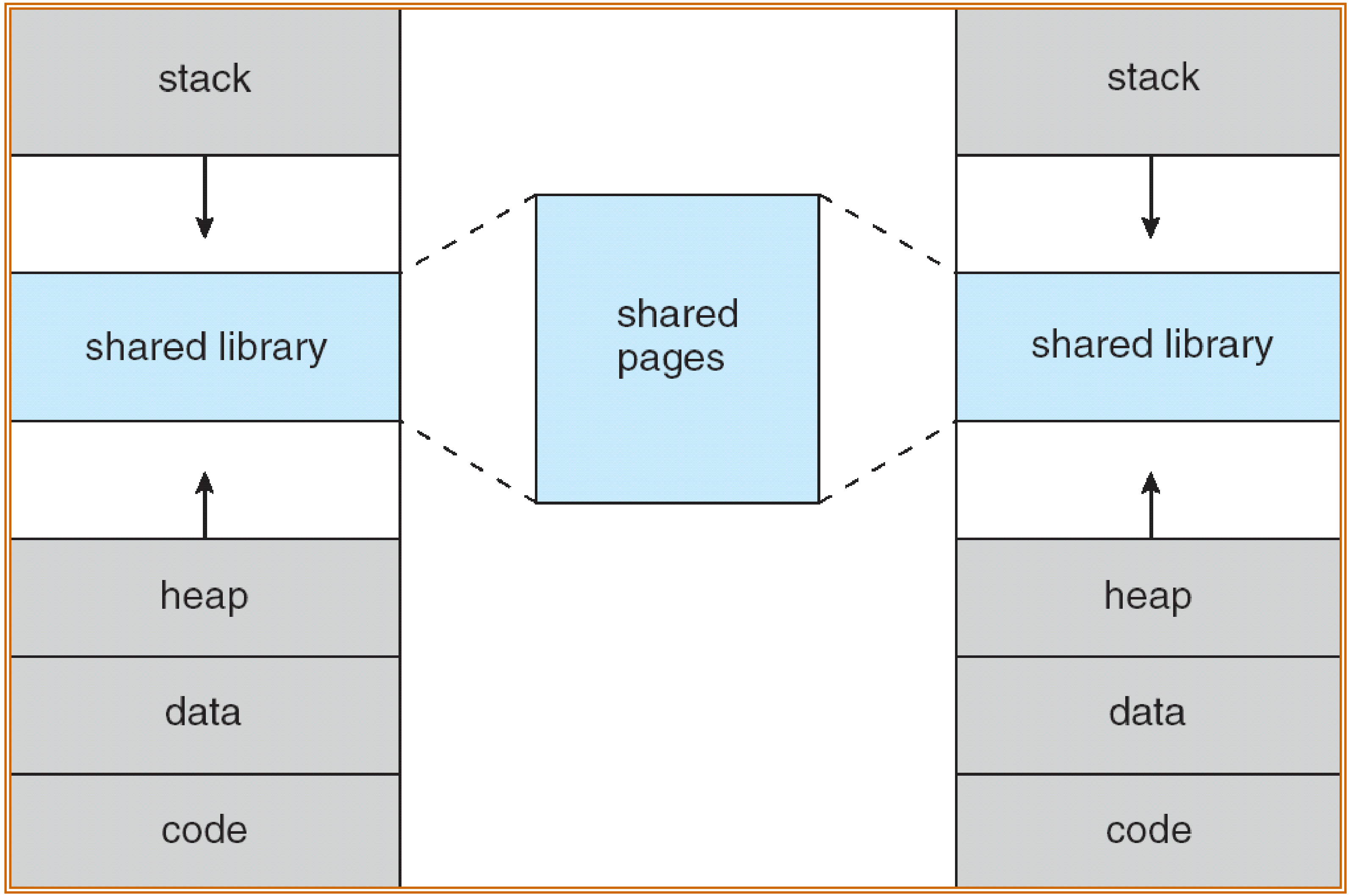
Virtual Memory That is Larger Than Physical Memory



Virtual-address Space



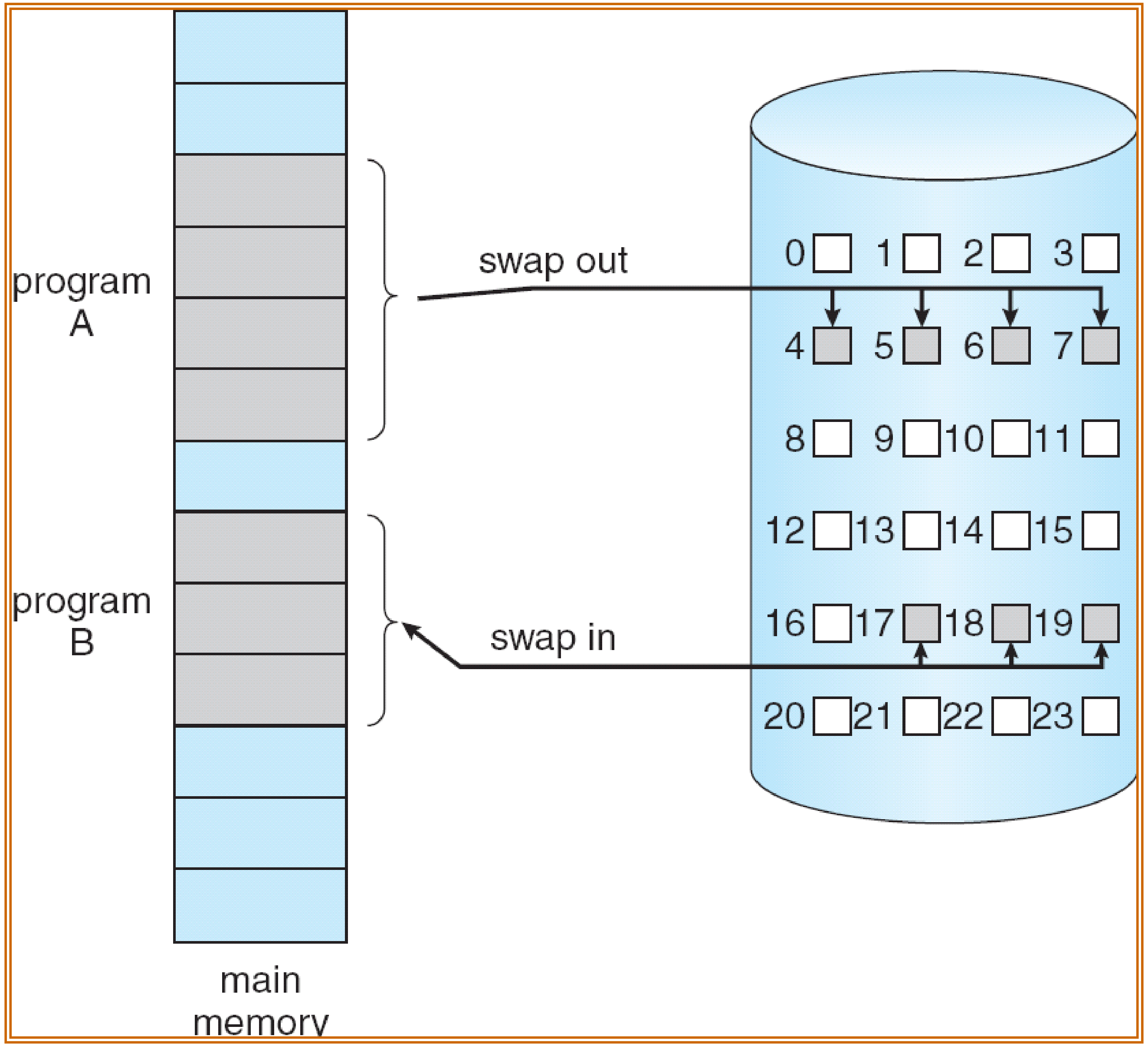
Shared Library Using Virtual Memory



Demand Paging

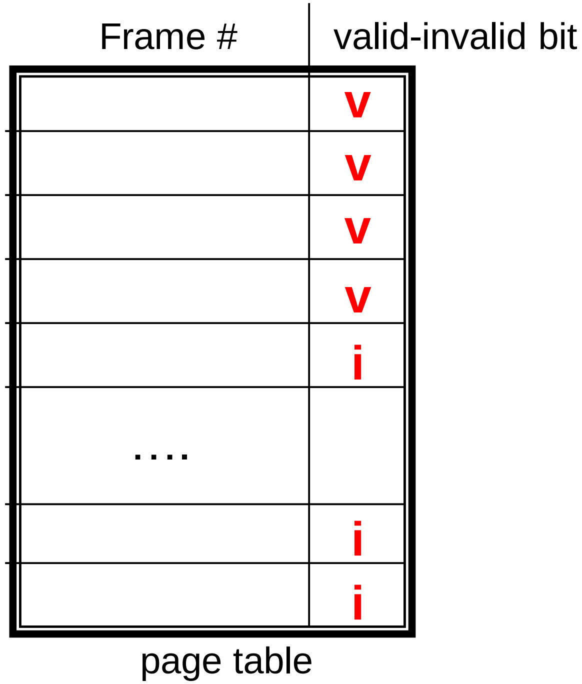
- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

Transfer of a Paged Memory to Contiguous Disk Space



Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:



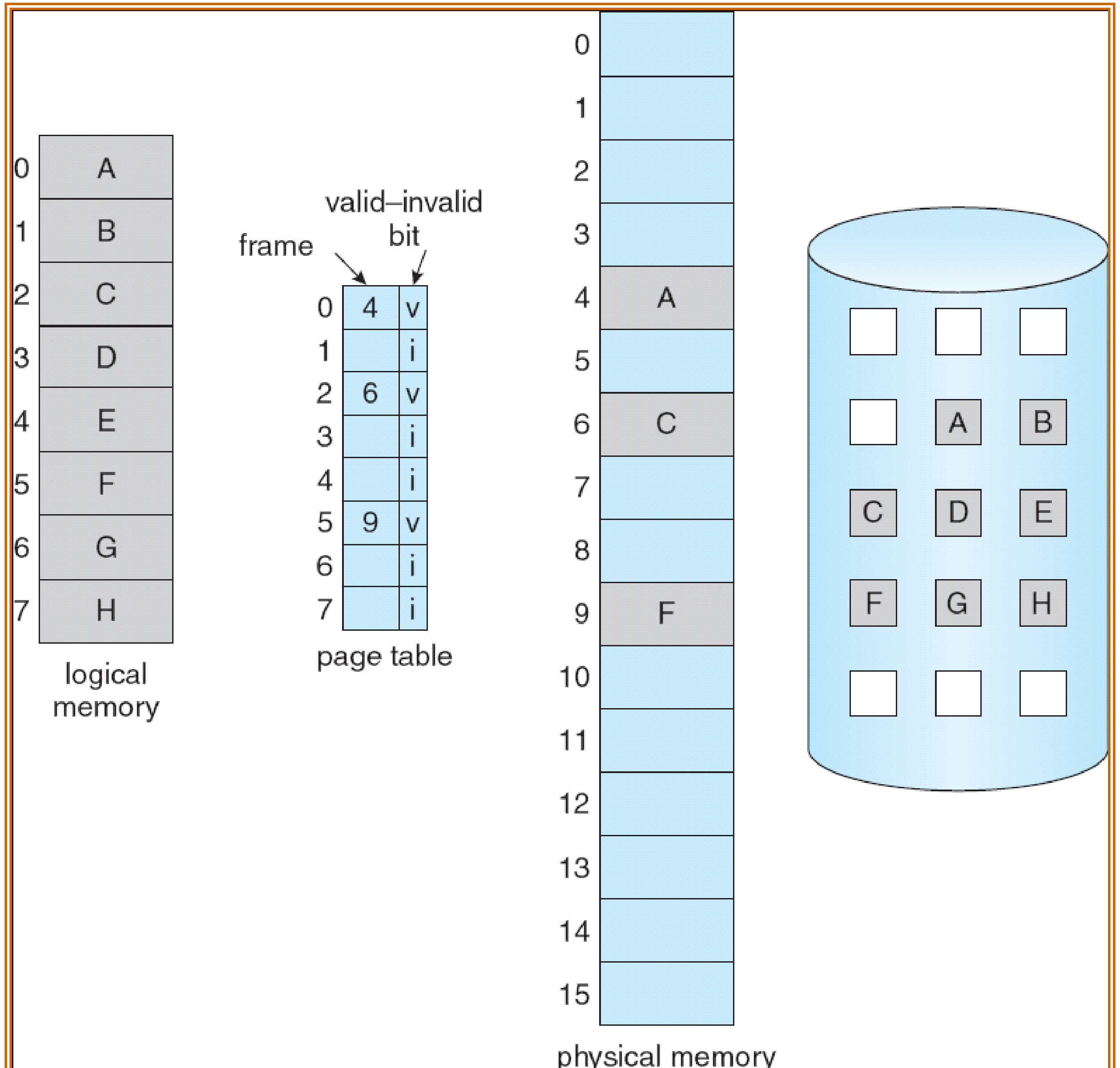
Frame #	valid-invalid bit
	v
	v
	v
	v
	i
….	
	i
	i

page table

- During address translation, if valid–invalid bit in page table entry

  is **I** ⇒ page fault

Page Table When Some Pages Are Not in Main Memory



	frame	valid–invalid bit
0	4	v
1		i
2	6	v
3		i
4		i
5	9	v
6		i
7		i

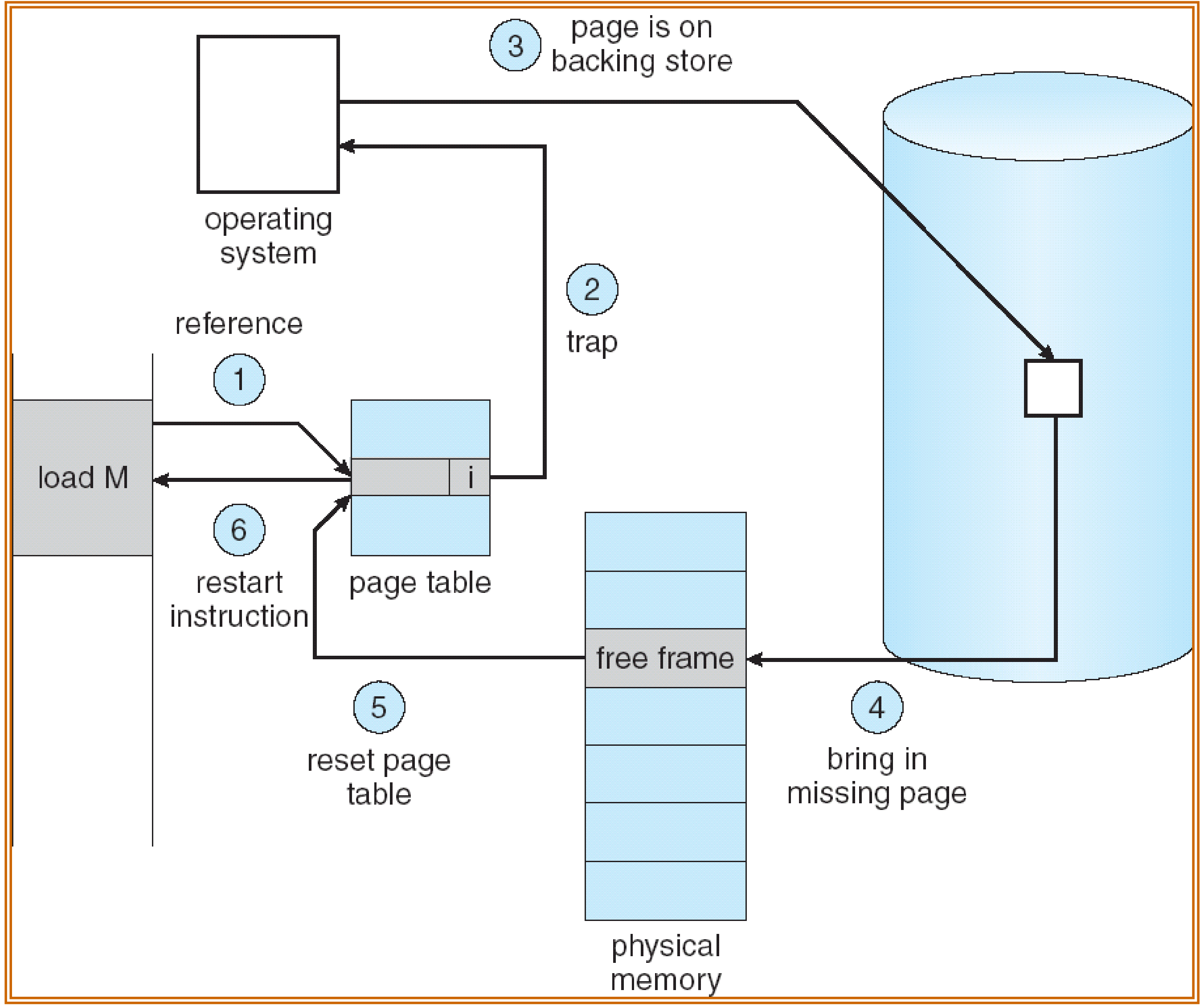
Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system:

  page fault

1. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
2. Get empty frame
3. Swap page into frame
4. Reset tables
5. Set validation bit = **v**
6. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Performance of Demand Paging

- Page Fault Rate $0 \le p \le 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault

- Effective Access Time (EAT)

$$
\begin{aligned}
\text{EAT} = {} & (1 - p) \times \text{memory access} \\
& + p\,(\text{page fault overhead} \\
& \qquad + \text{swap page out} \\
& \qquad + \text{swap page in} \\
& \qquad + \text{restart overhead} \\
& \quad)
\end{aligned}
$$

Demand Paging Example

- Memory access time = 200 nanoseconds

- Average page-fault service time = 8 milliseconds

- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p) x 200 + p x 8,000,000

  = 200 + p x 7,999,800

- If one access out of 1,000 causes a page fault, then

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!

What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

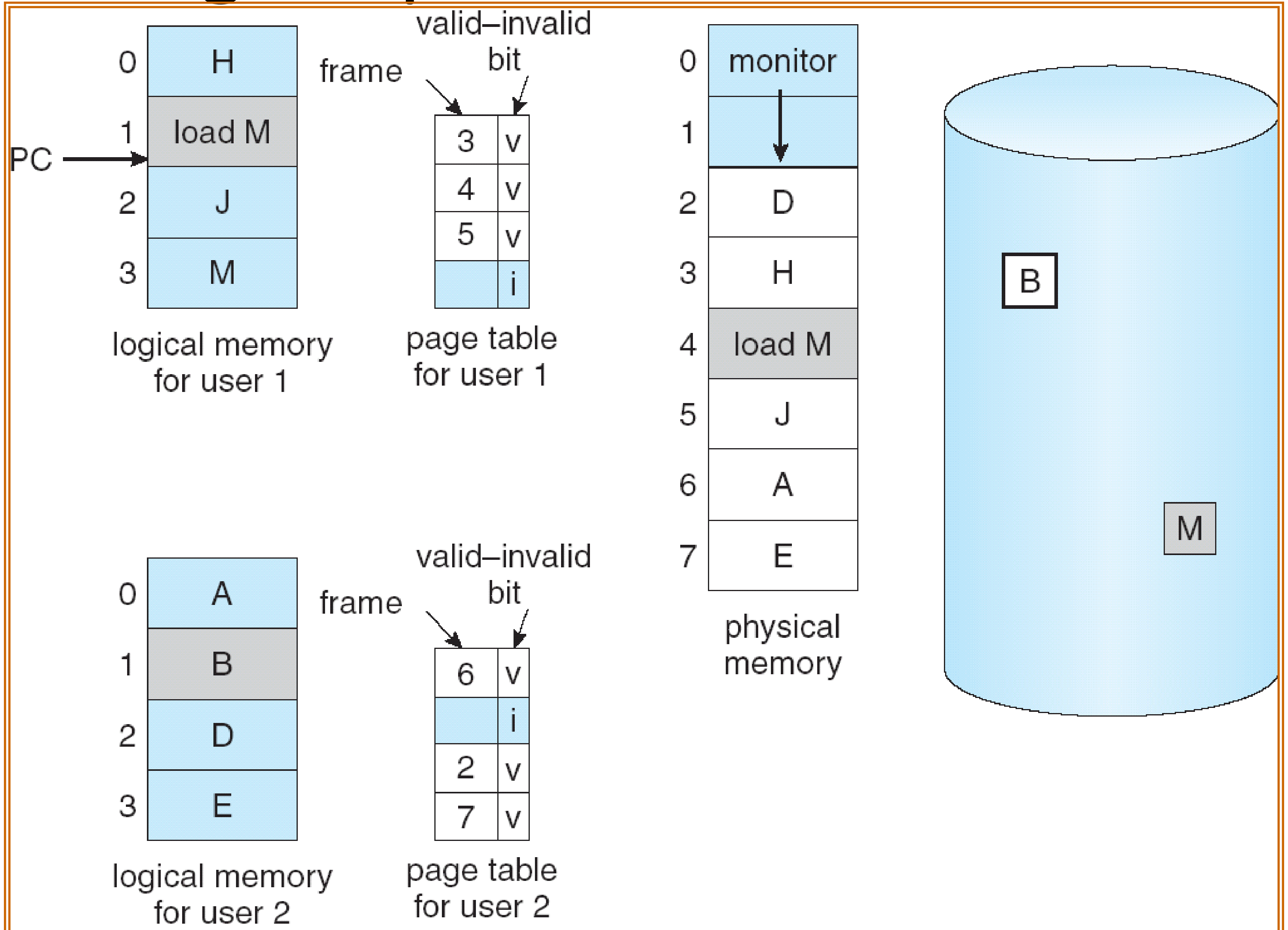
Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement

- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk

- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

Need For Page Replacement



Basic Page Replacement

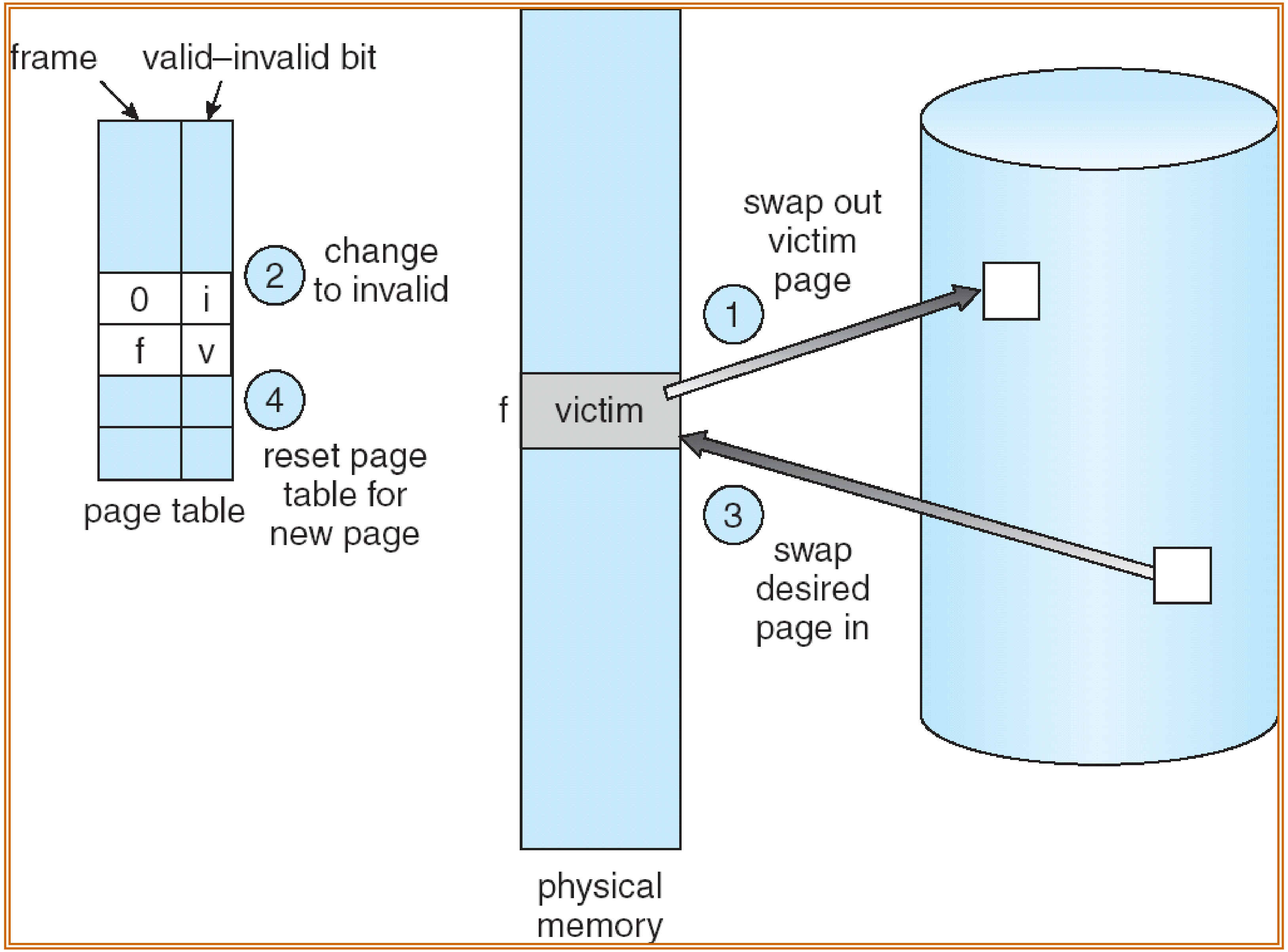
1. Find the location of the desired page on disk

2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim** frame

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Restart the process

Page Replacement



Page Replacement Algorithms

- Want lowest page-fault rate

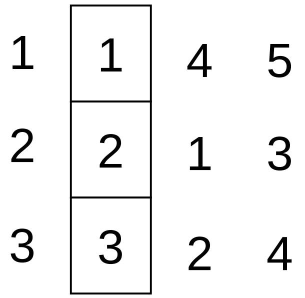
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string

- In all our examples, the reference string is

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

First-In-First-Out (FIFO) Algorithm

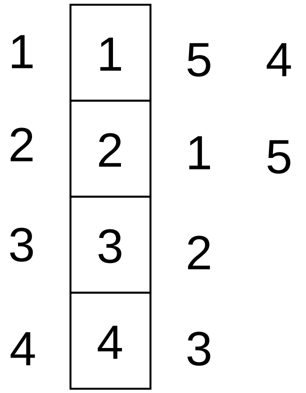
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)



1	1	4	5
2	2	1	3
3	3	2	4

9 page faults

- 4 frames



1	1	5	4
2	2	1	5
3	3	2	
4	4	3	

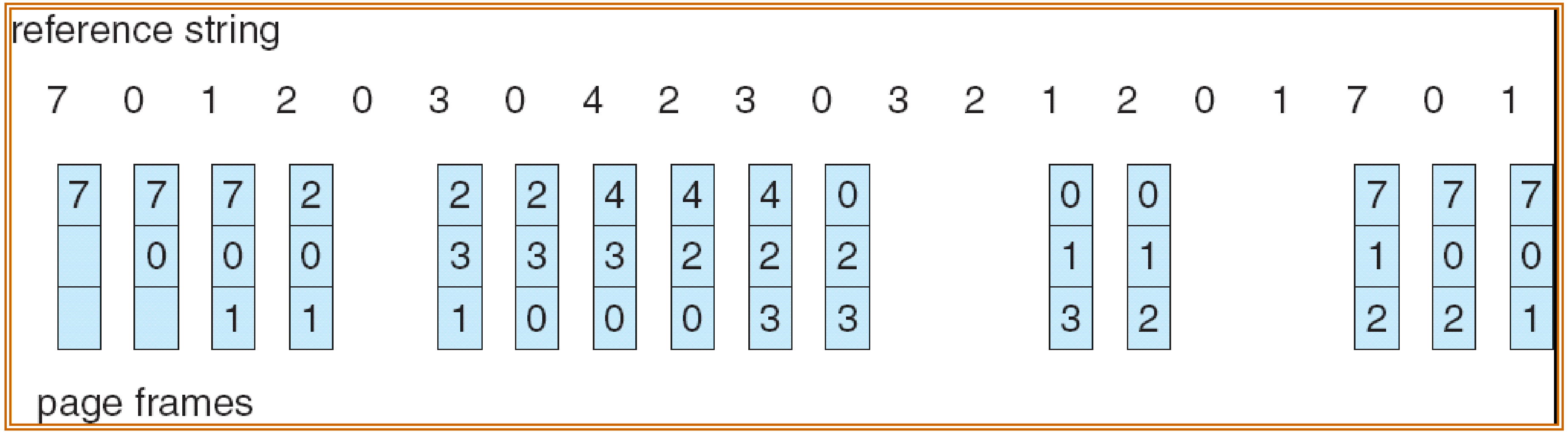
10 page faults

- Belady’s Anomaly: more frames ⇒ more page faults

FIFO Page Replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

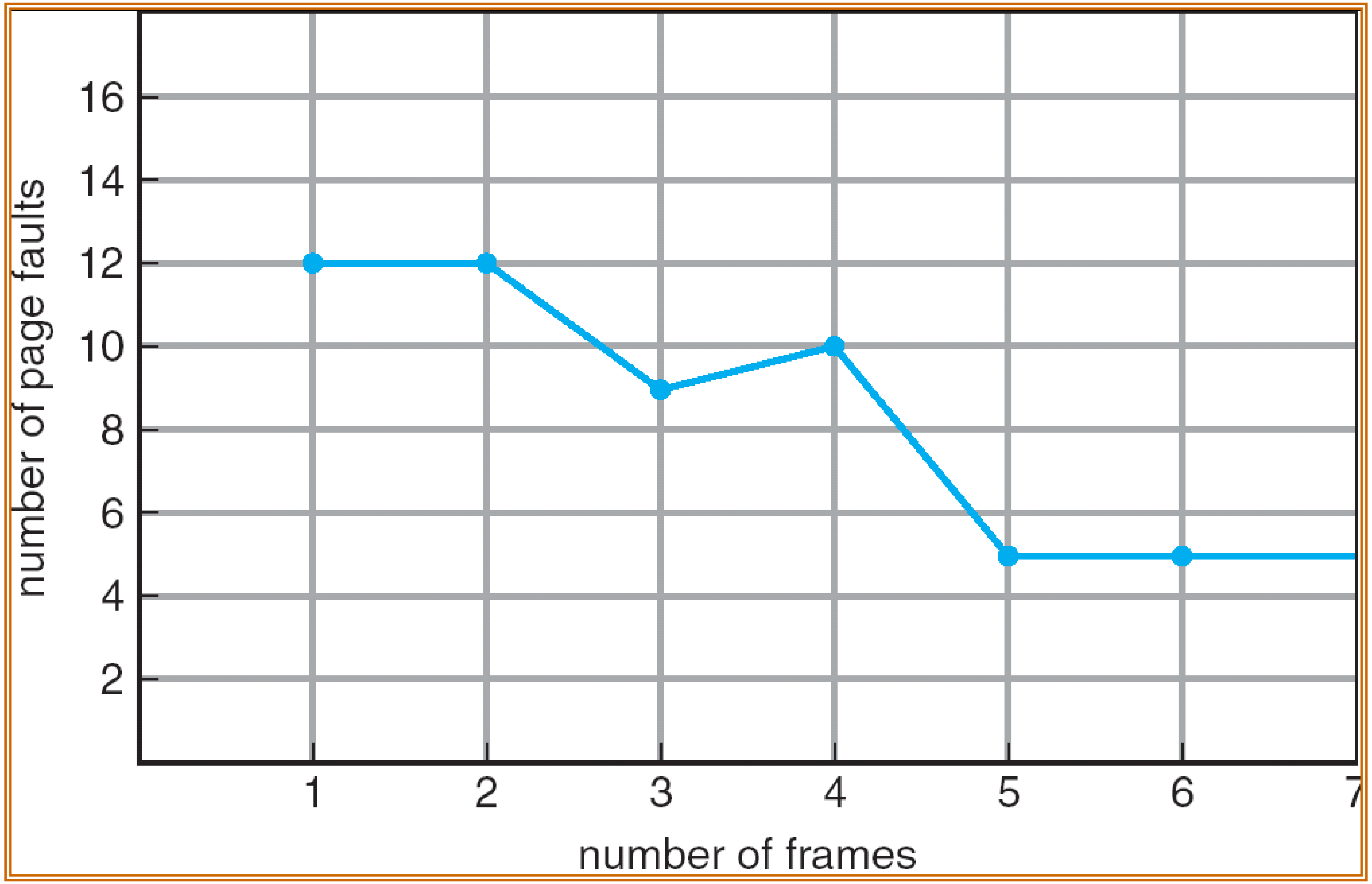


reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2	2	4	4	4	0			0	0			7	7	7
	0	0	0		3	3	3	2	2	2			1	1			1	0	0
		1	1		1	0	0	0	3	3			3	2			2	2	1

page frames

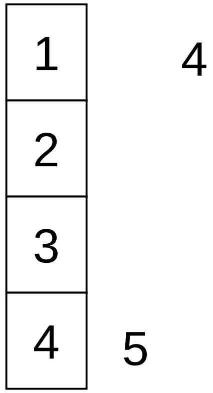
FIFO Illustrating Belady's Anomaly



Optimal Algorithm

- Replace page that will not be used for longest period of time
- 4 frames example

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



1 4

2

3

4 5

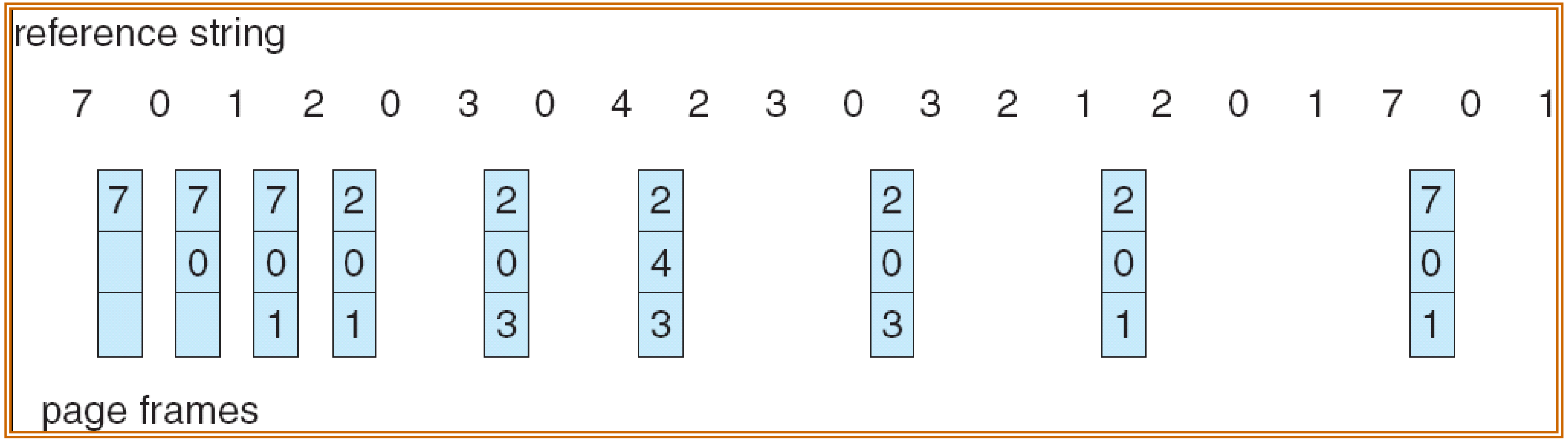
6 page faults

- How do you know this?
- Used for measuring how well your algorithm performs

Optimal Page Replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2	2	2	2	2	7
	0	0	0	0	4	0	0	0
		1	1	3	3	3	1	1

page frames

Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**

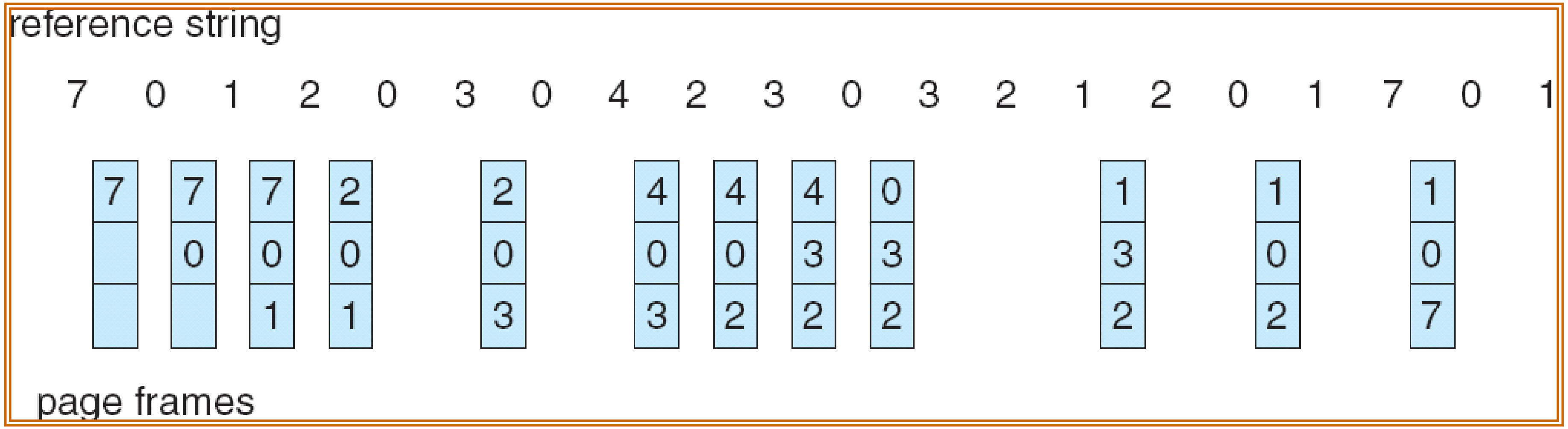
1	1	1	1	**5**
2	2	2	2	2
3	**5**	5	**4**	4
4	4	**3**	3	3

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to determine which are to change

LRU Page Replacement

reference string

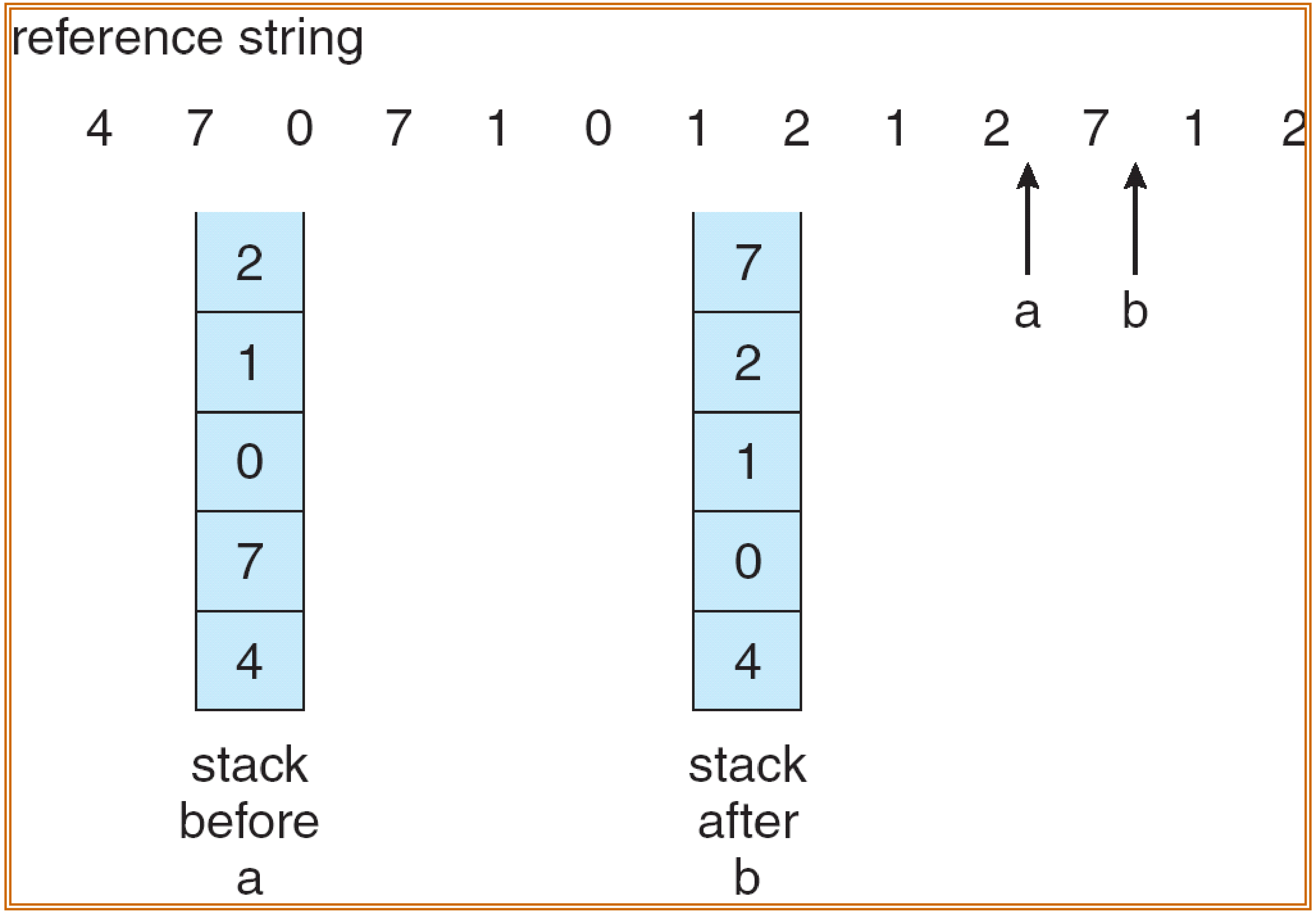
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



LRU Algorithm (Cont.)

- Stack implementation – keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement

Use Of A Stack to Record The Most Recent Page References



reference string

4 7 0 7 1 0 1 2 1 2 7 1 2

stack before a

stack after b

Allocation of Frames

- Each process needs *minimum* number of pages
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- Two major allocation schemes
  - fixed allocation
  - priority allocation

Fixed Allocation

- Equal allocation – For example, if there are 100 frames and 5 processes, give each process 20 frames.
- Proportional allocation – Allocate according to the size of process

  - $s_i = \text{size of process } p_i$
  - $S = \sum s_i$
  - $m = \text{total number of frames}$
  - $a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$

$$m = 64$$

$$s_i = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size
- If process P_i generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

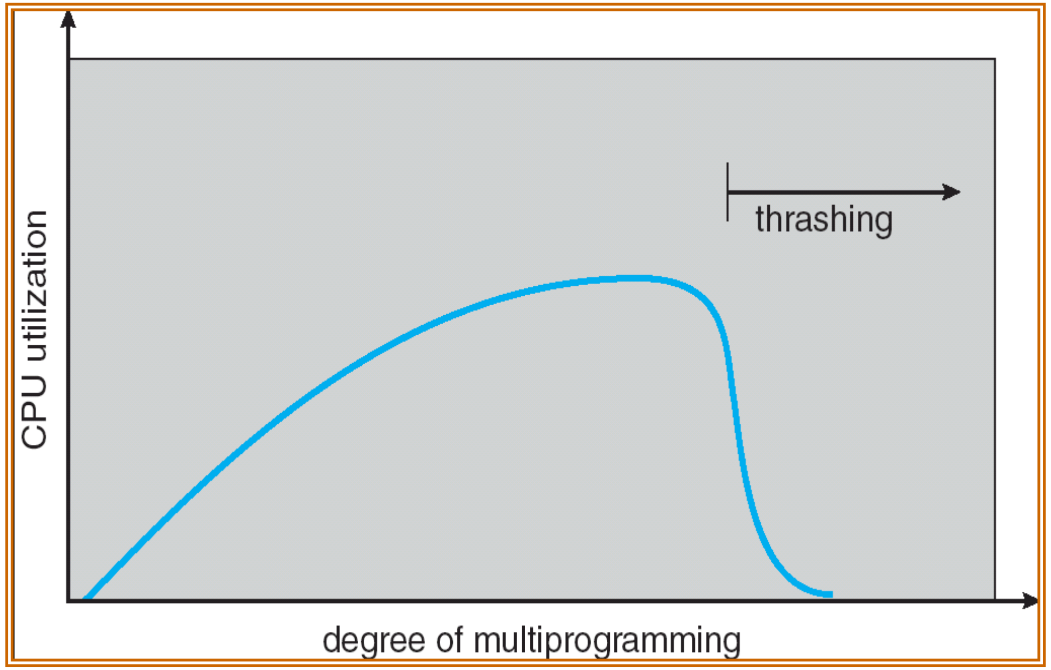
Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
- **Local replacement** – each process selects from only its own set of allocated frames

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
    - low CPU utilization
    - operating system thinks that it needs to increase the degree of multiprogramming
    - another process added to the system

- **Thrashing** ≡ a process is busy swapping pages in and out



Effect of Thrashing

Whenever thrashing starts, operating system tries to apply either **Global page replacement** Algorithm or **Local page replacement** algorithm.

Global Page Replacement

Since global page replacement can access to bring any page, it tries to bring more pages whenever thrashing found. But what actually will happen is, due to this, no process gets enough frames and by result thrashing will be increase more and more. So global page replacement algorithm is not suitable when thrashing happens.

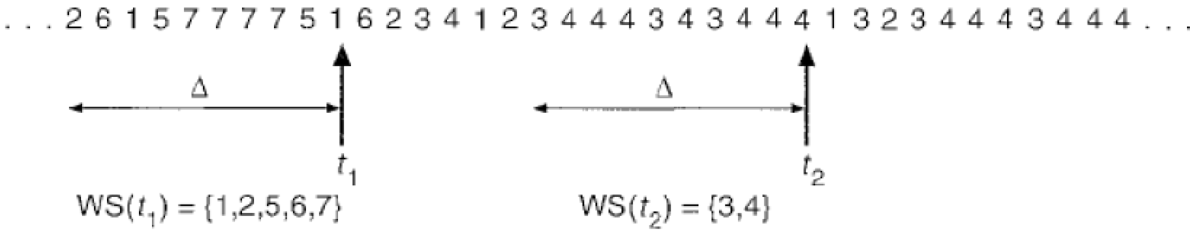
Local Page Replacement

Unlike global page replacement algorithm, local page replacement will select pages which only belongs to that process. So there is a chance to reduce the thrashing. But it is proven that there are many disadvantages if we use local page replacement. So local page replacement is just alternative than global page replacement in thrashing scenario.

Techniques to Handle Thrashing

1. Working Set Model

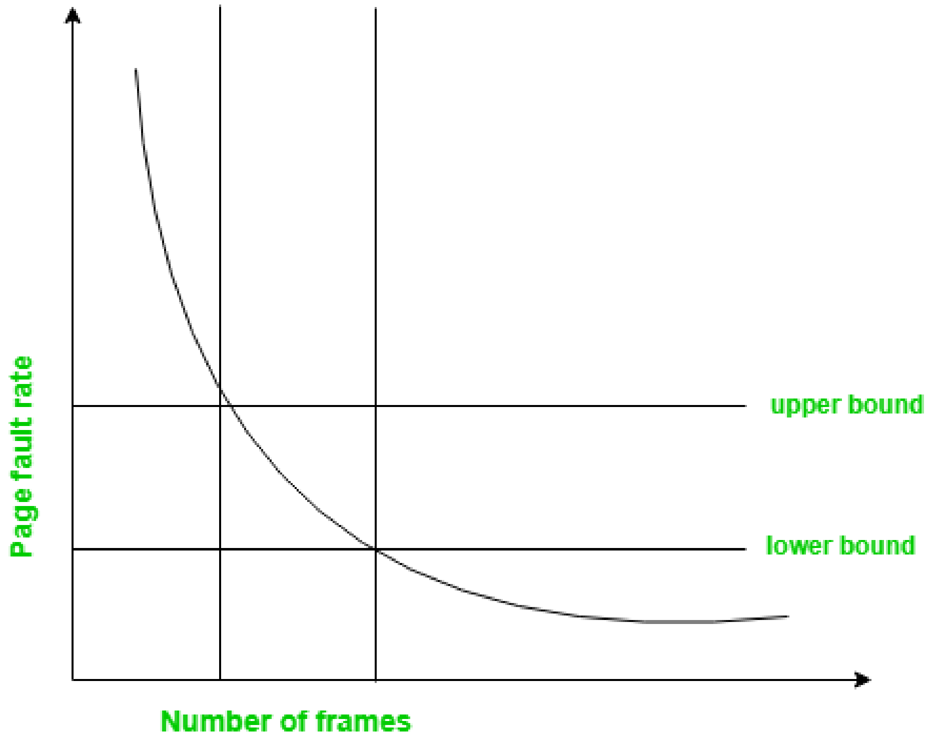
- Working set model is based on assumption of Locality. This model used a parameter Δ defines the working-set window.
- The concept is to checking the most recent Δ page references. Working set is a set of pages available in the most recent Δ or sometime also known as an approximation of the program's locality.
- If the page is in active use it will be in the working set. If it is no longer being used it will dropped from the working set Δ time units after its last reference. So working set is an approximation of program's locality.
- If Δ = 10 memory references, then the working set at time t1 is {1, 2, 5,6, 7}. At time t2, the working set has changed to {3, 4}.
- The accuracy of the working set depends on the selection of Δ. Because assume that Δ is too small, then it will not encompass the whole locality; but if Δ is too large, then it may overlap several localities.
- Main property of working set is its size. If we compute the working-set size, WSS for each process in the system, we can then consider that where D is the total demand for frames that each process is using the pages in it's working set. Thus process i needs WSS frames.
- If the total demand is greater than the total number of available frames (D > m), then it will cause thrashing to occur, because in this case some processes will not have enough frames. Once has been selected we can use this model easily.
- Here the role of operating system is that the operating system monitors the working set of each process and after that it allocates to that working set enough frames to provide it with its working-set size.
- If we have sufficient extra frames, then another process can be initiated. Here it is important to remember that if the sum of the **working-set sizes** increases and it is exceeding the total number of available frames then operating system selects a process to suspend.
- The process's pages are swapped out, and its frames are reallocated to other processes and the suspended process can be restarted later.



Working Set Model

2. Page Fault Frequency

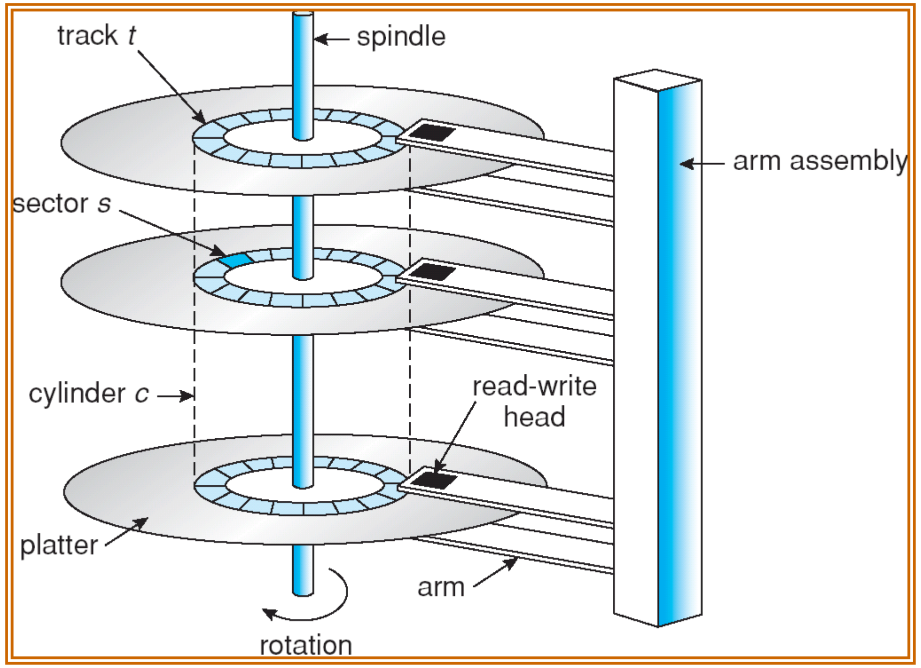
It is some direct approach than working set model. When thrashing occurring we know that it has few number of frames. And if it is not thrashing that means it has too many frames. Based on this property we assign an upper and lower bound for the desired page fault rate. According to page fault rate we allocate or remove pages. If the page fault rate become less than the lower limit, frames can be removed from the process. Similarly, if the page fault rate become more than the upper limit, more number of frames can be allocated to the process. And if no frames available due to high page fault rate, we will just suspend the processes and will restart them again when frames available.



Mass-Storage Systems

- Overview of Mass Storage Structure
- Disk Structure
- Disk Scheduling
- Disk Management

Moving-head Disk Mechanism



•Drives rotate at 60 to 200 times per second
•**Transfer rate** is rate at which data flow between drive and computer
•**Positioning time** (**random-access time**) is time to move disk arm to desired cylinder (**seek time**) and time for desired sector to rotate under the disk head (**rotational latency**)

Overview of Mass Storage Structure (Cont.)

- Magnetic tape
  - Was early secondary-storage medium
  - Relatively permanent and holds large quantities of data
  - Access time slow
  - Random access ~1000 times slower than disk
  - Mainly used for backup, storage of infrequently-used data, transfer medium between systems
  - Kept in spool and wound or rewound past read-write head
  - Once data under head, transfer rates comparable to disk
  - 20-200GB typical storage
  - Common technologies are 4mm, 8mm, 19mm, LTO-2 and SDLT

Disk Structure

- Disk drives are addressed as large 1-dimensional arrays of *logical blocks*, where the logical block is the smallest unit of transfer.

- The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially.
    - Sector 0 is the first sector of the first track on the outermost cylinder.
    - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

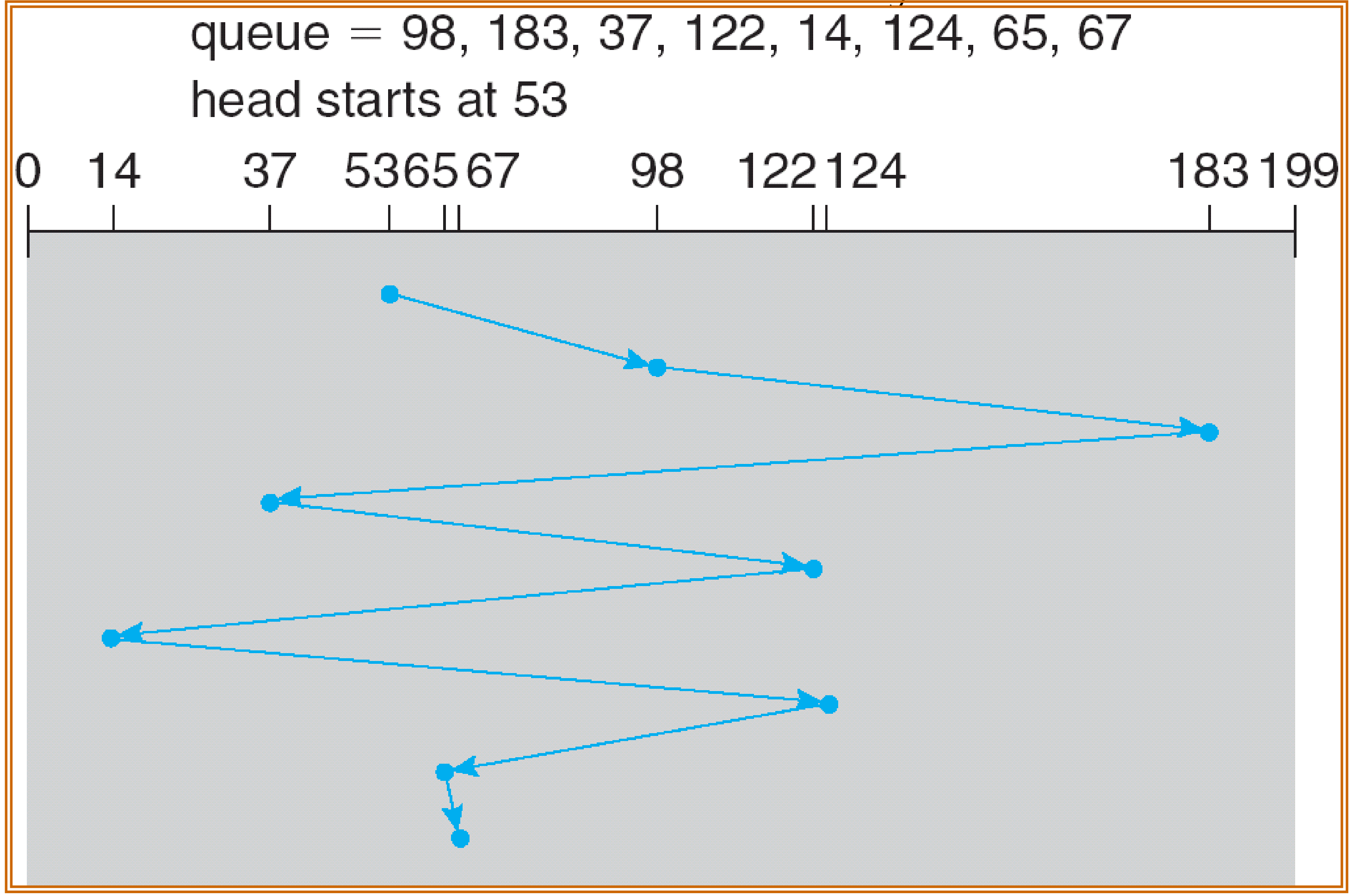
Disk Scheduling

- The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth.
- Access time has two major components
  - *Seek time* is the time for the disk are to move the heads to the cylinder containing the desired sector.
  - *Rotational latency* is the additional time waiting for the disk to rotate the desired sector to the disk head.
- Minimize seek time
- Seek time ≈ seek distance
- Disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

FCFS – First Come First Serve

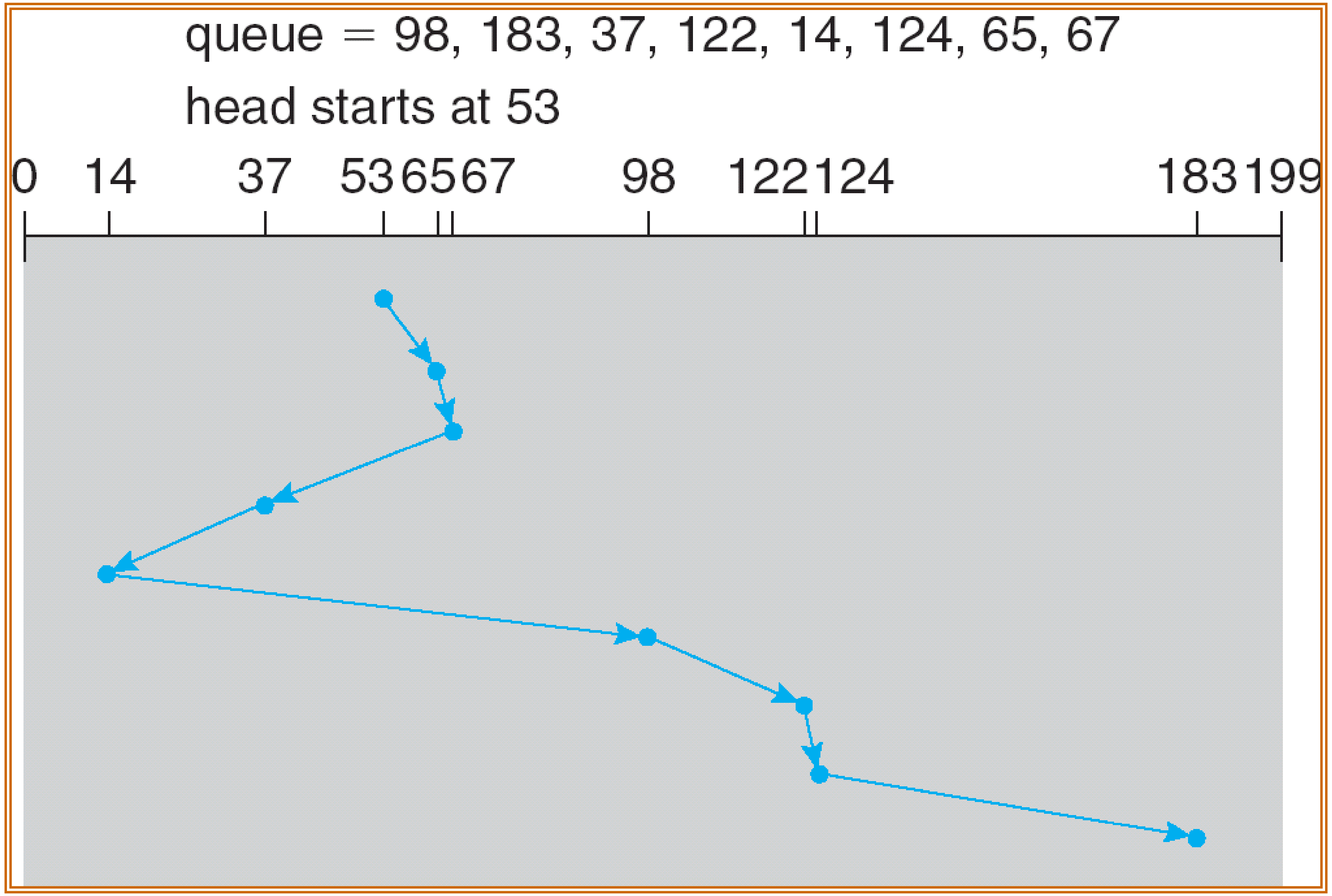
98, 183, 37, 122, 14, 124, 65, 67. Head pointer 53

Total head movement: 640 cylinders.



SSTF - Shortest Seek Time First

Total head movement: 236 cylinders.

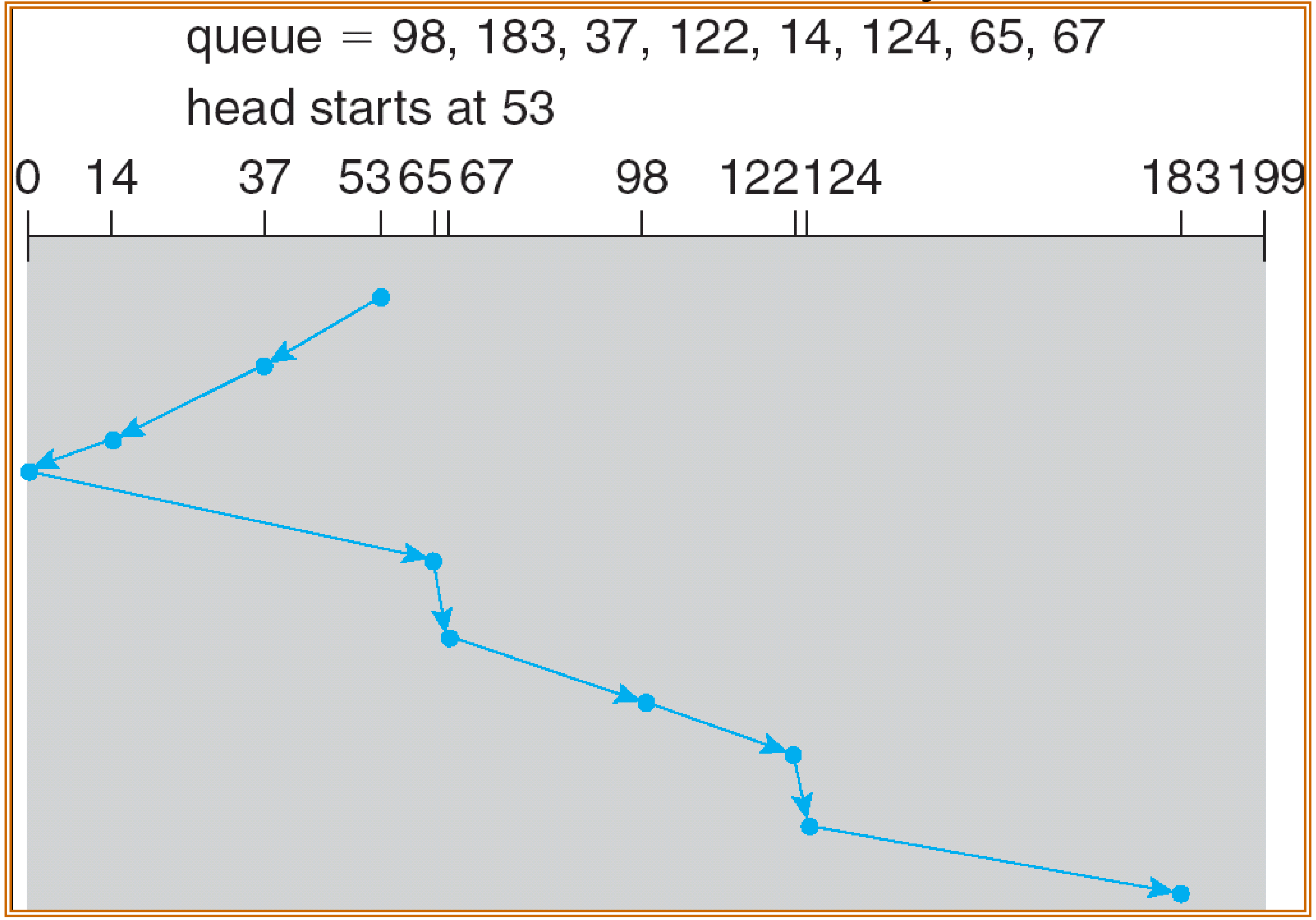


queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

SCAN

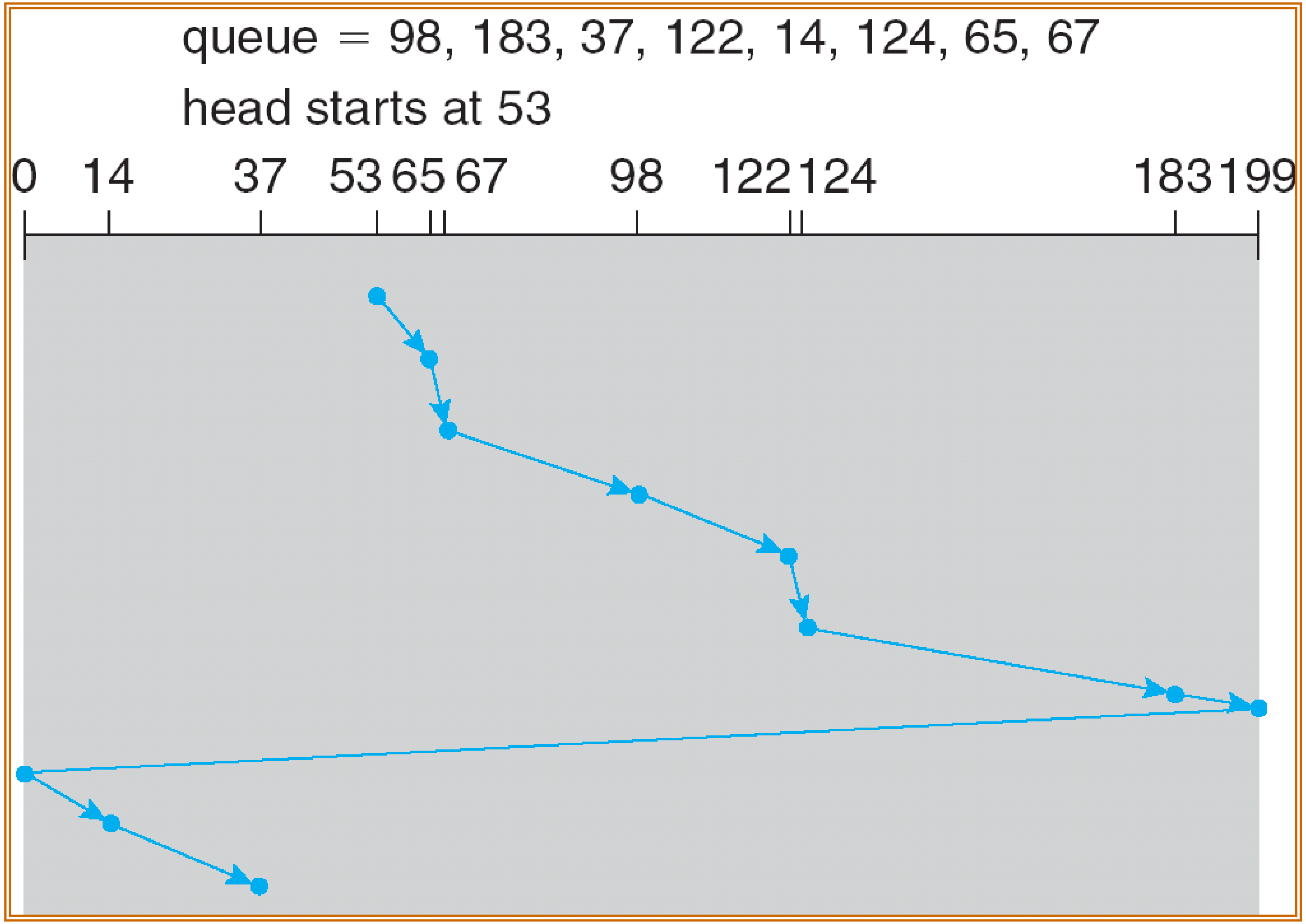
Total head movement: 208 cylinders



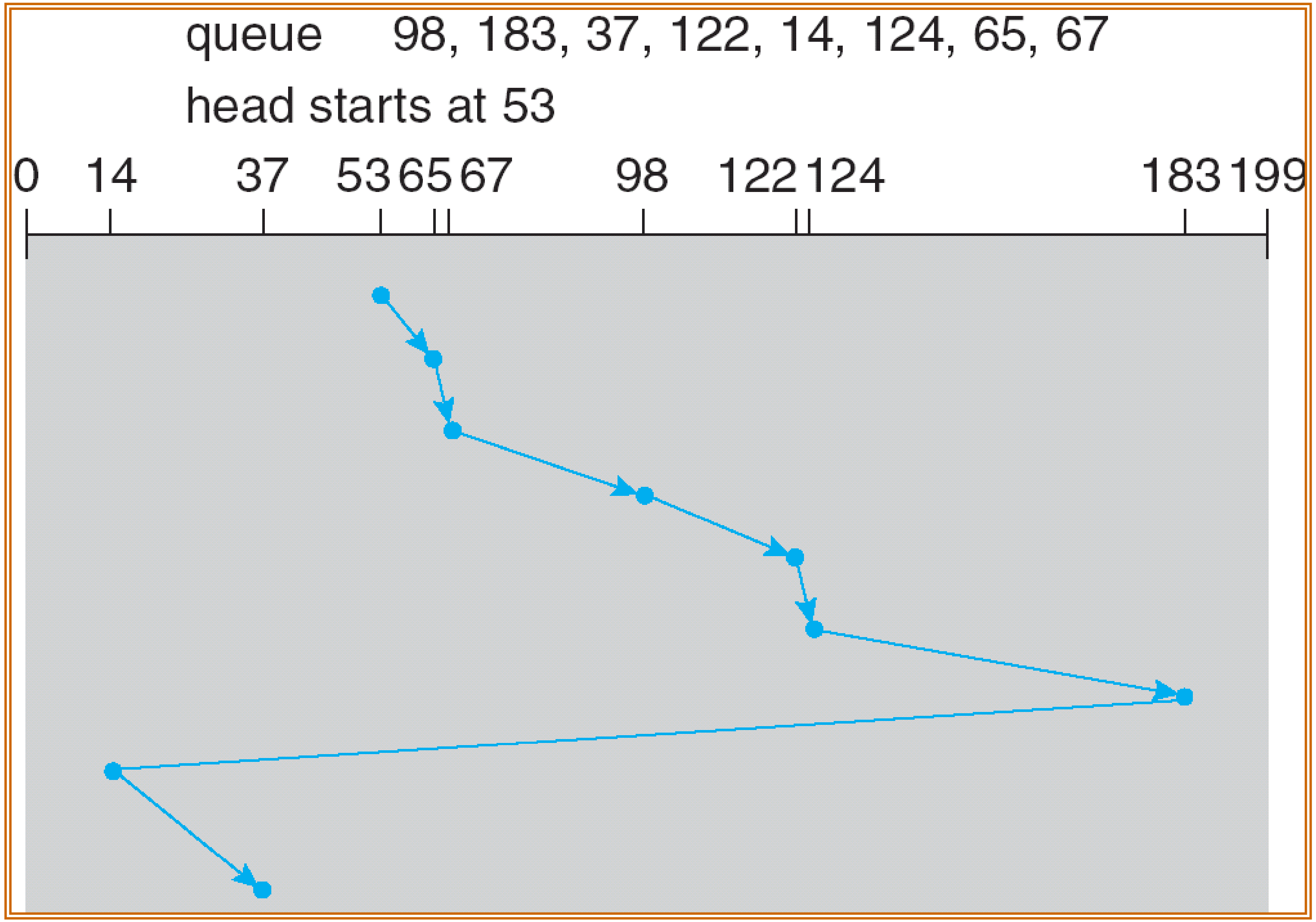
queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

C-SCAN


queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0
14
37
53
65
67
98
122
124
183
199

C-LOOK


queue 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0
14
37
53
65
67
98
122
124
183
199

Selecting a Disk-Scheduling Algorithm

- SSTF is common and has a natural appeal
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk.
- Performance depends on the number and types of requests.
- Requests for disk service can be influenced by the file-allocation method.
- The disk-scheduling algorithm should be written as a separate module of the operating system, allowing it to be replaced with a different algorithm if necessary.
- Either SSTF or LOOK is a reasonable choice for the default algorithm.

Swap-Space Management

- Swap-space — Virtual memory uses disk space as an extension of main memory.
- Swap-space
  - can be carved out of the normal file system
  - can be in a separate disk partition (more commonly)
- Swap-space management
  - 4.3BSD allocates swap space when process starts; holds *text segment* (the program) and *data segment.*
  - Kernel uses *swap maps* to track swap-space use.
  - Solaris 2 allocates swap space only when a page is forced out of physical memory, not when the virtual memory page is first created.

File Concept

- Contiguous logical address space

- Types:
  - Data
    - numeric
    - character
    - binary
  - Program

File Structure

- None - sequence of words, bytes
- Simple record structure
    - Lines
    - Fixed length
    - Variable length
- Complex Structures
    - Formatted document
    - Relocatable load file
- Can simulate last two with first method by inserting appropriate control characters
- Who decides:
    - Operating system
    - Program

File Attributes

- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk

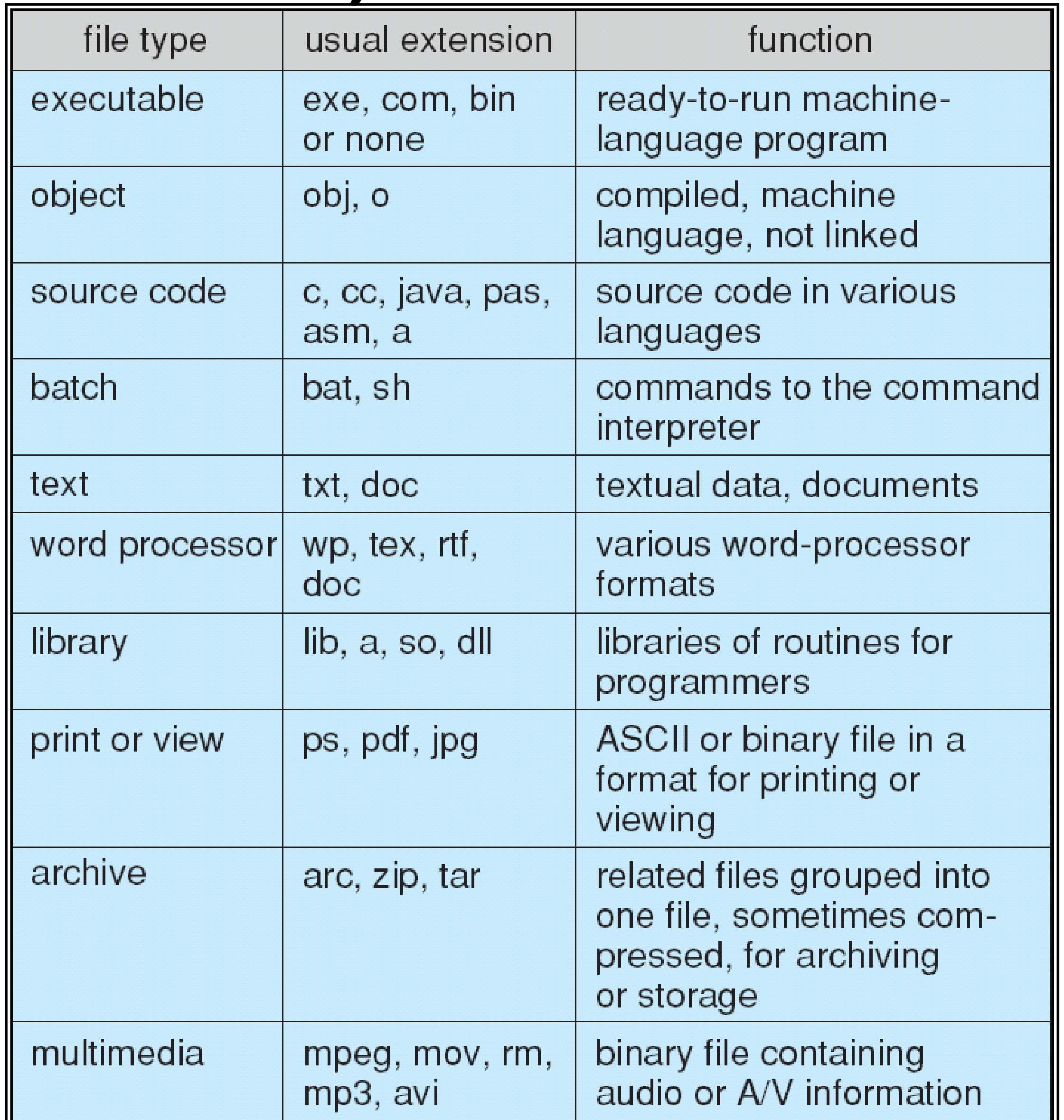
File Operations

- File is an **abstract data type**
- **Create**
- **Write**
- **Read**
- **Reposition within file**
- **Delete**
- **Truncate**
- *Open(F_i)* – search the directory structure on disk for entry F_i, and move the content of entry to memory
- *Close (F_i)* – move the content of entry F_i in memory to directory structure on disk

Open Files

- Several pieces of data are needed to manage open files:
  - File pointer: pointer to last read/write location, per process that has the file open
  - File-open count: counter of number of times a file is open – to allow removal of data from open-file table when last processes closes it
  - Disk location of the file: cache of data access information
  - Access rights: per-process access mode information

File Types – Name, Extension



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

Access Methods

- **Sequential Access**

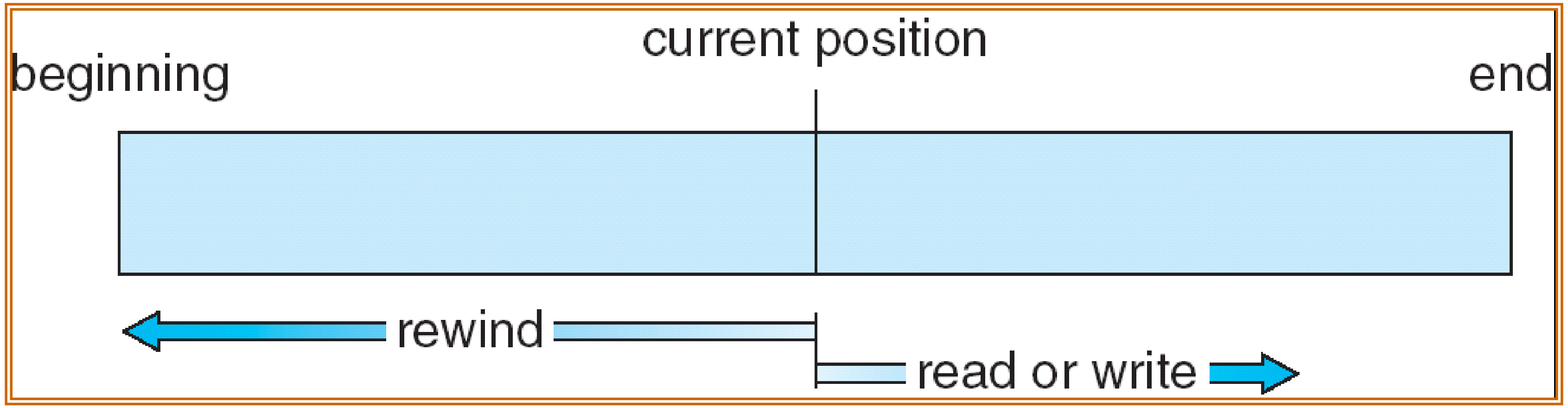
  read next
  write next
  reset
  no read after last write
  (rewrite)

- **Direct Access**

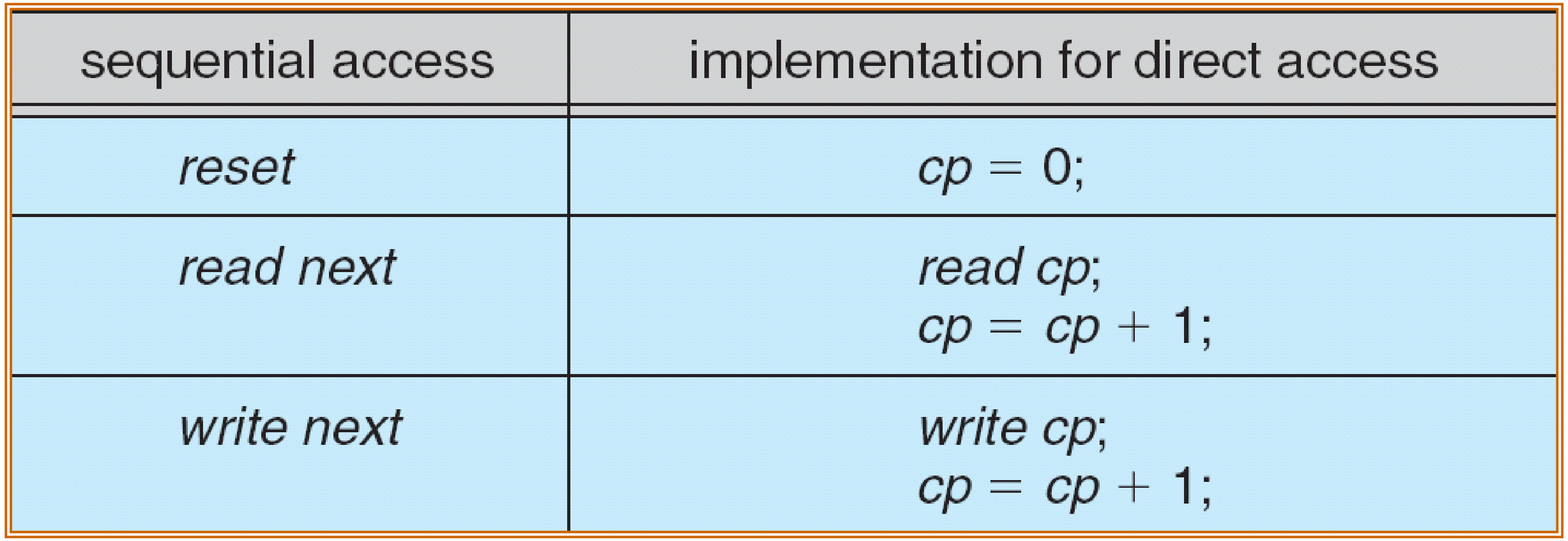
  read *n*
  write *n*
  position to *n*
  read next
  write next
  rewrite *n*

n = relative block number

Sequential-access File

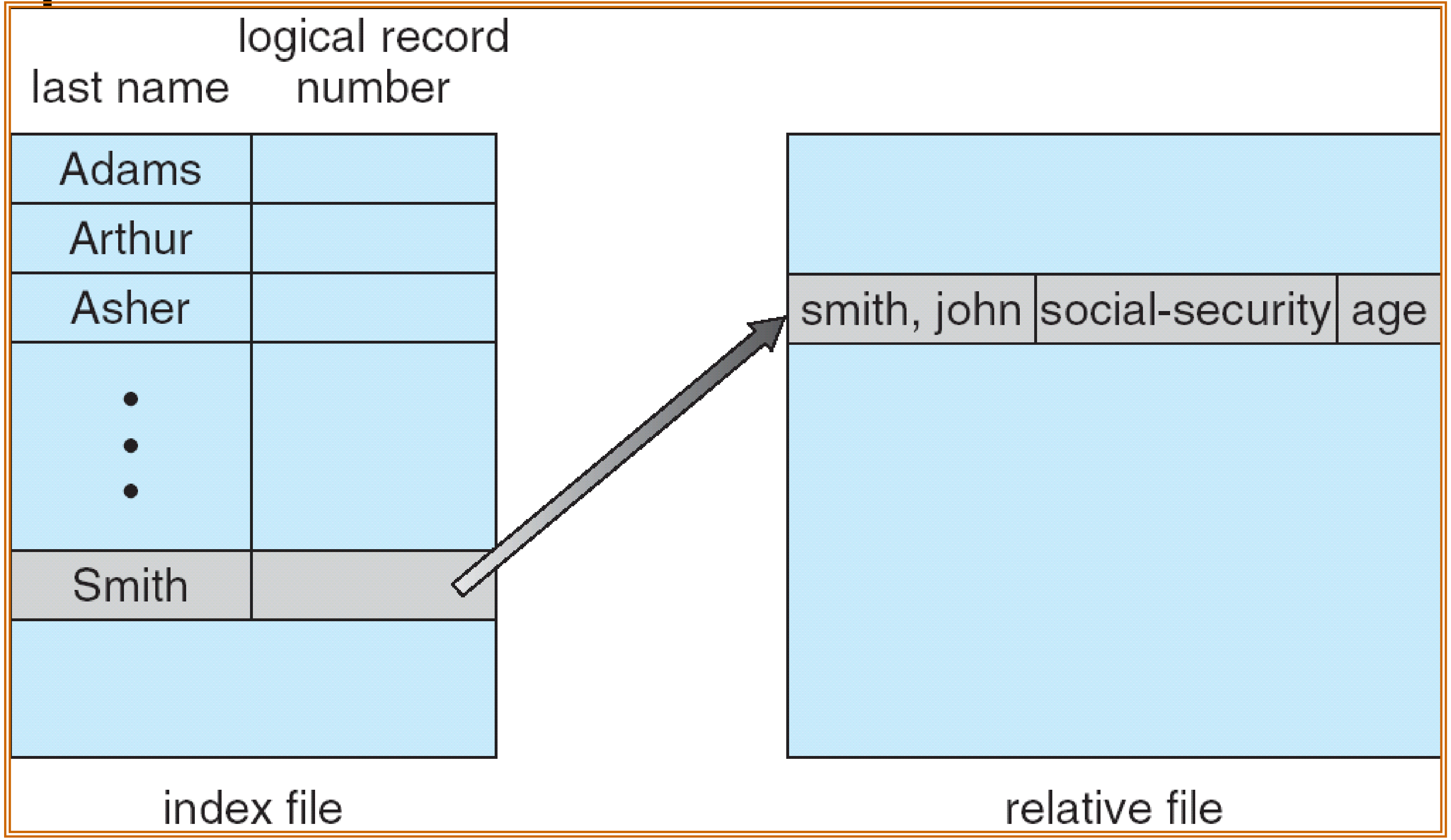


Simulation of Sequential Access on a Direct-access File



sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

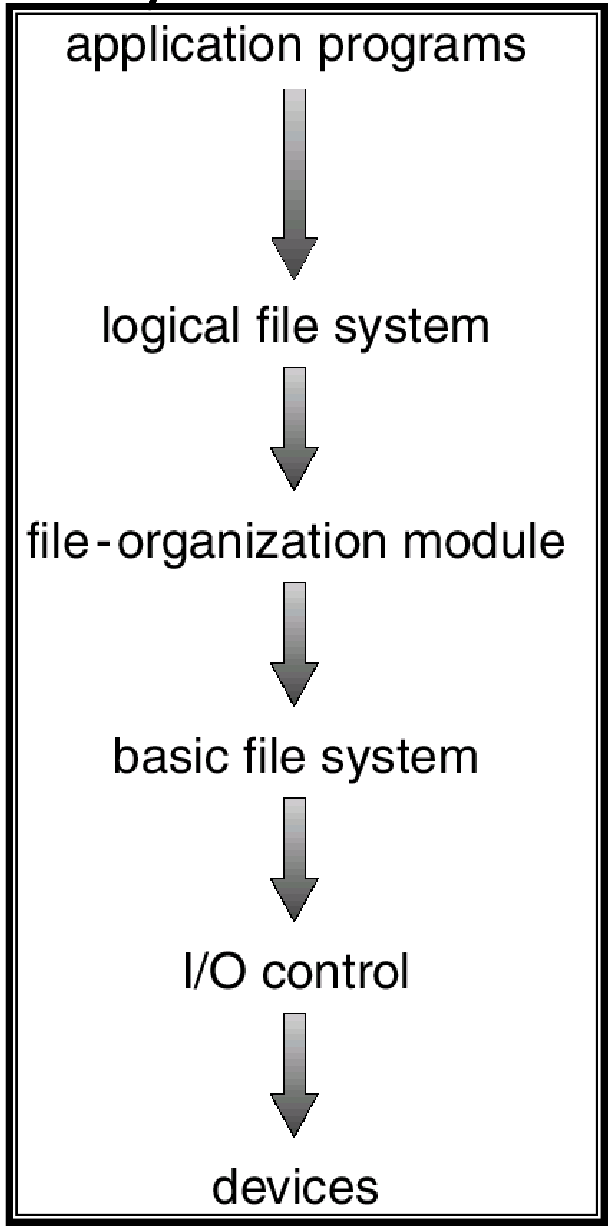
Example of Index and Relative Files



File-System Structure

- File structure
  - Logical storage unit
  - Collection of related information
- File system resides on secondary storage (disks).
- **Design problem:**
  - **How should the file system look to the user … attributes, operations, directory structure**
  - **Creating algorithms and data structures to map the logical file system to physical secondary storage device.**
- File system organized into layers.
- *File control block* ***(FCB)*** – storage structure consisting of information about a file **– stored on disk, copied to memory for use**.

Layered File System



Manages file attribute/control info
Manages directory structure:
symbolic file names to logical block#,
Uses File Control Block (FCB)

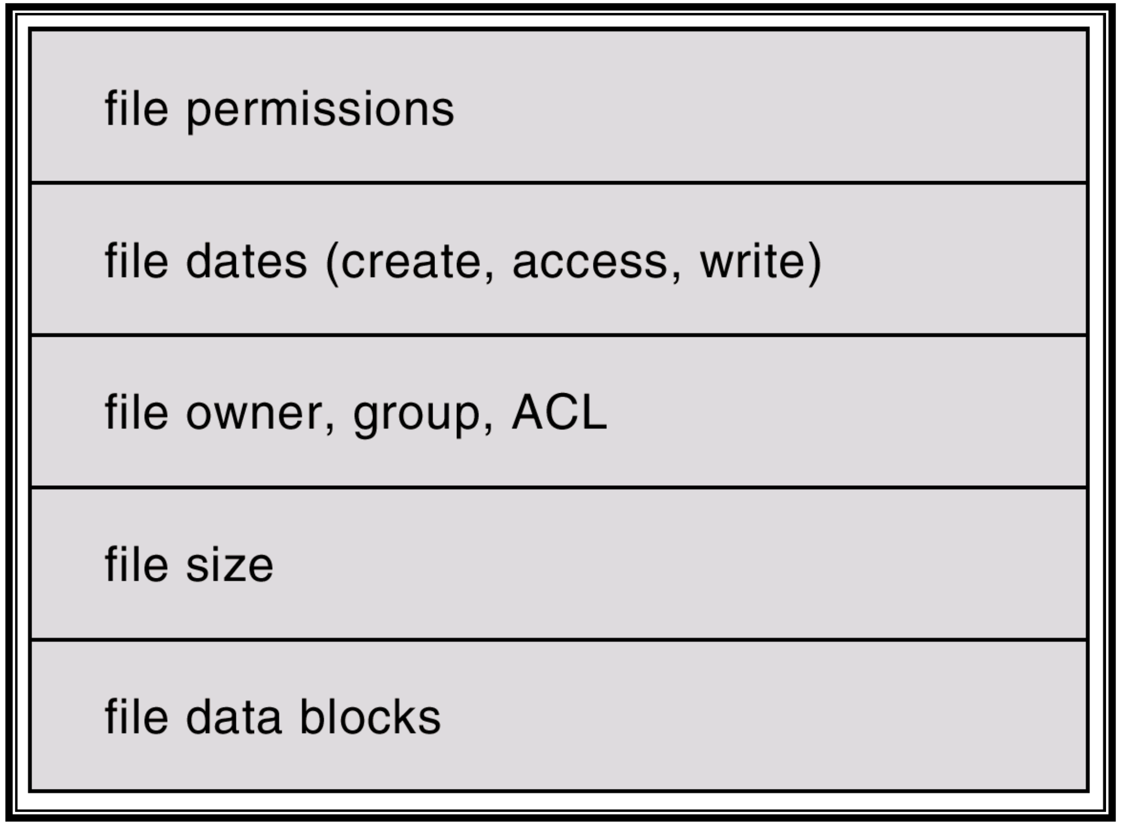
Logical block# in → physical block# out,
(knows location of file and allocation scheme used)

Generic cmds to dev drivers using disk addresses:
cylinder, track, sector

Device drivers: interfaces to hardware/controllers,
gets generic commands.

Hardware controllers

A Typical File Control Block



file permissions

file dates (create, access, write)

file owner, group, ACL

file size

file data blocks

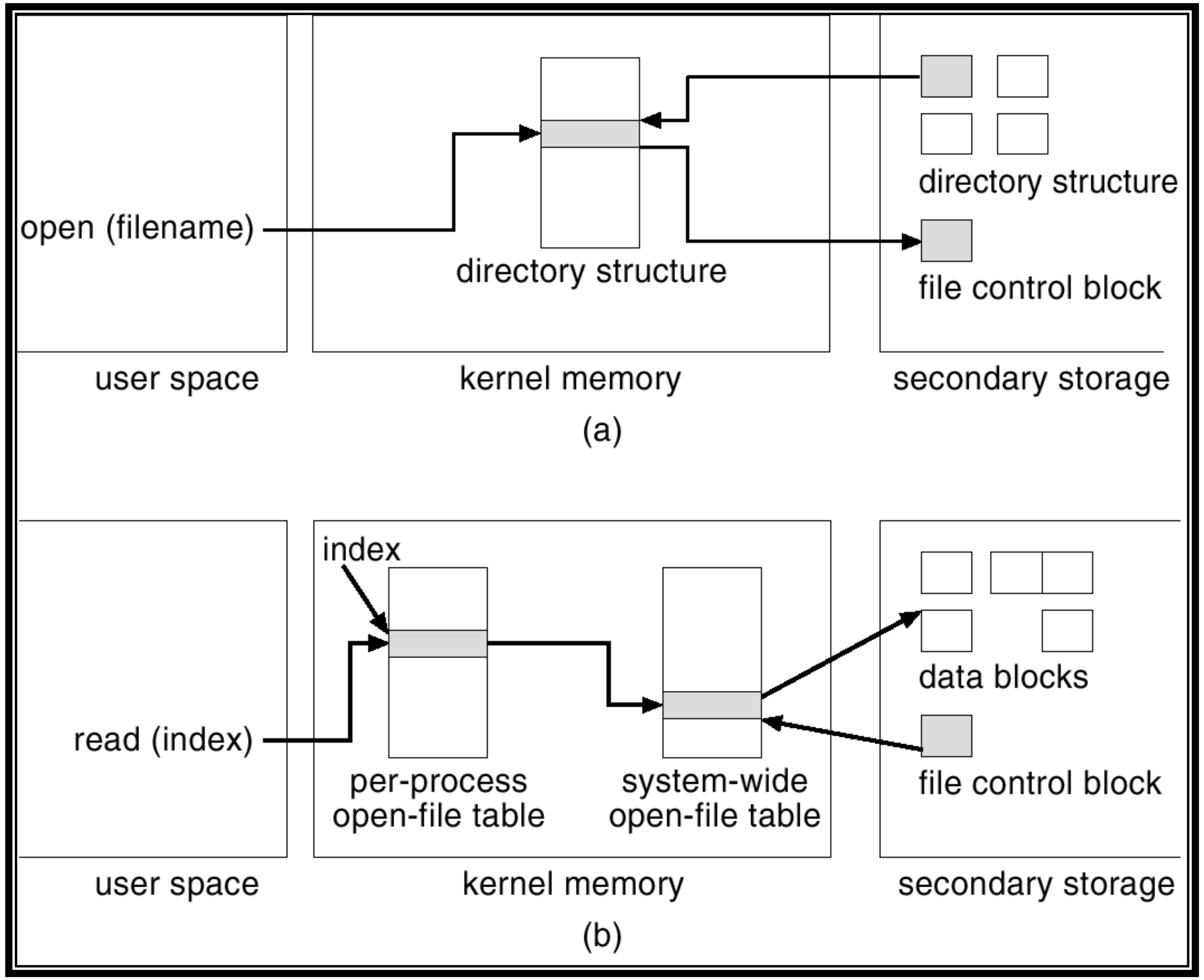
←Location on disk

Overview

- On-disk structures
  - Boot control block – 1st block of partition - used for booting if this is a bootable partition. May also have information on another partition which may be where booting may start (the Master Boot Record, MBR).
  - Partition control block- attributes of partition, FCB pointers, number blocks in partition, free block count, free block pointers, etc.
  - Directory structure
  - The FCB’s - (inode in UNIX)
- In-Memory structures
  - In memory copy of partition table
  - Cache of recently accesses directories
  - System-wide open file table copy of FCBs for all open files +more
  - Per-process open file table (pointers to system-wide open file table) – accessed via file descriptor returned from open call – reduces search time

In-Memory File System Structures

The following example illustrates the necessary file system structures provided by the operating systems



opening a file

reading a file

Scenario for file operations

- **Creating a file**
  - **Allocate a new FCB**
  - **Read appropriate directory into memory, and add the new file name and FCB to it**
  - **Write it back to disk**
- **Opening a file**
  - **Directory structure searched – parts cached un memory**
  - **FCB copied to System-wide-open file table**
  - **File descriptor returned which indexes into the per-process-open file table.**
- **Reading a file**
  - **Using file descriptor, access the per processes open file table**
  - **Which in turn gets the FCB from system wide open file table**
  - **FCB has information needed to access the data in the file**

Directory Implementation

- **Must search director for a required file**

- Linear list of file names with pointer to the data blocks.
  - simple to program
  - time-consuming to execute

- Hash Table – linear list with hash data structure.
  - decreases directory search time
  - *collisions* – situations where two file names hash to the same location
  - fixed size

File system allocation

- **Many files of variable size stored on the same disk**

- **Main problem is how to allocate disk blocks for these files so that disk space is utilized effectively and files can be accessed quickly.**

- **Three major allocations methods:**
  - **contiguous**
  - **Linked**
  - **indexed**

Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk.

- **Minimal head movement – high performance**
  (minimal head movement) - used by IBM VM system
- Simple – only starting location (block #) and length (number of blocks) are required.

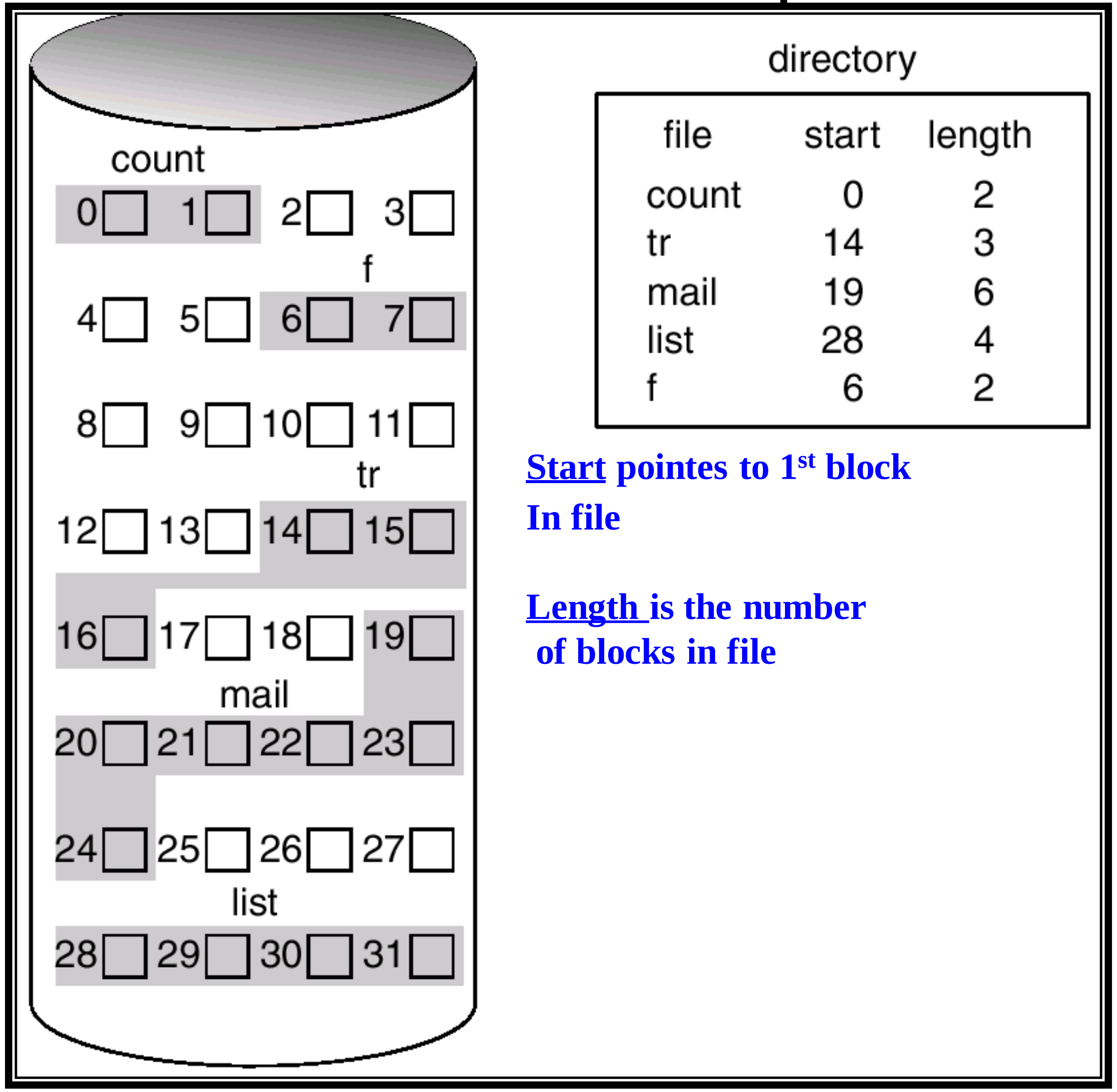
- Random access **– for file stating at block b, a block in that file at i blocks into the file would be at block b+i**

- **Good for sequential access also**
- Wasteful of space**– external fragmentation – same problem as in dynamic contiguous memory management (ch 9) – with same solutions.**

- Files cannot grow **– if we over-allocate then internal fragmentation**.
- **Difficult to estimate how much space is needed for file**

- **Modified contiguous allocation – give "extents" to make file grow – just postpones the original problem**

Contiguous Allocation of Disk Space



directory

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

Start pointes to 1st block
In file

Length is the number
of blocks in file

Extent-Based Systems

- Many newer file systems (I.e. Veritas File System) use a **modified contiguous allocation** scheme.

- Extent-based file systems allocate disk blocks in **extents**.

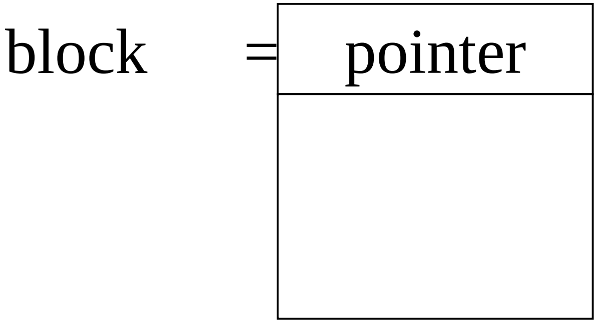
- An **extent** is a contiguous block of disks. Extents are allocated for file allocation. A file consists of one or more extents.
- **–solves growth problem, but just postpones the original problem- what should be the size of the extent: too large – internal fragmentation, too small: external fragmentation.**

Linked Allocation

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.

- **Pointer is to next block – file is treated like a linked list**

- ***Sequentially*** **follow pointers to access a given block**



Linked Allocation (Cont.)

- Simple – need only starting address
- Free-space management system – no waste of space
- **No random access**

File-allocation table (FAT) – disk-space allocation used by MS-DOS and OS/2. **– Combines linked with indexed**

Linked Allocation (Cont.)

- **Solves external, internal (minimal) fragmentation problem, and growth problem – only limit is free space pool**
  (as paging did in memory management, but no random access)

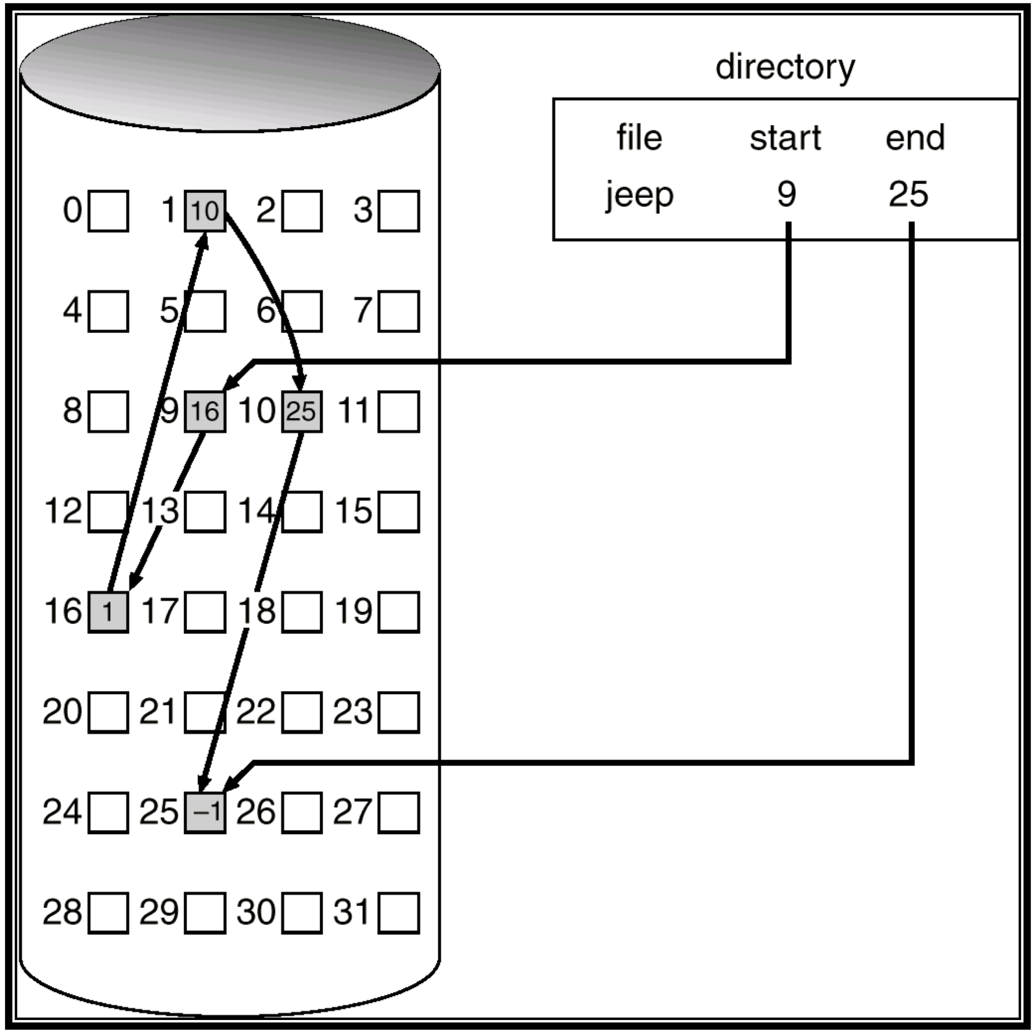
- **Only sequential access –direct access support poor (must be “simulated”)**

- **Access is slow and variable because of having to read all pointers in blocks before the one desired.**

- **Each block has wasted space for pointer – minimized by clustering blocks – but this may result in increased internal fragmentation**

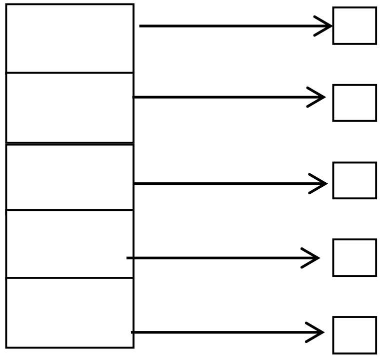
- **Reliability a problem: lost or damaged pointers**

Linked Allocation



Indexed Allocation

- Brings all pointers together into the *index block.*
- **One “logical” index block per file … in general may have multiple linked index blocks – see later**.
- Logical view.



index table

Solves internal, external fragmentation, and growth problem

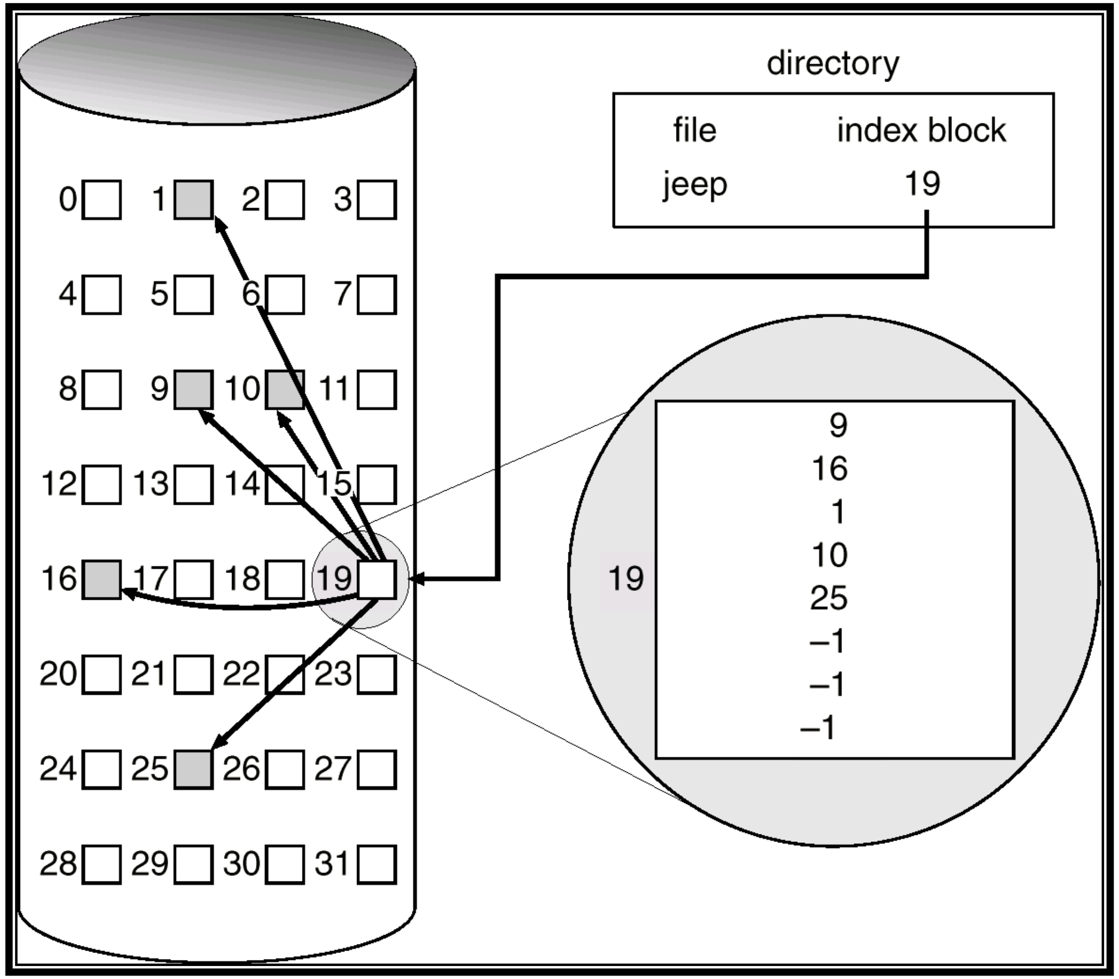
➔ In addition allows direct/random access

In simplest form, size of index block determines number of blocks in file

If index block too large, then wasted space for small files.

If index block not large enough then need to link index blocks together

Example of Indexed Allocation



Indexed Allocation (Cont.)

- Need index table **(ie., block)**
- Random access
- Dynamic access without external fragmentation, but have overhead of index block.
- **Index block permanently stored on disk, but may be cached in memory – to further improve disk performce. If cached, we have the old problem of keeping the disk updated – especially if we have a crash.**

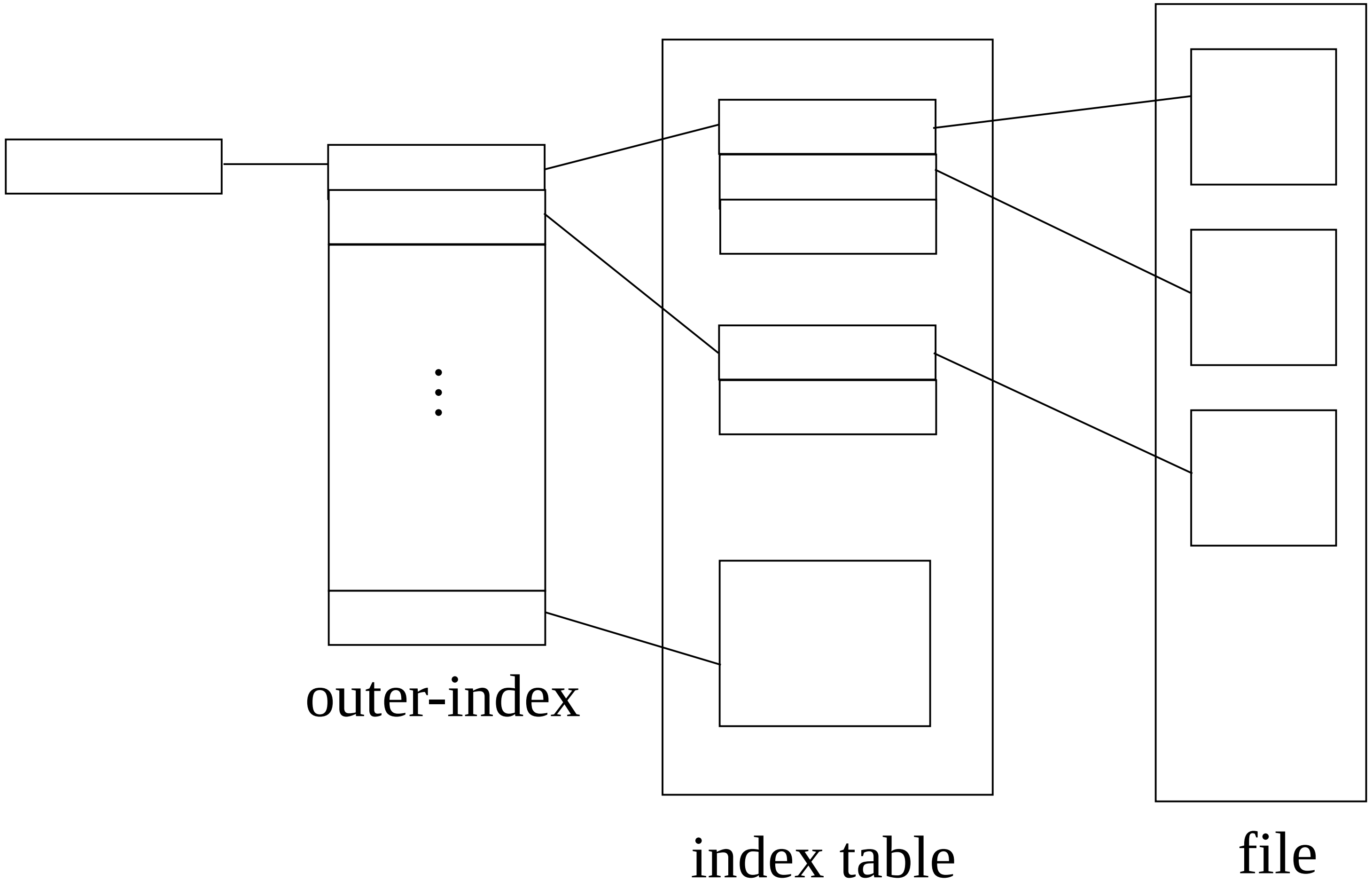
Indexed Allocation – Mapping (Cont.)

- Problem is when the file so large that there are not enough entries in a single index block.
- Want to map from logical to physical in a file of unbounded length
- Use a scheme in which the index blocks are themselves linked: Link blocks of index table (no limit on size).

- The last entry of a linked block of an index table is a pointer to the next index table
- Now no logical limit on the size of a file
- **Multilevel index** block is another approach to this problem – see below:
- Can combine linked index block idea with multi-level: UNIX
  I-node scheme –see below

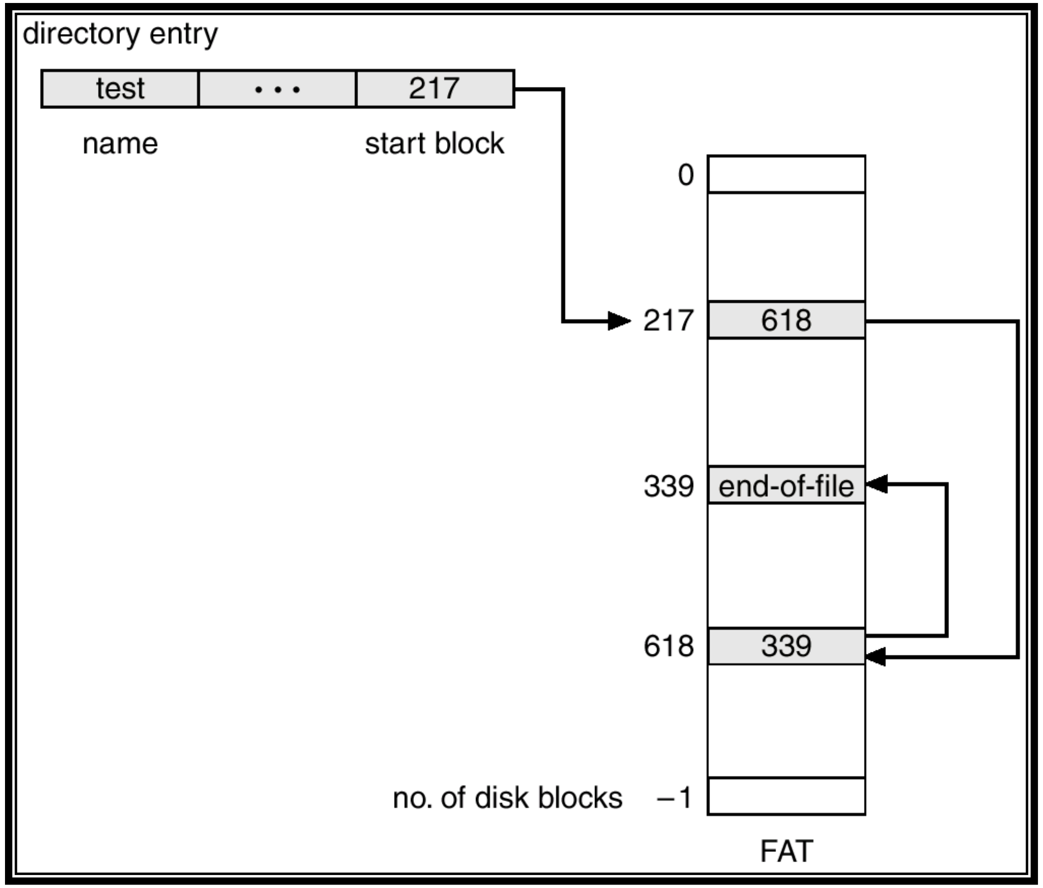
Indexed Allocation – Mapping (Cont.)

Multi-level Index blocks - See text



File-Allocation Table

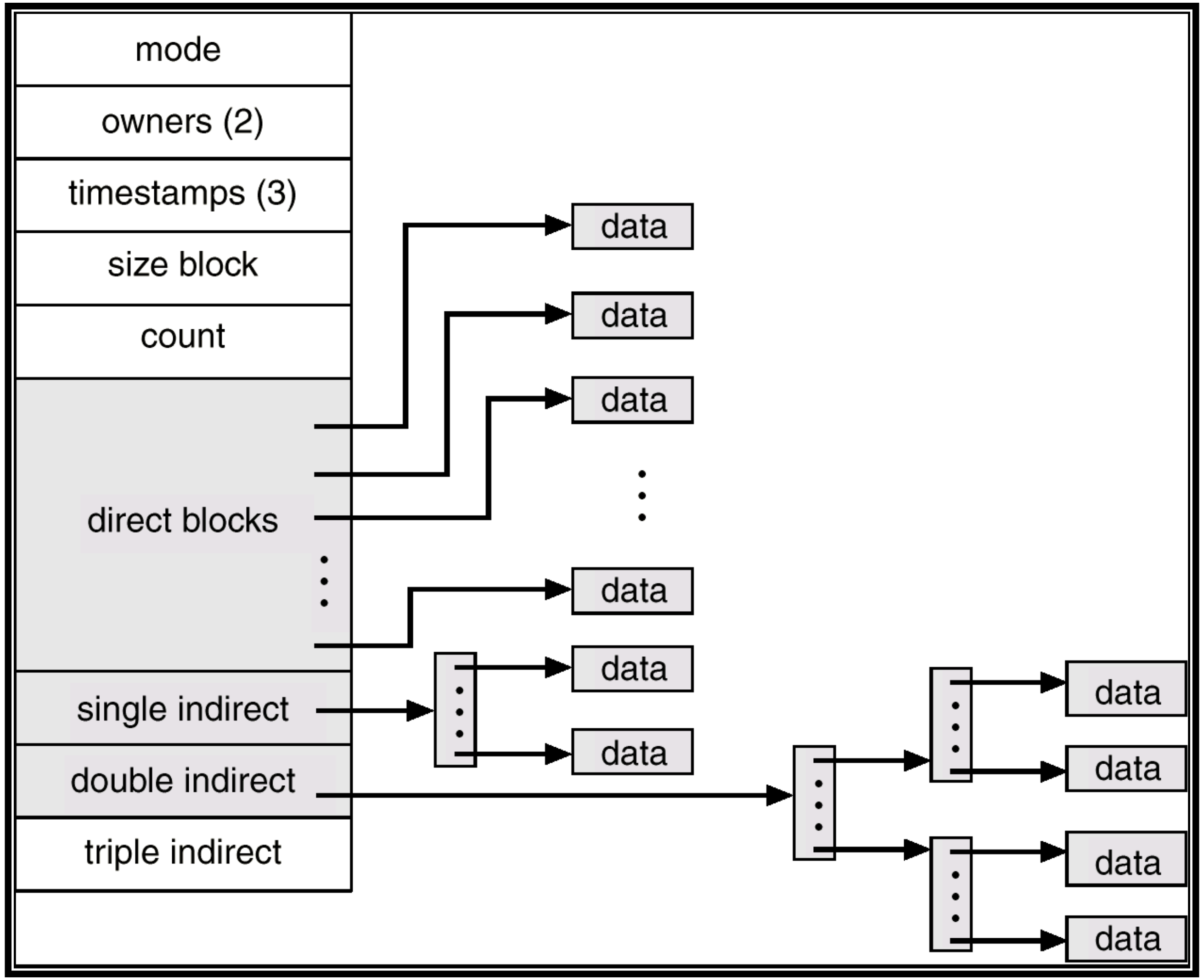
- Used for windows/OS/2 PC systems
- **Combines linked and indexed concepts**



Combined Scheme: UNIX (4K bytes per block)

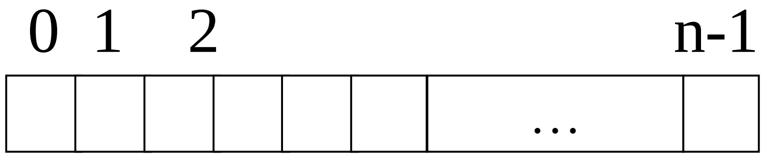
The UNIX Inode

Performance tradeoff favors small files



Free-Space Management

- Bit vector (*n* blocks)



$$\text{bit}[i] = \begin{array}{l} 1 \Rightarrow \text{block}[i] \text{ free} \\ \\ 0 \Rightarrow \text{block}[i] \text{ occupied} \end{array}$$

A bit for every block on disk – offset into map is block number

Block number calculation **(moving from left to right):**

(number of bits per word) *
(number of 0-value words) +
offset of first 1 bit **in 1st nonzero word**
Note:
“number of 0-value words” = # of leading all zero words

Free-Space Management (Cont.)

- Bit map requires extra space. Example:

  block size = 2^{12} bytes - 4K page (common)

  disk size = 2^{30} bytes (1 gigabyte) - on the small side

  $n = 2^{30}/2^{12} = 2^{18}$ bits (or 32K bytes)
  a 1.3 GB with 512byte page has bit map of 332KB - stresses memory if cached

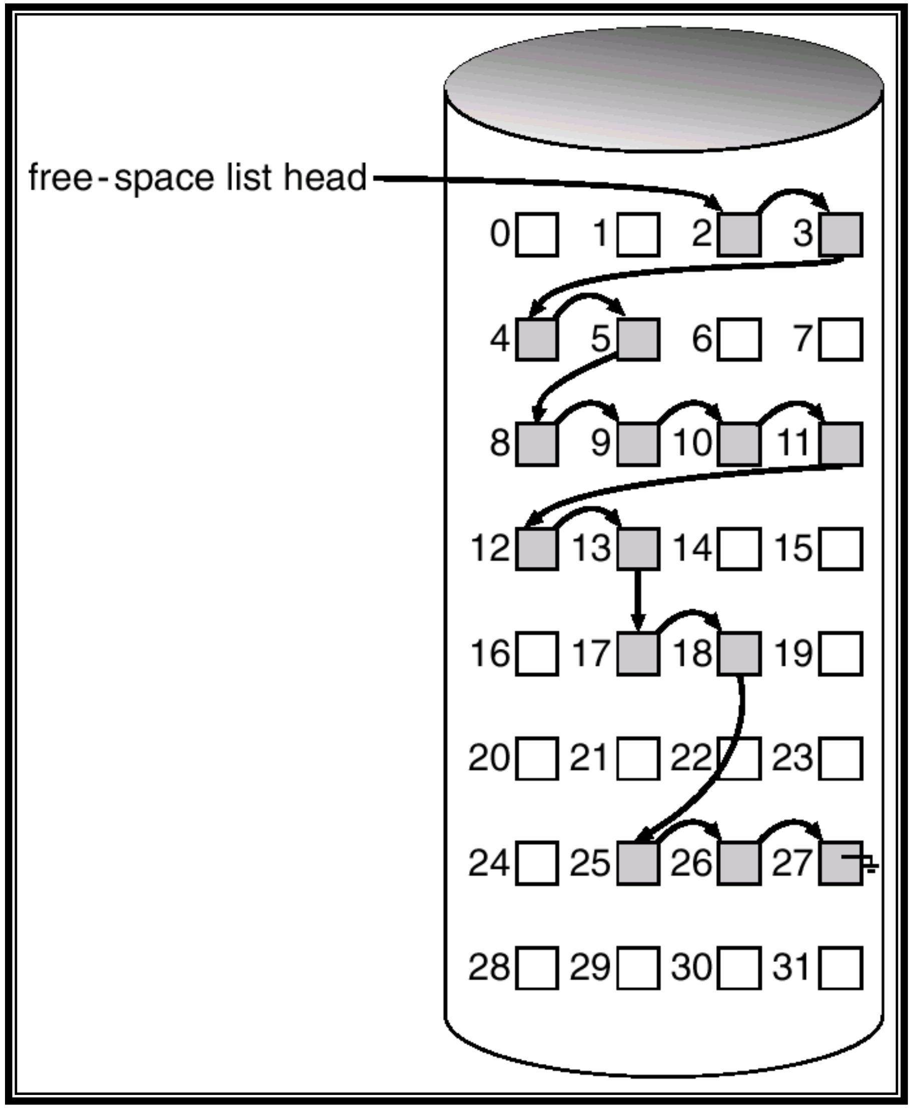
  - But easy to get contiguous files
- Linked list (free list)
  - Link together all free disk blocks. Keep pointer to 1st free block in a special location & cache it
  - Cannot get contiguous space easily
  - No waste of space
- Grouping – modification of free-list approach – an indexed approach
  - **1st block *in partition* (an index block) contains pointers to n free data blocks**
  - **Last block in the above n free blocks is another index block pointing to another n free blocks with the last of these being another pointer block, etc.**
  - **Similar the the linked index block approach for file allocation.**
  - **Advantage: the addresses of a large number of free blocks can be found quickly**
- Counting
  - **Keep track of groups of contiguous blocks.**
  - **Just need address of first block in group & number of free contiguous blocks following – similar to extents**
  - **Shorter index list**

Linked Free Space List on Disk

Free (unused) blocks physically Linked together.

Pointer to 1st block may be cached in Memory

The FAT method has free space Management built into the access scheme.



Efficiency and Performance

- Efficiency dependent on:
  - disk allocation and directory algorithms
  - types of data kept in file's directory entry **- maintaining "statistics" in the directory**
  - **UNIX: vary cluster size as file grows (minimizes internal fragmentation)**
  - **UNIX: Pre-allocates inodes on a disk to distribute this structure, wastes some space but improves performance - see below**

- Performance
  - disk cache – separate section of main memory for frequently used blocks
  - free-behind and read-ahead – techniques to optimize sequential access
  - improve PC performance by dedicating section of memory as virtual disk, or RAM disk. **History: was frequently done on PCs when all you had was a floppy.**
  - **UNIX: distribute inodes across partition - keep data blocks near files inode - minimize seeks**

- THANK YOU