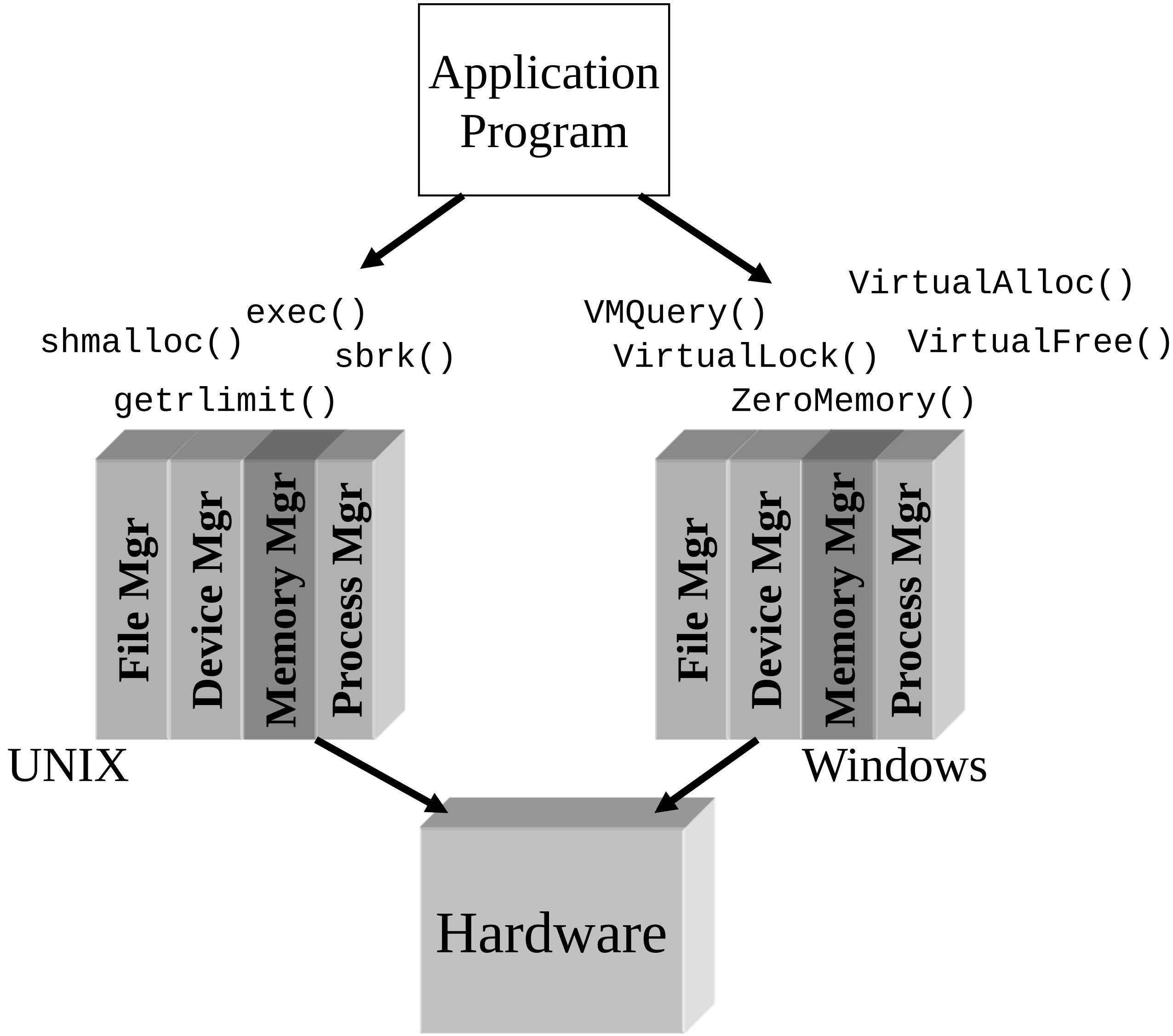
Memory Management

Raju Pandey
Department of Computer Sciences
University of California, Davis
Spring 2011

Overview

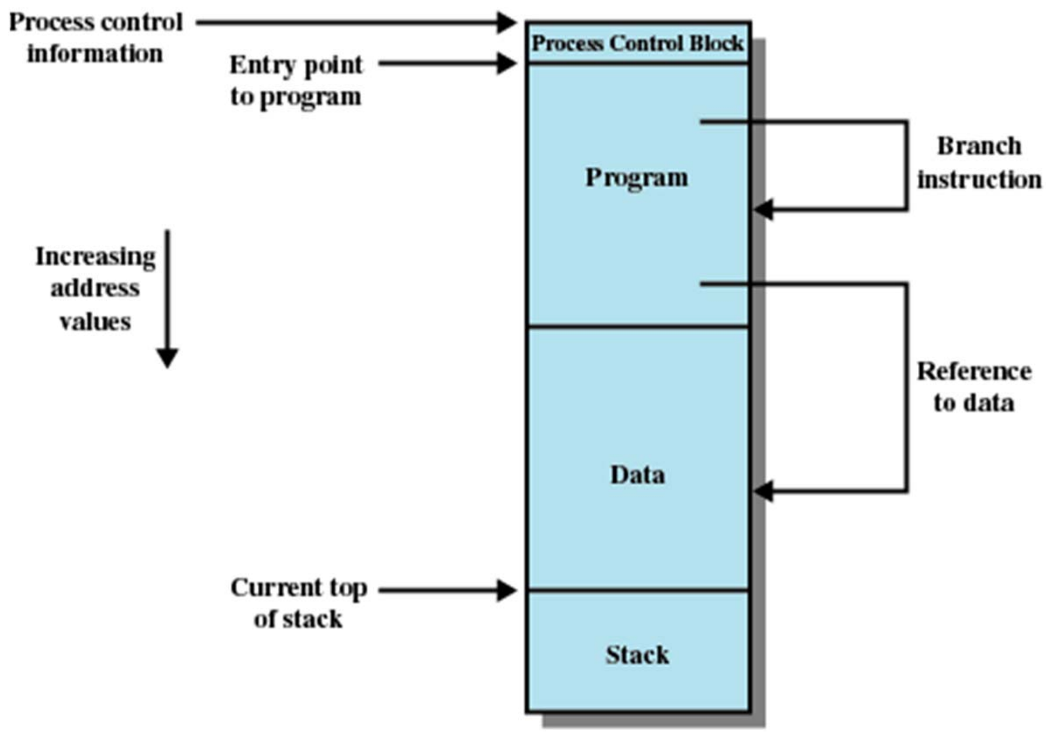
- Goals of memory management:
  - Subdividing memory to accommodate multiple processes
  - Memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time
- Preparing a Program for Execution
  - Program Transformations
  - Logical-to-Physical Address Binding
- Memory Partitioning Schemes
  - Fixed Partitions
  - Variable Partitions
- Allocation Strategies for Variable Partitions
- Dealing with Insufficient Memory

The External View of the Memory Manager



Memory Management Requirements

- Relocation
  - Programmer does not know where the program will be placed in memory when it is executed
  - While the program is executing, it may be swapped to disk and returned to main memory at a different location (relocated)
  - Memory references must be translated in the code to actual physical memory address



Memory Management Requirements

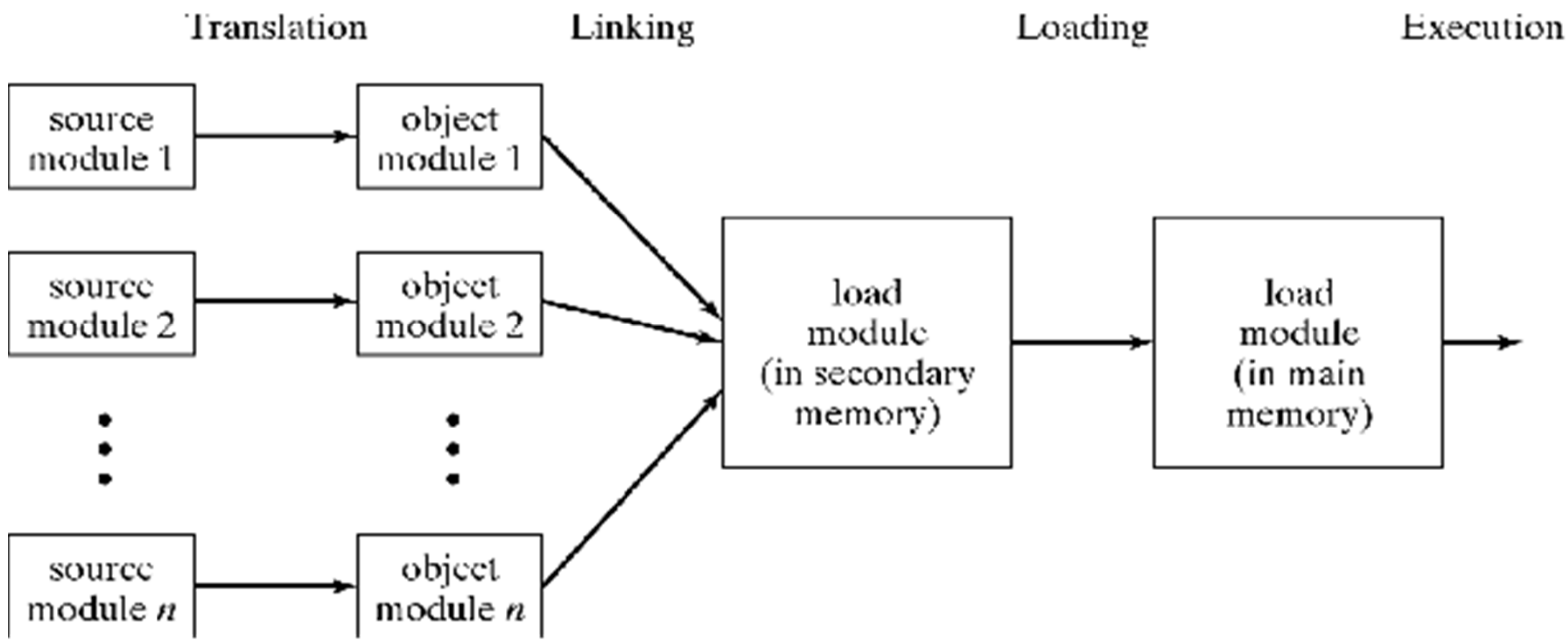
- Protection
  - Processes should not be able to reference memory locations in another process without permission
  - Impossible to check absolute addresses at compile time
  - Must be checked at run time
  - Memory protection requirement must be satisfied by the processor (hardware) rather than the operating system (software)
    - Operating system cannot anticipate all of the memory references a program will make
- Sharing
  - Allow several processes to access the same portion of memory
  - Better to allow each process access to the same copy of the program rather than have their own separate copy

Memory Management Requirements

- Logical Organization
  - Programs are written in modules
  - Modules can be written and compiled independently
  - Different degrees of protection given to modules (read-only, execute-only)
  - Share modules among processes
- Physical Organization
  - Memory available for a program plus its data may be insufficient
    - Overlaying allows various modules to be assigned the same region of memory
  - Programmer does not know how much space will be available

Preparing Program for Execution

- Program Transformations
  - Translation (Compilation)
  - Linking
  - Loading



A Sample Code Segment

```
...
static int gVar;
...
int proc_a(int arg){
        ...
        gVar = 7;
        put_record(gVar);
        ...
}
```

The Relocatable Object module

Code Segment

Relative Address	Generated Code	
0000	...	
...		
0008	entry	proc_a
...		
0220	load	=7, R1
0224	store	R1, 0036
0228	push	0036
0232	call	‘put_record’
...		
0400	External reference table	
...		
0404	‘put_record’ 0232	
...		
0500	External definition table	
...		
0540	‘proc_a’ 0008	
...		
0600	(symbol table)	
...		
0799	(last location in the code segment)	

Data Segment

Relative Address	Generated variable space
...	
0036	[Space for gVar variable]
...	
0049	(last location in the data segment)

The Absolute Program

Code Segment

Relative Address	**Generated Code**	
0000	(Other modules)	
...		
1008	entry	proc_a
...		
1220	load	=7, R1
1224	store	R1, 0136
1228	push	1036
1232	call	2334
...		
1399	(End of proc_a)	
...	(Other modules)	
2334	entry	put_record
...		
2670	(optional symbol table)	
...		
2999	(last location in the code segment)	

Data Segment

Relative Address	**Generated variable space**
...	
0136	[Space for gVar variable]
...	
1000	(last location in the data segment)

The Program Loaded at Location 4000

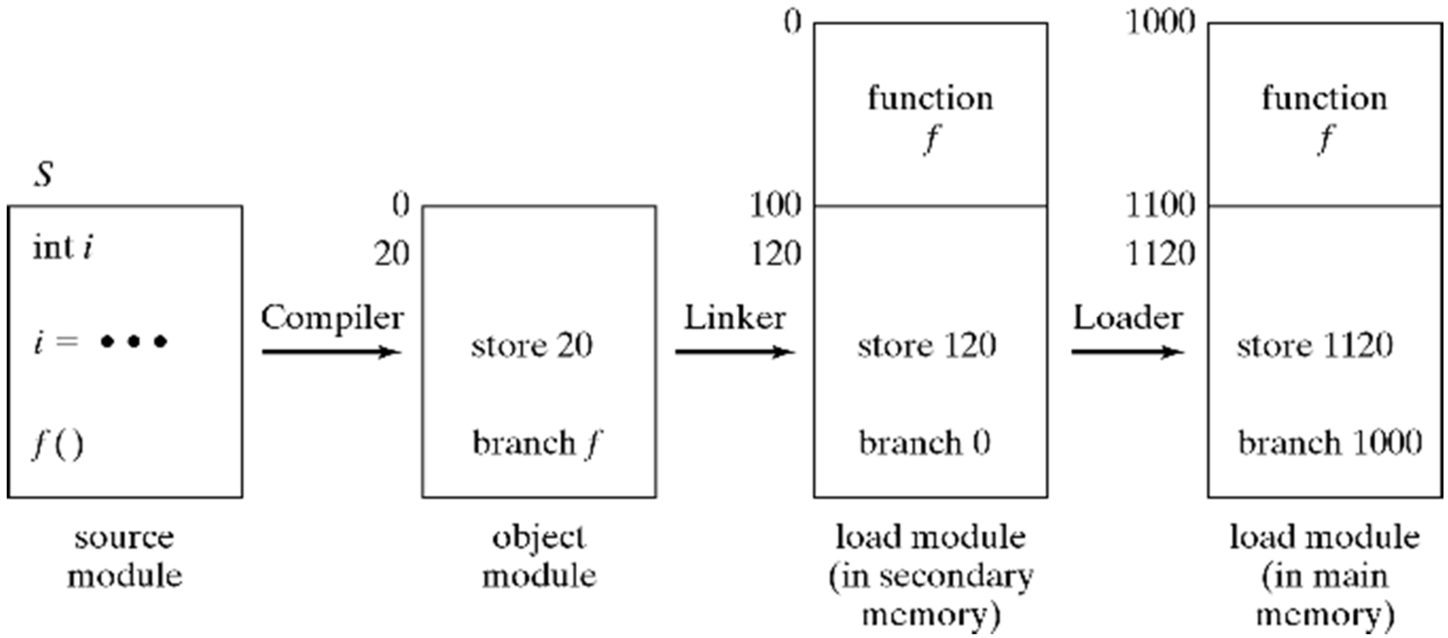
Relative Address	Generated Code
0000	(Other process's programs)
4000	(Other modules)
...	
5008	entry proc_a
...	
5036	[Space for gVar variable]
...	
5220	load =7, R1
5224	store R1, 7136
5228	push 5036
5232	call 6334
...	
5399	(End of proc_a)
...	(Other modules)
6334	entry put_record
...	
6670	(optional symbol table)
...	
6999	(last location in the code segment)
7000	(first location in the data segment)
...	
7136	[Space for gVar variable]
...	
8000	(Other process's programs)

Address Binding

- Assign Physical Addresses = Relocation
- Static binding
  - Programming time
  - Compilation time
  - Linking time
  - Loading time
- Dynamic binding
  - Execution time

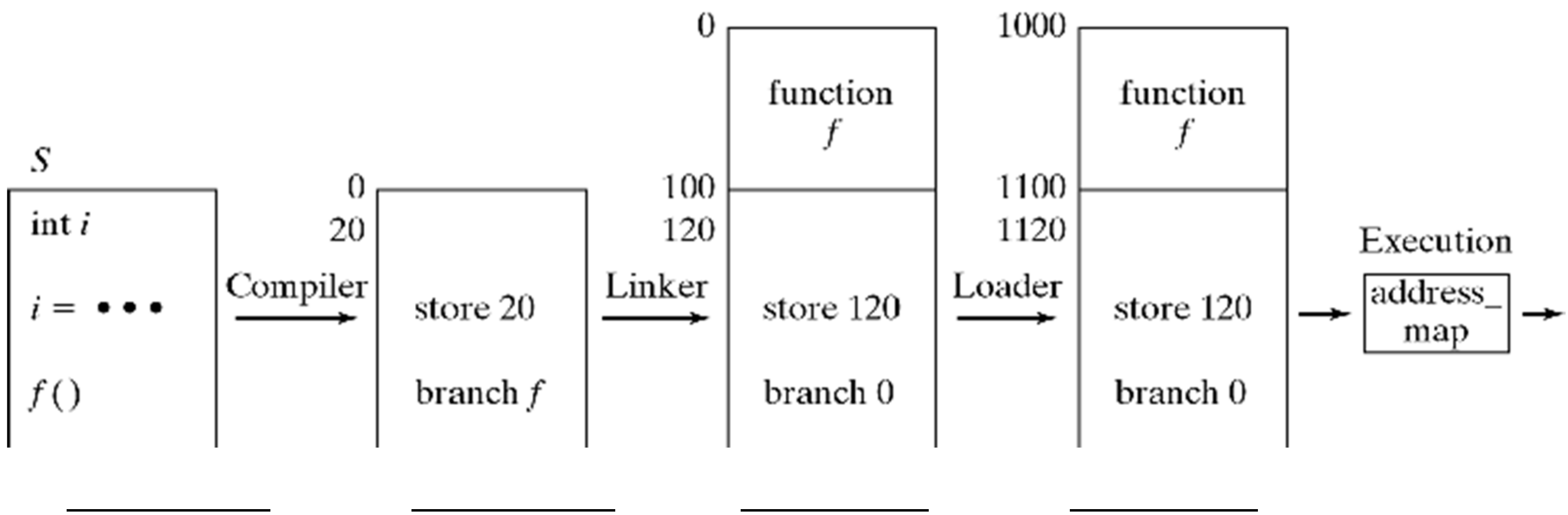
Static Address Binding

Static Binding = At Programming, Compilation, Linking, and/or Loading Time



Dynamic Address Binding

Dynamic Binding = At Execution Time



Address Binding

- How to implement dynamic binding
  - Perform for each address at run time:
    `pa = address_map(la)`
  - Simplest form of **`address_map`**:
    Relocation Register: **`pa = la + RR`**
  - More general form:
    Page/Segment Table

Fundamental Memory Management Problem

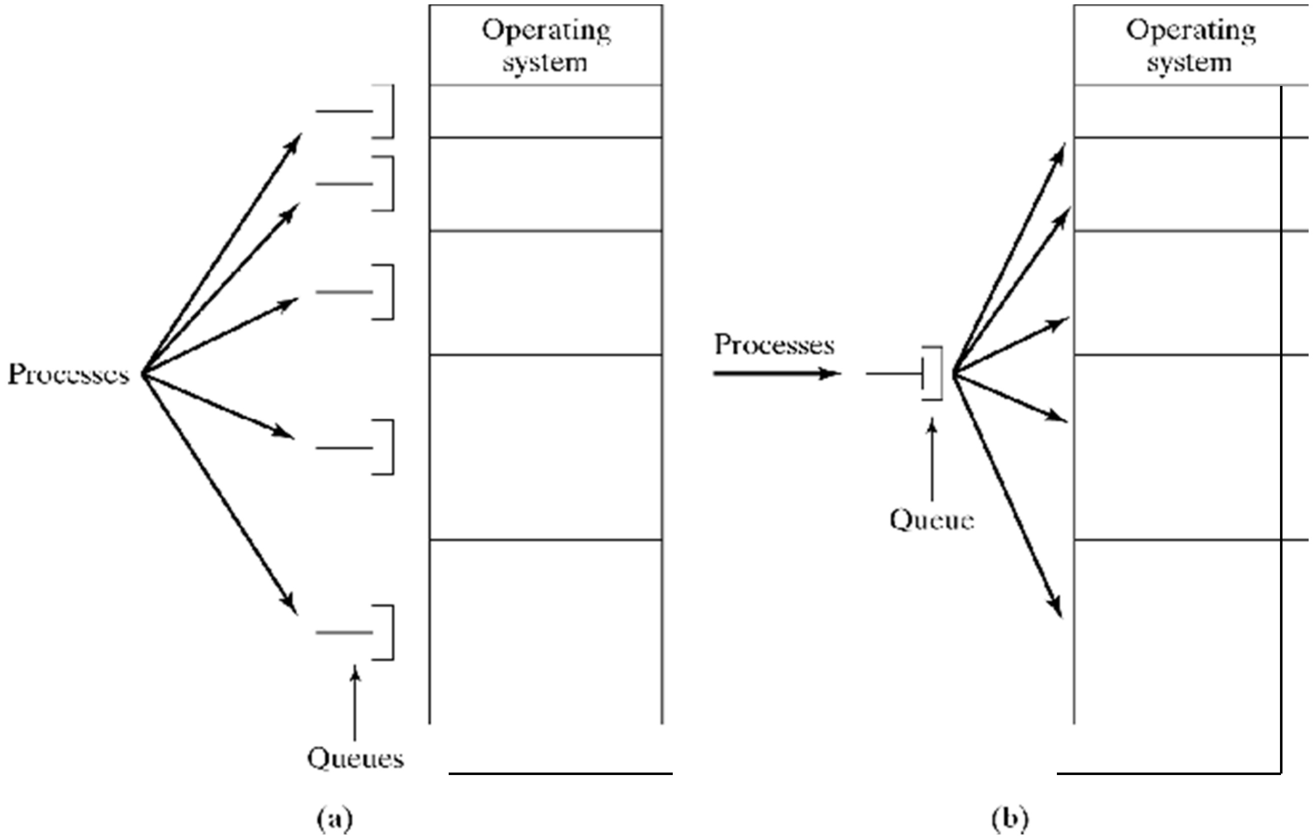
- How do we manage applications whose size may be larger than the size of memory available?
  - Partition in blocks and load as necessary
- How do we share memory resources among different processes?
- Achieved by partitioning memory
  - Look at several schemes

Memory Partitioning Schemes

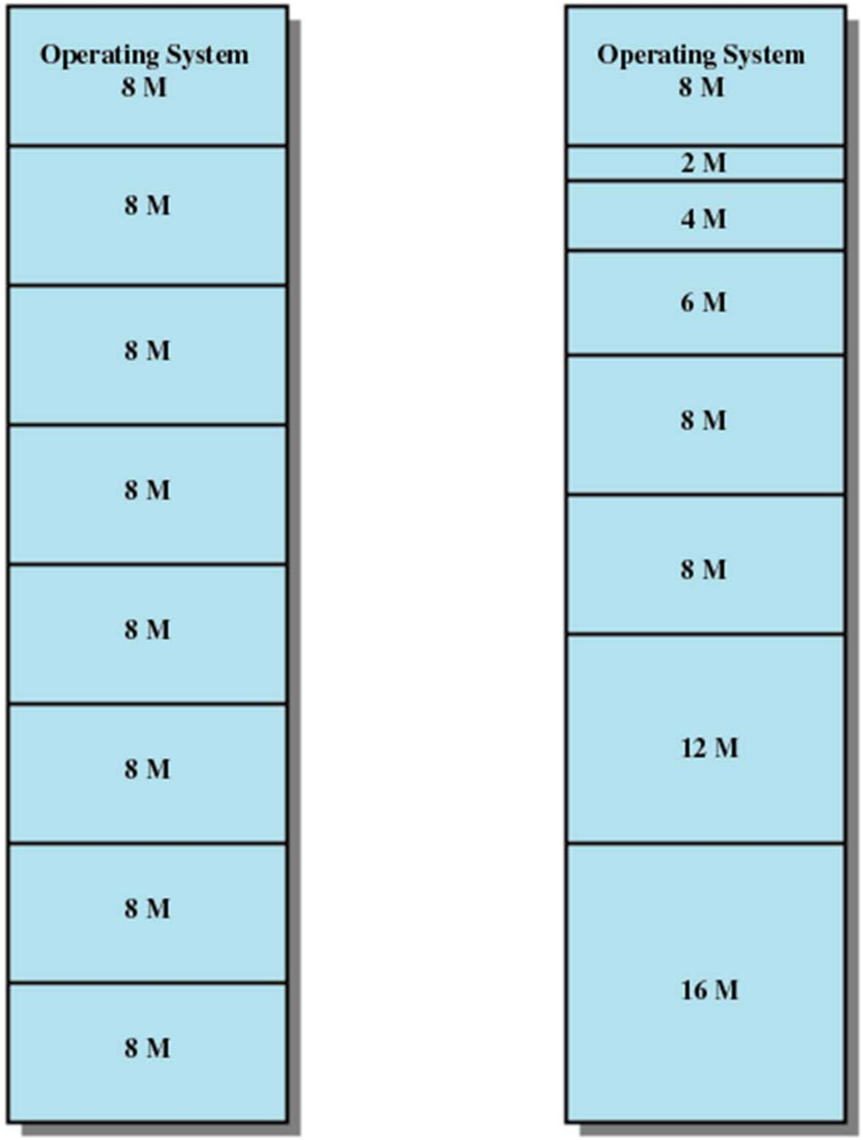
- Memory sharing schemes:
  - Single-program systems: 2 partitions (OS/user)
  - Multi-programmed:
    - Divide memory into partitions of different sizes
- Fixed partitions: size of partition determined at the time of OS initialization and cannot be changed
- Limitations of fixed partitions
  - Program size limited to largest partition
  - Internal fragmentation (unused space within partitions)
- How to assign processes to partitions
  - FIFO for each partition: Some partitions may be unused
  - Single FIFO: More complex, but more flexible
  - Choose the one that fits the best

Fixed Partitioning

Fixed partitions:
  1 queue per partition *vs* 1 queue for all partitions

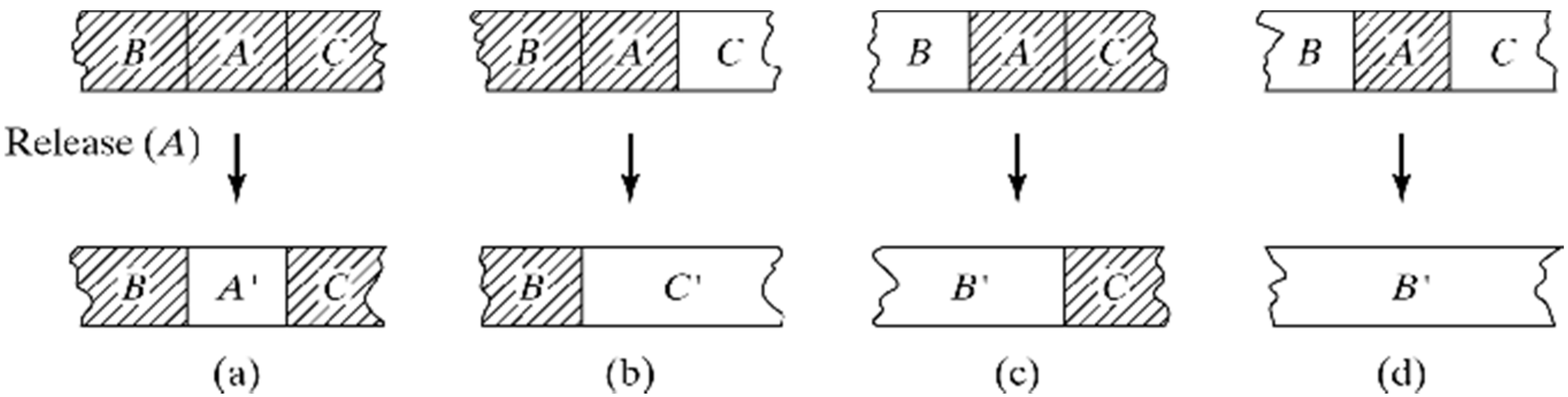


Example: Fixed Partitioning



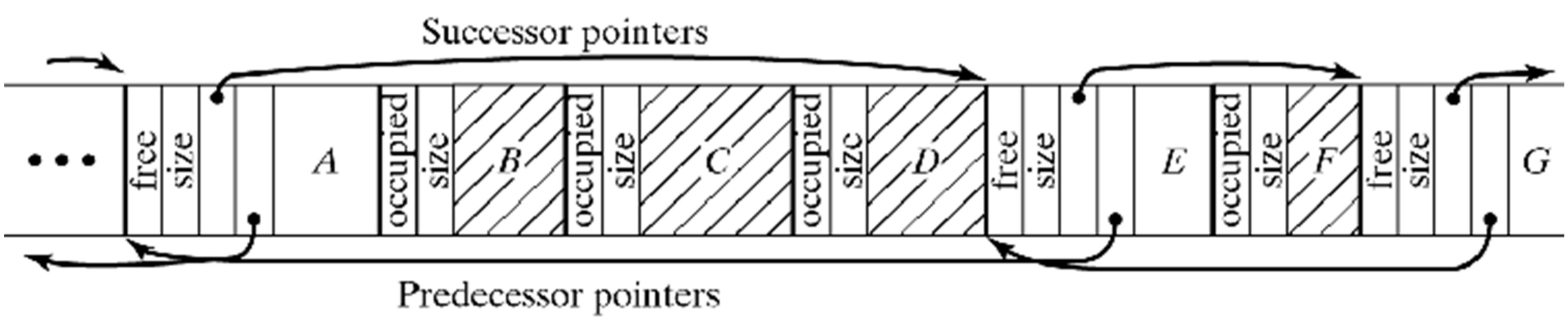
Variable Partitions

- Memory not partitioned a priori
- Each request is allocated portion of free space
- Memory = Sequence of variable-size blocks
  - Some are occupied, some are free (holes)
  - External fragmentation occurs
- Adjacent holes (right, left, or both) must be coalesced to prevent increasing fragmentation
- Major part of memory management: manage available partitions



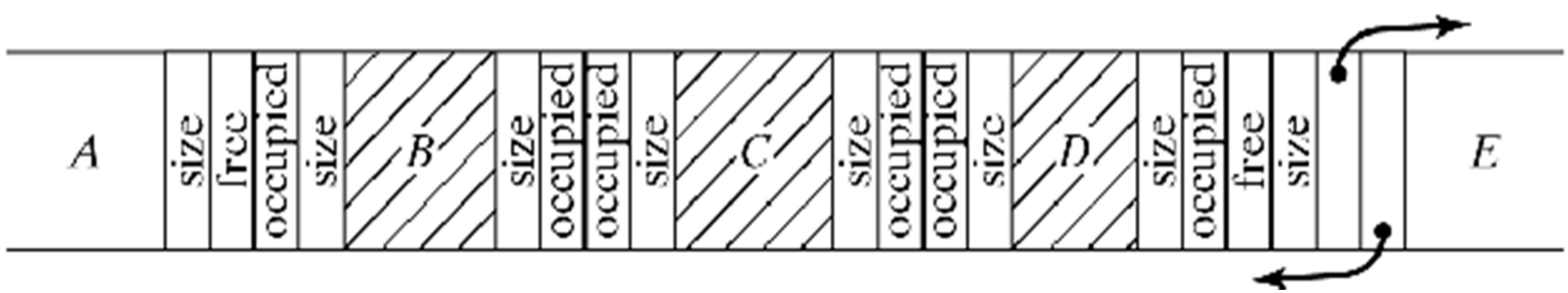
Variable Partitions:
Linked List Implementation 1

- All available space tied together through a linked list
- Type/Size tags at the start of each Block
- Holes (must be sorted by physical address)
  contain links to predecessor hole and to next hole
- Checking neighbors of released block ***b*** (=C below):
  - Right neighbor (easy): Use size of ***b***
  - Left neighbor (clever): Use sizes to find first hole to ***b***'s right, follow its predecessor link to first hole on ***b***'s left, and check if it is adjacent to ***b***.



Variable Partitions:
Linked List Implementation 2

- Better solution:
  Replicate tags at end of blocks (need not be sorted)
- Checking neighbors of released block ***b***:
  - Right neighbor: Use size of ***b*** as before
  - Left neighbor: Check its (adjacent) type/size tags



Bitmap Implementation

- Memory divided into fix-size blocks
- Each block represented by a 0/1 bit in a binary string: the "***bitmap***"
- Can be implemented as **`char`** or **`int`** array
- Operations use bit masks
  - Release: **`B[i] = B[i] & '11011111'`**
  - Allocate: **`B[i] = B[i] | '11000000'`**
  - Search: Repeatedly, Check left-most bit and Shift mask right: **`TEST = B[i] & '10000000'`**

The Buddy System

- Compromise between fixed and variable partitions
- Fixed number of possible hole sizes; typically, 2^i
  - Each hole can be divided (equally) into 2 *buddies.*
  - Track holes by size on separate lists
- When $\boldsymbol{n}$ bytes requested, find smallest $\boldsymbol{i}$ so that $\boldsymbol{n \leq 2^i}$
  - If hole of this size available, allocate it;
    otherwise, consider larger holes.
    Recursively split each hole into two buddies
    until smallest adequate hole is created
    .Allocate it and place other holes on appropriate lists
- On release, recursively coalesce buddies
  - Buddy searching for coalescing can be inefficient

The Buddy System

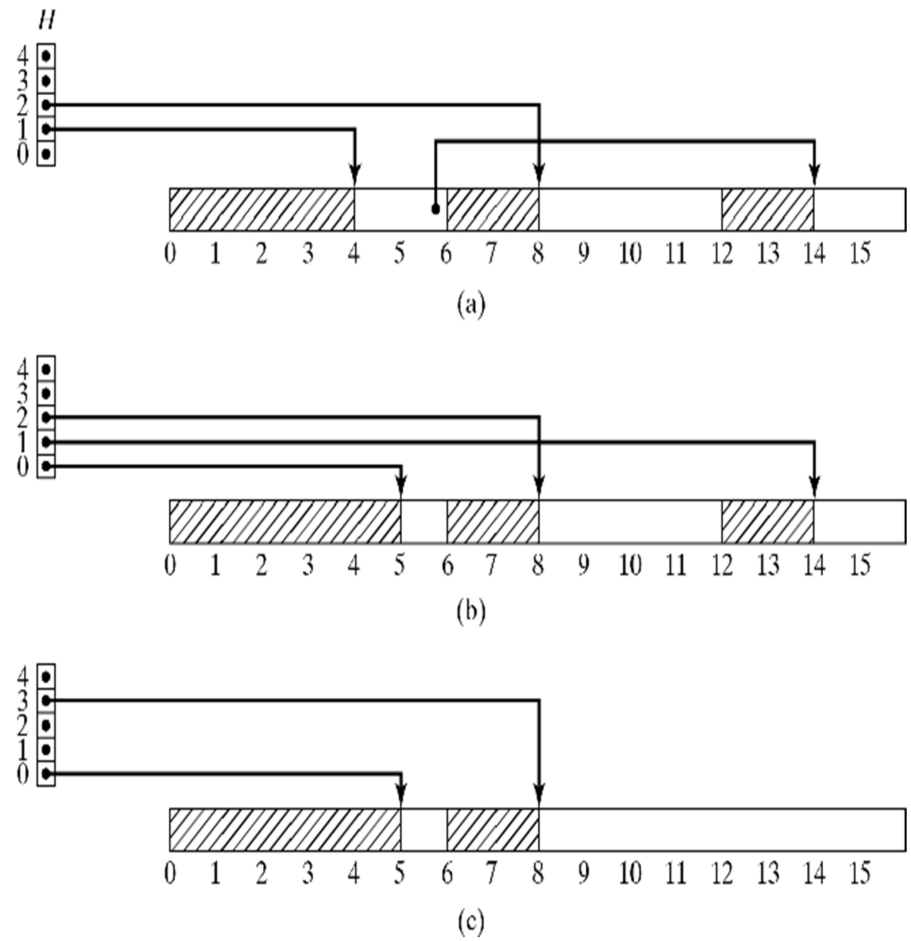
- Assume: Memory of 16 allocation units

Sizes: 1, 2, 4, 8, 16

a) 3 blocks allocated & 3 holes left

b) Block of size 1 allocated

c) Block 12-13 released



Allocation Strategies

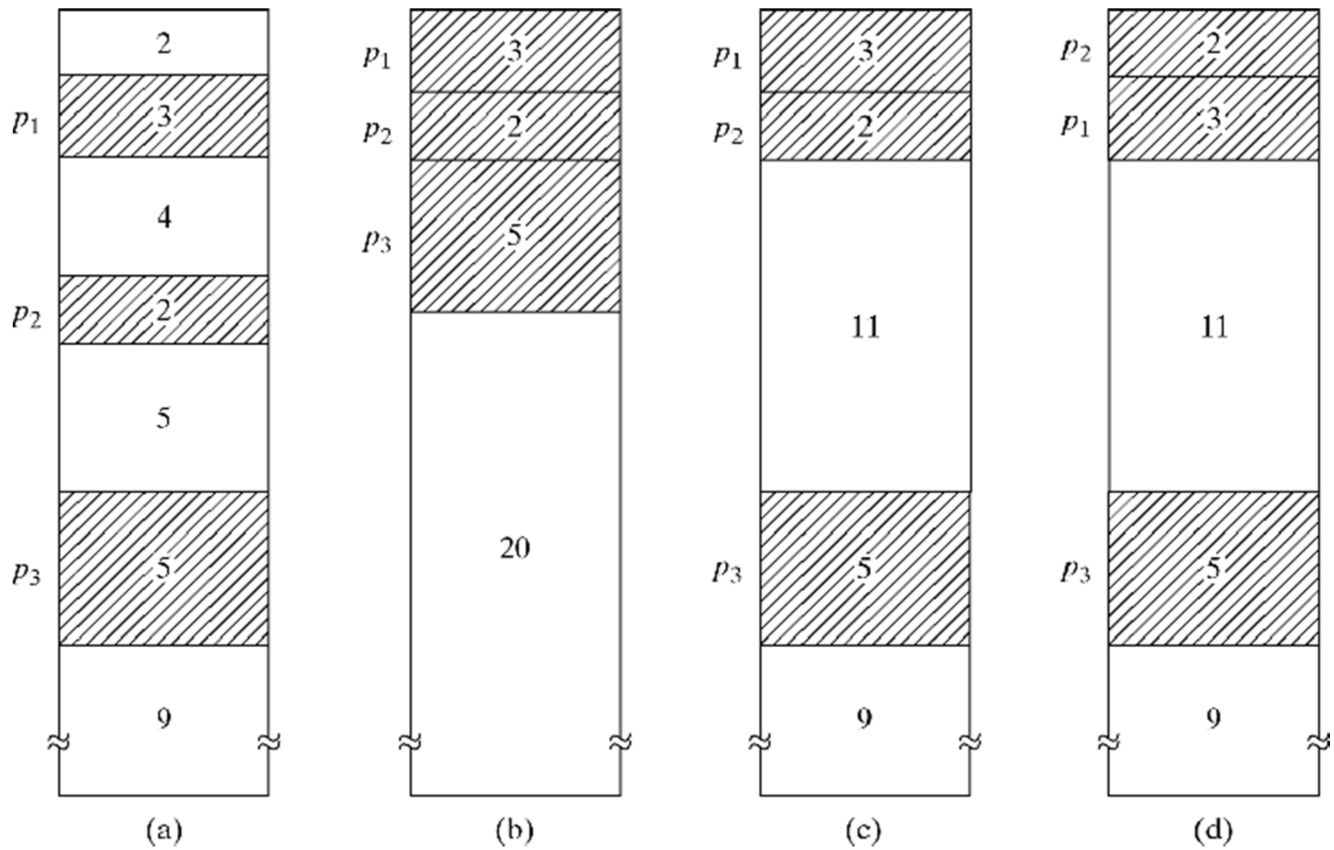
- Problem: Given a request for $\boldsymbol{n}$ bytes, find hole $\geq \boldsymbol{n}$
- Constraints:
  - Maximize memory utilization
    (minimize "*external fragmentation*")
  - Minimize search time
- Search Strategies:
  - First-fit: Always start at same place. Simplest.
  - Next-fit: Resume search. Improves distribution of holes.
  - Best-fit: Closest fit. Avoid breaking up large holes.
  - Worst-fit: Largest fit. Avoid leaving tiny hole fragments
- First Fit is generally the best choice

Dealing with Insufficient Memory

- Memory compaction
  - How much and what to move?
- Swapping
  - Temporarily move process to disk
  - Requires dynamic relocation
- Overlays
  - Allow programs large than physical memory
  - Programs loaded as needed according to calling structure.

Dealing with Insufficient Memory

Memory compaction

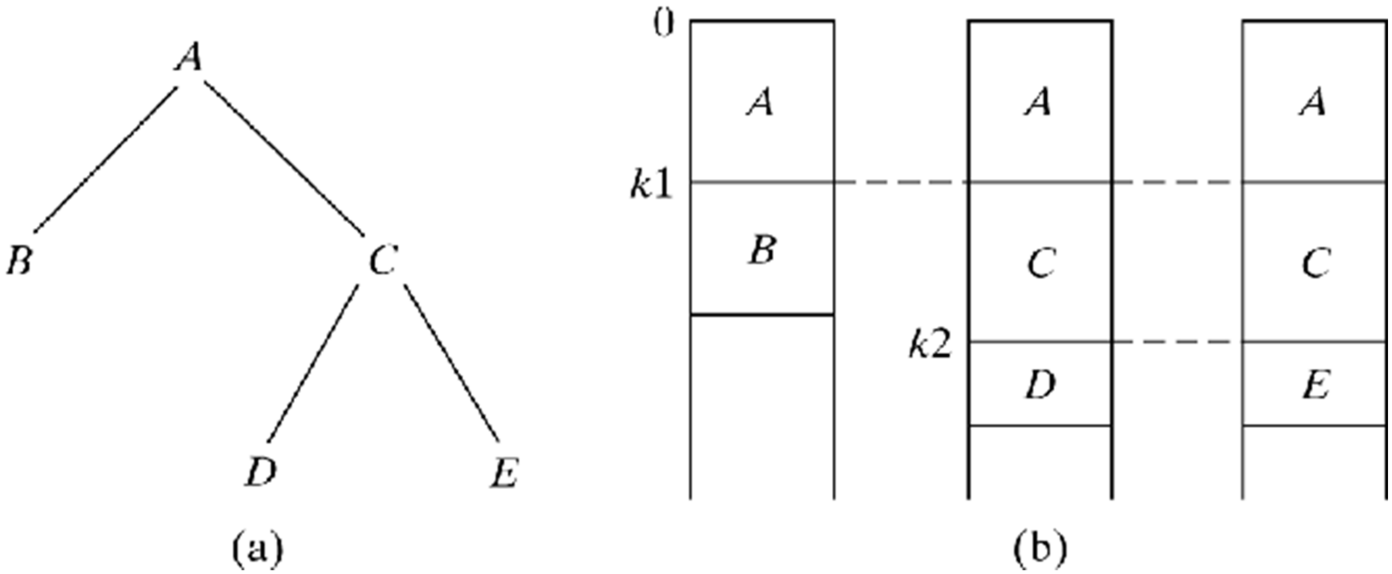


Initial Complete Partial Minimal Movement

Dealing with Insufficient Memory

Overlays

- Allow programs large than physical memory
- Programs loaded as needed
  according to calling structure



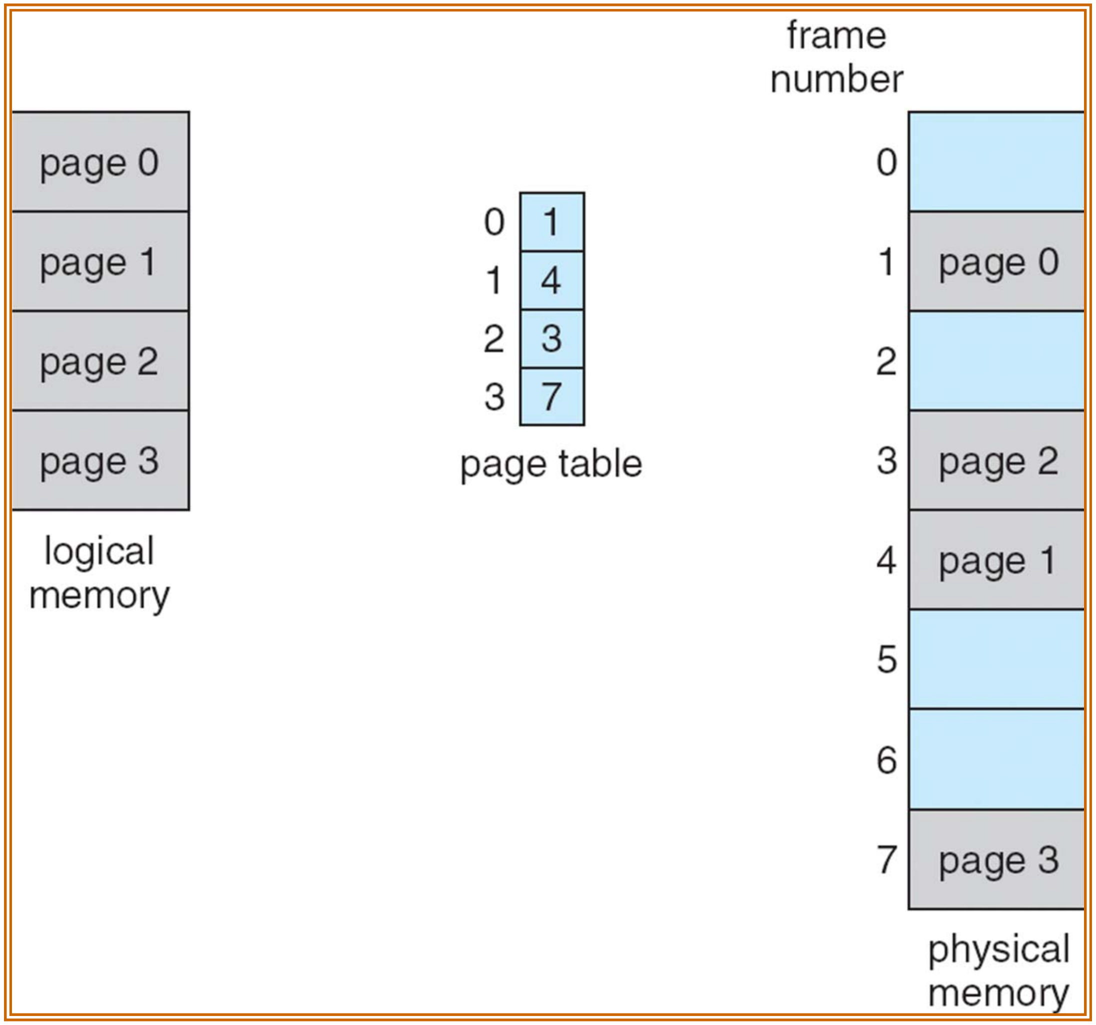
Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8192 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
- Internal fragmentation

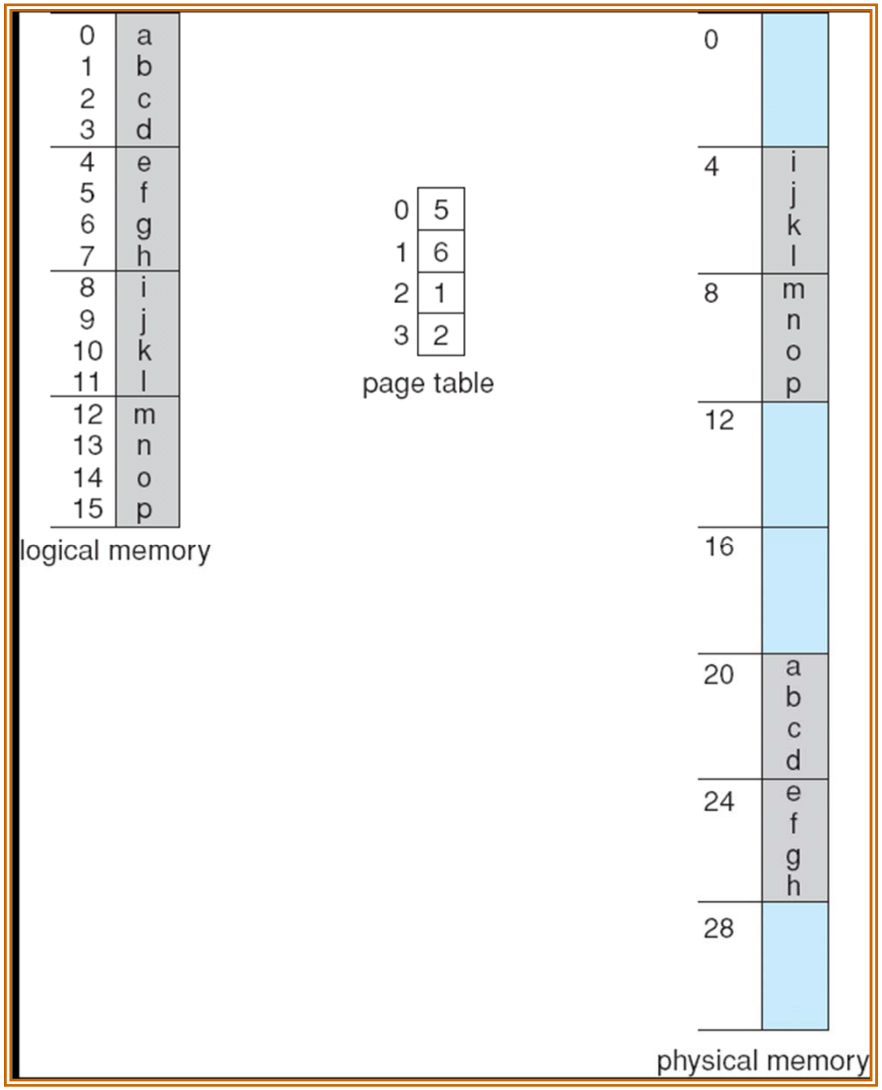
Address Translation Scheme

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

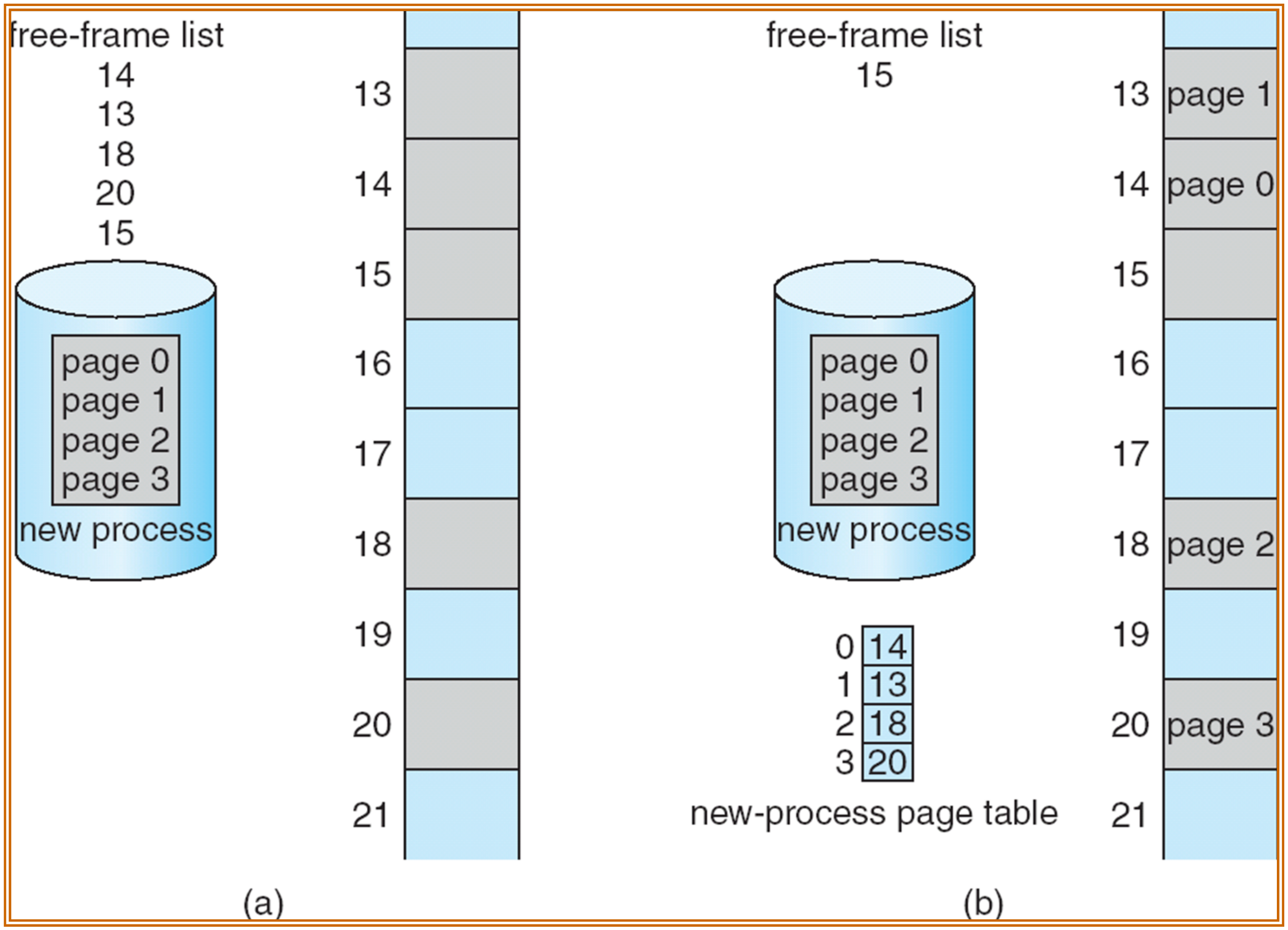
Paging Example



Paging Example



Free Frames



free-frame list
14
13
18
20
15
page 0
page 1
page 2
page 3
new process
13
14
15
16
17
18
19
20
21
(a)
free-frame list
15
page 0
page 1
page 2
page 3
new process
0 14
1 13
2 18
3 20
new-process page table
13 page 1
14 page 0
15
16
17
18 page 2
19
20 page 3
21
(b)

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)**

Associative Memory

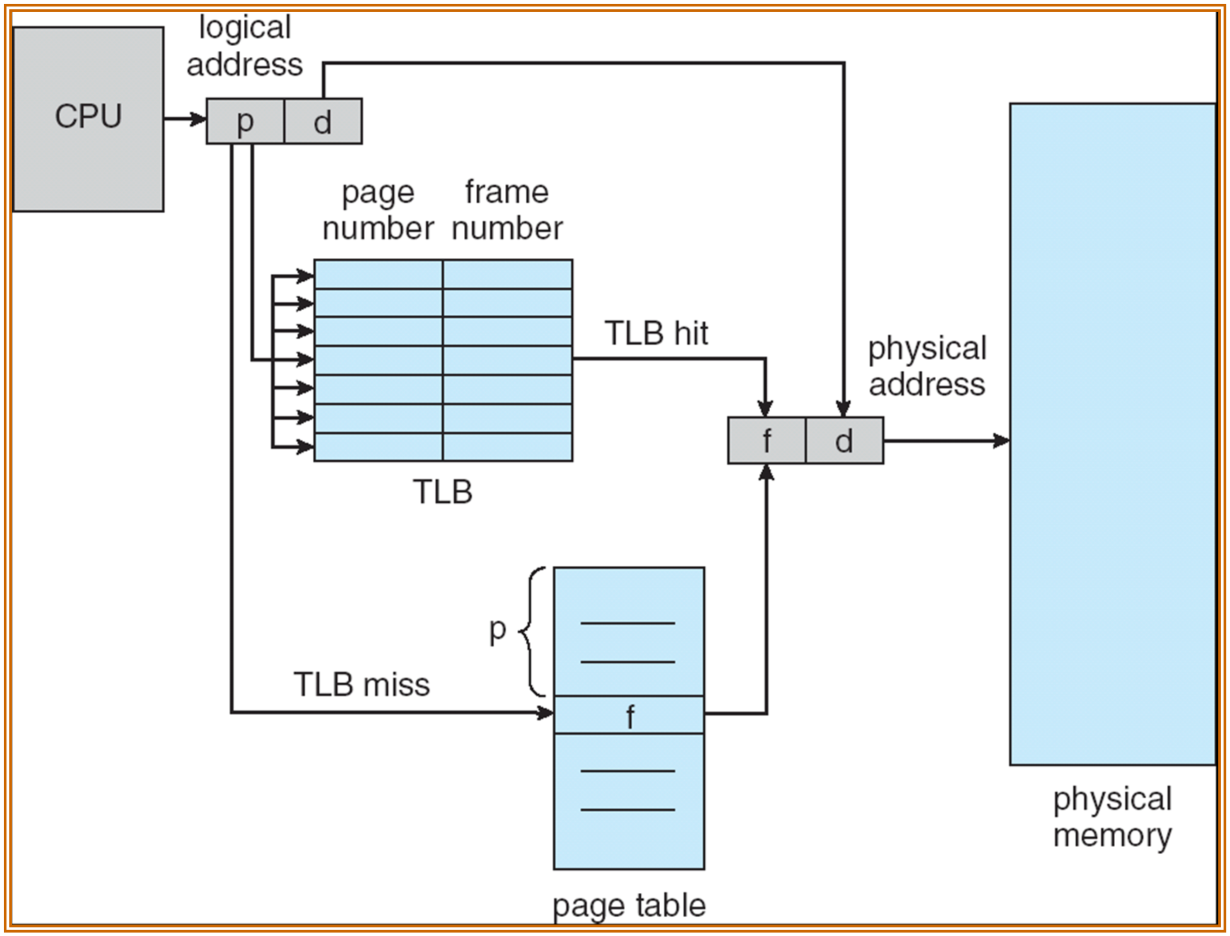
- Associative memory – parallel search

Page #	Frame #

Address translation (A´, A´´)

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

Paging Hardware With TLB



Effective Access Time

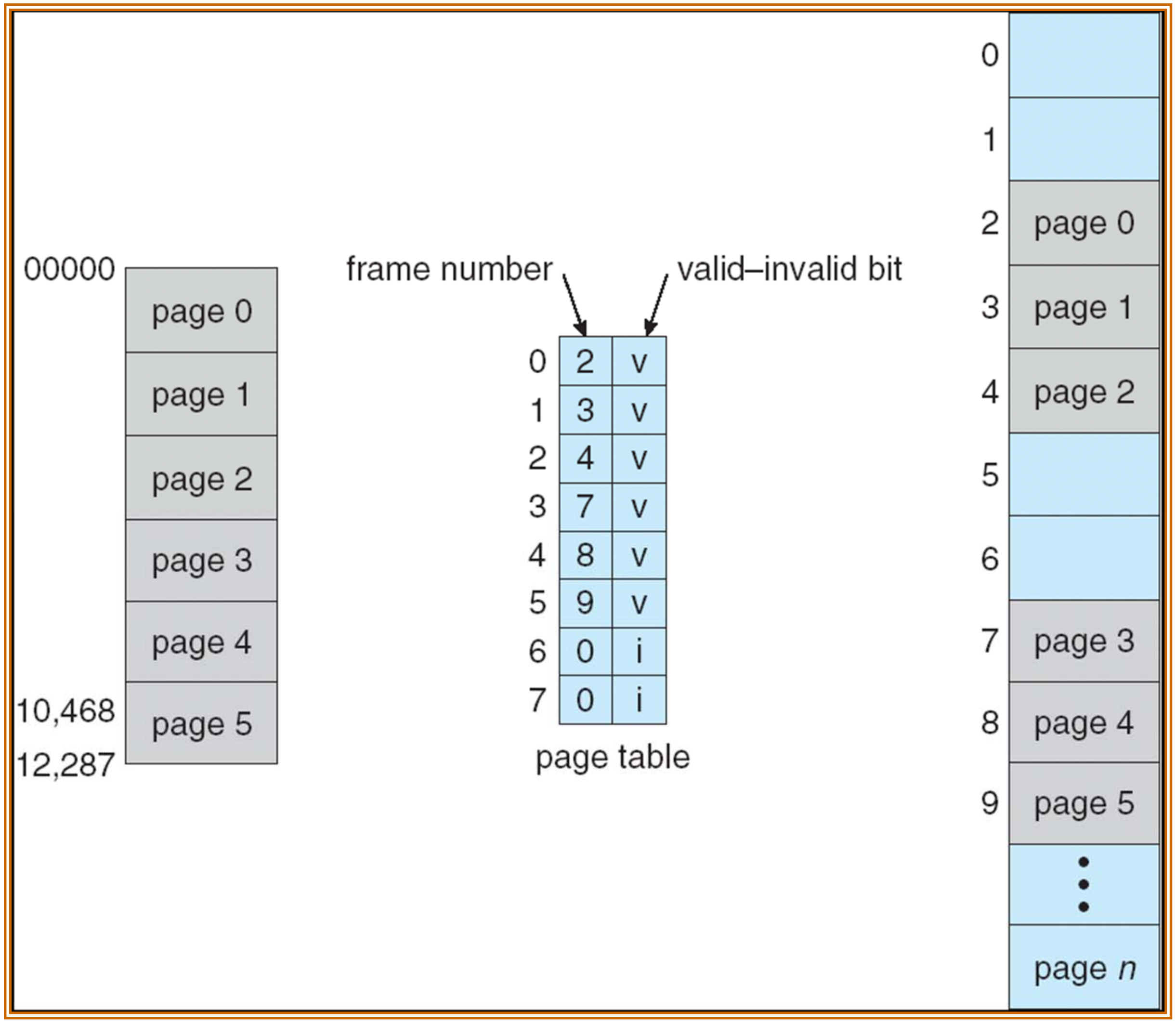
- Associative Lookup = ε time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ration related to number of associative registers
- Hit ratio = α
- **Effective Access Time** (EAT)

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

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



Page Table Structure

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table

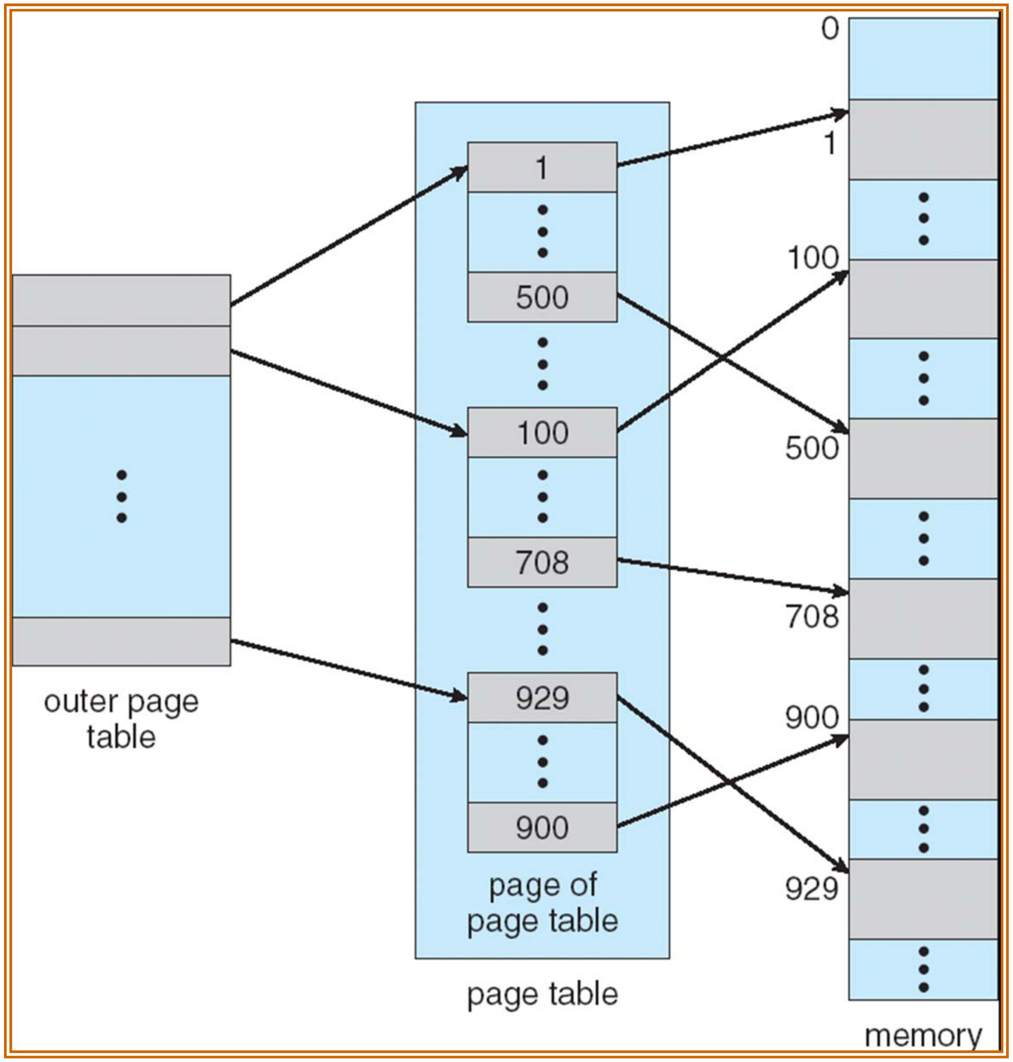
Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

page number		page offset
p_i	p_2	d
10	10	12

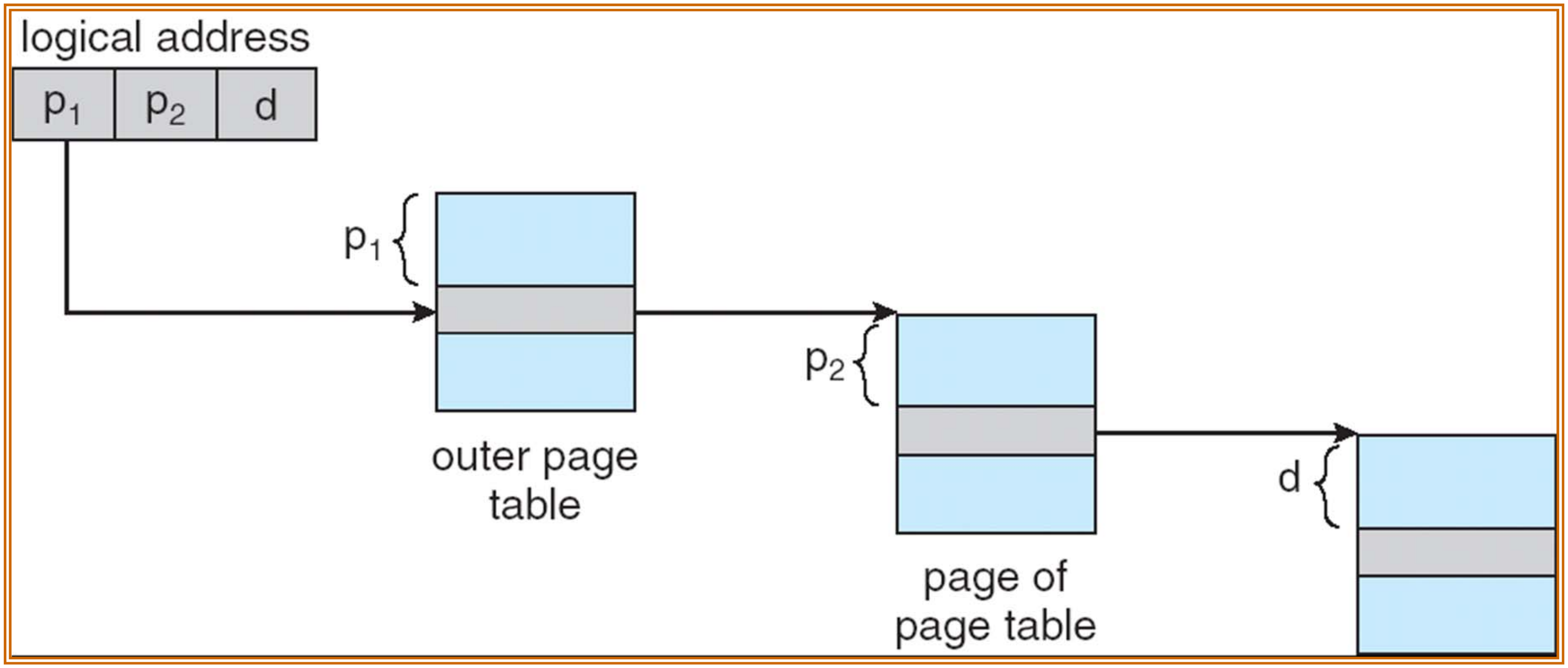
where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

Two-Level Page-Table Scheme



Address-Translation Scheme

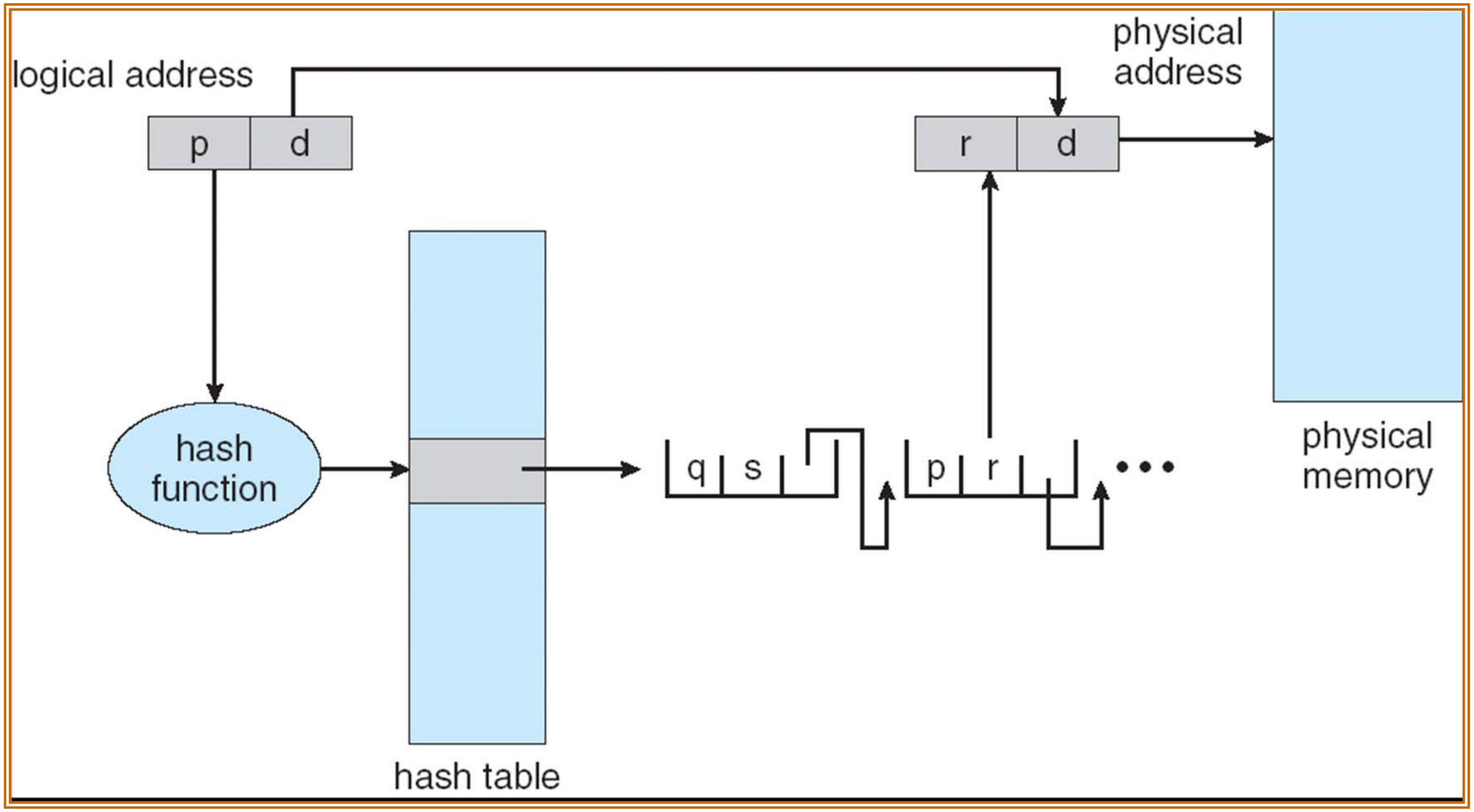
- Address-translation scheme for a two-level 32-bit paging architecture



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.

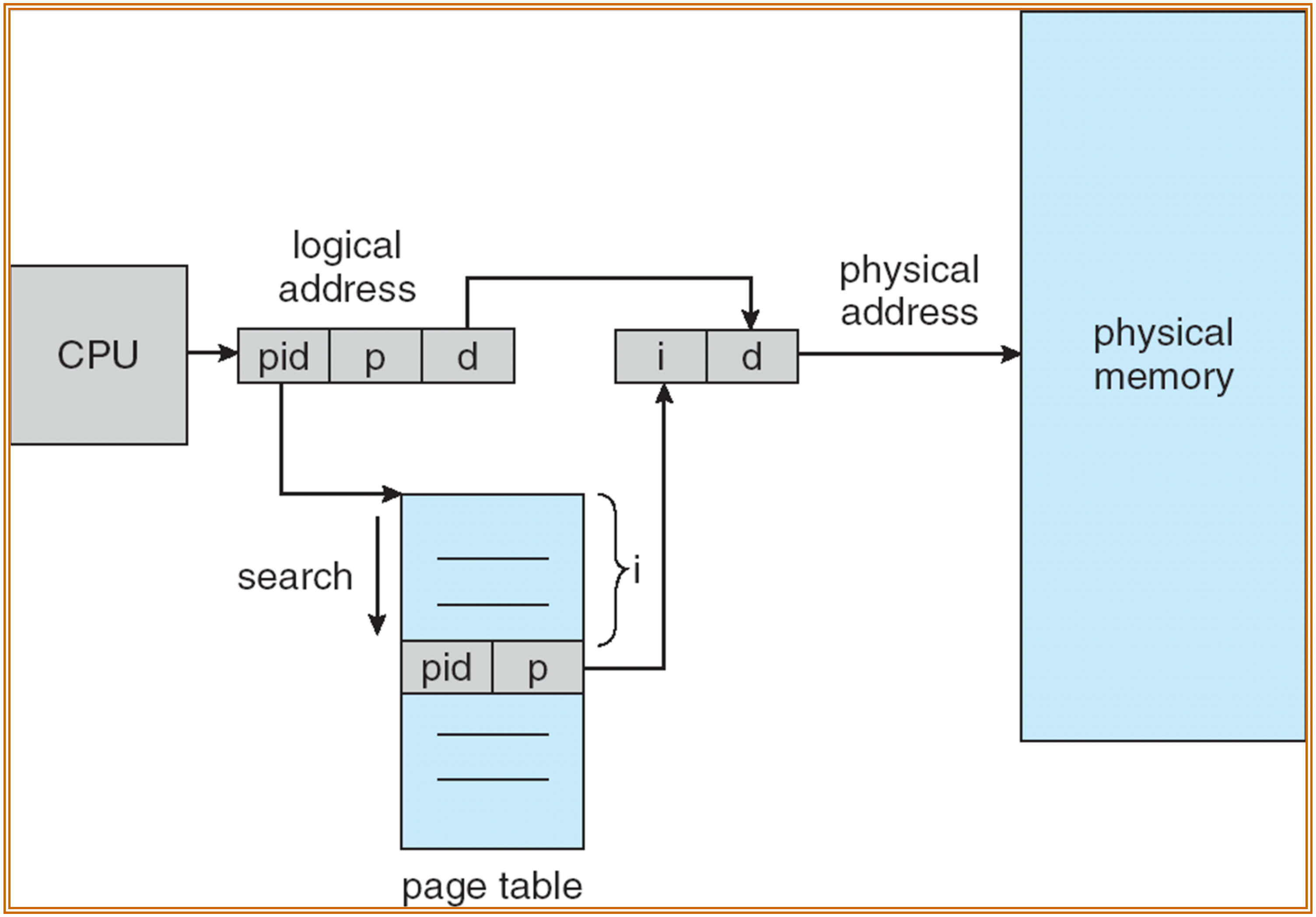
Hashed Page Table



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

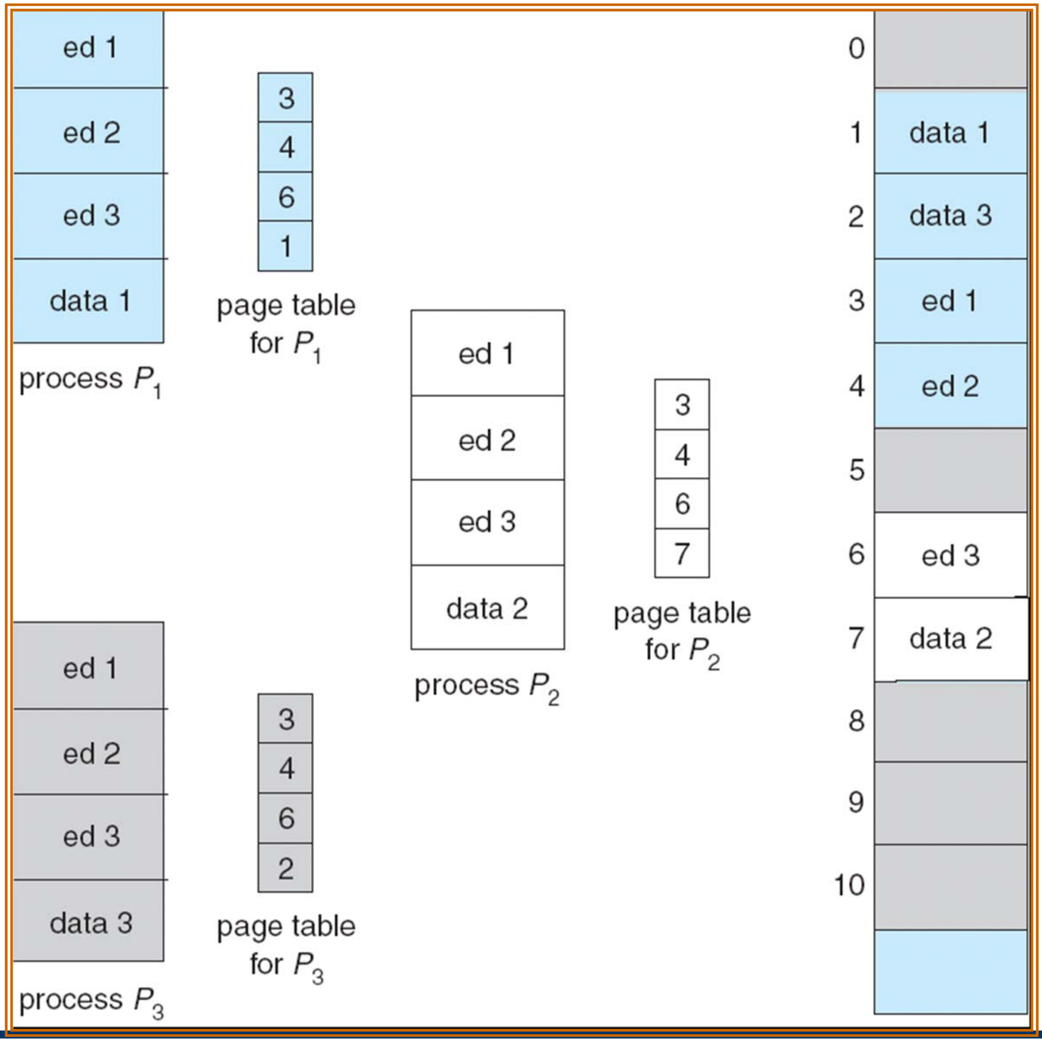
Inverted Page Table Architecture



Shared Pages

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Shared Pages Example



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,

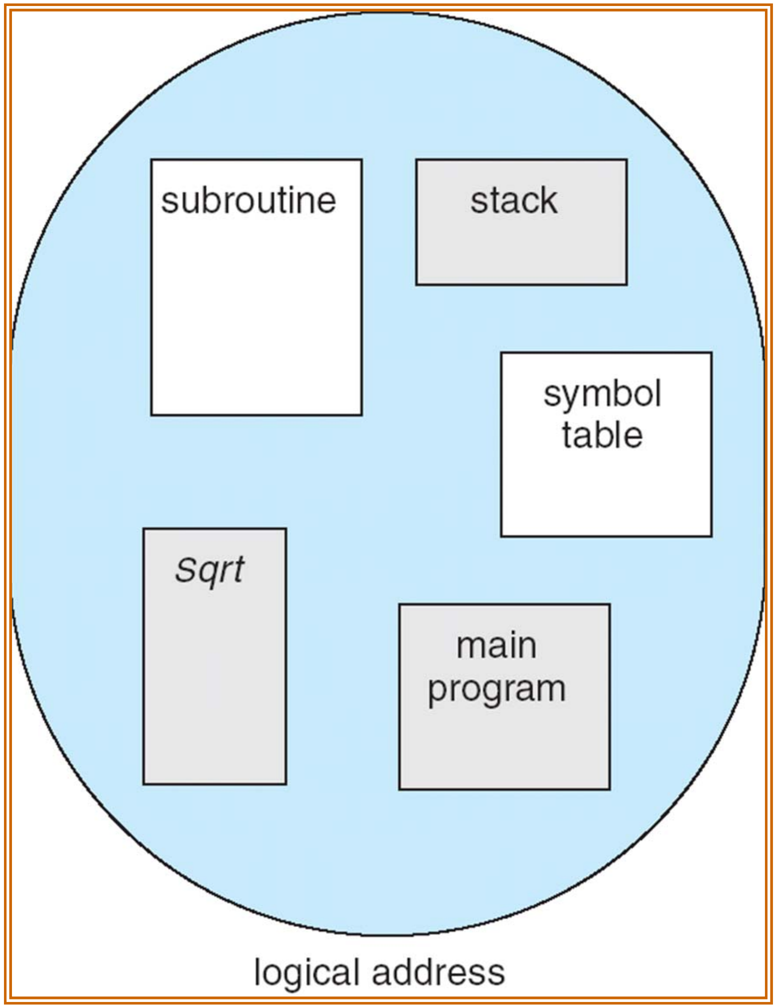
  local variables, global variables,

  common block,

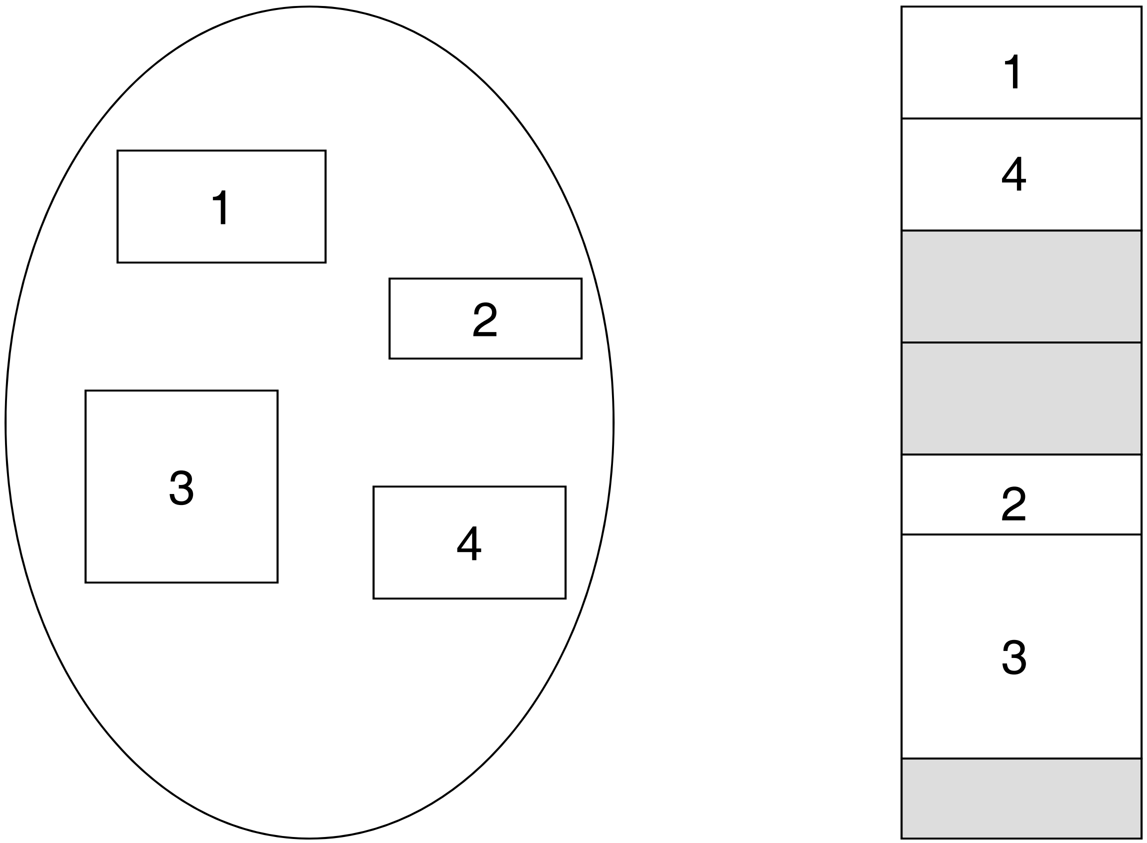
  stack,

  symbol table, arrays

User's View of a Program



Logical View of Segmentation



user space

physical memory space

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

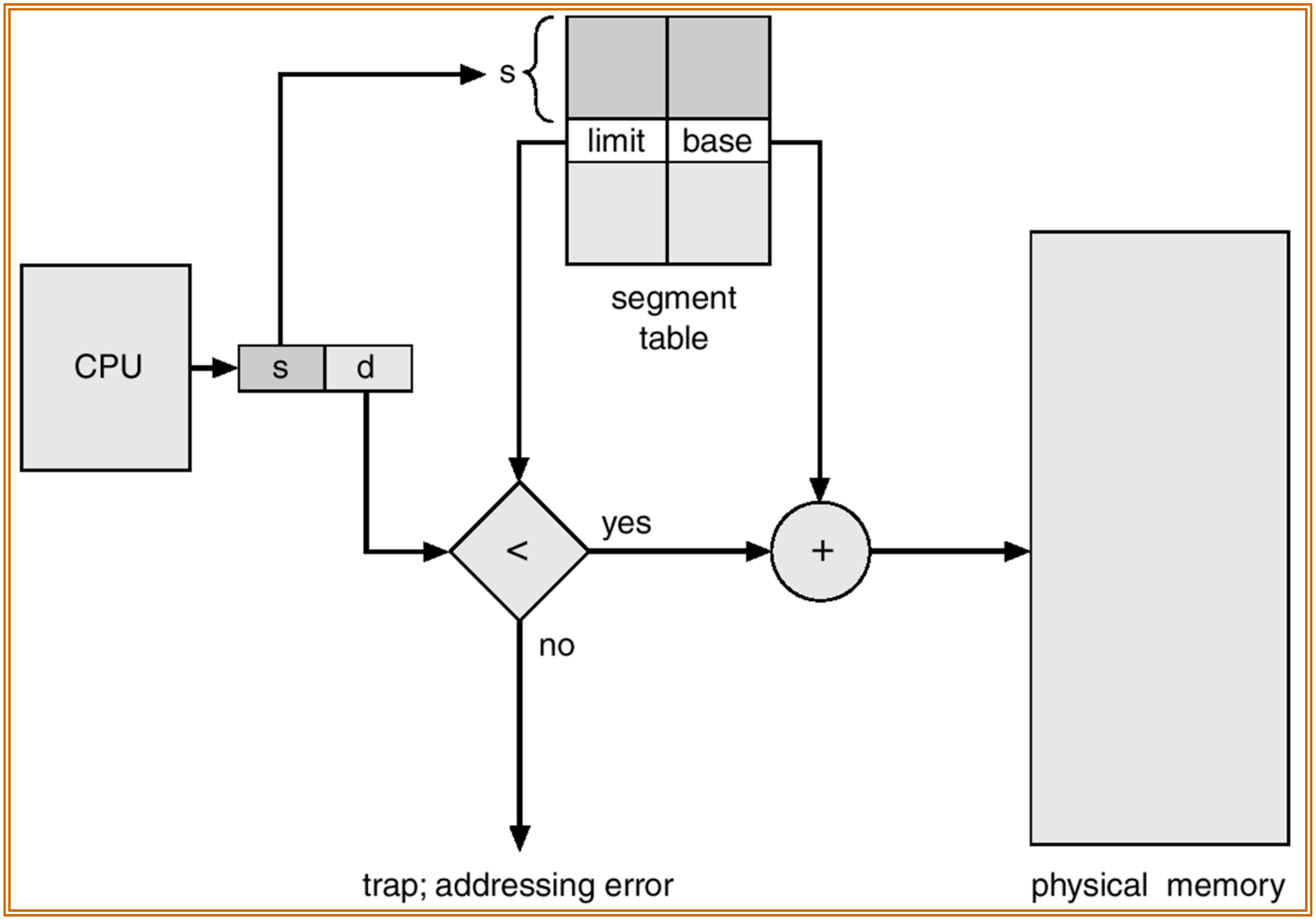
Segmentation Architecture (Cont.)

- **Relocation**.
  - dynamic
  - by segment table
- **Sharing**.
  - shared segments
  - same segment number
- **Allocation**.
  - first fit/best fit
  - external fragmentation

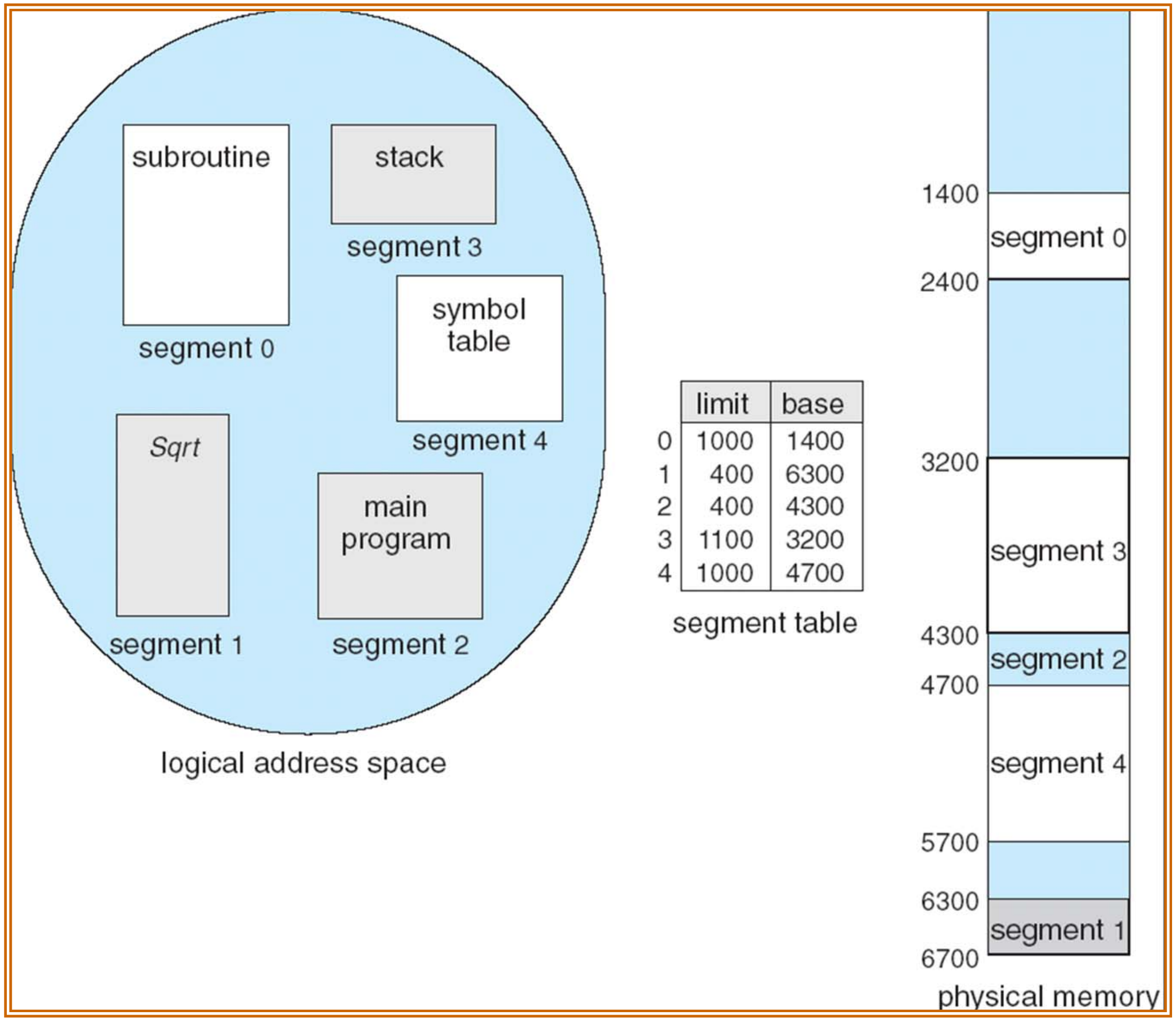
Segmentation Architecture (Cont.)

- 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

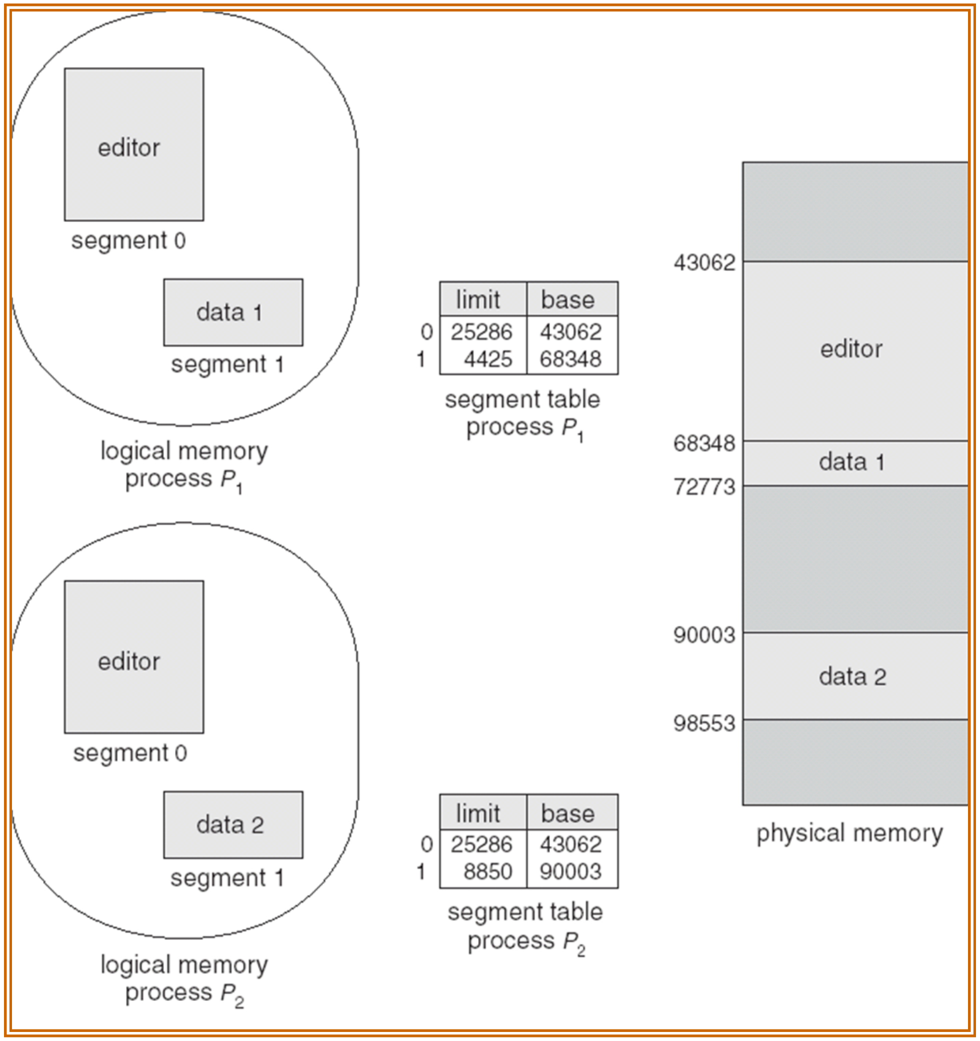
Segmentation Hardware



Example of Segmentation



Sharing of Segments



Virtual Memory

Virtual Memory

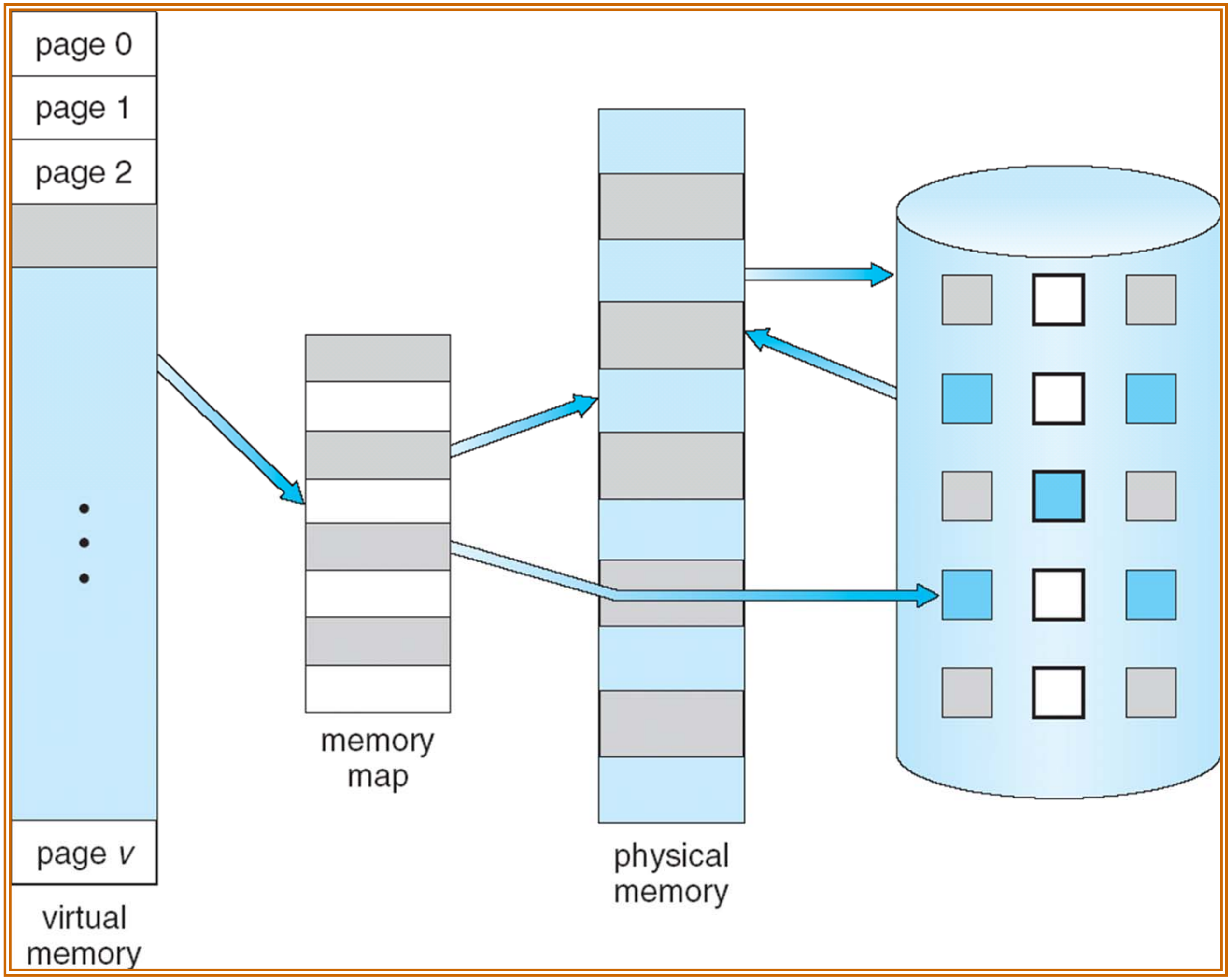
- Background
- Demand Paging
- Process Creation
- Page Replacement
- Allocation of Frames
- Thrashing
- Demand Segmentation
- Operating System Examples

Background

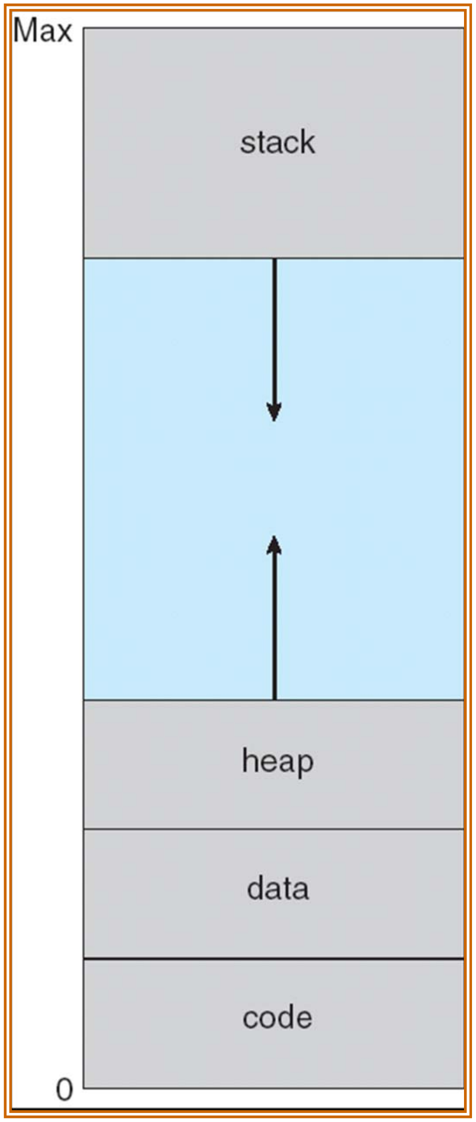
- **Virtual memory** – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - Allows address spaces to be shared by several processes.
  - Allows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

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

Virtual Memory That is Larger Than Physical Memory

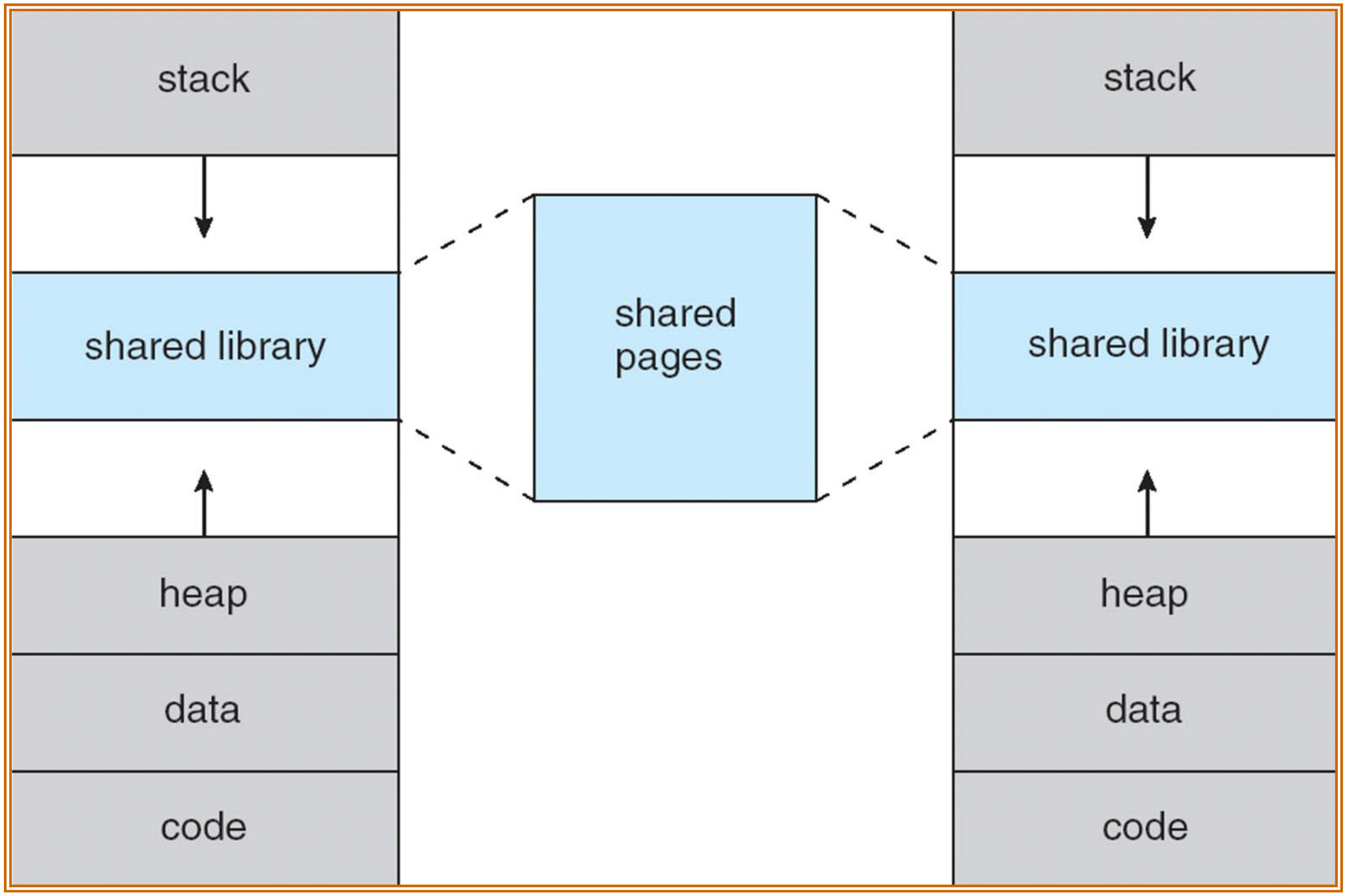


Virtual-address Space



Virtual Memory has Many Uses

- It can enable processes to share memory

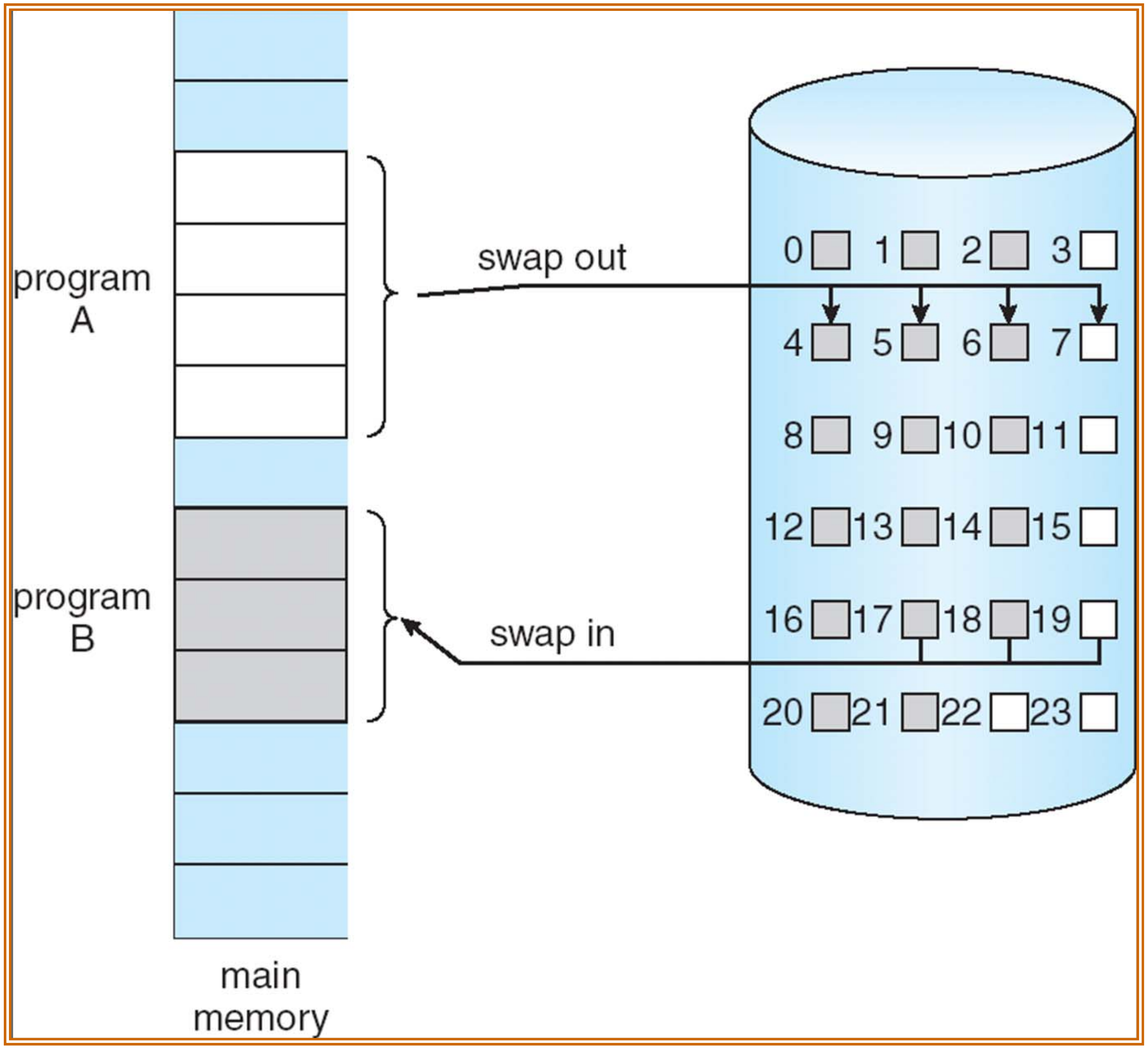


Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users

- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory

Transfer of a Paged Memory to Contiguous Disk Space



Valid-Invalid Bit

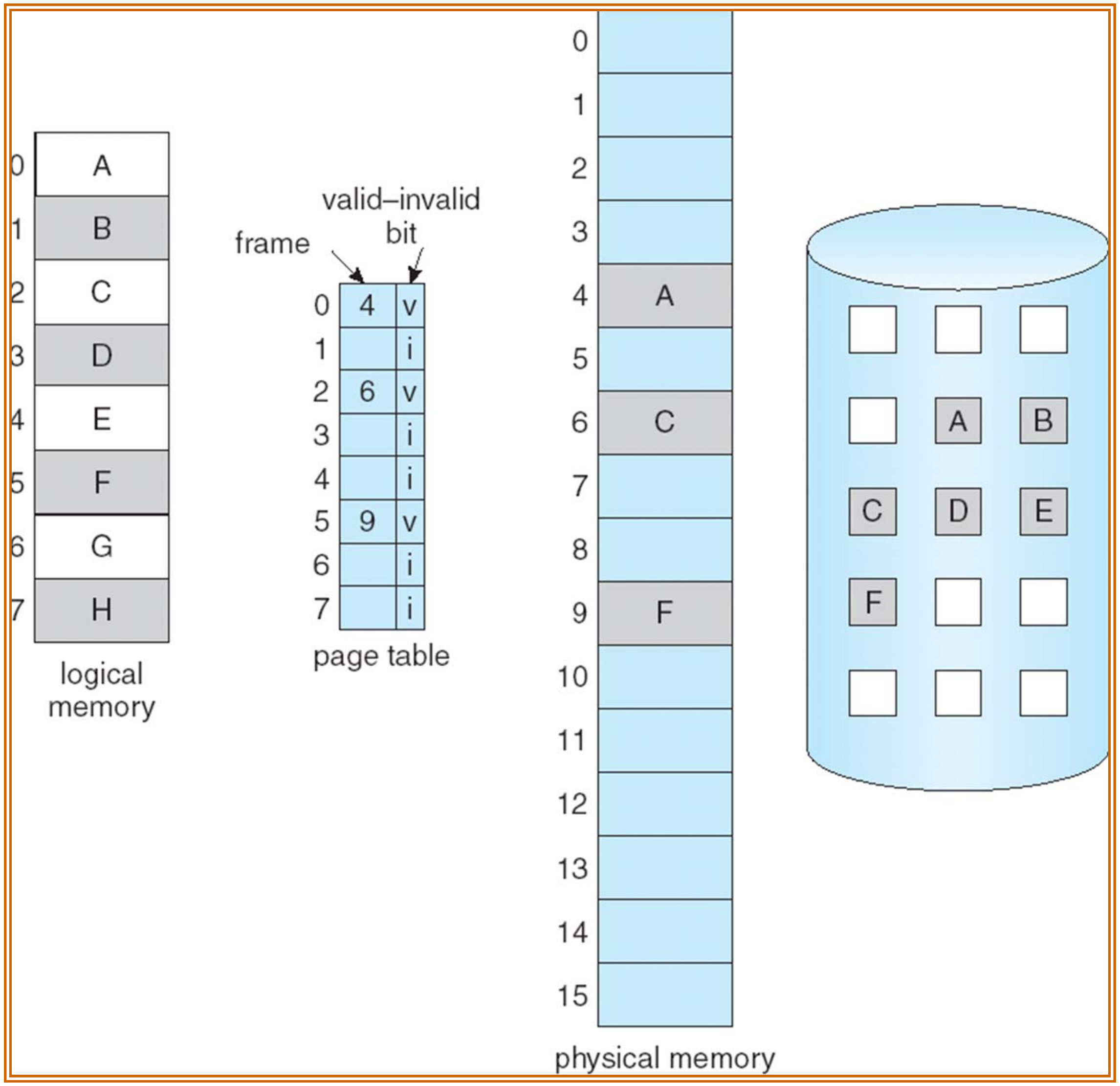
- With each page table entry a valid–invalid bit is associated
  (1 ⇒ in-memory, 0 ⇒ not-in-memory)
- Initially valid–invalid but is set to 0 on all entries
- Example of a page table snapshot:

Frame #	valid-invalid bit
	1
	1
	1
	1
	0
⋮	
	0
	0

page table

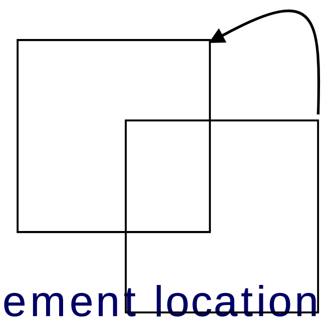
- During address translation, if valid–invalid bit in page table entry is 0 ⇒ page fault

Page Table When Some Pages Are Not in Main Memory



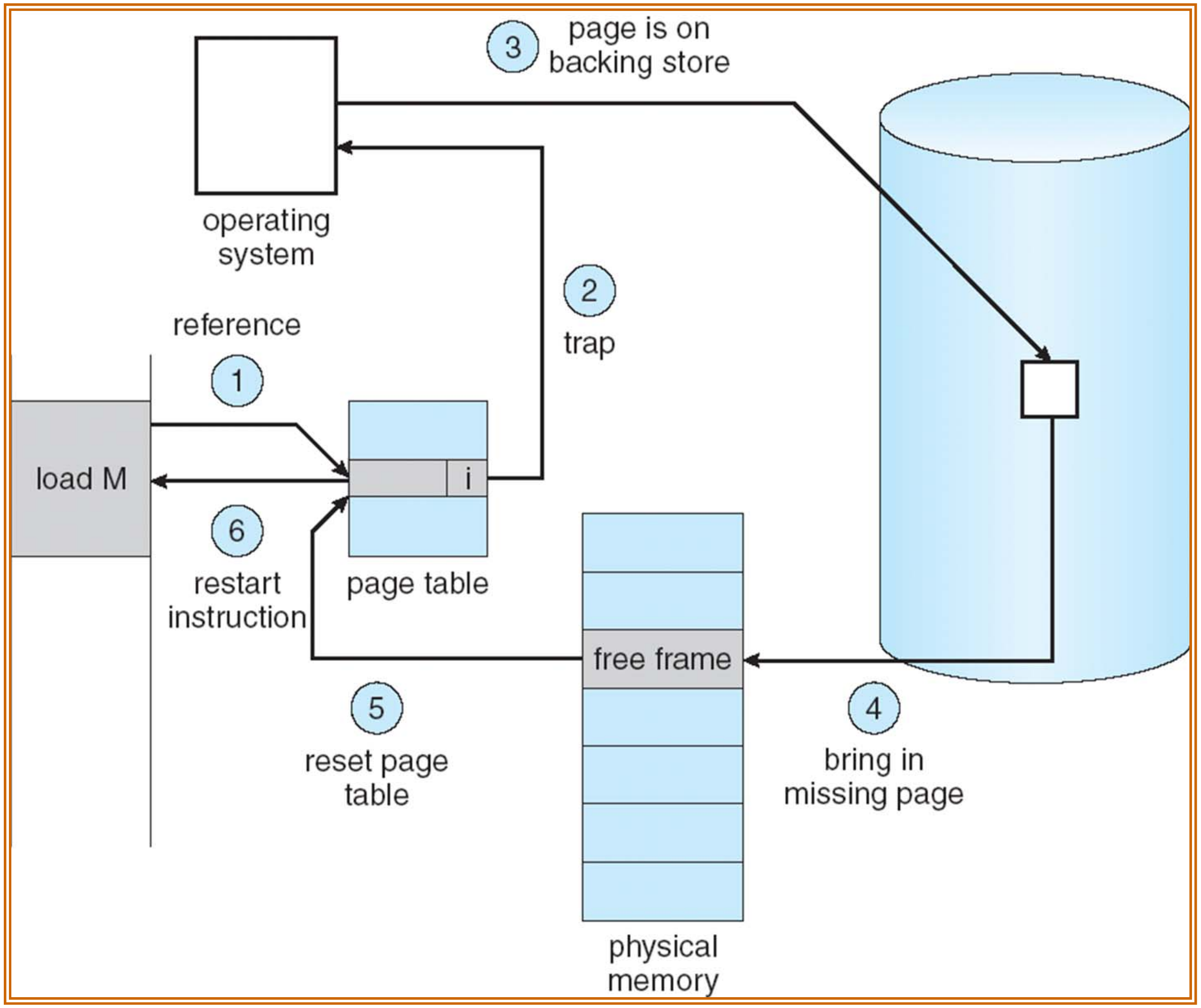
Page Fault

- If there is ever a reference to a page, first reference will trap to
  OS ⇒ page fault
- OS looks at another table to decide:
  - Invalid reference ⇒ abort.
  - Just not in memory.
- Get empty frame.
- Swap page into frame.
- Reset tables, validation bit = 1.
- Restart instruction: Least Recently Used
  - block move



  - auto increment/decrement location

Steps in Handling a Page Fault



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

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)

$$
\begin{aligned}
\text{EAT} = (1 - p) \text{ x memory access} \\
&+ p \text{ (page fault overhead} \\
&+ \text{[swap page out]} \\
&+ \text{swap page in} \\
&+ \text{restart overhead)}
\end{aligned}
$$

Demand Paging Example

- Memory access time = 1 microsecond
- 50% of the time the page that is being replaced has been modified and therefore needs to be swapped out
- Swap Page Time = 10 msec = 10,000 msec

$$EAT = (1 – p) \times 1 + p\,(15000)$$

$$1 + 15000P \qquad \text{(in msec)}$$

Process Creation

- Virtual memory allows other benefits during process creation:

  - Copy-on-Write

  - Memory-Mapped Files (later)

Copy-on-Write

- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory

  If either process modifies a shared page, only then is the page copied

- COW allows more efficient process creation as only modified pages are copied

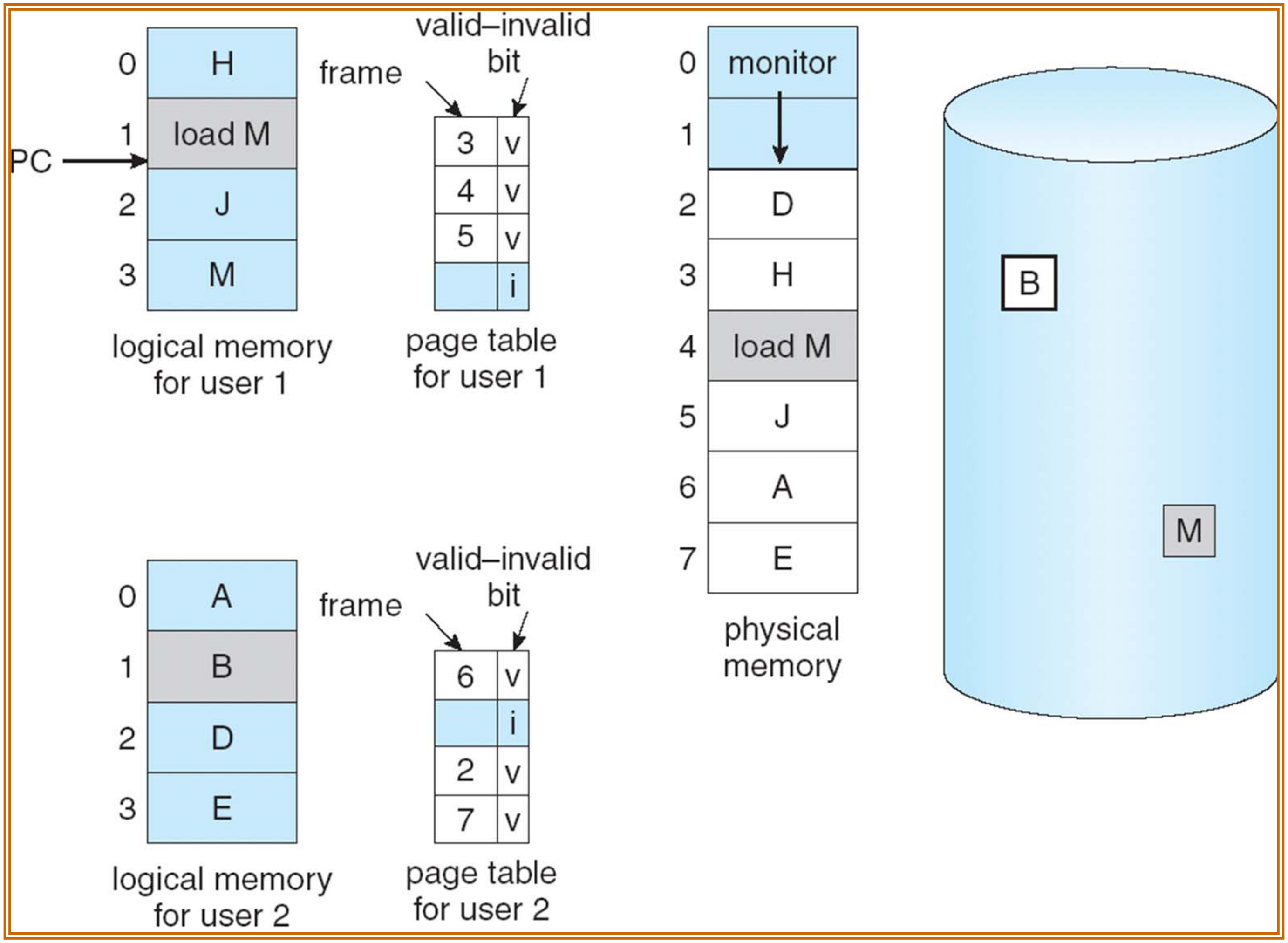
- Free pages are allocated from a **pool** of zeroed-out pages

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

Bring M from Memory -> Replace B



Basic Page Replacement

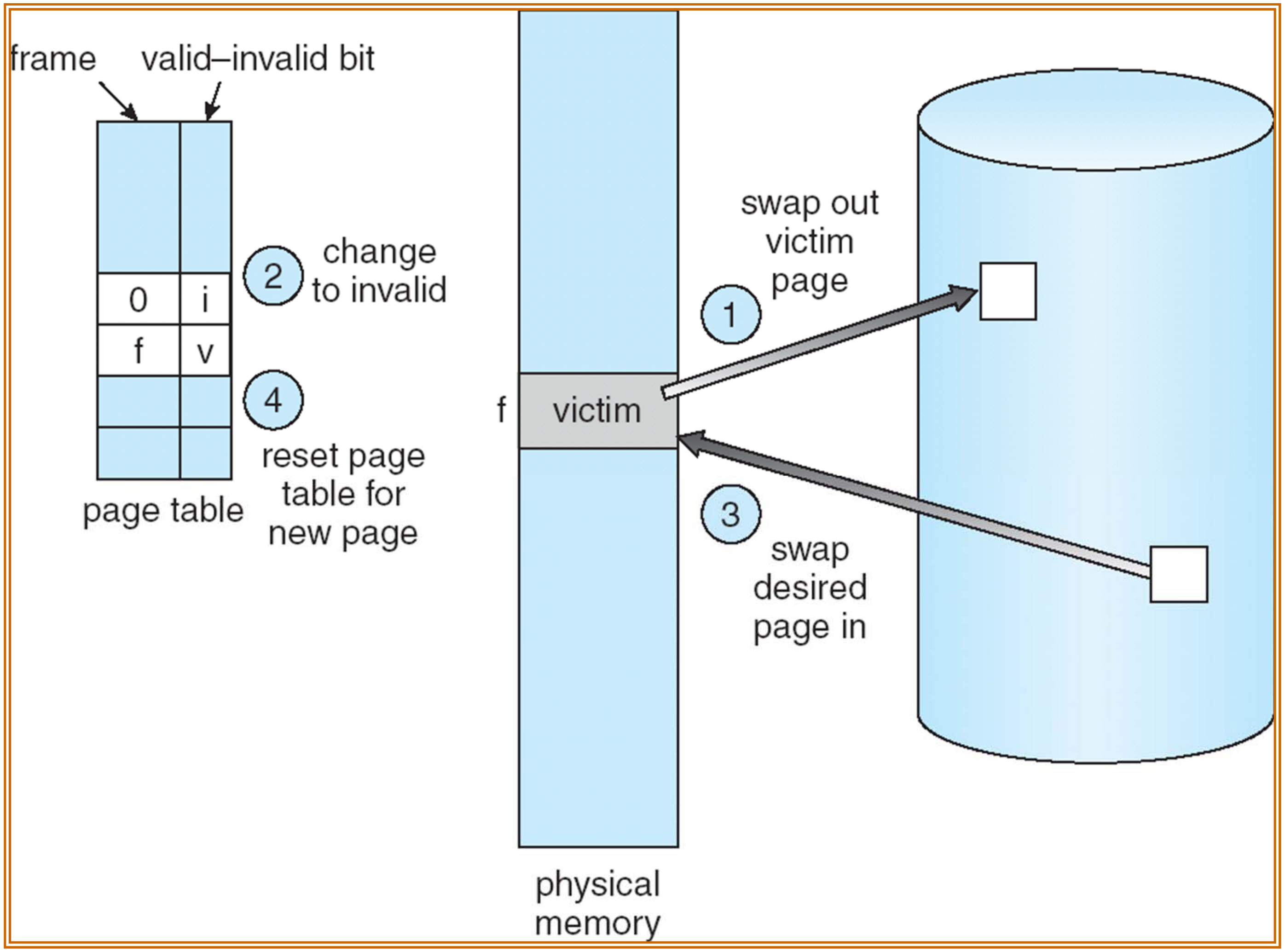
1. Find the location of the desired page on disk

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. Read the desired page into the (newly) free frame. Update the page and frame tables.

4. Restart the process

Page Replacement

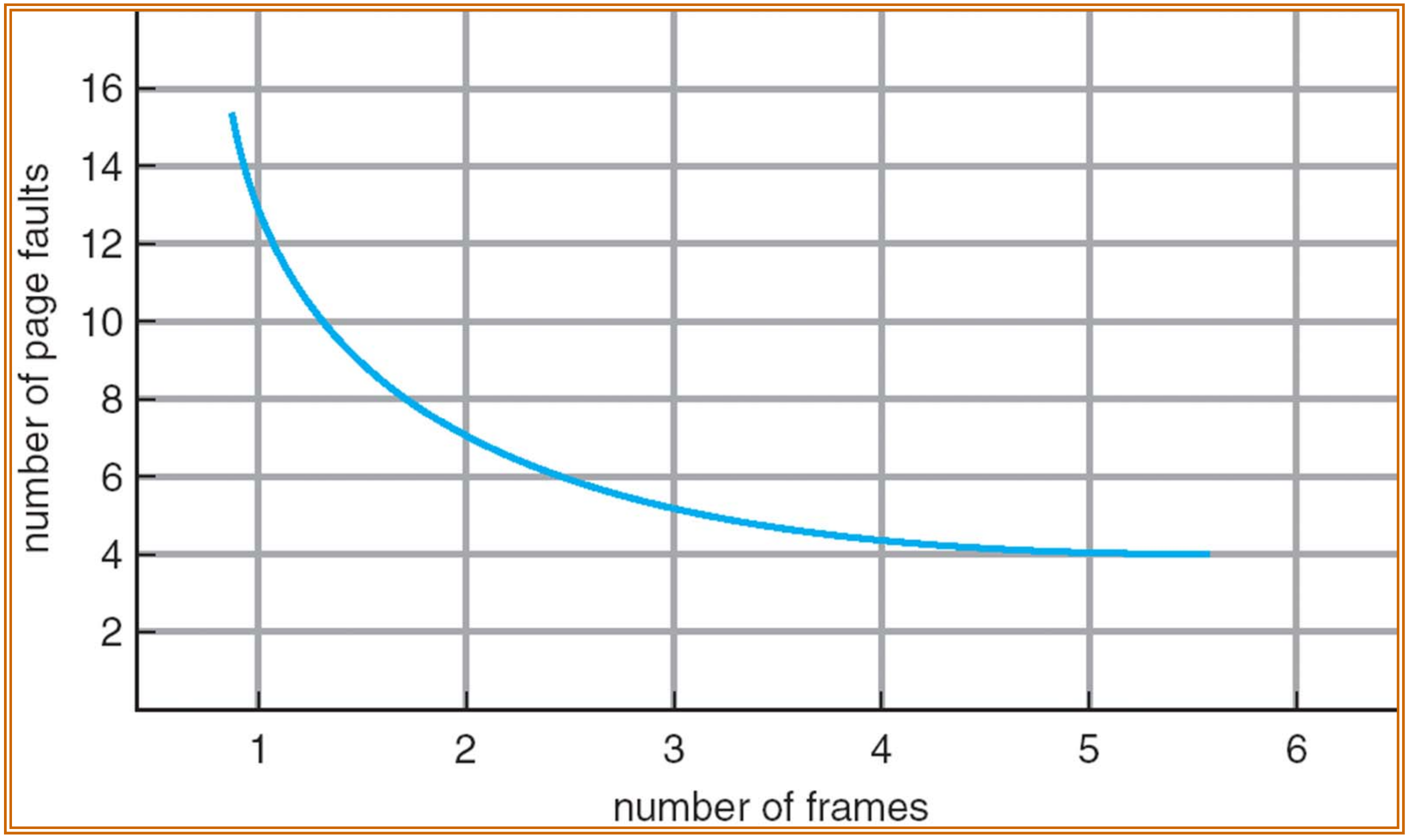


Page Replacement Algorithms

- 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

Graph of Page Faults Versus The Number of Frames



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)

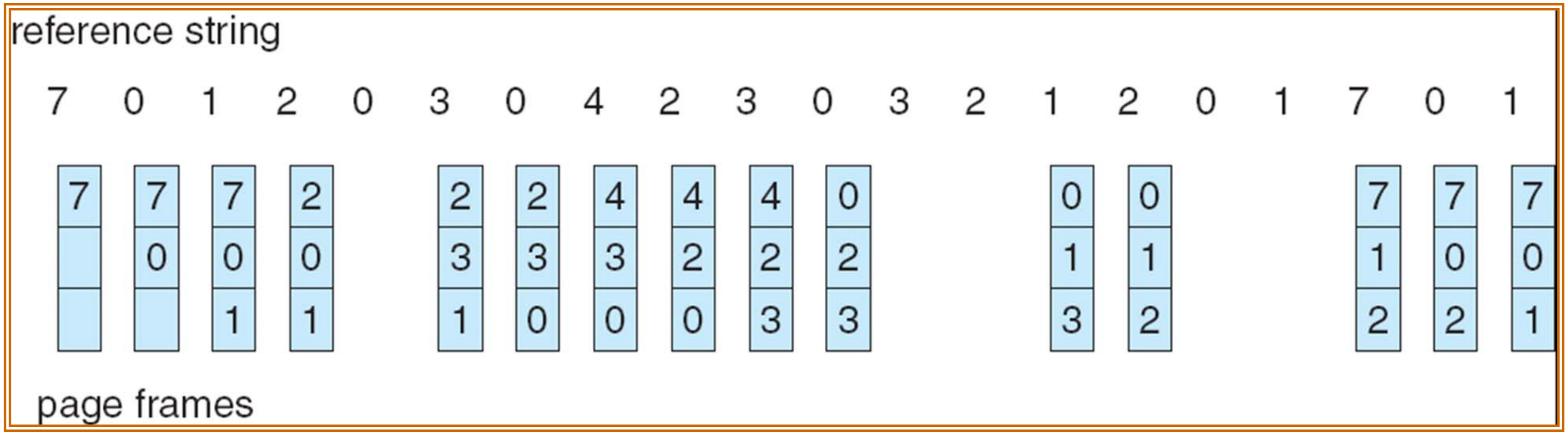
1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

- 4 frames

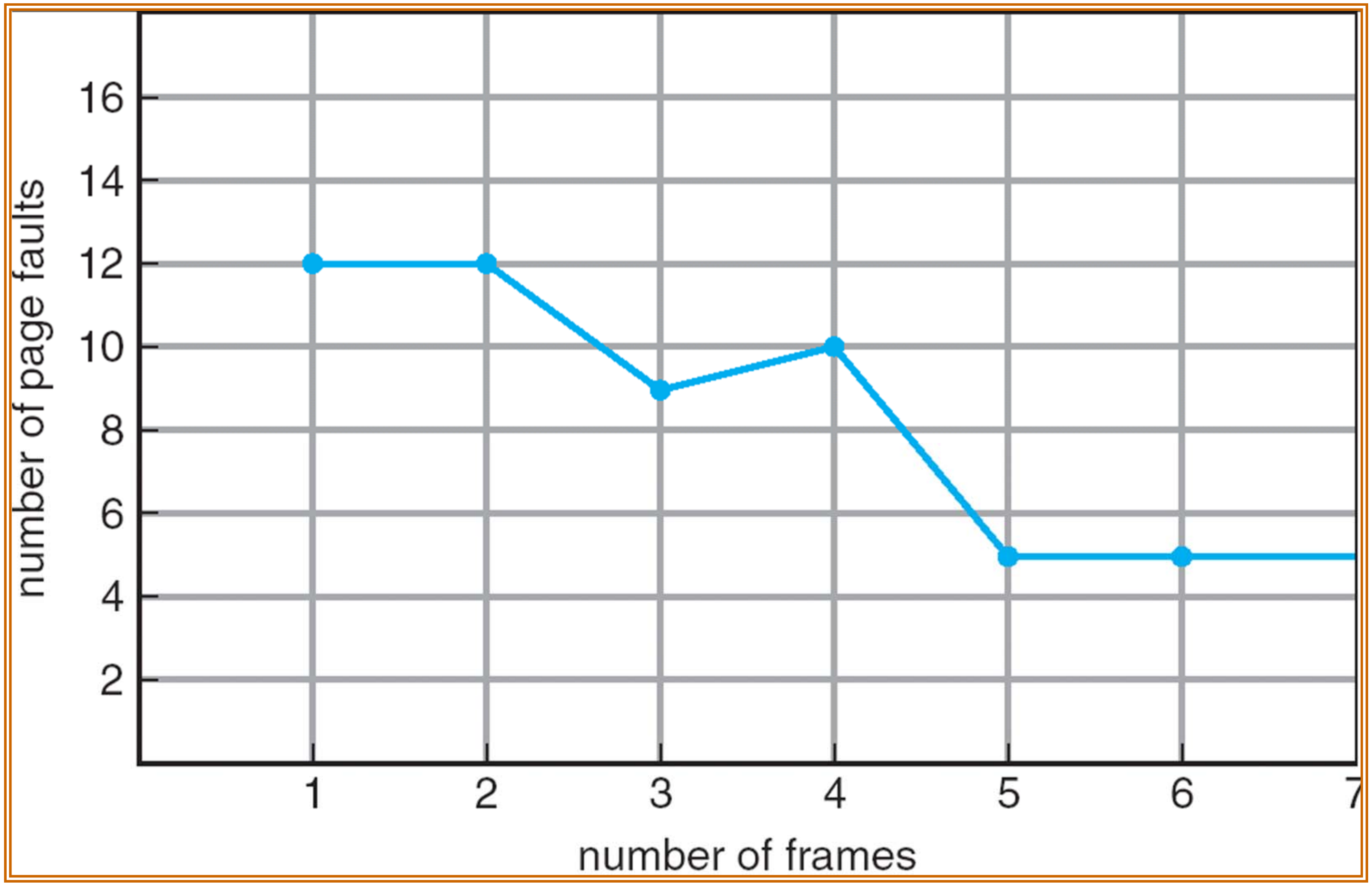
1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		

- FIFO Replacement – Belady's Anomaly
  - more frames leads to more page faults

FIFO Page Replacement



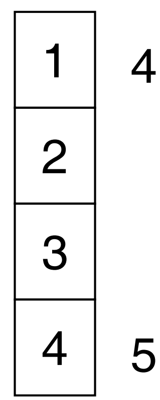
FIFO Illustrating Belady's Anomaly



Optimal Algorithm

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

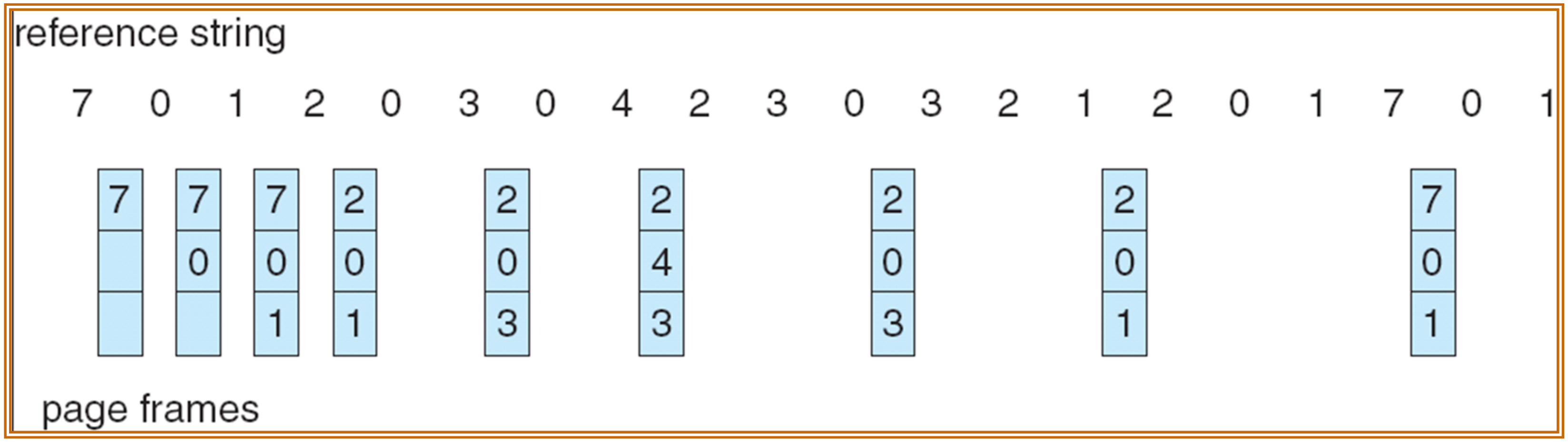
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



6 page faults

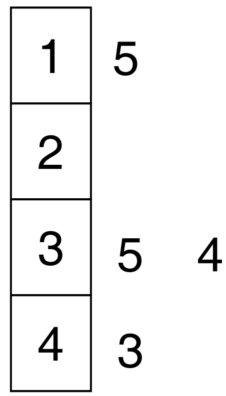
- How do you know this?
- Used for measuring how well your algorithm performs

Optimal Page Replacement



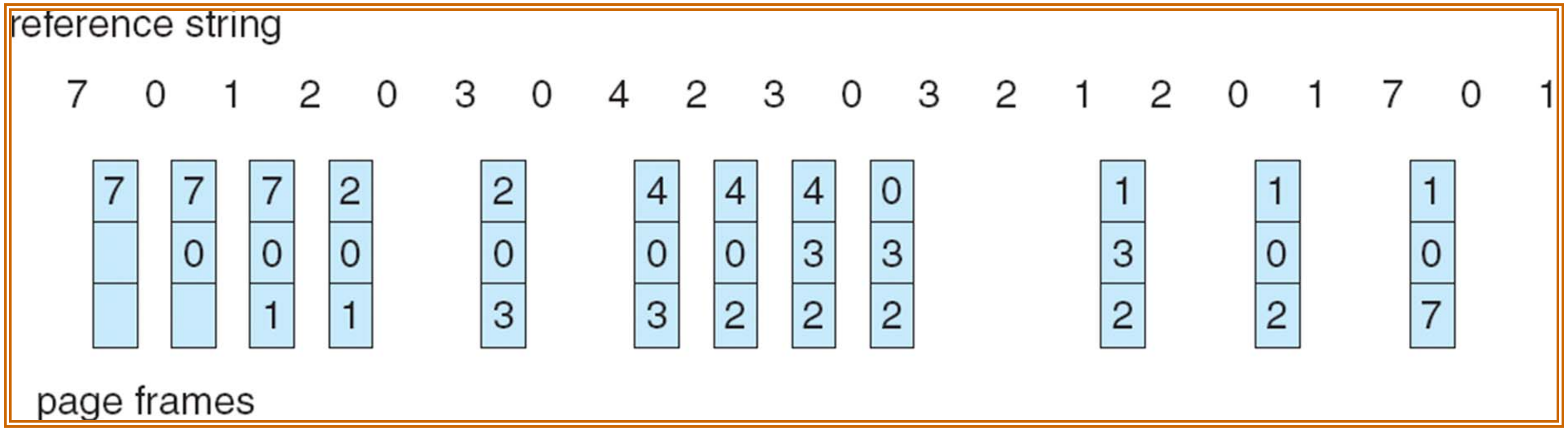
Least Recently Used (LRU) Algorithm

- Replace the page that has used least recently
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

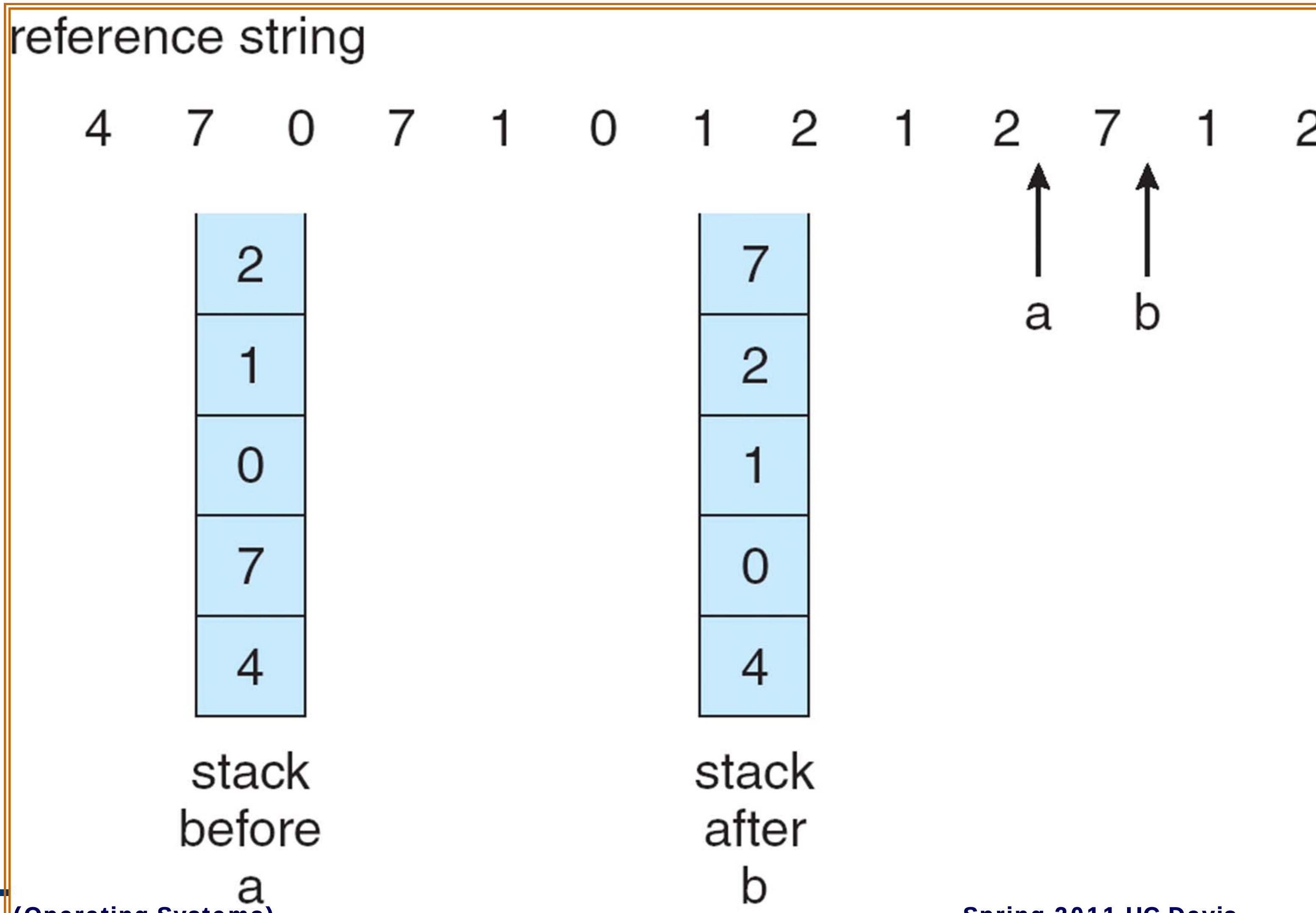


- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to determine which are to change
- Stack implementation – keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement

LRU Page Replacement



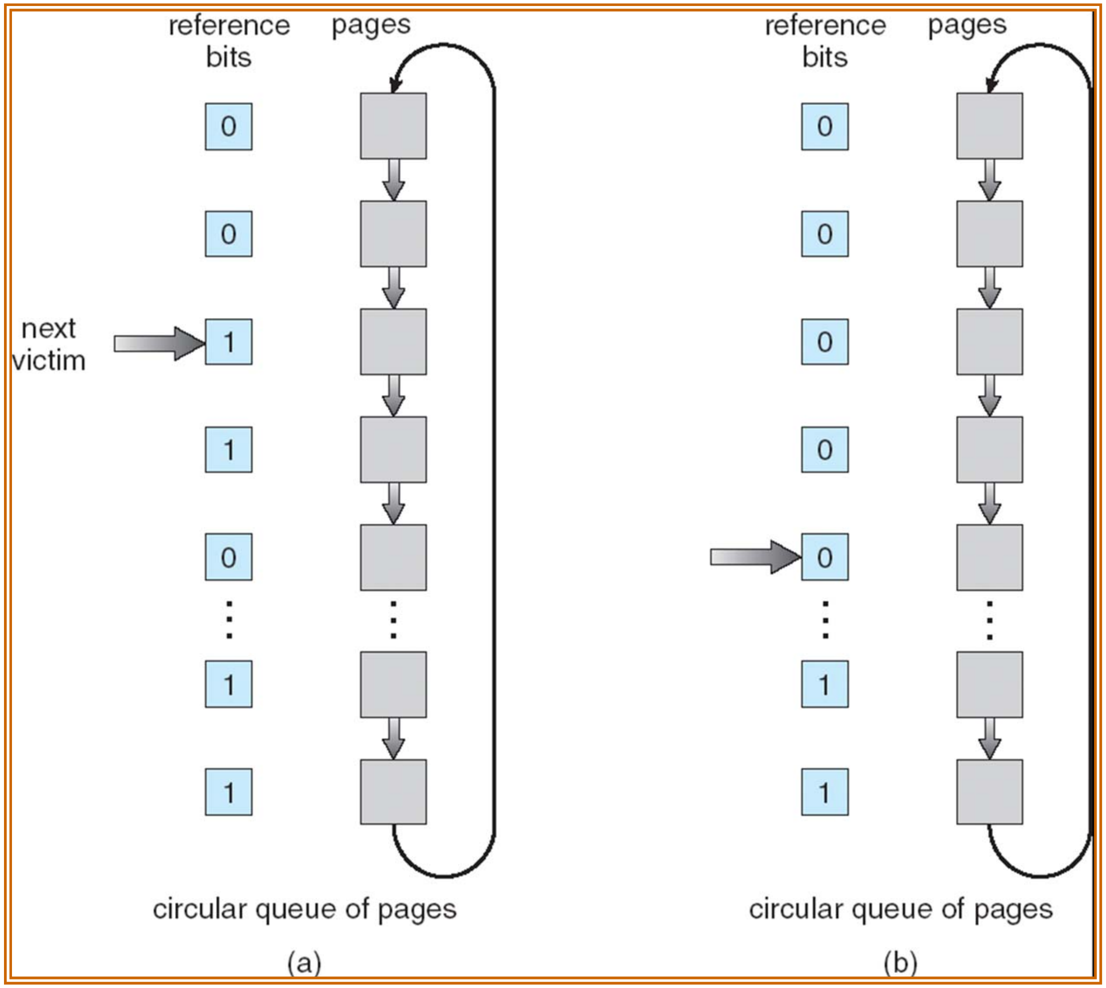
Use Of A Stack to Record The Most Recent Page References



LRU Approximation Algorithms

- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace the one which is 0 (if one exists). We do not know the order, however.
- Second chance
  - Need reference bit
  - Clock replacement
  - If page to be replaced (in clock order) has reference bit = 1 then:
    - set reference bit 0
    - leave page in memory
    - replace next page (in clock order), subject to same rules

Second-Chance (clock) Page-Replacement Algorithm



Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- **LFU Algorithm**: replaces page with smallest count
- **MFU Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Allocation of Frames

- Problem:
  - Given a set of frames and processes, how does one allocate frames to pages?
- Each process needs *minimum* number of pages
- Two major allocation schemes
  - fixed allocation
  - priority allocation

Fixed Allocation

- Equal allocation – e.g., if 100 frames and 5 processes, give each 20 pages
- 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

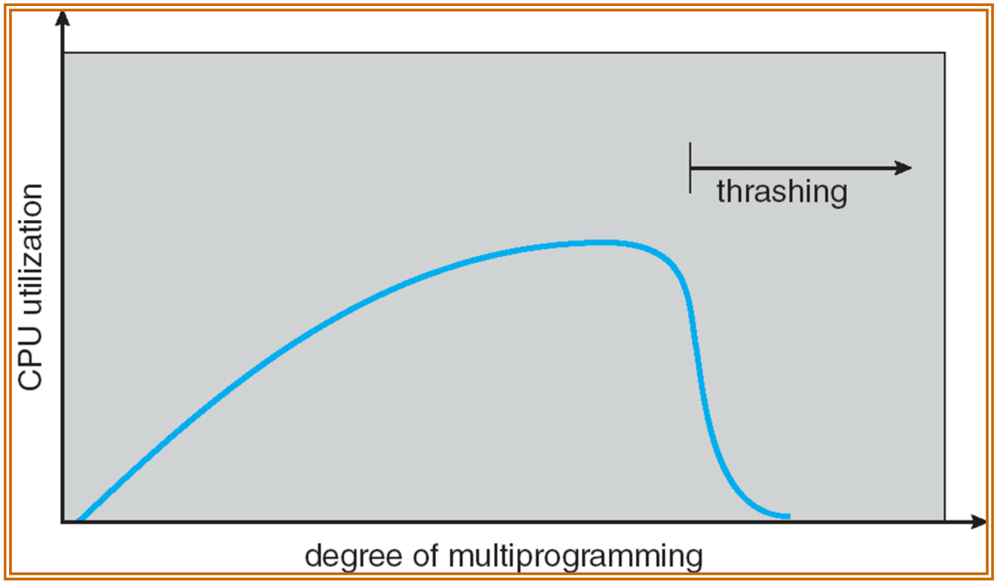
Global vs. Local Allocation

- **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

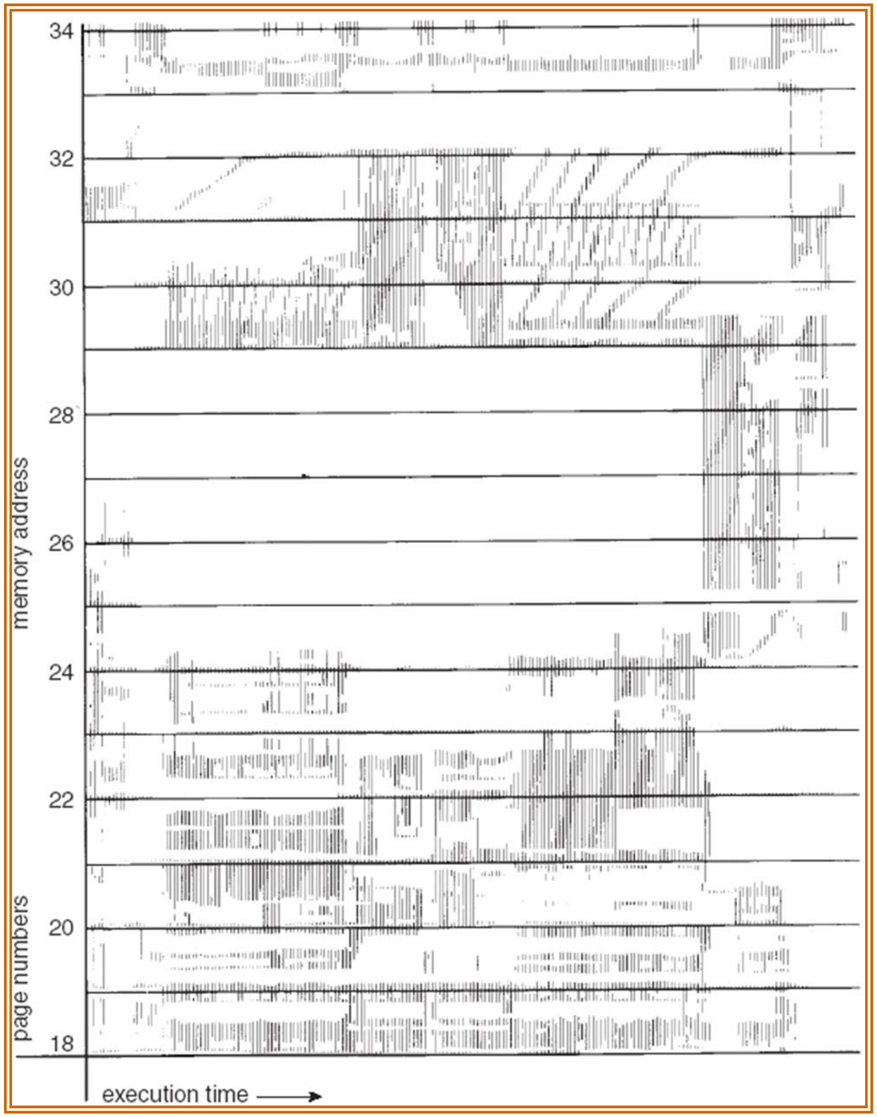
- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system
- **Thrashing** ≡ a process is busy swapping pages in and out

Thrashing



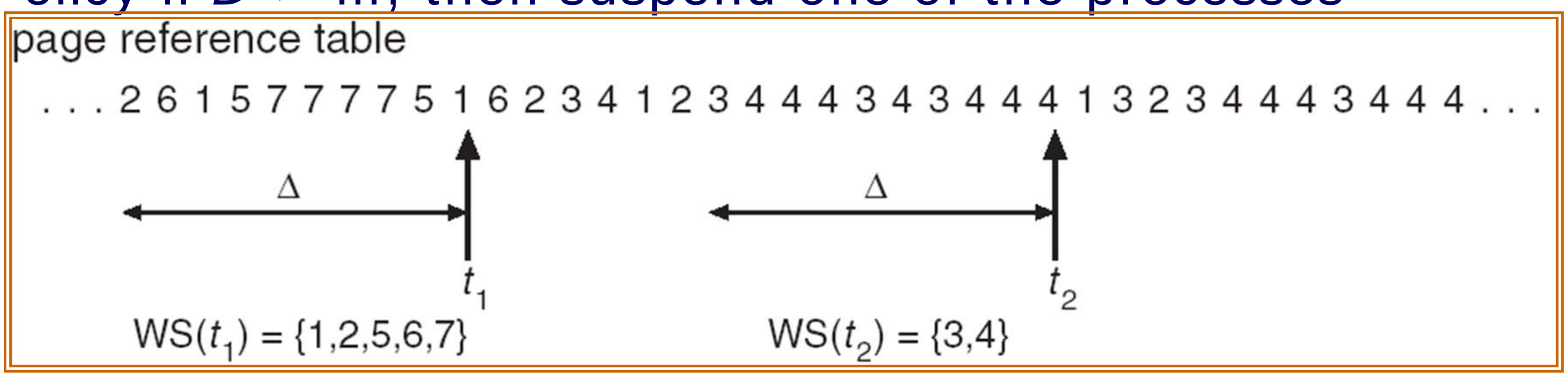
- Why does paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?
  Σ size of locality > total memory size

Locality In A Memory-Reference Pattern



Working-Set Model

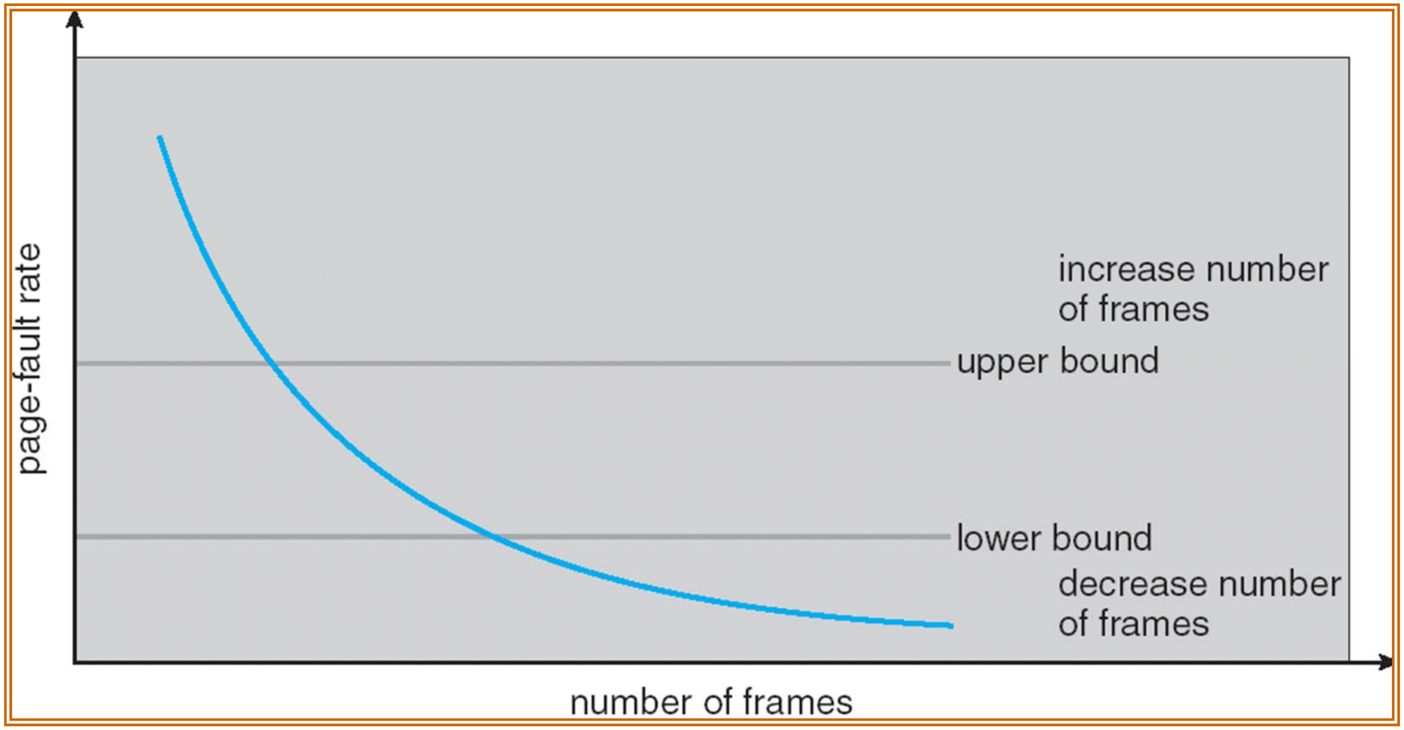
- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instruction
- WSS_i (working set of Process P_i) =
  total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\ WSS_i \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing (m = available frames)
- Policy if $D >$ m, then suspend one of the processes



Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10{,}000$
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

Page-Fault Frequency Scheme

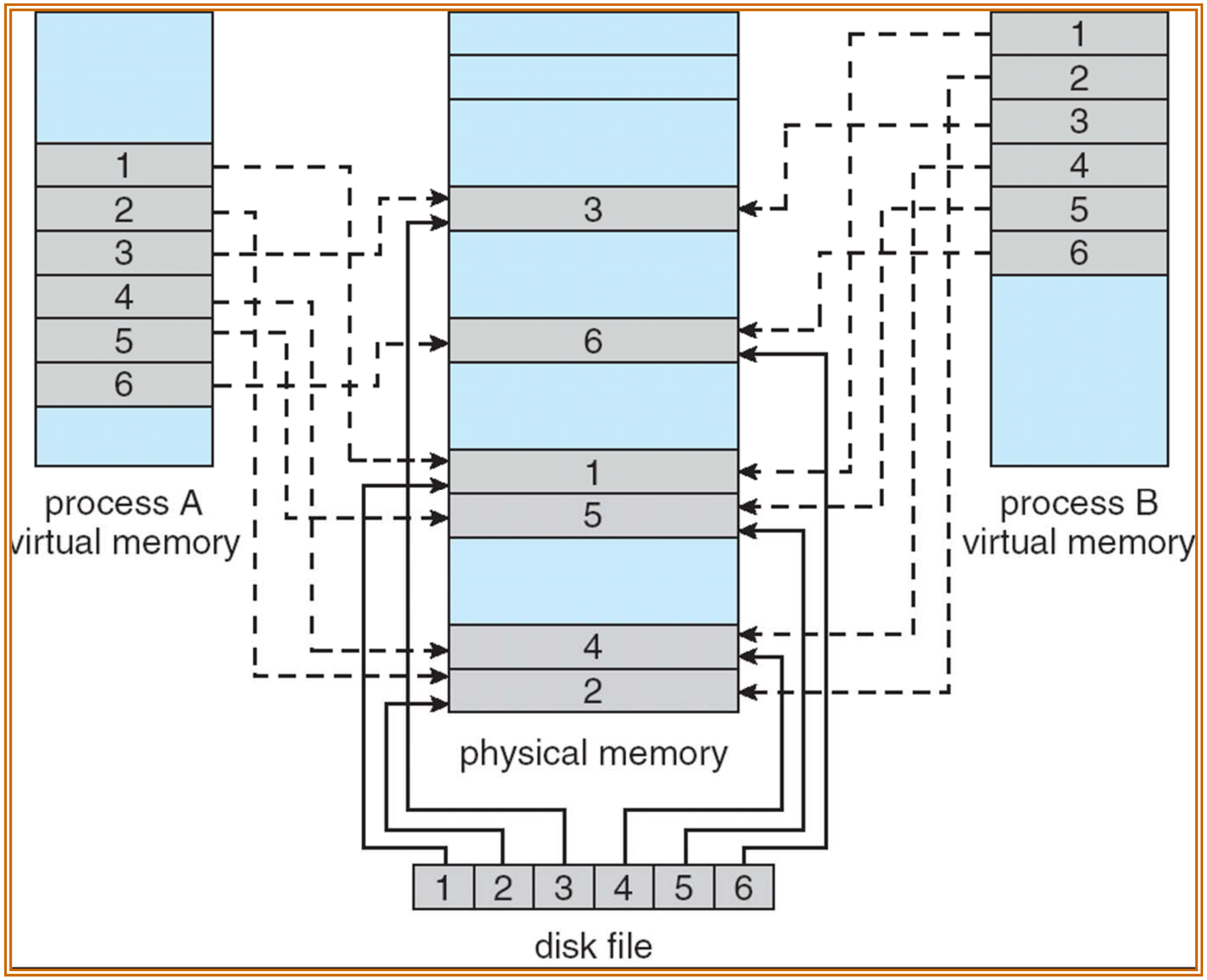


- Establish “acceptable” page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame

Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by *mapping* a disk block to a page in memory
- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
- Simplifies file access by treating file I/O through memory rather than **read() write()** system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared

Memory Mapped Files



Other Issues

- Prepaging
  - To reduce the large number of page faults that occurs at process startup
  - Prepage all or some of the pages a process will need, before they are referenced
  - But if prepaged pages are unused, I/O and memory was wasted
  - Assume s pages are prepaged and α of the pages is used
    - o Is cost of $s * \alpha$ save pages faults > or < than the cost of prepaging $s * (1- \alpha)$ unnecessary pages?
    - o α near zero $\Rightarrow$ prepaging loses
- Page size selection must take into consideration:
  - Fragmentation
    - o Smaller the size, better the utilization
  - table size:
    - o Smaller the page size, larger the page table size
  - I/O overhead
    - o Larger the page, longer it takes to load the page, however latency time and seek time dominate the overall time.
  - Locality
    - o Larger page size => lesser # of page faults

Other Issues (Cont.)

- **TLB Reach** - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in the TLB. Otherwise there is a high degree of page faults.

Other Issues (Cont.)

- **Increase the Page Size**. This may lead to an increase in fragmentation as not all applications require a large page size.

- **Provide Multiple Page Sizes**. This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation.

Other Issues (Cont.)

- Program structure
  - **int A[][] = new int[1024][1024];**
  - Each row is stored in one page
  - Program 1

```
for (j = 0; j < A.length; j++)
   for (i = 0; i < A.length; i++)
         A[i,j] = 0;
```

1024 x 1024 page faults

  - Program 2

```
for (i = 0; i < A.length; i++)
   for (j = 0; j < A.length; j++)
         A[i,j] = 0;
```

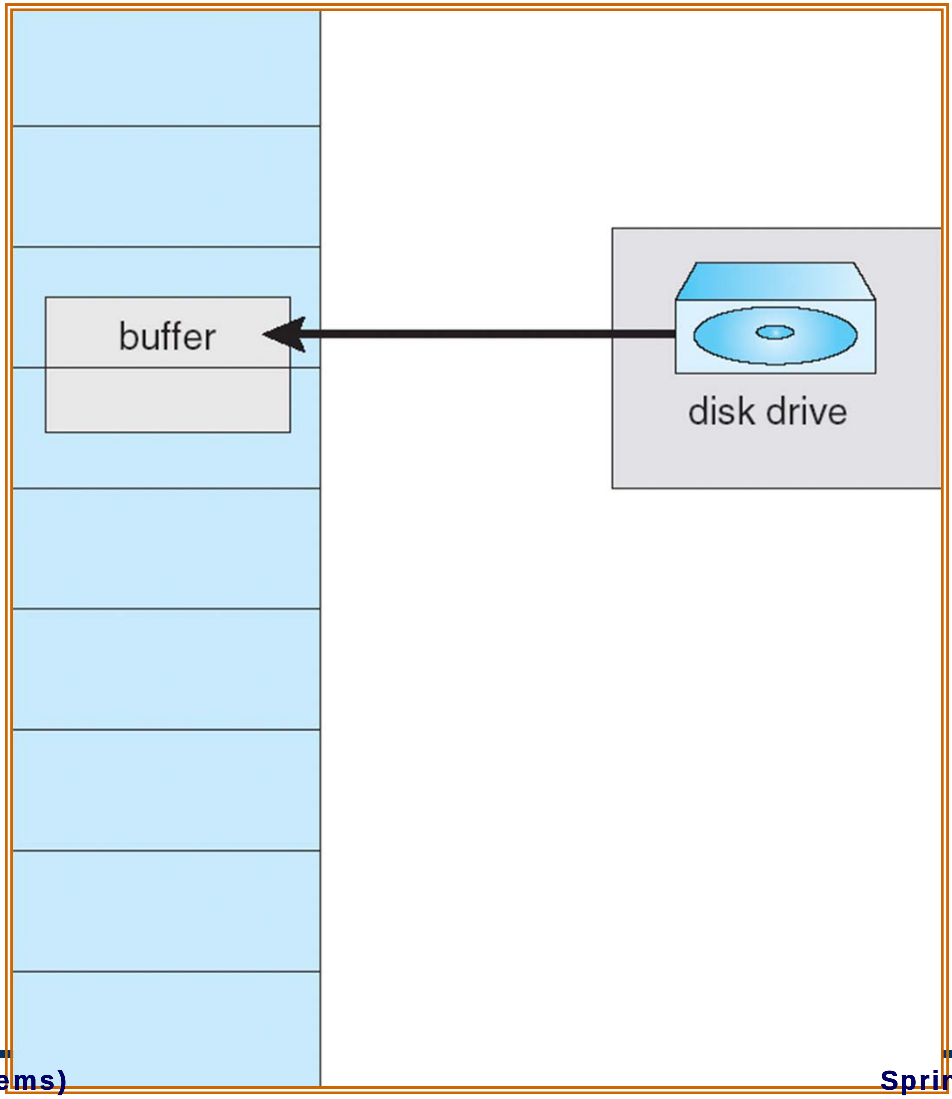
1024 page faults

Other Considerations (Cont.)

- **I/O Interlock** – Pages must sometimes be locked into memory

- Consider I/O. Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm.

Reason Why Frames Used For I/O Must Be In Memory



Demand Segmentation

- Used when insufficient hardware to implement demand paging.
- OS/2 allocates memory in segments, which it keeps track of through segment descriptors
- Segment descriptor contains a valid bit to indicate whether the segment is currently in memory.
  - If segment is in main memory, access continues,
  - If not in memory, segment fault.