Chapter 9: Virtual Memory

Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

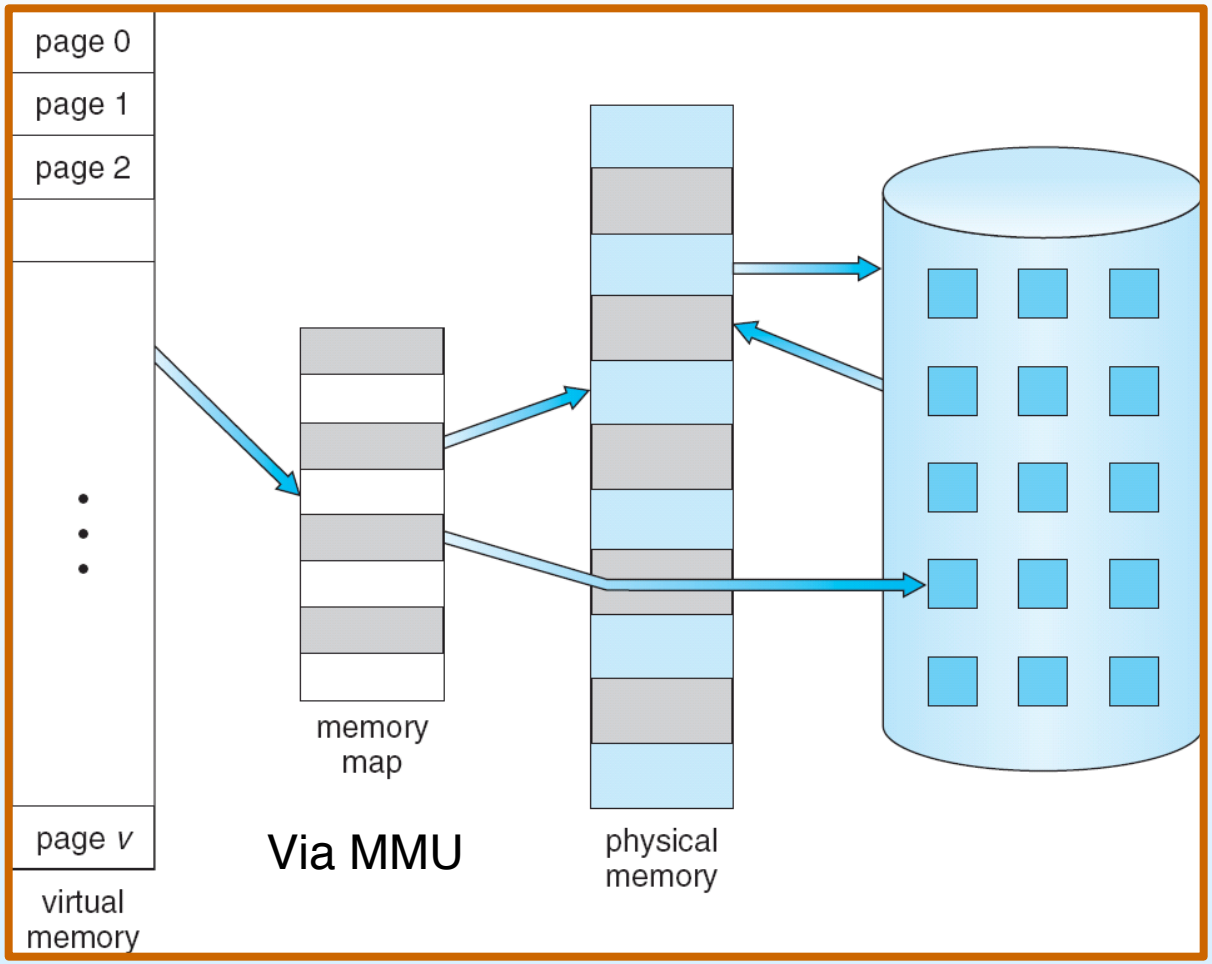
Objectives

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

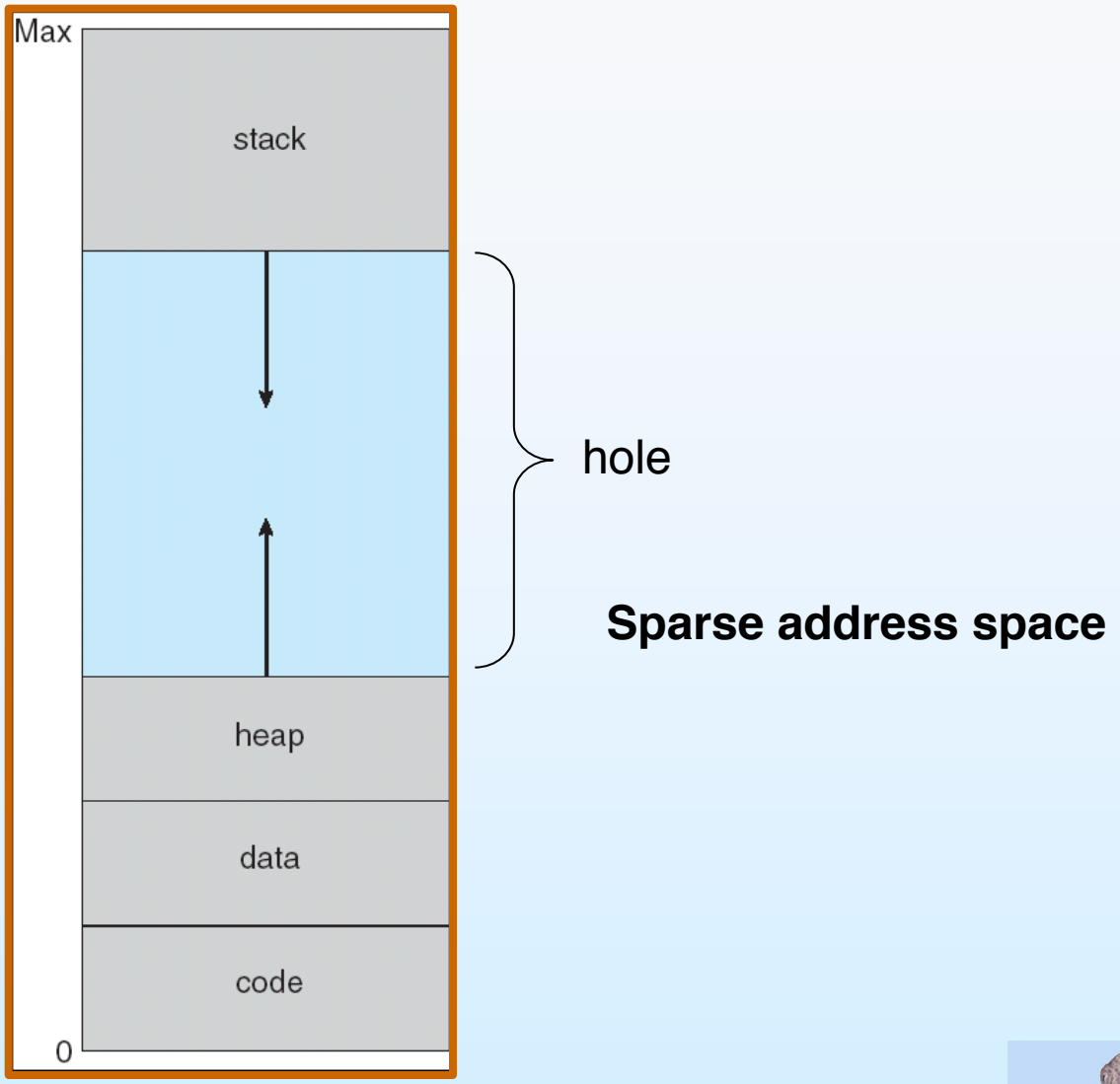
Background

- **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

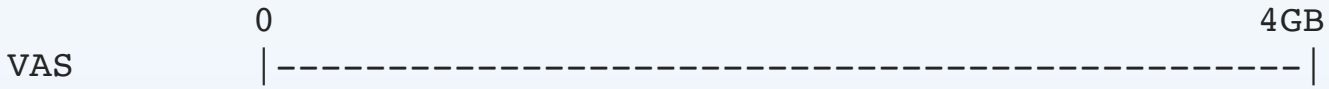


Other benefits

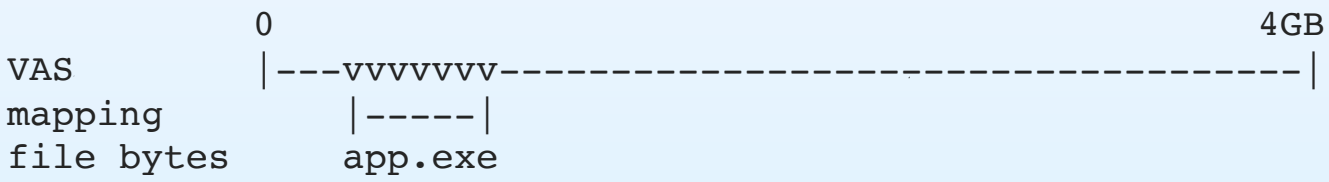
- System libraries can be shared by several processes through mapping of the shared object into a virtual address space
- Shared memory is enabled
- Pages can be shared during process creation (speeds up creation)

Why Virtual Memory

- Consider a 32 bit system, we have a memory space of 4G

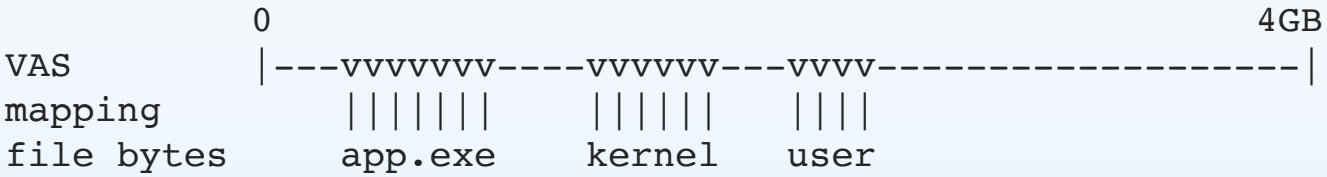


- Load app.exe into memory

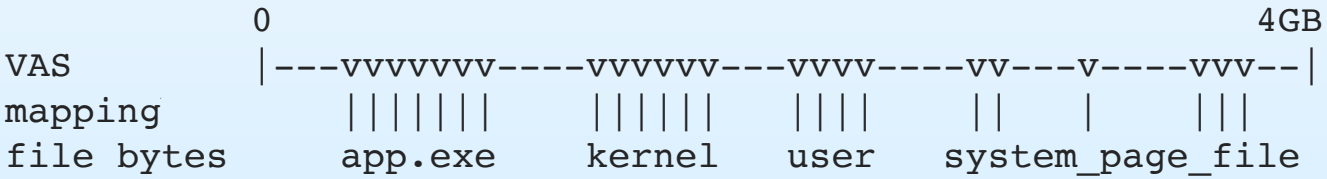


Why Virtual Memory (Continued)

- To run the app.exe, we also need some libraries from the system

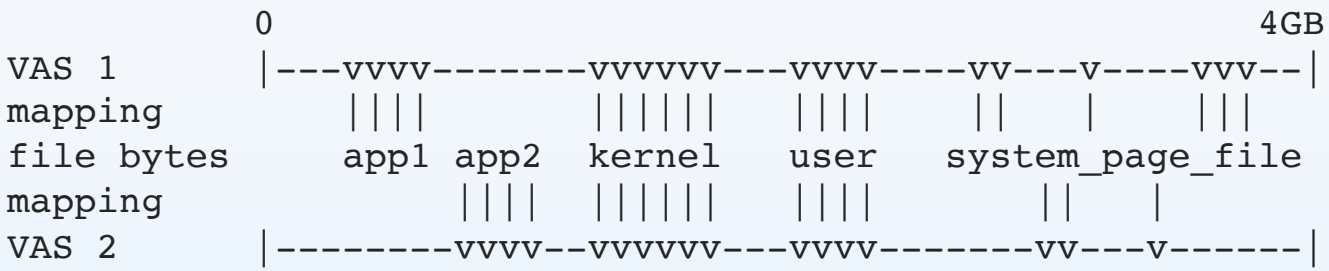


- App.exe requires some spaces to maintain its own data

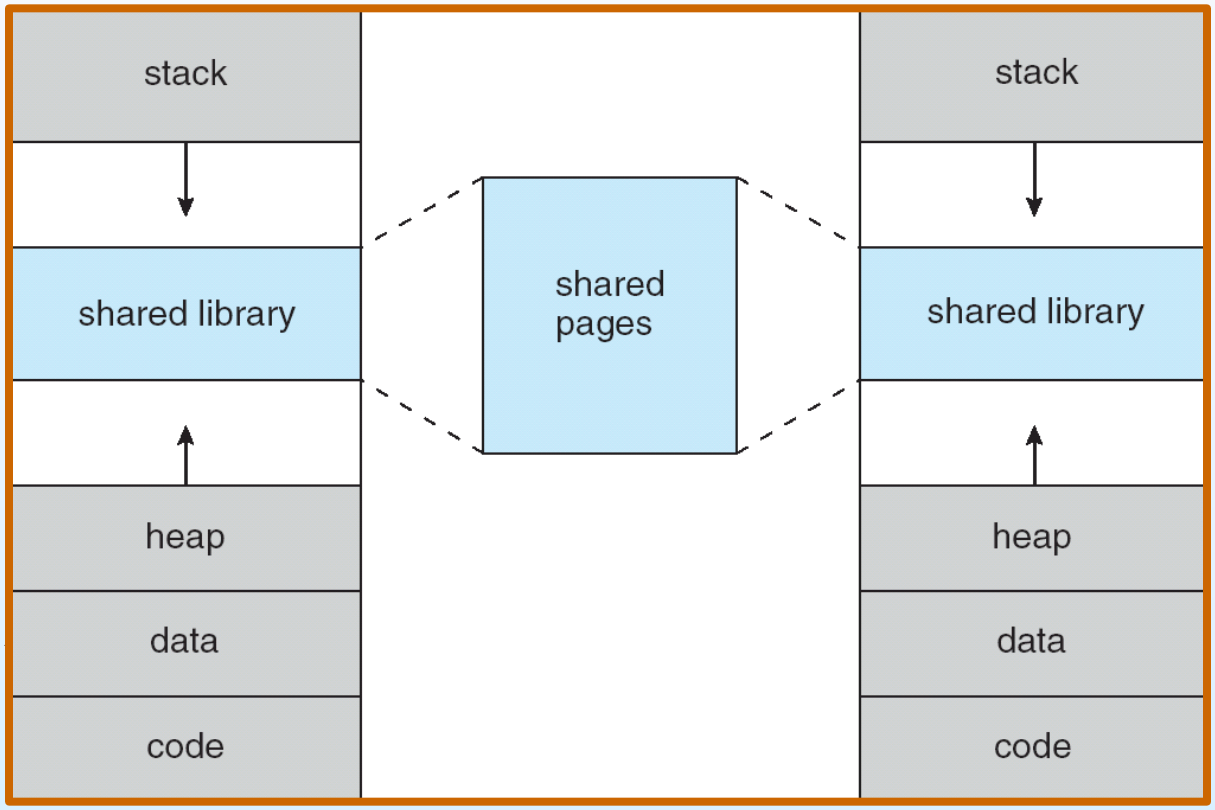


Why Virtual Memory (Continued)

- What if we have more users and more apps



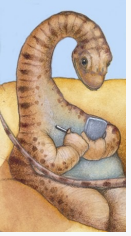
Shared Library Using Virtual Memory



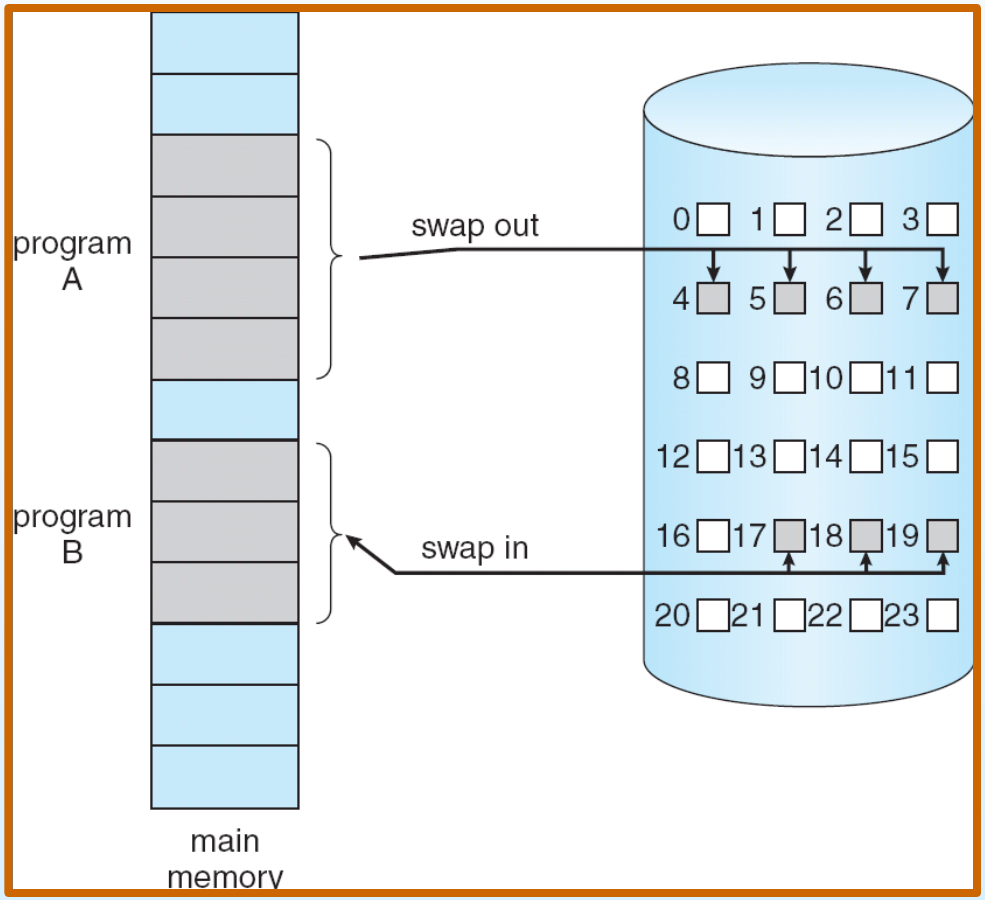
Demand Paging

- 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**

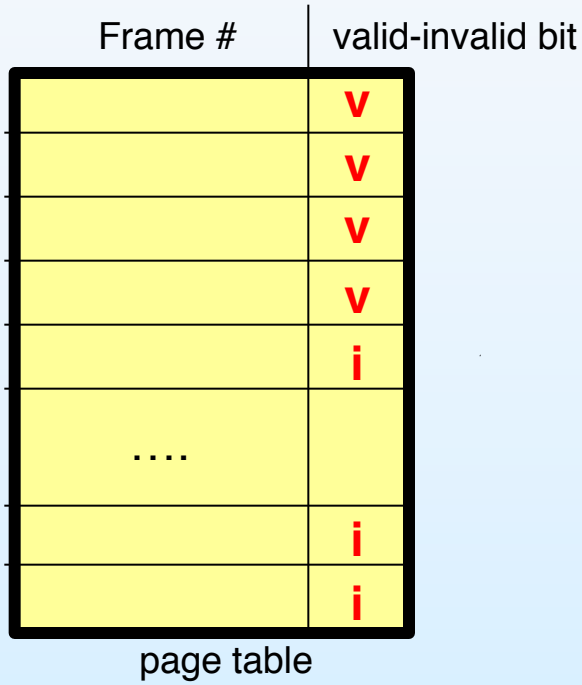


Transfer of a Paged Memory to Contiguous Disk Space



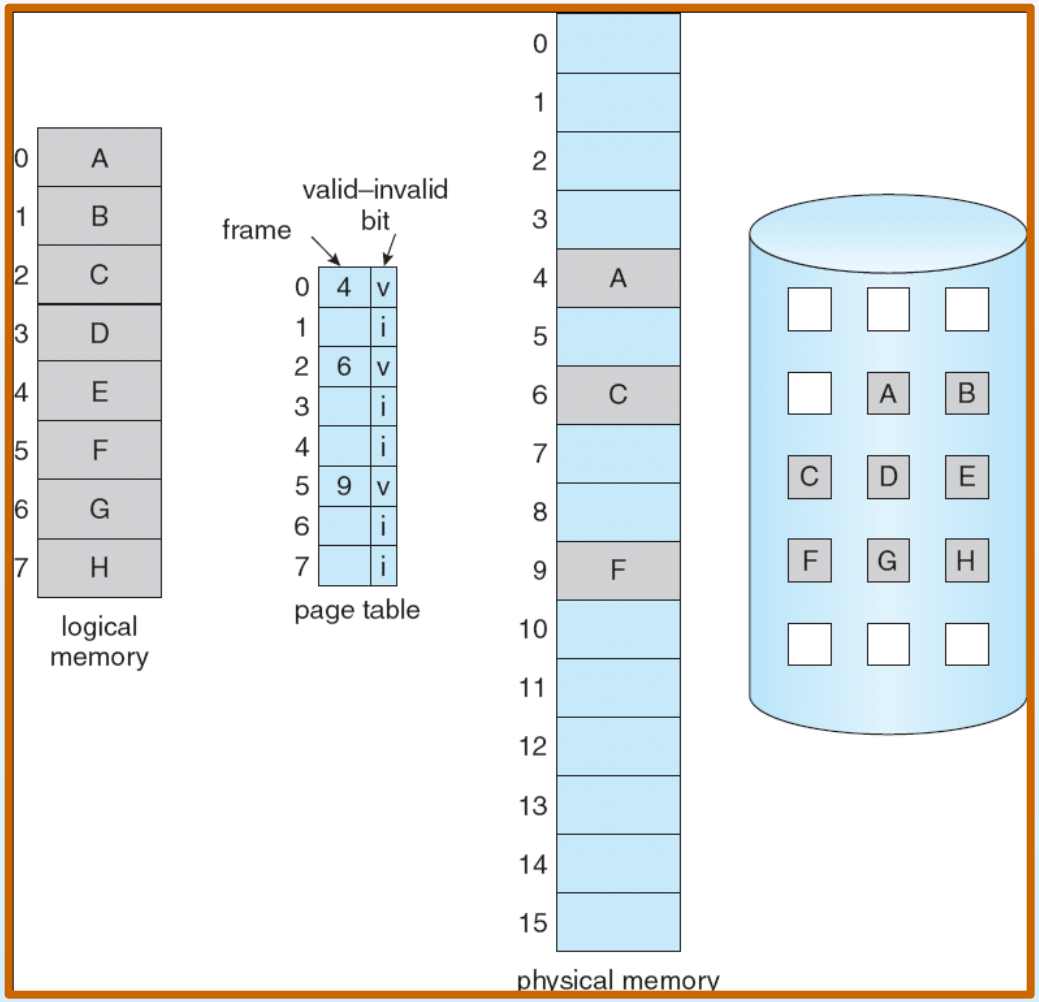
Valid-Invalid Bit

- 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:



- During address translation, if valid–invalid bit in page table entry is **I** ⇒ page fault (a trap to the OS)

Page Table When Some Pages Are Not in Main Memory



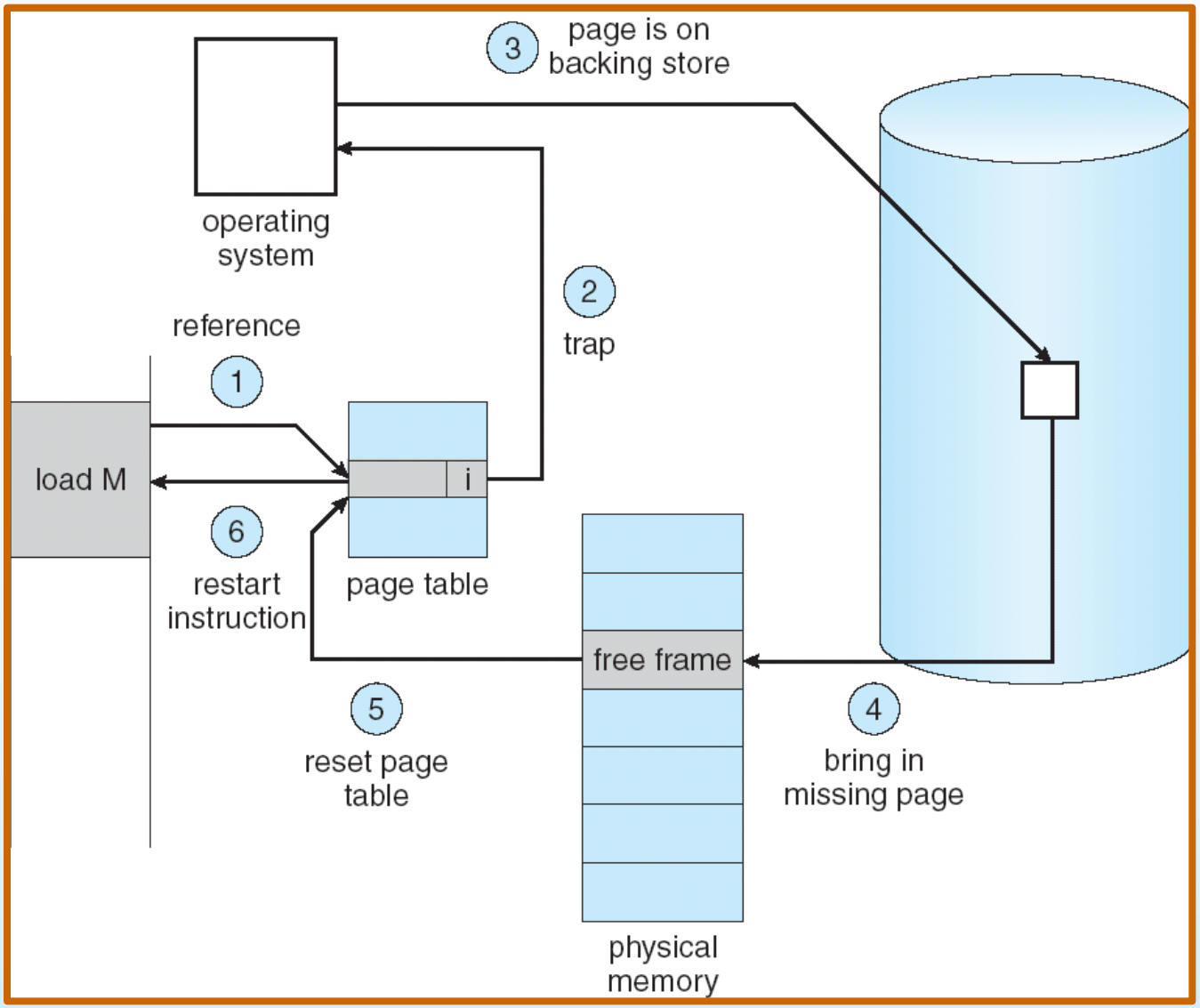
Page Fault

- 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 (kept with PCB) 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)

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

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!!

Process Creation

- Virtual memory allows other benefits during process creation:

  - Copy-on-Write

  - Memory-Mapped Files (later)

Copy-on-Write

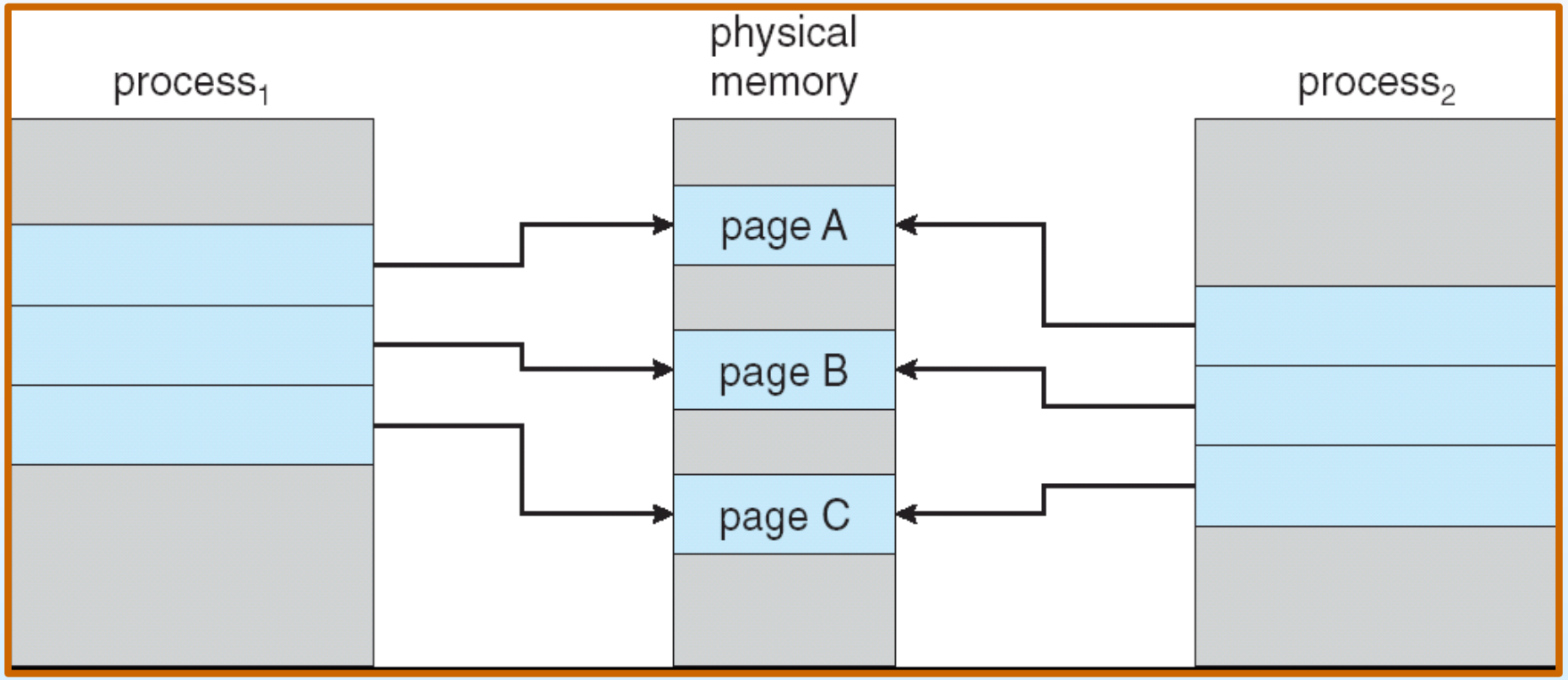
- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory

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

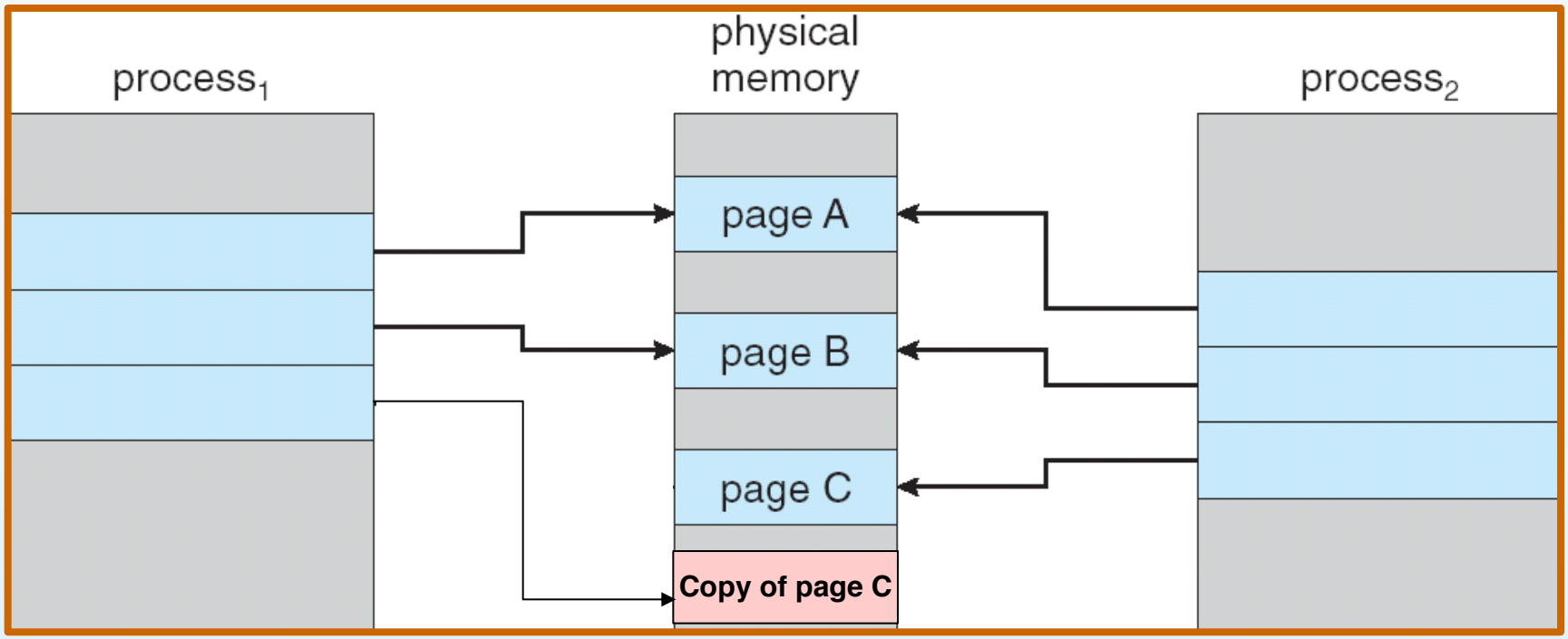
- COW allows more efficient process creation as only modified pages are copied

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

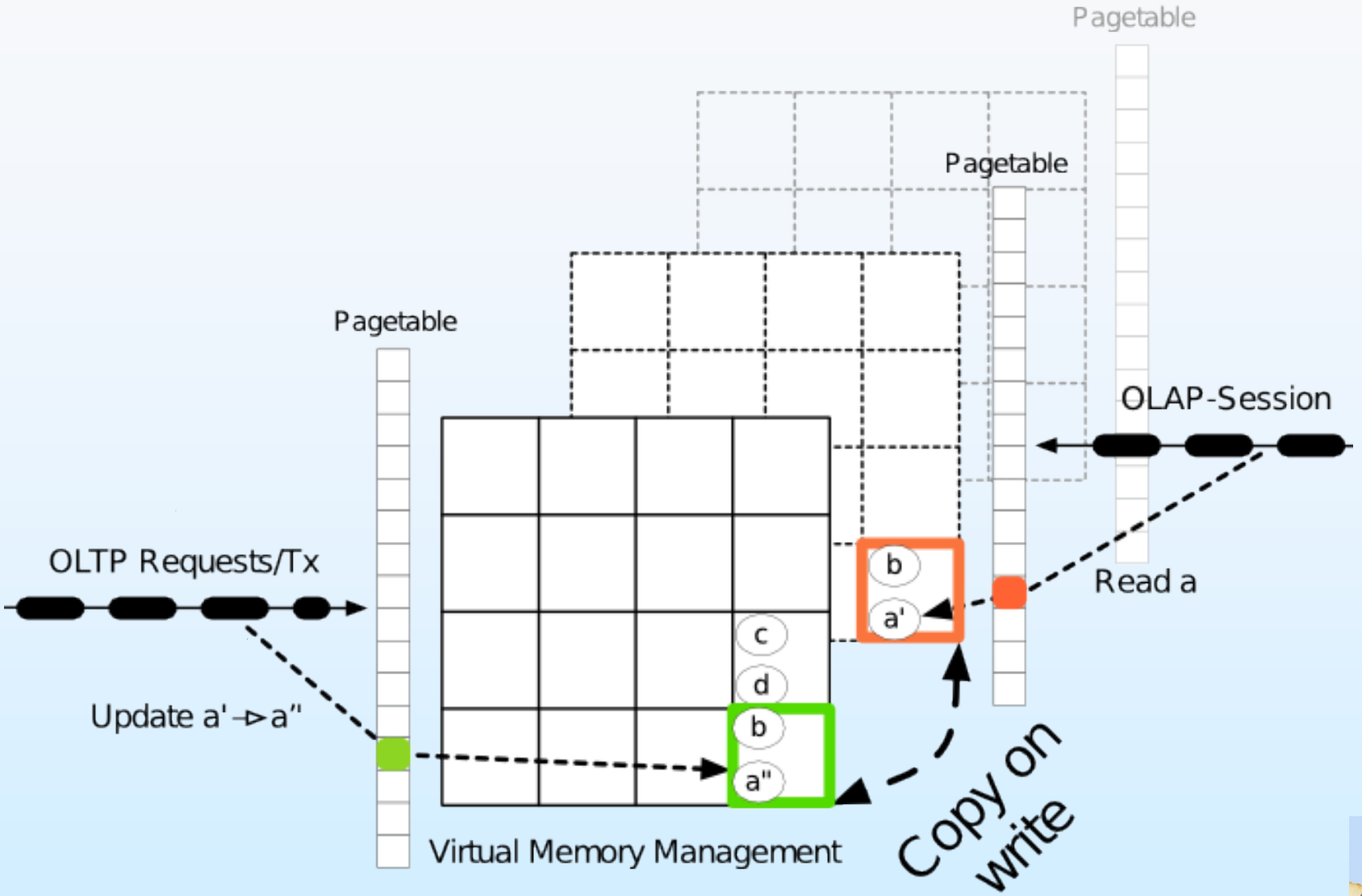
Before Process 1 Modifies Page C



After Process 1 Modifies Page C



Copy on Write in Research (Hyper DB)



Copy on Write in Research (Hyper DB)

VLDB 2016	**How Good Are Query Optimizers, Really?** Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, Thomas Neumann, 2016.	
arXiv preprint	**High-Speed Query Processing over High-Speed Networks** Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, Thomas Neumann, 2015.	pdf
FGDB 2015	**Efficient Integration of Data and Graph Mining Algorithms in Relational Database Systems** Manuel Then, Linnea Passing, Nina Hubig, Stephan Günnemann, Alfons Kemper, Thomas Neumann, 2015.	
VLDB 2015	**Efficient Processing of Window Functions in Analytical SQL Queries** Viktor Leis, Kan Kundhikanjana, Alfons Kemper, Thomas Neumann, 2015.	pdf
SIGMOD 2015	**Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems** Thomas Neumann, Tobias Mühlbauer, Alfons Kemper, 2015.	pdf
DanaC 2015	**High-Performance Main-Memory Database Systems and Modern Virtualization: Friends or Foes?** Tobias Mühlbauer, Wolf Rödiger, Andreas Kipf, Thomas Neumann, Alfons Kemper, 2015.	pdf
TKDE	**Scaling HTM-Supported Database Transactions to Many Cores** Viktor Leis, Alfons Kemper, Thomas Neumann, 2015.	
VLDB 2015	**The More the Merrier: Efficient Multi-Source Graph Traversal** Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, Huy T. Vo, 2015.	pdf
ICDE 2015	**Supporting Hierarchical Data in SAP HANA** Robert Brunel, Jan Finis, Gerald Franz, Norman May, Alfons Kemper, Thomas Neumann, Franz Faerber, 2015.	

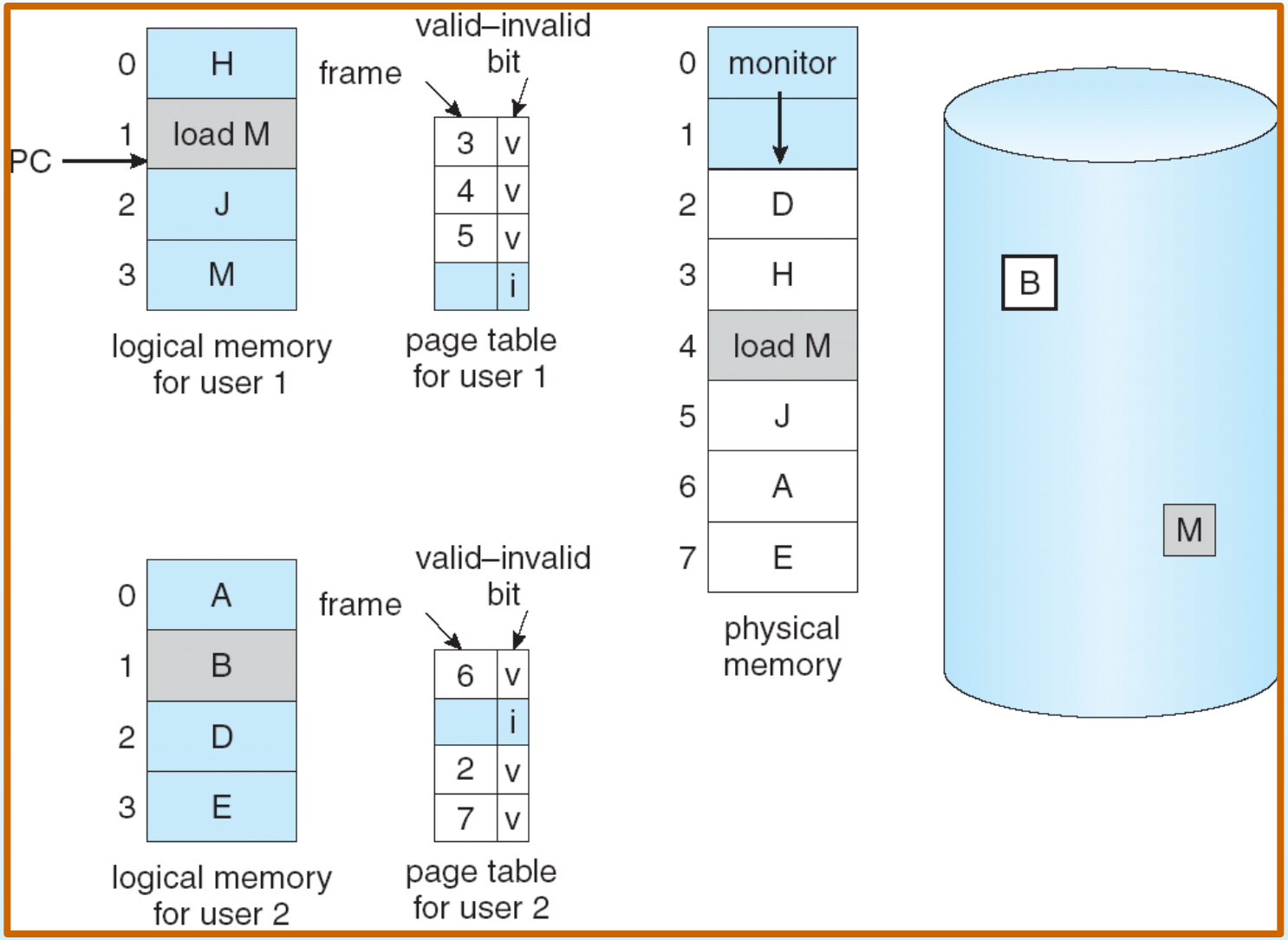
What happens if there is no free frame?

- 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

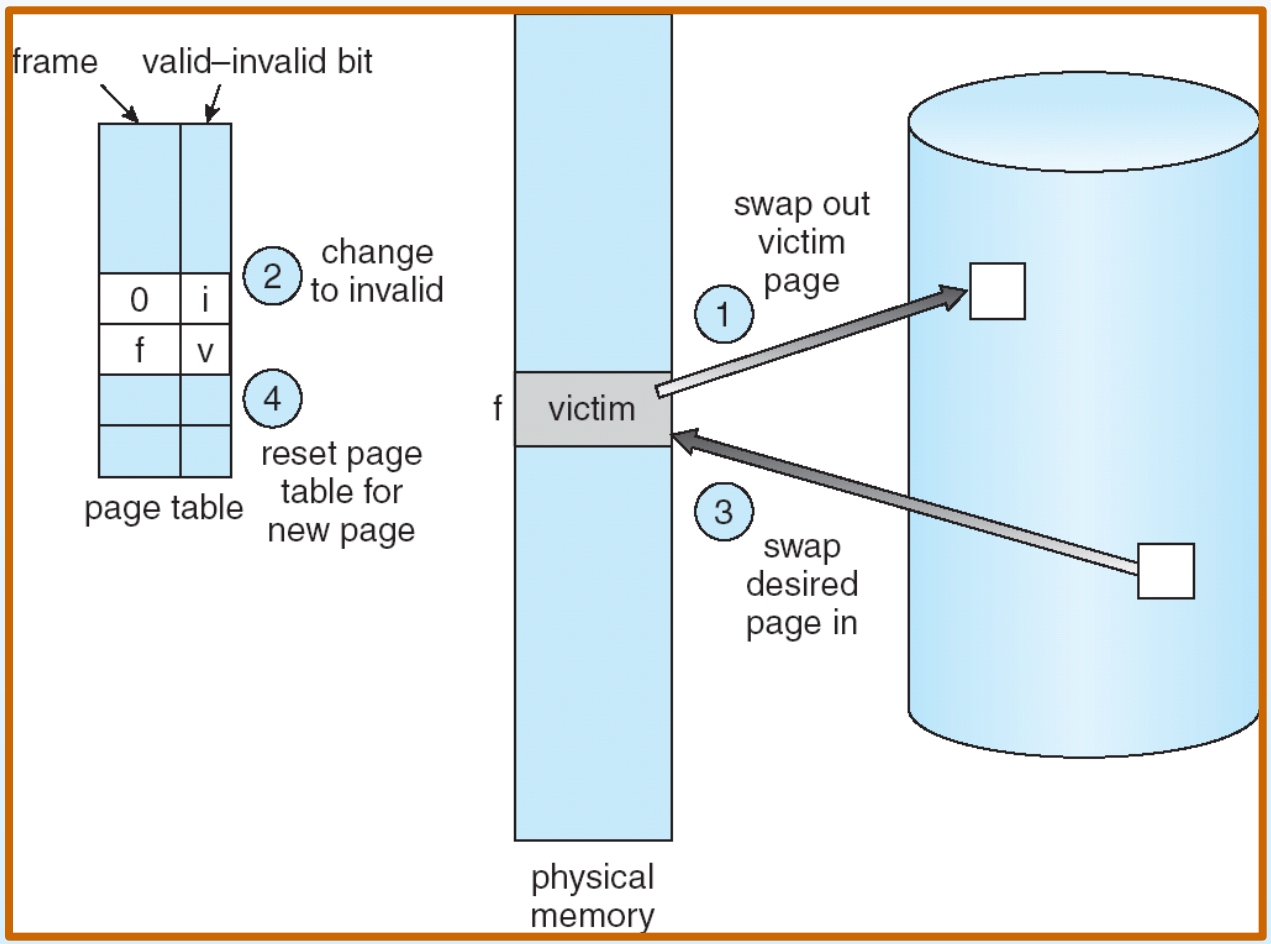
1. Find the location of the desired page on disk

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

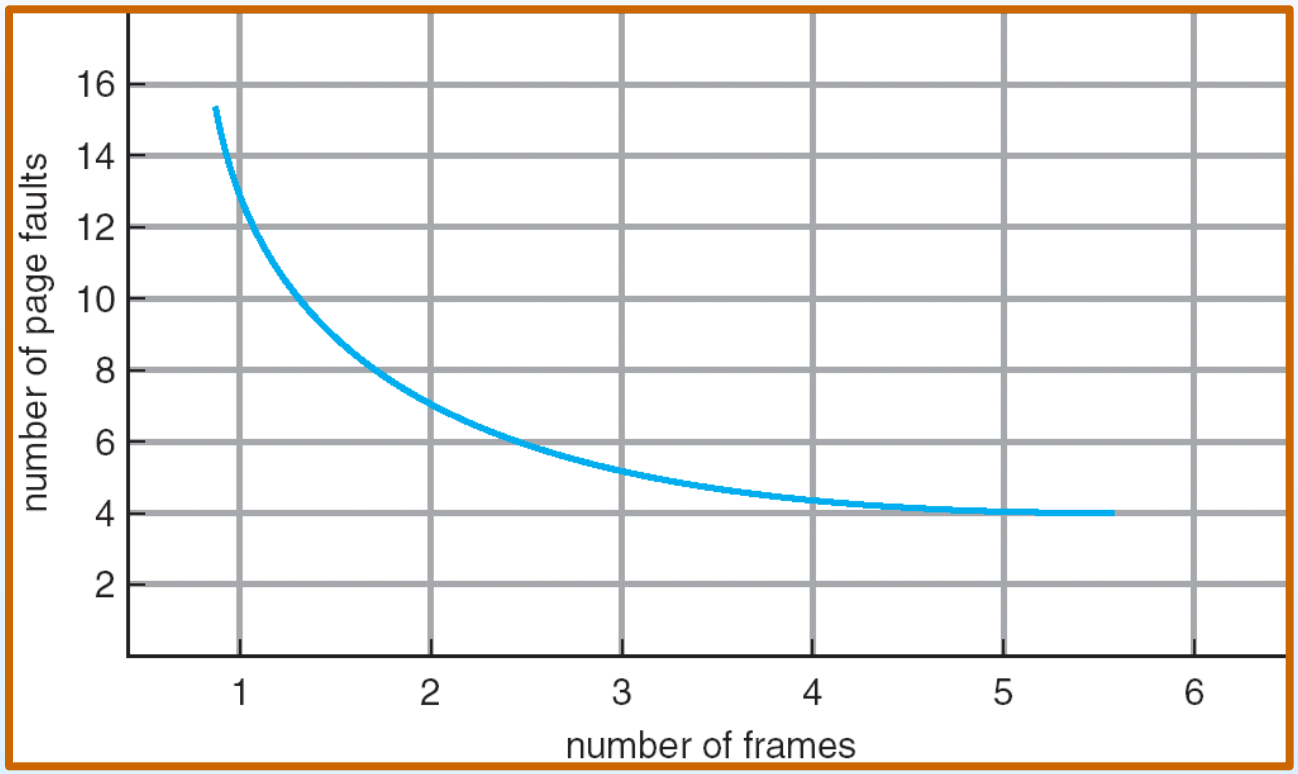


Page Replacement Algorithms

- 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

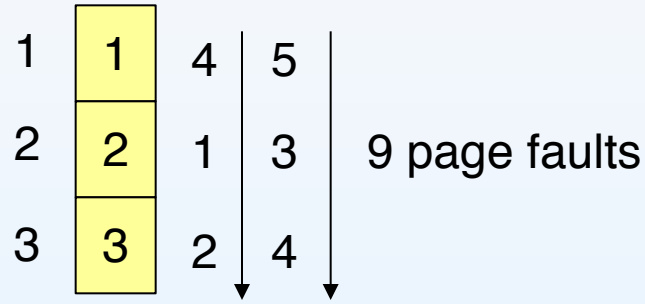
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Graph of Page Faults Versus The Number of Frames



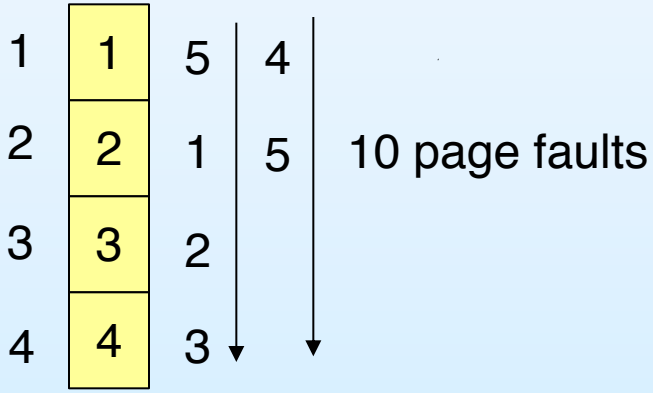
FIFO Page Replacement

- 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)



9 page faults

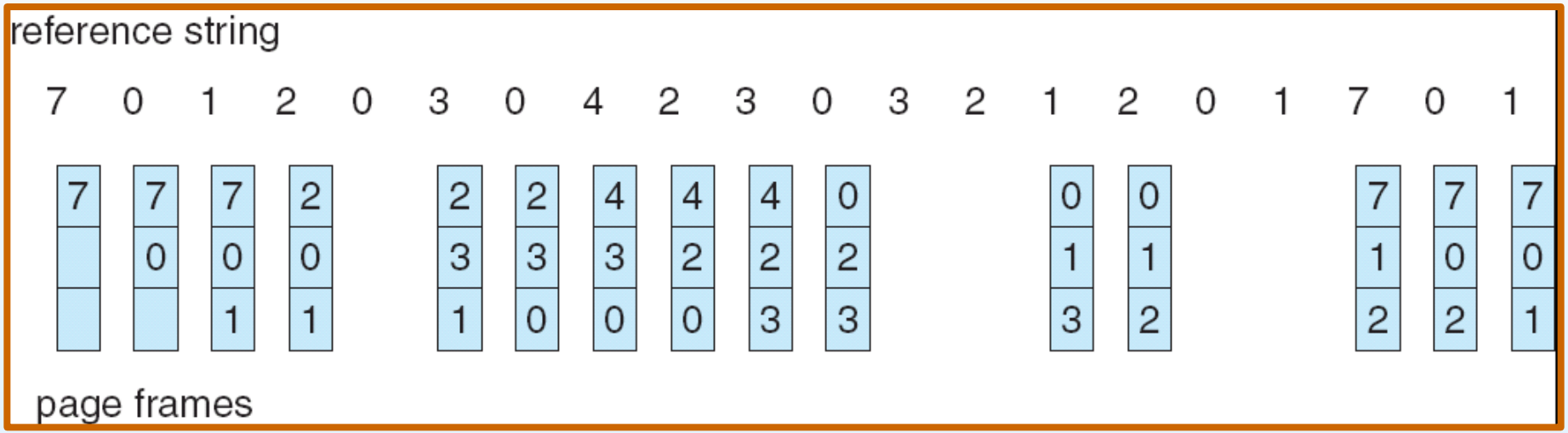
- 4 frames



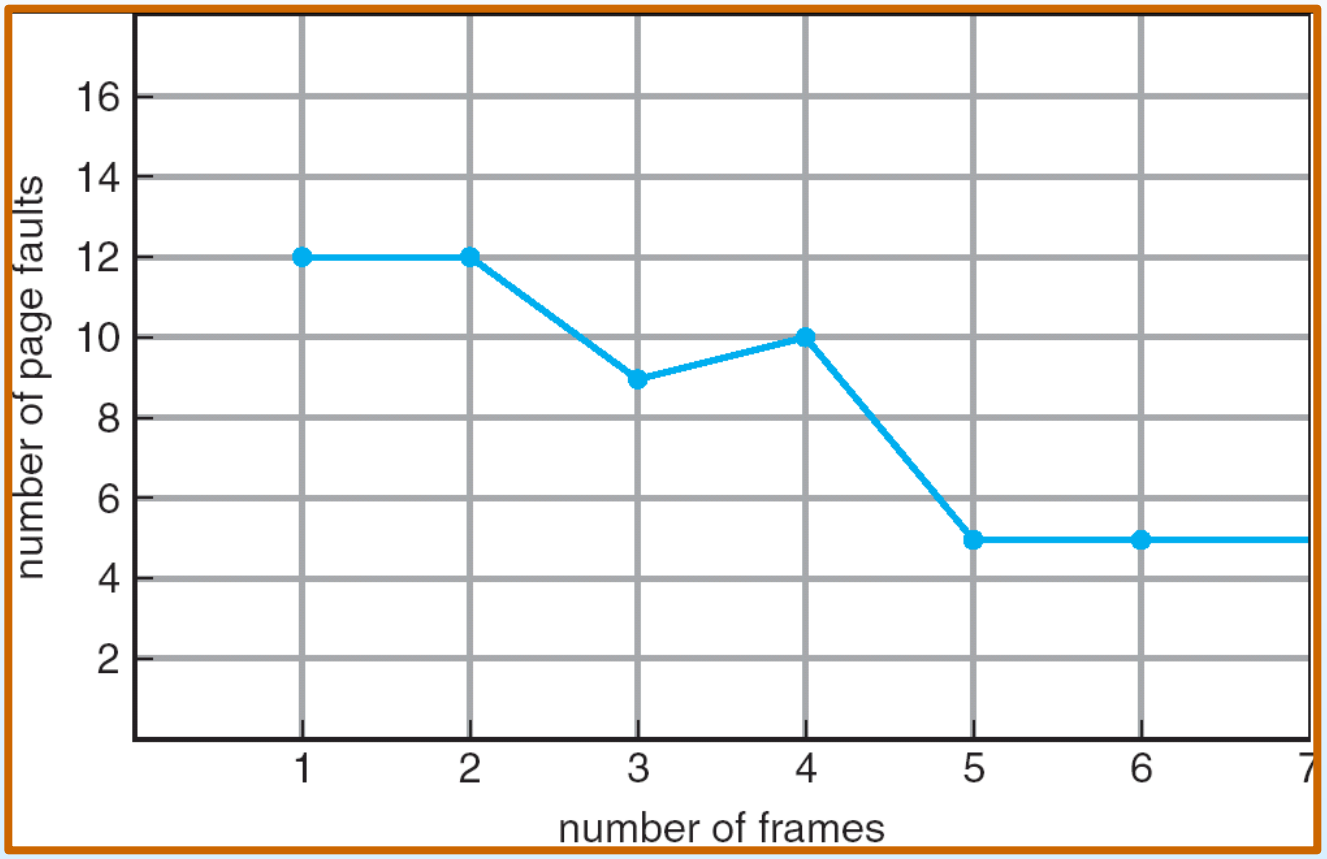
10 page faults

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

FIFO Page Replacement



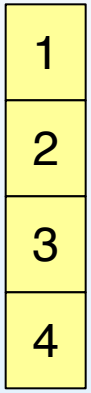
FIFO Illustrating Belady's Anomaly



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



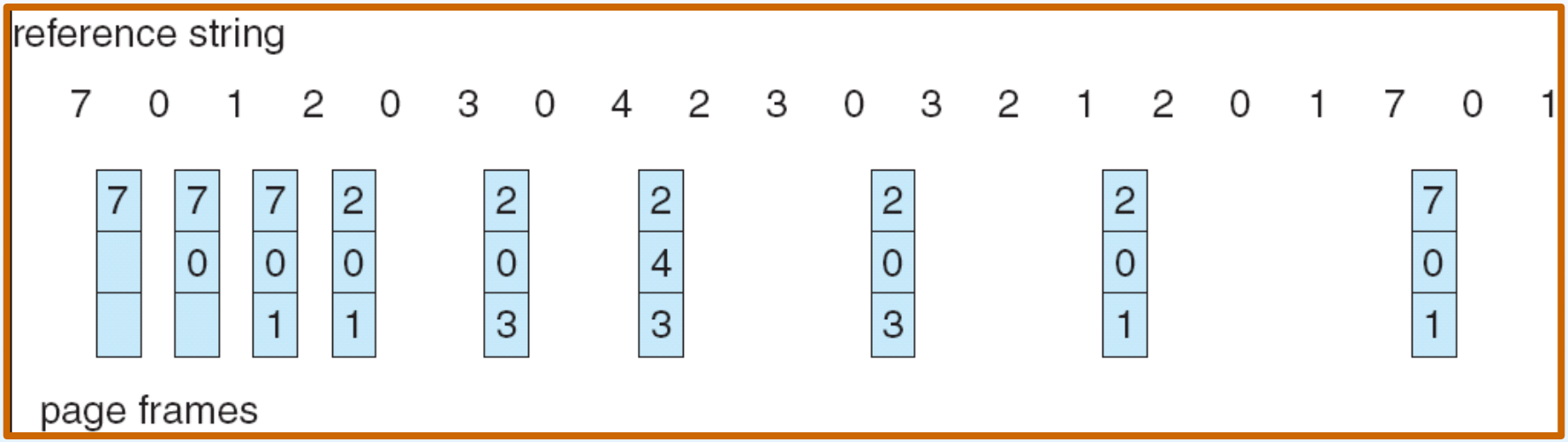
4

5

6 page faults

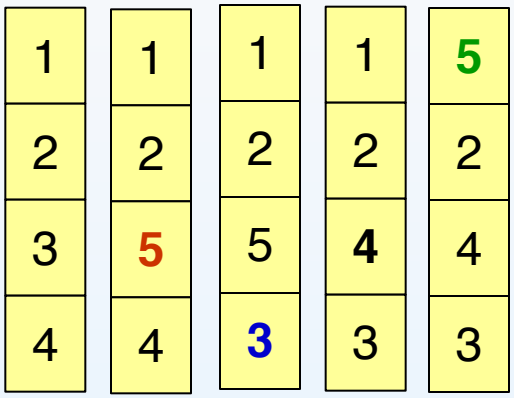
- How do you know this?
- Used for measuring how well your algorithm performs

Optimal Page Replacement



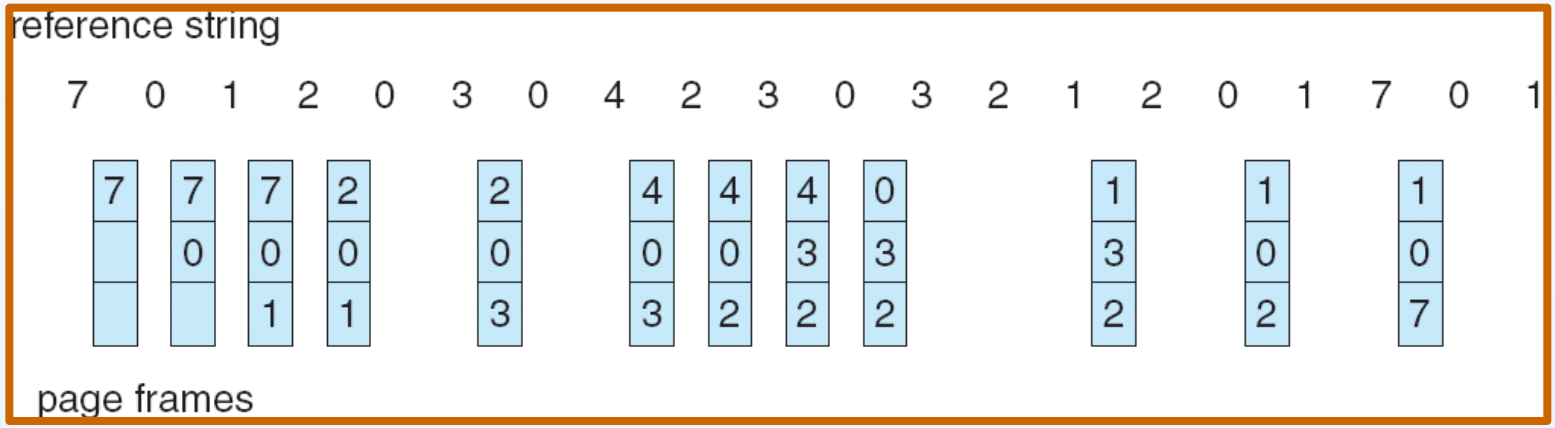
Least Recently Used (LRU) Algorithm

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



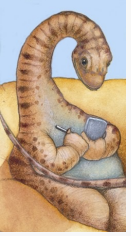
- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to determine which are to change

LRU Page Replacement

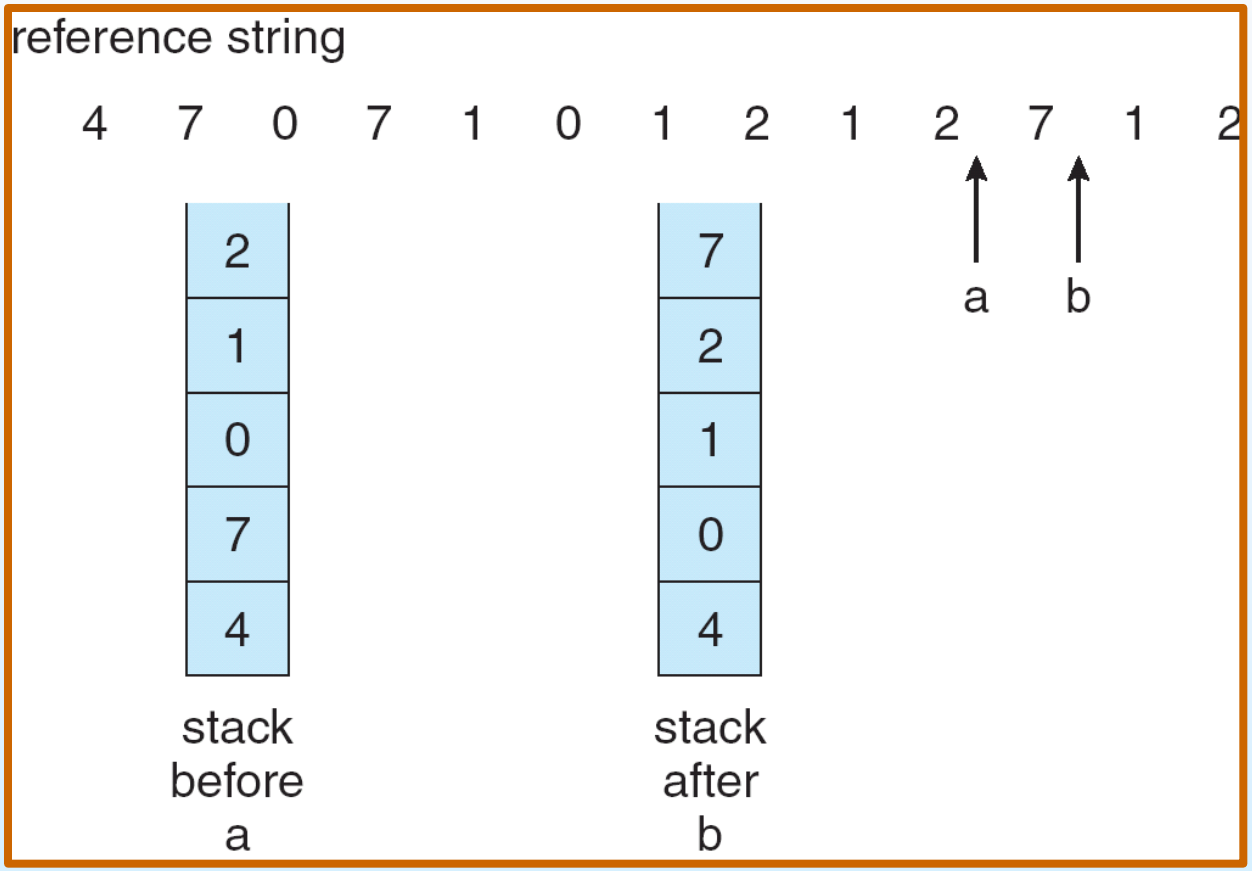


LRU Algorithm (Cont.)

- Stack implementation – keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - bottom item to be replaced
  - No search for replacement



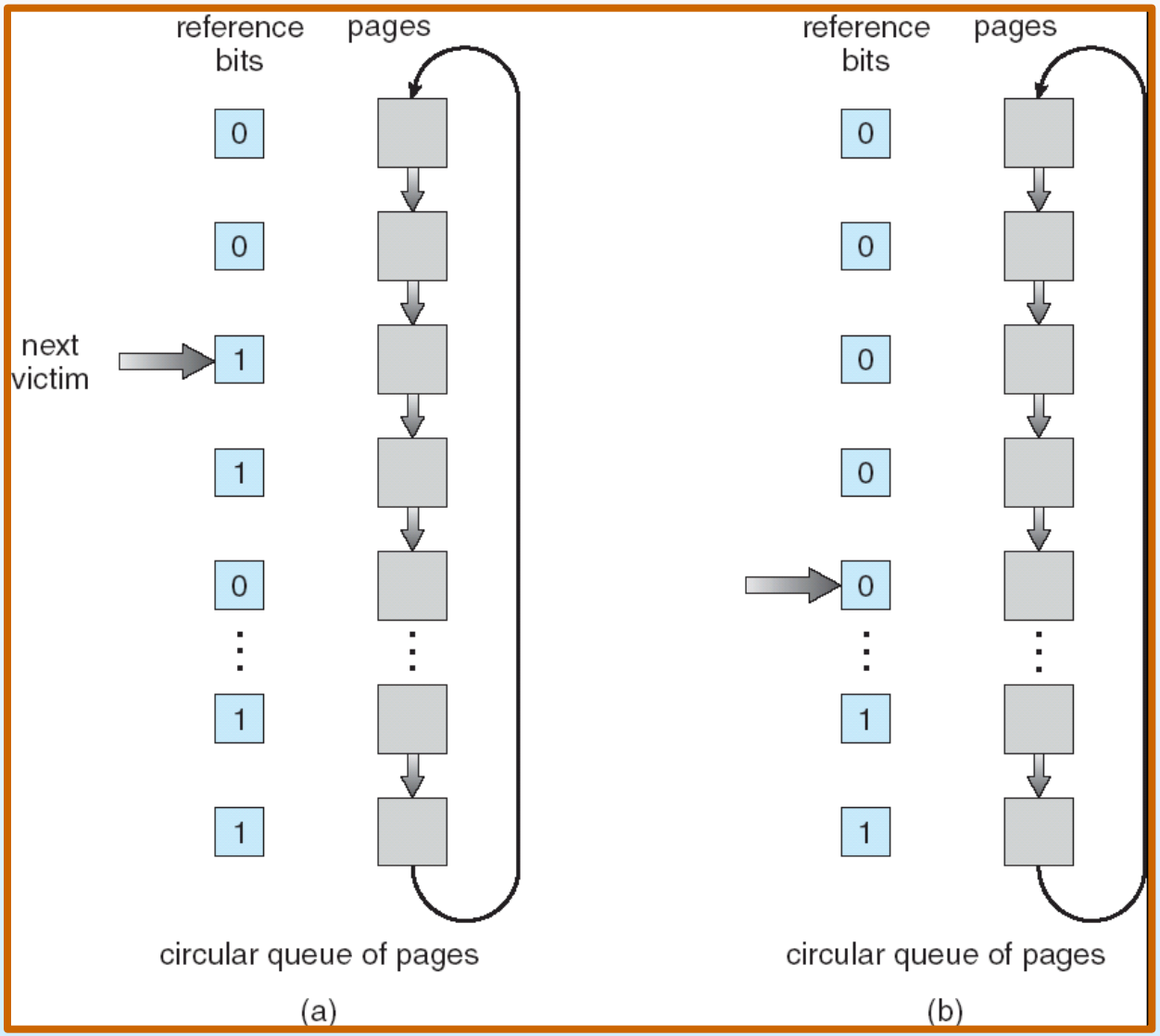
Use Of A Stack to Record The Most Recent Page References



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
    - 0000000 VS 00000001 VS 01001000
- 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

- Each process needs *minimum* number of pages — usually determined by computer architecture.
- Example: IBM 370 – 6 pages to handle Storage-to-Storage 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 = size of process p_i
  - $S = \sum s_i$
  - m = total number of frames
  - a_i = 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

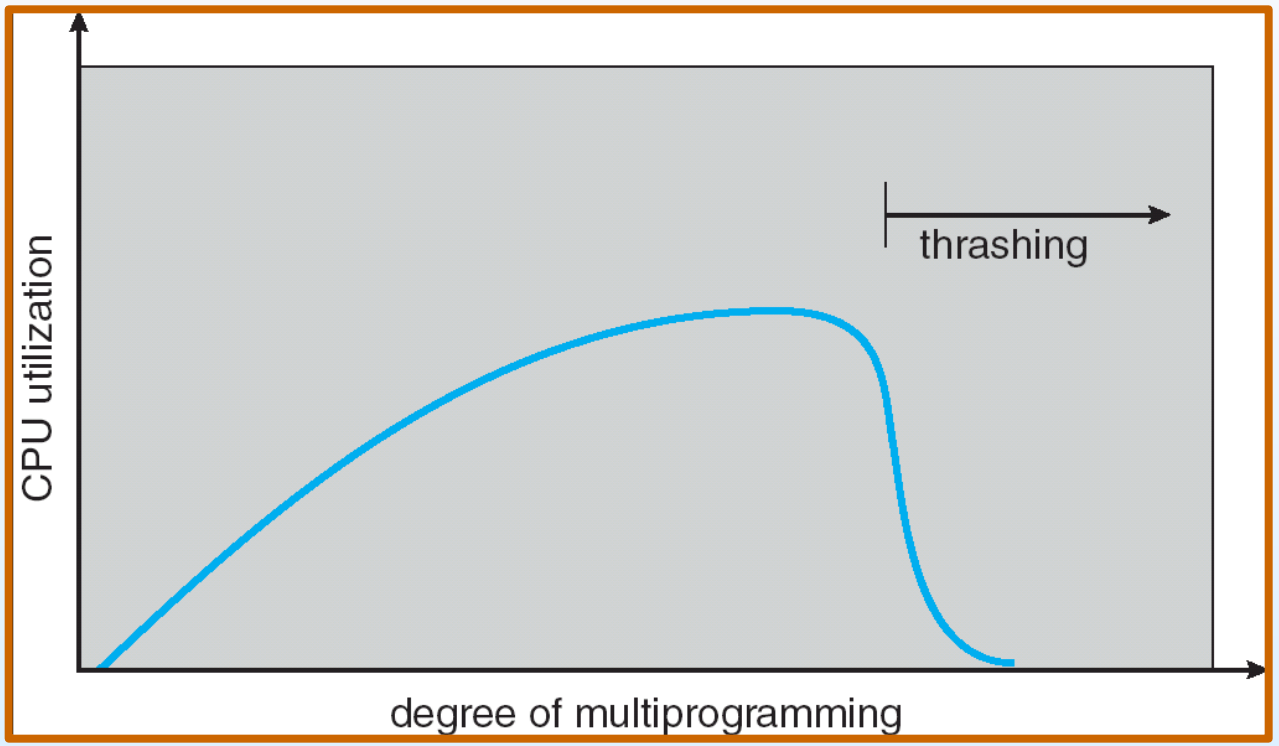
- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
- **Local replacement** – each process selects from only its own set of allocated frames

- Problem with global replacement: unpredictable page-fault rate. Cannot control its own page-fault rate. More common
- Problem with local replacement: free frames are not available for others. – Low throughput

Thrashing （颠簸）

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - Queuing at paging device, the ready queue becomes empty
  - 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 (Cont.)



Demand Paging and Thrashing

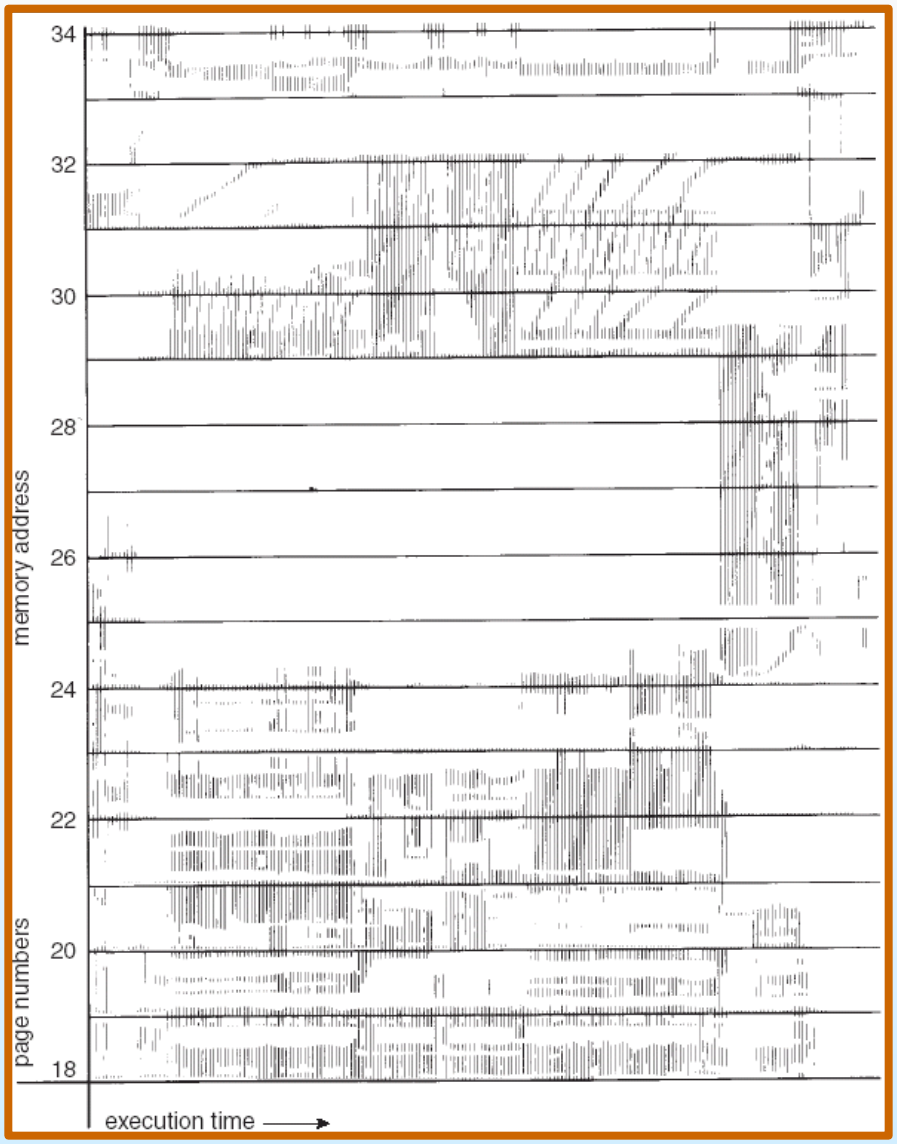
- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap

- Why does thrashing occur?
  Σ size of locality > total memory size

- To limit the effect of thrashing: local replacement algo cannot steal frames from other processes. But queue in page device increases effective access time.

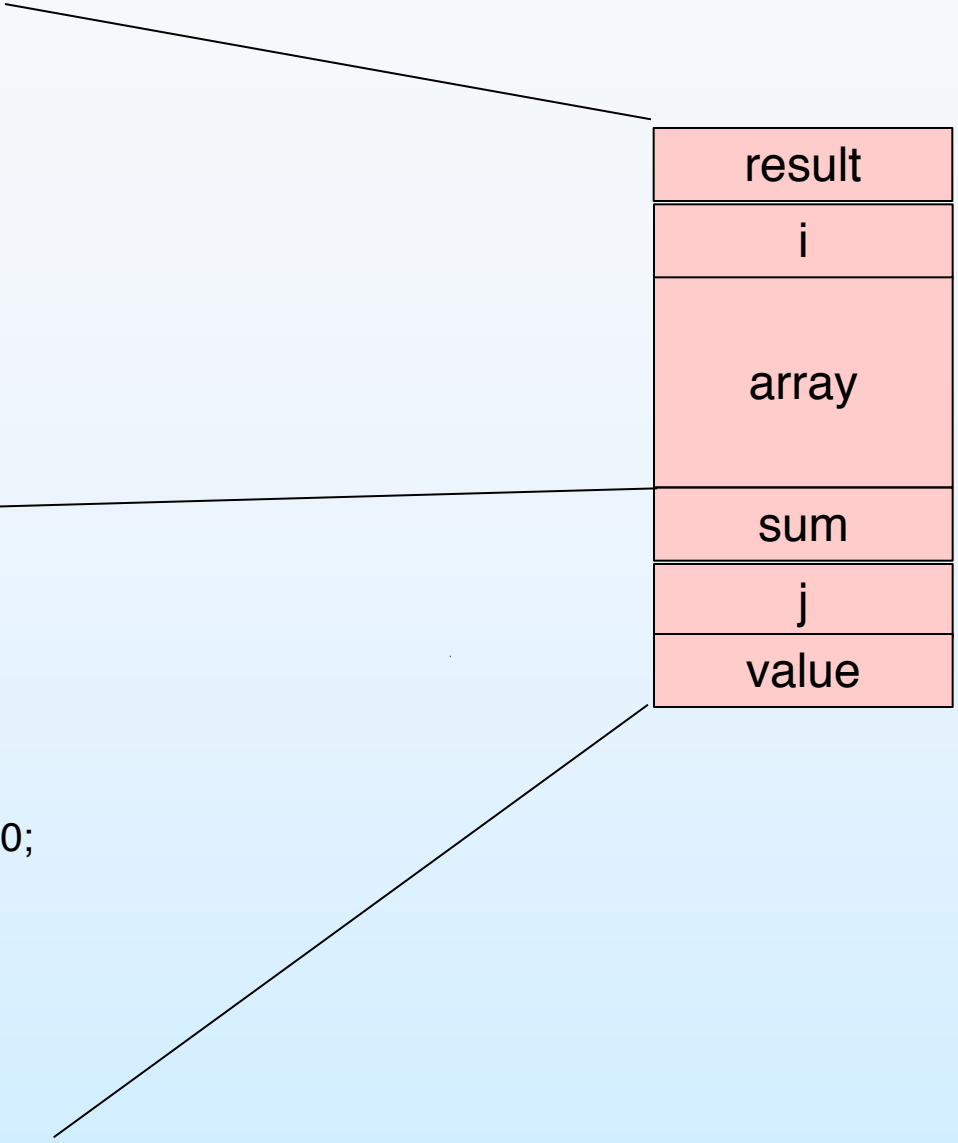
- To prevent thrashing: allocate memory to accommodate its locality

Locality In A Memory-Reference Pattern



Data Locality Example

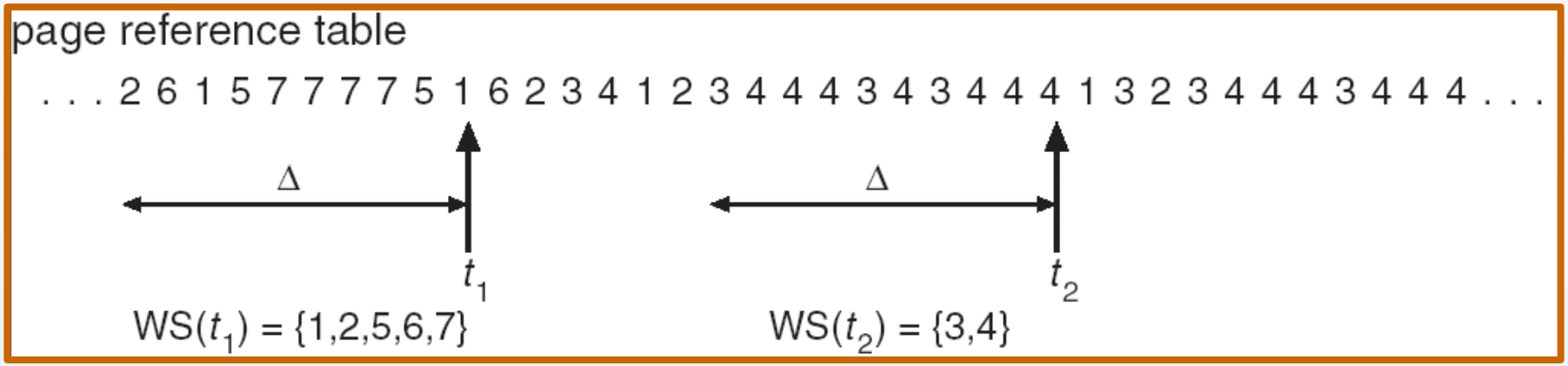
```
Function A()
{     int i=0; float result;
      float array[100];
      for (i=0; i< 100; i++){
            array[i]=i*PI;
      }
      result= B();
}

Function B()
{
      int j=0;
      float  value, sum;
      for(j=0; j< 100; j++){
            value = rand()*10;
            sum+= value;
      }
      return sum
}
```



Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instruction
- WSS_i (working set size 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 for all processes in the system
- if $D > m \Rightarrow$ Thrashing
- Policy if $D >$ m, then suspend one of the processes

Working-set model

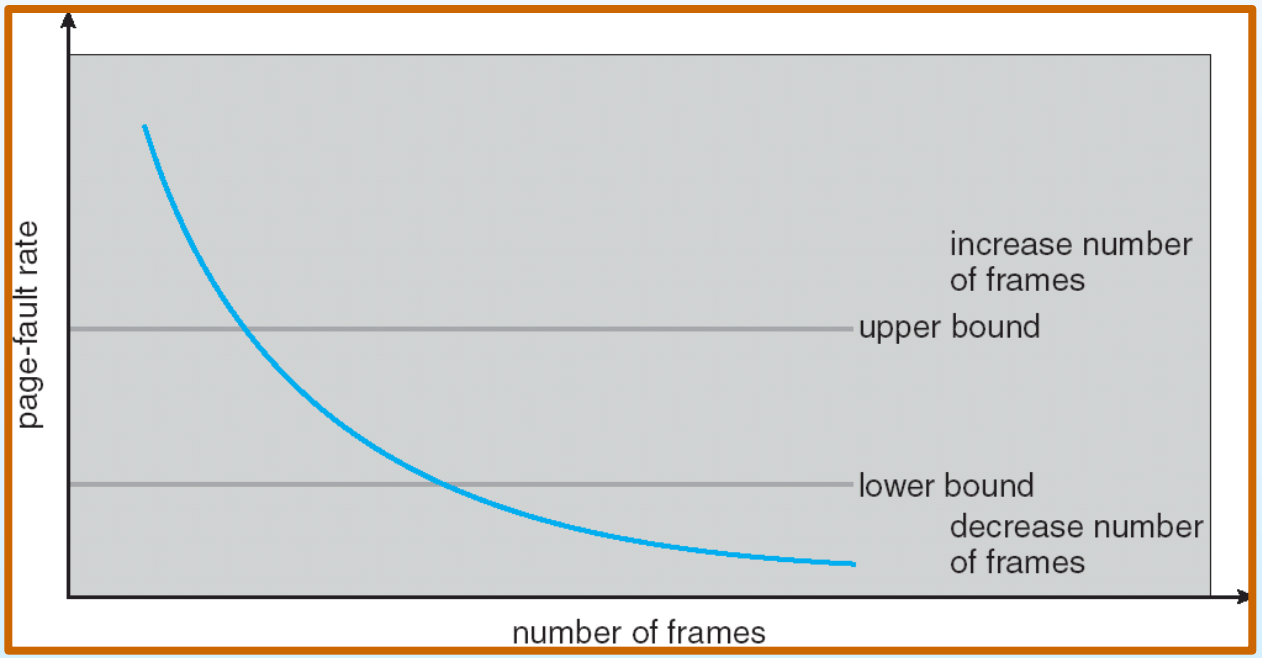


Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: Δ = 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? Cannot tell where a reference occurred in 5000 units.
- Improvement = 10 bits and interrupt every 1000 time units

Page-Fault Frequency Scheme

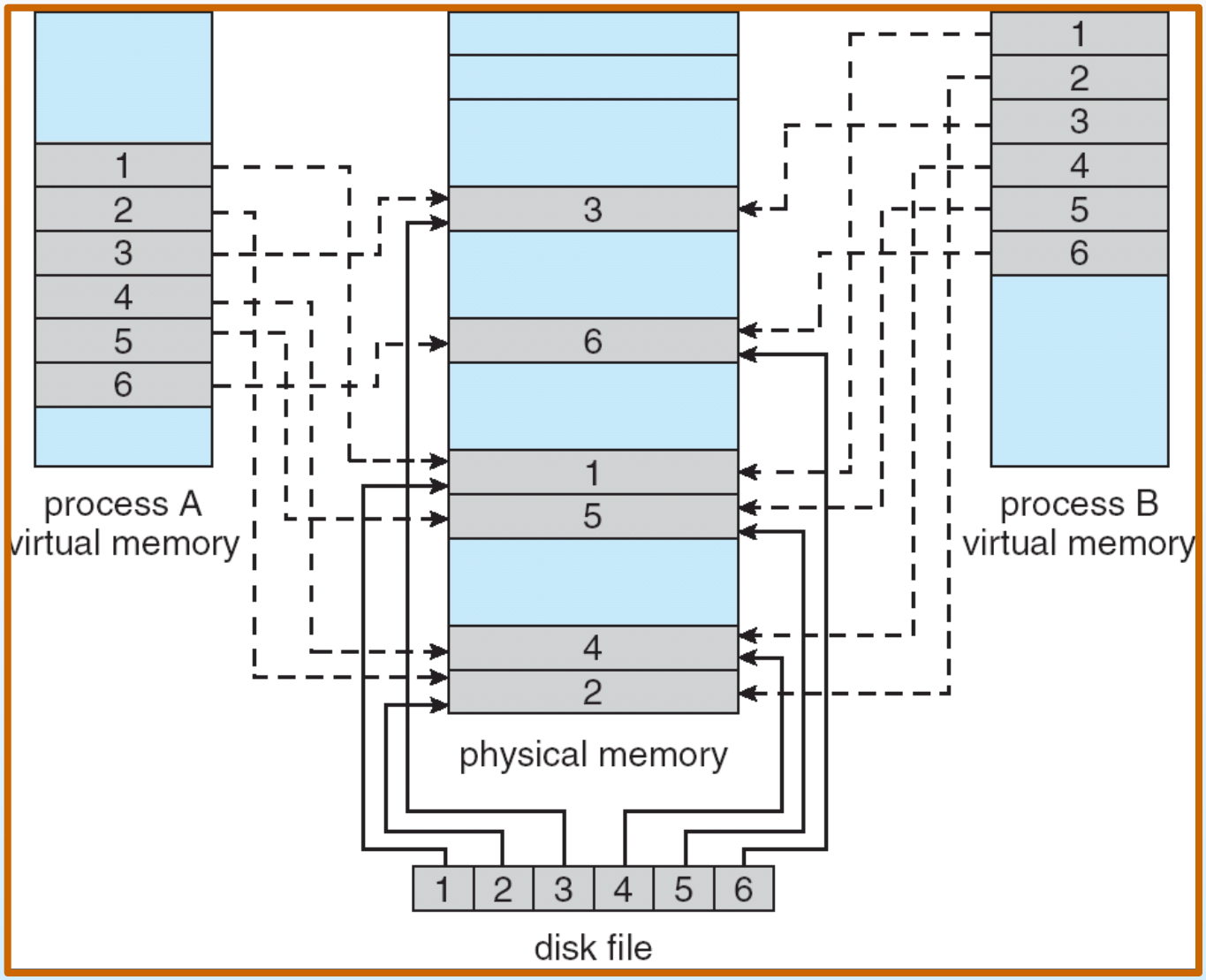
- Establish “acceptable” page-fault rate for each process
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Memory-Mapped Files

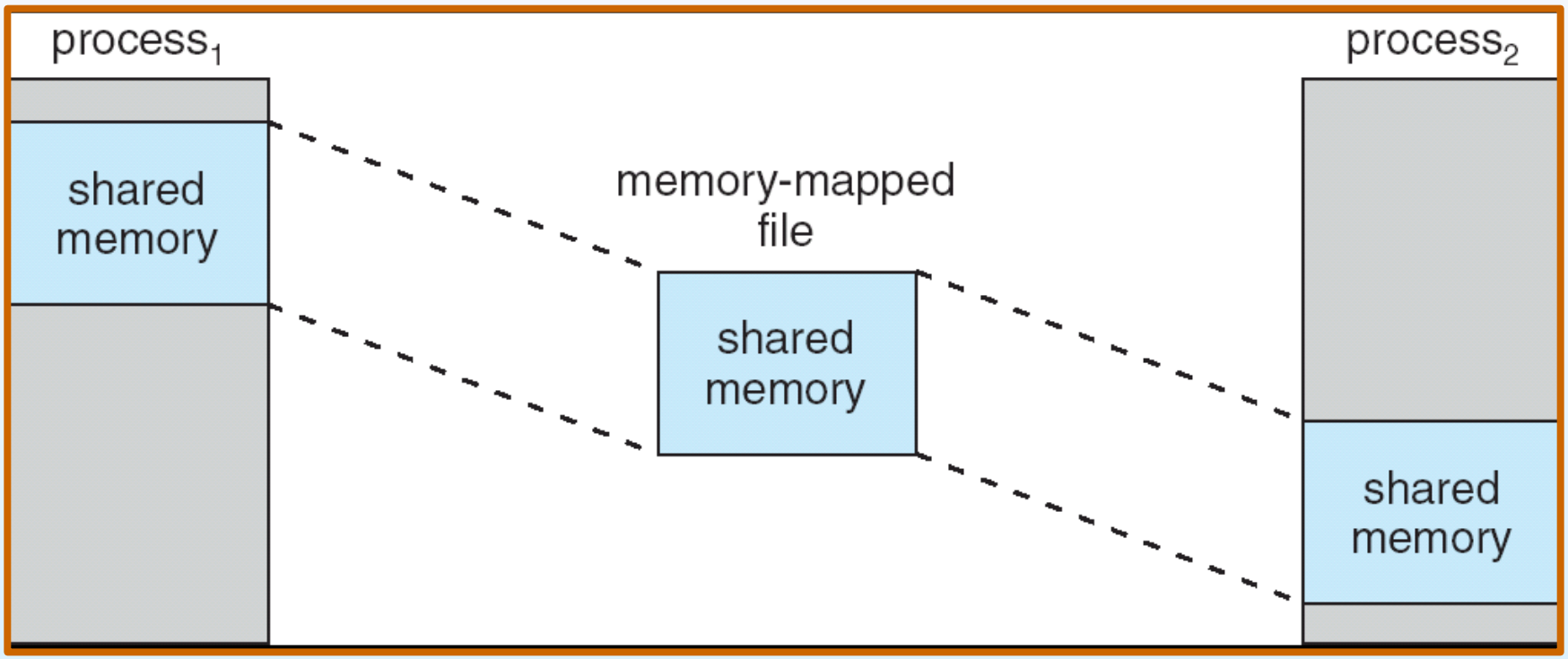
- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file 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



Memory-Mapped Shared Memory in Windows

Refer to text book pg 351



Allocating Kernel Memory

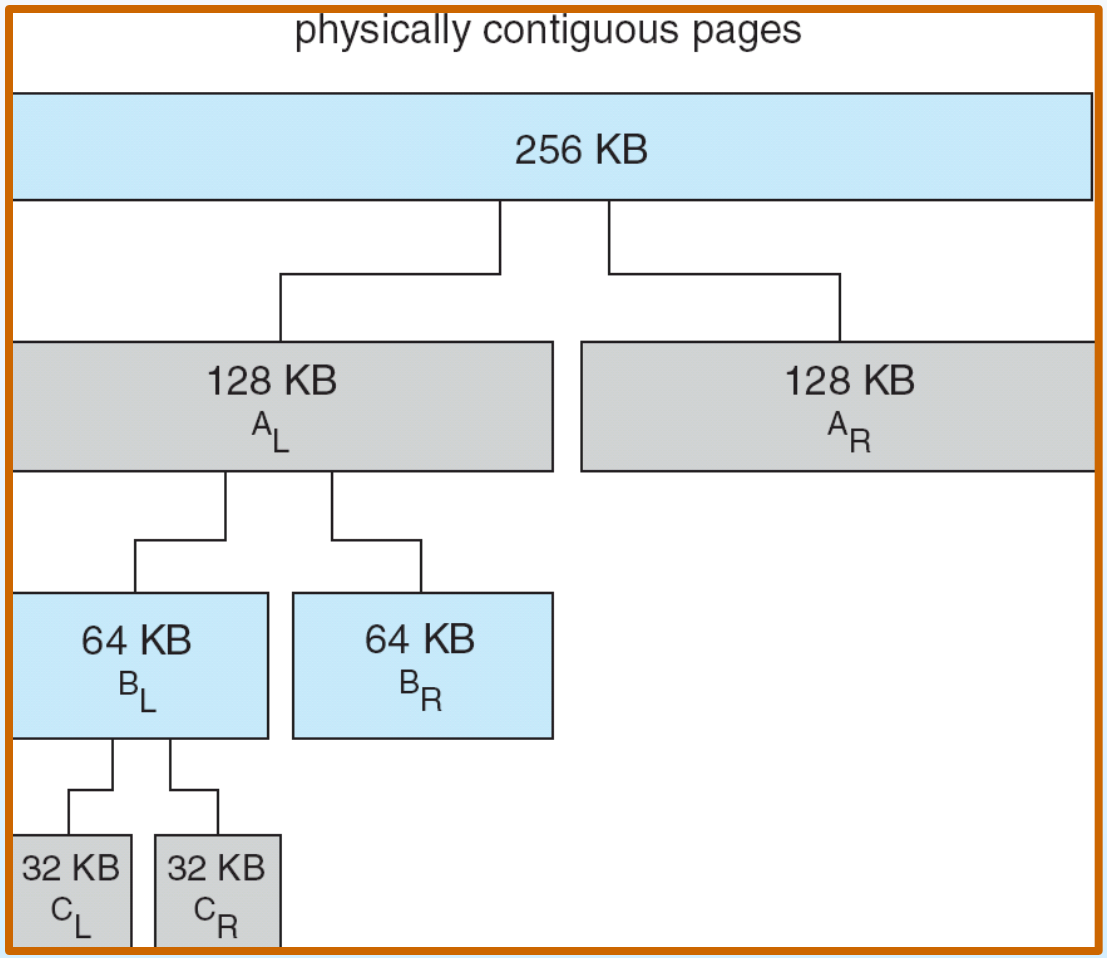
- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes – needs to reduce fragmentation
  - Some kernel memory needs to be contiguous (certain h/w device interacts with contiguous physical memory)

  Therefore, many systems do NOT utilize paging for kernel code and data.

Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available

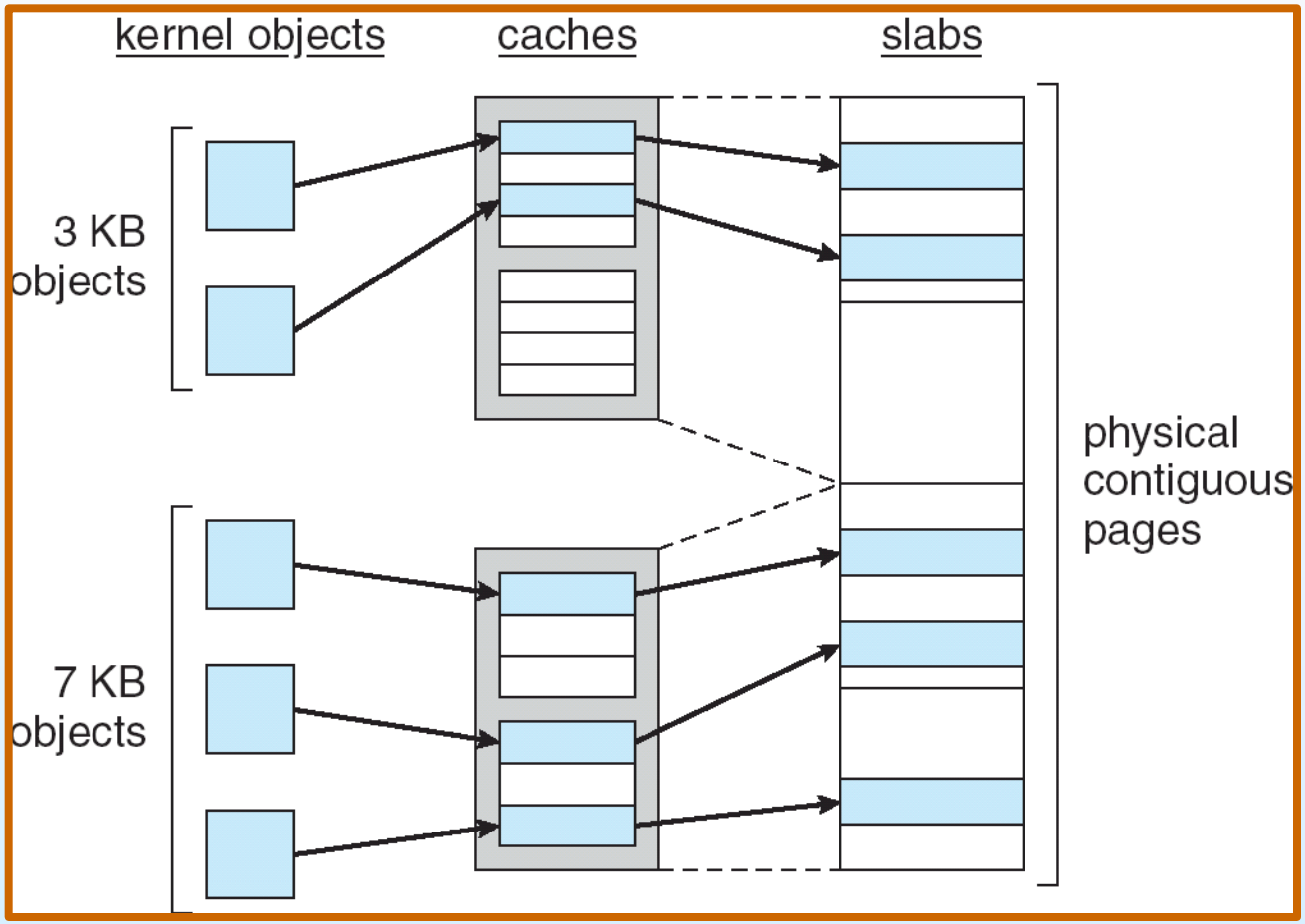
Buddy System Allocator



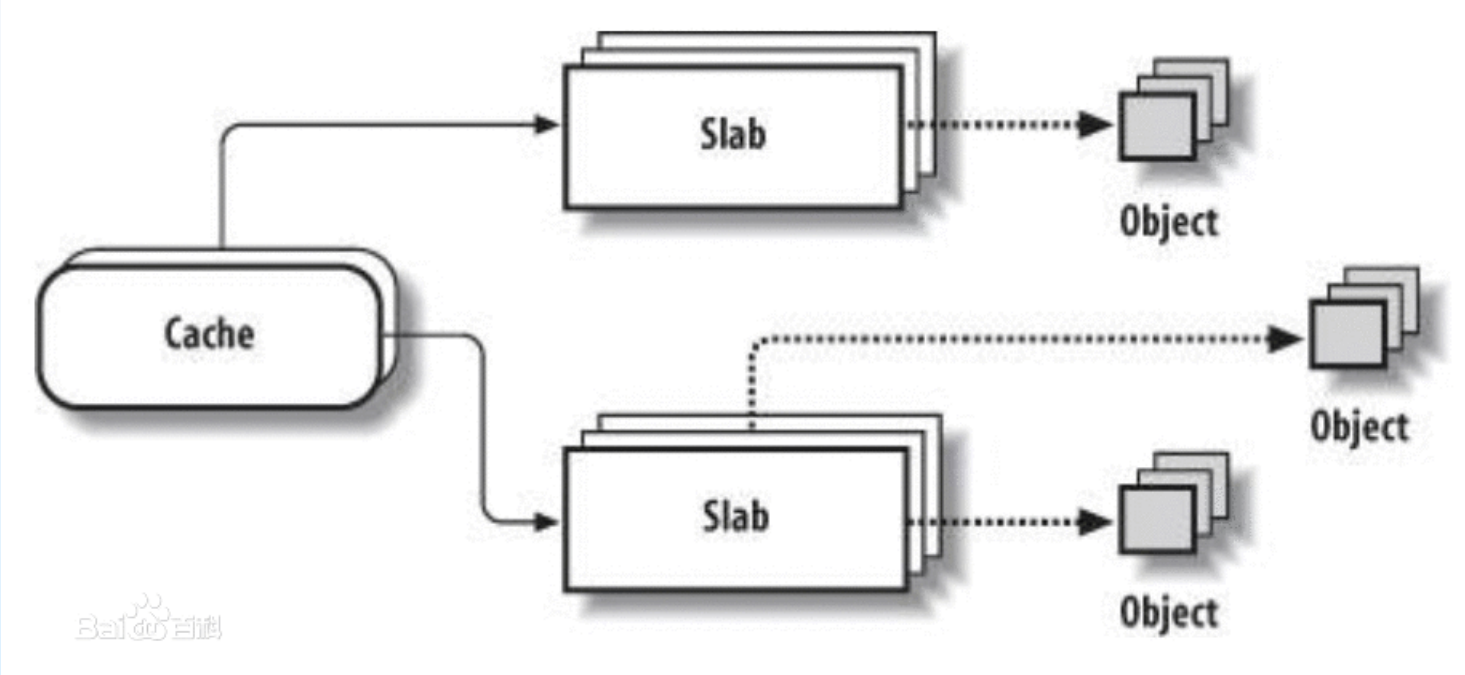
Slab Allocator

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

Slab Allocation



Slab Allocation



Other Issues -- Prepaging

- 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
    - Is benefit of *s * α* saved pages faults > or < than the cost of prepaging
      *s * (1- α)* unnecessary pages*?*
    - *α* near zero ⇒ prepaging loses

Other Issues – Page Size

- Page size selection must take into consideration:
  - Fragmentation – small page size
  - table size -> large page size
  - I/O overhead -> large page size
  - Locality -> small page size , accurate locality

Other Issues – TLB Reach

- 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
- 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 – Program Structure

- Program structure
  - `Int[128,128] data;`
  - Each row is stored in one page
  - Program 1

```
for (j = 0; j <128; j++)
    for (i = 0; i < 128; i++)
        data[i,j] = 0;
```

128 x 128 = 16,384 page faults

  - Program 2

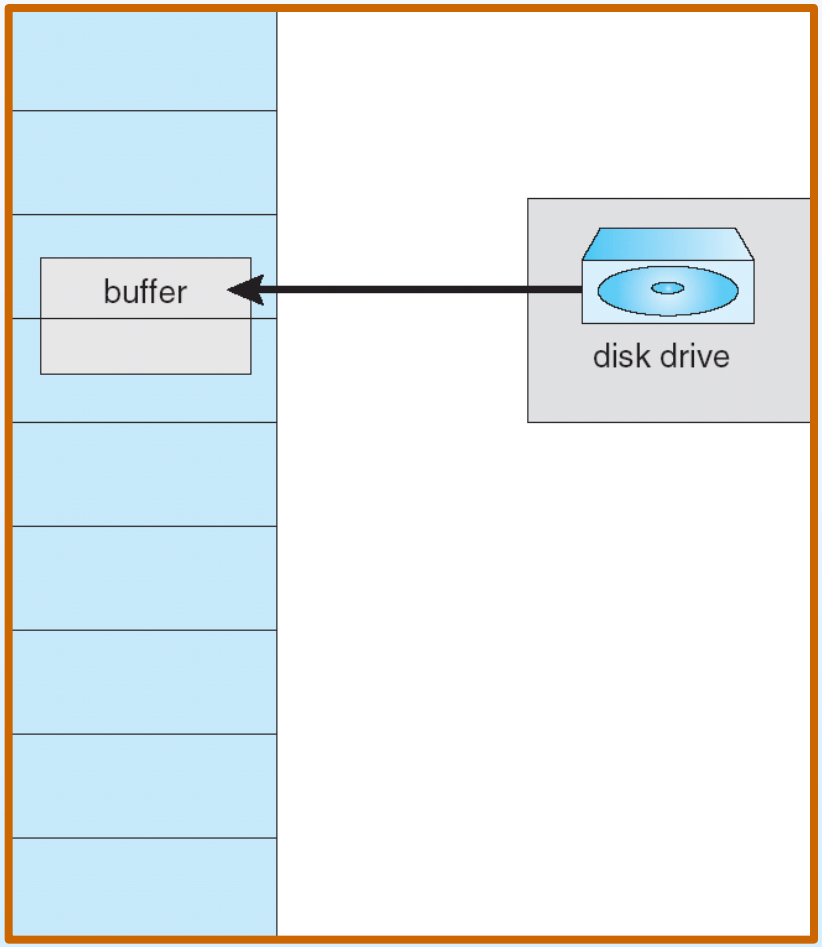
```
for (i = 0; i < 128; i++)
 for (j = 0; j < 128; j++)
     data[i,j] = 0;
```

128 page faults

Other Issues – I/O interlock

- **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



Operating System Examples

- Windows XP
- Solaris

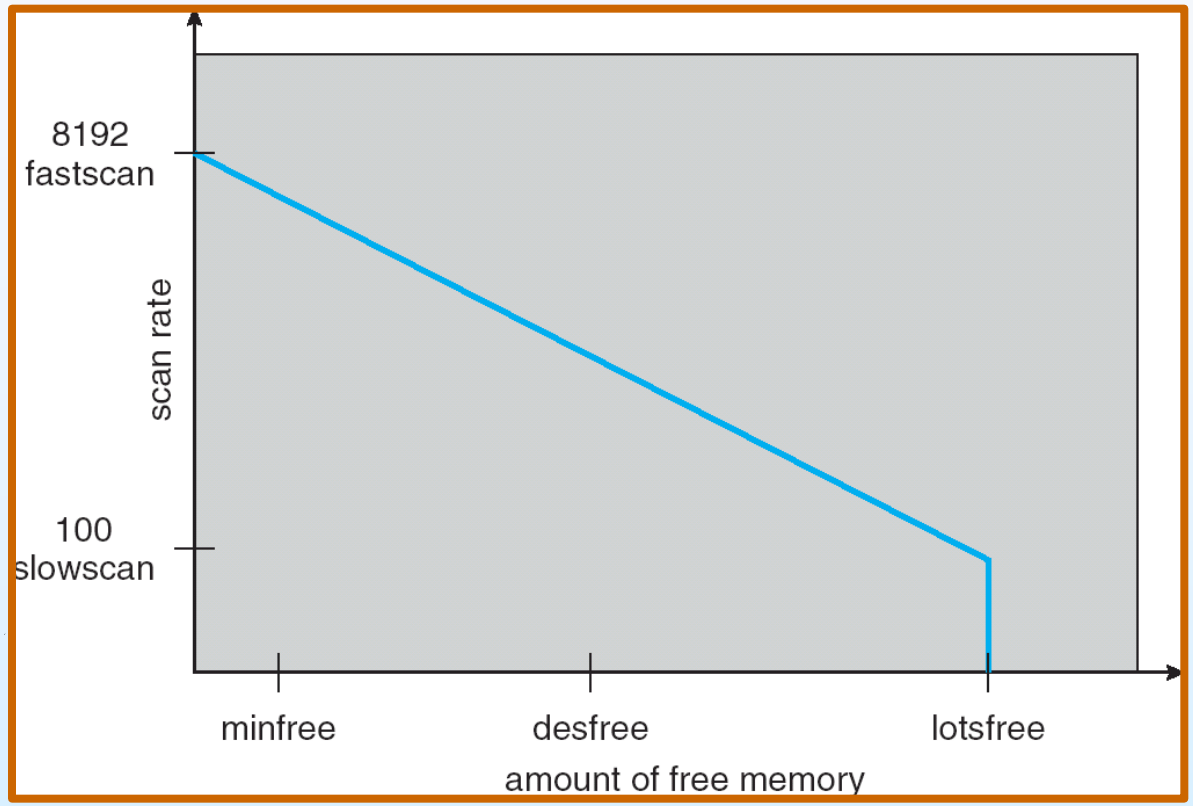
Windows XP

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page.
- Processes are assigned **working set minimum** and **working set maximum (50-345 pages)**
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum. If page fault at maximum, replace local pages.
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum

Solaris

- Maintains a list of free pages to assign faulting processes
- *Lotsfree* – threshold parameter (amount of free memory) to begin paging out
- *Desfree* – threshold parameter to increasing paging frequency
- *Minfree* – threshold parameter to begin swapping
- Paging is performed by *pageout* process
- Pageout scans pages using modified clock algorithm
- *Scanrate* is the rate at which pages are scanned. This ranges from *slowscan* to *fastscan*
- Pageout is called more frequently depending upon the amount of free memory available

Solaris 2 Page Scanner



End of Chapter 9

Page Replacement Problem

- A page-replacement algorithm should minimize the number of page faults. We can do this minimization by distributing heavily used pages evenly over all of memory, rather than having them compete for a small number of page frames. We can associate with each page frame a counter of the number of pages that are associated with that frame. Then, to replace a page, we search for the page frame with the smallest counter.

a. Define a page-replacement algorithm using this basic idea. Specifically address the problems of (1) what the initial value of the counters is, (2) when counters are increased, (3) when counters are decreased, and (4) how the page to be replaced is selected.

b. How many page faults occur for your algorithm for the following reference string, for four page frames?

1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2.

c. What is the minimum number of page faults for an optimal page-replacement strategy for the reference string in part b with four page frames?

Page Replacement Problem

- a.

(1) Initial value of the counters -- 0

(2) Counters are increased -- whenever a new page is associated with that frame

(3) Counters are decreased -- whenever one of the pages associated with that frame is no longer required

(4) How the page to be replaced is selected -- find a frame with the smallest counter. Use FIFO for breaking ties.

Page Replacement Problem

- b. The number of page faults is 13.

1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2

1	1	1	1	5			5	5	5	8		8			8	8			2		
	2	2	2	2			1	1	1	1		9			9	9			9		
		3	3	3			3	6	6	6		6			6	5			5		
			4	4			4	4	7	7		7			7	4			4		

- c. The number of page faults is 11 for optimal page-replacement strategy.

1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2

1	1	1	1	1			6	7	7		7				4			4			
	2	2	2	5			5	5	5		5				5			5			
		3	3	3			3	3	8		8				8			2			
			4	4			4	4	4		9				9			9