LUT University

LAND OF THE CURIOUS

JANUARY 17, 2023

OPERATING SYSTEMS AND SYSTEMS PROGRAMMING (CT30A3370) 6 CREDITS

Venkata Marella

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

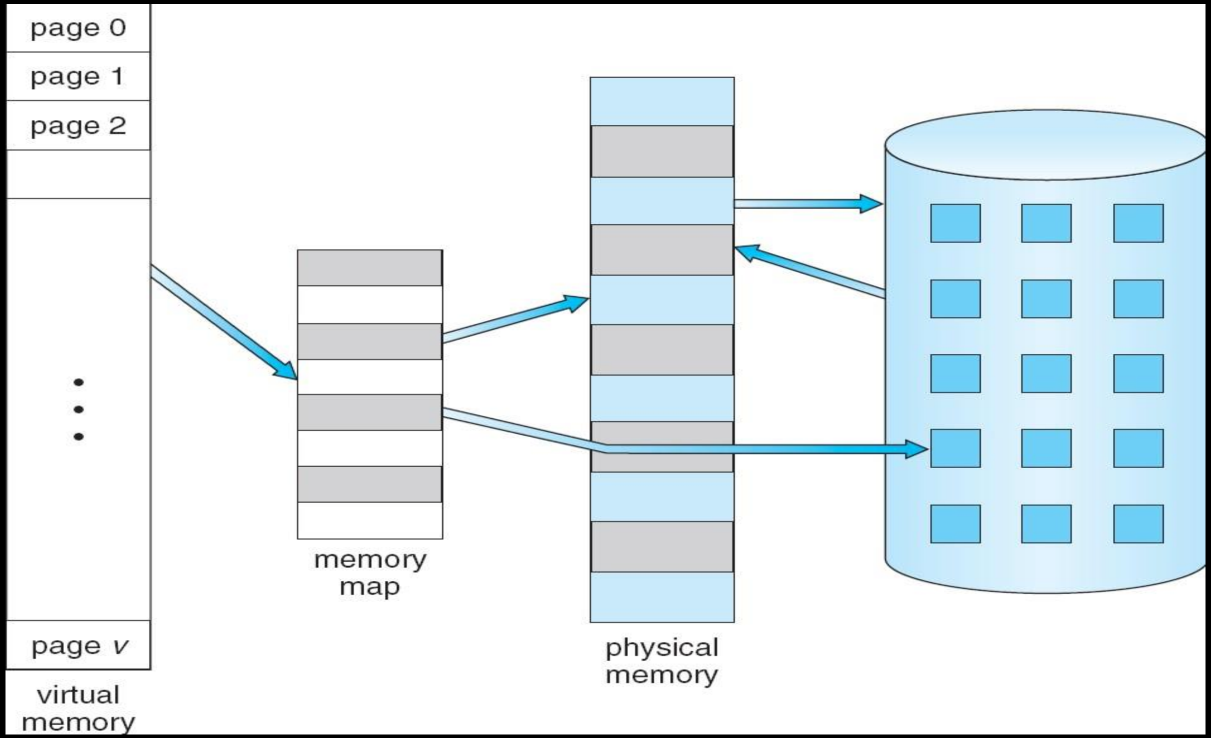
OBJECTIVES

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

BACKGROUND

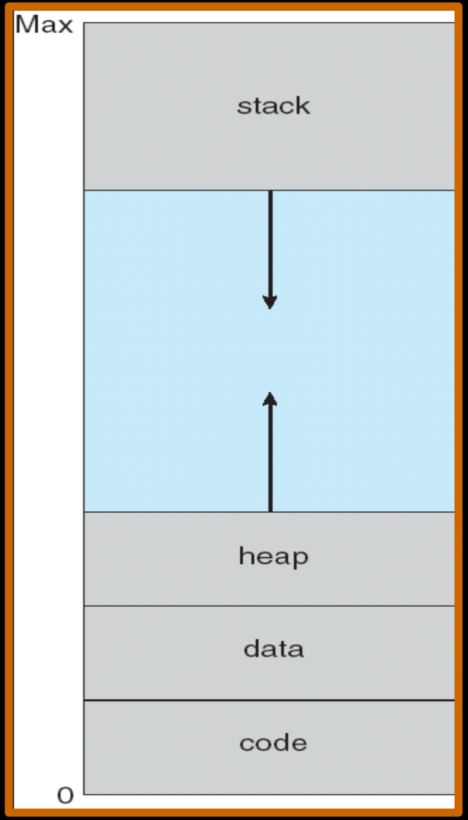
- **Virtual memory** – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for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

VIRTUAL MEMORY THAT IS LARGER THAN PHYSICAL MEMORY



Silberschatz, Galvin Operating System Concepts – 7th Edition and Gagne ©2005

VIRTUAL-ADDRESS SPACE



hole

Sparse 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

```
                 0                                                                     4GB
VAS              |_____________________________________________________________________|
```

» Load app.exe into memory

```
                 0                                                                     4GB
VAS              |____vvvvvvv__________________________________________________________|
mapping               |-----|
file bytes            app.exe
```

WHY VIRTUAL MEMORY (CONTINUED)

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

```
                0                                                 4GB
VAS            |---vvvvvvv----vvvvvv---vvvv-------------------|
mapping            |||||||     ||||||    ||||
file bytes         app.exe     kernel    user
```

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

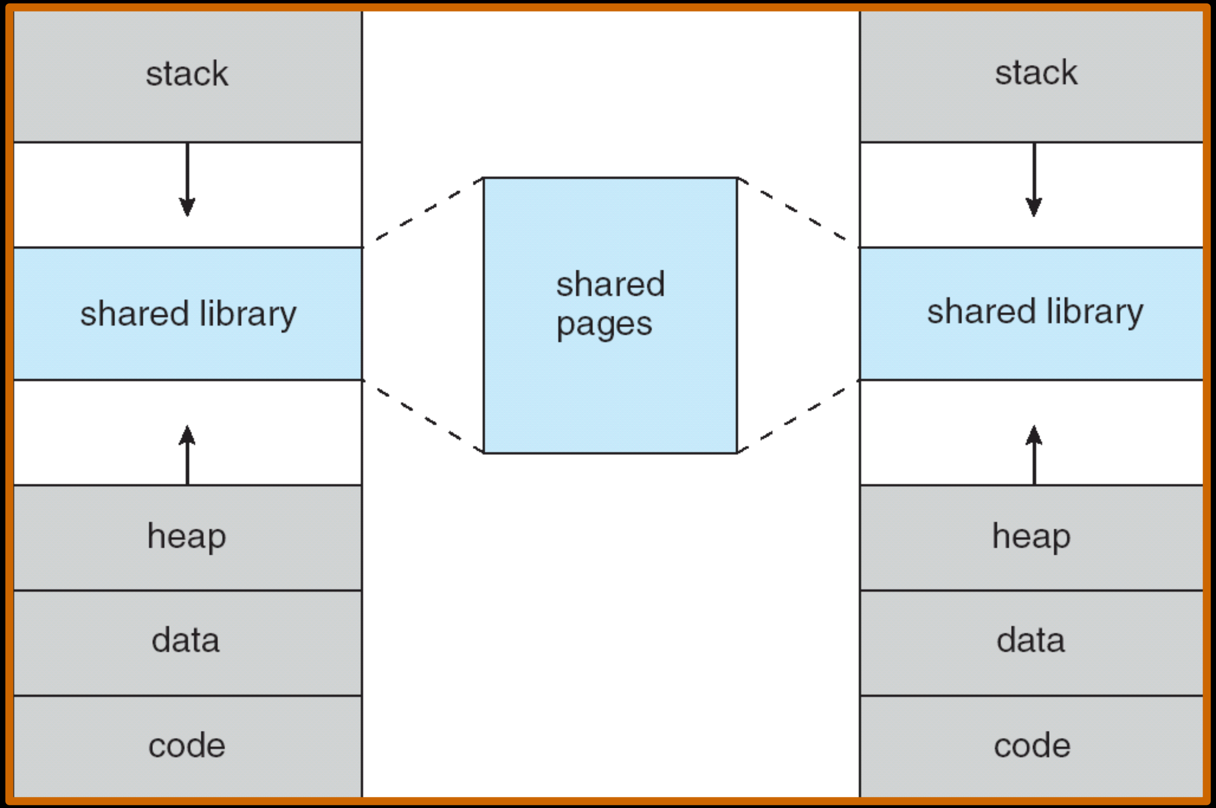
```
                0                                                 4GB
VAS            |---vvvvvvv----vvvvvv---vvvv----vv---v----vvv--|
mapping            |||||||     ||||||    ||||     ||   |     |||
file bytes         app.exe     kernel    user    system_page_file
```

WHY VIRTUAL MEMORY (CONTINUED)

» What if we have more users and more apps

```
            0                                                4GB
VAS 1       |---vvvv-------vvvvvv---vvvv----vv---v----vvv--|
mapping         ||||       ||||||   ||||    ||   |    |||
file bytes      app1 app2  kernel   user   system_page_file
mapping              ||||  ||||||   ||||       ||   |
VAS 2       |--------vvvv--vvvvvv---vvvv-------vv---v------|
```

SHARED LIBRARY USING VIRTUAL MEMORY

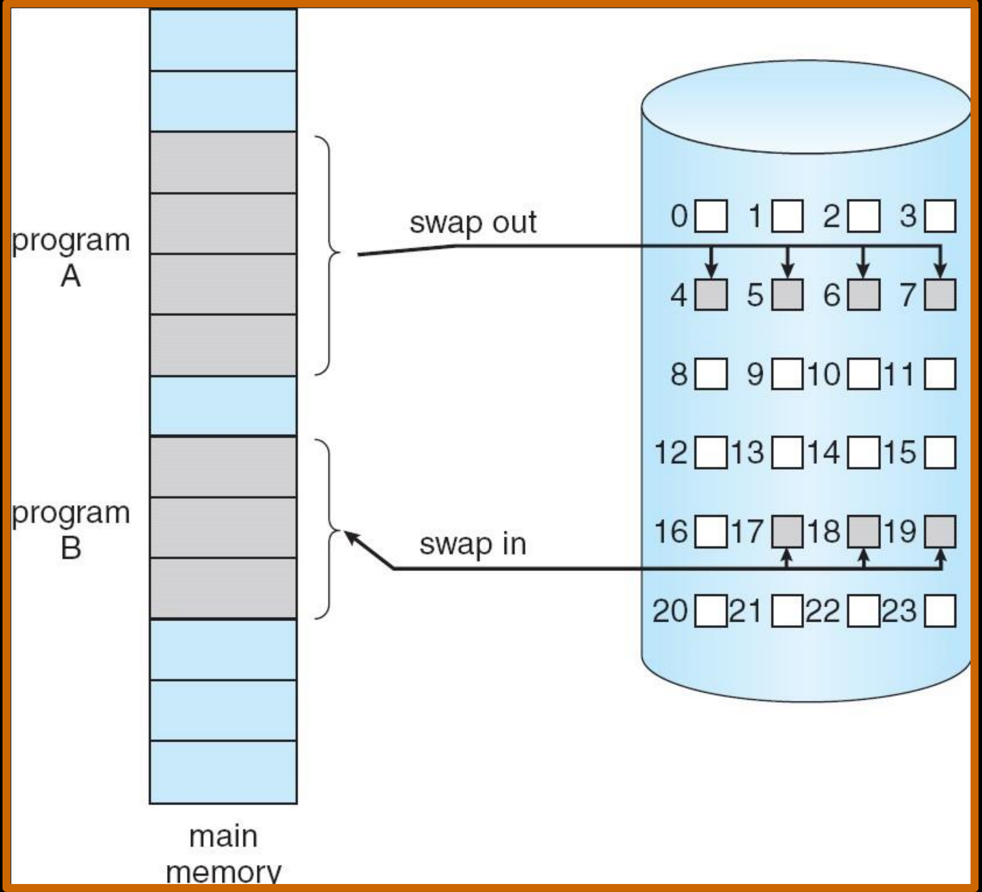


Silberschatz, Galvin Operating System Concepts – 7th Edition and Gagne ©2005

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

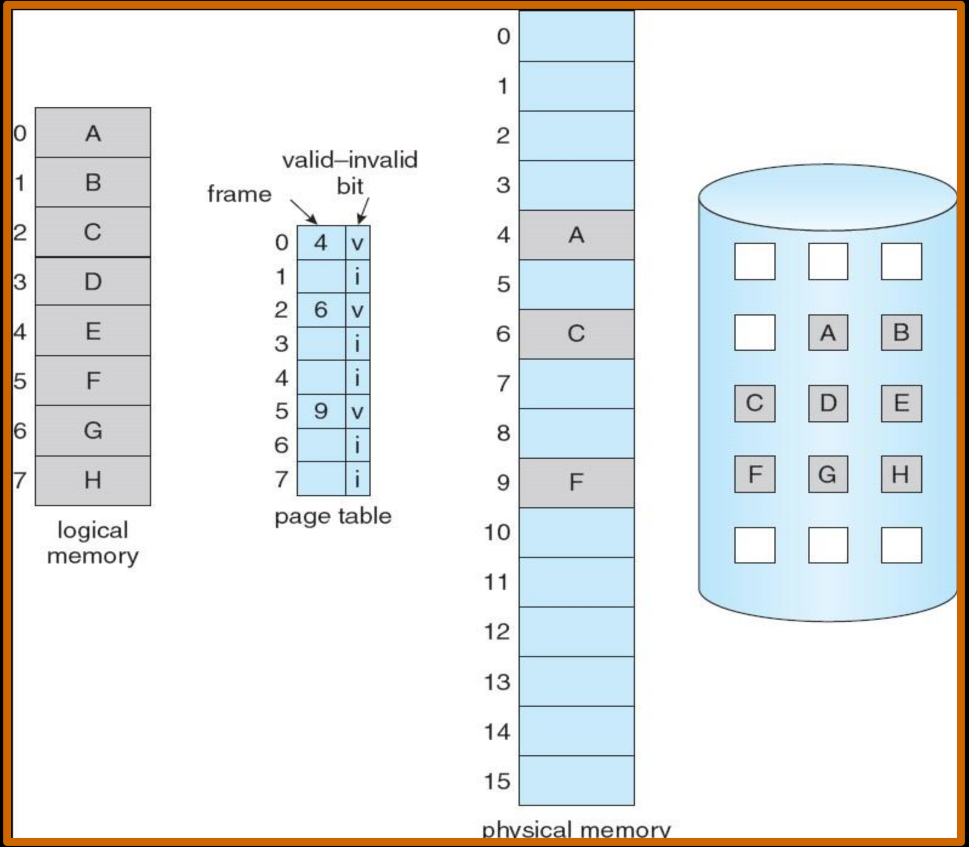
- With each page table entry a valid–invalid bit is associated (**v** ⇒in-memory, **i** ⇒not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

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 (a trap to the OS)

PAGE TABLE WHEN SOME PAGES ARE NOT IN MAIN MEMORY



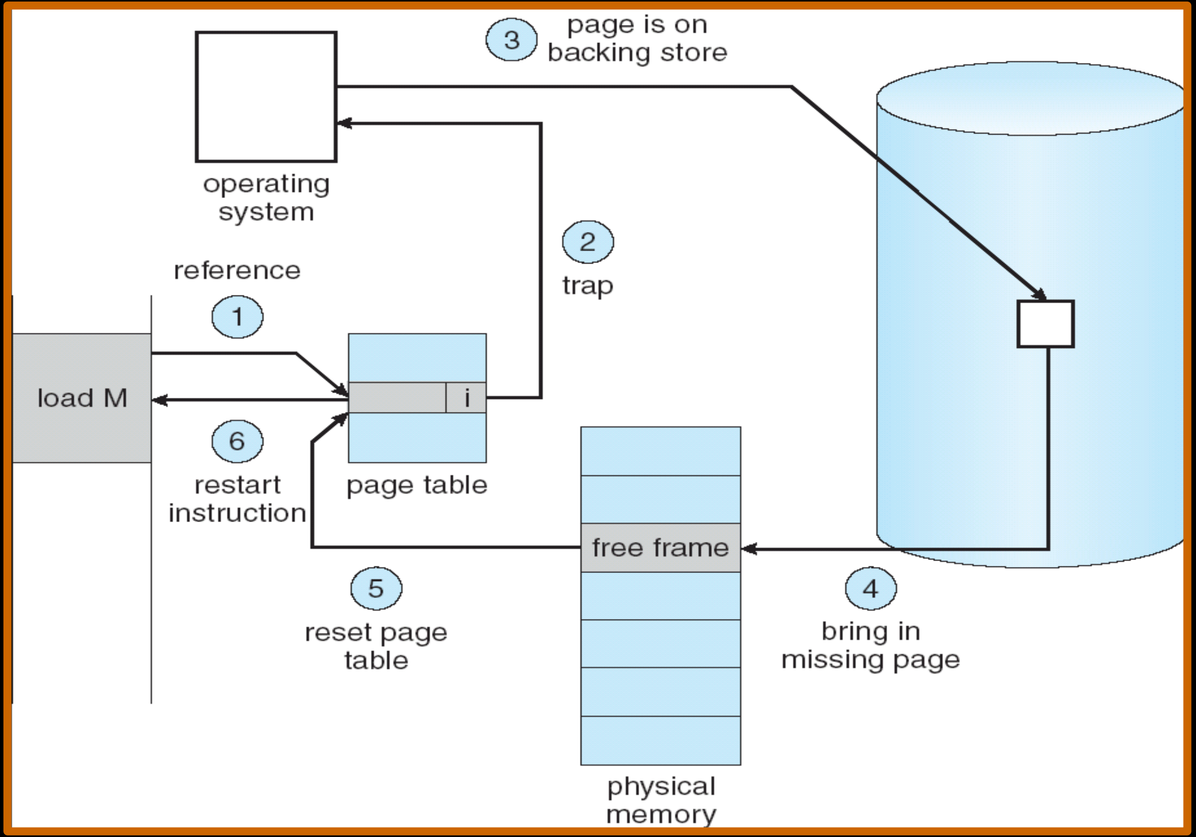
PAGE FAULT

- 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



Silberschatz, Galvin Operating System Concepts – 7th Edition and Gagne ©2005

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)

EAT = (1 – *p*) x memory access
+ *p* (page fault overhead
+ swap page out
+ swap page in
+ restart overhead
)

PROCESS CREATION

- Virtual memory allows other benefits during process creation:

  - Copy-on-Write

  - Memory-Mapped Files (later)

COPY-ON-WRITE

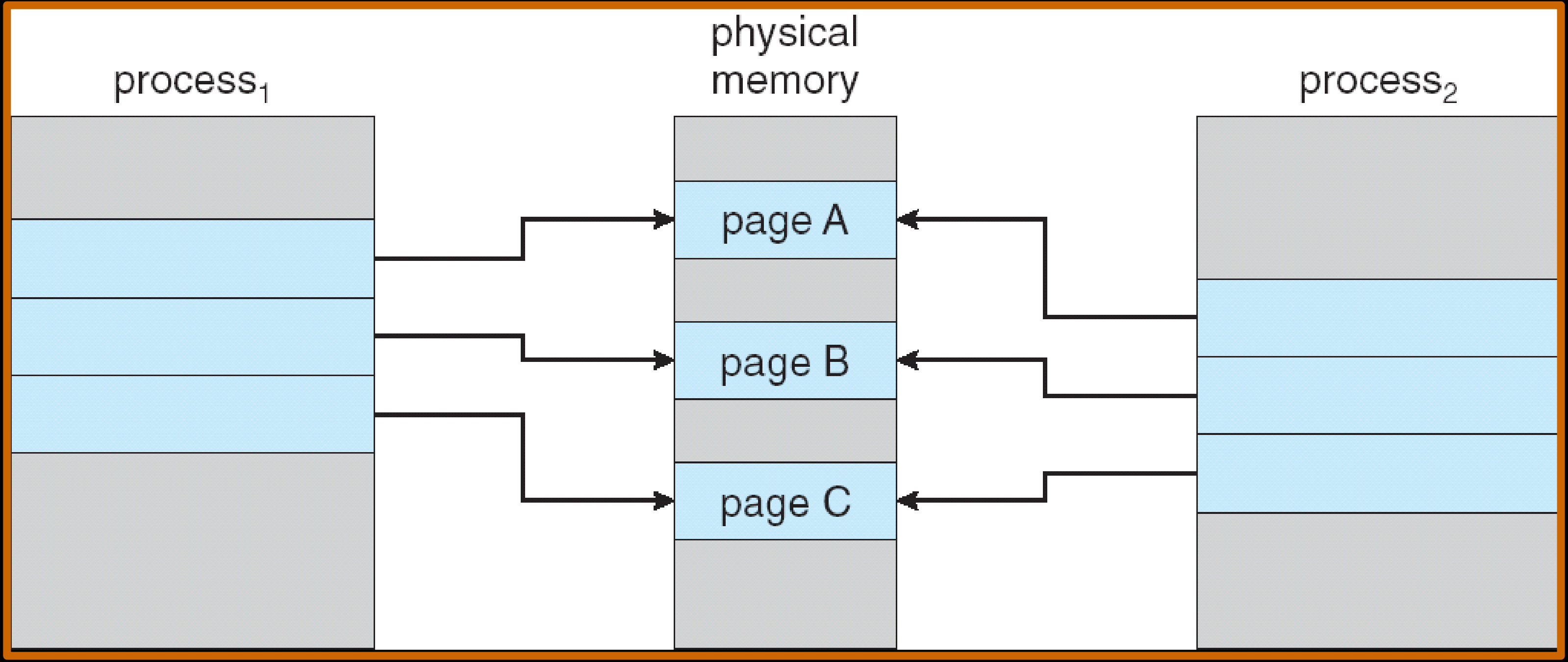
- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory

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

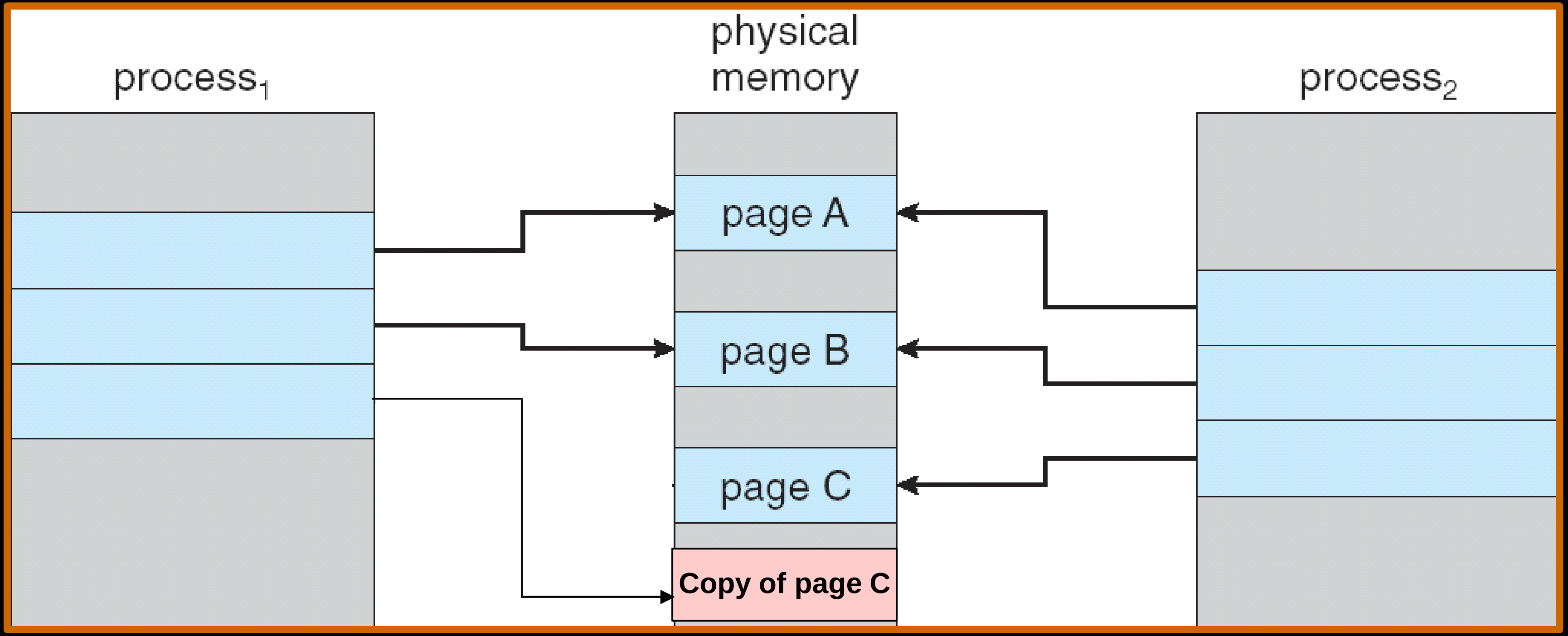
- COW allows more efficient process creation as only modified pages are copied

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

BEFORE PROCESS 1 MODIFIES PAGE C



AFTER PROCESS 1 MODIFIES PAGE C



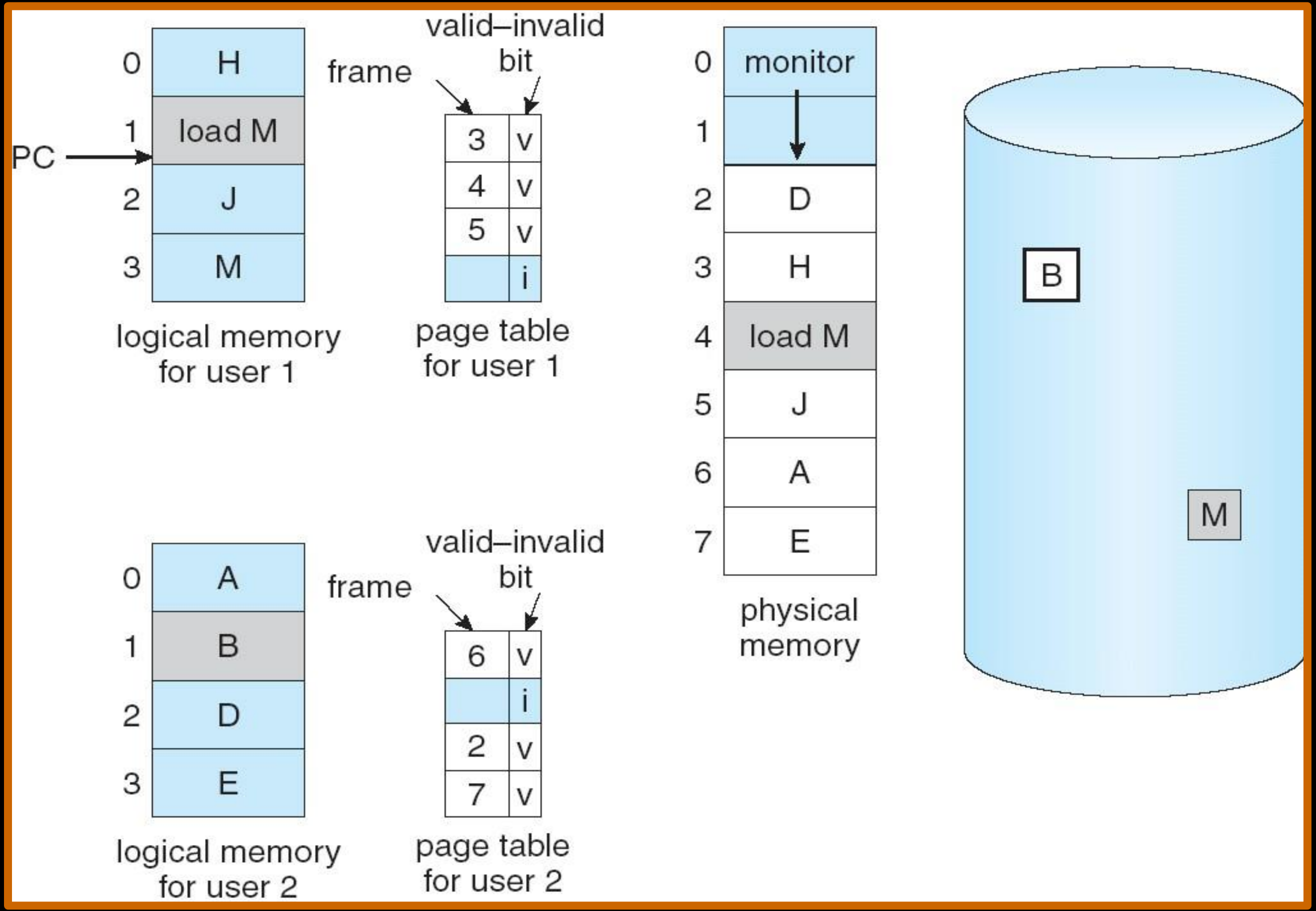
WHAT HAPPENS IF THERE IS NO FREE FRAME?

- 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

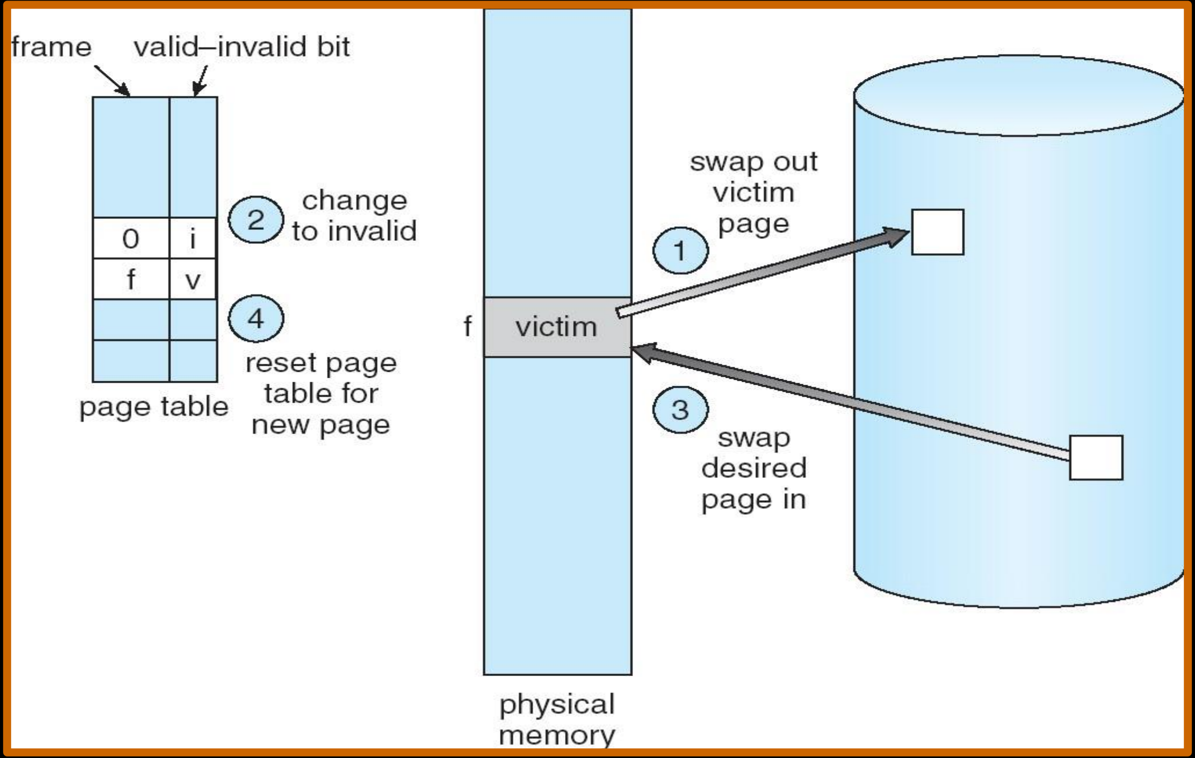
1. Find the location of the desired page on disk

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

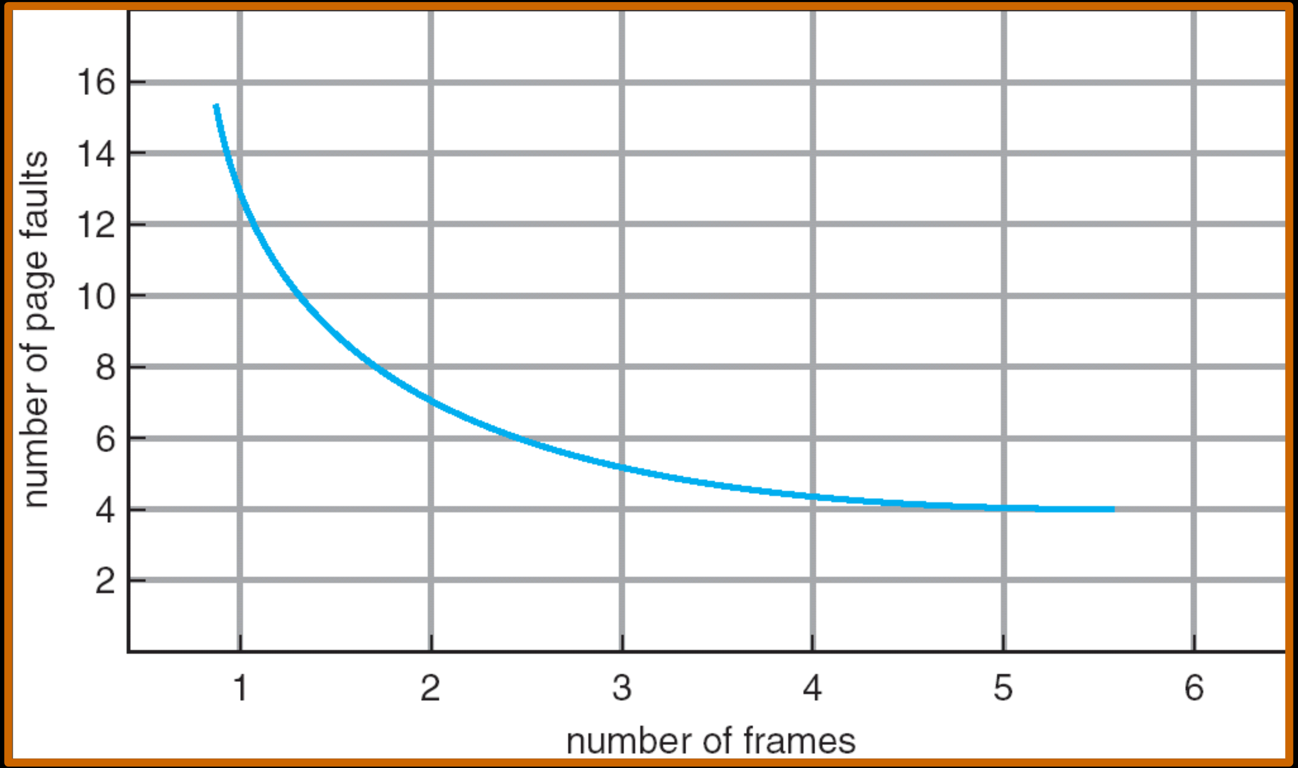
- Want lowest page-fault rate

- 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

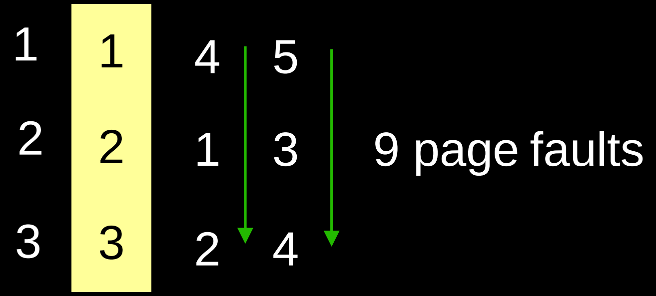
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

GRAPH OF PAGE FAULTS VERSUS THE NUMBER OF FRAMES

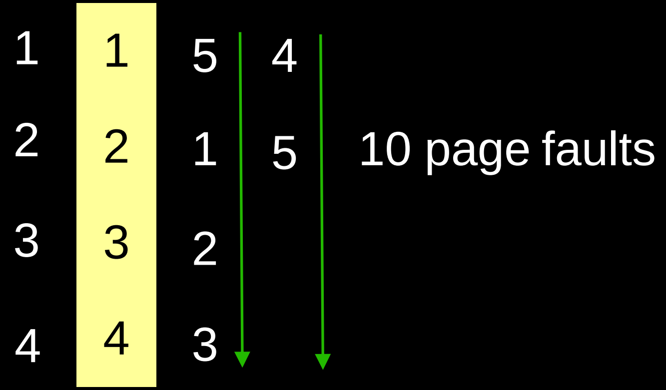


FIFO PAGE REPLACEMENT

- 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)

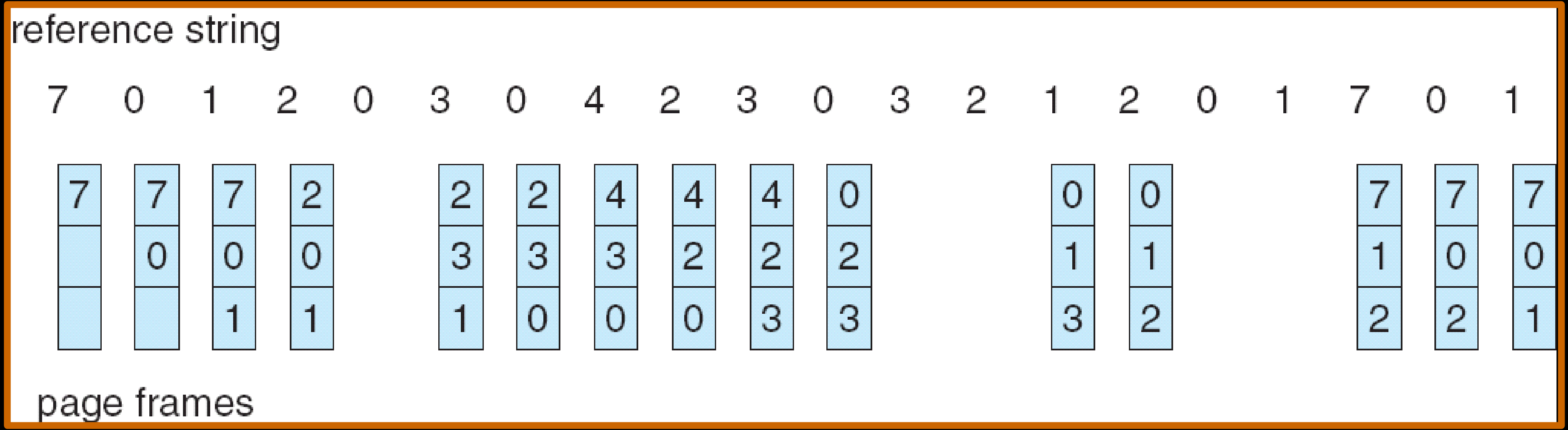


- 4 frames



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

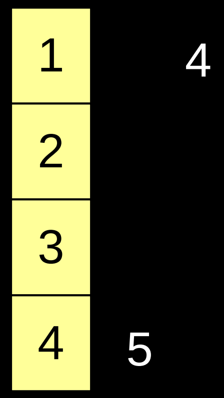
FIFO PAGE REPLACEMENT



OPTIMAL ALGORITHM

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

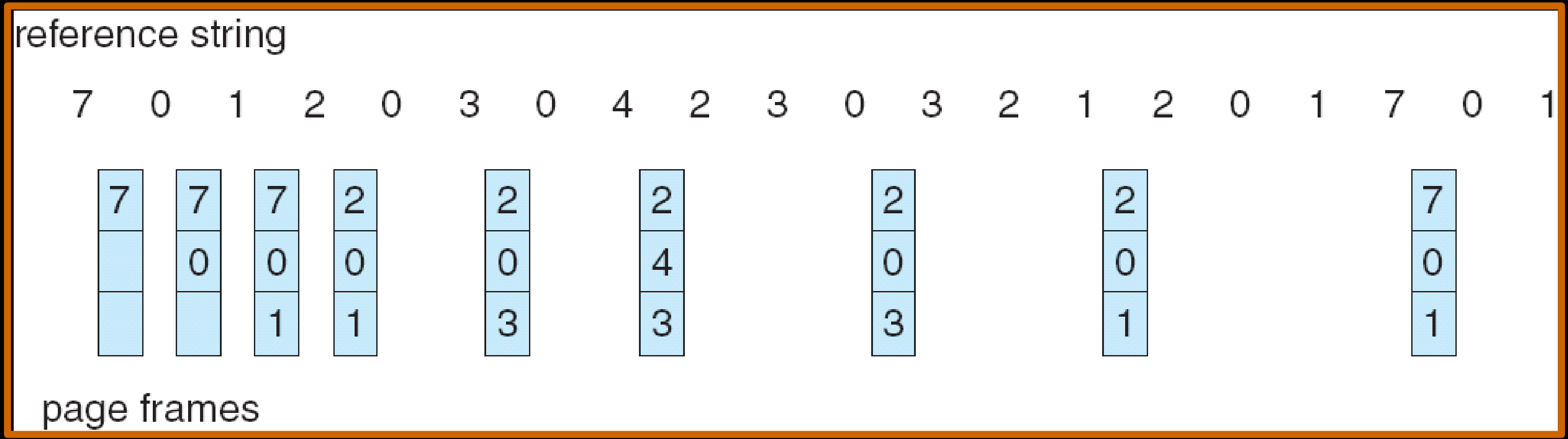
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



6 page faults

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

OPTIMAL PAGE REPLACEMENT



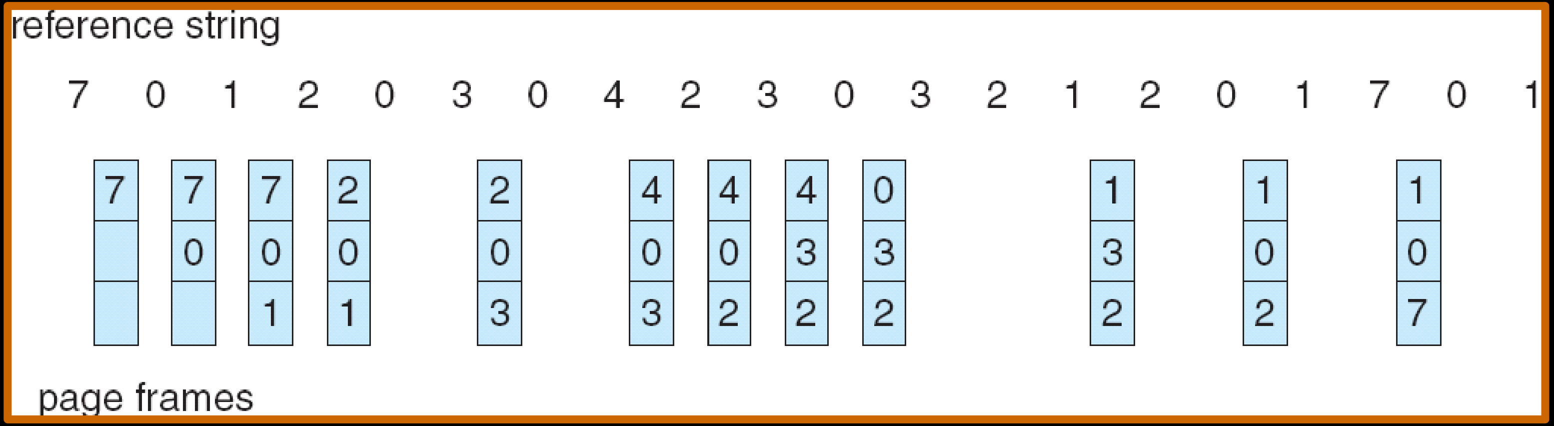
LEAST RECENTLY USED (LRU) ALGORITHM

- 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

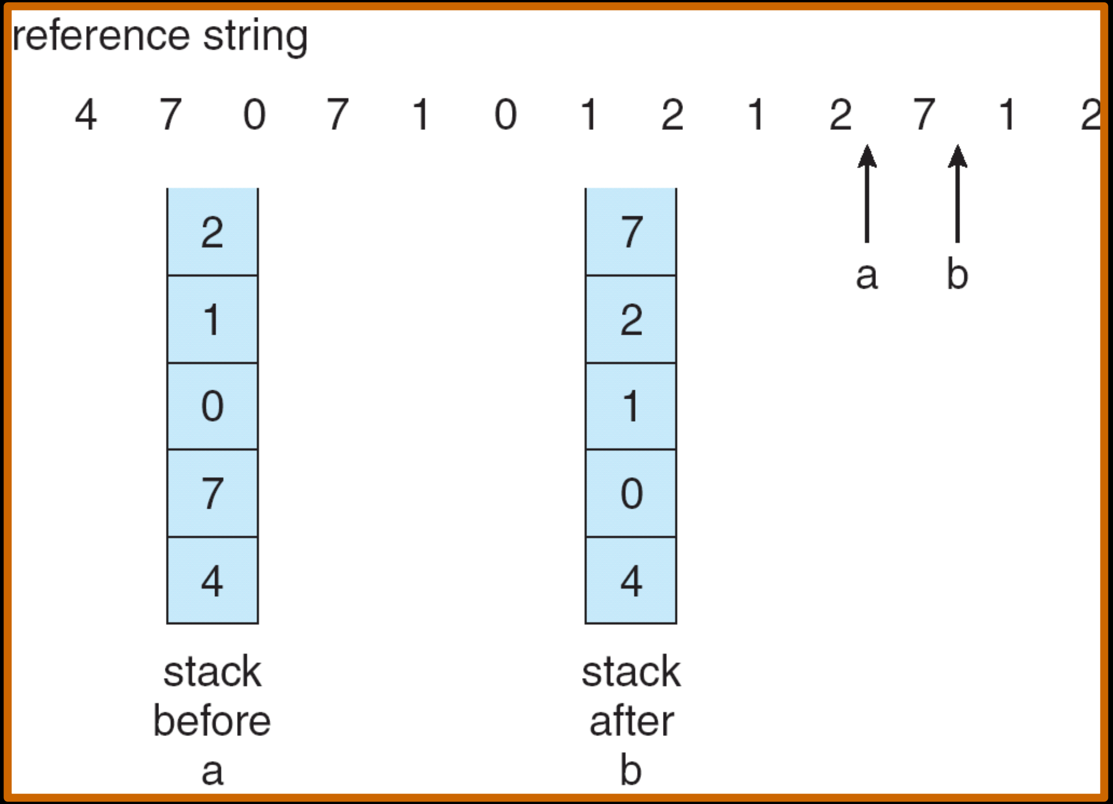
LRU PAGE REPLACEMENT



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

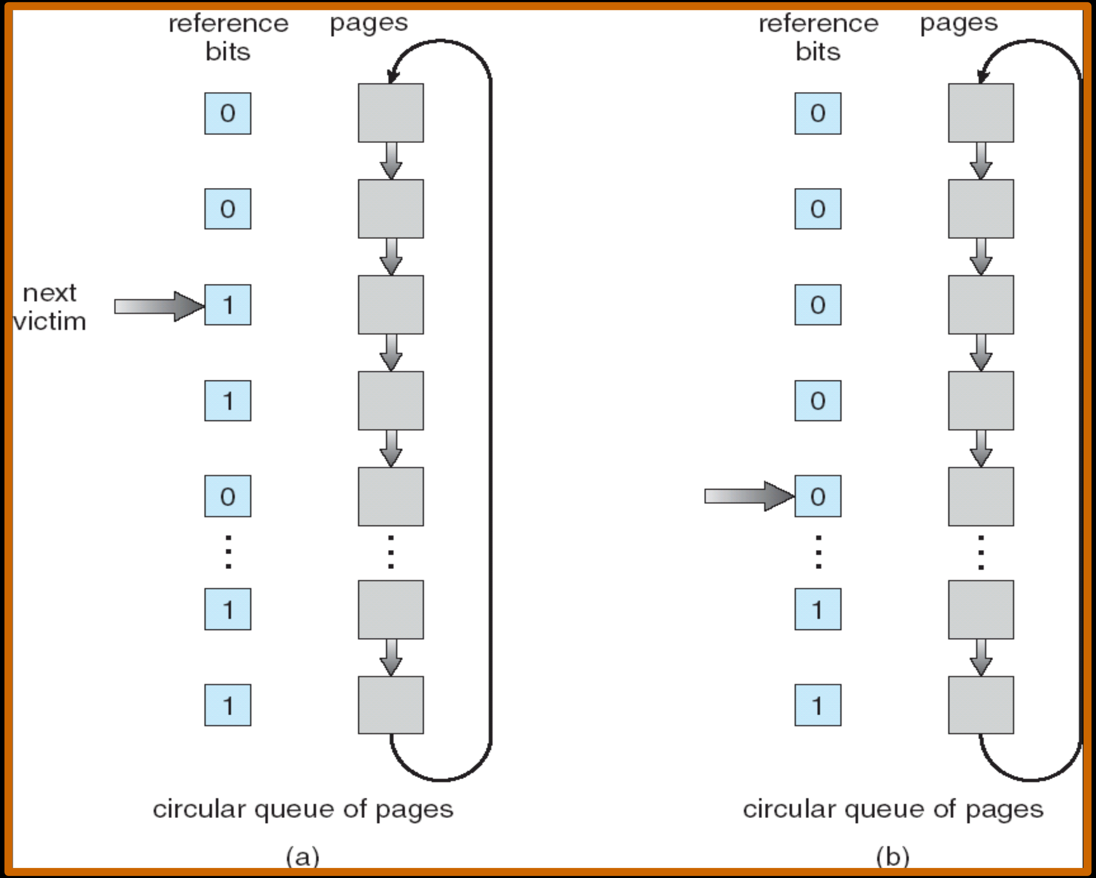


LRU APPROXIMATION ALGORITHMS

- 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

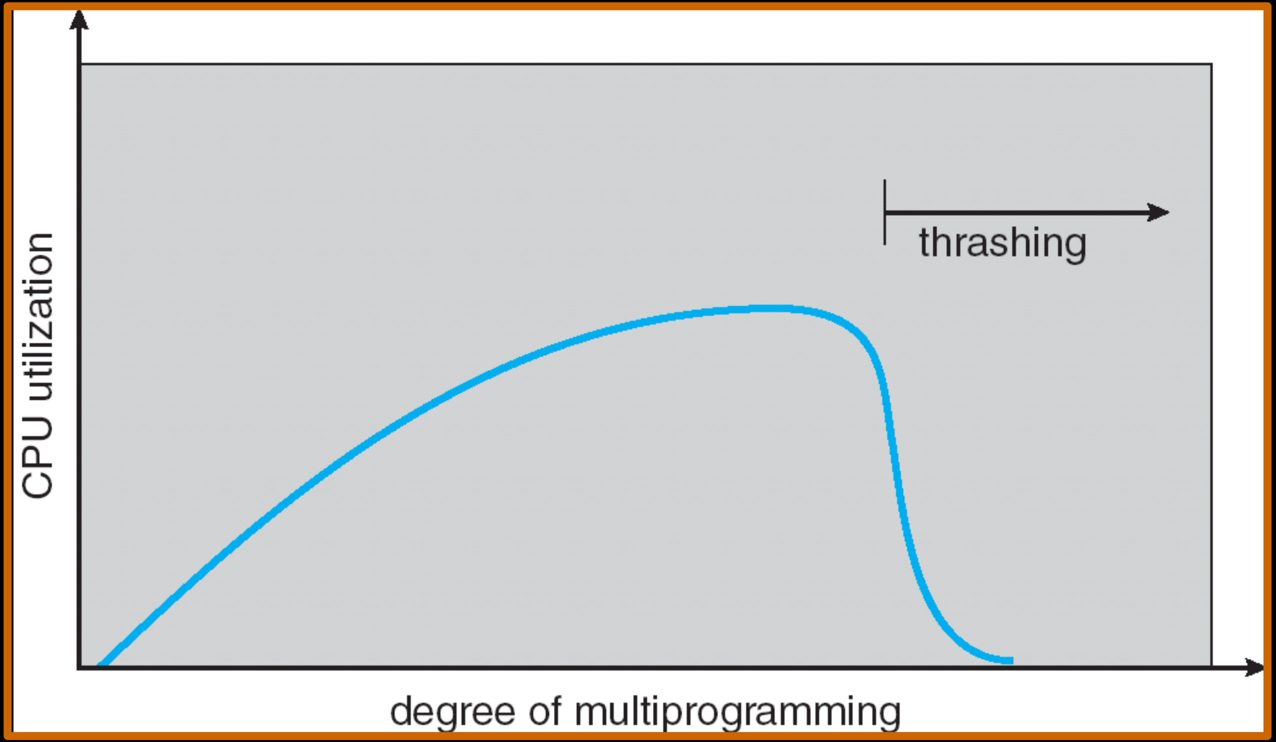
- Problem with global replacement: unpredictable page-fault rate. Cannot control its own page-fault rate. More common
- Problem with local replacement: free frames are not available for others. – Low throughput

THRASHING

- 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

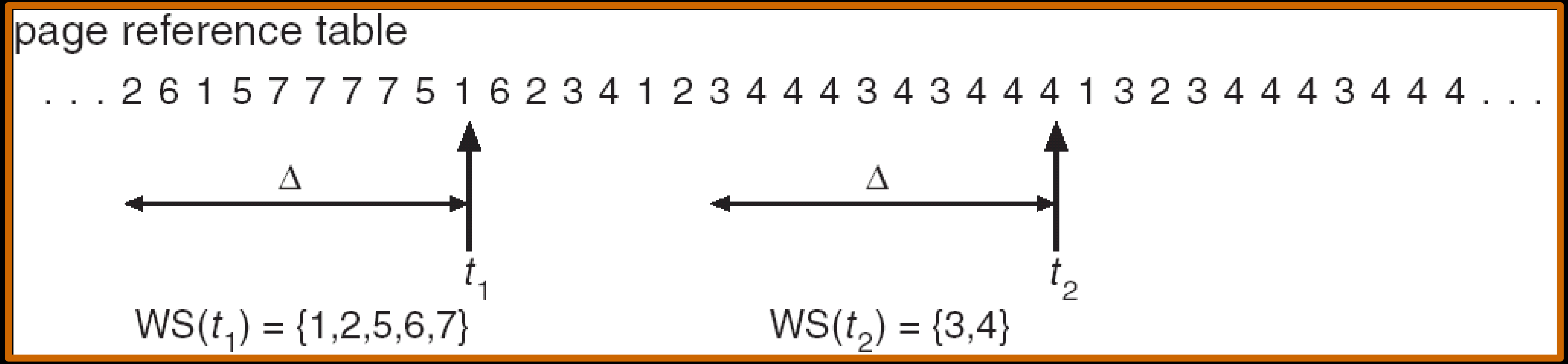
- To limit the effect of thrashing: local replacement algo cannot steal frames from other processes. But queue in page device increases effective access time.

- To prevent thrashing: allocate memory to accommodate its locality

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

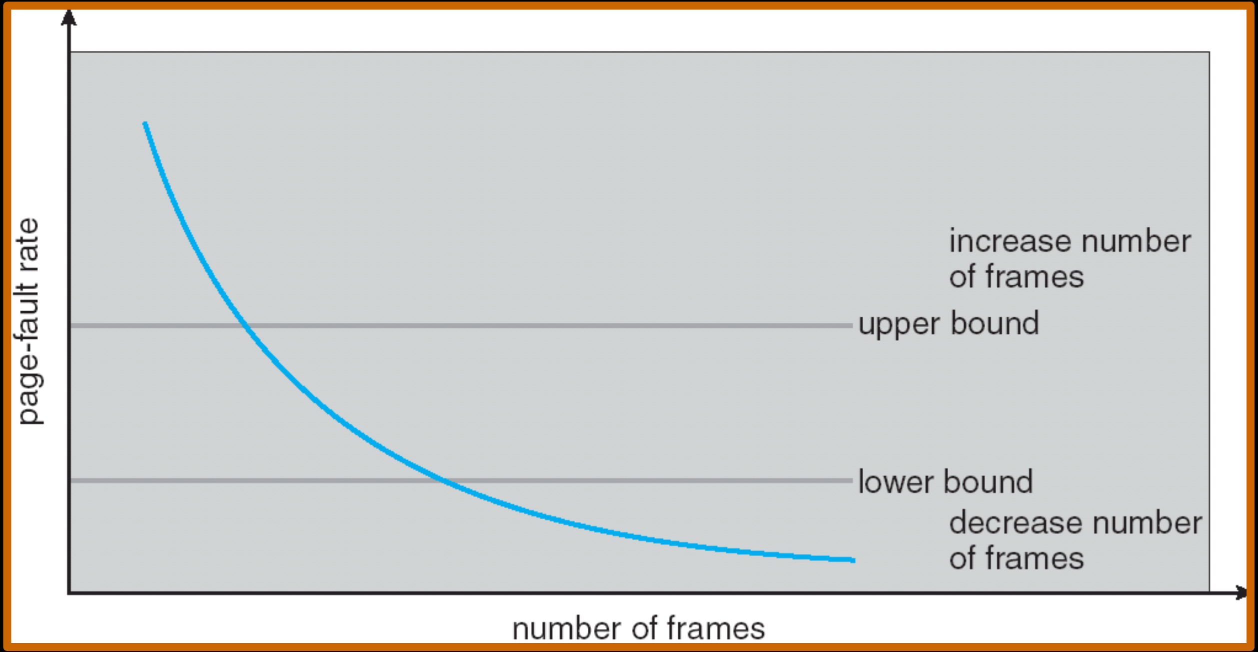


KEEPING TRACK OF THE WORKING SET

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10{,}000$
  - 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 ⇒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

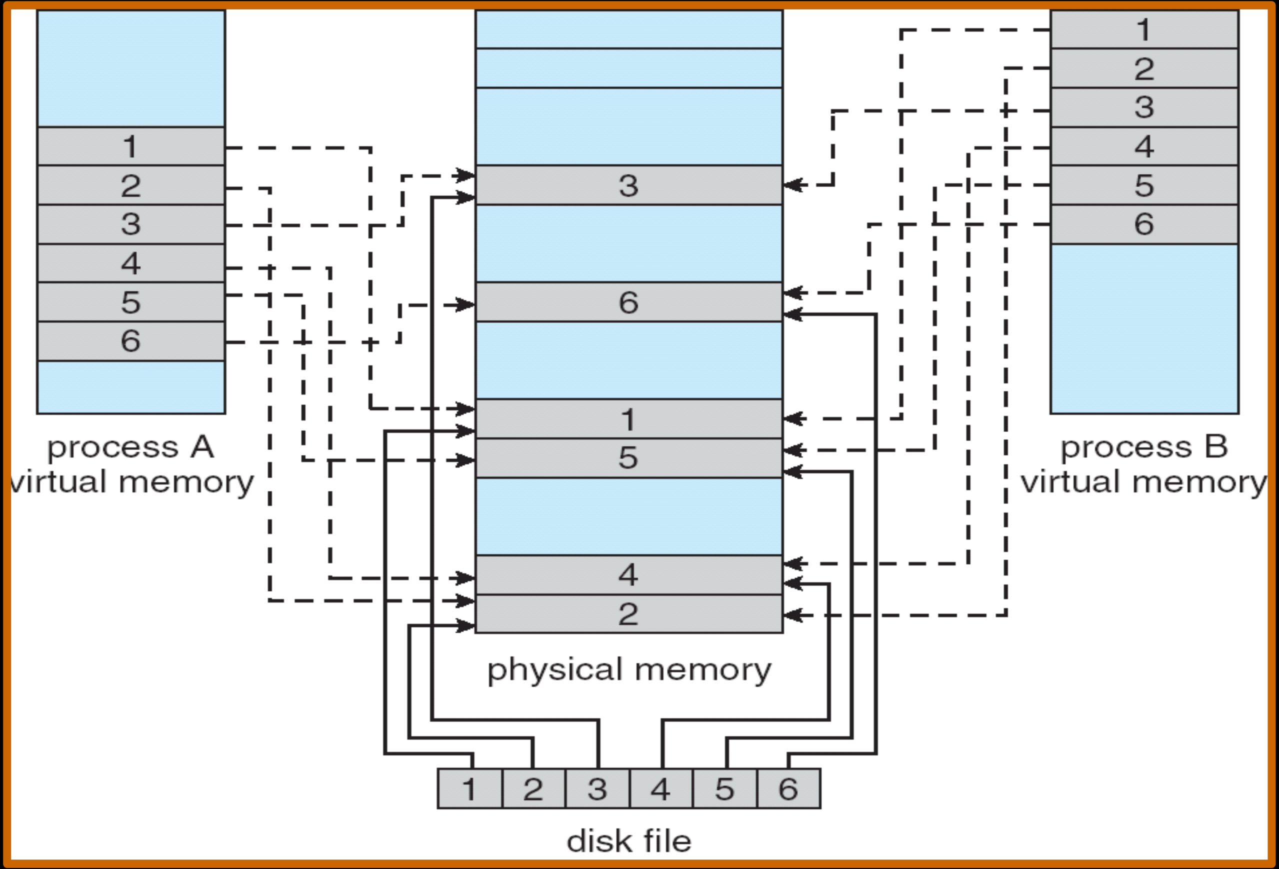
- Establish “acceptable” page-fault rate for each process
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



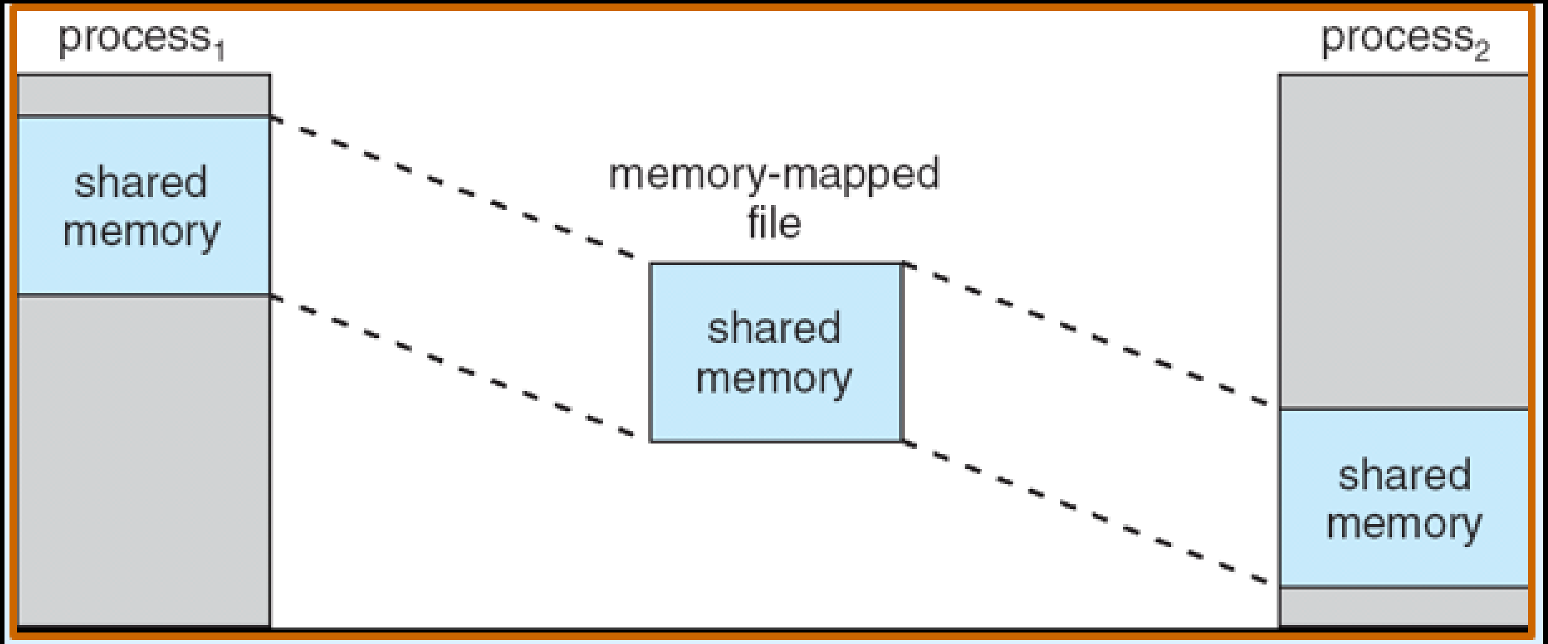
MEMORY-MAPPED FILES

- 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



ALLOCATING KERNEL MEMORY

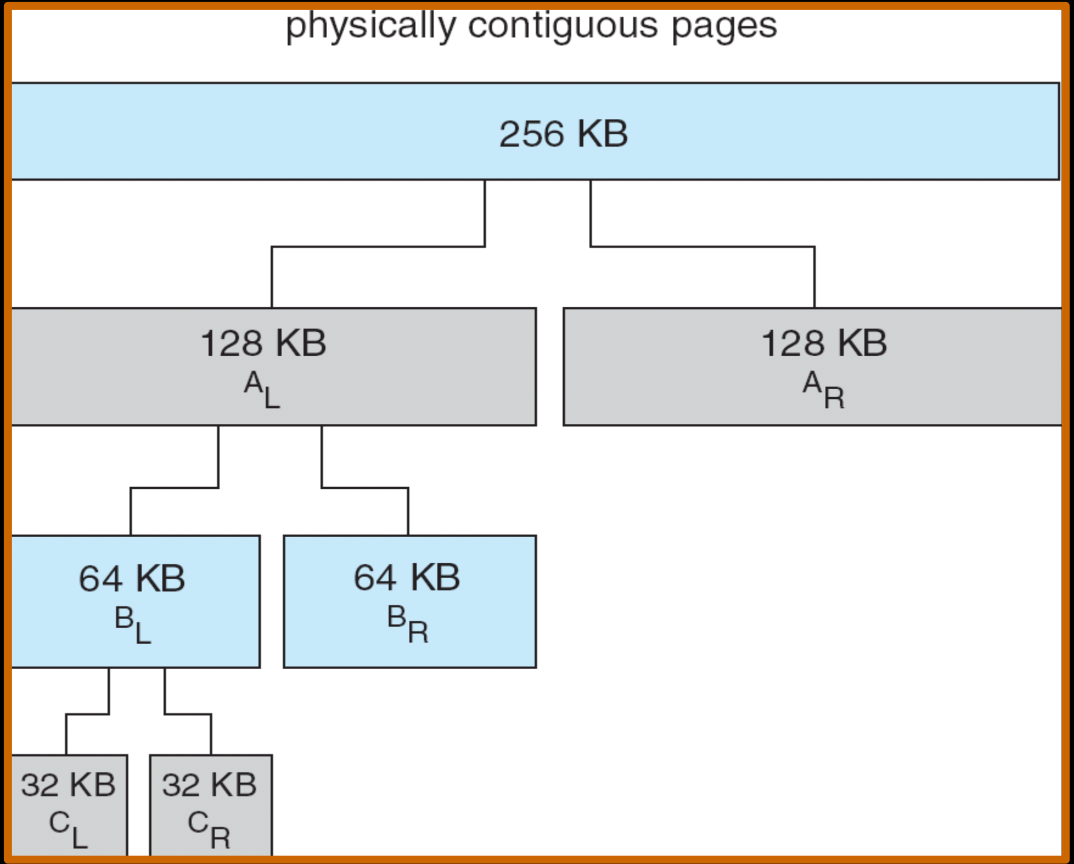
- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes – needs to reduce fragmentation
  - Some kernel memory needs to be contiguous (certain h/w device interacts with contiguous physical memory)

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

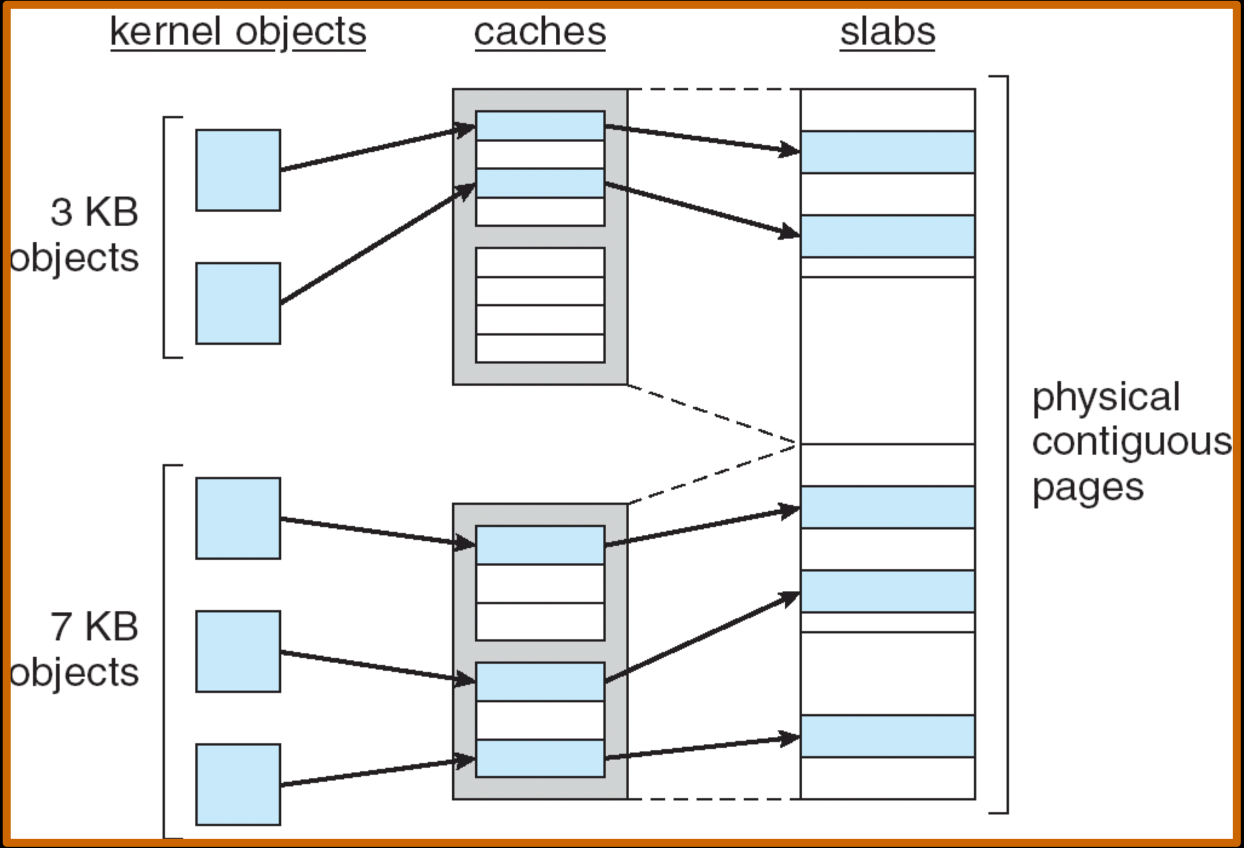
BUDDY SYSTEM ALLOCATOR



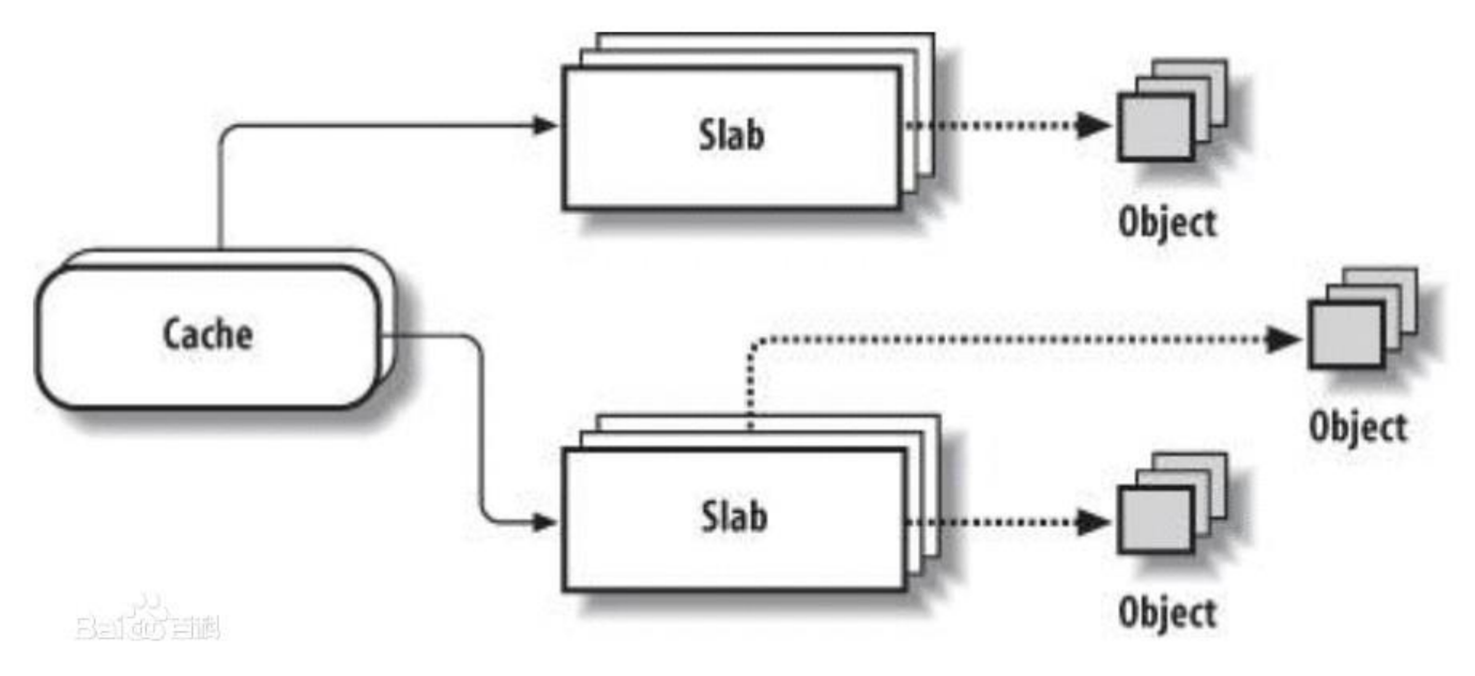
SLAB ALLOCATOR

- 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




LUT
University