Virtual Memory

Operating Systems (CS-2006)
SPRING 2022, FAST NUCES

COURSE SUPERVISOR: ANAUM HAMID
anaum.hamid@nu.edu.pk

Virtual Memory

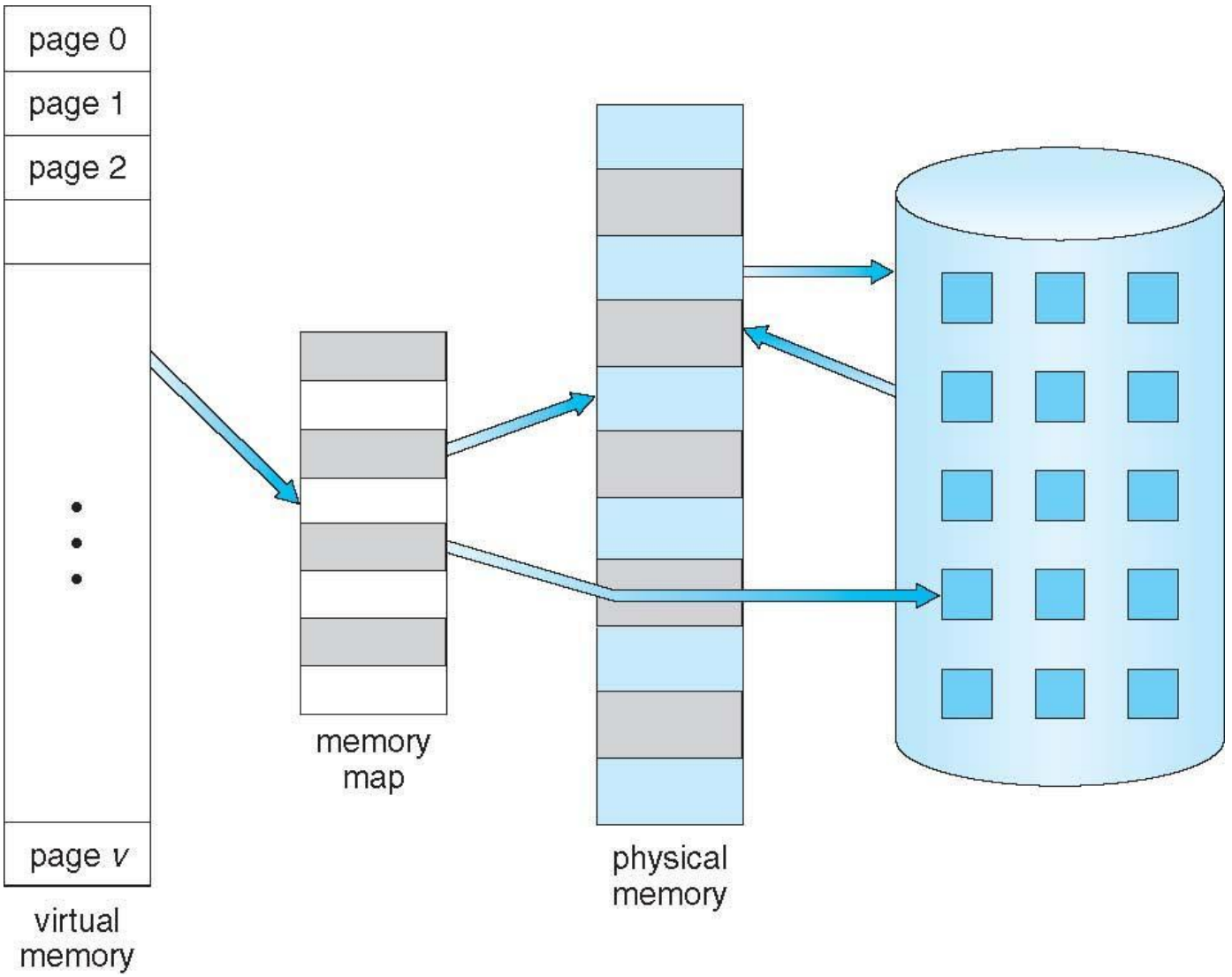
- **Virtual memory** is a **memory** management capability of an OS, uses hardware and software to allow a computer to compensate for physical **memory** shortages by temporarily transferring data from random access **memory** (RAM) to disk storage.
- Virtual memory is not RAM (Random access memory) on memory chips.
- Virtual memory is an area of hard drive or other storage space, which OS uses as swap space (overflow) for the physical RAM.
- When the demands of the software and data exceed the physical RAM size, the operating system copies to hard disk blocks of data from RAM that have been seldom used and releases that RAM for use by the current process.
- When the block (or Page) is required again, the OS makes room in RAM by copying another block to hard drive and copying in the data from the first block.

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
  - More programs running concurrently
  - Less I/O needed to load or swap processes.

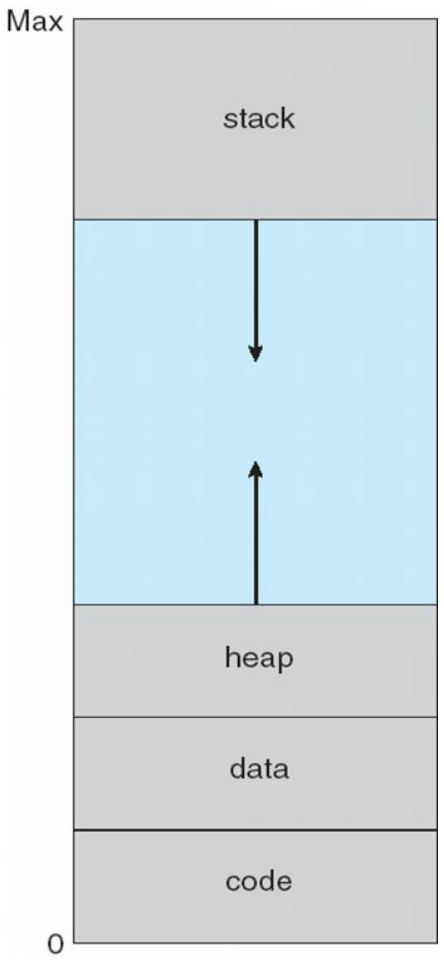
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory

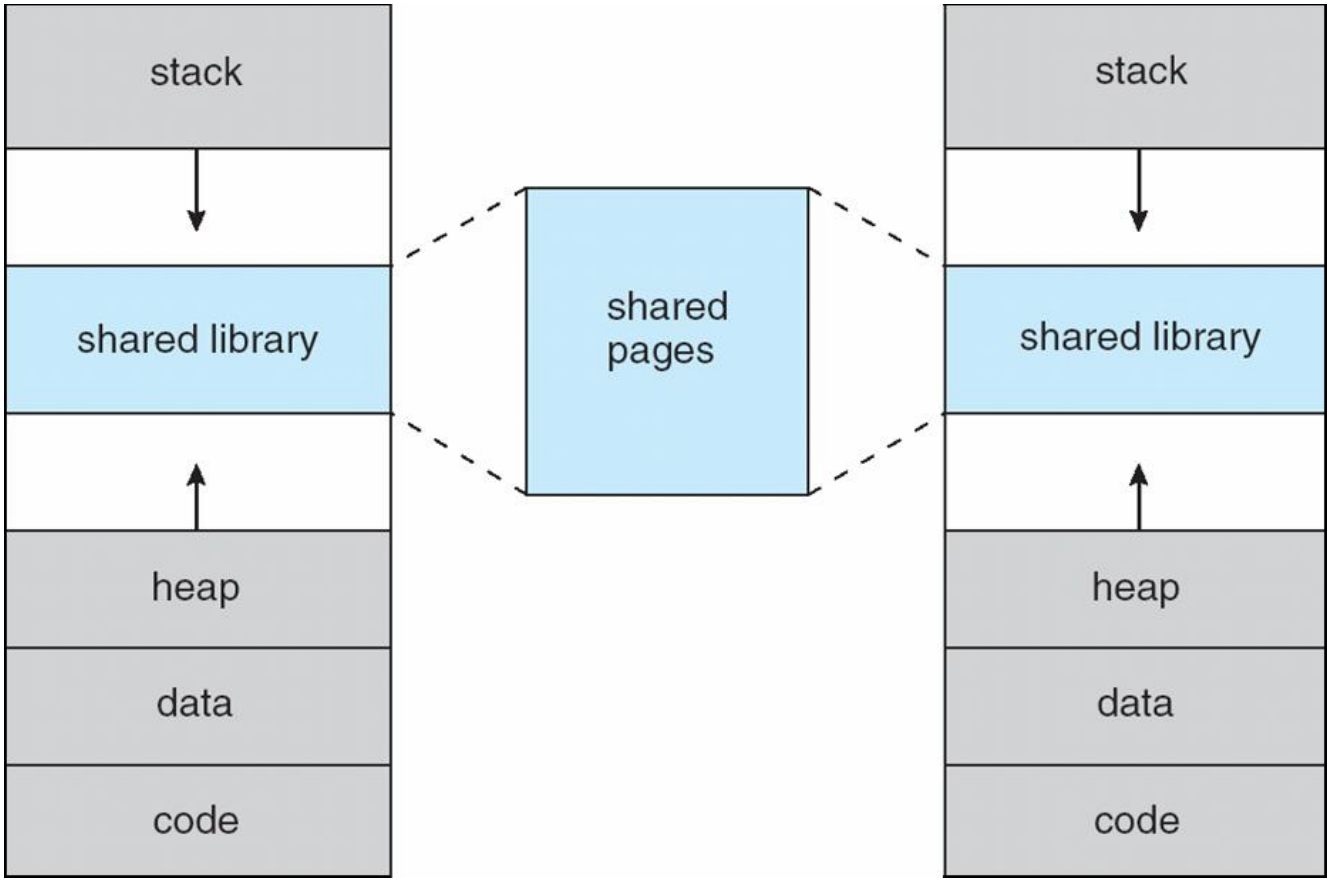


Virtual-address Space

- Virtual address spaces that include holes are known as sparse address spaces.
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during fork(), speeding process creation

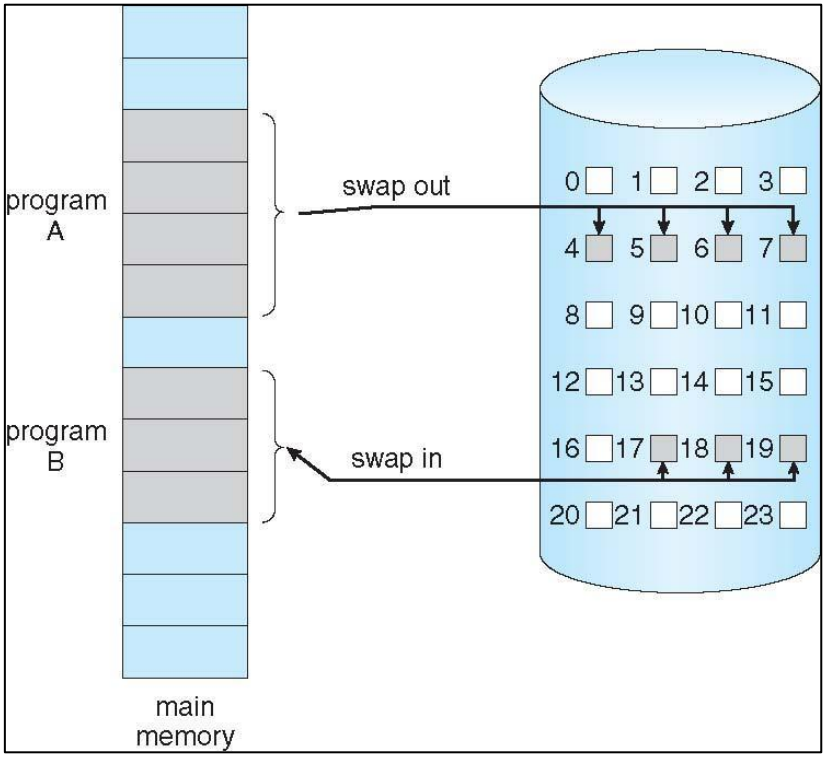


Shared Library Using Virtual Memory



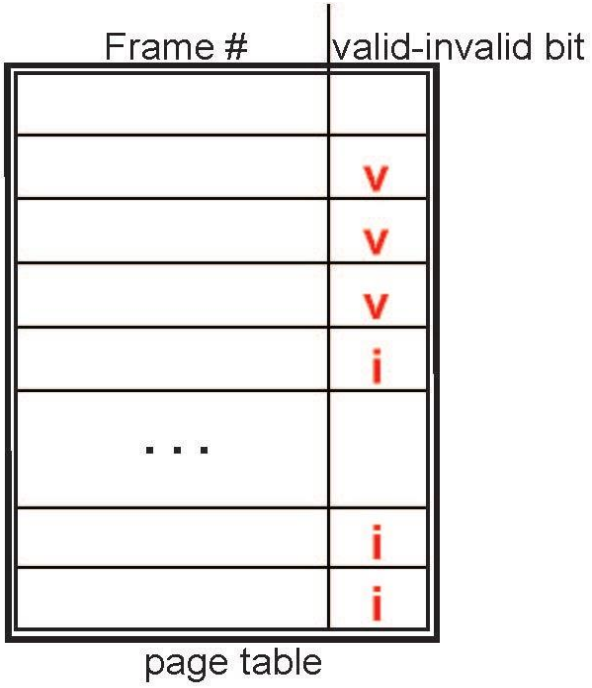
Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - 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



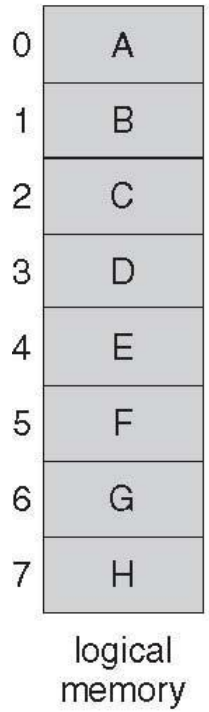
Valid-Invalid Bit

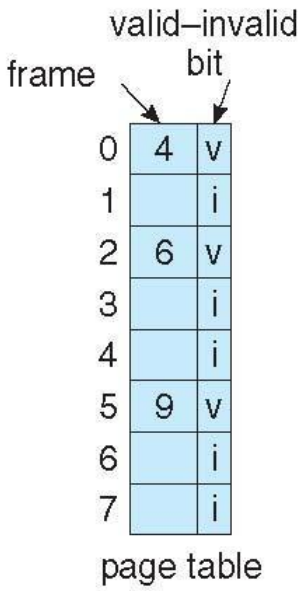
- With each page table entry, a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

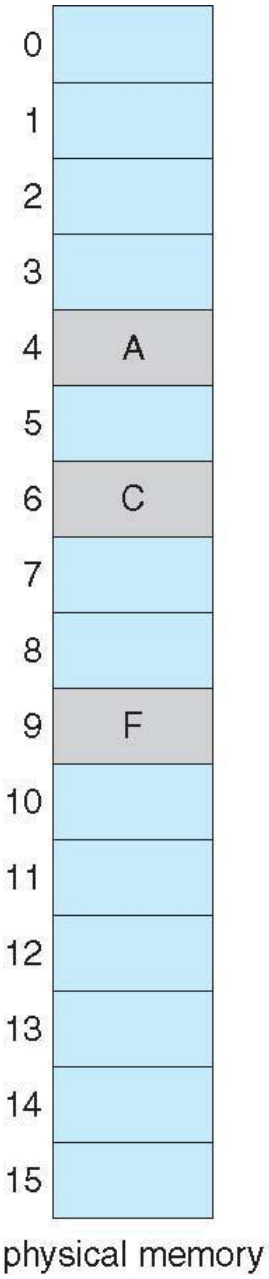


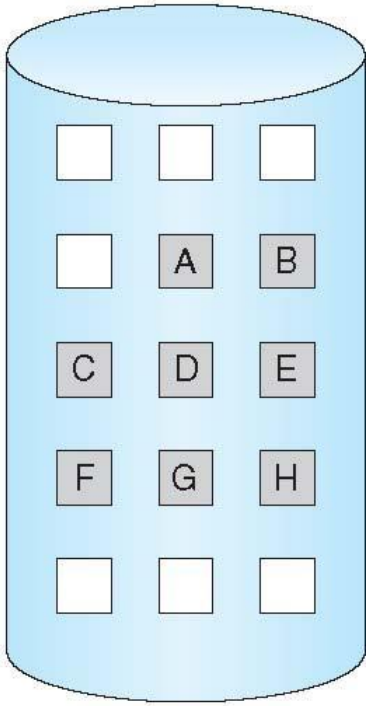
- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

Page Table When Some Pages Are Not in Main Memory









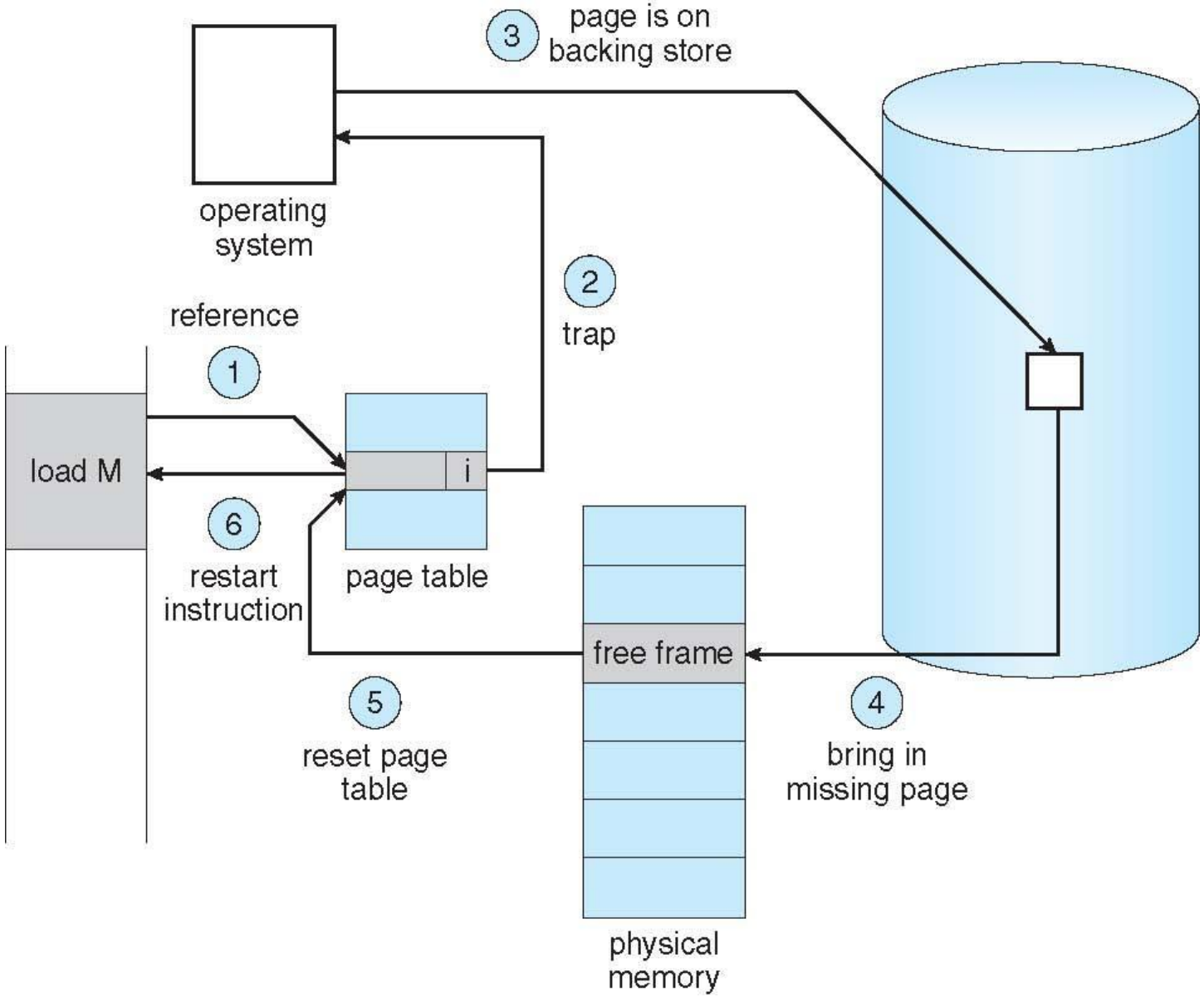
Page Fault

- 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 to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
   Set validation bit = **v**
5. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging**
- A given instruction could access multiple pages -> multiple page faults
  - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

Performance of Demand Paging

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

- Effective Access Time (EAT) =

  EAT = (1 – *p*) x ma + *p* (page fault time)

  page fault time = fault overhead+ swap page out+ swap page in

Page fault (EAT calculation)

- Memory access time = ma = 200 nanoseconds
- Average page-fault service time = **page fault time =** 8 milliseconds
- If one access out of 1,000 causes a page fault, then

  p = 1/1000 = 0.001
- **EAT = (1 – *p*) x ma + *p* (page fault time)**
- EAT = (1 – 0.001) x 200 + 0.001 x (8e -6)
- EAT = 8199.8 nanoseconds
- EAT = 8.199 microseconds

Millisecond = 10 e-3

Microsecond = 10e-6

Nanoseconds = 10e-9

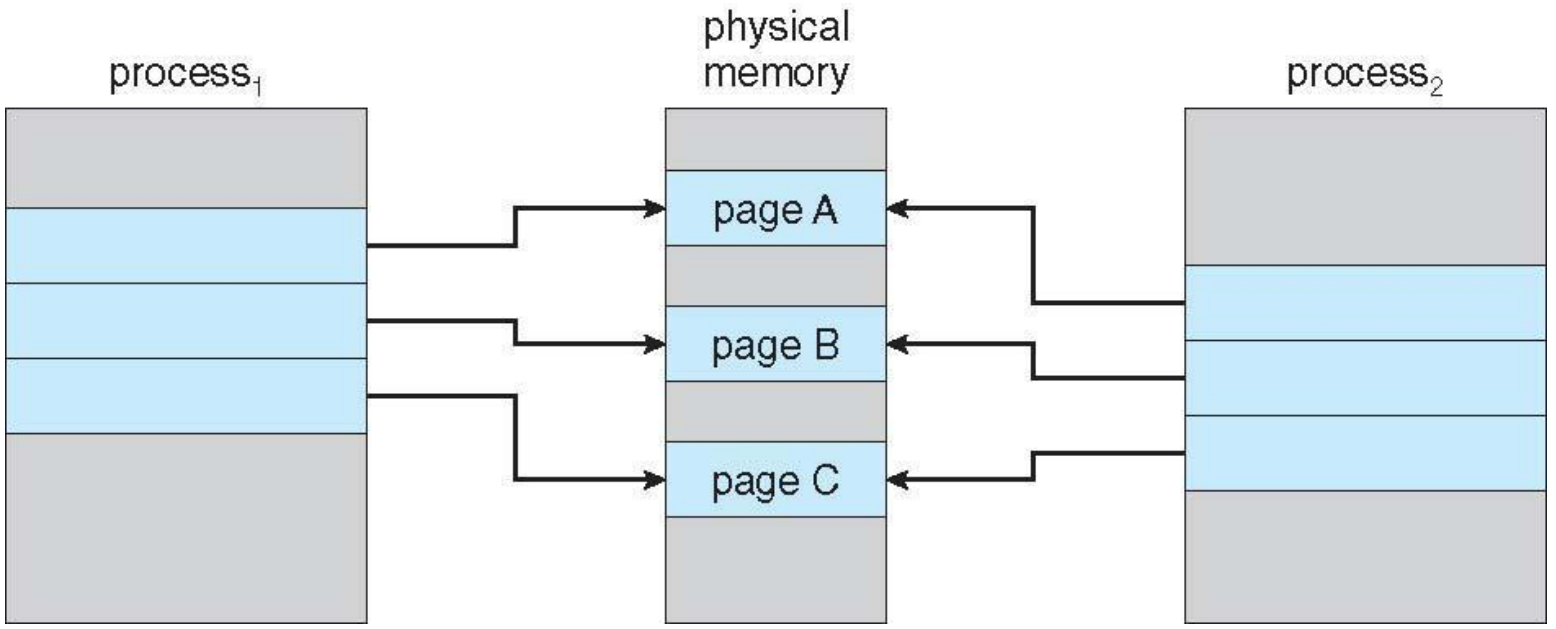
Demand Paging Optimizations

1. Copy entire process image to swap space at process load time
    1. Then page in and out of swap space
    2. Used in older BSD Unix
2. Demand page in from program binary on disk, Demand pages for such files are brought directly from the file system.
    1. Used in Solaris and current BSD

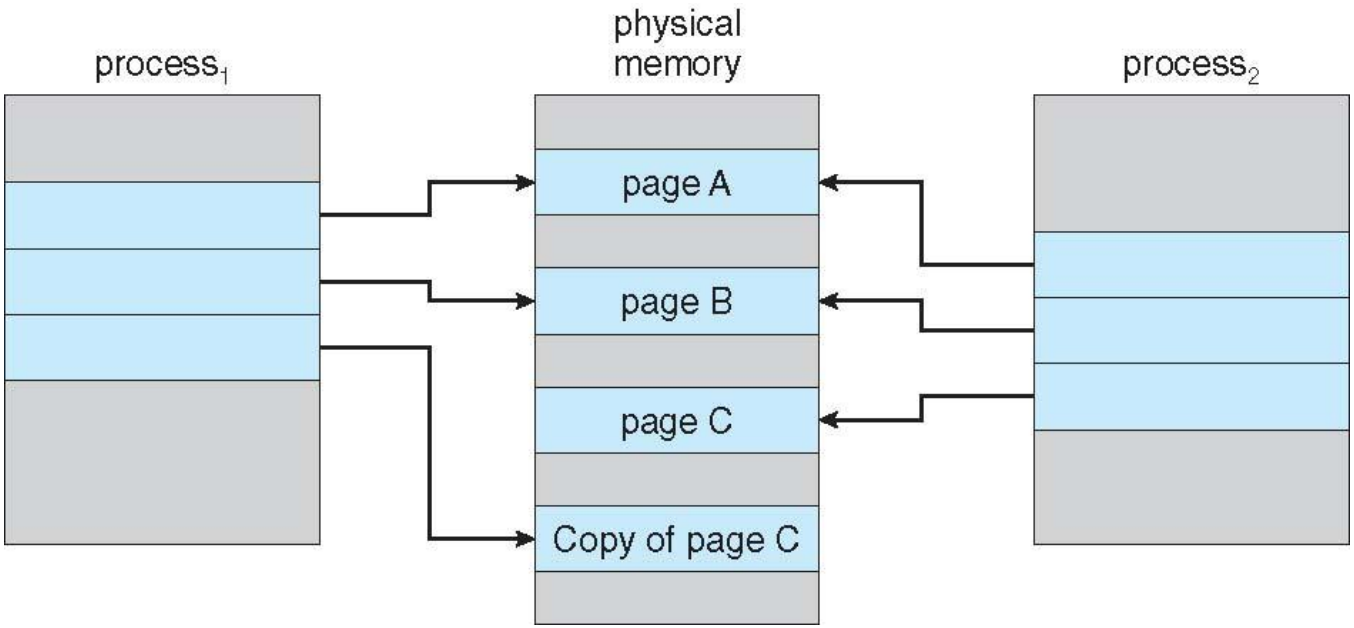
Copy-on-Write

- Copy-on-Write (COW) technique 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.
- In general, free pages are allocated from a pool of zero-fill-on-demand pages
- `vfork()` (for virtual memory fork) variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call `exec()`
  - Very efficient

Before Process 1 Modifies Page C



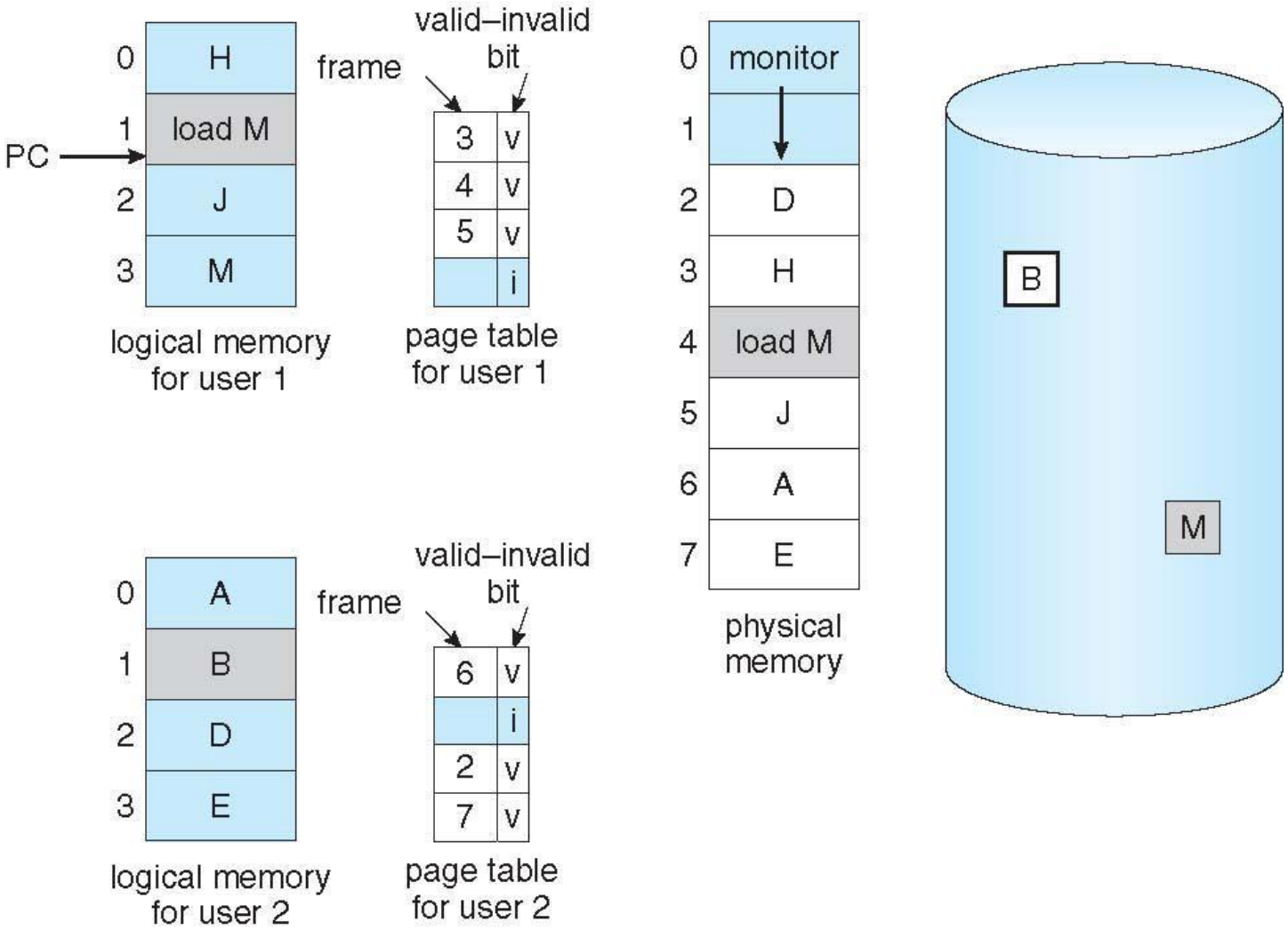
After Process 1 Modifies Page C



What Happens if There is no Free Frame?

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

Need For Page Replacement



Basic Page Replacement Steps

1. Find the location of the desired page on disk

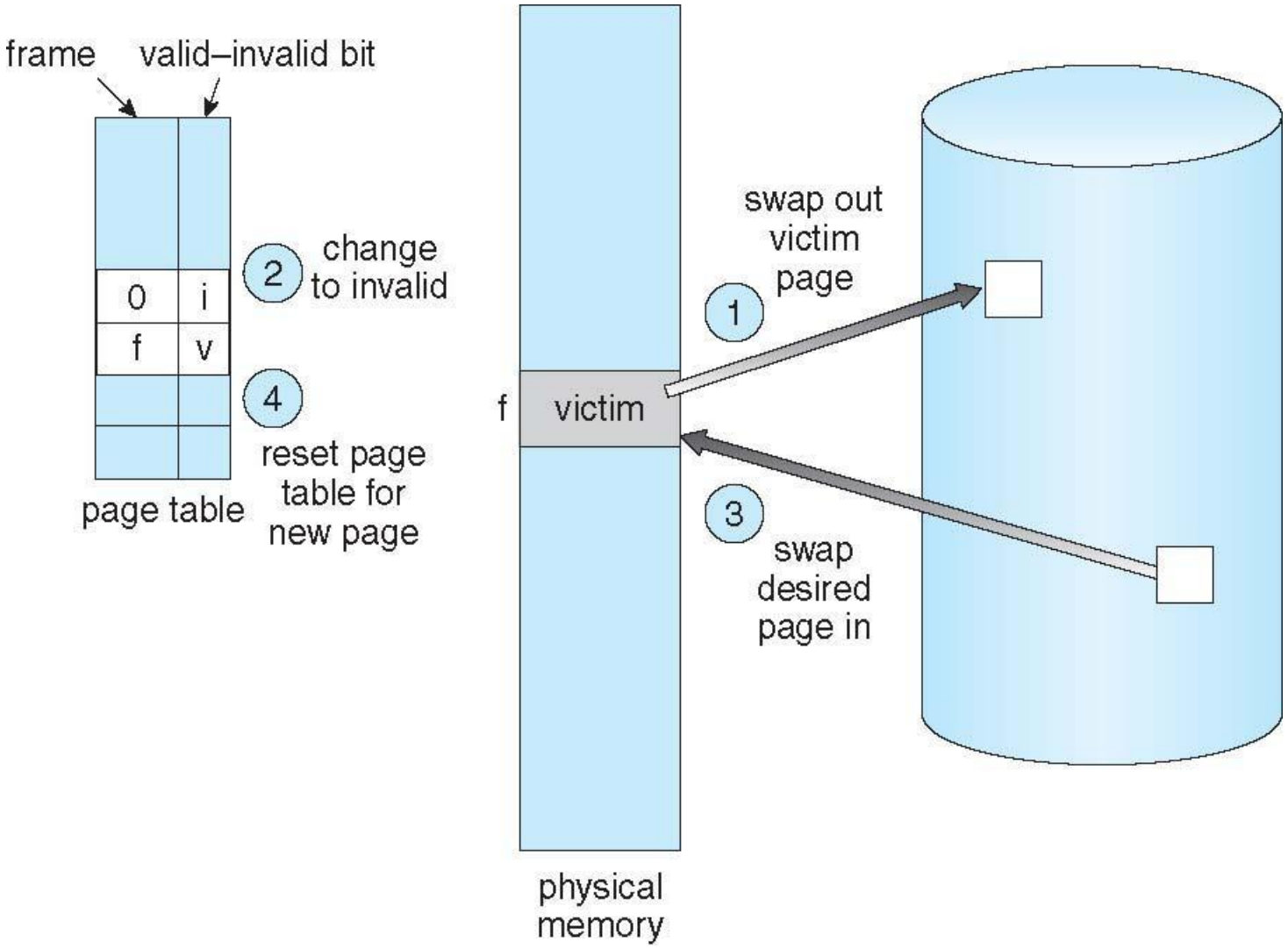
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a victim frame
     - Write victim frame to disk if dirty

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

4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2-page transfers for page fault – increasing EAT

Page Replacement



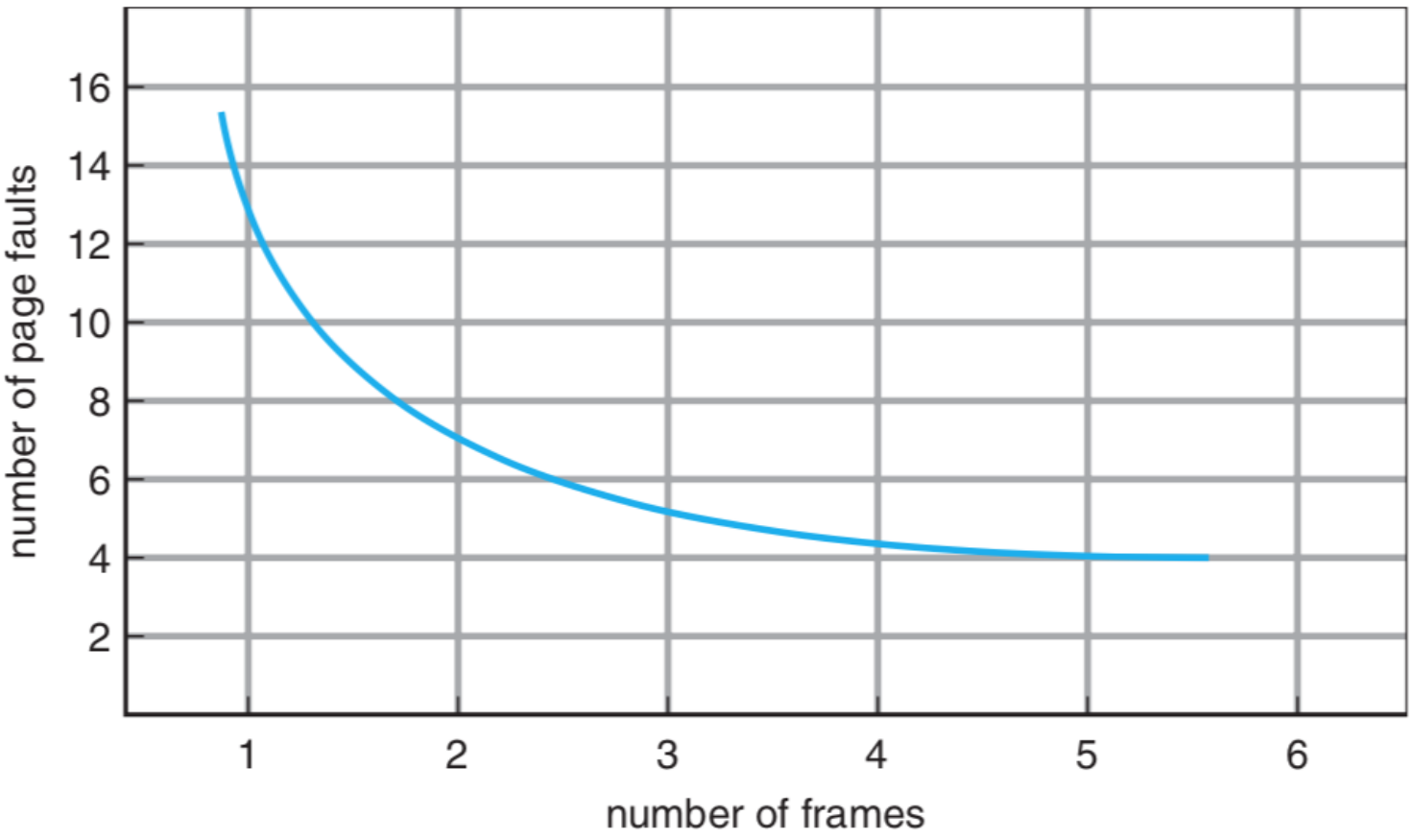
Page Replacement

- Page Replacement carried out by:

1. Page-replacement algorithm
2. Frame-allocation algorithm

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access

Graph of Page Faults Versus The Number of Frames



Page Replacement Algorithms

- Basic page replacement algorithms are:

1. FIFO
2. Optimal Solution
3. LRU Algorithm
4. LRU –Approximation

- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

6,0,1,2,0,3,0,5,2,3,0,3,0,3,2,1,2,0,1,6,0,1

First-In-First-Out (FIFO) Algorithm

- The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen.

Covered in Class

6,0,1,2,0,3,0,5,2,3,0,3,0,3,2,1,2,0,1,6,0,1

ratio	6	0	1	2	0	3	0	5	2	3	0	3	0	3	2	1	2	0	1	6	0	1
f1																						
f2																						
f3																						

Optimal Page Replacement Algorithm

- To Overcome Belady's anomaly, **optimal page-replacement algorithm introduced**—the algorithm that has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly.
- Such an algorithm does exist and has been called OPT or MIN. It is simply this:

➢ Replace the page that will not be used for the longest period of time.

- Use of this page-replacement algorithm guarantees the lowest possible page- fault rate for a fixed number of frames.

Optimal

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

ratio	3	0	1	2	7	0	3	0	4	2	3	0	3	0	3	2	1	2	0	1	7	0	1
f1																							
f2																							
f3																							

Least Recently Used (LRU) Algorithm

- LRU replacement associates with each page the time of that page's last use.
- When a page must be replaced, LRU chooses the page that has not been used for the longest period.
- LRU strategy as the optimal page-replacement algorithm looking backward in time, rather than forward.

LRU–

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

ratio	3	0	1	2	7	0	3	0	4	2	3	0	3	0	3	2	1	2	0	1	7	0	1
f1																							
f2																							
f3																							

LRU Algorithm (Cont.)

- 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 find smallest value
    - Search through table needed
- 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
  - But each update more expensive
  - No search for replacement

LRU Approximation Algorithms

LRU needs special hardware and still slow.
There are three LRU Approximation Algorithms:

1. Additional reference bit Algorithm
2. Second Chance Algorithm
3. Enhanced Second Chance Algorithm

LRU Approximation Algorithms

- Reference bit Algorithm
  - With each page associate a bit, initially = 0
  - When page is referenced, bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - We do not know the order, however
- Second-chance algorithm
  - Generally, FIFO, plus hardware-provided reference bit
  - Clock replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules

Second Chance Algorithm

- Covered in Class

Allocation of Frames

- Each process needs ***minimum*** number of frames
- 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*
- ***Maximum*** of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

Fixed Allocation

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool
- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

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

$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 62 \approx 4$$
$$a_2 = \frac{127}{137} \times 62 \approx 57$$

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
  - But then process execution time can vary greatly
  - But greater throughput so more common

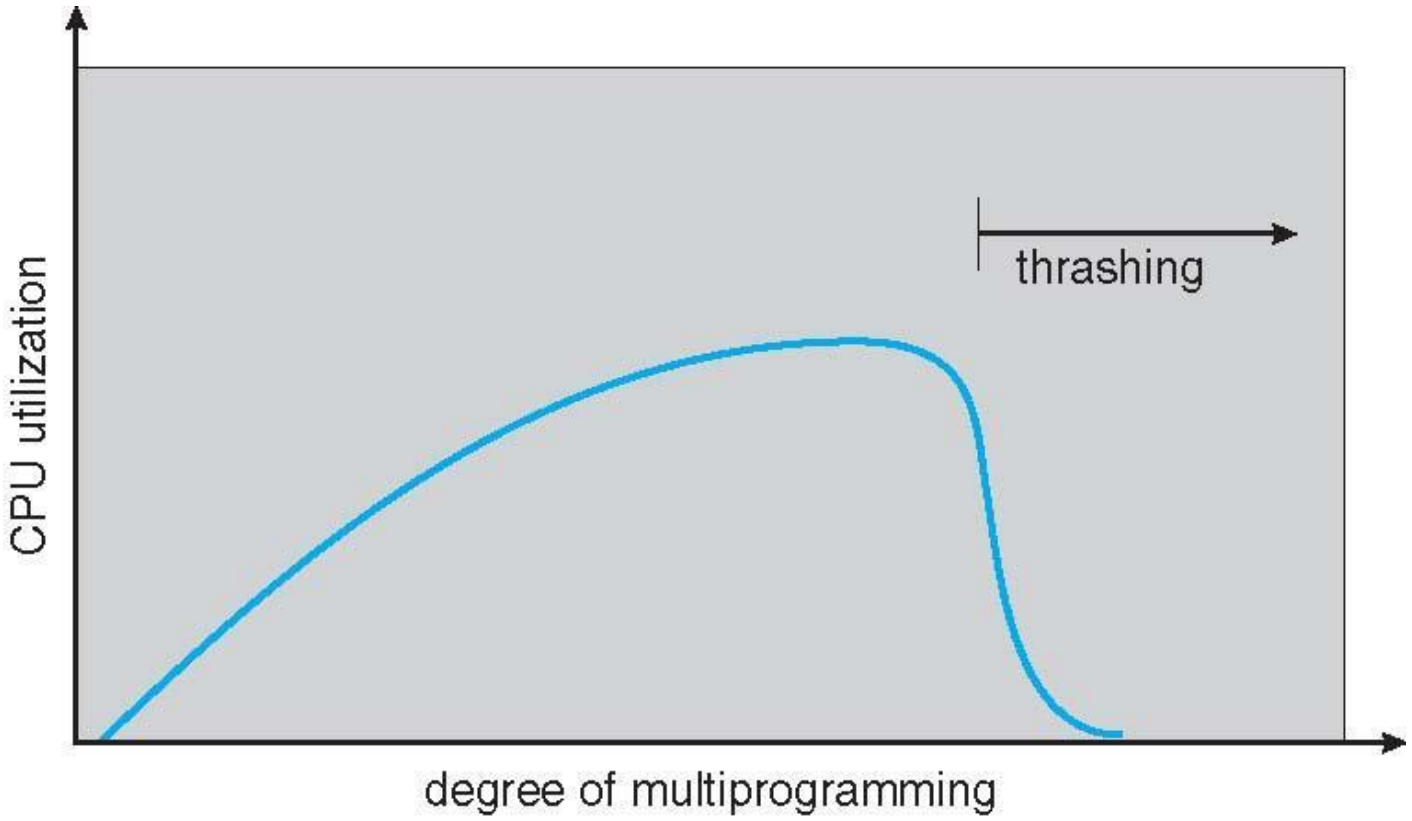
- Local replacement – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking 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.)



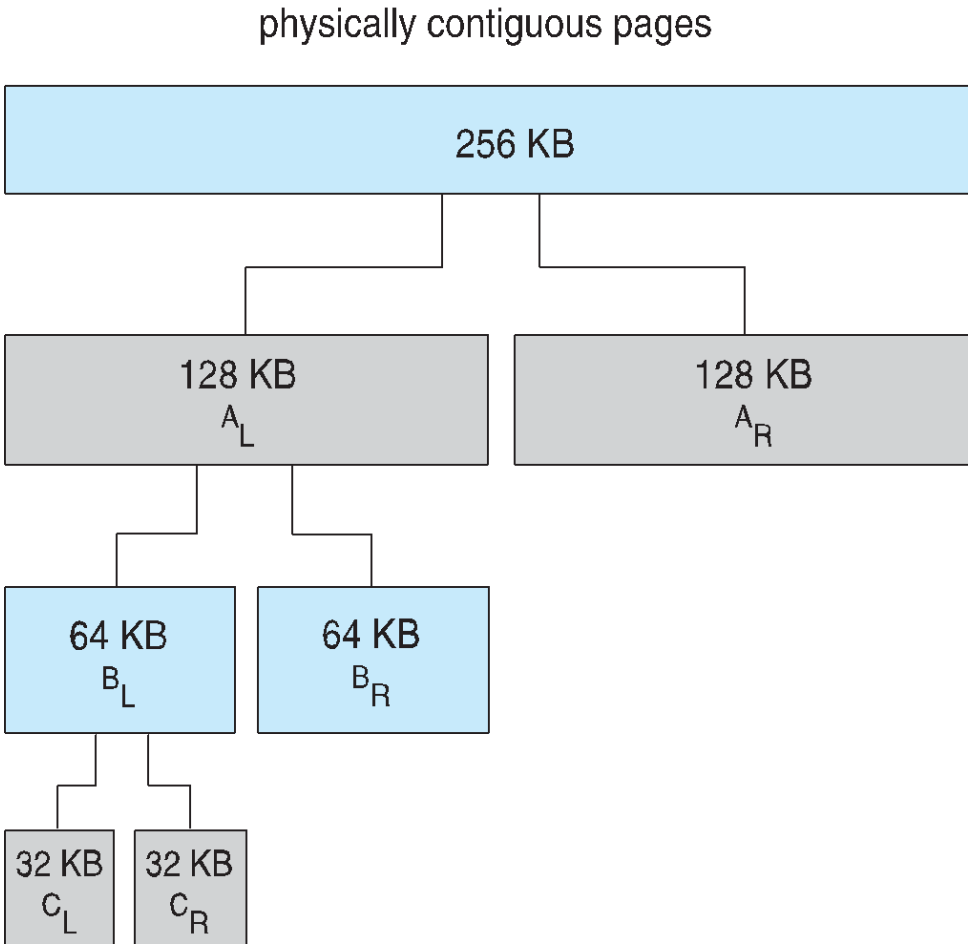
Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
    - I.e. for device I/O

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
- For example, assume 256KB chunk available, kernel requests 21KB
  - Split into A_L and A_R of 128KB each
    - One further divided into B_L and B_R of 64KB
      - One further into C_L and C_R of 32KB each – one used to satisfy request
- Advantage – quickly **coalesce** unused chunks into larger chunk
- Disadvantage - fragmentation

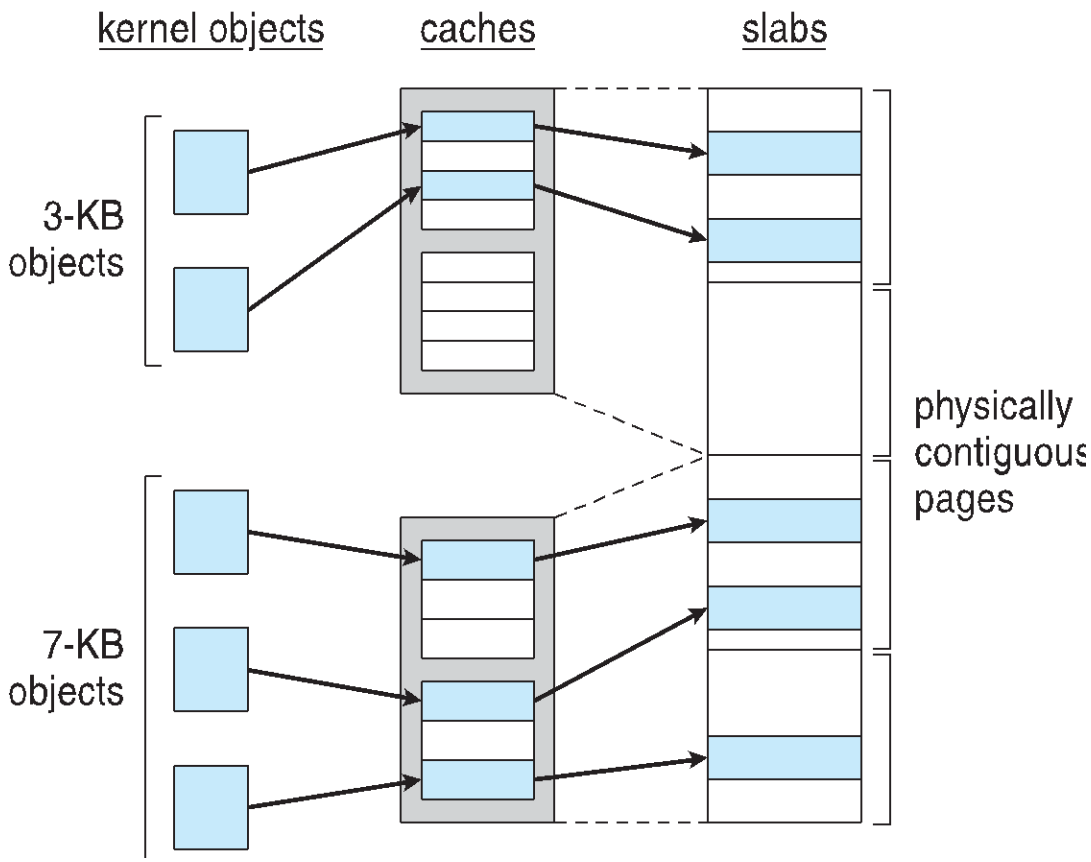
Buddy System Allocator



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

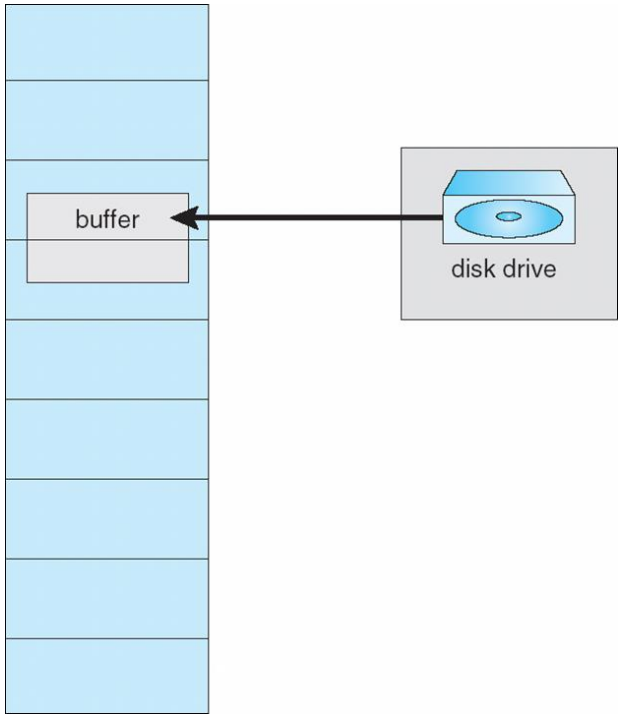


Other Considerations -- Prepaging

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

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
- Pinning of pages to lock into memory



Thank You!