CSI 4500 Operating Systems

Virtual Memory

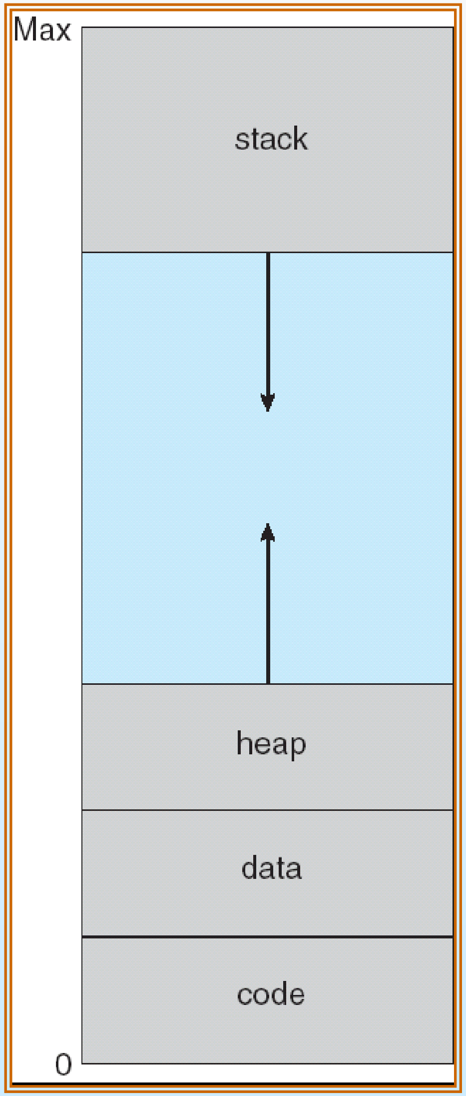
Objectives

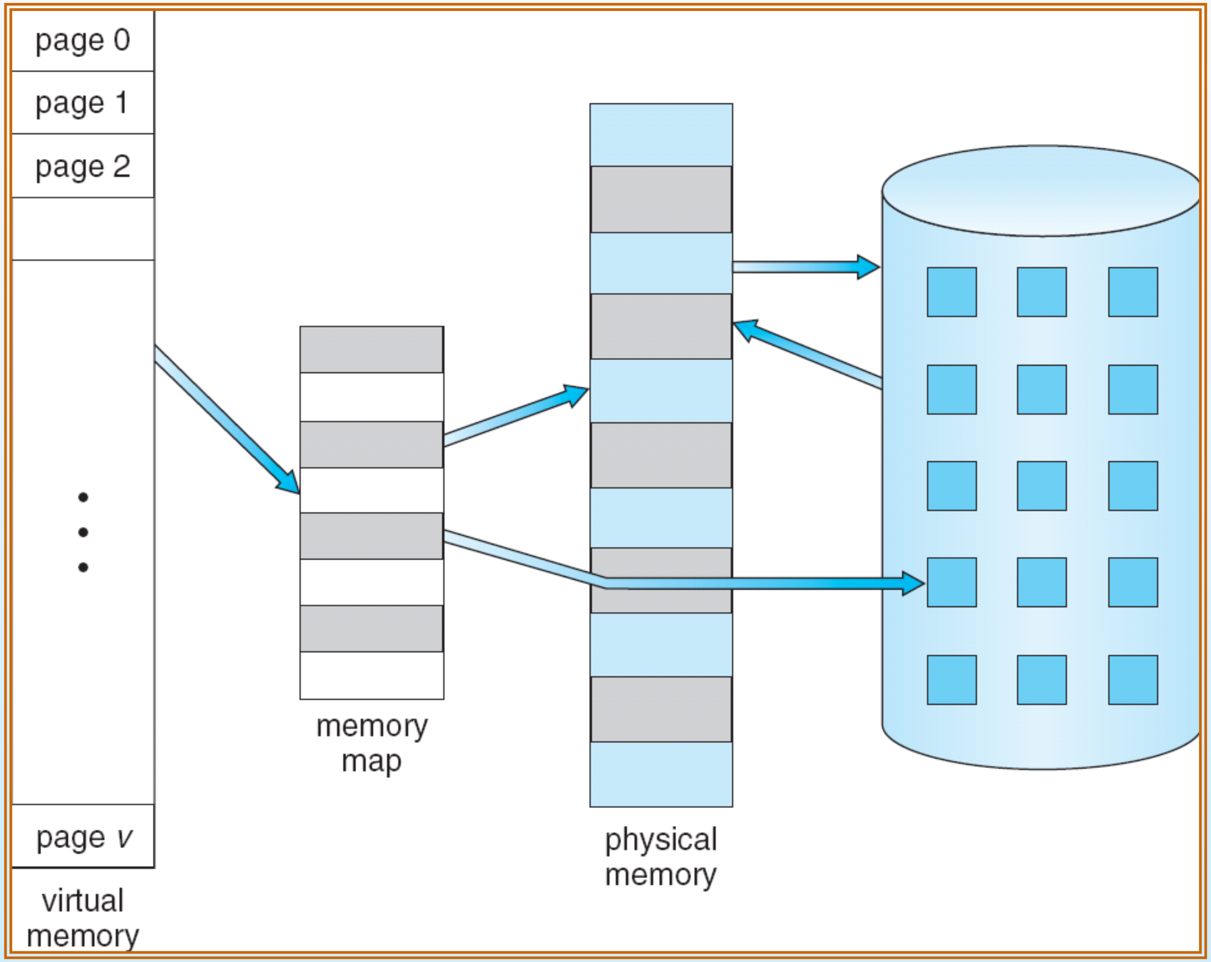
- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- To discuss the principle of the working-set model

Background

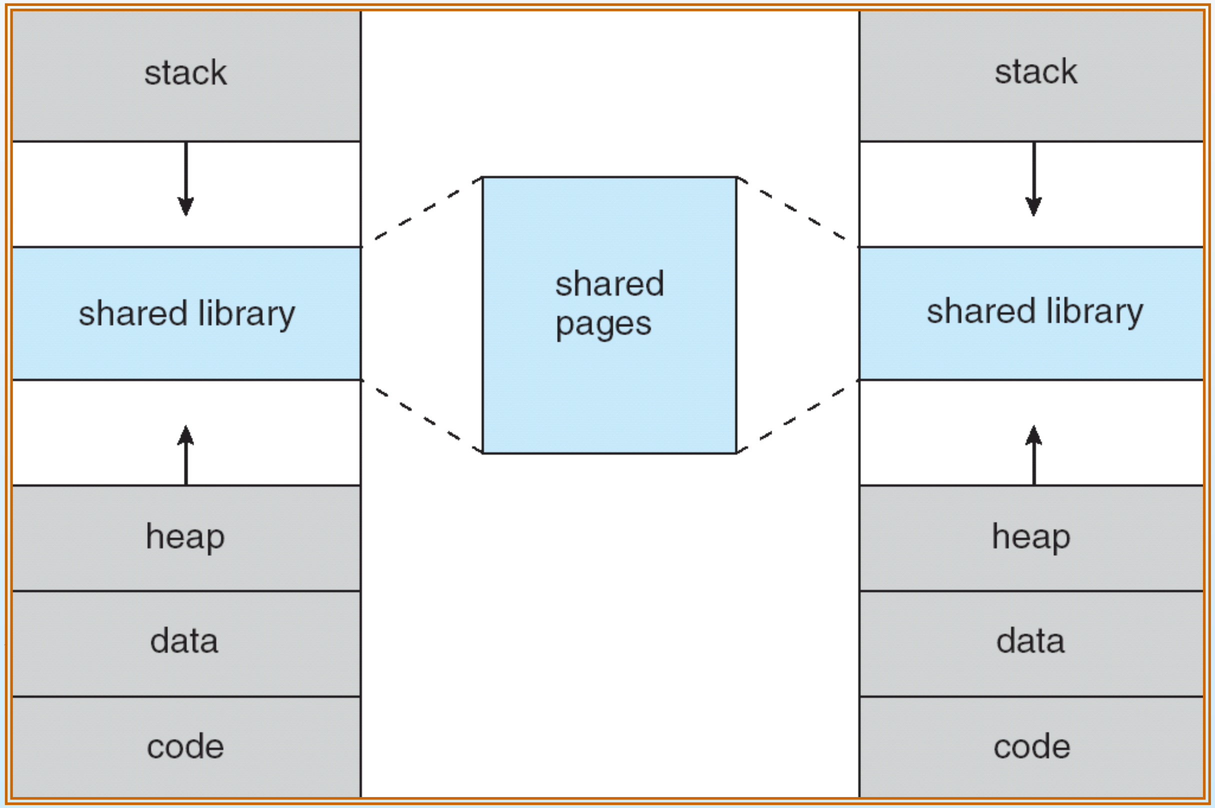
- **Virtual memory** – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for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

Virtual-address Space





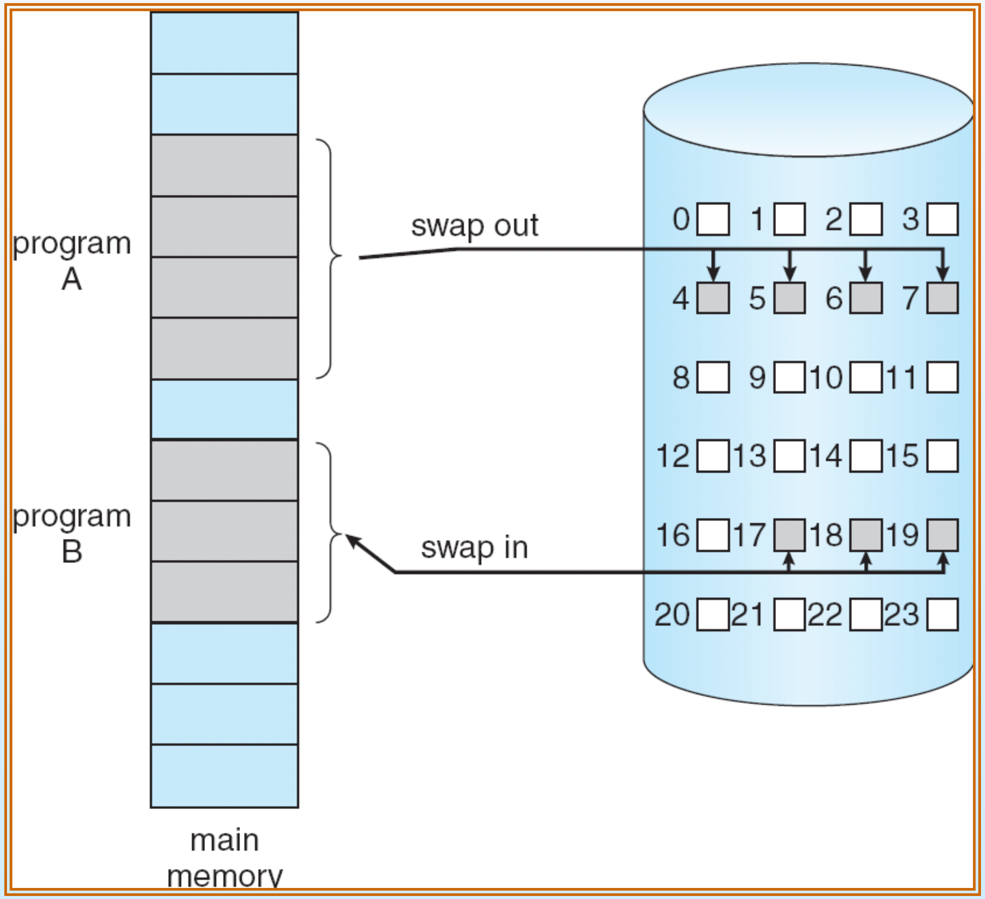
Shared Library Using Virtual Memory



Demand Paging

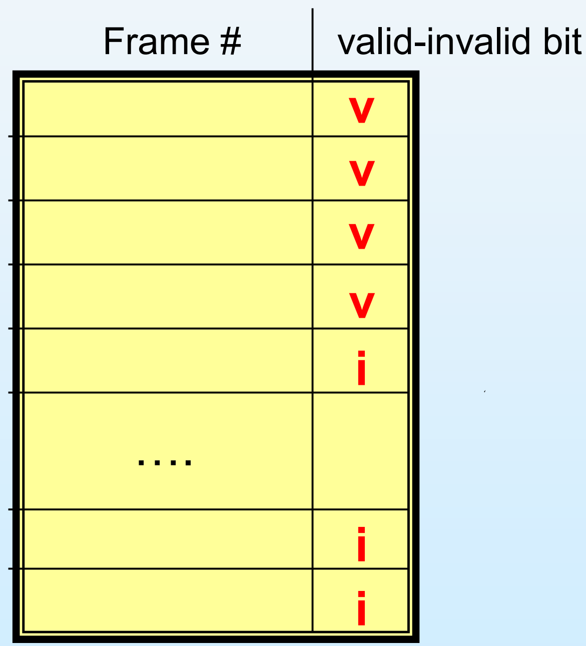
- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory

Paged Memory and Contiguous Disk Space



Valid-Invalid Bit

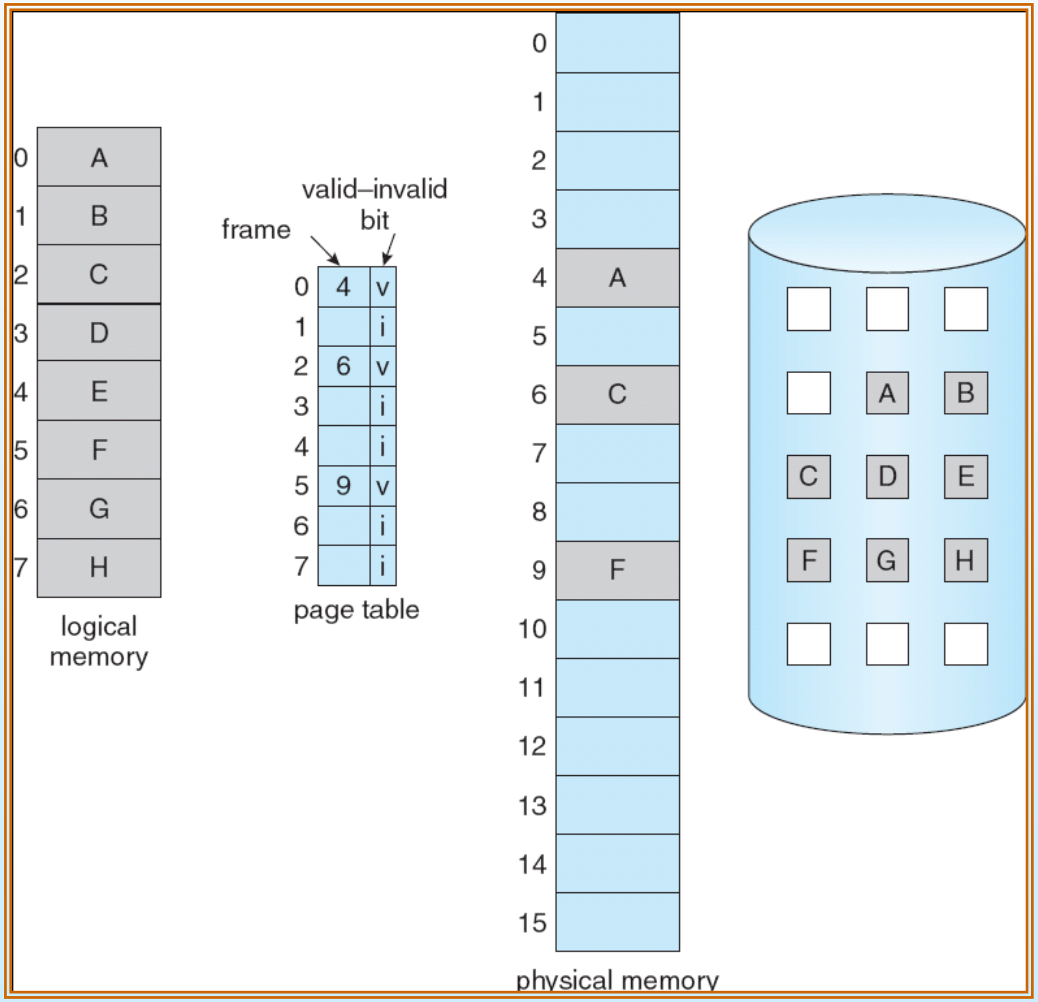
- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:



page table

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

Page Table Snapshot



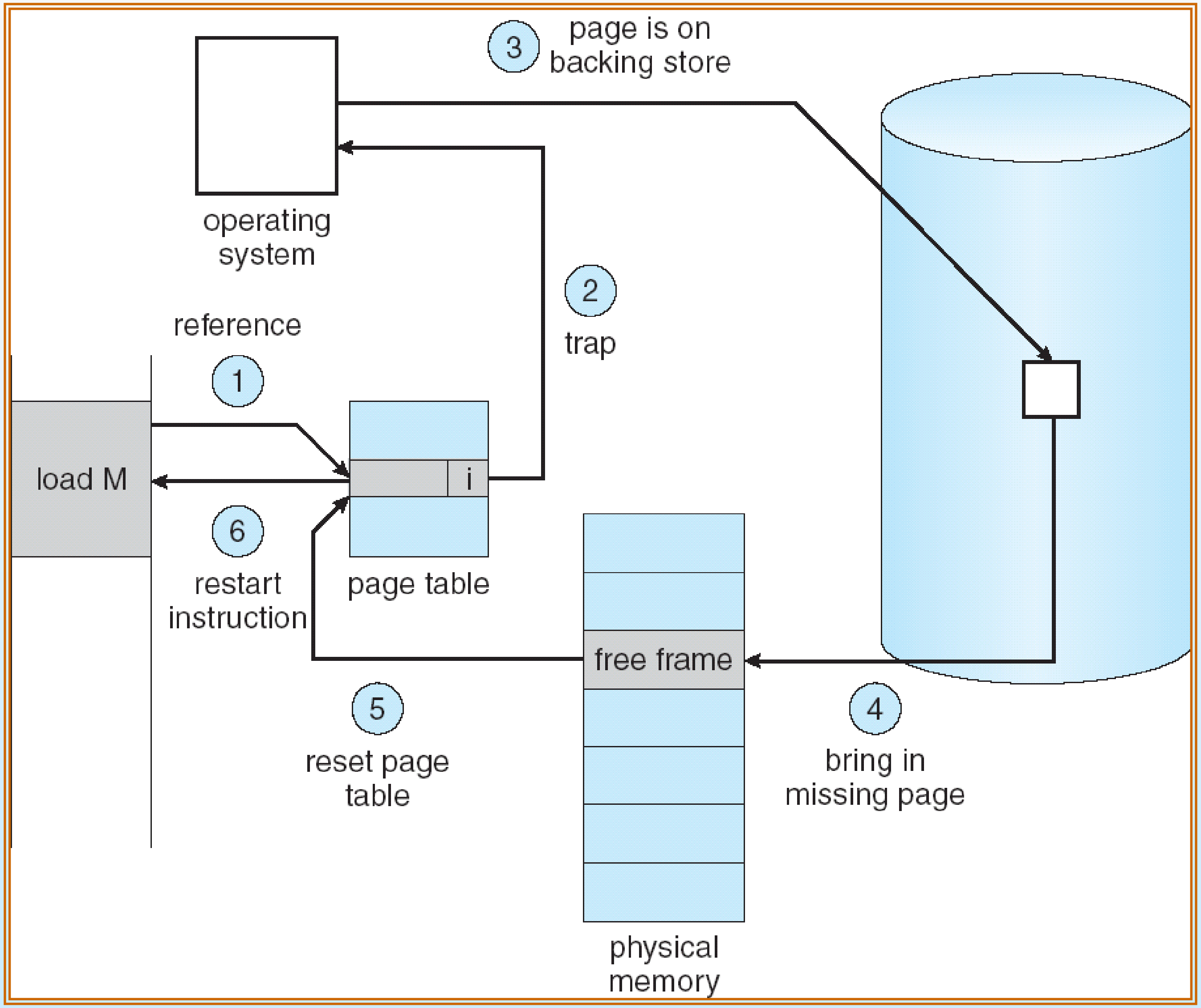
Page Fault

- 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 $\Rightarrow$ abort
   - Just not in memory
2. Get empty frame
3. Swap page into frame
4. Reset tables (*internal table* with PCB and *page table*)
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 $0 \leq p \leq 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

$$EAT = (1 - p) \text{ x memory access}$$
$$+ p \text{ x page fault time}$$

- page fault time
  - Service the page fault interrupt (***details***)
  - swap page out
  - swap page in
  - restart the process overhead

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p) x 200 + p x 8,000,000

  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then

  EAT = 8.2 microseconds. This is a slowdown by a factor of 40!!
- What if want slowdown by less than 10%?
  - 200ns x 1.1 < EAT $\Rightarrow$ p < 2.5 x 10^{-6}
  - This is about 1 page fault in 400000!

Process Creation

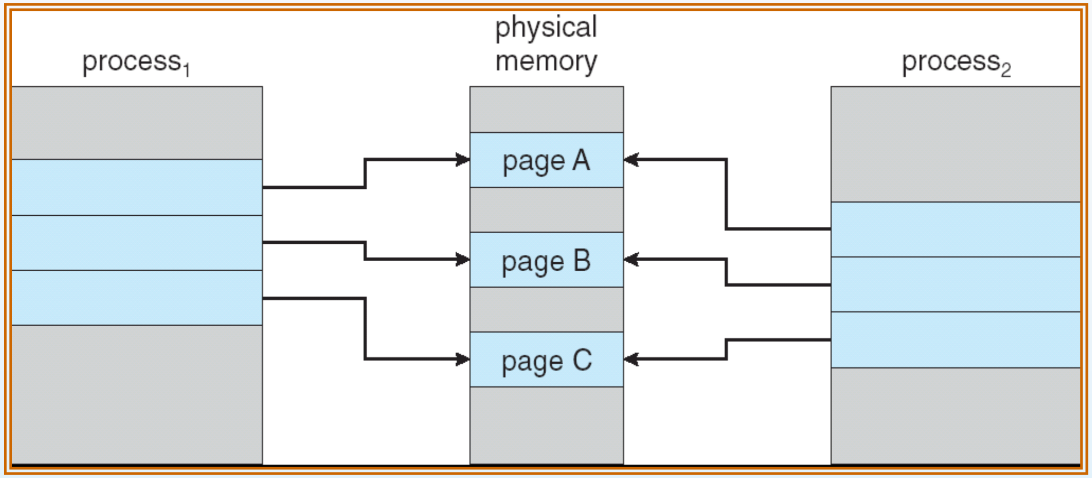
- Virtual memory allows other benefits during process creation:

  - Copy-on-Write

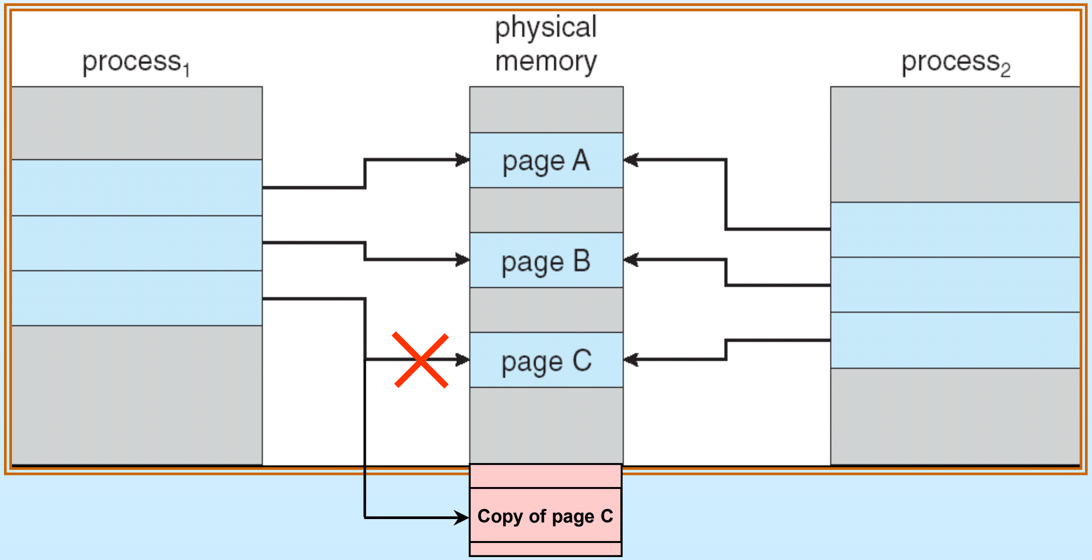
Copy-on-Write

- 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

COW illustration



Before Process 1 Modifies Page C

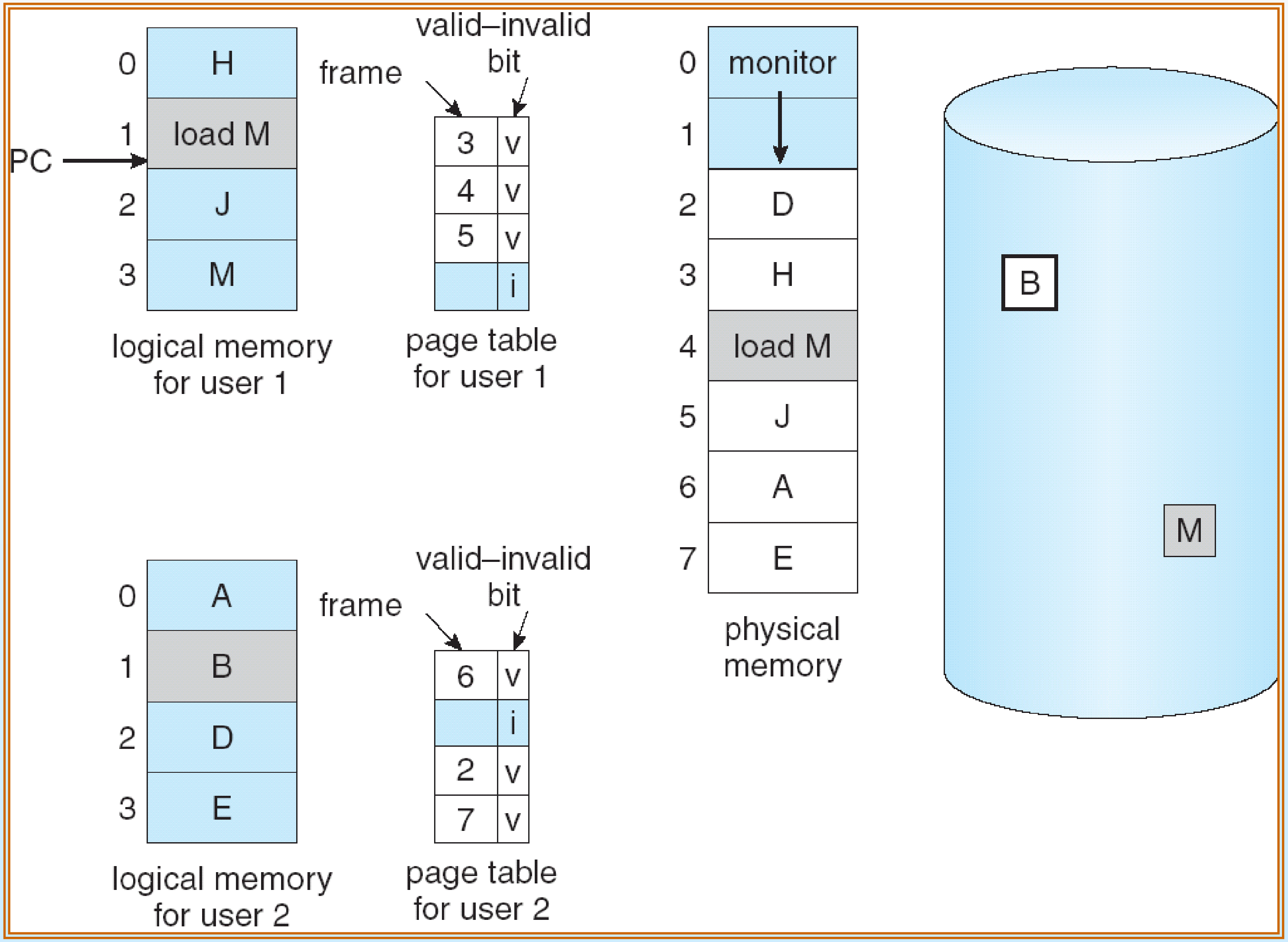


After Process 1 Modifies Page C

What happens if there is no free frame?

- **Page replacement** – find some page in memory, but not really in use, swap it out
  - Prevent over-allocation of memory by modifying page-fault service routine to include page replacement.
  - Same page may be brought into memory several times
  - **performance** – want an algorithm which will result in minimum number of page faults
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory.

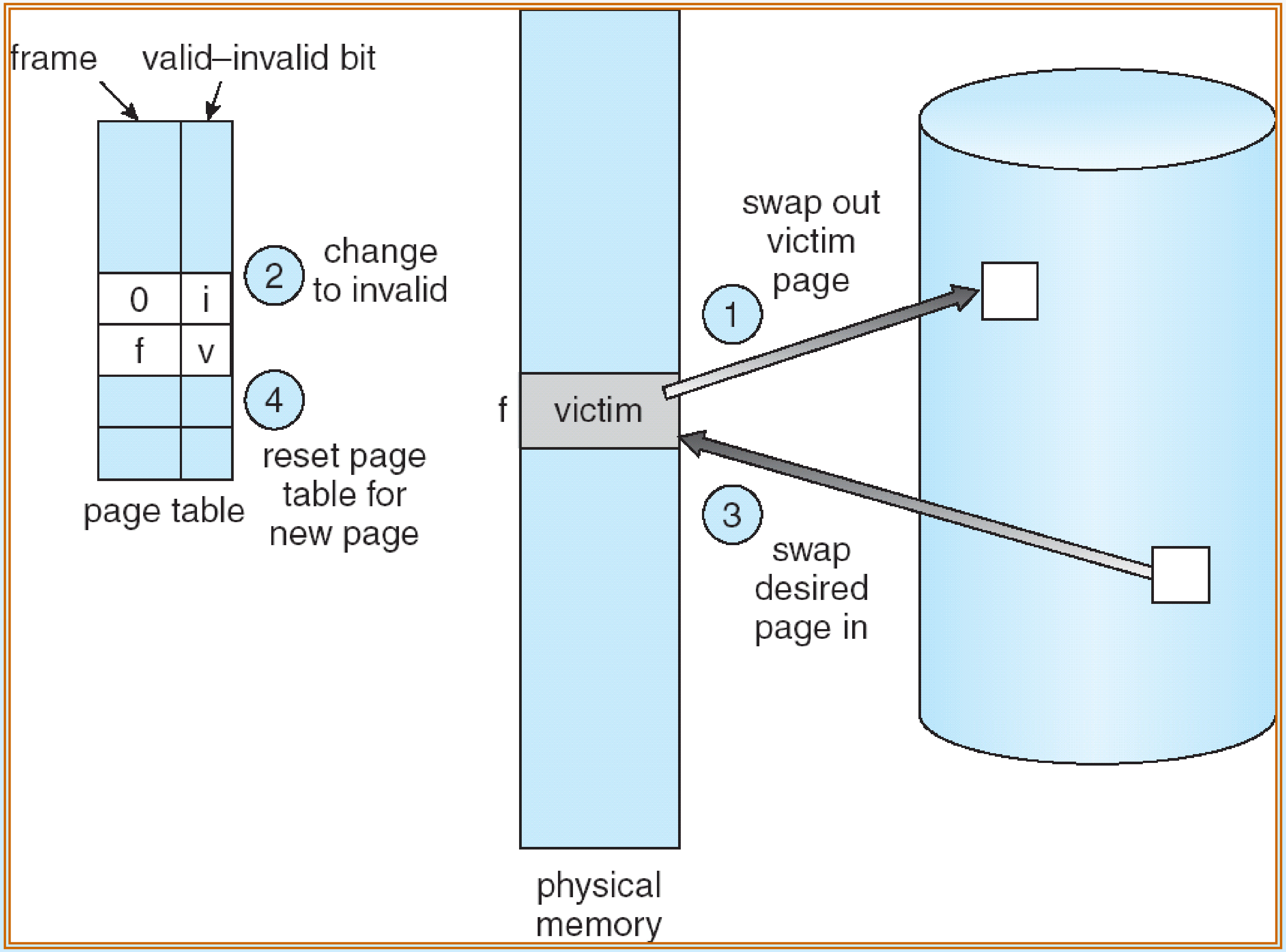
Need For Page Replacement



Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
   a) If there is a free frame, use it
   b) If there is no free frame, use a page replacement algorithm to select a **victim** frame
   c) Write the victim frame to the disk (*is it necessary?*)
   d) Update the page/frame tables accordingly
3. Bring the desired page into the (newly) free frame
   a) update the page and frame tables
4. Restart the user process

Page Replacement



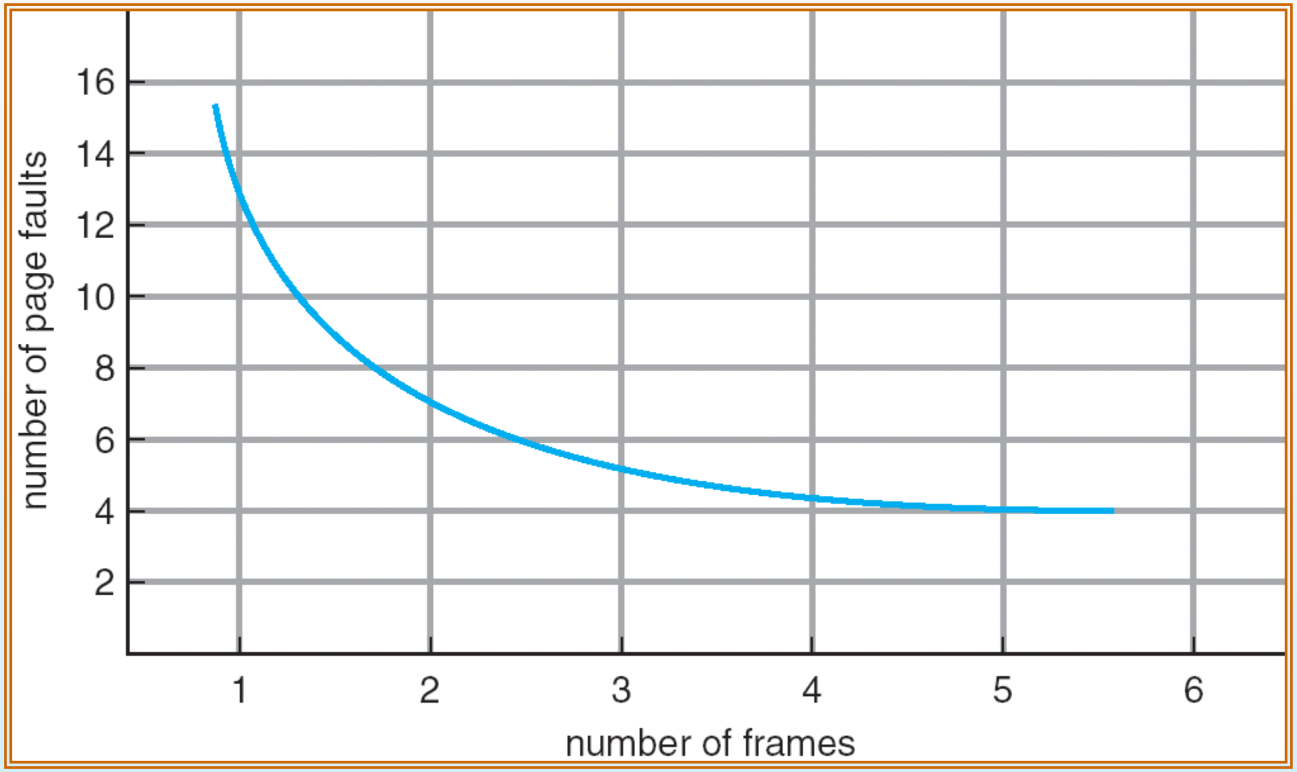
Page Replacement (Cont.)

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

Page Replacement Algorithms

- Replacement Performance - 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

Page Faults vs The Number of Frames

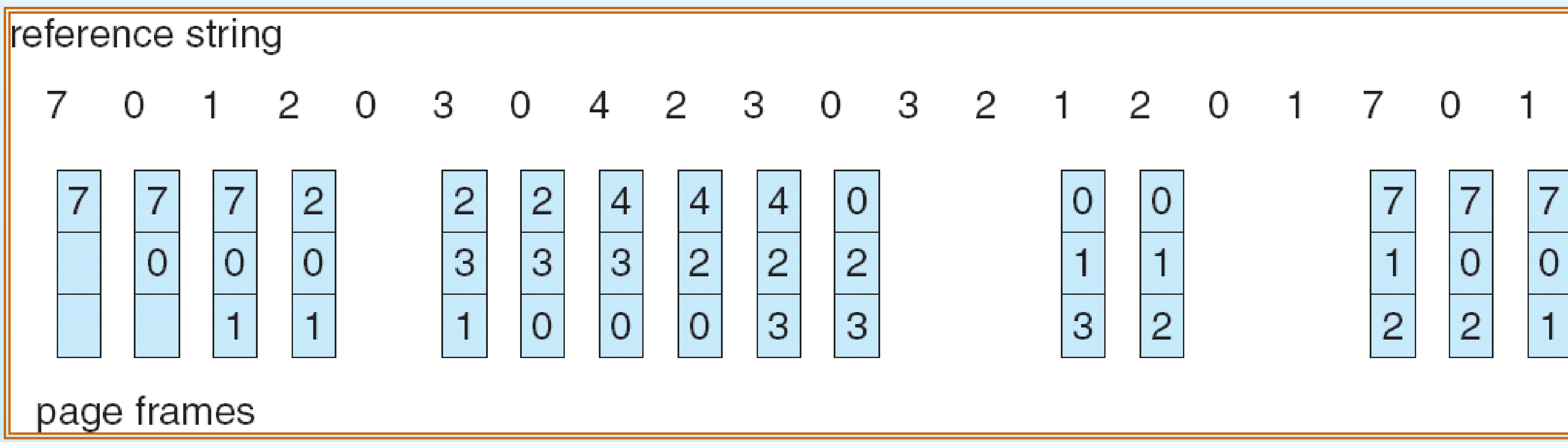


- # of frames increases, # of page faults decreases
  - Adding physical memory increases the # of frames

FIFO Page Replacement

- **FIFO replacement algorithm** – when a page must be replaced, the oldest page is chosen.
- Reference String:

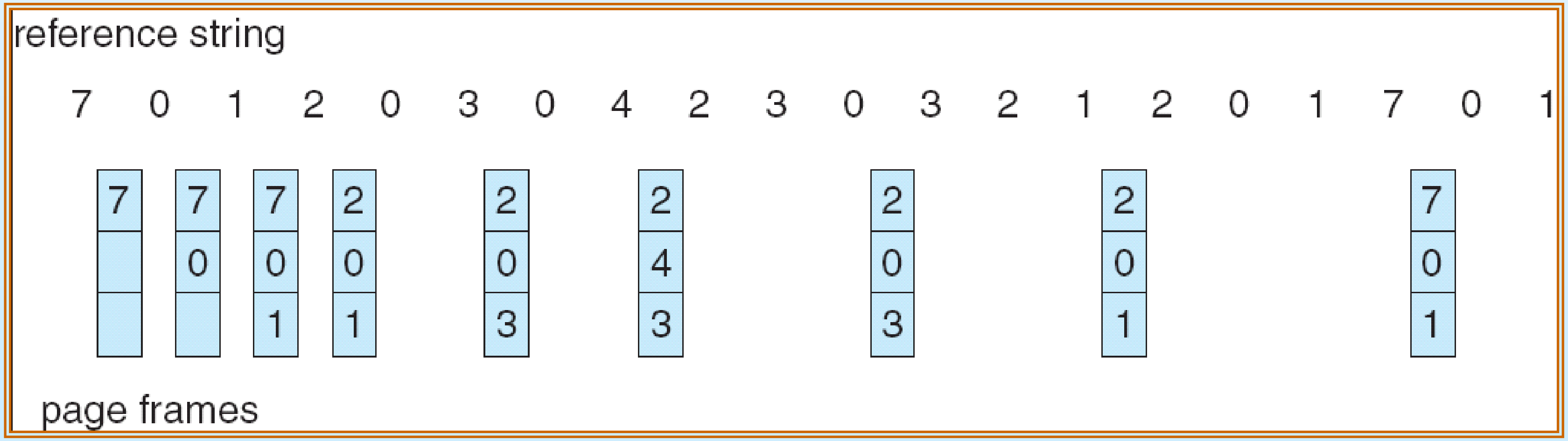
  7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
- # of Frames = 3, # of page faults = 15.



Optimal Page Replacement

- Replace page that will not be used for longest period of time
- Reference String:

  7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
- # of Frames = 3, # of page faults = 9.
- How do you know this?
- Used for measuring how well the other algorithms perform

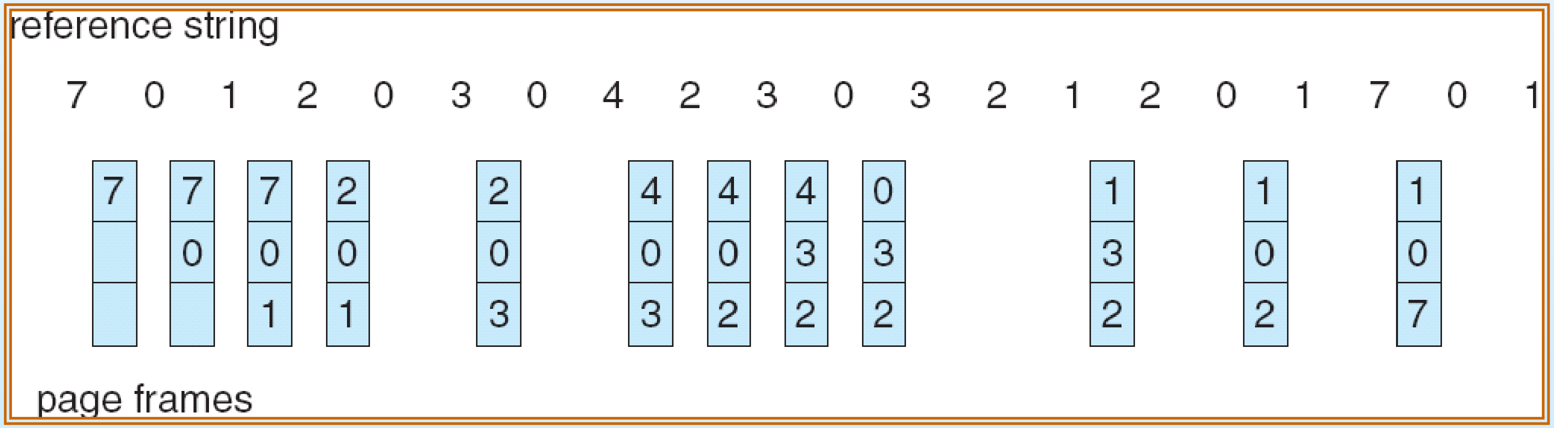


LRU Page Replacement

- Reference String:

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

- # of Frames = 3, # of page faults = 13.

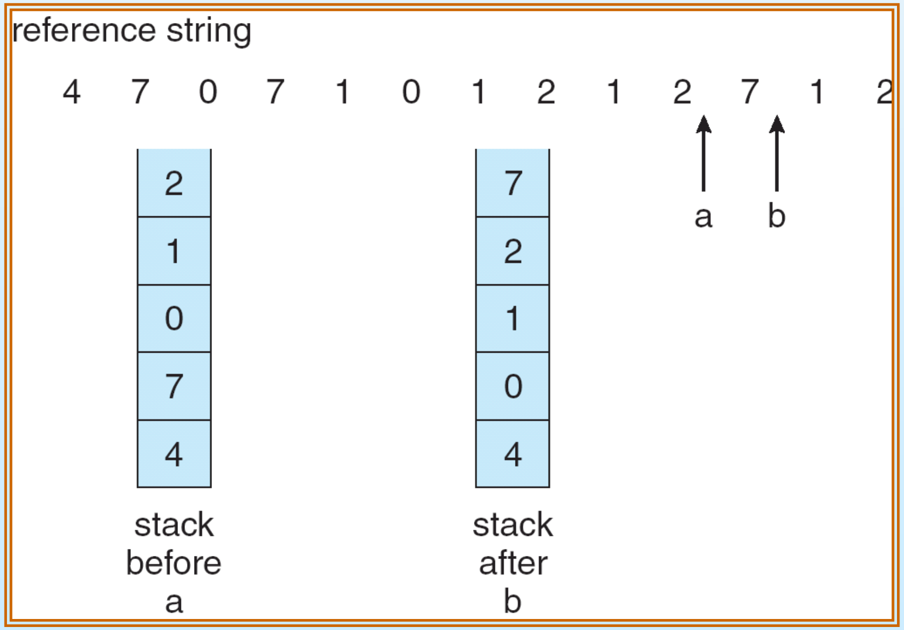


LRU Algorithm Implementation

- 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 replaced, look at the counters to determine which are to replace
- **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

LRU Algorithm Implementation (Cont.)

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



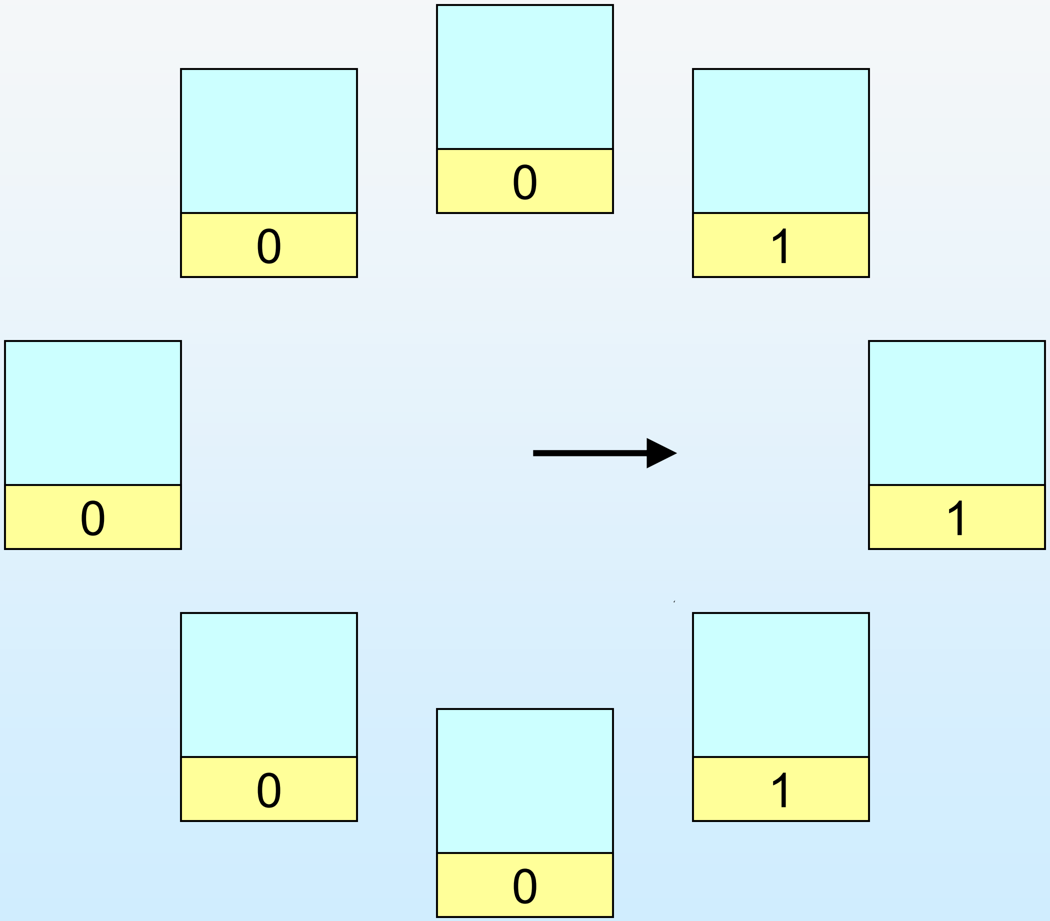
LRU Approximation Algorithms

- 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

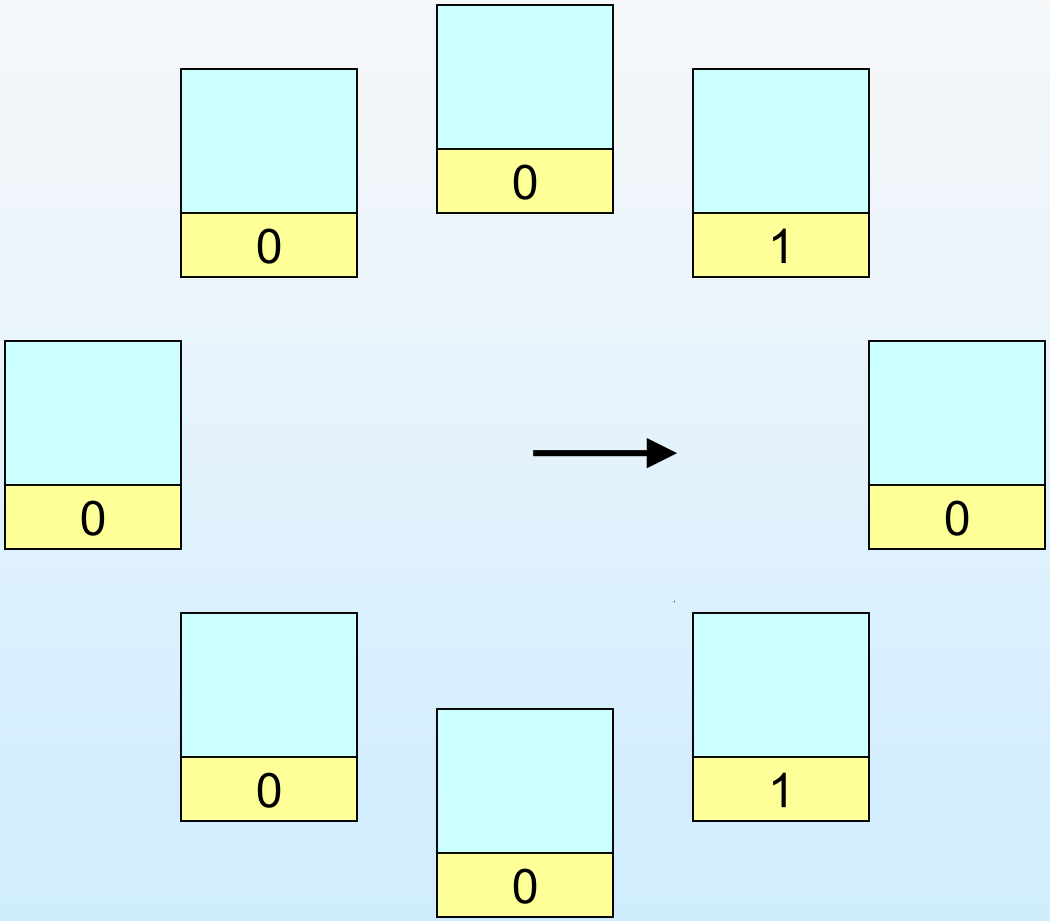
Clock Algorithm

- Replaces an old page, but not the oldest page
- Arranges physical pages in a circle
  - With a clock hand
- Each page has a ***used bit***
  - Set to 1 on reference
  - On page fault, sweep the clock hand
    - If the used bit == 1, set it to 0
    - If the used bit == 0, pick the page for replacement

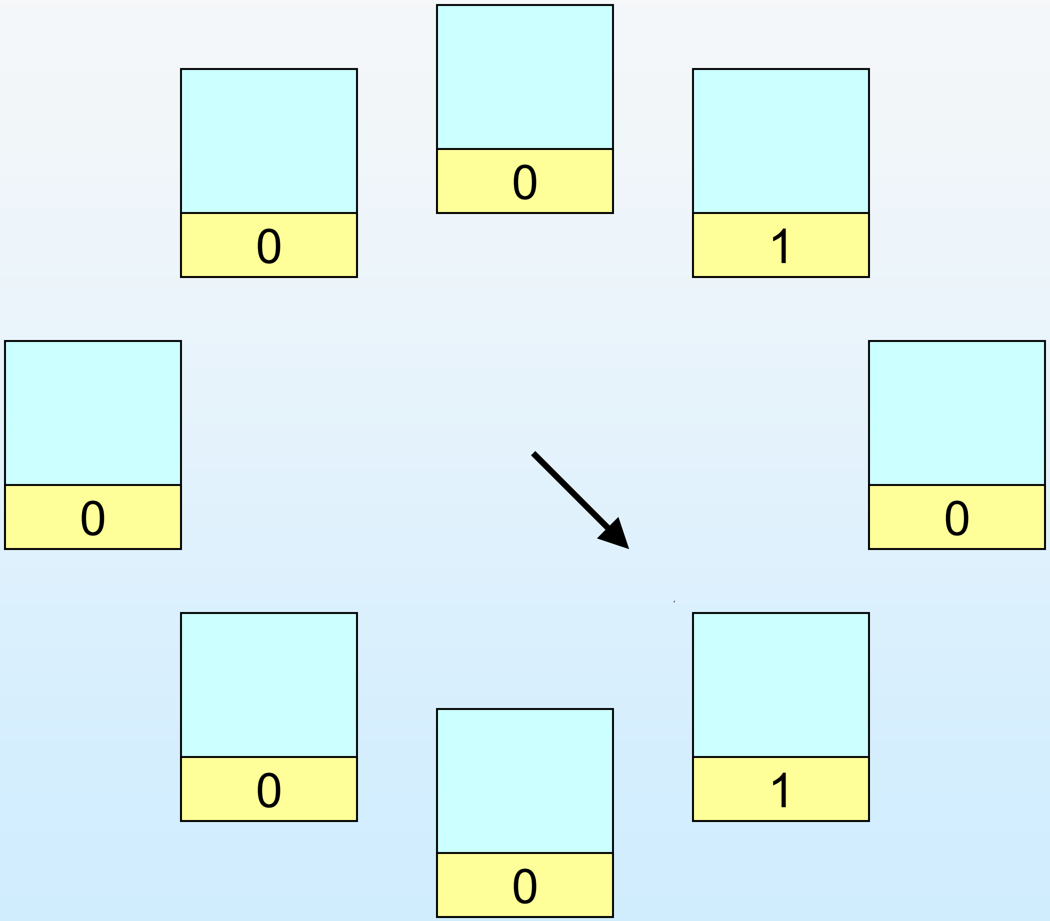
Clock Algorithm



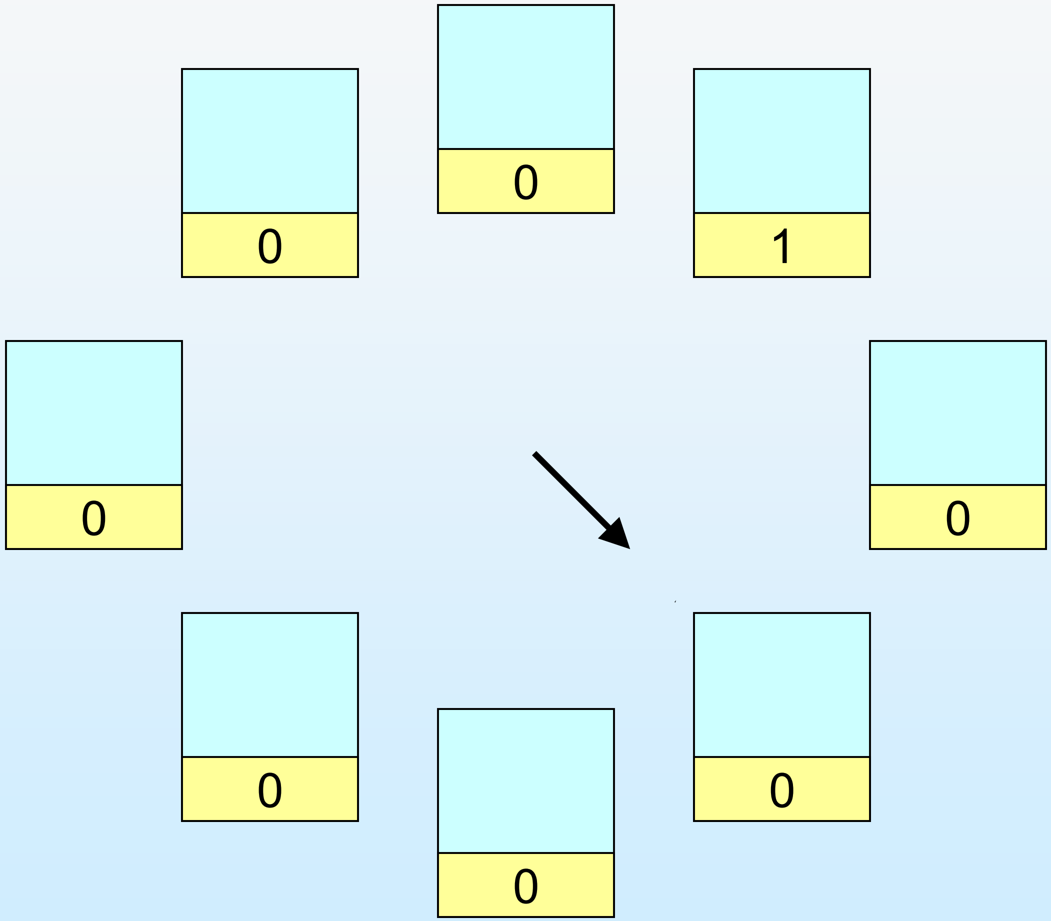
Clock Algorithm



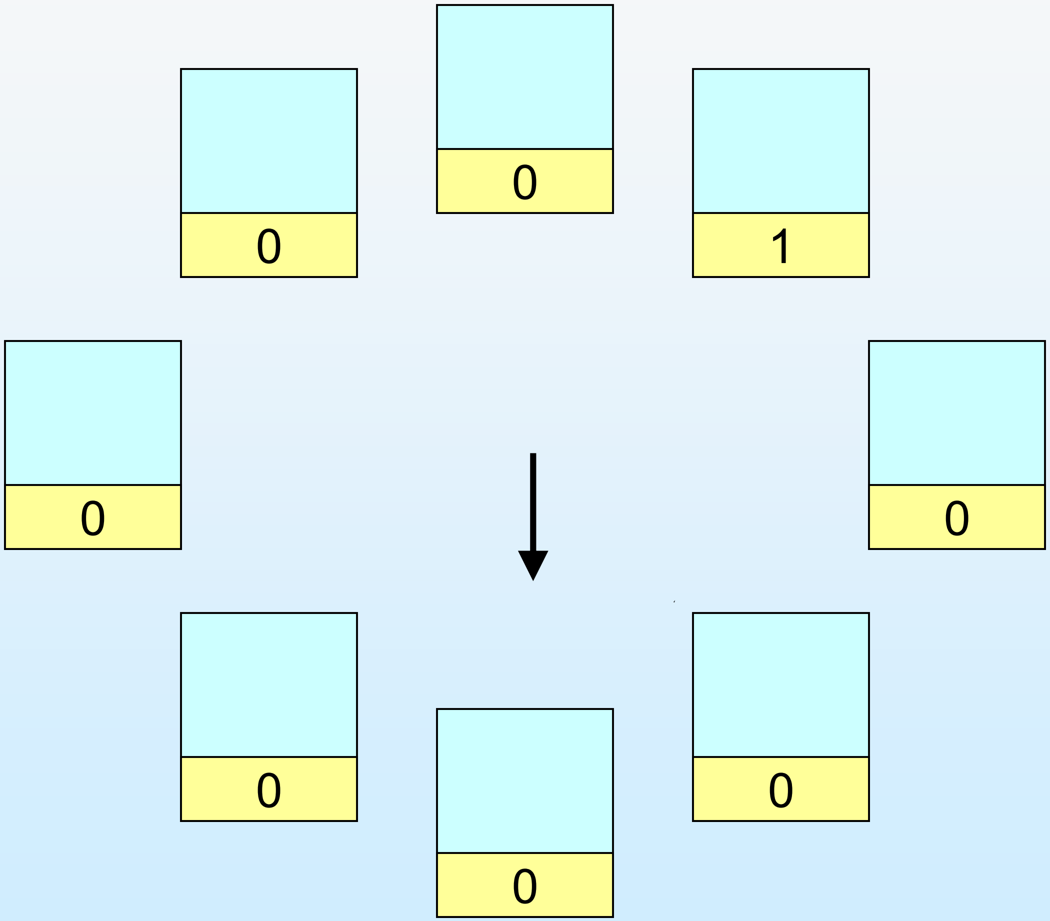
Clock Algorithm



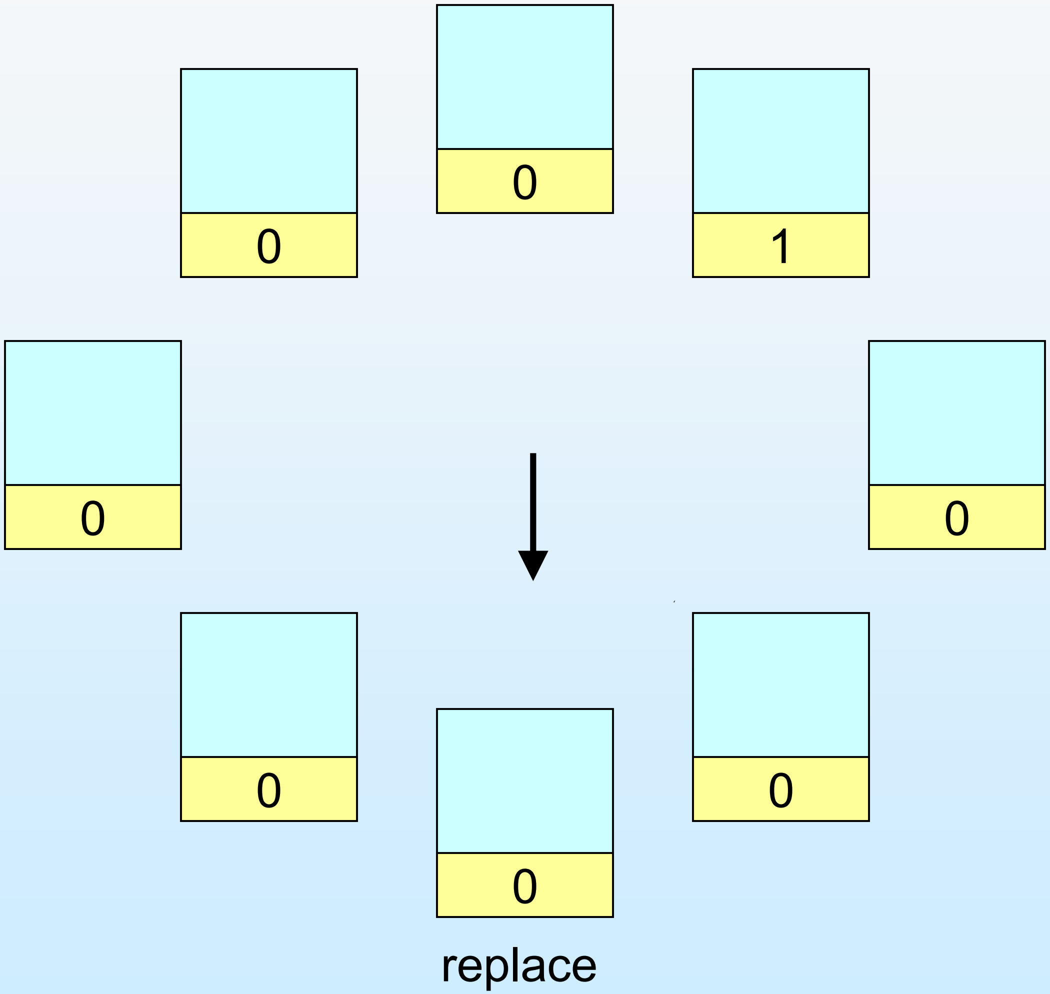
Clock Algorithm



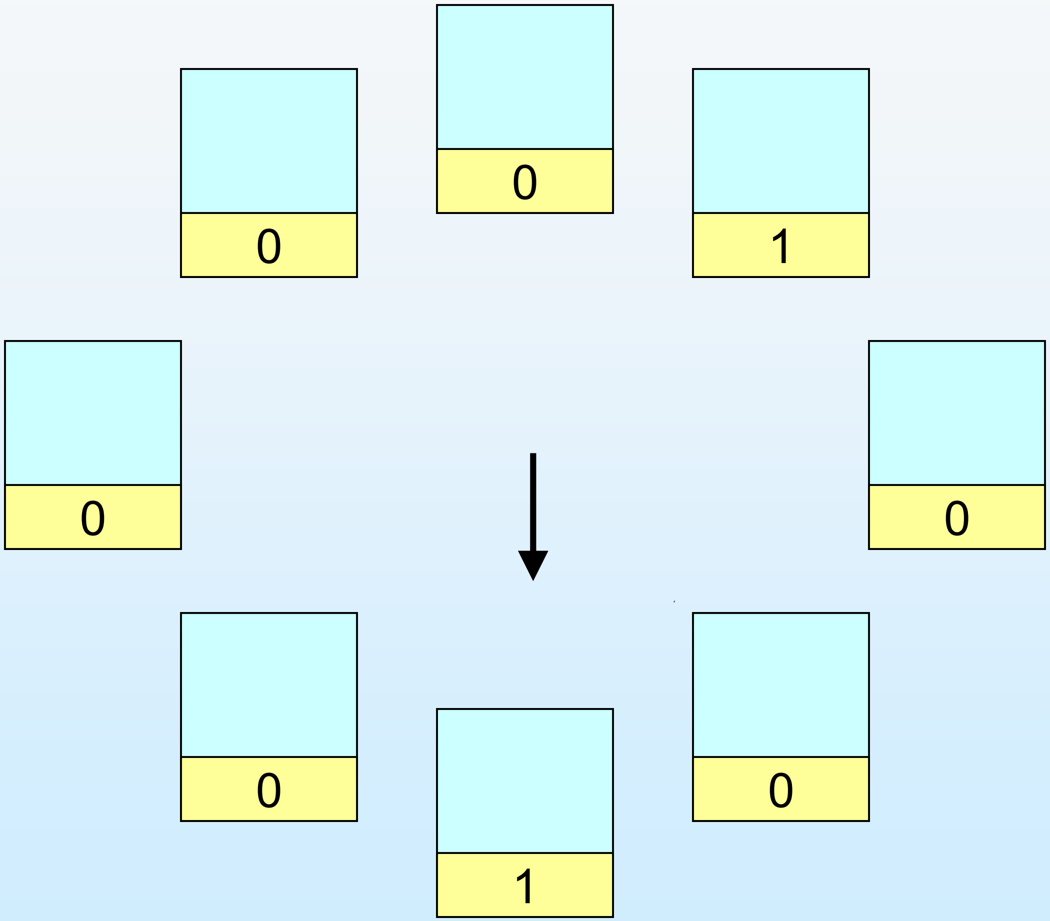
Clock Algorithm



Clock Algorithm



Clock Algorithm



N^{th} Chance Algorithm

- A variant of clocking algorithm
  - A page has to be swept N times before being replaced
  - N → ∞, Nth Chance Algorithm → LRU
  - Common implementation
    - N = 2 for modified pages
    - N = 1 for unmodified pages

States for a Page Table Entry

- **Used bit:** set when a page is referenced; cleared by the clock algorithm
- **Modified bit:** set when a page is modified; cleared when a page is written to disk
- **Valid bit:** set when a program can legitimately use this entry
- **Read-only:** set for a program to read the page, but not to modify it (e.g., code pages)

Multi-Programming Frame Allocation

- How are the page frames allocated to individual virtual memories of the various jobs running in a multi-programmed environment?
- **Simple solution**
  - Allocate a minimum number (??) of frames per process.
    - One page from the current executed instruction
    - Most instructions require two operands
    - include an extra page for paging out and one for paging in
- **Equal allocation:** For example, if there are 100 frames and 5 processes, give each process 20 frames.

Multi-Programming Frame Allocation

- **Proportional allocation:** Allocate according to the size of process
  - how do you determine job size: by run command parameters or dynamically?

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

$$m = 64$$
$$s_i = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 64 \approx 5$$
$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Global Replacement vs. Local Replacement

- If process P_i generates a page fault,
  - **local replacement**: select for replacement one of its frames
  - **global replacement**: select for replacement a frame from a process with lower priority number
    - All pages are in the single pool (e.g., UNIX)
      - Grabs memory from another process that needs less

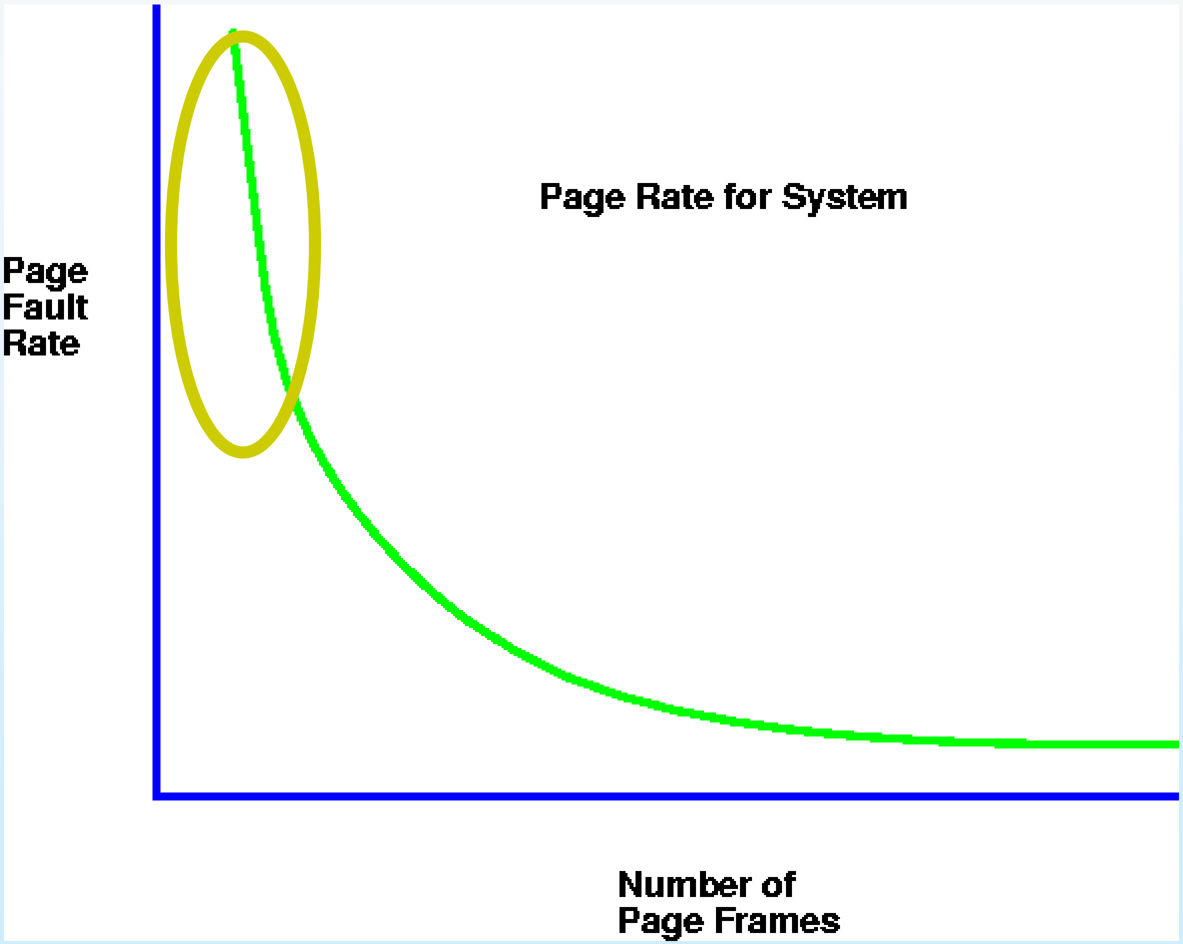
    + Flexible

    - One process can drag down the entire system
- Why is multi-programming frame allocation is important?
  - If not solved appropriately, it will result in a severe problem
    - Thrashing!!!

Thrashing

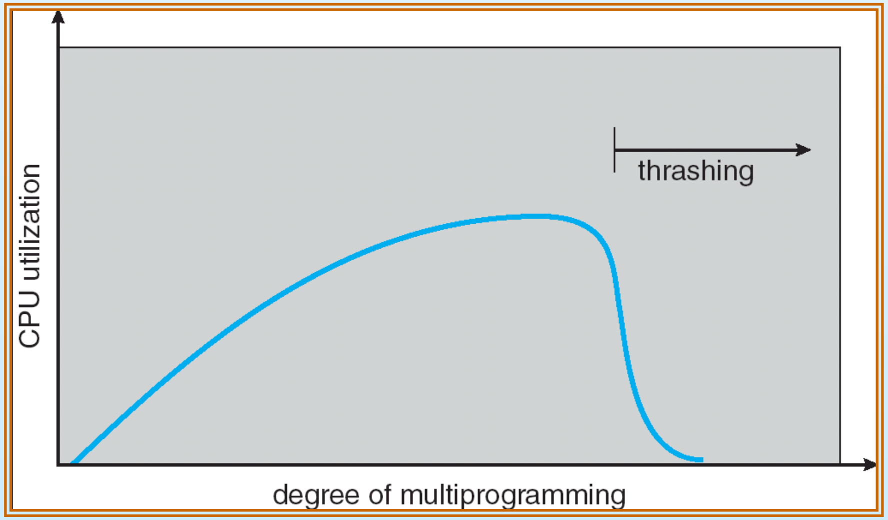
- Occurs when the memory is overcommitted
  - Pages are still needed are tossed out
- Example:
  - A process needs 50 memory pages
  - A machine has only 40 memory pages
  - Need to constantly move pages between memory and disk
- **Thrashing** ≡ a process is busy swapping pages in and out rather than execution.

Page Fault Rate vs. Frame Size Curve



Results of Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. What will happen?
  - low CPU and Memory utilization
    - How about I/O utilization?
  - operating system thinks that it needs to increase the degree of multiprogramming, **why?**
  - Starting new jobs will reduce the number of page frames available to each process, increasing the page fault requests.



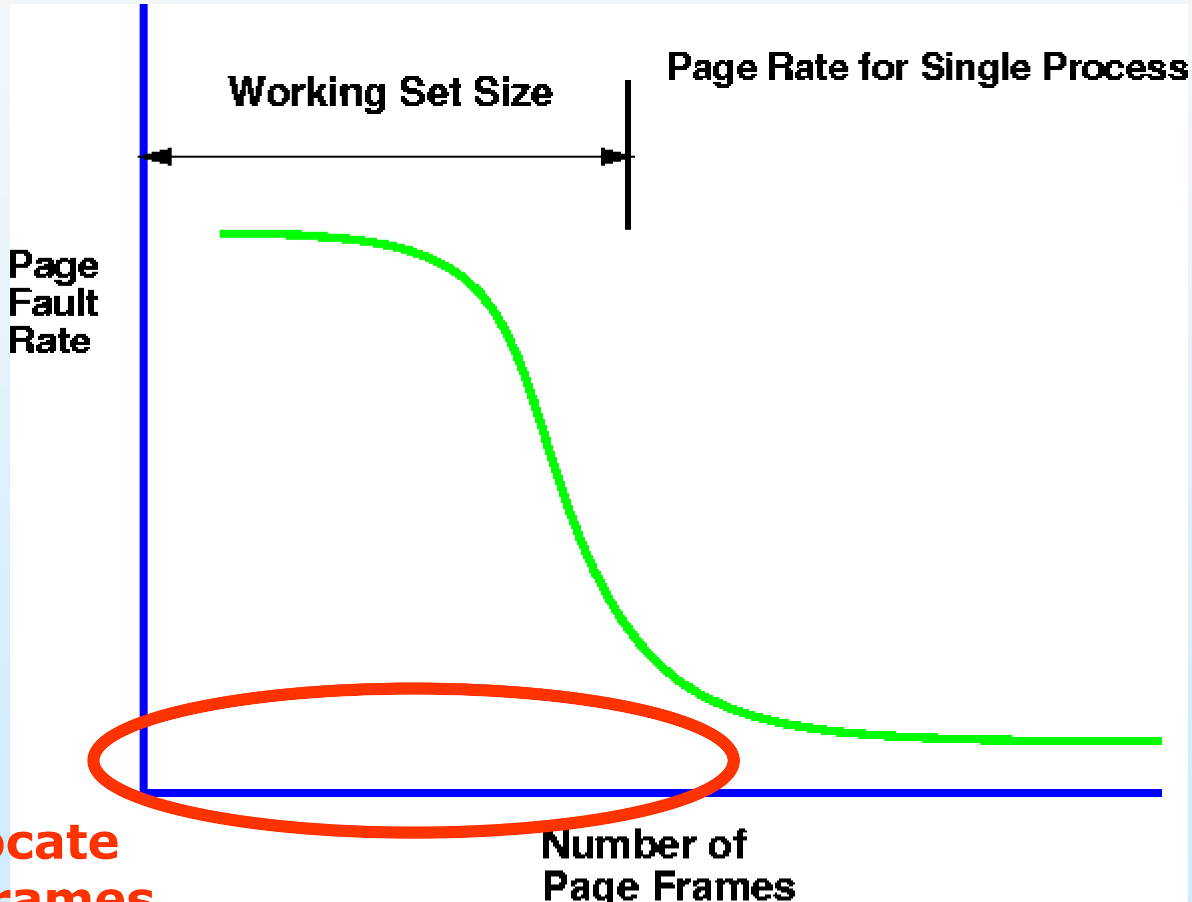
Why Thrashing?

- **Computations have locality**
- As sizes of frames decrease, the sizes of frames available are not large enough to contain the locality of the process.
- The processes start faulting heavily
  - Pages that are read in, are used and immediately paged out.
- How can we limit the effects of thrashing?

Solution: Working Set (1968, Denning)

- Main idea
  - **Figure out how much memory does a process need to keep most the recent computation in memory with very few page faults?**
- How?
  - The working set model assumes locality
    - **Locality** is a set of pages that are actively used together, may overlap
    - Process migrates from one locality to another
  - the principle of locality states that a program clusters its access to data and text temporally
    - A recently accessed page is more likely to be accessed again
  - Thus, as the number of page frames increases above some threshold, the page fault rate will drop dramatically

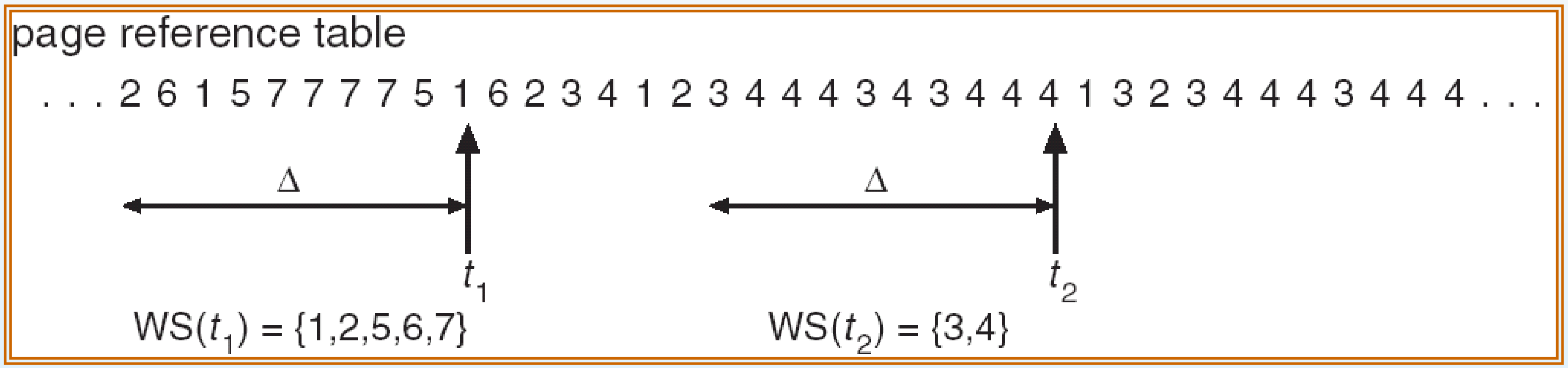
Working Set



At least allocate this many frames for this process

Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example:



- 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
- If sum of WSS_i > available frames => thrashing will occur

Working Set in Action to Prevent Thrashing

- Pseudo Algorithm

DO each process in the system

IF #free page frames > working set of some suspended *process*$_i$,

THEN activate *process*$_i$ and map in all its working set

END IF

IF working set size of some *process*$_k$ increases and no page frame is free,

THEN suspend *process*$_k$ and release all its pages

END IF

FOREVER

Implementation

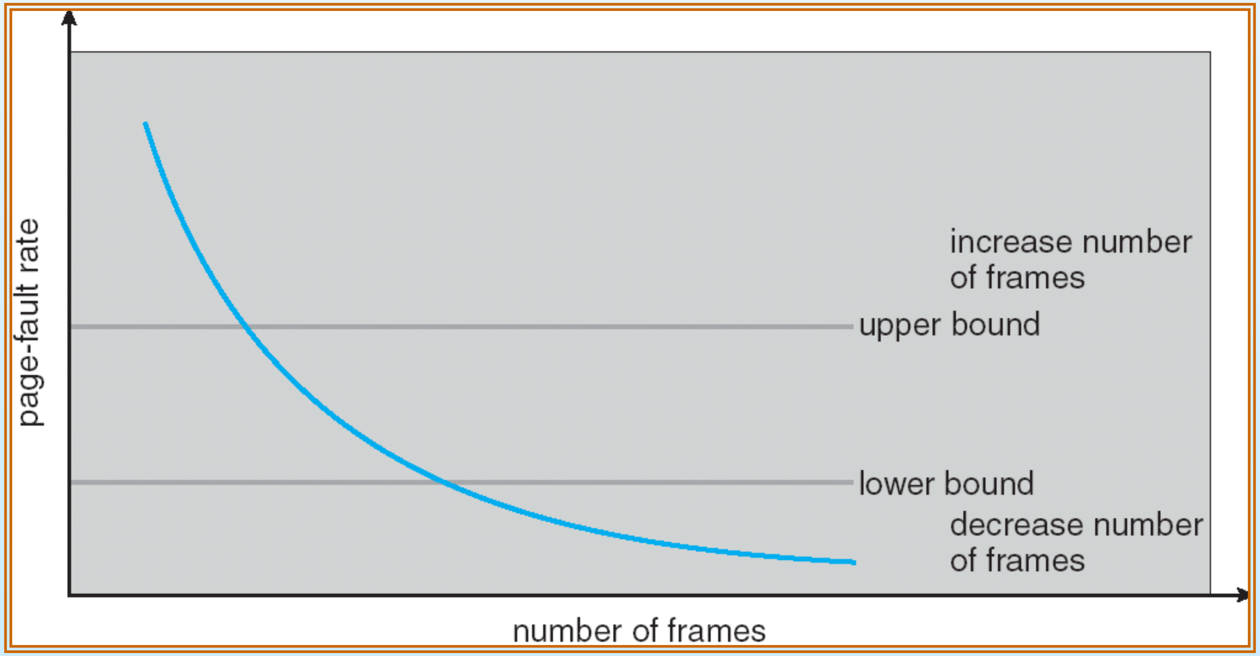
- Approximate working set model using timer and reference bit

- Set timer to interrupt after approximately x references.
  - Every 10,000 references

- Remove pages that have not been referenced and reset reference bit.

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



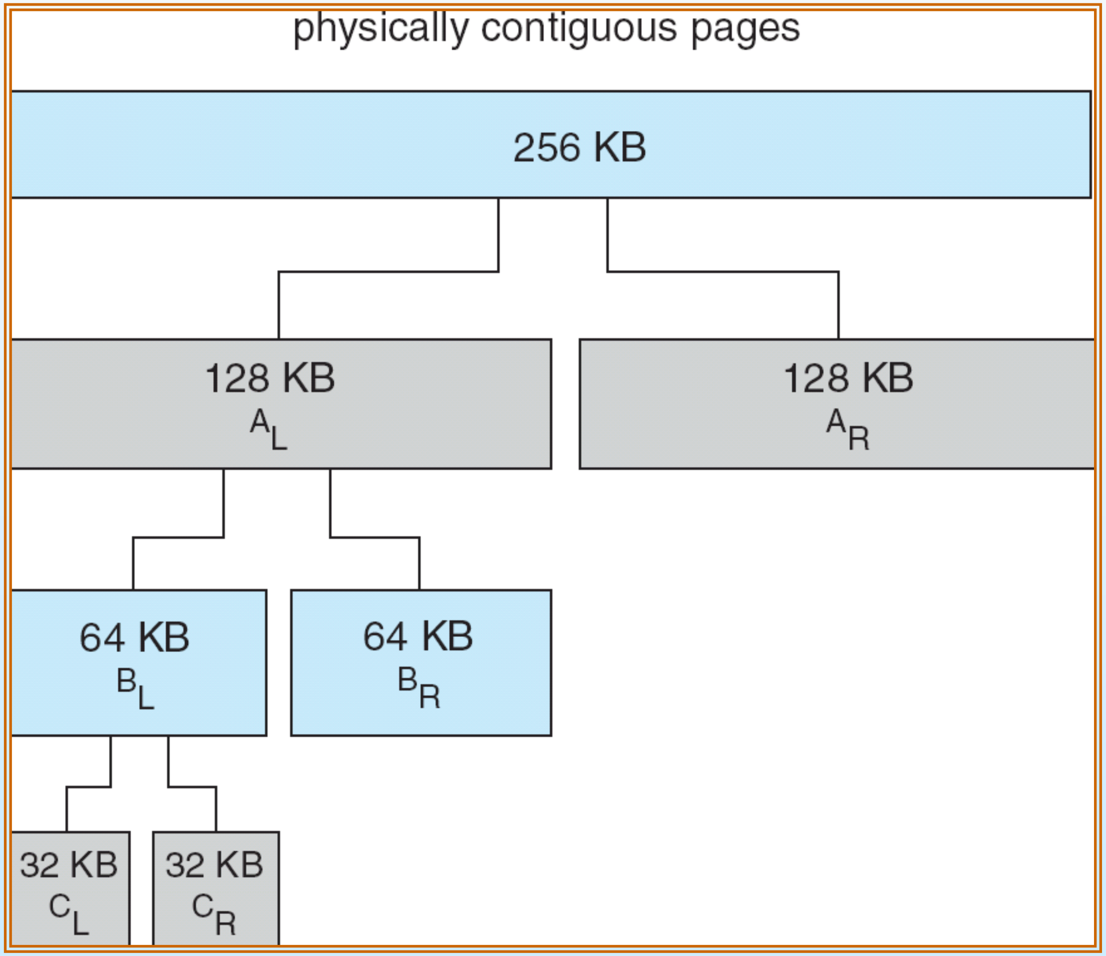
Allocating Kernel Memory

- 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
- Two strategies for managing kernel memory
  - Buddy System
  - Slab Allocation

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
- Pros and Cons

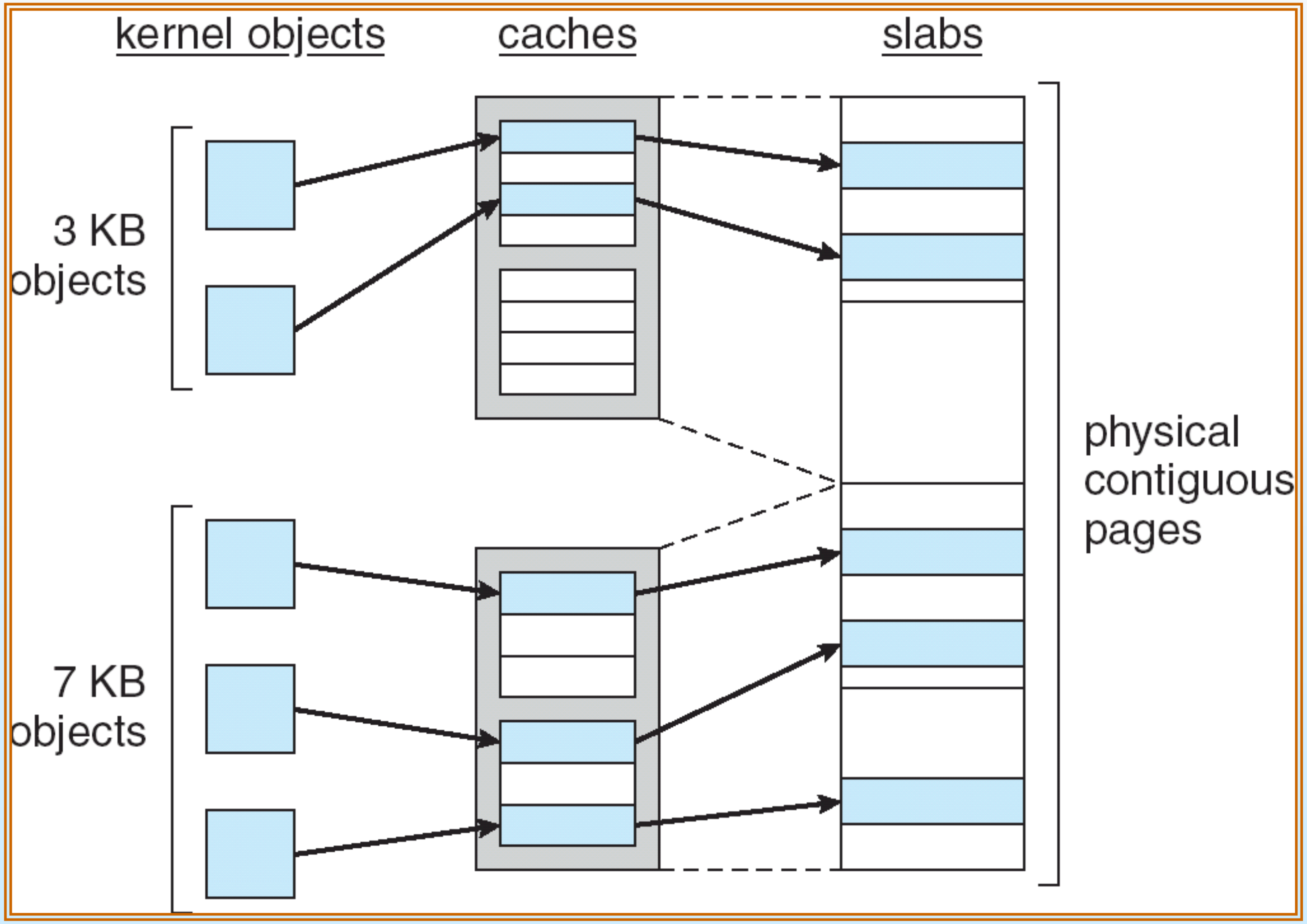
Buddy System Allocator



Slab Allocator

- **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 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
    - Compare the costs of $s * \alpha$ and $s * (1- \alpha)$?
    - α near zero $\Rightarrow$ prepaging loses

Other Issues – Page Size

- Page size selection must take into consideration:
  - Table size
    - E.g., 4 MB virtual memory (2^{22}) 4096 pages of 1KB; 512 pages of 8KB
  - Fragmentation
    - Internal fragmentation
    - Argue for small page size, why?
  - I/O overhead
    - More from later chapter of IO
  - Locality
    - Resolution
- How about 1 byte page size?

Other Issues – TLB Reach

- TLB Reach - The amount of memory accessible
- 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;`
  - Page size is 128*4 = 512 Bytes
  - 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

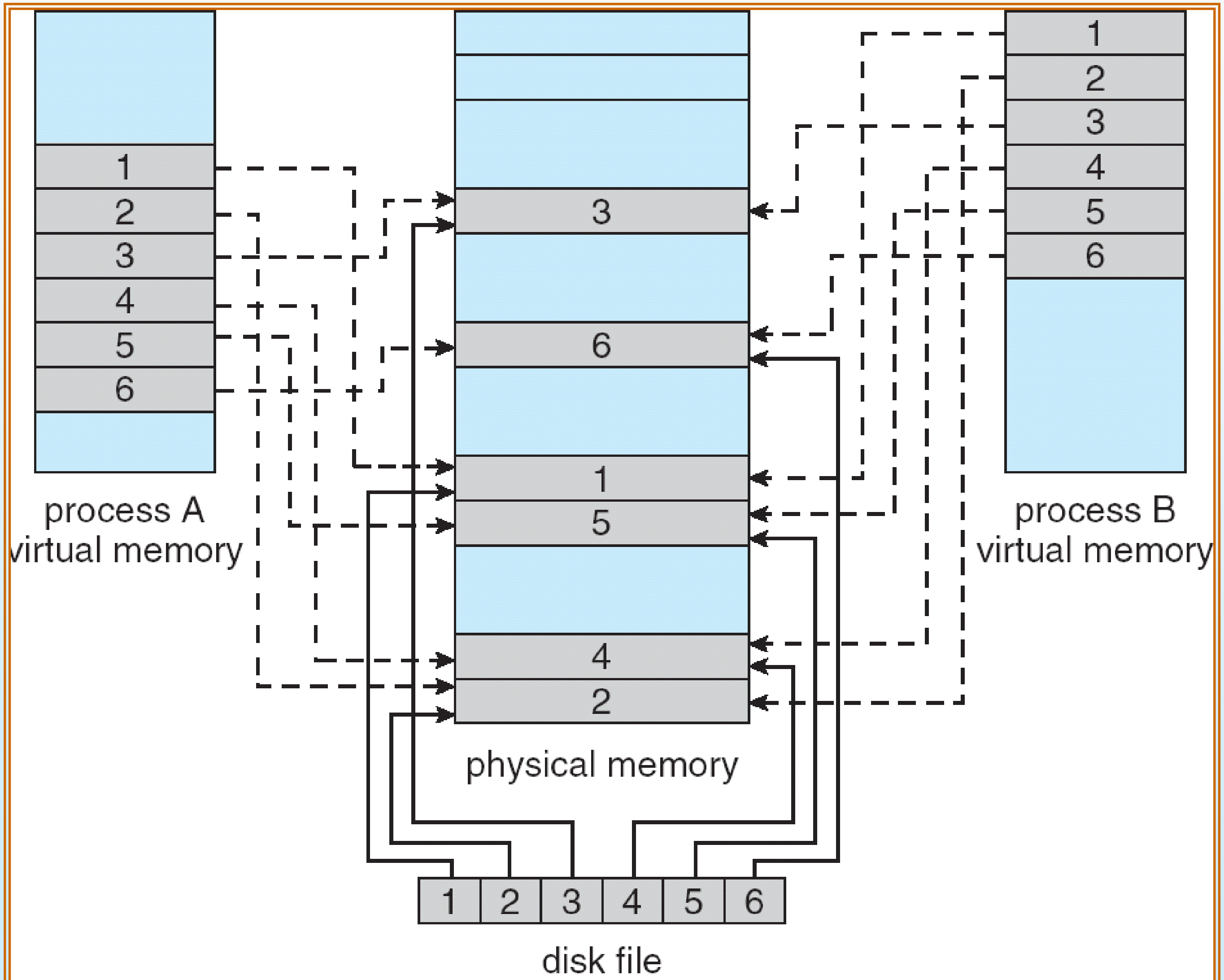
```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
          data[i,j] = 0;
```

128 page faults

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



Summary

- Virtual Memory enables us to map a large logical address space onto a smaller physical memory
- Demand Paging
  - Multiprogramming
  - CPU utilization
- Page replacement
- Page allocation policy
  - Working-set model
- Kernel memory allocation

Backup

Working Set

- LRU, 3 memory pages, 12 page faults

Memory page	A	B	C	D	A	B	C	D	E	F	G	H
1	A			D			C			F		
2		B			A			D			G	
3			C			B			E			H

Working Set

- LRU, 4 memory pages, 8 page faults

Memory page	A	B	C	D	A	B	C	D	E	F	G	H
1	A				*				E			
2		B				*				F		
3			C				*				G	
4				D				*				H

Working Set

- LRU, 5 memory pages, 8 page faults

Memory page	A	B	C	D	A	B	C	D	E	F	G	H
1	A				*					F		
2		B				*					G	
3			C				*					H
4				D				*				
5									E			