Operating Systems (INFR09047)

2019/2020 Semester 2

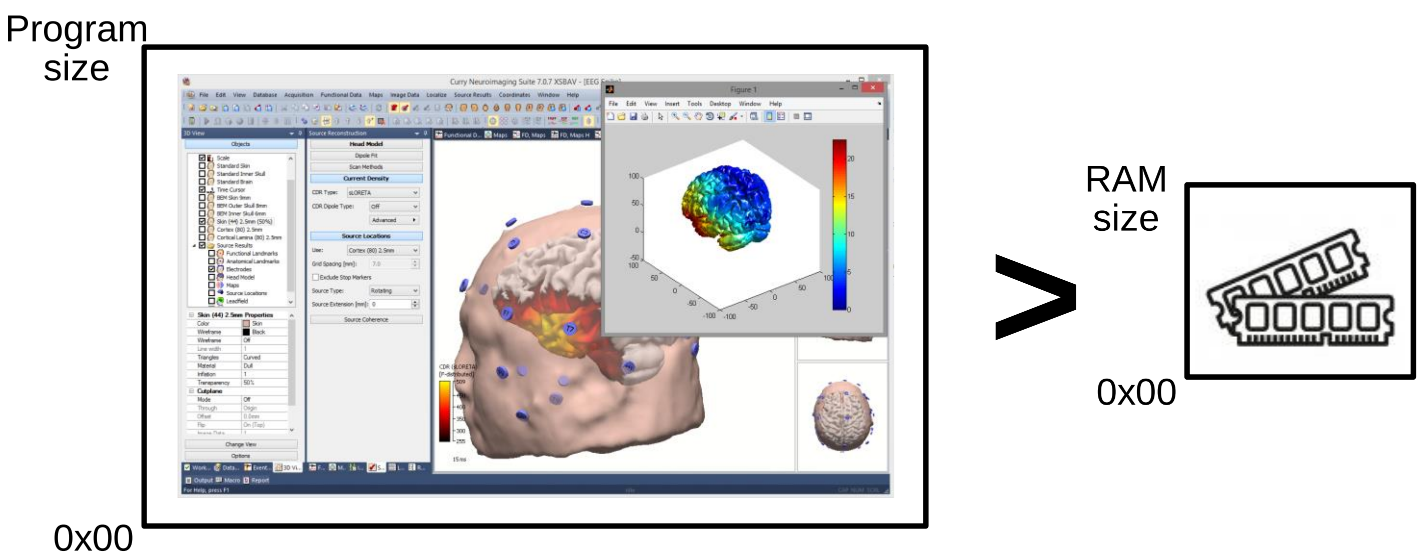
Virtual Memory

abarbala@inf.ed.ac.uk

Chapter 10

Large Programs

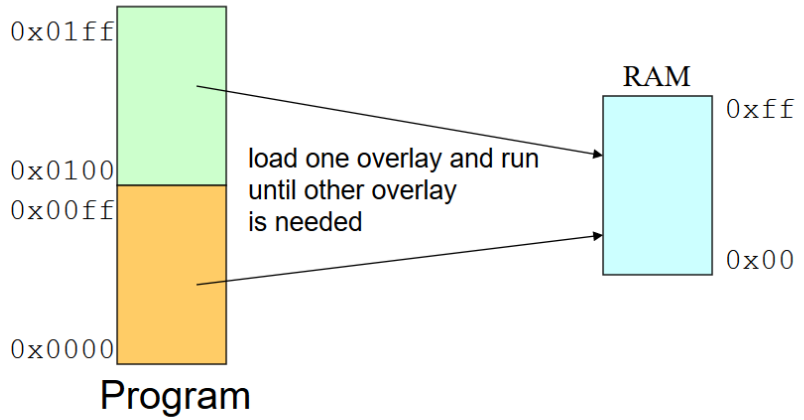
- How to deal with a program that is **bigger than** the available physical memory?



- Could a program **correctly execute** even if **it is not all in memory at all times**?

Overlays #1

- Only load **part** of the program at any time
- Programmer breaks **address space into pieces** that fit into memory
  - Constrained by **physical memory size**
- Pieces called **overlays**, are loaded and unloaded by the program



Overlays #2

- **Overlay manager** (part of the program, not the OS)
  - Loads an overlay when it is not in RAM
  - Eventually unloads an overlay previously in RAM
- Overlays **Mechanism**
  - One root segment (always in RAM)
    - Includes overlay manager
  - 2 or more memory partitions
  - Within each partition any number of overlay segments
  - Only 1 overlay segment can be in a partition at a given time

2_1	2_2	2_3	2_4	high address
1_1	1_2	1_3	1_4	
root segment				low address

- segments 1_1 .. 1_4 share the same memory area
- so do segments 2_1 .. 2_4

Virtual Memory

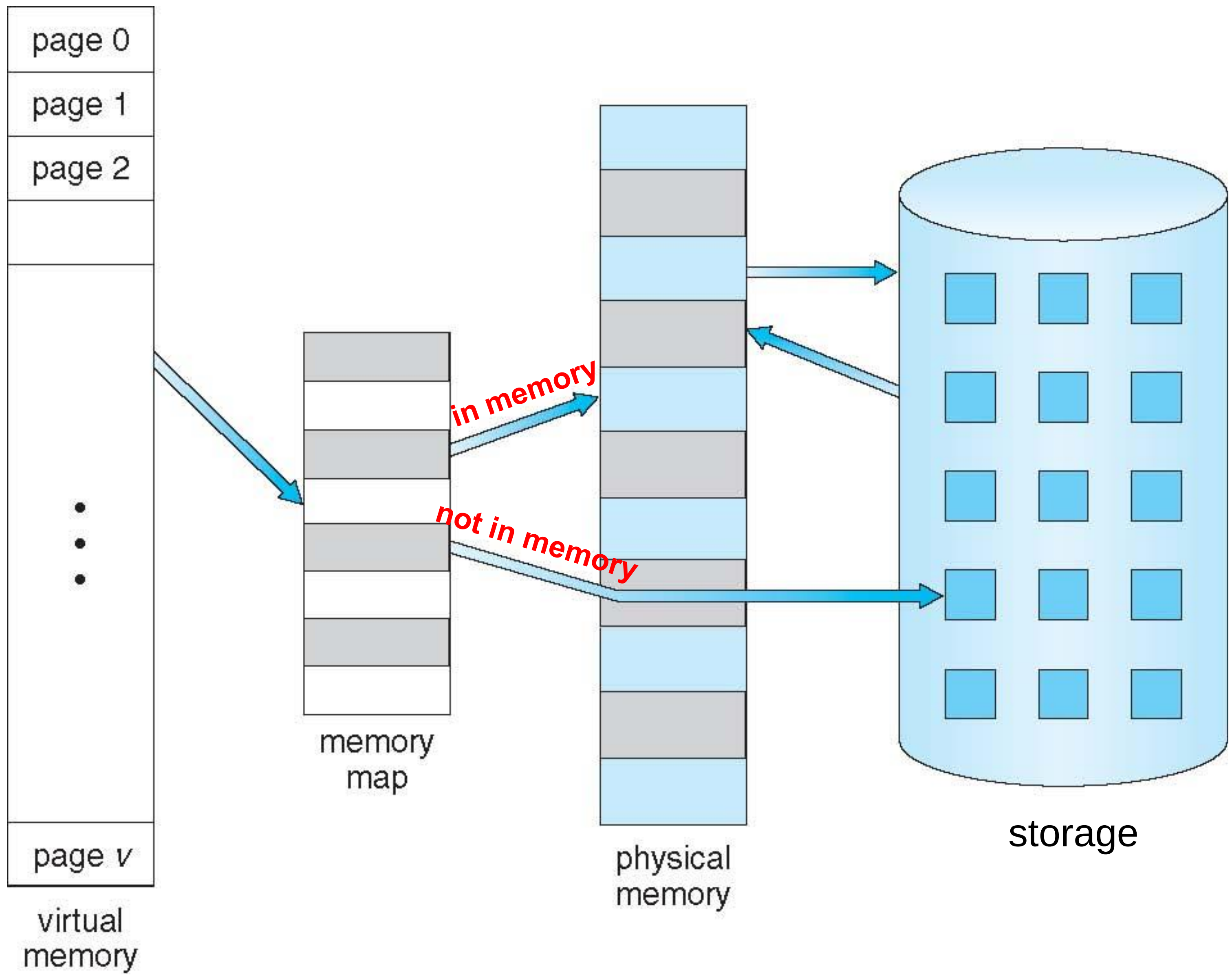
- Fully **decouples** address space from physical memory

- Allows a **larger logical address space** than physical memory

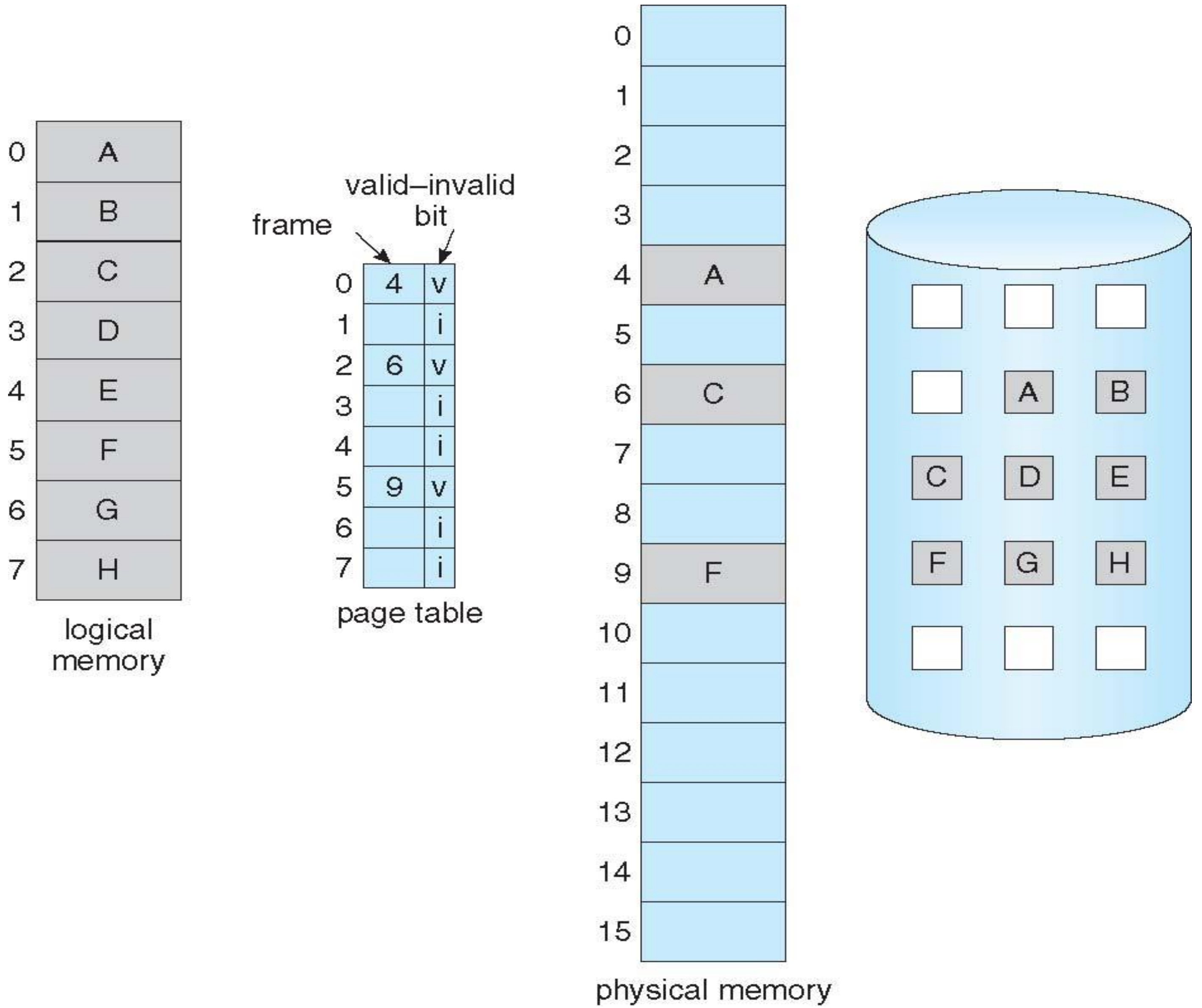
- **Paged Virtual Memory**
  - Based on hardware, with operating system support
  - Transparent to programmer, no programmer involvement

- All pages of address space **do not need** to be in memory
  - The full address space **on disk**
    - page-sized blocks
  - Main memory used as a **cache**
- Needed pages transferred to a free page frame in memory
  - If no free page frames available, find one to evict

Virtual Memory Larger Than Physical Memory



Page Table When Some Pages Are Not in Main Memory

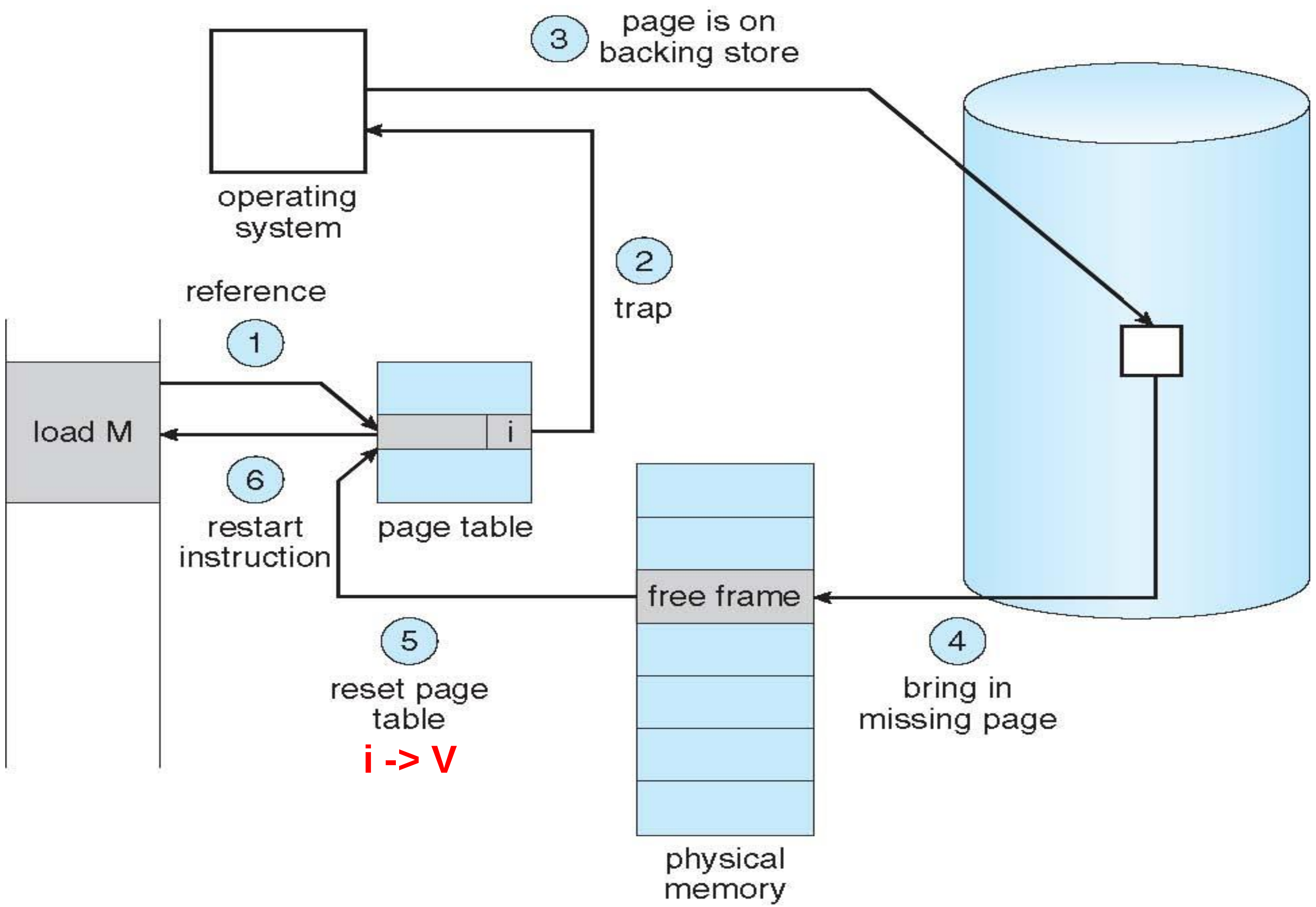


Page Fault

1. Software accesses a page that is not in memory
   – Before the access, the relative page table entry is invalid
2. Hardware triggers a **page fault** (exception)
3. Operating System looks check internal data structures
   – Invalid reference, abort the original software
   – Not in memory, continue
4. Operating System finds a free frame
   – Swaps page into frame via scheduled disk operation
5. Operating System set internal data structures to indicate page now in memory
   – Set valid bit
6. Operating System restarts the instruction that caused the **page fault**

(see next slides)

Steps in Handling a Page Fault



Valid pages are accessed directly by the hardware ***without OS involvement***

Demand Paging

- Pages brought into memory **when accessed first**
  - On program's demand
    - Program may start with no pages in memory
  - Only code/data needed by a process needs to be loaded
    - What's needed changes over time
- Few systems try to **anticipate** future needs
- Pages may be **clustered**
  - OS keeps track of pages that should come and go together
  - Bring in all when one is referenced

- Demand paging can be **expensive**
  - Heavily depends on storage latency

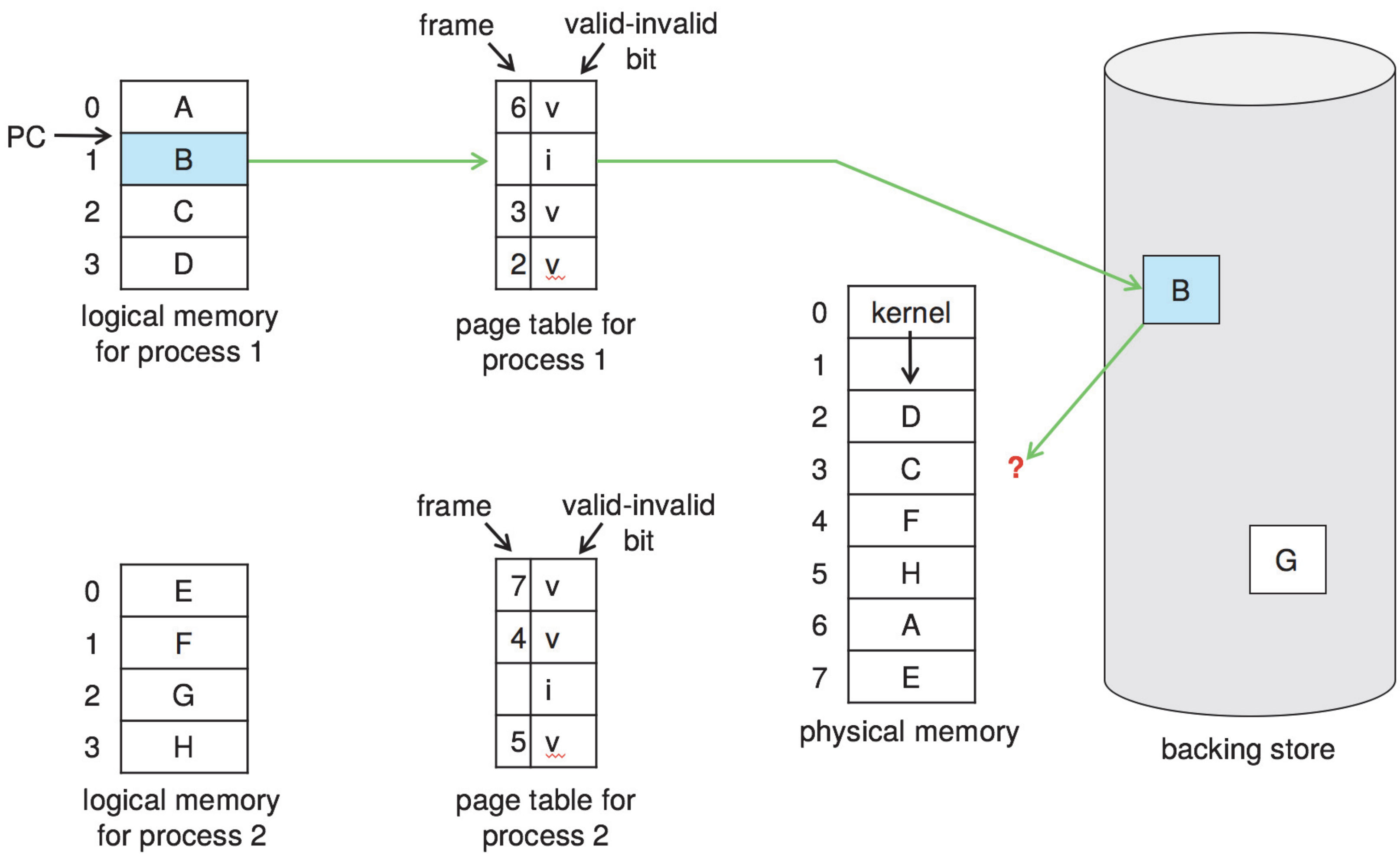
Page Allocation and Replacement

- When you read in a page, where does it go?
  - If there are free page frames, grab one
    - This is **page allocation**
  - If there are no free page frames, must **evict** one
    - This is **page replacement**
      - Mechanism
      - Algorithm

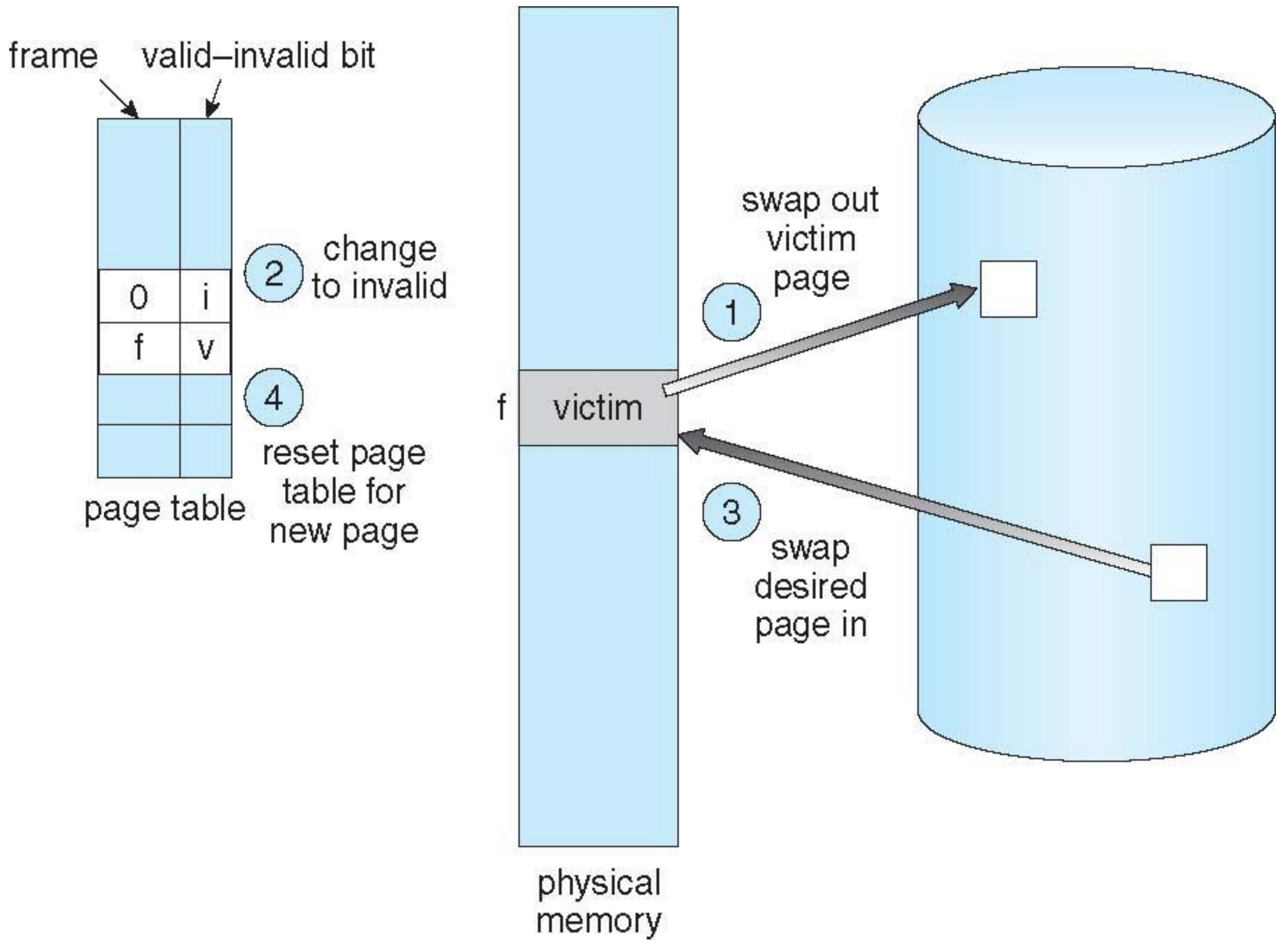
- OS tries to keep a pool of free pages around
  - To avoid the cost of eviction

- High degree of multiprogramming, causes over-allocation
  - All memory is in use
  - Need to evict

What to Do When All Memory is in Use?



Page Replacement Mechanism



Page Replacement Algorithm

- **What page to evict?**
  - Reduce page-fault rate by selecting **best victim page**
    - Reduce page-fault overhead
  - **Best victim page** is the one that will **never be touched again**
    - Don’t needed in the near future
  - **Belady’s Theorem**
    - Evicting the page that won’t be used for the longest period of time minimizes page fault rate

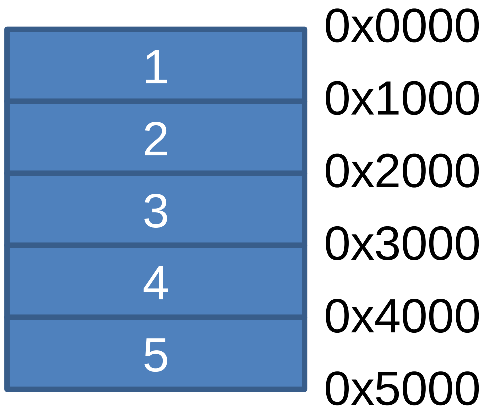
  - Evict **unmodified** pages first
    - **No need to write** them back to disk

- **Examine** page replacement algorithms
  - Assume that a process pages against itself
  - Using a fixed number of page frames

String of Memory References

- Ordered list of pages the program will reference
  - Example 1, 2, 3, 4, 1, 2, 5, …

```
MOV R0, 0x0123
MOV R1, 0x1234
MOV R2, 0x2345
MOV R3, 0x3456
MOV 0x0100, R0
MOV 0x1200, R1
MOV R4, 0x4567
```



First-In-First-Out (FIFO) Algorithm

- **Replace page that has been inserted first and is still in**

- 3 physical page frames, 5 virtual pages
- Reference string: **0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4**

Tail of List

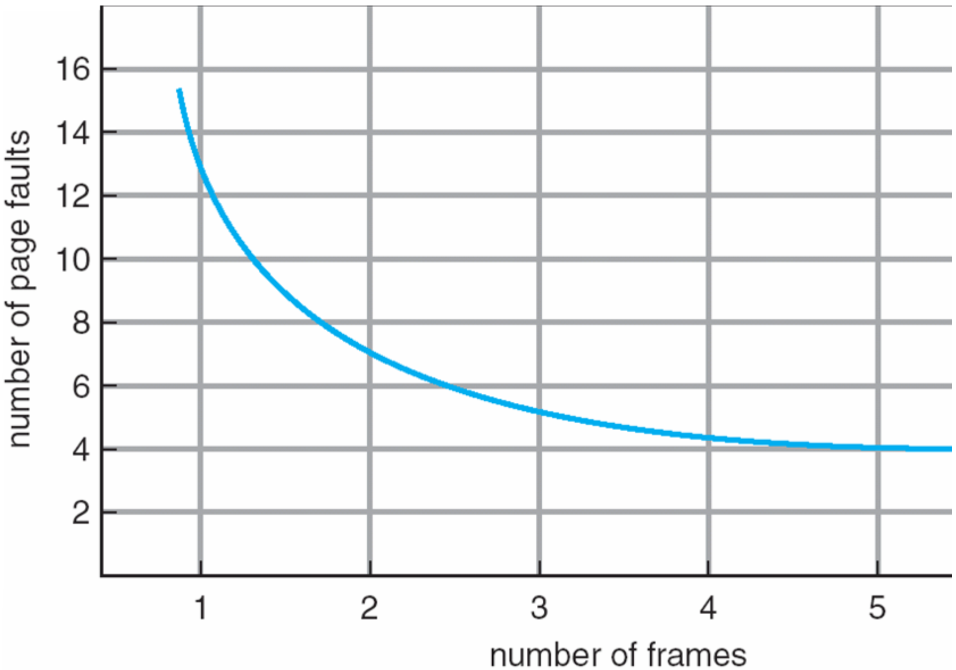
	0	1	2	3	0	1	4	0	1	2	3	4
	0	1	2	3	0	1	4	4	4	2	3	3
3 frames		0	1	2	3	0	1	1	1	4	2	2
			0	1	2	3	0	0	0	1	4	4
	P	P	P	P	P	P	P			P	P	

Beginning of List

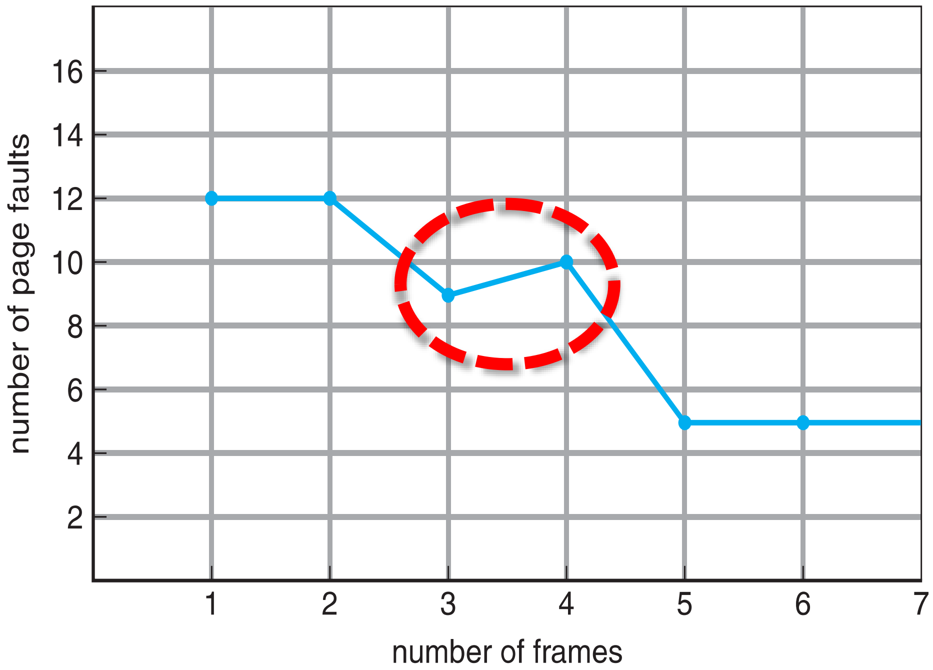
9 Page faults

- Easy to implement
  - Maintain a linked list of all pages in the order they come into memory

Belady’s Anomaly



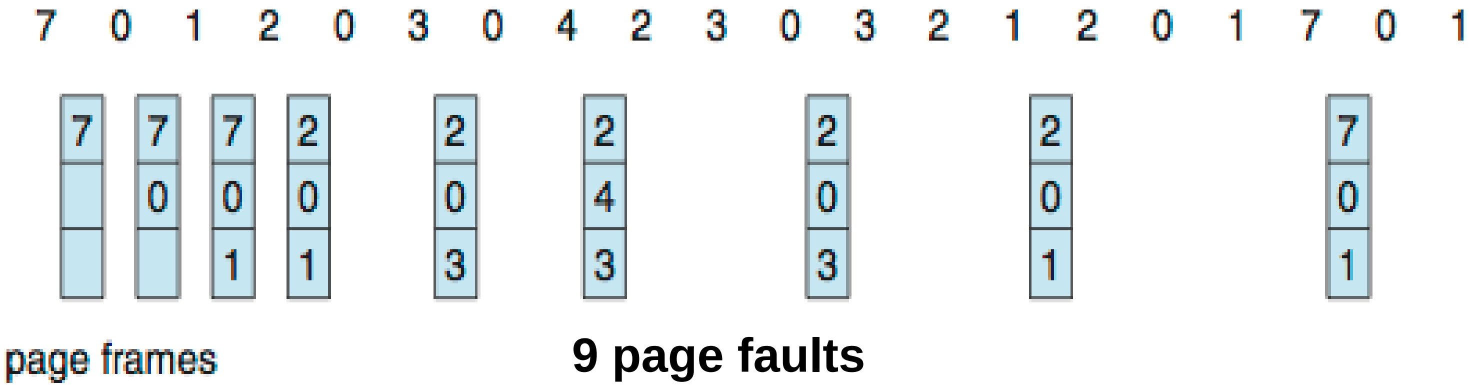
Expected Behavior
(more page frames less page faults)



FIFO Behavior
(more page frames do not
guarantee less page faults)

Optimal Algorithm

- **Replace page that will not be used for longest period**
  - lowest page-fault rate
  - Never suffer from Belady’s anomaly
- 3 physical page frames, 8 virtual pages
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**

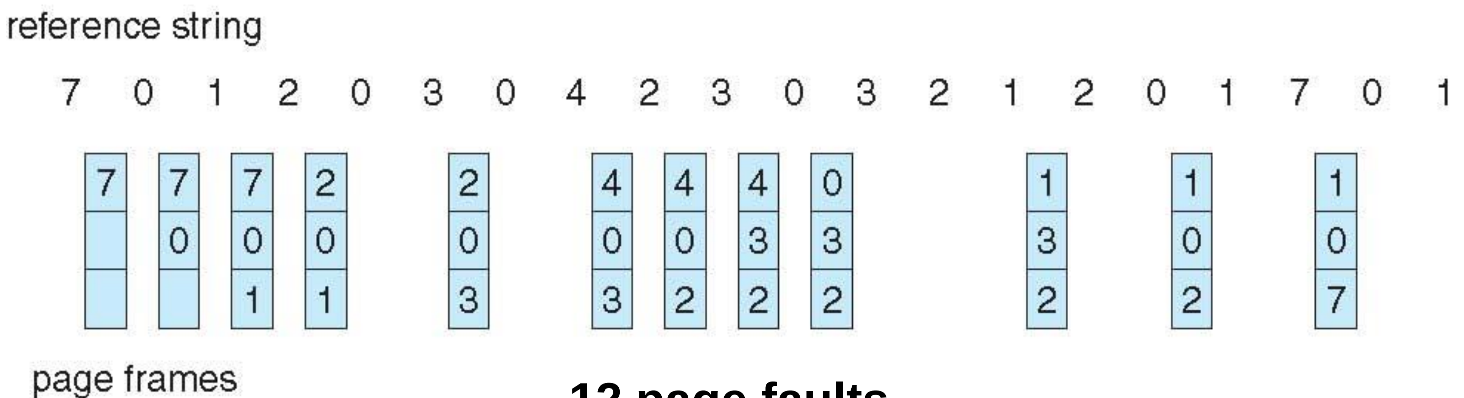


9 page faults

- How do you know what page will not be used?
  - **Can’t read the future**
- Used for measuring how well your algorithm performs

Least Recently Used (LRU) Algorithm

- **Replace page that has not been used in the most amount of time**
  - Use past knowledge rather than future
  - Never suffer from Belady's anomaly (stack algorithm)
- 3 physical page frames, 8 virtual pages
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**



12 page faults

- Generally good algorithm and frequently used
- How to implement?
  - Associate time of last use with each page
  - Requires substantial **hardware assistance**

Approximating LRU

- Use **page table entry bits**, maintained **by hardware**
  - Page referenced (if accessed or not)
  - Page modified (if access was in write)
- Keep a history/counter **for each page, in software**

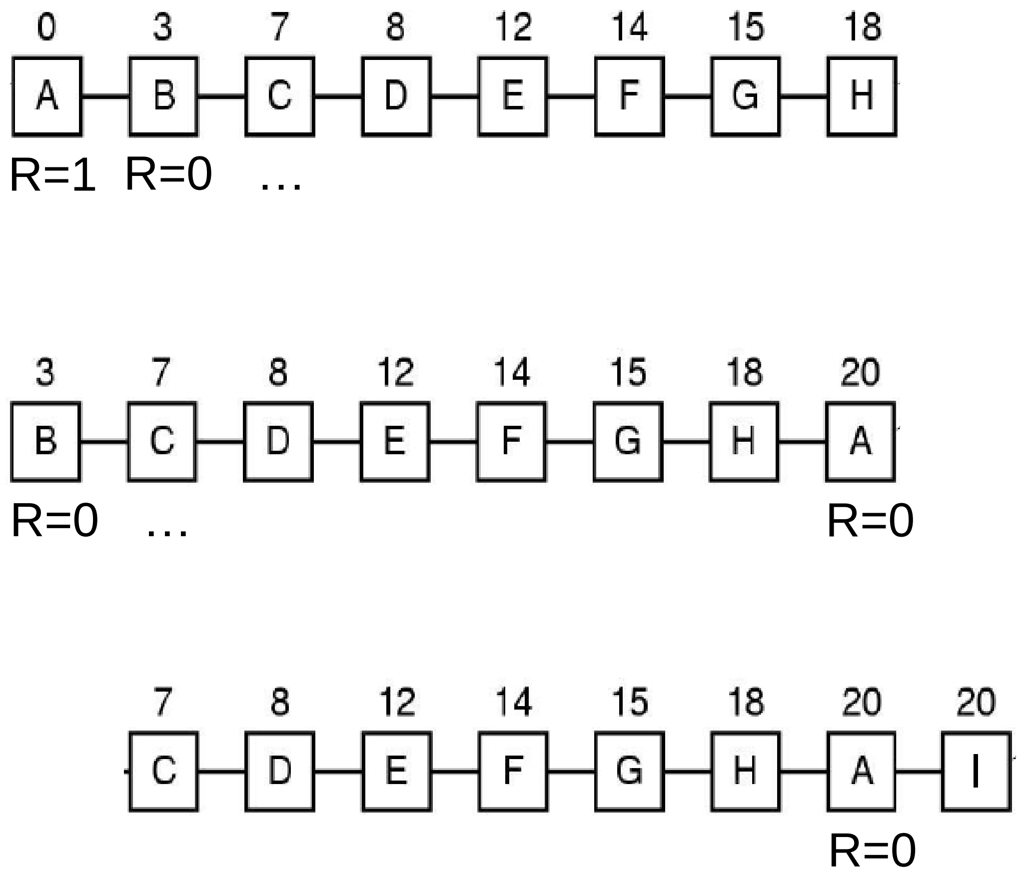
- **History-based** page replacement algorithms
  - Recording the reference bits at regular intervals
    - keep history bits in a table in memory
  - *Aging*
  - *Second-chance (clock)*
  - *Enhanced second-chance*
- **Counting-based** page replacement algorithms
  - Keep a counter of the number of references that have been made
  - *Least frequently used (LFU)*
    - the page with the smallest count be replaced
  - *Most frequently used (MFU)*
    - the page with the highest count be replaced

Second Chance

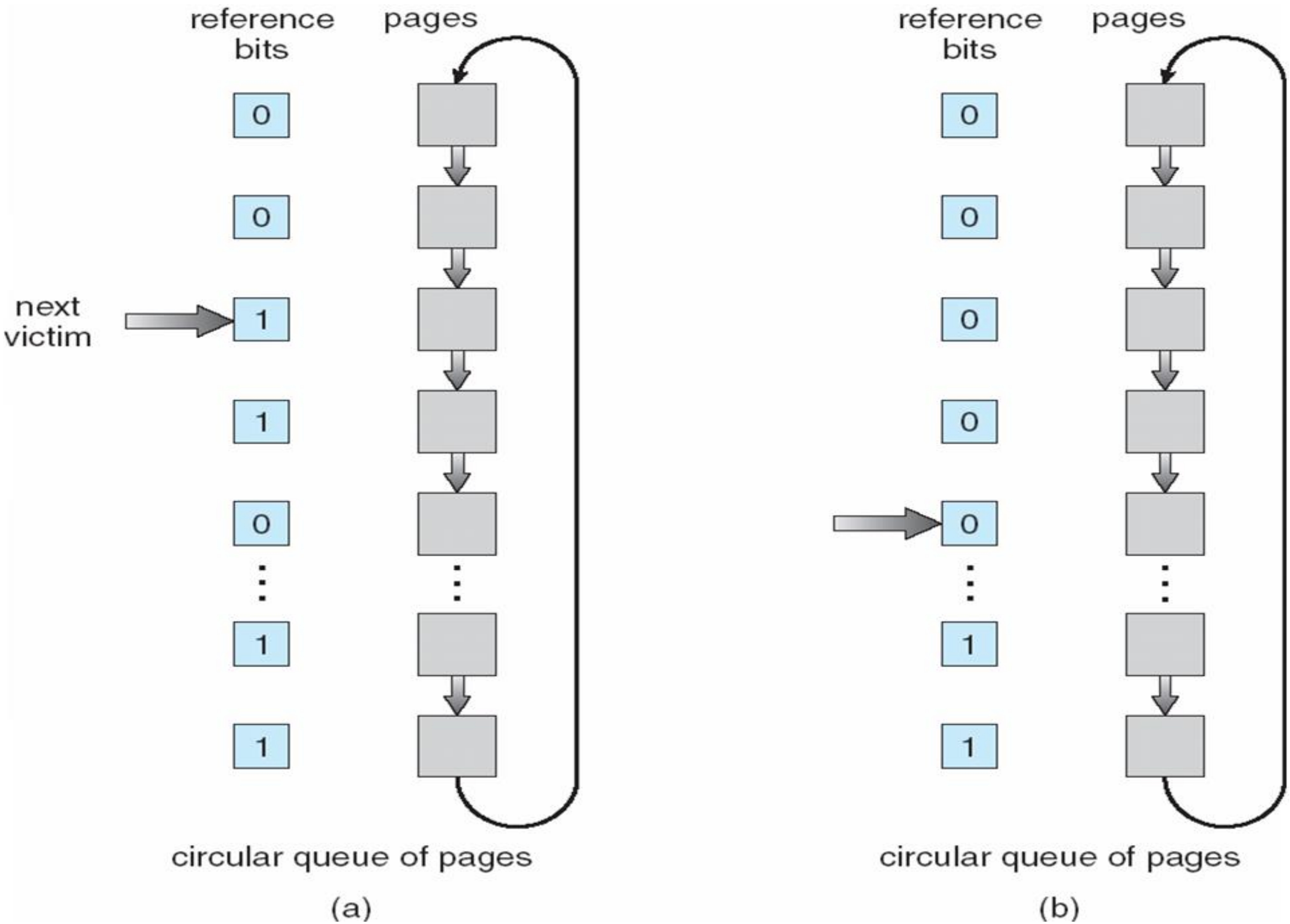
- FIFO variant
  - Adds the concept of usage (references)

- Examine pages in FIFO order starting from beginning of list
  - Consider “reference bit”, R=0 **has not** been referenced
    a) IF R=0, remove page, go to **c)**
    b) IF R=1, set R=0 and place it at the end of FIFO list (hence, the second chance), go to **a)**
    c) Add new page at the end of FIFO (with R=0)
  - If not enough replaces, revert to pure FIFO on second pass

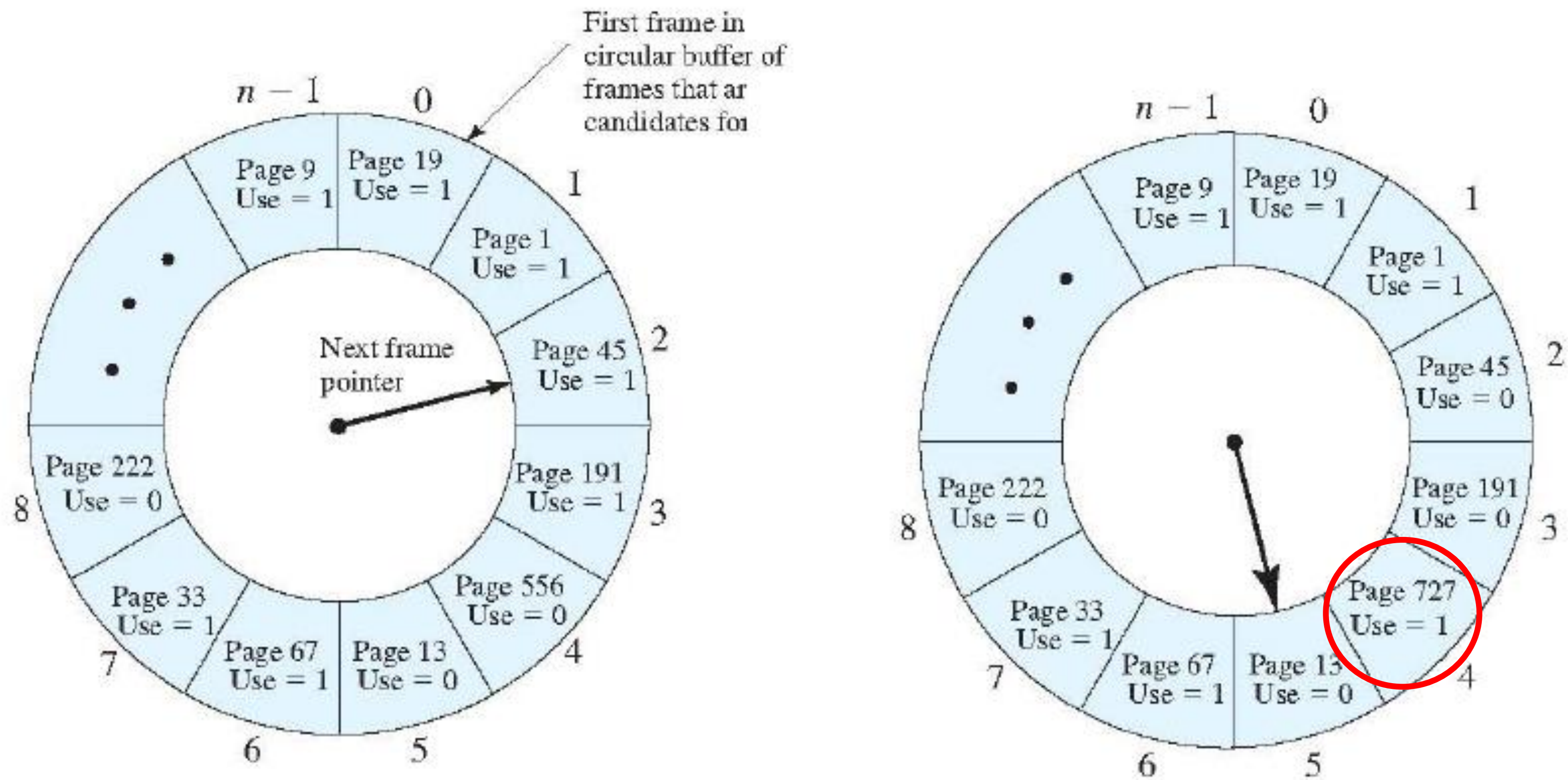
Second Chance: Example



Second Chance Clock



Second Chance Clock: Example



(a) State of buffer just prior to a page replacement

(b) State of buffer just after the next page replacement

“Use” = “Reference”

Frames Among Processes

- Frame **Allocation**
  - **Equal:** an equal share each
  - **Proportional:** a share based on the program size
  - …

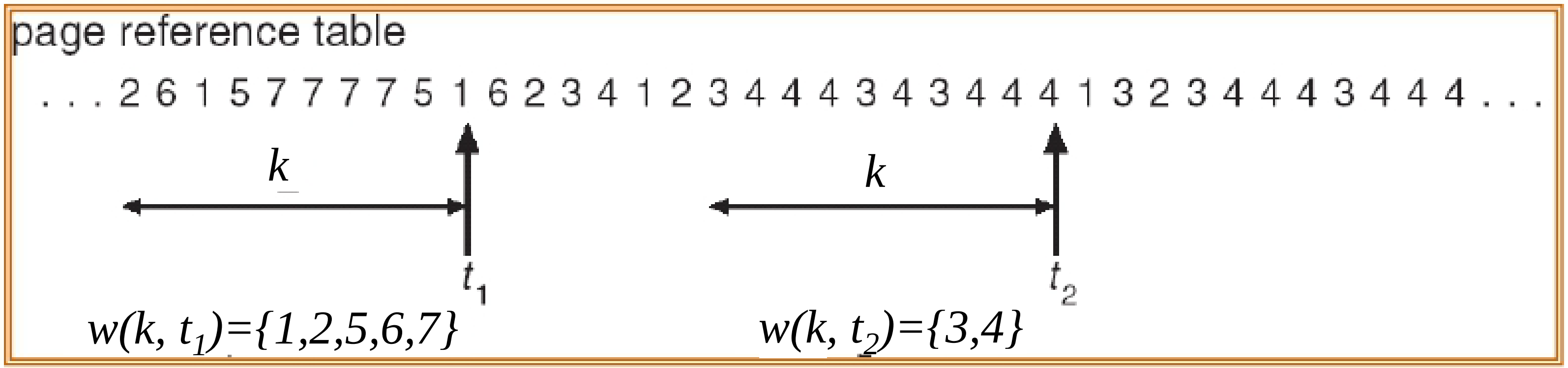
- Frame **Replacement**
  - **Local:** each process is given a limit of pages it can use
    - Process "pages against itself" (evicts its own pages)
      - Doesn't affect other processes
    - Poor utilization of (all) free page frames, and long access time
  - **Global:** the "victim" is chosen from among all page frames
    - Regardless of owner
    - Processes' page frame allocation can vary dynamically
    - Risk of global thrashing (see later)

How many pages a program really needs?

The *working set model of program behavior*

- Working set of a process is used to model the dynamic locality of its memory usage
  - working set = set of pages process currently "needs"
  - formally defined by Peter Denning in the 1960's
- **Definition**
  - WS(k,t) = {pages referenced in the time interval (t, t-k)}
    - *t:* time
    - *k:* working set *window* (measured in page refs)
    - A page is in WS only if it was referenced in the last k references
- Working set varies over the life of the program
  - so does the working set size

Working Set Model



Examples of working set for k = 10

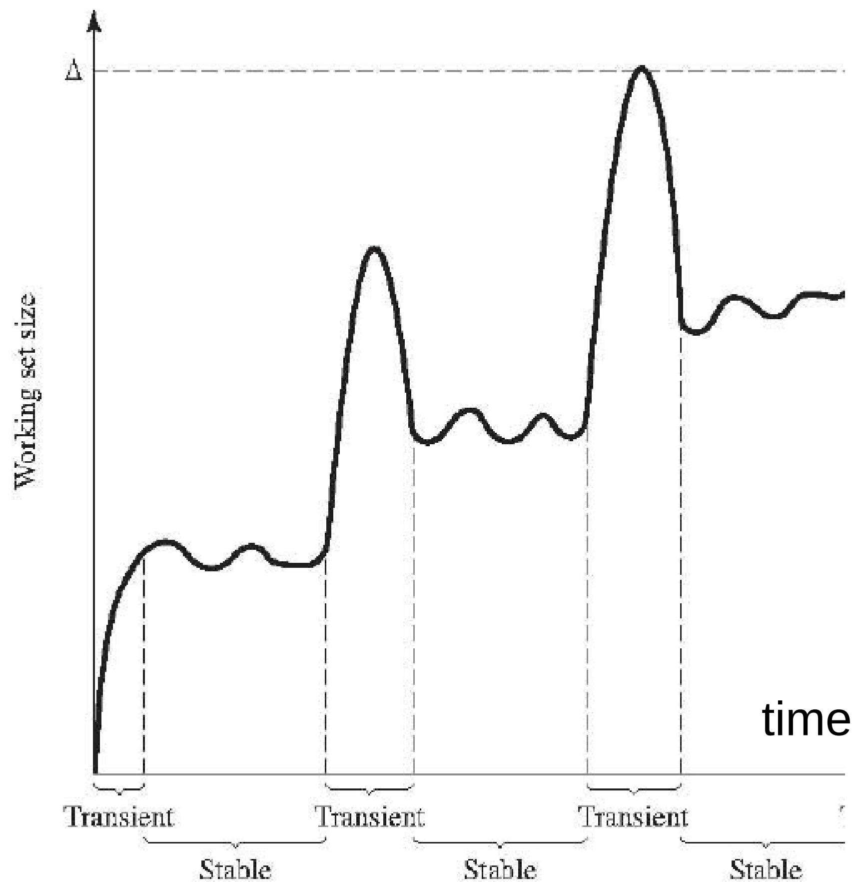
- Working set is the set of pages used by
  - the *k most recent m*emory references
- *w(k,t)* is the size of the working set at time *t*

Working Set as Defined by Window Size

Sequence of Page References	Window Size, k			
	2	3	4	5
24	24	24	24	24
15	24 15	24 15	24 15	24 15
18	15 18	24 15 18	24 15 18	24 15 18
23	18 23	15 18 23	24 15 18 23	24 15 18 23
24	23 24	18 23 24	•	•
17	24 17	23 24 17	18 23 24 17	15 18 23 24 17
18	17 18	24 17 18	•	18 23 24 17
24	18 24	•	24 17 18	•
18	•	18 24	•	24 17 18
17	18 17	24 18 17	•	•
17	17	18 17	•	•
15	17 15	17 15	18 17 15	24 18 17 15
24	15 24	17 15 24	17 15 24	•
17	24 17	•	•	17 15 24
24	•	24 17	•	•
18	24 18	17 24 18	17 24 18	15 17 24 18

Working Set Size

- Working set size, |WS(k,t)|
  - Changes with program locality
- During periods of poor locality
  - More pages are referenced
  - Working set size is larger
- The working set must be all in memory
  - Otherwise heavy faulting
  - Thrashing



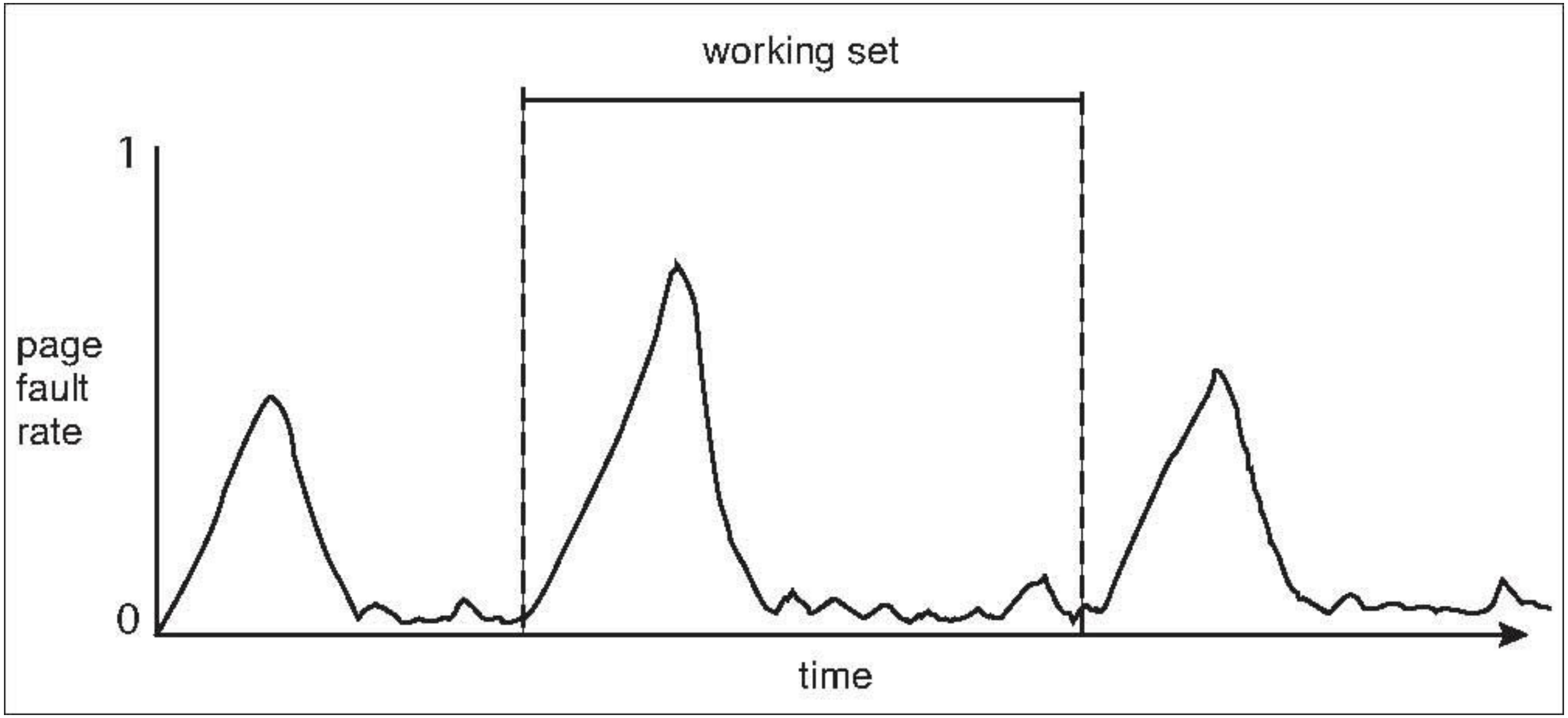
(Hypothetical) Working Set Allocation Algorithm

- Estimate |WS(k,0)| for a process
  - Allow process to start only if OS can provide that many frames
- Use a **local replacement algorithm**
  - Make sure that the working set are occupying the process's frames
- Track each process's working set size
  - Re-allocate page frames among processes dynamically

- How to keep track of processes' WSs?
  - Use reference bit with a fixed-interval timer interrupt

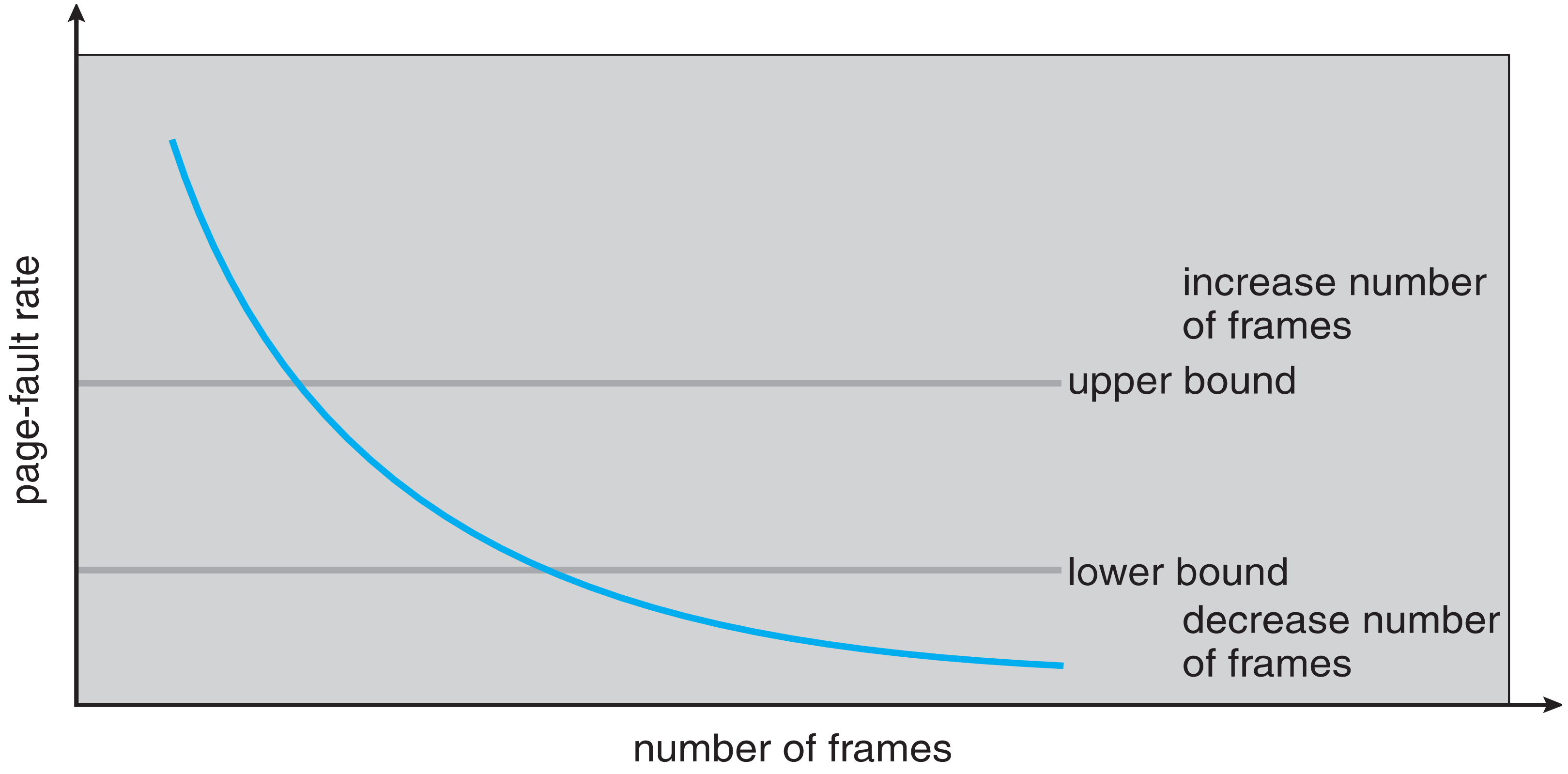
Working Sets and Page Fault Rates

- Relationship between **working set** and **page-fault rate** of a process
  - Working set changes over time
  - Page-fault rate peaks and then valley
- Can Page-fault rate/frequency be used to **steer allocations**?



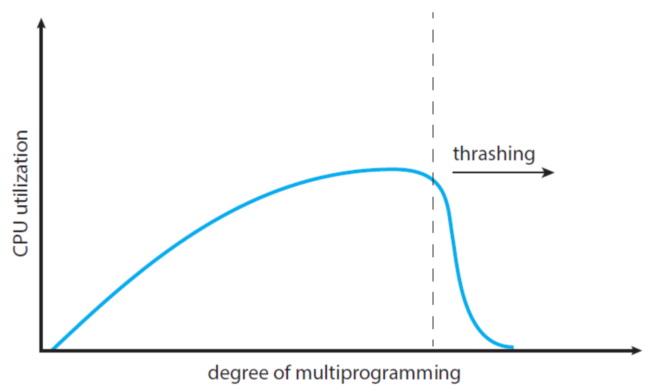
Page-Fault Frequency Allocation

- Establish “acceptable” **page-fault frequency** (**PFF**) rate
- Use a local **replacement algorithm**
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



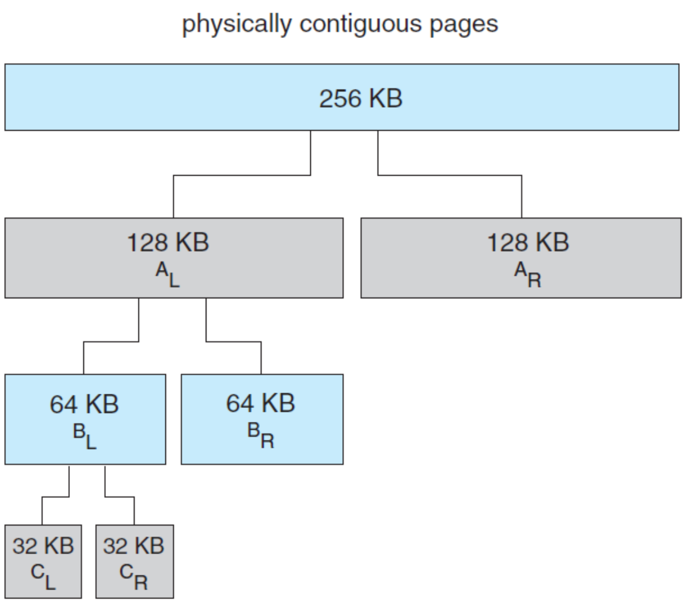
Thrashing

- System spends
  - Most of its time **servicing page faults**
  - Little time doing **useful work**
- Could be that there is **enough memory**
  - But a **poor replacement algorithm**
    - Incompatible with program behavior
- Could be that **memory is over-committed**
  - OS sees **CPU poorly utilized** and adds more processes
    - Many active processes, requesting memory



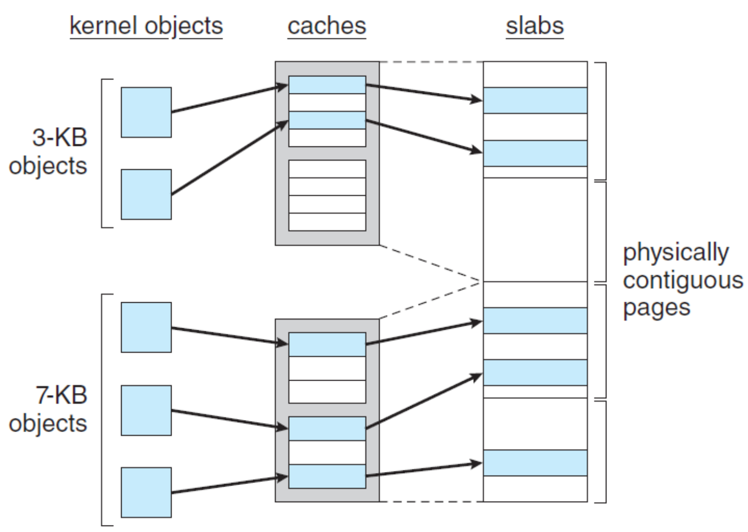
Kernel Memory Allocation: Buddy System

- Power of 2 allocator of **physically contiguous pages**
  - Satisfies requests in units sized as power of 2
  - Not power of 2 request size is rounded up
  - If request is smaller
    - Break down in 2 buddies that are also power of 2
  - Two equal size free buddies may be coalesced



Kernel Memory Allocation: Slab Allocation

- **Slab** is made up of one or more **physically contiguous pages**
- A **cache** consists of one or more slabs
- There is a cache for each **unique kernel data structure**
  - Each cache populated with objects
    - instantiations of the kernel data structure
- If there are **free slabs**, the allocation is immediate
  - No search for memory space
- SLOB and SLUB (more performant) variations in Linux



Summary

- Overlays
- Paged Virtual memory
- Page faults
- Demand paging
- Page replacement
  - FIFO, Optimal, LRU, Second Chance, Clock
  - local, global
- Locality
  - temporal, spatial
- Working set
- Thrashing
- Kernel Memory Allocation

CPU Cache vs. Virtual Memory “as a Cache”

- CPU Cache is a **hardware** component
  - It is **completely** transparent to the programmer
  - CPU cache **holds** data coming from memory
    - CPU cache fetches data from memory transparently
- Virtual memory “as a cache”, is a **hardware** component + **OS**
  - It is transparent to the application, but not to the OS
  - Virtual memory “as a cache” **holds** data coming from storage
    - The OS moves memory from storage to memory