Chapter 9
Virtual Memory

Zhi Wang
Florida State University

Content

- Background
- Demand paging
- Copy-on-write
- Page replacement
- Thrashing
- Memory-mapped files
- Operating-system examples

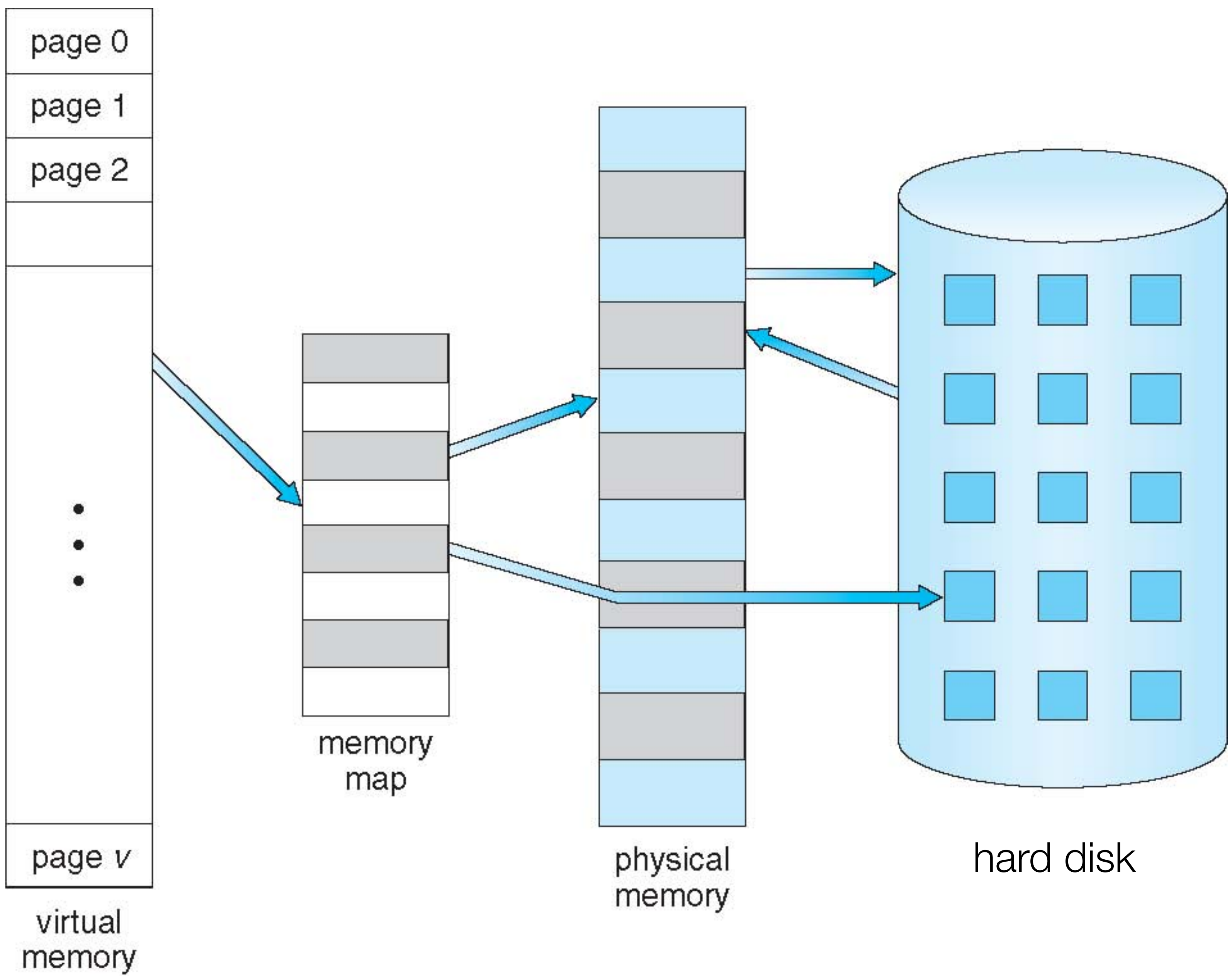
Background

- Code needs to be in memory to execute, but entire program **rarely** needed or used at the same time
  - error handling code, unusual routines, large data structures
- Consider ability to execute **partially-loaded program**
  - program no longer constrained by limits of physical memory
  - programs could be larger than physical memory

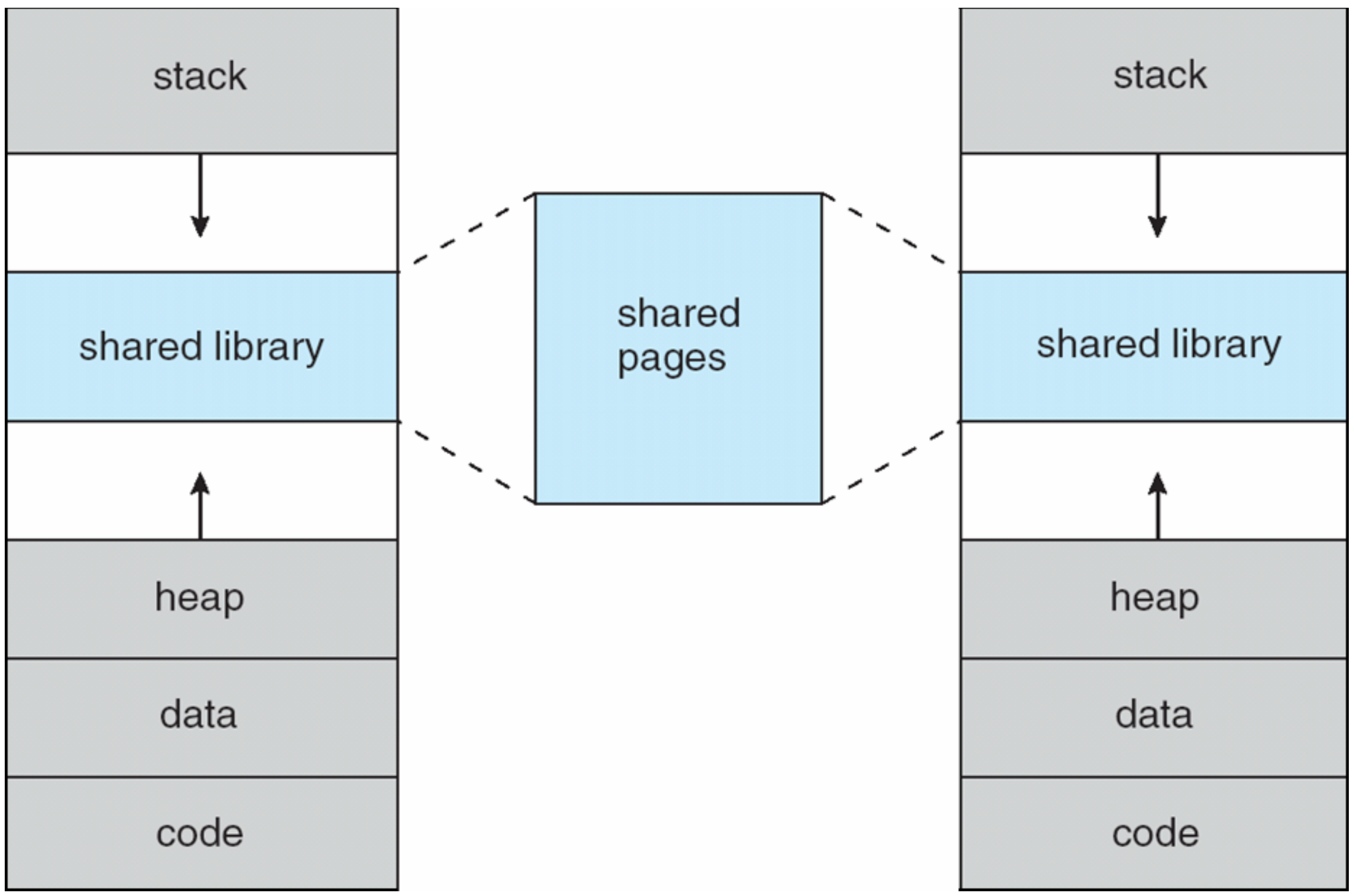
Background

- Virtual memory: separation of **logical memory** from **physical memory**
  - **only part of the program needs to be in memory for execution**
    - logical address space can be much larger than physical address space
    - more programs can run concurrently
    - less I/O needed to load or swap processes (part of it)
  - allows memory (e.g., shared library) to be shared by several processes
  - allows for more efficient process forking (**copy-on-write**)
- Virtual memory can be implemented via:
  - **demand paging**
  - **demand segmentation**

Virtual Memory Larger Than Physical Memory



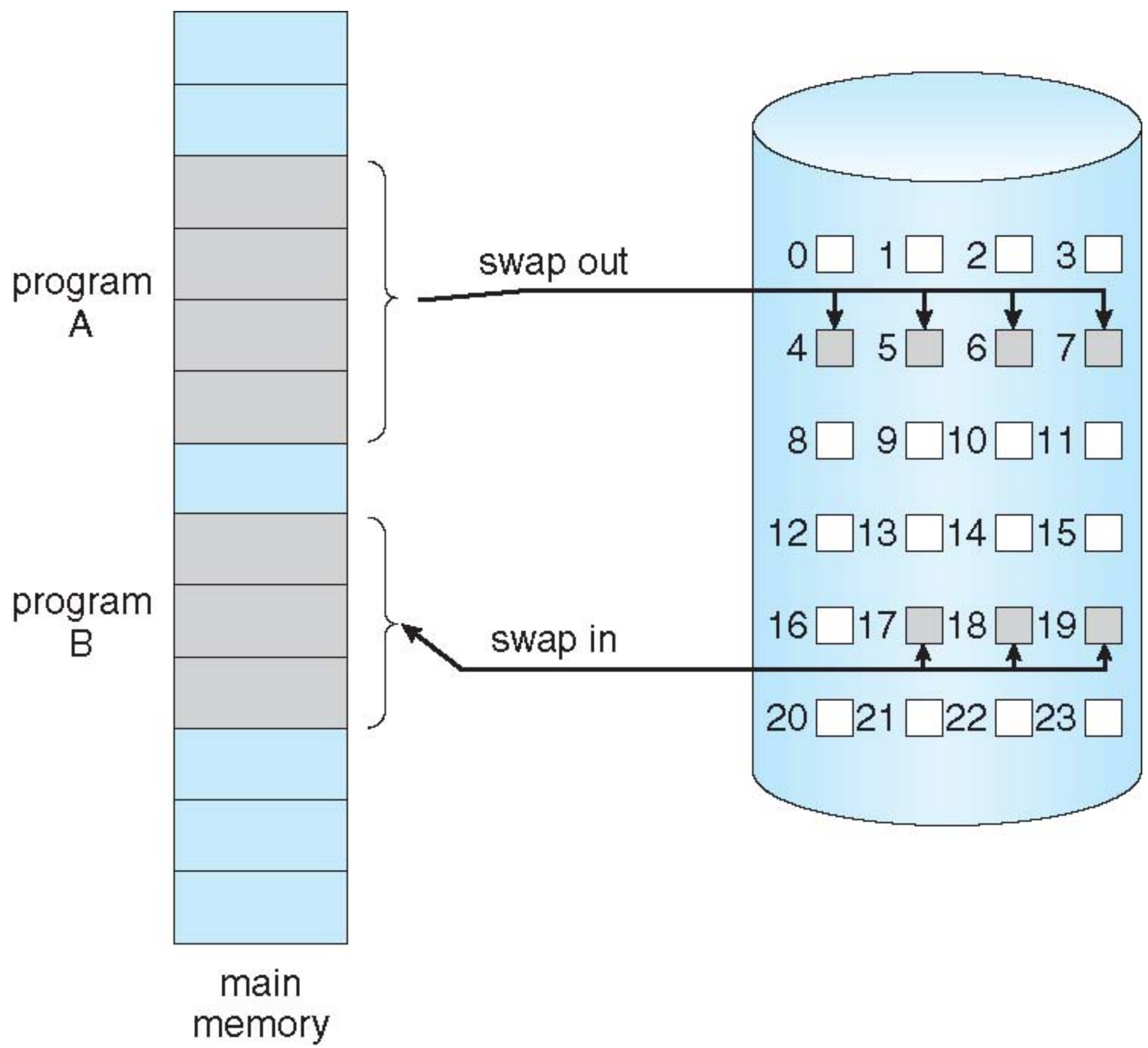
Shared Library Using Virtual Memory



Demand Paging

- **Demand paging** brings a page into memory only when it is accessed
  - if page is invalid ➠ abort the operation
  - if page is valid but not in memory ➠ bring it to memory via swapping
  - no unnecessary I/O, less memory needed, faster response, more apps
- **Lazy swapper**: never swaps a page in memory unless it will be needed
  - the swapper that deals with pages is also caller a pager
- **Pre-Paging**: pre-page all or some of pages a process will need, before they are referenced
  - it can reduce the number of page faults during execution
  - if pre-paged pages are unused, I/O and memory was wasted
    - although it reduces page faults, total I/O# likely is higher

Demand Paging

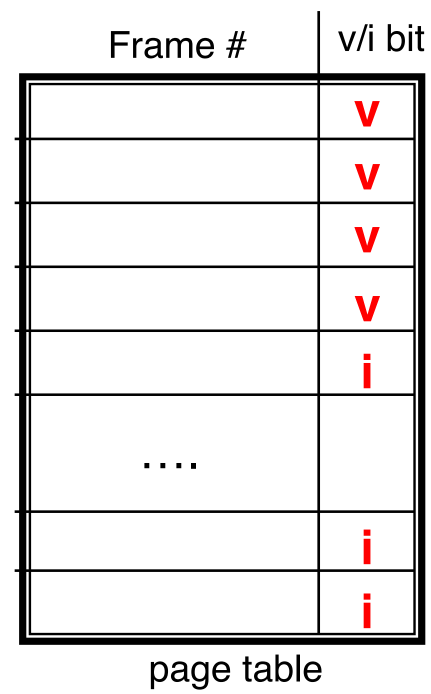


Demand Paging

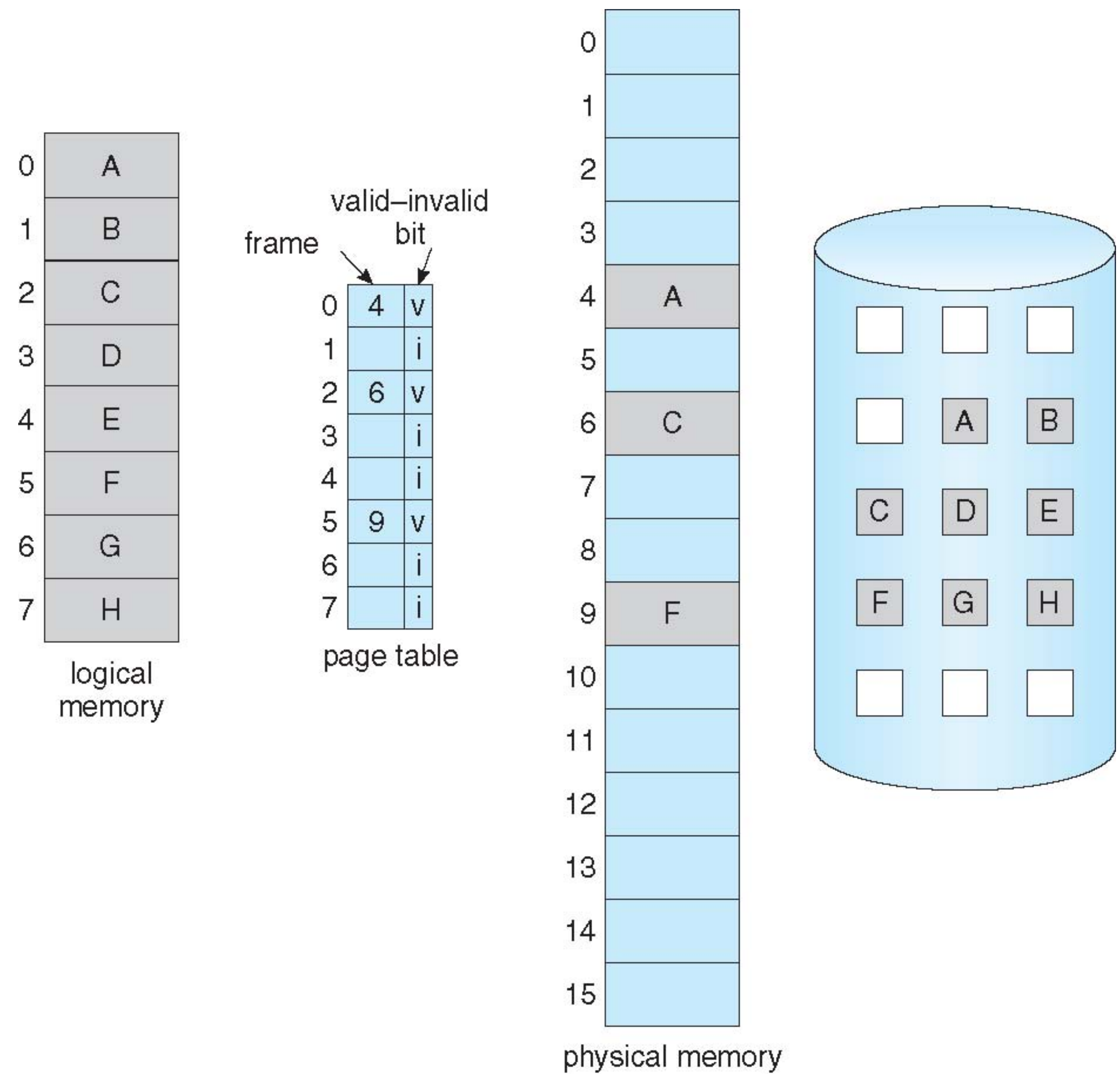
- Extreme case: start process with no pages in memory (aka. **pure demand paging**)
  - OS sets instruction pointer to first instruction of process
    - invalid page ➠ page fault
  - every page is paged in on first access
    - **program locality** reduces the overhead
  - an instruction could access multiple pages ➠ multiple page faults
    - e.g., instruction, data, and page table entries for them
- Demand paging needs hardware support
  - page table entries with **valid / invalid bit**
  - **backing storage** (usually disks)
  - **instruction restart**

Valid-Invalid Bit

- Each page table entry has a valid–invalid (present) bit
  - *V* ➠ in memory (memory is resident), *I* ➠ not-in-memory
  - initially, valid–invalid bit is set to *i* on all entries
  - during address translation, if the entry is invalid, it will trigger a **page fault**
- Example of a page table snapshot:



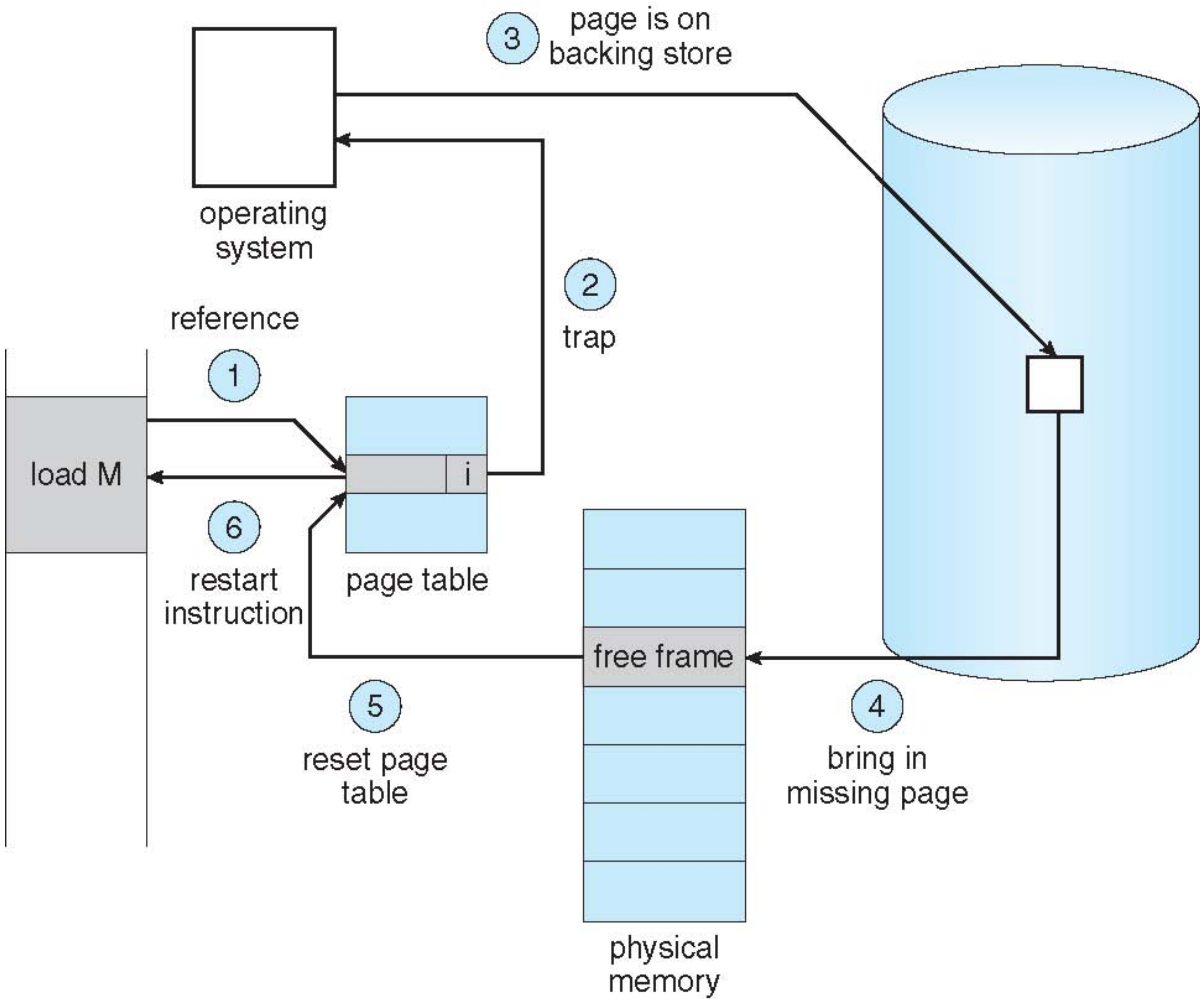
Page Table (Some Pages Are Not in Memory)



Page Fault

- First reference to a non-present page will trap to kernel: **page fault**
- Operating system looks at memory mapping to decide:
  - invalid reference ➠ deliver an exception to the process
  - valid but not in memory ➠ swap in
    - get an empty physical frame
    - swap page into frame via disk operation
    - set page table entry to indicate the page is now in memory
    - restart the instruction that caused the page fault

Page Fault Handling



Demand Paging: EAT

- 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):

  $(1 - p)$ x memory access + p x (

  page fault overhead +

  swap page out + swap page in +

  instruction restart overhead)

Demand Paging Example

- Assume memory access time: 200 nanoseconds, average page-fault service time: 8 milliseconds
  - EAT = (1 – p) x 200 + p x (8 milliseconds)

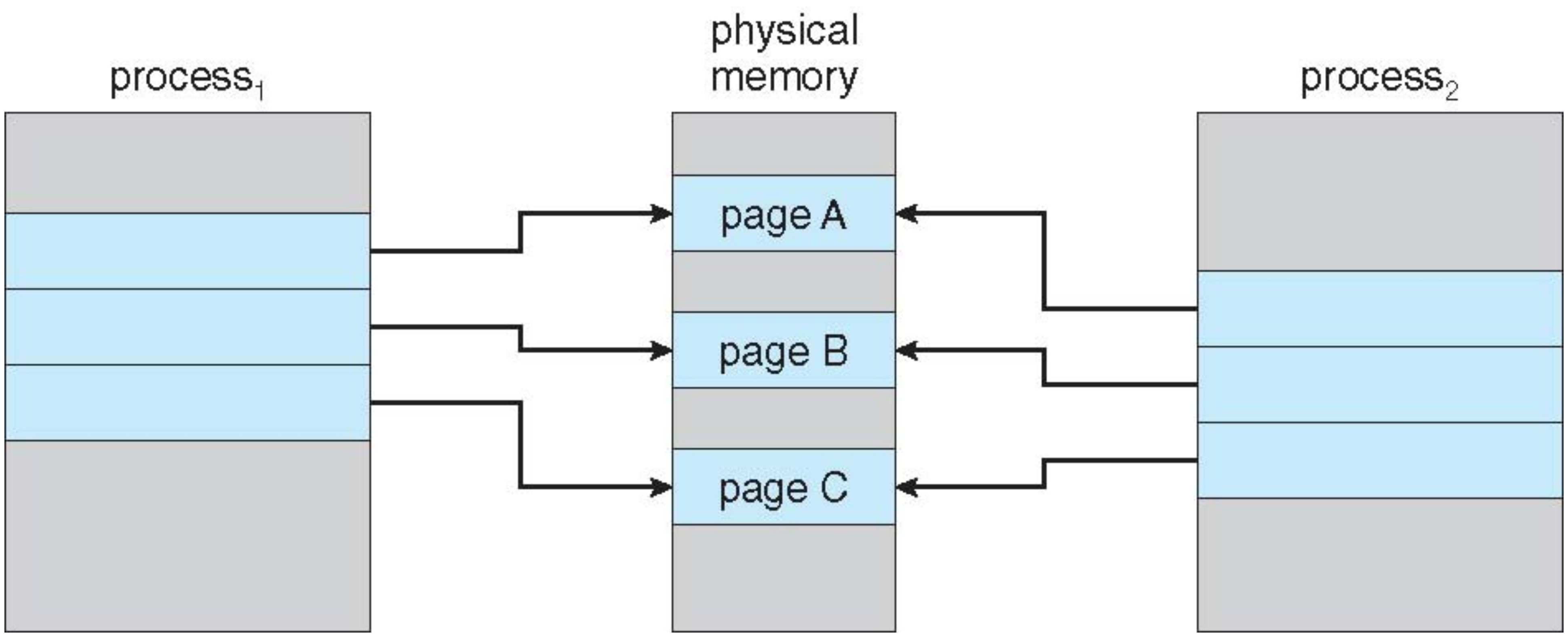
    = (1 – p) x 200 + p x 8,000,000

    = 200 + p x 7,999,800
  - if one out of 1,000 causes a page fault, then EAT = 8.2 microseconds
    - a slowdown by a factor of 40!
  - if want < 10 percent, less than one page fault in every 400,000 accesses

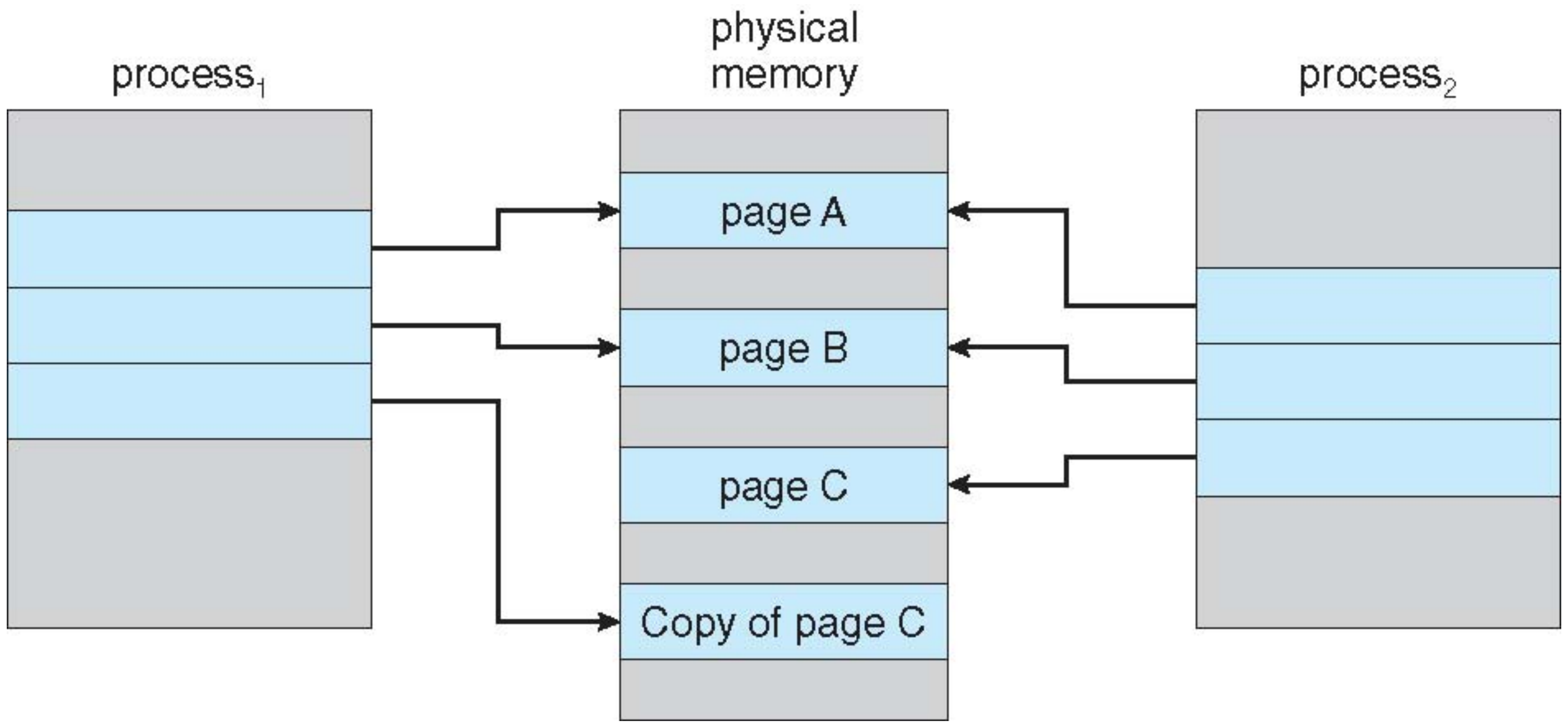
Copy-on-Write

- **Copy-on-write** (COW) allows parent and child processes to initially share the same pages in memory
  - the page is shared as long as no process modifies it
  - if either process modifies a shared page, only then is the page copied
- COW allows more efficient **process creation**
  - no need to copy the parent memory during fork
  - only changed memory will be copied later
- vfork syscall optimizes the case that child calls **exec** immediately after fork
  - parent is suspend until child exits or calls exec
  - child shares the parent resource, including the heap and the stack
    - child cannot return from the function or call exit
  - vfork could be fragile, it is invented when COW has not been implemented

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



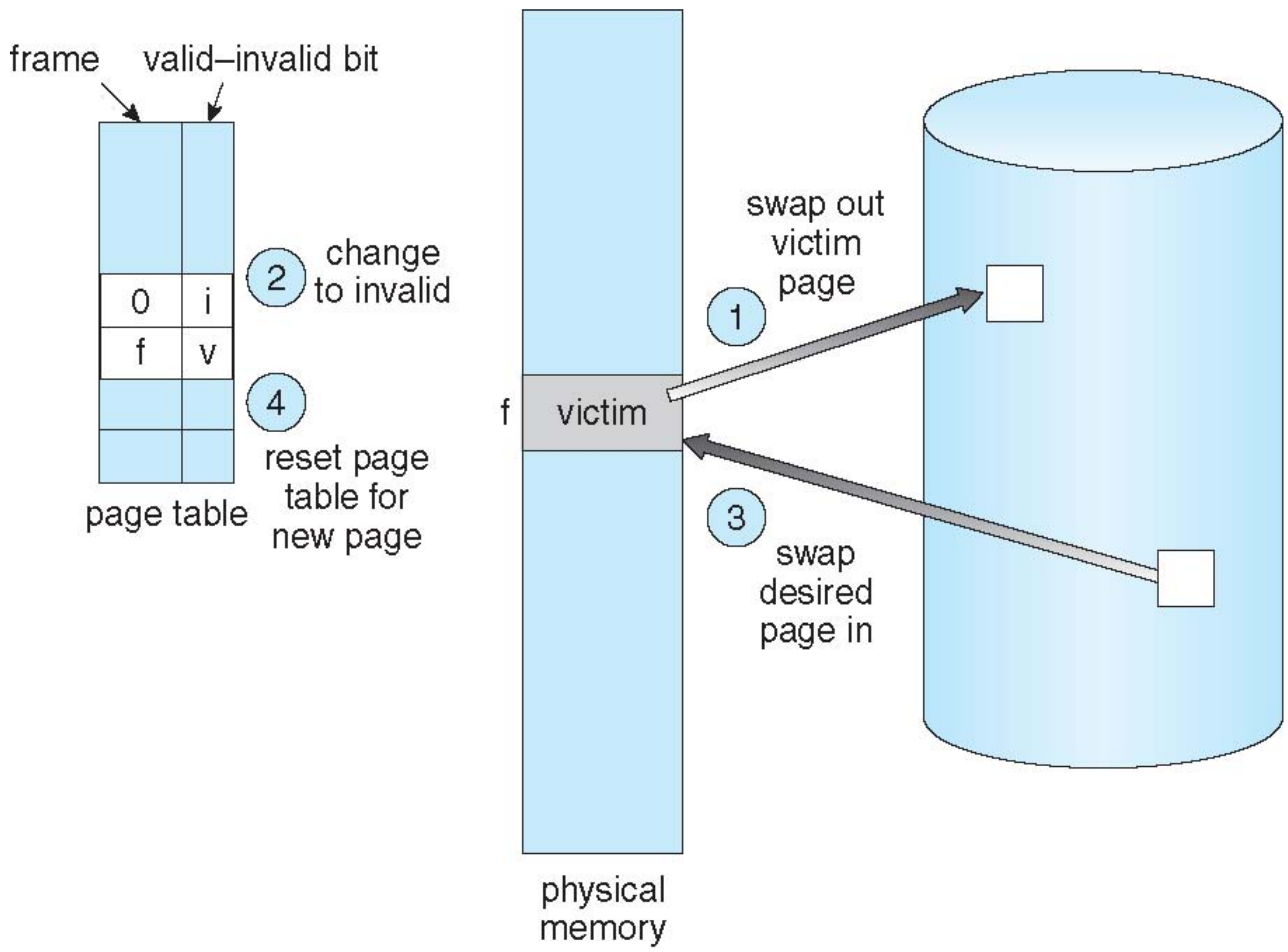
Page Replacement

- Memory is an important resource, system may run out of memory
- To prevent out-of-memory, swap out some pages
  - page replacement usually is a part of the page fault handler
  - policies to select victim page require careful design
    - need to reduce overhead and avoid **thrashing**
  - use modified (dirty) bit to reduce number of pages to swap out
    - only modified pages are written to disk
  - select some processes to kill (last resort)

Page Fault Handler (with Page Replacement)

- To page in a page:
  - find the location of the desired page on disk
  - find a free frame:
    - if there is a free frame, use it
    - if there is none, use a page replacement policy to pick a victim frame, write victim frame to disk if dirty
  - bring the desired page into the free frame; update the page tables
  - restart the instruction that caused the trap
- Note now potentially **2 page I/O** for **one page fault** ➠ increase EAT

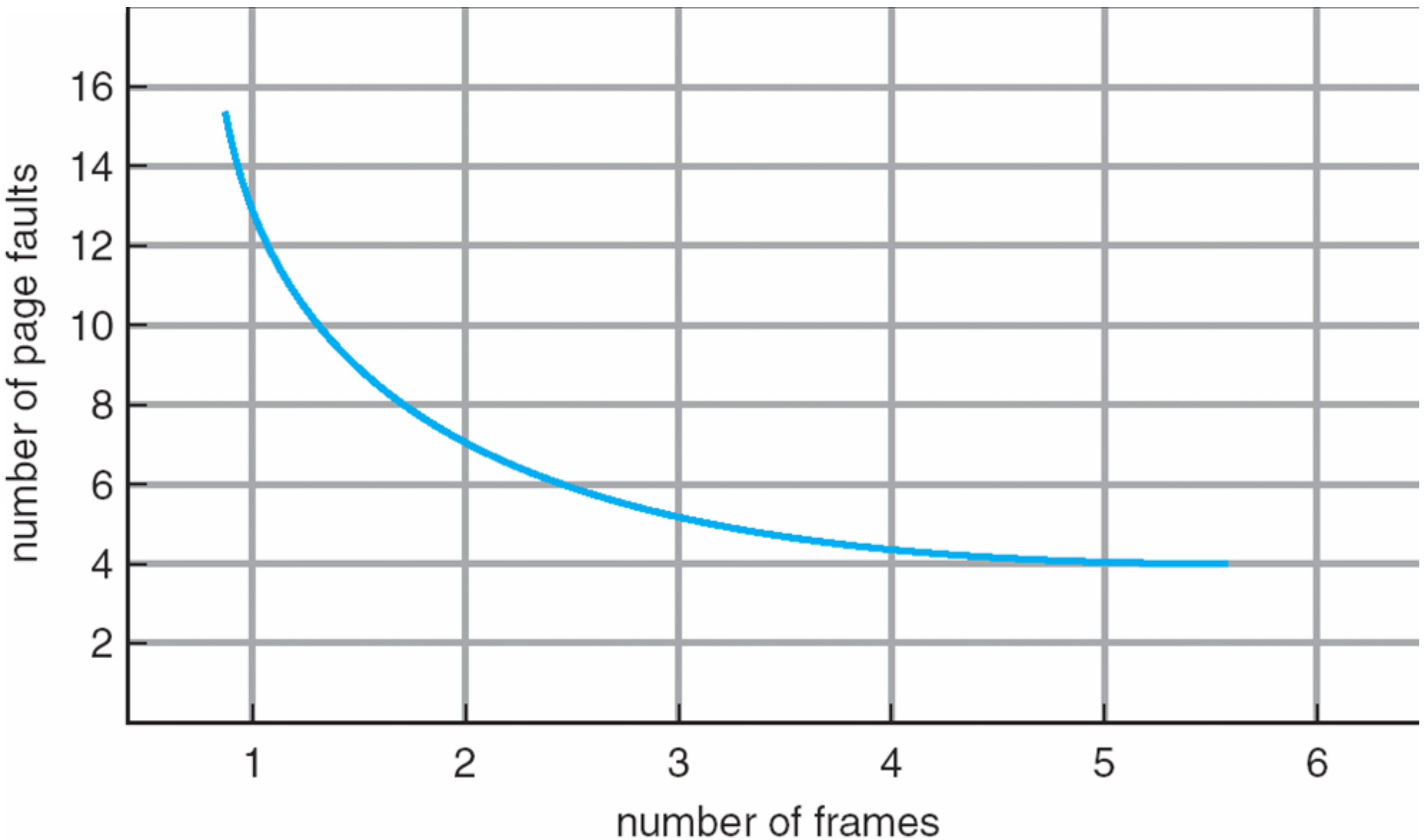
Page Replacement



Page Replacement Algorithms

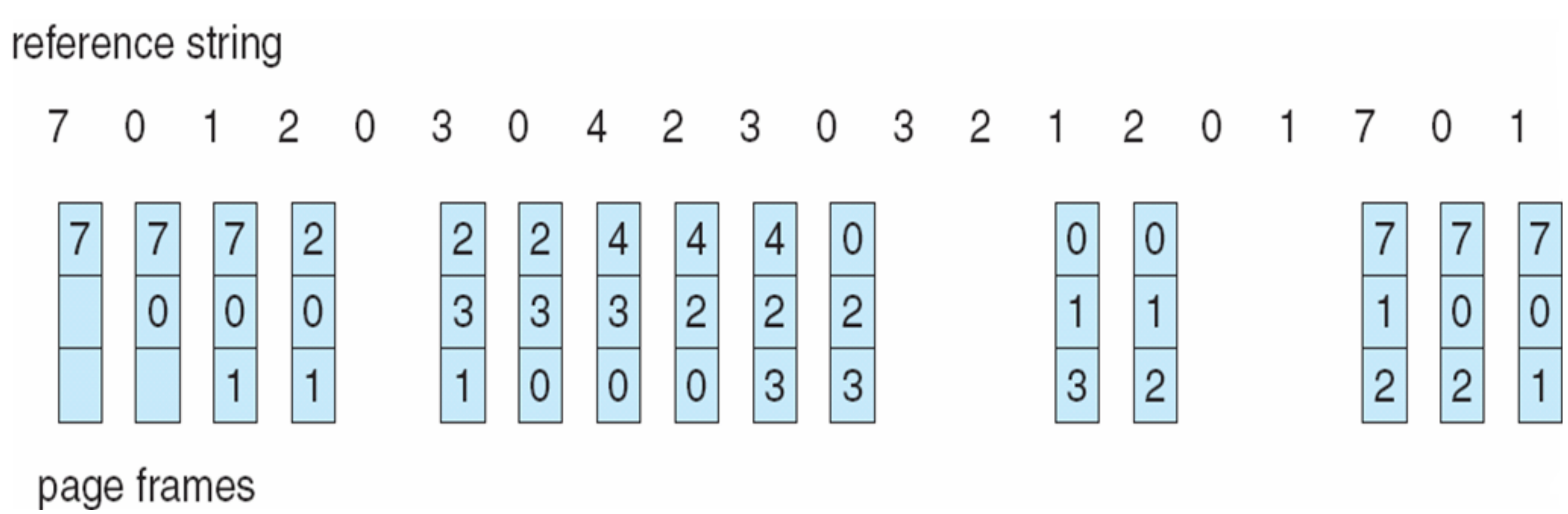
- Page-replacement algorithm should have lowest page-fault rate on both first access and re-access
  - **FIFO**, **optimal**, **LRU**, **LFU**, **MFU**…
- To evaluate a page replacement algorithm:
  - run it on a particular string of memory references (reference string)
    - string is just page numbers, not full addresses
  - compute the number of page faults on that string
    - repeated access to the same page does not cause a page fault
  - in all our examples, the reference string is 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Page Faults v.s. Number of Frames

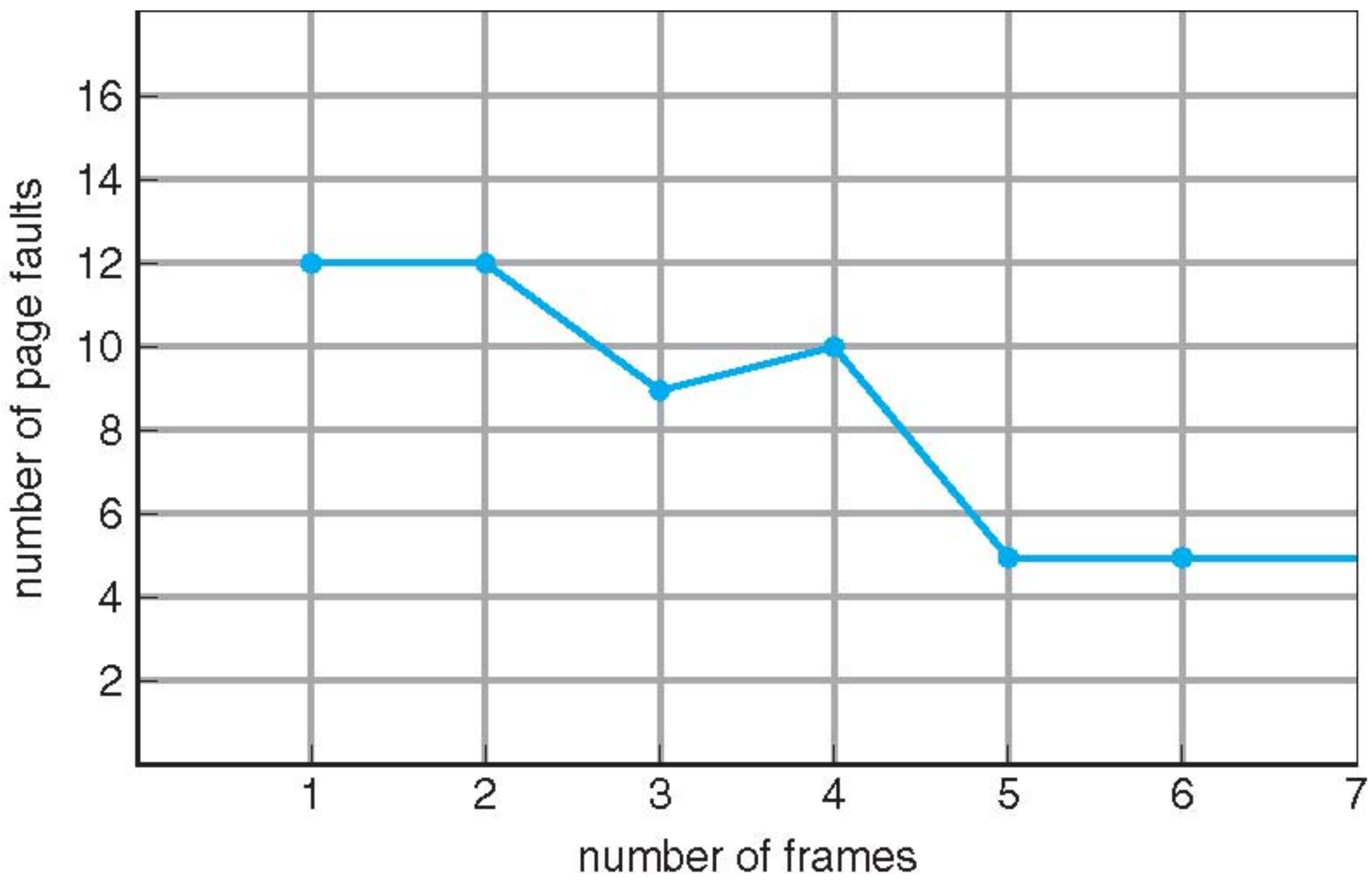


First-In-First-Out (FIFO)

- **FIFO**: replace the first page loaded
  - similar to sliding a window of n in the reference string
  - our reference string will cause 15 page faults with 3 frames
  - how about reference string of 1,2,3,4,1,2,5,1,2,3,4,5 /w 3 or 4 frames?
- For FIFO, adding **more frames** can cause **more page faults**!
  - **Belady's Anomaly**

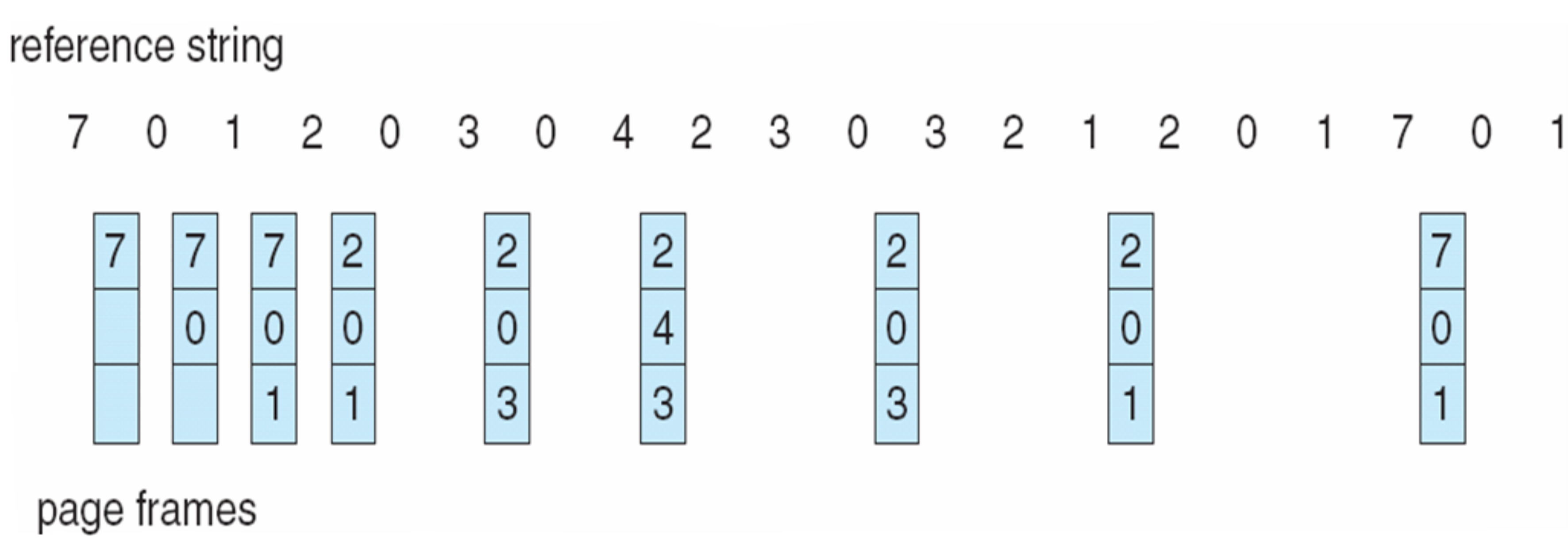


FIFO Illustrating Belady's Anomaly



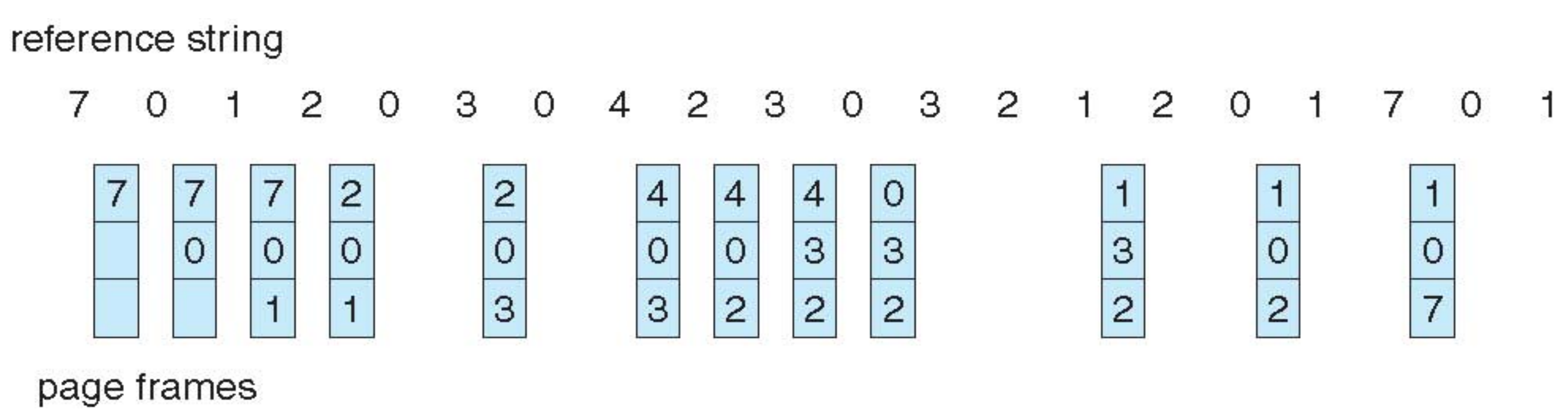
Optimal Algorithm

- **Optimal** : replace page that will not be used for the longest time
  - 9 page fault is optimal for the example on the next slide
- How do you know which page will not be used for the longest time?
  - can’t read the future
  - used for measuring how well your algorithm performs



Least Recently Used (LRU)

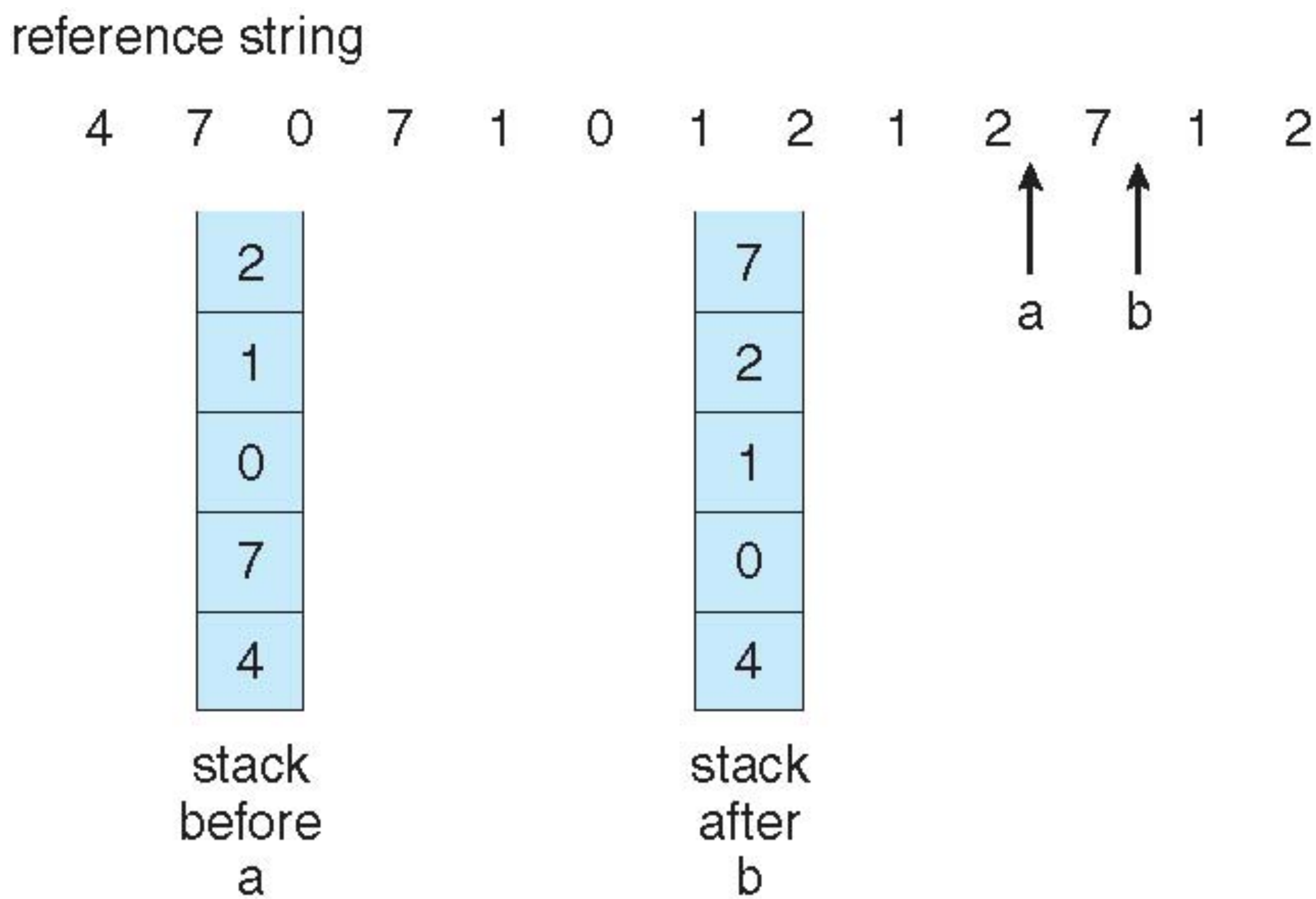
- **LRU** replaces pages that have not been used for the longest time
  - associate time of last use with each page, select pages w/ oldest timestamp
  - generally good algorithm and frequently used
  - 12 faults for our example, better than FIFO but worse than OPT
- LRU and OPT do **NOT** have Belady's Anomaly
- How to implement LRU?
  - **counter-based**
  - **stack-based**



LRU Implementation

- **Counter-based** implementation
  - every page table entry has a counter
  - every time page is referenced, copy the **clock** into the counter
  - when a page needs to be replaced, search for page with smallest counter
    - min-heap can be used
- **Stack-based** implementation
  - keep a stack of page numbers (in double linked list)
  - when a page is referenced, move it to the top of the stack
  - each update is more expensive, but no need to search for replacement

Stack-based LRU



LRU Implementation

- Counter-based and stack-based LRU have high performance overhead
- LRU approximation with a **reference bit**
  - associate with each page a reference bit, initially set to 0
  - when page is referenced, set the bit to 1 (done by the hardware)
  - replace any page with reference bit = 0 (if one exists)

Counting-based Page Replacement

- Keep the number of references made to each page
- **LFU** replaces page with the smallest counter
- **MFU** replaces page with the largest counter
  - based on the argument that page with the smallest count was probably just brought in and has yet to be used
- LFU and MFU are not common

Thrashing

- If a process doesn't have "enough" pages, page-fault rate may be high
  - page fault to get page, replace some existing frame
  - but quickly need replaced frame back
  - this leads to:

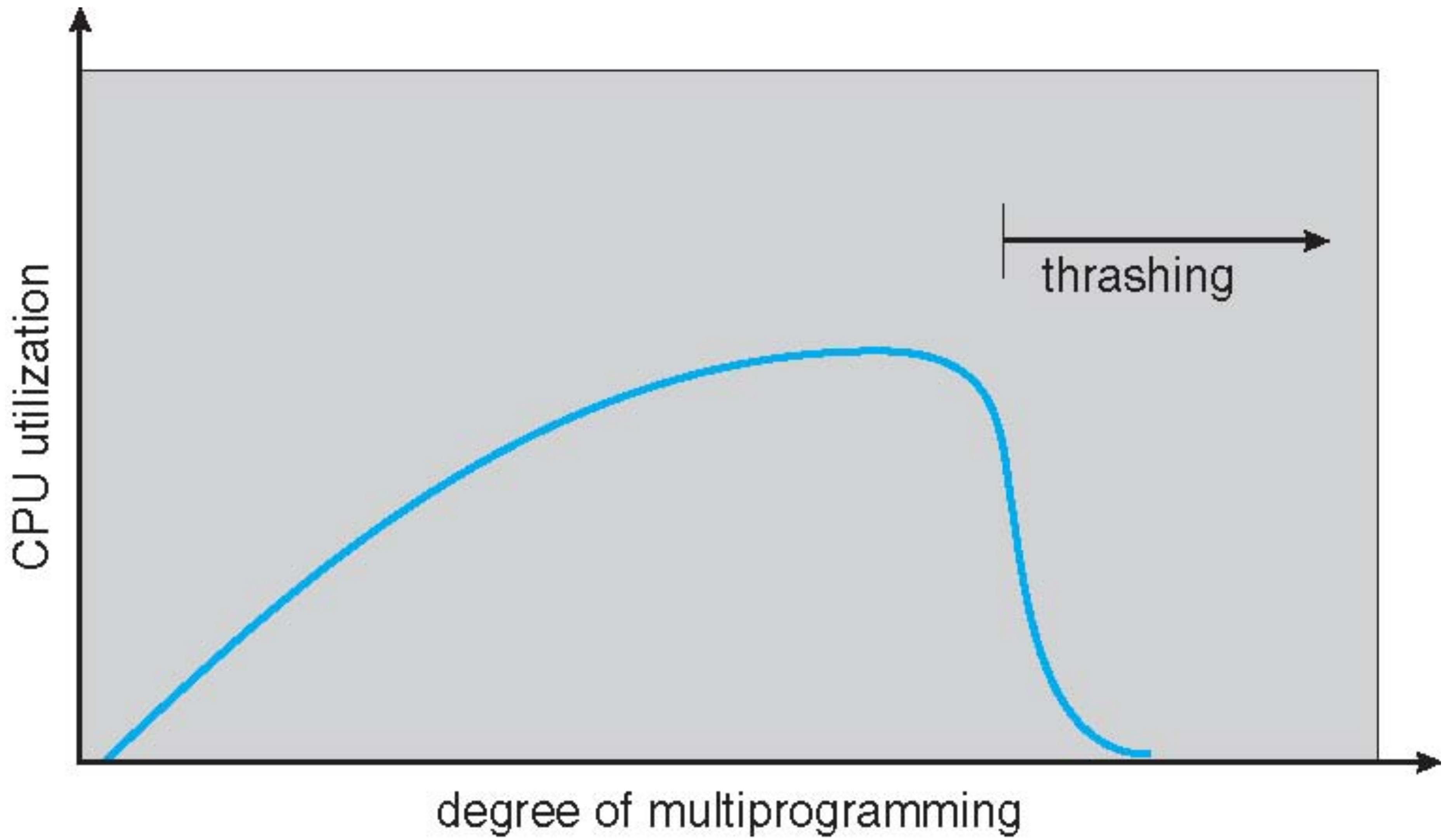
    low CPU utilization ➠

    kernel thinks it needs to increase the degree of

    multiprogramming to maximize CPU utilization ➠

    another process added to the system
- **Thrashing**: a process is busy swapping pages in and out

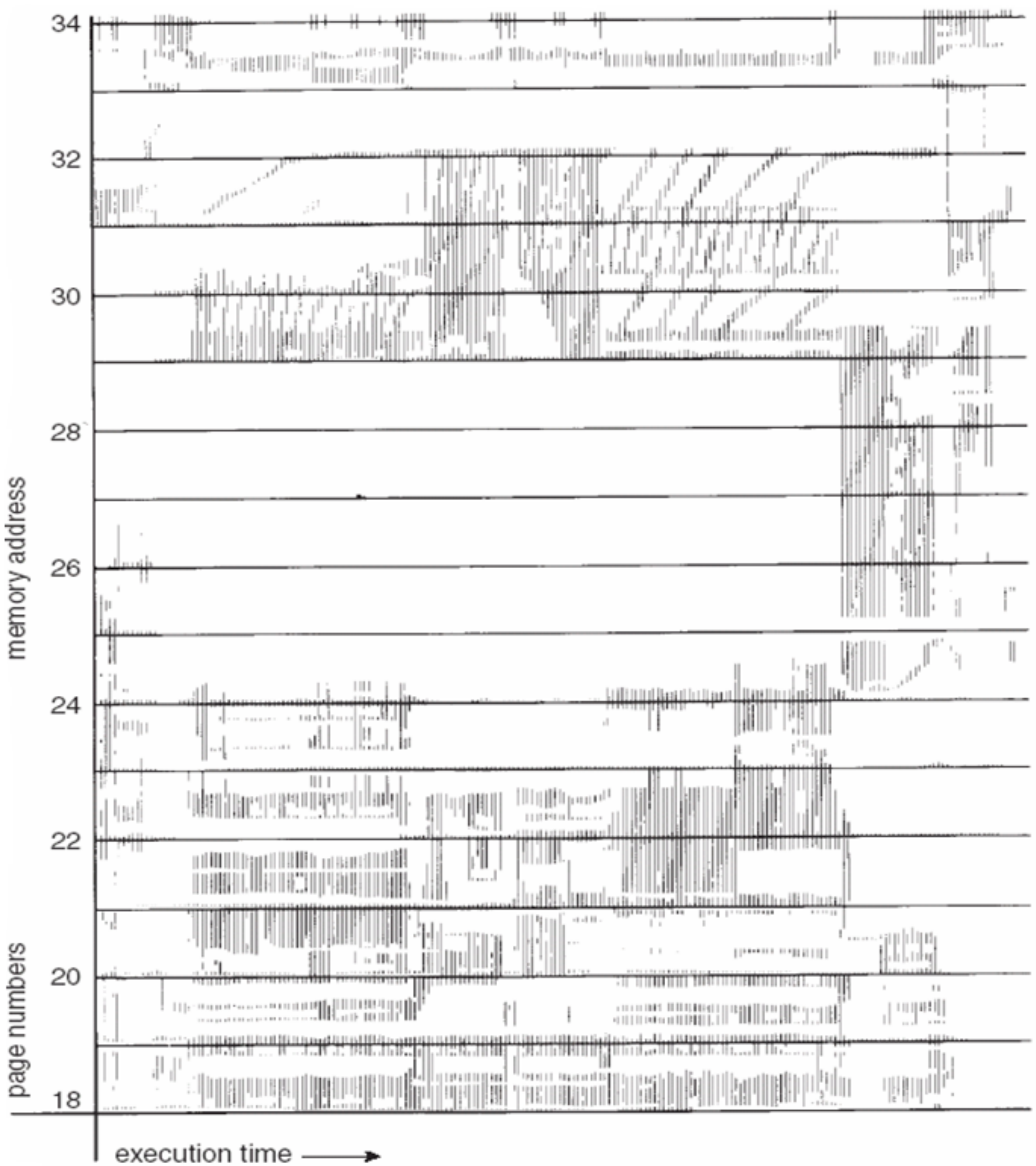
Thrashing



Demand Paging and Thrashing

- Why does demand paging work?
  - process memory access has **high locality**
  - process migrates from one locality to another, localities may overlap
- Why does thrashing occur?
  - total size of locality > total memory size

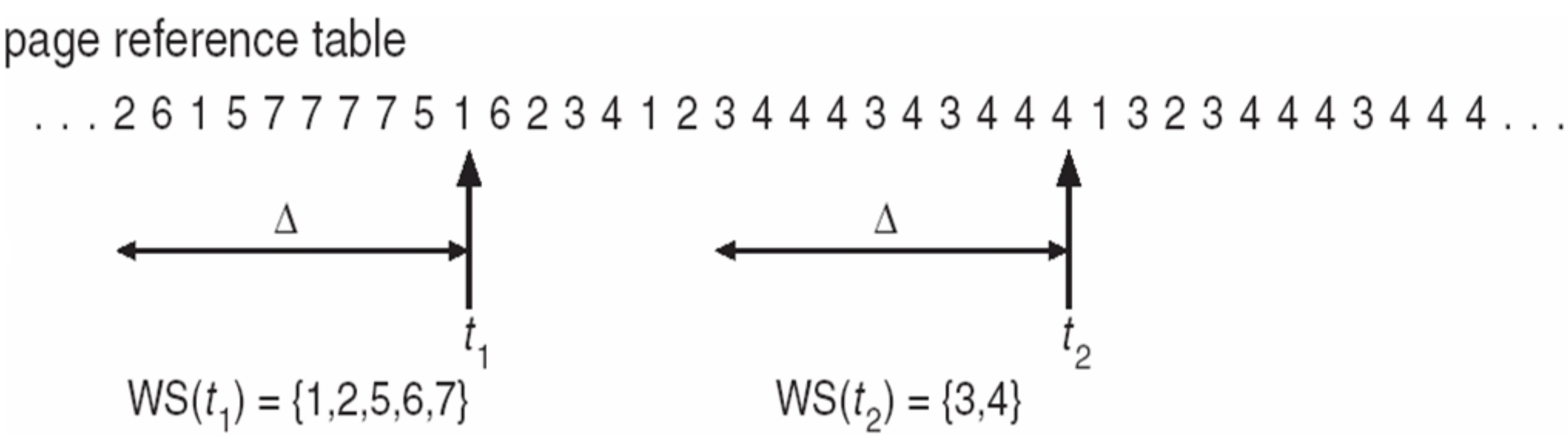
Memory Access Locality



Working-Set Model

- **Working-set window**(Δ): a fixed number of page references
  - if Δ too small ➟ will not encompass entire locality
  - if Δ too large ➟ will encompass several localities
  - if Δ = ∞ ➟ will encompass entire program
- **Working set** of process p_i (WSSi): total number of pages referenced in the most recent Δ (varies in time)
- **Total working sets**: $D = \sum WSS_i$
  - approximation of total locality
  - if D > m ➟ possibility of thrashing
  - to avoid thrashing: if D > m, suspend or swap out some processes

Working-Set Model



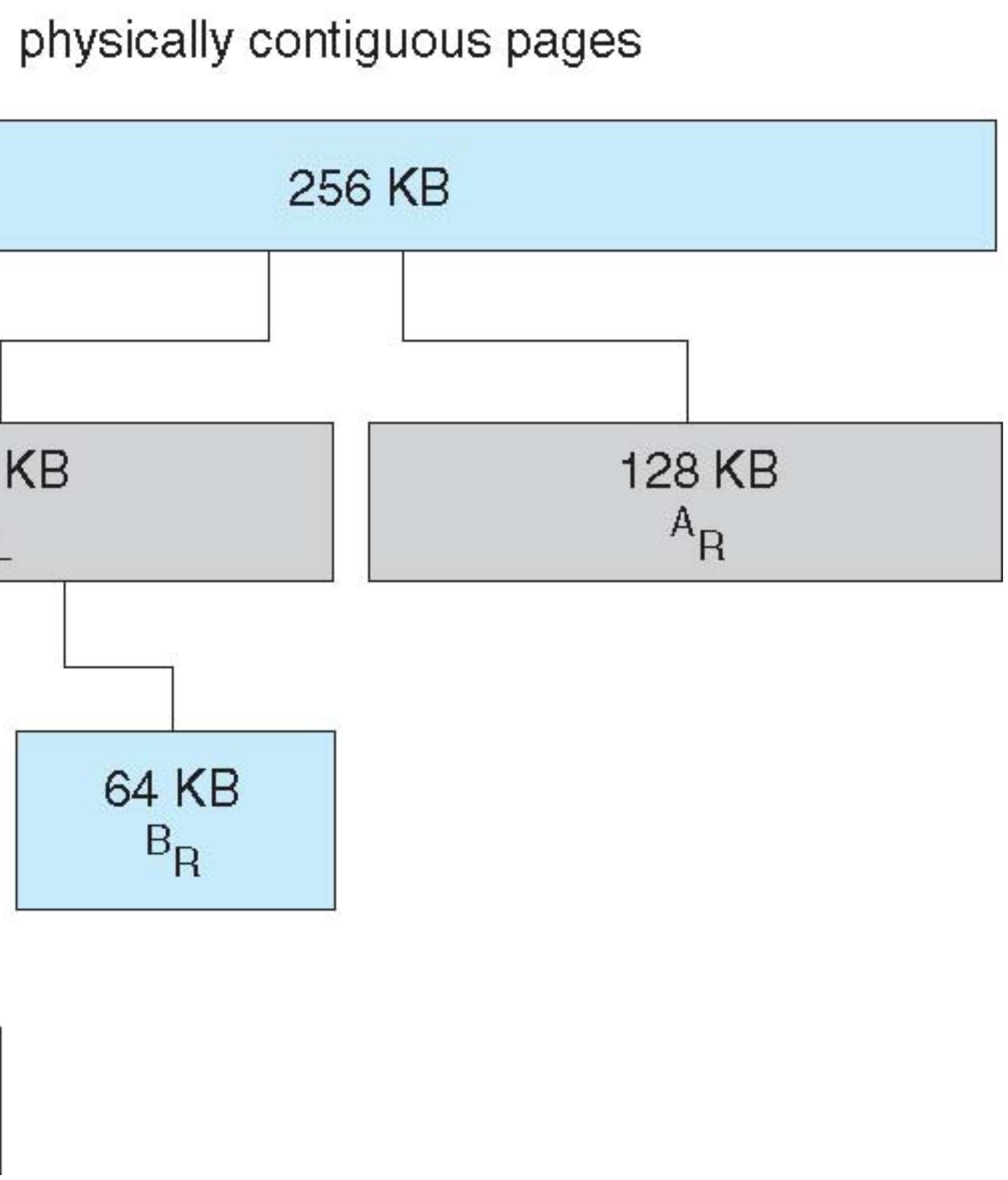
Kernel Memory Allocation

- Kernel memory allocation is treated differently from user memory
  - for kernel data structures, and for user applications
  - key to the OS performance: utilization, fairness, performance,…
- Kernel memory is often allocated from a free-memory pool
  - kernel requests memory for structures of varying sizes
  - some kernel memory needs to be **physically contiguous**
    - e.g., for device I/O

Buddy System

- Memory allocated using power-of-2 allocator
  - memory is allocated in units of the size of **power of 2**
    - round up a request to the closest allocation unit
    - split the unit into two "**buddies**" until a proper sized chunk is available
  - e.g., assume only 256KB chunk is available, kernel requests 21KB
    - split it into A_l and A_r of 128KB each
    - further split an 128KB chunk into B_l and B_r of 64KB
    - again, split a 64KB chunk into C_l and C_r of 32KB each
    - give one chunk for the request
- advantage: it can quickly coalesce unused chunks into larger chunk
- disadvantage: **internal fragmentation**

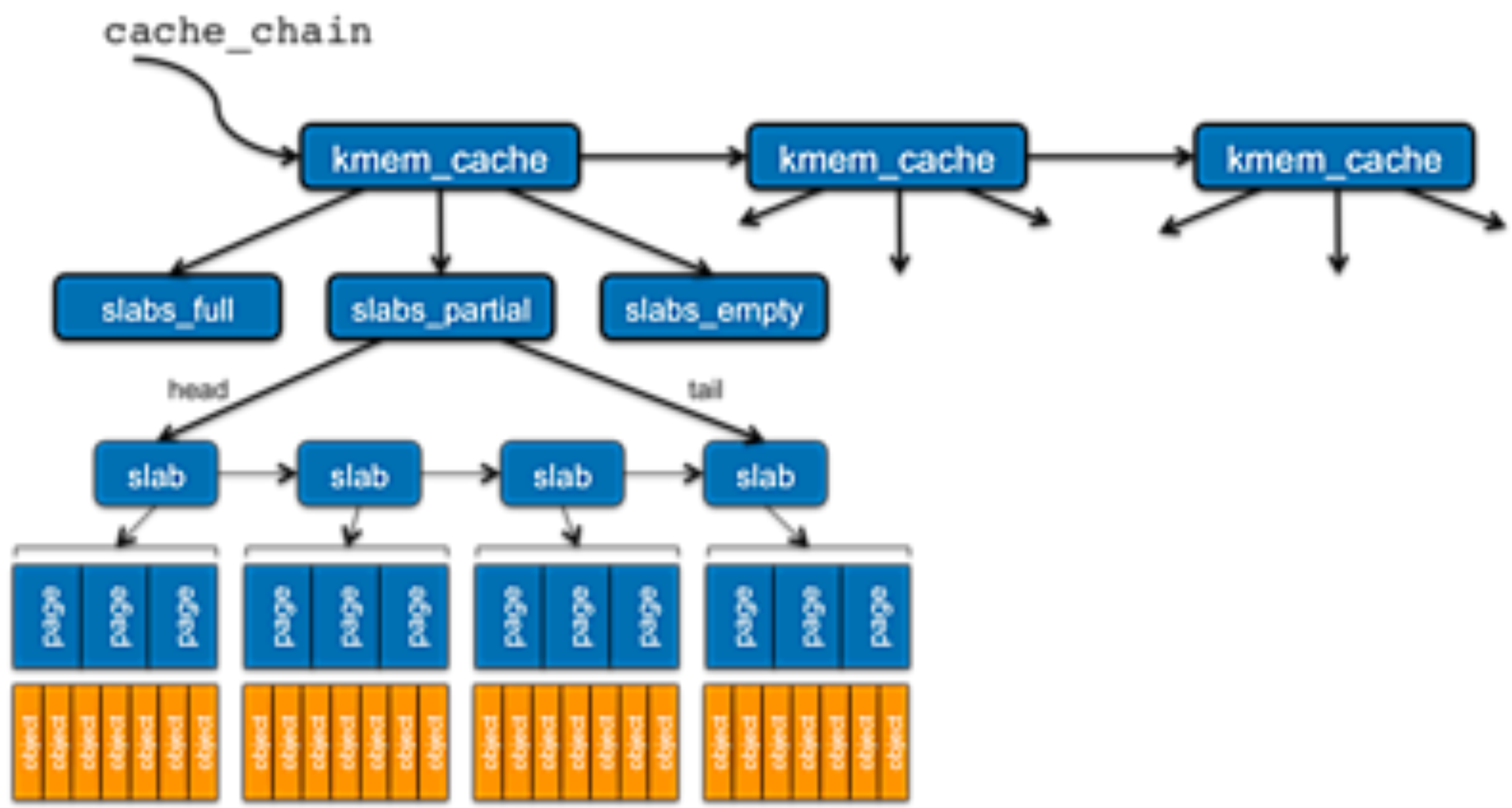
Buddy System Allocator



Slab Allocator

- Slab allocator is a **cache of objects**
  - a cache in a slab allocator consists of one or more slabs
  - a Slab contains one or more pages, divided into equal-sized objects
  - kernel uses one cache for each unique kernel data structure
    - when cache created, allocate a slab, divided the slab into free objects
    - objects for the data structure is allocated from free objects in the slab
    - if a slab is full of used objects, next object comes from an empty/new slab
- Benefits: **no fragmentation** and fast memory allocation
  - some of the object fields may be reusable; no need to initialize again

Slab Allocation (Linux)



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 to reduce **TLB pressure**
  - it may increase fragmentation as not all applications require large page sizes
  - multiple page sizes allow applications that require larger page sizes to use them without an increase in fragmentation

Other Issues: Program Structure

- Program structure can affect page faults
  - 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 (assume TLB only has one entry)

  - Program 2:

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

128 page faults

Operating System Examples

- Windows XP
- Solaris

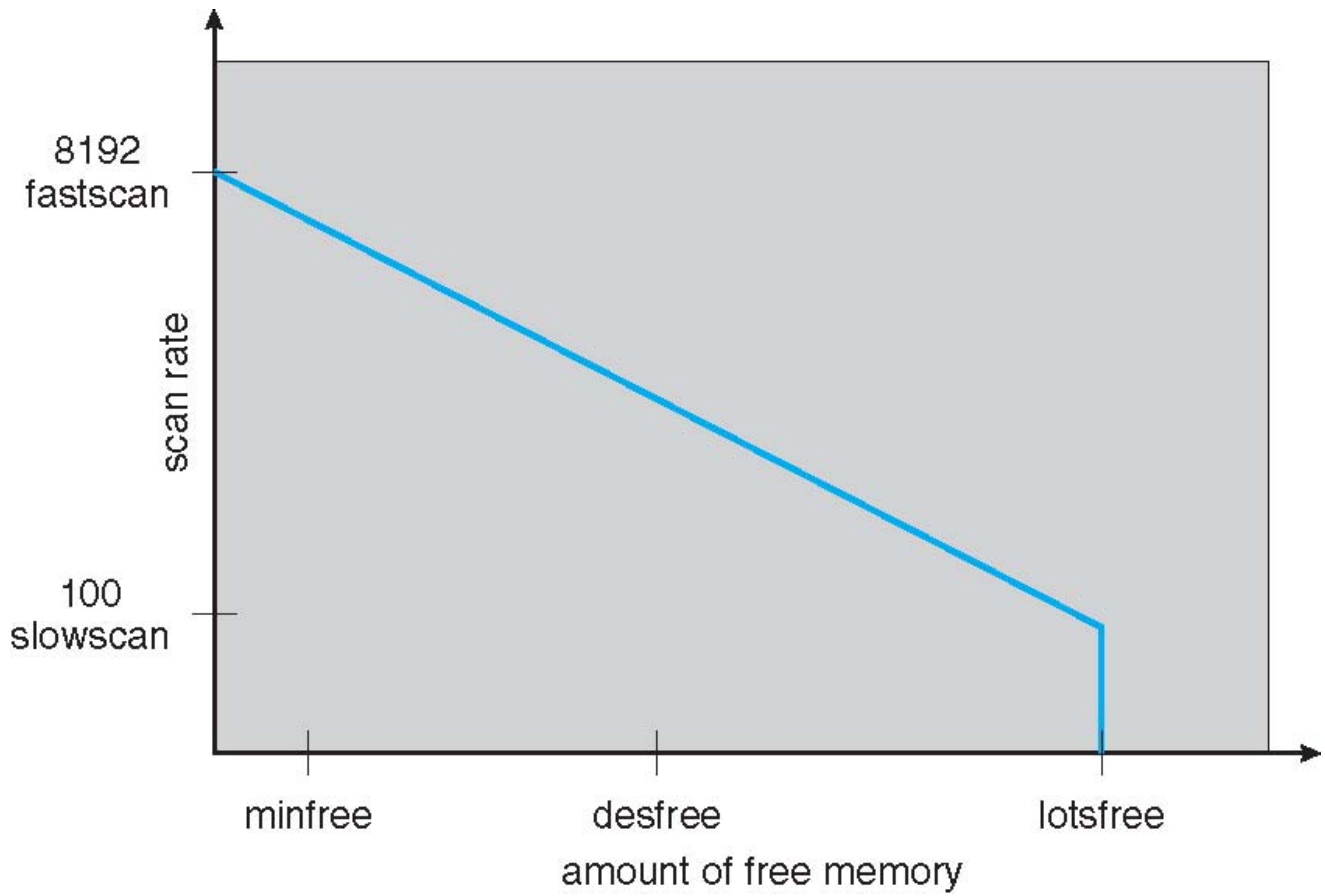
Windows XP

- Uses demand paging with clustering
  - clustering brings in pages surrounding the faulting page
- Processes are assigned working set minimum and set maximum
  - *wsmin*: minimum number of pages the process is guaranteed to have
  - *wsmax*: a process may be assigned as many pages up to its *wsmax*
- When the amount of free memory in the system falls below a threshold:
  - automatic working set trimming to restore the amount of free memory
  - it removes pages from processes that have more pages than the *wsmin*

Solaris

- Three thresholds to determine paging and swapping
  - *lotsfree*: threshold (amount of free memory) to begin paging
  - *desfree*: threshold parameter to increasing paging
  - *minfree*: threshold parameter to being swapping
- Pageout scans pages, looking for pages to replace
  - less free memory more frequent calls to page out
  - two scan rate: slow scan and fast scan
  - priority paging gives priority to process code pages

Solaris 2 Page Scanner



End of Chapter 9