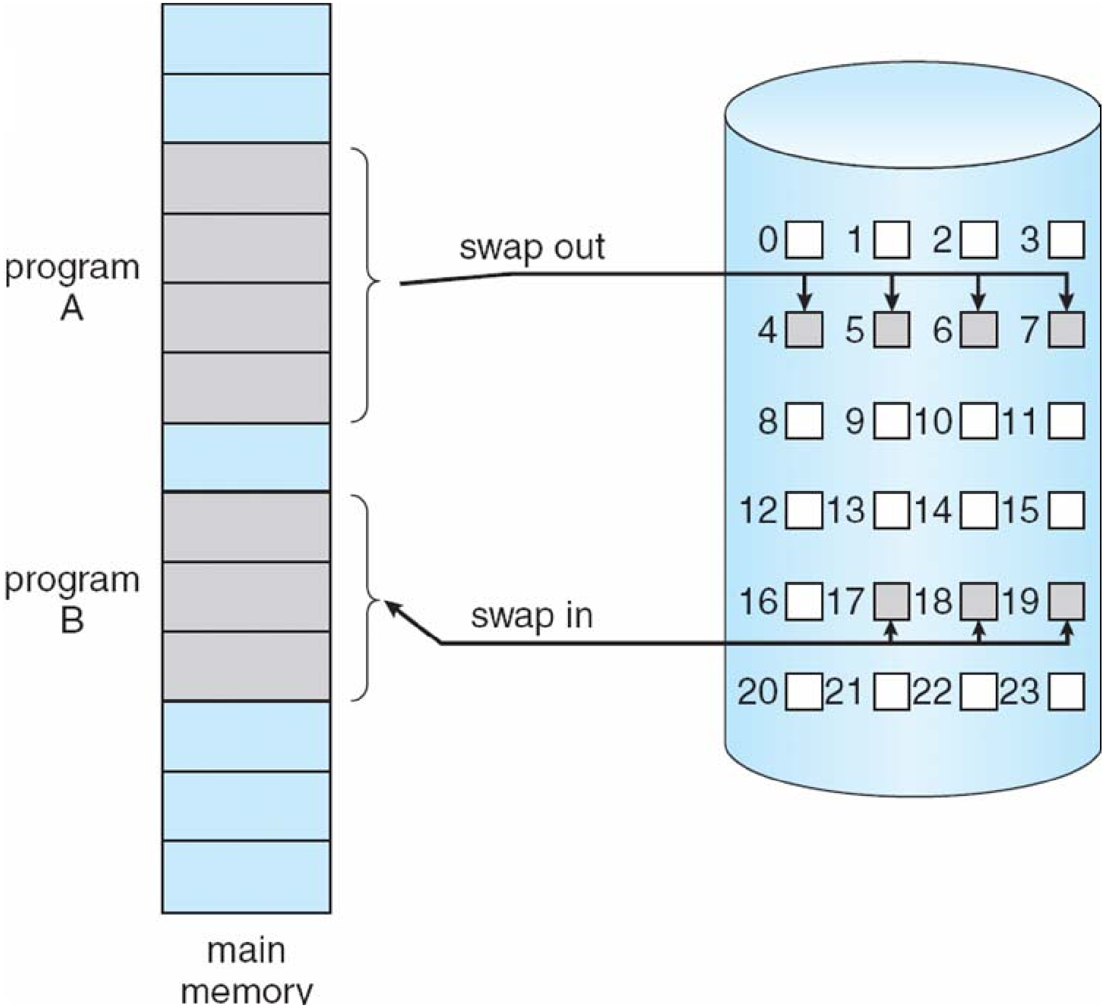
Chapter 4.9: Virtual-Memory Management

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations

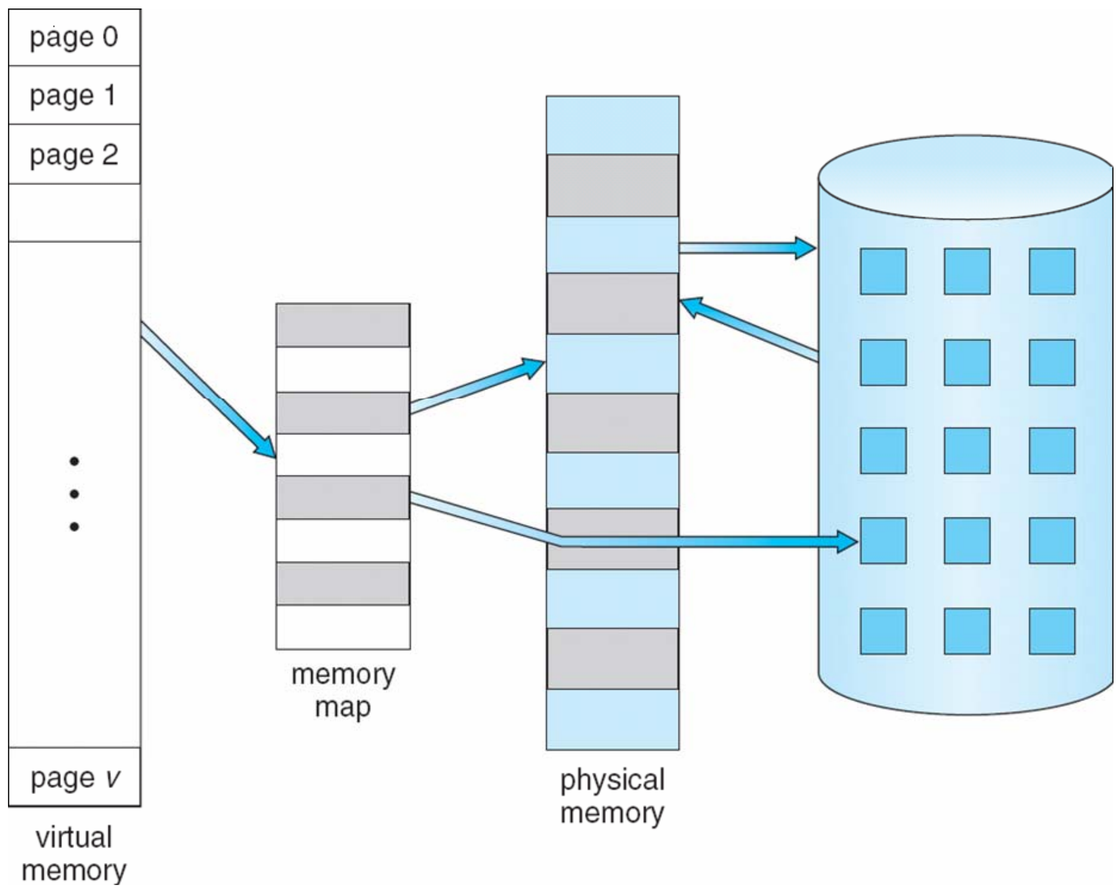
Background

- **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

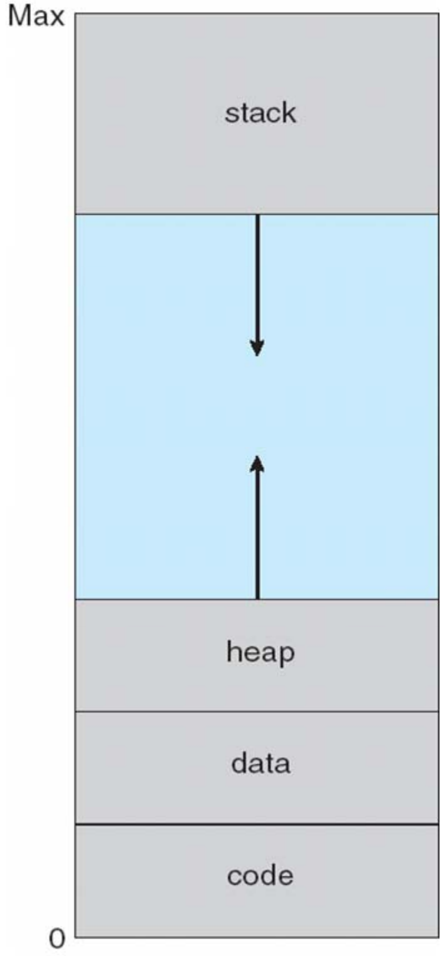
Transfer of a Paged Memory to Contiguous Disk Space



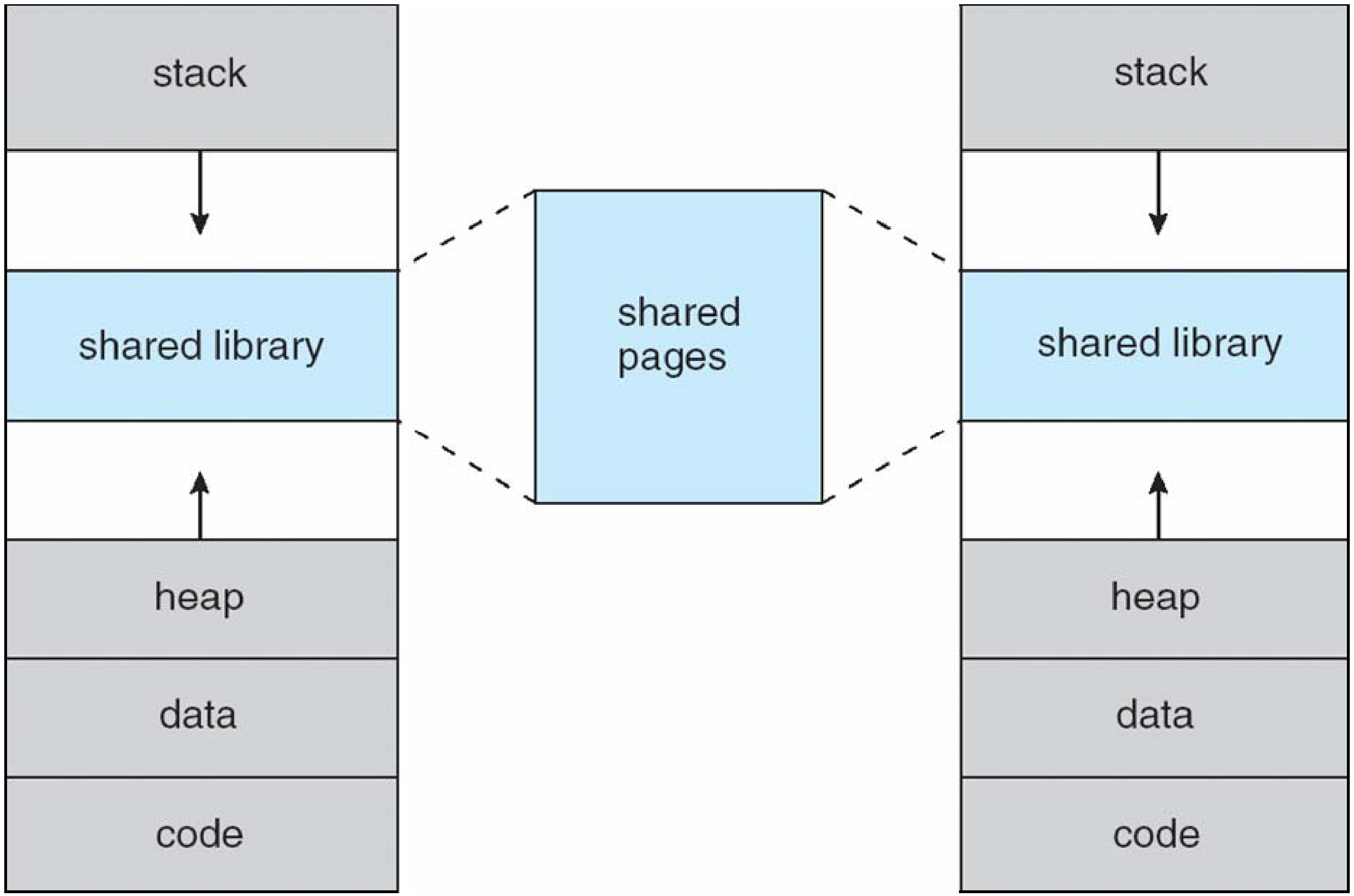
Virtual Memory That is Larger Than Physical Memory



Virtual-address Space



Shared Library Using Virtual Memory



Page Fetch Policy

- Demand paging transfers a page to RAM if a reference to that page has raised a page fault
  - CON: “Many” initial page faults when a task starts
  - PRO: You only transfer what you really need

- Pre-Paging transfers more pages from disk to RAM additionally to the demanded page
  - PRO: improves disk I/O throughput by reading chunks
  - CON: Pre-paging is highly speculative
    - wastes I/O bandwidth if page will never be used
    - can destroy the working set of another task in case of page stealing

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
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

Valid-Invalid Bit (Present Bit)

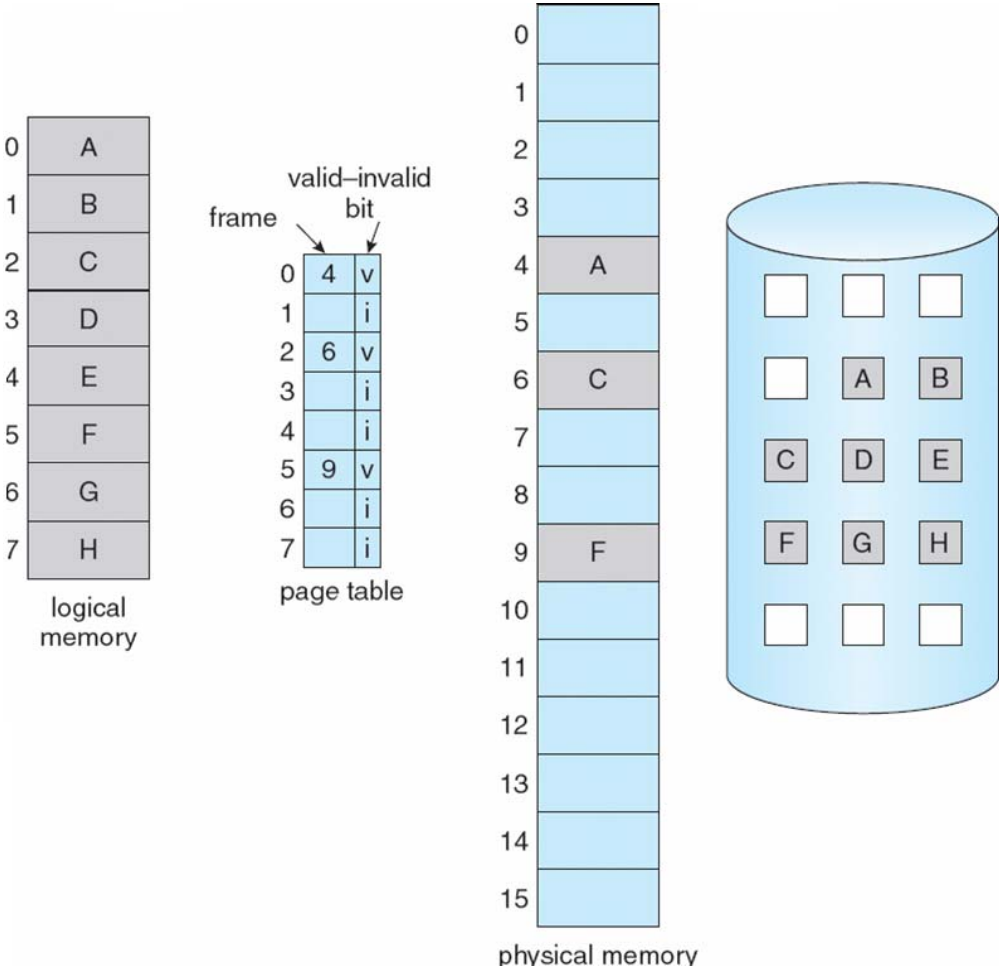
- With each page table entry a valid–invalid bit is associated
(**v** ⇒ in-memory, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

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

Page Table When Some Pages Are Not in Main Memory



Page Fault

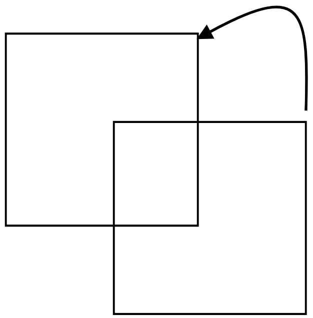
- If there is a reference to a page, first reference to that page will trap to operating system:

  page fault

1. Operating system looks at another table to decide:
   - Invalid reference $\Rightarrow$ 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

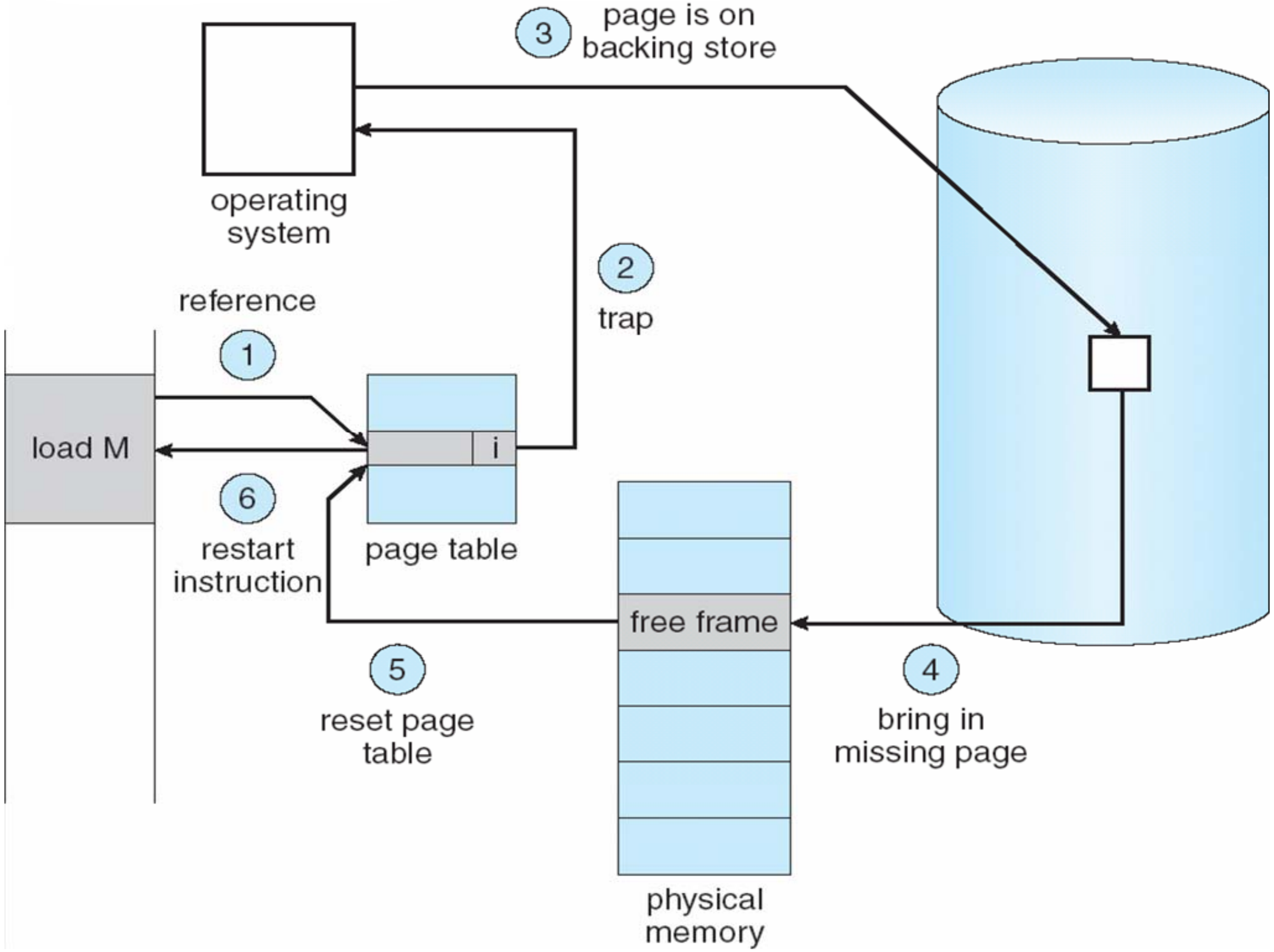
Page Fault (Cont.)

- Problems with instruction restart instruction
  - block move



  - auto increment/decrement multiple locations
- Solutions for consistent restart
  - Touch all relevant pages before operation starts
  - Keep all modified data in registers until page faults can't take place

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)

$$\begin{aligned} EAT = (1 - p) \times \text{memory access} \\ + p\,(\text{page fault overhead} \\ + \text{page fault service time} \\ + \text{restart overhead} \\ ) \end{aligned}$$

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!!

Benefits of Paged Virtual Memory

- Paged virtual memory allows other benefits during process creation:

  - Copy-on-Write

  - Memory-Mapped Files (later)

Copy-on-Write

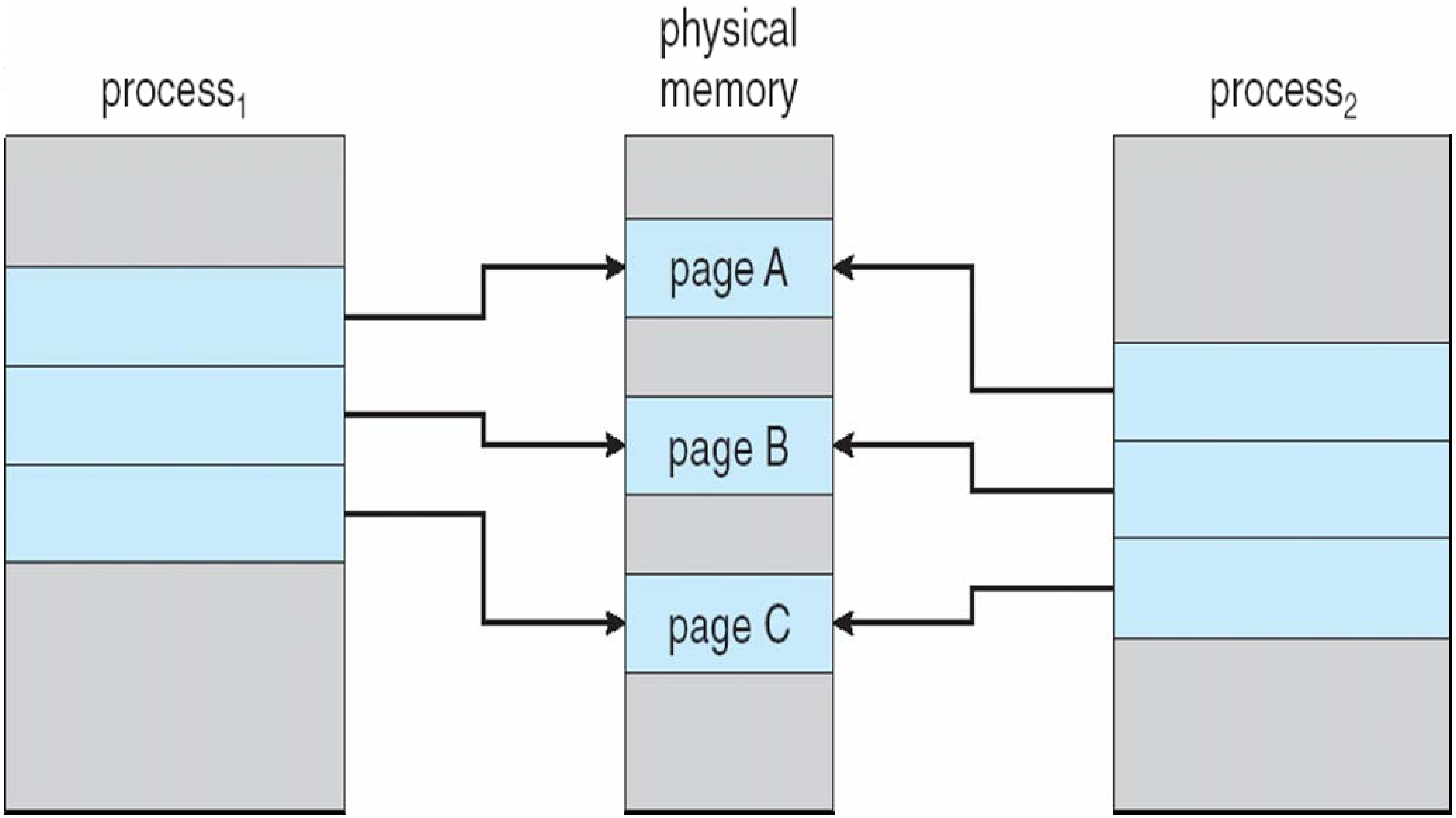
- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory

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

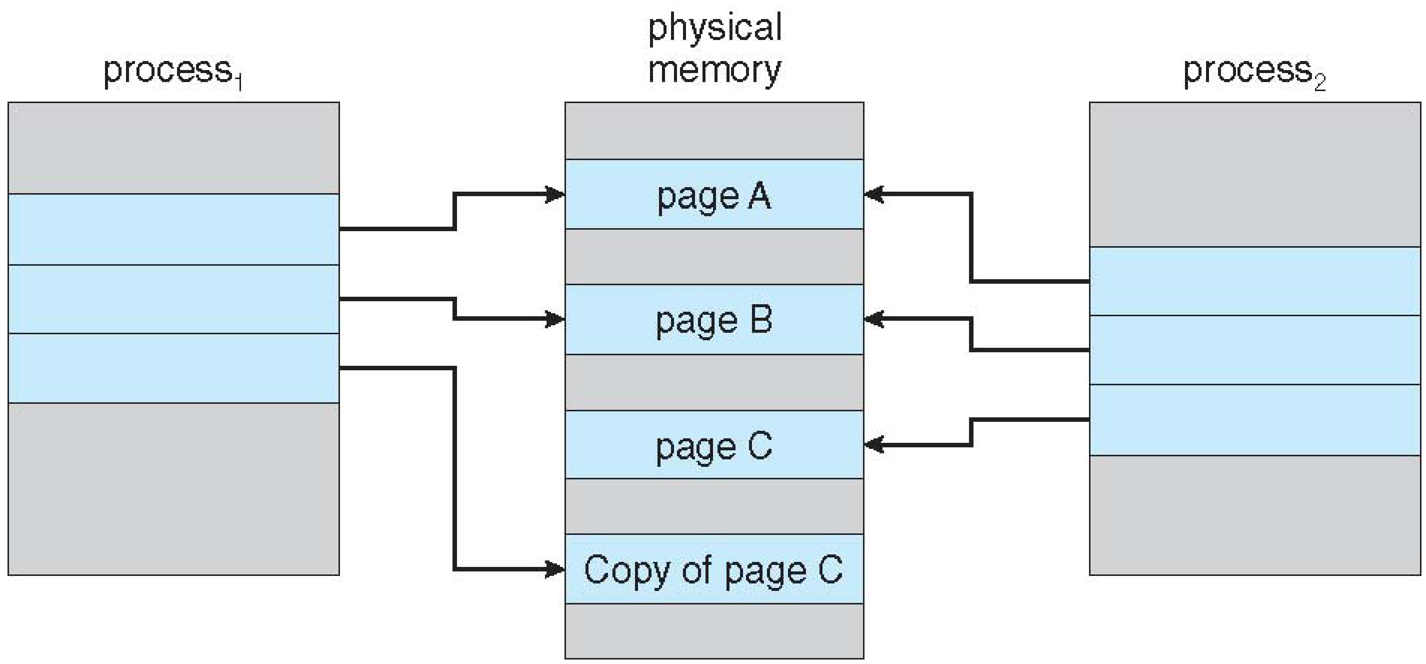
- COW allows more efficient process creation as only modified pages are copied

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

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



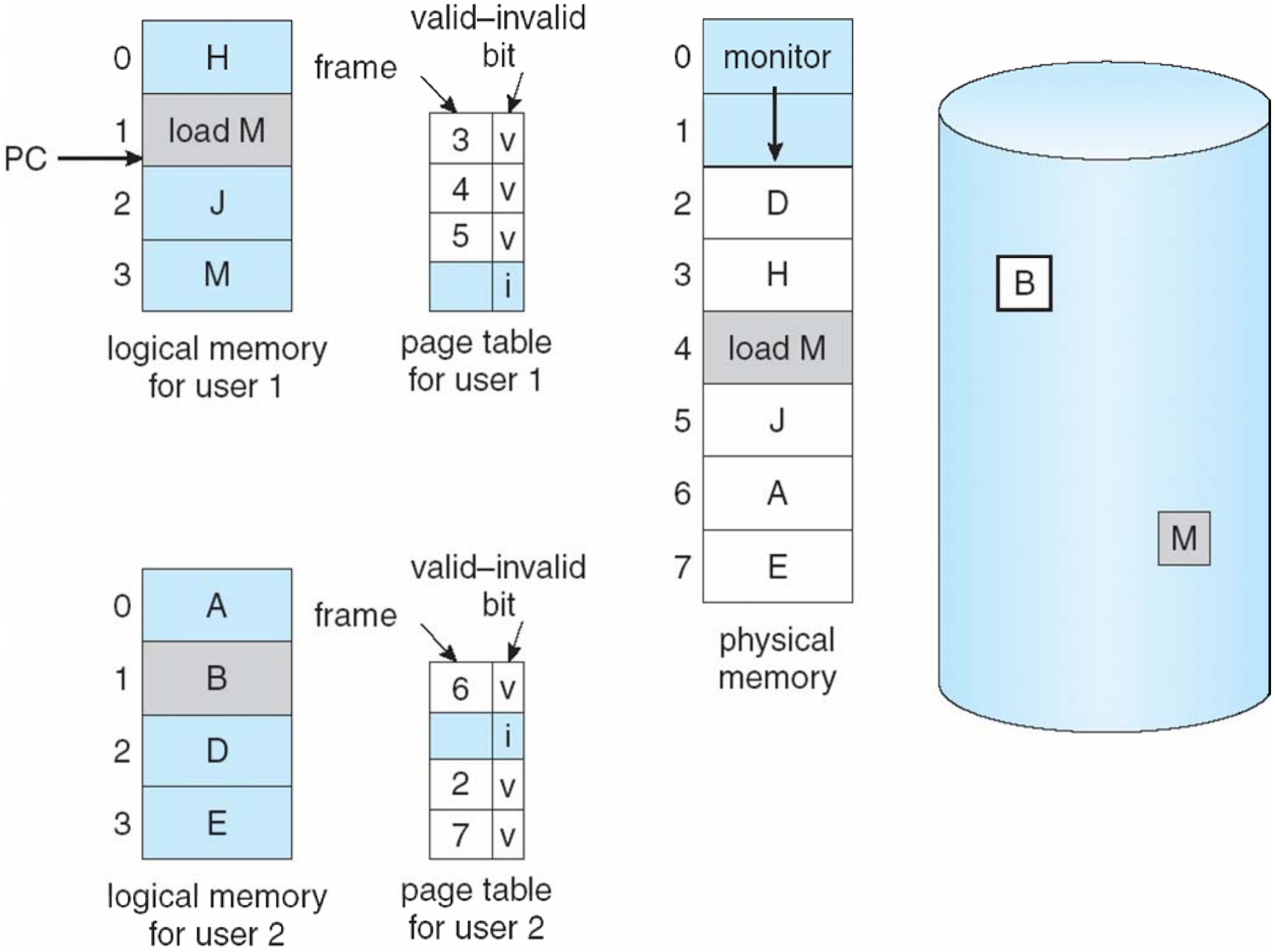
Page Replacement

- Page replacement – find the most fitting page in memory, but not really in use

➢ page it out
  - Algorithm (low administrative overhead)
  - Performance – want an algorithm which will result in minimum number of page faults
  - Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk
  - Same page may be brought into memory several times

➢ Large virtual memory can be provided on a smaller physical memory

Need For Page Replacement



Basic Page Replacement

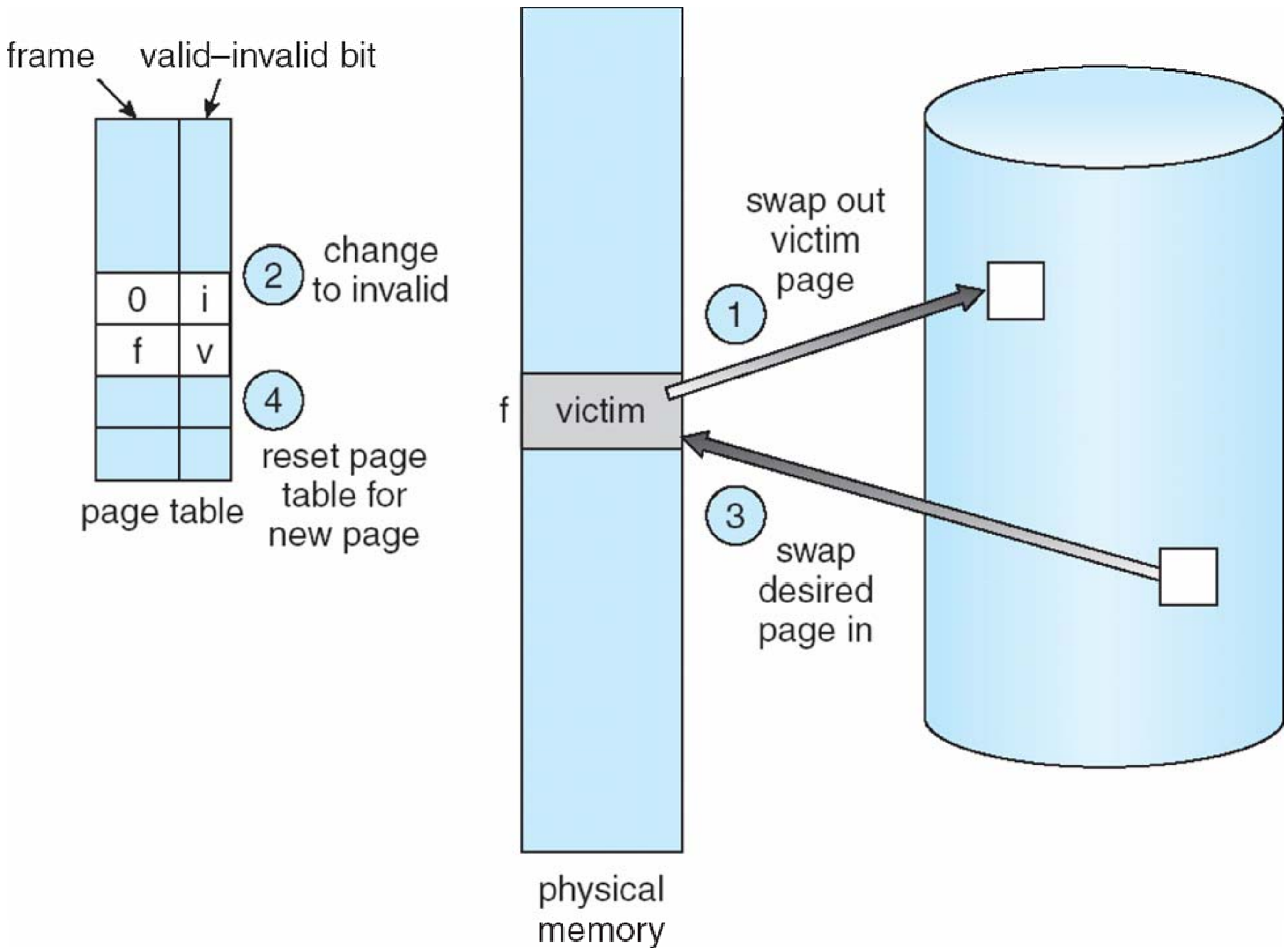
1. Find the location of the desired page on disk

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



Replacement Policy

- Not all page frames in memory can be replaced
  - Some pages are pinned to specific page frames:
    - Most of the kernel is resident, i.e. pinned
    - some DMA can only access physical addresses, i.e. their buffers must be pinned, too (I/O Interlock)
    - A real-time task might have to pin some/all of its pages (otherwise no one can guarantee its deadline)
- OS might decide that set of pages considered for next replacement should be:
  - Limited to frames of the task having initiated page fault
    $\Rightarrow$ local page replacement
  - Unlimited, i.e. also frames belonging to other tasks
    $\Rightarrow$ global page replacement

Cleaning Policy

When should we page-out a "dirty" page?

- Demand Cleaning
  - a page is transferred to disk only when its hosting page frame has been selected for replacement by the replacement policy

  $\Rightarrow$ page faulting activity must wait for 2 page transfers (out and in)

- Pre-Cleaning
  - dirty pages are transferred to disk before their page frames are needed

  $\Rightarrow$ transferring large clusters can improve disk throughput, but it makes few sense to transfer pages to disk if most of them will be modified again before they will be replaced

Cleaning Policy

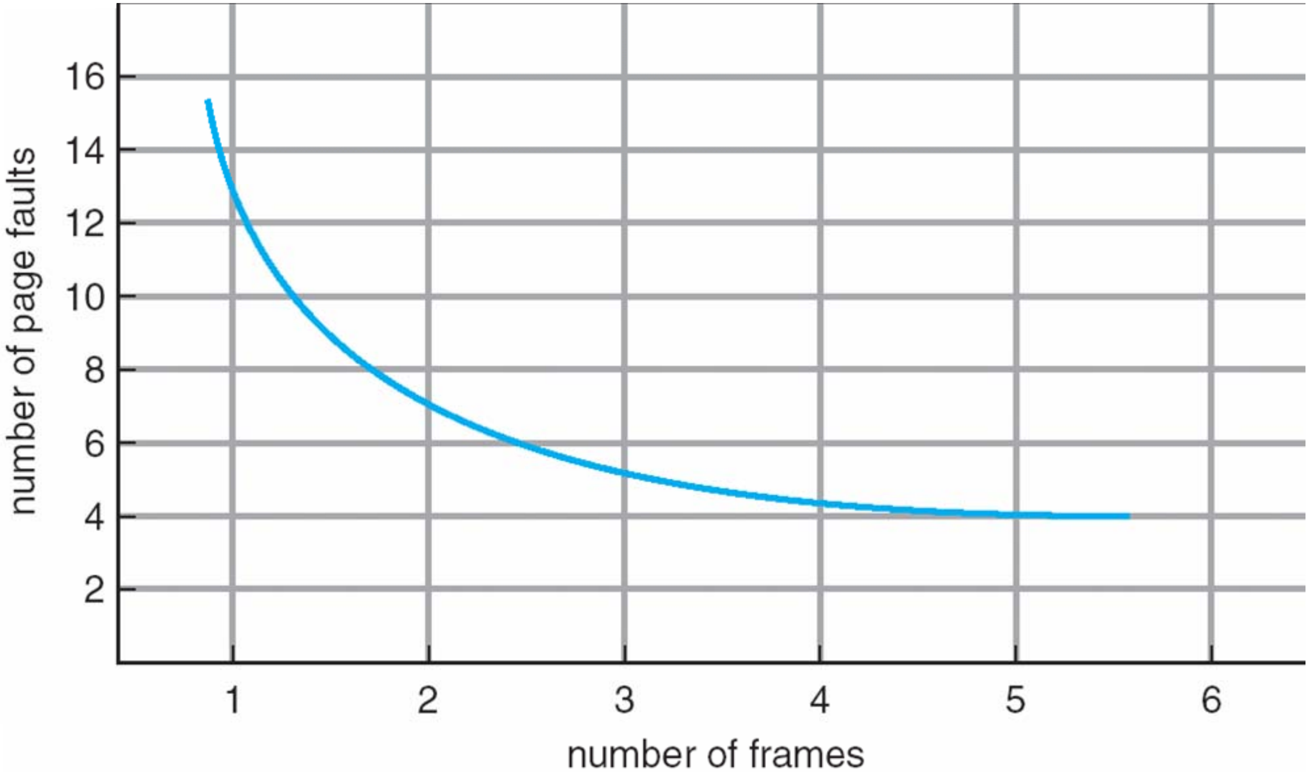
- Good compromise achieved with page buffering
  - Recall that pages chosen for replacement are maintained either in a free (unmodified) list or in a modified list
  - Pages of the modified list can be transferred to disk periodically
  - $\Rightarrow$ A good compromise since:
    - not all dirty pages are transferred to disk, only those that have been chosen for next replacement
    - transferring pages is done in batch (improving disk I/O)

Page Replacement Algorithms

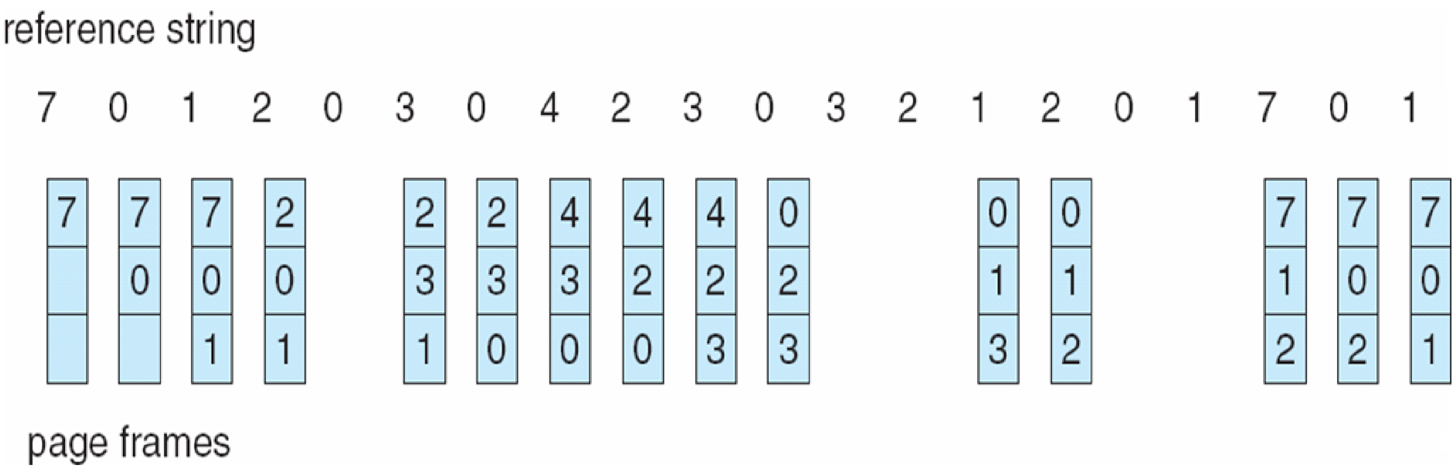
- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
- In all our examples, the reference string is

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

Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm



FIFO Anomaly

- 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)

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

9 page faults

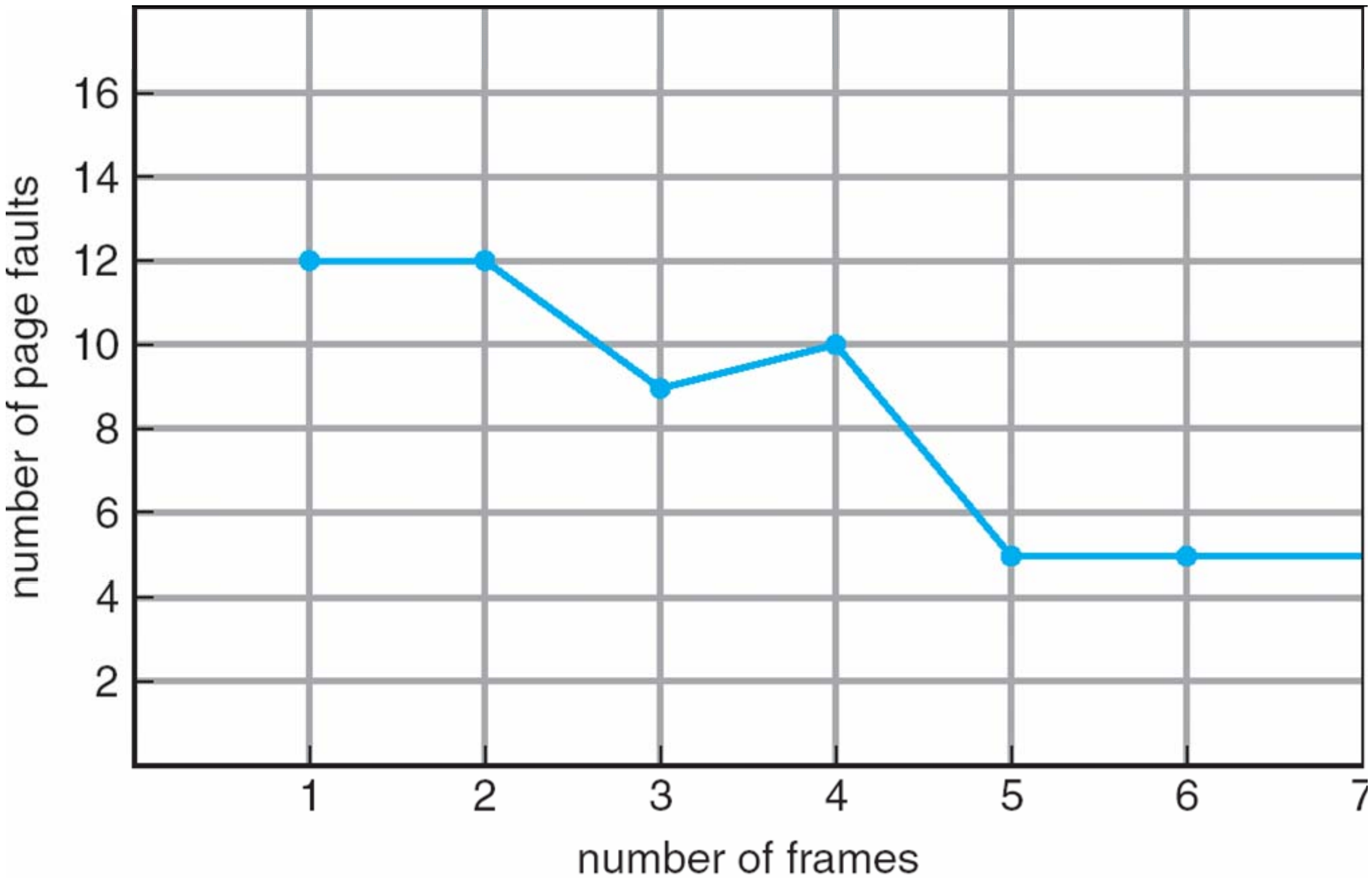
- 4 frames

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

10 page faults

- Belady’s Anomaly: more frames $\Rightarrow$ more page faults

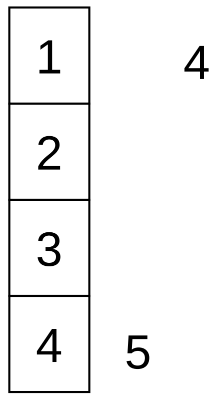
FIFO Illustrating Belady's Anomaly



Optimal Algorithm

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

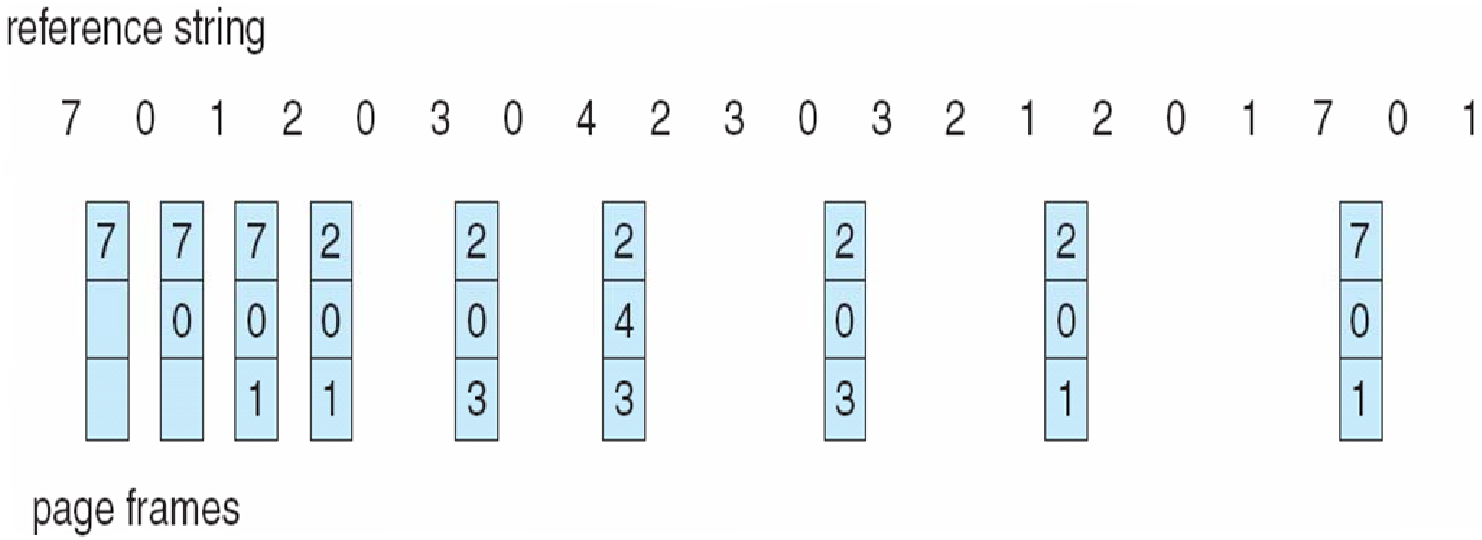
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



6 page faults

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

Optimal Page Replacement



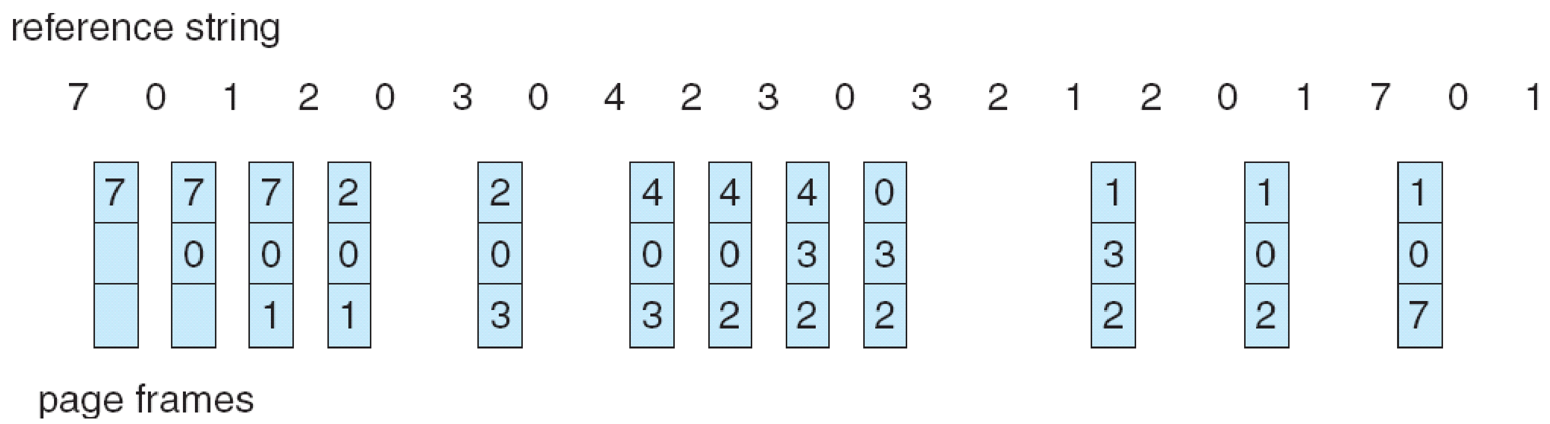
Least Recently Used (LRU) Algorithm

- 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

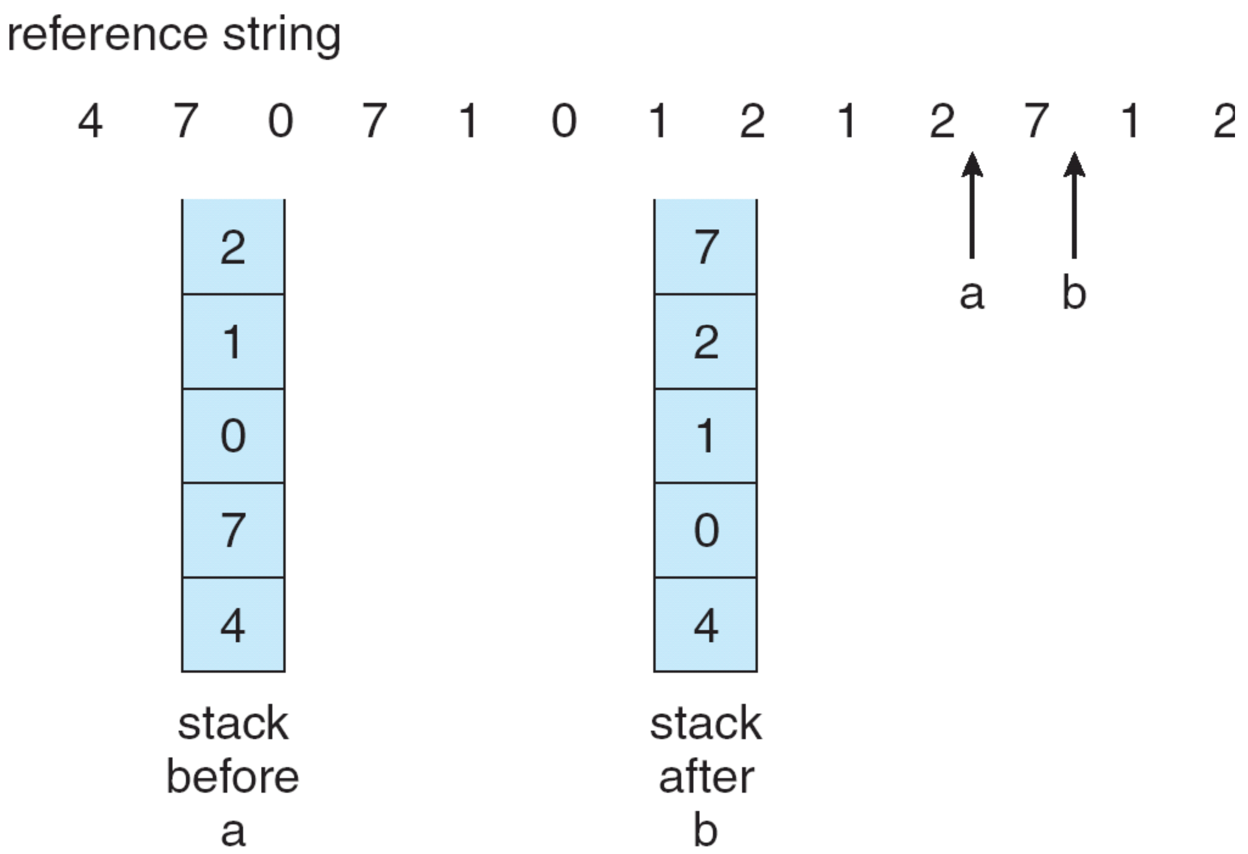
LRU Page Replacement



LRU Stack

- **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

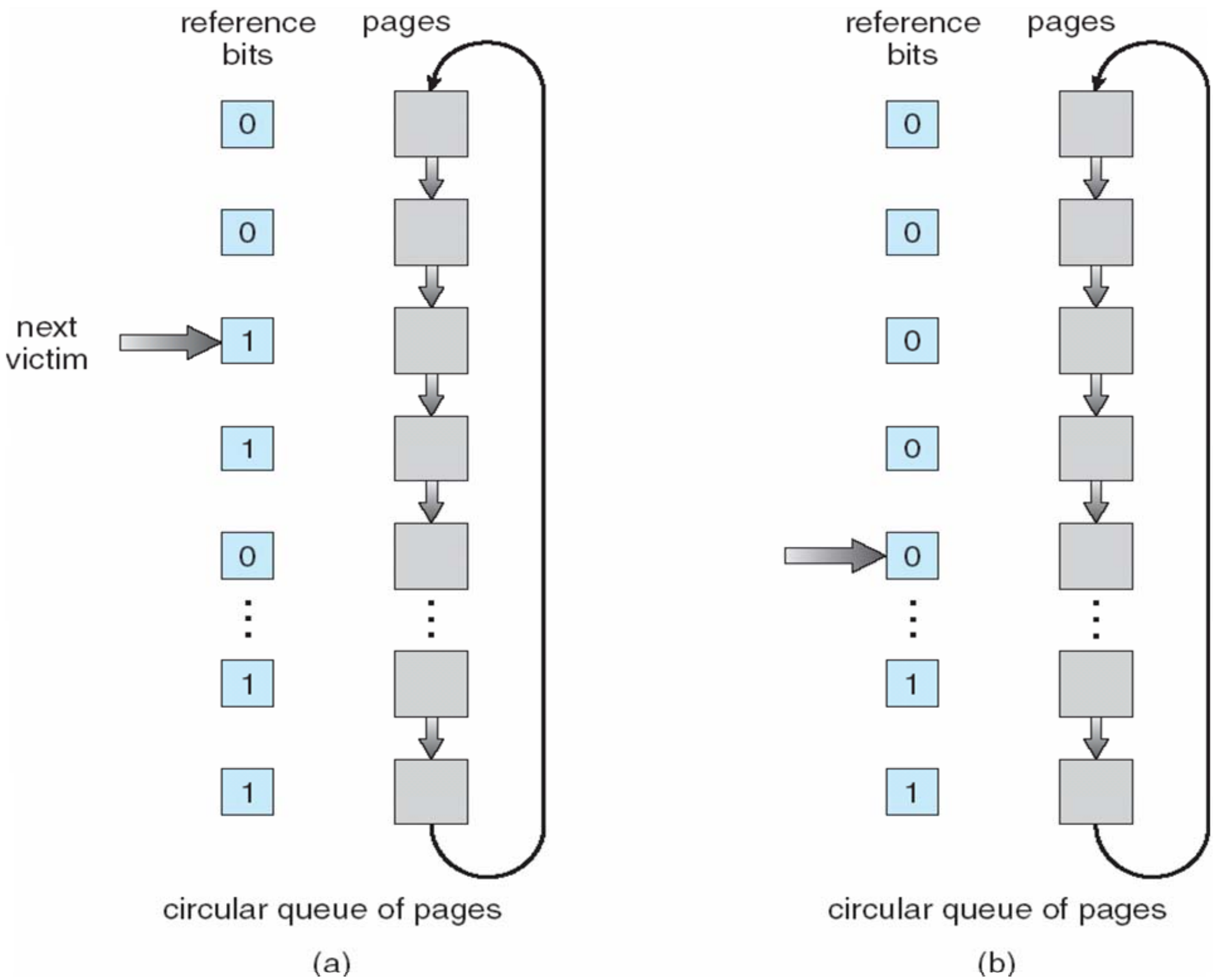
LRU Stack


reference string
4 7 0 7 1 0 1 2 1 2 7 1 2
a
b
2
1
0
7
4
stack before a
7
2
1
0
4
stack after b

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

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
- 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*
- 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

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

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

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

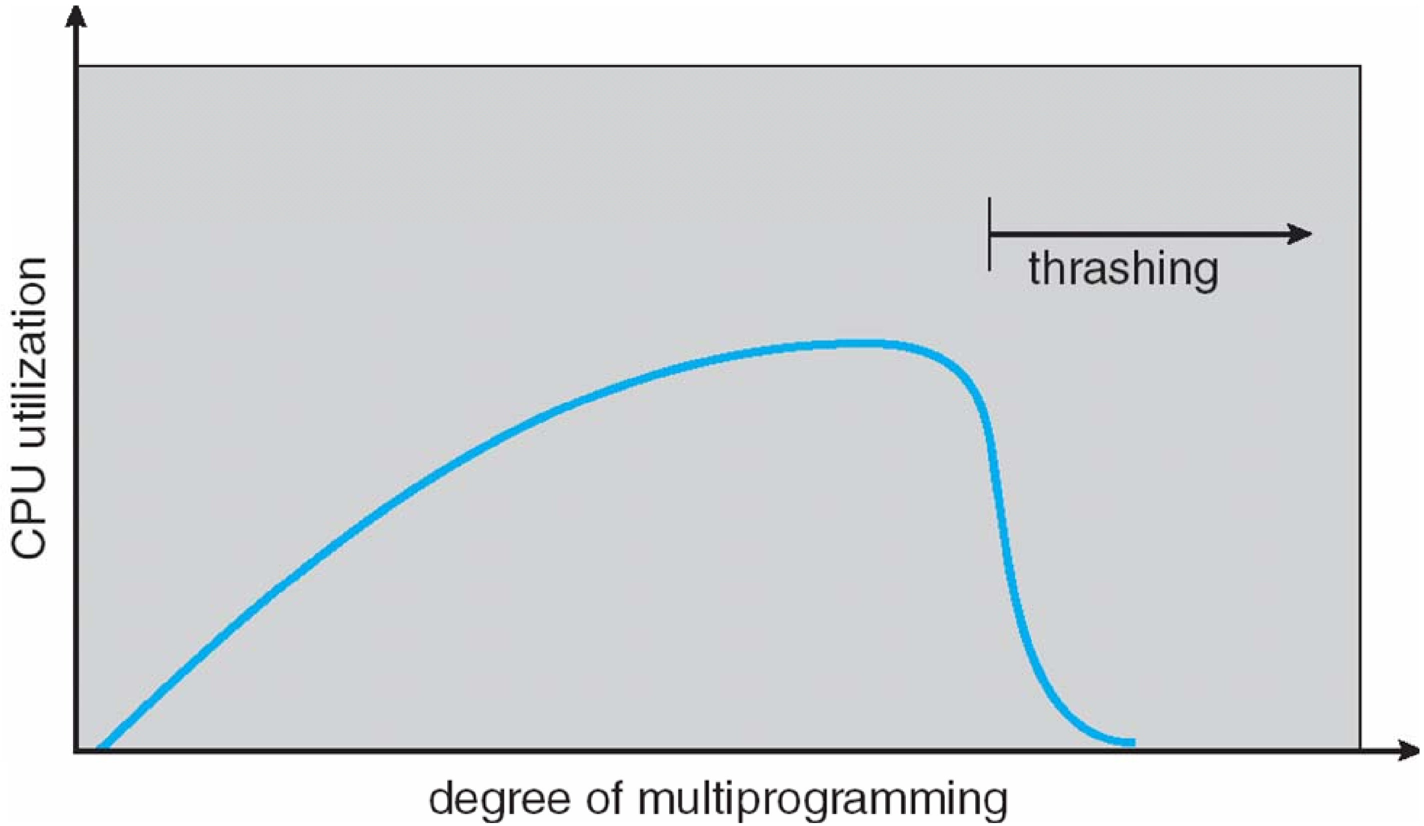
Global vs. Local Allocation

- **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

Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system
- **Thrashing** $\equiv$ a process is busy swapping pages in and out

Thrashing (Cont.)

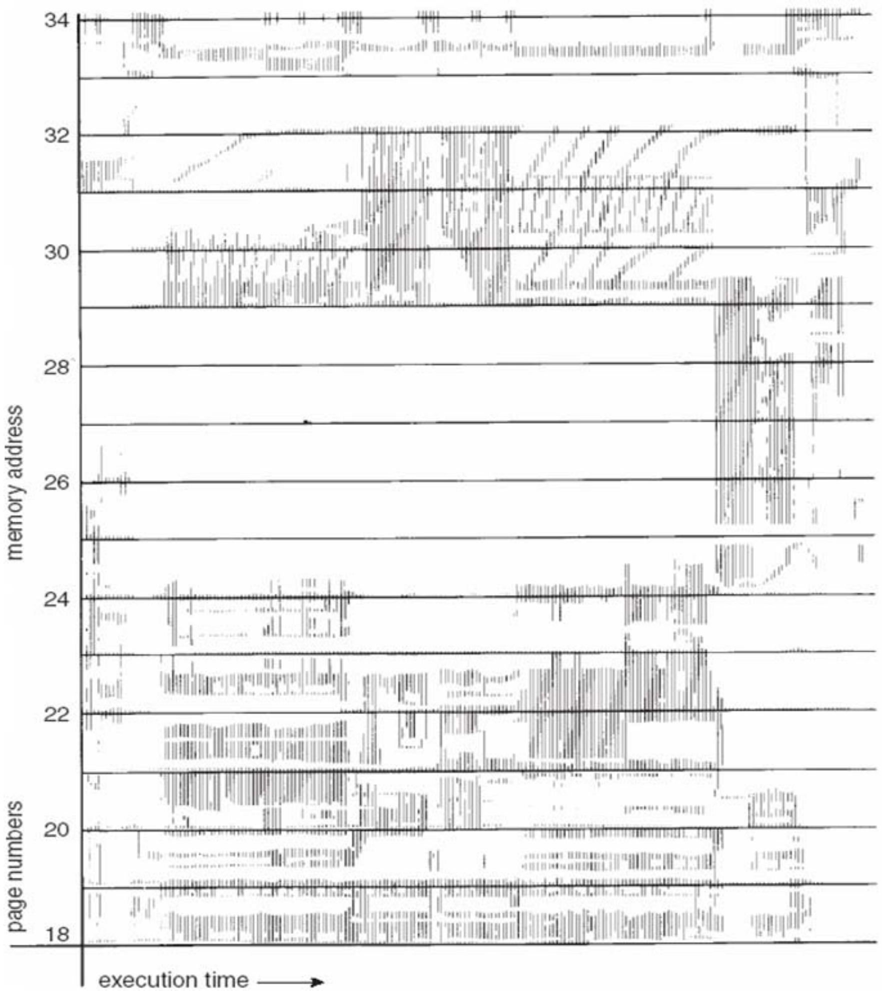


Demand Paging and Thrashing

- 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

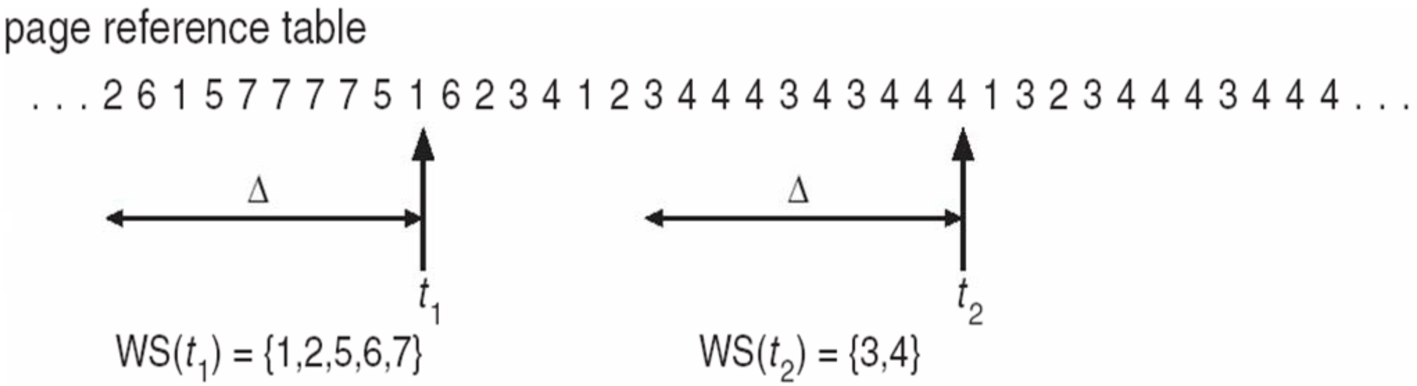
Locality In A Memory-Reference Pattern



Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instruction (instruction =? page_ref)
- WSS_i (working set 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
- if $D > m \Rightarrow$ Thrashing
- Policy if D > m, then suspend one of the processes

Working-set model

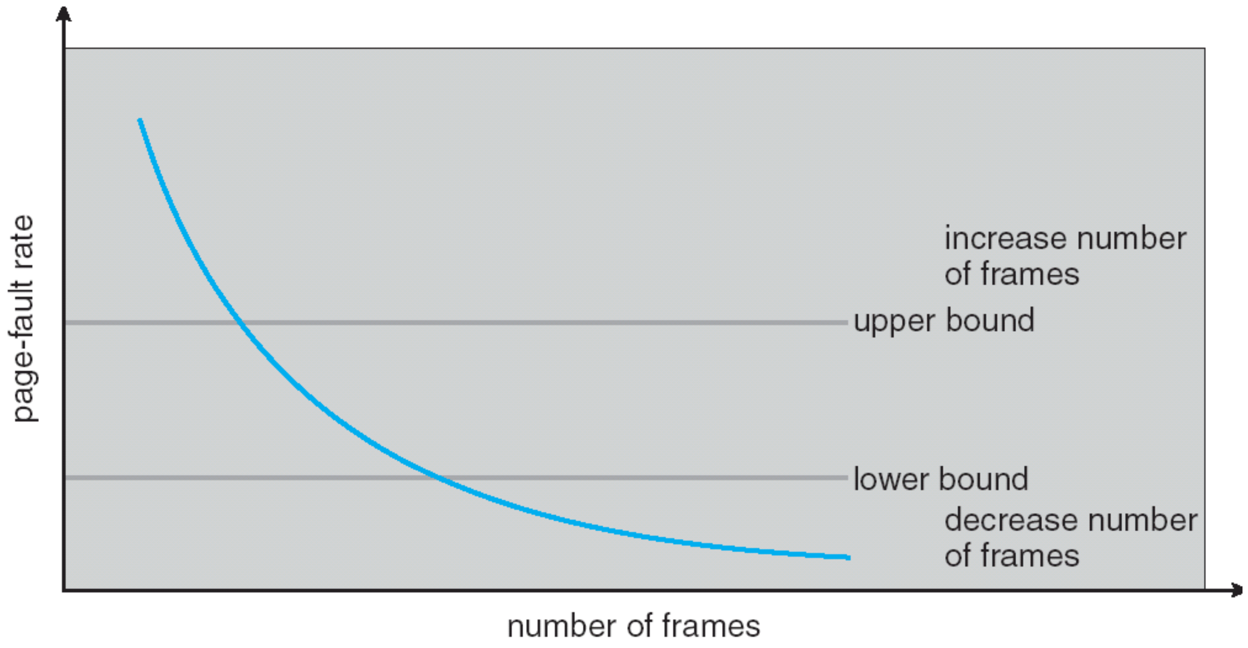


Keeping Track of the Working Set

- 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 $\Rightarrow$ page in working set
- Not accurate, because window is moving in large steps
  - ➢ Improvement = 10 bits and interrupt every 1000 time units

Page-Fault Frequency Scheme

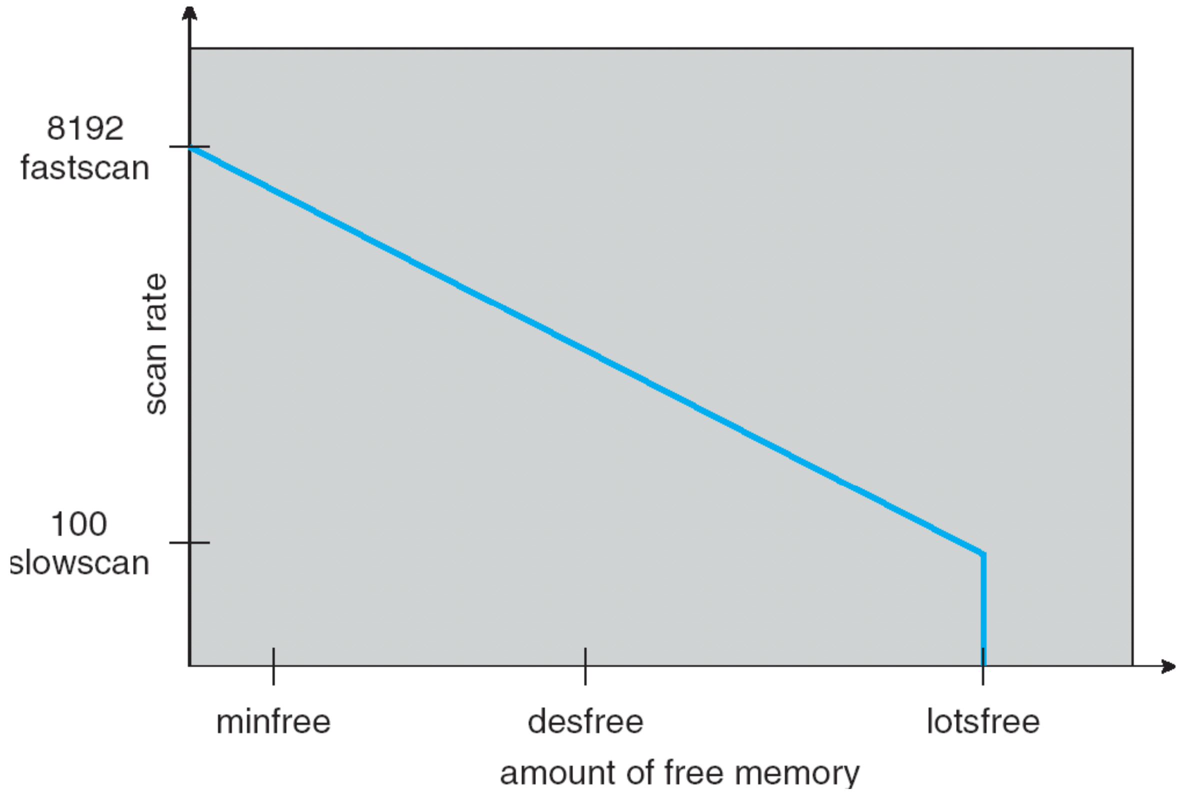
- Establish “acceptable” page-fault rate
  - If actual rate too low, process should lose frames
  - If actual rate too high, process should gain frames



Solaris

- Maintains a list of free pages to assign faulting processes
- *Lotsfree* – threshold parameter (amount of free memory) to begin paging
- *Desfree* – threshold parameter to increasing paging (**des**ired **free**)
- *Minfree* – threshold parameter to being swapping
- Paging is performed by *pageout* process
- Pageout scans pages using modified clock algorithm
- *Scanrate* is the rate at which pages are scanned. This ranges from *slowscan* to *fastscan*
- Pageout is called more frequently depending upon the amount of free memory available

Solaris 2 Page Scanner



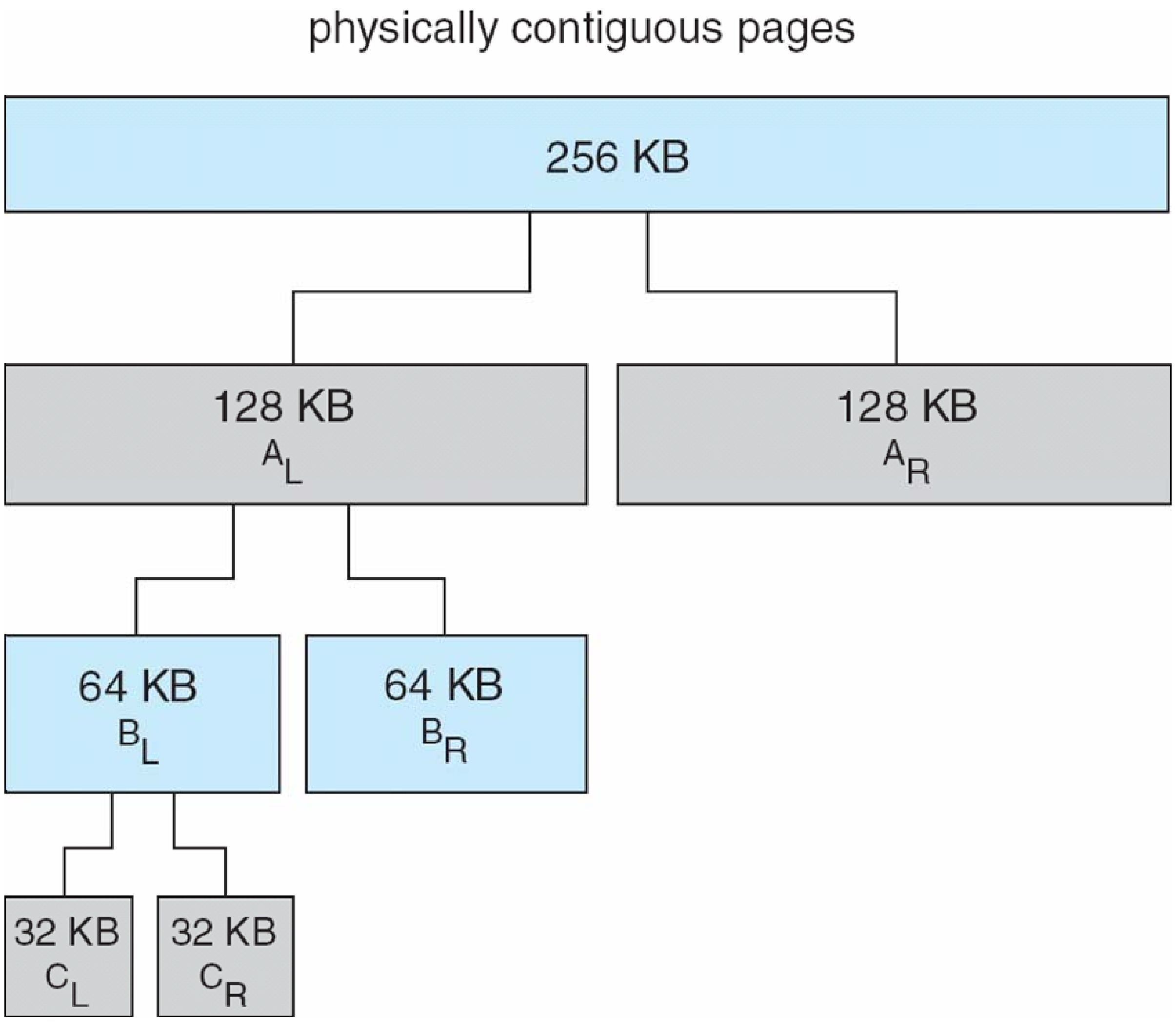
Allocating Kernel Memory

- 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

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

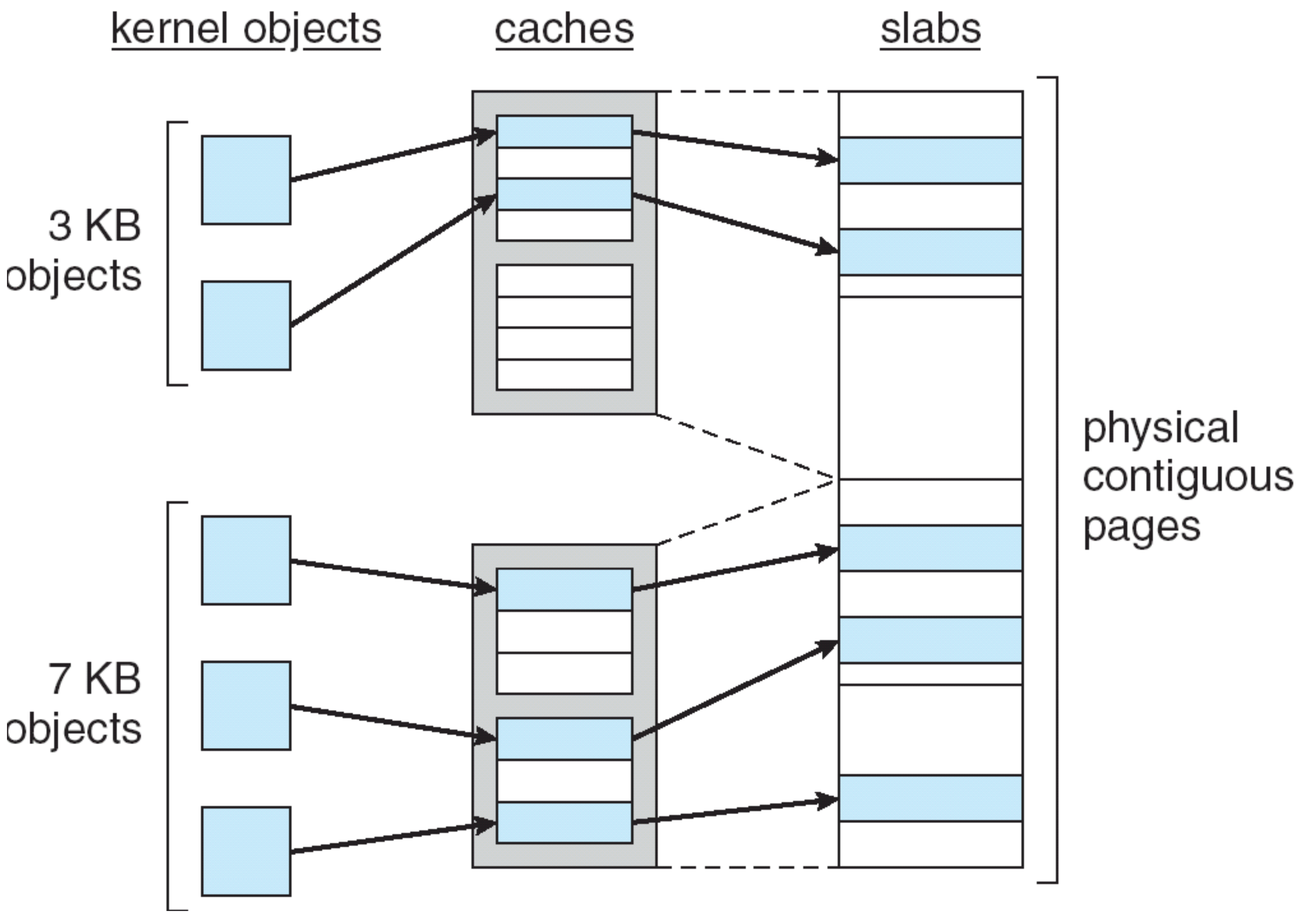
Buddy System Allocator



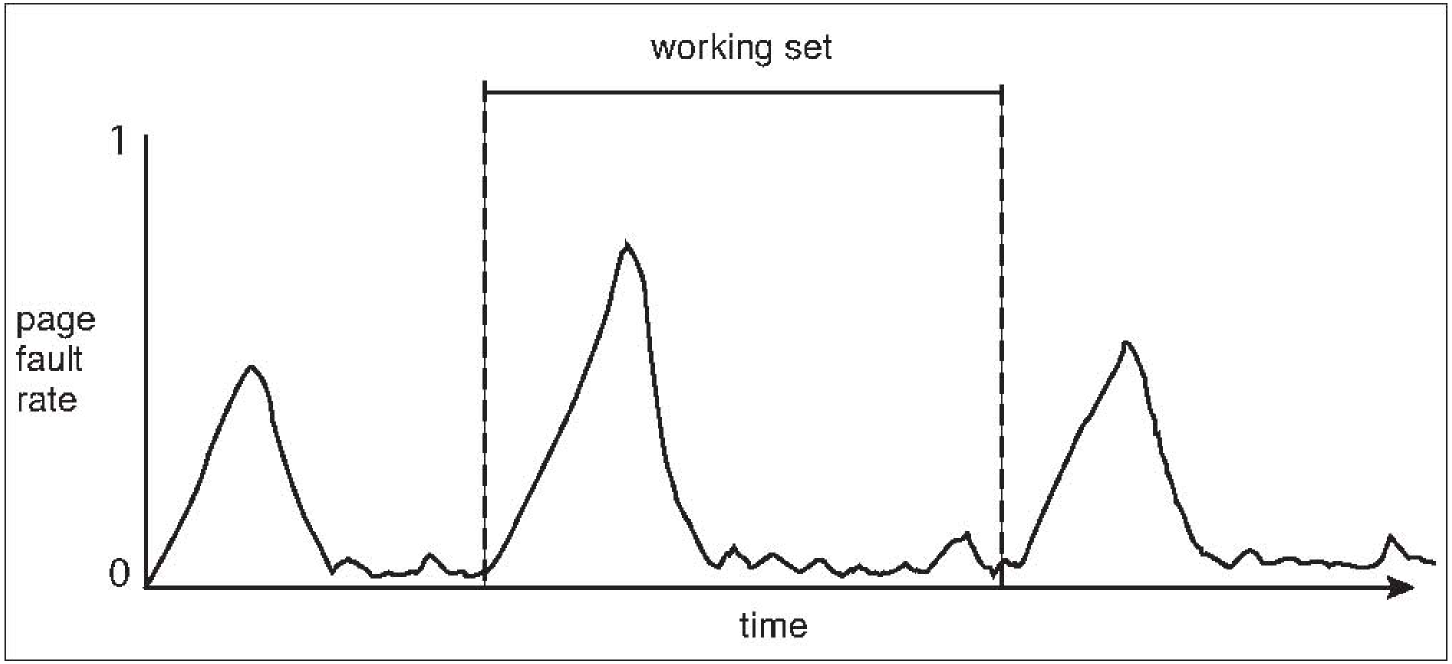
Slab Allocator

- 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



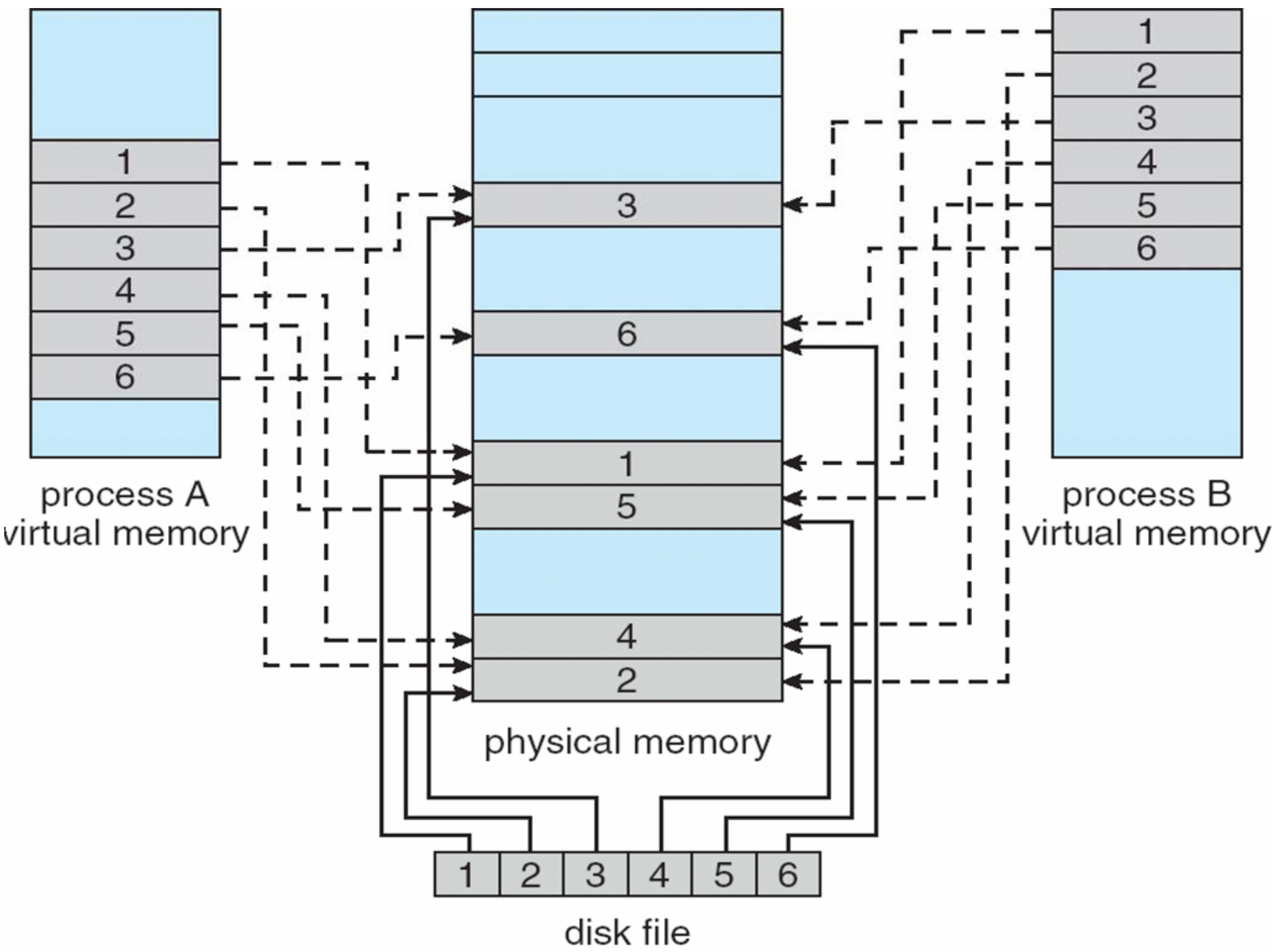
Working Sets and Page Fault Rates



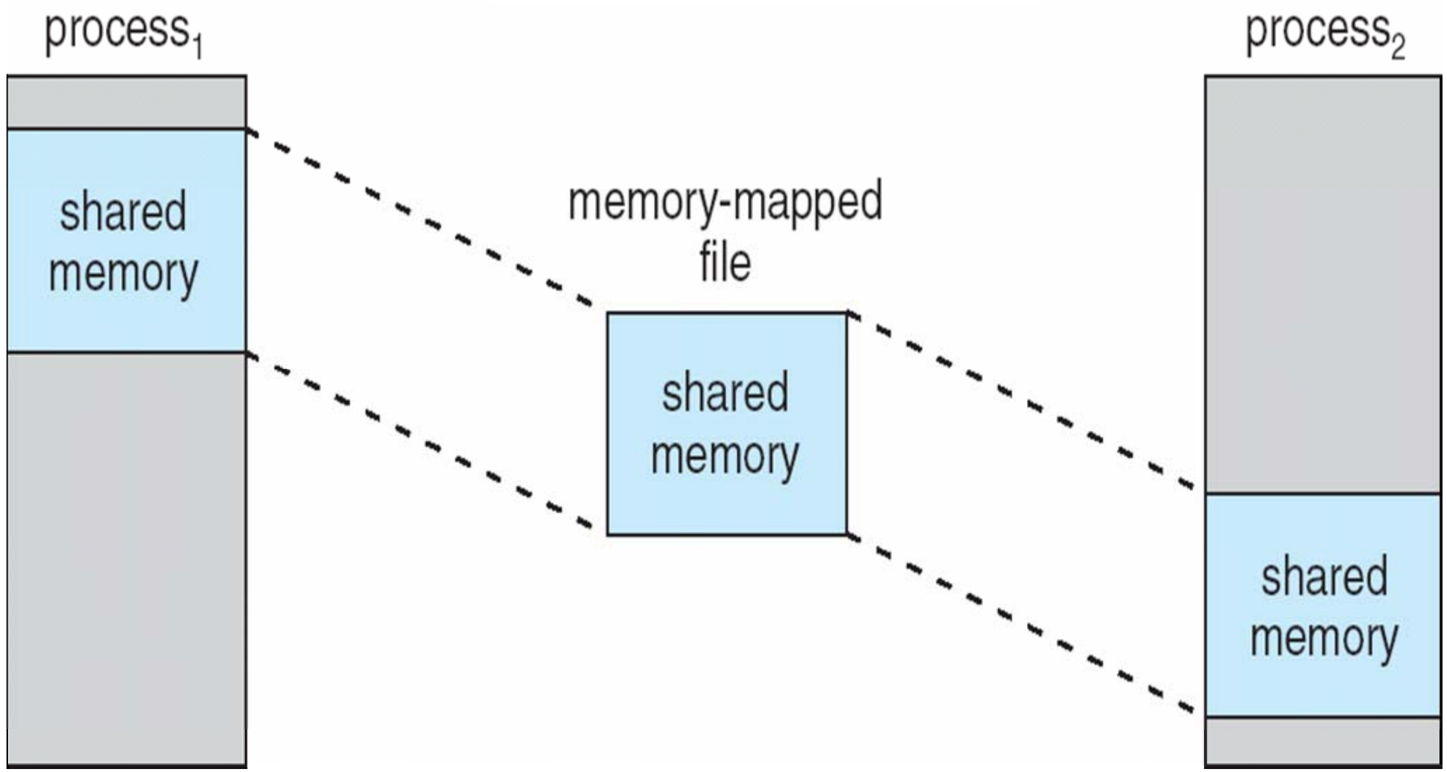
Other Issues -- Memory-Mapped Files

- 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



Other Issues – Page Size

- Page size selection must take into consideration:
  - fragmentation
  - table size
  - I/O overhead
  - locality

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
  - 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;`
  - Each row is stored in one page (e.g., 512 bytes page size)
  - 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