Virtual Memory

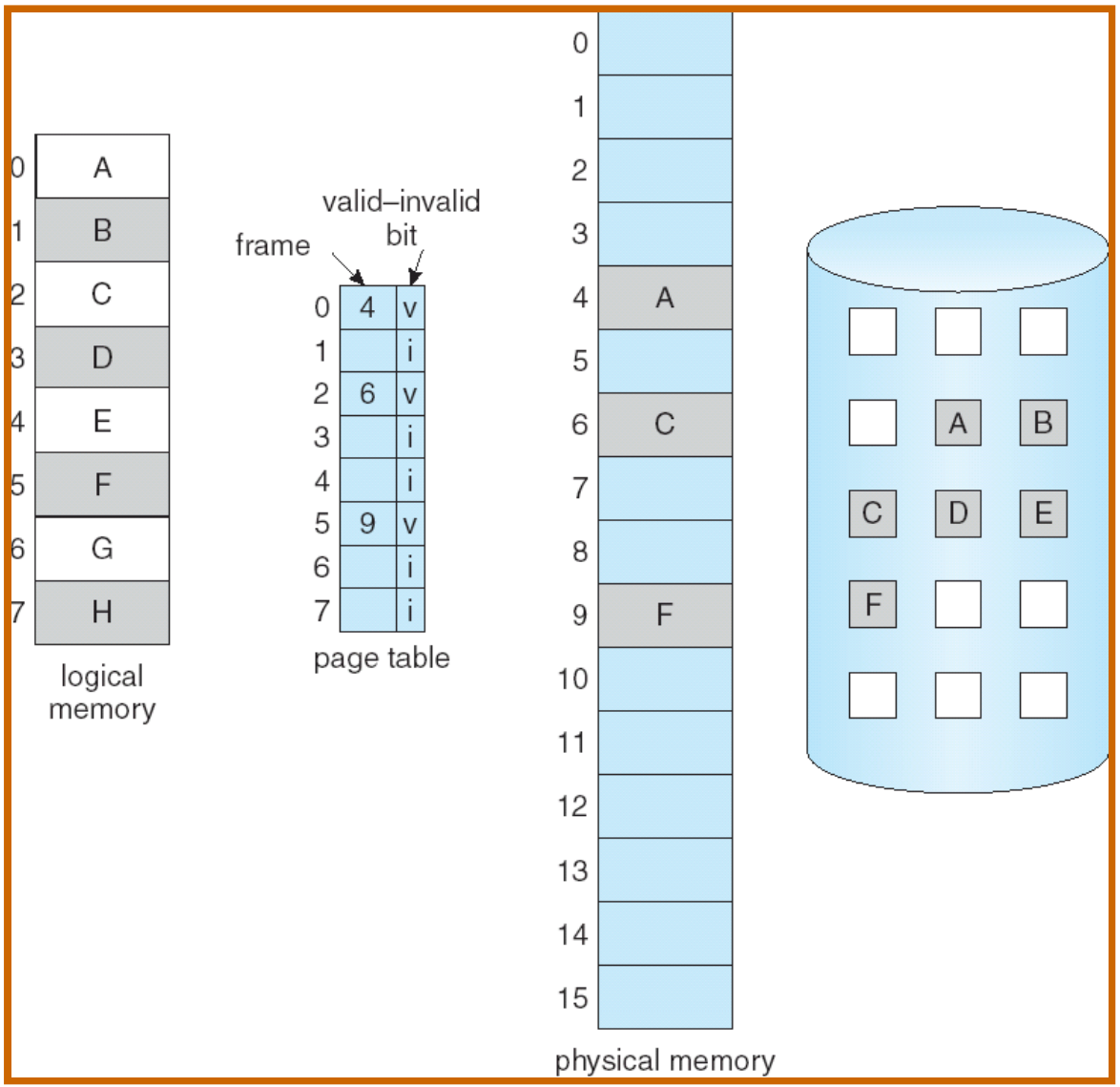
Virtual Memory

- **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.
  - Allows memory to be shared by many processes.
  - Allows for more efficient process creation.

Demand Paging

- Bring a page into memory only when it is referenced.
- Valid/Invalid bit can be used to determine if the page is in memory.
- Invalid bit has two possible meanings:
  - bad reference => abort
  - not-in-memory =>bring to memory
- Reference to an invalid page results in a page fault.

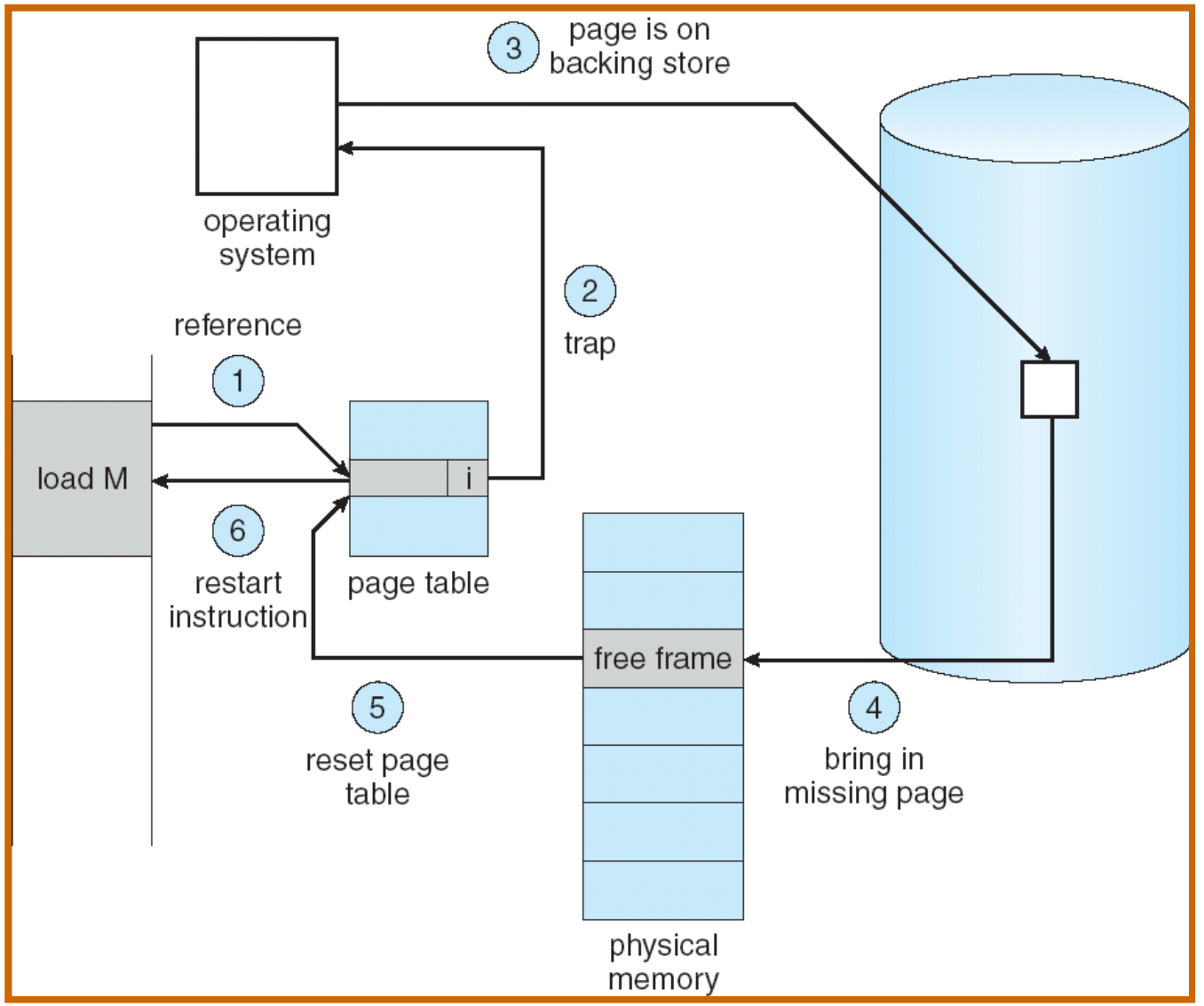
Paging Diagram



Responding to a Page Fault

- Operating system looks at PCB to decide:
  - Invalid reference => abort process.
  - Just not in memory =>
- Get an empty frame.
- Swap page into frame.
  - Requires disk I/O.
- Reset tables.
- Set validation bit = v.
- Restart the instruction that caused the page fault.

Page Fault Diagram



Performance of Demand Paging

- Page Fault Rate $0 \le p \le 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault

- Effective Access Time (EAT)

EAT = (1 – p) x memory access
+ p x (page fault overhead
+ swap page out
+ swap page in
+ restart overhead)

EAT 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.

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.
- Any ideas on how to implement this?

Page Replacement

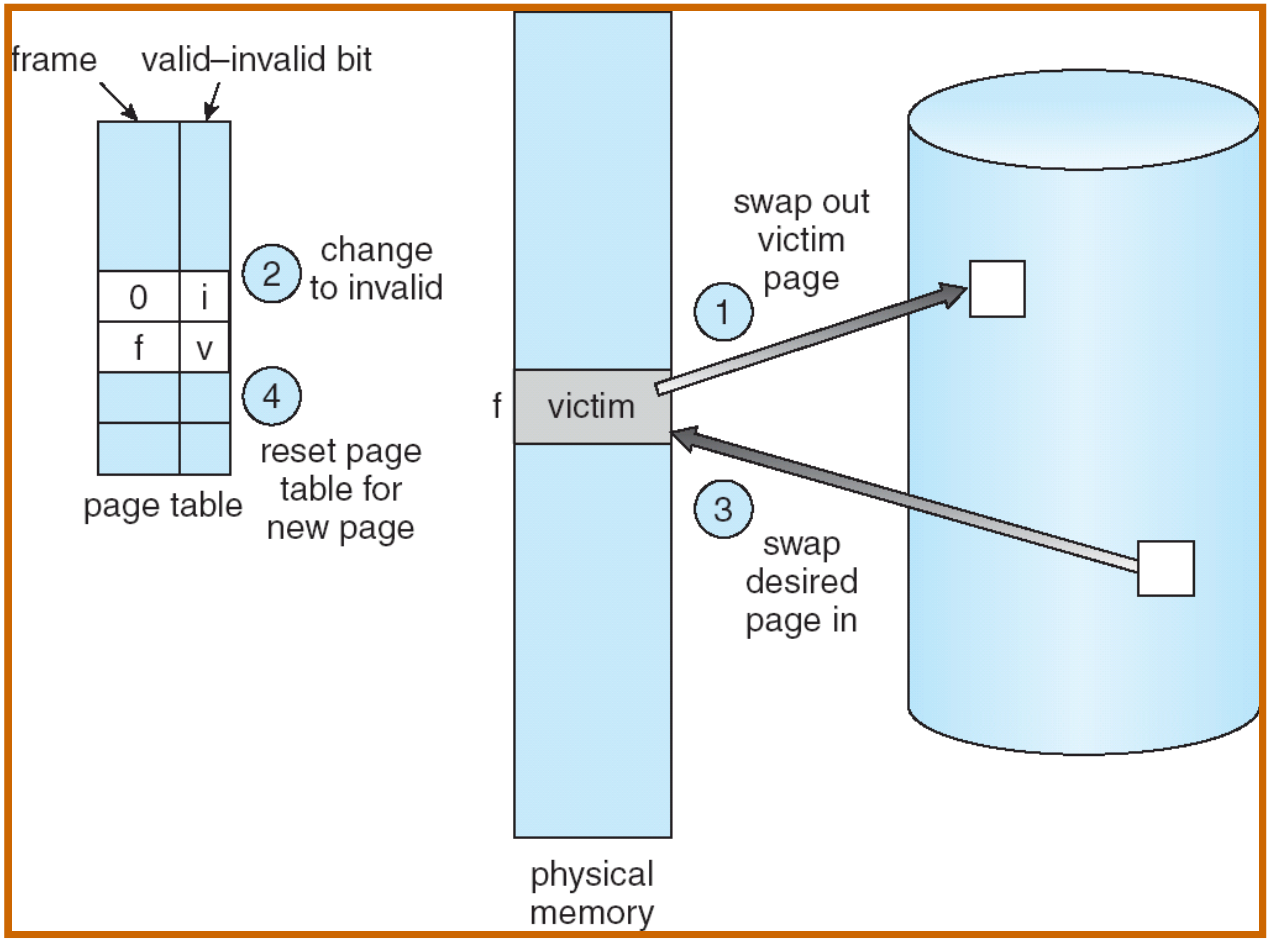
- What happens when there are no free frames to allocate?
- We could:
  - Abort the process.
  - Completely swap out another process.
  - Swap out a single frame.

Page Replacement

1. Find the location of the desired page on disk.
2. Obtain a free frame:
   - If there is a free frame available, use it.
   - If there is no free frame, use a page replacement algorithm to select a victim frame.
   - Write the victim frame to disk.
3. Bring the desired page into the free frame; update the page and frame tables.
4. Restart the process.

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

Page Replacement Diagram



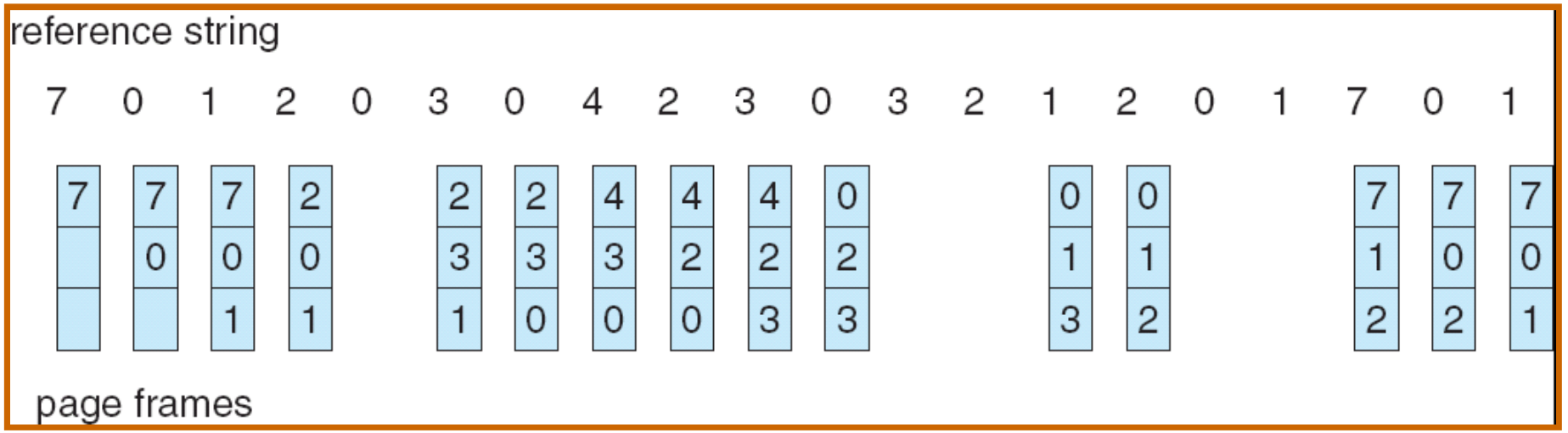
Page Replacement Algorithms

- Goal is to minimize # of page faults.
- We can evaluate an algorithm by counting page faults on a particular string of memory accesses.
  - reference string.
- Example from the book:
  - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- Why no repeats?

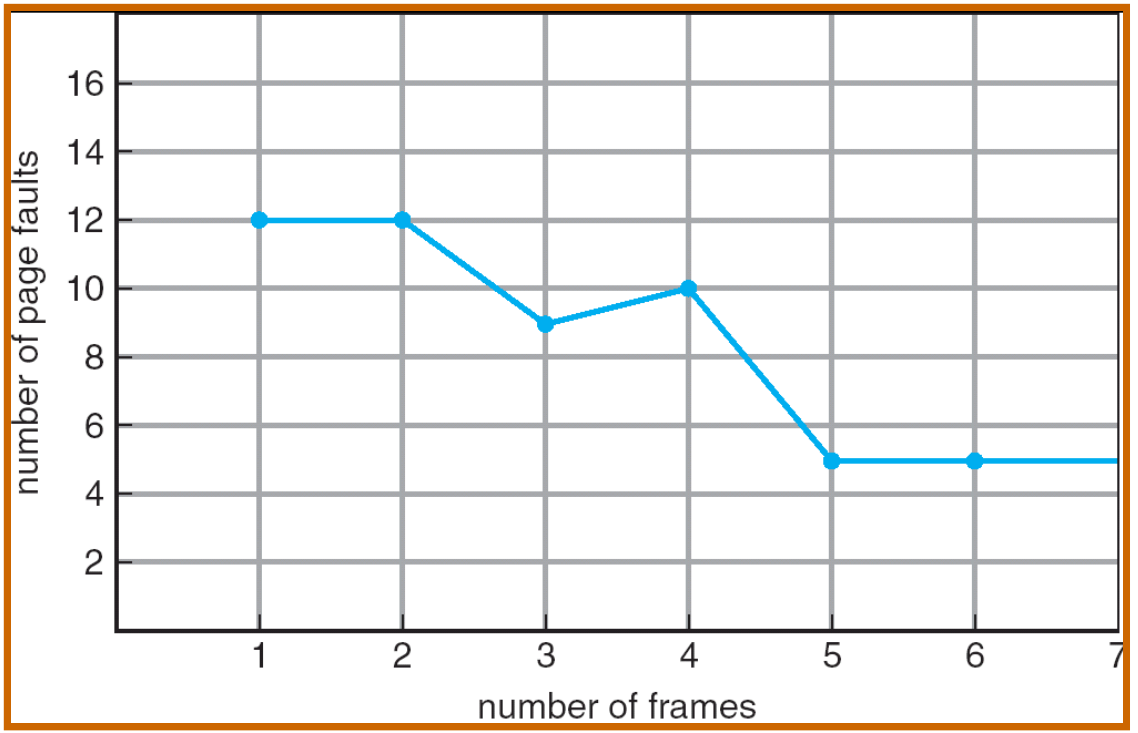
Page Replacement Algorithm: FIFO

- The “oldest” frame in memory is selected.
- What do you think?
- It has at least one bad property:
  - The number of page faults does not necessarily decrease when the number of frames increases.
  - Belady's anomaly.

FIFO Example



Belady's Anomaly



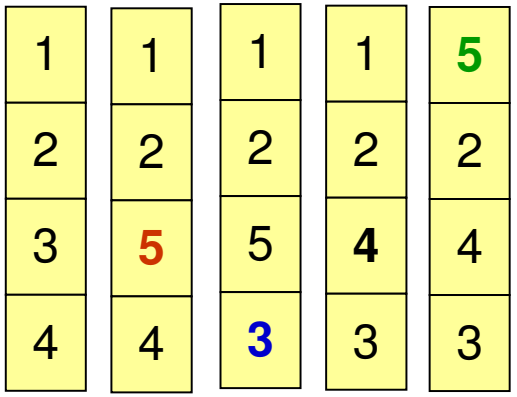
- This is a symptom of a non-optimal page replacement algorithm.
- What is the optimal page replacement algorithm?

Optimal Page Replacement Algorithm

- Replace the frame that will not be used for the longest period of time.
- Any practical difficulties with this?
- Provides a good reference for comparing other algorithms.

LRU Page Replacement

- Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**



- Counter implementation:
  - Every page entry has a counter; every time page is referenced, copy CPU clock into the counter.
  - When a page needs to be changed, look at the counters.
- Any drawbacks?

LRU Page Replacement

- Stack implementation – keep a stack of page numbers as a doubly linked list:
  - Page referenced:
    - move it to the top.
    - requires 6 pointers to be changed.
  - No search for replacement.
- Any drawbacks?
- Aside: When a page fault occurs we can afford to spend many cycles on the replacement algorithm.

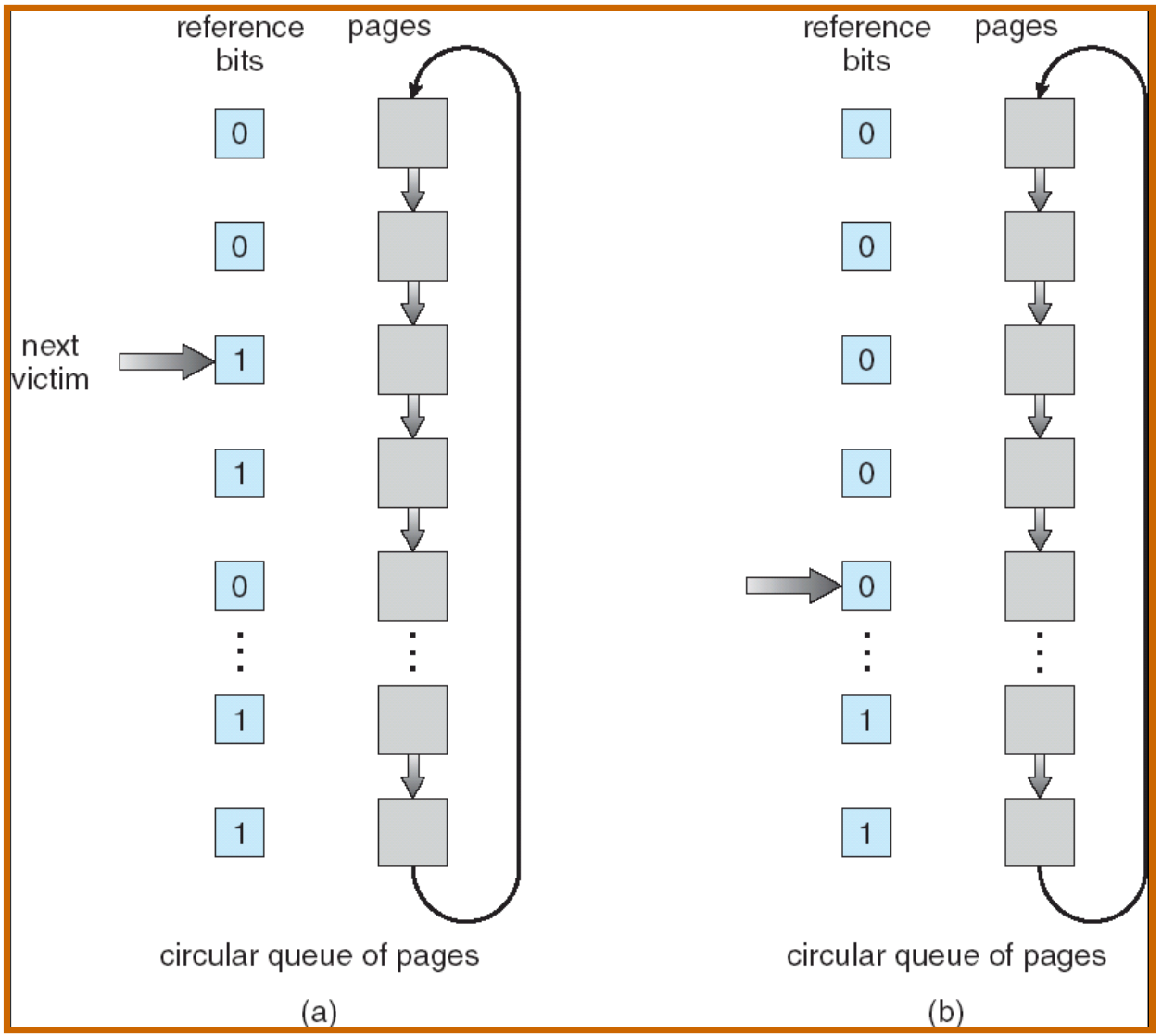
LRU Approximation Algorithms

- Reference bit:
  - With each page associate a bit, initially = 0.
  - When page is referenced bit set to 1.
  - Replace one which is 0 (if one exists).
  - Not very fine grained.
- Additional Reference Bits:
  - Keep an eight (*n*) bit history of reference bits.
  - Shift the bits right every 100ms.
  - Page with the lowest binary number is LRU.

LRU Approximation Algorithms

- Second Chance:
  - Need reference bit.
  - Clock replacement (FIFO).
  - 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.
- Enhanced Second Chance:
  - Consider dirty bit as well as reference bit.
  - Results in four classes of pages.

Second Chance Diagram



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.

Page Buffering

- Several variations:
  - Keep a pool of free frames.
    - When a victim is selected, copy it to a free frame from the pool.
    - Let the process restart before that frame actually gets written to disk.
  - Preemptively copy dirty frames to disk.
  - Keep a pool of free frames, and remember which pages they stored.
    - If one of those pages is requested again, no need to go to disk.

Global vs. Local Allocation

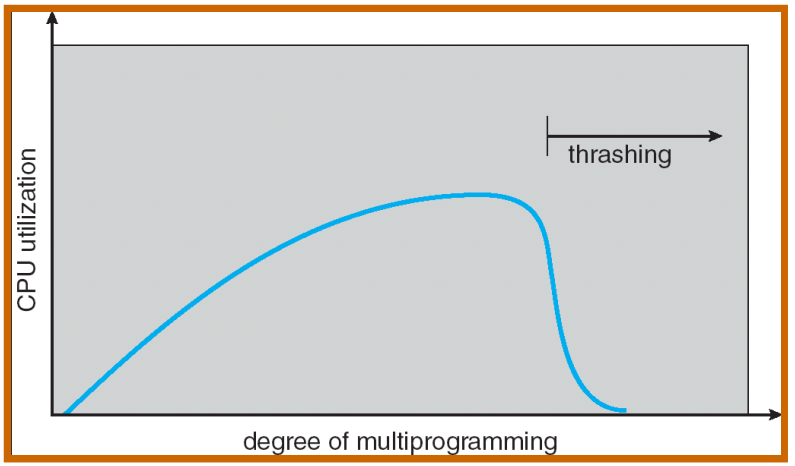
- Where do victim frames come from?
- Global replacement – process selects a replacement frame from the set of all frames.
  - One process can take a frame from another.
- Local replacement – each process selects from its own frames.
  - How many frames should each process get?

Allocation of Frames

- Every process needs an (architecture dependent) minimum number of frames.
  - If one instruction could use three frames, the process might *need* three frames.
- Two basic strategies:
  - Fixed allocation:
    - Equal or proportional.
  - Priority allocation.

Thrashing

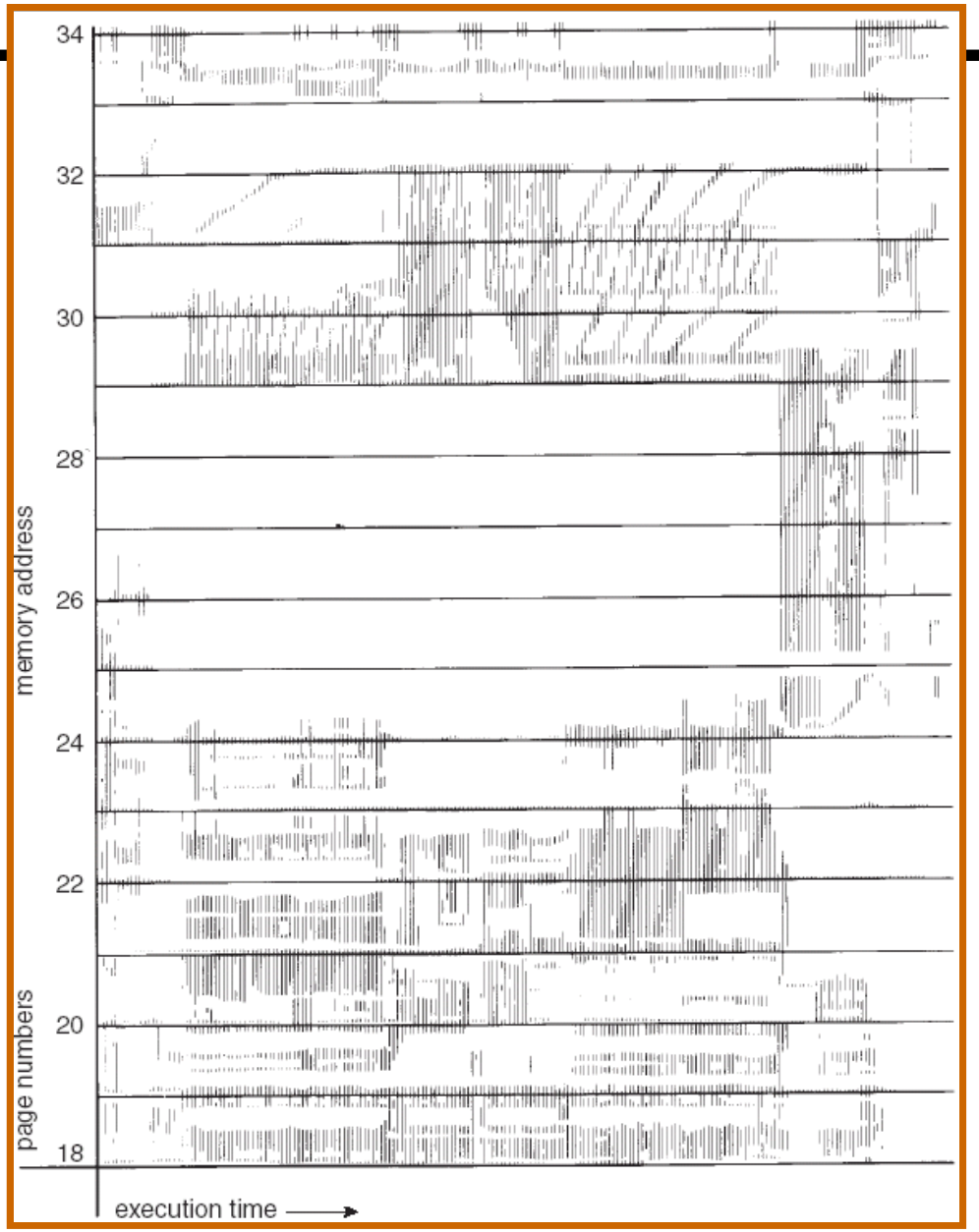
- If a process does not have “enough” pages, the page-fault rate is high.
- **Thrashing** – a process spends most of its time swapping.
- If the total memory usage of running processes is larger than the amount of memory available, this starts to happen.



Demand Paging and Thrashing

- Why does demand paging work?
- Locality model (the basic principle of caches)
  - Process migrates from one locality to another.
  - Localities may overlap.
- Why does thrashing occur?
  - Total size of localities > total memory size.

Locality Example



Working Set Model

- Δ = working-set window, a fixed number of page references.
- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ .
  - if Δ too small will not encompass entire locality.
  - if Δ too large will encompass several localities.
  - if $\Delta = \infty$ will encompass entire program

Using the Working Set Model

- $D = \sum WSS_i$, total demand frames
- if D > m, thrashing results
- If D > m, then suspend one process.
- Tracking the working set similar to approximating LRU pages.
  - Keep a record of reference bit settings.

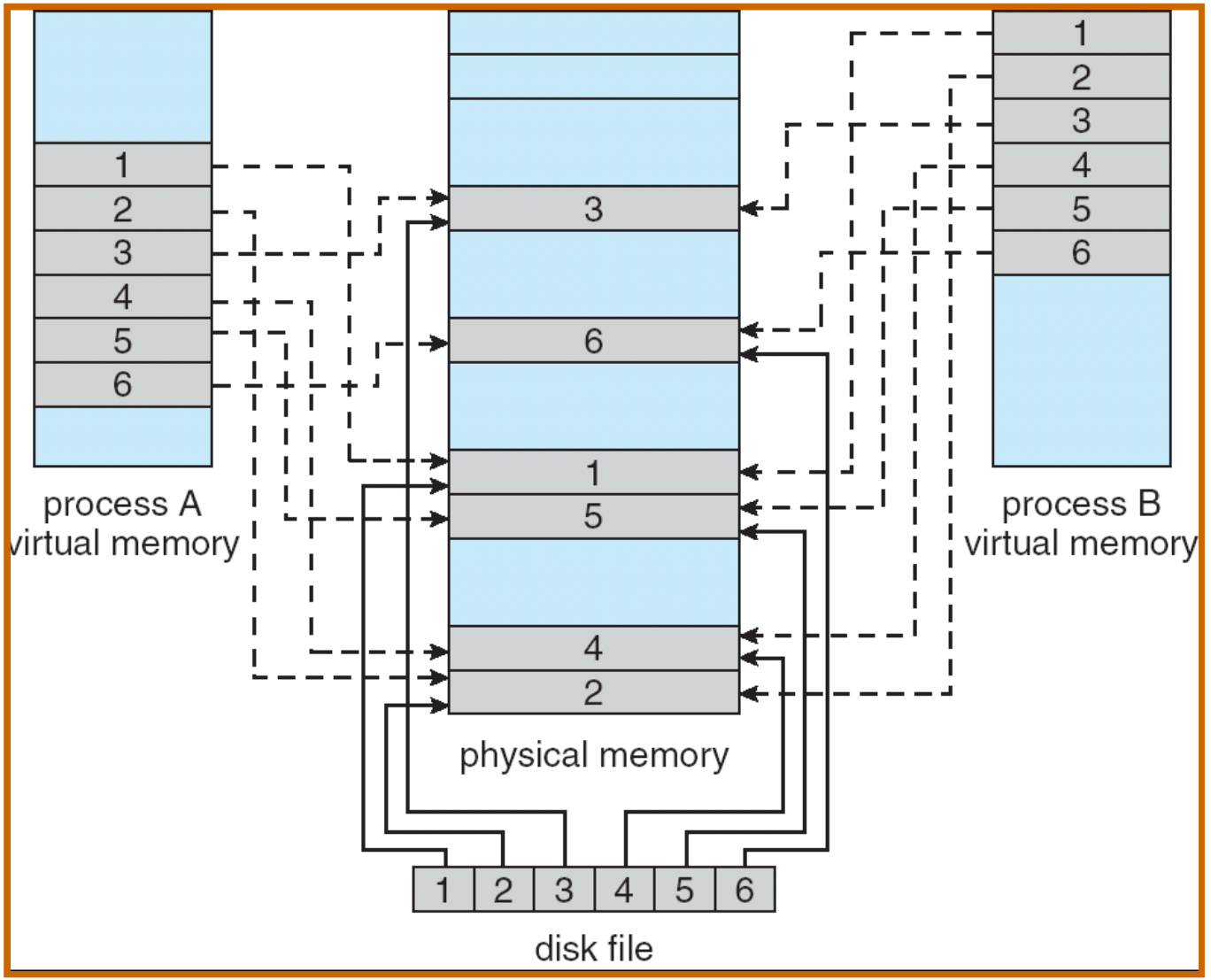
Page Fault Frequency

- Another mechanism to control thrashing.
- Just track the page fault rate for each process.
  - If the rate is high, allocate more frames.
  - If the rate is low, take frames away.
  - If a process needs more frames, but none are available, suspend a process.

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.
- The system call in Unix-like systems is **`mmap()`**.

Memory Mapped File Picture



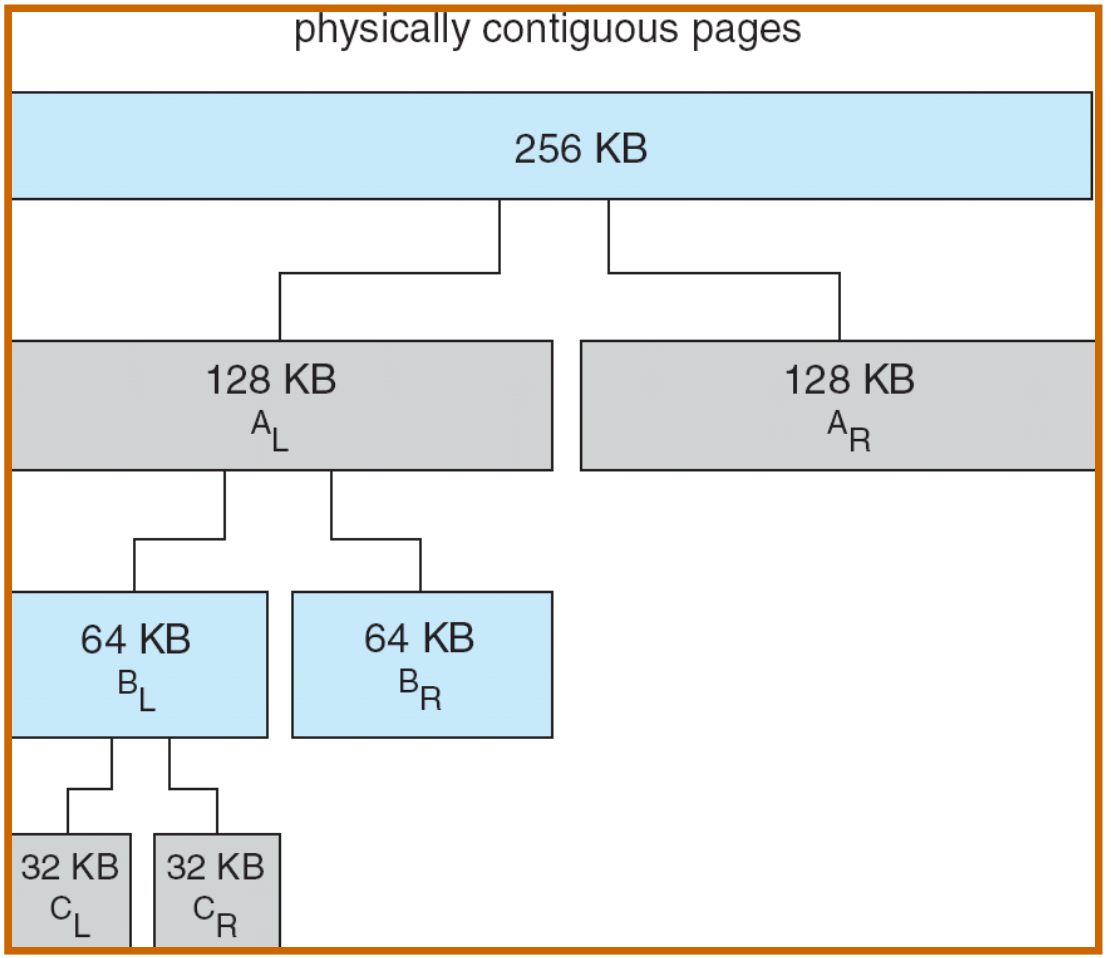
Kernel Memory Allocation

- Treated differently from user memory.
- Often allocated from a free-memory pool.
  - Some kernel memory needs to be contiguous.
- A couple of approaches:
  - Buddy system.
  - Slab memory 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.
  - 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.

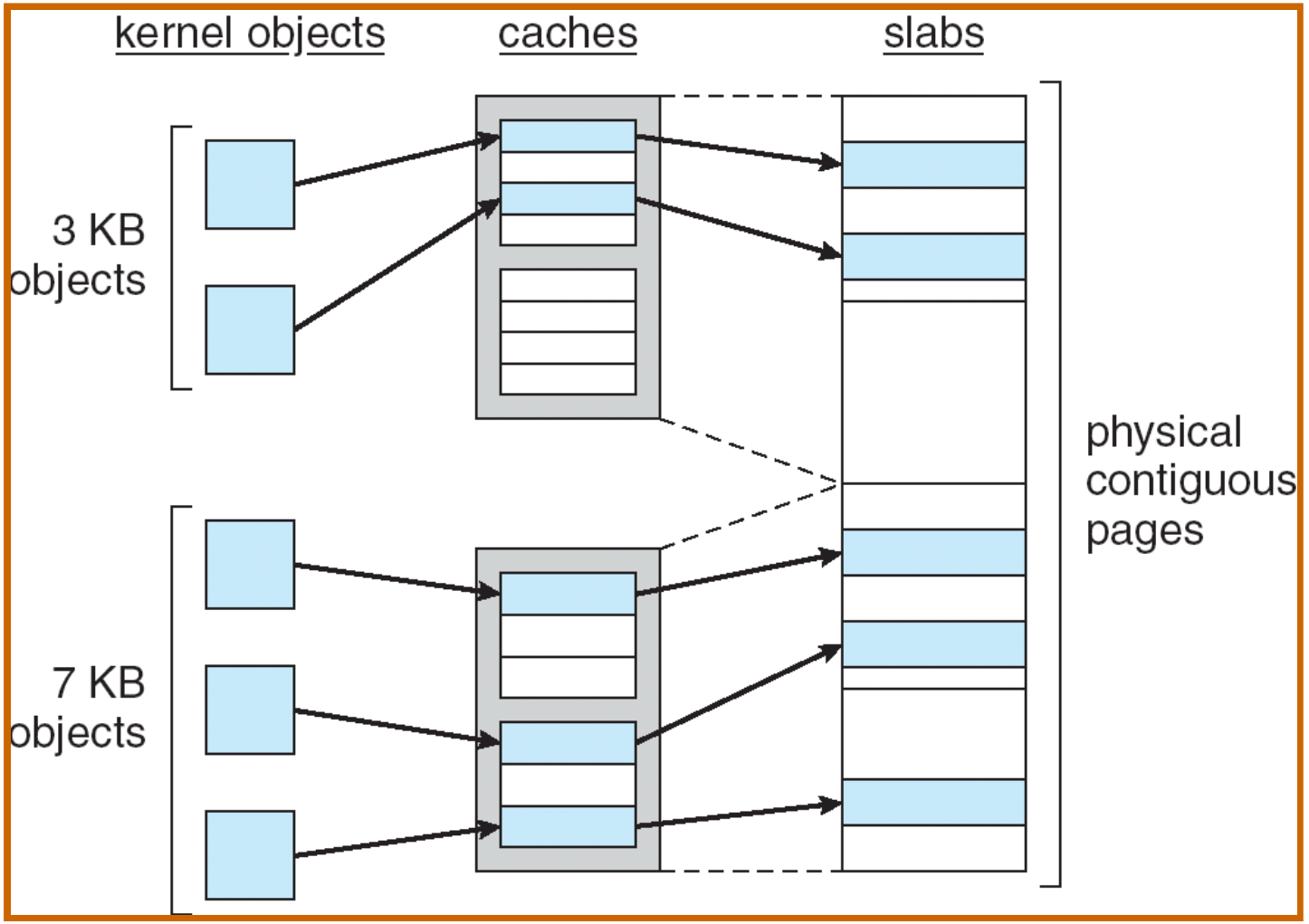
Buddy System



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
- Benefits include no fragmentation, fast memory request satisfaction

Slab Allocation



Other Issues

- Pre-paging.
- Page size.
- TLB-reach.

Program Structure

- Program structure
  - `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

  - Program 2

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

    128 page faults

Acknowledgments

- Portions of these slides are taken from Power Point presentations made available along with:
  - Silberschatz, Galvin, and Gagne. Operating System Concepts, Seventh Edition.
- Original versions of those presentations can be found at:
  - http://os-book.com/