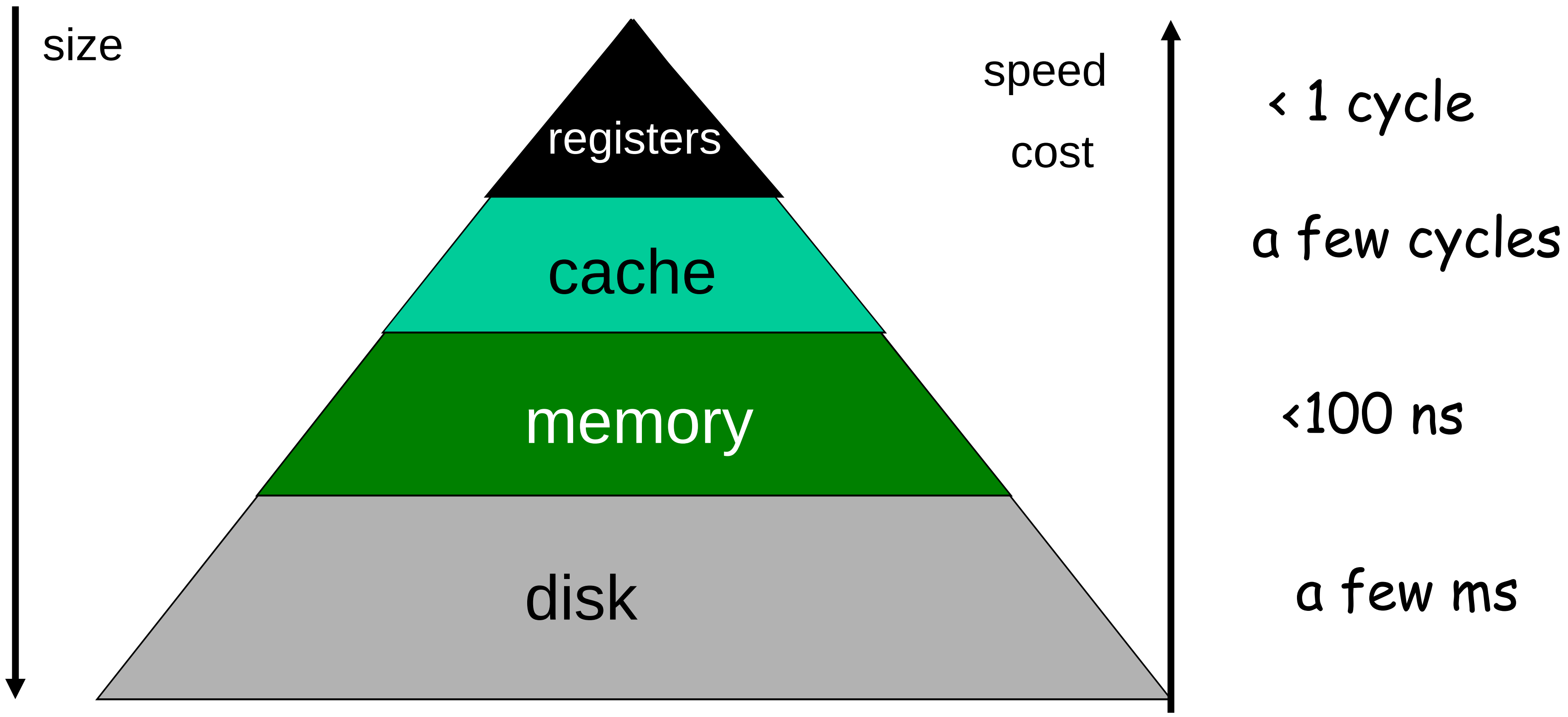
W4118 Operating Systems

Instructor: Junfeng Yang

Background: memory hierarchy

- Levels of memory in computer system



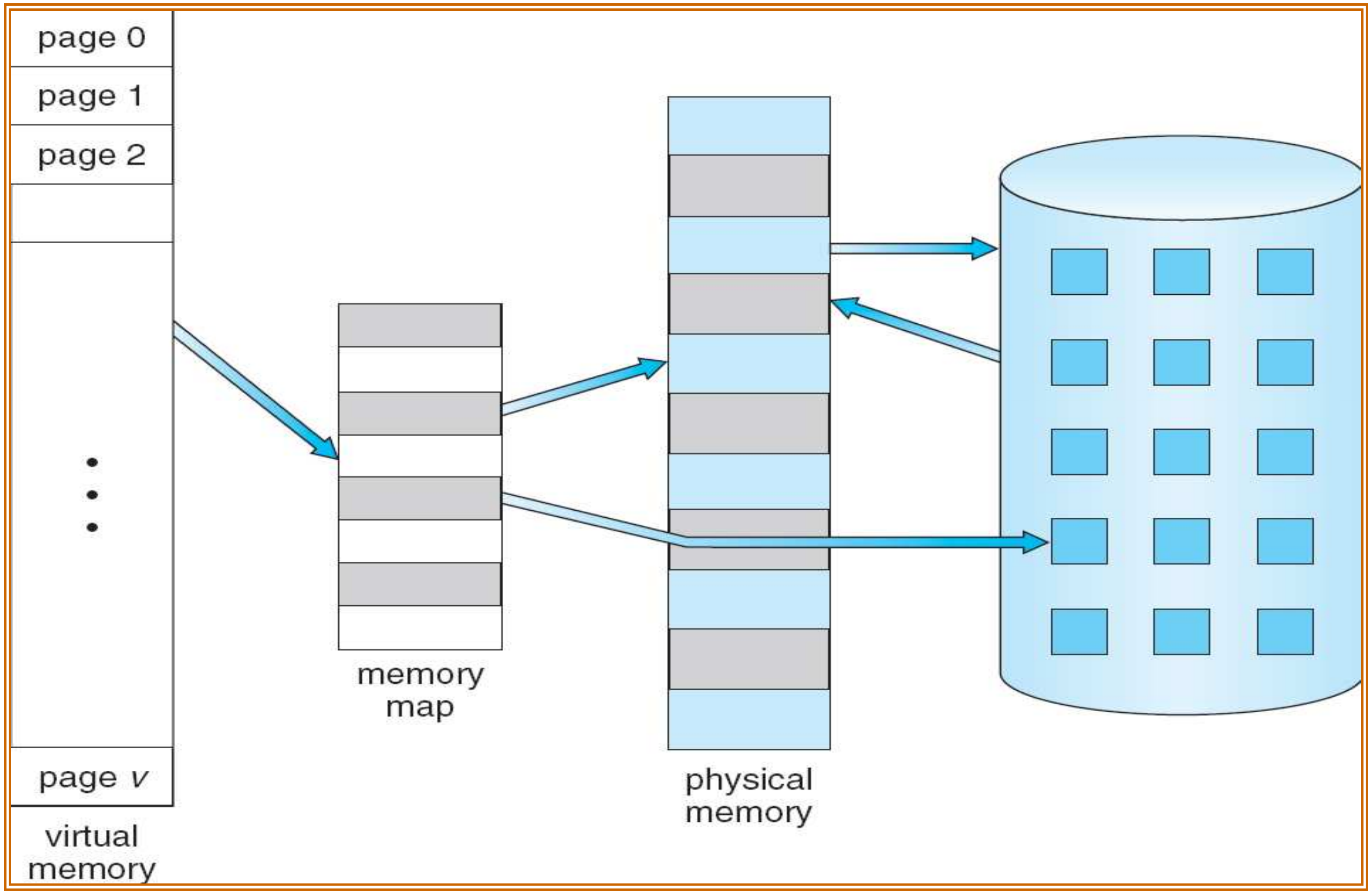
Virtual memory motivation

- Previous approach to memory management
  - Must completely load user process in memory
  - One large AS or too many AS ➔ out of memory
- Observation: locality of reference
  - Temporal: access memory location accessed just now
  - Spatial: access memory location adjacent to locations accessed just now
- Implication: process only needs a small part of address space at any moment!

Virtual memory idea

- OS and hardware produce illusion of a disk as fast as main memory
- Process runs when not all pages are loaded in memory
  - Only keep referenced pages in main memory
  - Keep unreferenced pages on slower, cheaper backing store (disk)
  - Bring pages from disk to memory when necessary

Virtual memory illustration



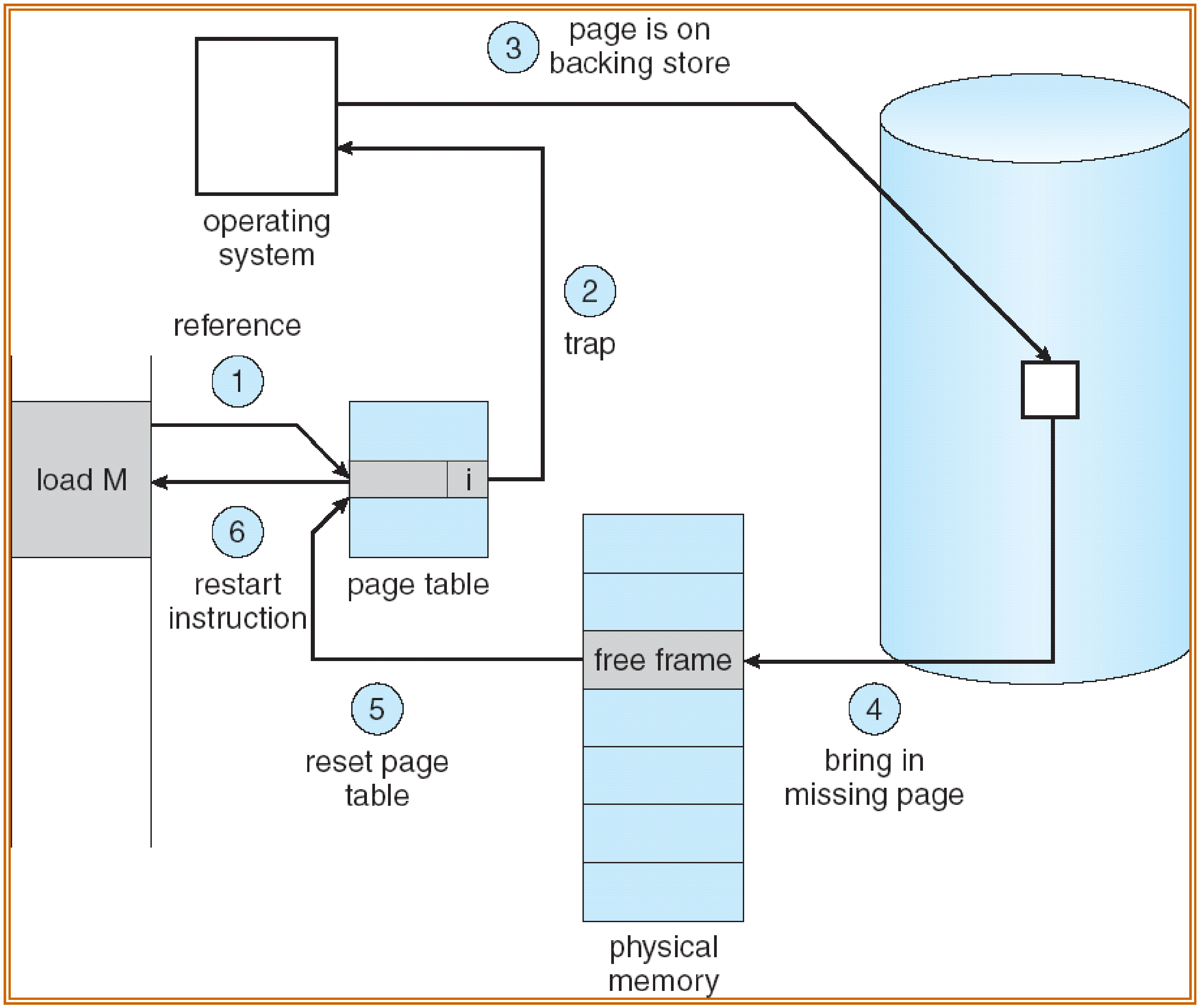
Virtual memory operations

- Detect reference to page on disk
- Recognize disk location of page
- Choose free physical page
  - OS decision: if no free page is available, must replace a physical page
- Bring page from disk into memory
  - OS decision: when to bring page into memory?
- Above steps need hardware and software cooperation

Detect reference to page on disk and recognize disk location of page

- Overload the valid bit of page table entries
- If a page is on disk, clear valid bit in corresponding page table entry and store disk location using remaining bits
- Page fault: if bit is cleared then referencing resulting in a trap into OS
- In OS page fault handler, check page table entry to detect if page fault is caused by reference to true invalid page or page on disk

Steps in handling a page fault



OS decisions

- Page selection
  - When to bring pages from disk to memory?
- Page replacement
  - When no free pages available, must select victim page in memory and throw it out to disk

Page selection algorithms

- Demand paging: load page on page fault
  - Start up process with no pages loaded
  - Wait until a page absolutely must be in memory
- Request paging: user specifies which pages are needed
  - Requires users to manage memory by hand
  - Users do not always know best
  - OS trusts users (e.g., one user can use up all memory)
- Prepaging: load page before it is referenced
  - When one page is referenced, bring in next one
  - Do not work well for all workloads
    - Difficult to predict future

Page replacement algorithms

- Optimal: throw out page that won't be used for longest time in future
- Random: throw out a random page
- FIFO: throw out page that was loaded in first
- LRU: throw out page that hasn't been used in longest time

Evaluating page replacement algorithms

- Goal: fewest number of page faults
- A method: run algorithm 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

Optimal algorithm

- Throw out page that won't be used for longest time in future

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

6 page faults

Problem: difficult to predict future!

Fist-In-First-Out (FIFO) algorithm

- Throw out page that was loaded in first

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

10 page faults

Problem: ignores access patterns

Fist-In-First-Out (FIFO) algorithm (cont.)

- Results with 3 physical pages

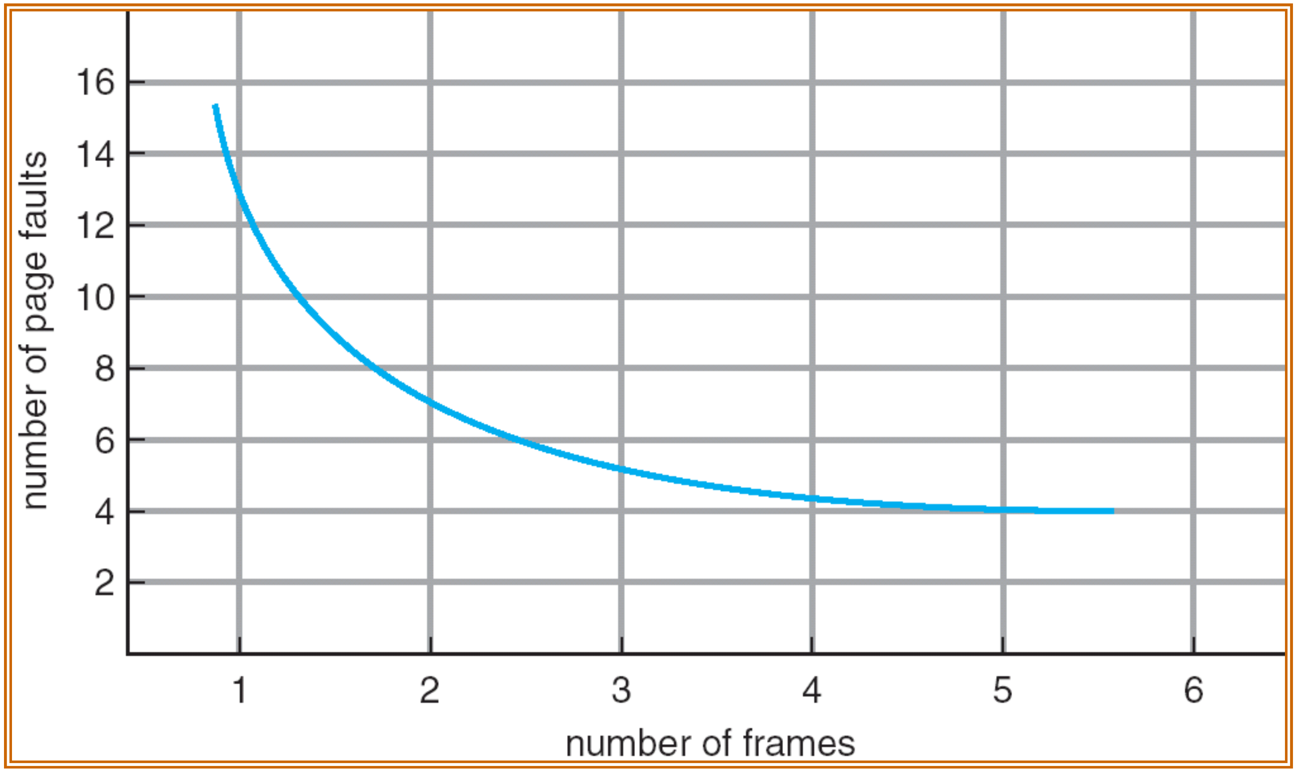
1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	4	4	4	5	5	5	5	5	5
	2	2	2	1	1	1	1	1	3	3	3
		3	3	3	2	2	2	2	2	4	4

9 page faults

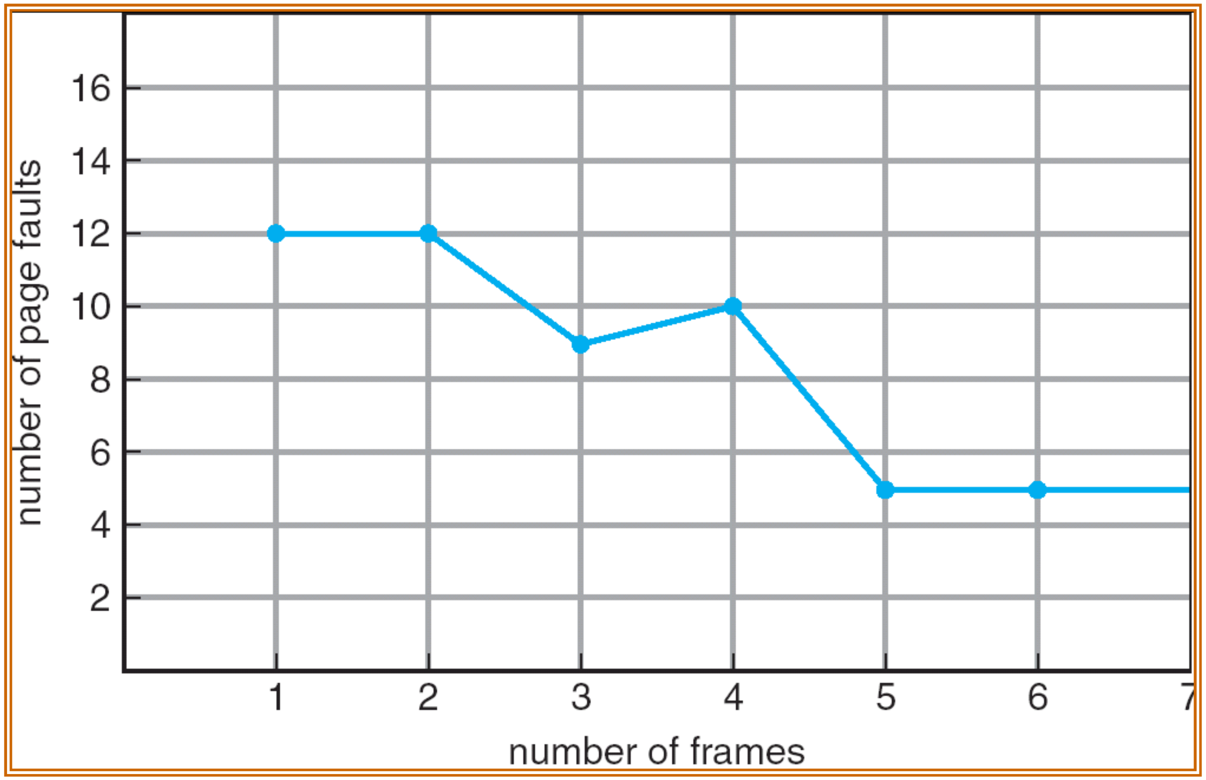
Problem: fewer physical pages → fewer faults!

belady anomaly

Ideal curve of # of page faults v.s. # of physical pages



FIFO illustrating belady's anomaly



Least-Recently-Used (LRU) algorithm

- Throw out page that hasn't been used in longest time. Can use FIFO to break ties

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

8 page faults

Advantage: with locality, LRU approximates Optimal

Implementing LRU: hardware

- A counter for each page
- Every time page is referenced, save system clock into the counter of the page
- Page replacement: scan through pages to find the one with the oldest clock
- Problem: have to search all pages/counters!

Implementing LRU: software

- A doubly linked list of pages
- Every time page is referenced, move it to the front of the list
- Page replacement: remove the page from back of list
  - *Avoid scanning of all pages*
- Problem: too expensive
  - Requires 6 pointer updates for each page reference
  - High contention on multiprocessor

LRU: concept vs. reality

- LRU is considered to be a reasonably good algorithm
- Problem is in implementing it efficiently
  - Hardware implementation: counter per page, copied per memory reference, have to search pages on page replacement to find oldest
  - Software implementation: no search, but pointer swap on each memory reference, high contention
- In practice, settle for efficient approximate LRU
  - Find an old page, but not necessarily the oldest
  - LRU is approximation anyway, so approximate more

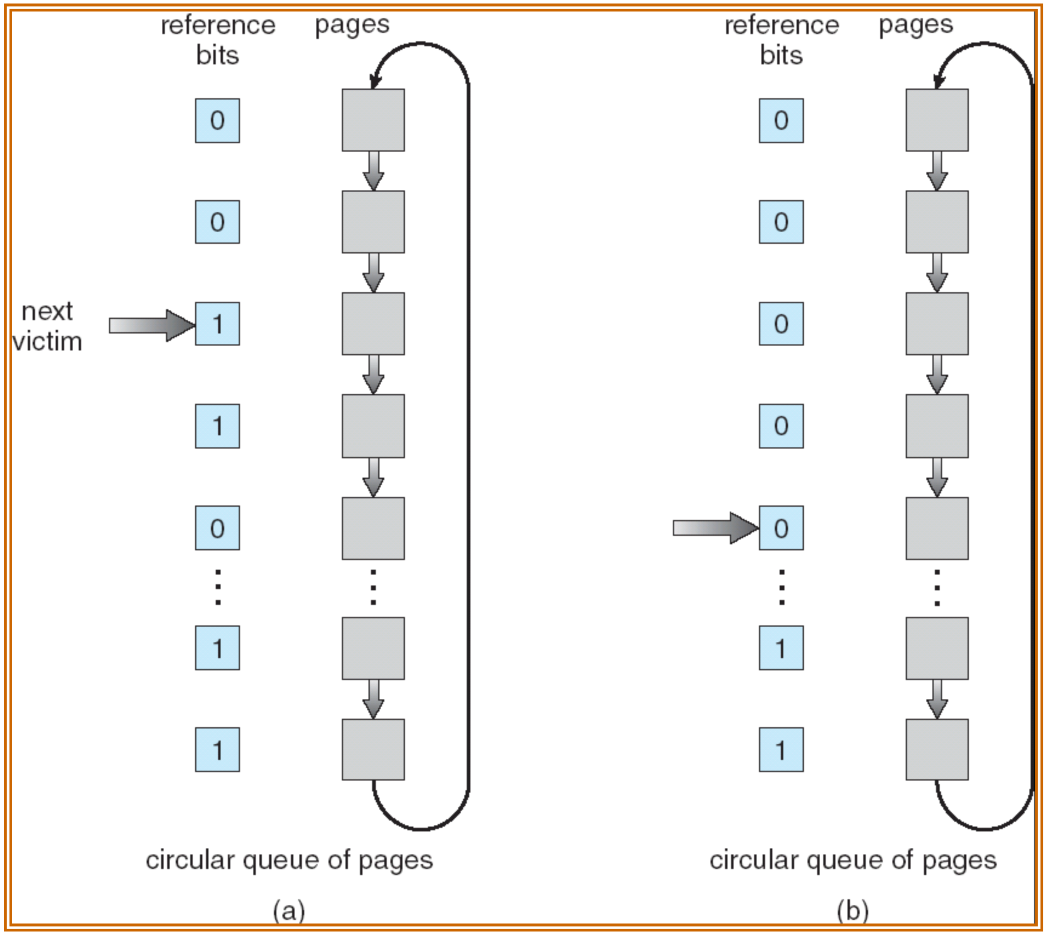
Clock (second-chance) algorithm

- Goal: remove a page that has not been referenced recently
  - good LRU-approximate algorithm
- Idea
  - A reference bit per page
  - Memory reference: hardware sets bit to 1
  - Page replacement: OS finds a page with reference bit cleared
  - OS traverses all pages, clearing bits over time
  - Combining FIFO with LRU: give the page FIFO selects to replace a second chance

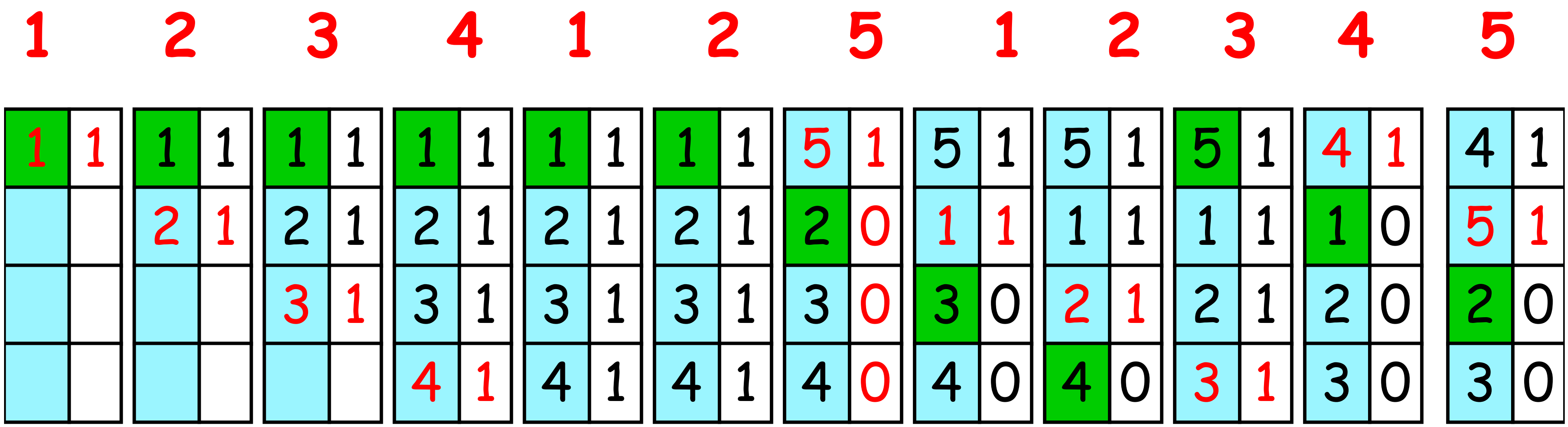
Clock algorithm implementation

- OS circulates through pages, clearing reference bits and finding a page with reference bit set to 0
- Keep pages in a circular list = clock
- Pointer to next victim = clock hand

A single step in Clock algorithm



Clock algorithm example



10 page faults

Advantage: simple to implemet!

Clock algorithm extension

- Problem of clock algorithm: does not differentiate dirty v.s. clean pages

- Dirty page: pages that have been modified and need to be written back to disk
  - More expensive to replace dirty pages than clean pages
  - One extra disk write (5 ms)

Clock algorithm extension (cont.)

- Use dirty bit to give preference to dirty pages
- On page reference
  - Read: hardware sets reference bit
  - Write: hardware sets dirty bit
- Page replacement
  - reference = 0, dirty = 0 → **victim page**
  - reference = 0, dirty = 1 → **skip** (don't change)
  - reference = 1, dirty = 0 → reference = 0, dirty = 0
  - reference = 1, dirty = 1 → reference = 0, dirty = 1
  - advance hand, repeat
  - If no victim page found, run swap daemon to flush **unreferenced dirty pages** to the disk, repeat

Summary of page replacement algorithms

- Optimal: throw out page that won't be used for longest time in future
  - Best algorithm if we can predict future
  - Good for comparison, but not practical
- Random: throw out a random page
  - Easy to implement
  - Works surprisingly well. Why? Avoid worst case
  - Random
- FIFO: throw out page that was loaded in first
  - Easy to implement
  - Fair: all pages receive equal residency
  - Ignore access pattern
- LRU: throw out page that hasn't been used in longest time
  - Past predicts future
  - With locality: approximates Optimal
  - Simple approximate LRU algorithms exist (Clock)

Current trends in memory management

- Less critical now
  - Personal computer v.s. time-sharing machines
  - Memory is cheap ➔ Larger physical memory
- Virtual to physical translation is still useful
  - "All problems in computer science can be solved using another level of indirection" David Wheeler
- Larger page sizes (even multiple page sizes)
  - Better TLB coverage
  - Smaller page tables, less page to manage
  - Internal fragmentation
- Larger virtual address space
  - 64-bit address space
  - Sparse address spaces
- File I/O using the virtual memory system
  - Memory mapped I/O: mmap()