

CS5460: Operating Systems

Lecture 12: Swapping & Advanced Paging Tricks

(Chapters 20, 21, 22)

Slide Credit: Andrea Arpaci-Dusseau

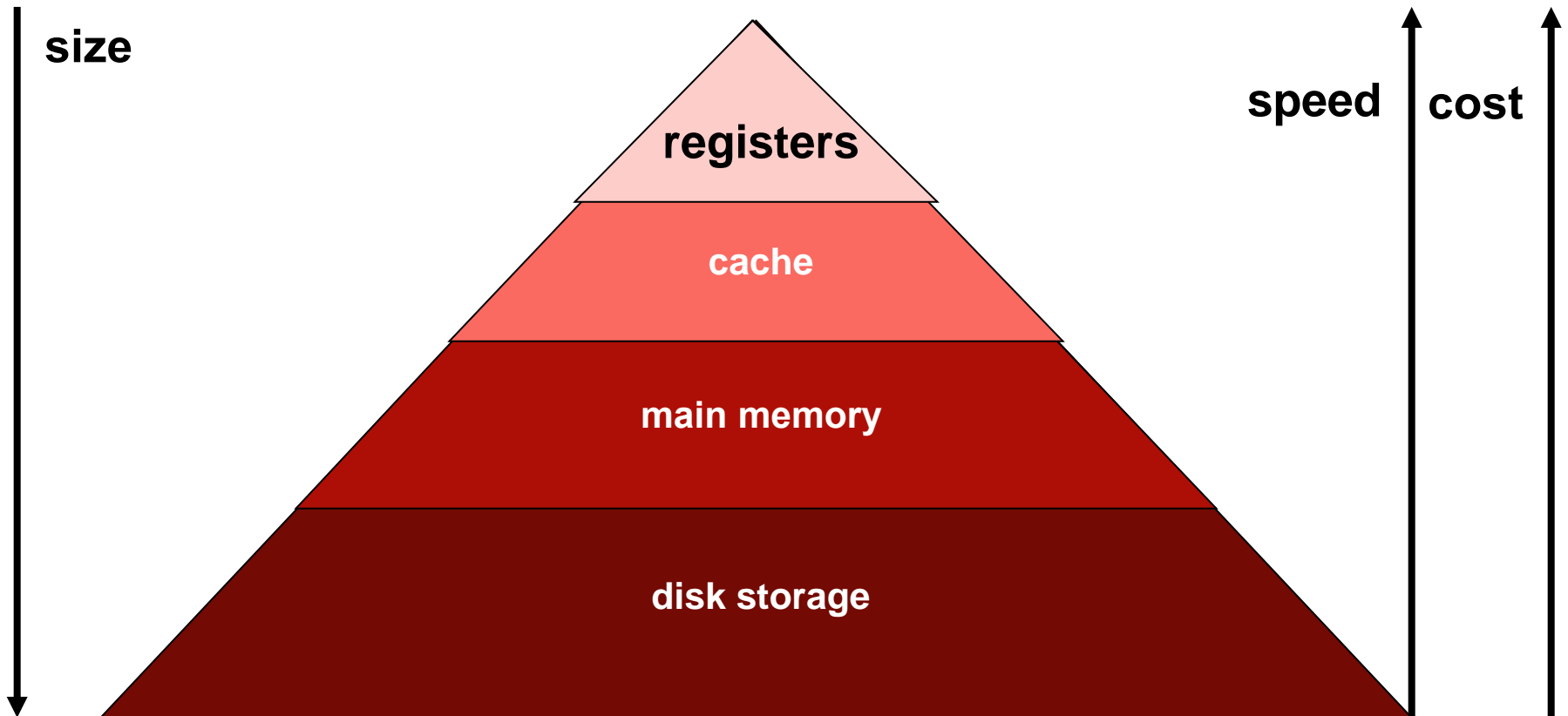
Assignments

- Assignment 3
 - xv6 Lottery Scheduler
 - Similar to getticks() but many more components
 - Due Thu Mar 18
 - **Note Thu deadline (since the exam is Tue Mar 16)**

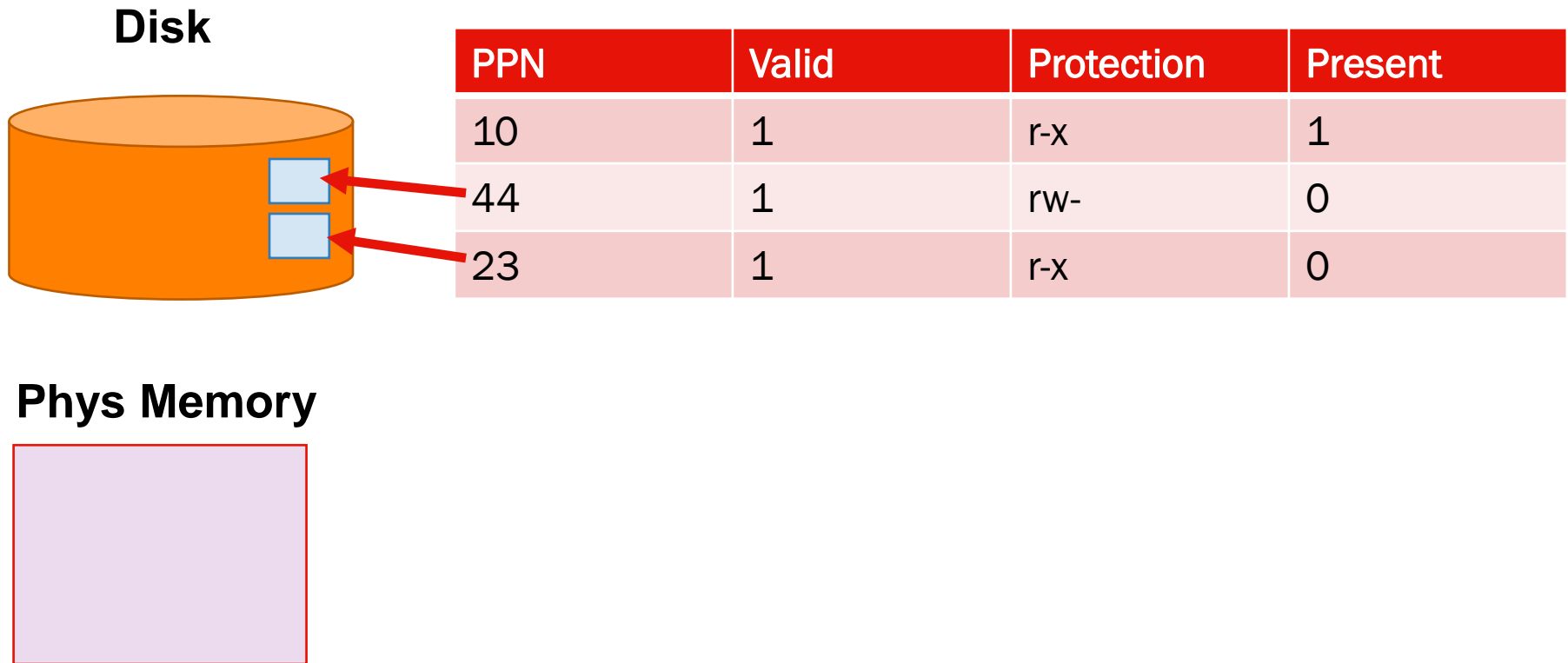
Memory Hierarchy

Leverage **memory hierarchy** of machine architecture

Each layer acts as “backing store” for layer above

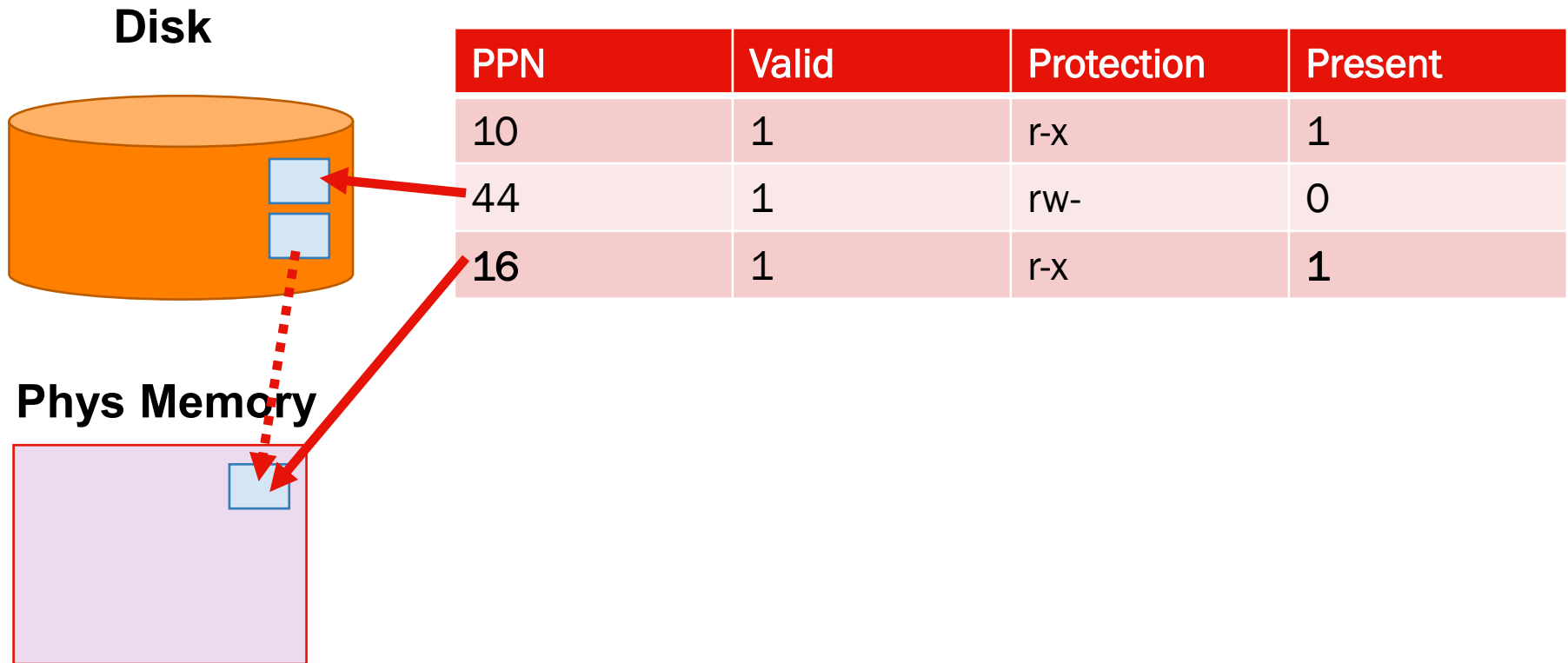


Present Bit



On process access to VPN 2 trap to OS (no present)
OS finds loads disk block 23 into a page frame
Swaps the PPN and then sets valid

Present Bit



On process access to VPN 2 trap to OS (no present)
OS finds loads disk block 23 into a page frame (at PPN 16)
Swaps the PPN and then sets valid

Virtual Memory Mechanisms

Hardware and OS cooperate to translate addresses

First, hardware checks TLB for virtual address

- if TLB hit, address translation is done; page is in physical memory

If **TLB miss**...

- Hardware or OS walk page tables
- If page is present, then page is in physical memory

If **page fault** (i.e., present bit is cleared)

- Trap into OS (not handled by hardware)
- OS selects victim page in memory to replace
 - Write victim page out to disk if modified (dirty bit in PTE)
- OS reads referenced page from disk into memory
- Page table is updated, present bit is set
- Process continues execution

What should scheduler do?

Virtual Memory Policies

Goal: Minimize number of page faults

- Page faults require milliseconds to handle (disk read)
- Implication: lots of time for OS to make good decision

OS has two decisions

- **Page selection**
 - When should a page on disk be **brought into** memory?
- **Page replacement**
 - Which in-memory page should be **thrown out** to disk?

Page Selection

- When should a page be brought from disk into memory?
- **Demand paging**: load page only when page fault occurs
 - Intuition: Wait until page must absolutely be in memory
 - When process starts: No pages are loaded in memory
 - Problems: Pay cost of page fault for every newly accessed page
- **Anticipatory/prefetching**: load page before referenced
 - OS predicts future accesses and brings pages into memory early
 - Works well for some access patterns (e.g., sequential)
- **Hints**: allow user-supplied hints about page references
 - User specifies: may need page in future, don't need this page anymore, or sequential access pattern, ...
 - Example: `madvise()` in Unix

Page Replacement

Which page in main memory should be selected as victim?

Write out victim page to disk if modified (dirty bit set)

If victim page is not modified (clean), just discard

OPT: replace page not used for longest time in future

+ guaranteed to minimize number of page faults

- OS must predict the future; not practical, but good for comparison

FIFO: replace page that has been in memory the longest

Intuition: first referenced long time ago, done with it now

+ fair, all pages get equal residency; easy (keep queue)

- some pages may always be needed

LRU: least-recently-used: replace page not used for longest time in past

Intuition: past predicts the future

+ with locality, LRU approximates OPT

- must track/order on time of each page access; some pathologies

Page Replacement Example

Page reference string: ABCABDADBCB

Metric:

Miss count

Three pages
of physical memory

	OPT	FIFO	LRU
ABC	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>
A	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>
B	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>
D	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>
A	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>
D	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>
B	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>
C	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>
B	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>	<div></div> <div></div> <div></div>

Page Replacement Comparison

Add more memory, what happens to performance?

- LRU, OPT: add memory, guaranteed to have fewer (or same number of) page faults
 - Smaller memory sizes are guaranteed to contain a subset of larger memory sizes
 - Stack property: smaller cache always subset of bigger
- FIFO: add memory, *usually* fewer page faults
 - Belady's anomaly: May actually have **more** page faults! 🤔

FIFO Performance may Decrease!

Consider access stream: ABCDABEABCDE

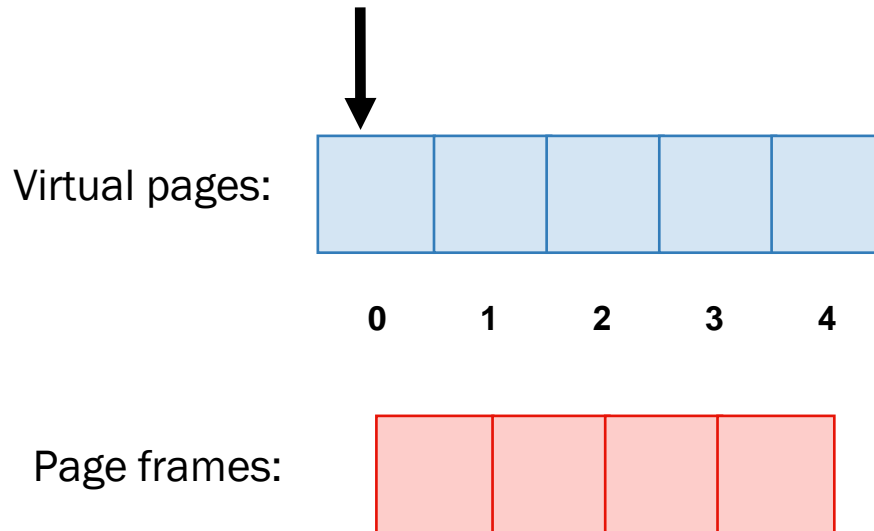
Consider physical memory size: 3 pages vs. 4 pages

How many misses with FIFO?

Problems w/LRU Replacement

- LRU does not consider frequency of accesses
 - Page accessed once in the past equal to one accessed N times?
 - Common workload problem:
 - One sequential scan of large region flushes memory
- Solution: Track frequency of accesses to page
- Pure LFU (Least-frequently-used) replacement
 - Problem: LFU can never forget pages from the far past
- Examples of other more sophisticated algorithms:
 - LRU-K and 2Q: Combines recency and frequency attributes
 - Expensive to implement, LRU-2 used in databases

LRU Troubles



Workload repeatedly accesses n pages in order,
but only $(n-1)$ page frames

Hitrate?

Sometimes random is better than “smarter” policy

Implementing LRU

Software Perfect LRU

- OS maintains ordered list of physical pages by reference time
- When page is referenced: move page to front of list
- When need victim: pick page at back of list
- Trade-off: slow on memory reference, fast on replacement

Hardware Perfect LRU

- Associate timestamp register with each page
- When page is referenced: store system clock in register
- When need victim: scan through registers to find oldest clock
- Trade-off: fast on memory reference, slow on replacement (especially as size of memory grows)

In practice, do not implement Perfect LRU

- LRU is an approximation anyway, so approximate more
- Goal: find an old page, but not necessarily the very oldest

Clock Algorithm

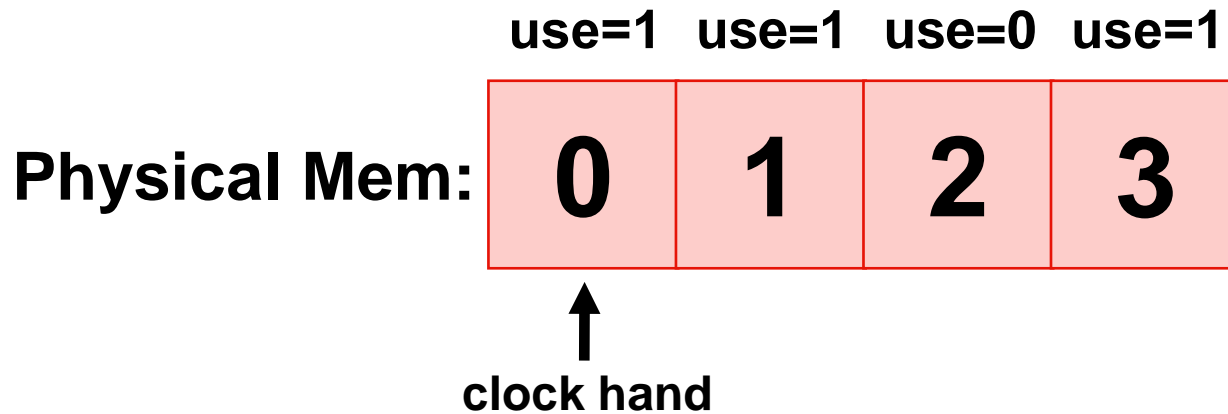
Hardware

- Keep **accessed** bit for each page frame (in page tables)
- When page is referenced: set **accessed** bit

Operating System

- Page replacement: look for page with **accessed** bit cleared (has not been referenced for awhile)
- Implementation:
 - Keep pointer to last examined page frame
 - Traverse pages in circular buffer
 - Clear **accessed** bits as search
 - Stop when find page with already cleared **accessed** bit, replace this page

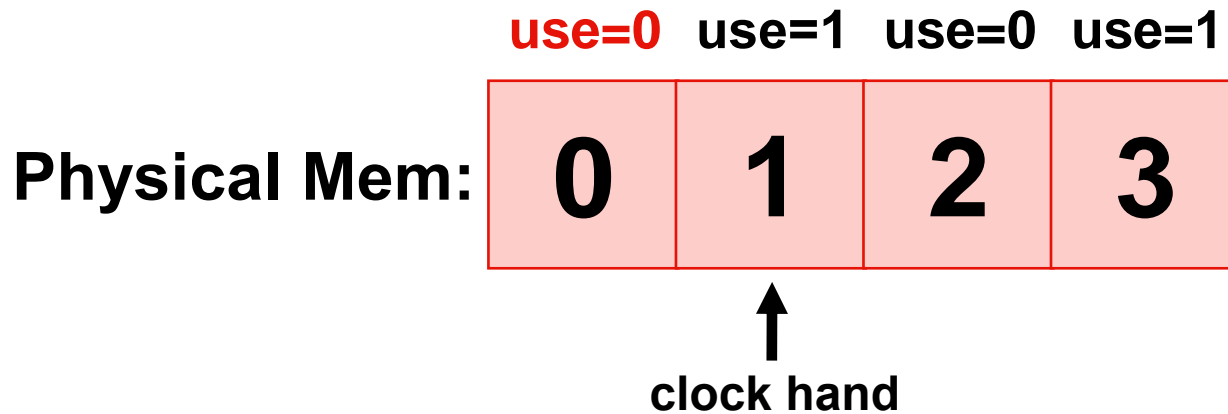
Clock: Look For a Page



Page Needing Page Frame



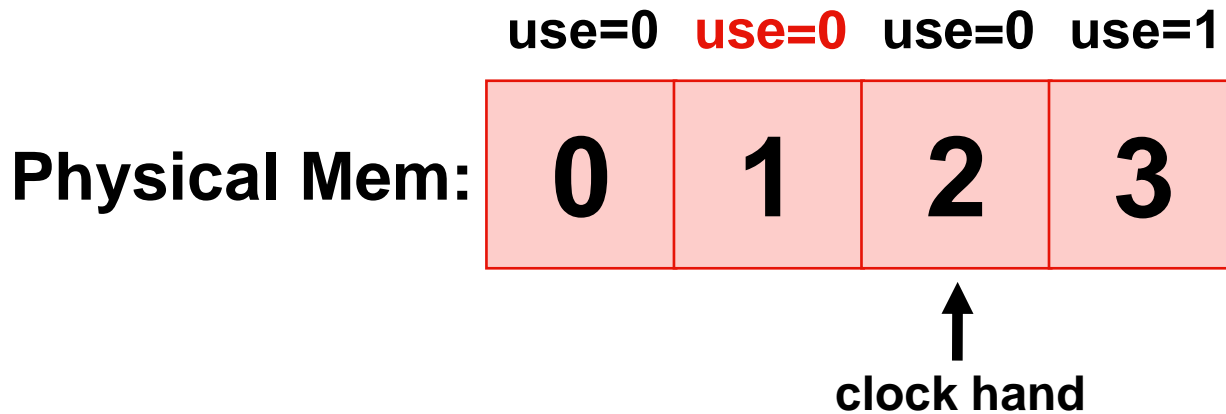
Clock: Look For a Page



Page Needing Page Frame



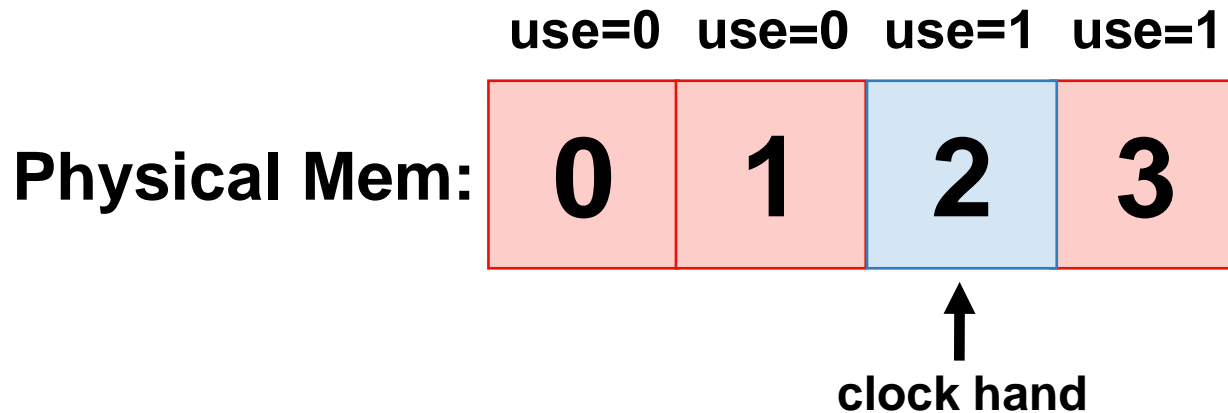
Clock: Look For a Page



Page Needing Page Frame



Clock: Look For a Page

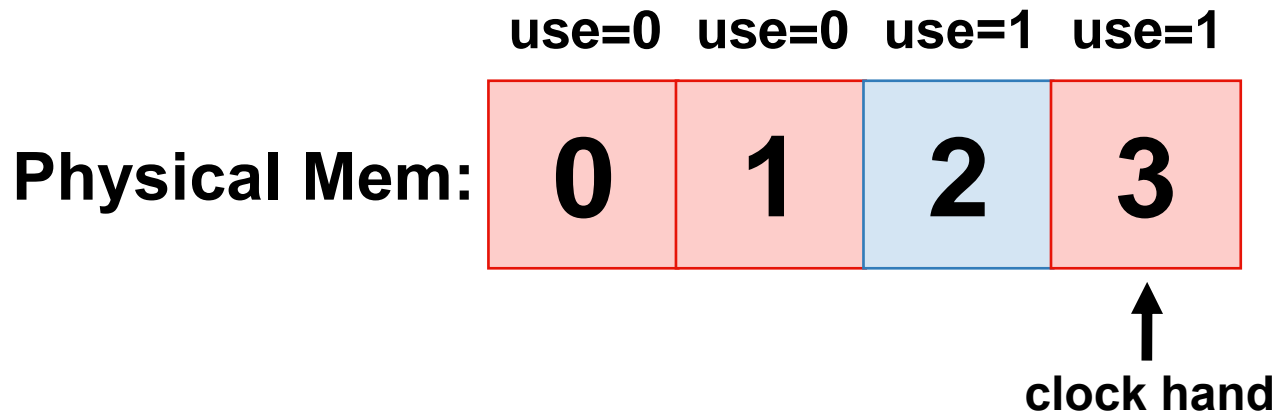


Page Needing Page Frame



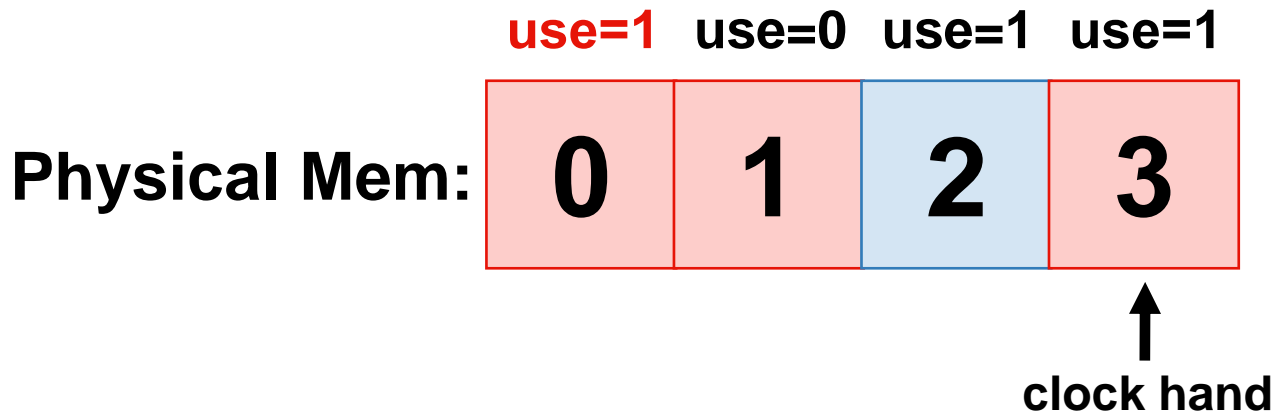
Evict page 2 because it has not been recently used

Clock: Look For a Page



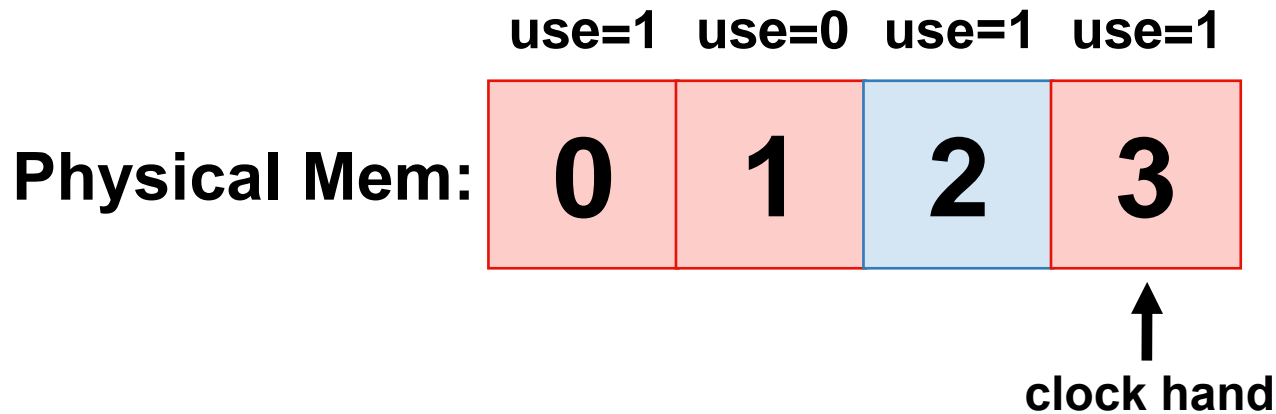
Advance hand so that frame 2 won't be reconsidered until clock hand loops back around

Clock: Look For a Page



page 0 is accessed...

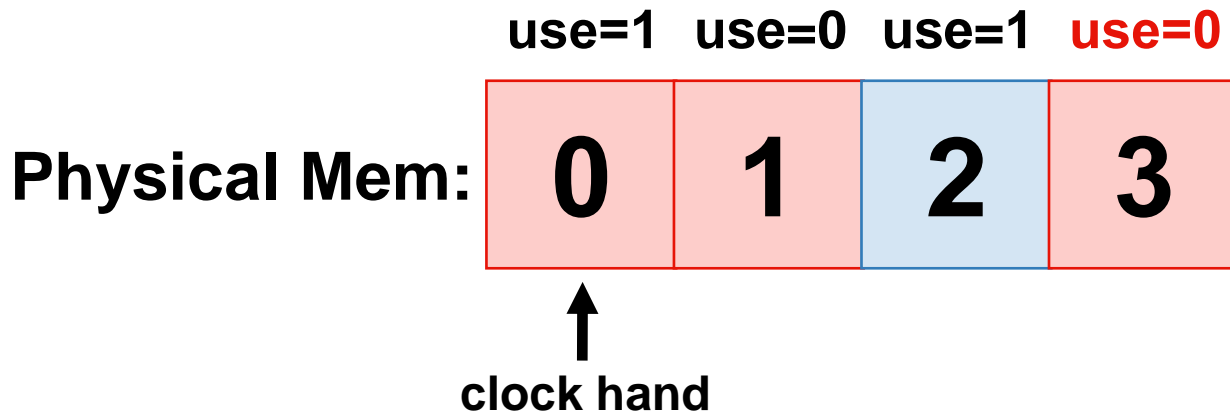
Clock: Look For a Page



Page Needing Page Frame



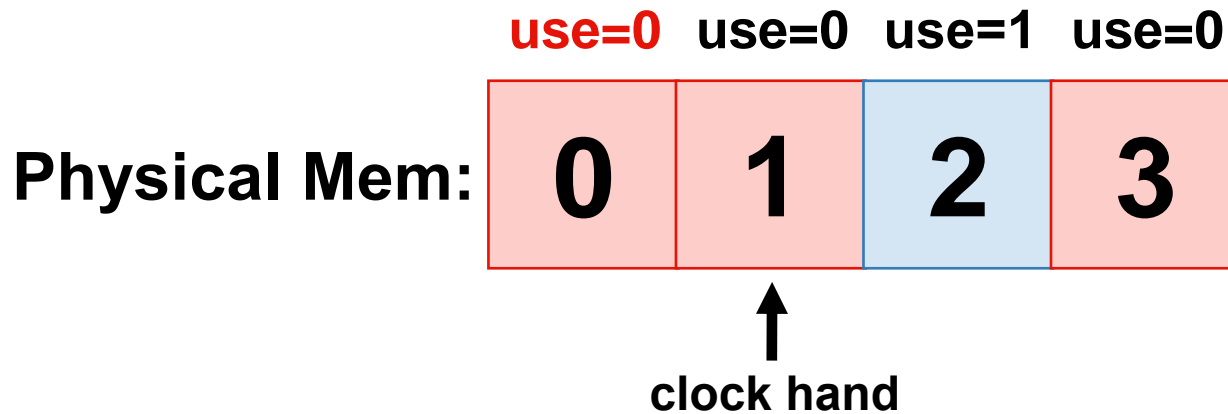
Clock: Look For a Page



Page Needing Page Frame



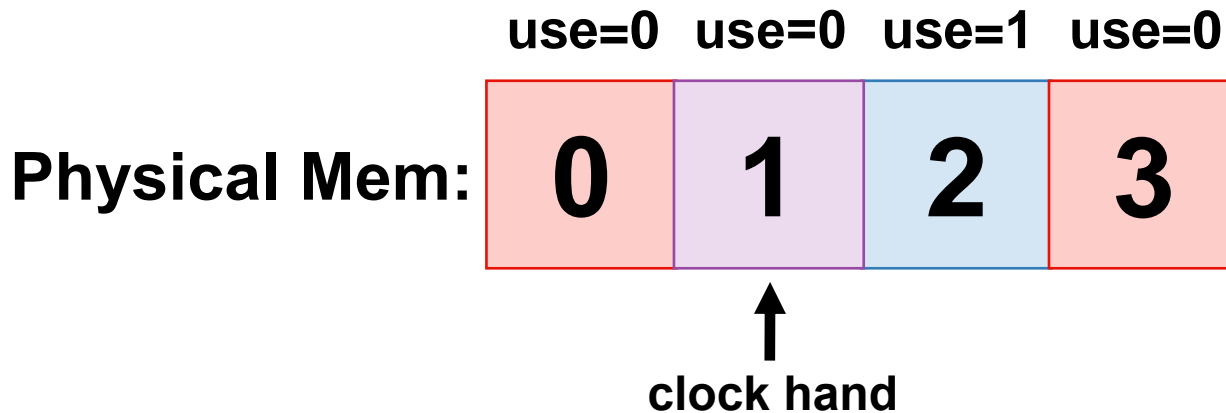
Clock: Look For a Page



Page Needing Page Frame



Clock: Look For a Page



Page Needing Page Frame



Evict page 1 because it has not been recently used,
store new page, advance hand to frame 2

Clock Extensions

Replace multiple pages at once

- Intuition:
Replacement algorithm and write single block to disk expensive
- Find multiple victims each time and track free list

Add software counter (“chance”)

- Intuition: Better ability to differentiate across pages (how much they are being accessed)
- Increment software counter if **accessed** bit is 0
- Replace when chance exceeds some specified limit

Use dirty bit to give preference to dirty pages

- Intuition: More expensive to replace dirty pages
 - Dirty pages must be written to disk, clean pages do not
- Replace pages that have **accessed** bit and **dirty** bit cleared

Thrashing

- **Working set:** collection of memory currently being used by a process
- If all working sets do not fit in memory → thrashing
 - One “hot” page replaces another
 - Percentage of accesses that generate page faults skyrockets
- Typical solution: “swap out” entire processes
 - Scheduler needs to get involved
 - Two-level scheduling policy → runnable vs memory-available
 - Need to be fair
 - Invoked when page fault rate exceeds some bound
- When swap devices are full, Linux invokes the “OOM killer”

Frame Allocation

- Who should we compete against for memory?
- **Global replacement:**
 - All pages for all processes come from single shared pool
 - Advantage: very flexible → can globally “optimize” memory usage
 - Disadvantages: thrashing more likely, can often do just the wrong thing (e.g., replace the pages of a process about to be scheduled)
 - Many OSes, including Linux, do this
- **Per-process replacement:**
 - Each process has private pool of pages → competes with itself
 - Alleviates inter-process problems, but not every process equal
 - Need to know working set size for each process
 - Windows has calls to set process’s min/max working set sizes

fork(), Copy-on-Write, & Laziness

- **Copy-on-write**: initially use shared pages for parent and child to share memory
 - On fork, child gets a copy of parent's page tables
 - (Re-)mark all pages read-only even if child/parent has write permissions
 - On write, trap, copy the page, record new location in page table, restart operation
- Parent/child share memory, unless one of them modifies memory contents after fork()
- Insight: much of parent/child address space remains unchanged after fork()
 - Saves space and work

Demand Zeroing

- Page frames cannot be reused directly
 - May contain sensitive data!
- OS zeroes pages before (re-)mapping them
- Can be lazy
 - Only zero a page frame when process accesses the memory
 - Even lazier: map same read-only zero page and use COW

mmap()

- System call to manipulate address space
- Map a file for demand paging
 - Can treat file as a big byte array
 - Other processes can map too to share state
- Map anonymous pages to add heap space
 - Can map regions larger than memory (how?)
 - Modern malloc() uses this instead of sbrk()
- Map pages that can be shared with children
 - On fork(), mappings copied without COW protection

What if no Hardware Support?

What can the OS do if hardware does not have
accessed bit (or dirty bit)?

- Can the OS “emulate” these bits?

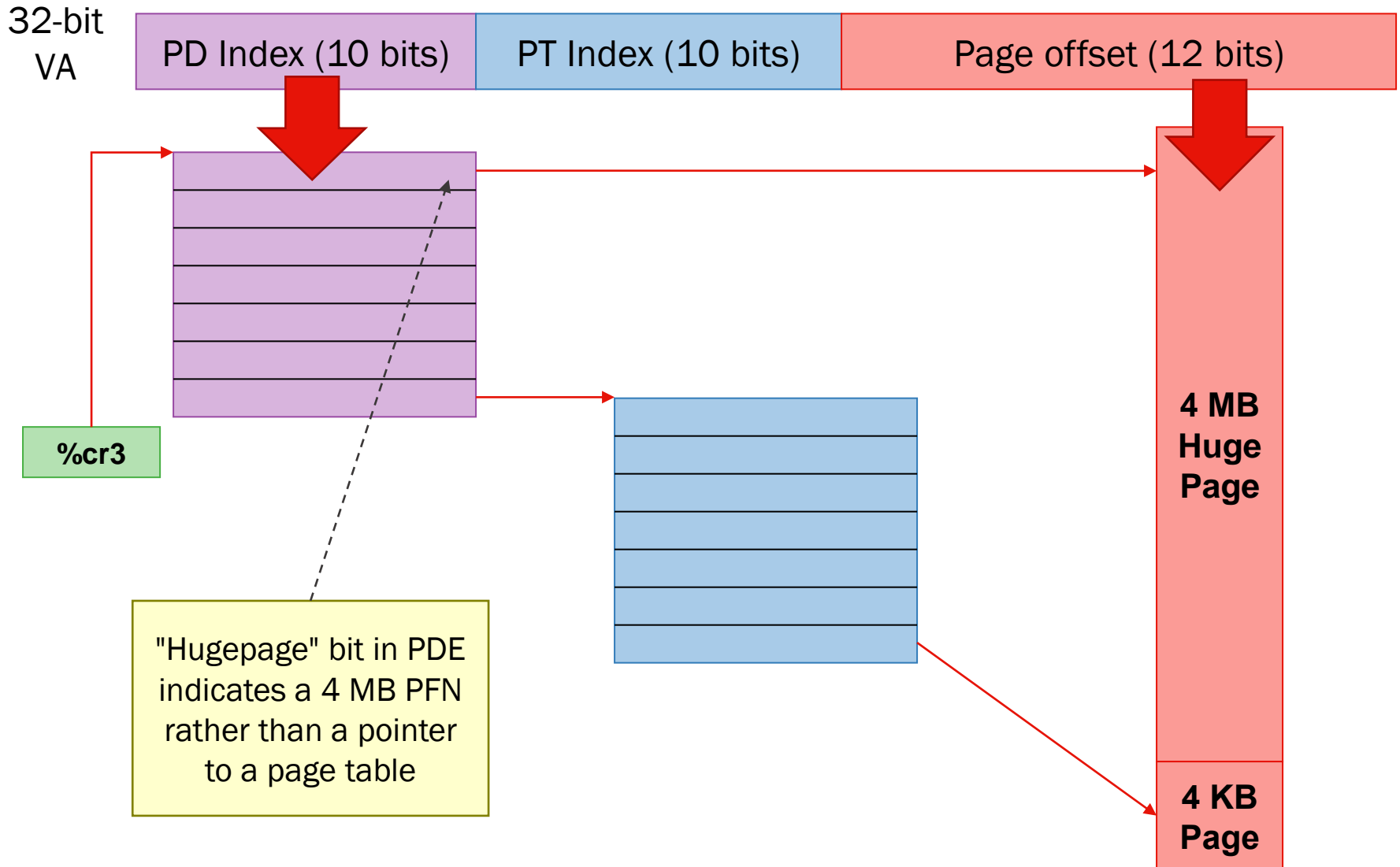
Leading question:

- How can the OS get control (i.e., generate a trap) every time accessed bit should be set? (i.e., when a page is accessed?)

Hugepages/Superpages

- **Problem:** TLB reach shrinking as % of memory size
- **Solution:** Hugepages
 - Permit (some) larger pages
 - For simplicity, restrict generality:
 - Same "coverage" as higher levels of multi-level page tables
 - Aligned to huge page size (e.g., 2 MB page aligned on 2 MB bdy)
 - Contiguous
- **Problem:** Restrictions limit applicability. How?

Example: Hugepage Usage



Hugepage Discussion

- What are good candidates for hugepages?
 - Kernel – or at least the portions of kernel that are not “paged”
 - Frame buffer
 - Large “wired” data structures
 - Scientific applications being run in “batch” mode
 - In-core databases
- How might OS exploit hugepages?
 - **Simple:** Few hardwired regions (e.g., kernel and frame buffer)
 - **Improved:** Provide system calls so applications can request it
 - **Holy grail:** OS watches page access behavior and determines which pages are “hot” enough to warrant hugepages
- Why might you **not** want to use hugepages?
- 32-bit Intel: 4 KB pages with 4 MB hugepages
- 64-bit Intel: 4 KB pages with 2 MB and 1 GB hugepages

Conclusions

Illusion of virtual memory:

Processes can run when sum of virtual address spaces is more than amount of physical memory

Mechanism:

- Use page table “present” bit
- OS handles page faults (or page misses) by reading in desired page from disk

Policy:

- Page selection – demand paging, prefetching, hints
- Page replacement – OPT, FIFO, LRU, others

Implementations (clock) perform approximation of LRU