CS5460: Operating Systems

Lecture 10: Multi-level Paging, Swapping & Advanced Paging Tricks

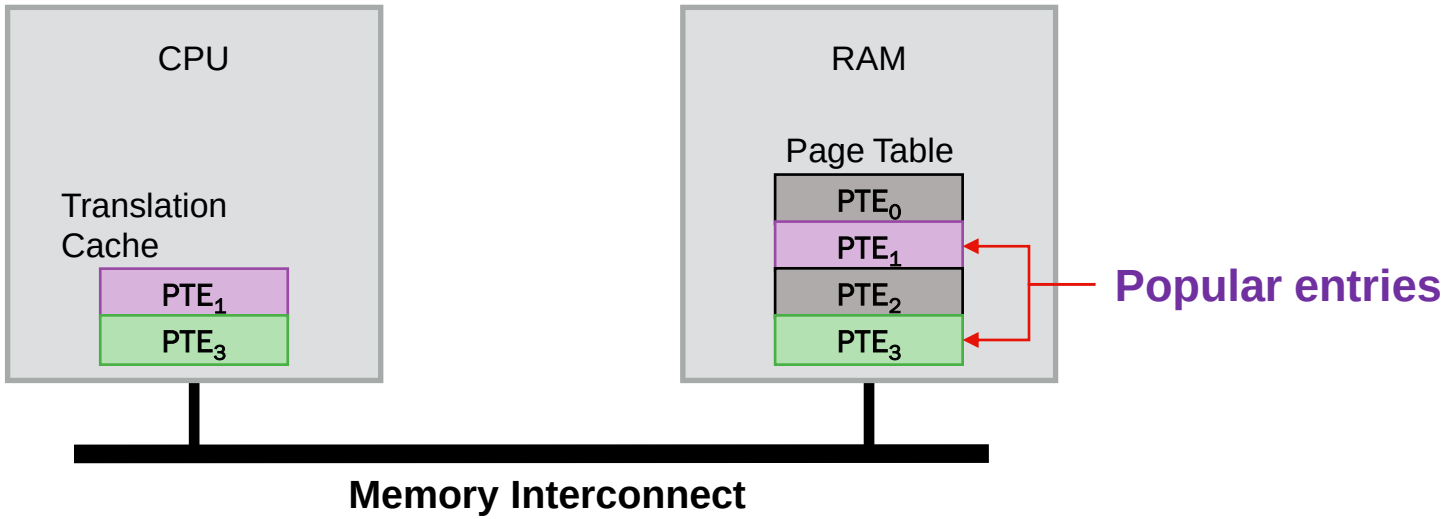
(Chapters 20, 21, 22)

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)**

Caching Page Translations



TLB: Translation Lookaside Buffer

Interposes on every memory access

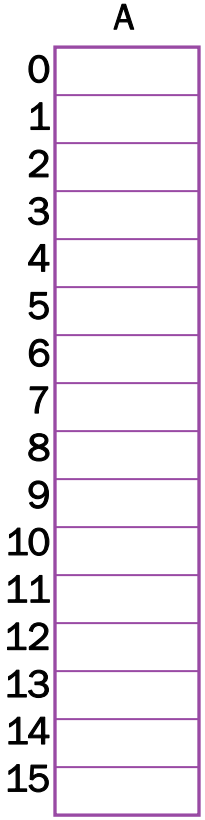
Caches PTEs, each describe VPN to PPN mapping

On access, if PTE in TLB skip page table, else look at page table

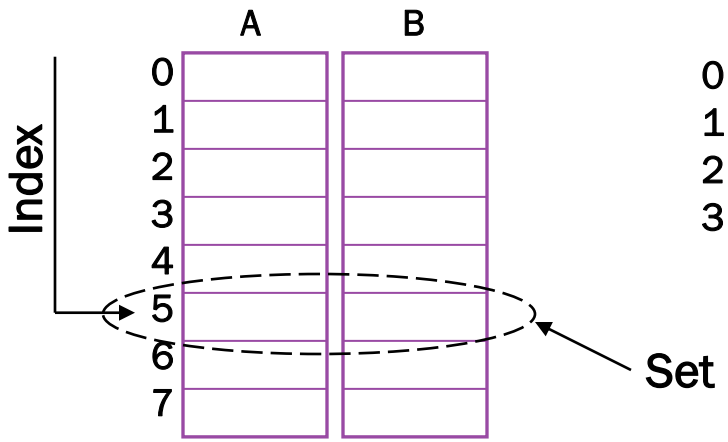
If page table entry present, then cache in TLB

TLB Organization

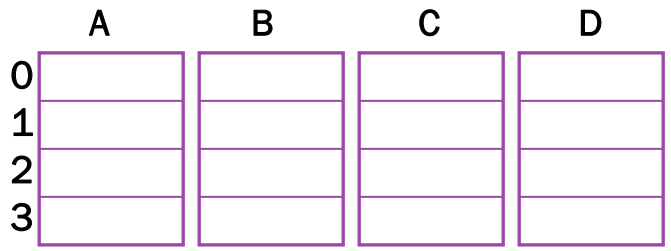
Various ways to organize a 16-entry TLB (artificially small)



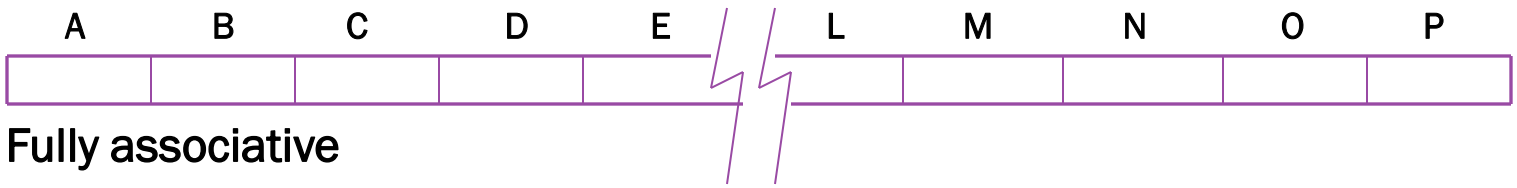
Direct mapped



Two-way set associative



Four-way set associative



Fully associative

Lookup

- Calculate set (tag % num_sets)
- Search for tag within resulting set

TLB Associativity Trade-offs

Higher associativity
- + Better utilization, fewer collisions
- – Slower
- – More hardware
- – Parallel search for all tags doesn't scale, so size of TLB is limited by propagation delay
  - – Generally need to know PA before CPU can use data in caches (3-5 ns)
  - – L2 and lower caches often physically indexed, physically tagged

Lower associativity
- + Fast
- + Simple, less hardware
- – Greater chance of collisions, lower TLB hit rate

TLBs used to be fully associative, but now multi-level TLBs:
32-entry 4-way L1 TLB, 1536-entry 12-way L2 TLB

Array Iterator (w/ TLB)

```
int sum = 0;
for (i = 0; i < 2048; i++){
        sum += a[i];
}
```

Assume following virtual address stream:
load 0x1000

load 0x1004

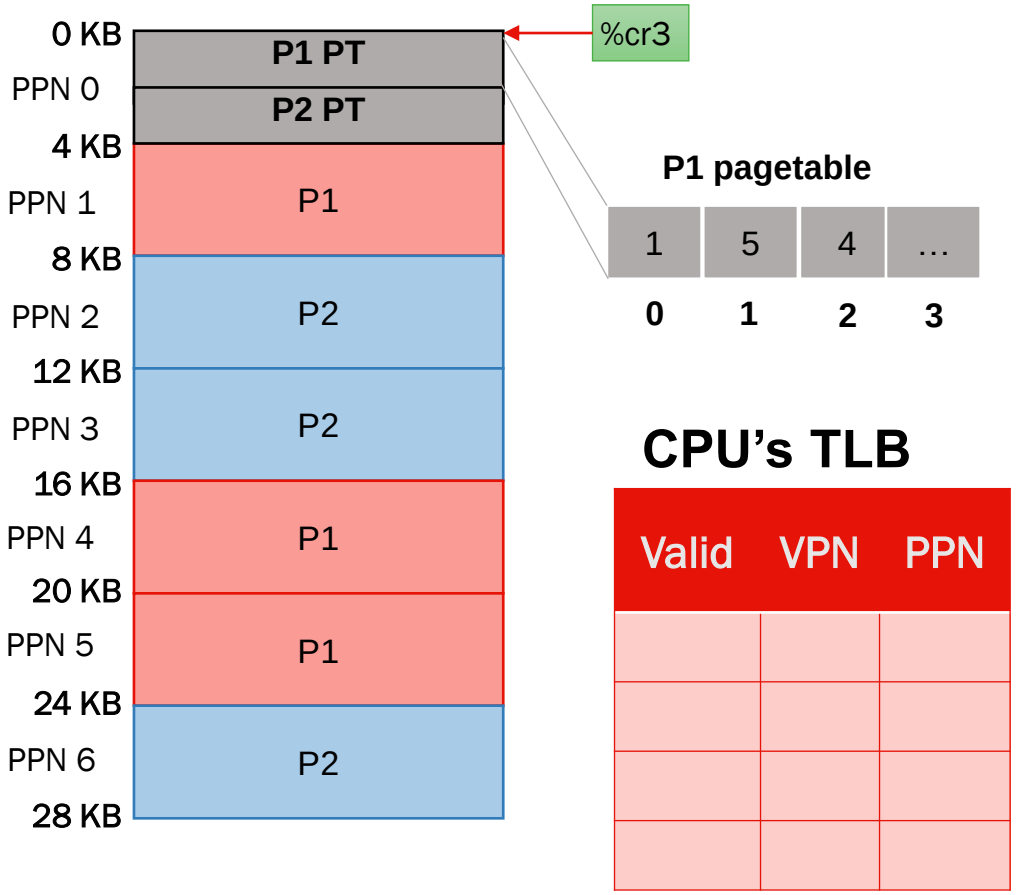
load 0x1008

load 0x100C

…

What will TLB behavior look like?

TLB Accesses: Sequential



VAs	PAs
load 0x1000	load 0x0004
	load 0x5000
load 0x1004	(TLB hit)
	load 0x5004
load 0x1008	(TLB hit)
	load 0x5008
load 0x100c	(TLB hit)
	load 0x500C
…	…
load 0x2000	load 0x0008
	load 0x4000
load 0x2004	(TLB hit)
	load 0x4004

TLB Performance

```
int sum = 0;
for (i=0; i<2048; i++) {
    sum += a[i];
}
```

Calculate miss rate of TLB for data:

TLB misses / # TLB lookups

TLB lookups?

= number of accesses to a = 2048

TLB misses?

= number of unique pages accessed
= 2048 / (elements of 'a' per 4K page)
= 2048 / (4096 / sizeof(int))
= 4096 / 1024
= 2

Miss rate?

2/2048 = 0.1%

Hit rate? (1 – miss rate)

99.9%

Hit rate better or worse with smaller pages?

TLB Performance

- How can system improve TLB performance (hit rate) given fixed number of TLB entries?

- Increase page size
  - Fewer unique page translations needed to access same amount of memory

- TLB “reach”:
  - Number of TLB entries * Page Size

TLB Performance

- Sequential array accesses almost always hit in TLB
  - Very fast!

- What access pattern will be slow?
  - Highly random, with no repeat accesses

Workload Access Patterns

Workload A

```
int sum = 0;

for (i=0; i<2048; i++) {
        sum += a[i];
}
```

Workload B

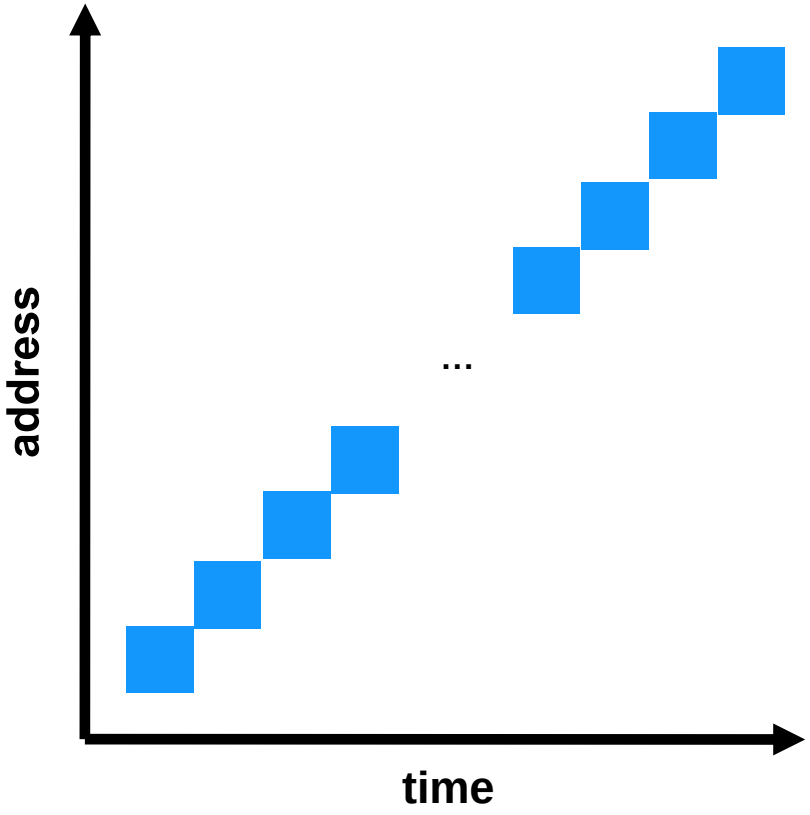
```
int sum = 0;

srand(1234);
for (i=0; i<1000; i++) {
    sum += a[rand() % N];
}

srand(1234);
for (i=0; i<1000; i++) {
    sum += a[rand() % N];
}
```

Does the TLB help
  Workload A?
  Workload B?
What does it depend on?

Workload Access Patterns

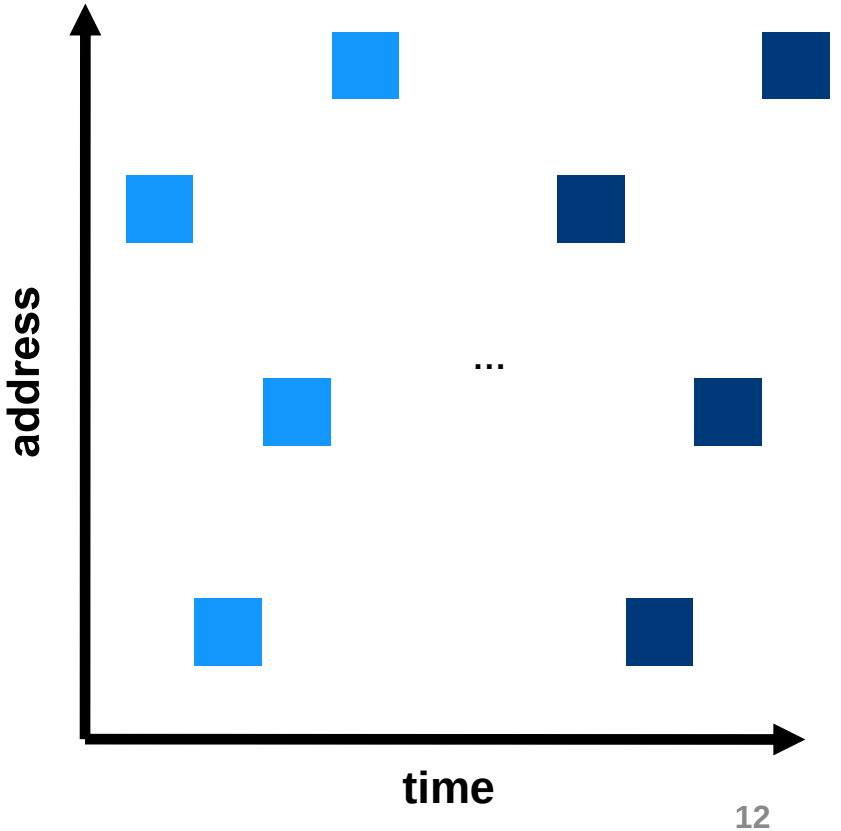
Workload A
Spatial Locality

Sequential Accesses



Workload B
Temporal Locality

Repeated Random Accesses



Workload Locality

Spatial Locality: future access will be to nearby addresses

Temporal Locality: future access repeats to the same data

What TLB characteristics are best for each type?

Spatial:

- Access same page repeatedly; need same VPN to PPN translation
- Same TLB entry reused

Temporal:

- Access same address near in future
- Same TLB entry reused in near future
- How near in future? How many TLB entries are there?

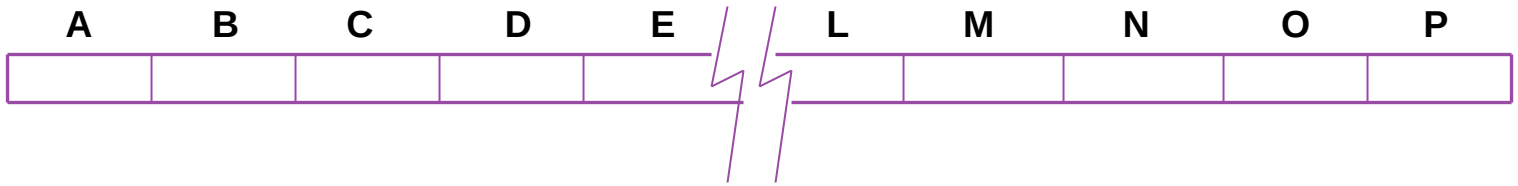
TLB Replacement Policies

LRU: evict Least-Recently Used TLB slot when needed

(More on LRU later in policies soon)

Random: Evict randomly choosen entry

Which is better?



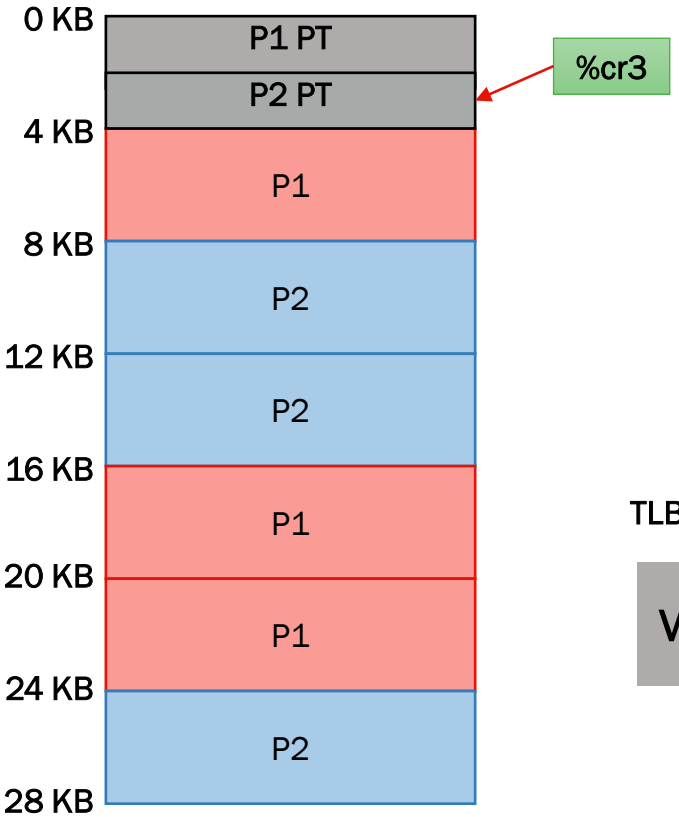
Context Switches

What happens if a process uses cached TLB entries from another process?

Solutions?

1. Flush TLB on each switch
   - Costly; lose all recently cached translations
2. Track which entries are for which process
   - Address Space Identifier (called PCIDs on Intel)
   - Tag each TLB entry with an 8-bit ASID
     - How many ASIDs do we get? (Intel has 4096)
     - Why not use PIDs?

TLB Example with ASID



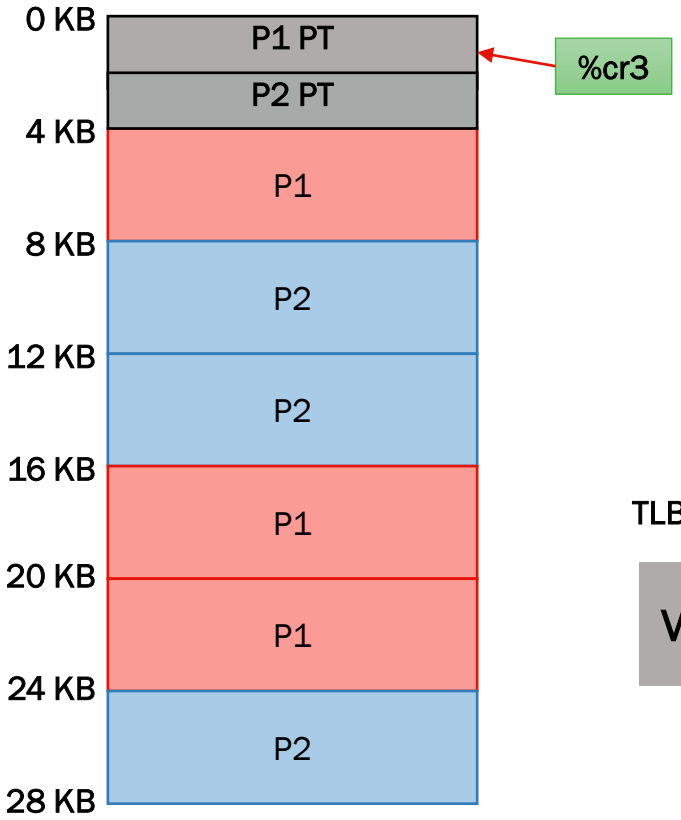
1	5	4	...	P1 pagetable (ASID 11)
6	2	3	...	P2 pagetable (ASID 12)

Virtual	Physical
load 0x1444 ASID: 12	load 0x2444

TLB:

Valid	Virt	Phys	ASID
1	1	2	12

TLB Example with ASID



1	5	4	...	P1 pagetable (ASID 11)
6	2	3	...	P2 pagetable (ASID 12)

Virtual		Physical
load 0x1444	ASID: 12	load 0x2444
load 0x1444	ASID: 11	load 0x5444

TLB:

Valid	Virt	Phys	ASID
1	1	2	12
1	1	5	11

No need to flush TLB on context switch; TLB hardware ensures cached entries from different processes don't interfere

TLB Performance

Context switches are expensive

Even with ASID, other processes "pollute" TLB

- Discard process A's TLB entries for process B's entries

Architectures can have multiple TLBs

- 1 TLB for data, 1 TLB for instructions
- 1 TLB for regular pages, 1 TLB for "super pages"

HW and OS Roles

Who handles TLB miss? **Hardware or OS?** Both have been used

If Hardware: CPU must know where pagetables are

- %cr3 register on x86
- Page table structure fixed and agreed upon between HW and OS
- HW "walks" the page table and fills TLB

If OS: CPU traps into OS upon TLB miss

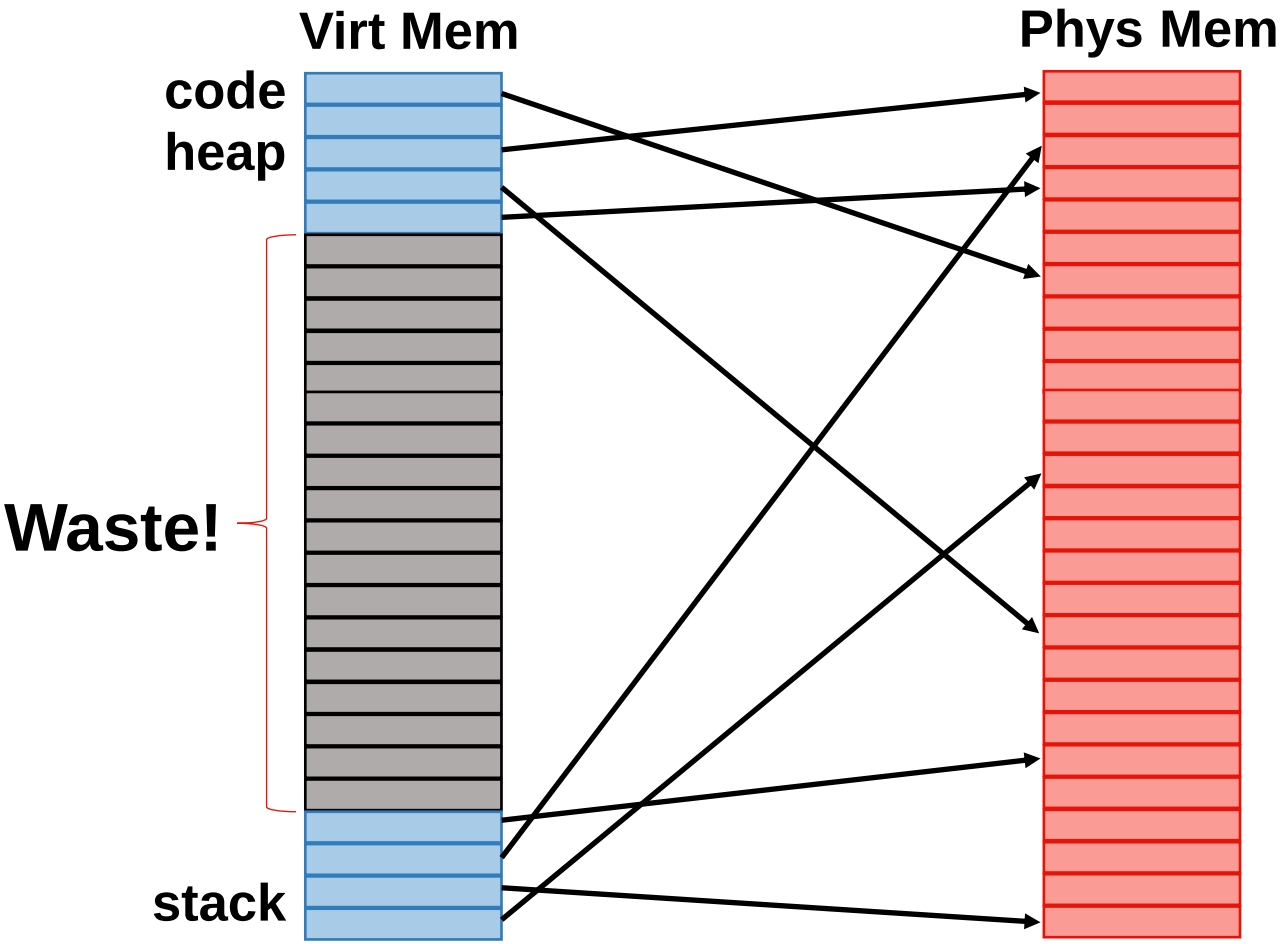
- "Software-managed TLB"
- OS interprets page tables as it chooses
- Modifying TLB entries is privileged
  - otherwise what could process do?

Need same protection bits in TLB as page table
(read/write/execute and kernel/user mode access)

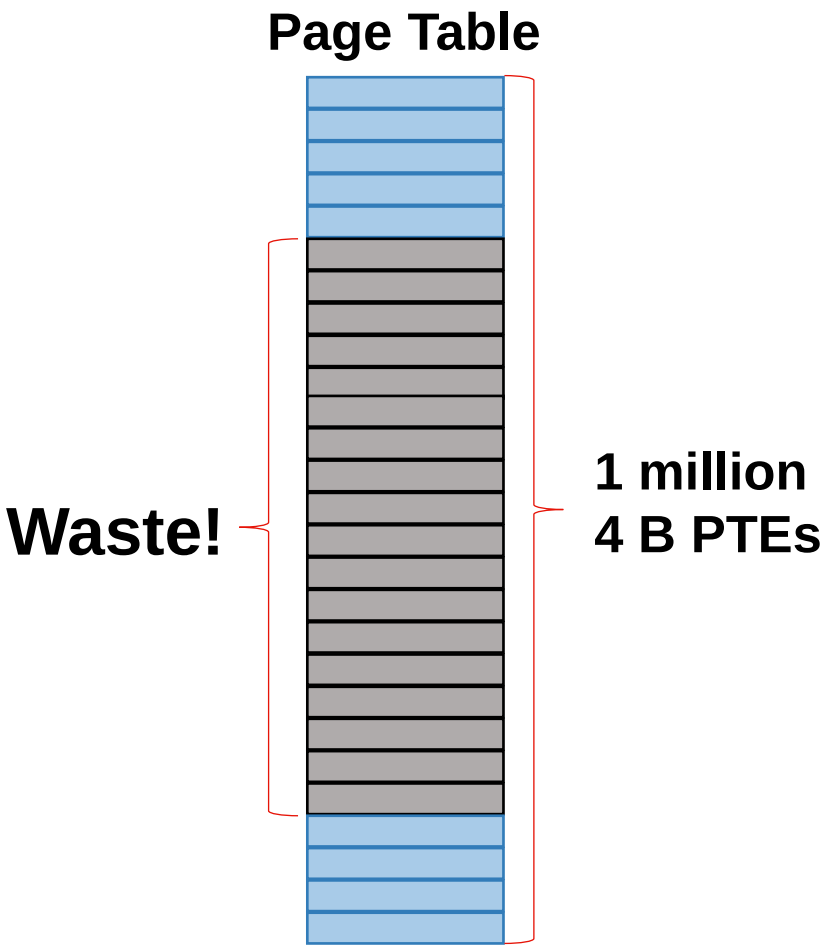
TLBs Summary

- Pages are great, but accessing page tables for every memory access is slow
- Cache recent page translations → TLB
  - Hardware performs TLB lookup on every memory access
- TLB performance depends strongly on workload
  - Sequential workloads perform well
  - Workloads with temporal locality can perform well
  - Increase TLB reach by increasing page size
- In different systems, hardware or OS handles TLB misses
- TLBs increase cost of context switches
  - Flush TLB on every context switch
  - Add ASID to every TLB entry

Big Tables due to Hole!



Most PTEs are Invalid (no PPN)



Avoid Simple Linear Page Table

- Use more complex page tables instead of just big array
- Any data structure is possible with software-managed TLB
  - Hardware looks for VPN in TLB on every memory access
  - If TLB does not contain VPN, TLB miss
    - Trap into OS and let OS find VPN to PPN translation
    - OS notifies TLB of VPN to PPN for future accesses

Other Approaches

1. Inverted Page Tables
2. Multi-level Page Tables
   - Page the page tables
   - Page the page tables of page tables…

Inverted Page Tables

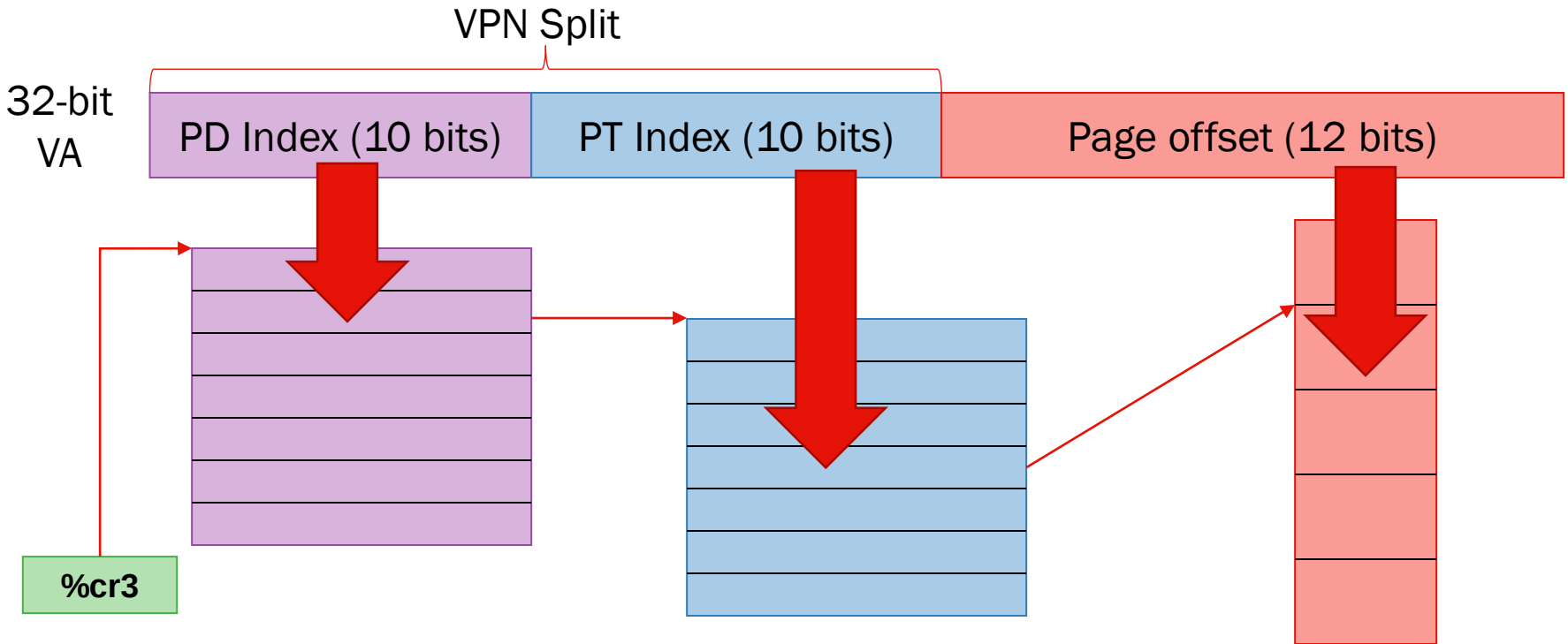
- Hash table indexed by VPN+ASID containing PPN
  - Only one table for whole system rather than per process
  - Only contains entries equal to allocated page frames (size is independent of number of VPNs)
- Complex data structure
  - Only done under software-controlled TLB
- On TLB miss
  - OS handles trap
  - Looks up hash(VPN+ASID) to find PPN
  - Populates TLB entry for VPN, ASID, PPN combination
  - Returns from trap
- For hw-controlled TLB, need well-defined, simple approach

Multi-level Page Tables

Goal: Let page tables be allocated non-contiguously

Idea: Page the page tables

- Creates multiple levels of page tables; outer level “page directory”
- Only allocate page tables for pages in use
- Used in x86 architectures (hardware can walk known structure)



Multi-level Paging Example

Page Directory

PPN	valid
0x3	1
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
0x92	1

Page Table @ PPN 0x3

PPN	valid
0x10	1
0x23	1
-	0
-	0
0x80	1
0x59	1
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0

Page Table @PPN 0x92

PPN	valid
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
0x55	1
0x45	1

translate 0x01ABC

translate 0x00000

translate 0xFEED0

20-bit address:

Address Format for MLP

30-bit address:

How should logical address be structured?

- How many bits for each paging level?

Goal?

- Each page table fits within a page
- PTE size * number PTE = page size
  - Assume PTE size = 4 bytes
  - Page size = 2^{12} bytes = 4KB
  - 2^2 bytes * number PTE = 2^{12} bytes
  - → number PTE per page = 2^{10}
- → # bits for selecting inner page = 10

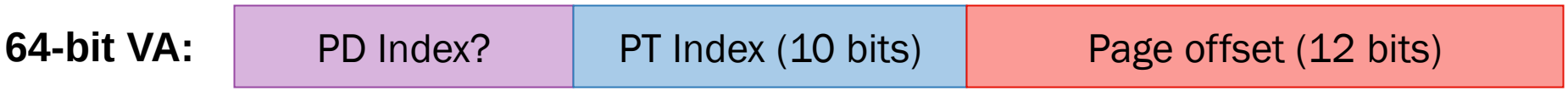
This is key!

Remaining bits for outer page:

- 30 – 10 – 12 = 8 bits

Problem with 2 Levels?

Problem: page directories (outer level) may not fit in a page



Solution:

- Split page directories into pieces
- Have a directory of page directories of page tables

How large is virtual address space with 4 KB pages, 4 byte PTEs, each page table fits in page given 1, 2, 3 levels?

4 KB / 4 bytes → 1024 entries per level
1 level: 1024 * 4 KB = 2^{22} bytes = 4 MB
2 levels: 1024 * 4 MB = 2^{32} bytes = 4 GB
3 levels: 1024 * 4 GB = 2^{42} bytes = 4 TB

Effective VA size?
3 levels enough for 64-bit VA?

Why are 4 B PTEs unlikely in this scenario?

Full System with TLBs

On TLB miss: lookups with more levels more expensive

How much does a miss cost?

Assume 3-level page table
Assume 256-byte pages
Assume 16-bit addresses
Assume ASID of current process is 211
How many physical accesses for each instruction?

(Ignore previous ops changing TLB)

ASID	VPN	PFN	Valid
211	0xbb	0x91	1
211	0xff	0x23	1
122	0x05	0x91	1
211	0x05	0x12	0

(a) 0xAA10: movl 0x1111, %edi

8 accesses

(b) 0xBB13: addl $0x3, %edi

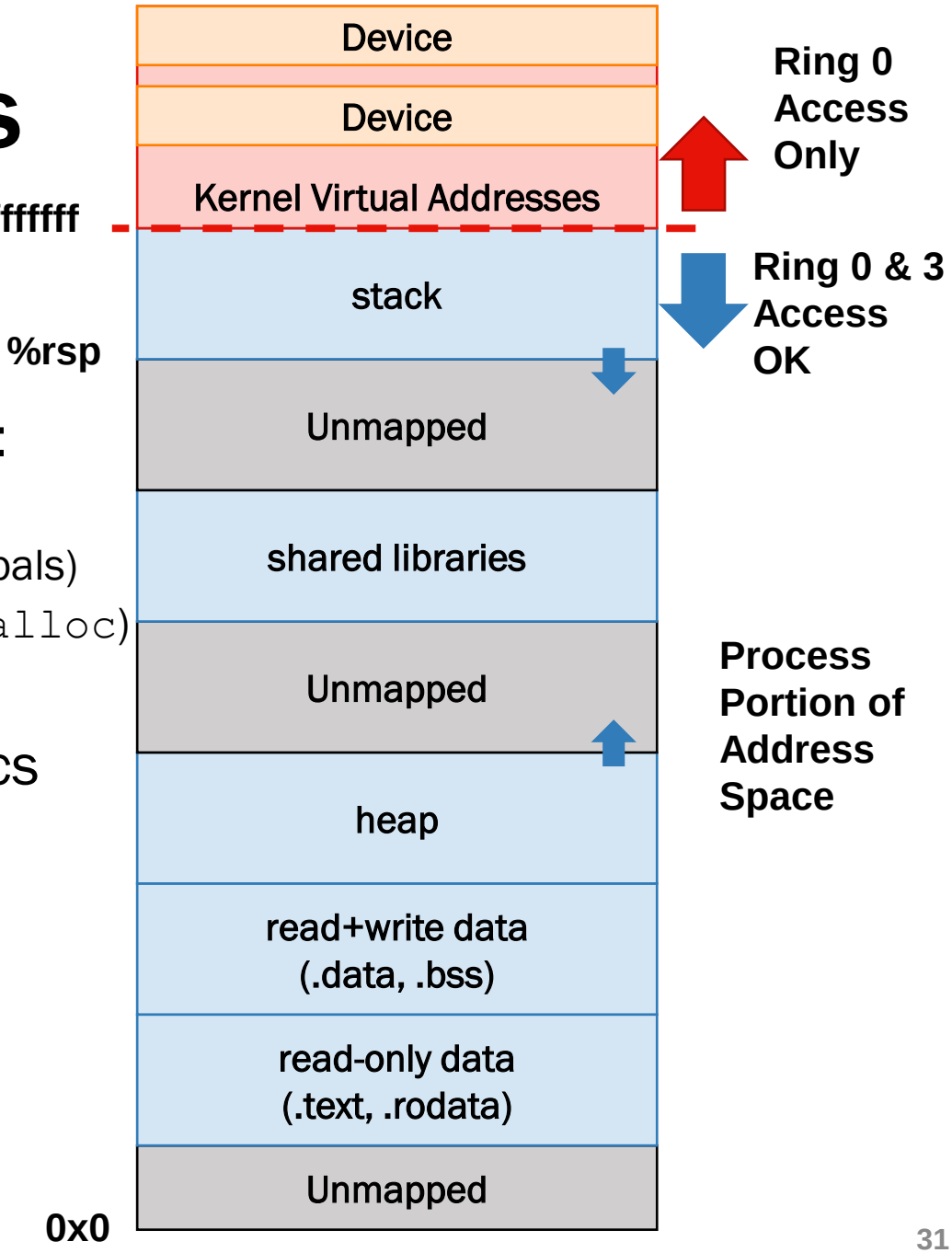
1 access

(c) 0x0519: movl %edi, 0xFF10

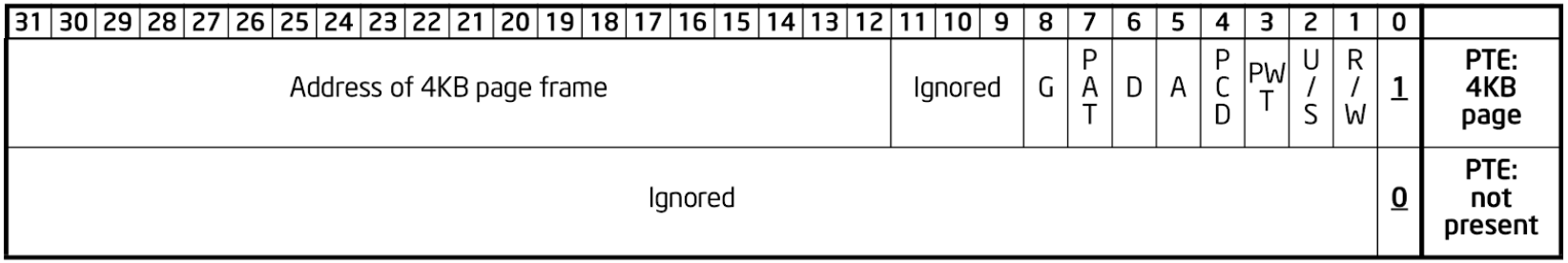
5 accesses

Virtual Address Space

- Parts of the address space:
  - Code: binary image of program
  - Data/BSS: Static variables (globals)
  - Heap: explicitly alloced data (`malloc`)
  - Stack: implicitly alloced data
- Kernel mapped into all procs
  - Until Meltdown
- CPU's MMU hardware:
  - Remaps VAs to PAs
  - Supports read-only, kernel-only
  - Detects accesses to unmapped regions



Intel 32-bit PTE Format



31–12	11–9	8	7	6	5	4	3	2	1	0	
Address of 4KB page frame	Ignored	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored										0	PTE: not present

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

- Important: PFN, Dirty (D), Accessed (A), User/Kernel (U/S), Writable (R/W), Present (Bit 0)
- Other stuff:
  - Global (G): PTE need not be flushed on TLB flush;
    - e.g. kernel PTEs which are the same for all processes
  - Caching Control (PAT, PCD, PWT) : Disable & Write-through
    - e.g. disable caching for addresses mapped to devices

Page Tables Summary

- Linear page tables require too much contiguous memory
  - Wasted space with invalid entries
  - And irregular (non-page) size create external fragmentation
- Many options for efficiently organizing page tables
- If OS traps on TLB miss, OS can use any data structure
  - Inverted page tables (hashing)
- If hardware handles TLB miss,
  page tables must follow hw format
  - Multi-level page tables used in x86 architecture
  - Each page table fits within a page
  - Page directory indexes over a process' page tables

Virtual Memory Motivation

OS goal: Support processes when not enough physical memory

- Single process with very large address space
- Multiple processes with combined address spaces

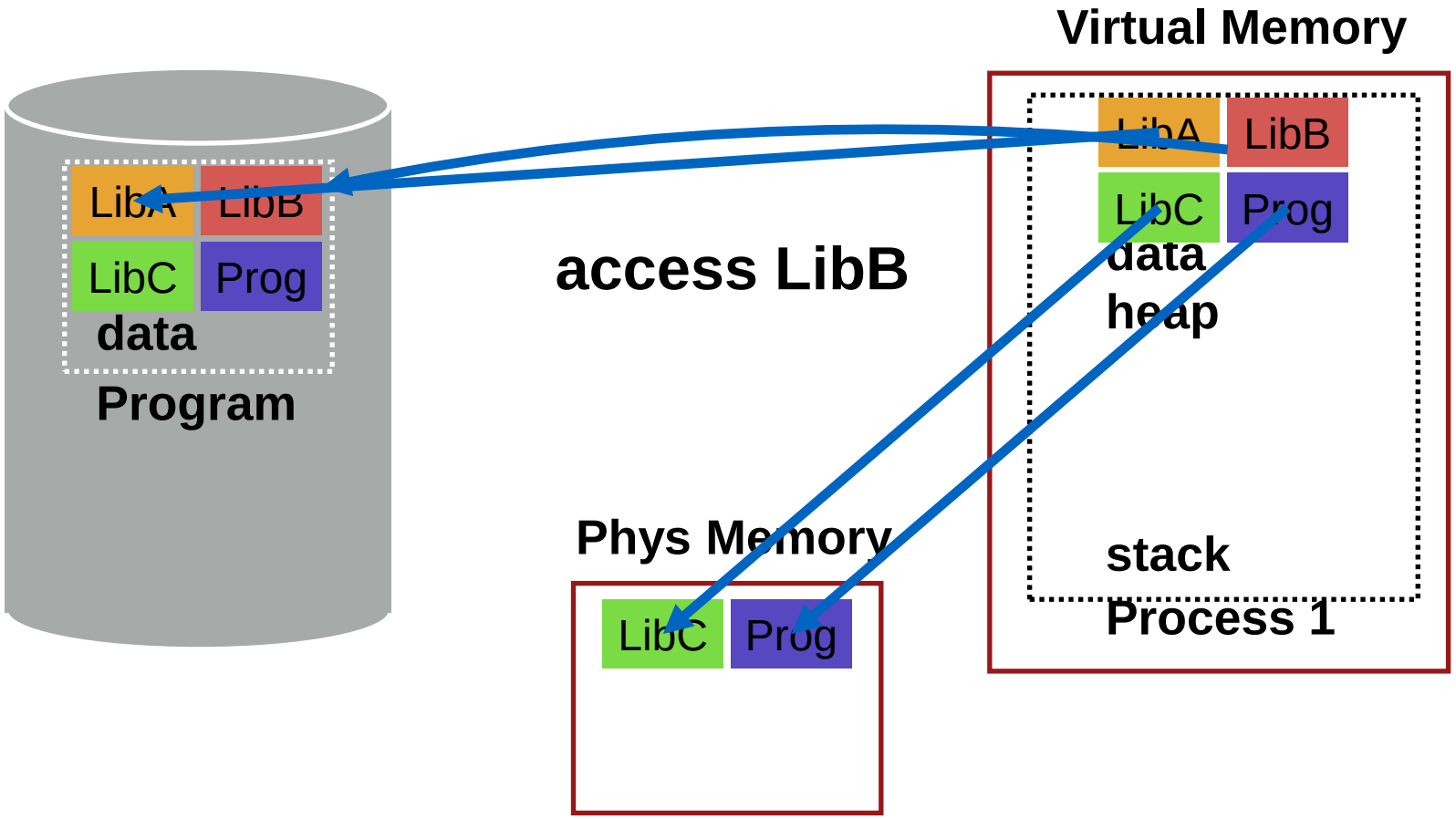
User code should be independent of amount of physical memory

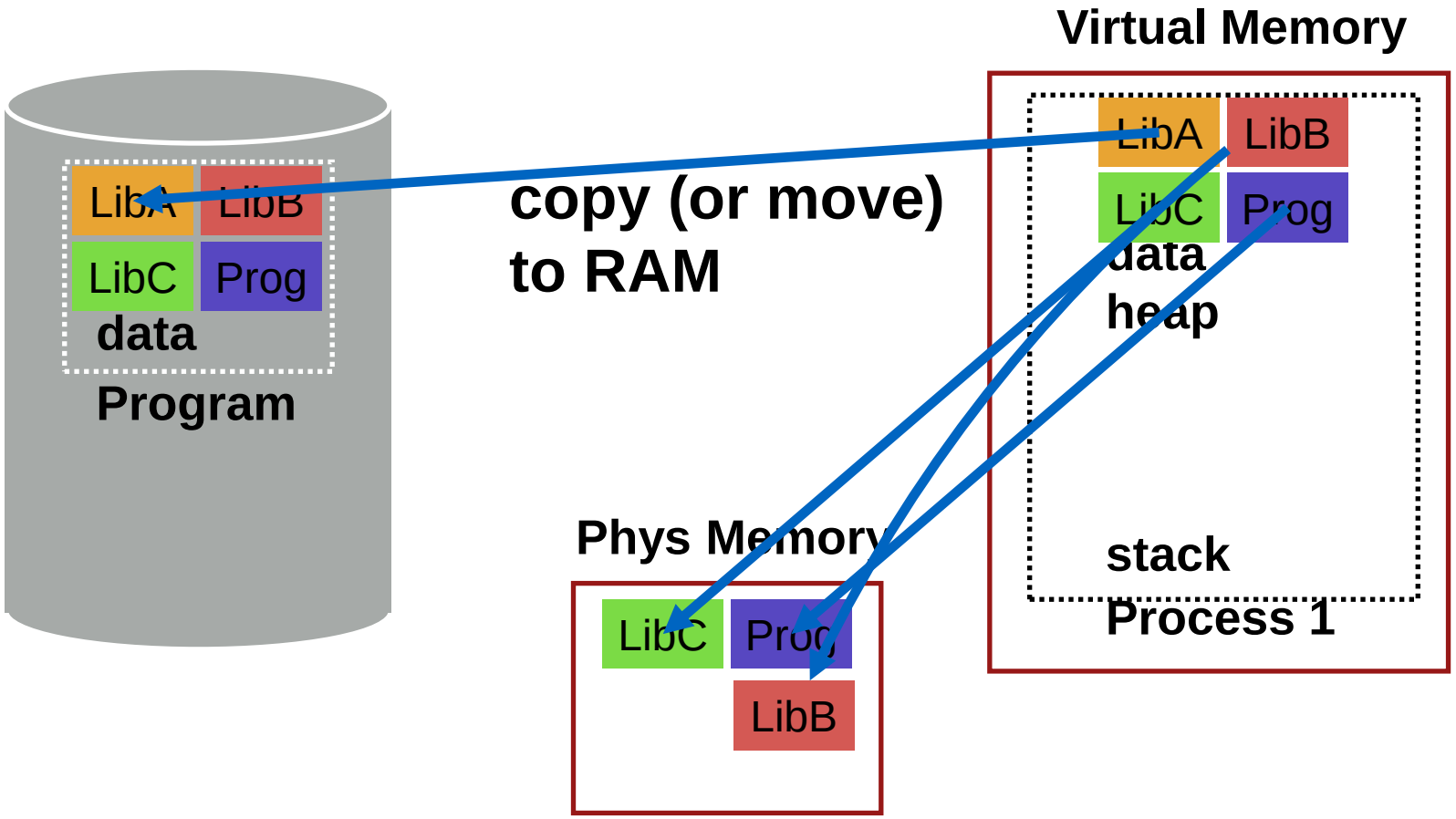
- Correctness, if not performance

Virtual memory: OS provides illusion of more physical memory

Why does this work?

- Relies on key properties of user processes (workload) and machine architecture (hardware)

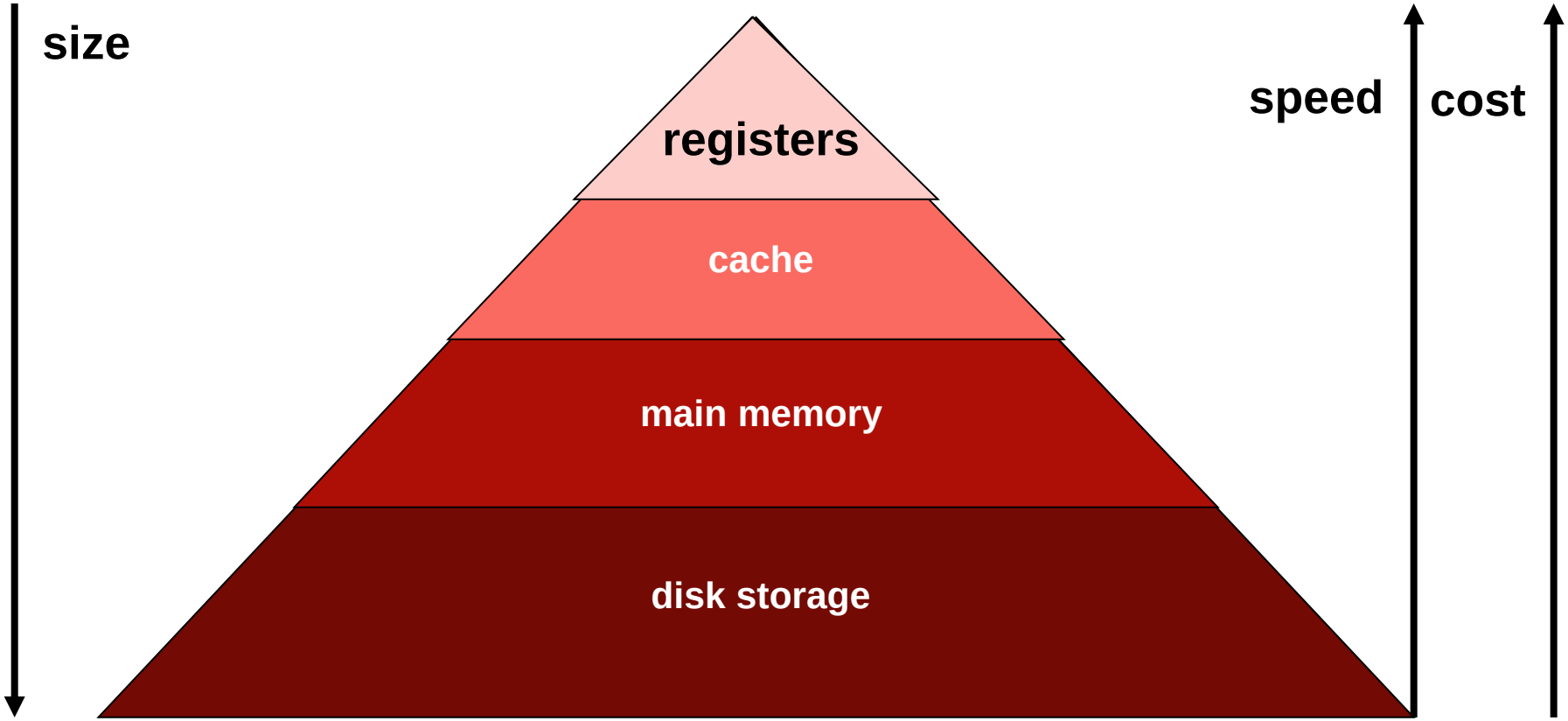

Virtual Memory
LibA
LibB
LibC
Prog
data
heap
stack
Process 1
access LibB
LibA
LibB
LibC
Prog
data
Program
Phys Memory
LibC
Prog


Virtual Memory
LibA
LibB
LibC
Prog
data
heap
stack
Process 1
copy (or move)
to RAM
LibA
LibB
LibC
Prog
data
Program
Phys Memory
LibC
Prog
LibB

Memory Hierarchy

Leverage memory hierarchy of machine architecture

Each layer acts as “backing store” for layer above



Virtual Memory Intuition

Idea: OS keeps unreferenced pages on disk

- Slower, cheaper backing store than memory

Process can run when not all pages are loaded into main memory

OS and hardware cooperate to provide illusion of large disk as fast as main memory

- Same behavior as if all of address space in main memory
- Hopefully have similar performance

Requirements:

- OS must have **mechanism** to identify location of each page in address space → in memory or on disk
- OS must have **policy** for determining which pages live in memory and which on disk

Virtual Address Space Mechanisms

Each page in virtual address space maps to one of three locations:

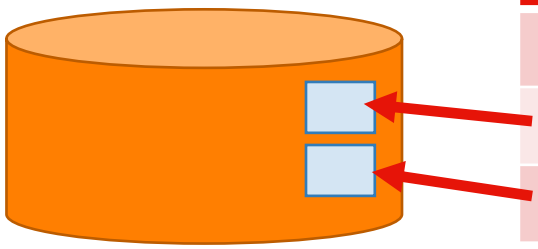
- Physical main memory: Small, fast, expensive
- Disk (backing store): Large, slow, cheap
- Nothing (error): Free

Use “present” bit in page tables

- permissions (r/w), valid, present
- Page in memory: present bit set in PTE
- Page on disk: present bit cleared
  - Causes trap into OS when page is referenced
  - OS tracks where on disk it can find this page
  - Instead of abort, fetch the page, change the PTE, and restart instruction

Present Bit

Disk

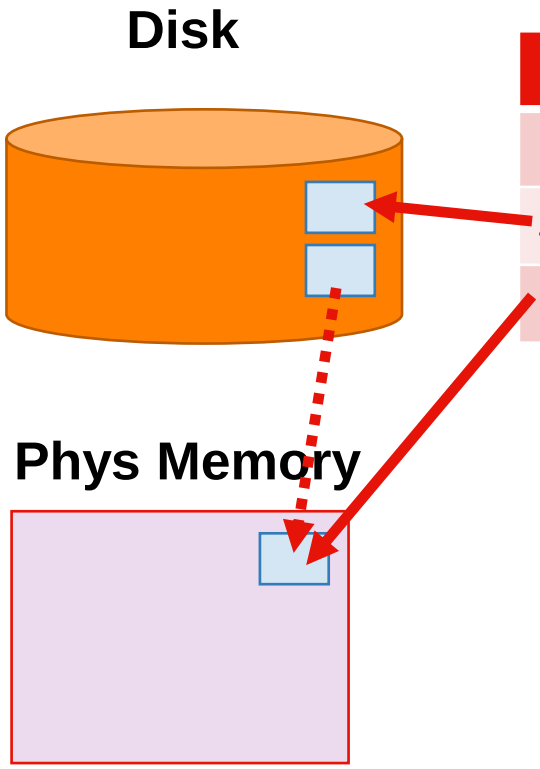


PPN	Valid	Protection	Present
10	1	r-x	1
44	1	rw-	0
23	1	r-x	0

Phys Memory

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



PPN	Valid	Protection	Present
10	1	r-x	1
44	1	rw-	0
16	1	r-x	1

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?

Mechanism for Continuing a Process

Continuing a process after a page fault is tricky

- Want page fault to be transparent to user
- Page fault may have occurred in middle of instruction
  - When instruction is being fetched
  - When data is being loaded or stored
- Requires hardware support
  - Precise interrupts: stop CPU pipeline so instructions before faulting instruction appear to have completed and those after appear to be un-started (so they can be restarted)

Complexity depends upon instruction set

- Can faulting instruction be restarted from beginning?
  - Must track side effects so hardware can undo
  - Intel x86 has string operations that are linear time!

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 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

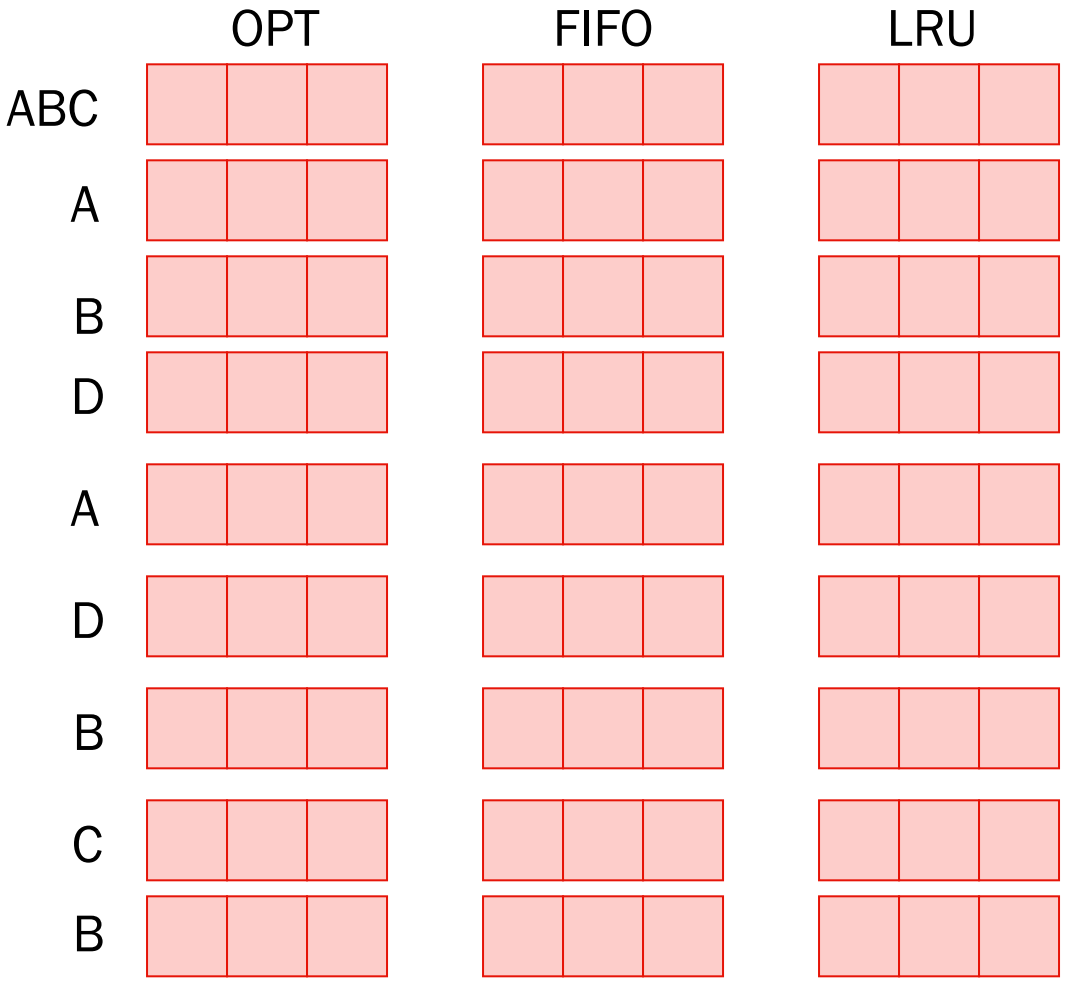
- must track/order on time of each page access; some pathologies

Page Replacement Example

Page reference string: ABCABDADBCB

Metric:
Miss count

Three pages
of physical memory



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

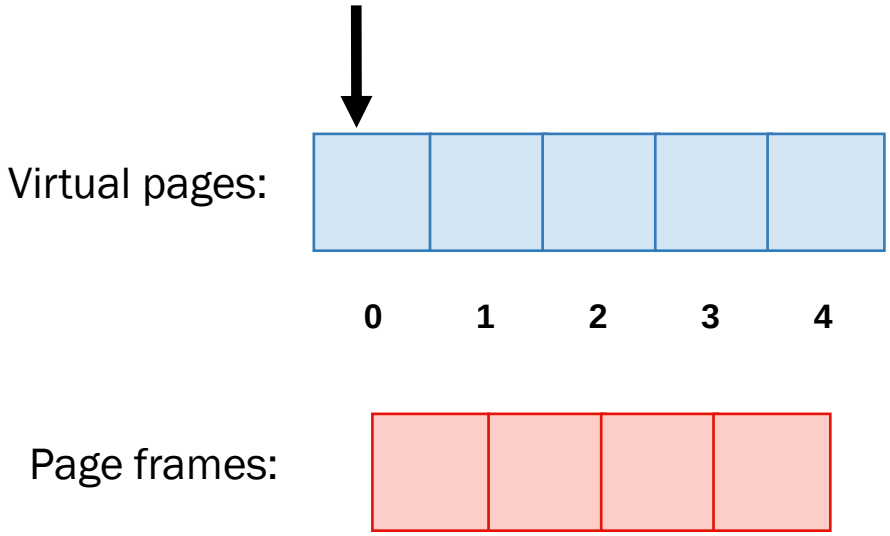
Consider physical memory size: 3 pages vs. 4 pages

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

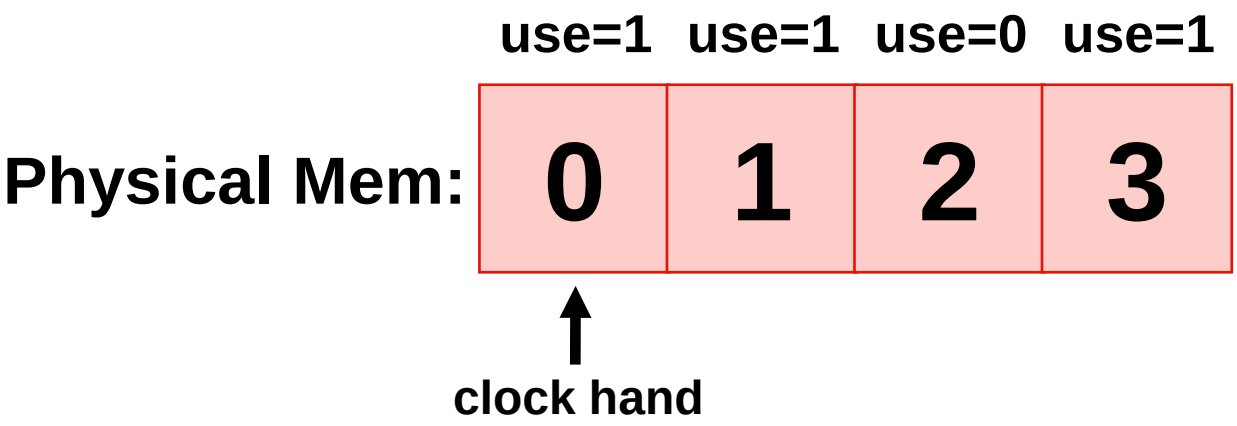
Hardware

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

Operating System

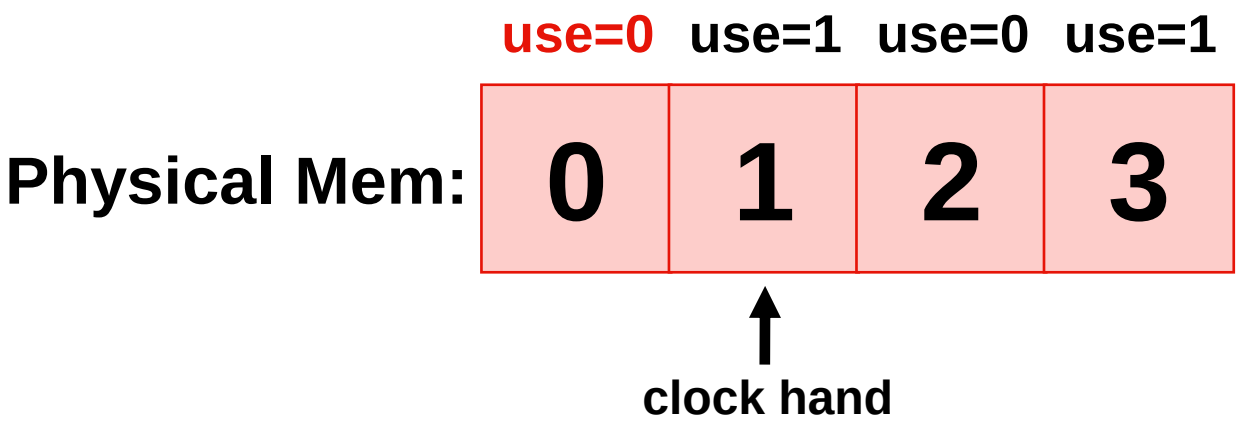
- Page replacement: look for page with accessed bit cleared (has not been referenced for awhile)
- Implementation:
  - Keep pointer to last examined page frame
  - Traverse pages in circular buffer
  - Clear accessed bits as search
  - Stop when find page with already cleared accessed bit, replace this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

Clock: Look For a Page



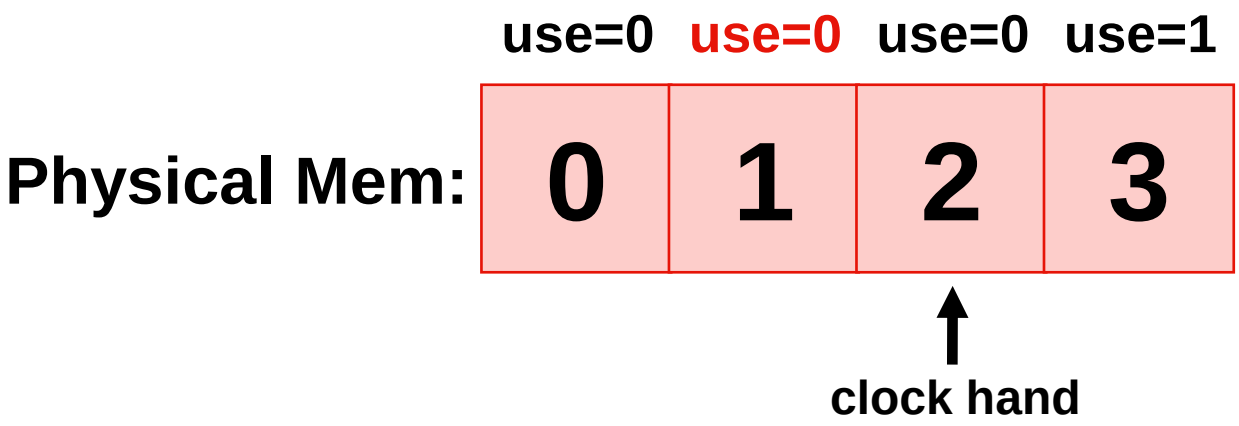
Page Needing Page Frame

Clock: Look For a Page



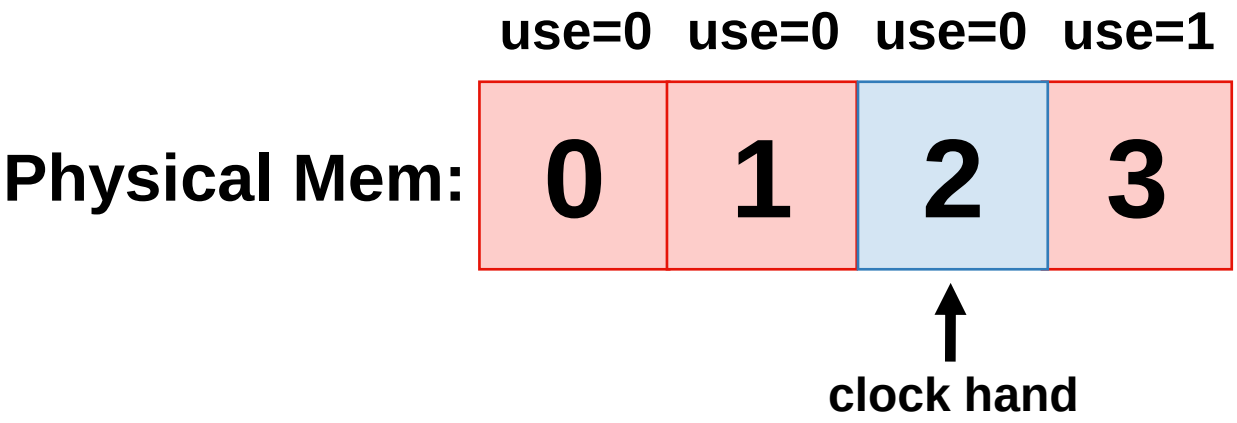
Page Needing Page Frame

Clock: Look For a Page



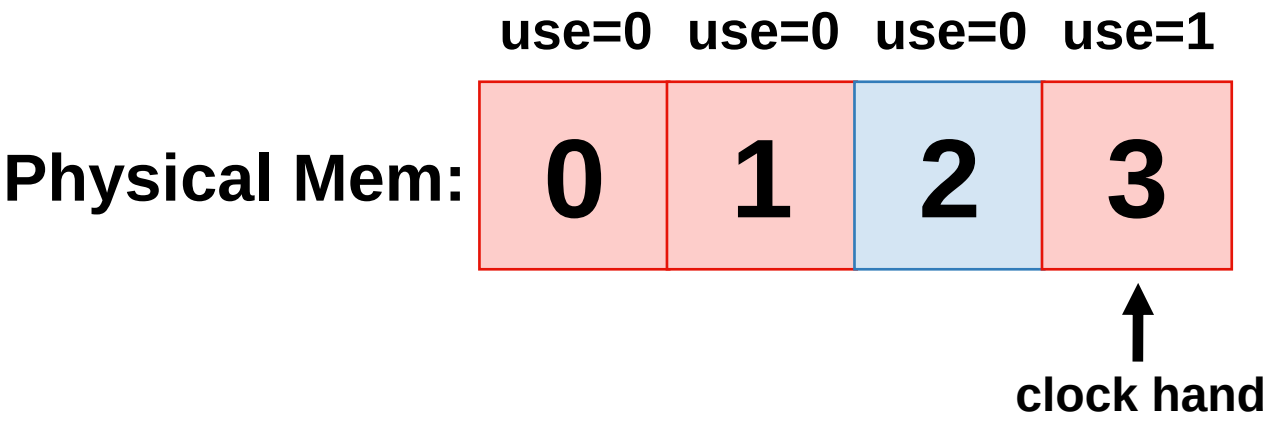
Page Needing Page Frame

Clock: Look For a Page



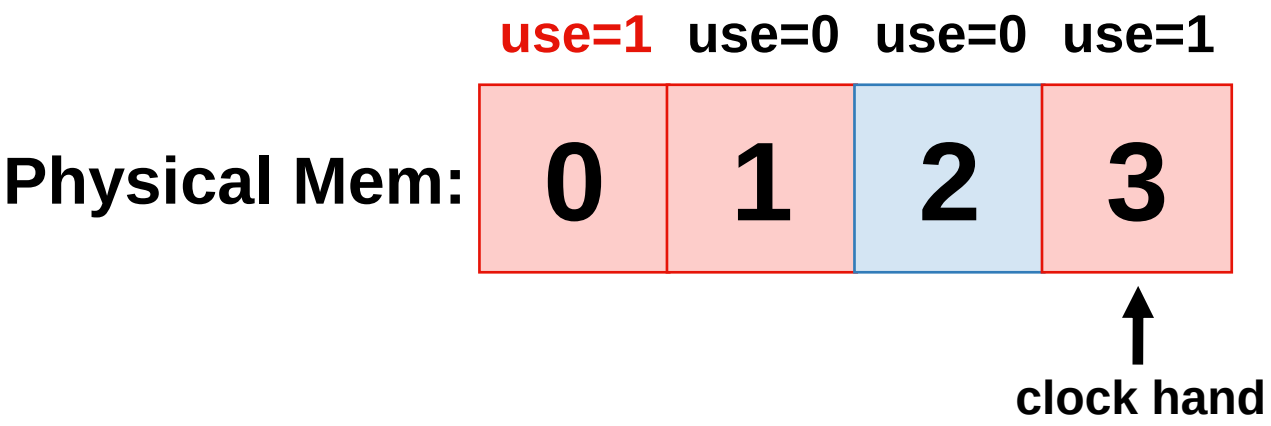
Evict page 2 because it has not been recently used

Clock: Look For a Page



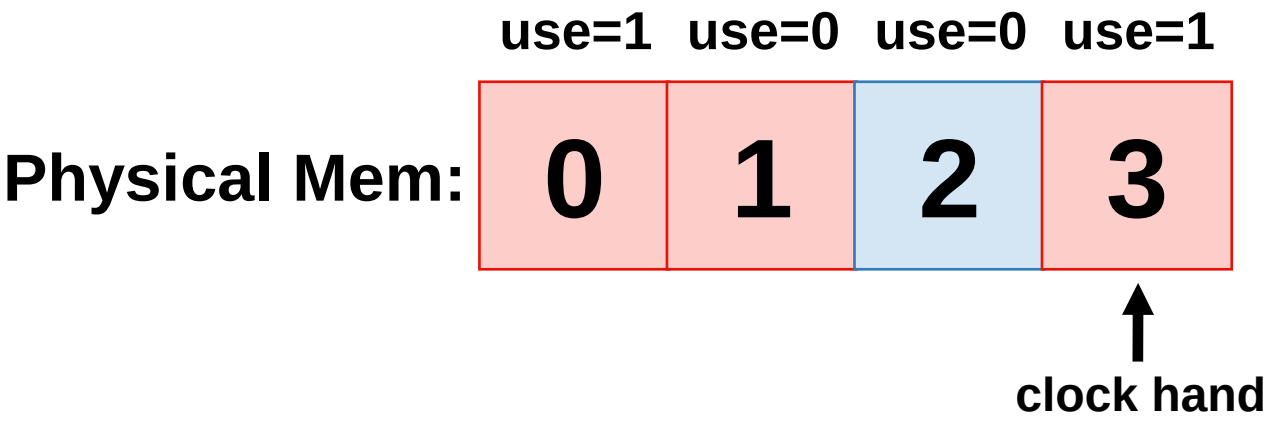
Advance hand so that frame 2 won't be reconsidered until clock hand loops back around

Clock: Look For a Page



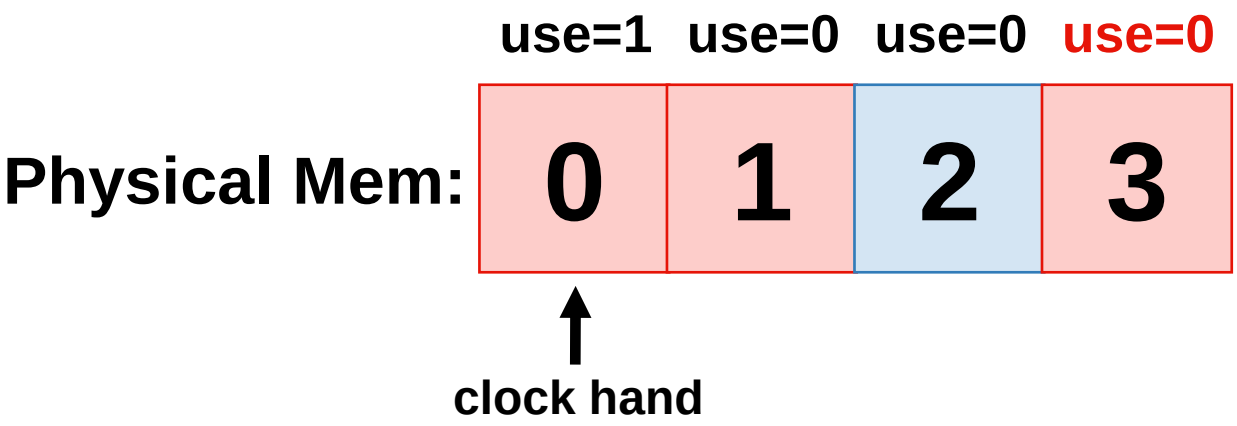
page 0 is accessed…

Clock: Look For a Page



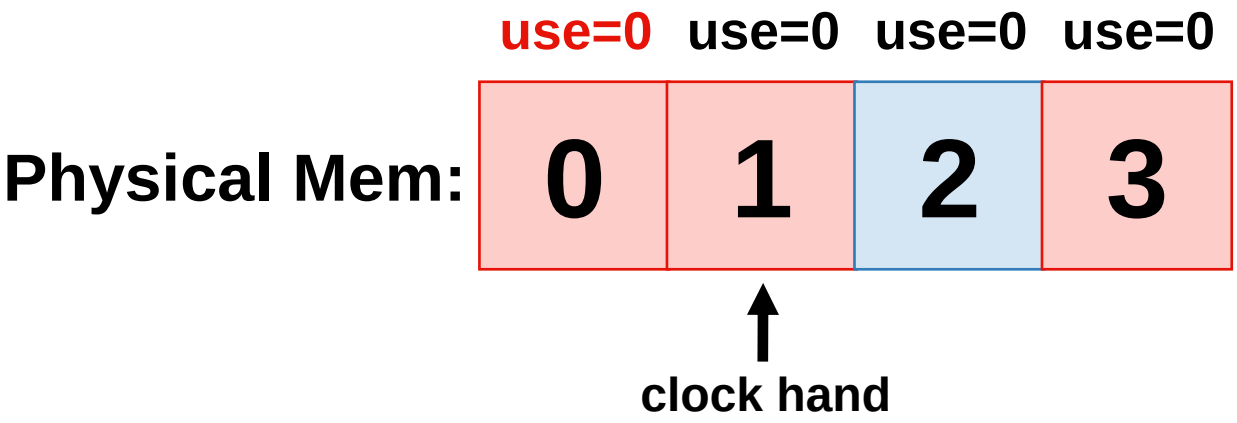
Page Needing Page Frame

Clock: Look For a Page



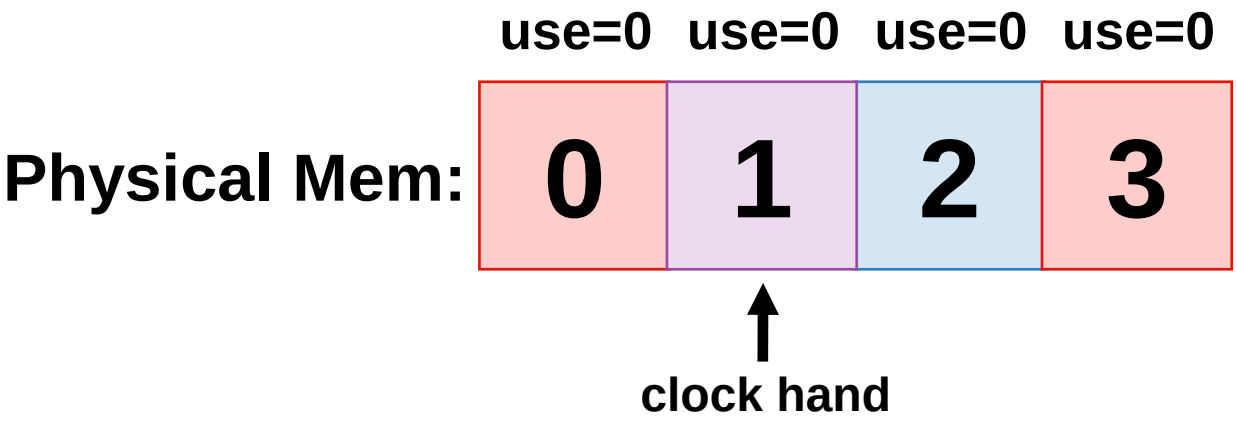
Page Needing Page Frame

Clock: Look For a Page



Page Needing Page Frame

Clock: Look For a Page



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:

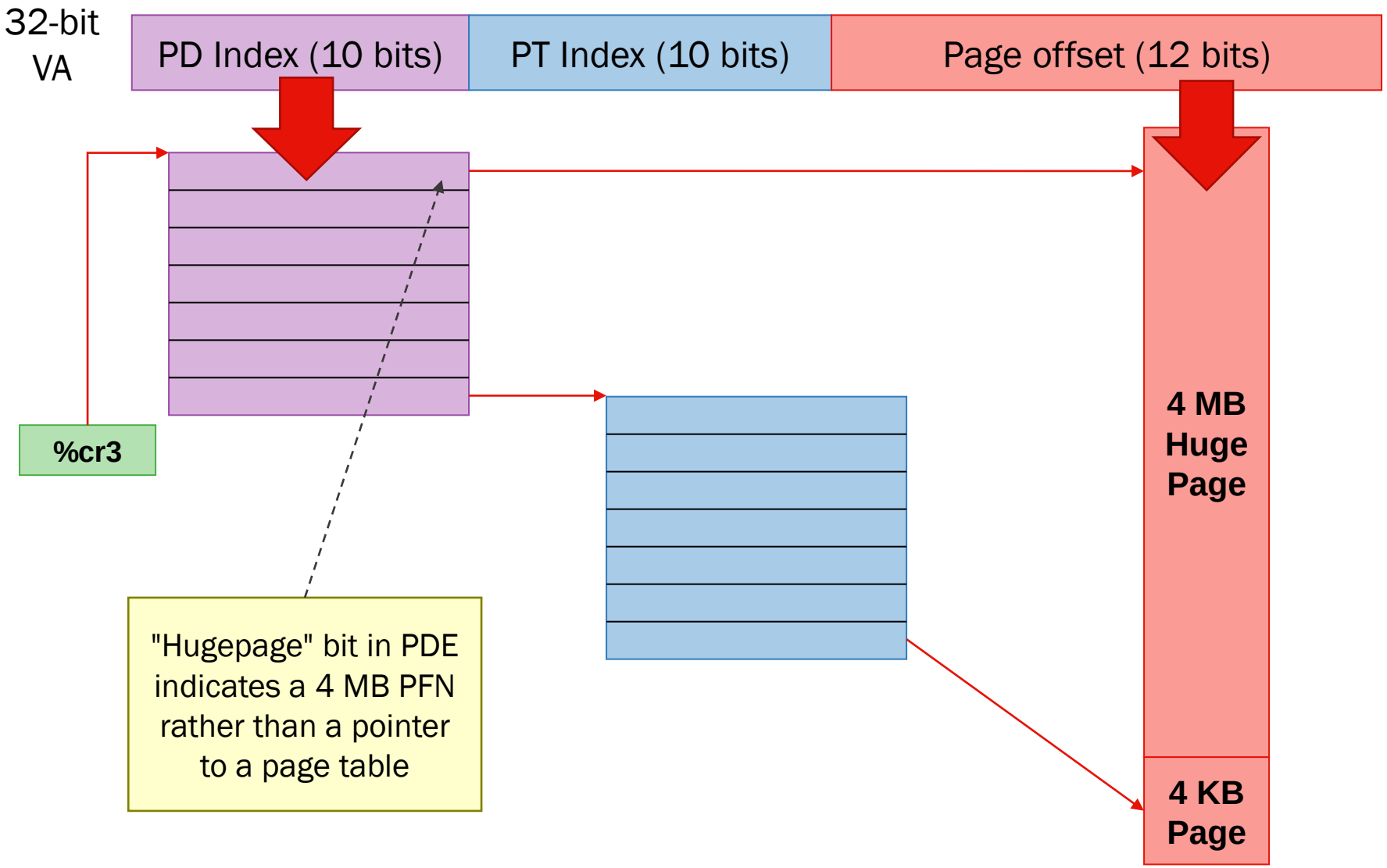
- How can the OS get control (i.e., generate a trap) every time accessed bit should be set? (i.e., when a page is accessed?)

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