Main Memory

CS 4410, Operating Systems

Fall 2016
Cornell University

Rachit Agarwal
Anne Bracy

See: Ch 8 & 9 in OSPP textbook

The slides are the product of many rounds of teaching CS 4410 by Professors Sirer, Bracy, Agarwal, George, and Van Renesse.

Main Memory

- Address Translation (Chapter 8)
- Caching & Virtual Memory (9.1-9.7)

New: all in the broader context of the OS
(and its perspective)

Social Network

Address Translation

- **Paged Translation**
- Efficient Address Translation

Paged Translation in the Abstract

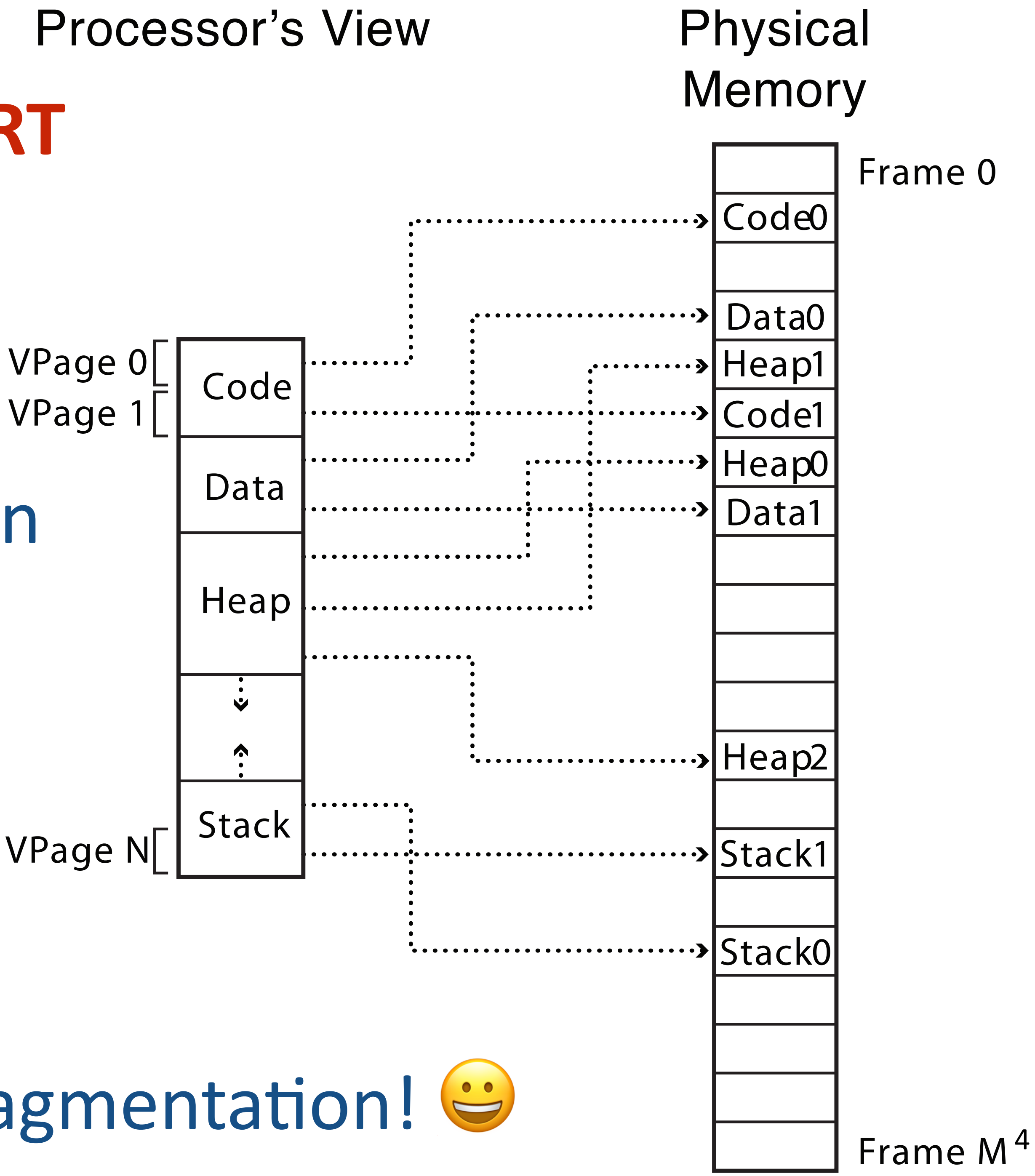
TERMINOLOGY ALERT

Page:

the data itself

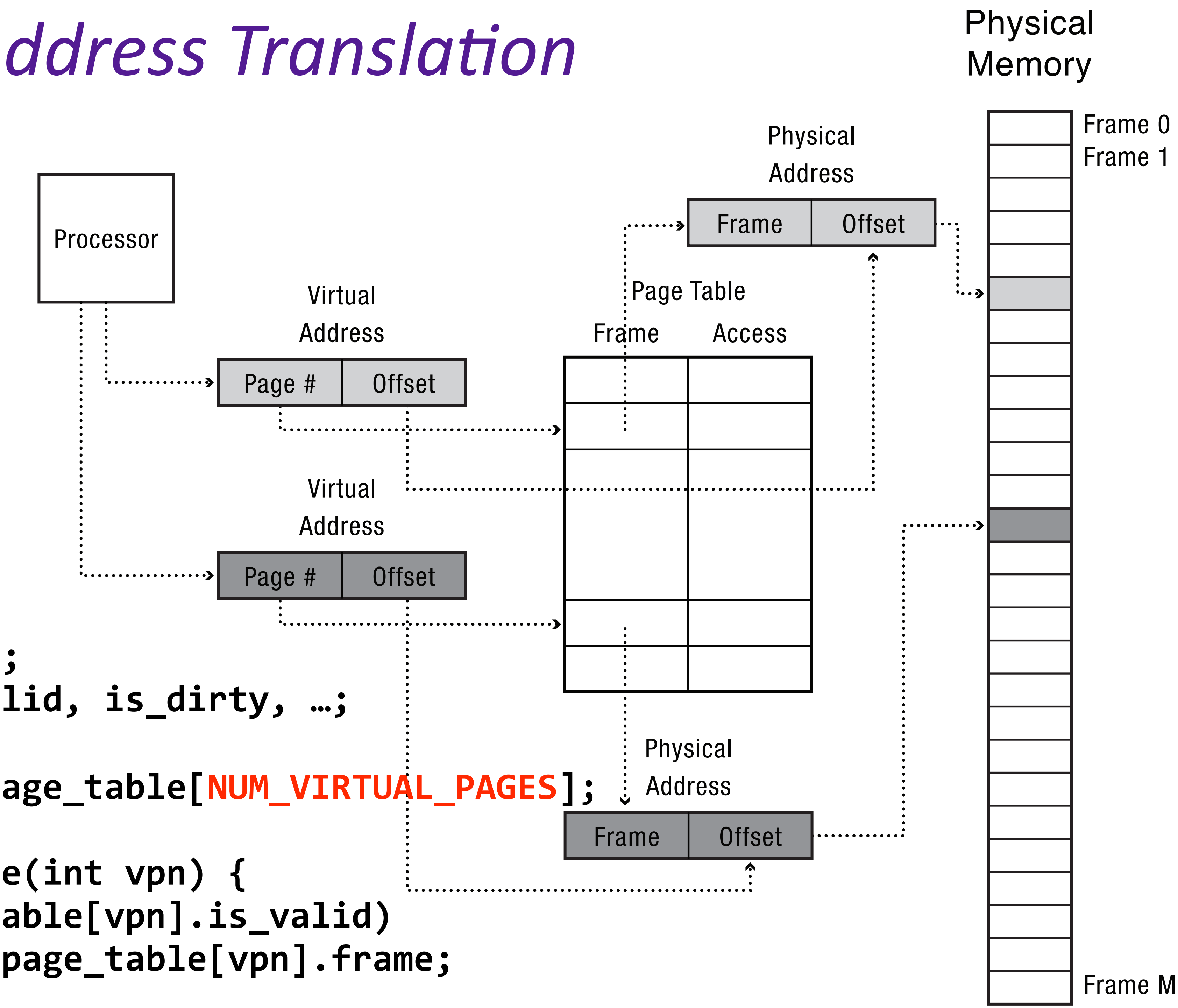
Frame:

the physical location



No more external fragmentation! 😀

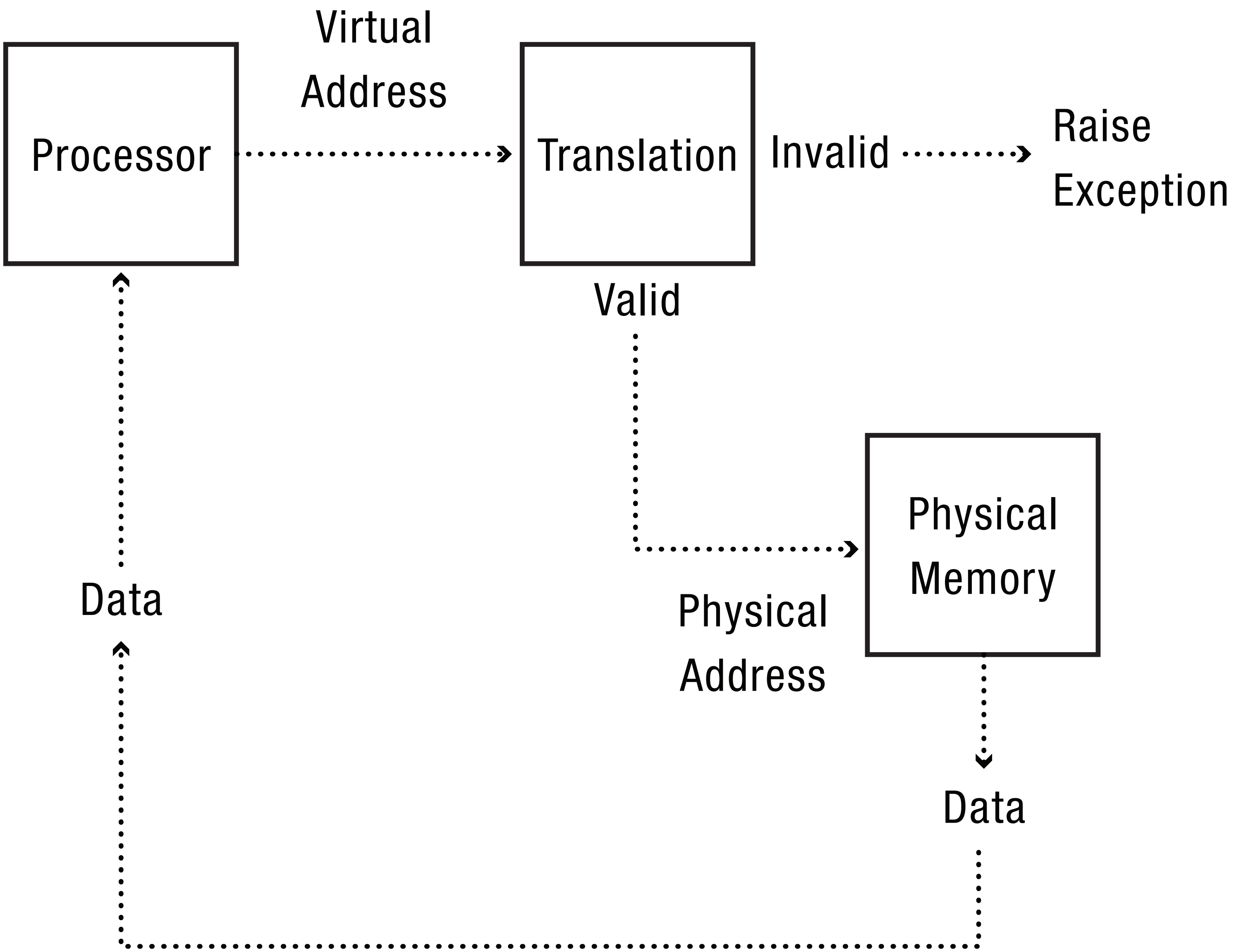
Paged Address Translation



```
struct {
   int frame;
   bit is_valid, is_dirty, …;
} PTE;
struct PTE page_table[NUM_VIRTUAL_PAGES];

int translate(int vpn) {
  if (page_table[vpn].is_valid)
     return page_table[vpn].frame;
  else…
}
```

Address Translation, Conceptually



5 Paging Questions

What is saved/restored on a context switch?

What if page size is very small?

What if page size is very large?

What if the address space is sparse?

What if the virtual address space is large?

5 Paging Questions

What is saved/restored on a context switch?

- Pointer to page table, size of page table
- Page Table itself is in main memory

What if page size is very small?

What if page size is very large?

What if the address space is sparse?

What if the virtual address space is large?

5 Paging Questions

What is saved/restored on a context switch?

What if page size is very small?

- Lots and lots of page table entries!

What if page size is very large?

What if the address space is sparse?

What if the virtual address space is large?

5 Paging Questions

What is saved/restored on a context switch?

What if page size is very small?

What if page size is very large?

- Internal fragmentation

What if the address space is sparse?

What if the virtual address space is large?

5 Paging Questions

What is saved/restored on a context switch?

What if page size is very small?

What if page size is very large?

What if the address space is sparse?

- Lots of wasted space in the page table
- Per-processor heaps
- Per-thread stacks
- Memory-mapped files
- Dynamically linked libraries

What if the virtual address space is large?

5 Paging Questions

What is saved/restored on a context switch?

What if page size is very small?

What if page size is very large?

What if the address space is sparse?

What if the virtual address space is large?

- Even more wasted space
- 32-bits, 4KB pages => 1M page table entries
- 64-bits => 4 quadrillion page table entries

Address Translation

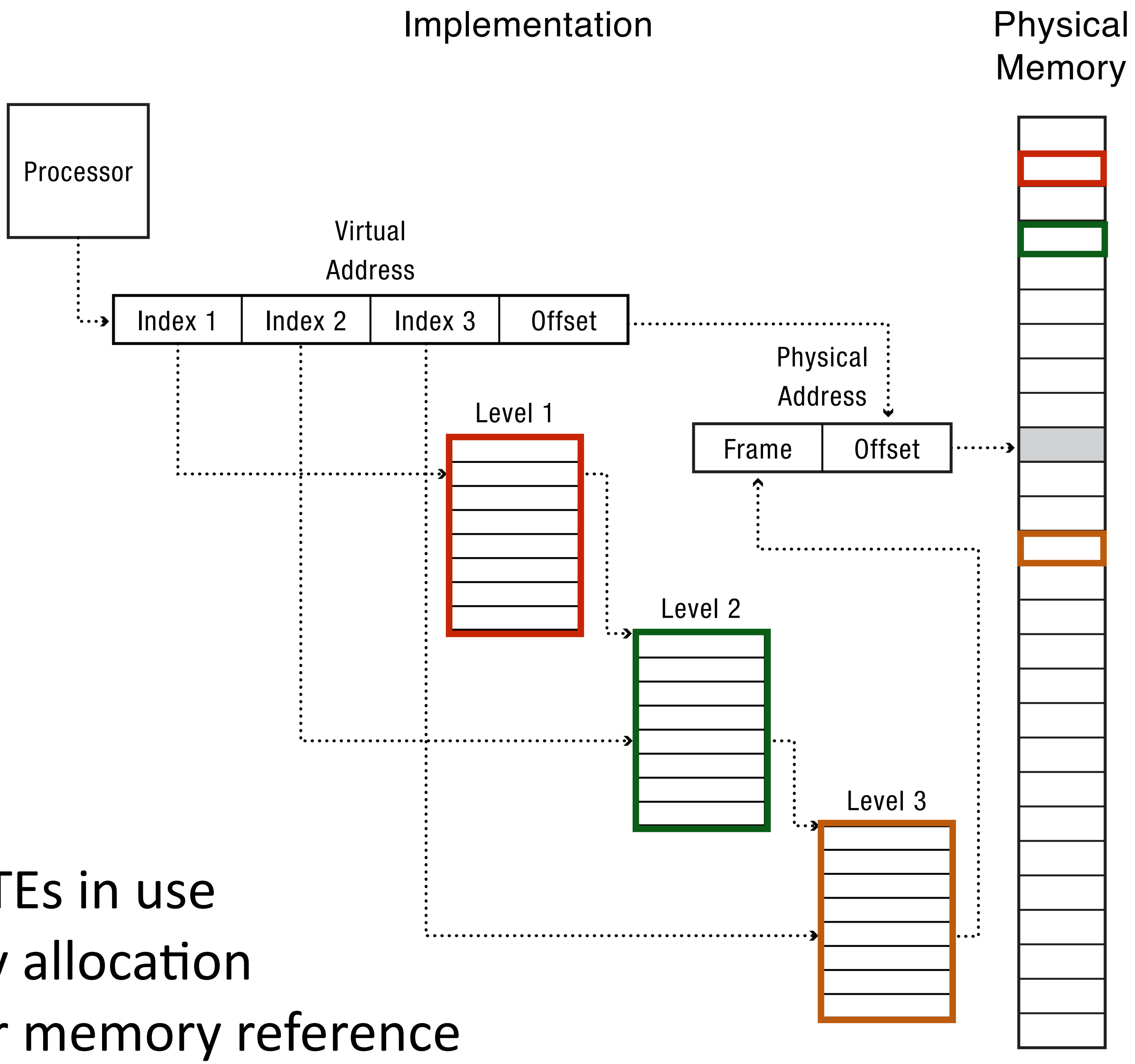
- Paged Translation
- **Efficient Address Translation**

    + Multi-level Page Tables

    + Inverted Page Tables

    + TLBs

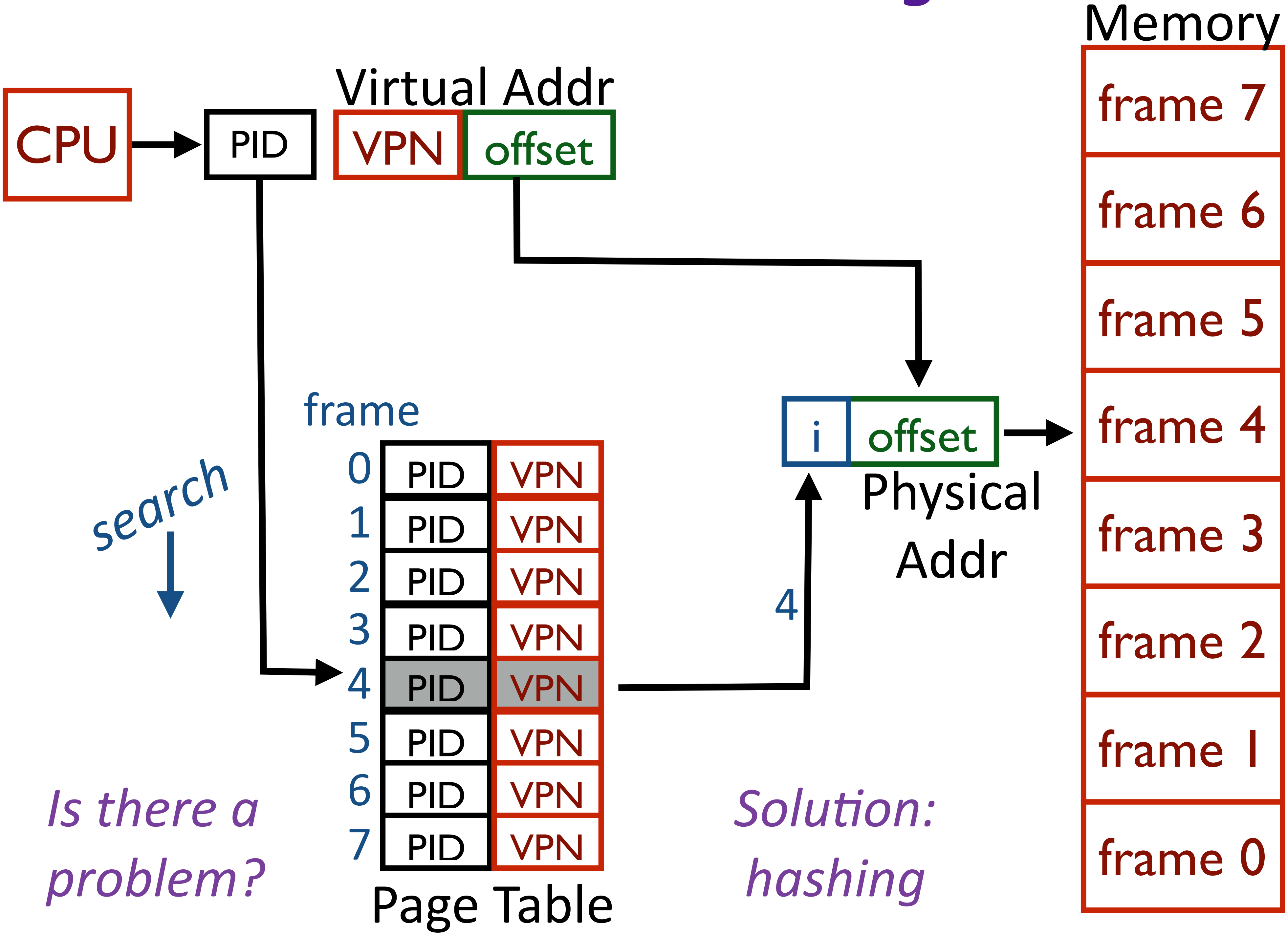
Multi-Level Page Tables to the Rescue!



+ Allocate only PTEs in use
+ Simple memory allocation
— 2+ lookups per memory reference

Back to the movies…

Can we do better? ***Inverted Page Table***



Is there a problem?

Solution: hashing

Complete Page Table Entry (PTE)

Valid	Protection R/W/X	Ref	Dirty	Index

Index is an index into

- table of memory frames (if bottom level)
- table of page table frames (if multilevel page table)
- backing store (if page is not valid)

Synonyms:

- Valid bit == Present bit
- Dirty bit == Modified bit
- Referenced bit == Accessed bit

(the contents of) ***A Virtual Page Can Be***

Mapped

- to a physical frame

Not Mapped (→ Page Fault)

- in a physical frame, but not currently mapped
- still in the original program file
- zero-filled (heap/BSS, stack)
- on backing store ("paged or swapped out")
- illegal: not part of a segment

  → Segmentation Fault

Address Translation

- Paged Translation
- Efficient Address Translation

  + Multi-level Page Tables

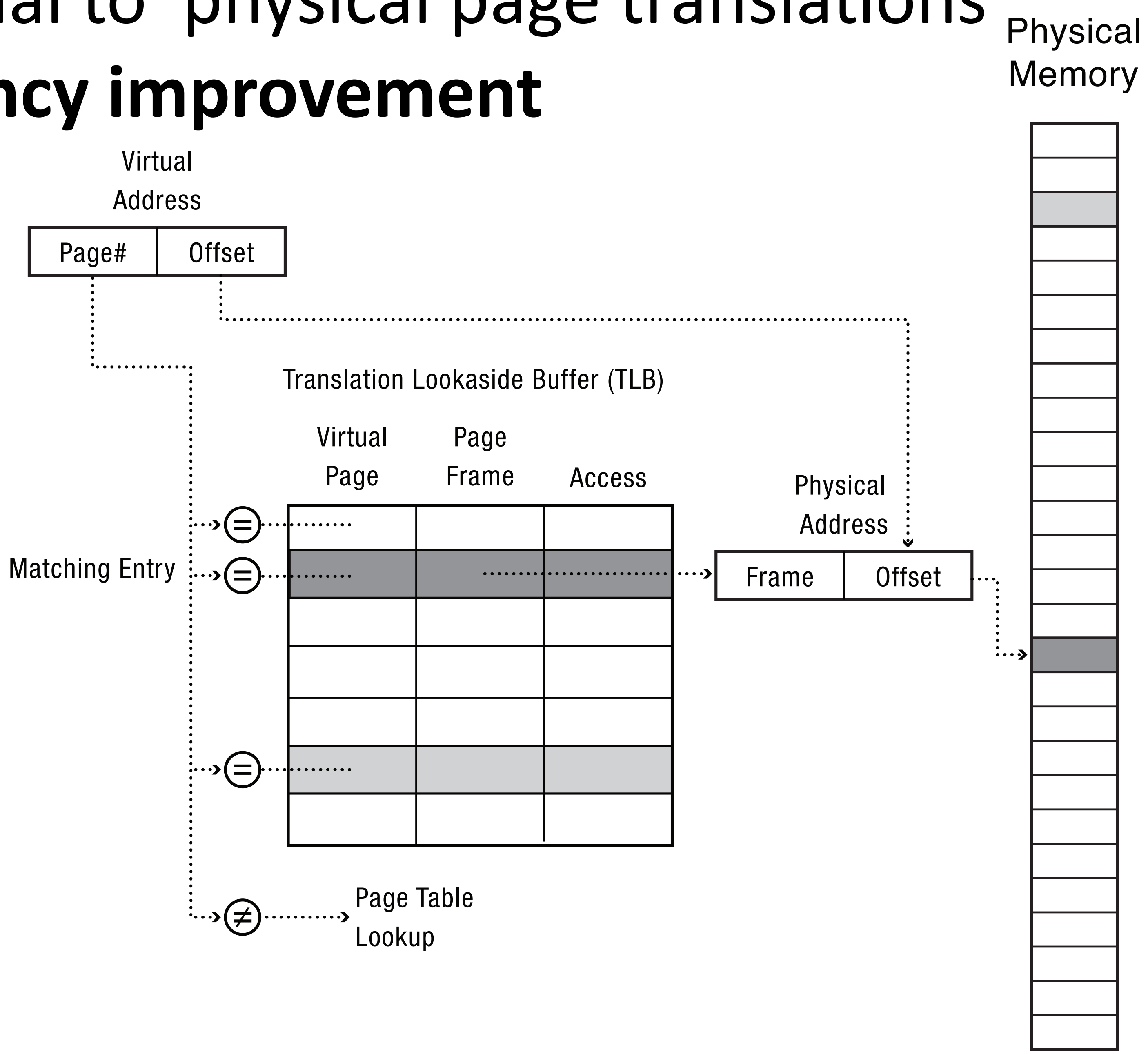
  + Inverted Page Tables

  + TLBs

Translation Lookaside Buffer

Cache of virtual to physical page translations

Major efficiency improvement



5 Translation Questions

When does the CPU access the TLB?

What happens on a TLB miss?

What happens to the TLB on a context switch?

What happens when a page is shared among many processes?

What happens when a page is swapped out?

5 Translation Questions

When does the CPU access the TLB?

- First thing!
- *While you access the L1 caches*

What happens on a TLB miss?

What happens to the TLB on a context switch?

What happens when a page is shared among many processes?

What happens when a page is swapped out?

5 Translation Questions

When does the CPU access the TLB?

What happens on a TLB miss?

- Trap to kernel, kernel fills TLB w/translation, resumes execution

What happens to the TLB on a context switch?

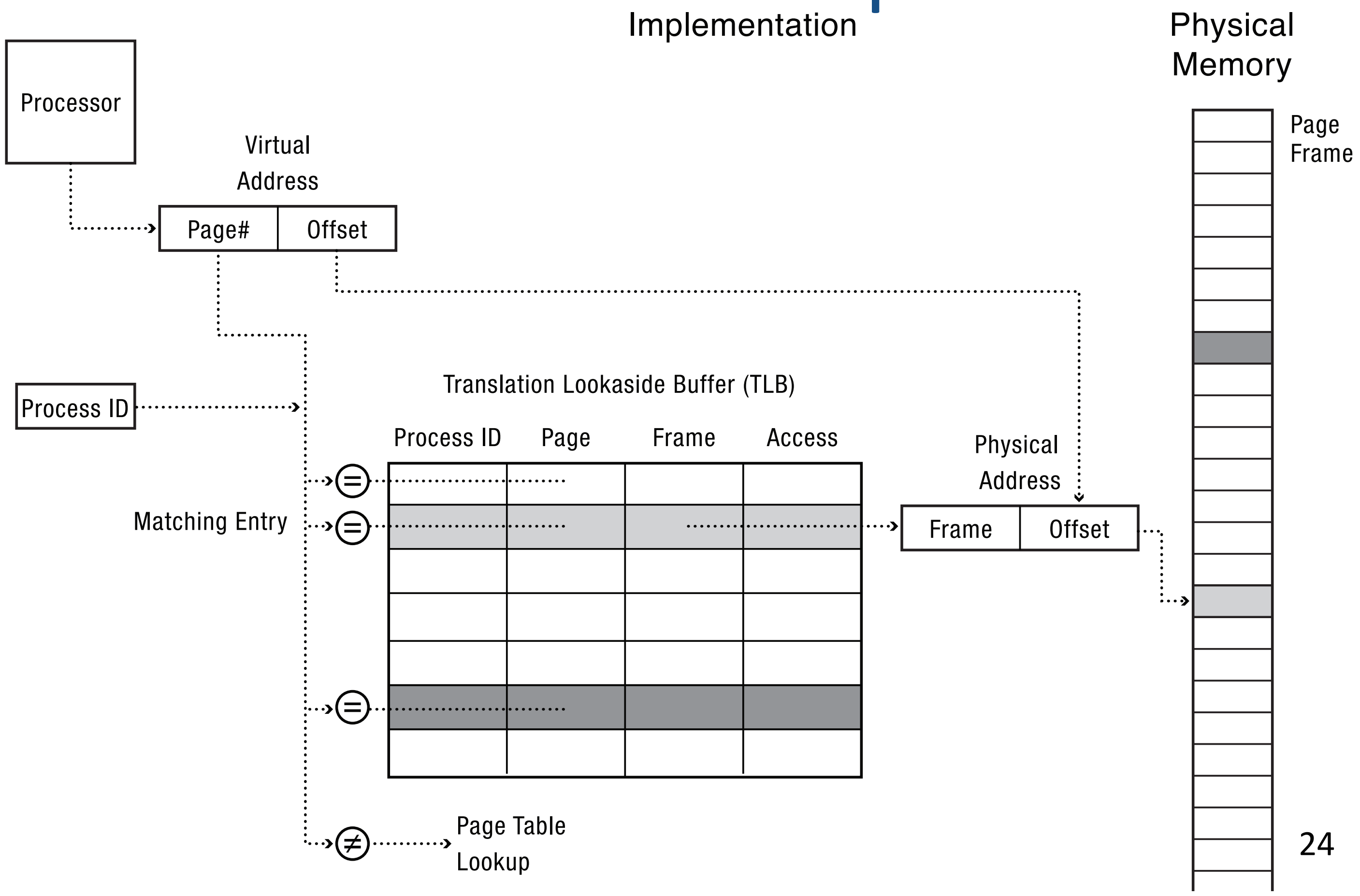
What happens when a page is shared among many processes?

What happens when a page is swapped out?

5 Translation Questions

What happens to the TLB on a context switch?

- Becomes totally useless? Flush?
- Tag the TLB with a PID
- TLB hit only if PID matches current process



5 Translation Questions

When does the CPU access the TLB?

What happens on a TLB miss?

What happens to the TLB on a context switch?

What happens when a page is shared among many processes?

- (Shared **frames** is more accurate)
- Examples: NULL Page (invalid to all, why?), exec-only (libraries), read-only data (strings),
- Mostly nothing changes…
- Need to indicate sharing in inverted page table

What happens when a page is swapped out?

5 Translation Questions

When does the CPU access the TLB?

What happens on a TLB miss?

What happens to the TLB on a context switch?

What happens when a page is shared among many processes?

What happens when a page is swapped out?

- Need to update the Page Table**(s)**
  - Core Map (frames → pages)
- Need to update the TLB
  - TLB Shootdown

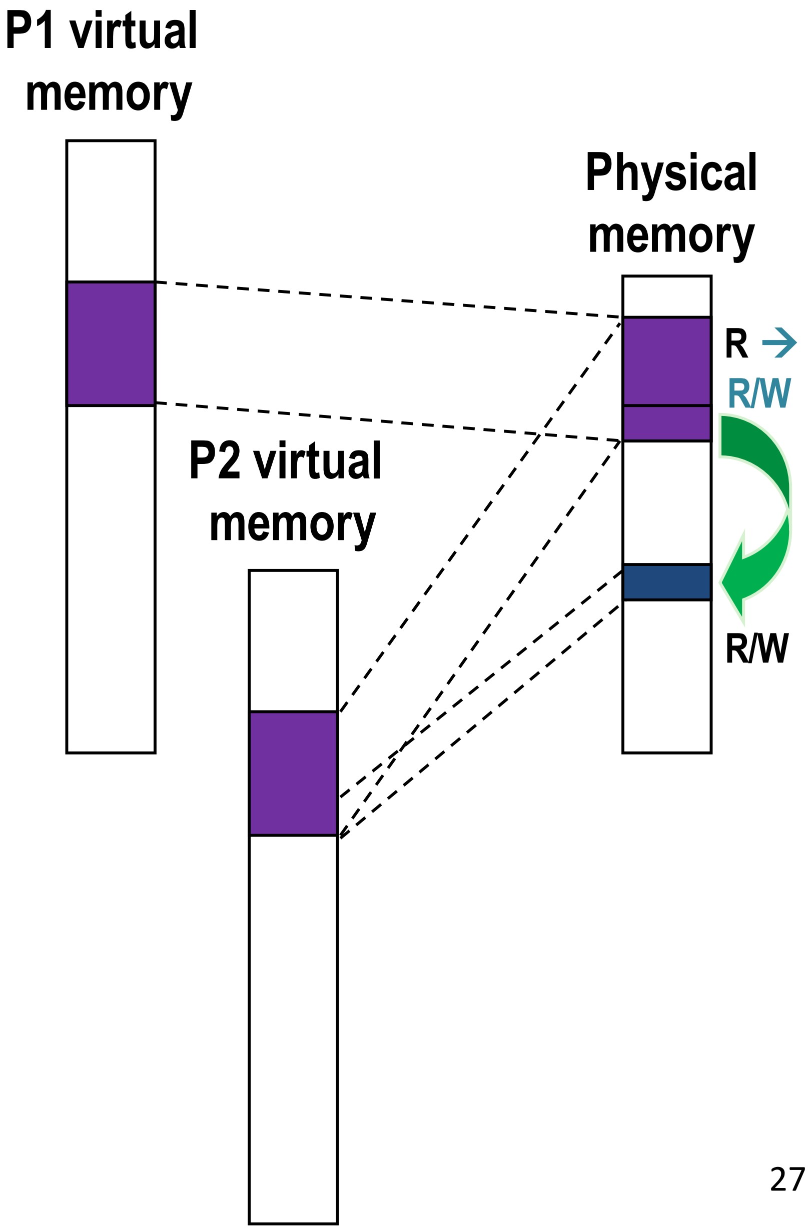
Nice Addr Translation Feature: Copy-on-Write

Useful for “fork()” and initialized data

Initially map page read-only

Upon page fault:

- Allocate a new frame
- Copy frame
- Map new page R/W
- Also map “other” page R/W



Address Translation Uses

Process isolation

- Keep a process from touching anyone else's memory, or the kernel's

Efficient interprocess communication

- Shared regions of memory between processes

Shared code segments

- common libraries used by many different programs

Program initialization

- Start running a program before it is entirely in memory

Dynamic memory allocation

- Allocate and initialize stack/heap pages on demand

MORE *Address Translation Uses*

Program debugging

- Data breakpoints when address is accessed

Memory mapped files

- Access file data using load/store instructions

Demand-paged virtual memory

- Illusion of near-infinite memory, backed by disk or memory on other machines

Checkpointing/restart

- Transparently save a copy of a process, without stopping the program while the save happens

Distributed shared memory

- Illusion of memory that is shared between machines

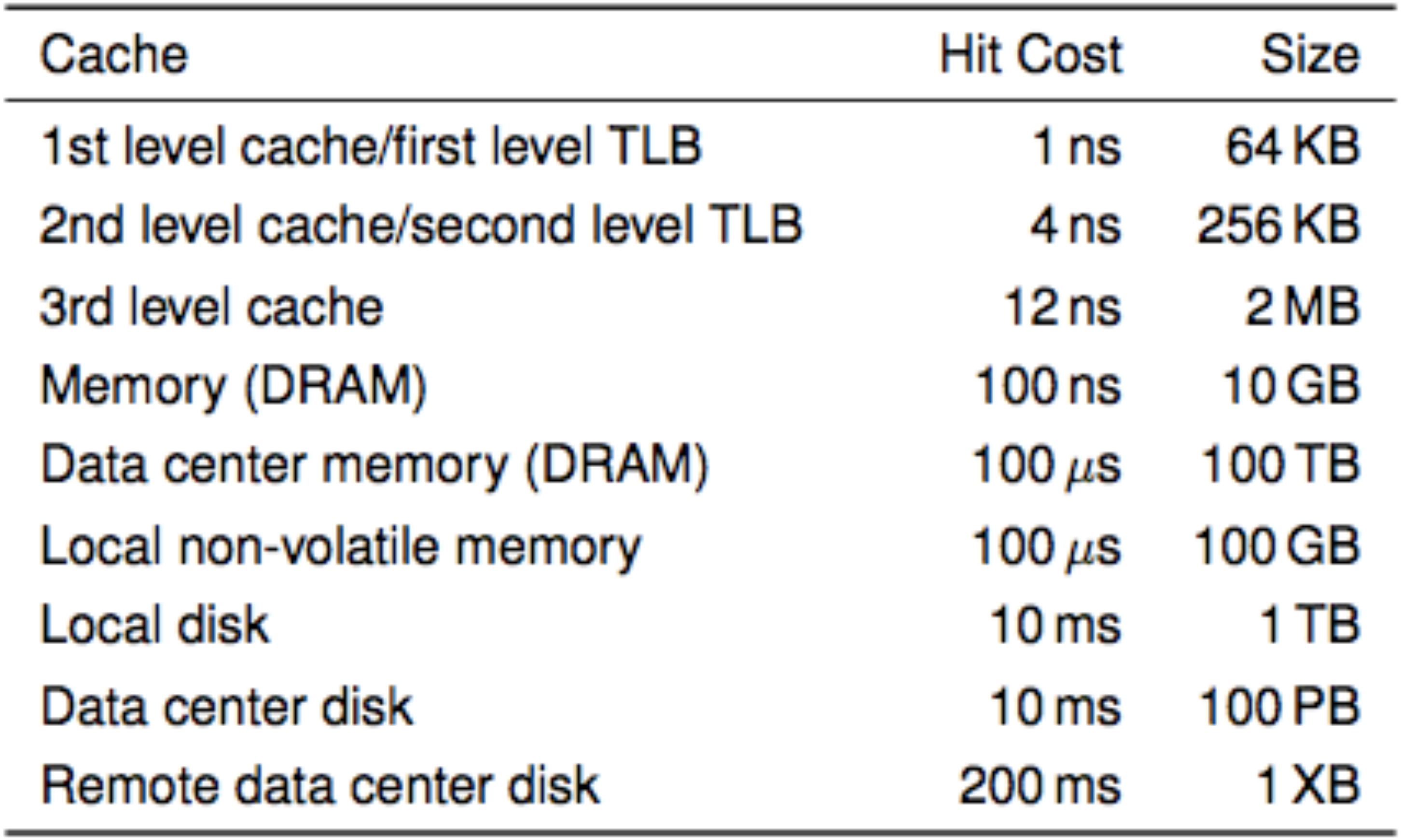
Caching

- Assignment: where do you put the data?
- Replacement: who do you kick out?
- Problems with Caching

What are some examples of caching?

- TLBs
- hardware caches
- internet naming
- web content
- web search
- email clients
- incremental compilation
- just in time translation
- virtual memory
- file systems
- branch prediction

Memory Hierarchy



Cache	Hit Cost	Size
1st level cache/first level TLB	1 ns	64 KB
2nd level cache/second level TLB	4 ns	256 KB
3rd level cache	12 ns	2 MB
Memory (DRAM)	100 ns	10 GB
Data center memory (DRAM)	100 μs	100 TB
Local non-volatile memory	100 μs	100 GB
Local disk	10 ms	1 TB
Data center disk	10 ms	100 PB
Remote data center disk	200 ms	1 XB

Every layer is a cache for the layer below it.

Caching

- **Assignment: where do you put the data?**
  - Which entry in the cache? — not much choice
  - Which frame in memory?
- Replacement: who do you kick out?
- Problems with Caching

Working Set

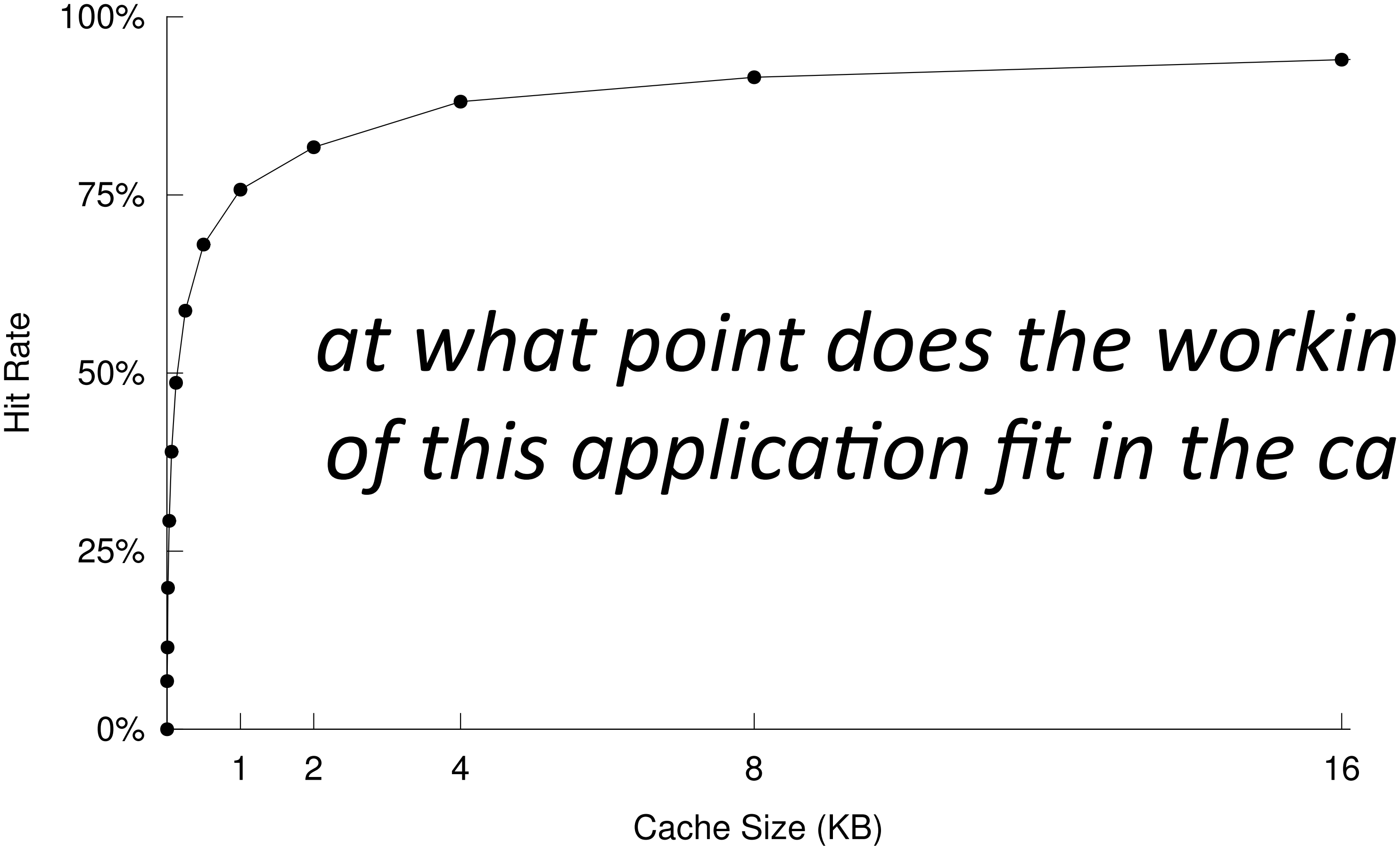
First Definition:

Collection of a process' most recently used pages

The Working Set Model for Program Behavior, Peter J. Denning, 1968

Formal definition:

Pages referenced by process in last Δ time-units



at what point does the working set of this application fit in the cache?

Thrashing

Excessive rate of paging

Cache lines evicted before they can be reused

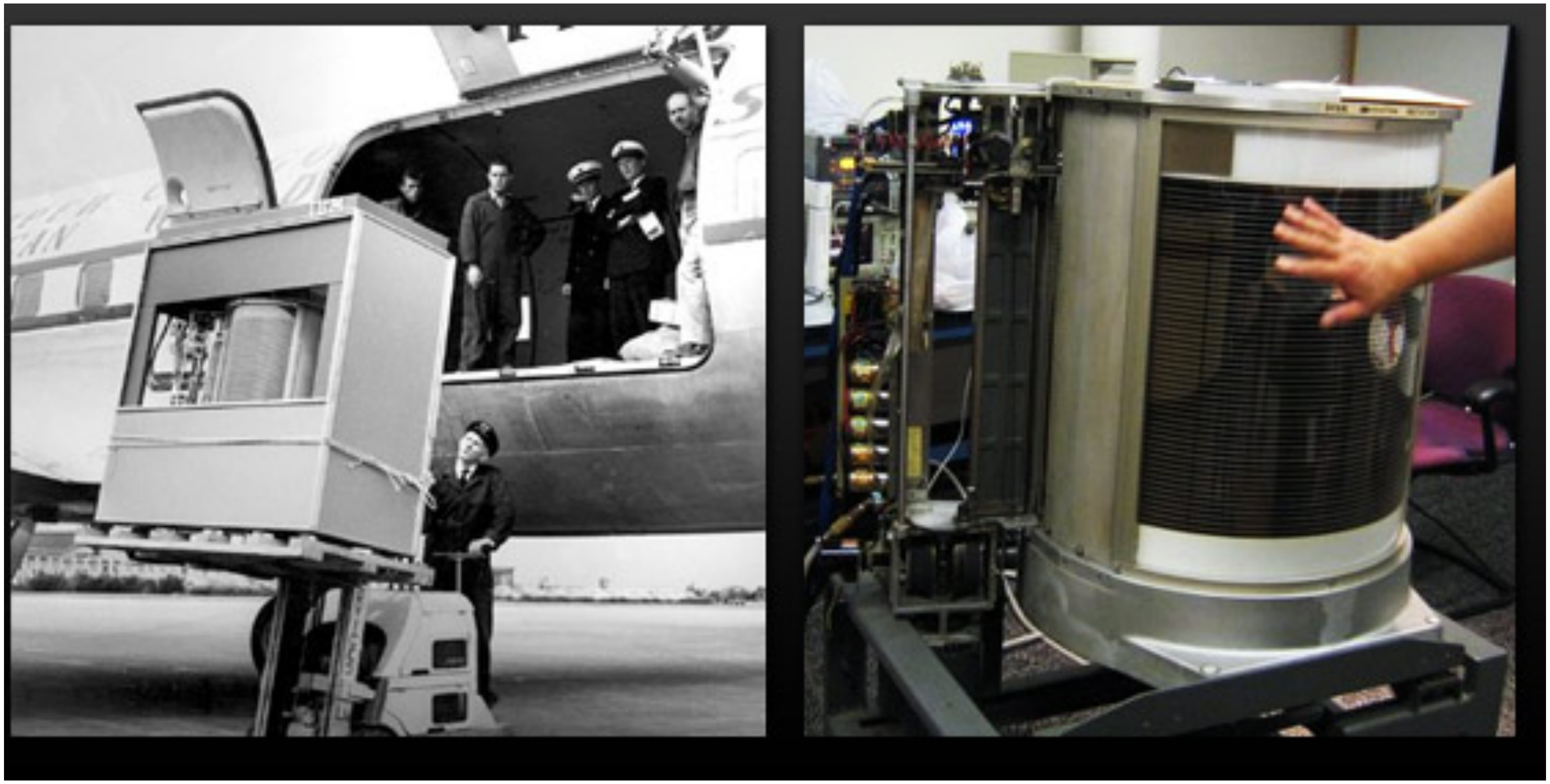
Causes:

- Cache not big enough to fit working set
- Bad luck (conflicts)
- Bad eviction policies (later)

Prevention:

- restructure your code
  (smaller working set, shift data around)
- restructure your cache

Why “thrashing”?



The first hard disk drive—the IBM Model 350 Disk File (came w/IBM 305 RAMAC, 1956).

Total storage = 5 million characters (just under 5 MB).

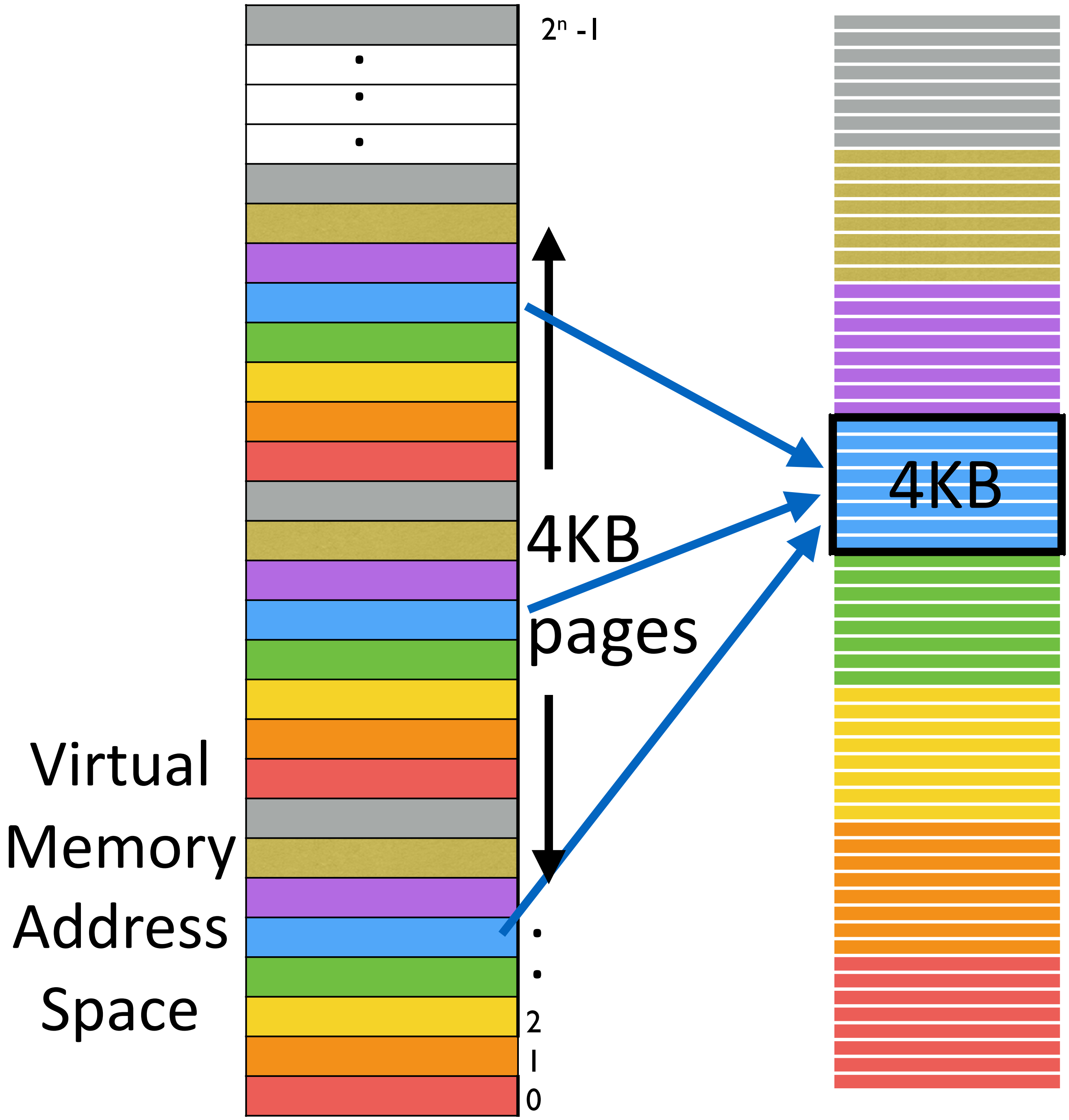
http://royal.pingdom.com/2008/04/08/the-history-of-computer-data-storage-in-pictures/

“Thrash” dates from the 1960’s, when disk drives were as large as washing machines. If a program’s working set did not fit in memory, the system would need to shuffle memory pages back and forth to disk. This burst of activity would violently shake the disk drive.

Caching

- **Assignment: where do you put the data?**
  - Which entry in the cache? — not much choice
  - Which frame in memory? — **lots of freedom**
- Replacement: who do you kick out?
- Problems with Caching

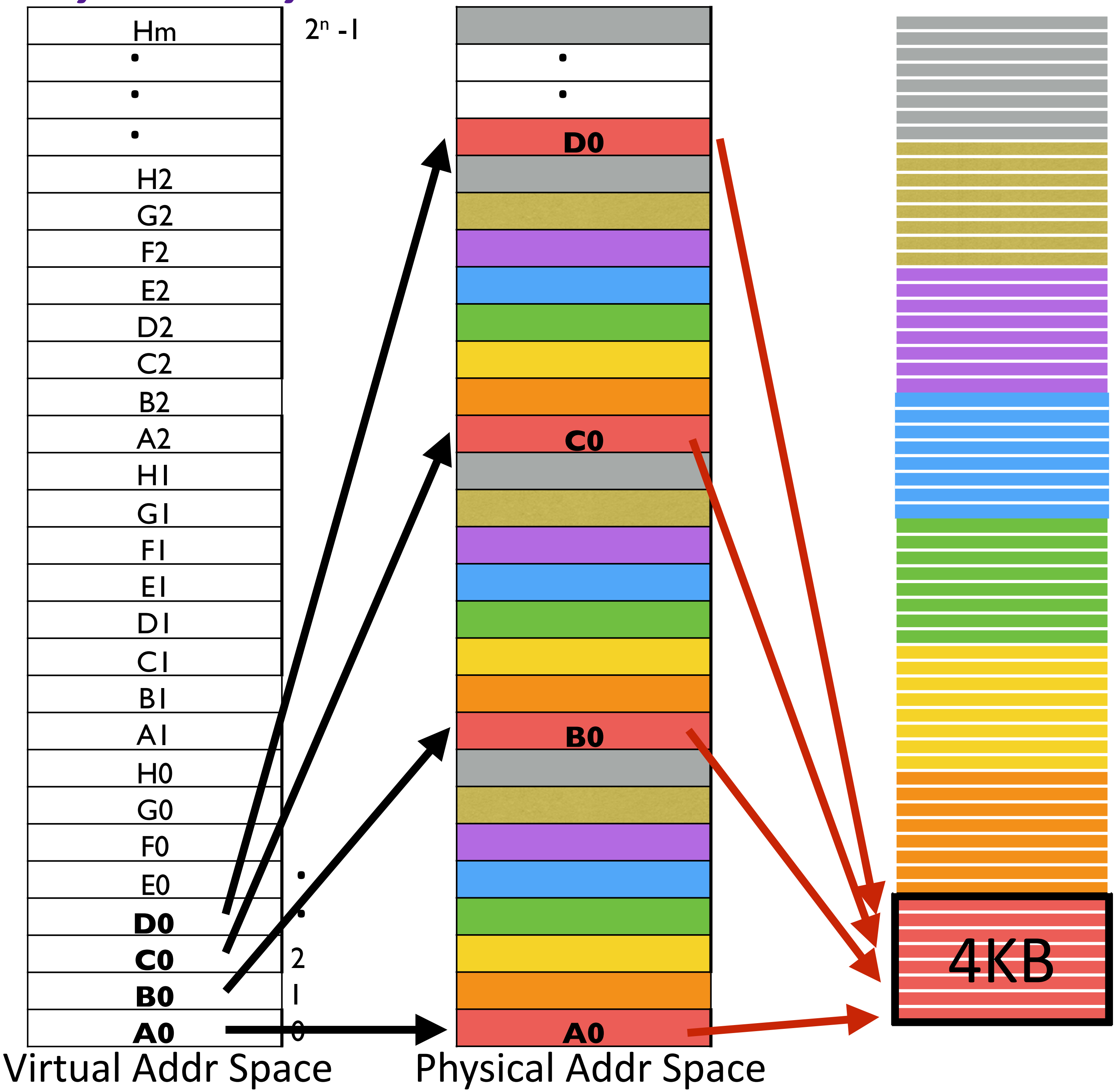
Virtually Addressed Caches



- each page occupies some # of consecutive cache entries
- same-colored pages mapped to sets of same color in cache
- Pages live across entire color range of the cache. Also supports spatial locality.

Virtually Addressed 32 KB L1 Cache

Physically Addressed Caches…



What if virtual pages are assigned to physical pages that are 64KB apart?

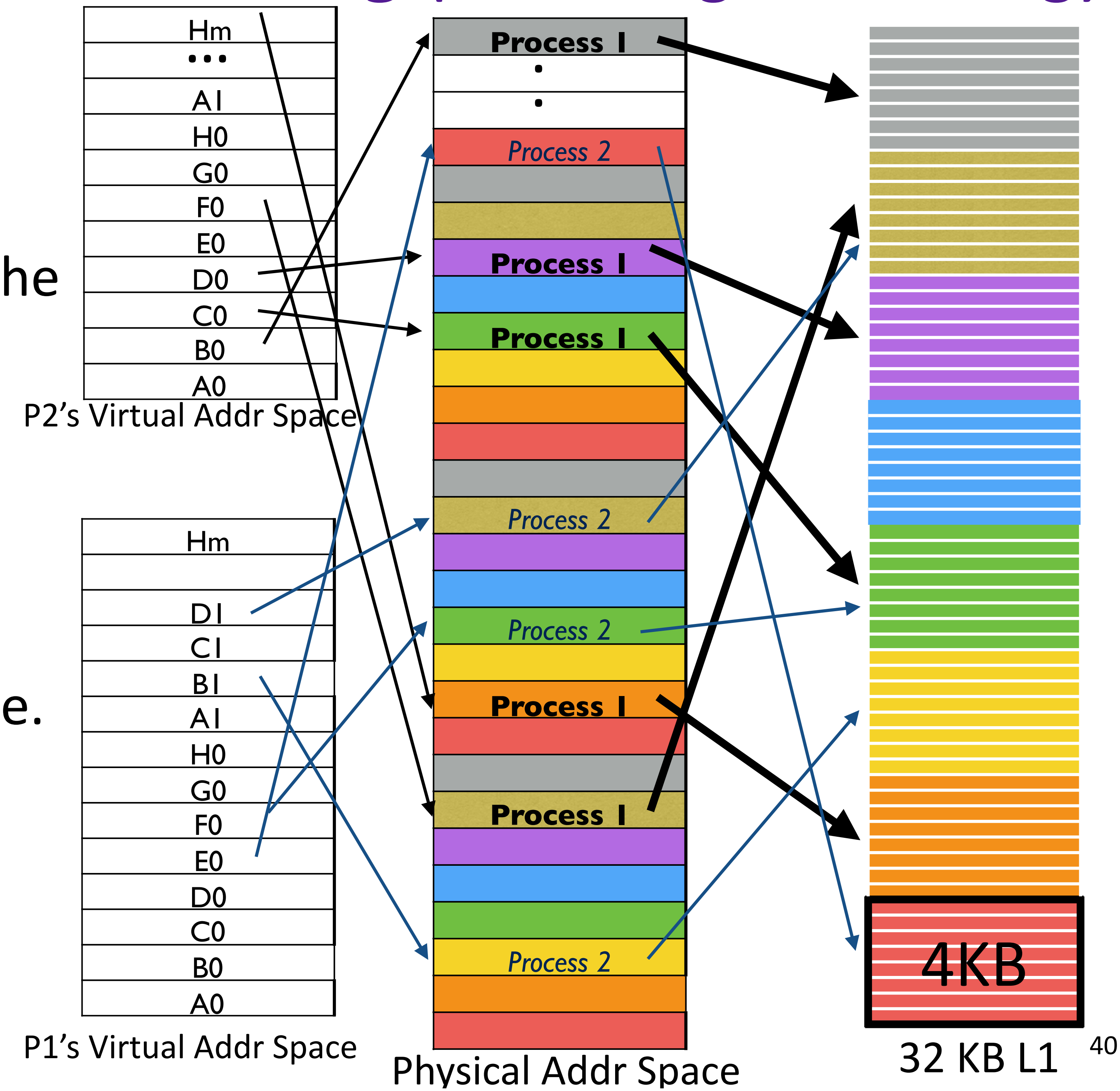
BAD: disrupts spatial locality

WORSE: cache effectively smaller

Physically Addressed 32 KB L1

Solution: ***Cache Coloring*** *(AKA Page Coloring)*

1. Color frames according to cache configuration.
2. Spread each process' pages across as many colors as possible.



Caching

- Assignment: where do you put the data?
- **Replacement: who do you kick out?**
- Problems with Caching

What happens when Memory is full?

Swapping vs. Paging

Swapping

- Loads entire process in memory, runs it, exit
- “Swap in” or “Swap out” a process
- Slow (for big, long-lived processes)
- Wasteful (might not require everything)

Paging

- Runs all processes concurrently, taking only pieces of memory (specifically, pages) away from each process
- Finer granularity, higher performance
- Paging completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

The verb “to swap” is also used to refer to pushing contents of a page out to disk in order to bring other content from disk; this is distinct from the noun “swapping”

Demand Paging on MIPS

1. TLB miss
2. Trap to kernel
3. Page table walk
4. Find page is invalid
5. Convert virtual address to file + offset
6. Allocate page frame
   - Evict page if needed
7. Initiate disk block read into page frame
8. Disk interrupt when DMA complete
9. Mark page as valid
10. Load TLB entry
11. Resume process at faulting instruction
12. Execute instruction

Demand Paging

1. TLB miss
2. Page table walk
3. Page fault (page invalid in page table)
4. Trap to kernel
5. Convert virtual address to file + offset
6. Allocate page frame
   - Evict page if needed
7. Initiate disk block read into page frame
8. Disk interrupt when DMA complete
9. Mark page as valid
10. Resume process at faulting instruction
11. TLB miss
12. Page table walk to fetch translation
13. Execute instruction

Evicting a Page Frame

- Select old page to evict
- Find all page table entries that refer to old page
  - If page frame is shared
- Set each page table entry to invalid
- Remove any TLB entries
  - Copies of now invalid page table entry
- Write changes on page back to disk, if necessary

Caching

- Assignment: where do you put the data?
- **Replacement: who do you kick out?**
  - Random: pros? cons?
  - FIFO
  - MIN
  - LRU
  - LFU
  - Approximating LRU
- Problems with Caching

First-In-First-Out (FIFO) Algorithm

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

4 frames (4 pages in memory at a time per process):

FRAMES				*time*	Request	Result
				0	1	miss
1				*1*	2	miss
1	2			*2*	3	miss
1	2	3		*3*	4	miss
1	2	3	4	*4*	1	hit
1	2	3	4	*5*	2	hit
1	2	3	4	*6*	5	miss
5	2	3	4	*7*	1	miss
5	1	3	4	*8*	2	miss
5	1	2	4	*9*	3	miss
5	1	2	3	*10*	4	miss
4	1	2	3	*11*	5	miss
4	5	2	3	*12*		

← contents of frames at time of reference

f marks arrival time of frame f

10 page faults ☹

Optimal Replacement Algorithm (MIN)

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

4 frames (4 pages in memory at a time per process):

FRAMES				*time*	Request	Result
				0	1	miss
1				*1*	2	miss
1	2			*2*	3	miss
1	2	3		*3*	4	miss
1	2	3	4	*4*	1	hit
1	2	3	4	*5*	2	hit
1	2	3	4	*6*	5	miss
1	2	3	5	*7*	1	hit
1	2	3	5	*8*	2	hit
1	2	3	5	*9*	3	hit
1	2	3	5	*10*	4	miss
1	2	3	5	*11*	5	miss
1	2	3	5	*12*		

7 page faults 😊

(is 7 actually good?)

Let's always use MIN! 🤔

← Which to kick out at *t=6* ?
MIN says the one you'll use
furthest in the future (here, 4)

use this as an upper-bound

Least Recently Used (LRU)

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

4 frames (4 pages in memory at a time per process):

FRAMES				*time*	Request	Result
				0	1	miss
1				*1*	2	miss
1	2			*2*	3	miss
1	2	3		*3*	4	miss
1	2	3	4	*4*	1	hit
1	2	3	4	*5*	2	hit
1	2	3	4	*6*	5	miss
1	2	5	4	*7*	1	hit
1	2	5	4	*8*	2	hit
1	2	5	4	*9*	3	miss
1	2	5	3	*10*	4	miss
1	2	4	3	*11*	5	miss
5	2	4	3	*12*		

8 page faults

← Which to kick out? LRU says the one used furthest back (here, 3)

← Which used furthest back? 4

← Which used furthest back? 5

← Which used furthest back? 1

Least Frequently Used (LFU)

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

4 frames (4 pages in memory at a time per process):

FRAMES				*time*	Request	Result	use count			
				0	1	miss				
1				*1*	2	miss	1			
1	2			*2*	3	miss	1	1		
1	2	3		*3*	4	miss	1	1	1	
1	2	3	4	*4*	1	hit	1	1	1	1
1	2	3	4	*5*	2	hit	2	1	1	1
1	2	3	4	*6*	5	miss	2	2	1	1
1	2	5	4	*7*	1	hit	2	2	1	1
1	2	5	4	*8*	2	hit	3	2	1	1
1	2	5	4	*9*	3	miss	3	3	1	1
1	2	5	3	*10*	4	miss	3	3	1	1
1	2	4	3	*11*	5	miss	3	3	1	1
1	2	4	5	*12*			3	3	1	1

8 page faults

← Which to kick out? 3

(let's break ties with FIFO)

← Which to kick out? 4

← Which to kick out? 5

← Which to kick out? 3

How to implement LRU?

In software, use a linked list:

- every hit moves you to the front of the list
- evict from the back of the list

In hardware:

- 2-way set-associative cache?
- 4-way set-associative cache?
- List of all your frames in memory?
    - *big* list, costly timestamps 😢
- per frame **`use bit`**

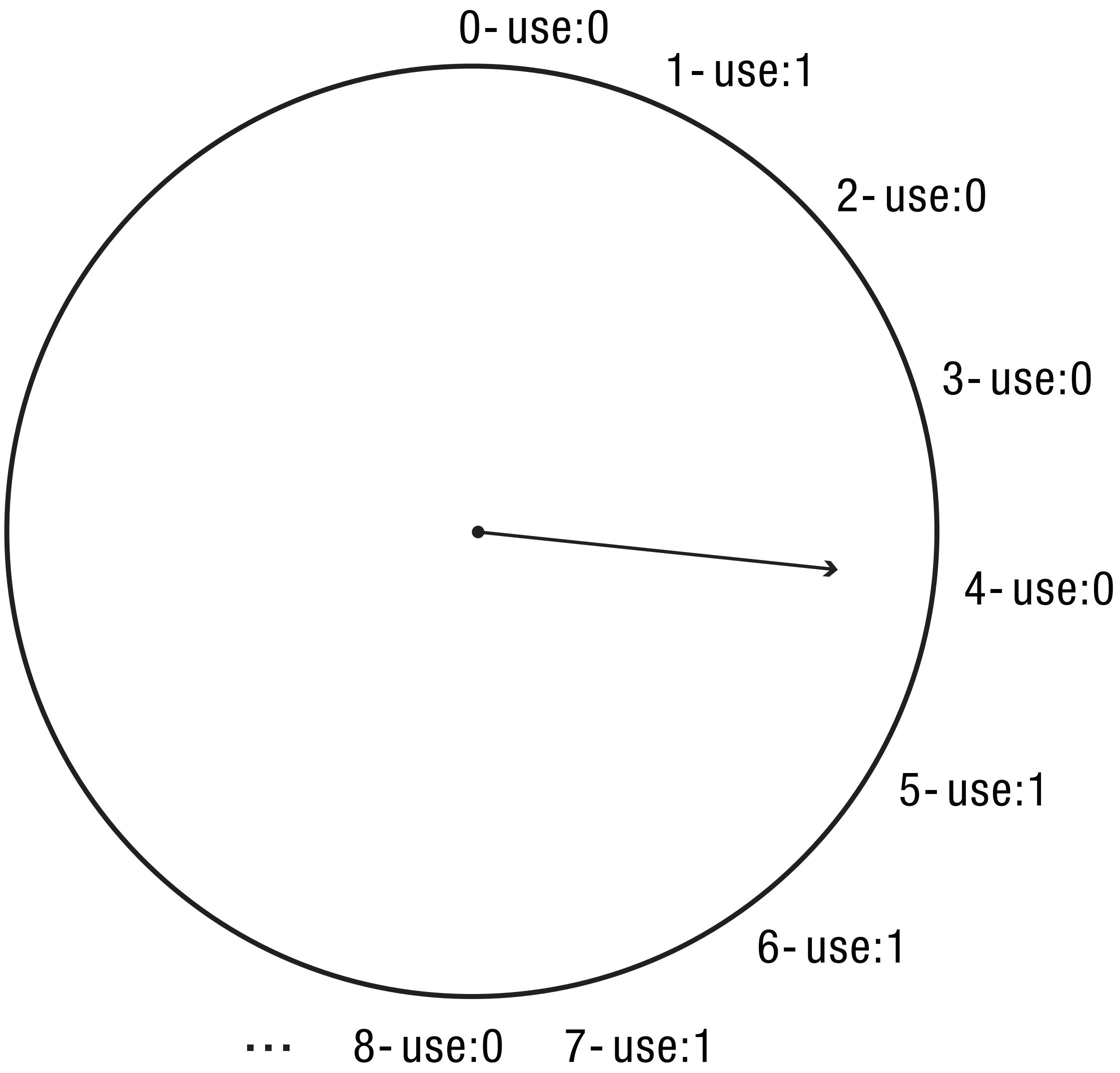
Clock Algorithm: Not Recently Used

Approximating LRU*

Periodically, sweep through all pages

- Used? Clear use bit
- Unused? reclaim
    - update core map
    - invalidate page table
    - write back if dirty
    - TLB shootdown
    - add to free list

Page Frames



*(*yes, LRU was already an approximation…)*

Clock Algorithm Problems

blue 1's were used after use bit was cleared by green hand

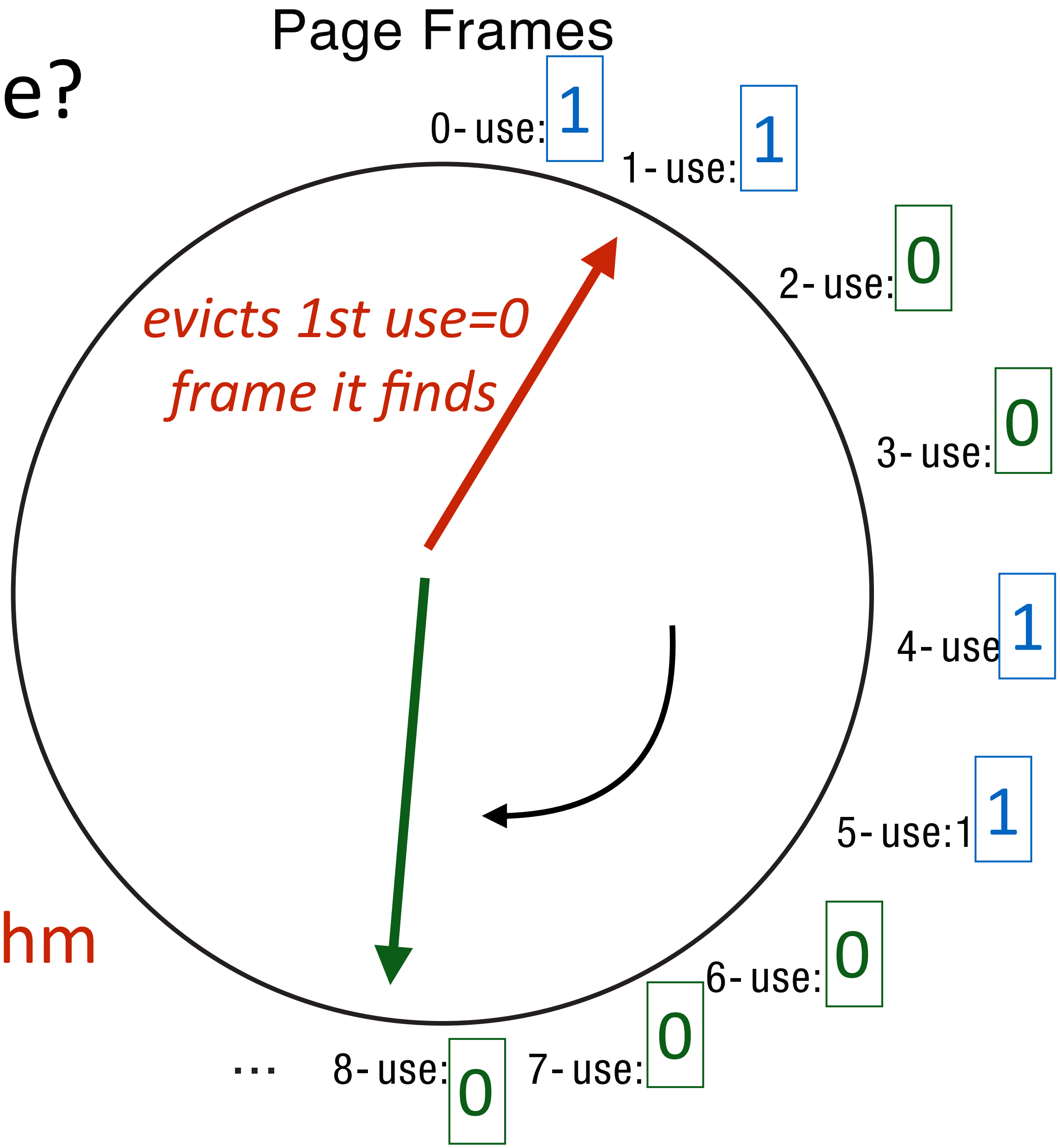
What if Memory is Large?

Leading edge clears use bit

- slowly clears history
- finds victim candidates

Trailing edge evicts pages with use bit set to 0

- fast: original clock algorithm
- slow: all pages look used



Caching

- Swapping & Paging
- Assigning a virtual page a physical frame
- Replacement Policies
- **Problems with Caching**
  - Ineffectiveness
  - Fairness

Exploiting LRU Eviction Policies

```
static char *workingSet;  // memory program wants to acquire
static int soFar;         // num pages program has so far
static sthread_t refreshThread;

// Thread touches pages in memory, keeping them recently used
void refresh () {
    int i;

    while (1) {
        // Keep every page in memory recently used.
        for (i = 0; i < soFar; i += PAGESIZE)
            workingSet[i] = 0;
    }
}

int main (int argc, char **argv) {
    // Allocate a giant array.
    workingSet = malloc(ARRAYSIZE);
    soFar = 0;

    // Create a thread to keep our pages in memory
    thread_create(&refreshThread, refresh, 0);

    // Touch every page to bring it into memory
    for (; soFar < ARRAYSIZE; soFar += PAGESIZE)
        workingSet[soFar] = 0;

    // Now that everything is in memory, run computation...
}
```