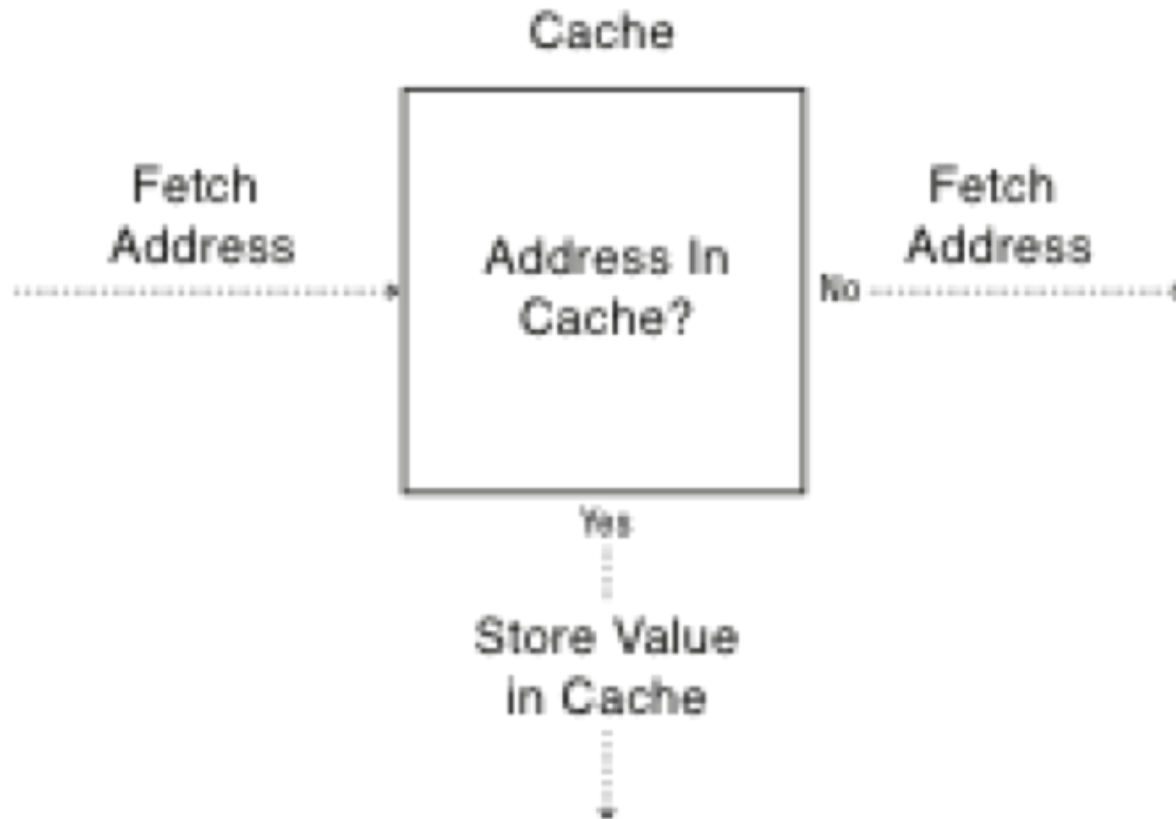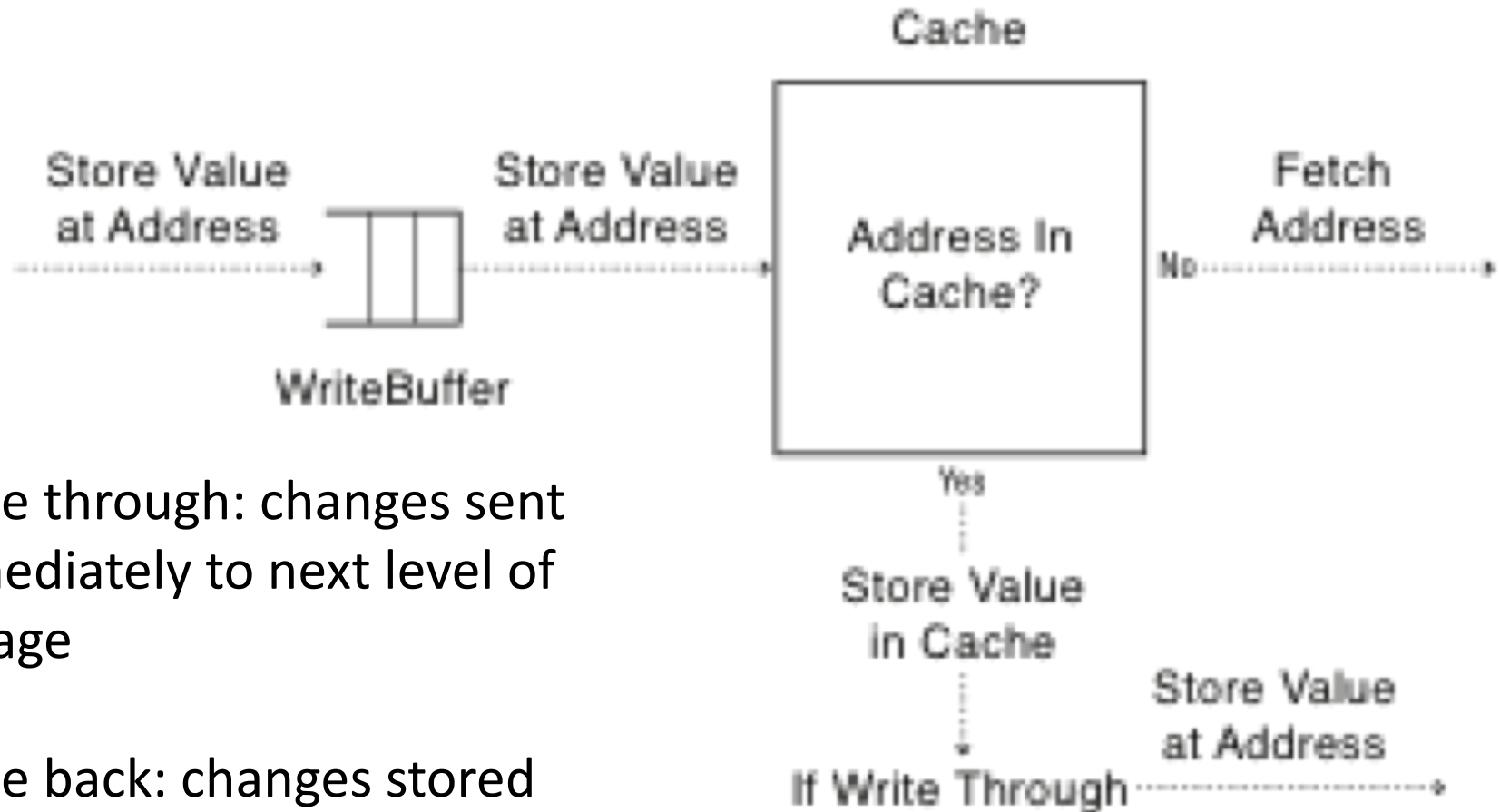# Caching and Demand-Paged Virtual Memory

Definitions

- Cache
  - Copy of data that is faster to access than the original
  - Hit: if cache has copy
  - Miss: if cache does not have copy
- Cache block
  - Unit of cache storage (multiple memory locations)
- Temporal locality
  - Programs tend to reference the same memory locations multiple times
  - Example: instructions in a loop
- Spatial locality
  - Programs tend to reference nearby locations
  - Example: data in a loop

Cache Concept (Read)

Cache Concept (Write)

Cache

Store Value at Address ┄┄┄> WriteBuffer ┄┄┄> Store Value at Address ┄┄┄> **Address In Cache?** ──No┄┄┄> **Fetch Address**

Yes ┆ Store Value in Cache ┆ If Write Through ┄┄┄> Store Value at Address ┄┄┄>

Write through: changes sent immediately to next level of storage

Write back: changes stored in cache until cache block is replaced

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

i7 has 8MB as shared 3rd level cache; 2nd level cache is per-core

Main Points

- Can we provide the illusion of near infinite memory in limited physical memory?
  - Demand-paged virtual memory
  - Memory-mapped files
- How do we choose which page to replace?
  - FIFO, MIN, LRU, LFU, Clock
- What types of workloads does caching work for, and how well?
  - Spatial/temporal locality vs. Zipf workloads

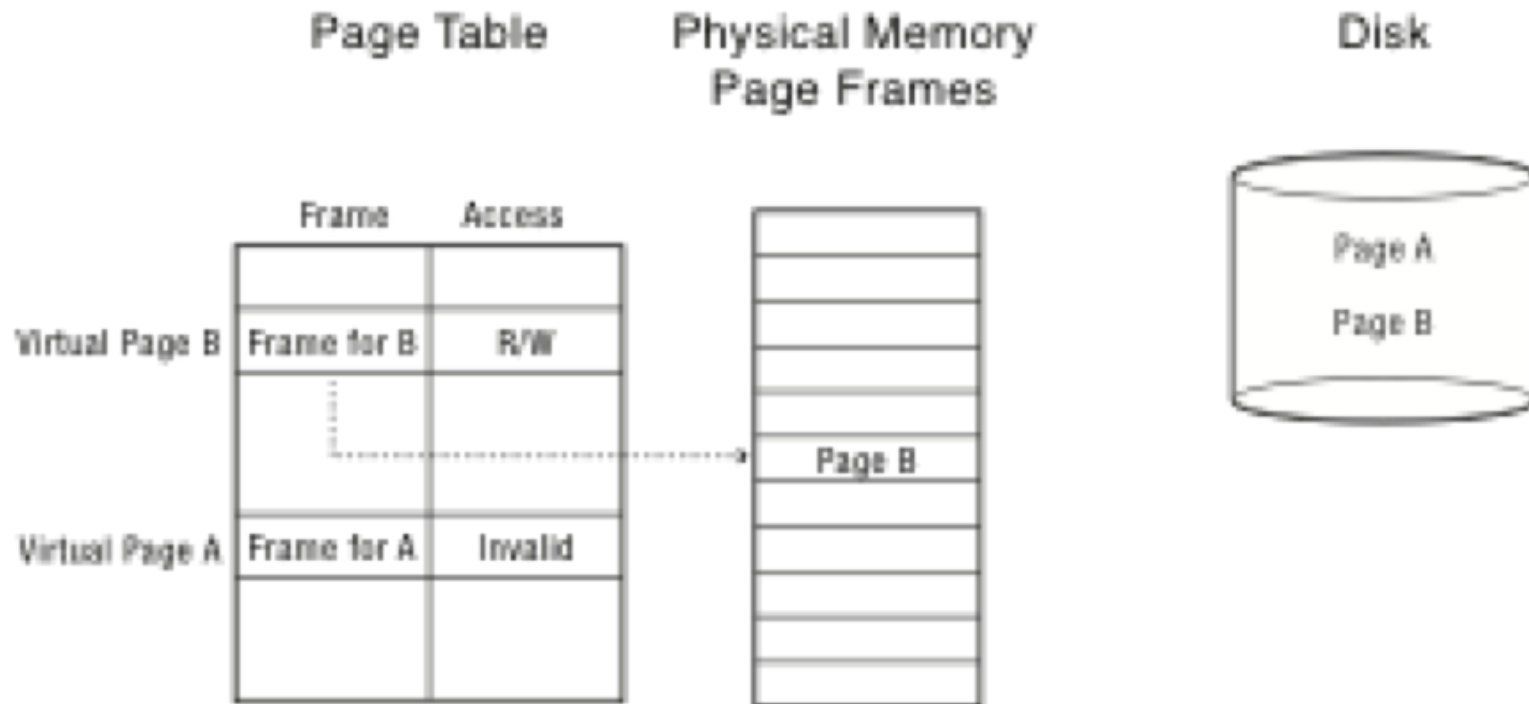# Hardware address translation is a power tool

- Kernel trap on read/write to selected addresses
  - Copy on write
  - Fill on reference
  - Zero on use
  - Demand paged virtual memory
  - Memory mapped files
  - Modified bit emulation
  - Use bit emulation

Demand Paging (Before)
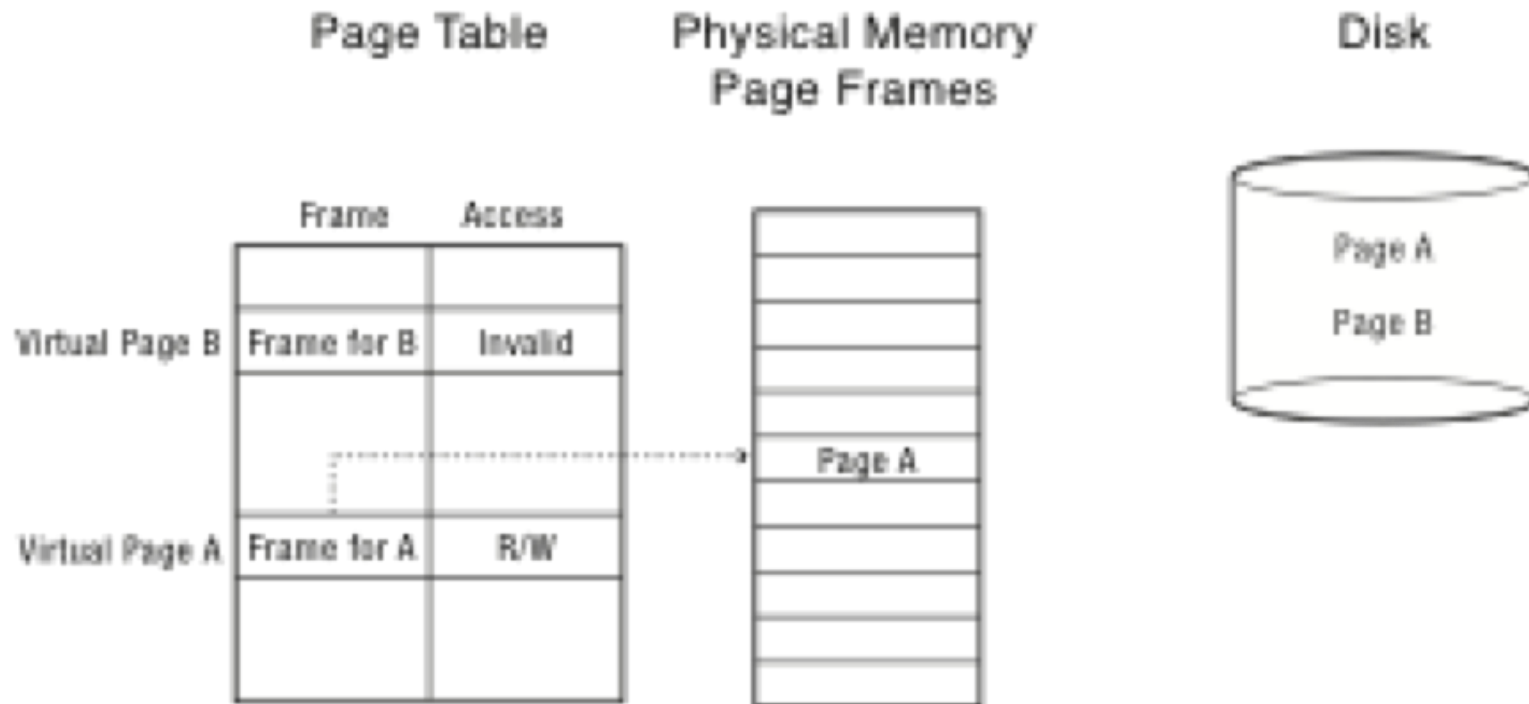
Demand Paging (After)

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
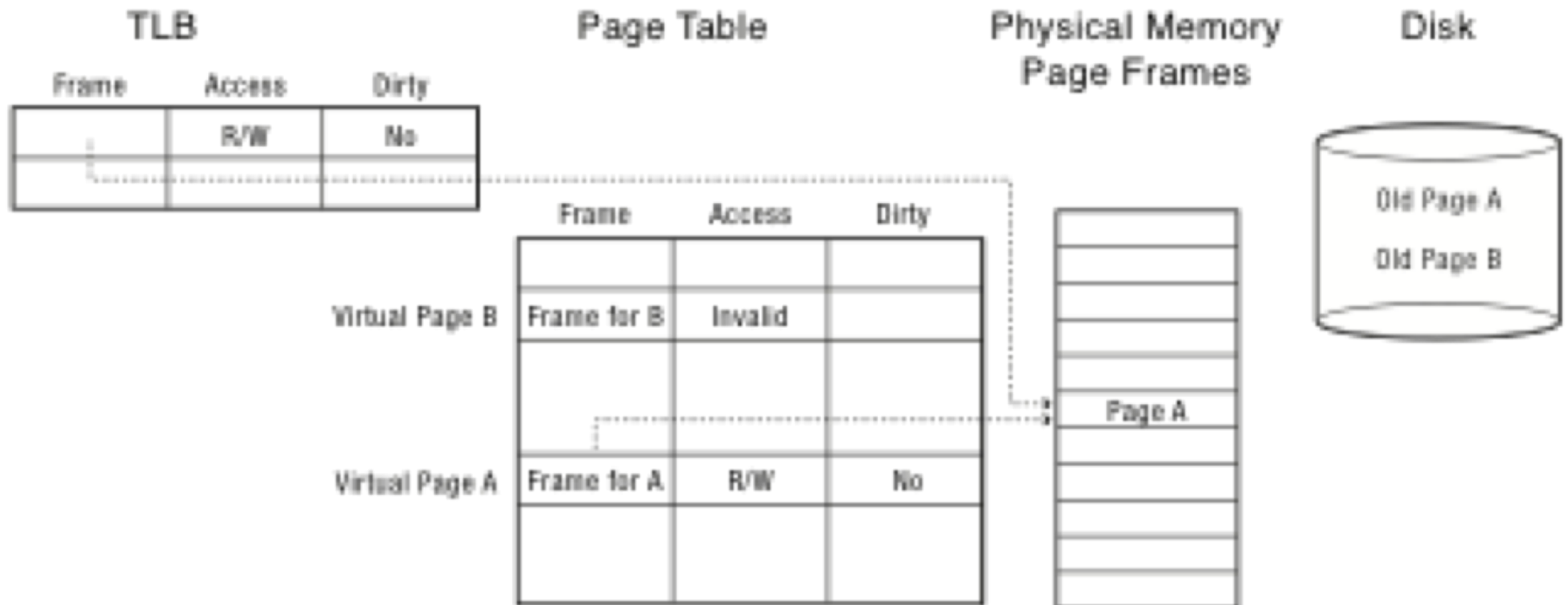13. Execute instruction

Allocating 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

How do we know if page has been modified?
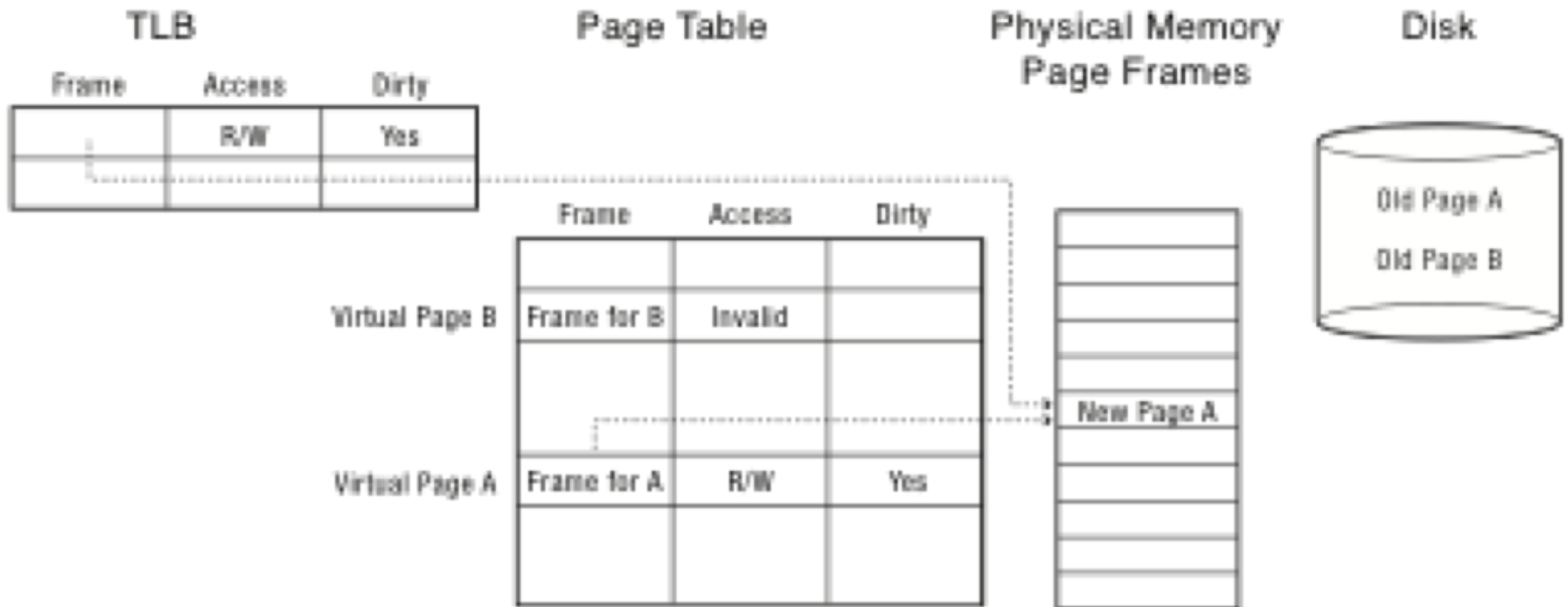
- Every page table entry has some bookkeeping
  - Has page been modified?
    - Set by hardware on store instruction
    - In both TLB and page table entry
  - Has page been recently used?
    - Set by hardware on in page table entry on every TLB miss
- Bookkeeping bits can be reset by the OS kernel
  - When changes to page are flushed to disk
  - To track whether page is recently used

Keeping Track of Page Modifications (Before)

TLB

Frame	Access	Dirty
	R/W	No

Page Table

	Frame	Access	Dirty
Virtual Page B	Frame for B	Invalid	
Virtual Page A	Frame for A	R/W	No

Physical Memory Page Frames

Page A

Disk

Old Page A

Old Page B

Keeping Track of Page Modifications (After)

TLB

Frame	Access	Dirty
	R/W	Yes

Page Table

	Frame	Access	Dirty
Virtual Page B	Frame for B	Invalid	
Virtual Page A	Frame for A	R/W	Yes

Physical Memory Page Frames

New Page A

Disk

Old Page A

Old Page B

Virtual or Physical Dirty/Use Bits

- Most machines keep dirty/use bits in the page table entry
- Physical page is
  - Modified if *any* page table entry that points to it is modified
  - Recently used if *any* page table entry that points to it is recently used
- On MIPS, simpler to keep dirty/use bits in the core map
  - Core map: map of physical page frames

Emulating Modified/Use Bits w/ MIPS Software Loaded TLB

- MIPS TLB entries have an extra bit: modified/unmodified
  - Trap to kernel if no entry in TLB, or if write to an unmodified page
- On a TLB read miss:
  - If page is clean, load TLB entry as read-only; if dirty, load as rd/wr
  - Mark page as recently used
- On a TLB write to an unmodified page:
  - Kernel marks page as modified in its page table
  - Reset TLB entry to be read-write
  - Mark page as recently used
- On TLB write miss:
  - Kernel marks page as modified in its page table
  - Load TLB entry as read-write
  - Mark page as recently used

Emulating a Modified Bit
(Hardware Loaded TLB)

- Some processor architectures do not keep a modified bit per page
  - Extra bookkeeping and complexity
- Kernel can emulate a modified bit:
  - Set all clean pages as read-only
  - On first write to page, trap into kernel
  - Kernel sets modified bit, marks page as read-write
  - Resume execution
- Kernel needs to keep track of both
  - Current page table permission (e.g., read-only)
  - True page table permission (e.g., writeable, clean)

Emulating a Recently Used Bit (Hardware Loaded TLB)

- Some processor architectures do not keep a recently used bit per page
  - Extra bookkeeping and complexity
- Kernel can emulate a recently used bit:
  - Set all recently unused pages as invalid
  - On first read/write, trap into kernel
  - Kernel sets recently used bit
  - Marks page as read or read/write
- Kernel needs to keep track of both
  - Current page table permission (e.g., invalid)
  - True page table permission (e.g., read-only, writeable)

Models for Application File I/O

- Explicit read/write system calls
  - Data copied to user process using system call
  - Application operates on data
  - Data copied back to kernel using system call
- Memory-mapped files
  - Open file as a memory segment
  - Program uses load/store instructions on segment memory, implicitly operating on the file
  - Page fault if portion of file is not yet in memory
  - Kernel brings missing blocks into memory, restarts process

Advantages to Memory-mapped Files

- Programming simplicity, esp for large files
  - Operate directly on file, instead of copy in/copy out
- Zero-copy I/O
  - Data brought from disk directly into page frame
- Pipelining
  - Process can start working before all the pages are populated
- Interprocess communication
  - Shared memory segment vs. temporary file

From Memory-Mapped Files to Demand-Paged Virtual Memory

- Every process segment backed by a file on disk
  - Code segment -> code portion of executable
  - Data, heap, stack segments -> temp files
  - Shared libraries -> code file and temp data file
  - Memory-mapped files -> memory-mapped files
  - When process ends, delete temp files
- Unified memory management across file buffer and process memory

Cache Replacement Policy

- On a cache miss, how do we choose which entry to replace?
  - Assuming the new entry is more likely to be used in the near future
  - In direct mapped caches, not an issue!

- Policy goal: reduce cache misses
  - Improve expected case performance
  - Also: reduce likelihood of very poor performance

A Simple Policy

- Random?
  - Replace a random entry

- FIFO?
  - Replace the entry that has been in the cache the longest time
  - What could go wrong?

FIFO in Action

Reference	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A				E				D				C		
2		B				A				E				D	
3			C				B				A				E
4				D				C				B			

Worst case for FIFO is if program strides through memory that is larger than the cache

MIN, LRU, LFU

- MIN
  - Replace the cache entry that will not be used for the longest time into the future
  - Optimality proof based on exchange: if evict an entry used sooner, that will trigger an earlier cache miss
- Least Recently Used (LRU)
  - Replace the cache entry that has not been used for the longest time in the past
  - Approximation of MIN
- Least Frequently Used (LFU)
  - Replace the cache entry used the least often (in the recent past)

LRU/MIN for Sequential Scan

LRU

Reference	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A				E				D				C		
2		B				A				E				D	
3			C				B				A				E
4				D				C				B			

MIN

Reference	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A					+					+				
2		B					+					+	C		
3			C					+	D					+	
4				D	E					+					+

LRU

Reference	A	B	A	C	B	D	A	D	E	D	A	E	B	A	C
1	A		+				+				+			+	
2		B			+								+		
3				C					E			+			
4						D		+		+					C

FIFO

Reference	A	B	A	C	B	D	A	D	E	D	A	E	B	A	C
1	A		+				+		E						
2		B			+						A			+	
3				C								+	B		
4						D		+		+					C

MIN

Reference	A	B	A	C	B	D	A	D	E	D	A	E	B	A	C
1	A		+				+				+			+	
2		B			+								+		C
3				C					E			+			
4						D		+		+					

Belady's Anomaly

FIFO (3 slots)

Reference	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					+
2		B			A			+		C		
3			C			B			+		D	

FIFO (4 slots)

1	A				+		E				D	
2		B				+		A				E
3			C						B			
4				D						C		

Clock Algorithm: Estimating LRU

- Periodically, sweep through all pages

- If page is unused, reclaim

- If page is used, mark as unused

Page Frames

0 - use:0
1 - use:1
2 - use:0
3 - use:0
4 - use:0
5 - use:1
6 - use:1
· · ·   8 - use:0   7 - use:1
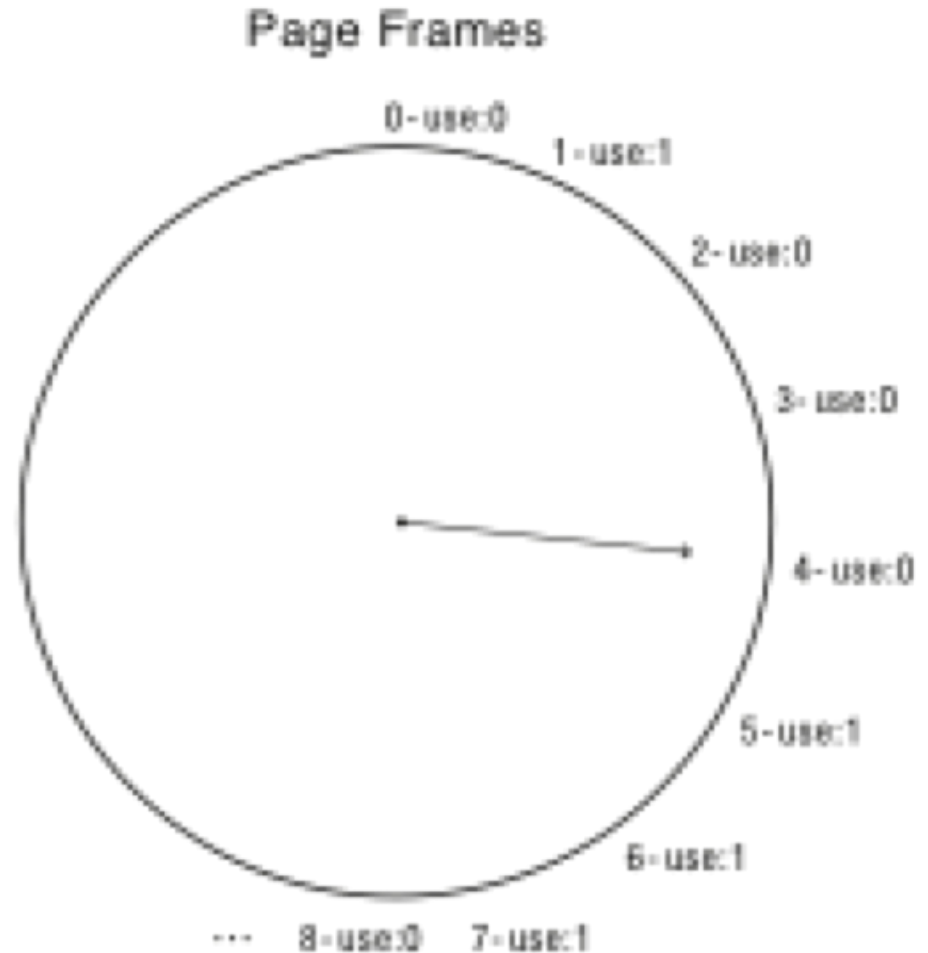
Nth Chance: Not Recently Used

- Instead of one bit per page, keep an integer
  - notInUseSince: number of sweeps since last use
- Periodically sweep through all page frames

```
if (page is used) {
    notInUseSince = 0;
} else if (notInUseSince < N) {
    notInUseSince++;
} else {
    reclaim page;
}
```

Implementation Note

- Clock and Nth Chance can run synchronously
  - In page fault handler, run algorithm to find next page to evict
  - Might require writing changes back to disk first
- Or asynchronously
  - Create a thread to maintain a pool of recently unused, clean pages
  - Find recently unused dirty pages, write mods back to disk
  - Find recently unused clean pages, mark as invalid and move to pool
  - On page fault, check if requested page is in pool!
  - If not, evict that page

Recap

- MIN is optimal
  - replace the page or cache entry that will be used farthest into the future

- LRU is an approximation of MIN
  - For programs that exhibit spatial and temporal locality

- Clock/Nth Chance is an approximation of LRU
  - Bin pages into sets of "not recently used"

Working Set Model

- Working Set: set of memory locations that need to be cached for reasonable cache hit rate

- Thrashing: when system has too small a cache

Cache Working Set
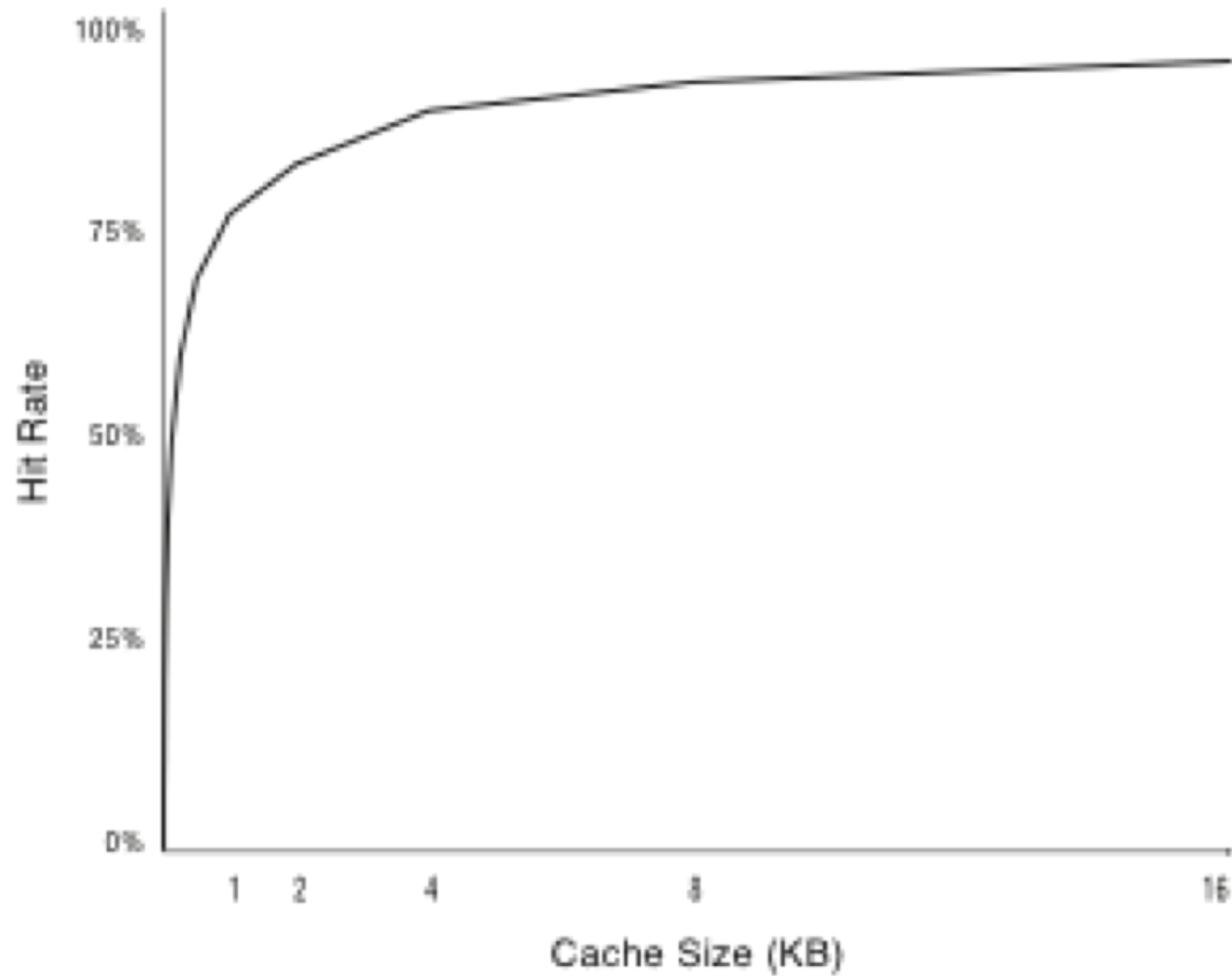
Phase Change Behavior
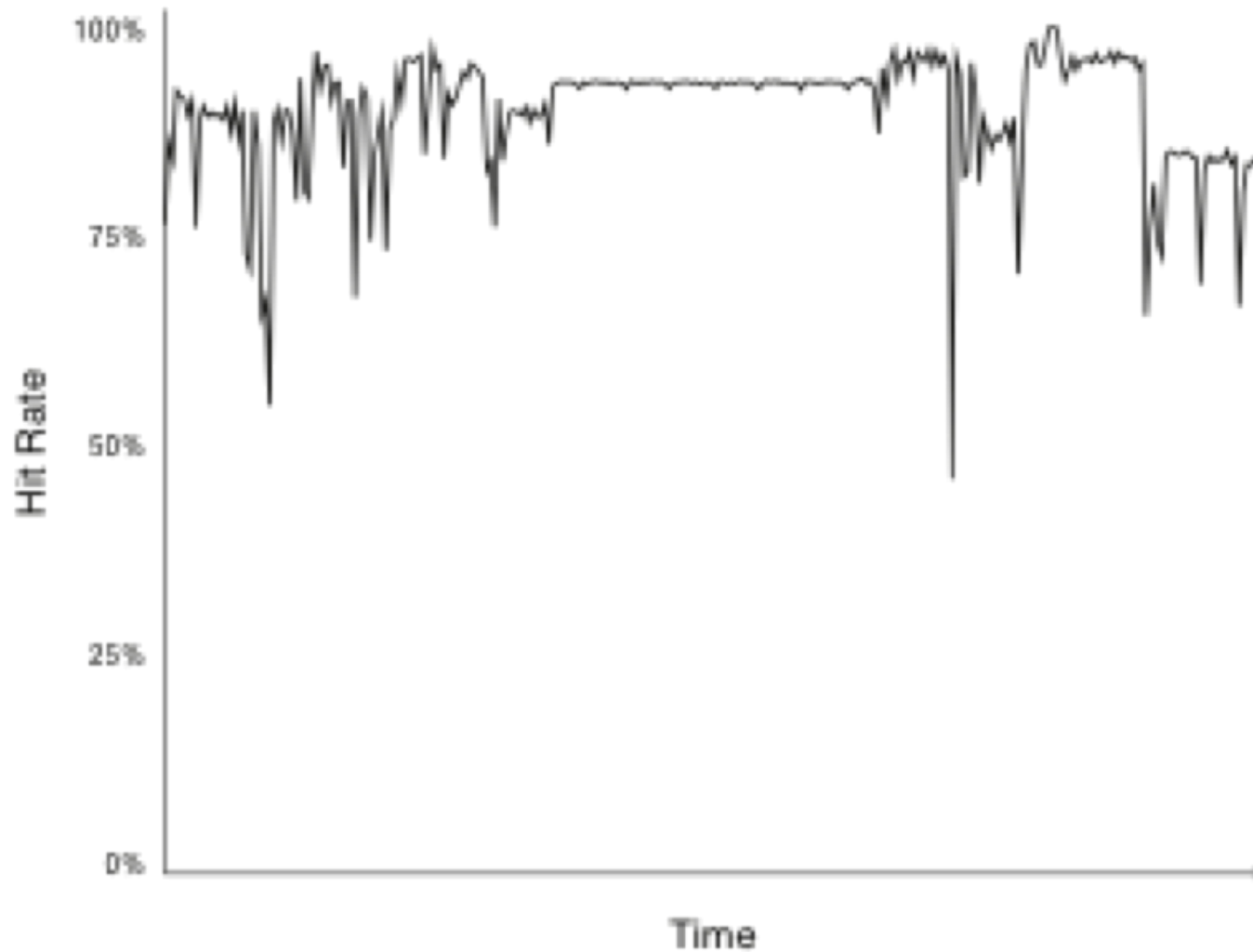
Question

- What happens to system performance as we increase the number of processes?
  - If the sum of the working sets > physical memory?

Zipf Distribution

- Caching behavior of many systems are not well characterized by the working set model

- An alternative is the Zipf distribution
  - Popularity ~ $1/k^c$, for kth most popular item, $1 < c < 2$

Zipf Distribution
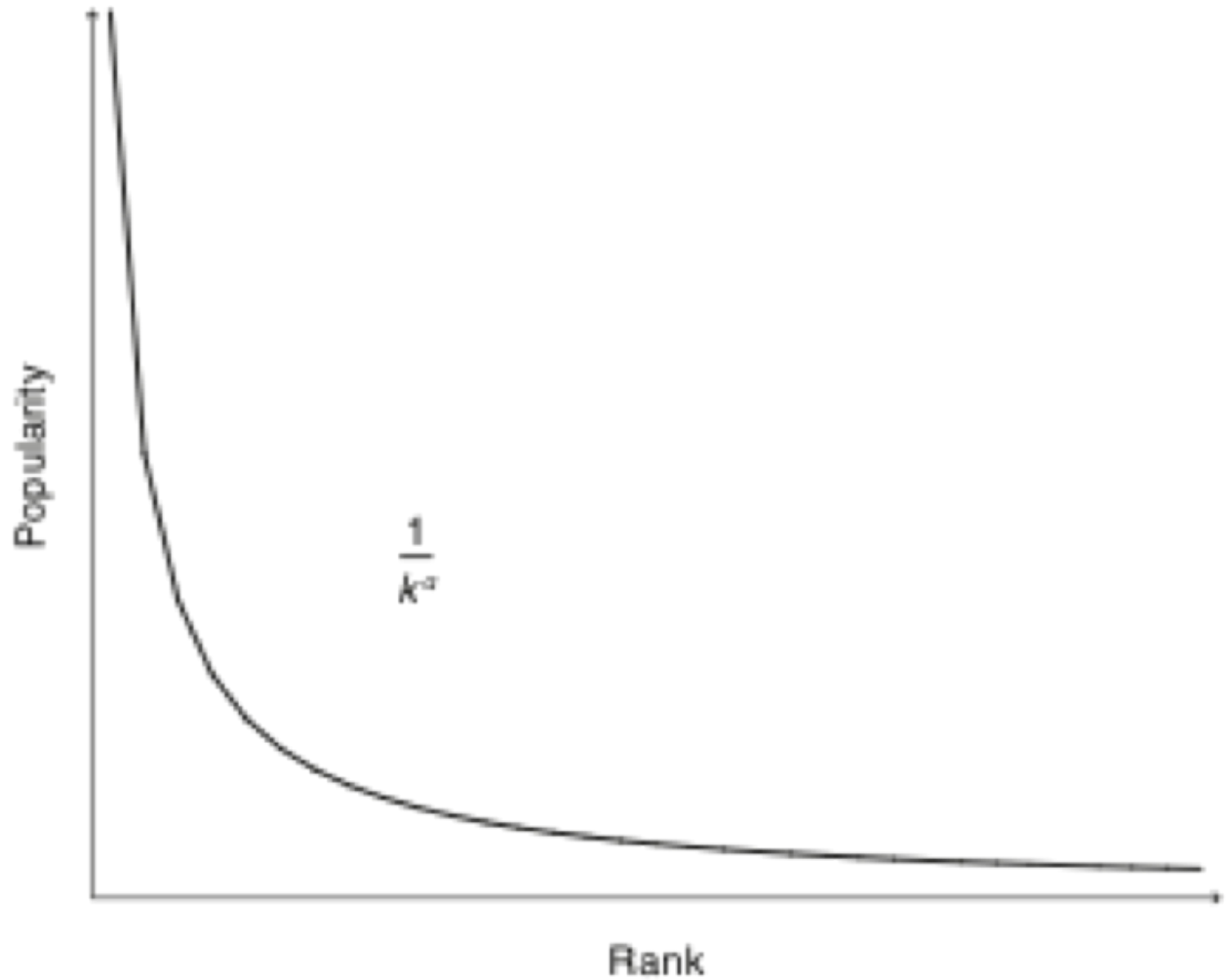
Zipf Examples

- Web pages
- Movies
- Library books
- Words in text
- Salaries
- City population
- …

Common thread: popularity is self-reinforcing
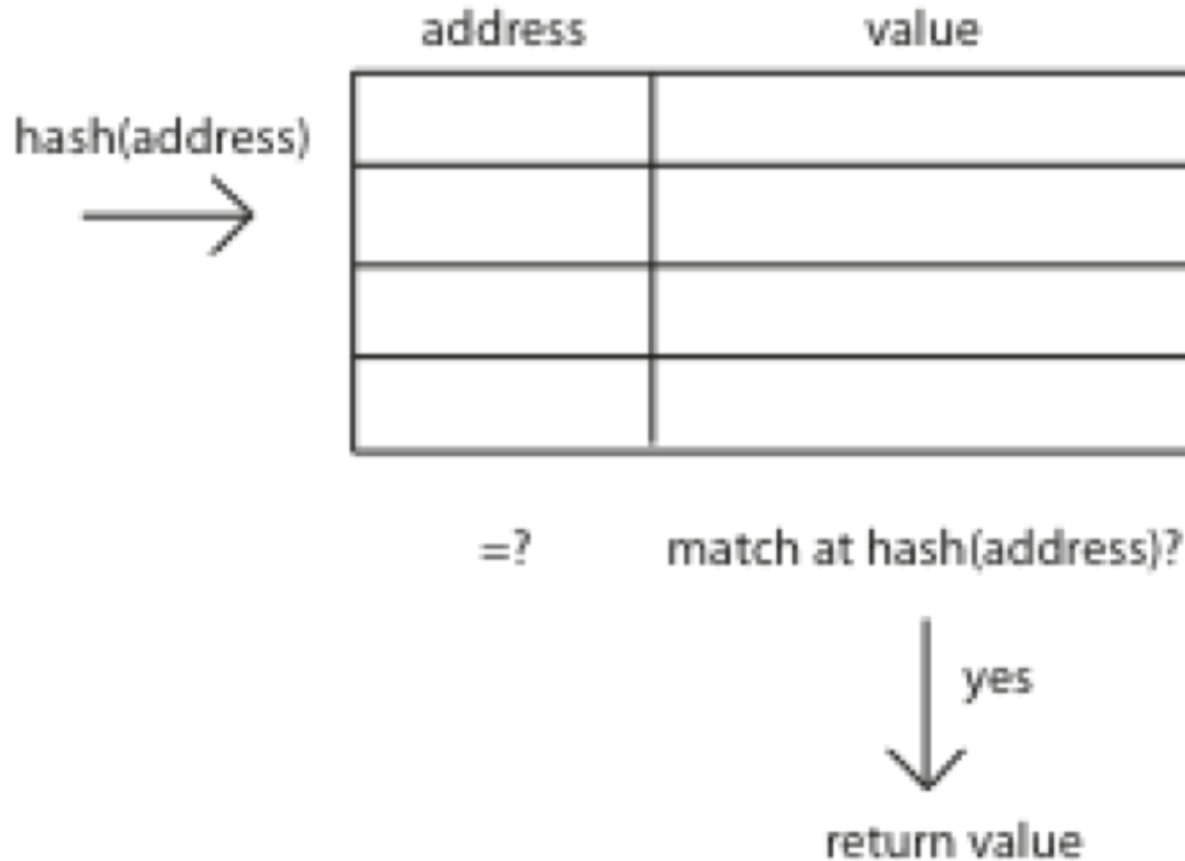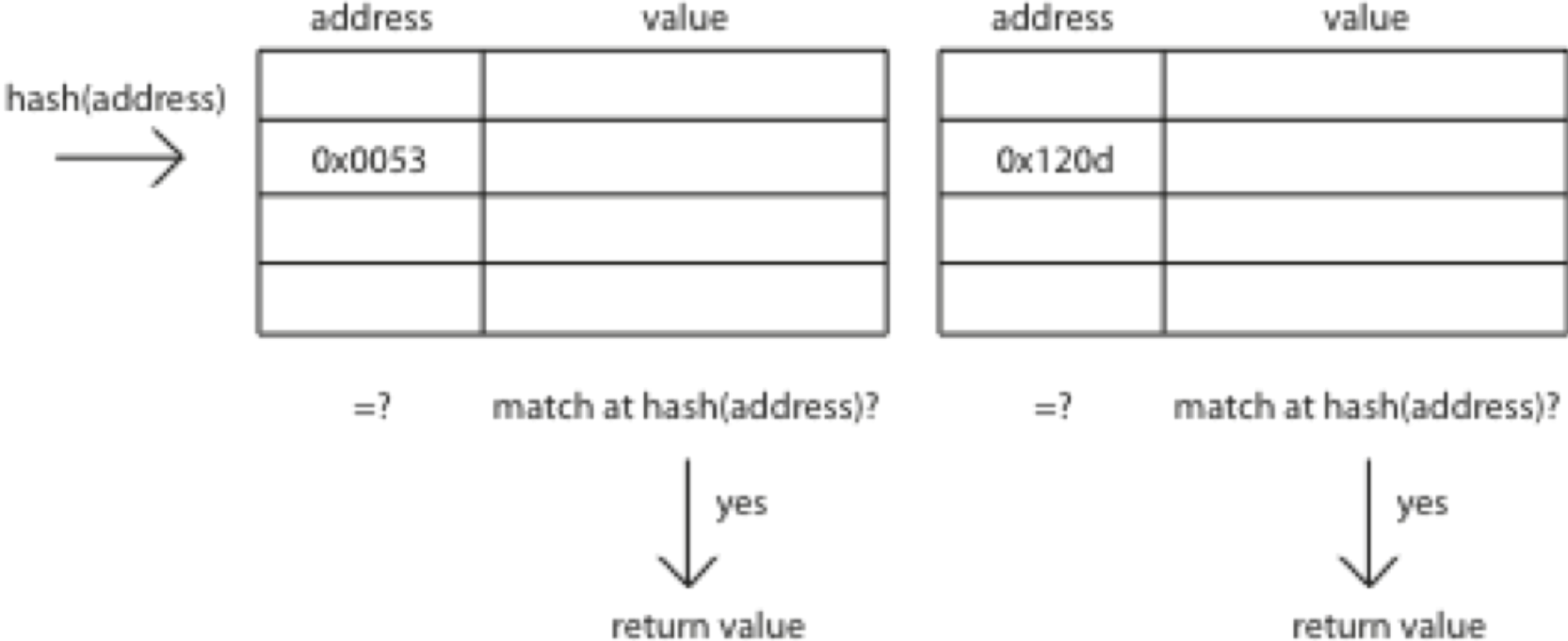
Zipf and Caching
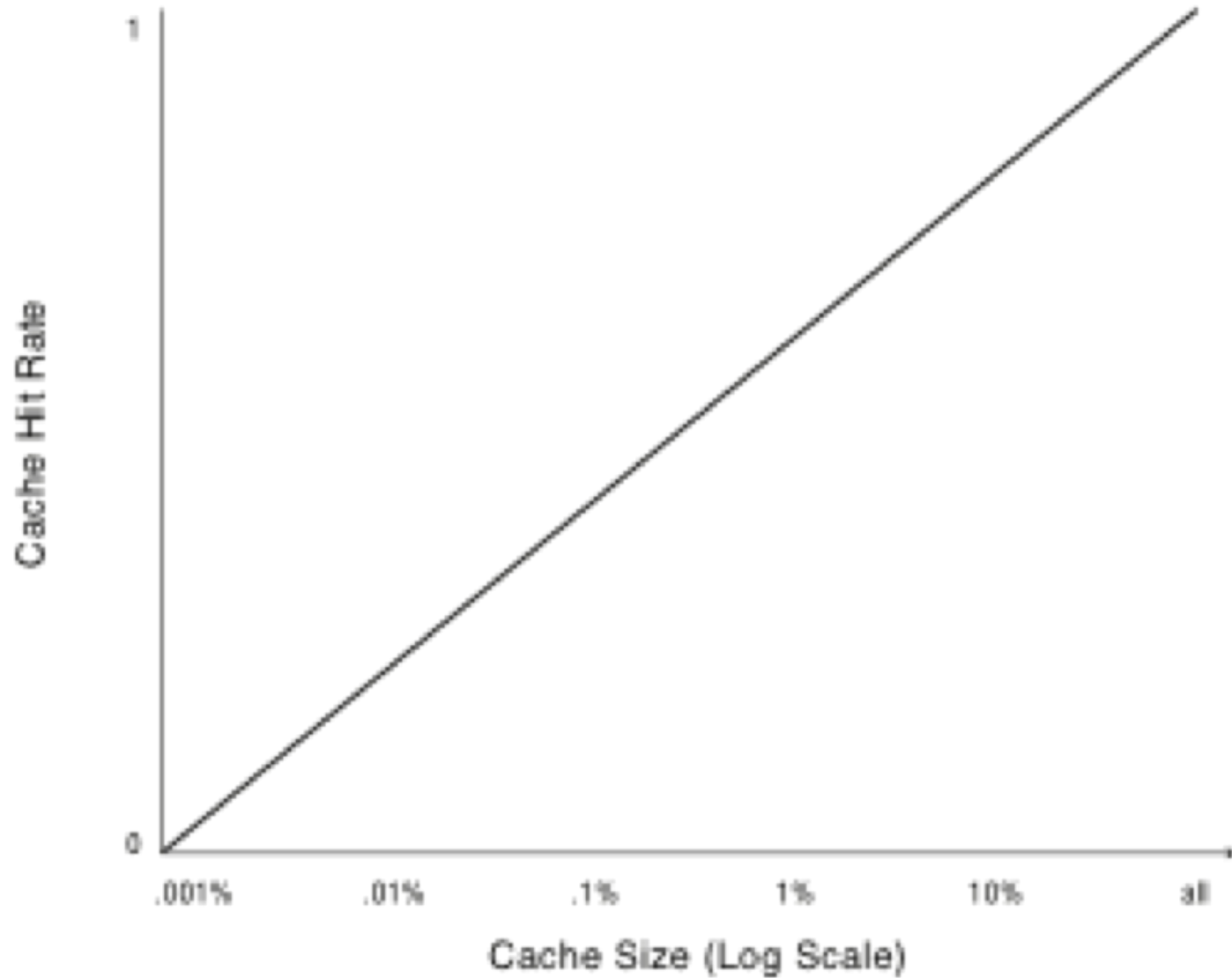
Cache Lookup: Fully Associative
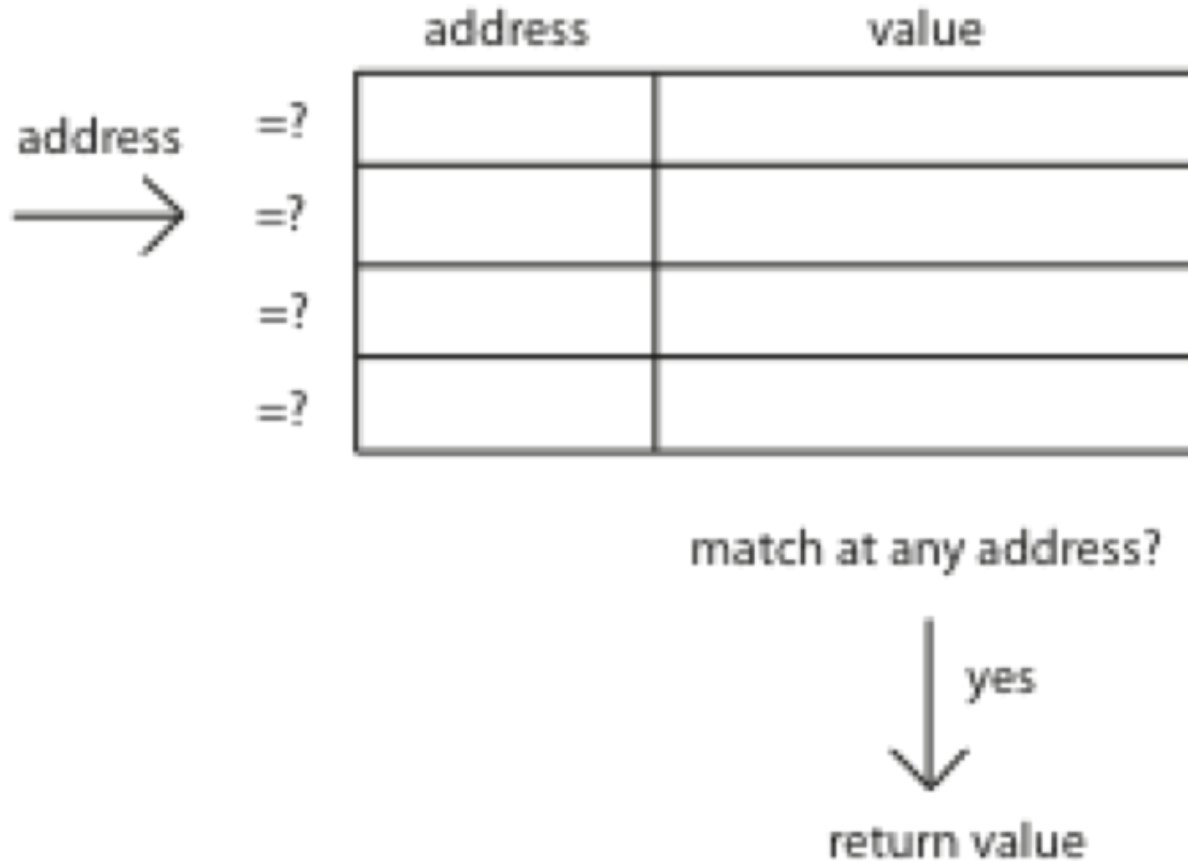
Cache Lookup: Direct Mapped

Cache Lookup: Set Associative

Page Coloring

- What happens when cache size >> page size?
  - Direct mapped or set associative
  - Multiple pages map to the same cache line
- OS page assignment matters!
  - Example: 8MB cache, 4KB pages
  - 1 of every 2K pages lands in same place in cache
- What should the OS do?

Page Coloring