CS 422/522 Design & Implementation
of Operating Systems

Lecture 14: Cache & Virtual Memory

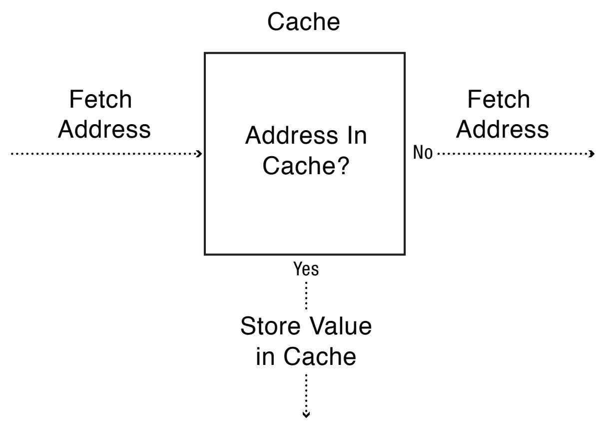
Zhong Shao
Dept. of Computer Science
Yale University

Acknowledgement: some slides are taken from previous versions of the CS422/522 lectures taught by Prof. Bryan Ford and Dr. David Wolinsky, and also from the official set of slides accompanying the OSPP textbook by Anderson and Dahlin.

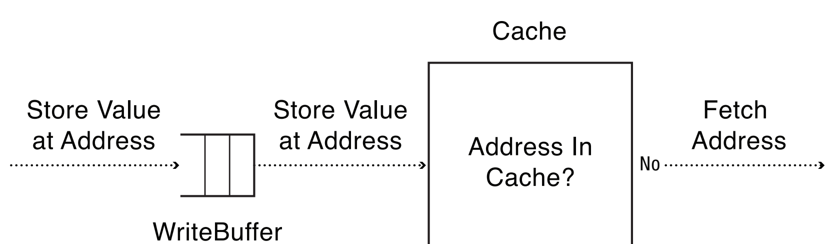
Definitions

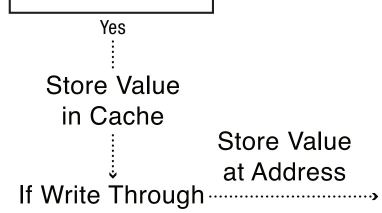
- Cache
  - Copy of data that is faster to access than the original
  - Hit: if cache has copy
  - Miss: if cache does not have copy
- Cache block
  - Unit of cache storage (multiple memory locations)
- Temporal locality
  - Programs tend to reference the same memory locations multiple times
  - Example: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

Cache concept (read)



Cache concept (write)





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

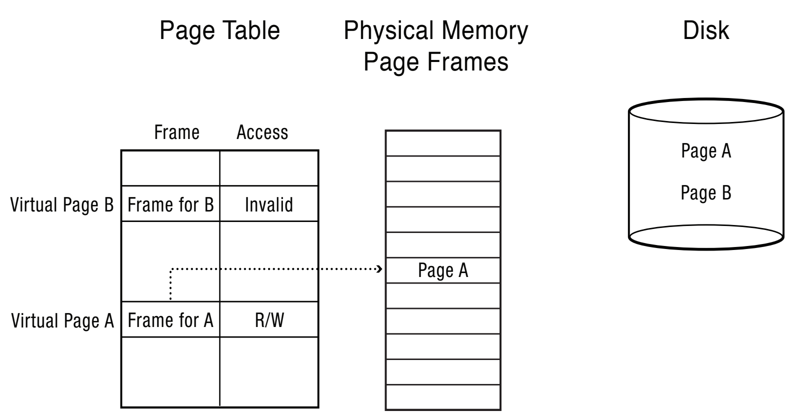
Main points

- 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

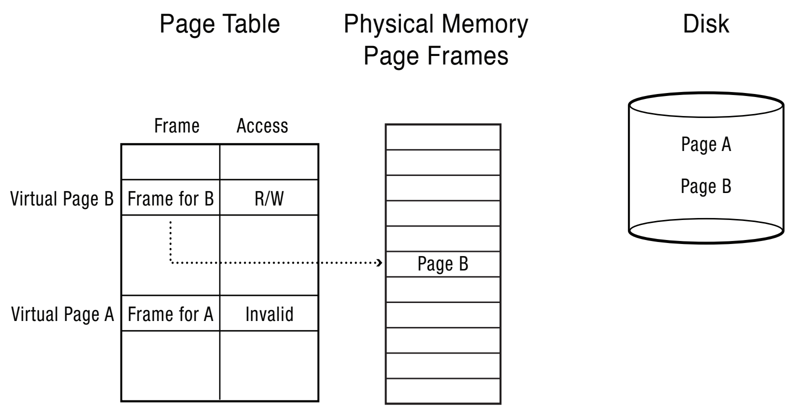
Hardware address translation is a power tool

- 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

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

Demand paging on MIPS (software TLB)

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

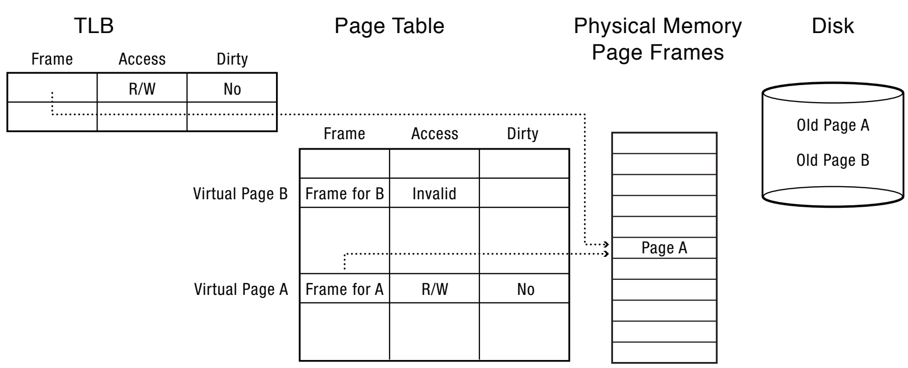
Allocating a page frame

- 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

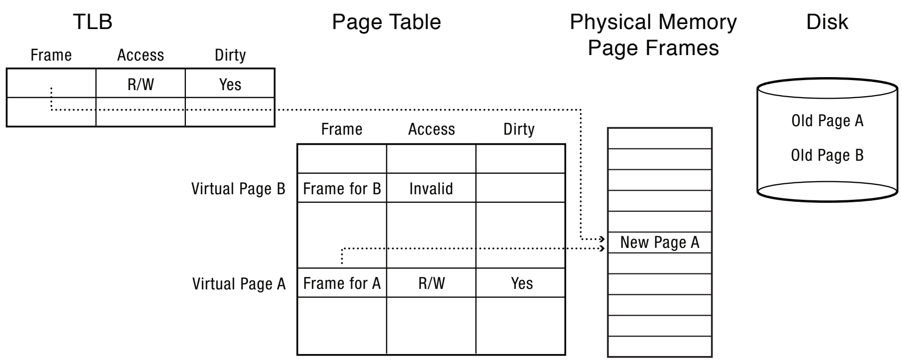
How do we know if page has been modified?

- 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 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)



Keeping track of page modifications (after)



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 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)

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

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

More page frames → fewer page faults?

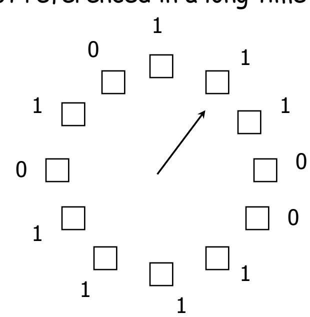
- Consider the following reference string with 3 page frames
  - FIFO replacement
  - **A, B, C, D, A, B, E,** A, B, **C, D,** E
  - **9 page faults!**
- Consider the same reference string with 4 page frames
  - FIFO replacement
  - **A, B, C, D,** A, B, **E, A, B, C, D, E**
  - 10 page faults
- This is called Belady's anomaly

Belady's anomaly (cont'd)

						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

- Approximate LRU
- Replace some old page, not the oldest unreferenced page
- Arrange physical pages in a circle with a clock hand
  - Hardware keeps "use bit" per physical page frame
  - Hardware sets "use bit" on each reference
  - If "use bit" isn't set, means not referenced in a long time
- On page fault:
  - Advance clock hand
  - Check "use bit"
  - If "1" clear, go on
  - If "0", replace page



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”

How many pages allocated to each process ?

- Each process needs minimum number of pages.
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages.
  - 2 pages to handle **from**.
  - 2 pages to handle **to**.
- Two major allocation schemes.
  - fixed allocation
  - priority allocation

Fixed allocation

- Equal allocation – e.g., if 100 frames and 5 processes, give each 20 pages.
- Proportional allocation – Allocate according to the size of process.

s_i = size of process p_i

$S = \sum s_i$

m = total number of frames

a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 64 \approx 5$$
$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Priority allocation

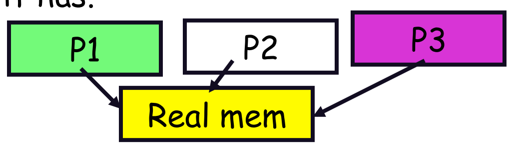
- Use a proportional allocation scheme using priorities rather than size.
- If process P_i generates a page fault,
  - select for replacement one of its frames.
  - select for replacement a frame from a process with lower priority number.

Global vs. local allocation

- Global replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another.
- Local replacement – each process selects from only its own set of allocated frames.

What to do when not enough memory?

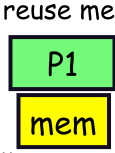
- Thrashing: processes on system require more memory than it has.



  * Each time one page is brought in, another page, whose contents will soon be referenced, is thrown out.
  * Processes will spend all of their time blocked, waiting for pages to be fetched from disk
  * I/O devices at 100% utilization but system not getting much useful work done
- **What we wanted: virtual memory the size of disk with access time of physical memory**
- **What we have: memory with access time = disk access**

Thrashing

- Process(es) "frequently"reference page not in mem
  - Spend more time waiting for I/O then getting work done
- Three different reasons
  - process doesn't reuse memory, so caching doesn't work
  - process does reuse memory, but it does not "fit"



  - individually, all processes fit and reuse memory, but too many for system.
- Which can we actually solve?

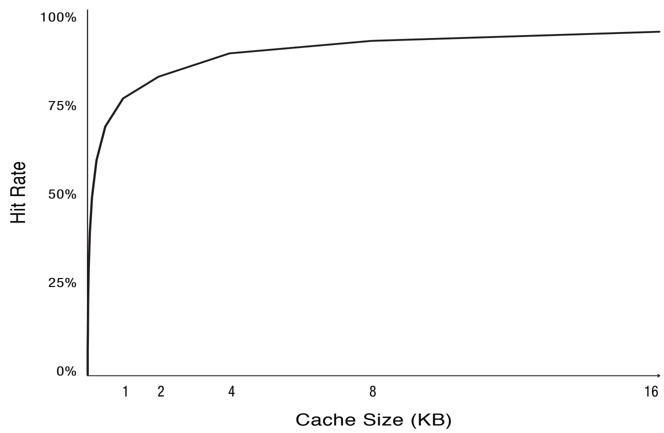
Making the best of a bad situation

- Single process thrashing?
  - If process does not fit or does not reuse memory, OS can do nothing except contain damage.
- System thrashing?
  - If thrashing arises because of the sum of several processes then adapt:
    - figure out how much memory each process needs
    - change scheduling priorities to run processes in groups whose memory needs can be satisfied (shedding load)
    - if new processes try to start, can refuse (admission control)

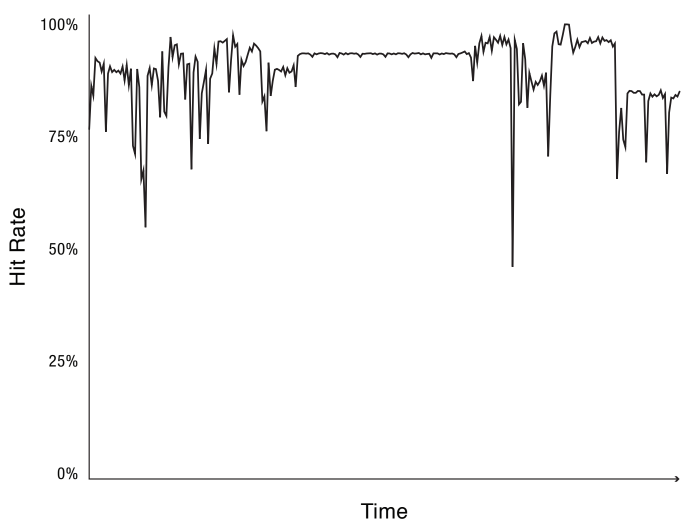
Working set model

- Working Set: set of memory locations that need to be cached for reasonable cache hit rate
- Size of working set = the important threshold
- The size may change even during execution of the same program.

Cache working set



Phase change behavior



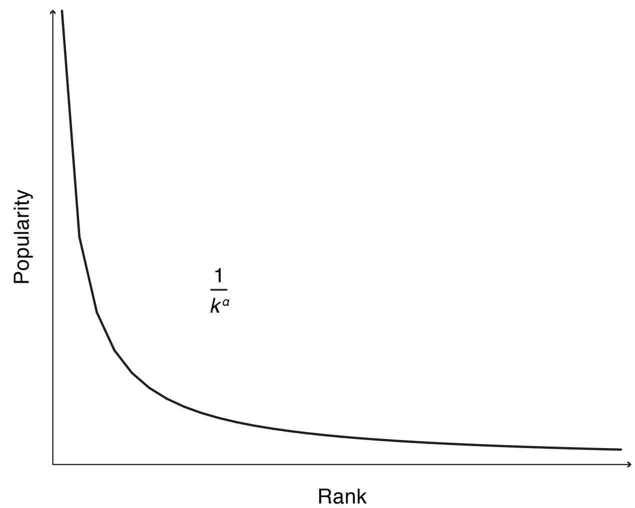
Question

- 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
  - "frequency inversely proportional to its rank in the frequency table (e.g., frequency word in English natural language)
    - Rank 1: "the" 7% (69,971 out of 1 million in "Brown Corpus")
    - Rank 2: "of" 3.5% (36,411 out of 1 million)
    - Rank 3: "and" 2.3% (28,852 out of 1 million)

Zipf distribution

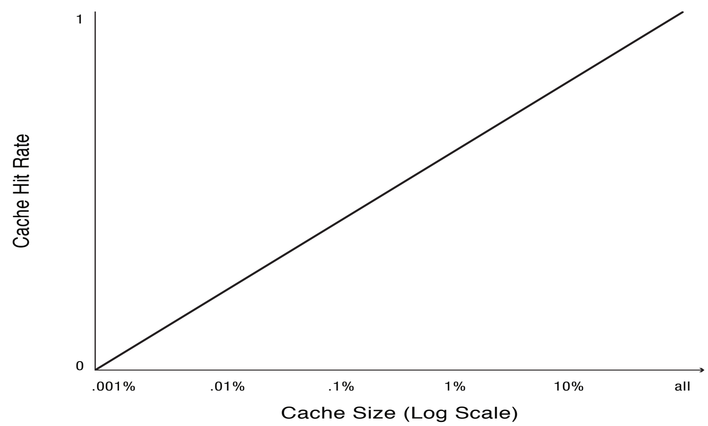


Zipf examples

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

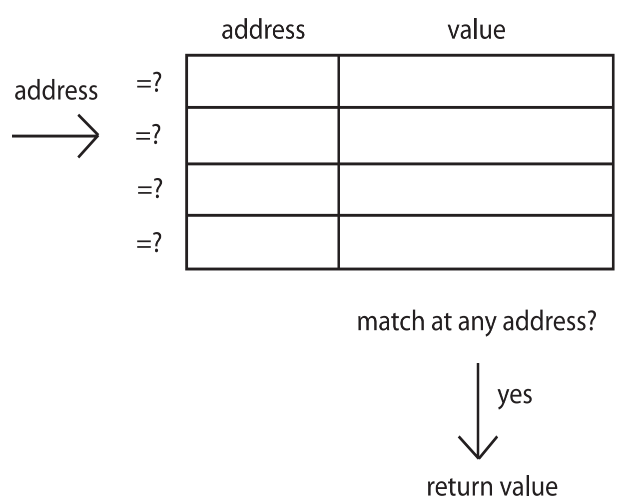
Common thread: popularity is self-reinforcing

Zipf and caching

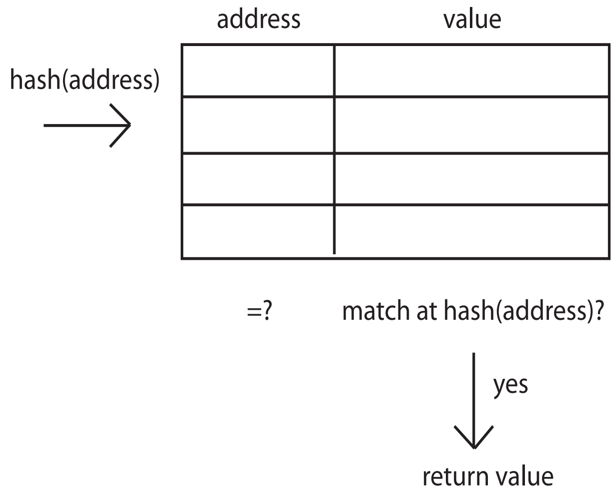


Increasing the cache size continues to improve cache hit rates, but with diminishing returns

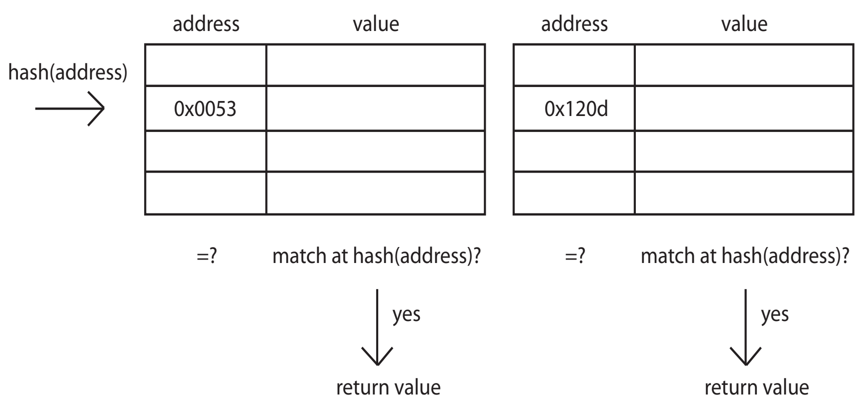
Cache lookup: fully associative



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