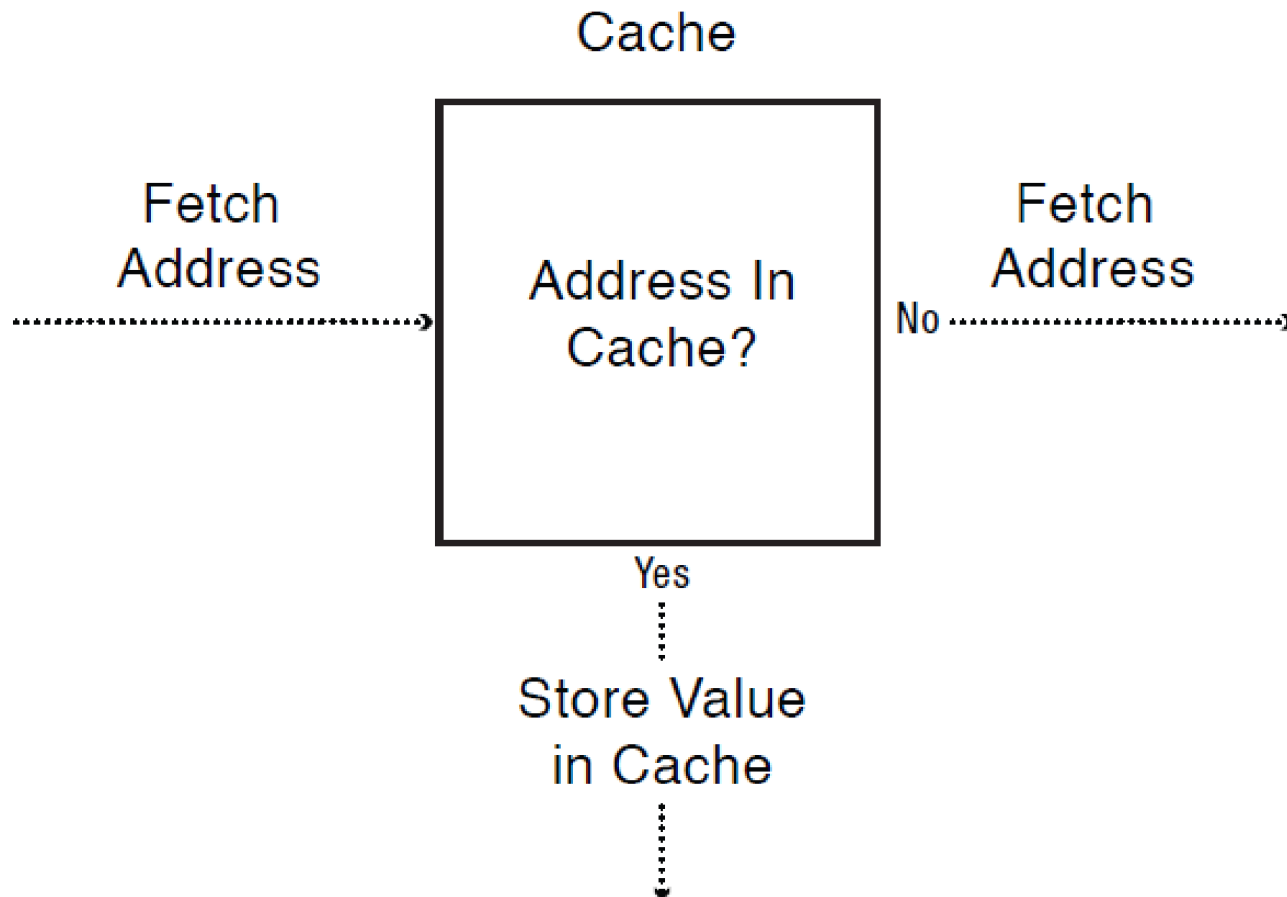


Caching and Demand-Paged Virtual Memory

Instructor: Youngjin Kwon

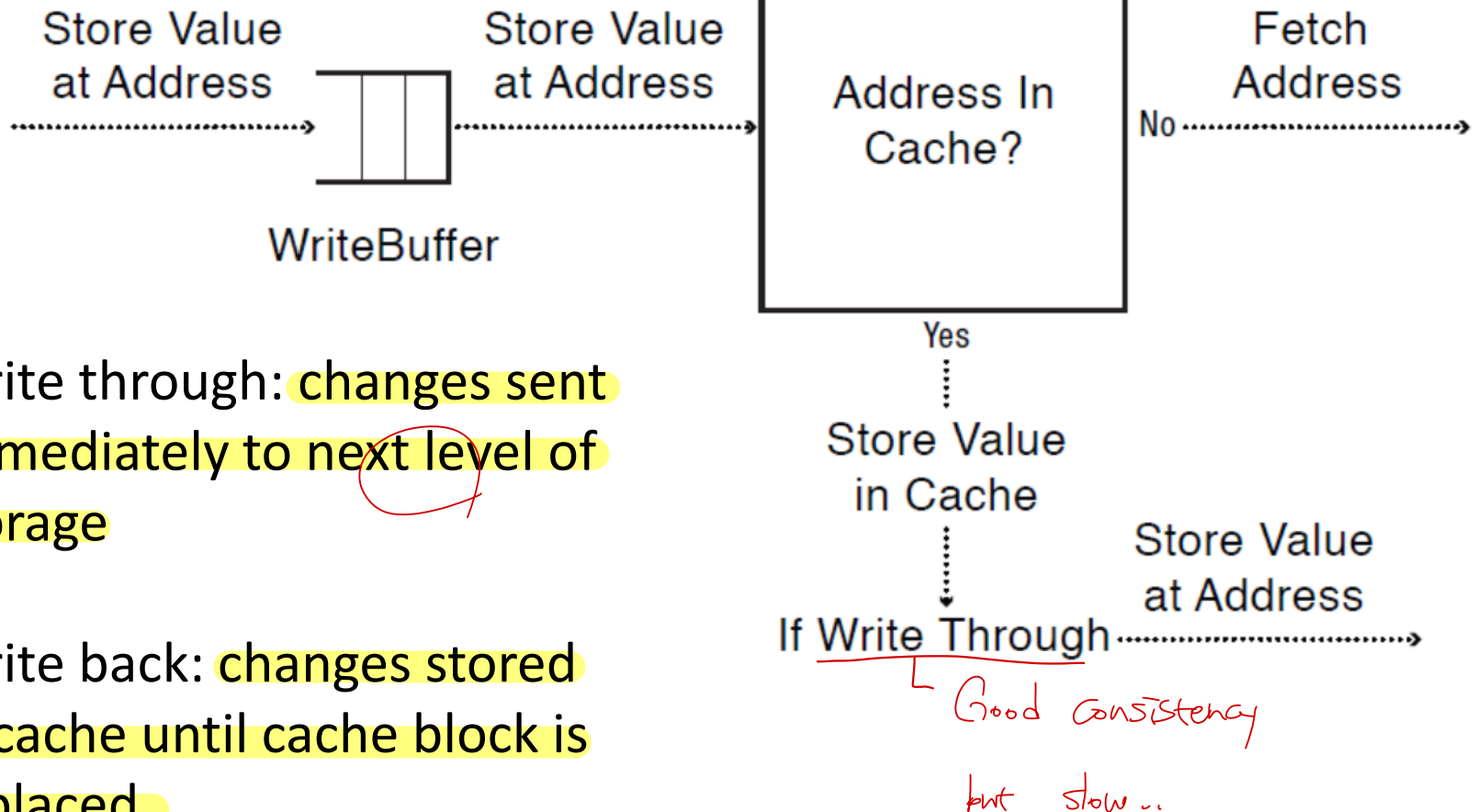
Cache Concept (Read)



Cache Concept (Write)

Read of 76 dominant 하기 때문

write through, backed 방식도 Cache가 있음.



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 μ s	100 TB
Local non-volatile memory	100 μ s	100 GB
Local disk <i>HDD usually</i>	10 ms	1 TB
Data center disk	10 ms	100 PB
Remote data center disk	200 ms	1 XB

Cache of memory

Cache of disk

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

Main Points

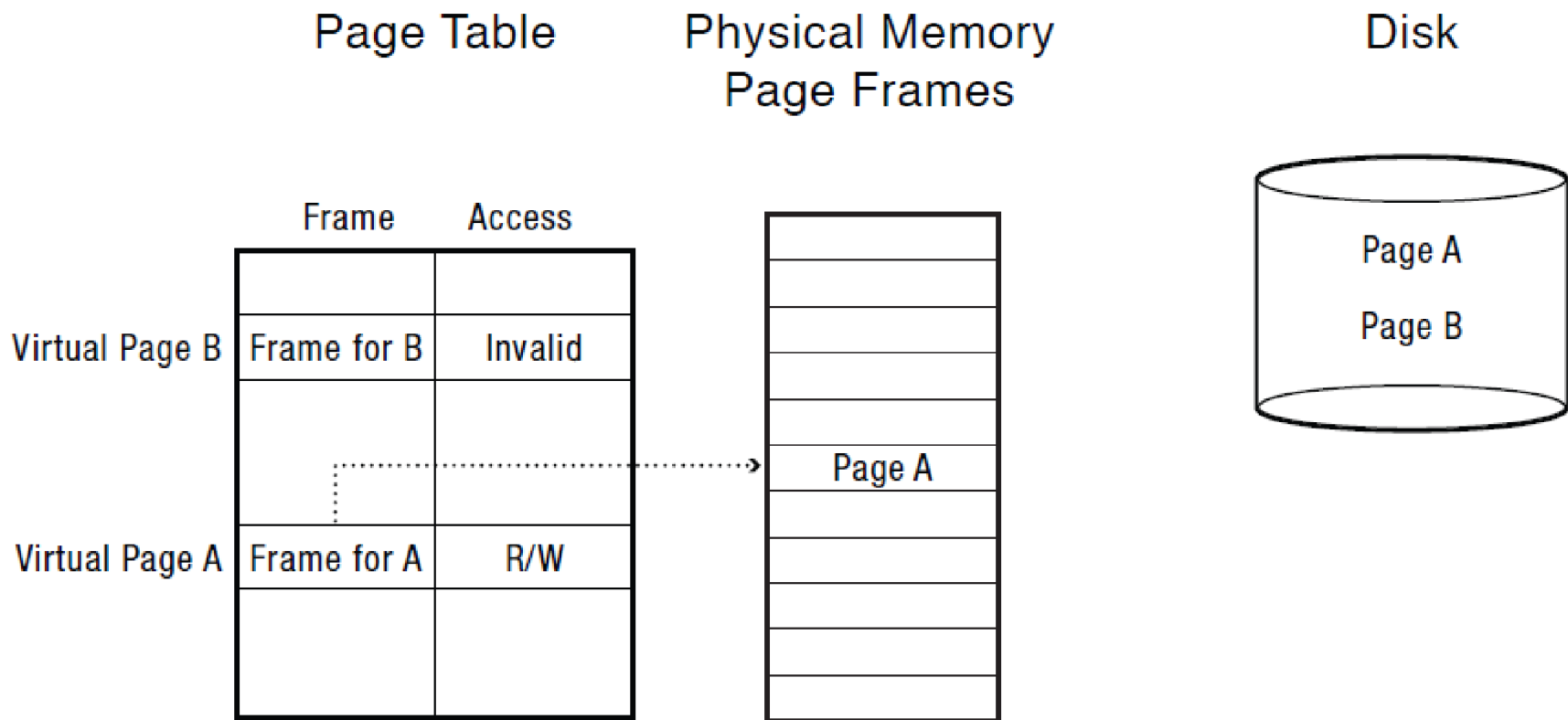
- Can we provide the illusion of ^{same as} ~~Almost~~ ^{storage size} (near infinite) memory in limited physical memory?
 - Demand-paged virtual memory
 - Memory-mapped files
- How do we choose which page to replace?
 - FIFO, MIN, LRU, Clock
- How to estimate working set of a process?



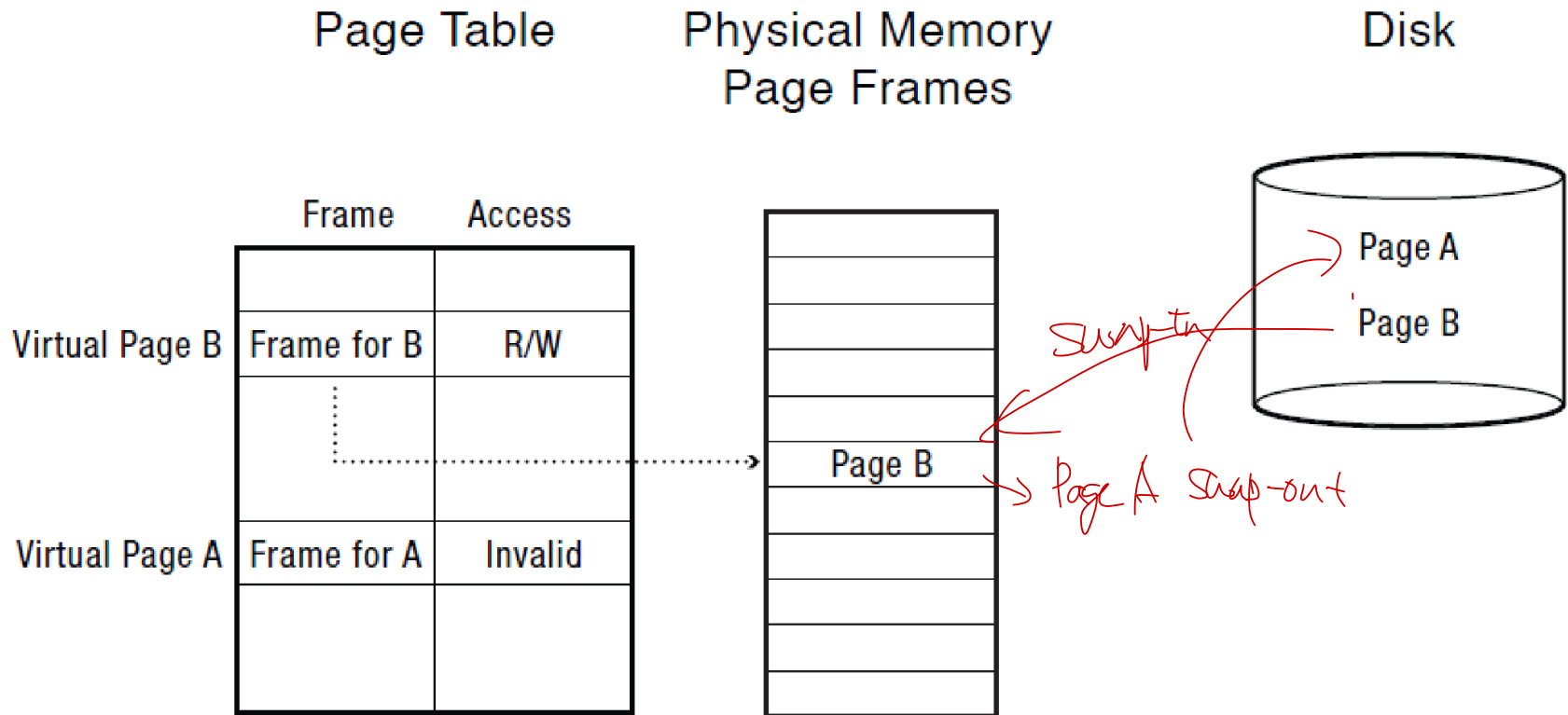
Hardware address translation is a power tool *중요한(?)*

- *Page fault*
(Kernel trap) on read/write to selected addresses
 - Copy on write
 - Fill on reference
 - Demand paged virtual memory
 - Memory mapped files

Demand Paging (Before)



Demand Paging (After)



Demand Paging on file-mapped memory


by hardware

1. TLB miss
2. Page table walk
3. Page fault (page invalid in page table) *→ page demand!*
4. Trap to kernel *→ OS*
5. Convert virtual address to file + offset *→ Filesystem*
6. Allocate page frame *→ PA*
 - 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 *(TLB set)*
13. Execute instruction

*↓
disk to PA swap in.*

Design Challenges (mechanism & policy)

- **How to resume a process after a fault?**

- Process might have been in the middle of an instruction!
-  – Save states during exception handling (we learned in the earlier lecture)
- To allocate a new page, what steps are required?

- **What to eject?**


- How to allocate physical pages amongst processes?
- Which of a particular process's pages to keep in memory?
- A poor choice can lead to horrible performance

To Allocate a New Page Frame

- Select old page to evict
- Find all page table entries that refer to old page
 - If page frame is shared
- What are next steps?

Write through
Write back

To Allocate a New 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  To prevent further access
- 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 (dirty bit)?
 - Set by hardware on store instruction
 - In both TLB and page table entry
 - Has page been recently used (use bit)?
 - 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 → Write through
 - To track whether page is recently used

Keeping Track of Page Modifications (Before)

Write on Page A



Frame	Access	Dirty
A	R/W	No

Page Table

Virtual Page B

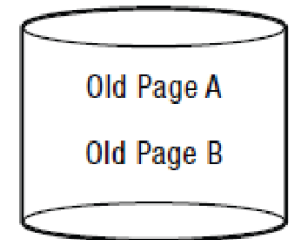
Virtual Page A

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

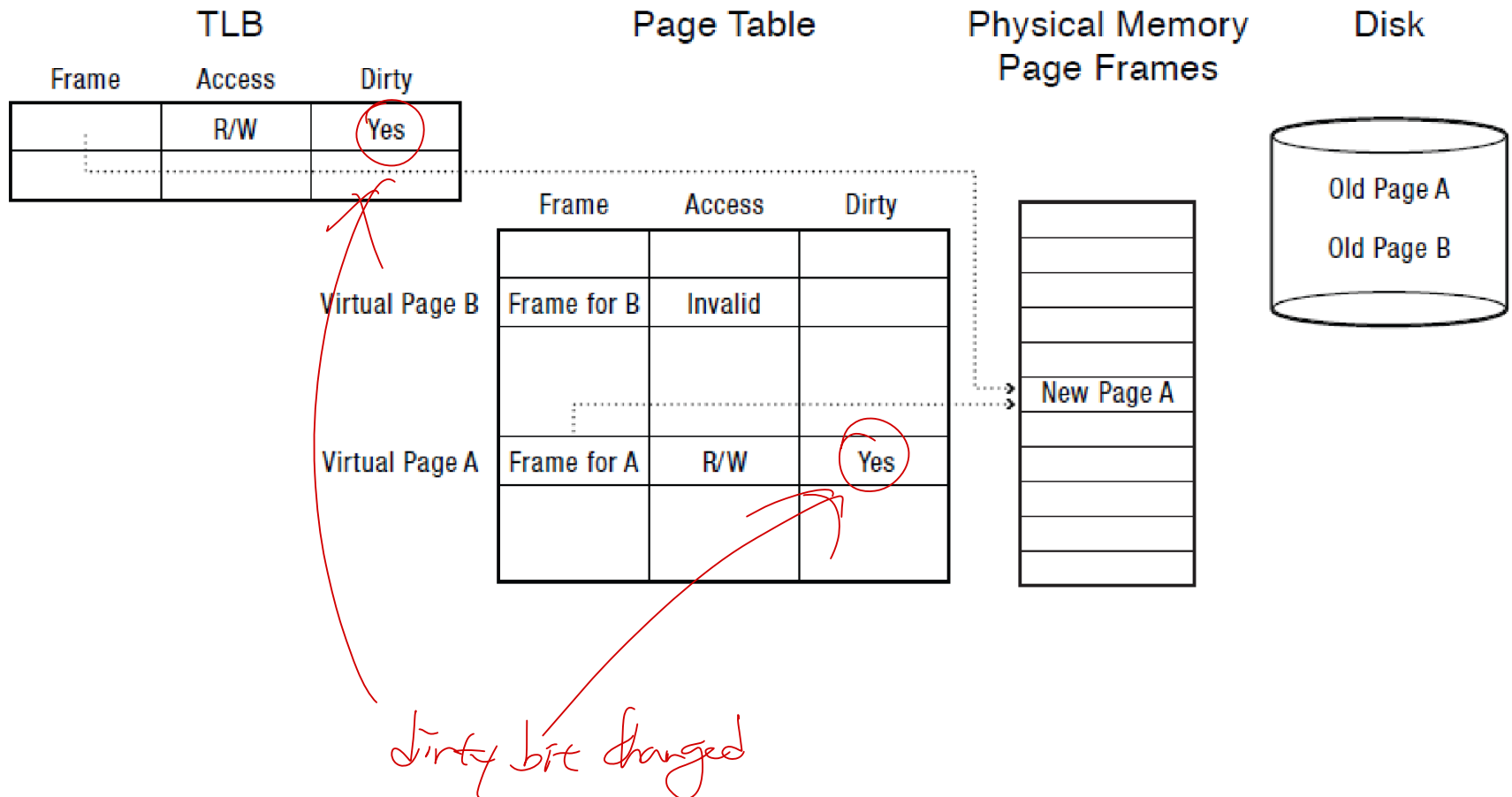
Physical Memory
Page Frames

Page A

Disk

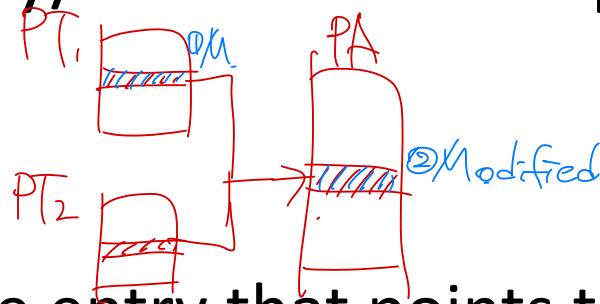


Keeping Track of Page Modifications (After)



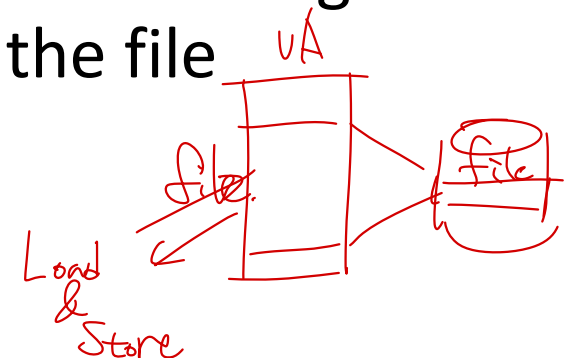
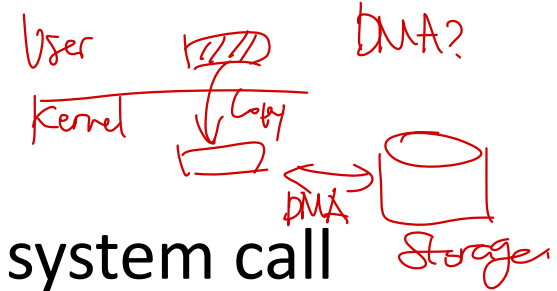
Virtual or Physical Dirty/Use Bits

- Most machines keep dirty/use bits in the page table entry (PTE)
- 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




Two 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** → fast! : Don't need to copy twice, Only once needed
 - Open file as a memory segment
 - Program uses load/store instructions on segment memory, implicitly operating on the file
 - What are next steps?



Two Models for Application File I/O

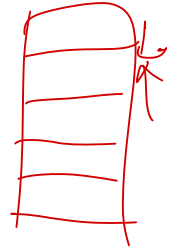
- Explicit read/write system calls
 - Data copied to user process using system call
 - Application operates on data
 - Data copied back to kernel using system call
- Memory-mapped files
 - Open file as a memory segment
 - Program uses load/store instructions on segment memory, implicitly operating on 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

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 *— due to demand paging*
 - Process can **start working before all the pages are populated**
- Interprocess communication
 - **Shared file pages**

File-backed memory vs Anonymous memory



- File-backed memory
 - Code segment -> code portion of executable
 - Data (initially loaded from executable) -> temp files
 - Shared libraries -> code file and temp data file
 - Memory-mapped files -> memory-mapped files
 - When process ends, delete temp files
- Anonymous memory
 - heap, stack segments -> no-mapping file (i.e., swap disk/file)

Design Challenges (mechanism & policy)

- **How to resume a process after a fault?**
 - Process might have been in the middle of an instruction!
 - Save states during exception handling (we learned in the earlier lecture)
- **What to eject?**
 - How to allocate physical pages amongst processes?
 - Which of a particular process's pages to keep in memory?
 - A poor choice can lead to horrible performance

Locality ¹⁹⁷¹ "Locality and working set"

- **Temporal locality**

- Locations referenced recently likely to be referenced again

ex. Sequential access,

- **Spatial locality** → *Data structure like array, "Directory" in file system.*
 - Locations (near recently referenced) locations are likely to be referenced soon

- Although the cost of paging is high, if it is infrequent enough it is acceptable

- Processes usually exhibit both kinds of locality during their execution, making paging practical

Working Set Model

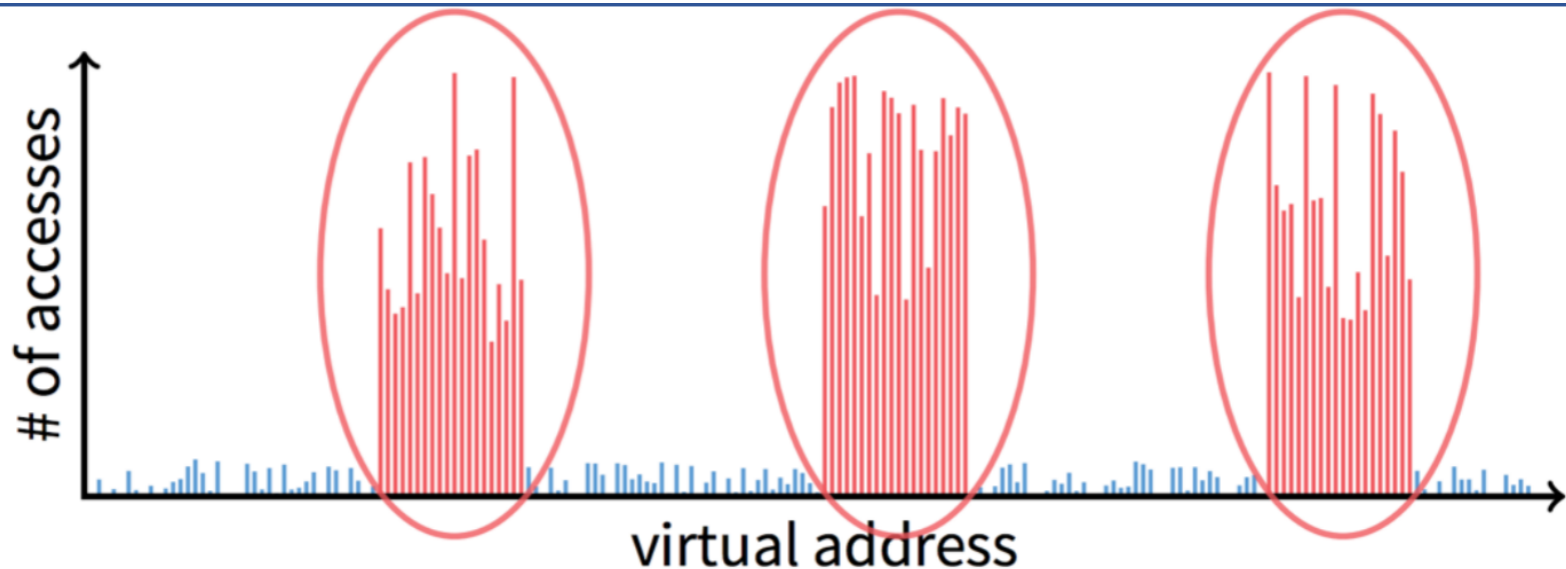


- Disk much, much slower than memory
 - Goal: run at memory speed, not disk speed
- 80/20 rule: 20% of memory gets 80% of memory accesses
 - Keep the hot 20% in memory
 - Keep the cold 80% on disk

→ "Benefit of cache"

Keep 20% hot memory
in cache

Working Set Model



- Disk much, much slower than memory
 - Goal: run at memory speed, not disk speed
- 80/20 rule: 20% of memory gets 80% of memory accesses
 - Keep the hot 20% in memory
 - Keep the cold 80% on disk

Working Set Model



- Disk much, much slower than memory
 - Goal: run at memory speed, not disk speed
- 80/20 rule: 20% of memory gets 80% of memory accesses
 - Keep the hot 20% in memory
 - Keep the cold 80% on disk

Cache Replacement Policy

- When a page fault occurs, the OS loads the faulted page from disk into a page frame of physical memory
- At some point, the process used all of the page frames it is allowed to use
 - This is likely (much) less than all of available memory
- When this happens, the OS must replace a page for each page faulted in *how can OS choose cold page as victim?*
 - It must evict a page to free up a page frame
- The page replacement algorithm determines how this is done
- **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 is the worst access pattern when using FIFO?

Sequential access

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			

miss

miss + evict

Sequential scan: worst case for FIFO is if program strides through memory that is larger than the cache

100% miss

Two policies: MIN, LRU

ideal *proposed*

- MIN (Belady's algorithm)
 - Replace the cache entry that **will not be used for the longest time into the future** → *ideal, how the OS knows this?*
 - Optimality proof based on exchange: if evict an entry used sooner, that will trigger an earlier cache miss → *feedback*
- Least Recently Used (LRU) → *Temporal Locality*
 - Replace the cache entry that **has not been used for the longest time in the past**
 - **Approximation of MIN** (Belady: future)

LRU

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

9 hits

C evict

D evict

FIFO

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

7 hits

first evict

A evict

B evict

C evict

D evict

MIN

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

9 hits

random

No future entry

Follow LRU

LRU/MIN for Sequential Scan

→ Sequential is also bad for LRU

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 <i>→ knows future</i>															
1	A					+					+			+	
2		B					+					+	C		
3			C					+	D					+	
4				D	E					+					+

most
D will be accessed later
→ evict B

Why sequential scan is adversarial to LRU?

25/12/01

* Should be better always, but, some access pattern... no.

Belady's Anomalyⁱⁿ



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		

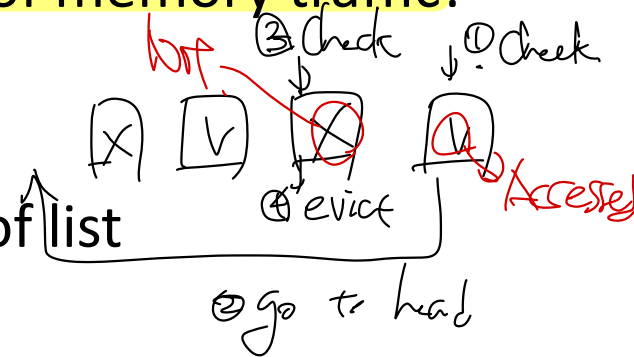


More physical memory doesn't always mean fewer faults

안녕하세요.

Straw Man LRU implementation

- Stamp PTEs with timer value
 - E.g., CPU has cycle counter
 - Automatically writes value to PTE on each page access
 - Scan page table to find oldest counter value = LRU page
 - Problem: Would cause a high volume of memory traffic!
- Keep doubly-linked list of pages
 - On access, remove page, place at tail of list
 - Problem: again, very expensive



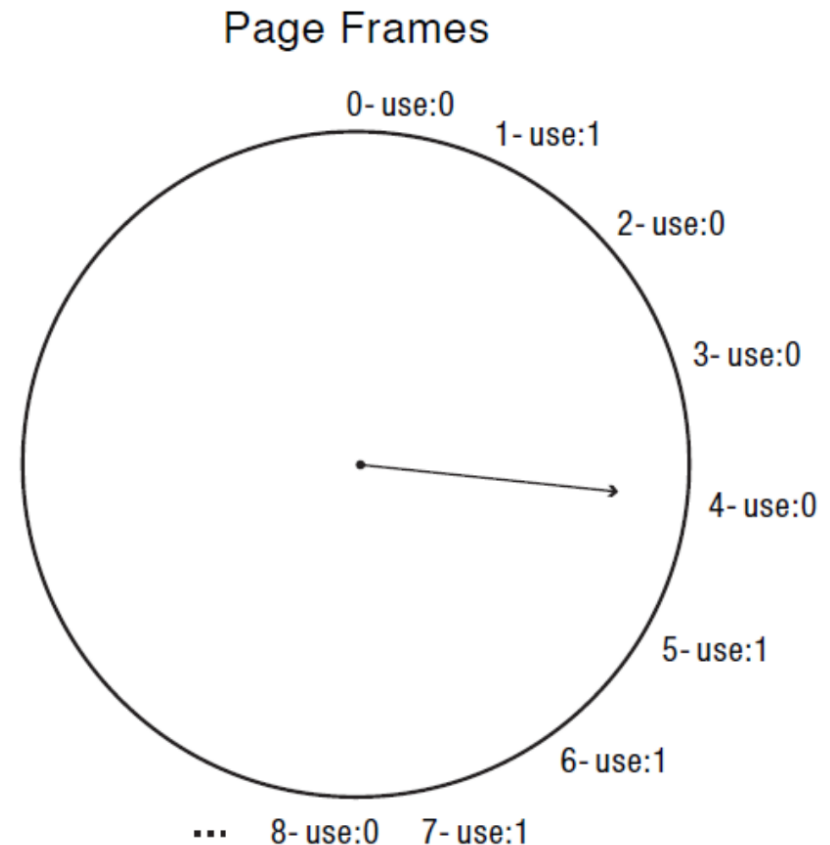
Head: most recently accessed

What to do? Just approximate LRU, don't try to do it exactly



Clock Algorithm: Estimating LRU

- Periodically, sweep through pages
- If page is unused, reclaim
- If page is used, mark as unused



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) {
    counter = 0; // counter means "notInUseSince"
    used bit = 0
} else {
    counter++;
    used bit = 0
}
```

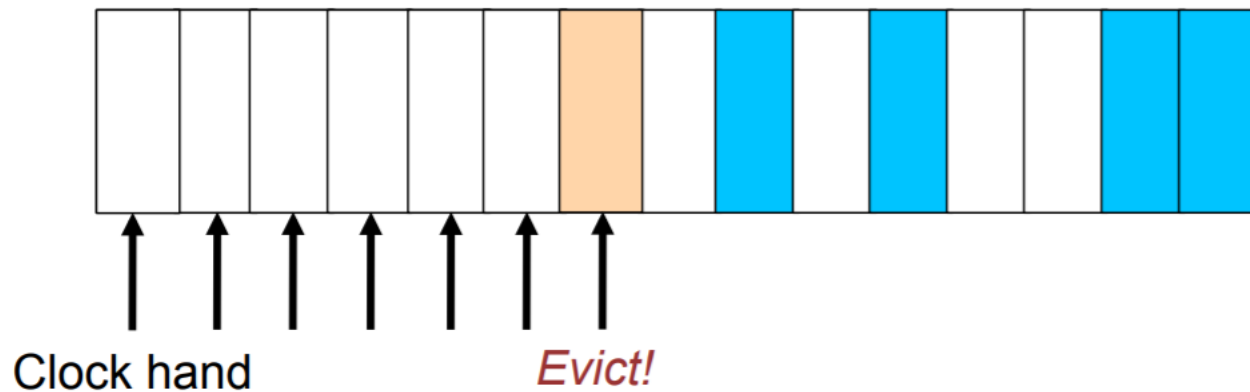
2nd chance example



2nd chance example

→ Used

- If used bit = 1, clear bit, set counter = 0, and advance hand to give it a 2nd-chance
- If used bit = 0 and counter ≥ 2 , evict this page



Other policies

- LFU (least frequently used) eviction
 - Instead of just used bit, count # times each page accessed
 - Least frequently accessed must not be very useful (or maybe was just brought in and is about to be used)
 - Decay usage counts over time (for pages that fall out of usage)

Huge overhead, Count every page fault

#win. → eviction

*→ 7/12/10/20 eviction 순서로
순서대로 순서가 있다.*
- MFU (most frequently used) algorithm
 - Because page with the smallest count was probably just brought in and has yet to be used
- Neither LFU nor MFU used very commonly

Fixed vs. Variable Space

- How to determine how much physical memory to give to each process?
- Fixed space algorithms
 - Each process is given a limit of pages it can use
 - When it reaches the limit, it replaces from its own pages
 - Local replacement
 - Some processes may do well while others suffer
- Variable space algorithms
 - Process' set of pages grows and shrinks dynamically
 - Global replacement
 - One process can ruin it for the rest

Swap Files

- What happens to the page that we choose to evict?
 - Depends on what kind of page (file-backed vs anonymous) it is!
 - ↳ Write back to disk
 - ↳ Just evict(?)
- OS maintains one or more swap files or partitions on disk
 - Special data format for storing pages that have been swapped out

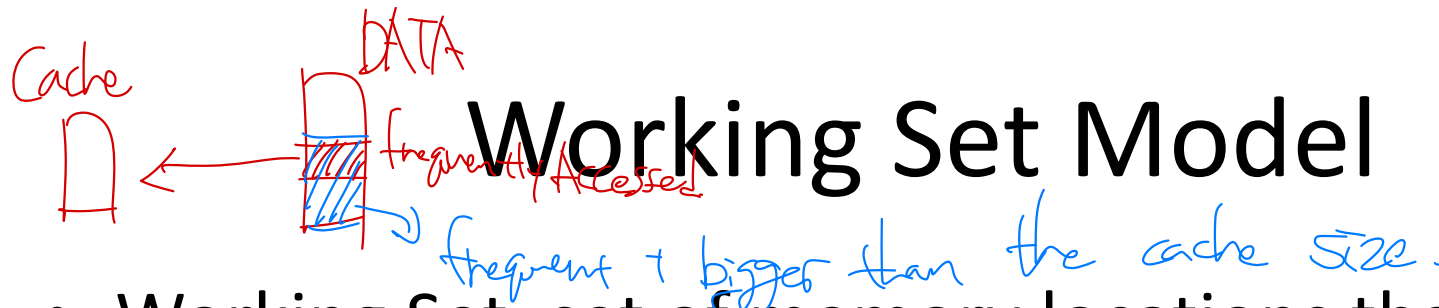
Page eviction

- How we evict a page depends on its type.
- Code page:
 - Just remove it from memory – can recover it from the executable file on disk!
- Unmodified (clean) data page: *→ less expensive to evict*
 - If the page has previously been swapped to disk, just remove it from memory
 - Assuming that page's backing store on disk has not been overwritten
 - If the page has never been swapped to disk, allocate new swap space and write the page to it
 - Exception: unmodified zero page – no need to write out to swap at all!
- Modified (dirty) data page:
 - If the page has previously been swapped to disk, write page out to the swap space
 - If the page has never been swapped to disk, allocate new swap space and write the page to it

dirty bit.
set by
H/W

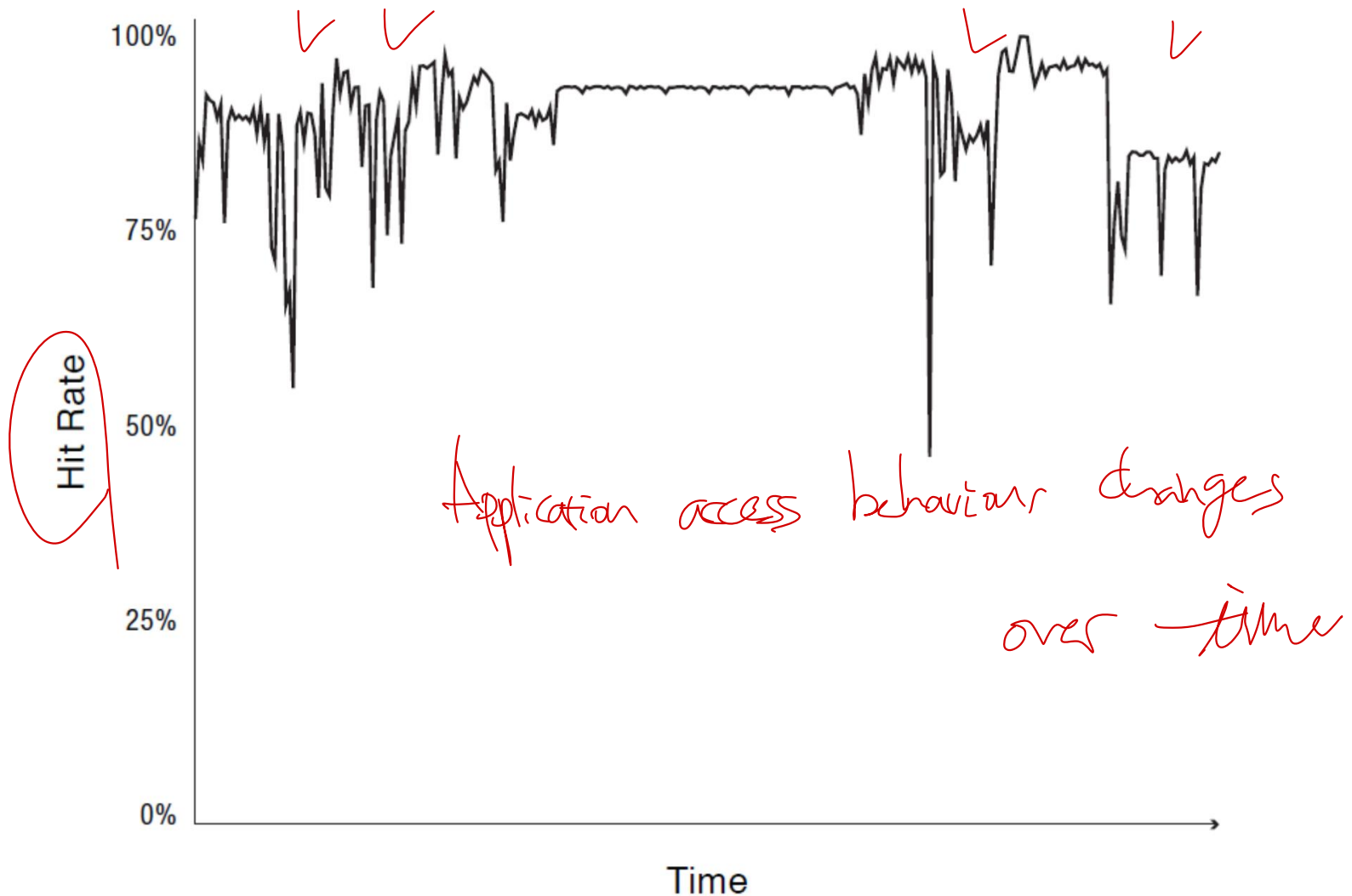
Recap

- 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”
b2d4



- Working Set: set of memory locations that need to be cached for reasonable cache hit rate
- The working set size is the # of unique pages in the working set
 - The number of pages referenced in the interval $(t-w, t)$
 - * – Poor locality means that working set size is ()
bigger than cache
- We want the working set to be the set of pages a process needs in memory to prevent heavy faulting
 - Each process has a parameter w that determines a working set with few faults

Phase Change Behavior



Working Set Problem

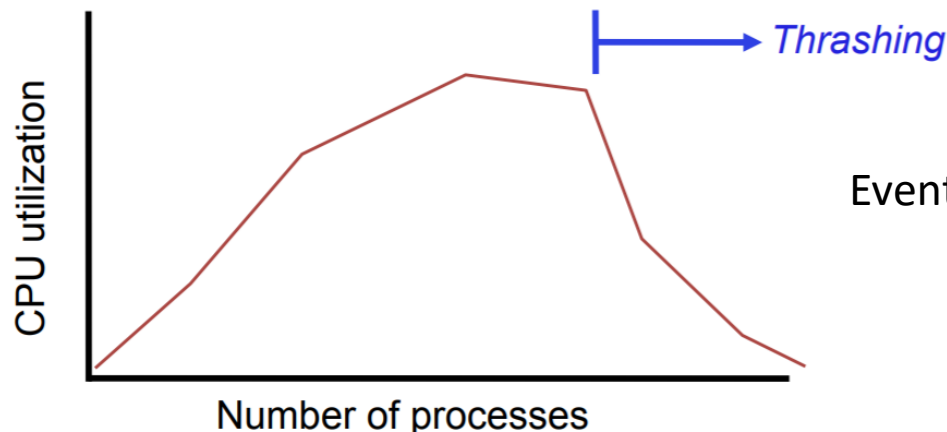
- Problems
 - How do we determine w ?
 - How do we know when the working set changes?
- Too hard to answer
 - So, working set is not used in practice as a page replacement algorithm
- However, it is still used as an abstracted terminology
 - The intuition is still valid
 - When people ask, “How much memory does Firefox need?”, they are in effect asking for the size of Firefox’s working set

Page Fault Frequency (PFF)

- Page Fault Frequency (PFF) is a variable space algorithm that uses a more ad-hoc approach
 - Monitor the fault rate for each process
 - If the fault rate is above a high threshold, give it more memory
 - So that it faults less
 - But not always work, why? → Access pattern changes over time -
 - If the fault rate is below a low threshold, take away memory
 - Should fault more
 - Likewise, not always work
- Hard to use PFF to distinguish between changes in locality and changes in size of working set

Thrashing

- Page replacement algorithms avoid thrashing
 - When OS spent most of the time in paging data back and forth from disk
 - Little time spent doing useful work (making progress)
 - In this situation, the system is **overcommitted**
 - No idea which pages should be in memory to reduce faults
 - Could just be that there isn't enough physical memory for all of the processes in the system



Eventually, no useful work gets done!