



# Memory Virtualization: Beyond Physical Memory

CS 571: *Operating Systems (Spring 2022)*  
Lecture 5

Yue Cheng

Some material taken/derived from:

- Wisconsin CS-537 materials created by Remzi Arpacı-Dusseau.

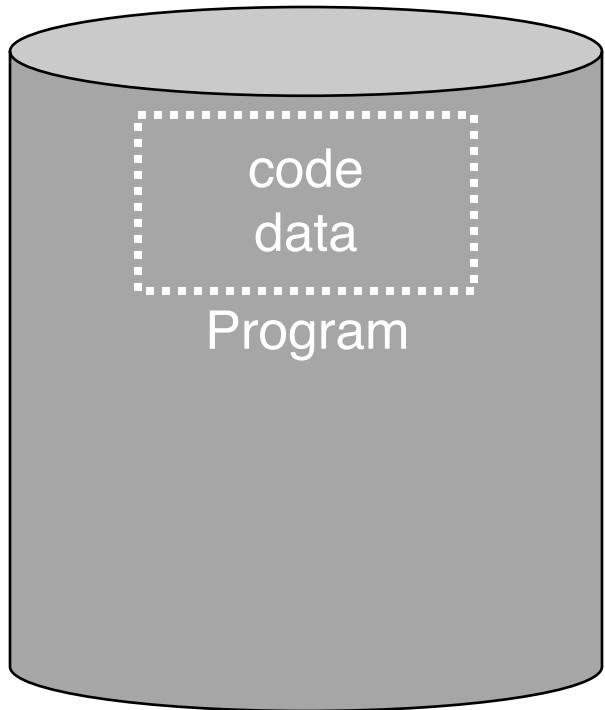
Licensed for use under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

# Today's outline

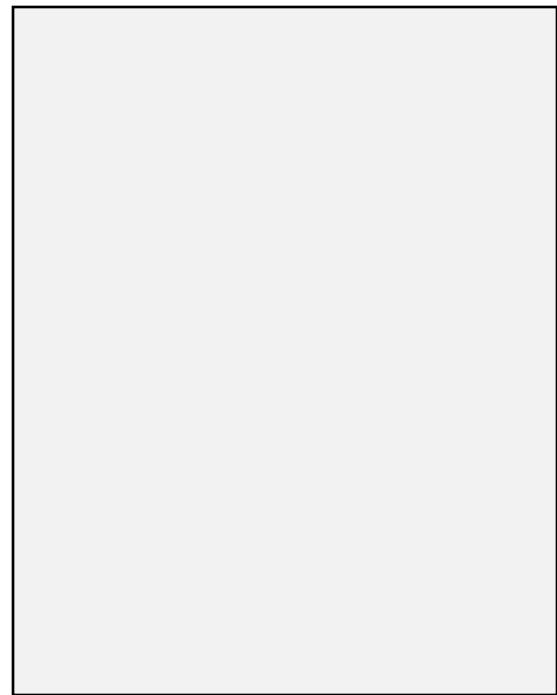
1. Mechanisms
2. Polices
  1. FIFO
  2. Random
  3. LRU
  4. MIN: Belady's optimal
  5. ARC
- 3. Misc. (TLB caching)

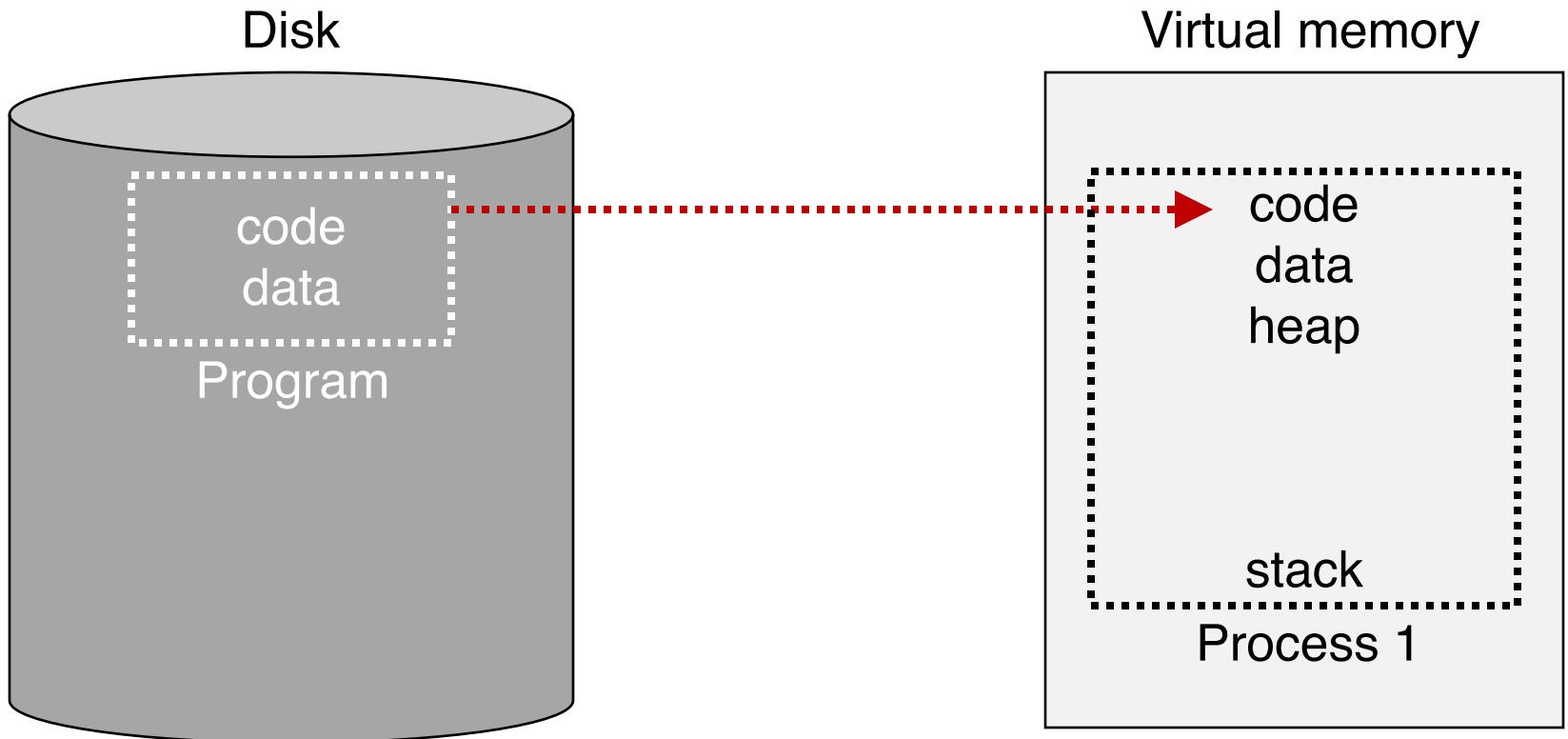
# Beyond Physical Memory: Mechanisms

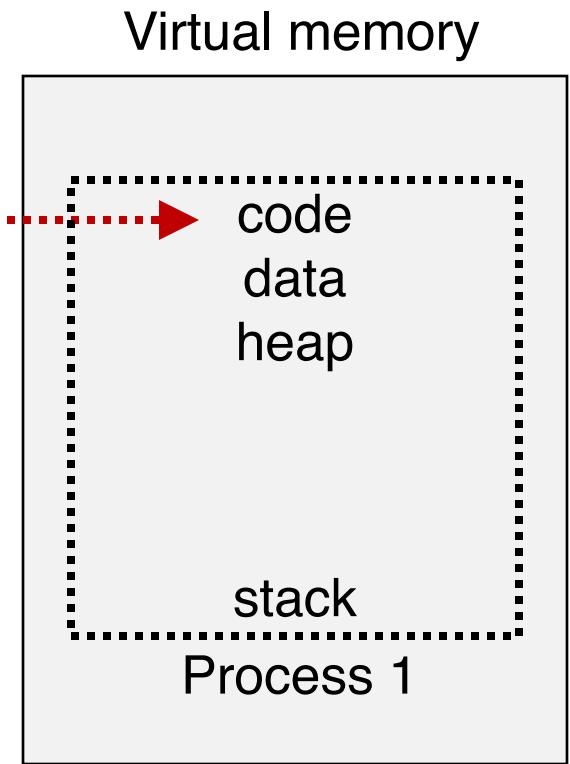
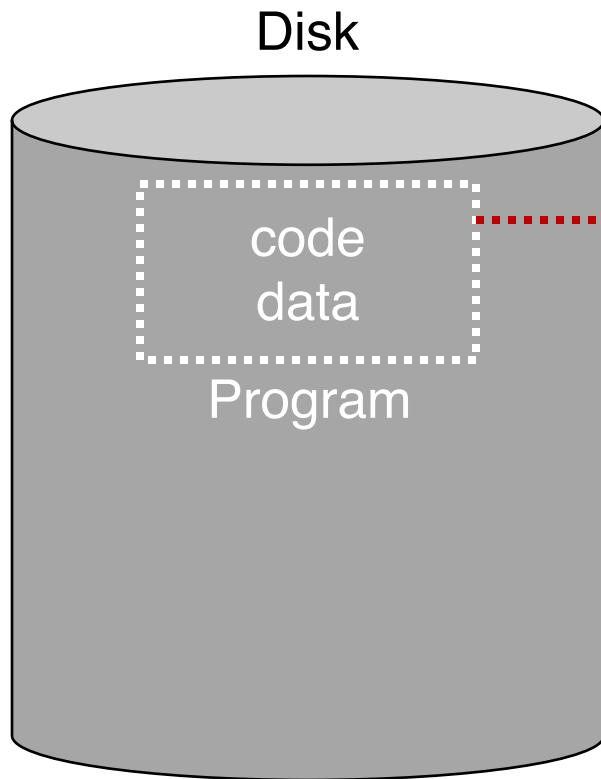
Disk



Virtual memory

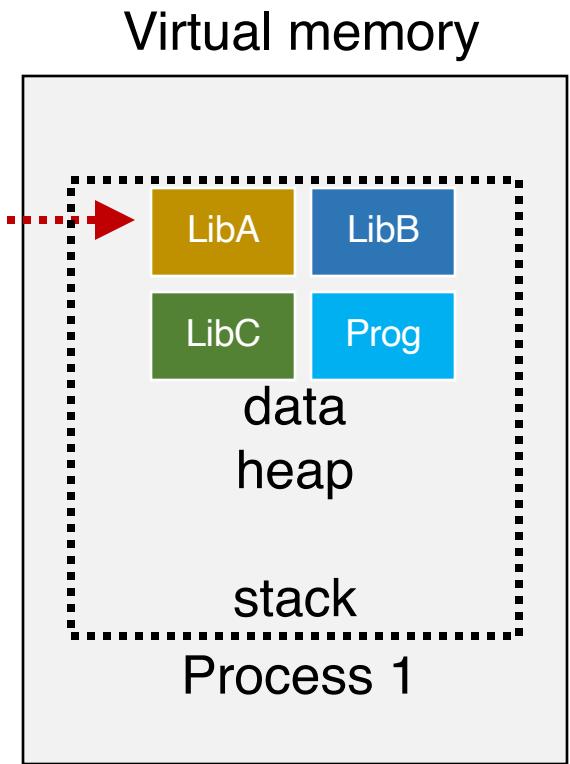
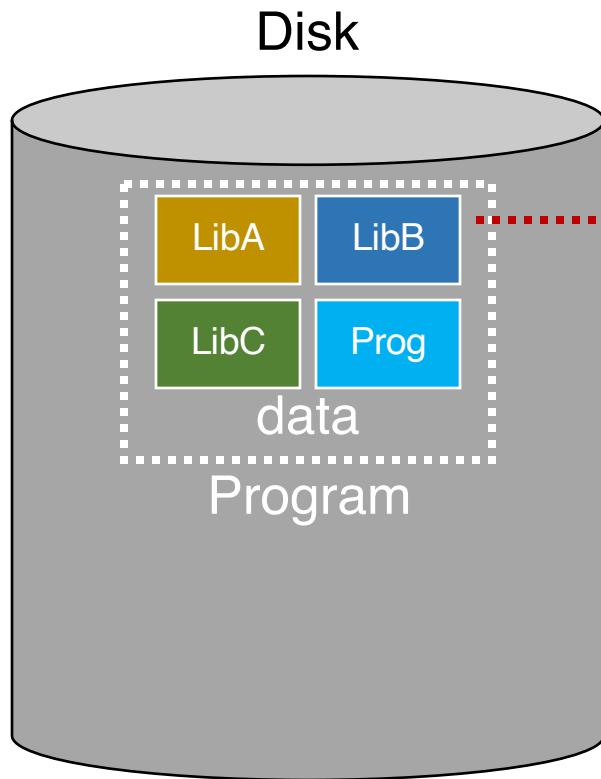






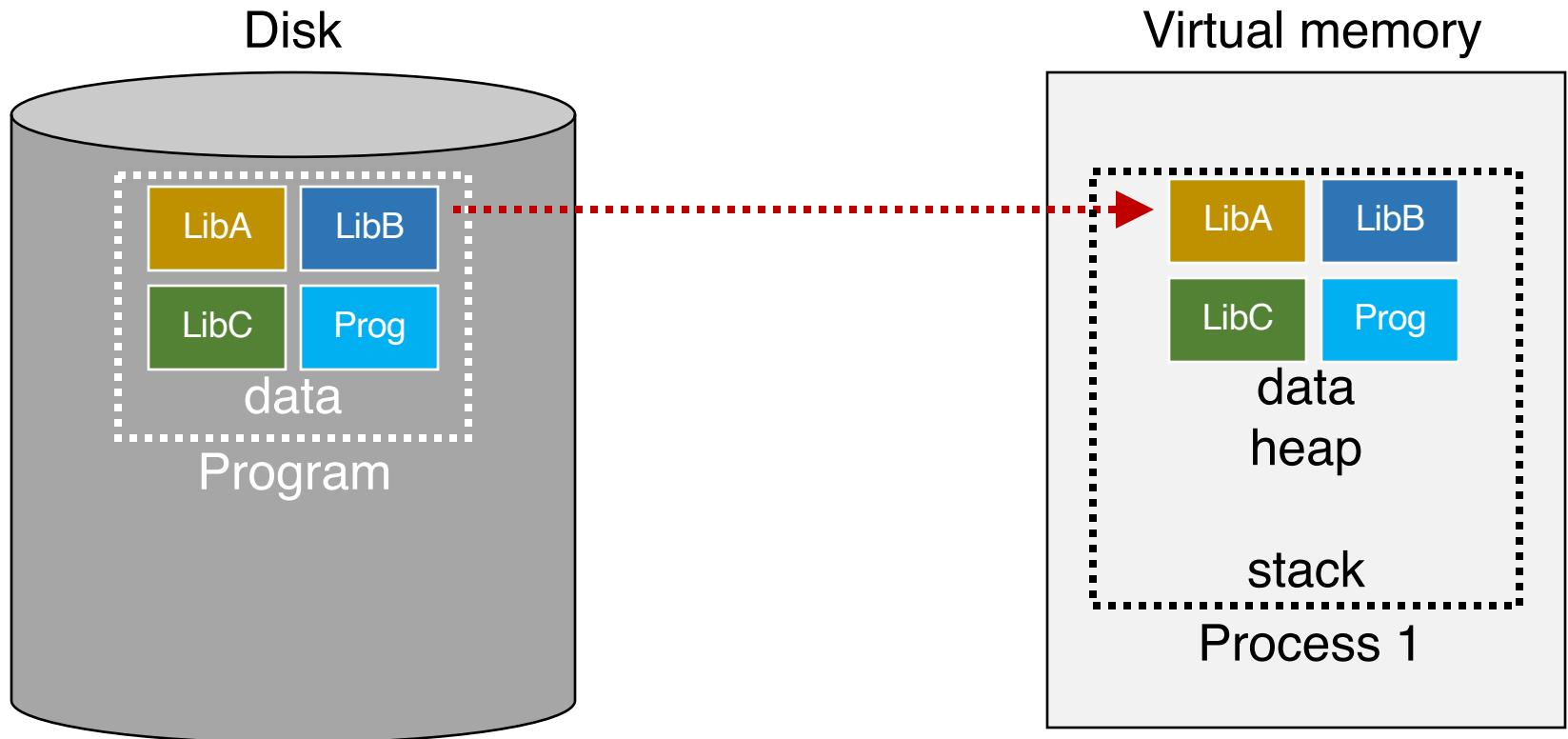
What's in code?

text

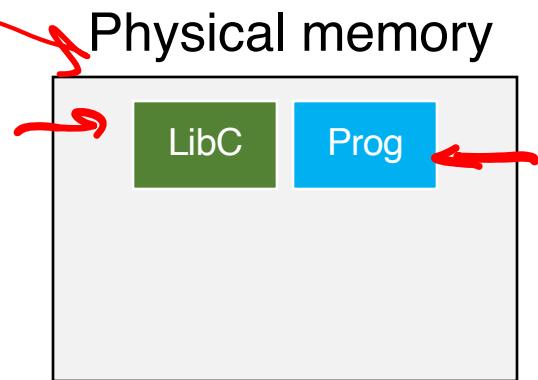
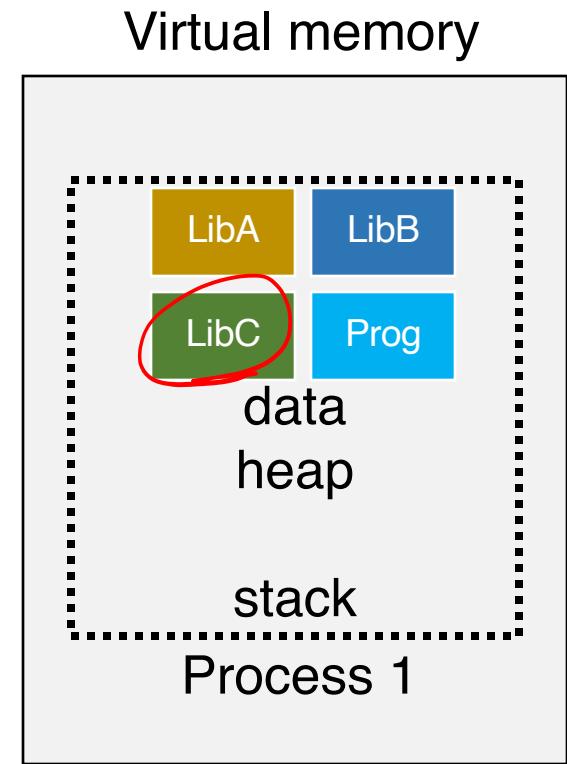
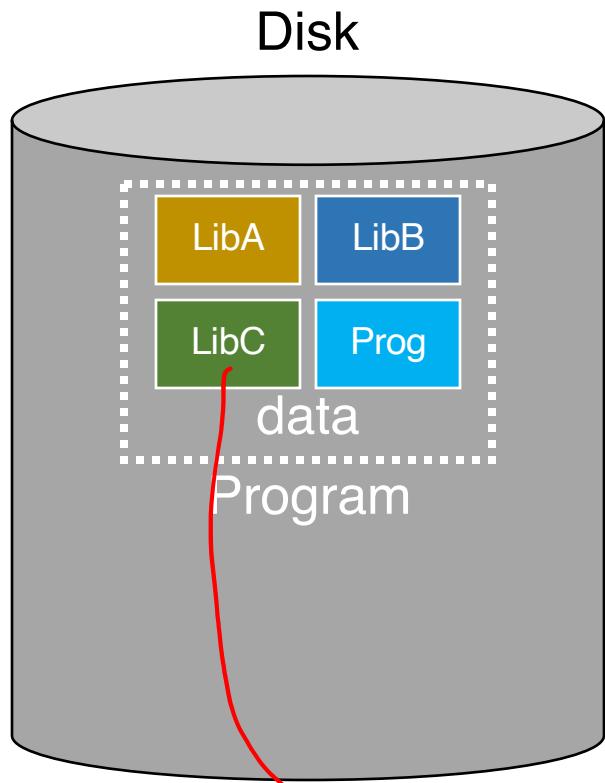


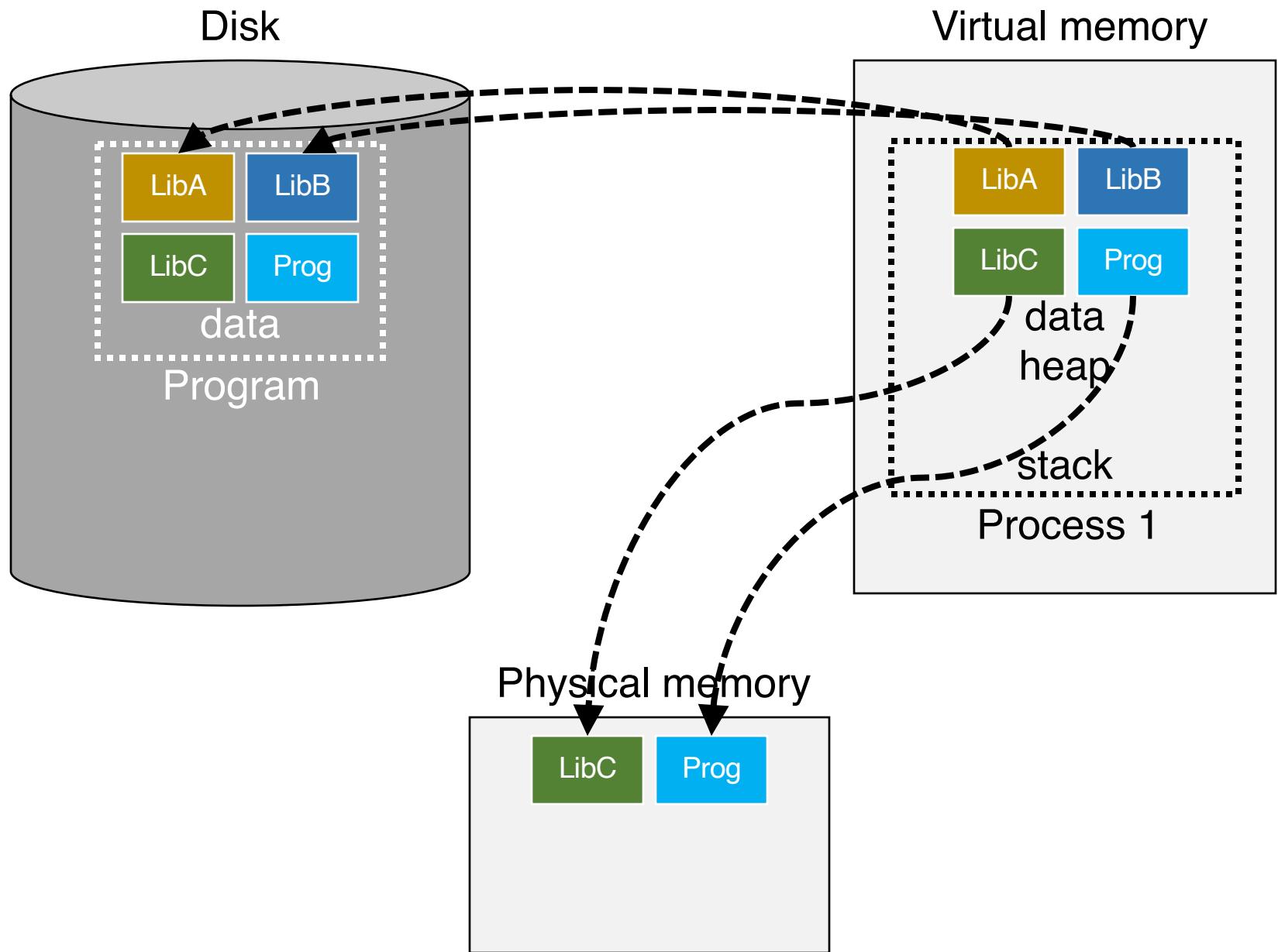
What's in code?

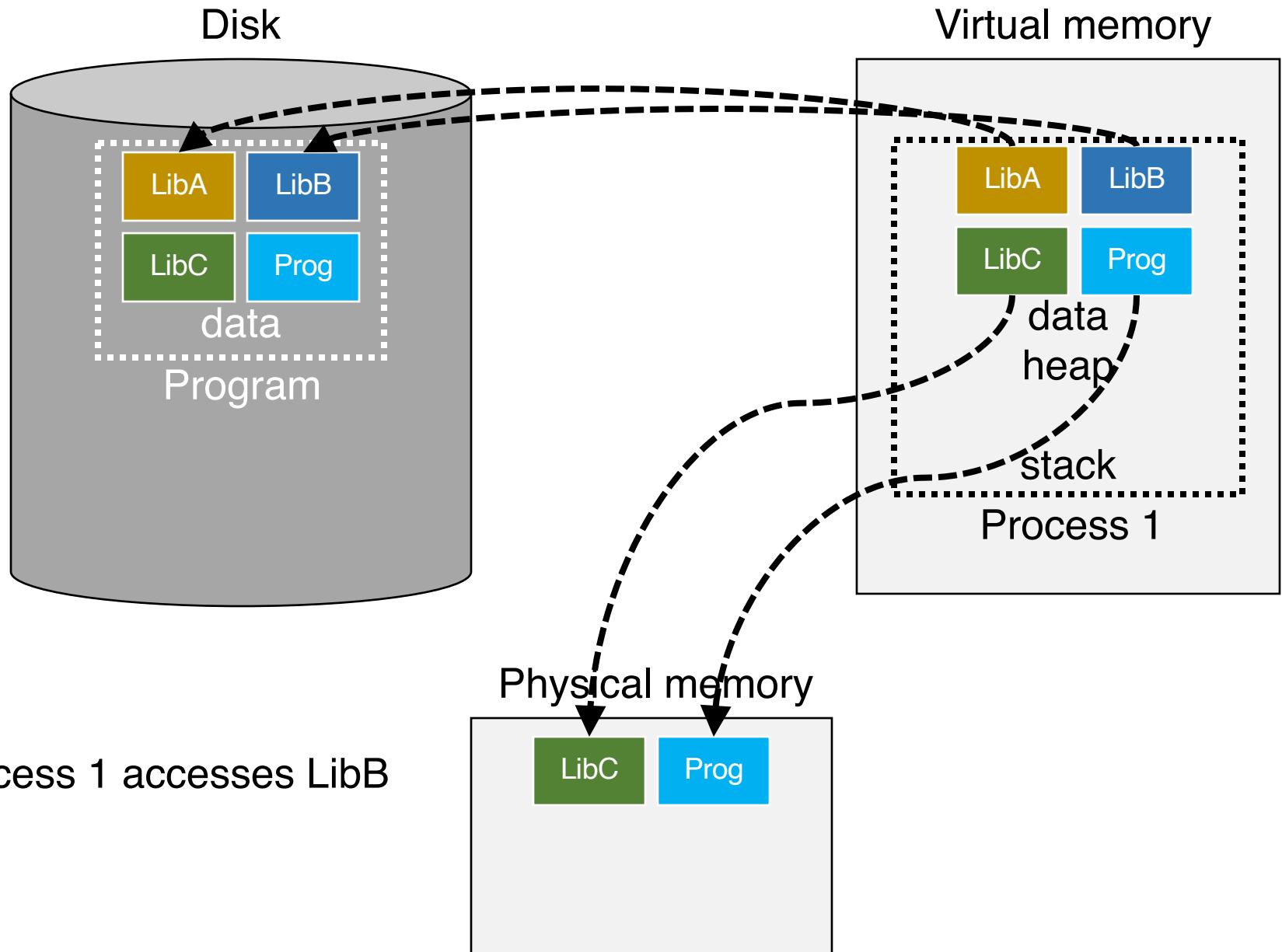
Many large libraries, some of which are rarely/never used

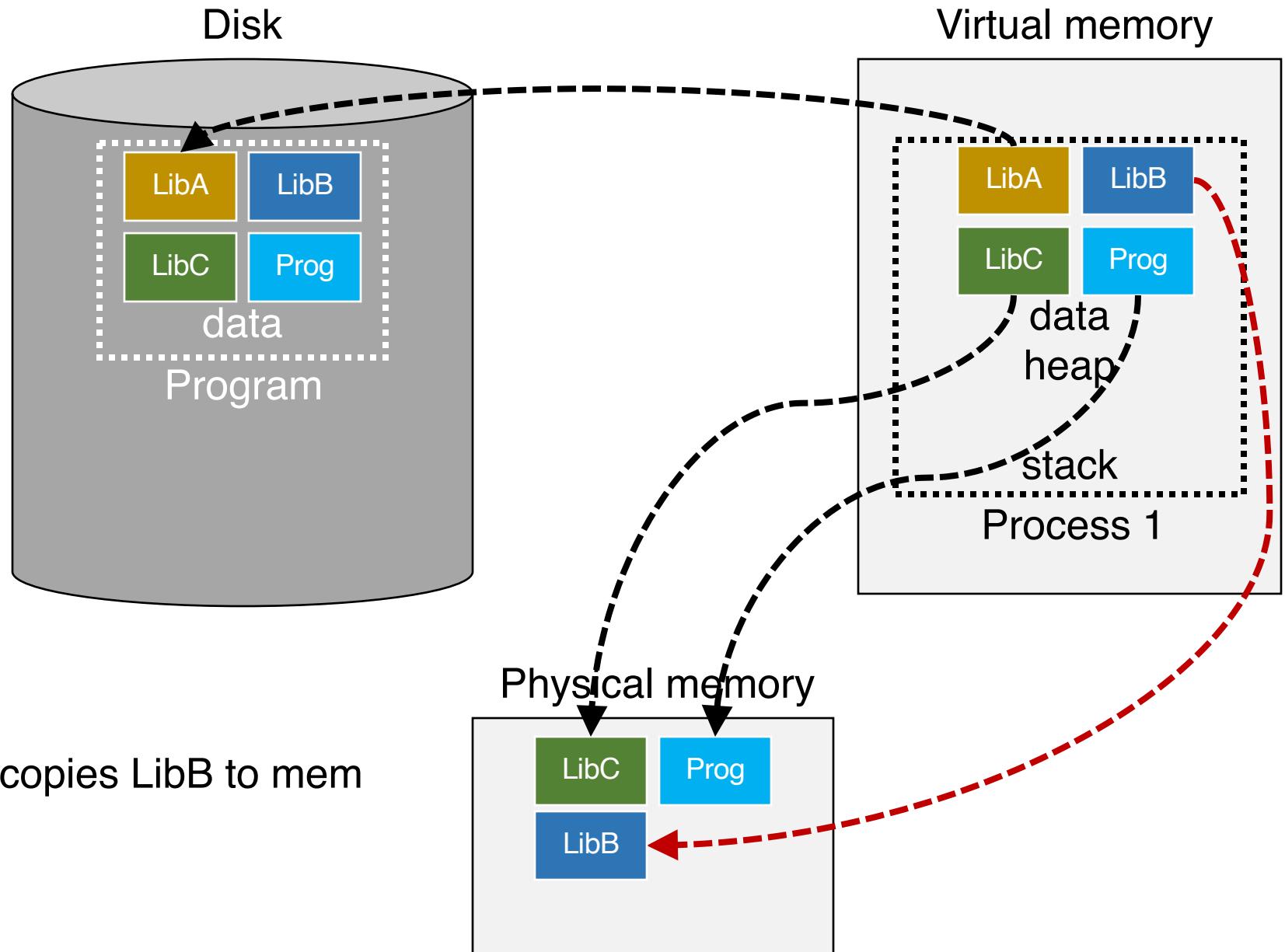


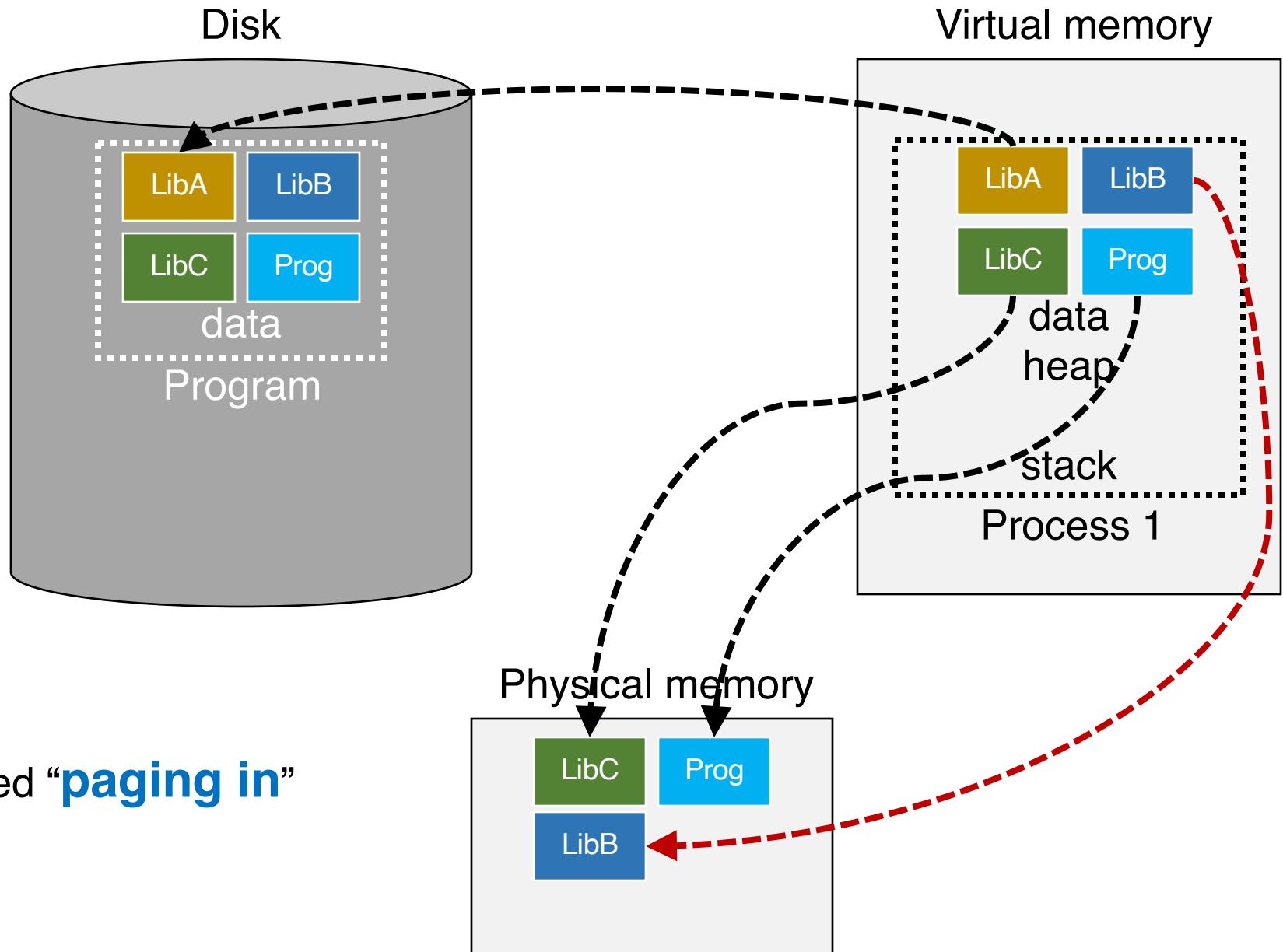
How to avoid wasting **physical pages** to back rarely used **virtual pages**?









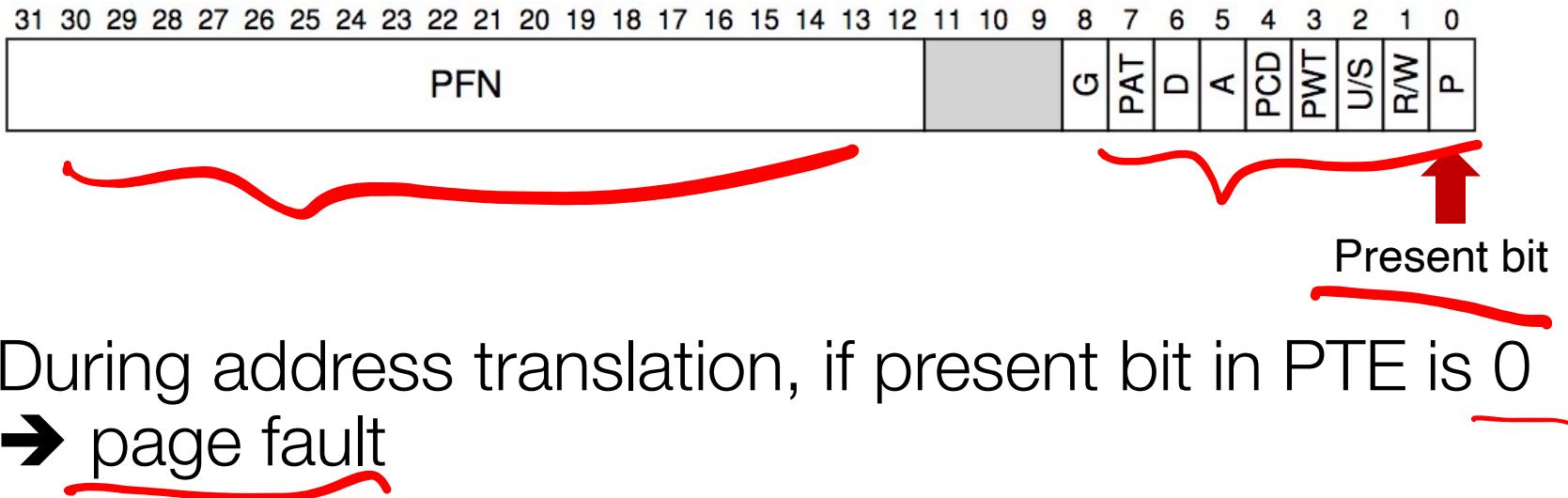


# How to Know Where a Page Lives?

# Present Bit

- With each PTE a present is associated
  - 1 → in-memory, 0 → out in disk

An 32-bit X86 page table entry (PTE)



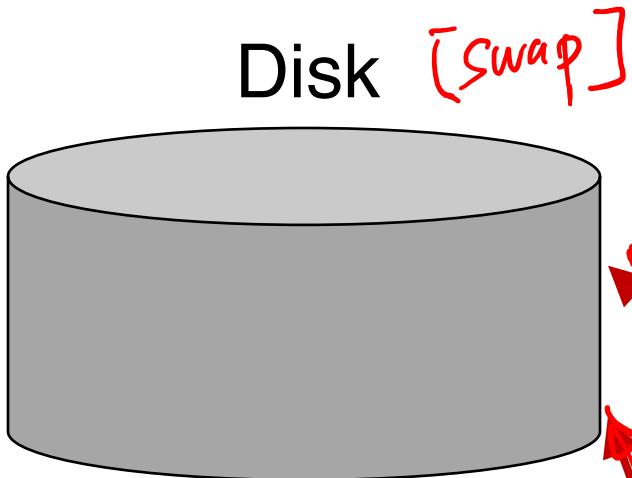
# Present Bit

VPN  $\xrightarrow{\text{IP of array.}}$  PFN

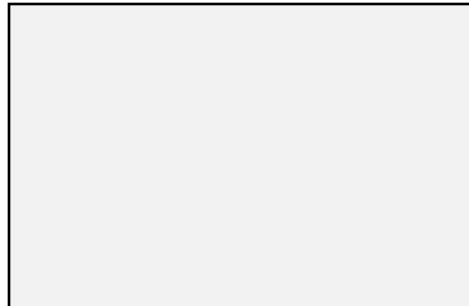
	PFN	valid	prot	present
0	5	1	r-x	1
1	-	0	-	-
2	-	0	-	-
3	60	1	rw-	0
4	-	0	0	-
5	-	0	0	-
6	-	0	0	-
7	4	1	rw-	1
8.	64	1	rw-	0

A page table

# Present Bit



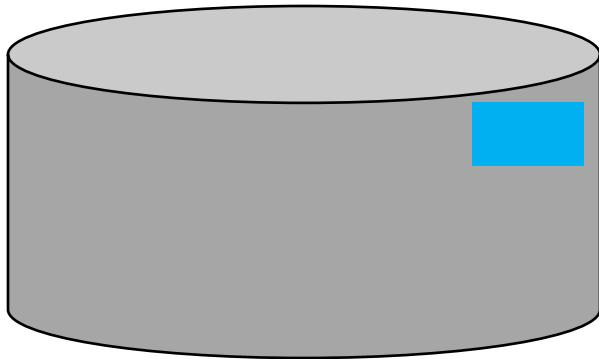
Phys memory



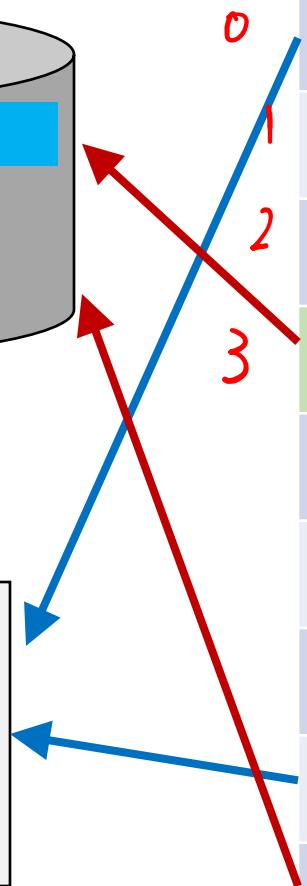
PFN	valid	prot	present
5	1	r-x	1
-	0	-	-
-	0	-	-
60	1	rw-	0
-	0	0	-
-	0	0	-
-	0	0	-
4	1	rw-	1
64	1	rw-	0

# Present Bit

Disk



Phys memory

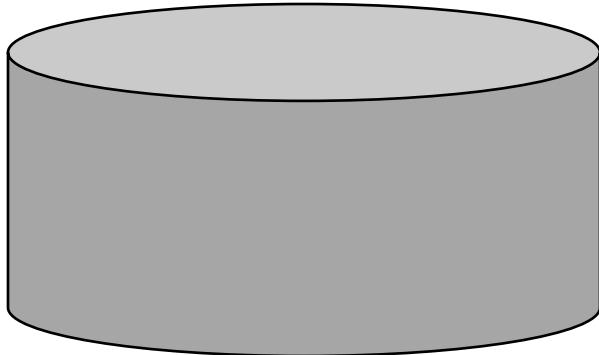


PFN	valid	prot	present
5	1	r-x	1
-	0	-	-
-	0	-	-
60	1	rw-	0 <span style="color: cyan;">access</span>
-	0	0	-
-	0	0	-
-	0	0	-
4	1	rw-	1
64	1	rw-	0

# Present Bit

malloc.  
sbrk. → heap.  
VA

Disk



Phys memory



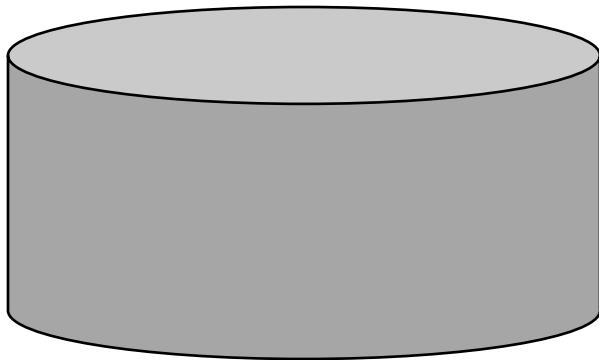
PFN	valid	prot	present
5	1	r-x	1
- 31	0	-	-
-	0	-	-
8	1	rw-	1
-	0	0	-
-	0	0	-
-	0	0	-
4	1	rw-	1
64	1	rw-	0

A small red horizontal line with a small red dot at its right end.

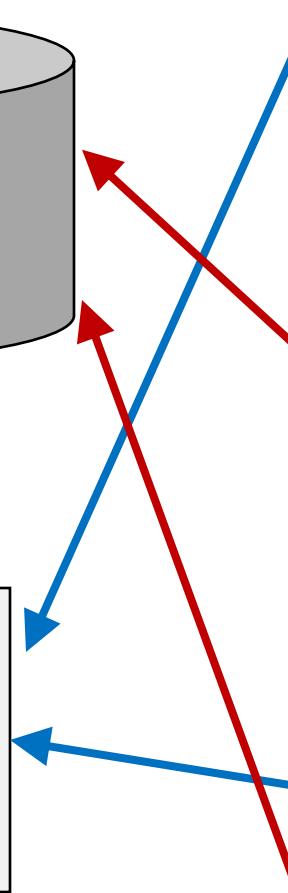
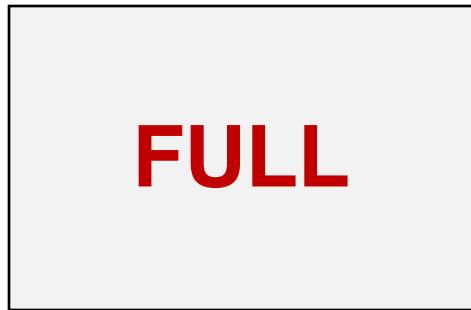
# What if **NO** Memory is Left?

# Present Bit

Disk

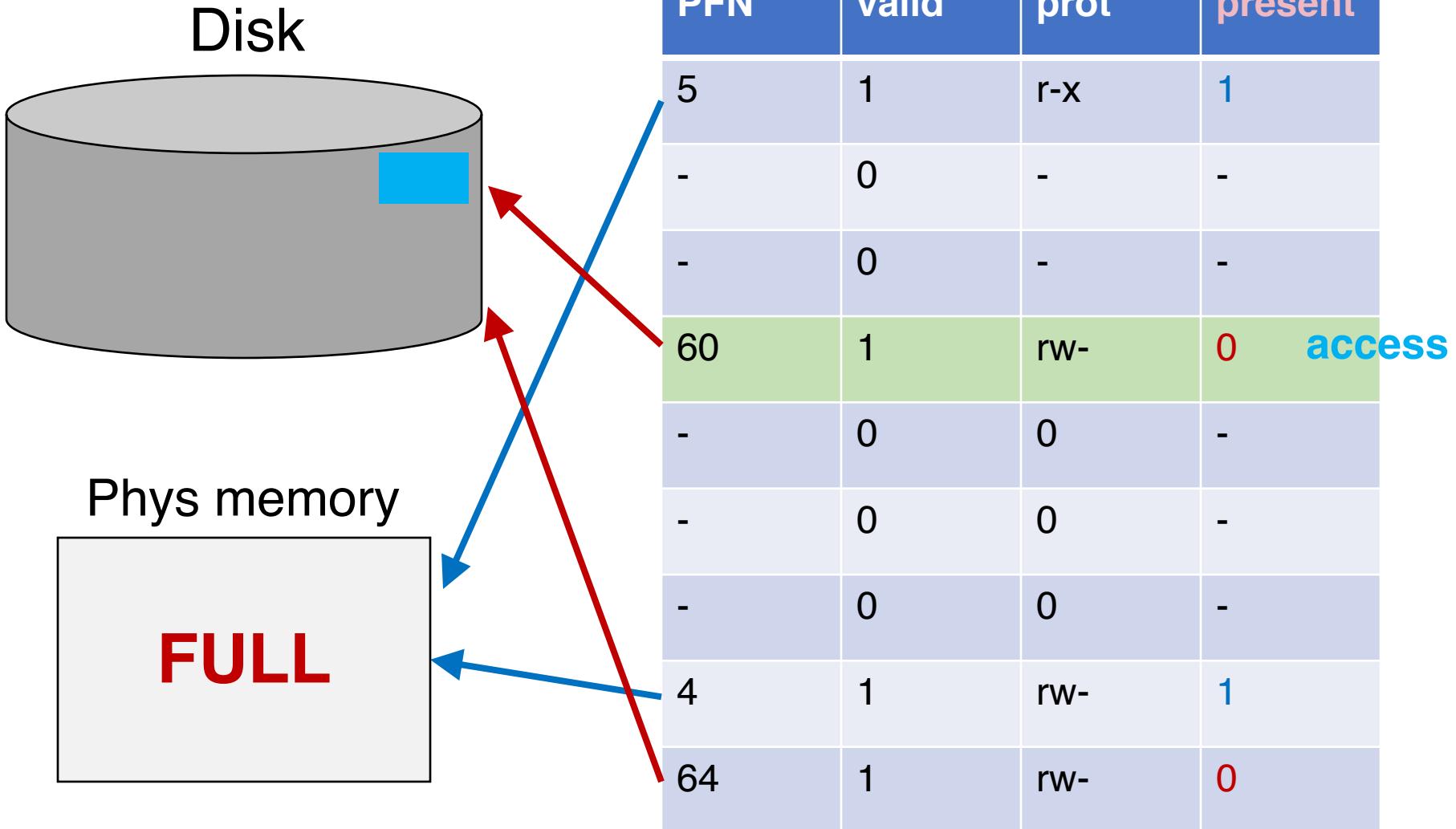


Phys memory

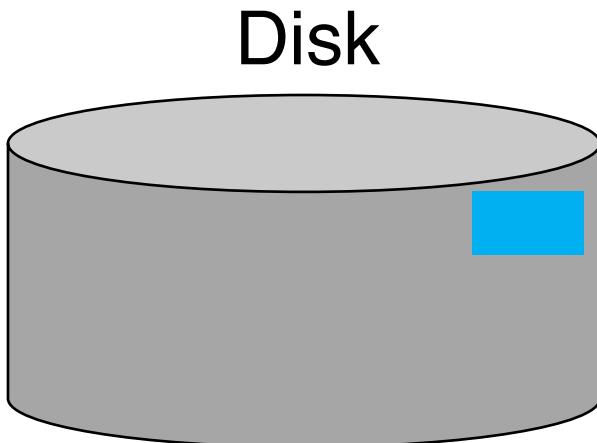


PFN	valid	prot	present
5	1	r-x	1
-	0	-	-
-	0	-	-
60	1	rw-	0
-	0	0	-
-	0	0	-
-	0	0	-
4	1	rw-	1
64	1	rw-	0

# Present Bit



# Present Bit

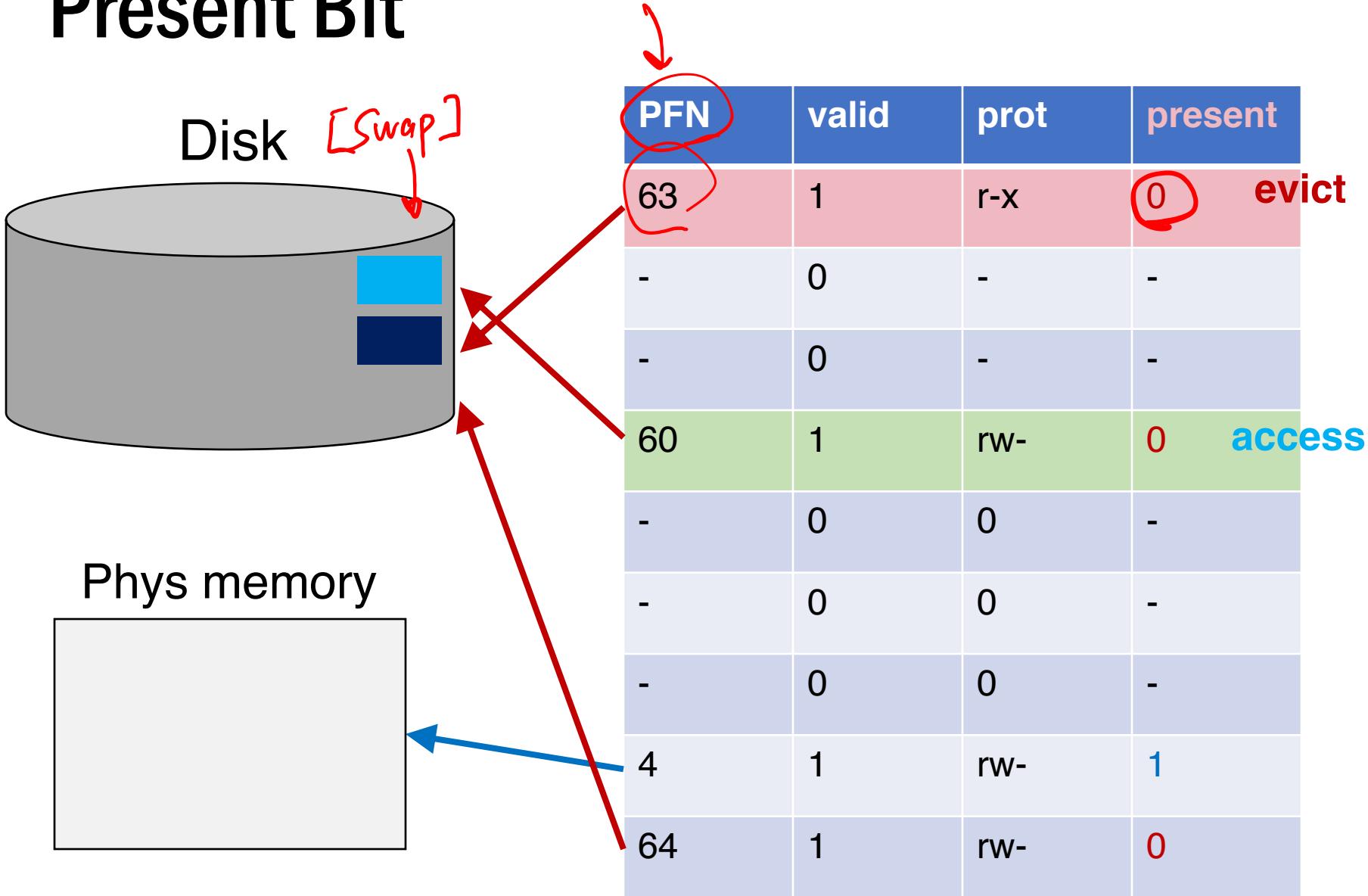


Phys memory



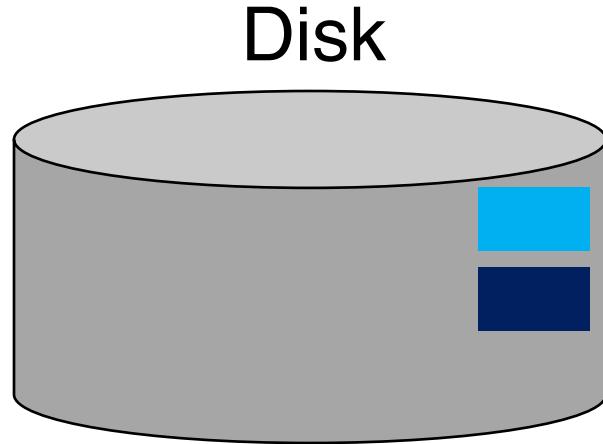
PFN	valid	prot	present
5	1	r-x	1 <b>evict</b>
-	0	-	-
-	0	-	-
60	1	rw-	0 <b>access</b>
-	0	0	-
-	0	0	-
-	0	0	-
4	1	rw-	1
64	1	rw-	0

# Present Bit



# Present Bit

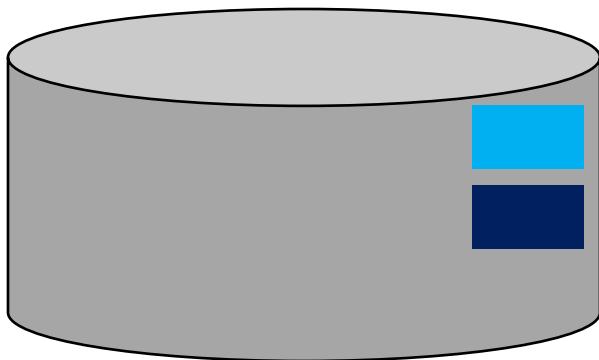
called “swapping out”  
or “paging out”



PFN	valid	prot	present
63	1	r-x	0 <b>evict</b>
-	0	-	-
-	0	-	-
60	1	rw-	0 <b>access</b>
-	0	0	-
-	0	0	-
-	0	0	-
4	1	rw-	1
64	1	rw-	0

# Present Bit

Disk



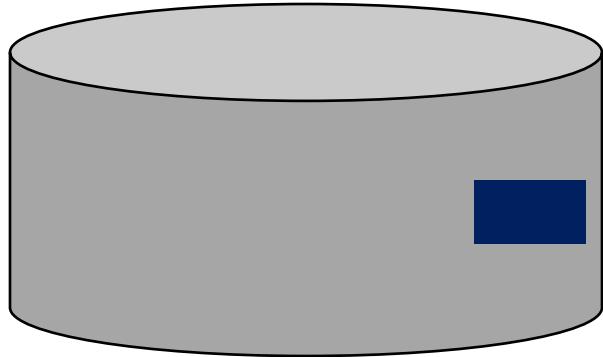
Phys memory



PFN	valid	prot	present
63	1	r-x	0
-	0	-	-
-	0	-	-
60	1	rw-	0 access
-	0	0	-
-	0	0	-
-	0	0	-
4	1	rw-	1
64	1	rw-	0

# Present Bit

Disk



Phys memory



again, another “swapping in”  
or “paging in”

PFN	valid	prot	present
63	1	r-x	0
-	0	-	-
-	0	-	-
5	1	rw-	1 access
-	0	0	-
-	0	0	-
-	0	0	-
4	1	rw-	1
64	1	rw-	0

# Why not Leave Page on Disk?

# Why not Leave Page on Disk?

- Performance: Memory vs. Disk
- How long does it take to access a 4-byte `int` from main memory vs. disk?

- DRAM: ~100ns
- HDD. • Disk: ~10ms

$$\text{SSD} = \frac{100 \mu\text{s.}}{\text{10 Gbps NIC.}} = \underline{1-2 \mu\text{s.}}$$

# Beyond the Physical Memory

- Idea: use the disk space as an extension of main memory
- Two ways of interaction b/w memory and disk
  - Demand paging
  - Swapping

# Demand Paging

- Bring a page into memory **only when it is needed (demanded)**
  - Less I/O needed
  - Less memory needed
  - Faster response
  - Support more processes/users
- Page is needed  $\Rightarrow$  use the reference to page
  - If not in memory  $\Rightarrow$  must bring from the disk

# Swapping

- Swapping allows OS to support the illusion of a large virtual memory for multiprogramming
  - Multiple programs can run “**at once**”
  - Better utilization
  - Ease of use
- Demand paging vs. swapping
  - On demand vs. page replacement under memory pressure

# Address Translation Steps

MMU :

Hardware: for each memory reference:

Extract **VPN** from **VA**

Check **TLB** for **VPN**

Fast → **TLB hit**:

Build **PA** from **PFN** and offset

Fetch **PA** from memory

**TLB miss**:

Slow → Fetch **PTE** → **PT**.

if (!valid): exception [segfault]

else if (!present): exception [page fault: page miss]

else: extract **PFN**, insert in **TLB**, retry

# Address Translation Steps

Hardware: for each memory reference:

Extract **VPN** from **VA**

Check **TLB** for **VPN**

**TLB hit:**

Build **PA** from **PFN** and offset

Fetch **PA** from memory

**TLB miss:**

→ Fetch **PTE**

→ if (!valid): exception [segfault]

→ else if (!present): exception [page fault: page miss]

else: extract **PFN**, insert in **TLB**, retry

- Q: Which steps are expensive??

# Address Translation Steps

Hardware: for each memory reference:

(cheap) Extract **VPN** from **VA**

(cheap) Check **TLB** for **VPN**

**TLB hit:**

(cheap) Build **PA** from **PFN** and offset

**(expensive)** Fetch **PA** from memory

**TLB miss:**

**(expensive)** Fetch **PTE**

**(expensive)** if (!valid): exception [segfault]

**(expensive)** else if (!present): exception [page fault: page miss]

(cheap) else: extract **PFN**, insert in **TLB**, retry

- Q: Which steps are expensive??

# Page Fault

- The act of accessing a page that is not in physical memory is called a **page fault**
- OS is invoked to service the page fault
  - **Page fault handler**
- In a canonical example, we assume that **PTE** contains the page address on disk
  - To avoid getting too deep into the page fault handling process

# Simplified: Page-Fault Handler (OS)

PFN = FindFreePage()  
if (PFN == -1)

PFN = EvictPage()  
DiskRead(PTE.DiskAddr, PFN)

PTE.present = 1

PTE.PFN = PFN

retry instruction

# Simplified: Page-Fault Handler (OS)

**PFN** = FindFreePage()

if (**PFN** == -1)

**PFN** = EvictPage()

→ DiskRead(**PTE**.DiskAddr, **PFN**)

**PTE**.present = 1

**PTE.PFN** = **PFN**

retry instruction

Q: which steps are expensive?

# Simplified: Page-Fault Handler (OS)

(cheap) **PFN** = FindFreePage()

(cheap) if (**PFN** == -1)

(depends)  
           **PFN** = EvictPage()

(expensive) DiskRead(**PTE**.DiskAddr, **PFN**)

(cheap) **PTE**.present = 1

(cheap) **PTE.PFN** = **PFN**

(cheap) retry instruction

Q: which steps are expensive?

# Simplified: Page-Fault Handler (OS)

(cheap) **PFN** = FindFreePage()

(cheap) if (**PFN** == -1)

(depends) **PFN** = EvictPage()

(expensive) DiskRead(**PTE**.DiskAddr, **PFN**)

What to evict?

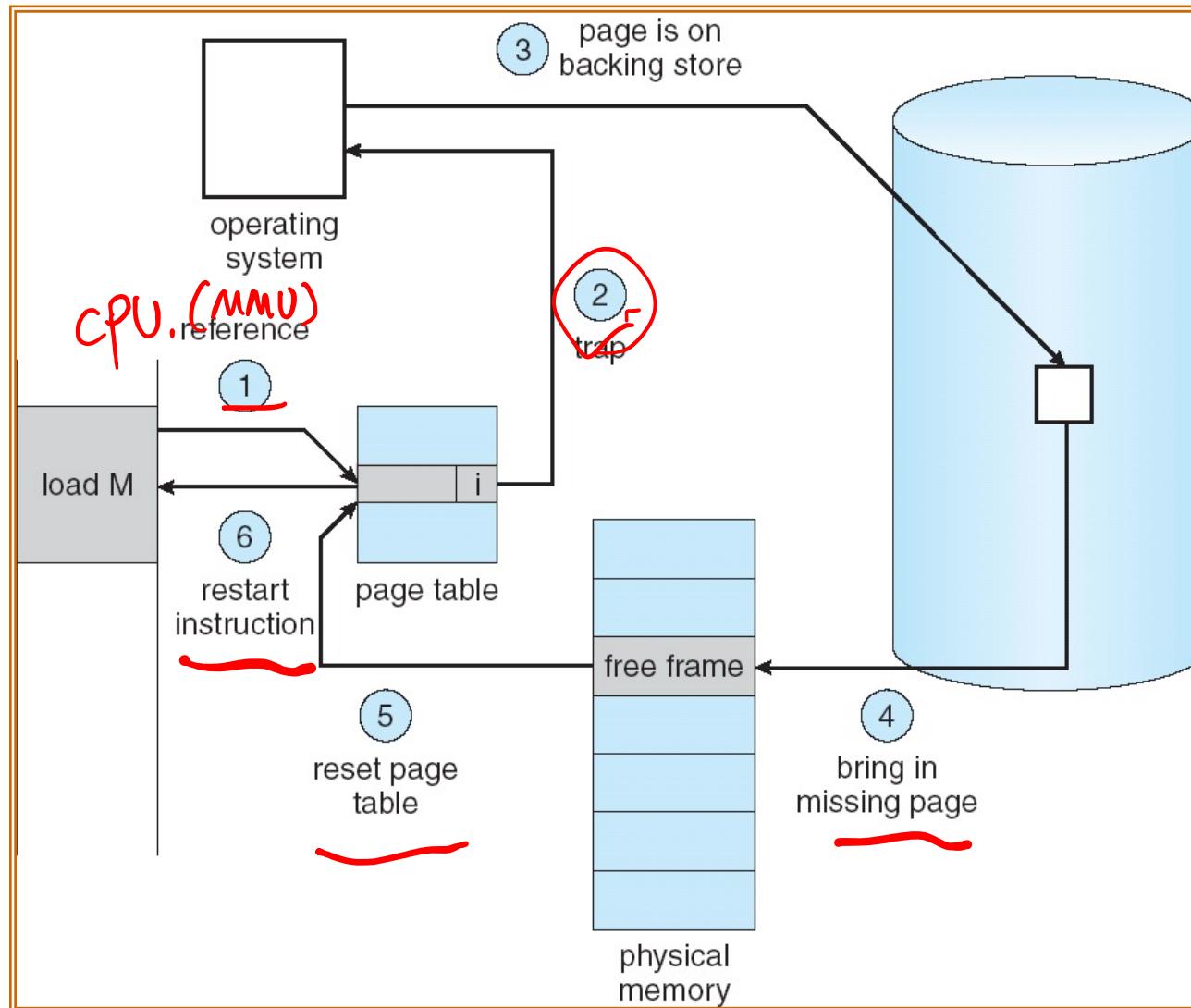
What to read?

(cheap) **PTE**.present = 1

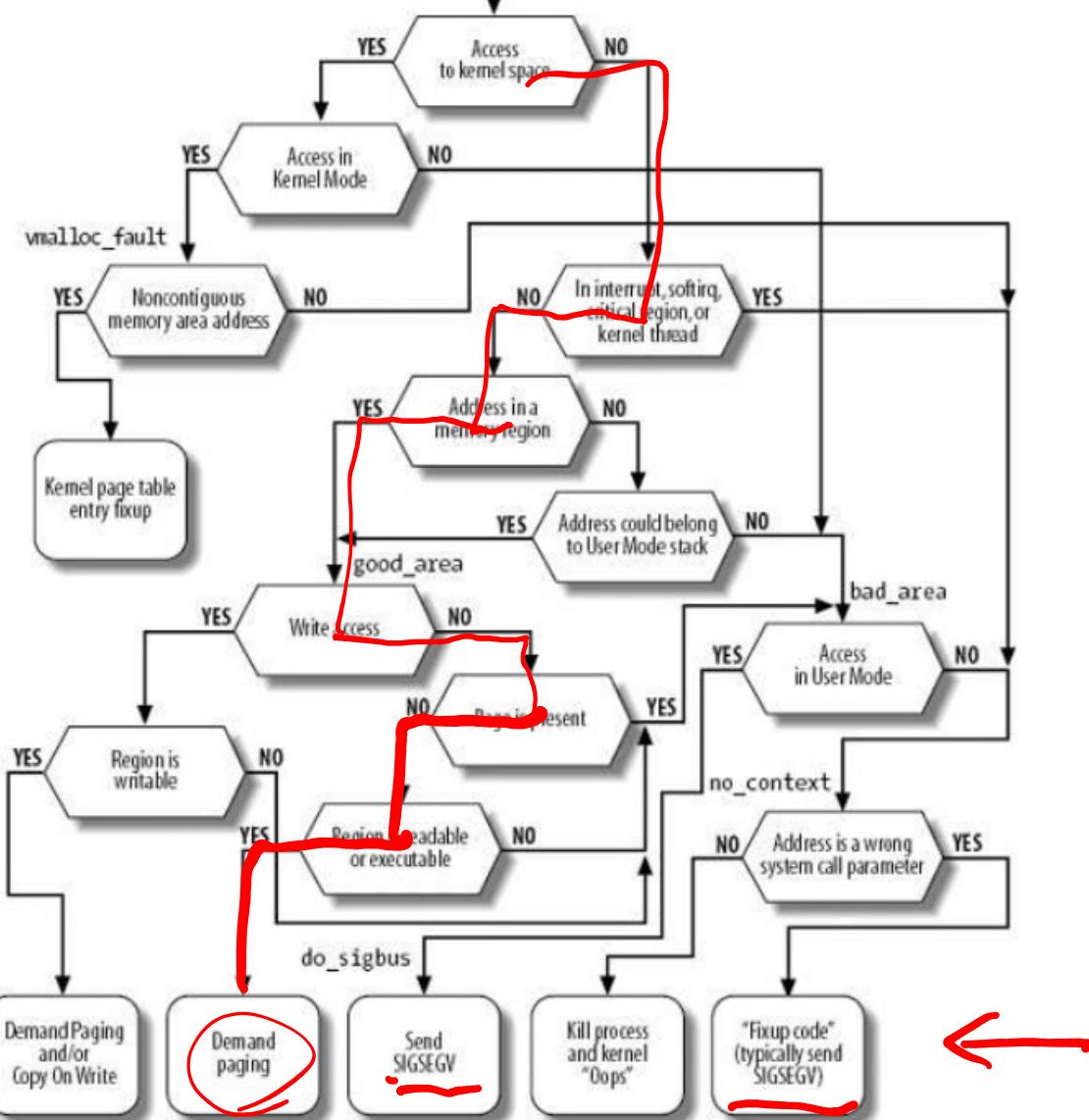
(cheap) **PTE.PFN** = **PFN**

(cheap) retry instruction

# Major Steps of A Page Fault



# Reality: The Page Fault Handler



Complex logic:  
Easier to read  
code than read a  
book!

# Impact of Page Faults

- Each page fault affects the system performance negatively
  - The process experiencing the page fault will not be able to continue until the missing page is brought to the main memory
  - The process will be **blocked** (moved to the waiting state)
  - Dealing with the page fault involves disk I/O
    - Increased demand to the disk drive
    - Increased waiting time for process experiencing page fault

# Memory as a Cache

- As we increase the degree of multiprogramming, **over-allocation of memory** becomes a problem
- What if we are unable to find a free frame at the time of the page fault?
- OS chooses to **page out** one or more pages to make room for new page(s) OS is about to bring in
  - The process to replace page(s) is called **page replacement policy**

# Memory as a Cache

- OS keeps a small portion of memory free proactively
  - High watermark (HW) and low watermark (LW)
- When OS notices free memory is below LW (i.e.,  
memory pressure)
  - A background thread (i.e., swap/page daemon) starts running to free memory
  - It evicts pages until there are HW pages available

# Beyond Physical Memory: Policies – What to Evict?

# Page Replacement

- Page replacement completes the separation between the logical memory and the physical memory
  - Large virtual memory can be provided on a smaller physical memory
- Impact on performance
  - If there are no free frames, two page transfers needed at each page fault!
- We can use a **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written back to disk

# Page Replacement Policy

- Formalizing the problem
  - Cache management: Physical memory is a cache for virtual memory pages in the system
  - Primary objective:
    - High performance
    - High efficiency
    - Low cost
  - Goal: **Minimize cache misses**
    - To minimize # times OS has to fetch a page from disk
    - -OR- **maximize cache hits**

# Average Memory Access Time

- Average (or effective) memory access time (**AMAT**) is the metric to calculate the effective memory performance

$$AMAT = (P_{Hit} \cdot T_M) + (P_{Miss} \cdot T_D)$$

- $T_M$ : Cost of accessing memory
- $T_D$ : Cost of accessing disk
- $P_{Hit}$ : Probability of finding data in cache (hit)
  - Hit rate
- $P_{Miss}$ : Probability of not finding data in cache (miss)
  - Miss rate

$$1 - P_{hit.}$$

100%.

# An Example

- Assuming
  - $T_M$  is 100 nanoseconds (ns),  $T_D$  is 10 milliseconds (ms)
  - $P_{\text{Hit}}$  is 0.9, and  $P_{\text{Miss}}$  is 0.1
  - $\text{AMAT} = 0.9 * 100\text{ns} + 0.1 * 10\text{ms} = 90\text{ns} + 1\text{ms} = 1.00009\text{ms}$  → 0.9 ms.
  - Or around 1 millisecond
- What if the hit rate is 99.9%?
  - Result changes to 10.1 microseconds (or  $\mu\text{s}$ )
  - Roughly 100 times faster!

0.9 - 0.91

1,000  $\mu\text{s}$ .

# First-In First-Out (FIFO)

# First-in First-out (FIFO)

- Simplest page replacement algorithm
- Idea: items are evicted in the order they are inserted
- Implementation: FIFO queue holds identifiers of all the pages in memory
  - We replace the page at the head of the queue
  - When a page is brought into memory, it is inserted at the tail of the queue

# FIFO Replacement Policy

- Idea: items are evicted in the order they are inserted
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

# FIFO Replacement Policy

- Idea: items are evicted in the order they are inserted
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

assume cache size 3

Access	Hit/Miss?	Evict	Resulting Cache State
0			
1			
2			
0			
1			
3			
0			
3			
1			
2			
1			

# FIFO Replacement Policy

- Idea: items are evicted in the order they are inserted
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

assume cache size 3

Access	Hit/Miss?	Evict	Resulting Cache State	
0	Miss		First-in →	0
1	Miss		First-in →	0, 1
2	Miss		First-in →	0, 1, 2
0				
1				
3				
0				
3				
1				
2				
1				

# FIFO Replacement Policy

- Idea: items are evicted in the order they are inserted
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State	assume cache size 3
0	Miss		First-in→ 0	
1	Miss		First-in→ 0, 1	
2	Miss		First-in→ 0, 1, 2	
0	Hit		First-in→ 0, 1, 2	
1				
3				
0				
3				
1				
2				
1				

# FIFO Replacement Policy

- Idea: items are evicted in the order they are inserted
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State	assume cache size 3
0	Miss	First-in→	0	
1	Miss	First-in→	0, 1	
2	Miss	First-in→	0, 1, 2	
0	Hit	First-in→	0, 1, 2	
1	Hit	First-in→	0, 1, 2	
3				
0				
3				
1				
2				
1				

# FIFO Replacement Policy

- Idea: items are evicted in the order they are inserted
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State	assume cache size 3
0	Miss	First-in→	0	
1	Miss	First-in→	0, 1	
2	Miss	First-in→	0, 1, 2	
0	Hit	First-in→	0, 1, 2	
1	Hit	First-in→	0, 1, 2	
3	Miss			
0				
3				
1				
2				
1				

# FIFO Replacement Policy

- Idea: items are evicted in the order they are inserted
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State	assume cache size 3
0	Miss		First-in→ 0	
1	Miss		First-in→ 0, 1	
2	Miss		First-in→ 0, 1, 2	
0	Hit		First-in→ 0, 1, 2	
1	Hit		First-in→ 0, 1, 2	
3	Miss	0	First-in→ 1, 2, 3	
0				
3				
1				
2				
1				

# FIFO Replacement Policy

- Idea: items are evicted in the order they are inserted
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State	assume cache size 3
0	Miss		First-in→ 0	
1	Miss		First-in→ 0, 1	
2	Miss		First-in→ 0, 1, 2	
0	Hit •		First-in→ 0, 1, 2	
1	Hit •		First-in→ 0, 1, 2	
3	Miss	0	First-in→ 1, 2, 3	
0	Miss	1	First-in→ 2, 3, 0	
3	Hit •		First-in→ 2, 3, 0	
1	Miss	2	First-in→ 3, 0, 1	
2	Miss	3	First-in→ 0, 1, 2	
1	Hit •		First-in→ 0, 1, 2	

# FIFO Replacement Policy

- Idea: items are evicted in the order they are inserted
- **Issue:** the “oldest” page may contain a heavily used data
  - Will need to bring back that page in near future

# FIFO Replacement Policy

*Working Set Size.  
(WSS)*

- FIFO: items are evicted in the order they are inserted

→ Example workload: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

↓  
(a) size 3

Access	Hit	State (after)
1		
2		
3		
4		
1		
2		
5		
1		
2		
3		
4		
5		

↓  
(b) size 4

Access	Hit	State (after)
1		
2		
3		
4		
1		
2		
5		
1		
2		
3		
4		
5		

# FIFO Replacement Policy

- FIFO: items are evicted in the order they are inserted
- Example workload: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

(a) size 3

Access	Hit	State (after)
1	no	1
2	no	1,2
3	no	1,2,3
4	no	2,3,4
1	no	3,4,1
2	no	4,1,2
5	no	1,2,5
1	yes	1,2,5
2	yes	1,2,5
3	no	2,5,3
4	no	5,3,4
5	yes	5,3,4

(b) size 4 *d*

Access	Hit	State (after)
1		
2		
3		
4		
1		
2		
5		
1		
2		
3		
4		
5		

# FIFO Replacement Policy

- FIFO: items are evicted in the order they are inserted
- Example workload: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

(a) size 3

Access	Hit	State (after)
1	no	1
2	no	1,2
3	no	1,2,3
4	no	2,3,4
1	no	3,4,1
2	no	4,1,2
5	no	1,2,5
1	yes	1,2,5
2	yes	1,2,5
3	no	2,5,3
4	no	5,3,4
5	yes	5,3,4

(b) size 4

Access	Hit	State (after)
1	no	1
2	no	1,2
3	no	1,2,3
4	no	1,2,3,4
1	yes	1,2,3,4
2	yes	1,2,3,4
5	no	2,3,4,5
1	no	3,4,5,1
2	no	4,5,1,2
3	no	5,1,2,3
4	no	1,2,3,4
5	no	2,3,4,5

# Belady's Anomaly

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  - Size-3 (3-frames) case results in 9 page faults
  - Size-4 (4-frames) case results in 10 page faults
- Program runs potentially slower w/ more memory!
- Belady's anomaly
  - More frames → more page faults for some access pattern

# Random

# Random Policy

- Idea: picks a random page to replace
- Simple to implement like FIFO
- No intelligence of preserving locality

# Random Policy

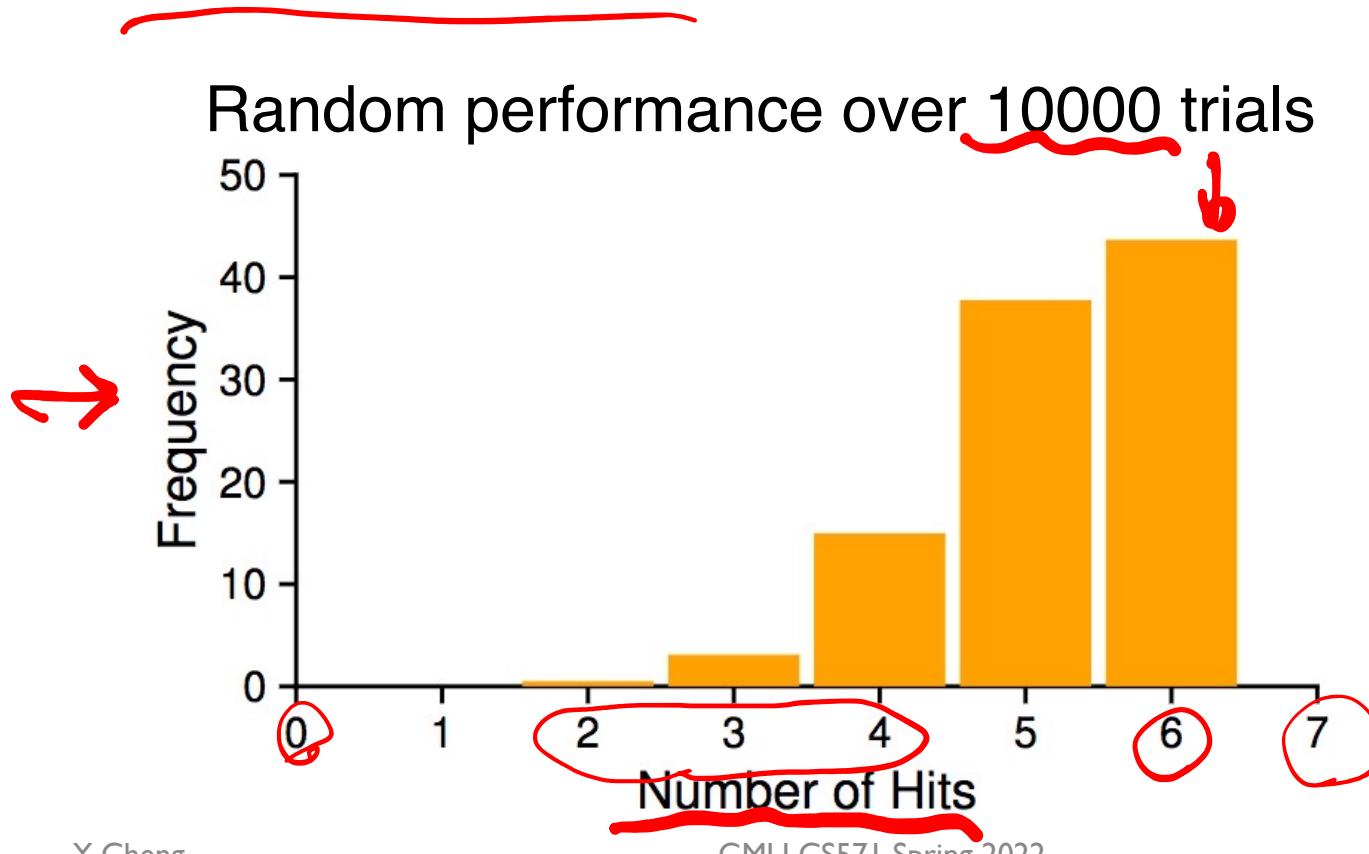
- Idea: picks a random page to replace
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	0	1, 2, 3
0	Miss	1	2, 3, 0
3	Hit		2, 3, 0
1	Miss	3	2, 0, 1
2	Hit		2, 0, 1
1	Hit		2, 0, 1

assume  
cache size 3

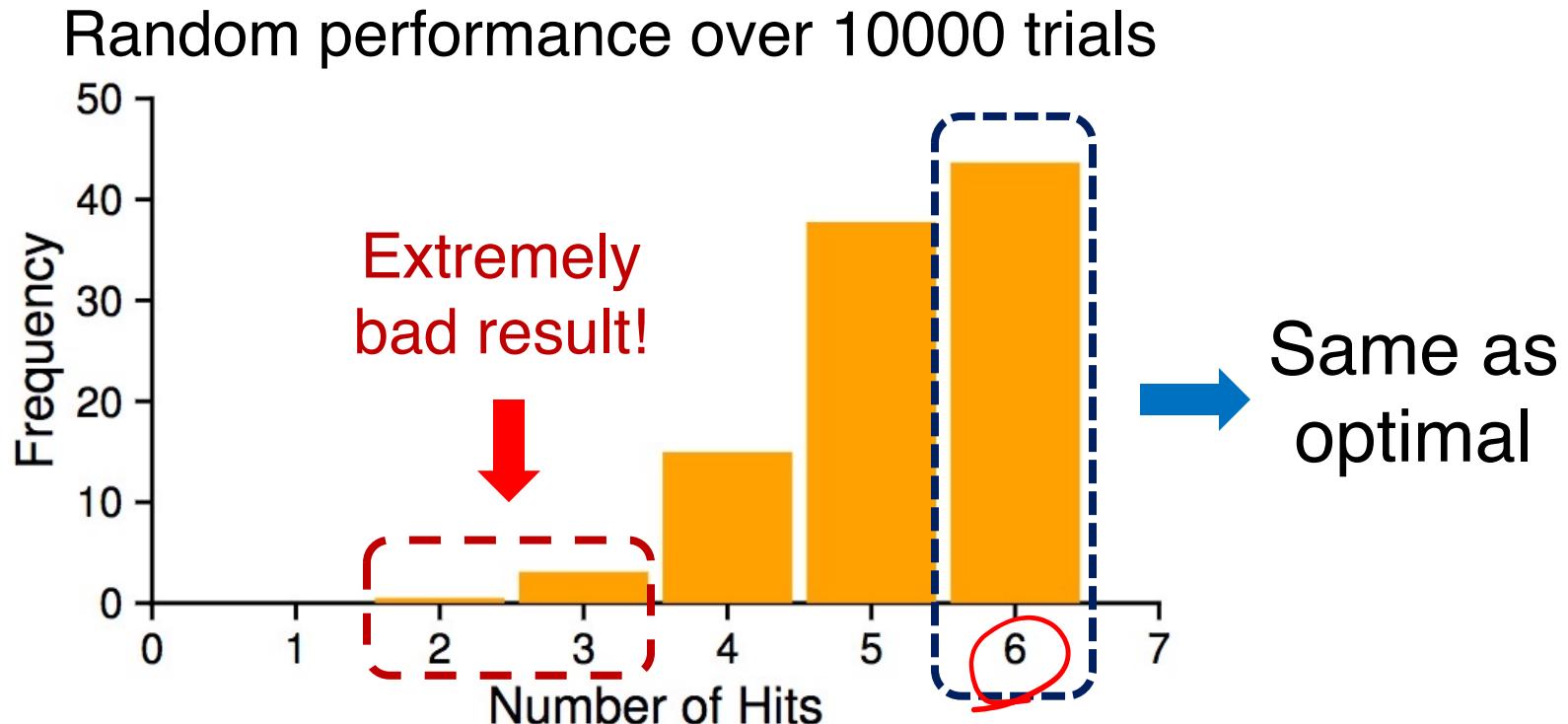
# How Random Policy Performs?

- Depends entirely on **how lucky you are**
- Example workload: 0 1 2 0 1 3 0 3 0 1 2 1



# How Random Policy Performs?

- Depends entirely on **how lucky you are**
- Example workload: 0 1 2 0 1 3 0 3 0 1 2 1



# Least-Recently-Used (LRU)

# Least-Recently-Used Policy (LRU)

- Use the recent pass as an approximation of the near future (**using history**)
- Idea: evict the page that has not been used for the longest period of time

# Least-Recently-Used Policy (LRU)

- Idea: evict the page that has not been used for the longest period of time
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0			
1			
2			
0			
1			
3			
0			
3			
1			
2			
1			

# Least-Recently-Used Policy (LRU)

- Idea: evict the page that has not been used for the longest period of time
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State	Cache size of 3
0	Miss	LRU→	0	← MRU
1	Miss	LRU→	0, 1	← MRU
2	Miss	LRU→	0, 1, 2	← MRU.
0				
1				
3				
0				
3				
1				
2				
1				

# Least-Recently-Used Policy (LRU)

- Idea: evict the page that has not been used for the longest period of time
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss	LRU→	0
1	Miss	LRU→	0, 1
2	Miss	LRU→	0, 1, 2
0	Hit	LRU→	1, 2, 0
1			
3			
0			
3			
1			
2			
1			

# Least-Recently-Used Policy (LRU)

- Idea: evict the page that has not been used for the longest period of time
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss	LRU→	0
1	Miss	LRU→	0, 1
2	Miss	LRU→	0, 1, 2
0	Hit	LRU→	1 2, 0
1	Hit	LRU→	2, 0, 1
3			
0			
3			
1			
2			
1			

# Least-Recently-Used Policy (LRU)

- Idea: evict the page that has not been used for the longest period of time
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		LRU→ 0
1	Miss		LRU→ 0, 1
2	Miss		LRU→ 0, 1, 2
0	Hit		LRU→ 1, 2, 0
1	Hit		LRU→ 2, 0, 1
3	Miss	2	LRU→ 0, 1, 3
0			
3			
1			
2			
1			

# Least-Recently-Used Policy (LRU)

- Idea: evict the page that has not been used for the longest period of time
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		LRU→ 0
1	Miss		LRU→ 0, 1
2	Miss		LRU→ 0, 1, 2
0	Hit		LRU→ 1, 2, 0
1	Hit		LRU→ 2, 0, 1
3	Miss	2	LRU→ 0, 1, 3
0	Hit		LRU→ 1, 3, 0
3			
1			
2			
1			

# Least-Recently-Used Policy (LRU)

- Idea: evict the page that has not been used for the longest period of time
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		LRU→ 0
1	Miss		LRU→ 0, 1
2	Miss		LRU→ 0, 1, 2
0	Hit		LRU→ 1, 2, 0
1	Hit		LRU→ 2, 0, 1
3	Miss	2	LRU→ 0, 1, 3
0	Hit		LRU→ 1, 3, 0
3	Hit		LRU→ 1, 0, 3
1			
2			
1			

# Least-Recently-Used Policy (LRU)

- Idea: evict the page that has not been used for the longest period of time
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		LRU→ 0
1	Miss		LRU→ 0, 1
2	Miss		LRU→ 0, 1, 2
0	Hit		LRU→ 1, 2, 0
1	Hit		LRU→ 2, 0, 1
3	Miss	2	LRU→ 0, 1, 3
0	Hit		LRU→ 1, 3, 0
3	Hit		LRU→ 1, 0, 3
1	Hit		LRU→ 0, 3, 1
2			
1			

# Least-Recently-Used Policy (LRU)

- Idea: evict the page that has not been used for the longest period of time
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

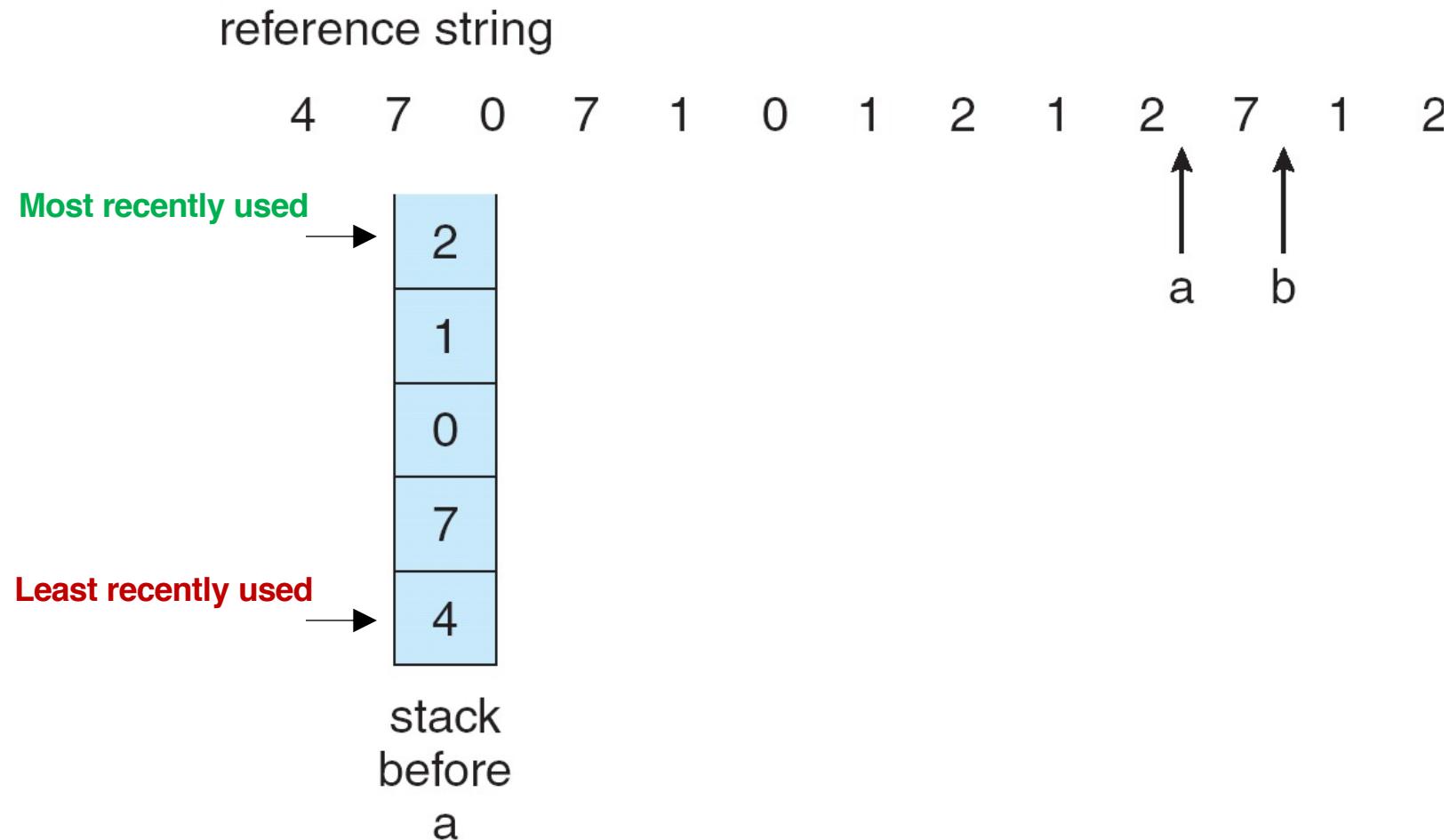
Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		LRU→ 0
1	Miss		LRU→ 0, 1
2	Miss		LRU→ 0, 1, 2
0	Hit		LRU→ 1, 2, 0
1	Hit		LRU→ 2 0, 1
3	Miss	2	LRU→ 0, 1, 3
0	Hit		LRU→ 1, 3, 0
3	Hit		LRU→ 1, 0, 3
1	Hit		LRU→ 0, 3, 1
2	Miss	0	LRU→ 3, 1, 2
1	Hit		LRU→ 3, 2, 1

# LRU Stack Implementation

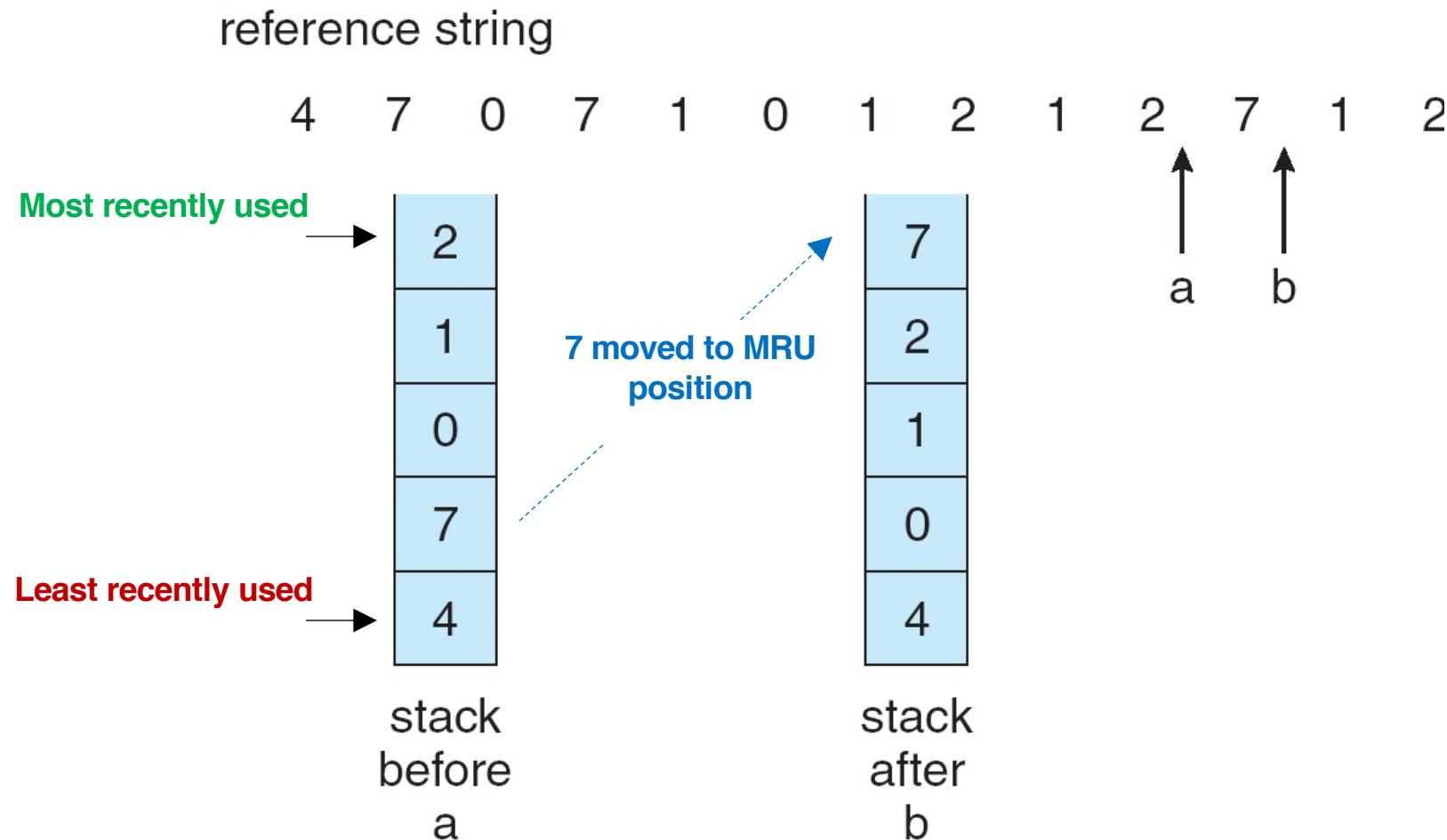
L F U  
↑  
PQ .

- Stack implementation: keep a stack of page numbers in a doubly linked list form
  - Page referenced, move it to the **top**
  - Requires quite a few pointers to be changed
  - **No search required** for replacement operation!

# Using a Stack to Approximate LRU



# Using a Stack to Approximate LRU



Offline. → Clairvoyant:

# Belady's Optimal

# MIN: The Optimal Replacement Policy

- Many years ago **Belady** demonstrated that there is a simple policy (MIN or OPT) which always leads to fewest number of misses
- Idea: evict the page that will be accessed furthest in the future
- Assumption: we know about the future
- Impossible to implement MIN in practice!
- But it is extremely useful as a **practical best-case baseline** for **comparison** purpose

# Proof of Optimality for Belady's MIN

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.307.7603&rep=rep1&type=pdf>

## A Short Proof of Optimality for the MIN Cache Replacement Algorithm

Benjamin Van Roy  
Stanford University

December 2, 2010

### Abstract

The MIN algorithm is an offline strategy for deciding which item to replace when writing a new item to a cache. Its optimality was first established by Mattson, Gecsei, Slutz, and Traiger [2] through a lengthy analysis. We provide a short and elementary proof based on a dynamic programming argument.

**Keywords:** analysis of algorithms, on-line algorithms, caching, paging

### 1 The MIN Algorithm

# MIN the Optimal

- Idea: evict the page that will be accessed furthest in the future
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

# MIN the Optimal

- Idea: evict the page that will be accessed furthest in the future
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0			
1			
2			
0			
1			
3			
0			
3			
1			
2			
1			
0	0		

assume  
cache size 3

# MIN the Optimal

- Idea: evict the page that will be accessed furthest in the future
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0			
1			
3			
0			
3			
1			
2			
1			

assume  
cache size 3

# MIN the Optimal

- Idea: evict the page that will be accessed furthest in the future
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3			
0			
3			
1			
2			
1			

assume  
cache size 3

# MIN the Optimal

- Idea: evict the page that will be accessed furthest in the future
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3			
0			
3			
1			
2			
1			

assume  
cache size 3

What to evict??

# MIN the Optimal

- Idea: evict the page that will be accessed furthest in the future
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3			
0			
3			
1			
2			
1			

assume  
cache size 3

Page 2 happens to  
be the one that will  
be accessed  
furthest in future!

2

What to evict??

# MIN the Optimal

- Idea: evict the page that will be accessed furthest in the future
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0			
3			
1			
2			
1			

assume  
cache size 3

# MIN the Optimal

- Idea: evict the page that will be accessed furthest in the future
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2			
1			

assume  
cache size 3

# MIN the Optimal

- Idea: evict the page that will be accessed furthest in the future
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2			
1			
0			

assume  
cache size 3

What to evict??

# MIN the Optimal

- Idea: evict the page that will be accessed furthest in the future
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2			
1			

Page 1 will be  
accessed right  
after page 2.  
Hence 1 is safe!

assume  
cache size 3

What to evict??

# MIN the Optimal

- Idea: evict the page that will be accessed furthest in the future
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	3	0, 1, 2
1			

assume  
cache size 3

# MIN the Optimal

- Idea: evict the page that will be accessed furthest in the future
- Example workload: 0 1 2 0 1 3 0 3 1 2 1 

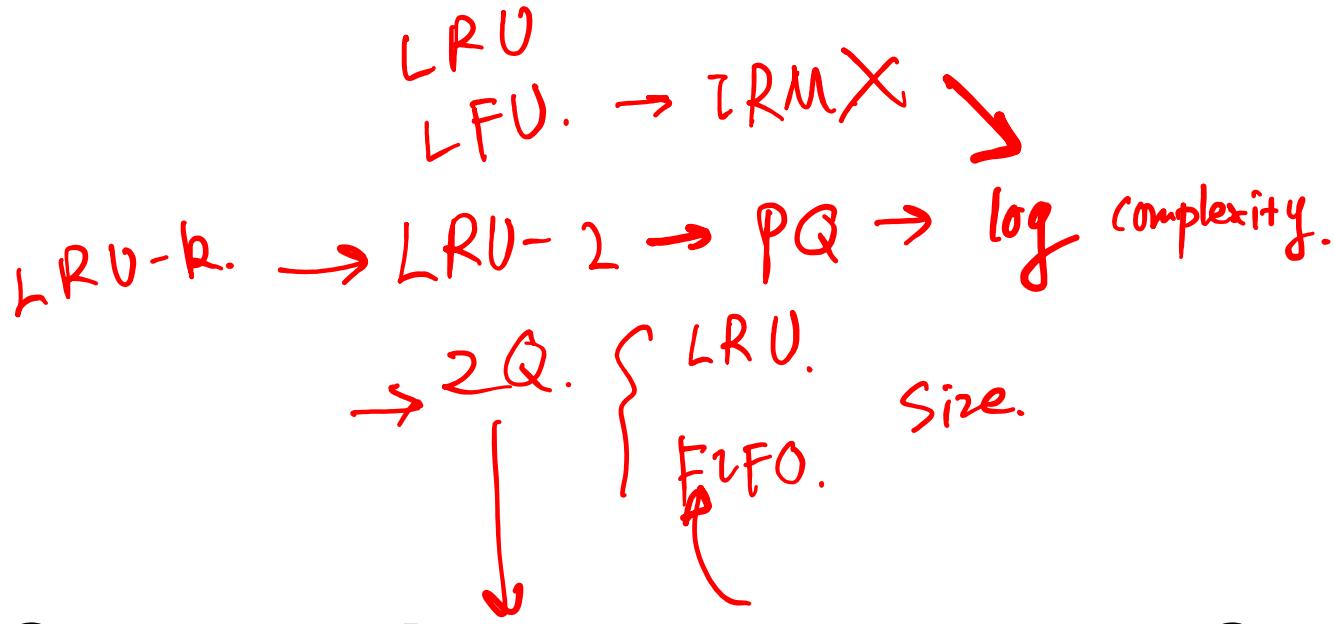
Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	3	0, 1, 2
1	Hit		0, 1, 2

assume  
cache size 3

# MIN the Optimal

- Idea: evict the page that will be accessed furthest in the future
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State	assume cache size 3
0	Miss		0	
1	Miss		0, 1	
2	Miss		0, 1, 2	
0	Hit		0, 1, 2	
1	Hit		0, 1, 2	
3	Miss	2	0, 1, 3	
0	Hit		0, 1, 3	
3	Hit		0, 1, 3	
1	Hit		0, 1, 3	
2	Miss	3	0, 1, 2	
1	Hit		0, 1, 2	



# ARC: Adaptive Replacement Cache

# Adaptive Replacement Cache

- ARC policy
  - Developed and patented by IBM
  - (...Dissuaded its adoption in open-source projects??)

## ARC: A SELF-TUNING, LOW OVERHEAD REPLACEMENT CACHE

Nimrod Megiddo and Dharmendra S. Modha  
IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120  
Email: {megiddo,dmodha}@almaden.ibm.com

**Abstract**— We consider the problem of cache management in a demand paging scenario with uniform page sizes. We propose a new cache management policy, namely, Adaptive Replacement Cache (ARC), that has several advantages.

In response to evolving and changing access patterns, ARC *dynamically, adaptively, and continually* balances between the recency and frequency components in an *online* and *self-tuning* fashion. The policy ARC uses a learning rule to adaptively and continually revise its assumptions about the workload.

The policy ARC is *empirically universal*, that is, it empirically performs as well as a certain *fixed replacement policy*—

compression [9] and list updating [10]. Any substantial progress in caching algorithms will affect the entire modern computational stack.

Consider a system consisting of two memory levels: *main* (or *cache*) and *auxiliary*. The cache is assumed to be significantly faster than the auxiliary memory, but is also significantly more expensive. Hence, the size of the cache memory is usually only a fraction of the size of the auxiliary memory. Both memories are managed in units of uniformly sized items known as

# Why ARC?

- Offline optimal (MIN): Replaces the page that has the greatest forward distance
  - Requires **knowledge of future**
  - Provides an **upper-bound**
- Recency (LRU)
  - **Most commonly used policy**
- Frequency (LFU)
  - Optimal under **independent reference model (IRM)**

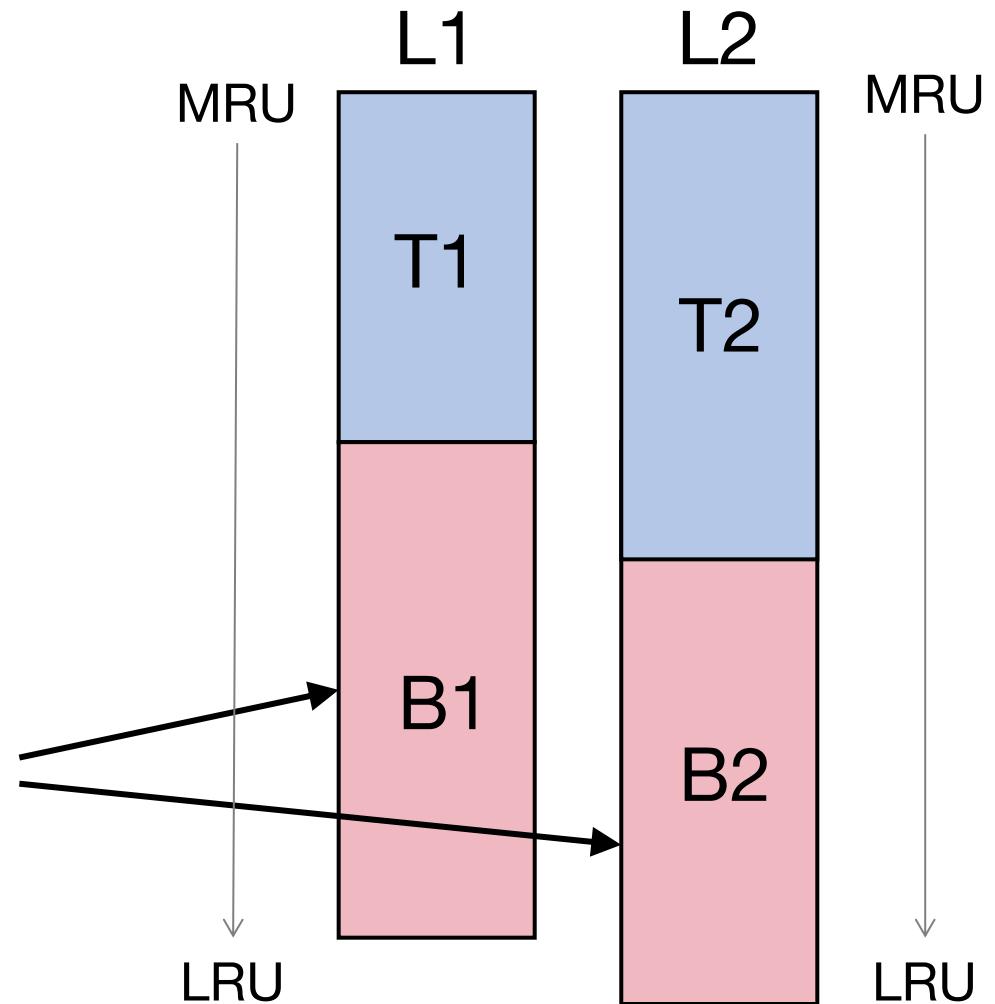
# Mechanisms

- Maintains two LRU lists
  - Pages that have been referenced **only once** (L1)
  - Pages that have been referenced **at least twice** (L2)
- Each list has the same length  $c$  as cache
- Cache contains tops of both lists: T1 and T2
- Bottoms B1 and B2 are not in cache
  - **Ghost cache**

# Mechanisms (cont.)

$$|T_1| + |T_2| = c$$

Ghost cache  
(pages not in memory)



# Policy

- ARC attempts to maintain a target size  $\text{target\_T1}$  for list T1 (parameter p)
  - ARC continually and dynamically revises  $\text{target\_T1}$
- When cache is full, ARC evicts:
  - The LRU page from T1 if:
$$|\text{T1}| \geq \text{target\_T1}$$
  - The LRU page from T2 otherwise

# Policy (cont.)

- If the missing page was in bottom B1 of L1:
  - ARC increases target\_T1
$$\text{target\_T1} = \min(\text{target\_T1} + \max(|B_2| / |B_1|, 1), c)$$
- If the missing page was in bottom B2 of L2:
  - ARC decreases target\_T1
$$\text{target\_T1} = \max(\text{target\_T1} - \max(|B_1| / |B_2|, 1), 0)$$

# Policy (cont.)

- Intuition
  - Two heuristics **compete** with each other
  - Each heuristic gets **rewarded** any time it can show that adding more pages to its top list would have avoided a cache miss
- ARC chooses whether it should care more about recency or frequency of access in eviction decisions
- Note that ARC has **no tunable parameter** (parameter-less)
  - Cannot get it wrong!

# Policy (cont.)

- ARC generally performs **much better** than LRU
  - Can achieve greater hit rates than LRU w/ the same cache size
  - Or, can achieve same hit rates as LRU w/ a much smaller cache

# Mini Exam 1

- 30 minutes next Wednesday, 03/09
  - 7:20 pm – 7:50 pm
  - Worksheet practices available on BB (course content)
- Open book, open note
- CPU scheduling
  - FIFO, SJF, RR, STCF
- Paging
  - VA → PA translation, PT

# Miscellaneous: TLB Caching

# TLB Replacement Policy

- Cache: When we want to add a new entry to a **full** TLB, an old entry must be evicted and replaced
- LRU policy
  - Intuition: A page entry that has not recently been used implies it won't likely to be used in the near future
- Random policy
  - Evicts an entry at random

# TLB Workloads

- Sequential array accesses can almost always hit in the TLB, and hence are very fast
- What pattern would be slow?

# TLB Workloads

- Sequential array accesses can almost always hit in the TLB, and hence are very fast
- What pattern would be slow?
  - Highly random, with no repeat accesses

# Workload Characteristics

## Workload A

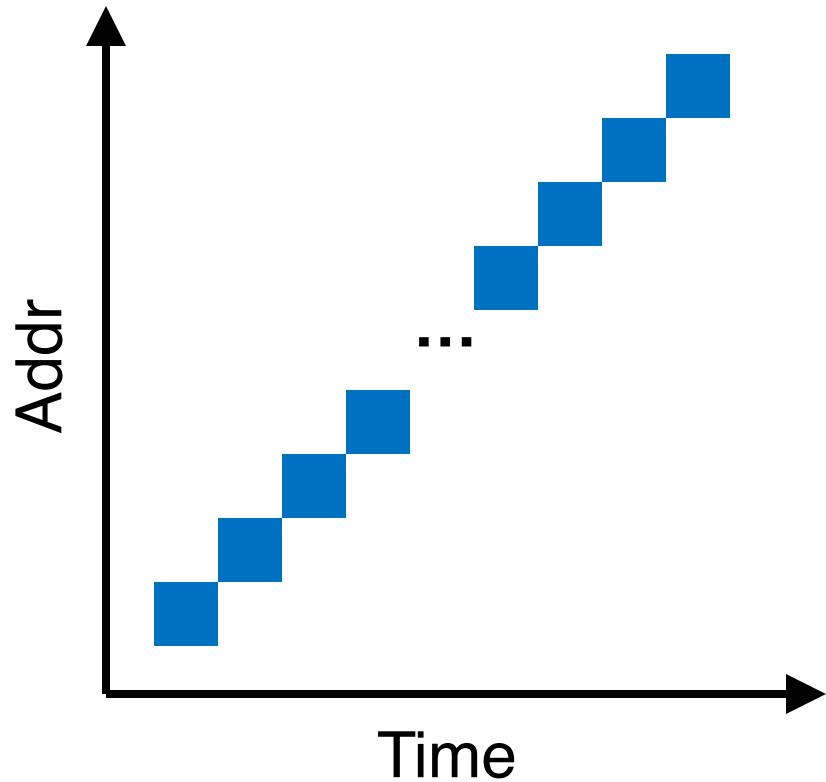
```
int sum = 0;  
for (i=0; i<1024; i++) {  
    sum += a[i];  
}
```

## Workload B

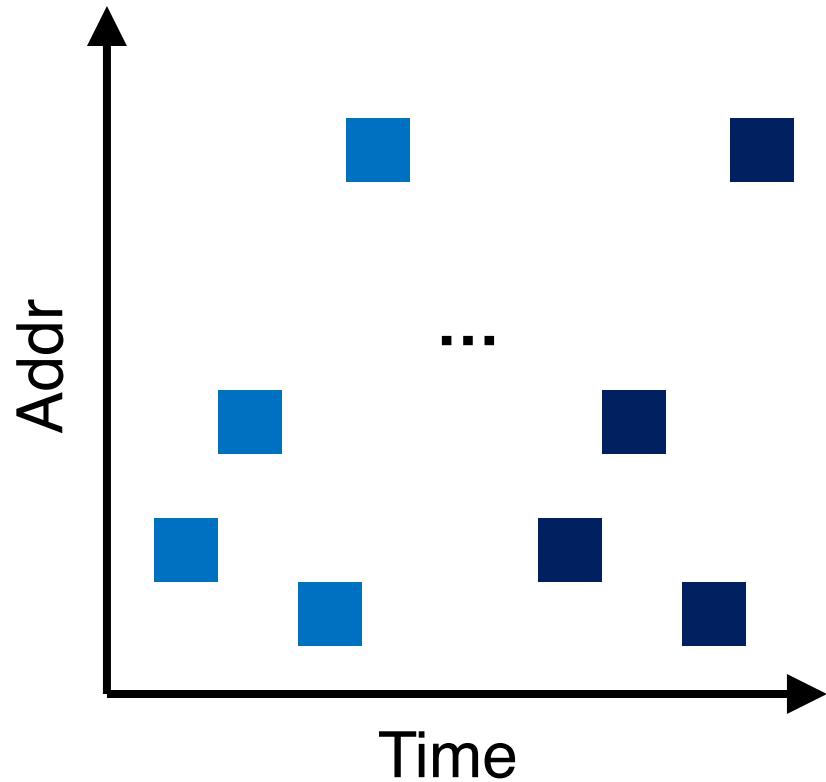
```
int sum = 0;  
srand(1234);  
for (i=0; i<512; i++) {  
    sum += a[rand() % N];  
}  
srand(1234); // same seed  
for (i=0; i<512; i++) {  
    sum += a[rand() % N];  
}
```

# Access Patterns

Workload A

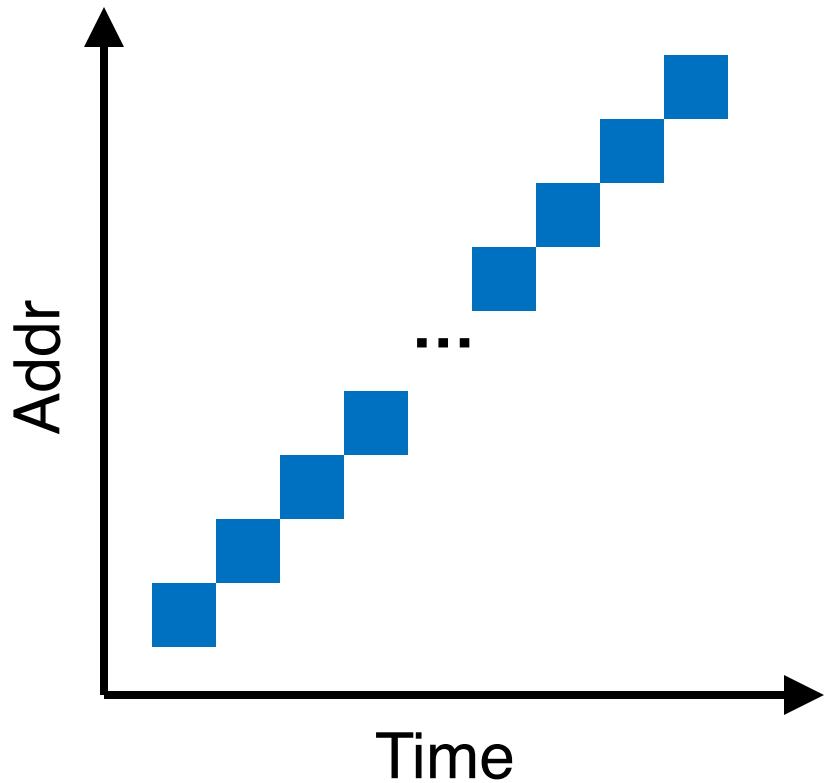


Workload B



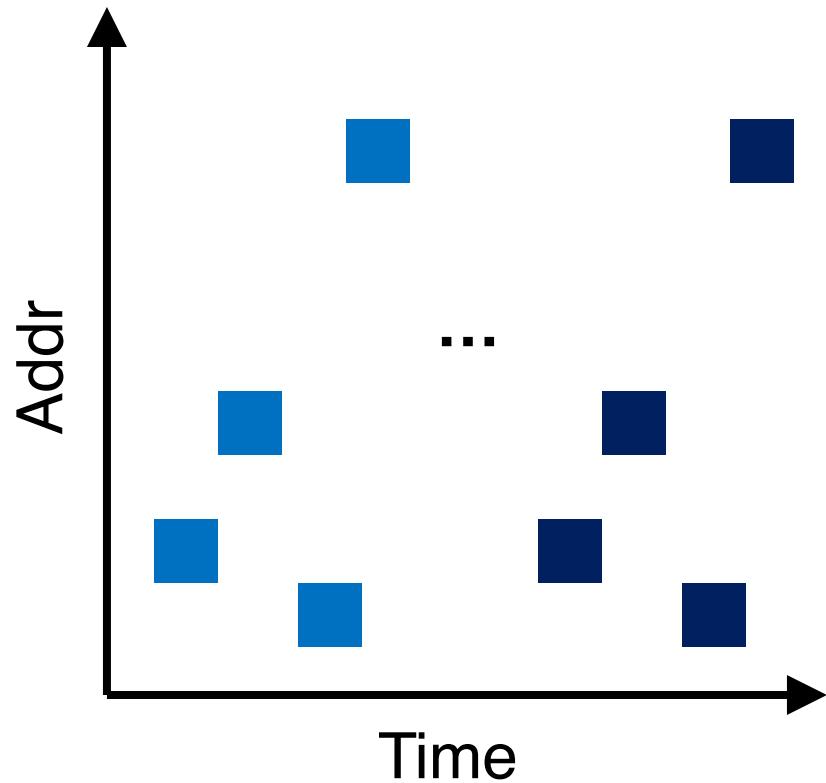
# Access Patterns

Workload A



**Spatial Locality**

Workload B



**Temporal Locality**

# Workload Locality

- **Spatial locality:**
  - Future access will be to nearby addresses
- **Temporal locality:**
  - Future access will be repeated to the same data

# Workload Locality

- **Spatial locality:**
  - Future access will be to nearby addresses
- **Temporal locality:**
  - Future access will be repeated to the same data
- Q: What TLB characteristics are best for each type?

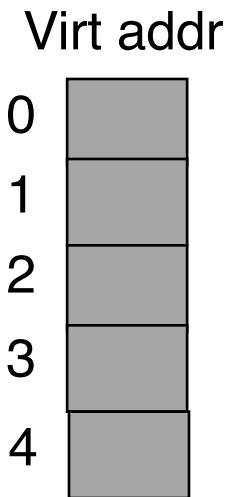
# Workload Locality

- **Spatial locality:**
  - Future access will be to nearby addresses
- **Temporal locality:**
  - Future access will be repeated to the same data
- Q: What TLB characteristics are best for each type?
  - One TLB entry holds the translation for one memory page: all accesses to that particular page benefit from this single TLB entry (**spatial** locality)
  - TLB is a small cache (if supporting LRU): memory accesses with **temporal** locality benefit

# TLB Replacement Policy

- Cache: When we want to add a new entry to a **full** TLB, an old entry must be evicted and replaced
- **Least-recently-used (LRU)** policy
  - Intuition: A page entry that has not recently been used implies it won't likely to be used in the near future
- **Random** policy
  - Evicts an entry at random

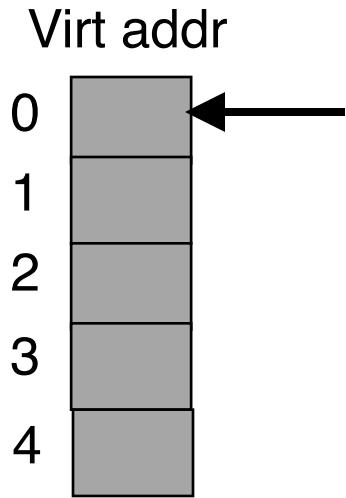
# LRU Trouble



CPU's TLB cache

Valid	Virt	Phys
0		
0		
0		
0		

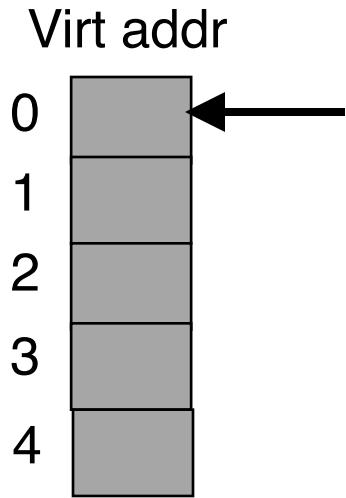
# LRU Trouble



CPU's TLB cache

Valid	Virt	Phys
1	0	?
0		
0		
0		

# LRU Trouble

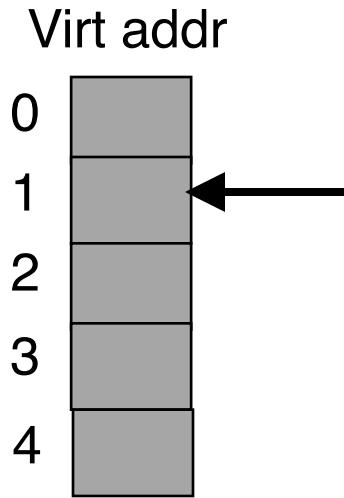


CPU's TLB cache

Valid	Virt	Phys
1	0	?
0		
0		
0		

**TLB miss**

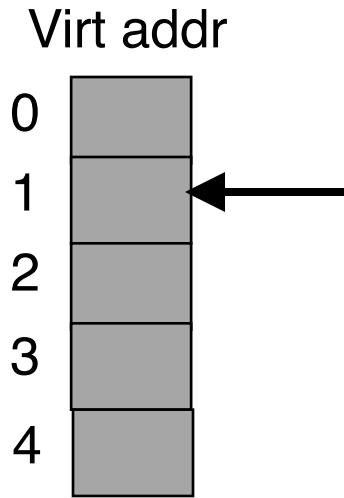
# LRU Trouble



CPU's TLB cache

Valid	Virt	Phys
1	0	?
1	1	?
0		
0		

# LRU Trouble

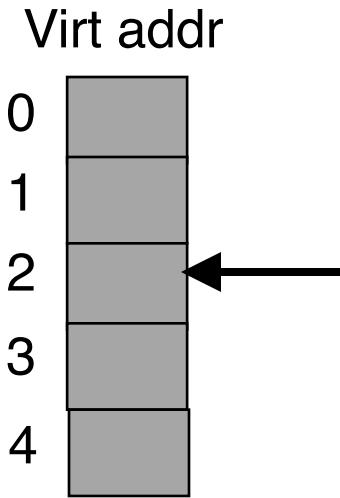


CPU's TLB cache

Valid	Virt	Phys
1	0	?
1	1	?
0		
0		

**TLB miss**

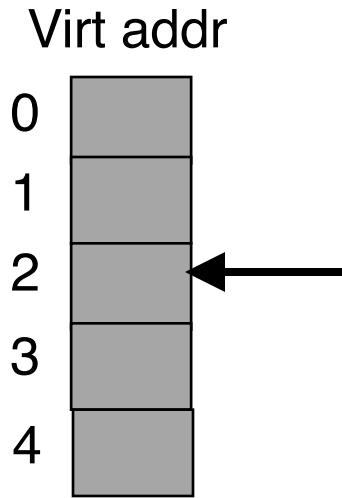
# LRU Trouble



CPU's TLB cache

Valid	Virt	Phys
1	0	?
1	1	?
1	2	?
0		

# LRU Trouble

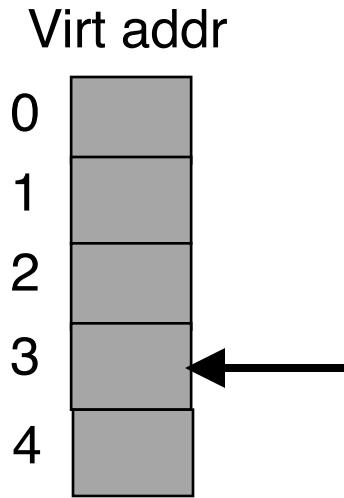


CPU's TLB cache

Valid	Virt	Phys
1	0	?
1	1	?
1	2	?
0		

**TLB miss**

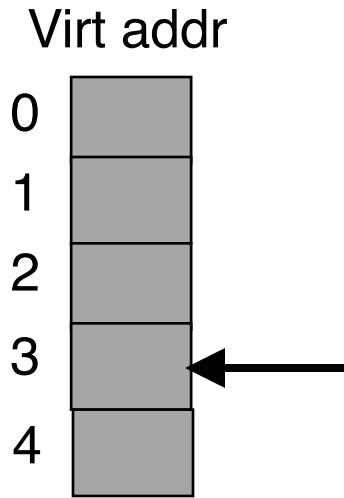
# LRU Trouble



CPU's TLB cache

Valid	Virt	Phys
1	0	?
1	1	?
1	2	?
1	3	?

# LRU Trouble

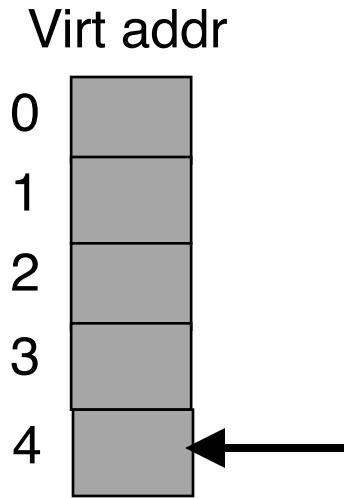


CPU's TLB cache

Valid	Virt	Phys
1	0	?
1	1	?
1	2	?
1	3	?

**TLB miss**

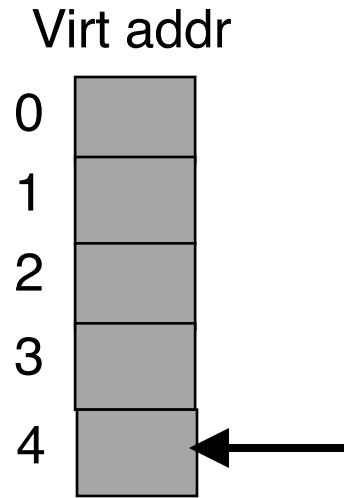
# LRU Trouble



CPU's TLB cache

Valid	Virt	Phys
1	0	?
1	1	?
1	2	?
1	3	?

# LRU Trouble

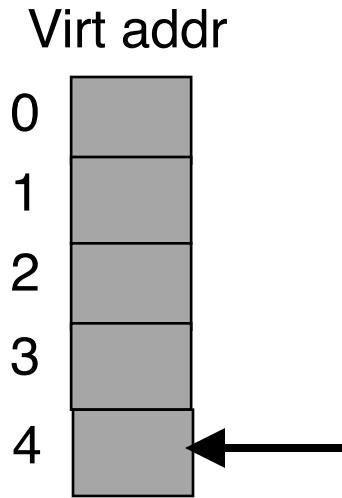


CPU's TLB cache

Valid	Virt	Phys
1	0	?
1	1	?
1	2	?
1	3	?

Now, **0** is the least-recently used item in TLB

# LRU Trouble

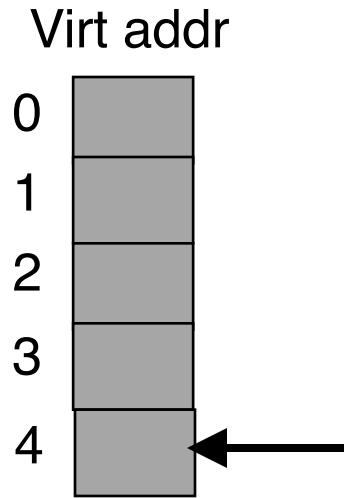


CPU's TLB cache

Valid	Virt	Phys
1	4	?
1	1	?
1	2	?
1	3	?

Replace 0 with 4

# LRU Trouble



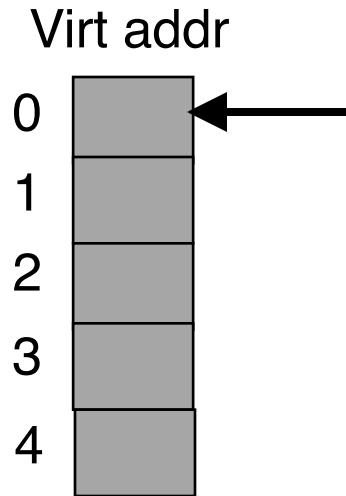
CPU's TLB cache

Valid	Virt	Phys
1	4	?
1	1	?
1	2	?
1	3	?

**TLB miss**

Replace 0 with 4

# LRU Trouble

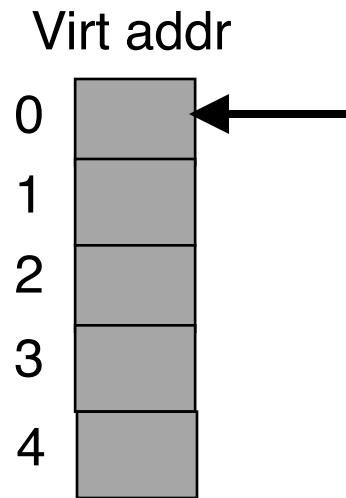


CPU's TLB cache

Valid	Virt	Phys
1	4	?
1	1	?
1	2	?
1	3	?

Accessing 0 again, which was unfortunately just evicted...

# LRU Trouble



CPU's TLB cache

Valid	Virt	Phys
1	4	?
1	0	?
1	2	?
1	3	?

## TLB miss

Accessing 0 again, which was unfortunately just evicted...  
Replace 1 (which is the least-recently used item at this point) with 0...

# Takeaway

- LRU
- Random
- When is each better?
  - Sometimes random is better than a “smart” policy!