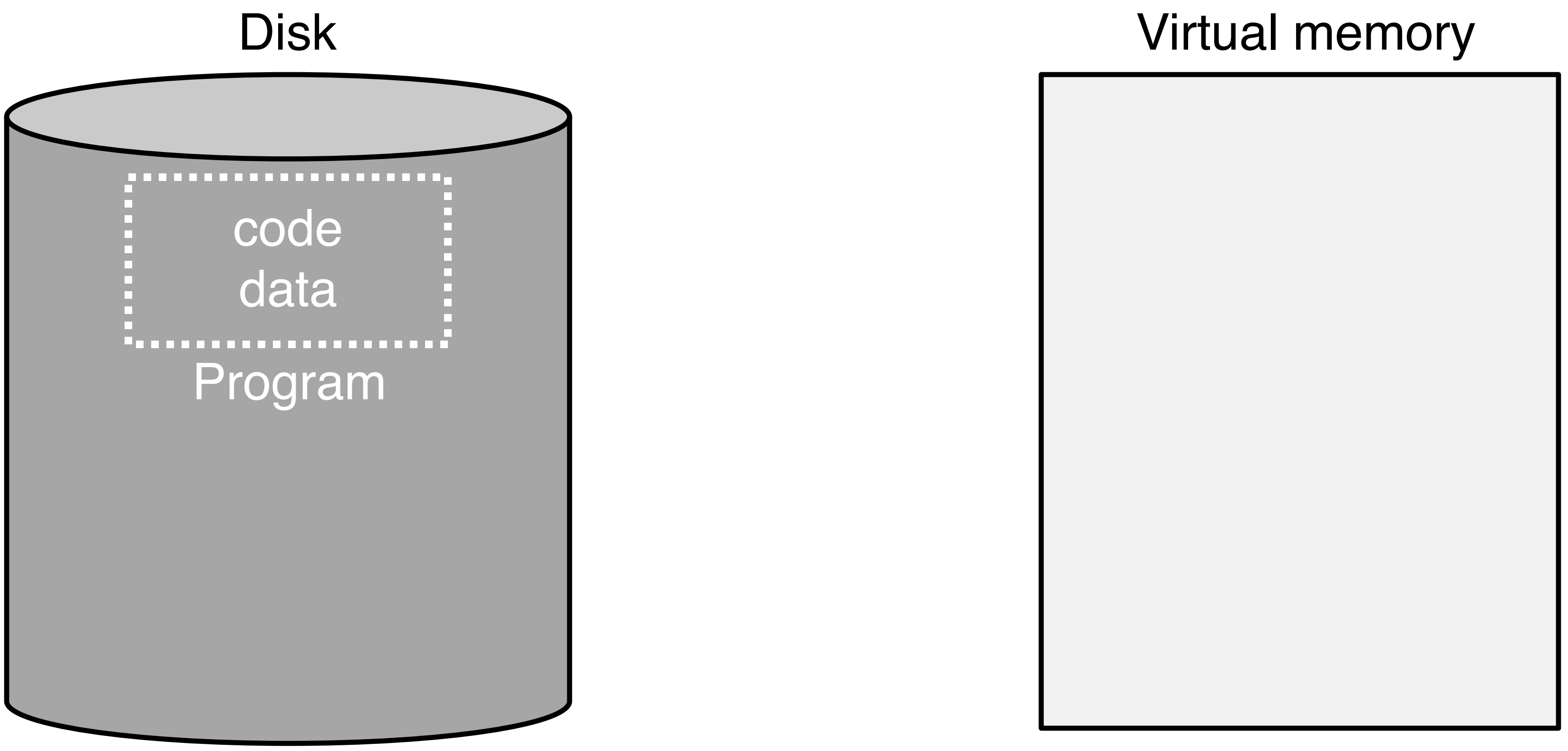
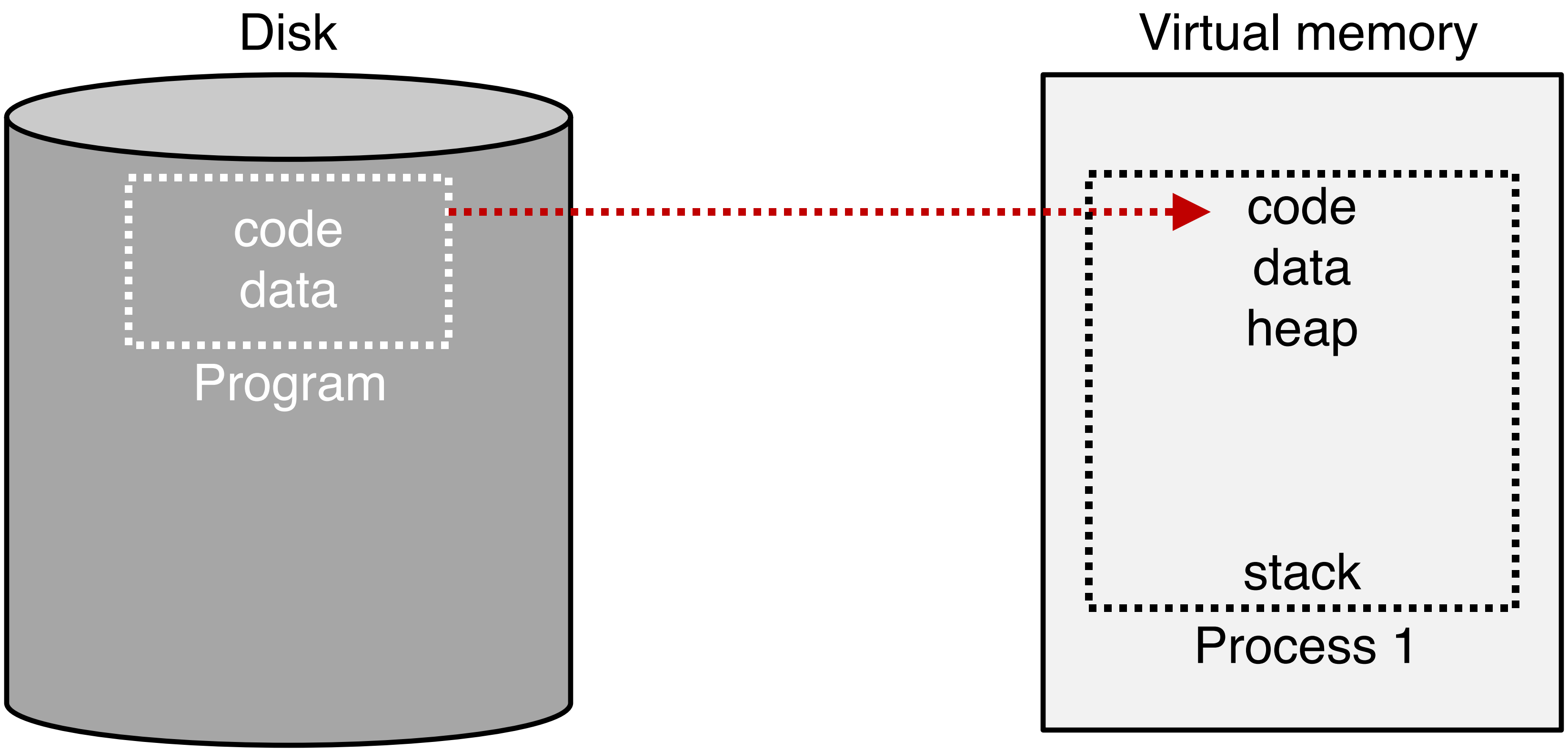
CS 471 Operating Systems

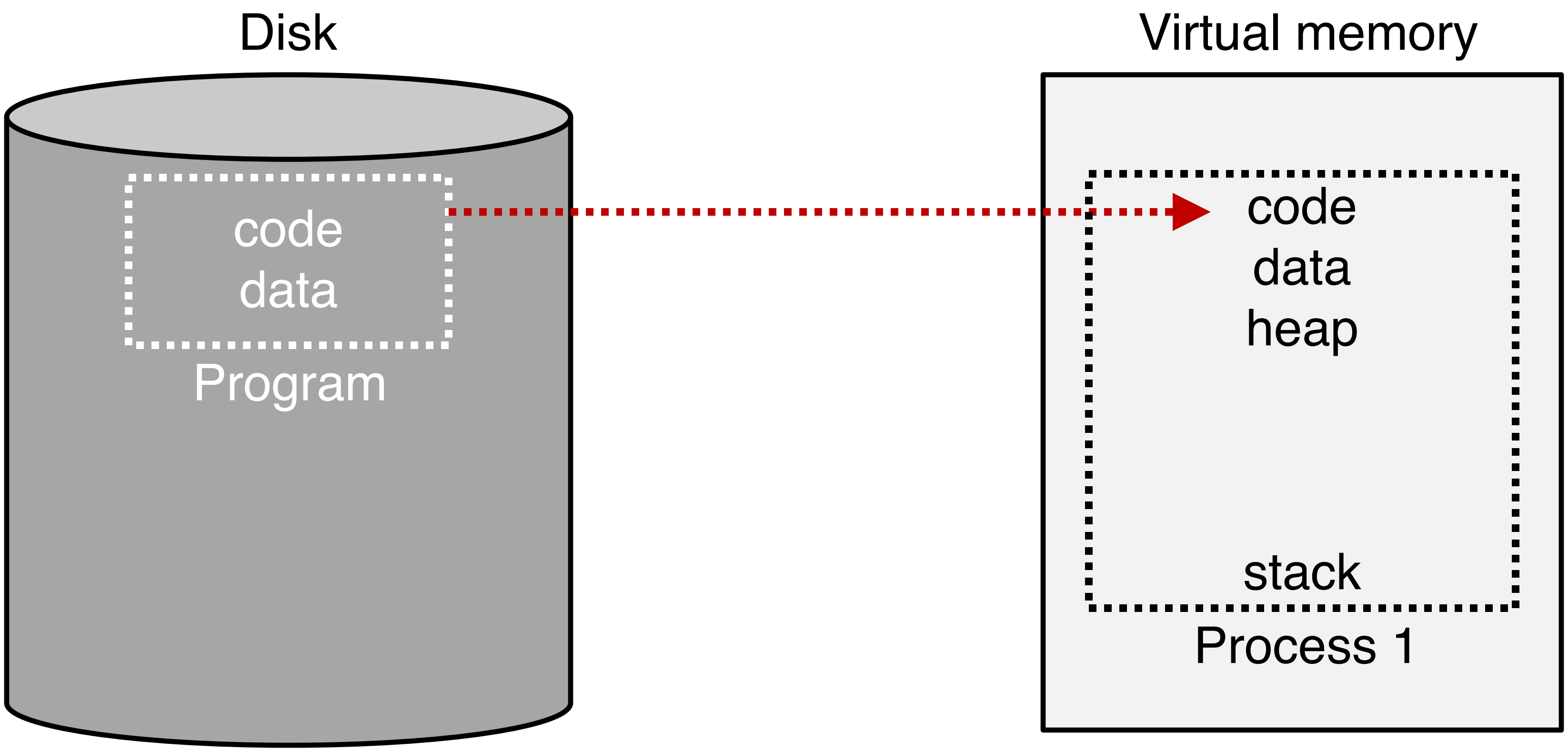
Yue Cheng

George Mason University
Spring 2019

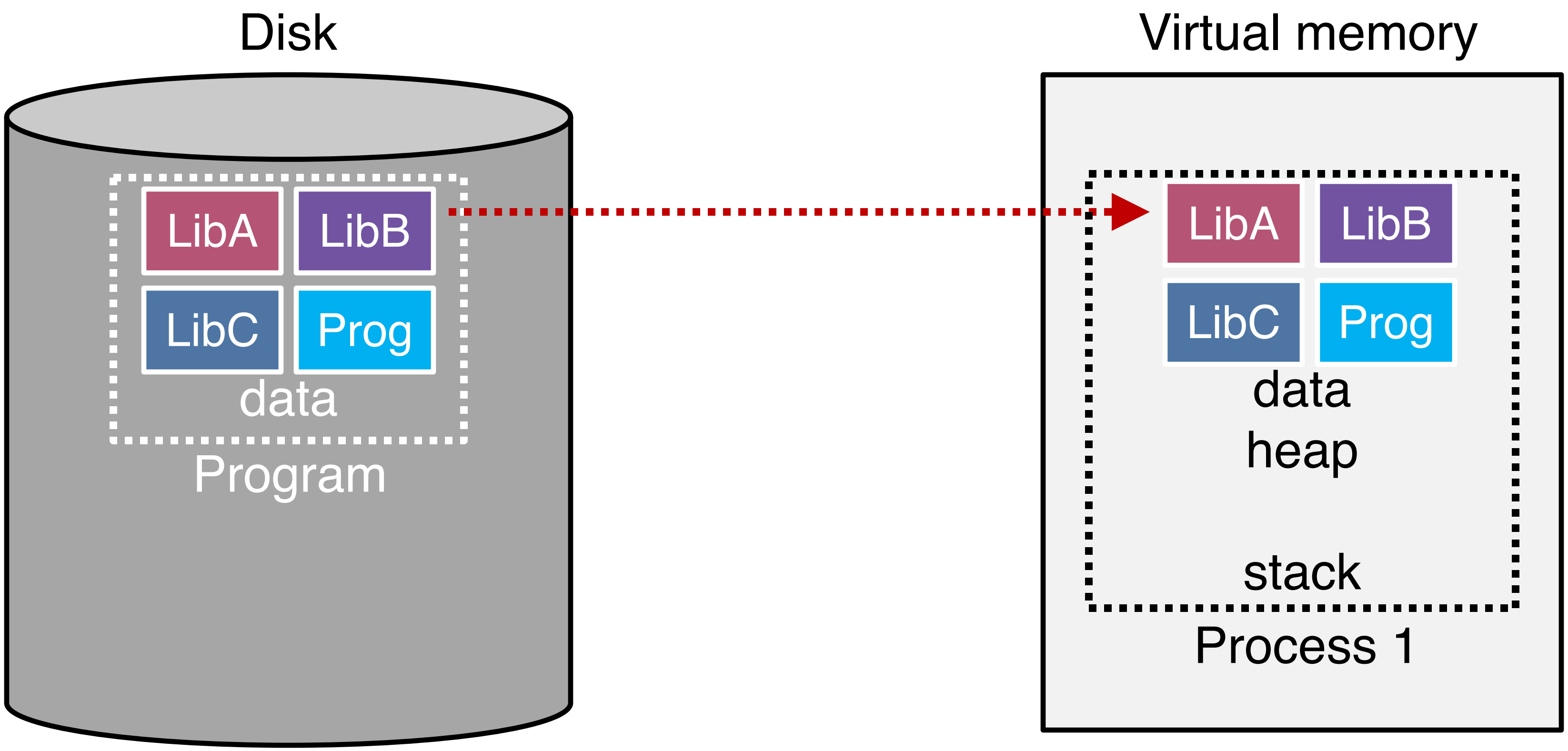
Swapping:
Beyond Physical Memory


Disk
code
data
Program
Virtual memory


Disk
code
data
Program
Virtual memory
code
data
heap
stack
Process 1

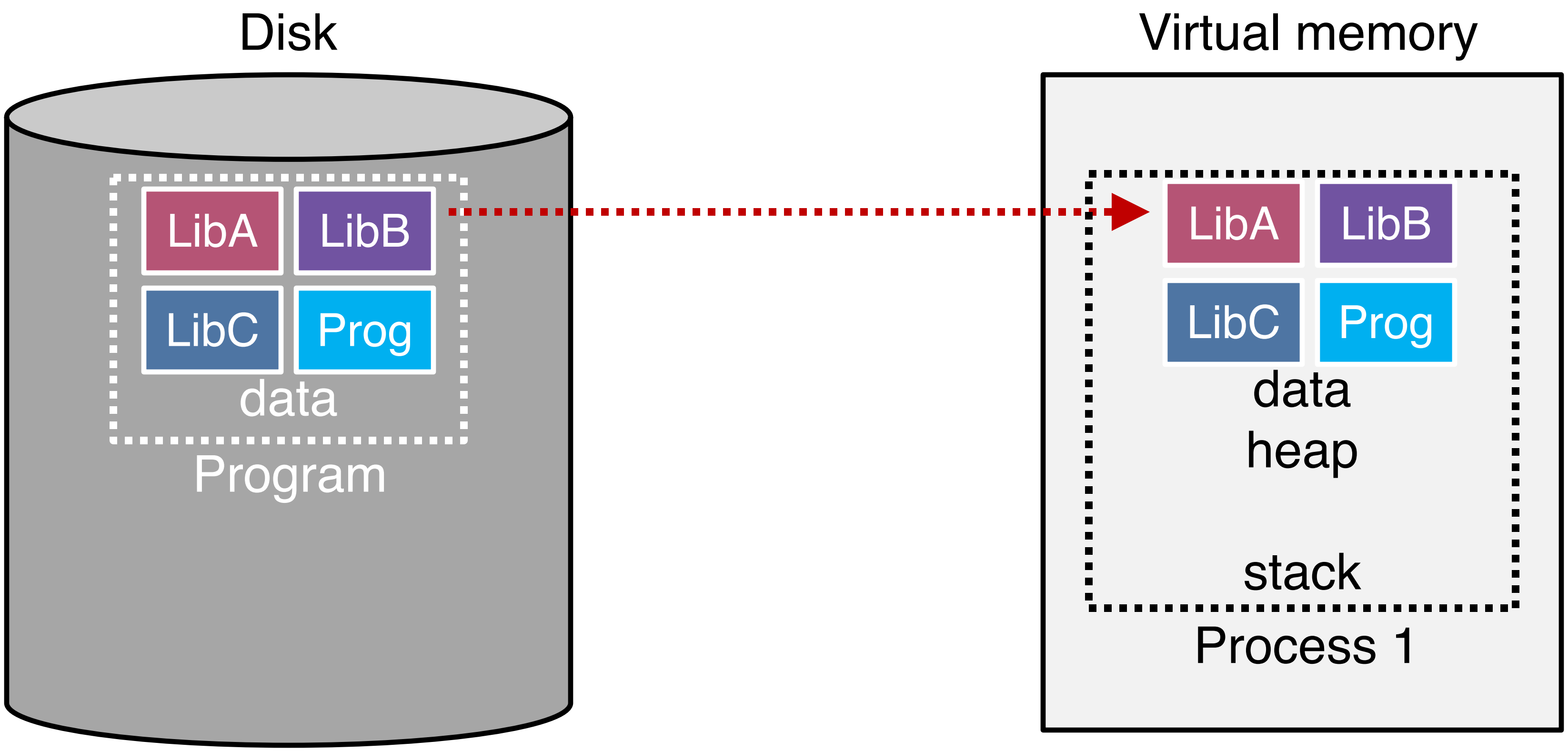


What's in code?

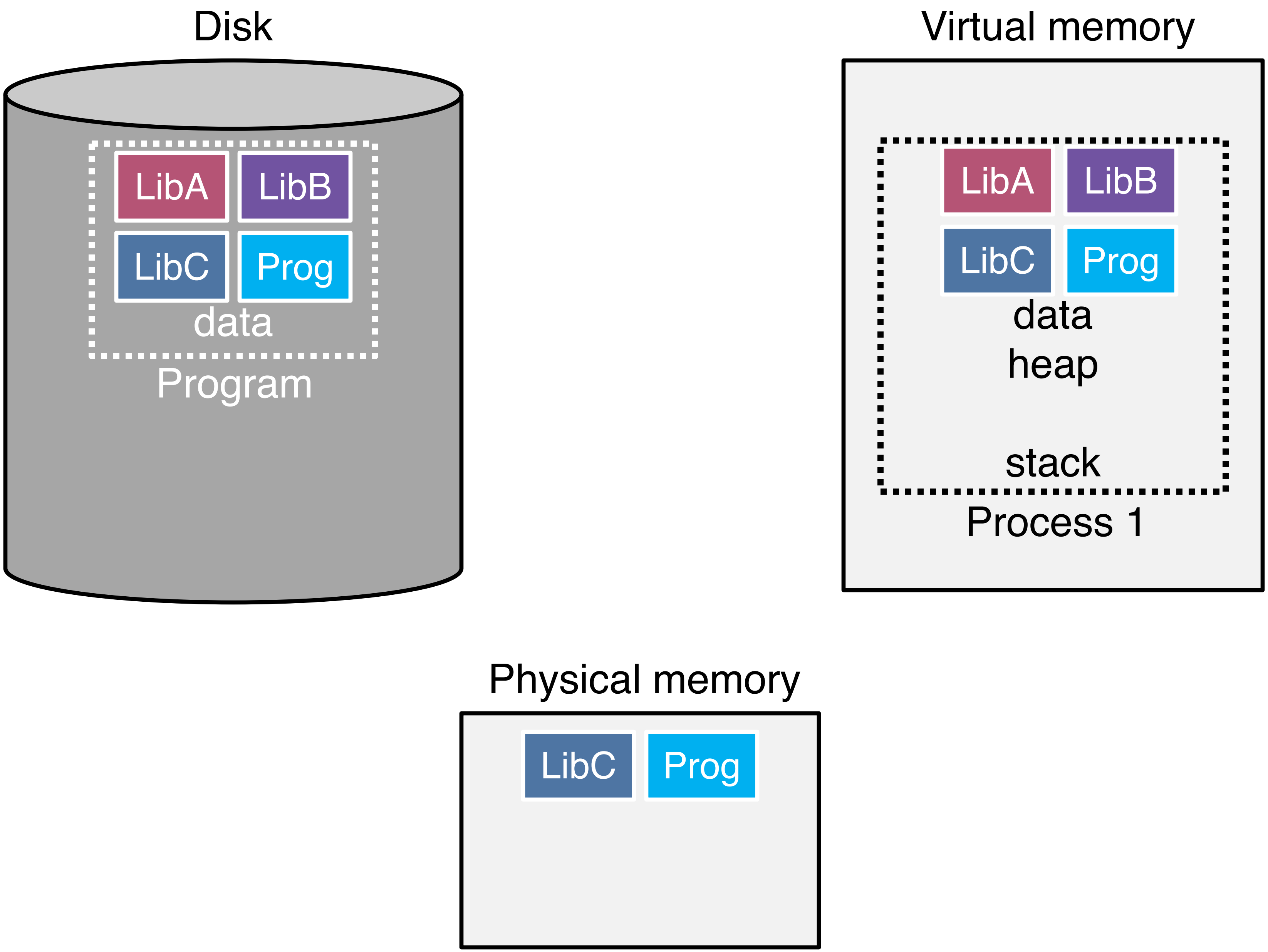


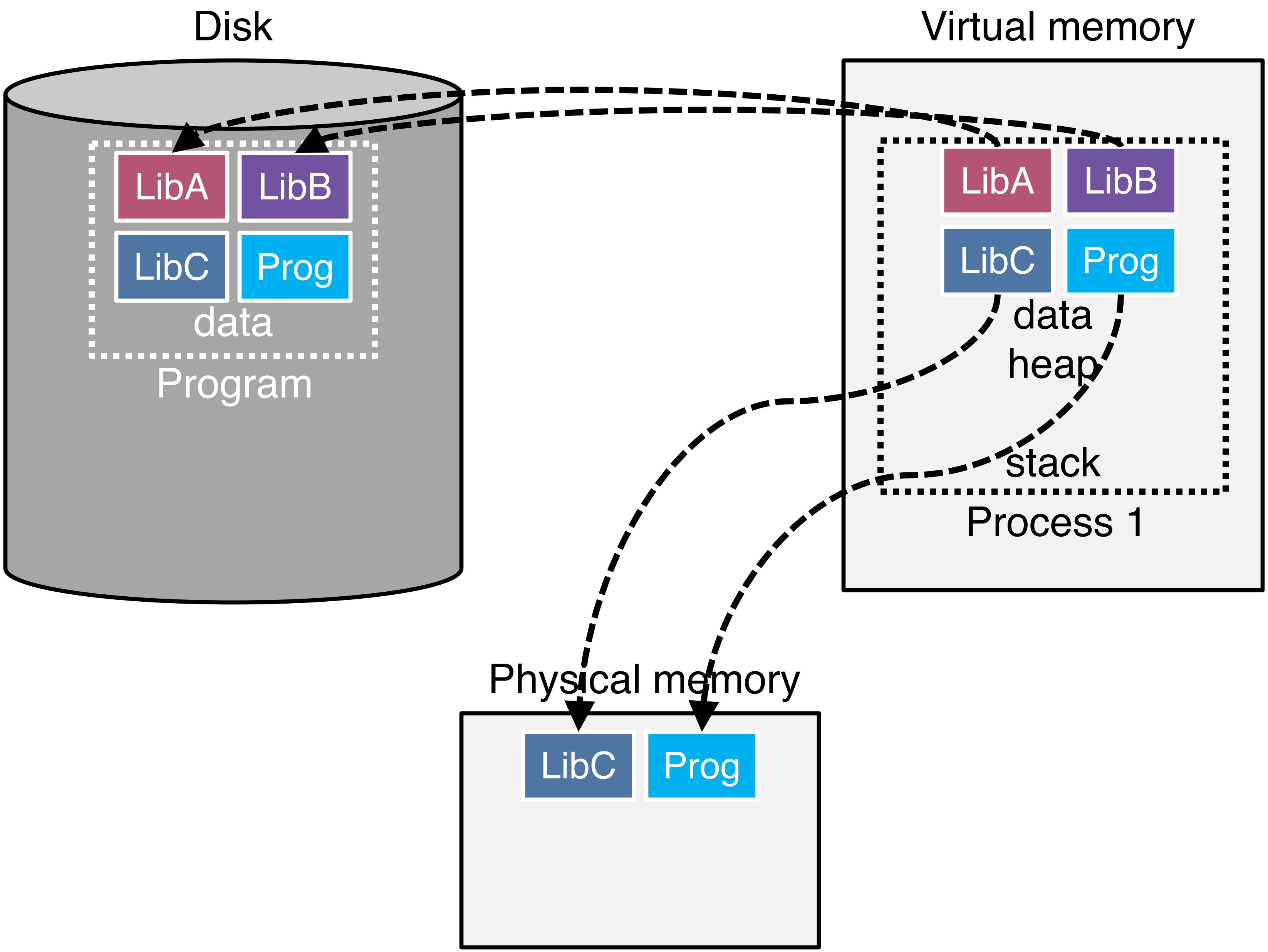
What's in code?

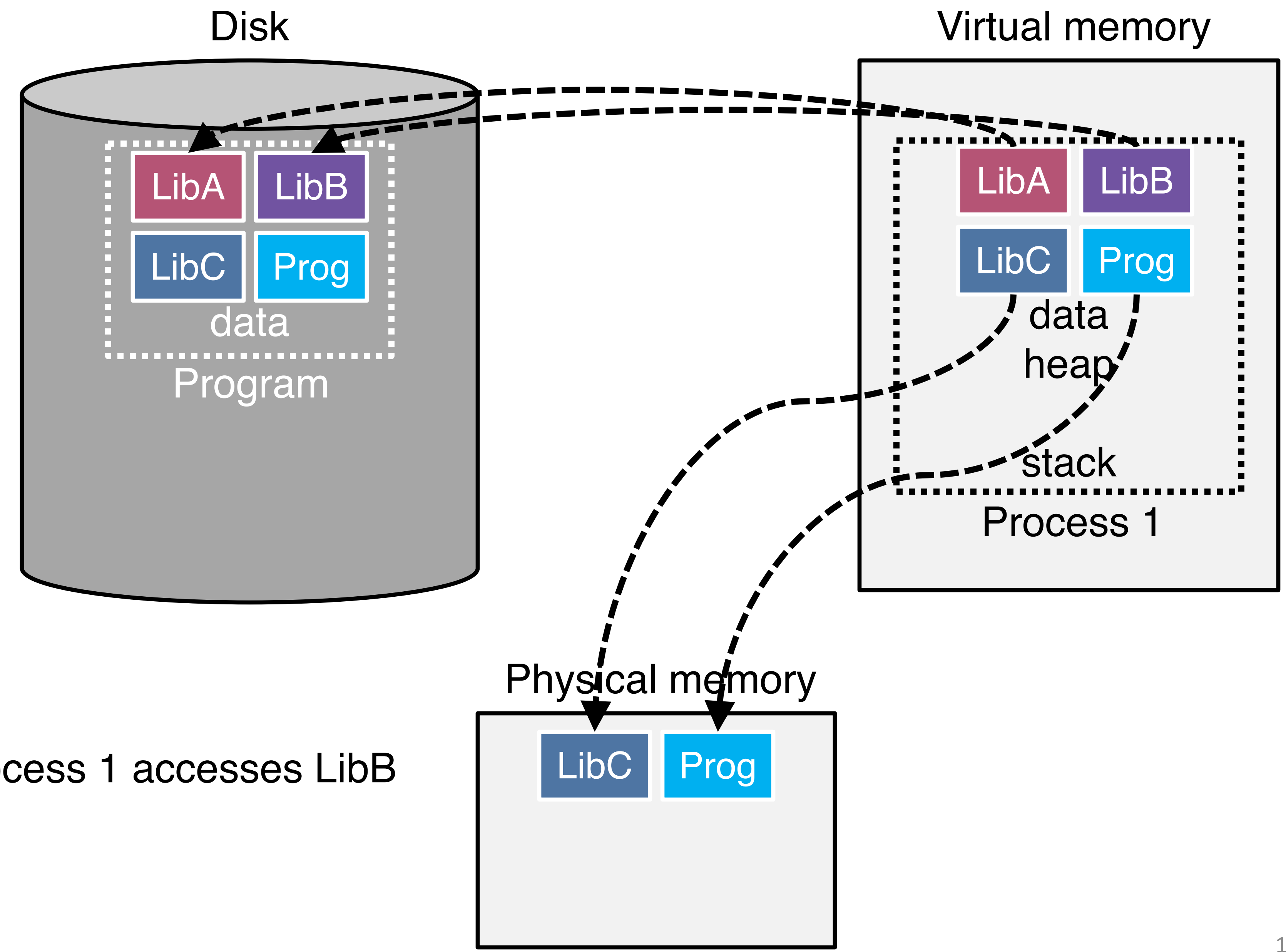
Many large libraries, some of which are rarely/never used



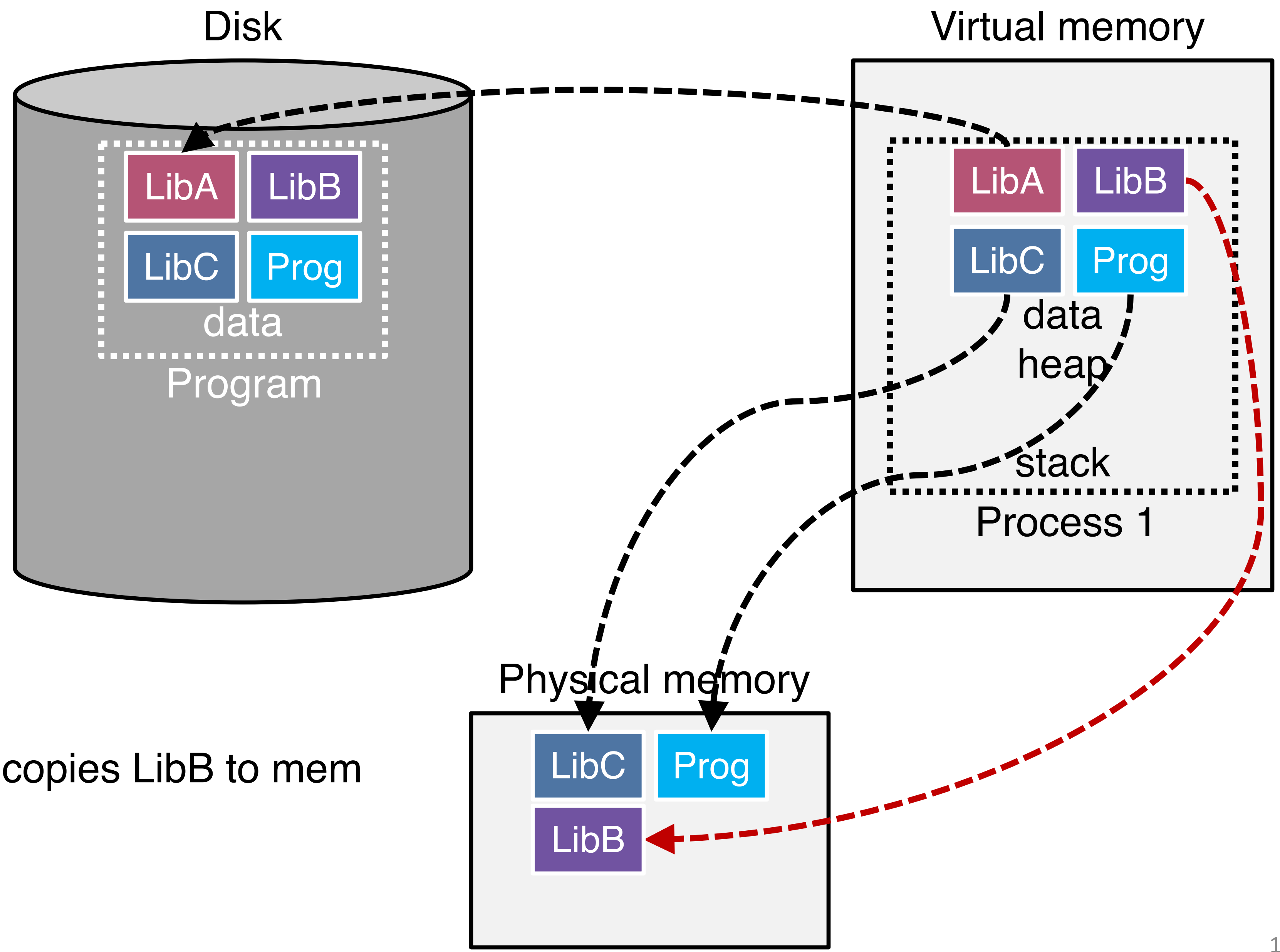
How to avoid wasting physical pages to back rarely used virtual pages?


Disk
LibA
LibB
LibC
Prog
data
Program
Virtual memory
LibA
LibB
LibC
Prog
data
heap
stack
Process 1
Physical memory
LibC
Prog

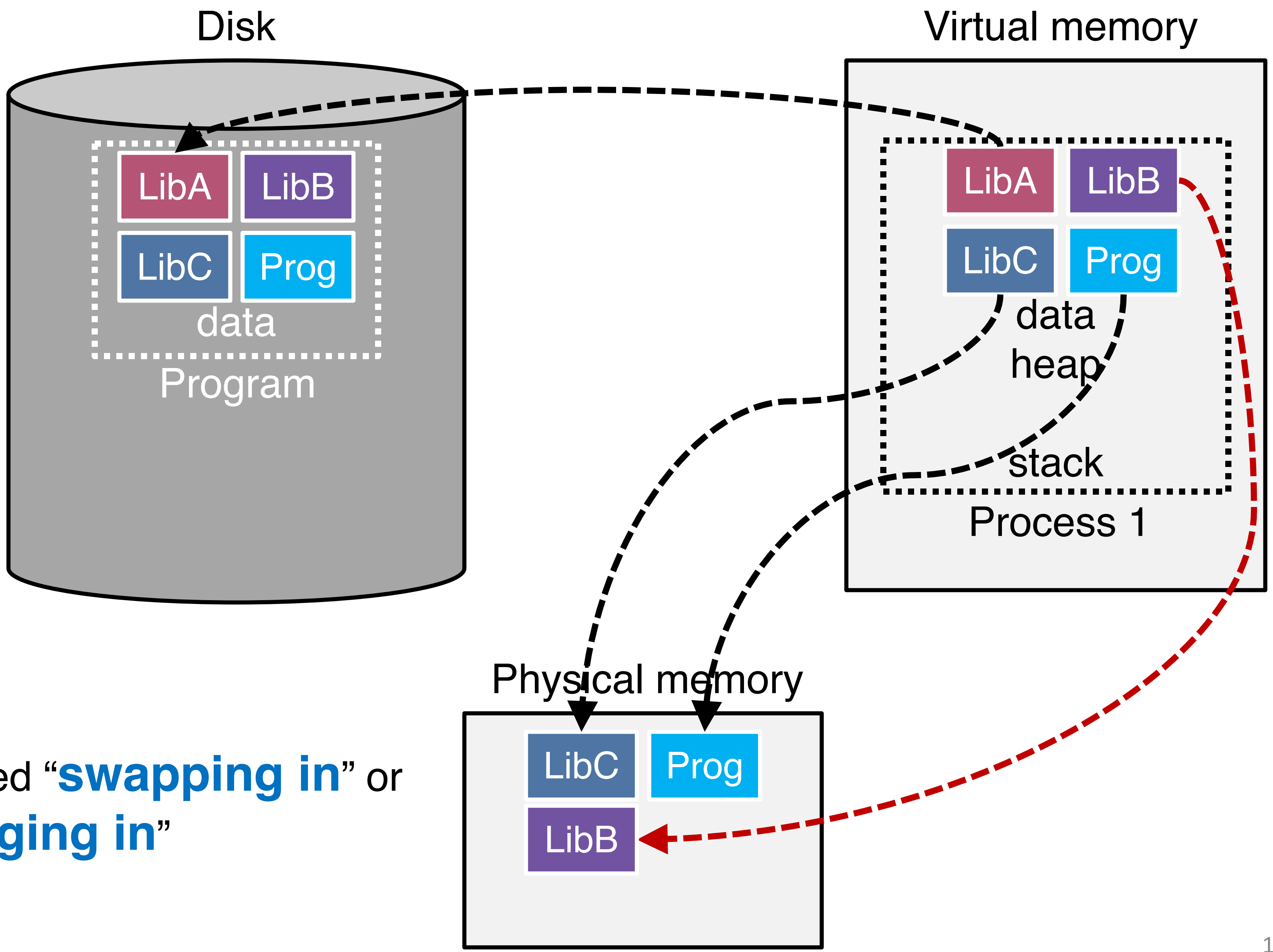

Disk
Virtual memory
LibA
LibB
LibC
Prog
data
Program
LibA
LibB
LibC
Prog
data
heap
stack
Process 1
Physical memory
LibC
Prog



Process 1 accesses LibB



OS copies LibB to mem



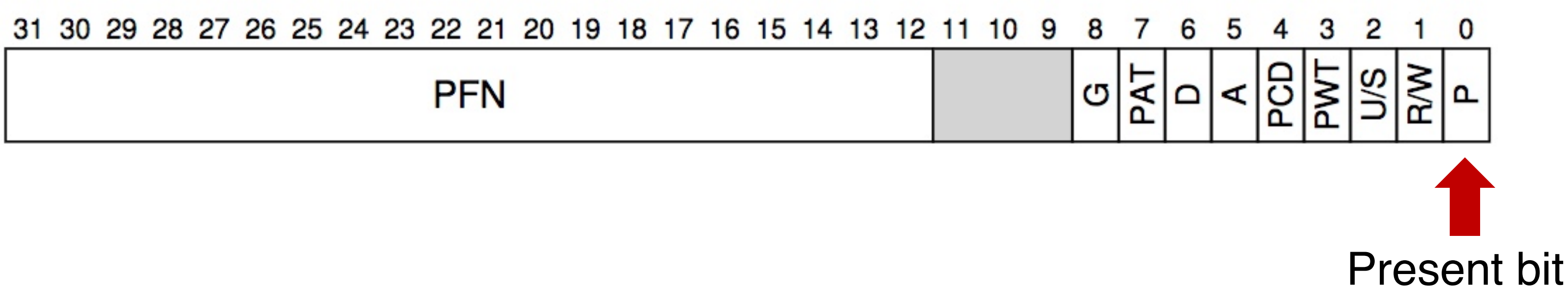
called “**swapping in**” or “**paging in**”

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)



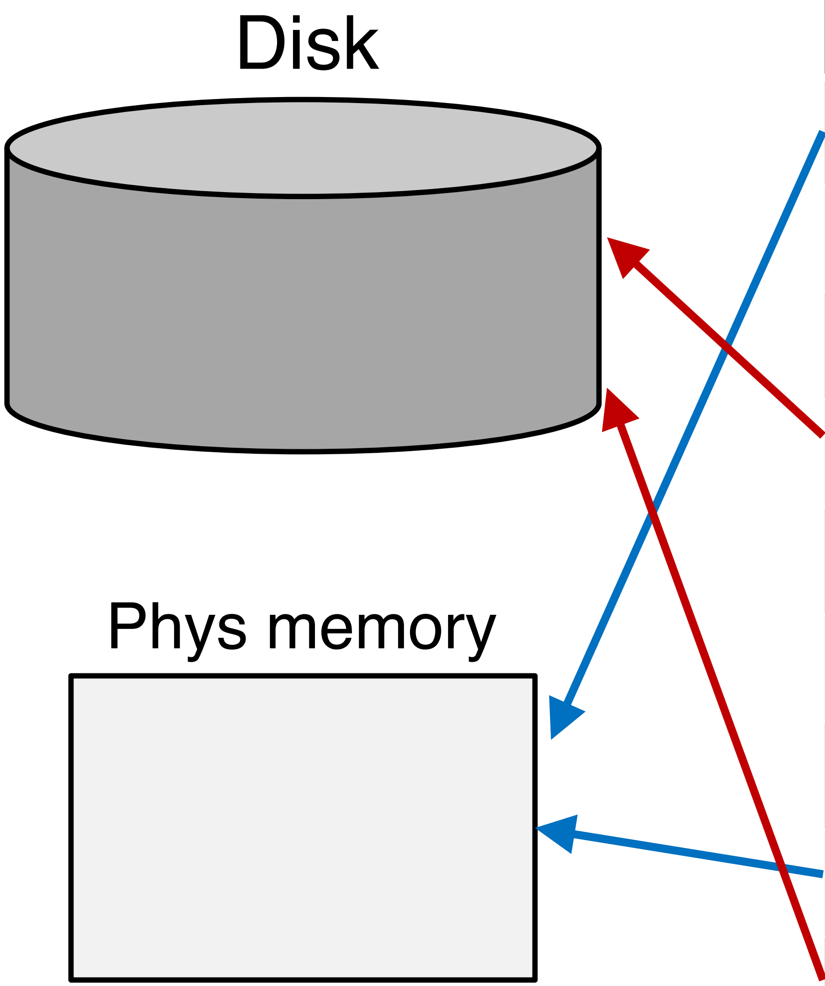
- During address translation, if present bit in PTE is 0 ➔ page fault

Present Bit

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

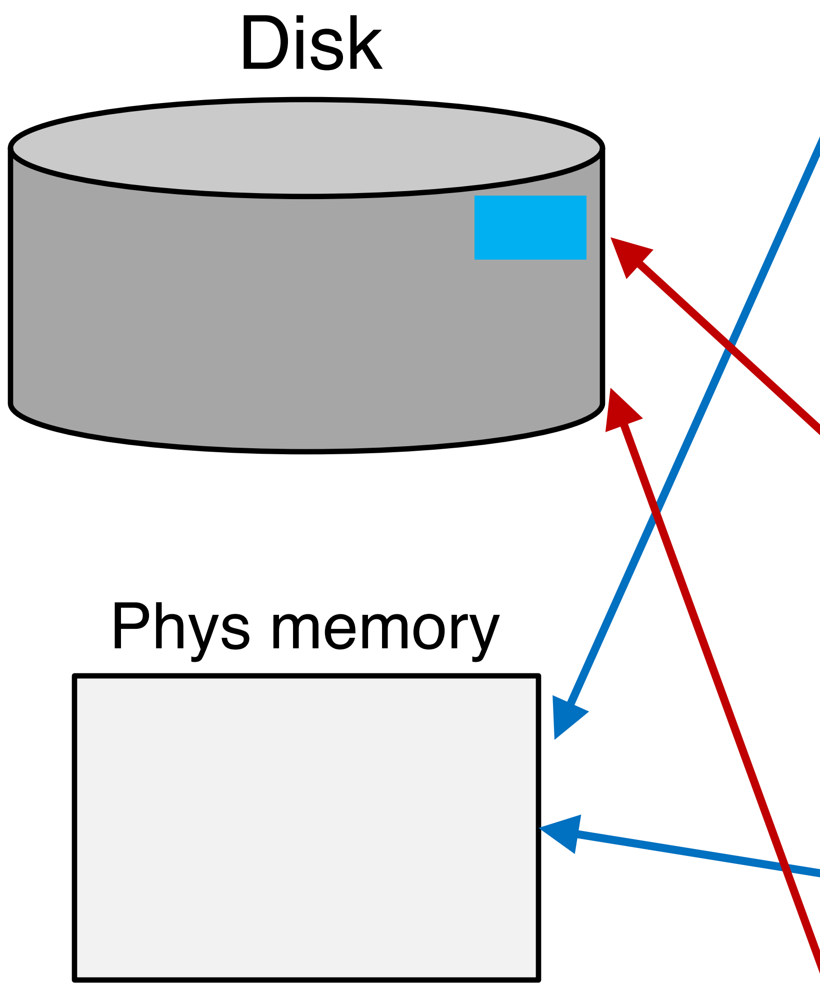
Page table

Present Bit



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

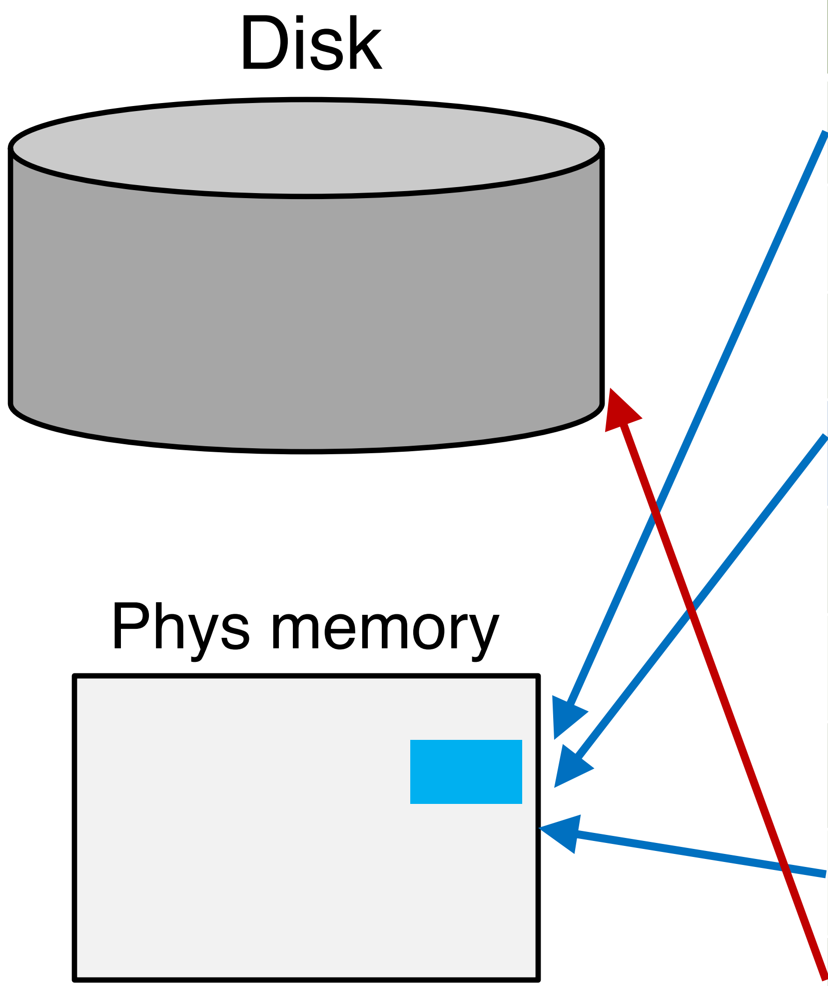
Present Bit



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

access

Present Bit

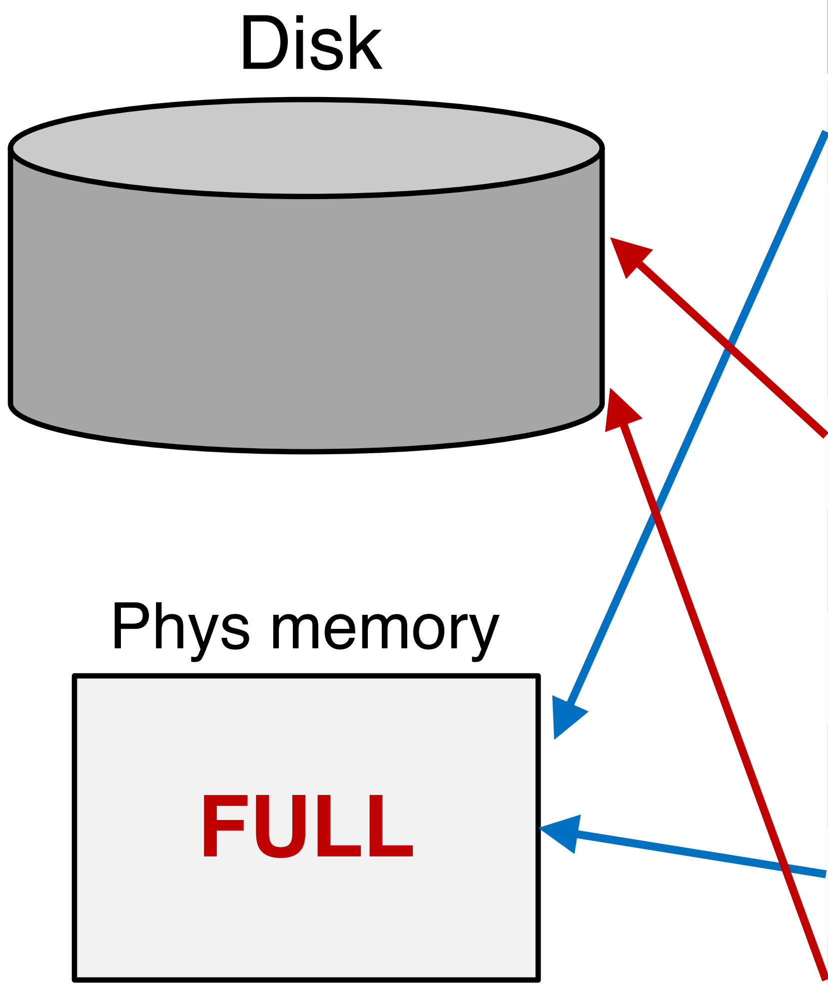


PFN	valid	prot	present
5	1	r-x	1
-	0	-	-
-	0	-	-
8	1	rw-	1
-	0	0	-
-	0	0	-
-	0	0	-
4	1	rw-	1
64	1	rw-	0

access

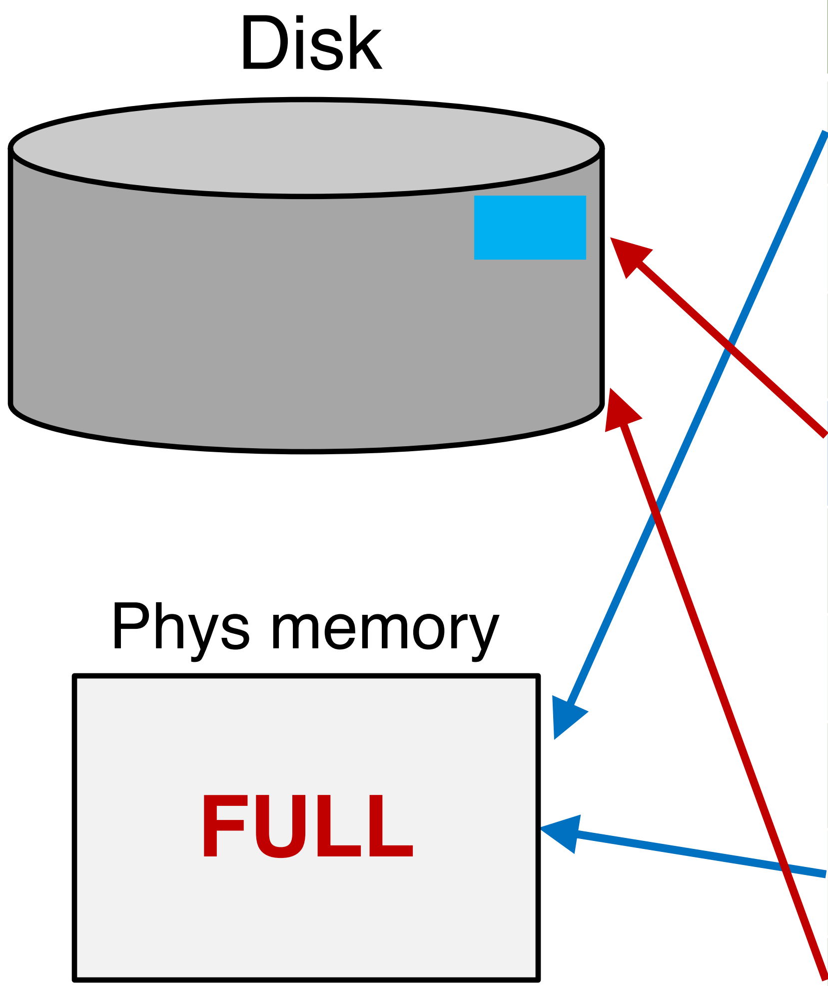
What if NO Memory is Left?

Present Bit



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

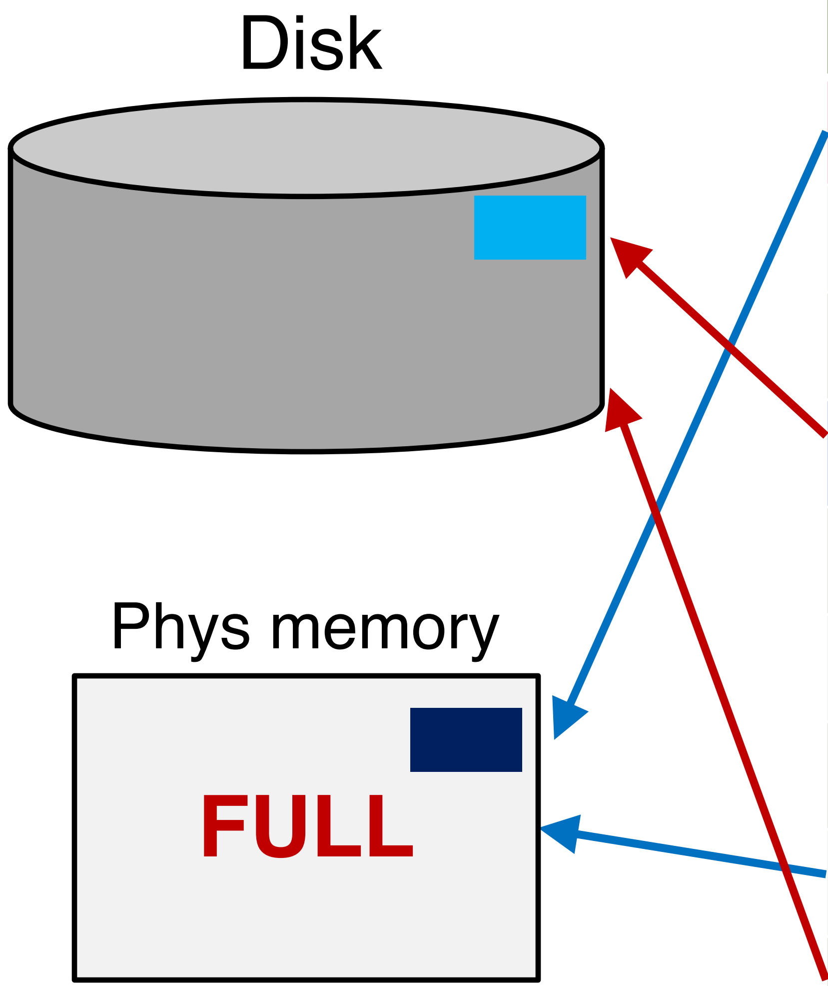
Present Bit



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

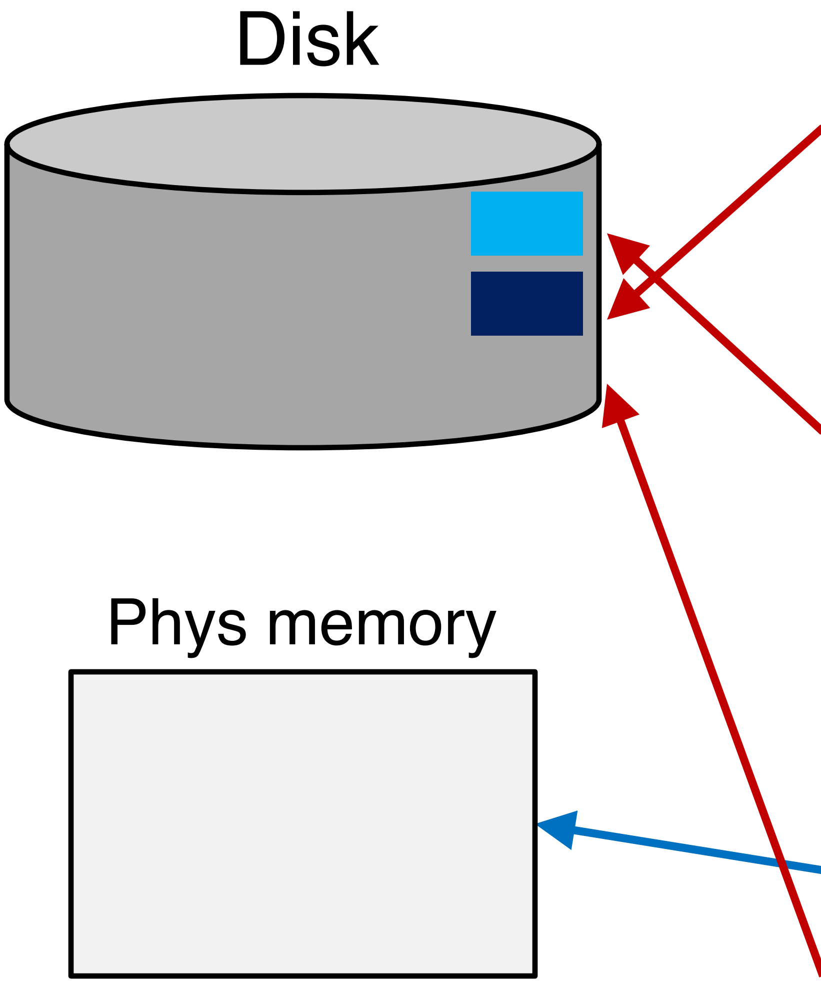
access

Present Bit



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

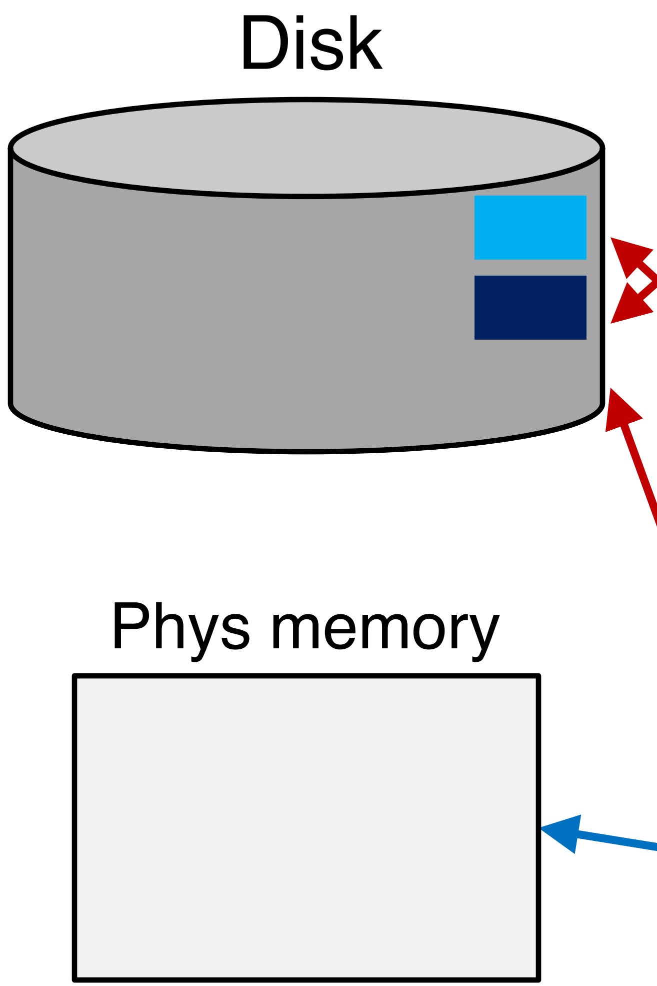
Present Bit



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

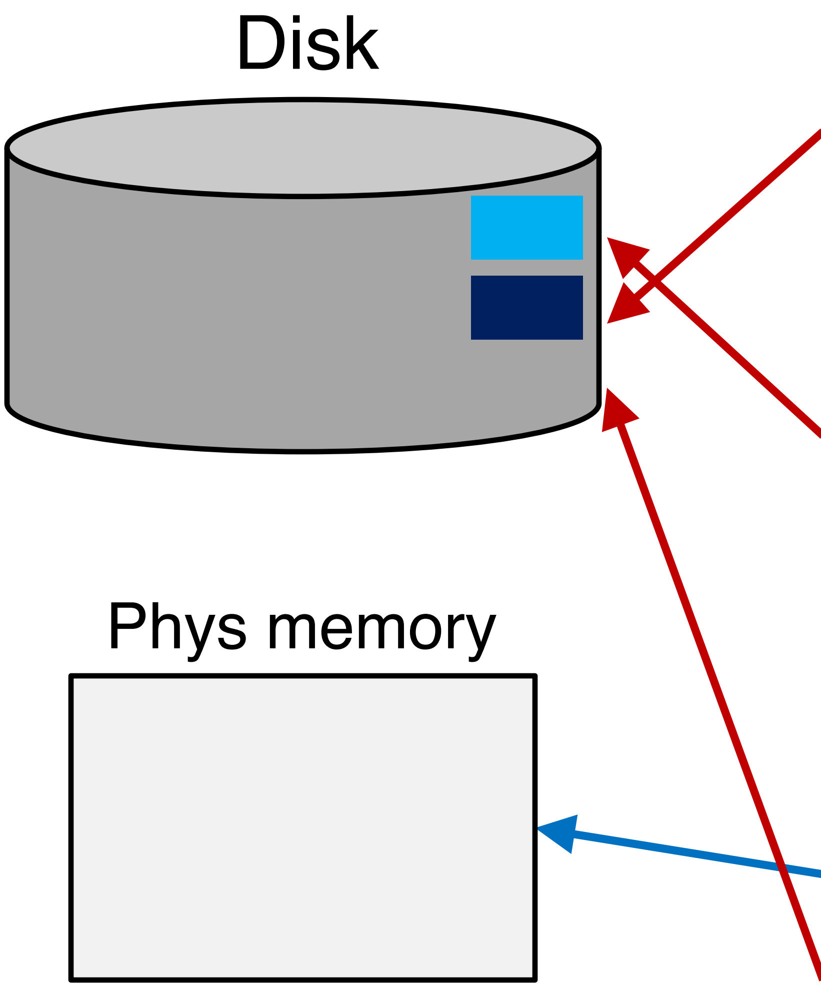
called "**swapping out**"
or "**paging out**"

Present Bit



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

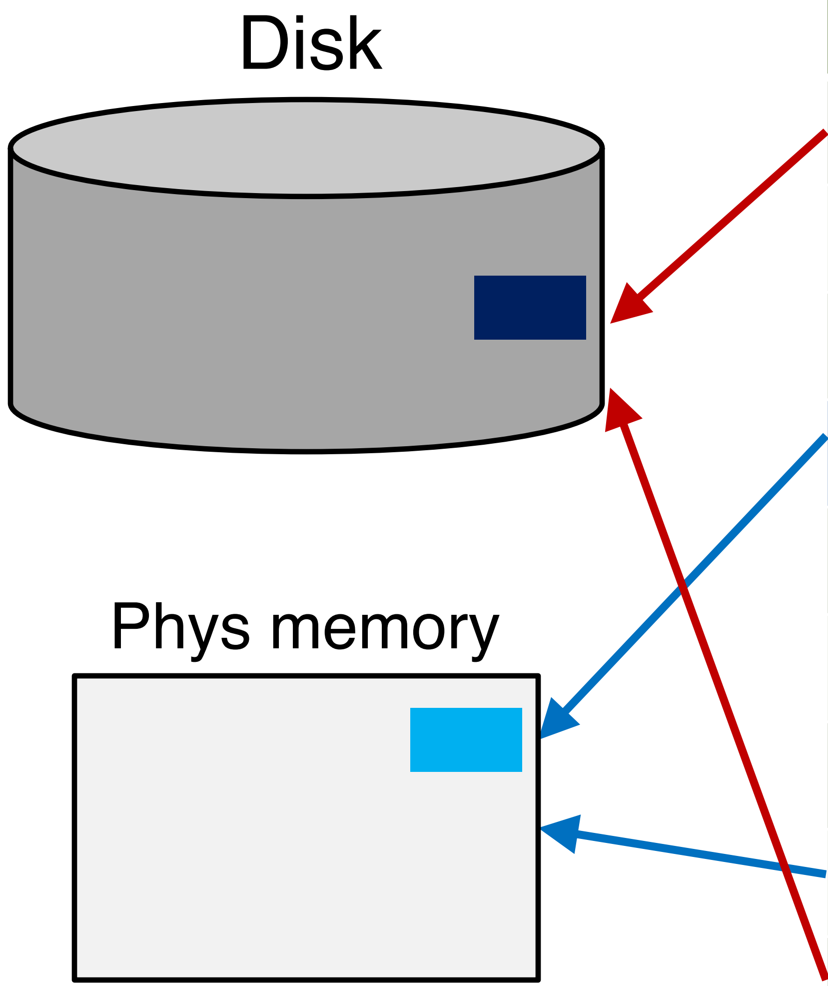
Present Bit



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

access

Present Bit



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

access

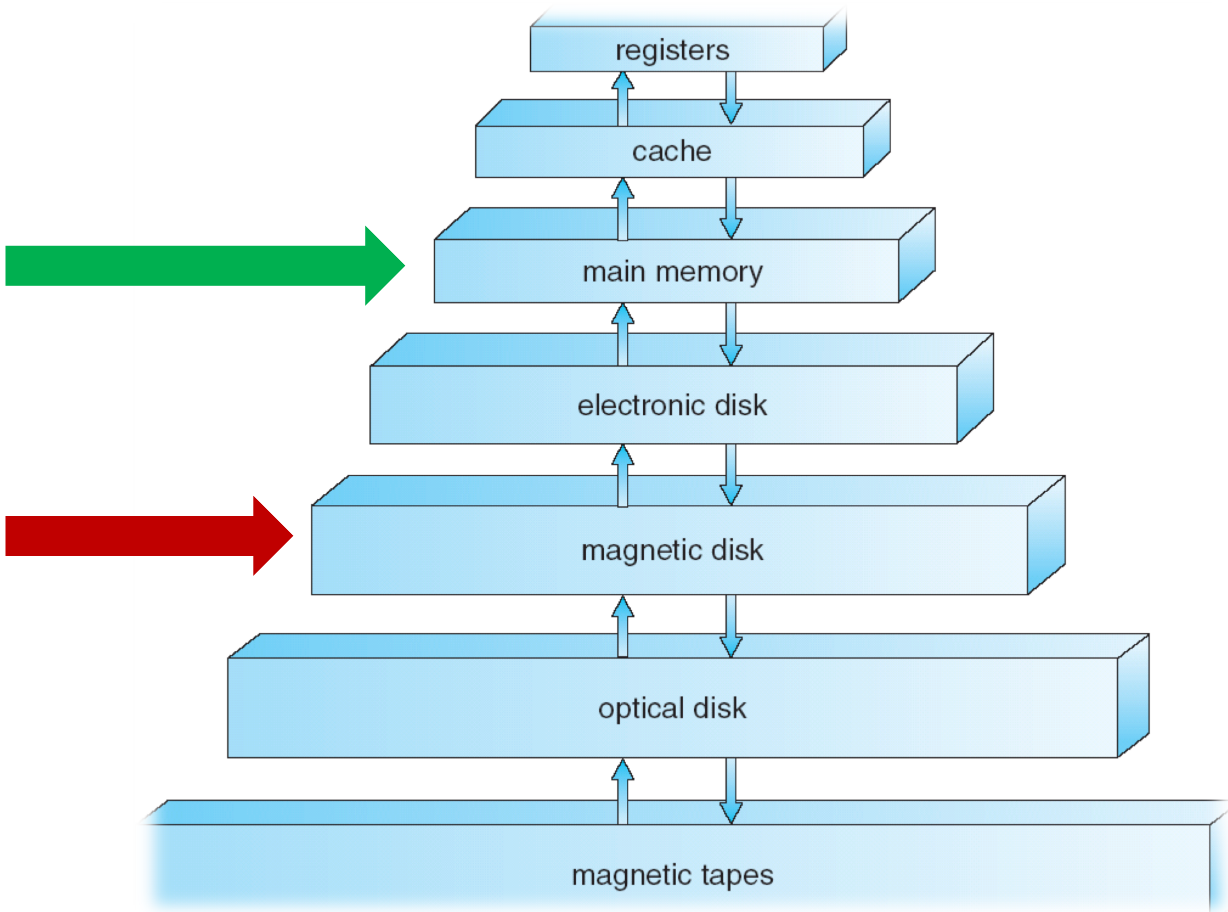
again, another "**swapping in**" or "**paging in**"

Why not Leave Page on Disk?

Storage Hierarchy

Main memory:
Smaller capacity
Faster accesses

Secondary storage:
Larger capacity
Way slower accesses



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
  - Disk: ~10ms

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

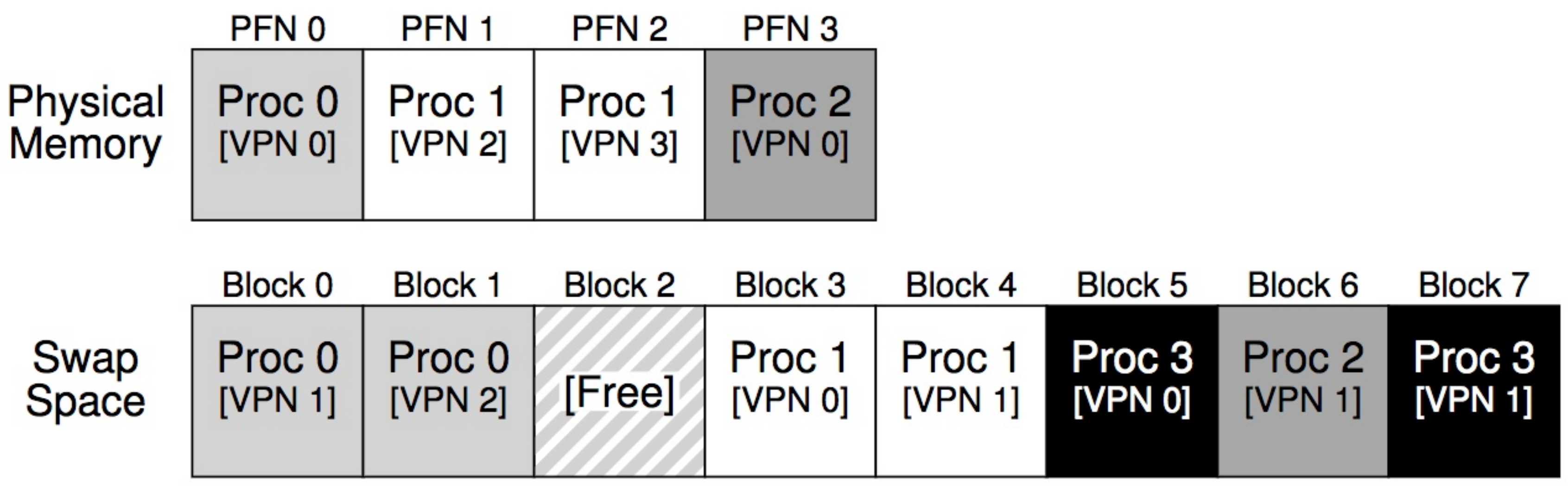
Swapping

- 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

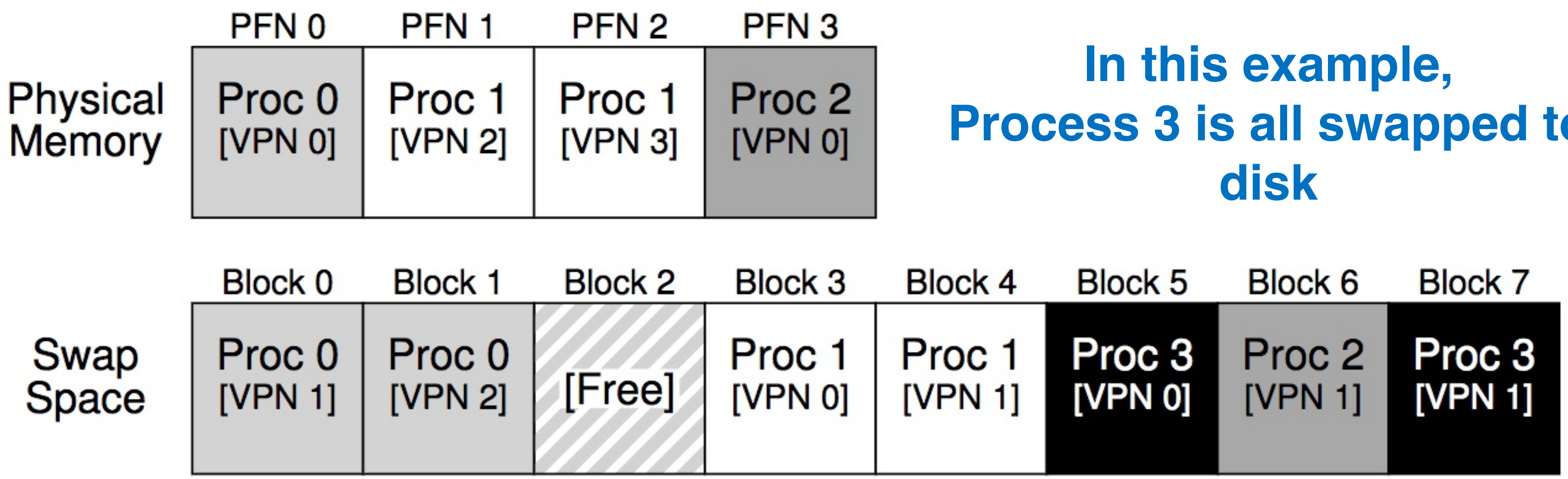
Swapping

- 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



Swap Space

- Part of disk space reserved for moving pages back and forth
  - Swap pages out of memory
  - Swap pages into memory from disk
- OS reads from and writes to the swap space at page-sized unit



In this example, Process 3 is all swapped to disk

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
- Typically, **PTE** contains the page address on disk

Page-Fault Handler (OS)

```
PFN = FindFreePage()
if (PFN == -1)
        PFN = EvictPage()
DiskRead(PTE.DiskAddr, PFN)
PTE.present = 1
PTE.PFN = PFN
retry instruction
```

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?

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?

Page-Fault Handler (OS)

(cheap) PFN = FindFreePage()

(cheap) if (PFN == -1)

(depends) PFN = EvictPage()

(expensive) DiskRead(PTE.DiskAddr, PFN)

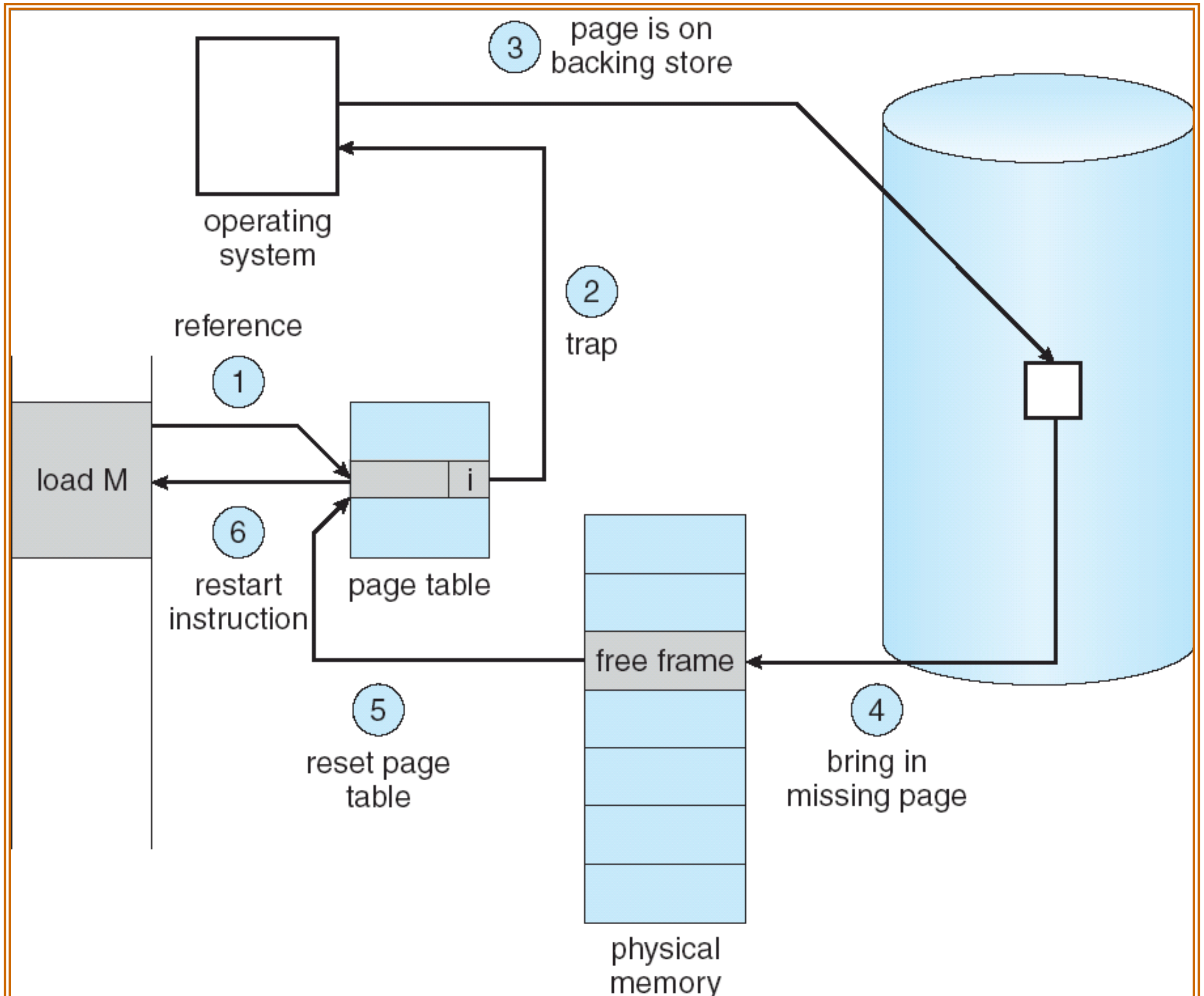
What to evict?
What to read?

(cheap) PTE.present = 1

(cheap) PTE.PFN = PFN

(cheap) retry instruction

Major Steps of A Page Fault



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

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

An Example

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

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 0 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 0 1 2 1

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

assume cache size 3

FIFO Replacement Policy

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

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	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				

assume cache size 3

FIFO Replacement Policy

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

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	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

assume
cache size 3

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

- 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

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

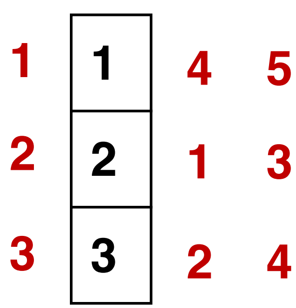
Access	Hit	State (after)
1	no	1
2	no	1,2
3	no	1,2,3
4	no	1,2,3,4
1	yes	1,2,3,4
2	yes	1,2,3,4
5	no	2,3,4,5
1	no	3,4,5,1
2	no	4,5,1,2
3	no	5,1,2,3
4	no	1,2,3,4
5	no	2,3,4,5

Belady's Anomaly

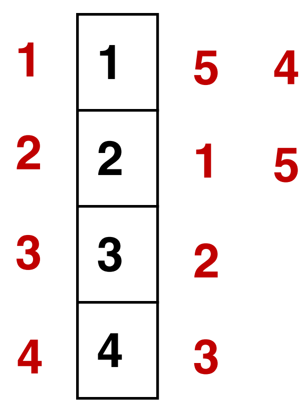
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  - Size-3 (3-frames) case results in 9 page faults
  - Size-4 (4-frames) case results in 10 page faults

- Program runs potentially slower w/ more memory!

- Belady's anomaly
  - More frames ➔ more page faults for some access pattern



9 page faults



10 page faults

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 0 1 2 1

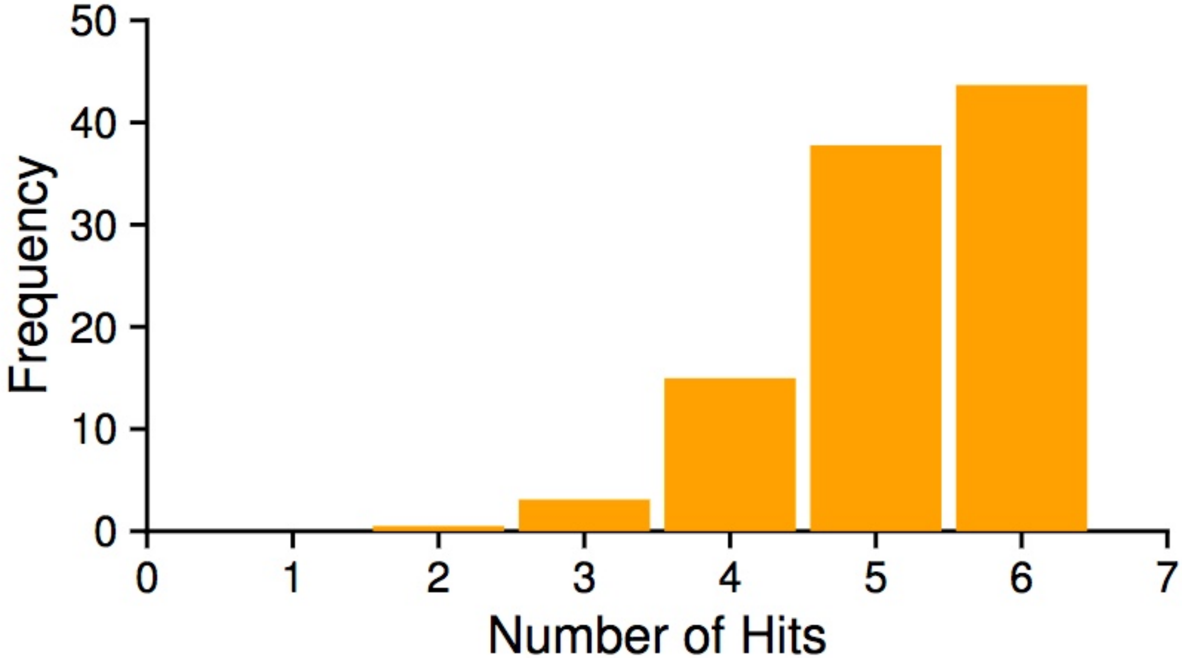
Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	0	1, 2, 3
0	Miss	1	2, 3, 0
3	Hit		2, 3, 0
1	Miss	3	2, 0, 1
2	Hit		2, 0, 1
1	Hit		2, 0, 1

assume cache size 3

How Random Policy Performs?

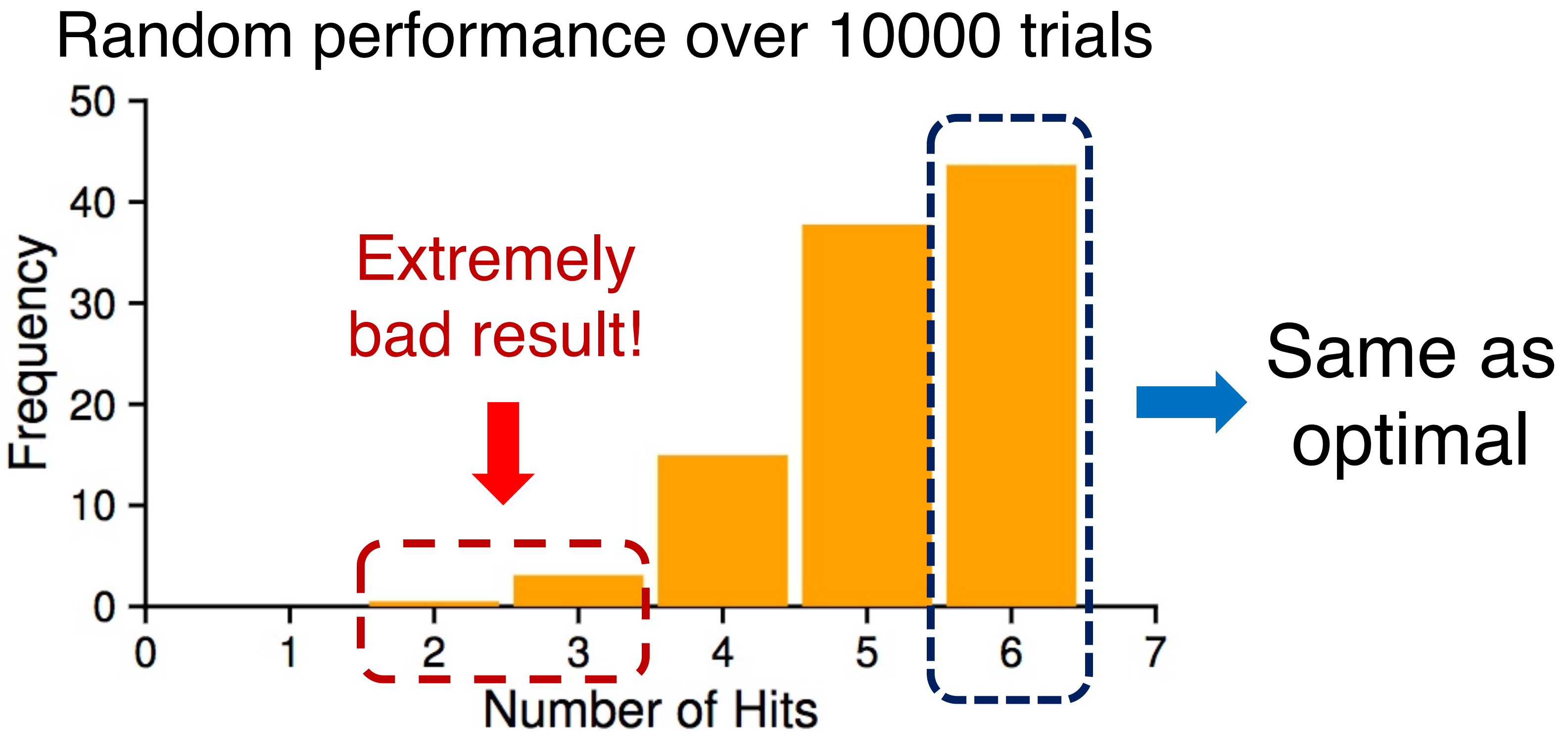
- Depends entirely on **how lucky you are**
- Example workload: 0 1 2 0 1 3 0 3 0 1 2 1

Random performance over 10000 trials



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



Belady’s Optimal

OPT: The Optimal Replacement Policy

- Many years ago **Belady** demonstrated that there is a simple policy (OPT or MIN) 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 OPT in practice!

- But it is extremely useful as a practical best-case baseline for comparison purpose

Proof of Optimality for Belady's Optimal Replacement Policy

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

Erasing Belady's Limitations

https://www.usenix.org/conference/atc16/technical-sessions/presentation/cheng

Erasing Belady's Limitations: In Search of Flash Cache Offline Optimality

Yue Cheng, *Virginia Polytechnic Institute and State University;* Fred Douglis, Philip Shilane, Michael Trachtman, and Grant Wallace, *EMC Corporation;* Peter Desnoyers, *Northeastern University;* Kai Li, *Princeton University*

https://www.usenix.org/conference/atc16/technical-sessions/presentation/cheng

OPT the Optimal

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

OPT the Optimal

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

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

assume cache size 3

OPT the Optimal

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

OPT the Optimal

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

OPT the Optimal

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

OPT the Optimal

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

What to evict??

OPT the Optimal

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

OPT the Optimal

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

OPT the Optimal

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

What to evict??

OPT the Optimal

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

assume cache size 3

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!

What to evict??

OPT the Optimal

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

OPT the Optimal

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

OPT the Optimal

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

The optimal number of cache hits is **6** for this workload!

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 0 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 0 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				
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 0 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 0 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 0 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 0 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 0 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 0 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)

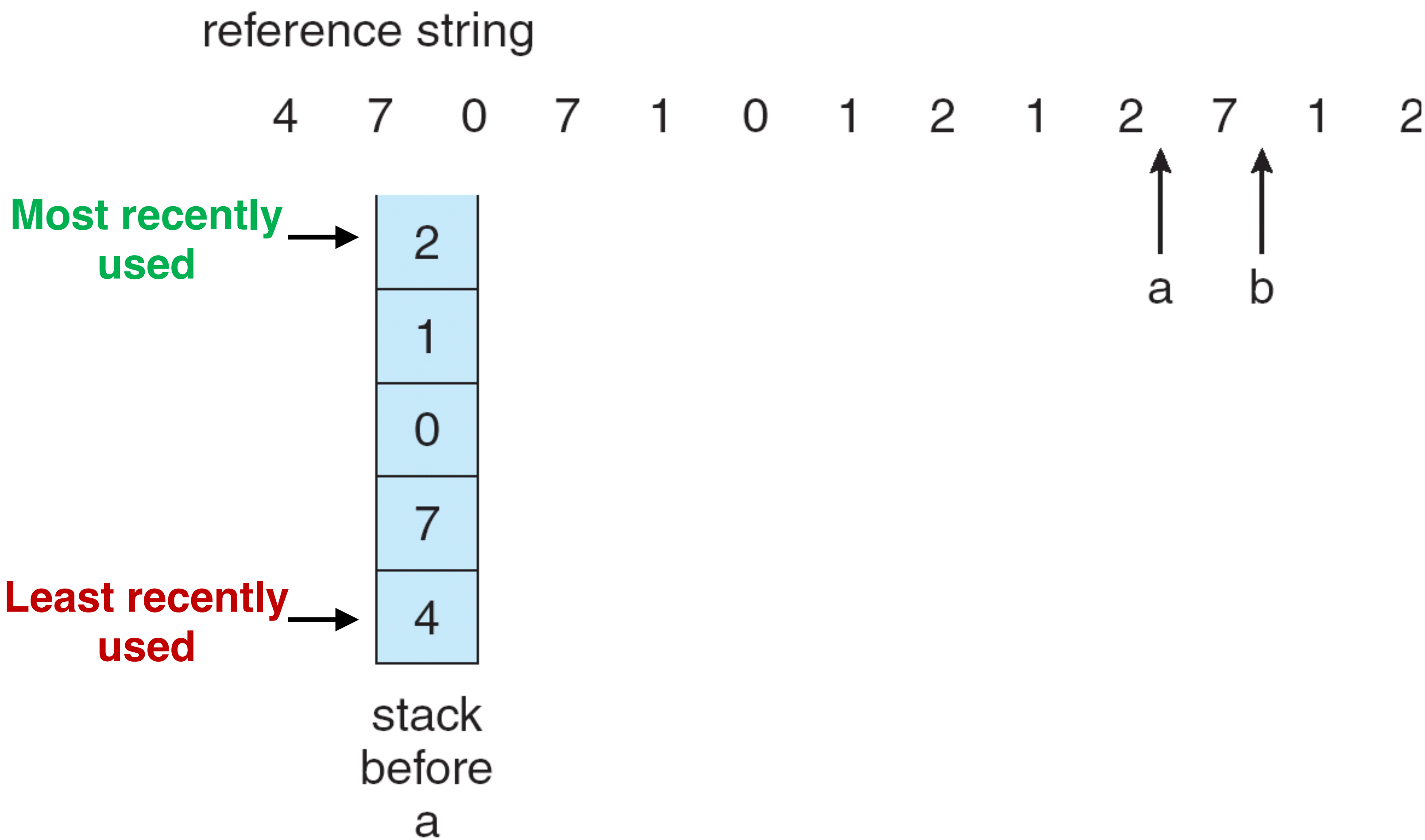
- Idea: evict the page that has not been used for the longest period of time
- Example workload: 0 1 2 0 1 3 0 3 0 1 2 1

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

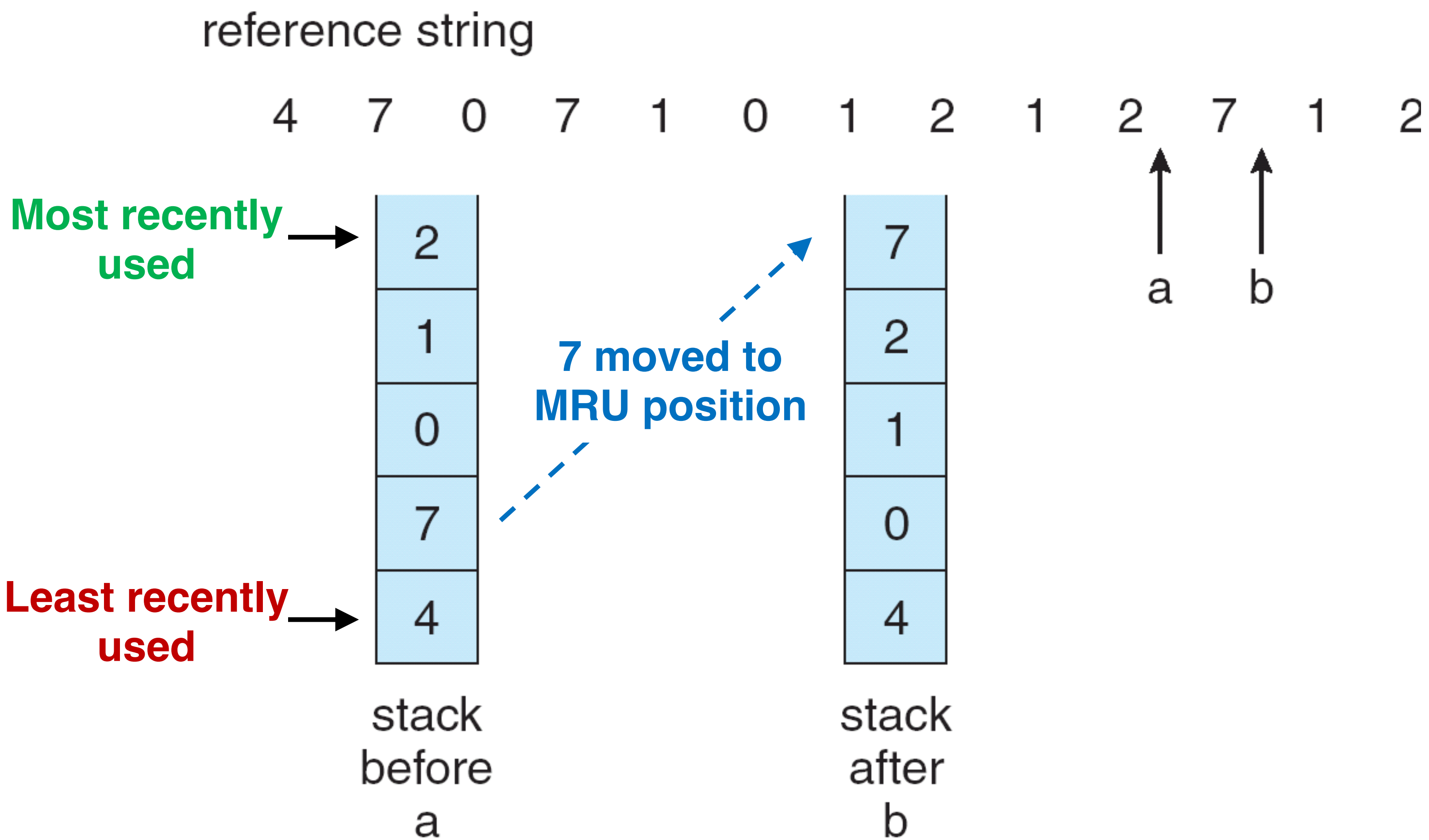
LRU Stack Implementation

- 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



LRU Hardware Support

- Sophisticated hardware support may involve high overhead/cost!

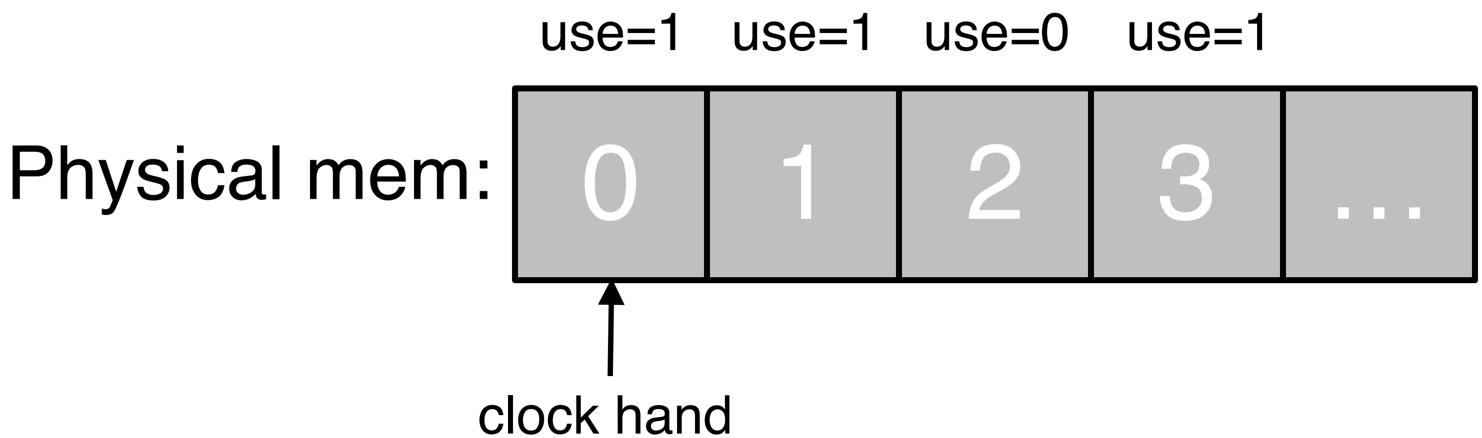
- Some limited HW support is common:

  Reference (or use) bit

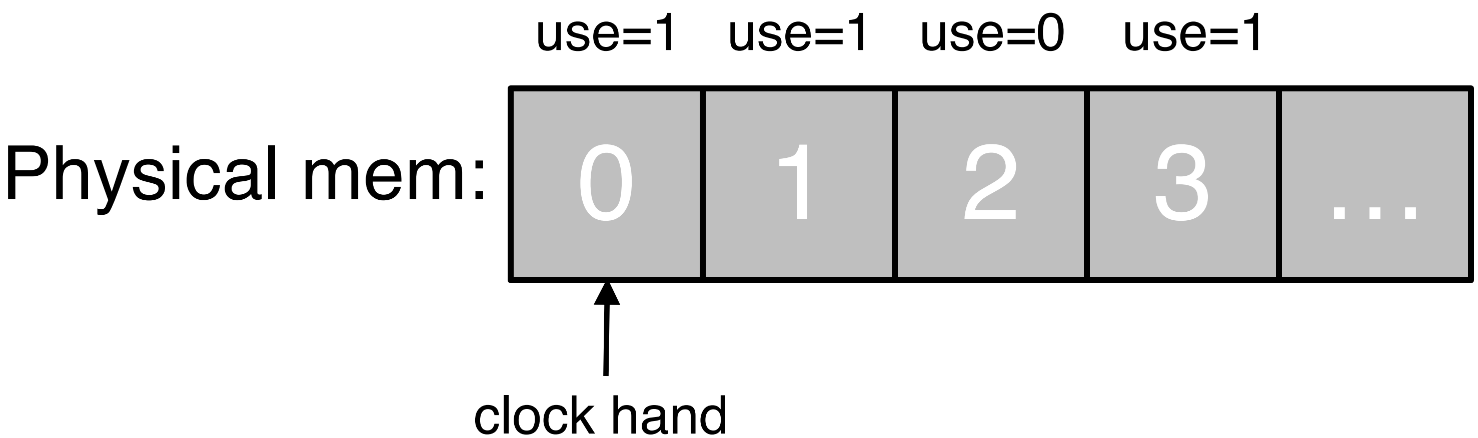
  - With each page associate a bit, initially set to 0
  - When the page is referenced, bit set to 1
  - By examining the reference bits, we can determine which pages have been used
  - **We do not know the *order* of use, however!**

- Cheap approximation
  - Useful for **clock** algorithm

Clock: Look For a Page

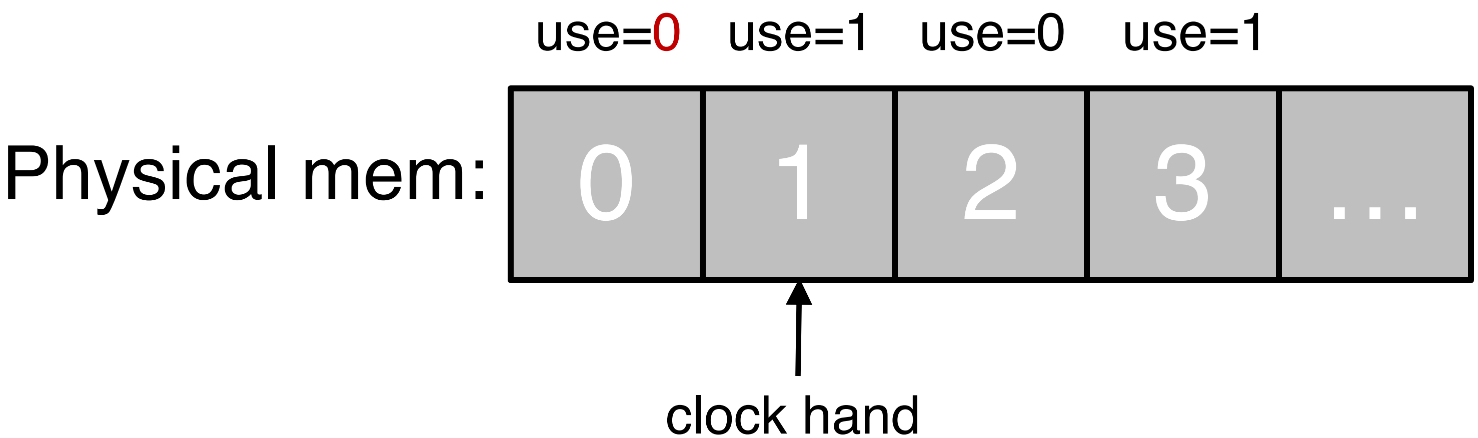


Clock: Look For a Page



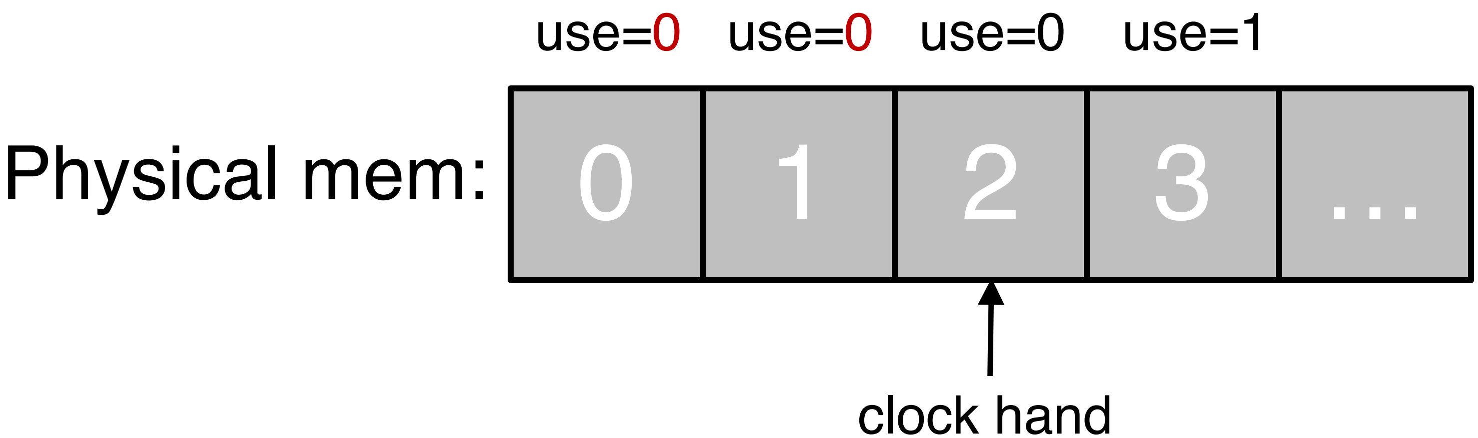
Mem is full, and to evict a page to make room

Clock: Look For a Page



Mem is full, and to evict a page to make room

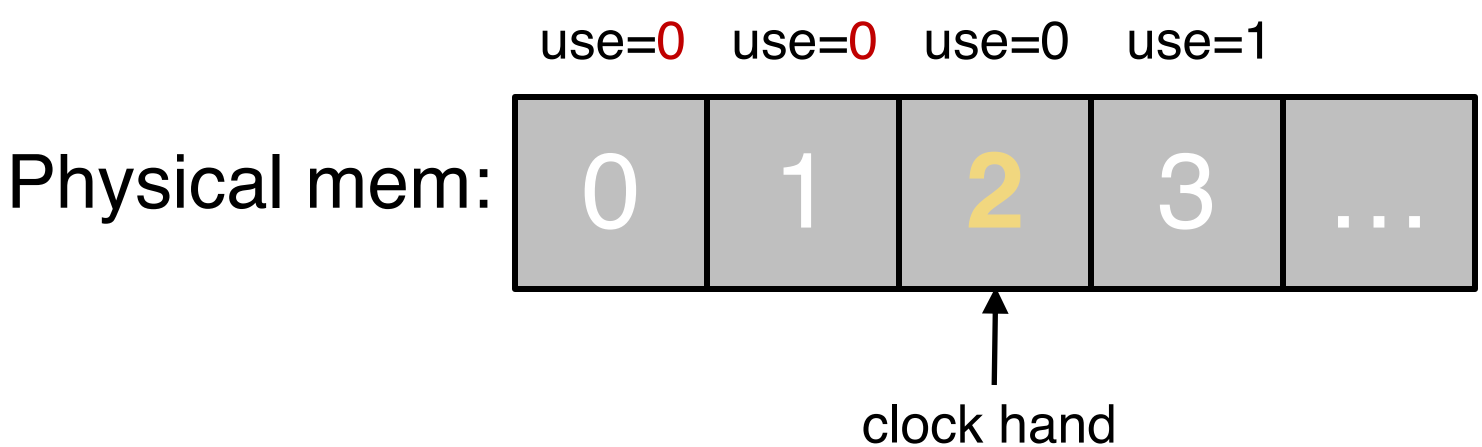
Clock: Look For a Page



Mem is full, and to evict a page to make room

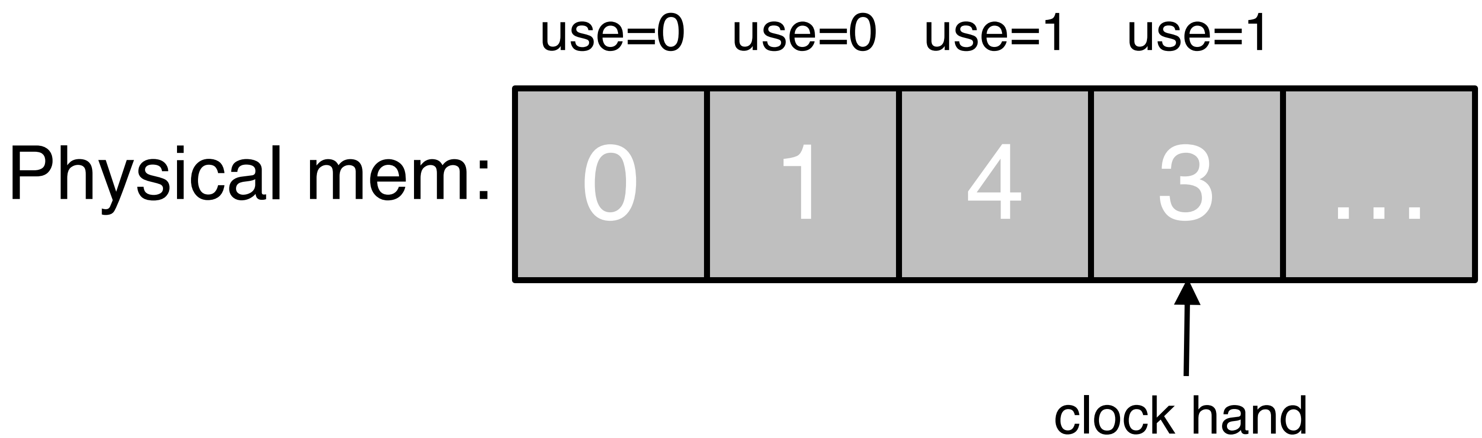
Clock: Look For a Page

Evict **page 2** because it has not been recently used



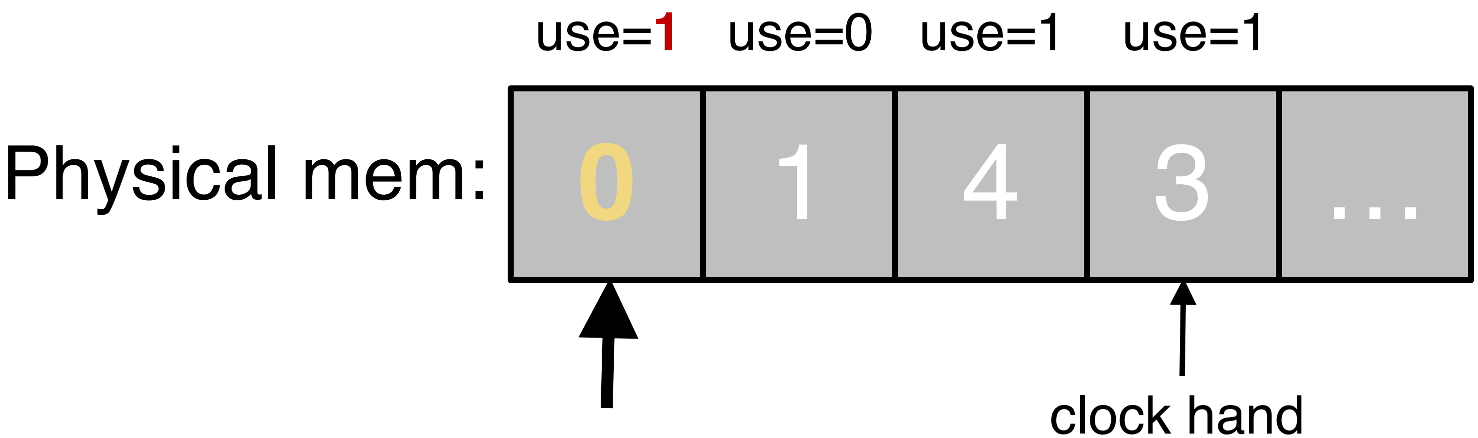
Mem is full, and to evict a page to make room

Clock: Look For a Page

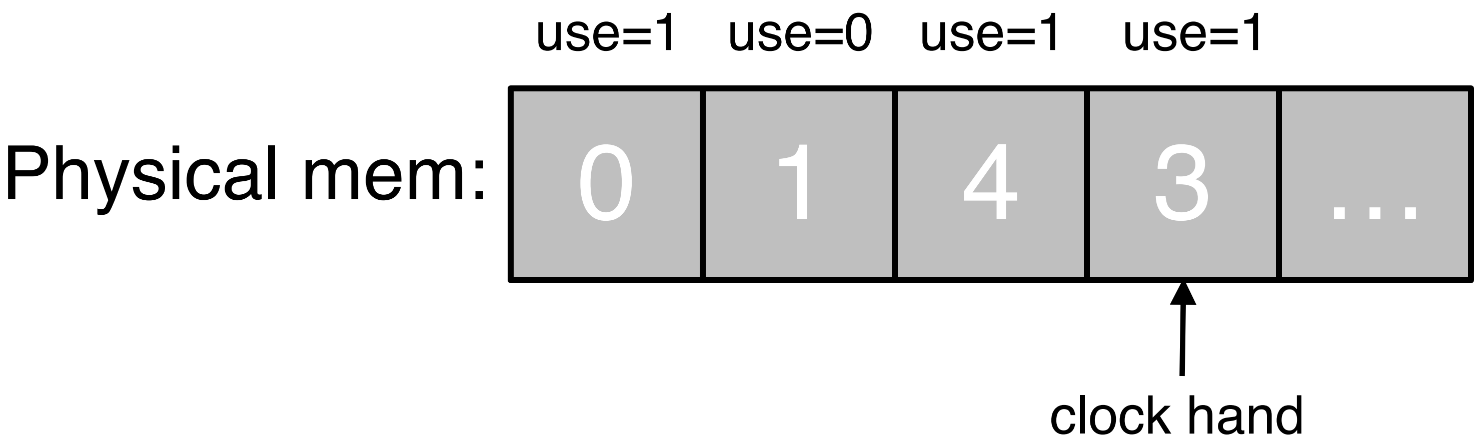


Clock: Access a Page

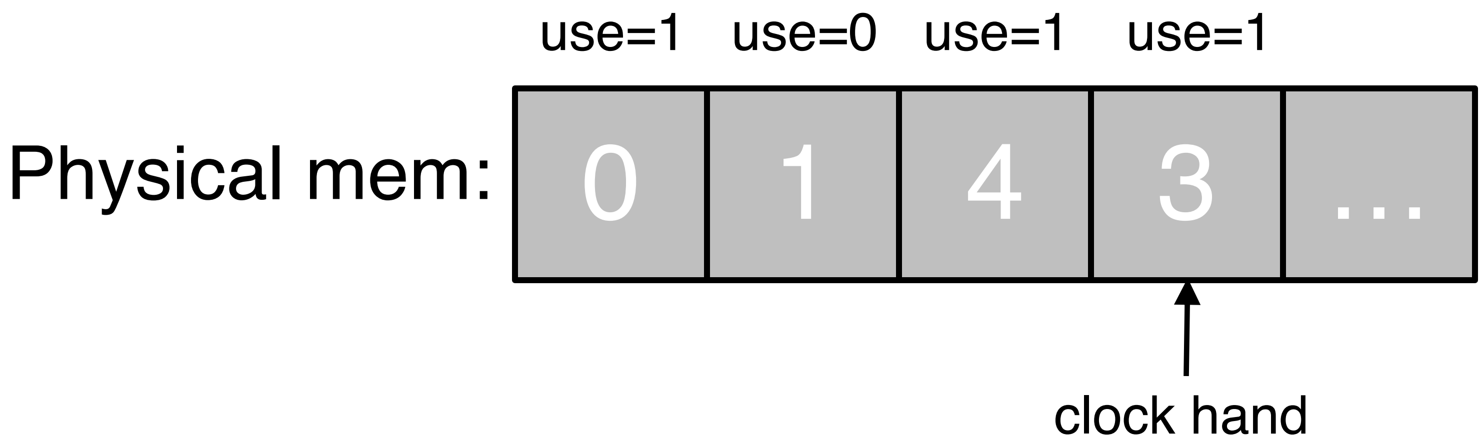
page 0 is accessed



Clock: Look For a Page

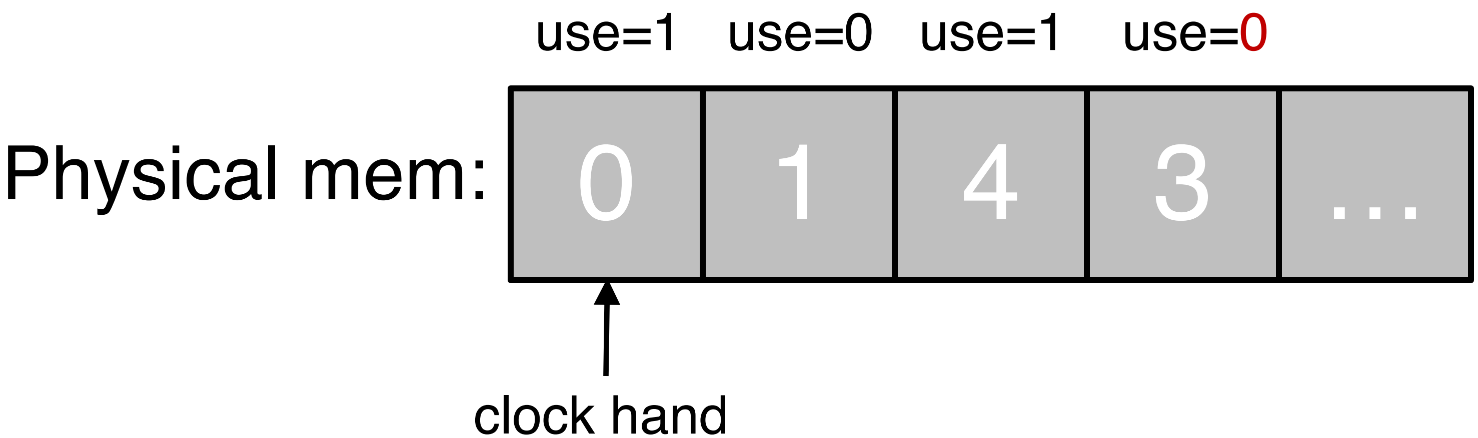


Clock: Look For a Page



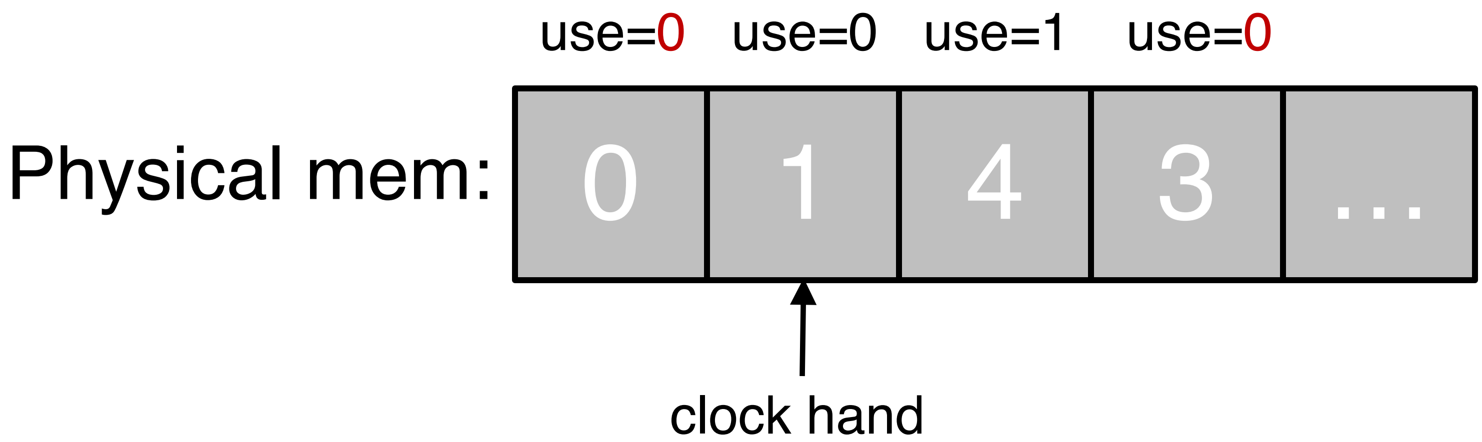
Mem is full, and to evict a page to make room

Clock: Look For a Page



Mem is full, and to evict a page to make room

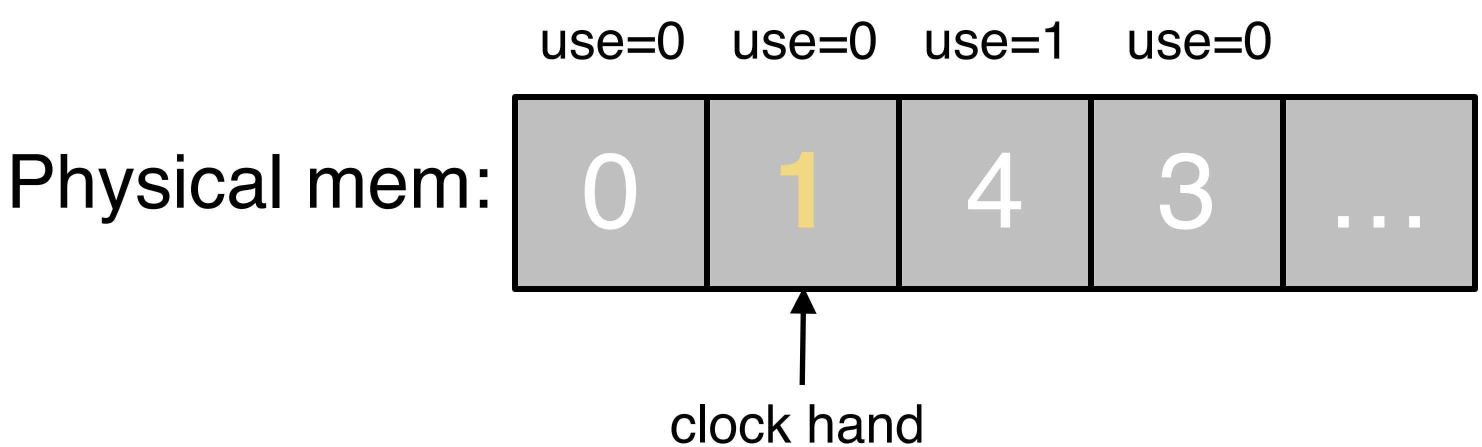
Clock: Look For a Page



Mem is full, and to evict a page to make room

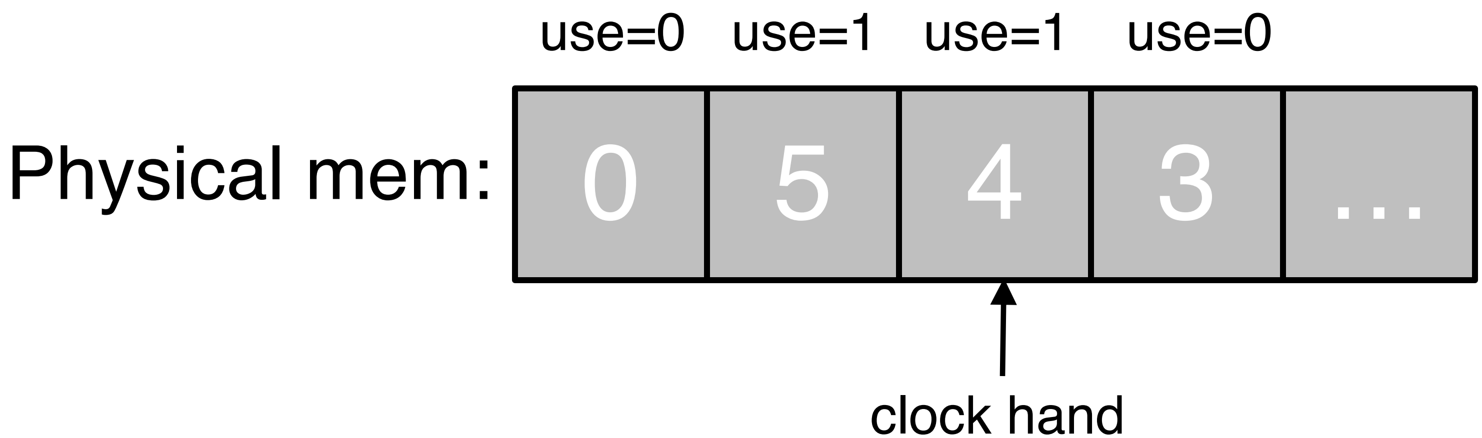
Clock: Look For a Page

Evict **page 1** because it has not been recently used



Mem is full, and to evict a page to make room

Clock: Look For a Page



Summary: Page Replacement Policies

- FIFO
  - Why it might work? Maybe the one brought in the longest ago is one we are not using now
  - Why it might not work? No real info to tell if it’s being used or not
  - Suffers “**Belady’s Anomaly**”

Summary: Page Replacement Policies

- FIFO
  - Why it might work? Maybe the one brought in the longest ago is one we are not using now
  - Why it might not work? No real info to tell if it's being used or not
  - Suffers "**Belady's Anomaly**"
- Random
  - Sometimes non intelligence is better

Summary: Page Replacement Policies

- FIFO
  - Why it might work? Maybe the one brought in the longest ago is one we are not using now
  - Why it might not work? No real info to tell if it's being used or not
  - Suffers "**Belady's Anomaly**"
- Random
  - Sometimes non intelligence is better
- OPT
  - Assume we know about the future
  - Not practical in real cases: **offline** policy
  - However, can be used as a **best case baseline** for comparison purpose

Summary: Page Replacement Policies

- FIFO
  - Why it might work? Maybe the one brought in the longest ago is one we are not using now
  - Why it might not work? No real info to tell if it's being used or not
  - Suffers "**Belady's Anomaly**"
- Random
  - Sometimes non intelligence is better
- OPT
  - Assume we know about the future
  - Not practical in real cases: **offline** policy
  - However, can be used as a **best case baseline** for comparison purpose
- LRU
  - Intuition: we can't look into the future, but let's look at past experience to make a good guess
  - Out "bet" is that pages used recently are ones which will be used again (**principle of locality**)