CS 471 Operating Systems

Yue Cheng

George Mason University
Fall 2019

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 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
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 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				
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 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				
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 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				
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 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 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 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

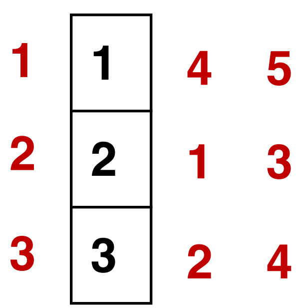
Access	Hit	State (after)
1	no	1
2	no	1,2
3	no	1,2,3
4	no	1,2,3,4
1	yes	1,2,3,4
2	yes	1,2,3,4
5	no	2,3,4,5
1	no	3,4,5,1
2	no	4,5,1,2
3	no	5,1,2,3
4	no	1,2,3,4
5	no	2,3,4,5

Belady’s Anomaly

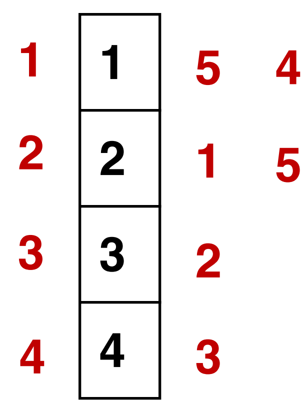
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  - Size-3 (3-frames) case results in 9 page faults
  - Size-4 (4-frames) case results in 10 page faults

- 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 1 2 1

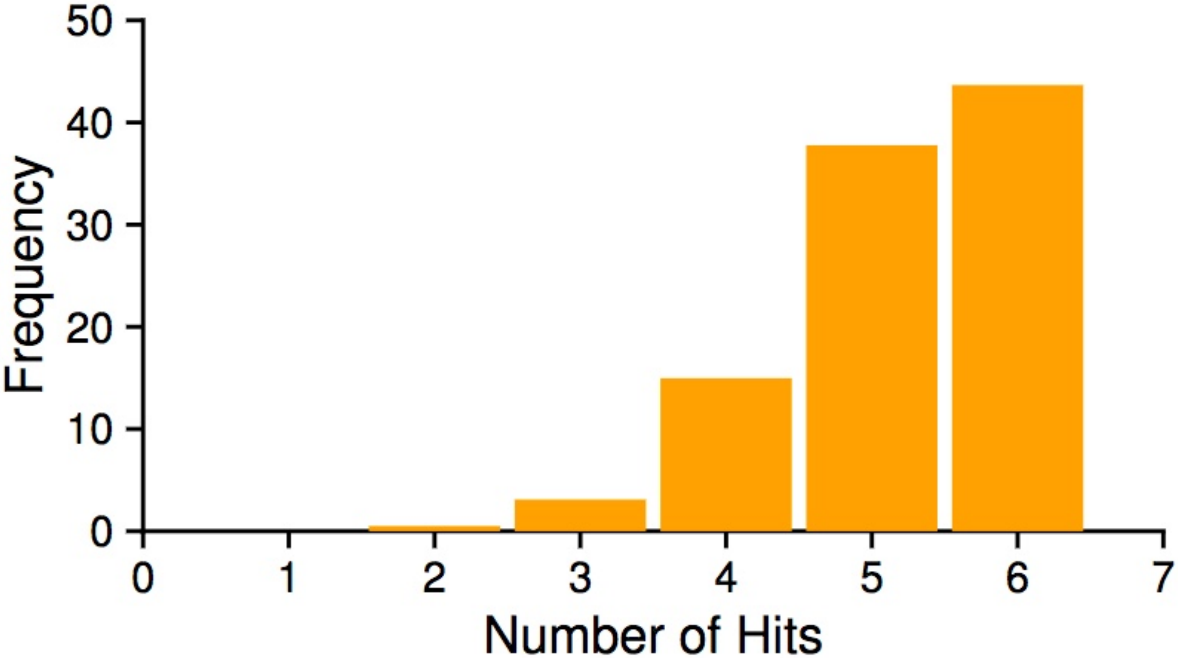
Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	0	1, 2, 3
0	Miss	1	2, 3, 0
3	Hit		2, 3, 0
1	Miss	3	2, 0, 1
2	Hit		2, 0, 1
1	Hit		2, 0, 1

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

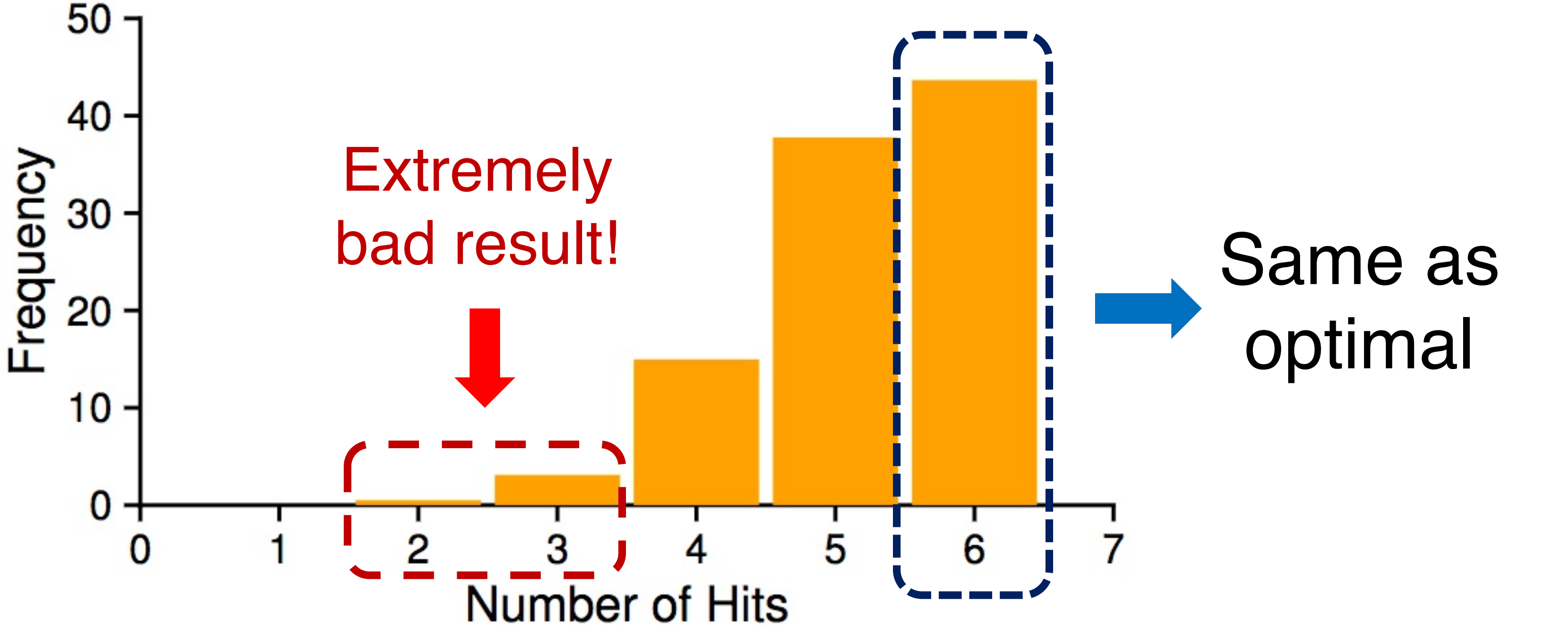
Random performance over 10000 trials



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

Random performance over 10000 trials



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

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 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 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 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 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 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 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 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 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 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 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 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 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 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 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	
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 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)

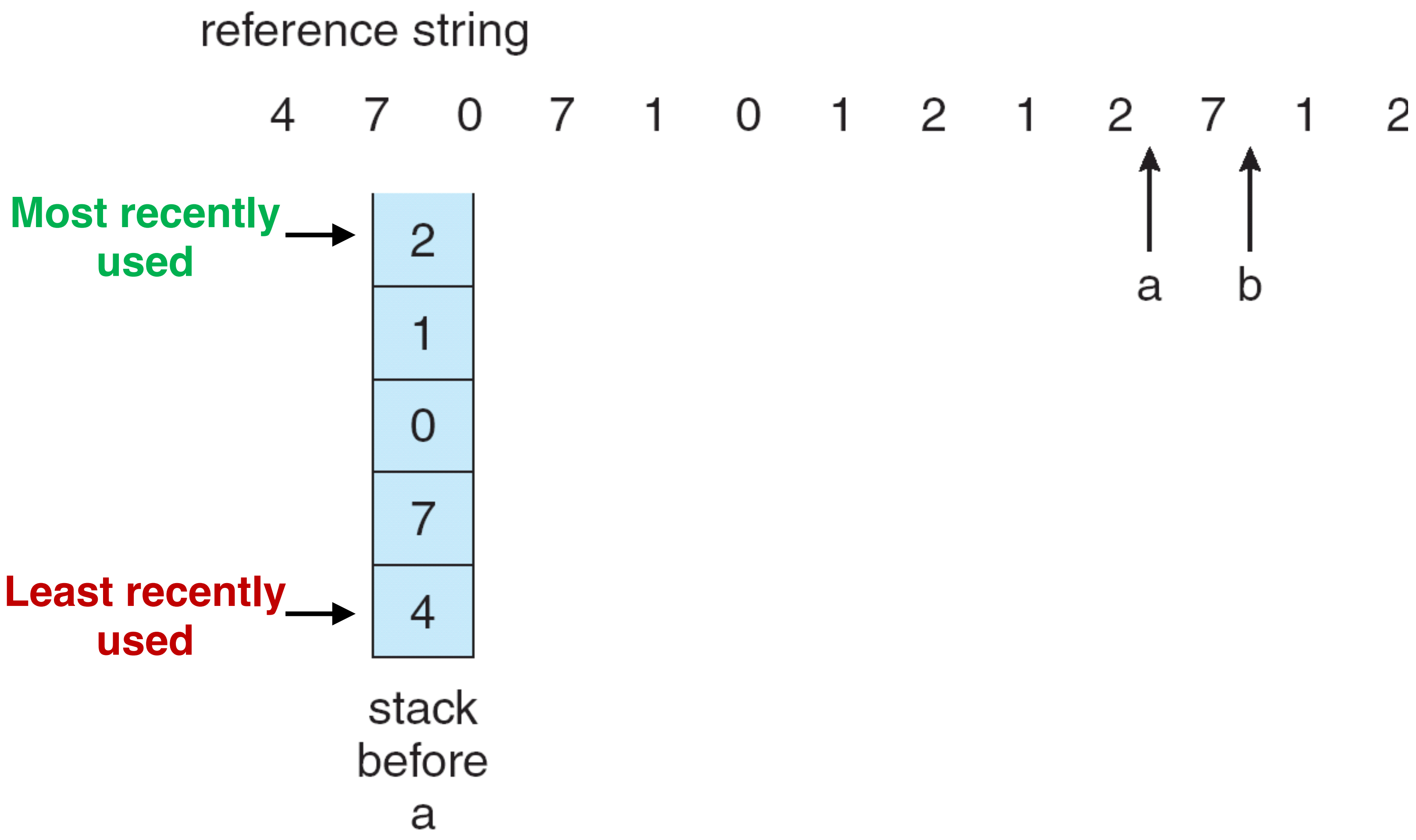
- Idea: evict the page that has not been used for the longest period of time
- Example workload: 0 1 2 0 1 3 0 3 1 2 1

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

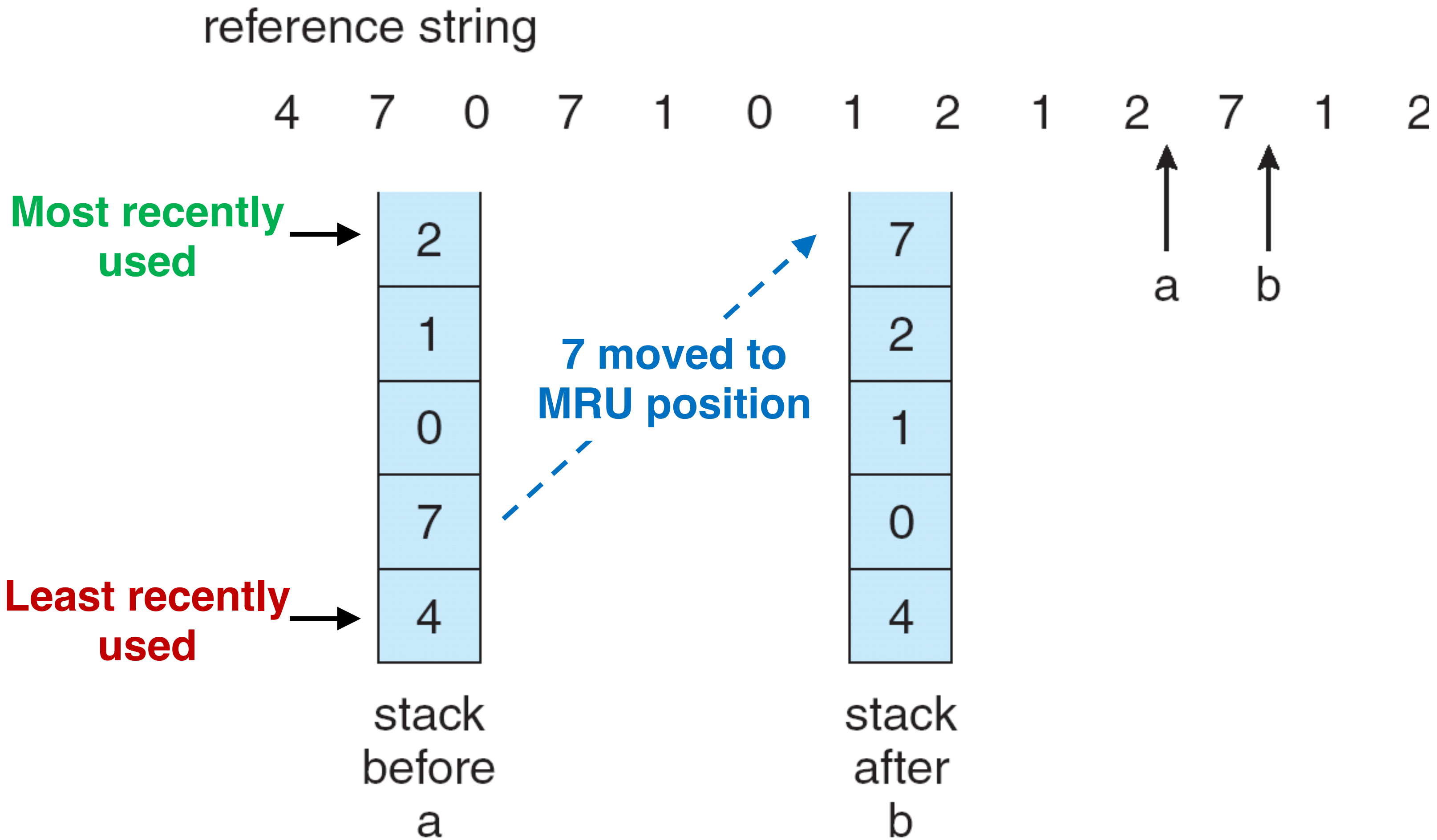
LRU Stack Implementation

- 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!

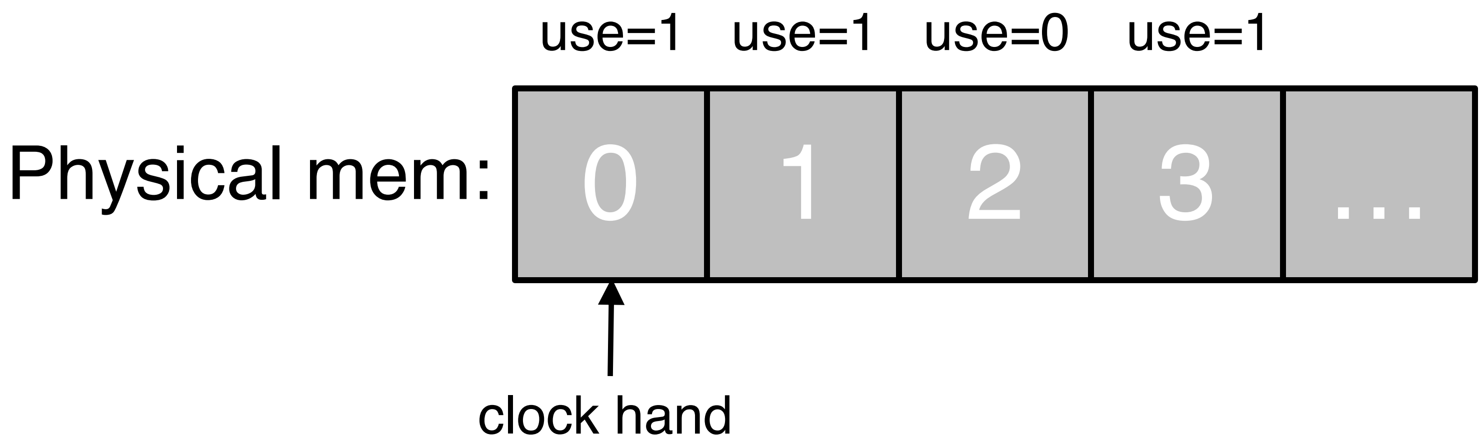
- Some limited HW support is common:

  Reference (or use) bit

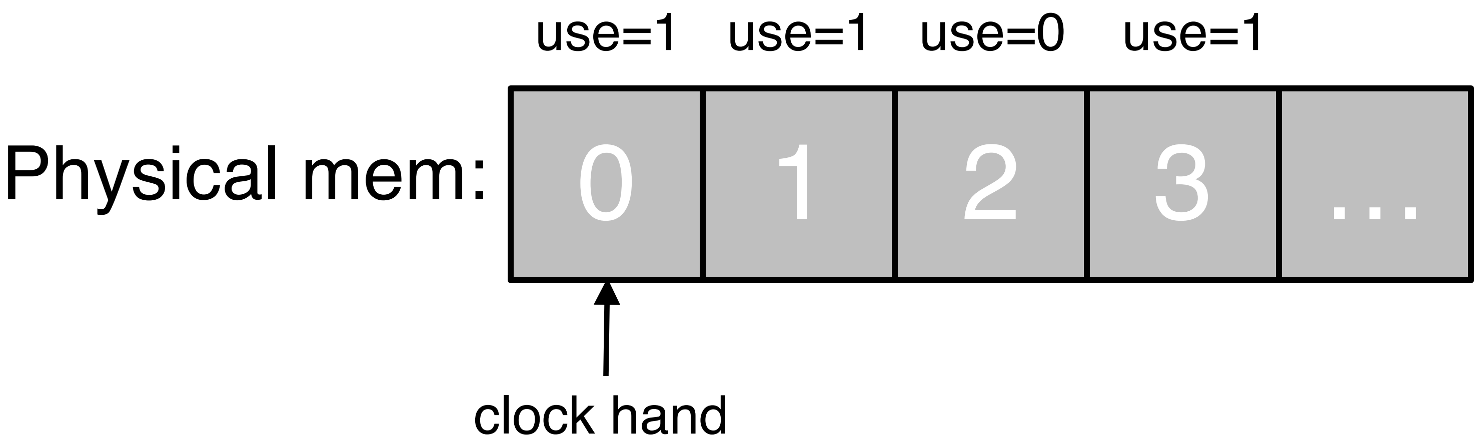
  - With each page associate a bit, initially set to 0
  - When the page is referenced, bit set to 1
  - By examining the reference bits, we can determine which pages have been used
  - **We do not know the *order* of use, however!**

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

Clock: Look For a Page

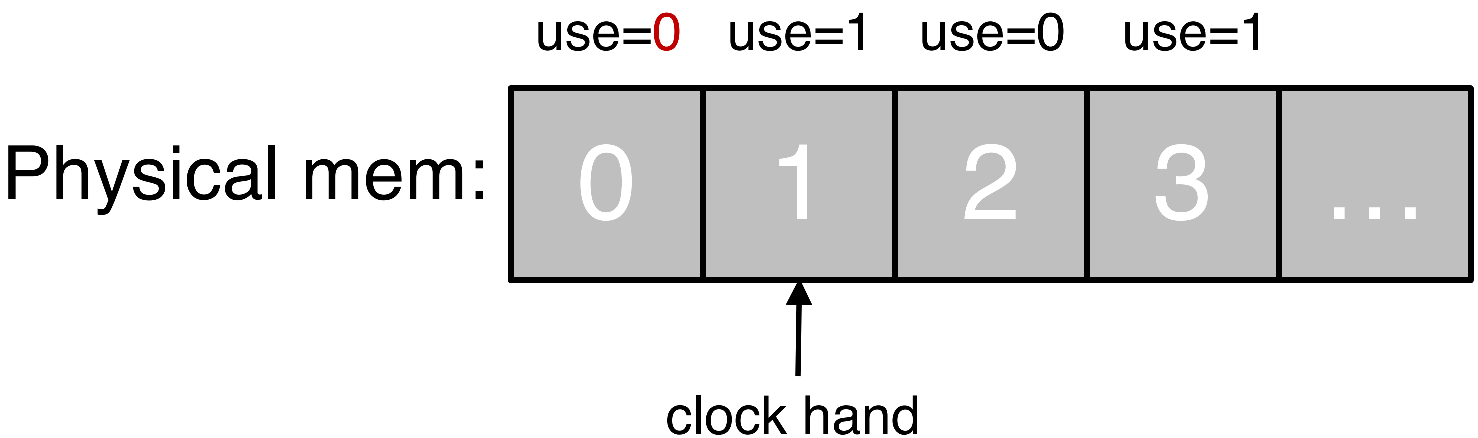


Clock: Look For a Page



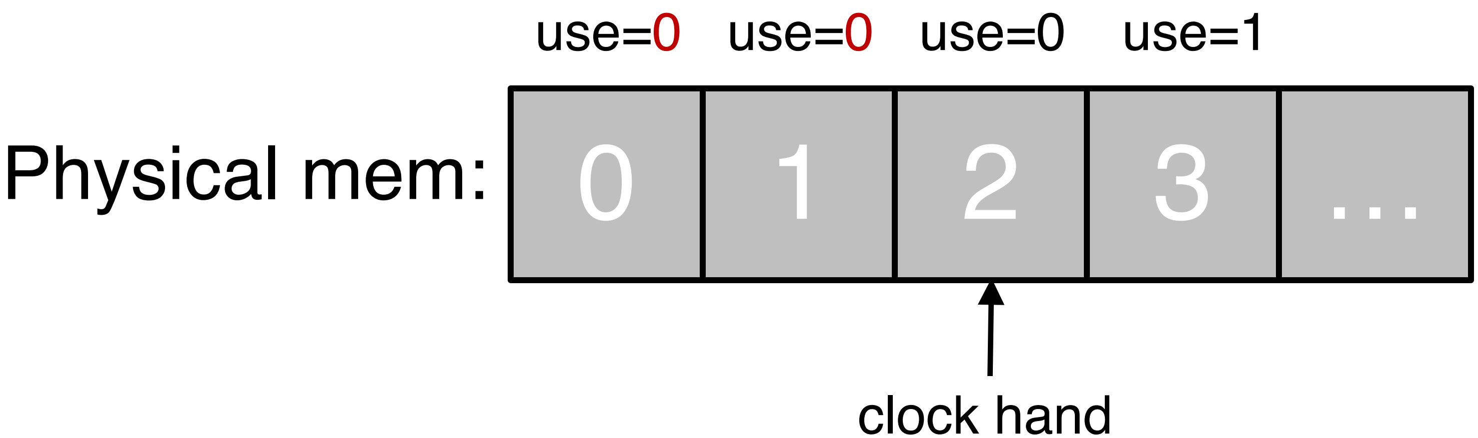
Mem is full, and to evict a page to make room

Clock: Look For a Page



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

Clock: Look For a Page

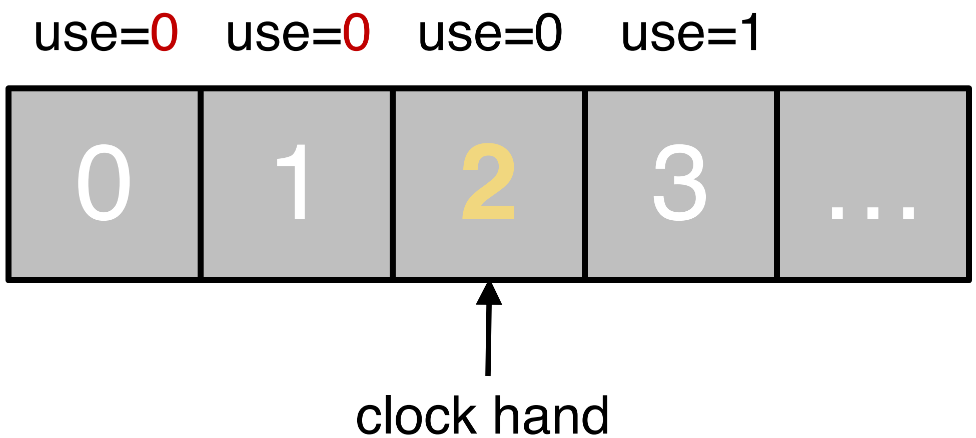


Mem is full, and to evict a page to make room

Clock: Look For a Page

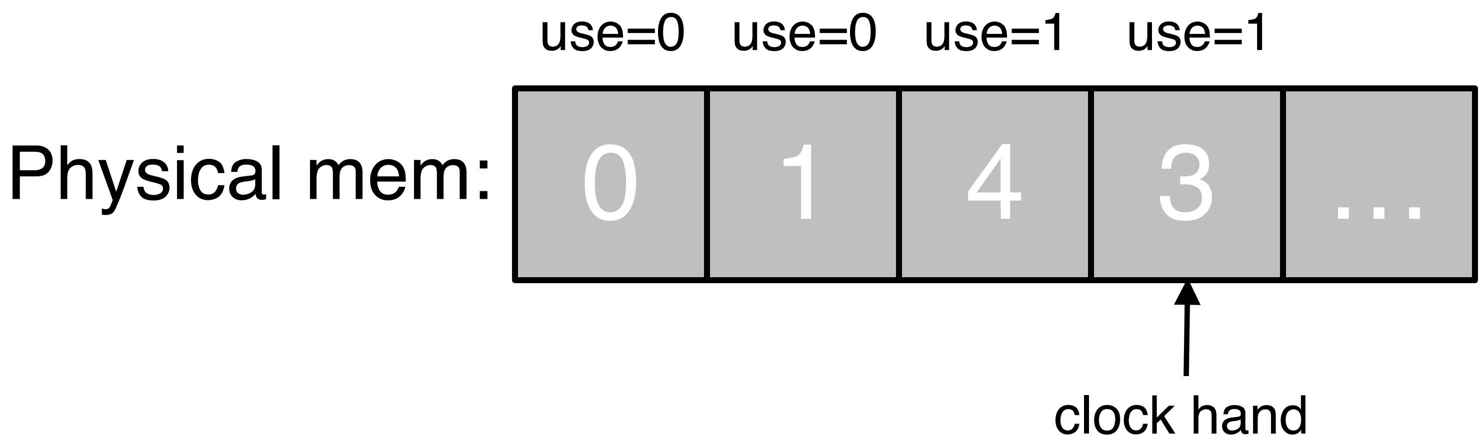
Evict **page 2** because it has not been recently used

Physical mem:



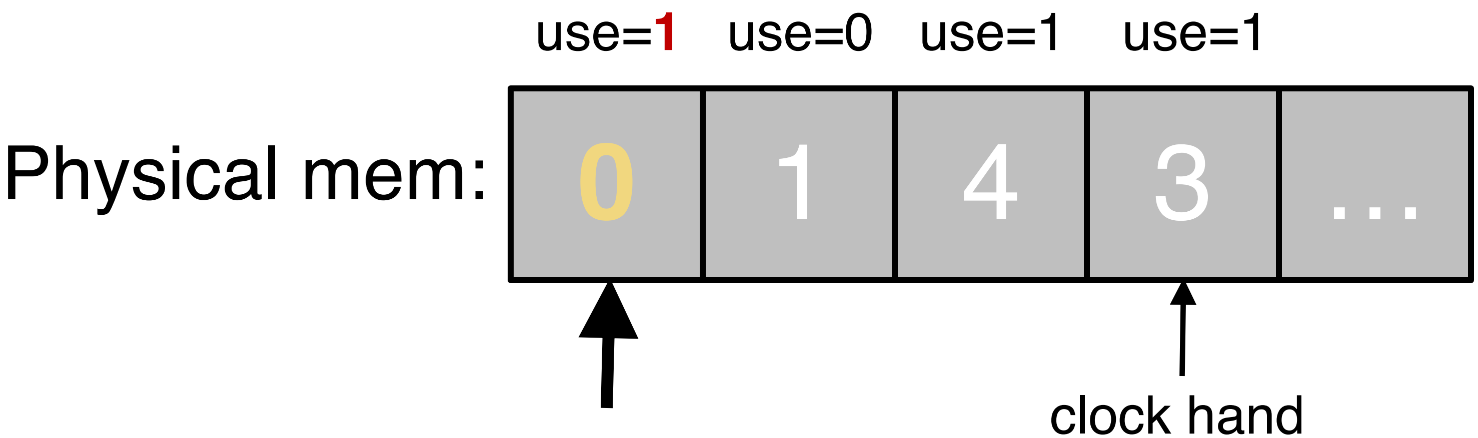
Mem is full, and to evict a page to make room

Clock: Look For a Page

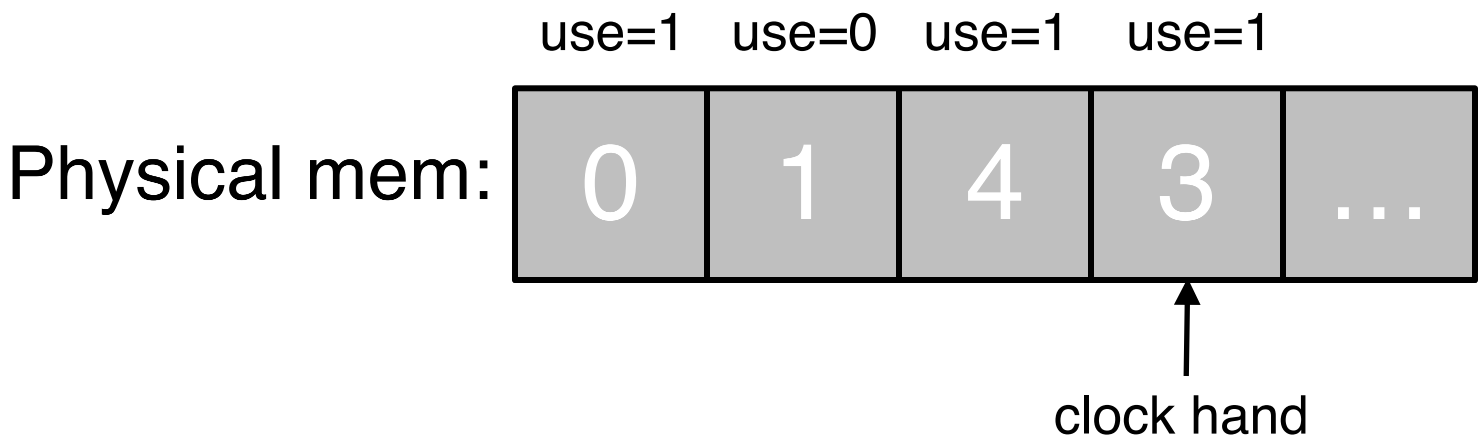


Clock: Access a Page

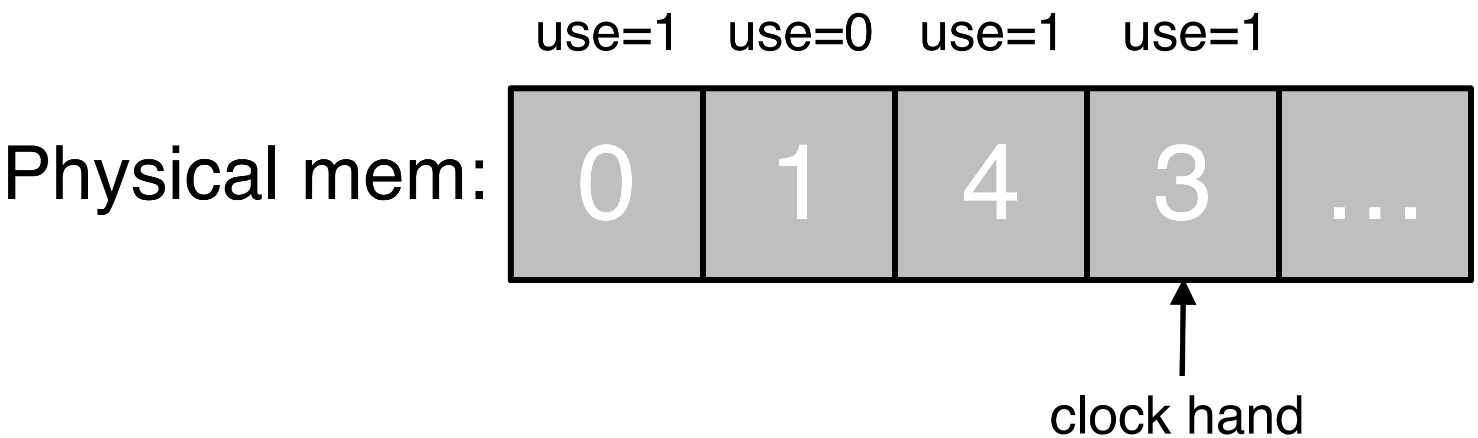
page 0 is accessed



Clock: Look For a Page

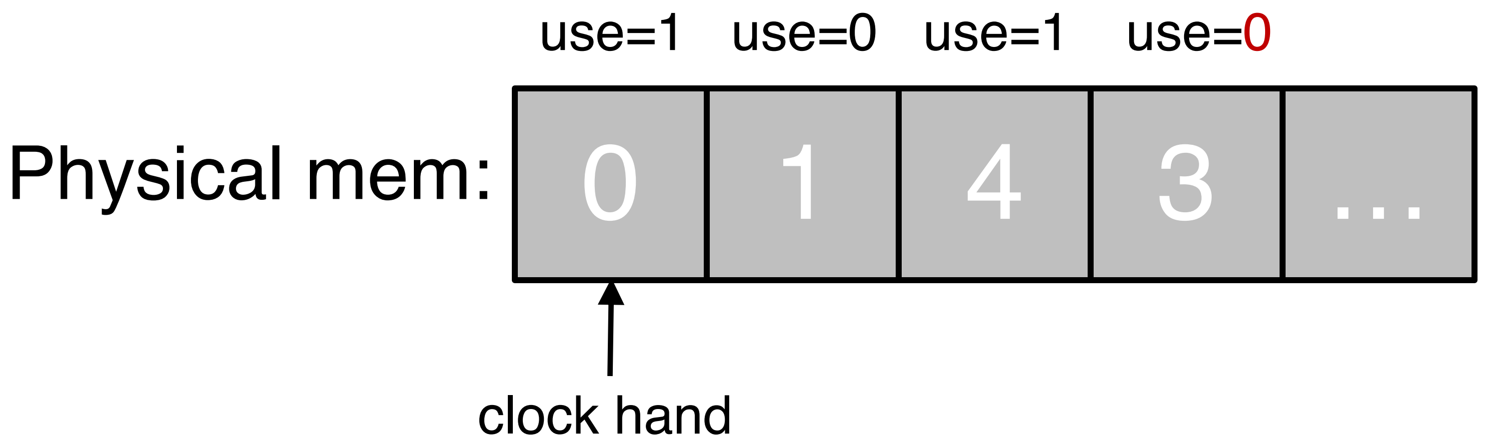


Clock: Look For a Page



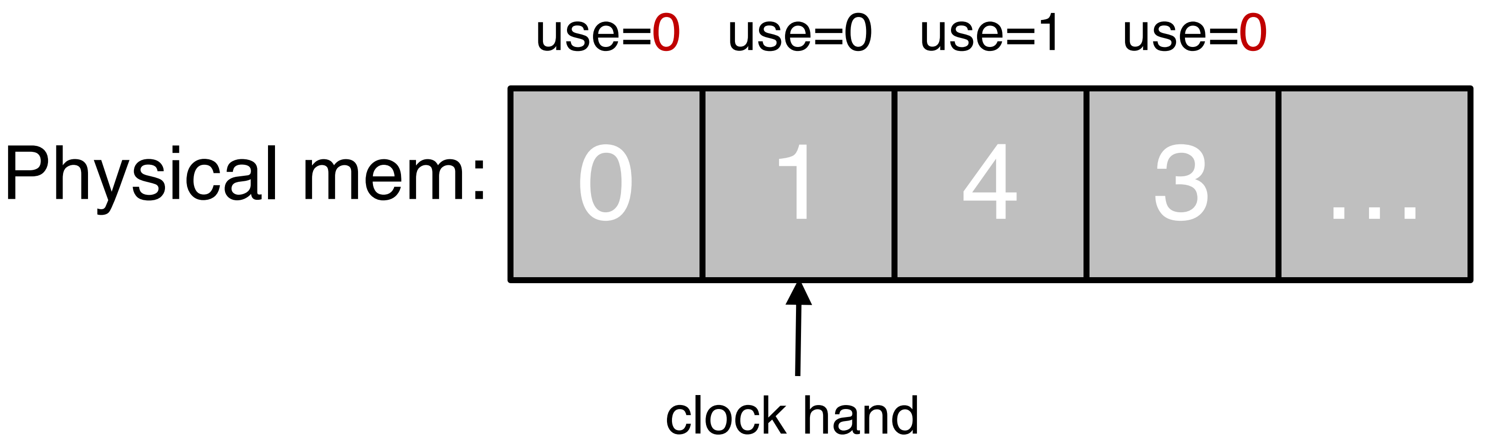
Mem is full, and to evict a page to make room

Clock: Look For a Page



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

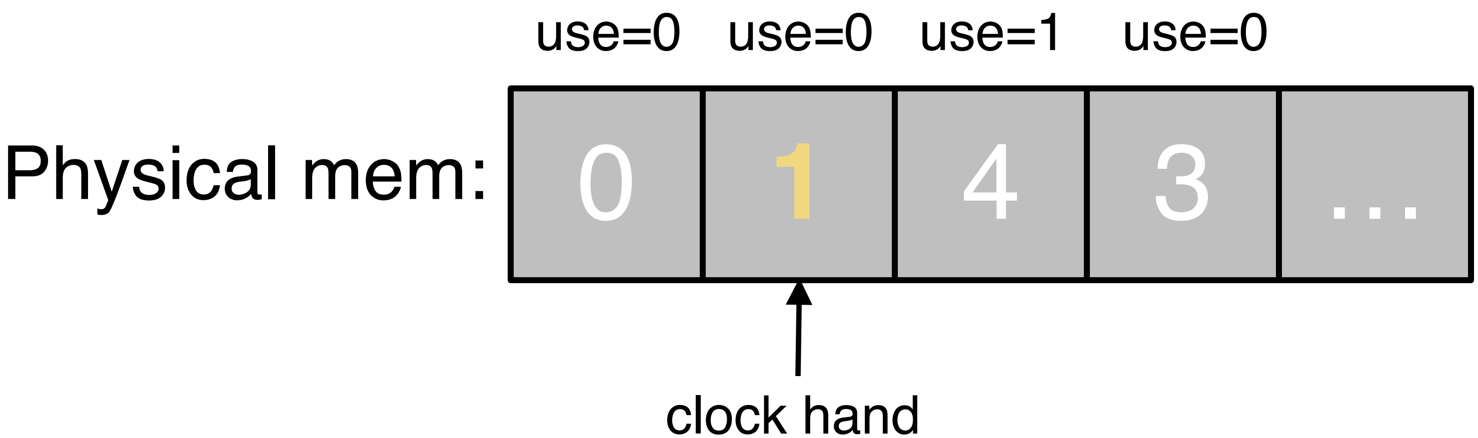
Clock: Look For a Page



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

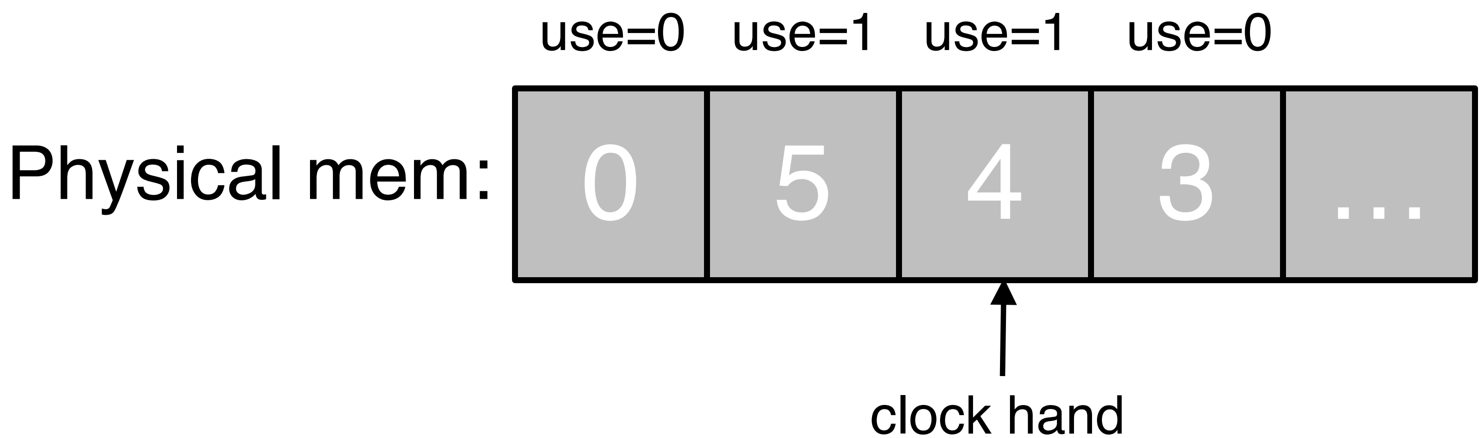
Clock: Look For a Page

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**)

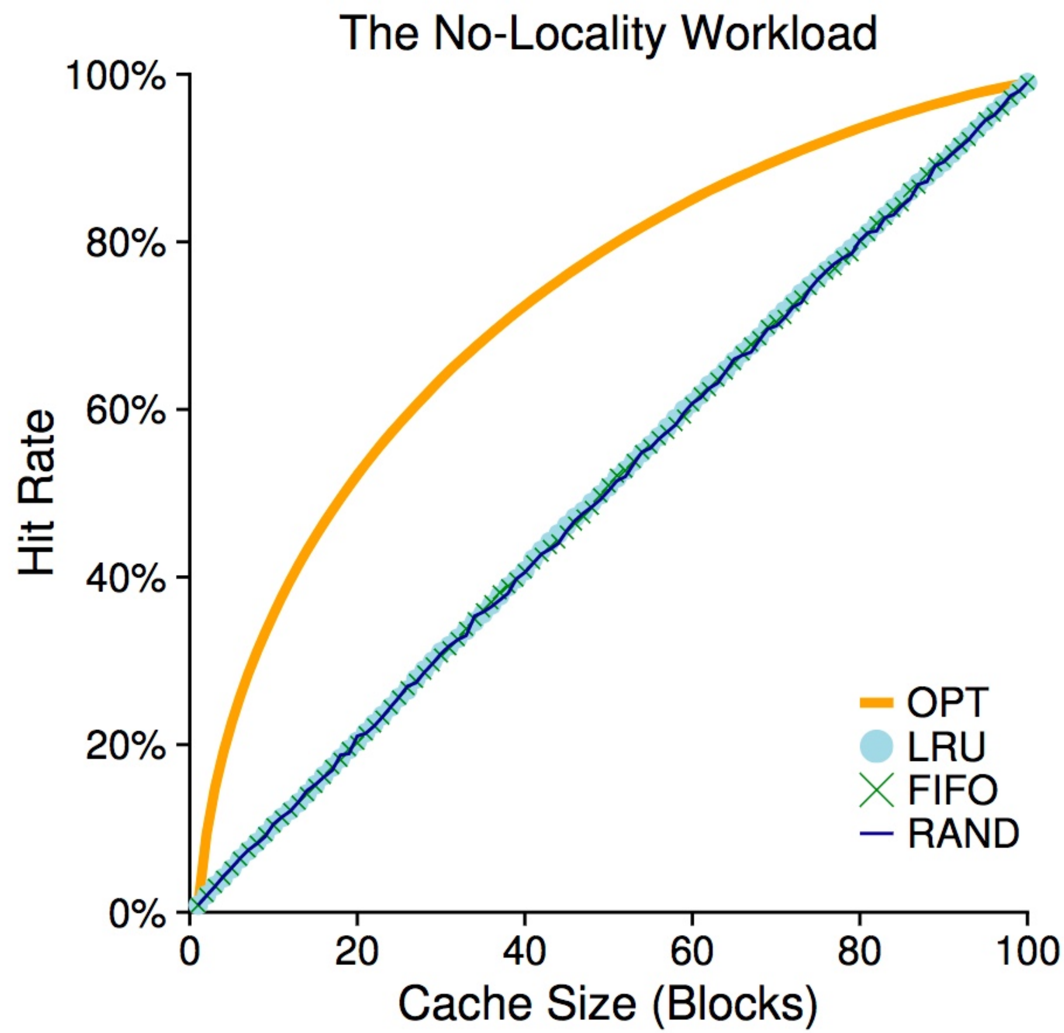
Page Replacement Workload Examples

Workload Examples

- A simple workload
  - Workload consists of a working set of 100 pages
  - Workload issues 10,000 access requests

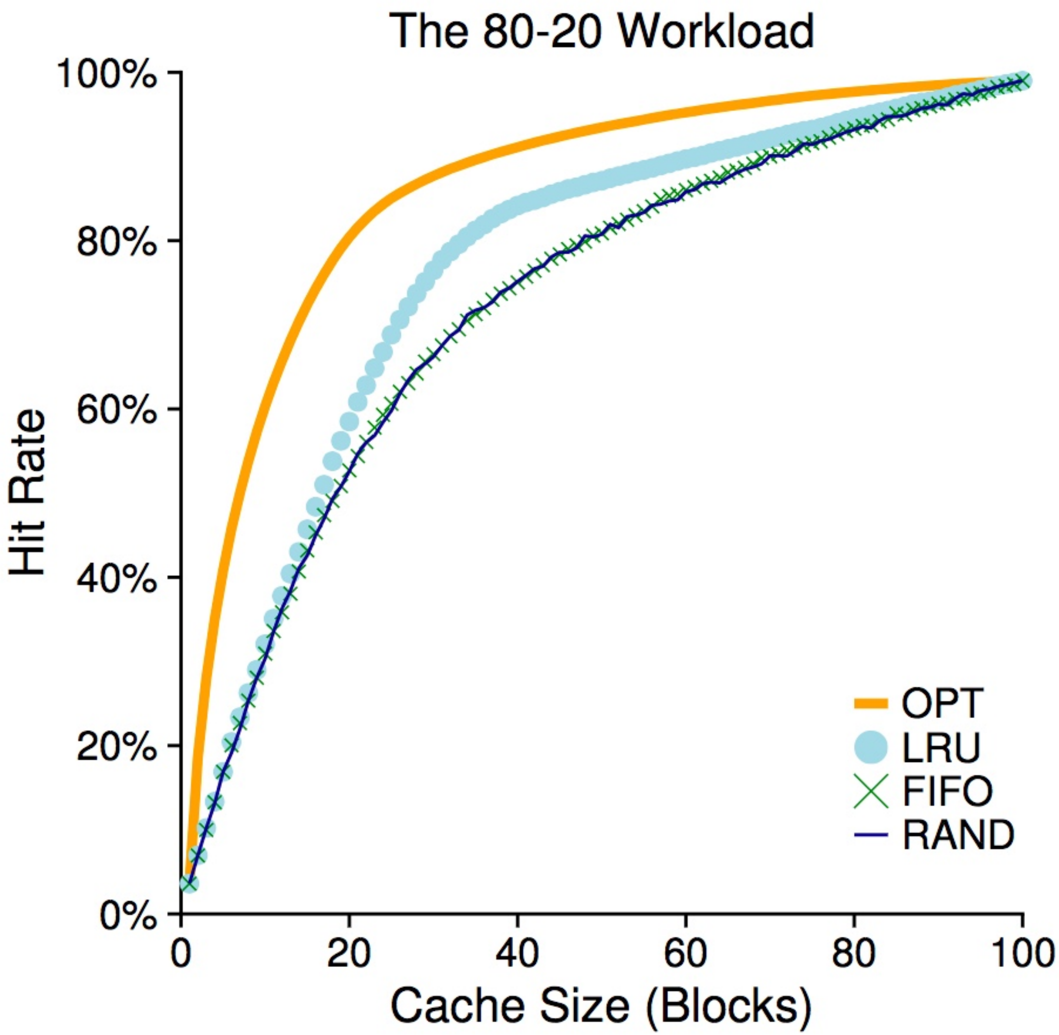
- Four replacement policies
  - OPT: The optimal
  - LRU: Least-recently used
  - FIFO: First-in first-out
  - RAND: Random

The No-Locality Workload



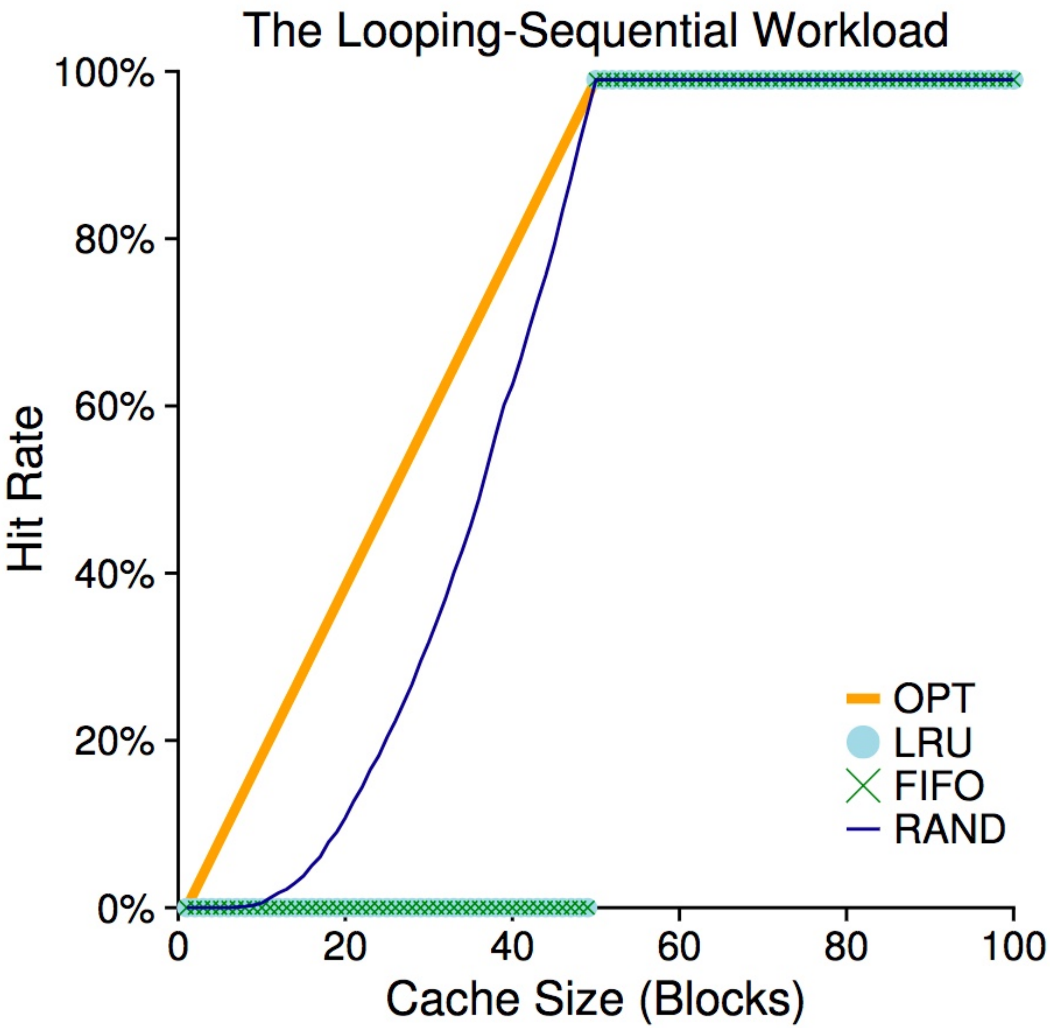
Each reference is to a random page within the set of accessed pages

The 80-20 Workload



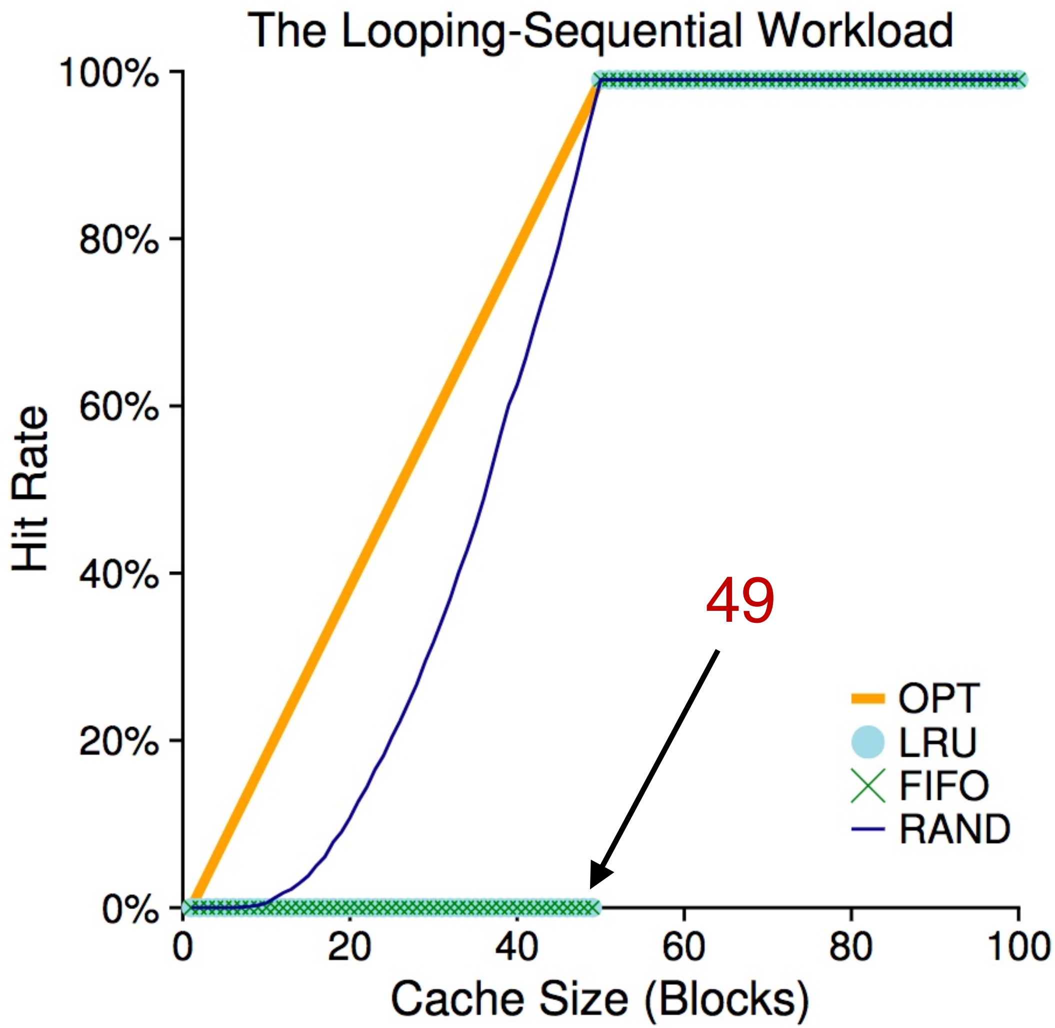
80-20: 80% of the refs are made to 20% of the pages ("**hot**" pages)

The Looping-Sequential Workload



Loop first 50 pages starting from 0 to 49 for a total of 10,000 accesses

The Looping-Sequential Workload



Loop first 50 pages starting from 0 to 49 for a total of 10,000 accesses

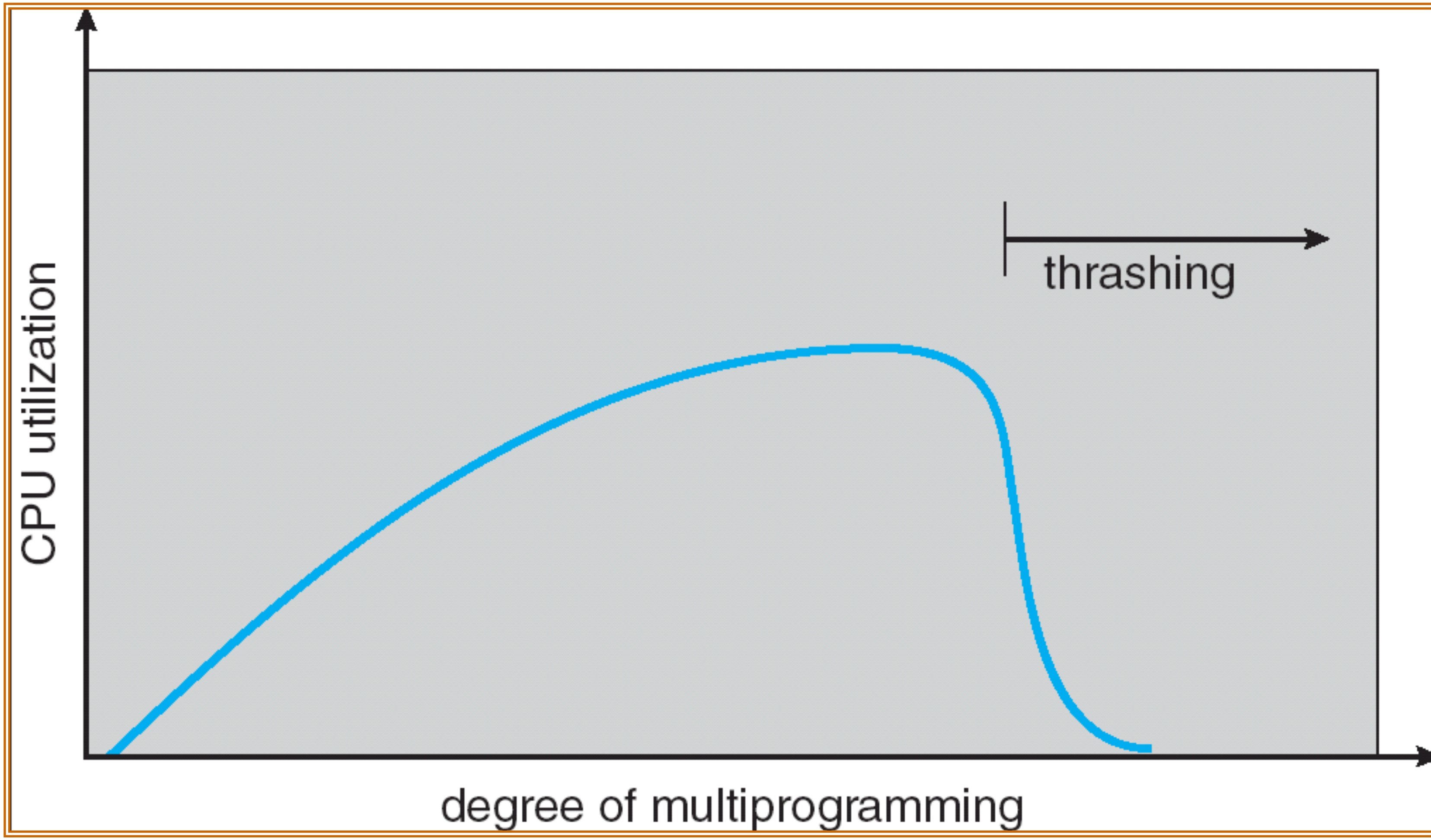
Thrashing

Thrashing

- High-paging activity: The system is spending more time paging than executing

- How can this happen?
  - OS observes low CPU utilization and increases the degree of multiprogramming
  - Global page-replacement algorithm is used, it takes away frames belonging to other processes
  - But these processes need those pages, they also cause page faults
  - Many processes join the waiting queue for the paging device, CPU utilization further decreases
  - OS introduces new processes, further increasing the paging activity

CPU Utilization vs. the Degree of Multiprogramming



How to Avoid Thrashing?

- To avoid thrashing, earlier OS did admission control to only run a subset of processes

- Some current OS takes more draconian approach
  - E.g., some Linux runs an out-of-memory killer to choose a memory-intensive process and kill it

Review: Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - Support more processes/users
- Page is needed ⇒ use the reference to page
  - If not in memory ⇒ must bring from the disk
- Demand paging versus swapping
  - Fetching the page in only on demand vs. kicking out one victim then paging in one under mem pressure

Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap

- **Why does thrashing occur?**
  `Σ size of locality > total memory size`
  `Or Σ working set size > total memory size`

- Definition of **working set size** (WSS): number of unique items that are accessed

Impact of Program Structures on Memory Performance

Impact of Program Structure on Memory Performance

- Consider an array named data with `128*128` elements
- Each row is stored in one page (of size 128 words)

Impact of Program Structure on Memory Performance

- Consider an array named data with `128*128` elements
- Each row is stored in one page (of size 128 words)
- **Program 1**

```
for (j = 0; j <128; j++)
  for (i = 0; i < 128; i++)
    data[i][j] = 0;
```

128 x 128 = **16,384** page faults

Impact of Program Structure on Memory Performance

- Consider an array named data with `128*128` elements
- Each row is stored in one page (of size 128 words)
- **Program 1**

```
for (j = 0; j <128; j++)
    for (i = 0; i < 128; i++)
        data[i][j] = 0;
```

128 x 128 = **16,384** page faults

- **Program 2**

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i][j] = 0;
```

Only **128** page faults