TLB & Page Replacement

Computer Operating Systems, Fall 2023

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Andy Jiang	Haoyun Qin	Kevin Bernat	Ryoma Harris
Audrey Yang	Jason hom	Leon Hertzberg	Shyam Mehta
August Fu	Jeff Yang	Maxi Liu	Tina Kokoshvili
Daniel Da	Jerry Wang	Ria Sharma	Zhiyan Lu
Ernest Ng	Jinghao Zhang	Rohan Verma	

Poll Everywhere

pollev.com/tqm

- How is proj1 milestone going?

Administrivia

- Project 1 is out now
  - The milestone was due **YESTERDAY** Wed 9/27 @ 11:59 pm
    late deadline: 11:59 pm on Sun, Oct 01
  - Project is due 11:59 pm on Wed, Oct 11
    late deadline 11:59 pm on Sun, Oct 15
- For project 1 full submission, please do a group submission on gradescope (one of you submits but you add your partner to the submission)
- Check-in out tomorrow-ish, due Tuesday @ 1 pm
- Recitation next week tentatively on process groups, terminal control and waitpid

Poll Everywhere

pollev.com/tqm

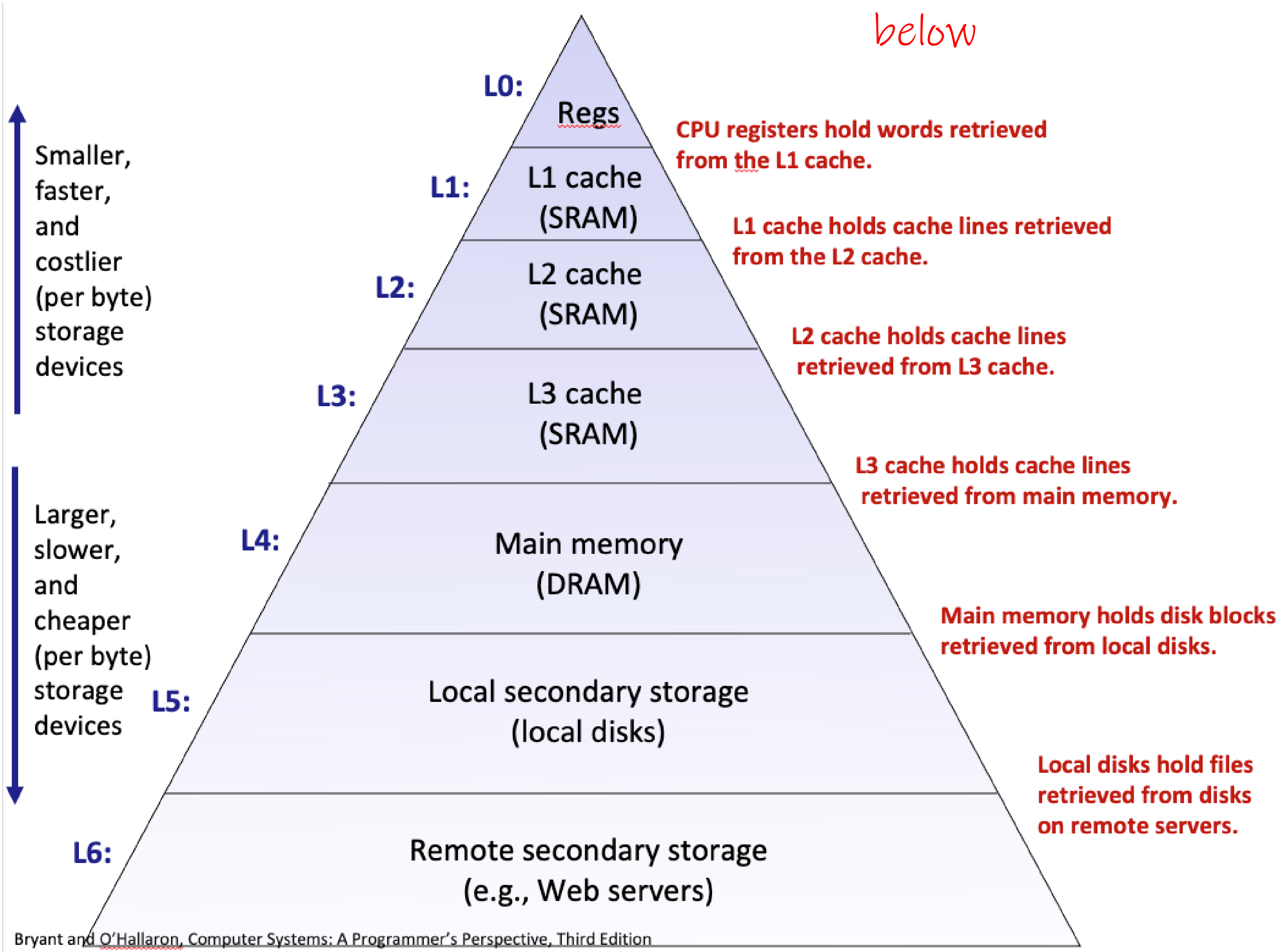
- Any questions, comments or concerns from last lecture?

Lecture Outline

- **Memory Hierarchy**
- TLB
- Page Replacement: High Level
  - FIFO
  - Reference Strings
  - Beladys
- LRU
- Thrashing
- FIFO w/ Reference bit

Memory Hierarchy

Each layer can be thought of as a "cache" of the layer below



Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Data Access Time

- Data is stored on a physical piece of hardware
- The distance data must travel on hardware affects how long it takes for that data to be processed
- Example: data stored closer to the CPU is quicker to access
  - We see this already with registers. Data in registers are stored on the chip and are faster to access than memory
- Data that is further away can't be used immediately, so CPU can't be fully utilized, CPU has to wait for data

Principle of Locality

- The tendency for the CPU to access the same set of memory locations over a short period of time

- Two main types:
  - **Temporal Locality**: If we access a portion of memory, we will likely reference it again soon
  - **Spatial Locality**: If we access a portion of memory, we will likely reference memory close to it in the near future.

- The OS can take advantage of these tendencies to help with page management

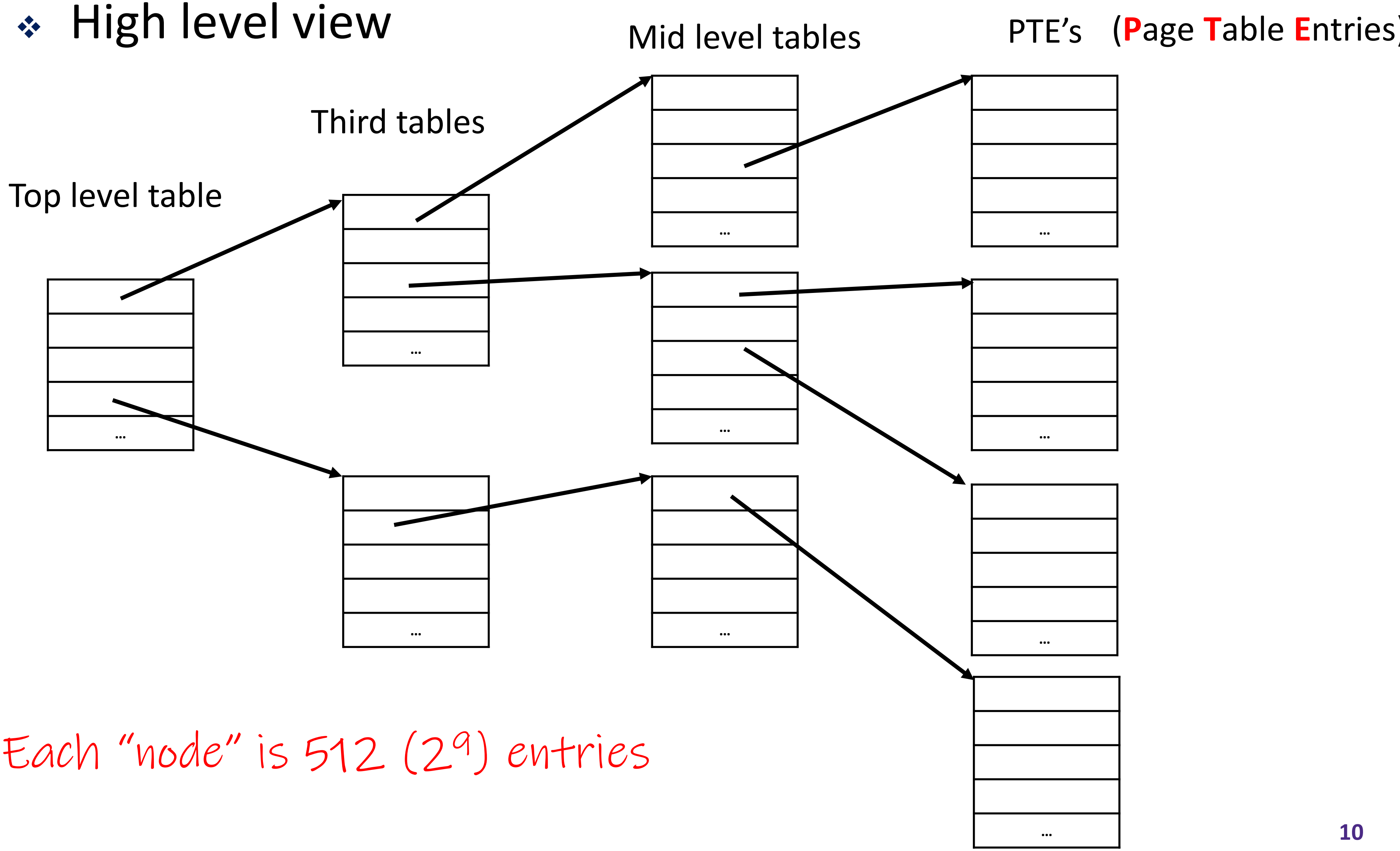
Multi Level Page Table

- If you’ve heard of a Trie or a prefix tree, then this is basically that
- On a 64-bit address, we keep the bottom 12 bits for the page offset, and the upper 52 for the page number.
- We can split the page number into 4 groups of 9 bits (ignore the remainder)

Ignored 16 bits	G offset 9 bits	U offset 9 bits	M offset 9 bits	PTE offset 9 bits	Page Offset 12 bits

Diagram

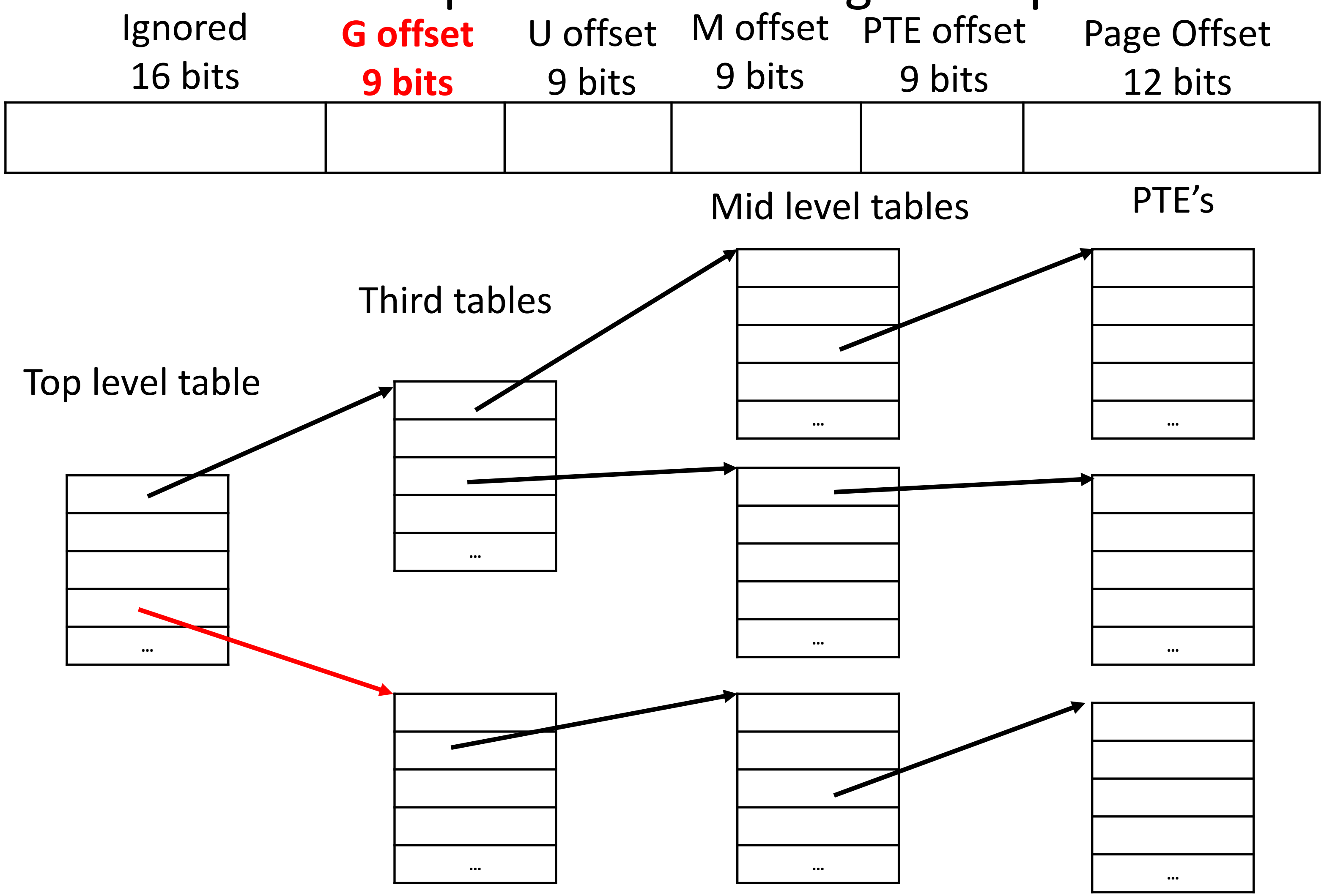
- High level view



Each "node" is 512 (2^9) entries

Looking up an address

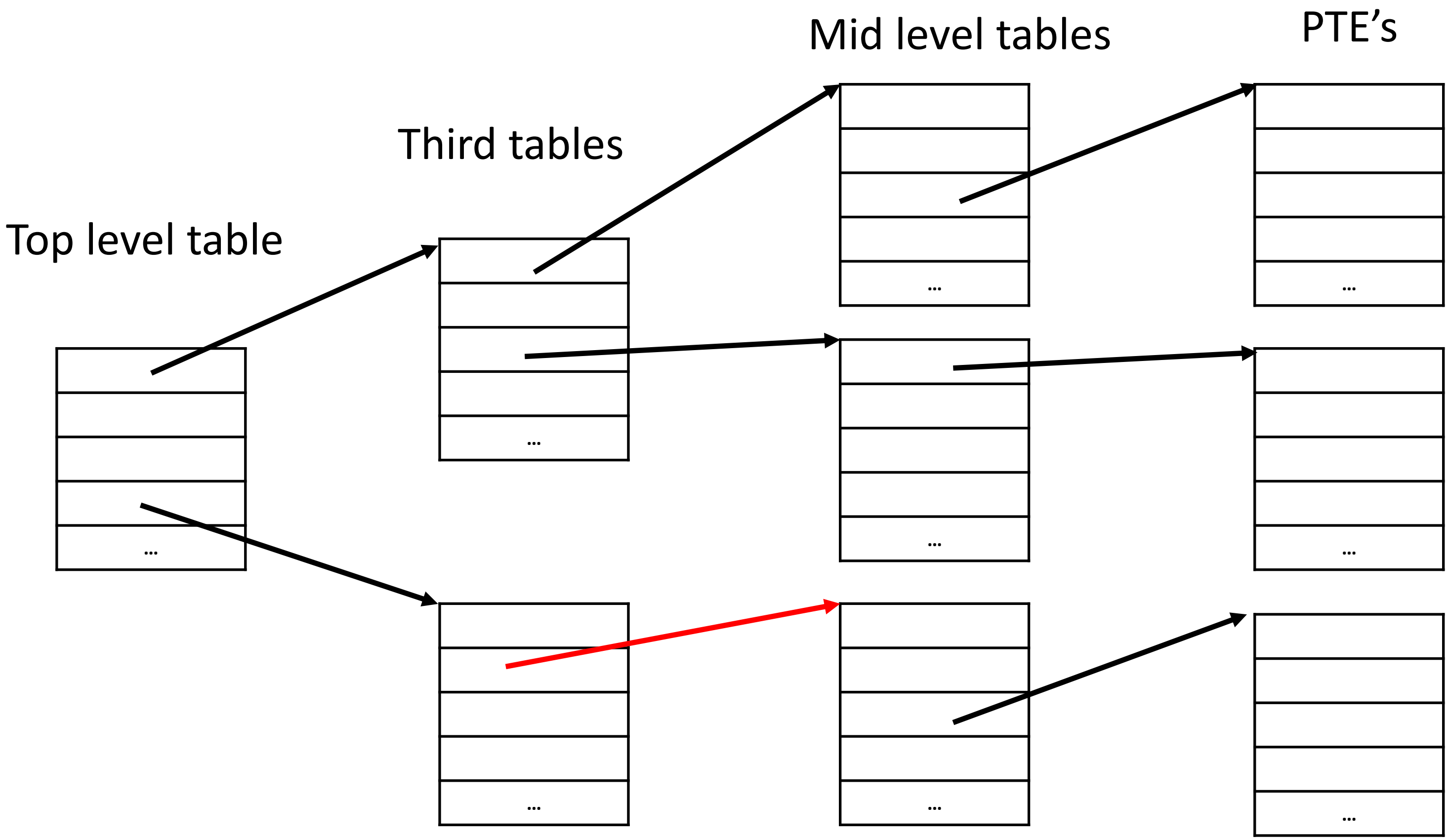
- First index into top level table using the top 9-bit chunk



Looking up an address

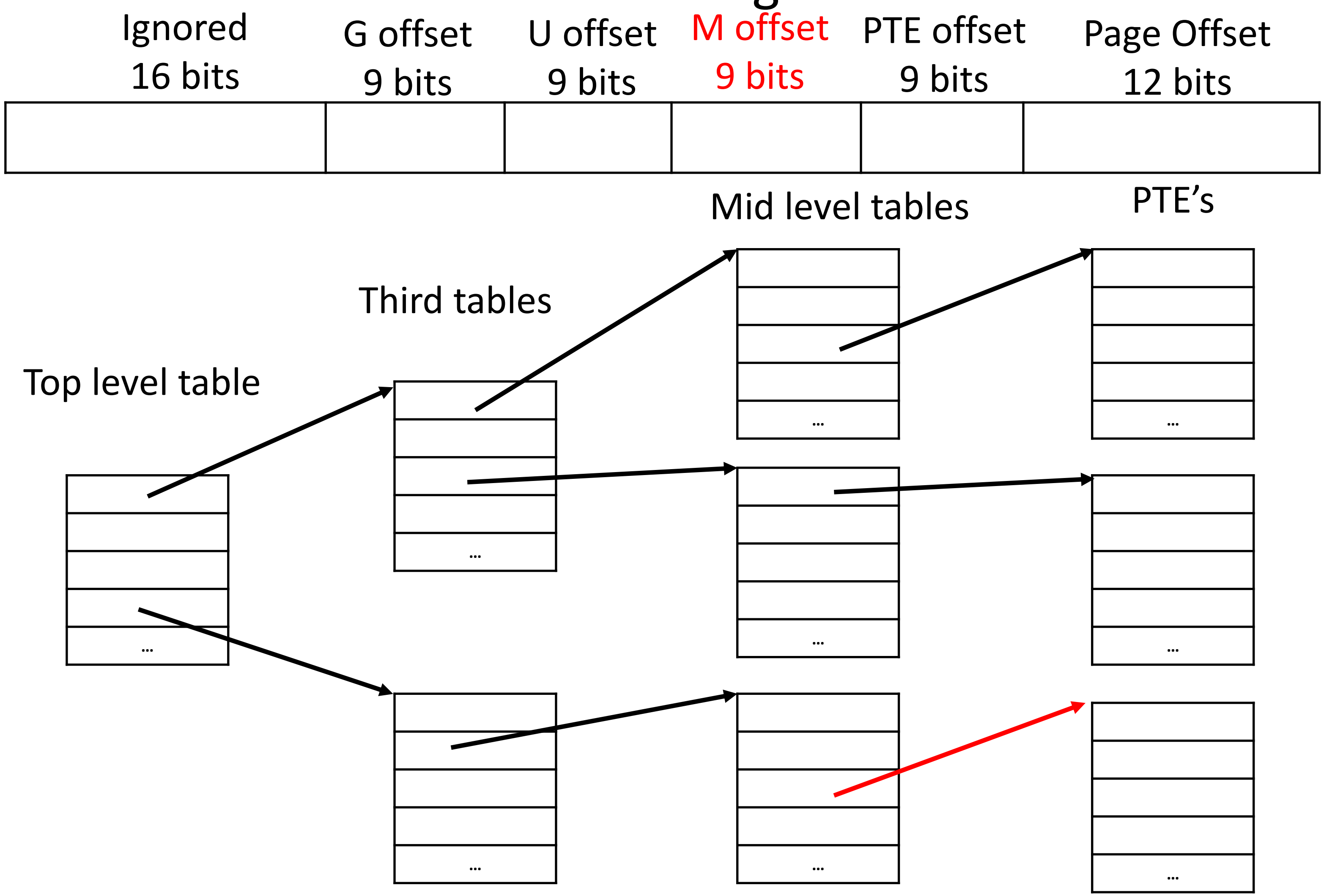
- Index into next level table using the next 9-bit chunk

Ignored	G offset	U offset	M offset	PTE offset	Page Offset
16 bits	9 bits	9 bits	9 bits	9 bits	12 bits



Looking up an address

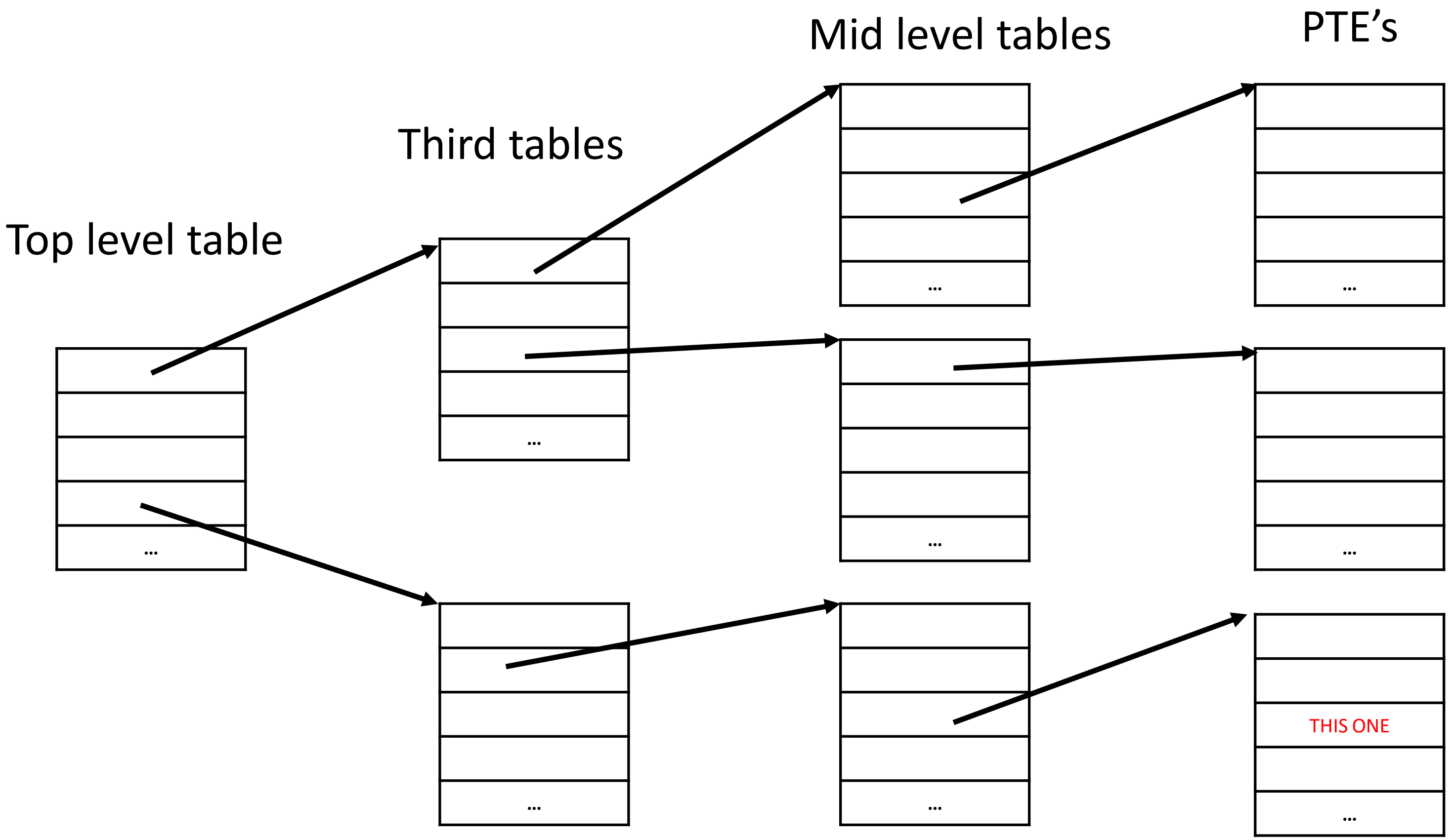
- Index into next level table using the next 9-bit chunk



Looking up an address

- Access the page table entry based on the last 9 bits

Ignored	G offset	U offset	M offset	**PTE offset**	Page Offset
16 bits	9 bits	9 bits	9 bits	**9 bits**	12 bits



Analysis pt. 2

- Take advantage of **spatial locality**: if a particular memory location is referenced, it is likely that it and nearby memory locations will be accessed soon
  - If pages near each other in memory are accessed, they will in the same nodes in the tree! Not every page access requires the creation of a mid-level node
  - I'll revisit the idea of locality later

- **What was once just one memory access to lookup page frame is now four memory accesses ☹**
  - **This can be very expensive time-wise**
  - **There is hardware (TLB) that helps a lot with this ☺ (more now)**

Lecture Outline

- Memory Hierarchy
- **TLB**
- Page Replacement: High Level
  - FIFO
  - Reference Strings
  - Beladys
- LRU
- Thrashing
- FIFO w/ Reference bit

TLB

- Transition Lookaside Buffer

- A special piece of hardware memory that is quick to do lookups in. **Stores <u>recent</u> virtual page to physical frame translations.**
  - Hardware for TLB is special, it can quickly check all entries to see if a specific virtual page number translation is in their or not
    - Hardware is expensive, so the TLB is kept relatively small usually
    - Usually quicker hardware -> more expensive. To save cost, things using special hardware are kept smaller

- **<u>TLB prevents MMU from having to read the page table on each translation</u>**

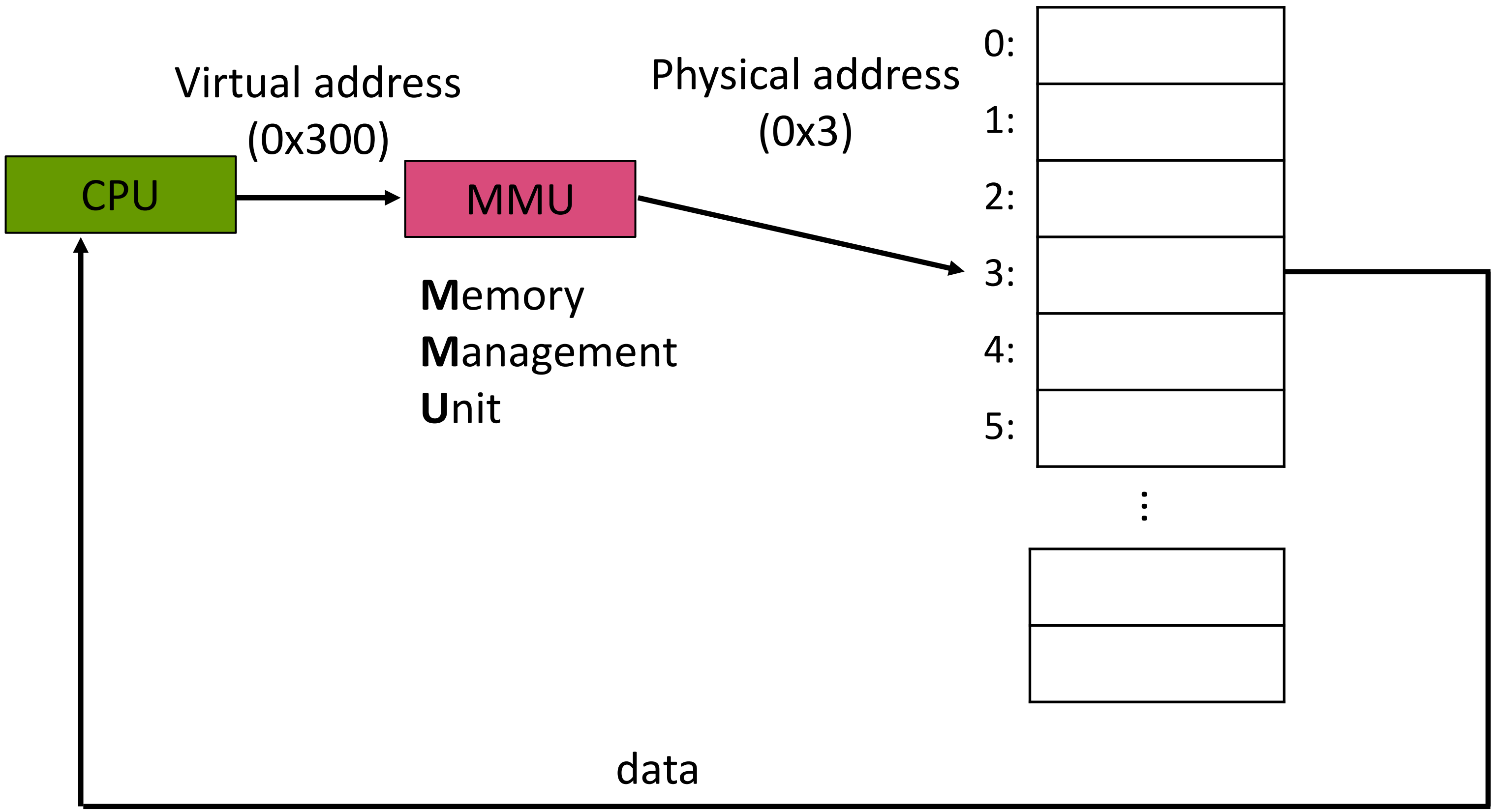
TLB Locality

- Can only store a subset of the translations of the translations in the page table

- TLB takes advantage of temporal locality to decide which pages should be stored inside of it
  - Pages that are accessed are likely to be accessed soon in the future

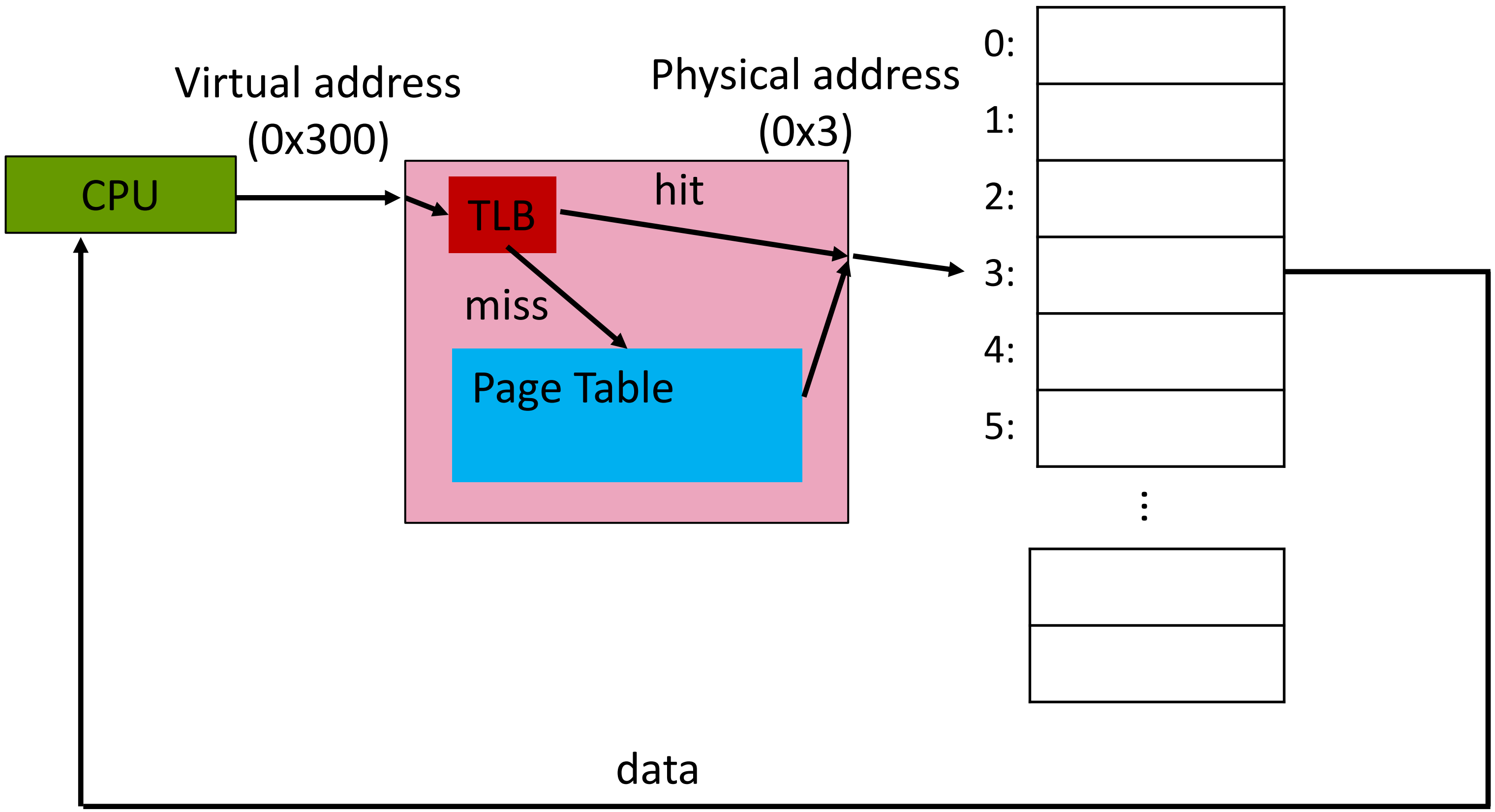
High Level View

- Programs don't know about physical addresses; virtual addresses are translated into them by the MMU



High Level View

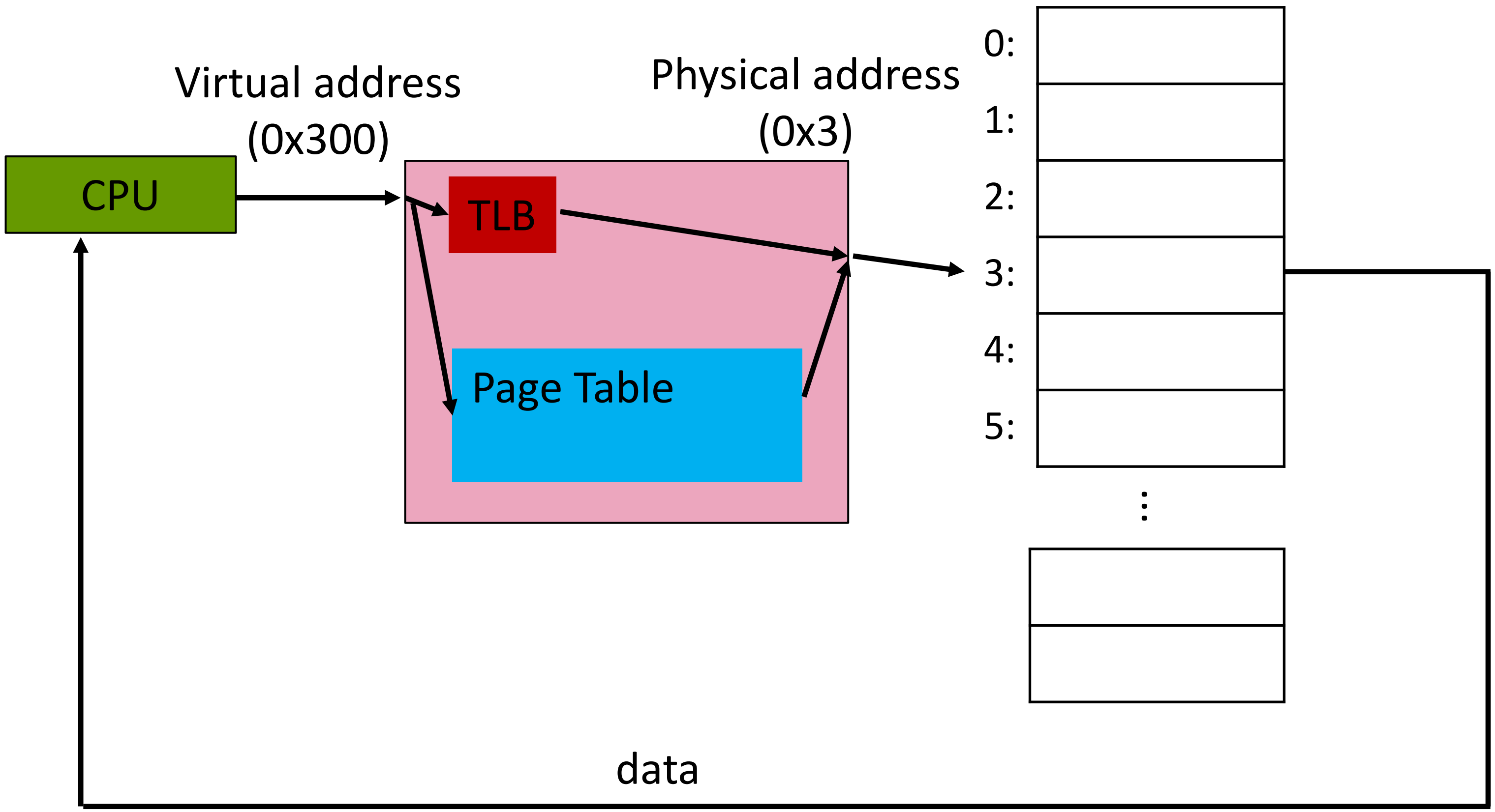
- MMU Translation is a bit complicated, has multiple steps



High Level View: Slight optimization

- MMU Translation is a bit complicated, has multiple steps

Can check TLB and page table in parallel



TLB: More Details

- Entries in the TLB need to store:
  - The virtual page -> physical frame mapping
  - Dirty & Permission bits stored in TLB

- TLB Entries need to be kept in sync with the page table
  - If a TLB entry is updated, the page table must be synced to have the updated dirty bit value
  - If a page is evicted from the page table, but is in the TLB, then that entry must be removed from the TLB

- To maintain process isolation, one of two things
  - When we switch executing processes, the TLB is cleared
  - TLB entries also contain a PID tag to enforce isolation

Lecture Outline

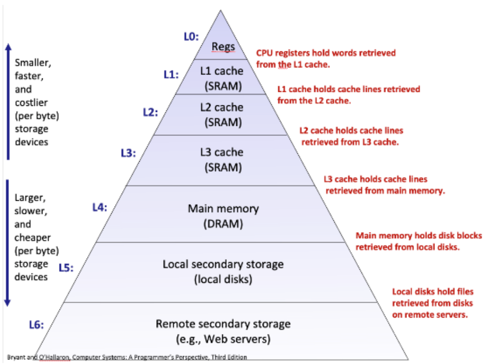
- Memory Hierarchy
- TLB
- **Page Replacement: High Level**
  - **FIFO**
  - **Reference Strings**
  - **Beladys**
- LRU
- Thrashing
- FIFO w/ Reference bit

Page Replacement

- The operating system will sometimes have to evict a page from physical memory to make room for another page.

- If the evicted page is access again in the future, it will cause a page fault, and the Operating System will have to go to Disk to load the page into memory again

- Remember this? Disk access is very very slow (relatively speaking).
  - How can we minimize disk accesses?
  - How can we try to ensure the page we evict from memory is unlikely to be used again in the future?



Reference String

- A reference string is a string representing a sequence of virtual page accesses. By a given process on some input.
  - E.g., `0 1 2 3 4 1 2 9 5 3 2 2 …`
  - Page 0 is accessed, then 1, then 2, then 3 …

- These strings are useful for reasoning about page replacement policies, and how they act on certain page access patterns

FIFO Replacement

- One way to decide which pages can be evicted is to use FIFO (First in First Out)

- If a page needs to be evicted from physical memory, then the page that has been in memory the longest (since it was last brought into memory) can be evicted.

FIFO Replacement

- If we have 4 frames, and the reference string:
  4 1 1 2 3 4 **5**
  - Red numbers indicate that accessing the page caused a page fault. Accessing 5 also causes 4 to be evicted from physical memory

	Ref str:	4	1	1	2	3	4	5
Newest		4	1	1	2	3	3	5
			4	4	1	2	2	3
					4	1	1	2
Oldest						4	4	1

Poll Everywhere

pollev.com/tqm

- Given the following reference string, how many page faults occur when using a FIFO algorithm
- 1 2 3 4 1 2 5 1 2 3 4 5

- Assume that
  - physical memory has three frames
  - we can ignore sharing those frames with other processes.
  - Physical Memory starts empty

- Part 2: If we didn’t have to follow a strict policy, what is the “optimal” pages that could be evicted to minimize faults? How many less faults would we have?

Poll Everywhere

pollev.com/tqm

- Given the following reference string, how many page faults occur when using a FIFO algorithm
- 1 2 3 4 1 2 5 1 2 3 4 5
- FIFO

	Ref str:	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
Newest		1	2	3	4	1	2	5	5	5	3	4	4
			1	2	3	4	1	2	2	2	5	3	3
Oldest				1	2	3	4	1	1	1	2	5	5
Victim					1	2	3	4			1	2	

- 9 faults

Poll Everywhere

pollev.com/tqm

- Given the following reference string, how many page faults occur when using a FIFO algorithm
- 1 2 3 4 1 2 5 1 2 3 4 5
- Theoretical optimal?

	Ref str:	1	2	3	4	1	2	5	1	2	3	4	5
		1	2	3	4	4	4	5	5	5	3	4	4
			1	2	2	2	2	2	2	2	5	3	3
				1	1	1	1	1	1	1	2	5	5
Victim					3			4			1	2	

- 7 faults

"optimal" replacement

- If you knew the exact sequence of page accesses in advance, you could optimize for smallest number of page faults

- Always replace the page that is furthest away from being used again in the future
  - How do we predict the future??????
  - You can't, but you can make a "best guess" (later in lecture)

- Optimal replacement is still a handy metric. Used for testing replacement algorithms, see how an algorithm compares to various "optimal" possibilities.

Poll Everywhere

pollev.com/tqm

- Given the following reference string, how many page faults occur when using a FIFO algorithm
- 3 2 1 0 3 2 4 3 2 1 0 4

- Assume that
  - physical memory has three frames
  - we can ignore sharing those frames with other processes.
  - Physical Memory starts empty

- Part 2: What is we had 4 page frames, how many faults would we have?

Poll Everywhere

pollev.com/tqm

- Given the following reference string, how many page faults occur when using a FIFO algorithm
- 3 2 1 0 3 2 4 3 2 1 0 4

- Three page frames

	Ref str:	**3**	**2**	**1**	**0**	**3**	**2**	**4**	**3**	**2**	**1**	**0**	**4**
Newest		3	2	1	0	3	2	4	4	4	1	0	0
			3	2	1	0	3	2	2	2	4	1	1
Oldest				3	2	1	0	3	3	3	2	4	4
Victim					3	2	1	0			3	2	

- 9 faults

Poll Everywhere

pollev.com/tqm

- Given the following reference string, how many page faults occur when using a FIFO algorithm
- 3 2 1 0 3 2 4 3 2 1 0 4
- Four page frames

	Ref str:	3	2	1	0	3	2	4	3	2	1	0	4
Newest		3	2	1	0	0	0	4	3	2	1	0	4
			3	2	1	1	1	0	4	3	2	1	0
				3	2	2	2	1	0	4	3	2	1
Oldest					3	3	3	2	1	0	4	3	2
Victim								3	2	1	0	4	3

- 10 faults

Bélády's anomaly

- Sometimes increasing the number of page frames results in an increase in the number of page faults ☹
- This behaviour is something that we want to avoid/minimize the possibility of.
- Stack based algorithms (Optimal, LIFO, LRU) avoid this issue

Lecture Outline

- Memory Hierarchy
- TLB
- Page Replacement: High Level
  - FIFO
  - Reference Strings
  - Beladys
- **LRU**
- Thrashing
- FIFO w/ Reference bit

LRU (Least Recently Used)

- If a page is used recently, it is likely to be used again in the near future
- Use past knowledge to predict the future
- Replace the page that has had the longest time since it was last used

	Ref str:	4	0	1	2	0	3	0	4	2	3	0	3
Most recently used		4	0	1	2	0	3	0	4	2	3	0	3
			4	0	1	2	0	3	0	4	2	3	0
LRU				4	0	1	2	2	3	0	4	2	2
Victim					4		1		2	3	0	4	

Poll Everywhere

pollev.com/tqm

- What if there are four frames instead of 3? How Many Page Faults?

	Ref str:	4	0	1	2	0	3	0	4	2	3	0	3
Most recently used													
LRU													
Victim													

Poll Everywhere

pollev.com/tqm

- What if there are four frames instead of 3? How Many Page Faults?
  - 6 faults

	Ref str:	4	0	1	2	0	3	0	4	2	3	0	3
Most recently used		4	0	1	2	0	3	0	4	2	3	0	3
			4	0	1	2	0	3	0	4	2	2	0
				4	0	1	2	2	3	0	4	4	2
LRU					4	4	1	1	2	3	0	3	4
Victim							4		1				

LRU Implementation?

- To implement this properly, there are a couple possibilities
  - we would need to timestamp each memory access and keep a sorted list of these pages
    - High overhead, timestamps can be tricky to manage :/
  - Keep a counter that is incremented for each memory access Look through the table to find the lowest counter value on eviction
    - Looking through the table can be slow
    - How do you distinguish a process that has been accessed a lot in the past vs one accessed a little more recently?
  - Whenever a page is accessed find it in the stack of active pages and move it to the bottom

LRU Approximation: Reference Bit & Clock

- It is expensive to do bookkeeping every time a page is accessed. Minimize the bookkeeping if possible

- When we access a page, we can update the reference bit for that PTE to show that it was accessed recently
  - This is done automatically by hardware, when accessing memory.
  - Setting a bit to 1 is much quicker than managing time stamps and re-organizing a stack

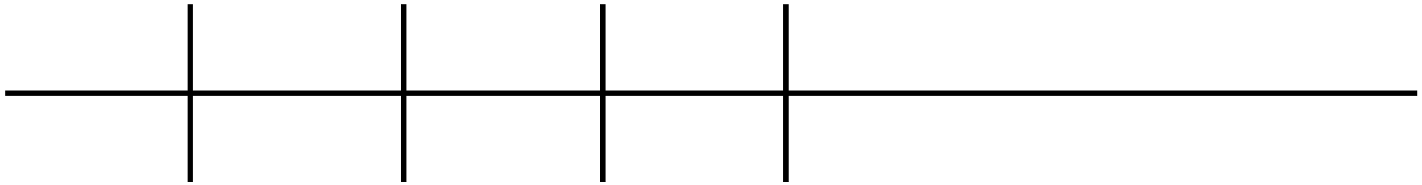
- We could check the reference bit at some clock interval to see if the page was used at all in the last interval period

LRU Approximation: Aging

- Each page gets an 8-bit counter.
- On clock interval and for every page:
  - shift the counter to the right by 1 bit
  - copy the reference bit into the MSB of the counter.
  - Reference bit in the PTE is reset to 0
- If we read the counter as an unsigned integer, then a larger value means the counter was accessed more recently

Aging Illustration

- Timeline



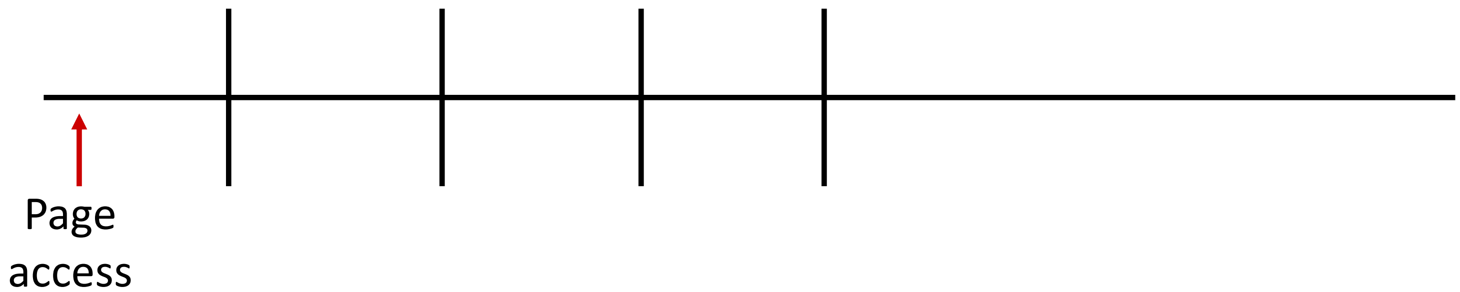
- Counter:

0	0	0	0	0	0	0	0

- Ref bit: 0

Aging Illustration

- Timeline



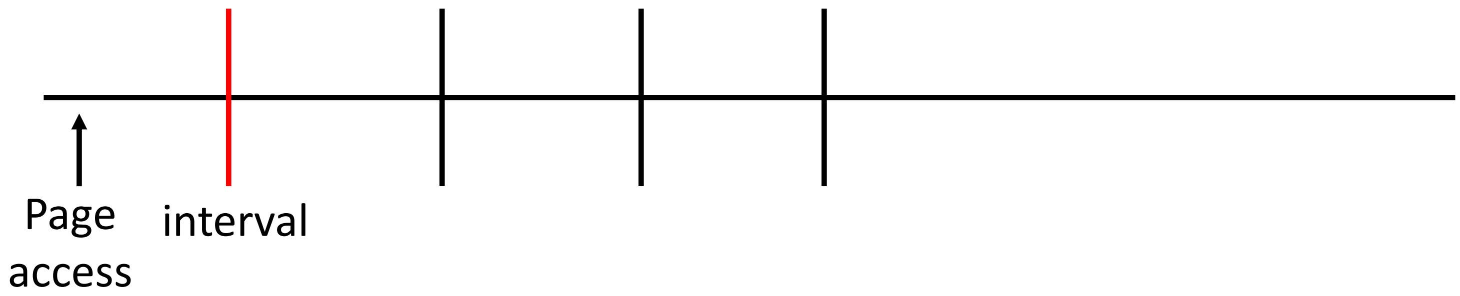
- Counter:

0	0	0	0	0	0	0	0

- Ref bit: 1

Aging Illustration

- Timeline



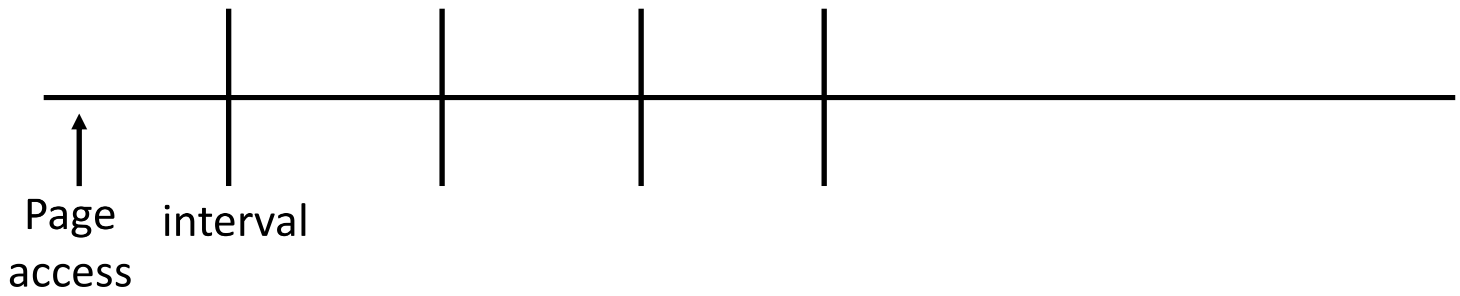
- Counter:

1	0	0	0	0	0	0	0

- Ref bit: 0

Aging Illustration

- Timeline



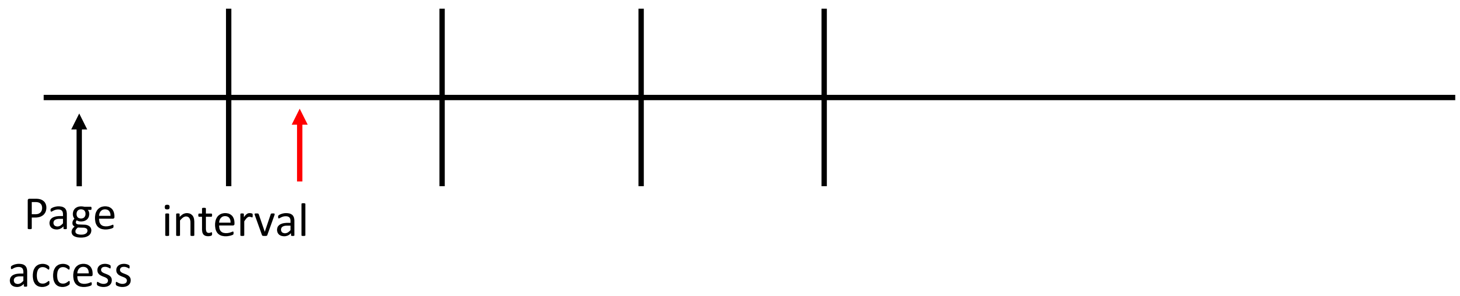
- Counter:

1	0	0	0	0	0	0	0

- Ref bit: 0

Aging Illustration

- Timeline



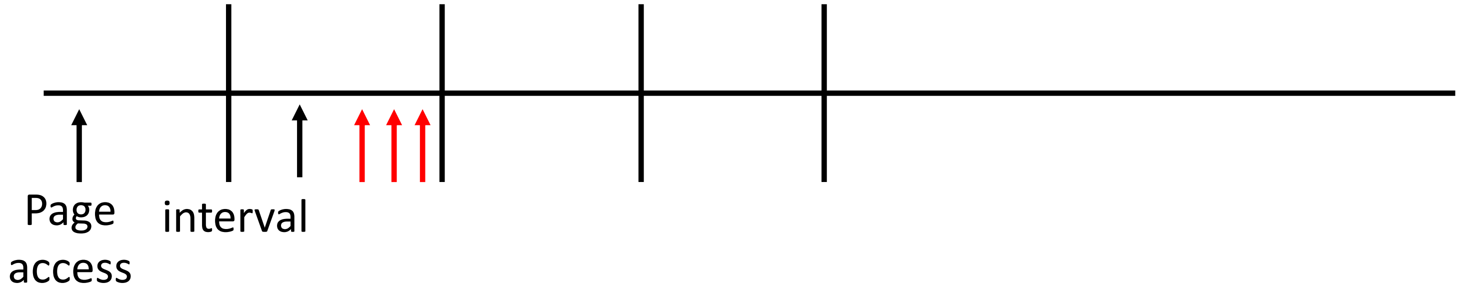
- Counter:

1	0	0	0	0	0	0	0

- Ref bit: 1

Aging Illustration

- Timeline



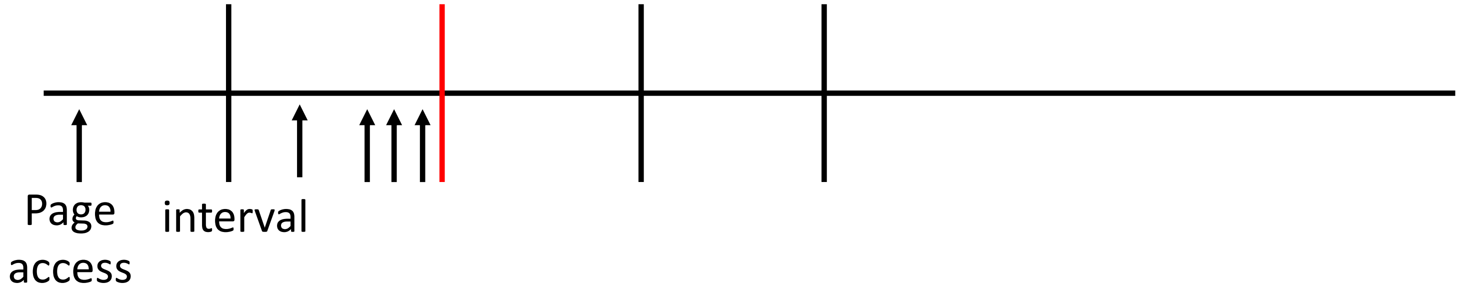
- Counter:

1	0	0	0	0	0	0	0

- Ref bit: 1

Aging Illustration

- Timeline



- Counter:

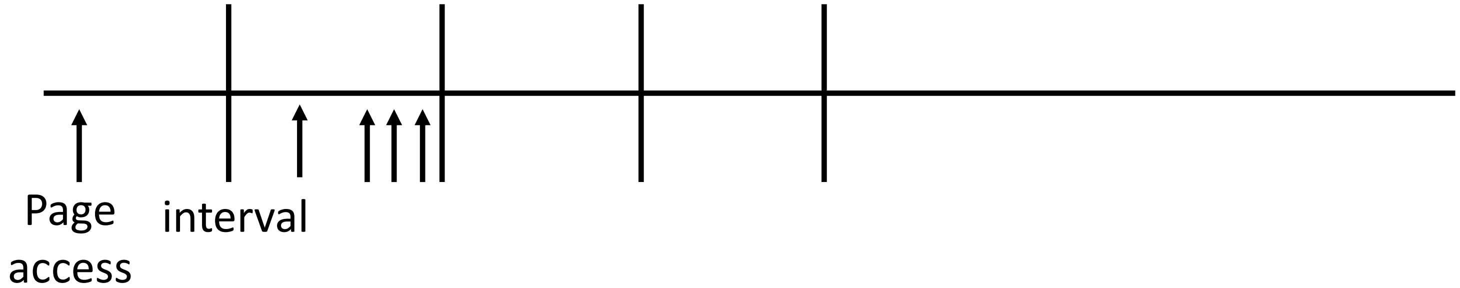
1	1	0	0	0	0	0	0

- Ref bit: 0

Same change to counter regardless of number of accesses in the interval, and when the accesses happened in the interval

Aging Illustration

- Timeline



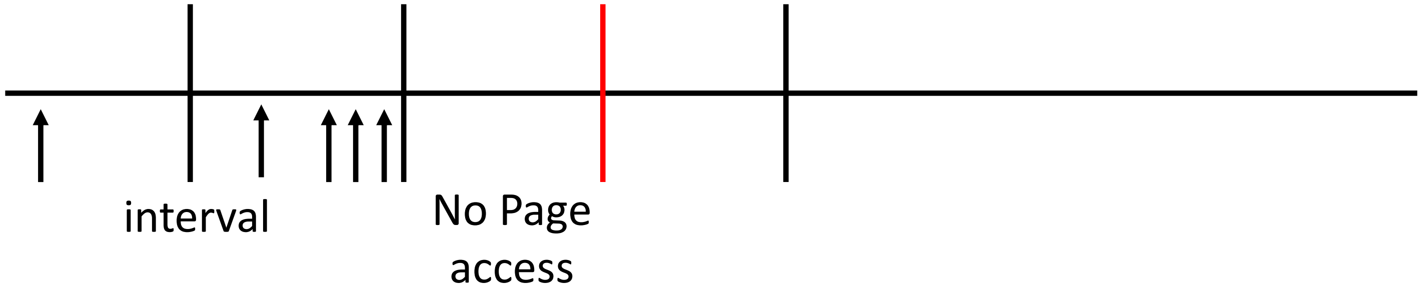
- Counter:

1	1	0	0	0	0	0	0

- Ref bit: 0

Aging Illustration

- Timeline



- Counter:

0	1	1	0	0	0	0	0

- Ref bit: 0

Aging: Analysis

- Analysis
  - Low overhead on clock tick and memory access
  - Still must search page table for entry to remove
  - Insufficient information to handle some ties
    - Only one bit information per clock cycle
    - Information past a certain clock cycle is lost

Lecture Outline

- Memory Hierarchy
- TLB
- Page Replacement: High Level
  - FIFO
  - Reference Strings
  - Beladys
- LRU
- **Thrashing**
- FIFO w/ Reference bit

Thrashing

- This is not specific to LRU, but it is easiest to demonstrate with LRU

- When the physical memory of a computer is overcommitted, causing almost constant page faults (which are slow)
  - Overcommitment most commonly happens when there are too many processes, and thus too much memory needed
  - Can also happen with a few processes, if the process needs too much memory

Thrashing: LRU Example

- Consider the following example with three page frames and LRU

	Ref str:	0	1	2	3	0	1	2	3	0	1	2	3
Most recently used		0	1	1	2	0	1	2	3	0	1	2	3
			0	1	2	3	0	1	2	3	0	1	2
LRU				0	1	2	3	0	1	2	3	0	1
Victim					0	1	2	3	0	1	2	3	0

- Page fault on every memory access ☹

Thrashing: Multiprogramming

- It is good to have more processes running, then we can have better utilization of CPU.
  - While one process waits on something, another can run
  - More on CPU Utilization later
- As we use more processes running at once, more memory is needed, can cause thrashing ☹

Lecture Outline

- Memory Hierarchy
- TLB
- Page Replacement: High Level
  - FIFO
  - Reference Strings
  - Beladys
- LRU
- Thrashing
- **FIFO w/ Reference bit**

FIFO Analysis

- Remember FIFO? The first page replacement algorithm we covered?
  - Evict the page that has been in physical memory the longest

- Analysis:
  - Low overhead. No need to do any work on each memory access, instead just need to do something when loading a new page into memory & evicting an existing page
  - Not the best at predicting which pages are used in the future ☹

- Could we modify FIFO to better suit our needs?

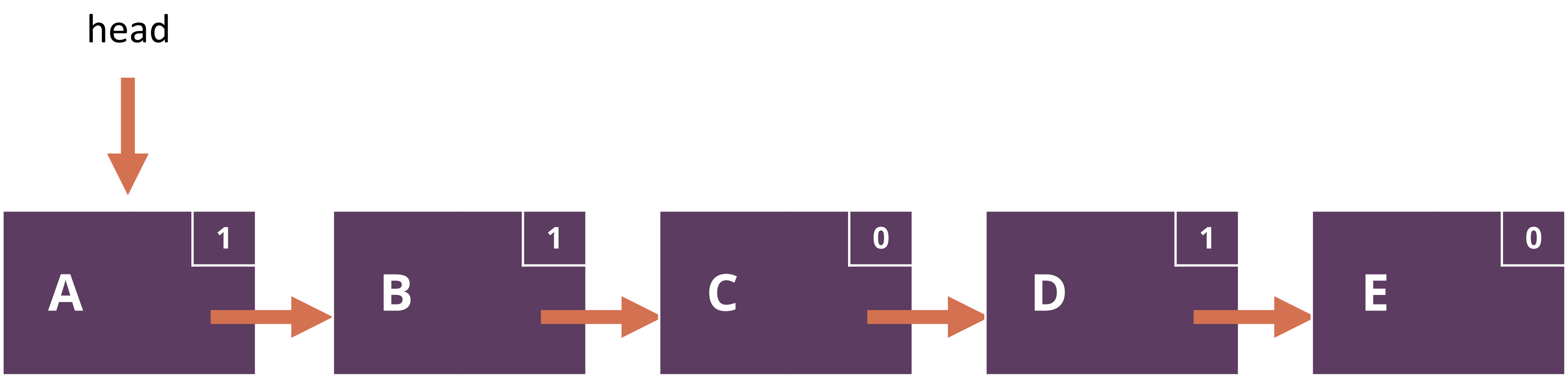
Second Chance

- Second chance algorithm is very similar to FIFO
  - Still have a FIFO queue
  - When we take the first page of the queue, instead of immediately evicting it, we instead check to see if the reference bit is 1 (was used in the last time interval)
  - If so, move it to the end of the queue
  - Repeat until we find a value that does not have the reference bit set (if all pages have reference bit as 1, then we eventually get back to the first page we looked at)

Second Chance Example

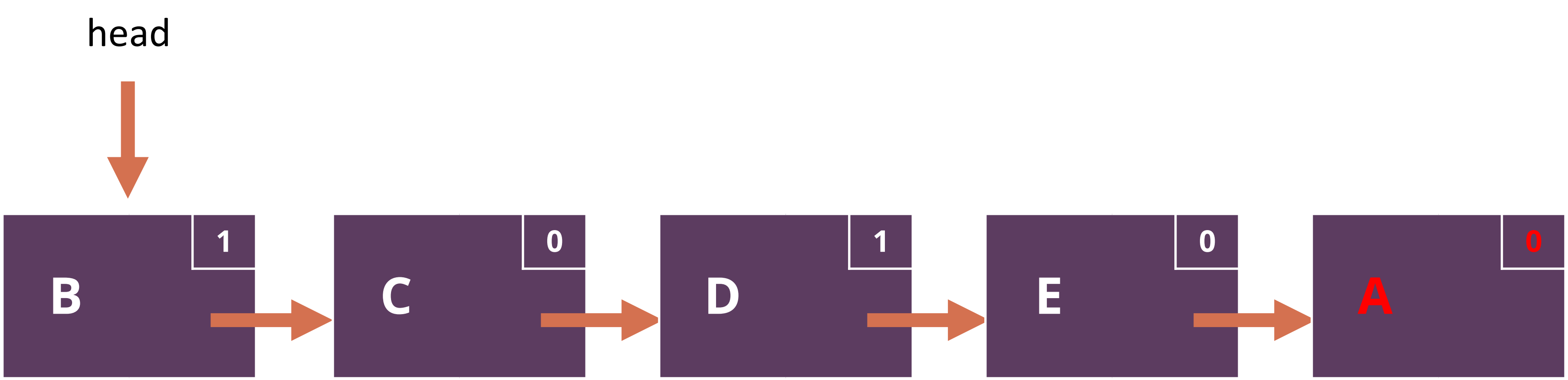
- If we need to evict a page: start at the front

- Reference bit is 1, so set to 0 and move to end



Second Chance Example

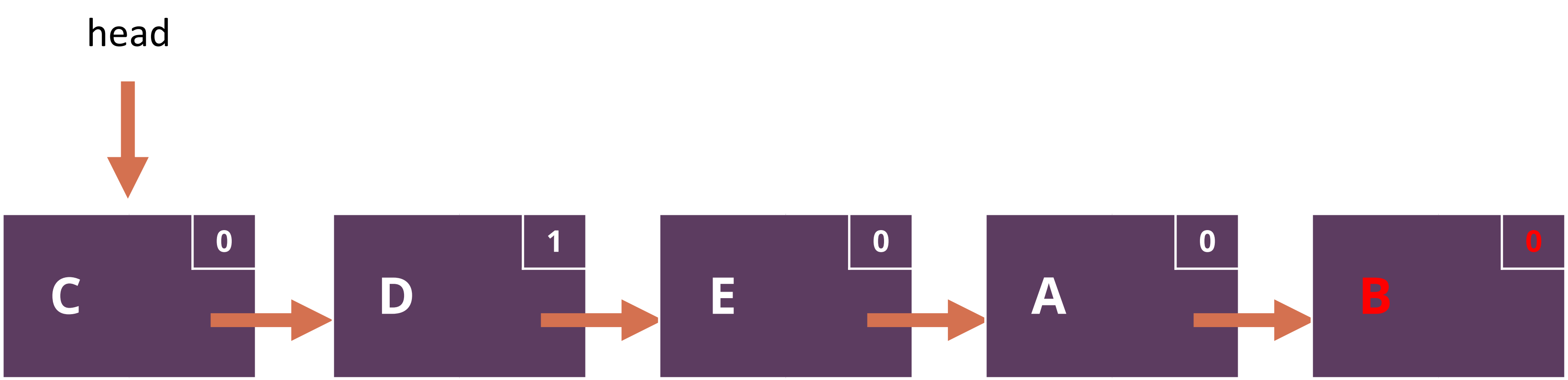
- If we need to evict a page: start at the front
- Reference bit is 1, so move to end



Second Chance Example

- If we need to evict a page: start at the front

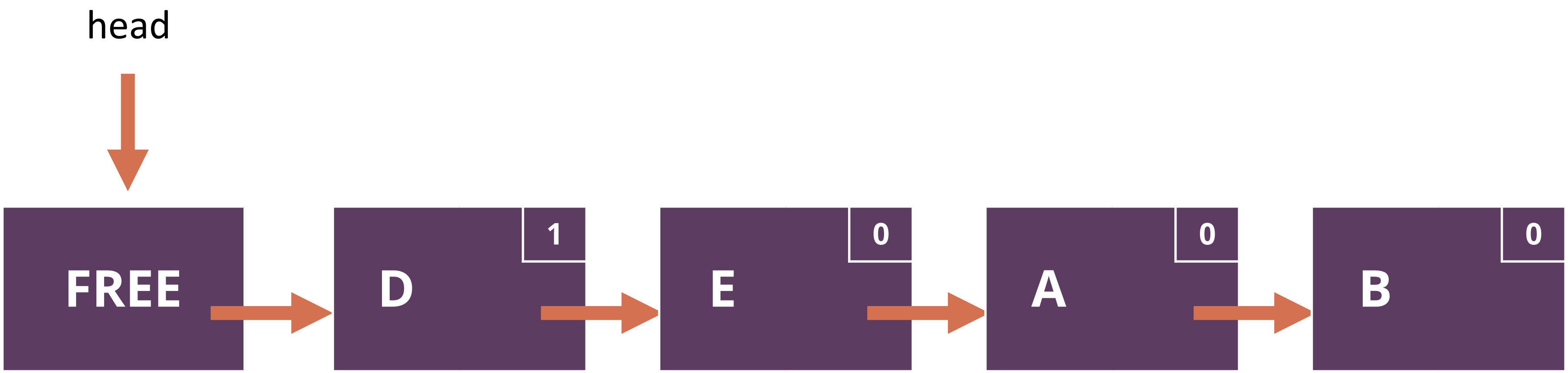
- Reference bit is 1, so move to end



Second Chance Example

- If we need to evict a page: start at the front

- Found a page with reference bit = 0, evict Page C!

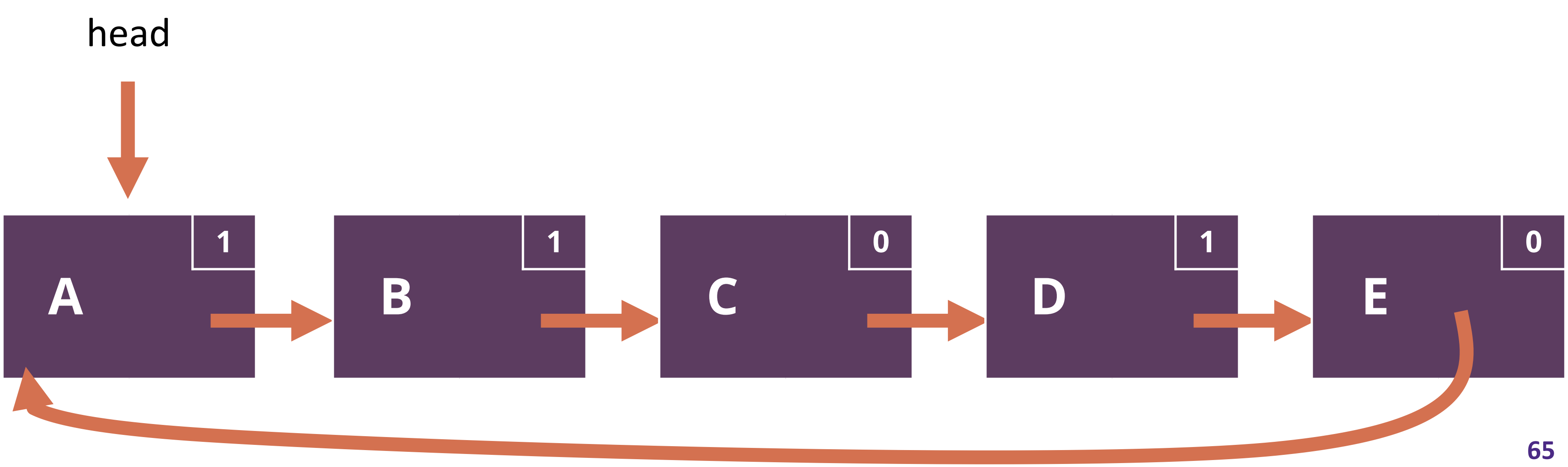


Clock

- Optimization on the second chance algorithm
- Have the queue be circular, thus the cost to moving something to the “end” is minimal

Clock Example

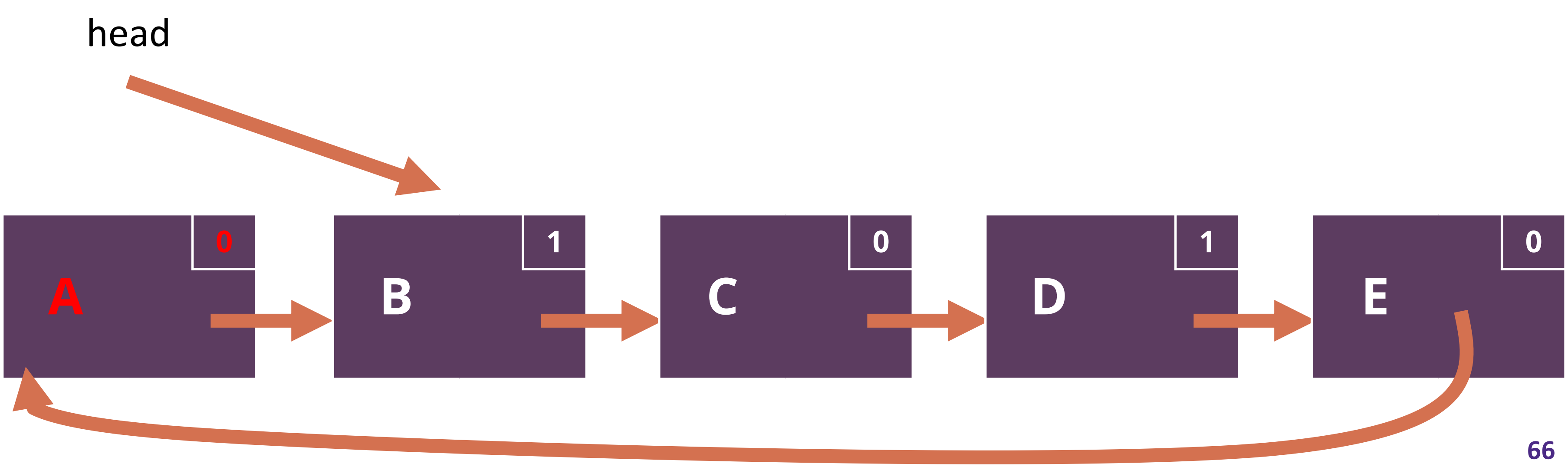
- If we need to evict a page: start at the front
- Reference bit is 1, so set to 0 and move to end



Clock Example

- If we need to evict a page: start at the front

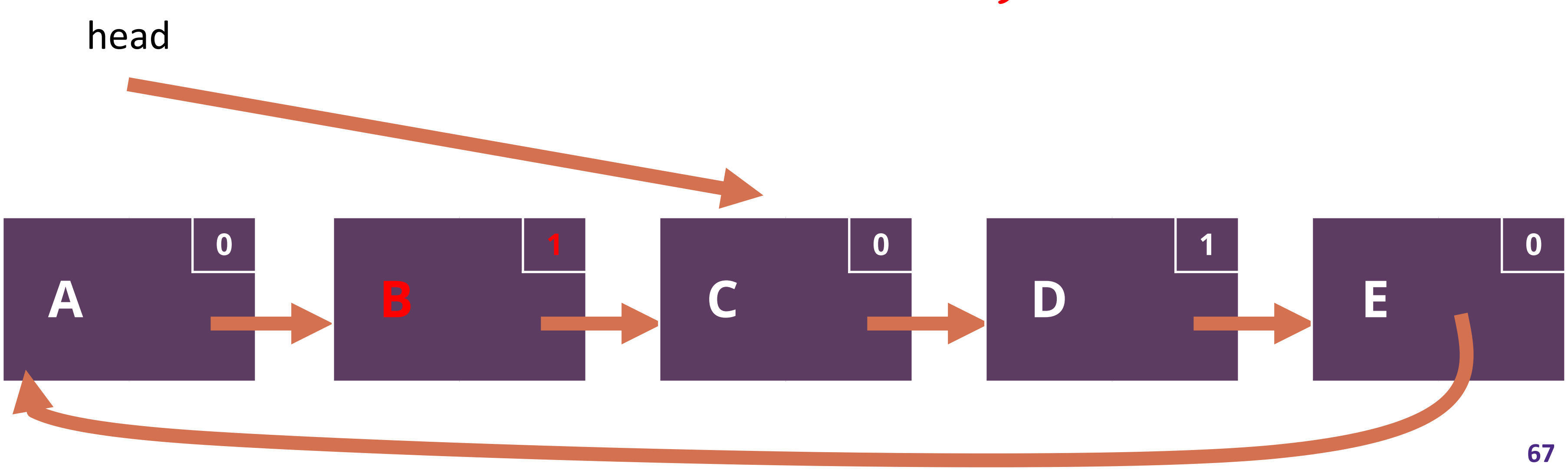
- Reference bit is 1, so set to 0 and move to end



Clock Example

- If we need to evict a page: start at the front
- Reference bit is 1, so set to 0 and move to end

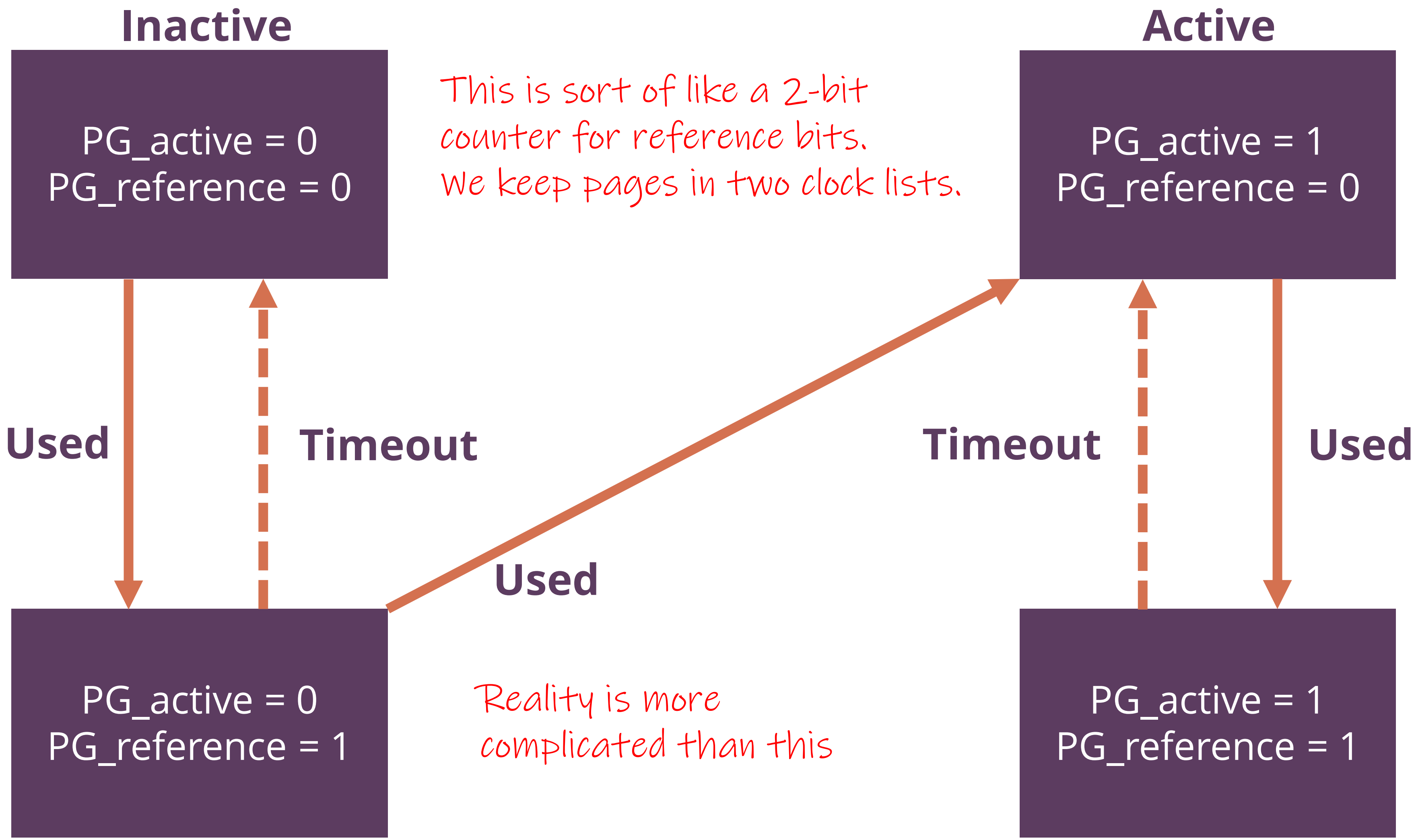
Can also be modified to prefer to evict clean pages instead of dirty pages



Linux

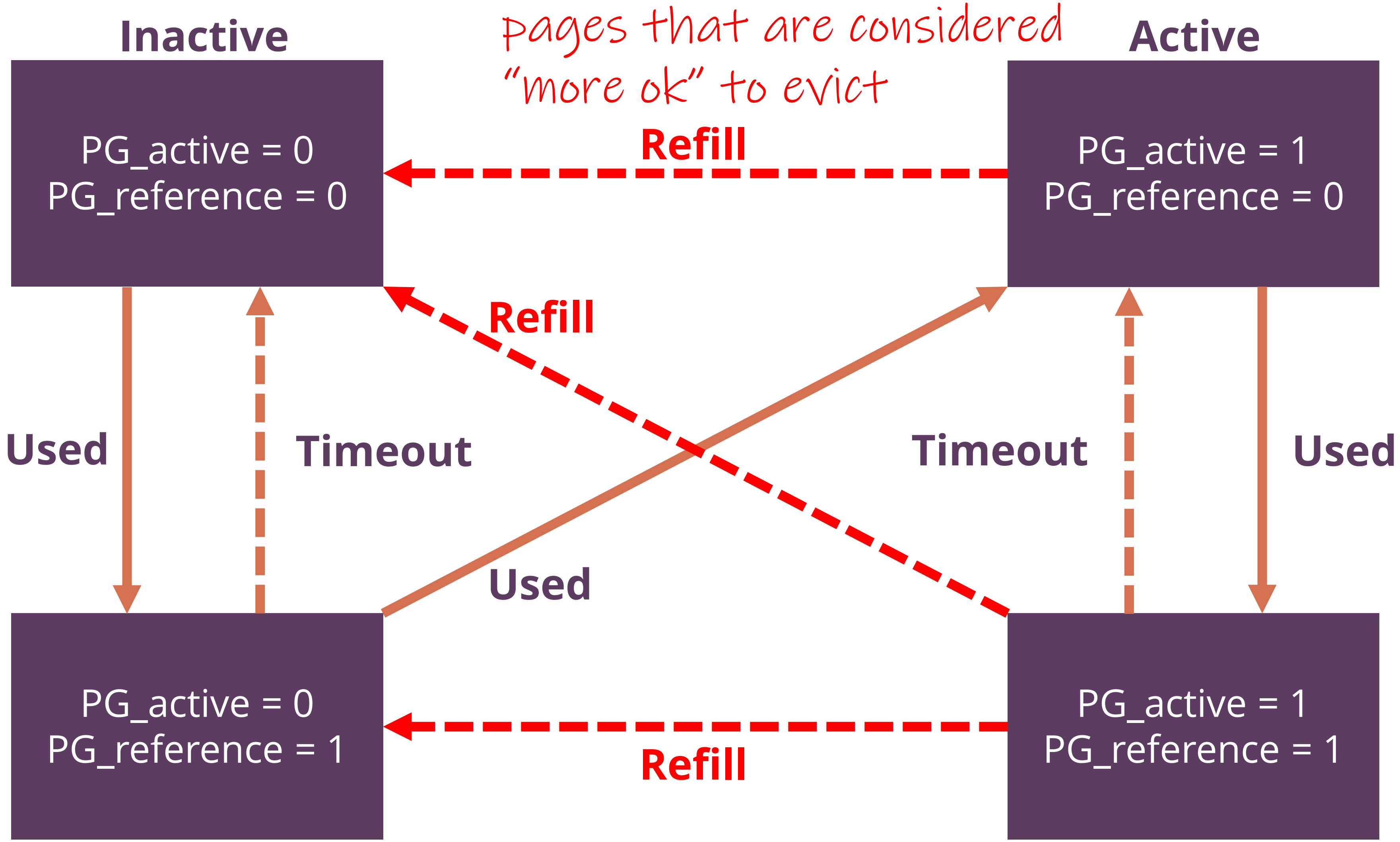
- Two Clock lists: Active and Inactive
  - Reclaim from inactive list first
  - If page has not been referenced recently, move to inactive list
  - If page is referenced:
    - Set reference flag to be true
    - Move to active list next time it is accessed
  - Two page accesses to be declared active
  - If second access does not happen, reference flag is reset periodically
- After two timeouts, move a page to inactive state

Linux diagram



Linux diagram

Linux will want to keep a good ratio of inactive to active, so that there are always some pages that are considered "more ok" to evict



Active should be ~2/3 of pages at most