Page Replacement

Computer Operating Systems, Fall 2025

Instructors: Joel Ramirez

Head TAs: Maya Huizar, Akash Kaukuntla, Vedansh Goenka, Joy Liu

TAs:

Eric Zou, Joseph Dattilo, Aniket Ghorpade, Shriya Sane, Zihao Zhou, Eric Lee, Shruti Agarwal, Yemisi Jones, Connor Cummings, Shreya Mukunthan, Alexander Mehta, Raymond Feng, Bo Sun, Steven Chang, Rania Souissi, Rashi Agrawal, Sana Manesh

pollev.com/cis5480

- What's your favorite coffee order?

Lecture Outline

- Page Replacement: High Level
  - FIFO
  - Reference Strings
  - Beladys
- LRU
- Thrashing
- FIFO w/ Reference bit

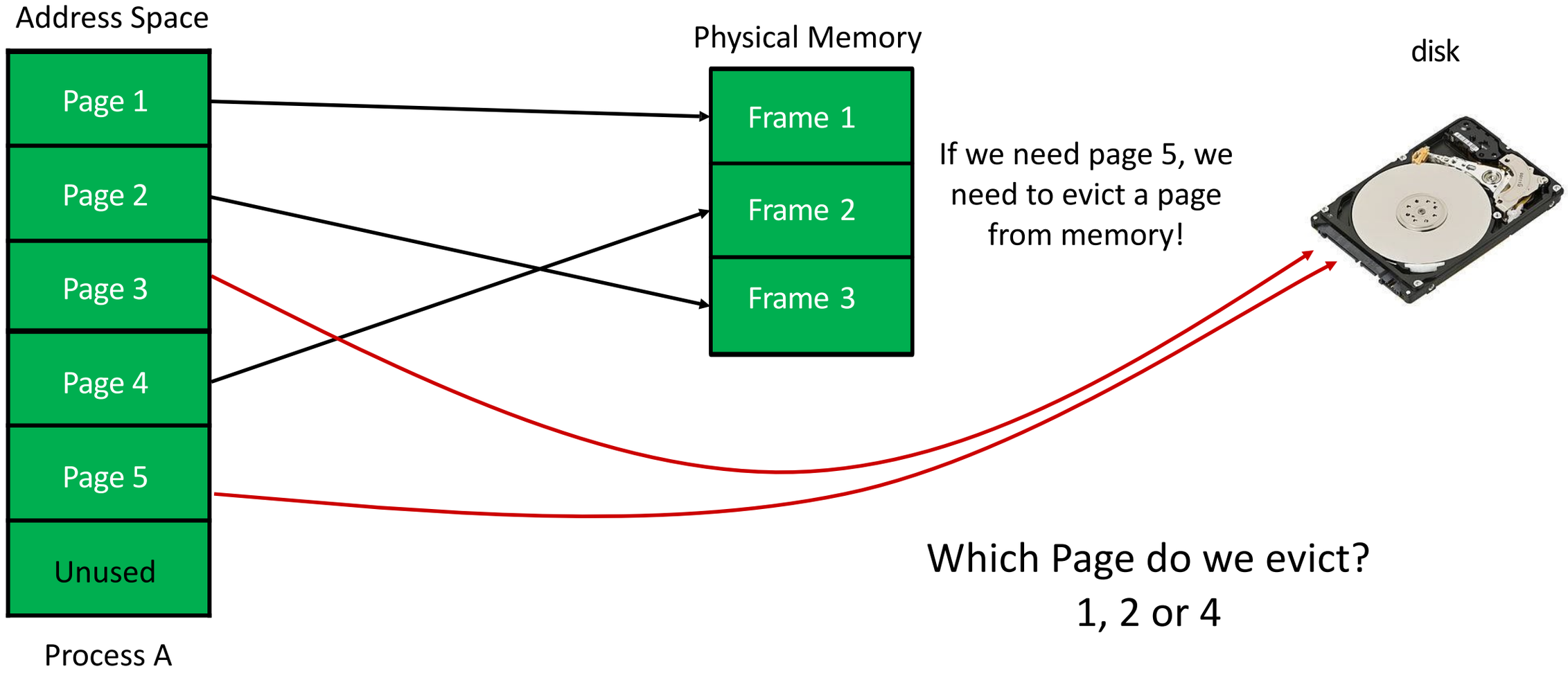
Page Replacement

- 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

Evicting a Page

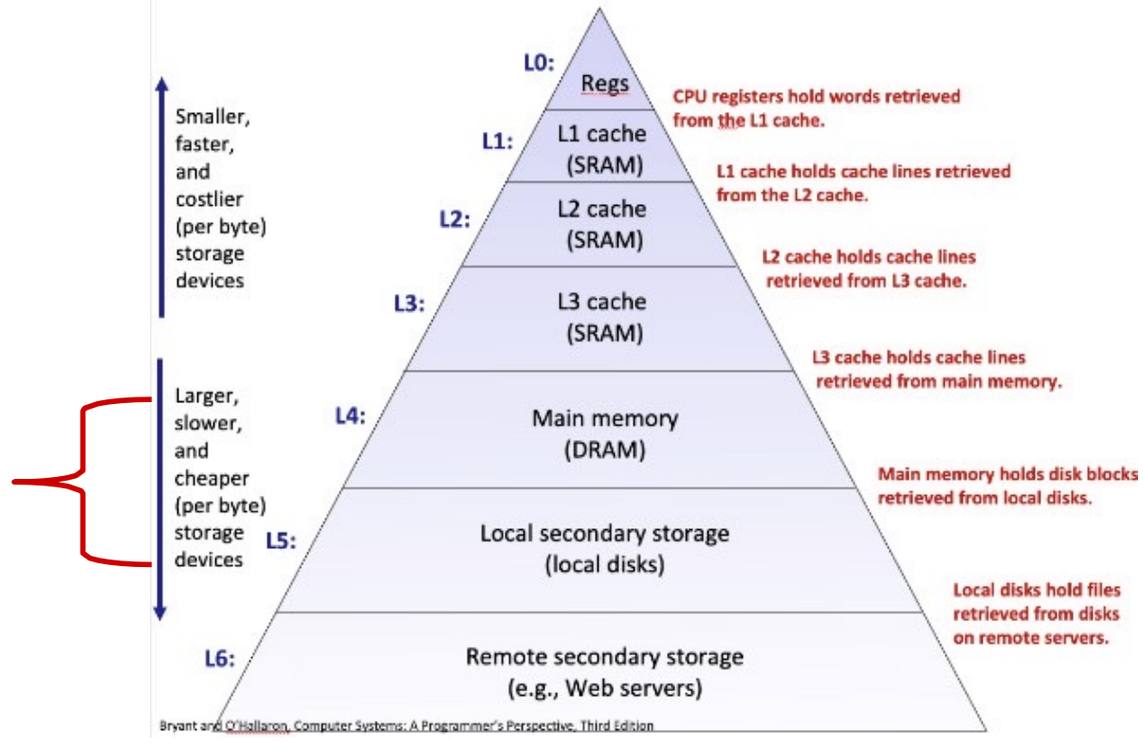
- Physical Memory is limited in size. Not All Pages can exist in memory.



Page Replacement

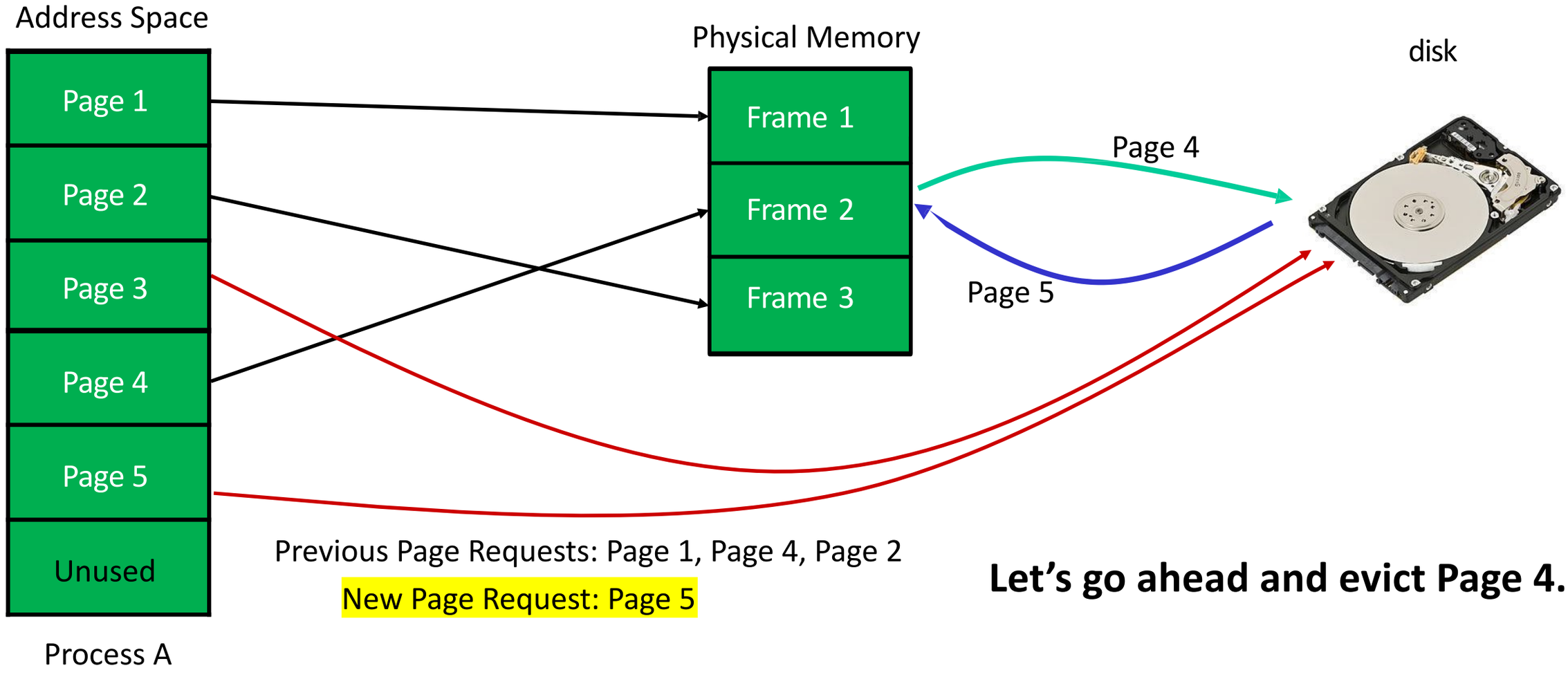
- Disk access is very very slow (relatively speaking).
  - How can we minimize disk accesses?
  - How can we try to ensure the page we evict from memory is unlikely to be
  - used again in the future?

We are at this Level:



Evicting a Page

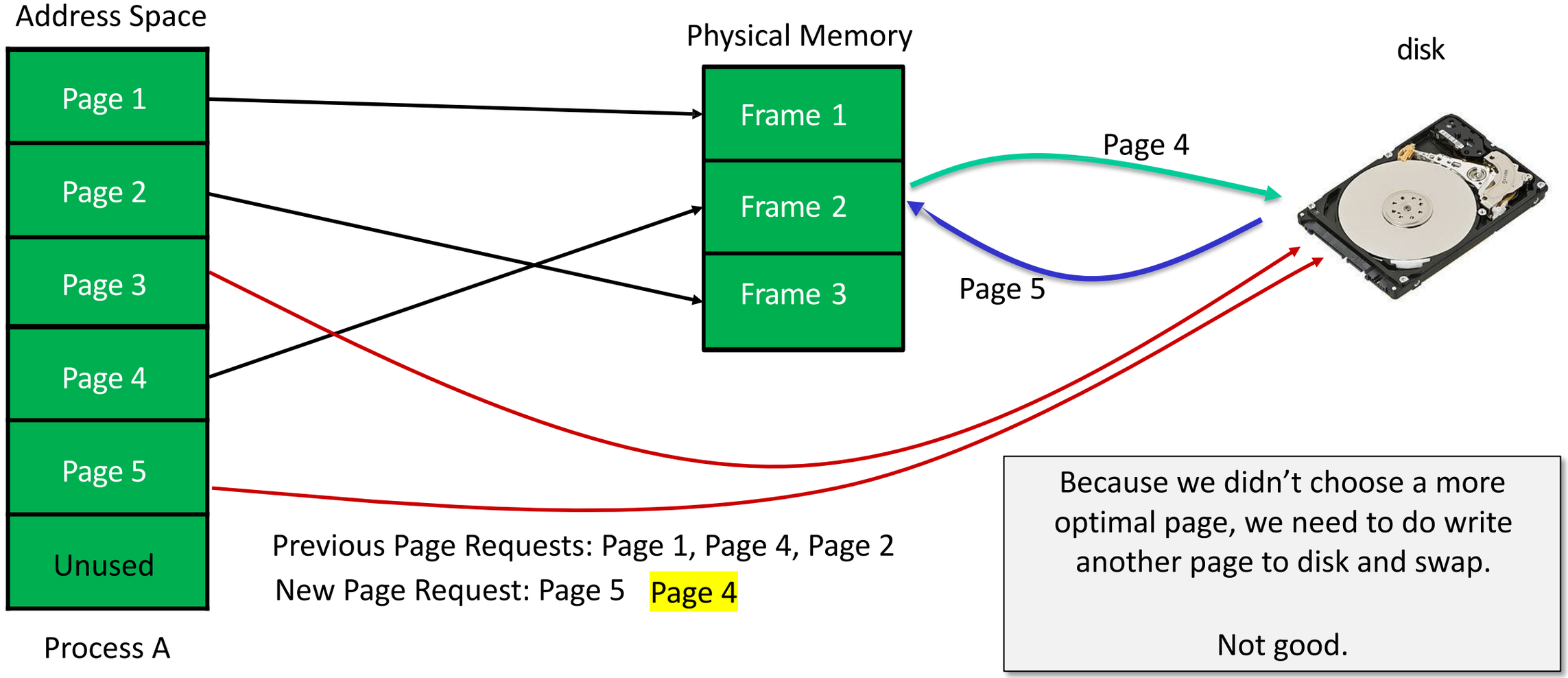
- Physical Memory is limited in size. Not All Pages can exist in memory.



Let's go ahead and evict Page 4.

Evicting a Page

- Physical Memory is limited in size. Not All Pages can exist in memory.



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 …

- Having the page access history, we can now optimize for which page to select when evicting!

Fault vs Eviction

- Page Fault
  - When a corresponding page is not in memory, we need to load the page from memory.
- Eviction
  - When there are too many pages in memory, and we need to evict one to make space for another.

Just because there is a page fault does
not mean there is an eviction.

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 can be evicted.

```c
#define MAX_PAGES 4

typedef struct page_stack {
    short *stack; //array
    short size;
} page_stack;
```

```c
short evict(page_stack *ps){
    // There's none to evict,
    if(ps->size == 0) return -1;
    ps->size--;
    short page_num = (ps->stack)[0];
    memmove(ps->stack, ps->stack + 1, ps->size);
    return page_num;
}
```

*simplified code, doesn't work for all edge cases

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 can be evicted.

```
void add(page_stack *ps, short page_num){
    for(int i = 0; i < ps->size; i++){
        if(ps->stack[i] == page_num){
                return; // In Mem.
                // No page fault
        }
    }
    if(ps->size == MAX_PAGES)
        evict(ps); //page eviction!
    //page fault, bring into memory
    (ps->stack)[ps->size] = page_num;
    ps->size++; //increment size
}
```

```
short evict(page_stack *ps){
    // There's none to evict
    if(ps->size == 0) return -1;
    ps->size--;
    short page_num = (ps->stack)[0]; // write to disk
    memmove(ps->stack, ps->stack + 1, ps->size);
    return page_num;
}
```

*simplified code, doesn't work for all edge cases

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

```
short page_str[] = {4, 1, 1, 2, 3, 4, 5};

page_stack ps = {0}; // No pages in memory at start.

ps.stack = (short *) malloc(MAX_PAGES * sizeof(short));

for(short x: page_str){ // for x in page_str
        add(&ps, x);
}
```

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
  - For those who like tables :)

	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/cis5480

- Given the following reference string, how many page faults (not evictions) occur when using a FIFO algorithm given no pages are in memory at the start.

1 2 3 4 1 2 5 1 2 3 4 5

```
#define MAX_PAGES 3

typedef struct page_stack {
    short *stack; //array
    short size;
} page_stack;
```

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

Poll Everywhere

pollev.com/cis5480

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

“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/cis5480

- 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

```
#define MAX_PAGES 3

typedef struct page_stack {
    short *stack; //array
    short size;
} page_stack;
```

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

Poll Everywhere

pollev.com/cis5480

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

Poll Everywhere

pollev.com/cis5480

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

Bélády's Anomaly

- Sometimes increasing the number of page frames in the data structure results in an increase in the number of page faults :/
- This behavior is something that we want to avoid/minimize the possibility of.
- Some algorithms avoid this anomaly (LRU, LIFO, etc.)

Lecture Outline

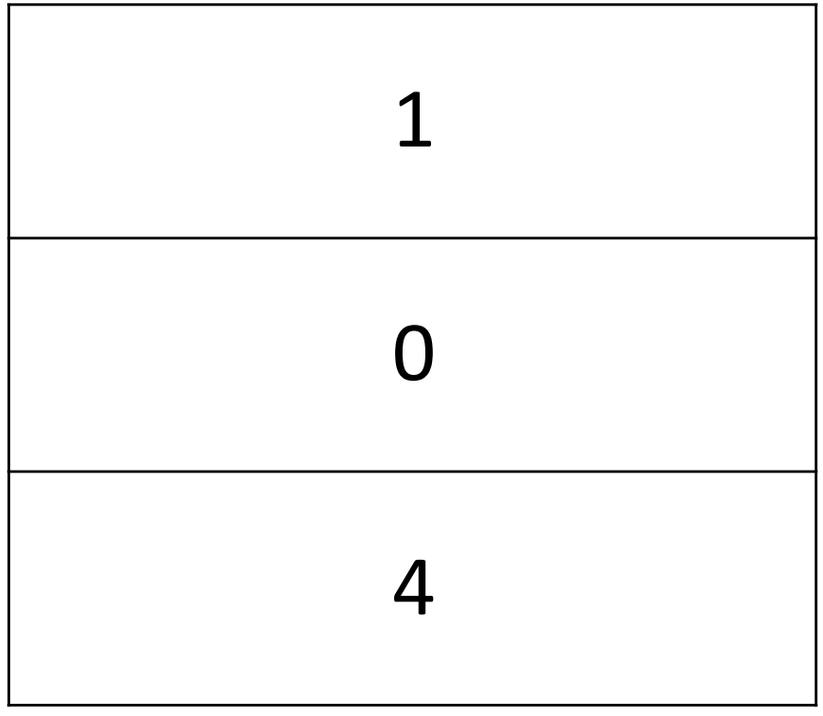
- Page Replacement: High Level
  - FIFO
  - Reference Strings
  - Beladys
- LRU
- Thrashing
- FIFO w/ Reference bit

LRU (Least Recently Used)

- Assumption:
  - If a page is used recently, it is likely to be used again in the future
- Use prior knowledge to predict the future (update posterior)
- Replace the page that has had the longest time since it was last used
- Sorta Reminiscent of a Priority Queue, where smaller time since last access indicates lower priority of eviction. (But too complicated)

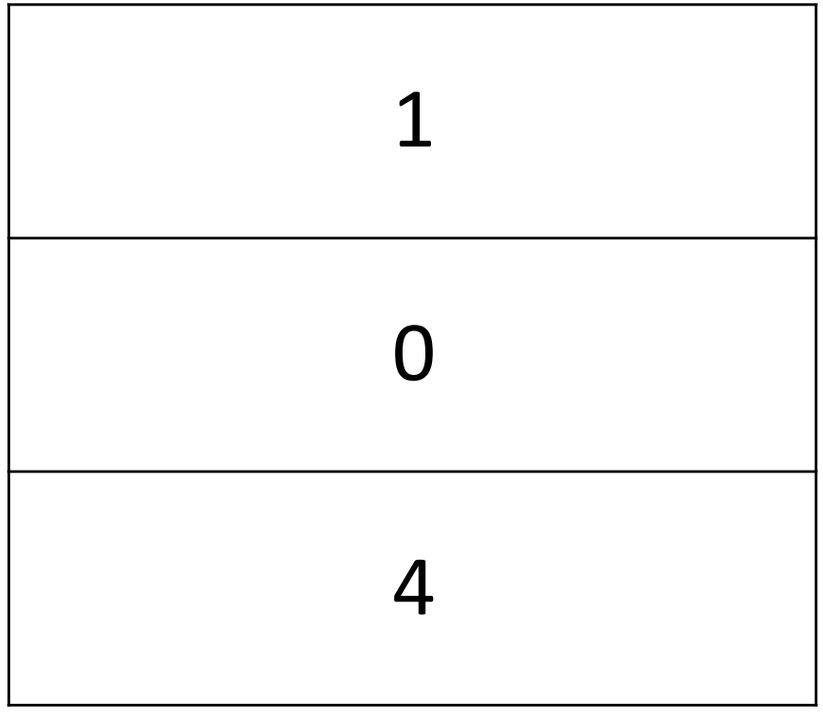
Small Example: 3 Pages of Space

4, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3



Small Example: 3 Pages of Space

4, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3

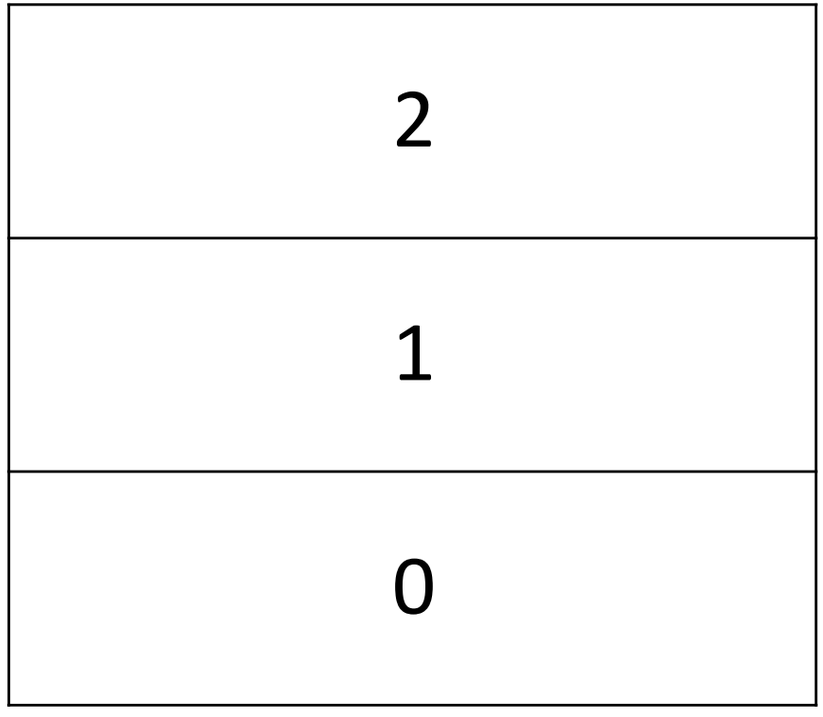


To Evict

To make space for Page 2 – we need to evict page 4.

Small Example: 3 Pages of Space

4, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3

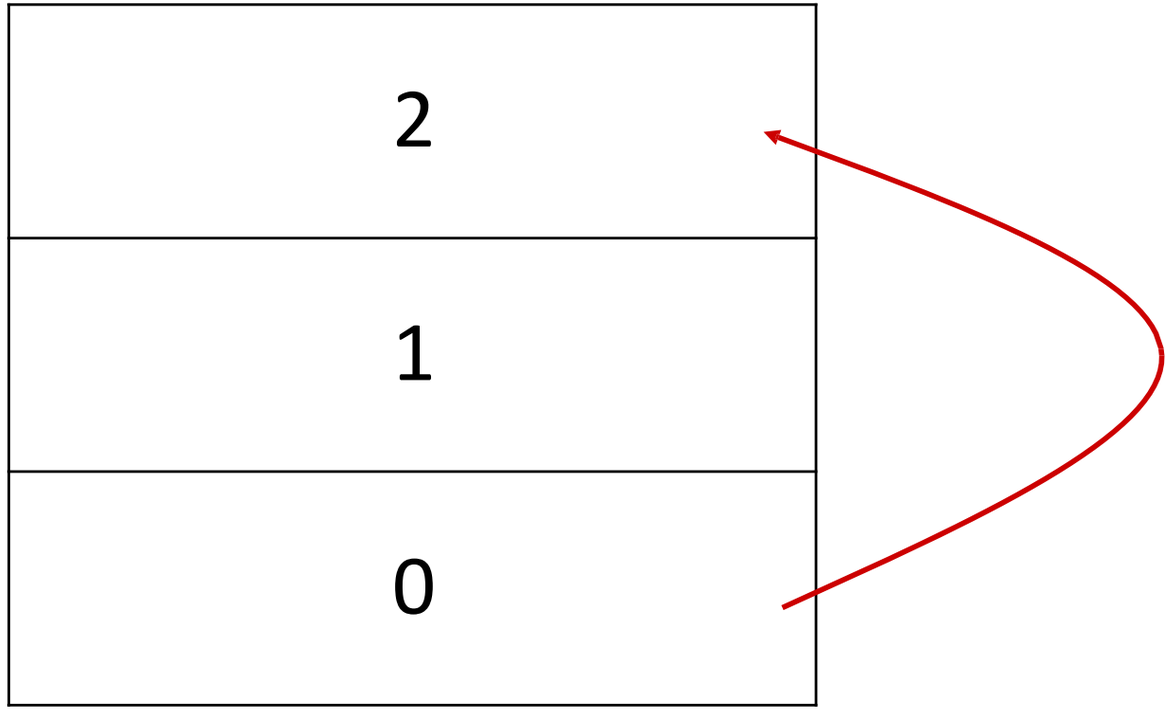


To Evict

To make space for Page 2 – we need to evict page 4.

Small Example: 3 Pages of Space

4, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3

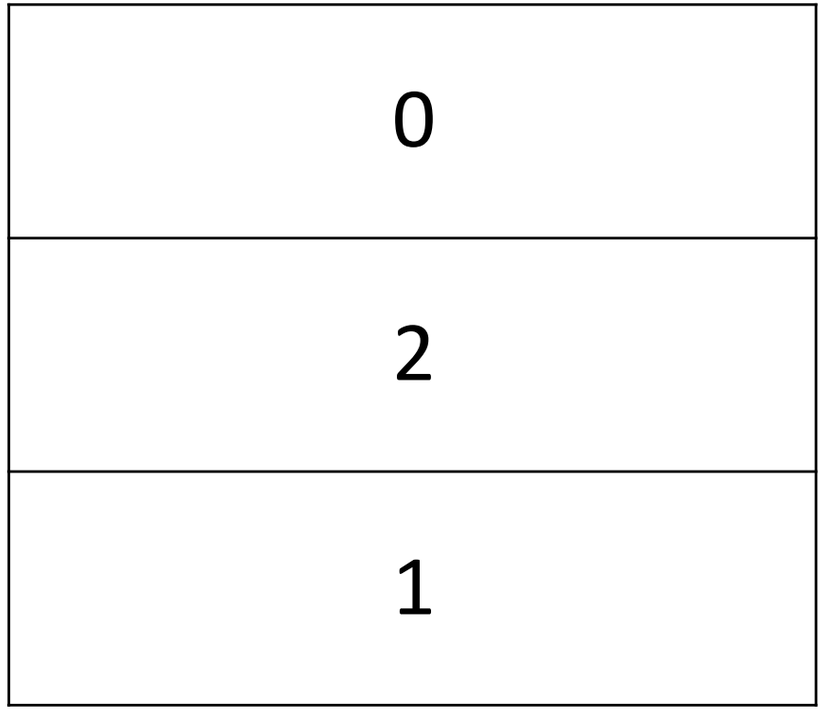


To Evict

As we access 0 again,
we move to the top of the stack

Small Example: 3 Pages of Space

4, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3



Observation:
The ‘order’ of the values when using the LRU is always a (non-contiguous) subsequence of page accesses.

Lots of ways to reason about this…try to find a way that makes more sense to you.

Poll Everywhere

pollev.com/cis5480

- Now, using the same Reference String with LRU, let's try to fill this table out…
- If we use 4 frames, how many page faults will there be?

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

LRU Implementation

- Couple of 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
    - Should you weigh time of access more when it's more recent? (e.g. 3,3,3,3,3,3,3,3,3,1,1)
  - 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

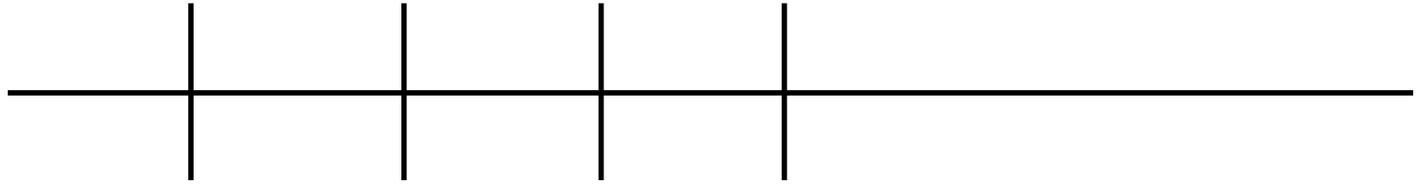
LRU Approximation: Aging

- Each page gets an 8-bit “counter”.
- On clock interval and for every page:
  - shift the counter to the right by 1 bit (>> 1)
  - write the reference bit into the MSB of the counter.
  - Current reference bit is reset to 0

- If we read the counter as an unsigned integer, then a larger value means the counter was accessed more recently
  - Right shifting allows us to take into consideration time since the last access as we
  - essentially divide the value by 2 if there were no accesses.

Aging Illustration

- Timeline



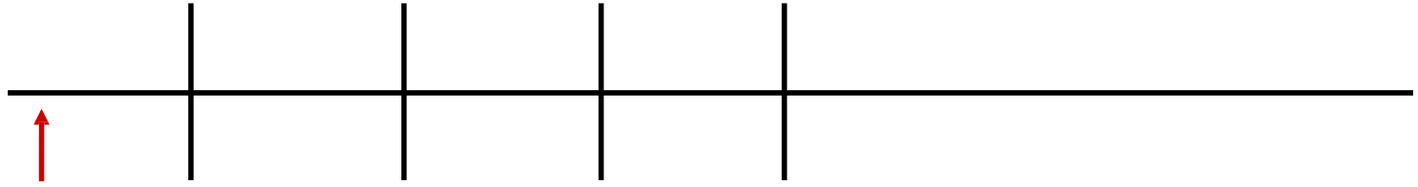
- Counter:

0	0	0	0	0	0	0	0

- Ref Bit: 0

Aging Illustration

- **Timeline**



page access

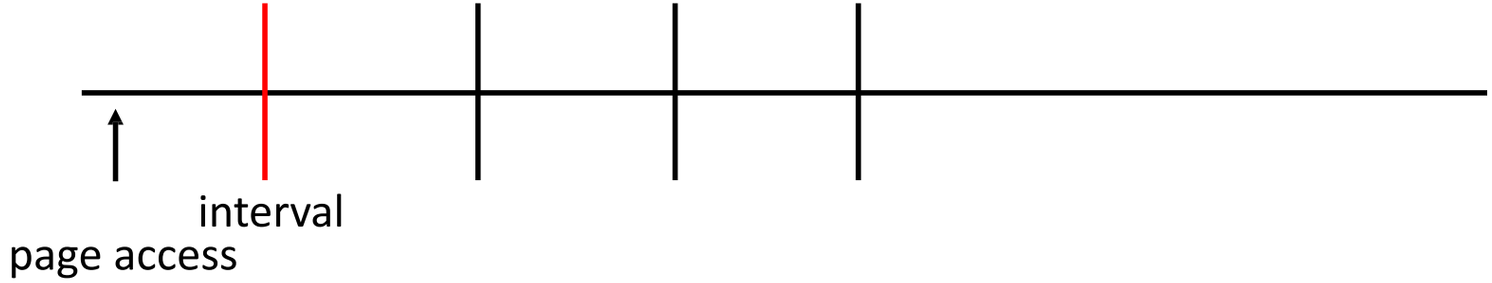
- **Counter:**

0	0	0	0	0	0	0	0

- **Ref Bit: 1**

Aging Illustration

- **Timeline**



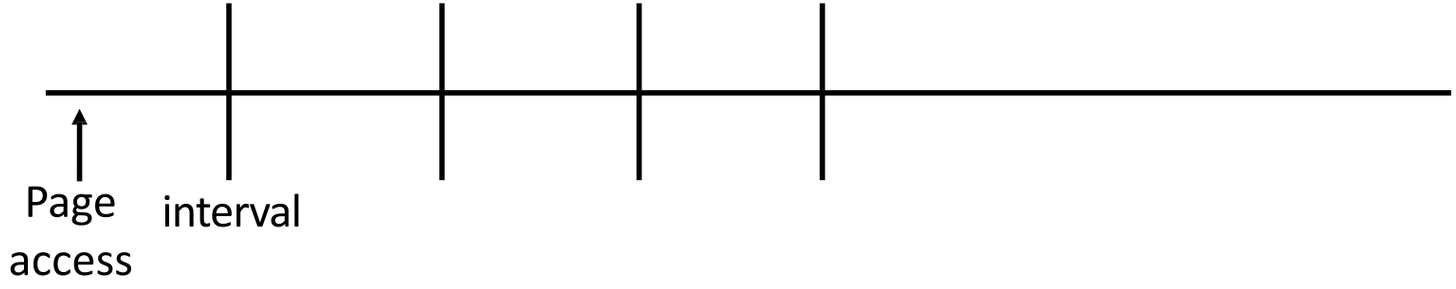
- **Counter:**

1	0	0	0	0	0	0	0

- **Ref Bit: 0**

Aging Illustration

- **Timeline**



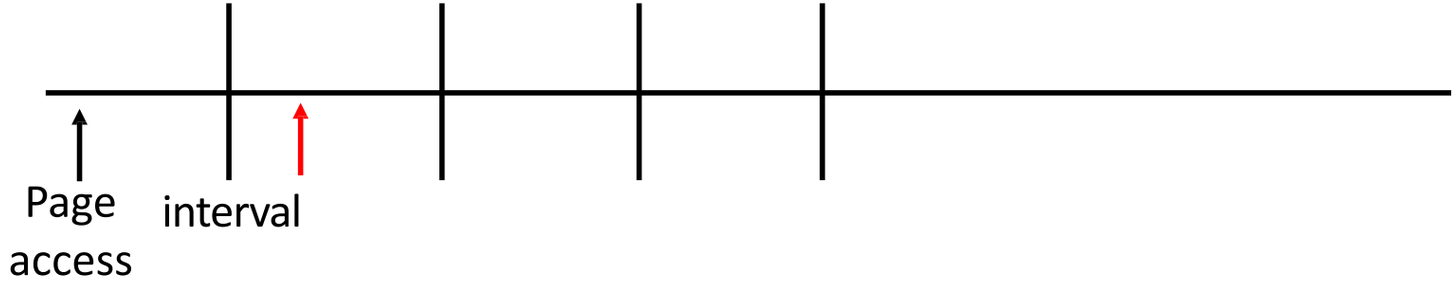
- **Counter:**

1	0	0	0	0	0	0	0

- **Ref Bit: 0**

Aging Illustration

- **Timeline**



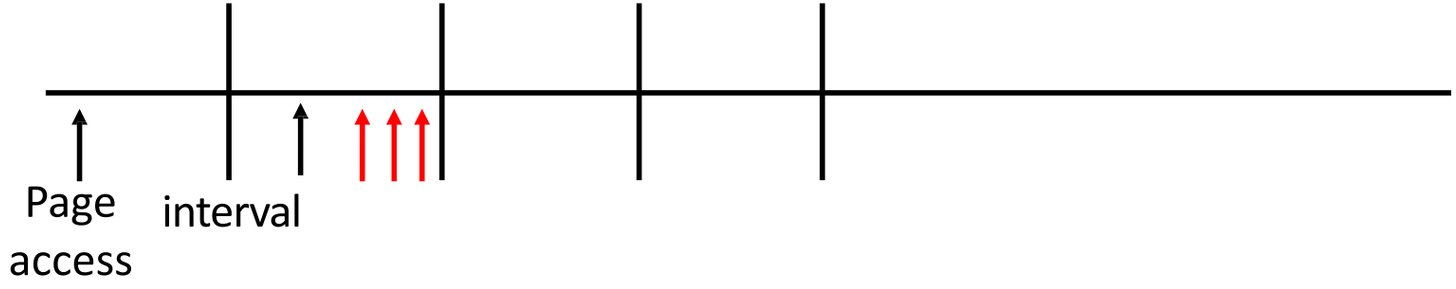
- **Counter:**

1	0	0	0	0	0	0	0

- **Ref Bit: 1**

Aging Illustration

- **Timeline**



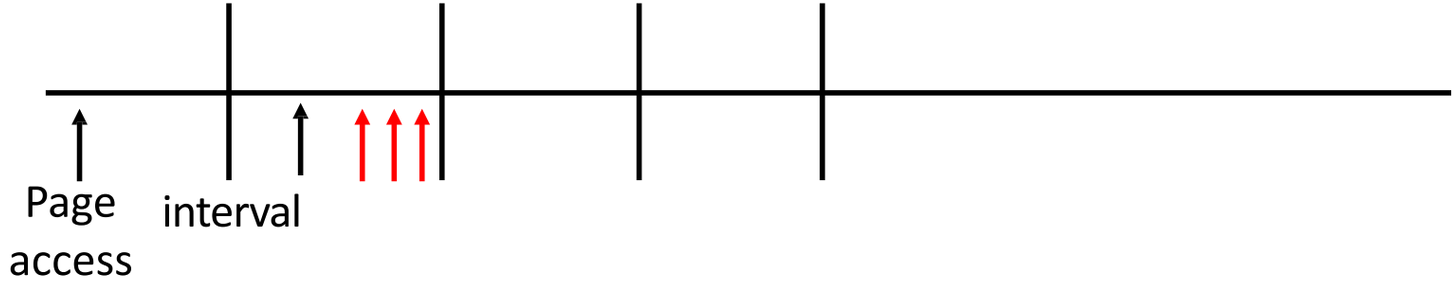
- **Counter:**

1	0	0	0	0	0	0	0

- **Ref Bit: 1**

Aging Illustration

- **Timeline**



- **Counter:**

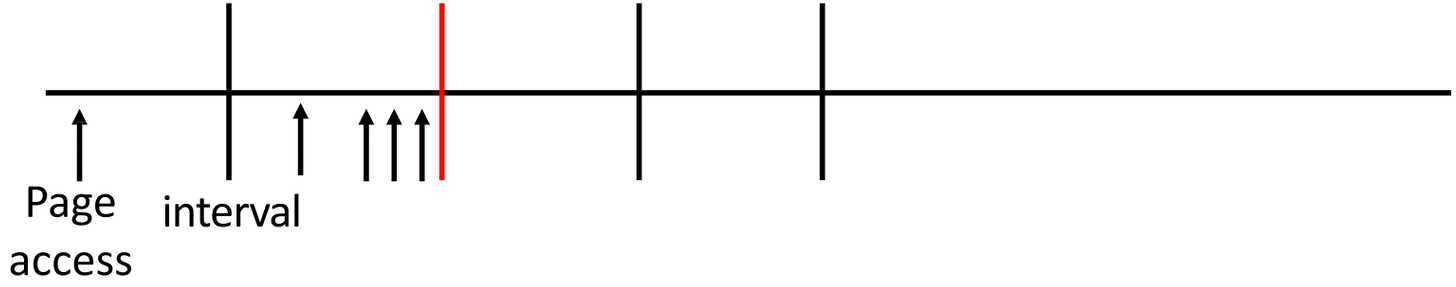
1	0	0	0	0	0	0	0

- **Ref Bit: 1**

```
counter = (uint8_t) counter >> 1
counter = counter | (ref << 7)
```

Aging Illustration

- **Timeline**



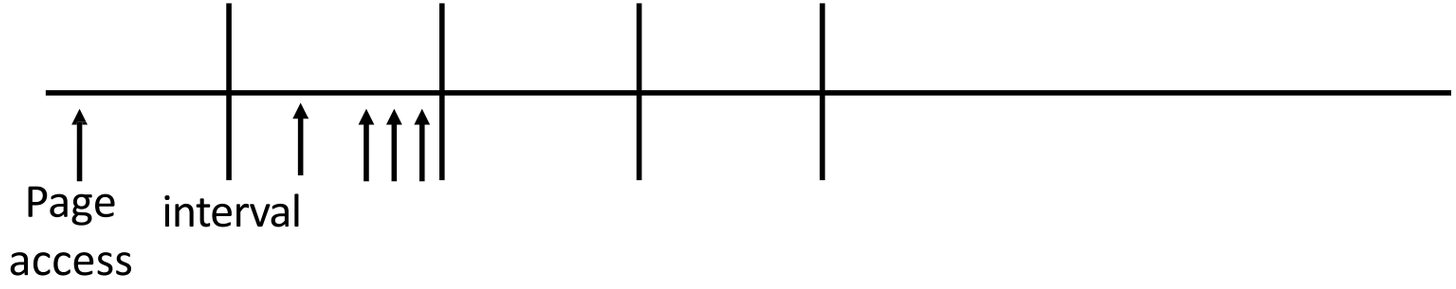
- **Counter:**

1	1	0	0	0	0	0	0

- **Ref Bit: 0**

Aging Illustration

- **Timeline**



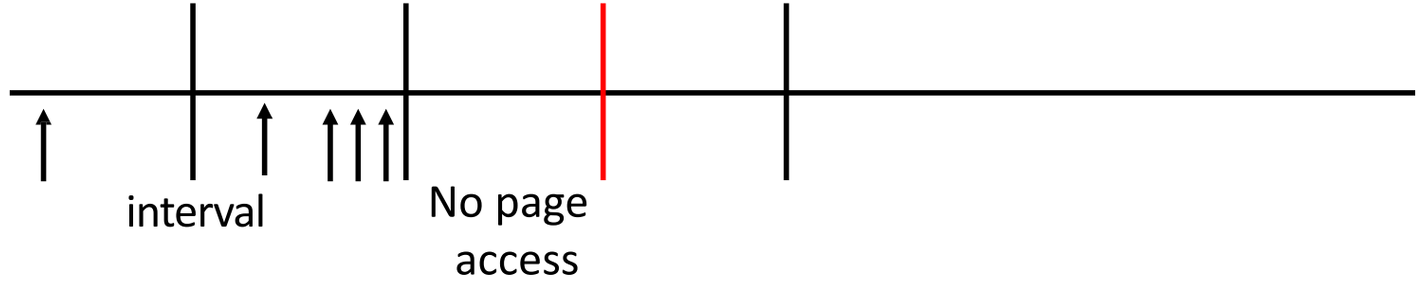
- **Counter:**

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/update
  - Insufficient information to handle some ties
    - Only one bit information per clock cycle
    - Information past a certain clock cycle is lost

Lecture Outline

- 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

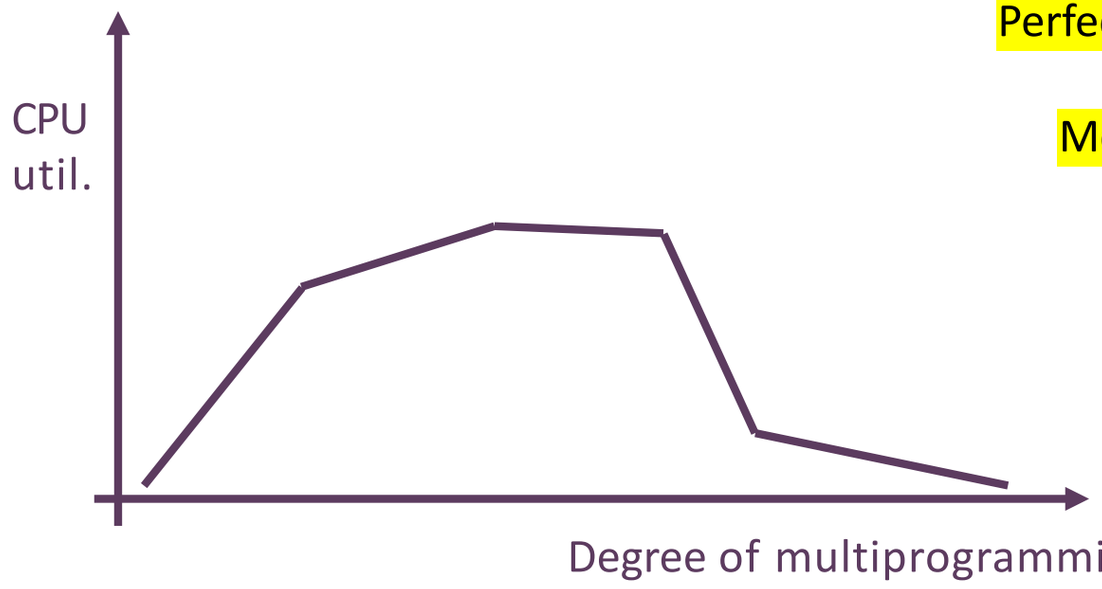
Thrashing

- Consider the following example with three page frames and LRU

LRU	Ref str:	0	1	2	3	0	1	2	3	0	1	2	3
Recent		0	1	2	3	0	1	2	3	0	1	2	3
			0	1	2	3	0	1	2	3	0	1	2
To Evict				0	1	2	3	0	1	2	3	0	1
Evicted					0	1	2	3	0	1	2	3	0

Thrashing

- 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 :/



Perfect multithreading/multiprocessing is a balance between Memory Speed/Size vs CPU Speed

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?

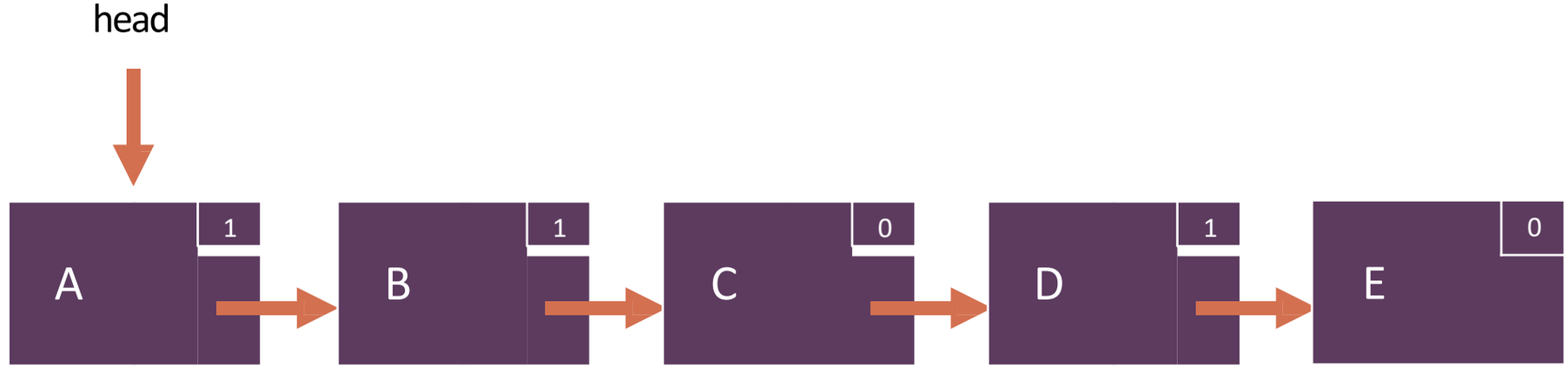
Second Chance

- 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

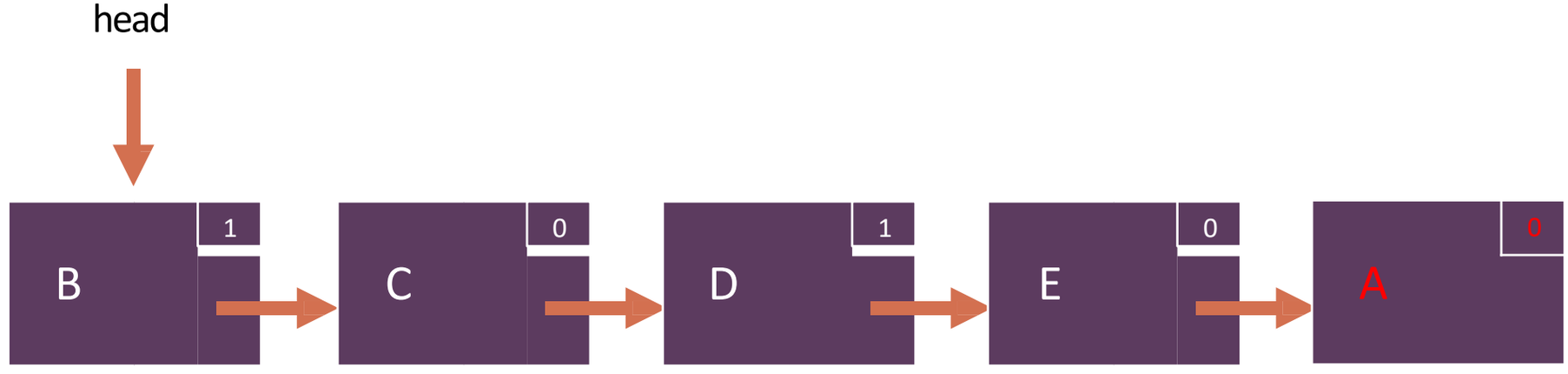
- If we need to evict a page: start at the front

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



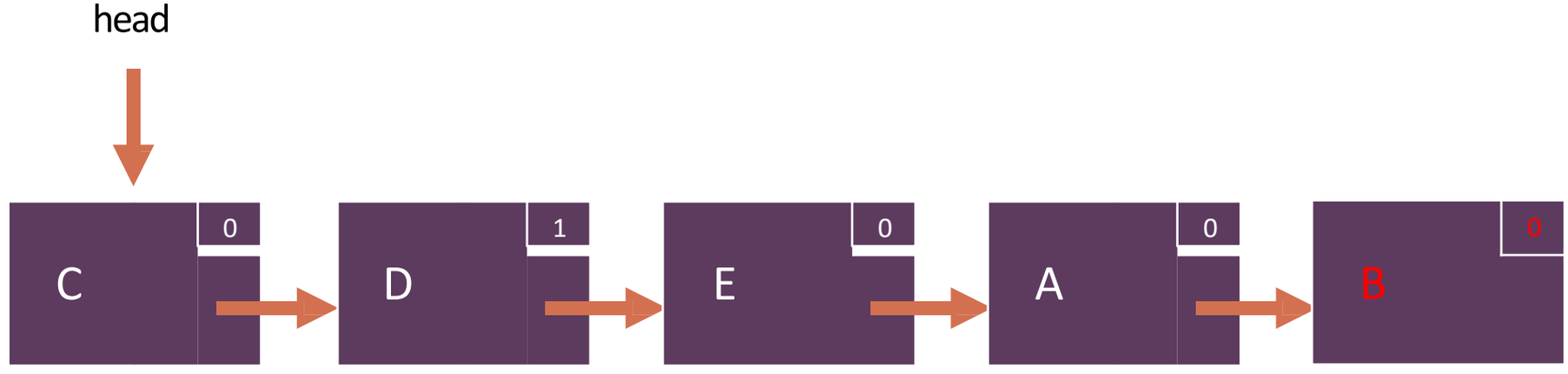
Second Chance Example

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



Second Chance Example

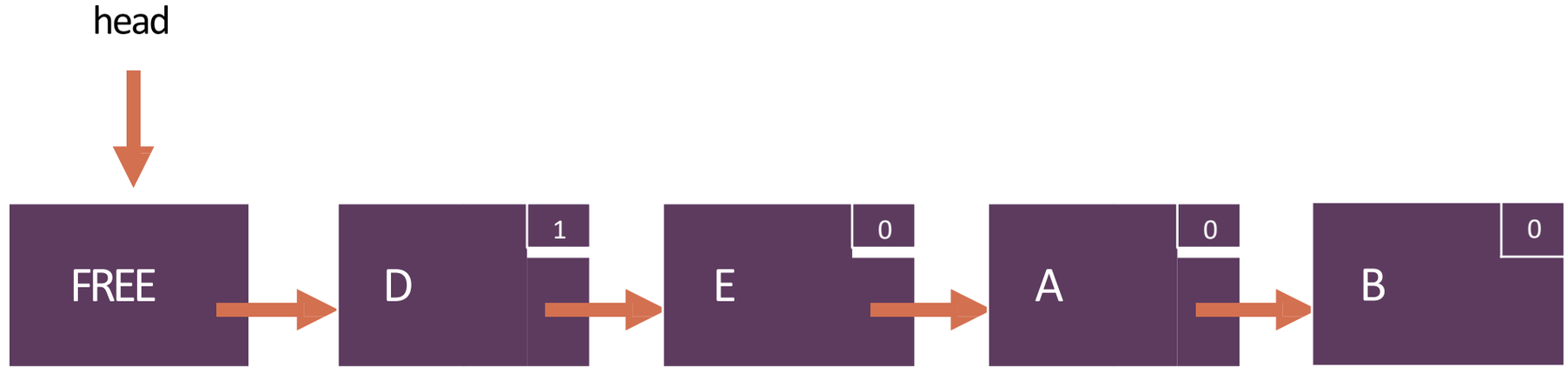
- If we need to evict a page: start at the front
- Reference bit is 1, so move to end



Second Chance Example

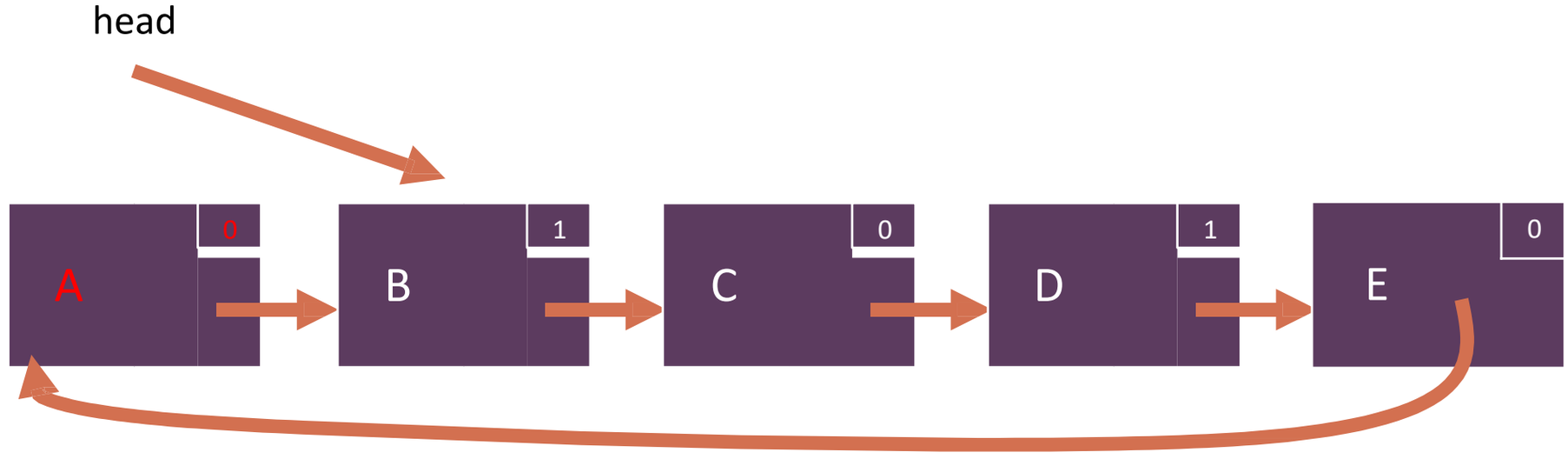
- If we need to evict a page: start at the front

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



Second Chance Clock

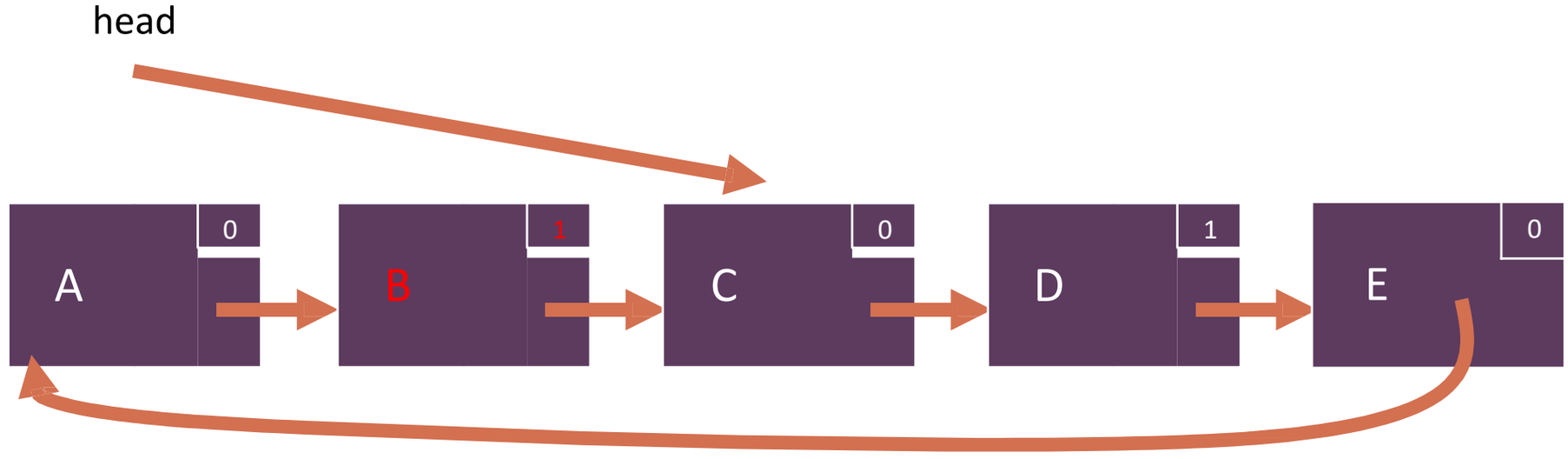
- Optimization on the second chance algorithm
- Have the queue be circular, thus the cost to moving something to the “end” is minimal



Second Chance Clock

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

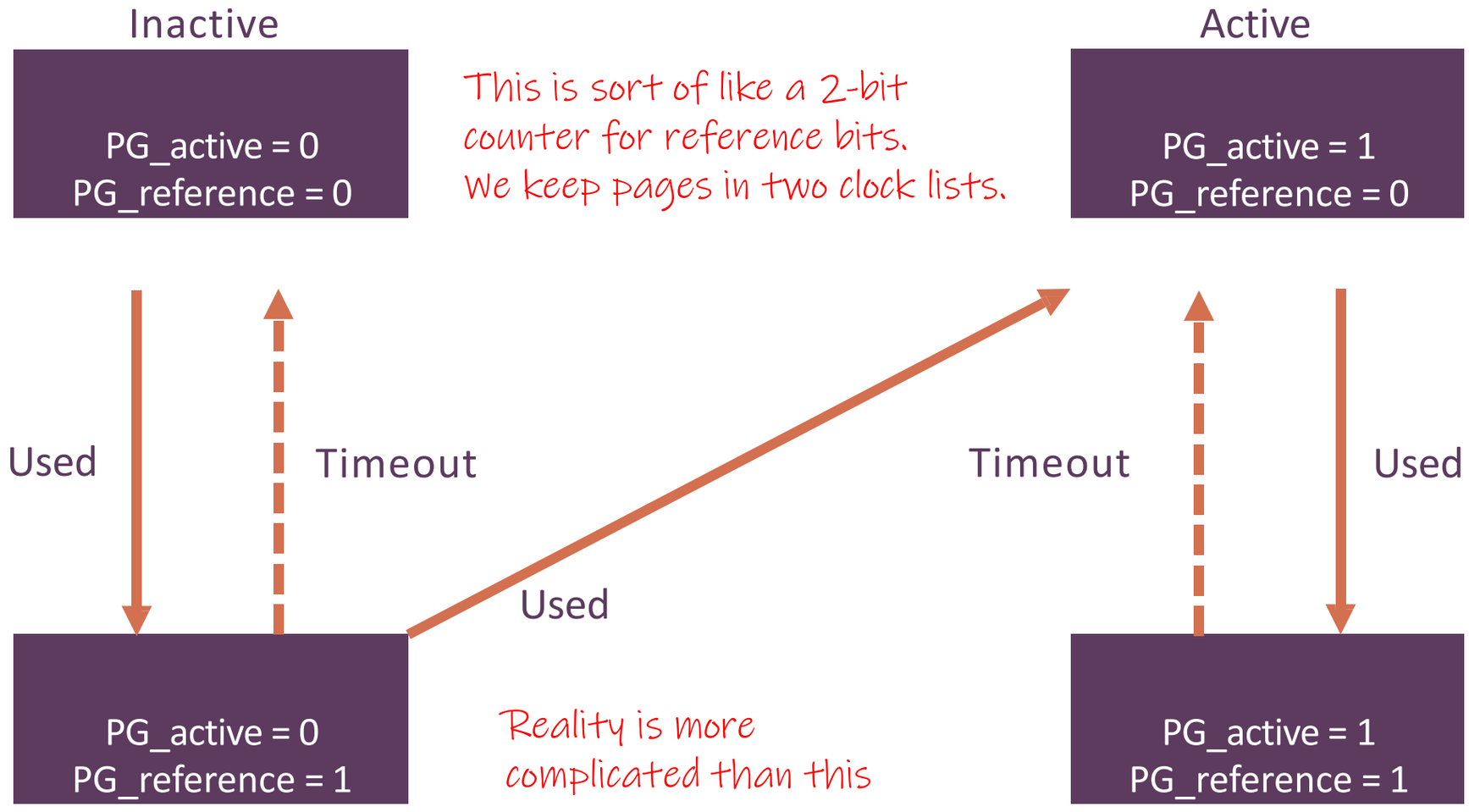
Can also be modified to prefer to evict clean pages instead of dirty pages



Linux Two-List Clock Page Replacement Algorithm

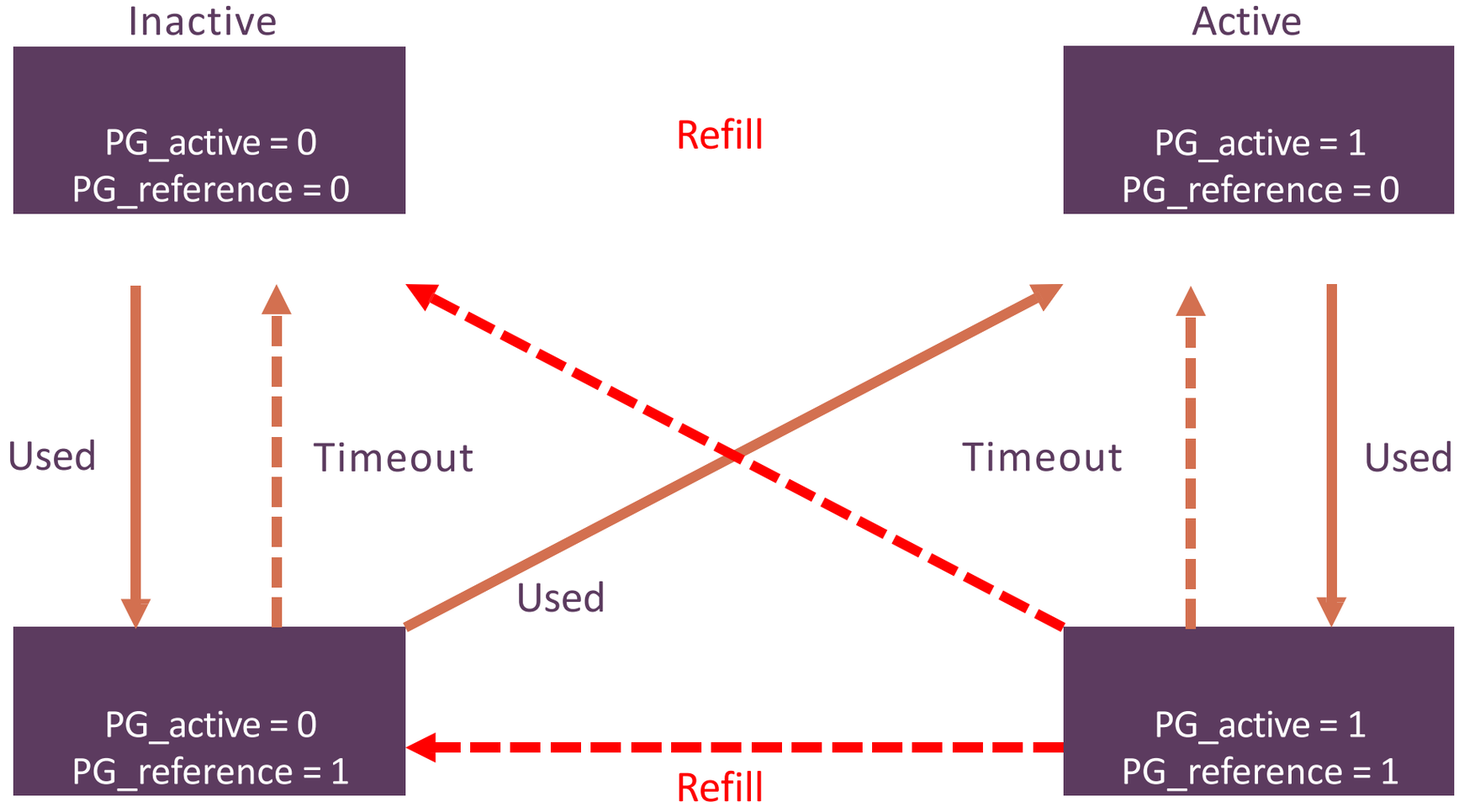
- Maintains two lists: Active list and Inactive list
- Eviction Priority:
  - Chose a page from the inactive list first
- Page Access Behavior:
  - If a page has not been referenced recently, move it to the inactive list
- If a page is referenced:
  - Set its reference flag to true
  - It will be moved to the active list on the next access
  - Two accesses are required for a page to become active
- Decay Mechanism:
  - If the second access doesn't happen, the reference flag is reset periodically
  - After two timeouts without activity, the page is moved to the inactive list

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"



Active should be ~2/3 of pages at most

Now, let's tie it all together.

Virtual Memory in the Context of Forking

- Behind the scenes the kernel, TLB, & MMU all work together to enforce isolation.
- When we fork, the child inherits the virtual address space of the parent explaining how all the pointers and addresses don't change.
  - But, do the mappings need to change? Do we need to make a copy of everything within the memory of the parent? The heap, stack, text, etc?

- Usually, when you fork it is followed by what call?
  - Think about in the context of Shredder, Penn-Shell, etc…
  - **You usually call execvp**! (How is this important?)

Fork and Virtual Memory

```
int main(){
    int x = 3;
    int* ptr = &x;

    printf("[Before Fork]\t x = %d\n", x);
    printf("[Before Fork]\t ptr = %p\n", ptr);

    if (fork() == 0) {
        printf("[Child]\t\t x = %d\n", x);
        printf("[Child]\t\t ptr = %p\n", ptr);
        return EXIT_SUCCESS;
    }
    waitpid(pid, NULL, 0);
    printf("[Parent]\t x = %d\n", x);
    printf("[Parent]\t ptr = %p\n", ptr);

    return EXIT_SUCCESS;
}
```

Shared by Parent and Child

Virtual Addr	Values
	0x00
	0x00
	0x00
0xffffee402b28	**0x03**
	0x00
	0x00
	0xff
	0xff
	0xee
	0x40
	0x2b
0xffffee402b20	0x28

TLB + MMU

Page P

0x00
0x00
0x00
0x03
0x00
0x00
0xff
0xff
0xee
0x40
0x2b
0x28

At this point, two processes have been spawned that share the same virtual address space.

But what should we do about their mappings?

Remember, we want to enforce isolation. Changes made by one process shouldn't be viewable by another. An easy solution is to give each individual mappings and then copy over their contents!

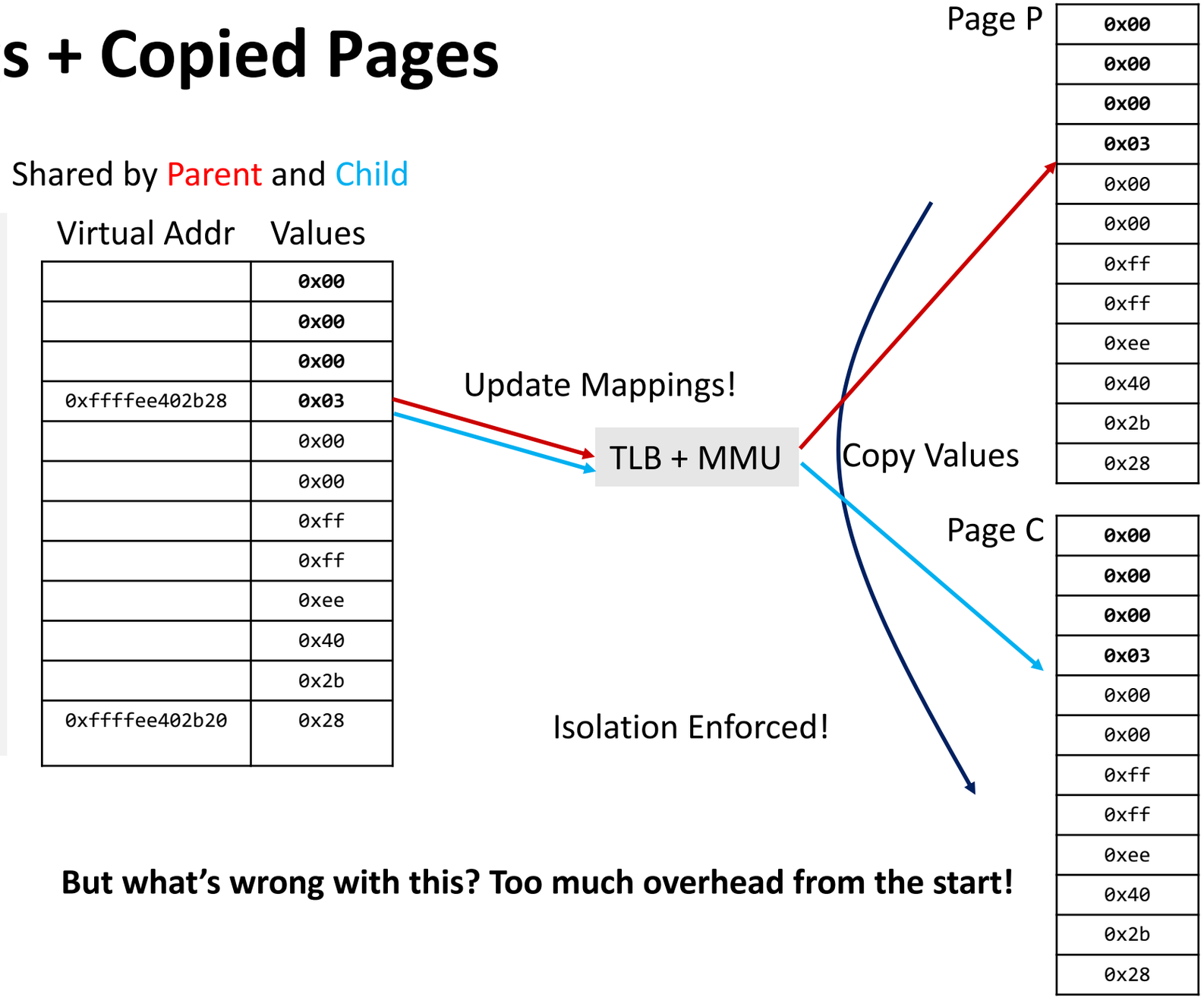
Separate Mappings + Copied Pages

```
int main(){
    int x = 3;
    int* ptr = &x;

    printf("[Before Fork]\t x = %d\n", x);
    printf("[Before Fork]\t ptr = %p\n", ptr);

    if (fork() == 0) {
        printf("[Child]\t\t x = %d\n", x);
        printf("[Child]\t\t ptr = %p\n", ptr);
        return EXIT_SUCCESS;
    }
    waitpid(pid, NULL, 0);
    printf("[Parent]\t x = %d\n", x);
    printf("[Parent]\t ptr = %p\n", ptr);

    return EXIT_SUCCESS;
}
```



But what's wrong with this? Too much overhead from the start!

Sharing Pages and Tables…

```
int main(){
    int x = 3;
    int* ptr = &x;

    printf("[Before Fork]\t x = %d\n", x);
    printf("[Before Fork]\t ptr = %p\n", ptr);

    if (fork() == 0) {
        printf("[Child]\t\t x = %d\n", x);
        printf("[Child]\t\t ptr = %p\n", ptr);
        return EXIT_SUCCESS;
    }
    waitpid(pid, NULL, 0);
    printf("[Parent]\t x = %d\n", x);
    printf("[Parent]\t ptr = %p\n", ptr);

    return EXIT_SUCCESS;
}
```

Shared by Parent and Child

Virtual Addr	Values
	0x00
	0x00
	0x00
0xffffee402b28	**0x03**
	0x00
	0x00
	0xff
	0xff
	0xee
	0x40
	0x2b
0xffffee402b20	0x28

TLB + MMU

Page P

0x00
0x00
0x00
0x03
0x00
0x00
0xff
0xff
0xee
0x40
0x2b
0x28

Remember, we want to enforce isolation. Changes made by one process shouldn't be viewable by another. But take a look at the code here. **Do either of them make changes?**

Sharing Pages and Tables…

```
int main(){
    int x = 3;
    int* ptr = &x;

    printf("[Before Fork]\t x = %d\n", x);
    printf("[Before Fork]\t ptr = %p\n", ptr);

    if (fork() == 0) {
        printf("[Child]\t\t x = %d\n", x);
        printf("[Child]\t\t ptr = %p\n", ptr);
        return EXIT_SUCCESS;
    }
    waitpid(pid, NULL, 0);
    printf("[Parent]\t x = %d\n", x);
    printf("[Parent]\t ptr = %p\n", ptr);

    return EXIT_SUCCESS;
}
```

Shared by Parent and Child

Virtual Addr	Values
	0x00
	0x00
	0x00
0xffffee402b28	**0x03**
	0x00
	0x00
	0xff
	0xff
	0xee
	0x40
	0x2b
0xffffee402b20	0x28

TLB + MMU

these are loads.

Page P

0x00
0x00
0x00
0x03
0x00
0x00
0xff
0xff
0xee
0x40
0x2b
0x28

If either process changes the values at these locations, you want to ***ensure the other process can not see them.***

However, if all the processes do is read from them, then they can share identical mappings. No problem!

However, we need to make sure to mark each page as read only! (Why?)

Copy-On-Write

```
int main(){
    int x = 3;
    int* ptr = &x;

    printf("[Before Fork]\t x = %d\n", x);
    printf("[Before Fork]\t ptr = %p\n", ptr);

    if (fork() == 0) {
        x++;
        printf("[Child]\t\t x = %d\n", x);
        printf("[Child]\t\t ptr = %p\n", ptr);
        return EXIT_SUCCESS;
    }
    x--;
    waitpid(pid, NULL, 0);
    printf("[Parent]\t x = %d\n", x);
    printf("[Parent]\t ptr = %p\n", ptr);

    return EXIT_SUCCESS;
}
```

Shared by Parent and Child

Virtual Addr	Values
	0x00
	0x00
	0x00
0xffffee402b28	**0x03**
	0x00
	0x00
	0xff
	0xff
	0xee
	0x40
	0x2b
0xffffee402b20	0x28

TLB + MMU

Page P

0x00
0x00
0x00
0x03
0x00
0x00
0xff
0xff
0xee
0x40
0x2b
0x28

Now, we've changed the program to write to the pages. Can we still keep the previous design? ***Yes.***

When we fork the process, do not copy the entire pages from the start. However, mark the pages as ***read only***.

This would require updating the entries in the table…

However, now when a process attempts to ***write to the page*** the kernel will receive a ***fault from the MMU!***

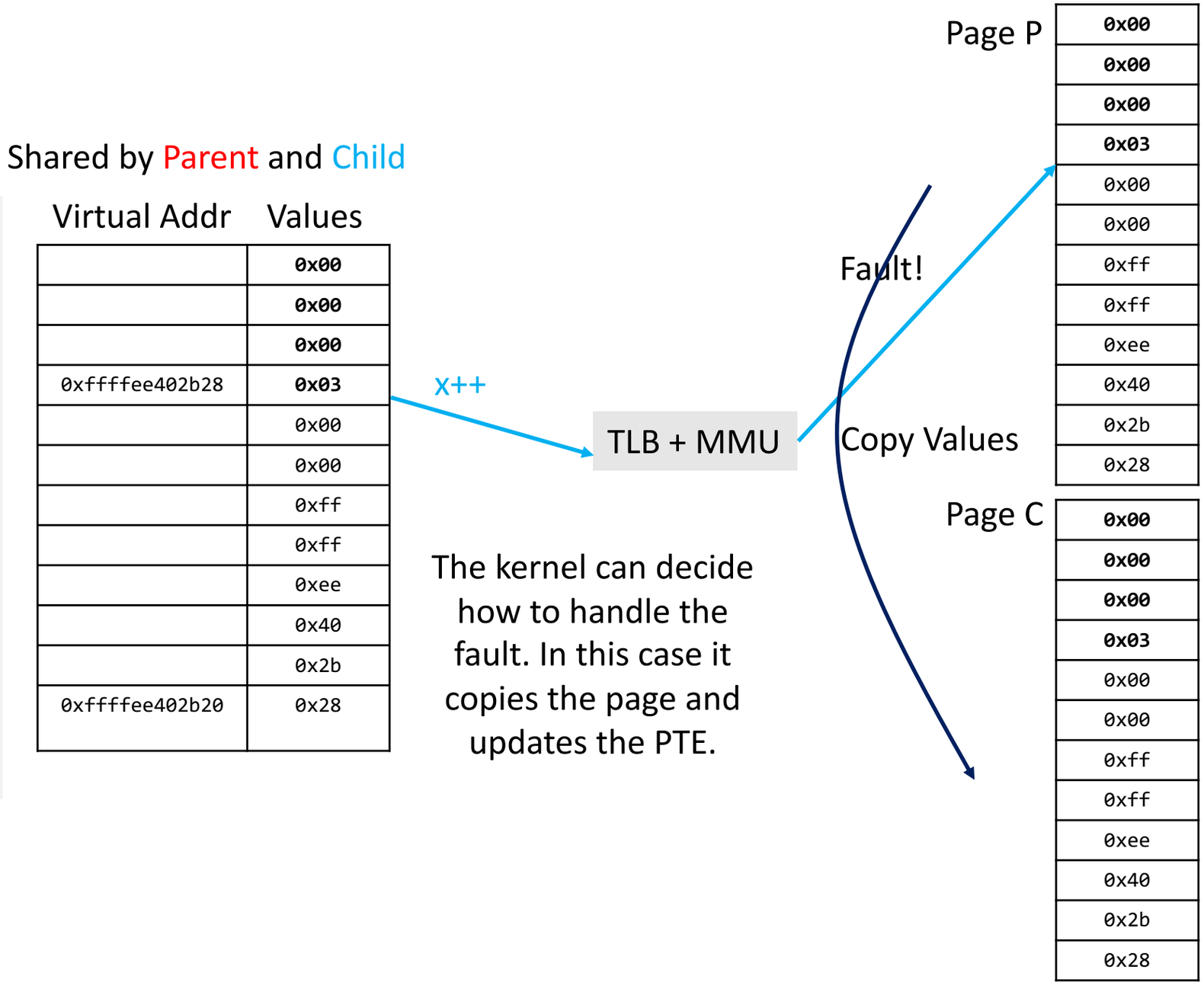
Copy-On-Write

```
int main(){
    int x = 3;
    int* ptr = &x;

    printf("[Before Fork]\t x = %d\n", x);
    printf("[Before Fork]\t ptr = %p\n", ptr);

    if (fork() == 0) {
        x++;
        printf("[Child]\t\t x = %d\n", x);
        printf("[Child]\t\t ptr = %p\n", ptr);
        return EXIT_SUCCESS;
    }
    x--;
    waitpid(pid, NULL, 0);
    printf("[Parent]\t x = %d\n", x);
    printf("[Parent]\t ptr = %p\n", ptr);

    return EXIT_SUCCESS;
}
```



Copy-On-Write

```
int main(){
    int x = 3;
    int* ptr = &x;

    printf("[Before Fork]\t x = %d\n", x);
    printf("[Before Fork]\t ptr = %p\n", ptr);

    if (fork() == 0) {
        x++;
        printf("[Child]\t\t x = %d\n", x);
        printf("[Child]\t\t ptr = %p\n", ptr);
        return EXIT_SUCCESS;
    }
    x--;
    waitpid(pid, NULL, 0);
    printf("[Parent]\t x = %d\n", x);
    printf("[Parent]\t ptr = %p\n", ptr);

    return EXIT_SUCCESS;
}
```

Shared by Parent and Child

Virtual Addr	Values
	0x00
	0x00
	0x00
0xffffee402b28	**0x03**
	0x00
	0x00
	0xff
	0xff
	0xee
	0x40
	0x2b
0xffffee402b20	0x28

x++

TLB + MMU

The kernel can decide how to handle the fault. In this case it copies the page and updates the PTE.

Page P

0x00
0x00
0x00
0x03
0x00
0x00
0xff
0xff
0xee
0x40
0x2b
0x28

Page C

0x00
0x00
0x00
0x03
0x00
0x00
0xff
0xff
0xee
0x40
0x2b
0x28

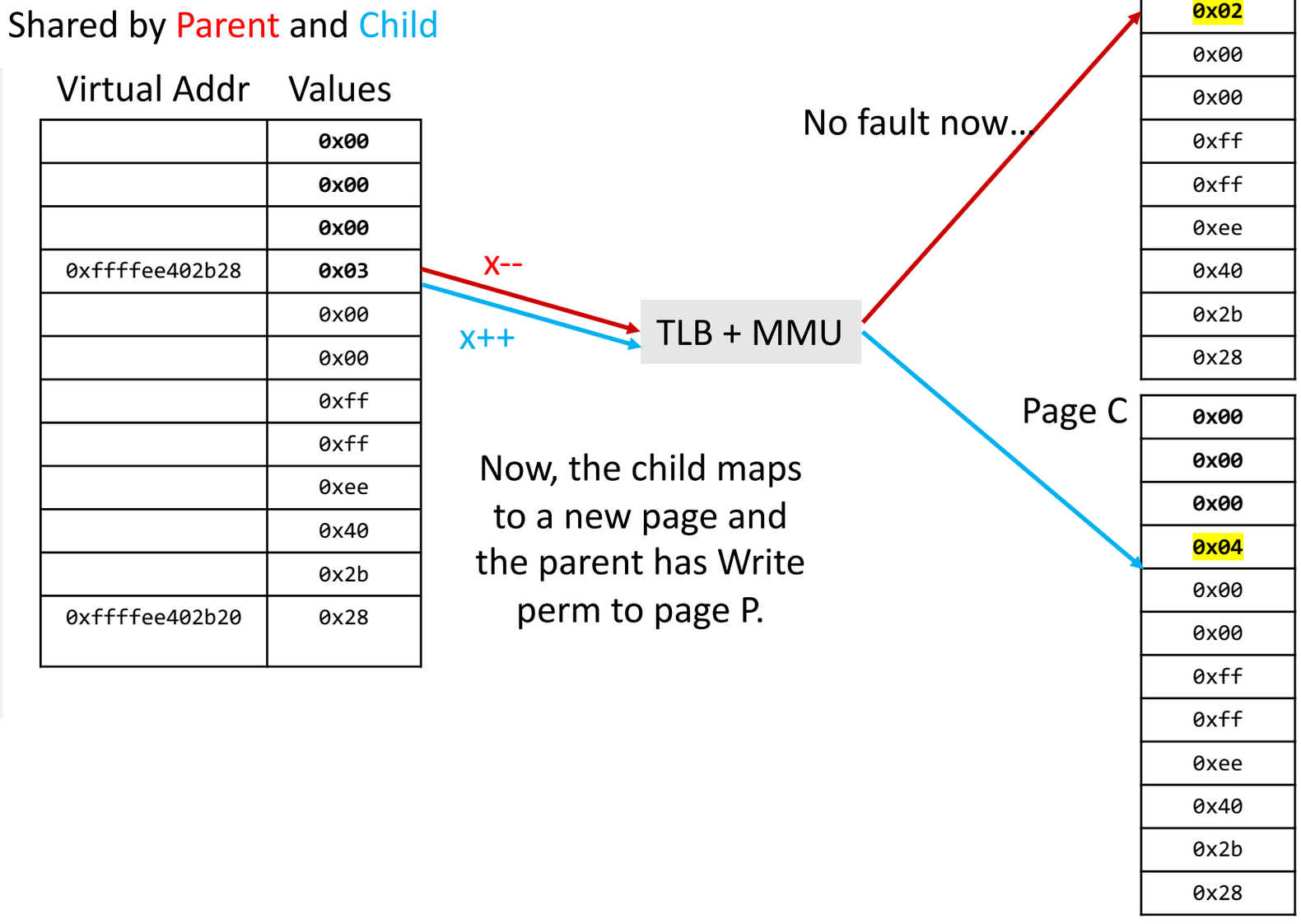
Copy-On-Write

```
int main(){
    int x = 3;
    int* ptr = &x;

    printf("[Before Fork]\t x = %d\n", x);
    printf("[Before Fork]\t ptr = %p\n", ptr);

    if (fork() == 0) {
        x++;
        printf("[Child]\t\t x = %d\n", x);
        printf("[Child]\t\t ptr = %p\n", ptr);
        return EXIT_SUCCESS;
    }
    x--;
    waitpid(pid, NULL, 0);
    printf("[Parent]\t x = %d\n", x);
    printf("[Parent]\t ptr = %p\n", ptr);

    return EXIT_SUCCESS;
}
```



Copy-On-Write

- All pages unchanged as the parent and child have identical state after the fork.
- When we fork, the appropriate mappings are set to read only.
  - We then use the MMU to enforce these permissions by triggering a Fault.
- The kernel handles the fault appropriately, in our case, by updating the page table entries and copying the page over for the process that performed the write.
  - Note: This is done on a page by page basis. Just because a process writes to a singular page doesn't mean it's time to copy over the entire page table and all the pages mapped.
- Greatly reduces the overhead! Now if you `execvp` after forking**, there aren't any pages that are "wasted" as none were allocated for you nor any copied for you.**

Virtual Memory

- And that's the majority of Virtual Memory…

- Super Optional Reading: ARMv7 Documentation
  - Chapter B3 Virtual Memory System Architecture (VMSA)
  - Outlines the hardware design and how it should be used with software…
  - ARMv6 has a simpler design…check it out for some light reading.

- See the RISC-V Address Translation Process…

Poll Everywhere

pollev.com/cis5480

- 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
Evicted					1	2	3	4			1	2	

9 Faults, 6 Evictions!

Poll Everywhere

pollev.com/cis5480

- 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
Newest		1	2	3	4	4	4	5	5	5	3	4	4
			1	2	2	2	2	2	2	2	5	3	3
Oldest				1	1	1	1	1	1	1	2	5	5
Evicted					3			4			1	2	

7 Faults, 4 Evictions!

Poll Everywhere

pollev.com/cis5480

- 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	

Poll Everywhere

pollev.com/cis5480

- 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

Poll Everywhere

pollev.com/cis5480

- Now, using the same Reference String with LRU, let's try to fill this table out…
- If we use 4 frames, how many page faults will there be?

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

6 Faults, 2 evictions!

Poll Everywhere

pollev.com/cis5480

- Now, using the same Reference String with LRU, let's try to fill this table out…
- If we use 4 frames, how many page faults will there be?
  - 6 Faults!
- Easier for ***me*** to think about this in terms of subsequences

4, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3

At the start, we have 4 faults as the first four pages in the reference string are unique!

The current subsequence is 4, 0, 1, 2

Poll Everywhere

pollev.com/cis5480

- Now, using the same Reference String with LRU, let's try to fill this table out…
- If we use 4 frames, how many page faults will there be?
  - 6 Faults!
- Easier for ***me*** to think about this in terms of subsequences

4, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3

Now, we see a zero. A zero is already in our substring, so there's no faults/evictions necessary. We remove the previous zero from the subsequence. And the "new" zero leads the sequence now.

Poll Everywhere

pollev.com/cis5480

- Now, using the same Reference String with LRU, let's try to fill this table out…
- If we use 4 frames, how many page faults will there be?
  - 6 Faults!
- Easier for ***me*** to think about this in terms of subsequences

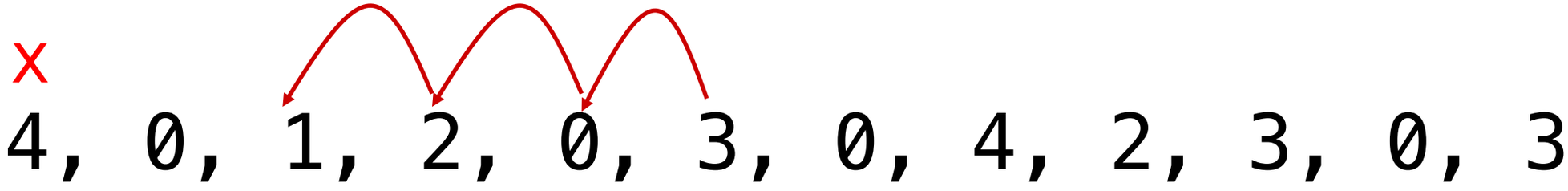
4, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3

With our subsquence being the maximum length and with a new page in view, we perform our first eviction. The last value in the sequence is always the one to be removed using LRU.

Poll Everywhere

pollev.com/cis5480

- Now, using the same Reference String with LRU, let's try to fill this table out…
- If we use 4 frames, how many page faults will there be?
  - 6 Faults!
- Easier for ***me*** to think about this in terms of subsequences

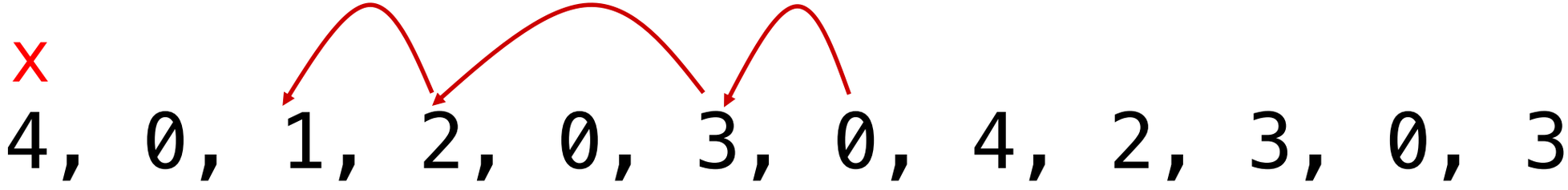


With our subsquence being the maximum length and with a new page in view, we perform our first eviction. The last value in the sequence is always the one to be removed using LRU.

Poll Everywhere

pollev.com/cis5480

- Now, using the same Reference String with LRU, let's try to fill this table out…
- If we use 4 frames, how many page faults will there be?
  - 6 Faults!
- Easier for ***me*** to think about this in terms of subsequences

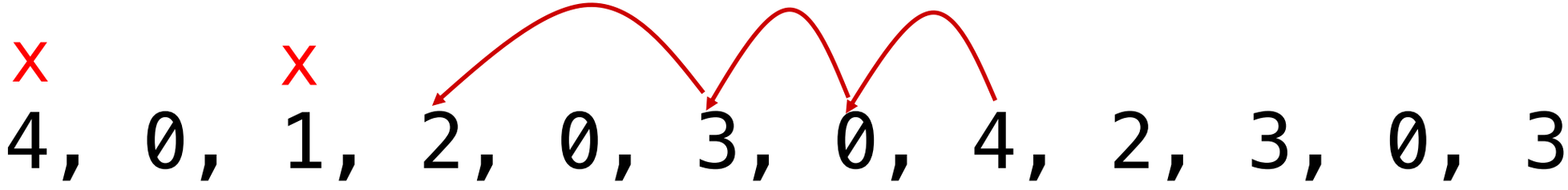


We see another zero, we know what to do.

Poll Everywhere

pollev.com/cis5480

- Now, using the same Reference String with LRU, let's try to fill this table out…
- If we use 4 frames, how many page faults will there be?
  - 6 Faults!
- Easier for ***me*** to think about this in terms of subsequences

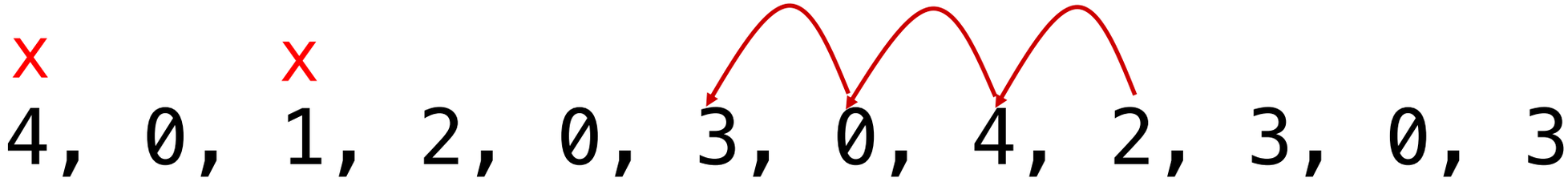


We perform our second eviction!

Poll Everywhere

pollev.com/cis5480

- Now, using the same Reference String with LRU, let's try to fill this table out…
- If we use 4 frames, how many page faults will there be?
  - 6 Faults!
- Easier for ***me*** to think about this in terms of subsequences



We see another two, no eviction or fault necessary as it was already in the subsequence.

Poll Everywhere

pollev.com/cis5480

- Now, using the same Reference String with LRU, let's try to fill this table out…
- If we use 4 frames, how many page faults will there be?
  - 6 Faults!
- Easier for ***me*** to think about this in terms of subsequences

Same thing with the three…

Now, we see that the rest of the pages are already in the sequence so there's no need to continue this.

2 Evictions (With 2 corresponding faults) and the initial 4 faults.