Georgia Tech Computer Science

CS2200
Systems and Networks
Spring 2022

Lecture 17:
Virtual Memory pt 3

Alexandros (Alex) Daglis
School of Computer Science
Georgia Institute of Technology
adaglis@gatech.edu

Lecture slides adapted from Bill Leahy and Charles Lively of Georgia Tech

Announcements

Exam 2 logistics announced on Canvas

- Will be synchronous in-person during lab next week (minus few exceptions with good reason)

Why the change? Must balance multiple objectives

- Relieve stress
- Foster peer learning
- Make managing exam tractable for teaching team

This is why we will still release questions in advance

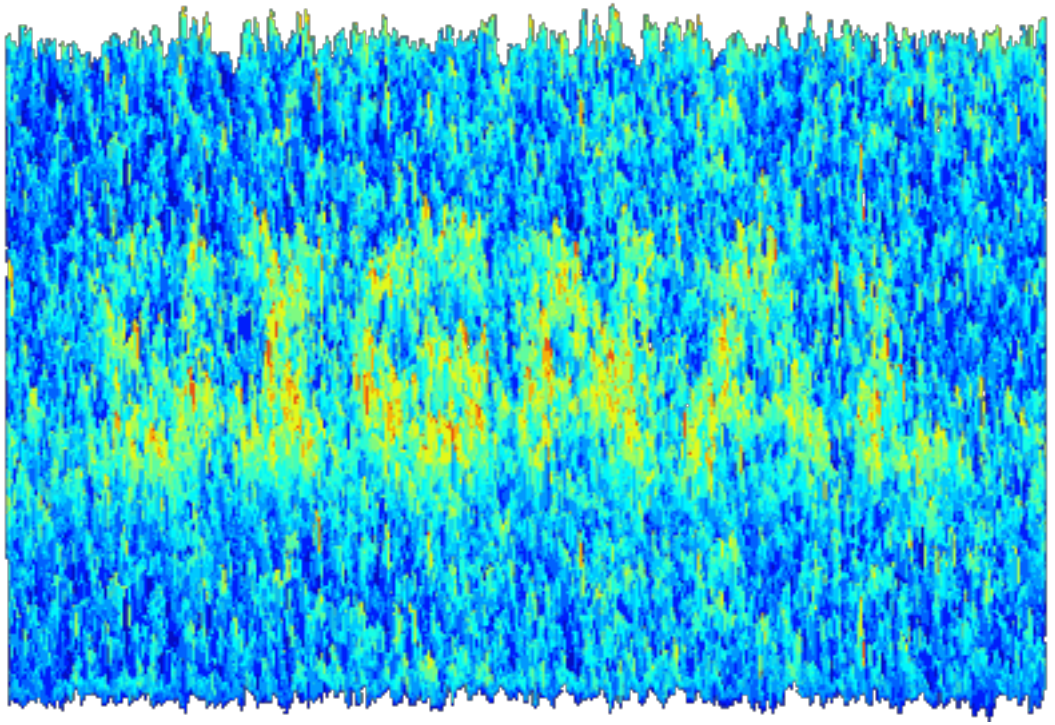
This is why we had to make some changes

Update on TP questions

- Answer A will always be "I just want the participation credit"
- If you were answering randomly, please select this instead

Why?

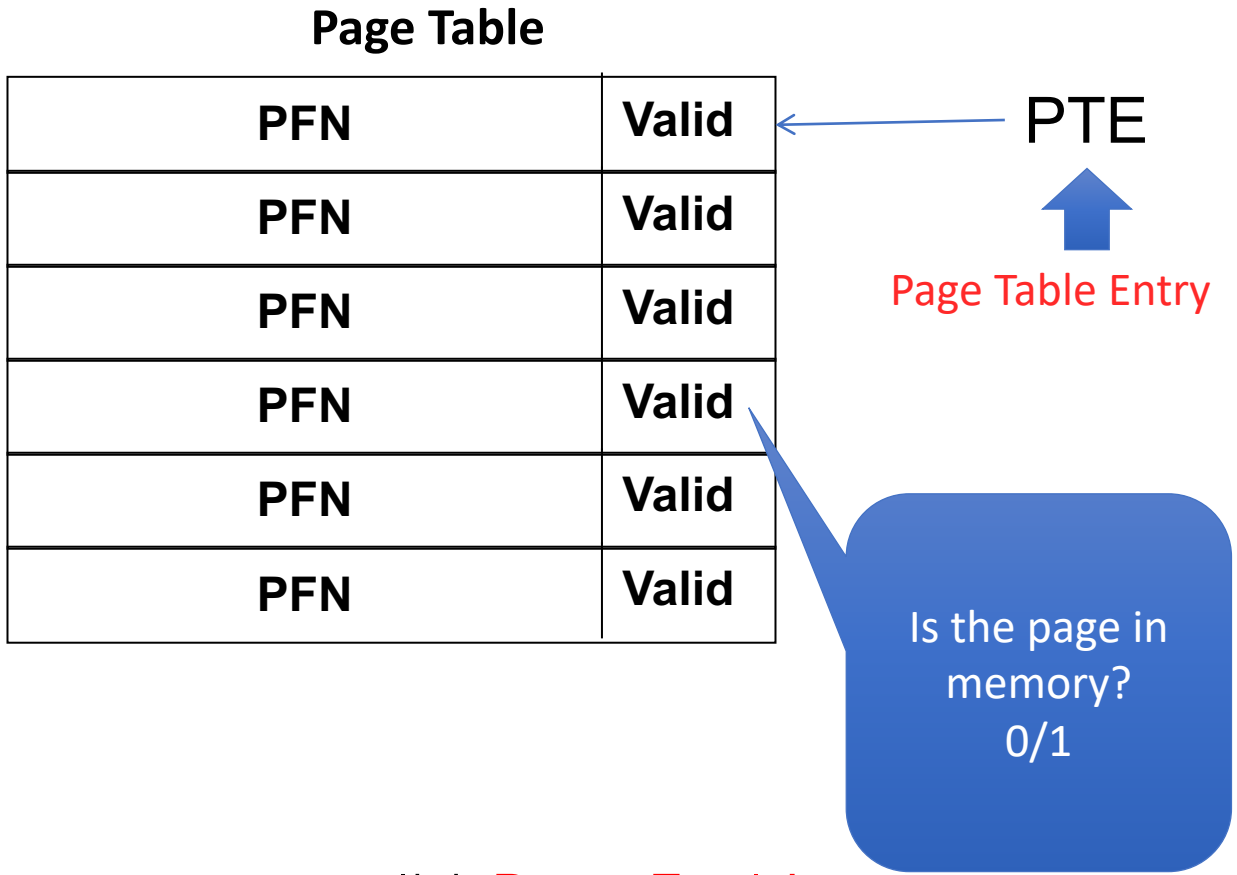
- Help me separate signal from noise



Paged memory allocation

- Allocate all at once at load time? → Not good: slow
- Allocate on demand? → Better utilization

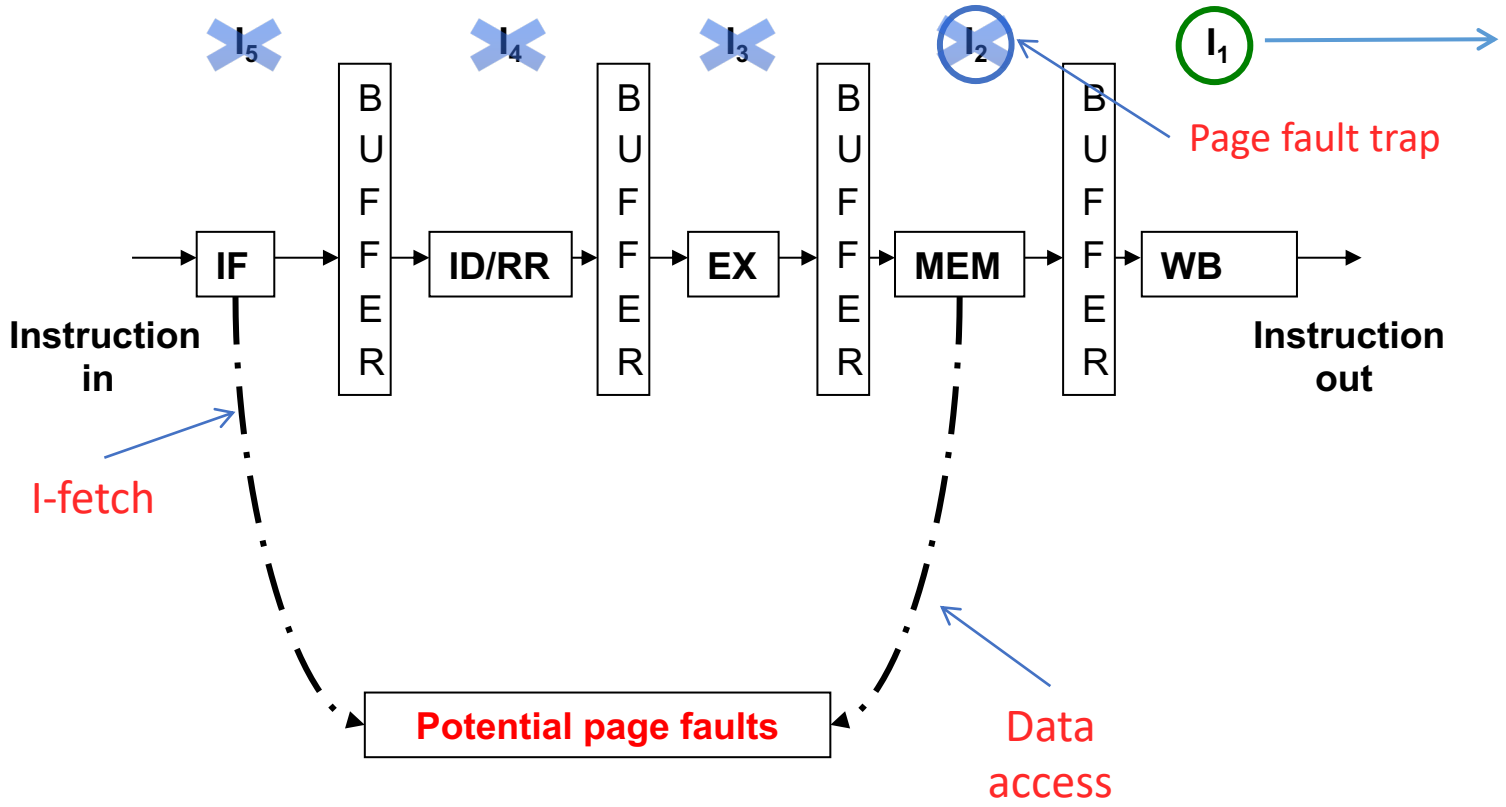
Demand paging



When VPN's corresponding entry not valid: Page Fault!

Ramification of demand paging

Where can a page fault hit us?



Let I_1 complete
Squash I_2-I_5
Trap to page fault handler,
saving PC of I_2 for restart
after page fault is serviced

As you may have guessed, we'll need to make the original PC value part of the pipeline buffers

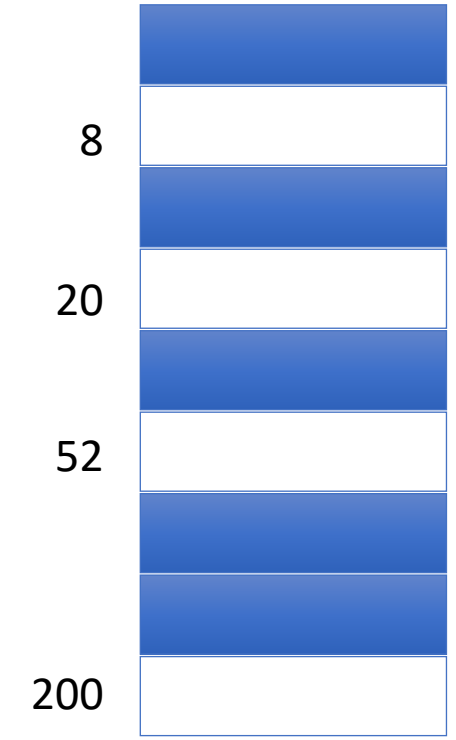
Page fault handler

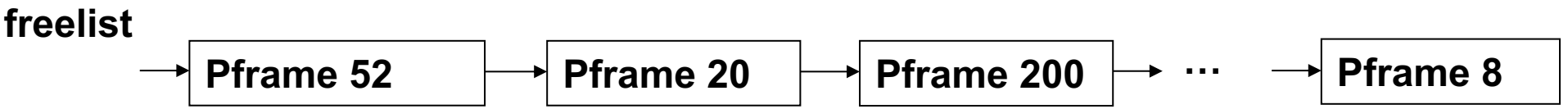
1. Find a free page frame
2. Load the faulting virtual page from disk into the page frame
3. Give up the CPU while waiting for the paging I/O to complete
4. Update the page table entry for the faulting page
5. Place the PCB of the process back in the ready_q of the scheduler
6. Call the scheduler

Three Data structures for page fault handler

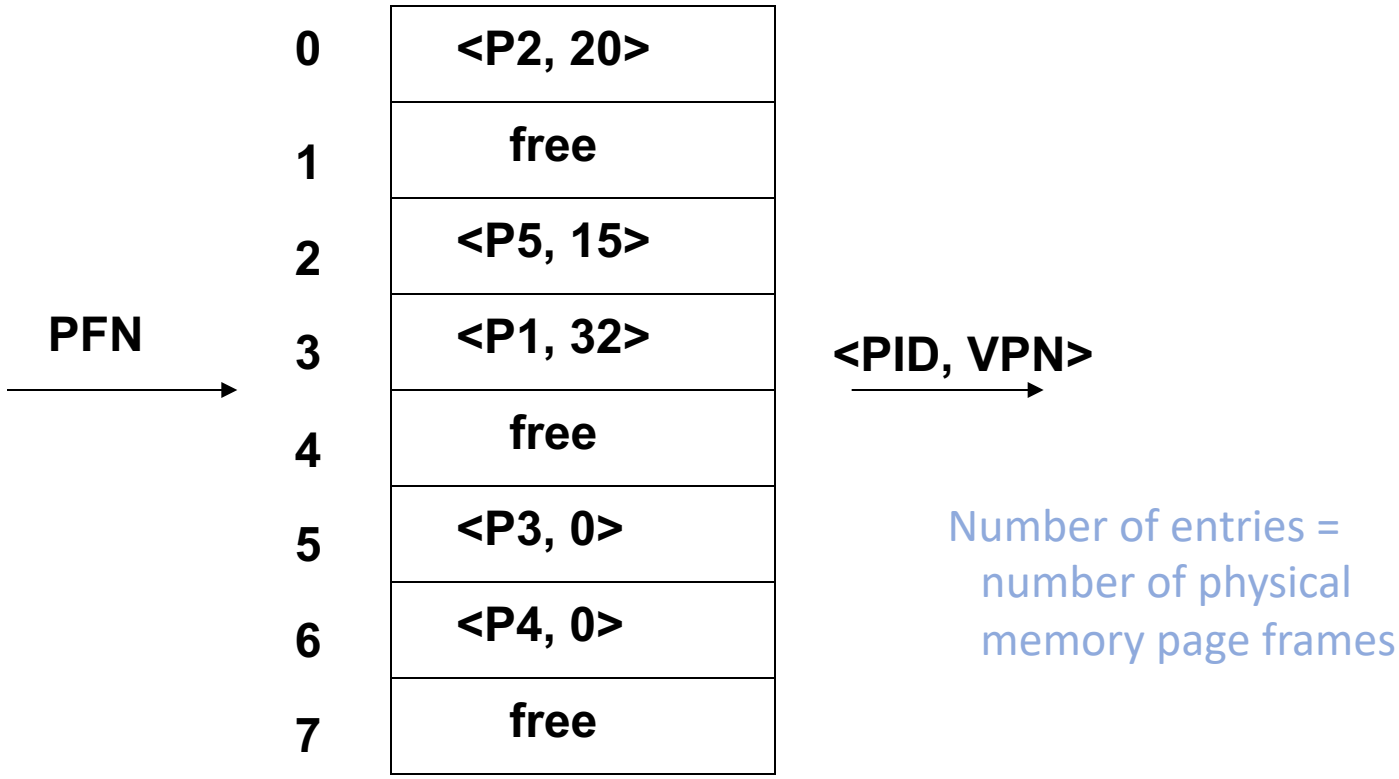
(1/3): Freelist

```
struct pframe {
        address PFN;
        …
        pframe *next;
};
```





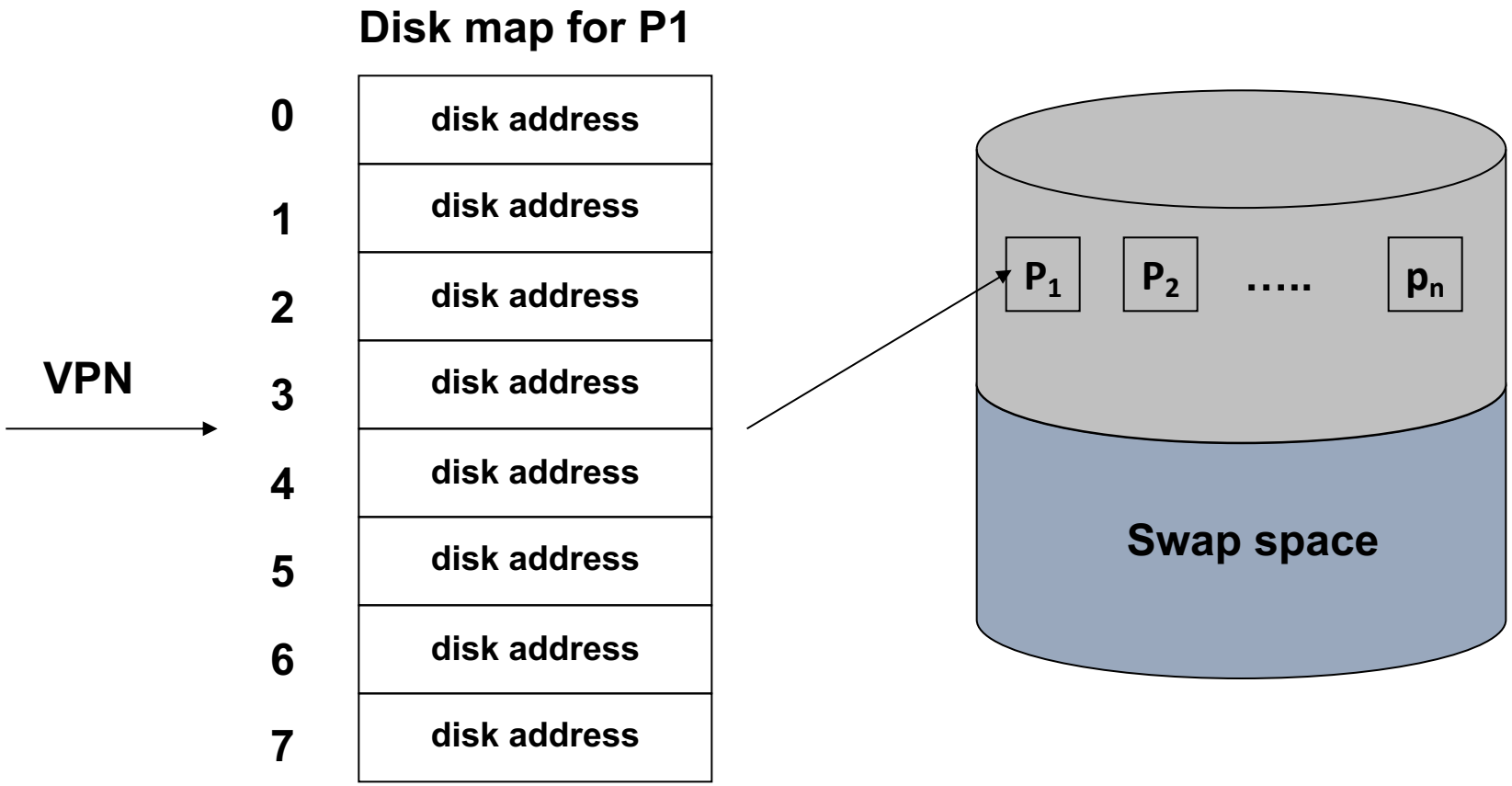
(2/3): Frame Table



Reverse mapping compared to PT

We need this for evicting pages

(3/3): Disk Map


Disk map for P1
0
1
2
3
4
5
6
7
disk address
disk address
disk address
disk address
disk address
disk address
disk address
disk address
VPN
P1
P2
.....
pn
Swap space

Virtual memory manager data structures

Per process	PCB	Holds saved PTBR register
	Page table	VPN → PFN mapping Dual role: Memory manager uses it for setup Hardware uses on each memory access
	Disk map	VPN to disk block mapping needed for bringing missing pages from disk to physical memory
Per system	Free list	Free page frames in physical memory
	Frame table	PFN to <PID, VPN> mapping needed for evicting pages from physical memory

PCB for demand paging

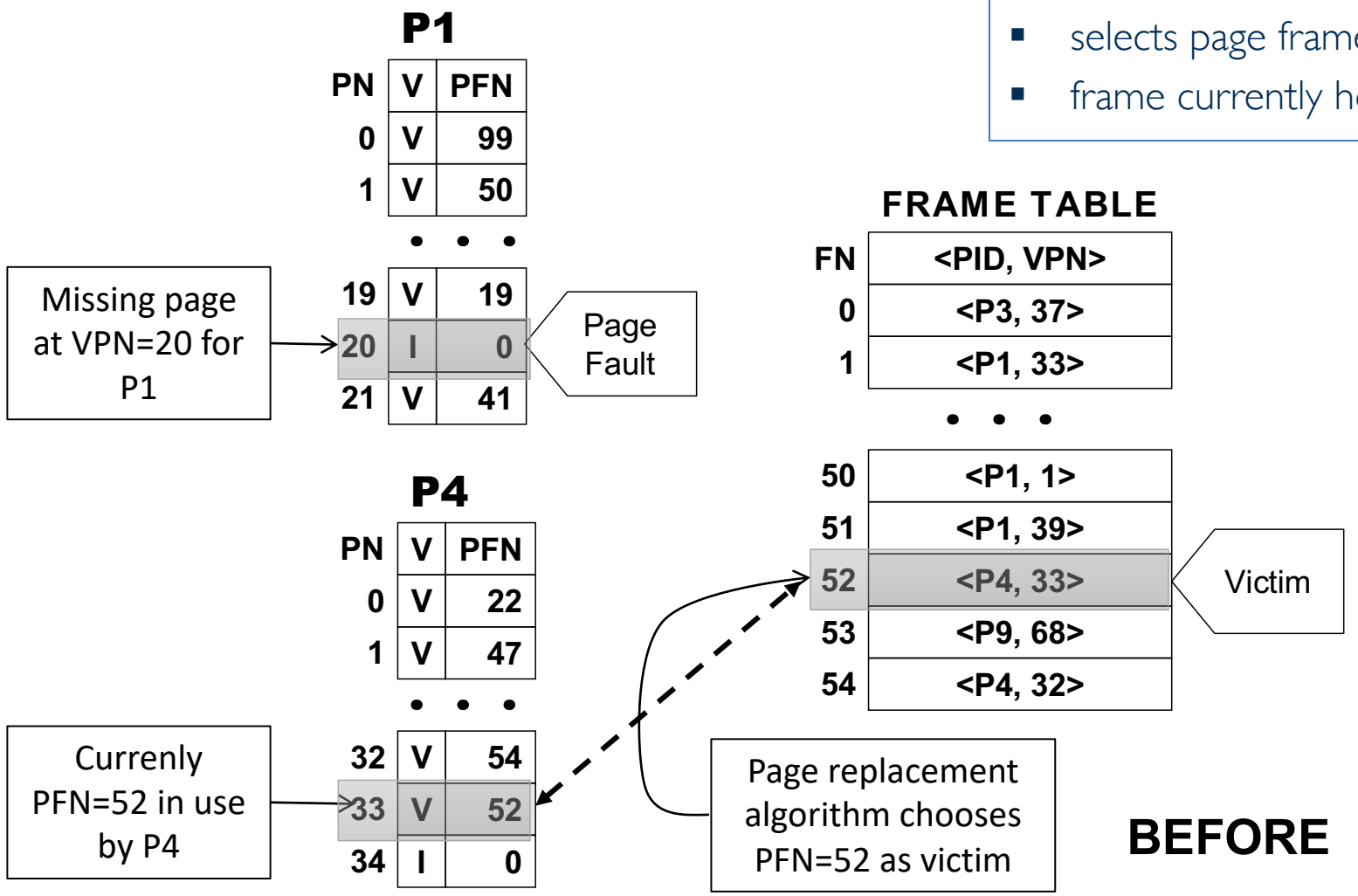
```
enum  state_type {new, ready, running,
                  waiting, halted};
typedef struct control_block_type {
  enum state_type state;
  address PC;
  int reg_file[NUMREGS];
  struct control_block *next_pcb;
  int priority;
  address PTBR;
  disk_address *disk_map;
  ….
  ….
} control_block;
```

Example

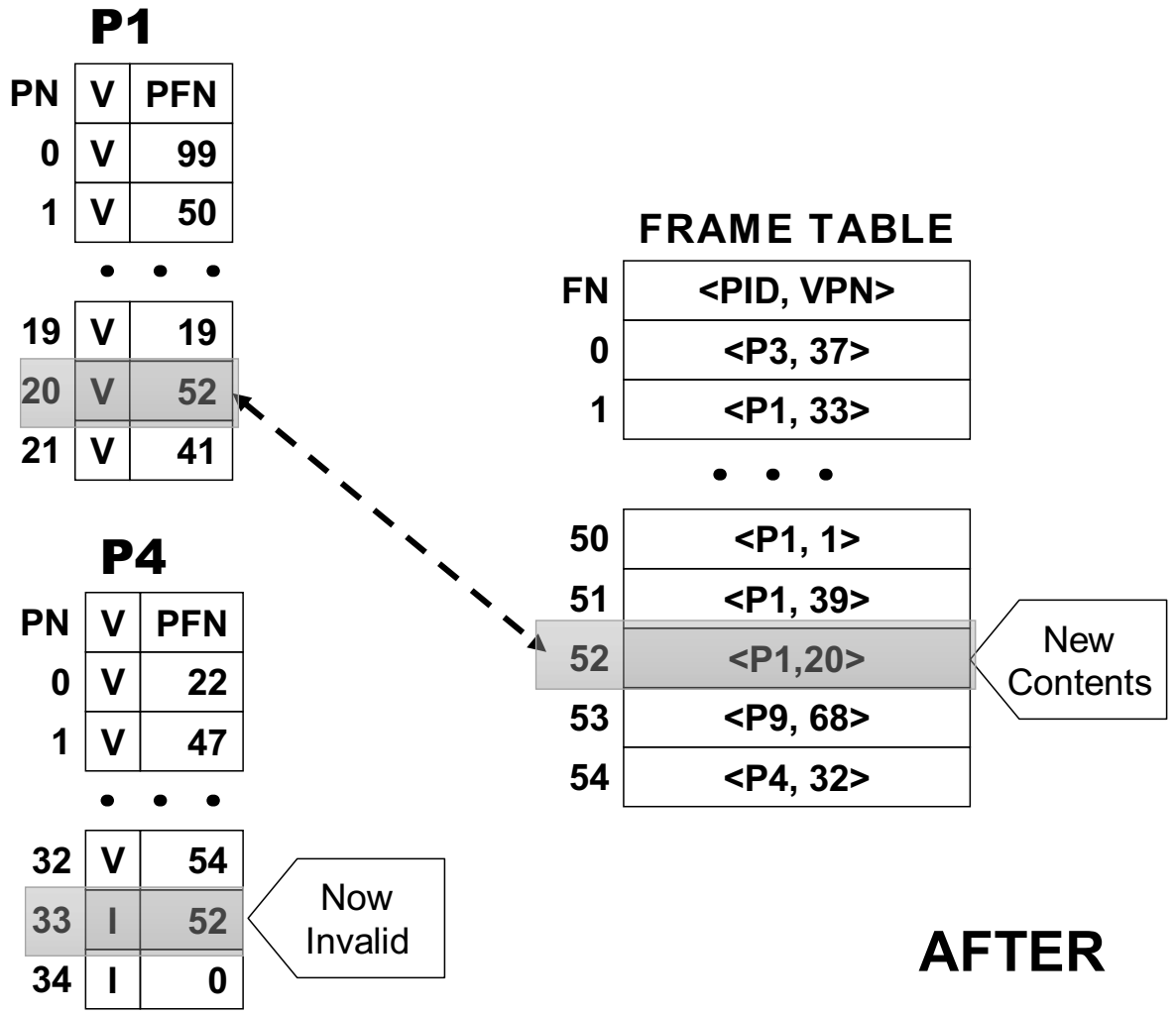
- process P1 page fault at VPN = 20.
- free-list is empty.
- selects page frame PFN = 52 as the victim.
- frame currently houses VPN = 33 of process P4

Before

- process P1 page fault at VPN = 20.
- free-list is empty.
- selects page frame PFN = 52 as the victim.
- frame currently houses VPN = 33 of process P4



After

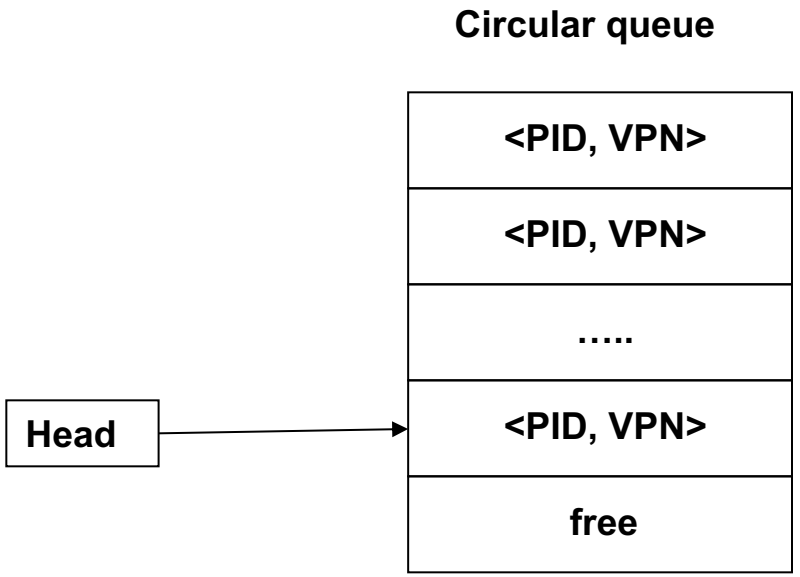

P1
PN V PFN
0 V 99
1 V 50
19 V 19
20 V 52
21 V 41
P4
PN V PFN
0 V 22
1 V 47
32 V 54
33 I 52
34 I 0
Now Invalid
FRAME TABLE
FN <PID, VPN>
0 <P3, 37>
1 <P1, 33>
50 <P1, 1>
51 <P1, 39>
52 <P1,20>
53 <P9, 68>
54 <P4, 32>
New Contents
AFTER

With paged memory management there can be…

A. I just want the participation credit
B. External fragmentation
C. Internal fragmentation
D. No fragmentation
E. Both internal and external fragmentation

FIFO Page Replacement

- Manage the Frame Table like a queue

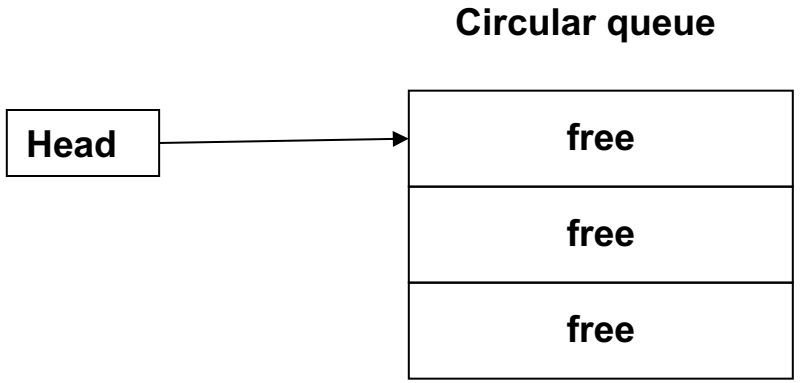


FIFO example

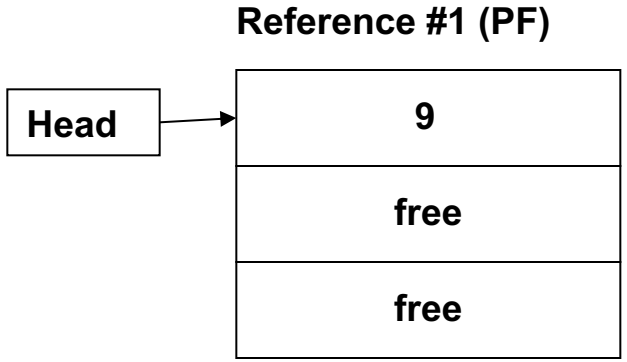
Consider a sequence of page references by a process:

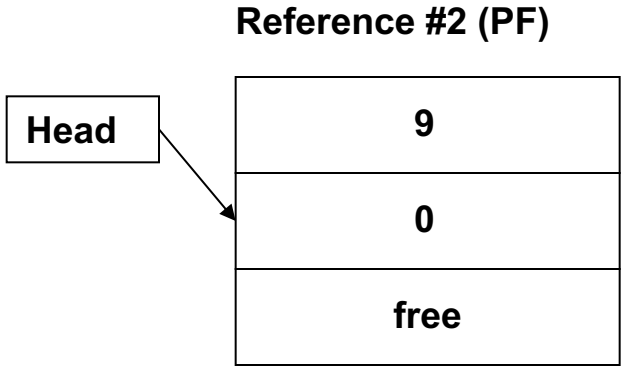
Reference number:	1	2	3	4	5	6	7	8	9	10	11	12	13
Virtual page number:	9	0	3	4	0	5	0	6	4	5	0	5	4

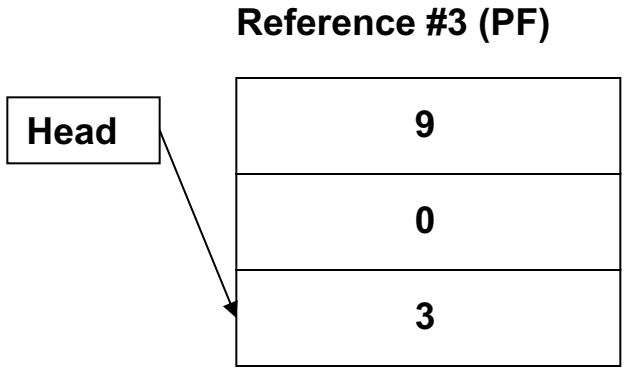
Assume there are 3 physical frames.

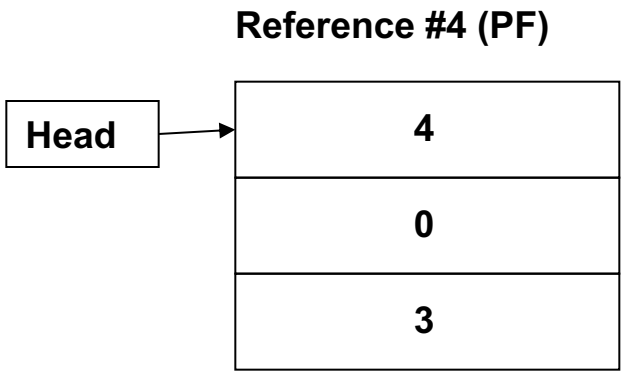


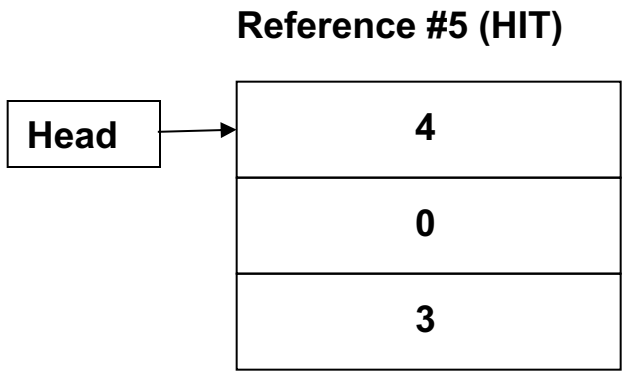
Ref:	1	2	3	4	5	6
VPN:	9	0	3	4	0	5

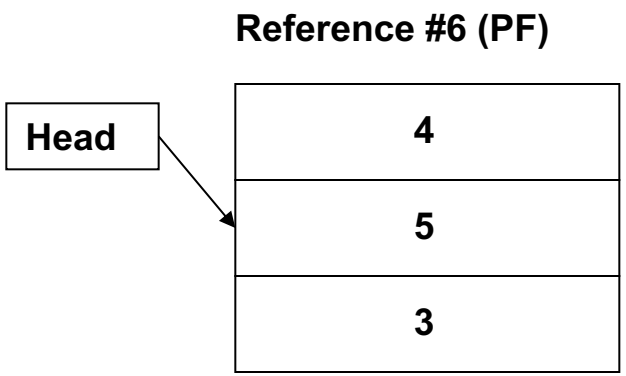












Session ID:
dag2200

Size of the FIFO queue?

A. I just want the participation credit
B. 42
C. Number of physical page frames
D. Number of virtual pages
E. No clue

Belady's Min

Consider a string of page references by a process:

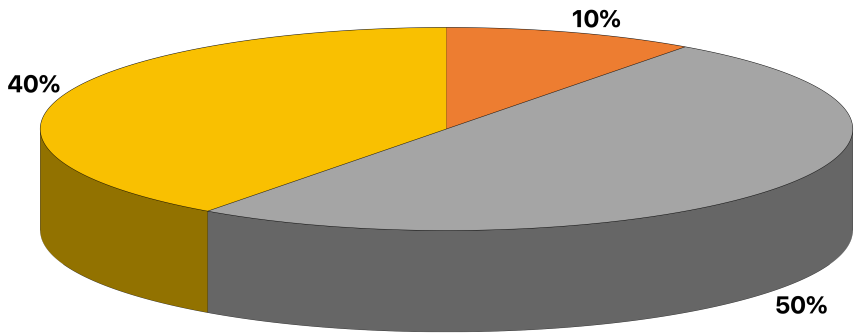
Reference number:	1	2	3	4	5	6	7	8	9	10	11	12	13
Virtual page number:	9	0	3	4	0	5	0	6	4	5	0	5	4

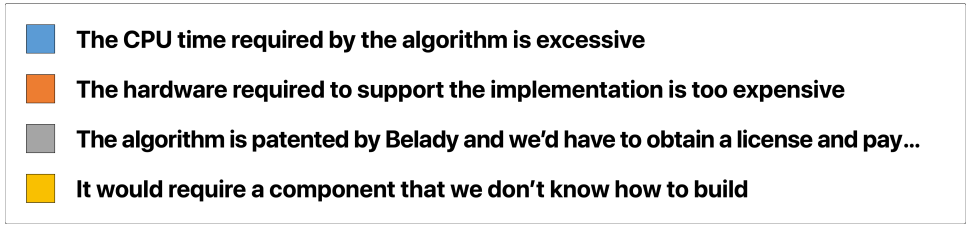
Look into the future!

- Theoretically the best algorithm
- As the victim page, choose the page with the longest time to its next reference
- Merely requires us to predict the future
- If we had pages 0-9 in memory, which pages should we evict first?
- (1, 2, 7, 8) can go first. Then 6, 5, 4, 3, 0, 9

Why not implement Belady's Min as a page replacement algorithm?

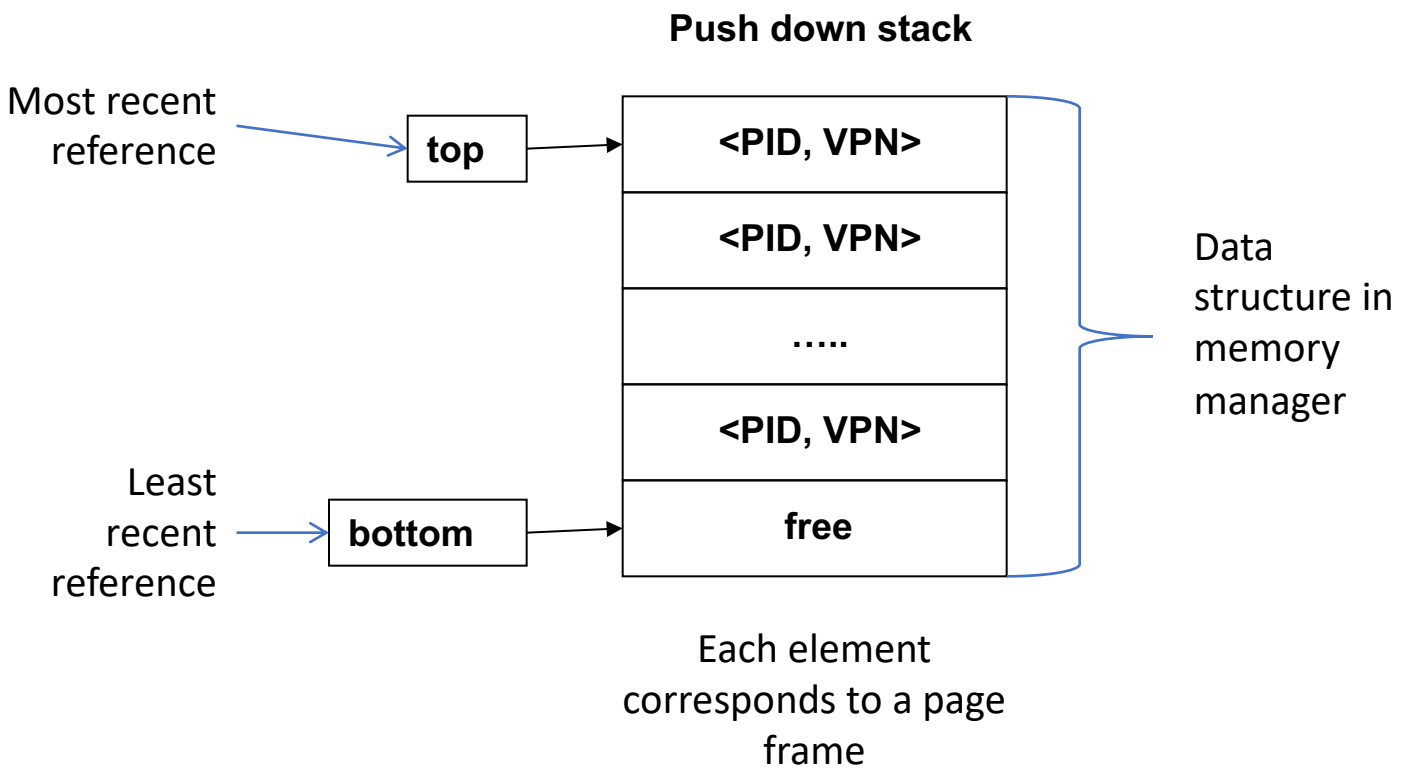
A. I just want the participation credit
B. The CPU time required by the algorithm is excessive
C. The hardware required to support the implementation is too expensive
D. The algorithm is patented by Belady and we'd have to obtain a license and pay royalties
E. It would require a component that we don't know how to build





LRU

- Use the past as a predictor of the future.

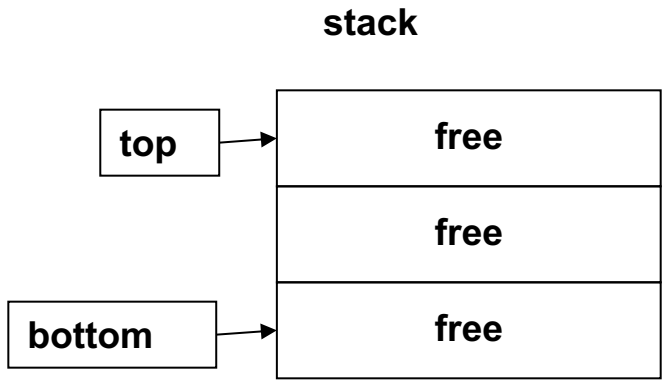


LRU example

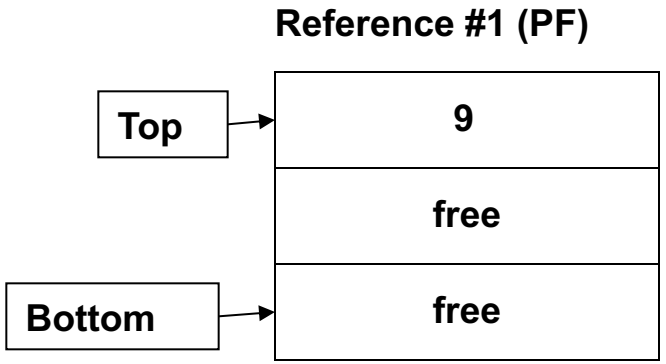
Consider a string of page references by a process:

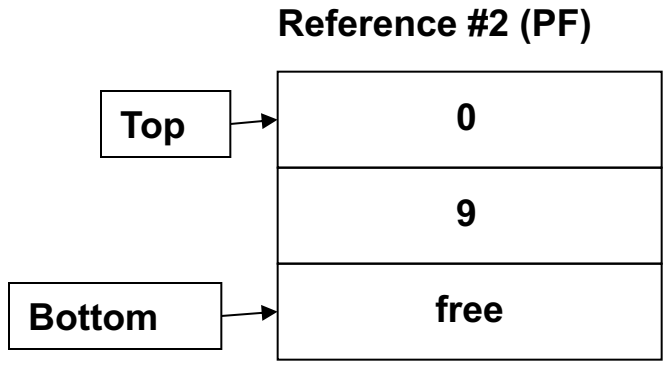
Reference number:	1	2	3	4	5	6	7	8	9	10	11	12	13
Virtual page number:	9	0	3	4	0	5	0	6	4	5	0	5	4

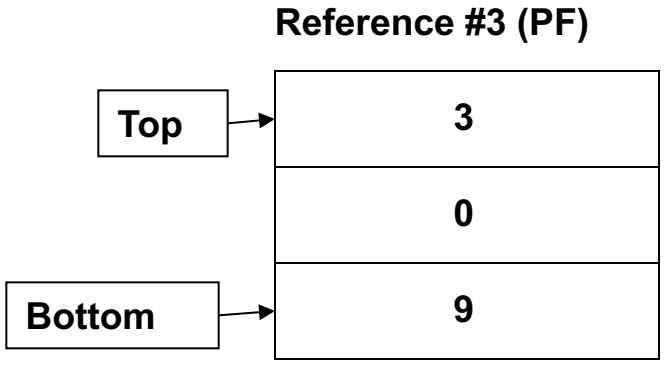
Assume there are 3 physical frames.

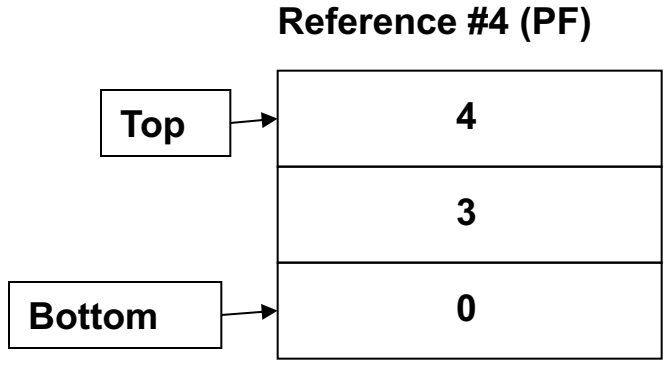


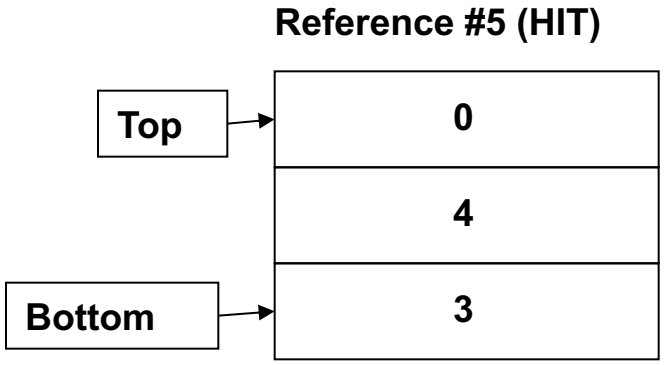
Ref:	1	2	3	4	5	6
VPN:	9	0	3	4	0	5

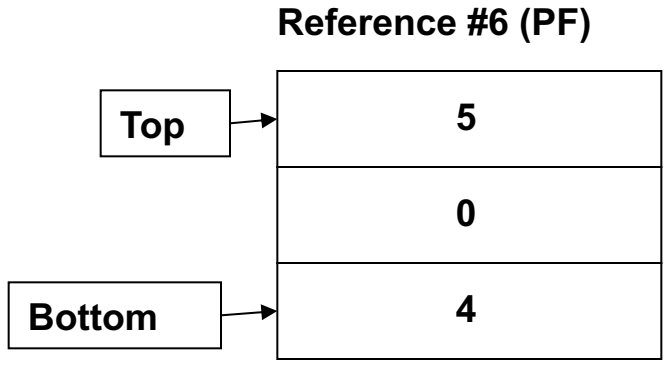












Session ID: dag2200

Size of LRU "stack"

A. I just want the participation credit
B. 42
C. Number of virtual pages
D. Number of physical frames
E. No clue

Problems with LRU

- Memory references are known to the hardware, but memory management (i.e. victim selection) is in software
- One possibility: make the stack shared by HW & SW
  - Implement stack in hardware
  - Let hardware update stack on each reference
  - Let software (OS) use this stack as a data structure
- Will it work?
- Still, no. The size of the stack is the number of page frames (i.e., quite large), so an additional memory write is required for each memory reference

Ways to approximate LRU

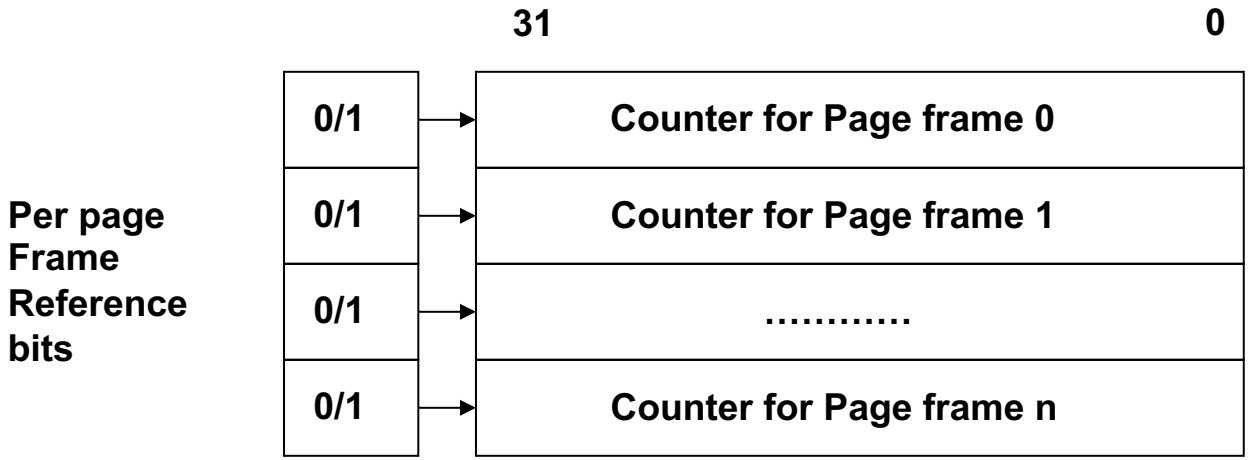
- Use a small hardware register stack
  - Remember the last 16 or 32 references?

- Reference bit per physical frame
  - Add a "referenced" bit to each PTE
  - Set it each time the hardware uses the PTE to translate a memory reference
  - If it's already set, don't set it again.

Page Table

PFN	Ref	Valid
PFN	Ref	Valid
PFN	Ref	Valid

Approximate LRU with ref bits

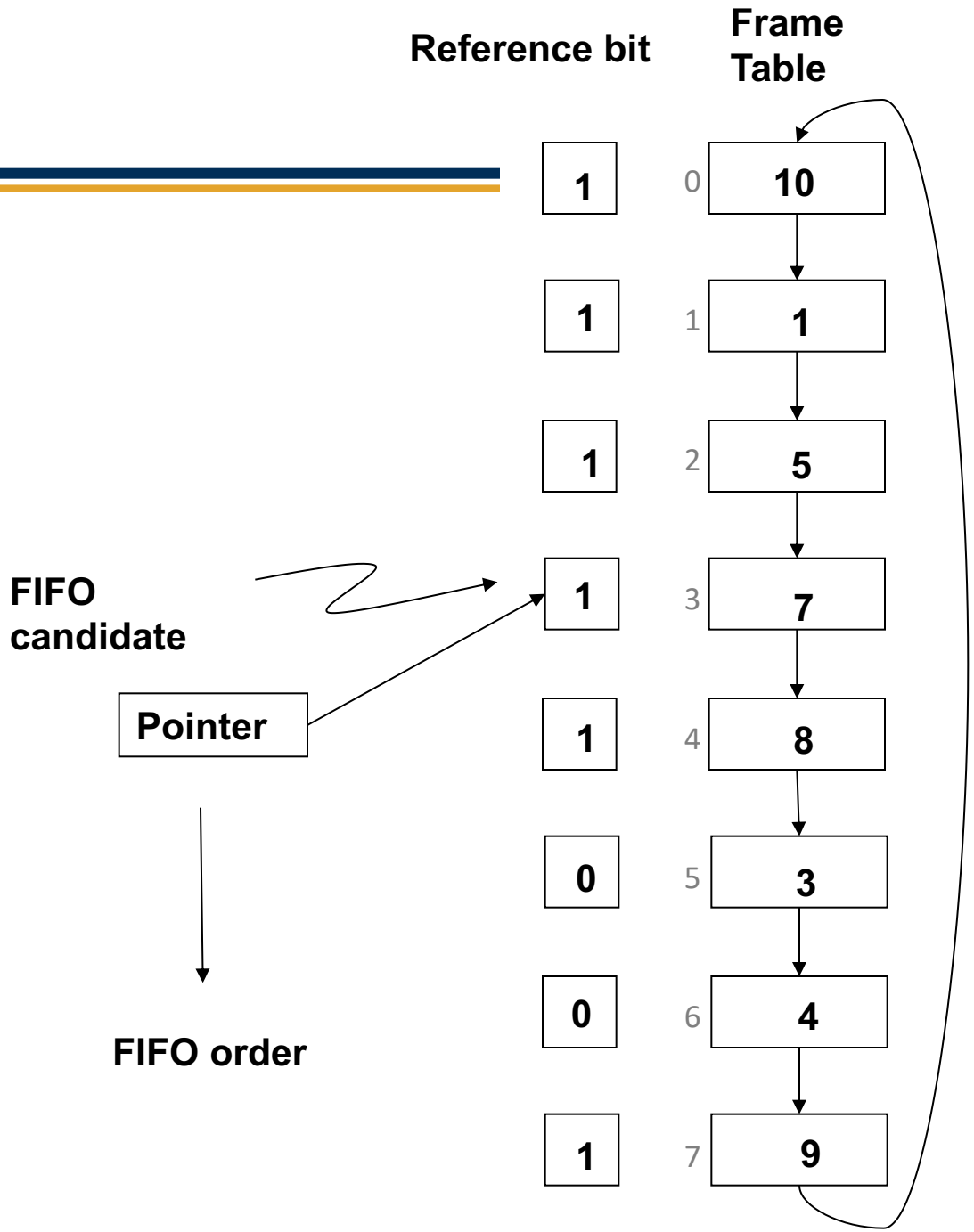


- Keep ref bits in PT
  - Set bit when page referenced → done by hardware
- Paging daemon → background OS process
  - Flush ref bits to software "counters" periodically
    counter = (ref << 31 | counter >> 1)
  - Clear ref bits
- Victim? → The page with the counter that has the lowest value

“Second chance” page replacement using reference bits

1. Initially clear all the reference bits
2. As the process runs, set referenced bits on each page referenced
3. If a page has to be evicted, the memory manager selects a page in a FIFO manner
4. If the chosen victim's referenced bit is set, the manager clears the referenced bit and moves to the next page
5. The victim is the first page that doesn't have the referenced bit set

Second chance replacement

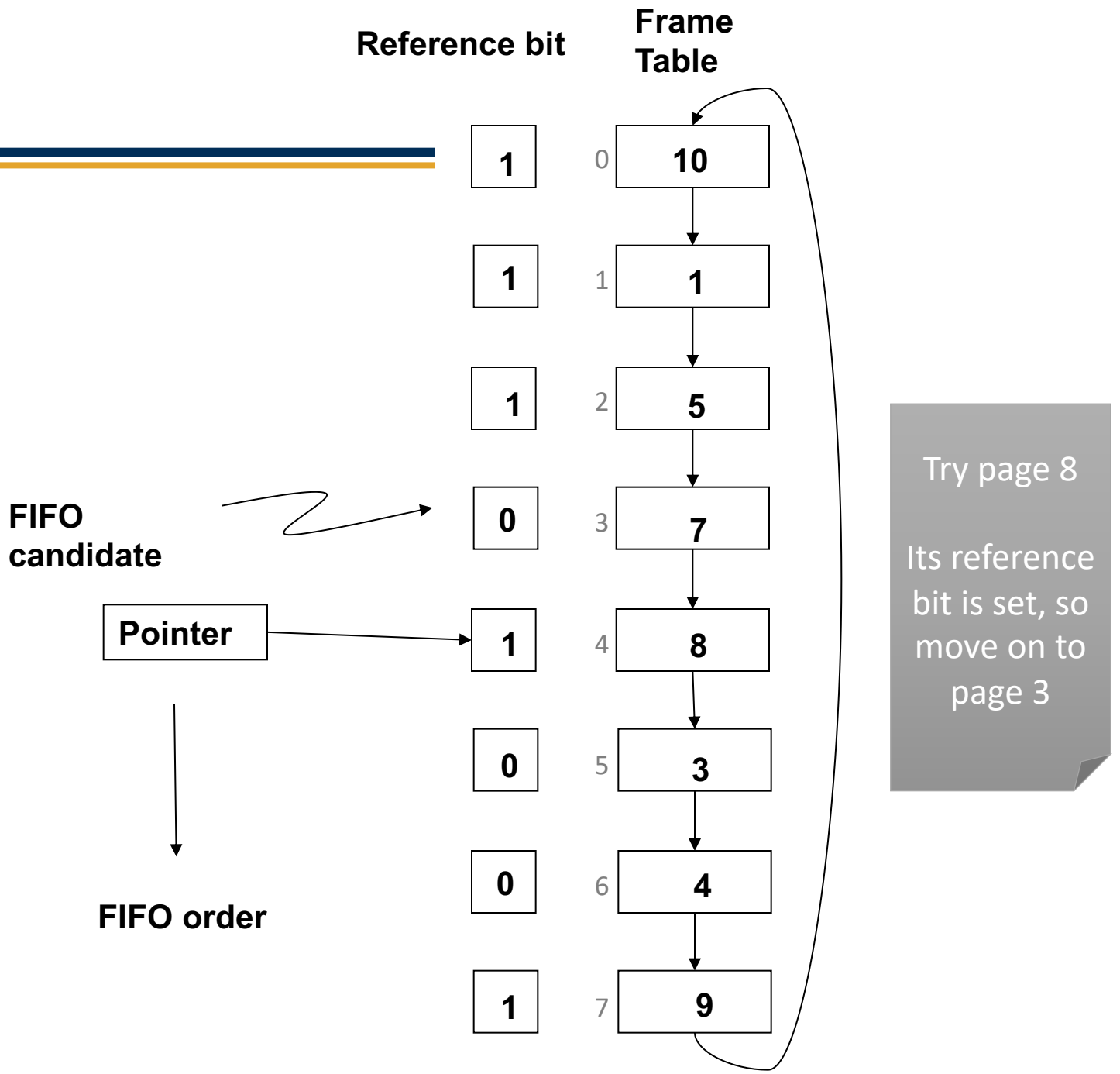


Access to VPN 2 page faults!

Try page 7

Its reference bit is set, so move on to page 8

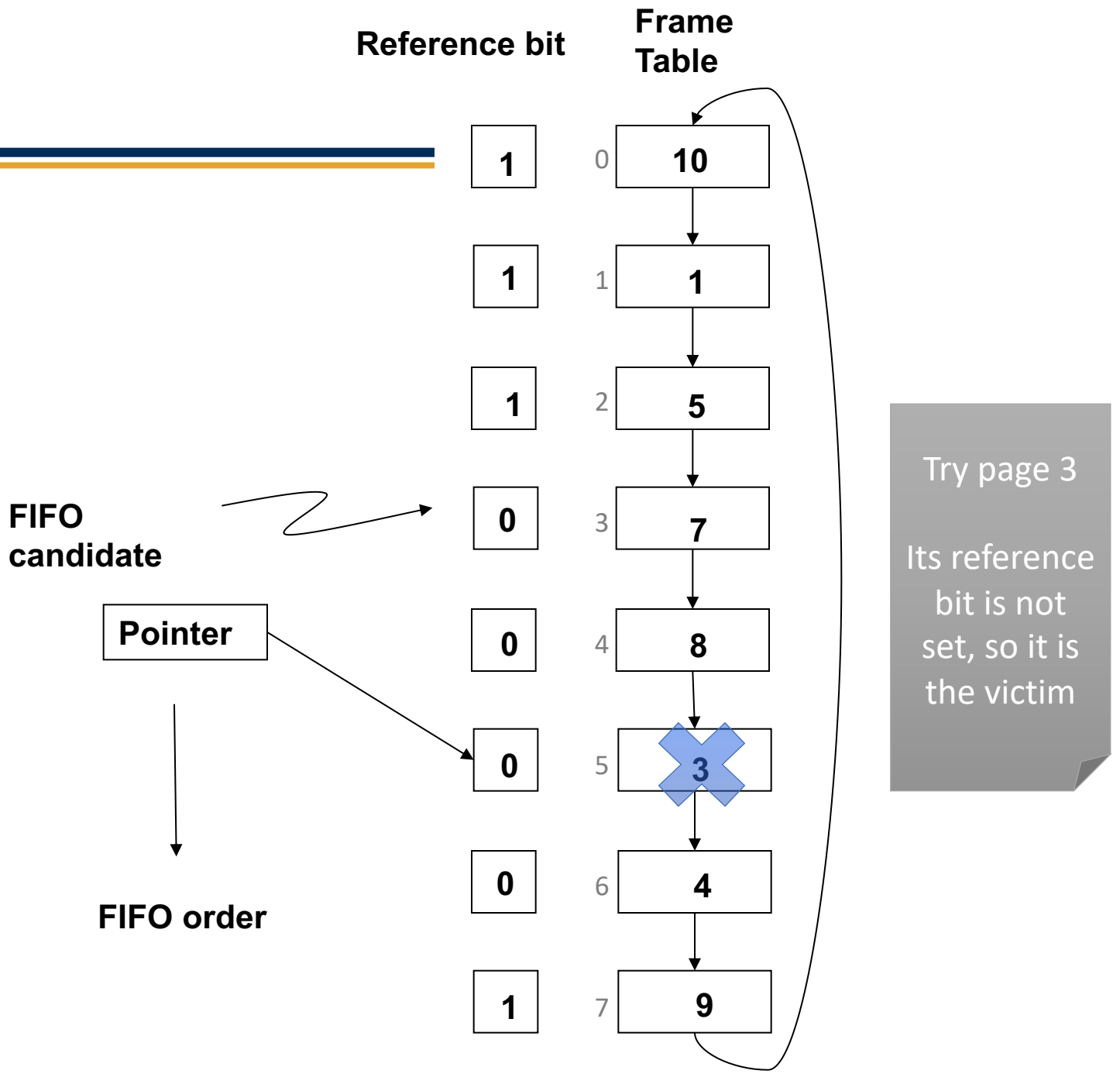
Second chance replacement



Try page 8

Its reference bit is set, so move on to page 3

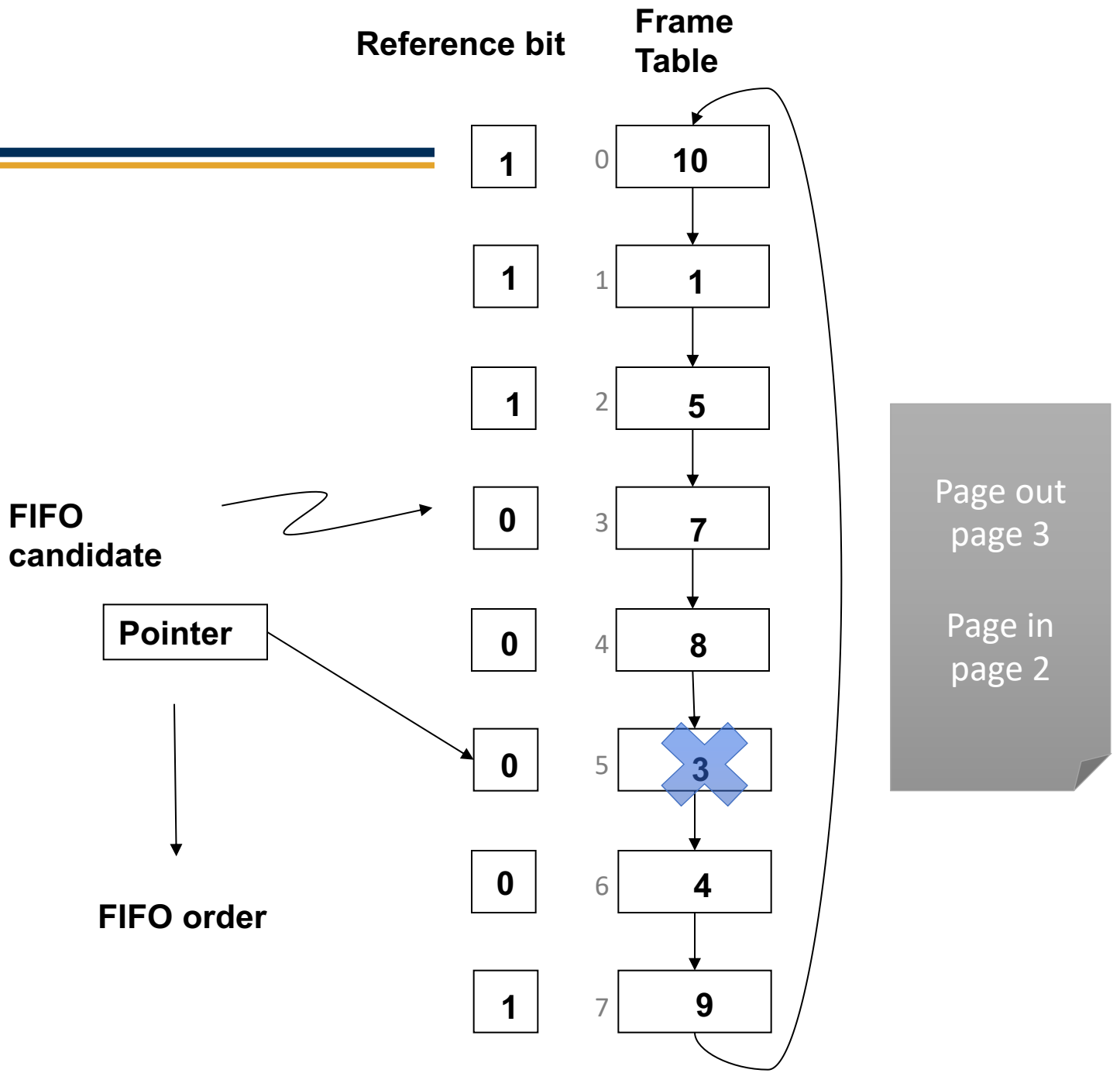
Second chance replacement



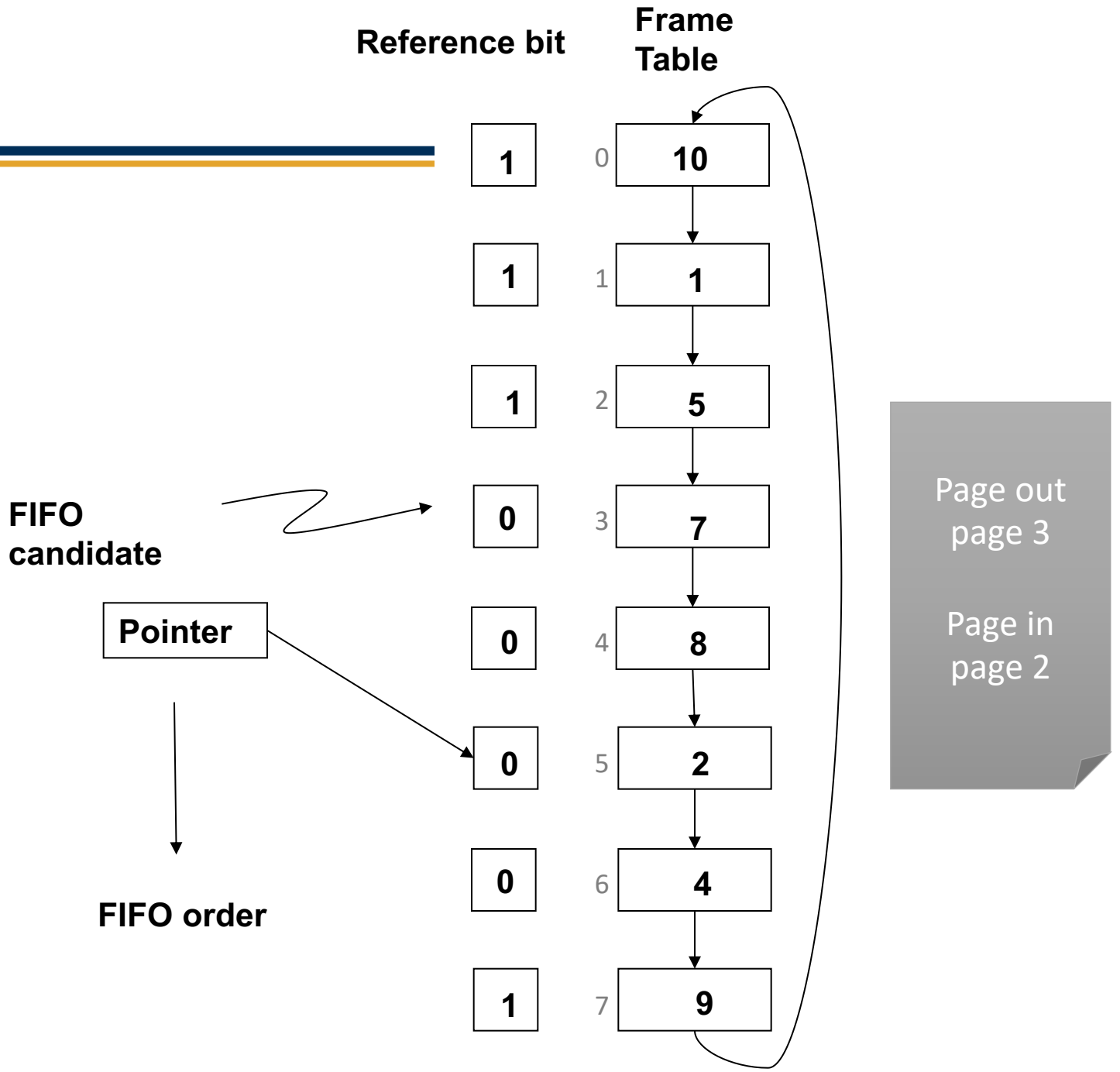
Try page 3

Its reference bit is not set, so it is the victim

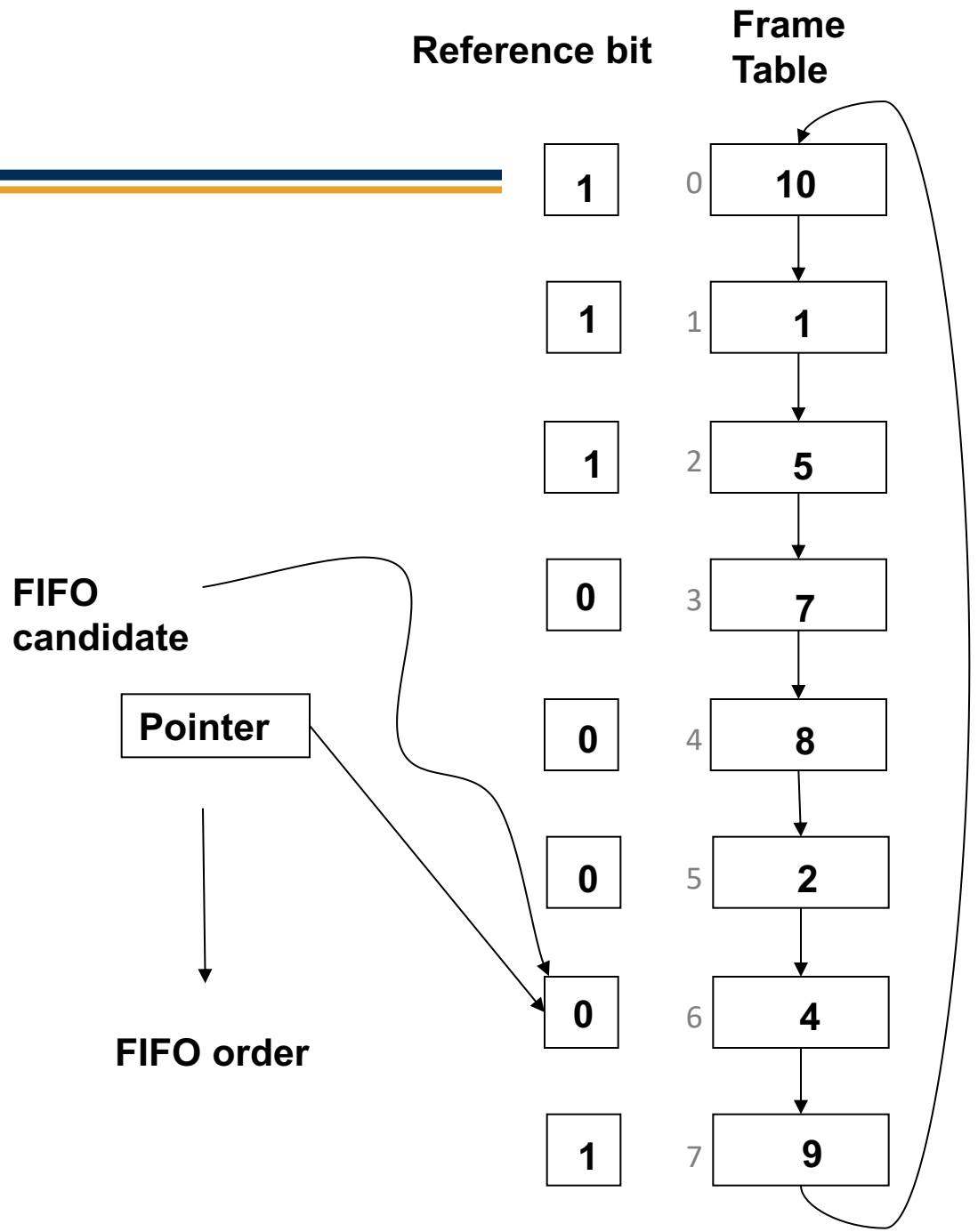
Second chance replacement



Second chance replacement



Second chance replacement



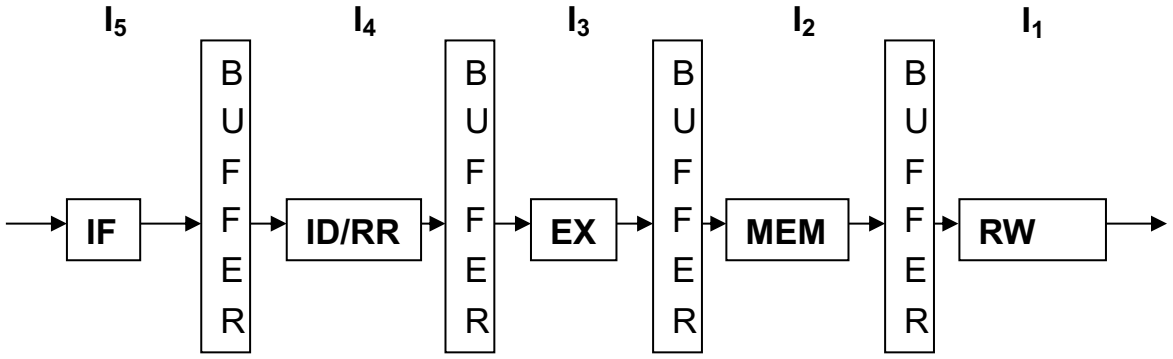
Move the pointer past the just-read page 2

Wait for the next page fault…

Page replacement algorithms

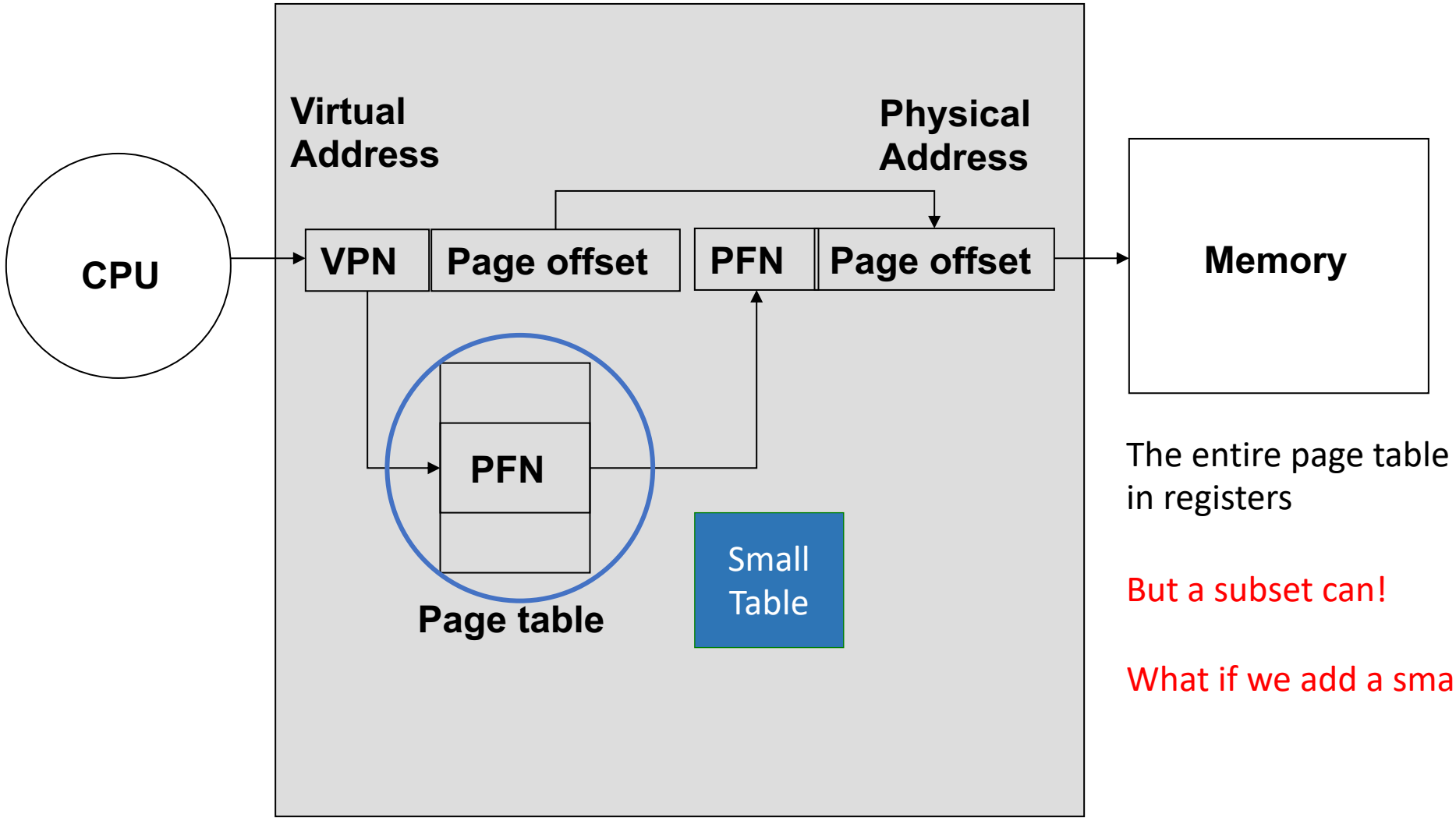
ALGORITHM	HARDWARE ASSIST	COMMENTS
FIFO	None	Could lead to performance anomalies
Belady's MIN	An oracle	Provably optimal; not realizable in hardware; useful as a standard
True LRU	Push down stack	Expected performance close to optimal; infeasible
Approximate LRU #1	A small hardware stack	Expected performance close to optimal; worst-case performance may be similar to FIFO
Approximate LRU #2	Reference bit per page	Expected performance close to optimal; moderate hardware complexity
Second chance replacement	Reference bit per page	Expected performance better than FIFO; memory manager implementation simplified compared to LRU schemes

Back to our pipelined processor



- With virtual memory…
- Every memory access requires two memory accesses!
  - PTE
  - Actual memory word
- This is bad news for the pipeline
- At least one bubble for every instruction

Speeding up address translation

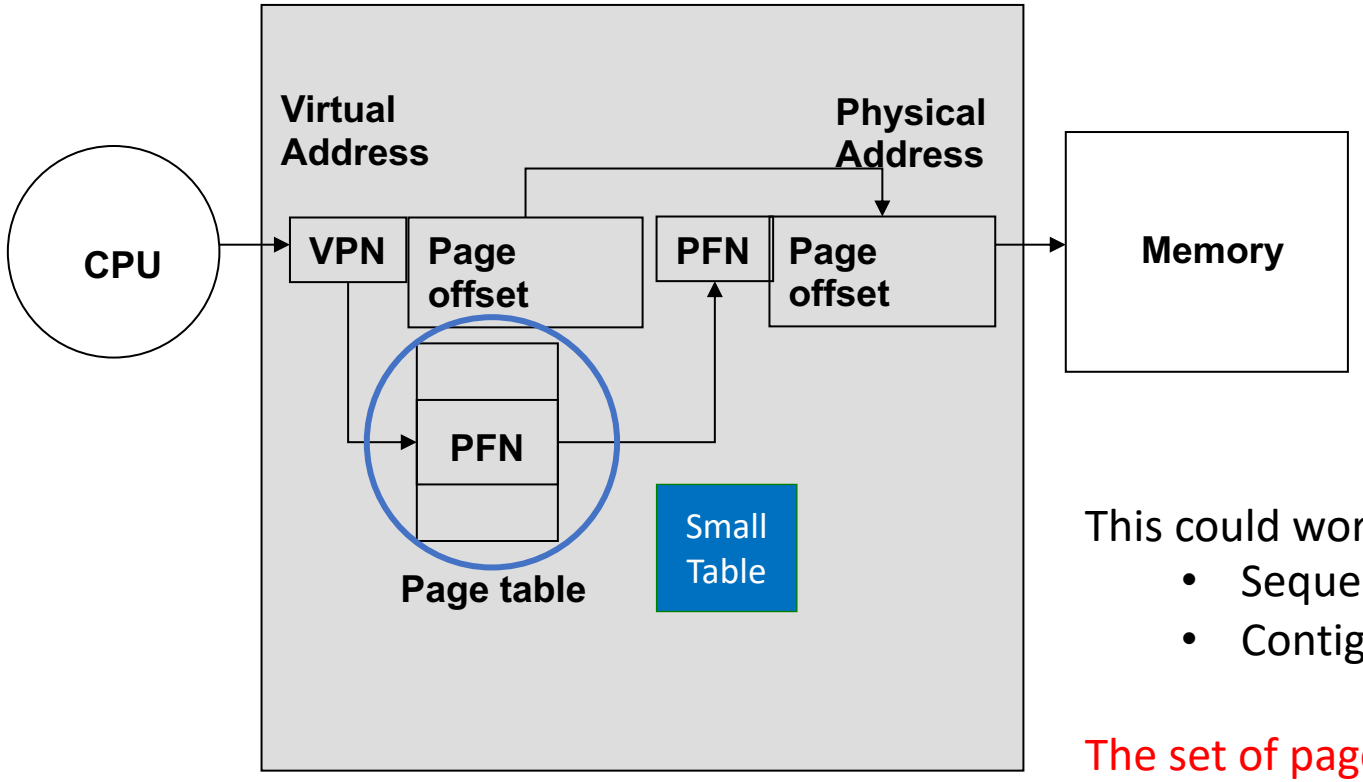


The entire page table won't fit in registers

But a subset can!

What if we add a small table?

Why will this work?



This could work because of the **locality** of a program

- Sequential instructions
- Contiguous data structures

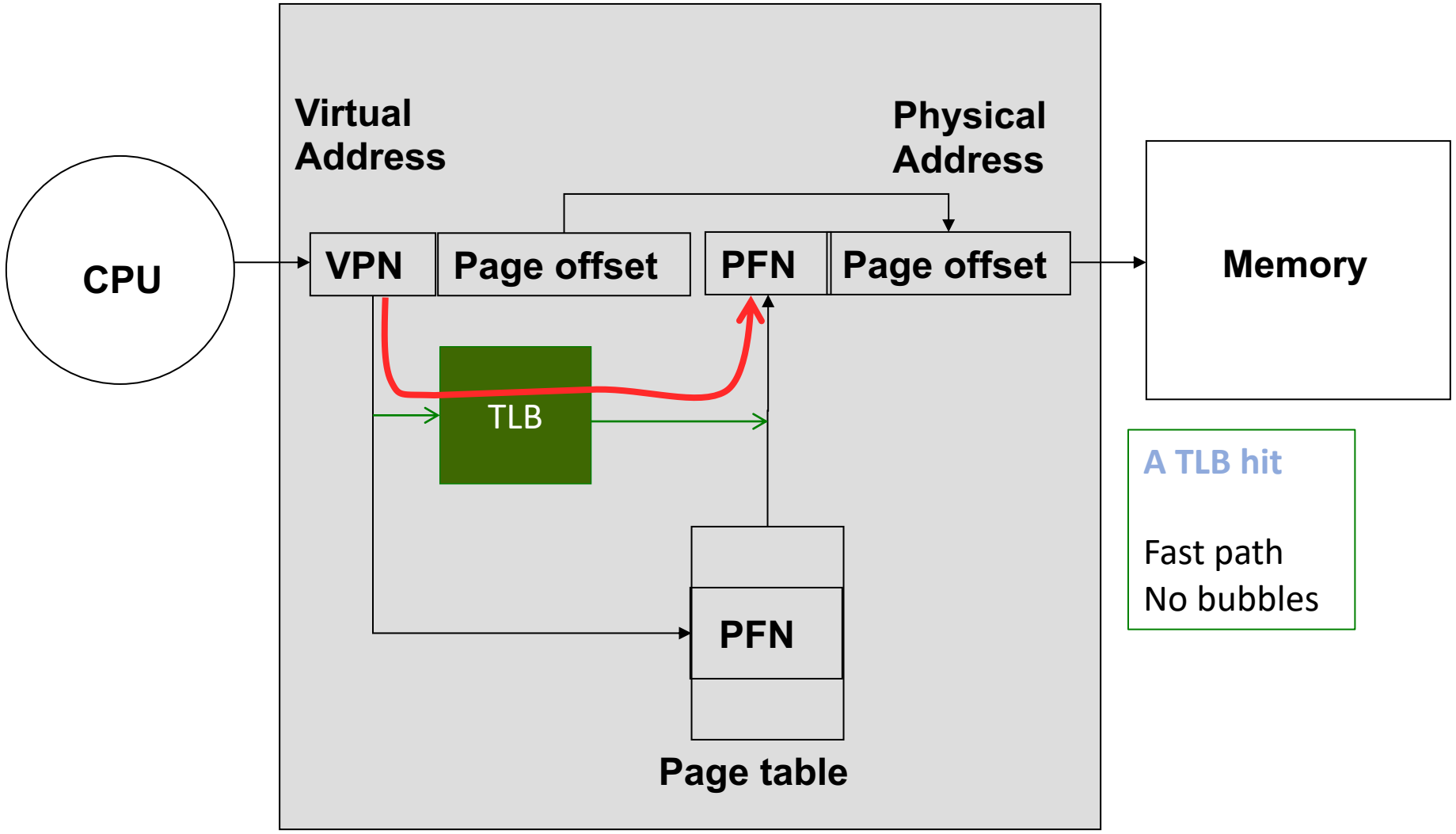
The set of pages being referenced at a given time is called the **Working Set**

TLB (translation lookaside buffer)

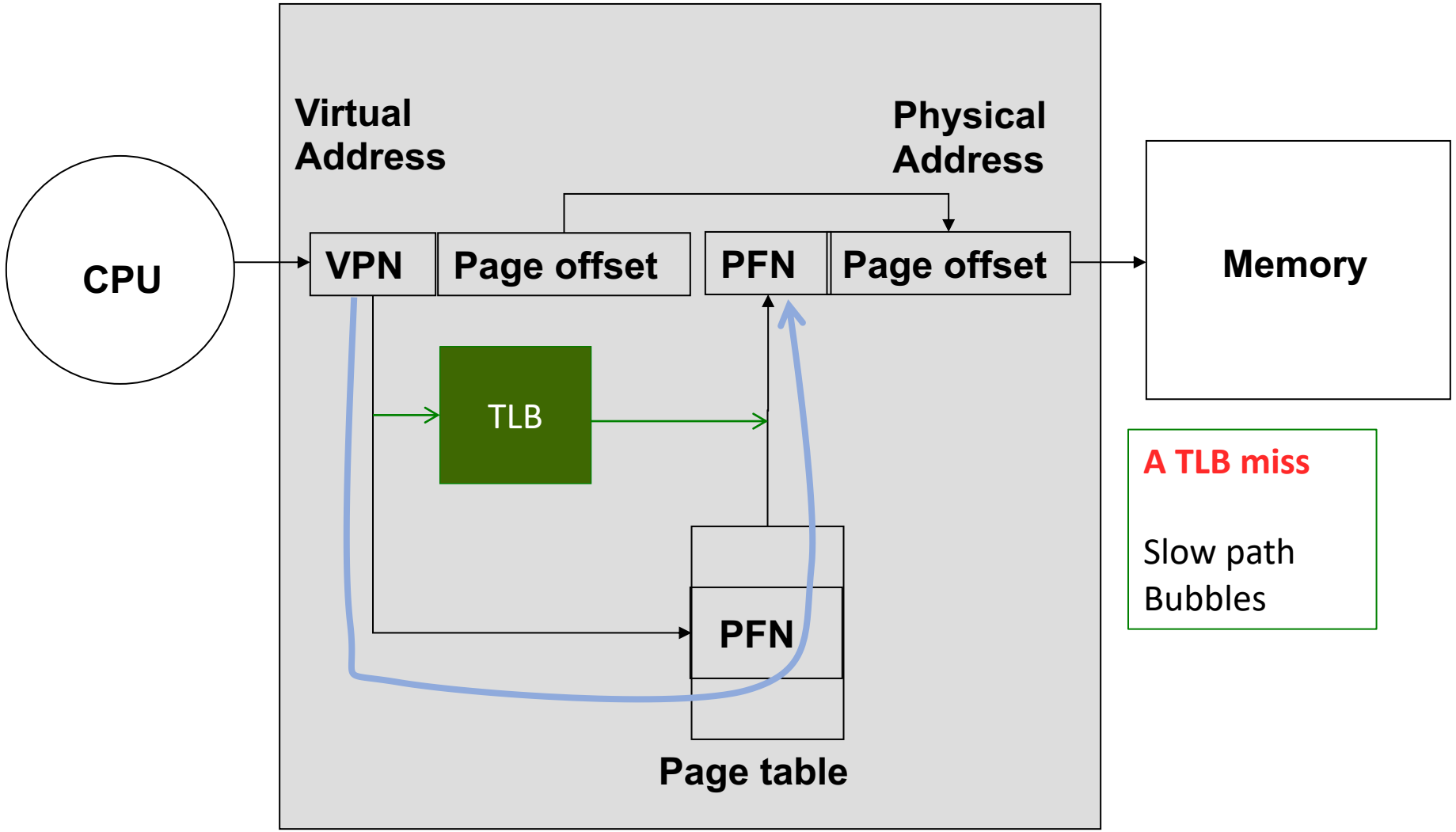
- It will look like the following table
- It is an **associative** memory: it can search on a match on the first two columns and output the corresponding last two columns in one cycle

USER/KERNEL	VPN	PFN	VALID/INVALID
U	0	122	V
U	XX	XX	I
U	10	152	V
U	11	170	V
K	0	10	V
K	1	11	V
K	3	15	V
K	XX	XX	I

Speeding up address translation



Speeding up address translation



TLB

Specific to each process

Same for all processes

USER/KERNEL	VPN	PFN	VALID/INVALID
U	0	122	I V
U	XX	XX	I I
U	10	152	I V
U	11	170	I V
K	0	10	V
K	1	11	V
K	3	15	V
K	XX	XX	I

What's the implication of the U entries for a context switch?

➔ We'll need to flush the U entries on context switch

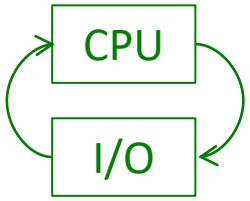
Another new instruction

- The LC-2200 is going to need
  - **Purge TLB**
  - or **TLB flush**
- Can only be executed by the kernel

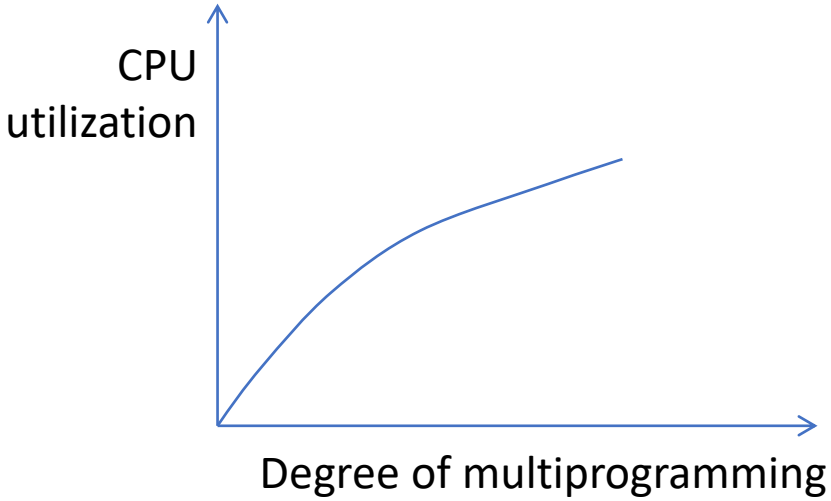
Upon a context switch...

A. I just want the participation credit
B. The entire TLB must be flushed
C. Only the kernel portion of the TLB must be flushed
D. Only the user portion of the TLB must be flushed
E. Leave the poor TLB alone!

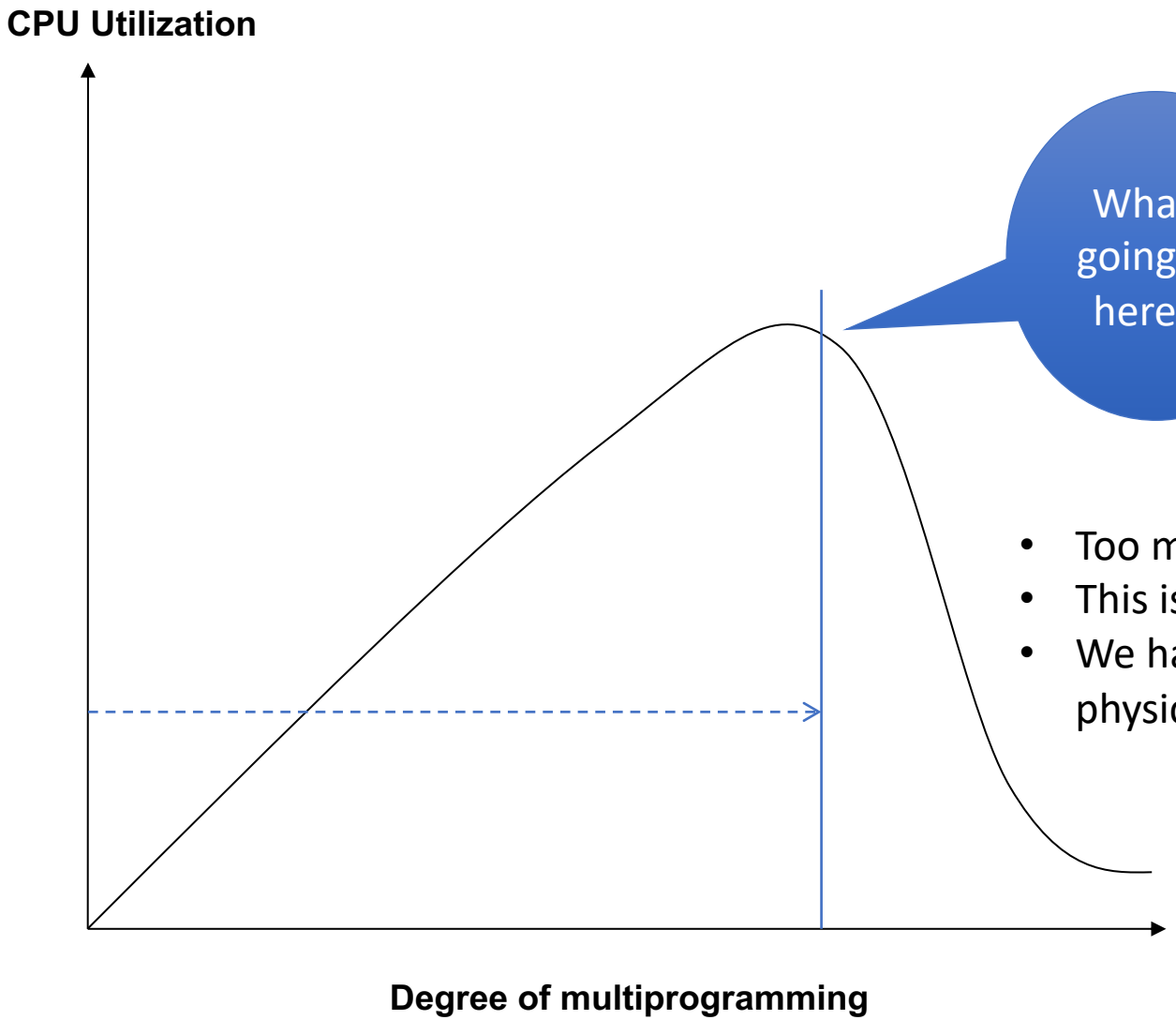
Given the nature of a process



- We want to increase multiprogramming to keep the CPU busy doing useful work
- This is what we want to see:



Extending the utilization curve



Thrashing

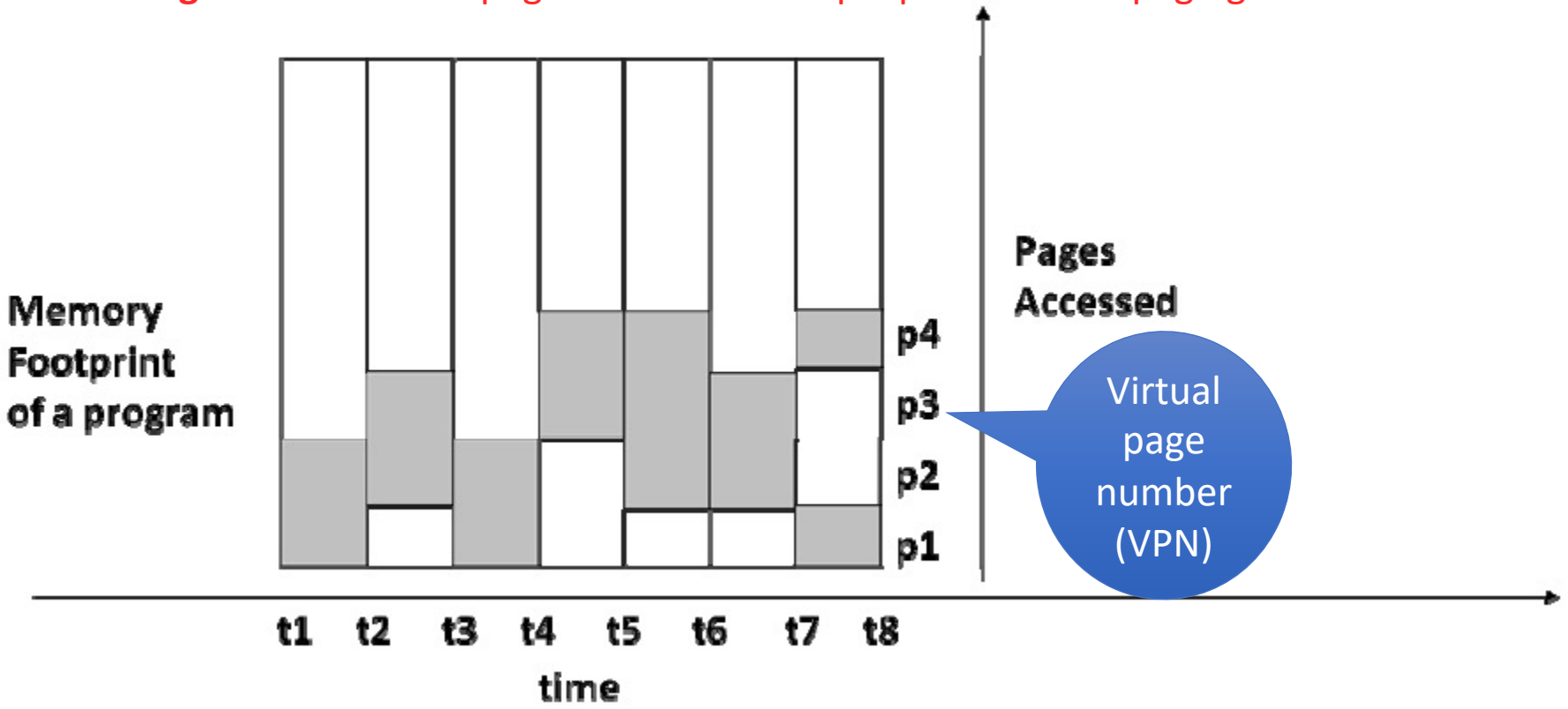
Nobody is getting any useful work done

- Too many processes asking for memory
- This is overcommitment of memory
- We have a bigger working set than physical memory can hold

Paging is implicit I/O on behalf of a process
→ System became I/O bound
→ Throughput drops precipitously

Working set of a program

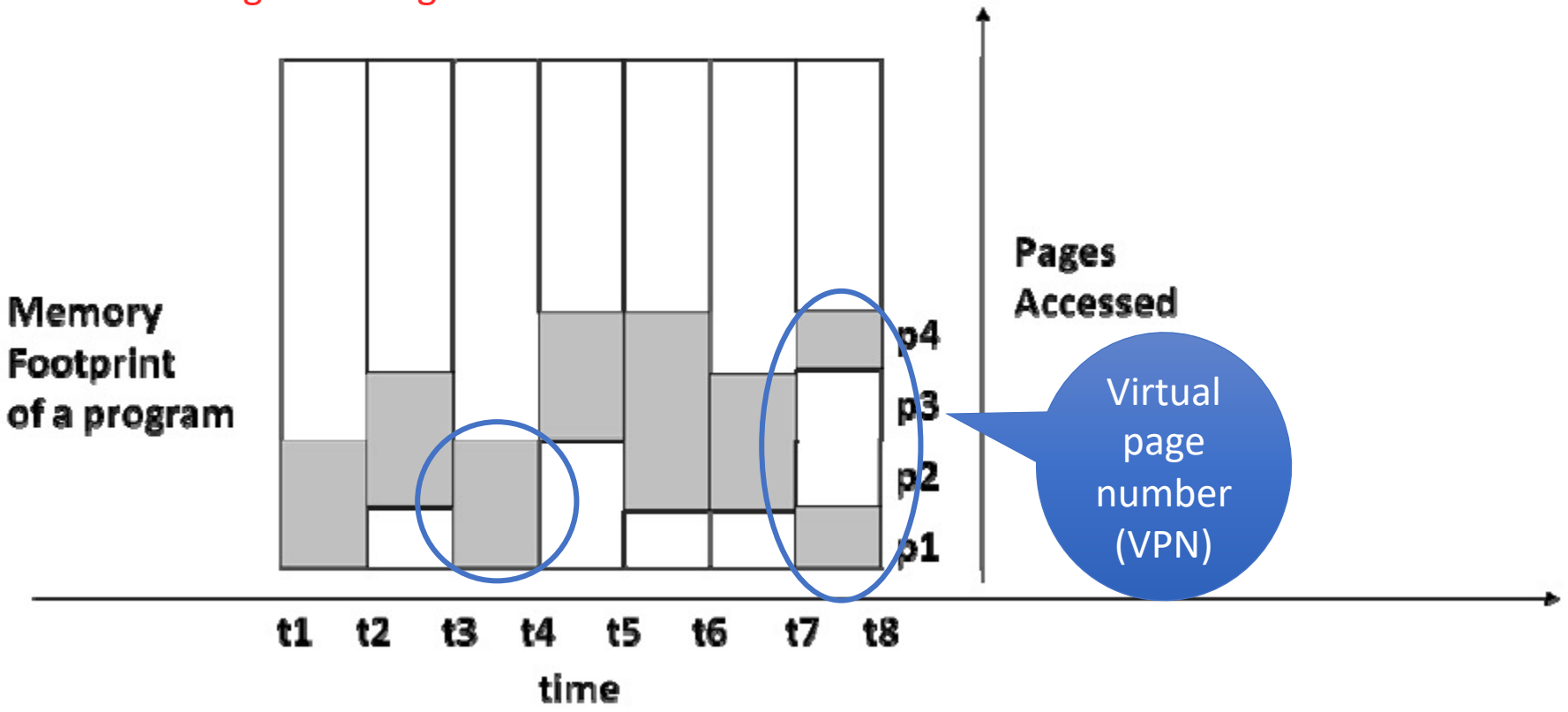
Working set is the set of pages needed to keep a process from paging



Working set size: number of page frames needed to hold working set

Working set of a program

The working set changes over time



$WS_{t3\text{-}t4}$ = { p1, p2 }

$WS_{t7\text{-}t8}$= { p4, p1 }

Memory pressure

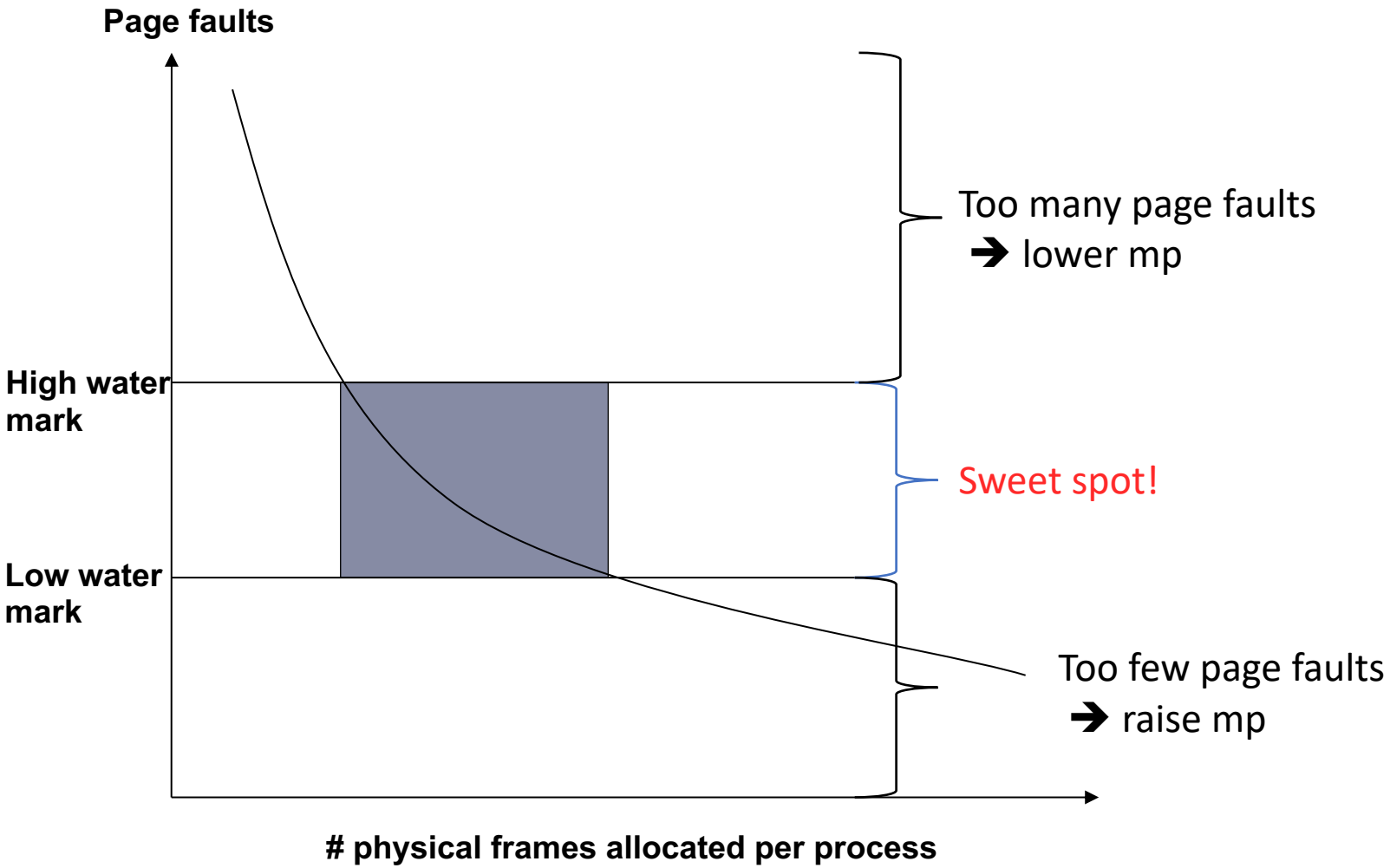
$$memory.pressure = \sum_{i=1}^{n} WSS_i$$

- $P_1, P_2, P_3, \ldots$ are processes in memory each with a working set WSS_i
- The count of active processes signifies the degree of multiprogramming
- How do we control the degree of multiprogramming
  - **∑WSS > total physical memory**

    ➔ swap out some processes
  - **∑WSS < total physical memory**

    ➔ increase degree of multiprogramming

Controlling thrashing



Page faults are disruptive…

- … from a process point of view
  - ➔ implicit I/O
- … from a CPU-utilization perspective
  - ➔ overhead that doesn't contribute to work

- We need to limit impact of page faults to improve system performance

We can tell a system is thrashing if

A. I just want the participation credit
B. It has too few page faults per second
C. It has too many page faults per second
D. The ratio of I/O operations to CPU operations is not optimal
E. The combined working set of all processes is greater than the number of available page frames

If only it were as easy as B! Thrashing implies too many page faults, but too many page faults don't always imply thrashing! Applications can be changing the pages in use without changing their working set size, for instance.

In reality, to diagnose thrashing, you'd look for a high paging rate, low CPU utilization, and several processes waiting on paging I/O for several seconds. Those metrics together are a good clue.

Session ID: dag2200

We can reduce thrashing by

A. I just want the participation credit
B. Using a medium-term scheduler that suspends processes until the condition improves
C. Reducing the physical memory size
D. Adding additional processes to increase the multiprogramming factor
E. Reducing the page size

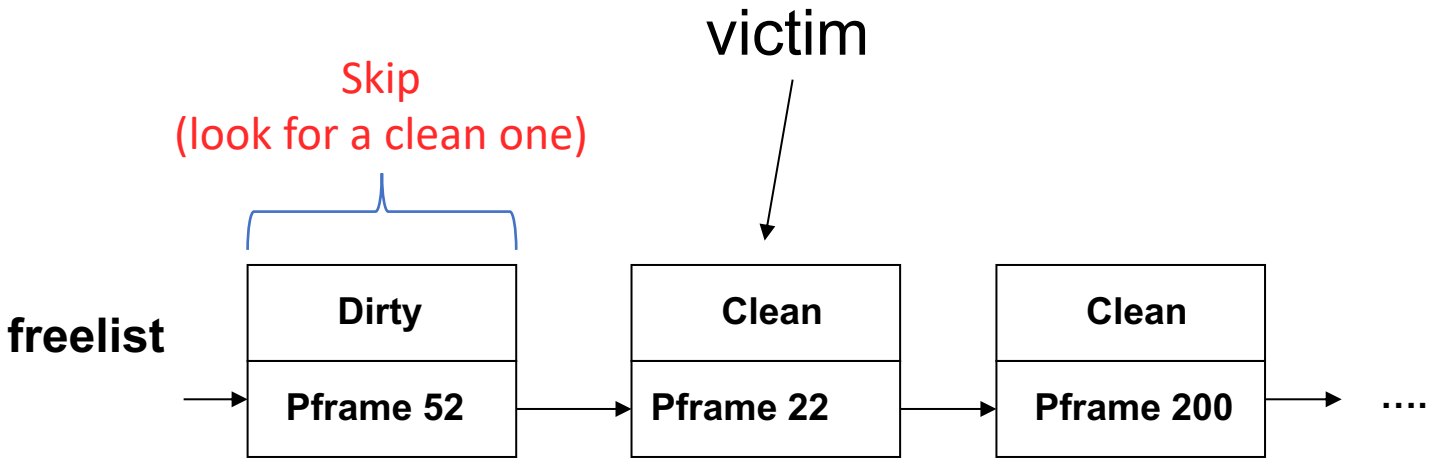
Of course this begs the question of how the medium-term scheduler is going to figure out that the system is thrashing…

Paging Optimizations

Accelerate page-in by removing functionality from critical path

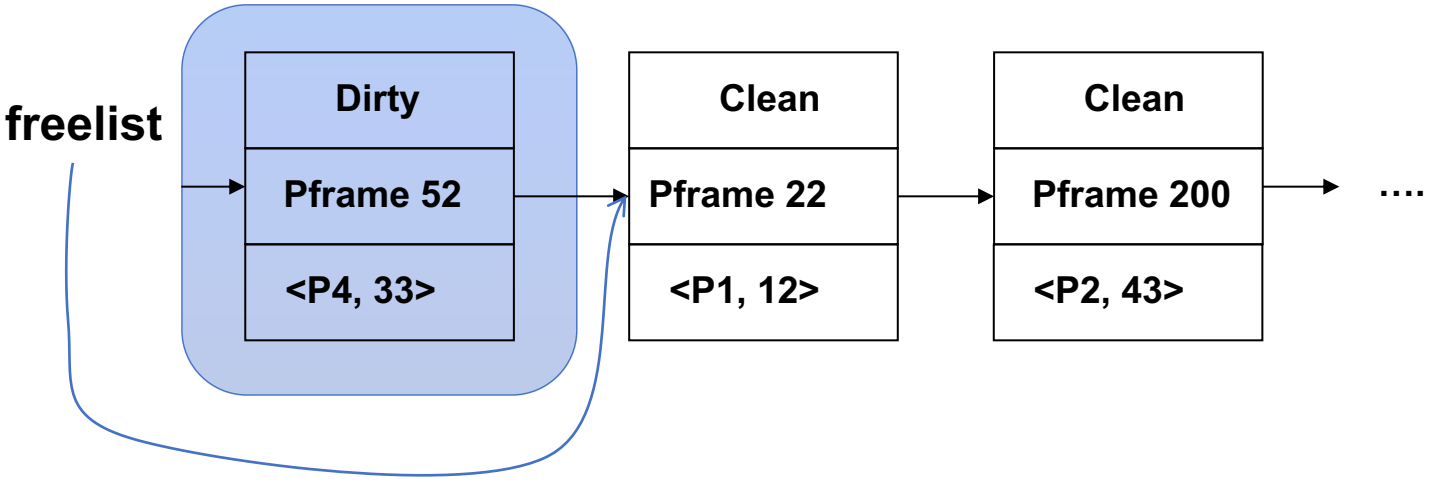
- Keep a (small) pool of free frames
  - Don't wait to start page replacement algorithm on a page fault

- Page replacement
  - Background activity of OS when CPU is not in use
  - If I/O is not busy, write out a "dirty" page which makes it "clean"



Reverse mapping to page table

- Gives a "third" chance for reuse of a page before being kicked out
- P4 is running and page faults on VPN=33
- No need to go to disk!
- Just remap PFN=52 into PT for P4, VPN=33 and take it out of the freelist



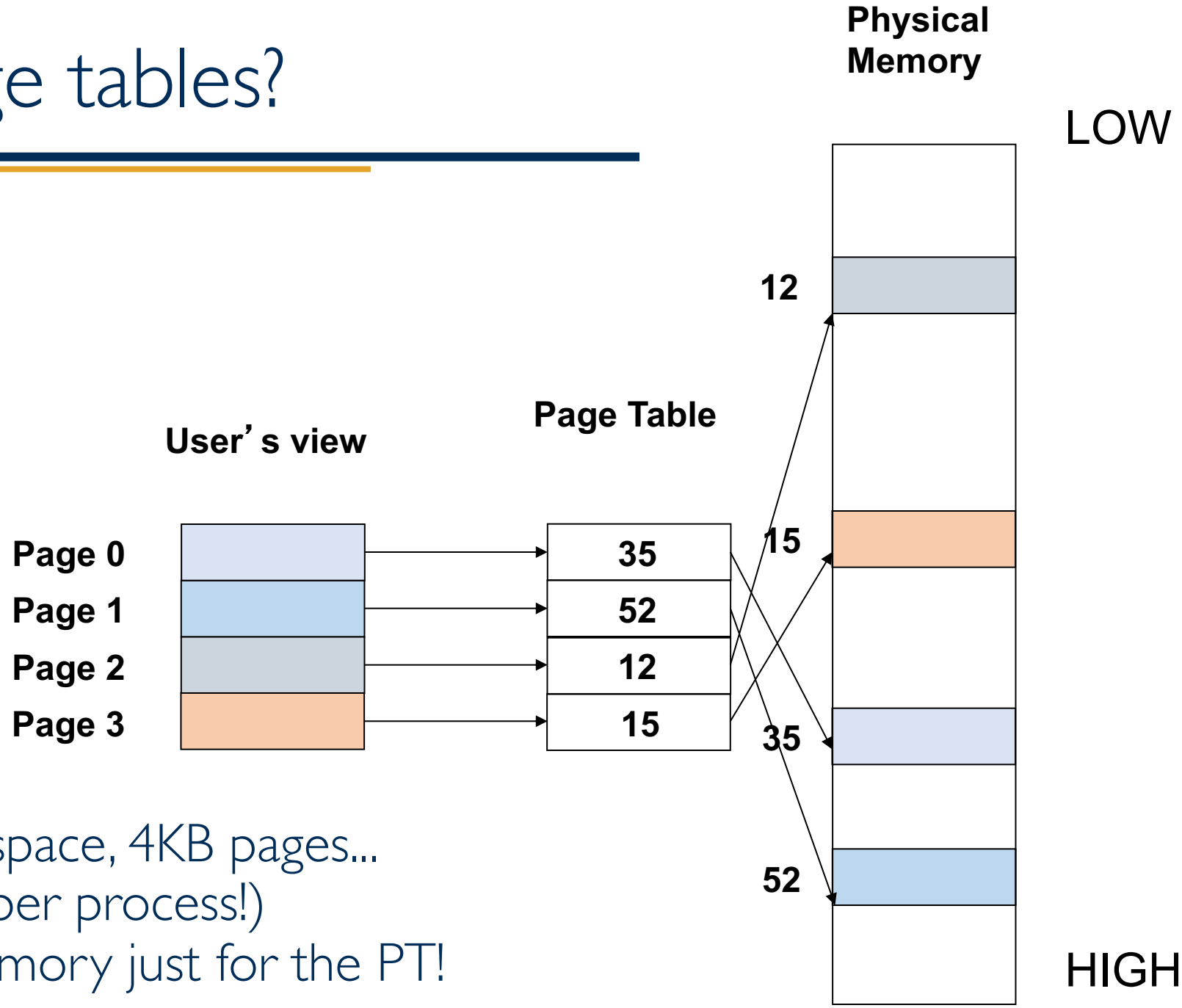
Linux VM and kswapd

```
$ free -h
             total       used       free     shared    buffers     cached
Mem:           15G       7.1G       8.5G       164K       703M       2.4G
…
-/+ buffers/cache:       4.0G        11G
Swap:         2.0G        26M       1.9G
```

- Kswapd
  - Paging daemon
  - Runs when "free" memory is low (about 2% of memory)
  - Uses a modified version of second-chance replacement
  - Links victim pages into the free list and sets their "invalid" PTE bits

- Page fault handler:
  - If the target page is still on the free list, it is reclaimed by removing it from the free list, marking its PTE bit "valid", and writing it out if it's dirty
  - Otherwise, the first frame in the Free List is removed, the target page is read into it, and the target page's PTE is modified to point to it and "valid" is set

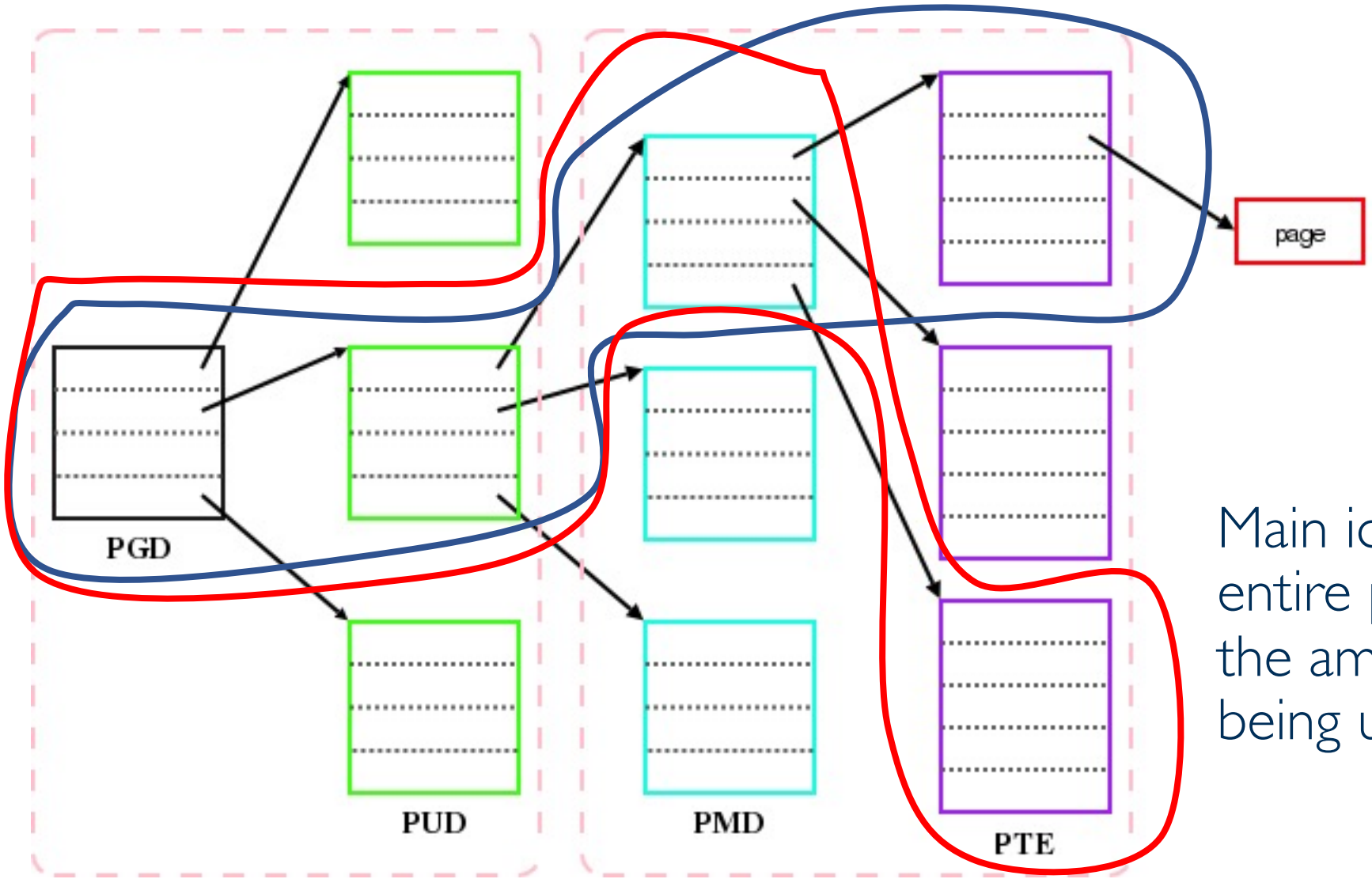
How big are page tables?



With 48-bit virtual address space, 4KB pages...

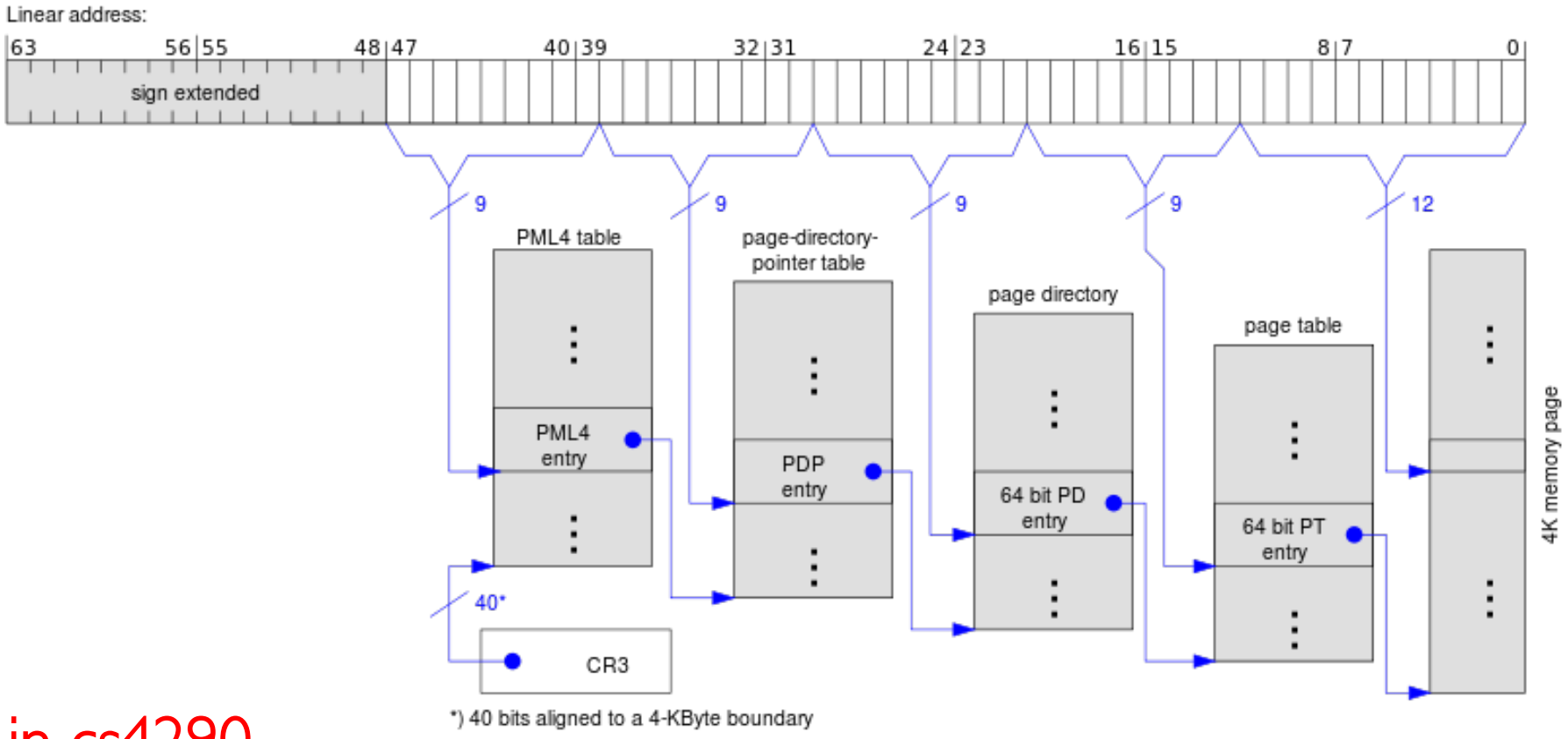
- Need 2^36 PT entries (per process!)
- That's several GBs of memory just for the PT!

Teaser: Hierarchical Page Table



Main idea: only load a fraction of entire page table, analogous to the amount of VM that is actually being used by the process

Intel's X86-64 4-level Page Tables



More about this in cs4290