Georgia Tech Computer Science

CS2200
Systems and Networks
Spring 2022

Lecture 16:
Virtual Memory pt 2

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

- Don’t forget to sign up for Project 2 demo

Recap

Hardware	Software
Fence register	User/kernel split
Bounds register	Static relocation
Base + limit register	Dynamic relocation

Next
Allocation policies
Paging

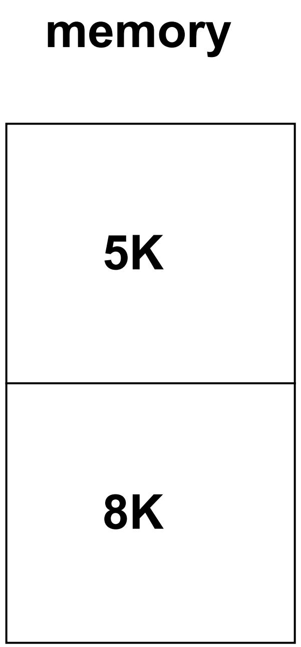
Memory allocation by OS

- Fixed size partition
- Variable size partition
- Both use the hardware base + limit registers

Fixed size partitions

OS memory manager allocation table

Occupied bit	Partition Size	Process
0	5K	---
0	8K	---



memory

5K

8K

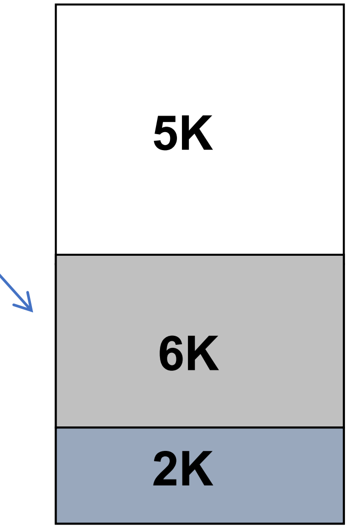
```
Struct AT_entry {
                int occupied;
                int size;
                int pid;
};
```

P1 needs 6K of memory

Allocation table

Occupied bit	Partition Size	Process
0	5K	---
1	8K	P1

Memory



P1 needs 6K of memory

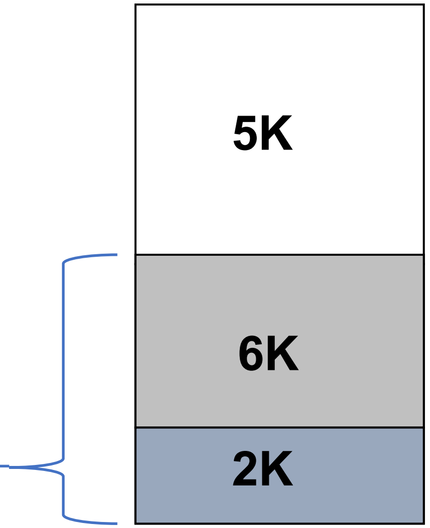
Allocation table

Table size is fixed = number of partitions

Occupied bit	**Partition Size**	**Process**
0	**5K**	**---**
1	**8K**	**P1**

Needs only 6K

Memory



Internal fragmentation

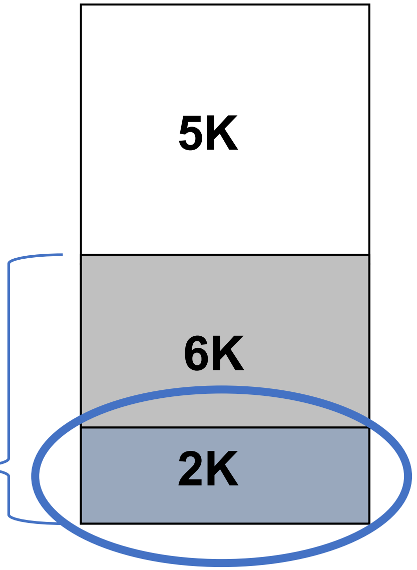
Internal fragmentation = size of partition – actual used

Allocation table

Occupied bit	Partition Size	Process
0	5K	---
1	8K	P1

Needs only 6K

Memory



Wasted (internal fragmentation)

Another process needs 6K memory

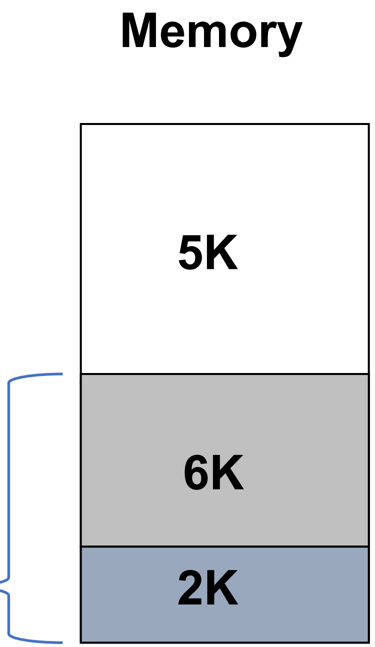
Do we have it?

Memory manager has only a 5K partition…

Not possible

Allocation table

Occupied bit	Partition Size	Process
0	5K	---
1	8K	P1

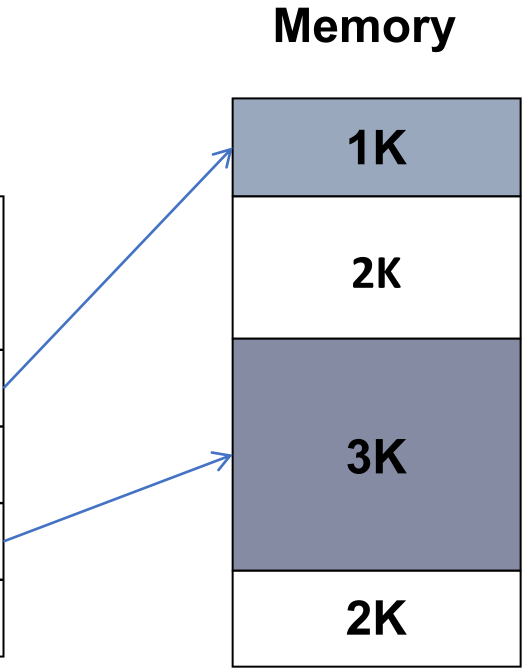


External fragmentation

Consider P3 that needs 4K of memory
Is it possible to allocate?
Memory is available, but not contiguous

Allocation table

Occupied bit	Partition Size	Process
1	1K	P1
0	2K	---
1	3K	P2
0	2K	---



External fragmentation
= ∑ All non-contiguous free memory partitions
And we have 4K of non-continuous memory here
Which gives us 4K of **external fragmentation**

Fragmentation

- Internal fragmentation
  - Size of partition – actual memory used
- External fragmentation
  - ∑ **All non-contiguous free partitions**

Fixed size partition memory management

Pros

- Simplicity

Cons

- Fragmentation
    - Internal
    - External

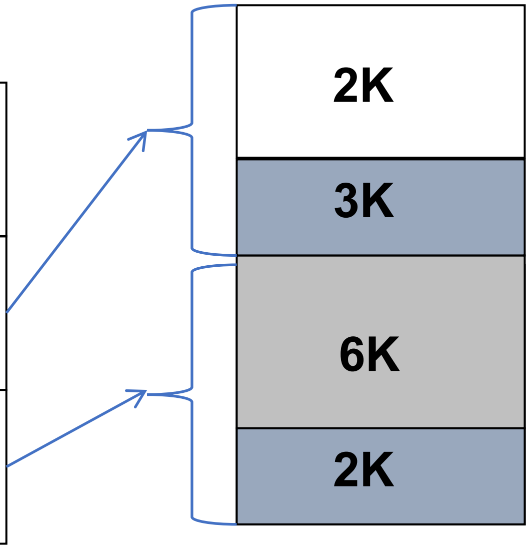
Session ID: dag2200

Total **internal** fragmentation is…

Allocation table

Occupied bit	Partition Size	Process
1	5K	P2 (need 2k)
1	8K	P1 (need 6K)

Memory

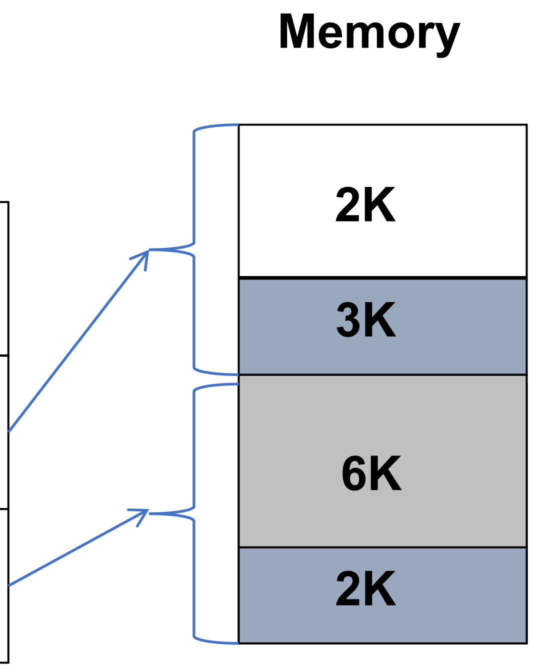


A. 2K
B. 3K
C. 5K
D. 8K

Total **external** fragmentation is…

Allocation table

Occupied bit	Partition Size	Process
1	5K	P2 (need 2k)
1	8K	P1 (need 6K)



A. 0K
B. 2K
C. 5K
D. 8K

Overcoming internal fragmentation

- Allocate exactly what is needed
- Variable size partitions

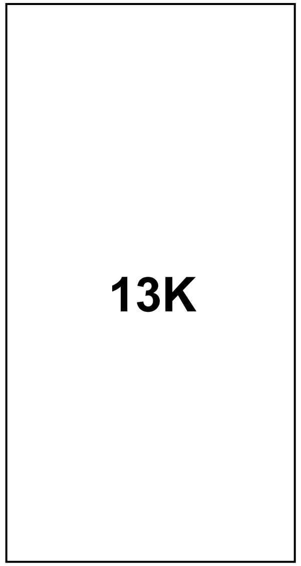
Variable size partitions

Memory manager allocation table

Start address	Size	Process
0	13K	FREE

```
Struct AT_entry {
        int start;
        int size;
        int pid;
};
```

memory

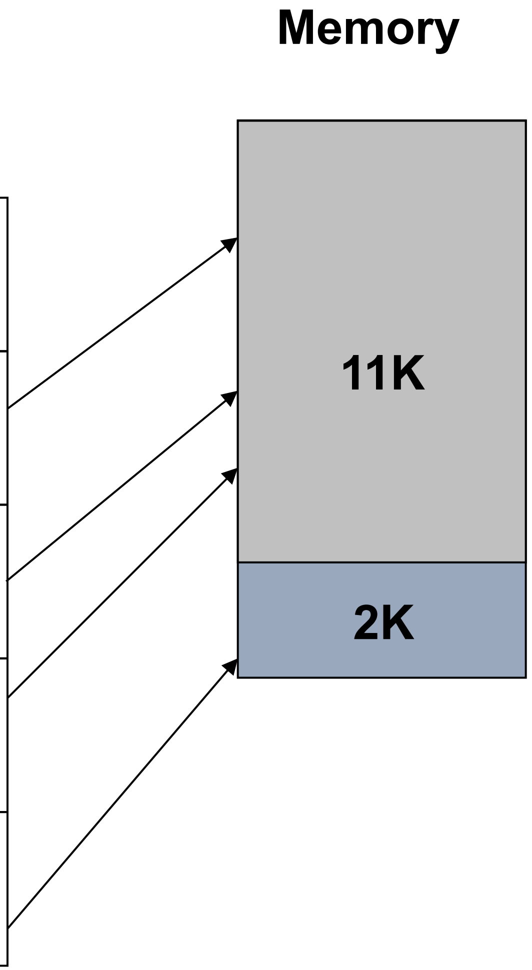


Partition table a little while later

Grows and shrinks as partitions get created and released
→ No internal fragmentation!

Allocation table

Start address	Size	Process
0	2K	P1
2K	6K	P2
8K	3K	P3
11K	2K	FREE

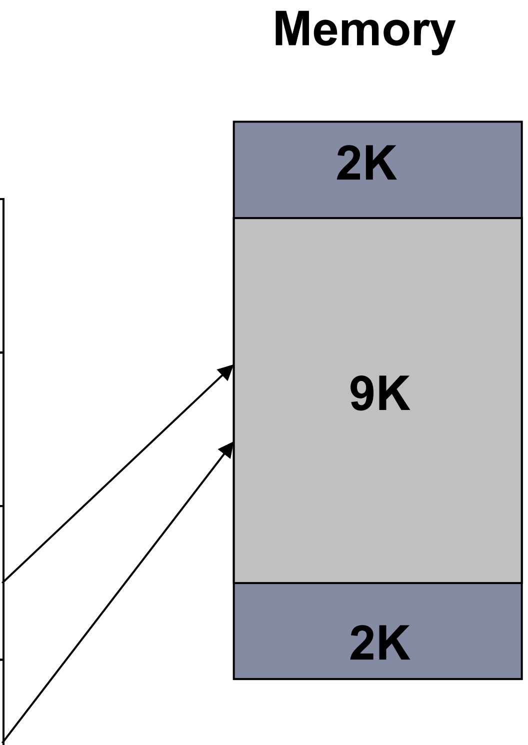


P1 exits

Allocation table

Start address	Size	Process
0	2K	P1 → FREE
2K	6K	P2
8K	3K	P3
11K	2K	FREE

External fragmentation

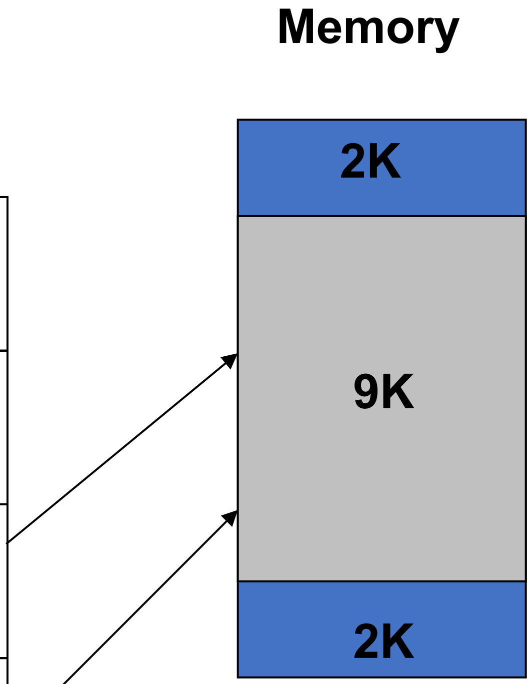


New request:
P4 needs 4K
Possible?

P2 exits

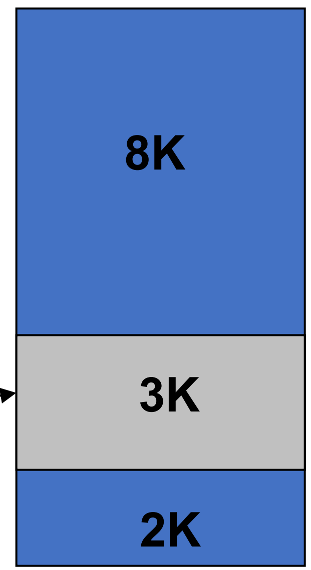
Allocation table

Start address	Size	Process
0	2K	FREE
2K	6K	P2 → FREE
8K	3K	P3
11K	2K	FREE



Coalescing two free areas

Memory



Allocation table

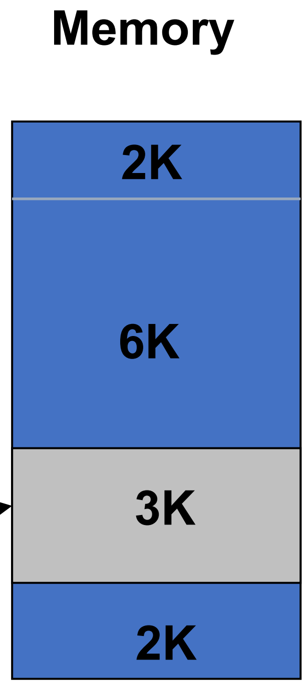
Start address	Size	Process
0	8K	FREE
8K	3K	P3
11K	2K	FREE

Reducing external fragmentation

- Best fit algorithm
  - A little better memory utilization
- First fit algorithm
  - A little quicker but less space efficient in the average case

Allocation table

Start address	Size	Process
0	8K	FREE
8K	3K	P3
11K	2K	FREE

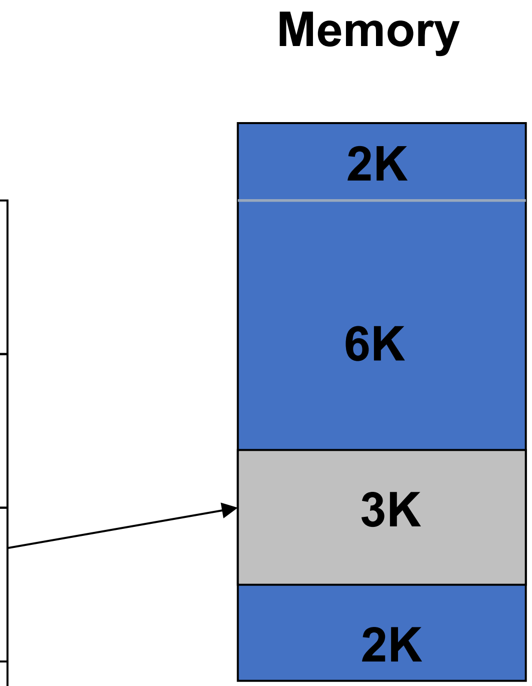


Compaction

- Request for 9K

Allocation table

Start address	Size	Process
0	8K	FREE
8K	3K	P3
11K	2K	FREE



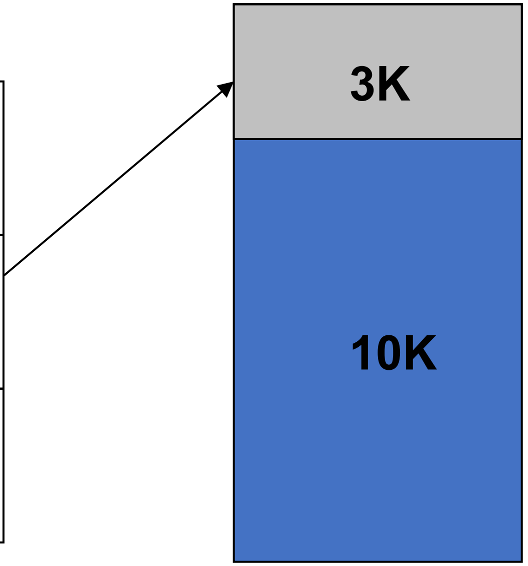
Compaction

- Relocate P3
- Create contiguous space
- Note this is a rather expensive action

Allocation table

Start address	Size	Process
0	3K	P3
3K	10K	FREE

Memory



Session ID: dag2200

With variable size partition memory management there is …

A. No external fragmentation

B. No internal fragmentation

C. No fragmentation

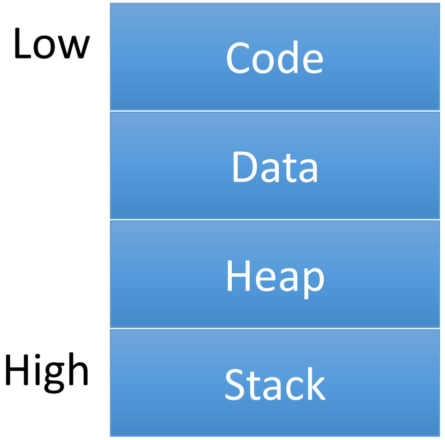
D. Both internal and external fragmentation

External fragmentation with variable size partitions

- Can limit full usage of memory resources
- Compaction is too expensive

How might we solve the external fragmentation problem?

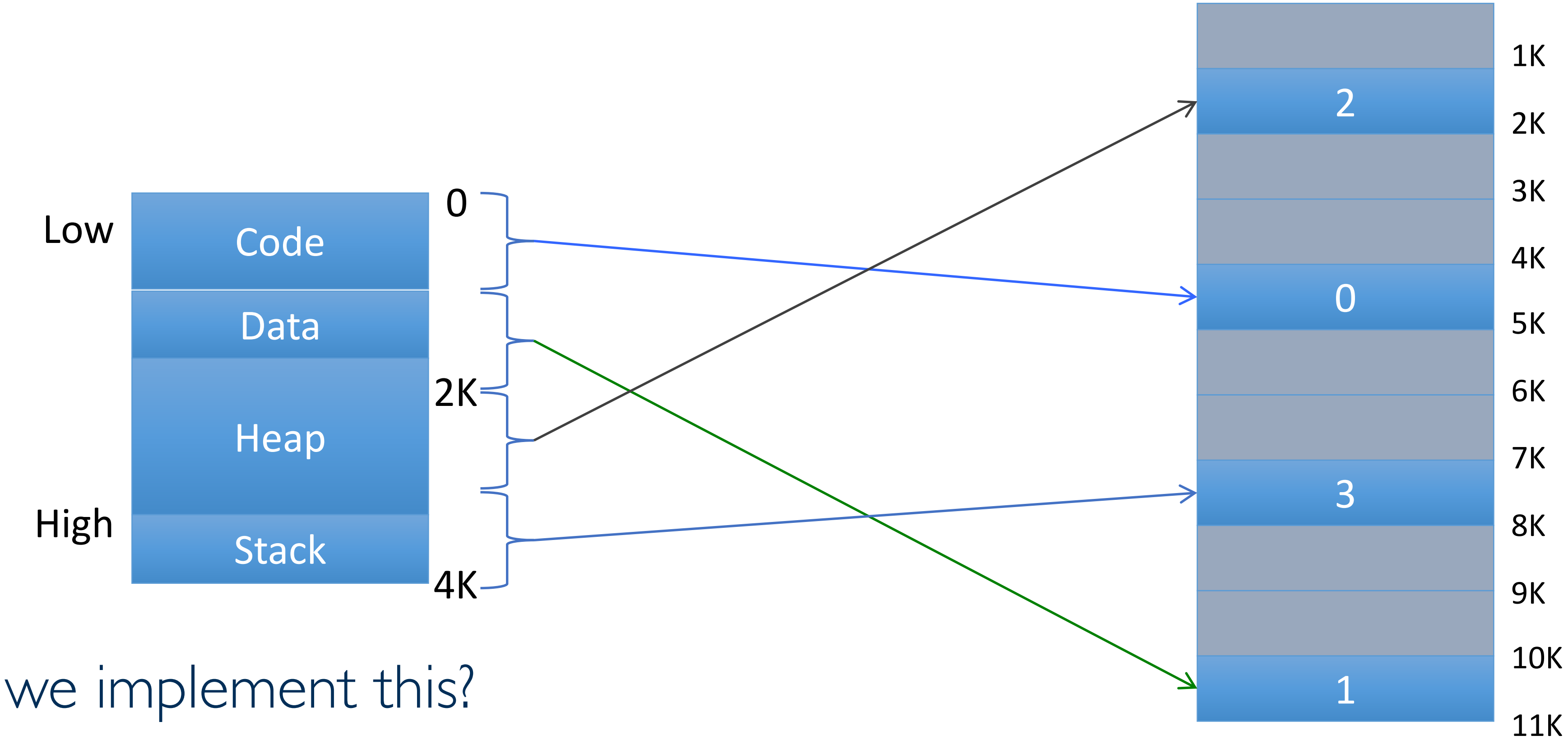
- Our process' memory footprint looks like this



- What's the main limiting assumption?
- That the process address space is contiguous!

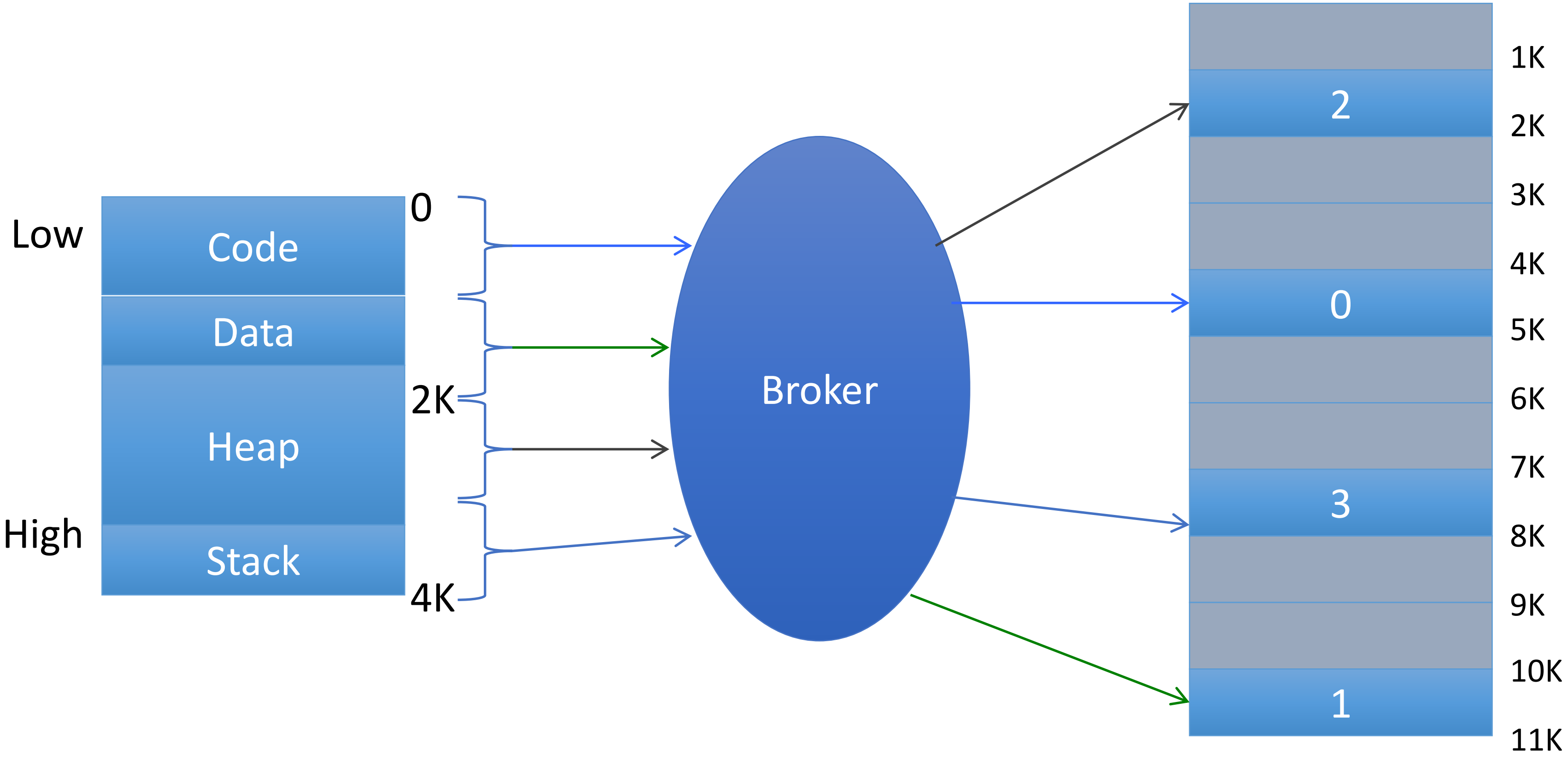
How might we solve the external fragmentation problem?

- What if we could store our memory footprint in non-contiguous memory locations?



- But how can we implement this?

Use more sophisticated broker between CPU and memory



contiguous

Non-contiguous

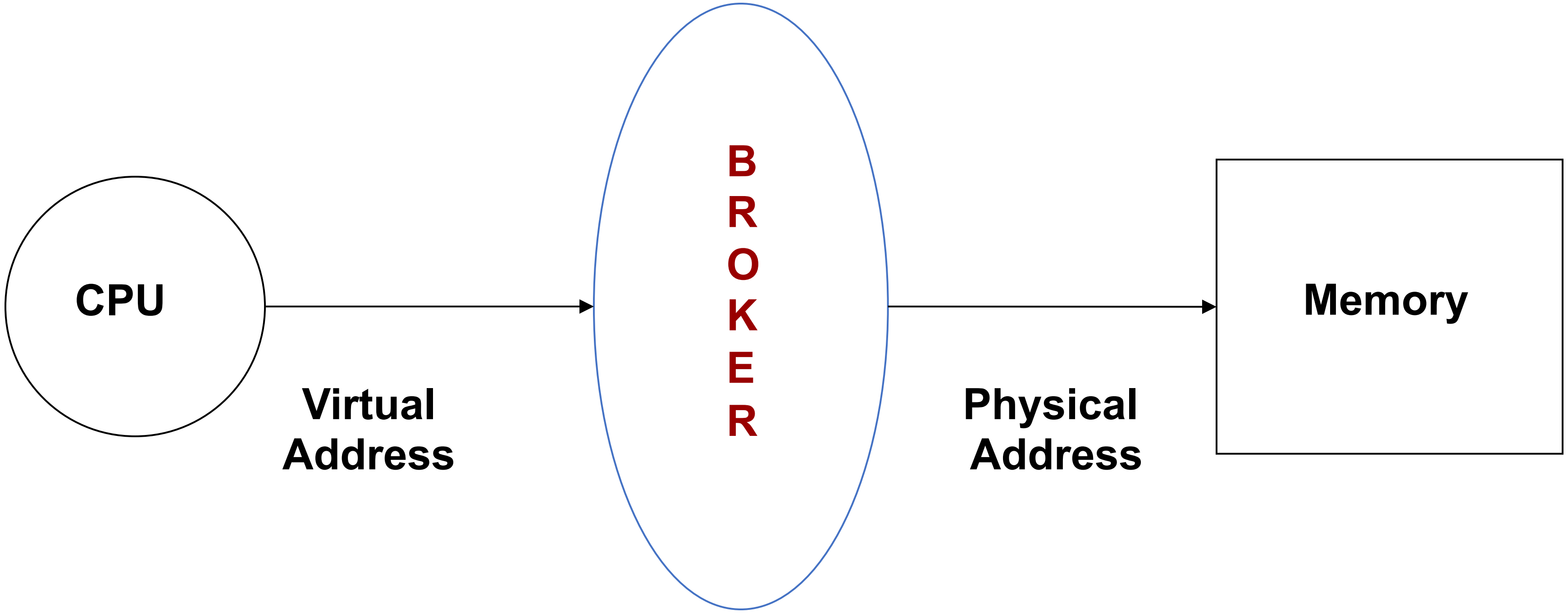
Broker

This broker maps

- Virtual address (VA) from the CPU

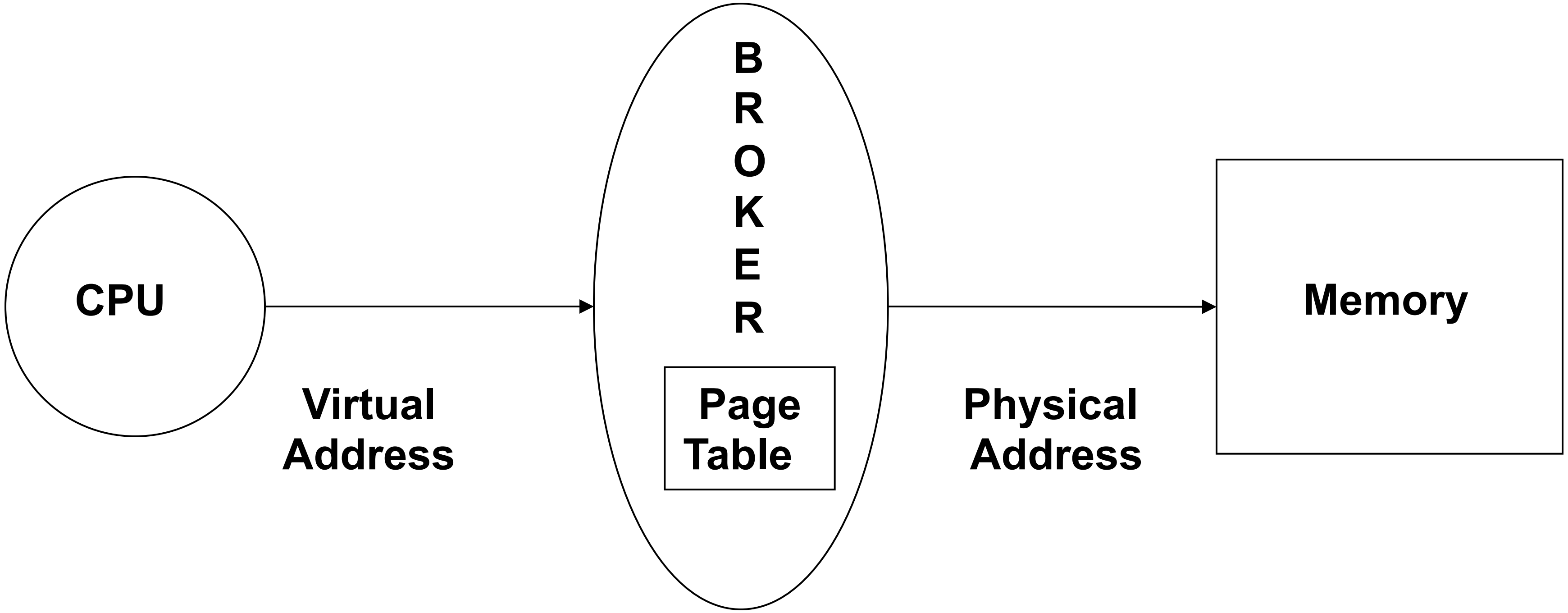
  to

- Physical address (PA) in memory



Broker

- How does Broker map VA to PA?
- Perhaps like a phone directory?
  - Who sets it up? ➔ The OS
  - Who looks it up? ➔ The hardware, on every access



How big is this table?

- At the lower limit, we could map the whole program
  - There would be only one entry in the page table
  - Isn’t that the same as Base + Limit?

- At the upper limit, we could map every word
  - The table (# of entries) would be the size of the address space divided by the word size
  - Not practical at all

- So to strike a balance, we choose a **page size** to map
  - Bigger pages get us more **internal fragmentation** (average is ½ of the last page)
  - Smaller pages get us a bigger page table and take more CPU time to manage it

Choosing a page size

- When memory was expensive (and small)
  - Page sizes were 512 to 2048 bytes

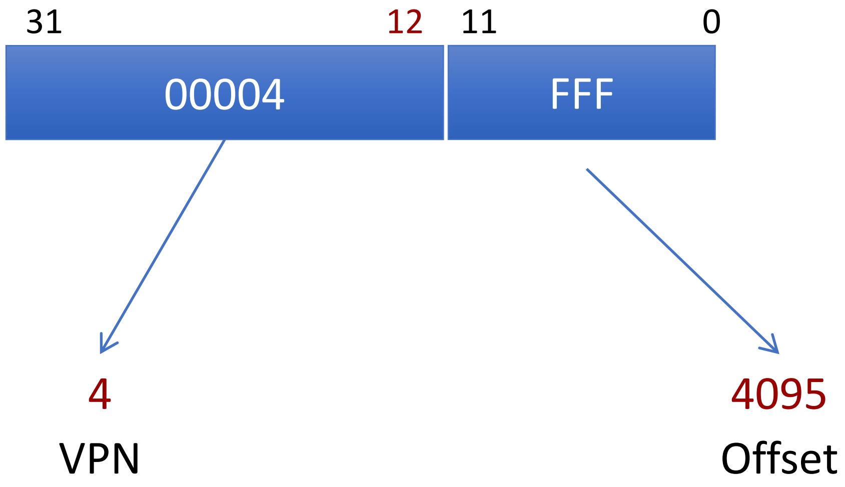
- These days
  - 4 KB up to 1 GB
  - Often it's configurable per process

- Page size is always a power of 2
  - The power of two allows us to split the VA into **virtual page number** (VPN) and **offset** within the page at a bit boundary
  - If the page size is 2^n, the lower **n** bits are the **offset** within the page
    and bits **n** and up are the **virtual page number**

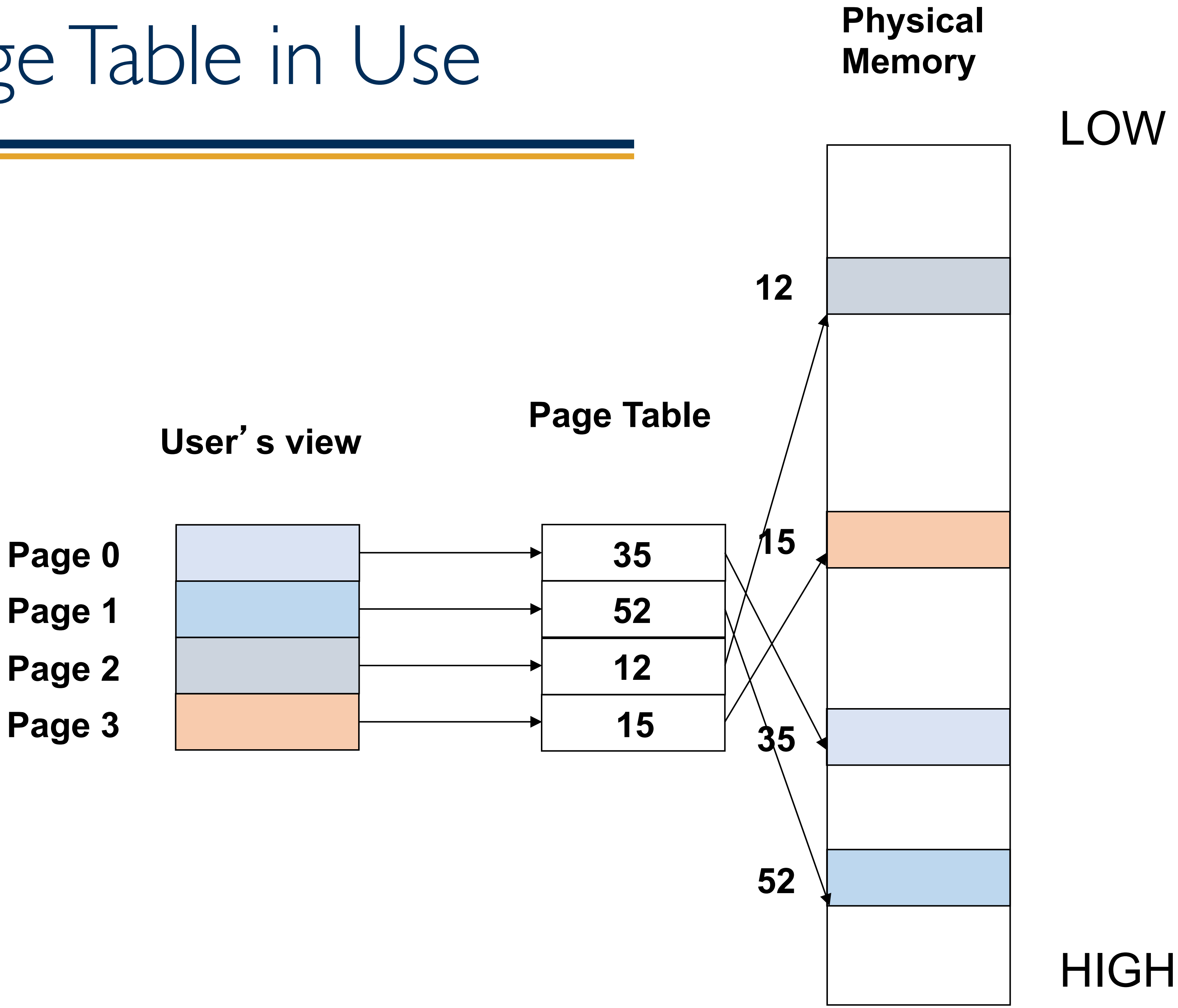
Splitting up a virtual address

- Say we have a 4KB page size and a 32 bit virtual address space
  - 4KB is 2^{12}, so the bottom 12 bits are offset and the top 20 bits are VPN

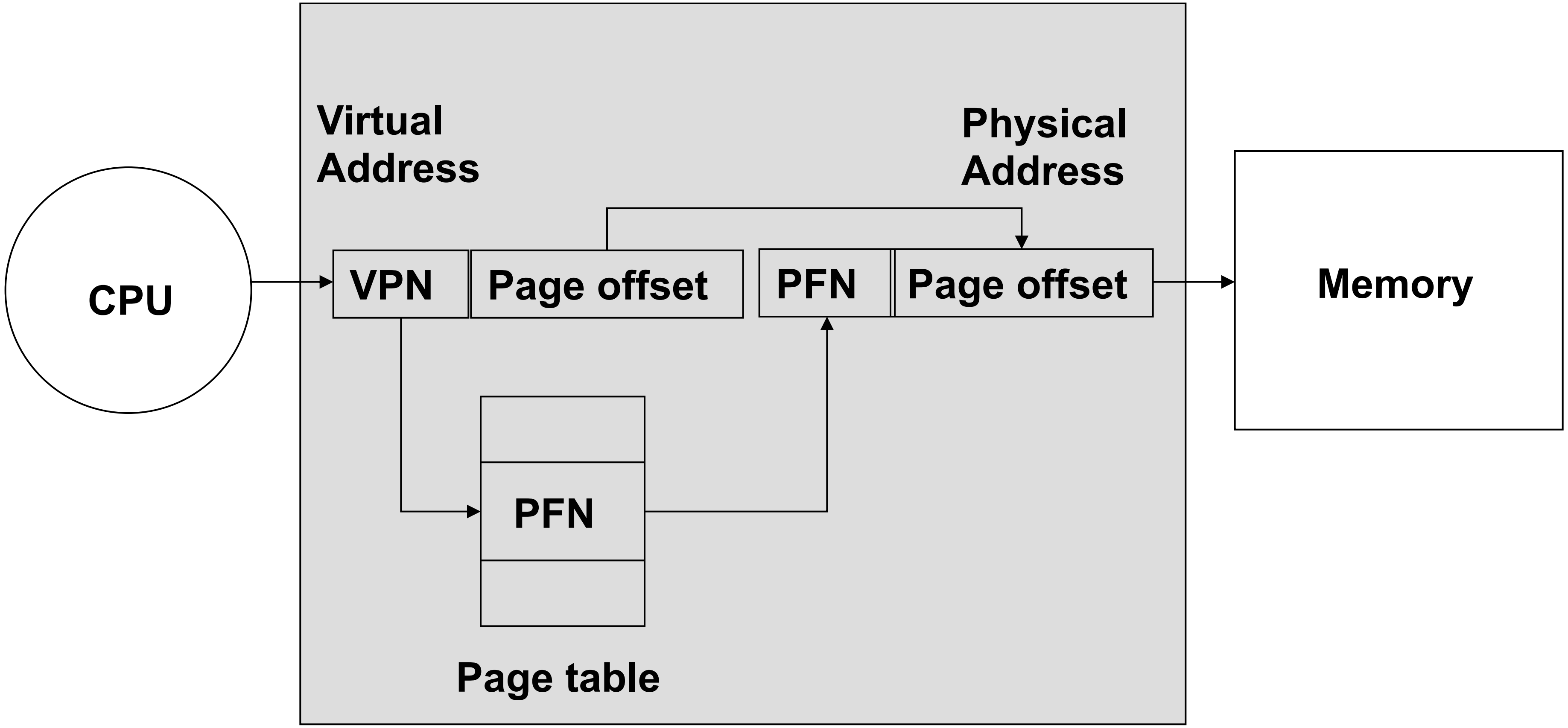
- For example, for virtual address 0x00004FFF:



Page Table in Use

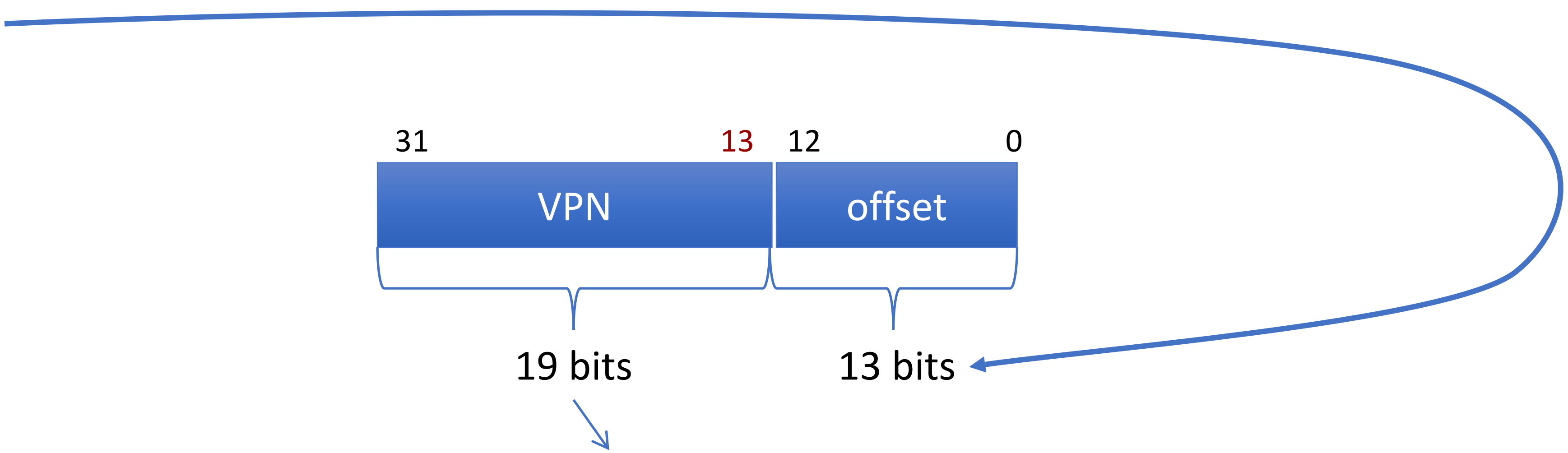


Address translation



Examples

- Consider a memory system with a 32-bit virtual address. Let us assume the pagesize is 8K Bytes.
- How big is the page table (# of entries)?
- 8k = 2^{13}



- VPN is 19 bits, so page table is 2^{19} or 524,288 entries

Examples

Consider a memory system with **32-bit virtual addresses** and **24-bit physical memory addresses.** Assume that the pagesize is 4K Bytes.

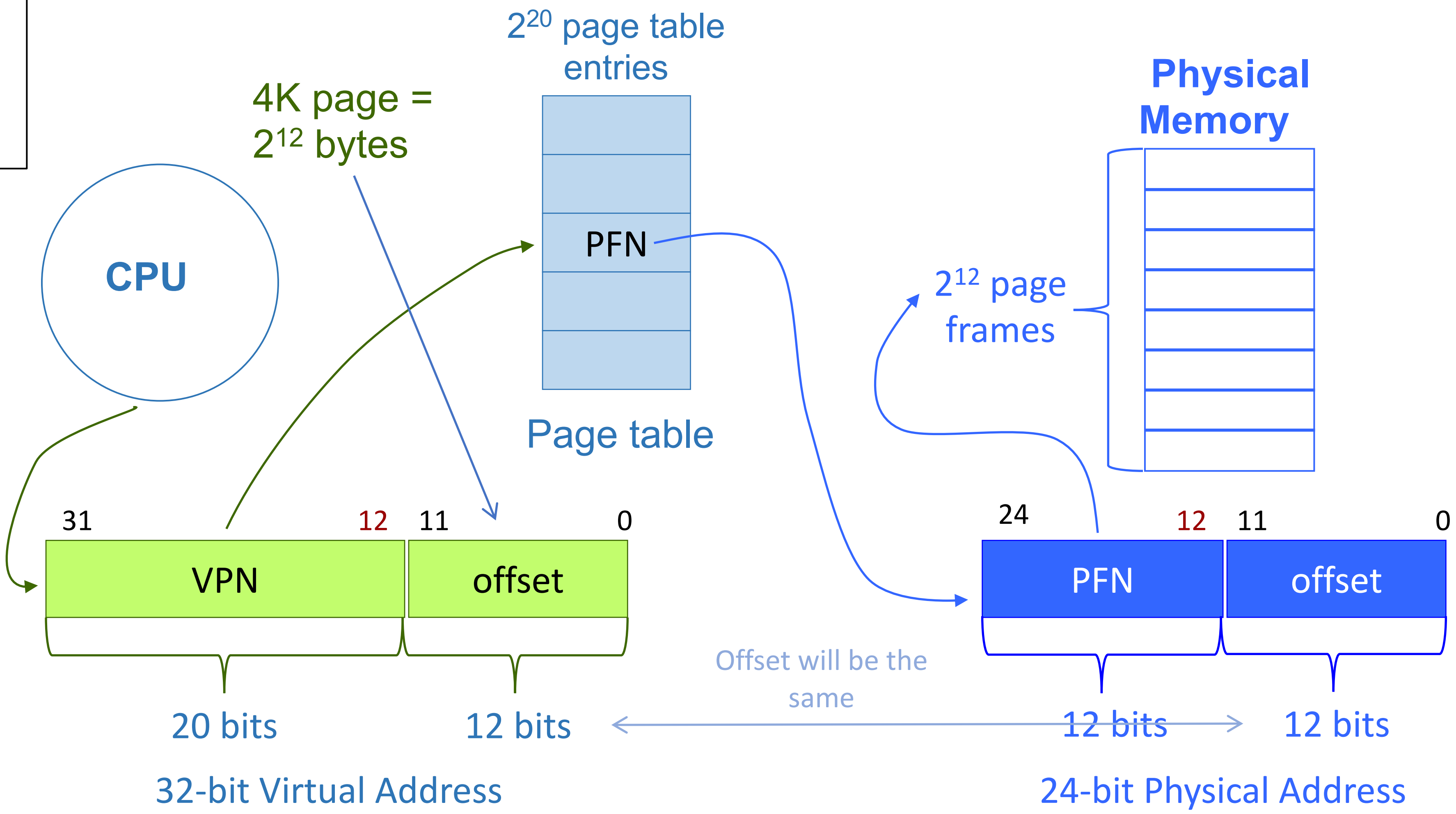
- How many page frames in memory?
- How big is the page table?
- How many page frames are there in this memory system?

Example

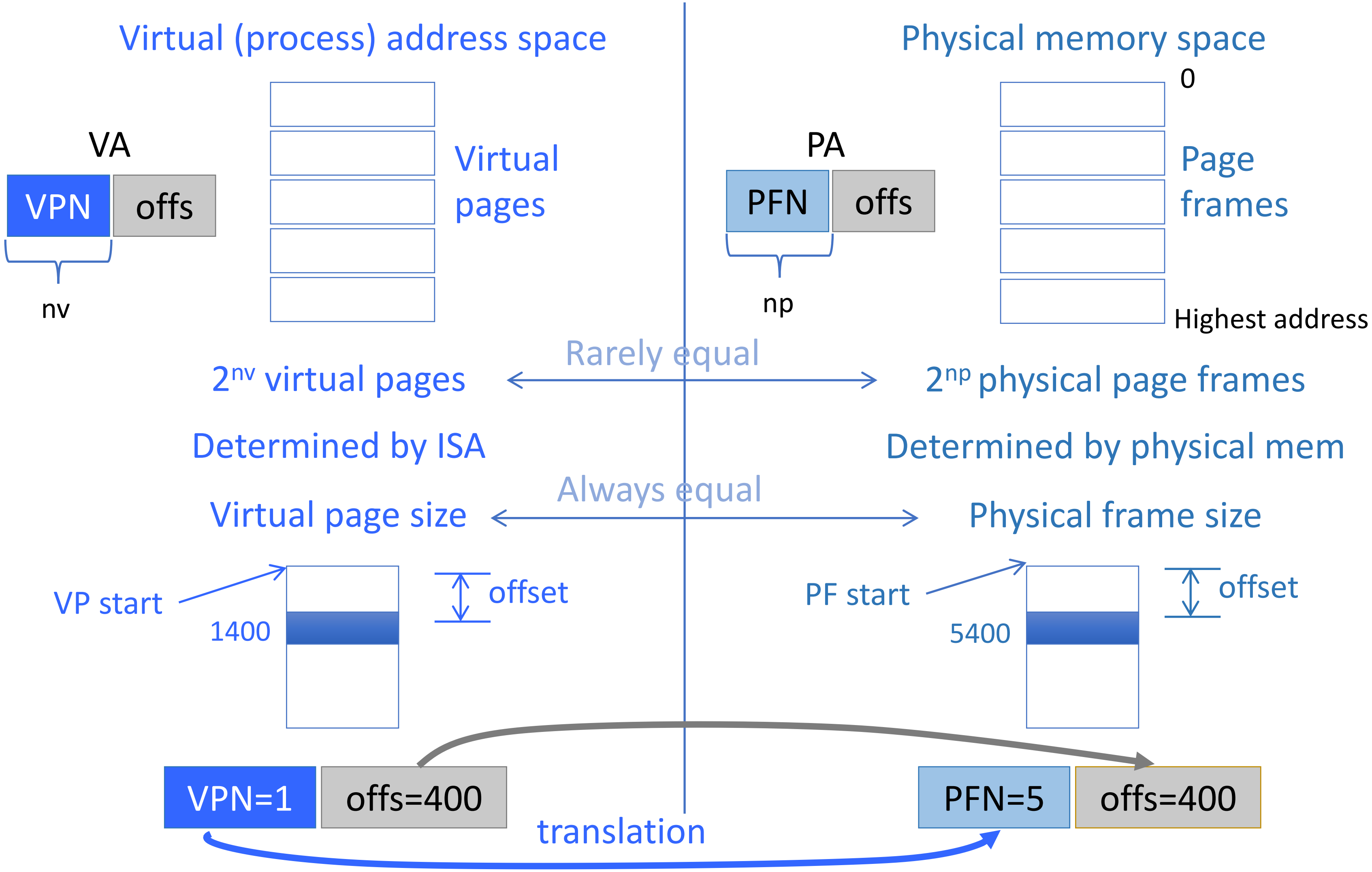
32-bit VA

24-bit PA

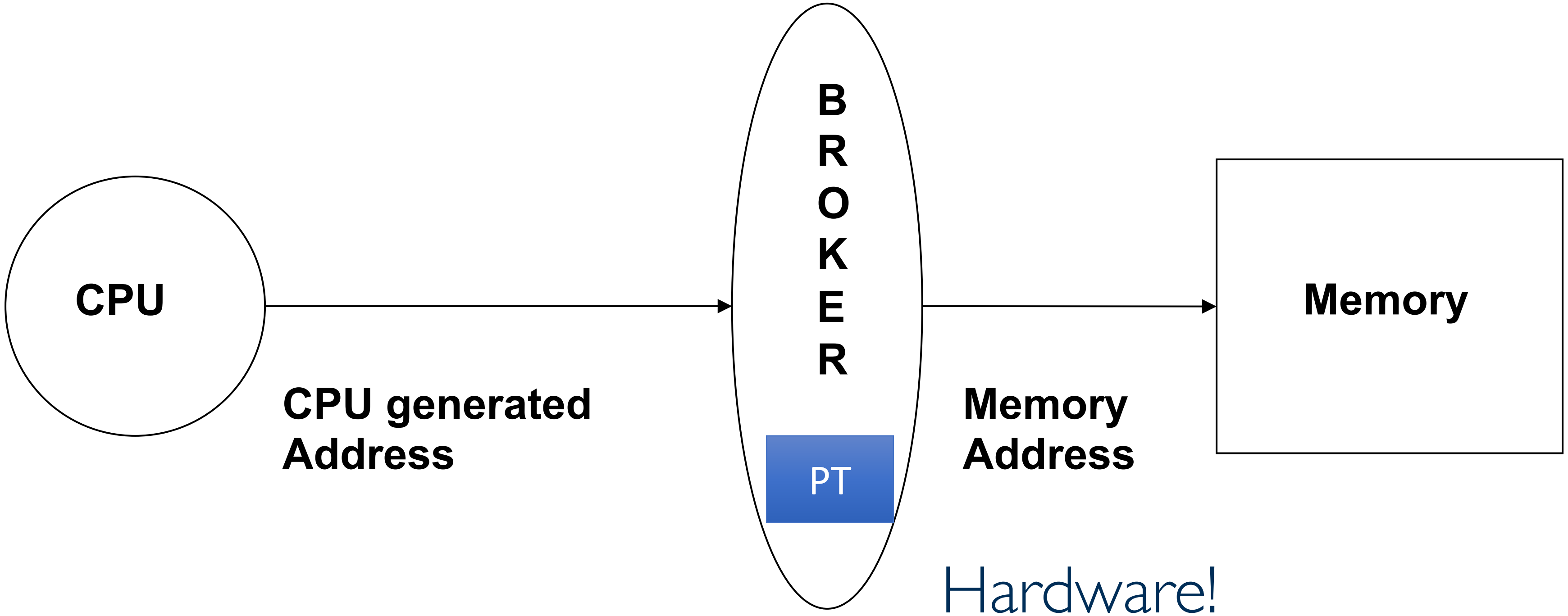
4KB page size



Important facts about paging



So exactly where is the page table?

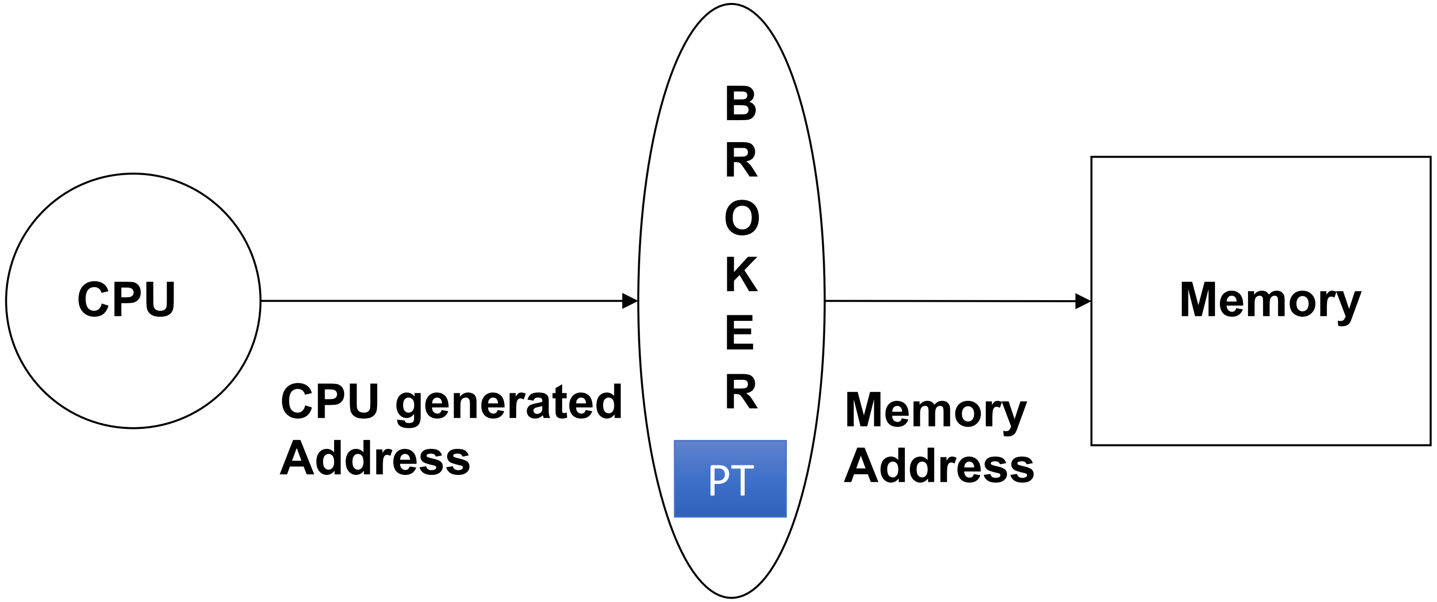


Hardware!

But it's not a special device

It's in physical memory!

How many page tables are there?

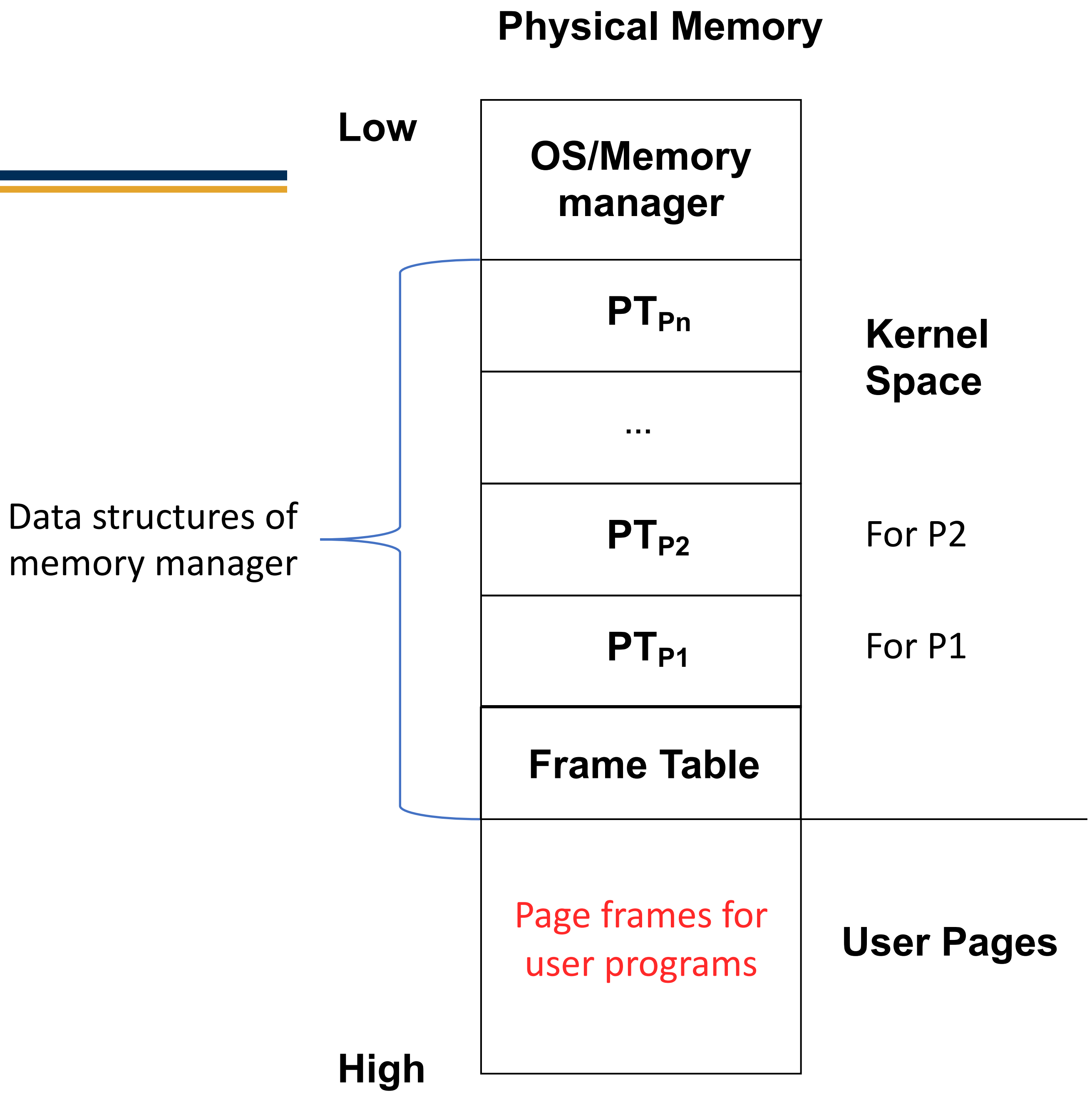


Process 1, 2, 3, …, n in memory

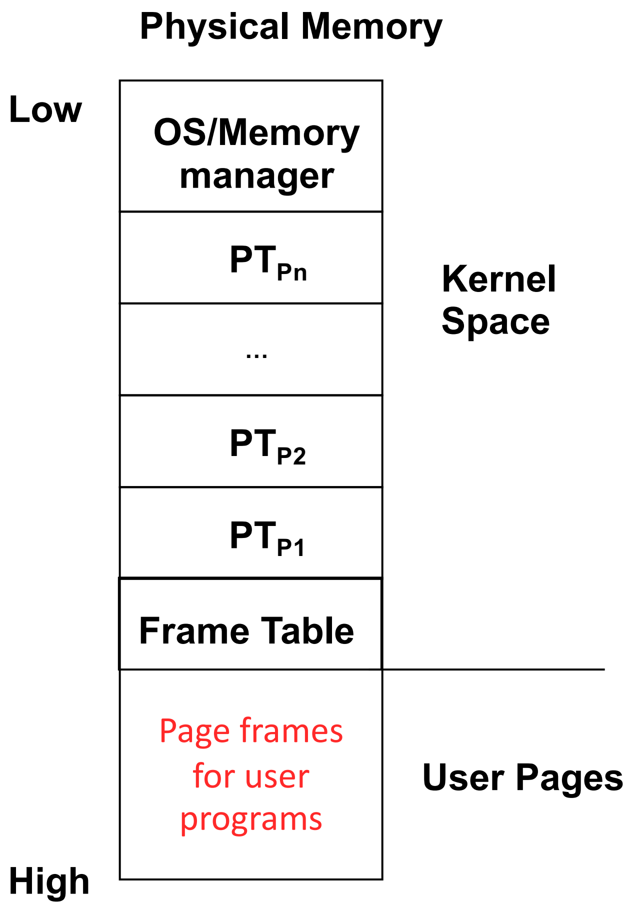
We'll need **n** page tables!

We need as many page tables as the number of processes!

Physical memory layout



What hardware assist does LC-2200 need?



- Just one new register: PTBR
- This holds the base physical address of the page table for the running process
- And of course hardware to look up the PFN from the page table for each memory reference

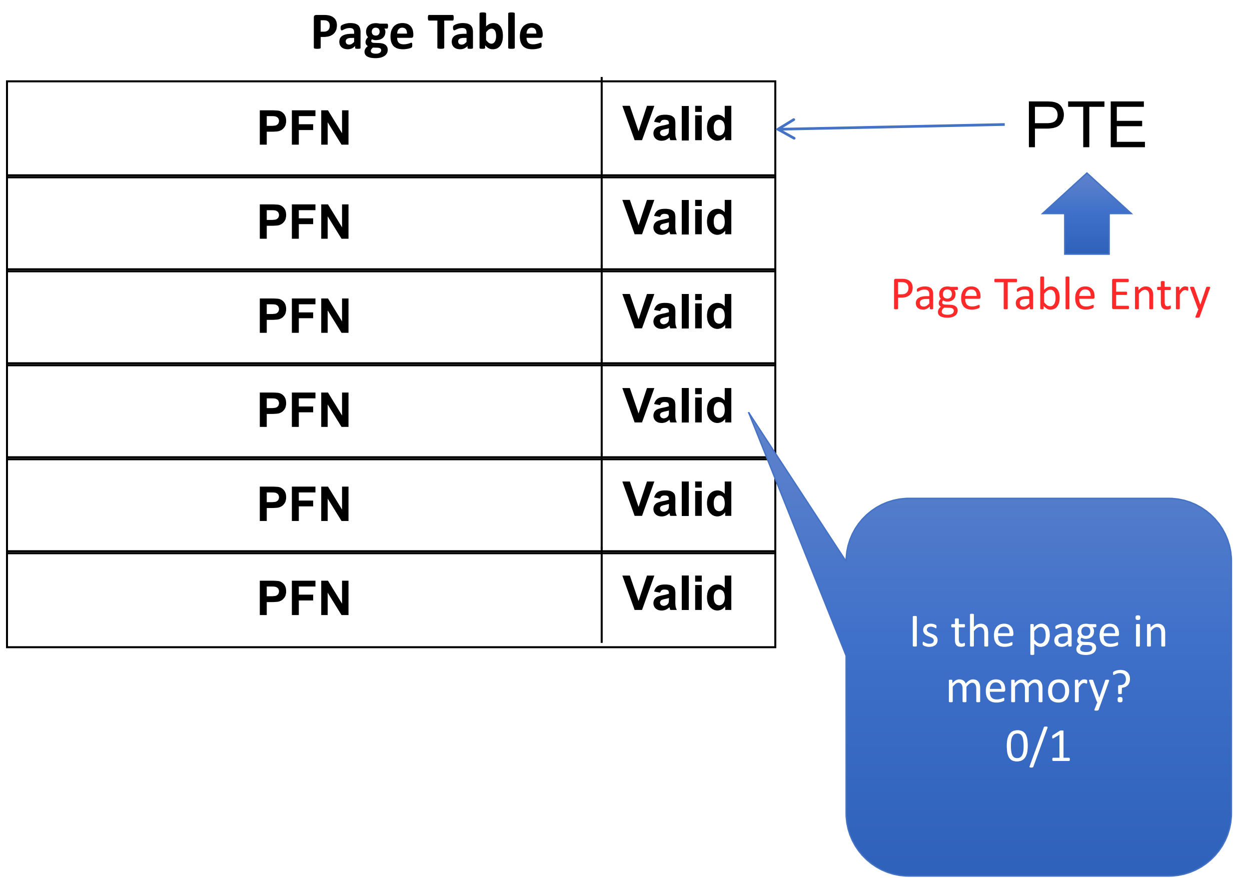
PCB

```
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;
  ….
  ….
} control_block;
```

Paged memory allocation

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

Demand paging



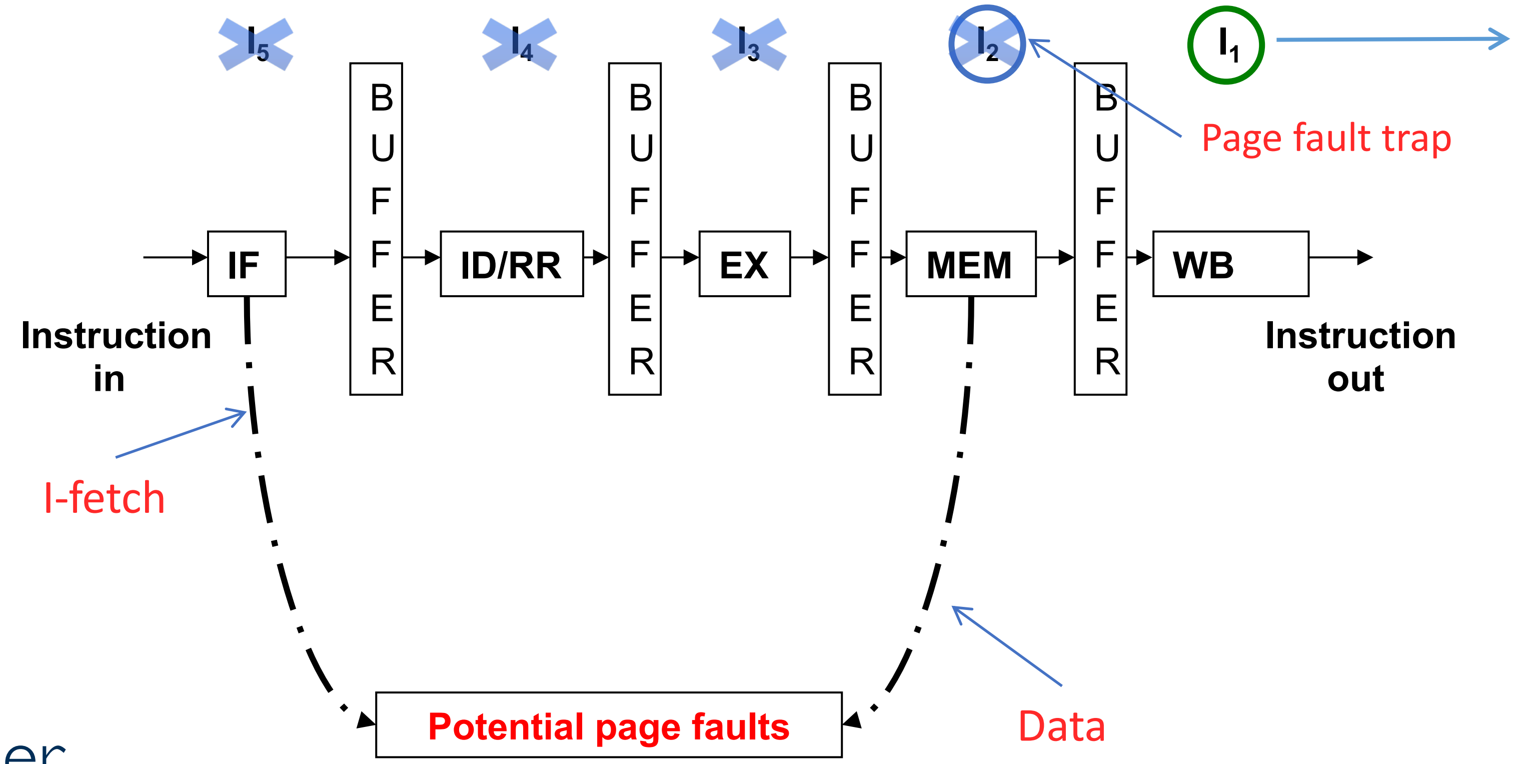
Page Table

PFN	**Valid**
PFN	**Valid**
PFN	**Valid**
PFN	**Valid**
PFN	**Valid**
PFN	**Valid**

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

Ramification of demand paging

Where can a page fault hit us?



Page fault trap

I-fetch

Data access

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

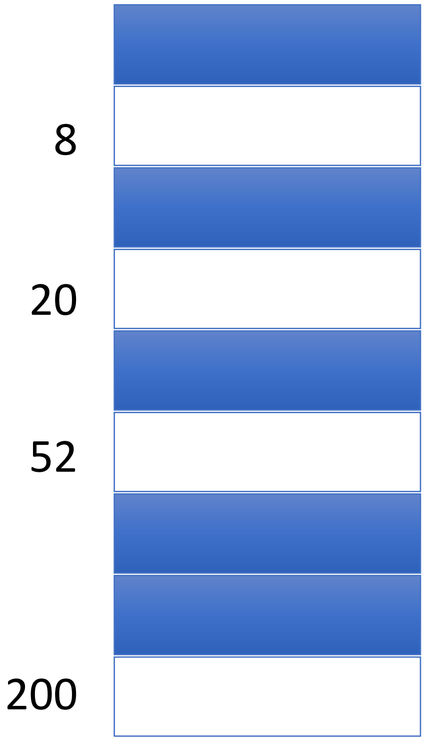
Page fault handler

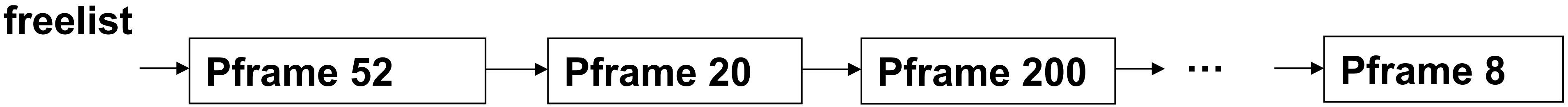
1. Find a free page frame
2. Load the faulting virtual page from disk into the page frame
3. Give up the CPU while waiting for the paging I/O to complete
4. Update the page table entry for the faulting page
5. Place the PCB of the process back in the ready_q of the scheduler
6. Call the scheduler

Three Data structures for page fault handler

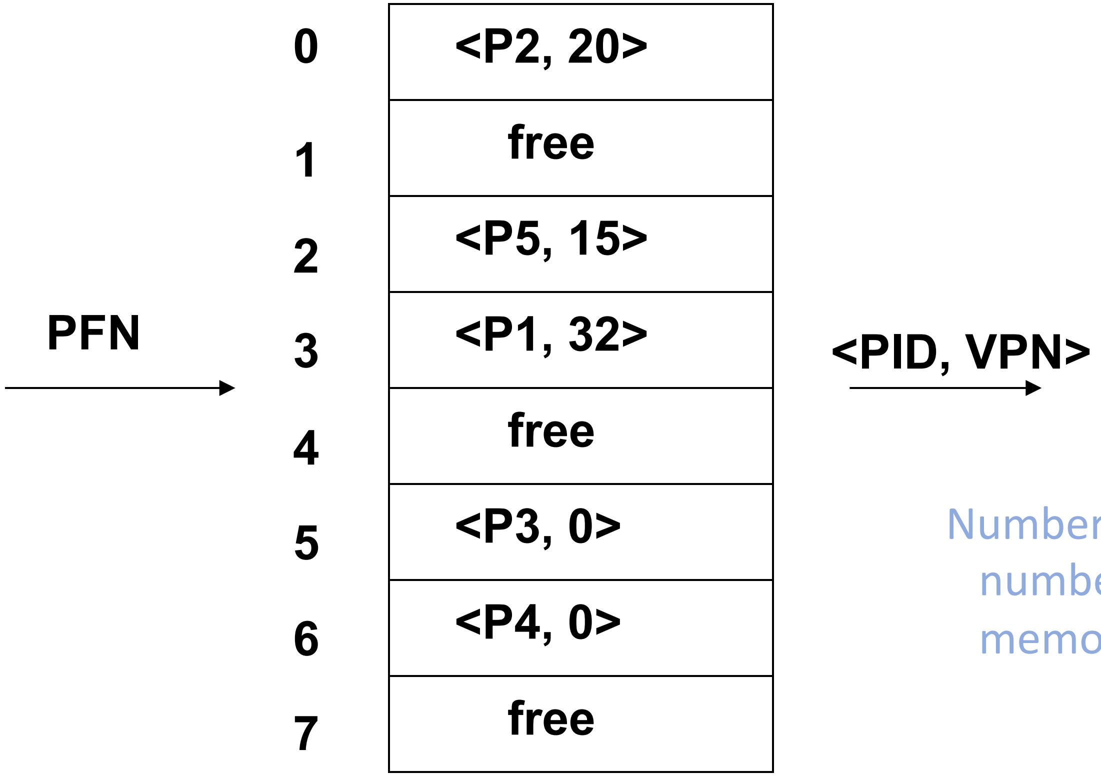
(1/3): Freelist

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





(2/3): Frame Table



0	<P2, 20>
1	free
2	<P5, 15>
3	<P1, 32>
4	free
5	<P3, 0>
6	<P4, 0>
7	free

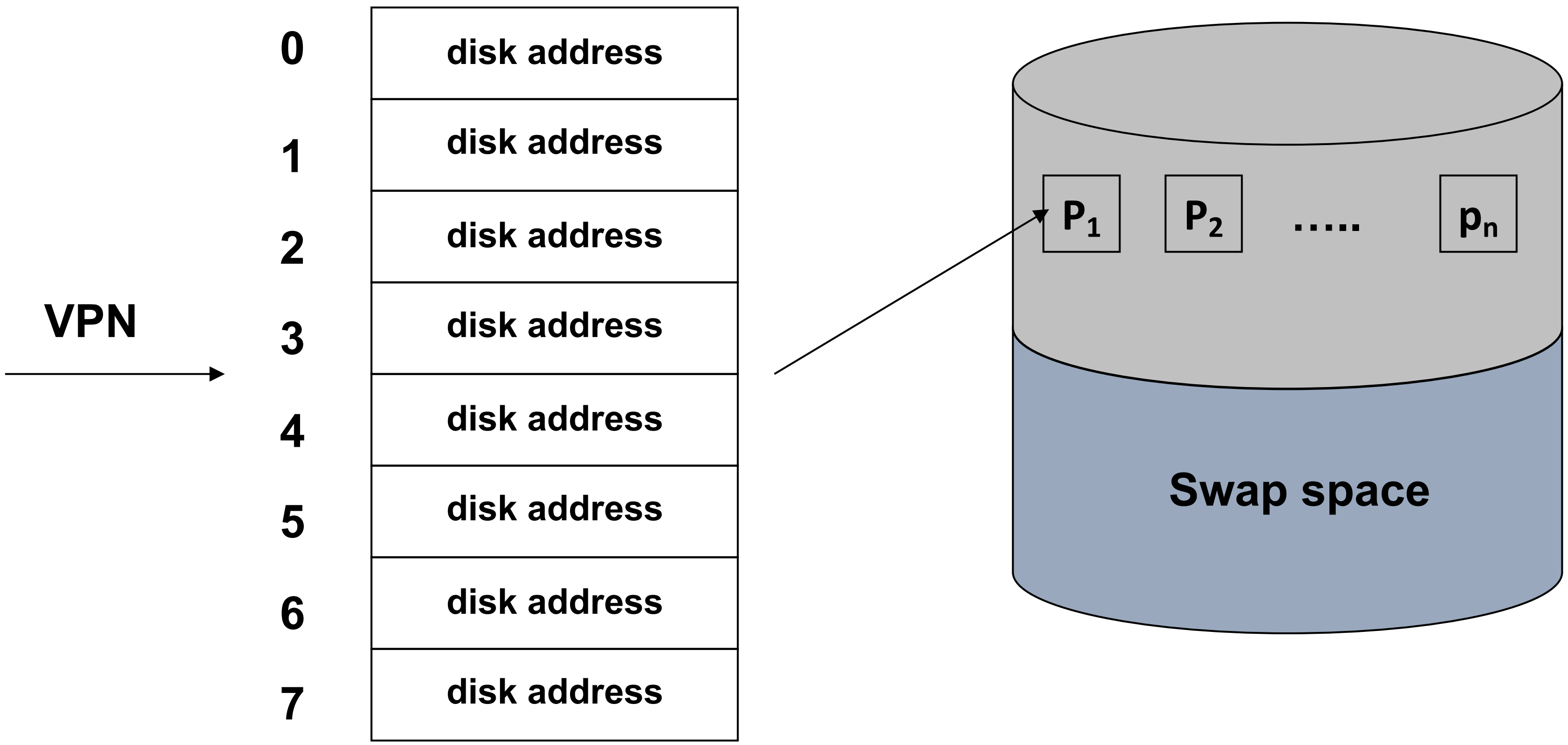
Number of entries =
number of physical
memory page frames

Reverse mapping compared to PT

We need this for evicting pages

(3/3): Disk Map

Disk map for P1



0	disk address
1	disk address
2	disk address
3	disk address
4	disk address
5	disk address
6	disk address
7	disk address

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

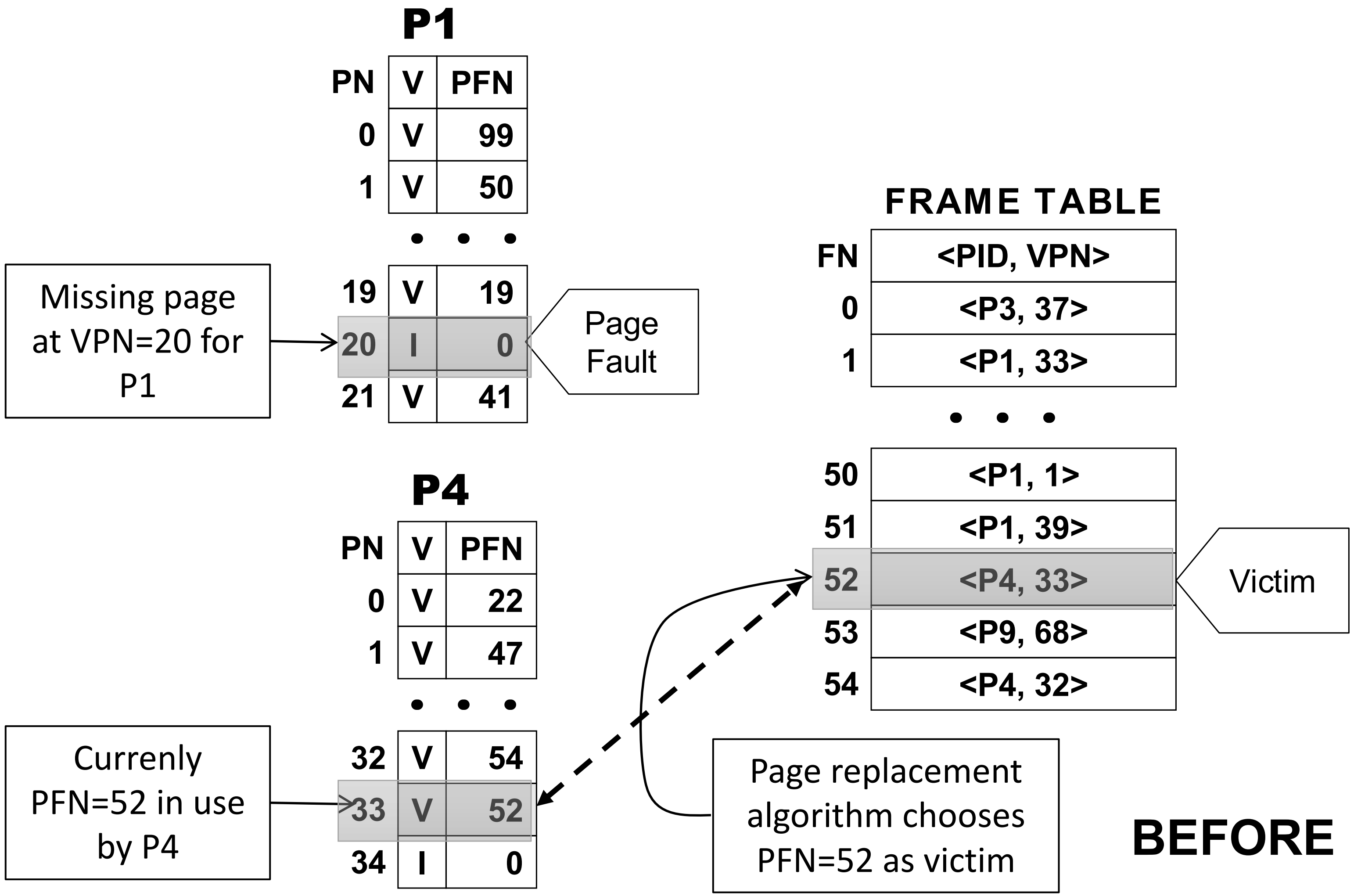
PCB

```
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



P1

PN	V	PFN
0	V	99
1	V	50
	•	• •
19	V	19
20	I	0
21	V	41

Missing page at VPN=20 for P1

Page Fault

P4

PN	V	PFN
0	V	22
1	V	47
	•	• •
32	V	54
33	V	52
34	I	0

Currenly PFN=52 in use by P4

FRAME TABLE

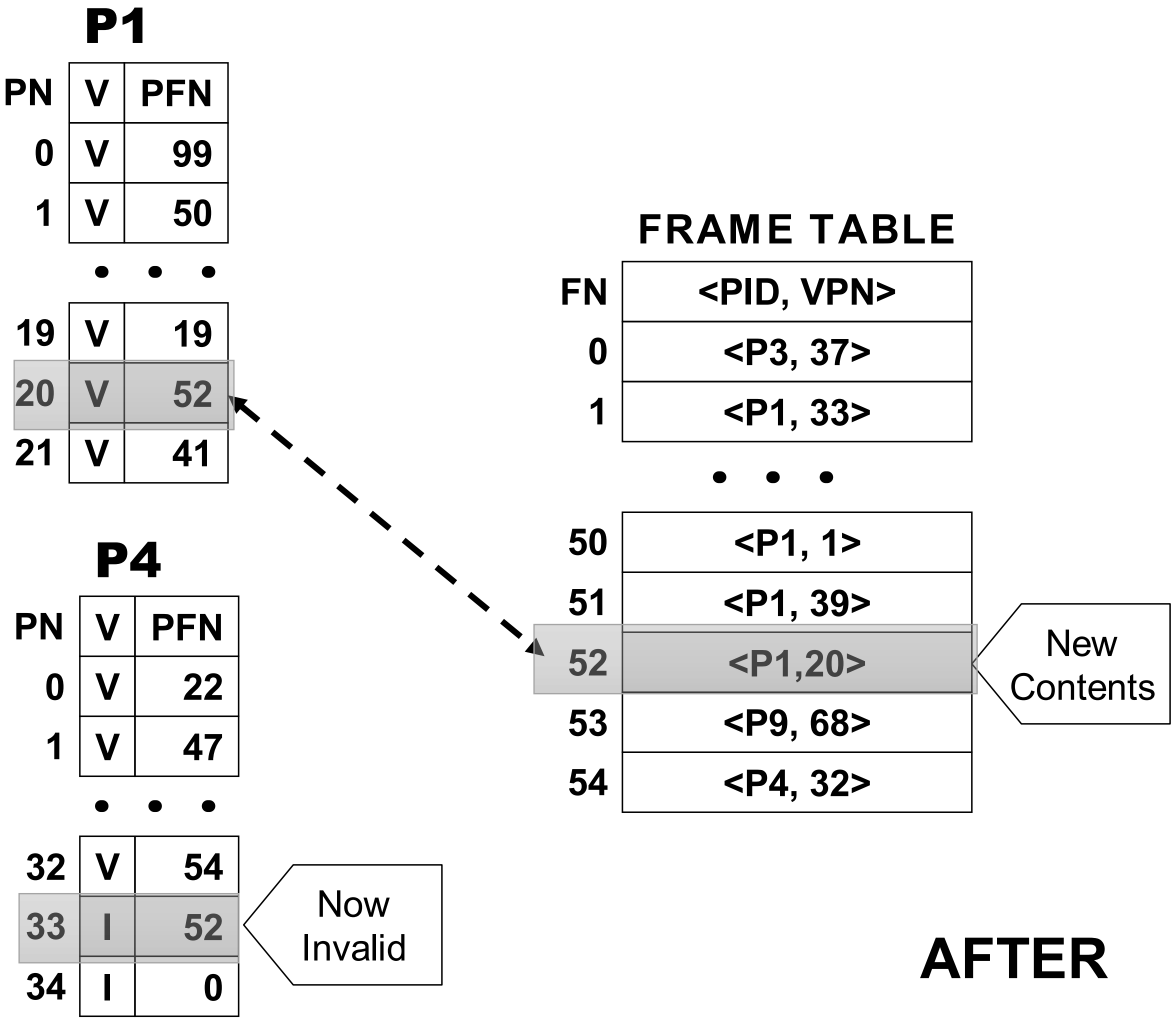
FN	<PID, VPN>
0	<P3, 37>
1	<P1, 33>
	• • •
50	<P1, 1>
51	<P1, 39>
52	<P4, 33>
53	<P9, 68>
54	<P4, 32>

Victim

Page replacement algorithm chooses PFN=52 as victim

BEFORE

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

Session ID: dag2200

With paged memory management there can be…

A. External fragmentation

B. Internal fragmentation

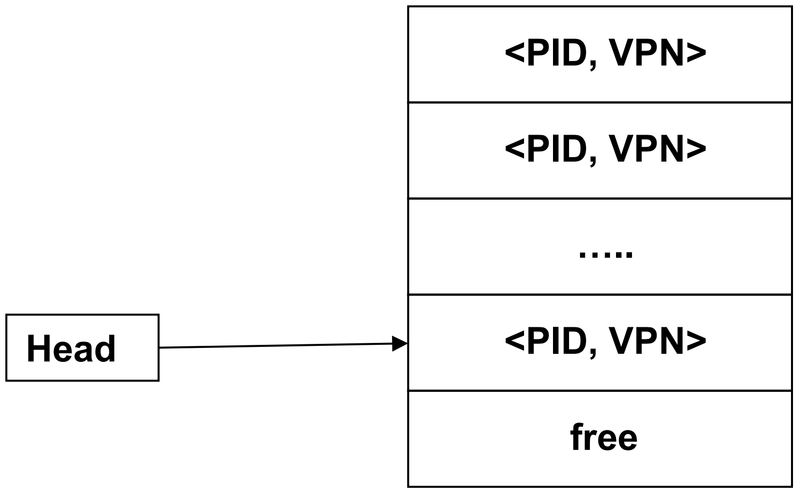
C. No fragmentation

D. Both internal and external fragmentation

FIFO Page Replacement

- Manage the Frame Table like a queue

Circular queue



<PID, VPN>

<PID, VPN>

…..

<PID, VPN>

free

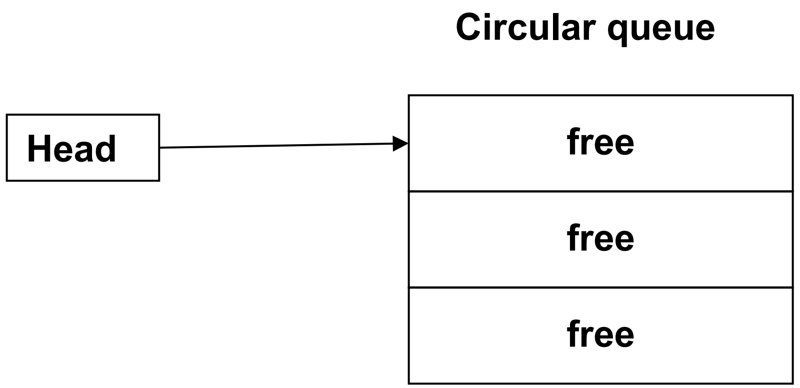
Head

FIFO example

Consider a sequence of page references by a process:

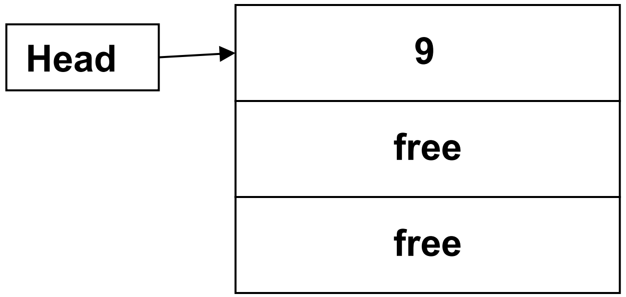
Reference number:	1	2	3	4	5	6	7	8	9	10	11	12	13
Virtual page number:	9	0	3	4	0	5	0	6	4	5	0	5	4

Assume there are 3 physical frames.

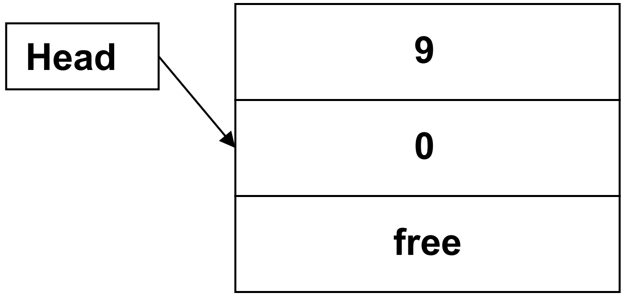


Ref:	1	2	3	4	5	6
VPN:	9	0	3	4	0	5

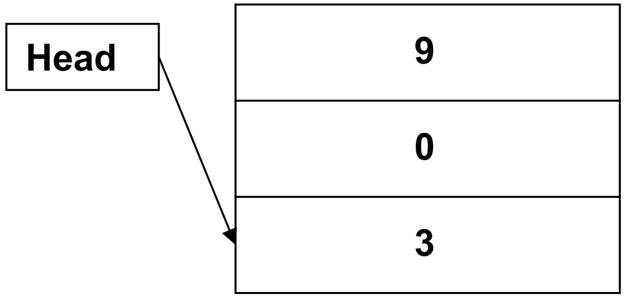
Reference #1 (PF)



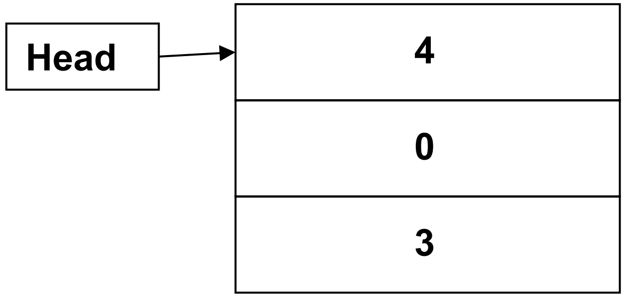
Reference #2 (PF)



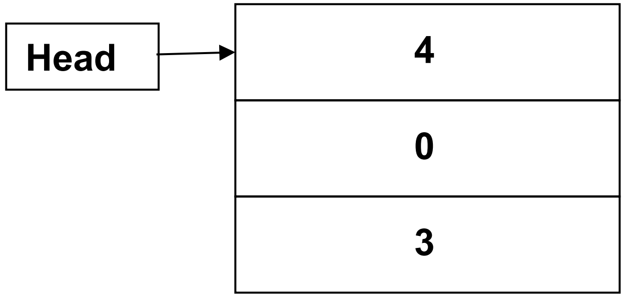
Reference #3 (PF)



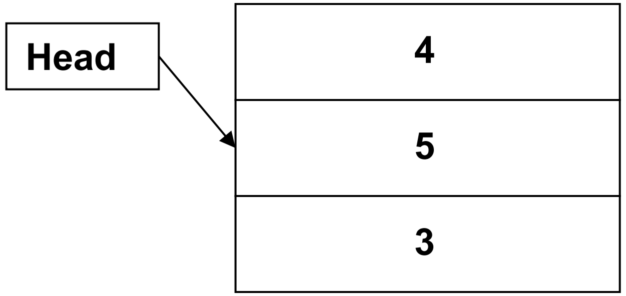
Reference #4 (PF)



Reference #5 (HIT)



Reference #6 (PF)



Session ID: dag2200

Size of the FIFO queue?

A. 42

B. Number of physical page frames

C. Number of virtual pages

D. 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

Session ID: dag2200

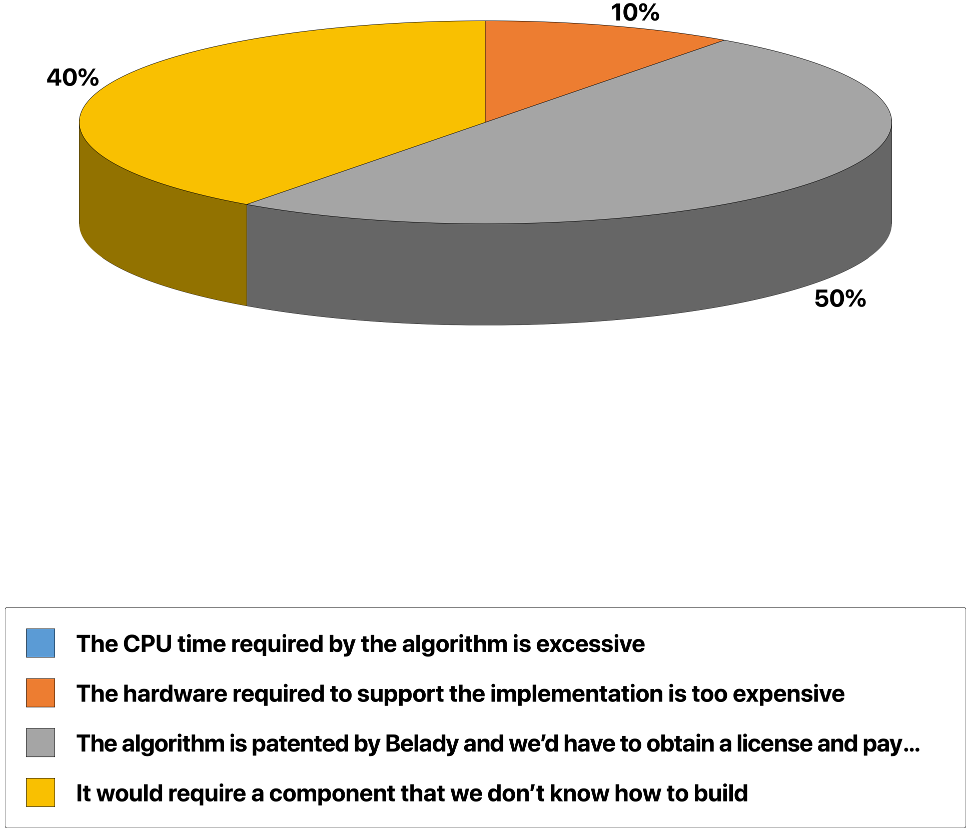
Why not implement Belady's Min as a page replacement algorithm?

A. The CPU time required by the algorithm is excessive

B. The hardware required to support the implementation is too expensive

C. The algorithm is patented by Belady and we'd have to obtain a license and pay royalties

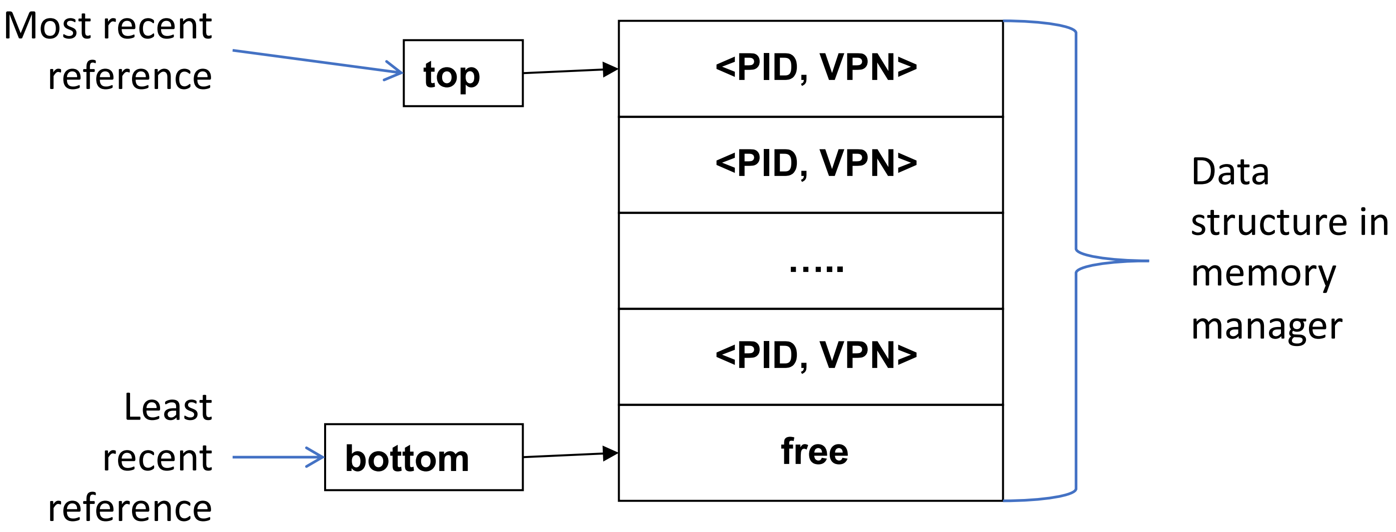
D. It would require a component that we don't know how to build



LRU

- Use the past as a predictor of the future.

Push down stack



Most recent reference

top

<PID, VPN>

<PID, VPN>

.....

<PID, VPN>

free

Least recent reference

bottom

Data structure in memory manager

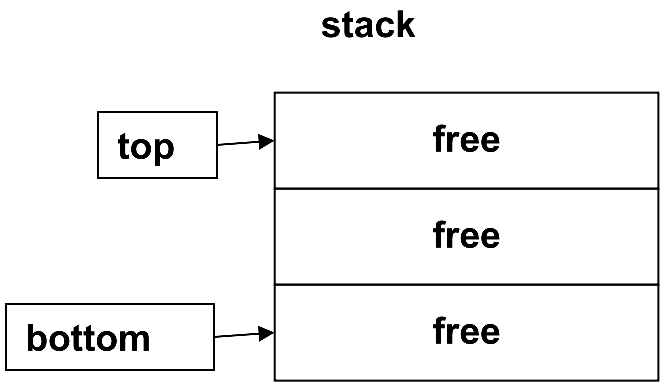
Each element corresponds to a page frame

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.



stack

top

bottom

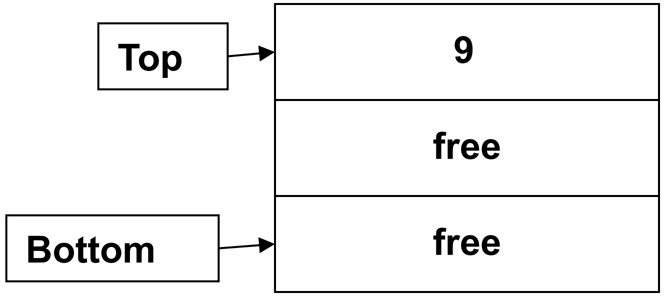
free

free

free

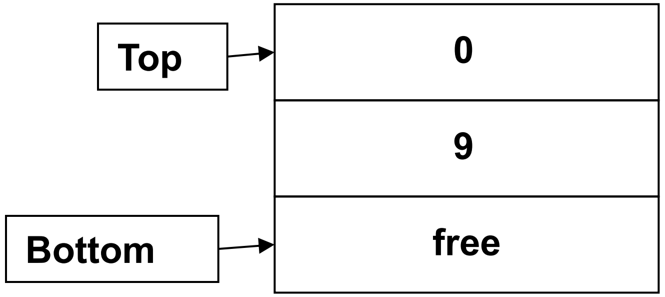
Ref:	1	2	3	4	5	6
VPN:	9	0	3	4	0	5

Reference #1 (PF)



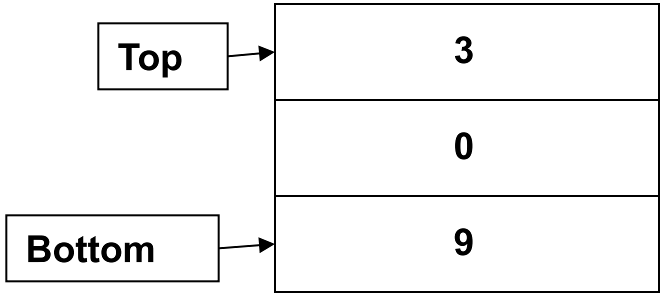
Top
Bottom
9
free
free

Reference #2 (PF)



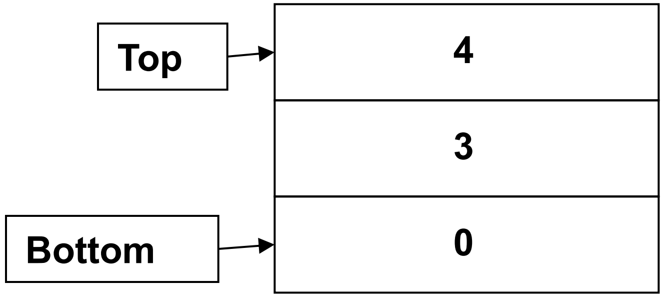
Top
Bottom
0
9
free

Reference #3 (PF)



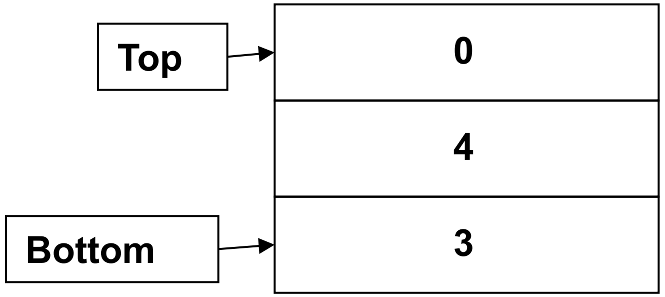
Top
Bottom
3
0
9

Reference #4 (PF)



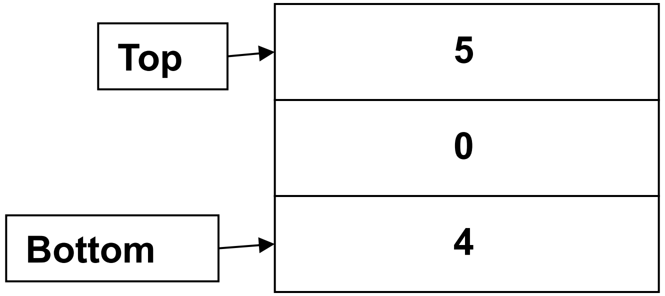
Top
Bottom
4
3
0

Reference #5 (HIT)



Top
Bottom
0
4
3

Reference #6 (PF)



Top
Bottom
5
0
4

Session ID: dag2200

Size of LRU "stack"

A. 42

B. Number of virtual pages

C. Number of physical frames

D. 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 a memory write is required for each memory reference

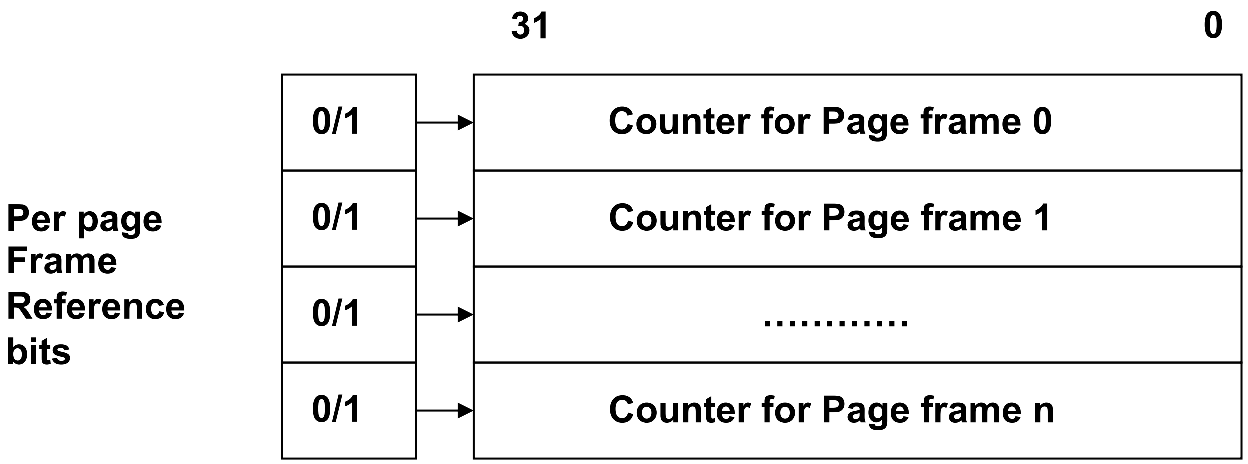
Ways to approximate LRU

- 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



Per page Frame Reference bits

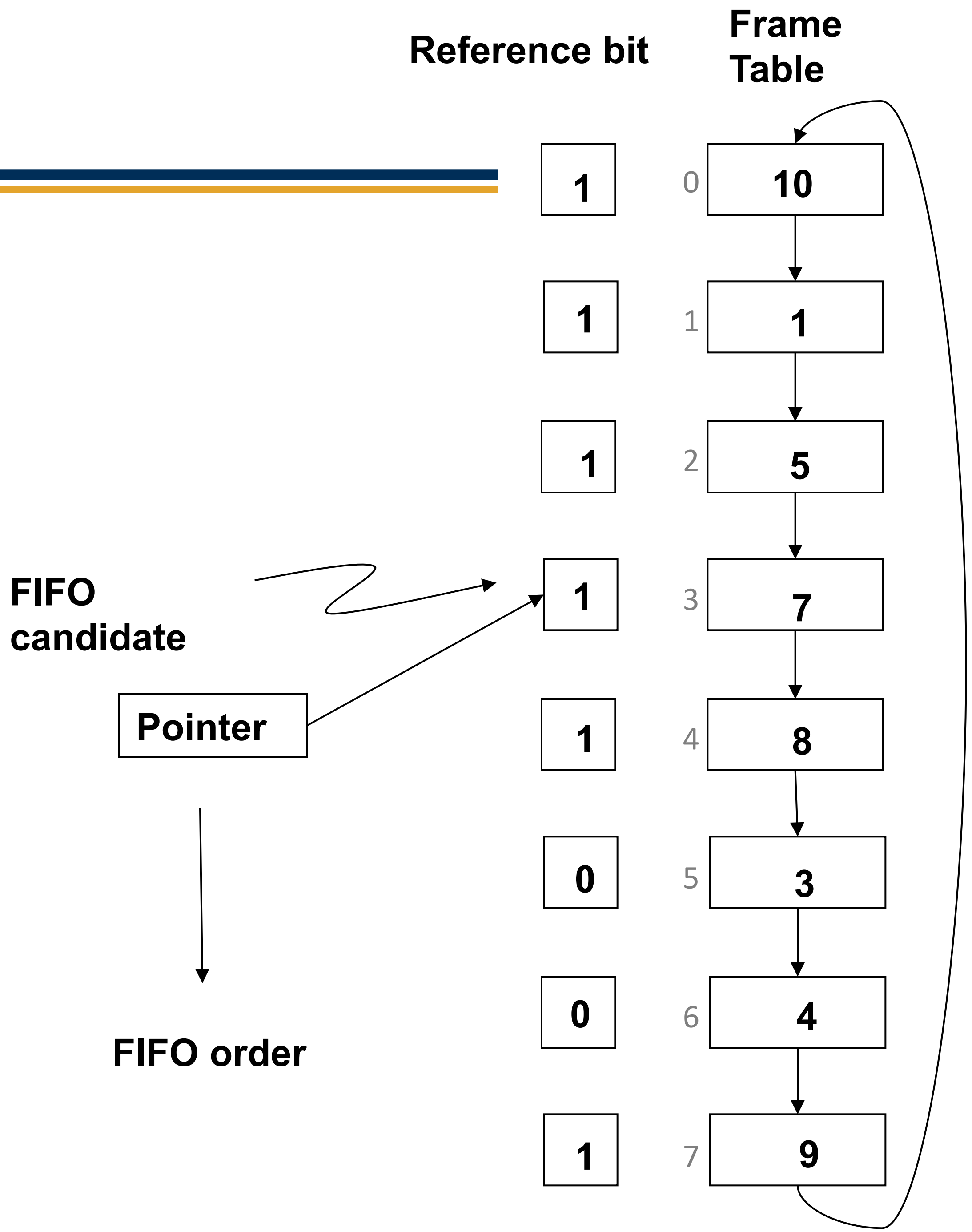
	31 … 0
0/1 →	Counter for Page frame 0
0/1 →	Counter for Page frame 1
0/1 →	…………
0/1 →	Counter for Page frame n

- 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

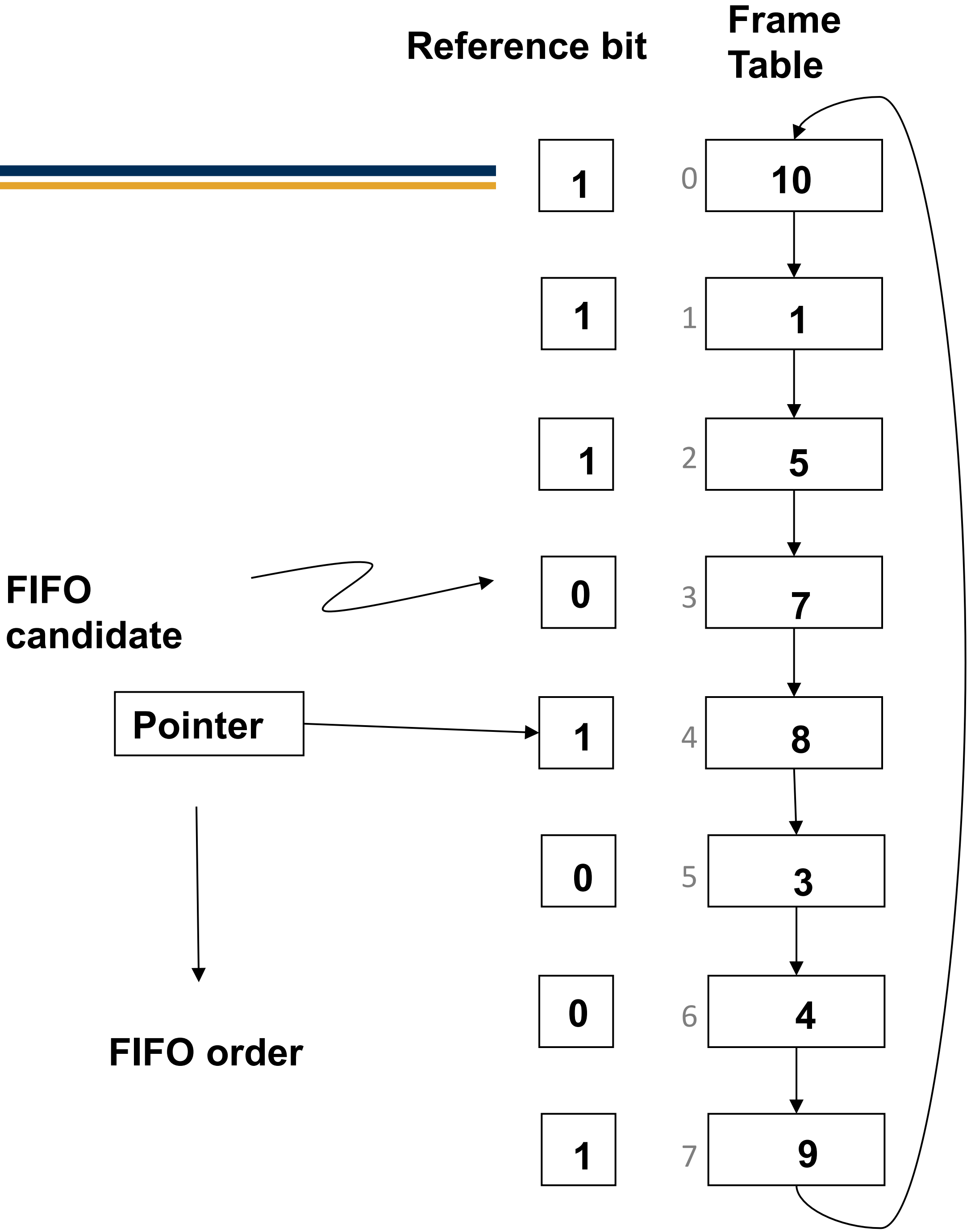


Reference bit		Frame Table
1	0	10
1	1	1
1	2	5
1	3	7
1	4	8
0	5	3
0	6	4
1	7	9

Try page 7

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

Second chance replacement

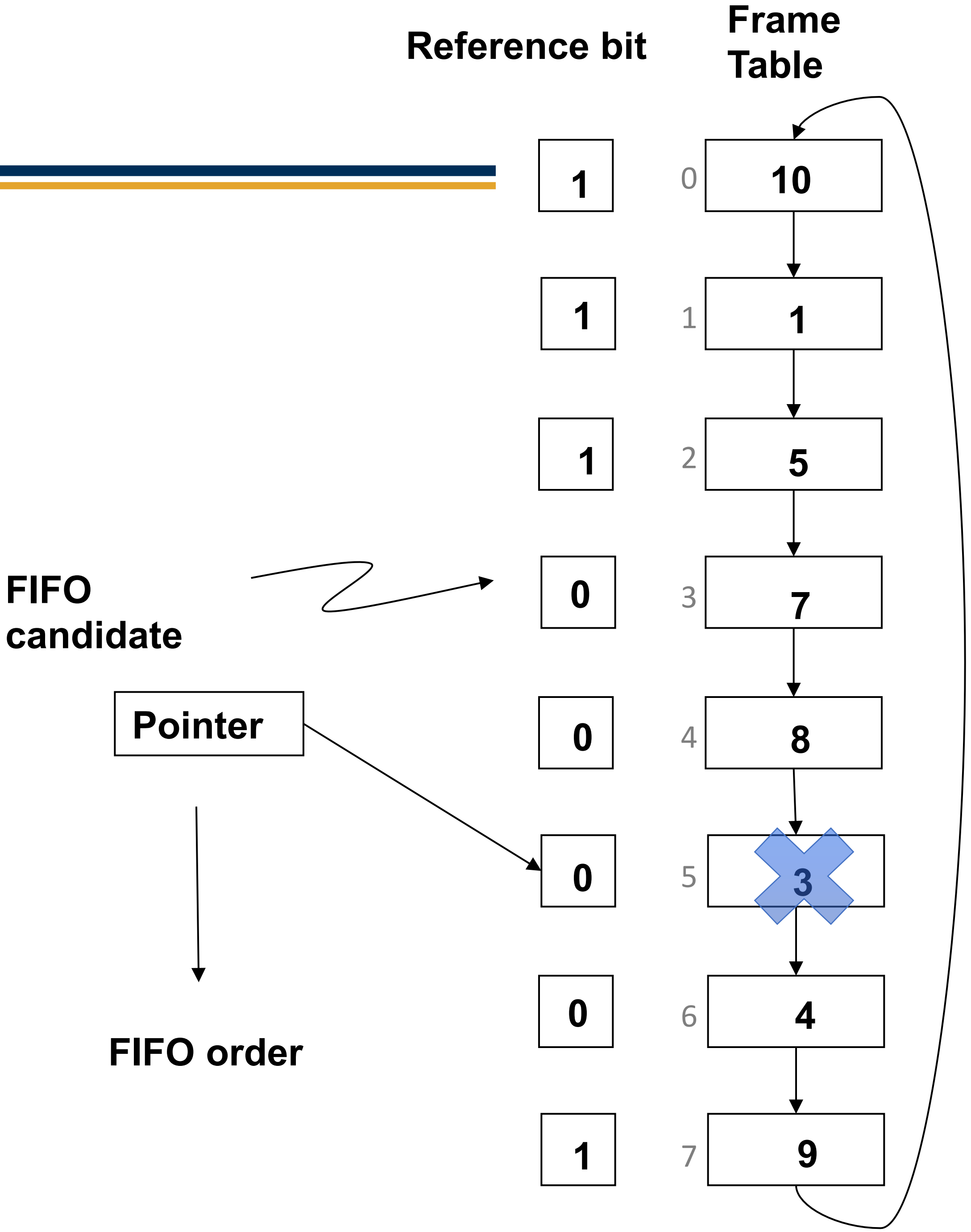


	Reference bit	Frame Table
0	1	10
1	1	1
2	1	5
3	0	7
4	1	8
5	0	3
6	0	4
7	1	9

Try page 8

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

Second chance replacement

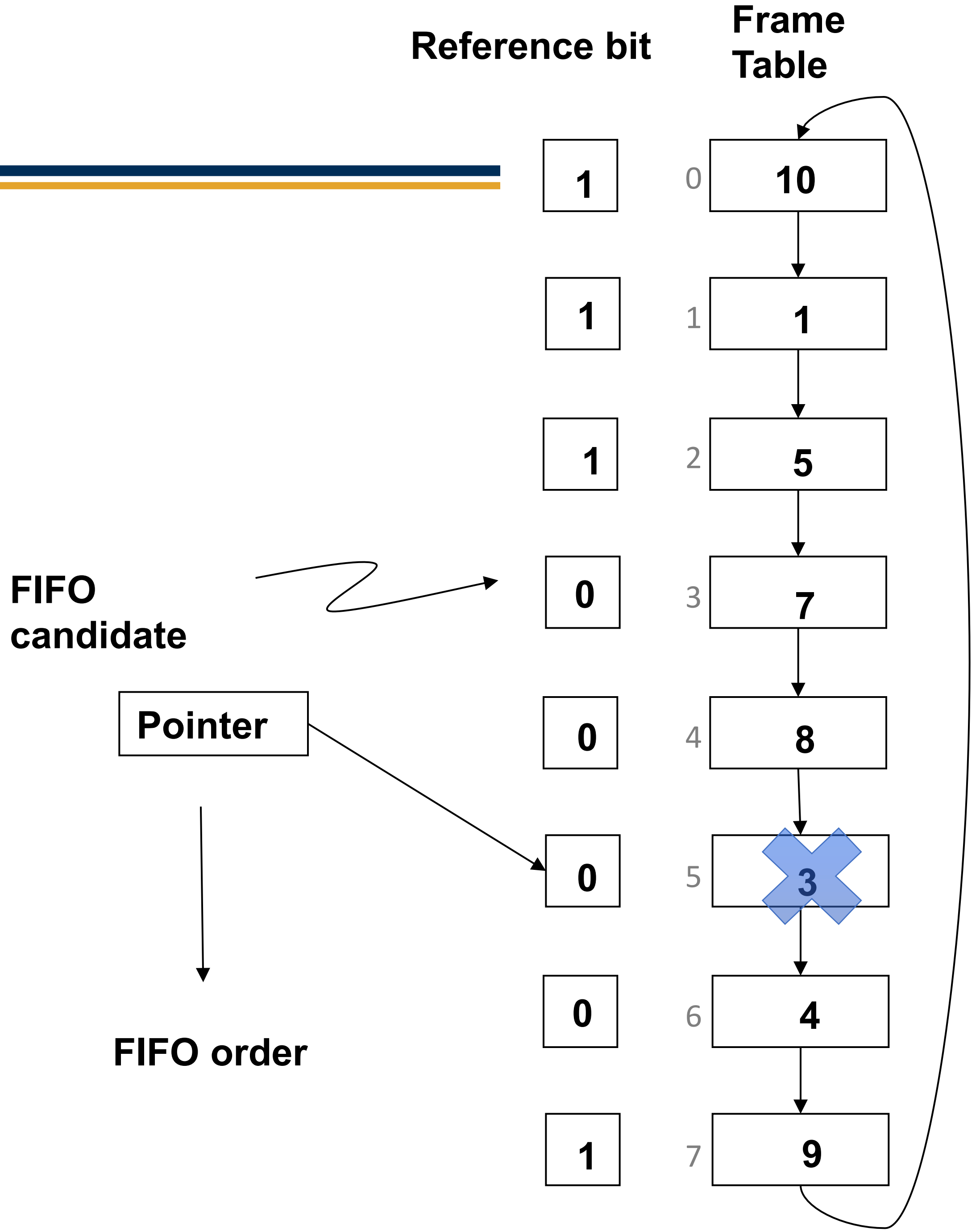


Reference bit		Frame Table
1	0	10
1	1	1
1	2	5
0	3	7
0	4	8
0	5	3
0	6	4
1	7	9

Try page 3

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

Second chance replacement

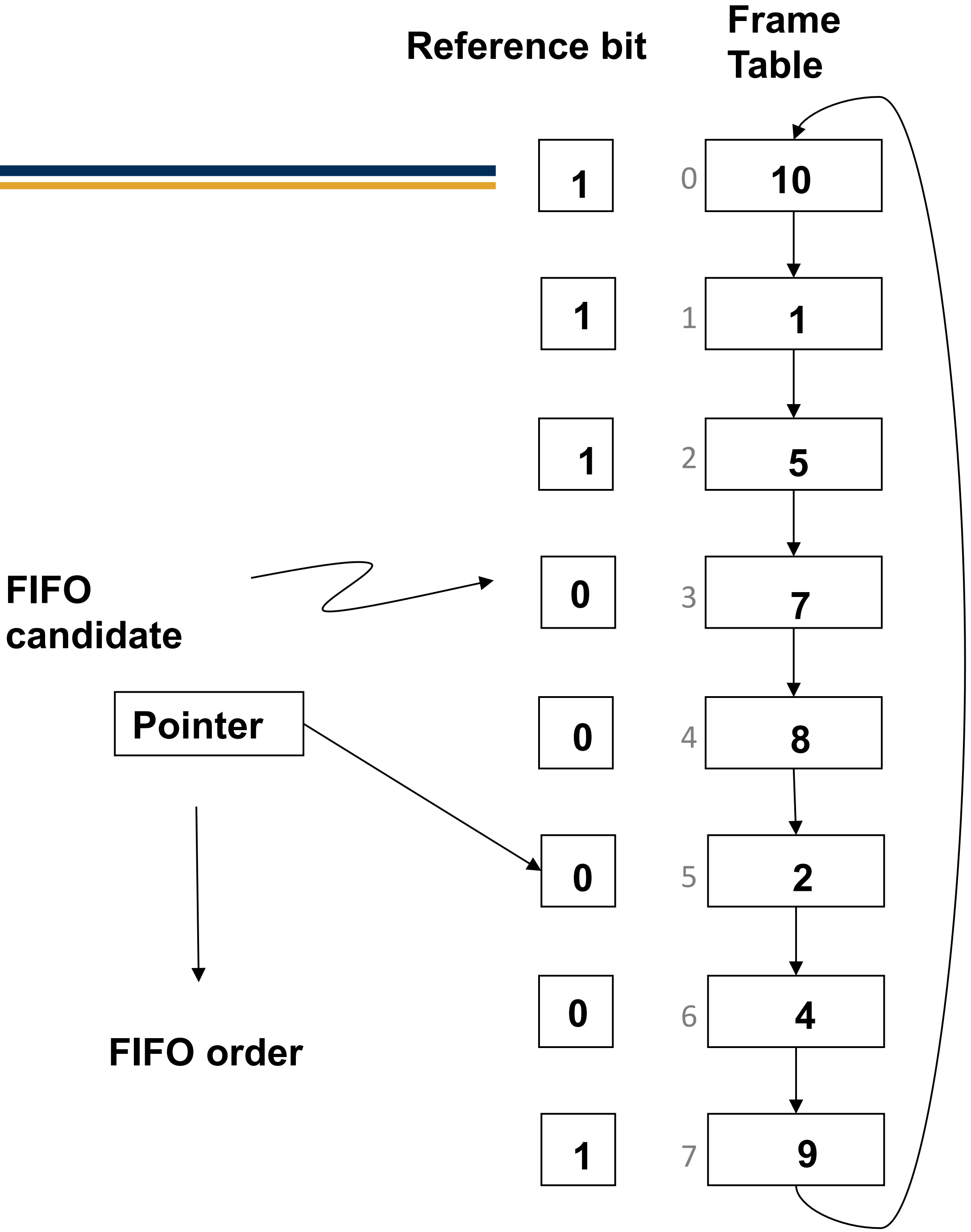


	Reference bit	Frame Table
0	1	10
1	1	1
2	1	5
3	0	7
4	0	8
5	0	~~3~~
6	0	4
7	1	9

Page out page 3

Page in page 2

Second chance replacement

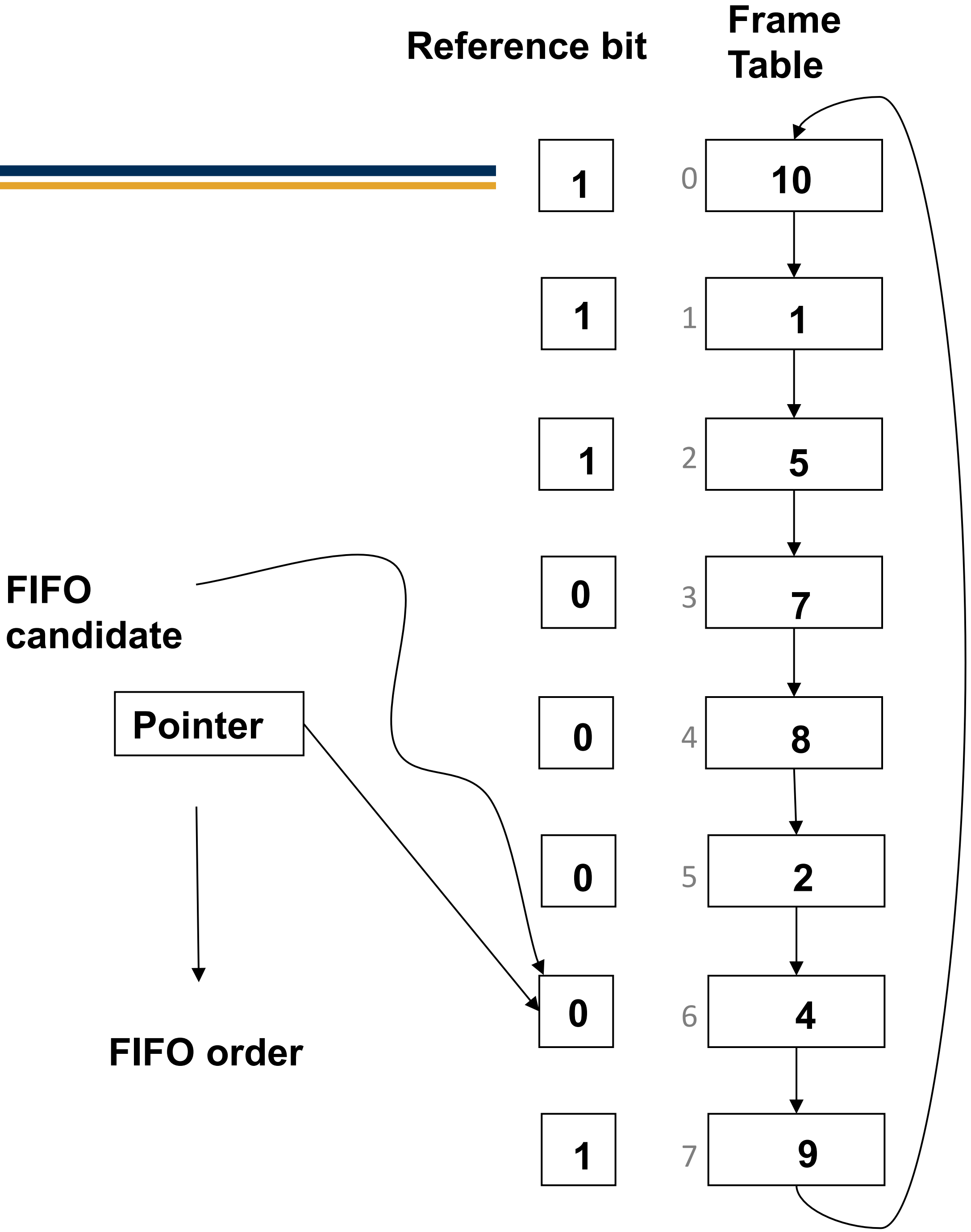


	Reference bit	Frame Table
0	1	10
1	1	1
2	1	5
3	0	7
4	0	8
5	0	2
6	0	4
7	1	9

Page out page 3

Page in page 2

Second chance replacement



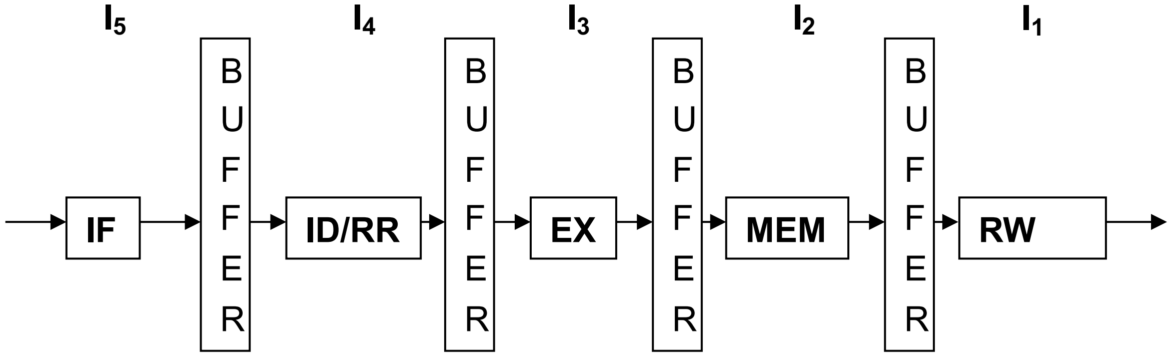
Move the pointer past the just-read page 2

Wait for the next page fault…

Page replacement algorithms

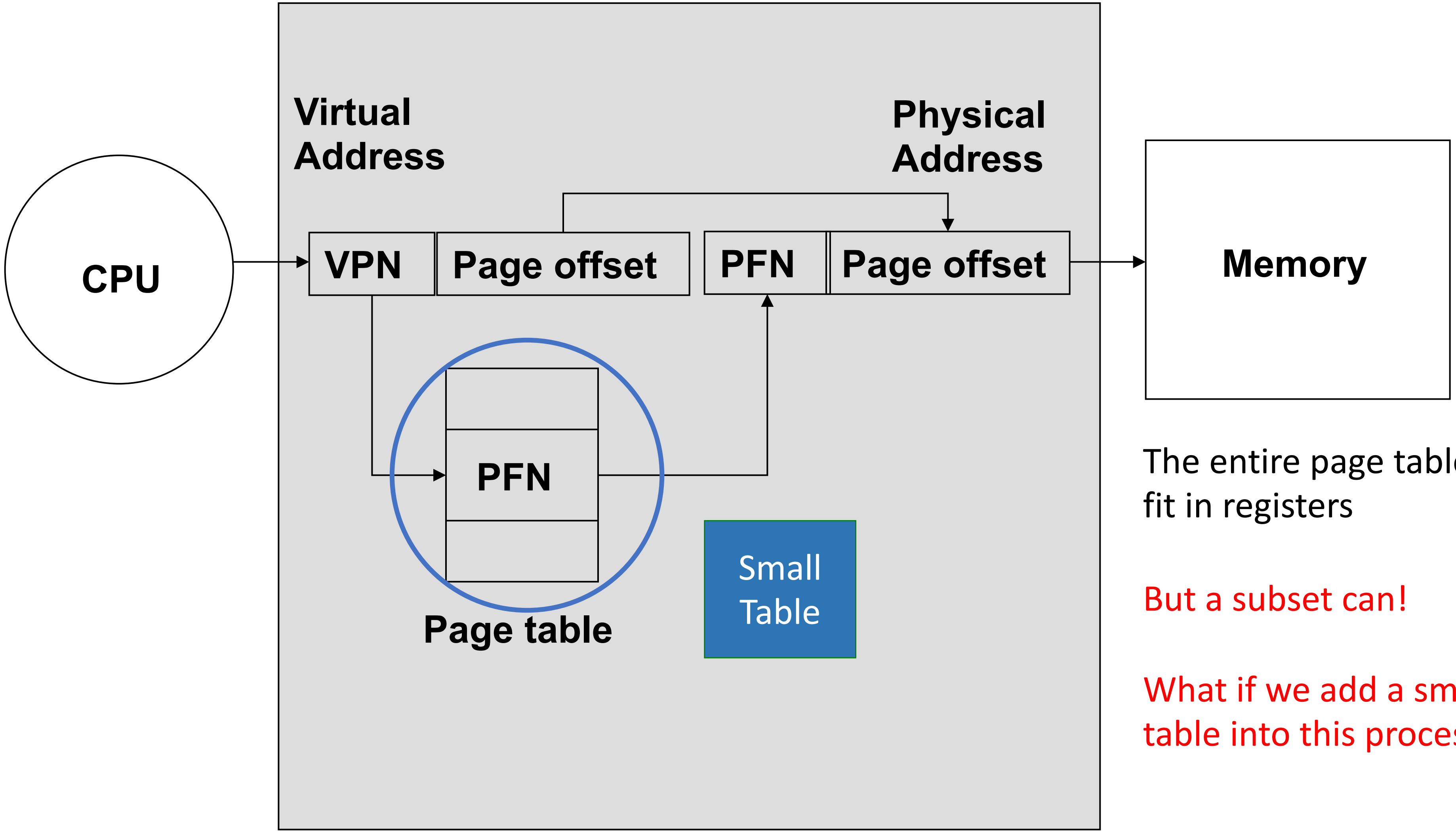
ALGORITHM	HARDWARE ASSIST	COMMENTS
FIFO	None	Could lead to performance anomalies
Belady’s MIN	An oracle	Provably optimal; not realizable in hardware; useful as a standard
True LRU	Push down stack	Expected performance close to optimal; infeasible
Approximate LRU #1	A small hardware stack	Expected performance 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

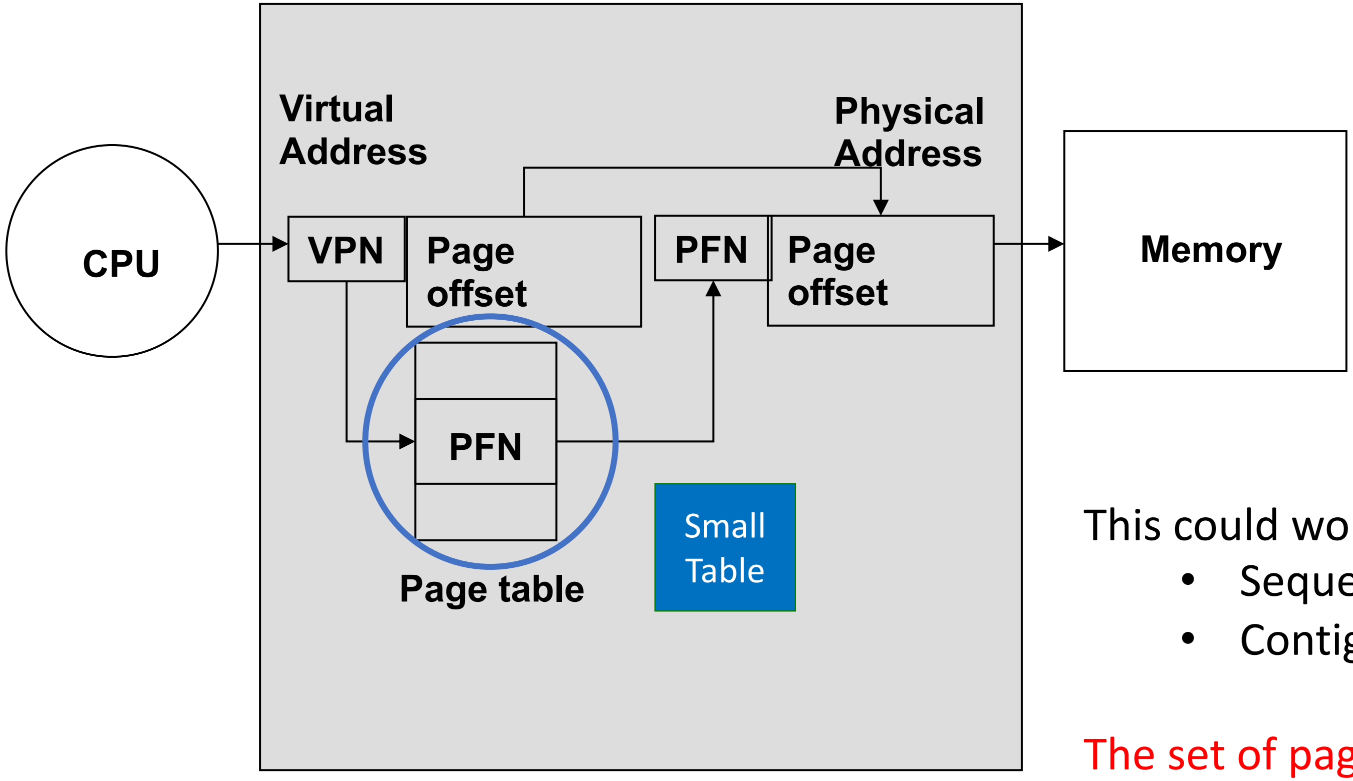


The entire page table won't fit in registers

But a subset can!

What if we add a small table into this process?

Why will this work?



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

- Sequential instructions
- Contiguous data structures

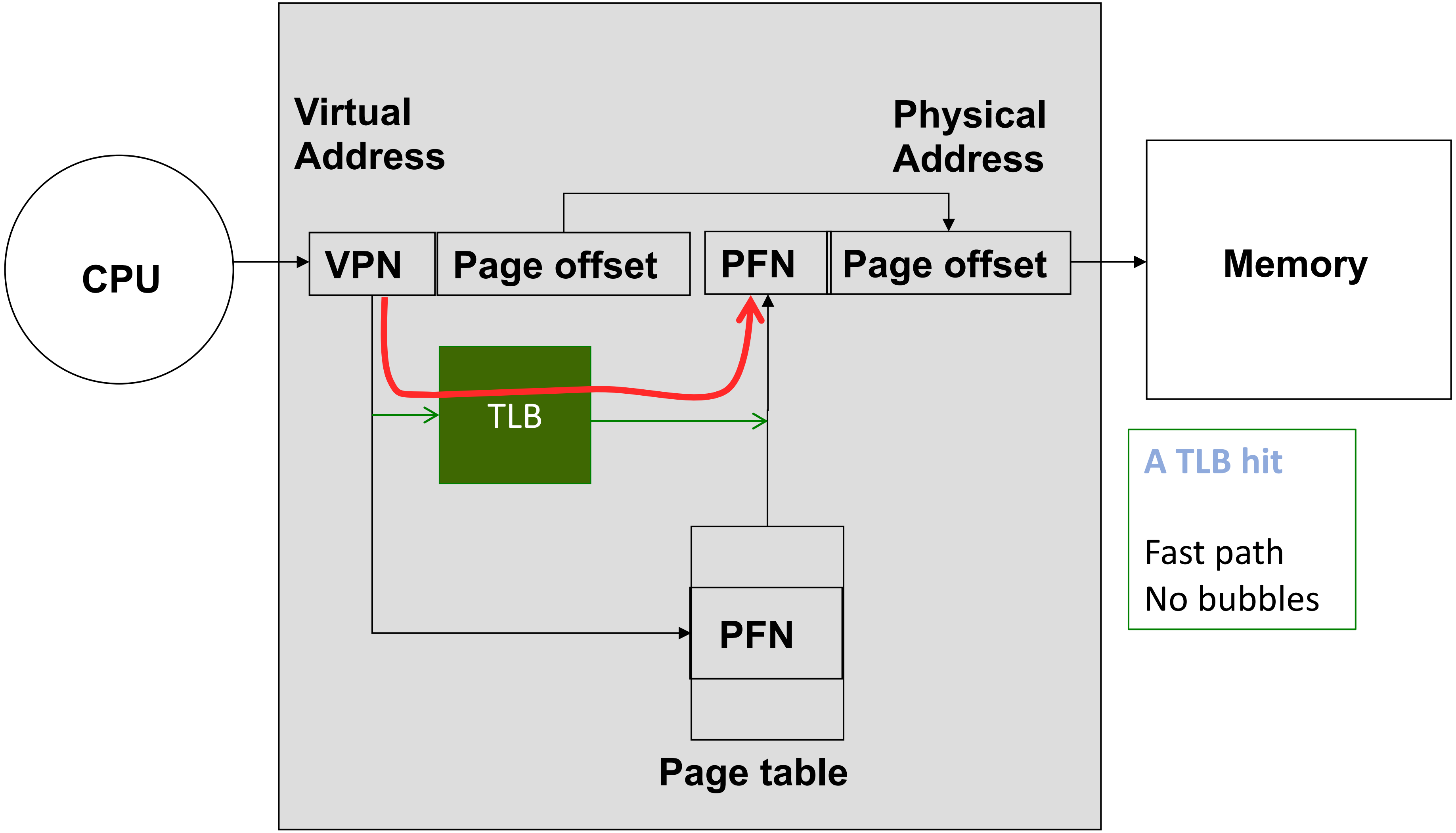
The set of pages being referenced at a given time is called the **Working Set**

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

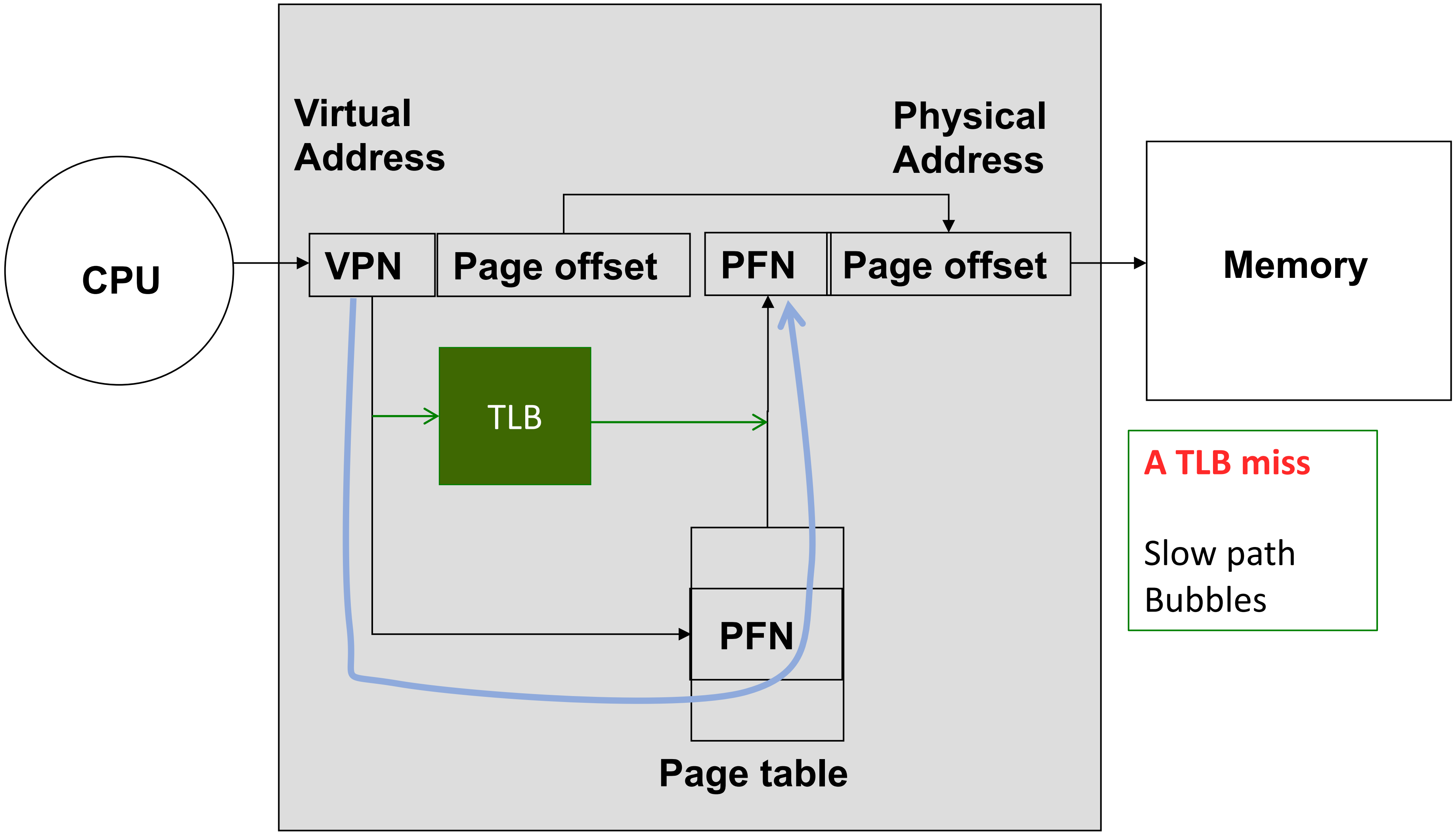
Speeding up address translation



A TLB hit

Fast path
No bubbles

Speeding up address translation



A TLB miss

Slow path
Bubbles

TLB

	USER/KERNEL	VPN	PFN	VALID/INVALID
Specific to each process	U	0	122	I ~~V~~
	U	XX	XX	I ~~I~~
	U	10	152	I ~~V~~
	U	11	170	I ~~V~~
Same for all processes	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…

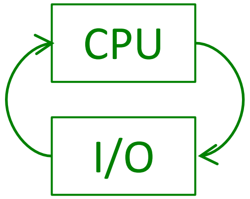
A. The entire TLB must be flushed

B. Only the kernel portion of the TLB must be flushed

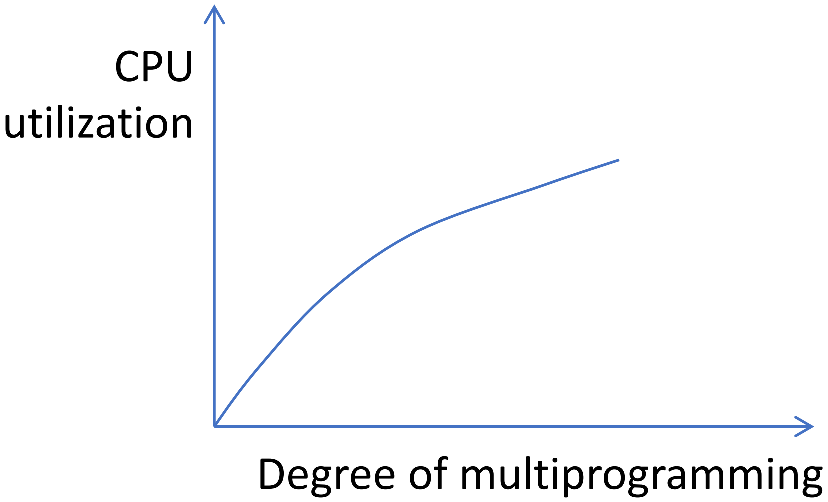
C. Only the user portion of the TLB must be flushed

D. Leave the poor TLB alone!

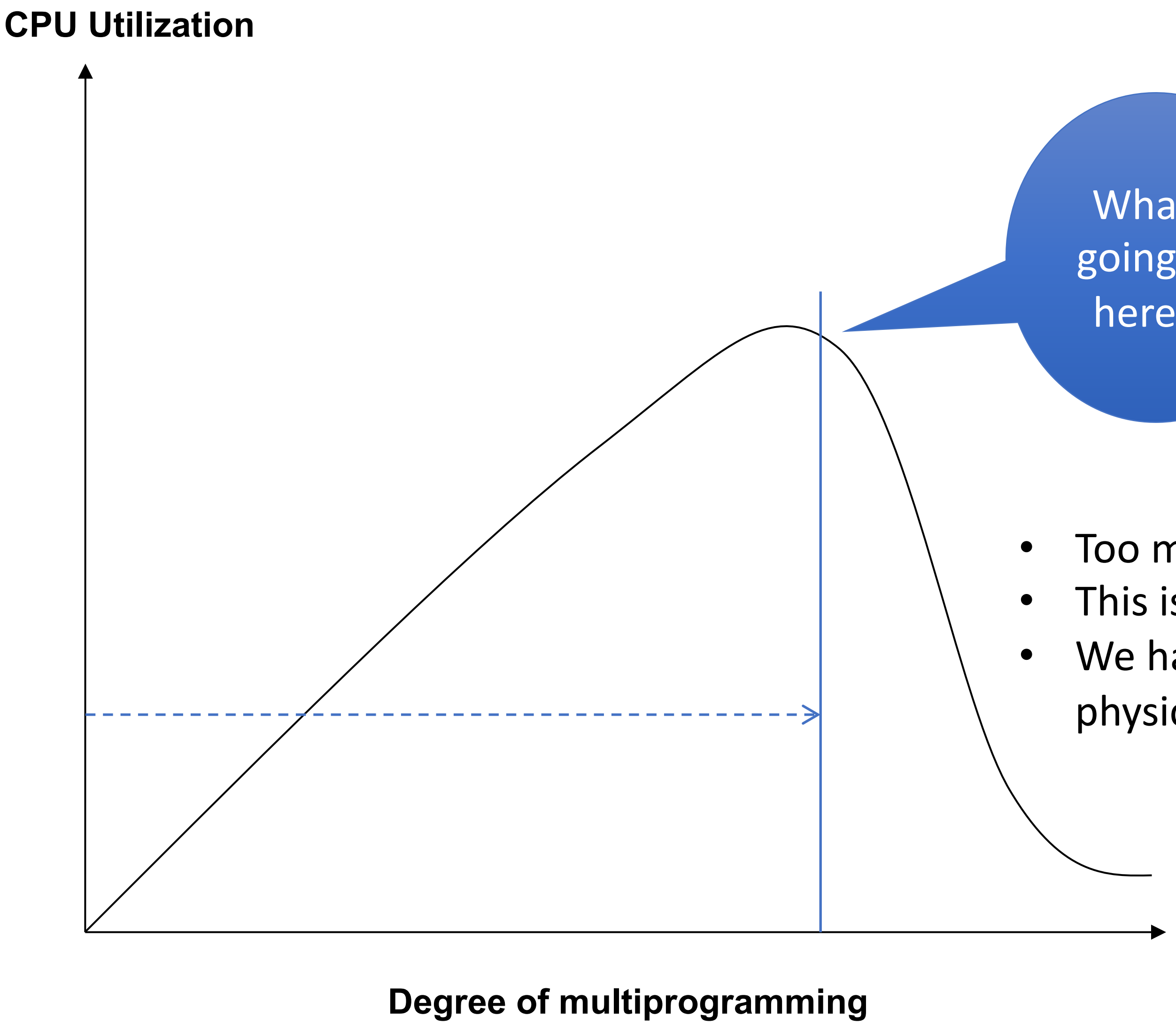
Given the nature of a process



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



Extending the utilization curve



What's going on here??

Thrashing

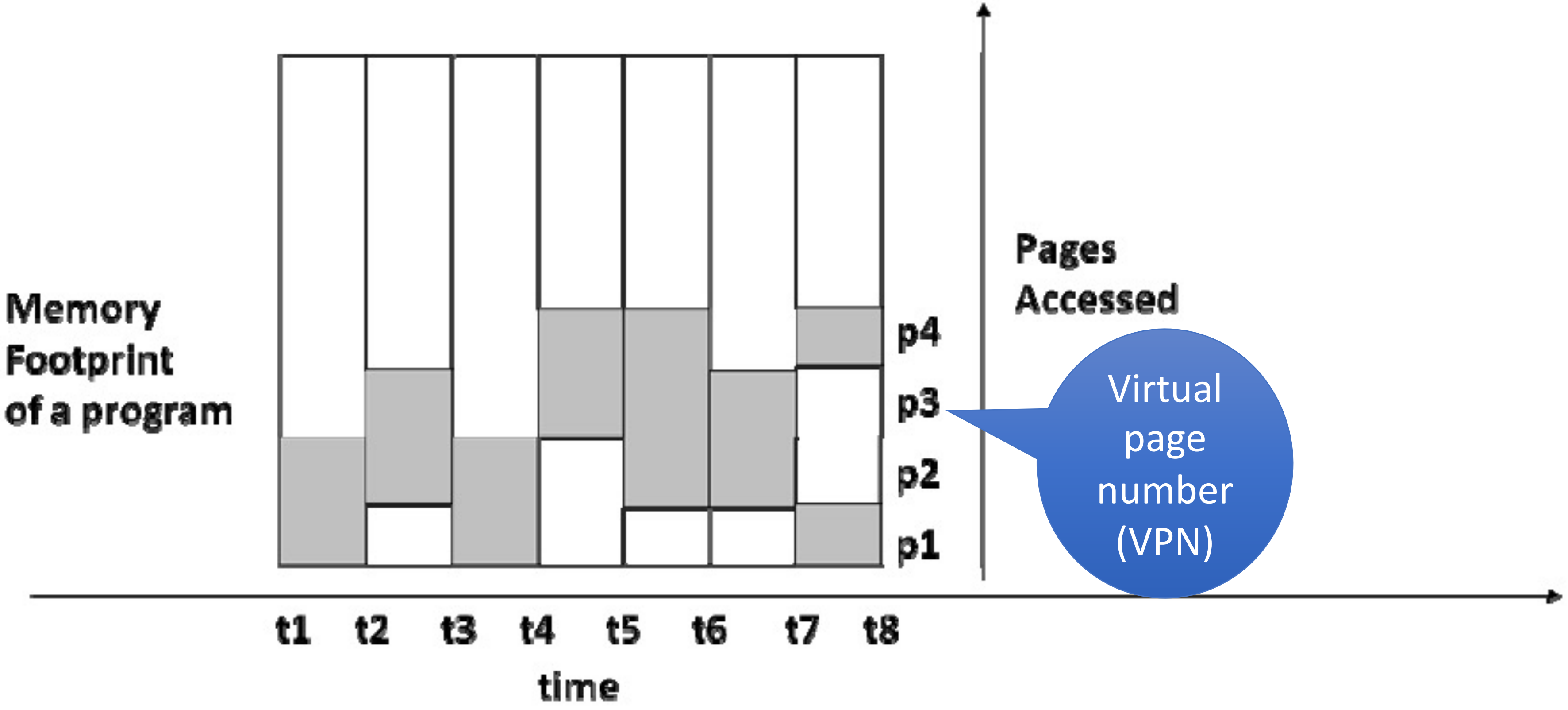
Nobody is getting any useful work done

- 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

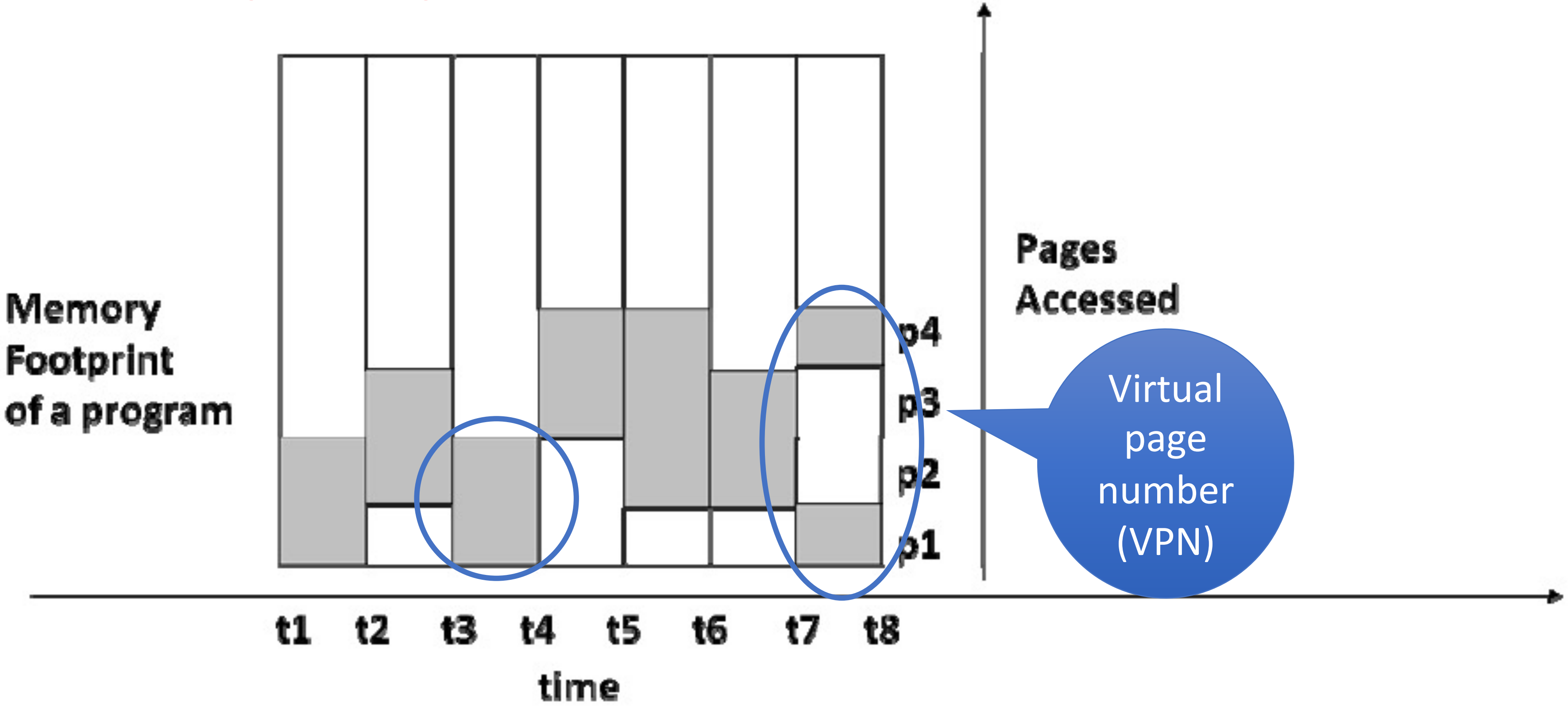
Working set is the set of pages needed to keep a process from paging



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

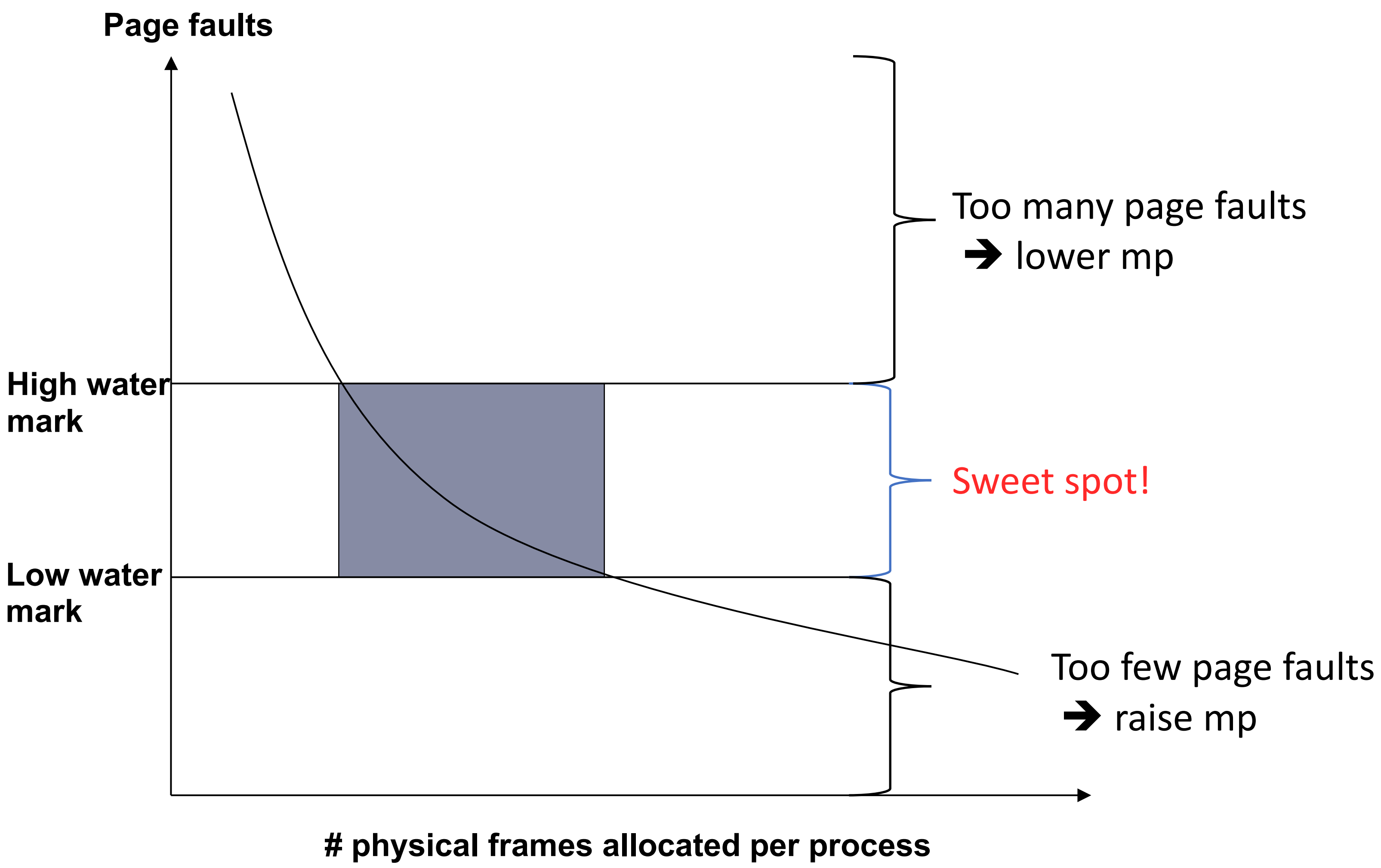
$$memory.pressure = \sum_{i=1}^{n} WSS_i$$

- P_1, P_2, P_3, … 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



Too many page faults
➔ lower mp

Sweet spot!

Too few page faults
➔ raise mp

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. It has too few page faults per second

B. It has too many page faults per second

C. The ratio of I/O operations to CPU operations is not optimal

D. 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. Using a medium-term scheduler that suspends processes until the condition improves

B. Reducing the physical memory size

C. Adding additional processes to increase the multiprogramming factor

D. Reducing the page size

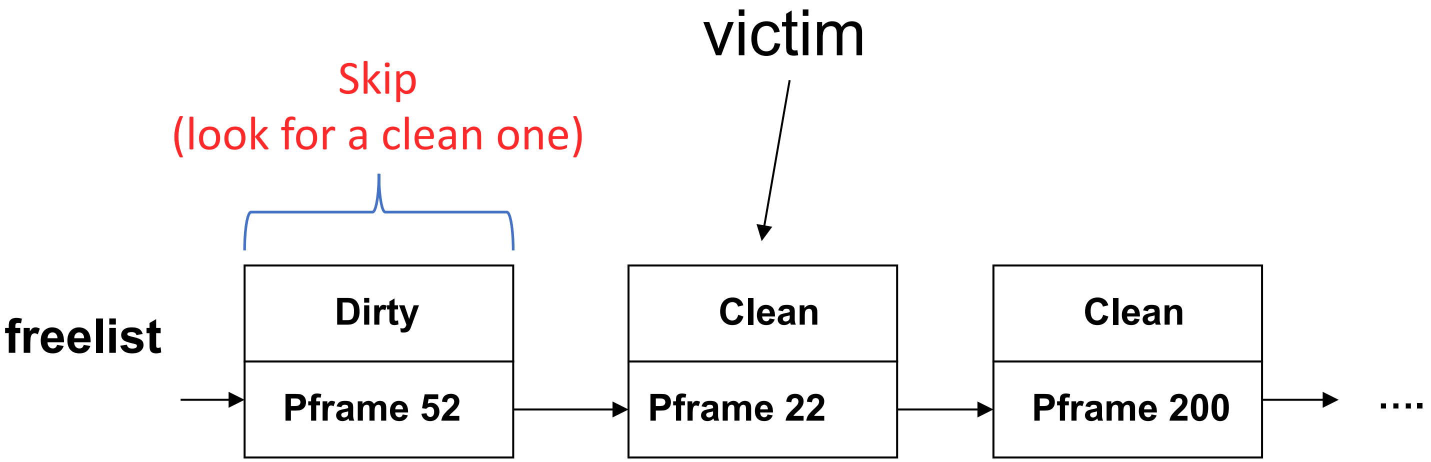
Of course this begs the question of how the medium-term scheduler is going to figure out that the system is thrashing…

Paging Optimizations

Accelerate page-in by removing functionality from critical path

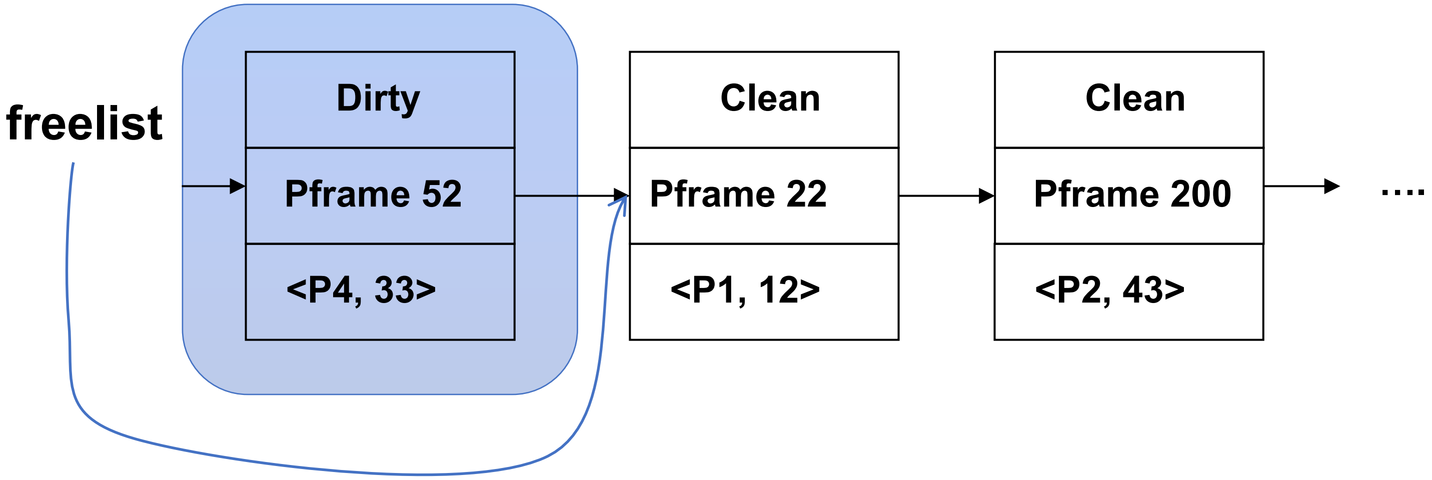
- Keep a (small) pool of free frames
  - Don't wait to start page replacement algorithm on a page fault

- 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



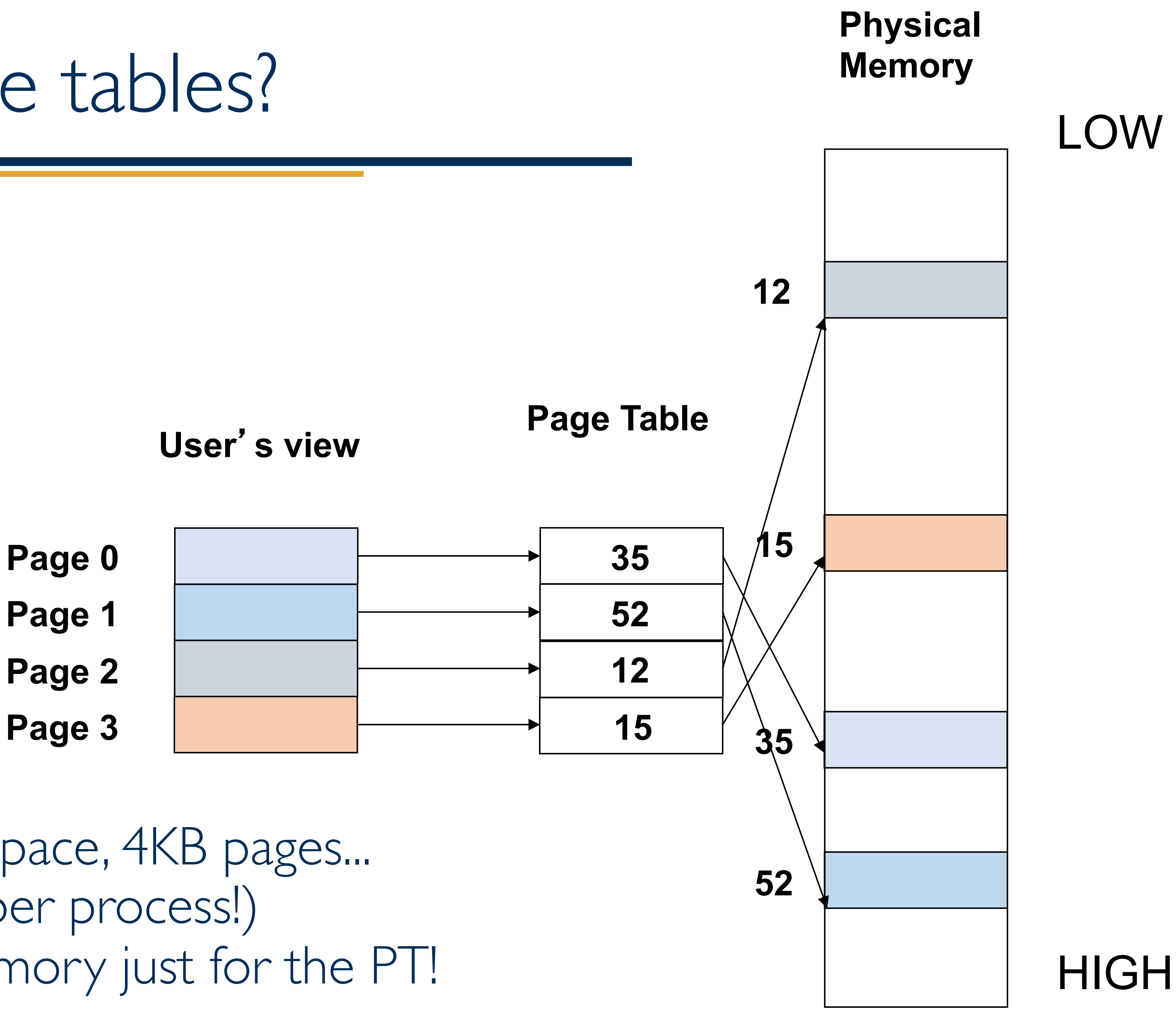
Linux VM and kswapd

```
$ 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

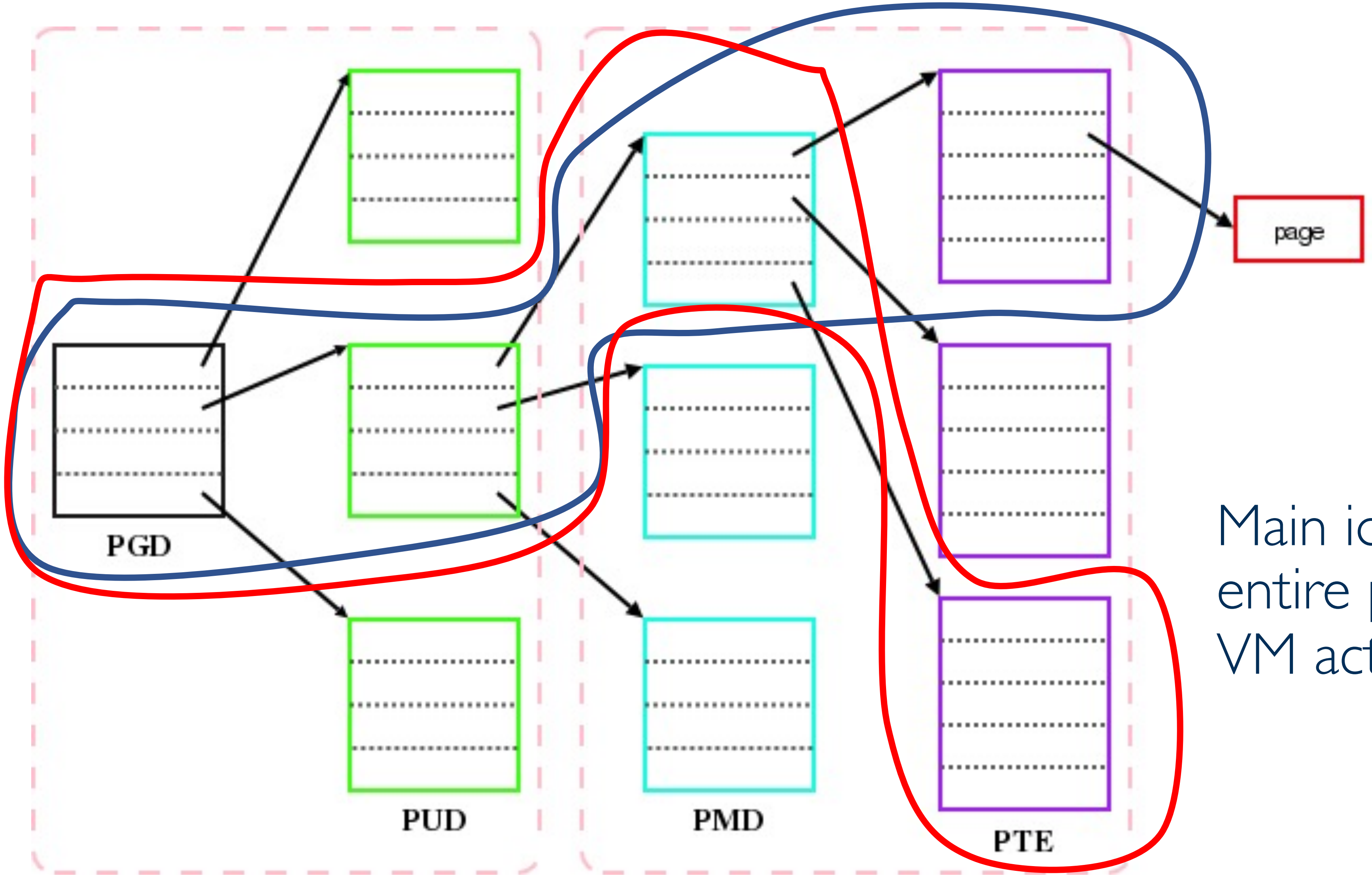
How big are page tables?



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

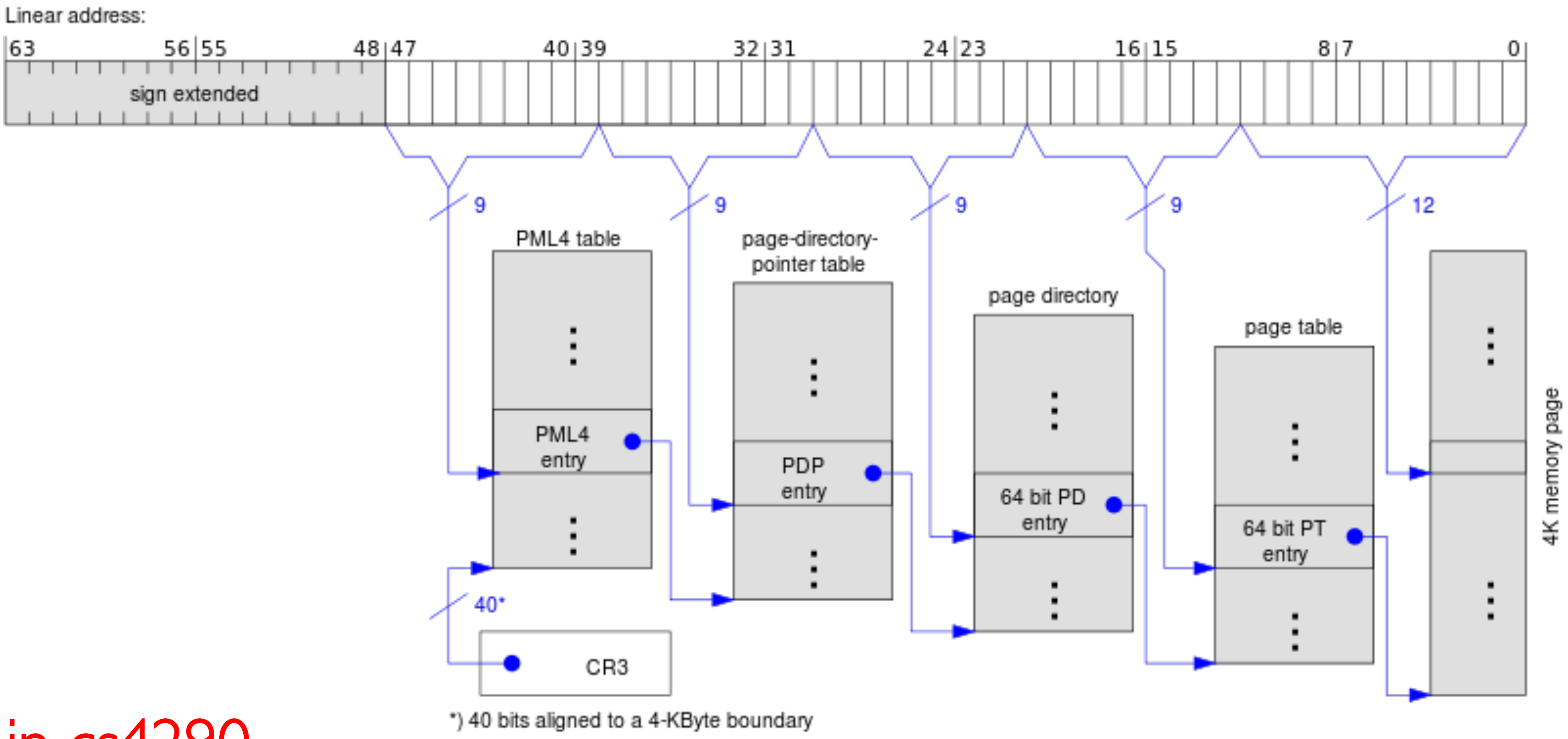
- Need 2^36 PT entries (per process!)
- That's several GBs of memory just for the PT!

Teaser: Hierarchical Page Table



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

Intel's X86-64 4-level Page Tables



More about this in cs4290