



# Memory Management

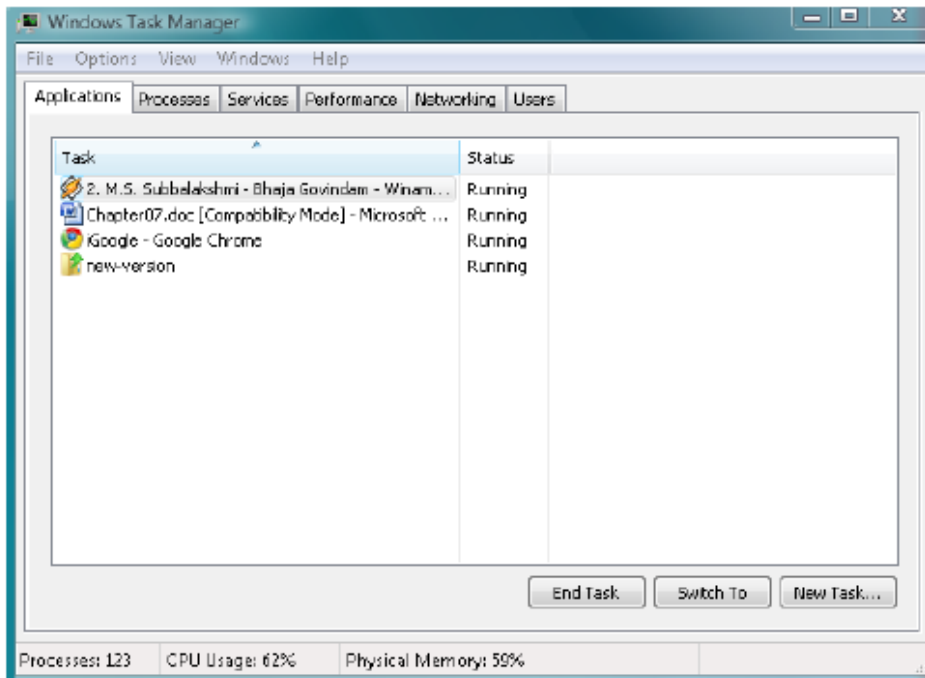
Ramachandran & Leahy, Chapters 7 & 8



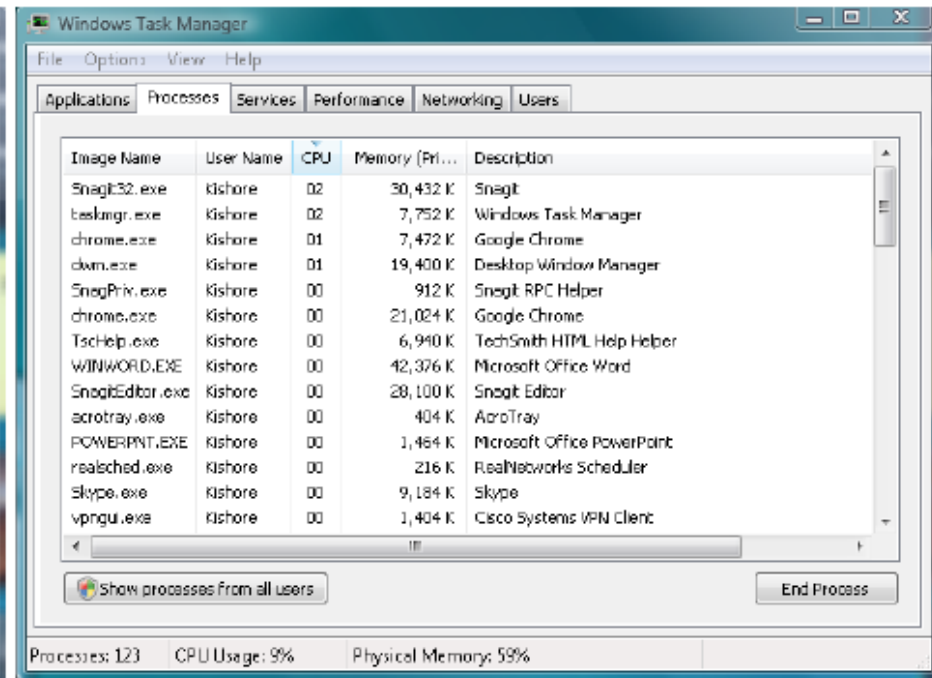
# So far

Software	Hardware
High level language + compiler + linker	ISA, addressing modes, stack, registers
Program discontinuities (INT handler in OS)	Interrupt support, processor implementation (simple + pipeline)
OS process scheduler + loader	Process as an abstraction, context switching
OS memory management	Hardware support for memory management

# A “small” laptop’s workload



↑ 4 applications



↑ 123 processes!

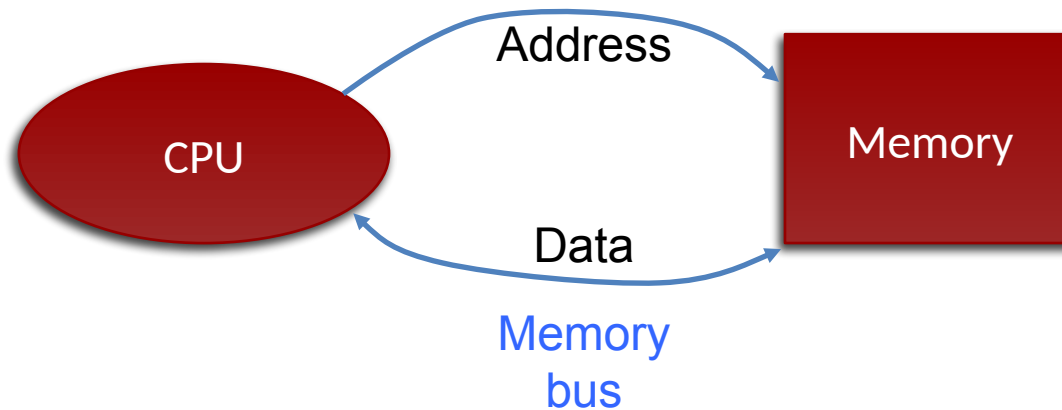
1208MB of 2GB memory in use

# Goals of memory management

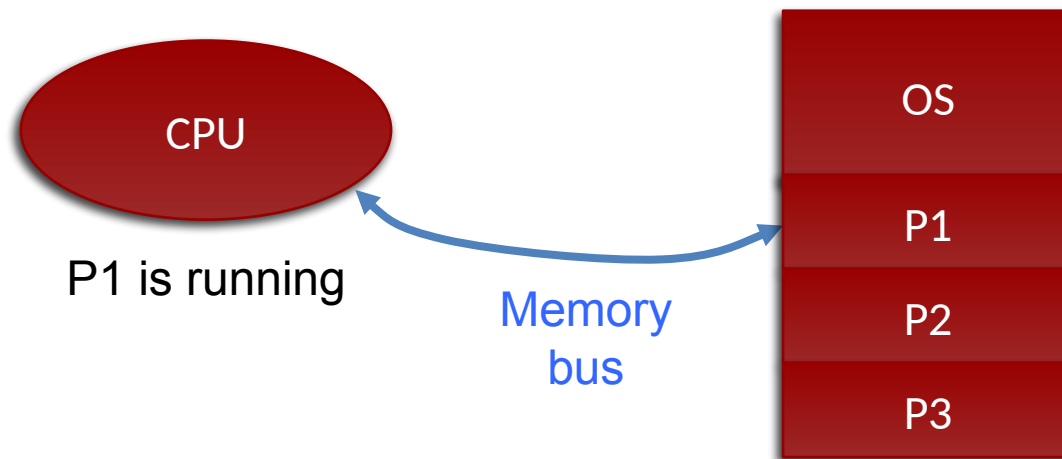
**What functionalities do we want to provide?**

- Improved resource utilization
- Independence and protection
- Liberation from resource limitations
- Sharing of memory by concurrent processes
- So far there's no hardware in the LC-2200 to “manage” memory

# Nothing in LC-2200 to manage memory

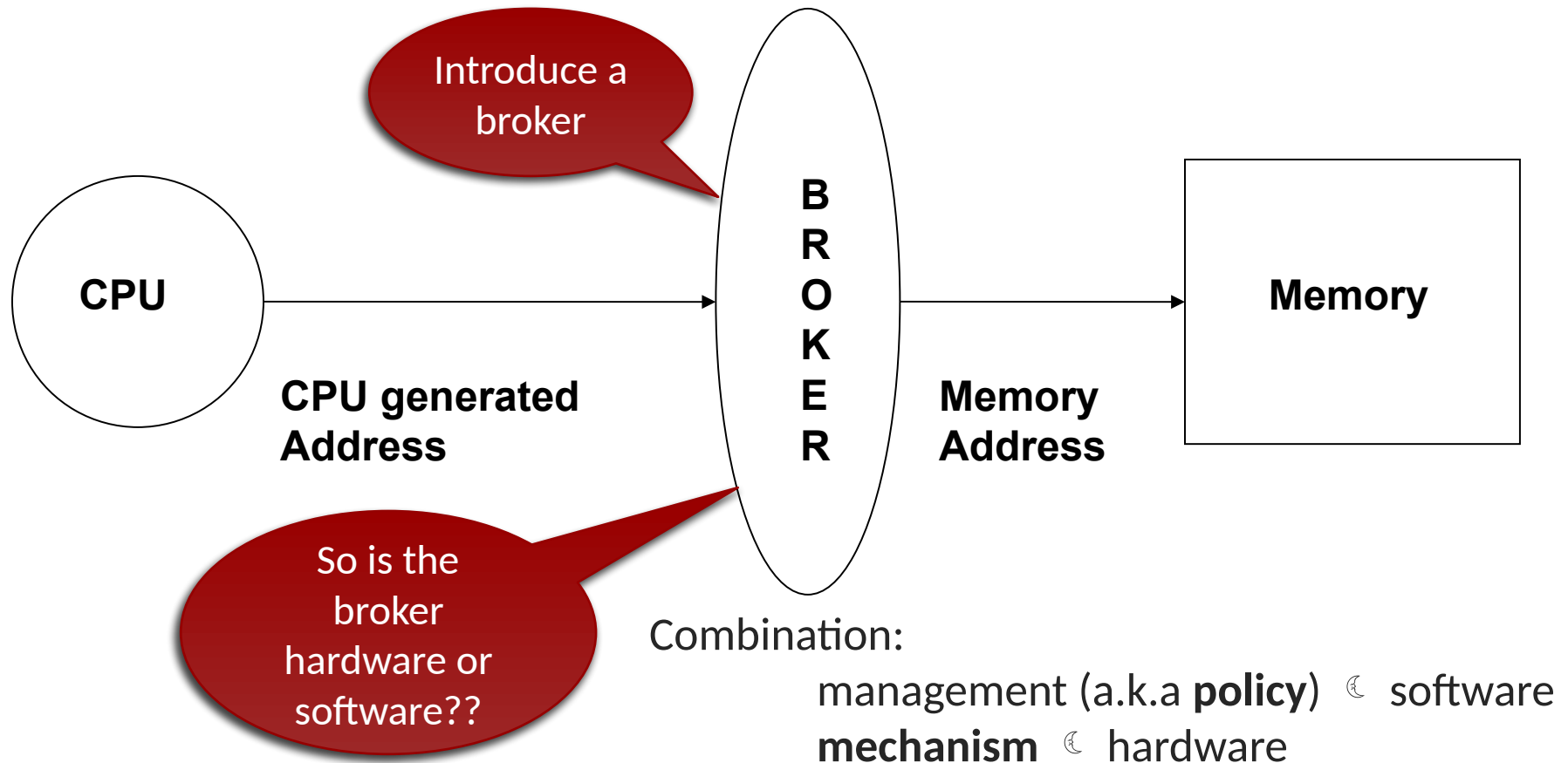


# What's missing in LC-2200?



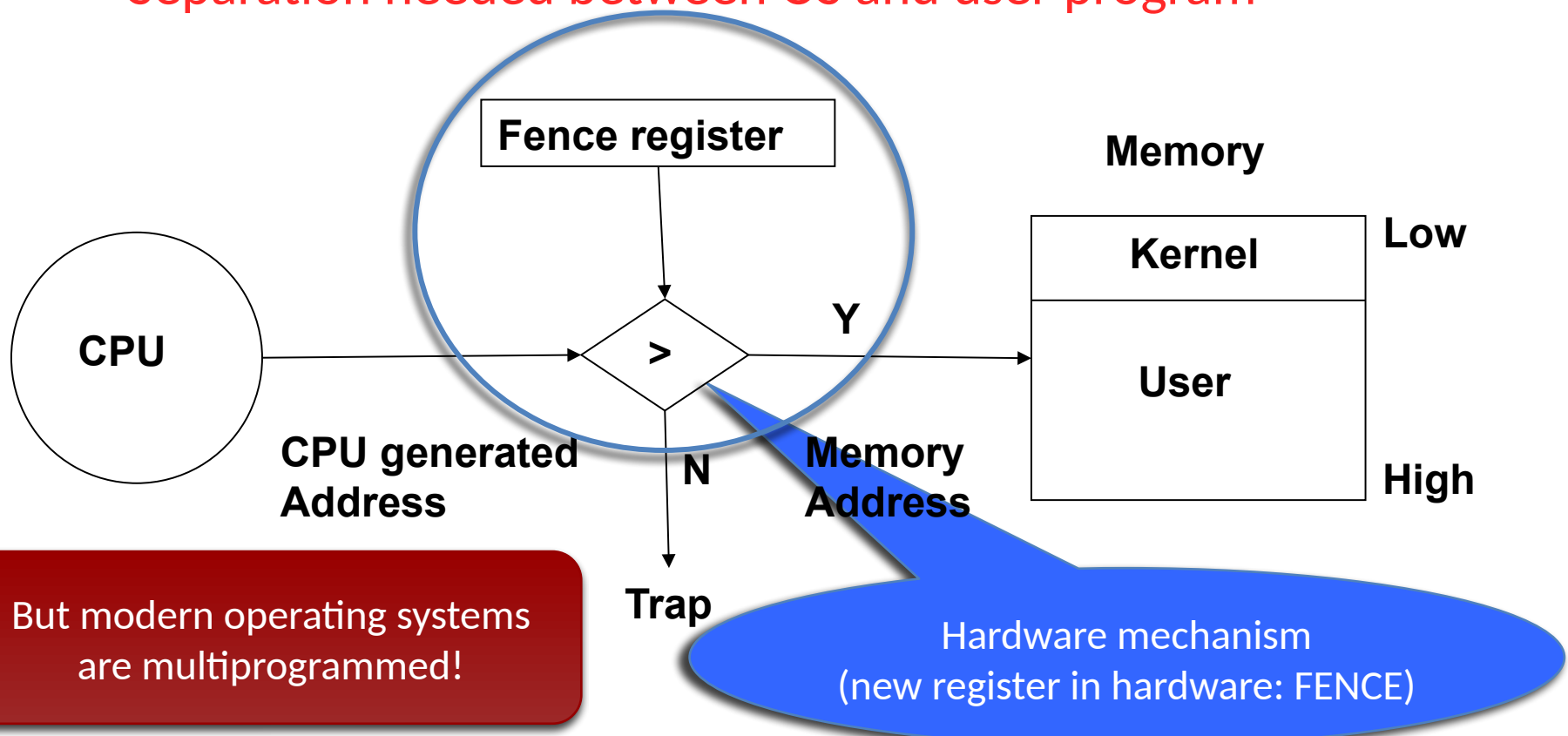
- Nothing to protect the OS, P2, and P3 from P1
- No way to move P1 – it knows where it is loaded in memory
- Can't run more processes than we have memory for
- Memory use is exclusive to a process

# To manage memory...



# Simple management

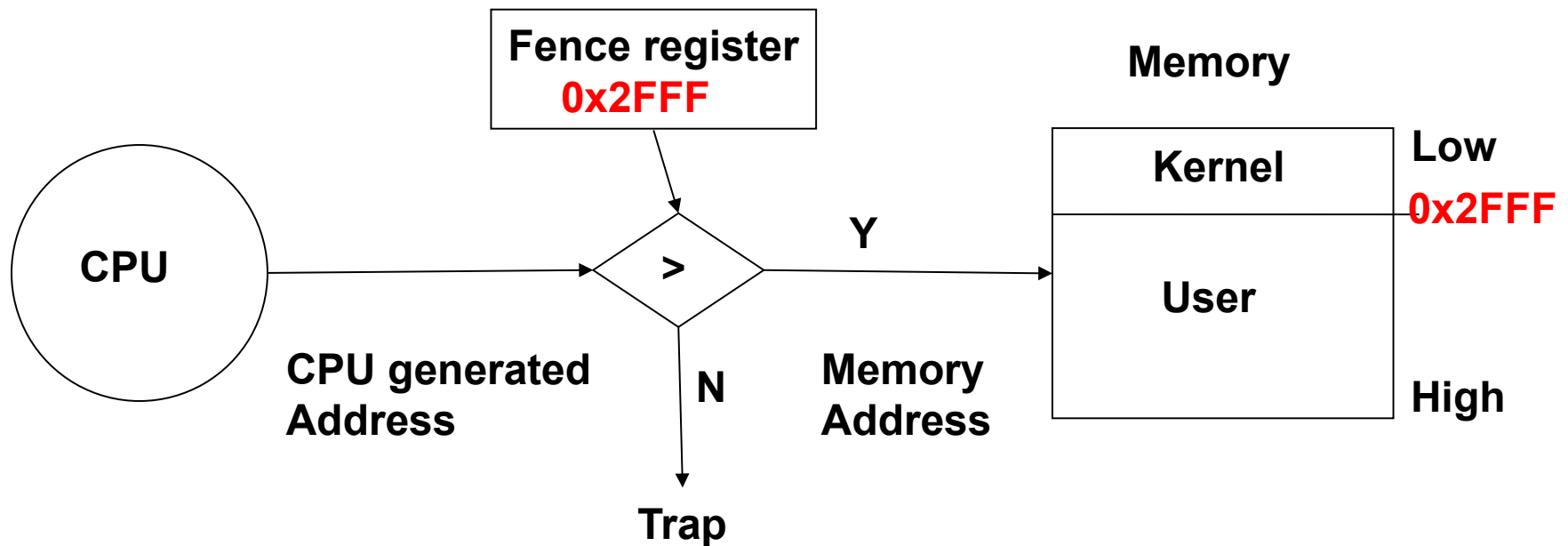
- One user process at a time
- Separation needed between OS and user program





# Fence register example

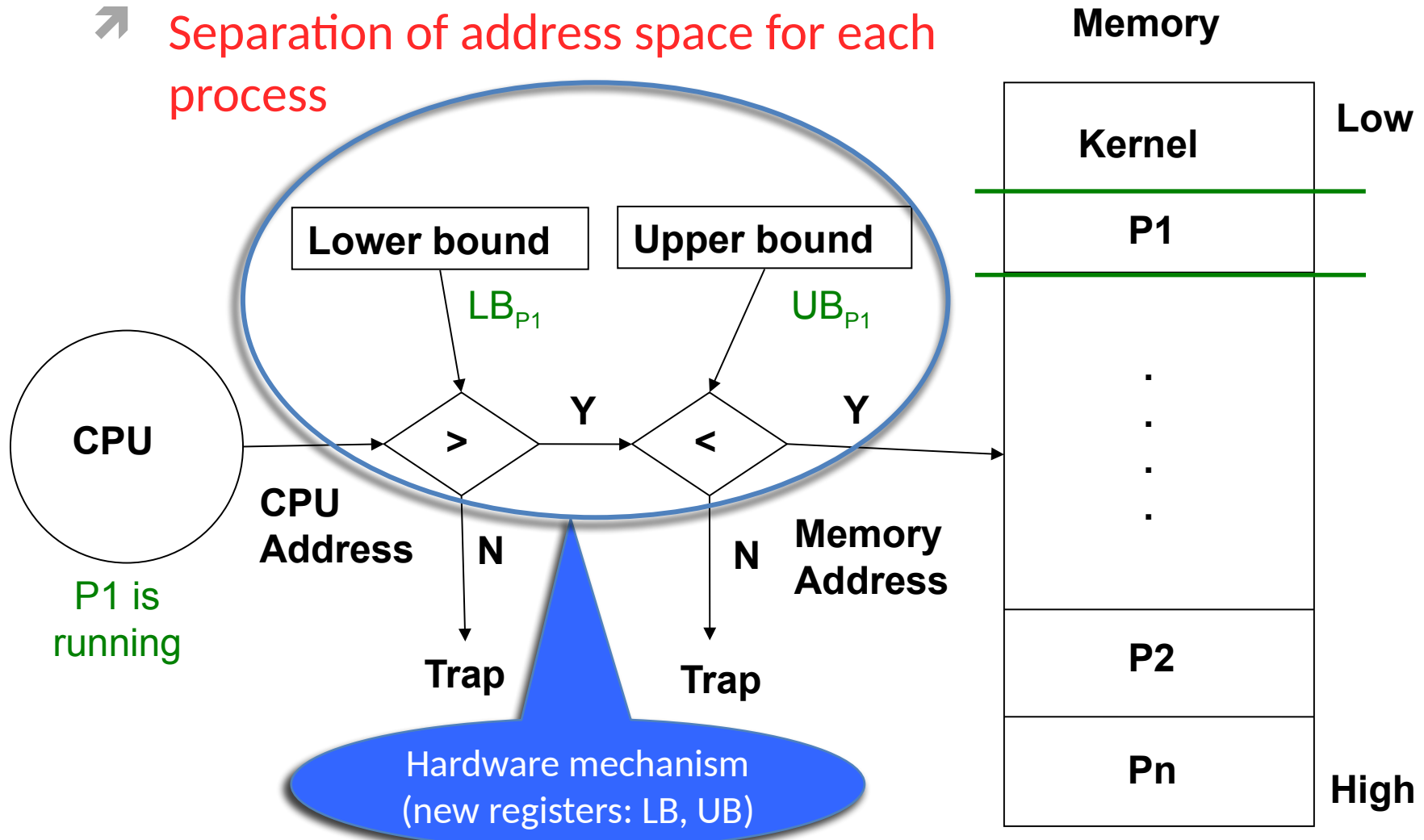
- Provide protection for the OS
- Performed by *hardware* without help from the CPU



If the CPU is in "kernel" state, execute the memory operation  
Else if address is  $\geq 0x3000$ , execute the memory operation  
Else cause a "memory protection violation" trap

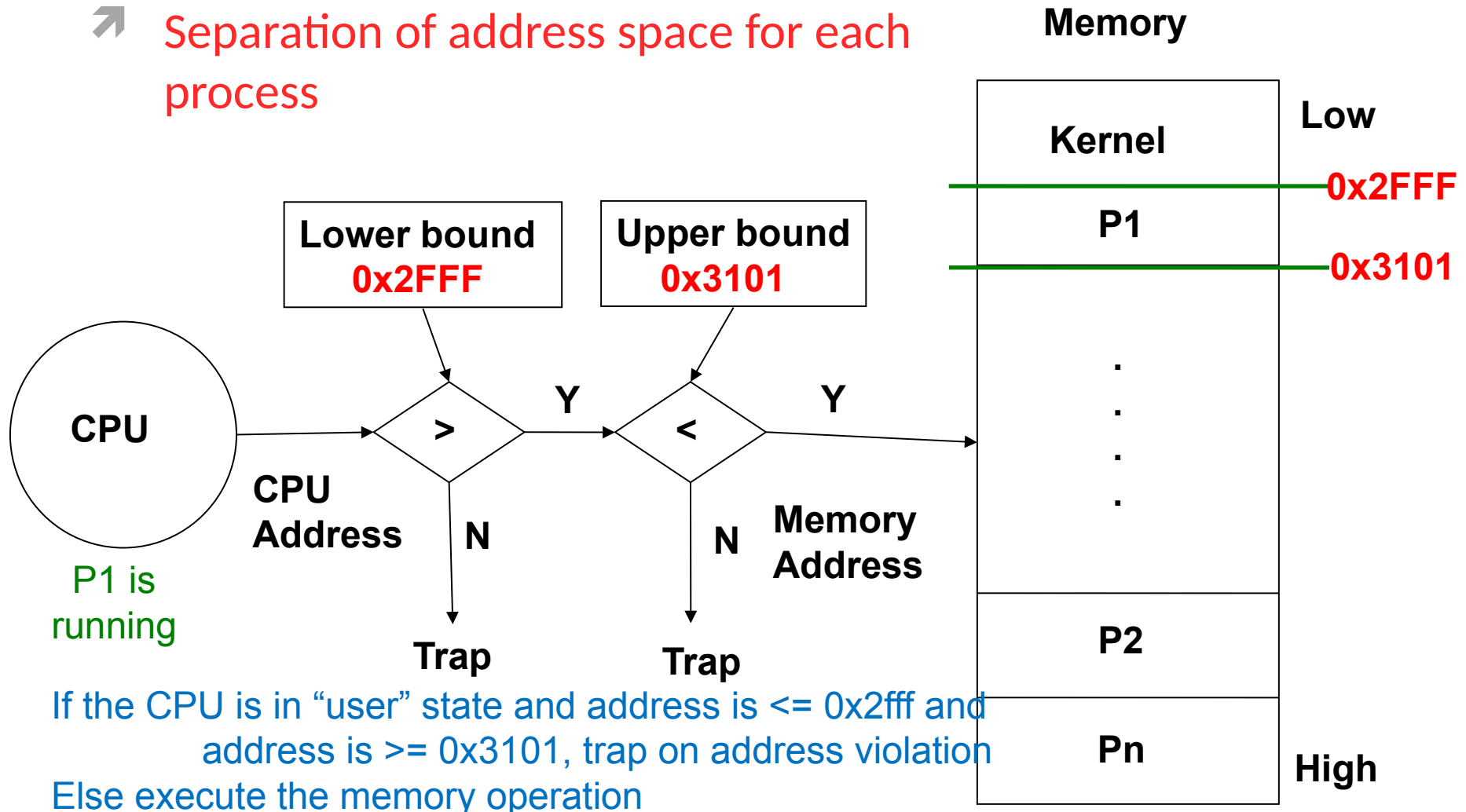
# Multiprogrammed OS

➤ Separation of address space for each process



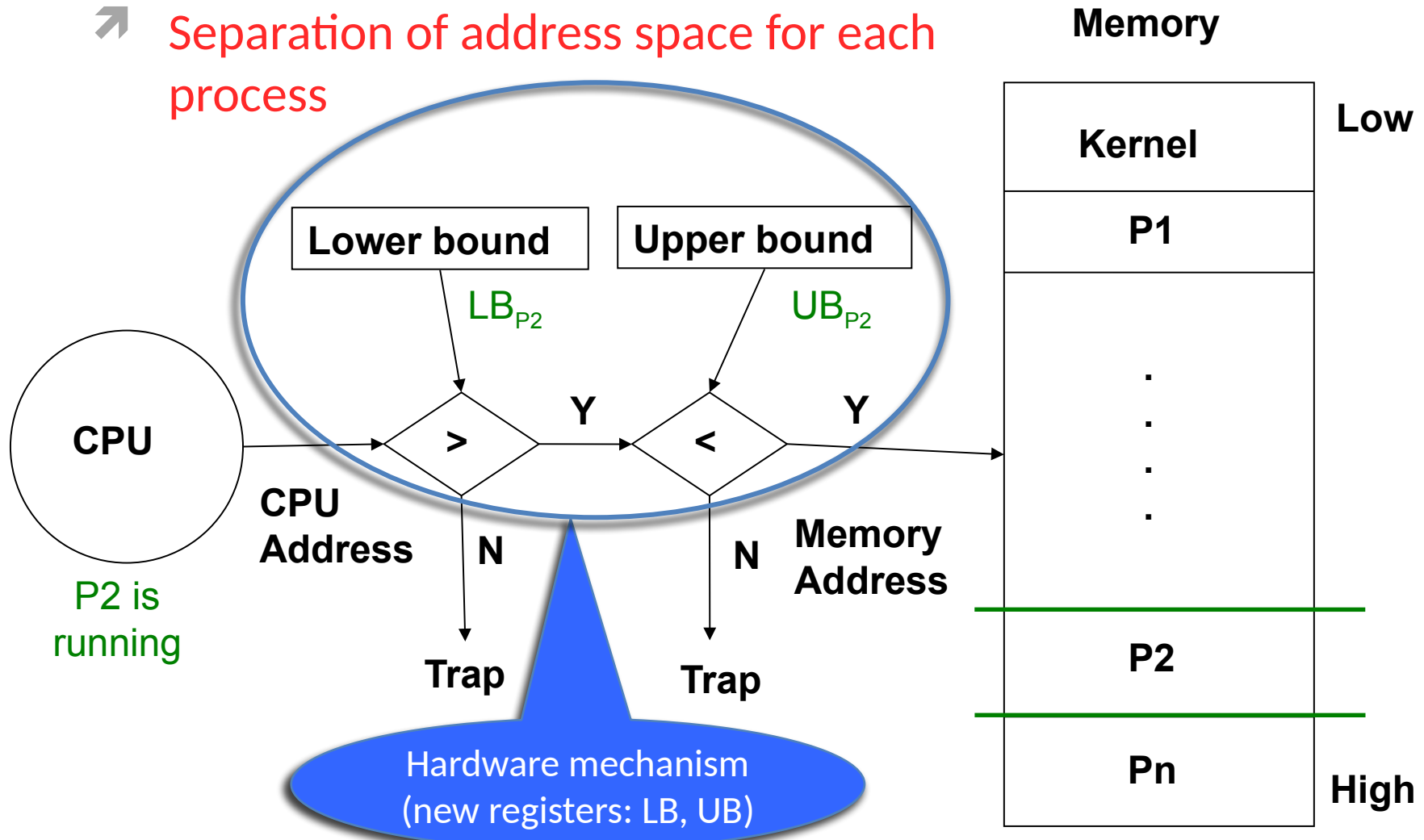
# Bounds registers example

➤ Separation of address space for each process



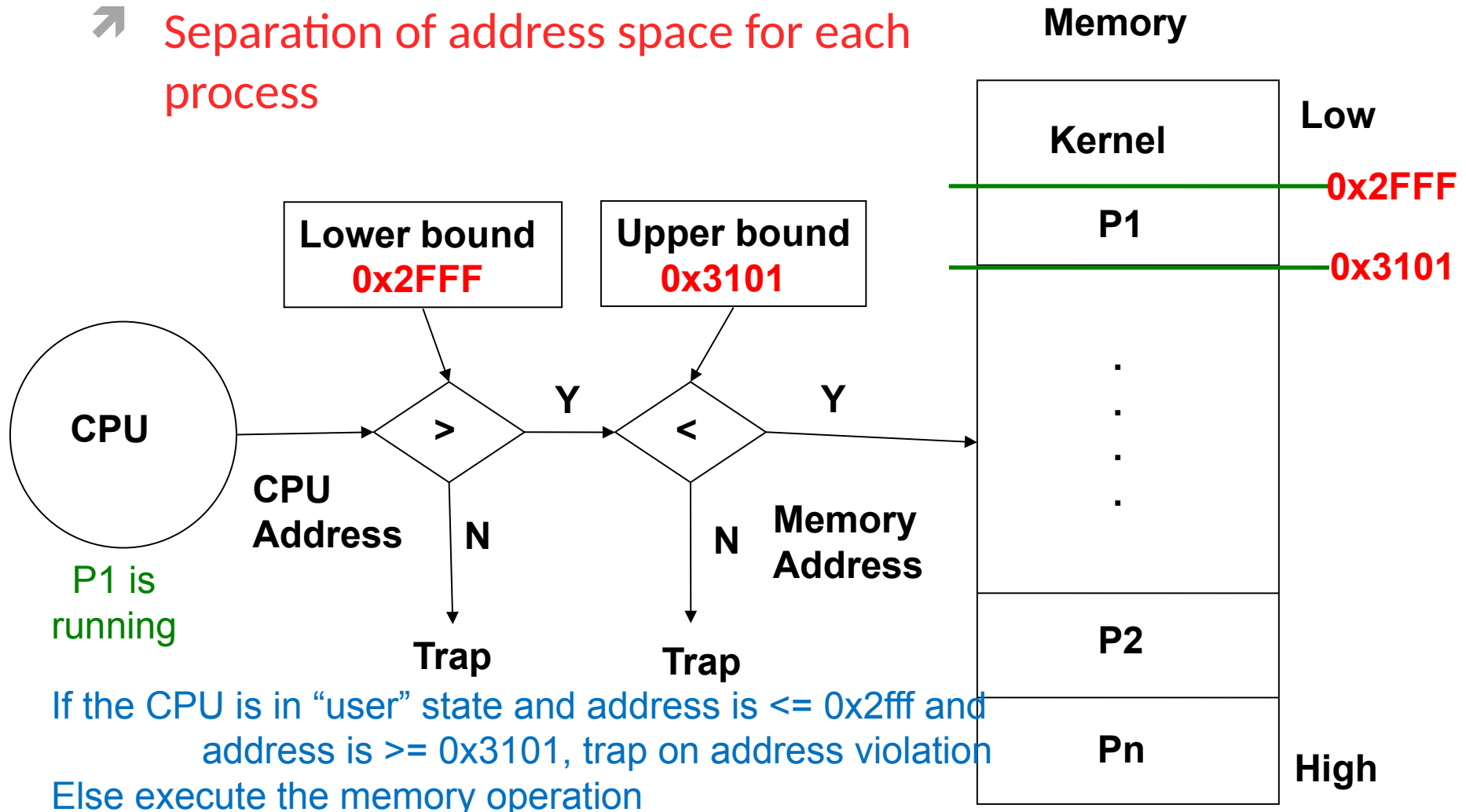
# Multiprogrammed OS

➤ Separation of address space for each process



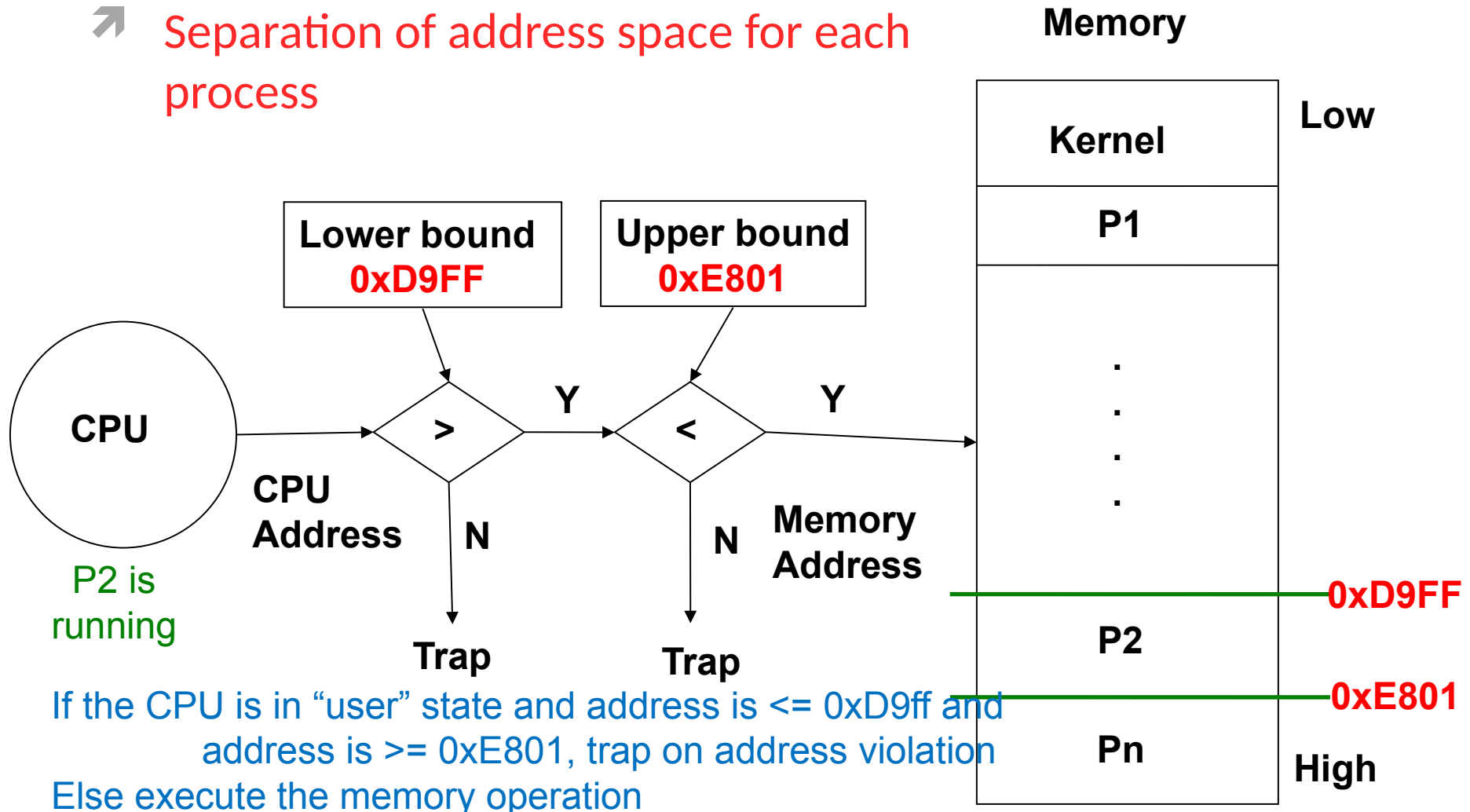
# Bounds registers example

➤ Separation of address space for each process




# Bounds registers example

➤ Separation of address space for each process



# Question

What needs to happen to ensure that LB and UB correspond to the currently running process?

- A. Restore LB and UB to the system stack
- B. Restore LB and UB to values defined in the source code file
-  C. Restore LB and UB to values from the PCB
- D. Calculate LB and UB from the process-id

The number of the day is 36,040.

# 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 memory_footprint; ???
    ...
    ...
} control_block;
```

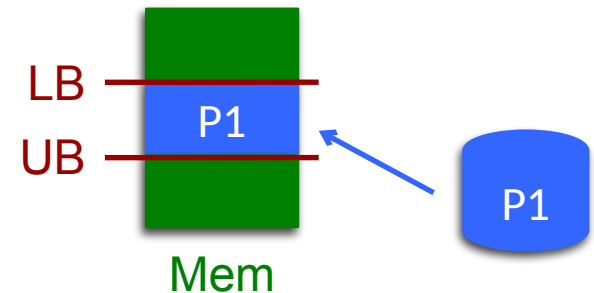


# PCB

```
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 LB;
    address UB;
    ...
    ...
} control_block;
```

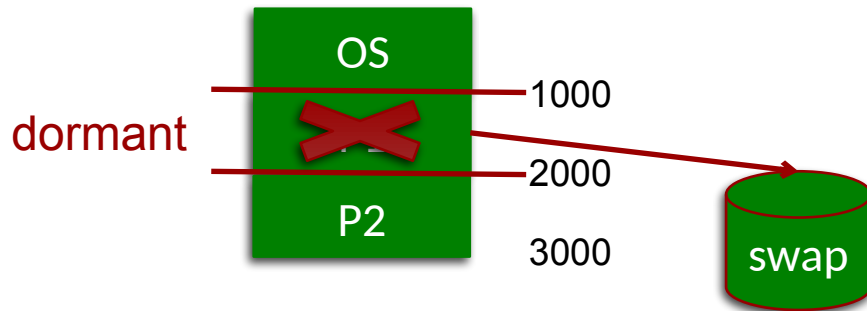
# “Management” by the OS

- At load time (loader)
  - Place P1 in memory
  - Set LB and UB in the PCB
- At context switch time (dispatcher)
  - Load LB and UB from the PCB into the new CPU registers



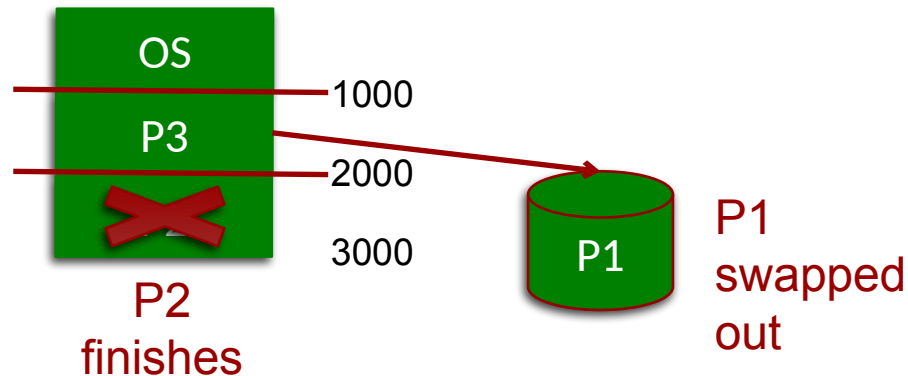
# Limits of “bounds register” mechanism?

- Relocation isn't possible
- Swapping-in can be a problem



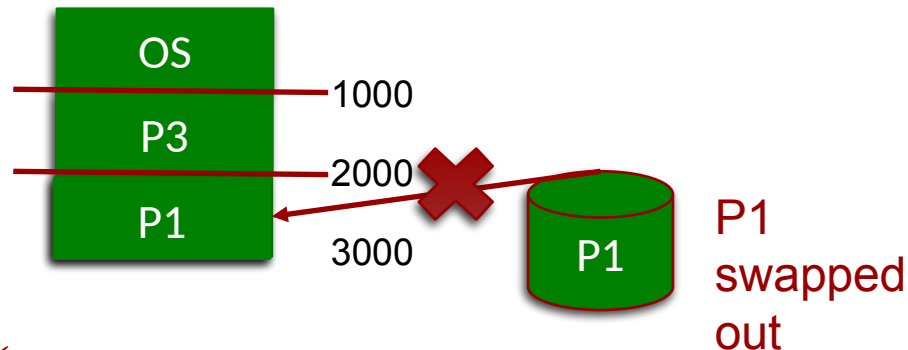
# Limits of “bounds register” mechanism?

- Relocation isn't possible
- Swapping in is a problem



# Limits of “bounds register” mechanism?

- Relocation isn't possible
- Swapping in is a problem



- ✓ We need to bring P1 back into memory
- ✓ Where?
- ✓ We have an empty spot...
- ✓ Will this work?

# Can we move an LC-2200 process?

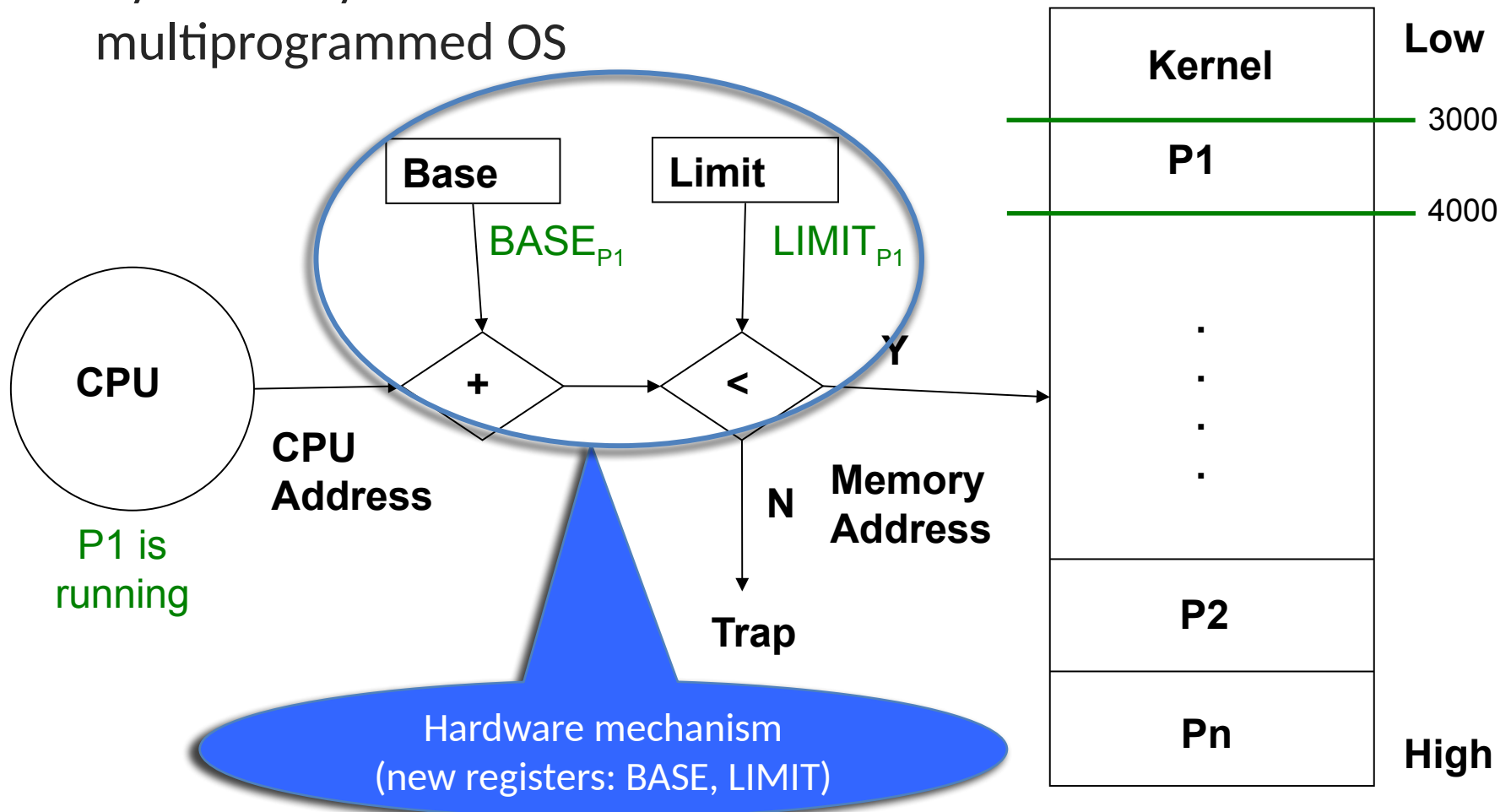
- When do we lock-in memory addresses?
  - At assembly/link time Ⓢ called static relocation
  - P1 can only be in memory between 1000 & 2000
  - That gives us poor memory utilization
  - i.e. not **dynamically relocatable**
- **So the program must be loaded into the address range into which it was linked. There's no way we could move it later.**
- How about we forbid absolute addresses in our program code?
  - It would be relocatable at load time

# Can we make P1 dynamically relocatable?

- Don't hard code addresses in executable
- Set memory bounds at load time rather than link time
- This implies making all addresses relative to some base register or the PC
- But what happens when we execute
  - LEA
  - JALR
- ... Memory addresses get saved in registers!
  - Where can they be copied or modified from there?
  - Can we find them so we can fix them?
  - This is why we'd say the LC-2200 code is **not dynamically relocatable**
  - **You still can't move LC-2200 code once it's started to run**
- IBM did this in System/360, but the CDC 6600 didn't...

# But there's a hardware solution...

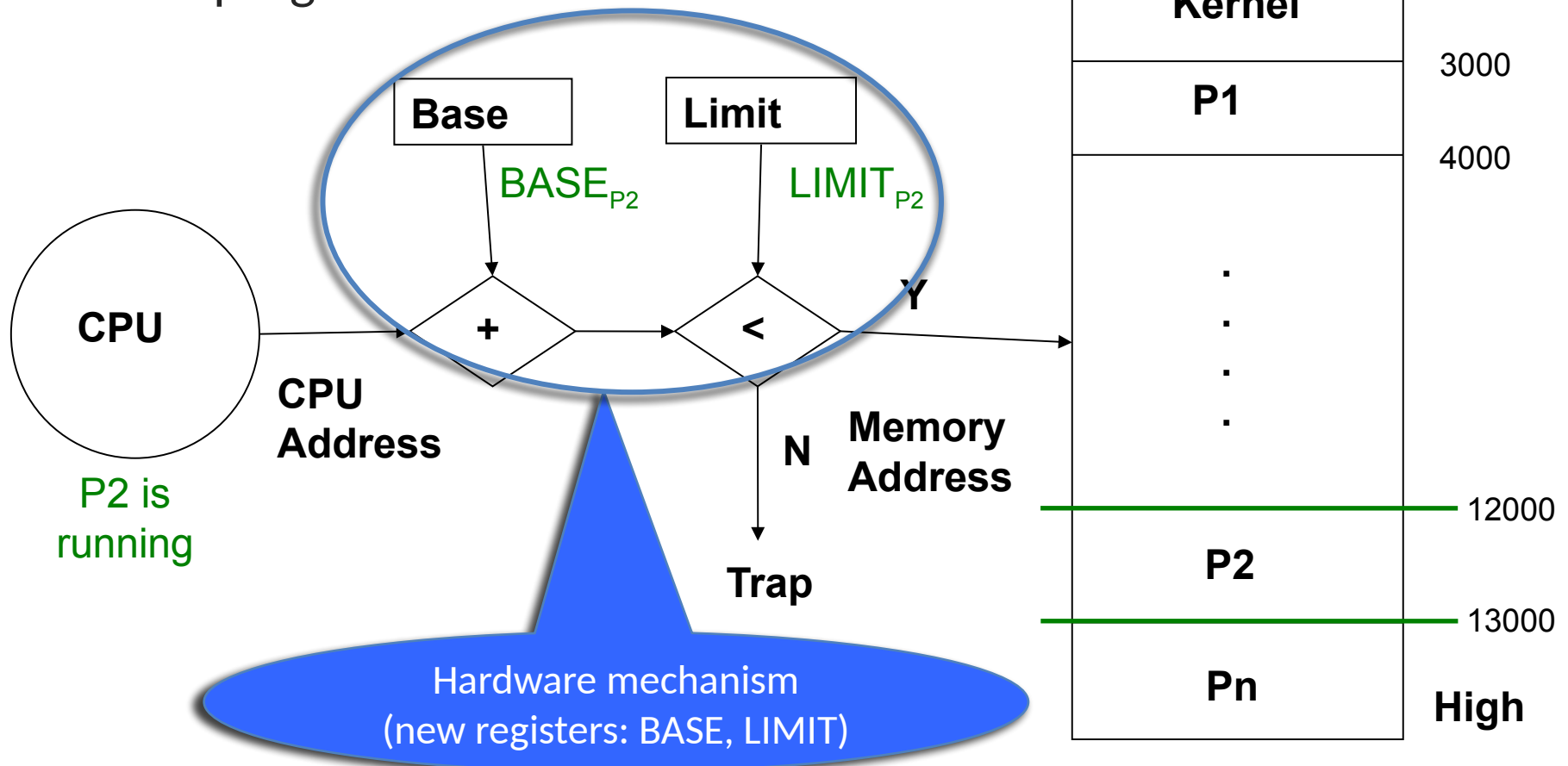
- Dynamically relocatable multiprogrammed OS



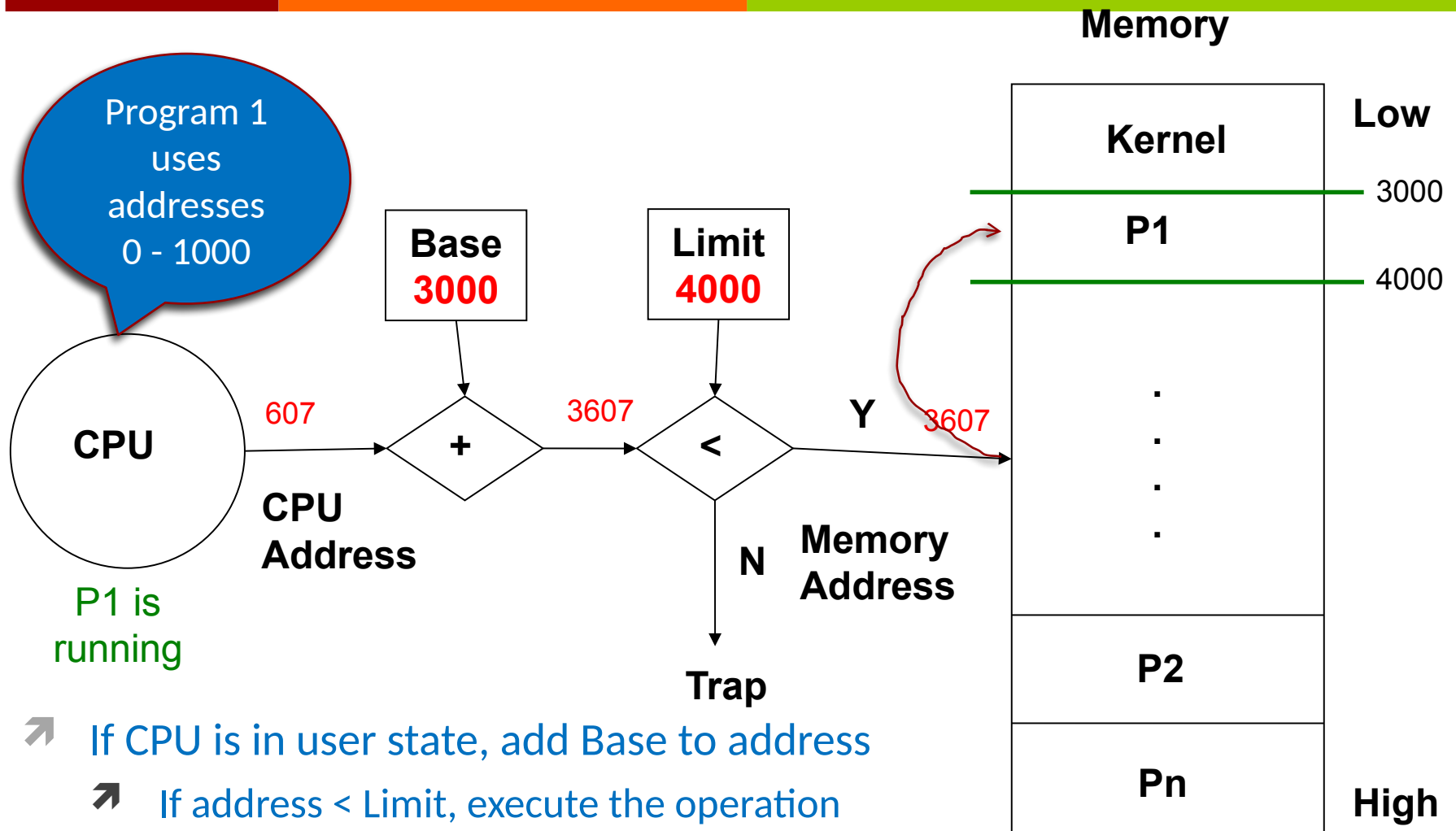


# But there's a hardware solution...

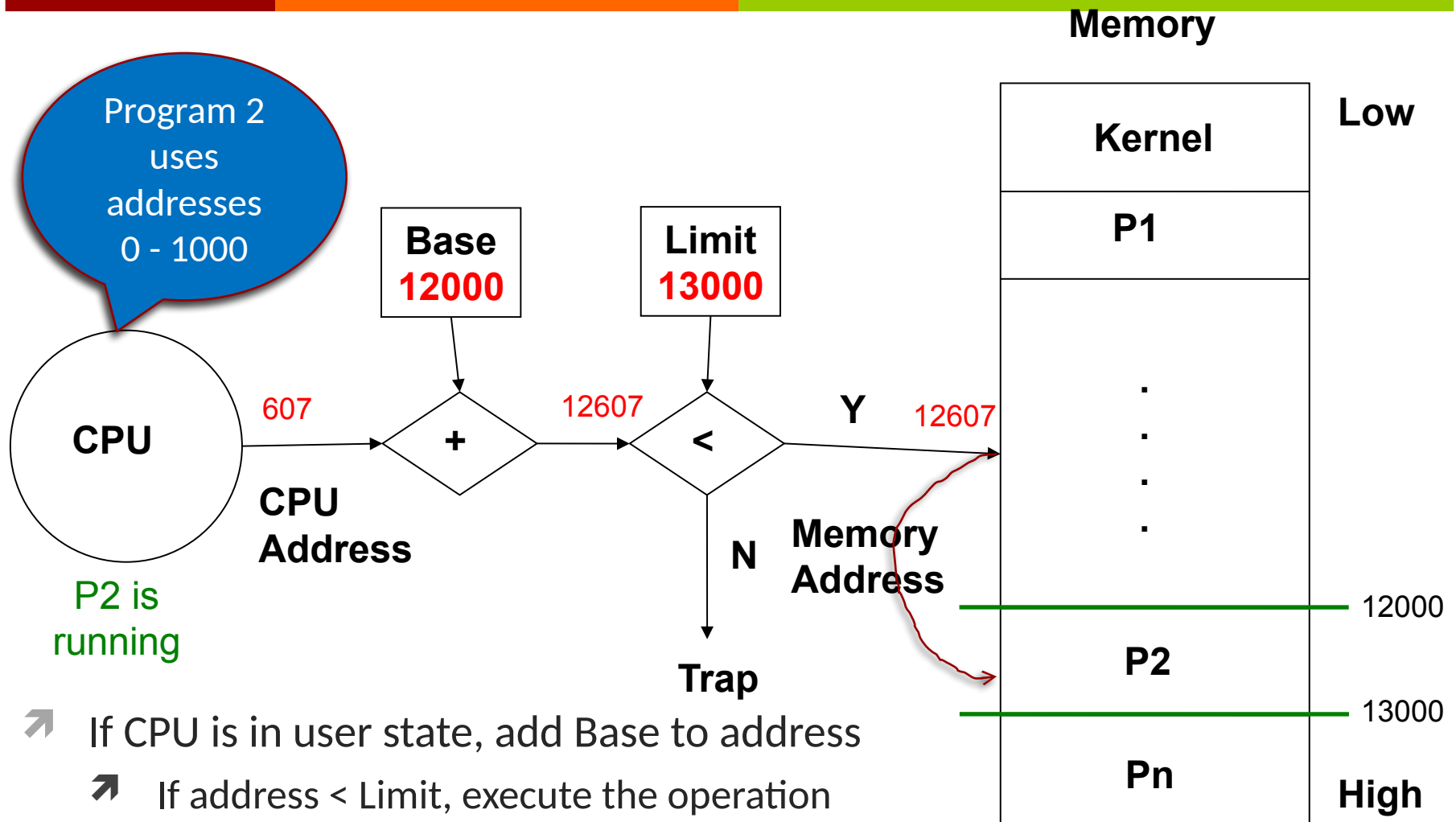
- Dynamically relocatable multiprogrammed OS



# Base & limit register example



# Base & limit register example



# PCB

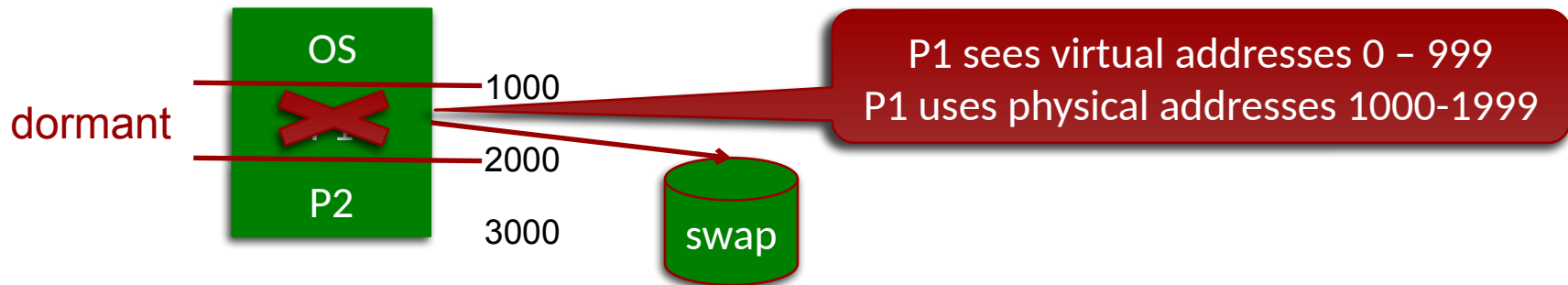
```
enum state_type {new, ready, running,
                 waiting, halted};
typedef struct control_block_type {
    enum state_type state;
    address PC;
    int reg_file[NUMREGS];
    struct control_block *next_pcb;
    int priority;
    address memory_footprint; ???
    ...
    ...
} control_block;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

# 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 BASE;
    address LIMIT;
    ...
    ...
} control_block;
```

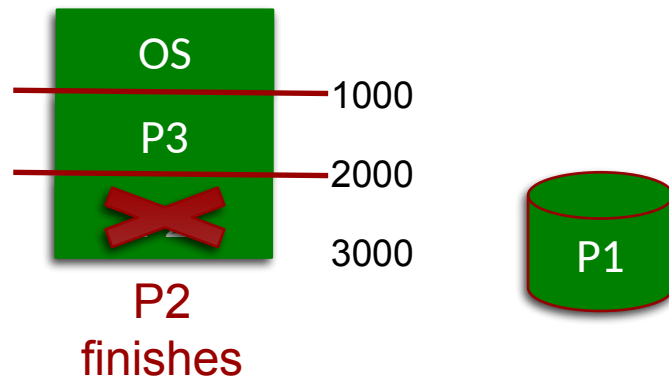
# The previous example with BASE + LIMIT

- Now all of P1 and P2's addresses appear to start at zero
- This is our first instance of a **virtual address** where the process sees memory addresses *different* from the physical addresses we've been working with so far!



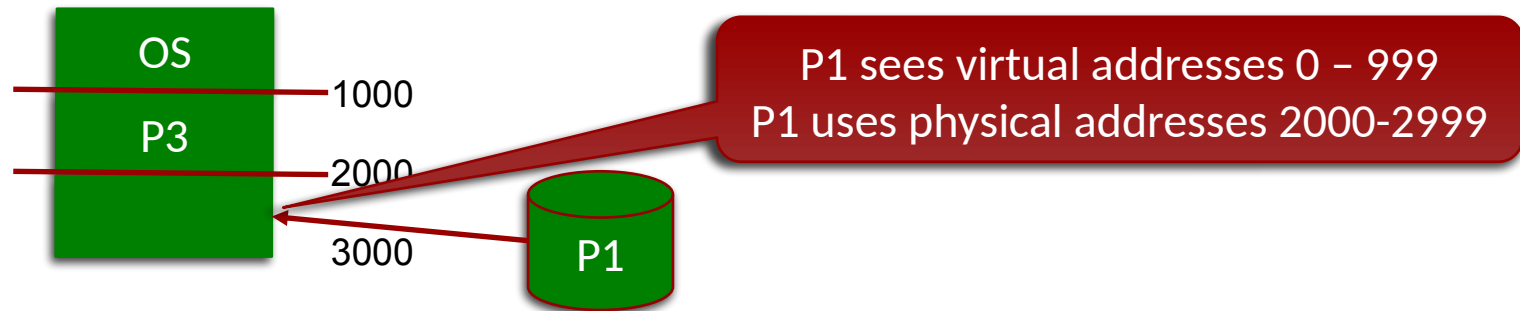
# The previous example with BASE + LIMIT

- Now all of P1 and P2's addresses appear to start at zero
- This is our first instance of a **virtual address** where the process sees memory addresses *different* from the physical addresses we've been working with so far!



# The previous example with BASE + LIMIT

- Now all of P1 and P2's addresses appear to start at zero
- This is our first instance of a **virtual address** where the process sees memory addresses *different* from the physical addresses we've been working with so far!




- ✓ We need to bring P1 back into memory
- ✓ Where?
- ✓ We have an empty spot...
- ✓ Will this work?
- ✓ It will if we set BASE & LIMIT to 2000 & 3000



# Question

To support dynamic relocation on the LC-2200, we would need...

- A. Fence register
-  B. Base + Limit registers
- C. Bounds registers
- D. LC-2200 works just fine as it is

Today's number is 24,987

# 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

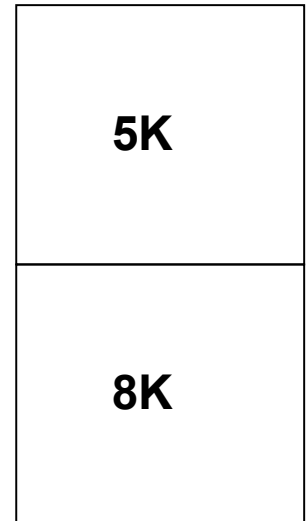
- Fixed size partition
- Variable size partition
- Both use the hardware base + limit registers

# Fixed size partitions

memory

## OS memory manager allocation table

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



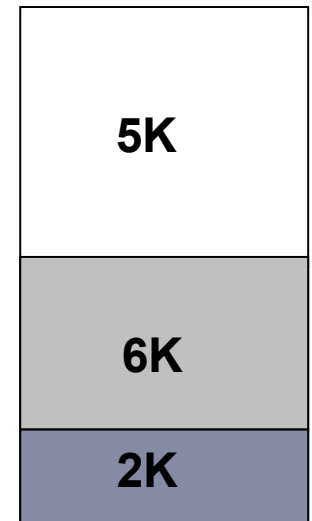
```
Struct AT_entry {  
    int occupied;  
    int size;  
    int pid;  
};
```

# P1 needs 6K of memory

**Allocation table**

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

**Memory**



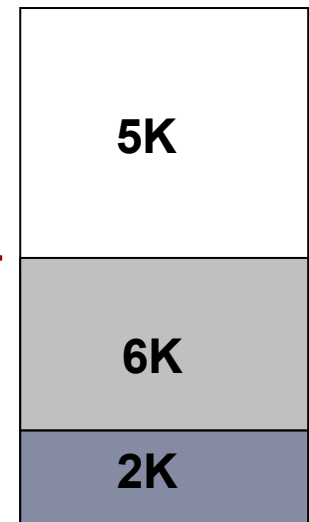
# P1 needs 6K of memory

Allocation table

Memory

Table size is fixed = number of partitions

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



Needs only  
6K

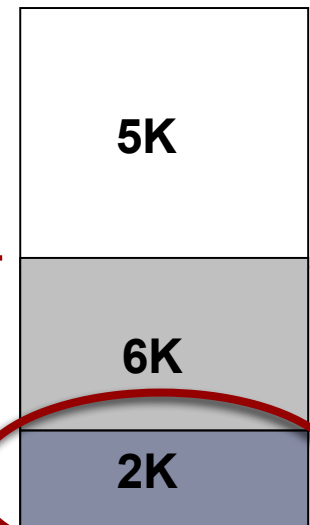
# Another process needs 6K memory

**Internal fragmentation** = size of partition – actual used

**Allocation table**

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

**Memory**



Needs only  
6K

Wasted  
(internal  
fragmentation)

# Another process needs 6K memory

Do we have it?

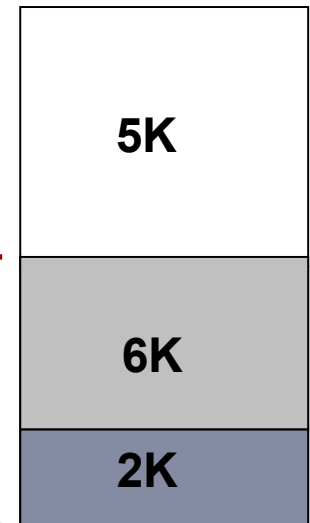
Memory manager has only a 5K partition...

Not possible

Allocation table

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

Memory





# External fragmentation

Consider P3 that needs 4K of memory

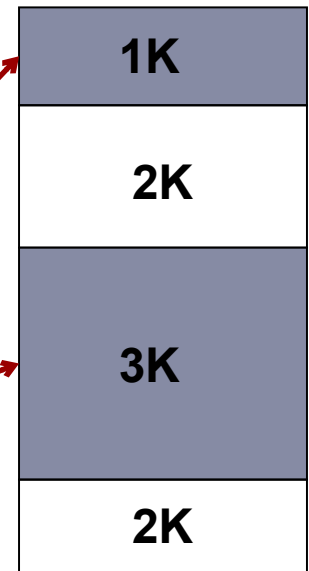
Is it possible to allocate?

Memory is available, but not contiguous

Allocation table

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

Memory



External fragmentation

=  $\sum$  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
  - $\sum$  All non-contiguous free partitions
  - If there is only one free partition, we say the external fragmentation is zero

# Fixed size partition memory management

- Virtues
  - Simplicity
- Bad news
  - Fragmentation
    - Internal
    - External

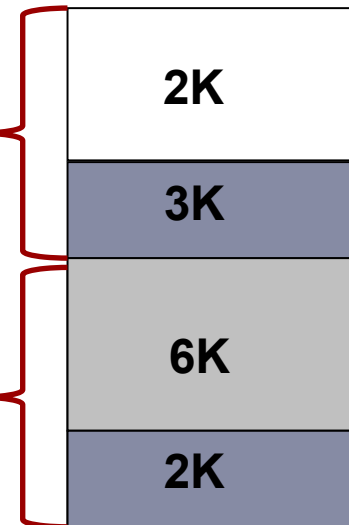
# Question

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

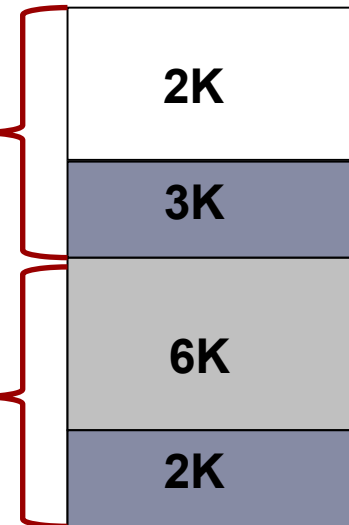
# Question

Total external fragmentation is...

**Allocation table**

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

**Memory**



A. 0K

C. 5K

B. 2K

D. 8K

# Overcoming internal fragmentation

- Allocate exactly what is needed
- Variable size partitions

# Variable size partitions

memory

Memory manager allocation table

Start address	Size	Process
0	13K	FREE

13K

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

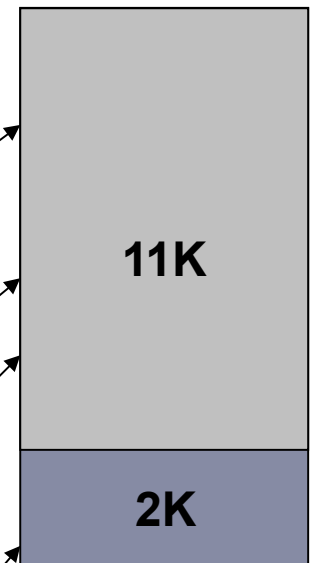
# Partition table a little while later

Memory

Allocation table

Grows and shrinks as partitions get created and released

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



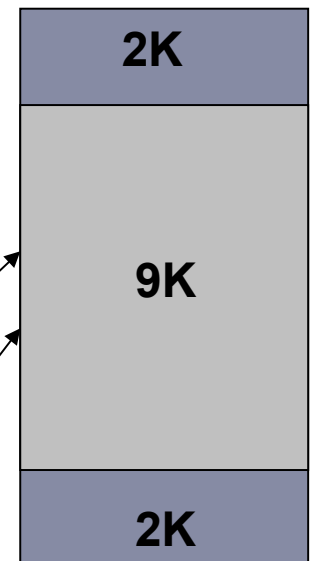


# P1 exits

Memory

Allocation table

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



New request:  
P4 needs 4K  
Possible?

External fragmentation

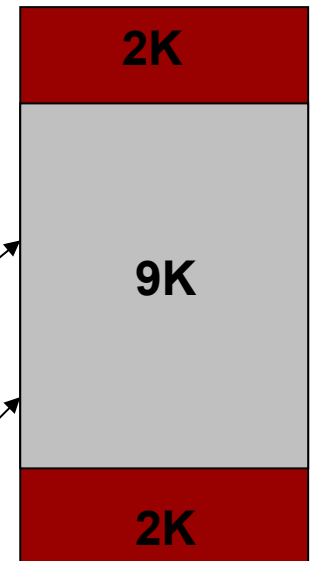


# P2 exits

Allocation table

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

Memory

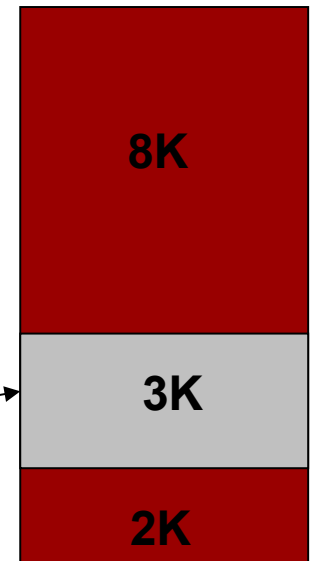


# Coalescing two free areas

**Allocation table**

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

**Memory**



# Reducing external fragmentation

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

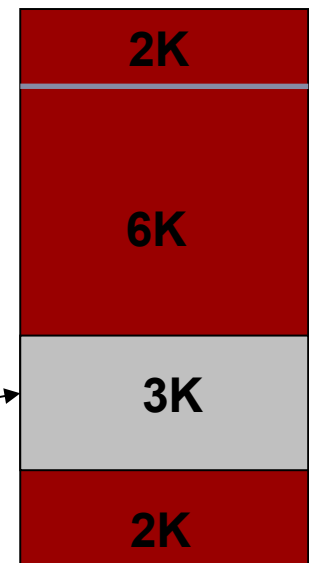
# Compaction

➤ Request for 9K

Allocation table

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

Memory



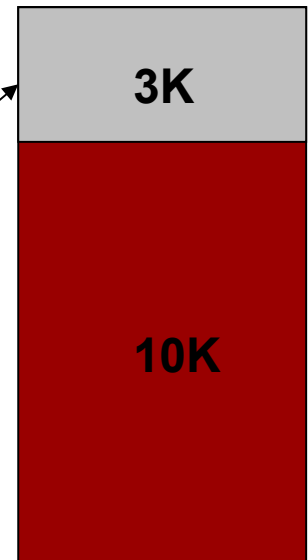
# Compaction

- 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**



# Question

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

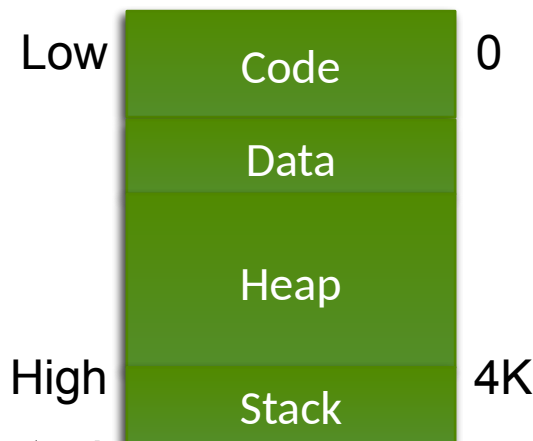


- Can limit full usage of memory resources
- Compaction is too expensive



# How might we solve the external fragmentation problem?

➤ Our memory footprint looks like this:

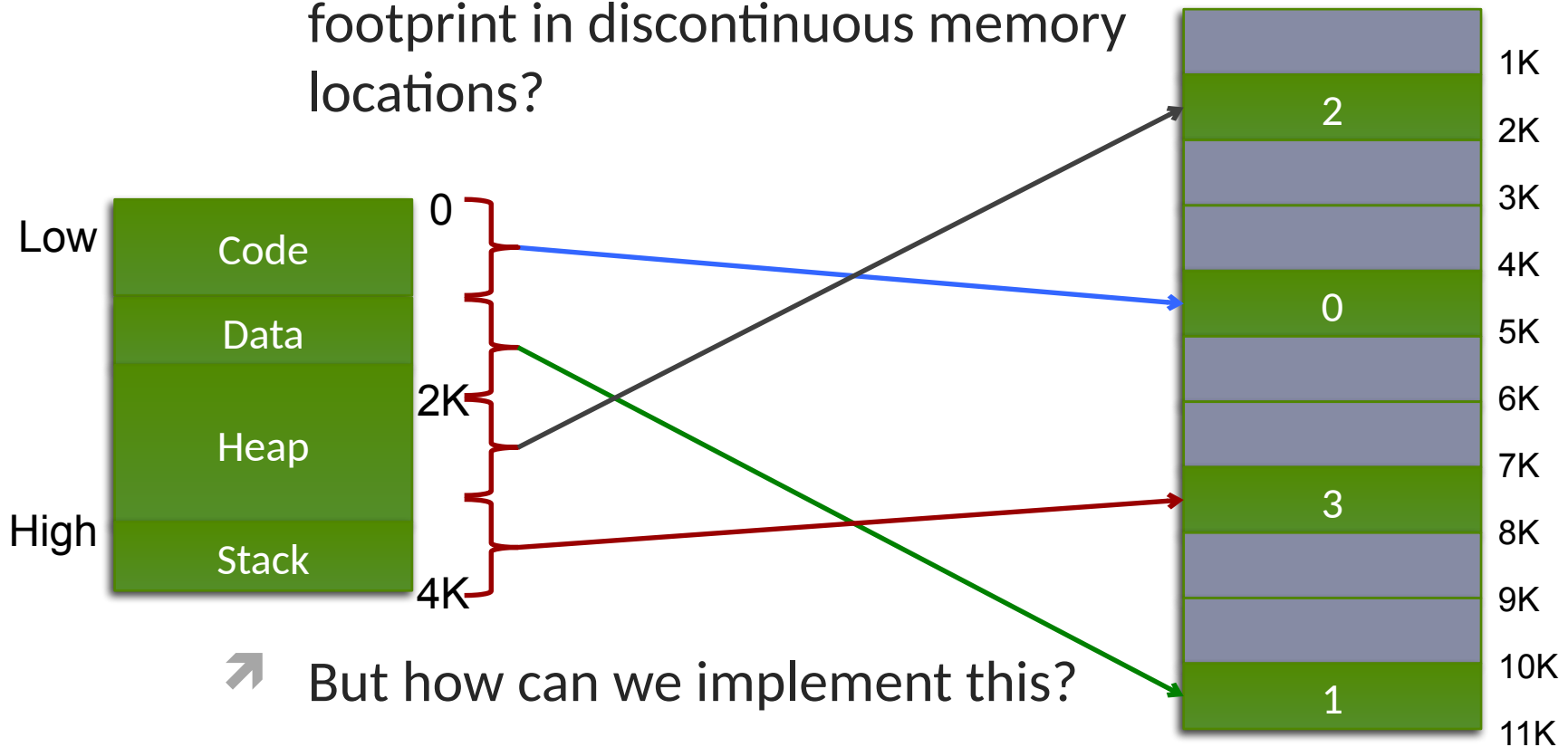


➤ What's the main limiting assumption?

➤ That the process address space is contiguous!

# How might we solve the external fragmentation problem?

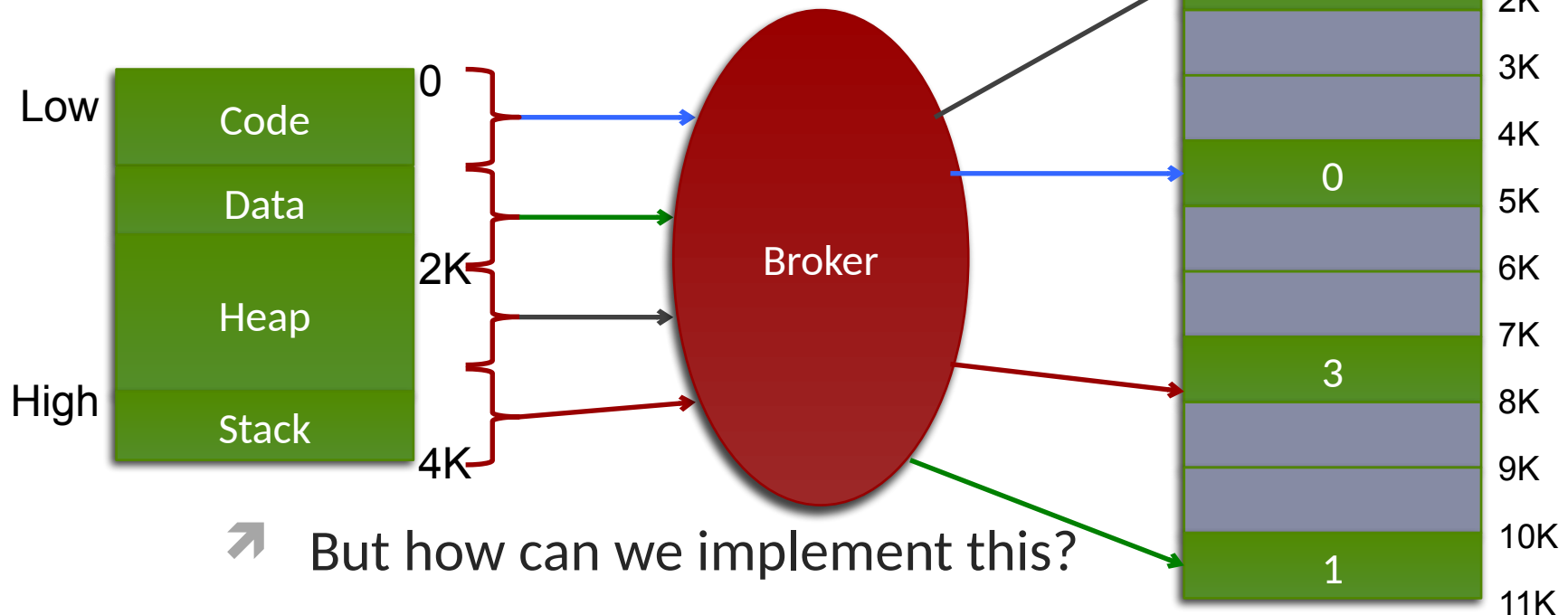
➤ What if we could store our memory footprint in discontinuous memory locations?



➤ But how can we implement this?

Could the broker help?

➤ What if we could store our memory footprint in discontinuous memory locations?

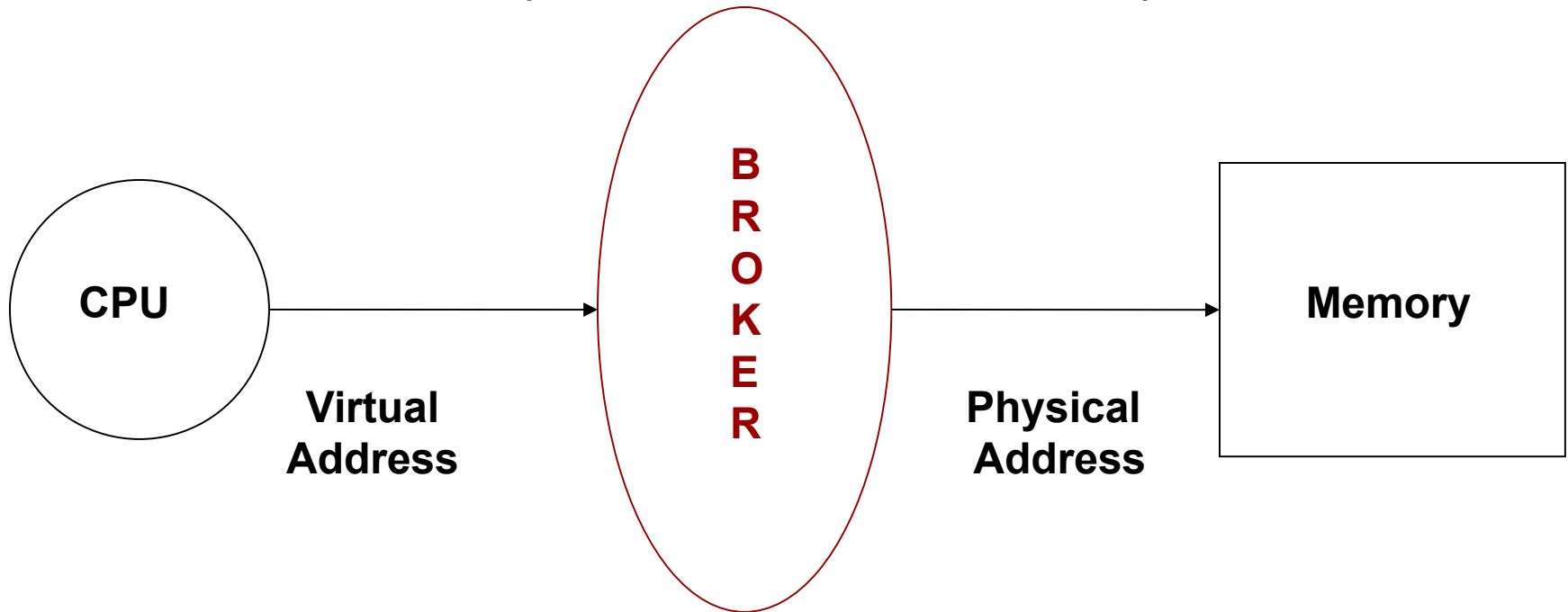


contiguous

discontiguous

# Broker for paging

- This broker maps
  - Virtual address (VA) from the CPU
  - to
  - Physical address (PA) to memory



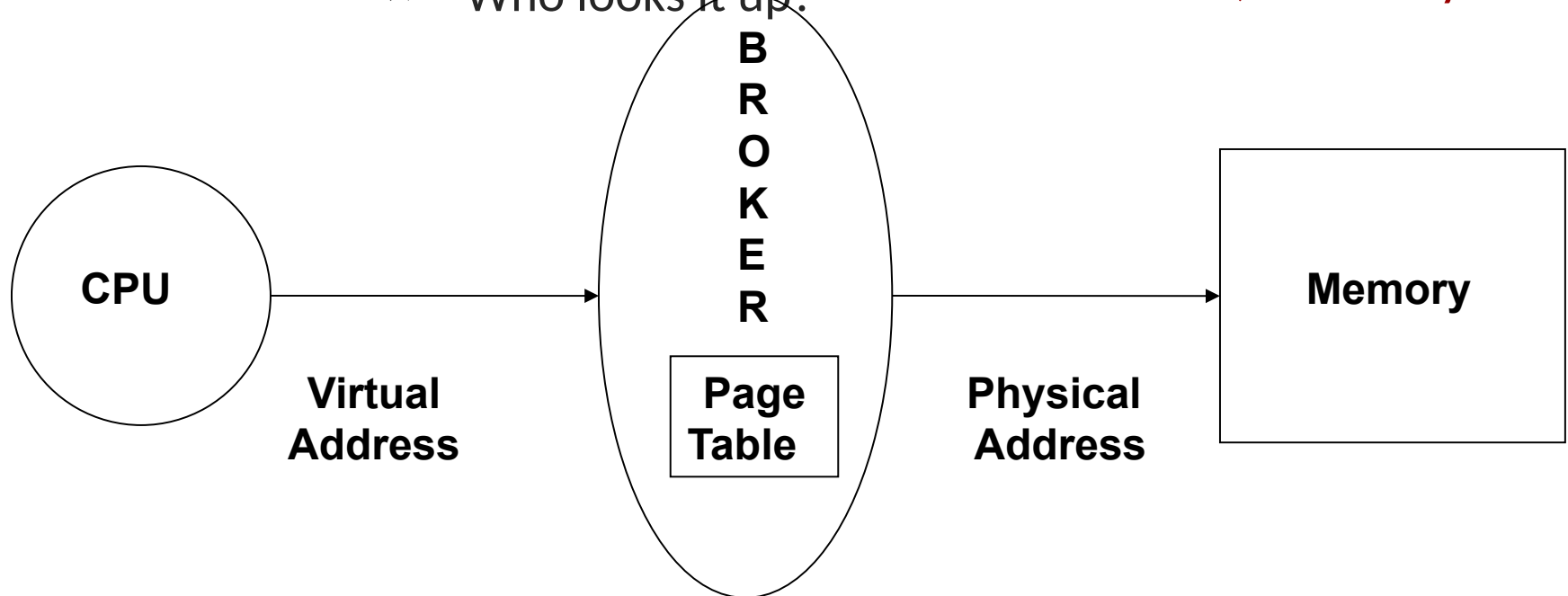
# Broker for paging

➤ How does Broker map VA to PA?

➤ Perhaps like a phone directory?

➤ Who sets it up? ⑨ The OS

➤ Who looks it up? ⑨ The hardware, on every access



# How big is this table?

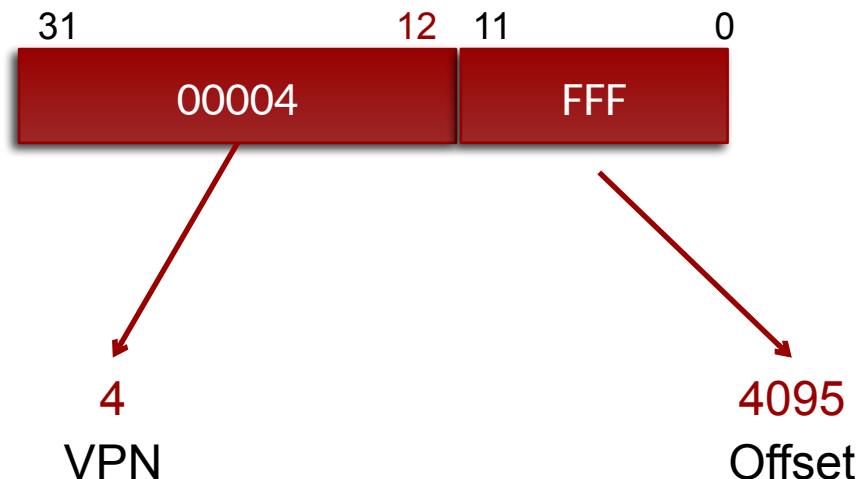
- At the lower size limit, we could map the whole program
  - There would be only one entry in the page table
  - That's the same as Base + Limit, no?
- At the upper size limit, we could map every word
  - The table would be the size of the address space times the address size in words
  - Not very practical
- So in between, we choose a **page size** to map
  - Bigger pages get us more **internal fragmentation** (average is  $\frac{1}{2}$  of the last page)
  - Smaller pages get us a bigger page table and take more CPU time to manage it

# Choosing a page size

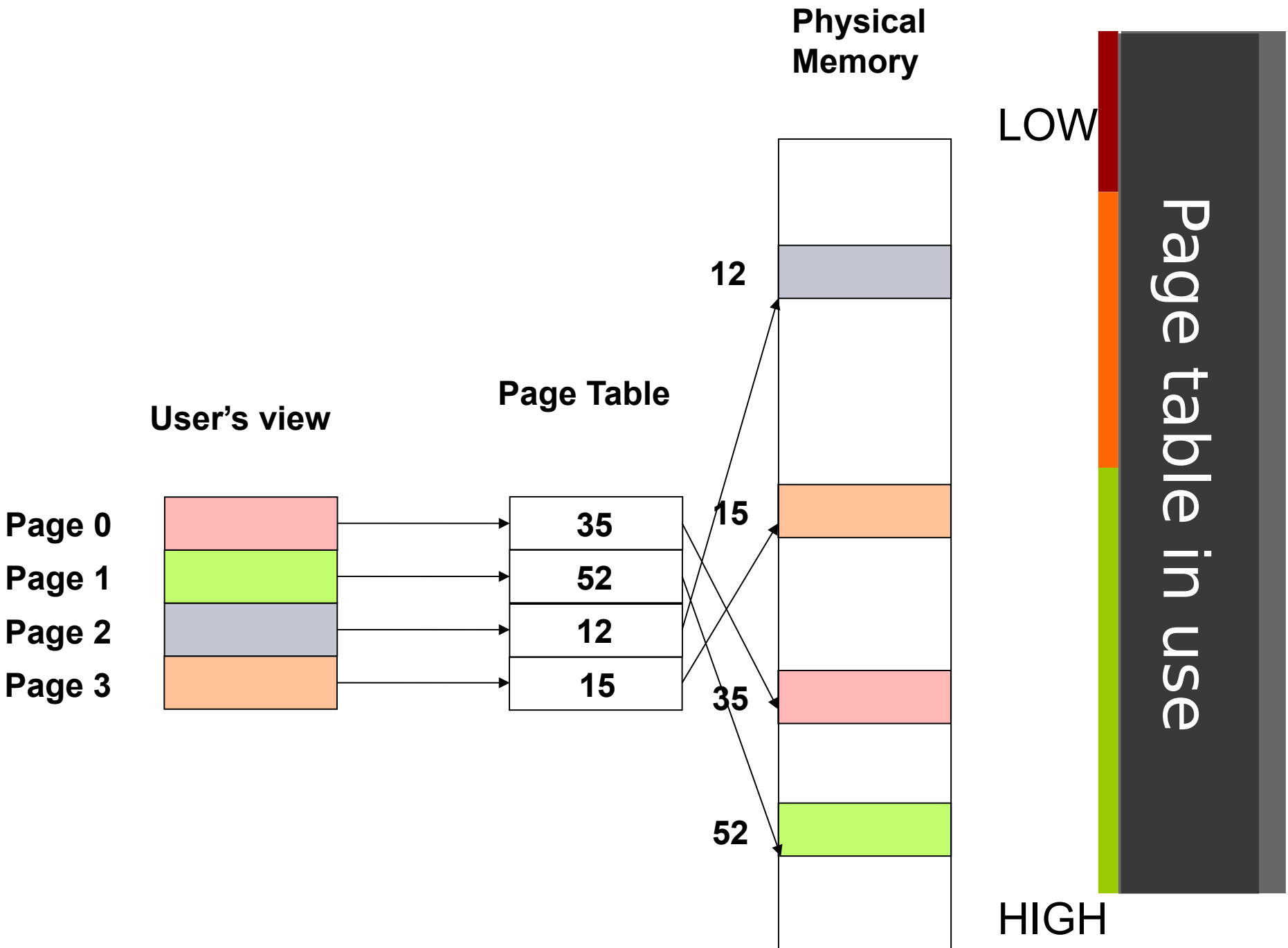
- When memory was expensive (and small)
  - Page sizes were 512 to 2048 bytes
- These days
  - 4 KB up to 1 GB
  - Often it's settable per process
- Page size is always a power of 2
  - The power of two allows us to split the VA into **virtual page number** (VPN) and **offset** within the page at a bit boundary
  - If the page size is  $2^n$ , the lower **n** bits are the **offset** and bit **n** and up are the **virtual page number**

# Splitting up a virtual address

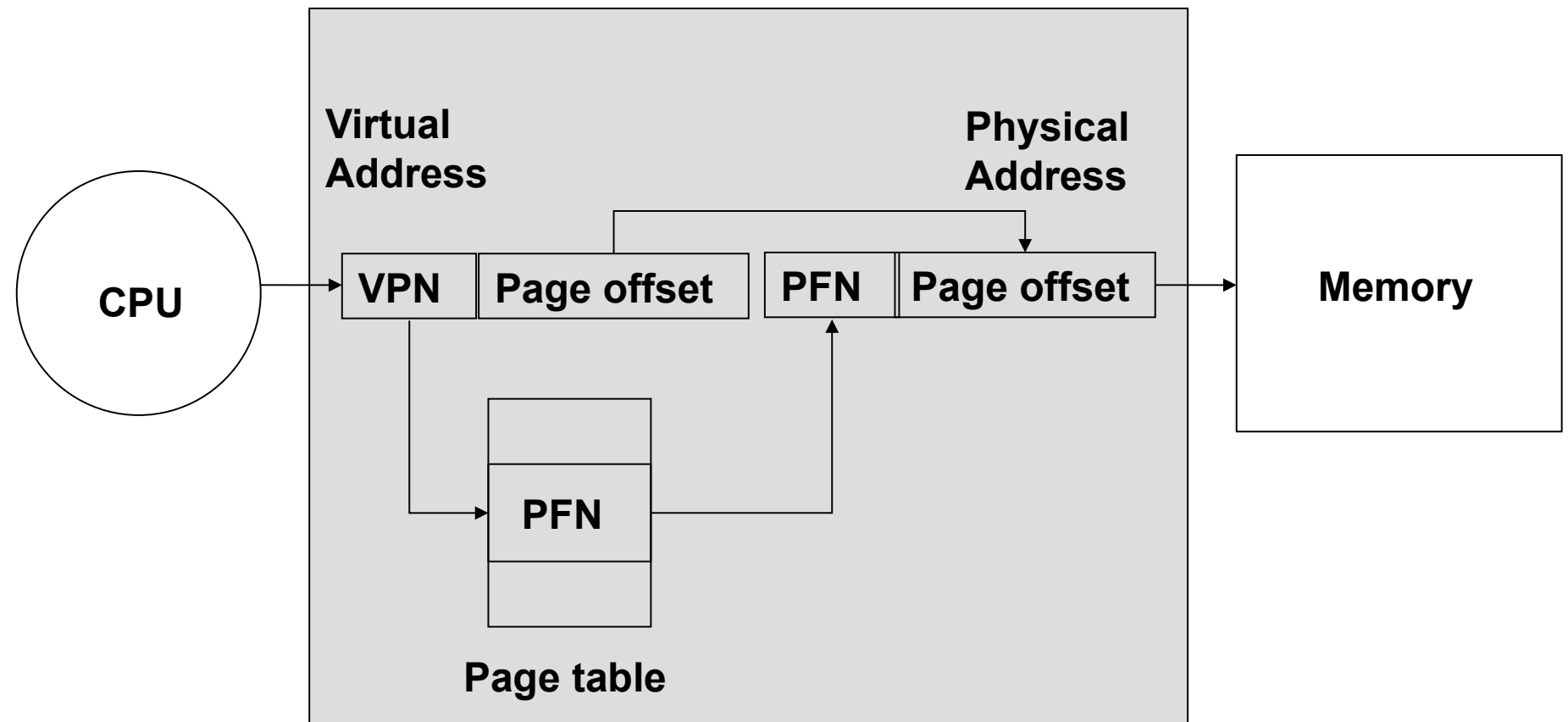
- Say we have a 4KB page size and a 32 bit virtual address space
  - 4KB is  $2^{12}$ , so the bottom 12 bits are offset and the top 20 bits are VPN
- For example, for virtual address 0x00004FFF







# Address translation

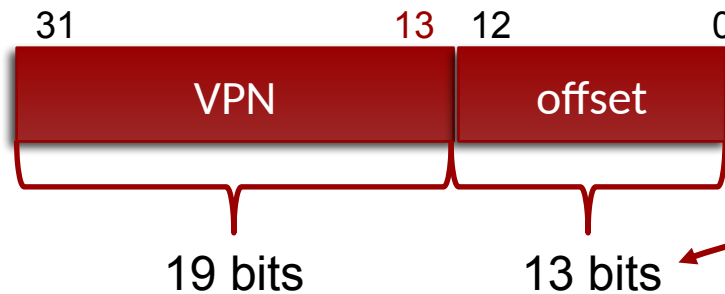


# Examples

➤ Consider a memory system with a 32-bit virtual address. Let us assume the pagesize is 8K Bytes.

➤ How big is the page table?

➤  $8k = 2^{13}$

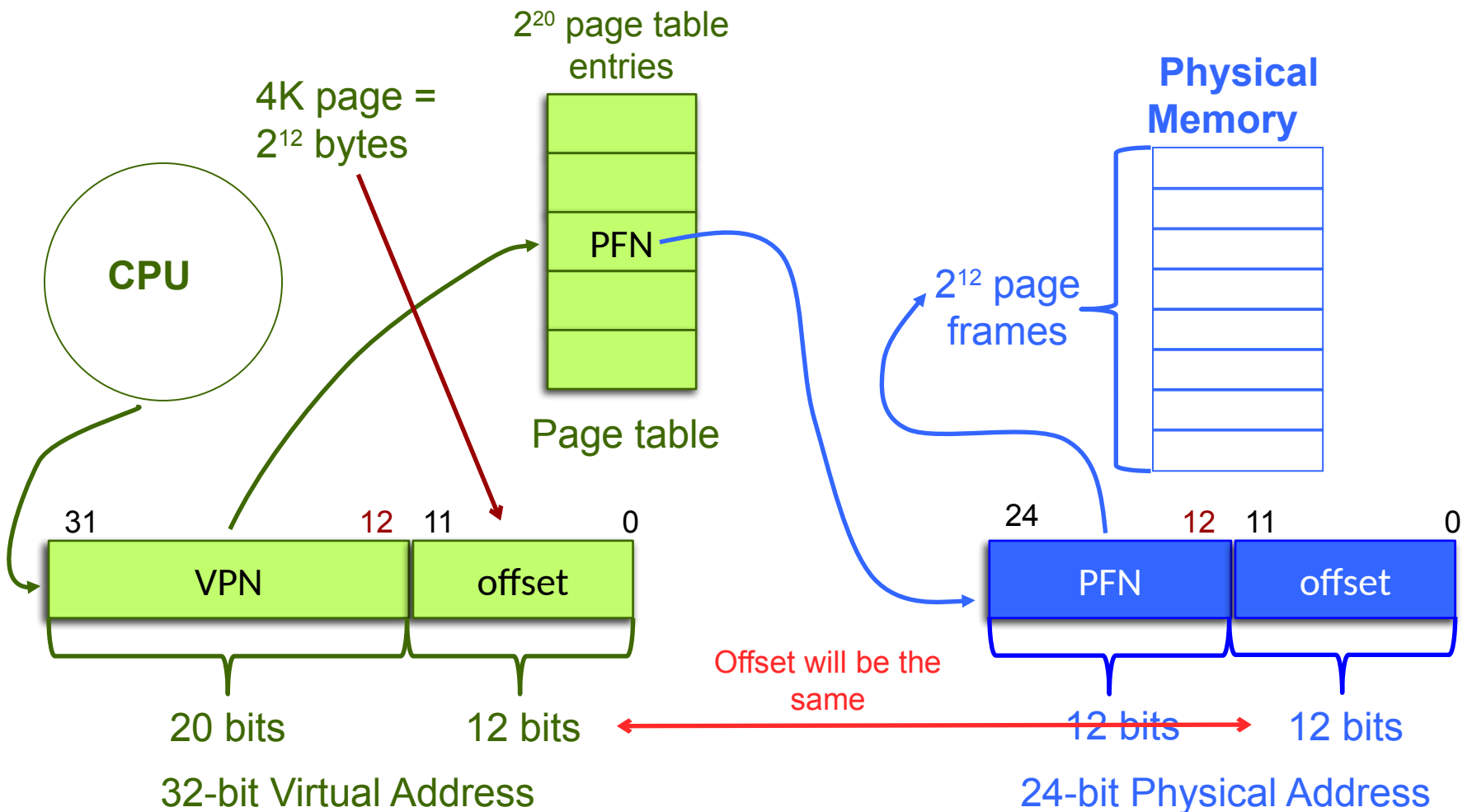


➤ VPN is 19 bits, so page table is  $2^{19}$  or 524,288 entries

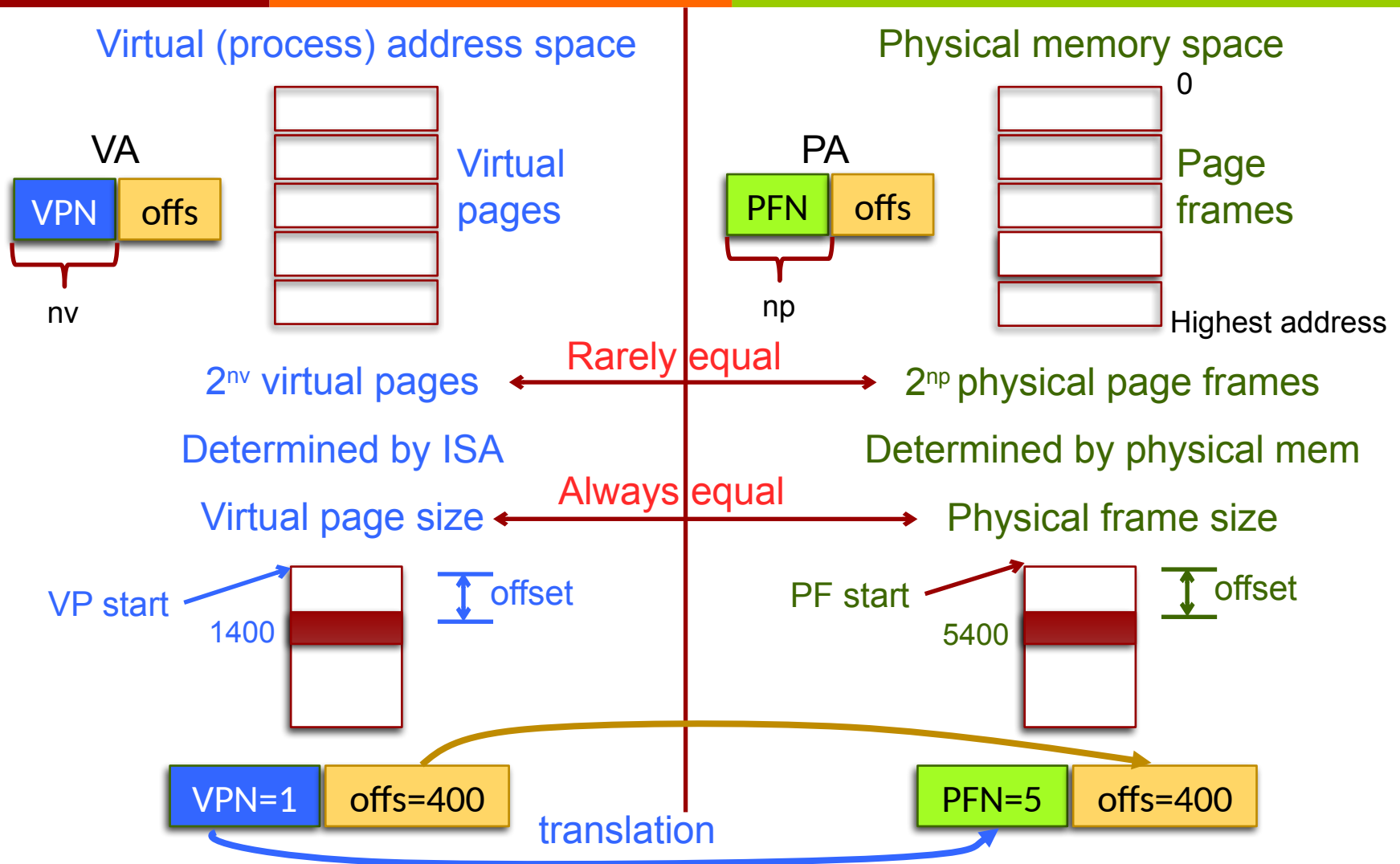
# Examples

- 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 can be in memory?
    - PFN bits =  $24 - 12 = 12$ ;  $2^{12}$  frames
  - How big is the page table?
    - VPN bits =  $32 - 12 = 20$ ;  $2^{20}$  entries

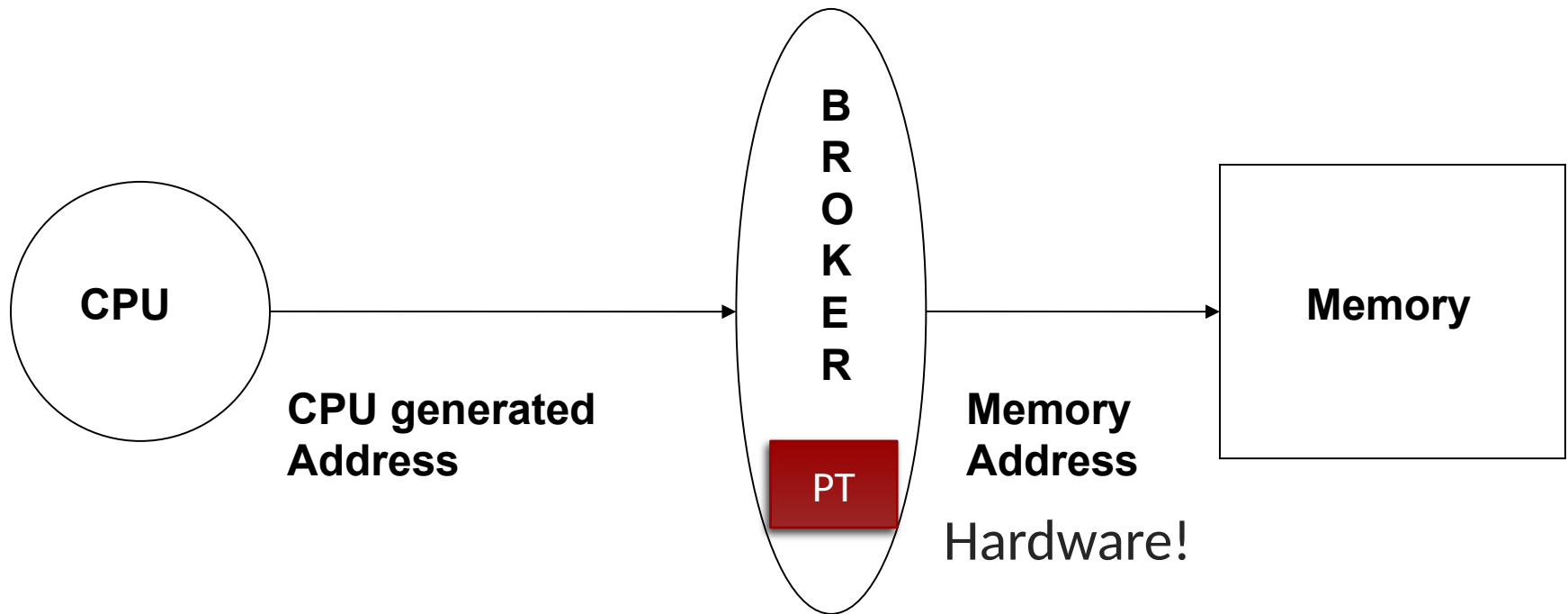
# Example



# Important facts about paging



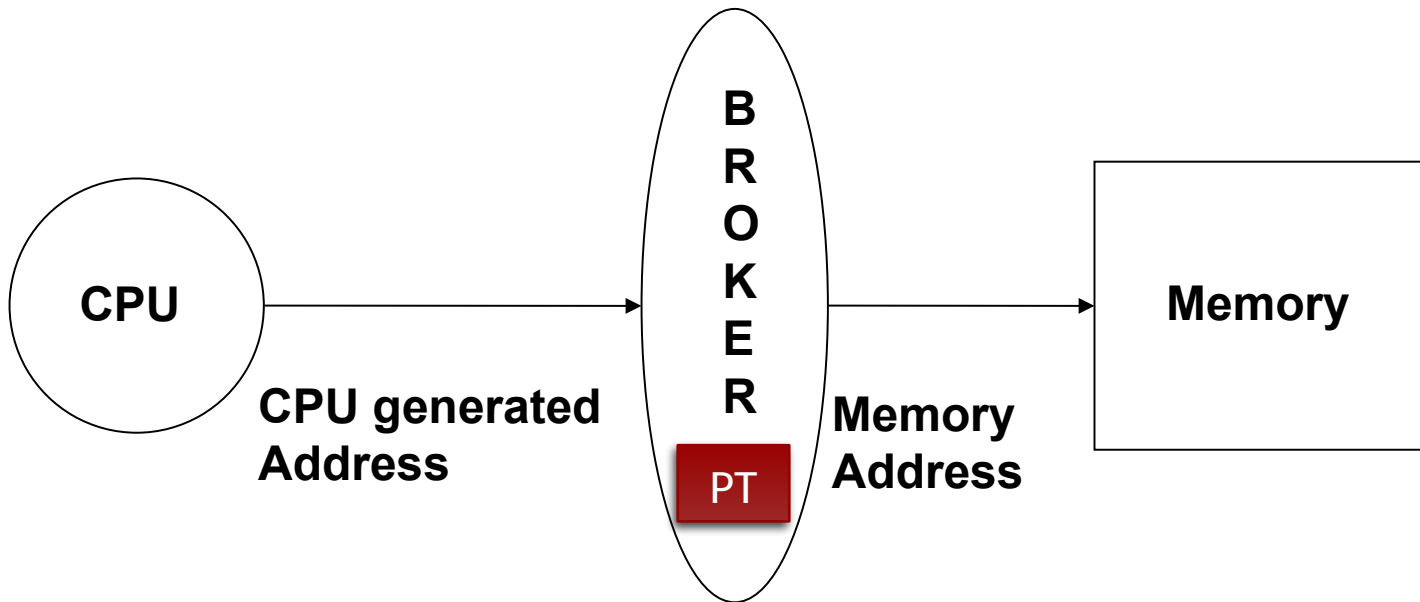
# So exactly where is the page table?



But it's not a special device

**It's in physical memory!**

# How many page tables are there?



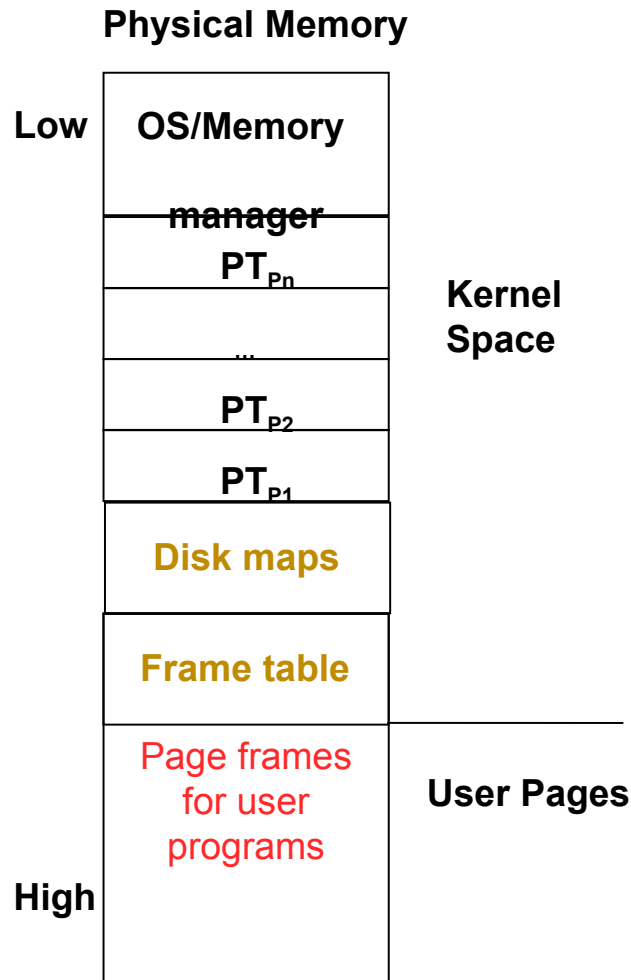
Process  
1, 2, 3, ..., n in  
memory

We'll need n  
page tables!

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



# 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 in the broker to look up the PFN from the page table for each memory reference

# What hardware assist does LC-2200 need?

PTBR

**Page Table**



PFN
PFN
PFN
PFN
PFN
PFN

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

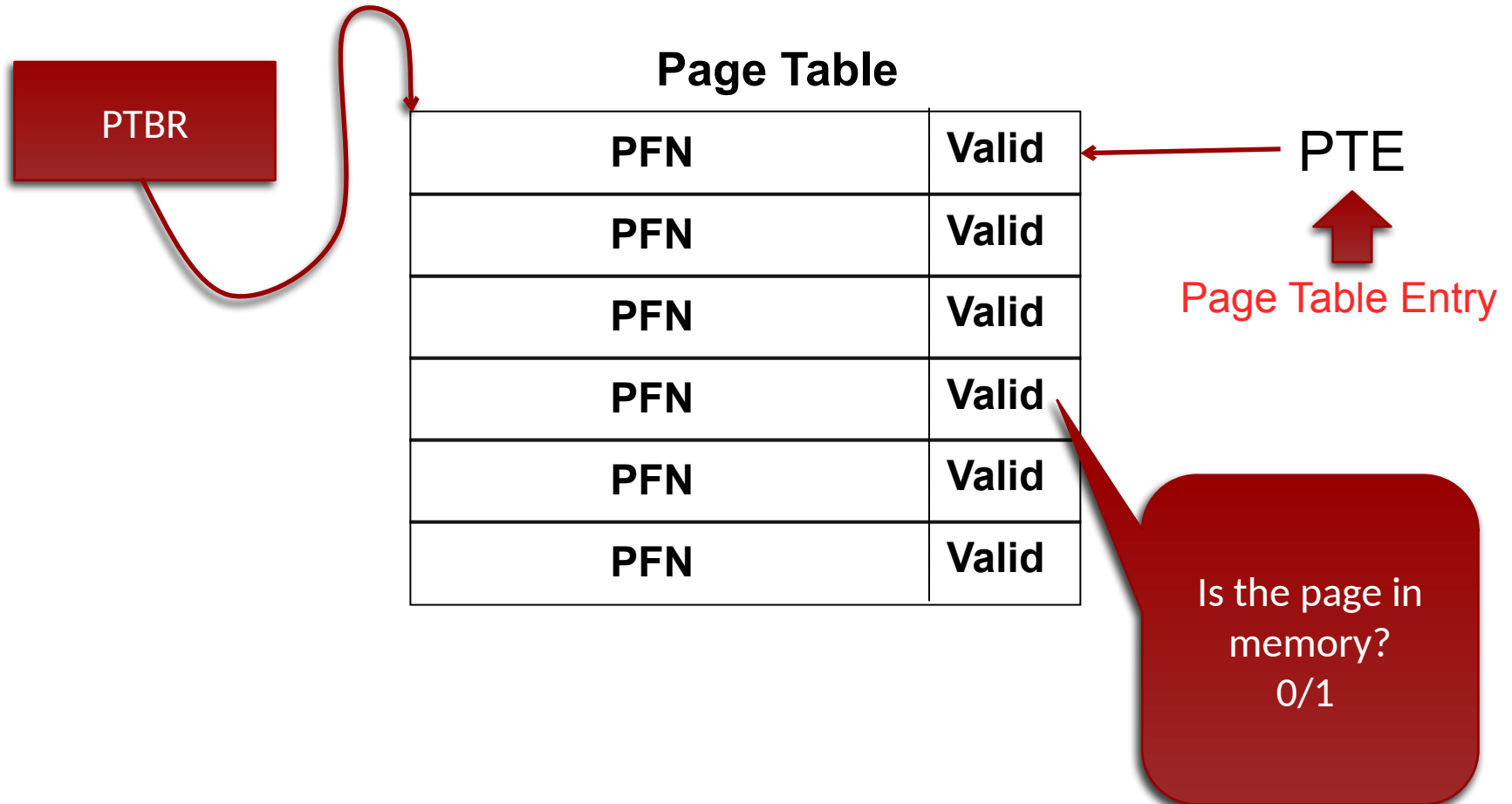
# PCB

```
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

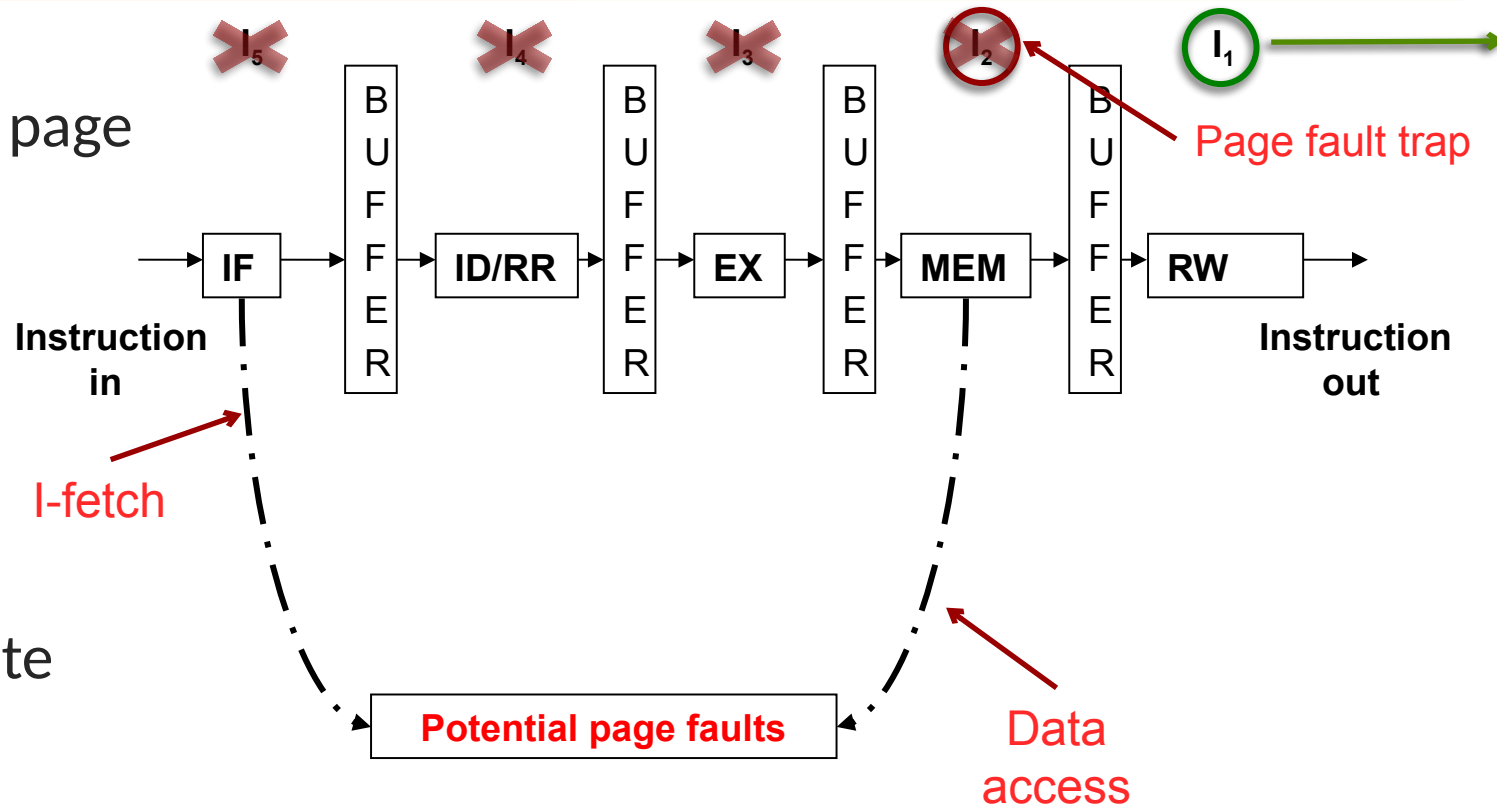
- Allocate all at once at load time? ⌚ Not good: slow
- Allocate on demand? ⌚ Better utilization

# Demand paging



# Ramification of demand paging

Where can a page fault hit us?



Let  $I_1$  complete  
Squash  $I_2-I_5$

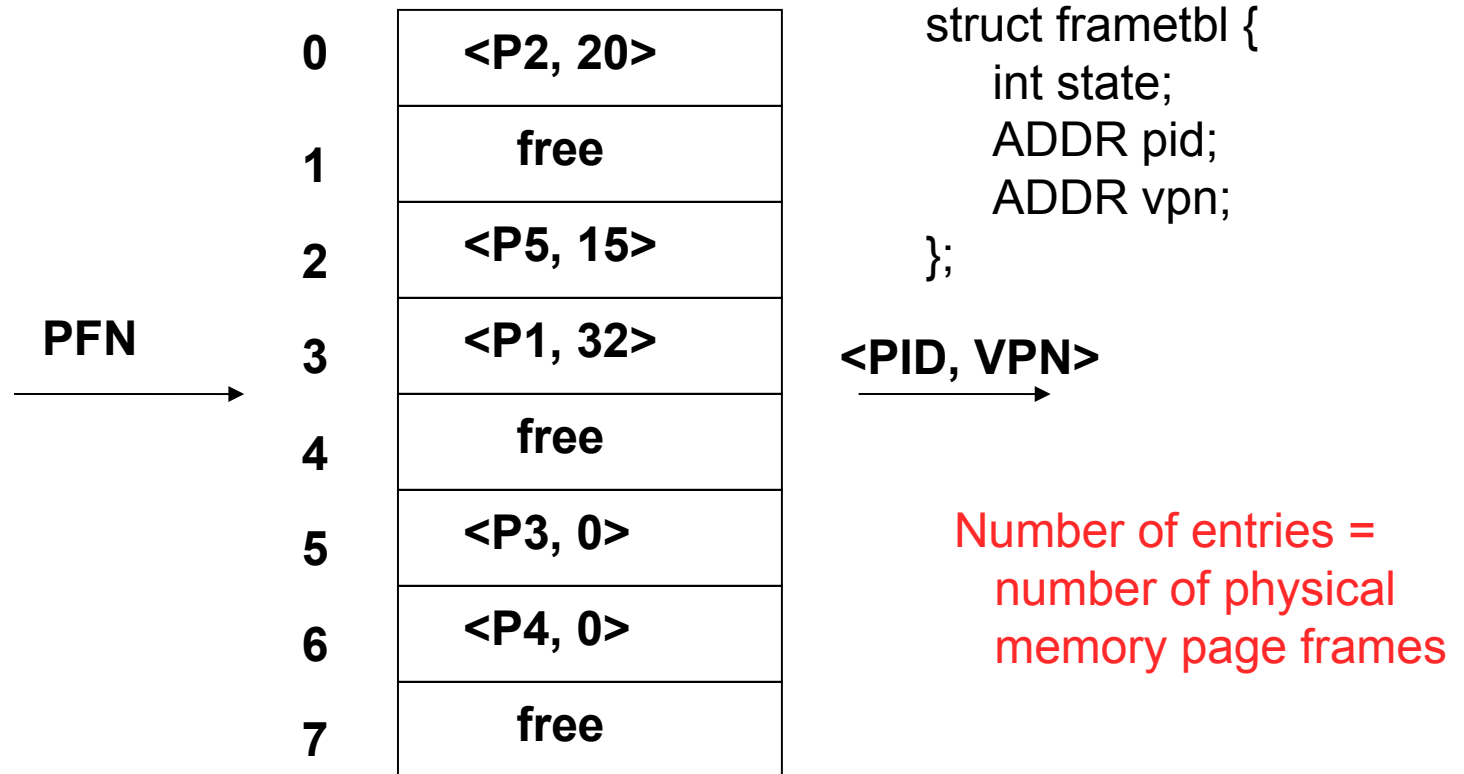
Trap to page fault handler,  
saving PC of  $I_2$  for restart  
after page fault is serviced

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

# Page fault handler

- Find a free page frame
- Load the faulting virtual page from disk into the page frame
- Give up the CPU while waiting for the paging I/O to complete
- Update the page table entry for the faulting page
- Place the PCB of the process back in the ready\_q of the scheduler
- Call the scheduler

# Frame Table



Page table maps forward; Frame table maps in reverse

We need this for evicting pages

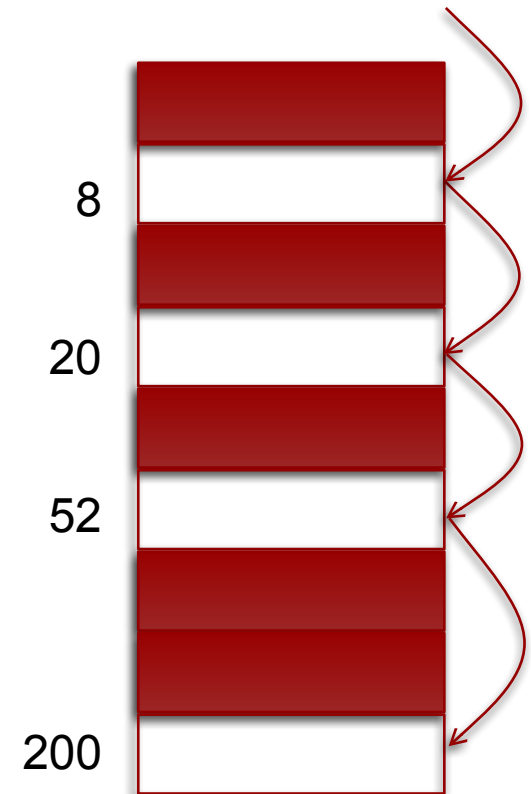


# Free list for page fault handler

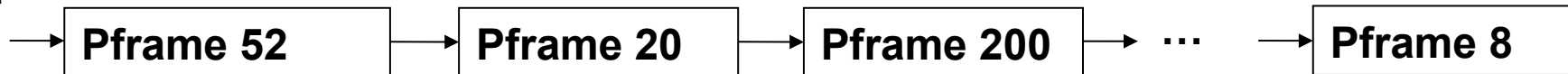
## Use a linked list

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

## Or link free frame table entries together



freelist

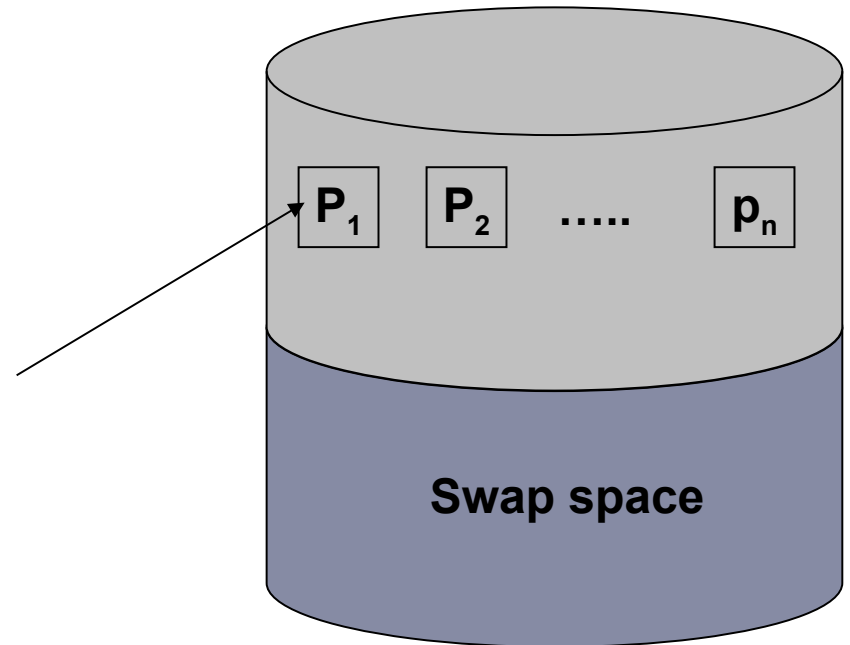


# Disk Map

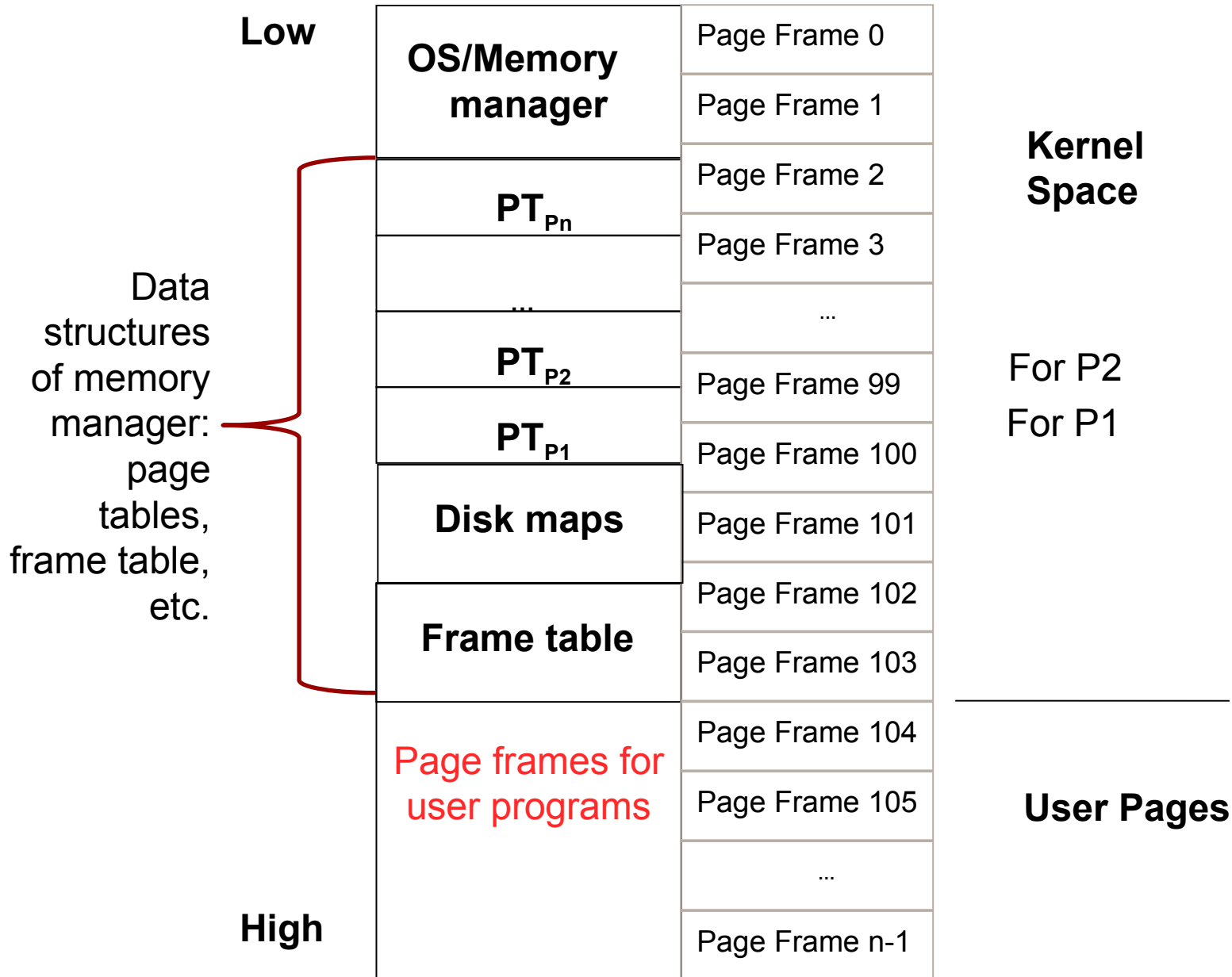
Disk map for P1

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

VPN →

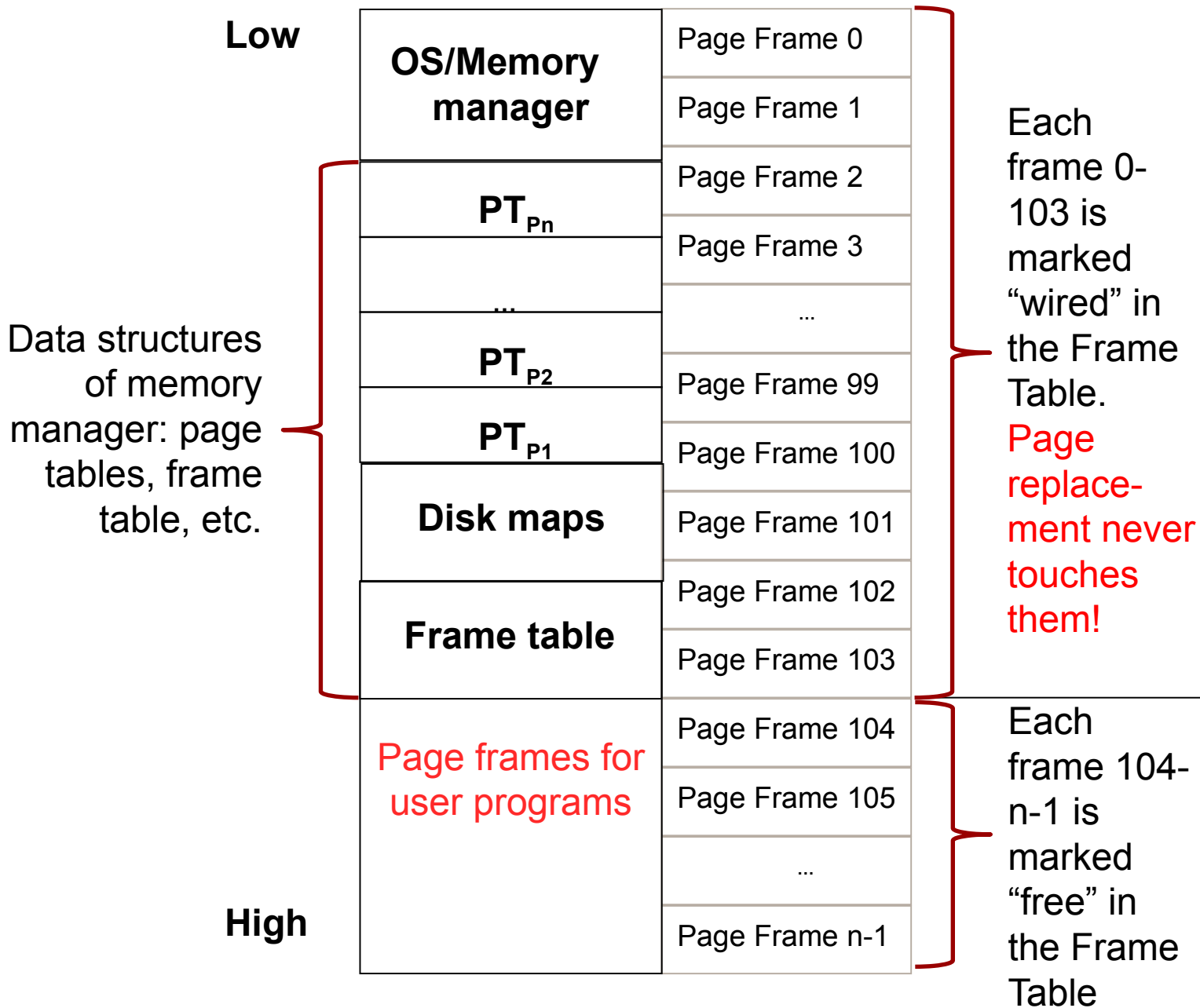


# Physical Memory



Physical memory layout

# Physical Memory



Physical memory layout

# Virtual memory manager data structures

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

## FRAME TABLE

FN	<PID, VPN>
0	<P3, 37>
1	<P1, 33>

...

## P4

PN	V	PFN
0	V	22
1	V	47

...

32	V	54
33	V	52
34	I	0

Currently  
PFN=52 in  
use by P4

50	<P1, 1>
51	<P1, 39>
52	<P4, 33>
53	<P9, 68>
54	<P4, 32>

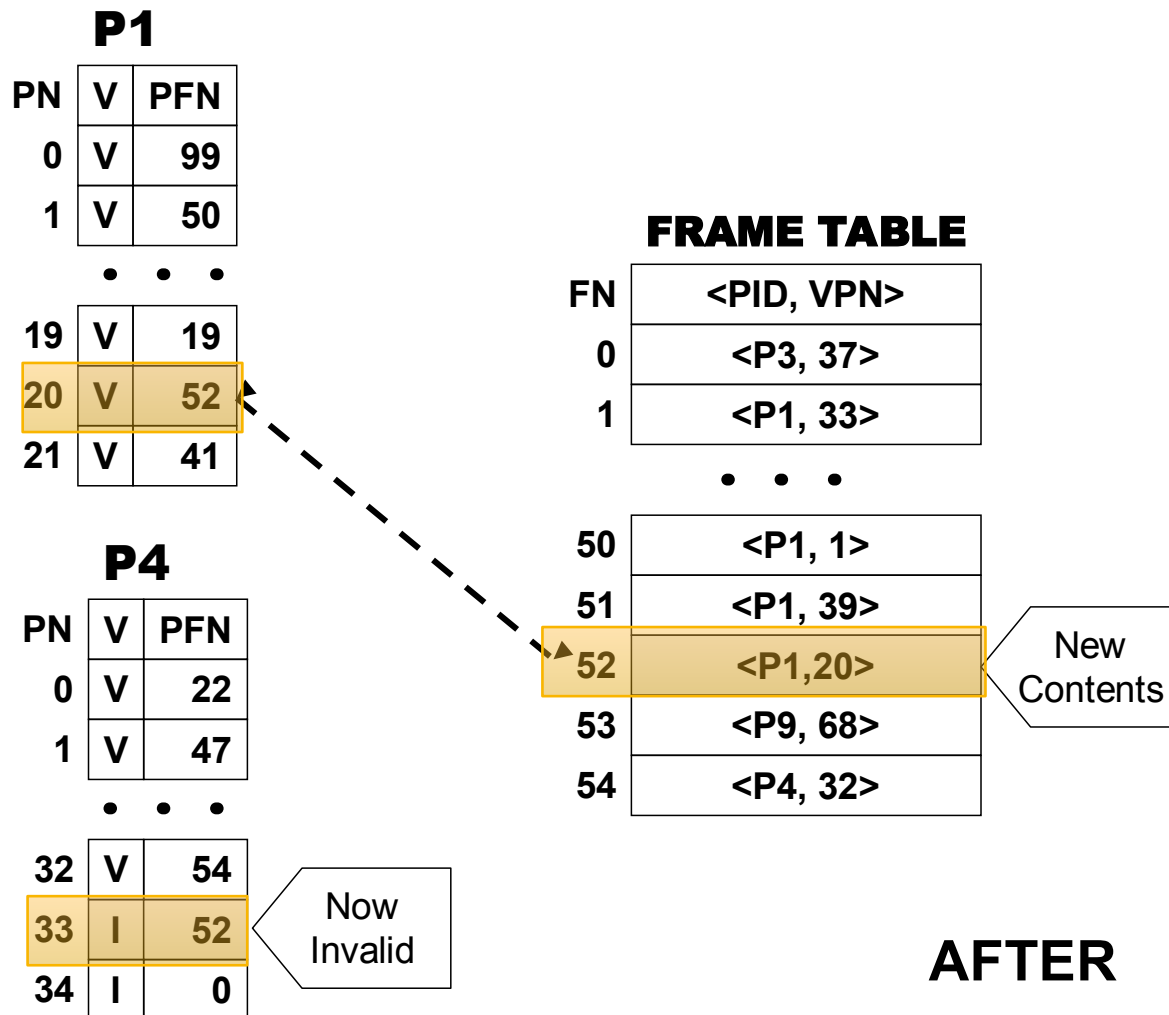
Victim

Page replacement  
algorithm chooses  
PFN=52 as victim

**BEFORE**




# After



# Question

With paged memory management there can be

- A. External fragmentation
-  B. Internal fragmentation
- C. No fragmentation
- D. Both internal and external fragmentation

Today's number is 36,050

## User level

Program 1

Program 2

.....

Program n

## Kernel

ready\_q

PCB<sub>1</sub>

PCB<sub>2</sub>

...

CPU scheduler

FT

PT<sub>1</sub>

DM<sub>1</sub>

PT<sub>2</sub>

DM<sub>2</sub>

⋮

⋮

freelis  
t

Pframe

Pframe

...

Memory Manager

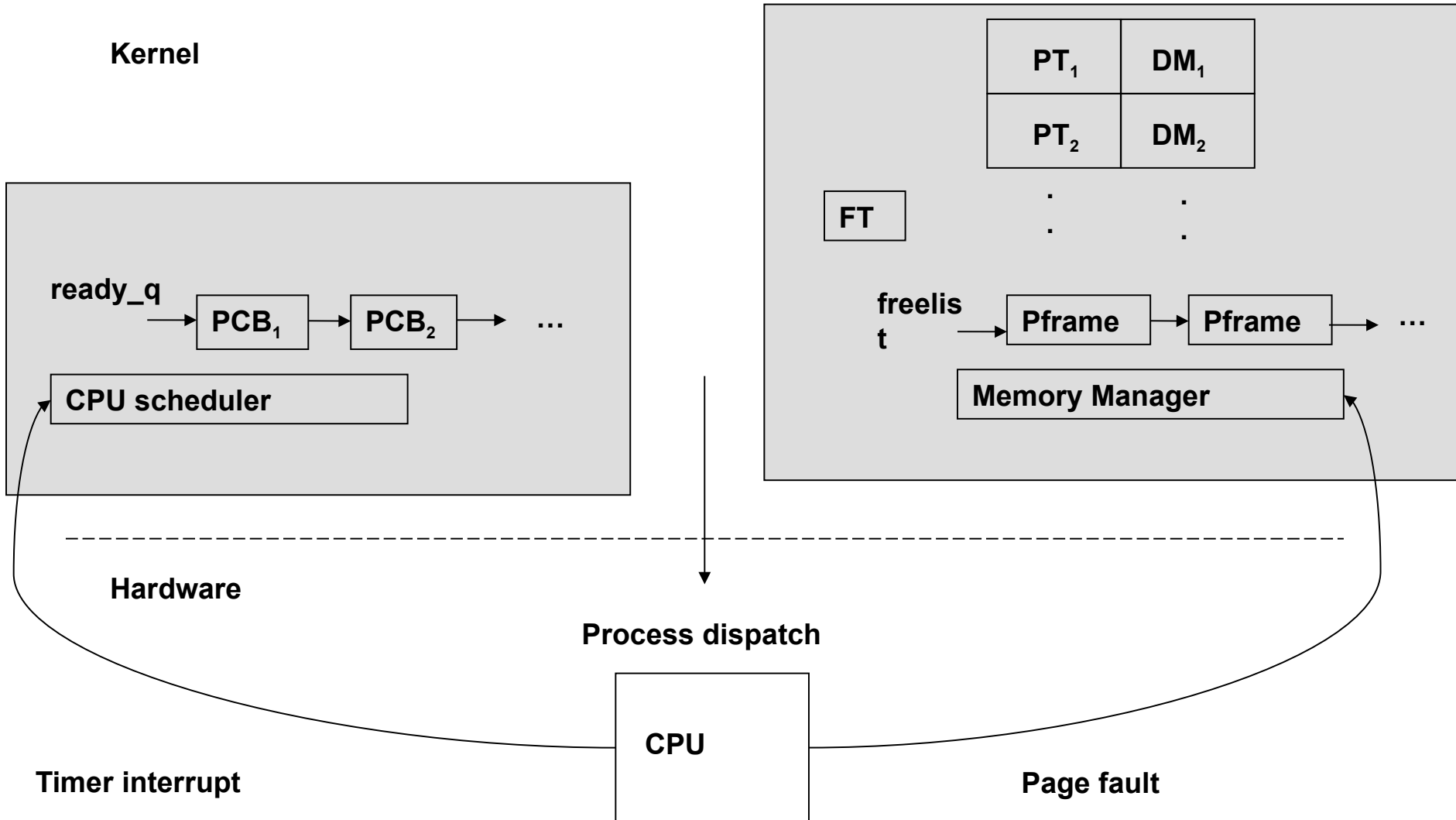
## Hardware

Process dispatch

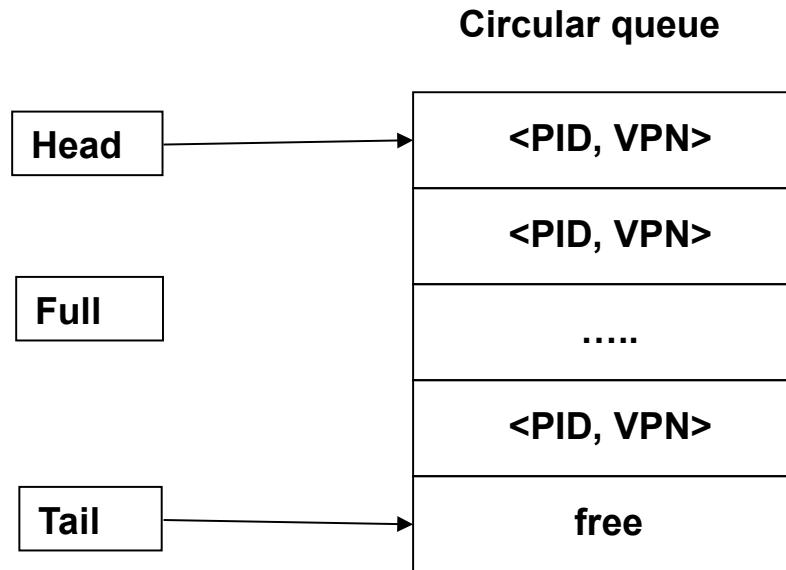
CPU

Timer interrupt

Page fault



# FIFO Page Replacement



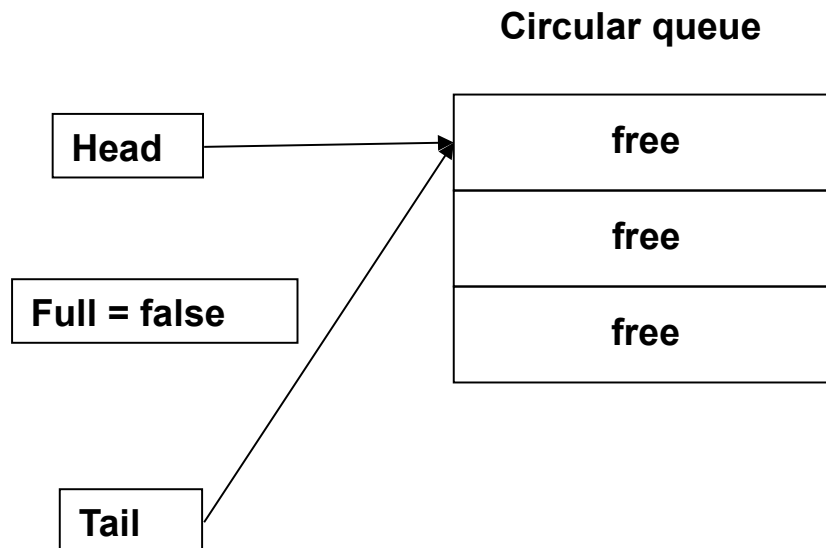
# FIFO example

Consider a string of page references by a process:

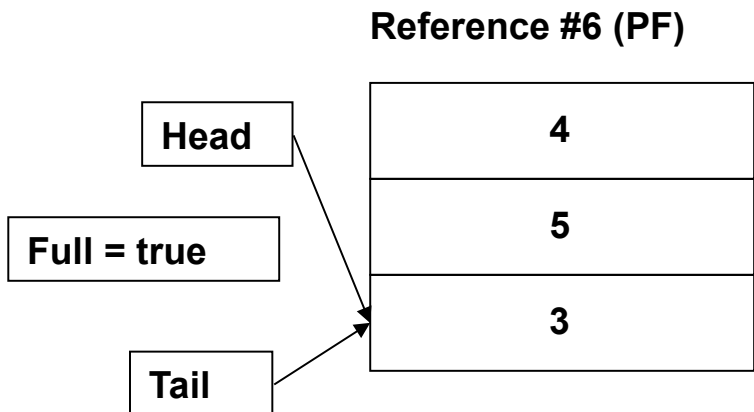
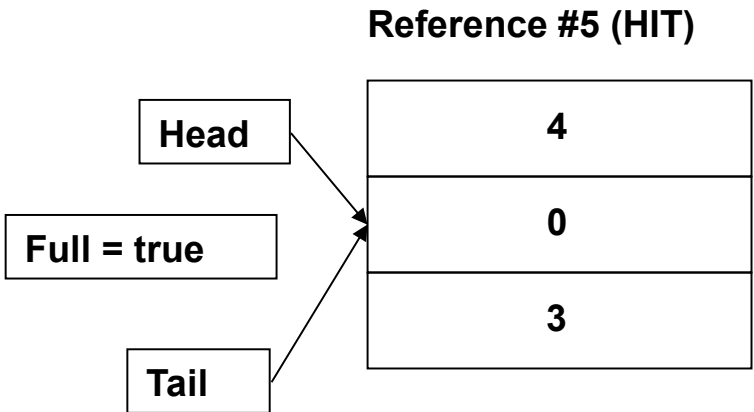
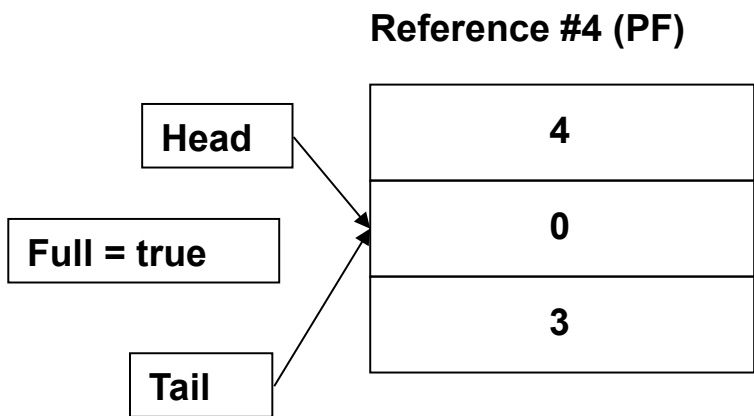
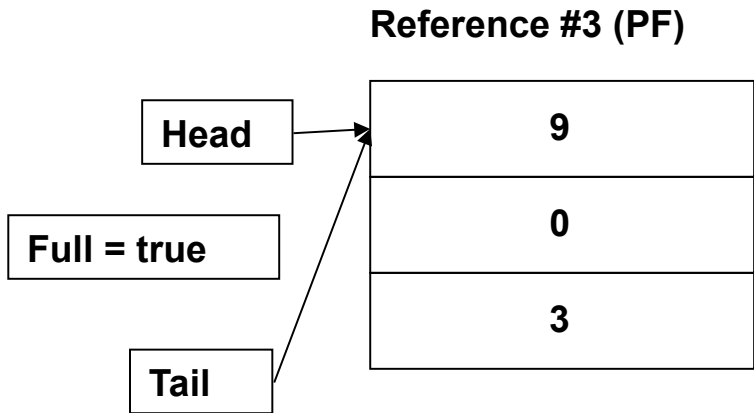
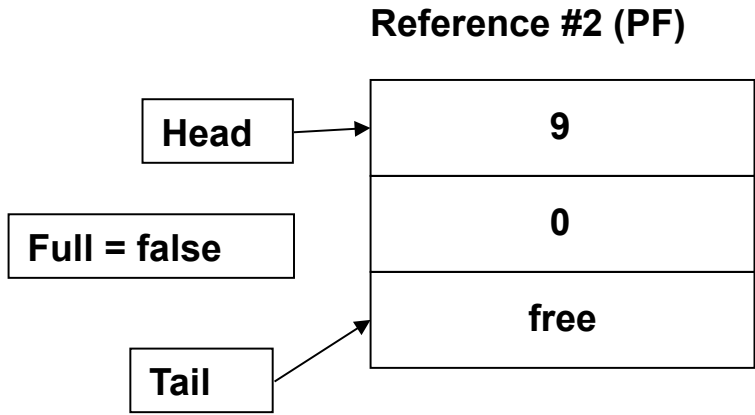
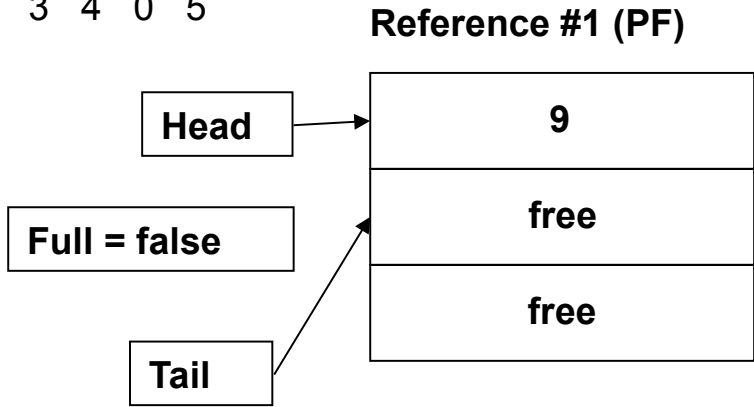
Reference number: 1 2 3 4 5 6 7 8 9 10 11 12 13

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

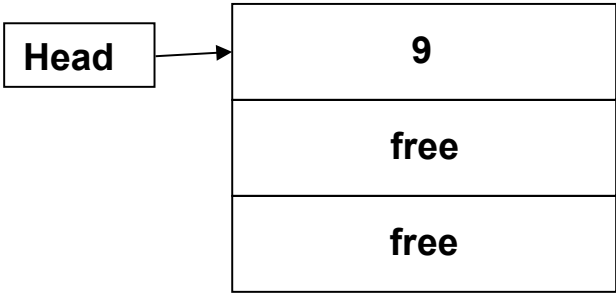


# Using the Frame Table as a Queue

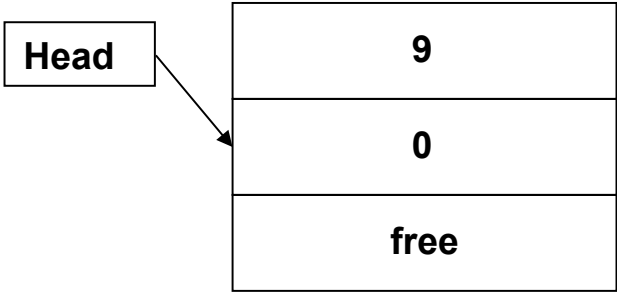
- In real life, there is very little reason to have time stamps or a separate queue for FIFO page replacement.
- That's because the Frame Table will serve that purpose nicely and not take up any extra space!
- So in your mind, you can think of a queue ordered by timestamp for implementing FIFO, but it's never actually implemented that way.
- How? When the system starts up
  - Point the Head pointer to the first Frame Table entry; you don't really need the Tail pointer or Full flag
  - Treat the end of the Frame Table as the limit of the circular queue after which the queue wraps around
  - As you fill in the pages, they will naturally be ordered by time so when you wrap around, you will be considering the first-in pages first!

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

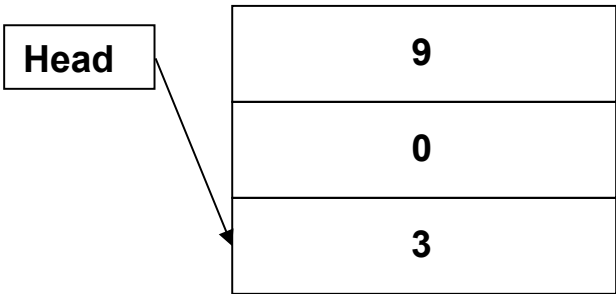
Reference #1 (PF)



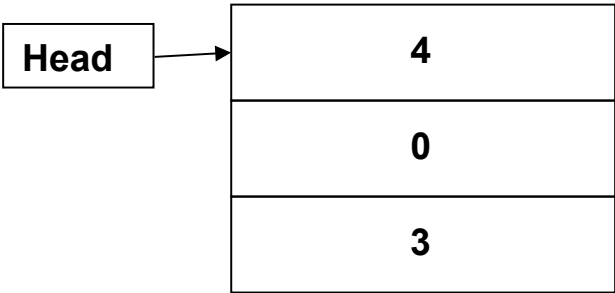
Reference #2 (PF)



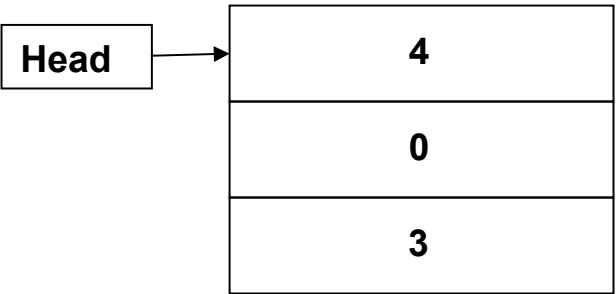
Reference #3 (PF)



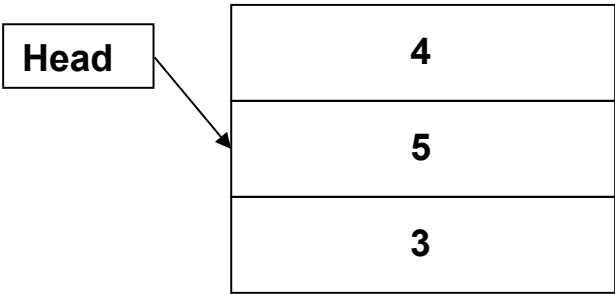
Reference #4 (PF)



Reference #5 (HIT)



Reference #6 (PF)





# Frame Table as FIFO Queue

- If you read carefully, the textbook implies that this is the way to implement FIFO replacement, but it never makes that statement crystal clear
- Also, you should be able to see that our Head pointer is acting like a “clock hand,” pointing to each frame in turn
- We’ll use that term a bit later when we implement a version of LRU paging (which will still be based on this FIFO algorithm!)

# Size of the FIFO queue

- A. 3
- 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

- 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

# Question

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

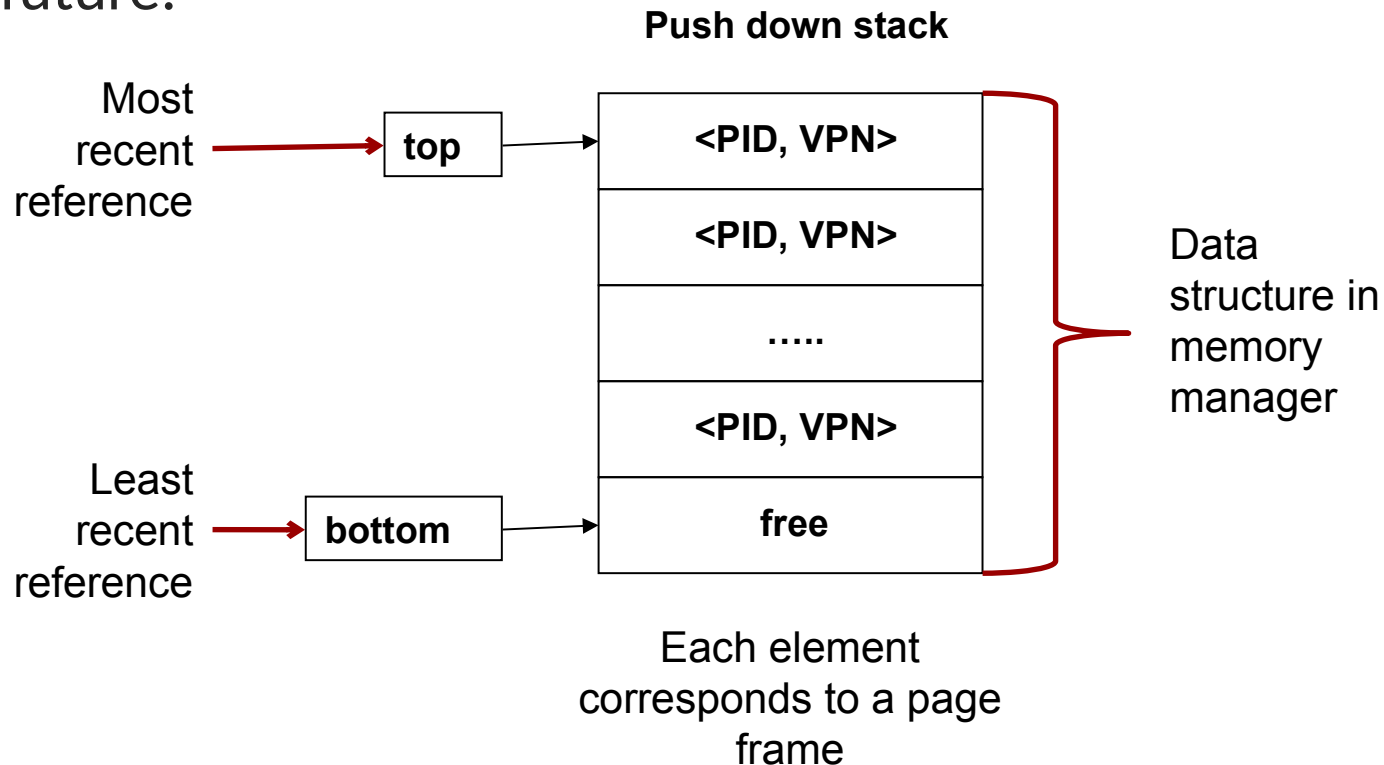
- A. The CPU time required by the algorithm is excessive
- B. The hardware required to support the implementation is too expensive
- C. The algorithm is patented in the US and we'd have to obtain a license and pay royalties
- D. It would require a component that we don't know how to build



Today's number is 10202

# LRU

- Use the past as a predictor of the future.



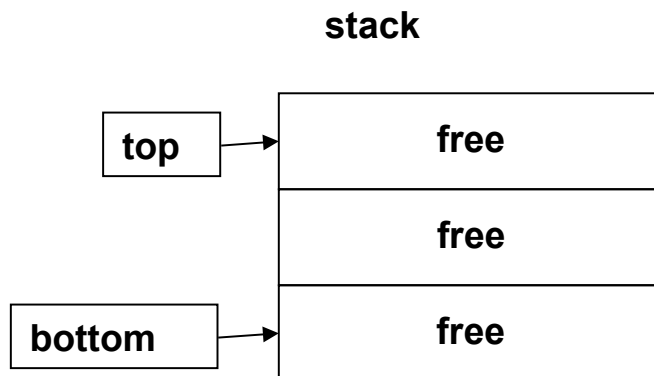
# LRU example

Consider a string of page references by a process:

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

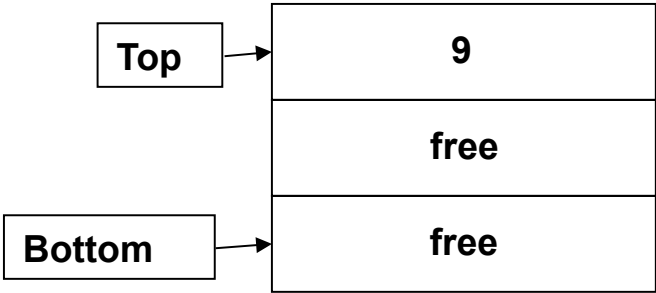
-----  
Virtual page number: 9 0 3 4 0 5 0 6 4 5 0 5 4

Assume there are 3 physical frames.

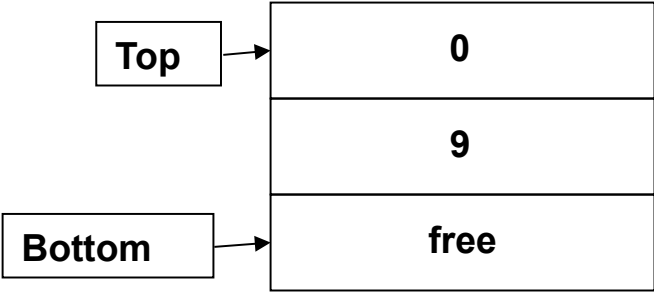


Ref: 1 2 3 4 5 6  
VPN: 9 0 3 4 0 5

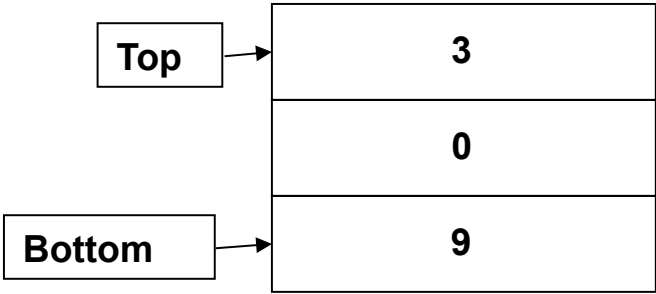
Reference #1 (PF)



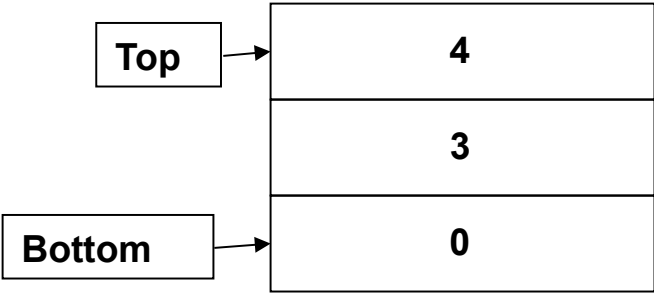
Reference #2 (PF)



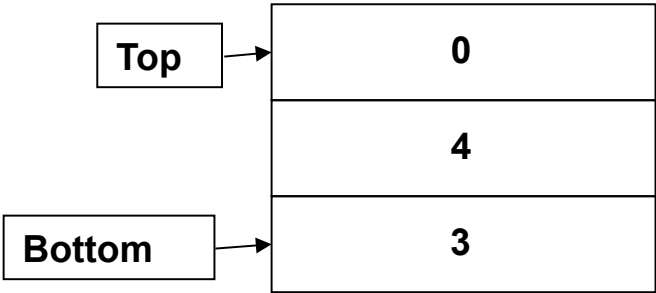
Reference #3 (PF)



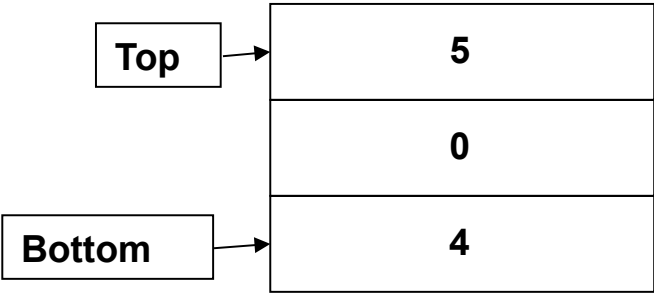
Reference #4 (PF)



Reference #5 (HIT)




Reference #6 (PF)



# Question

How many entries are needed for a complete LRU “stack”

- A. 1
- B. Number of virtual pages
-  C. Number of physical page 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 and a memory write is required for each memory reference

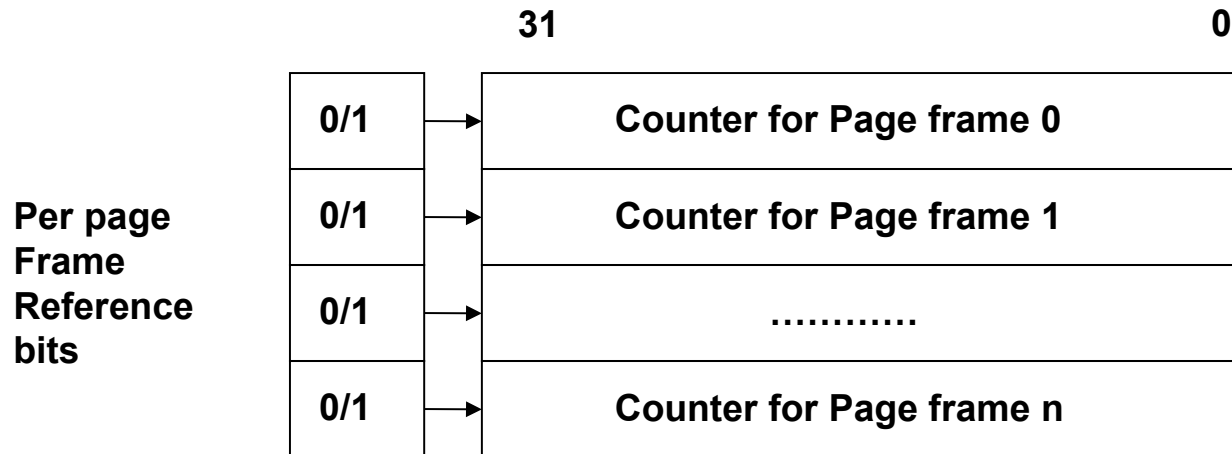
# Ways to approximate LRU

- 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**

<b>PFN</b>	<b>Ref</b>	<b>Valid</b>
<b>PFN</b>	<b>Ref</b>	<b>Valid</b>
<b>PFN</b>	<b>Ref</b>	<b>Valid</b>

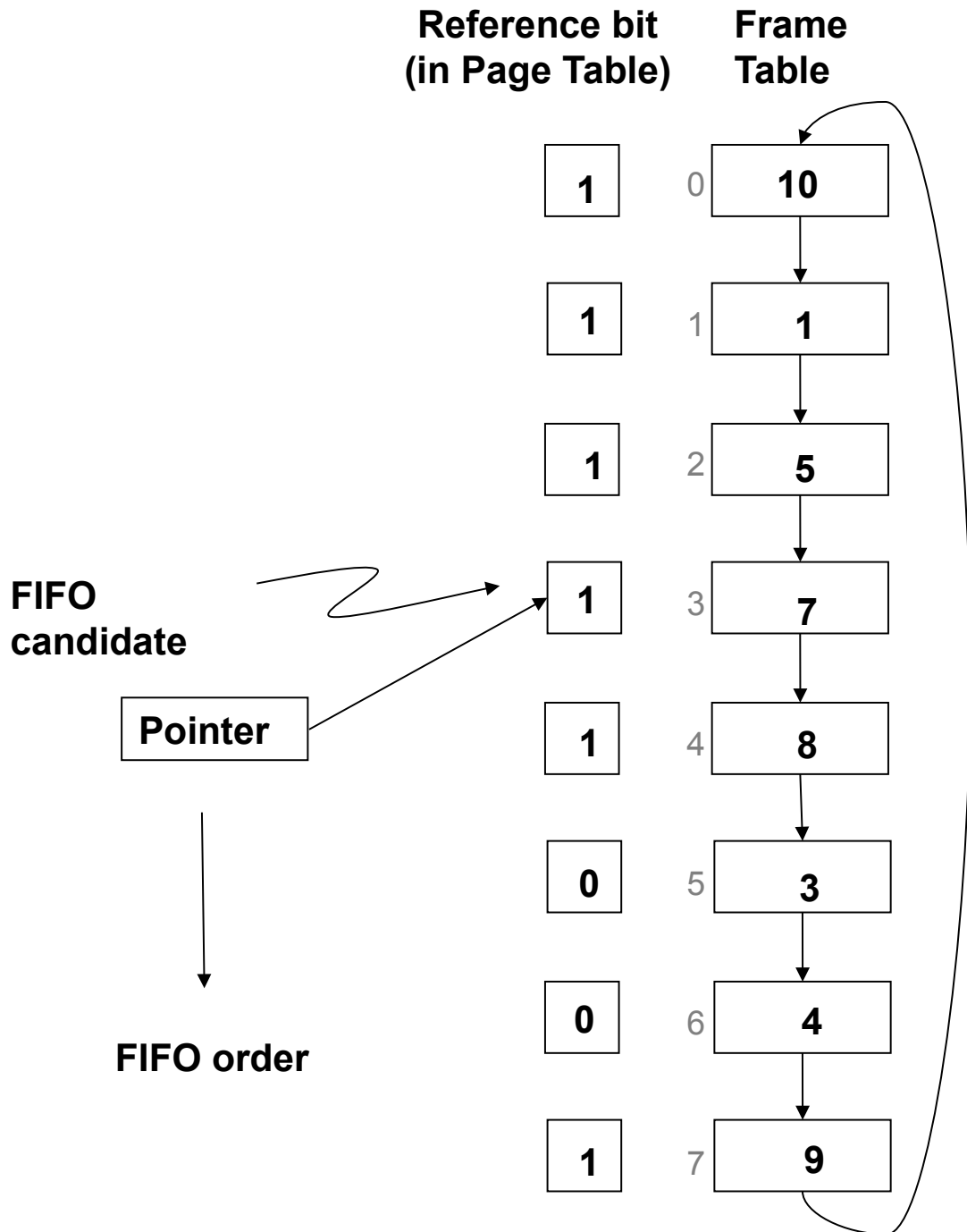
# Approximate LRU with ref bits



- 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  
$$\text{counter} = (\text{ref} \ll 31 \mid \text{counter} \gg 1)$$
  - **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 referenced 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



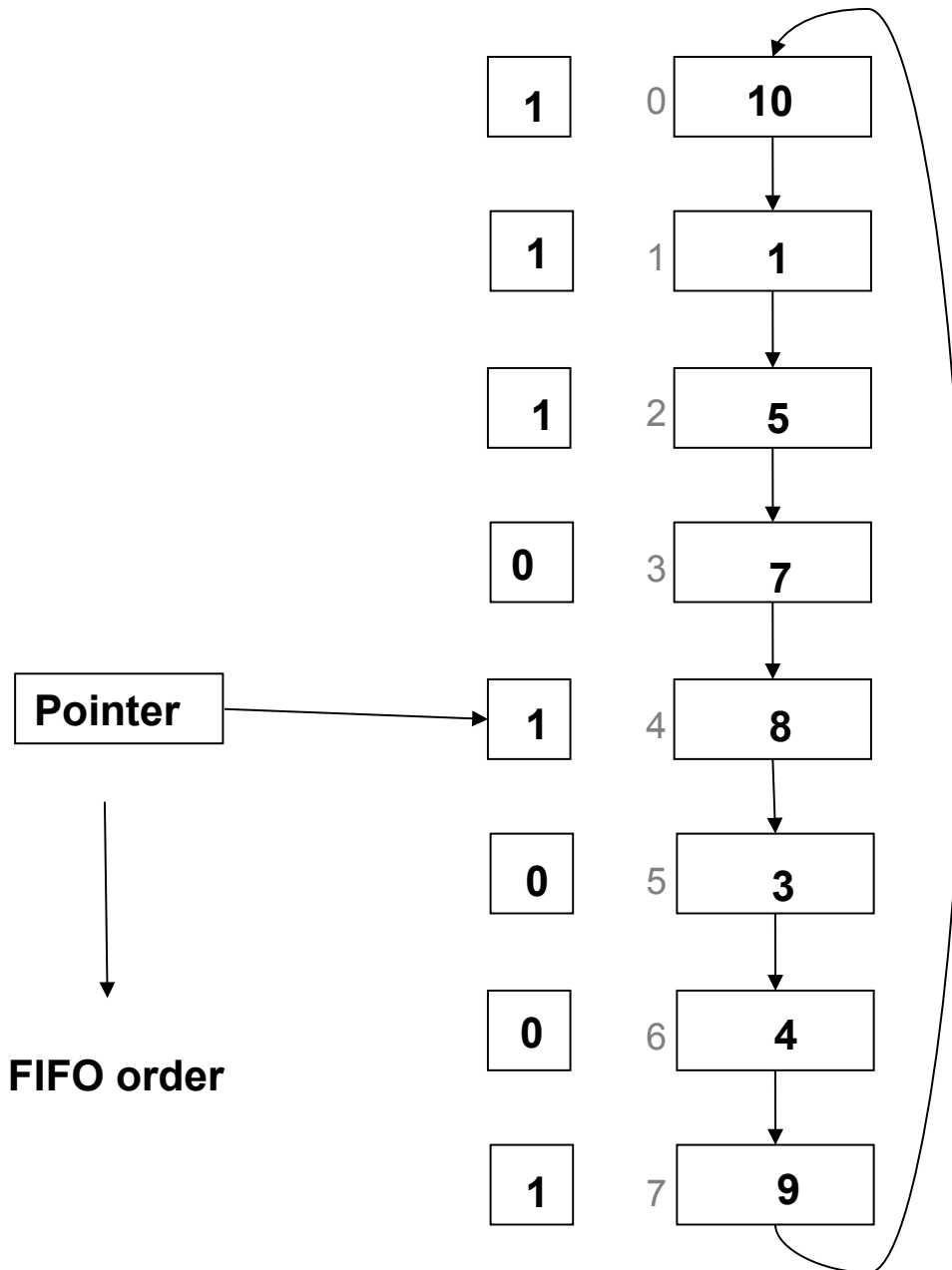
Try page 7

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

# Second chance replacement

## Reference bit (in Page Table)

## Frame Table

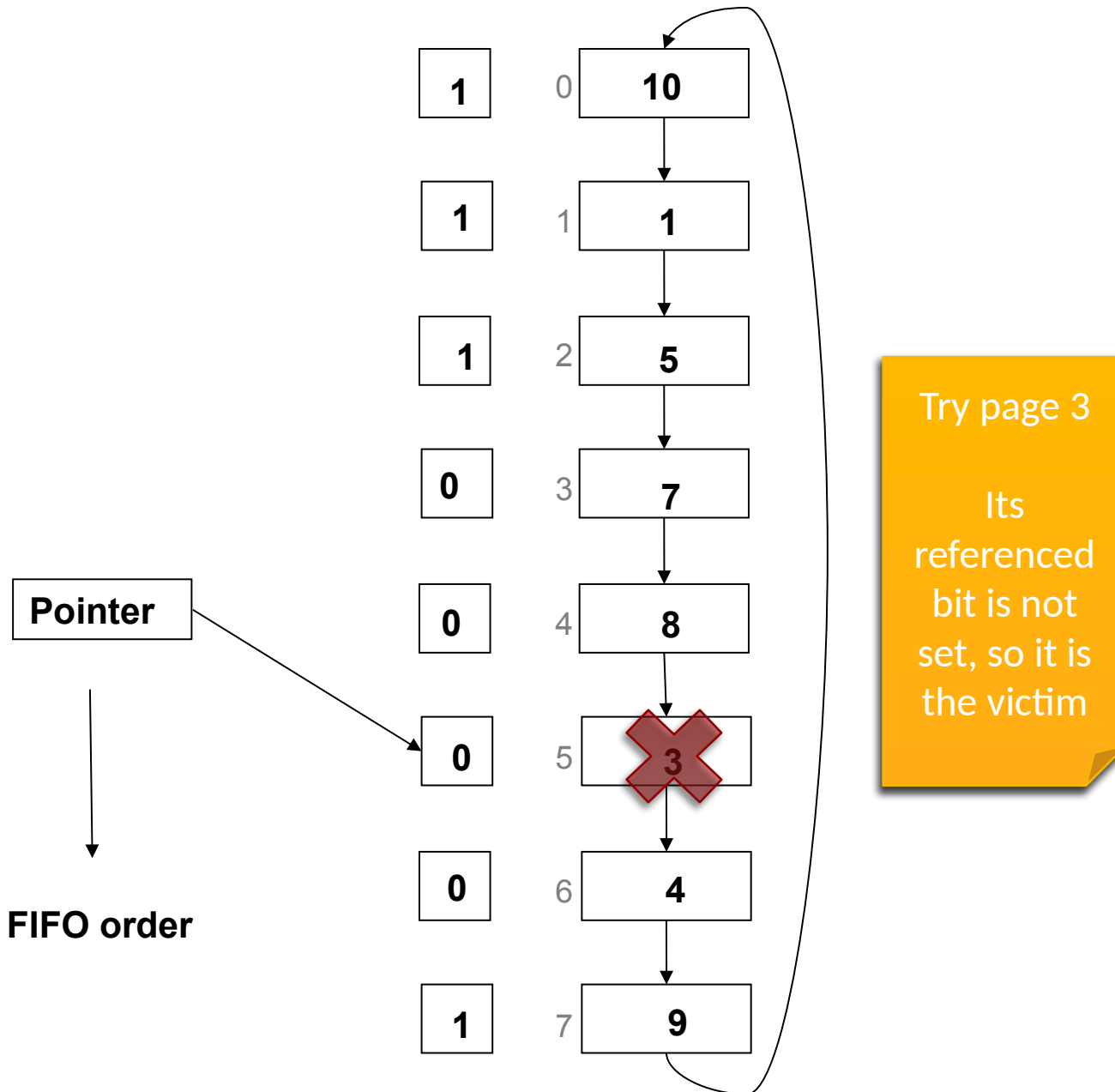


Try page 8

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

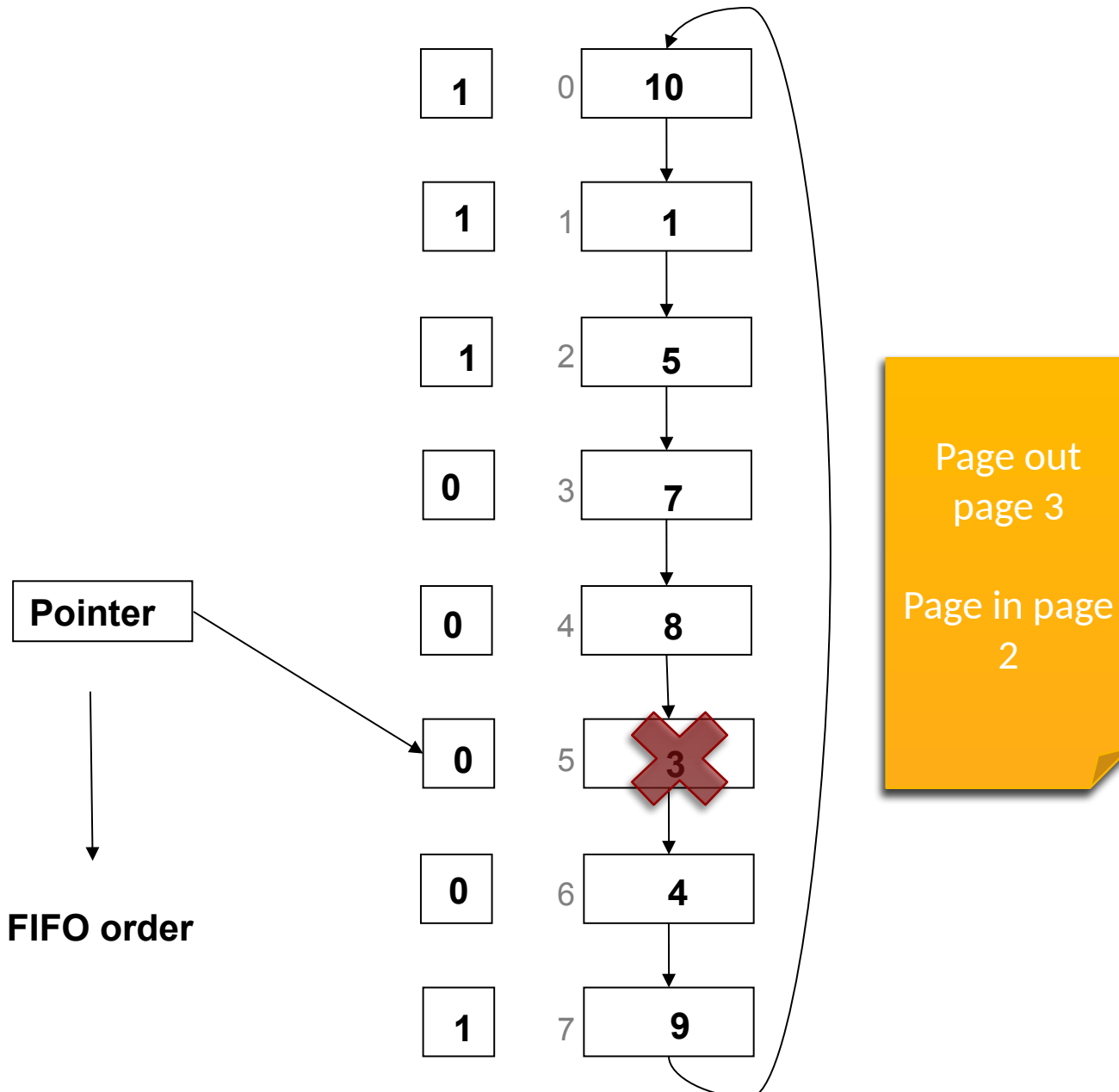
Second chance  
replacement

# Reference bit (in Page Table) Frame Table



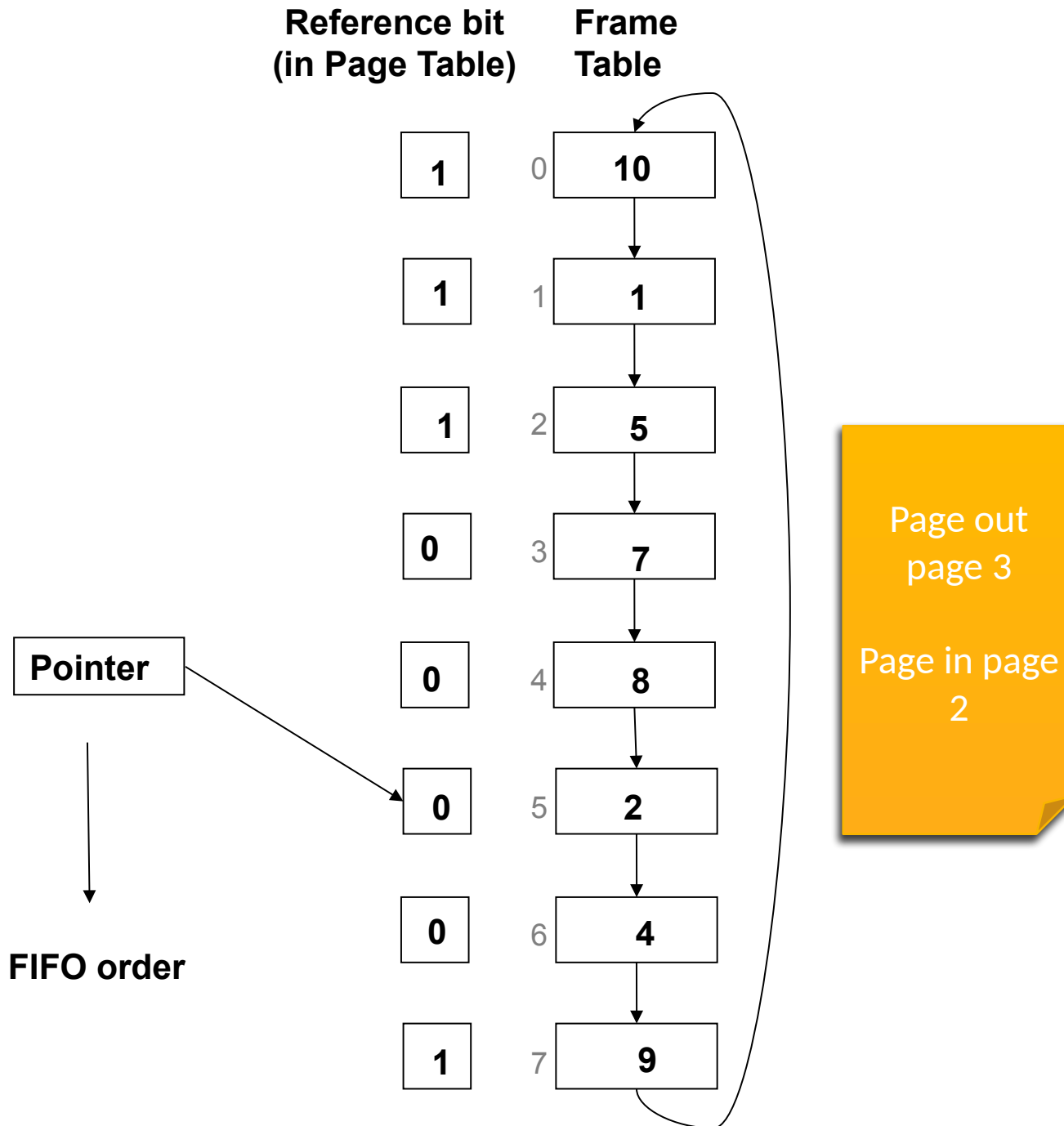
# Second chance replacement

# Reference bit (in Page Table)      Frame Table



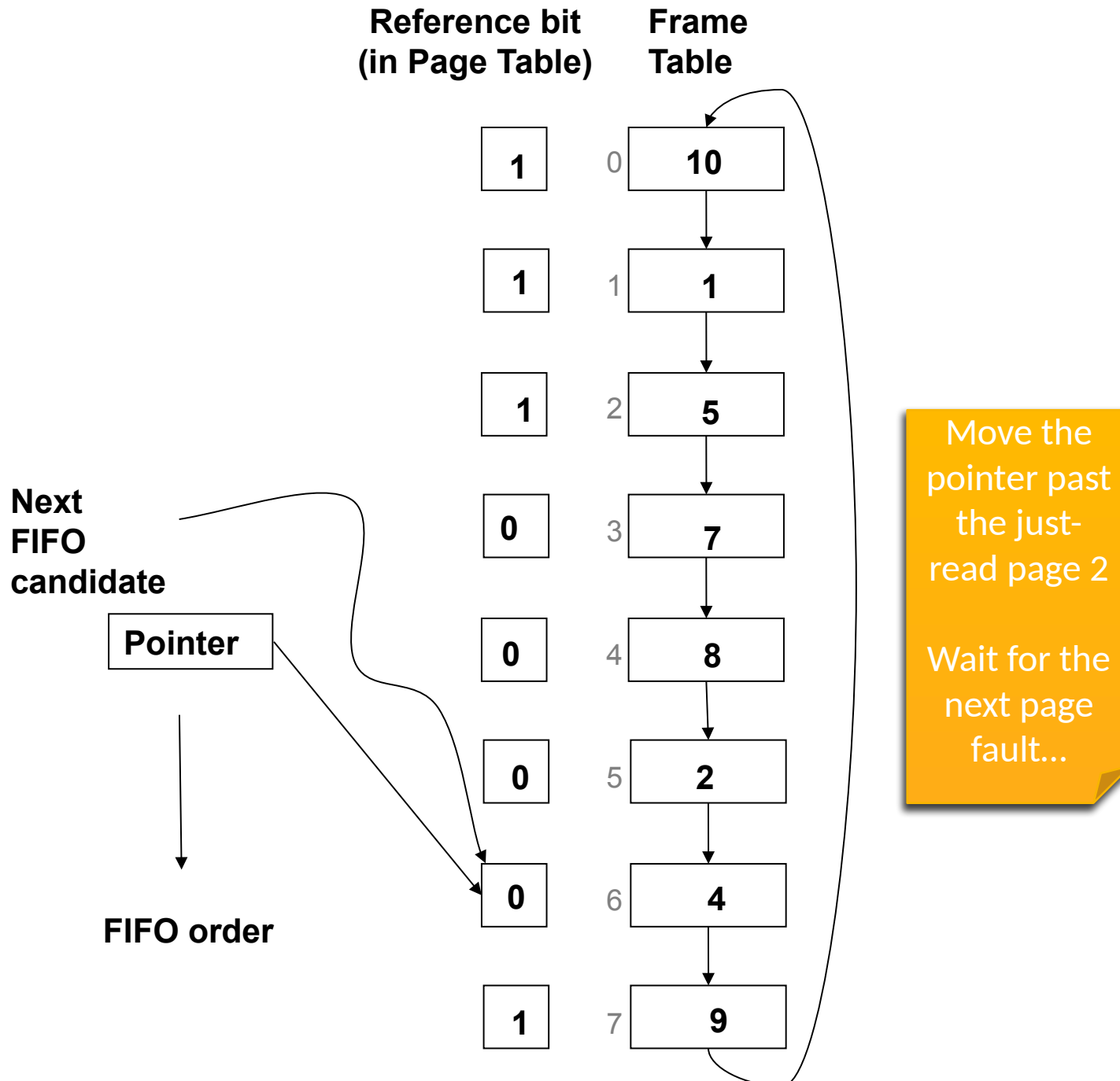
Second chance  
replacement





Second chance  
replacement

# Second chance replacement



# Page replacement algorithms

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

# Page Table Entry (PTE) Example

VPN	Valid	PFN	Referenced (optional)	Dirty (optional)	Read-only (optional)	Other info
0	1	12	0	0	1	
1	1	16	0	0	0	
2	0	14	0	0	0	
3	1	36	1	0	0	
4	1	9	1	0	0	No execute

```
typedef struct ft_entry {
    uint8_t protected;           /* set if the frame holds a page that should be
                                   immune from eviction */

    uint8_t mapped;              /* set if the frame is mapped. */
    uint8_t referenced;          /* set if the entry has been recently accessed. */
    pcb_t *process;              /* A pointer to the owning process's PCB */
    vpn_t vpn;                   /* The VPN mapped by the process using this frame. */
} fte_t;
```

```
/**
 * An entry in the page table.
 *
 * Note that the VPN is not stored in the entry - it's the index into
 * the page table!
 */
```

```
typedef struct ptable_entry {
    uint8_t valid;               /* set if the entry is mapped to a valid frame. */
    uint8_t dirty;              /* set if the entry is modified while in main
                                   memory */
    pfn_t pfn;                  /* The physical frame number (PFN) this entry
                                   maps to. */
    swap_id_t sid;              /* The swap entry mapped to this page. Use this
                                   to read to/write from the page to disk using
                                   swap_read() and swap_write() */
} pte_t;
```

Oops!

# Back to our pipelined processor

➤ With virtual memory...

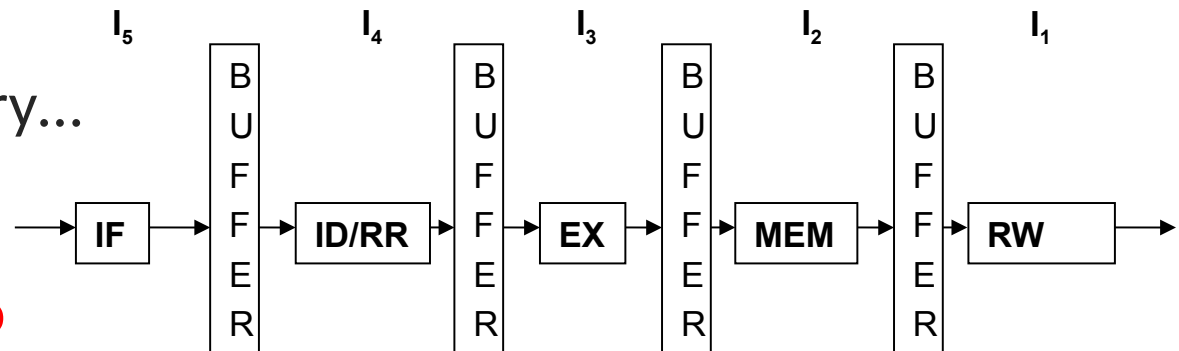
➤ Every memory access requires two memory accesses!

➤ PTE

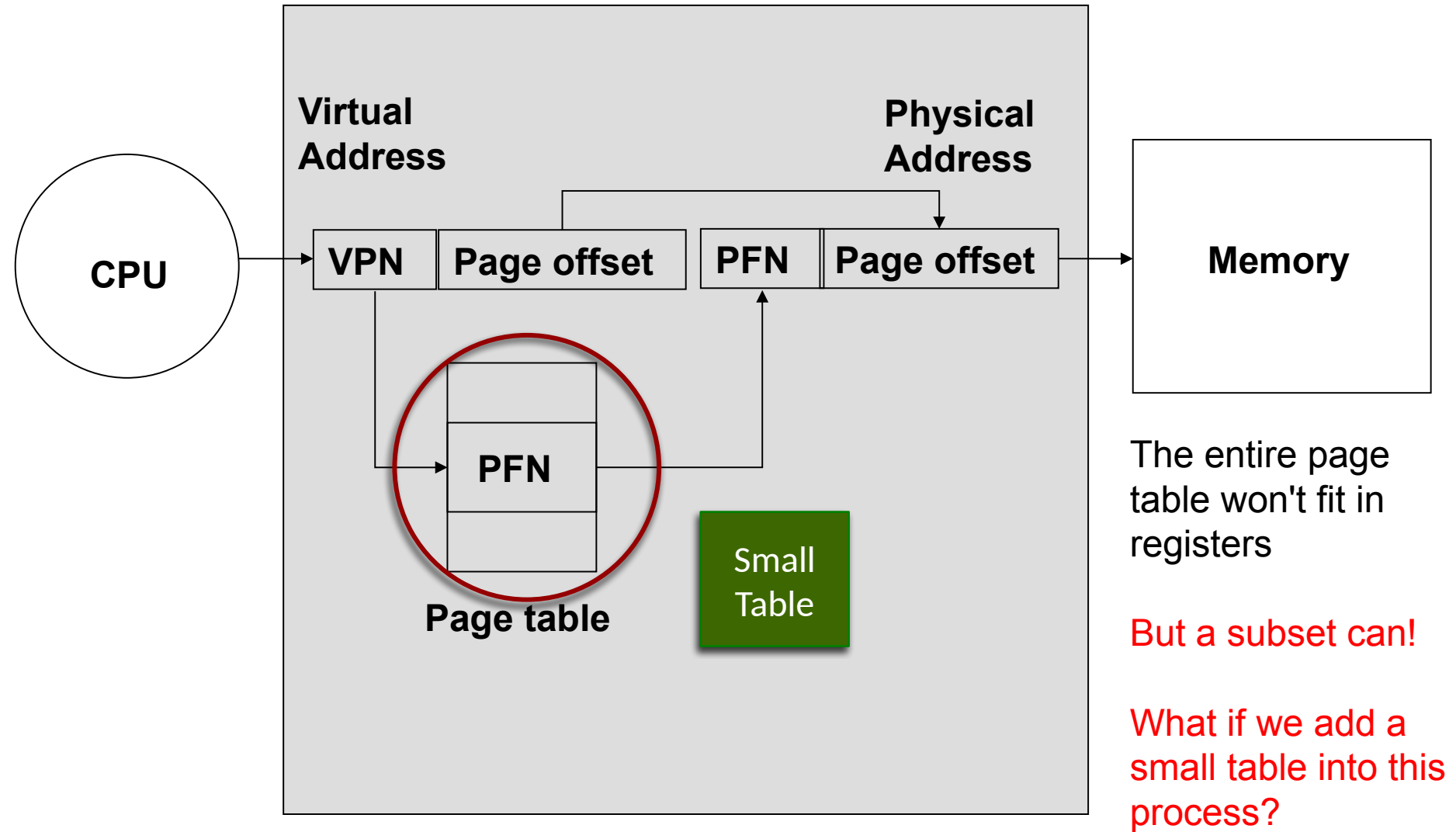
➤ Actual memory word

➤ This is bad news for the pipeline

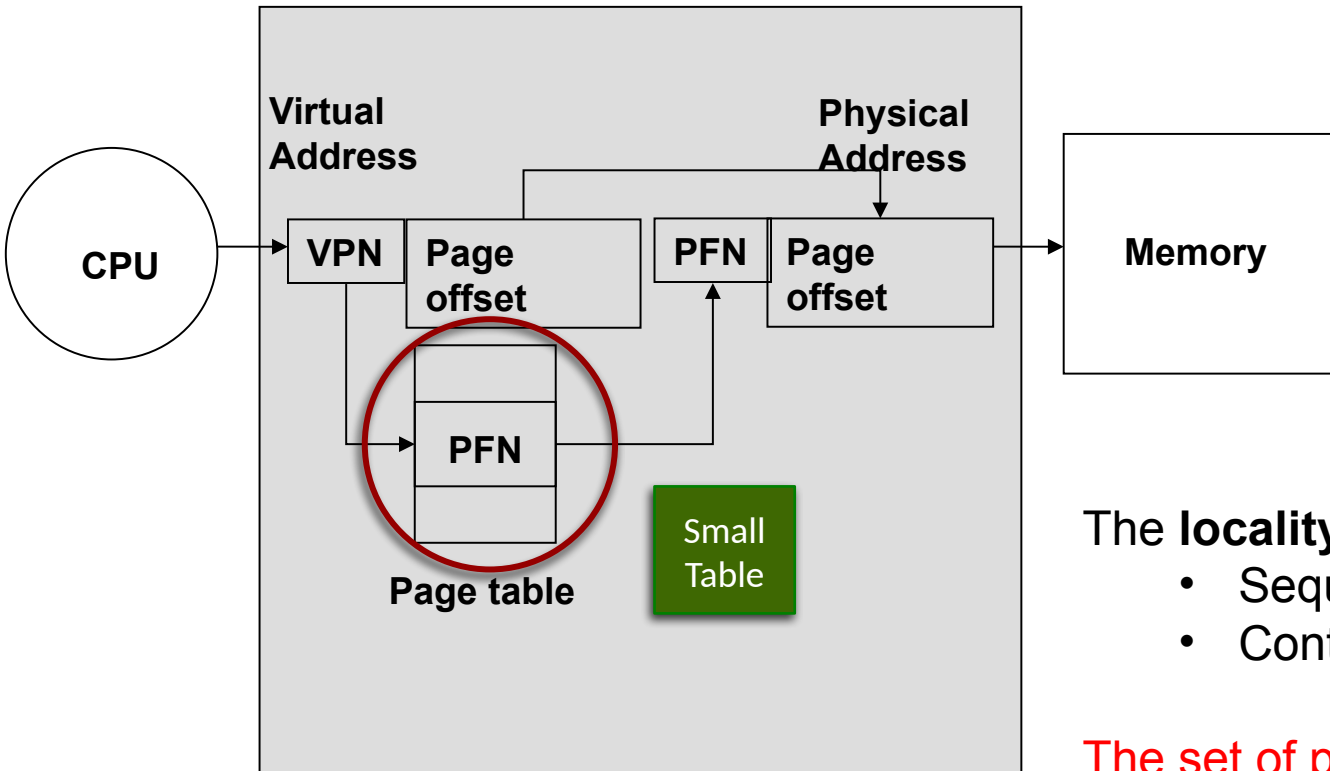
➤ At least one bubble for every instruction



# Speeding up address translation



# Why will this work?



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**

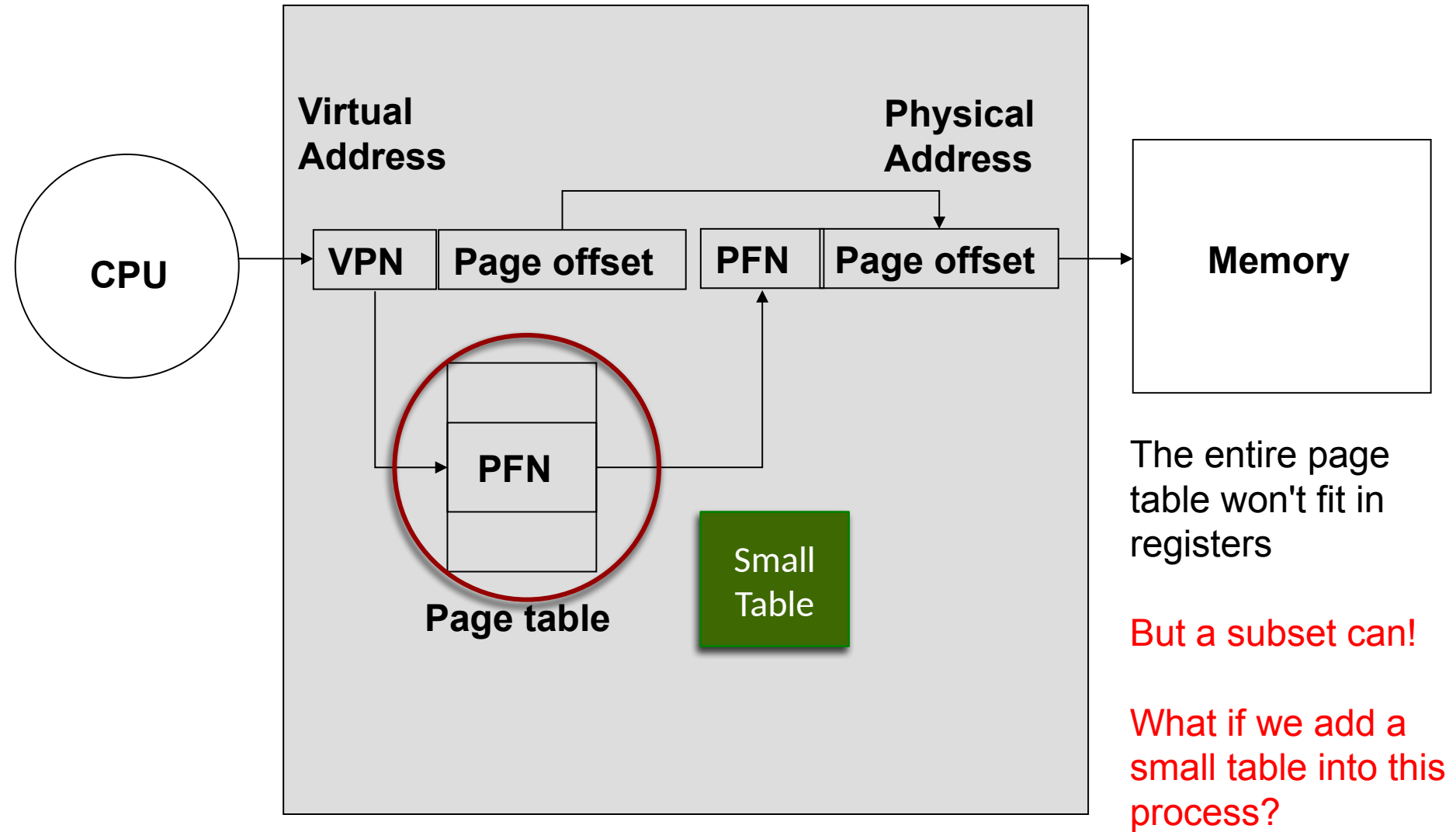


# TLB (translation lookaside buffer)

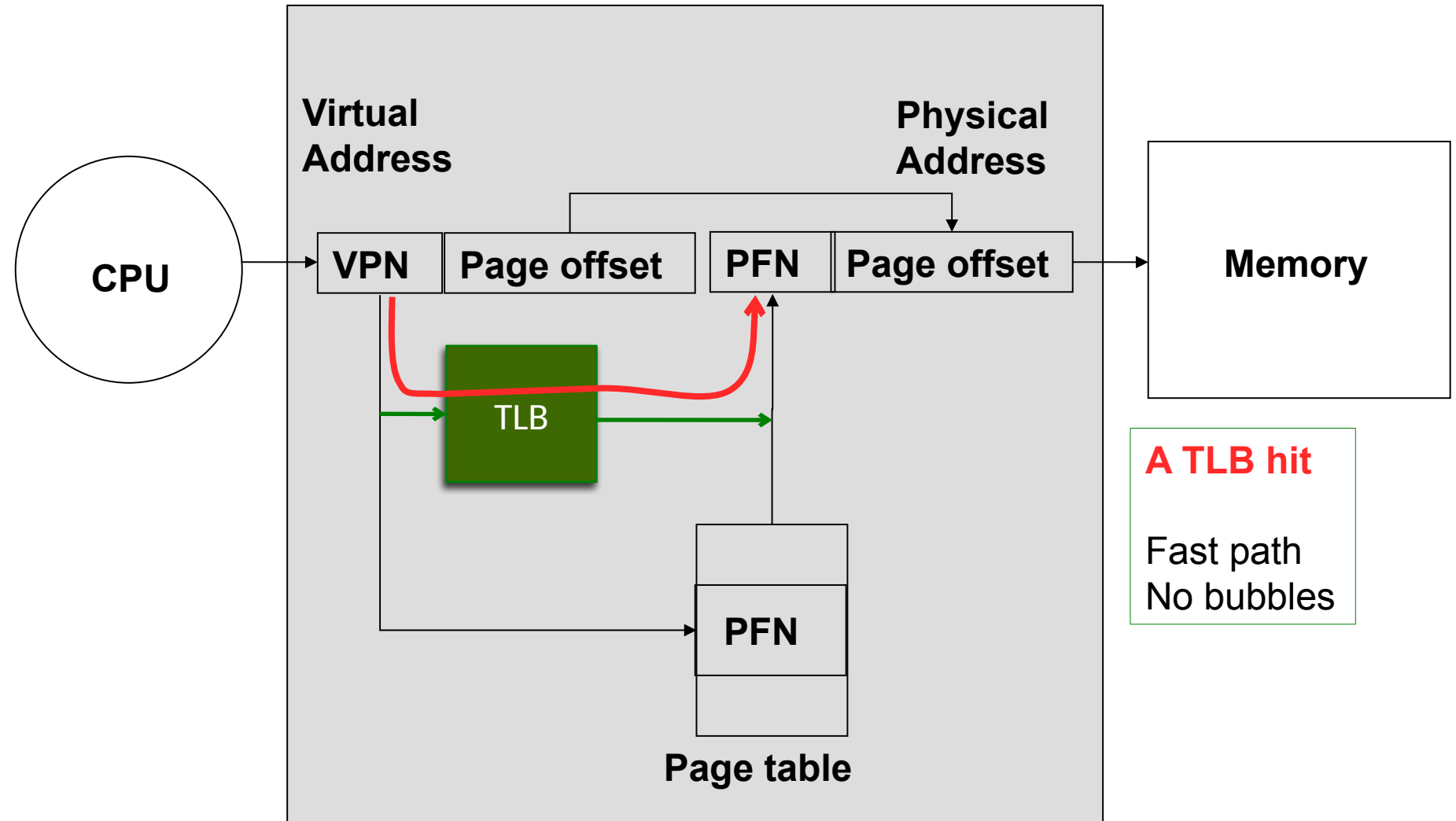
- It will look something like the following table
- It is an **associative** memory, which means it can search on a match on the first two columns and output the corresponding last two columns in one cycle

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

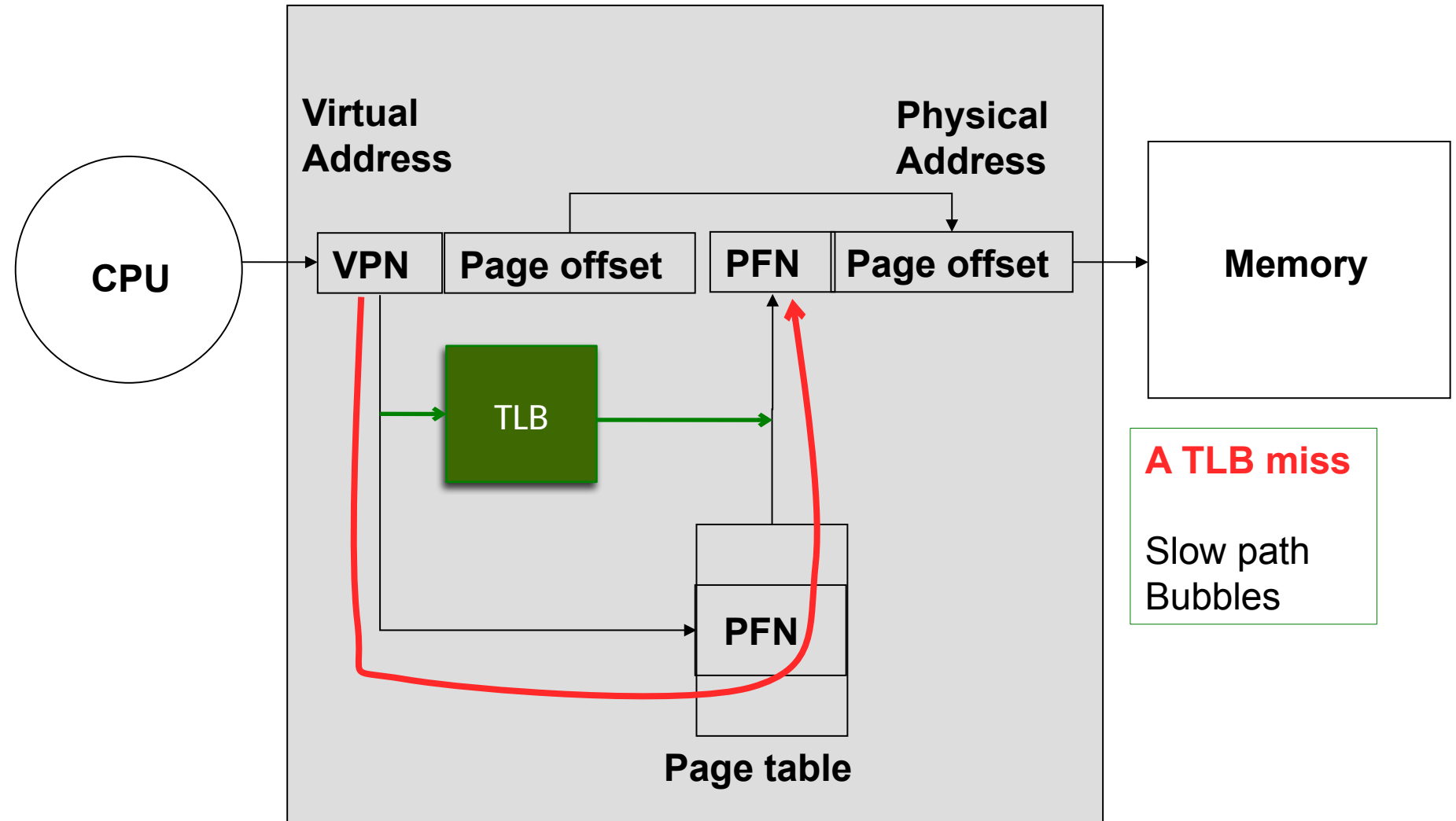
# Speeding up address translation



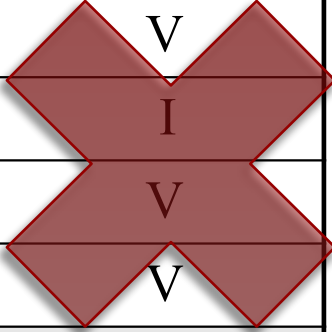
# Speeding up address translation



# Speeding up address translation



# 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
- Who can execute this instruction?
  - **It can only be executed by the kernel**

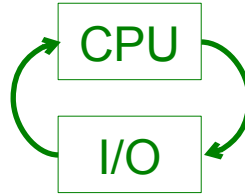
# Question

Upon a context switch...

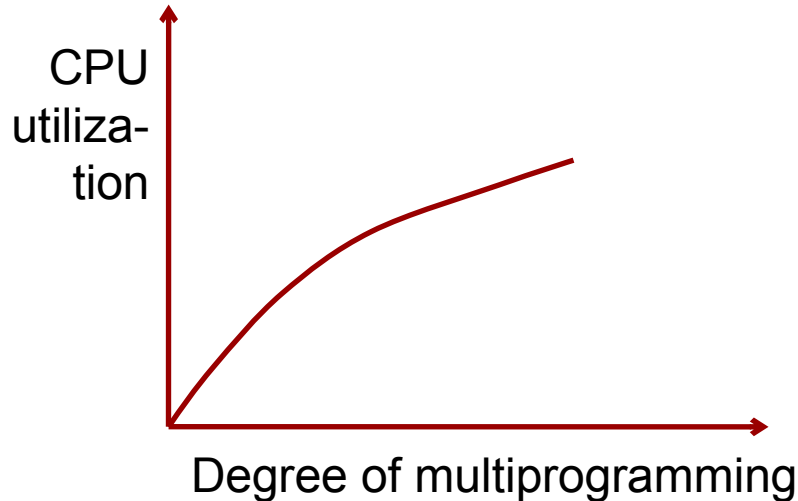
- A. The entire TLB is flushed
- B. Only the kernel portion of the TLB is flushed
-  C. Only the user portion of the TLB is flushed
- D. No change to the TLB
- E. Is class over yet?

Today's number is 30,332

# Given the nature of a process

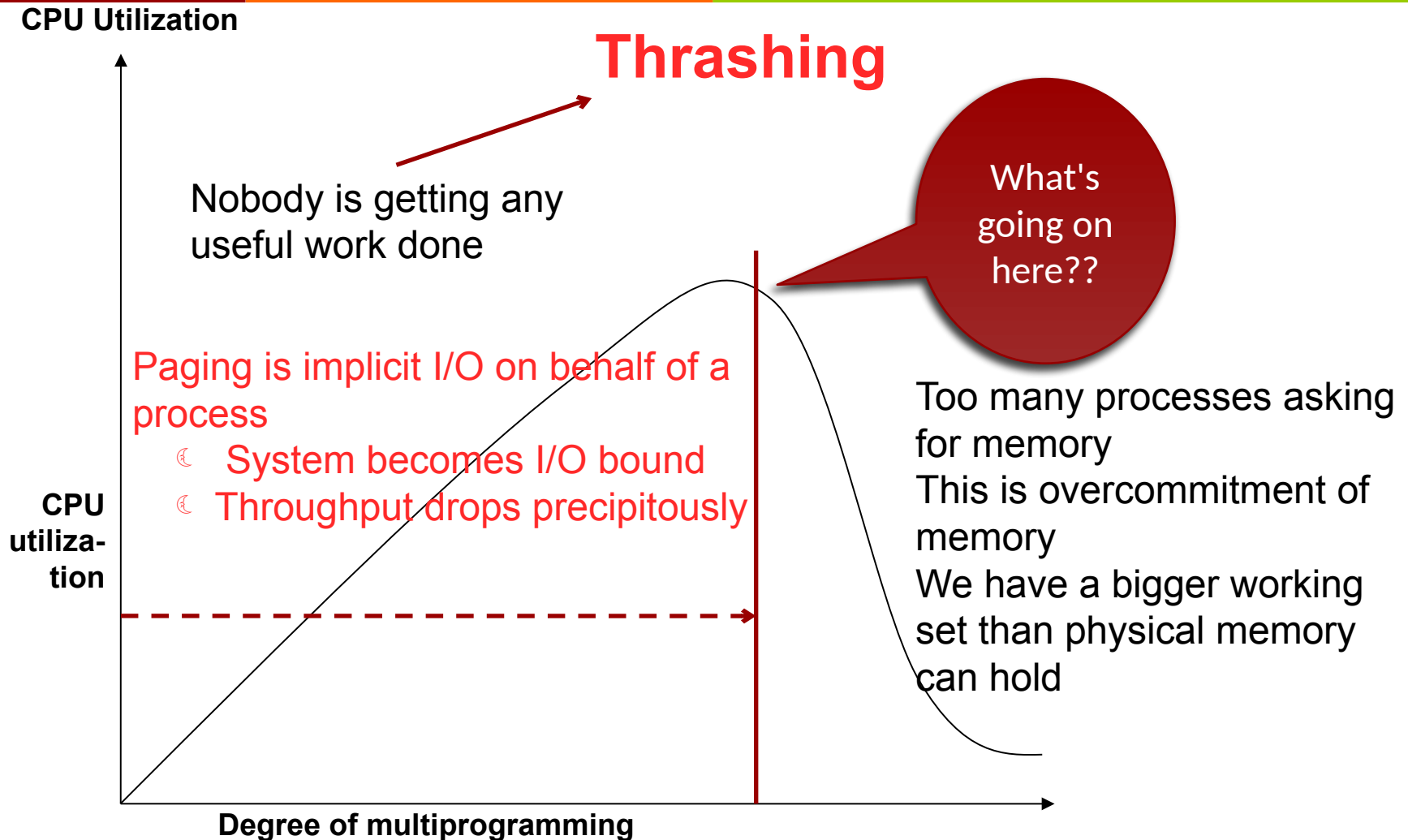


- We want to increase multiprogramming to keep the CPU busy doing useful work
- This is what we want to see:



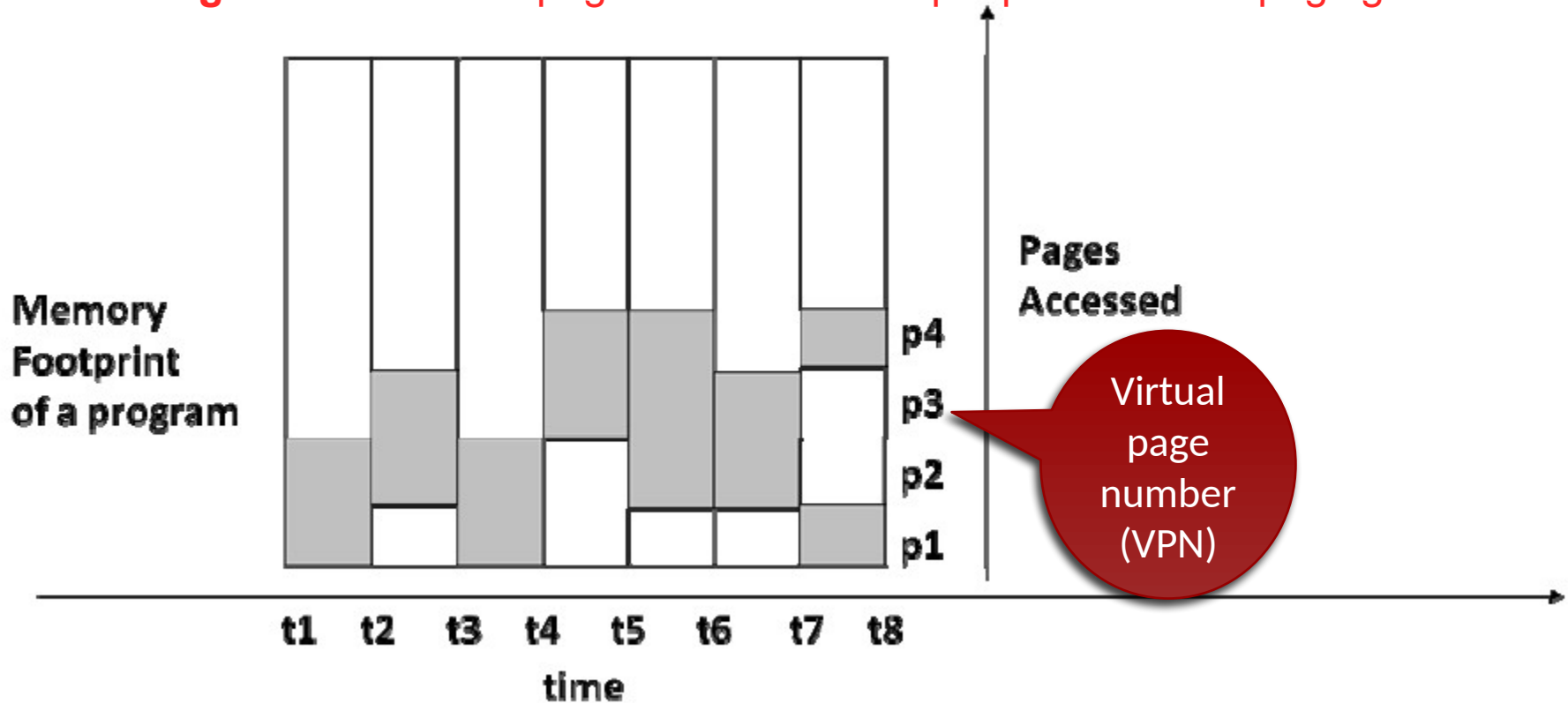


# Extending the utilization curve



# Working set of a program

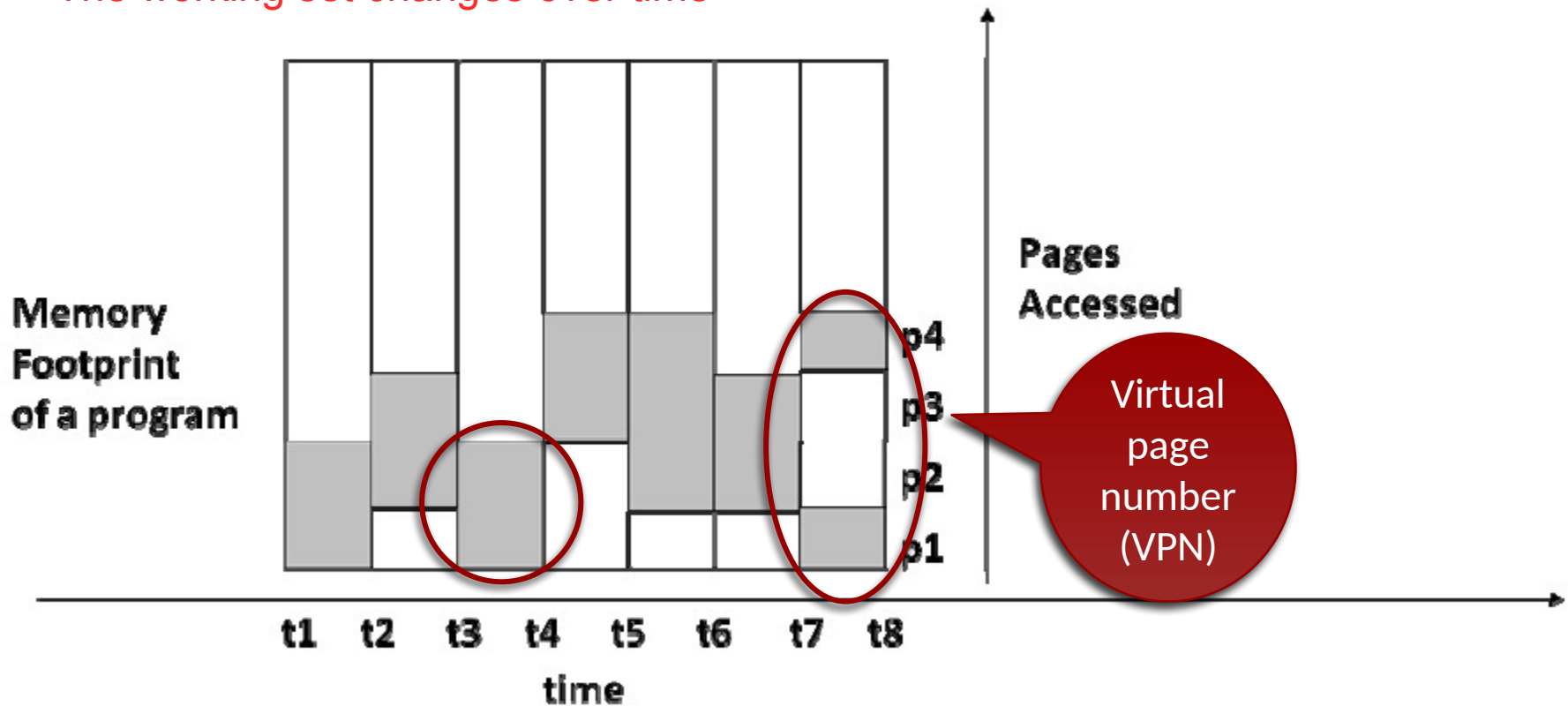
**Working set** is the set of pages needed to keep a process from paging



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

# Working set of a program

The working set changes over time

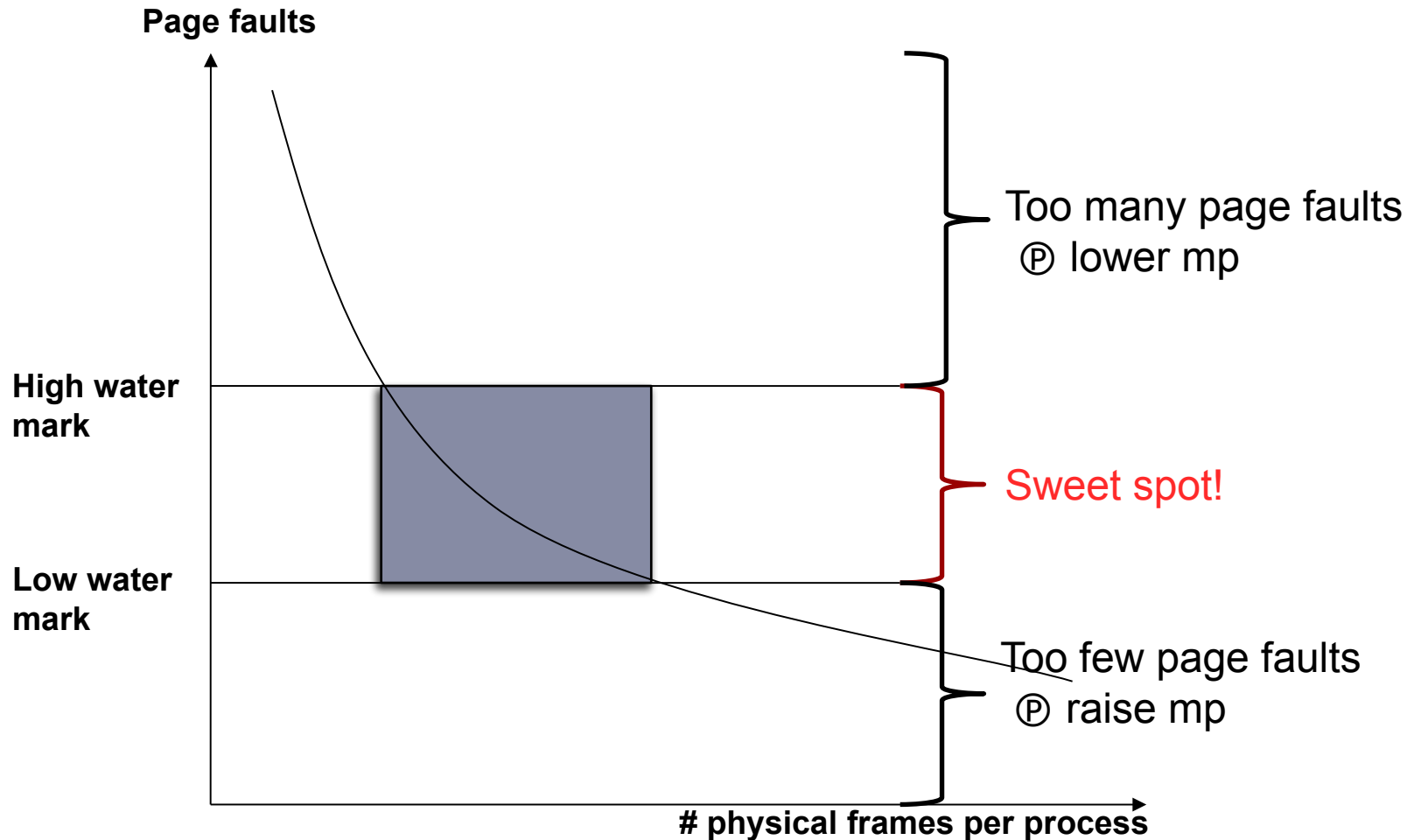


# Memory pressure

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


- $P_1, P_2, P_3, \dots$  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
  - $\sum WSS > \text{total physical memory}$ 
    - Ⓟ swap out some processes
  - $\sum WSS < \text{total physical memory}$ 
    - Ⓟ 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

# Question

We can tell a system is thrashing when

- 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



Today's  
number is  
81,818

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 RL, 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.

# Question

We can reduce thrashing by



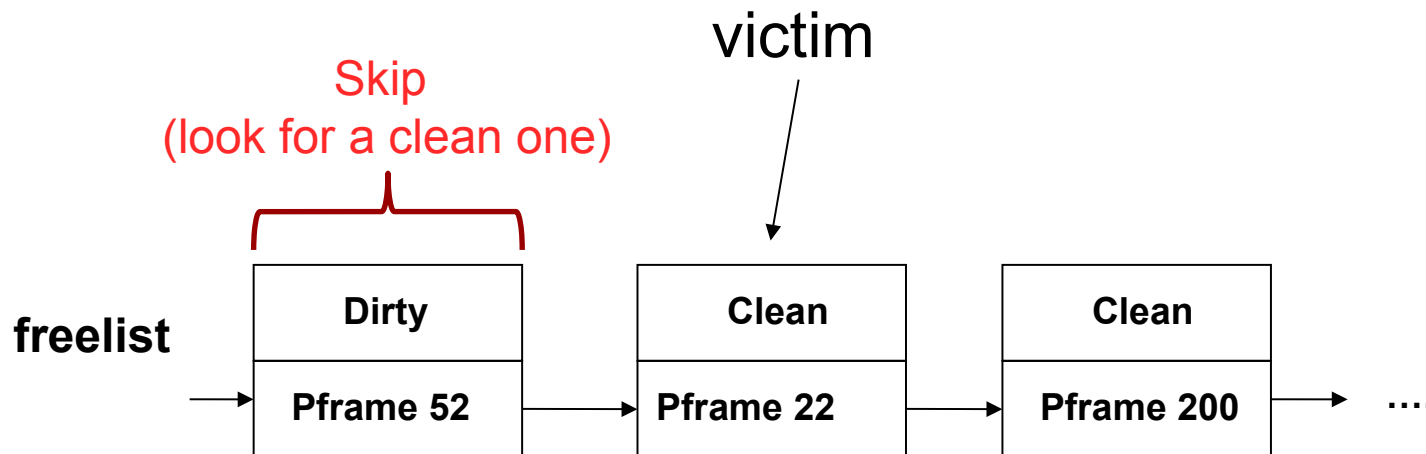
- A. Using a medium-term scheduler that suspends processes until the condition improves
- B. Reducing the physical memory size
- C. Adding additional processes to increase the multiprogramming factor
- D. Reducing the page size

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



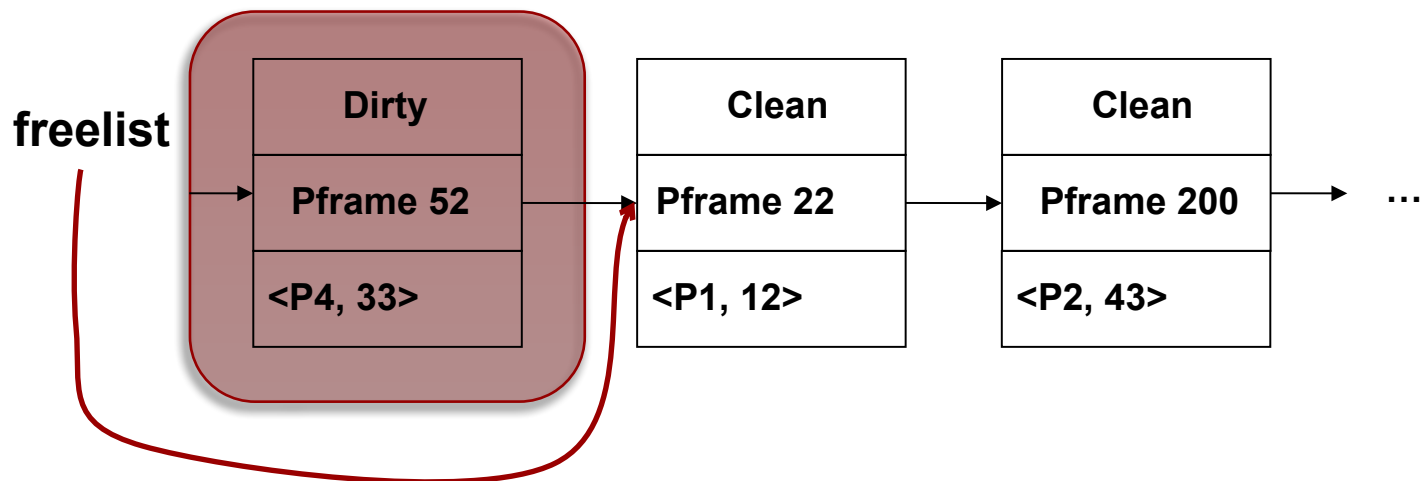
# Optimizations

- Keep a (small) pool of free frames
  - Overlap I/O with processing
  - 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
- Record the PID and VPN when the page is put on the free list
- Example: 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, writing it out if it's dirty



## Page fault handler:



If the target page is still on the free list, it is reclaimed by removing it from the free list and marking its PTE bit “valid”



Otherwise, the first frame on 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

# Segmented virtual memory

- Segmentation is a system allowing a process's memory space to be subdivided into chunks of memory each associated with some aspect of the overall program
- Segmented virtual memory came first, ca. 1960, see Burroughs B5500.
- Segmentation isn't transparent like paging, but it is much more aware of program and data structures
- It's visible to the ISA, so it doesn't work as a retrofit to ISAs with contiguous memory models. IBM couldn't use it in the 1970s because they had already committed to upper/lower bound register memory management and a contiguous address space.
- Hence, we have paging which could be transparently slipped under existing memory management. Other manufacturers followed suit because paging didn't require their applications to be aware of paging.

# Segmented virtual memory

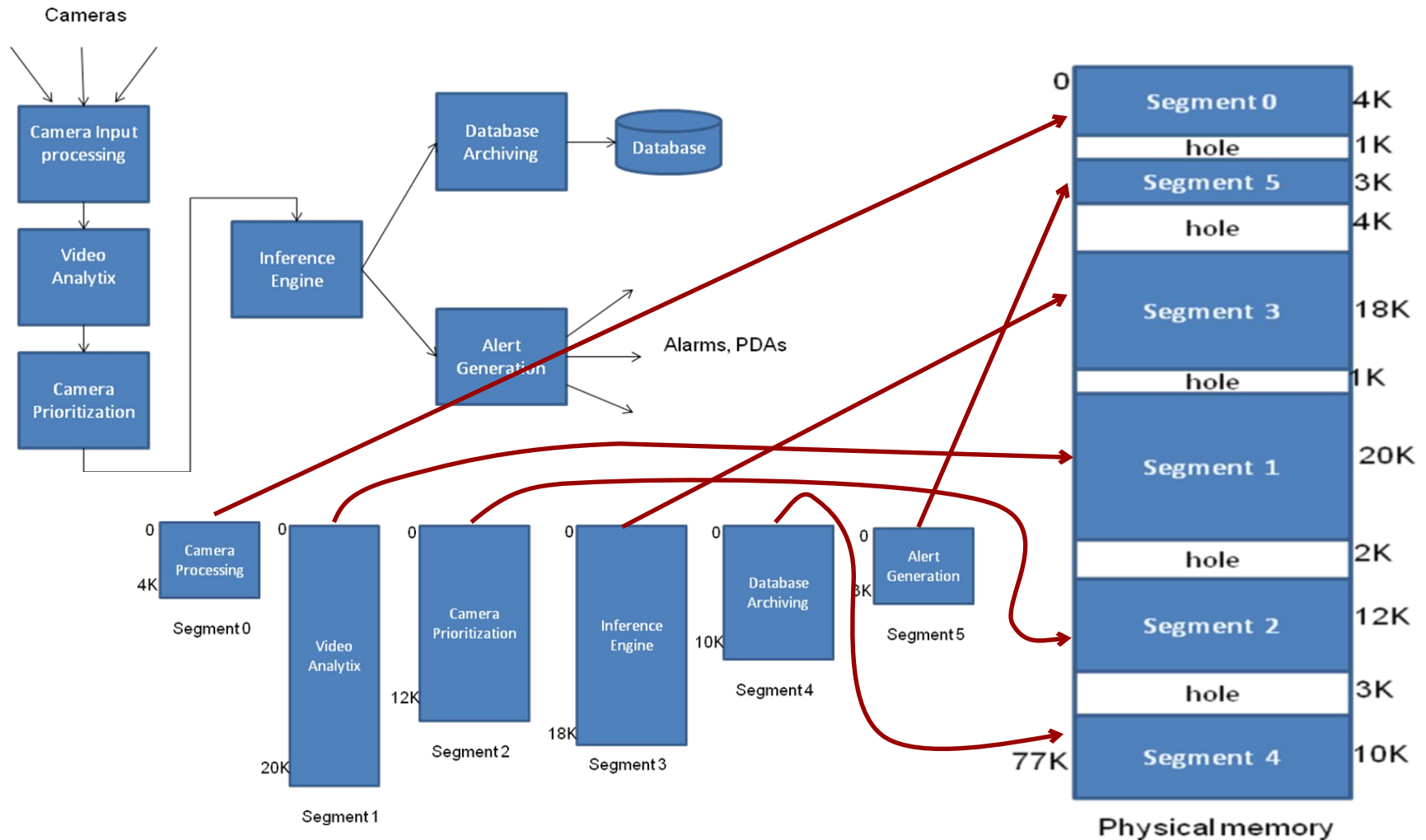
- Segmentation is a system allowing a process's memory space to be subdivided into chunks of memory each associated with some aspect of the overall program
- Segments can be different sizes
- Segmented virtual memory came first, but it's visible to the ISA, so it doesn't work as a retrofit to ISAs with contiguous memory models
- Typical segments
  - Code segments (up to 1 per function)
  - Global data
  - Heap (perhaps arrays in separate segments)
  - Stack (perhaps arrays in separate segments)

# Segmented virtual memory

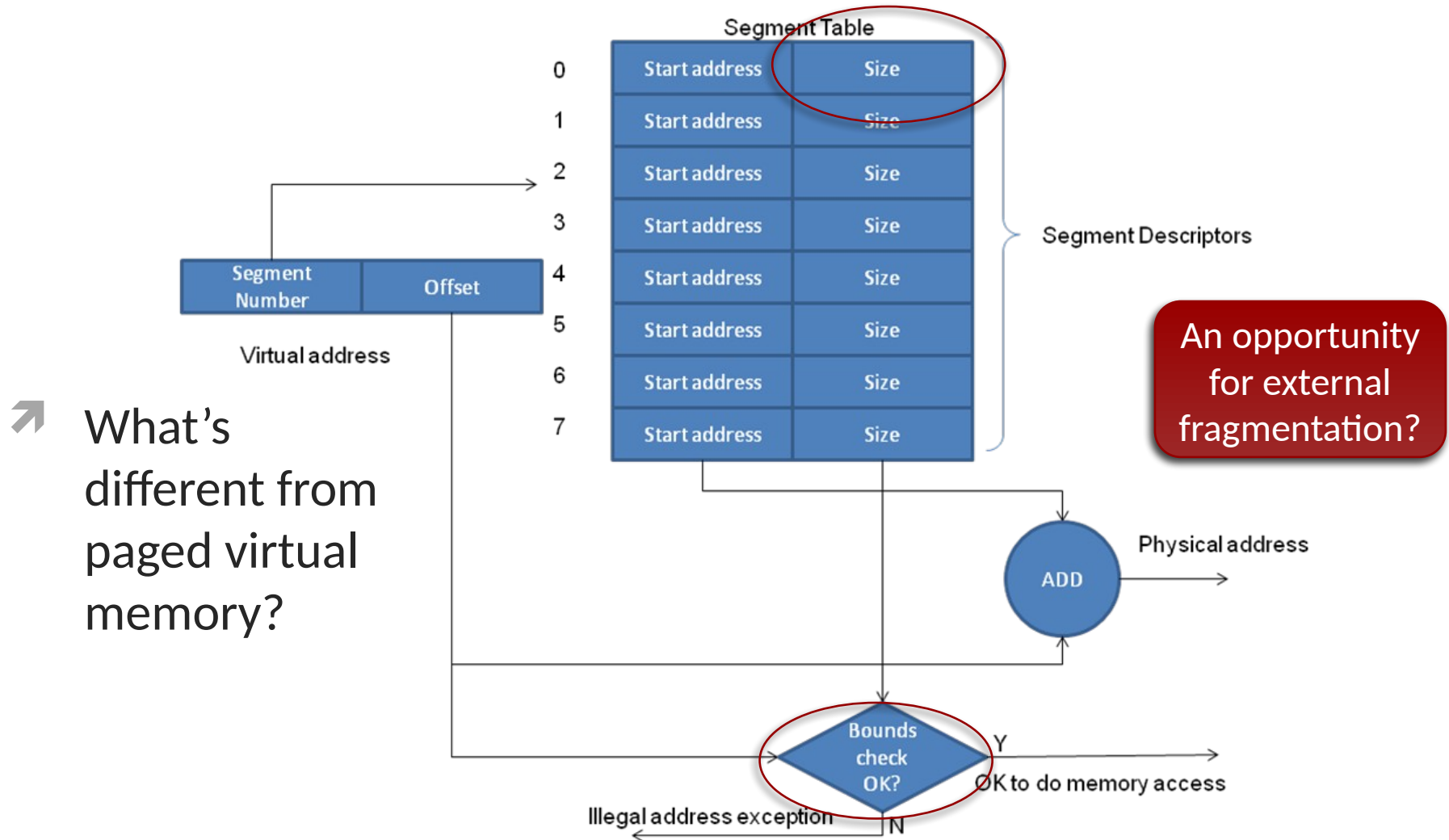
- Process address space divided up into  $n$  distinct segments based on the program organization
- Each segment has
  - A number
  - A size
- Each segment starts at its own address 0 and goes up to its size – 1.
- Segment addressing

Segment Number	Segment offset
----------------	----------------

# Segmented virtual memory example



# Hardware for segmentation

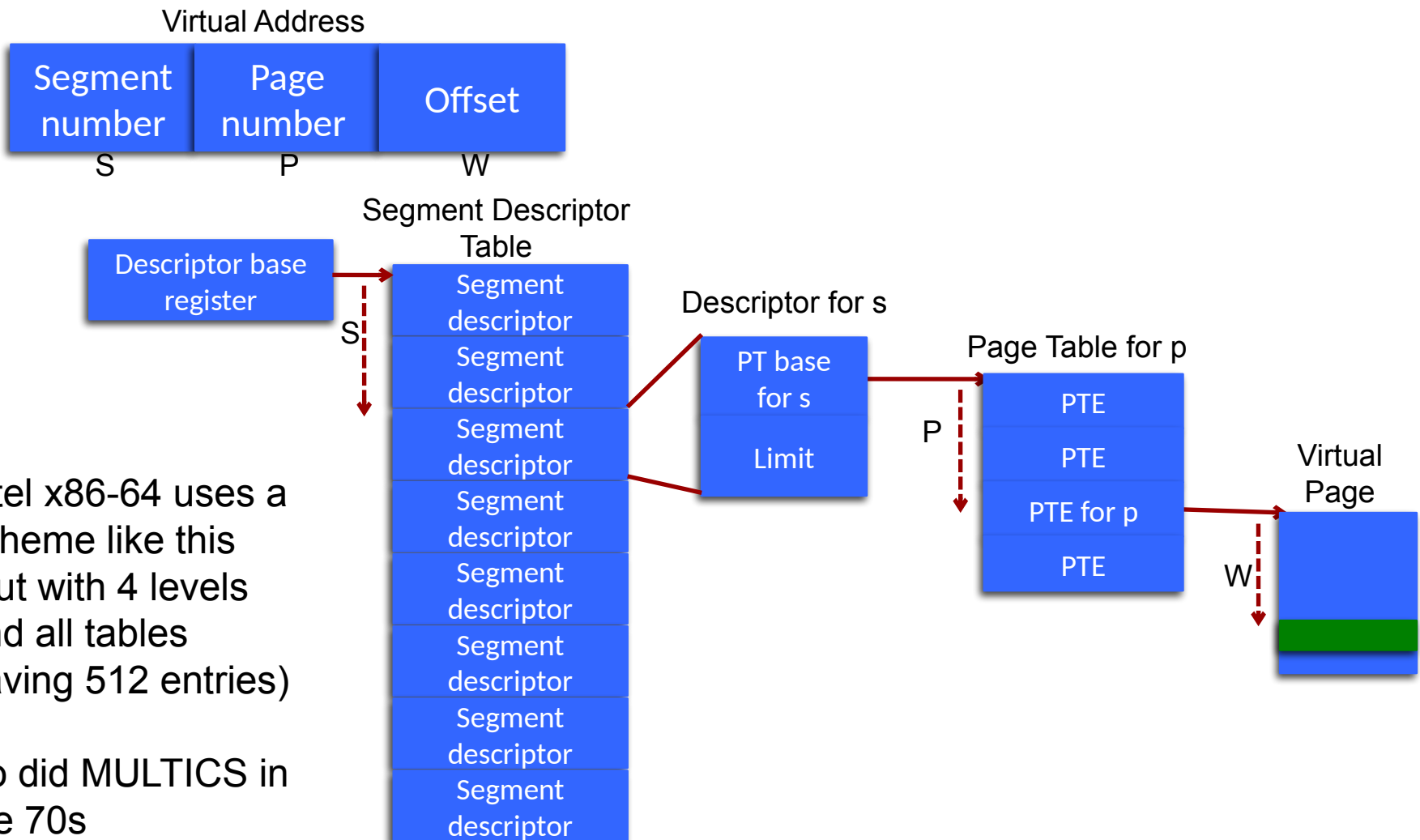




# 64-bit Virtual Addresses?

- Page tables can be BIG and we need one for each process in memory
- For example a 64-bit virtual address with 4KB pages means the VPN is  $64 - 12 = 52$  bits long.
- If a page table entry is 64 bits, how much space does one use?
- $2^{52}$  entries \*  $2^3$  bytes =  $2^{55}$  bytes =  $2^{15} * 2^{40}$  bytes  
= 32768 Terabytes(!)
- And we still need one for each process!
- Is there a problem here?

# Paged segmentation

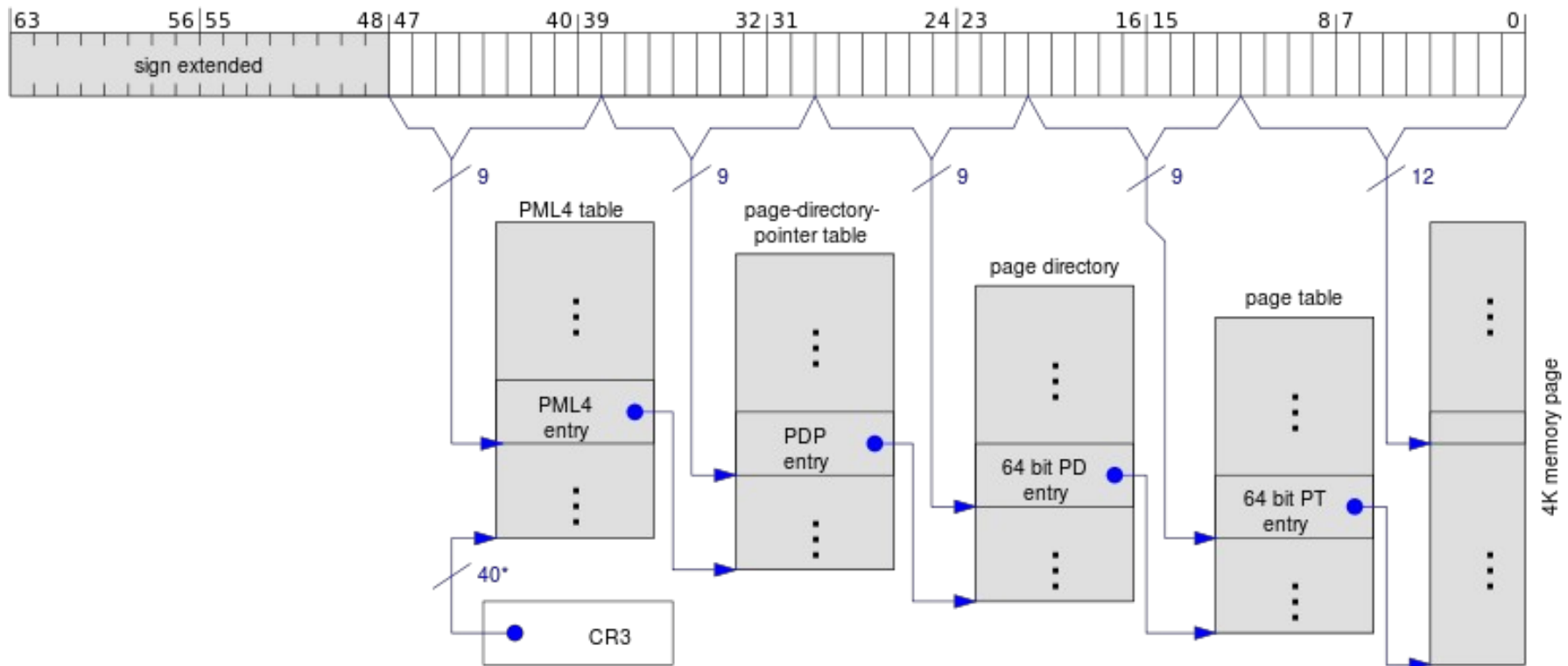


# 64-bit Physical Addresses?

- How many bytes can you address with 64 bits?
- Obviously,  $2^{64}$ , right? Which is how big?
- $1.84 \times 10^{19}$  or 1,840 followed by 15 zeros.  
That's 1,840,000 terabytes or 1.84 exabytes
- Largest Intel/AMD memory supported today is 16 TB
- Do we need all 64 physical address bits?
- 48 bits ( $2^{48}$ ) will get us 281 TB. That should be enough for current chips to become obsolete.

# Intel's X86-64 4-layer Page Tables

Linear address:



\*) 40 bits aligned to a 4-KByte boundary

# Question

Which of these is False?



- A. Variable length partitions have internal but no external fragmentation
- B. Paged memory systems have internal but no external fragmentation
- C. Segmented memory systems have external but no internal fragmentation
- D. Paged segmented memory systems have internal but no external fragmentation