Georgia Tech Computer Science

CS2200
Systems and Networks
Spring 2022

Lecture 15:
Virtual Memory

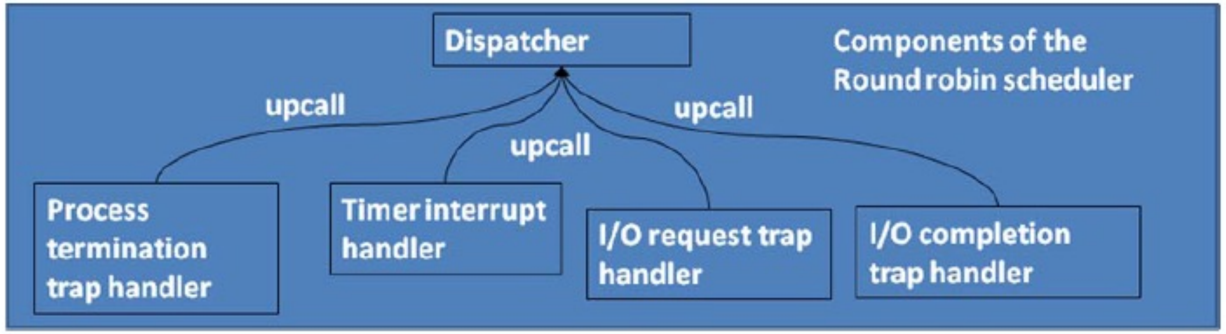
Alexandros (Alex) Daglis
School of Computer Science
Georgia Institute of Technology
adaglis@gatech.edu

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

Today's Agenda

- Wrapping up Chapter 6 – Scheduling
- Starting Part II – Memory System
  - Chapters 7,8

Who does what in the OS


Dispatcher
Components of the Round robin scheduler
upcall
upcall
upcall
upcall
Process termination trap handler
Timer interrupt handler
I/O request trap handler
I/O completion trap handler

```
Scheduler:
            run scheduling algorithm
            get head of ready queue;
            set timer;
dispatch {  save context in PCB;
            restore context from PCB at head of ready list;
            return

Timer interrupt handler:
            mark PCB as timer expired;
            call the scheduler & then return from interrupt;

I/O request trap:
            initiate I/O operation;
            move PCB to I/O queue and mark as waiting;
            call the scheduler & then return from interrupt;

I/O completion interrupt handler:
            mark I/O buffer completed;
            move PCB of I/O completed process to ready queue;
            call the scheduler & then return from interrupt;

Process termination trap handler:
            mark PCB as Halted and freeable;
            call the scheduler & then return from interrupt;
```

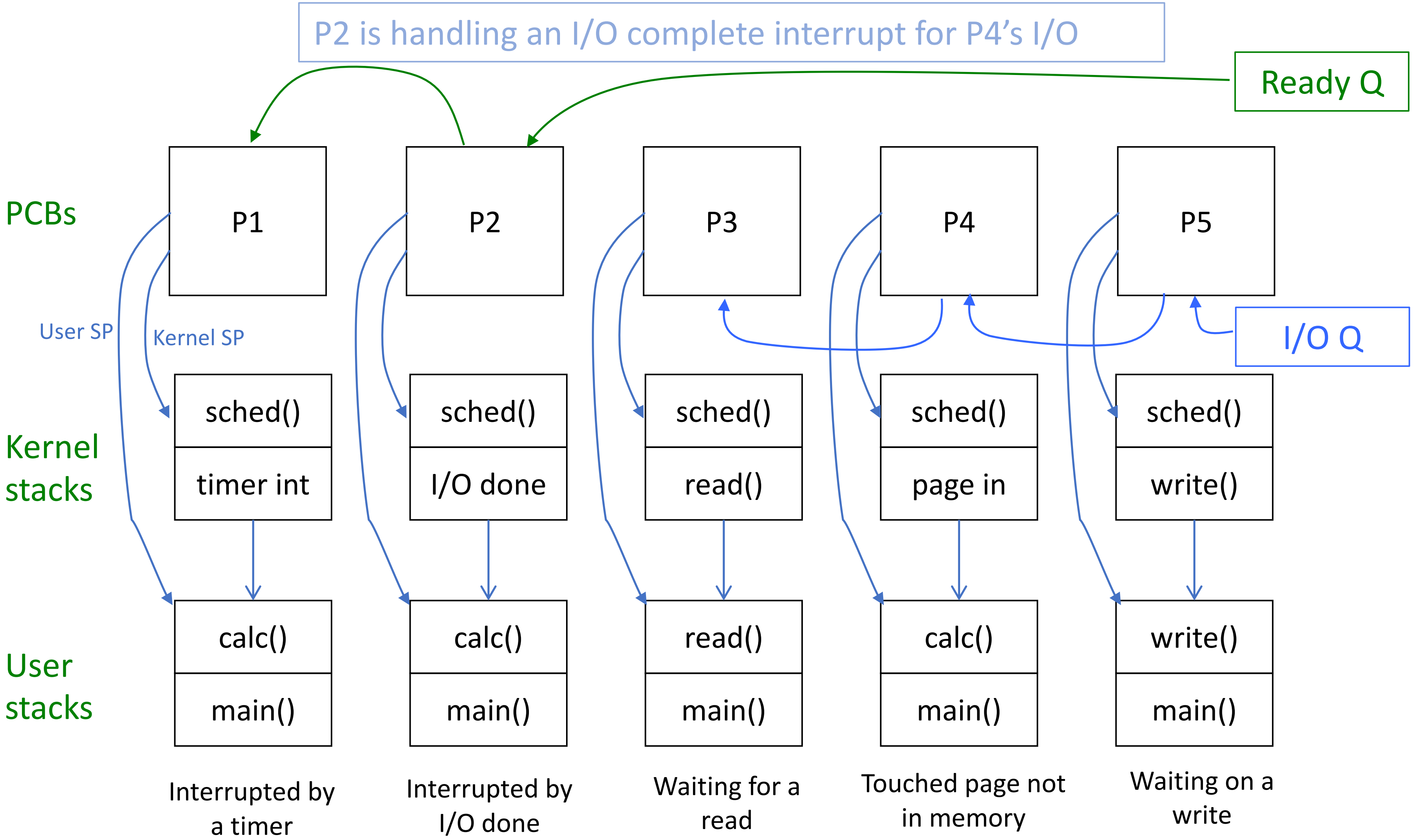
Our Example Program

Assume our test program is written in the following way

```
main() {
    while (more data) {
      read(); // Read case in from a file
      calc(); // Do a complicated calculation
      write();// Write the results to a file
    }
}
```

Now let's run five copies of it…

Process structures



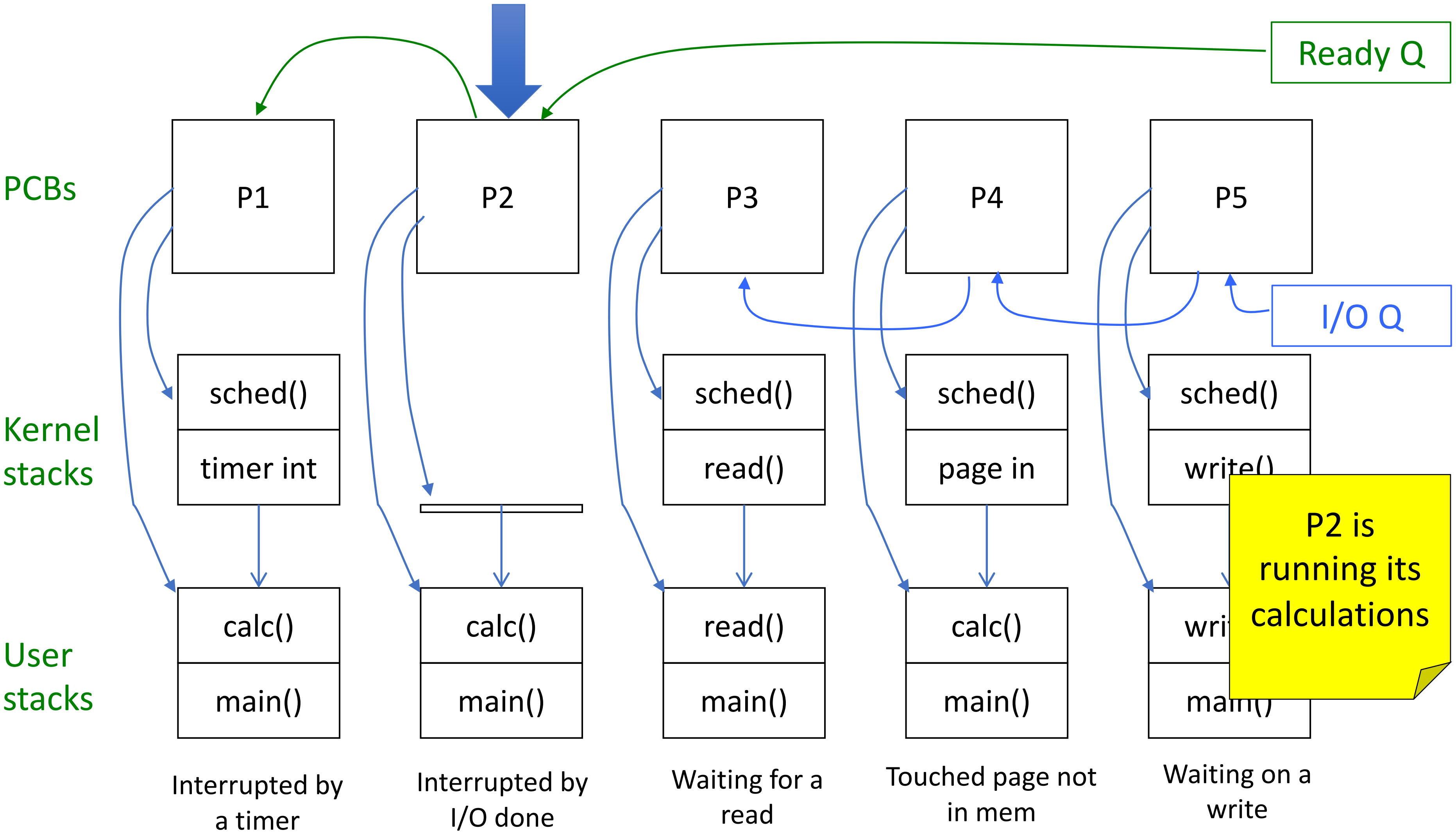
An example using the previous diagram

- Last process to run was P2
- P2's interrupt handler marked P4's I/O complete since that is the I/O that was pending on the interrupting device
- That puts P4 back on the ready list (and off the I/O list)
- P2's interrupt handler calls the scheduler
  - The winner is P4
    - Remember: P4's return to the ready list was caused by the I/O complete interrupt that P2 took
  - Leave P2 on the ready list & adjust its quantum to reflect the CPU time it's used
  - Save the processor state in P2's PCB
  - Complete the context switch to P4 by loading state from P4's PCB
  - Return using the currently loaded state (i.e. return to P4)

Let’s do the example again

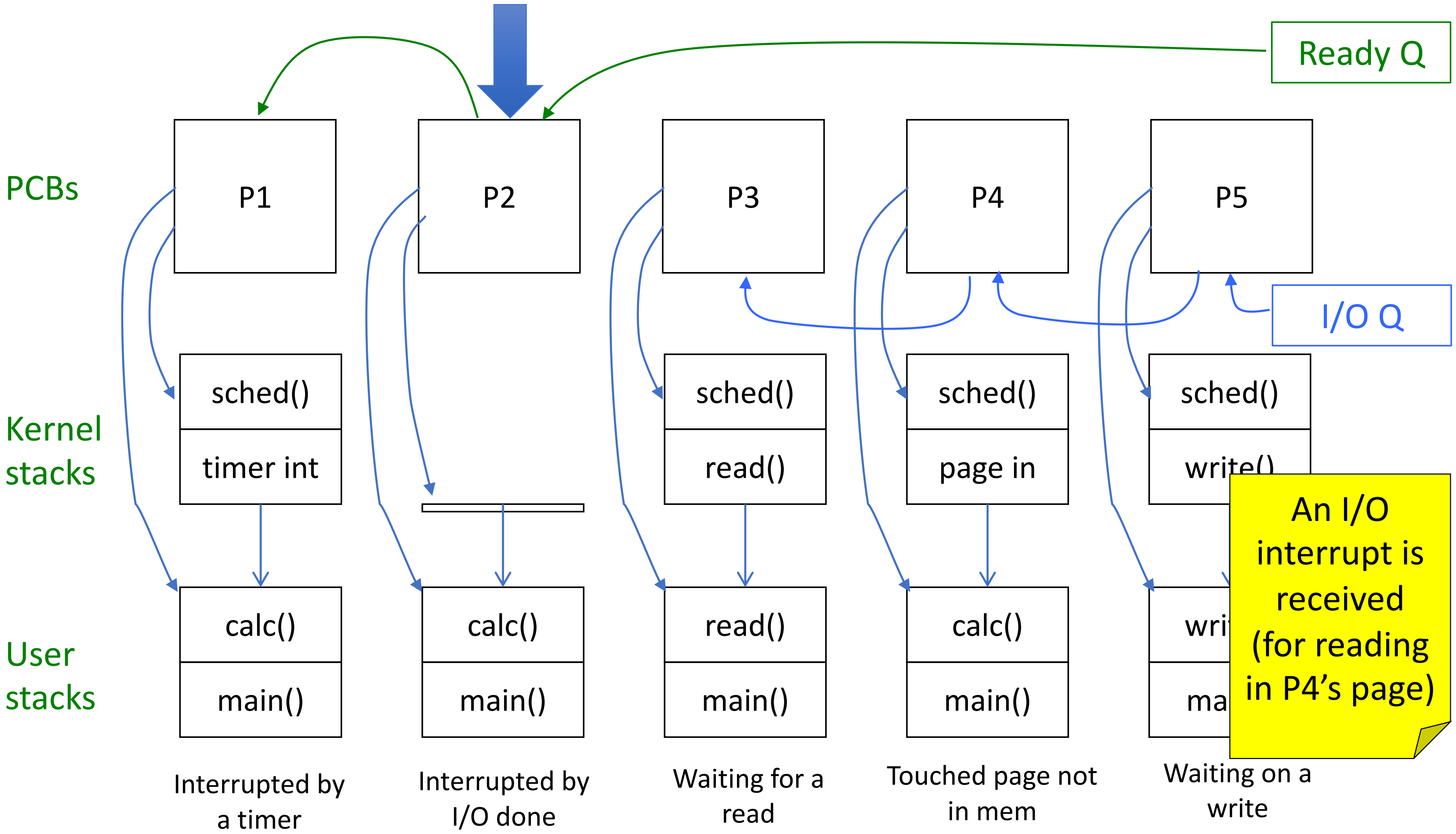
- Foreshadowing chapter 7:
  - With demand paging memory management, when a user program references a part (page) of the program that’s not resident in physical memory,
    - The hardware causes a page-fault trap
    - The operating system treats a page-fault trap as a request to read in the faulting page from disk,
    - Then change the memory map to reflect the newly-resident page,
    - and reissue the instruction that page-faulted
- Pay close attention to the slight-of-hand that switches us from P2 to P4
- We start the example with P2 running its calculations in user state

Process example in detail!



P2 is running its calculations

Process example



Ready Q

PCBs

P1

P2

P3

P4

P5

I/O Q

Kernel stacks

sched()

timer int

sched()

read()

sched()

page in

sched()

write()

User stacks

calc()

main()

calc()

main()

read()

main()

calc()

main()

wri

ma

Interrupted by a timer

Interrupted by I/O done

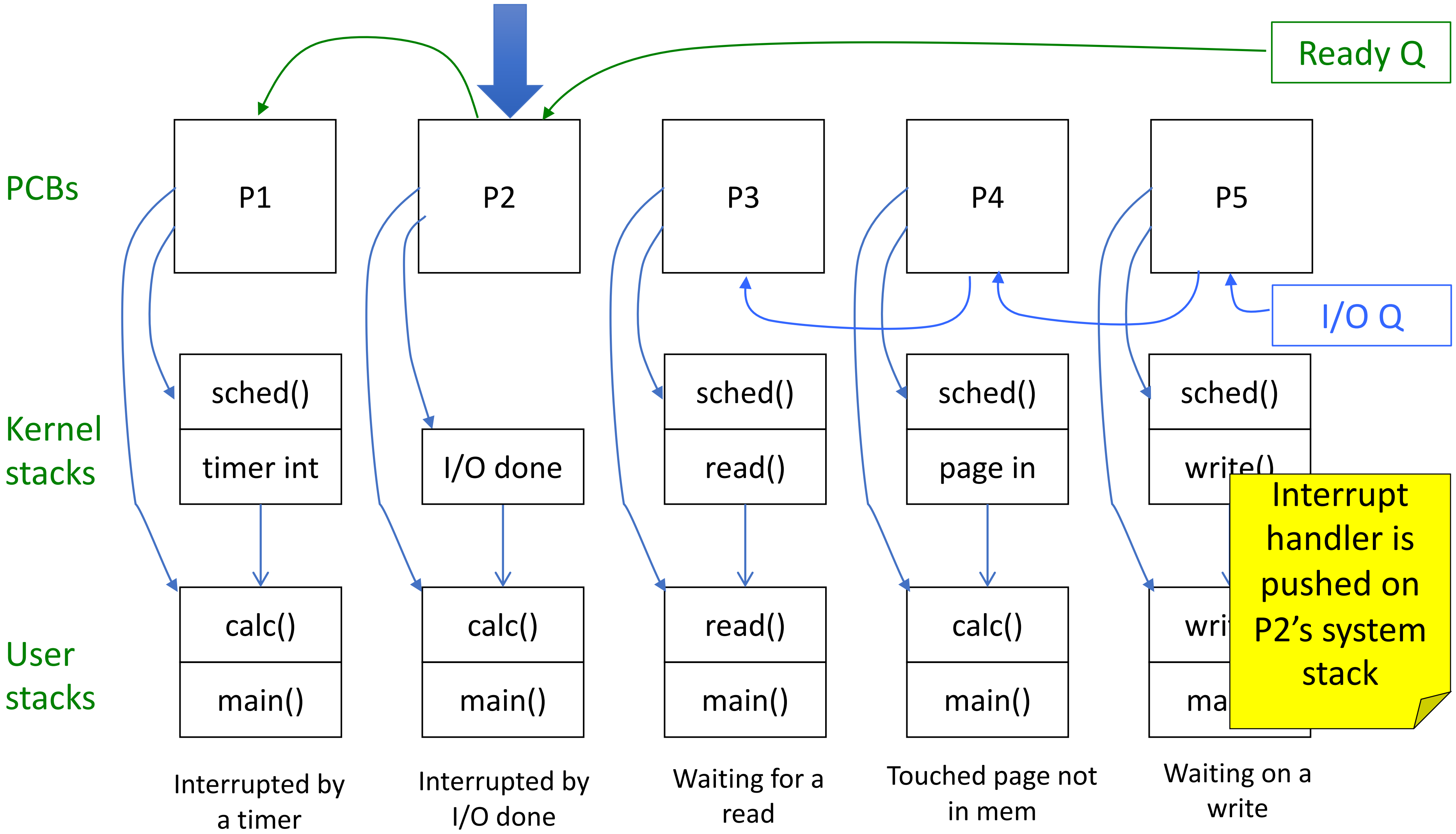
Waiting for a read

Touched page not in mem

Waiting on a write

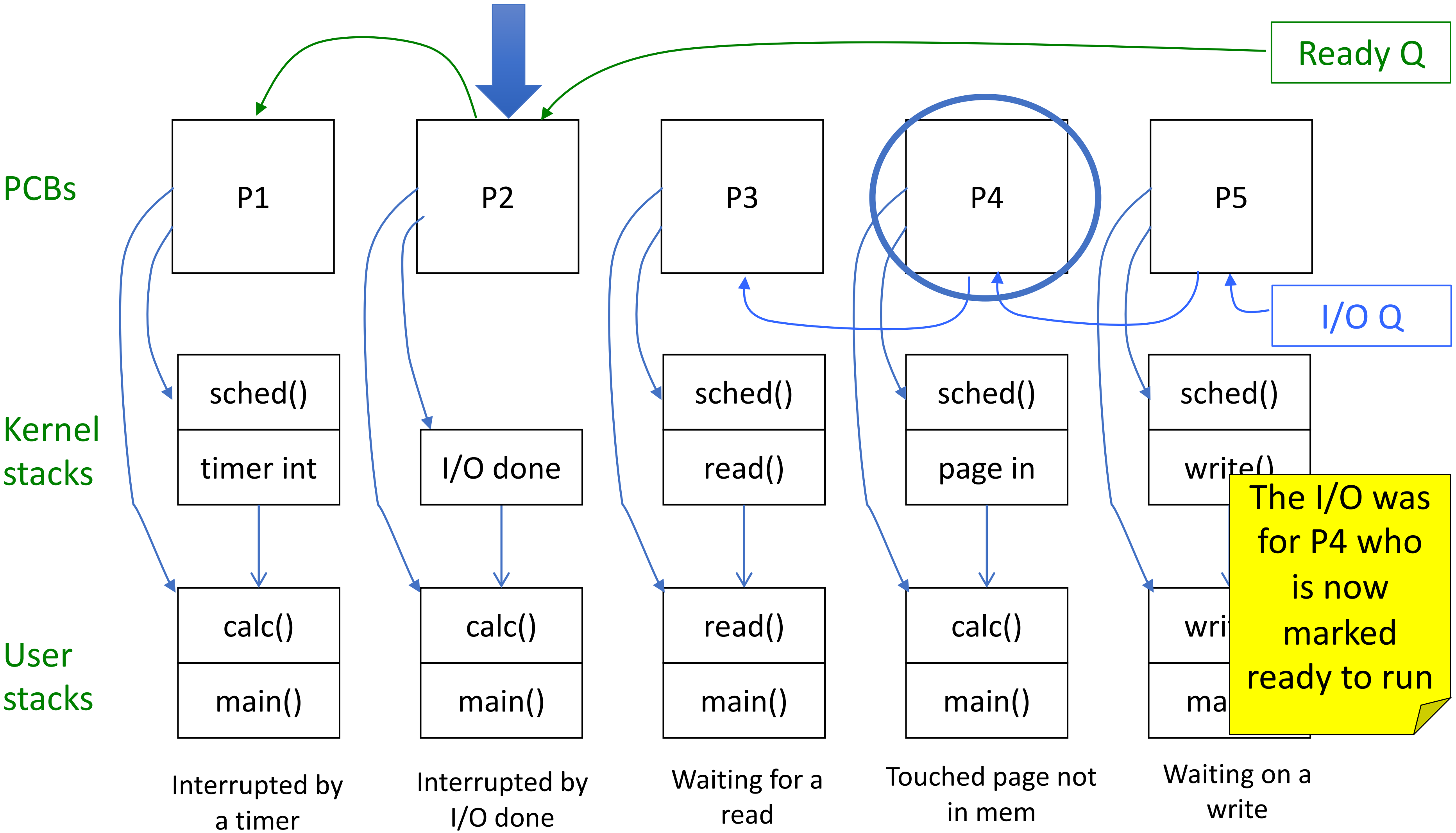
An I/O interrupt is received (for reading in P4's page)

Process example



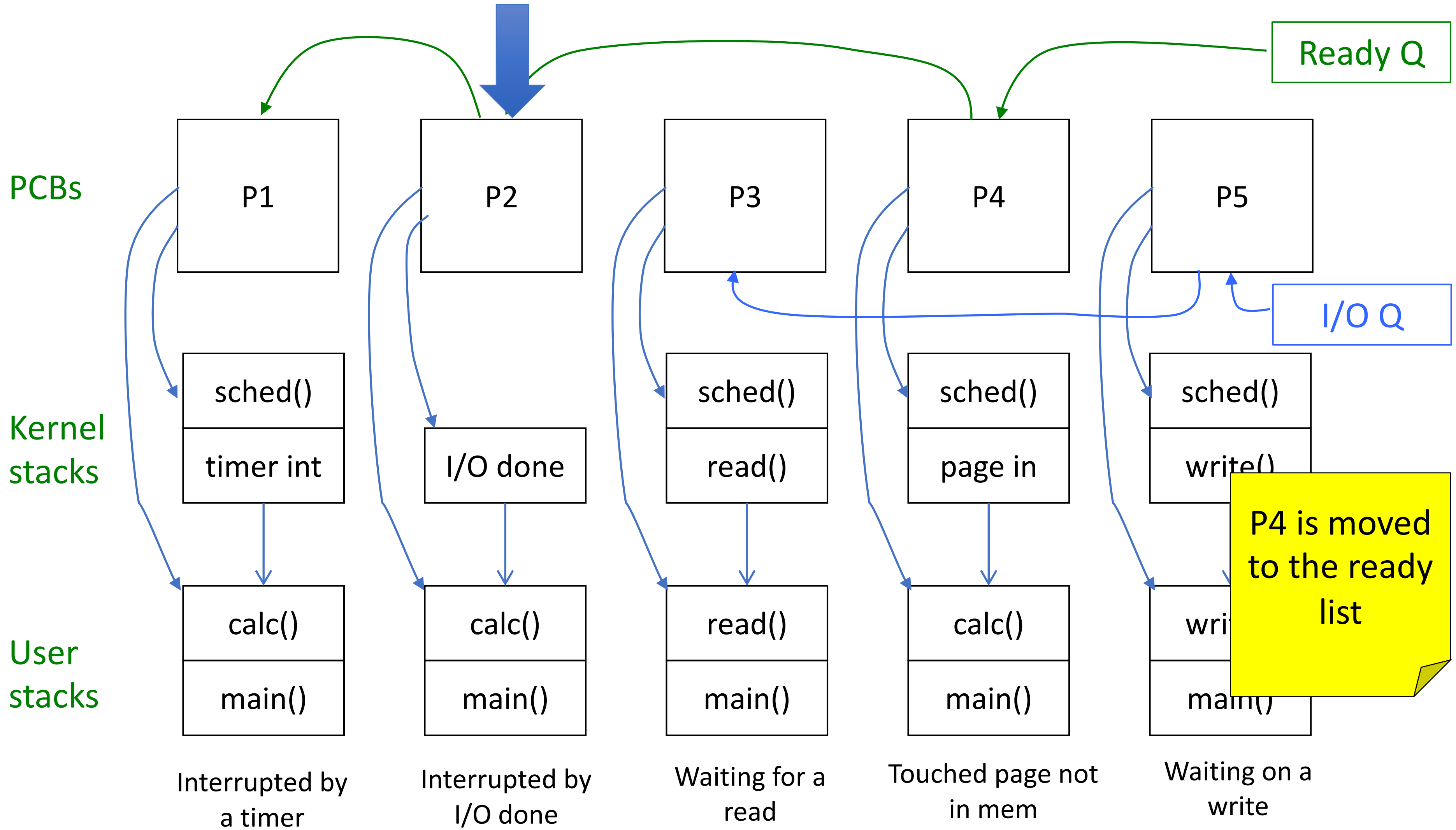
Interrupt handler is pushed on P2's system stack

Process example



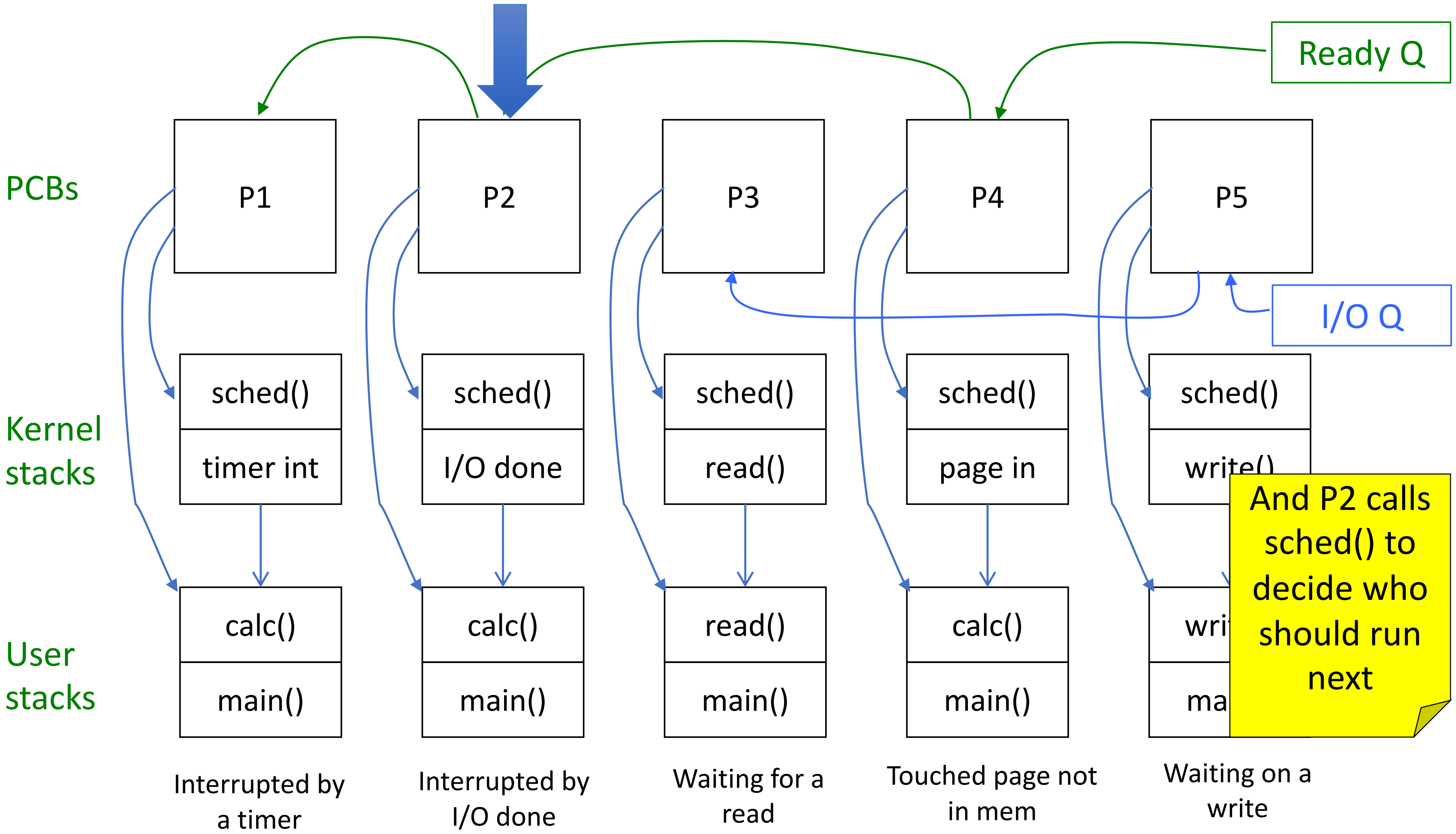
The I/O was for P4 who is now marked ready to run

Process example



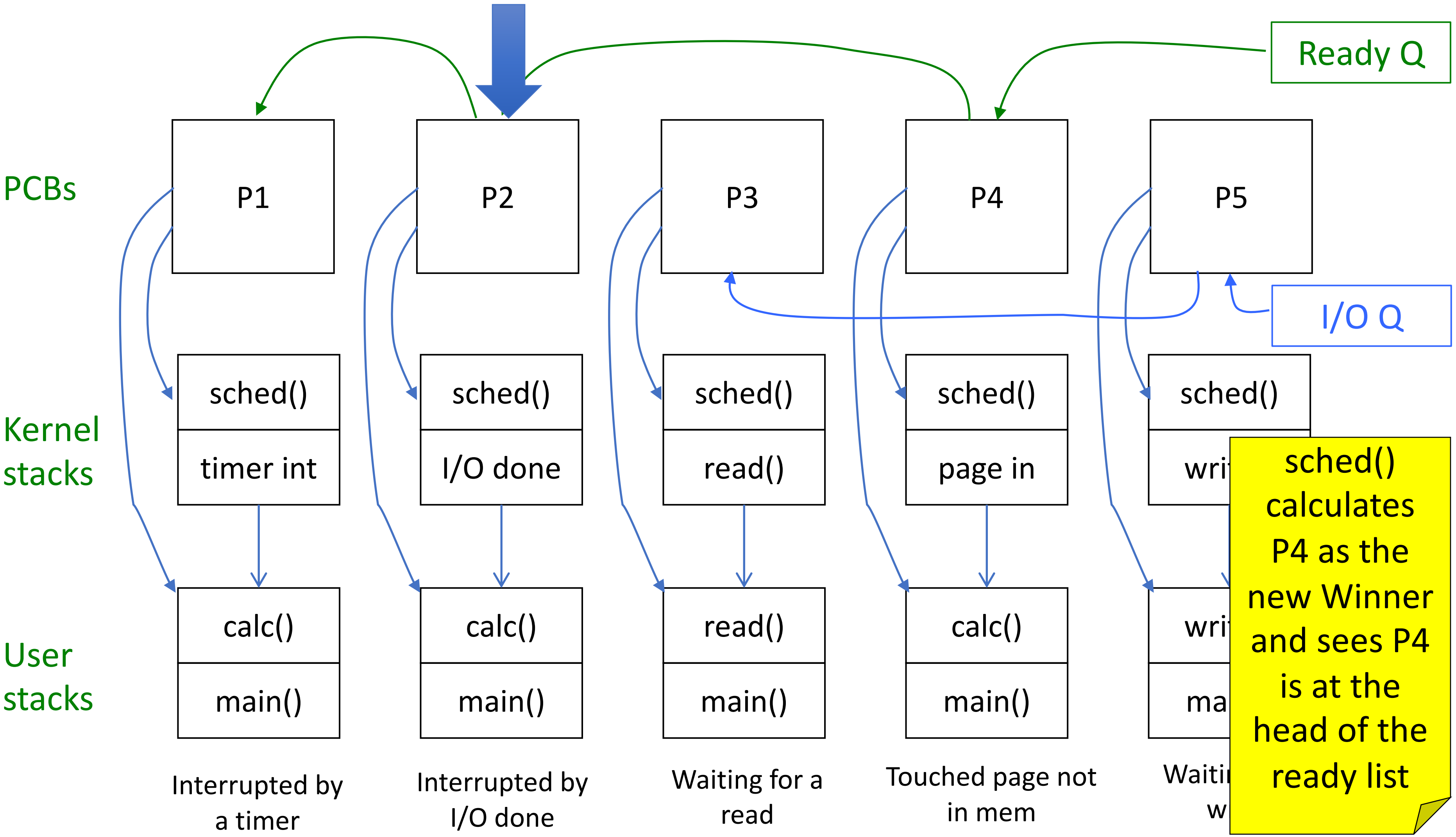
P4 is moved to the ready list

Process example



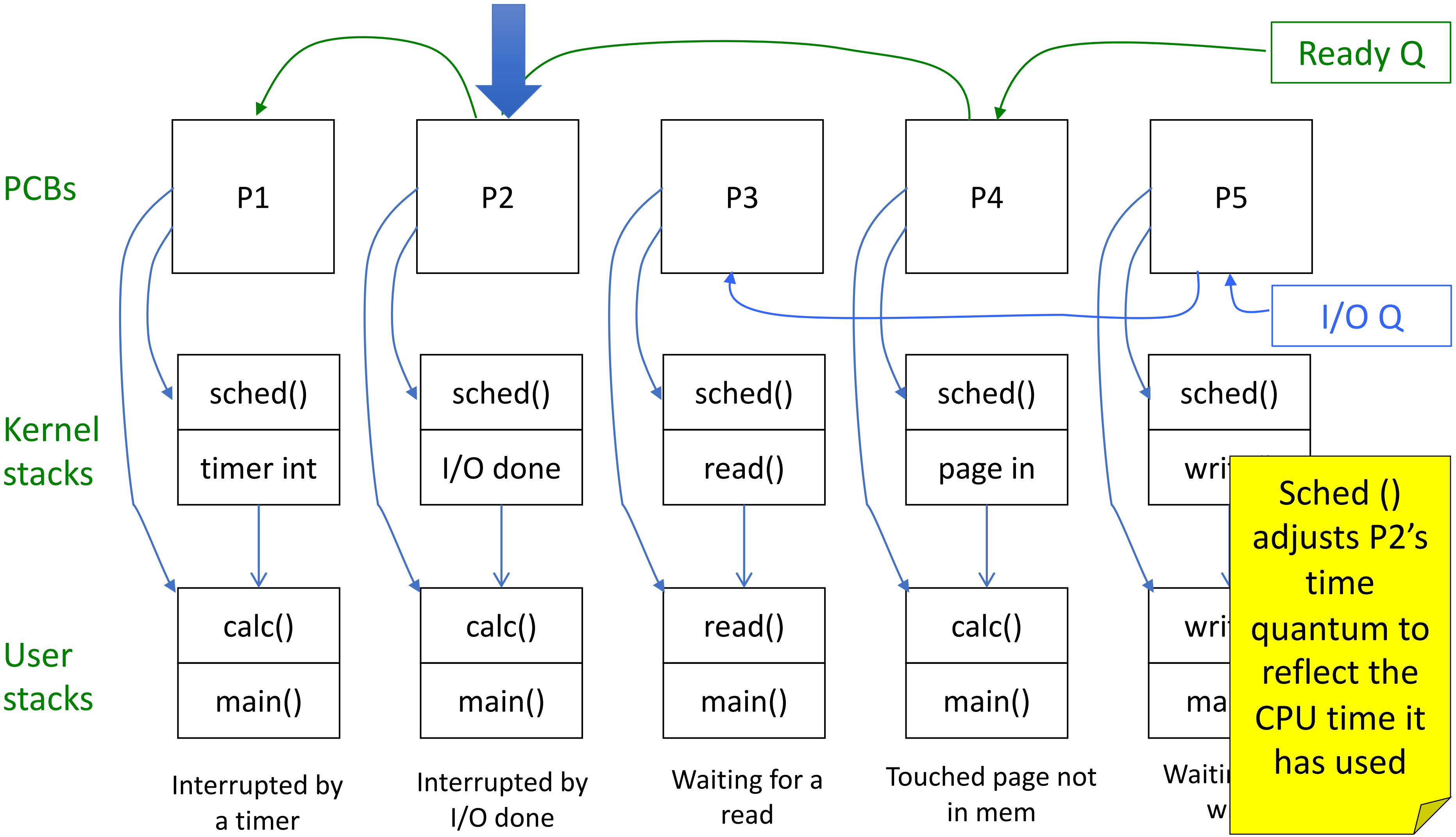
And P2 calls sched() to decide who should run next

Process example



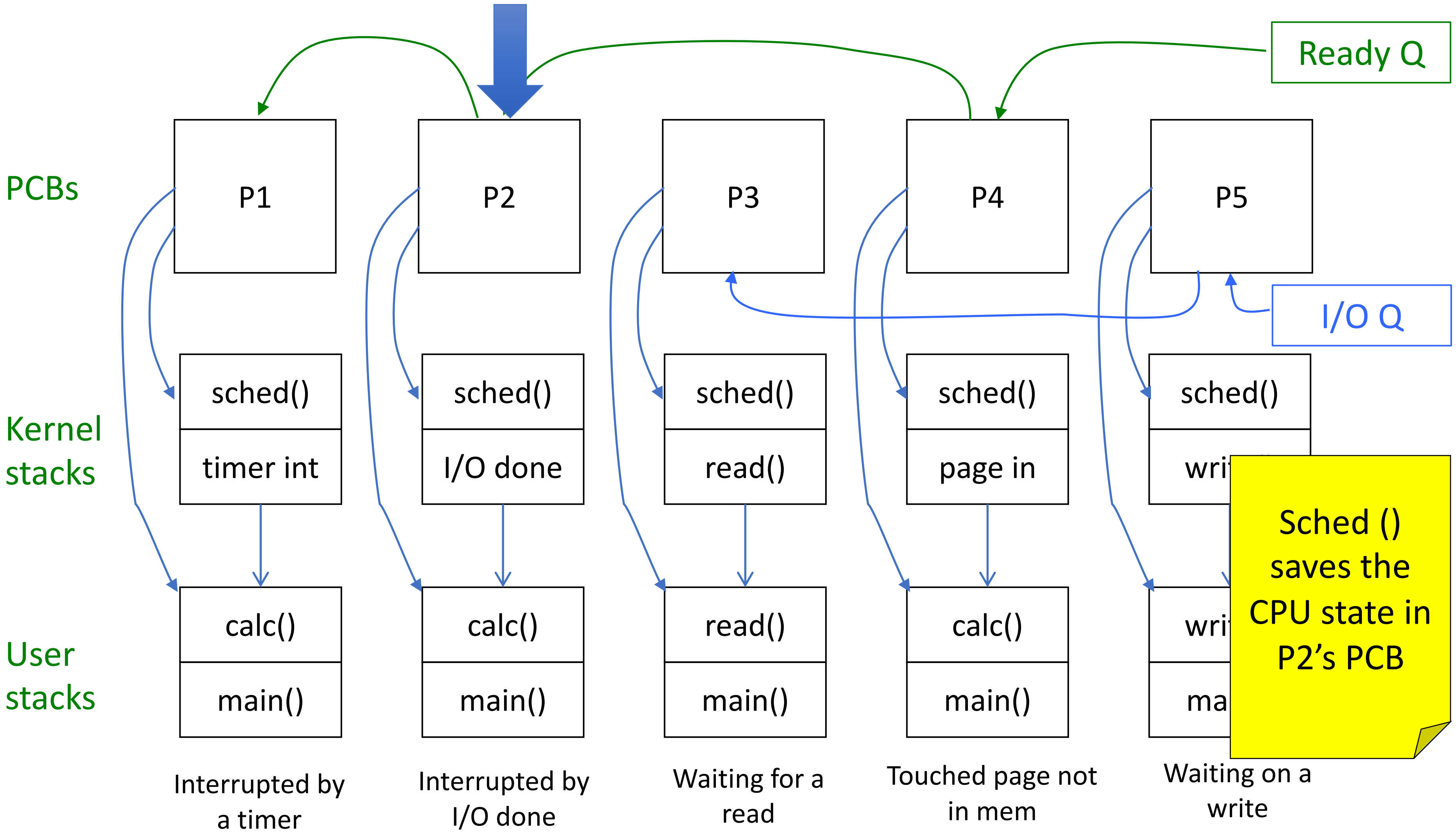
sched() calculates P4 as the new Winner and sees P4 is at the head of the ready list

Process example



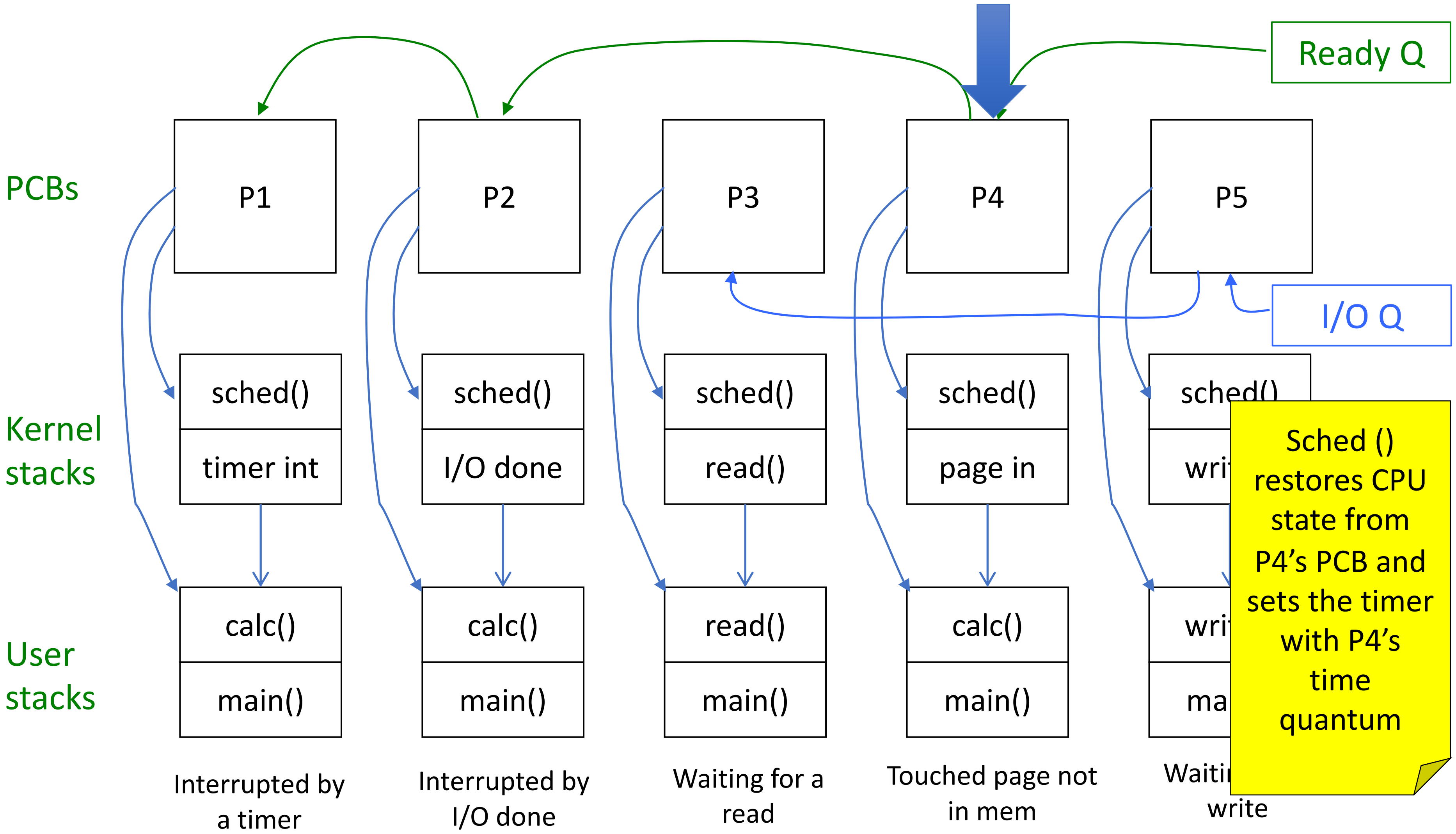
Sched () adjusts P2's time quantum to reflect the CPU time it has used

Process example



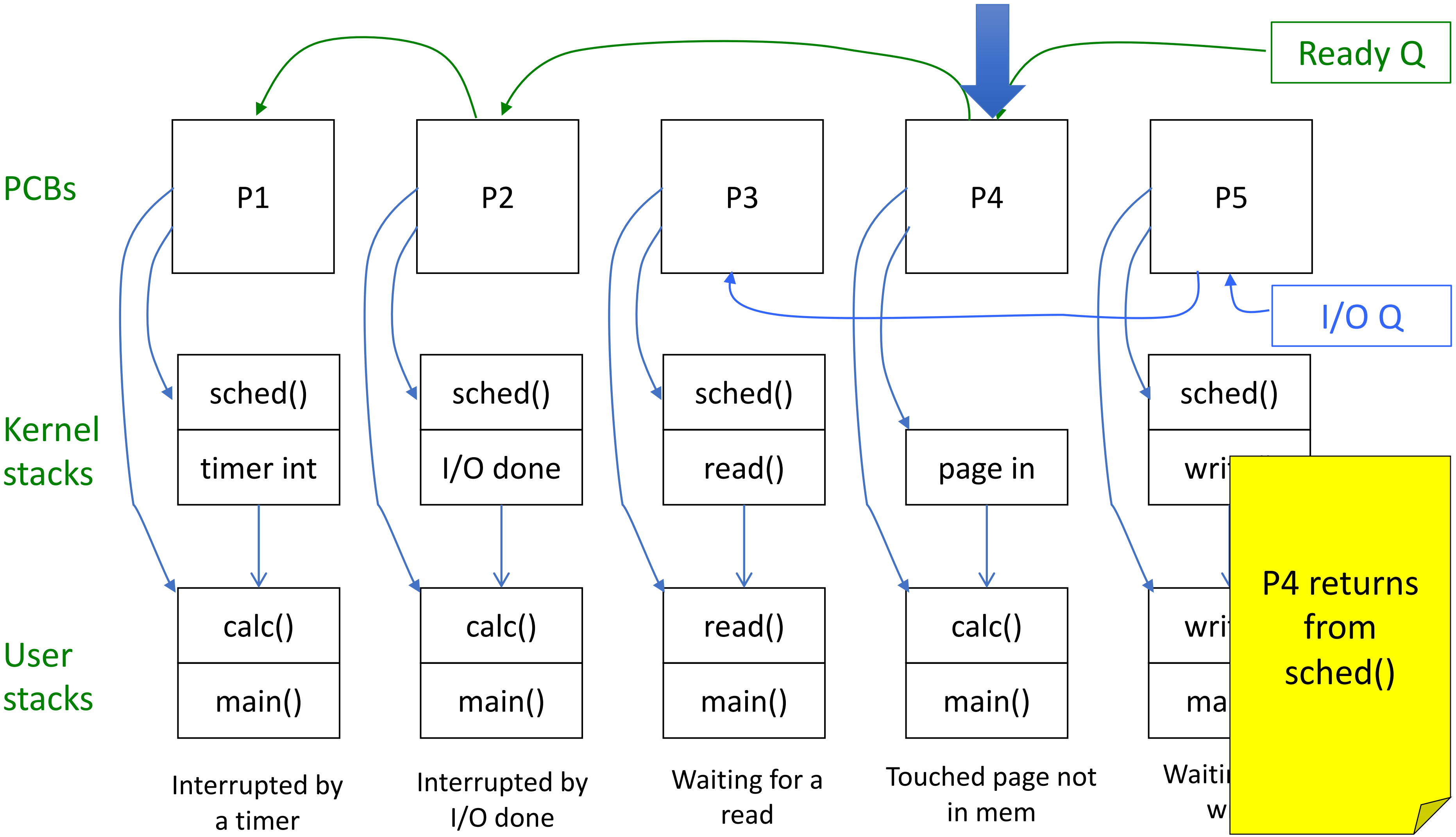
Sched () saves the CPU state in P2's PCB

Process example



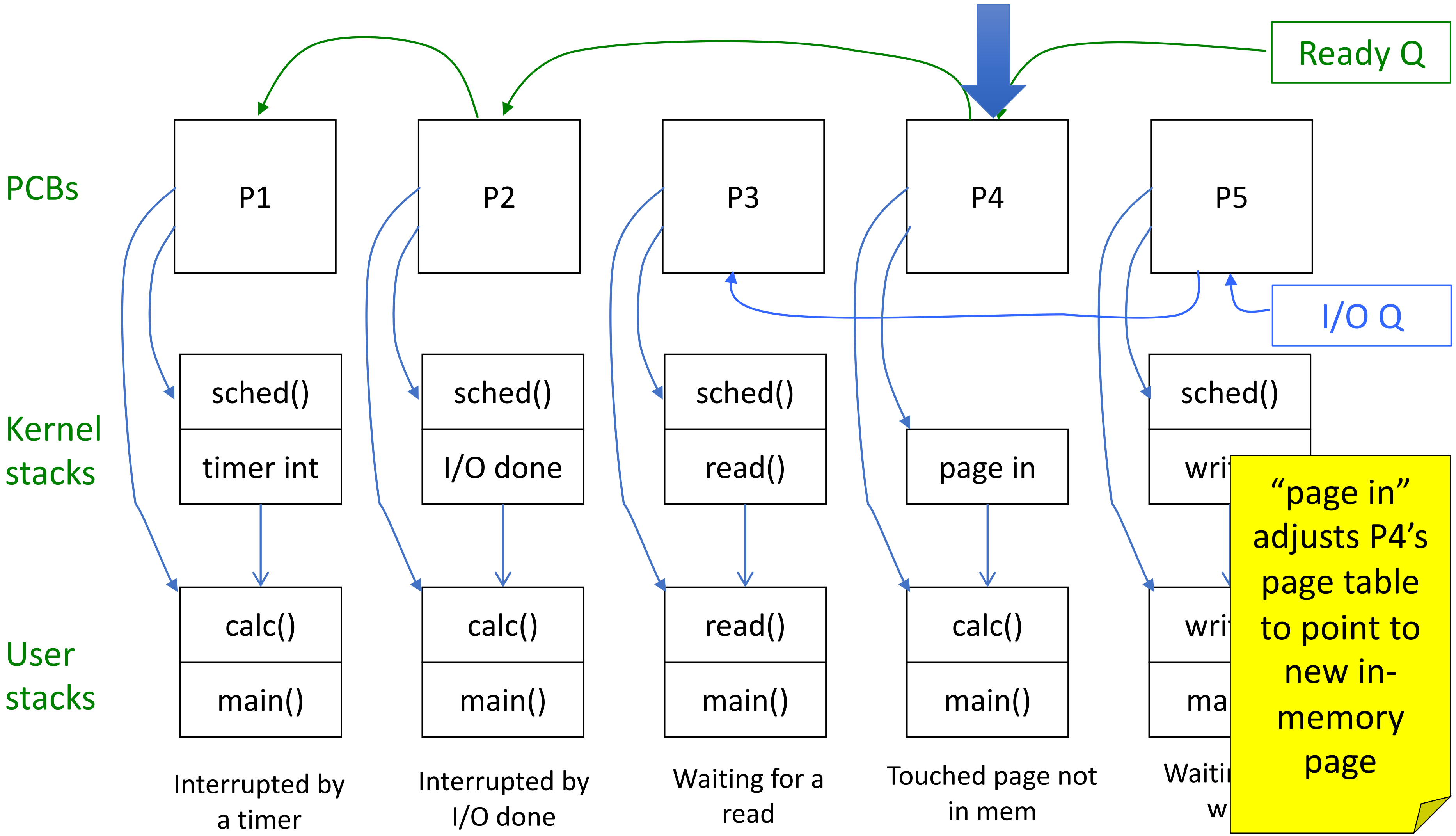
Sched () restores CPU state from P4's PCB and sets the timer with P4's time quantum

Process example



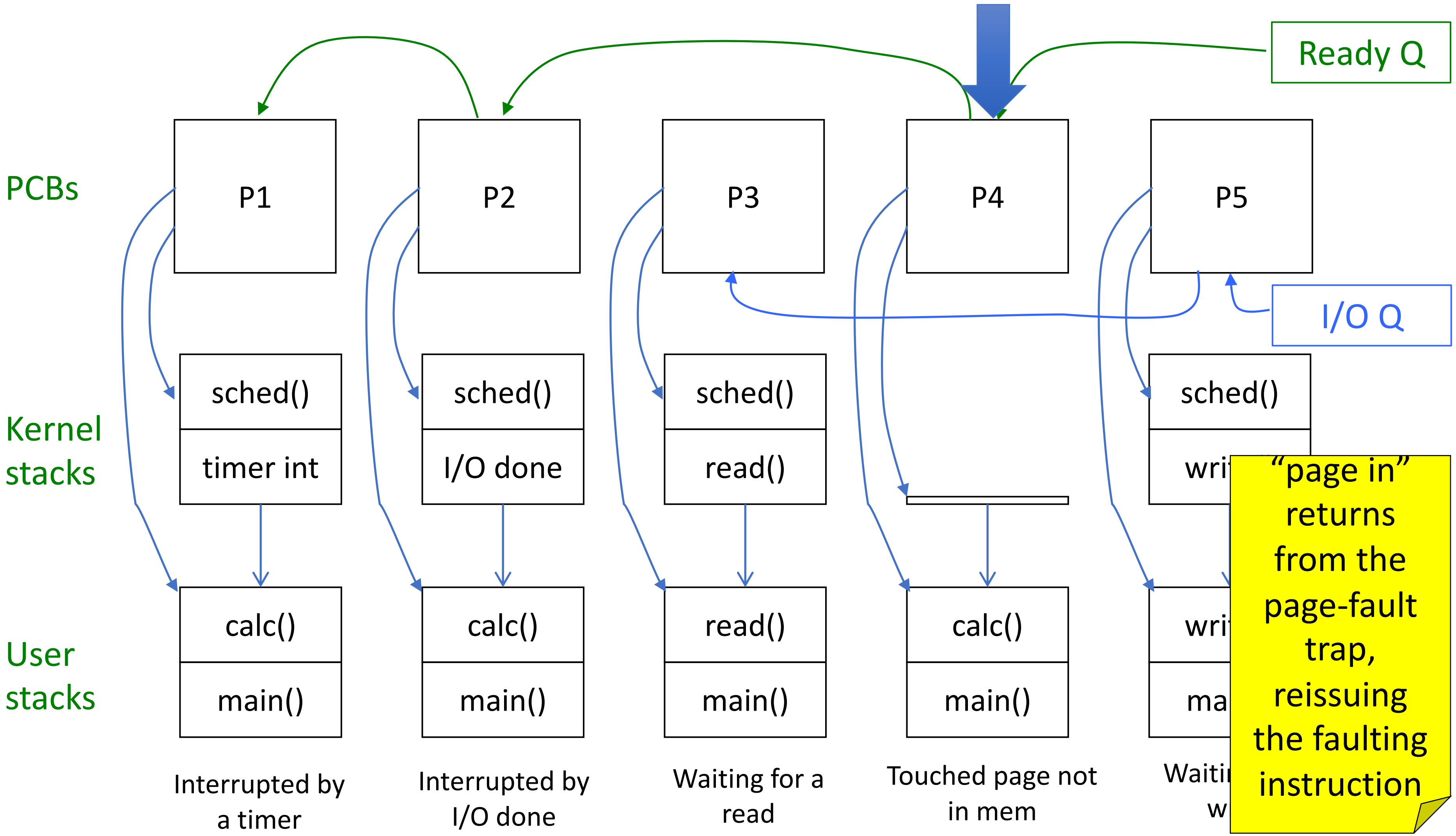
P4 returns from sched()

Process example



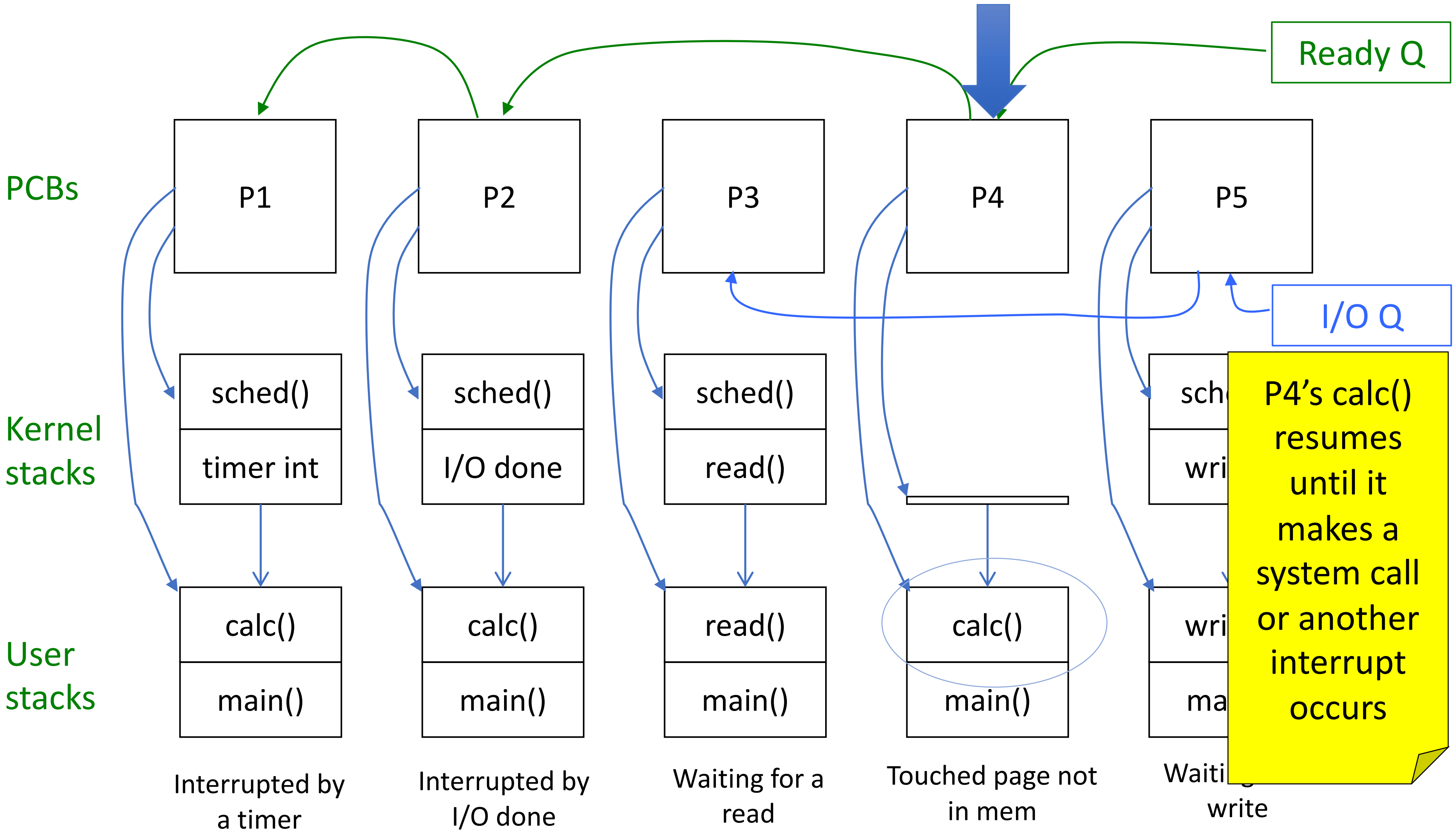
"page in" adjusts P4's page table to point to new in-memory page

Process example



"page in" returns from the page-fault trap, reissuing the faulting instruction

Process example



P4's calc() resumes until it makes a system call or another interrupt occurs

Session ID: dag2200

On context switch, the scheduler saves the volatile state of the current process in

A. The system stack

B. The PCB for that process

C. The user stack

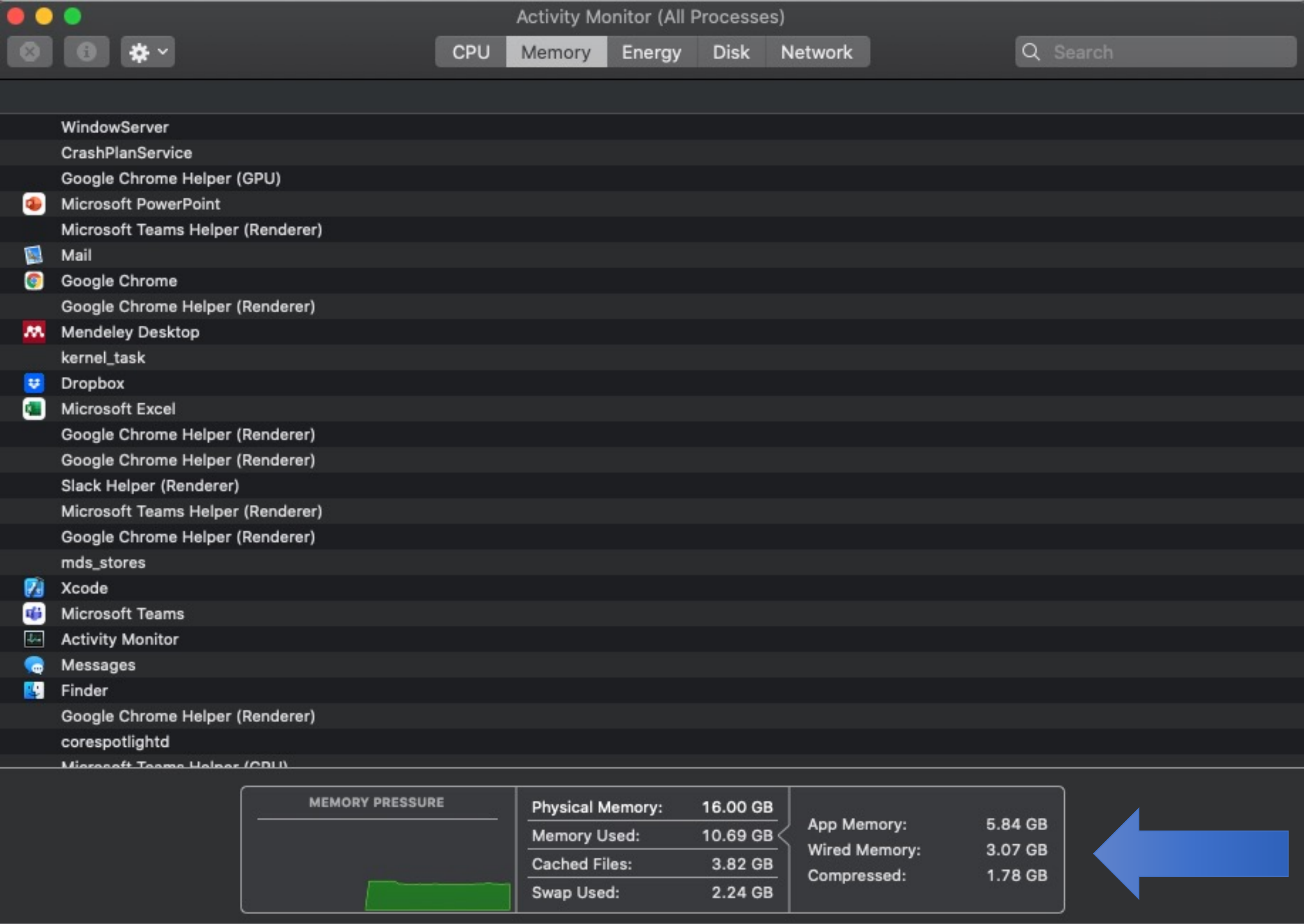
D. The heap space of the process

Memory Management

So far

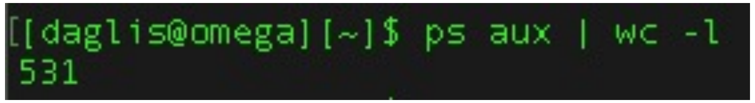
Software	Hardware
High level language + compiler + linker	ISA, addressing modes, stack, registers
Program discontinuities (INT handler in OS)	Interrupt support, processor implementation (simple + pipeline)
OS process scheduler + loader	Process as an abstraction, context switching
OS memory management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

A "small" laptop's workload



WindowServer
CrashPlanService
Google Chrome Helper (GPU)
Microsoft PowerPoint
Microsoft Teams Helper (Renderer)
Mail
Google Chrome
Google Chrome Helper (Renderer)
Mendeley Desktop
kernel_task
Dropbox
Microsoft Excel
Google Chrome Helper (Renderer)
Google Chrome Helper (Renderer)
Slack Helper (Renderer)
Microsoft Teams Helper (Renderer)
Google Chrome Helper (Renderer)
mds_stores
Xcode
Microsoft Teams
Activity Monitor
Messages
Finder
Google Chrome Helper (Renderer)
corespotlightd
Microsoft Teams Helper (GPU)

Physical Memory:	16.00 GB		
Memory Used:	10.69 GB	App Memory:	5.84 GB
Cached Files:	3.82 GB	Wired Memory:	3.07 GB
Swap Used:	2.24 GB	Compressed:	1.78 GB



```
[[daglis@omega][~]$ ps aux | wc -l
 531
```

531 processes!

10.69GB of 16GB memory in use

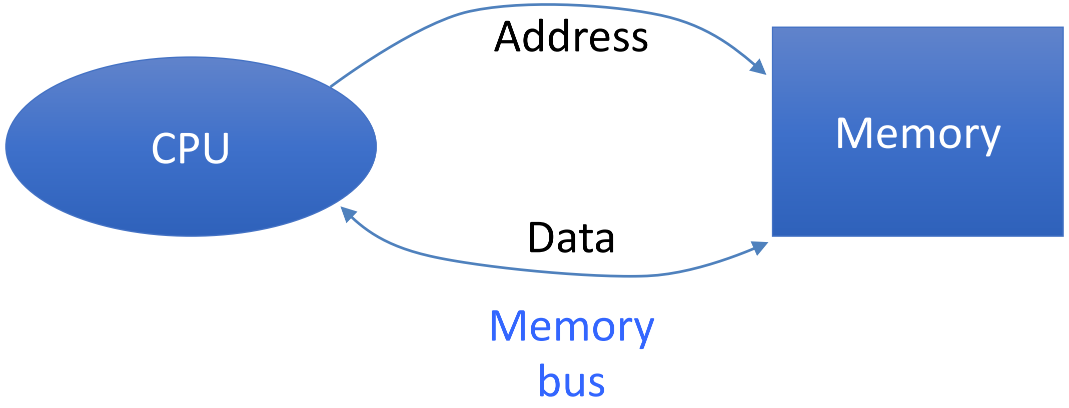
Goals of memory management

What functionalities do we want to provide?

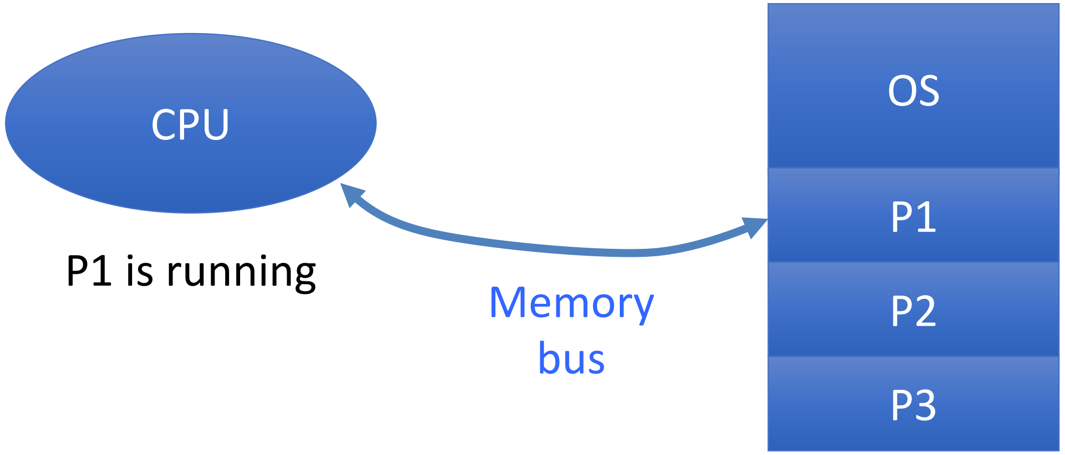
- Improved resource utilization
- Independence and protection
- Liberation from resource limitations
- Sharing of memory by concurrent processes

- So far there's no hardware in the LC-2200 to "manage" memory

Nothing in LC-2200 to manage memory

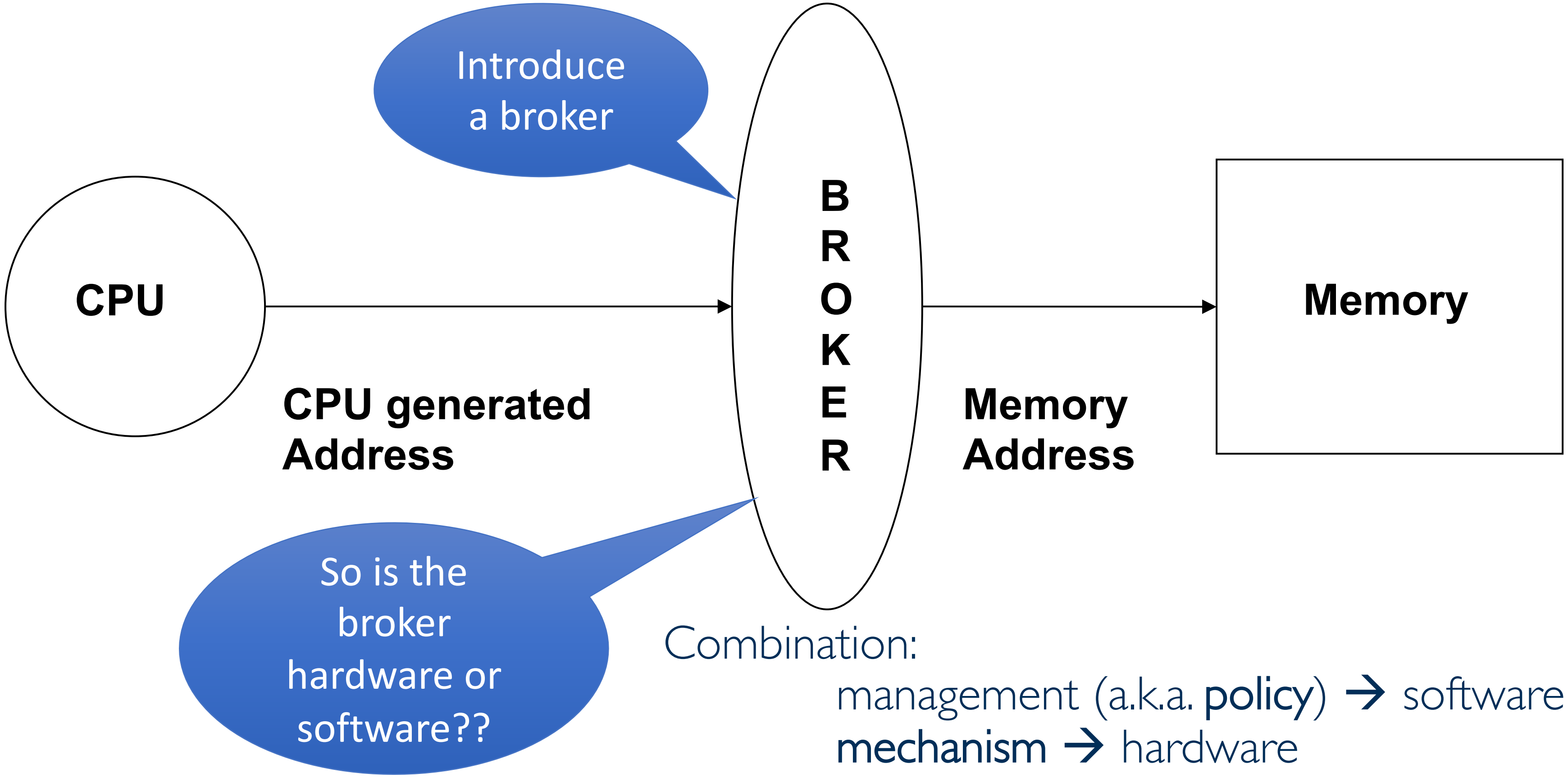


What’s missing in LC-2200?



- 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…

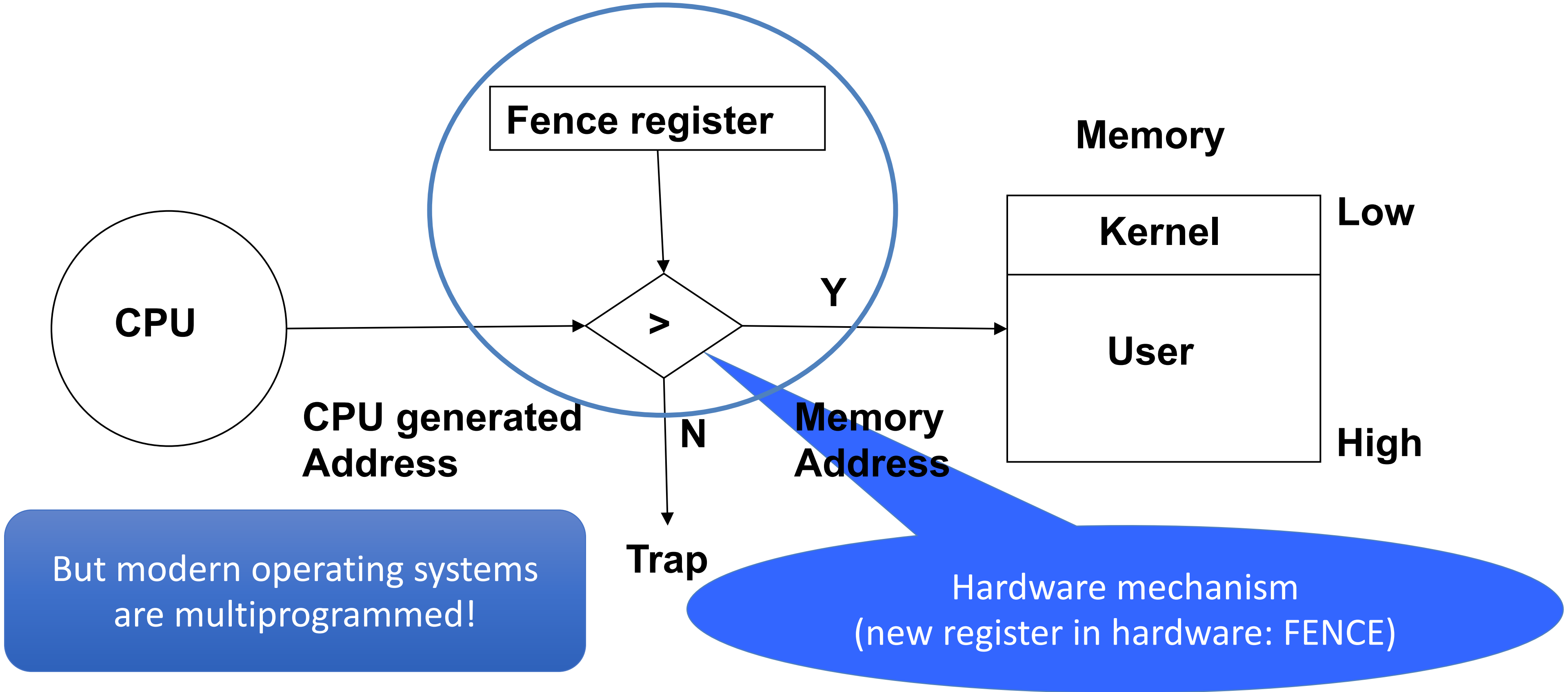


Combination:

management (a.k.a. **policy**) → software
mechanism → hardware

Simple management

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

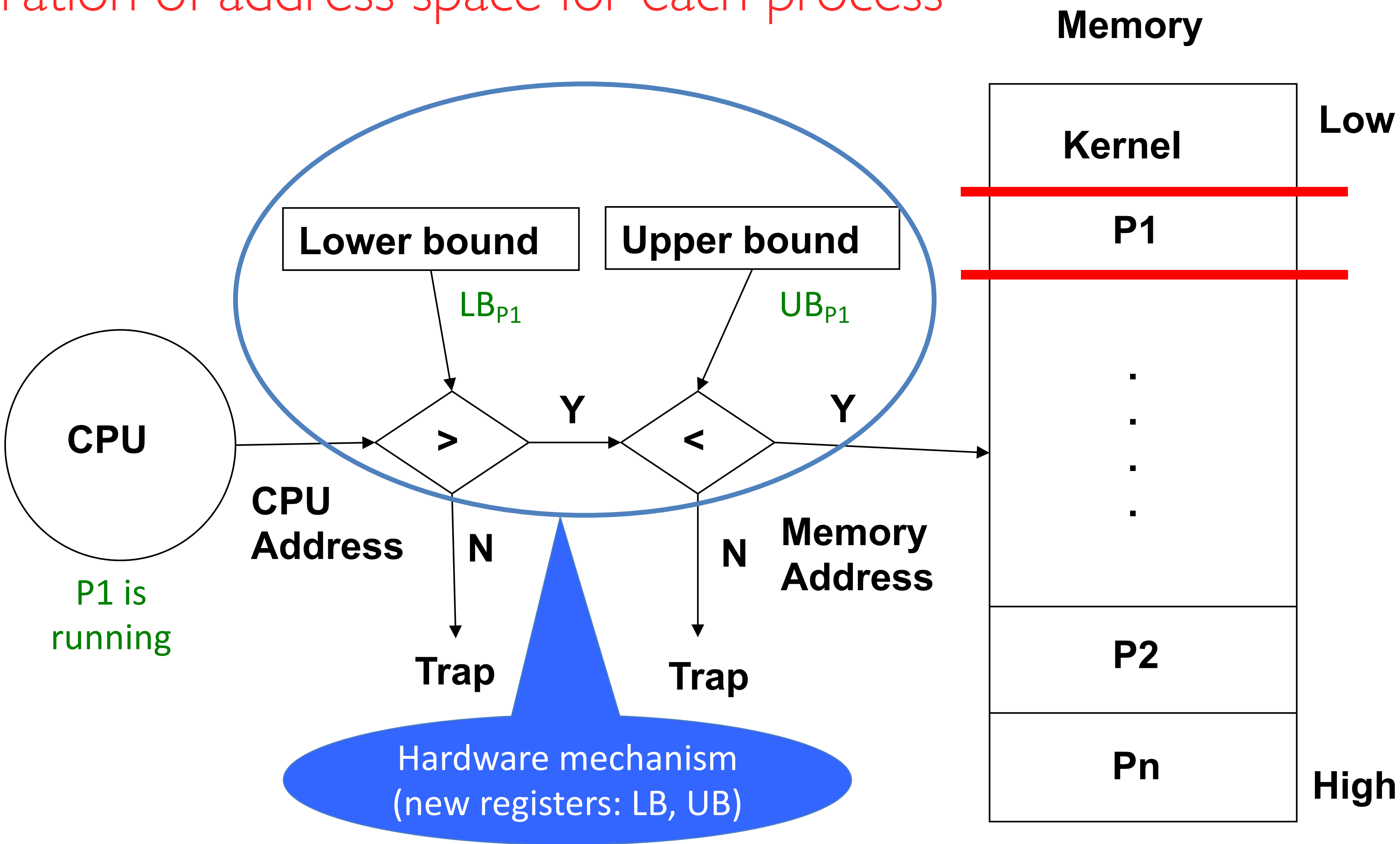


But modern operating systems are multiprogrammed!

Hardware mechanism (new register in hardware: FENCE)

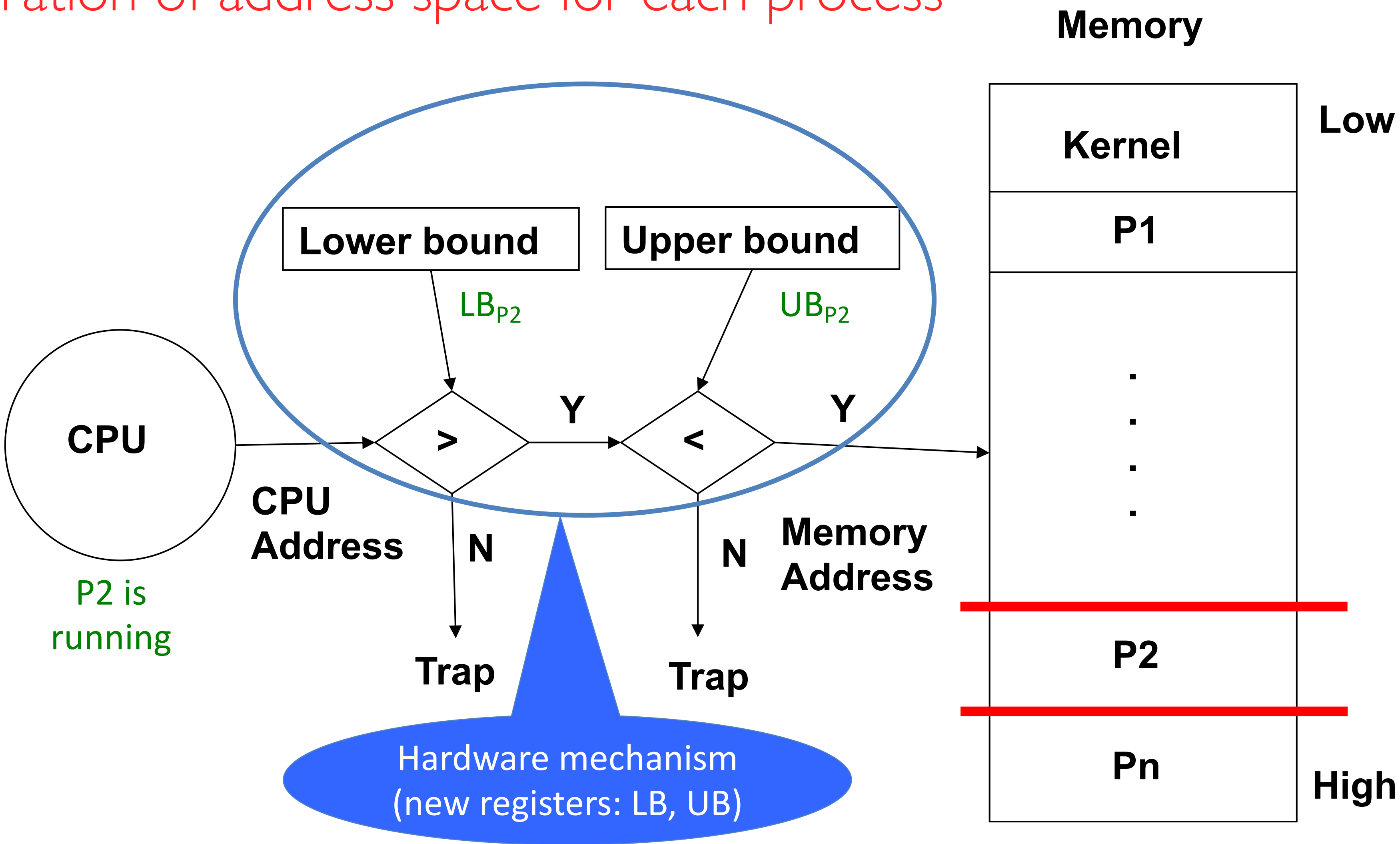
Multiprogrammed OS

- Separation of address space for each process



Multiprogrammed OS

- Separation of address space for each process



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 in the PCB

D. Calculate LB and UB from the process-id

PCB

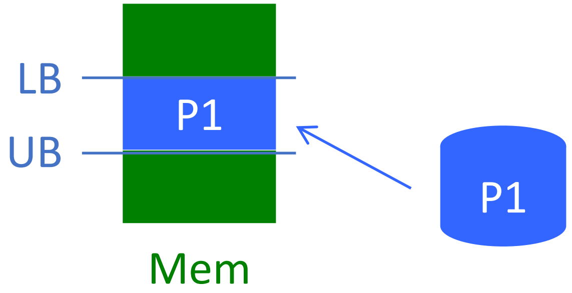
```
enum  state_type {new, ready, running,
                  waiting, halted};
typedef struct control_block_type {
  enum state_type state;
  address PC;
  int reg_file[NUMREGS];
  struct control_block *next_pcb;
  int priority;
  address memory_footprint; ????
  ….
  ….
} control_block;
```

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;
```

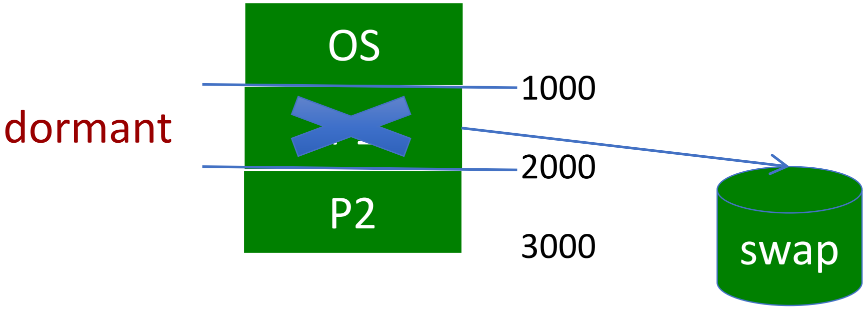
"Management" by the OS

- At link time
  - Linker determines address range used
  - 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



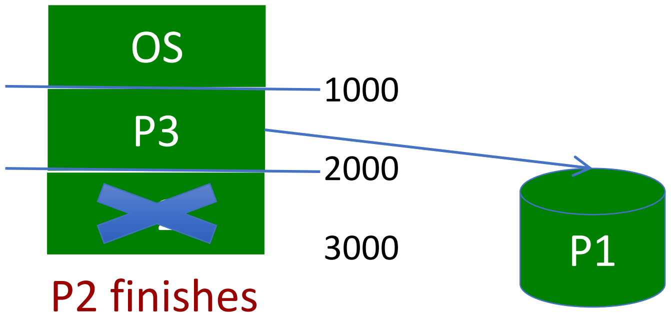
Limits of "bounds register" mechanism?

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



Limits of "bounds register" mechanism?

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

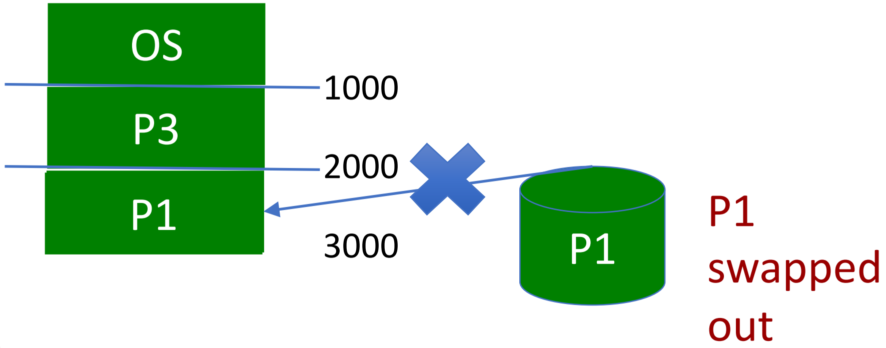


P2 finishes

P1 swapped out

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 an LC-2200 process "take root"?

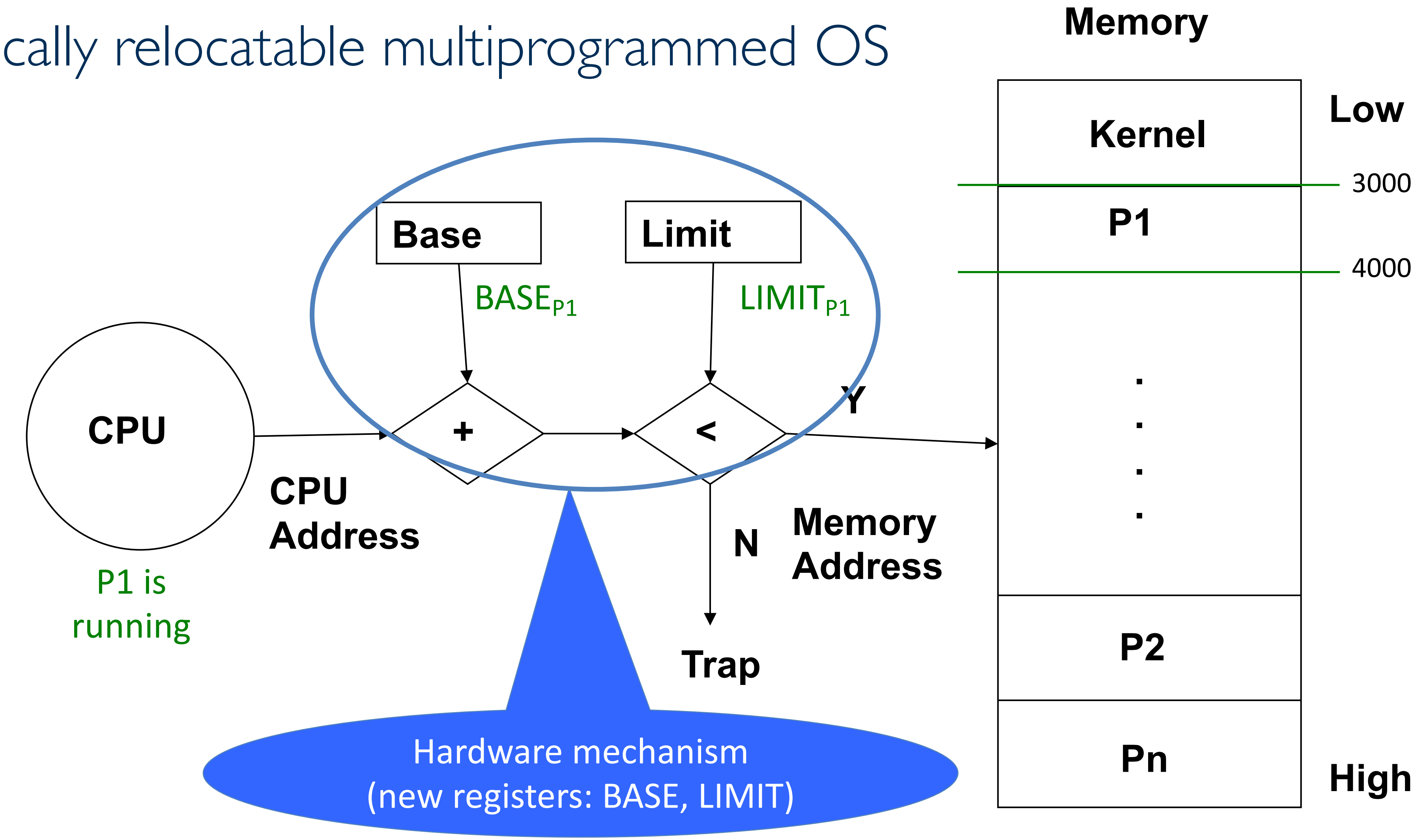
- All our addresses are hardcoded
  - In process P1, what about
    - LEA
    - JALR
- When do we lock-in memory addresses?
  - At link time ➔ static relocation
  - P1 can only be in memory between 1000 & 2000
  - That gives us poor memory utilization
  - i.e. not **dynamically relocatable**
- So the program isn't going to work after swap-in

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 memory addresses get saved in registers
  - Where can they go from there?
  - Can we find them so we can fix them?
  - This is why we’d say the LC-2200 code is **not relocatable**
  - You can’t move LC-2200 code once it’s started to run

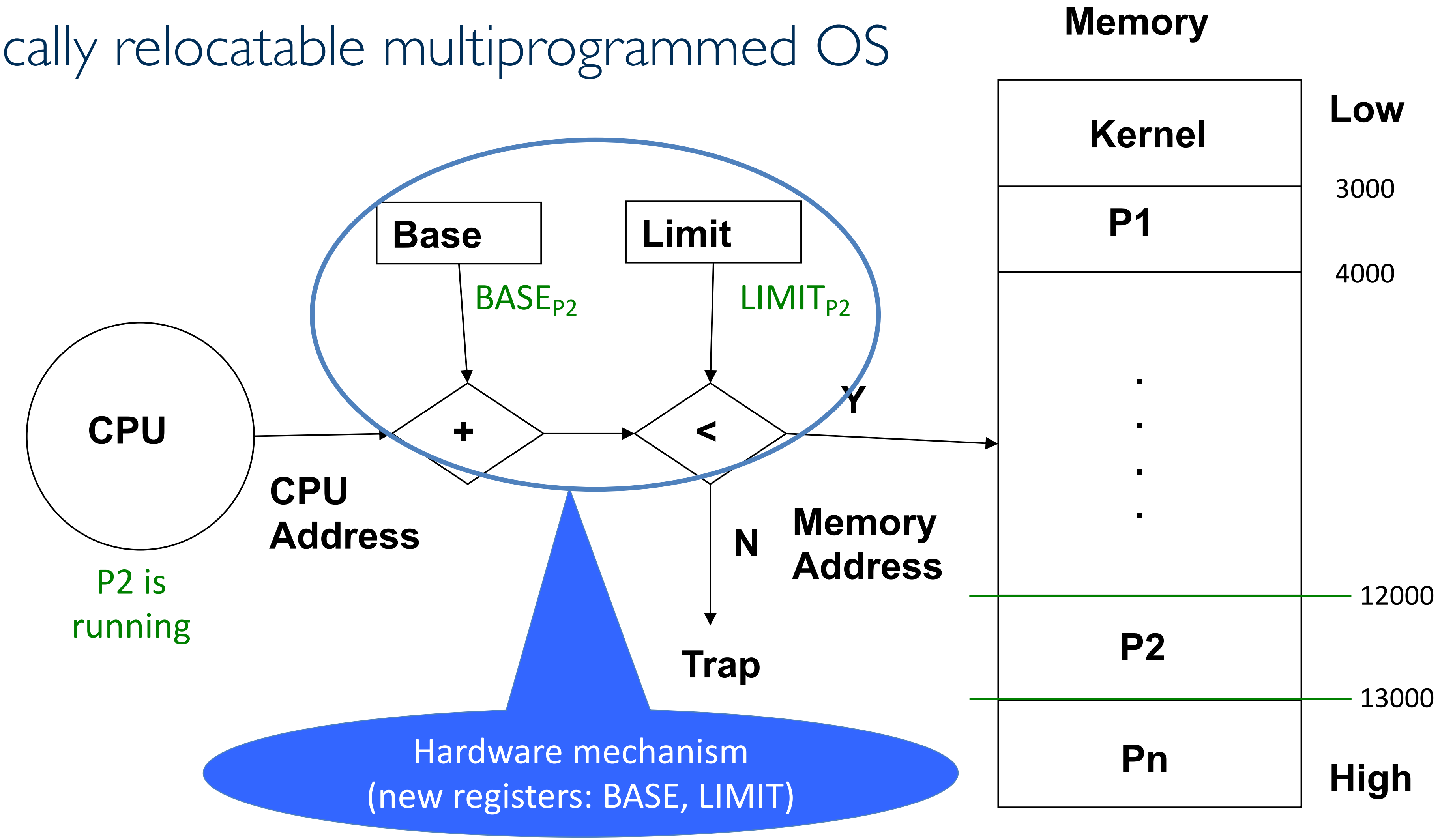
But there's a hardware solution…

- Dynamically relocatable multiprogrammed OS



But there's a hardware solution…

- Dynamically relocatable multiprogrammed OS



PCB

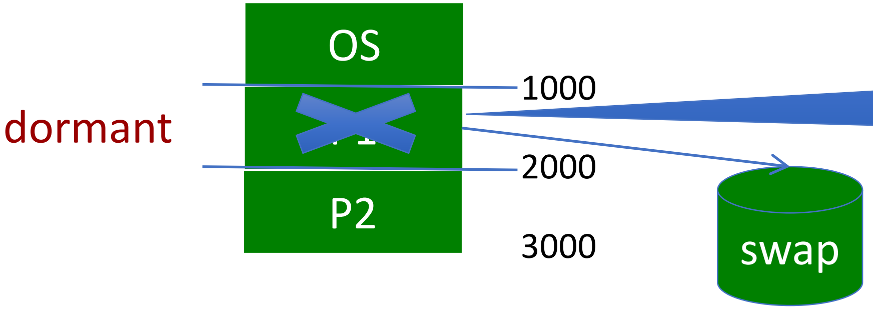
```
enum  state_type {new, ready, running,
                        waiting,
  halted};
typedef struct control_block_type {
  enum state_type state;
  address PC;
  int reg_file[NUMREGS];
  struct control_block *next_pcb;
  int priority;
  address memory_footprint; ????
  ….
  ….
} control_block;
```

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 last example with BASE + LIMIT

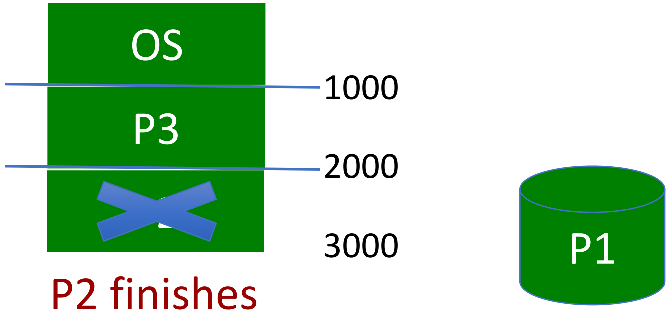
- Now all of P1 and P2's addresses are relative to zero
- This is our first instance of a **virtual address** where the process sees memory addresses *different* from the physical addresses we've been working with so far!



P1 sees virtual addresses 0 – 999
P1 uses physical addresses 1000-1999

The last example with BASE + LIMIT

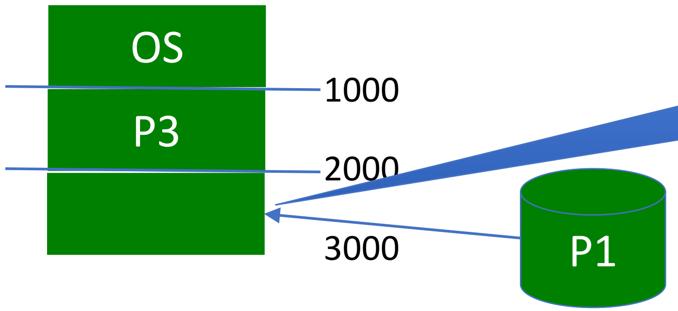
- Now all of P1 and P2's addresses are relative to zero
- This is our first instance of a **virtual address** where the process sees memory addresses *different* from the physical addresses we've been working with so far!



P2 finishes

The last example with BASE + LIMIT

- Now all of P1 and P2's addresses are relative to 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!



P1 sees virtual addresses 0 – 999
P1 uses physical addresses 2000-2999

- ✓ 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

Session ID:
dag2200

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

Recap

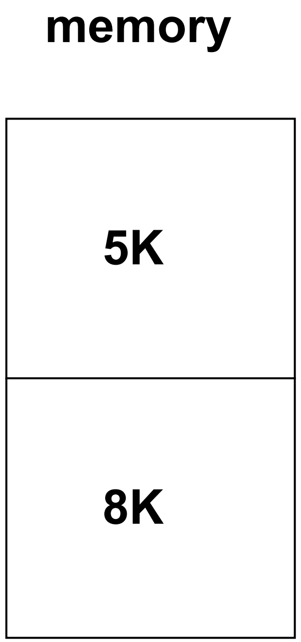
Hardware	Software
Fence register	User/kernel split
Bounds register	Static relocation
Base + limit register	Dynamic relocation

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

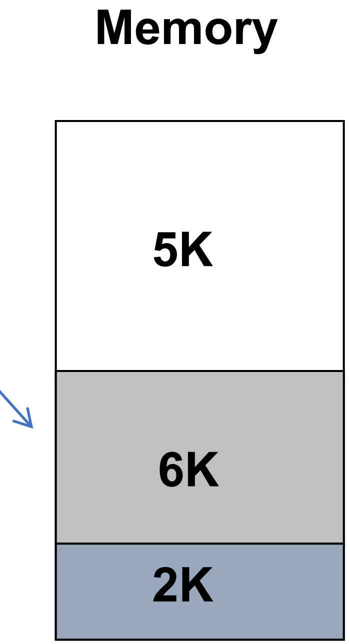
Occupied bit	Partition Size	Process
0	5K	---
0	8K	---

```
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



P1 needs 6K of memory

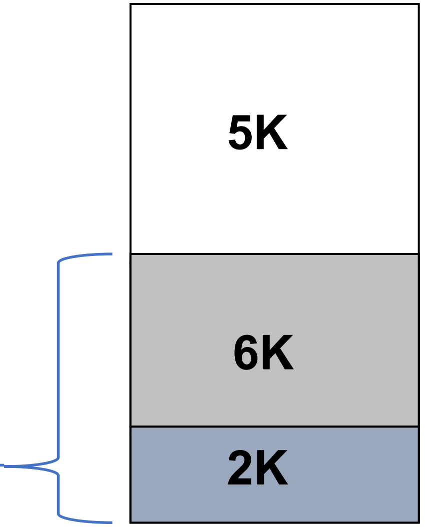
Allocation table

Table size is fixed = number of partitions

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

Needs only 6K

Memory



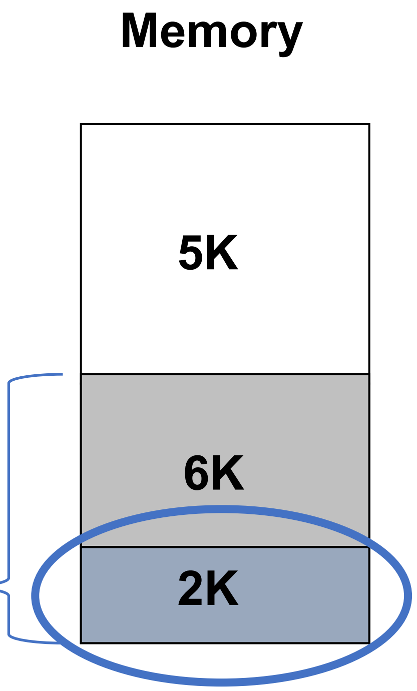
Internal fragmentation

Internal fragmentation = size of partition – actual used

Allocation table

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

Needs only 6K



Wasted (internal fragmentation)

Another process needs 6K memory

Do we have it?

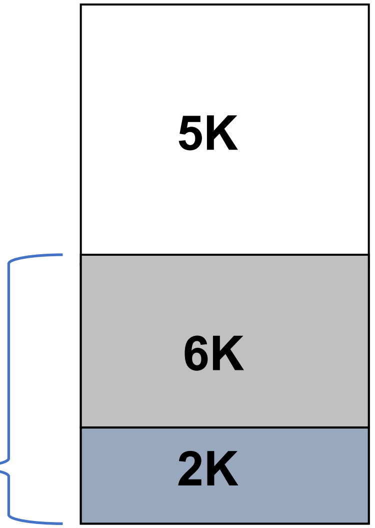
Memory manager has only a 5K partition…

Not possible

Allocation table

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

Memory



External fragmentation

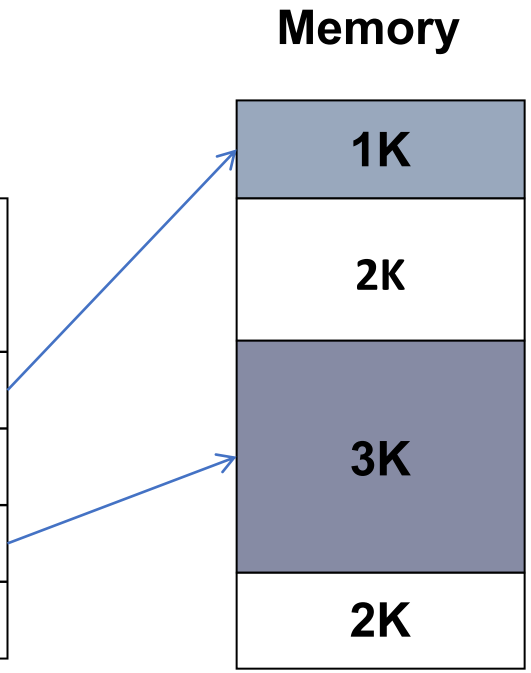
Consider P3 that needs 4K of memory

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	---



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

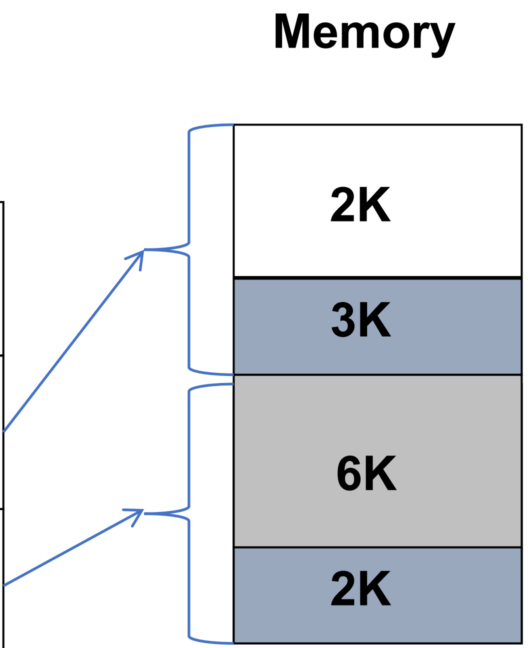
- Fragmentation
  - Internal
  - External

Session ID: dag2200

Total **internal** fragmentation is…

Allocation table

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



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

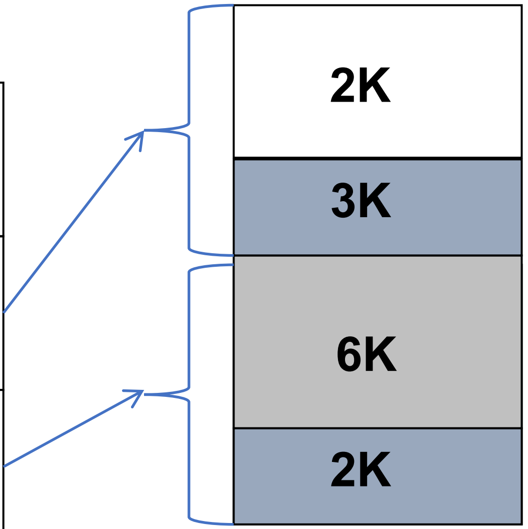
Session ID:
dag2200

Total **external** fragmentation is…

Allocation table

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

Memory



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

Overcoming internal fragmentation

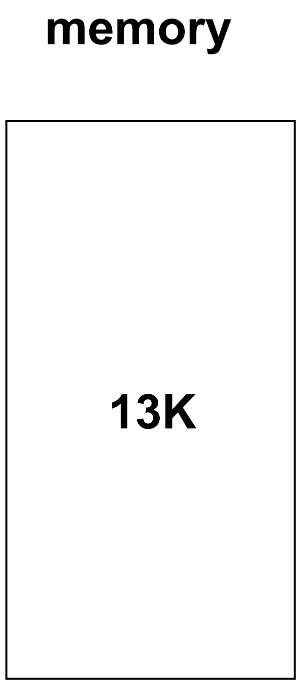
- Allocate exactly what is needed
- Variable size partitions

Variable size partitions

Memory manager allocation table

Start address	Size	Process
0	13K	FREE

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

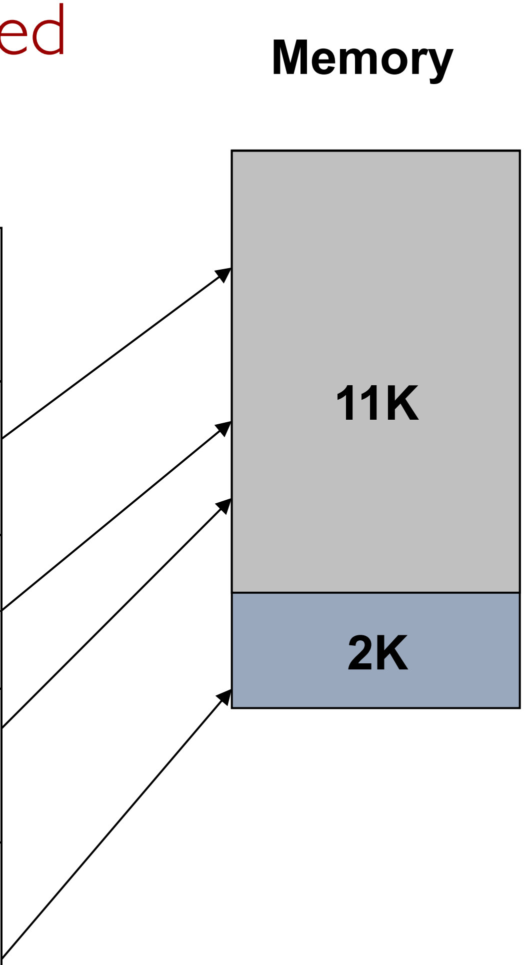


Partition table a little while later

Grows and shrinks as partitions get created and released

Allocation table

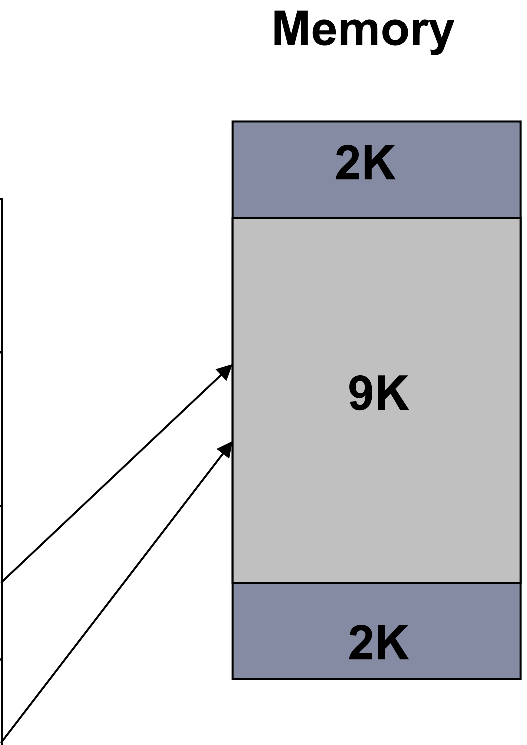
Start address	Size	Process
0	2K	P1
2K	6K	P2
8K	3K	P3
11K	2K	FREE



P1 exits

Allocation table

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



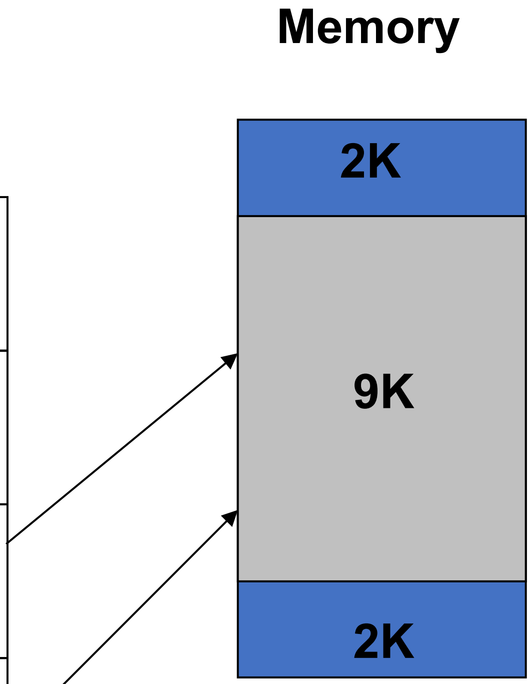
New request:
P4 needs 4K
Possible?

External fragmentation

P2 exits

Allocation table

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

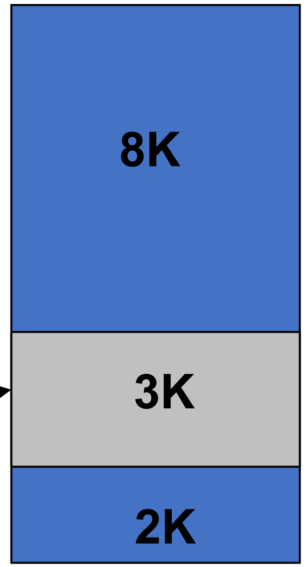


Coalescing two free areas

Allocation table

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

Memory



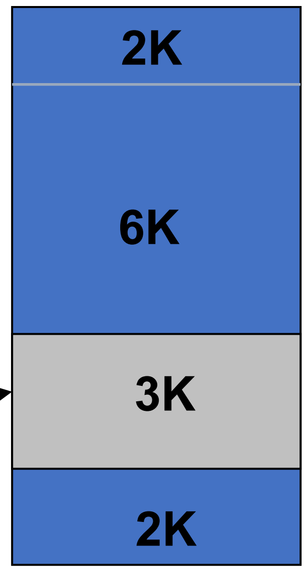
Reducing external fragmentation

- 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

Memory



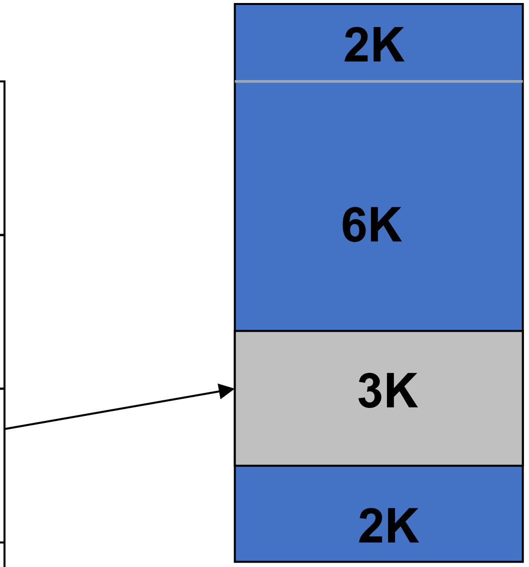
Compaction

- Request for 9K

Allocation table

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

Memory

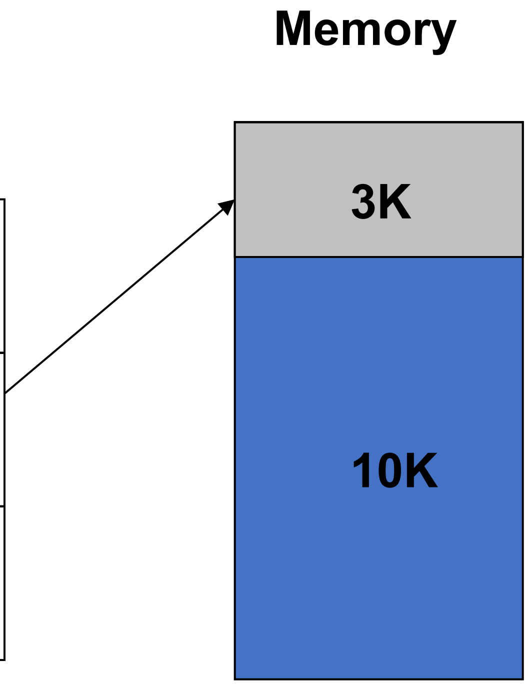


Compaction

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

Allocation table

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



Session ID: dag2200

With variable size partition memory management there is …

A. No external fragmentation

B. No internal fragmentation

C. No fragmentation

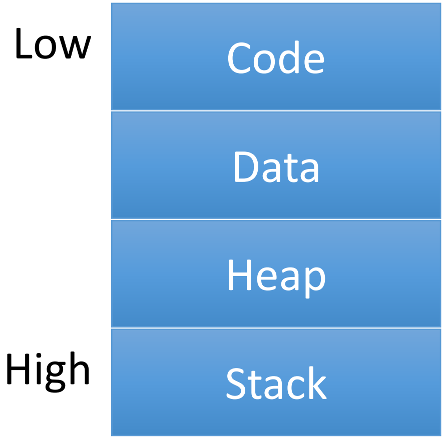
D. Both internal and external fragmentation

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 memory footprint looks like this



Low

Code

Data

Heap

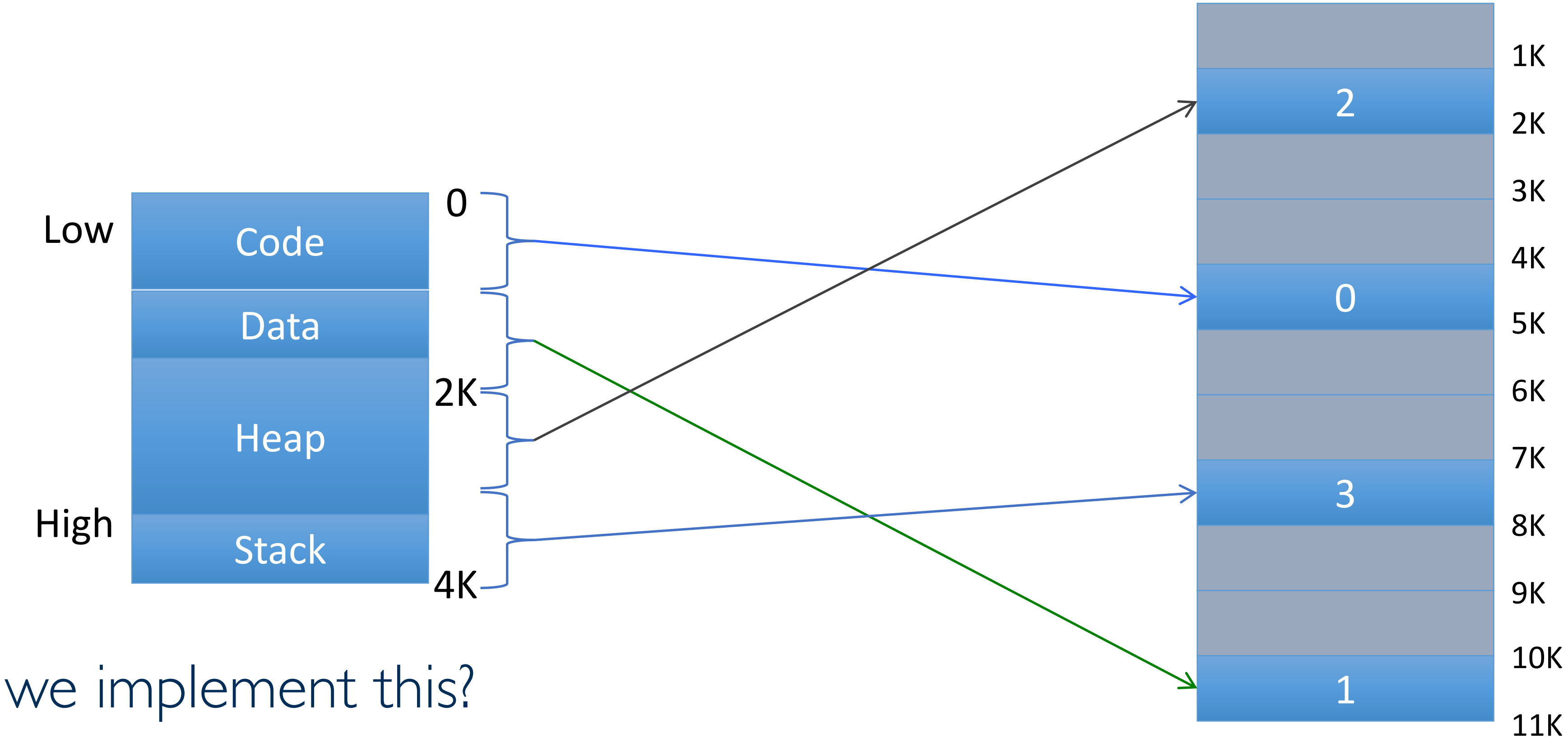
High

Stack

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

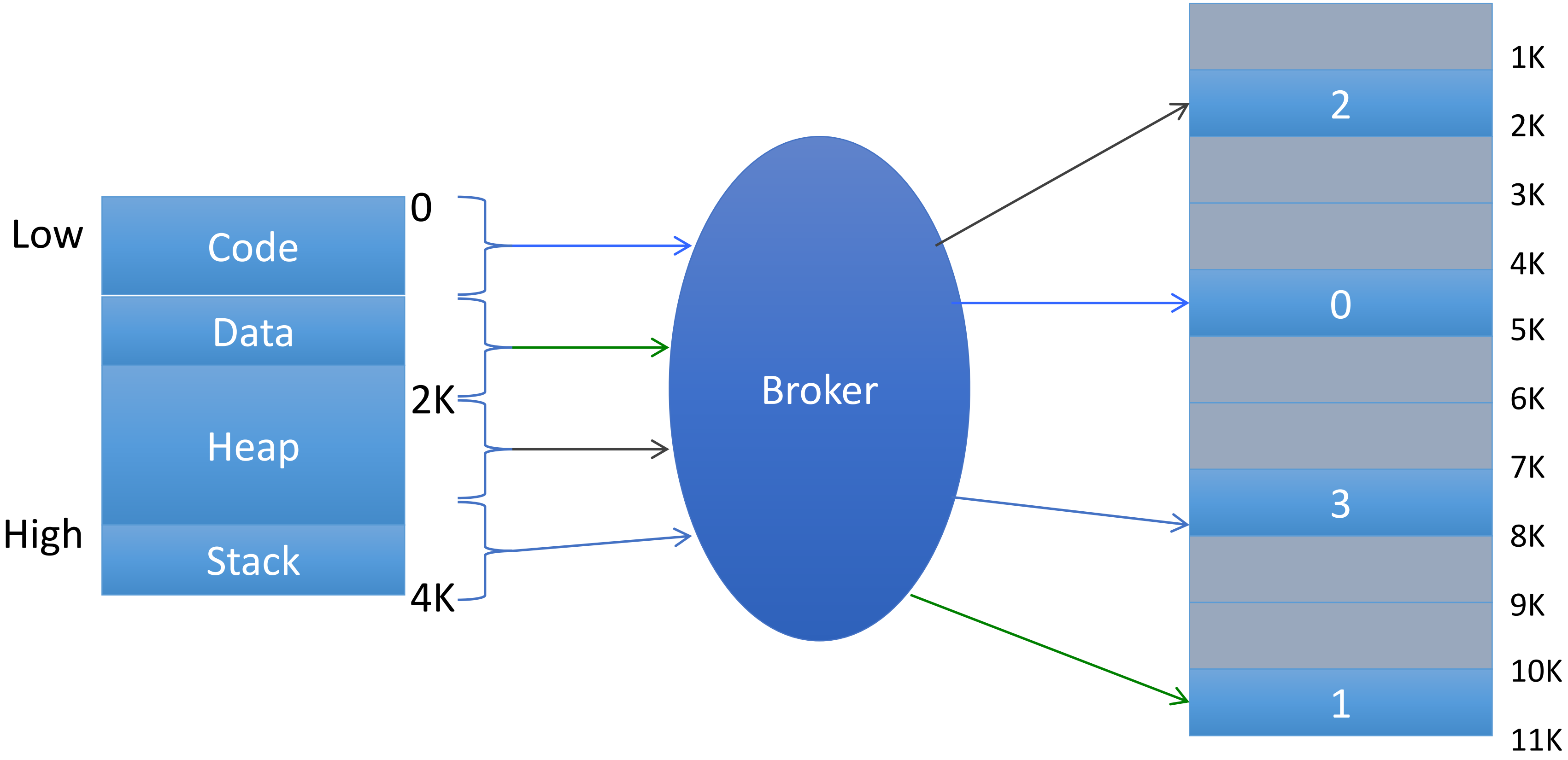
How might we solve the external fragmentation problem?

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



- But how can we implement this?

Use more sophisticated broker between CPU and memory



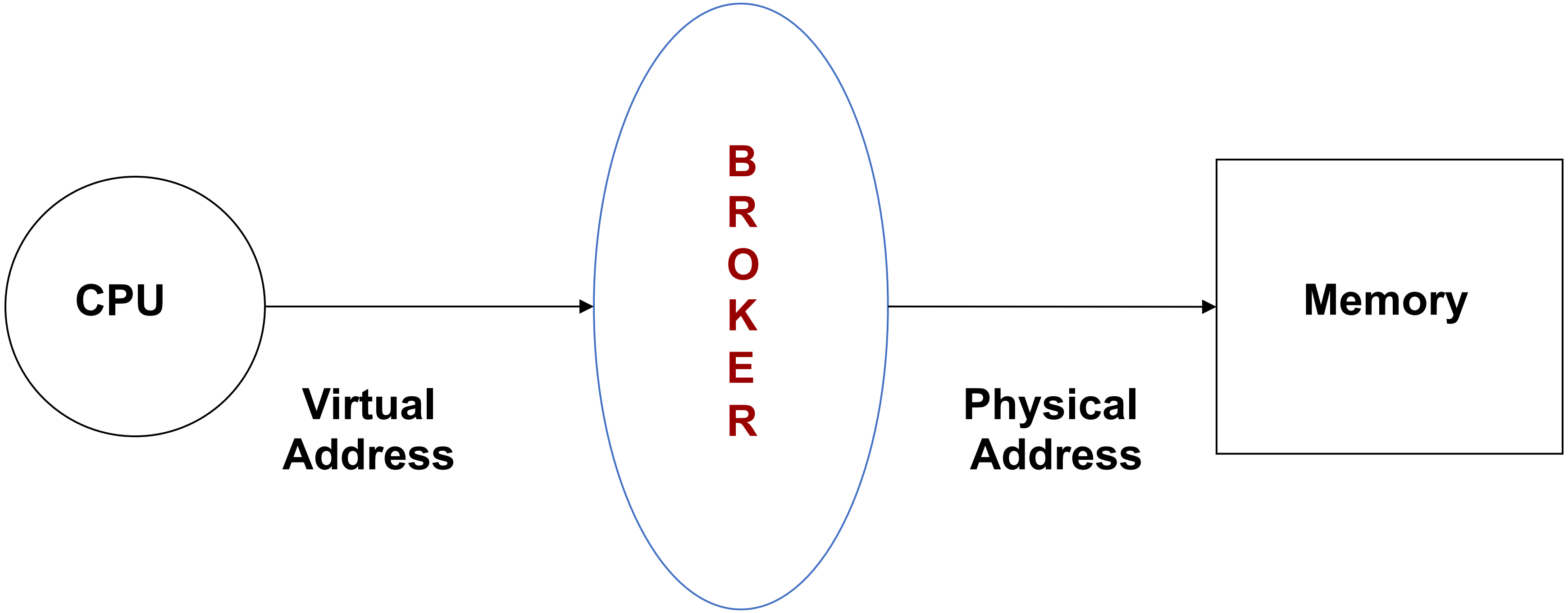
contiguous

discontiguous

Broker

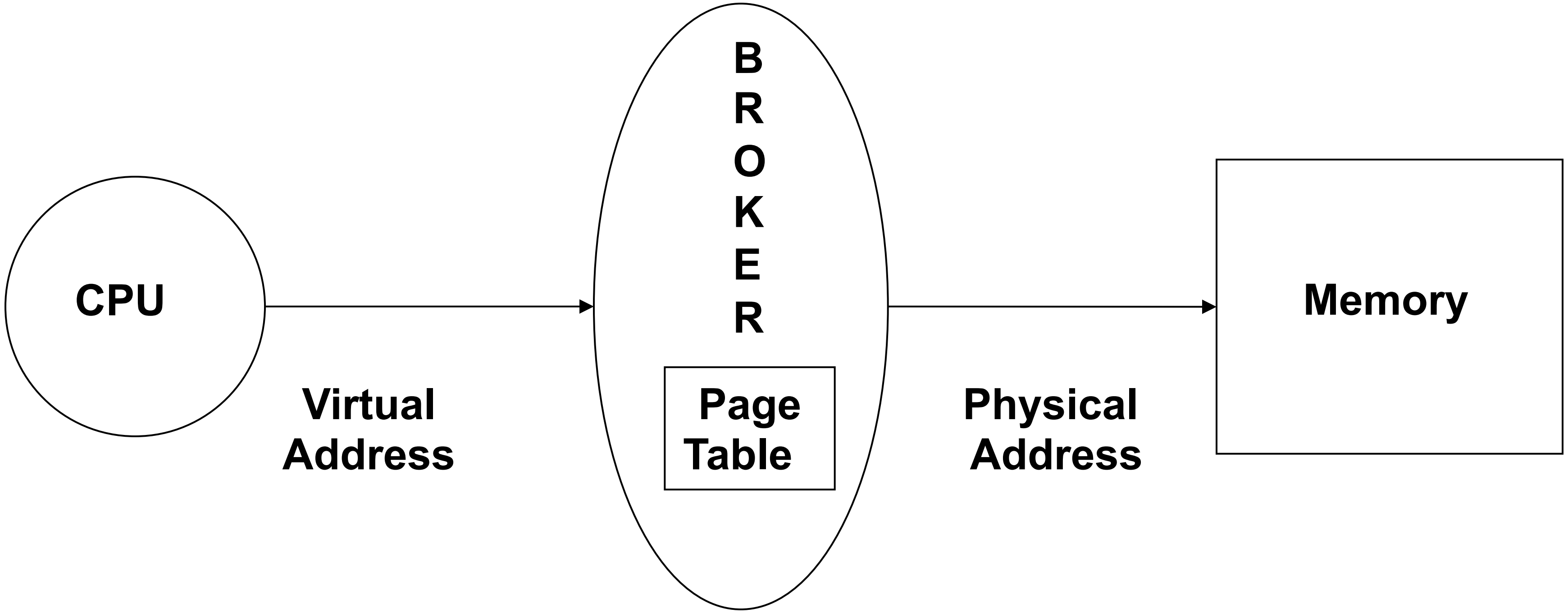
- This broker maps
  - Virtual address (VA) from the CPU

    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 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

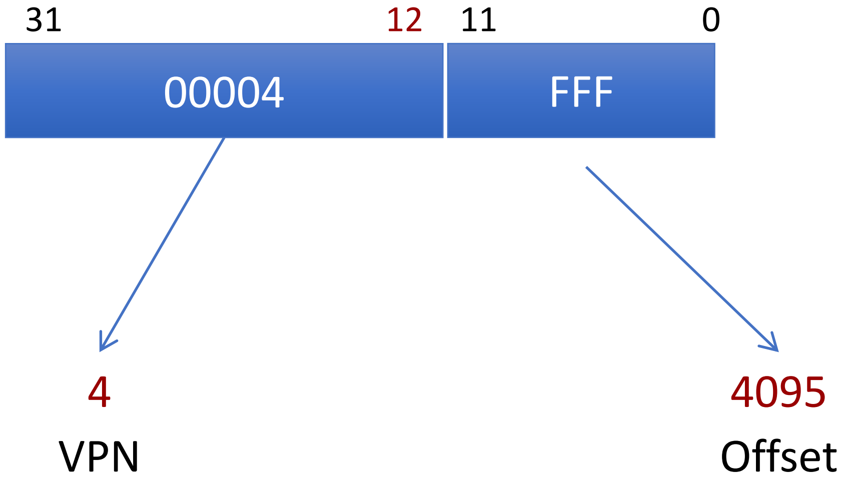
- These days
  - 4 KB up to 1 GB
  - Often it's configurable per process

- 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**

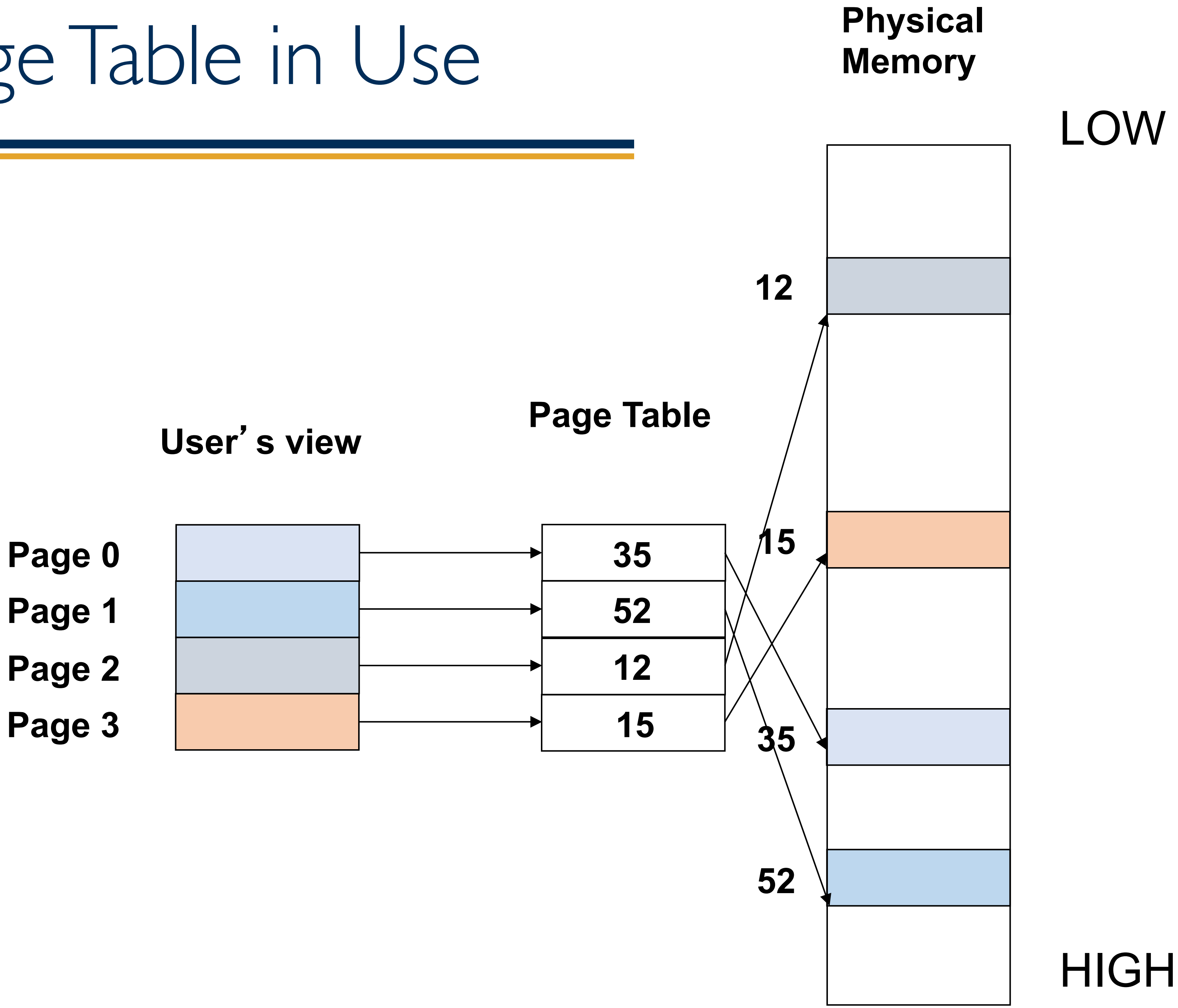
Splitting up a virtual address

- 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

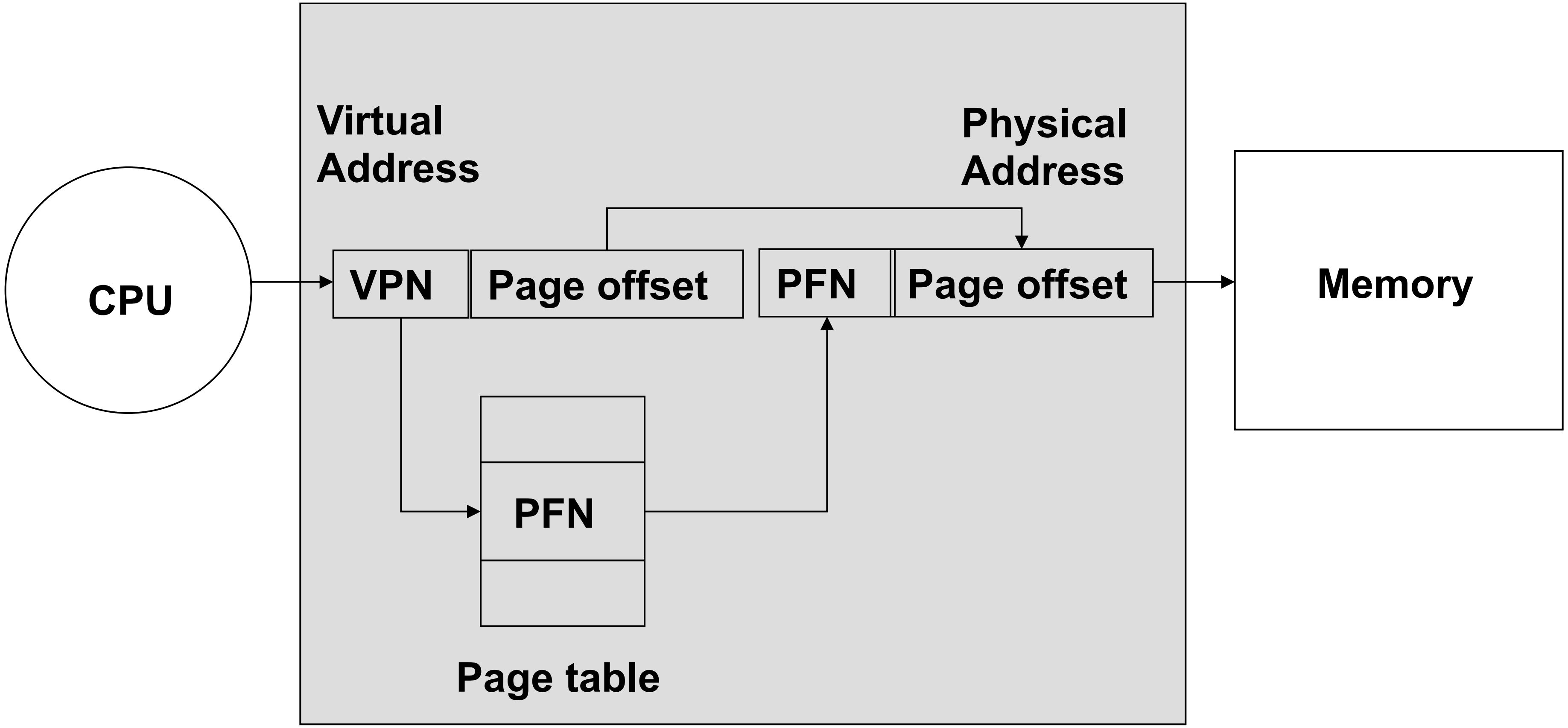
- For example, for virtual address 0x00004FFF:



Page Table in Use

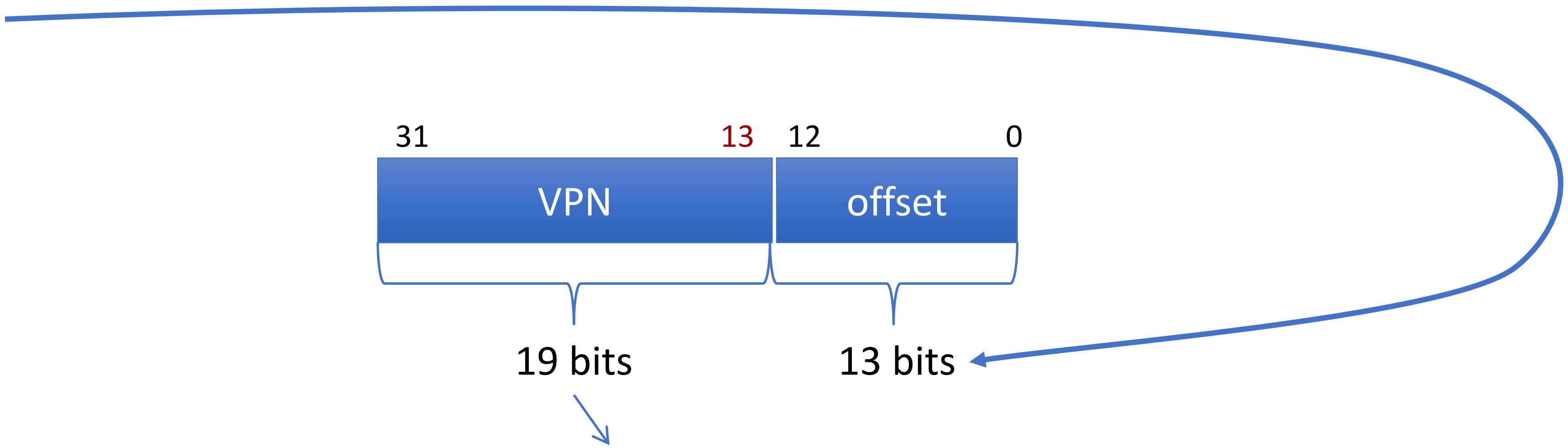


Address translation



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?
- 8k = 2^{13}



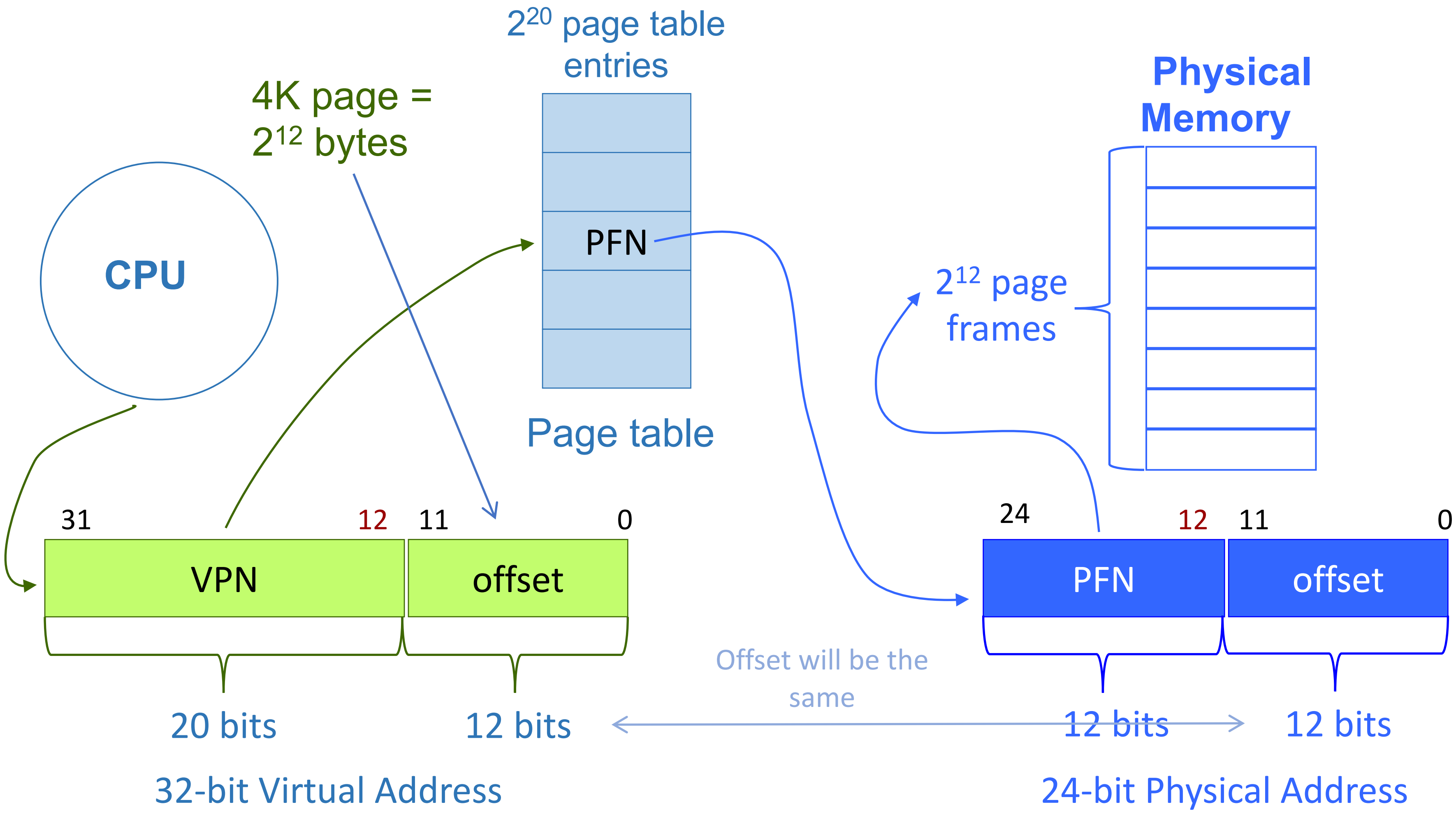
- VPN is 19 bits, so page table is 2^{19} or 524,288 entries

Examples

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

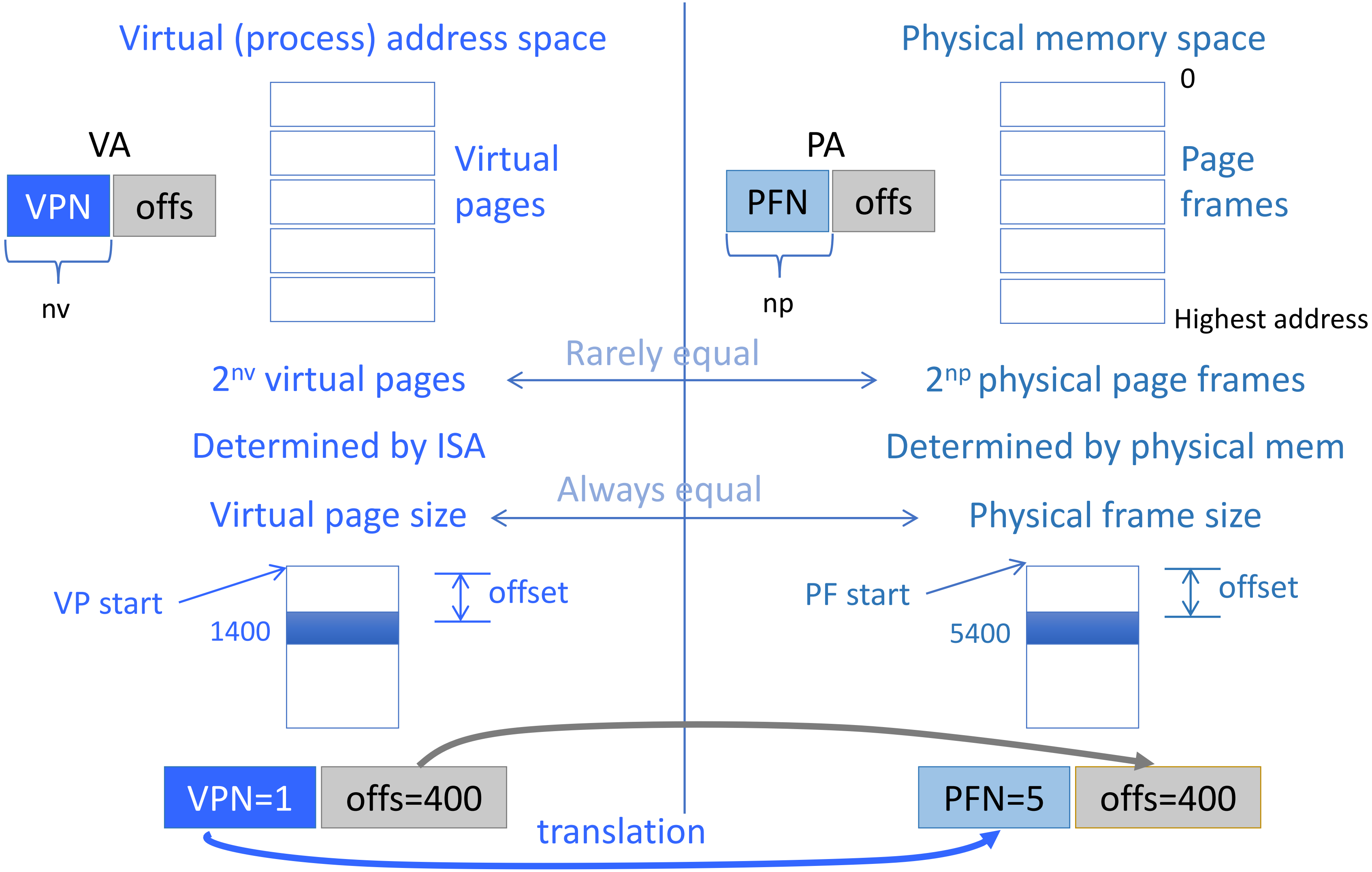
- How many page frames in memory?
- How big is the page table?
- How many page frames are there in this memory system?

Example



2^{20} page table entries
4K page = 2^{12} bytes
CPU
PFN
Page table
Physical Memory
2^{12} page frames
31
12
11
0
VPN
offset
20 bits
12 bits
32-bit Virtual Address
Offset will be the same
24
12
11
0
PFN
offset
12 bits
12 bits
24-bit Physical Address

Important facts about paging



Virtual (process) address space

VA

VPN offs

nv

Virtual pages

Physical memory space

0

PA

PFN offs

np

Page frames

Highest address

2^{nv} virtual pages

Rarely equal

2^{np} physical page frames

Determined by ISA

Determined by physical mem

Virtual page size

Always equal

Physical frame size

VP start

1400

offset

PF start

5400

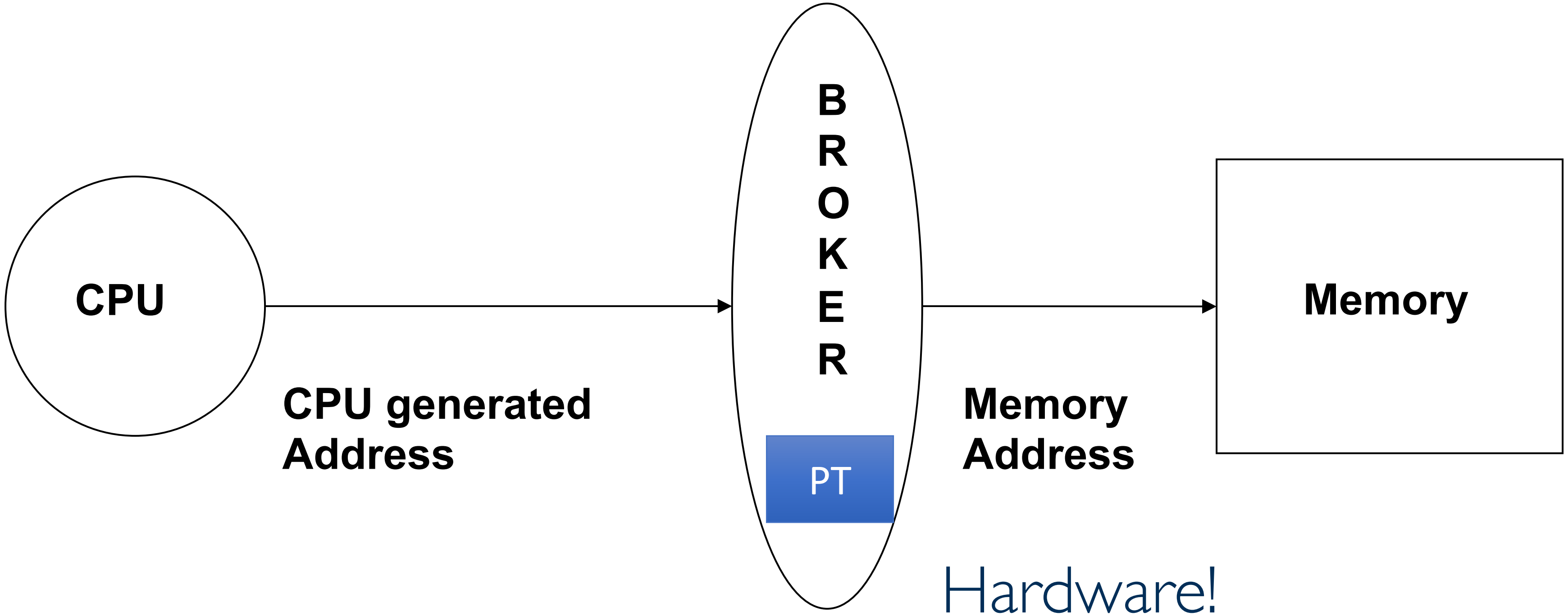
offset

VPN=1 offs=400

PFN=5 offs=400

translation

So exactly where is the page table?

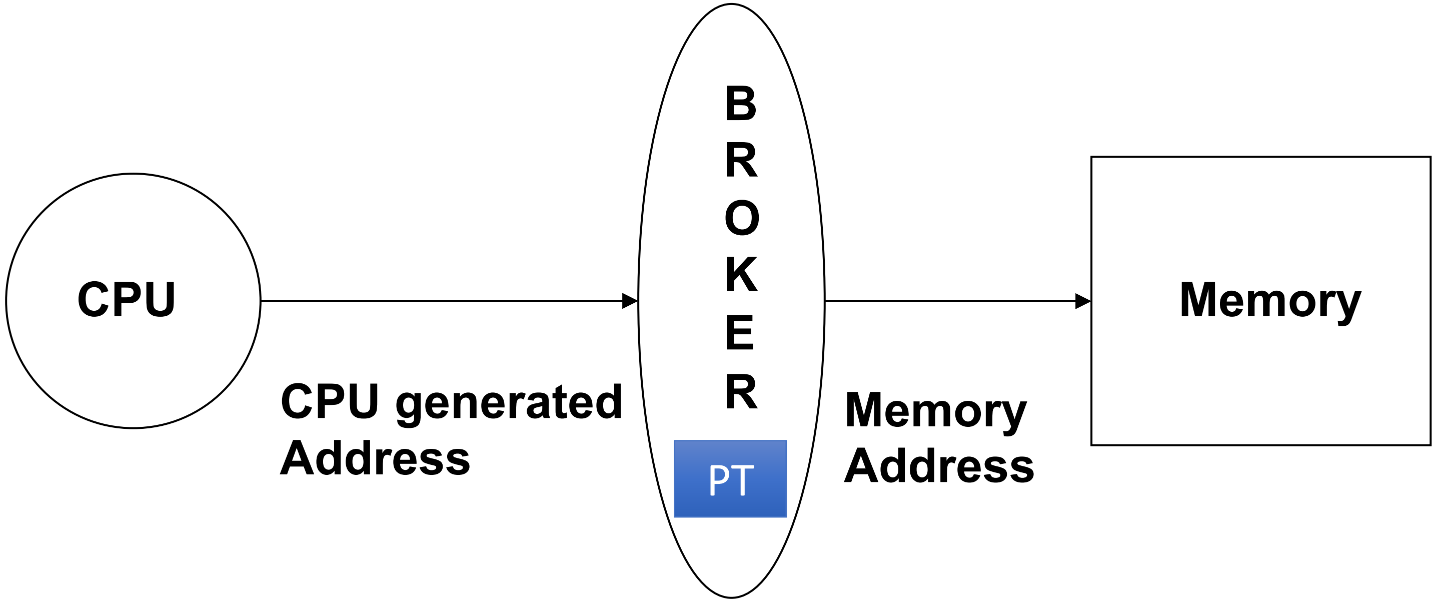


Hardware!

But it’s not a special device

It’s in physical memory!

How many page tables are there?

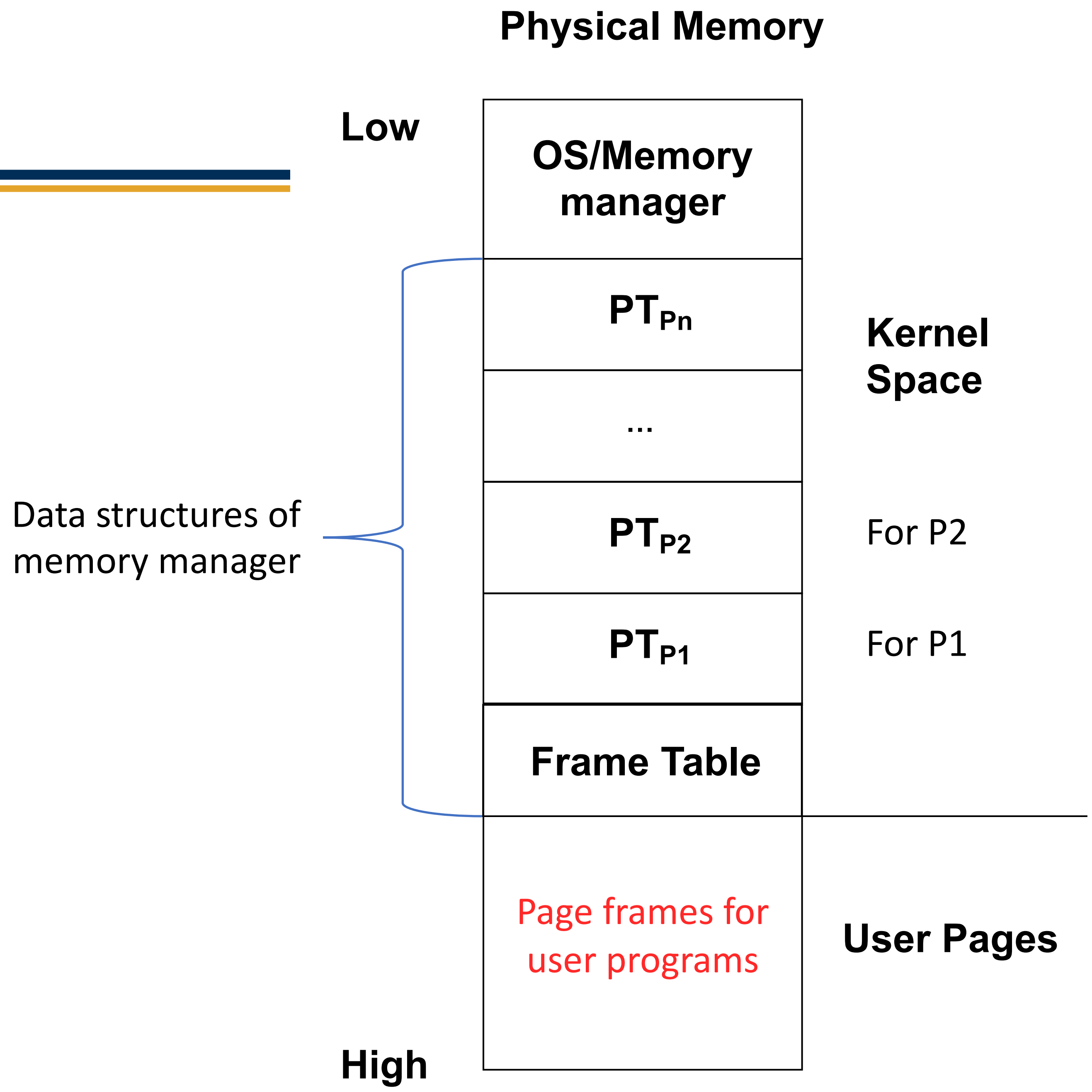


Process 1, 2, 3, …, n in memory

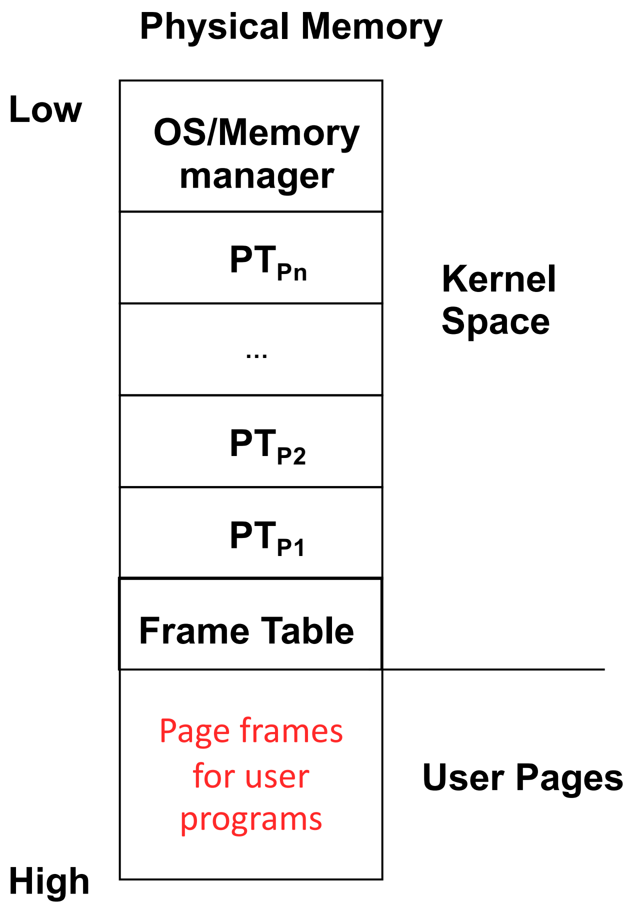
We'll need **n** page tables!

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

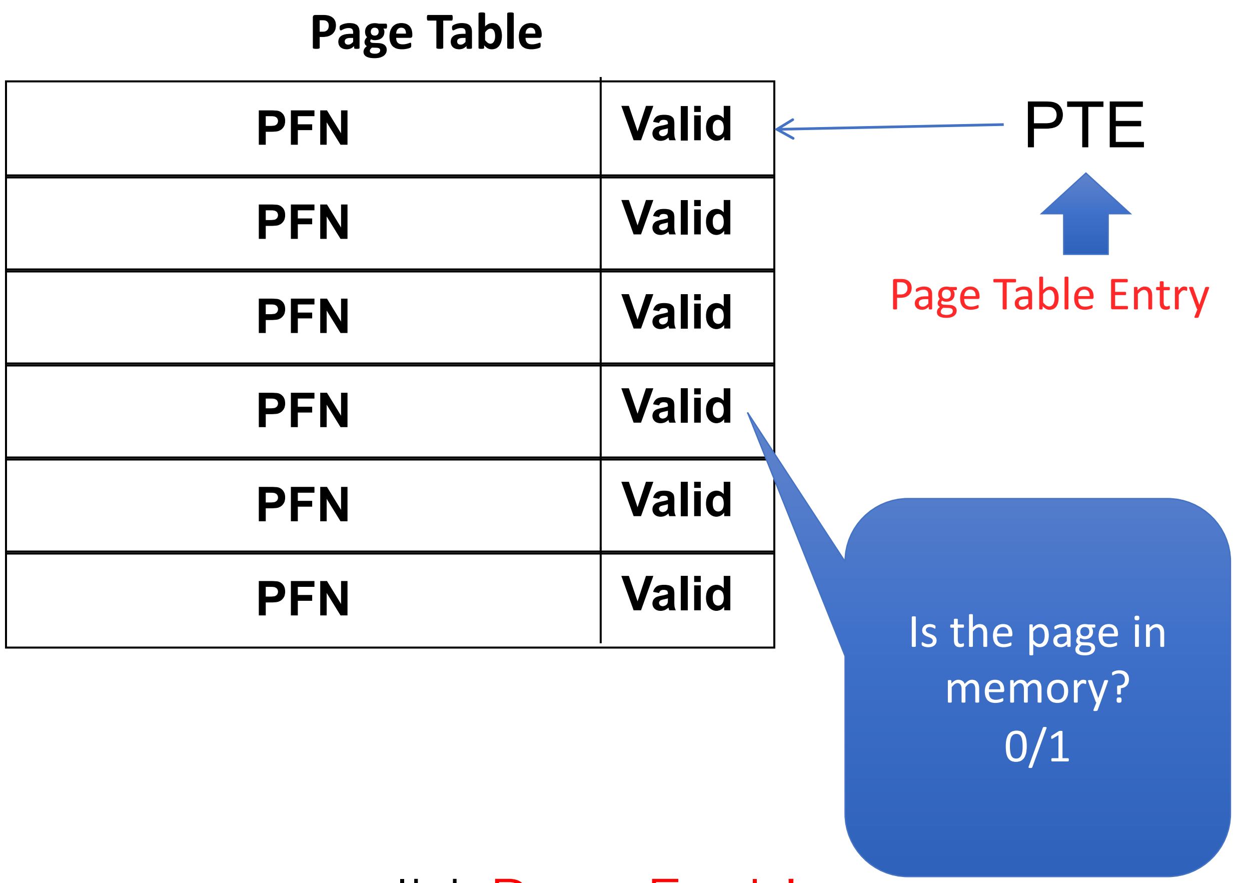
PCB

```
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**

PTE

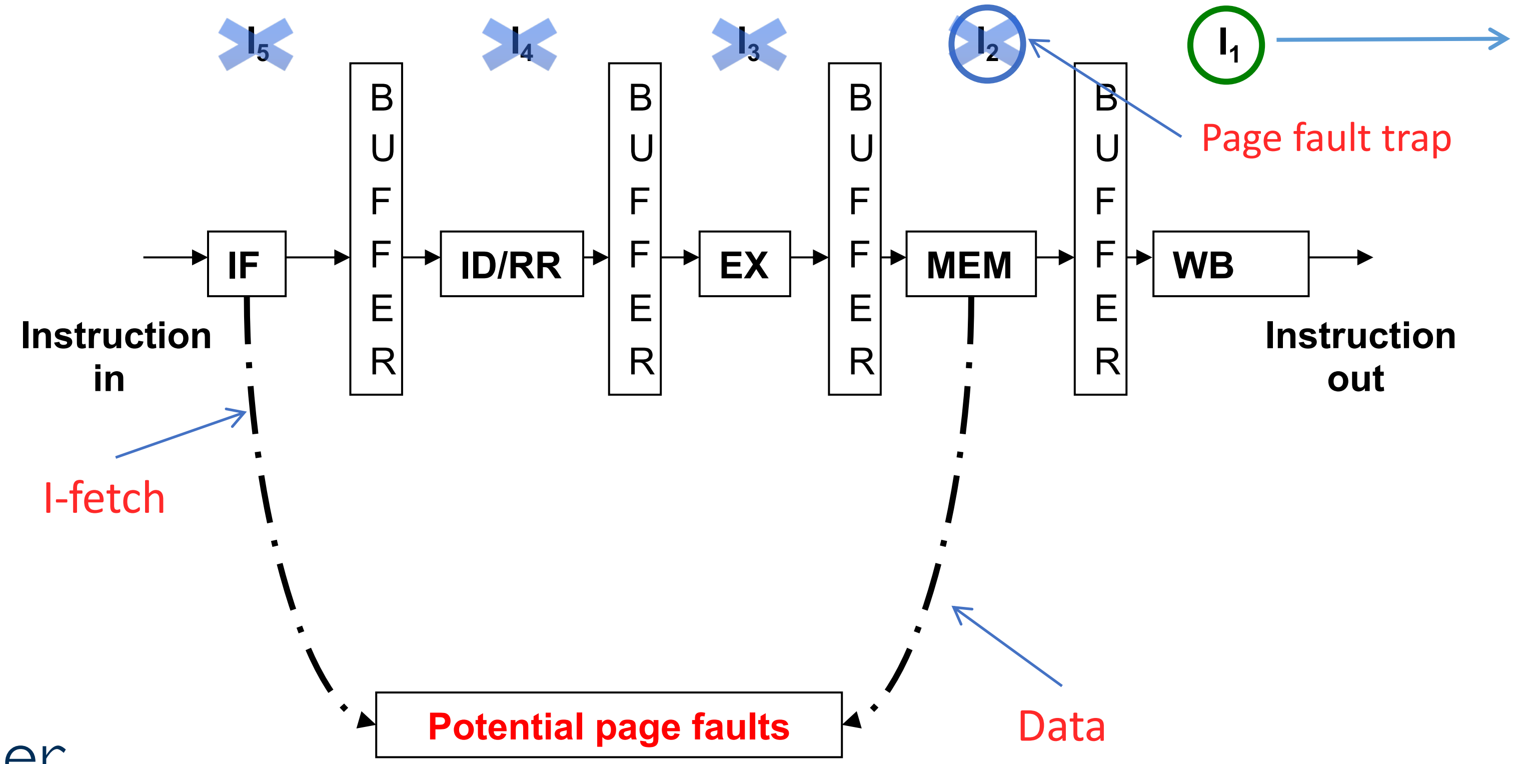
Page Table Entry

Is the page in memory?
0/1

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

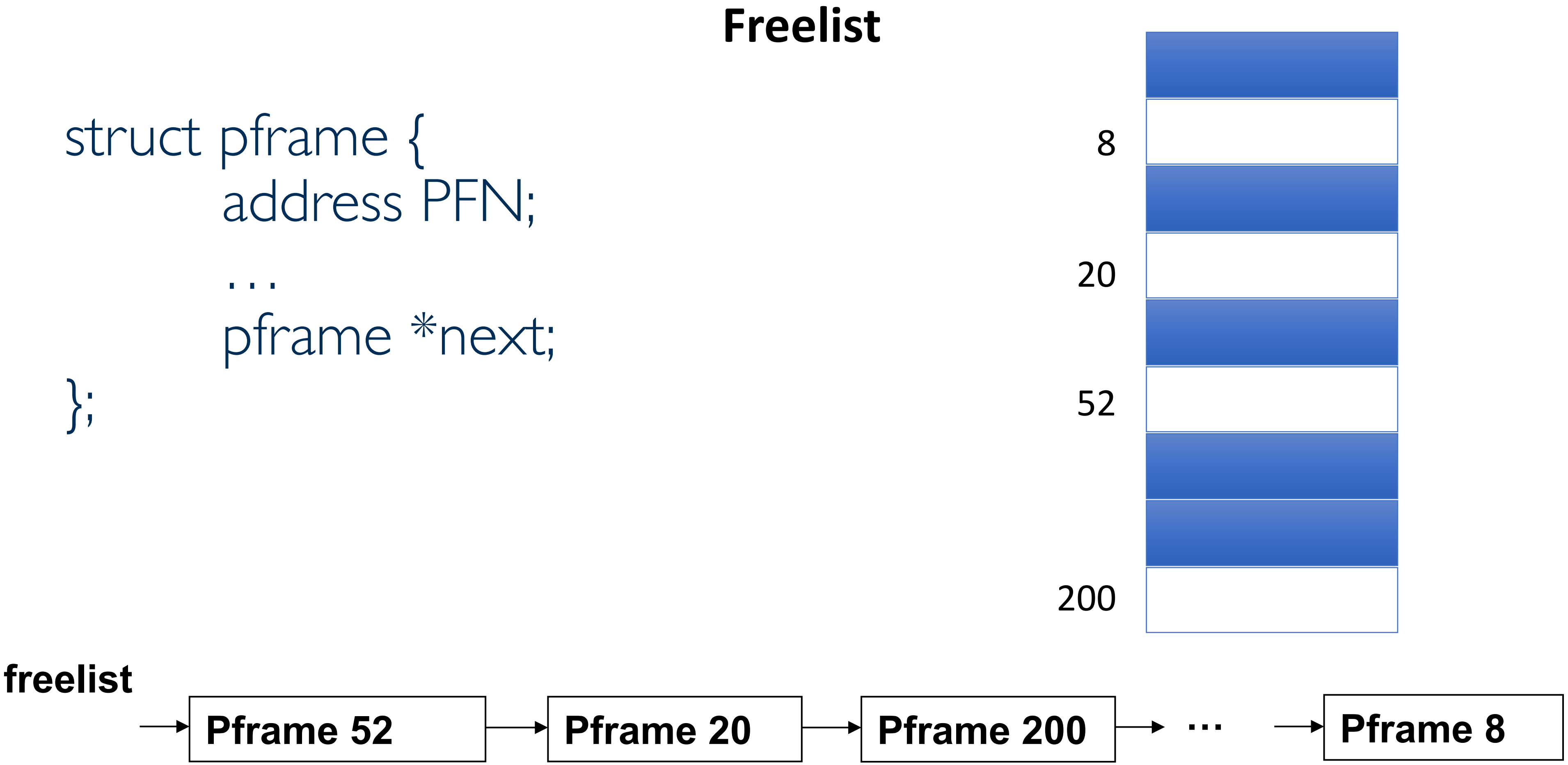
As you may have guessed, we'll need to make the original PC value part of the pipeline buffers

Page fault handler

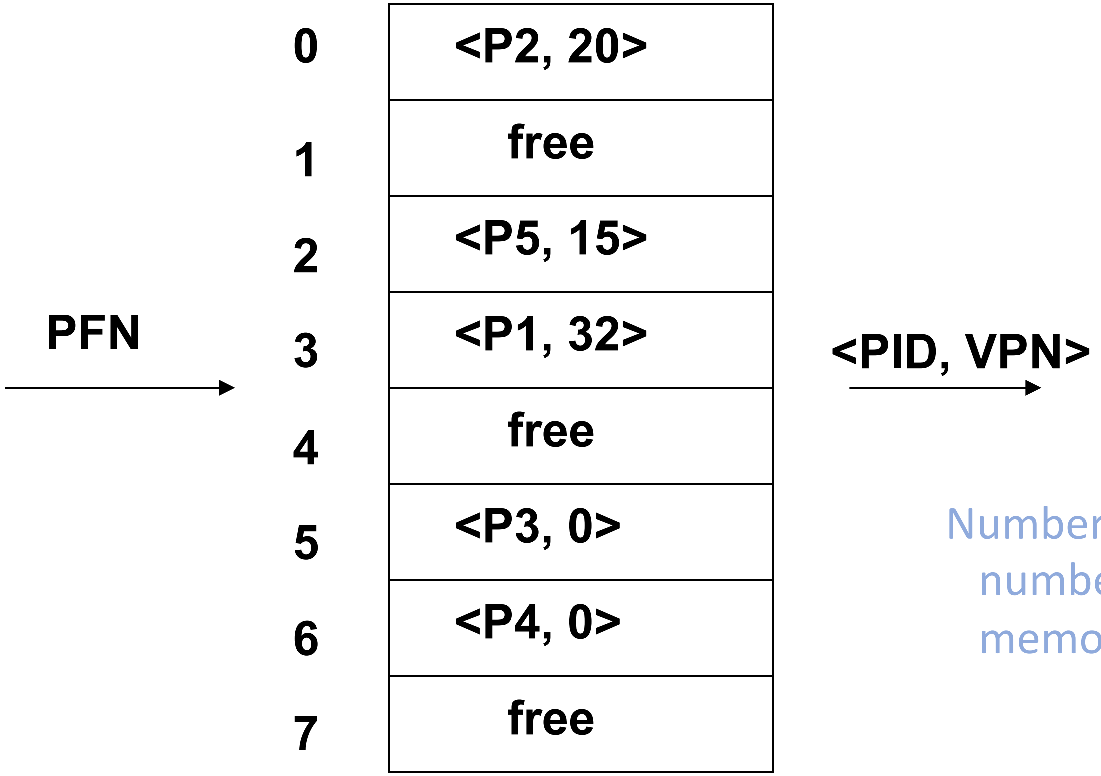
1. Find a free page frame
2. Load the faulting virtual page from disk into the page frame
3. Give up the CPU while waiting for the paging I/O to complete
4. Update the page table entry for the faulting page
5. Place the PCB of the process back in the ready_q of the scheduler
6. Call the scheduler

Data structures for page fault handler

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



Frame Table



PFN →

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

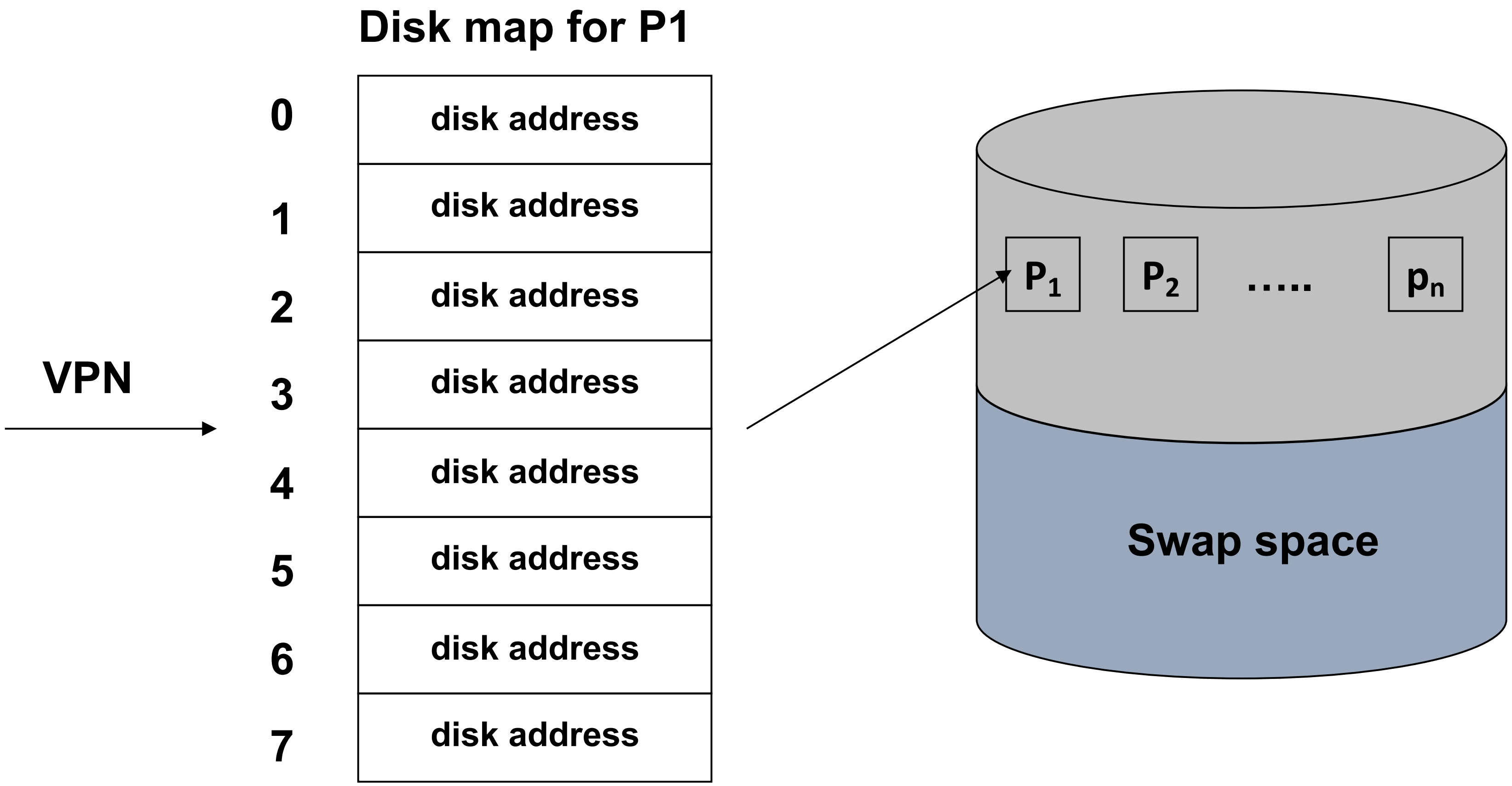
→ <PID, VPN>

Number of entries =
number of physical
memory page frames

Reverse mapping compared to PT

We need this for evicting pages

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

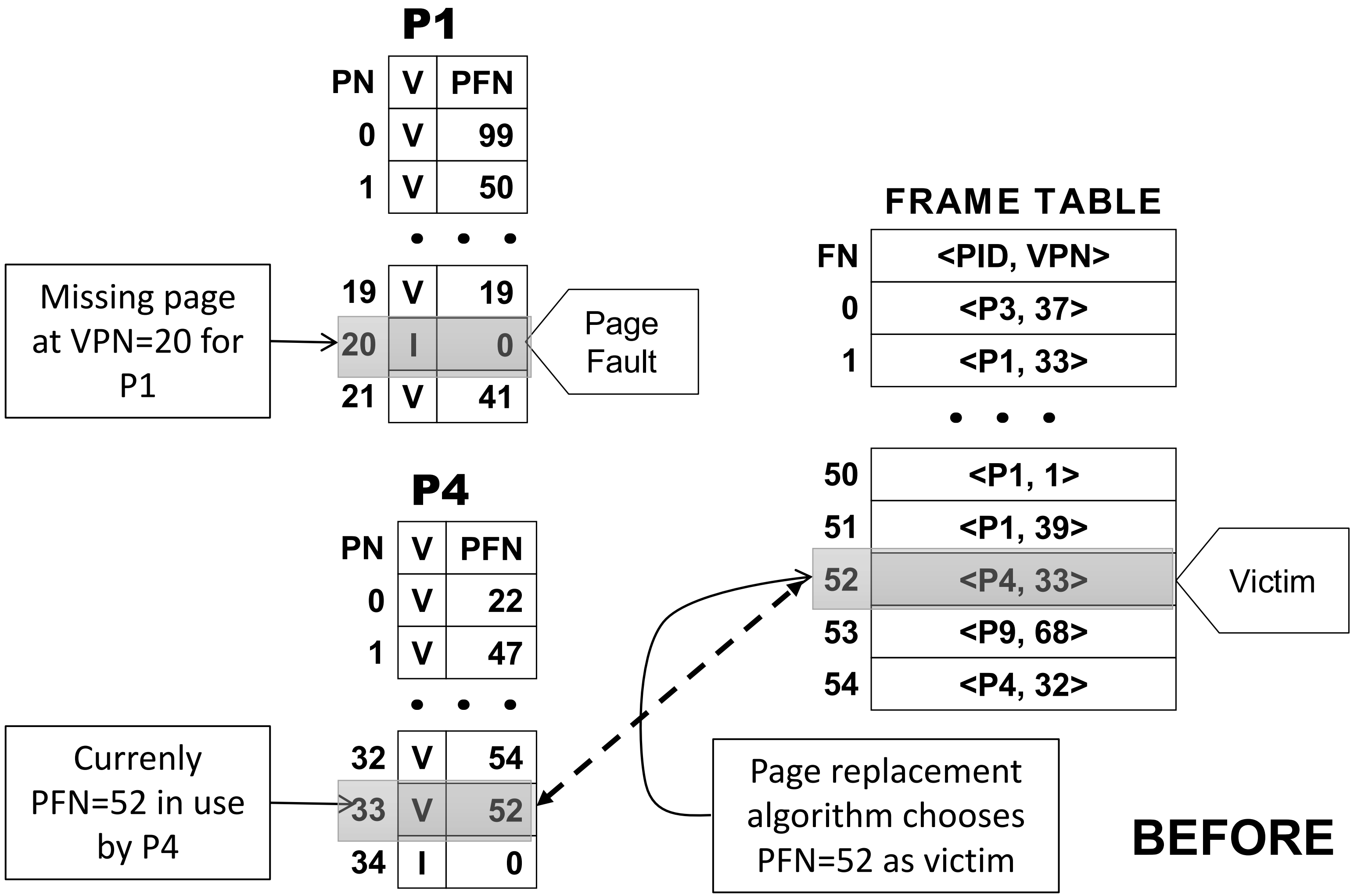
PCB

```
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

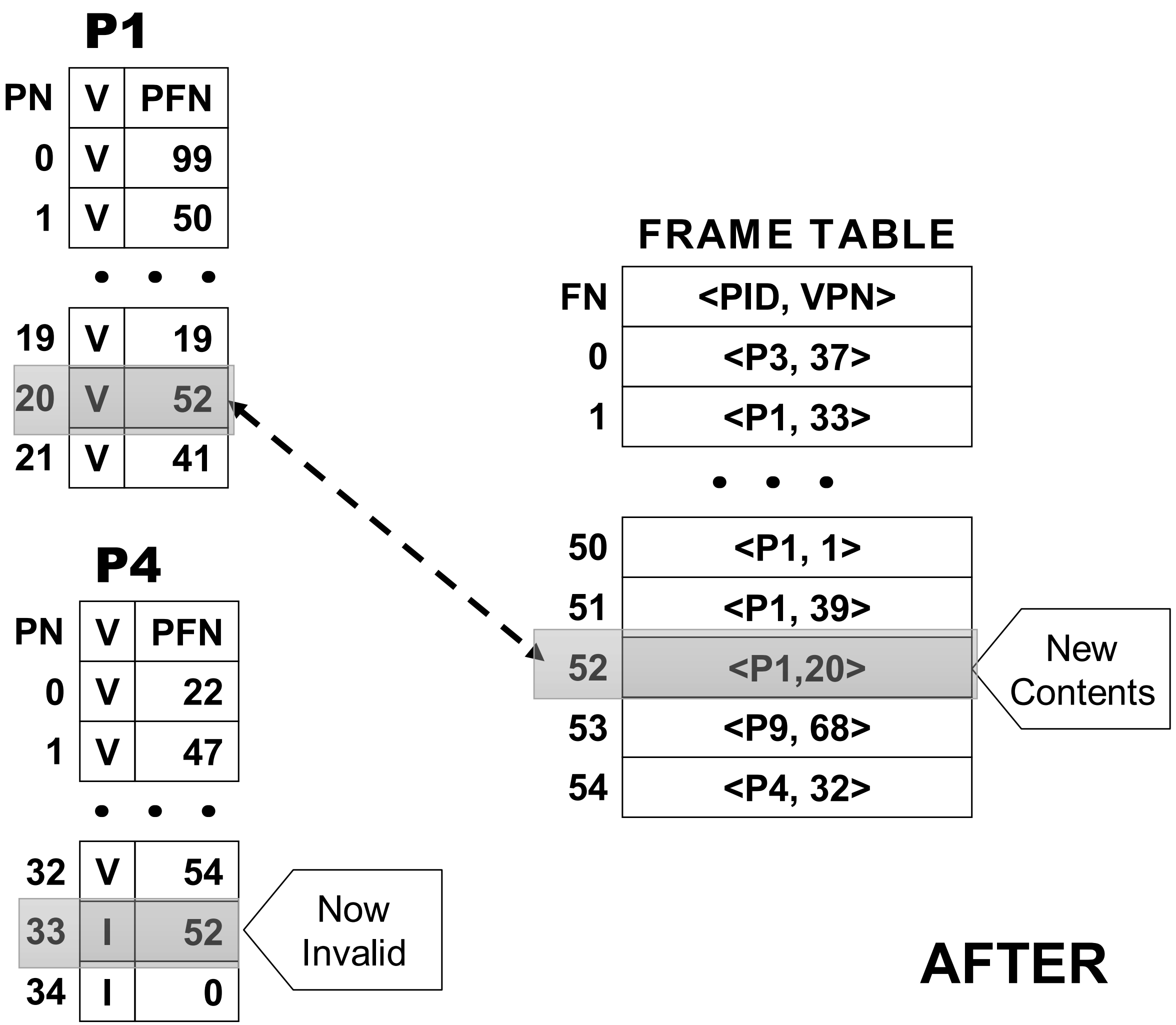
FN	<PID, VPN>
0	<P3, 37>
1	<P1, 33>
	• • •
50	<P1, 1>
51	<P1, 39>
52	<P4, 33>
53	<P9, 68>
54	<P4, 32>

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

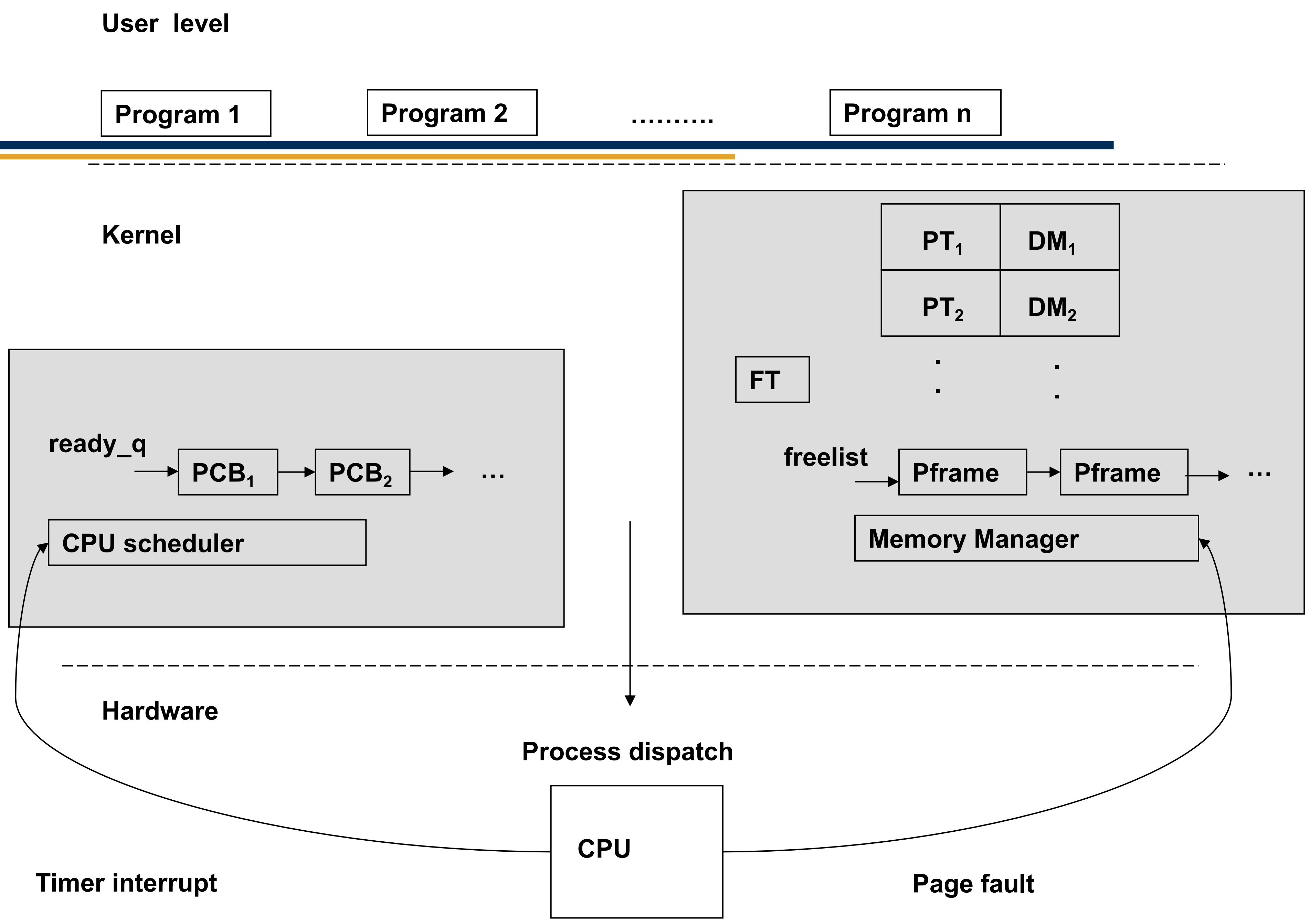
With paged memory management there can be…

A. External fragmentation

B. Internal fragmentation

C. No fragmentation

D. Both internal and external fragmentation



User level

Program 1

Program 2

……….

Program n

Kernel

PT_1

DM_1

PT_2

DM_2

FT

ready_q

PCB_1

PCB_2

…

CPU scheduler

freelist

Pframe

Pframe

…

Memory Manager

Hardware

Process dispatch

CPU

Timer interrupt

Page fault