Georgia Tech Computer Science

CS2200
Systems and Networks
Spring 2024

Lecture 15:
Virtual Memory

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

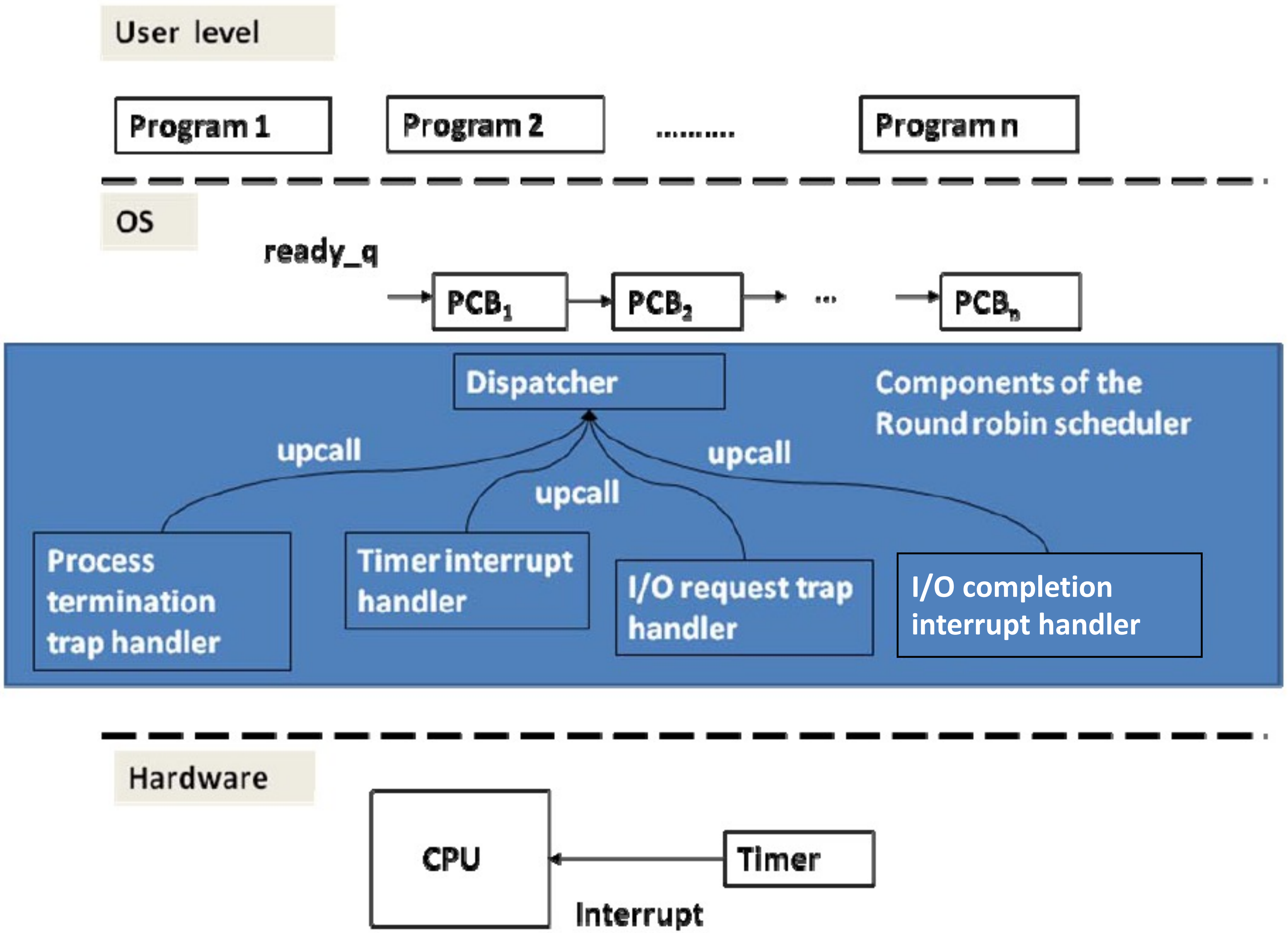
Lecture slides adapted from Bill Leahy and Charles Lively of Georgia Tech

Today's Agenda

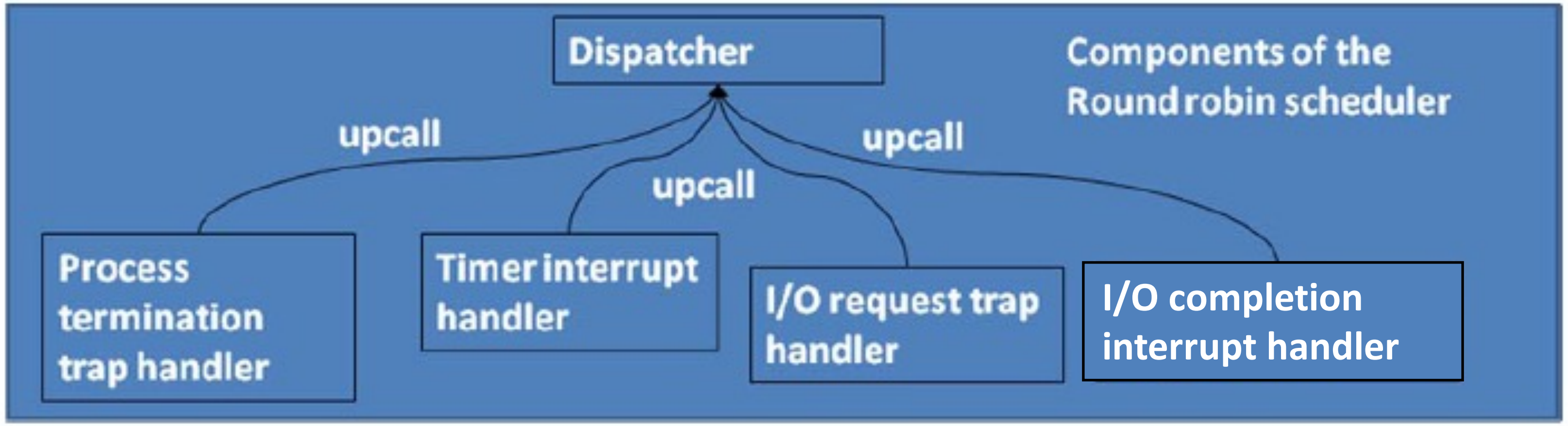
- Wrap up scheduling
- Starting Part II – Memory System
  - Chapters 7,8

Implementing the process abstraction

- The OS uses timer interrupts to trigger context switches
- You can think of the scheduler/dispatcher as part of the interrupt handlers(!)
- All of the relevant data structures are initialized by the OS initialization code before interrupts are turned on


User level
Program 1
Program 2
....
Program n
OS
ready_q
PCB1
PCB2
...
PCBn
Dispatcher
Components of the
Round robin scheduler
upcall
upcall
upcall
Process
termination
trap handler
Timer interrupt
handler
I/O request trap
handler
I/O completion
interrupt handler
Hardware
CPU
Timer
Interrupt

Who does what in the OS



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 trap;

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 trap;

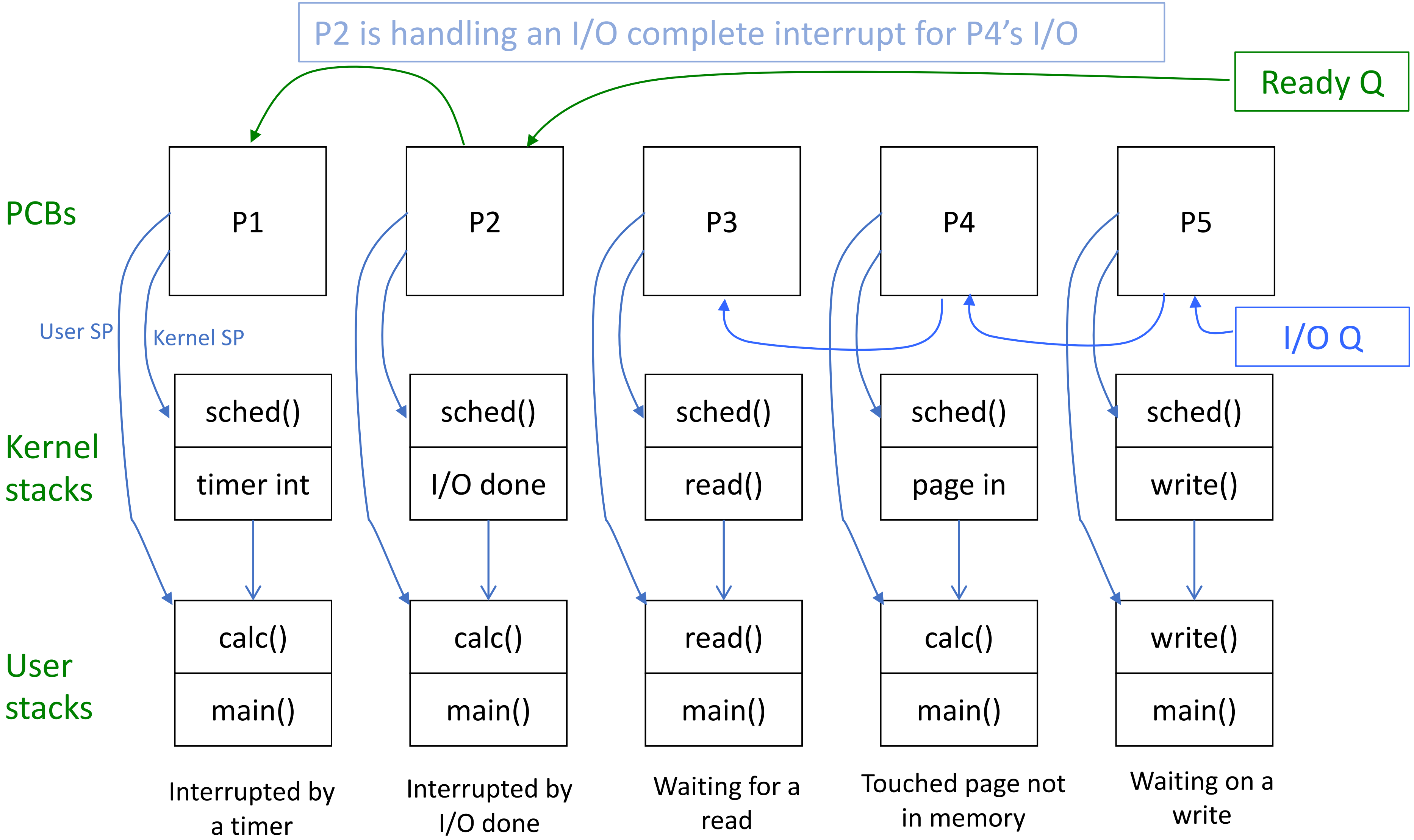
Our Example Program

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



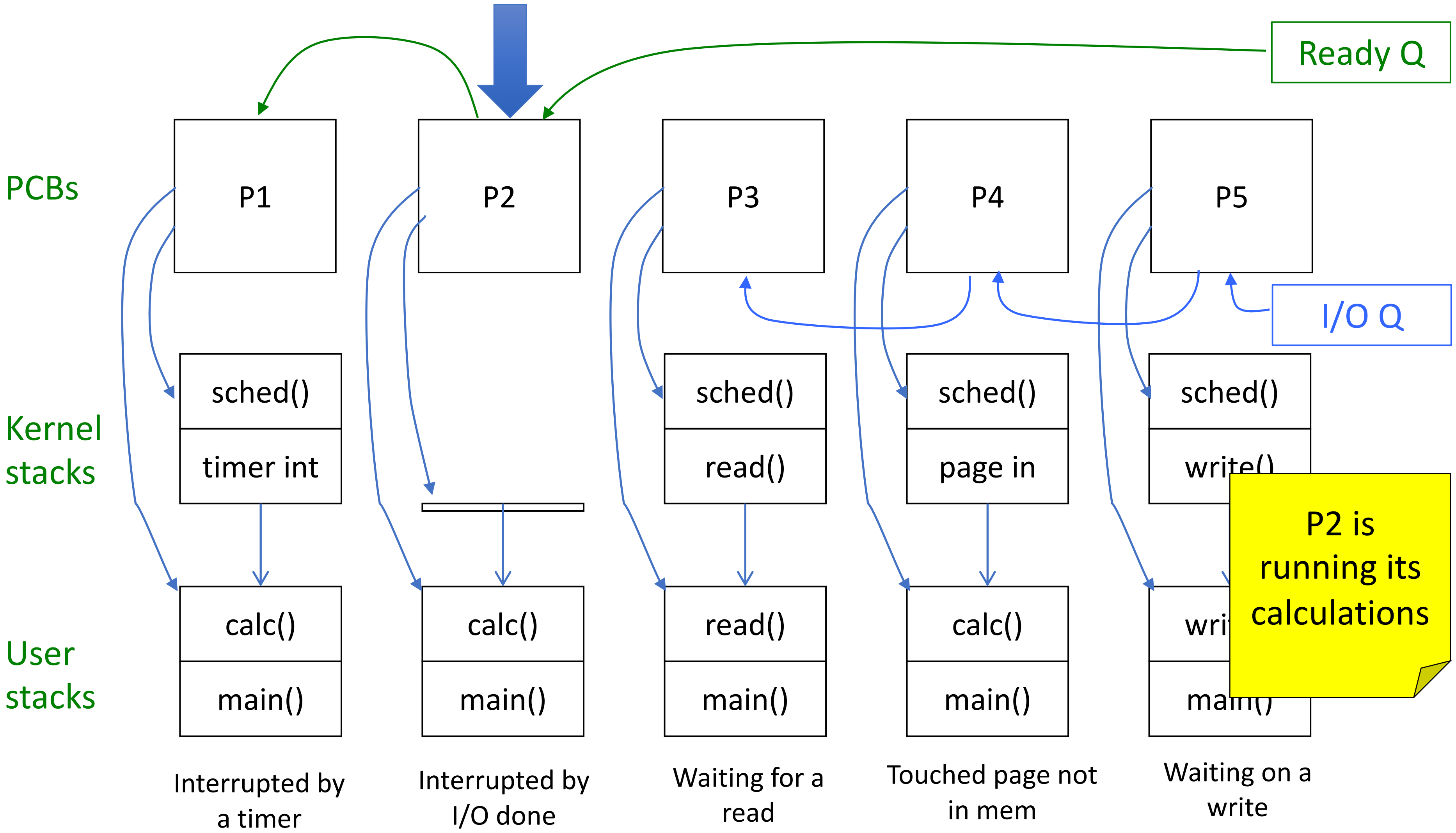
An example using the previous diagram

- 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 queue (and off the I/O queue)
- P2's interrupt handler calls the scheduler
  - Assume the scheduler decides that the winner is P4
    - Remember: P4's return to the ready queue 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 and kernel stack
  - Return using the currently loaded state (i.e. return to P4)

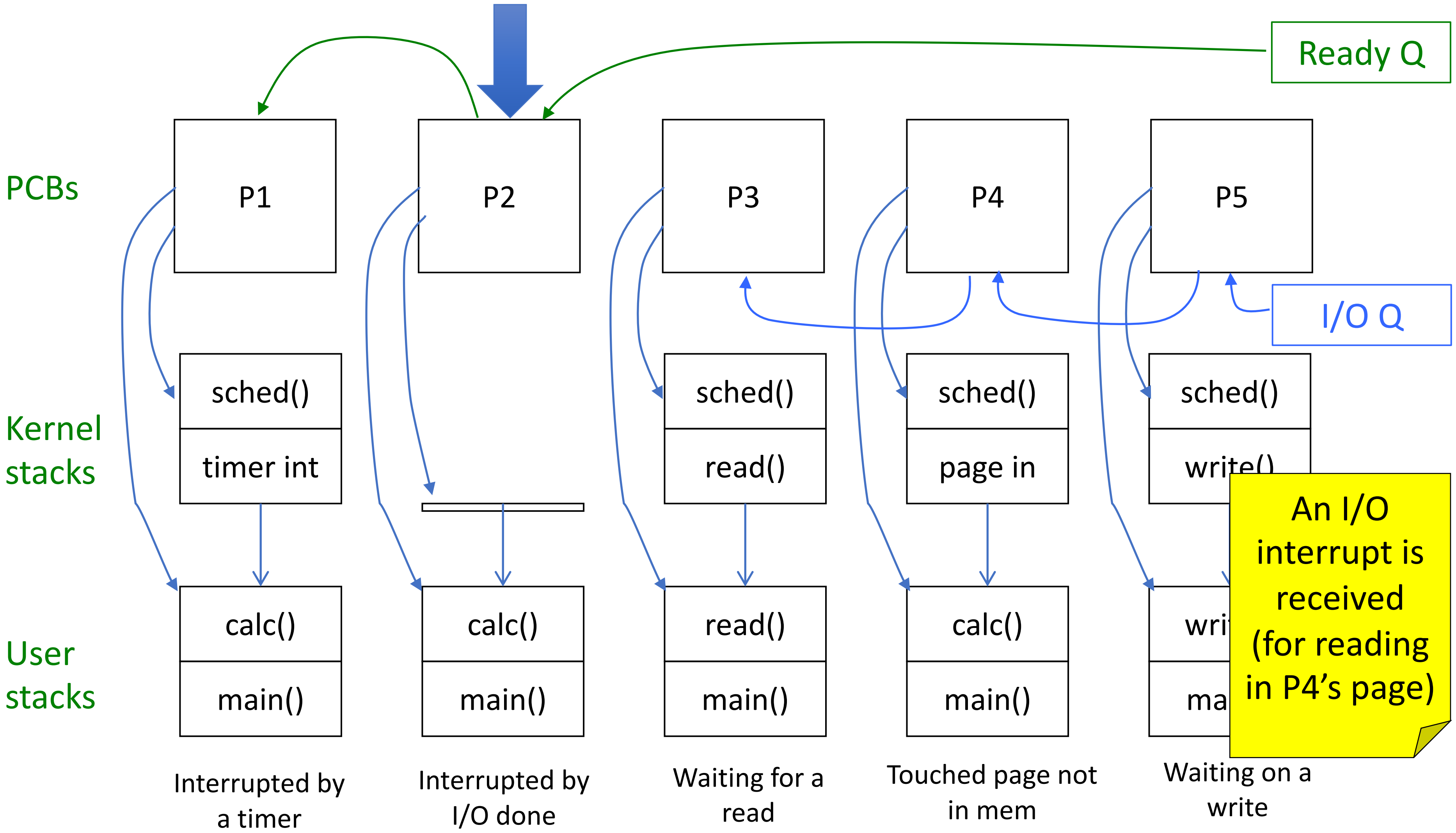
Let’s do the example again

- 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 sleight-of-hand that switches us from P2 to P4
- We start the example with P2 running its calculations in user state

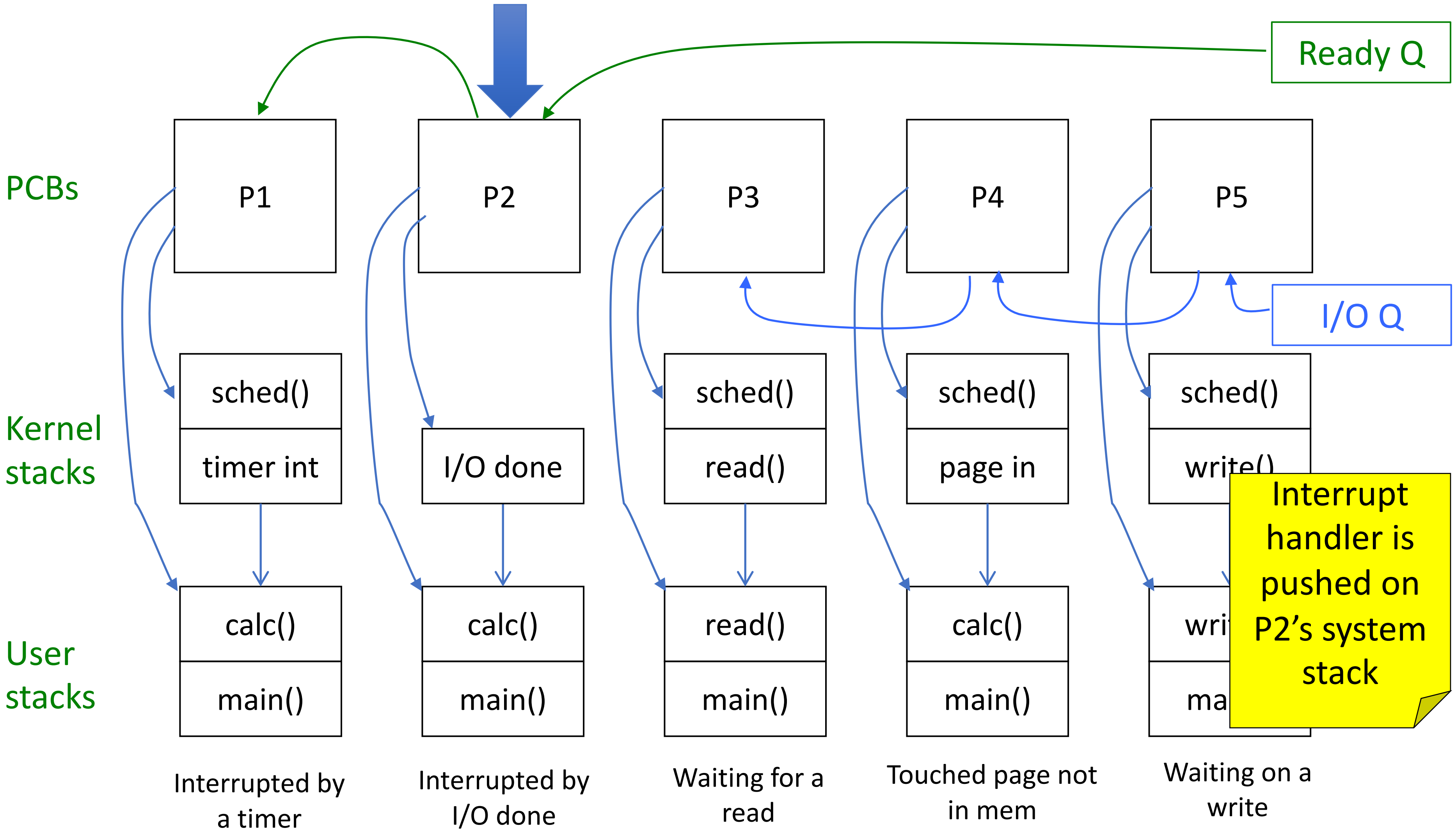
Process example in detail!



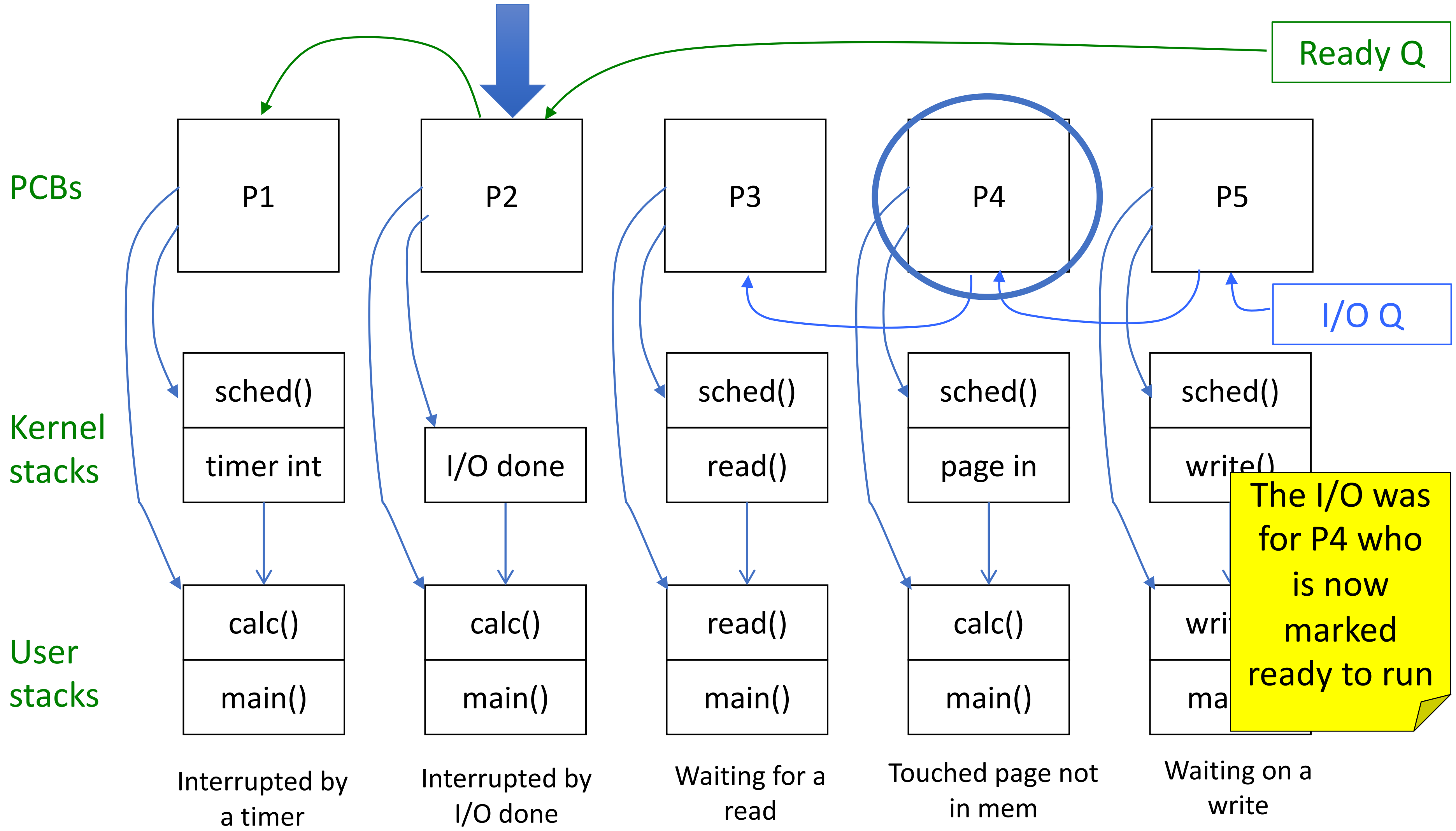
Process example



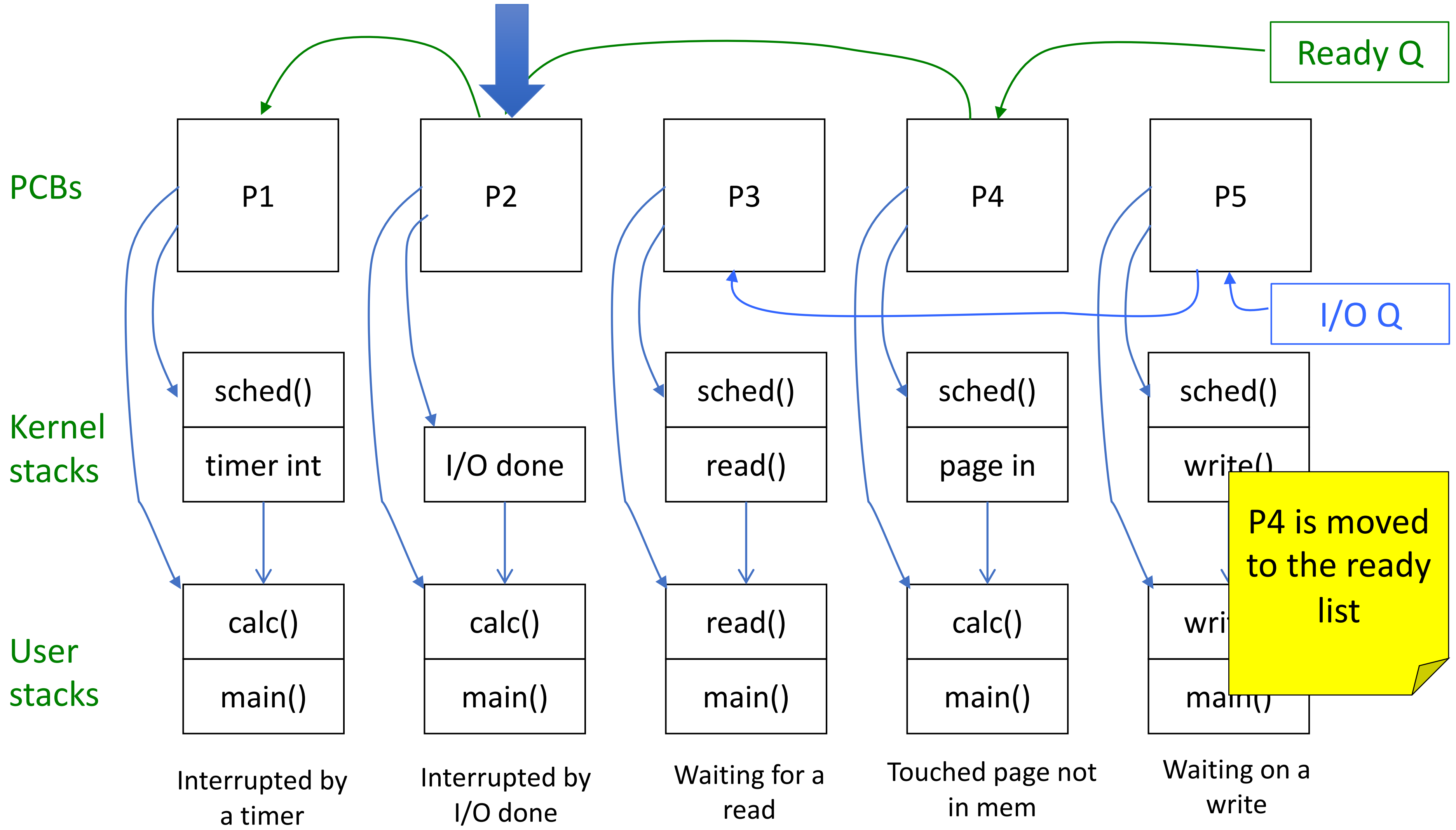
Process example



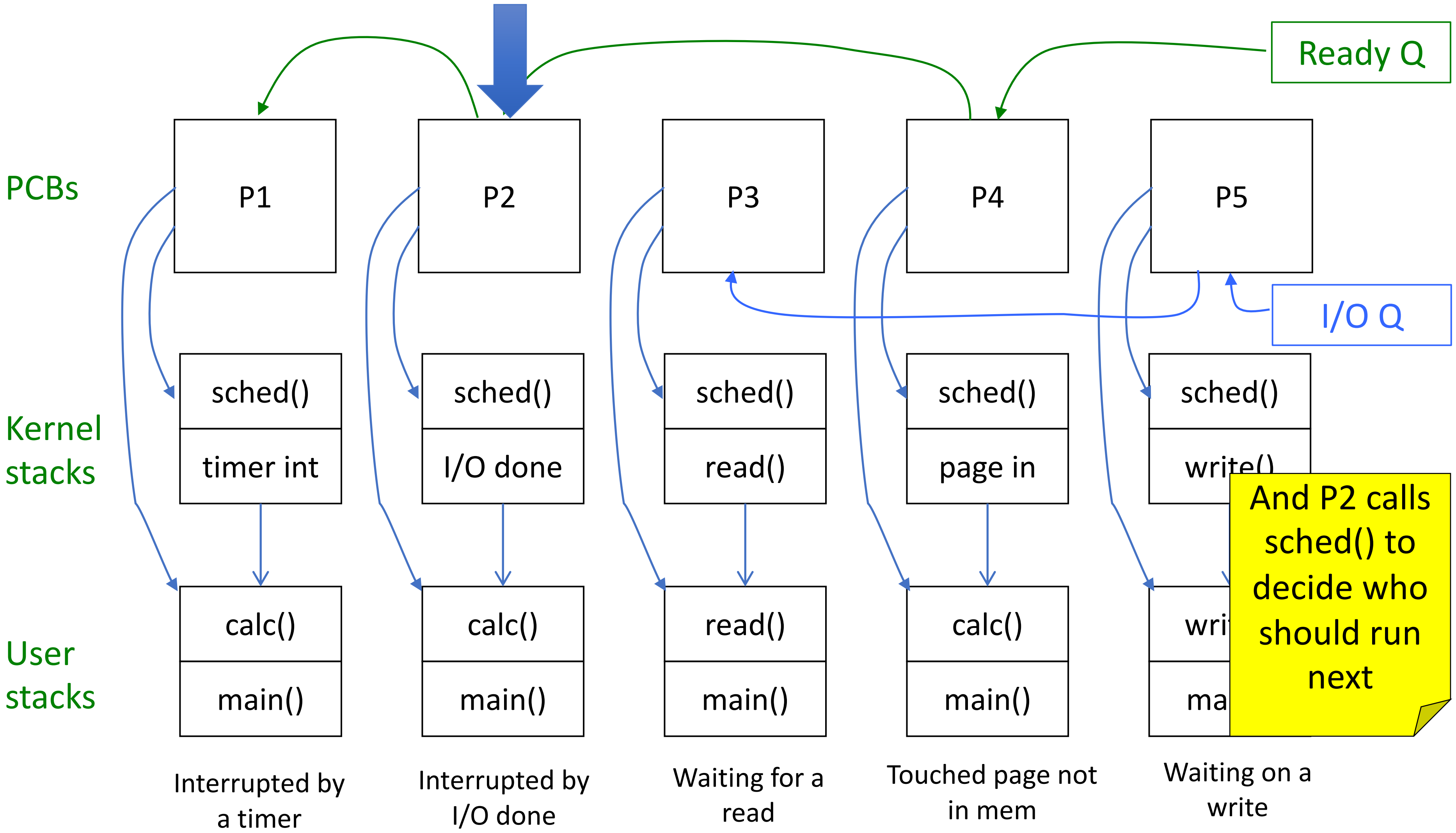
Process example



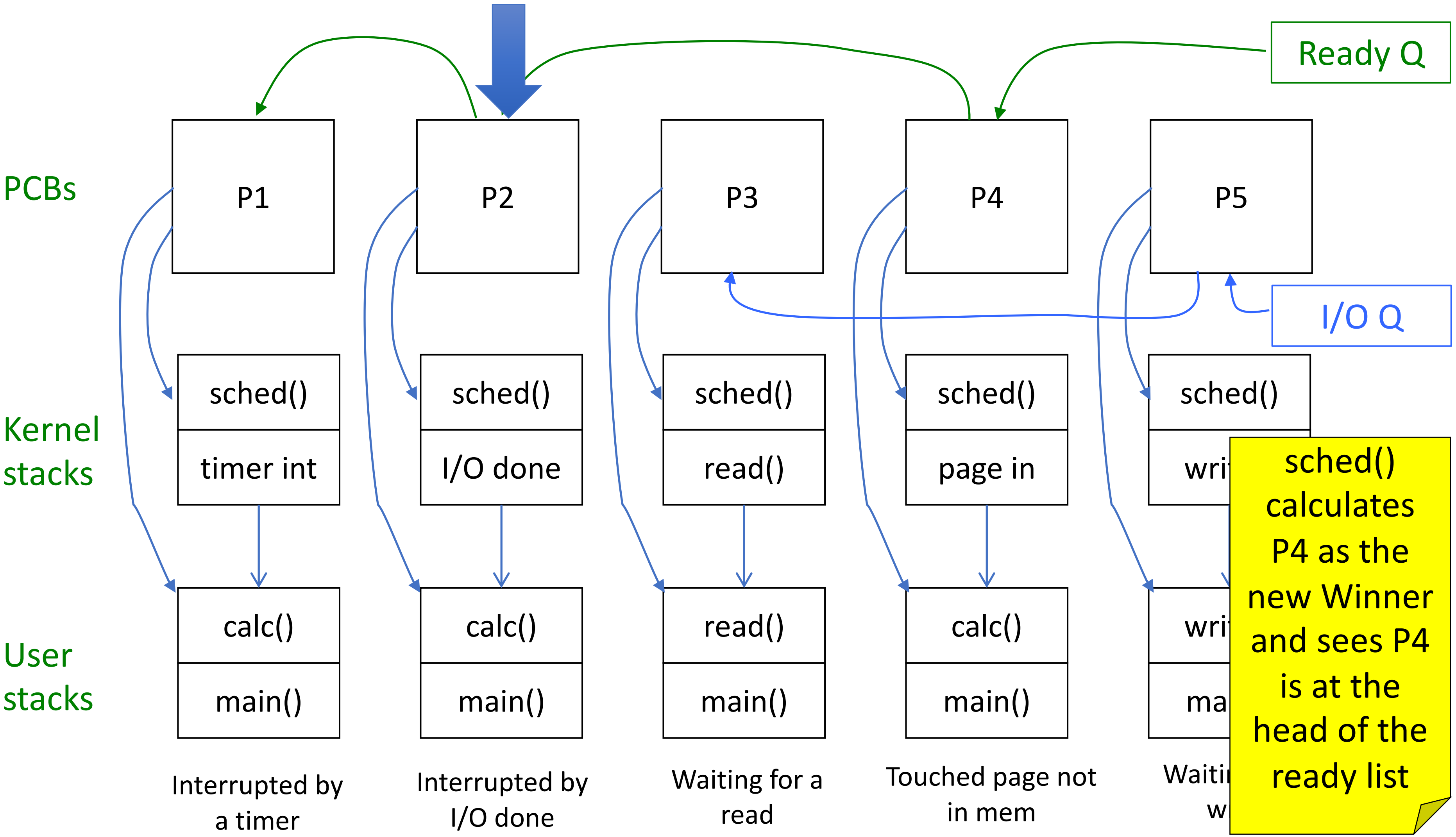
Process example



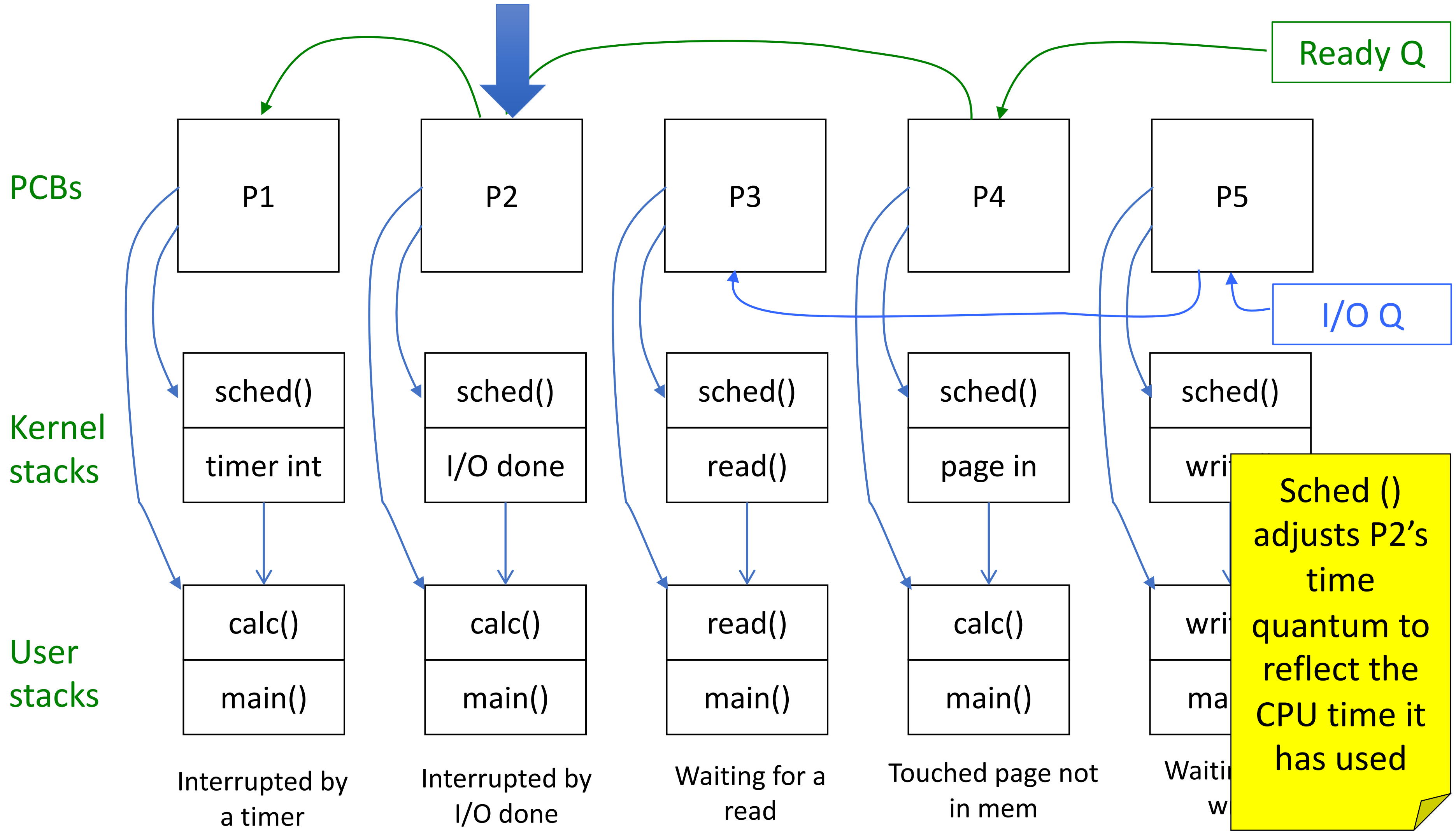
Process example



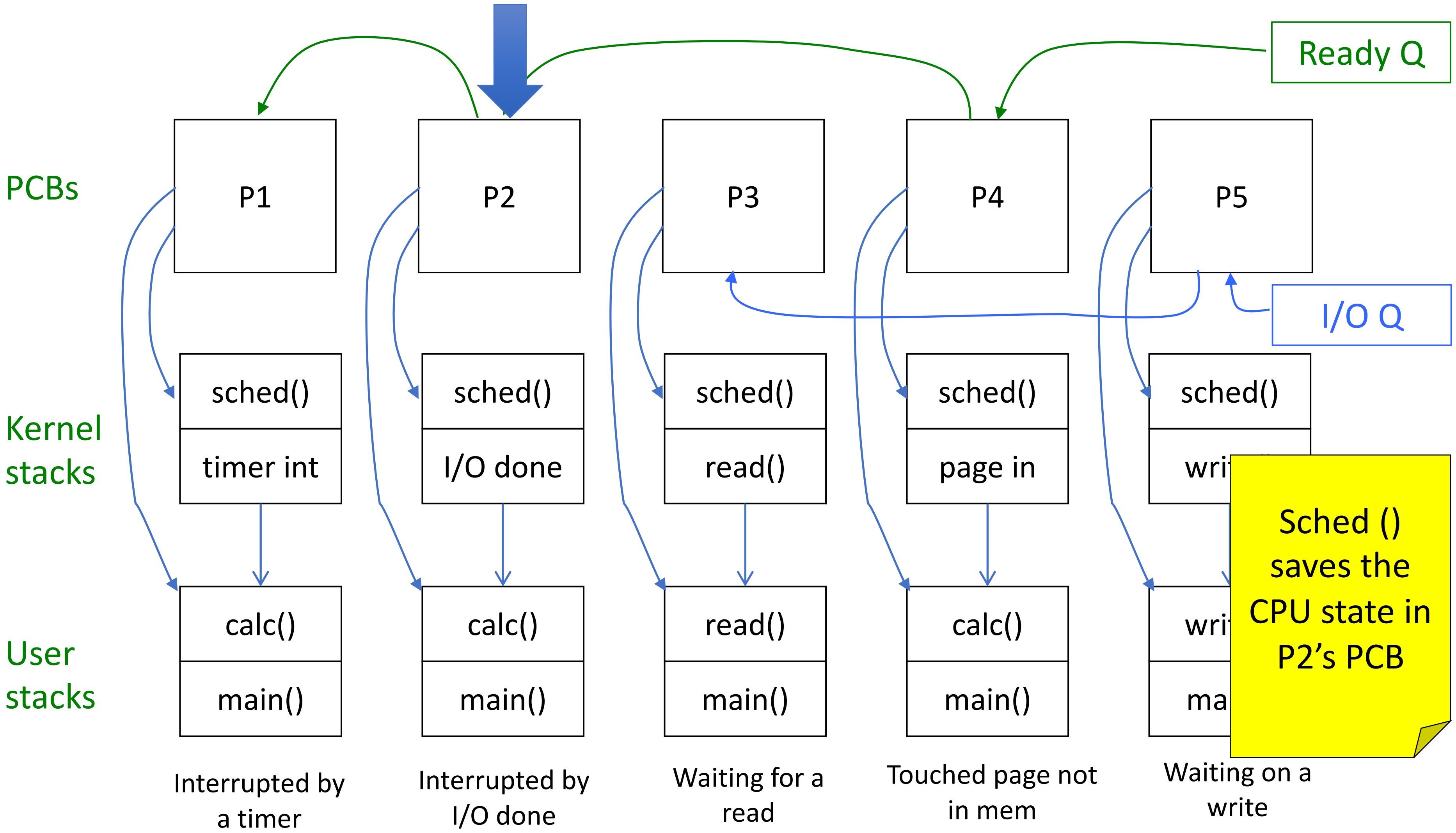
Process example



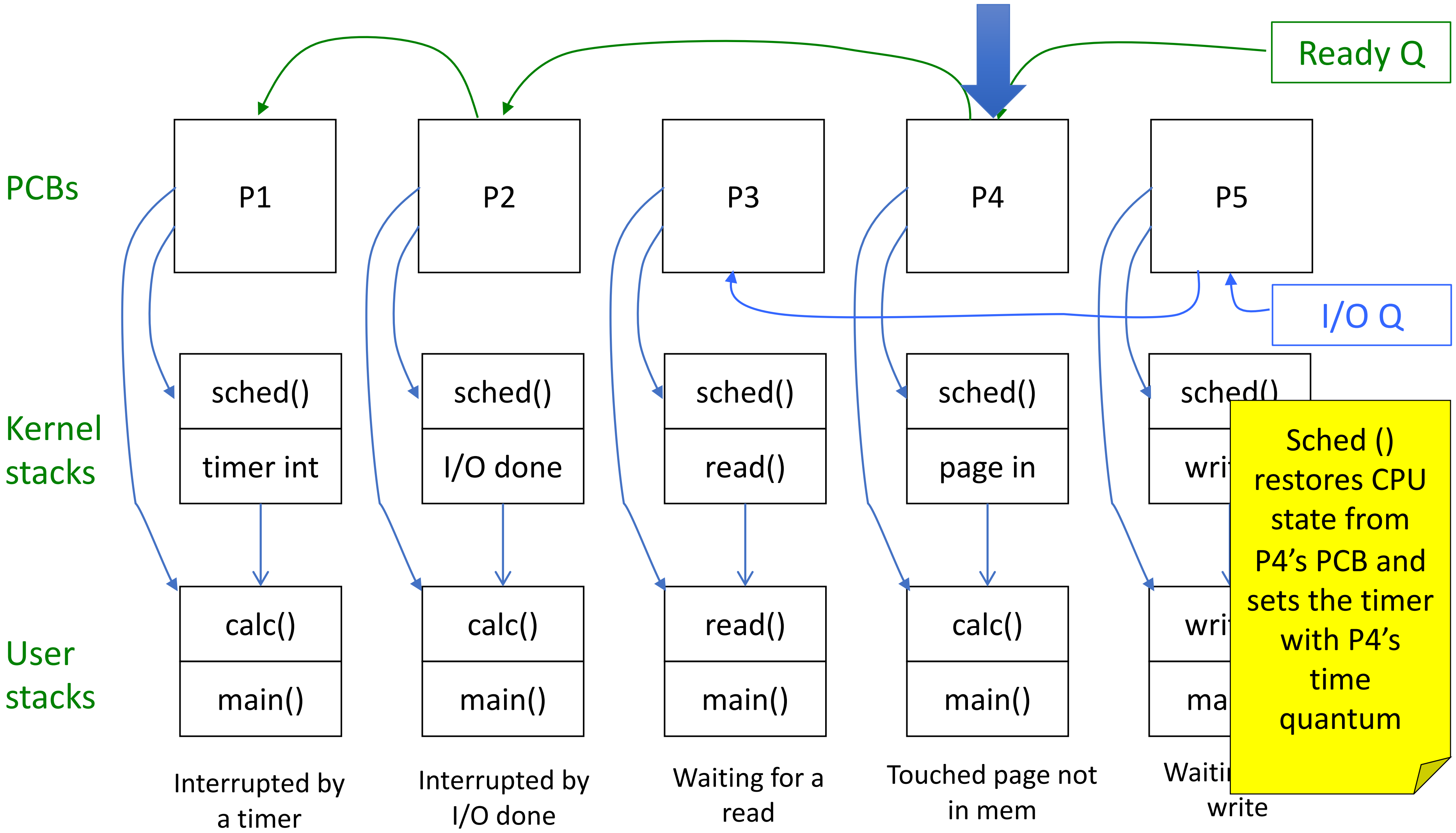
Process example



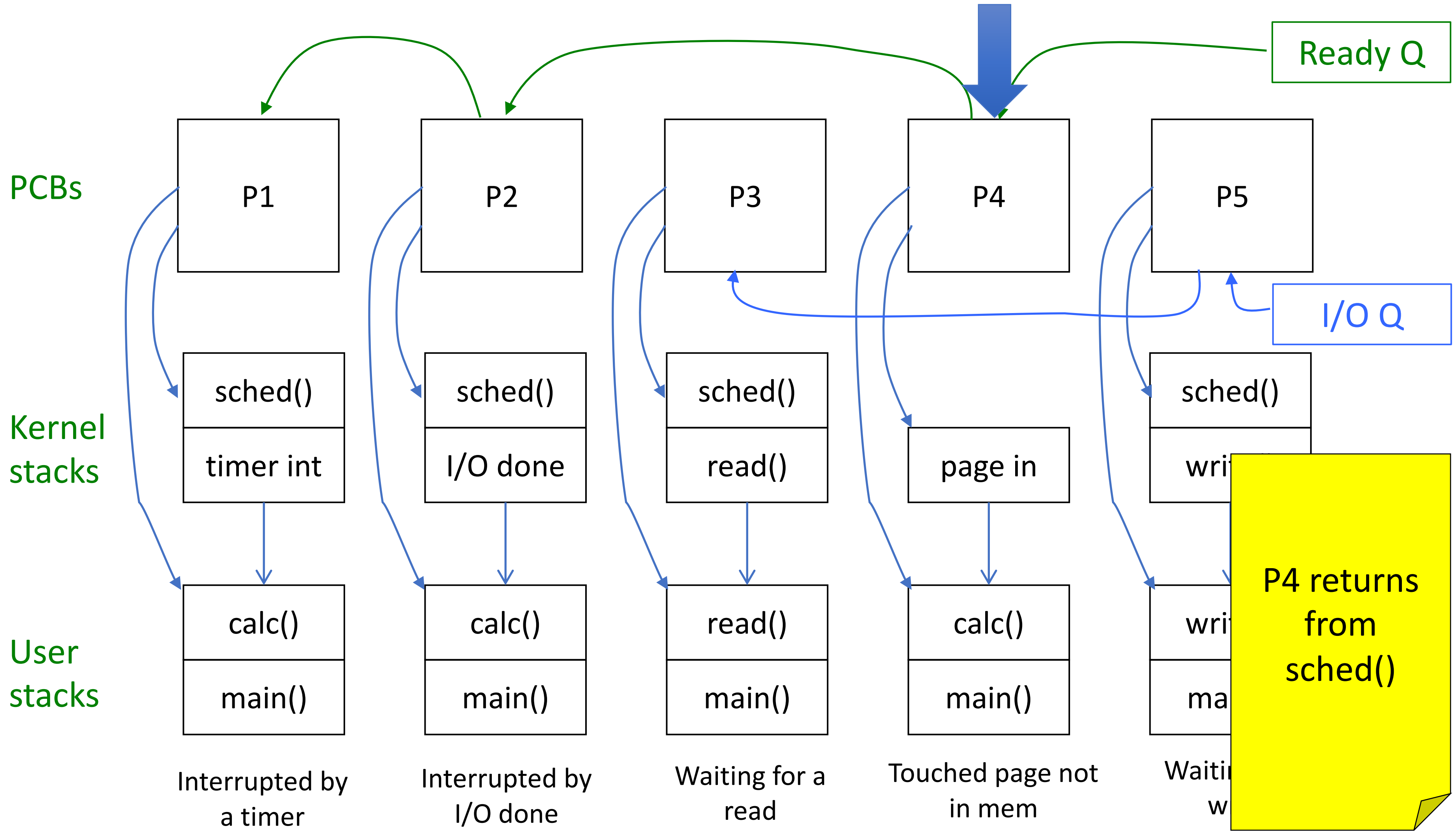
Process example



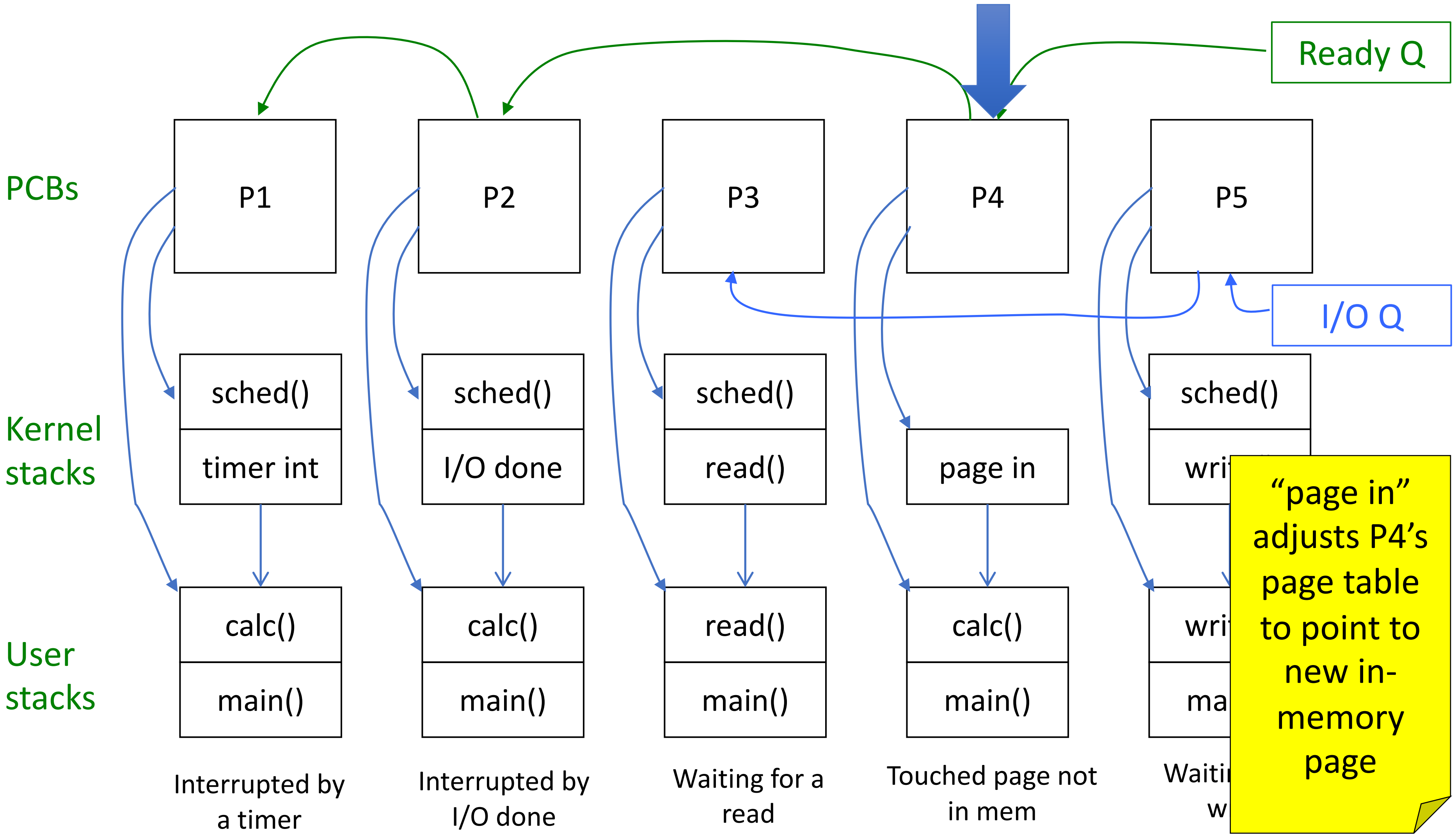
Process example



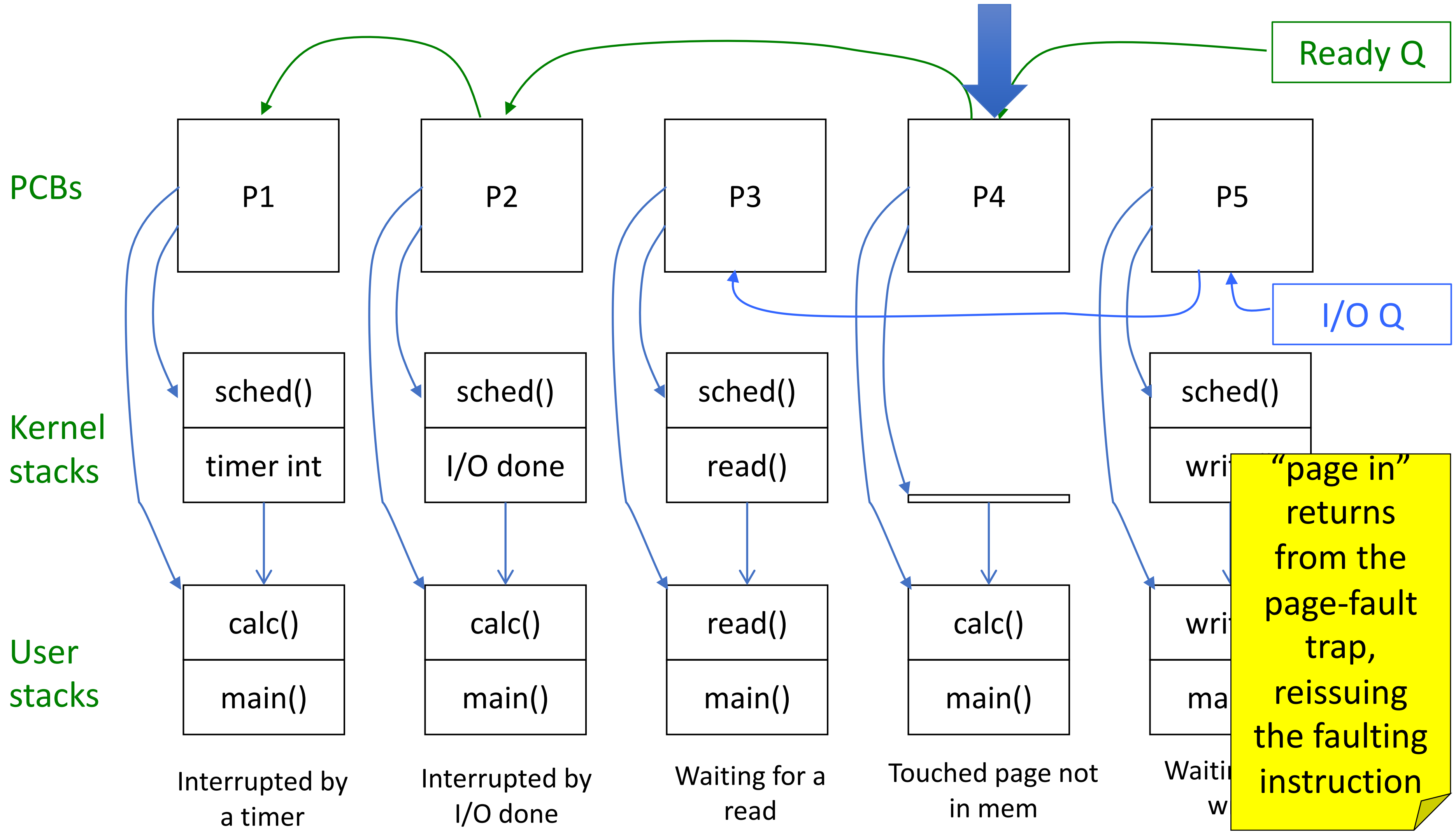
Process example



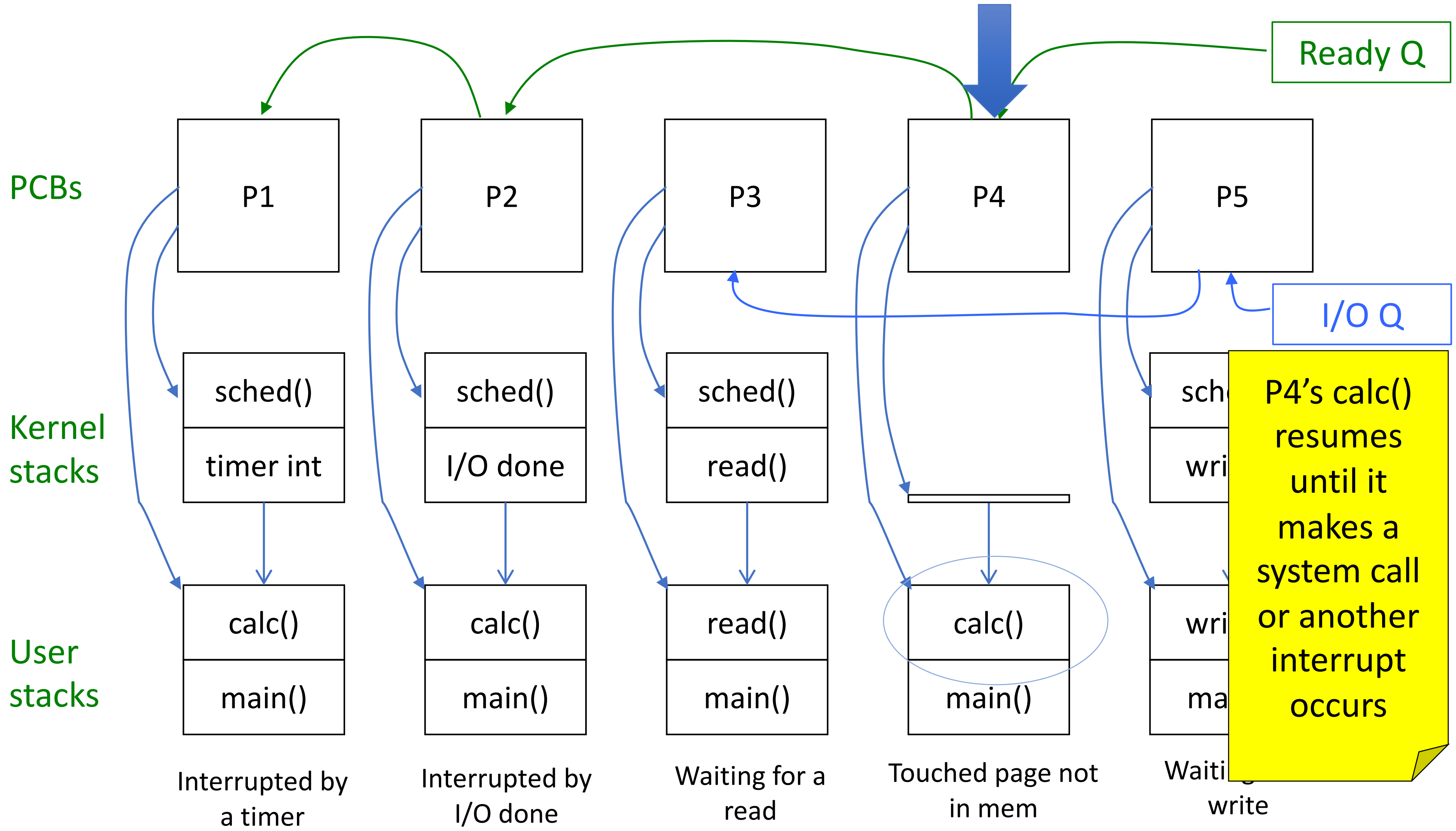
Process example



Process example



Process example



Session ID: dag2200

A preemptive scheduler…

- 0% A. Can only be implemented with a timer interrupt
- 0% B. Can only be implemented with I/O completion interrupt
- 0% C. Can only be implemented with a system call trap
- 0% D. Can be implemented with any type of interrupt

Session ID: dag2200

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

0% A. The system stack

0% B. The PCB for that process

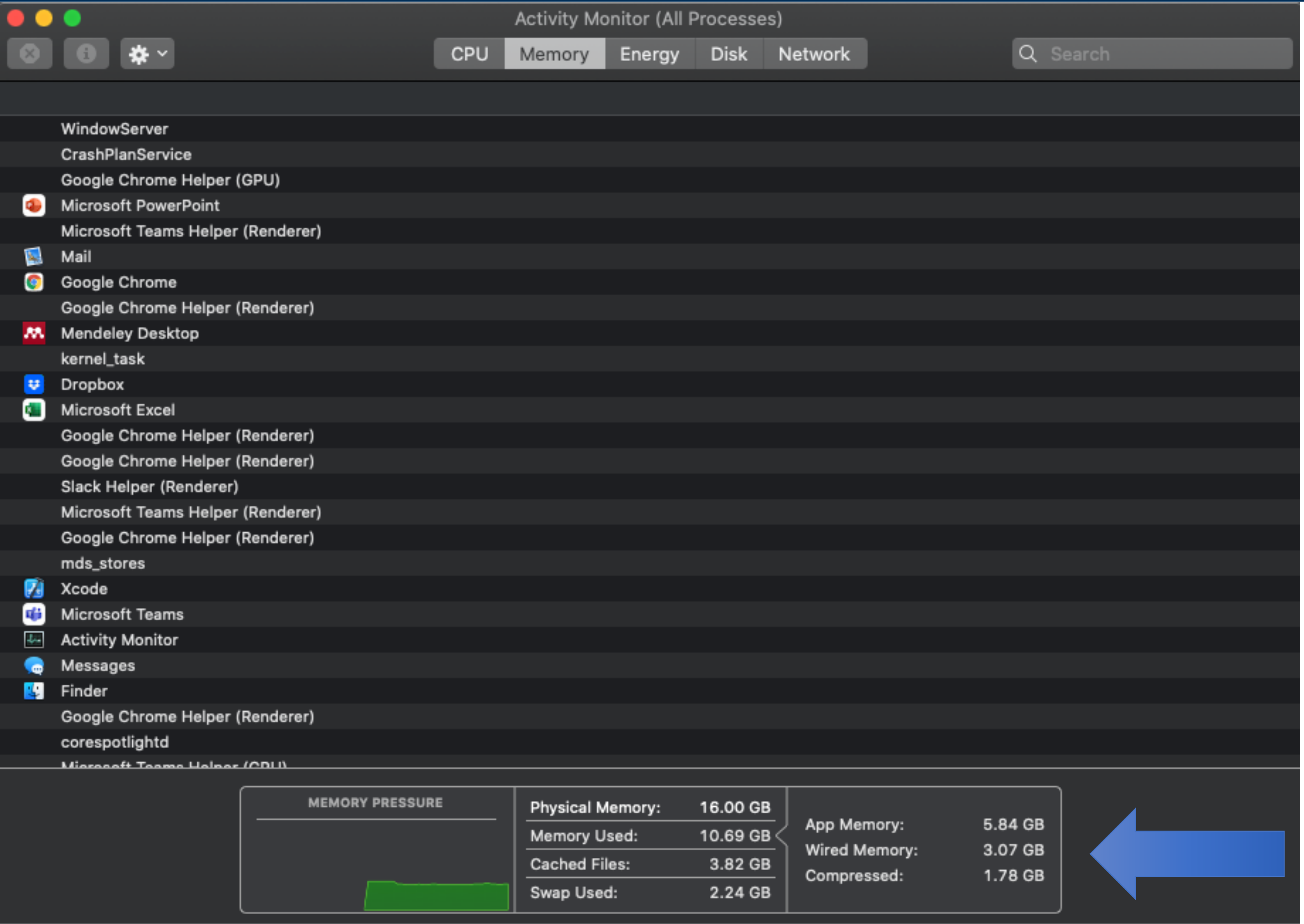
0% C. The user stack

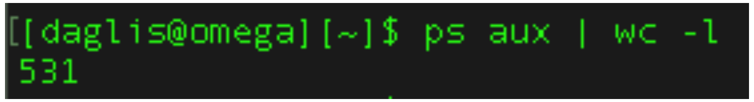
0% D. The heap space of the process

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





531 processes!

10.69GB of 16GB memory in use

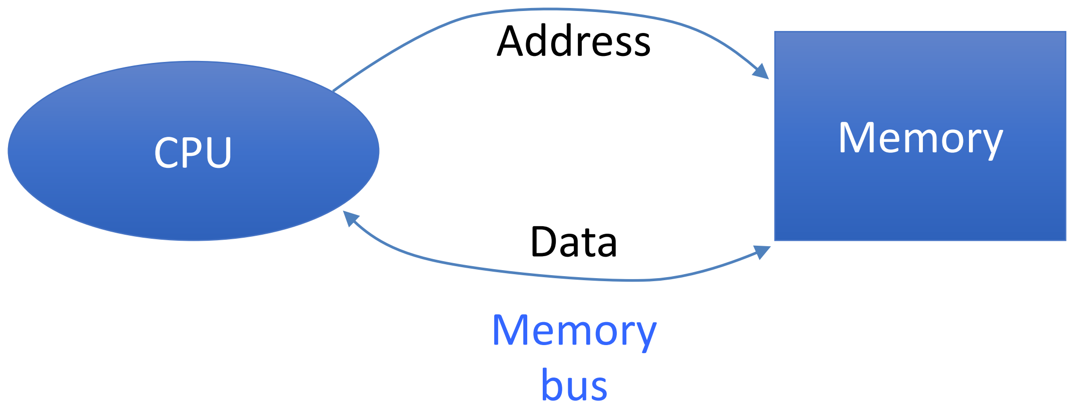
Goals of memory management

What functionalities do we want to provide?

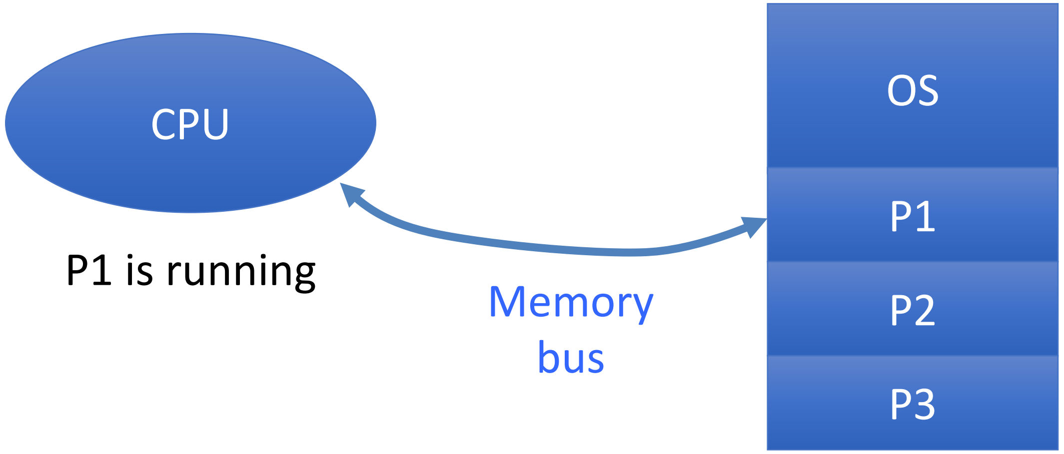
- Improved resource utilization
- Independence and protection
- Liberation from resource limitations
- Sharing of memory by concurrent processes

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

Nothing in LC-2200 to manage memory

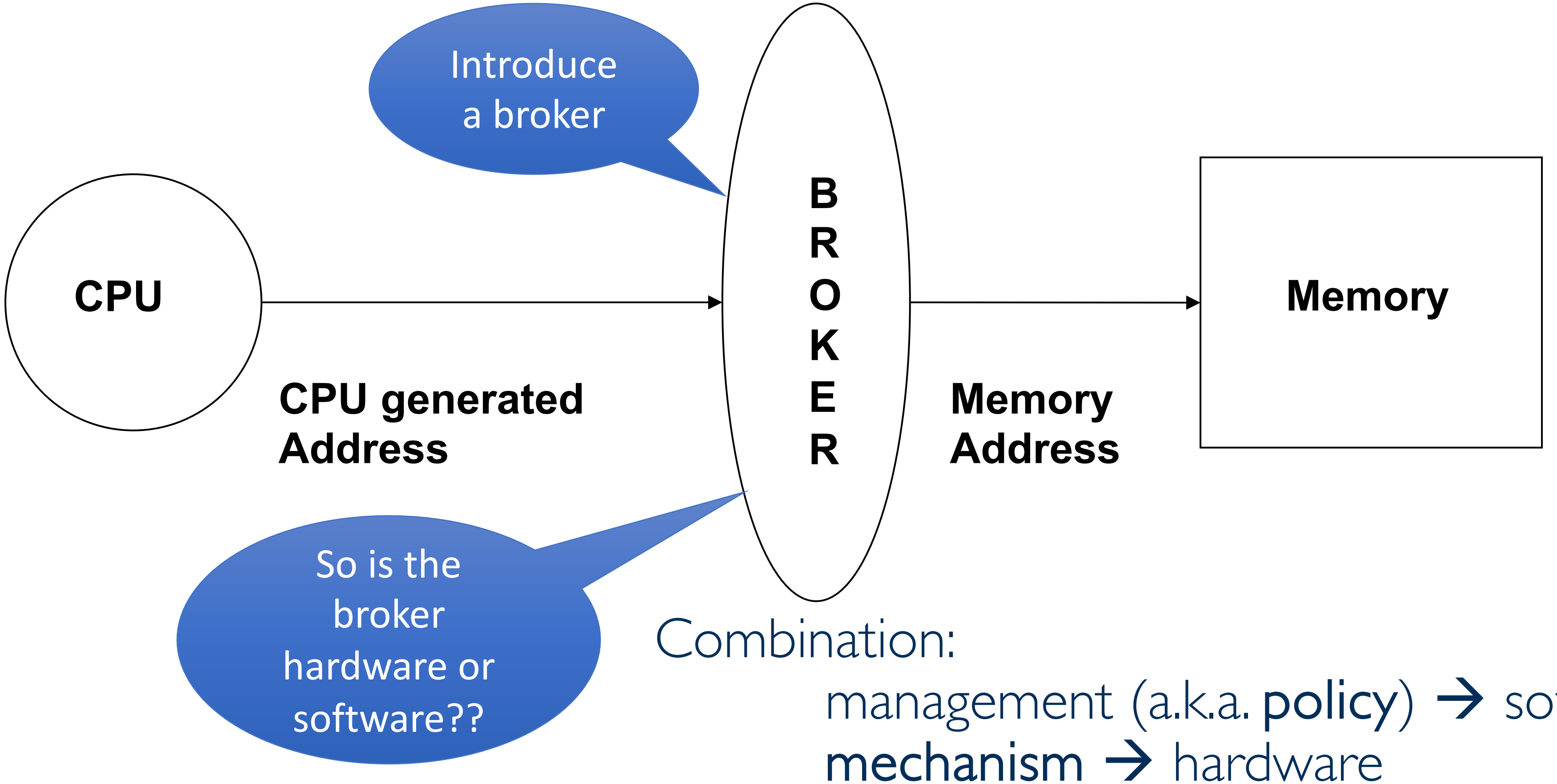


What's missing in LC-2200?



- 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…

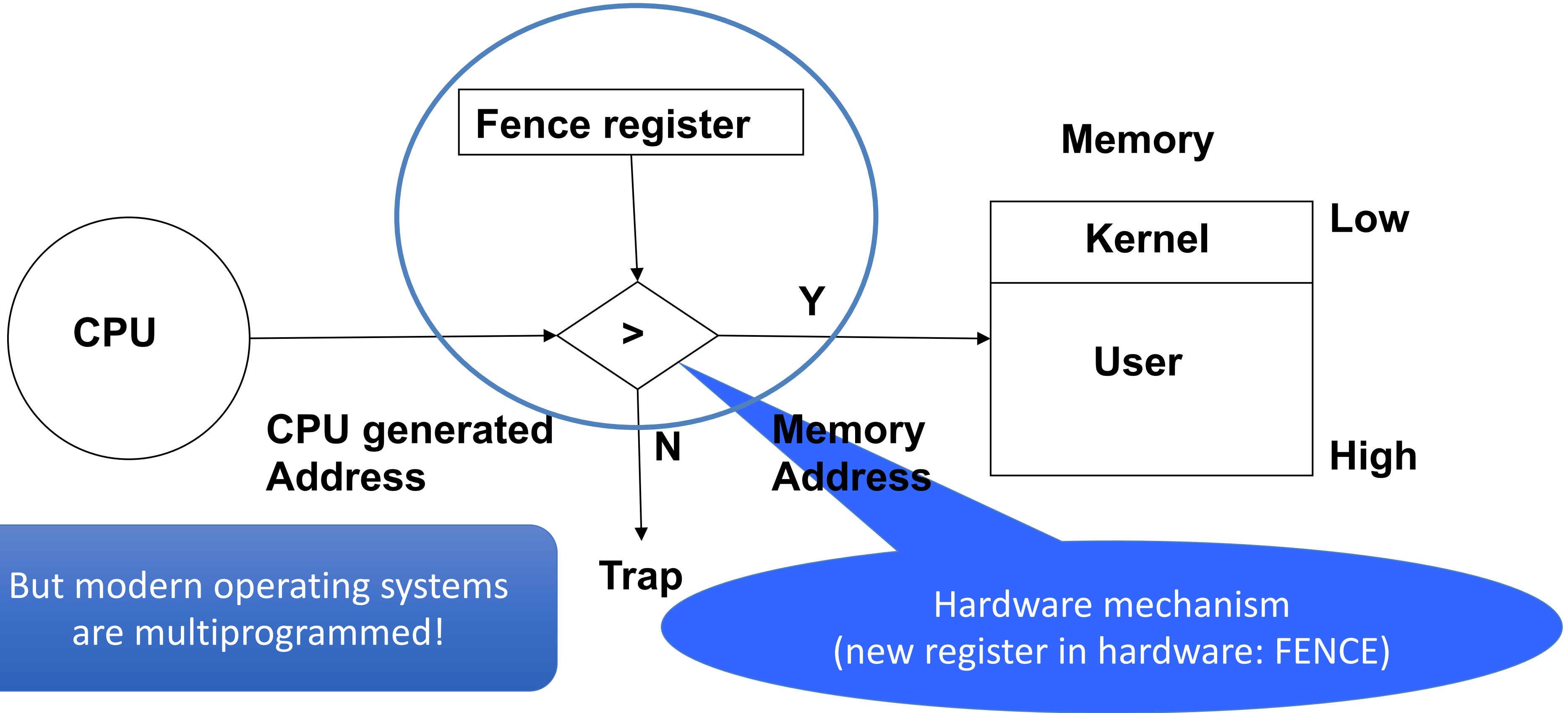


Combination:

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

Simple management

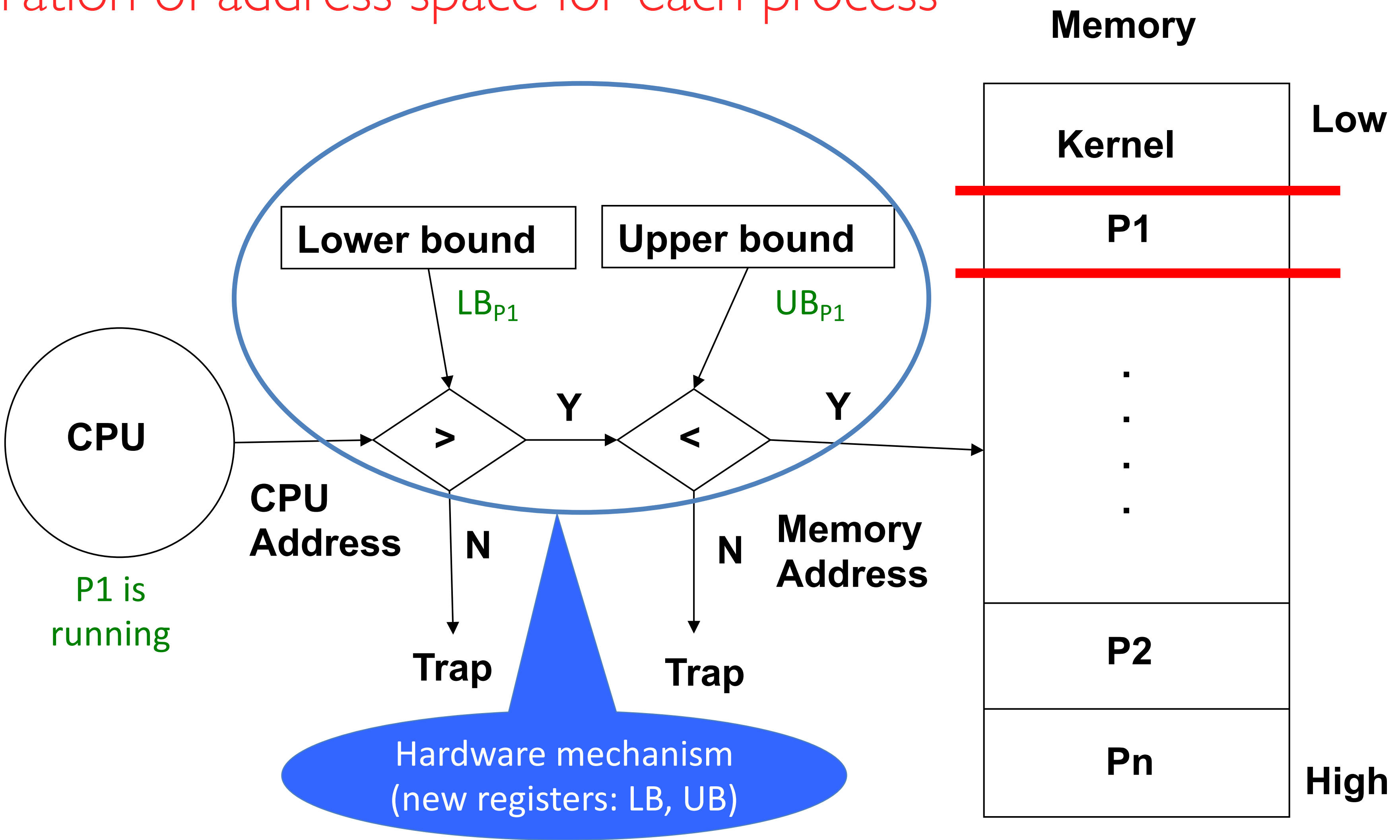
- One user process at a time
- Separation needed between OS and user program



But modern operating systems are multiprogrammed!

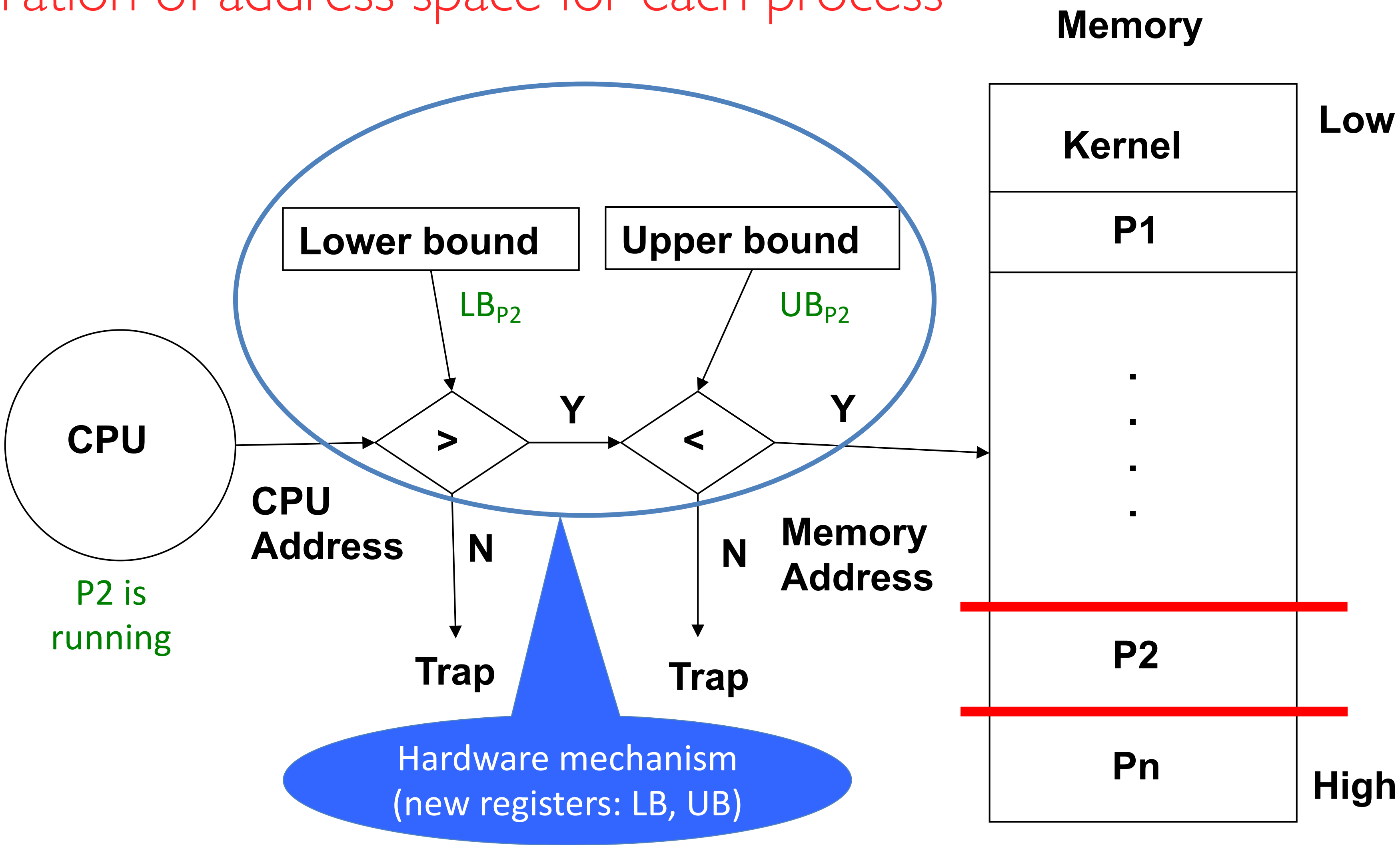
Multiprogrammed OS

- Separation of address space for each process



Multiprogrammed OS

- Separation of address space for each process



Session ID: dag2200

What needs to happen…

…to ensure that LB and UB correspond to the currently running process?

- 0% A. Restore LB and UB to the system stack
- 0% B. Restore LB and UB to values defined in the source code file
- 0% C. Restore LB and UB to values found in the PCB
- 0% D. Calculate LB and UB from the process-id

PCB

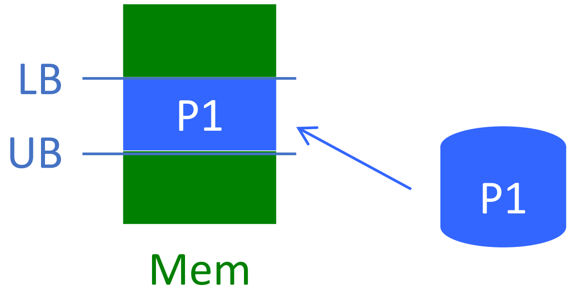
```
enum  state_type {new, ready, running,
                  waiting, halted};
typedef struct control_block_type {
  enum state_type state;
  address PC;
  int reg_file[NUMREGS];
  struct control_block *next_pcb;
  int priority;
  address memory_footprint; ????
  ….
  ….
} control_block;
```

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;
```

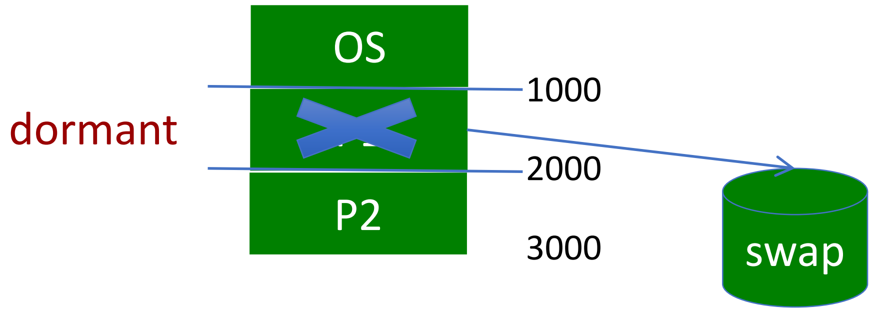
"Management" by the OS

- 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



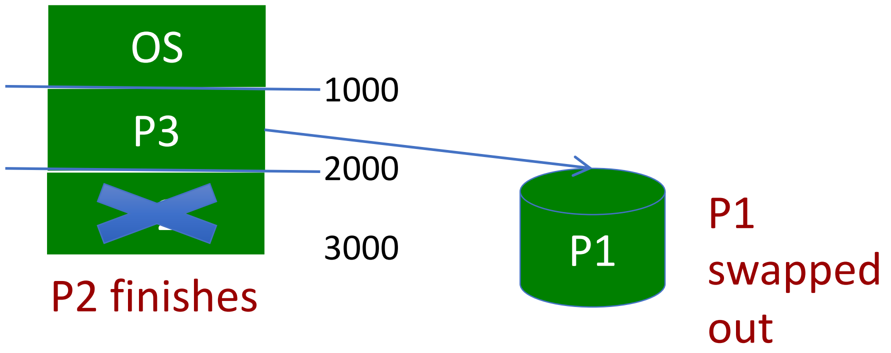
Limits of "bounds register" mechanism?

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



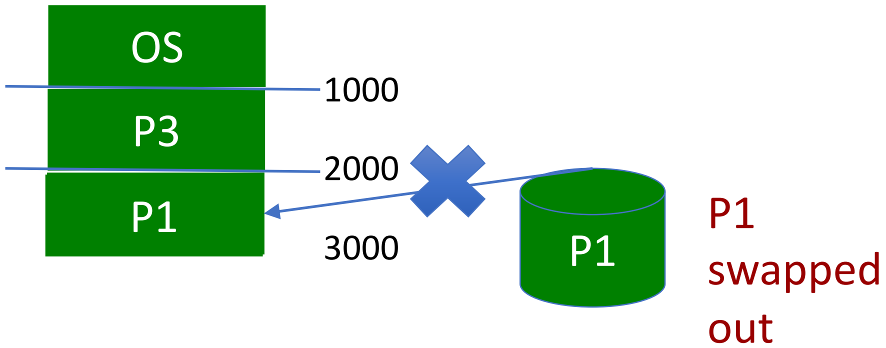
Limits of "bounds register" mechanism?

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



Limits of "bounds register" mechanism?

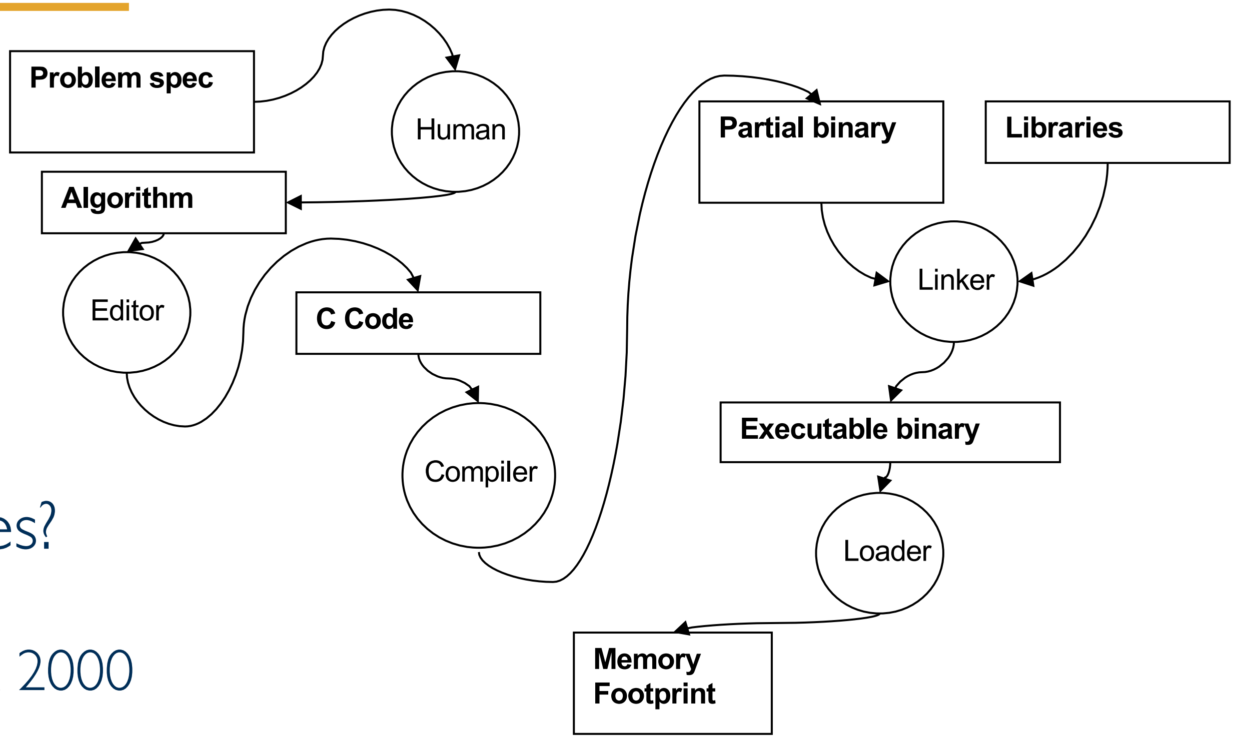
- Relocation isn't possible
- Swapping in is a problem



- ✓ 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

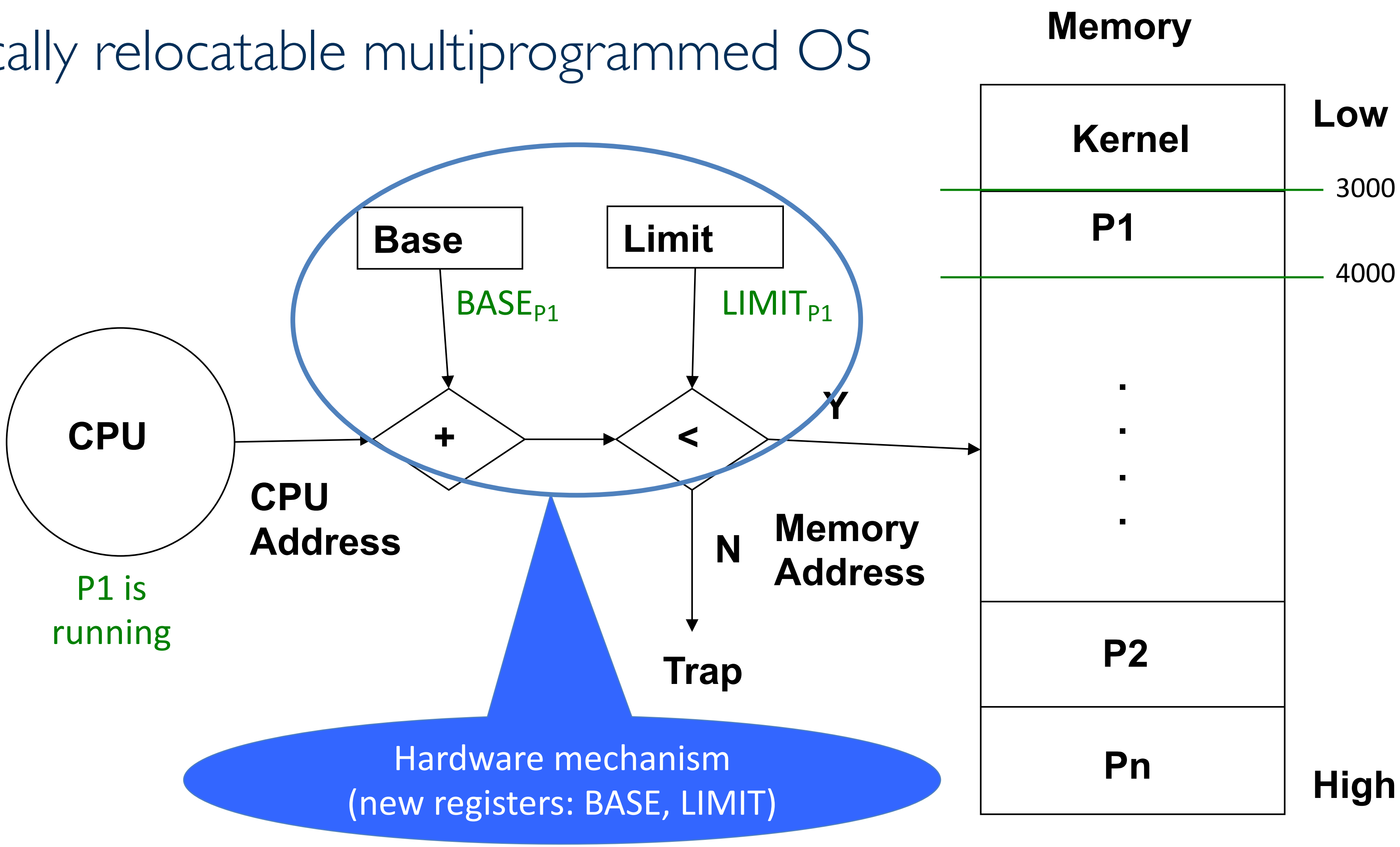


Can we make P1 dynamically relocatable?

- 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

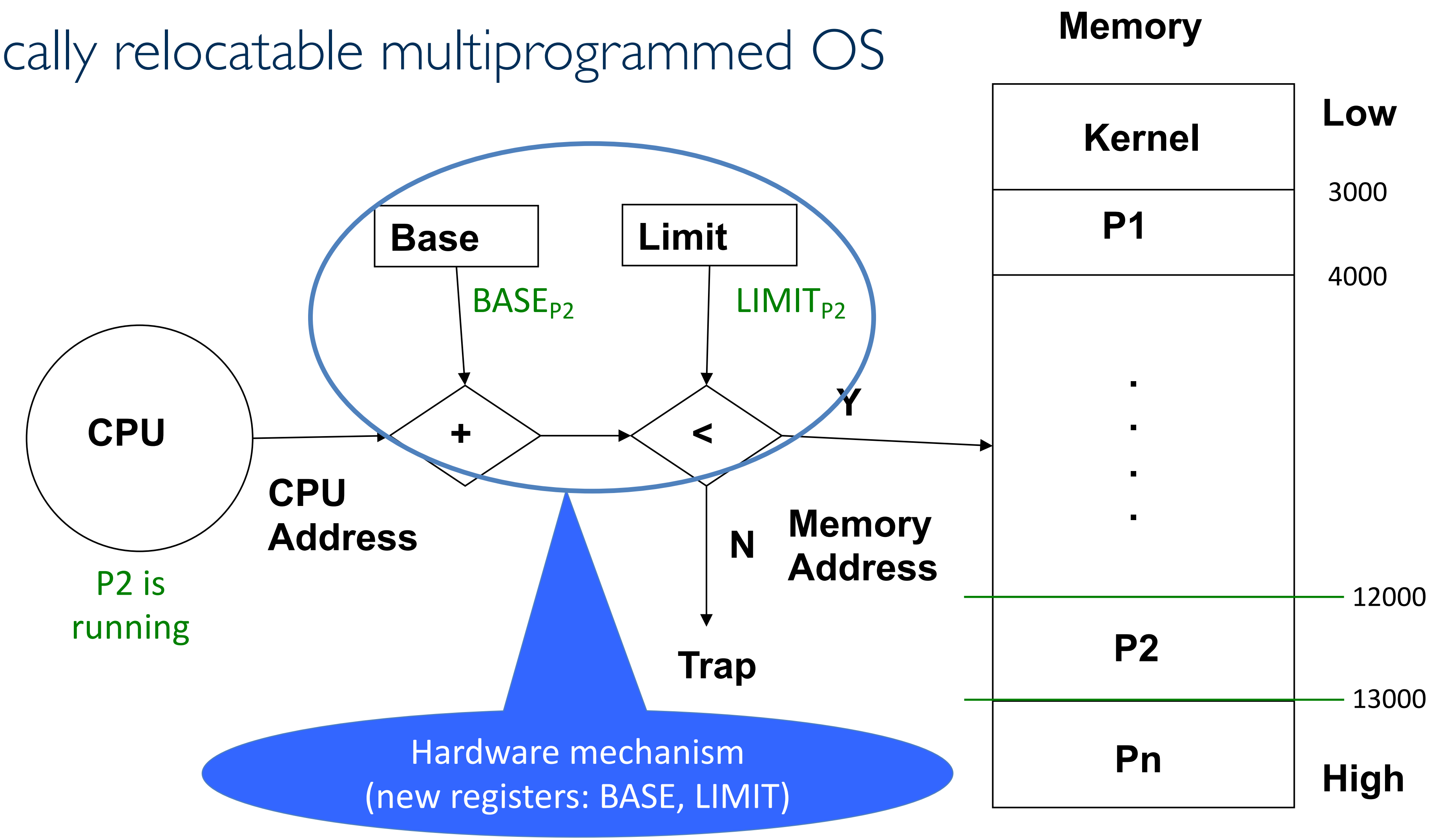
But there's a hardware solution…

- Dynamically relocatable multiprogrammed OS



But there's a hardware solution…

- Dynamically relocatable multiprogrammed OS



PCB

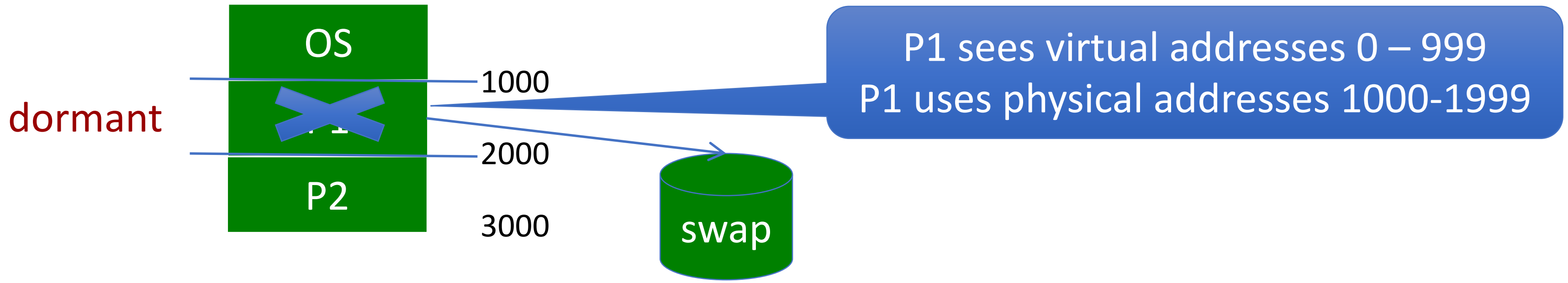
```
enum  state_type {new, ready, running,
                                          waiting,
  halted};
typedef struct control_block_type {
  enum state_type state;
  address PC;
  int reg_file[NUMREGS];
  struct control_block *next_pcb;
  int priority;
  address memory_footprint; ????
  ….
  ….
} control_block;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

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;
```

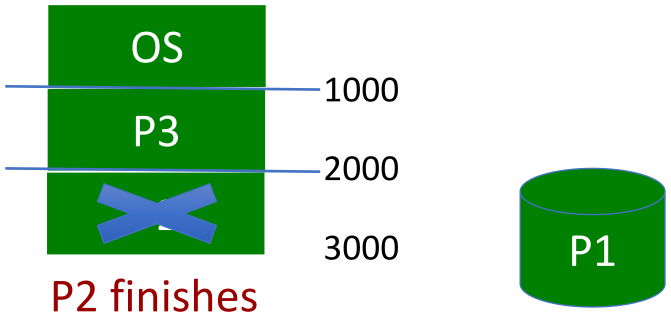
The last example with BASE + LIMIT

- 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!



The last example with BASE + LIMIT

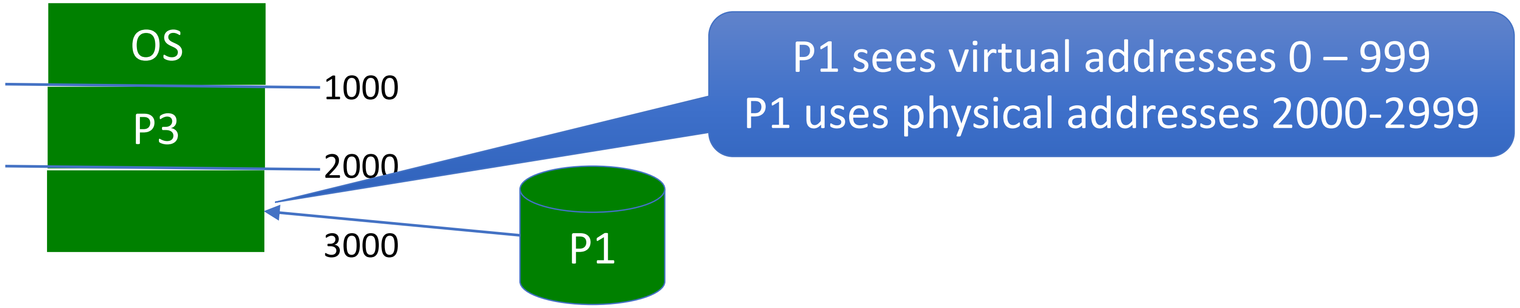
- Now all of P1 and P2’s addresses are relative to zero
- This is our first instance of a **virtual address** where the process sees memory addresses *different* from the physical addresses we’ve been working with so far!



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!



- ✓ 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

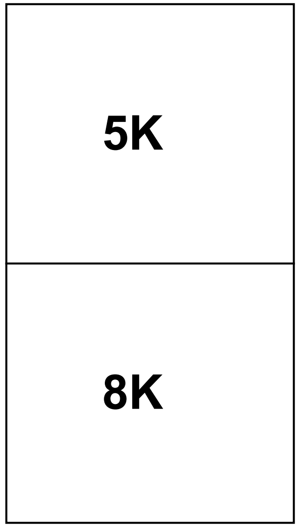
Fixed size partitions

OS memory manager allocation table

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

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

memory

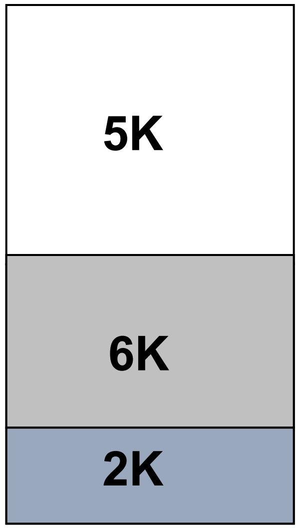


P1 needs 6K of memory

Allocation table

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

Memory



P1 needs 6K of memory

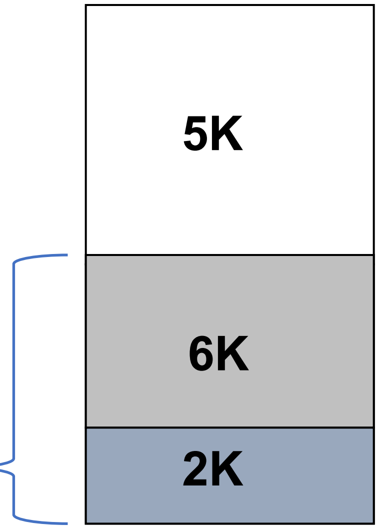
Allocation table

Table size is fixed = number of partitions

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

Needs only 6K

Memory



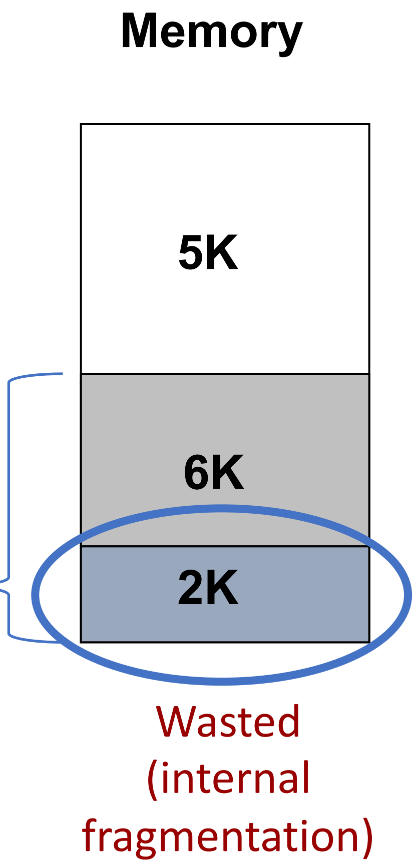
Internal fragmentation

Internal fragmentation = size of partition – actual used

Allocation table

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

Needs only 6K



Another process needs 6K memory

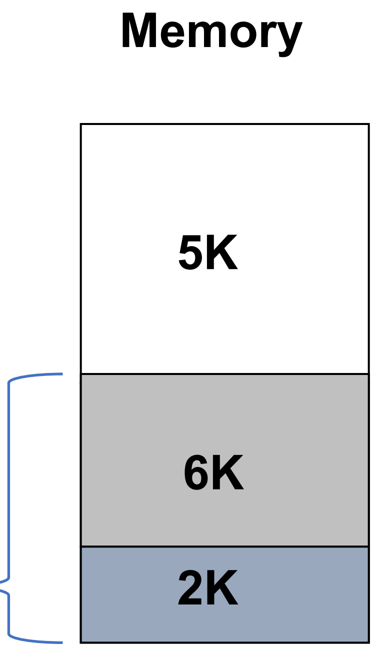
Do we have it?

Memory manager has only a 5K partition…

Not possible

Allocation table

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



External fragmentation

Consider P3 that needs 4K of memory

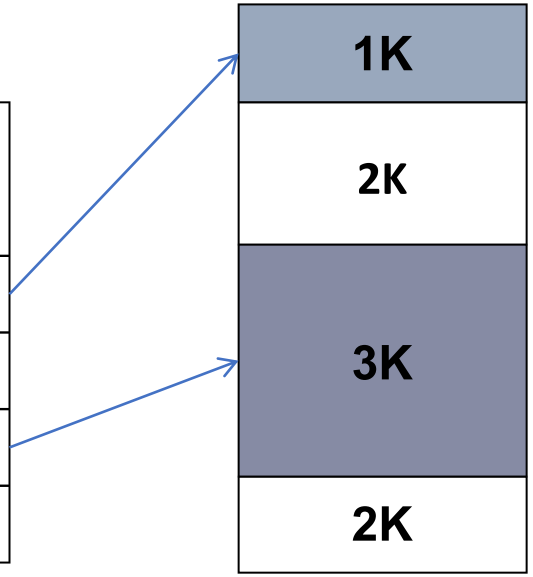
Is it possible to allocate?

Memory is available, but not contiguous

Allocation table

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

Memory



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
  - $\sum$ **All non-contiguous free partitions**

Fixed size partition memory management

Pros

- Simplicity

Cons

- Fragmentation
  - Internal
  - External

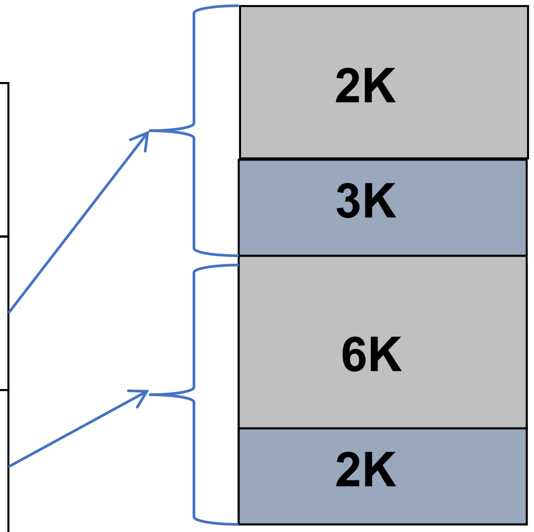
Session ID: dag2200

Total **internal** fragmentation is…

Allocation table

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

Memory

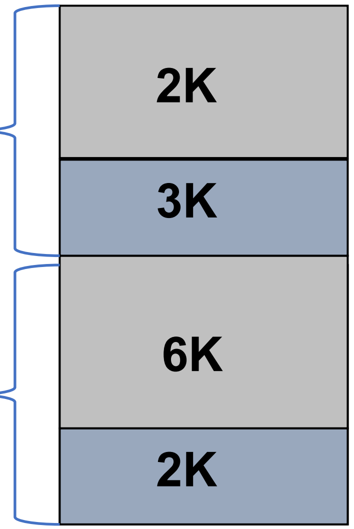


A. 2K
B. 3K
C. 5K
D. 8K

Session ID: dag2200

Total **external** fragmentation is…

Memory



Allocation table

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

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

Overcoming internal fragmentation

- Allocate exactly what is needed
- Variable size partitions

Variable size partitions

Memory manager allocation table

Start address	Size	Process
0	13K	FREE

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

memory

13K

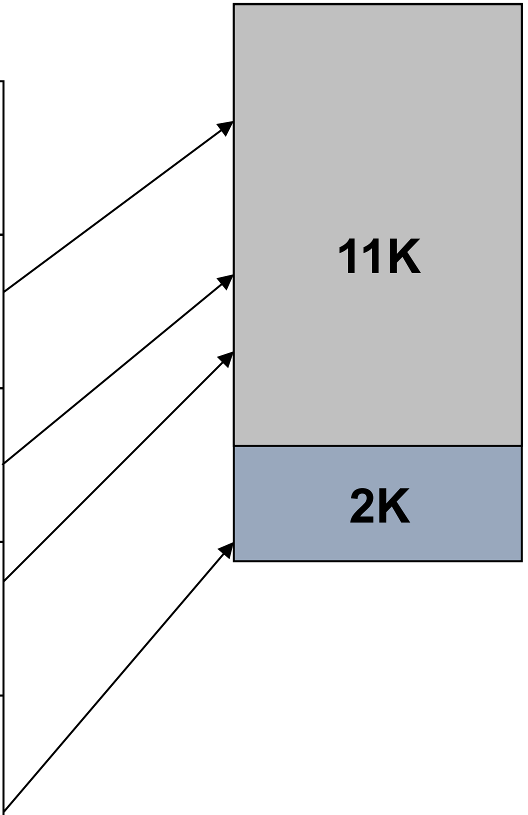
Partition table a little while later

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

Memory

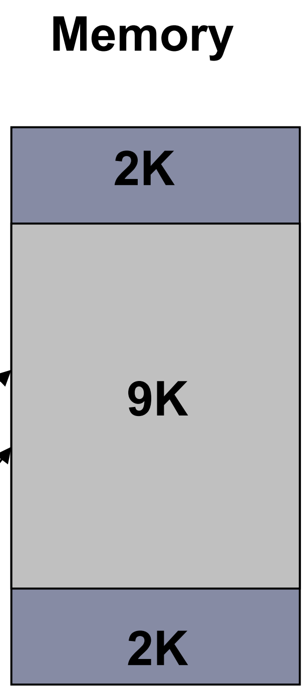


P1 exits

Allocation table

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

External fragmentation



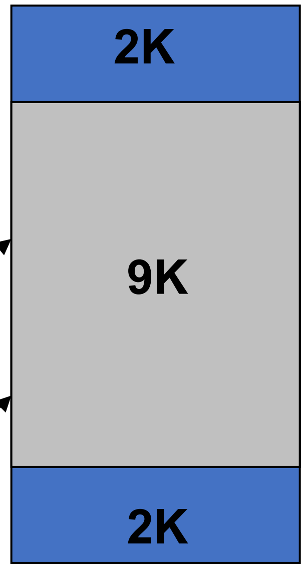
New request:
P4 needs 4K
Possible?

P2 exits

Allocation table

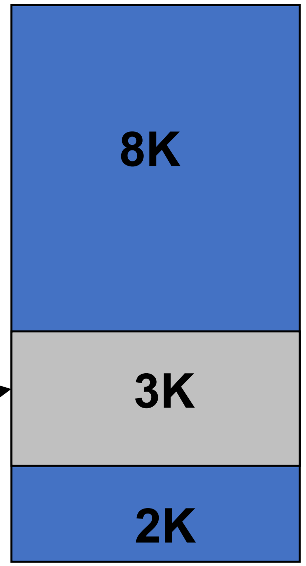
Start address	Size	Process
0	2K	FREE
2K	6K	P2 → FREE
8K	3K	P3
11K	2K	FREE

Memory



Coalescing two free areas

Memory



Allocation table

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

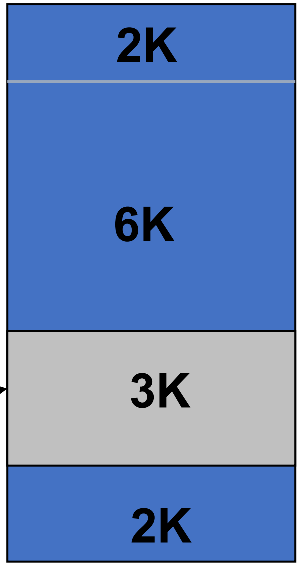
Reducing external fragmentation

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

Allocation table

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

Memory



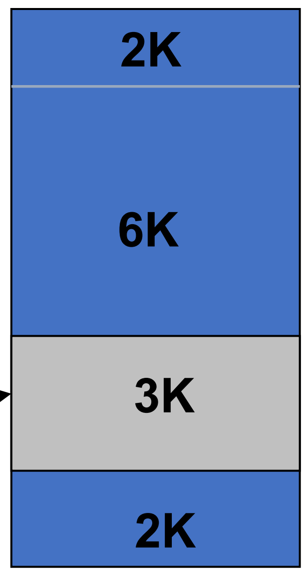
Compaction

- Request for 9K

Allocation table

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

Memory

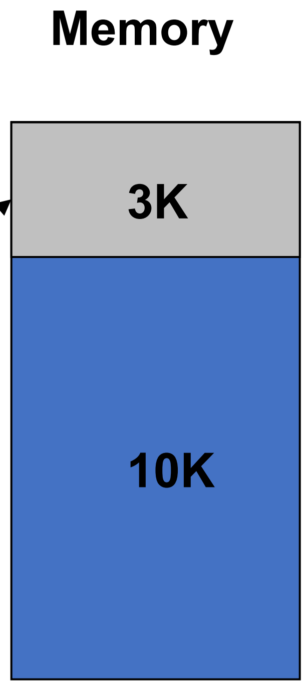


Compaction

- 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 …

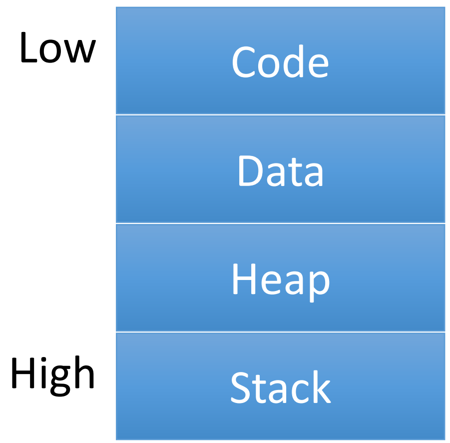
A. No external fragmentation
B. No internal fragmentation
C. No fragmentation
D. Both internal and external fragmentation

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?

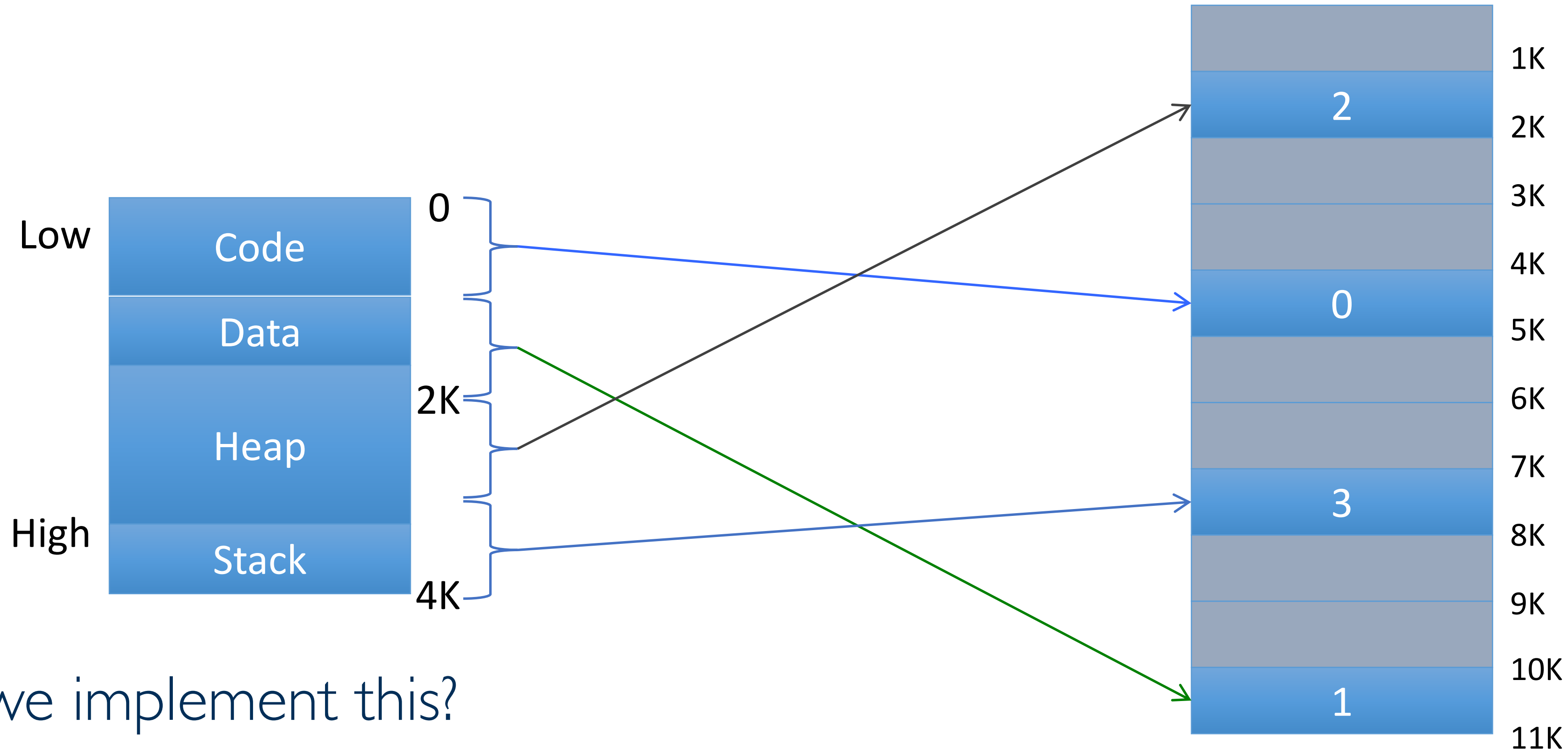
- Our memory footprint looks like this



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

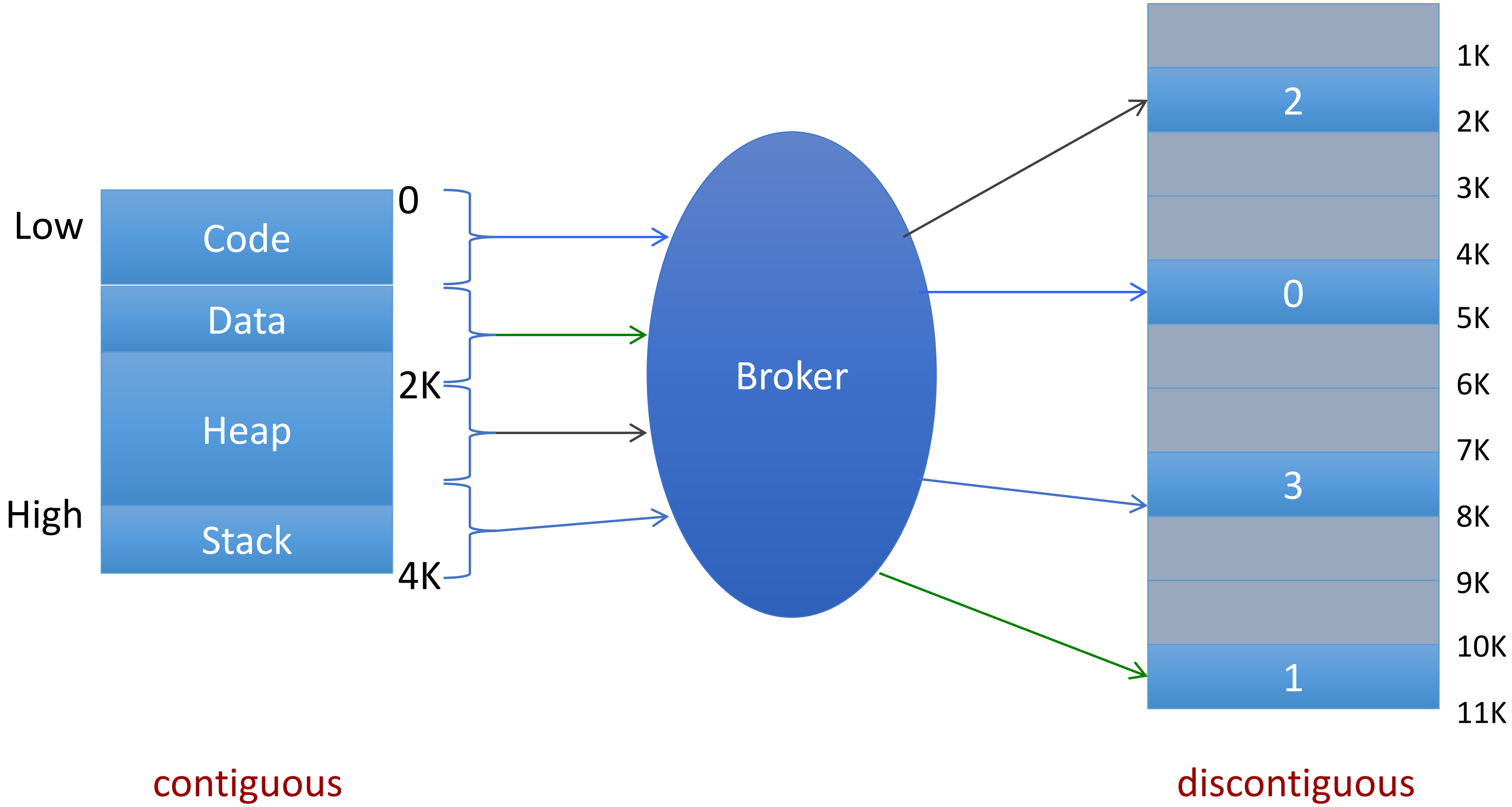
How might we solve the external fragmentation problem?

- 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



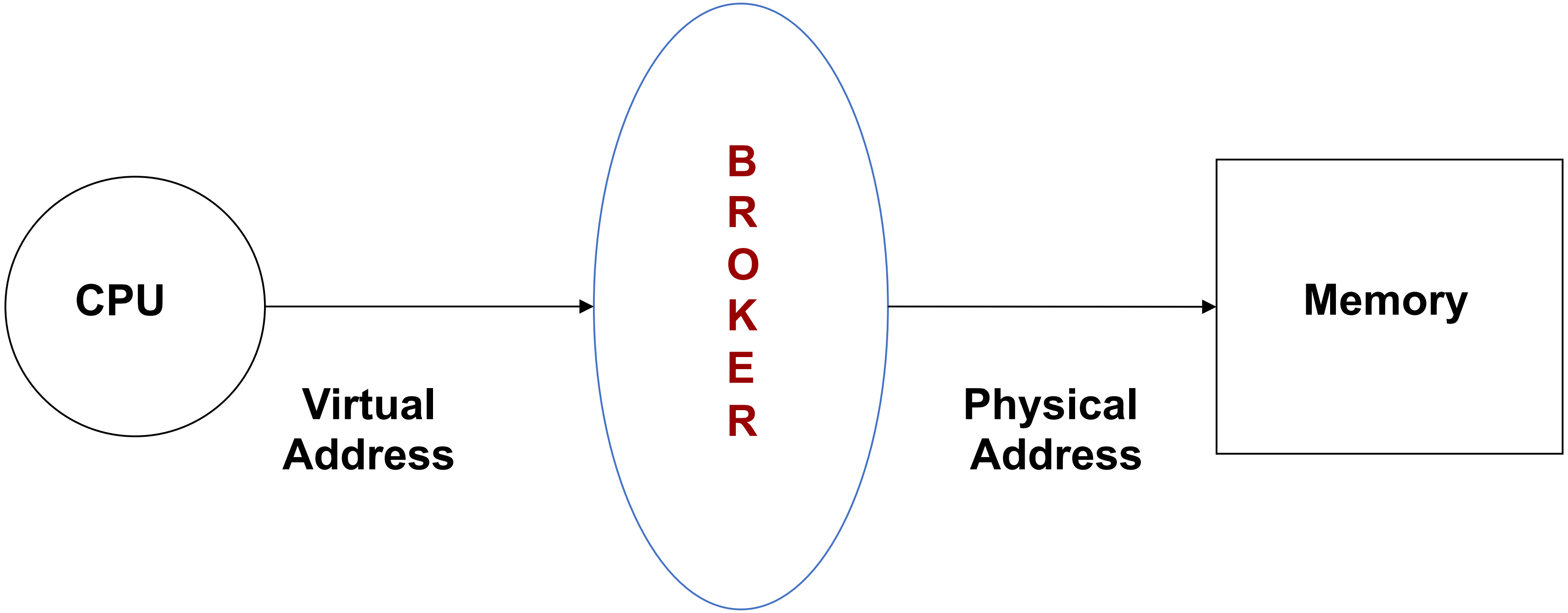
Broker

This broker maps

- Virtual address (VA) from the CPU

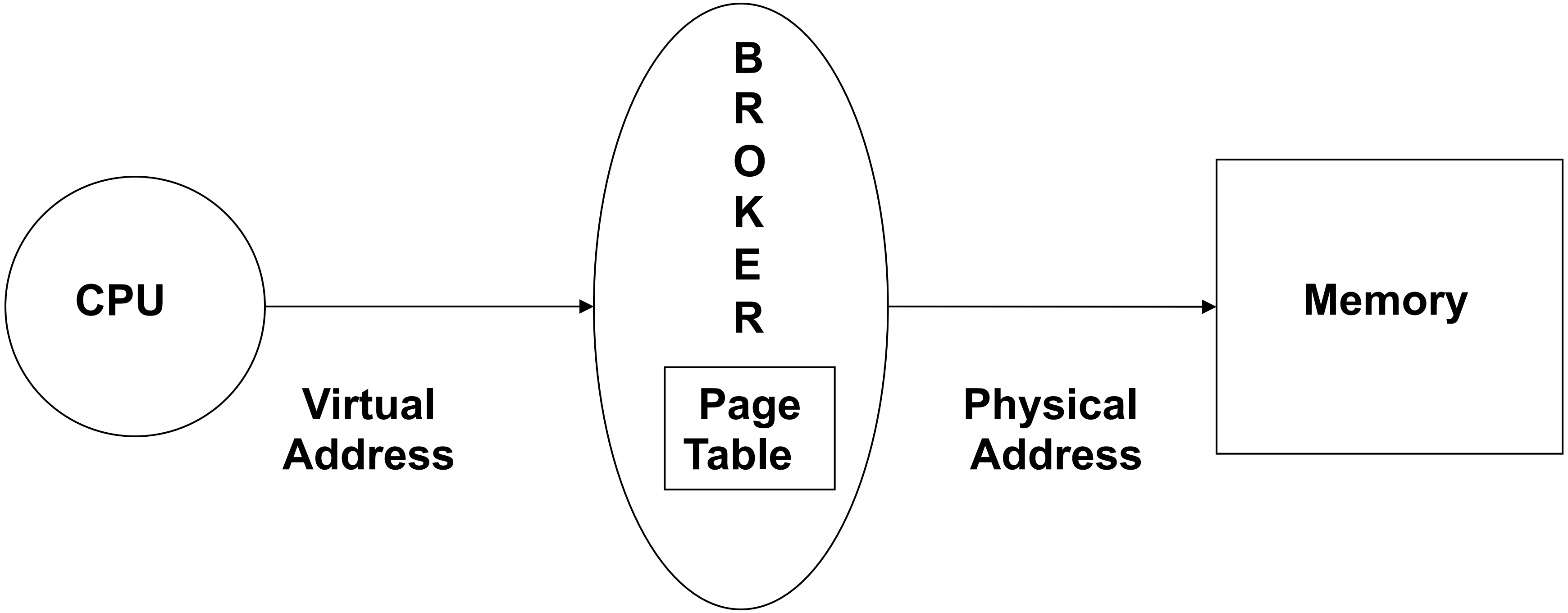
  to

- 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

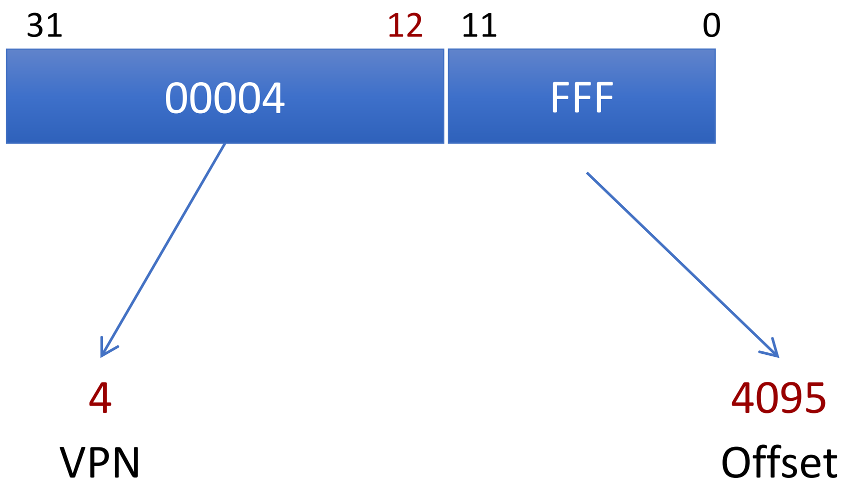
- These days
  - 4 KB up to 1 GB
  - Often it's configurable per process

- 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**

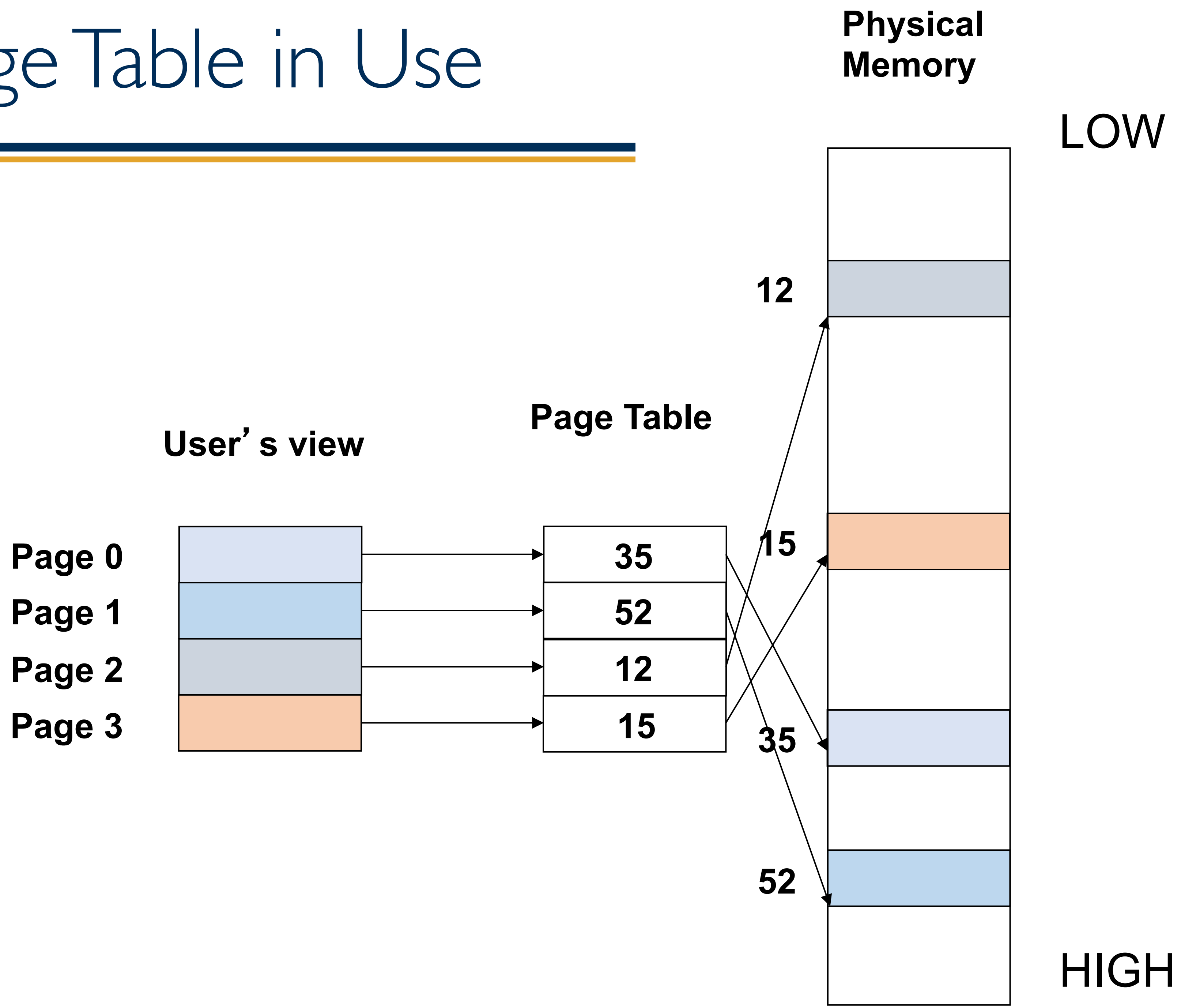
Splitting up a virtual address

- 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

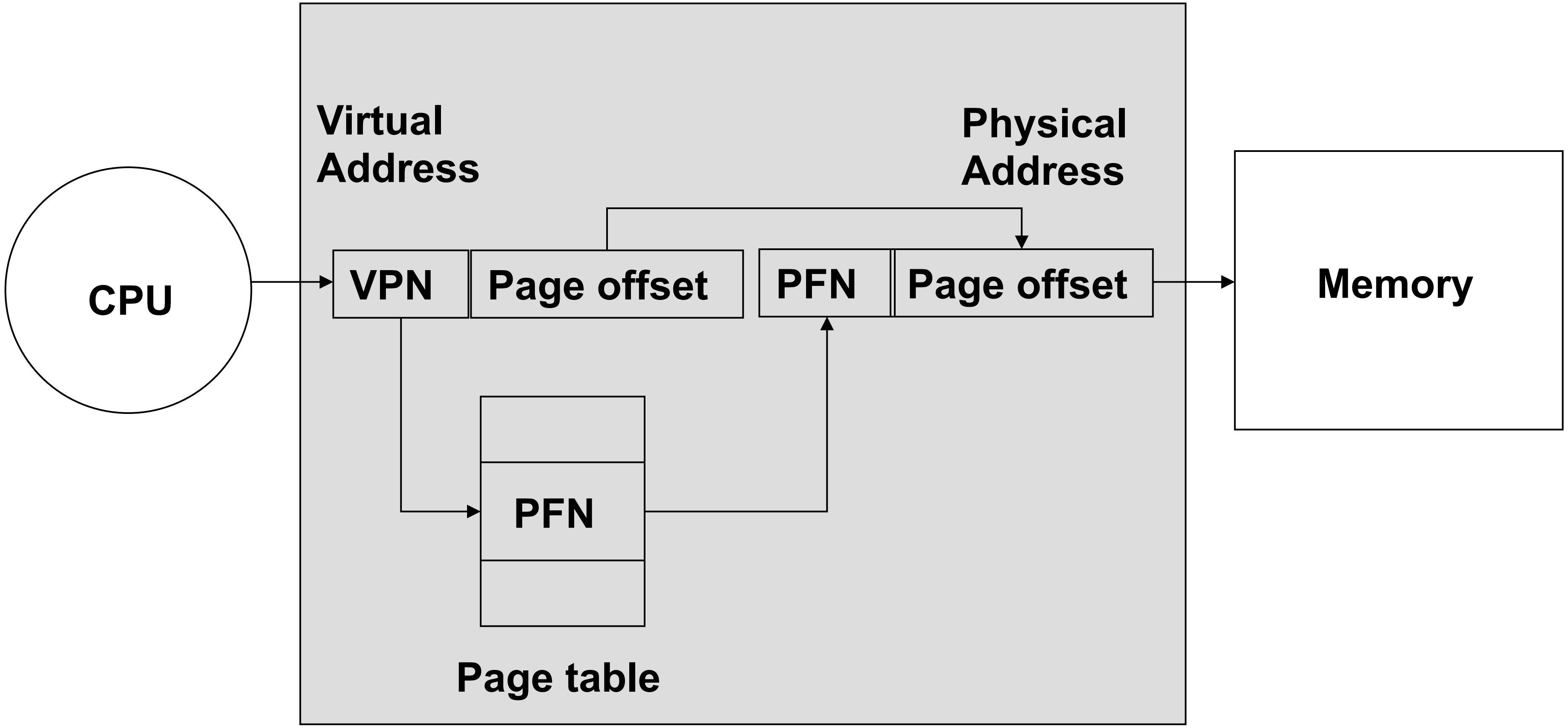
- For example, for virtual address 0x00004FFF:



Page Table in Use

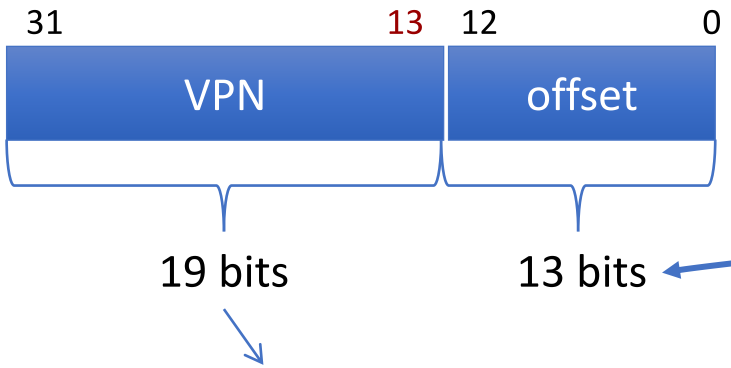


Address translation



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?
- $8k = 2^{13}$



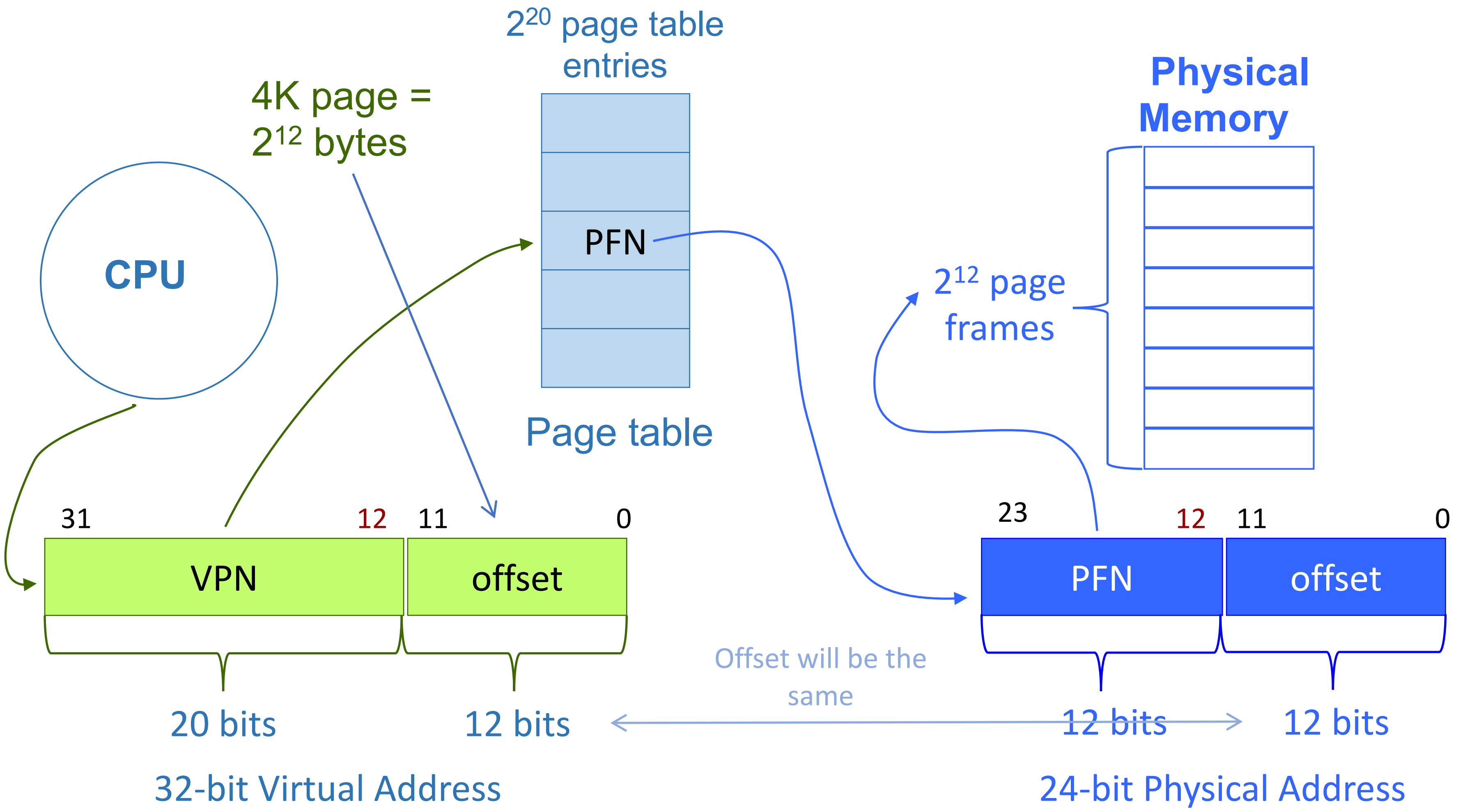
- VPN is 19 bits, so page table is 2^{19} or 524,288 entries

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?

Example



2^{20} page table entries
4K page = 2^{12} bytes
Physical Memory
CPU
PFN
2^{12} page frames
Page table
31
12
11
0
VPN
offset
23
12
11
0
PFN
offset
Offset will be the same
20 bits
12 bits
12 bits
12 bits
32-bit Virtual Address
24-bit Physical Address

Important facts about paging