

CS2200

Systems and Networks

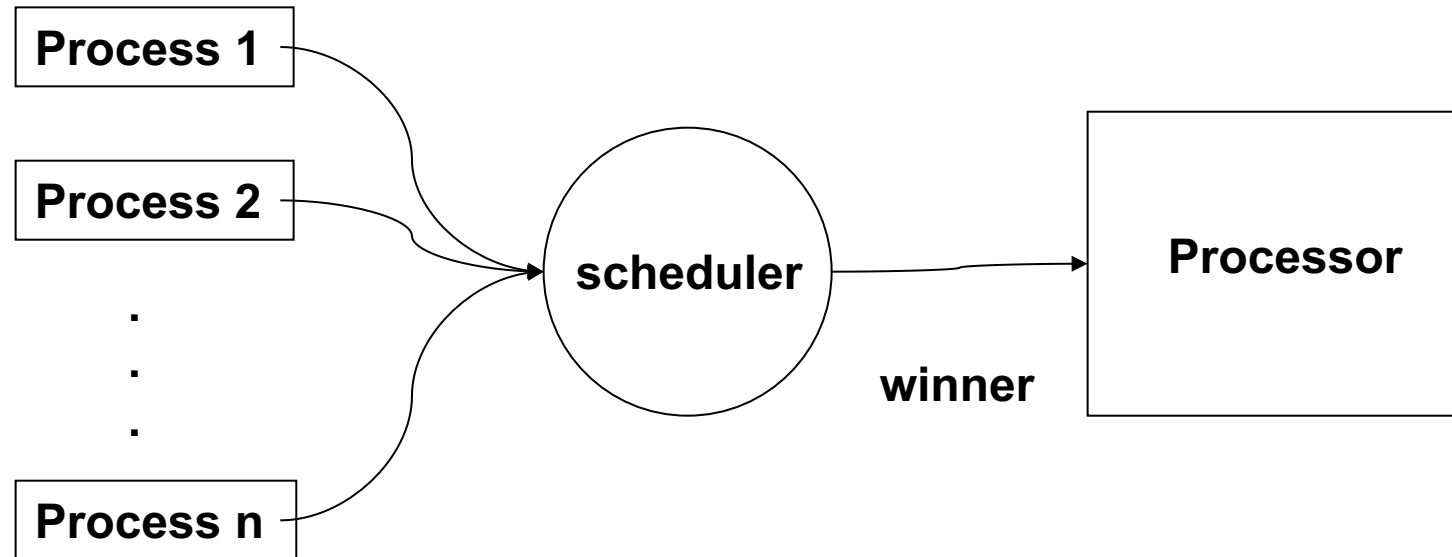
Spring 2024

Lecture 14:

Process Scheduling Algorithms

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

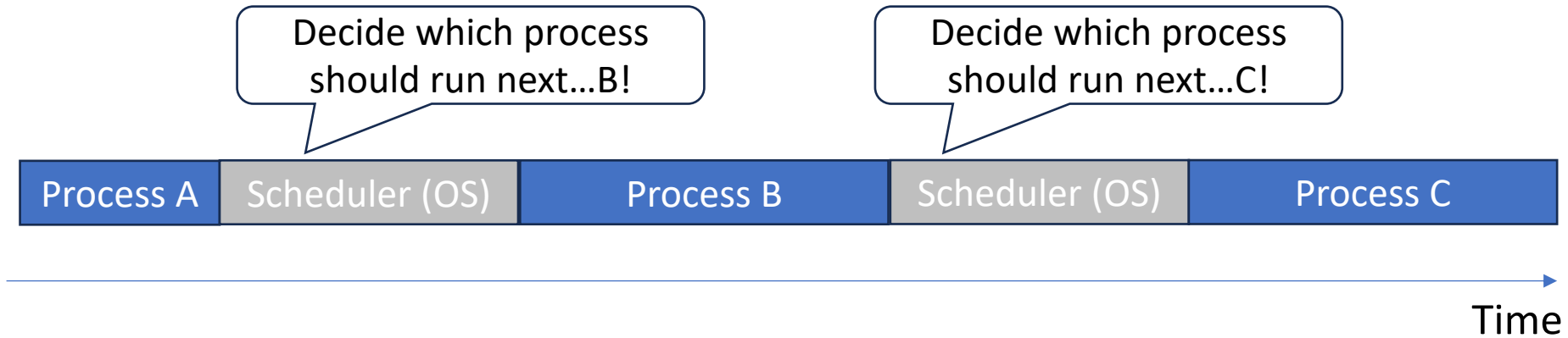
Recap



- Scheduler runs in *software*; part of the OS
- If we have one processor, and a process already runs on it...
how can the scheduler itself run on the processor and decide next process to run?

What needs to happen...

Who runs on processor:




...but how will the running process stop to let the scheduler (OS) run? Two options:

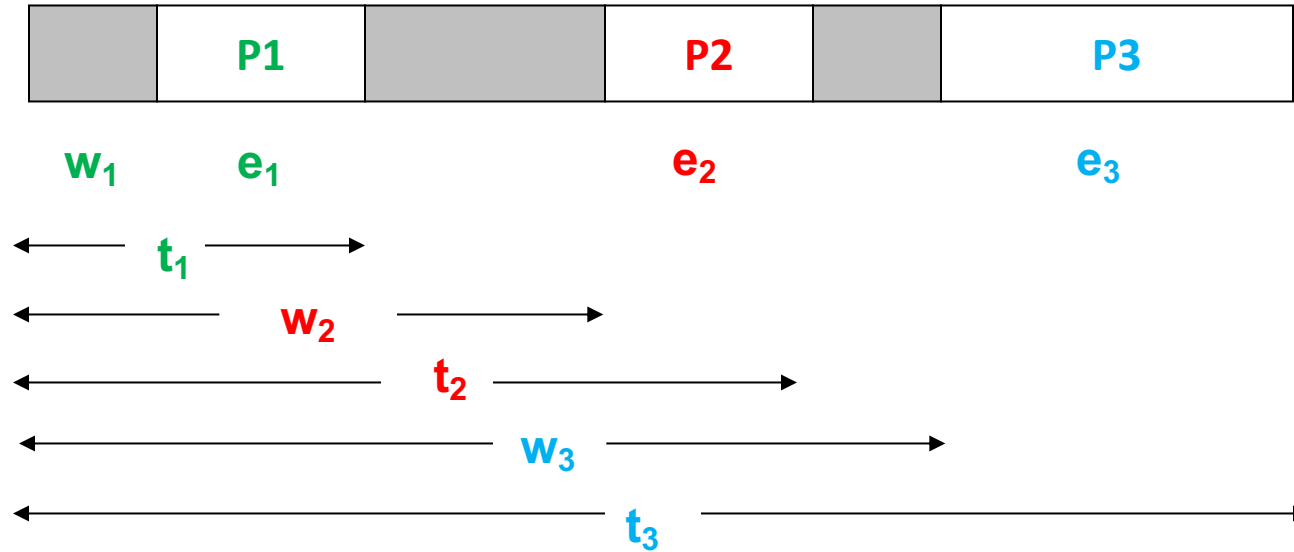
- Non-preemptive scheduling: running process willfully yields control to OS (e.g., trap, exit, IO burst)
- Preemptive scheduling: OS forces periodic yield (e.g., by setting a timer interrupt)

Steps in scheduling

This whole
process is
called a
“context
switch”

- 
- Grab the attention of the processor
 - Save the state of the current process
 - Select a new process to run
 - Dispatch the selected process

Metrics



For process P_i

- w_i = wait time
- e_i = execution time
- t_i = elapsed time
(turnaround time)

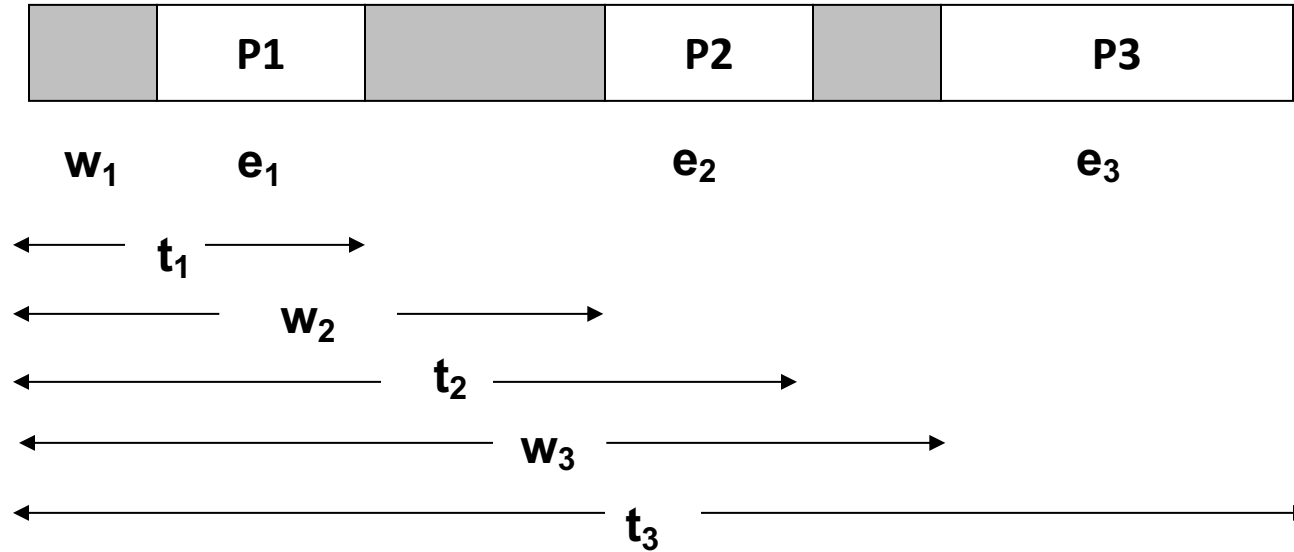
Throughput?

Avg. Turnaround Time?

Avg. Wait Time?

Response time?

Metrics



For process P_i

- w_i = wait time
- e_i = execution time
- t_i = elapsed time
(turnaround time)

System Centric

Throughput?

Avg. Turnaround Time?

Avg. Wait Time?

User Centric


Response time?

$3 / t_3$ jobs/sec

$(t_1 + t_2 + t_3) / 3$ sec

$(w_1 + w_2 + w_3) / 3$ sec

$R_{P1} = t_1$, $R_{P2} = t_2$, $R_{P3} = t_3$

Name	Notation	Units	Description
CPU Utilization	-	%	Percentage of time the CPU is busy
Throughput	n/T	Jobs/s	System-centric metric quantifying the number of jobs n executed in time interval T
Avg. Turnaround time (t_{avg})	$(t_1 + t_2 + \dots + t_n)/n$	Secs	System-centric metric quantifying the average time it takes for a job to complete
Avg. Waiting time (w_{avg})	$(w_1 + w_2 + \dots + w_n)/n$	Secs	System-centric metric quantifying the average waiting time that a job experiences
Response time	t_i	Secs	User-centric metric quantifying the turnaround time for a specific job I
Variance in Response time	$E[(t_i - t_{avg})^2]$	Secs ²	User-centric metric that quantifies the statistical variance of the actual response time (t_i) experienced by a process (P_i) from the expected value (t_{avg})
Starvation	-	-	User-centric qualitative metric that signifies denial of service to a particular process or a set of processes due to some intrinsic property of the scheduler
Convoy effect 	-	-	User-centric qualitative metric that results in a detrimental effect to some set of processes due to some intrinsic property of the scheduler [This often appears as a “convoy” of short jobs waiting for the completion of a long job; non-preemptive FCFS is the convoy effect’s native habitat.]

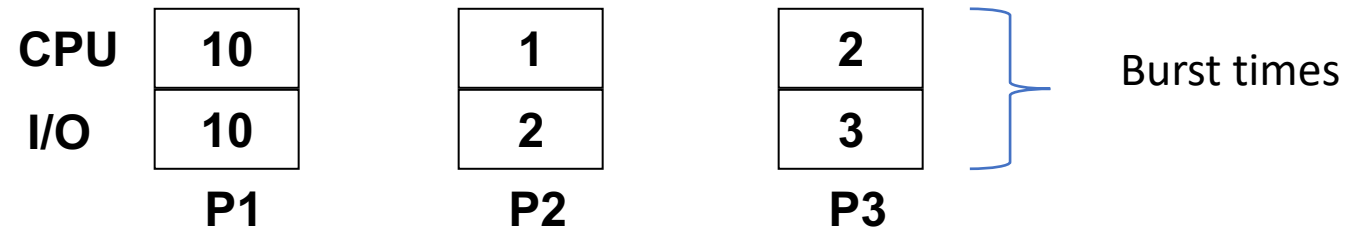


The most user-centric metric of a scheduler is...

- 0% A. Throughput
- 0% B. Average waiting time
- 0% C. Average turnaround time
- 0% D. CPU utilization
- 0% E. Response time
- 0% F. None of the above

Non-preemptive scheduling algorithms

- FCFS
 - SJF
 - Priority
 - Resource requirements:
- } Intrinsic property
 → Extrinsic property



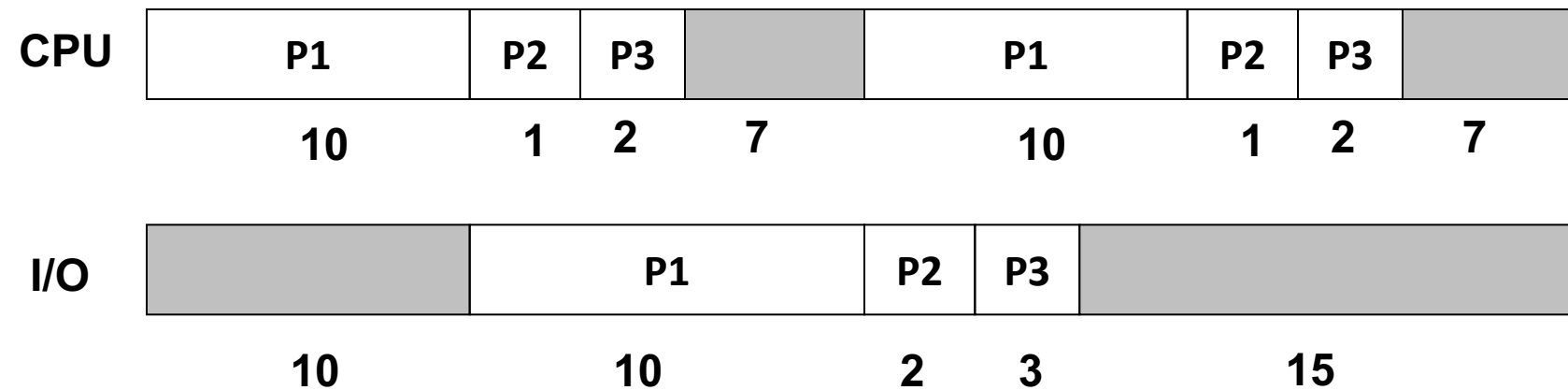
- Arrival order
 - P1, P2, P3 in order at nearly the same time

Assume each process goes through following steps

1. CPU burst
2. I/O Burst
3. CPU Burst
4. Done

FCFS

CPU	10	1	2
I/O	10	2	3
	P1	P2	P3

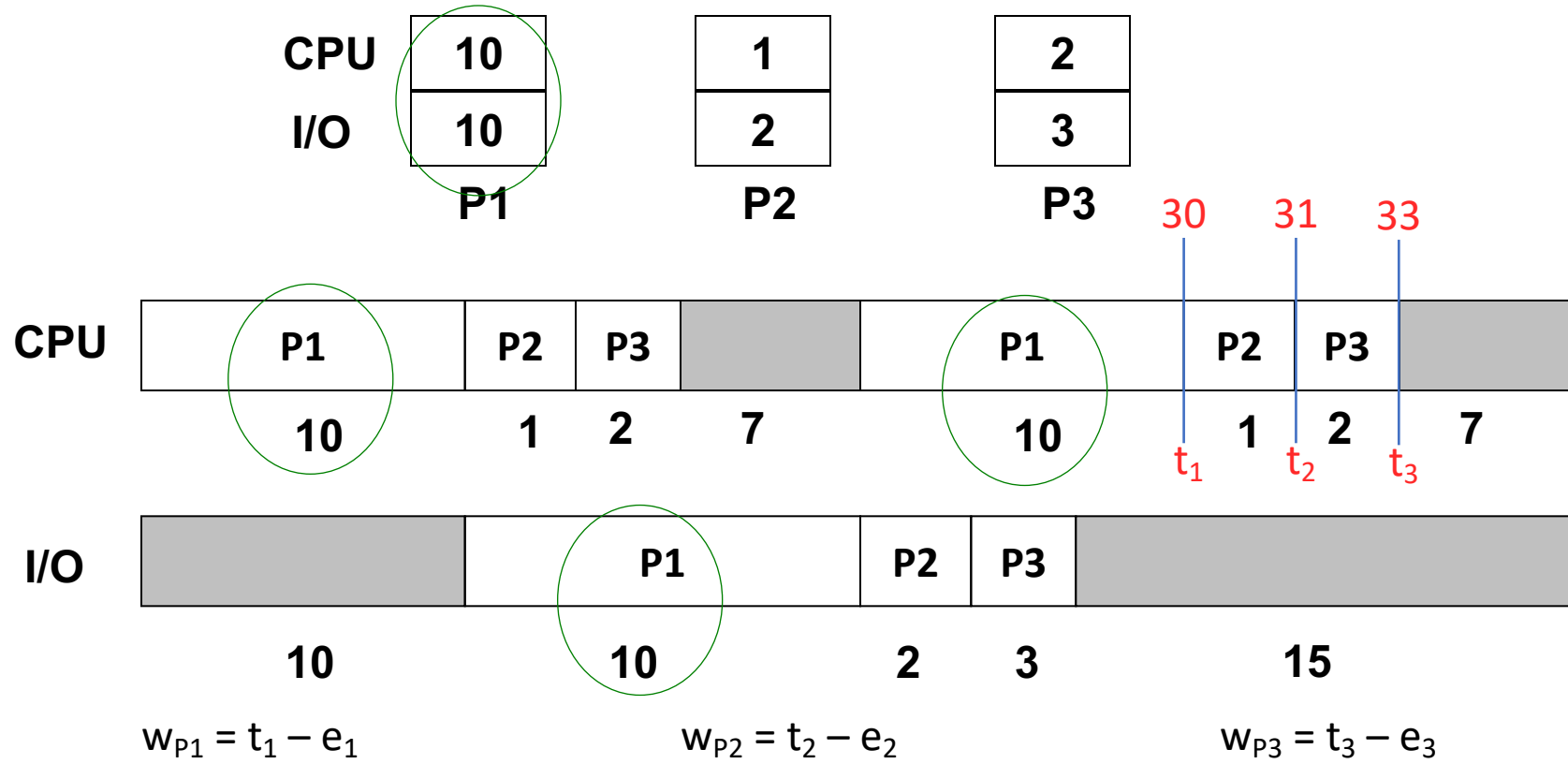


$$t_i = w_i + e_i$$

What are the waiting times?

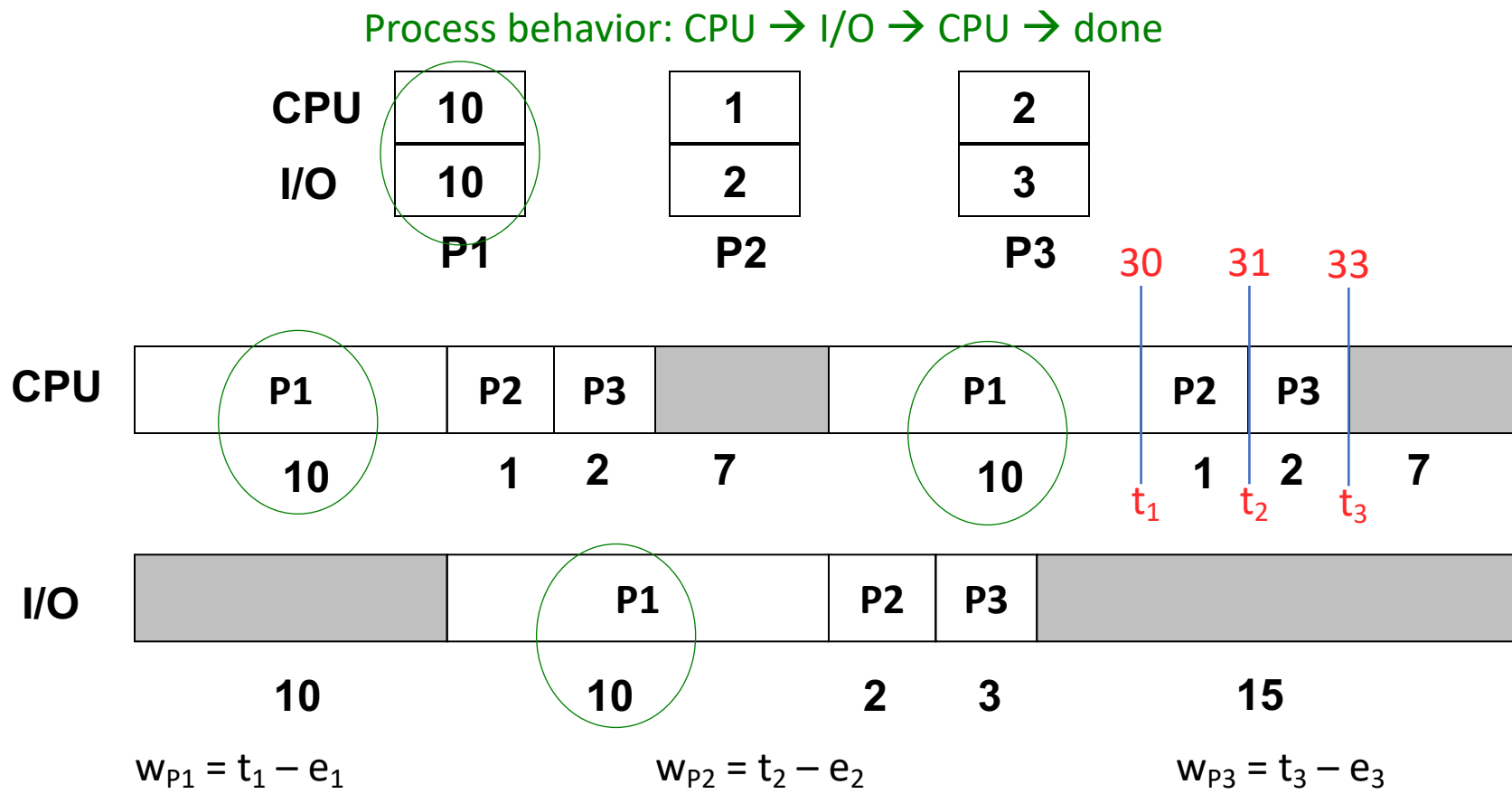
FCFS

Process behavior: CPU → I/O → CPU → done



$e_1 = ?$

FCFS

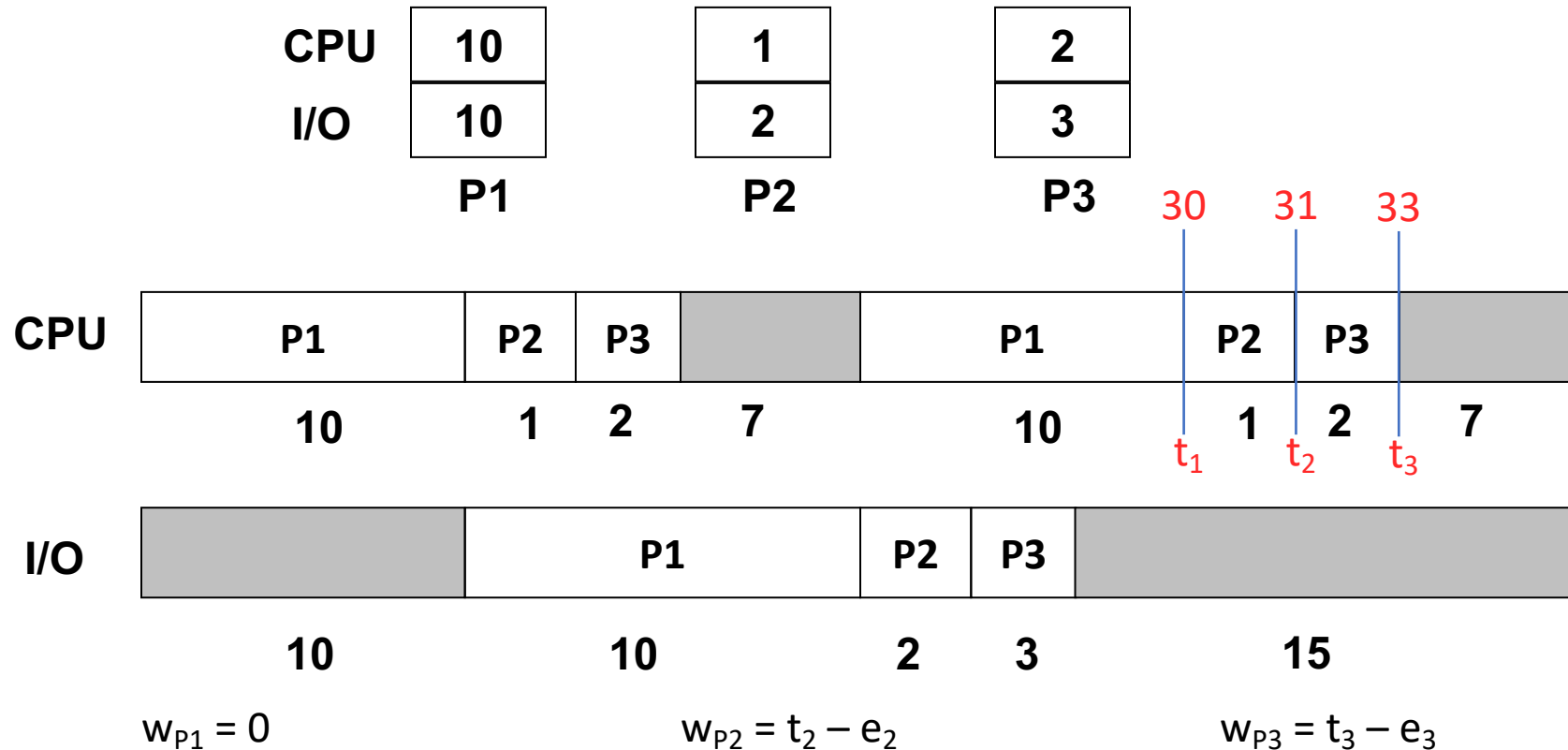


$$e_1 = 10 + 10 + 10, t_1 = 30$$

$$w_{P1} = 30 - 30 = 0$$

Individual Activity!

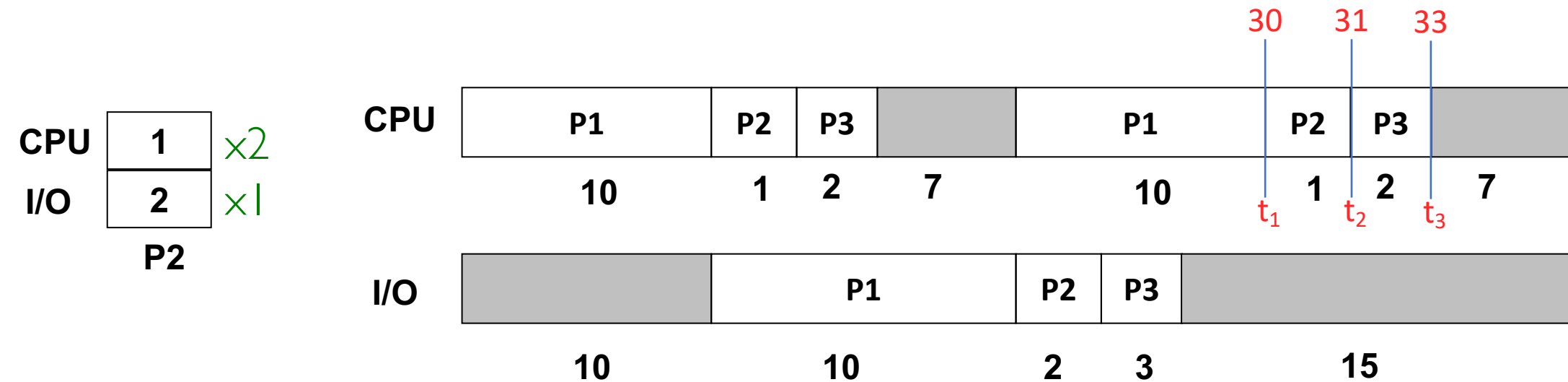
You do the same thing for P2 and P3 (compute w_{P2} and w_{P3})



$$W_{P1} = 30 - 30 = 0$$

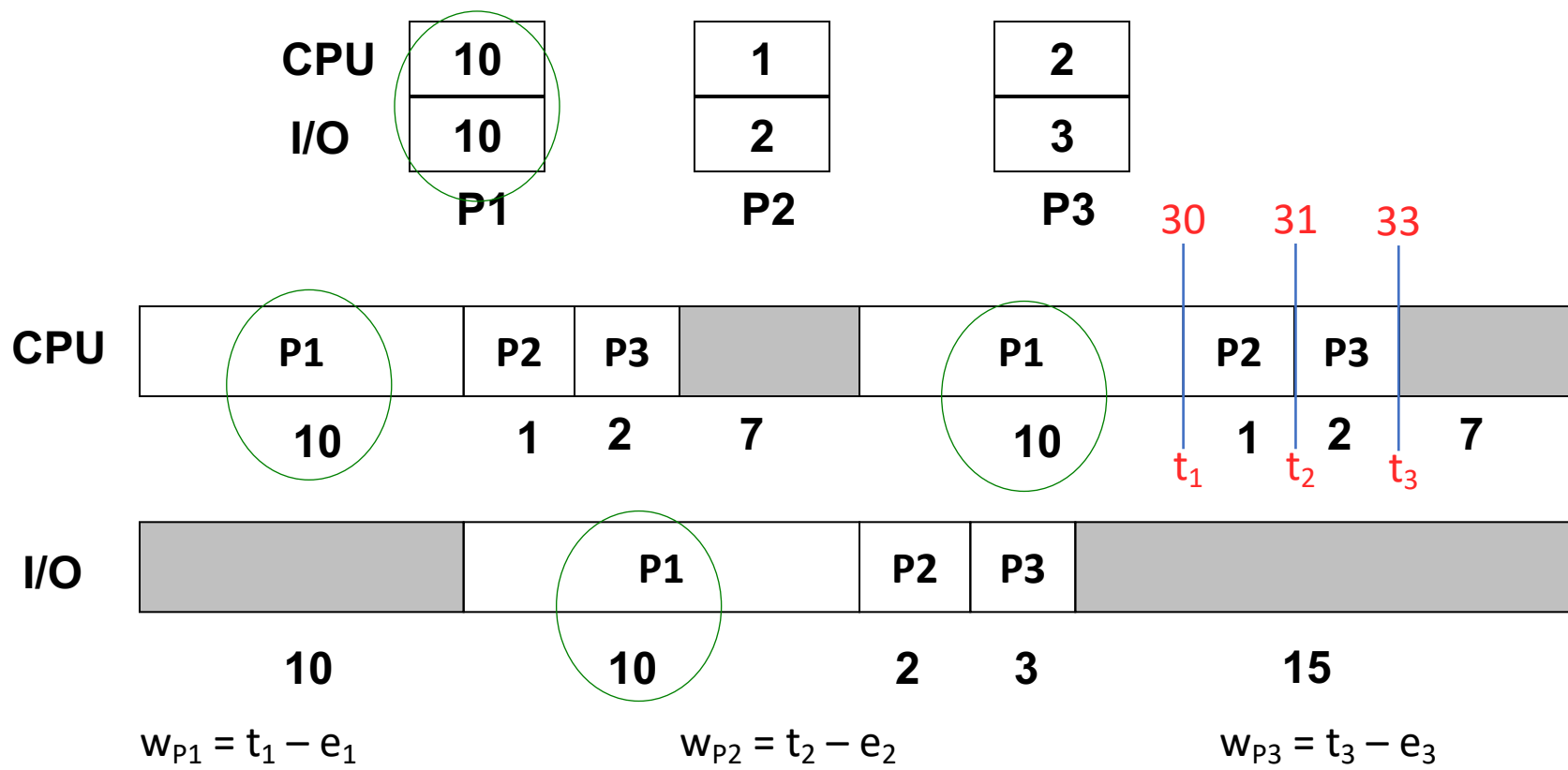
What is the wait time for P2?

- 0% A. Didn't understand how to work it out
- 0% B. 0
- 0% C. 26
- 0% D. 27
- 0% E. Forgot how to subtract



FCFS

Process behavior: CPU → I/O → CPU → done



$$w_{P1} = t_1 - e_1$$

$$w_{P2} = t_2 - e_2$$

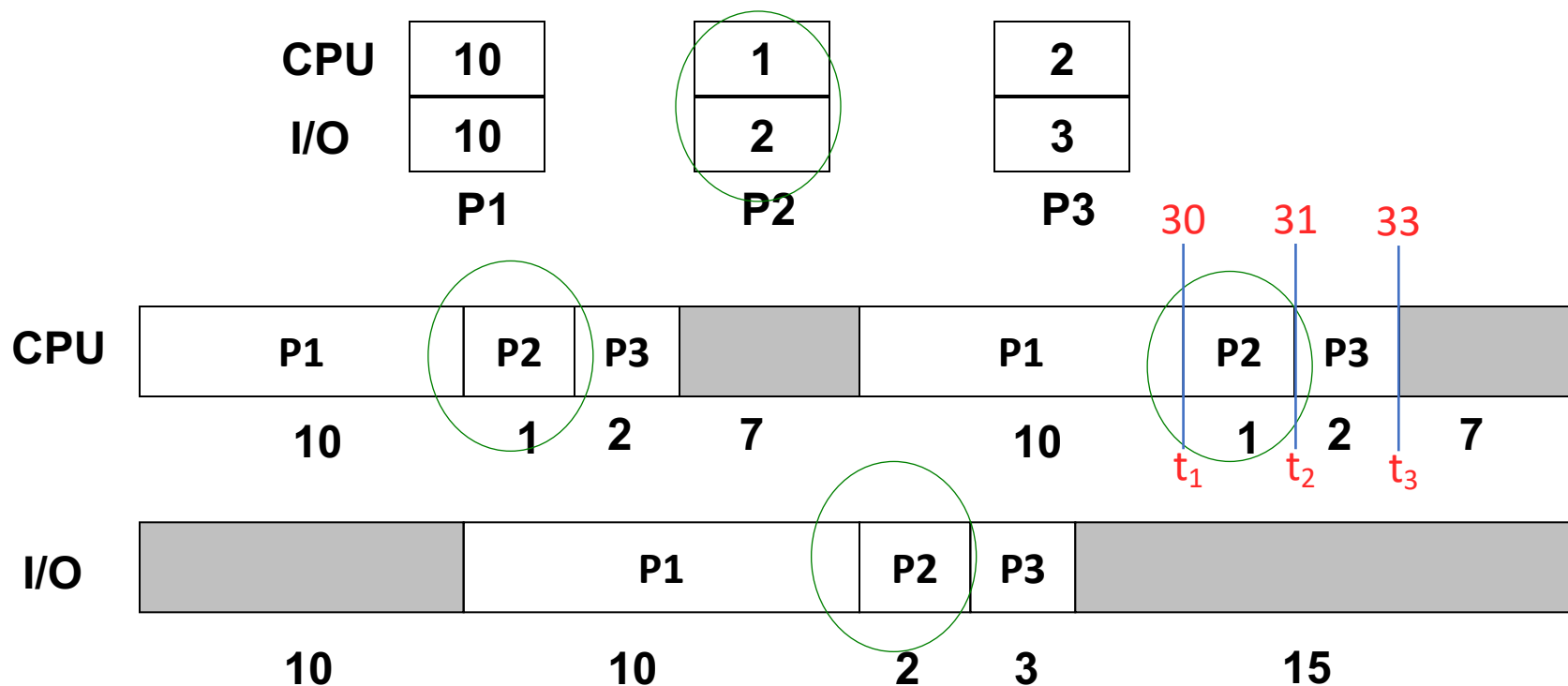
$$w_{P3} = t_3 - e_3$$

$$e_1 = 10 + 10 + 10, t_1 = 30$$

$$w_{P1} = 30 - 30 = 0$$

FCFS

Process behavior: CPU → I/O → CPU → done



$$w_{P1} = t_1 - e_1$$

$$w_{P2} = t_2 - e_2$$

$$w_{P3} = t_3 - e_3$$

$$e_1 = 10 + 10 + 10, t_1 = 30$$

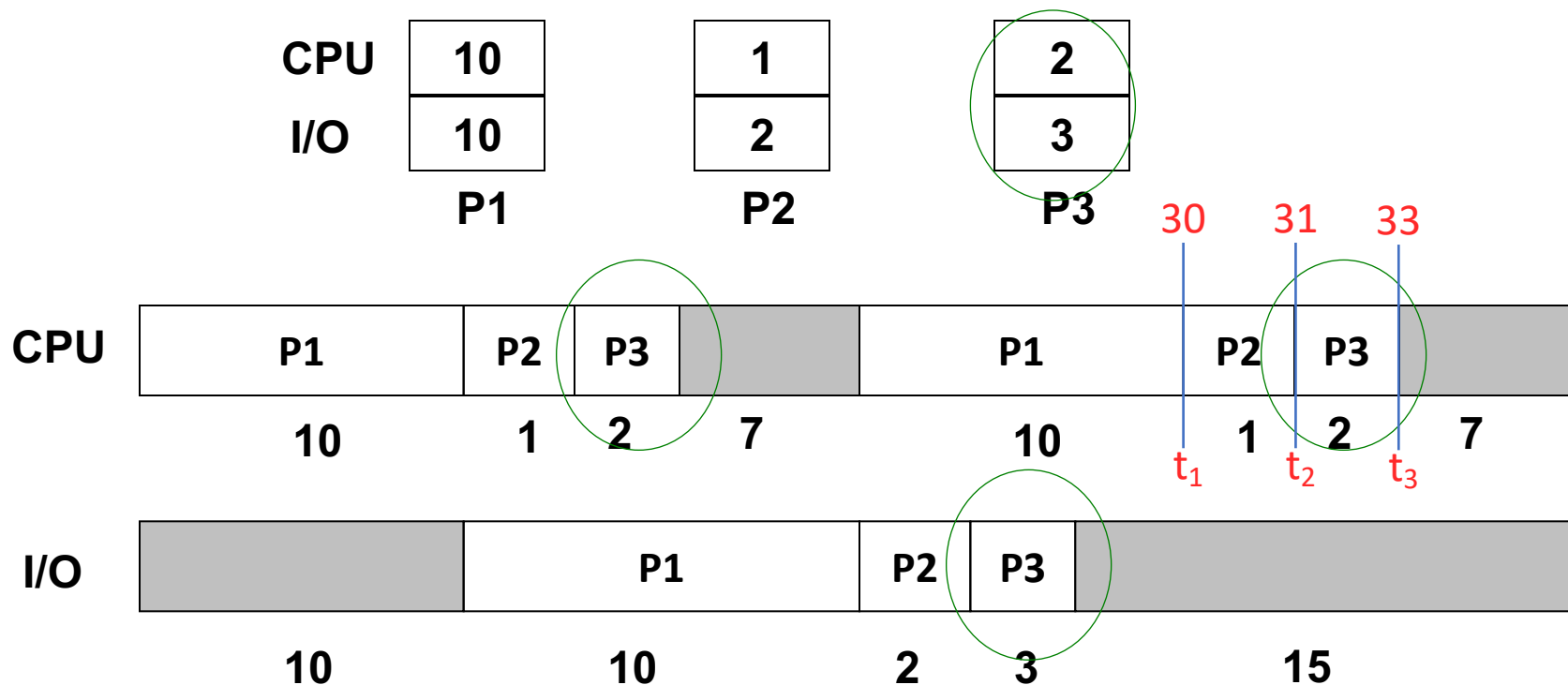
$$e_2 = 1 + 2 + 1, t_2 = 31$$

$$w_{P1} = 30 - 30 = 0$$

$$w_{P2} = 31 - 4 = 27$$

FCFS

Process behavior: CPU → I/O → CPU → done



$$w_{P1} = t_1 - e_1$$

$$e_1 = 10 + 10 + 10, t_1 = 30$$

$$w_{P1} = 30 - 30 = 0$$

$$w_{P2} = t_2 - e_2$$

$$e_2 = 1 + 2 + 1, t_2 = 31$$

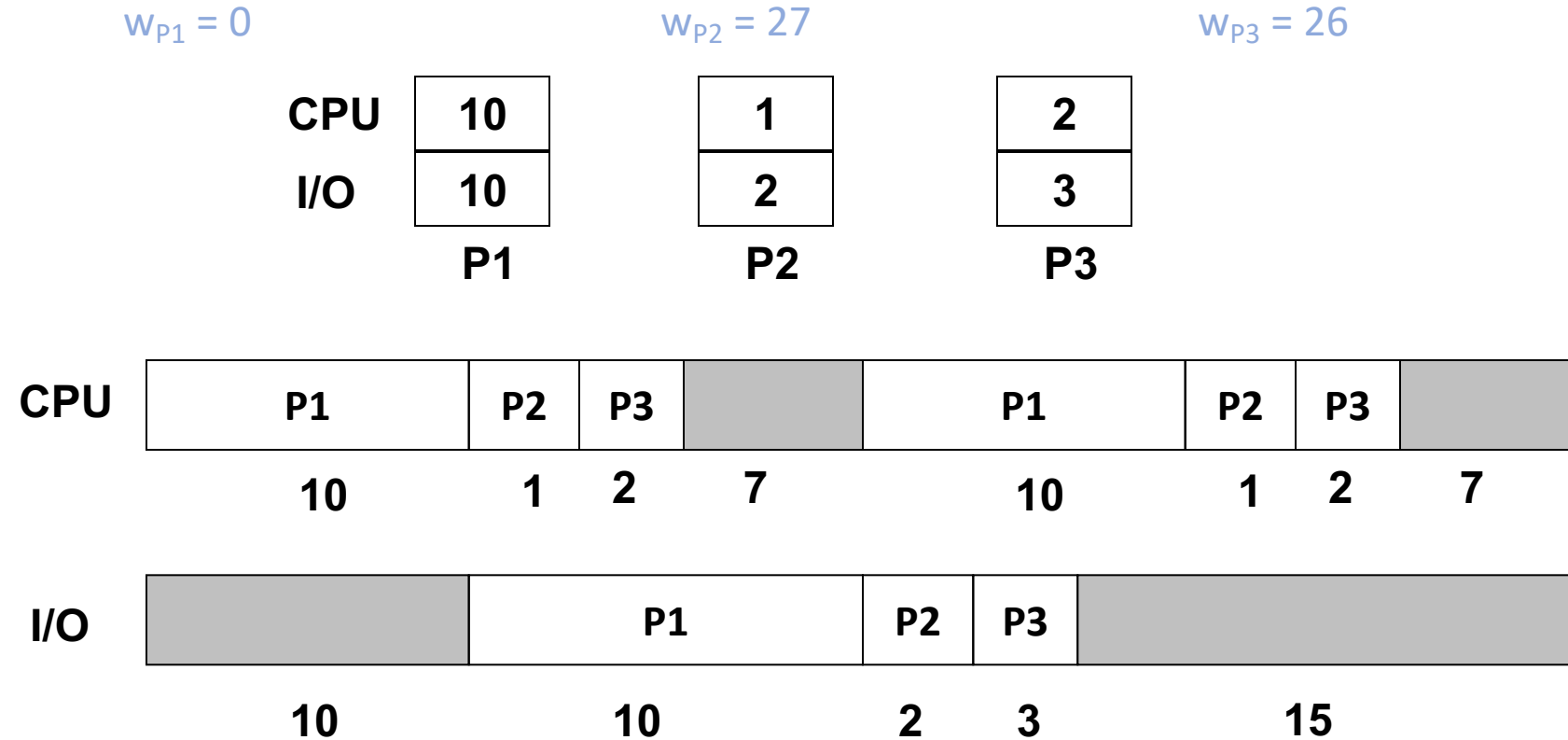
$$w_{P2} = 31 - 4 = 27$$

$$w_{P3} = t_3 - e_3$$

$$e_3 = 2 + 3 + 2, t_3 = 33$$

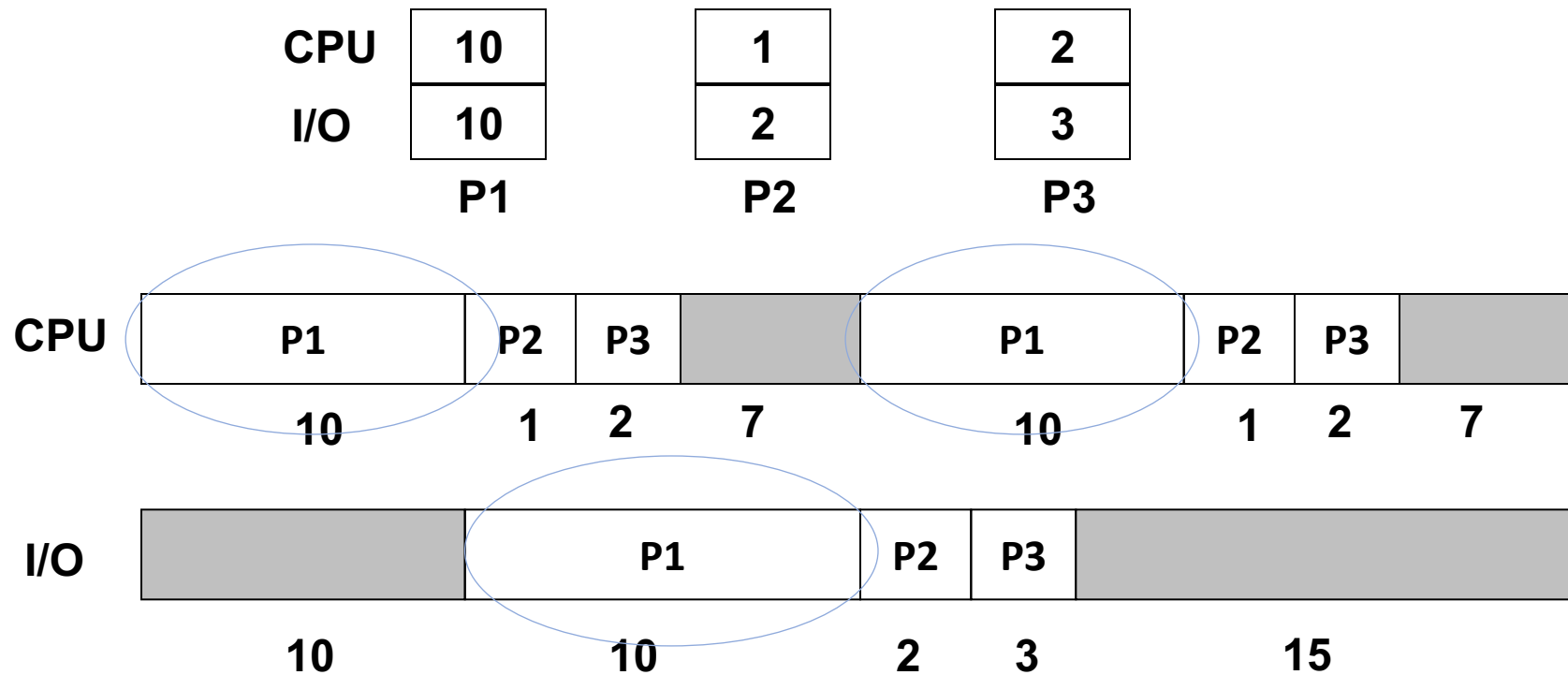
$$w_{P3} = 33 - 7 = 26$$

FCFS



- High average waiting times

FCFS



- High average waiting times – in this case $(0+26+27=)53 / 3$
- High average turnaround times – in this case $(30+31+33=)94 / 3$
- Convoy effect

Non-preemptive scheduling algorithms

~~FCFS~~

- SJF
- Priority
- Resource requirements:

Of all ready processes, the one with the shortest CPU burst goes first

CPU	10
I/O	10
P1	

1
2
P2

2
3
P3

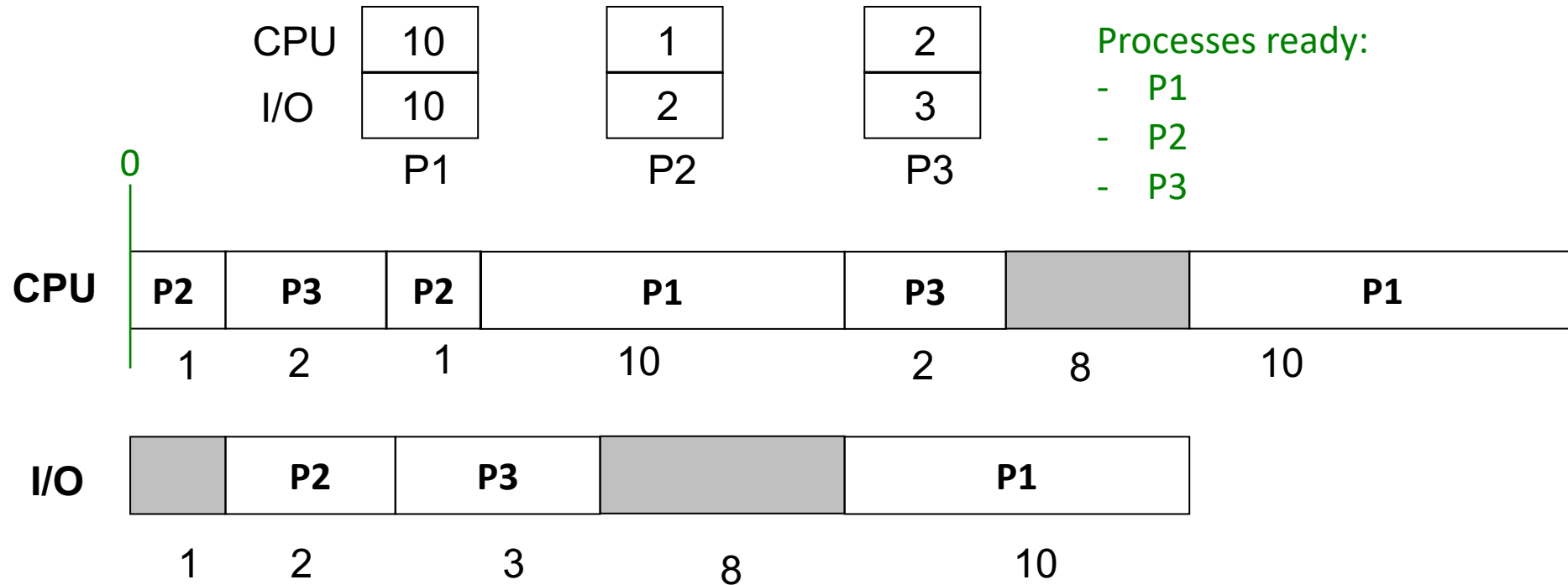
Burst times

- Arrival order
 - P1, P2, P3 in order at nearly the same time

Assume each process goes through following steps

1. CPU burst
2. I/O Burst
3. CPU Burst
4. Done

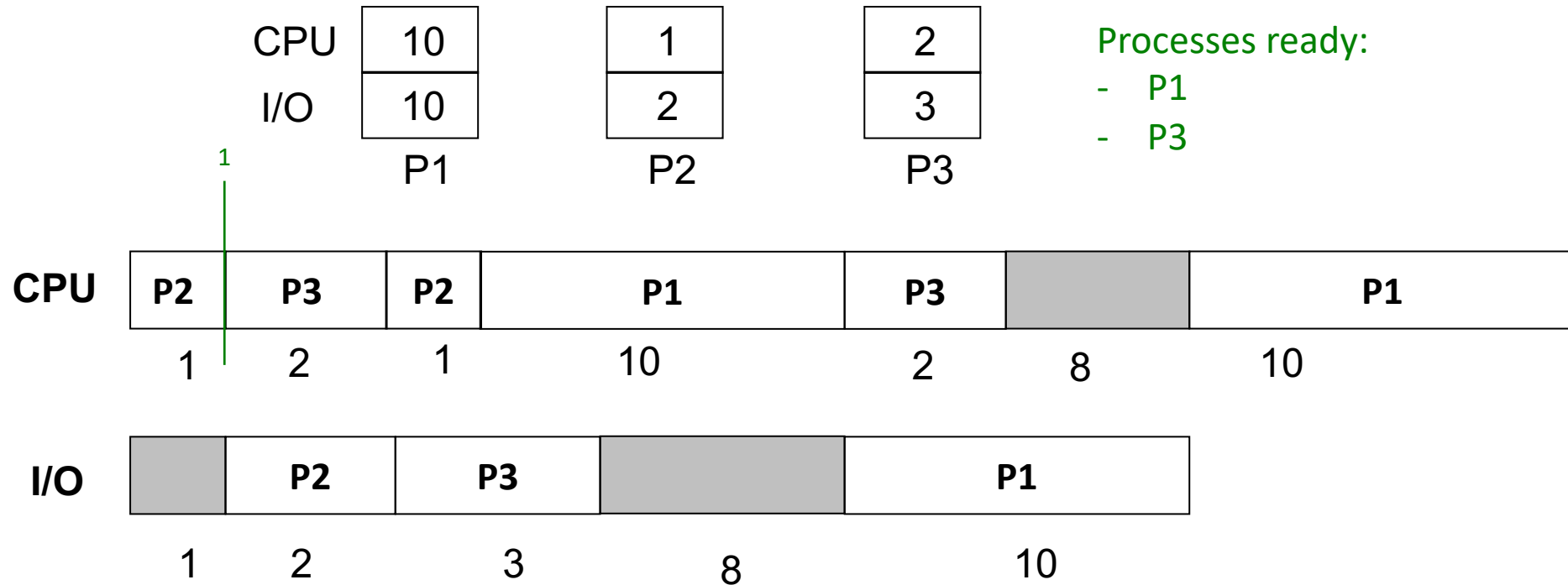
SJF



Straightforward:

P2 is ready with shortest CPU burst, so it gets run first

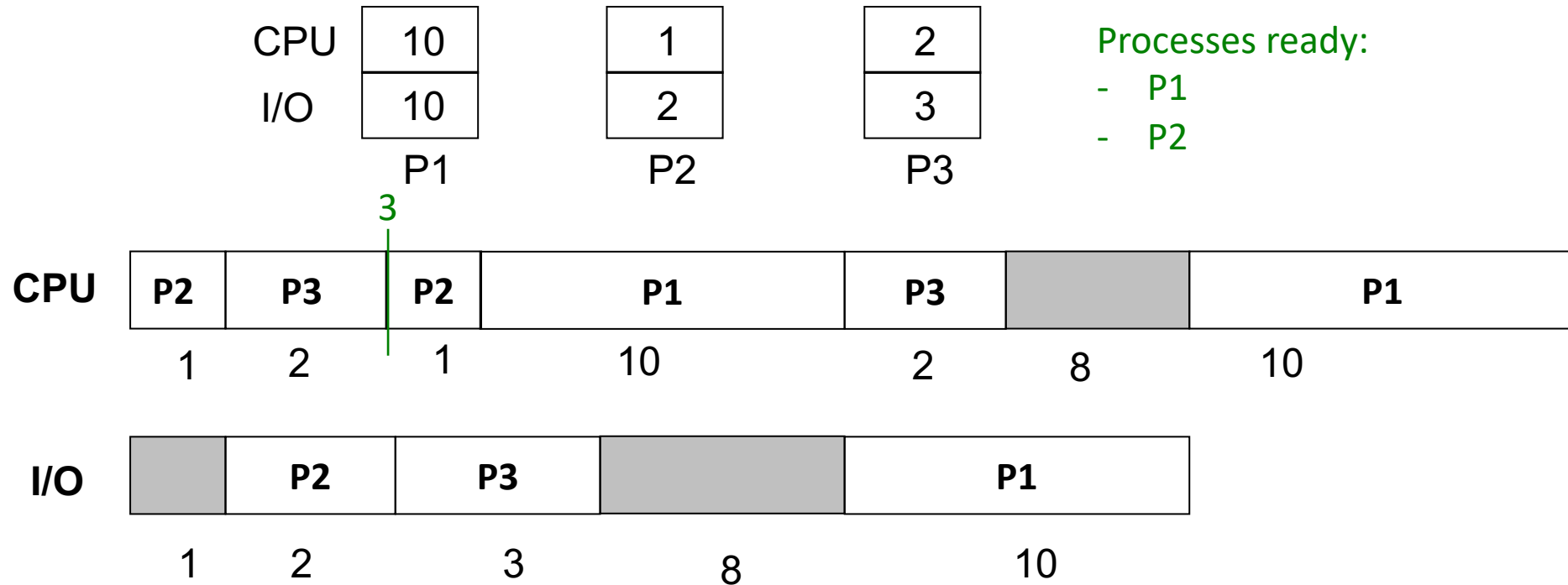
SJF



P1 and P3 are both “ready”

➔ the scheduler picks P3 because it's shortest

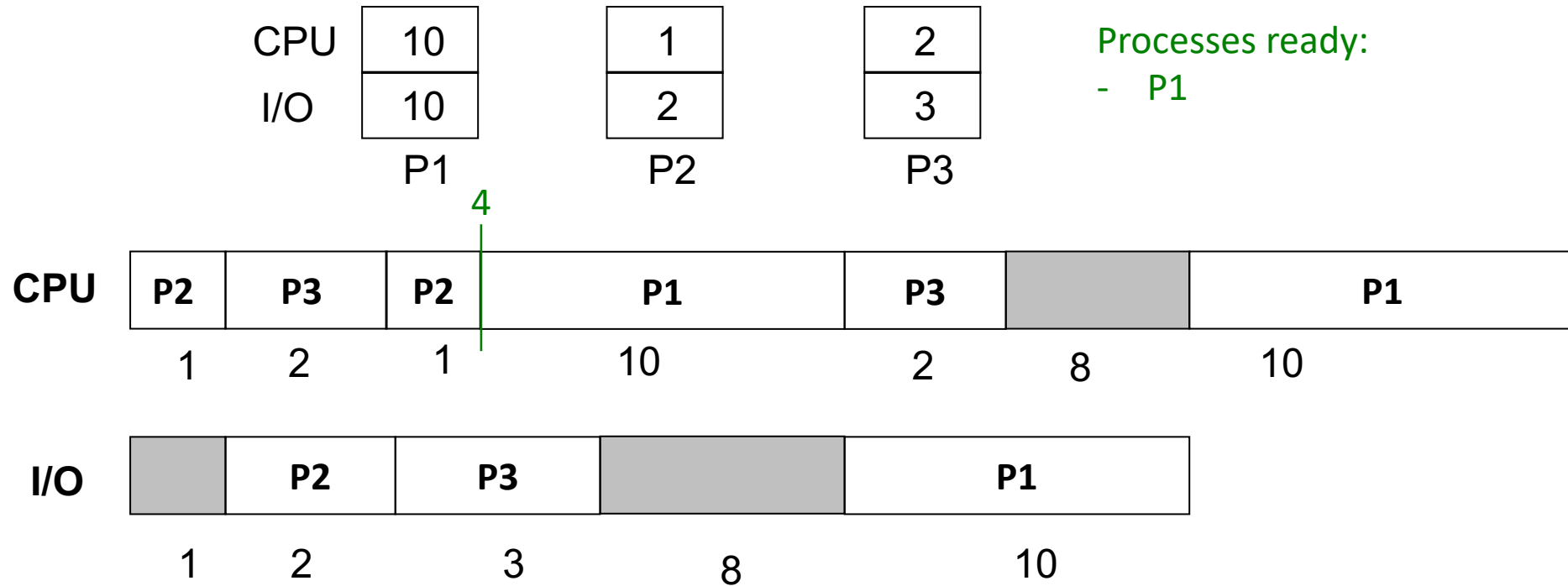
SJF



P1 and P2 are both “ready”

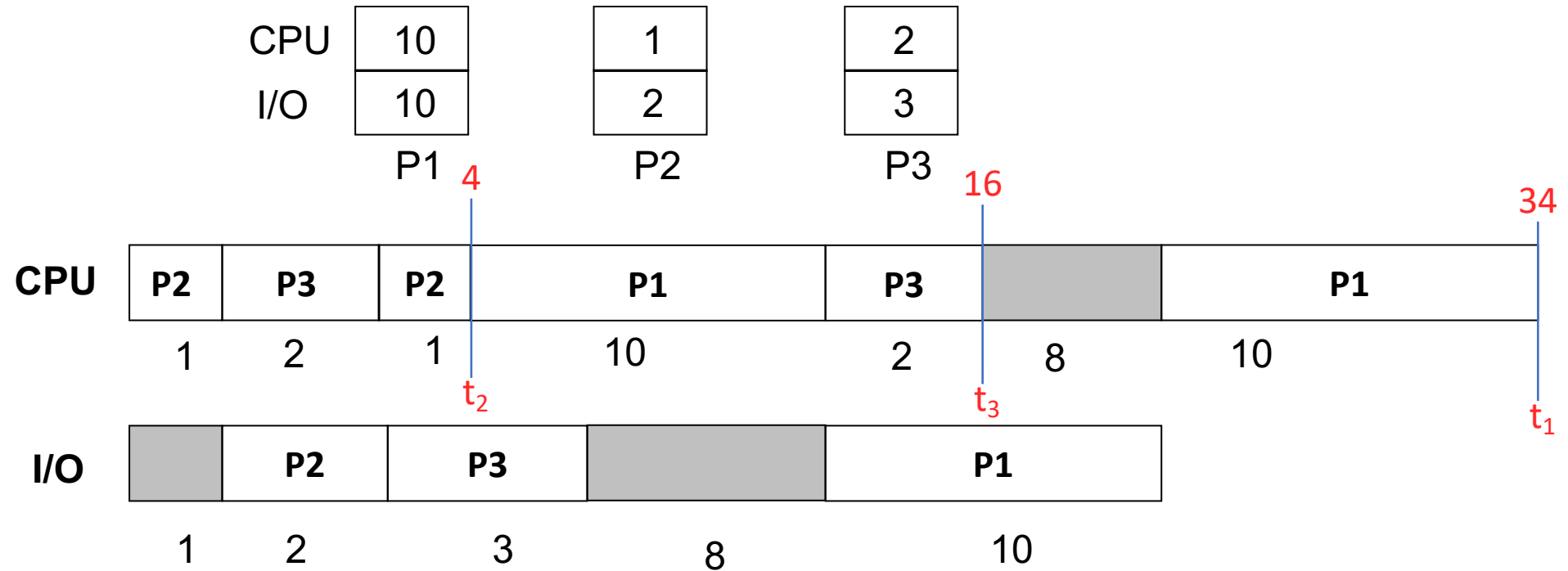
→ the scheduler picks P2 because it's shortest

SJF



Only P1 is ready to run

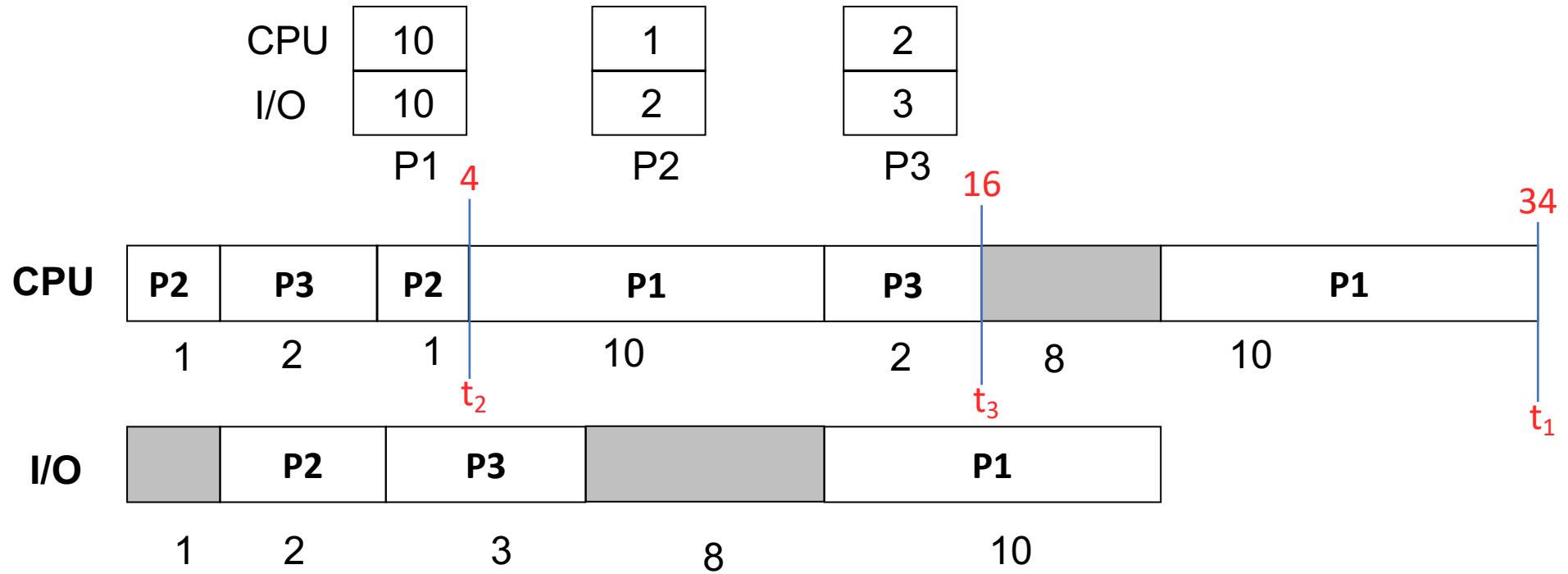
SJF



And so on until we get to the end

So what are the waiting times?

SJF

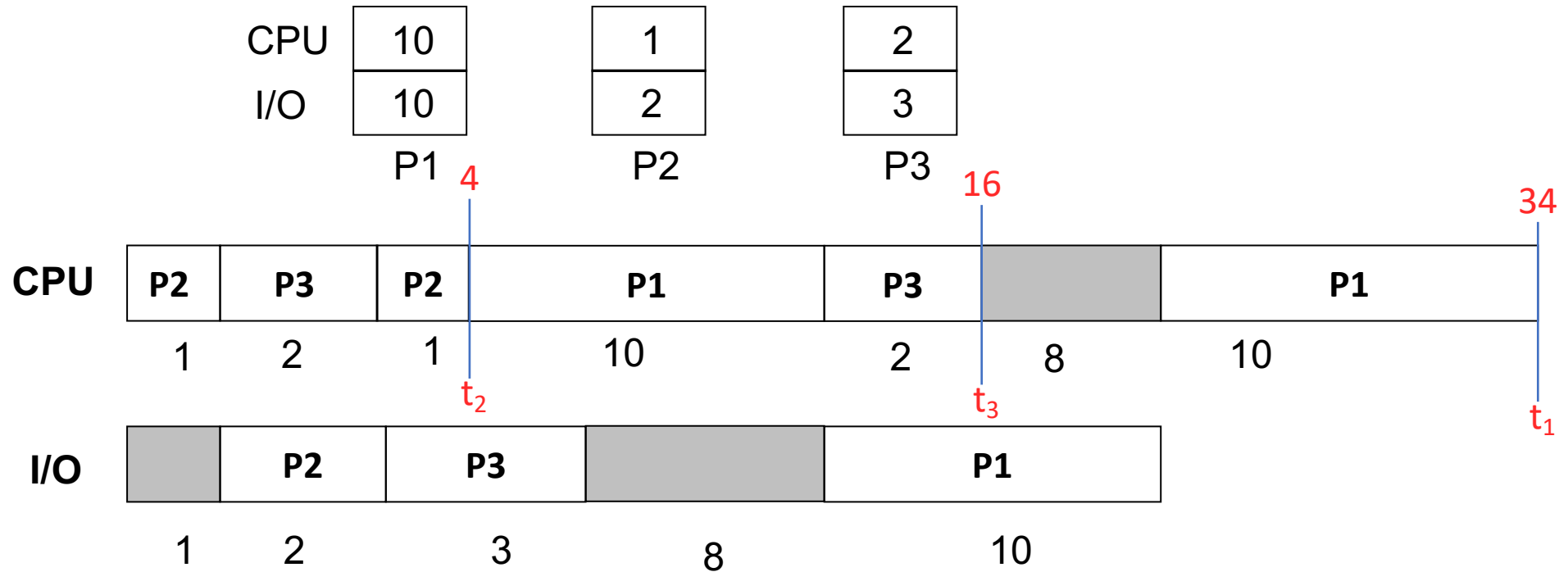


$$e_1 = 30, t_1 = 34$$

$$e_2 = 4, t_2 = 4$$

$$e_3 = 7, t_3 = 16$$

SJF



$$e_1 = 30, t_1 = 34$$

$$w_{P1} = t_1 - e_1$$

$$w_{P1} = 34 - 30$$

$$w_{P1} = 4$$

$$e_2 = 4, t_2 = 4$$

$$w_{P2} = t_2 - e_2$$

$$w_{P2} = 4 - 4$$

$$w_{P2} = 0$$

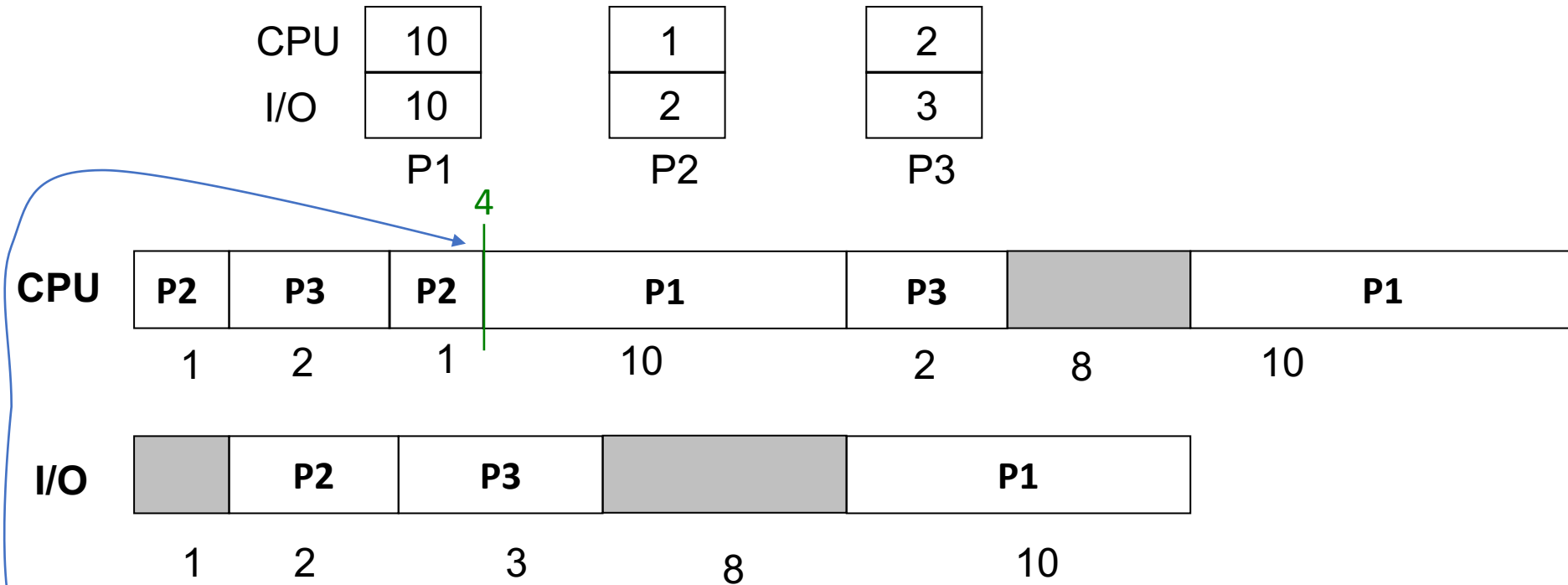
$$e_3 = 7, t_3 = 16$$

$$w_{P3} = t_3 - e_3$$

$$w_{P3} = 16 - 7$$

$$w_{P3} = 9$$

SJF



- Low average waiting time $\rightarrow (4+0+9=)14 / 3$ (FCFS was $53 / 3$)
- Low average turnaround time $\rightarrow (3+4+16=)54 / 3$ (FCFS was $94 / 3$)
- Potential for unfairness \rightarrow starvation



Assuming all the processes arrive at time zero, the throughput of the system is

6% A. 1/11 processes/unit-time

45% B. 3/24 processes/unit-time

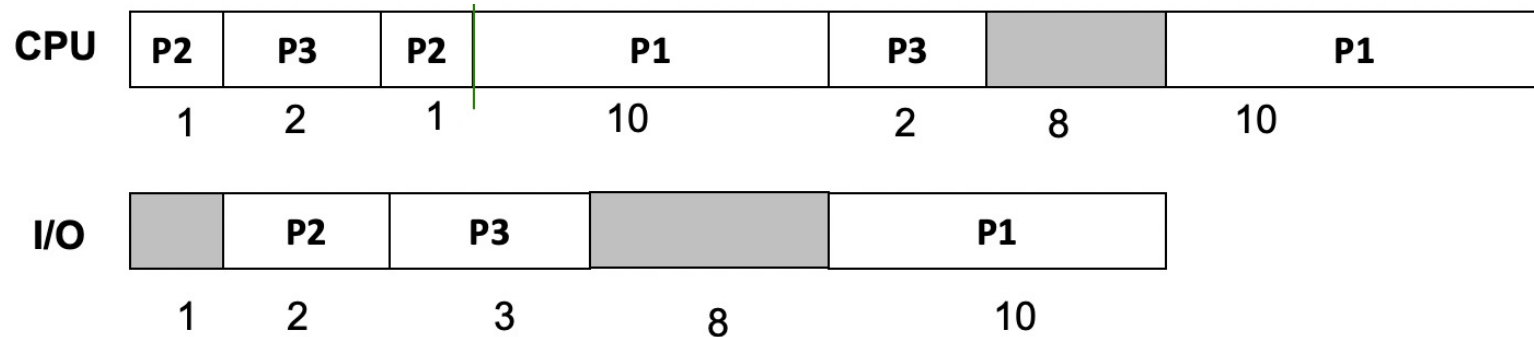
48% C. 3/34 processes/unit-time

Throughput is simply jobs/time, so

3 jobs

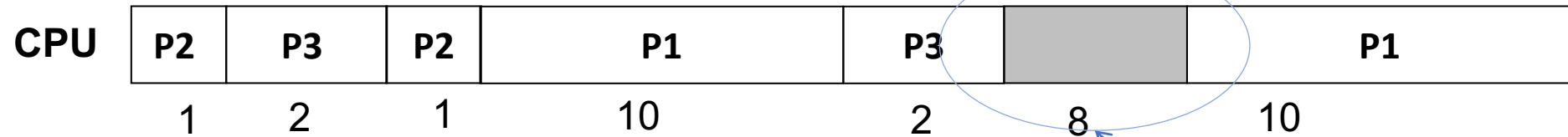
34 seconds of elapsed time

3/34 is the throughput

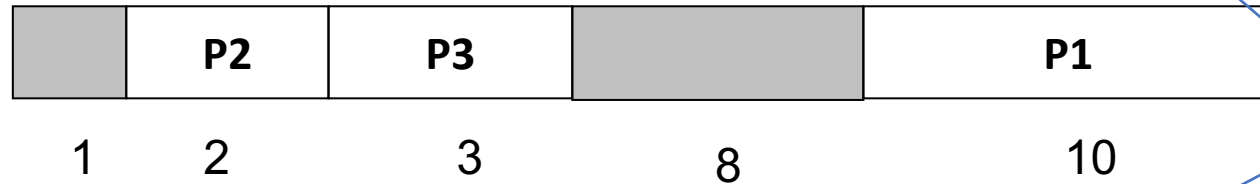


SJF vs. FCFS

SJF

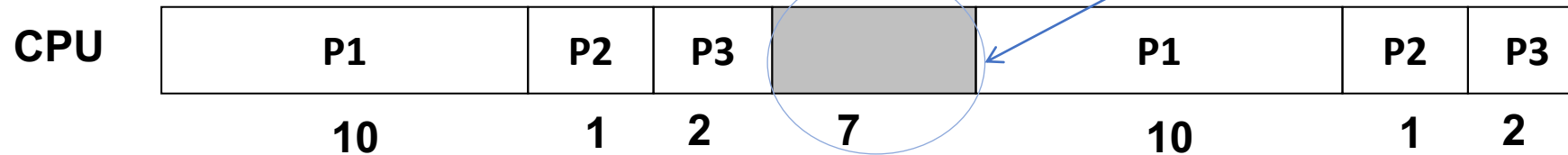


I/O

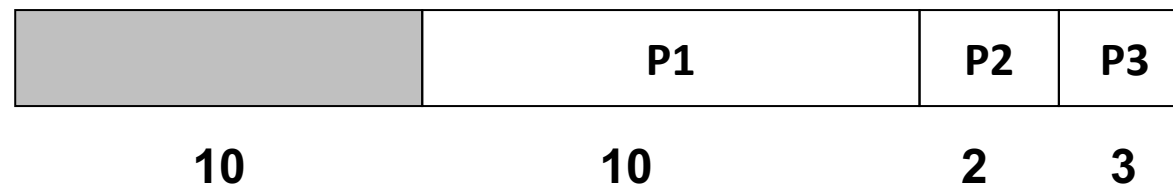


Note SJF slightly underutilizes the CPU

FCFS



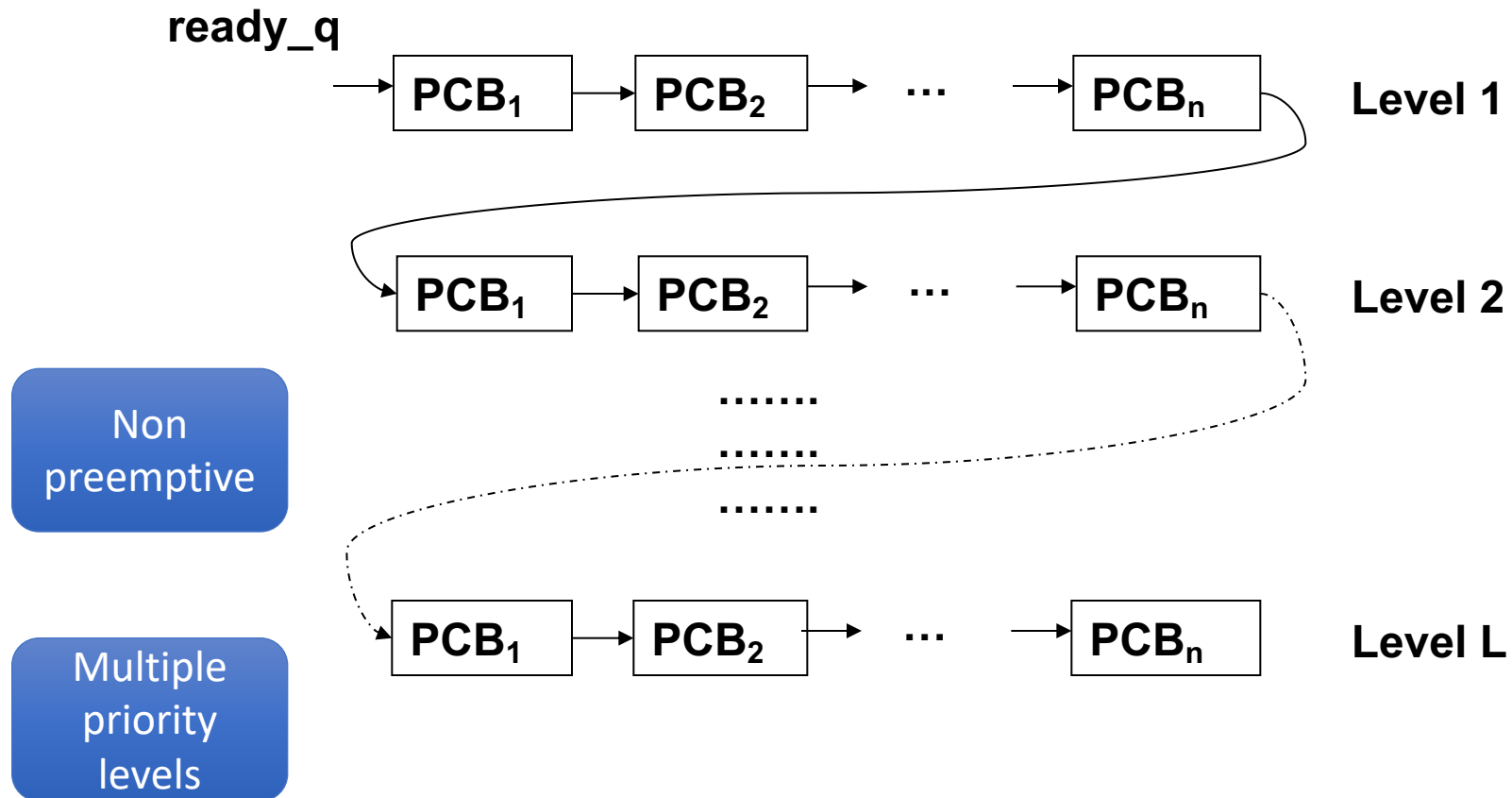
I/O



Non-preemptive scheduling algorithms

- FCFS
 - SJF
 - Priority
- } Intrinsic property
- Extrinsic property

(Extrinsic) Priority scheduler




Preemptive Scheduling

- Yank processor from currently running process at an “opportune moment” to give it to a “higher priority” process
- Questions
 - What are “opportune moments”?
 - How can we determine “higher priority”?

Preemptive Schedulers

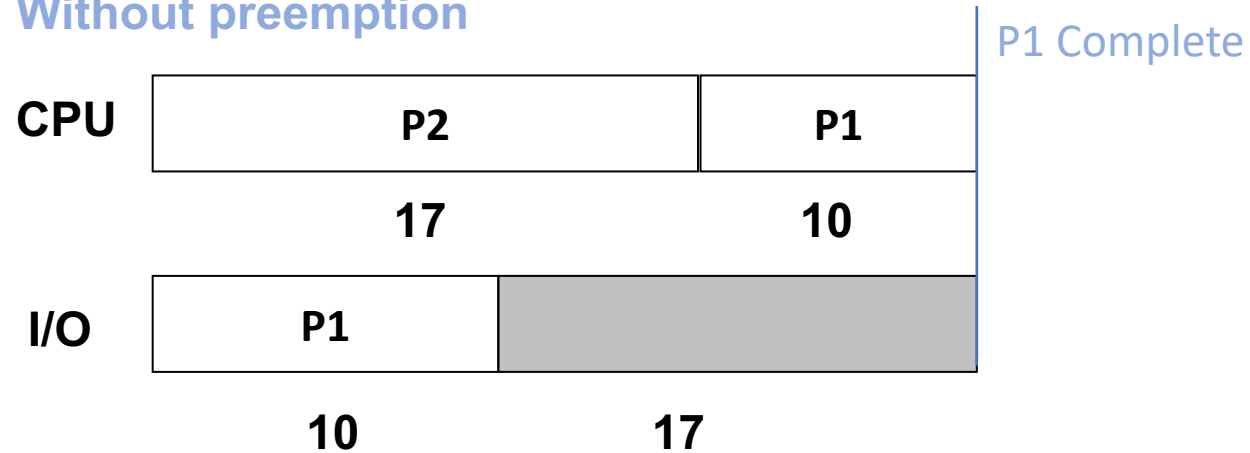
- FCFS with preemption
- SJF with preemption
 - SRTF (Shortest Remaining Time First)
- Priority with preemption
- Round robin



One opportune moment is when a process rejoins the ready queue after I/O completion

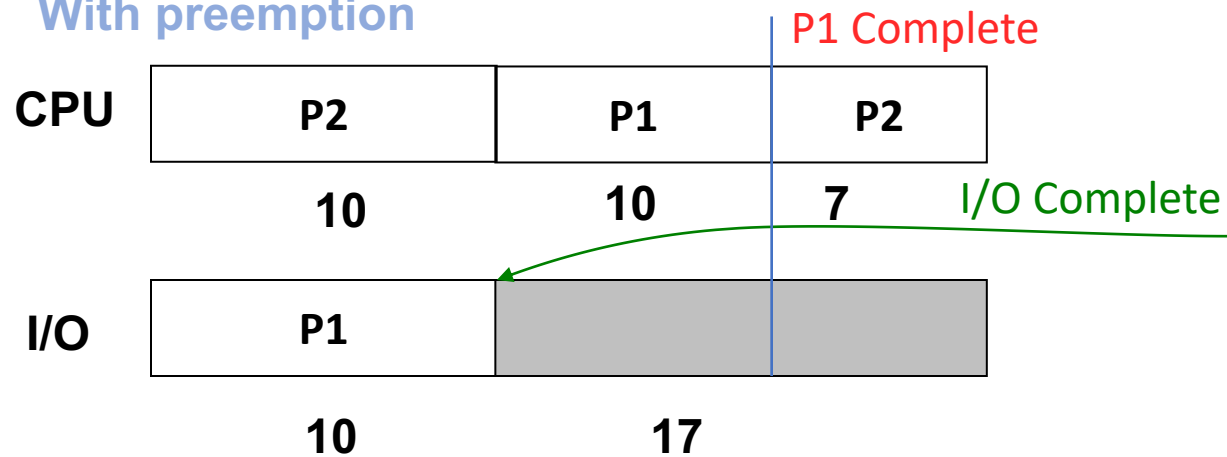
FCFS with preemption

Without preemption



Assume P1 has earlier arrival time than P2

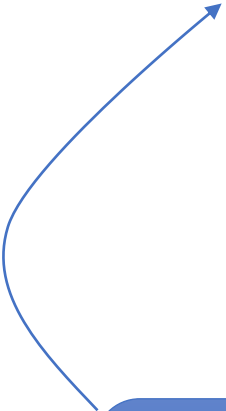
With preemption



- Deschedule P2
- Schedule P1

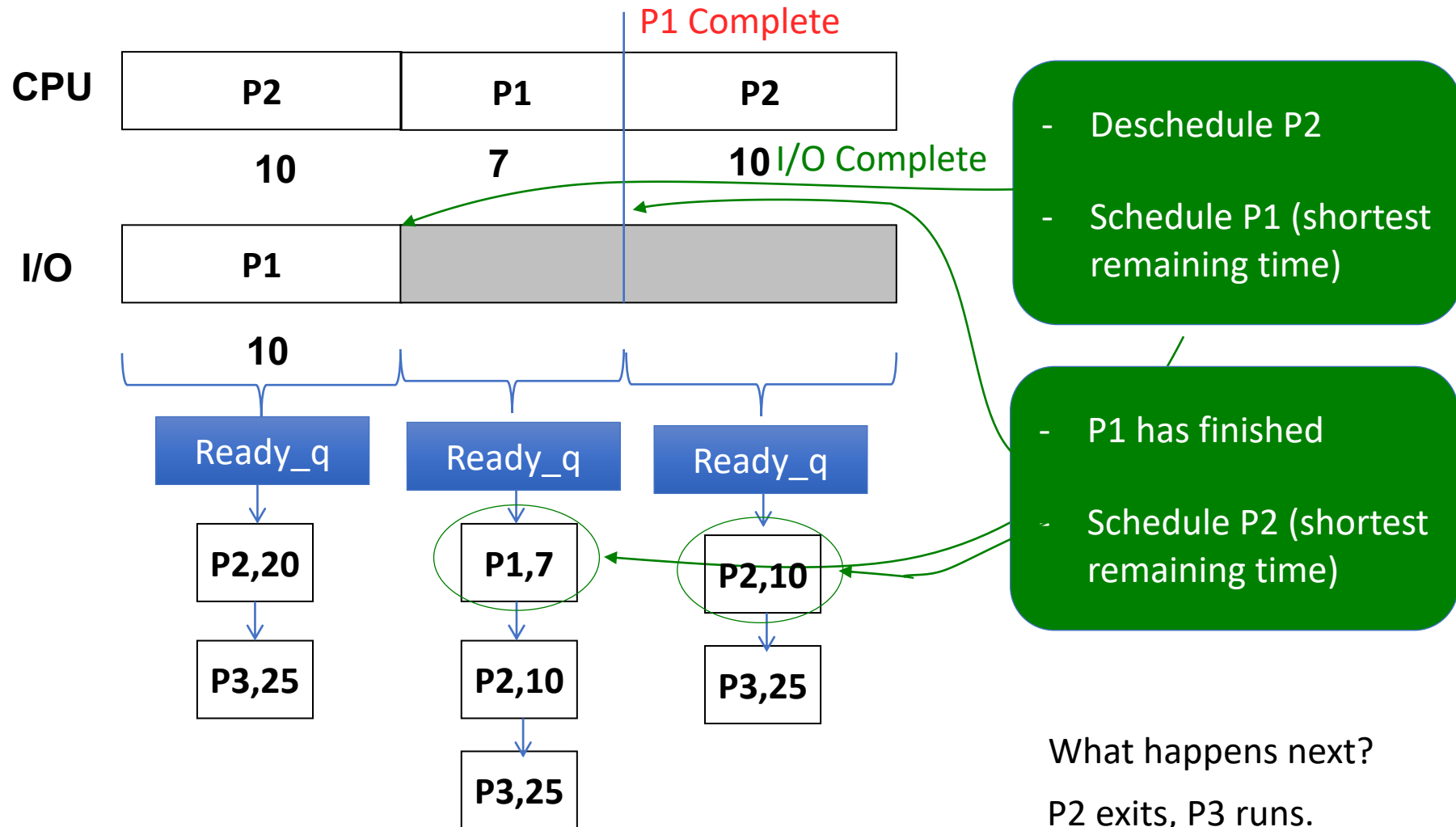
Preemptive Schedulers

- ~~FCFS with preemption~~
- SJF with preemption
 - SRTF (Shortest Remaining Time First)
- Priority with preemption
- Round robin

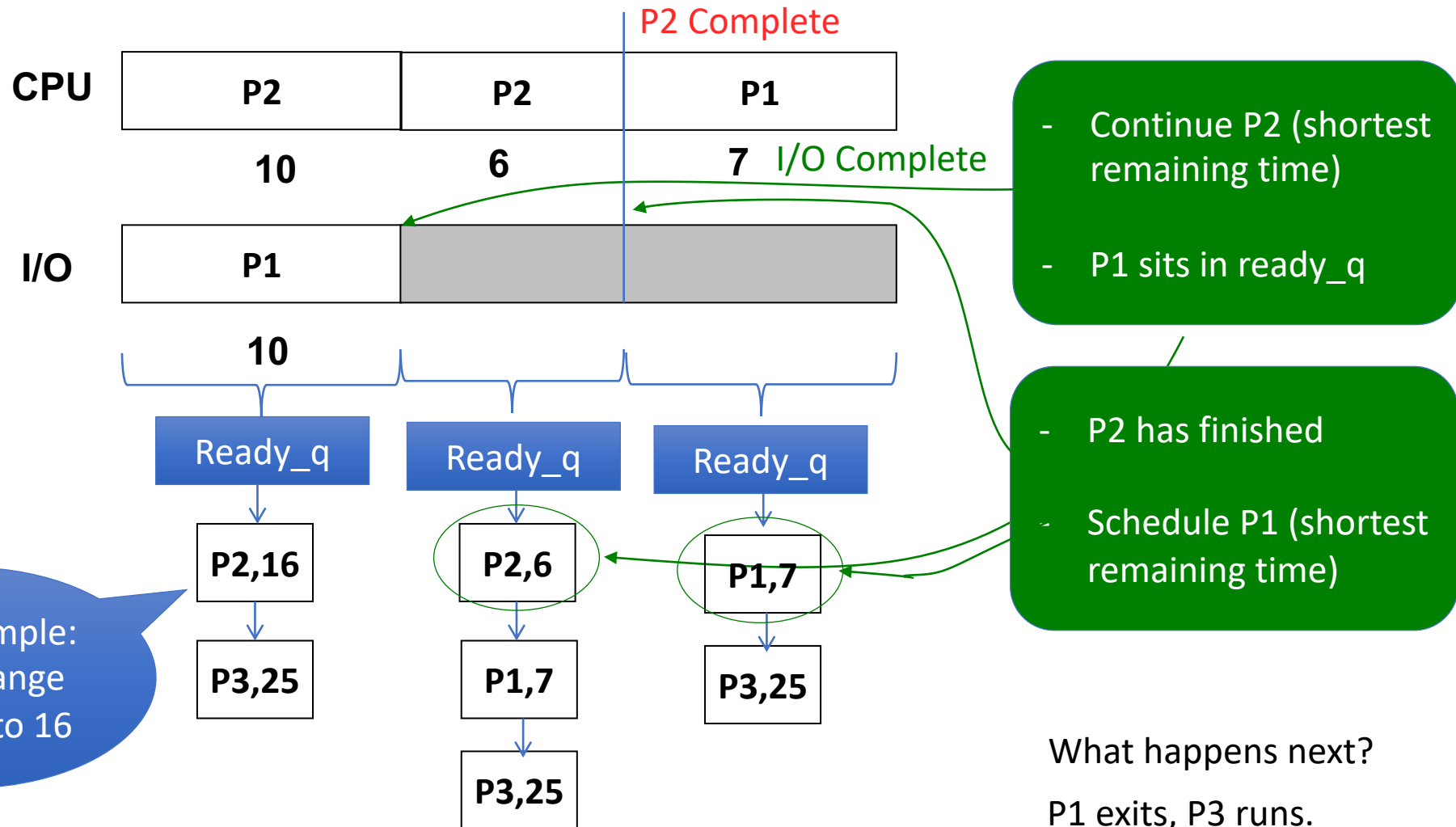


One opportune moment is when a process rejoins the ready queue after I/O completion; estimate remaining time to make preemption decision

SRTF with preemption

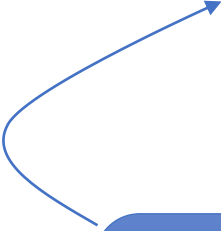


SRTF with preemption



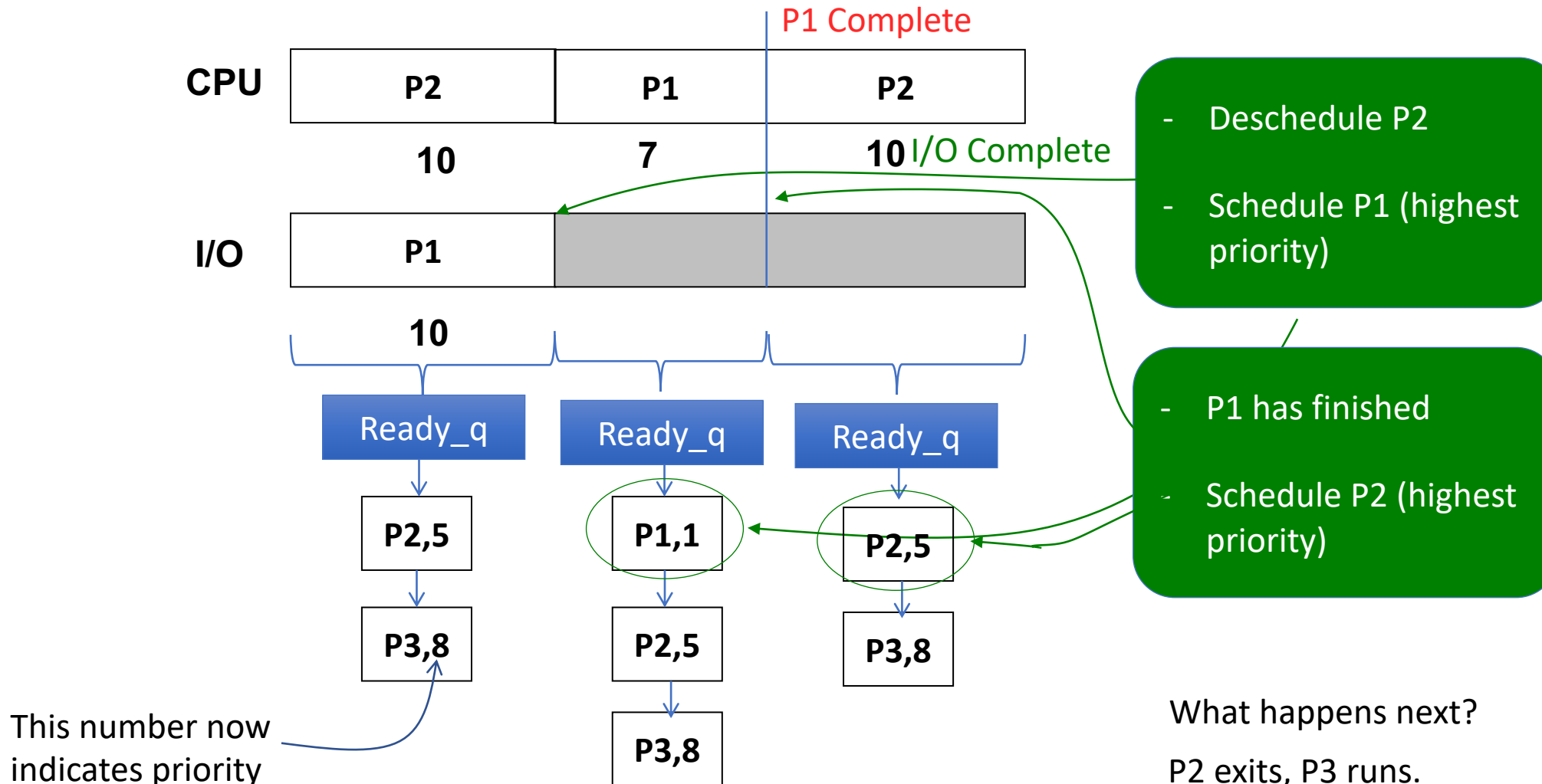
Preemptive Schedulers

- ~~FCFS with preemption~~
- ~~SJF with preemption~~
 - ~~SRTF (Shortest Remaining Time First)~~
- Priority with preemption
- Round robin



Reevaluate priority on each I/O completion (i.e., whenever there's a change in the Ready Queue)

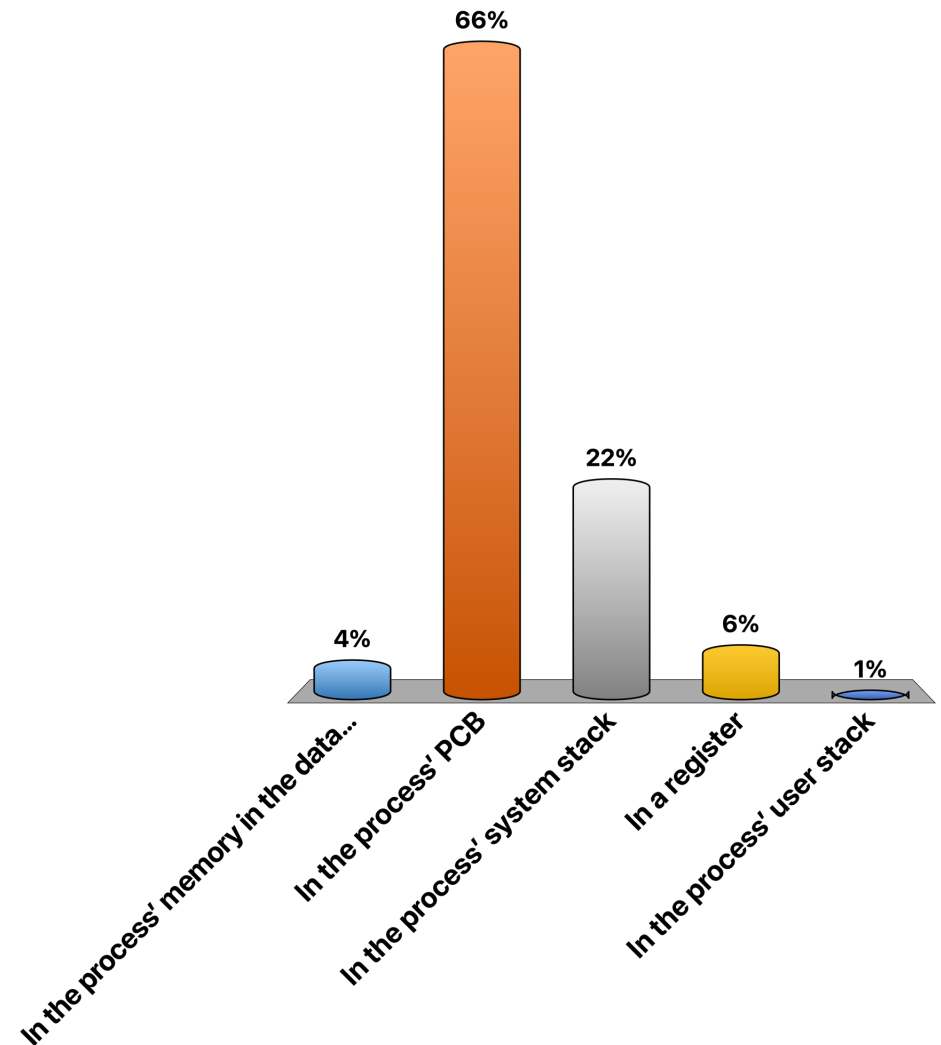
Priority with preemption





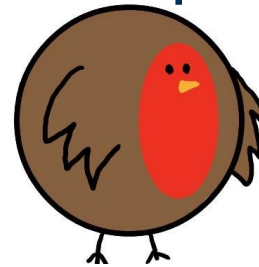
For priority scheduling, where would we store the priority for each process?

- A. In the process' memory in the data area
- B. In the process' PCB
- C. In the process' system stack
- D. In a register
- E. In the process' user stack



Preemptive Schedulers

- ~~FCFS with preemption~~
- ~~SJF with preemption~~
 - ~~SRTF (Shortest Remaining Time First)~~
- ~~Priority with preemption~~
- Round robin

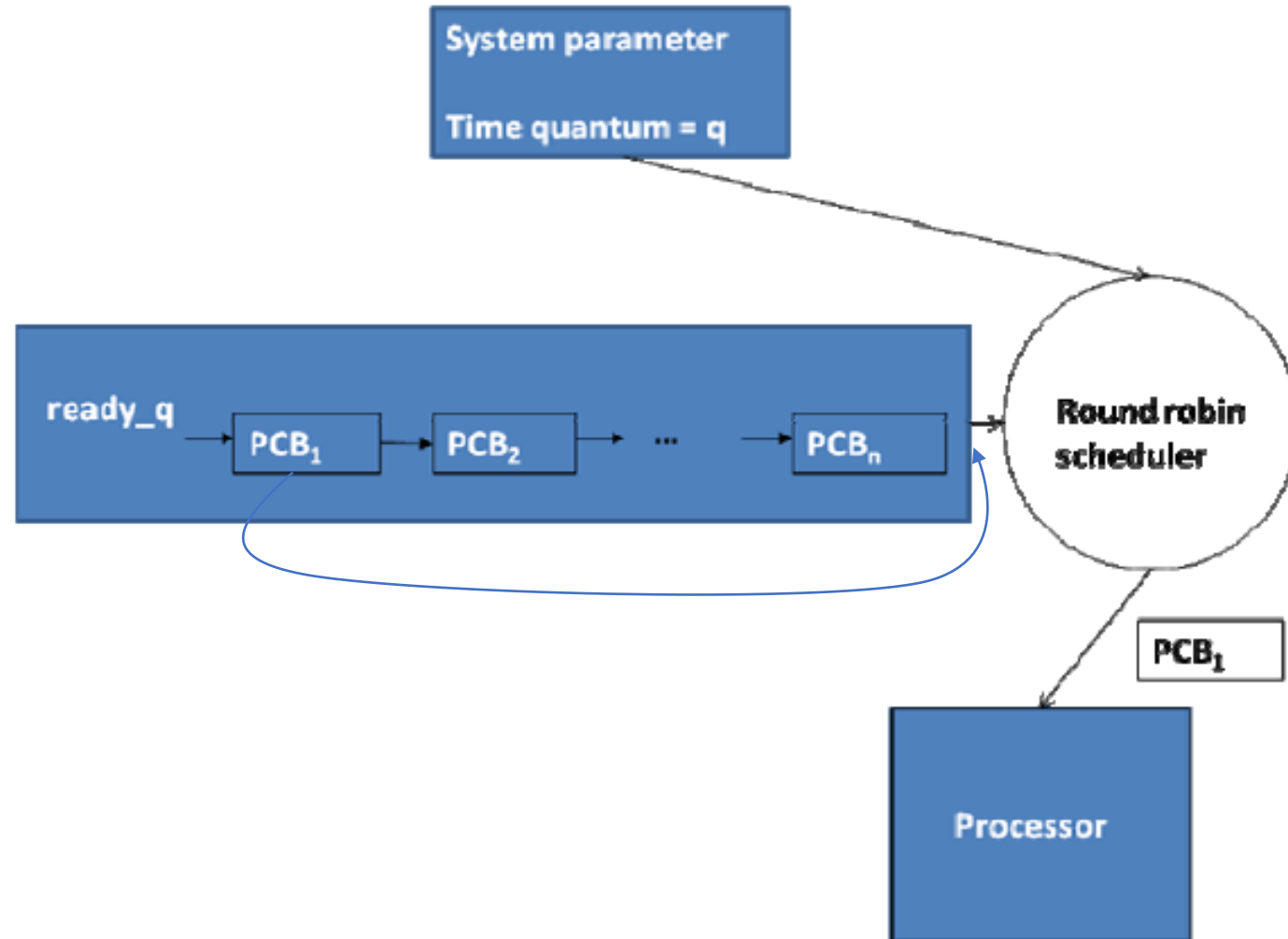


One opportune moment is when a timer interrupts; if process has used its allotted time quantum, give the next process a chance

Round Robin

- RR is preemptive and requires a timer interrupt
- When a process starts, it is given a time quantum (time slice) which limits the continuous CPU time it may use
- When a process is dispatched, the timer is set to interrupt at the end of the remaining time quantum
- If a process uses up its remaining time quantum
 - The process is interrupted
 - The scheduler is called to put the process at the end of the ready list
 - The process' remaining time quantum is reset

Round Robin



Recall: 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;
```

Recall: 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 time_left;
    address memory_footprint;
    ...
    ...
} control_block;
```

RR example

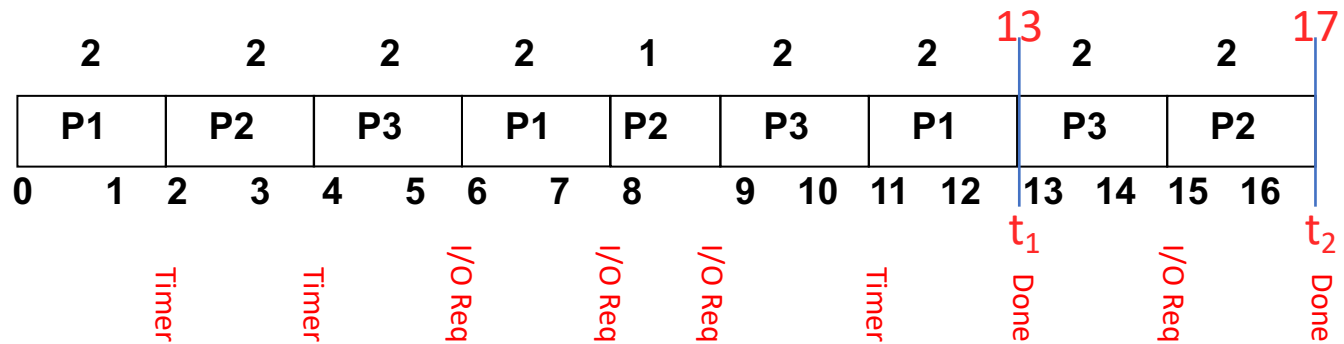
What is the wait time for each of the three processes with round robin scheduling and timeslice = 2?

	CPU	I/O	CPU	I/O
P1	4	2	2	
P2	3	2	2	
P3	2	2	4	2

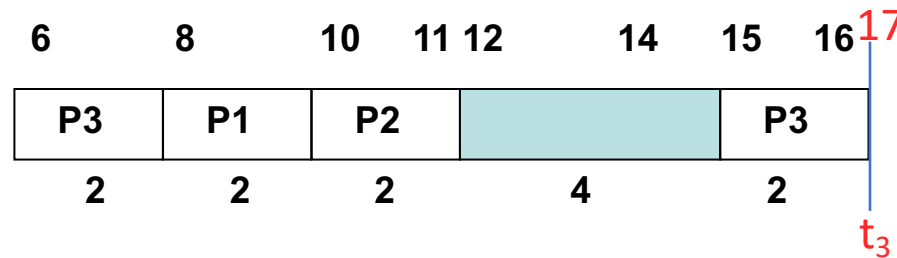
RR solution sketch

	CPU	I/O	CPU	I/O
P1	4	2	2	
P2	3	2	2	
P3	2	2	4	2

CPU Schedule (Round Robin, timeslice = 2)



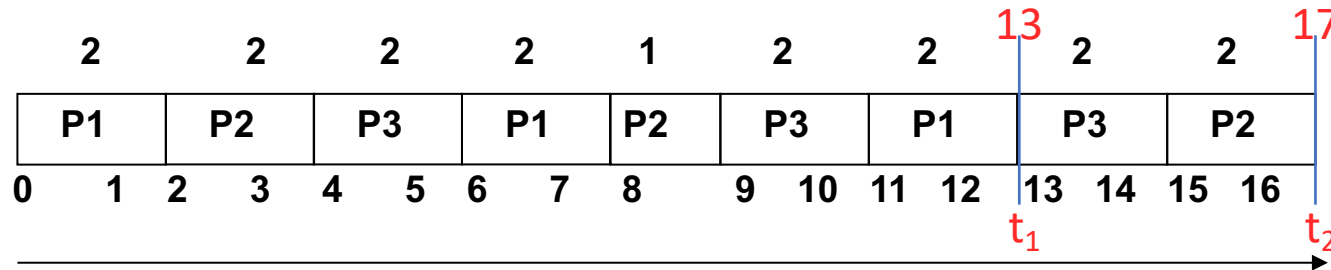
I/O Schedule



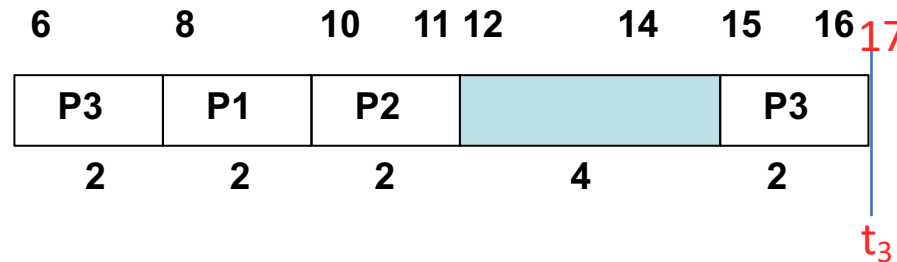
RR solution sketch

	CPU	I/O	CPU	I/O
P1	4	2	2	
P2	3	2	2	
P3	2	2	4	2

CPU Schedule (Round Robin, timeslice = 2)



I/O Schedule



$$w_1 = t_1 - e_1$$

$$e_1 = 2+2+2+2, t_1=13$$

$$w_1 = 13 - 8 = 5$$

$$w_2 = t_2 - e_2$$

$$e_2 = 2+1+2+2, t_2=17$$

$$w_2 = 17 - 7 = 10$$

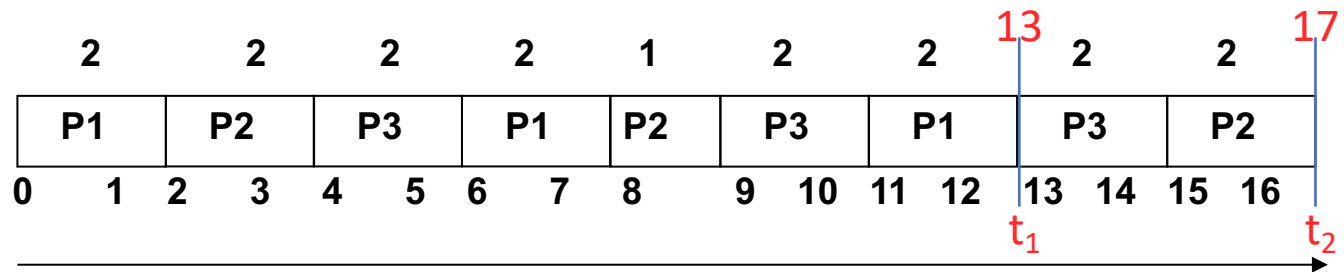
$$w_3 = t_3 - e_3$$

$$e_3 = 2+2+2+2+2, t_3=17$$

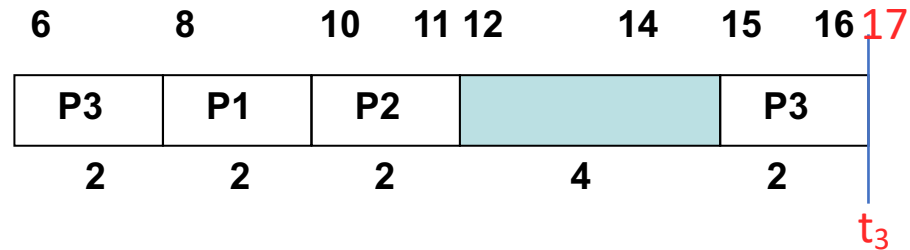
$$w_3 = 17 - 10 = 7$$

RR solution sketch

CPU Schedule (Round Robin, timeslice = 2)



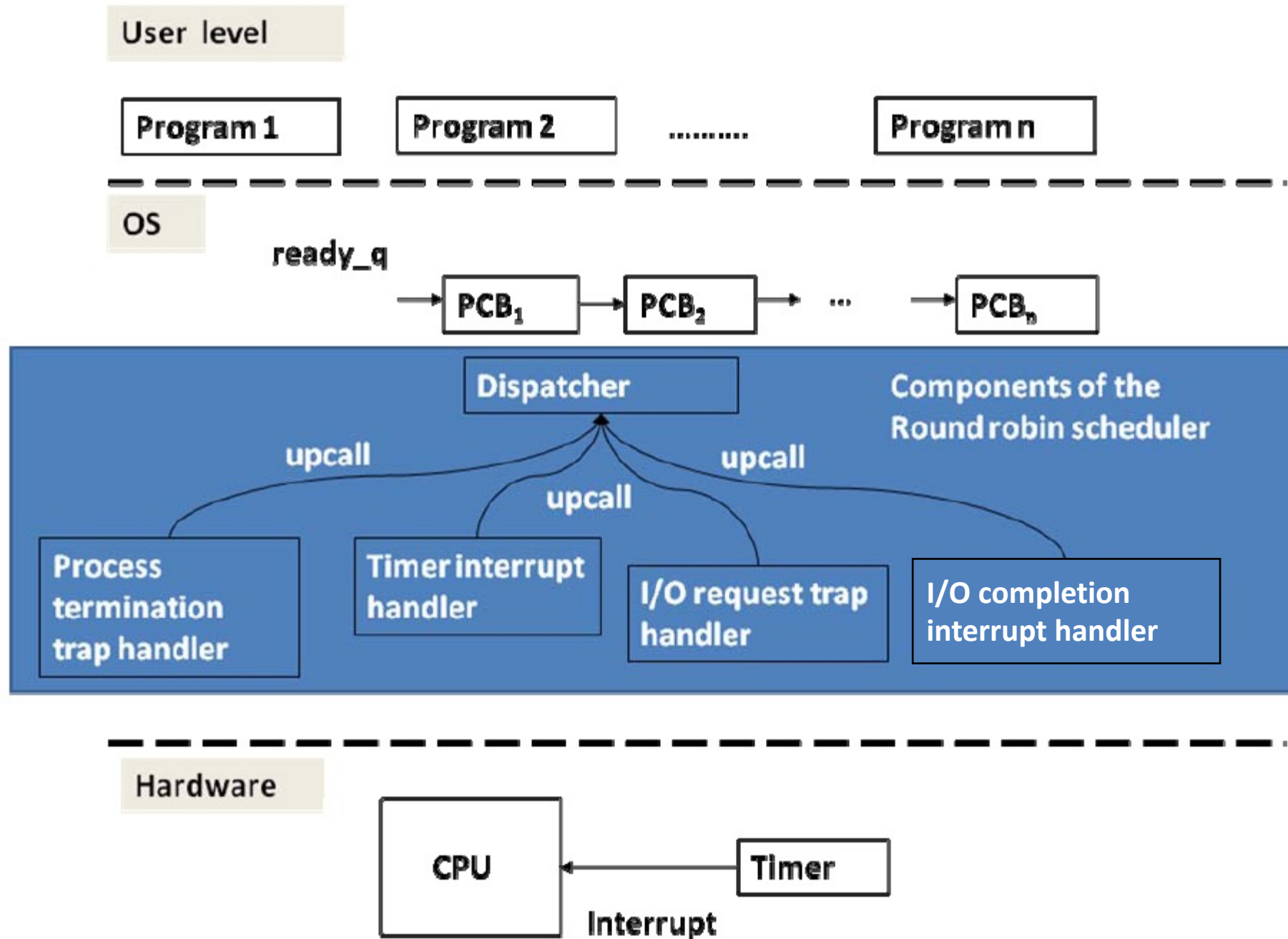
I/O Schedule



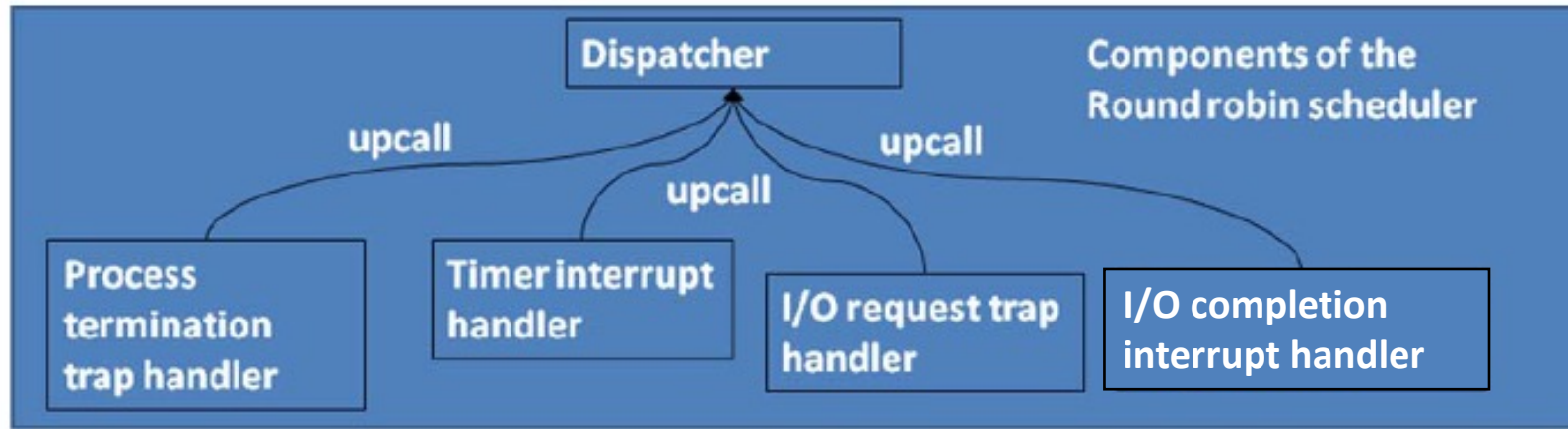
- Potential for unfairness? → no starvation, no convoy effect
- Average waiting time → $(5+7+10=)22 / 3$
- Average turnaround time → $(13+17+17=)47 / 3$

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



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;

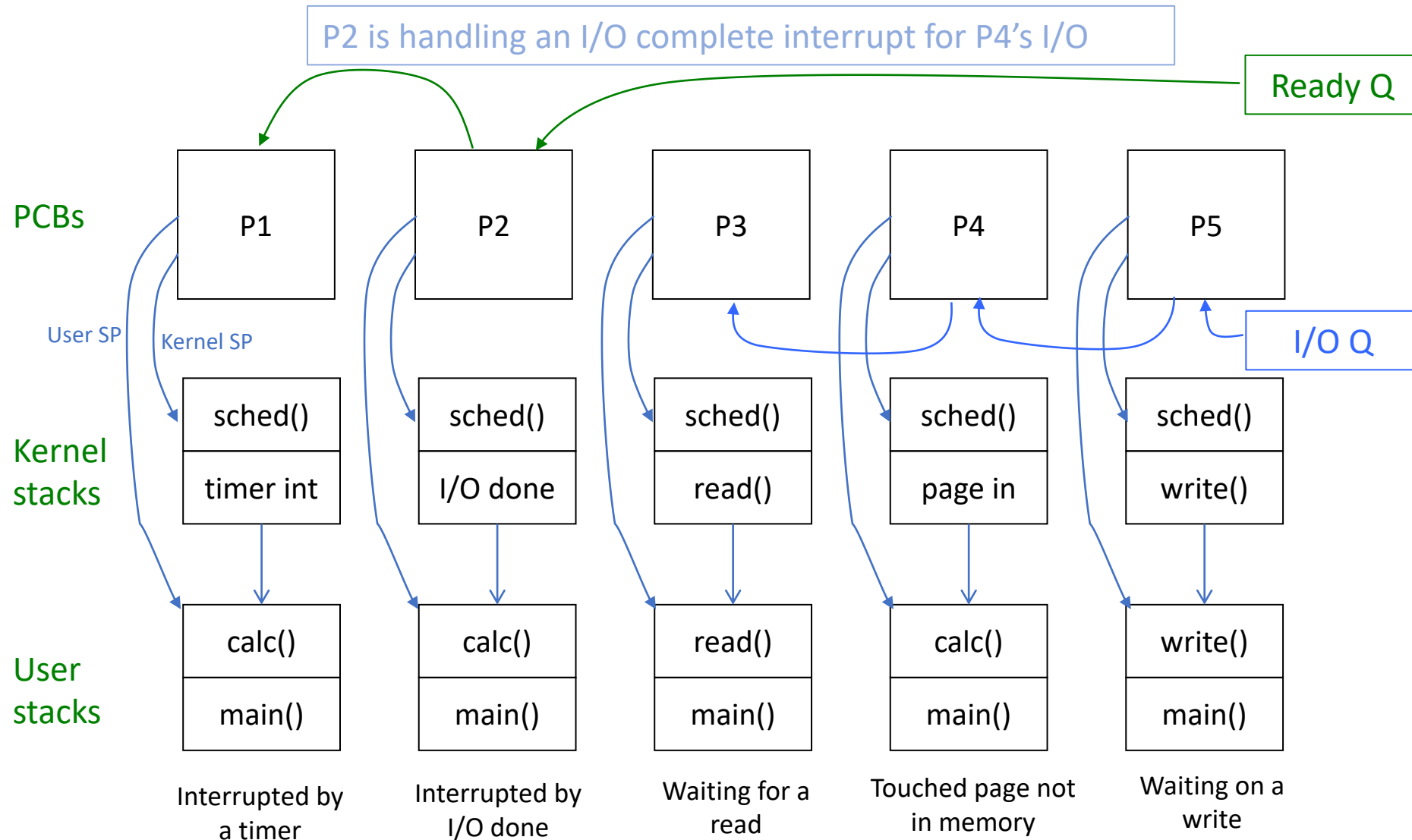
Our Example Program

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



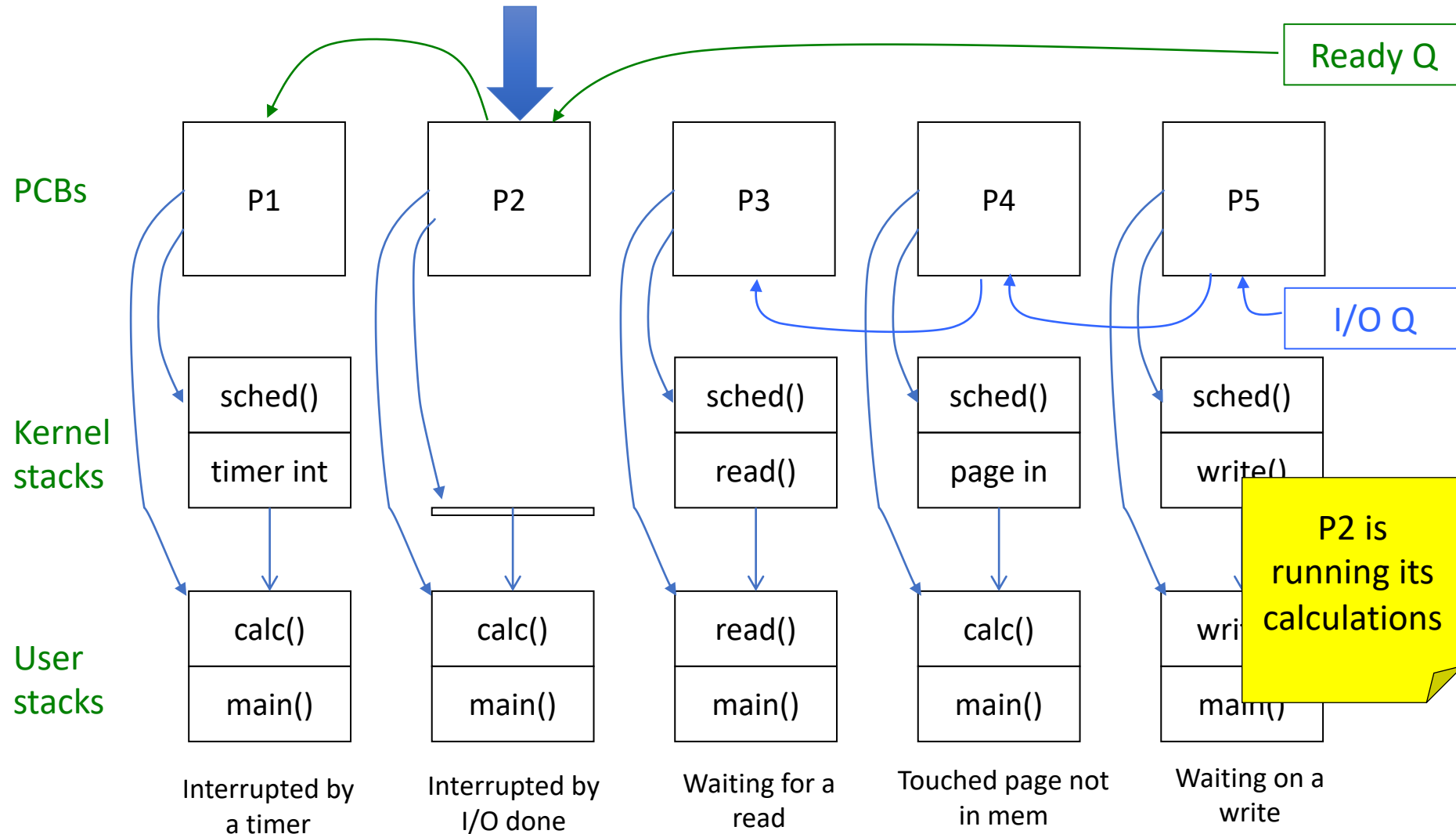
An example using the previous diagram

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

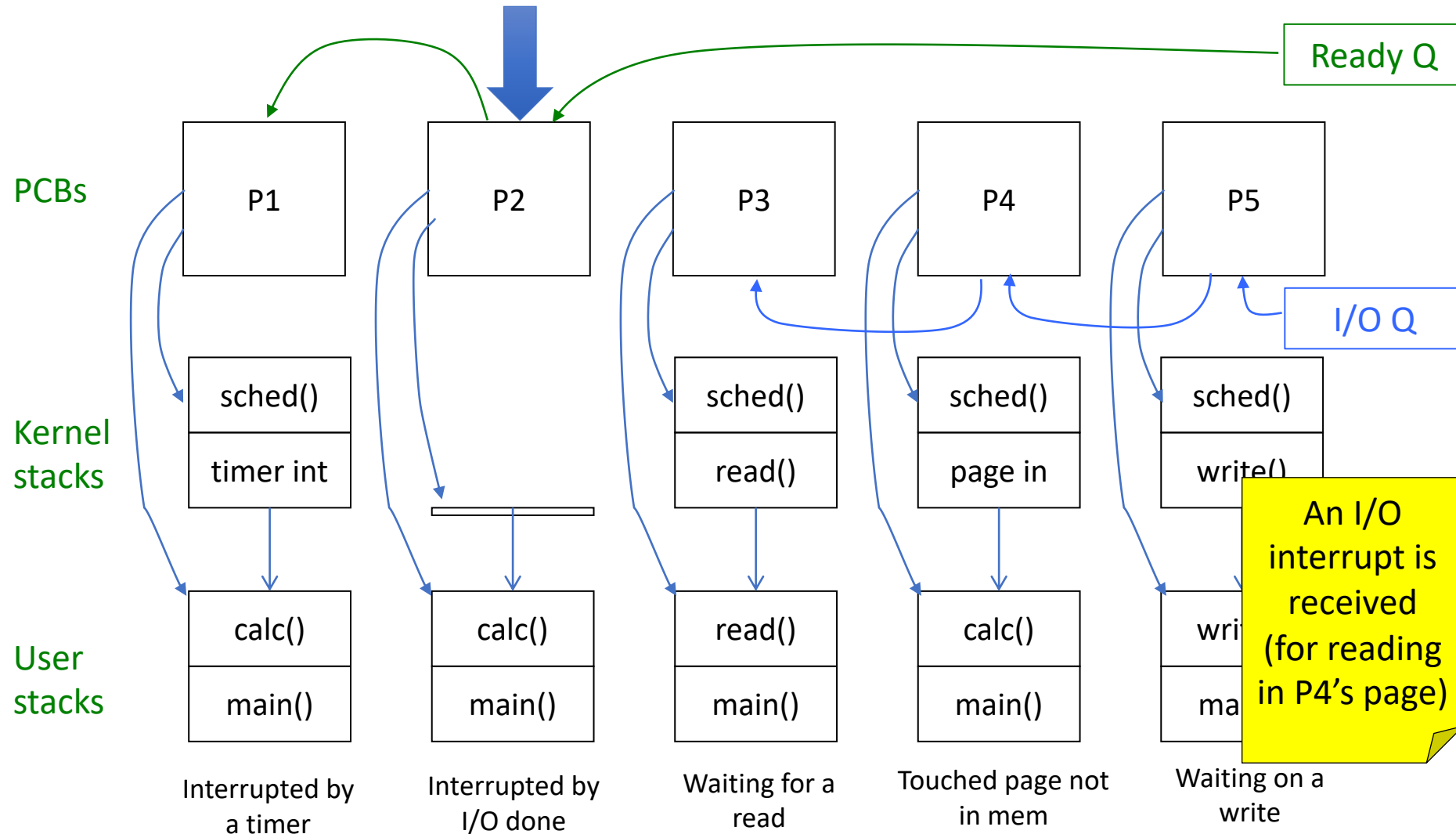
Let's do the example again

- 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

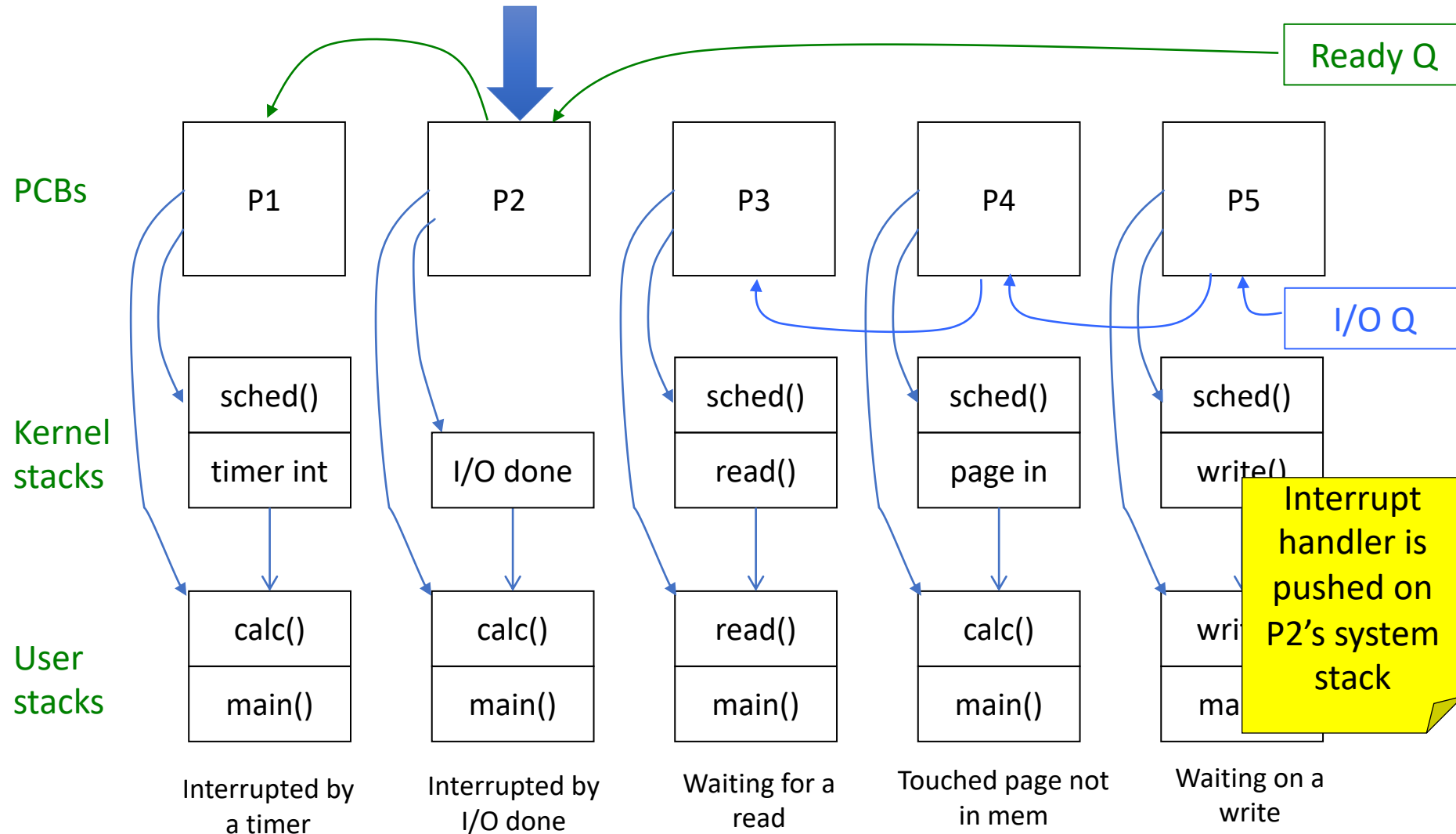
Process example in detail!



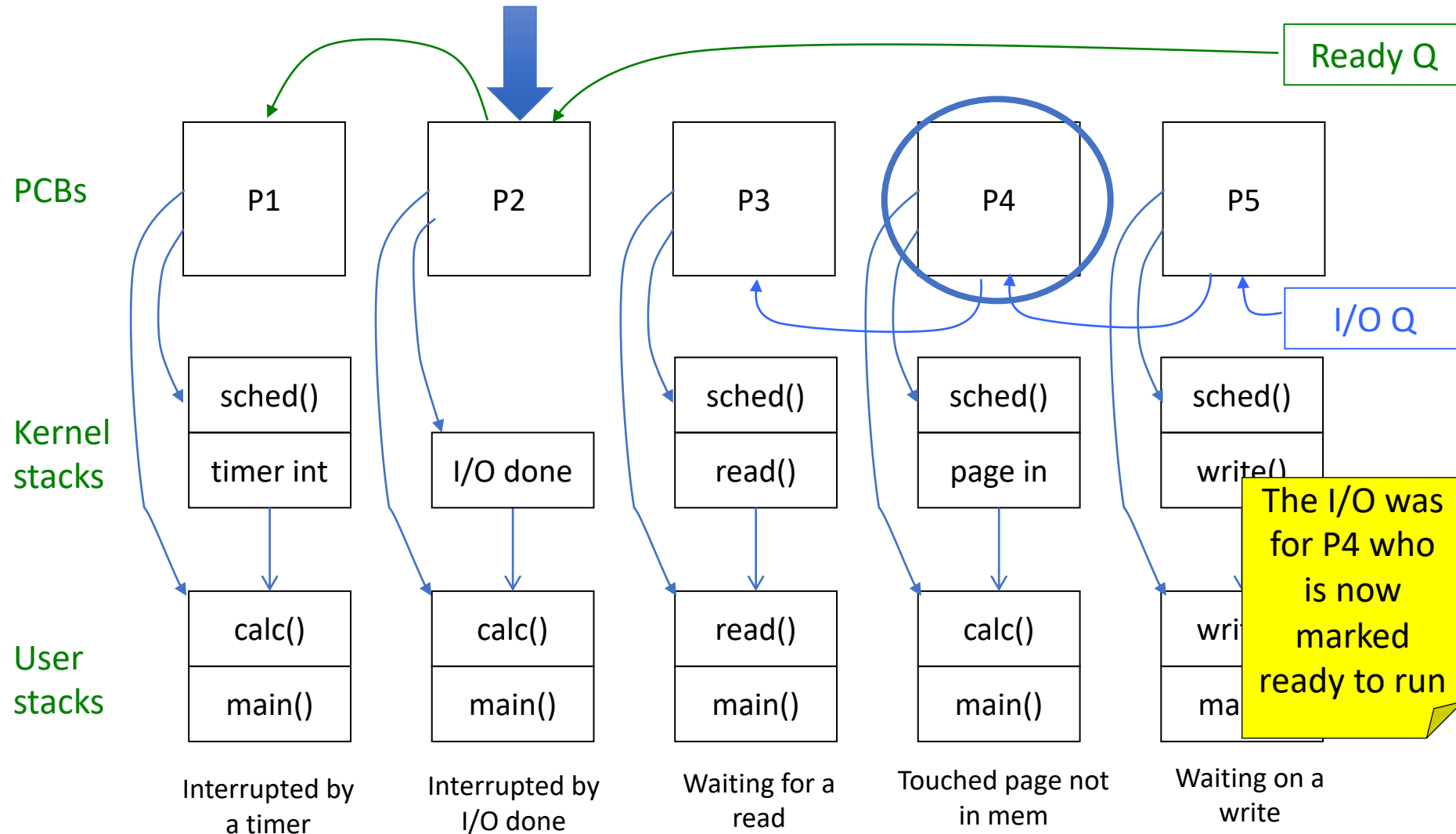
Process example



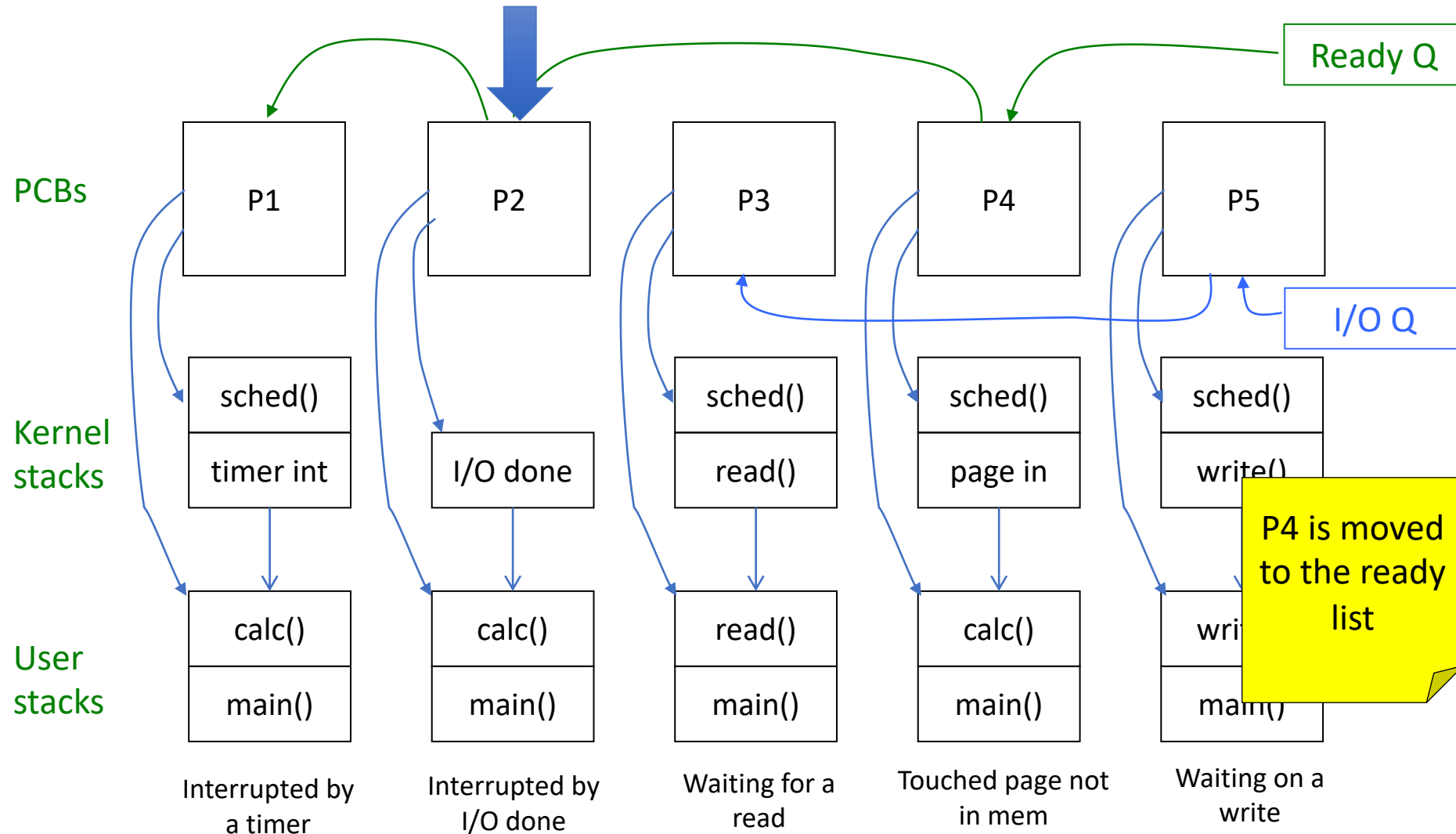
Process example



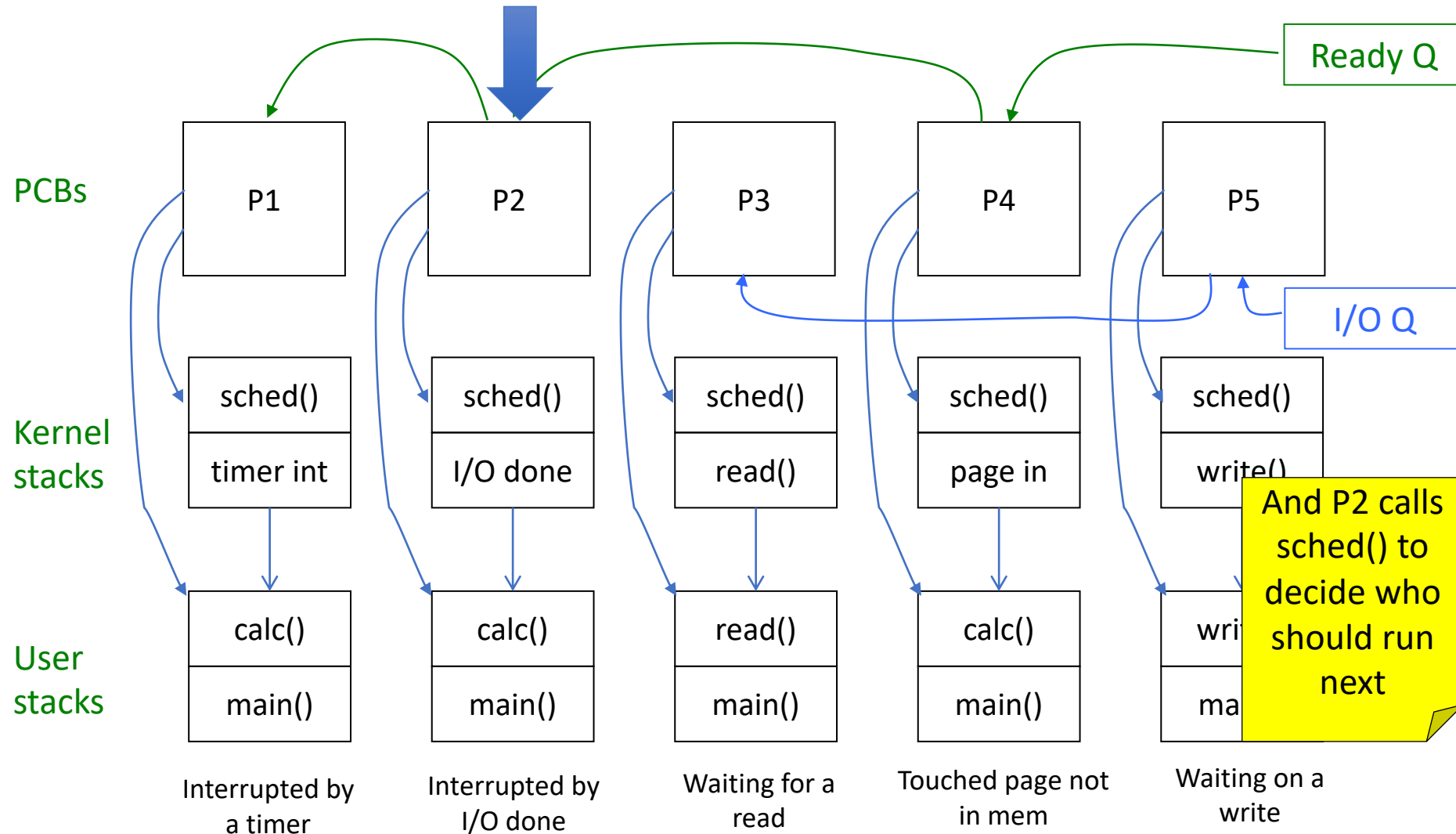
Process example



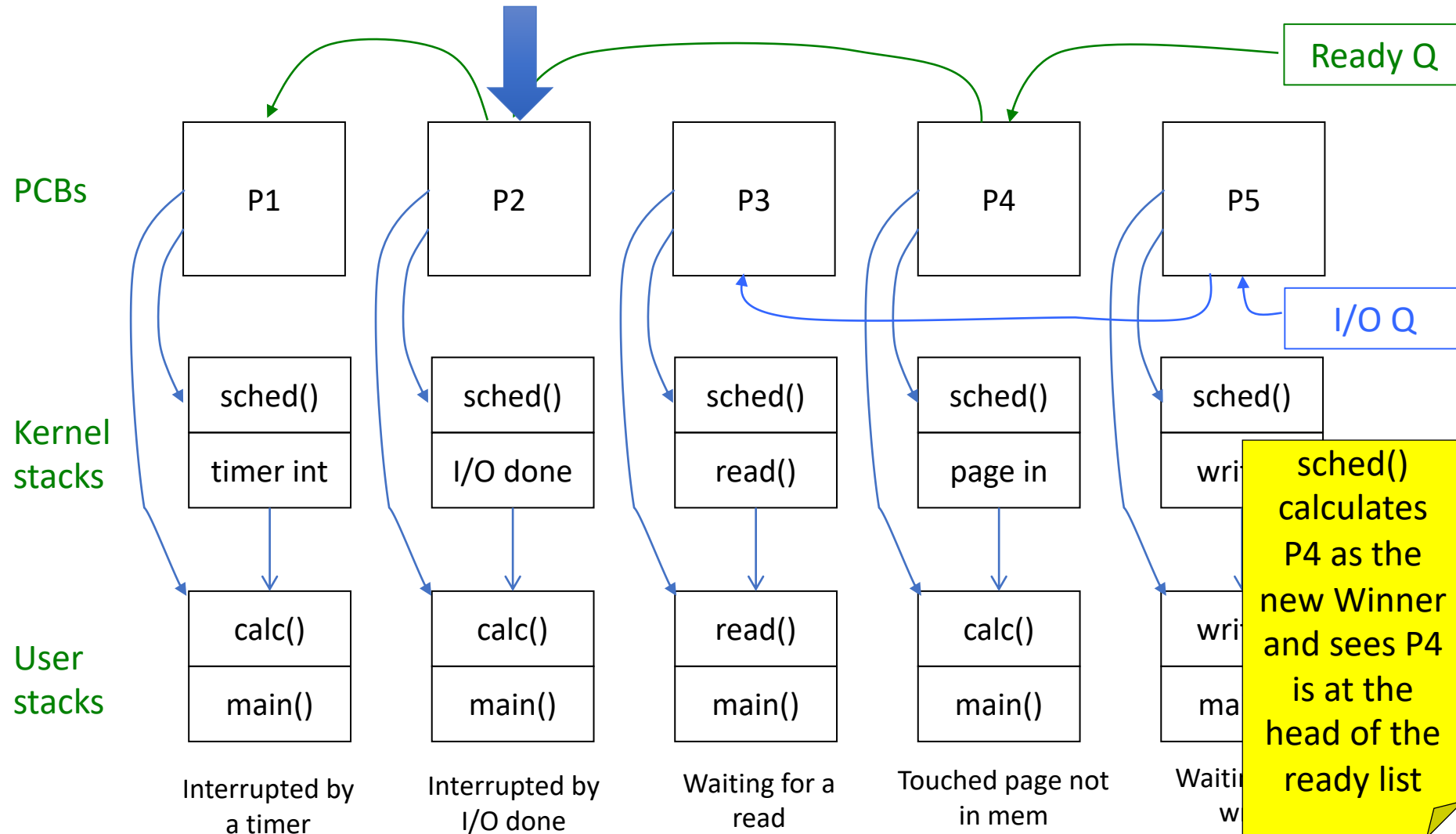
Process example



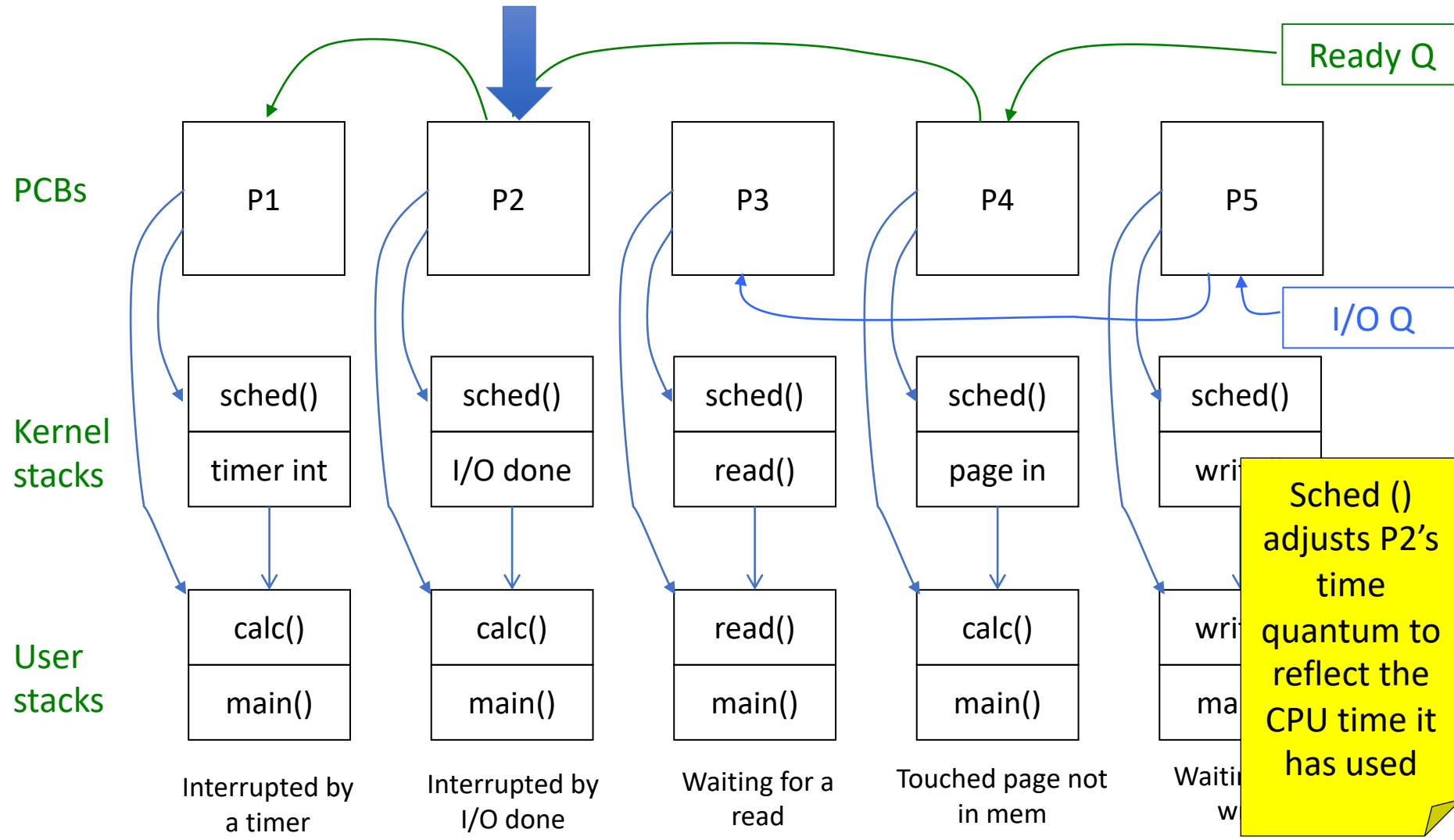
Process example



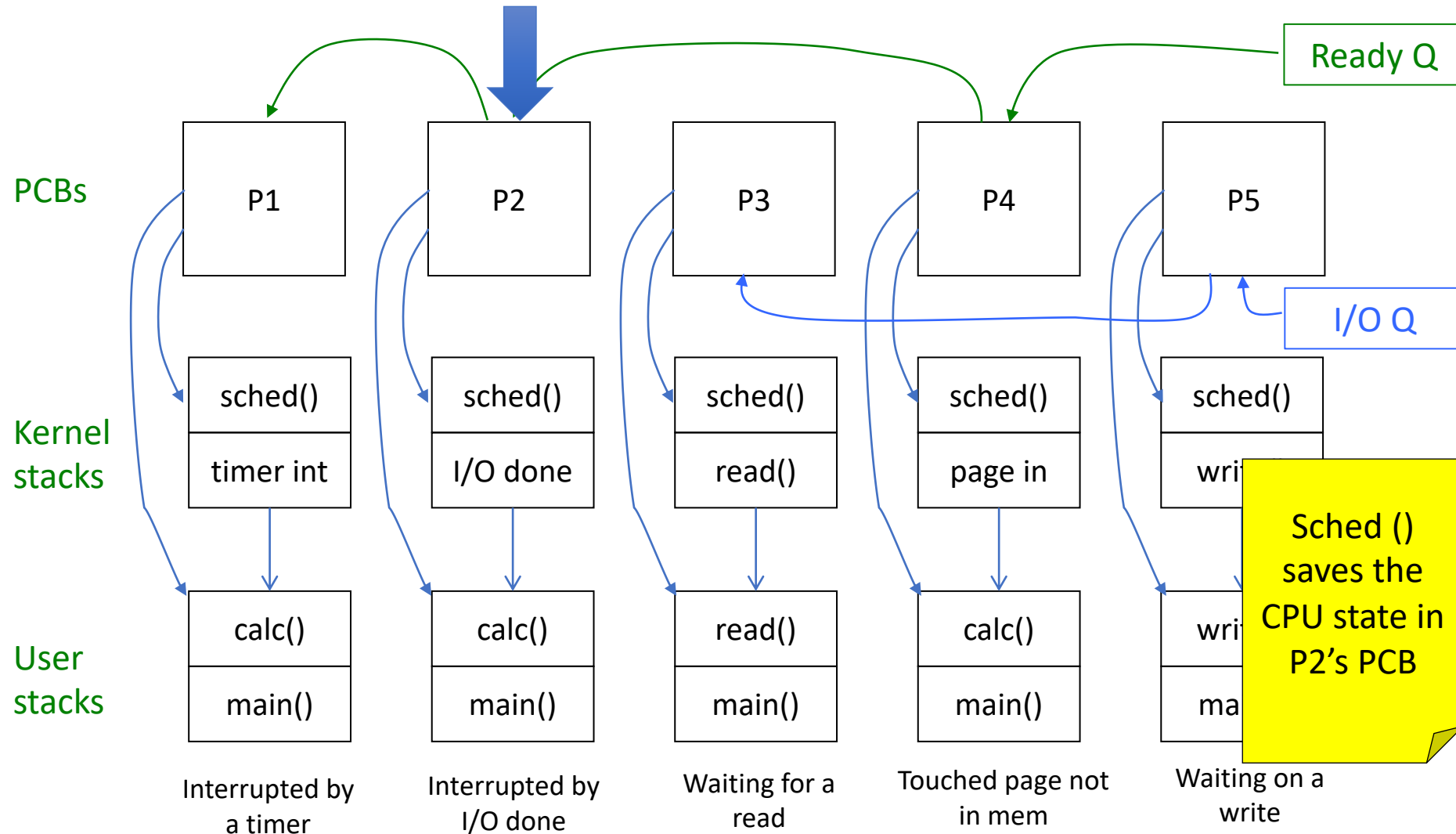
Process example



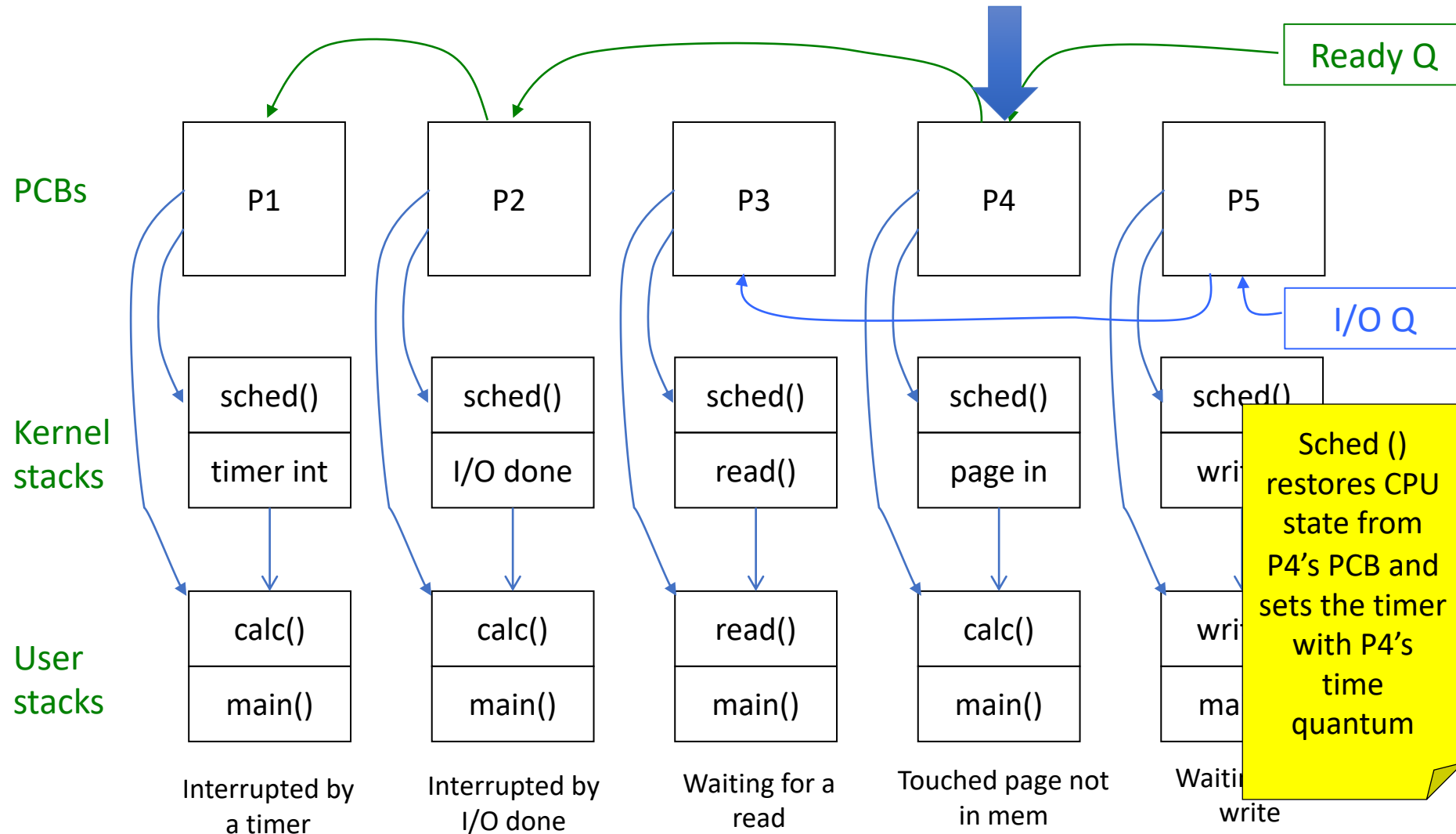
Process example



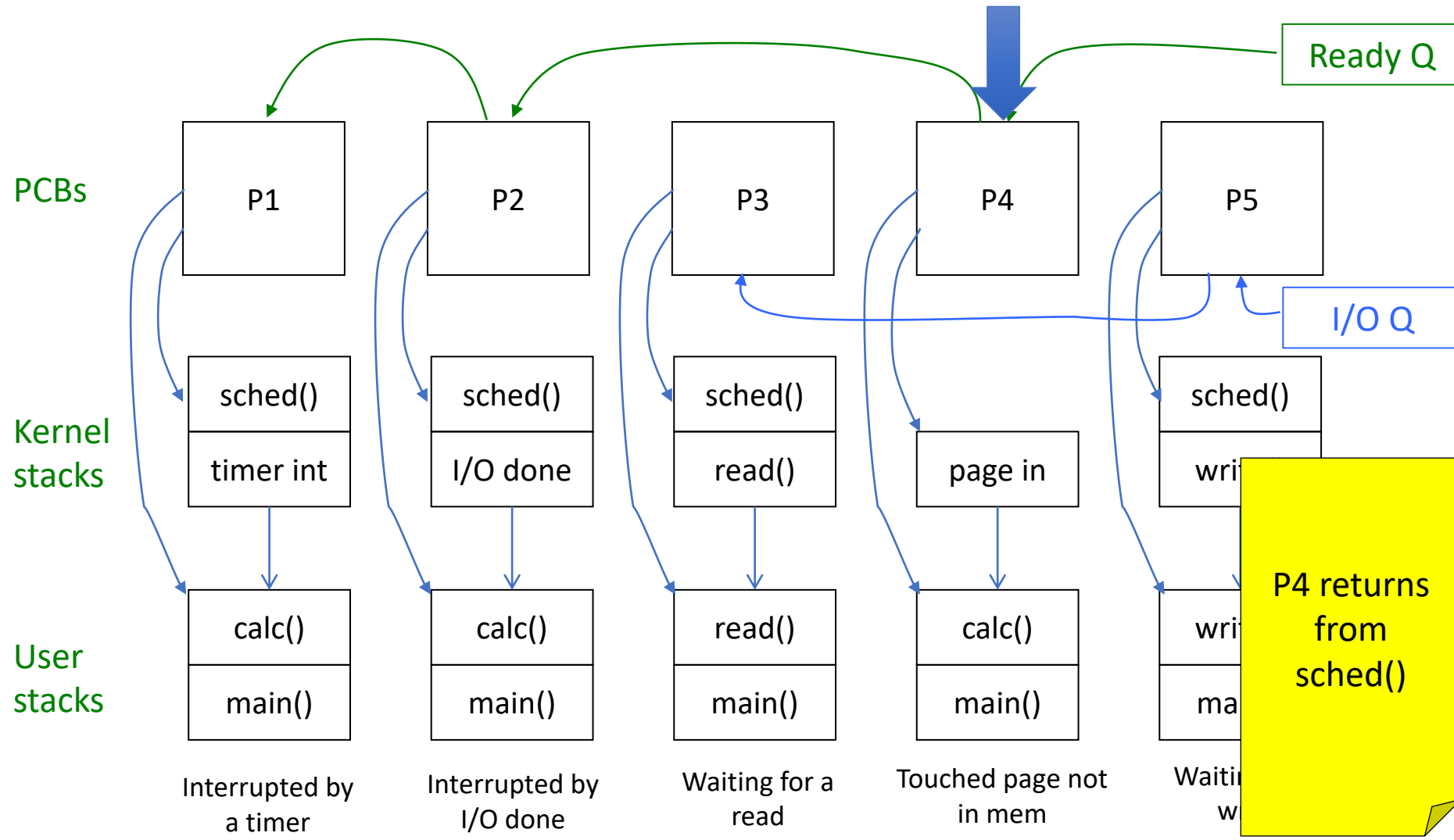
Process example



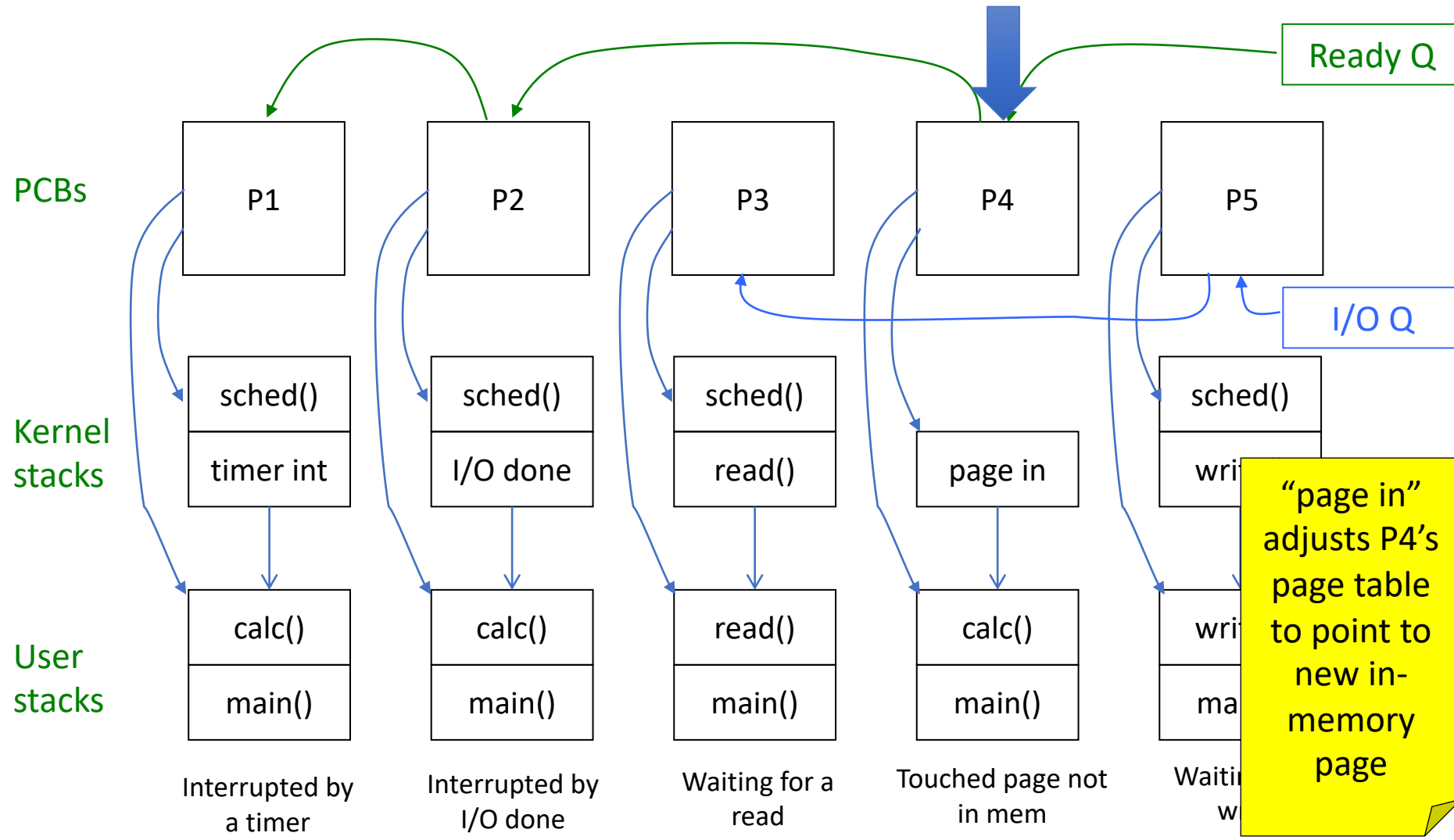
Process example



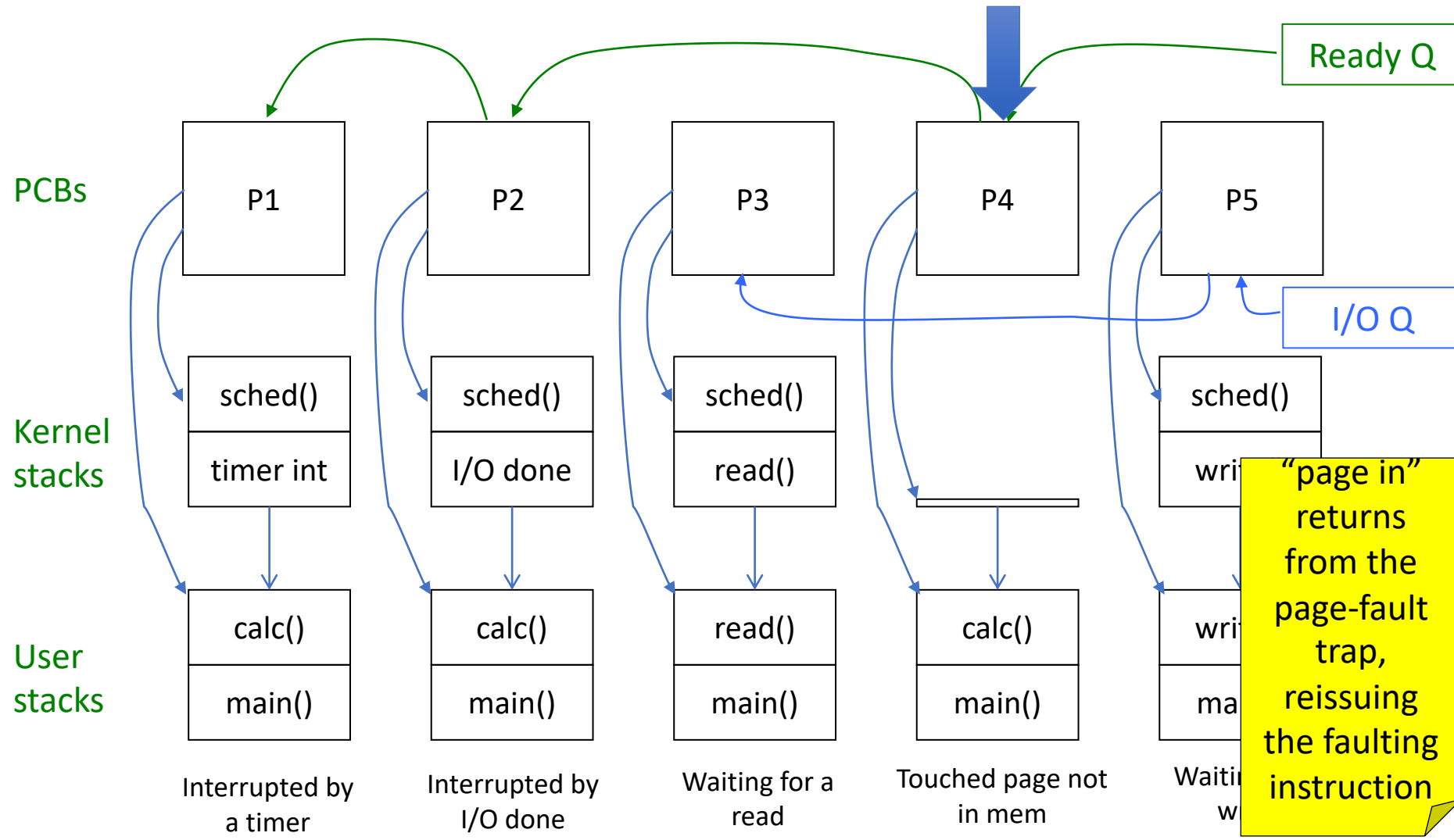
Process example



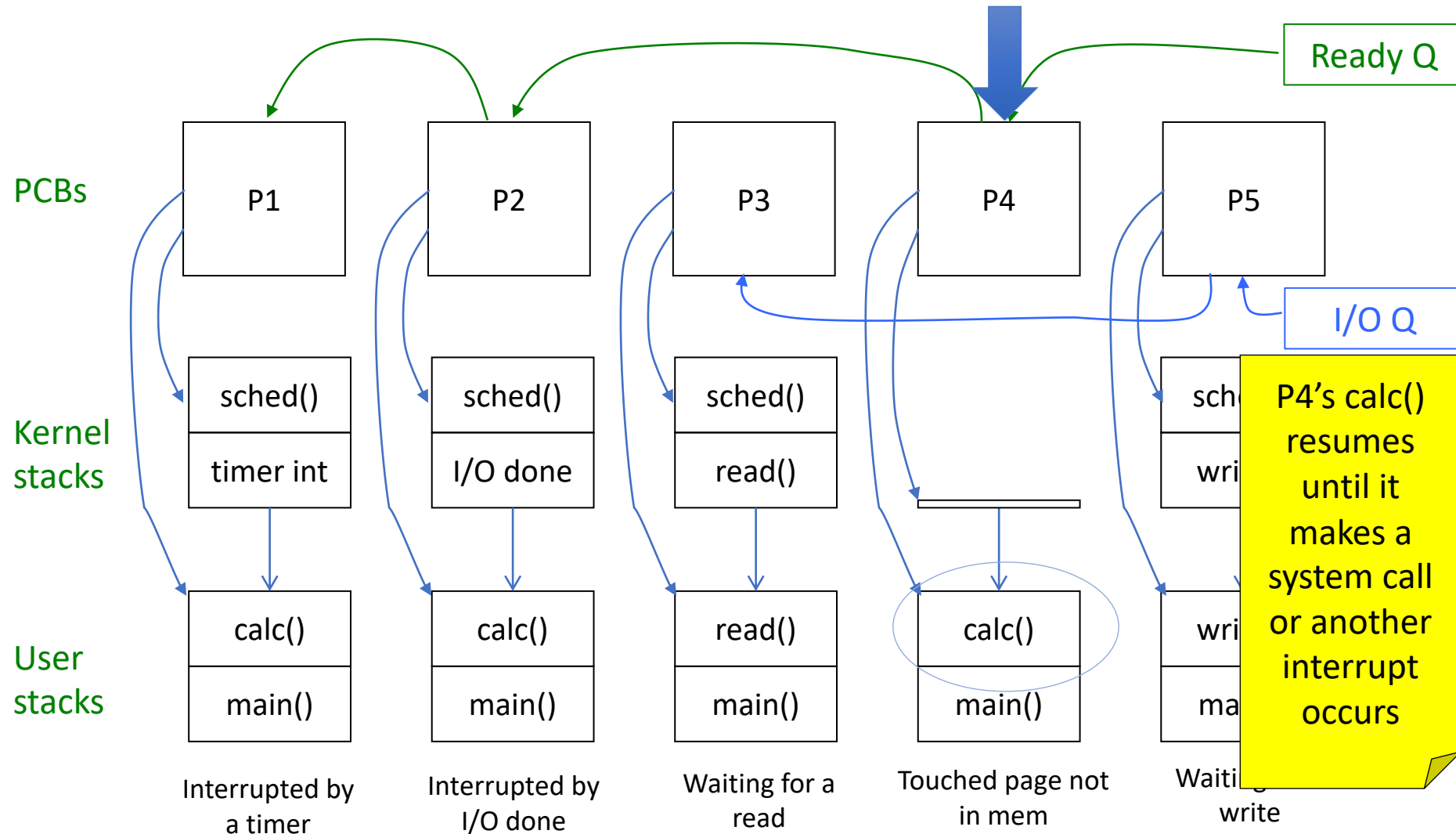
Process example



Process example



Process example





A preemptive scheduler...

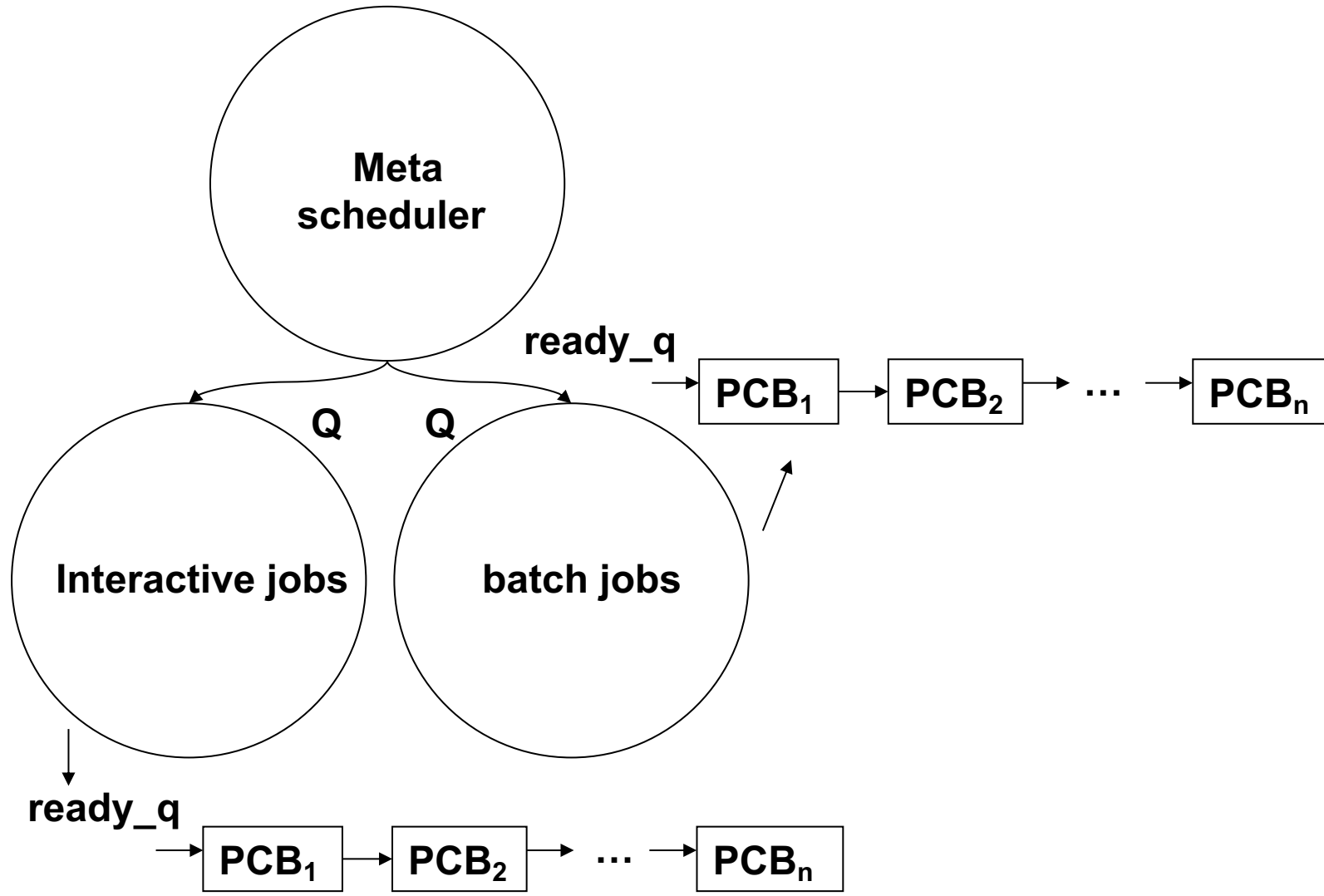
- 0% A. Can only be implemented with a timer interrupt
- 0% B. Can only be implemented with I/O completion interrupt
- 0% C. Can only be implemented with a system call trap
- 0% D. Can be implemented with any type of interrupt



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

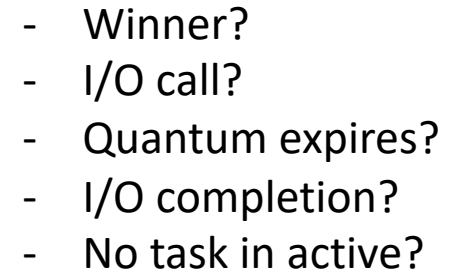
Multi-Level Scheduler



Name	Property	Scheduling criterion	Pros	Cons
FCFS	Intrinsically non-preemptive; could accommodate preemption at time of I/O completion events	Arrival time (intrinsic property)	Fair; no starvation;	high variance in response time; convoy effect
SJF	Intrinsically non-preemptive; could accommodate preemption at time of new job arrival and/or I/O completion events	Expected execution time of jobs (intrinsic property)	Preference for short jobs; provably optimal for response time; low variance in response times	Potential for starvation; bias against long running computations
Priority	Could be either non-preemptive or preemptive	Priority assigned to jobs (extrinsic property)	Highly flexible since priority is not an intrinsic property, its assignment to jobs could be chosen commensurate with the needs of the scheduling environment	Potential for starvation
SRTF	Similar to SJF but uses preemption	Expected remaining execution time of jobs	Similar to SJF	Similar to SJF
Round robin	Preemptive allowing equal share of the processor for all jobs	Time quantum	Equal opportunity for all jobs;	Overhead for context switching among jobs

Linux – a case study

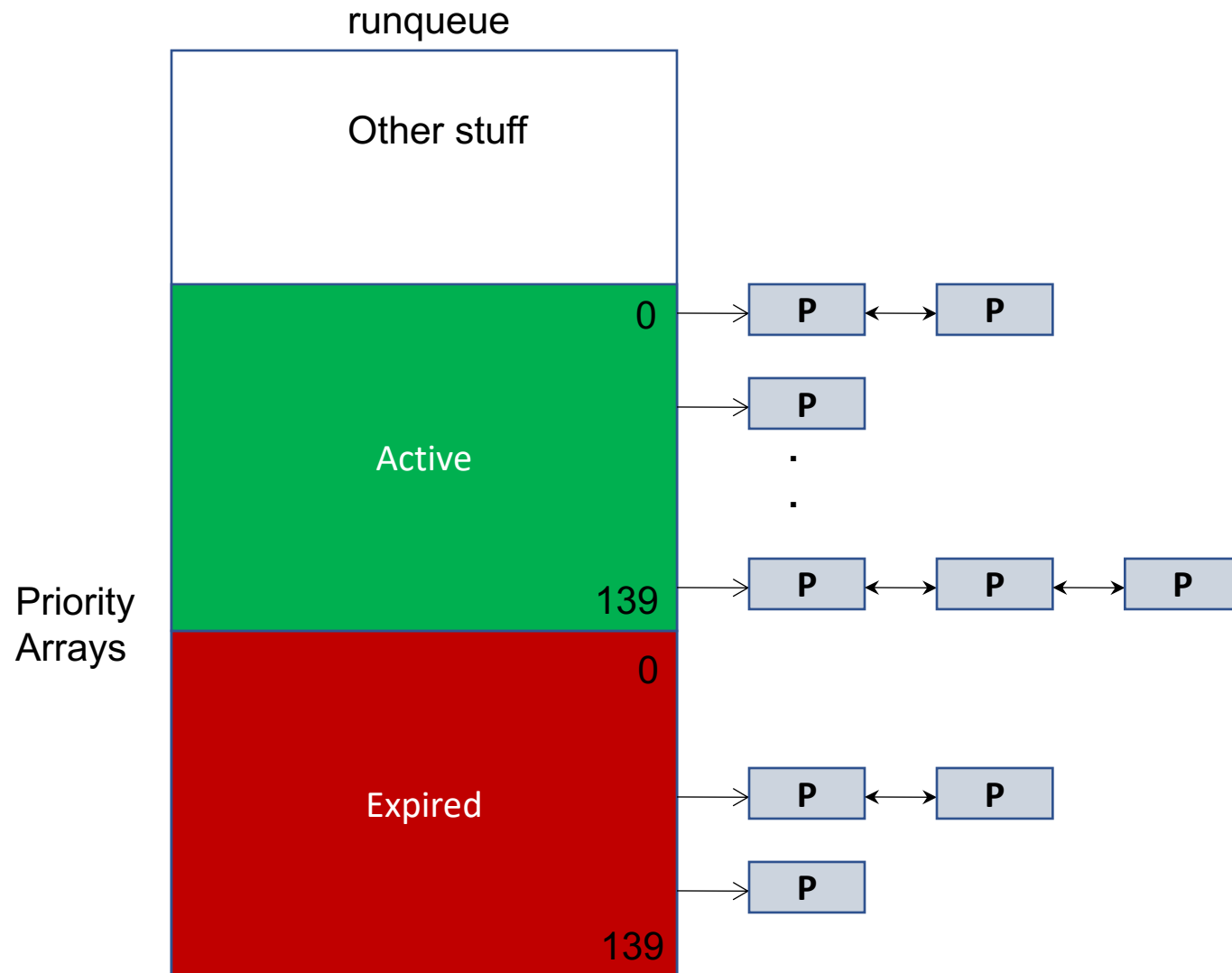
- Markets
 - Desktop (interactive) and server
- Goals
 - Efficiency, interactivity, real-time, no starvation
- Three classes of tasks
 - Real-time FCFS, real-time RR, timeshared
 - 140 priority levels
 - 0-99 for real-time; remaining for timeshared
 - Carrot and stick approach
 - Starvation threshold



- Winner?
- I/O call?
- Quantum expires?
- I/O completion?
- No task in active?

Linux scheduling algorithm

- Winner is the first task in the highest priority list in the active array
- If the task blocks (due to I/O) put it aside and pick the next highest one to run
- If the time quantum runs out (doesn't apply to FCFS tasks) for the current task, place it in the expired array
- I/O completion, place the relevant task in the active array at the right priority level, adjusting its remaining time quantum
- When the active array is empty, flip the active and expired array pointers and continue with the scheduling algorithm (i.e. the expired array becomes the active array and vice versa).



It always takes a constant time to pick the winner, no matter how many processes are running.

FCFS example

Example :

Consider a non-preemptive FCFS process scheduler. There are three processes in the scheduling queue and the arrival order is P1, P2, and P3. The arrival order is always respected when picking the next process to run on the processor. Scheduling starts at time $t = 0$, with the following CPU and I/O burst times:

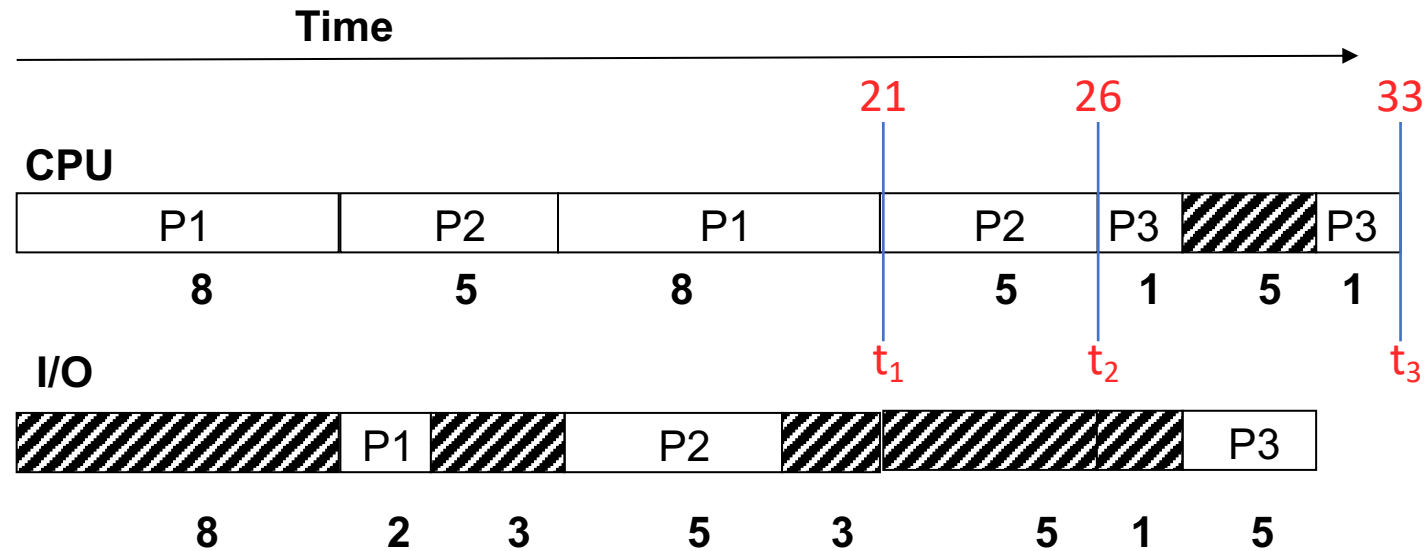
	CPU Burst Time	I/O Burst Time
P1	8	2
P2	5	5
P3	1	5

Each process terminates after completing the following sequence of three actions:

CPU burst | I/O Burst | CPU Burst

- Show the CPU and I/O timelines that result with FCFS scheduling from $t = 0$ until all three processes complete.
- What is the response time for each process?
- What is the waiting time for each process?

FCFS solution sketch



$$w_1 = t_1 - e_1$$

$$e_1 = 8 + 2 + 8, t_1 = 21$$

$$w_1 = 21 - 18 = 3$$

$$w_2 = t_2 - e_2$$

$$e_2 = 5 + 5 + 5, t_2 = 26$$

$$w_2 = 26 - 15 = 11$$

$$w_3 = t_3 - e_3$$

$$e_3 = 1 + 5 + 1, t_3 = 33$$

$$w_3 = 33 - 7 = 26$$

FCFS example 2

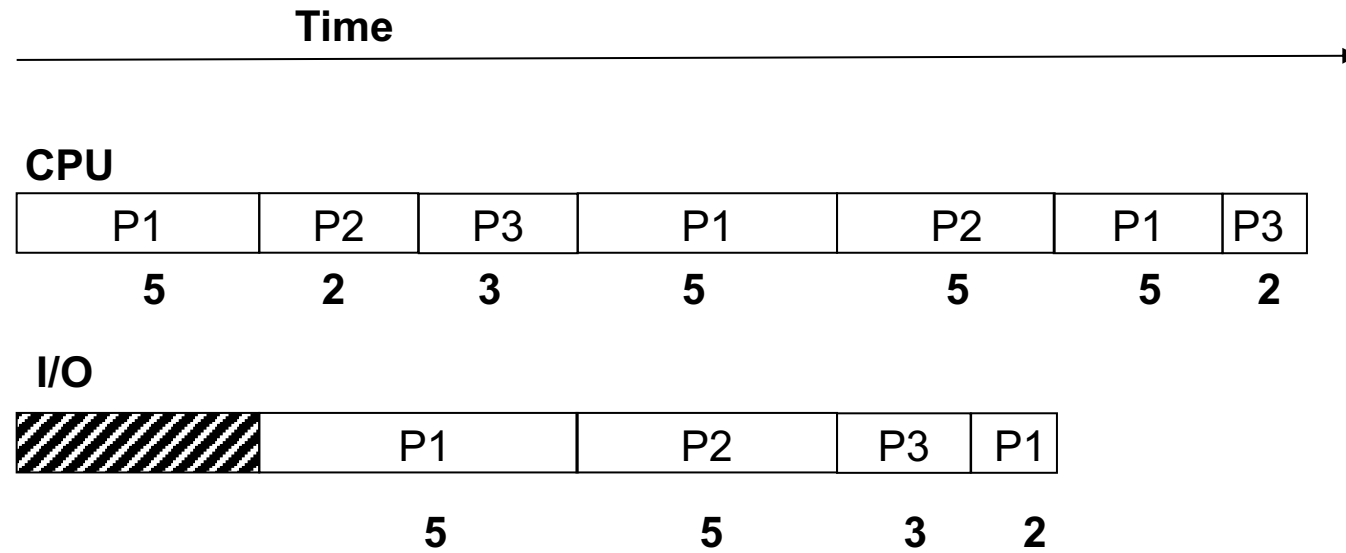
Consider a non-preemptive FCFS process scheduler. There are three processes in the scheduling queue and assume that all three of them are ready to run. As the scheduling discipline suggests, the scheduler always respects the arrival time in selecting a winner. Assume that P1, P2, and P3 arrive in that order into the system. Scheduling starts at time $t = 0$.

The CPU and I/O burst patterns of the three processes are as shown below:

	CPU	I/O	CPU	I/O	CPU
P1	5	5	5	3	5
P2	2	5	5		
P3	3	2	2		

Show the CPU and I/O timelines that result with FCFS scheduling from $t = 0$ until all three processes complete.

Solution sketch



SJF example

Consider a non-preemptive Shortest Job First (SJF) process scheduler. There are three processes in the scheduling queue and assume that all three of them are ready to run. As the scheduling discipline suggests, always the shortest job that is ready to run is given priority. Scheduling starts at time $t = 0$. The CPU and I/O burst patterns of the three processes are as shown below:

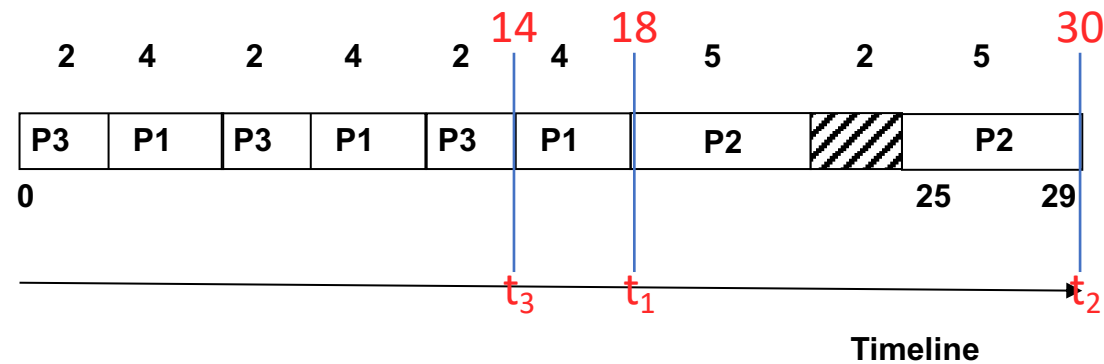
	CPU	I/O	CPU	I/O	CPU
P1	4	2	4	2	4
P2	5	2	5		
P3	2	2	2	2	2

Each process exits the system once its CPU and I/O bursts as shown above are complete.

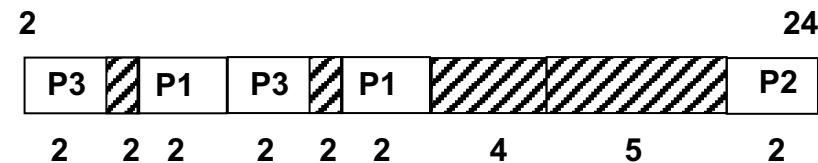
- Show the CPU and I/O timelines that result with SJF scheduling from $t = 0$ until all three processes exit the system.
- What is the waiting time for each process?
- What is the average throughput of the system?

SFJ example solution sketch

CPU Schedule (SJF)



I/O Schedule



$$w_1 = t_1 - e_1$$

$$e_1 = 4 + 2 + 4 + 2 + 4, t_1 = 18$$

$$w_1 = 18 - 16 = 2$$

$$w_2 = t_2 - e_2$$

$$e_2 = 5 + 2 + 5, t_2 = 30$$

$$w_2 = 30 - 12 = 18$$

$$w_3 = t_3 - e_3$$

$$e_3 = 2 + 2 + 2 + 2 + 2, t_3 = 14$$

$$w_3 = 14 - 10 = 4$$