Georgia Tech Computer Science

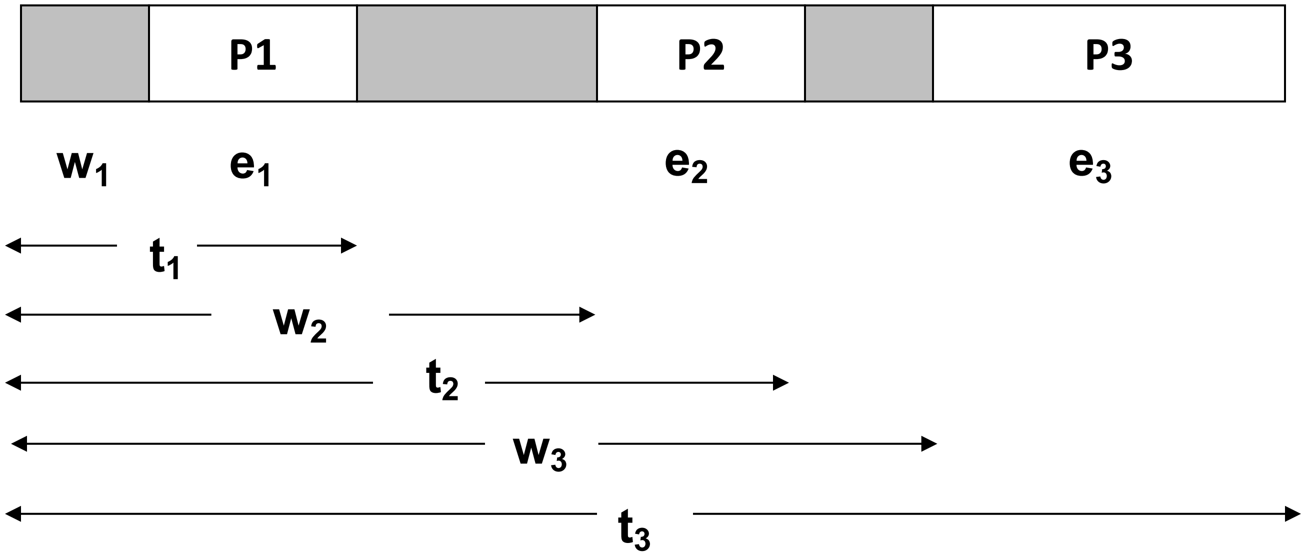
CS2200
Systems and Networks
Spring 2022

Lecture 14: Processor Scheduling Algorithms

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

Metrics



w_i, e_i, and t_i, are respectively the wait time, execution time, and the elapsed time (turnaround time) for a job j_i

System Centric

- **Throughput?** — 3 / t_3 jobs/sec
- **Avg. Turnaround Time?** — $(t_1+t_2+t_3)/3$ sec
- **Avg. Wait Time?** — $(w_1+w_2+w_3)/3$ sec

User Centric →

- **Response time?** — $R_{P1}=t_1, R_{P2}=t_2, R_{P3}=t_3$

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})	$(t_1+t_2+\ldots+t_n)/n$	Secs	**System-centric metric quantifying the average time it takes for a job to complete**
Avg. Waiting time (w_{avg})	$(w_1+w_2+\ldots+w_n)/n$	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.]**

Non-preemptive scheduling algorithms

- FCFS
- SJF

Intrinsic property

- Priority → Extrinsic property
- Resource requirements:

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

Burst times

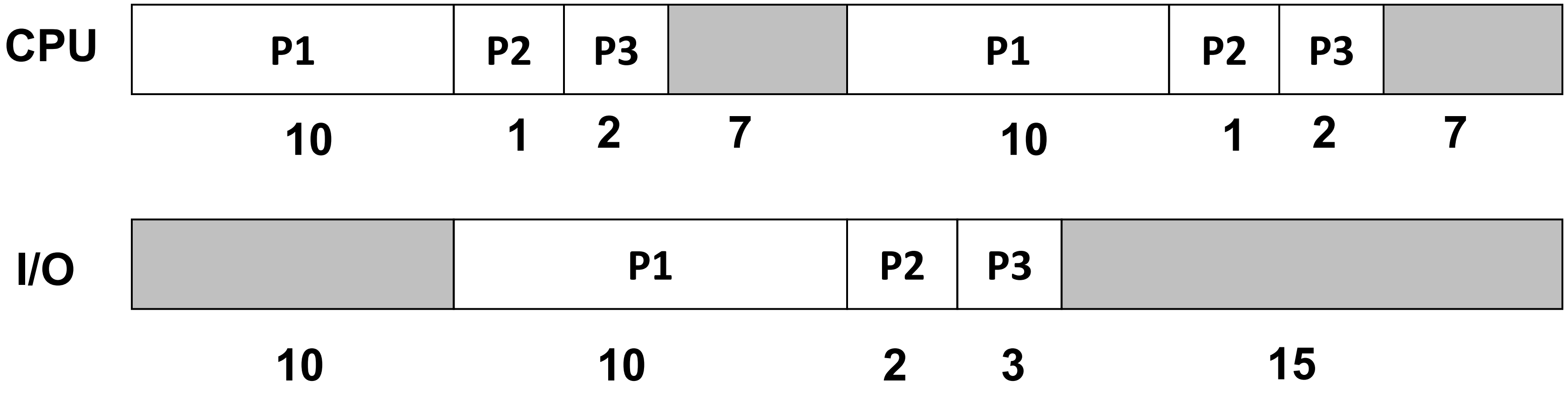
- Arrival order
  - P1, P2, P3 in order at nearly the same time

Assume each process uses

- CPU burst
- I/O Burst
- CPU Burst
- Done

FCFS

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



CPU	P1	P2	P3		P1	P2	P3	
	10	1	2	7	10	1	2	7

I/O		P1	P2	P3	
	10	10	2	3	15

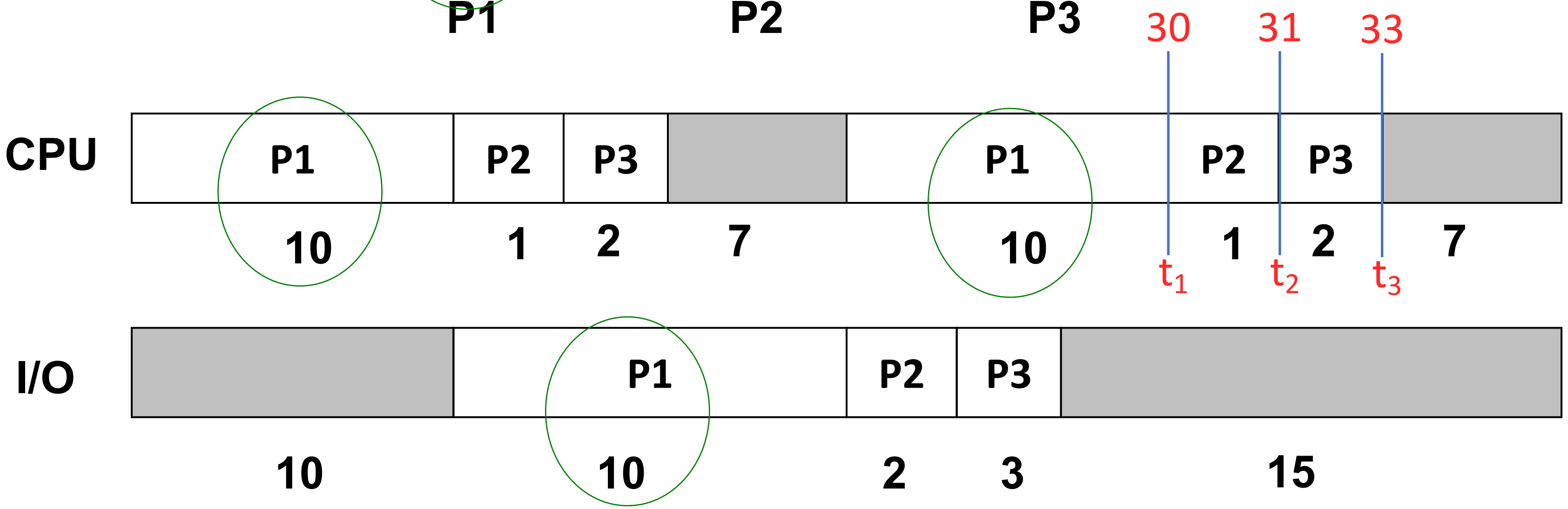
$t_i = w_i + e_i$

What are the waiting times?

FCFS

CPU → I/O → CPU → done is the process behavior

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



$w_{P1} = t_1 - e_1$

$w_{P2} = t_2 - e_2$

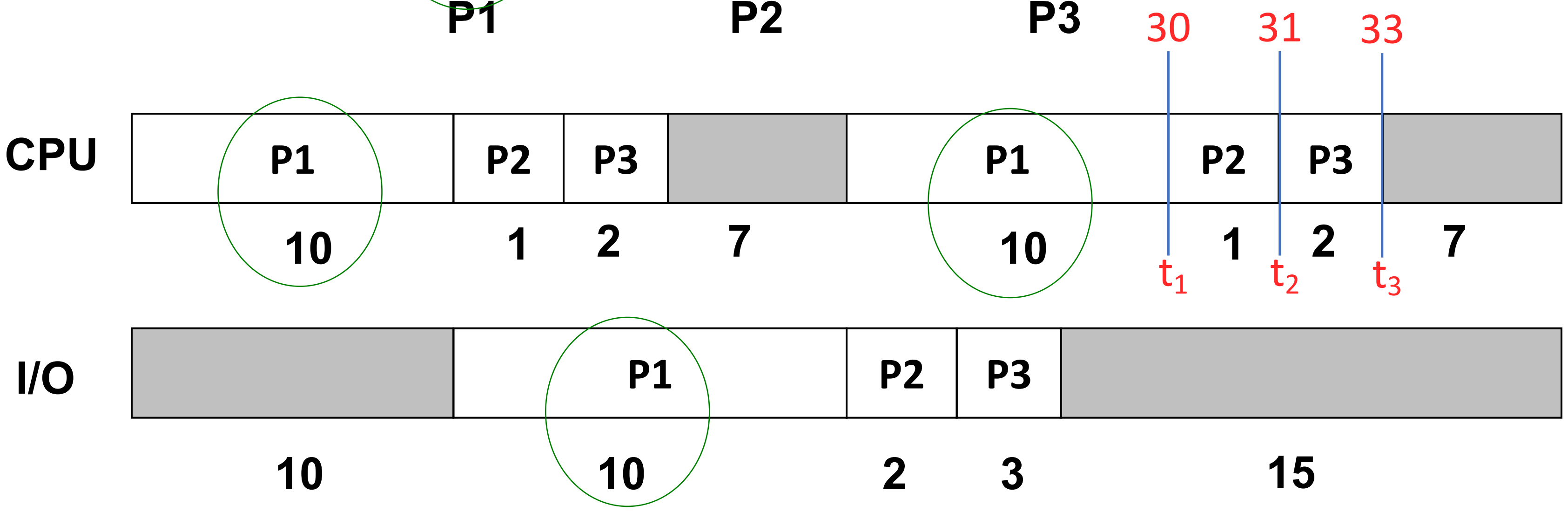
$w_{P3} = t_3 - e_3$

$e_1 = ?$

FCFS

CPU → I/O → CPU → done is the process behavior

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



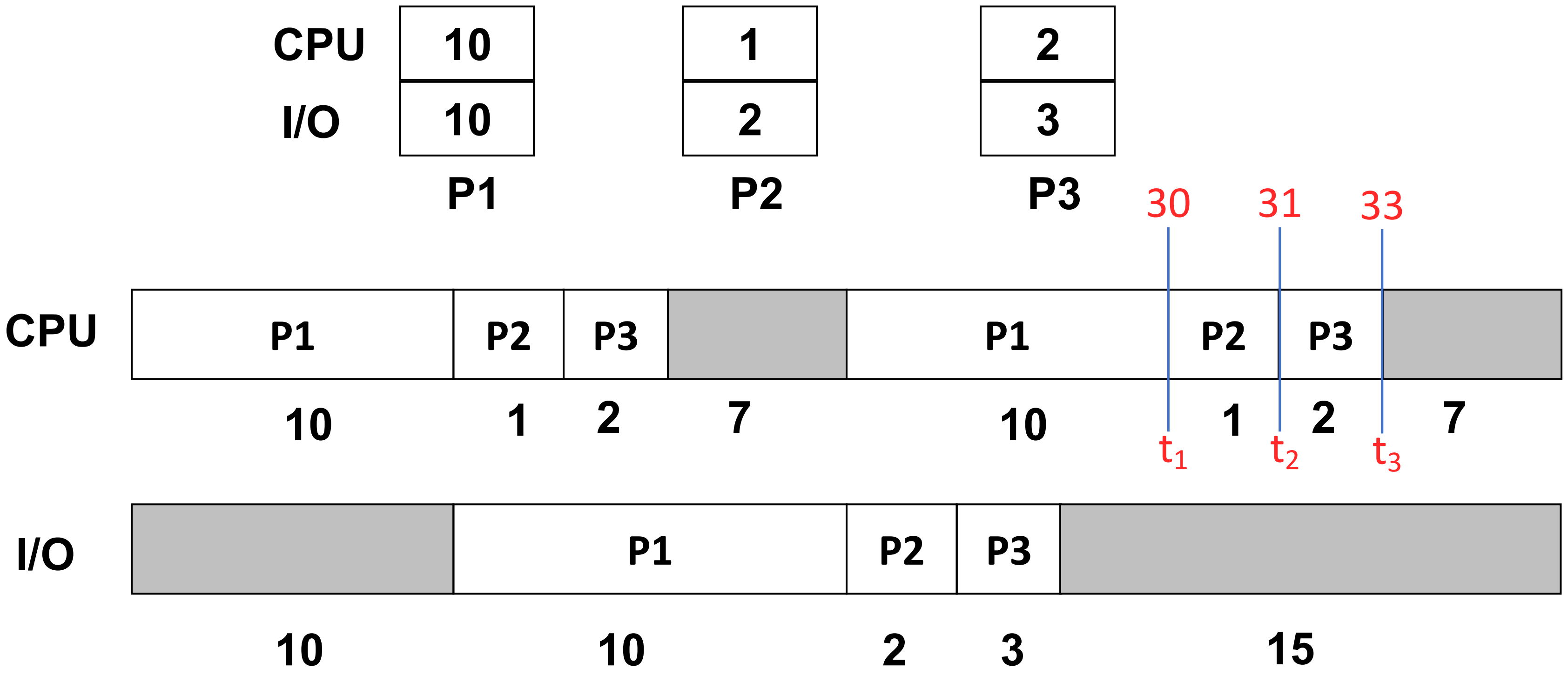
$w_{P1} = t_1 - e_1$ $w_{P2} = t_2 - e_2$ $w_{P3} = t_3 - e_3$

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

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

Individual Activity!

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



$w_{P1} = 0$

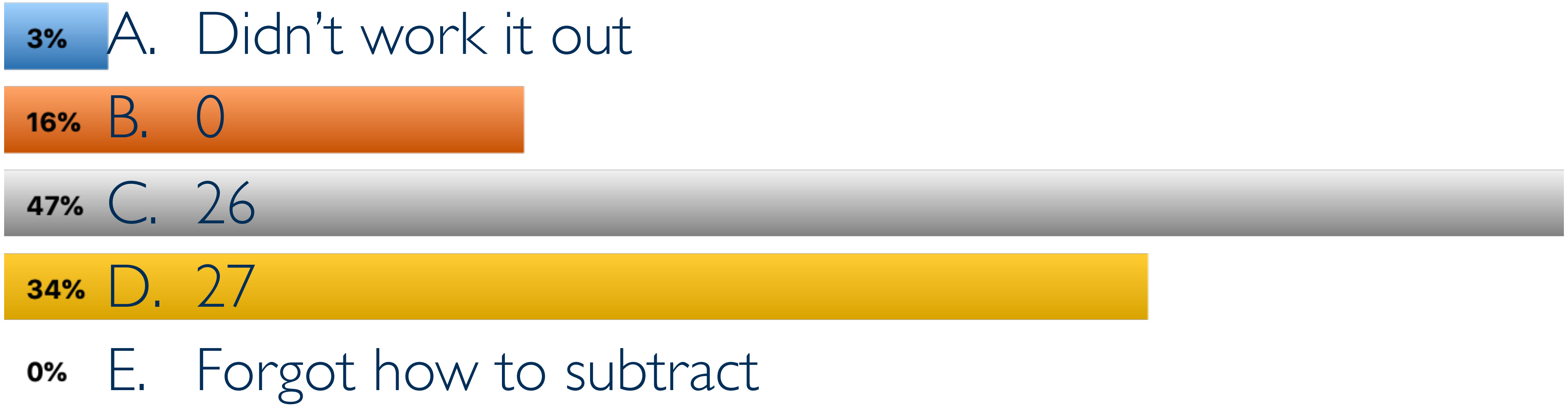
$w_{P2} = t_2 - e_2$

$w_{P3} = t_3 - e_3$

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

Session ID: dag2200

What is the wait time for P2?



3% A. Didn’t work it out

16% B. 0

47% C. 26

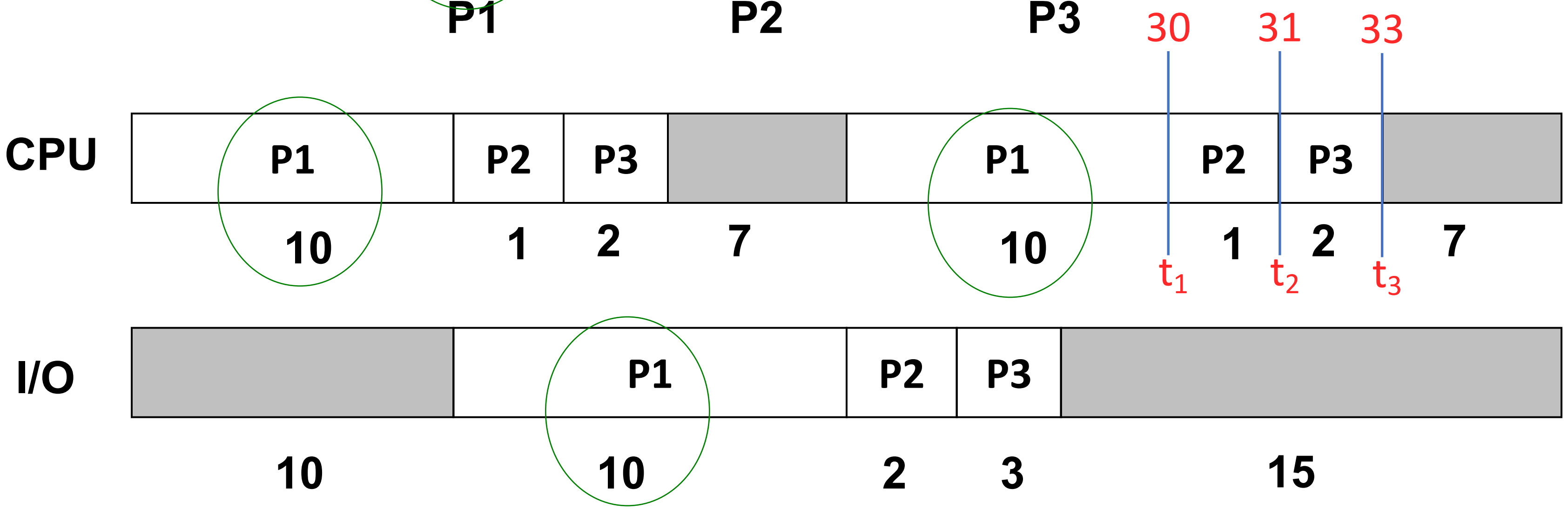
34% D. 27

0% E. Forgot how to subtract

FCFS

CPU → I/O → CPU → done is the process behavior

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



$w_{P1} = t_1 - e_1$

$w_{P2} = t_2 - e_2$

$w_{P3} = t_3 - e_3$

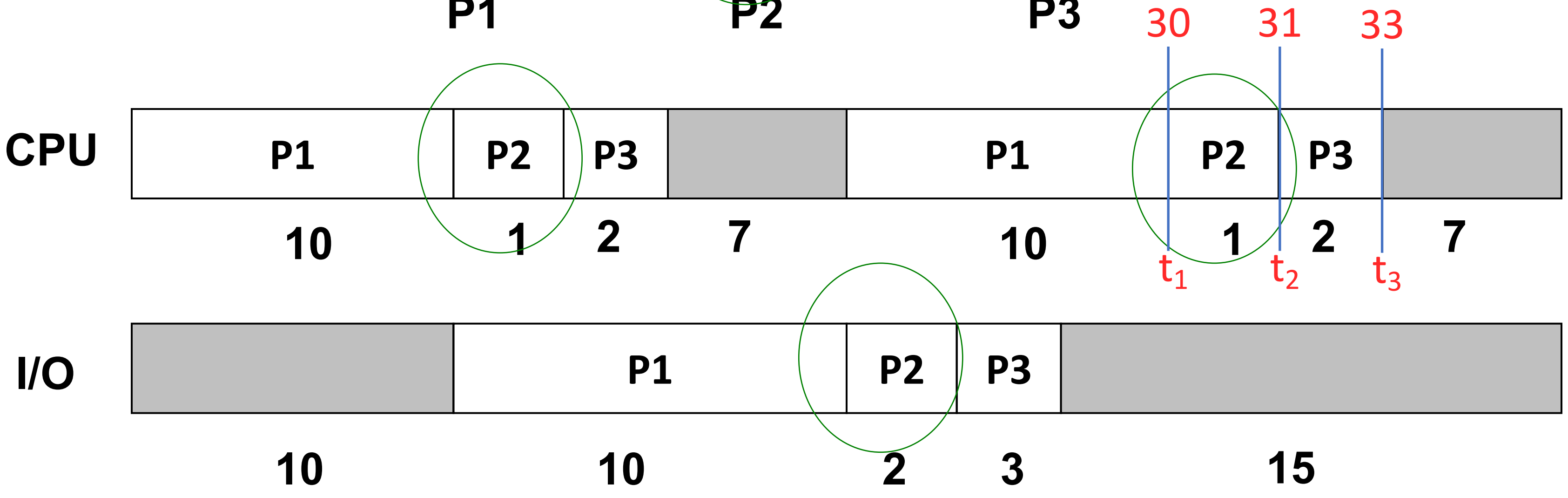
$e_1 = 10+10+10$, $t_1 = 30$

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

FCFS

CPU → I/O → CPU → done is the process behavior

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



$w_{P1} = t_1 - e_1$

$w_{P2} = t_2 - e_2$

$w_{P3} = t_3 - e_3$

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

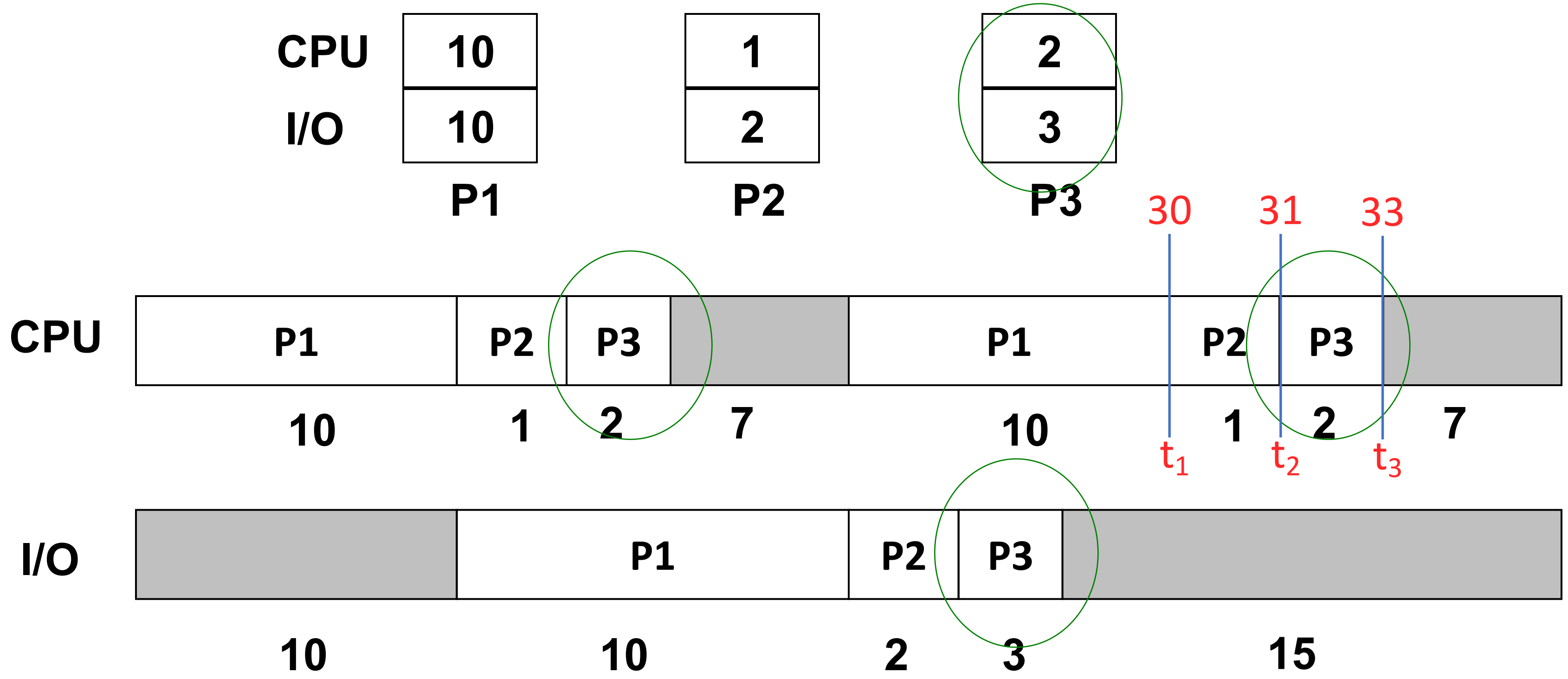
$w_{P1} = 30 - 30 = 0$

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

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

FCFS

CPU → I/O → CPU → done is the process behavior



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

$w_{P1} = t_1 - e_1$

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

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

$w_{P2} = t_2 - e_2$

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

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

$w_{P3} = t_3 - e_3$

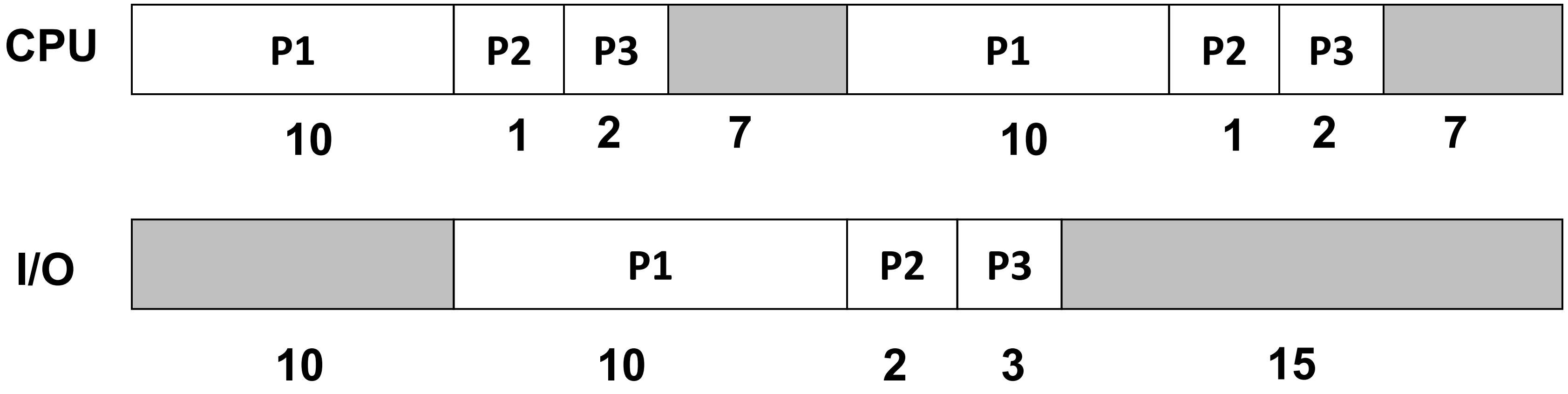
$e_3=2+3+2$, ${}_{t3=}33$

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

FCFS

$w_{P1} = 0$ $w_{P2} = 27$ $w_{P3} = 26$

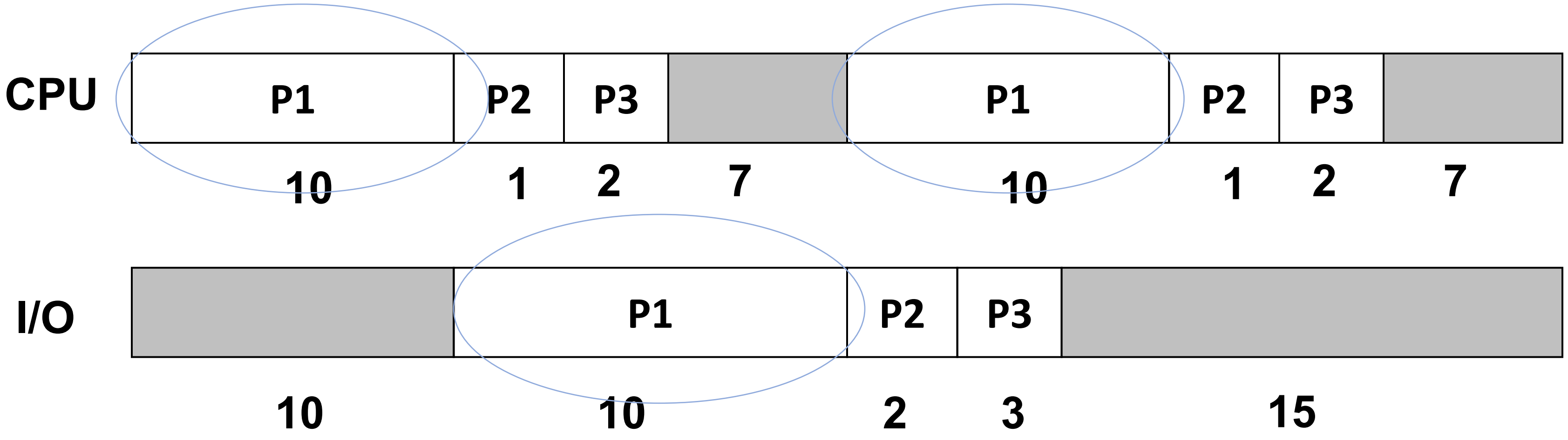
	P1	P2	P3
CPU	10	1	2
I/O	10	2	3



- High average waiting times

FCFS

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



- High average waiting times – in this case 53 / 3
- High average turnaround times – in this case 94 / 3
- Convoy effect

Non-preemptive scheduling algorithms

- ~~FCFS~~
- SJF
- Priority
- Resource requirements:

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

Burst times

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

Assume each process uses

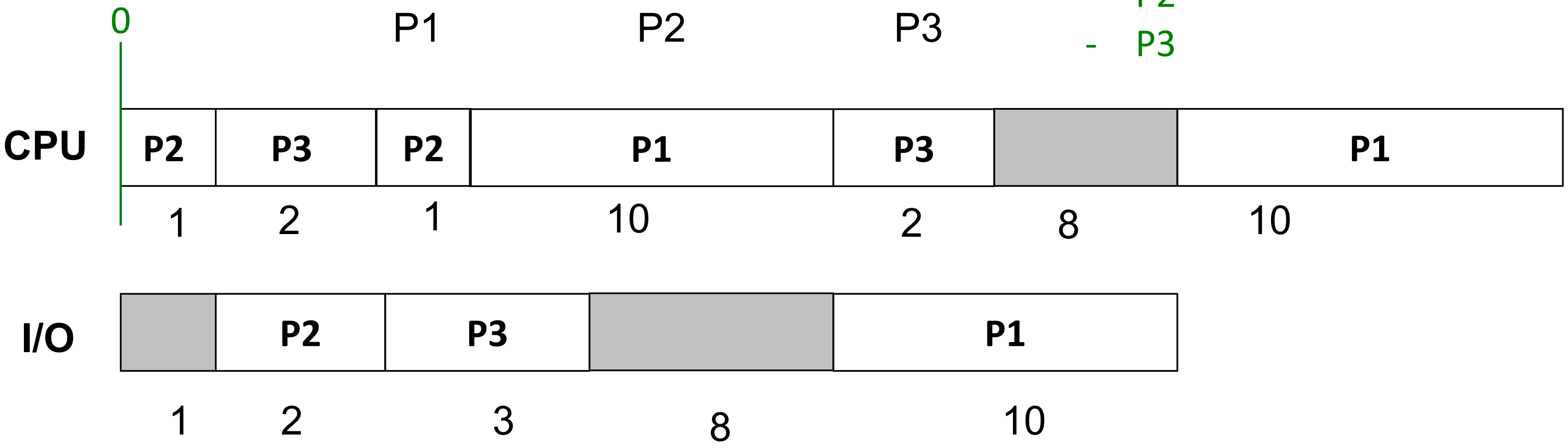
- CPU burst
- I/O Burst
- CPU Burst
- Done

SJF

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

Processes ready:

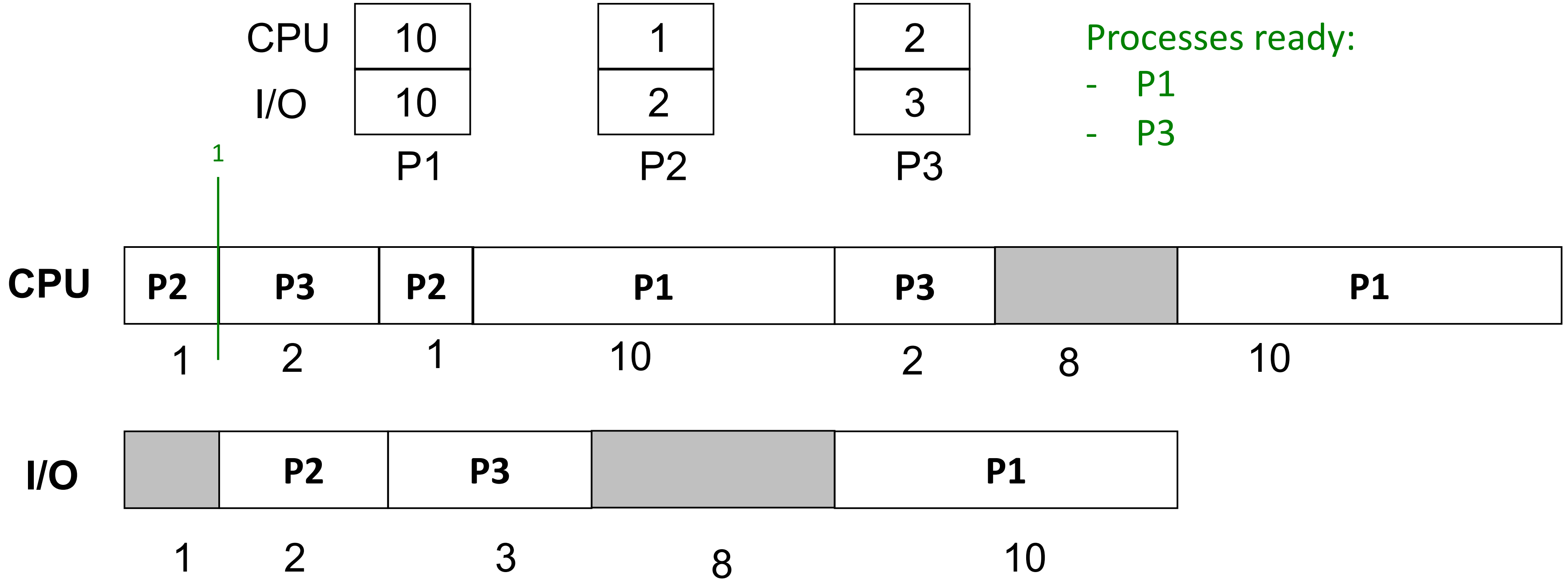
- P1
- P2
- P3



Straightforward:

P2 is shortest, so it gets run first

SJF



Processes ready:

- P1
- P3

P1 and P3 are both "ready"

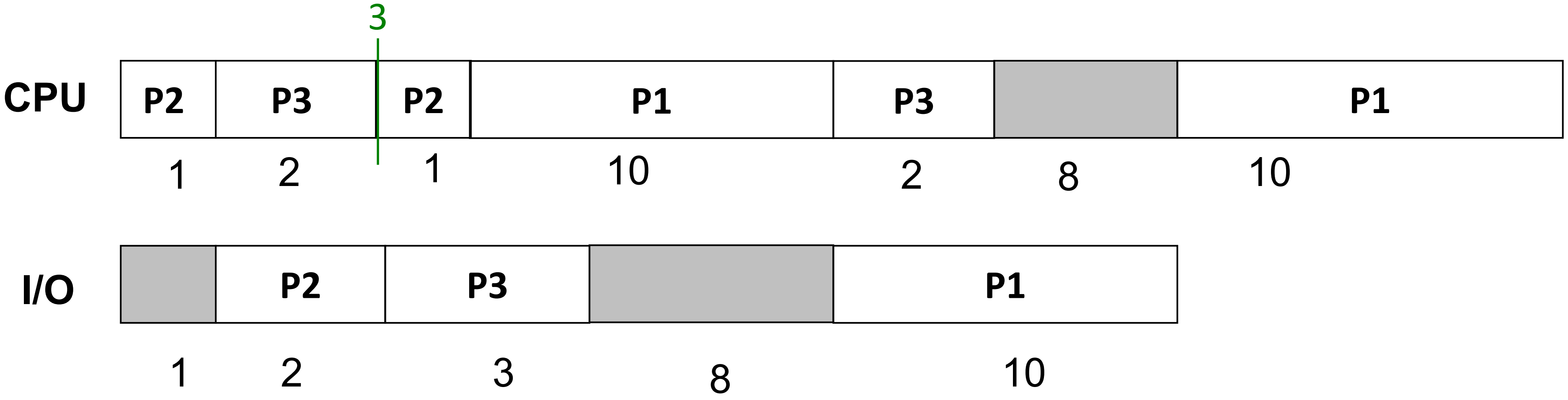
➔ the scheduler picks P3 because it's shortest

SJF

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

Processes ready:

- P1
- P2



P1 and P2 are both “ready”

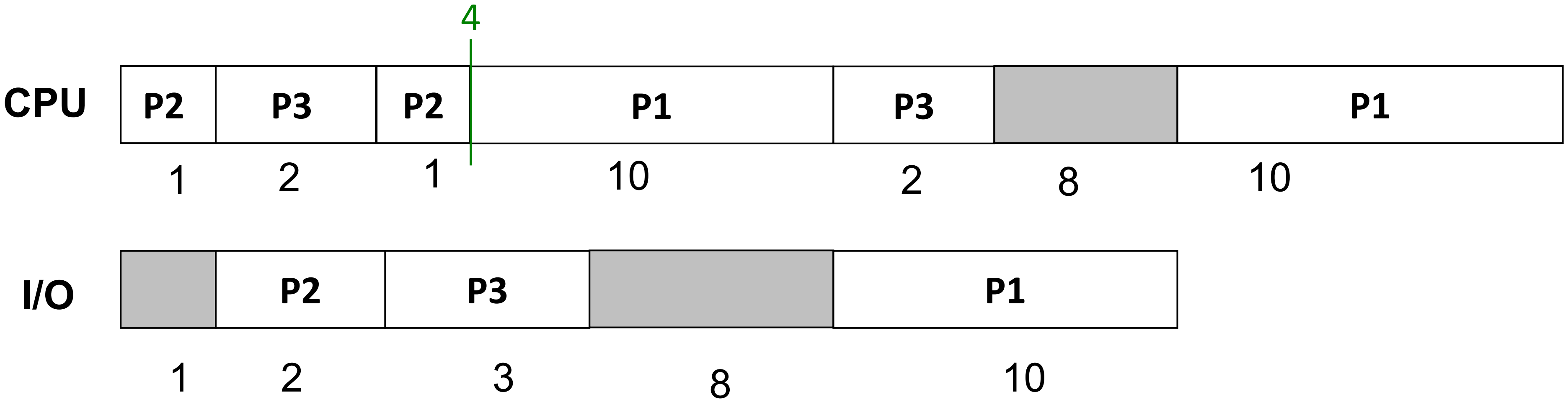
➔ the scheduler picks P2 because it’s shortest

SJF

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

Processes ready:

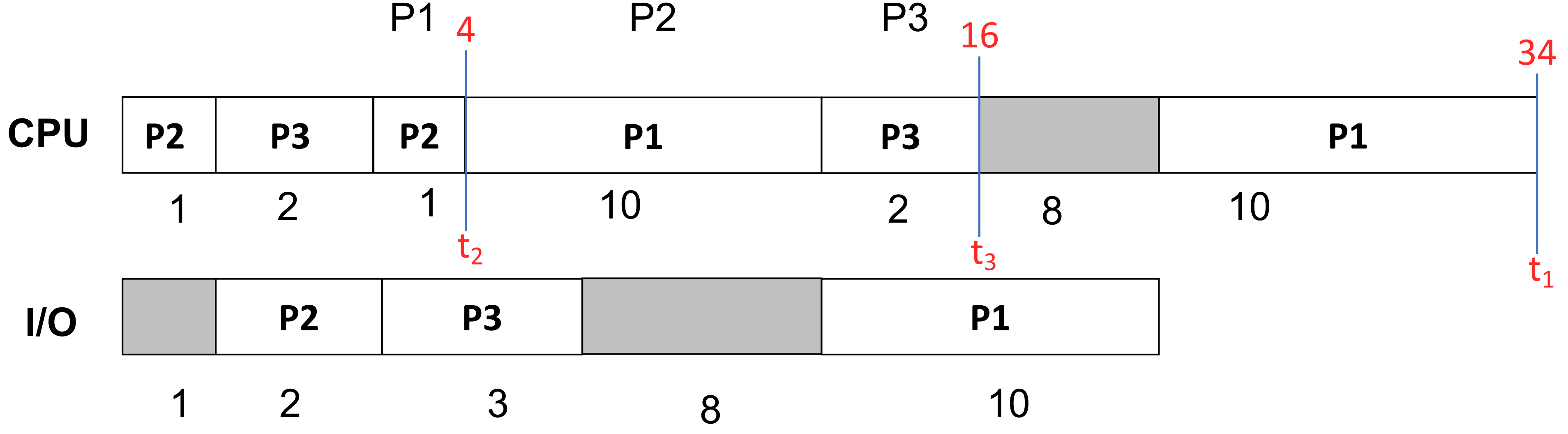
- P1



Only P1 is ready to run

SJF

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

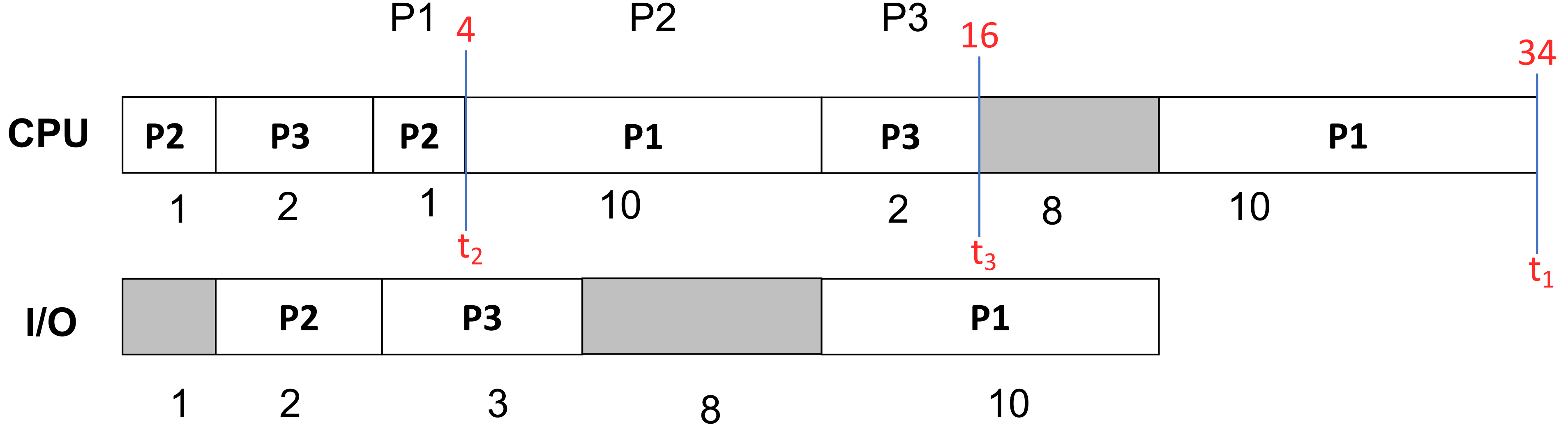


And so on until we get to the end

So what are the waiting times?

SJF

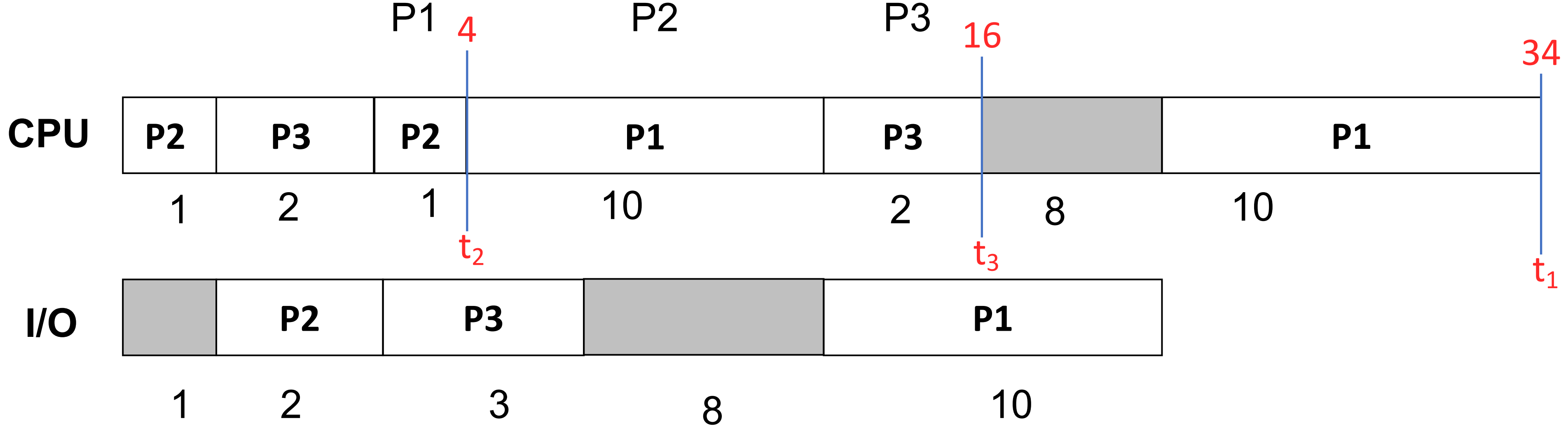
	P1	P2	P3
CPU	10	1	2
I/O	10	2	3



$e_1 = 30, t_1 = 34$ $e_2 = 4, t_2 = 4$ $e_3 = 7, t_3 = 16$

SJF

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



$e_1 = 30, t_1 = 34$

$w_{P1} = t_1 - e_1$

$w_{P1} = 34 - 30$

$w_{P1} = 4$

$e_2 = 4, t_2 = 4$

$w_{P2} = t_2 - e_2$

$w_{P2} = 4 - 4$

$w_{P2} = 0$

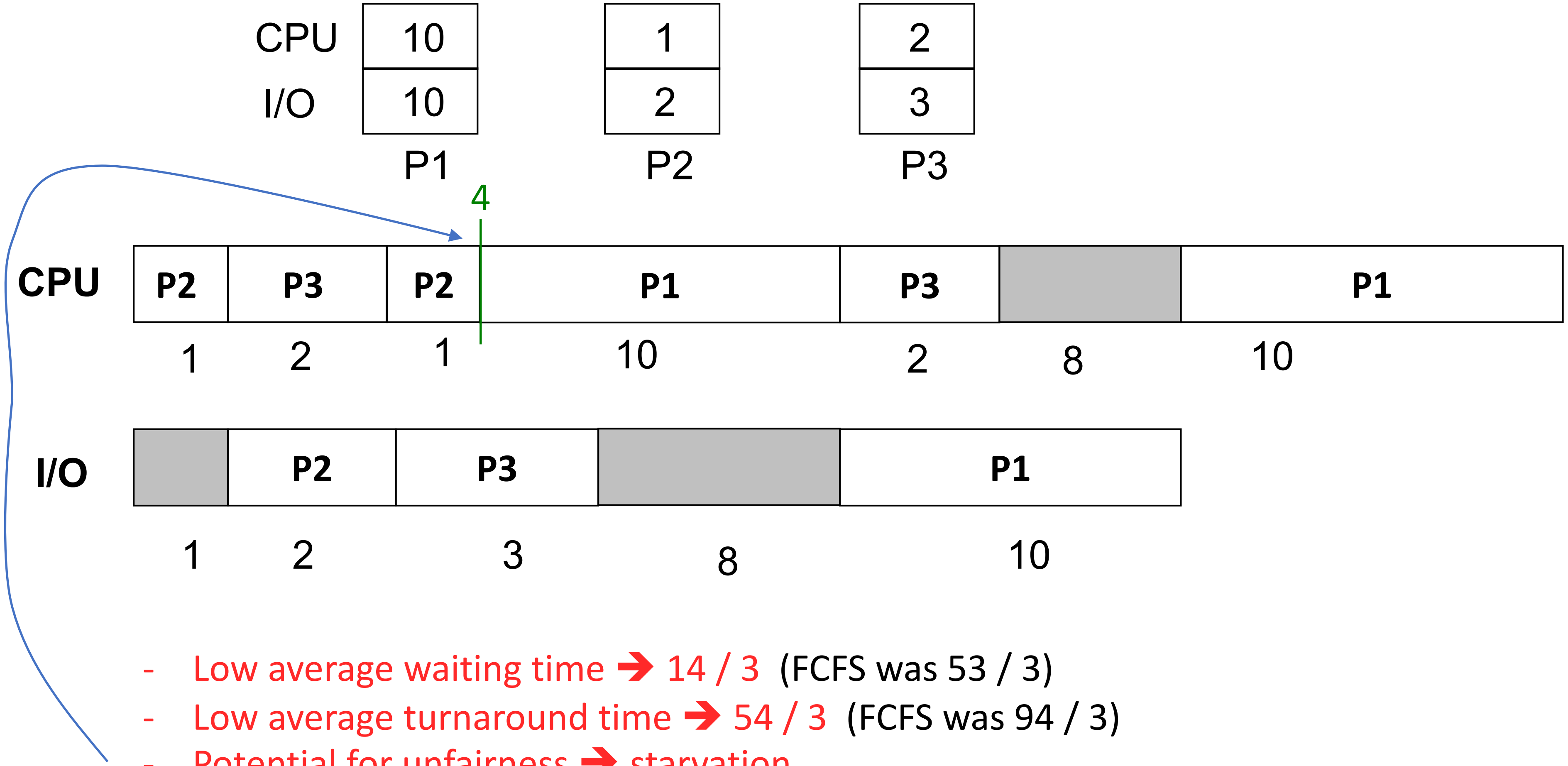
$e_3 = 7, t_3 = 16$

$w_{P3} = t_3 - e_3$

$w_{P3} = 16 - 7$

$w_{P3} = 9$

SJF

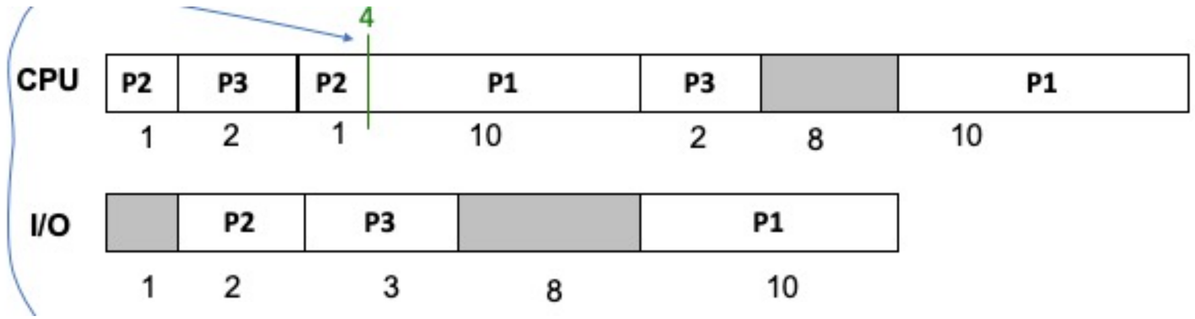


- Low average waiting time ➔ 14 / 3 (FCFS was 53 / 3)
- Low average turnaround time ➔ 54 / 3 (FCFS was 94 / 3)
- Potential for unfairness ➔ starvation

Session ID: dag2200

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

- 5% A. 1/11 processes/unit-time
- 32% B. 3/24 processes/unit-time
- 63% C. 3/34 processes/unit-time

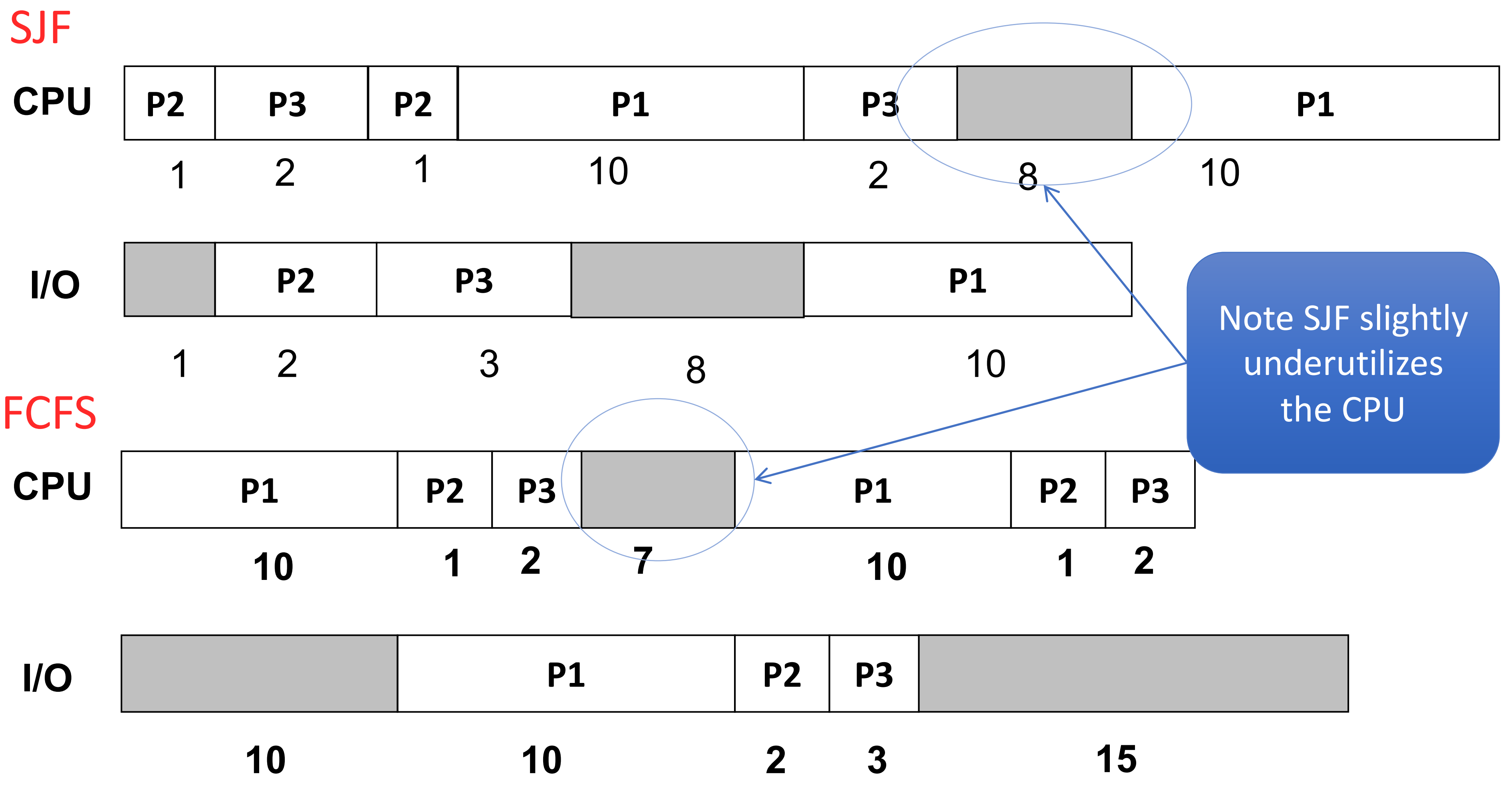


Throughput is simply jobs/time, so

3 jobs
34 seconds of elapsed time

3/34 is the throughput

SJF vs. FCFS

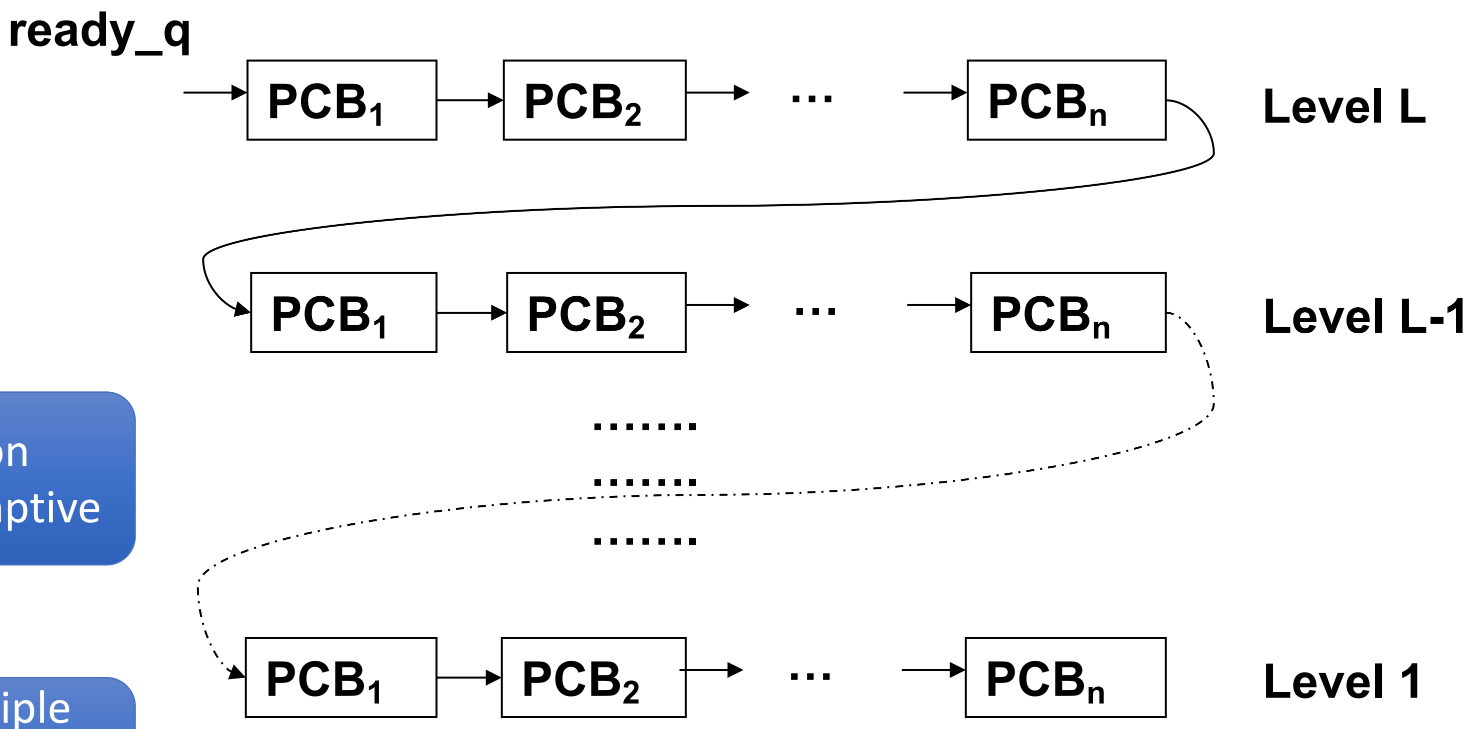


Note SJF slightly underutilizes the CPU

Non-preemptive scheduling algorithms

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

(Extrinsic) Priority scheduler



Non preemptive

Multiple priority levels

Preemptive Scheduling

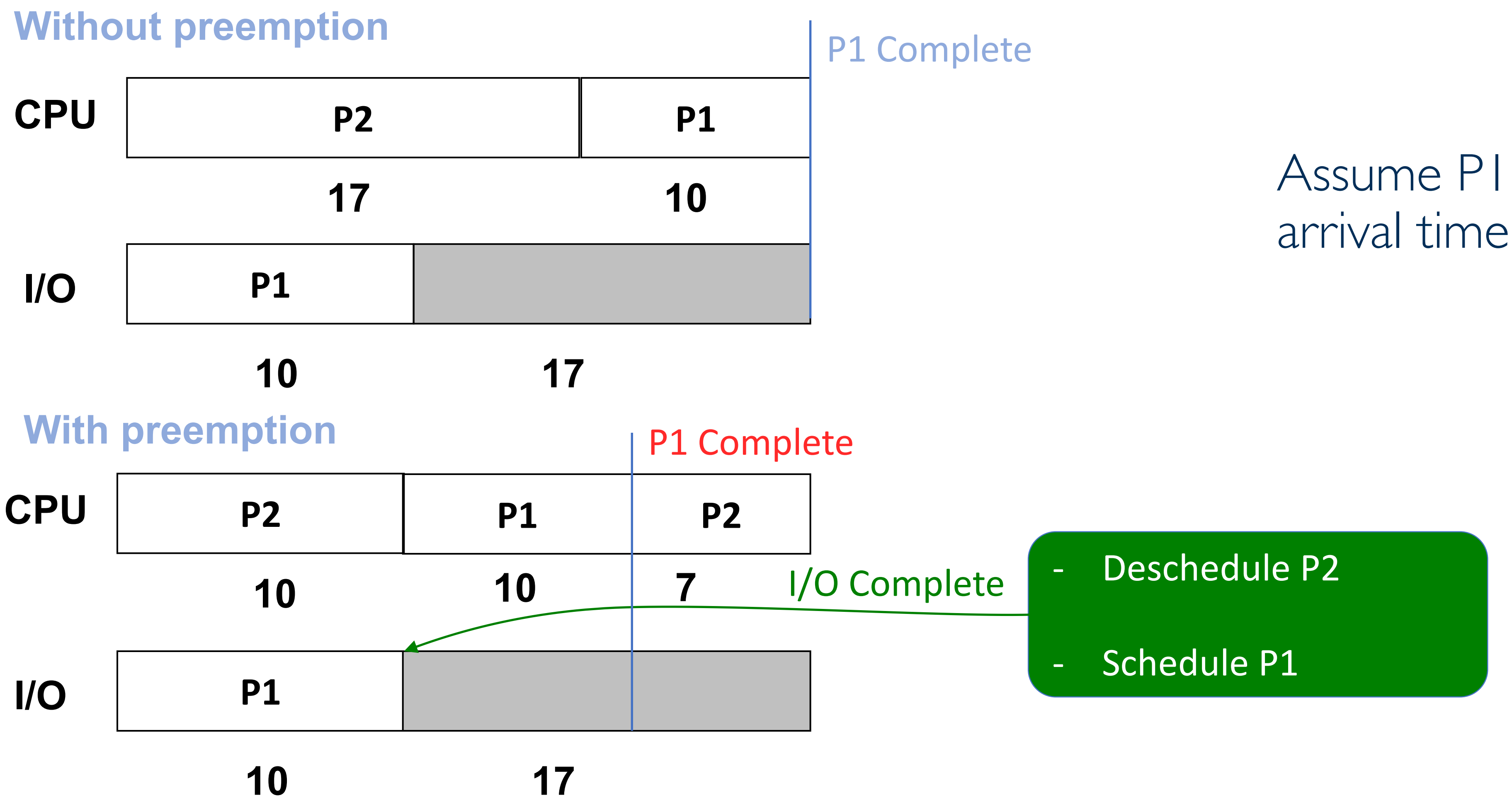
- Yank processor from currently running process at an "opportune moment" to give it to a "higher priority" process
- Questions
  - What are "**opportune moments**"?
  - How can we determine "**higher priority**"?

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

P1 Complete

CPU

P2

P1

17

10

I/O

P1

10

17

With preemption

P1 Complete

CPU

P2

P1

P2

10

10

7

I/O Complete

- Deschedule P2
- Schedule P1

I/O

P1

10

17

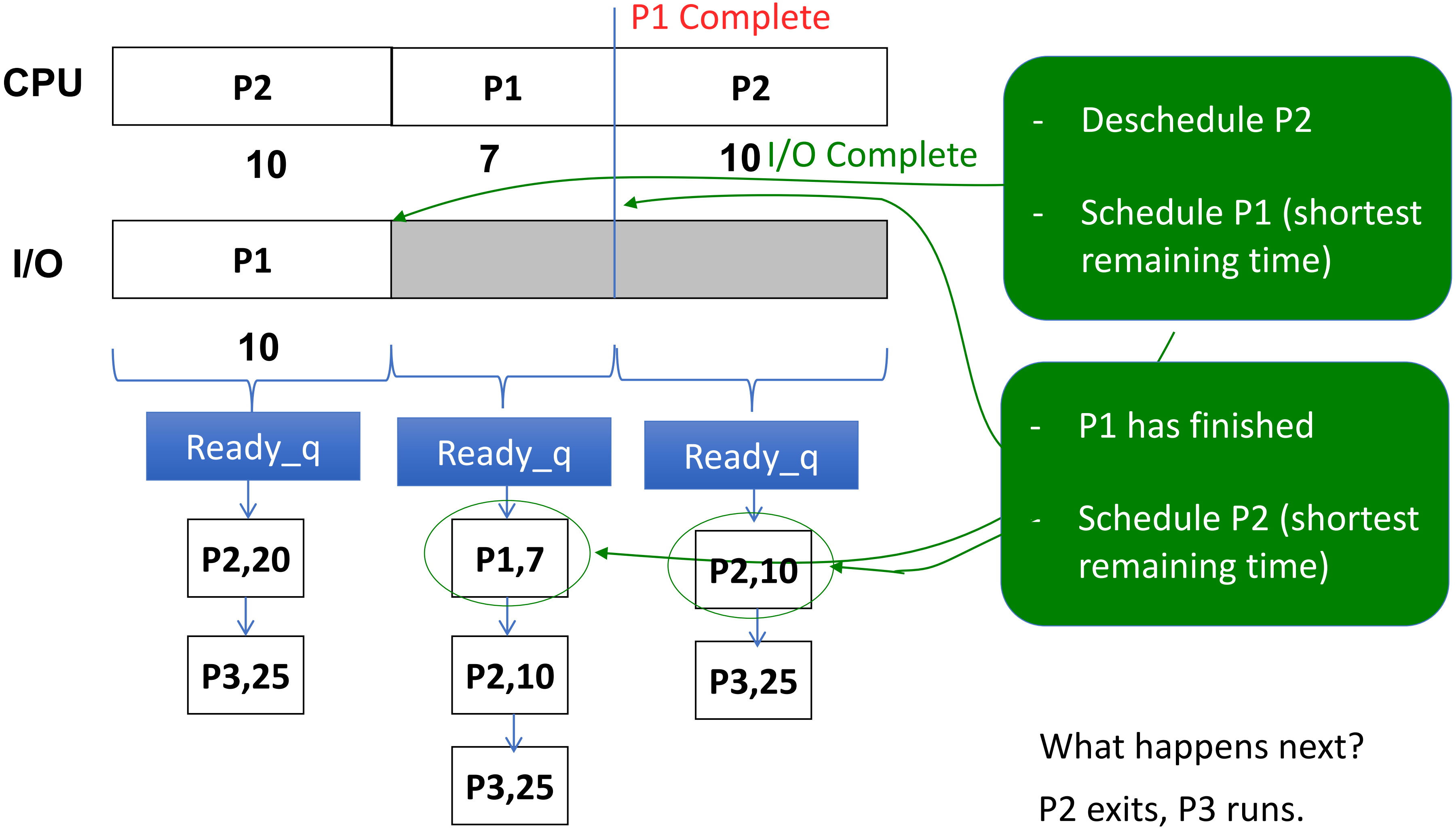
Assume P1 has earlier arrival time than P2

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



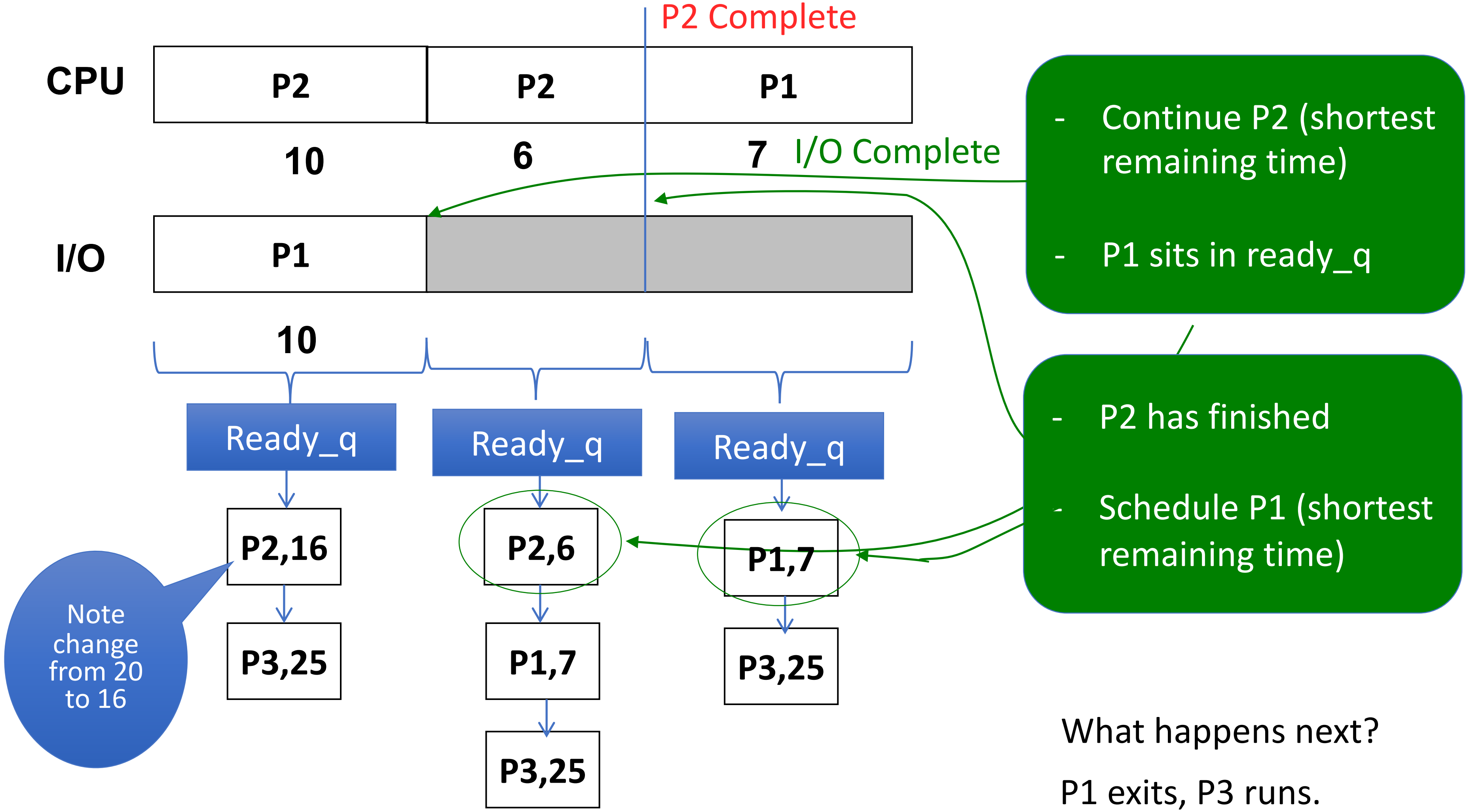
- Deschedule P2
- Schedule P1 (shortest remaining time)

- P1 has finished
- Schedule P2 (shortest remaining time)

What happens next?

P2 exits, P3 runs.

SRTF with preemption



- Continue P2 (shortest remaining time)
- P1 sits in ready_q

- P2 has finished
- Schedule P1 (shortest remaining time)

What happens next?

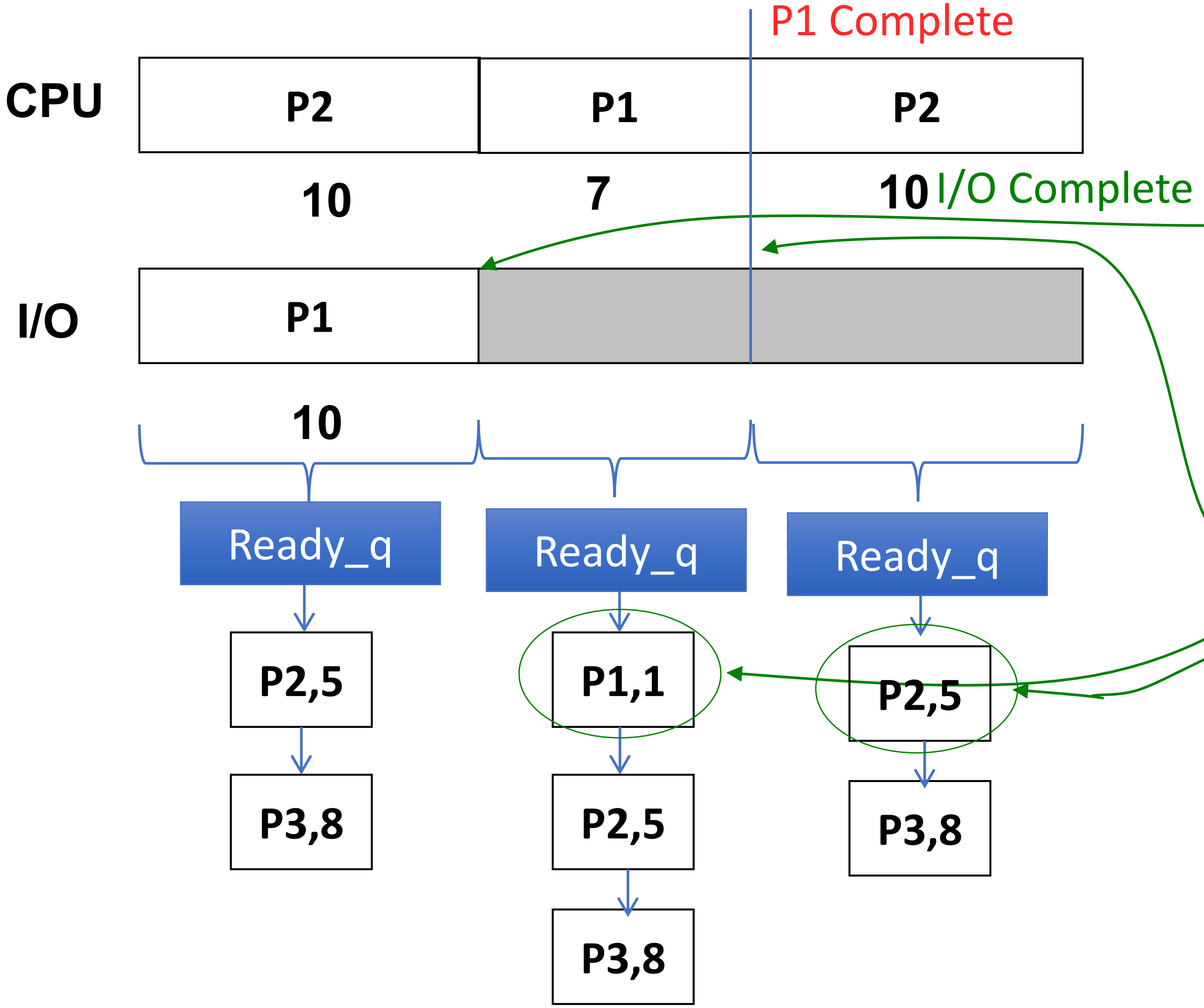
P1 exits, P3 runs.

Preemptive Schedulers

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

Reevaluate priority on I/O completion

Priority with preemption



- Deschedule P2
- Schedule P1 (highest priority)

- P1 has finished
- Schedule P2 (highest priority)

What happens next?

P2 exits, P3 runs.

Session ID: dag2200

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

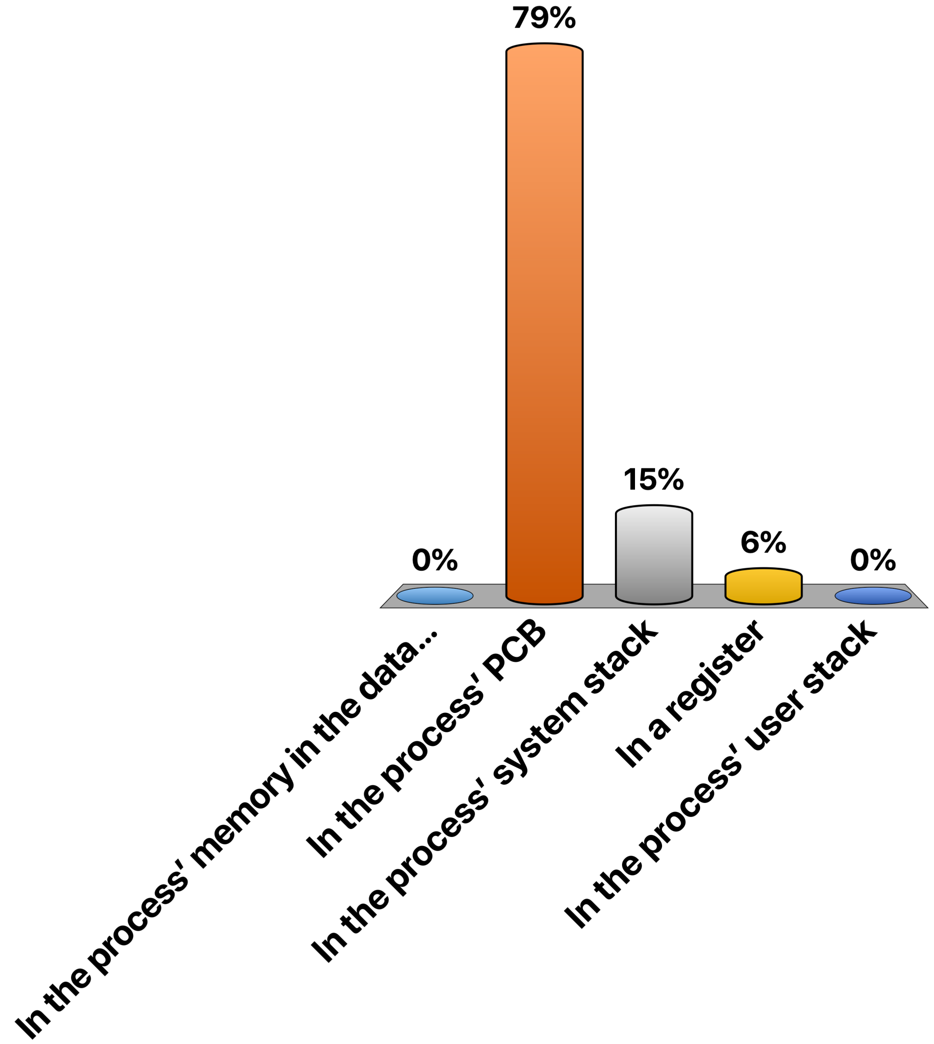
A. In the process' memory in the data area

B. In the process' PCB

C. In the process' system stack

D. In a register

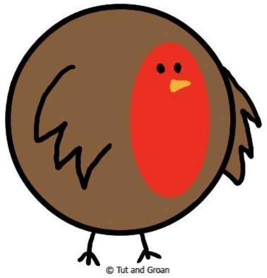
E. In the process' user stack



Preemptive Schedulers

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

Round Robin



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

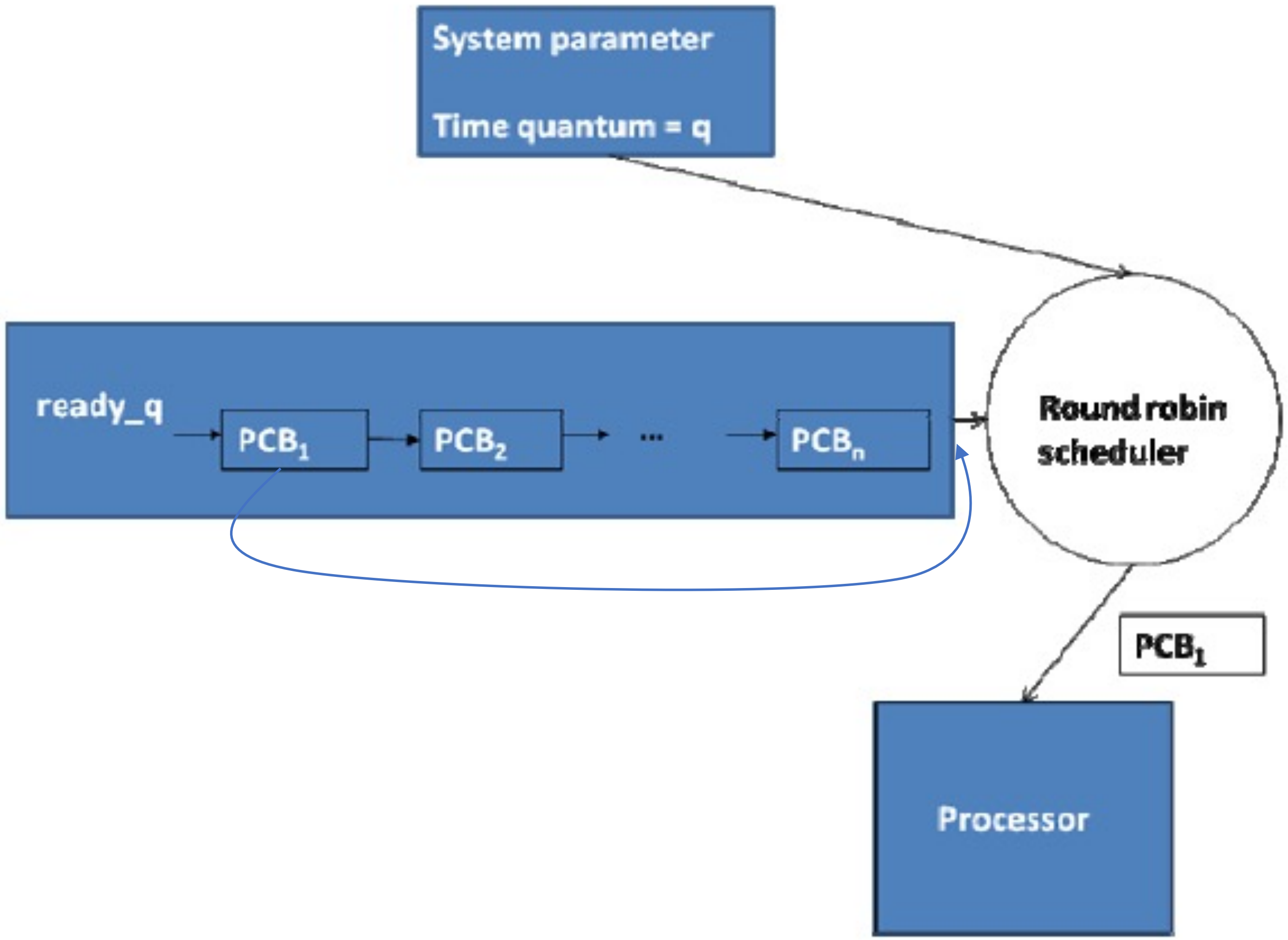
Recall: PCB

```
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;
```

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
- If an interrupt (other than timer) occurs, the process' remaining time quantum is reduced by the amount of time it has used prior to the interrupt

Round Robin


System parameter
Time quantum = q
ready_q
PCB$_1$
PCB$_2$
...
PCB$_n$
Round robin scheduler
PCB$_1$
Processor

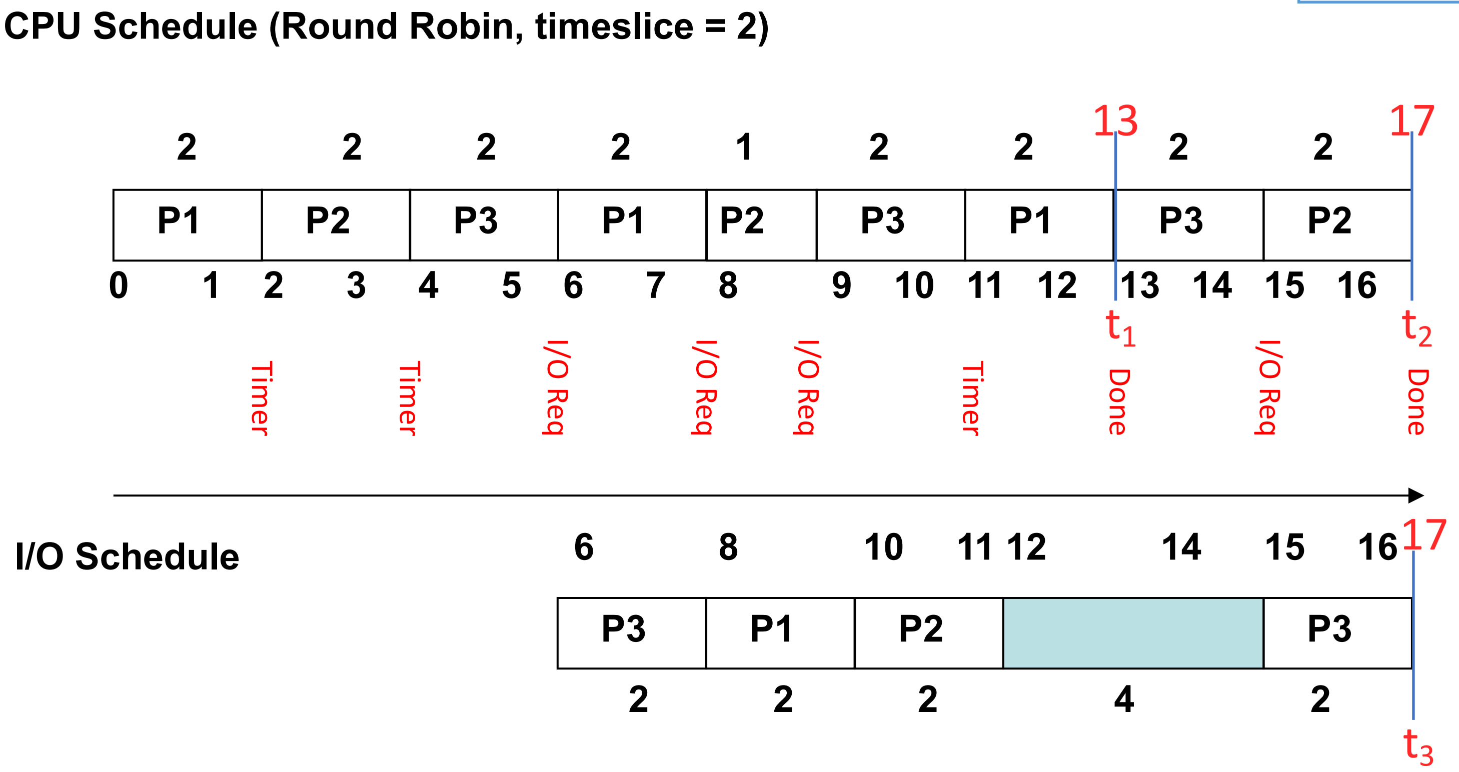
RR example

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

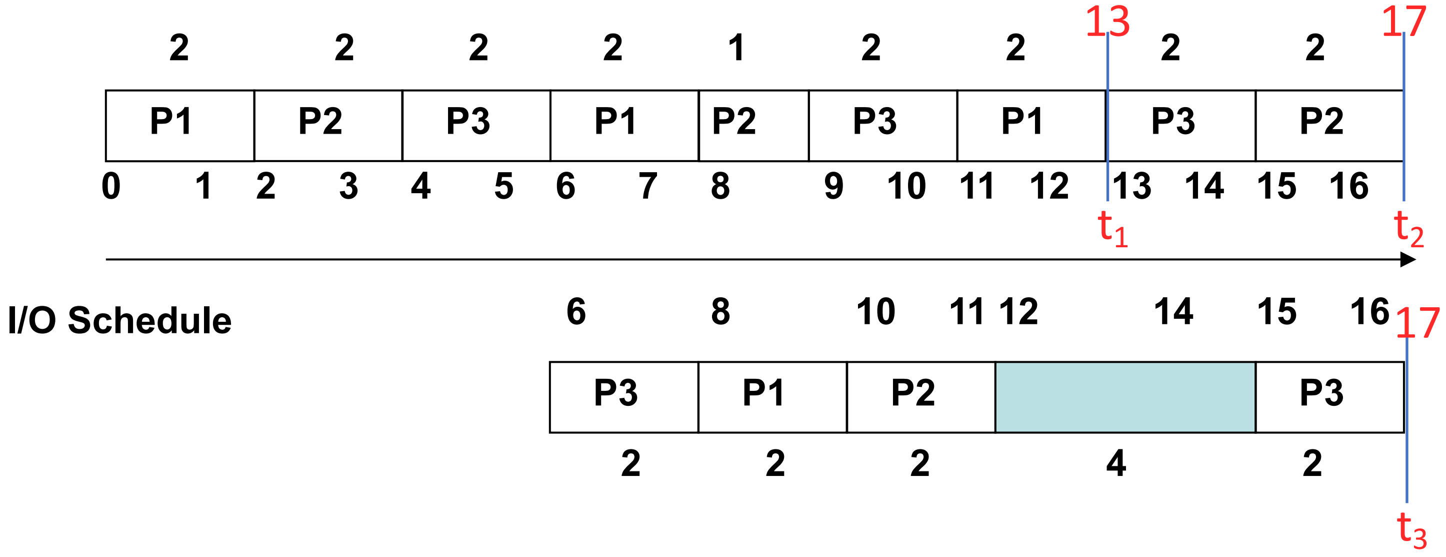
	CPU	I/O	CPU	I/O
P1	4	2	2	
P2	3	2	2	
P3	2	2	4	2



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)



$w_1 = t_1 – e_1$

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

$w_1 = 13 – 8 = 5$

$w_2 = t_2 – e_2$

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

$w_2 = 17 – 7 = 10$

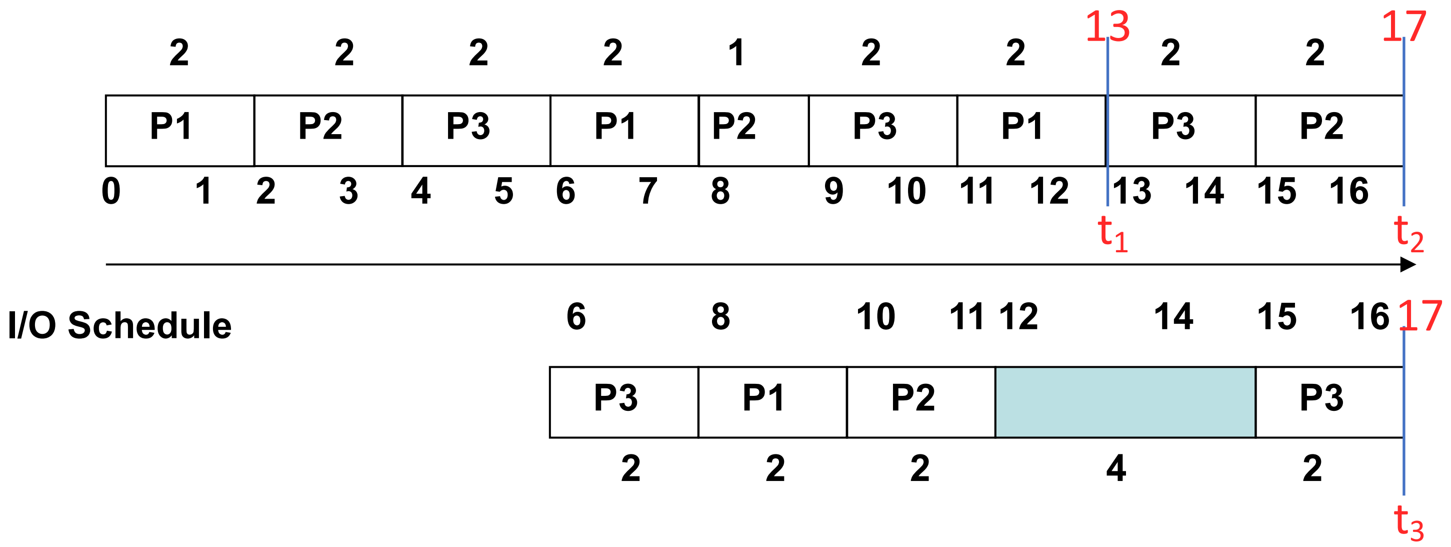
$w_3 = t_3 – e_3$

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

$w_3 = 17 – 10 = 7$

RR solution sketch

CPU Schedule (Round Robin, timeslice = 2)



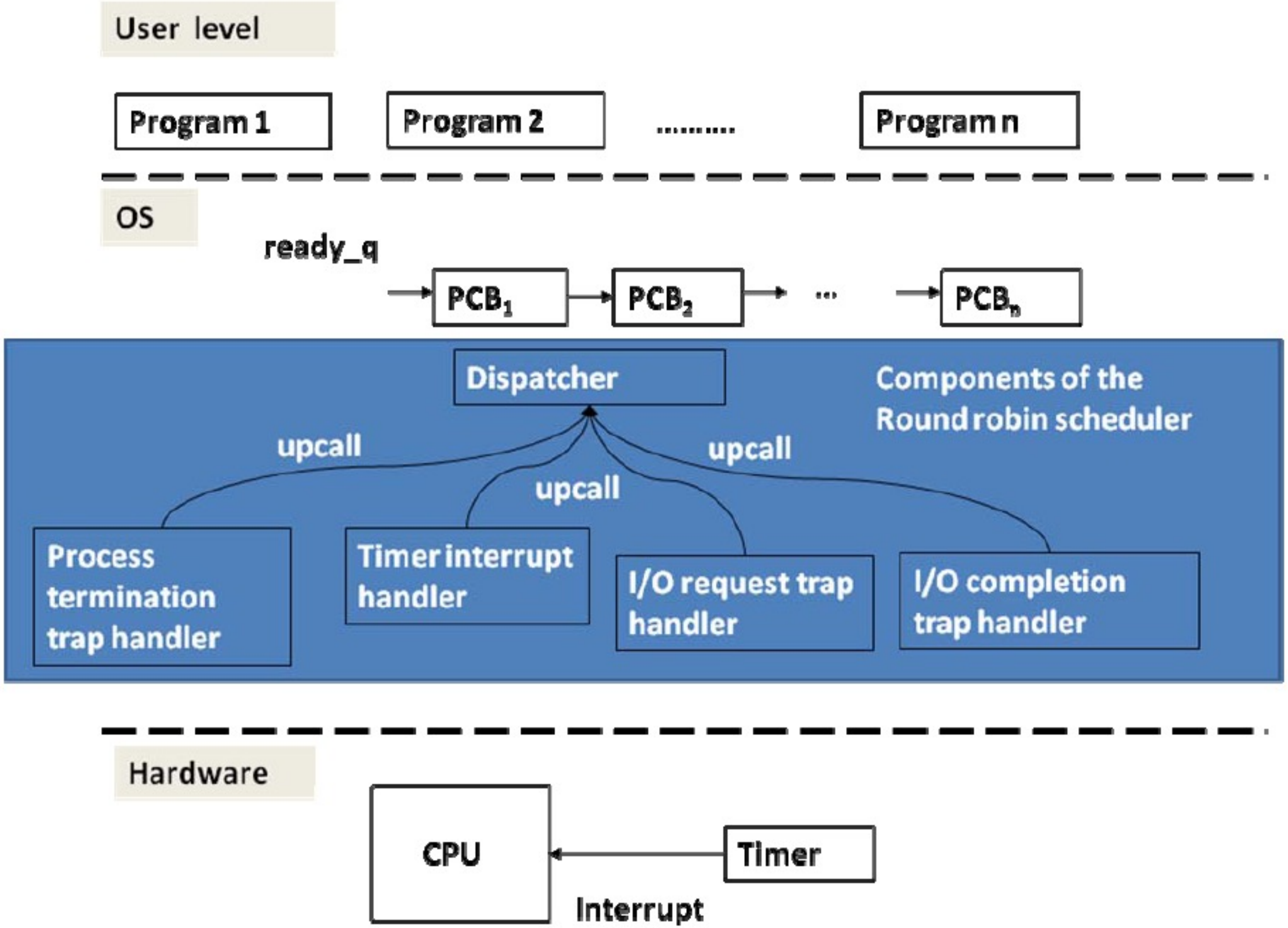
- Potential for unfairness? ➔ no starvation, no convoy effect
- Average waiting time ➔ (5+7+10=)22 / 3
- Average turnaround time ➔ 47 / 3

Recall: PCB

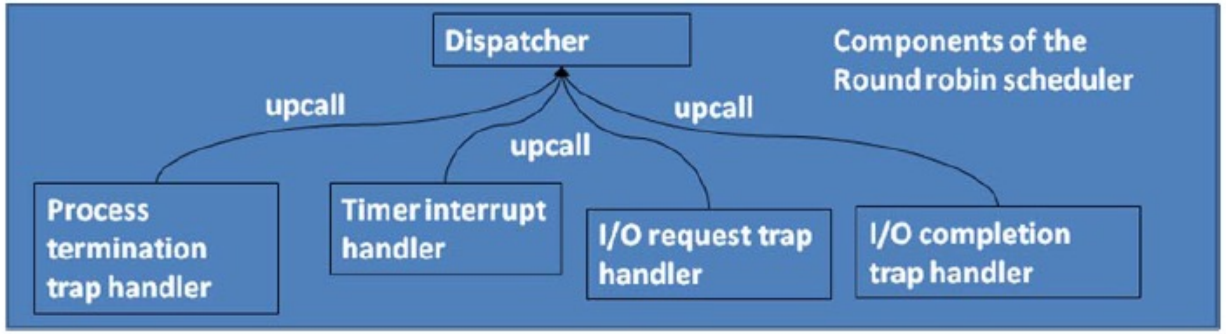
```
enum  state_type {new, ready, running,
               waiting, halted};
typedef struct control_block_type {
   enum state_type state;
   address PC;
   int reg_file[NUMREGS];
   struct control_block *next_pcb;
   int time_left;
   address memory_footprint;
   ….
   ….
} control_block;
```

Implementing the process abstraction

- The OS uses timer interrupts to trigger context switches
- You can think of the schedule/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 trap handler
Hardware
CPU
Timer
Interrupt

Who does what in the OS


Dispatcher
Components of the Roundrobin scheduler
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;
          ┌ save context in PCB;
dispatch ─┤ 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;
```

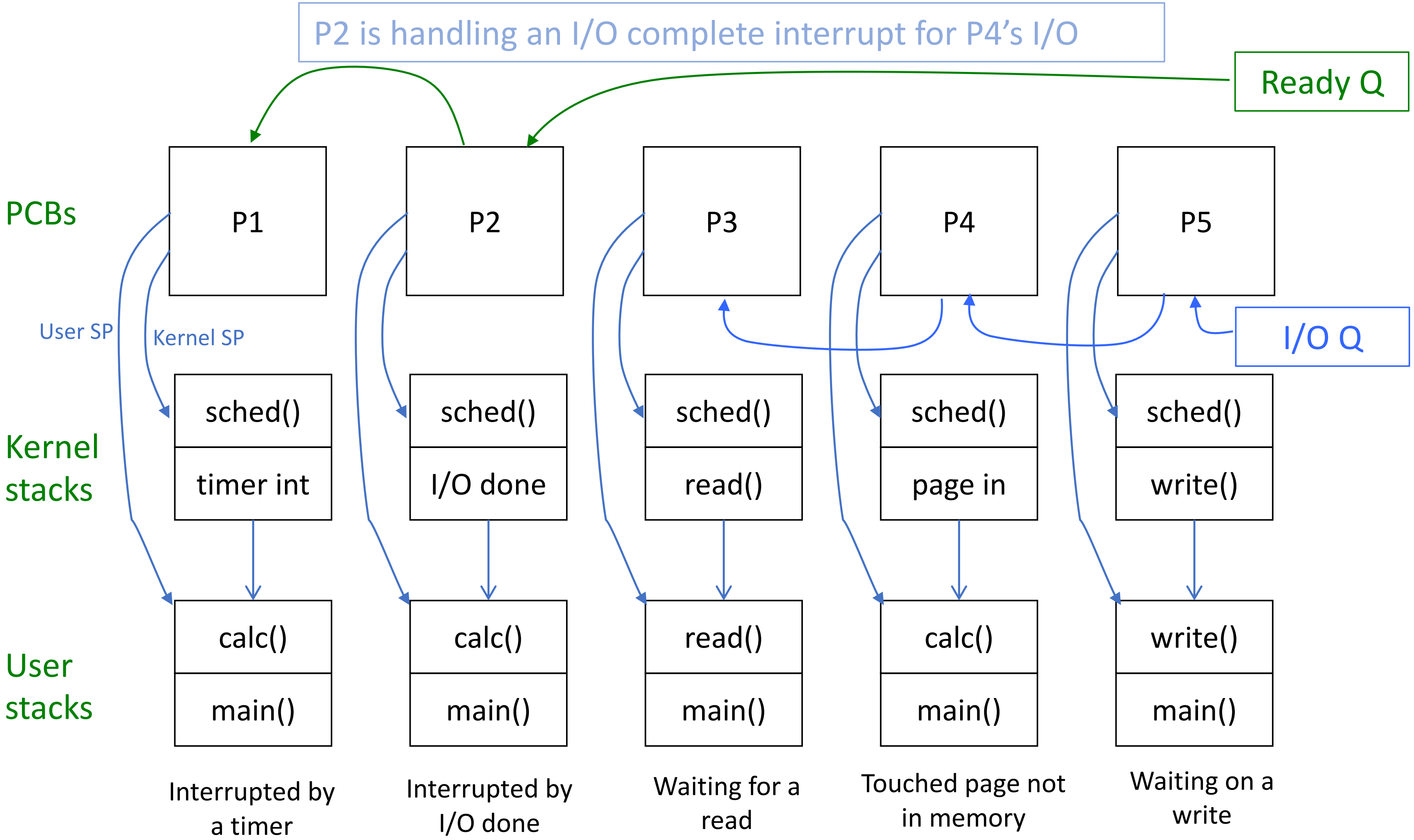
Our Example Program

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



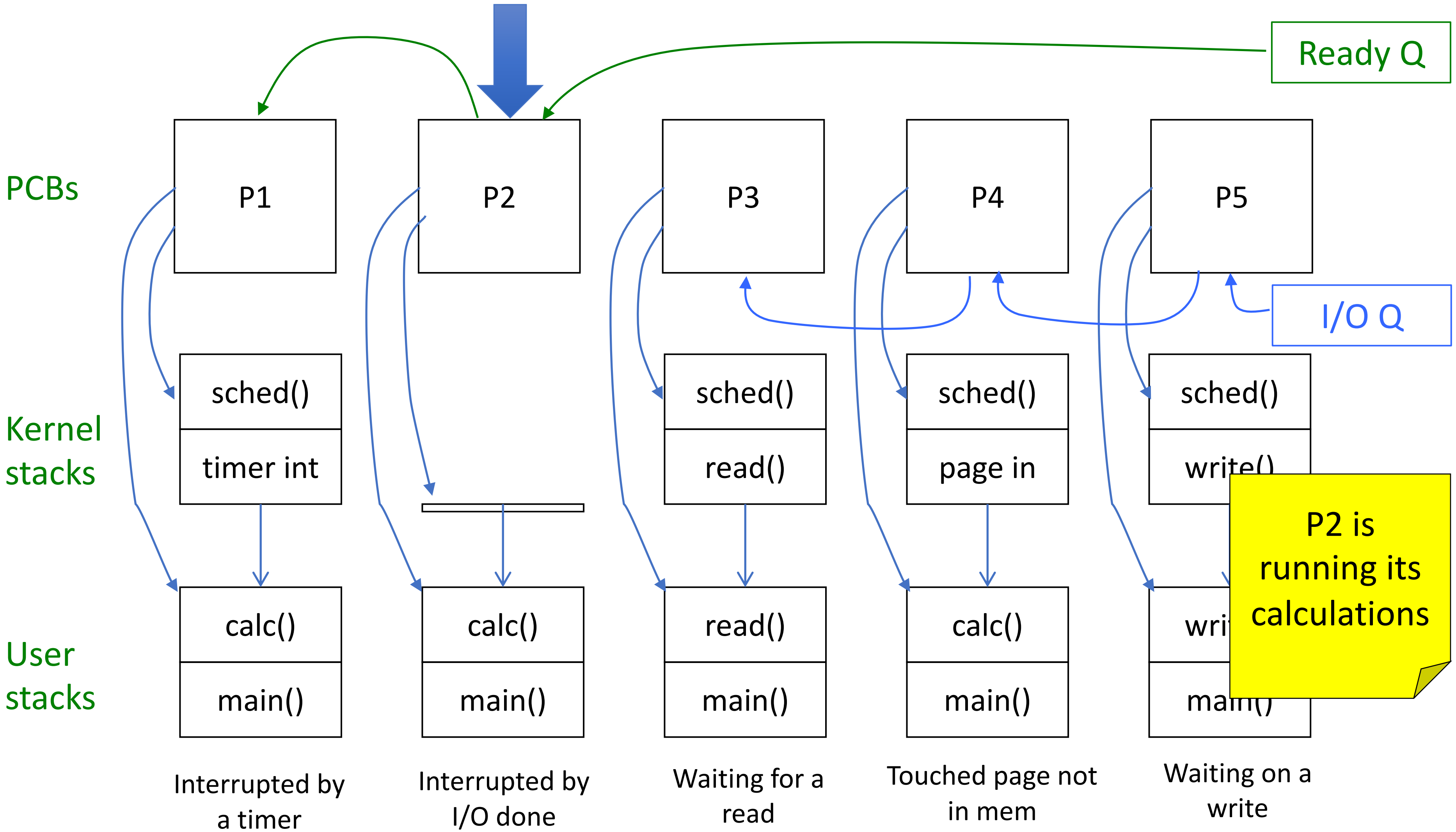
An example using the previous diagram

- 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 and kernel stack
  - 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

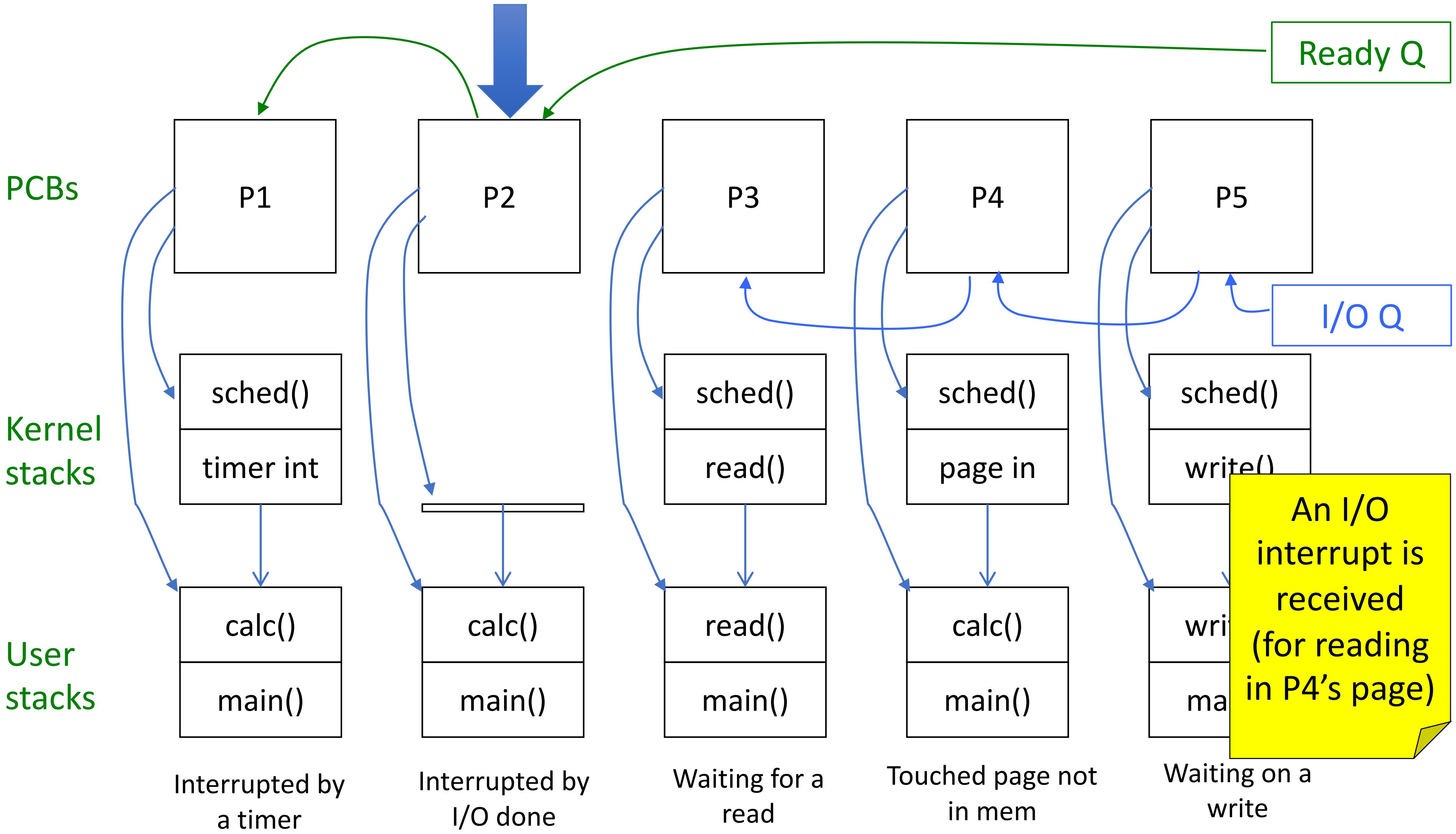
- Foreshadowing chapter 7:
  - With demand paging memory management, when a user program references a part (page) of the program that’s not resident in physical memory,
    - The hardware causes a page-fault trap
    - The operating system treats a page-fault trap as a request to read in the faulting 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 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!



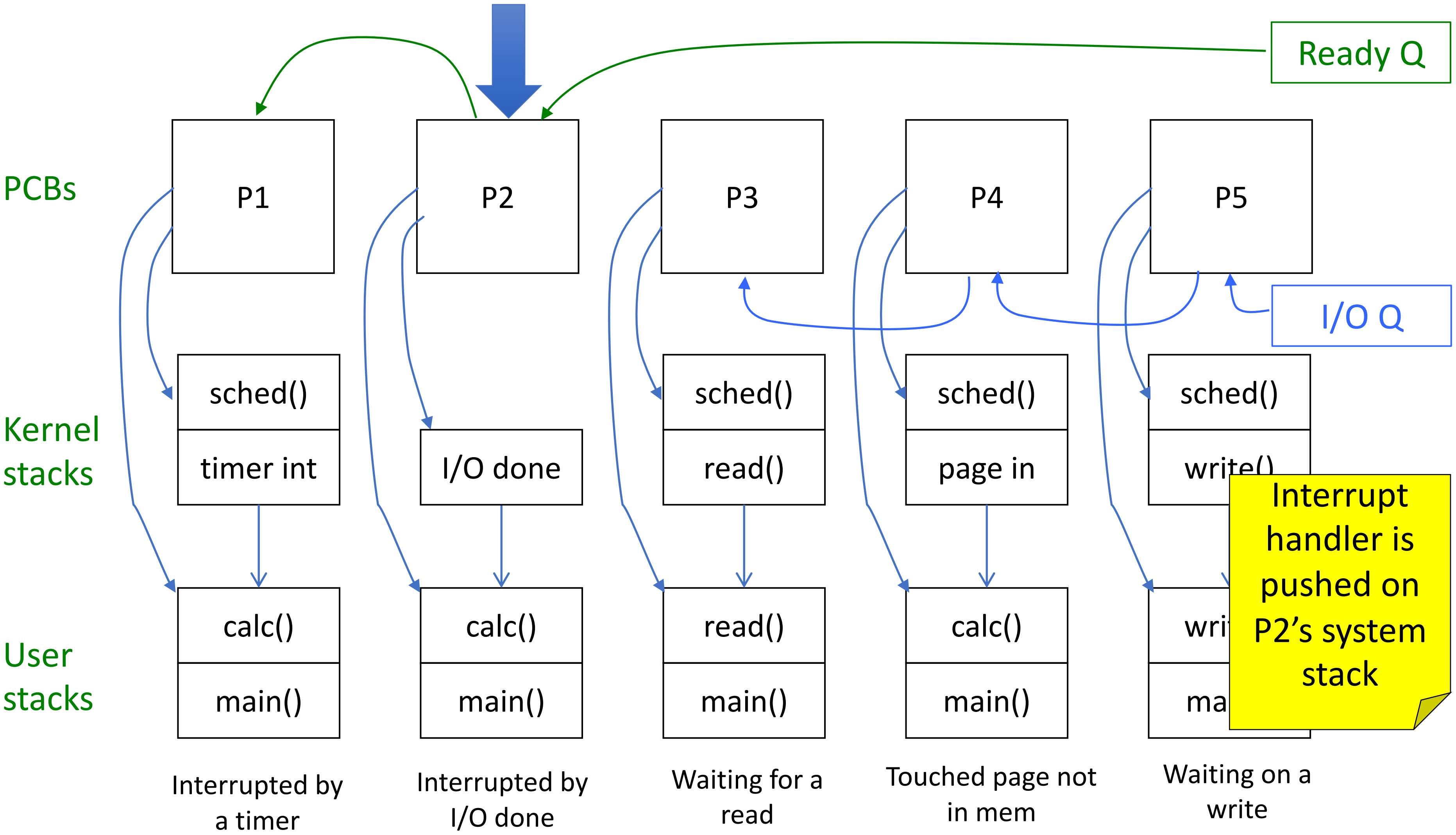
P2 is running its calculations

Process example



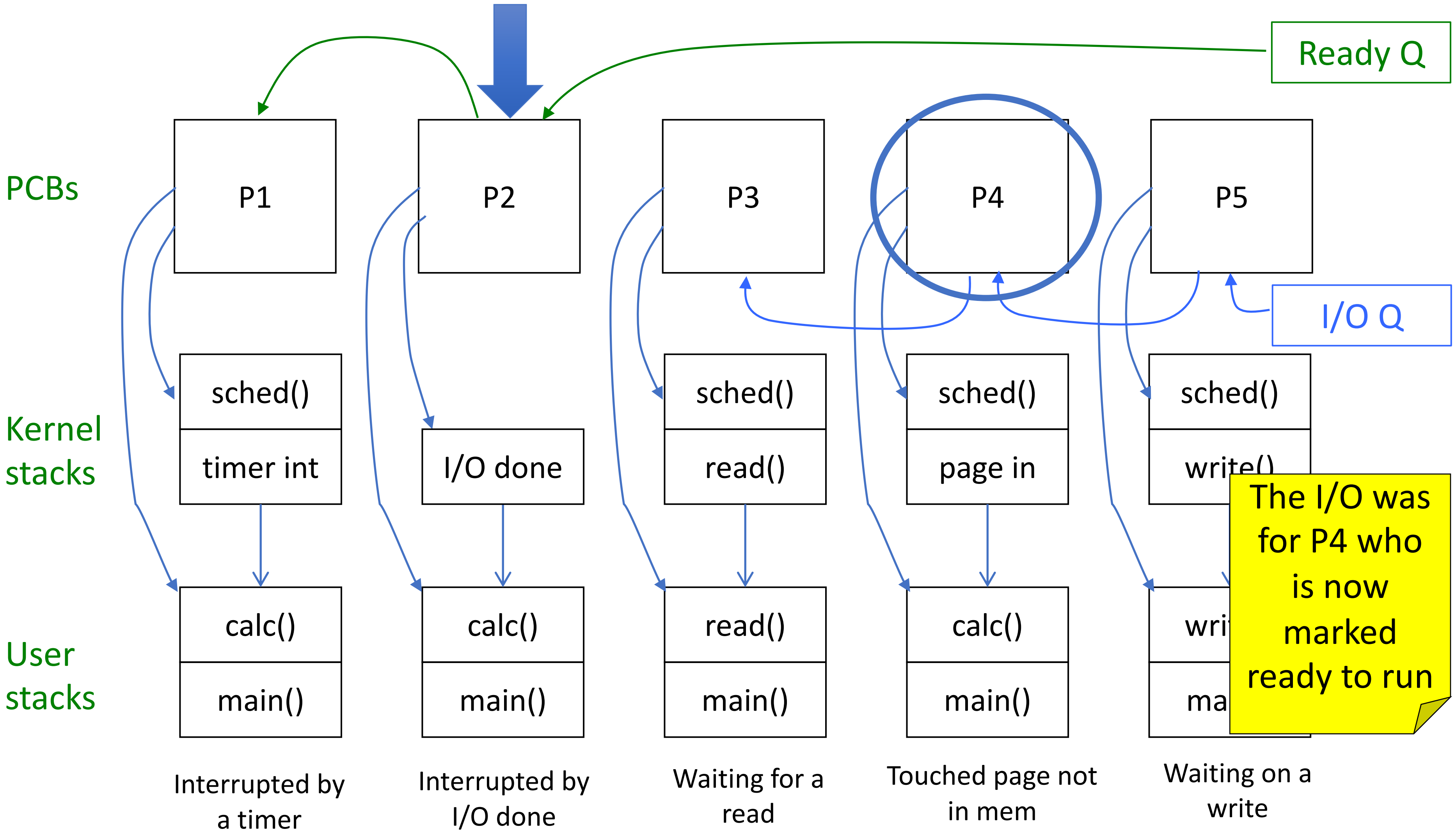
An I/O interrupt is received (for reading in P4's page)

Process example



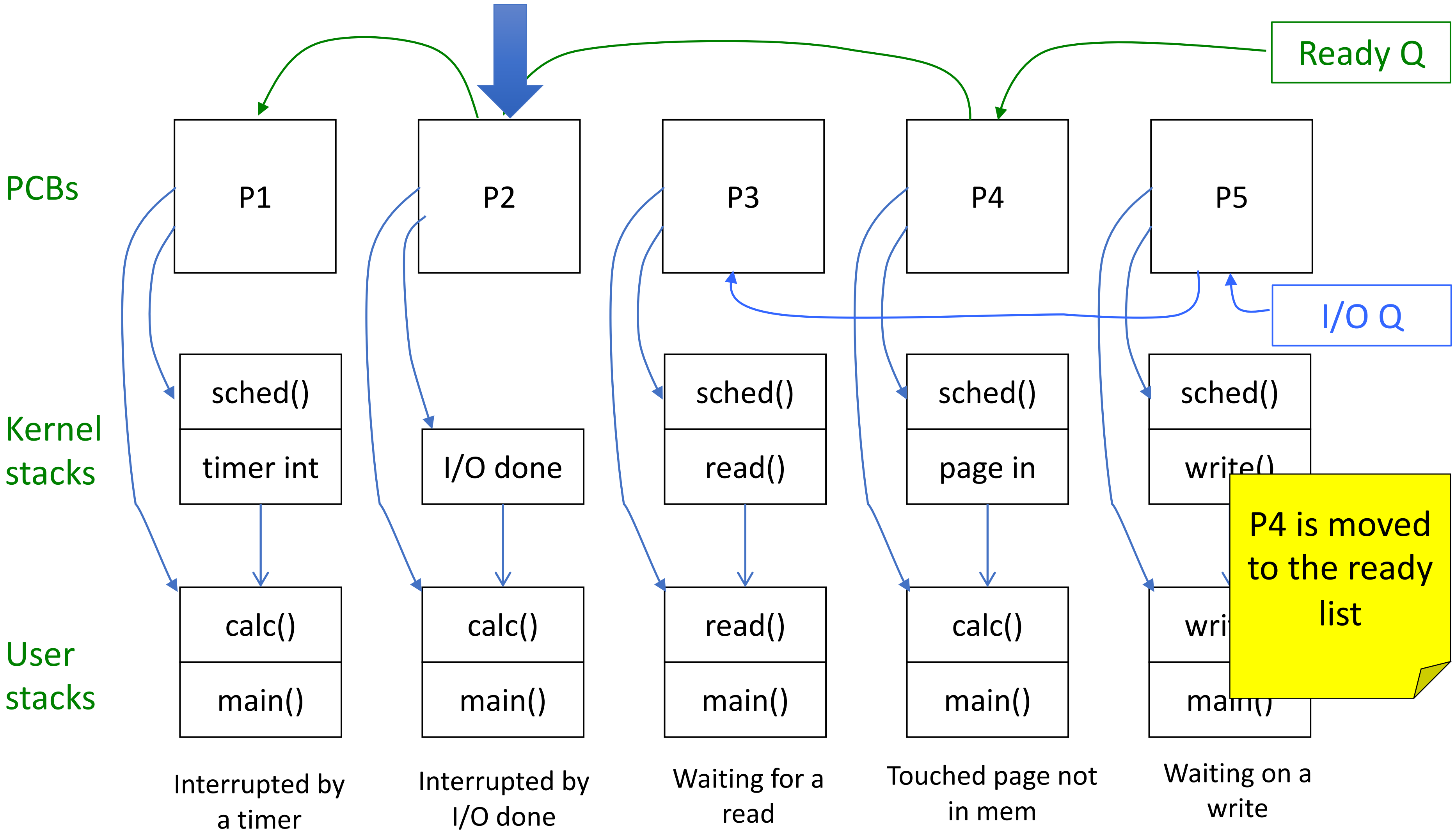
Interrupt handler is pushed on P2's system stack

Process example



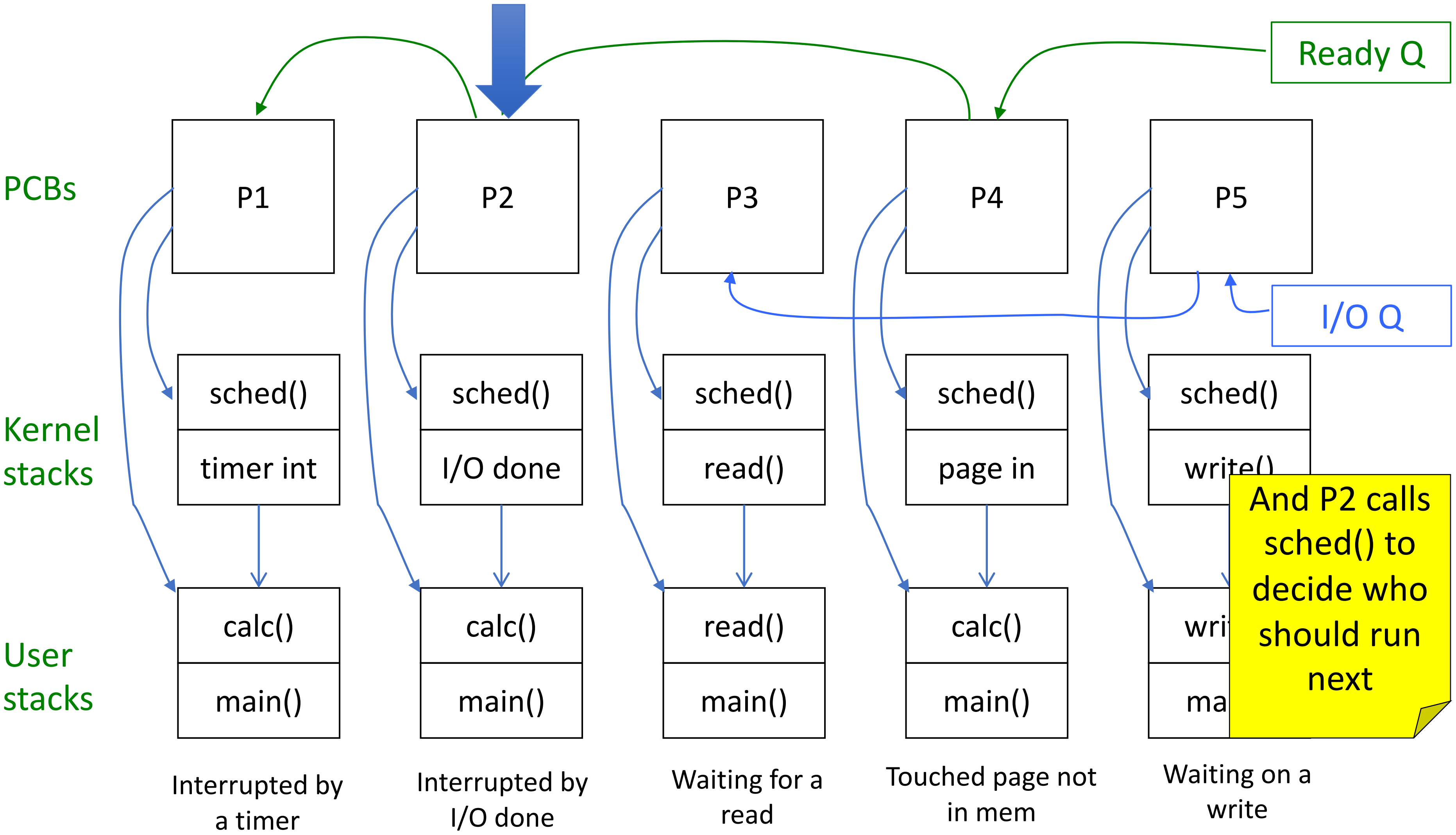
The I/O was for P4 who is now marked ready to run

Process example



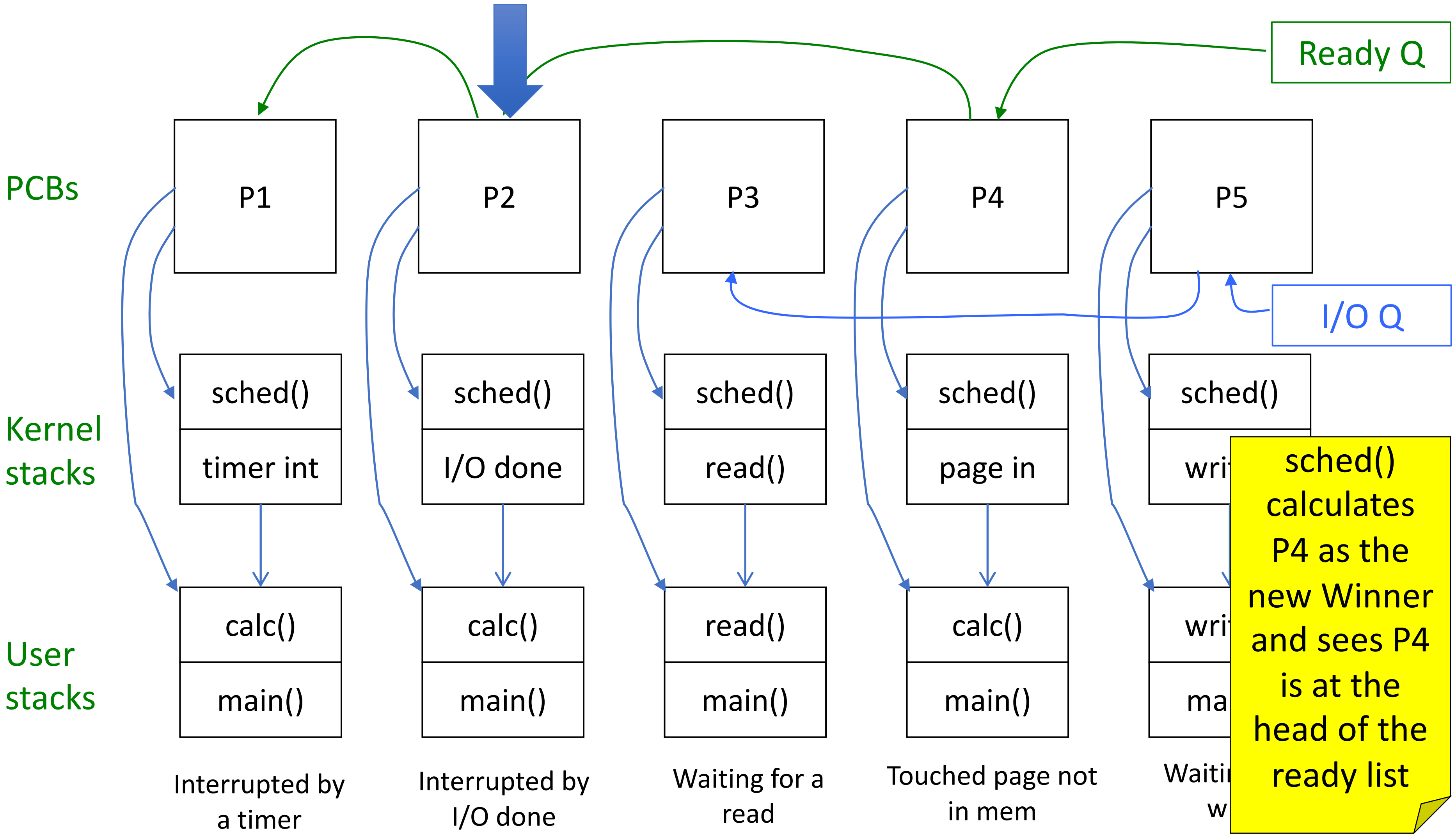
P4 is moved to the ready list

Process example



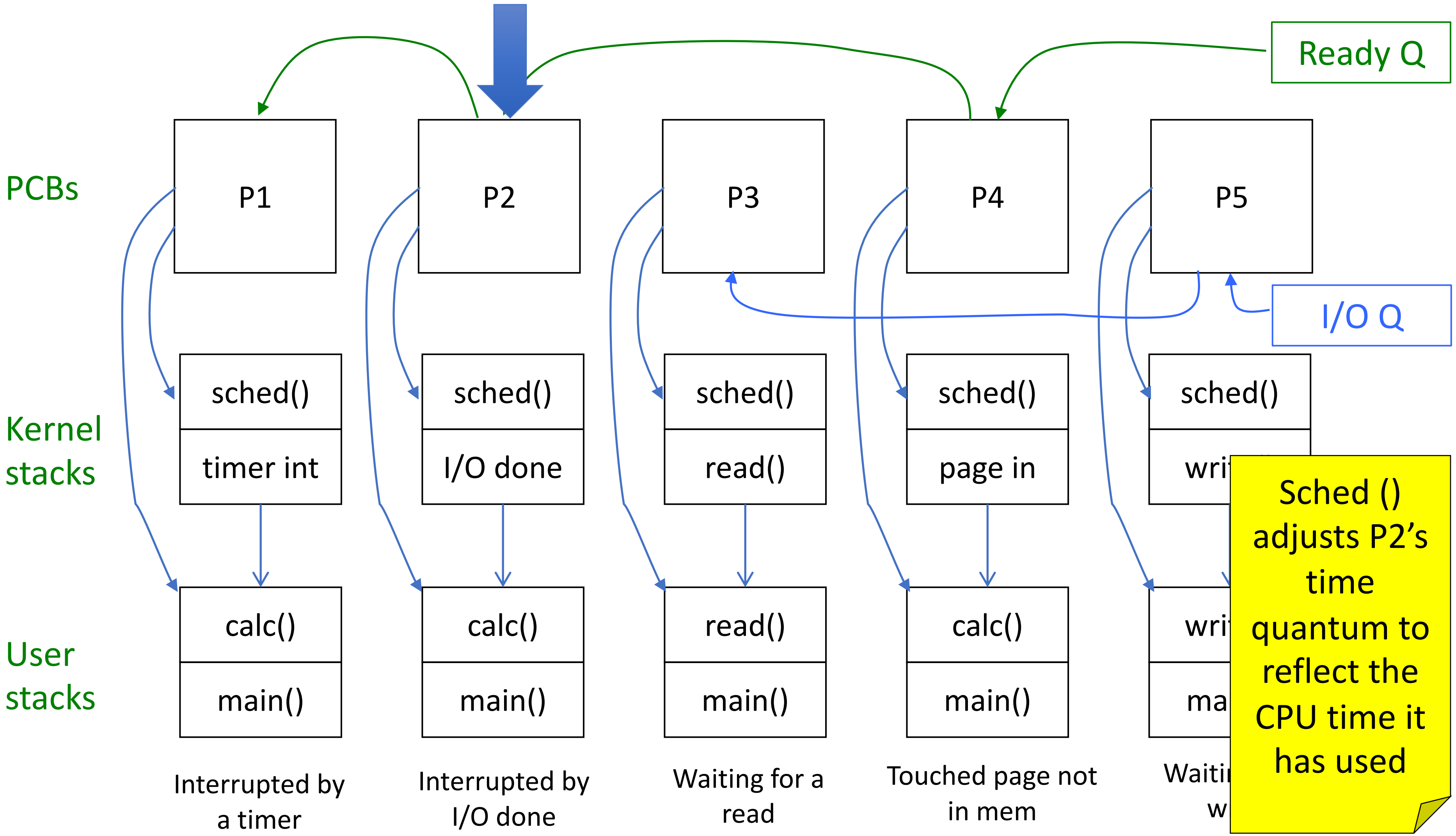
And P2 calls sched() to decide who should run next

Process example



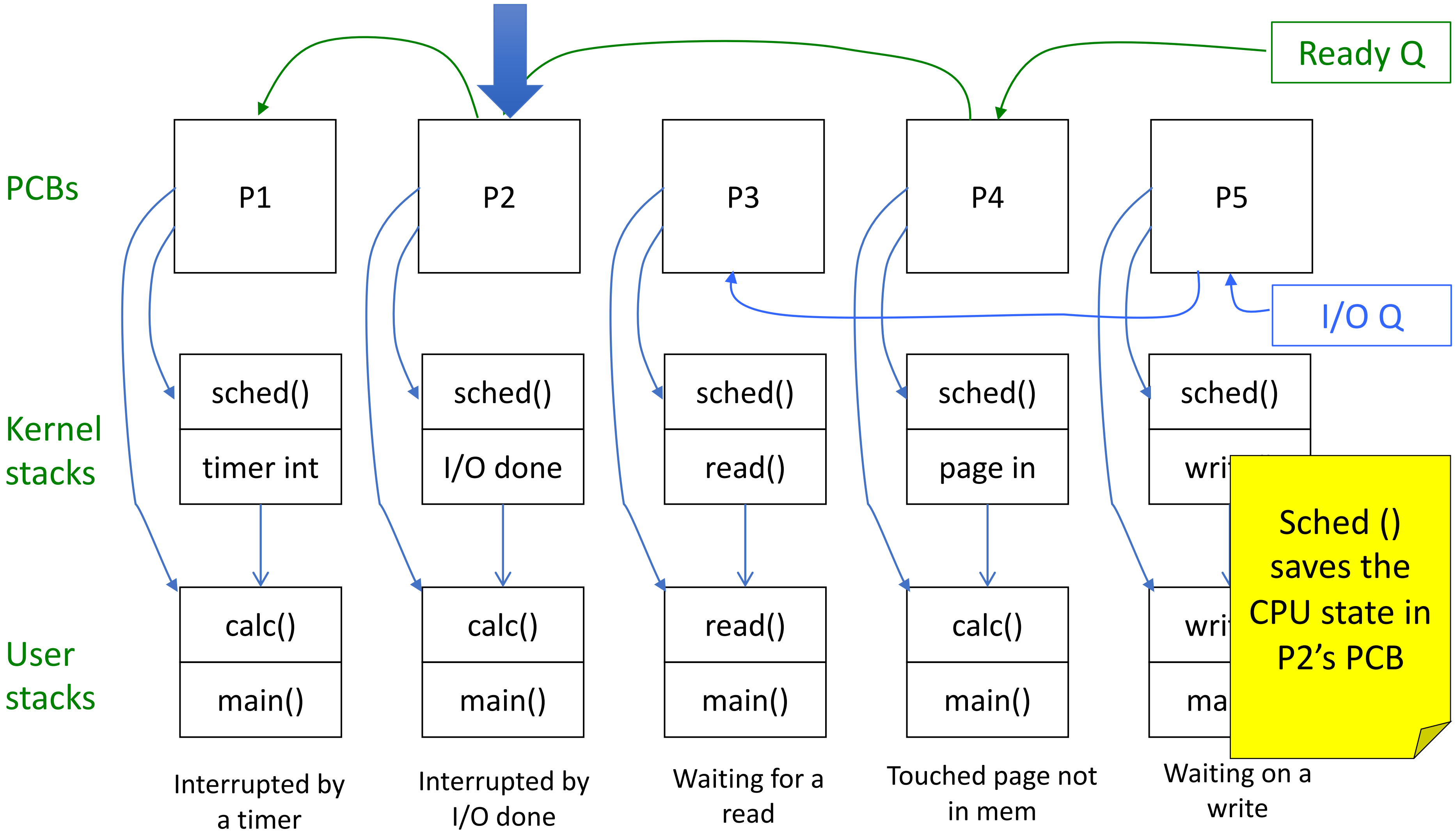
sched() calculates P4 as the new Winner and sees P4 is at the head of the ready list

Process example



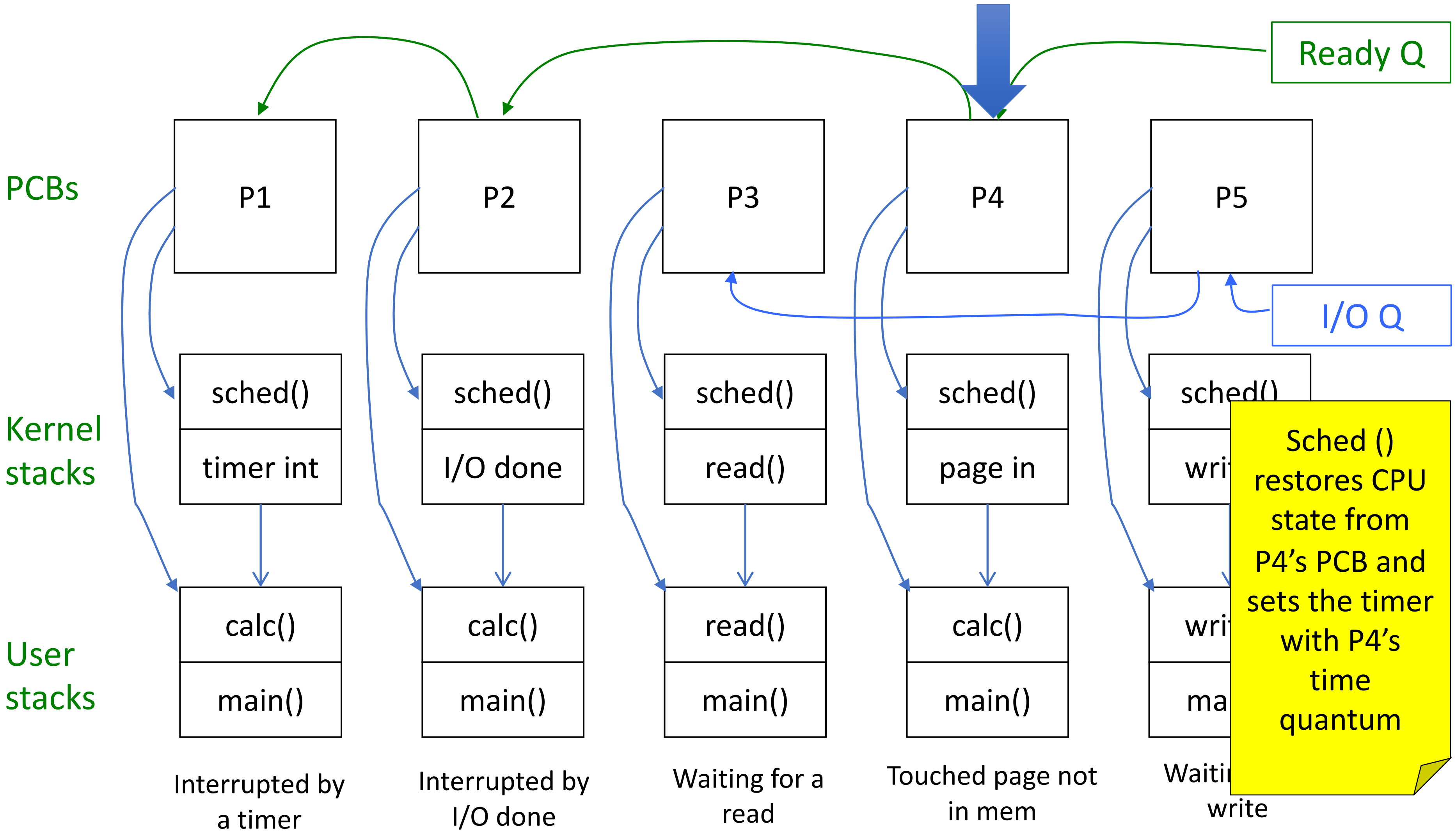
Sched () adjusts P2's time quantum to reflect the CPU time it has used

Process example



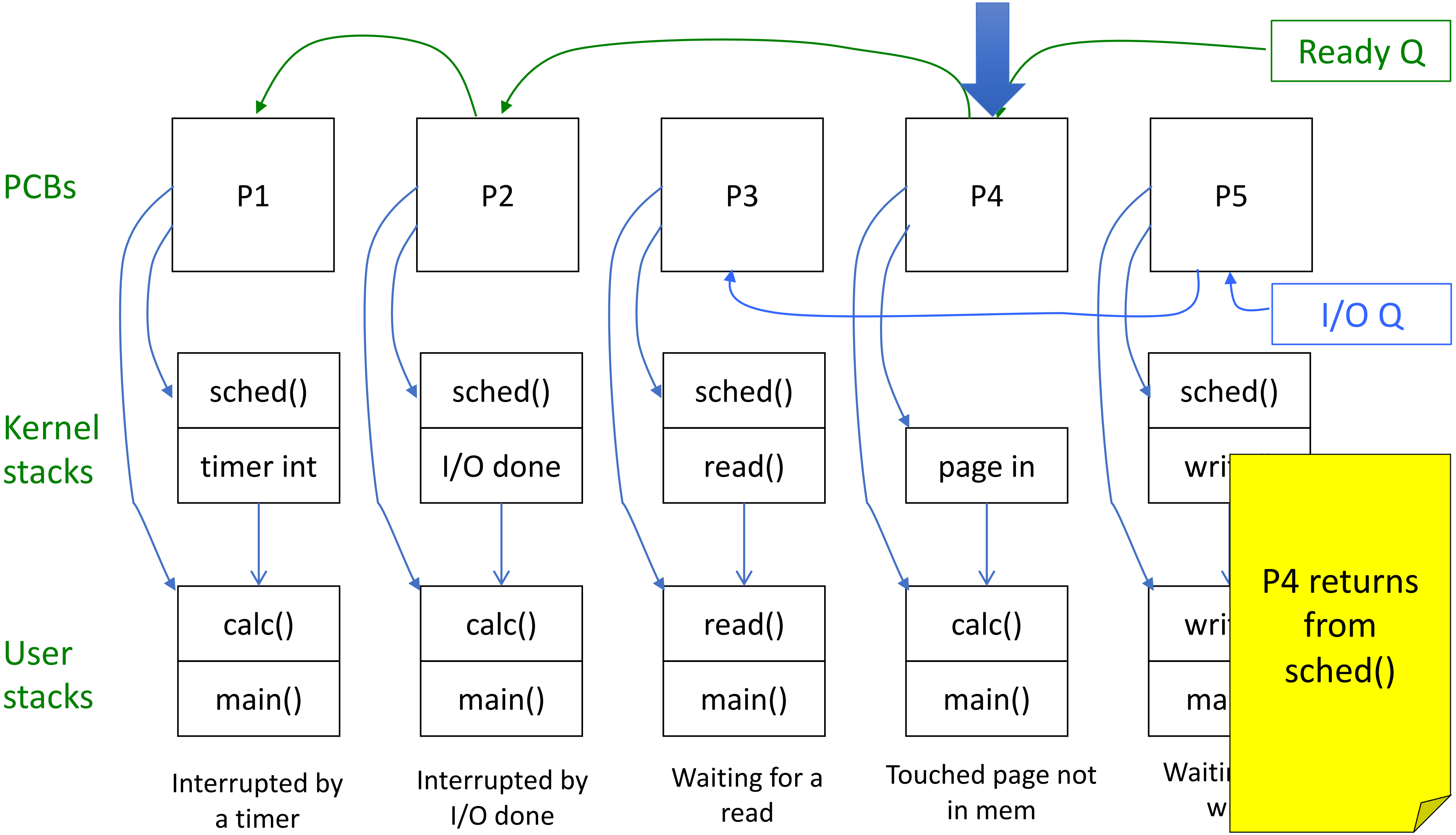
Sched () saves the CPU state in P2's PCB

Process example



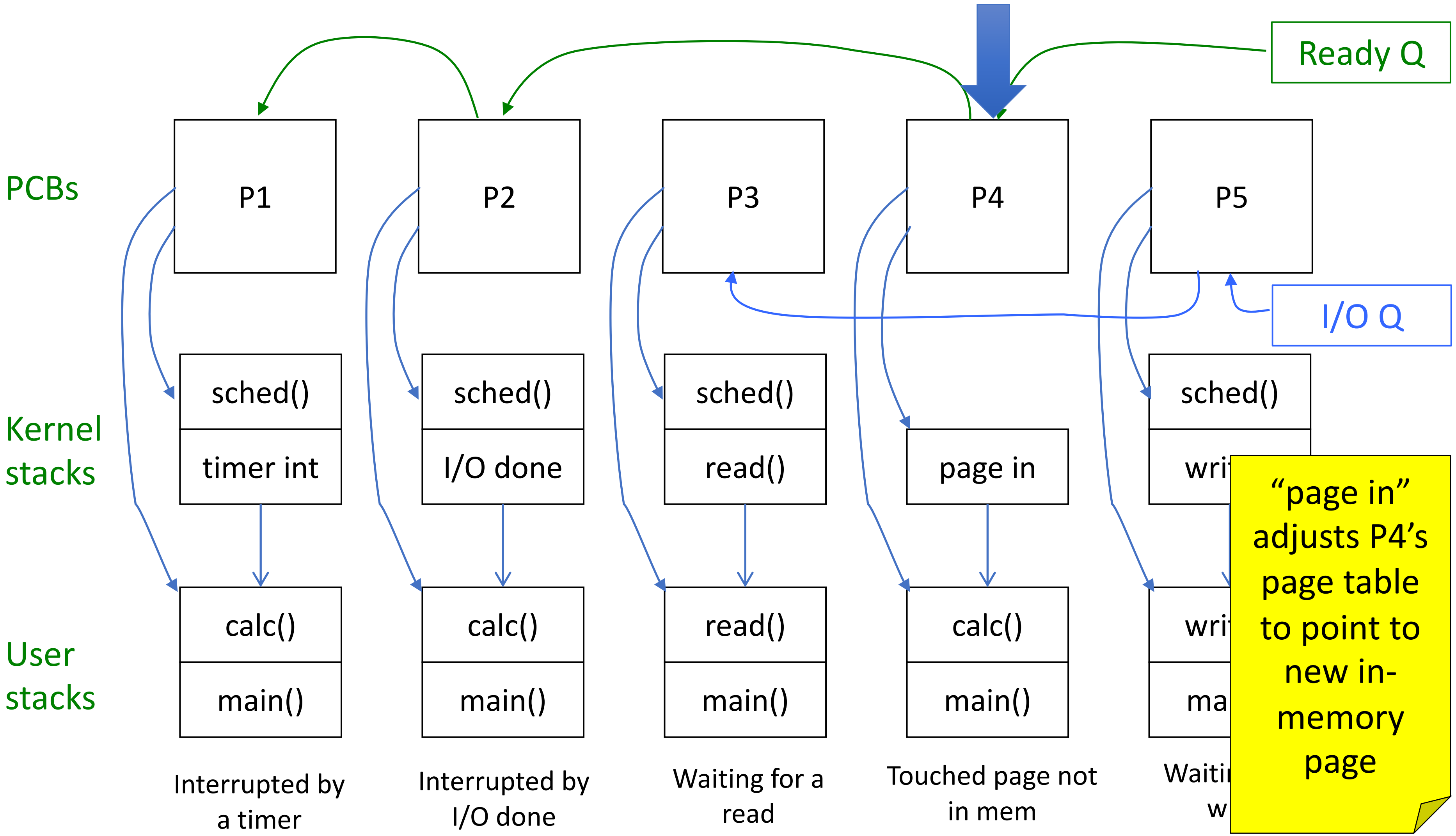
Sched () restores CPU state from P4's PCB and sets the timer with P4's time quantum

Process example



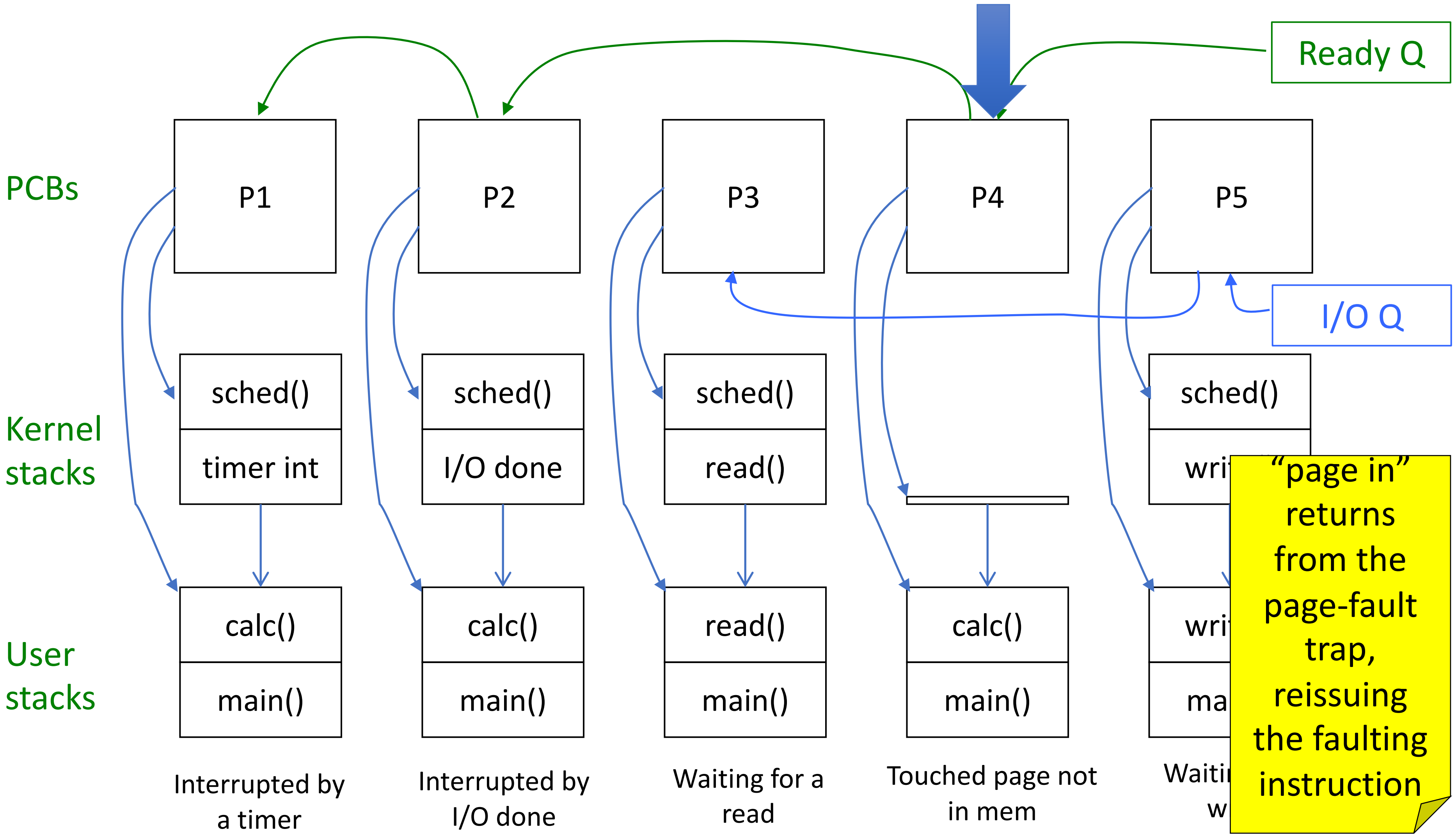
P4 returns from sched()

Process example



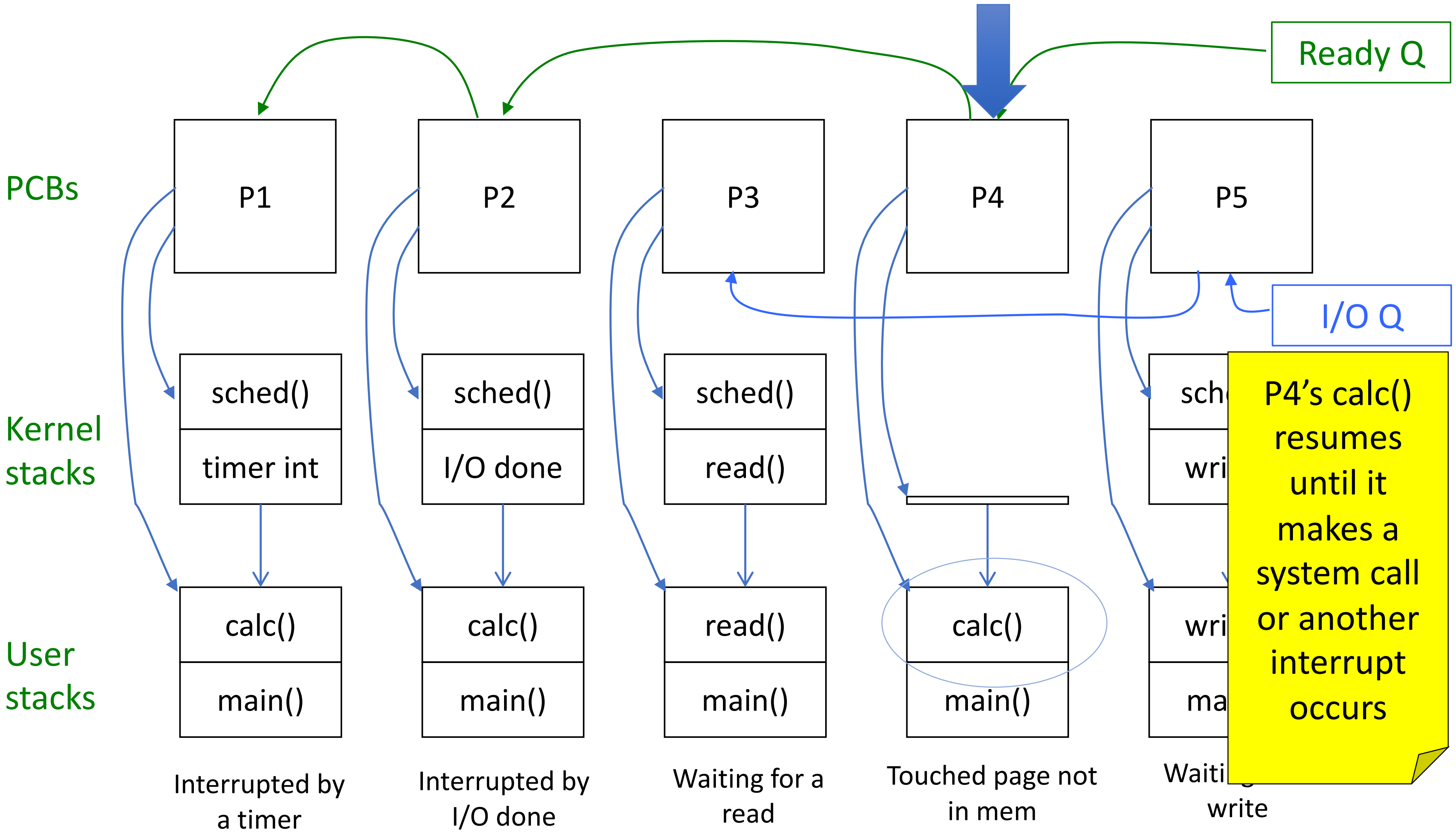
"page in" adjusts P4's page table to point to new in-memory page

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

A preemptive scheduler…

A. Can only be implemented with a timer interrupt

B. Can only be implemented with I/O completion interrupt

C. Can only be implemented with a system call trap

D. Can be implemented with any type of interrupt

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

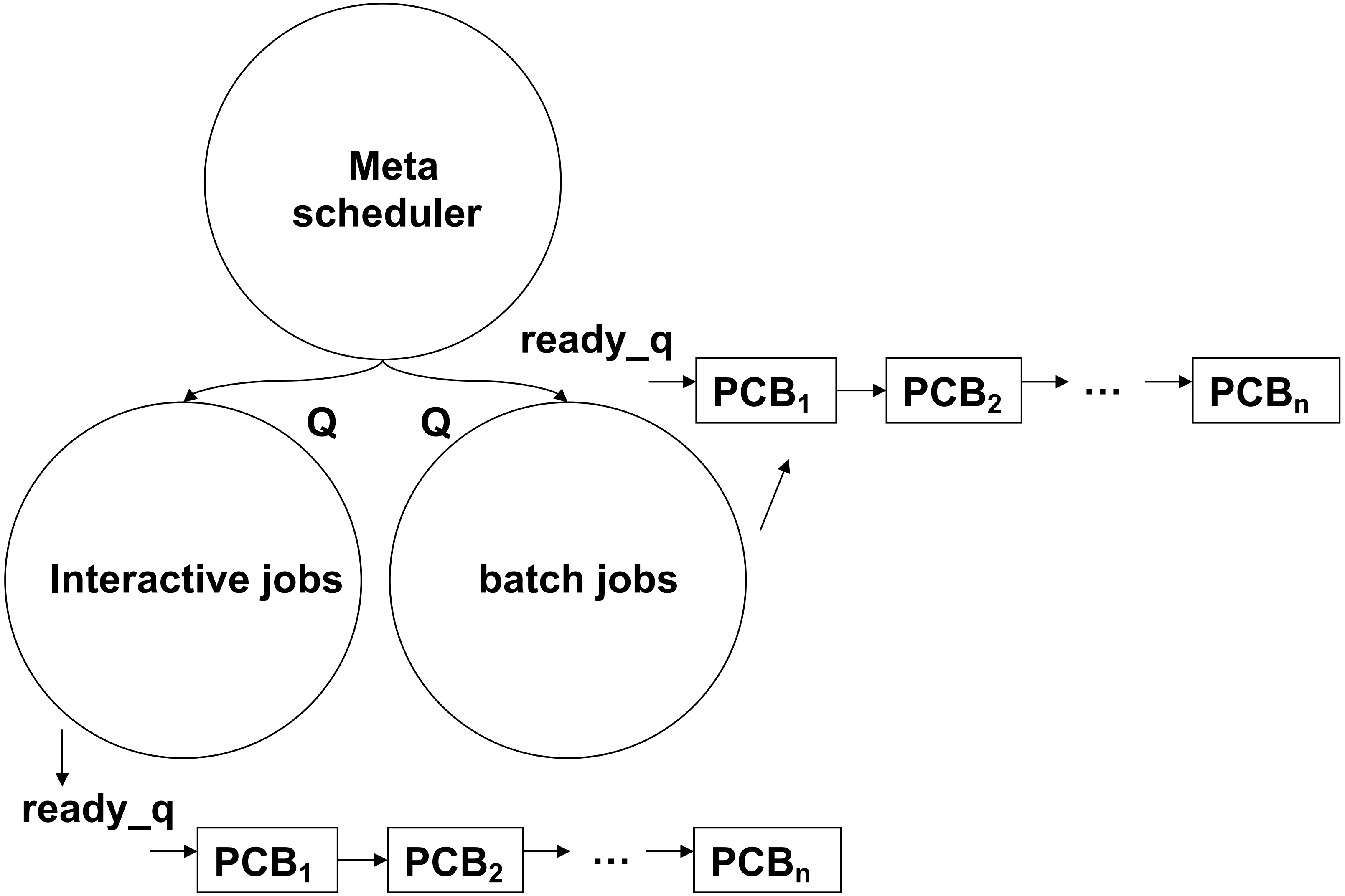
A. The system stack

B. The PCB for that process

C. The user stack

D. The heap space of the process

Multi-Level Scheduler

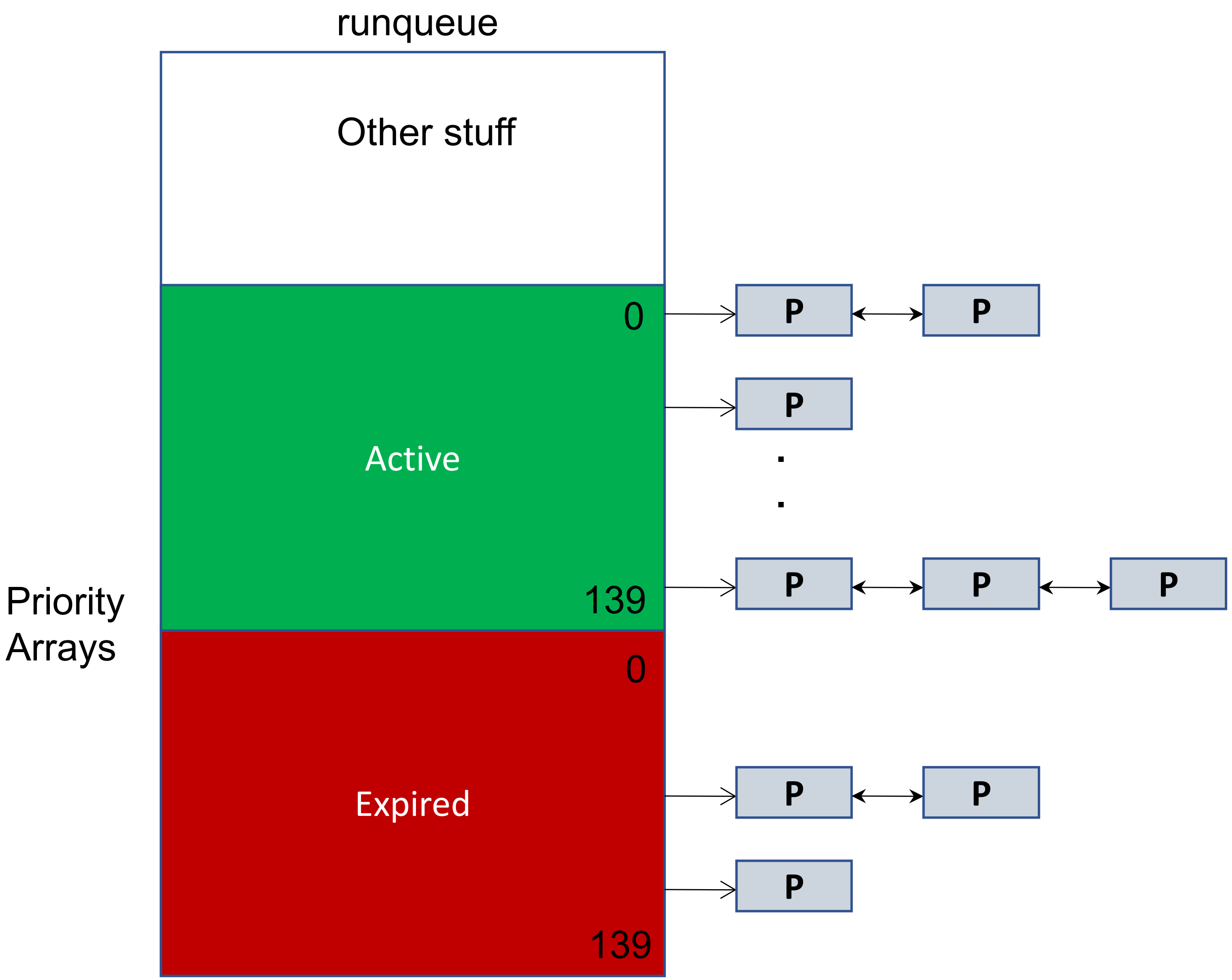


Name	Property	Scheduling criterion	Pros	Cons
FCFS	Intrinsically non-preemptive; could accommodate preemption at time of I/O completion events	Arrival time (intrinsic property)	Fair; no starvation;	high variance in response time; convoy effect
SJF	Intrinsically non-preemptive; could accommodate preemption 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

Example: Linux scheduler (Kernel 2.6)

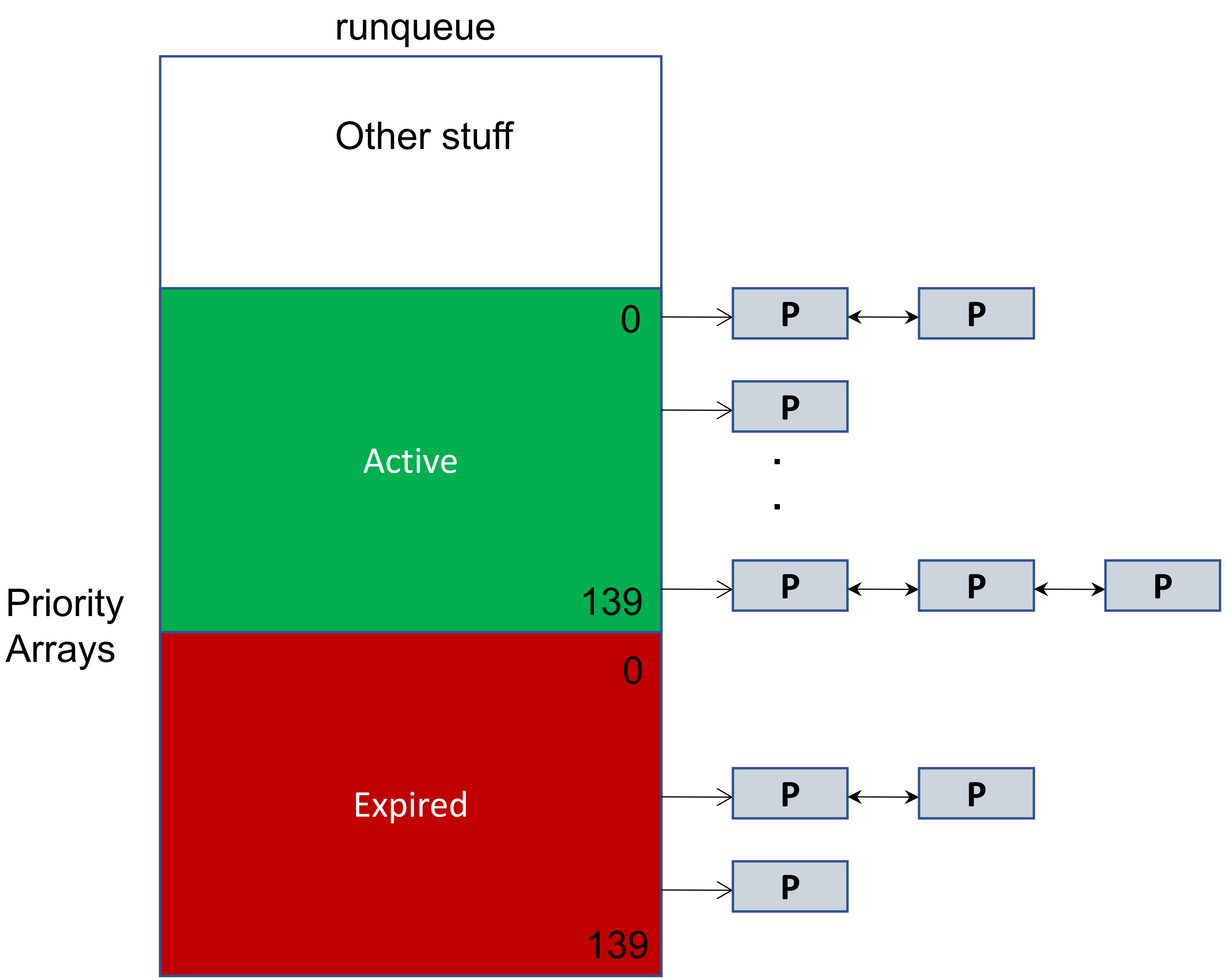


- Winner?
- I/O call?
- Quantum expires?
- I/O completion?
- No task in active?

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).

Why is this scheduler O(1)?



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:

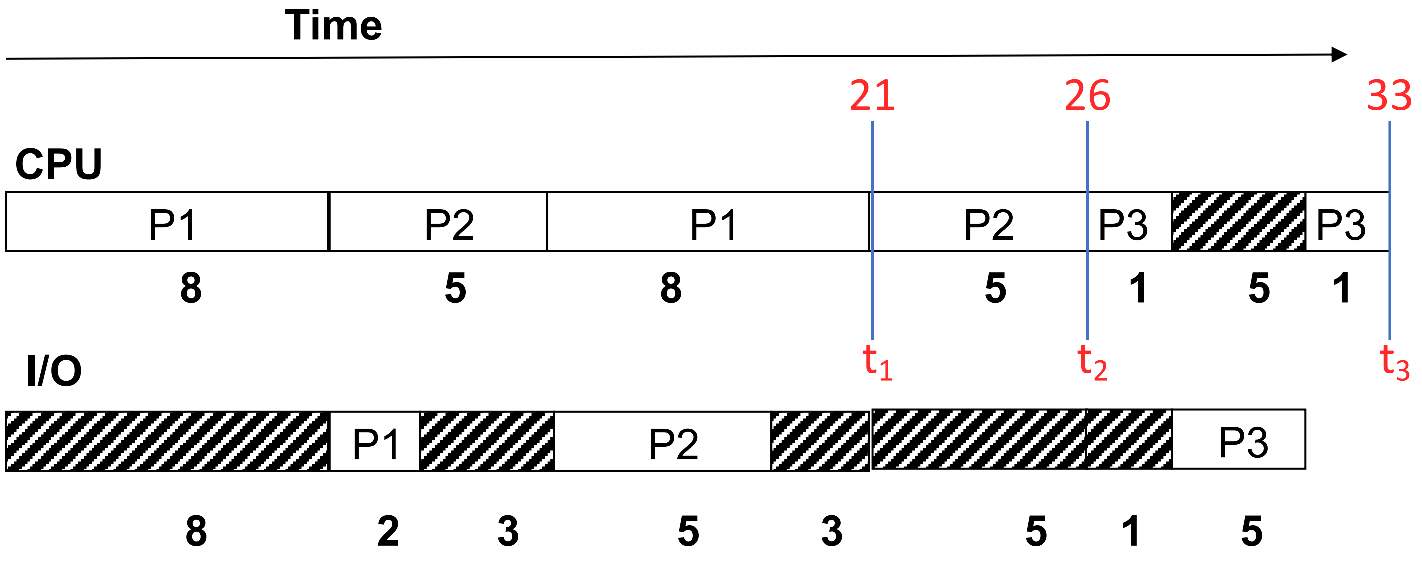
CPU burst | I/O Burst | CPU Burst

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

b) What is the response time for each process?

c) 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

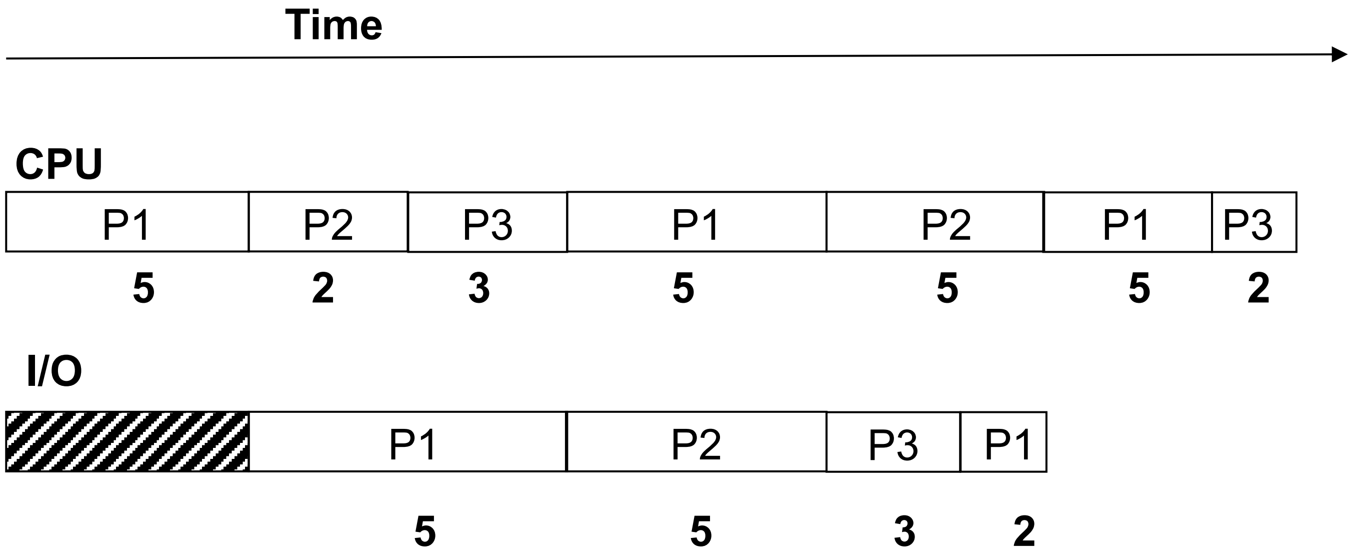
Consider a non-preemptive FCFS process scheduler. There are three processes in the scheduling queue and assume that all three of them are ready to run. As the scheduling discipline suggests, the scheduler always respects the arrival time in selecting a winner. Assume that P1, P2, and P3 arrive in that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

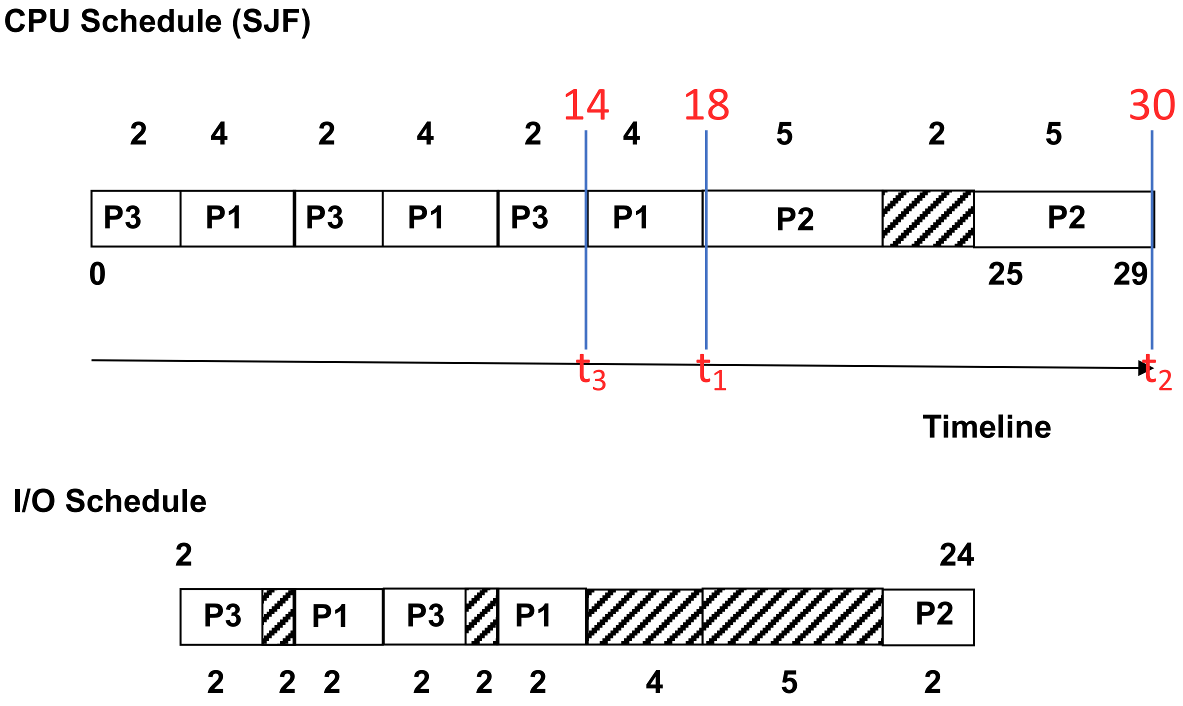
Each process exits the system once its CPU and I/O bursts as shown above are complete.

a) Show the CPU and I/O timelines that result with SJF scheduling from t = 0 until all three processes exit the system.

b) What is the waiting time for each process?

c) What is the average throughput of the system?

SFJ example solution sketch



$w_1 = t_1 – e_1$

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

$w_1 = 18 – 16 = 2$

$w_2 = t_2 – e_2$

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

$w_2 = 30 – 12 = 18$

$w_3 = t_3 – e_3$

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

$w_3 = 14 – 10 = 4$