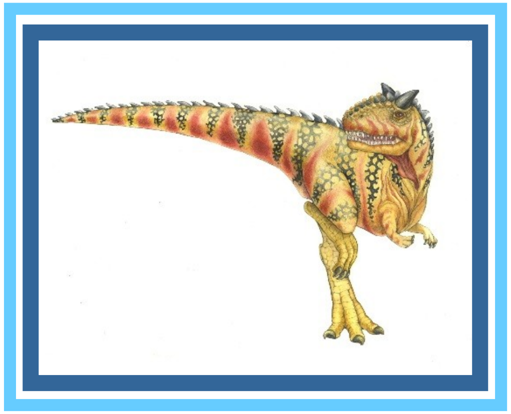
Theoretical Unit 3

(Book Chapters 5)

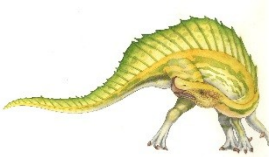
CPU Scheduling

(edited & enhanced by rufino@ipb.pt, 2025/2026)



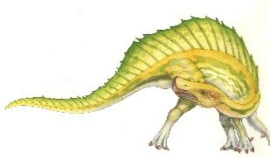
Unit 3: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Exercises



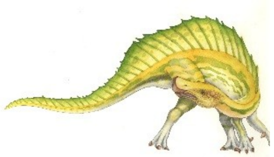
Objectives

- Describe various CPU scheduling algorithms
- Assess CPU scheduling algorithms based on scheduling criteria
- Explain the issues related to multiprocessor and multicore scheduling
- Describe various real-time scheduling algorithms
- Describe the scheduling algorithms used in the Windows, Linux, and Solaris operating systems
- Apply modeling and simulations to evaluate CPU scheduling algorithms



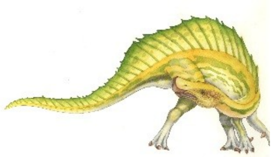
Theoretical Unit 3

3.1 Basic Concepts



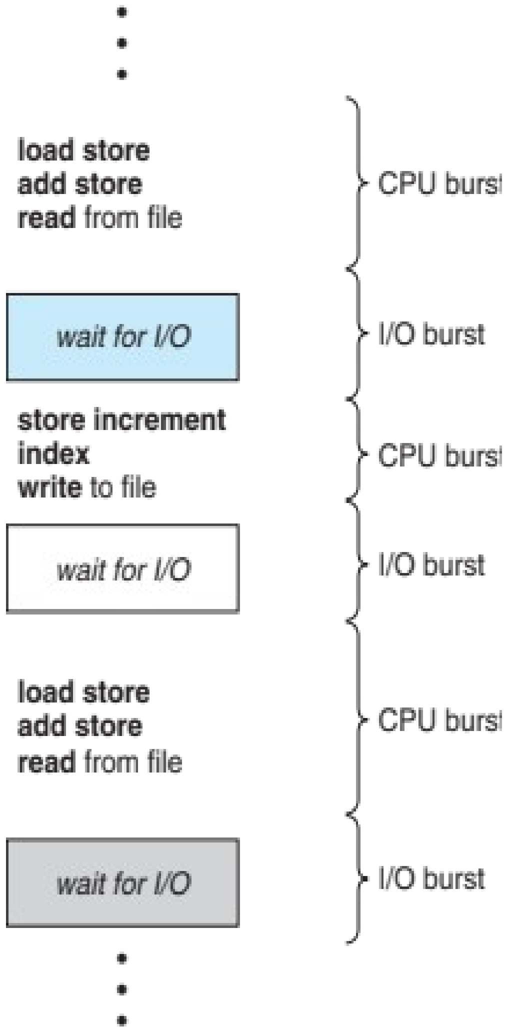
Basic Concepts

- **Multiprogramming** goal: to maximize system resources utilization
  - OS keeps several processes in RAM, some **ready** to run (awaiting for CPU), one or more **running**, while privileged operations are conducted by the OS / HW on behalf of **blocked** processes
  - With several processes in RAM, it’s possible to maximize system resources utilization (e.g. CPU and HW resources/peripherals)
  - **Single-core / mono-processor** system: supports a single process running at any time; still, it allows switching the CPU among many processes; thus, by allowing **concurrent execution** of several processes, it supports **pseudo-parallelism**
  - **Multi-core / multi-processor system**: with several cores, **true parallel execution** of different processes becomes possible
- when a process leaves the CPU, the short-term scheduler must apply a **scheduling criteria**, to select a ready process to assign it the CPU



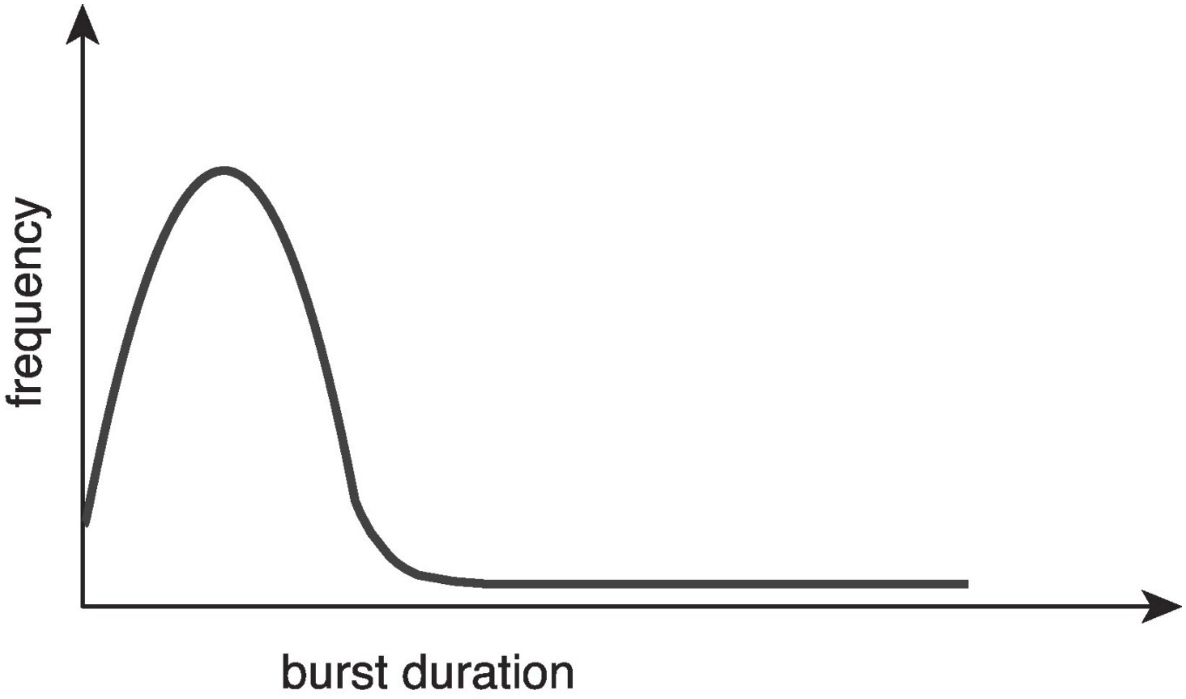
Basic Concepts

- **CPU–I/O Burst Cycle**
  - during its lifetime, processes alternate between **CPU bursts** (execution periods) and **I/O bursts** (periods in which await for I/O operations or other privileged operations)



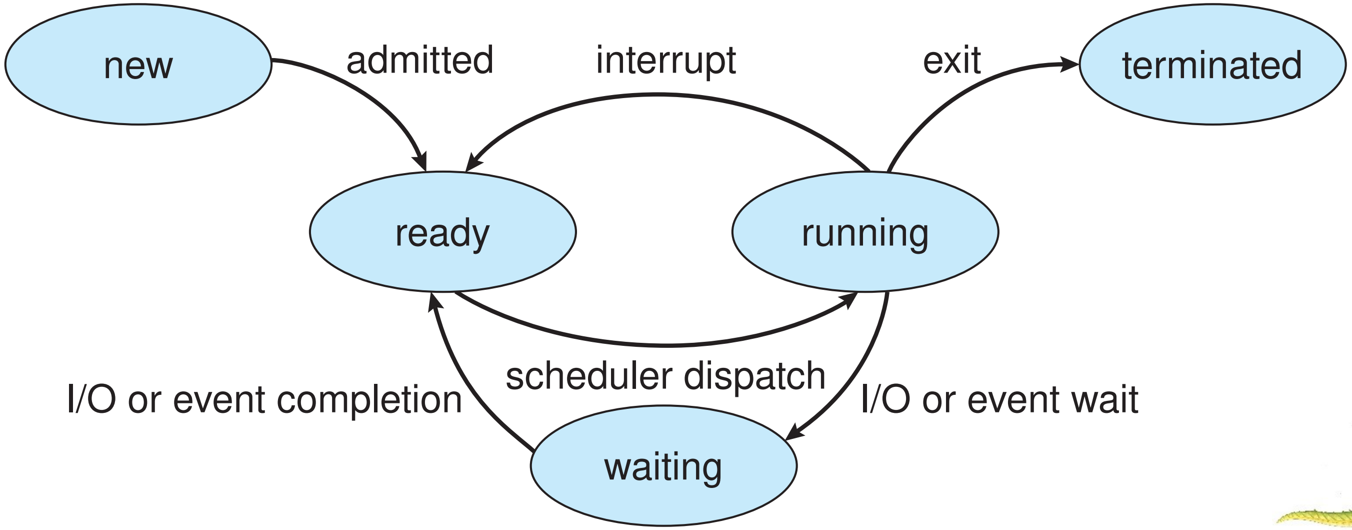
Histogram of CPU-burst Times

- **CPU–I/O Burst Cycle**
  - duration and frequency of CPU and I/O-bursts may vary
  - but short (large) CPU-bursts are generally the most (less) frequent
  - **CPU bound** process: few and large CPU-bursts
  - **IO bound** process: many and short CPU-bursts



CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state (frees up the CPU)
  2. Switches from running to ready state (frees up the CPU)
  3. Switches from waiting to ready (service requested by process A finalized; it is opportune to chose one process for execution – it may be A or not ...)
  4. Terminate (frees up the CPU)



CPU Scheduler

- Scheduling under situations 1 and 4 is **nonpreemptive (cooperative)**
  - processes release the CPU by their own will, and it is mandatory for the OS to chose another process to run, from those that are *ready*
  - note that a pure cooperative scheduling doesn’t need a timer circuit
  - examples: MS-DOS, Netware, Windows 3.x, Mac OS <= 9
- All other scheduling is **preemptive (enforced)**
  - examples: UNIX, VMS, BSD, Windows >= 95/NT, Linux, Mac OS X
  - implies special care when accessing shared data:
    - a process A shouldn’t release the CPU to a process B while A is accessing resources that can also be modified by process B
  - implies considering the possibility of preemption during the execution of a primitive while in kernel-mode; usually, preemption is honored only after current primitive finishes its execution or IO is invoked
  - consider different interrupts that share some resources: to prevent simultaneous access, interrupts are disabled before a critical section (provided it only has a few instructions) and re-enabled after

CPU Scheduler

- How often does context switch happens in a real system ?
- **Linux**: knowable using `vmstat` command and `/proc` file system
  - `vmstat` shows system wide information

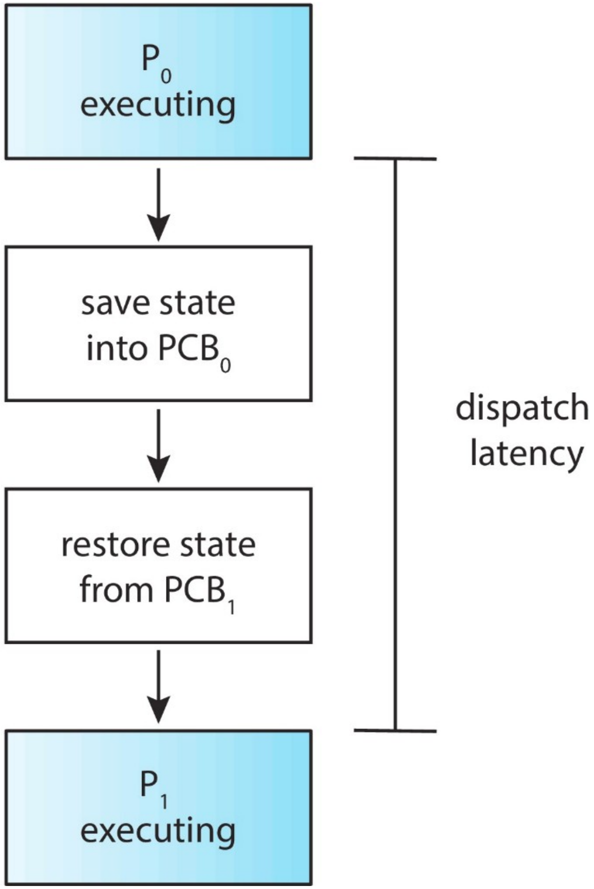
```
$ vmstat #with no parameters, shows average values since boot
... --system--  ------cpu-----
...   in   cs     us sy id wa st
...   184  428     2  1 97  0  0

# in = number of interrupts per second, including the clock
# cs = number of context switches per second
# us = time running non-kernel code (user time)
# sy = time running kernel code (system time)
# id = time idle; wa = time waiting for IO; st = (not used)
```

  - `/proc` file system shows process specific information

```
$ cat /proc/1/status
Name:   systemd
State:  S (sleeping)
Pid:    1
PPid:   0
voluntary_ctxt_switches: 94086
nonvoluntary_ctxt_switches: 662
```

Dispatcher

- The **Dispatcher** takes control of the CPU when a user-level process leaves the CPU; this involves:
  - switching to supervisor mode
  - switching context (save state in a PCB)
  - transfer control to the short-term scheduler
- The **Dispatcher** gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context (load state from a PCB)
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another; should be smallest possible

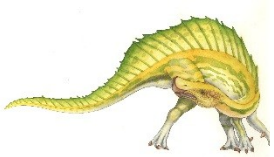


Theoretical Unit 3

3.2 Scheduling Criteria

Scheduling Criteria

- **Global Metrics** (system-wide metrics)
  - **CPU utilization** – keep the CPU as busy as possible
  - **Throughput** – number of processes that complete their execution per time unit; keep throughput as high as possible
    (short / long processes => high / low throughput)
- **Local Metrics** (process-specific metrics)
  - **Turnaround time** – total time to execute a process, starting at its submission and ending at its termination (time consumed on job queue + ready queue + IO queues + executing)
  - **Waiting time** – amount of time a process waits for CPU in the ready queue (directly affected by the scheduling criteria)
  - **Response time** – amount of time it takes from the moment a process is created until its first response produced (excludes time spent in output when in the blocked state); it is a better metric for time-sharing environments than turnaround time

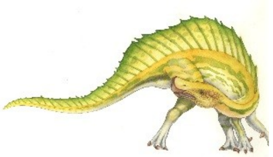


Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

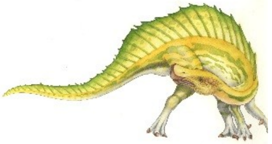
- Another criteria would be to optimize the average of these metrics (but that would hide extreme values)

- For interactive systems, it is better to minimize the variance of the response time: it becomes preferable a system with reasonable and predictable response time, than one with lower and unpredictable response time



Theoretical Unit 3

3.3 Scheduling Algorithms



First- Come, First-Served (FCFS) Scheduling

- **FCFS = First-Come, First-Served**

- processes are assigned CPU by their arrival order in the ready queue
- duration of the next CPU burst for each process is unknown beforehand
  - exceptions aside (*), processes use CPU as long as they want
- **non-preemptive**, inadequate to interactive systems
  - processes are not forced to share CPU time with others
- a process releases the CPU
  - **voluntarily**: ends, or invokes a privileged operation (it blocks and when the operation concludes it returns to the ready queue)
  - **(*) forcibly**: due to an interrupt arrival (e.g., notification of conclusion of an IO operation of another process), returning to the ready queue
- may result in high average waiting times
  - very sensitive to the arrival order of processes (see next)

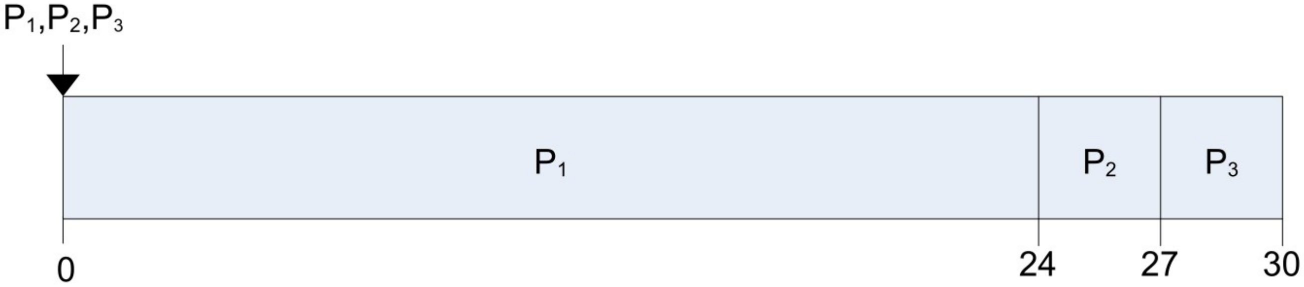
First- Come, First-Served (FCFS) Scheduling

Example: 3 processes in the ready queue, arrival order is P_1 , P_2 , P_3

Process	Arrival Order	Total CPU Time
P_1	1	24
P_2	2	3
P_3	3	3

(times assumed in **ms**; also in next examples)

- The Gantt chart for the schedule is:



- **Waiting times**: $P_1 = 0$ ms; $P_2 = 24$ ms; $P_3 = 27$ ms; average = **17 ms**
- **Turnaround times**: $P_1 = 24$ ms; $P_2 = 27$ ms; $P_3 = 30$ ms; average = **27 ms**

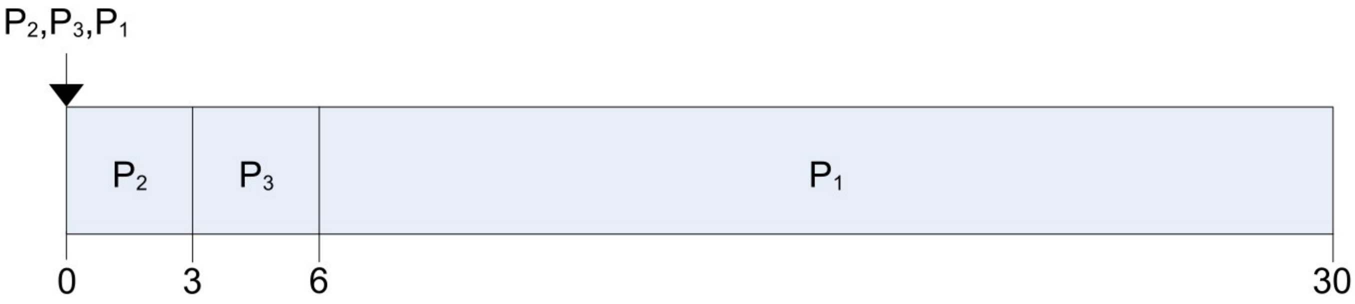
In this and in the next example: i) all processes are already in the queue at instant 0 (though they arrived in a certain order); ii) each process needs a single CPU burst, iii) the duration of the CPU burst is only known after the process leaves the CPU.

FCFS Scheduling (Cont.)

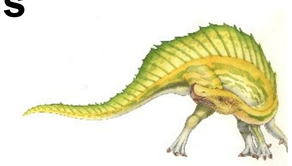
Example: same as previous example, but arrival order is P_2 , P_3 , P_1

Process	Arrival Order	Total CPU Time
P_1	3	24
P_2	1	3
P_3	2	3

- The Gantt chart for the schedule is:



- **Waiting times**: $P_1 = 6$ ms; $P_2 = 0$ ms; $P_3 = 3$ ms; **average = 3 ms**
- **Turnaround times**: $P_1 = 30$ ms; $P_2 = 3$ ms; $P_3 = 6$ ms; **average = 13 ms**
- Much better than previous case !

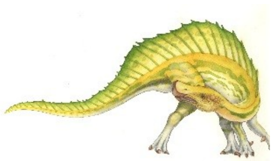


FCFS Scheduling (Cont.)

- **Convoy effect** - short processes behind long processes
  - generates low CPU and IO utilization
  - Consider one CPU-bound and many I/O-bound processes
  - When the CPU-bound process is executing, the I/O-bound processes will all end up queued in the ready queue; in this situation, the I/O resources are not being used
  - When the CPU-bound blocks in an I/O operation, the I/O-bound processes will move through the CPU very fast and will all block in I/O; this will overload I/O and leave the CPU idle
  - Eventually, the CPU-bound process will leave I/O, become ready and enter the CPU; after a while, I/O bound processes follow and will end up in the ready queue, thus closing this vicious cycle

Shortest-Job-First (SJF) Scheduling

- **SJF = Shortest-Job-First** (or **Shortest-Next-CPU-Burst**)
- assigns each process its (predicted) next CPU burst
  - but a process may end up consuming more/less
- use the predictions to schedule the process with the shortest time
  - untie using arrival order or other secondary criteria
- a processes releases the CPU
  - by the same reasons as in FCFS
  - if a process with a shorter prediction enters the ready queue (*)
- **(*) Preemptive SJF** or **Shortest-Remaining-Time-First (SRTF)**
  - the running process will be preempted by a ready process with a predicted CPU burst smaller than what's left of the running process
- SJF is optimal – ensures minimum average waiting time: by prioritizing short processes, the waiting time of short processes decreases more, in comparison to the growth of waiting time of long processes

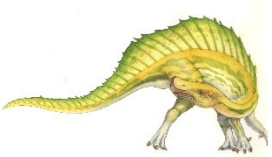


Determining Length of Next CPU Burst

- **initial prediction**: given by user (job submission) or defined by OS
- **next predictions**: only necessary if process returns to ready queue
  - based on the previous prediction and effective CPU bust

1. $t_n =$ actual length of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n .$

- **next predictions simplified** (examples and exercises):
  - $\tau_{n+1} = \tau_n - t_n$ if $t_n \le \tau_n$ (if the previous prediction was not fully consumed, the remaining time will be the next prediction)
  - $\tau_{n+1} = t_n$ if $t_n > \tau_n$ (if the previous prediction was not enough, the actual time consumed will be the next prediction)

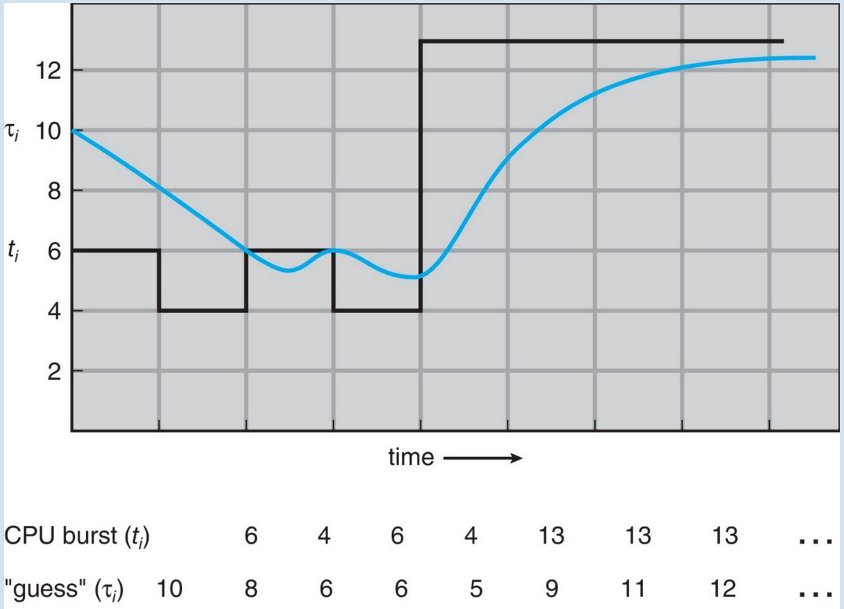


Moving Averaging

- $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\,\tau_n .$
- α =0
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- α =1
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \dots$$
$$+ (1 - \alpha)^j \alpha\, t_{n-j} + \dots$$
$$+ (1 - \alpha)^{n+1} \tau_0$$

- Since both α and (1 - α) are less than or equal to 1, each successive term has less weight than its predecessor



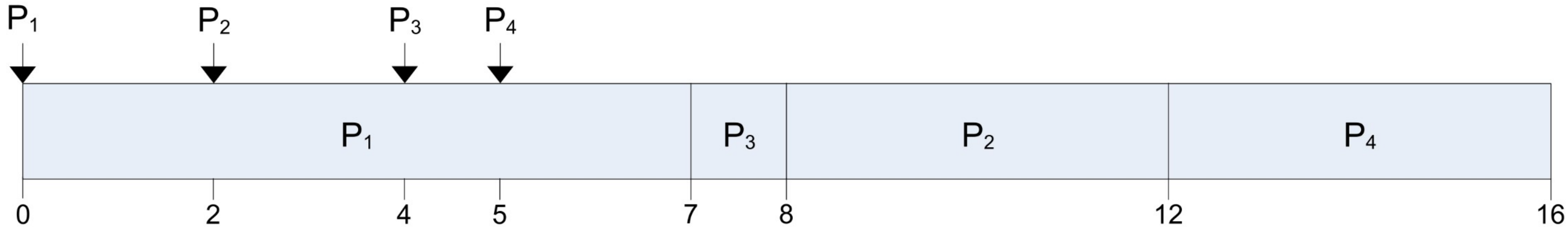
CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

example with **α** set to ½

Examples of SJF

- **Non-preemptive SJF**

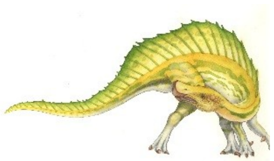
		CPU Bursts(ms)		
Process	Arrival Instants	Predicted	Effective	Total
P_1	0	7	7	7
P_2	2	4	4	4
P_3	4	1	1	1
P_4	5	4	4	4



- **Average waiting time** = (0 + (8-2) + (7-4) + (12-5)) / 4 = **4 ms** (FCFS: 4.75 ms)

- **Average turnar. time** = ((7-0)+(12-2)+(8-4)+(16-5)) / 4 = **8 ms** (FCFS: 8.75 ms)

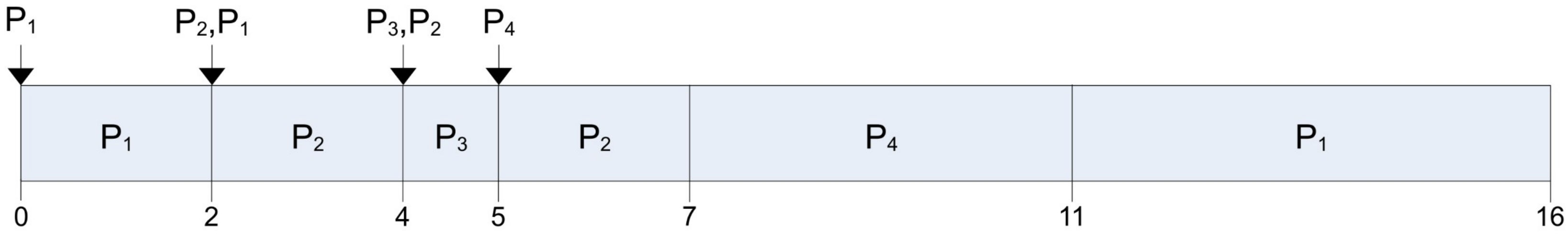
In this and in the next example: i) processes arrive to ready queue at different moments; ii) each process brings with it a single CPU burst prediction; iii) no process will exceed this initial prediction.



Examples of SJF

- **Preemptive SJF (SRTF)** with next prediction simplified:

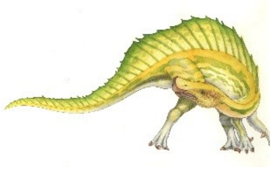
		CPU Bursts (ms)		
Process	Arrival Instants	Predicted	Effective	Total $= \sum$ Effective
P_1	0,2	7,5=7-2	2,5	2+5=7
P_2	2,4	4,2=4-2	2,2	2+2=4
P_3	4	1	1	1
P_4	5	4	4	4



- **Average waiting time** = ((0+11-2)+(0+5-4) +0+(7-5)) / 4 = **3 ms**

- **Average turnaround time** = ((16-0) + (7-2) + (5-4) + (11-5)) / 4 = **7 ms**

- A small improvement over the previous scenario.

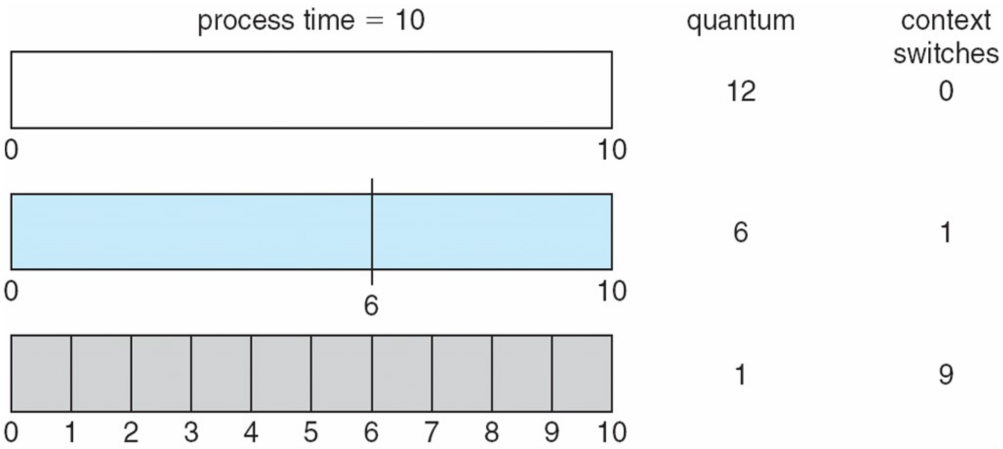


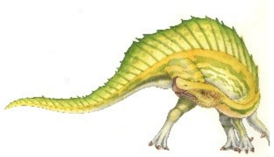
Round Robin (RR)

- RR is similar to FCFS but with a limit to the CPU burst length
- the ready queue is managed in FIFO order (like in FCFS)
- each scheduled process gets a small unit of CPU time (**quantum** or ***q***): 10-100 ms; the *timer* generates an interrupt every *q* ms
- A process releases the CPU
  - by the same reasons as in FCFS
  - due to the *timer* interrupt, if it consumes its entire quantum
- it is a **preemptive** algorithm, for **interactive (time-sharing)** systems
  - processes are forced to share CPU time among them
  - CPU bursts have typically a short length
- RR translates in lower response times (benefits **interactivity**, but has higher turnaround times than other scheduling methods)

Round Robin (RR)

- with n process, each gets 1/*n* of the CPU time, in chunks of at most *q* time units at once; no process waits more than (*n*-1)*q* time units (it may wait less, once quantums may not be fully exhausted)
- **quantum vs context switch time**
  - *q* too large => RR decays to FIFO
  - *q* too small => too much overhead from context switch
  - *q* must be sufficiently large with respect to context switch
    - *q* usually 10ms to 100ms, context switch < 10 usec
    - 80% of CPU bursts should be shorter than *q*

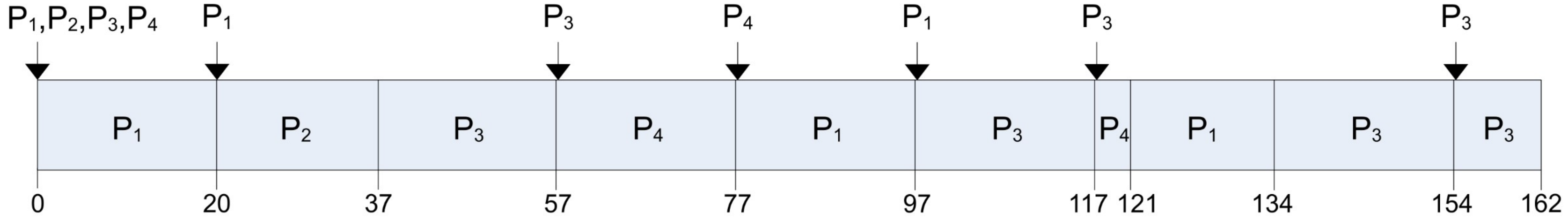




Example of RR with Quantum = 20

All processes in the ready queue at instant 0, but with an arrival order.
Execution may involve several CPU bursts, all of maximum duration ***q***.
Effective total CPU time knowable only when the process terminates.

Process	Arrival Instants	CPU Bursts (ms)	Total CPU Time (ms)
P_1	0, 20, 97	20, 20, 13	$\sum = 53$
P_2	0	17	$\sum = 17$
P_3	0, 57, 117, 154	20, 20, 20, 8	$\sum = 68$
P_4	0, 77	20, 4	$\sum = 24$



Note that when a single process remains (P3) it still consumes CPU time in slices.

Average waiting time: ((0+77-20+121-97)+(20-0)+(37-0+97-57+134-117)+(57-0+117-77))/4 = **73 ms**

Average turnaround time: ((134-0)+(37-0)+(162-0)+(121-0))/4 = **113.5 ms**

Average waiting time with non-preemptive SJF:

((17+24) + 0 + (17+24+53) + 17)/4 = **38 ms** (lower waiting time; what about response time ?)

Priority Scheduling

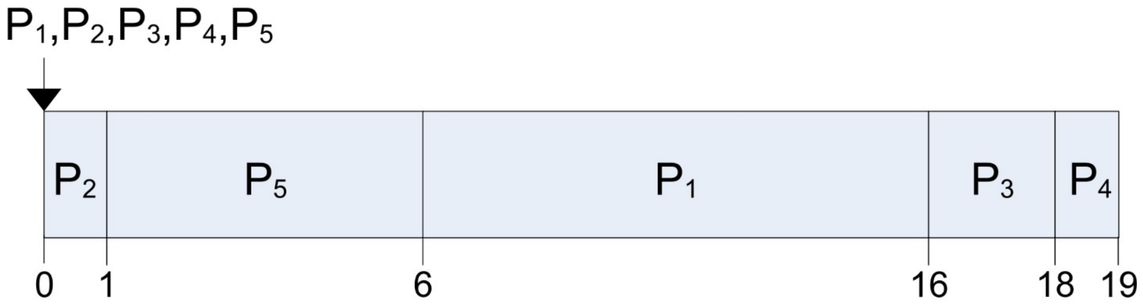
- A priority number is associated with each process
  - **internal priorities**: OS defined (deadlines, RAM limits, IO/CPU-bound nature); **external priorities**: political/economical factors
- The CPU is given to the process with the highest priority
  - untie with arrival order (or other supplied criteria)
- FCFS is priority scheduling by arrival order; SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Preemptive and Non-Preemptive variants
- A process leaves the CPU:
  - as in SJF, but considering the generic priority being used

- Problem: **Starvation** – low priority processes may never execute
- Solution: **Aging** – as time pass, increase the priority of the process

Example of Priority Scheduling

Non-preemptive. All processes in the ready queue at instant 0. One CPU burst per process. CPU burst duration known only when a process ends.

Process	Priority	Total CPU Time (ms)
P_1	3	10
P_2	1	1
P_3	4	2
P_4	5	1
P_5	2	5



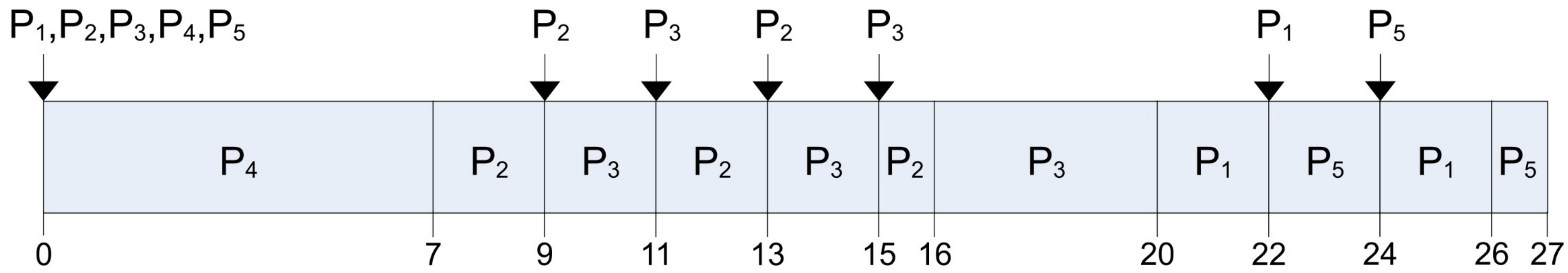
Average waiting time = (6 + 0 + 16 + 18 + 1) / 5 = **8.2 ms**

Average turnaround time = (16 + 1 + 18 + 19 + 6) / 5 = **12 ms**

Priority Scheduling combined with RR

- Processes of higher priority executed without preemption
- Processes of equal priority executed with preemption under RR
- **Example** (q=2; all processes already in the ready queue at instant 0):

Process	Arrival Instants	Priority	CPU Bursts (ms)	Total CPU Time (ms)
P_1	0, 22	3	2, 2	$\sum=4$
P_2	0, 9, 13	2	2, 2, 1	$\sum=5$
P_3	0, 11, 15	2	2, 2, 4	$\sum=8$
P_4	0	1	7	$\sum=7$
P_5	0, 24	3	2, 1	$\sum=3$



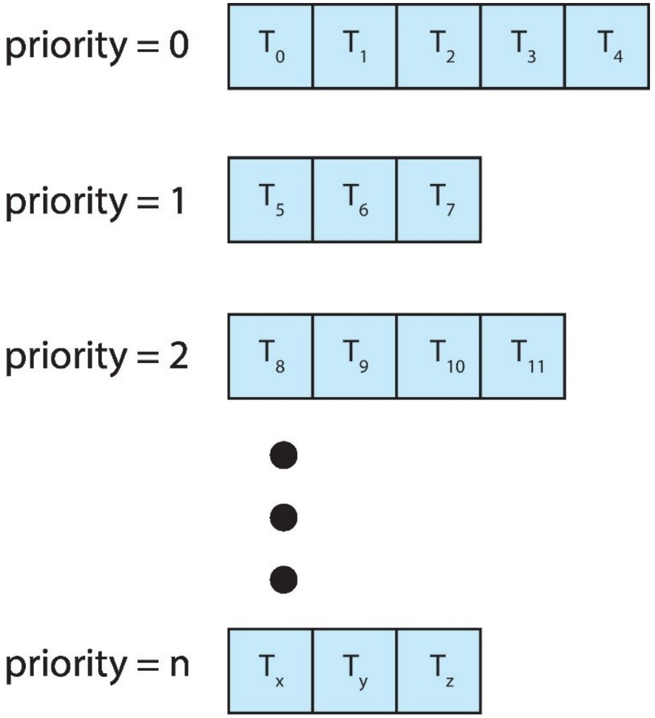
Average waiting time = ((20+2)+(7+2+2)+(9+2+1)+0+(22+2))/5 = **13.8 ms**

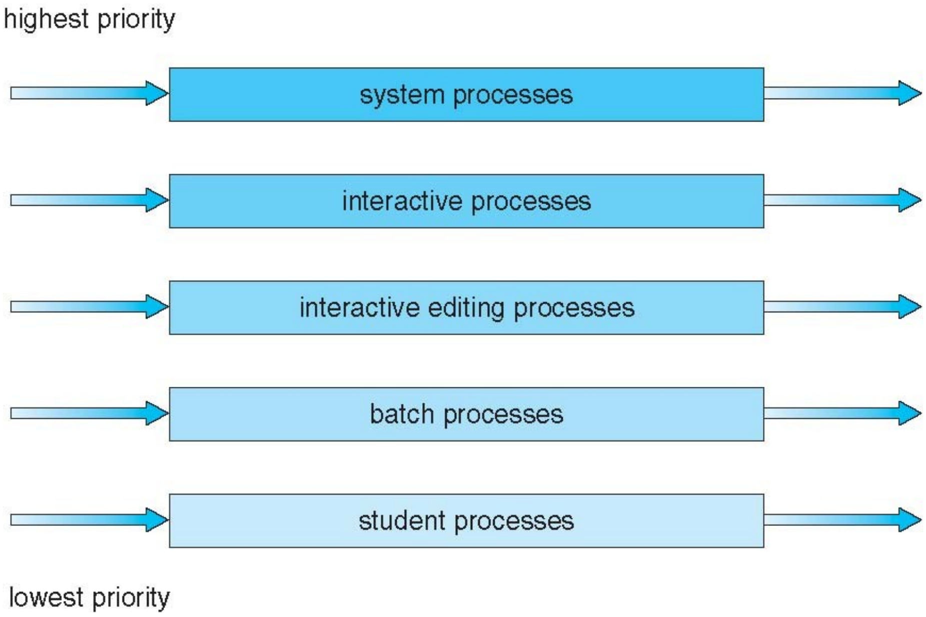
Average turnaround time = (26+16+20+7+27)/5 = **19.2 ms**

Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (for interactive processes)
  - **background** (for batch processes)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - foreground – **RR**
  - background – **FCFS**
- Scheduling must be done between the queues:
  - **Fixed priority scheduling**: serve all from foreground, and only then serve from background; possibility of **starvation** …
  - **Time slice scheduling**: each queue gets a certain amount / share of CPU time which it can schedule among its processes (i.e., 80% to foreground in RR, 20% to background in FCFS)

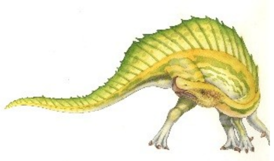
Multilevel Queue Scheduling



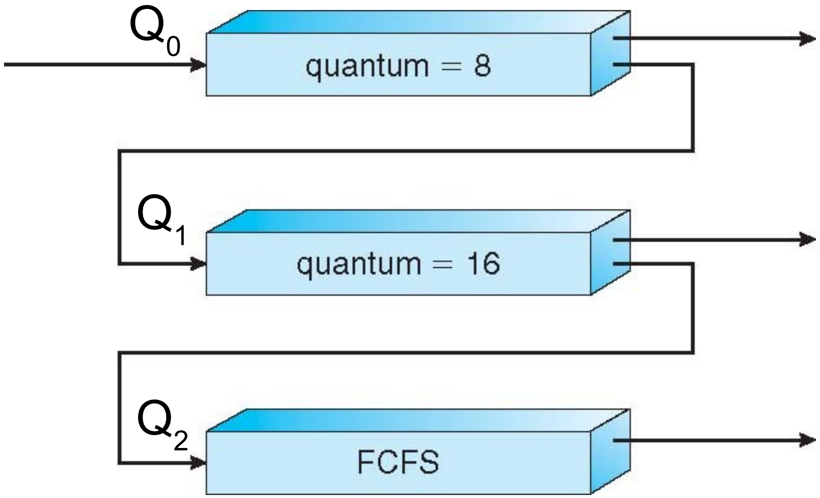


Multilevel Feedback Queue

- A process can **move between the various queues**
- Queues with **higher priority** will be for **interactive / IO-bound** processes
  - these processes quickly move out from the CPU
- **CPU-bound** processes will be demoted to **lower-priority queues**
  - **aging**: promote old CPU-bound processes to higher priority queues
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine which queue a process will enter first
  - method used to determine when to upgrade /demote a process



Example of Multilevel Feedback Queue



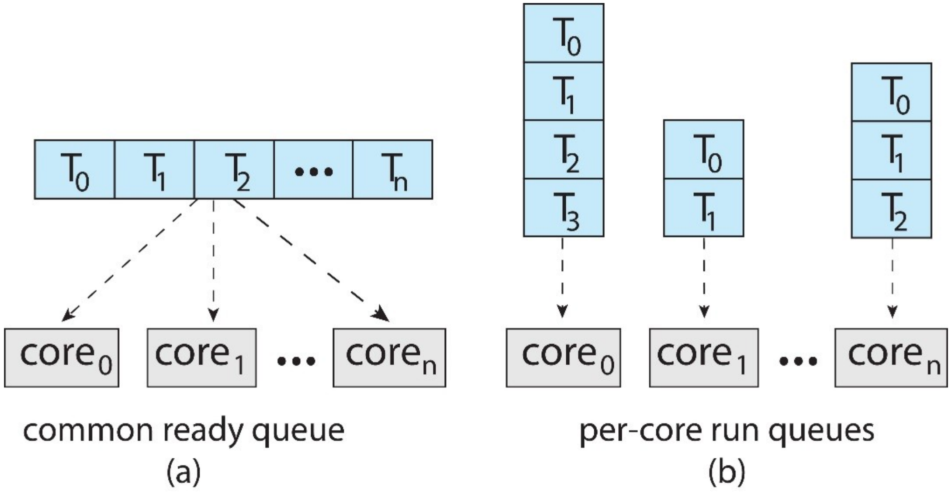
- Three queues:
  - Q_0 – FCFS with time quantum 8 milliseconds
  - Q_1 – FCFS with time quantum 16 milliseconds
  - Q_2 – regular FCFS (no time limit)
- Scheduling
  - A new job always enters Q_0
    - When it gains CPU, a job receives 8 ms; If it doesn't finish in 8 ms, it's moved to Q_1
  - If Q_0 is empty, Q_1 jobs are scheduled
    - When it gains CPU, a Q_1 job receives 16 ms; If it doesn't finish in 16 ms, it's moved to Q_2
  - If Q_0 and Q_1 are empty, Q_2 jobs are scheduled in a pure FCFS fashion, executing until they finish
  - Problem: **starvation** - processes in a queue scheduled only if higher-priority queues are empty
  - Solution: process **aging** (move to the upper queue)
- Scheduling variants with Q0 and Q1 under RR
  - a) a process is demoted only if it returns to the current queue >n times; b) a process is promoted to a queue with a smaller quantum if it doesn't fully use the quantum of the current queue

Theoretical Unit 3

3.4 Multiple Processor Scheduling

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
  - example: processes that need a specific IO device must run on a CPU with whom the device shares a communication bus
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, thus only one processor performs scheduling
- **Symmetric multiprocessing** (**SMP**) – processors do self-scheduling
  - each CPU has its own private queue of ready processes
  - or, all processes share a common ready queue (need to avoid i) double scheduling of the same process, ii) no scheduling at all)

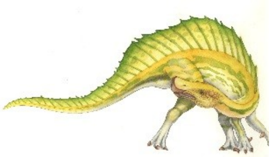


Multiple-Processor Scheduling – Load Balancing

- With SMP, there is a need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
  - needed with separate scheduling queues (one per CPU)

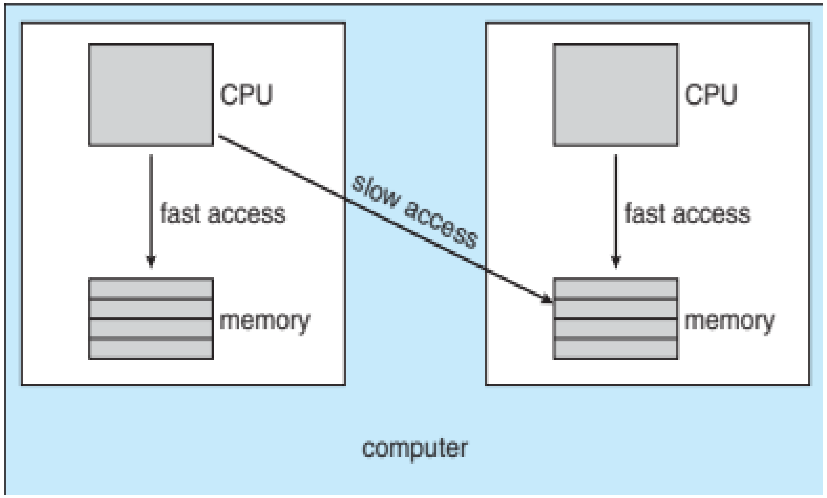
- **Push migration** – periodic task checks load on each processor, and pushes tasks from overloaded CPU to other CPUs
- **Pull migration** – idle CPUs pulls waiting task from busy CPUs
- Both techniques can be used combined
  - Example (Linux): push migration every 200ms, plus pull migration anytime a scheduling queue becomes empty

- Load Balancing collides with CPU affinity …



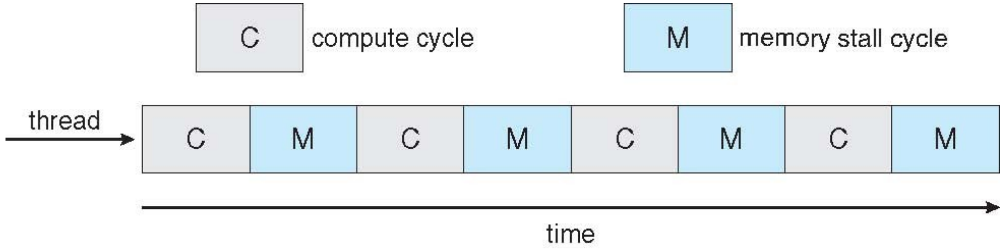
Multiple-Processor Scheduling – CPU Affinity

- **Processor affinity** – process has affinity for processor where it runs
  - goal: avoid process migration between CPUs (cache invalidation)
  - **soft affinity / hard affinity:** tolerate / prohibit process migration
  - variations including **processor sets (Solaris):** processes may migrate between CPUs of the same set (which share some caches)
  - affinity is also influenced by the memory architecture of the system
  - example: with NUMA (Non-Uniform Memory Access), RAM sticks close (local) to a CPU are accessed faster; thus, code and data should reside on the local RAM of the CPU where the program runs

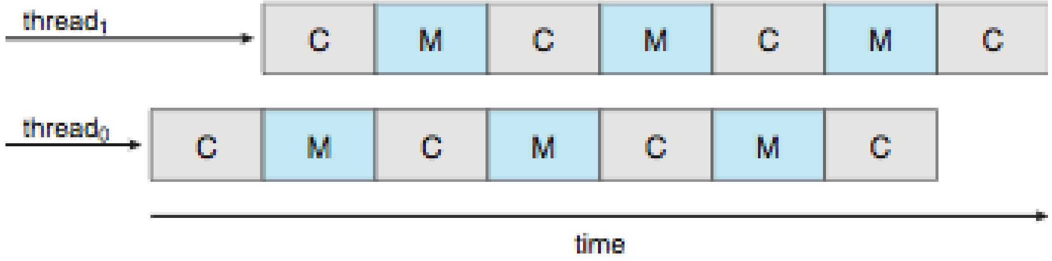


Multiple-Processor Scheduling – Hyper-Threading

- traditional SMPs: single-core CPU, with a single-thread per core
- multi-core: place multiple processor cores on same physical chip
  - sub-utilization with a single-thread per core, due to memory stalls (awaiting for instructions/data from RAM, after a cache miss)

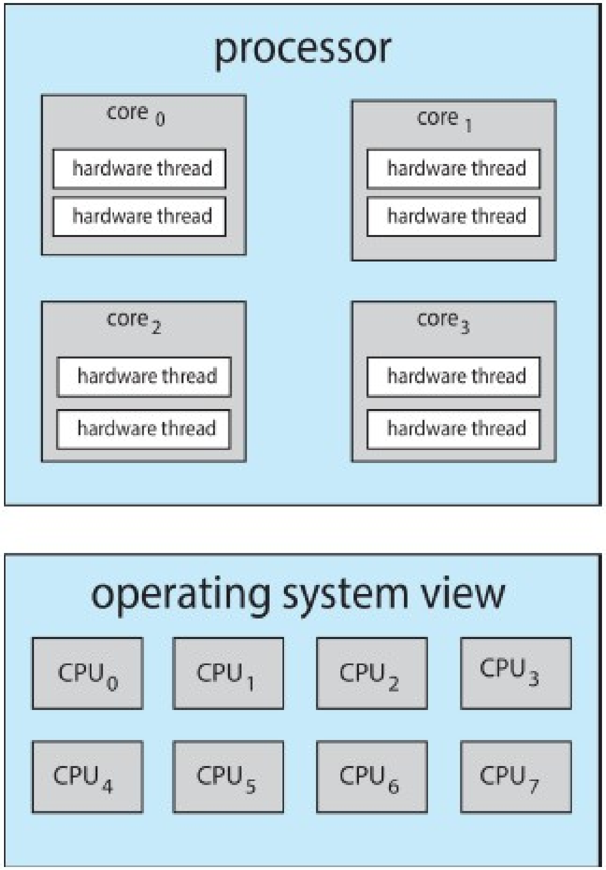


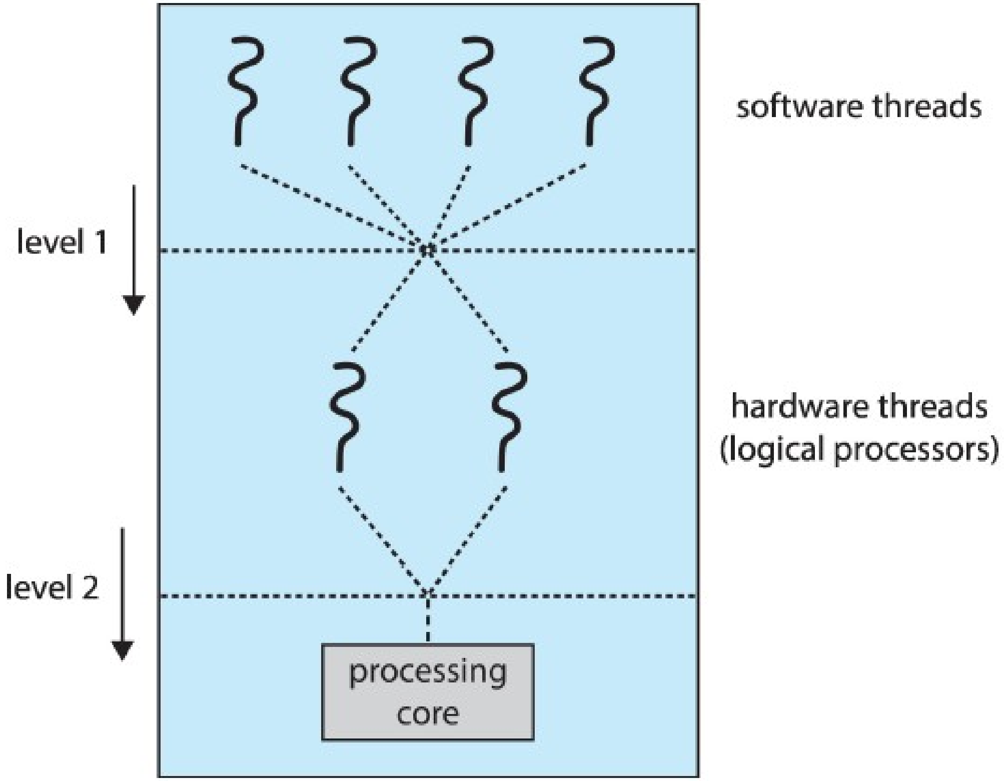
  - better utilization with several hardware threads per core (hyper-threading) (make progress on another thread while memory retrieve happens)



  - each hardware-thread is regarded by the OS as a logical-core, to which assigns a process/software-thread under a scheduling algorithm; but each physical core chooses which hardware-thread to run at a time

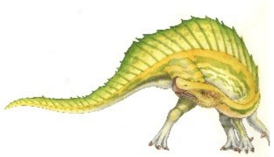
Multiple-Processor Scheduling – Hyper-Threading





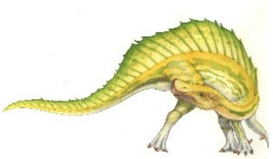
Virtualization and Scheduling

- Virtualization software schedules multiple guests onto one or more CPU
- Therefore, even if the underlying HW has only one CPU, the virtualization layer will multiplex this single CPU like if it were a real SMP system
- Moreover, each guest will be doing its own scheduling
  - Not knowing it doesn't own the CPUs
    - not entirely true anymore (e.g., paravirtualization)
  - Can result in poor response time
    - guests scheduling decisions may counteract hypervisor's
  - Can effect time-of-day clocks in guests
    - clock drifts due to overloading of the real CPUs
- To sum it up:
  - virtualization and guests scheduling are often in conflict



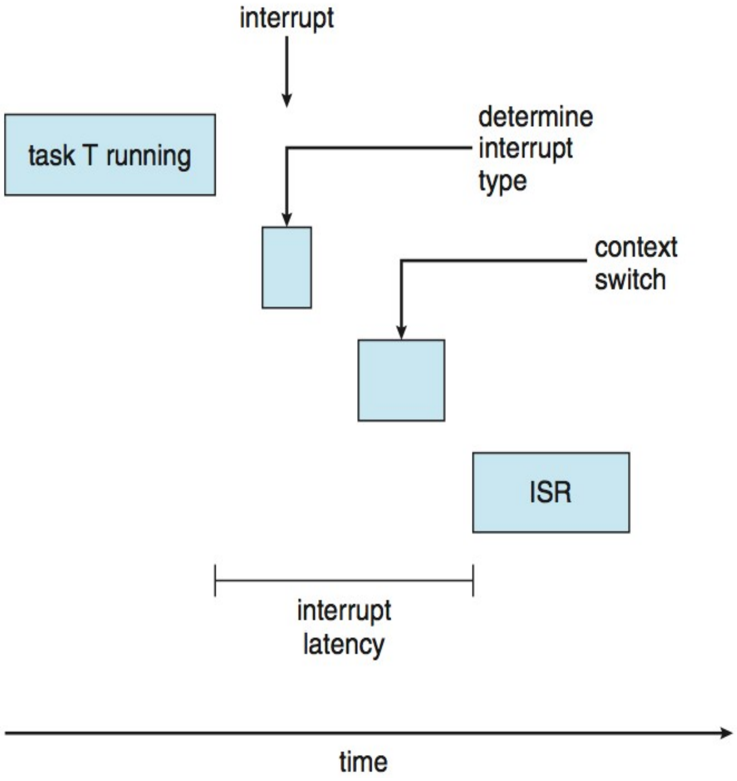
Theoretical Unit 3

3.5 Real-Time CPU Scheduling



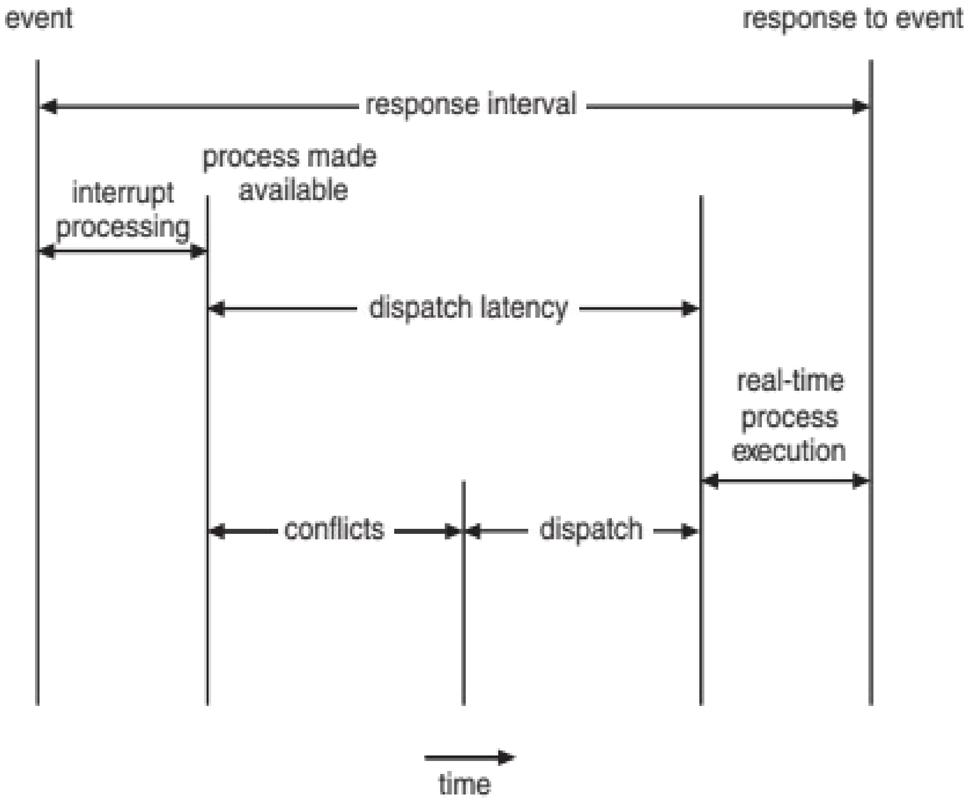
Real-Time CPU Scheduling

- Can present obvious challenges
- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline
- Two types of latencies affect performance
  1. **Interrupt latency** – time from arrival of interrupt to start of routine that services interrupt
  2. **Dispatch latency** – time for schedule to take current process off CPU and switch to another



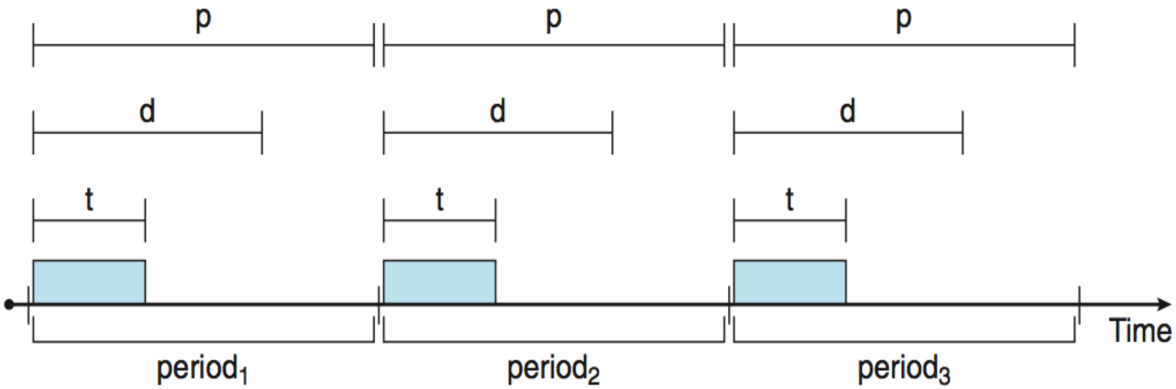
Real-Time CPU Scheduling (Cont.)

- Conflict phase of dispatch latency:
  1. Preemption of any process running in kernel mode
  2. Release by low-priority process of resources needed by high-priority processes



Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing time *t*, deadline *d,* period *p*
  - 0 ≤ *t* ≤ *d* ≤ *p*
  - **Rate** of periodic task is 1/*p*



Appendix - Exercises

Exercise 3.1 (Scheduling Algorithms) (1/7)

Consider the following processes and scheduling attributes:

Process	Arrival Instant	Priority	Total CPU Time (ms)
P_1	0	2	5
P_2	1	1	4
P_3	2	3	2

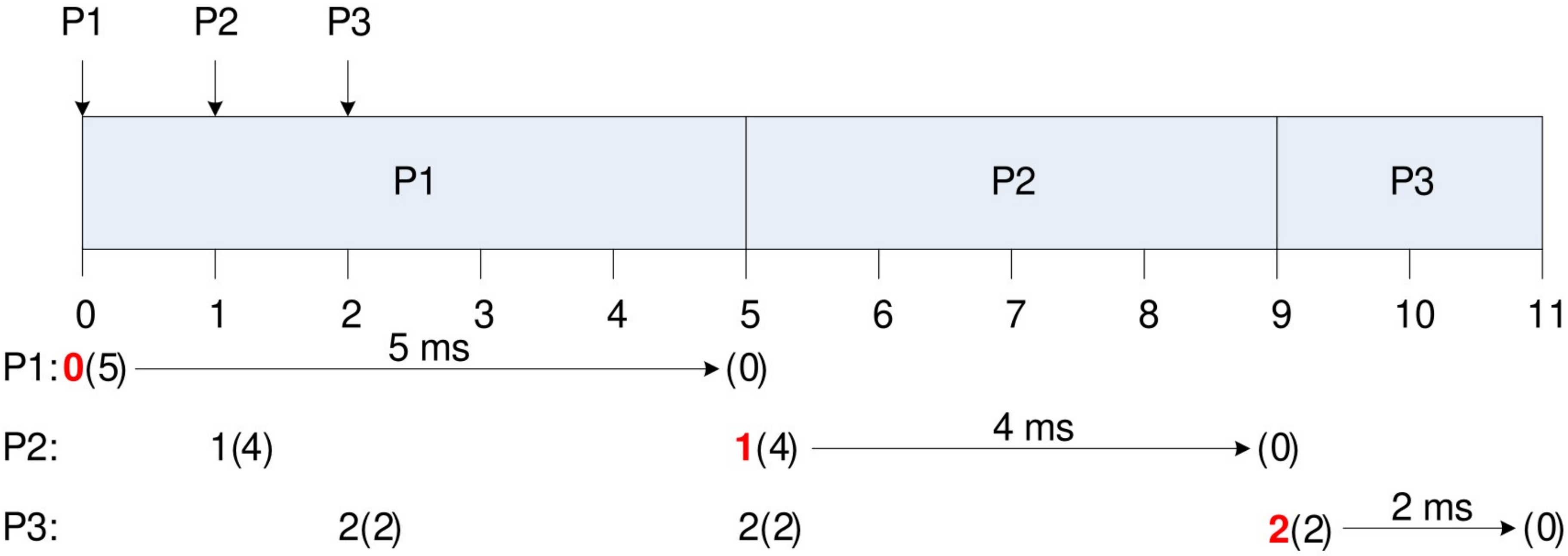
a) Draw Gantt charts, showing the execution of the above processes using the scheduling policies FCFS, SJF non-preemptive, SJF preemptive, Priorities non-preemptive, Priorities preemptive, and RR (q=2); for SJF policies assume the initial predicted CPU Burst to be equal to the Total CPU Time; for Priorities policies assume that a lower Priority index implies a stronger priority.

b) Determine the average waiting time and average turnaround time for each one of the previous scenarios.

Exercise 3.1 (Scheduling Algorithms) (2/7)

- FCFS

Process	**Arrival Instant**	Total CPU Time (ms)
P_1	**0**	5
P_2	**1**	4
P_3	**2**	2

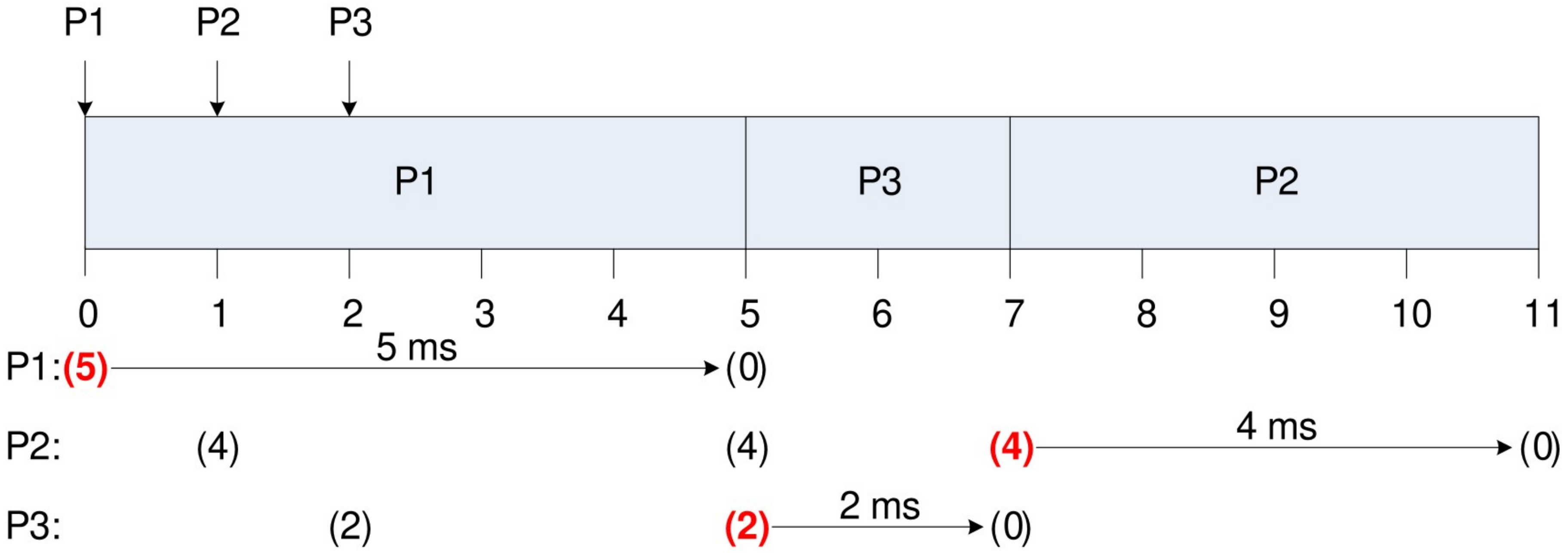


- average waiting time: (0 + (5-1) + (9-2)) / 3 = 11/3 = 3.66(6) ms
- average turnaround time: [(5-0) + (9-1) + (11-2)] / 3 = 22/3 = 7.33(3) ms

Exercise 3.1 (Scheduling Algorithms) (3/7)

- SJF non-preemptive

		CPU Bursts (ms)		
Process	Arrival Instant	Predicted	Effective	Total
P_1	0	**5**	5	5
P_2	1	**4**	4	4
P_3	2	**2**	2	2

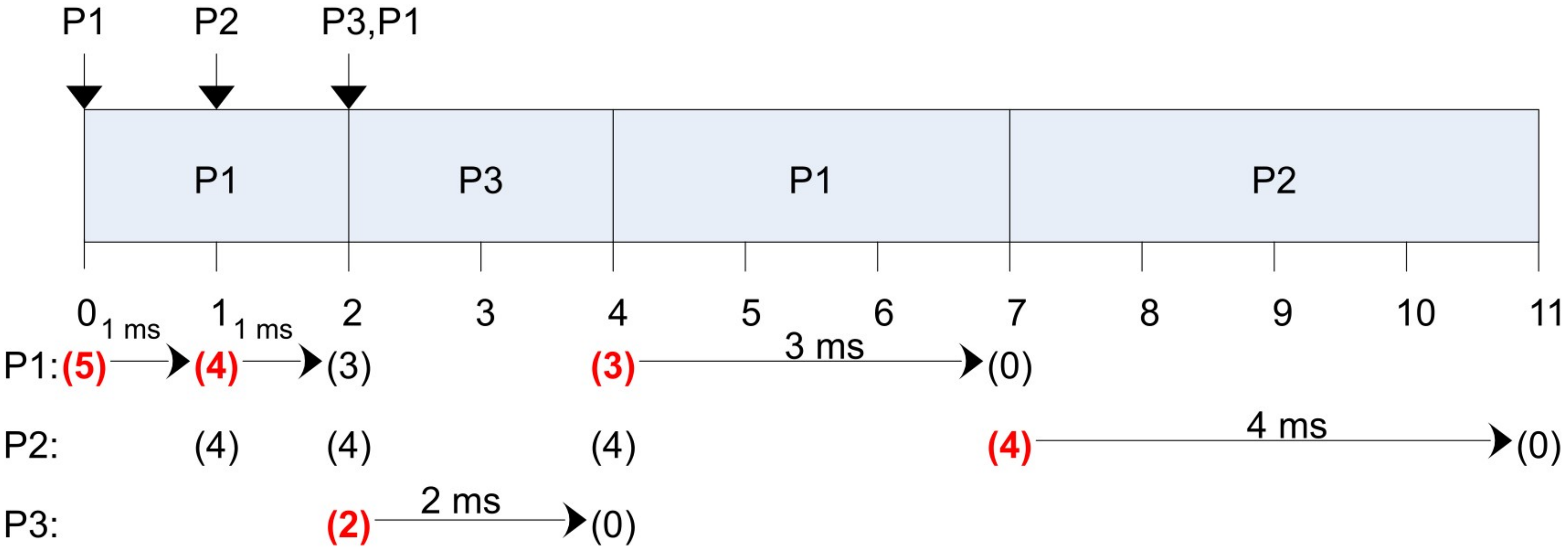


- average waiting time: $(0 + (7\text{-}1) + (5\text{-}2)) / 3 = 9/3 = 3$ ms
- average turnaround time: $[\ (5\text{-}0) + (11\text{-}1) + (7\text{-}2)\ ] / 3 = 20/3 = 6.66(6)$ ms

Exercise 3.1 (Scheduling Algorithms) (4/7)

- SJF preemptive

		CPU Bursts (ms)		
Process	Arrival Instants	**Predicted**	Effective	Total
P_1	0, 2	**5,4=5**-1(*),**3=4**-1(**)	1(*),1(**),3	1+1+3=5
P_2	1	**4**	4	4
P_3	2	**2**	2	2

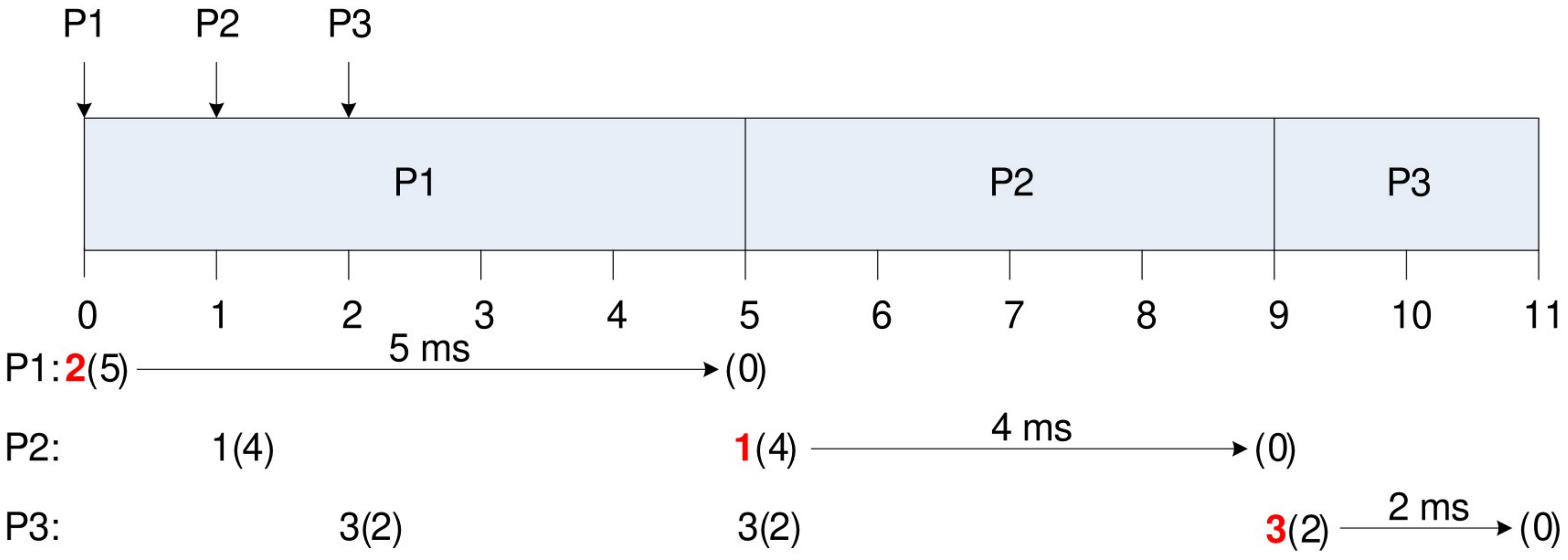


- average waiting time: ((4-2) + (7-1) + 0) / 3 = 8/3 = 2.66(6) ms
- average turnaround time: [(7-0) + (11-1) + (4-2)] / 3 = 19/3 = 6.33(3) ms

Exercise 3.1 (Scheduling Algorithms) (5/7)

- Priorities (generic) non-preemptive

Process	Arrival Instant	Priority	Total CPU Time (ms)
P_1	0	2	5
P_2	1	1	4
P_3	2	3	2



P1: 2(5) → 5 ms → (0)

P2: 1(4) 1(4) → 4 ms → (0)

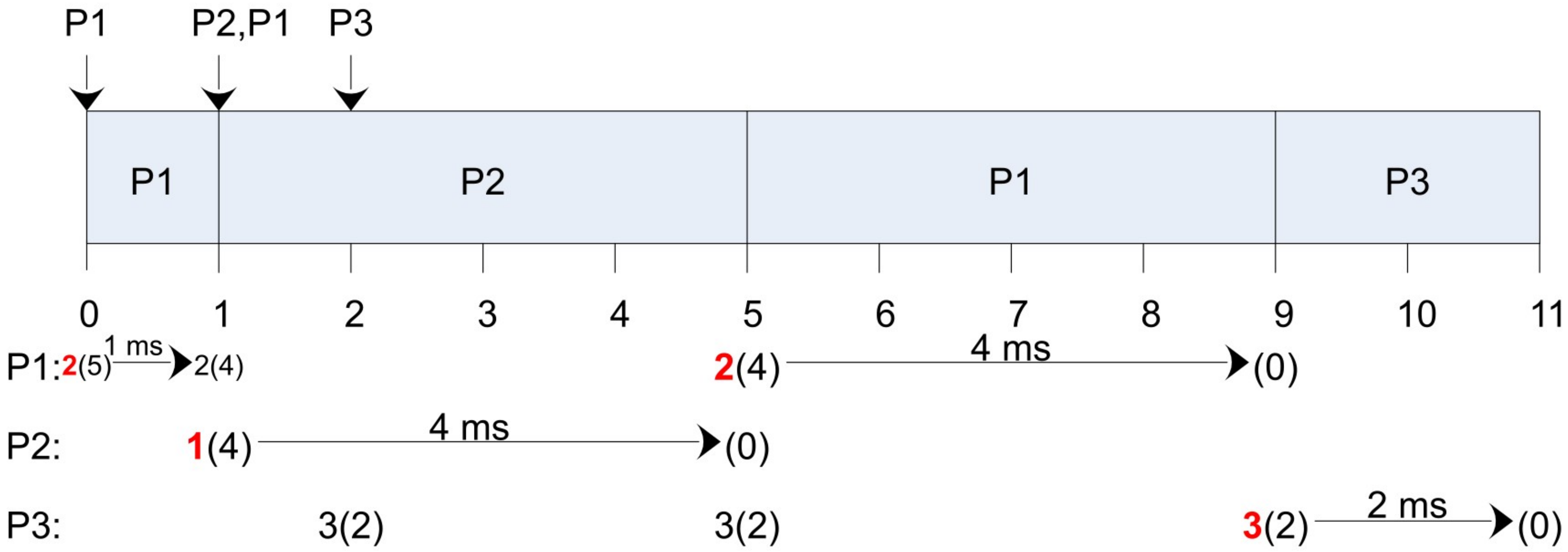
P3: 3(2) 3(2) 3(2) → 2 ms → (0)

- average waiting time: (0 + (5-1) + (9-2)) / 3 = 11/3 = 3.66(6) ms
- average turnaround time: [(5-0) + (9-1) + (11-2)] / 3 = 22/3 = 7.33(3) ms

Exercise 3.1 (Scheduling Algorithms) (6/7)

- Priorities (generic) preemptive

Process	Arrival Instants	**Priority**	CPU Bursts (ms)	Total CPU Time (ms)
P_1	0,1	**2**	1,4	5
P_2	1	**1**	4	4
P_3	2	**3**	2	2

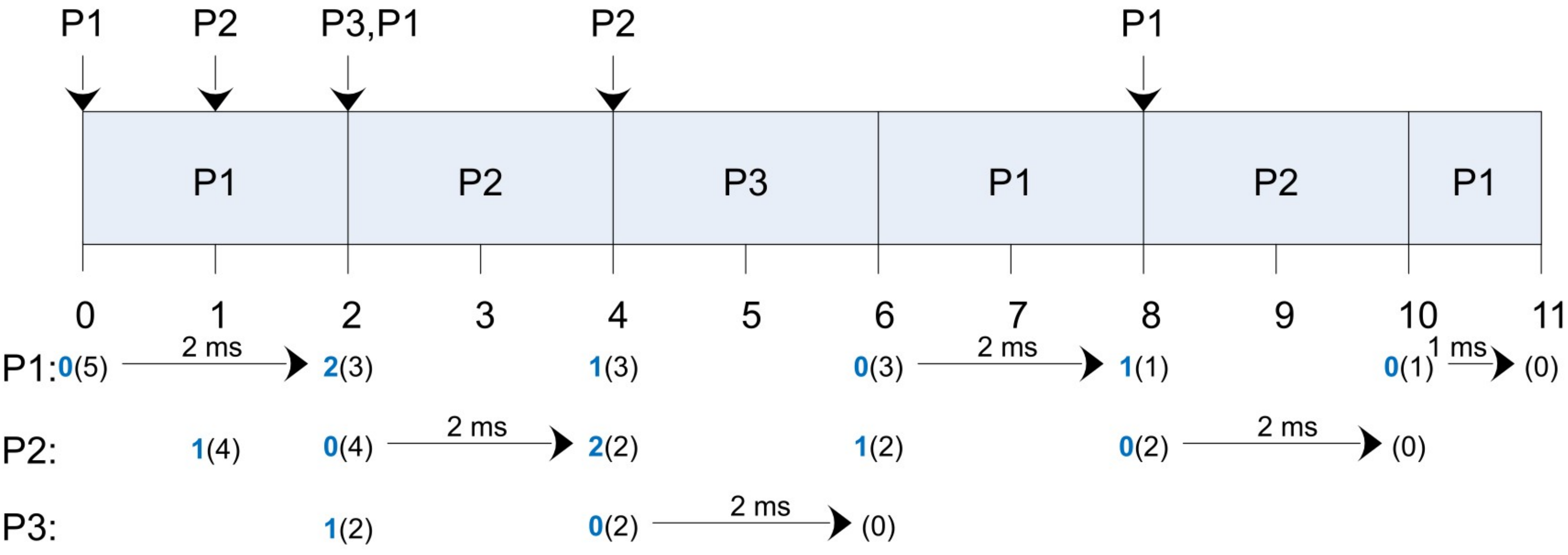


- average waiting time: [(0 + (5-1)) + 0 + (9-2)] / 3 = 11/3 = 3.66(6) ms
- average turnaround time: [(9-0) + (5-1) + (11-2)] / 3 = 22/3 = 7.33(3) ms

Exercise 3.1 (Scheduling Algorithms) (7/7)

- RR (**q=2**)

Process	**Arrival Instants**	CPU Bursts (ms)	Total CPU Time (ms)
P_1	0,2,8	2,2,1	$\sum=5$
P_2	1,4	2,2	$\sum=4$
P_3	2	2	$\sum=2$



P1 P2 P3,P1 P2 P1

P1 | P2 | P3 | P1 | P2 | P1

0 1 2 3 4 5 6 7 8 9 10 11

P1: **0**(5) 2 ms → **2**(3) **1**(3) **0**(3) 2 ms → **1**(1) **0**(1) 1 ms → (0)

P2: **1**(4) **0**(4) 2 ms → **2**(2) **1**(2) **0**(2) 2 ms → (0)

P3: **1**(2) **0**(2) 2 ms → (0)

- **blue numbers**: position in the ready queue (**0** means selected to execute)
- note that at instant 2, P3 (newly arrived) enters the queue ahead of P1
- avg. waiting time: [(0+6-2+10-8) + (2-1+8-4) + (4-2)] / 3 = 13/3 = 4.33(3) ms
- avg. turnaround time: [(11-0) + (10-1) + (6-2)] / 3 = 24/3 = 8 ms

Exercise 3.2 (Scheduling Algorithms) (1/2)

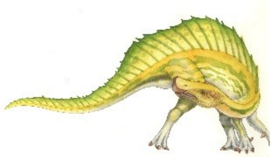
Consider the following processes and scheduling attributes:

Process	Arrival Instant	Priority	Total CPU Time (ms)
P_1	0	4	6
P_2	1	3	4
P_3	2	2	2
P_4	3	1	1

a) Draw Gantt charts, showing the execution of the above processes using the scheduling policies FCFS, SJF non-preemptive, SJF preemptive, Priorities non-preemptive, Priorities preemptive, and RR (q=2); for SJF policies assume the initial predicted CPU Burst to be equal to the Total CPU Time; for Priorities policies assume that a lower Priority index implies a stronger priority,

b) Determine the average waiting time and average turnaround time for each one of the previous scenarios.

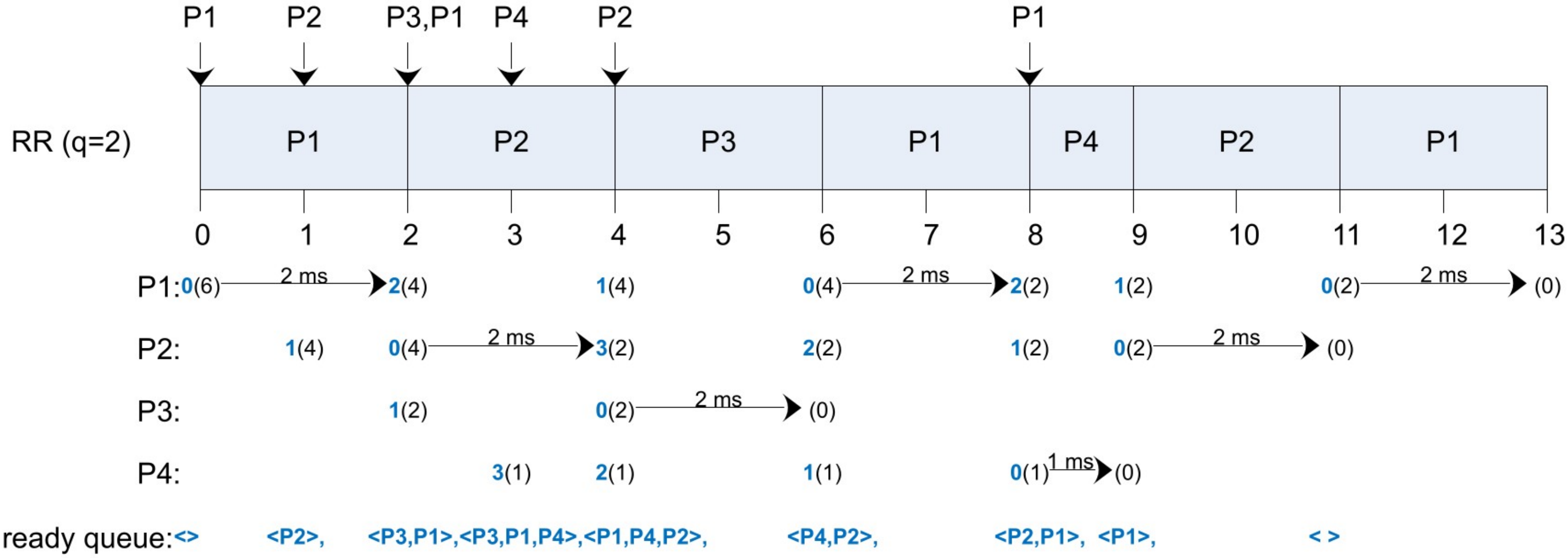
Note: in the next slide the solution for RR is provided, once it represents a special case; for the other algorithms, use the solutions of exercise 4.1 as a reference.



Exercise 3.2 (Scheduling Algorithms) (2/2)

- for RR, there is now the possibility of old processes (e.g., P_1) returning to the ready queue before new processes arrive (e.g., P_4)

Process	Arrival Instants	CPU Bursts (ms)	Total CPU Time (ms)
P_1	0,6,11	2,2,2	$\sum=6$
P_2	1,4	2,2	$\sum=4$
P_3	2	2	$\sum=2$
P_4	3	1	$\sum=1$


P1
P2
P3,P1
P4
P2
P1
RR (q=2)
P1
P2
P3
P1
P4
P2
P1
0
1
2
3
4
5
6
7
8
9
10
11
12
13
P1: 0(6) 2 ms 2(4) 1(4) 0(4) 2 ms 2(2) 1(2) 0(2) 2 ms (0)
P2: 1(4) 0(4) 2 ms 3(2) 2(2) 1(2) 0(2) 2 ms (0)
P3: 1(2) 0(2) 2 ms (0)
P4: 3(1) 2(1) 1(1) 0(1) 1 ms (0)
ready queue: <> <P2>, <P3,P1>,<P3,P1,P4>,<P1,P4,P2>, <P4,P2>, <P2,P1>, <P1>, < >

Exercise 3.3 (Scheduling Algorithms) (1/1)

Consider the following processes and scheduling attributes:

Process	Arrival Instant	Priority	CPU Time for 1st Output (ms)	Total CPU Time (ms)
P_1	0	4	1	5
P_2	1	3	2	4
P_3	2	1	2	3
P_4	3	2	1	2

a) Draw Gantt charts, showing the execution of the above processes using the scheduling policies FCFS, SJF non-preemptive, SJF preemptive, Priorities non-preemptive, Priorities preemptive, and RR (q=2); for SJF policies assume the initial predicted CPU Burst to be equal to the Total CPU Time; for Priorities policies assume that a lower Priority index implies a stronger priority.

b) Determine the average waiting time, the average turnaround time, and the average response time for each of the previous scenarios.

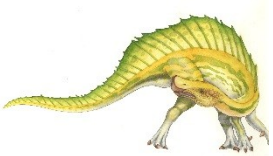
Note: recall that an output request makes a process to block in a I/O queue; assume that the process returns to the ready queue (to its tail, if using FCFS or RR) after its successor in the CPU was already chosen, but before the next scheduling decision.

Theoretical Unit 3

References:

- "Operating System Concepts, 10th Ed.", Silberschatz & Galvin, Addison-Wesley, 2018: Chapter 5

- “The Fancy Algorithms That Make Your Computer Feel Smoother" (https://www.youtube.com/watch?v=O2tV9q6784k)



End of Theoretical Unit 3