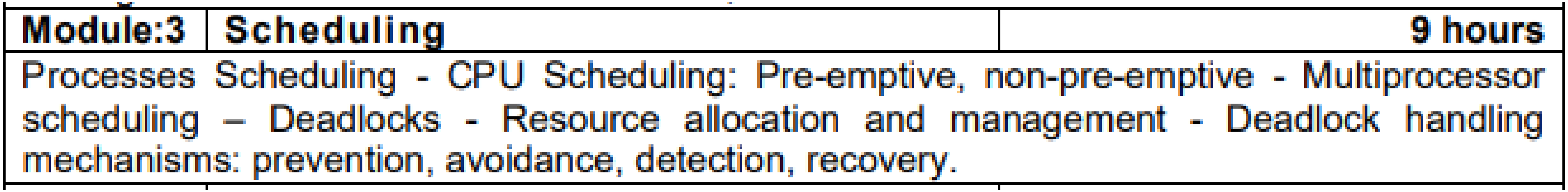
Module: 3

Scheduling



Module:3	Scheduling	9 hours
Processes Scheduling - CPU Scheduling: Pre-emptive, non-pre-emptive - Multiprocessor scheduling – Deadlocks - Resource allocation and management - Deadlock handling mechanisms: prevention, avoidance, detection, recovery.		

Process Scheduling

- The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Categories of Scheduling

There are two categories of scheduling:

1. **Non-preemptive:** Here the resource can't be taken from a process until the process completes execution. The switching of resources occurs when the running process terminates and moves to a waiting state.
2. **Preemptive:** Here the OS allocates the resources to a process for a fixed amount of time. During resource allocation, the process switches from running state to ready state or from waiting state to ready state. This switching occurs as the CPU may give priority to other processes and replace the process with higher priority with the running process.

CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - 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 running

Scheduling Criteria

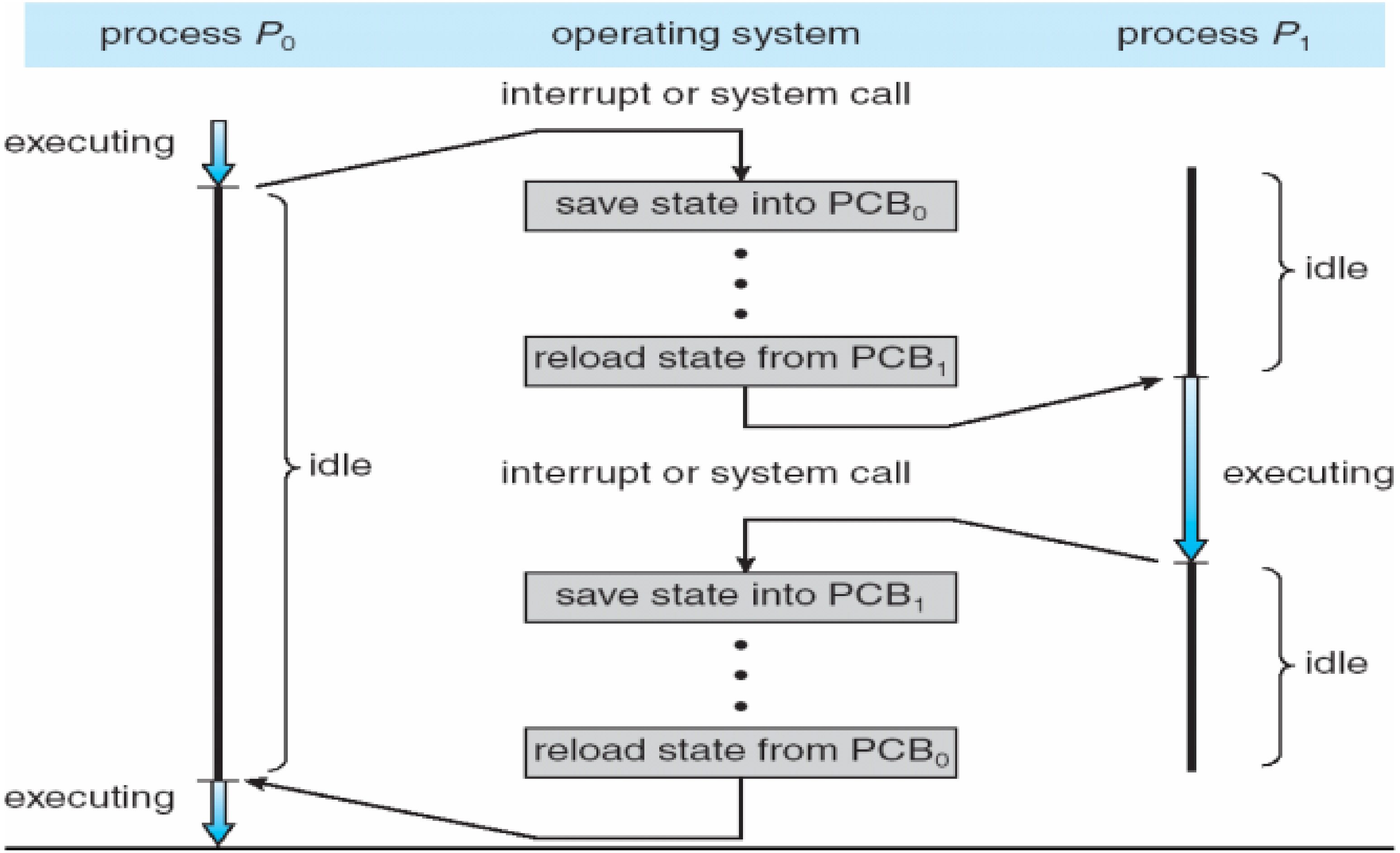
- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Scheduling Algorithm Optimization Criteria

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

- **CPU Burst:** A CPU burst is a period during which a process actively uses the CPU to execute its instructions. During this phase, the process performs computations, manipulates data, and executes various instructions that are part of its program.
- **I/O Burst:** An I/O burst, also known as an I/O operation or I/O wait, is a period during which a process is waiting for input/output (I/O) operations to be completed. During this phase, the process does not use the CPU but instead waits for data to be read from or written to external devices, such as disks, keyboards, printers, or network interfaces.

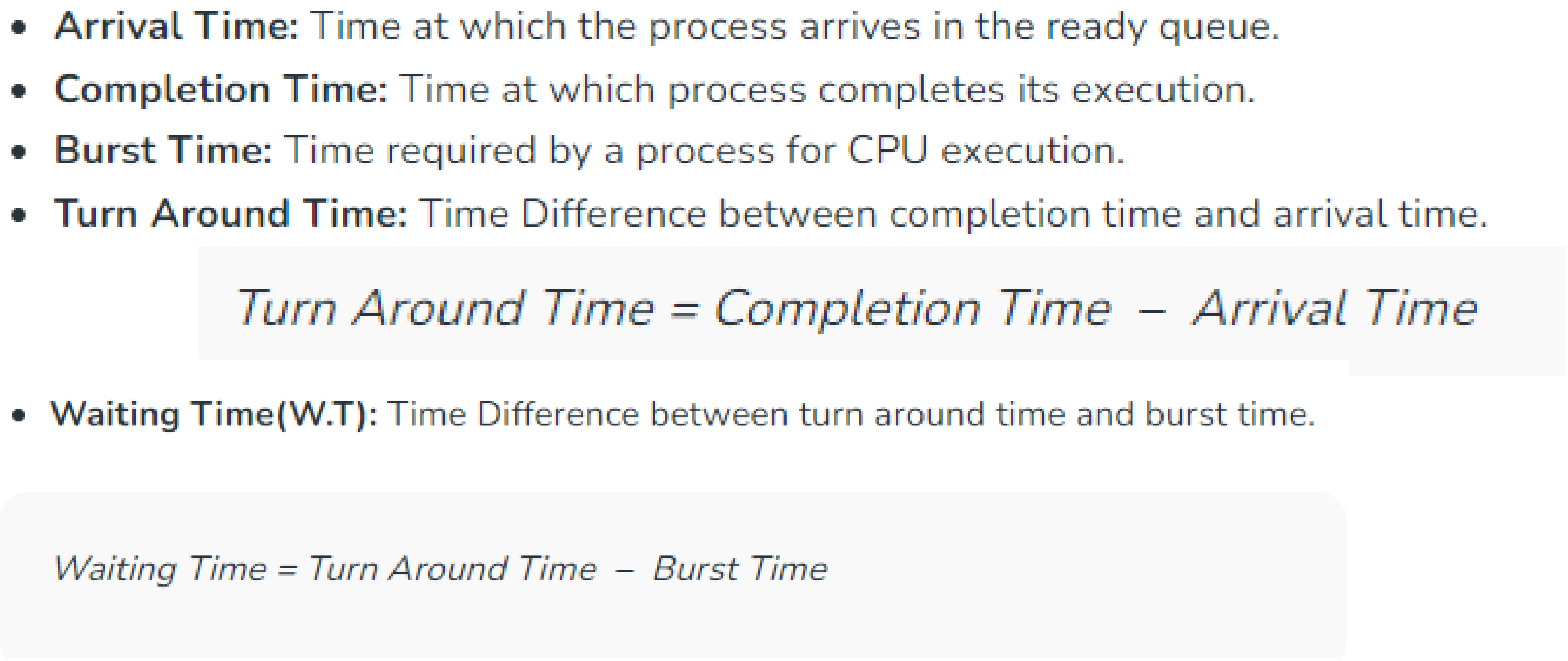
Context Switching



- A context switching is the procedure to restore and store the state of context of CPU in process control block(PCB) so that the process execution can be resumed from the same point at a later time.
- Whenever the process is switched following information is stored:
- Program counter
- Scheduling information
- Base register's and limit register's value
- Registers which are currently being used
- Changed state
- I/O state

CPU Scheduling

- Different terminologies to take care of in any CPU Scheduling algorithm?



- **Arrival Time:** Time at which the process arrives in the ready queue.
- **Completion Time:** Time at which process completes its execution.
- **Burst Time:** Time required by a process for CPU execution.
- **Turn Around Time:** Time Difference between completion time and arrival time.

$$Turn\ Around\ Time = Completion\ Time\ -\ Arrival\ Time$$

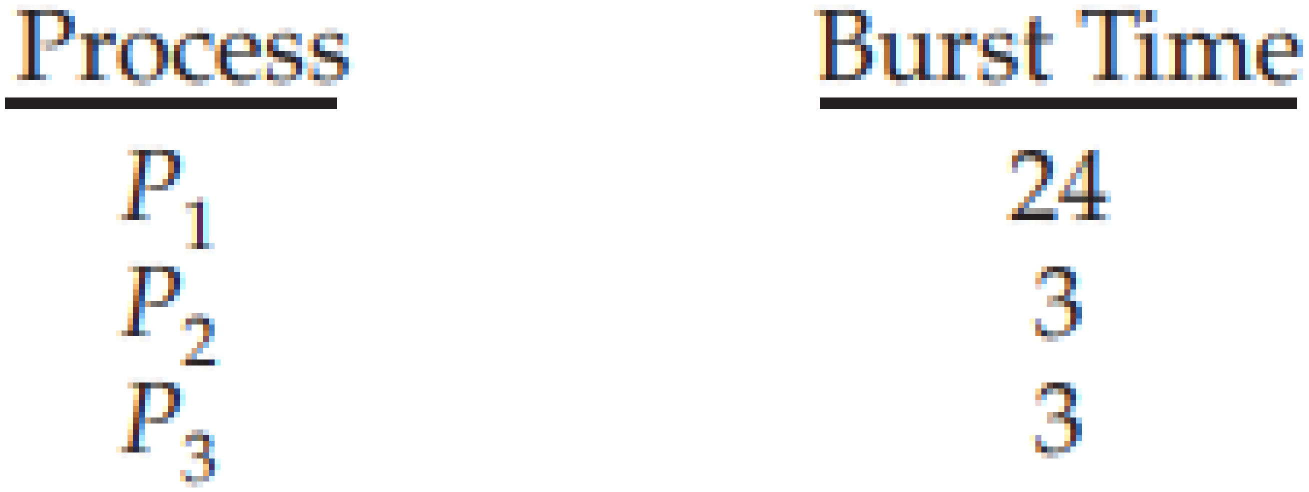
- **Waiting Time(W.T):** Time Difference between turn around time and burst time.

$$Waiting\ Time = Turn\ Around\ Time\ -\ Burst\ Time$$

Scheduling Algorithms

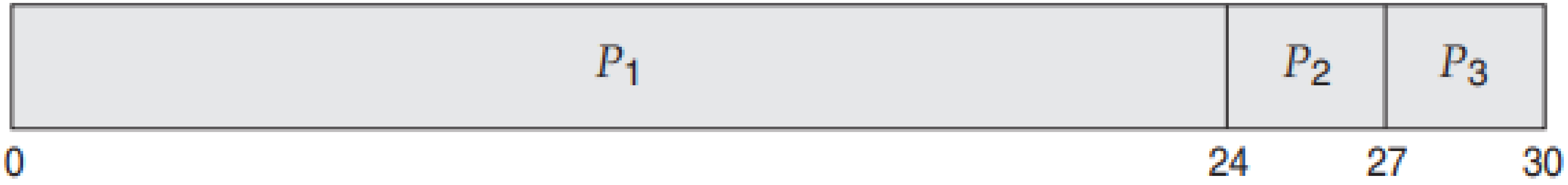
- First-Come, First-Served Scheduling
- The process that requests the CPU first is allocated the CPU first.
- The implementation of the FCFS policy is easily managed with a FIFO queue.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue.
- When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.
- On the negative side, the average waiting time under the FCFS policy is often quite long.

- Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

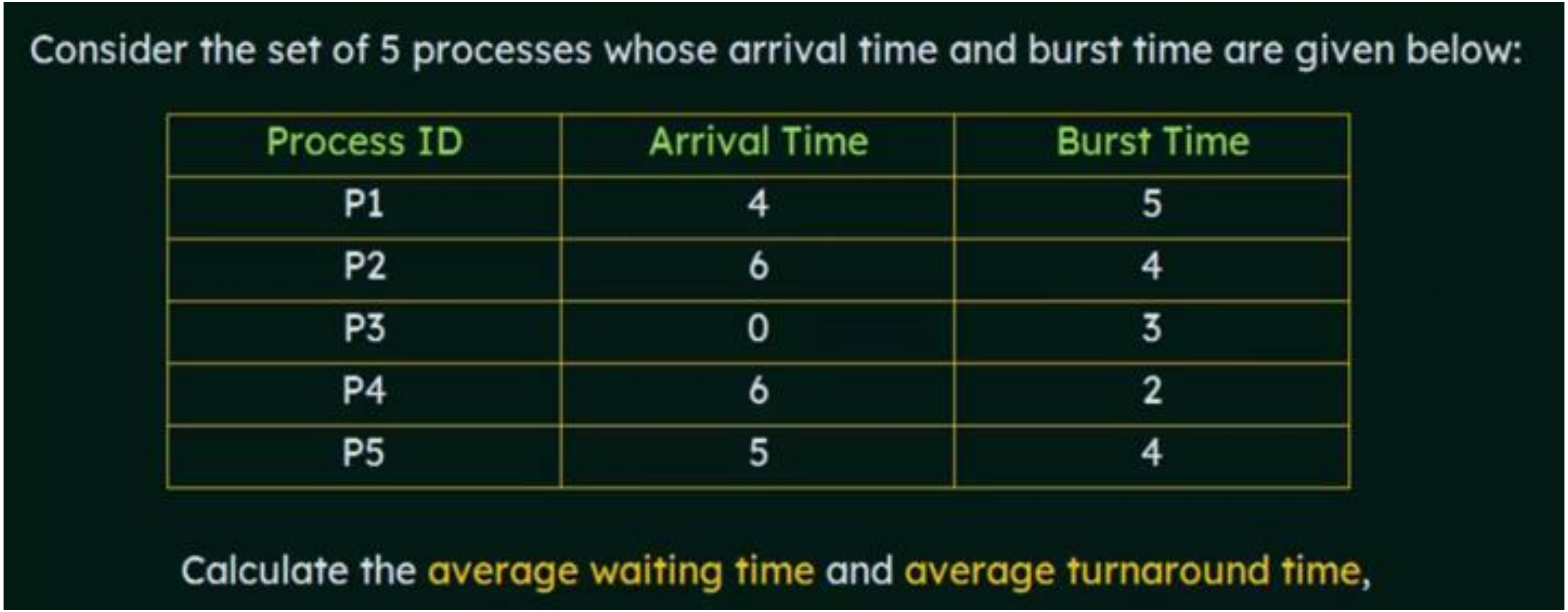


Process	Burst Time
P_1	24
P_2	3
P_3	3

- Draw Gantt Chart



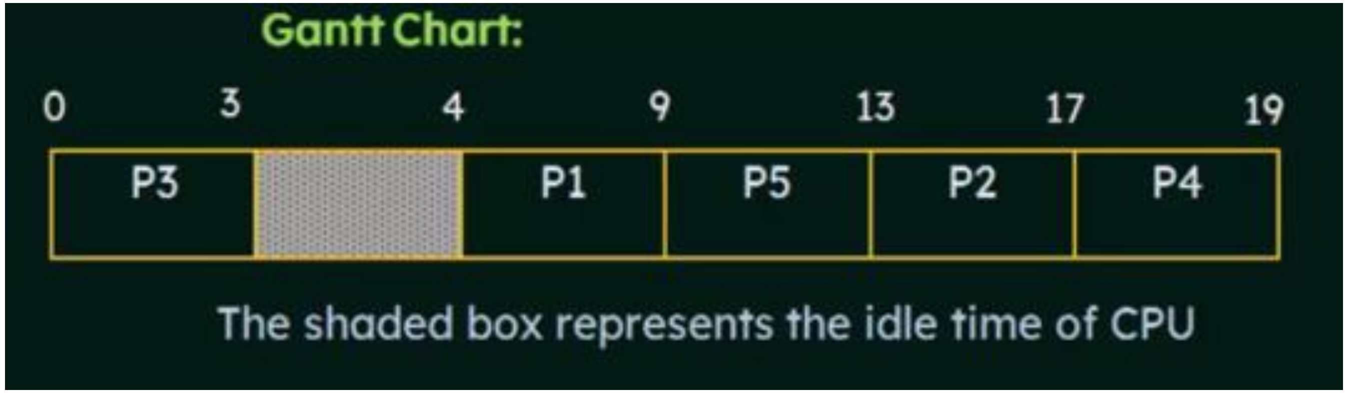
- Avg. Waiting Time: 17 ms
- Avg. Turnaround Time: 27 ms



Consider the set of 5 processes whose arrival time and burst time are given below:

Process ID	Arrival Time	Burst Time
P1	4	5
P2	6	4
P3	0	3
P4	6	2
P5	5	4

Calculate the average waiting time and average turnaround time,



Gantt Chart:

The shaded box represents the idle time of CPU

- **<u>Turn around time</u>**
- T(P3)= 3-0=3
- T(P1)=9-4=5
- T(P5)=13-5=8
- T(P2)=17-6=11
- T(P4)=19-6=13

- **<u>Waiting Time:</u>**
- W(P3)= 3-3=0
- W(P1)=5-5=0
- W(P5)=8-4=4
- W(P2)=11-4=7
- W(P4)=13-2=11

Consider the set of 3 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time
P1	0	2
P2	3	1
P3	5	6

- Average Turn Around time = (2 + 1 + 6) / 3 = 9 / 3 = 3 unit
- Average waiting time = (0 + 0 + 0) / 3 = 0 / 3 = 0 unit

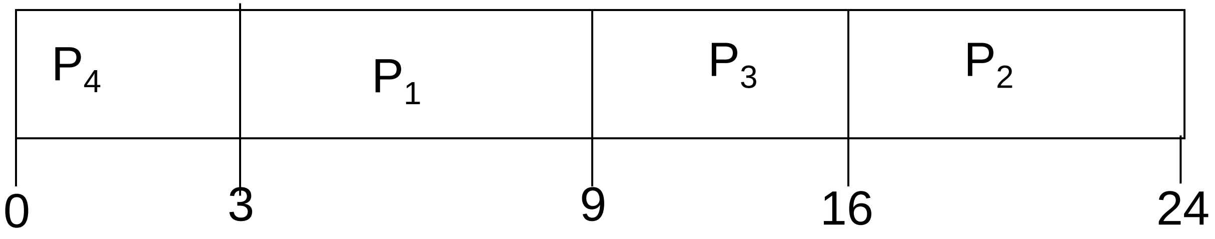
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request.

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



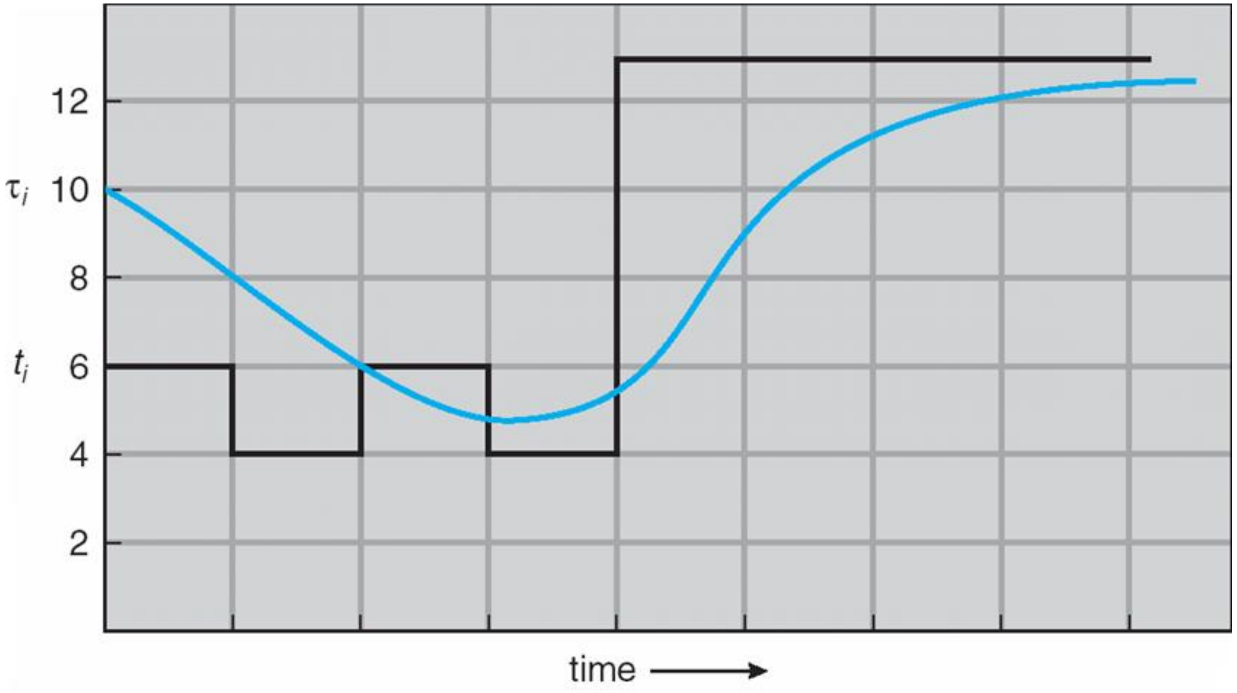
- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Determining Length of Next CPU Burst

- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging

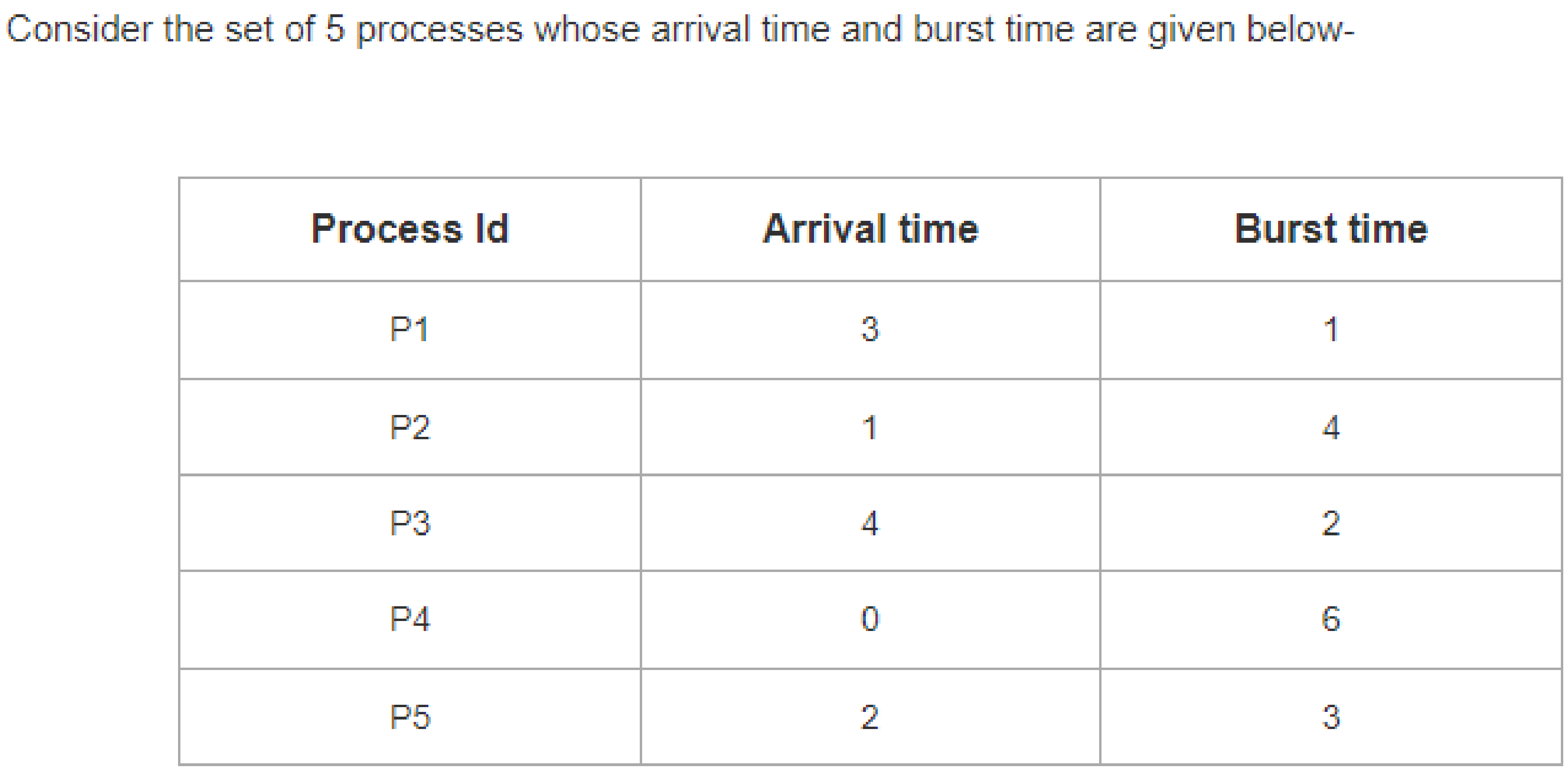
1. $t_n =$ actual length of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define : $\tau_{n+1} = \alpha\ t_n + (1 - \alpha)\ \tau_n.$

Prediction of the Length of the Next CPU Burst



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

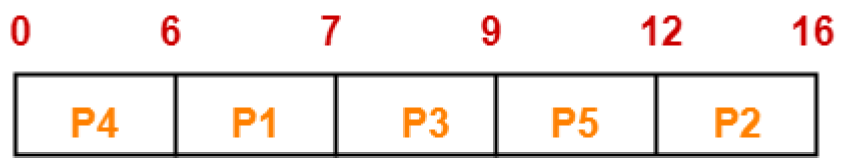
- New Estimate = α * Actual Burst Time + (1 - α) * Previous Estimate



Consider the set of 5 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time
P1	3	1
P2	1	4
P3	4	2
P4	0	6
P5	2	3

- If the CPU scheduling policy is SJF non-preemptive, calculate the average waiting time and average turn around time.



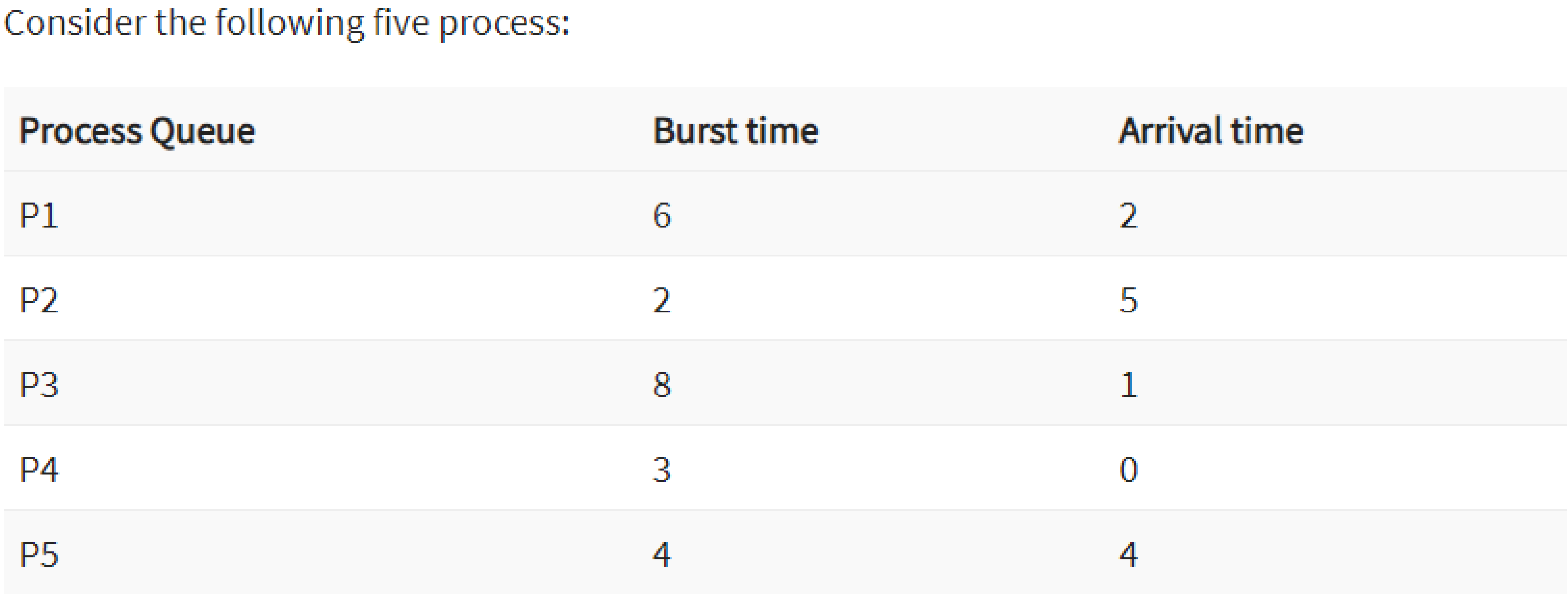
Gantt Chart

Process Id	Exit time	Turn Around time	Waiting time
P1	7	7 – 3 = 4	4 – 1 = 3
P2	16	16 – 1 = 15	15 – 4 = 11
P3	9	9 – 4 = 5	5 – 2 = 3
P4	6	6 – 0 = 6	6 – 6 = 0
P5	12	12 – 2 = 10	10 – 3 = 7

- Average Turn Around time = (4 + 15 + 5 + 6 + 10) / 5 = 40 / 5 = 8 unit
- Average waiting time = (3 + 11 + 3 + 0 + 7) / 5 = 24 / 5 = 4.8 unit

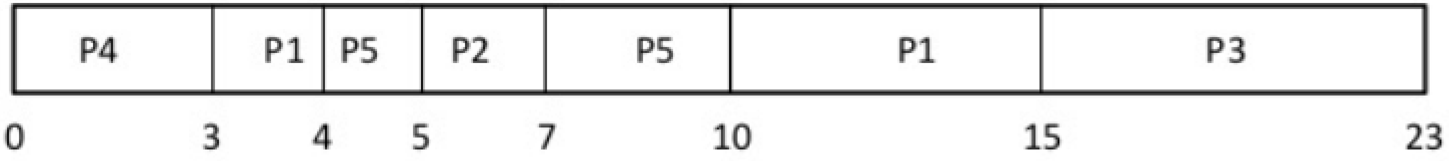
- Preemptive SJF scheduling is sometimes called shortest-remaining time-first scheduling

Apply Pre-emptive SJF, calculate Avg. waiting and turn around time



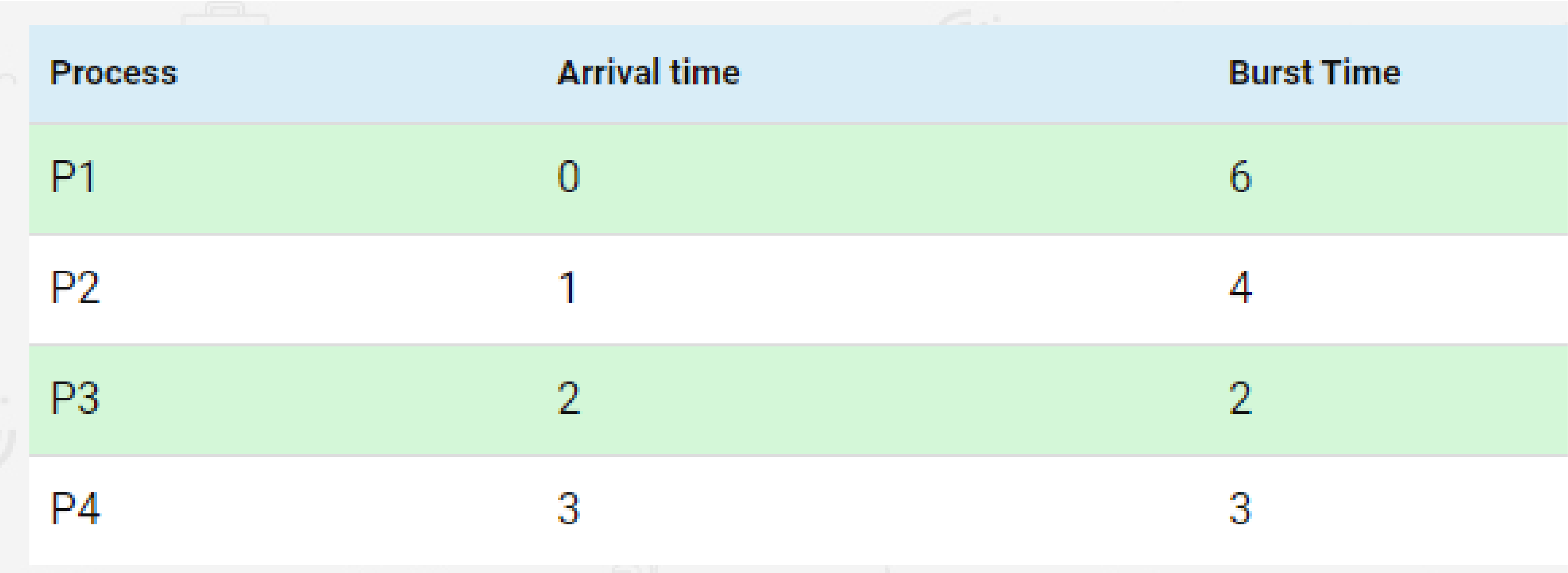
Consider the following five process:

Process Queue	Burst time	Arrival time
P1	6	2
P2	2	5
P3	8	1
P4	3	0
P5	4	4

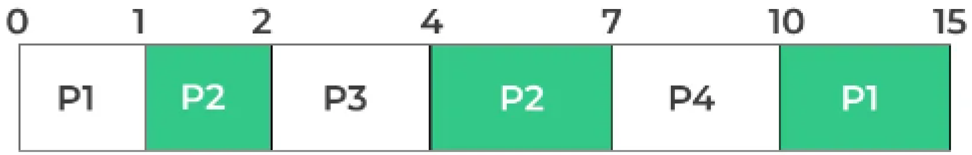


- Avg. turn around time= (3+13+6+2+22)/5= 9.2 ms
- Avg. waiting time= (0+7+2+0+14)/5= 4.6 ms

Apply Pre-emptive SJF, calculate Avg. waiting and turn around time



Process	Arrival time	Burst Time
P1	0	6
P2	1	4
P3	2	2
P4	3	3



- Avg. turn around time= 30/4 = 7.5 ms
- Avg. waiting time=15/4= 3.75 ms

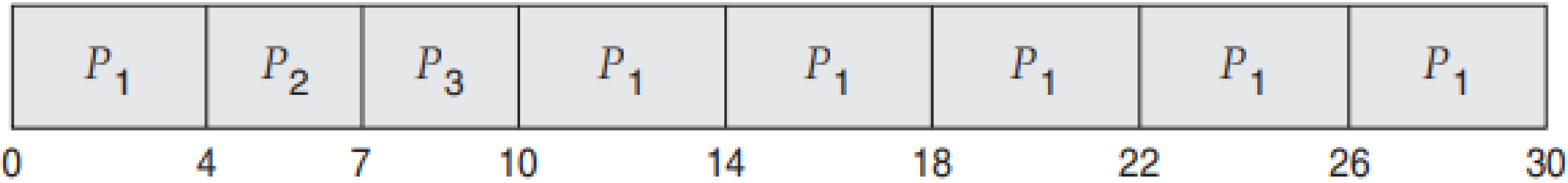
Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- We can predict wait time: If there are *n* processes in the ready queue and the time quantum is *q*, then each process 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.
- Performance
  - *q* large ⇒ FIFO
  - *q* small ⇒ may hit the context switch wall: *q* must be large with respect to context switch, otherwise overhead is too high

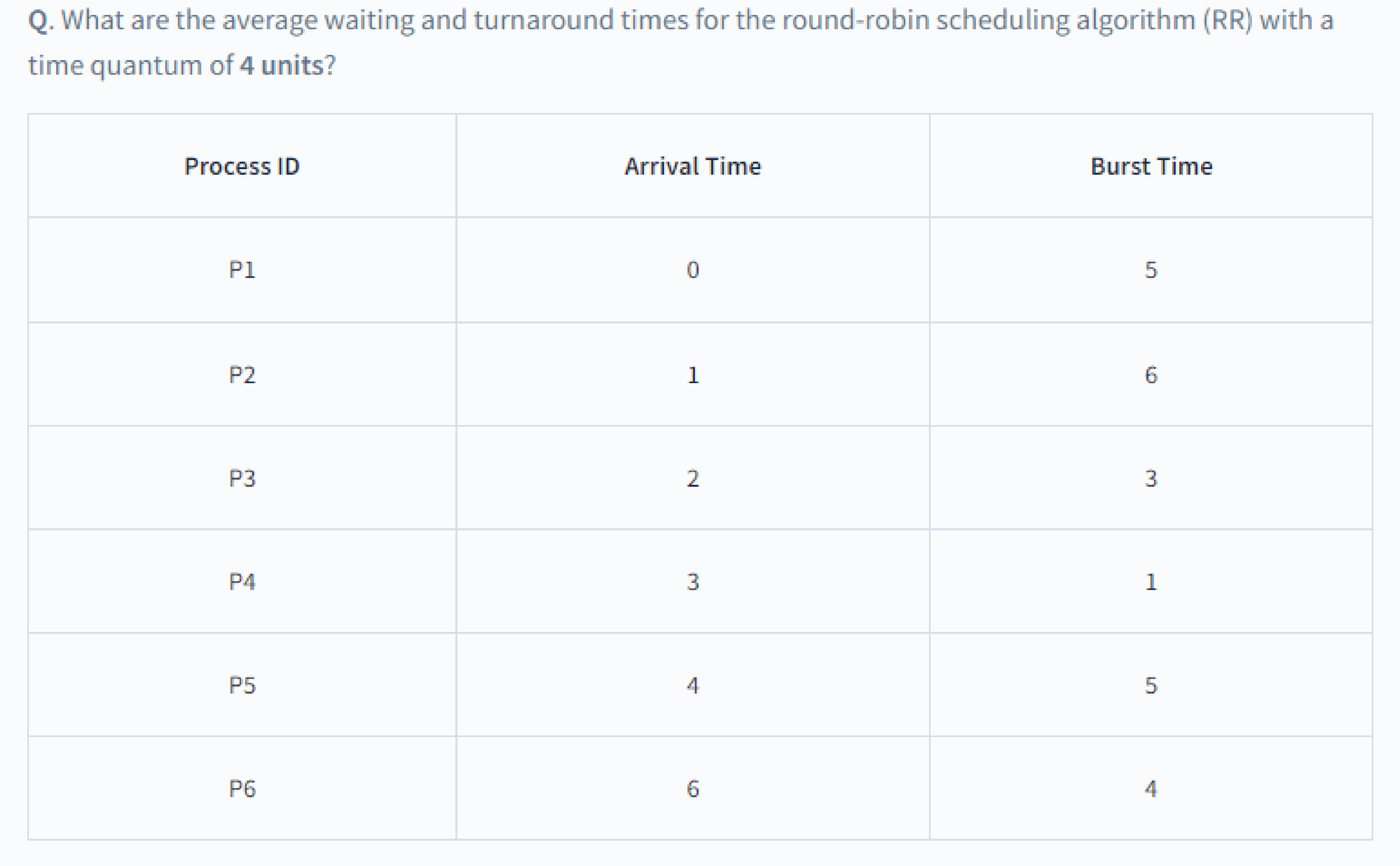
Imagine a small office where employees are using a shared printer to print their documents. The printer can only handle one document at a time, and each document takes a certain amount of time to print. The office manager has implemented a time-sharing mechanism to ensure that each employee gets a fair chance to use the printer.
Processes in the scenario correspond to employees who want to print their documents. The burst time represents the time it takes to print each document. The time quantum of 4 milliseconds represents how long each employee can use the printer before the manager switches to the next employee to ensure fairness.

Let's assign the processes and burst times:

- Process P1: Employee A has a large document to print that takes 24 seconds to print.
- Process P2: Employee B has a small document that takes 3 seconds to print.
- Process P3: Employee C also has a small document that takes 3 seconds to print.



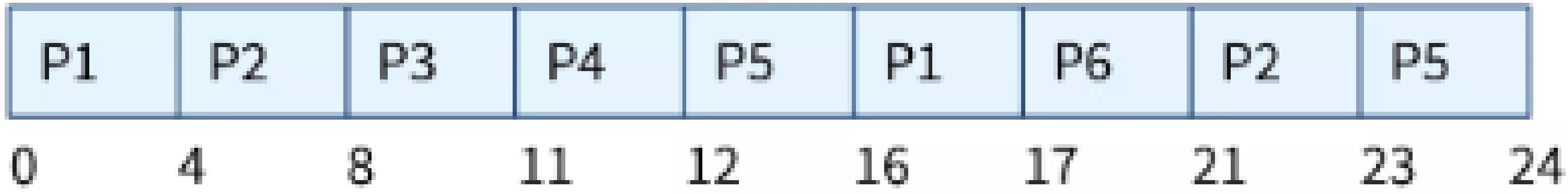
- Avg. turn around time= 47/3 = 15.67 ms
- Avg. waiting time=17/3= 5.67 ms



Q. What are the average waiting and turnaround times for the round-robin scheduling algorithm (RR) with a time quantum of **4 units**?

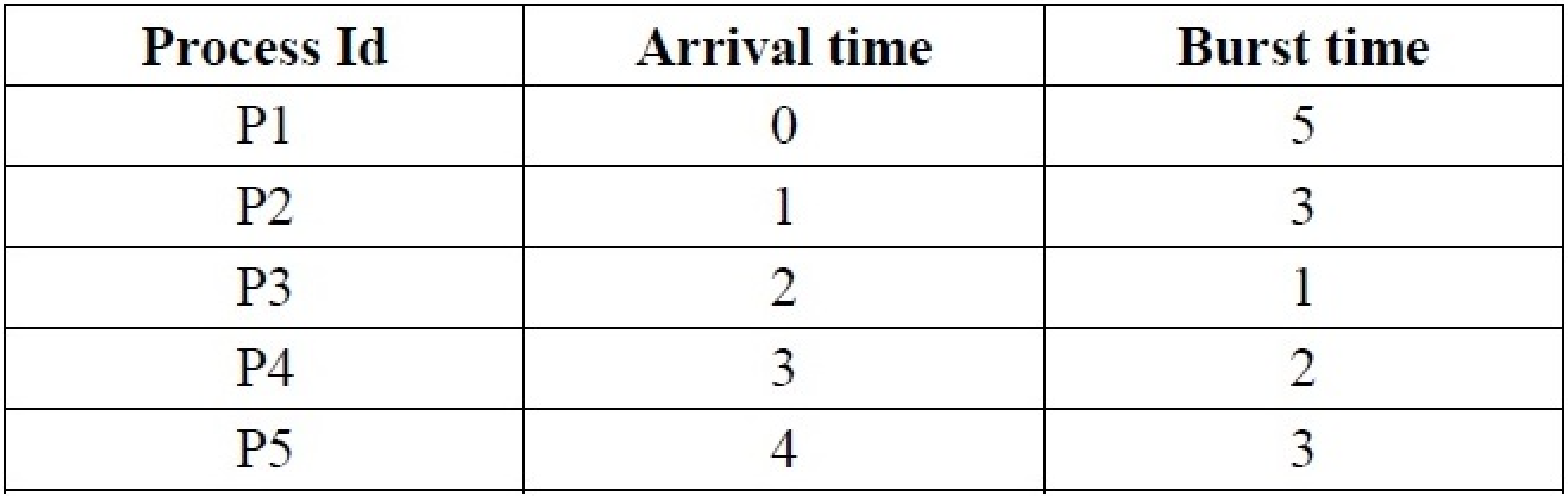
Process ID	Arrival Time	Burst Time
P1	0	5
P2	1	6
P3	2	3
P4	3	1
P5	4	5
P6	6	4

Gantt Chart

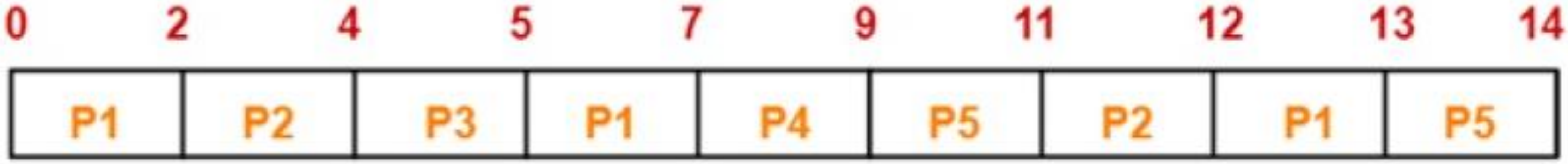


- Avg. turn around time= 15.33 ms
- Avg. waiting time=11.33 ms

- Consider a System with 5 process with their arrival time and burst time as shown in following table. If the Time Quantum is of 2 ms then calculate the Average Waiting time and average turnaround time.



Process Id	Arrival time	Burst time
P1	0	5
P2	1	3
P3	2	1
P4	3	2
P5	4	3

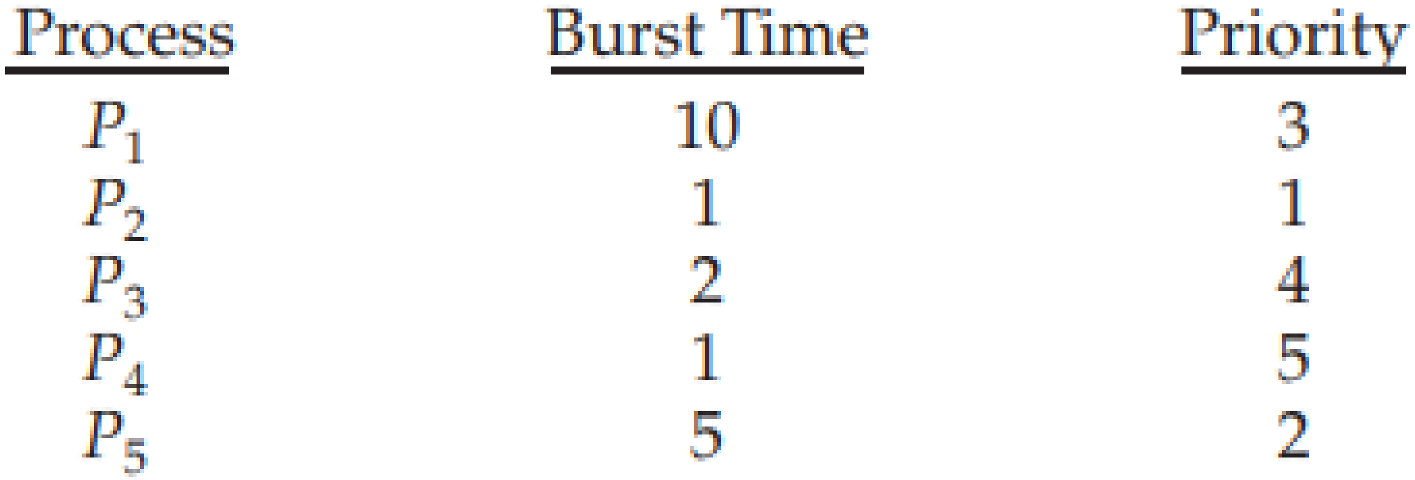


- Average Waiting time = (13+11+3+6+10)/5= 8.6 ms
- Average turnaround time =(8+8+2+4+7)/5= 5.8 ms

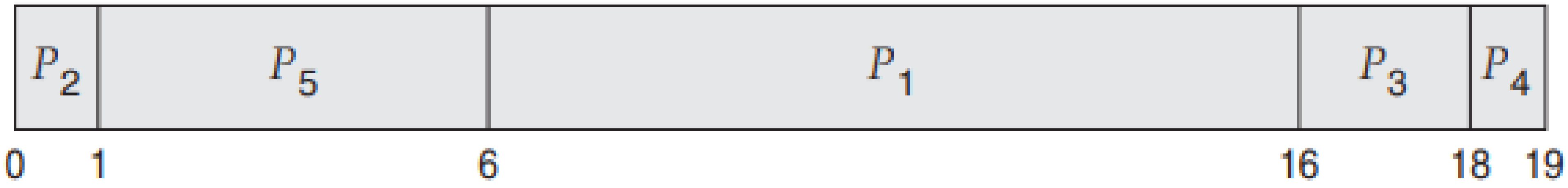
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
    - Preemptive
    - Non-preemptive
- Note that SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

- consider the following set of processes, assumed to have arrived at time 0 in the order P1, P2, ···, P5, with the length of the CPU burst given in milliseconds:

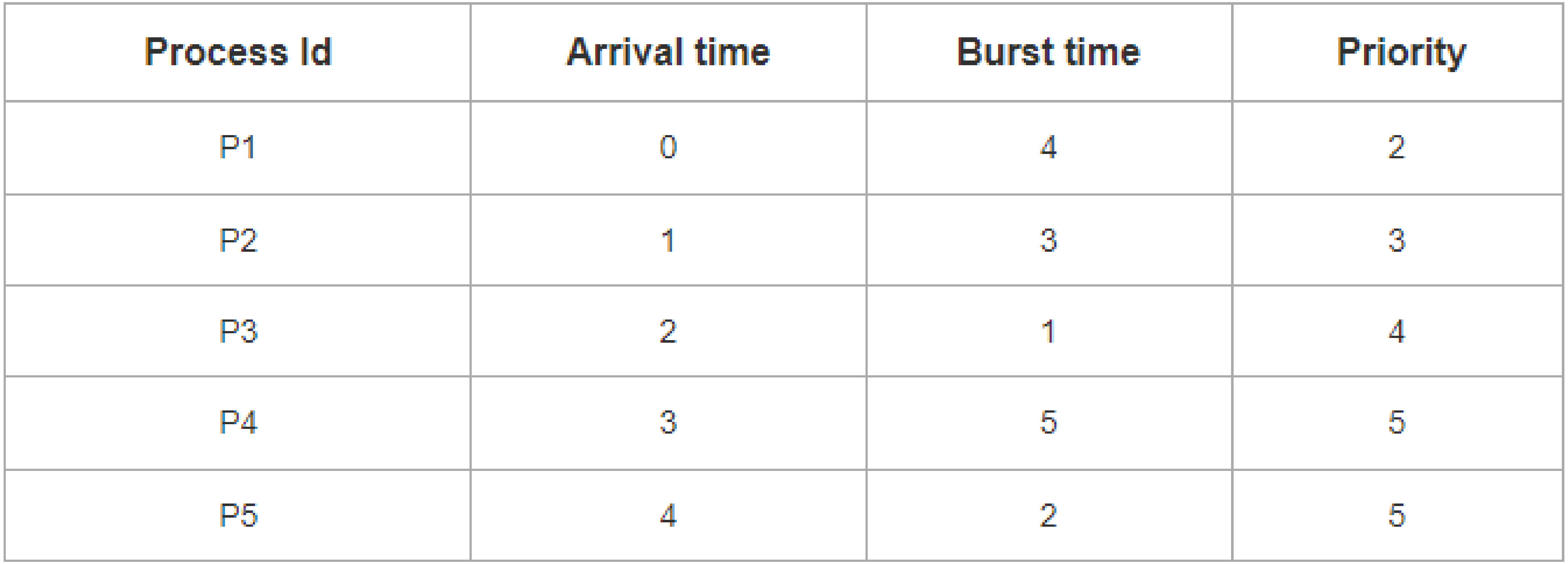


Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

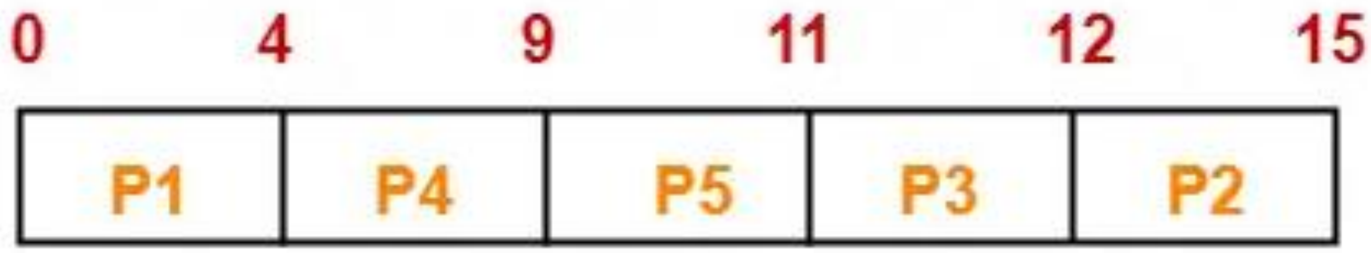


- Average Waiting time = (18+16+6+1+0)/5= 8.2 ms
- Average turnaround time =(19+18+16+6+1)/5= 12 ms

- Apply CPU scheduling policy of priority non-preemptive, calculate the average waiting time and average turn around time. (Higher number represents higher priority)



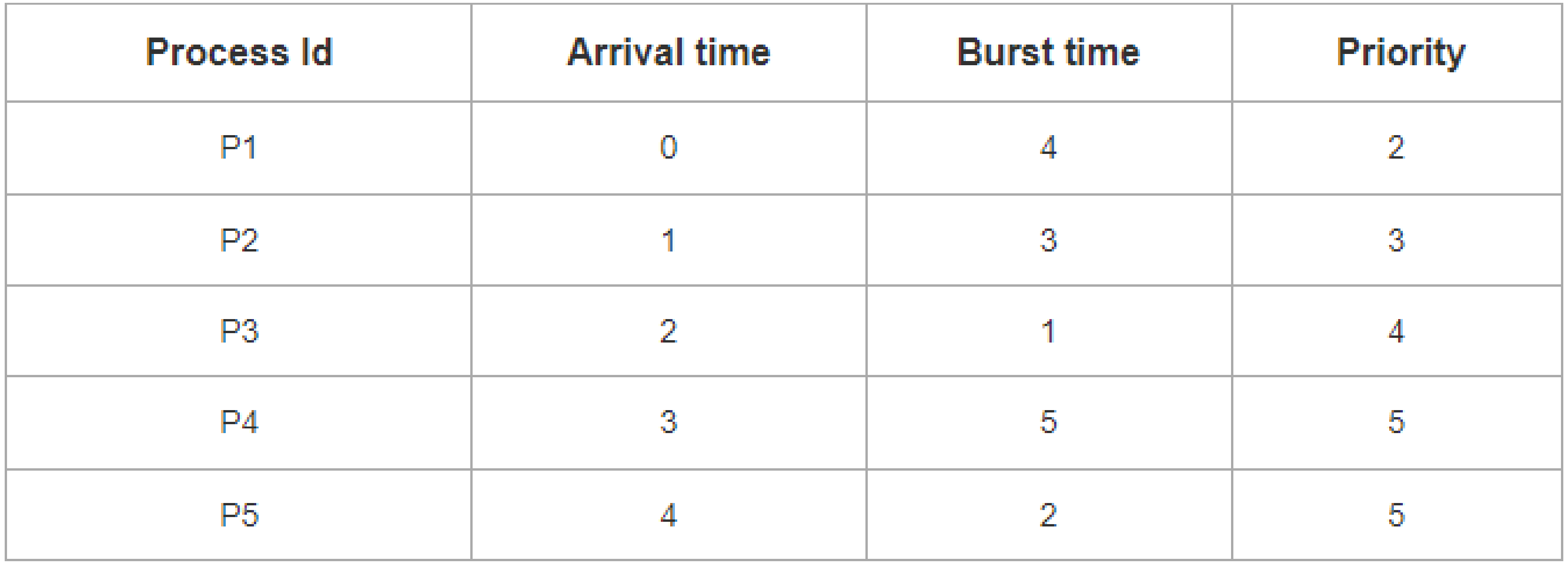
Process Id	Arrival time	Burst time	Priority
P1	0	4	2
P2	1	3	3
P3	2	1	4
P4	3	5	5
P5	4	2	5



Gantt Chart

- Average Turn Around time = (4 + 14 + 10 + 6 + 7) / 5 = 41 / 5 = 8.2 unit
- Average waiting time = (0 + 11 + 9 + 1 + 5) / 5 = 26 / 5 = 5.2 unit

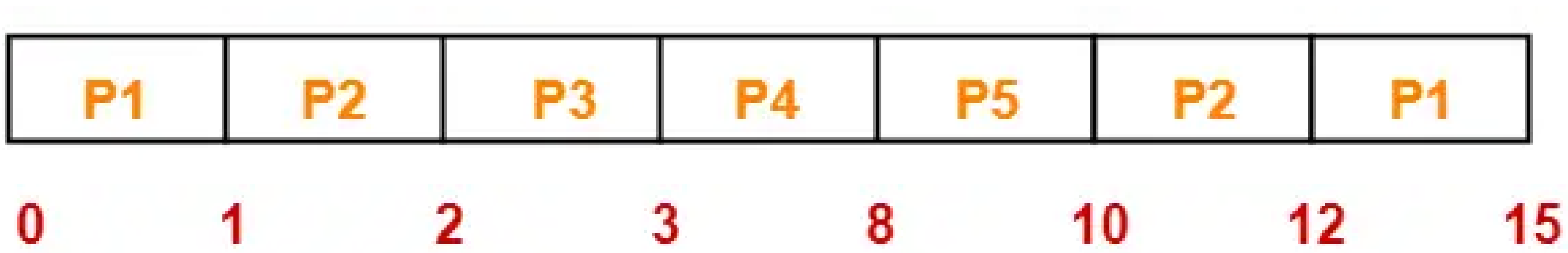
Consider the set of 5 processes whose arrival time and burst time are given below-



Process Id	Arrival time	Burst time	Priority
P1	0	4	2
P2	1	3	3
P3	2	1	4
P4	3	5	5
P5	4	2	5

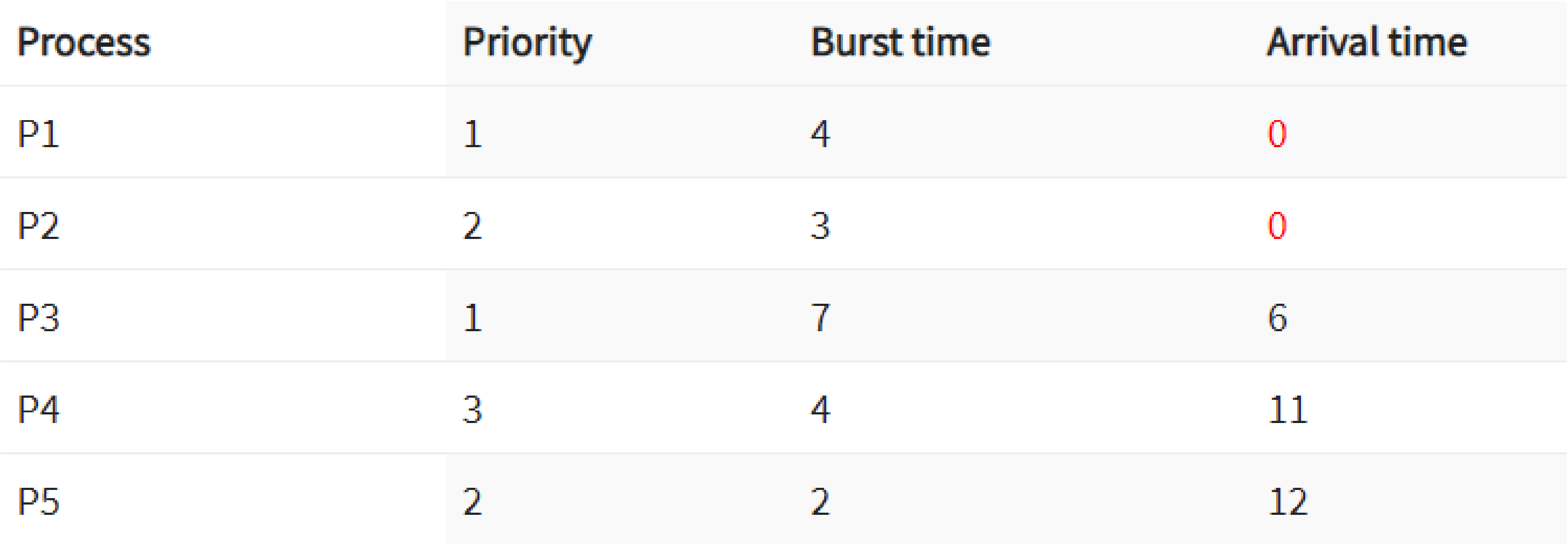
- If the CPU scheduling policy is priority preemptive, calculate the average waiting time and average turn around time. (Higher number represents higher priority)

Gantt Chart-

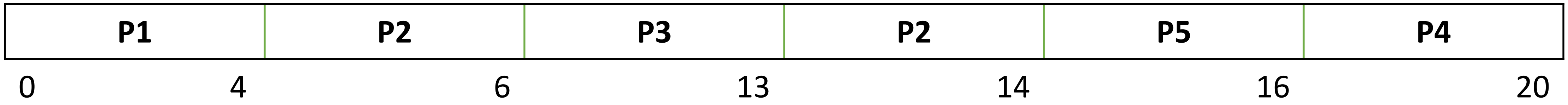


- Average Turn Around time = (15 + 11 + 1 + 5 + 6) / 5 = 38 / 5 = 7.6 unit
- Average waiting time = (11 + 8 + 0 + 0 + 4) / 5 = 23 / 5 = 4.6 unit

- Consider following five processes P1 to P5. Each process has its unique priority, burst time, and arrival time. Apply pre-emptive priority scheduling (Priority 1 is the highest and 3 is the lowest).Calculate average waiting and turn around time.



Process	Priority	Burst time	Arrival time
P1	1	4	0
P2	2	3	0
P3	1	7	6
P4	3	4	11
P5	2	2	12



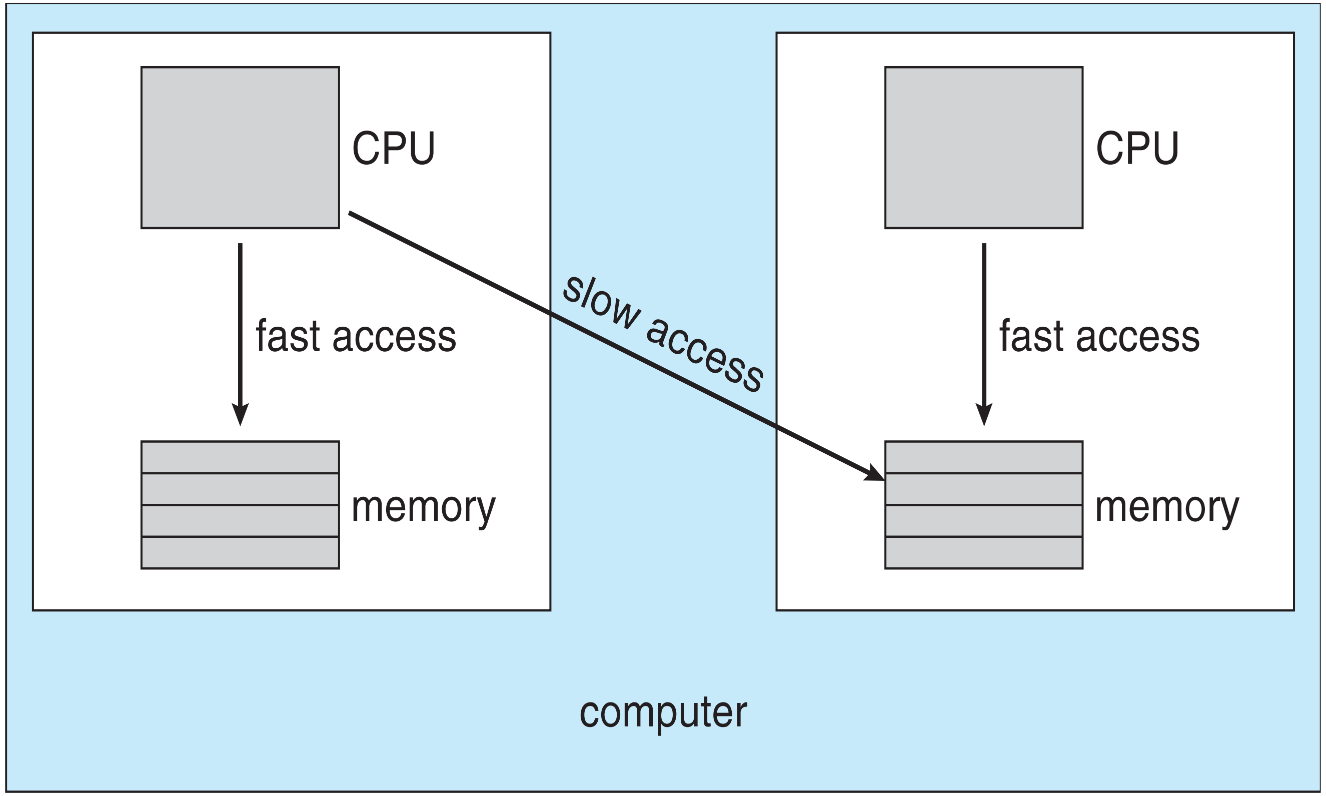
P1 P2 P3 P2 P5 P4

0 4 6 13 14 16 20

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing** **(SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
  - Currently, most common
- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity**
  - **hard affinity**
  - Variations including **processor sets**

NUMA and CPU Scheduling



Note that memory-placement algorithms can also consider affinity

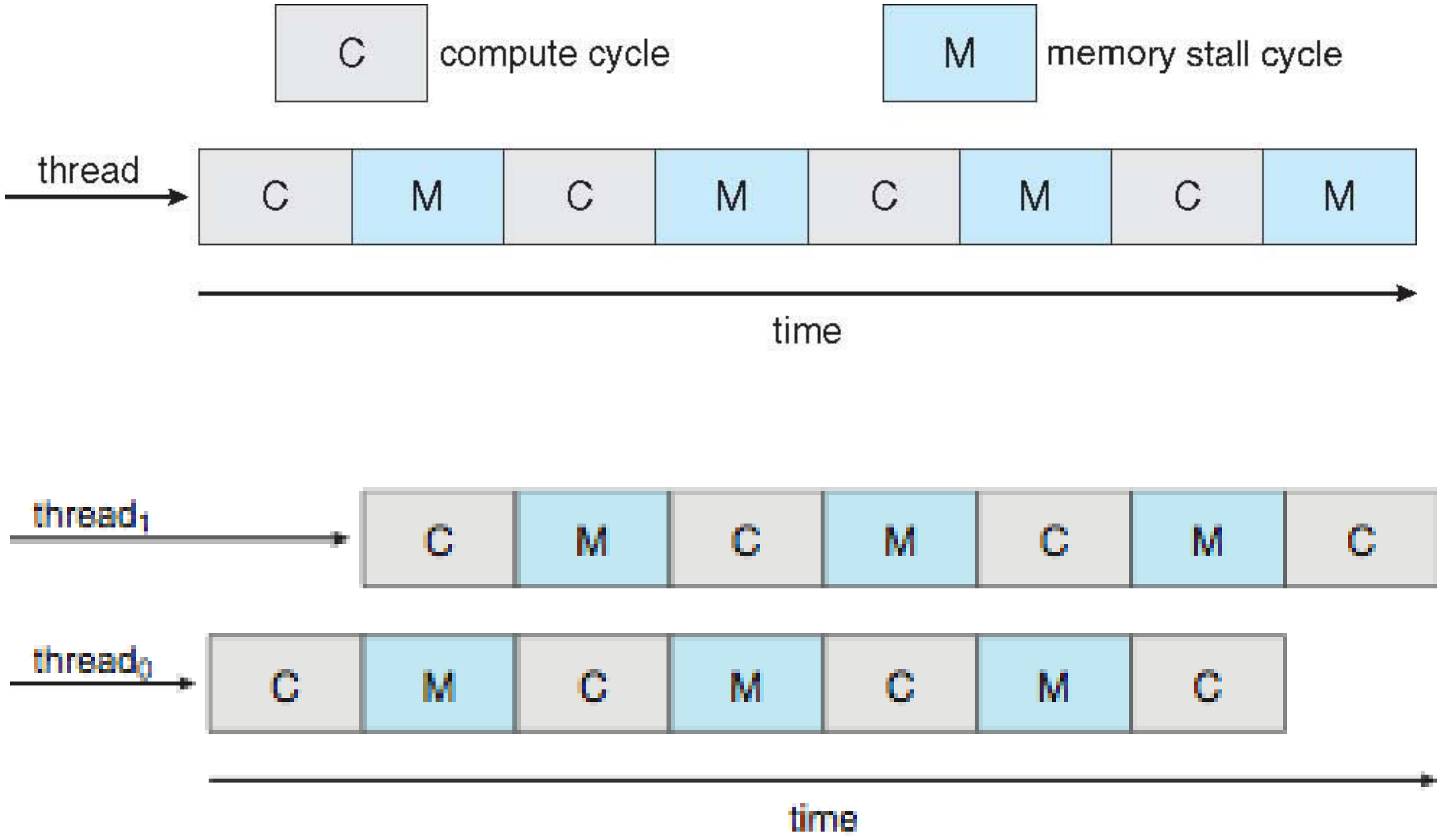
Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor

Multicore Processors

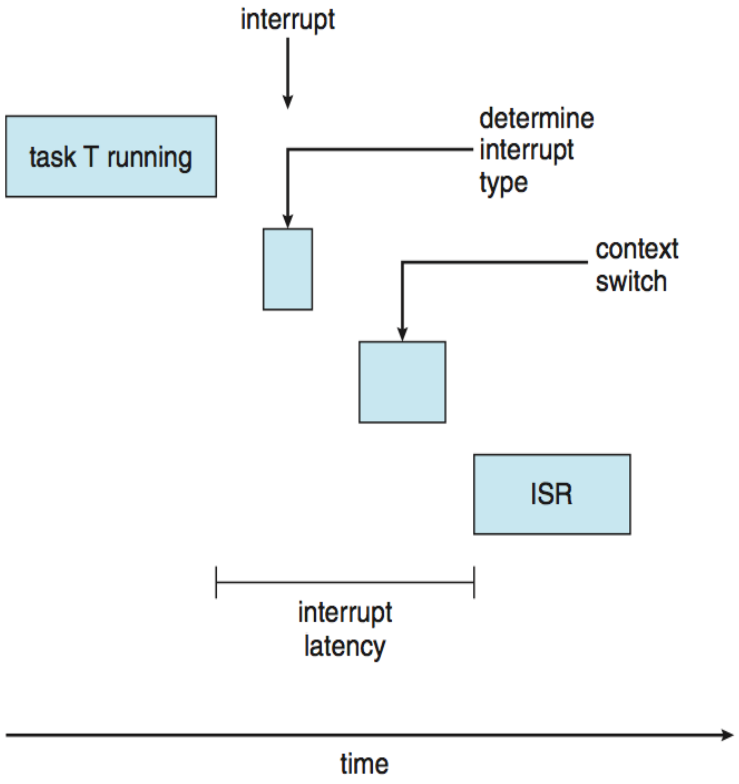
- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens

Multithreaded Multicore System



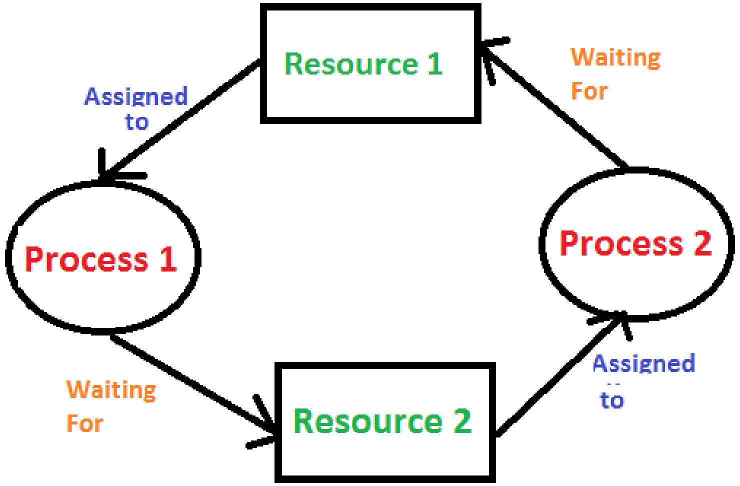
Real-Time CPU Scheduling

- Can present obvious challenges
- **Soft real-time systems** – nc 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



Deadlocks

A ***deadlock*** is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.



System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion**: only one process at a time can use a resource
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait**: there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

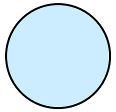
Resource-Allocation Graph

A set of vertices *V* and a set of edges *E*.

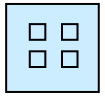
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

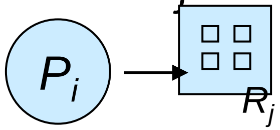
- Process



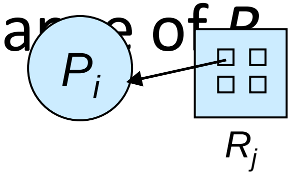
- Resource Type with 4 instances



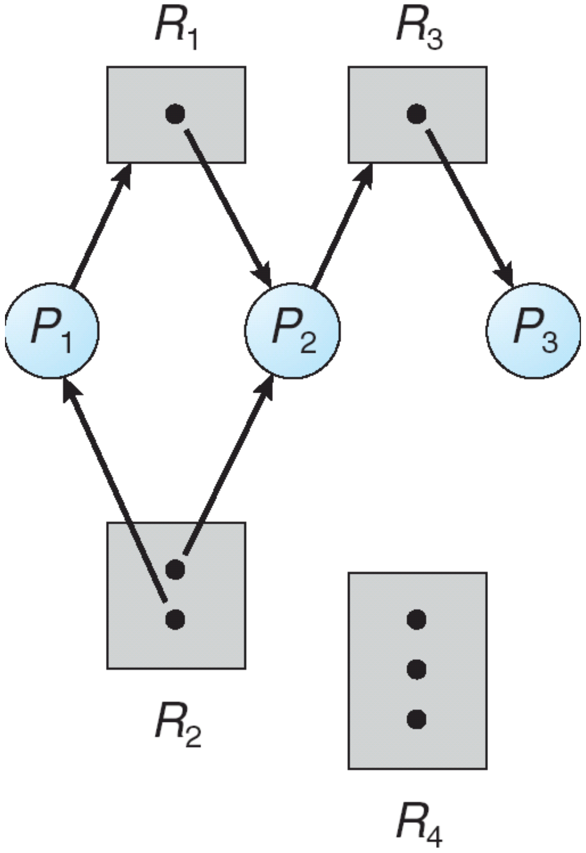
- P_i requests instance of R_j



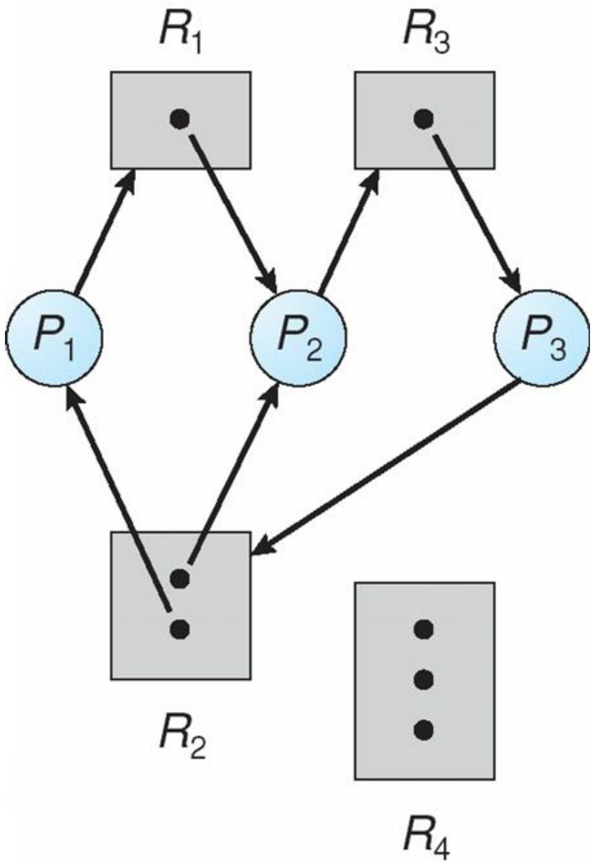
- P_i is holding an instance of R_j



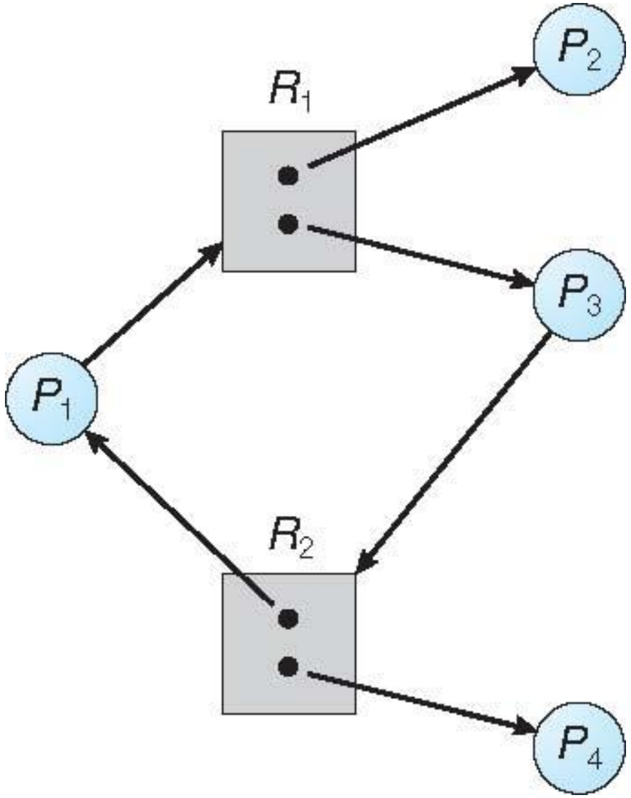
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Basic Facts

- If graph contains no cycles ⇒ no deadlock
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

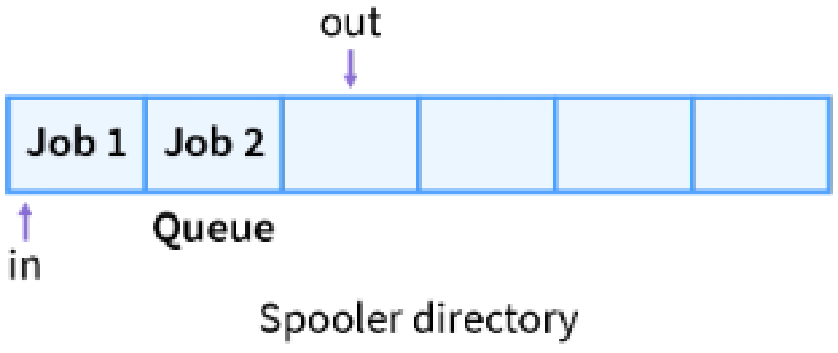
- Ensure that the system will ***never*** enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidence
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources

Spooling (Simultaneous Peripheral Operations On-line) is a computer term that refers to a technique used in operating systems to improve the efficiency of input/output operations, particularly for devices such as printers, disk drives, and other peripherals. **It involves temporarily storing data from input or output processes in a buffer** or a temporary location to allow for smoother and more efficient data transfer between the computer and the peripheral device.



in : pointer to next file to printed.

out : pointer to next empty slot.

- **Challenges of Spooling:**
- Spooling can only be used for the resources with associated memory, like a Printer.
- It may also cause **race condition**. A race condition is a situation where two or more processes are accessing a resource and the final results cannot be definitively determined.
- **For Example:** In printer spooling, if process A overwrites the job of process B in the queue, then process B will never receive the output.

Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources

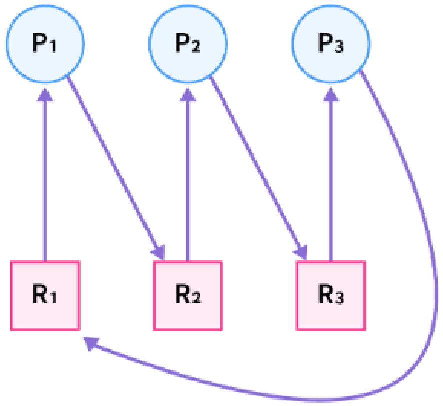
- Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
- Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

For example: If a process has resources R1, R2, and R3 and it is waiting for resource R4, then it has to release R1, R2, and R3 and put a new request of all resources again.

- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration



Let's suppose we have two processes P and Q acquiring resources Ri and Rj. For deadlock to occur, P need to hold Ri and request Rj , while Q holds Rj and requests Ri. And if we define linear ordering (increasing or decreasing) of resources, this implies that i < j (for P's request order) and j < i (for Q's request order), which is impossible thus preventing deadlock.

Deadlock Avoidance

- Deadlock Avoidance is an OS strategy to prevent deadlocks. Processes declare their maximum resource needs to the OS. The OS simulates resource allocation and checks if it can fulfill all needs, averting potential deadlocks.

Deadlock Avoidance

Requires that the system has some additional ***a priori*** information available

- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

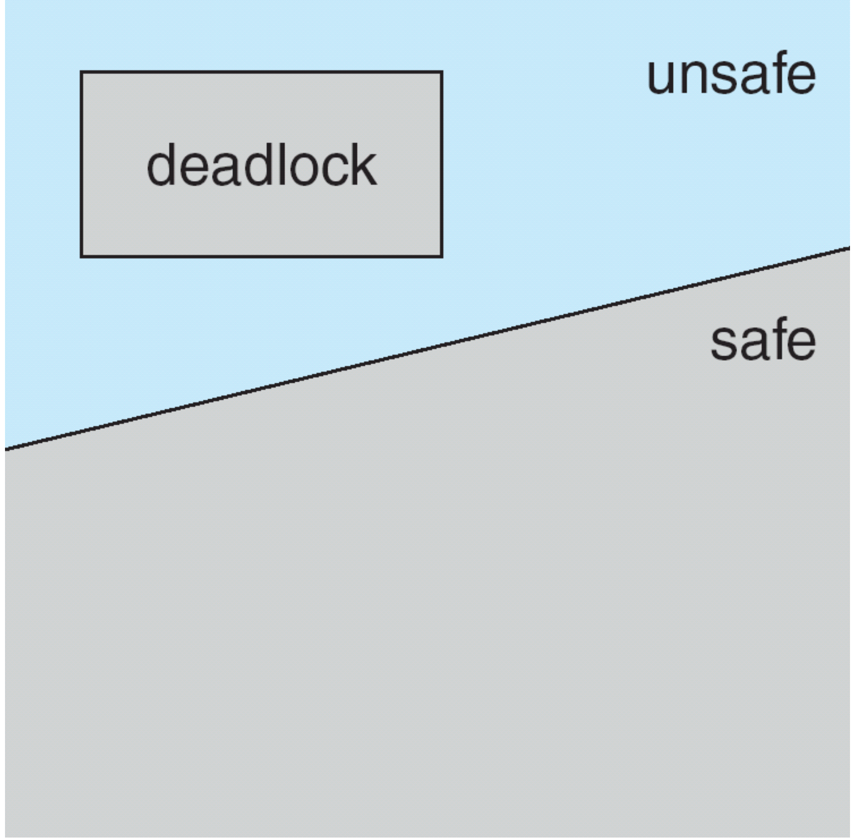
Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < I$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Basic Facts

- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will never enter an unsafe state.

Safe, Unsafe, Deadlock State



Avoidance Algorithms

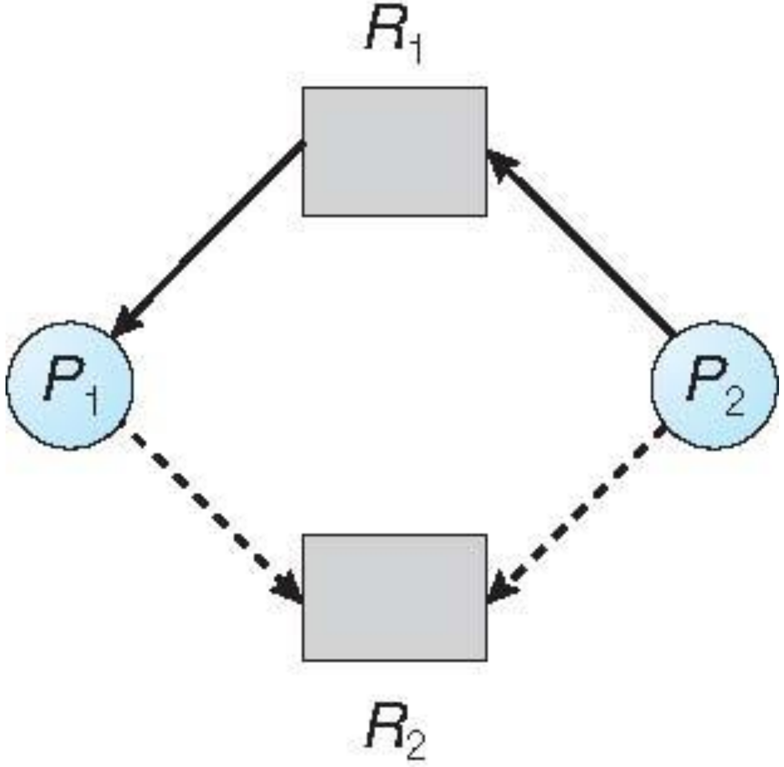
- Single instance of a resource type
  - Use a resource-allocation graph

- Multiple instances of a resource type
  - Use the banker’s algorithm

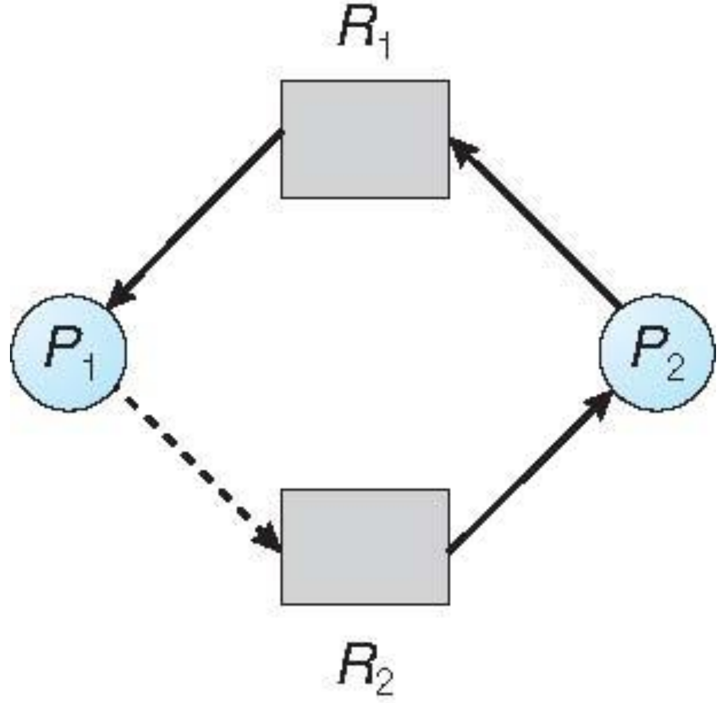
Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker’s Algorithm

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker's Algorithm

Let *n* = number of processes, and *m* = number of resources types.

- **Available**: Vector of length *m*. If available [*j*] = *k*, there are *k* instances of resource type R_j available
- **Max**: *n x m* matrix. If *Max* [*i,j*] = *k*, then process P_i may request at most *k* instances of resource type R_j
- **Allocation**: *n* x *m* matrix. If Allocation[*i,j*] = *k* then P_i is currently allocated *k* instances of R_j
- **Need**: *n* x *m* matrix. If *Need*[*i,j*] = *k*, then P_i may need *k* more instances of R_j to complete its task

Need [*i,j*] = *Max*[*i,j*] – *Allocation* [*i,j*]

Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize:

   Work = Available

   Finish **[*i*] = *false* for *i* = 0, 1, …, *n*- 1**

2. Find an ***i*** such that both:

   (a) ***Finish*** **[*i*] = *false***

   (b) ***Need*$_i$** $\leq$ ***Work***

   If no such ***i*** exists, go to step 4

3. ***Work = Work + Allocation*$_i$**

   Finish*[*i*] = *true

   go to step 2

4. If ***Finish*** **[*i*] == *true*** for all ***i***, then the system is in a safe state

Resource-Request Algorithm for Process P_i

$\boldsymbol{Request_i}$ = request vector for process $\boldsymbol{P_i}$. If $\boldsymbol{Request_i\,[j] = k}$ then process $\boldsymbol{P_i}$ wants $\boldsymbol{k}$ instances of resource type $\boldsymbol{R_j}$

1. If $\boldsymbol{Request_i} \leq \boldsymbol{Need_i}$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $\boldsymbol{Request_i} \leq \boldsymbol{Available}$, go to step 3. Otherwise $\boldsymbol{P_i}$ must wait, since resources are not available
3. Pretend to allocate requested resources to $\boldsymbol{P_i}$ by modifying the state as follows:

$$\boldsymbol{Available = Available - Request_i;}$$
$$\boldsymbol{Allocation_i = Allocation_i + Request_i;}$$
$$\boldsymbol{Need_i = Need_i - Request_i;}$$

- □ If safe ⇒ the resources are allocated to $\boldsymbol{P_i}$
- □ If unsafe ⇒ $\boldsymbol{P_i}$ must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

- 5 processes P_0 through P_4;
  3 resource types:
  A (10 instances), *B* (5instances), and *C* (7 instances)
- Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example (Cont.)

- The content of the matrix ***Need*** is defined to be ***Max – Allocation***

	Need		
	A	*B*	*C*
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

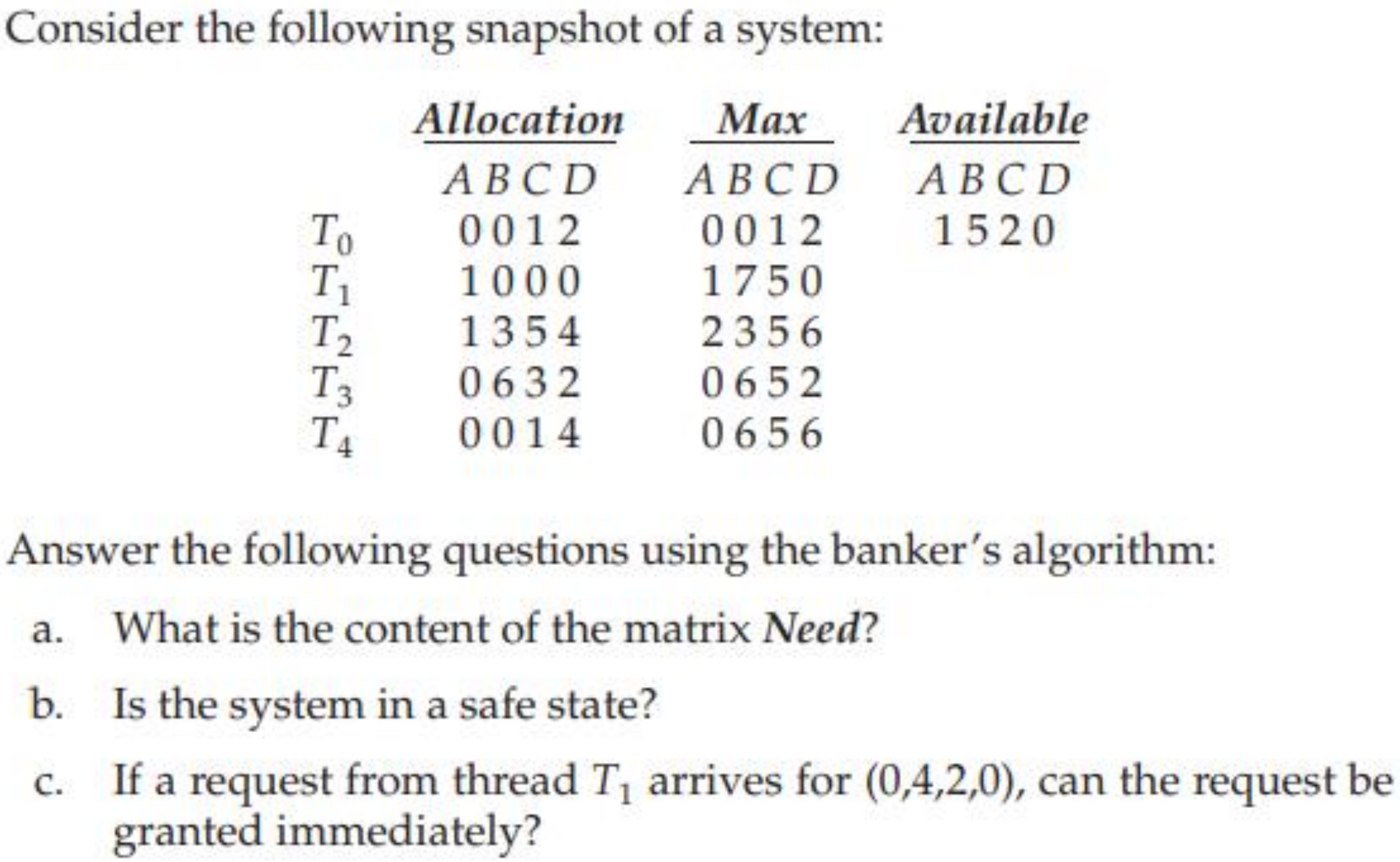
- The system is in a safe state since the sequence < P_1, P_3, P_4, *P0*, *P2*> satisfies safety criteria

P_1 Request (1,0,2)

- Check that Request $\leq$ Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ true

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence < $\boldsymbol{P_1, P_3, P_4, P_0, P_2}$> satisfies safety requirement
- Can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?
- Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted?



Consider the following snapshot of a system:

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
T_0	0 0 1 2	0 0 1 2	1 5 2 0
T_1	1 0 0 0	1 7 5 0	
T_2	1 3 5 4	2 3 5 6	
T_3	0 6 3 2	0 6 5 2	
T_4	0 0 1 4	0 6 5 6	

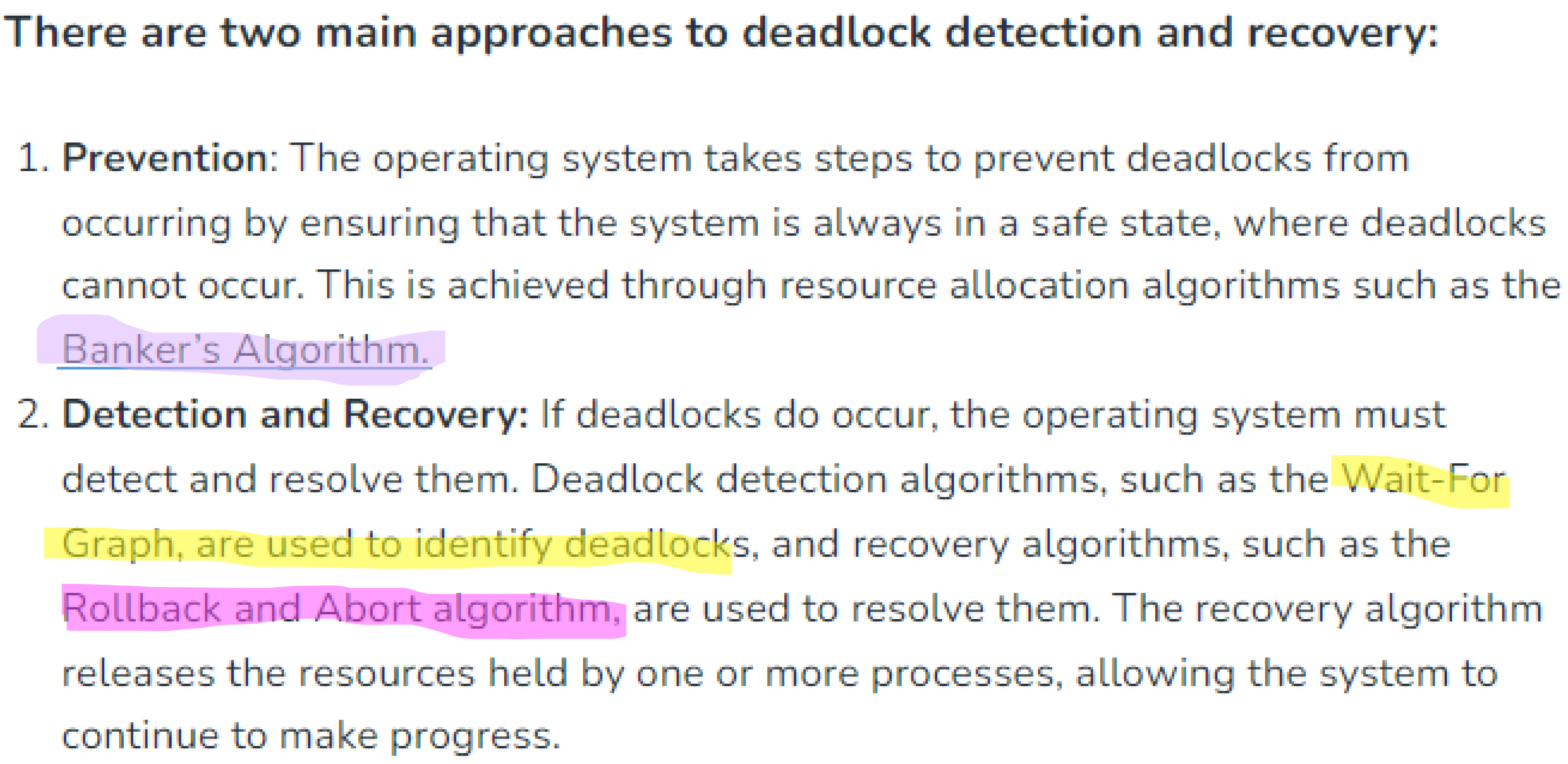
Answer the following questions using the banker′s algorithm:

a. What is the content of the matrix ***Need***?

b. Is the system in a safe state?

c. If a request from thread T_1 arrives for (0,4,2,0), can the request be granted immediately?

Deadlock Detection



There are two main approaches to deadlock detection and recovery:

1. **Prevention**: The operating system takes steps to prevent deadlocks from occurring by ensuring that the system is always in a safe state, where deadlocks cannot occur. This is achieved through resource allocation algorithms such as the Banker's Algorithm.
2. **Detection and Recovery:** If deadlocks do occur, the operating system must detect and resolve them. Deadlock detection algorithms, such as the Wait-For Graph, are used to identify deadlocks, and recovery algorithms, such as the Rollback and Abort algorithm, are used to resolve them. The recovery algorithm releases the resources held by one or more processes, allowing the system to continue to make progress.

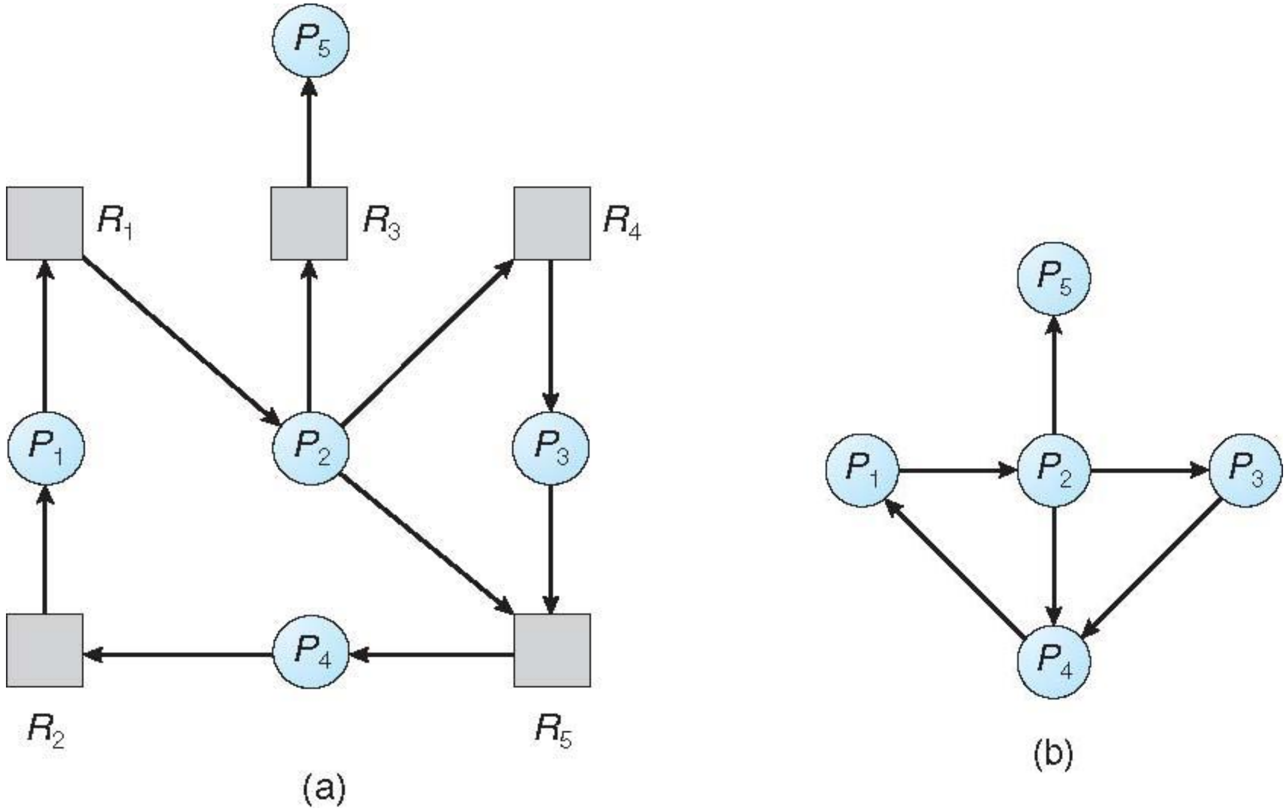
Single Instance of Each Resource Type

- Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j

- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock

- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph   Corresponding wait-for graph

Several Instances of a Resource Type

- **Available***:* A vector of length ***m*** indicates the number of available resources of each type
- **Allocation***:* An ***n* x *m*** matrix defines the number of resources of each type currently allocated to each process
- **Request***:* An ***n* x *m*** matrix indicates the current request of each process. If ***Request* [*i*][*j*] = *k***, then process P_i is requesting ***k*** more instances of resource type R_j.

Detection Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively Initialize:

   (a) ***Work*** **=** ***Available***

   (b) For ***i*** **= 1,2, …,** ***n***, if ***Allocation***$_i$ **≠ 0**, then ***Finish*****[i] =** ***false***; otherwise, ***Finish*****[i] =** ***true***

2. Find an index ***i*** such that both:

   (a) ***Finish*****[*i*] ==** ***false***

   (b) ***Request***$_i$ **≤** ***Work***

   If no such ***i*** exists, go to step 4

Detection Algorithm (Cont.)

3. ***Work = Work + Allocation**$_i$*
Finish**[i] **= true
go to step 2

4. If ***Finish[i] == false***, for some ***i***, 1 ≤ ***i*** ≤ ***n***, then the system is in deadlock state. Moreover, if ***Finish**[i]* **==** ***false***, then ***P**$_i$* is deadlocked

Algorithm requires an order of O(*m* x *n*^{2}) operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

- Five processes ***P_0*** through ***P_4***; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)

- Snapshot at time ***T_0***:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence <***P_0, P_2, P_3, P_1, P_4***> will result in ***Finish[i] = true*** for all ***i***

Example (Cont.)

- ***P_2*** requests an additional instance of type ***C***

	Request		
	A	*B*	*C*
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

- State of system?
  - Can reclaim resources held by process ***P_0***, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes ***P_1***, ***P_2***, ***P_3***, and ***P_4***

Apply deadlock detection algorithm to solve the following problem. There are five processes and 4 resource types.

	ALLOCATION				REQUEST				AVAILABLE			
	A	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
P1	1	0	0	0	0	1	0	0	2	0	0	0
P2	0	1	0	0	0	0	1	0				
P3	0	0	1	0	0	0	0	1				
P4	0	1	0	1	1	0	0	0				
P5	0	0	0	1	0	0	0	0				

Do a step by step execution of Dead Lock Detection Algorithm to find the processes are in deadlock? If the system has no deadlock show the execution of sequence processes.