



Operating Systems ICS 431

Weeks 6 - 7

Ch. 5: Processes Scheduling

Dr. Tarek Helmy El-Basuny

Ch: 5 Process Scheduling

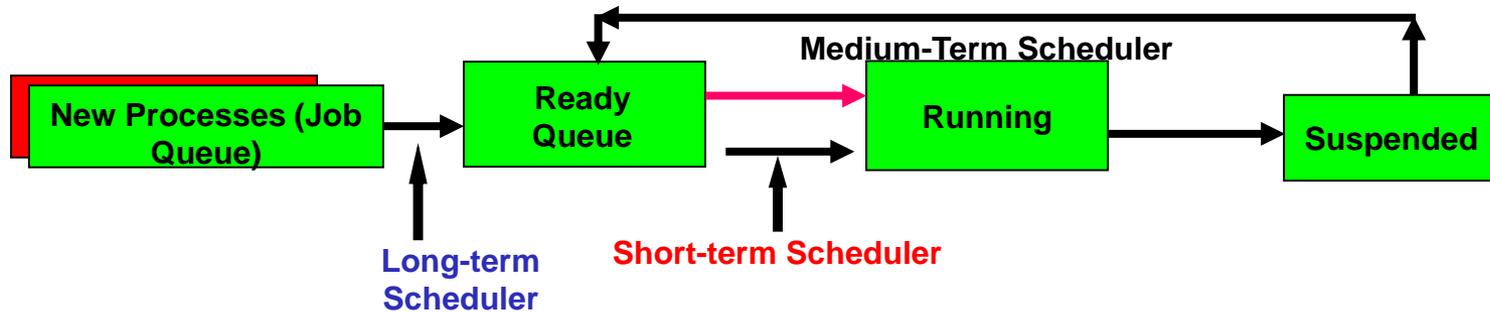
- Schedulers Overview (Short, Medium and Long)
- Time Quantum/Burst Time and Context Switch
- Scheduler, Dispatcher and Swapper modules in the OS
- Basic Assumptions for Process Scheduling
- Types of Short-term Scheduling Algorithms (Preemptive, and Non-preemptive)
 - Scheduling decision and the Types of processes
- Evaluation Criteria of Scheduling Algorithms
- Scheduling Algorithms
 - First-Come, First-Served (FCFS)
 - Shortest Job First (SJF)
 - Shortest Remaining Time First (SRTF)
 - Process Burst Length Prediction
 - Round Robin
 - Priority-Based
 - Multilevel Queue
 - Multilevel Feedback Queue
- Scheduling Algorithms Evaluation
 - Deterministic and Queuing models
 - Simulations, and Implementation
- Scheduling Polices in Different OS
- **Multiprocessor Systems: Just Introduction, Scheduling of Multiprocessors systems**

Ch: 5 (Process Scheduling) Objectives

- Recognize the importance of CPU scheduling for the operation of computer systems.
- Understand the decision policy of different CPU scheduling algorithms.
- Know about the strengths and weaknesses of each algorithm.
- Obtain the knowledge to evaluate and select the appropriate scheduling algorithm for different computing environments.

Schedulers Overview

- Processes entering the system are put into a **job queue**.
- Processes ready to be executed are kept in the **ready queue**.
- Processes waiting for a particular device are placed in one of the **I/O queues**.



- **Scheduler**: a module decides which process will be dispatched to the CPU.
- **Dispatcher**: a module gives control of the CPU to the process selected by the short-term scheduler.
- **Swapper** - manages transfer of processes between main memory and virtual memory (medium-term scheduler)
- **Dispatch/Context latency**: Time it takes for the OS to stop one running process and to start running another one.
- In the Context/Dispatch time, the OS will:
 - Remove the “old” process from the CPU by the dispatcher.
 - Select the next process by the scheduler.
 - Allocate the “new” process to the CPU by the dispatcher.

Time Quantum/Burst Time

- How does the OS interleave execution of many processes on the CPU?
- Answer: By sharing the CPU time among the processes
- Time Quantum/Burst Time: Amount of CPU time given to each process.

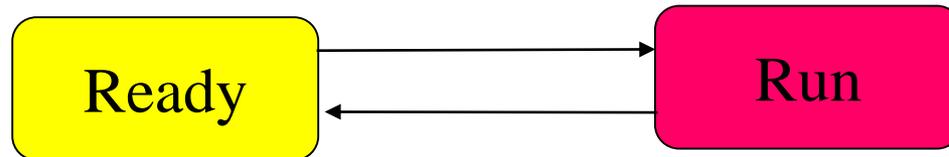
Q: What happens if the quantum time is too short?

A: Too much overhead for context switching.

Q: What if the time quantum time is too long?

A: Poor response times, some processes will starve

A Context Switch occurs **when a process exchange is made between the ready and run states.**



During the Context Switch from Process A to B

1. OS Kernel:

- Sets the PC register with the address of the Interrupt Handler (IH), which is an address in the interrupt vector of OS kernel.
- Setting the mode bit to switch from user mode to kernel mode.
- Copies A's state from CPU registers to A's PCB.
- Sets A's state to Ready or Blocked.
- Inserts a pointer of A's PCB on Ready-Queue or I/O-Queue.

2. Scheduler: Selects a process B to run, based on its scheduling policy.

3. OS Kernel:

- Sets B's state to Running,
- Copies B's state from B's PCB to CPU registers.
- Kernel transfers control to B and switches from kernel mode back to user mode.

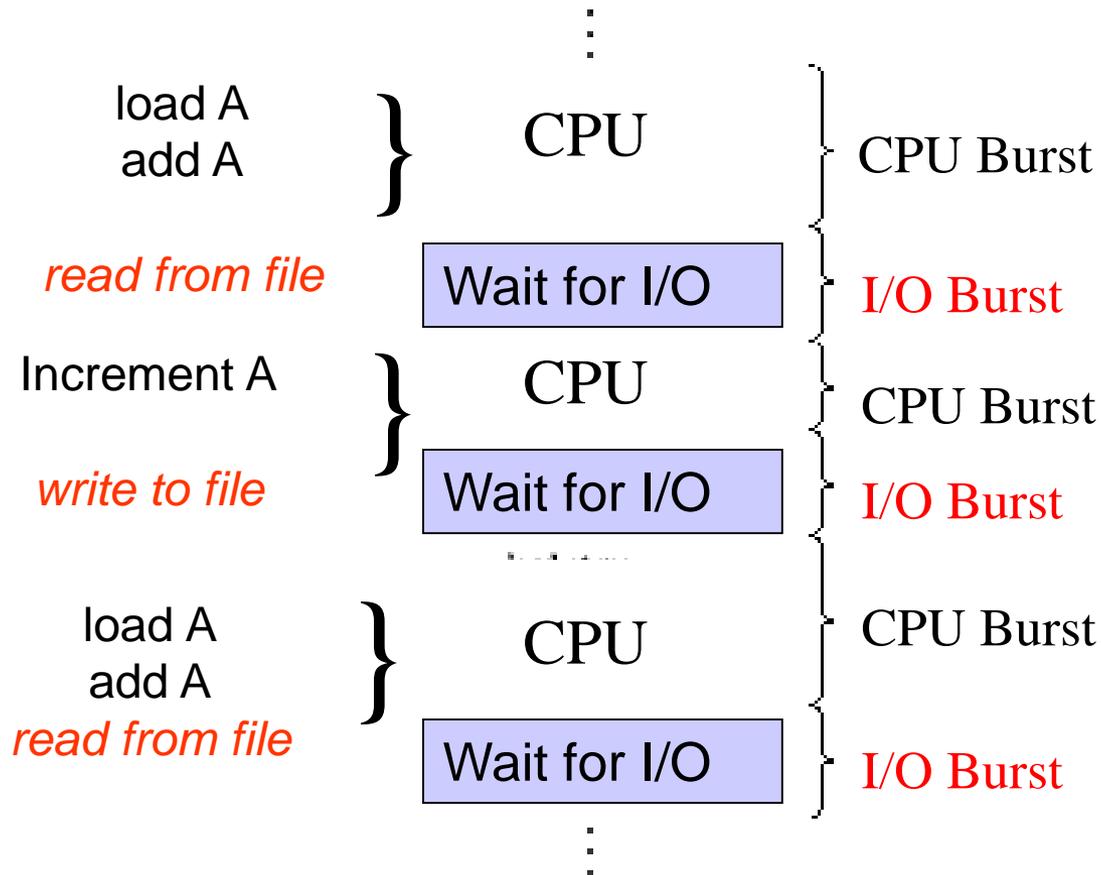
Assumptions for Process Scheduling

Basic assumptions behind most scheduling algorithms:

- In a single processor system, only one process can be in running state at a time
- In a multi-core/multi-processor system, many processes can be in running state at a time.
- In this course we are going to take care scheduling into a single processor.
- There is a pool of ready processes which compete for the CPU/CPU's or resources.
- The processes are either **dependent**/independent.
- In this course we are going to take care scheduling of independent processes.
- The job of the scheduler is to **distribute** the CPU time among different ready processes "**fairly**" (according to some definition of fairness) and in a way that **optimizes some performance criteria**.
 - i.e. to maximum CPU utilization , concurrent execution is recommended.
- **Key to the success of multiprocessing scheduling:**
 - Process execution consists of **cycles** of CPU execution and **cycles** of I/O waiting.
- CPU-I/O bound processes:
 - An **I/O-bound** process will have many I/O-bound instructions.
 - A **CPU-bound** process will have many long CPU-bound instructions.

Sequence of CPU and I/O Bursts

- A process will run for a while (CPU burst), perform some IO (I/O burst), then run for a while again (next CPU burst) and so on.
- The frequency of I/O and CPU operations depend on the process type.



Short-Term (CPU) Scheduler

- **CPU Scheduler:** is a module/process of the operating system
 - Is responsible for the selection of the next process to be executed.
- The selection is done according to a scheduling algorithm
 - **The scheduler** decision policy should be simple
 - **The scheduler** uses resources of the computer system,
 - In particular CPU time, memory
 - **The scheduler** should not consume too much CPU time
 - Otherwise the overhead is too high

Preemptive vs. Non-Preemptive Scheduler

Scheduling is Preemptive:

- If the CPU can be taken away from a process during its execution due to:
 - Occurrence of interrupts
 - Arrival of higher priority process
 - A change of status
 - Time limit
- Prevents a process from using the CPU for too long
- May lead to race conditions which will be solved by using process synchronization
- Supported by Windows, UNIX, Linux, Mac-OS 8 for the PowerPC platform

Scheduling is non-Preemptive:

- If, once the CPU has been allocated to a process, the process can keep the CPU until:
 - It releases the CPU, either by terminating or switching to the waiting state.
- Supported by old versions of Microsoft Windows and Mac OSs.
 - Simple, and easy to implement
 - May lead to starvation
 - Not suited for multi-user systems as it decreases the responsiveness.

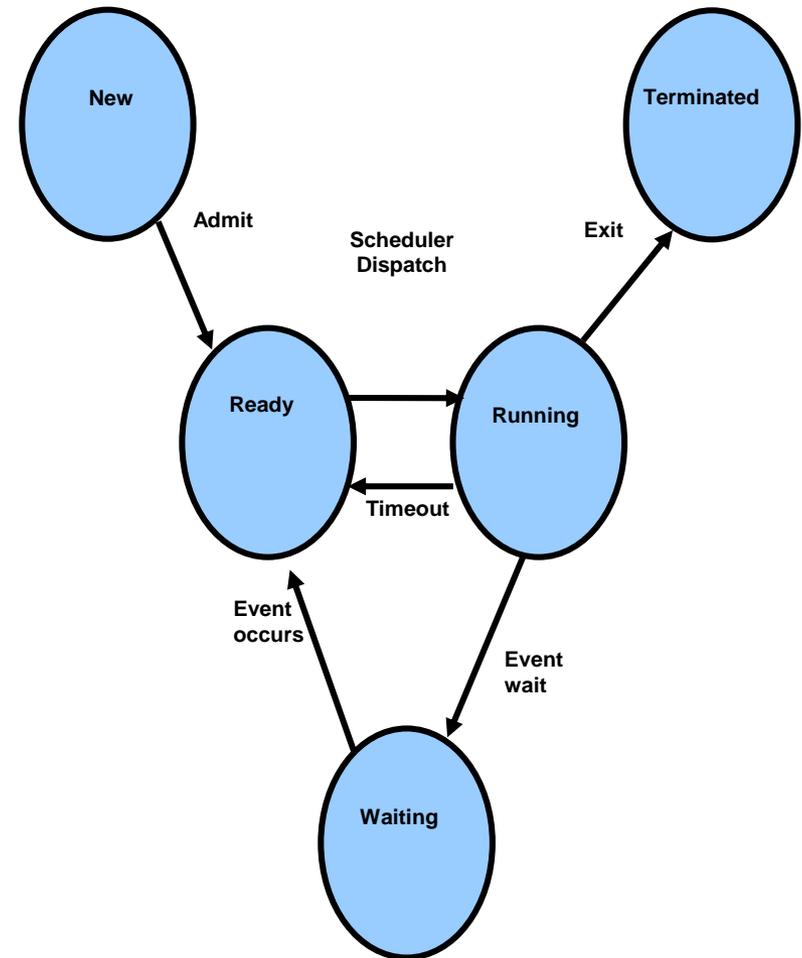
Scheduling Decision for Processes

- **Batch Process**
 - Users are not waiting at their terminals for a response time.
 - Should scheduling be preemptive or non-Preemptive?
- Since a batch process does not need any responsiveness, the OS can use non-preemptive and save the cost of context switch.
- **Interactive Process**
 - Environment has interactive users
 - Should scheduling be preemptive or non-Preemptive?
- **Real Time Process**
 - Processes may have real-time constraints
 - Should scheduling be preemptive or non-Preemptive?
- Since a real-time process needs more responsiveness, so it is recommended to be scheduled preemptively.

CPU or Short-Term Scheduler

- Selects one of the processes in the ready queue to be **allocated to the CPU**.
- Scheduling decisions may take place when a process:
 1. Switches from **running** to **waiting** state. **ex. I/O request.**
 2. Switches from **running** to **ready** state. **ex. Due to interrupt, time quantum expiring.**
 3. Switches from **waiting** to **ready**. **ex. Completion of I/O.**
 4. Switches from new to ready.
 5. Terminates.

– **Scheduling under 1 and 5 are non-preemptive while 2, 3 and 4 are preemptive.**



Evaluation Criteria of Scheduling Algorithms

- **Fairness**
 - Does the algorithm fairly share the CPU time among processes?
- **CPU utilization:** The fraction of time a CPU is executing processes. (ratio of in-use time/total observation time). Note that CPU might be busy but in context switching.
- **Resources utilization:** Efficient utilization of all resources, are all resources used?
- **Throughput:** # of processes that complete their execution per a time unit.
- **Turnaround Time:** Elapsed time from the submission/arrival of a process to its completion.
- **Waiting Time:** Amount of time a process has been waiting in the ready queue, not in the waiting I/O queues, may consist of several separate periods.
- **Response Time (interactive users):** Amount of time it takes from the submission until the first response is given to the process, not output (for time-sharing environment).
- **Context Switches:** # of context switches: Indication for the amount of overhead
- **Complexity of the scheduling algorithm**
 - Indication of the time needed to take a decision and select the next process.
 - It also indicates the amount of memory going to be used by the algorithm.

Average Waiting, Turn around and Response Time

- **Waiting Time**: how much time the process spends on the ready queue waiting to execute (this does not include time the process spends doing or waiting for I/O).
 - T_W : Amount of time a process has been waiting in the ready queue, it can be accumulated times, i.e. in round robin scheduling policy.
- **Waiting Time** starts when the process enters the ready queue (**not when it enters the system**).
- **Average Waiting Time** (AWT of n processes) = $\Sigma T_W / n$
- **Response Time** (of one process):
 - $T_R = T_{\text{Start to use CPU}} - T_{\text{Arrive}}$
- **Average Response Time** (ART of n processes) = $\Sigma T_R / n$
- **Turnaround Time** (of one process)
 - $T_T = T_{\text{Finish the process}} - T_{\text{Arrive}}$
- **Average Turnaround Time** (ATT of n processes) = $\Sigma T_T / n$

Ideal Process Scheduling Algorithm

- **Should Maximize**
 - CPU and resources utilization
 - Keep the CPU and resources busy all the time.
 - Throughput
 - # processes completed within a time unit.
- **Should Minimize**
 - Waiting Time:
 - Minimizes it leads to minimizing the Turnaround time.
 - Turnaround Time
 - Minimizes it leads to maximizing the throughput.
 - Response Time
 - Minimizes it leads to maximizing the user's satisfaction.

Factors in Scheduling

- The scheduler determines the order in which processes are going to be executed.
- Factors affect the scheduler's decision and implementation:
 - Which policy used to select a process? (will affect the response time)
 - How long may the process keep the CPU? (time burst/quantum)
 - How are conflicting requests resolved? (i.e. two processes have the same priority or same arrival time) while applying the priority or FCFS policies.
 - Is the process CPU- or I/O-bound?
 - Is the process interactive or batch?
 - Execution time used so far (historical time bursts)
 - Execution time required to complete (future time bursts)
 - Preemption frequency (how many time a process has been preempted?)
 - Page fault frequency (how many time a process has been swapped out and in?)
 - Processes dependency
 - etc..

First-Come, First-Serve (FCFS) Scheduling Policy

- The process that arrives first is allocated the CPU first.
- Simply implement the Ready queue as a FIFO queue.

Example

Process	Burst Time
P1	24
P2	3
P3	3

- Arrival order: P1 , P2 , P3
 - The Gantt Chart for scheduling is:



- Waiting time for P1 = 0; P2 = 24; P3 = 27
- $T_w = T_{\text{Start to use CPU}} - T_{\text{Arrive}}$
 - Average waiting time: $(0 + 24 + 27)/3 = 17$
 - Do you think, changing the order of processes arrival will affect the average waiting time and the throughput?

FCFS Scheduling Policy (Cont.)

- If the arrival order is: P2 , P3 , P1 .
 - The Gantt chart for scheduling is:



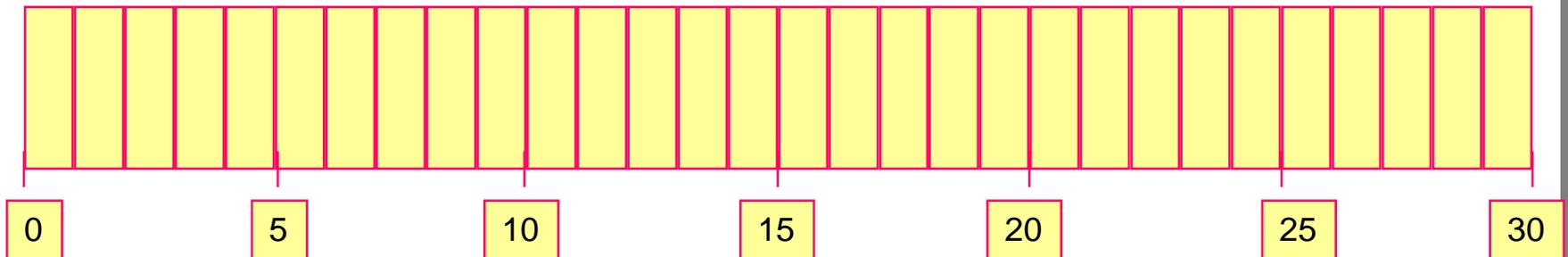
P	BT
P2	3
P3	3
P1	24

- Waiting time for P1 = 6; P2 = 0; P3 = 3
- $T_W = T_{\text{Start to use CPU}} - T_{\text{Arrive}}$
 - Average waiting time: $(6 + 0 + 3)/3 = 3$
 - Much better than previous case.
 - *At time slot 6, the throughput will be 2 but in the previous order, the throughput was 0, means more productive than the previous order.*

FCFS Policy Example 2

<i>Process #</i>	<i>Arrival Time</i>	<i>Burst Length</i>	<i>Priority</i>
P1	0	6	1
P2	0	15	1
P3	0	3	1
P4	0	4	1
P5	0	2	1

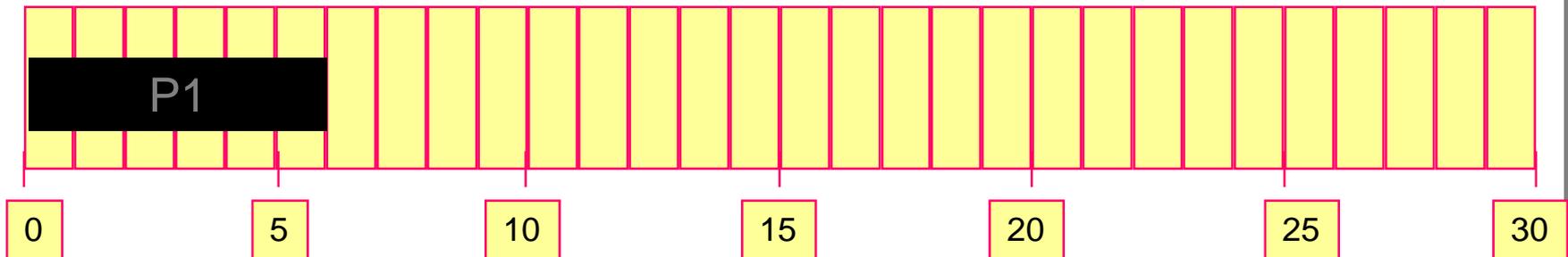
- Since the arrival time of all processes is 0 and all have the same priority, the order of processes will be used to resolve the conflict.



FCFS Policy Example 2

<i>Process #</i>	<i>Arrival Time</i>	<i>Burst Length</i>	<i>Priority</i>
P1	0	6	1
P2	0	15	1
P3	0	3	1
P4	0	4	1
P5	0	2	1

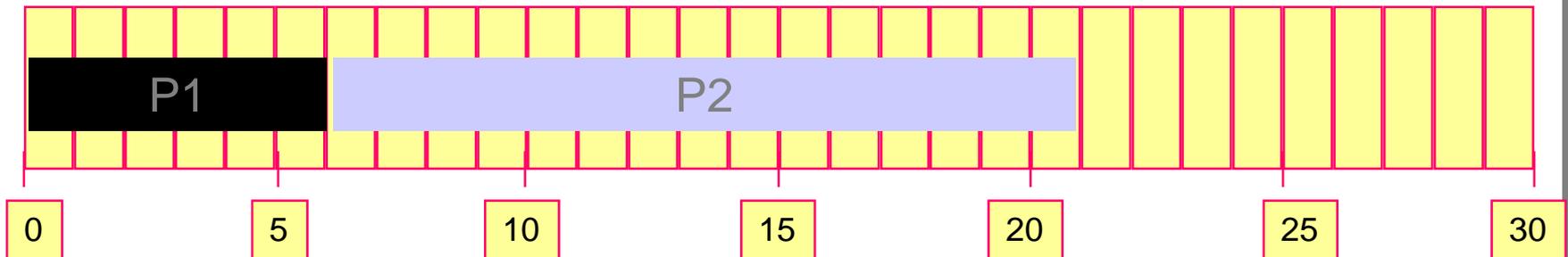
- Since the arrival time of all processes is 0 and all have the same priority, the order of processes will be used to resolve the conflict.



FCFS Policy Example 2

<i>Process #</i>	<i>Arrival Time</i>	<i>Burst Length</i>	<i>Priority</i>
P1	0	6	1
P2	0	15	1
P3	0	3	1
P4	0	4	1
P5	0	2	1

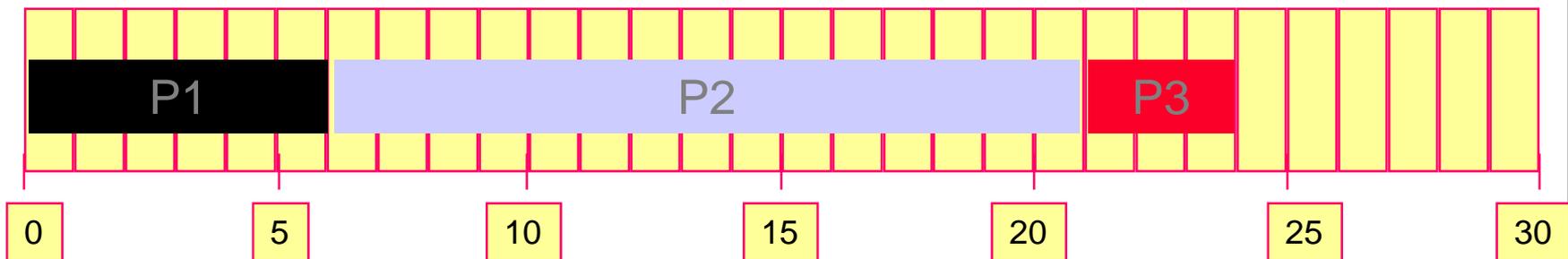
- Since the arrival time of all processes is 0 and all have the same priority, the order of processes will be used to resolve the conflict.



FCFS Policy Example 2

<i>Process #</i>	<i>Arrival Time</i>	<i>Burst Length</i>	<i>Priority</i>
P1	0	6	1
P2	0	15	1
P3	0	3	1
P4	0	4	1
P5	0	2	1

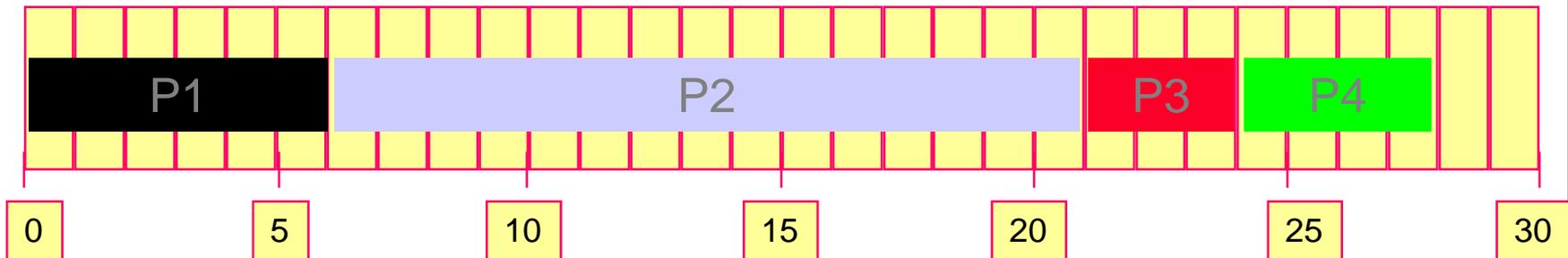
- Since the arrival time of all processes is 0 and all have the same priority, the order of processes will be used to resolve the conflict.



FCFS Policy Example 2

<i>Process #</i>	<i>Arrival Time</i>	<i>Burst Length</i>	<i>Priority</i>
P1	0	6	1
P2	0	15	1
P3	0	3	1
P4	0	4	1
P5	0	2	1

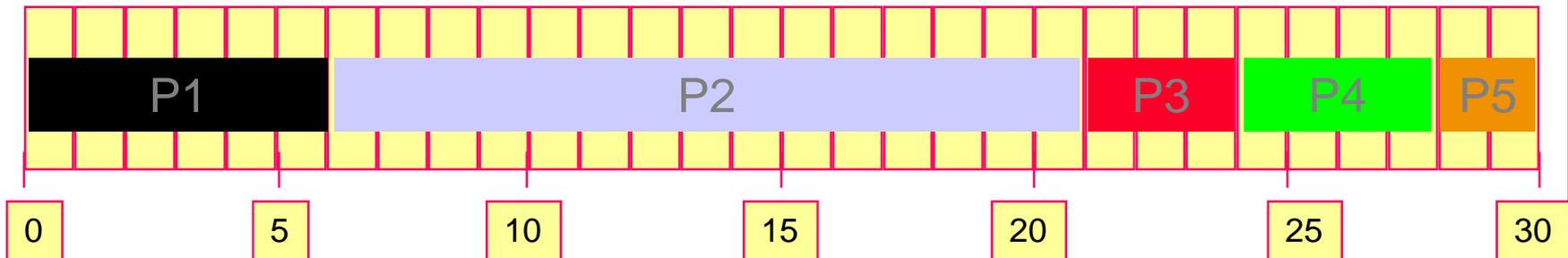
- Since the arrival time of all processes is 0 and all have the same priority, the order of processes will be used to resolve the conflict.



FCFS Policy Example 2

<i>Process #</i>	<i>Arrival Time</i>	<i>Burst Length</i>	<i>Priority</i>
P1	0	6	1
P2	0	15	1
P3	0	3	1
P4	0	4	1
P5	0	2	1

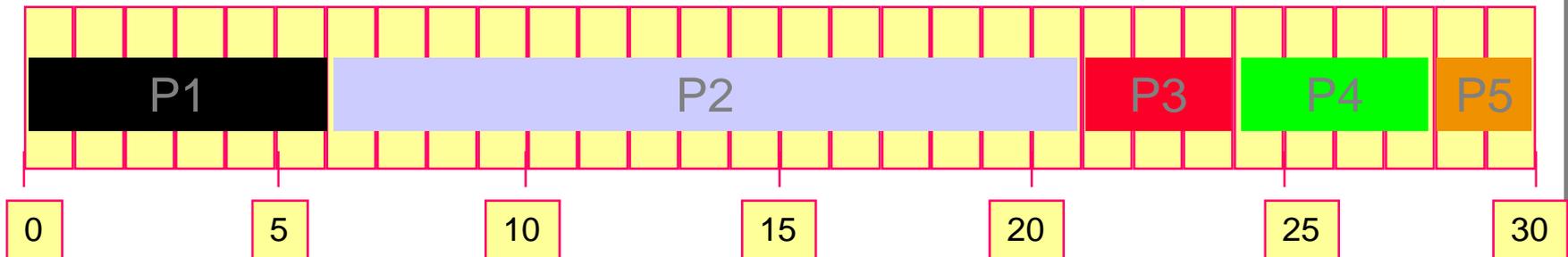
- Since the arrival time of all processes is 0 and all have the same priority, the order of processes will be used to resolve the conflict.



FCFS Policy Example 2

<i>Process #</i>	<i>Arrival Time</i>	<i>Burst Length</i>	<i>Priority</i>
P1	0	6	1
P2	0	15	1
P3	0	3	1
P4	0	4	1
P5	0	2	1

- Since the arrival time of all processes is 0 and all have the same priority, the order of processes will be used to resolve the conflict.



<i>Process #</i>	<i>Waiting Time</i>	<i>Response Time</i>	<i>Turnaround Time</i>	<i>#of Context Switches</i>
P1	0	0	6	1
P2	6	6	21	1
P3	21	21	24	1
P4	24	24	28	1
P5	28	28	30	1
<i>Average</i>	$79/5 = 15.8$	$79/5 = 15.8$	21.8	1

FCFS Policy Properties

- Very simple, easy to implement
 - Uses a FIFO queue
 - Very quick selection of the next process
 - Constant time, independent of the number of processes in the ready queue.
- Non-preemptive
- Often long average waiting and response times.
- Not suitable for multi-user systems!, Why?
- Since serving the short jobs first will improve the average waiting time which in fact will improve the throughput.
- Why do not we sort the jobs based on their burst lengths and then serve the shortest job first.

Shortest-Job-First (SJF) Policy

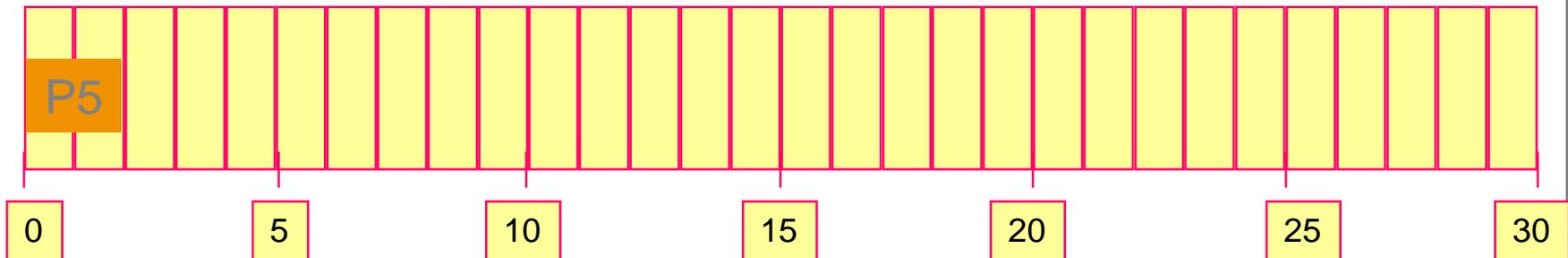
- **Shortest Job First (SJF)** or **Shortest Job Next (SJN)** or **Shortest Process Next (SPN)**: Select the waiting process with the smallest execution/burst time to execute first.
- How do we know the burst length of each process?
- Predict for each process the length of CPU time it needs. *One of its limitations.*
- Use these predicted lengths to schedule the process with the shortest time first.
- **Shortest-Job-First (SJF)**:
 - **It could be Non-Preemptive**: once CPU is given to the process, it cannot be preempted until the processes completes its burst time.
 - Do scheduling when a process **finishes**;
 - **Assume arrival time 0 for all processes!**, **practically is not.**
 - Assume the same priorities **for all processes!**, **practically is not.**
 - Different (expected) burst lengths, **how about if two have the same length.**
 - **It could be Preemptive**: upon arrival of a shorter process than the running one, **the CPU can be preempted and given to another process.**
- SJF is optimal: it gives minimum average waiting time for a given set of processes. Let us verify that.

SJF Policy Example

Process #	Arrival Time	Burst Length	Priority
P1	0	6	1
P2	0	15	1
P3	0	3	1
P4	0	4	1
P5	0	2	1

$$WT = T_{\text{Start to use CPU}} - T_{\text{Arrive}}$$

$$TT = T_{\text{Finish the process}} - T_{\text{Arrive}}$$

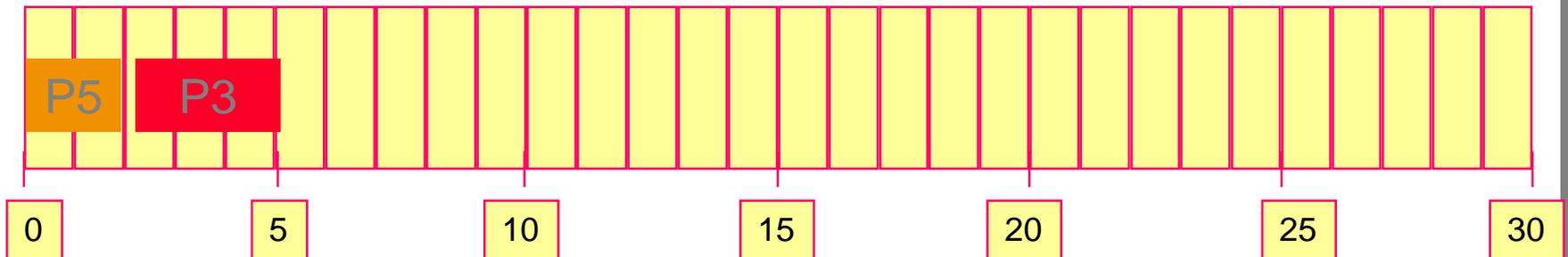


SJF Policy Example

Process #	Arrival Time	Burst Length	Priority
P1	0	6	1
P2	0	15	1
P3	0	3	1
P4	0	4	1
P5	0	2	1

$$WT = T_{\text{Start to use CPU}} - T_{\text{Arrive}}$$

$$TT = T_{\text{Finish the process}} - T_{\text{Arrive}}$$

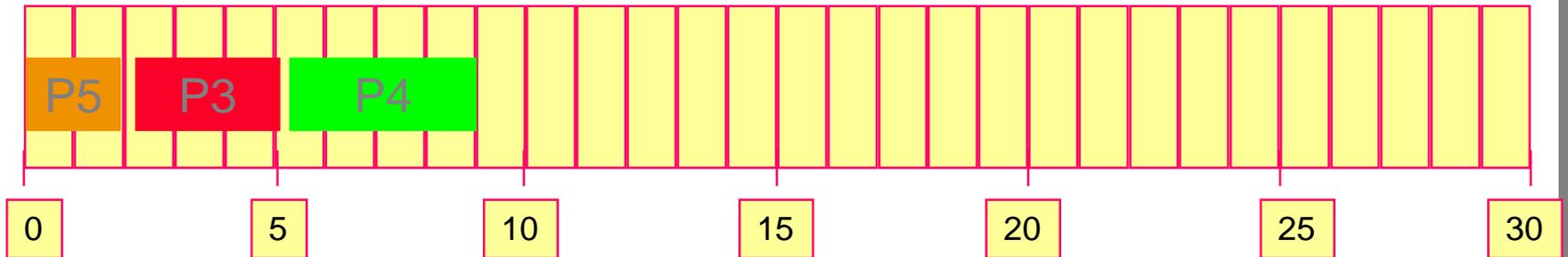


SJF Policy Example

Process #	Arrival Time	Burst Length	Priority
P1	0	6	1
P2	0	15	1
P3	0	3	1
P4	0	4	1
P5	0	2	1

$$WT = T_{\text{Start to use CPU}} - T_{\text{Arrive}}$$

$$TT = T_{\text{Finish the process}} - T_{\text{Arrive}}$$

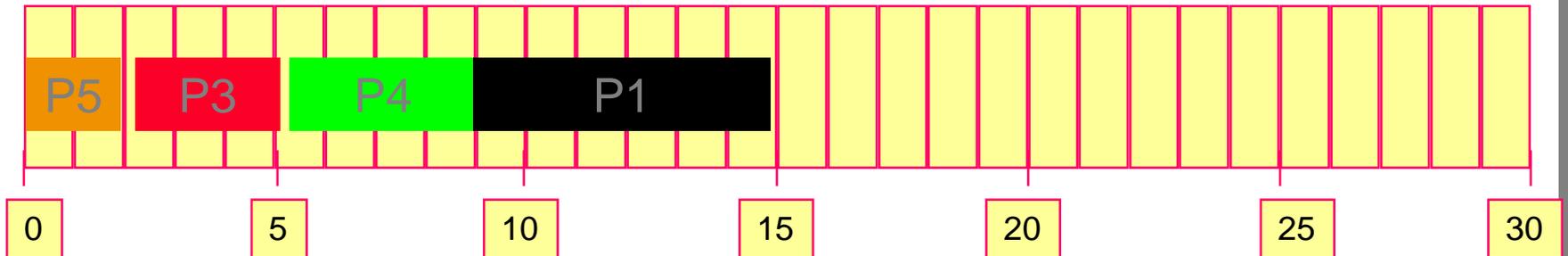


SJF Policy Example

<i>Process #</i>	<i>Arrival Time</i>	<i>Burst Length</i>	<i>Priority</i>
P1	0	6	1
P2	0	15	1
P3	0	3	1
P4	0	4	1
P5	0	2	1

$$WT = T_{\text{Start to use CPU}} - T_{\text{Arrive}}$$

$$TT = T_{\text{Finish the process}} - T_{\text{Arrive}}$$

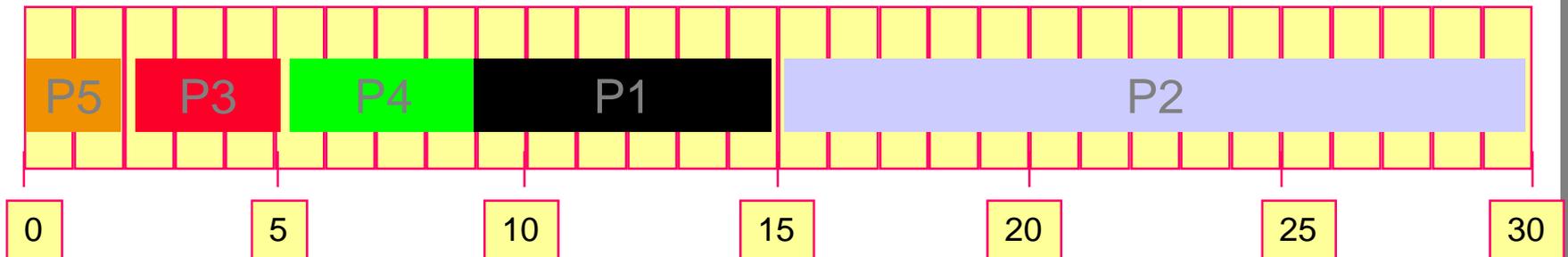


SJF Policy Example

Process #	Arrival Time	Burst Length	Priority
P1	0	6	1
P2	0	15	1
P3	0	3	1
P4	0	4	1
P5	0	2	1

$$WT = T_{\text{Start to use CPU}} - T_{\text{Arrive}}$$

$$TT = T_{\text{Finish the process}} - T_{\text{Arrive}}$$

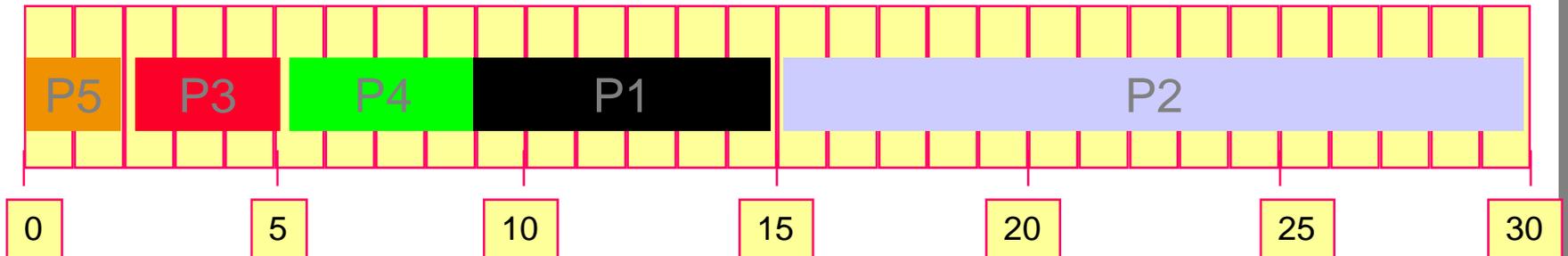


SJF Policy Example

<i>Process #</i>	<i>Arrival Time</i>	<i>Burst Length</i>	<i>Priority</i>
P1	0	6	1
P2	0	15	1
P3	0	3	1
P4	0	4	1
P5	0	2	1

$$WT = T_{\text{Start to use CPU}} - T_{\text{Arrive}}$$

$$TT = T_{\text{Finish the process}} - T_{\text{Arrive}}$$



<i>Process #</i>	<i>Waiting Time</i>	<i>Response Time</i>	<i>Turnaround Time</i>	<i>#of Context Switches</i>
P1	9	9	15	1
P2	15	15	30	1
P3	2	2	6	1
P4	5	5	9	1
P5	0	0	2	1
<i>Average</i>	$31/5 = 6.2$	$31/5 = 6.2$	$62/5 = 12.4$	1

SJF: Different Arrival Times

- Slight modification
 - Different arrival times for the processes
 - Same priorities
 - Different (**predicted**) burst lengths
 - The processes waiting in the ready queue are added to the diagram (**select the shortest among processes waiting in the ready queue**). Non-arrived processes should not be counted when deciding the shortest one.

SJF Example 1: Different Arrival Times

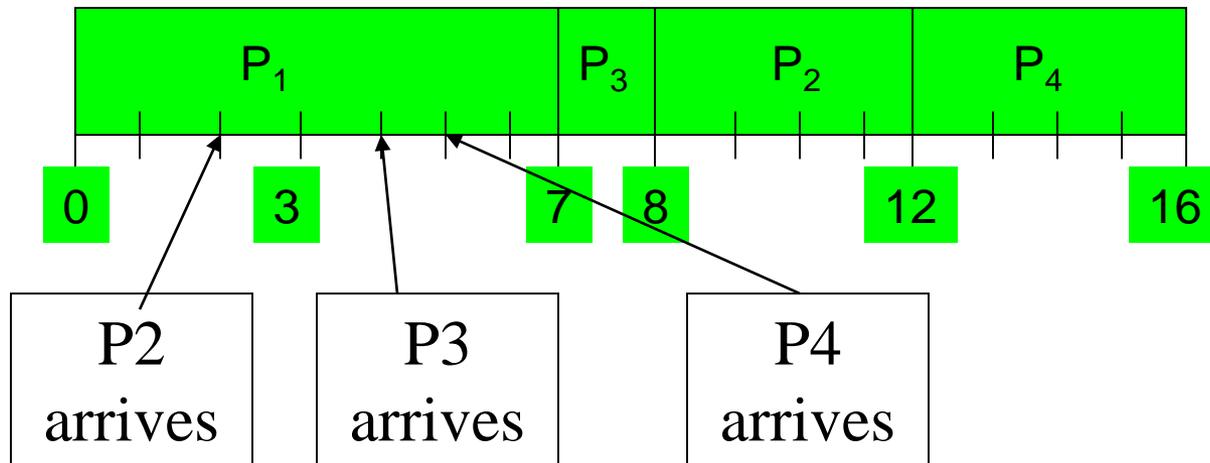
<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P ₁	0.0	7
P ₂	2.0	4
P ₃	4.0	1
P ₄	5.0	4

- $T_W = T_{\text{Start to use CPU}} - T_{\text{Arrive}}$

$$AWT = ((0-0) + (8-2) + (7-4) + (12-5))/4 = 4$$

- $T_T = T_{\text{Finish the job}} - T_{\text{Arrive}}$

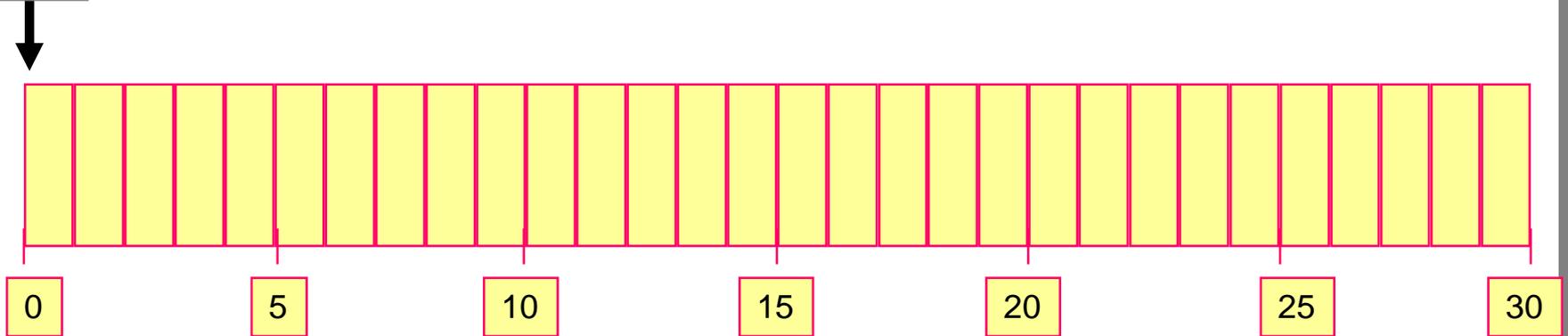
$$ATT = ((7-0) + (12-2) + (8-4) + (16-5))/4 = 8$$



SJF Example 2: Different Arrival Times

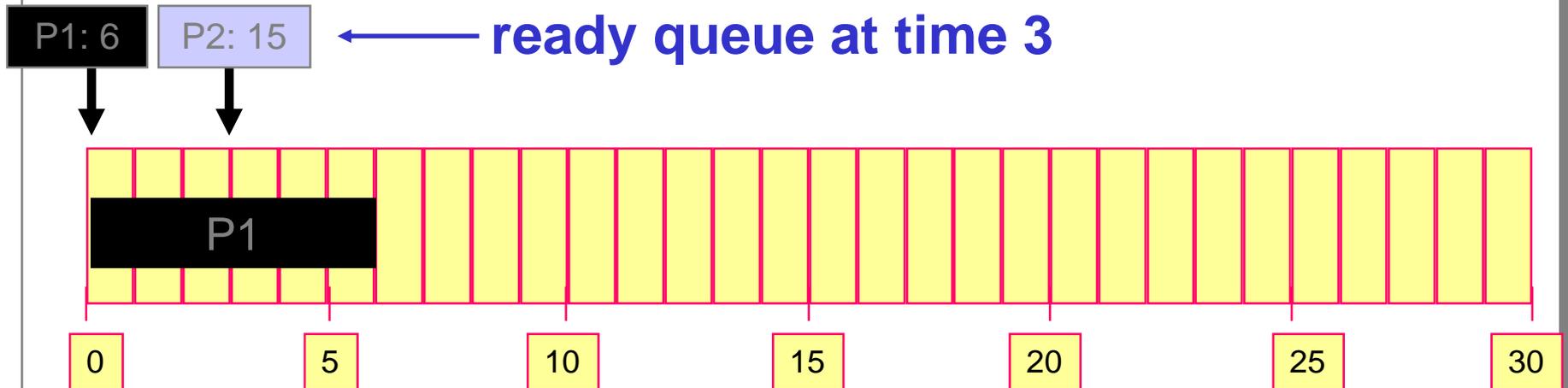
<i>Process #</i>	<i>Arrival Time</i>	<i>Burst Length</i>	<i>Priority</i>
P1	0	6	1
P2	3	15	1
P3	5	3	1
P4	8	4	1
P5	14	2	1

P1: 6 ← ready queue at time 0



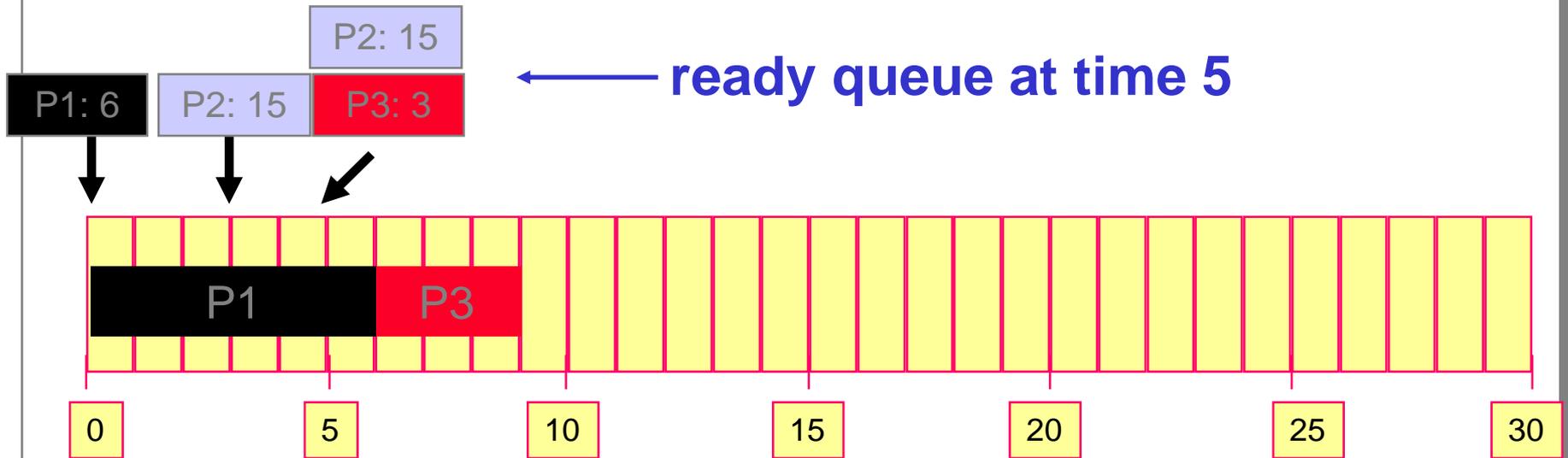
SJF Example 2: Different Arrival Times

Process #	Arrival Time	Burst Length	Priority
P1	0	6	1
P2	3	15	1
P3	5	3	1
P4	8	4	1
P5	14	2	1



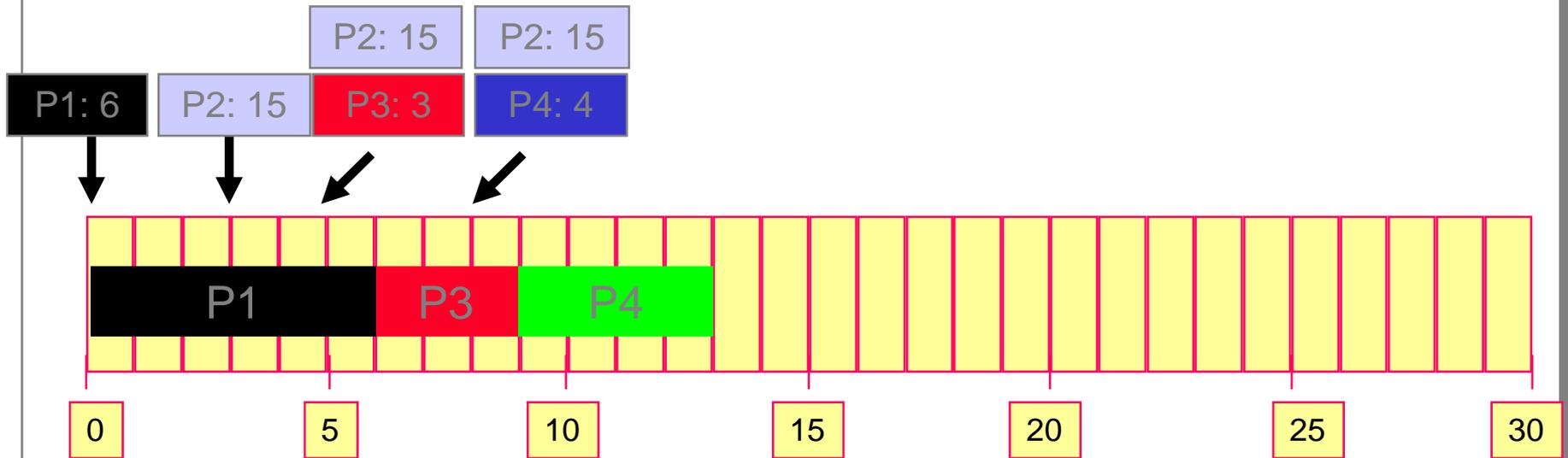
SJF Example 2: Different Arrival Times

Process #	Arrival Time	Burst Length	Priority
P1	0	6	1
P2	3	15	1
P3	5	3	1
P4	8	4	1
P5	14	2	1



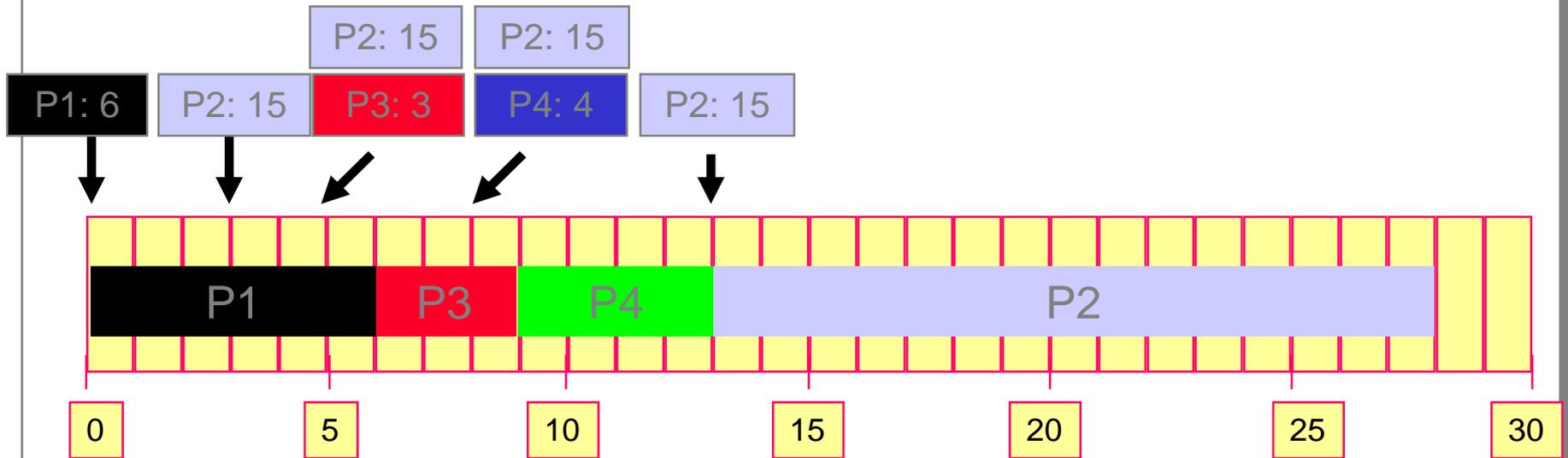
SJF Example 2: Different Arrival Times

<i>Process #</i>	<i>Arrival Time</i>	<i>Burst Length</i>	<i>Priority</i>
P1	0	6	1
P2	3	15	1
P3	5	3	1
P4	8	4	1
P5	14	2	1



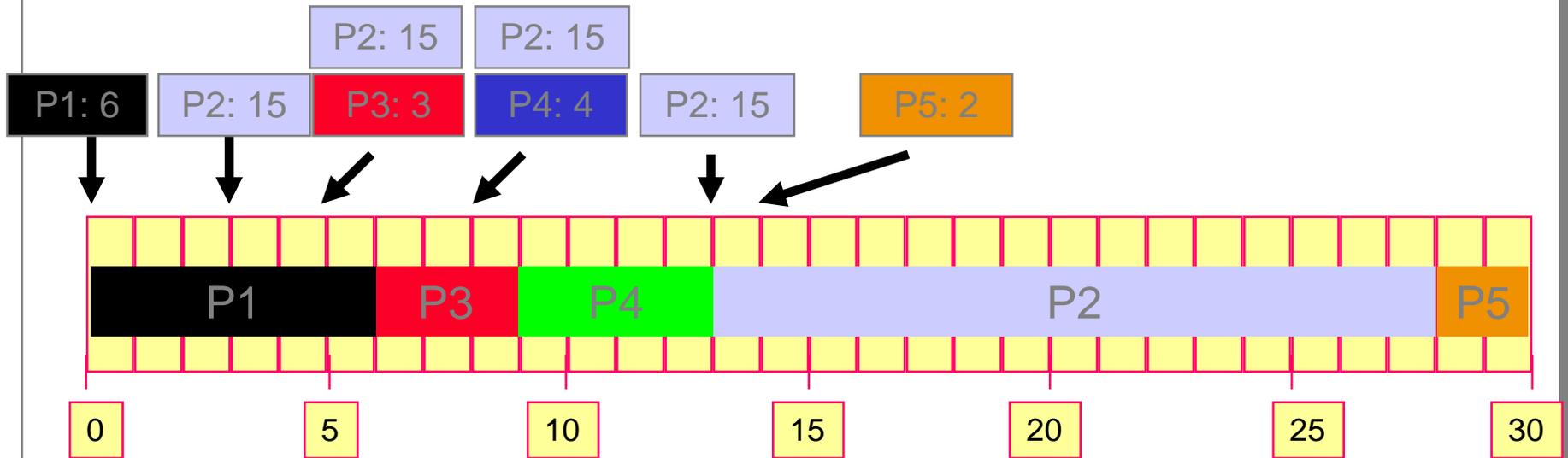
SJF Example 2: Different Arrival Times

<i>Process #</i>	<i>Arrival Time</i>	<i>Burst Length</i>	<i>Priority</i>
P1	0	6	1
P2	3	15	1
P3	5	3	1
P4	8	4	1
P5	14	2	1



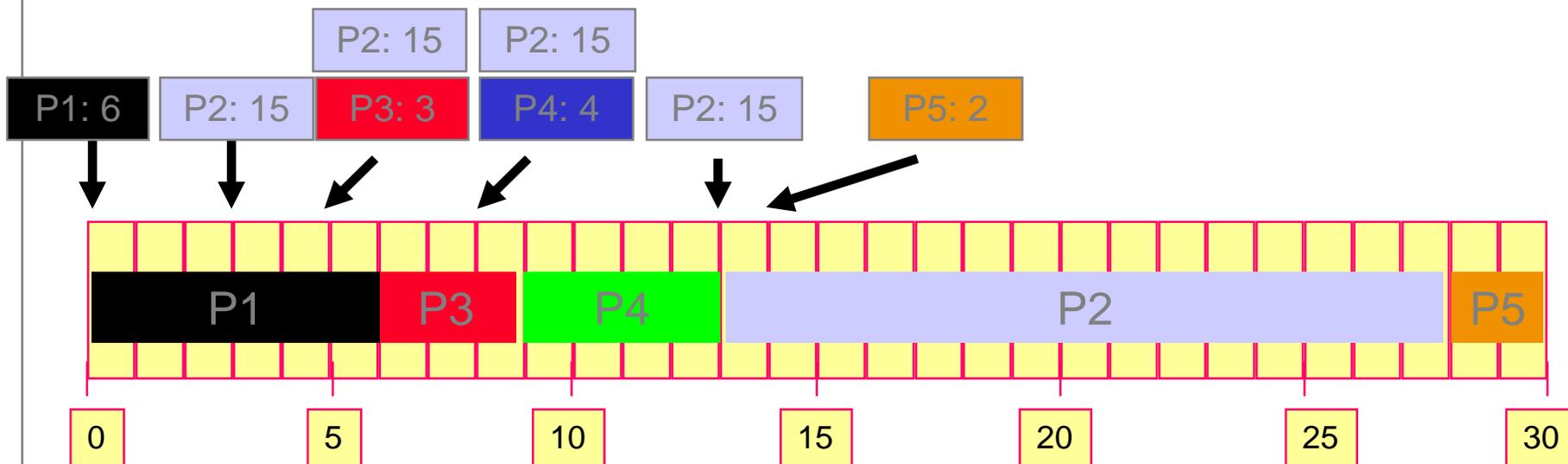
SJF Example 2: Different Arrival Times

Process #	Arrival Time	Burst Length	Priority
P1	0	6	1
P2	3	15	1
P3	5	3	1
P4	8	4	1
P5	14	2	1



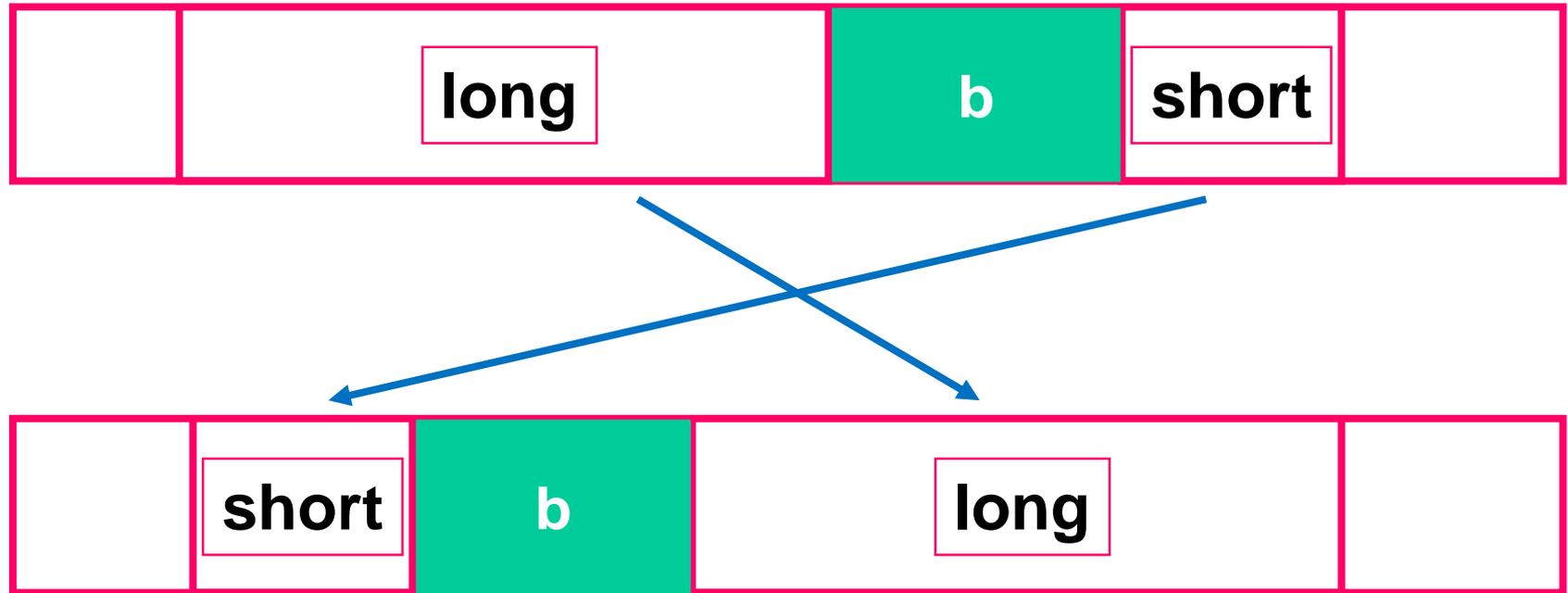
SJF Example 2: Different Arrival Times

Process #	Arrival Time	Burst Length	Priority
P1	0	6	1
P2	3	15	1
P3	5	3	1
P4	8	4	1
P5	14	2	1



Process #	Waiting Time	Response Time	Turnaround Time	#of Context Switches
P1	0	0	6	1
P2	$13 - 3 = 10$	$13 - 3 = 10$	$28 - 3 = 25$	1
P3	$6 - 5 = 1$	$6 - 5 = 1$	$9 - 5 = 4$	1
P4	$9 - 8 = 1$	$9 - 8 = 1$	$13 - 8 = 5$	1
P5	$28 - 14 = 14$	$28 - 14 = 14$	$30 - 14 = 16$	1
Average	$26/5 = 5.2$	$26/5 = 5.2$	$50/5 = 10$	1

SJF Optimality



- This graph Proofs that the SJF algorithms is optimal
 - Better responsiveness
 - Minimum average waiting time
 - Improves the throughput
- What is the difficulties of theses algorithms?

SJF Policy Properties

- Much better average waiting time for processes when we use SJF than FCFS.
 - Optimal with respect to the average waiting time.
- Non-preemptive
- Relies on knowing the length of the CPU bursts
 - In general, it is difficult to impossible to get it accurately.
- Implementation is more complex than FCFS
- Time of selecting a process is variant.
 - Linear w.r.t. number of processes in the ready queue
- Impractical due to burst length prediction problem
- Starvation is possible
 - If new, short processes keep on arriving, long processes may never be served.
- How do we overcome this problem?

Ch: 5 Process Scheduling

- Schedulers Overview (Short, Medium and Long)
- Time Quantum/Burst Time and Context Switch
- Scheduler, Dispatcher and Swapper modules in the OS
- Basic Assumptions for Process Scheduling
- Types of Short-term Scheduling Algorithms (Preemptive, and Non-preemptive)
 - Scheduling decision and the Types of processes
- Evaluation Criteria of Scheduling Algorithms
- Scheduling Algorithms
 - First-Come, First-Served (FCFS)
 - Shortest Job First (SJF)
 - Shortest Remaining Time First (SRTF)
 - Process Burst Length Prediction
 - Round Robin
 - Priority-Based
 - Multilevel Queue
 - Multilevel Feedback Queue
- Scheduling Algorithms Evaluation
 - Deterministic and Queuing models
 - Simulations, and Implementation
- Scheduling Polices in Different OS
- Multiprocessor Systems: Just Introduction, Scheduling of Multiprocessors systems

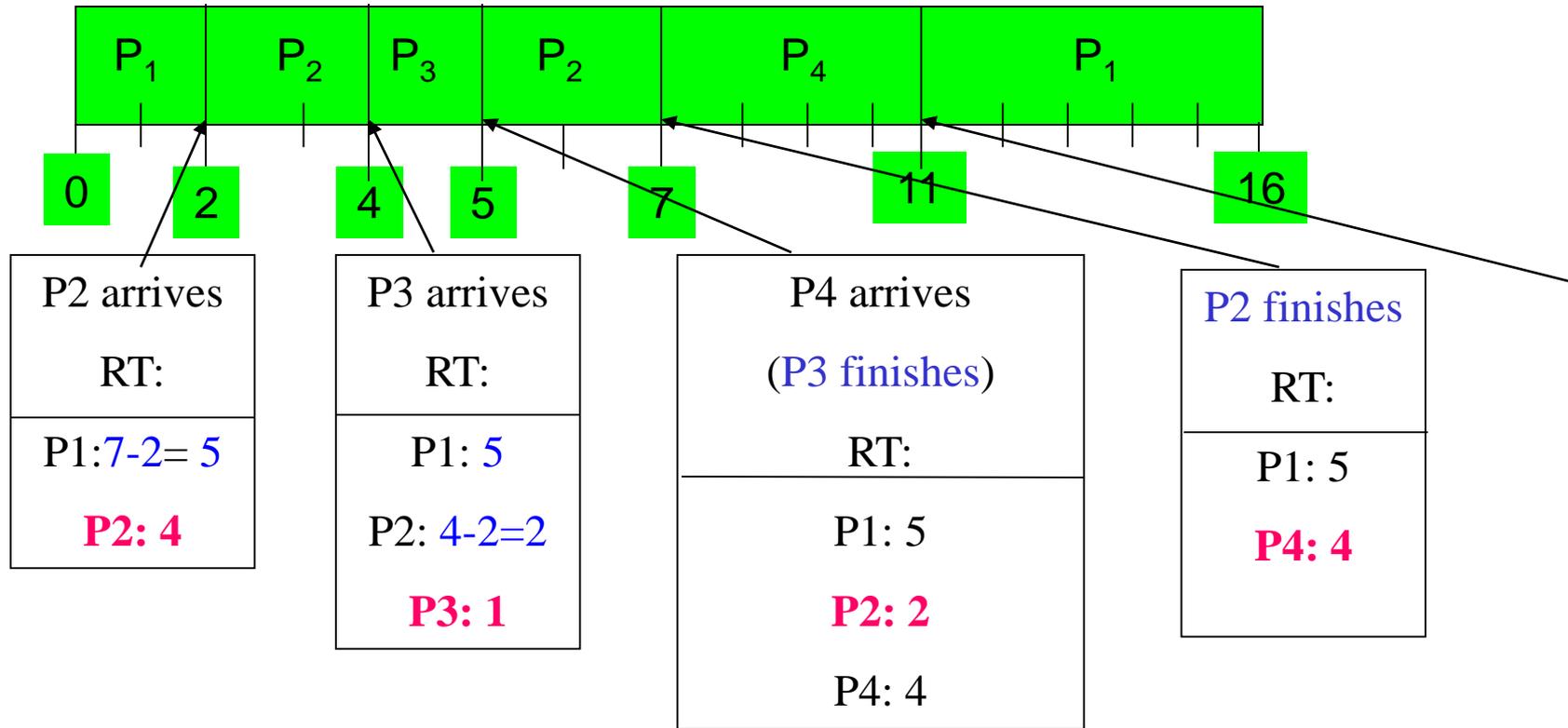
Shortest-Remaining-Time-First (SRTF) Policy

Shortest-Remaining-Time-First (SRTF)

- **Preemptive:** If a new process arrives with burst length less than remaining time of current executing process, **preempt and make a context switch**.
- **Remaining time:** (The process burst length - the time the CPU has already spent serving this process).
- **A scheduling decision must be made when**
 - A process is done with its CPU burst
 - A new process arrives in the ready queue
- **Arrival time of new processes is important**
- It is important to keep track of the processes currently in the ready queue
 - **The policy used here:** preempted processes go to the end of the ready queue.
 - It may depend on another policy for implementation.

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P ₁	0.0	7
P ₂	2.0	4
P ₃	4.0	1
P ₄	5.0	4

Shortest-Remaining-Time-First

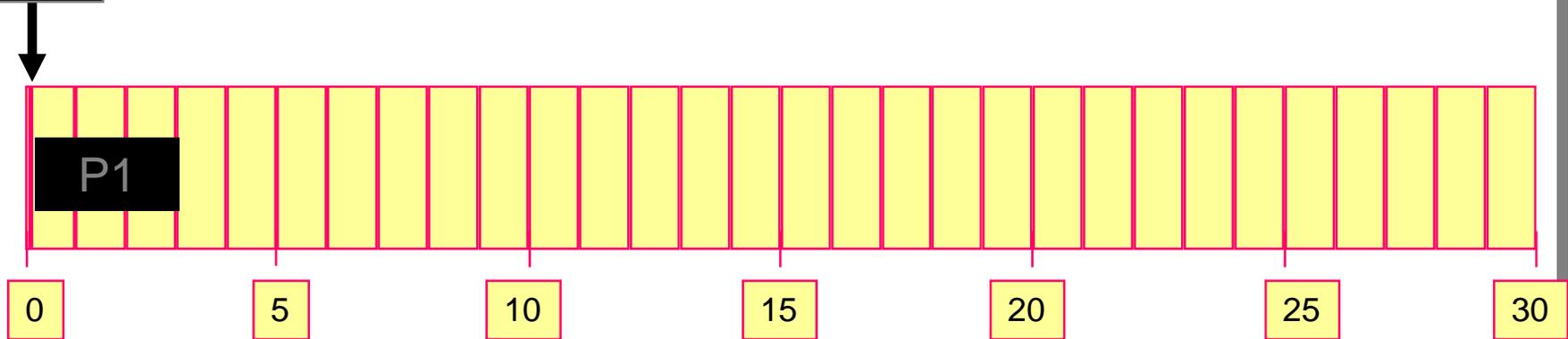


- $AWT = \Sigma (T_{\text{Start to use CPU}} - T_{\text{Arrive}}) / n = (9 + 1 + 0 + 2) / 4 = 3$
- $ATT = \Sigma (T_{\text{Finish the job}} - T_{\text{Arrive}}) / n = (16 + 5 + 1 + 6) / 4 = 7$

SRTF Policy Example

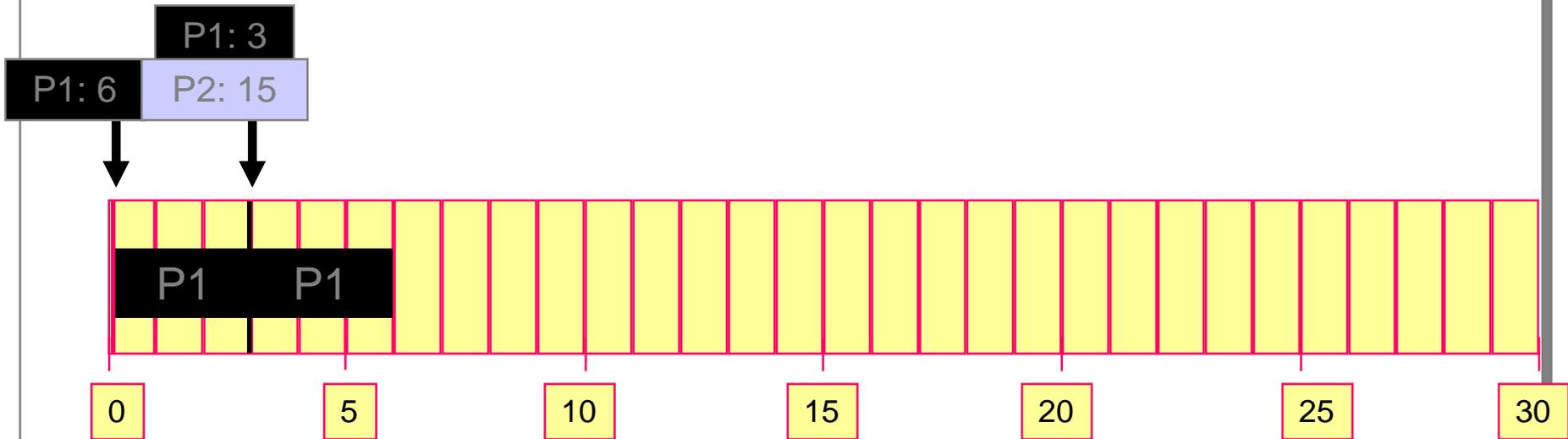
<i>Process #</i>	<i>Arrival Time</i>	<i>Burst Length</i>	<i>Priority</i>
P1	0	6	1
P2	3	15	1
P3	9	3	1
P4	14	4	1
P5	17	2	1

P1: 6



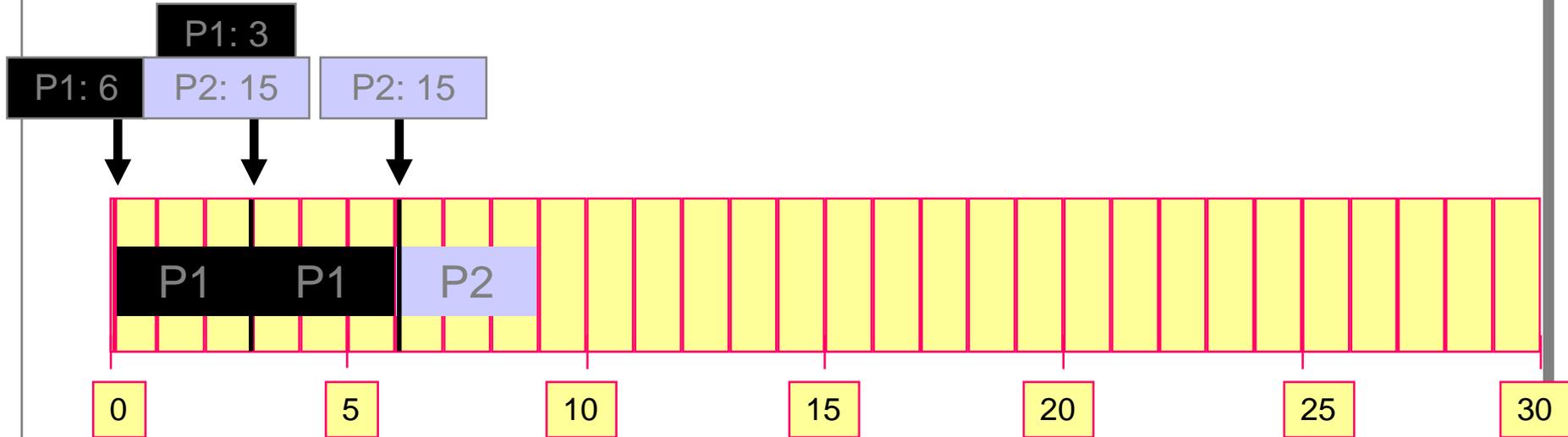
SRTF Policy Example

<i>Process #</i>	<i>Arrival Time</i>	<i>Burst Length</i>	<i>Priority</i>
P1	0	6	1
P2	3	15	1
P3	9	3	1
P4	14	4	1
P5	17	2	1



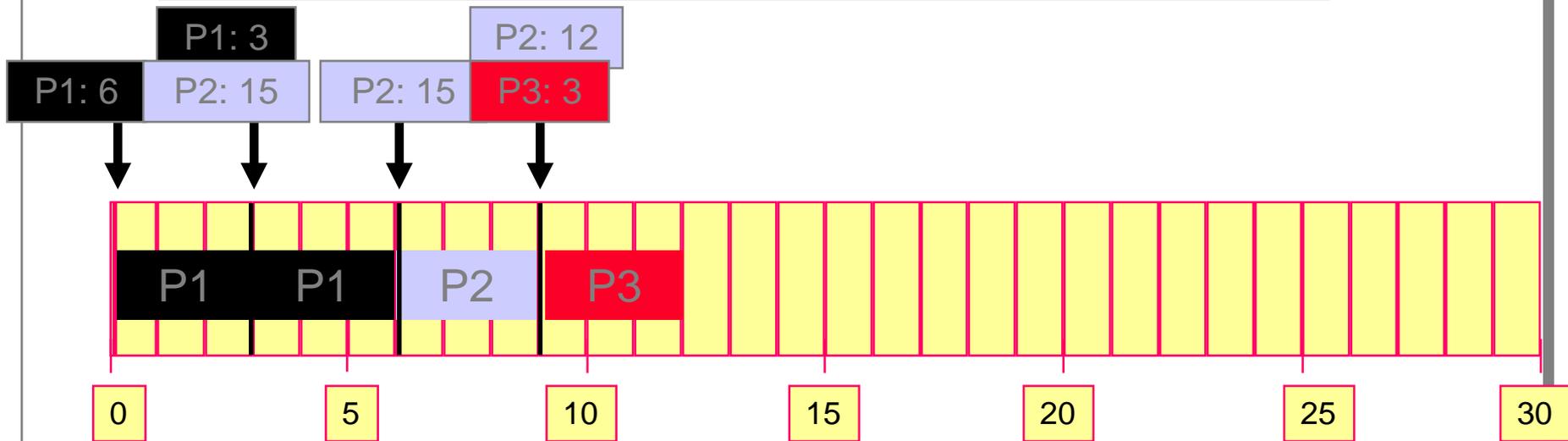
SRTF Policy Example

<i>Process #</i>	<i>Arrival Time</i>	<i>Burst Length</i>	<i>Priority</i>
P1	0	6	1
P2	3	15	1
P3	9	3	1
P4	14	4	1
P5	17	2	1



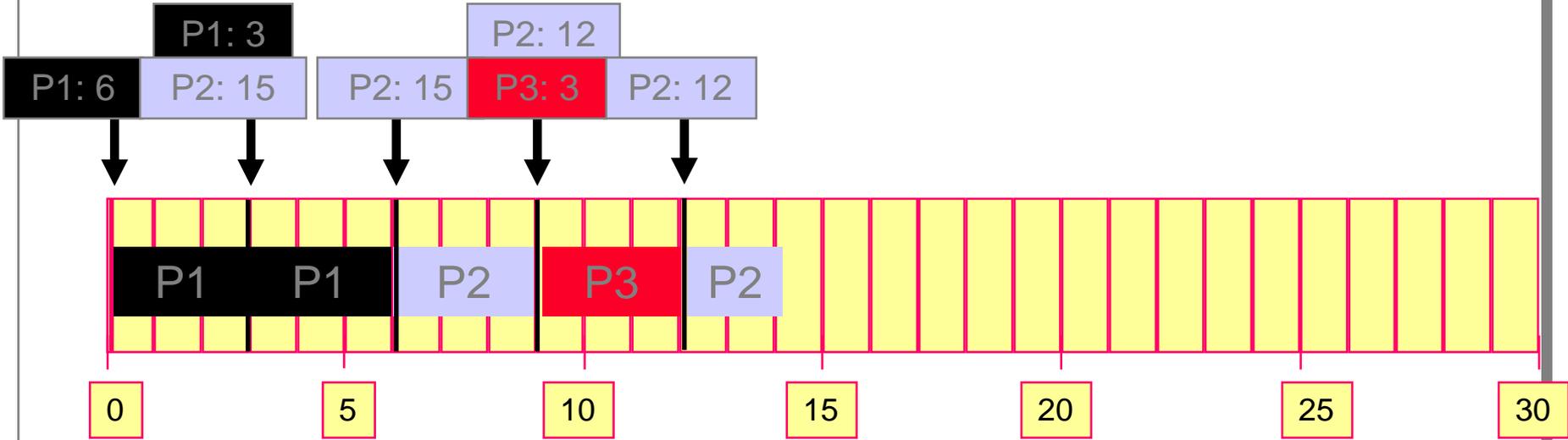
SRTF Policy Example

<i>Process #</i>	<i>Arrival Time</i>	<i>Burst Length</i>	<i>Priority</i>
P1	0	6	1
P2	3	15	1
P3	9	3	1
P4	14	4	1
P5	17	2	1



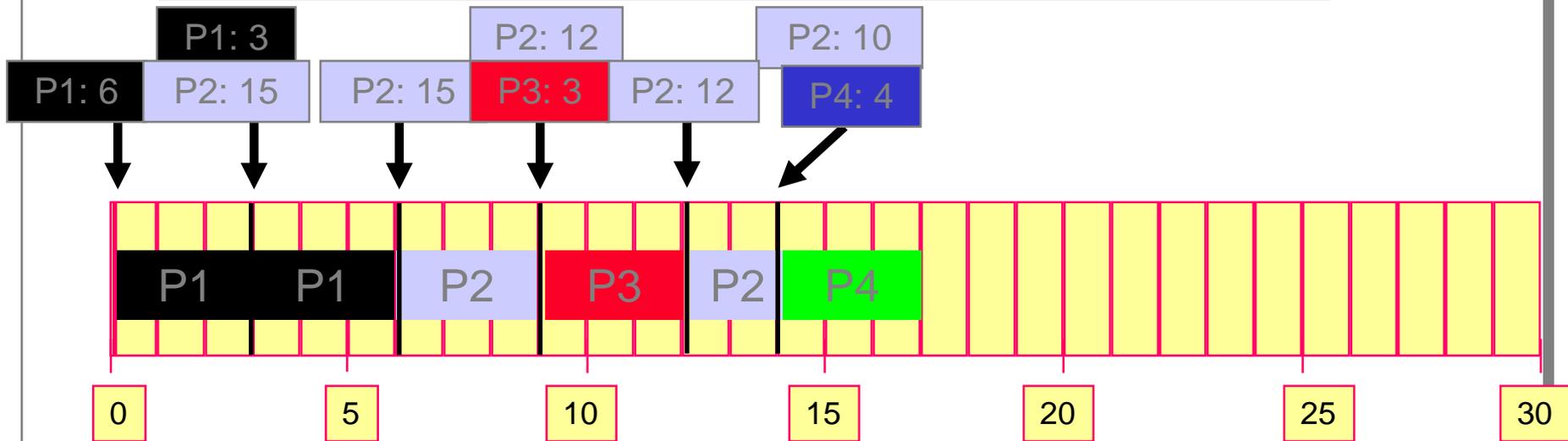
SRTF Policy Example

<i>Process #</i>	<i>Arrival Time</i>	<i>Burst Length</i>	<i>Priority</i>
P1	0	6	1
P2	3	15	1
P3	9	3	1
P4	14	4	1
P5	17	2	1



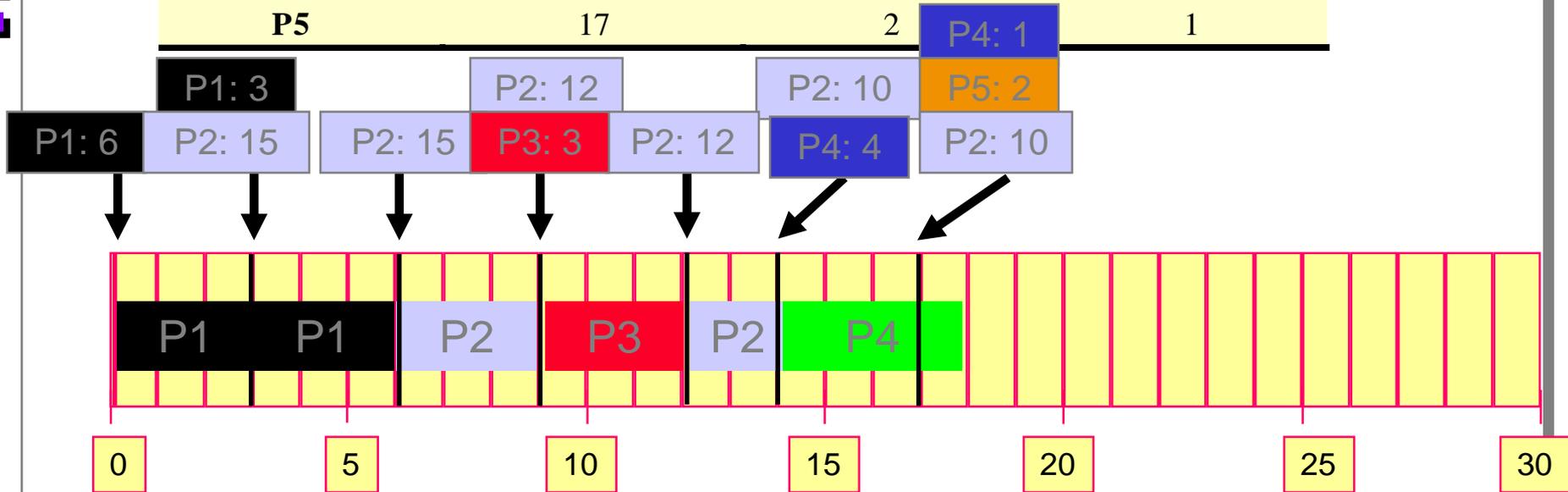
SRTF Policy Example

Process #	Arrival Time	Burst Length	Priority
P1	0	6	1
P2	3	15	1
P3	9	3	1
P4	14	4	1
P5	17	2	1



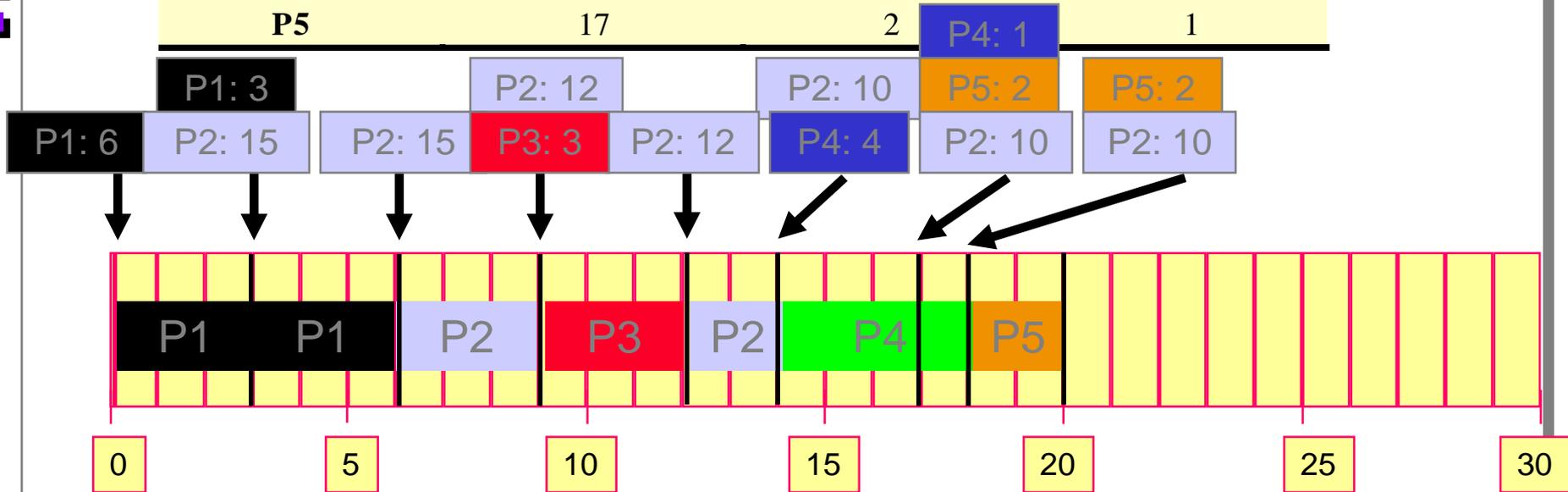
SRTF Policy Example

Process #	Arrival Time	Burst Length	Priority
P1	0	6	1
P2	3	15	1
P3	9	3	1
P4	14	4	1
P5	17	2	1



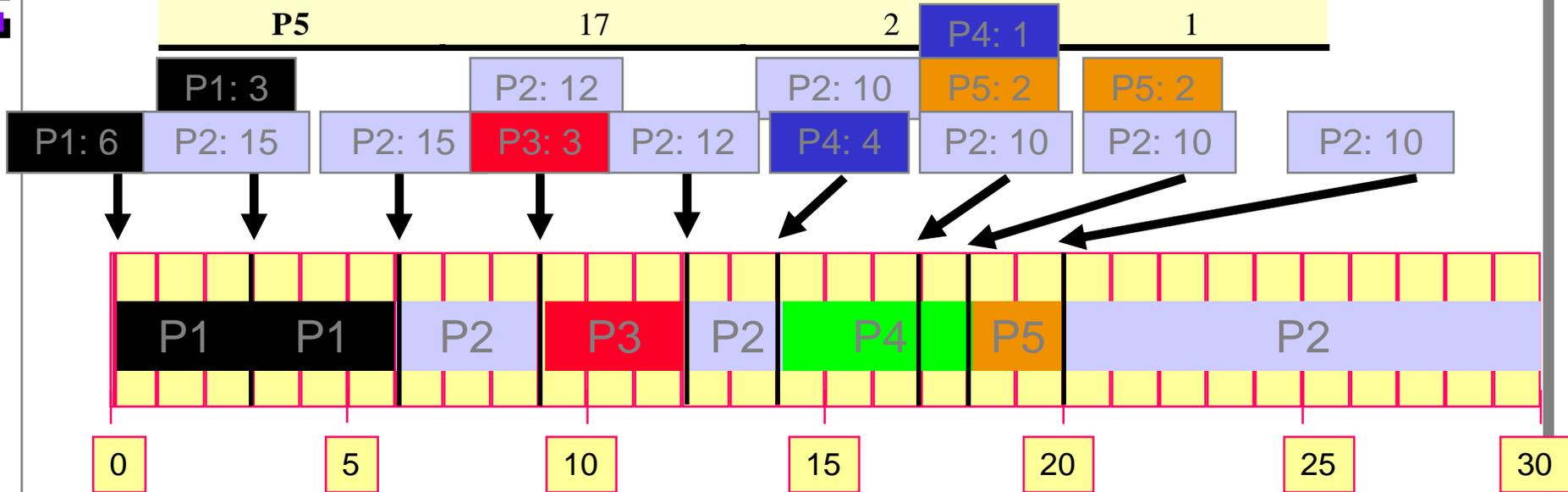
SRTF Policy Example

Process #	Arrival Time	Burst Length	Priority
P1	0	6	1
P2	3	15	1
P3	9	3	1
P4	14	4	1
P5	17	2	1



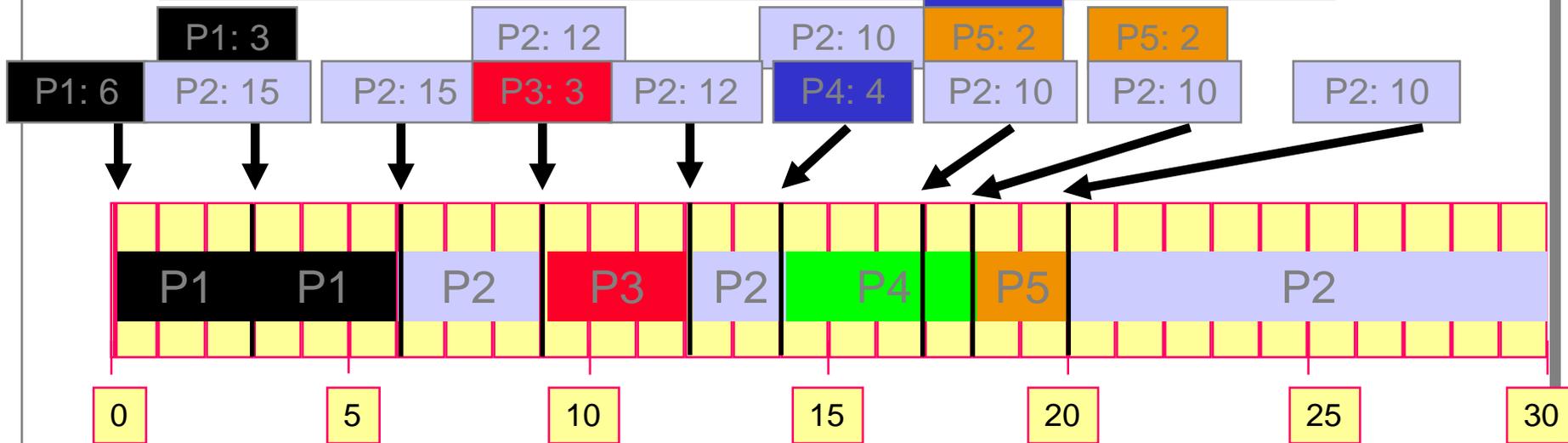
SRTF Policy Example

Process #	Arrival Time	Burst Length	Priority
P1	0	6	1
P2	3	15	1
P3	9	3	1
P4	14	4	1
P5	17	2	1



SRTF Policy Example

Process #	Arrival Time	Burst Length	Priority
P1	0	6	1
P2	3	15	1
P3	9	3	1
P4	14	4	1
P5	17	2	1



Process #	Waiting Time	Response Time	Turnaround Time	#of Context Switches
P1	0	0	6	1 (2)
P2	$(6-3) + (12-9) + (20-14) = 12$	$(6-3) = 3$	$(30 - 3) = 27$	3
P3	0	0	$(12-9) = 3$	1
P4	0	0	$(18-14) = 4$	1 (2)
P5	$(18-17) = 1$	$(18-17) = 1$	$(20-17) = 3$	1
Average	$13/5 = 2.6$	$13/5 = 2.6$	$36/5 = 7.2$	7

SRTF Policy Properties

- Good response time for short processes
 - Attractive for multi-user systems
- Preemptive version of the SJF algorithm
 - Higher overhead (context switches)
- Starvation possible
- Impractical due to burst length prediction problem

FCFS, SJF, SRTF Policies **Comparison**

- With the same set of processes, we can compare the performance of the FCFS, SJF and SRTF.

<i>Process #</i>	<i>Arrival Time</i>	<i>Burst Length</i>	<i>Priority</i>
P1	0	6	1
P2	3	15	1
P3	9	3	1
P4	14	4	1
P5	17	2	1

	FCFS	SJF	SJF (different arrival times)	SRTF
AWT	15.8	6.2	5.2	2.6
ART	15.8	6.2	5.2	2.6
ATT	21.8	12.4	10	7.2

- What does the lowest AWT, ART, and ATT mean?
- Is it supporting the optimality of the algorithms?
- What is the main headache of this algorithm?

Burst Length Prediction

- Practically, the CPU burst length of a process is not known and needs to be predicted/estimated.
 - Algorithms such as SJF, SRTF in their pure form can't be used in practical systems **unless there is an algorithm to predict the next process burst length**. How?
- **The past can be a good predictor of the future, moreover**
 - Recent bursts might be given more importance than older bursts.
- The CPU burst length can be estimated **based on the previous CPU bursts of the process**.
- This requires analysis of the code to be executed while the scheduling decision is made.
- That means, additional overhead during the scheduling decision
 - Time to estimate next CPU burst of the process and calculate the remaining time.
 - Memory space to keep values of recent CPU burst lengths.

Burst Length Prediction

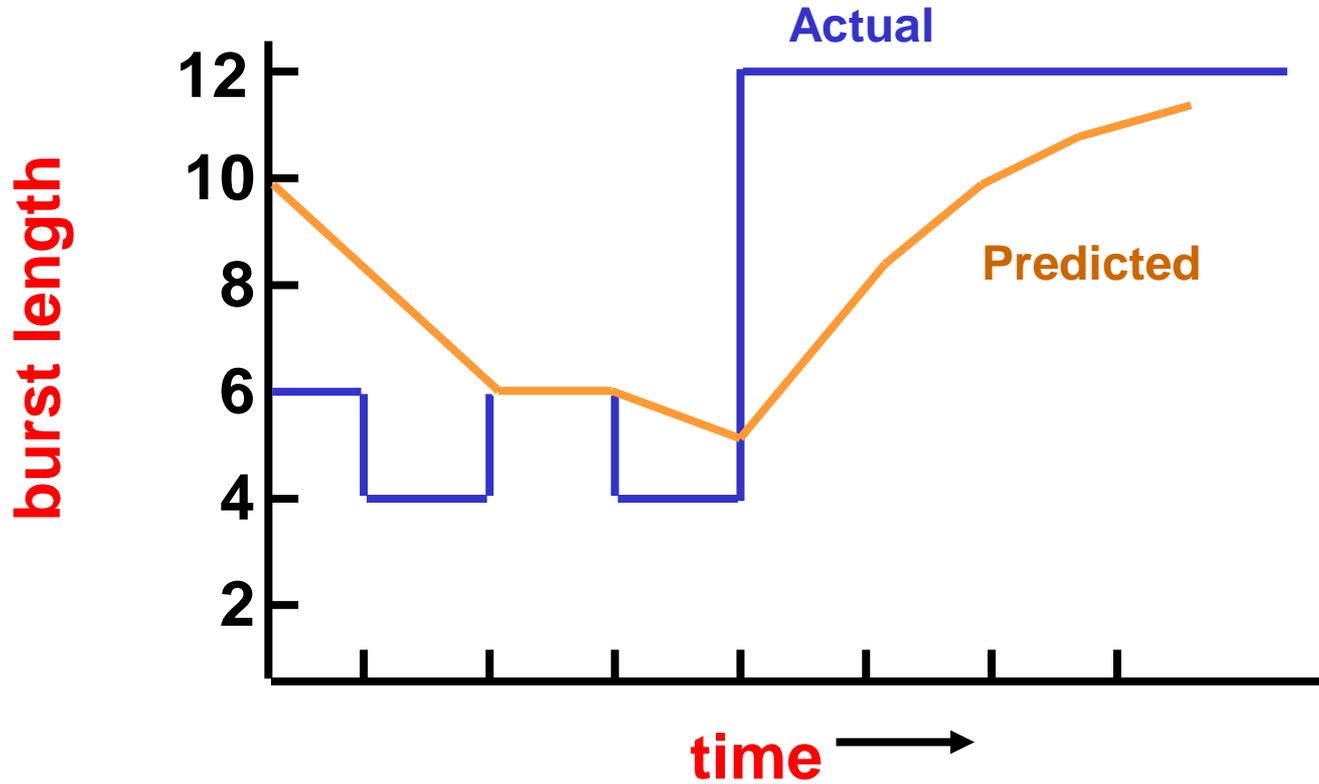
- A generic formula to estimate the length of the next CPU burst from the lengths of previous ones is:
 - $E_i = \alpha * A_{i-1} + (1 - \alpha) * E_{i-1}$
 - E_i : estimated burst length of a process at time i
 - A_{i-1} : actual burst length of a process at time $(i-1)$
 - E_{i-1} : previously estimated burst of a process at time $(i-1)$
 - α : factor to balance the importance of recent and not so recent bursts
- If $\alpha = 0$ then $E_i = E_{i-1}$
- If $\alpha = 1$ then $E_i = A_{i-1}$
- At time t_i we estimate the length of the next CPU burst based on information we have about previous bursts according to:
 - $E_i = \alpha * A_{i-1} + (1 - \alpha) * E_{i-1}$
- We have to select an initial value for α (here 0.75)
- For the first burst A_0 , we randomly set it, after that we use the measured time for the previous burst A_{i-1} .
- The measured burst time may be significantly different from the estimate.

Burst Length Prediction: Example

$$E_i = \alpha * A_{i-1} + (1 - \alpha) * E_{i-1}$$

Time	Estimate	Actual Burst (say 10)
T_0	$0.75 * 10 + 0.25 * 0 = 7.5$	6
T_1	$0.75 * 6 + 0.25 * 7.5 = 6.375$	3
T_2	$0.75 * 3 + 0.25 * 6.375 = 3.85$	7
T_3	$0.75 * 7 + 0.25 * 3.85 = 6.2$	12
T_4	$0.75 * 12 + 0.25 * 6.2 = 9.55$	

Example of Predicting CPU Bursts



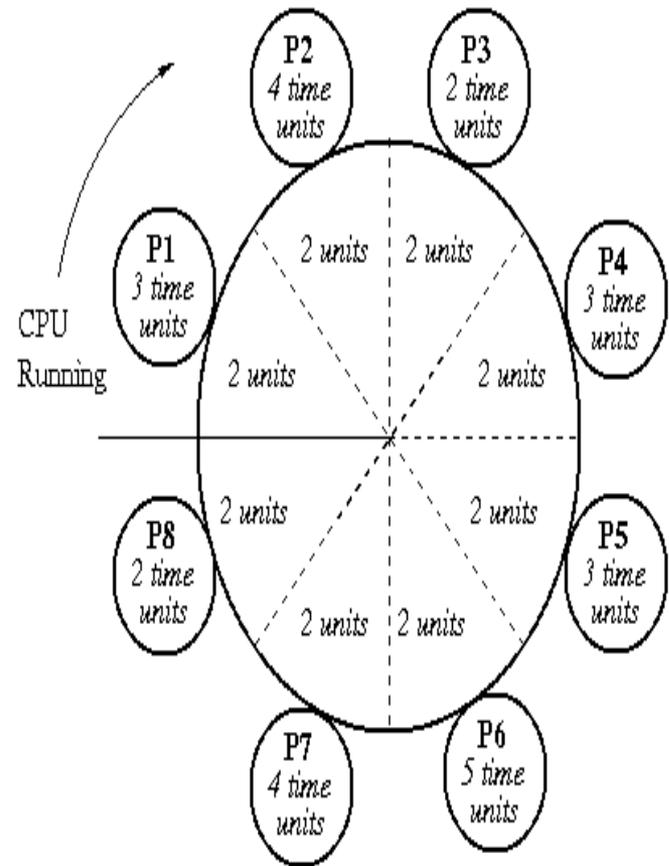
Actual CPU Burst A_{i-1} 6 4 6 4 12 12 .. 12

Predicted E_i 10 → 8 → 6 → 6 → 5 → 8.5 → 10 → 11

Vertical arrows point from the actual burst values (6, 4, 6, 4, 12, 12, 12) down to the predicted values (8, 6, 6, 5, 8.5, 10, 11).

Round Robin (RR) Policy

- It is a preemptive scheduling algorithm.
- Each process takes an equal or un-equal share of CPU time (**time quantum**) in turn. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- 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 time large** \Rightarrow FCFS (**most of the processes will finish in that quantum time**)
 - **q time small** \Rightarrow **more context switches**, overhead is too high and throughput is low.
 - **q time** must be large with respect to **context switch**, otherwise overhead is too high.



- The average waiting time of RR is often long, but
- The average response time is often less, good for multi/user /processing.

Example of Round Robin Policy

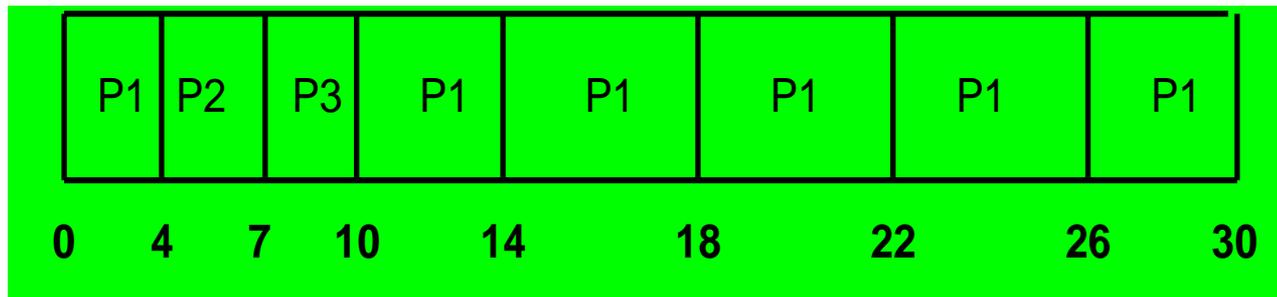
Process Burst-time

P_1 24

P_2 3

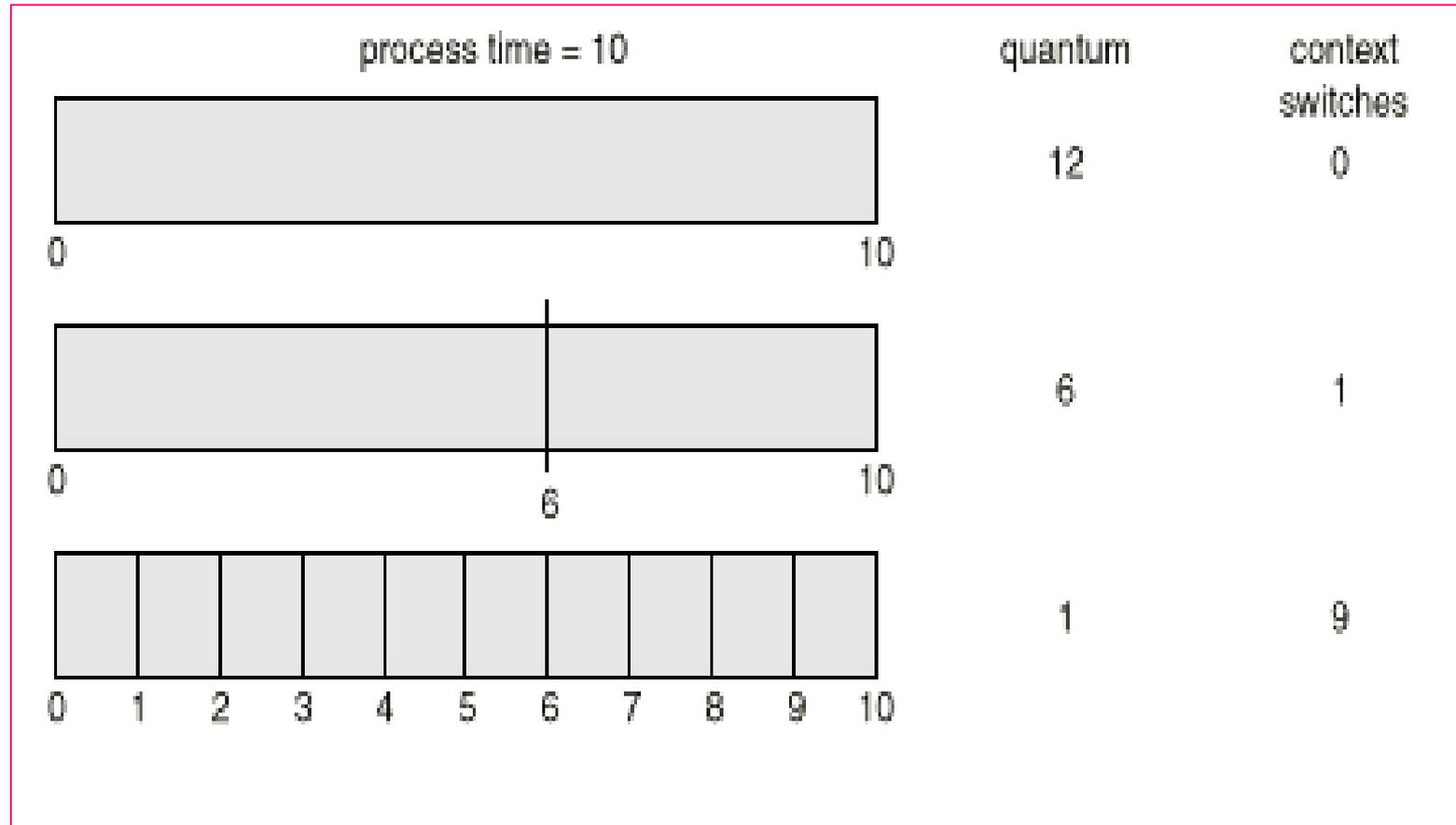
P_3 3

Set quantum at 4 milliseconds



- In the **RR policy**, the scheduling decision is made when:
 - A process is done
 - Expiration of the quantum time
- RR causes higher average turnaround time than SJF,
- **But improves the average response time and fairness** (starvation free)

Smaller Time Quantum Increases Context Switches

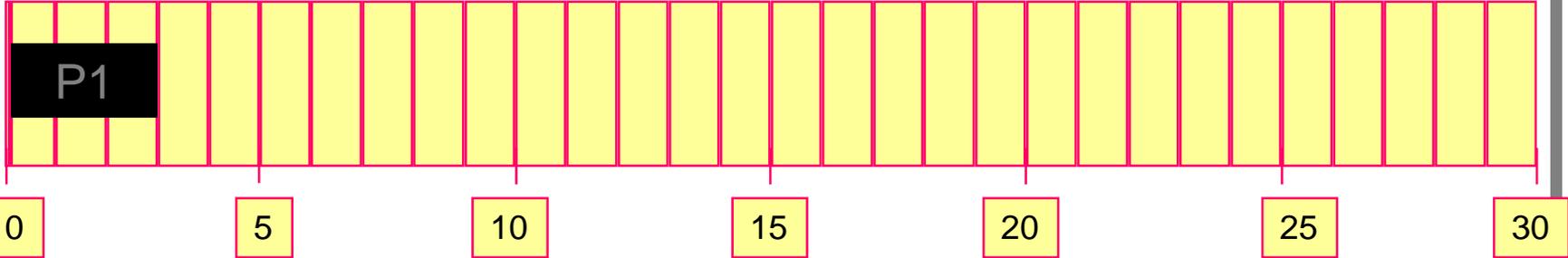


RR Policy Example: Preemptive

Process #	Arrival Time	Burst Length	Priority
P1	0	6	1
P2	3	15	1
P3	9	3	1
P4	14	4	1
P5	17	2	1

Time quantum:
3 units

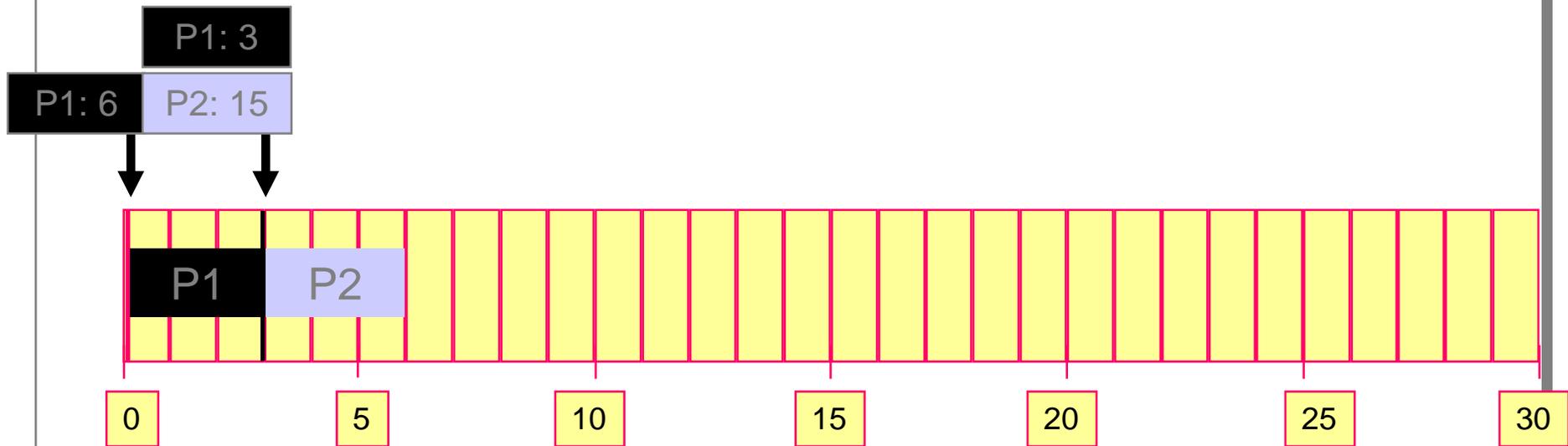
P1: 6



RR Policy Example: Preemptive

Process #	Arrival Time	Burst Length	Priority
P1	0	6	1
P2	3	15	1
P3	9	3	1
P4	14	4	1
P5	17	2	1

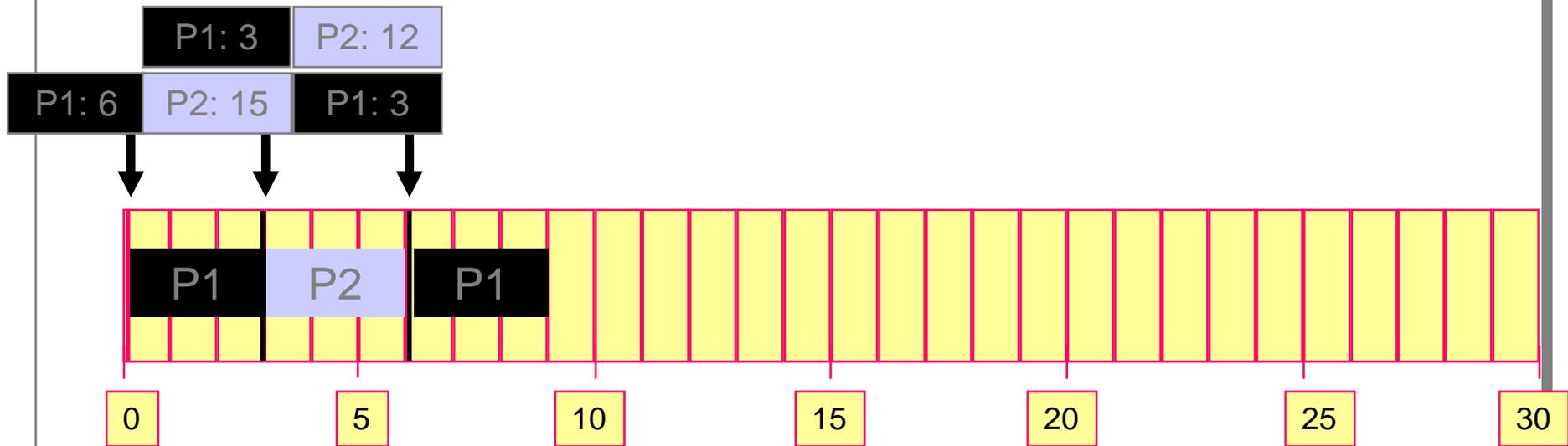
Time quantum:
3 units



RR Policy Example: Preemptive

Process #	Arrival Time	Burst Length	Priority
P1	0	6	1
P2	3	15	1
P3	9	3	1
P4	14	4	1
P5	17	2	1

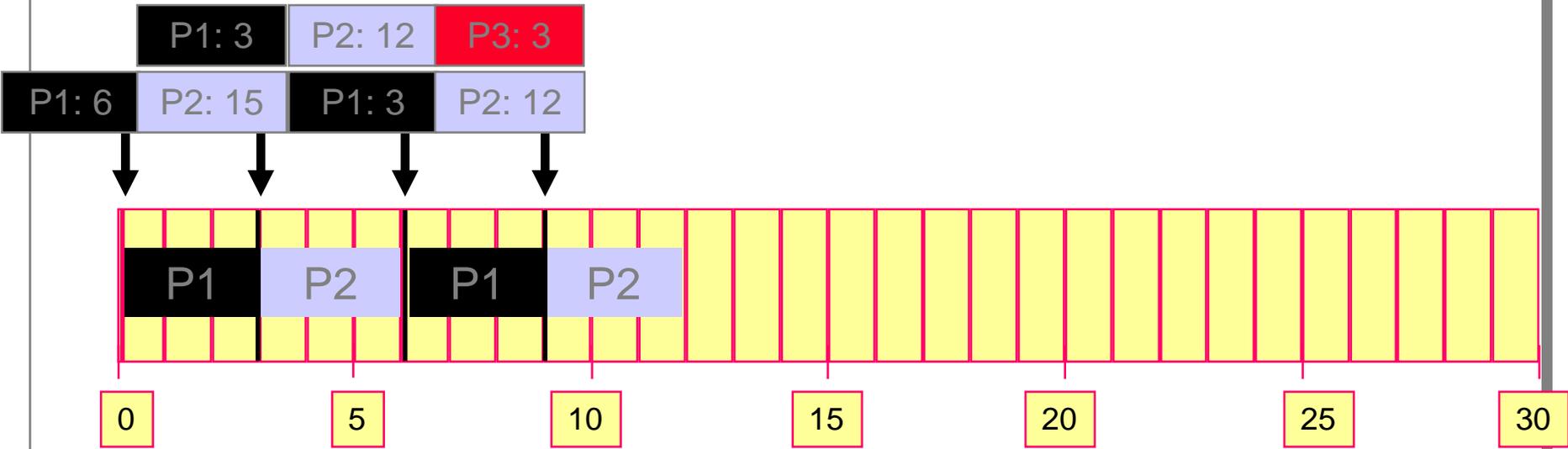
Time quantum:
3 units



RR Policy Example: **Preemptive**

<i>Process #</i>	<i>Arrival Time</i>	<i>Burst Length</i>	<i>Priority</i>
P1	0	6	1
P2	3	15	1
P3	9	3	1
P4	14	4	1
P5	17	2	1

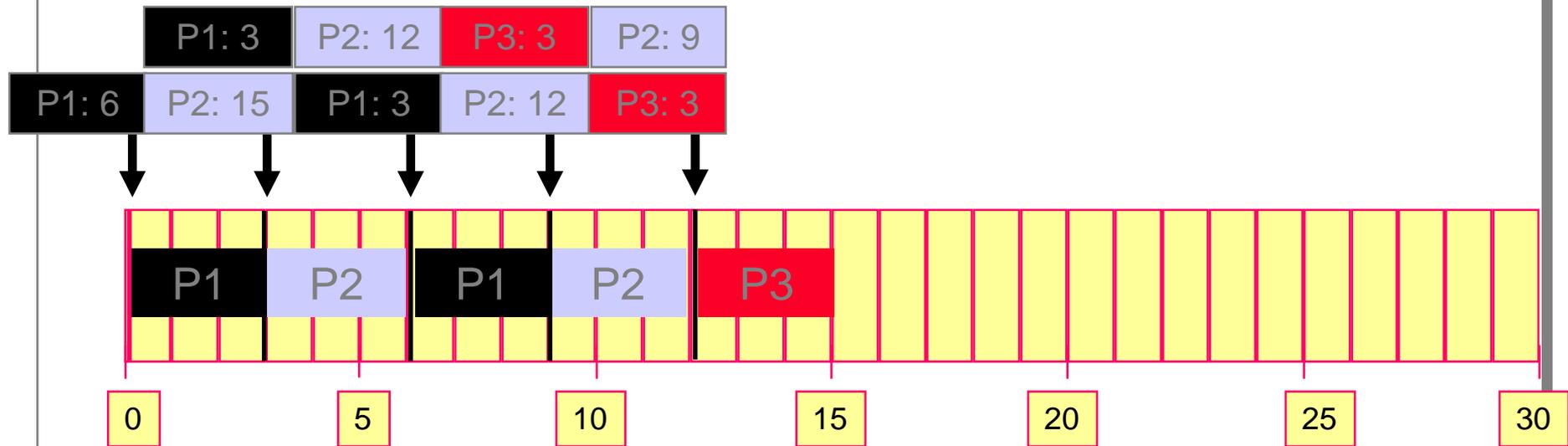
Time quantum:
3 units



RR Policy Example: Preemptive

Process #	Arrival Time	Burst Length	Priority
P1	0	6	1
P2	3	15	1
P3	9	3	1
P4	14	4	1
P5	17	2	1

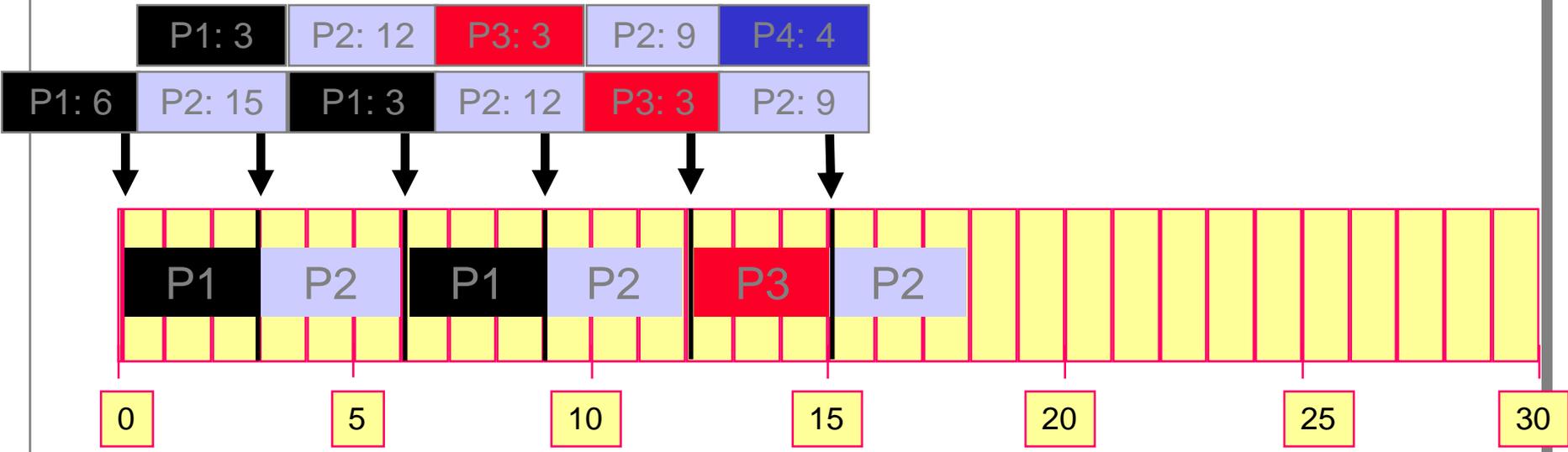
Time quantum:
3 units



RR Policy Example: Preemptive

<i>Process #</i>	<i>Arrival Time</i>	<i>Burst Length</i>	<i>Priority</i>
P1	0	6	1
P2	3	15	1
P3	9	3	1
P4	14	4	1
P5	17	2	1

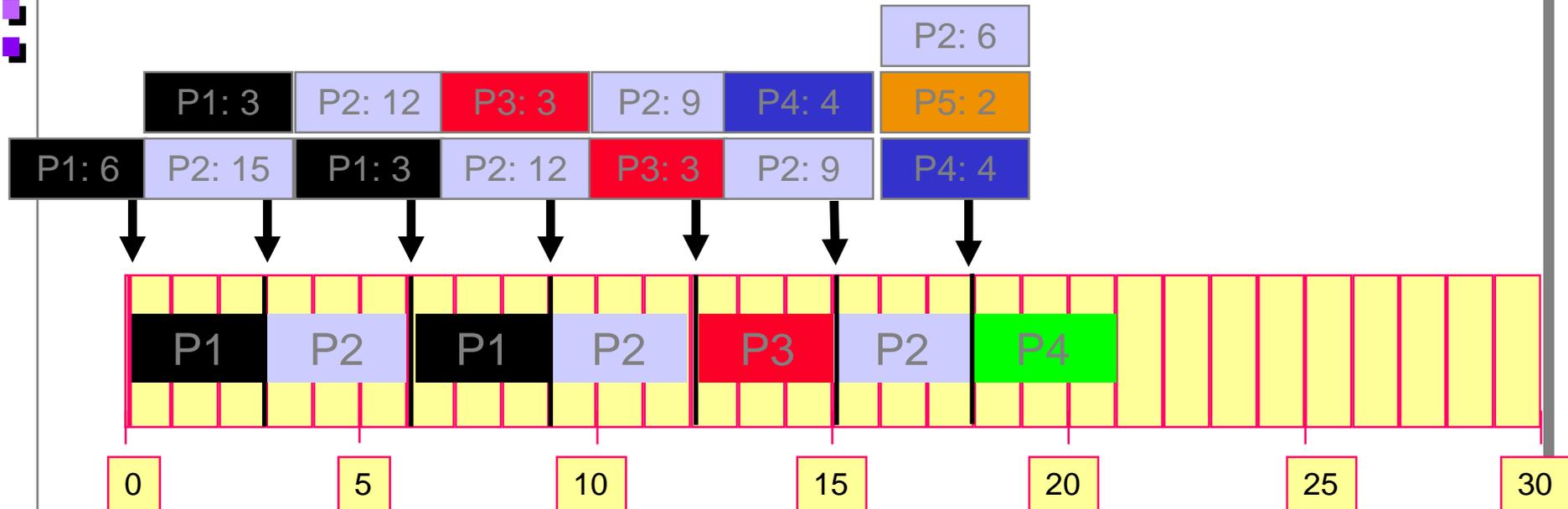
Time quantum:
3 units



RR Policy Example: Preemptive

Process #	Arrival Time	Burst Length	Priority
P1	0	6	1
P2	3	15	1
P3	9	3	1
P4	14	4	1
P5	17	2	1

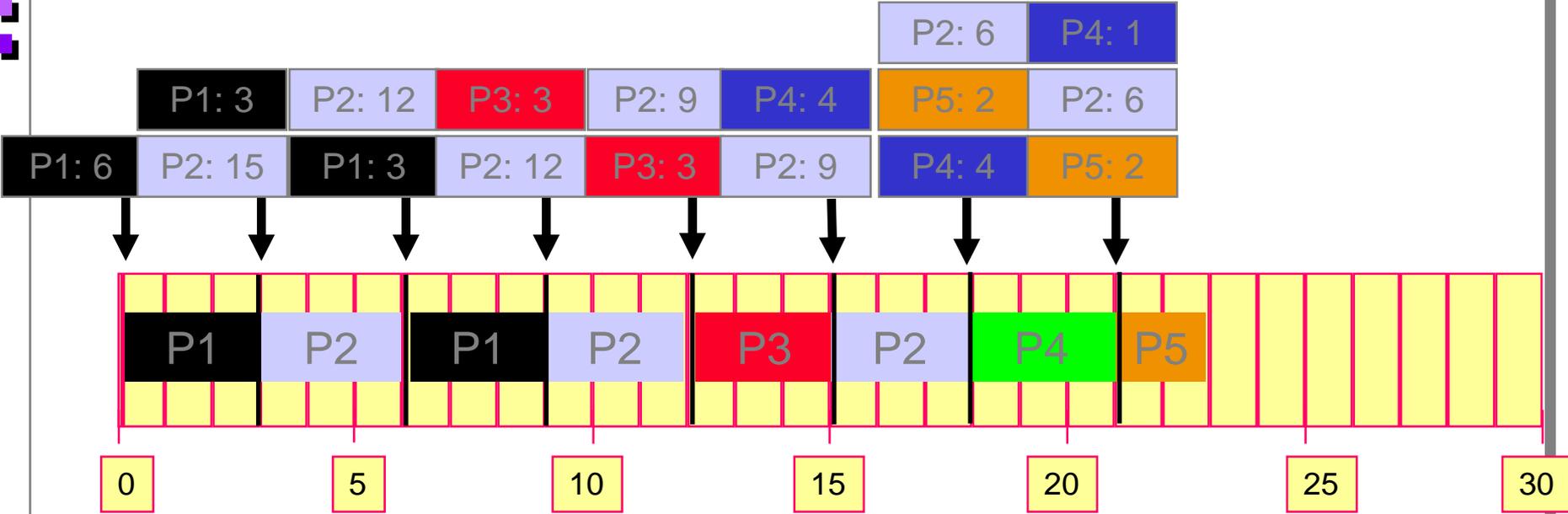
Time quantum:
3 units



RR Policy Example

<i>Process #</i>	<i>Arrival Time</i>	<i>Burst Length</i>	<i>Priority</i>
P1	0	6	1
P2	3	15	1
P3	9	3	1
P4	14	4	1
P5	17	2	1

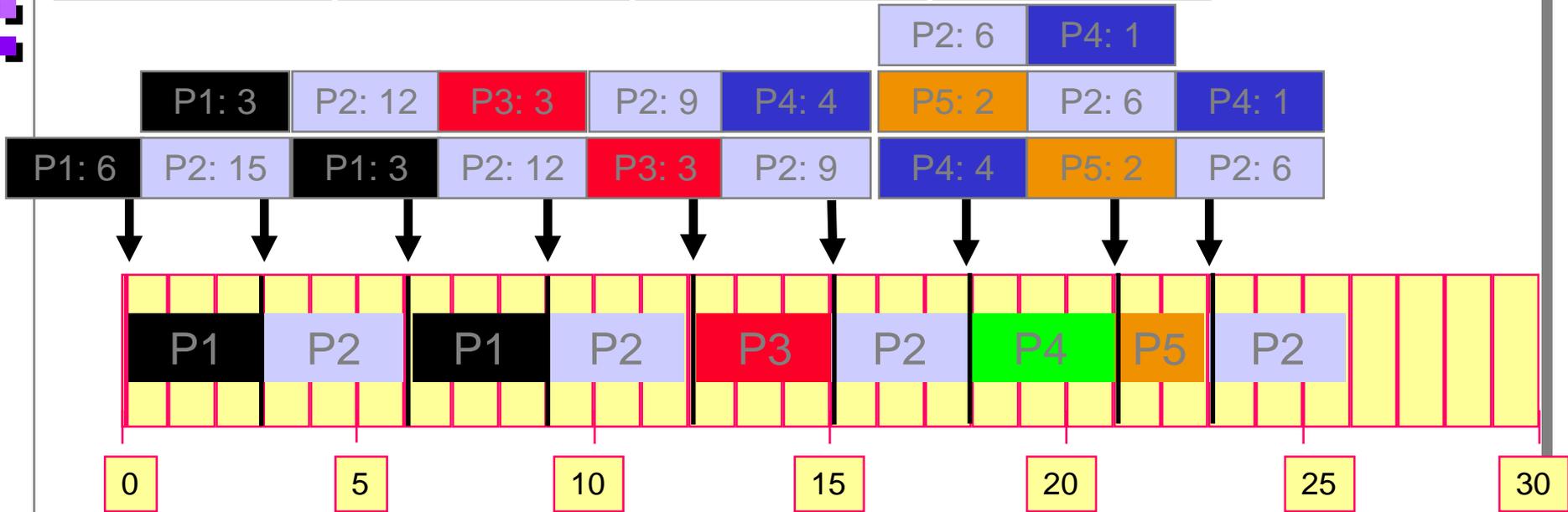
Time quantum:
3 units



RR Policy Example

<i>Process #</i>	<i>Arrival Time</i>	<i>Burst Length</i>	<i>Priority</i>
P1	0	6	1
P2	3	15	1
P3	9	3	1
P4	14	4	1
P5	17	2	1

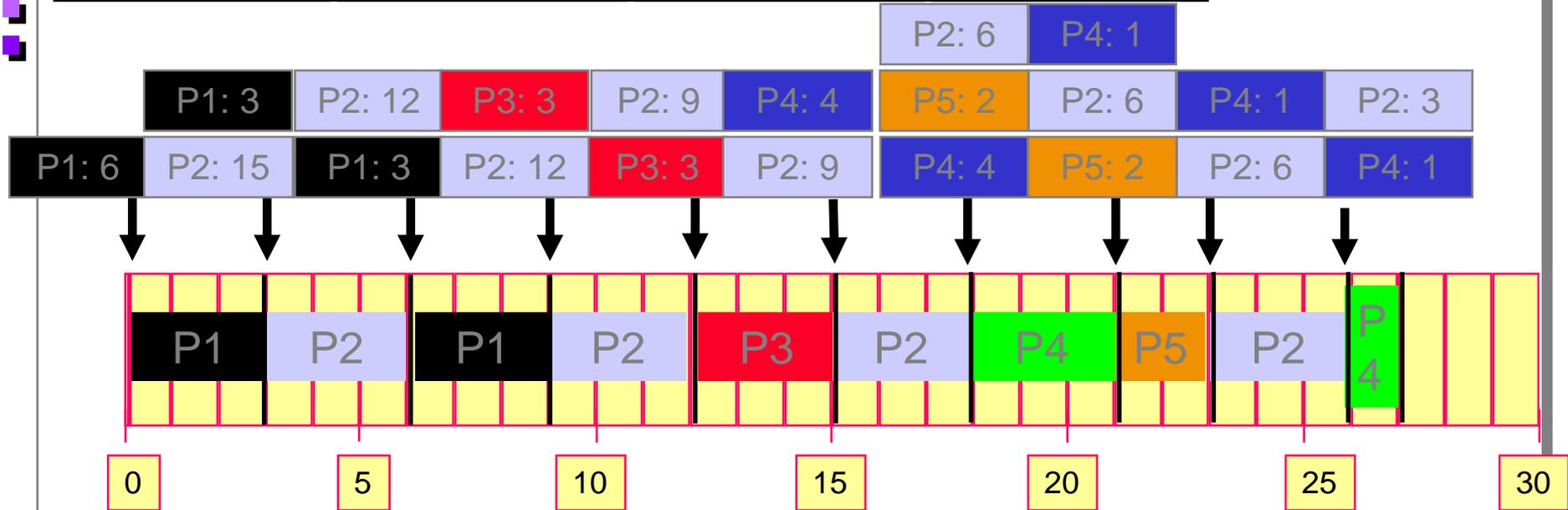
Time quantum:
3 units



RR Policy Example

Process #	Arrival Time	Burst Length	Priority
P1	0	6	1
P2	3	15	1
P3	9	3	1
P4	14	4	1
P5	17	2	1

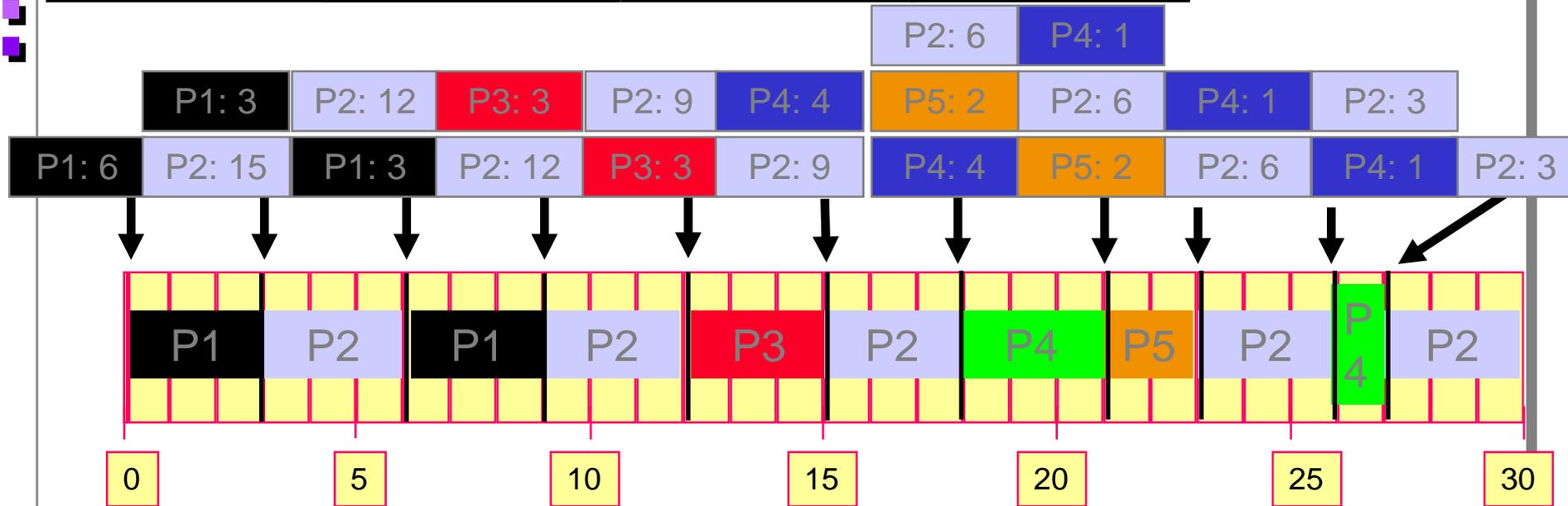
Time quantum:
3 units



RR Policy Example

Process #	Arrival Time	Burst Length	Priority
P1	0	6	1
P2	3	15	1
P3	9	3	1
P4	14	4	1
P5	17	2	1

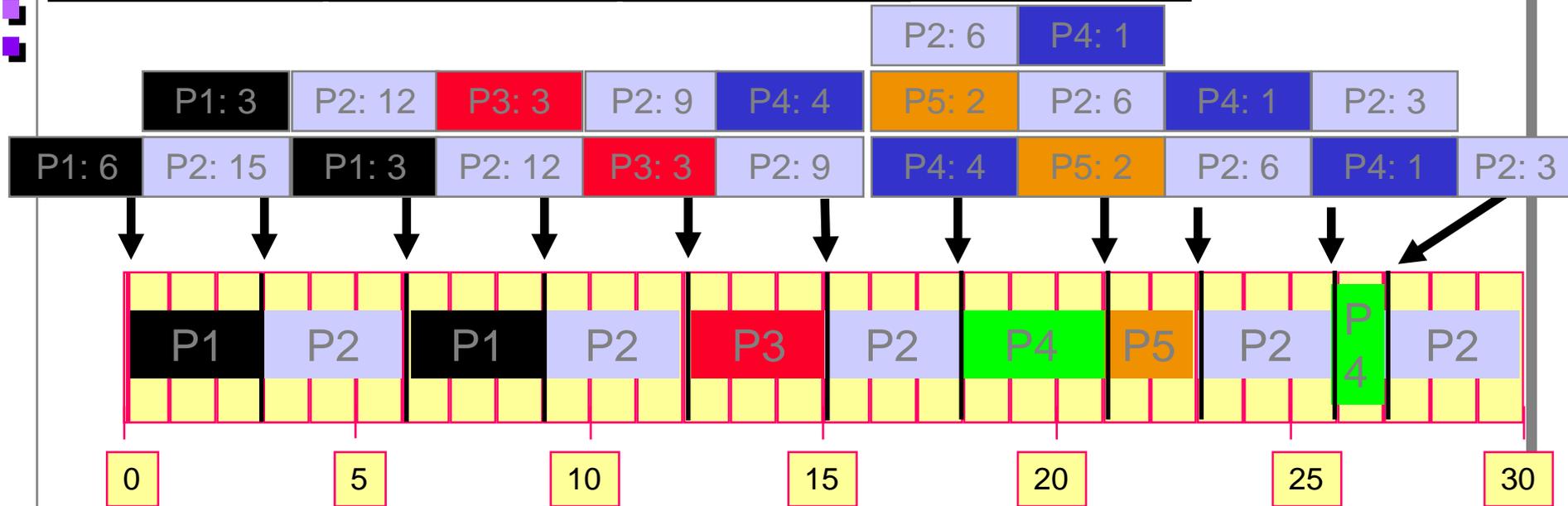
Time quantum:
3 units



RR Policy Example

Process #	Arrival Time	Burst Length	Priority
P1	0	6	1
P2	3	15	1
P3	9	3	1
P4	14	4	1
P5	17	2	1

Time quantum:
3 units



Process #	Waiting Time	Response Time	Turnaround Time	# of Context Switches
P1	3	0	9	2
P2	$(9-6) + (15-12) + (23-18) + (27-26) = 12$	0	$(30 - 3) = 27$	5
P3	$(12-9) = 3$	$(12-9) = 3$	$(15-9) = 6$	1
P4	$(18-14) + (26-21) = 9$	$(18-14) = 4$	$(27-14) = 13$	2
P5	$(21-17) = 4$	$(21-17) = 4$	$(23-17) = 6$	1
Average	$28/5 = 5.6$	$11/5 = 2.2$	$52/5 = 10.4$	$11/5 = 2.2$

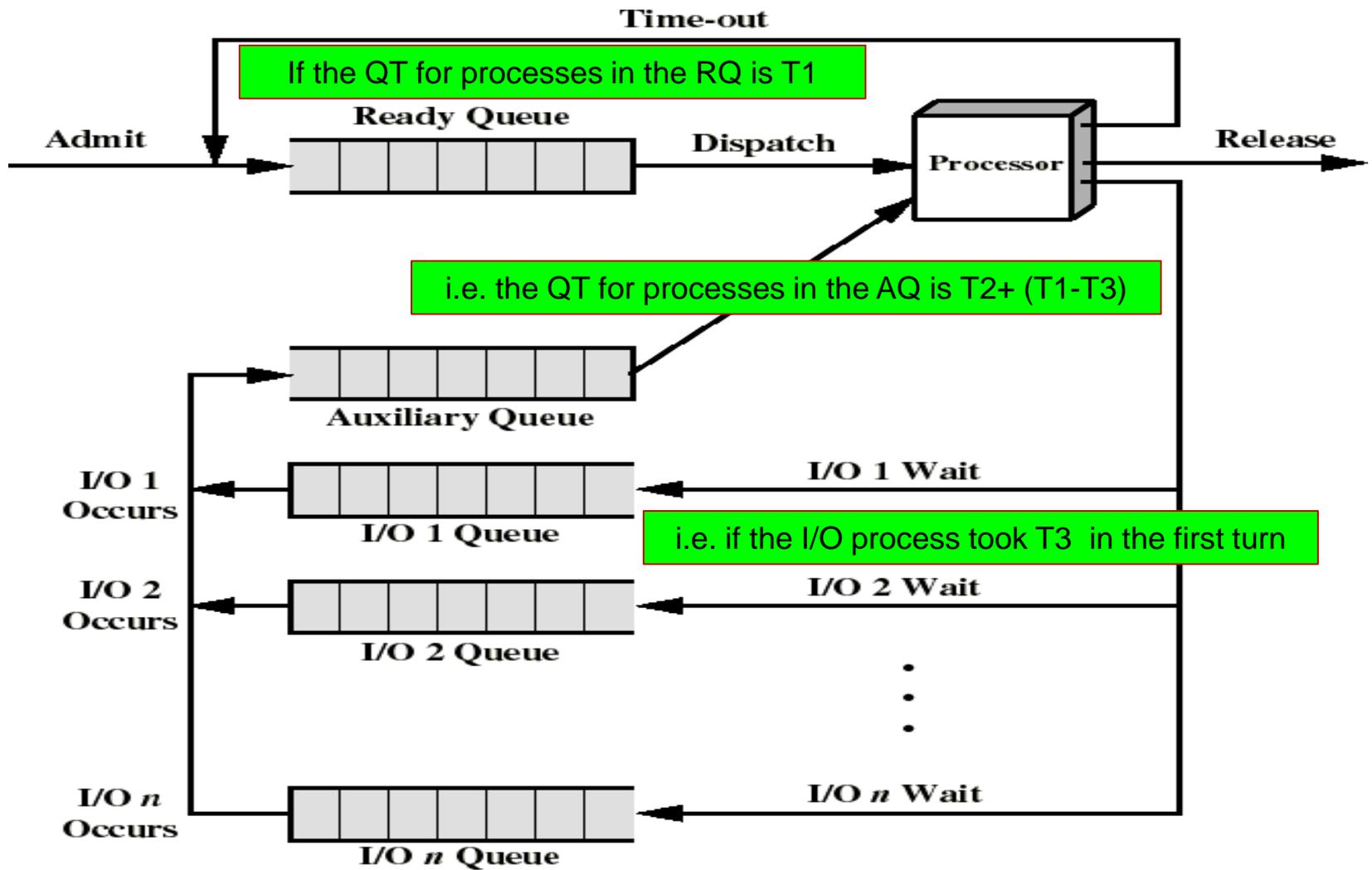
Round Robin Policy **Characteristics**

- It is a preemptive and can be non-preemptive. **How?**
- Used in time sharing environments, **WHY?**
- Sensitive to the length of the time quantum [**Research area!**] due to the overhead of context switch.
- **How to predict the best unit of CPU time to be given to the processes?**
- It favors CPU-bound processes than I/O-bound processes.
- I/O-bound processes are not favored [**Research area!**]
 - Every time an I/O request occurs the process has to go back to the end of the ready queue !!!
 - **Variation:**
 - Using a separate queue with higher priorities for processes “returning” from I/O activities.

Round Robin Policy: **Disadvantages**

- **It favors CPU-bound processes:**
 - An I/O bound process uses the CPU for a time that is less than the time quantum **then blocked waiting for I/O** and latter added to the end of the ready queue.
 - A CPU-bound process runs for all its time quantum and is put back into the ready queue (thus getting in front of blocked processes).
- **A solution: use virtual Ready Queue:**
 - When an I/O-bound process has completed, the blocked process is **moved to an auxiliary queue which gets preference over the main ready queue.**
 - A process dispatched from the auxiliary queue should be favored. i.e.
 - Given more time quantum or more frequency. i.e.
 - Take two processes from the auxiliary queue and one from the ready queue.

Queuing for Virtual Round Robin



RR Policy and Fairness

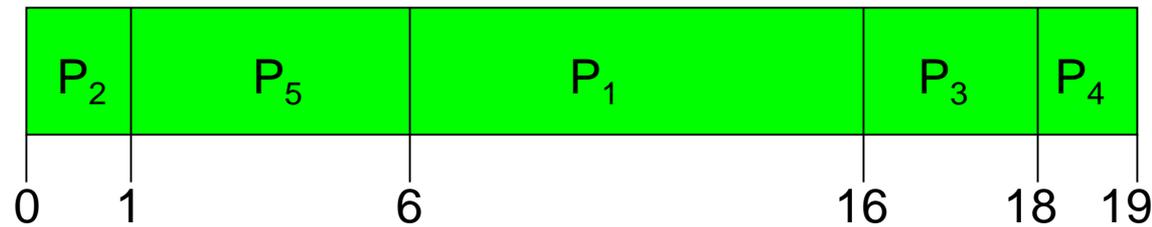
- Tries to give a fair share of CPU to each process (i.e. $1/N$ of the CPU time to each process if there are N processes of the same type).
- The algorithm should compute the ratio of actual CPU time used to the CPU time that the process should have been assigned (for a fair CPU assignment)
 - Runs the process with the lowest ratio until the ratio moves above its closest competitor.
- Takes into account the owner of the process and the number of processes per user when scheduling in multiuser system.
- Is CPU time shared among the users or processes fairly?
 - **Example:** User 1 has 4 processes, A, B, C, D, and user 2 has 1 process E. Then with round robin scheduling the execution sequence would be: **A B E C D E**

Priority Scheduling Policy

- A priority number is associated with each process (based on a certain criteria).
- The CPU is allocated to the process with the highest priority (smallest integer \equiv highest priority)
 - It can be Preemptive or Non-preemptive
 - Equal priority processes are scheduled in FCFS order.
 - **SJF** is a priority scheduling where the priority is assigned based on the predicted next CPU burst time.
- **Problem \equiv Starvation** (low priority processes may never execute)
 - **Aging**: Increase the process priority the longer it stays ready but is not run.

Example of Priority Scheduling: **Non-Preemptive Version**

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2



$$AWT = \frac{\sum (T_{\text{Start to use CPU}} - T_{\text{Arrive}})}{n} = \frac{(6+0+16+18+1)}{5} = 8.2$$

Priorities Assignment

- **Rate Monotonic Scheduling (RMS)**
 - A static/fixed priority assignment scheme
 - Assign priorities inversely proportional to the **execution times**
 - Shortest execution time first
 - Longest execution time first
 - **According to recourses utilization**
 - A process with less recourses utilization first
 - A process with more recourses utilization first
- Lower-priority may suffer from starvation.
- **Solution:** allow a process to change its priority based on its age or execution history.
- **Earliest Deadline First (EDF) Scheduling (for real time processes)**
 - Dynamic priority assignment scheme.
 - Priorities are assigned according to absolute deadlines:
 - The earlier the absolute deadline, the higher the priority.
 - EDF can achieve 100% CPU utilization while still guaranteeing all the deadlines.
- Dynamic Priority algorithms provide better processor utilization than Fixed Priority algorithms.

Hybrid Priorities Assignment

- To improve predictability for critical processes, use a combination of fixed and dynamic priority algorithms.
- Processes divided based on criticality into “critical and non-critical” processes.
- Critical processes scheduled using fixed priority assignment.
- Non-critical processes scheduled based on dynamic priority assignment.

Preemptive Priority Algorithm

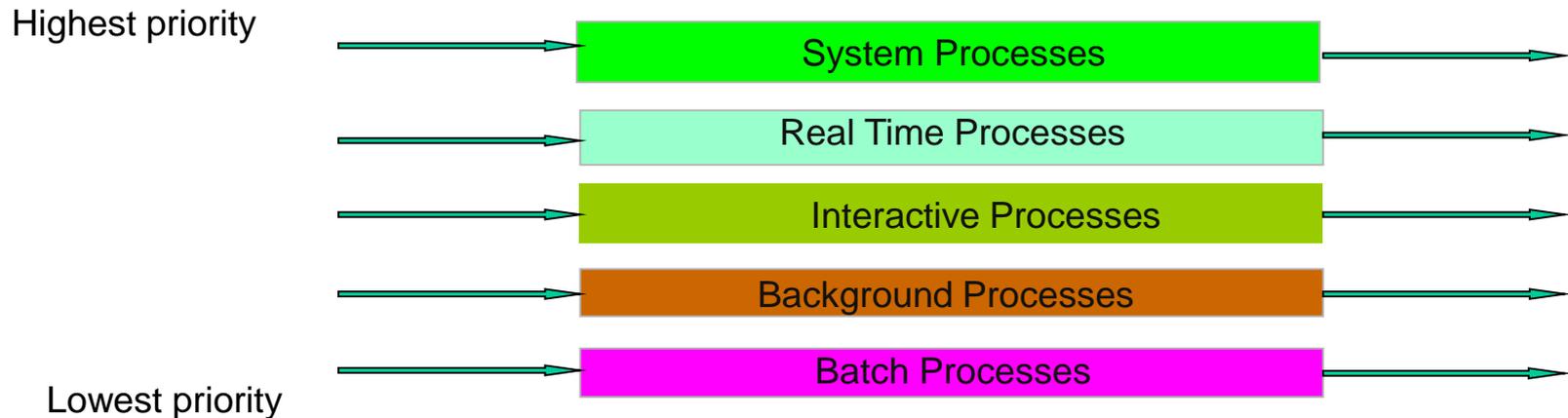
- When a process arrives at the ready queue with a higher priority than the running process, the new one preempts the running process.
 - Low priority processes can starve
 - i.e. UNIX uses aging to prevent starvation
 - **aging**: if a process has not received service for a long time, its priority is increased again.

Multi-level Queue Scheduling

- Till now we were using one ready queue for all ready processes and one scheduling algorithm to schedule all of them.
- A multi-level queue-scheduling (MLQ) algorithm **partitions the ready queue into several separate queues.**
- Created for situations in which processes are easily classified into groups:
 - **Foreground (interactive) processes, Background (batch) processes.**
- These two types of processes have **different response-time** requirements, and thus, **different scheduling** policies are needed.
- The processes are permanently assigned to one queue, based on some properties of the process. (e.g. memory size, priority, or type).
- Each queue has its own scheduling algorithm. For e.g. **the foreground queue might be scheduled by an RR algorithm** (improve throughput under light load), while the background queue is scheduled by a FCFS algorithm.
- Scheduling must be done between the queues.
 - **Fixed priority scheduling:** Serve all from higher-priority then from lower-priority; **possibility of starvation.**
 - **Time slice:** Each queue gets a certain amount of CPU time which it can schedule amongst its processes; e.g., 80% to foreground in RR 20% to background in FCFS

Multilevel Queue Scheduling

Example: A MLQ with 5 queues:



Possibility I

- If each queue has absolute priority over lower-priority queues then no process in the lower priority queues could run unless the processes of the highest-priority queue were all done.
- For example, in the above figure no process in the batch queue could run unless the queues for system processes, interactive processes, and interactive editing processes will all empty.

Possibility II

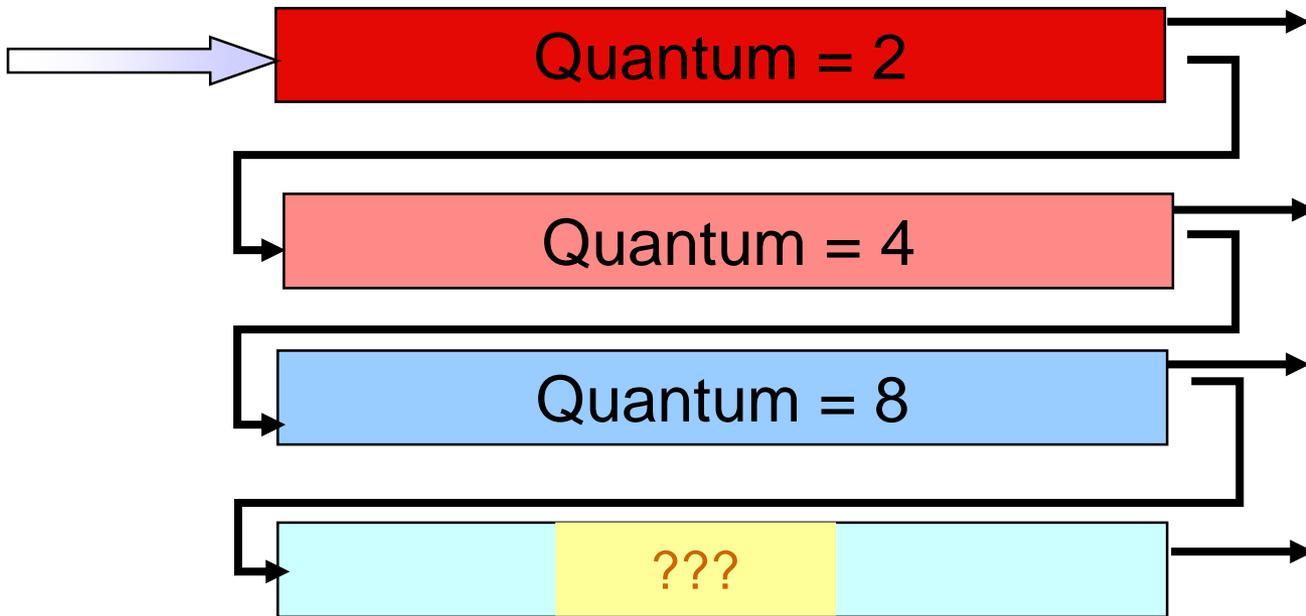
- If there is a time slice between the queues then each queue gets a certain amount of CPU times, which can then schedule it among the processes in its queue. For instance;
 - 30% of the CPU time to System Processes using Priority Scheduling Policy
 - 25 % of CPU time for RT processes using RR.
 - 20 % of CPU time for Interactive using SRT
 - 15% of the CPU time to background queue using SRTF.
 - 10% of the CPU time to Batch processes using FCFS

Multilevel Feedback Queues

- The problems with **Multilevel Queue Scheduling** are that:
- One queue may be full while the other may be empty: leads to wasting CPU time assigned to it.
- This could create some problems with CPU hogging and starvation.
- If a very long process is put on Q1, it could hog all of the time given to Queue 1, preventing the other processes from running.
- Once a process is put on a queue, it must remain there till its completion.
- **Proposed Solution**
 - Feedback queues attempt to solve this by allowing a process to move between various queues by:
 - Separating processes with different CPU-burst characteristics.
 - Leaving I/O-bound and interactive processes in the higher-priority queues.
 - Migrating a process waiting too long in a lower-priority queue to a higher-priority queue (**aging can be implemented this way**).

Multilevel Feedback Queues

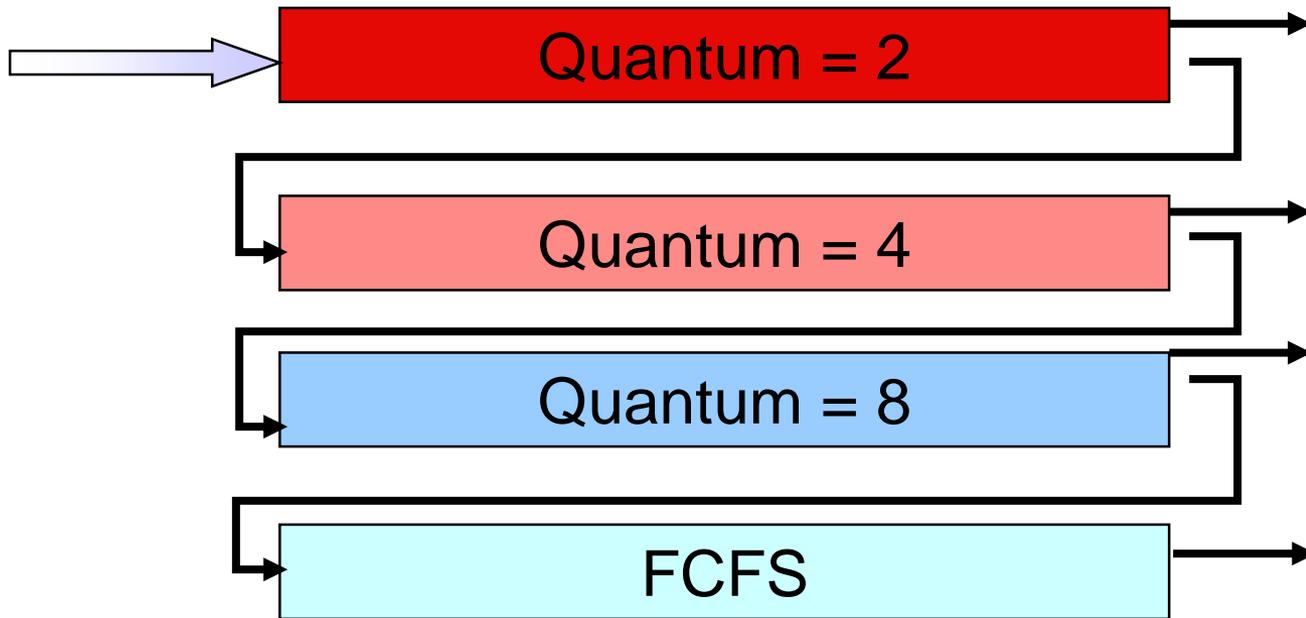
Highest priority



Lowest priority

Multilevel Feedback Queues

Highest priority



Lowest priority

Multilevel Feedback Queues

- Multilevel-feedback-queue scheduler implementation affected by the following parameters:
 - Number of queues
 - Scheduling algorithm for each queue
 - Method used to determine when to upgrade a process.
 - Method used to determine which queue a process will enter when that process needs service (i.e. all ready processes enter at the end of queue 0) .
 - Method used to determine when to downgrade a process to a lower-priority queue (i.e. move down one level at the end of each quantum)

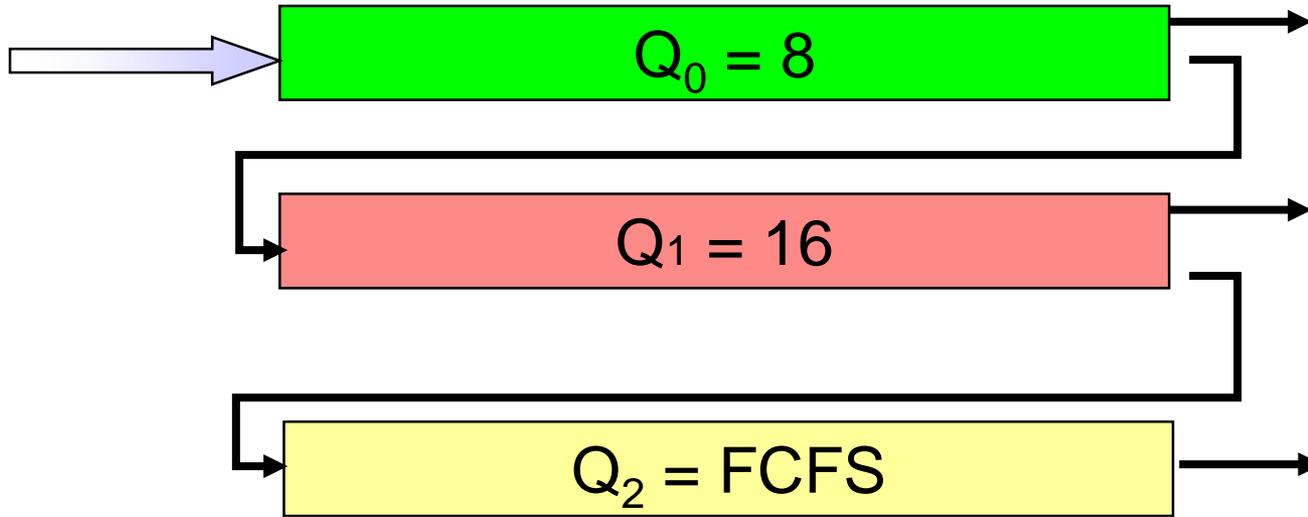
Ch: 5 Process Scheduling

- Schedulers overview (i.e. long, short, medium term schedulers)
- Scheduler, Dispatcher and Swapper modules in the OS
- Types of Scheduling Algorithms
 - Preemptive, and Non-preemptive
- Which of the Process states transition are Preemptive and which are Non-preemptive.
- Categories of Processes (i.e. batch, interactive, real-time) and the appropriate scheduling
- Evaluation Criteria of Scheduling Algorithms (Fairness, Throughput, Resources' utilization, WT, TT, RT, Context Switches, complexity)
- Ideal Process Scheduling Algorithm (i.e. should maximize what and minimize what)
- Factors affect the scheduler decision & implementation (i.e. Preemption frequency, type of the process, resolving conflicting requests, Page fault frequency, etc.)
- Scheduling Algorithms
 - First-Come, First-Served (FCFS)
 - Shortest Job First (SJF)
 - Shortest Remaining Time First (SRTF)
 - Process Burst Length Prediction
 - Round Robin, its Characteristics and Disadvantages
 - Priority-Based: Priorities assignment, its disadvantages and how to cover them
 - Multilevel Queue and Multilevel Feedback Queue
 - Example of MLFQ Scheduling algorithm
- Scheduling Algorithms Evaluation
 - Deterministic Model, Queuing Model, Simulations, Implementation
- Scheduling Polices in Different OS
- Scheduling in Multiple-Processor Systems
- Introduce Ch. 6 which is going to be studied by yourself

Example: Multilevel Feedback Queue

- Three queues:
 - Q_0 – Time quantum 8 milliseconds
 - Q_1 – Time quantum 16 milliseconds
 - Q_2 – Time quantum ∞ , what is the appropriate scheduling algorithm?
- Scheduling
 - A new job enters queue Q_0 , when it gains CPU, it receives 8 milliseconds.
 - If it does not finish in 8 milliseconds, it moves to queue Q_1 .
 - At Q_1 , the job receives 16 milliseconds, if it still does not complete, it is preempted and moved to queue Q_2 .

Multilevel Feedback Queues



Multilevel Feedback Queues

P1	P2	P3	P4	P1	P2	P4	P1	P4	
0	8	16	21	29	45	47	63	67	69

- In Q_0 (Quantum time = 8):
 - P1 uses 8 and needs 20 more
 - P2 uses 8 and needs 2 more
 - P3 uses 5 and terminates
 - P4 uses 8 and needs 18 more
- So P1, P2, P4 will go to Q_1 (Quantum time = 16):

P	BT
P1	28
P2	10
P3	5
P4	26



P	BT
P1	20
P2	2
P4	18

Multilevel Feedback Queues

P1	P2	P3	P4	P1	P2	P4	P1	P4	
0	8	16	21	29	45	47	63	67	69

- P1, P2, P4 in Q_1 (Quantum time = 16):
 - P1 uses 16 and needs 4 more;
 - P2 uses 2 and terminates;
 - P4 uses 16 and needs 2 more;
- So P1(4), P4(2) will go to Q_2 (Quantum time = ∞)
 - P1 uses 4 and terminates;
 - P4 uses 2 and terminates.
- **AWT, ART & ATT ?**
- You need to calculate them for all processes.

P	BT
P1	20
P2	2
P4	18



P	BT
P1	4
P4	2

Comparison of the Algorithms

- After studying the following scheduling algorithms:
 1. First-Come, First-Served (FCFS)
 2. Shortest Job First (SJF)
 3. Shortest Remaining Time First (SRTF)
 4. Round robin
 5. Priority Scheduling
 6. Multilevel queue
 7. Multilevel feedback queue
- Which one is best?
- The answer depends on:
 - The system workload (**which is extremely variable**),
 - The type of concurrently running processes (batch, real time, interactive, etc.),
 - Hardware support for the dispatcher,
 - Relative weighting of the performance metric criteria (**response time, CPU utilization, throughput, etc.**)
- Hence the answer depends on too many factors.

CPU Scheduling Algorithm Evaluation

- Define the criteria for evaluation and comparison:
 - Maximize CPU utilization with maximum response time $<$ threshold value.
 - Keep the CPU as busy as possible.
 - Maximize throughput, how?
 - Make the CPU executes and completes as many processes as it can.
- Environment in which the scheduling algorithm is used will change
 - An algorithm may be good today, but not good tomorrow?
- Evaluation methods
 - Deterministic modeling
 - Queuing models
 - Simulations
 - Implementation

Deterministic Modeling

- **Deterministic modeling:**
 - Deterministic modeling is an analytic evaluation,
 - Consider a predefined workload (**set of processes**) and evaluate the performance of each algorithm for that workload **to determine the best decision is what.**
 - Simple, fast, and gives exact numbers,
 - Too specific, and requires too much exact knowledge to be useful.

Example: Compare FCFS, SJF, RR, &MLFQ

- Consider the following scenario with 3 processes P1, P2, and P3.

Processes	CPU Time	Arrival Time
P1	44	0
P2	9	1
P3	13	2

- Assume the context switch takes one time unit.
- Assume the scheduler itself takes 2 time units and is ready to act at time 2.
- Let us say our goal is to maximize the throughput.
- To maximize the throughput,
 - What is the evaluation Criteria we should calculate?
 - Turn around time
- What is the best scheduling algorithm for executing them?

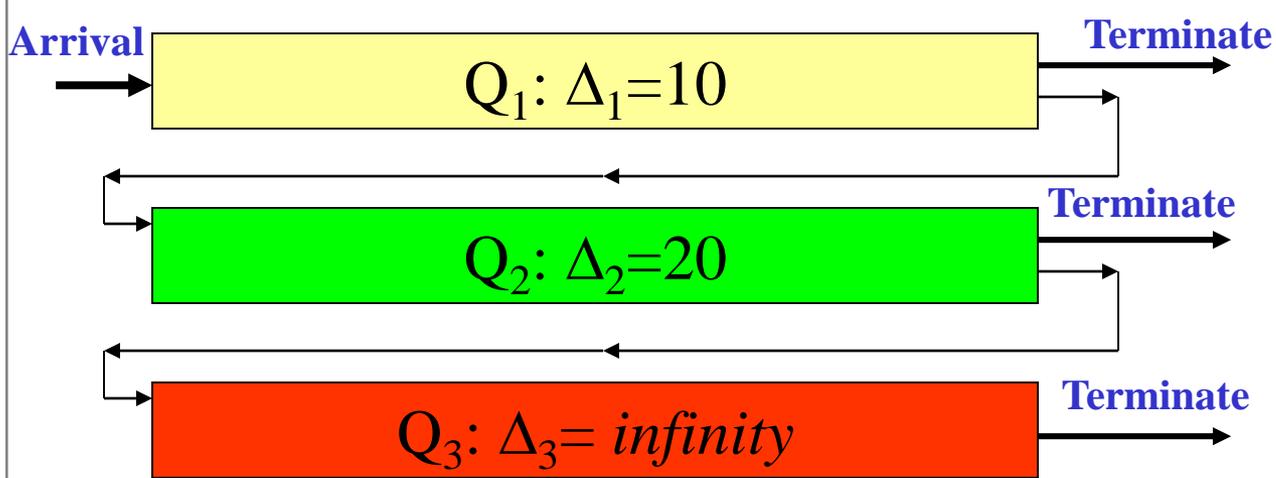
RR (Quantum = 10)



$$ATT = [(75-0) + (22-1) + (48-2)] / 3 = 47.33$$

Processes	CPU Time	Arrival Time
P1	44	0
P2	9	1
P3	13	2

MLFQ



$$ATT = [(73-0) + (22-1) + (54-2)] / 3 = \mathbf{48.67}$$

Processes	CPU Time	Arrival Time
P1	44	0
P2	9	1
P3	13	2

Compare FCFS, SJF, RR, &MLFQ

Scheduling Algorithm	ATT	Order
FCFS	56.33	4
SJF	34.33	1
RR	47.33	2
MLFQ	48.67	3

- That means, we can use case based learning mechanism and if we have similar set of processes and our objective is to maximize the throughput, the SJF is the best decision

Queuing Models

- Mathematical and statistical analysis
- Use the **distribution of processes** (CPU, I/O bound) **arrival** and the average waiting time to compute the average length of ready queue which will affect the (throughput, utilization, ...).

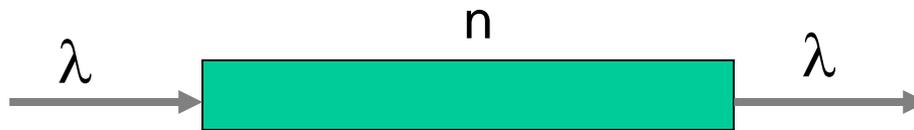
- **Example: Little's formula**

$$n = \lambda \times W$$

n: average queue length (**efficiency of the algorithm**)

λ : average arrival rate (**incoming processes**)

W: average waiting time (**depends on the scheduling algorithm**)



- if $\lambda=7$ processes/sec, and $W=2$ sec. $\rightarrow n=14$
- As far as the algorithm is **minimizing** the average queue length n , it will be good.
- Can only handle simple algorithms and distributions.

Simulations

- Simulate a computer system model:
 - Use data structure techniques to model **queues, CPU, devices, timers, etc...**
 - Simulator modifies the system state to **reflect the activities of the devices, CPU, the scheduler, etc.**
- Data to drive the simulation
 - Random-number generated according to probability distributions
 - Processes, CPU- and I/O-burst times, arrivals/departures
- Disadvantage – expensive
- **Many open source CPU scheduling simulators are available on the web: i.e.**
 - <http://cpuss.codeplex.com/>
 - http://www.jimweller.net/jim/java_proc_sched/applet.html
 - <http://www.clusterresources.com/products/maui-cluster-scheduler.php>
 - <http://www.redbooks.ibm.com/abstracts/sg246038.html>
 - <http://www.platform.com/workload-management/high-performance-computing>
 - http://en.wikipedia.org/wiki/Portable_Batch_System
 - <http://www.oracle.com/technetwork/oem/grid-engine-166852.html>
 - <http://oar.imag.fr>

Implementation

- In an open source OS like UNIX, Linux
- Propose or refine the code of any scheduling algorithm,
- Put the coded algorithm in the real system for evaluation under real operating conditions.
- Evaluate its performance empirically
- **Difficulty**
 - Costly in coding the algorithm,
 - Needs to have an open source OS.

Windows 2000, NT, XP, 7 & 8 Scheduling

- Windows uses a **priority-based pre-emptive scheduling** algorithm and 31 priority levels (31 is the highest).
- Priorities are assigned to both processes and threads.
- A running thread will run until it is preempted by a higher-priority one, terminates, time quantum ends, calls a blocking system call
- Ranges of priorities are defined for processes and threads:
 - **Processes** - idle, normal, and very high
 - **Threads** - idle, below normal, normal, above normal, highest
- Dynamic priority may be adjusted up or down by the OS to reflect changing conditions.
 - Lower (not below base priority) when its time quantum runs out
 - Priority raised up when it is released from a wait operation
 - The higher level depends on the reason for wait
 - Waiting for keyboard I/O gets a large priority increase
 - Waiting for disk I/O gets a moderate priority increase
 - Process in the foreground window get a higher priority

Windows Process Priorities

priority classes

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

levels within class

- Compute until end of slice: priority lowered
- After waiting for I/O: priority raised
 - If I/O was mouse, keyboard: priority raised even more
 - Favors interactive processes
- Foreground process (window) receives larger slice than background process (3x)

Windows 10 Scheduling

- Windows 10 uses a round-robin technique with a multi-level feedback queue for priority scheduling.
- The ready queue is partitioned into multiple queues of different priorities.
- The system assigns processes to queue based on their CPU burst characteristic.
- If a process consumes too much CPU time, it is placed into a lower priority queue.
- Process that waits too long in a lower priority queue may be moved to a higher priority queue.

Linux Scheduling

- Separate Time-sharing and real-time scheduling algorithms
- Allow only processes in **user** mode to be preempted
 - A process may not be preempted while it is running in kernel mode, even if a real-time process with a higher priority is available to run
 - Soft real-time system
- Two algorithms: time-sharing and real-time
- Time-sharing
 - Prioritized credit-based – process with most credits is scheduled next
 - 1 credit subtracted when timer interrupt occurs
 - When credit = 0, process suspended, another one chosen
 - When all ready processes have credit = 0, re-crediting occurs
 - $\text{credits} = \text{credits} / 2 + \text{priority}$ (internally: high priority = high int)
 - So, if process did I/O before credits = 0, it has more credits next time
 - I/O-bound, interactive processes accumulate more credits
 - Two real-time scheduling classes: FCFS (non-pre-emptive) and RR (pre-emptive)
 - PLUS a priority for each process
 - Always runs the process with the highest priority
 - Equal priority → runs the process that has been waiting longest

Unix System Scheduling

- Adopts multilevel feedback with RR
- There is priority involved with each queue.
- If a running process does not block or complete, it is preempted within 1 second.
- Priority is determined according to process type and execution history.

$$P_j(i) = \text{Base}_j + \text{CPU}_j(i-1) + \text{nice}_j$$

$$\text{CPU}_j(i) = U_j(i) / 2 + \text{CPU}_j(i-1) / 2$$

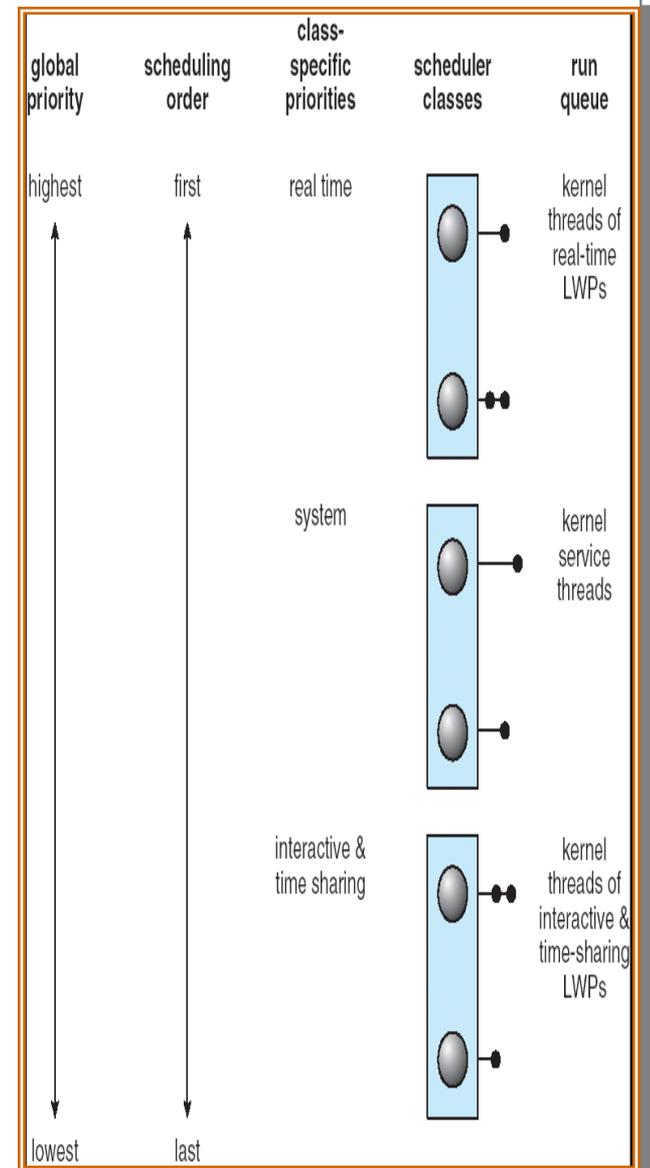
- $P_j(i)$ = Priority of P_j at the beginning of interval i (lower values equal higher priority).
- Base_j = Base priority of P_j
- $U_j(i)$ = CPU use of P_j in interval i
- $\text{CPU}_j(i)$ = Exponentially weighted average CPU use of P_j through interval i
- nice_j = User control adjustable factor.

Unix System Scheduling

- The priority of a process is computed once per second.
- The purpose of the base priority is to divide all processes evenly into fixed bands of priority levels.
- The CPU and nice components are restricted to prevent a process from migrating out of its assigned band.
- The bands are swapper, block I/O device control, file manipulation, character I/O device control and user processes.

Solaris 2 Scheduling

- Priority-based process scheduling
 - **Classes:** real time, system, time sharing, interactive
 - Each class has different priority and scheduling algorithm
- Each thread assigns a scheduling class and priority
- Real-time processes run before a process in any other class.
- System class is reserved for kernel use (paging, scheduler)
 - The scheduling policy for the system class does not time-slice
- Time-sharing/interactive: multilevel feedback queue
- The selected thread runs on the CPU until it blocks, uses its time slices, or is preempted by a higher-priority thread.
 - Multiple threads have the same priority → RR



Summary

- CPU scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it. The CPU is allocated to the selected process by the dispatcher.
- FCFS is the simplest scheduling algorithm but it can cause short processes to wait very long.
- SJF provides the shortest average waiting time. Implementing SJF is difficult, due to the difficulty in predicting the length of the next CPU burst.
- SJF is a special case of the general priority scheduling algorithm, which allocates the CPU to the highest-priority process. Both SJF and priority may suffer from starvation. Aging is a technique to prevent starvation.

Summary

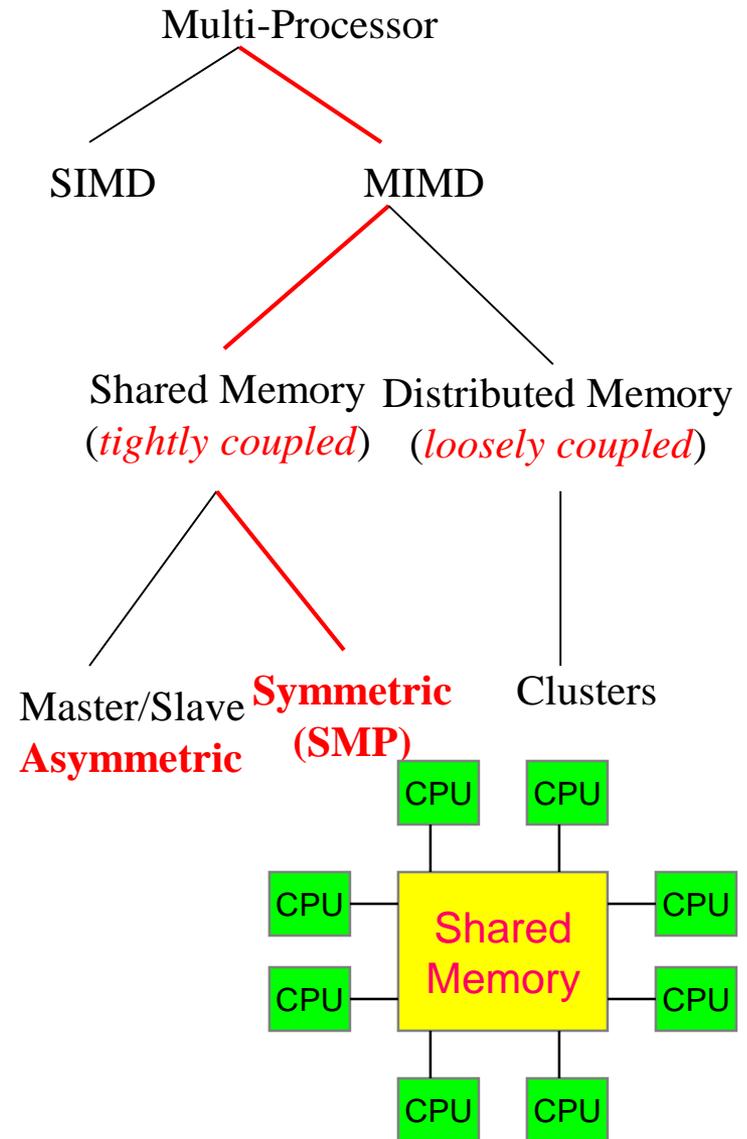
- RR scheduling is appropriate for time-sharing systems. RR allocates the CPU to the first process in the ready queue for q time units. The major problem is the selection of the time quantum. If the quantum is too large, RR degenerates to FCFS; if q is too small, overhead due to context switching becomes excessive.
- The FCFS algorithm is non-preemptive
- The RR algorithm is preemptive.
- The SJF and priority algorithms may be either preemptive or non-preemptive.
- Multi-level queue algorithm allows different algorithms to be used for various classes of processes. The most common is a foreground interactive queue which uses RR scheduling, and a background batch queue which uses FCFS scheduling.
- Multi-level feedback queues allow processes to move from one queue to another.

Multiple-Processor Systems: FYI only

- So far we looked at CPU scheduling algorithms for single processor systems. If multiple CPUs exist, the scheduling problem becomes more complex.

Classifications of Multiprocessor Systems:

- Loosely coupled multiprocessor, or clusters
 - Each processor has its own memory and I/O channels.
- Functionally specialized processors
 - Such as I/O processor
 - Controlled by a master processor
- Symmetric Multi-Processors (SMP):
 - The OS code and data structures are global (stored in the shared memory and equally available to each processor).
 - Controlled by operating system
 - One copy of OS in memory, any CPU can use it.
 - OS must ensure the consistency of shared data.



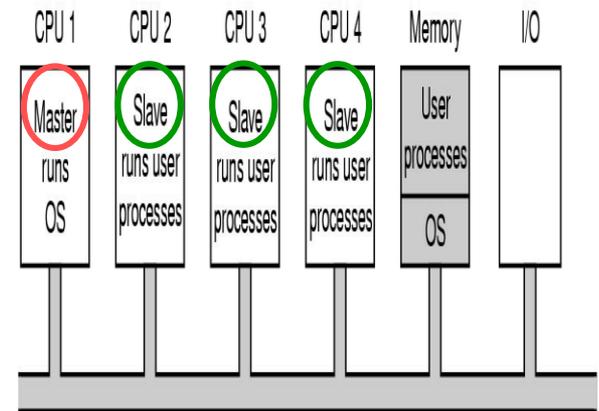
Multiple-Processor Scheduling: Design Issues

- Scheduling on a multiprocessor involves 3 interrelated design issues:
 - Assignment of processes to processors
 - One of the design issues that must be decided is (which process to assign to each processor?).
 - Actual dispatching of a process [Process Scheduling]
 - Use of multithreading on individual processors [Threads Scheduling]
- It is two dimensional problem
 - Which processor runs the next process? (processor scheduler, coordinator processor)
 - Which process runs next? (process scheduler, per processor)

Assignment of processes to processors

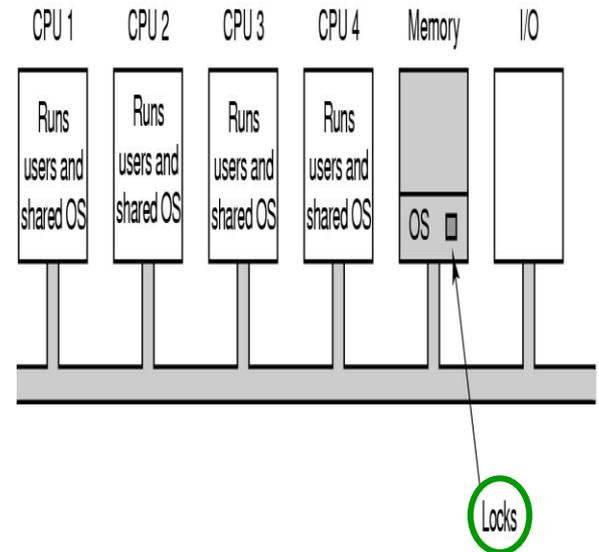
1. Master/slave architecture (Asymmetric MP Scheduling)

- Main kernel functions always run on a particular processor/master. **Master is responsible for scheduling**
- **Slave processors execute user processes.**
- Only one processor accesses the system data structures, alleviating the need for data sharing.
- No coherency problems
- Failure of master brings down the whole system.
- Master can become a performance bottleneck.



2. Peer to Peer architecture (Symmetric MP Scheduling)

- OS executes on any processor
- Each processor does self-scheduling
- Processes may be in a global ready queue, or each processor may have its own ready queue
- **Mutual exclusion problems** (Make sure two processors do not choose the same process; needs lots of synchronization)
- SMP is supported by all modern operating systems: Windows, Solaris, Linux, Mac OS X.



Assignment of processes to processors

- **Dynamic vs. static assignment of processes to processors:** Types of policies:
 - **Static:** decisions have been known to system
 - **Dynamic:** uses load information
 - **Adaptive:** policy varies according to load
- **Exploit cache affinity:** try to schedule a process/thread on the same processor that a process/thread executed last.
- **Context switch overhead**
 - Quantum sizes larger on multiprocessors than uni-processors
 - Preemptive versus non-preemptive
- **Measure of load:** Queue lengths at CPU, CPU utilization
 - **Transfer policy:** when to transfer/migrate a process?
 - Threshold-based policies are common and easy
 - **Selection policy:** which process to transfer/migrate?
 - Prefer new processes
 - Transfer/migration cost should be small compared to execution cost
 - Select processes with long execution times
 - **Location policy:** where to transfer/migrate the process?

Dynamic vs. Static Assignment

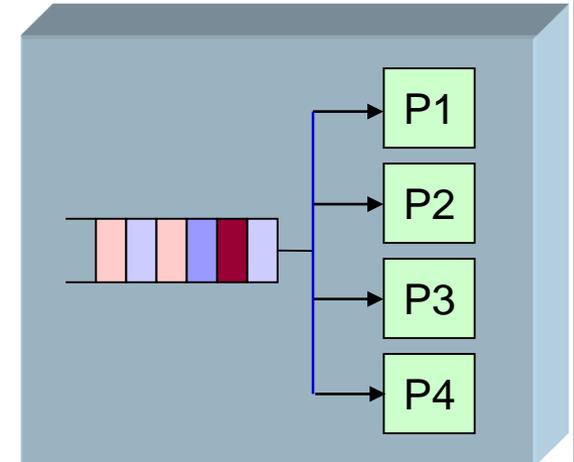
- The assignment of processes to processors can be **static** (till the process completes) or **dynamic** (may change each time a process is dispatched).
- **Static assignment** maintains a separate ready queue for each processor.
 - Loads can become unbalanced if the processes in one queue run longer than those in another queue.
 - Coordinating agent as a middle layer is needed to make assignment decisions [**Hot Research area!!**].
 - If processes on **different processors are dependent**, the schedulers on the processors must synchronize the processes according to some synchronization and resource access-control protocol.
- **Dynamic assignment**: use a **Global queue** and processes are allowed to migrate from processor to processor.
 - When a process returns to the **Ready** state, it returns to a shared queue and when it is dispatched will run on the next free processor.

Multiprocessor Scheduling: Global Queue

- A single scheduling algorithm is used to schedule all tasks.

Important Difference:

- Schedule to any available processor.
 - May cause overheads if a processes migrates to a new processor.
 - Static scheduling reduces this problem.
- Appropriate only on tightly-coupled/multi-core systems.

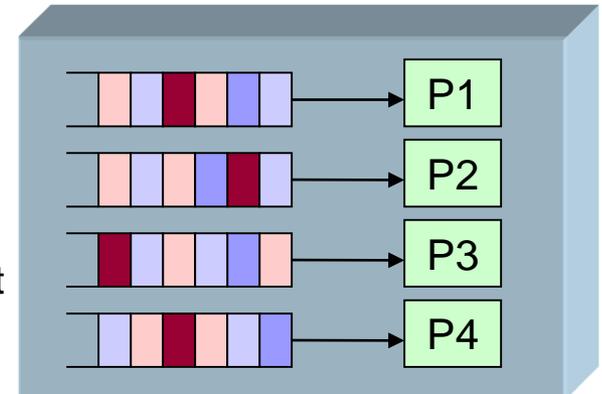


Multiprocessor Scheduling: Partitioning

- Partition processes so that each process always runs on the same processor.
 1. Dedicate short-term queue for each processor
 2. Schedule processes on each processor using uni-processor algorithms.
 3. Less overhead

Problems with Partitioning

- Variations in computation costs may necessitate a re-partitioning. Processor could be idle while another processor has a backlog.
- Which is good? Scheduling dependent processes or threads at the same time across multiple CPUs or to the same CPU?
- Shall partitions size be fixed or dynamically modified?
- **Equal-partition**: constant number of processes allocated evenly to all processors: **Low overhead**.
- **Dynamic**: dynamically reallocates processes to maximize utilization: **High utilization**.
- **Problematic because**:
 - Re-partitioning may cause **task migrations**...
 - Partitioning is designed to avoid tasks migrations.



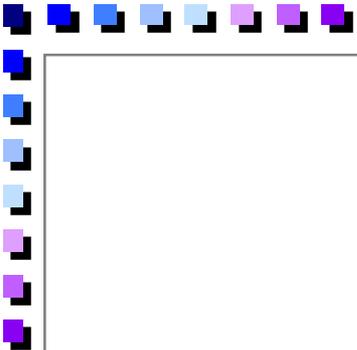
Multiprocessor Scheduling: Processor Affinity

- On a SMP, each processor has its own local cache or caches
- Scheduling a process where it has “Cash affinity” improves performance by reducing cache penalties.
- Suppose process P is running on CPU # 0 for this time slice
- The cache of CPU # 0 contains many entries copied from P’s address space (PCB)
- After context switch, P is assigned to CPU # 1
- The cache of CPU # 1 has no entries of P’s address space
- Must re-fetch from main memory
 - Very expensive
- If P was assigned to CPU # 0 instead, re-fetching would be minimized
- Some OSs try to re-assign a process to the CPU it was running last. This is called **Processor/Cash affinity**
- **Cash affinity** can be either soft or hard affinity
 - Soft affinity: “best effort” to get same CPU but no guarantee
 - Hard affinity: A process can request and get same CPU, guaranteed
- **Load Balancing:**
 - SMP systems must keep the workload balanced across all processors
 - Only necessary in systems where each processor has its own queue.

- Two general approaches
- **Push migration:** A specific process periodically checks the load on each processor.
 - If it finds an imbalance, it moves (pushes) processes to idle processors
- **Pull migration:** An idle processor pulls a waiting process from a busy processor.
- **Hybrid.** Uses both push and pull.
 - **Example:** Linux scheduler implements both.
 - Linux runs balancing algorithm every 200 milliseconds (**push**)
 - Or whenever the run queue for a processor is empty (**pull**)
- **Problem:** load balancing often counteracts the benefits of processor affinity
 - If using push or pull migration will take a process from its processor
 - This violates processor affinity
 - **No absolute rule governing which policy is best**
 - In some systems an idle processor always pulls a process from a non-idle process.
 - In other systems process are moved only if the imbalance exceeds a threshold.

Ch. 6 Process Synchronization

- To practice collaborative learning as we agreed, to foster self learning skill, and to help us progress in the course,
- You need to study Ch. 6 “**Process Synchronization**” and it will be included in the Major Exam II.
- Office hours can be used to answer any inquires about this chapter.
- **Your objectives out of this chapter is to know:**
 - Why synchronization? Necessity of synchronization in OS
 - How do processes work with resources that must be shared between them?
 - What is a critical section?
 - How to ensure that only one process can access the critical section?
 - What is atomic operation? It executes without interruptions, all or none.
 - Dangers of handling the critical section without synchronization.
 - Different algorithms to synchronize two processes enter of critical section.
- **Evaluating synchronization algorithms of handling a critical section where every algorithm should allow Mutual Exclusion, Progress and bounded waiting.**
 - Synchronization tools
 - Semaphores, and types of Semaphores
 - Incorrect usages of Semaphores
 - Classical problems of synchronization
 - Monitors
 - Synchronization methods in different OSs.



The End!!

Thank you

Any Questions?