Real-time Systems and CPU Scheduling

CPEN333 – System Software Engineering

2021 W2

University of British Columbia

Introduction

- We briefly discussed that a real-time system is a system for which we want the results within definite timescales; otherwise, it is considered that the system is just not working properly.

- We have also discussed that one of the responsibilities of the operating system is scheduling, where the OS decides at any time which process is going to run and on which CPU/core.

- In this set of slides, we review the characteristics of real-time systems, and also discuss CPU scheduling and its role in real-time systems.

Learning Objectives

- To discuss real-time systems
- To introduce CPU scheduling, which is the basis for multi-programmed operating systems
- To describe various CPU-scheduling algorithms
- To discuss real-time scheduling

Real-time systems

- A real-time system implies that there is something significant and important about its response time, involving real-time scheduling.

- A real-time system gives some level of guarantee that the system response is within some specified time constraint (also referred to as deadline).

Real-time systems classification

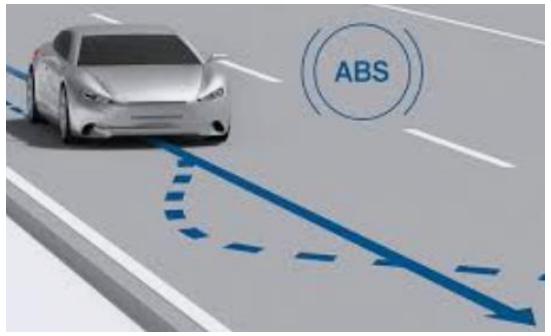
- What distinguishes a real-time system from a non-real-time system is the importance of timeliness.

- We can classify real-time systems on how critical timeliness is.
  - Hard real-time systems: the task must be serviced by its deadline, otherwise it is considered a system failure.

  - Soft real-time systems: timeliness is still of the essence, but missing the deadline is not considered a complete failure. Though the usefulness and the quality of service is considered degraded if the deadline is passed (and how much).
- The above definitions are not precise unfortunately, but still they help us communicate the expectation more clearly.

Embedded systems

- Embedded systems are specific-purpose (so not general-purpose) computing systems that are embedded within a bigger system. They are used to provide intelligence and computing to those devices.



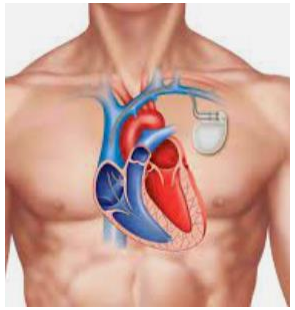
- Many practical real-time systems are embedded systems that use sensors to obtain real-world input, process it, and generate a response by controlling some actuators.

Examples of hard real-time systems

- Airbag control system

must act within milliseconds
to a fraction of a second

- Heart Pacemaker



- Sea-Viper missile defence system

must act within a few seconds to a
few minutes depending on the range

Example of soft real-time systems

- **Interactive web browsing**: We expect timeliness, and some level of delay is annoying but may not render it useless or failed.

- **Audio, video or multimedia systems**: Some occasional delay degrades the user experience, but may not be considered a complete failure.

Note that under certain conditions/applications we may categorize apps like the above differently.

Scheduling and responsiveness

- In a modern operating system, many processes are to run concurrently and it is the job of the operating system to efficiently make that happen.

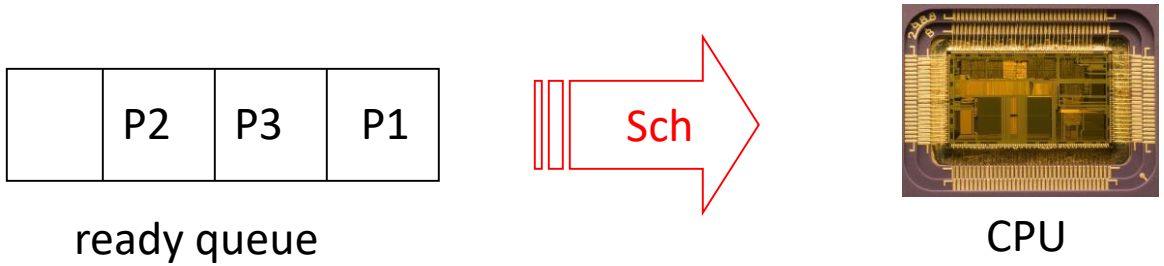
- So scheduling is an important part of the operating system responsibility for process management. An OS may schedule processes, threads or both.

- In a real-time system, tasks must be scheduled more strictly so that the required timeliness is achieved.

CPU Scheduler

- The CPU is one of the primary computer resources and should be scheduled before use

➢ Scheduling is the task of selecting one process from among the processes in memory that are ready to execute, and allocate the CPU to it

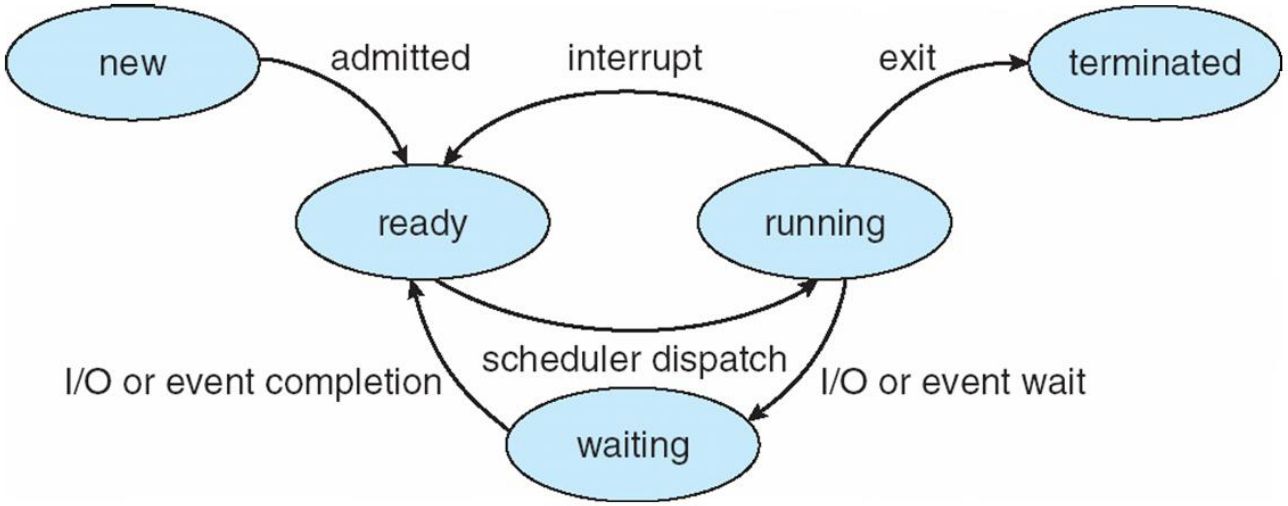


➢ The success of CPU scheduling depends on the observed property of processes

- ❖ In our daily life, we are mostly used to first-in first-out queues (FIFO).
  - The above ready queue though is not necessarily FIFO.

CPU Scheduler (cont.)

- 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 state
  4. Terminates



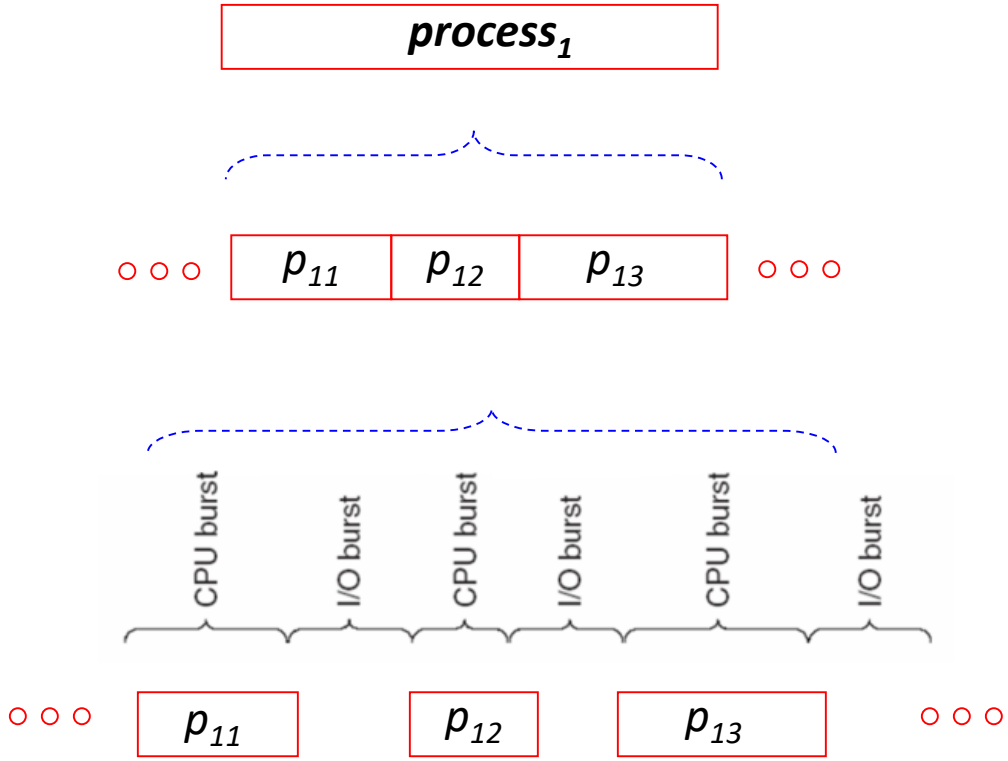
CPU Scheduler (cont.)

- In the previous slide, scheduling under 1 (running to waiting state) and 4 (terminating) is non-preemptive (or cooperative)
  - Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state (for example, because it needs to do IO)
  - Note that here non-preemptive terminating is when the task is done and voluntarily terminates.

- All other scheduling is preemptive
  - requires special hardware (e.g. a timer, interrupts, …) as well as OS support
  - new mechanisms are needed to coordinate access to shared data (for example synchronization that we already discussed)

Process Partition into Smaller CPU Bursts

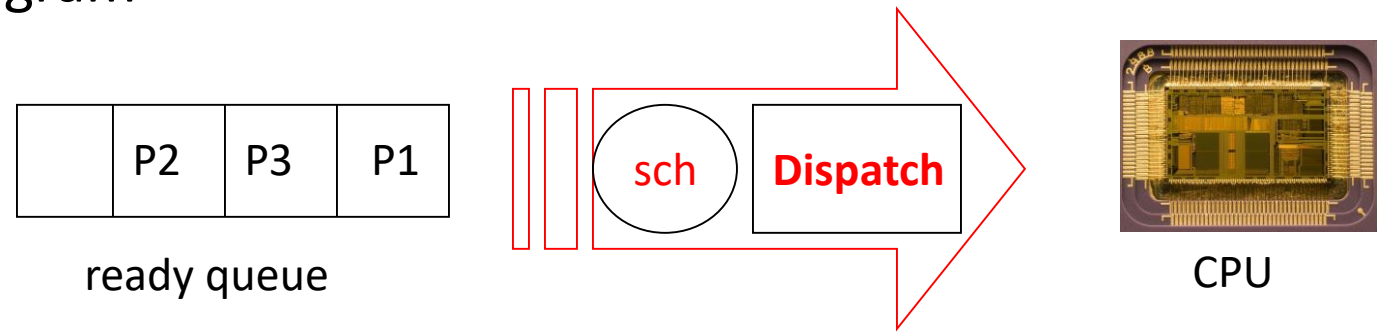
- Note that each process (here, in this example, ***process$_1$***) is generally divided into a number of CPU bursts.
- This is because a process needs to do IO once in a while. Also the OS may take the CPU away from a process to allow concurrency and responsiveness.
- In an IO-bound process, there are usaually many short CPU bursts, but in a CPU-bound process, there are a few long CPU bursts.



$$\text{In general: } process_i = \sum_k P_{ik}$$

Dispatcher

- When a process is selected by the scheduler, the dispatcher module gives control of the CPU to the process selected
  - This involves:
    - switching context
    - jumping to the proper location in the user program to restart that program



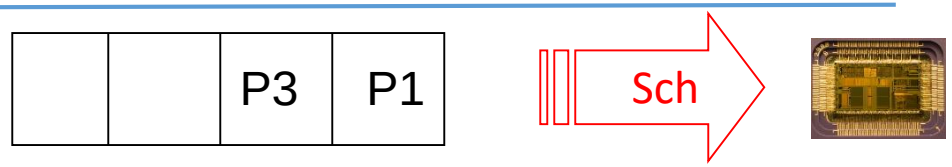
- The dispatcher must be as fast as possible
  - Dispatch latency is the time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

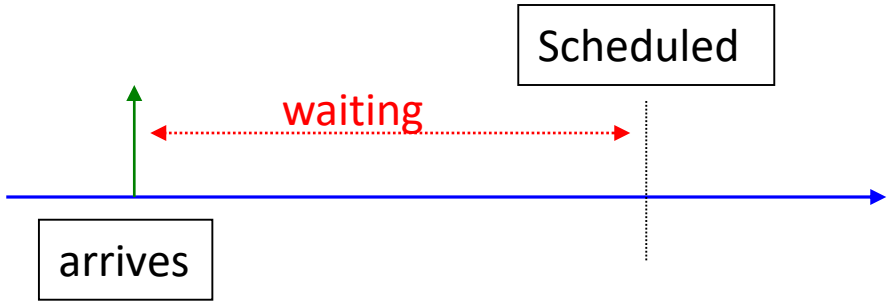
- Different scheduling algorithms have different properties.
  - In choosing which scheduling algorithm to use in a particular situation, we must consider its properties.
- Many criteria have been used for comparing scheduling algorithms.
  - **Waiting time** – amount of time a process has been waiting in the ready queue
  - **Turnaround time** – amount of time to execute a particular process (includes waiting time in memory and ready queue, executing in CPU and doing IO)
  - **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not the final/overall output.
    - In an interactive system we may be interested din the response time.
  - **CPU utilization** – keep the CPU as busy as possible
  - **Throughput** – # of processes that complete their execution per time unit

Scheduling Criteria - Example

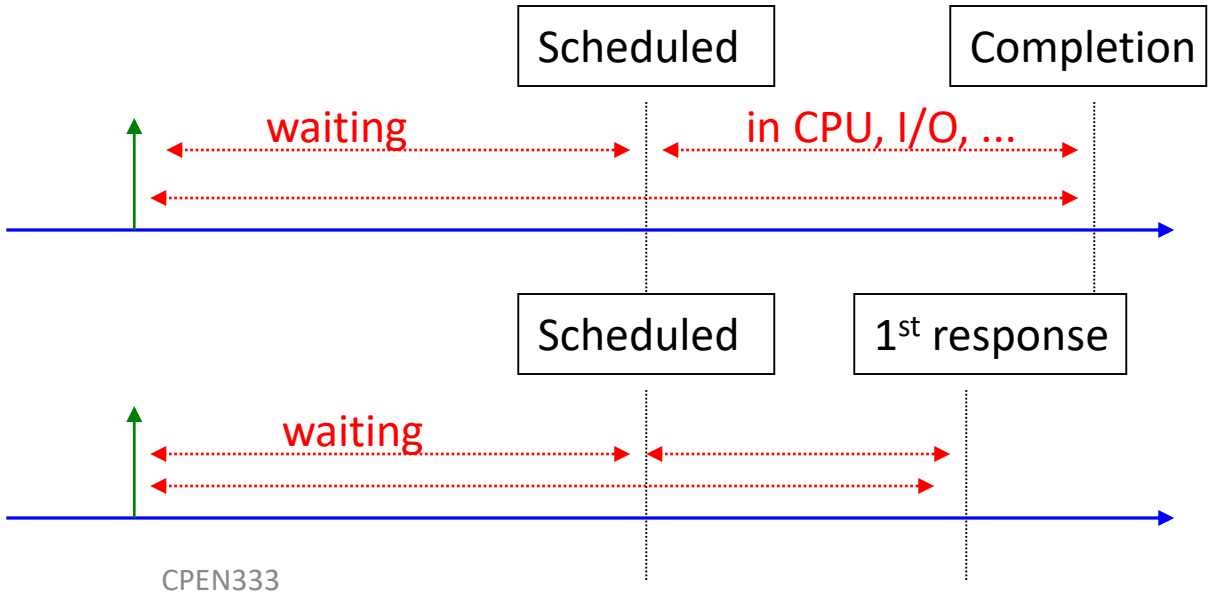
- Assume a CPU burst P3 to be scheduled:



- Waiting time



- Turnaround time



- Response time

Scheduling Algorithm Optimization Criteria

- We usually aim to minimize or maximize a criterion for optimization (choosing the best element from a set of available alternatives)
  - aim to minimize: turnaround time, or waiting time, or response time
  - aim to maximize: CPU utilization, or throughput
- In most cases, we optimize an average measure.
  - However, under many other circumstances, the optimization may be on other statistical measures, such as minimum or maximum values, or variance.

Scheduling Algorithms

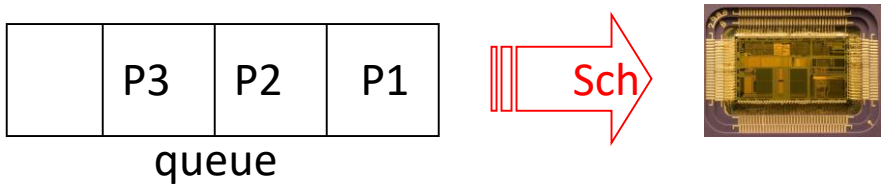
- There are many different scheduling algorithms.

- In the following slides, we are going to study:
  - First-Come, First-Served (FCFS)
  - Shortest-Job-First (SJF)
  - Priority Scheduling
  - Round Robin (RR)

First-Come, First-Served (FCFS) Scheduling

- We are very familiar with the first-come first-served (FCFS, aka first-in first-out (FIFO)) queuing in our normal lives.
  - bus line, bank queue, …
  - It is based on our perception of fairness.

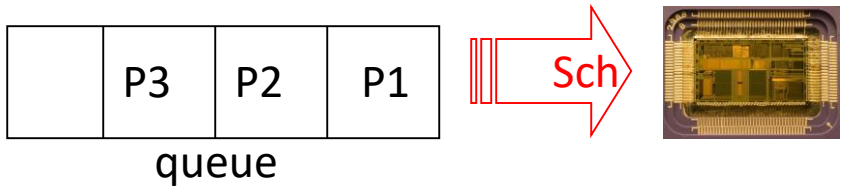
- In this scheme, the process that requests the CPU first is allocated the CPU first, and so forth.



In this example, P1 is scheduled first, then P2, then P3

FCFS Example 1

- Example: Suppose that CPU bursts of processes arrive in the order: P1, P2, P3



Process	Burst Time
P_1	24
P_2	3
P_3	3

- A Gantt chart is a bar chart that illustrates a particular schedule, including start and finish times. So, the Gantt Chart for the above example schedule is:

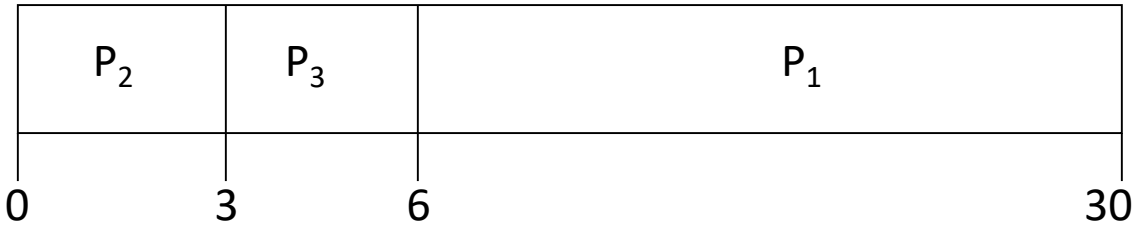
- The waiting time for P1 is 0, for P2 is 24, and for P3 is 27.
- Thus the average waiting time is equal to (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont)

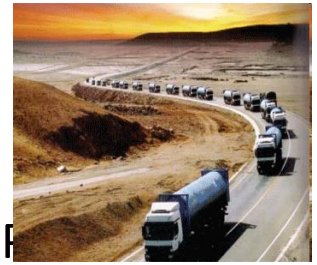
- Now let's consider another scenario; suppose that this time the same processes arrive in the following order: P2 , P3 , P1

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart for the schedule is changed to:



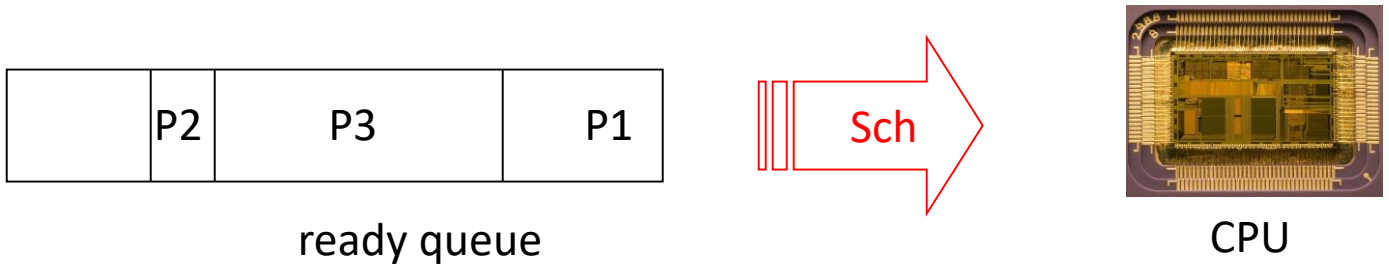
- Thus the waiting time for P1 is 6, for P2 is 0, and for P3 is 3.
  - The average waiting time is now: (6 + 0 + 3)/3 = 3
  - which is much better than the previous case



- The above two scenarios show that the average time under FCFS policy is generally not minimal and may vary substantially.
  - Convoy effect: all other short processes should wait for a long process to get off the CP

Shortest-Job-First (SJF) Scheduling

- A different approach to CPU scheduling is the shortest-job-first scheduling algorithm. We start with non-preemptive SJF.
- This algorithm associates with each process the length of its next CPU burst.
- When the CPU is available, it is assigned to the process with the smallest next CPU burst



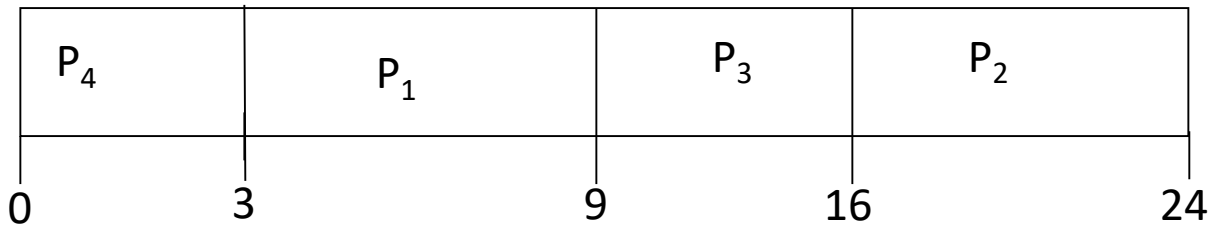
In this example, P2 is the smallest so it is picked first (out of P1, P2, P3)

SJF Scheduling (cont)

- Let's look at an example:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- The Gantt chart for this SJF scheduling is:



- The average waiting time is now: (3 + 16 + 9 + 0) / 4 = 7

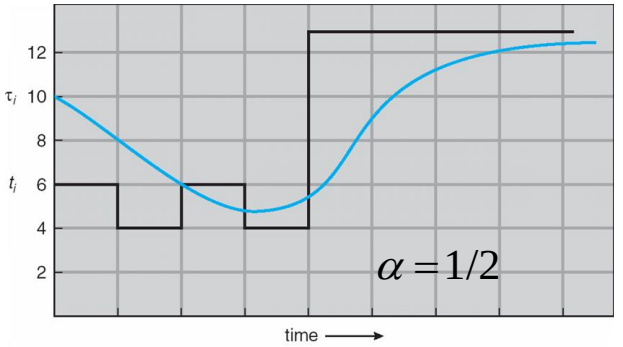
- SJF is optimal, in that it gives the minimum average waiting time for a given set of processes
  - The real difficulty is knowing the length of the next CPU request

Determining Length of Next CPU Burst

- A process consists of a number of CPU bursts. We may *not know* the length of the next CPU burst (which is to be scheduled), but we may be able to *predict* its value.
  - With short-term CPU scheduling, we can try to approximate SJF scheduling
- In order to compute an approximation of the length of the next CPU burst, we use the length of the previous CPU bursts (which is known), using exponential averaging

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. a parameter α, $0 \le \alpha \le 1$
4. Define :

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$$



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

Examples of Exponential Averaging

- The parameter, α, controls the relative weight of recent and past history in our prediction.

- Let's consider the following two extremes: $(\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n)$
  - if $\alpha = 0$
    - $\tau_{n+1} = \tau_n$
    - Recent history does not count
  - if $\alpha = 1$
    - $\tau_{n+1} = t_n$
    - Only the actual last CPU burst counts

- If we expand the formula, we get: $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\alpha\, t_{n-1} + \ldots + (1-\alpha)^j \alpha\, t_{n-j} + \ldots + (1-\alpha)^{n+1} \tau_0$
  - Since both α and $(1-\alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

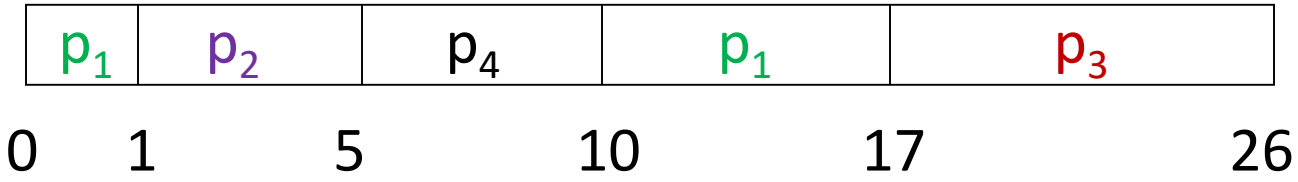
Preemptive SJF (cont)

- An SJF algorithm can be preemptive, that is, it will preempt a currently executing process. The preemptive SJF is called shortest-remaining-time-first.
- Consider the following example:

Note: So far in all examples the arrival time values were not considered. In this example, we are considering arrival times.

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- The Gantt chart for this SJF scheduling is:



- The average waiting time is now: [(10-1)+(1-1)+(17-2)+ (5-3)]/4 = 6.5

Priority Scheduling

- In the priority scheduling,
  - a priority number (integer) is associated with each process
    - usually: smallest integer ≡ highest priority
  - and the CPU is allocated to the process with the highest priority

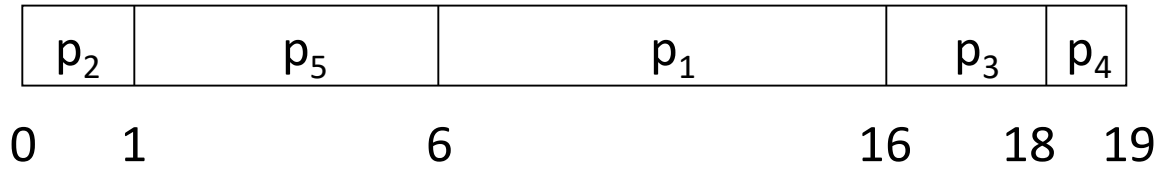
- The SFJ is a special case of the general priority scheduling algorithm.
  - In SJF, the priority is the predicted next CPU burst time

Priority Scheduling Example

- Consider the following example:

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

- The Gantt chart for this priority scheduling is:



- The average waiting time is now: (6+0+16+18+1)/5 = 8.2

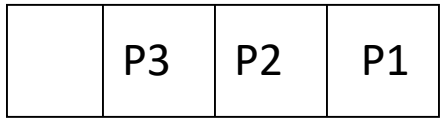
Starvation and aging technique

- A major problem with the priority scheduling algorithm is the possibility of indefinite blocking or Starvation
  - This algorithm may leave some low priority processes waiting indefinitely

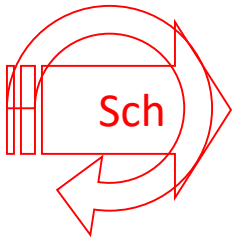
- A solution to the above problem is the aging technique:
  - Aging is the technique of gradually increasing the priority of processes that are waiting in the system as time progresses

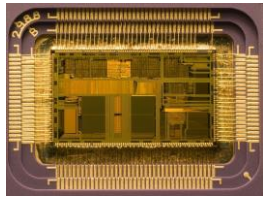
Round Robin (RR) Scheduling

- The Round Robin scheduling algorithm is designed especially for time-sharing systems.
- Each process gets a small unit of CPU time (called *time quantum* or *time slice*), usually with a length of 10 to 100 milliseconds.
  - After this time has elapsed, the process is preempted and added to the end of the ready queue.



ready queue





CPU

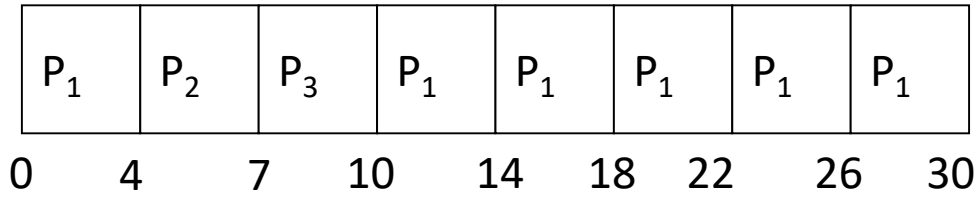
- In RR, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process).
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.

Example of RR

- Let's take a look at an example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Assuming a time quantum of 4, the Gantt chart is:

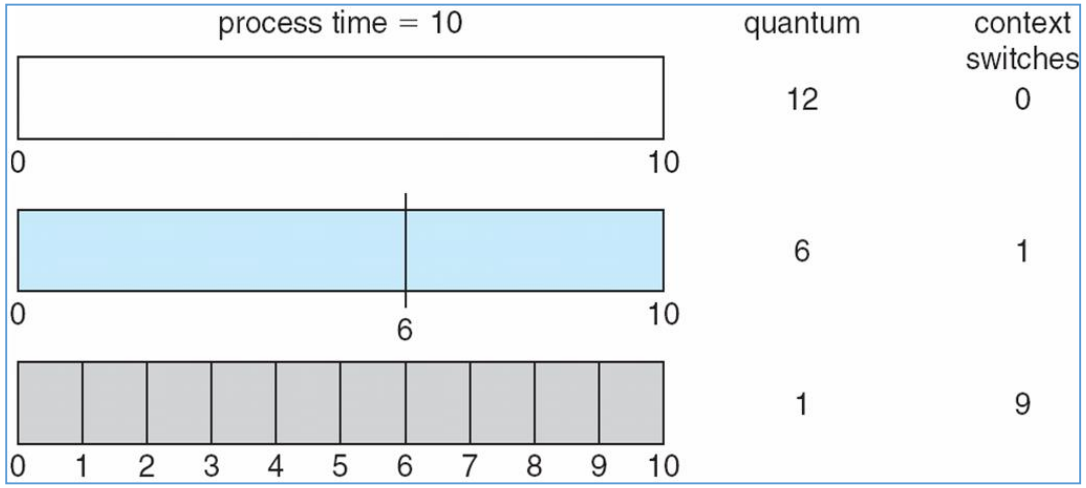


- The average waiting time is now:[(10-4)+4+7]/3 = 5.66

- Typically, higher average turnaround than SJF, but better *response*

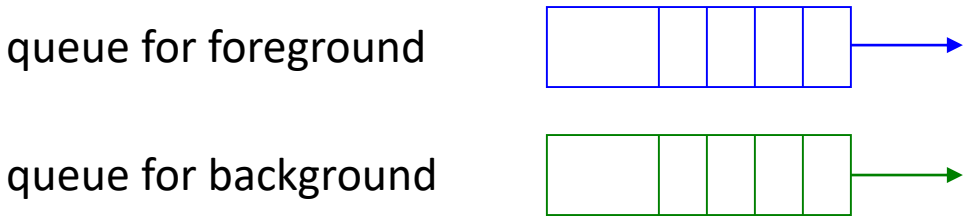
Time Quantum and Context Switch Time

- The performance of the RR algorithm depends on
  - the size of the time quantum (q)
    - q very large ⇒ FIFO
    - q small ⇒ processor sharing: This creates the appearance that each process is running on its own CPU at 1/n the speed of the real processor
    - q must be large with respect to context switch, otherwise overhead is too high
  - the context switching effect

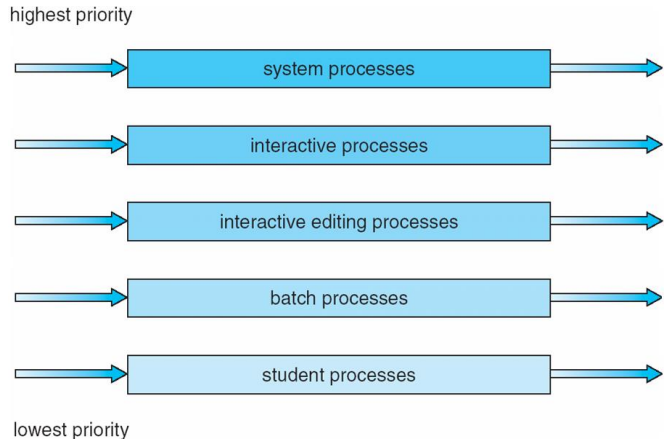


Multilevel Queue

- In many situations, we prefer to classify the processes into different groups.
  - For example, a common division is made between *foreground* and *background* processes.
  - These two types have different scheduling requirements or priorities.

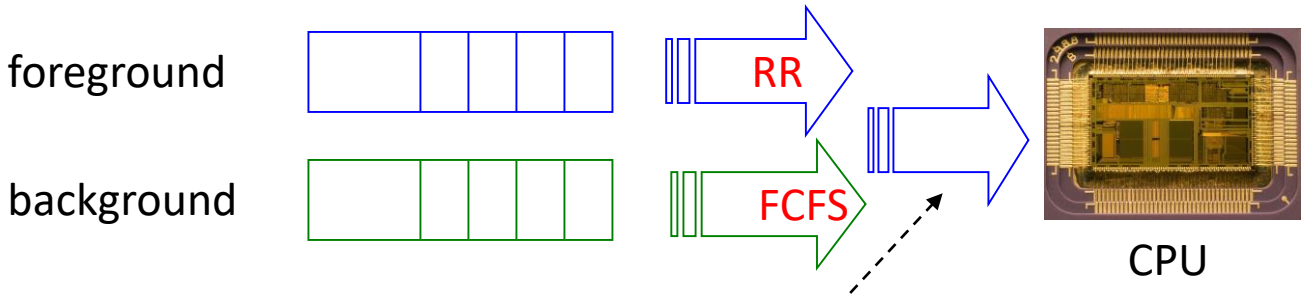


- A multi-level queue scheduling algorithm partitions the ready queue into several separate queues.



Multilevel Queue Scheduling

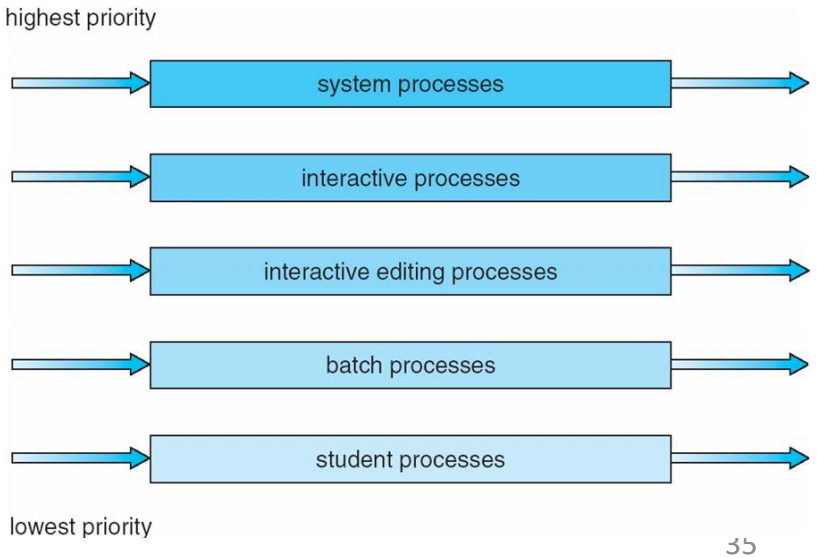
- In multi-level queue scheduling,
  - each process is assigned to some queue,
    - based on some property of the process (e.g. priority, process type, memory size, …)
  - each queue may have its own scheduling algorithm
    - for example, for the two-level queue:
      - the foreground queue might be scheduled by RR
      - the background queue might be scheduled by FCFS



  - in addition, there must be scheduling among the queues

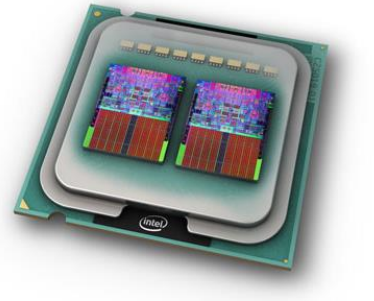
Multilevel Queue Scheduling (cont)

- One possibility is to let each queue have absolute priority over lower-priority queues.
  - fixed priority scheduling; There is the possibility of starvation.
    - e.g., serve all from foreground then from background.
- Another possibility is to time slice among the queues.
  - each queue gets a certain amount of CPU time which it can schedule amongst its processes
    - e.g. 80% to foreground in RR and 20% to background in FCFS
- Here is another example:



Multiple-Processor Scheduling

- Our discussion thus far has focused on the problems of scheduling the CPU in a system with a single processor



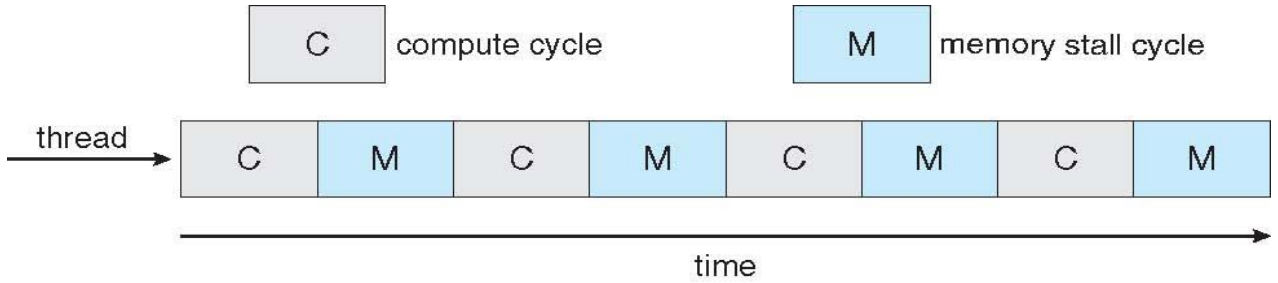
- If multiple CPUs or multiple cores are available:
  - *load sharing* become possible,
  - however, the scheduling problem becomes more complex

- Load balancing attempts to keep the workload evenly distributed across all processes in symmetric multiprocessing (SMP).

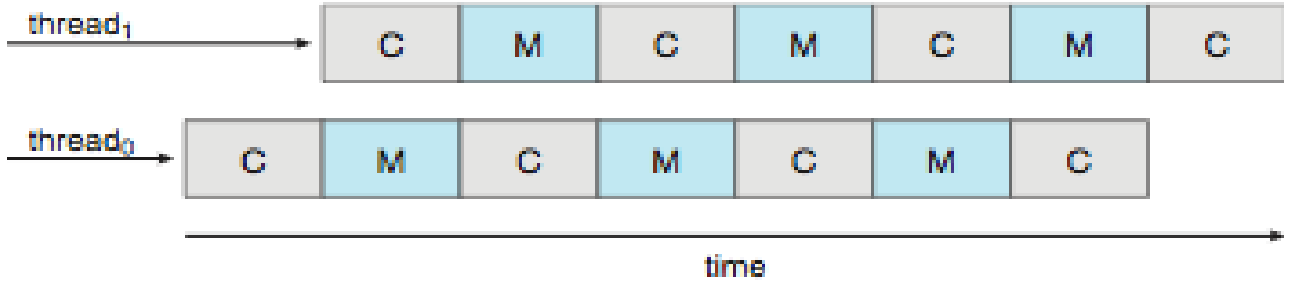
- Processor affinity – process has affinity for the processor on which it is currently running
  - **soft affinity** (it is possible for a process to migrate between processors)
  - **hard affinity**

Multicore Scheduling

- Most processor nowadays are multicore.
- Memory stall is the amount of time a processor is waiting for data to be available when accessing memory.
  - In this example, the processor core spends up to 50 percent of its time in memory stall.



- Multiple threads can be assigned to each core.
  - To make progress on another thread when a thread is in memory stall.



Real-time Scheduling

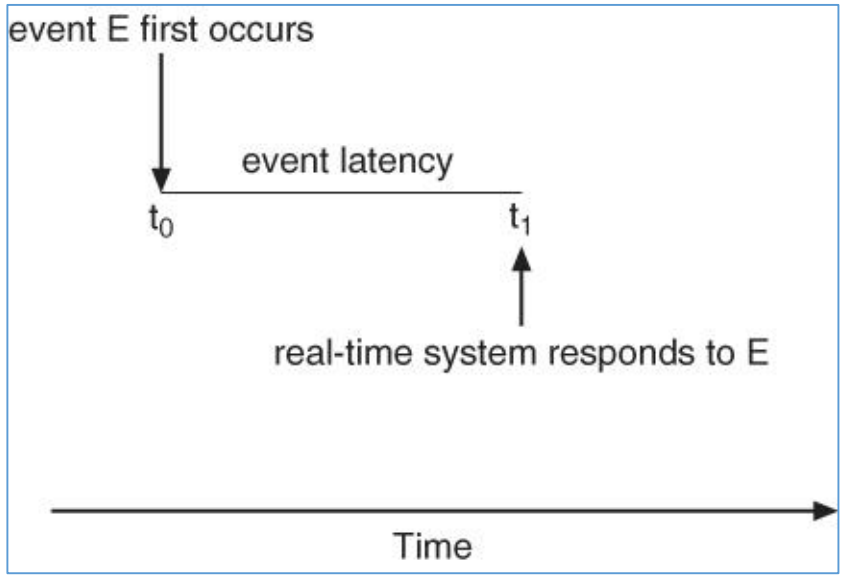
- How to guarantee real-time behaviour?
  - **keeping the system as simple as is has to be**: the more a system has to deal with, the harder it can guaranteed timeliness.
  - **utilizing hardware**: using hardware interrupt and capabilities may enable a system to guarantee real-time behaviour.
  - **well-designed scheduling** (or even using specialized real-time OS such as RTOS)

- Real-time scheduling can present obvious challenges. We must control and minimize latencies.

Event latency

- Many real-time systems are event-driven.
  - They wait for an event in real time to happen and to respond to.
  - An event can be
    - in software (a timer expires) or
    - in hardware (a self-driving car's sensor detects an obstacle and interrupts the CPU for action)

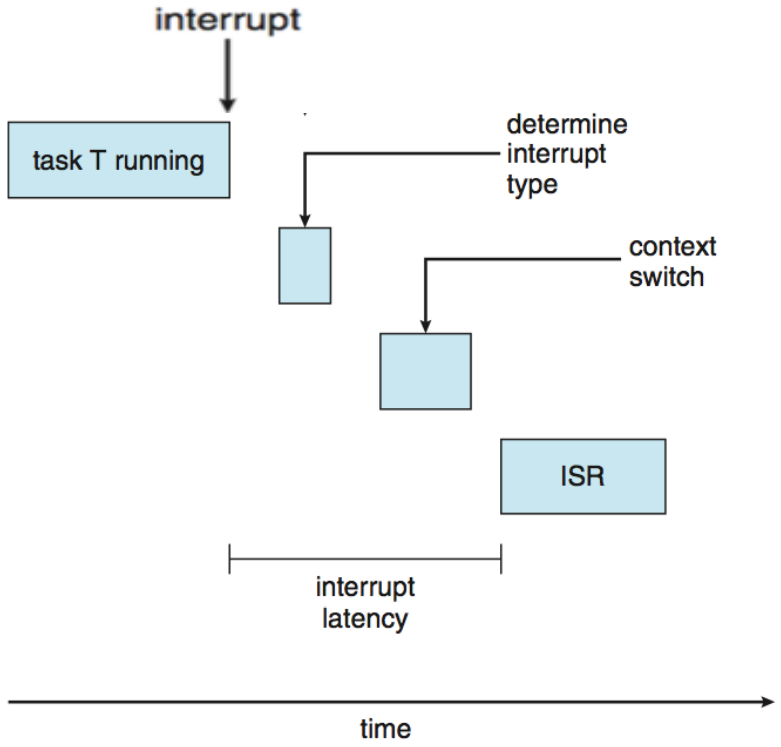
- Event latency: amount of time that elapses from an event occurs to when it is serviced.



Types of latencies

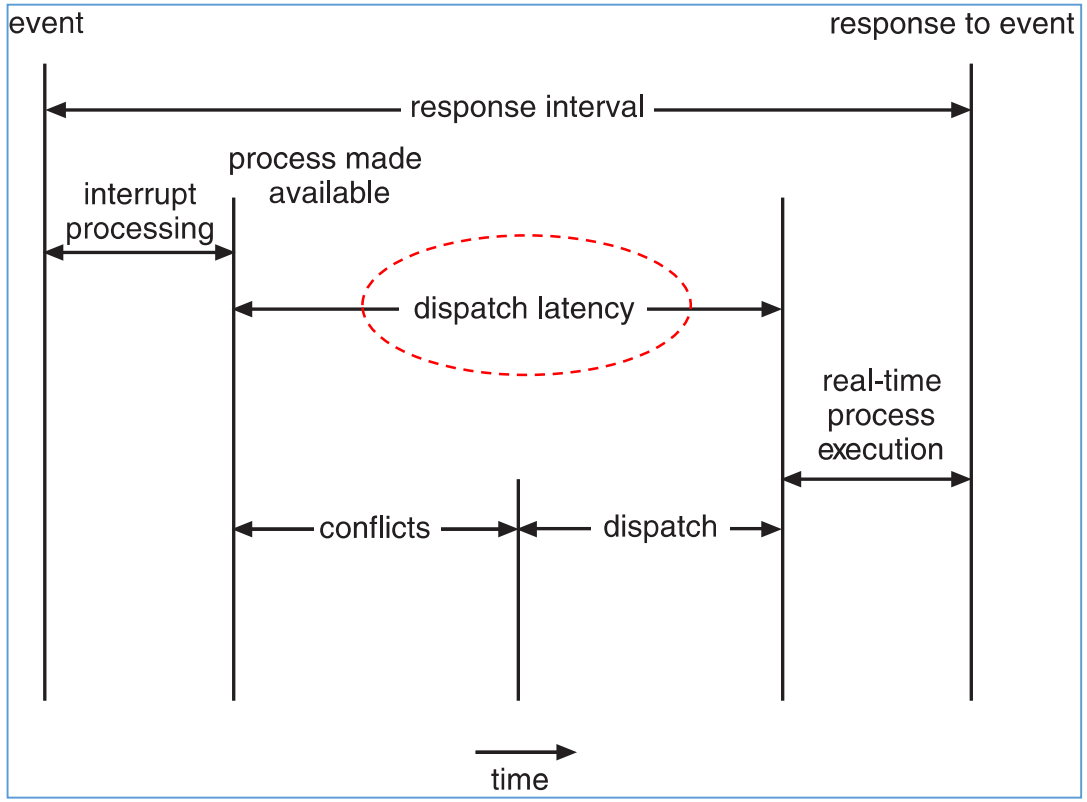
- Two types of latencies affect performance: Interrupt latency and dispatch latency
- Interrupt latency: time from the arrival of interrupt to the start of the routine that services interrupt
  - ISR = interrupt service routine

Interrupt latency =
time to finish executing current instruction
+ time to determine interrupt type
+ context switch time



Real-Time CPU Scheduling (Cont.)

- Dispatch latency: time for scheduler to take current process off CPU and switch to another

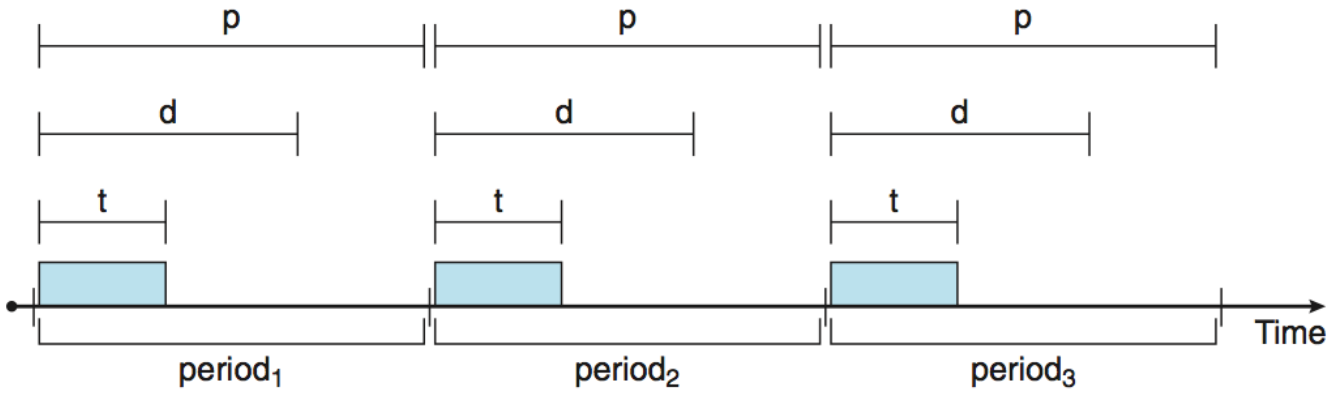


- Conflict phase of dispatch latency:
  - Preemption of any process running in kernel mode (OS)
  - Release by low-priority process of resources needed by high-priority processes

dispatch latency

Priority-Based RT Scheduling

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



Algorithm Evaluation

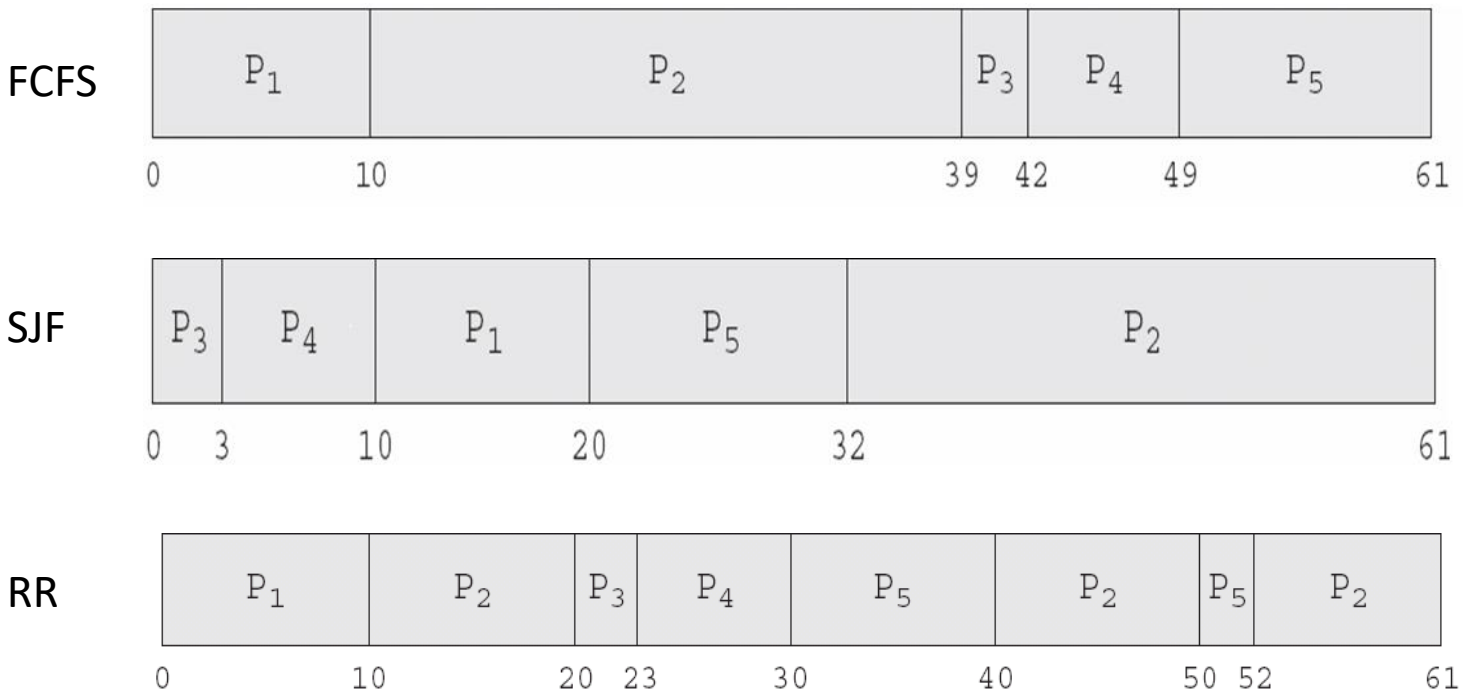
- How do we select a scheduling algorithm to implement?
  - First we need to select the desired criteria
  - Then we need to evaluate the algorithms under consideration

- There are three main evaluation methods:
  - Deterministic modeling: takes a particular predetermined workload and defines the performance of each algorithm for that workload
  - Queueing models: a stochastic method based on the theory of queues
    - Usually involves rather difficult mathematical analysis
  - Simulation: We may also simulate the algorithms and compare the results.

Deterministic Evaluation Example

- Example of evaluation based on average waiting time

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12



FCFS: average waiting time = 28

SJF: average waiting time = 13

RR: average waiting time = 23

References

- Some sections from chapter 6 of the Operating Systems Concepts

Acknowledgement: This set of slides is partly based on the PPTs provided by the Wiley's companion website for the operating system concepts book (including textbook images, when not explicitly mentioned/referenced).