Chapter 05
CPU Scheduling

The key to Multiprogramming

Outline

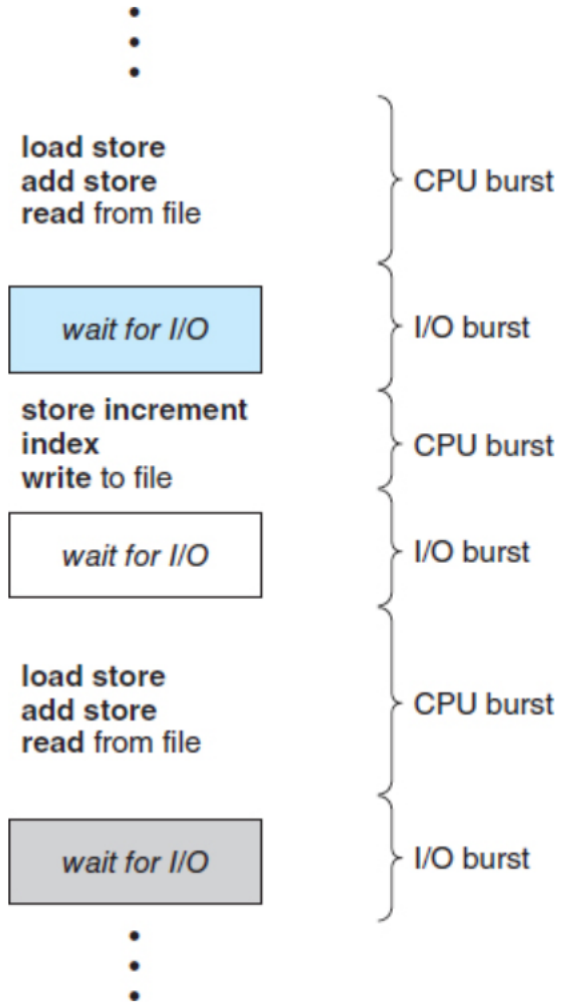
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Algorithm Evaluation

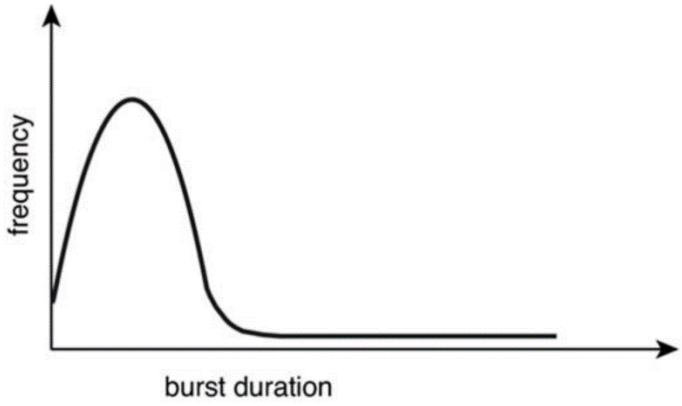
Objectives

- Describe various CPU scheduling algorithms.
- Assess CPU scheduling algorithms based on scheduling criteria.
- Explain the issues related to multiprocessor and multicore scheduling.
- Describe various real-time scheduling algorithms.
- Apply modeling and simulations to evaluate CPU scheduling algorithms.

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a cycle of CPU execution and I/O wait
- CPU burst followed by I/O burst
- CPU burst distribution is of main concern





CPU Scheduler

- Short-term scheduler selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways

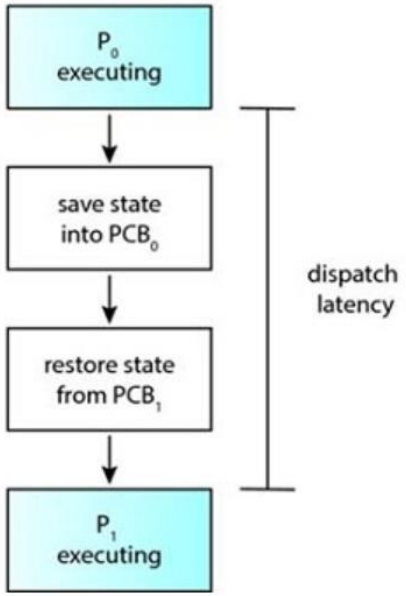
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates

CPU Scheduler

- No choice for 1 and 4
  - A new process must be selected for execution.
  - Nonpreemptive (cooperative) if scheduling is only occurred under these two circumstances.
- Virtually all modern operating systems use preemptive scheduling
- However, preemption may cause problem
  - Consider preemption when accessing **shared data**
    - Occur when one process is updating the share data
  - Consider preemption while in **kernel mode**
    - Occur when changing important kernel data
  - Consider preemption in **interrupts**
    - Occur during crucial OS activities

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- Dispatch latency – time it takes for the dispatcher to stop one process and start another running



Scheduling Criteria

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

Scheduling Algorithm Optimization Criteria

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

Scheduling Algorithms

- First-Come, First-Served Scheduling (FCFS)
- Shortest-Job-First Scheduling (SJF)
- Round-Robin Scheduling (RR)
- Priority Scheduling
- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling

First- Come, First-Served Scheduling

- Abbr. FCFS scheduling
- The process that requests the CPU first is allocated the CPU first
- Managed with a FIFO queue

FCFS Scheduling

FCFS Scheduling

Process
P1

FCFS scheduling for the five processes that arrived in the order P1, P2, P3, P4, P5

FCFS Scheduling

FCFS Scheduling

Process
P1
P2

FCFS scheduling for the five processes that arrived in the order P1, P2, P3, P4, P5

FCFS Scheduling

FCFS Scheduling

Process
P1
P2
P3

FCFS scheduling for the five processes that arrived in the order P1, P2, P3, P4, P5

FCFS Scheduling

FCFS Scheduling

Process
P1
P2
P3
P4

FCFS scheduling for the five processes that arrived in the order P1, P2, P3, P4, P5

FCFS Scheduling

FCFS Scheduling

Process
P1
P2
P3
P4
P5

FCFS scheduling for the five processes that arrived in the order P1, P2, P3, P4, P5

FCFS Scheduling

FCFS Scheduling

Process	Burst Time
P1	7
P2	3
P3	12
P4	5
P5	9

associated with each process is a **burst time**

FCFS Scheduling

FCFS Scheduling

Process	Burst Time
P1	7
P2	3
P3	12
P4	5
P5	9

Gantt Chart

we will use a **Gantt Chart** to demonstrate the order in which processes are selected for scheduling according to the FCFS scheduling algorithm

FCFS Scheduling

FCFS Scheduling

Process	Burst Time
P1	7
P2	3
P3	12
P4	5
P5	9

Gantt Chart

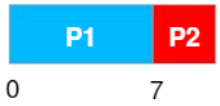
process P1 is selected first as it is the first process to arrive; it begins running at time 0

FCFS Scheduling

FCFS Scheduling

Process	Burst Time
P1	7
P2	3
P3	12
P4	5
P5	9

Gantt Chart



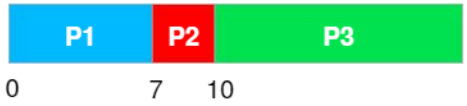
process P1 runs to completion after which process P2 is selected to run at time 7

FCFS Scheduling

FCFS Scheduling

Process	Burst Time
P1	7
P2	3
P3	12
P4	5
P5	9

Gantt Chart



process P3 is selected next and begins running at time 10

FCFS Scheduling

FCFS Scheduling

Process	Burst Time
P1	7
P2	3
P3	12
P4	5
P5	9

Gantt Chart

at time 22, process P4 is selected to run

FCFS Scheduling

FCFS Scheduling

Process	Burst Time
P1	7
P2	3
P3	12
P4	5
P5	9

Gantt Chart

because process P5 arrived last, it is the last process selected by the FCFS scheduling algorithm and begins running at time 27

FCFS Scheduling

FCFS Scheduling

Process	Burst Time
P1	7
P2	3
P3	12
P4	5
P5	9

Gantt Chart

at time 36, process P5 finishes running

FCFS Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3 The Gantt Chart for the schedule is:

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling

Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:

- Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- Convoy effect - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes
- FCFS is nonpreemptive.
  - Troublesome for interactive systems

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
  - ***shortest-next-CPU-burst*** algorithm

SJF Scheduling

SJF Scheduling

Process
P1
P2
P3
P4

SJF scheduling for the four processes P1, P2, P3, P4

SJF Scheduling

SJF Scheduling

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

associated with each process is a **burst time**

SJF Scheduling

SJF Scheduling

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

Gantt Chart

we will use a **Gantt Chart** to demonstrate the order in which processes are selected for scheduling according to the SJF scheduling algorithm

SJF Scheduling

SJF Scheduling

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

Gantt Chart

0

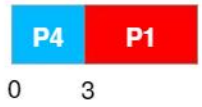
process P4 is selected first as it is the process with the shortest CPU burst; it begins running at time 0

SJF Scheduling

SJF Scheduling

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

Gantt Chart



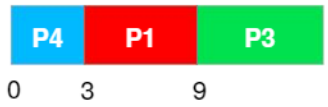
process P4 runs to completion, after which process P1 is selected as it has the next shortest CPU burst; process P1 begins to run at time 3

SJF Scheduling

SJF Scheduling

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

Gantt Chart



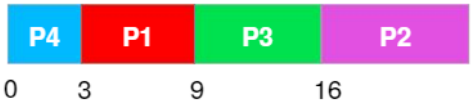
process P1 runs to completion, and process P3 is selected as it has the next shortest CPU burst; process P3 begins to run at time 9

SJF Scheduling

SJF Scheduling

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

Gantt Chart



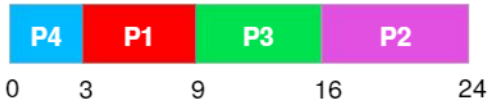
as process P2 has the longest CPU burst, it is scheduled last and begins to run at time 16

SJF Scheduling

SJF Scheduling

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

Gantt Chart



process P2 finishes executing at time 24

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart

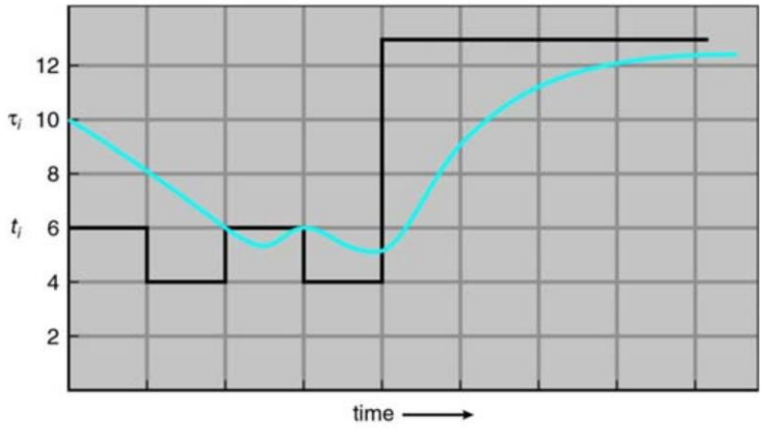
- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

SJF Scheduling

- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user

Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging
  1. t_n = actual length of n^{th} CPU burst
  2. τ_{n+1} = predicted value for the next CPU burst
  3. $\alpha, 0 \le \alpha \le 1$
  4. Define: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$
- Commonly, α set to ½ and τ_0 set to a constant such as system average



Examples of Exponential Averaging

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

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

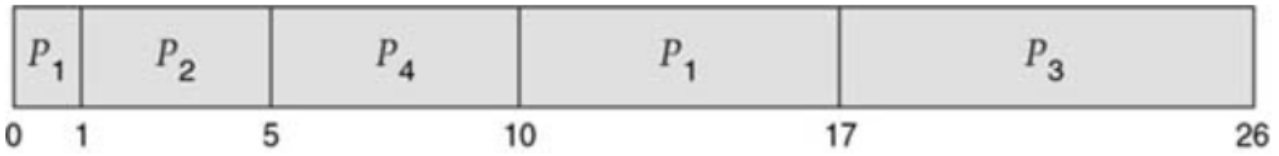
- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Example of Shortest-remaining-time-first

- Preemptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling
- Now we add the concepts of varying arrival times and preemption to the analysis

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

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

Round Robin (RR) Scheduling

- Each process gets a small unit of CPU time (time quantum *q*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- Timer interrupts every quantum to schedule next process
- Implemented as FIFO queue

RR Scheduling

RR Scheduling

Process
P1
P2
P3
P4
P5

RR scheduling for the five processes that arrived in the order P1, P2, P3, P4, P5

RR Scheduling

RR Scheduling

Process	Burst Time
P1	7
P2	3
P3	12
P4	5
P5	9

associated with each process is a **burst time**

RR Scheduling

RR Scheduling

Process	Burst Time
P1	7
P2	3
P3	12
P4	5
P5	9

Gantt Chart

we will use a **Gantt Chart** to demonstrate the order in which processes are selected for scheduling according to the RR scheduling algorithm

Time Quantum

the **time quantum** is the amount of time a process can run before it is preempted and the CPU is assigned to another process

the time quantum for this example is three units of time

RR Scheduling

RR Scheduling

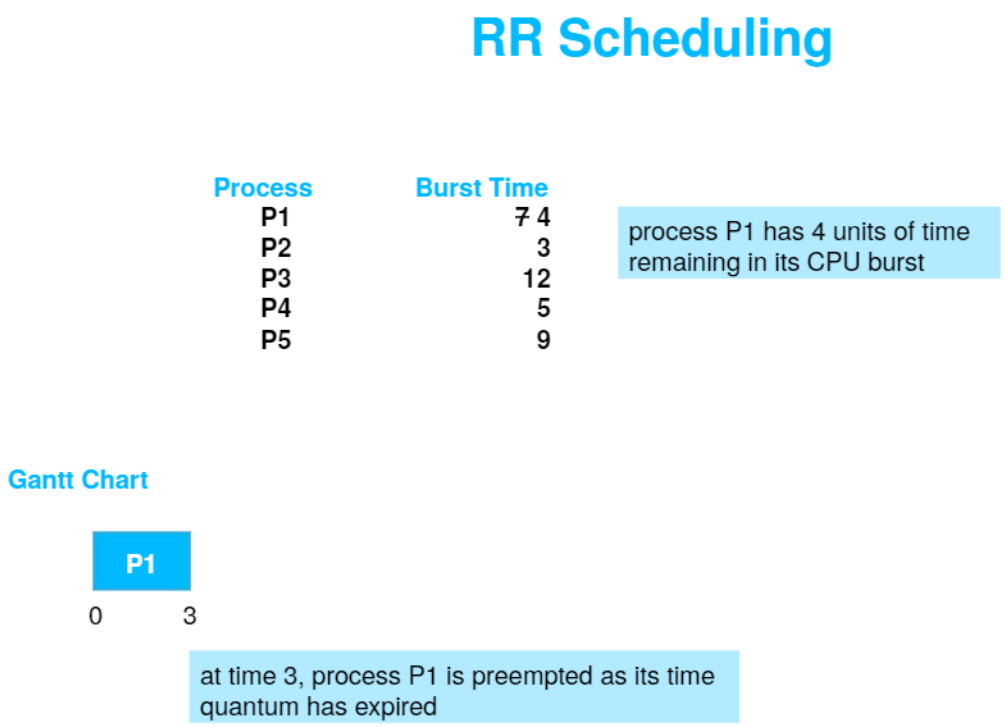
Process	Burst Time
P1	7
P2	3
P3	12
P4	5
P5	9

Gantt Chart

0

process P1 is selected first as it is the first process to arrive; it begins running at time 0 where it will run for the duration of a time quantum – three units

RR Scheduling

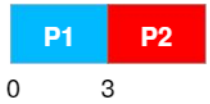


RR Scheduling

RR Scheduling

Process	Burst Time
P1	~~7~~ 4
P2	3
P3	12
P4	5
P5	9

Gantt Chart



process P2 will run next, beginning at time 3

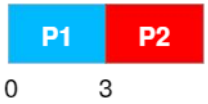
RR Scheduling

RR Scheduling

Process	Burst Time
P1	~~7~~ 4
P2	~~3~~ 0
P3	12
P4	5
P5	9

process P2 completes its CPU burst

Gantt Chart



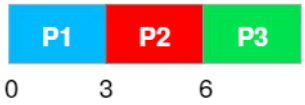
process P2 runs for the duration of a time quantum

RR Scheduling

RR Scheduling

Process	Burst Time
P1	~~7~~ 4
P2	~~3~~ 0
P3	12
P4	5
P5	9

Gantt Chart



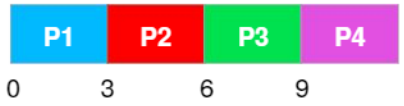
process P3 is selected next

RR Scheduling

RR Scheduling

Process	Burst Time
P1	~~7~~ 4
P2	~~3~~ 0
P3	~~12~~ 9
P4	5
P5	9

Gantt Chart



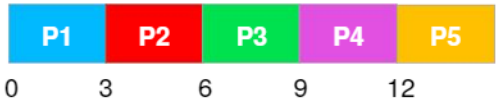
followed by process P4 at time 9

RR Scheduling

RR Scheduling

Process	Burst Time
P1	~~7~~ 4
P2	~~3~~ 0
P3	~~12~~ 9
P4	~~5~~ 2
P5	9

Gantt Chart



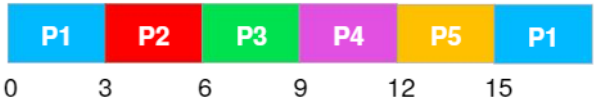
process P5 begins running at time 12

RR Scheduling

RR Scheduling

Process	Burst Time
P1	~~7~~ 4
P2	~~3~~ 0
P3	~~12~~ 9
P4	~~5~~ 2
P5	~~9~~ 6

Gantt Chart



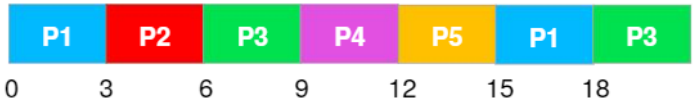
process P1 runs for a time quantum beginning at time 15

RR Scheduling

RR Scheduling

Process	Burst Time
P1	~~7~~ ~~4~~ 1
P2	~~3~~ 0
P3	~~12~~ 9
P4	~~5~~ 2
P5	~~9~~ 6

Gantt Chart



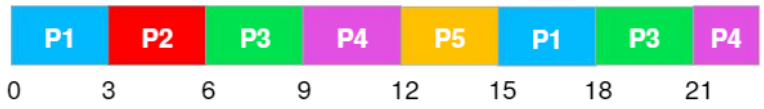
process P3 runs for a time quantum beginning at time 18

RR Scheduling

RR Scheduling

Process	Burst Time
P1	~~7~~ ~~4~~ 1
P2	~~3~~ 0
P3	~~12~~ ~~9~~ 6
P4	~~5~~ 2
P5	~~9~~ 6

Gantt Chart



at time 21, process P4 begins to run

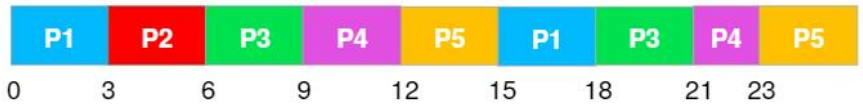
RR Scheduling

RR Scheduling

Process	Burst Time
P1	~~7~~ ~~4~~ 1
P2	~~3~~ 0
P3	~~12~~ ~~9~~ 6
P4	~~5~~ ~~2~~ 0
P5	~~9~~ 6

process P4 only runs for the duration of its CPU burst

Gantt Chart



at time 23, process P5 runs

RR Scheduling

RR Scheduling

Process	Burst Time
P1	~~7~~ ~~4~~ 1
P2	~~3~~ 0
P3	~~12~~ 9 6
P4	~~5~~ ~~2~~ 0
P5	9 6 3

Gantt Chart

at time 26, process P1 runs

RR Scheduling

RR Scheduling

Process	Burst Time
P1	~~7~~ ~~4~~ ~~1~~ 0
P2	~~3~~ 0
P3	~~12~~ ~~9~~ 6
P4	~~5~~ ~~2~~ 0
P5	~~9~~ ~~6~~ 3

process P1 only runs for one time unit

Gantt Chart

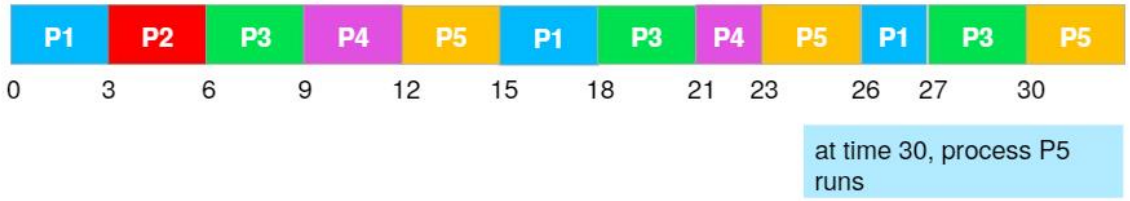
at time 27, process P3 runs

RR Scheduling

RR Scheduling

Process	Burst Time
P1	~~7~~ ~~4~~ ~~1~~ 0
P2	~~3~~ 0
P3	~~12~~ ~~9~~ ~~6~~ 3
P4	~~5~~ ~~2~~ 0
P5	~~9~~ ~~6~~ 3

Gantt Chart

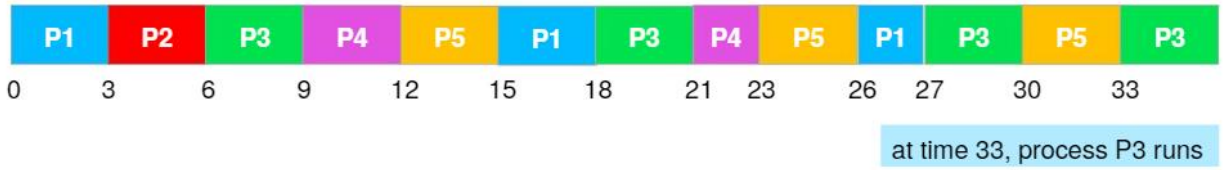


RR Scheduling

RR Scheduling

Process	Burst Time
P1	~~7~~ ~~4~~ ~~1~~ 0
P2	~~3~~ 0
P3	~~12~~ ~~9~~ ~~6~~ 3
P4	~~5~~ ~~2~~ 0
P5	~~9~~ ~~6~~ ~~3~~ 0

Gantt Chart



RR Scheduling

RR Scheduling

Process	Burst Time
P1	~~7~~ ~~4~~ ~~1~~ 0
P2	~~3~~ 0
P3	~~12~~ ~~9~~ ~~6~~ ~~3~~ 0
P4	~~5~~ ~~2~~ 0
P5	~~9~~ ~~6~~ ~~3~~ 0

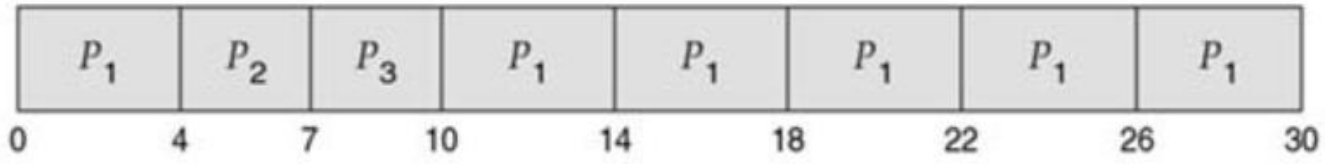
Gantt Chart

process P3 finishes at time 36

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

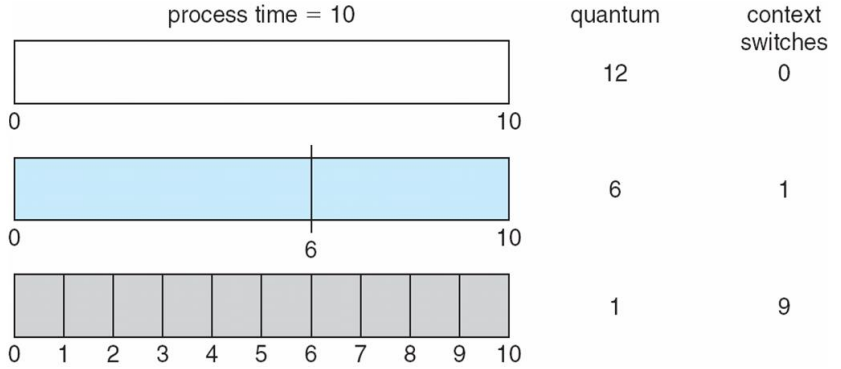
- The Gantt chart is:



- Average waiting time: [(10 - 4) + 4 + 7] / 3 = 5.66

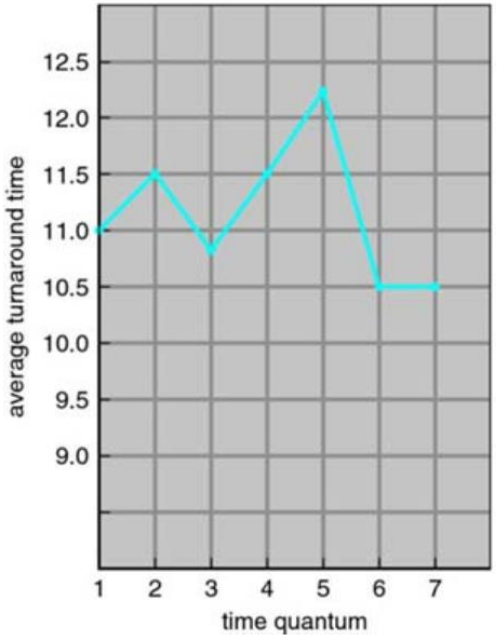
Time Quantum and Context Switch Time

- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Performance
  - *q* large ⇒ FCFS
  - *q* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high



Time Quantum and Context Switch Time

- Turnaround time also depends on the size of the time quantum.
  - Improved if most processes finish their next CPU burst in a time quantum.
  - Typically, higher average turnaround than SJF, but better ***response***
  - q should be large compared to context switch time
  - q usually 10ms to 100ms, context switch < 10 usec



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time

Priority Scheduling

Priority Scheduling

Process
P1
P2
P3
P4
P5

priority scheduling for the five processes P1, P2, P3, P4, P5

Priority Scheduling

Priority Scheduling

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

associated with each process is a **burst time**

Priority Scheduling

Priority Scheduling

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

also associated with each process is a **priority** where a lower numeric value indicates a higher relative priority

Priority Scheduling

Priority Scheduling

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

Gantt Chart

we will use a **Gantt Chart** to demonstrate the order in which processes are selected for scheduling according to the priority scheduling algorithm

Priority Scheduling

Priority Scheduling

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

Gantt Chart

0

process P2 is selected first as it has the highest priority; P2 begins running at time 0

Priority Scheduling

Priority Scheduling

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

Gantt Chart

after process P2 finishes, process P5 is selected as it has the next highest priority; P5 begins running at time 1

Priority Scheduling

Priority Scheduling

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

Gantt Chart

after process P5 finishes, process P1 is selected as it has the next highest priority; P1 begins running at time 6

Priority Scheduling

Priority Scheduling

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

Gantt Chart

after process P1 finishes, process P3 is selected next and begins running at time 16.

Priority Scheduling

Priority Scheduling

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

Gantt Chart

as process P4 has the lowest relative priority, it is selected last and begins running at time 18

Priority Scheduling

Priority Scheduling

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

Gantt Chart

process P4 finishes at time 19

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	100	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

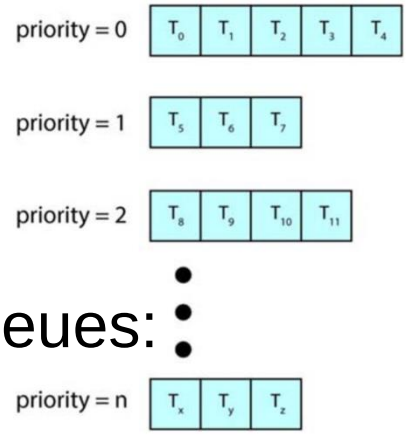
- Priority scheduling Gantt Chart

- Average waiting time (6 + 0 + 16 + 18 + 1) / 5 = 8.2 msec

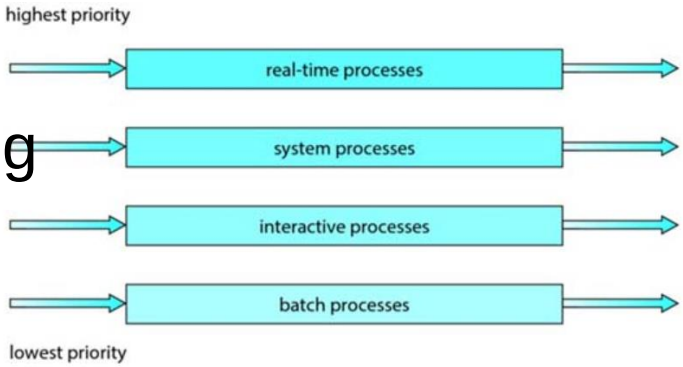
Priority Scheduling

- Priorities can be defined either internally or externally.
  - Internally by using some measurable quantities
    - E.g. time limits, memory requirements, #open files, ratio of average I/O burst to average CPU burst
  - Externally by using criteria outside OS
    - E.g. importance of process, fund paid for computer use, …
- Priority scheduling can be either preemptive or nonpreemptive.
- Problem ≡ Starvation (indefinite blocking) – low priority processes may never execute
- Solution ≡ Aging – as time progresses increase the priority of the process

Multilevel Queue



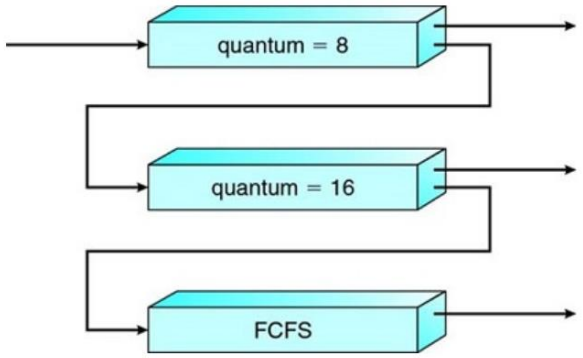
- Ready queue is partitioned into separate queues:
  - foreground (interactive)
  - background (batch)
- Process permanently in a queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS



- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR and 20% to background in FCFS

Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS



- Scheduling
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2

Multilevel Feedback Queue

- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Thread Scheduling

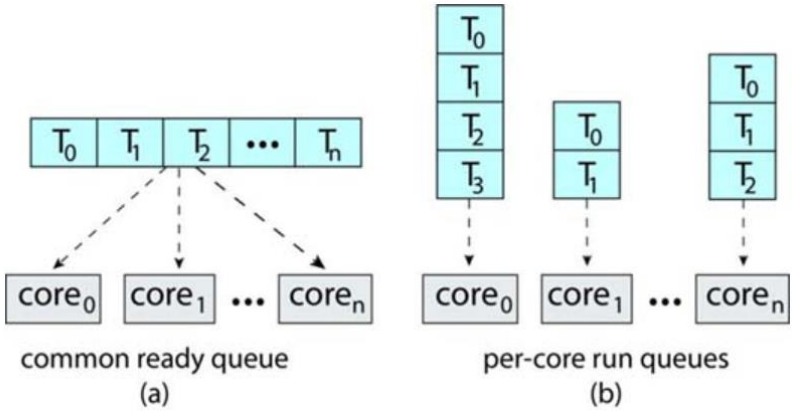
- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as process-contention scope (PCS) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is system-contention scope (SCS) – competition among all threads in system
  - One-to-one model only uses SCS such as Windows and Linux

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available.
- The term multiprocessor now refers to
  - Multicore CPUs
  - Multithreaded cores
  - NUMA systems

  } Homogeneous
  - Heterogeneous multiprocessing

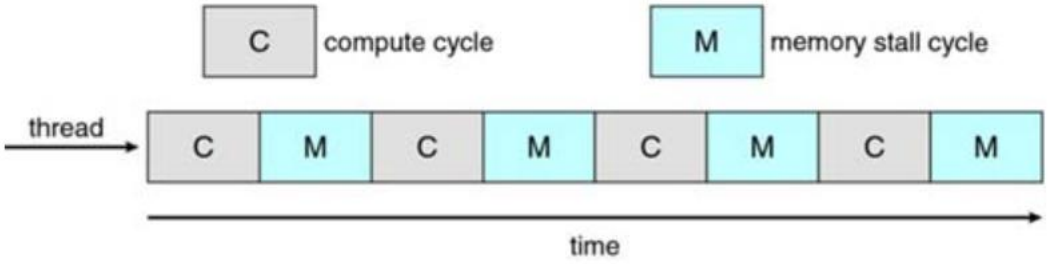
Approaches to Multipl Scheduling



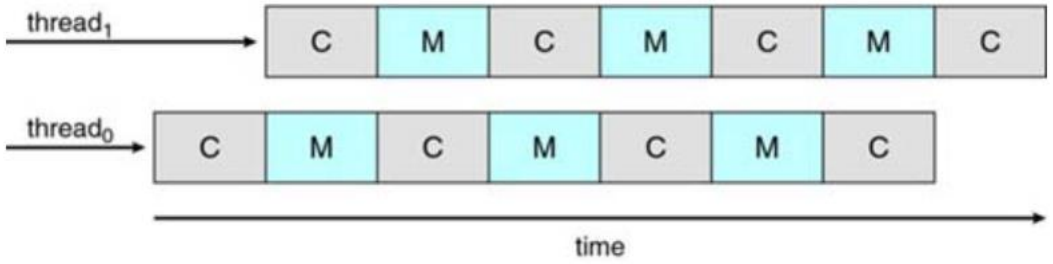
- Homogeneous processors within a multiprocessor
- Asymmetric multiprocessing – only one processor (master server) accesses the system data structures, alleviating the need for data sharing
  - Bottleneck at master server
- Symmetric multiprocessing (SMP) – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes with two possibilities
  - One common ready queue for all threads
    - Performance bottleneck on accessing the shared queue
  - Own private queue of threads per processor
    - Performance bottleneck on workloads

Multicore Processors

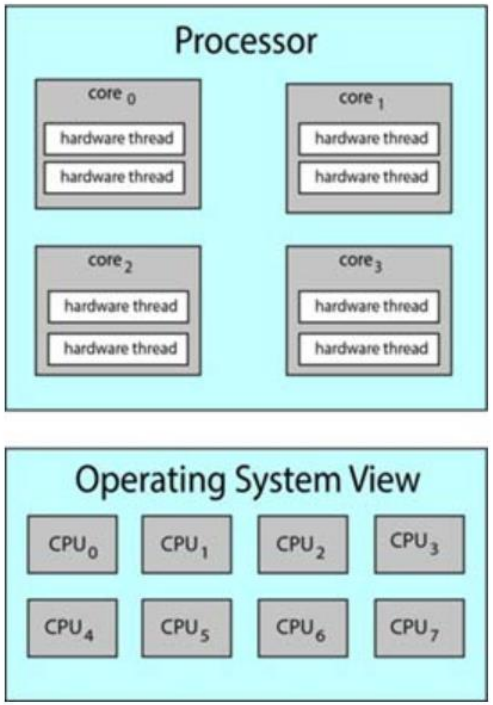
- Contemporary computer hardware now place multiple processor cores on same physical chip
  - Each core as a separate logical CPU
  - Faster and consumes less power
- Complicate scheduling issues
  - Memory stall – processor spends a significant amount of time waiting for the data to become available.
    - Modern processors are faster than memory
    - Cache miss



Multicore Proce

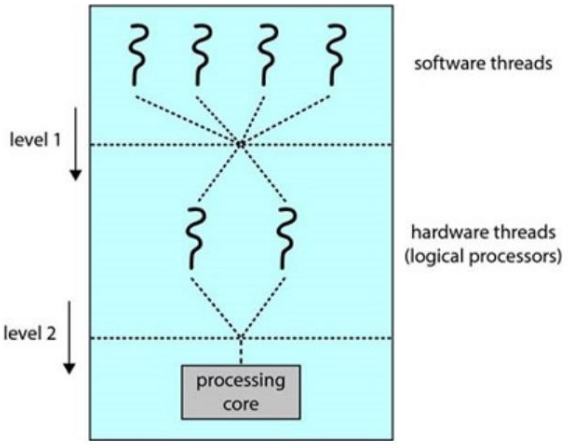


- Many recent hardware designs have implemented multithreaded processing cores in which two (or more) hardware threads are assigned to each core.
  - Interleaving threads to improve CPU utilization
- Chip multithreading (CMT)- multiple hardware threads in a processing core
  - Each hardware thread as a logical CPU from an operating system perspective.
  - Intel use the term hyper-threading
  - A.k.a. simultaneous multithreading
- Two ways to multithread
  - Coarse-grained
    - Threads switching when a memory stall occurs.
    - High cost of switching for flushing pipeline
  - Fine-grained
    - Switching at a finer level of granularity
    - Low cost of switching but include in the architectural design



Multicore Processors

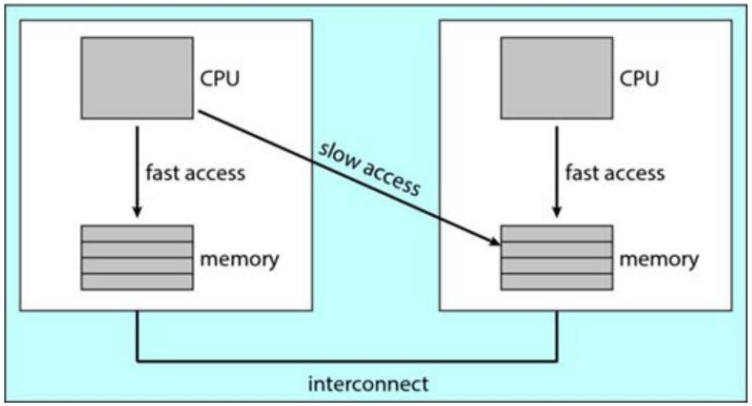
- Resources must be shared among its hardware threads: a core processes one thread at a time
  - E.g. caches, pipelines
- Two levels of scheduling
  - Level 1: OS select which software threads to run on which hardware threads (logical CPUs)
    - Use scheduling algorithm previously described
  - Level 2: Processing core selects which hardware threads to run.
    - Round-robin by UltraSPARC T3
    - Urgency value (from 0 to 7) by Intel Itanium
  - Two levels are **not** necessarily mutually exclusive.
    - If OS can aware the level of processor resource sharing, it can schedule software threads to different logical processors.



Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- Load balancing attempts to keep workload evenly distributed
  - Necessary only on systems where each processor has its own private ready queue
  - What should be balanced?
- Two approaches
  - Push migration – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
  - Pull migration – idle processors pulls waiting task from busy processor
  - Often in parallel use of the two approaches

Processor Affinity



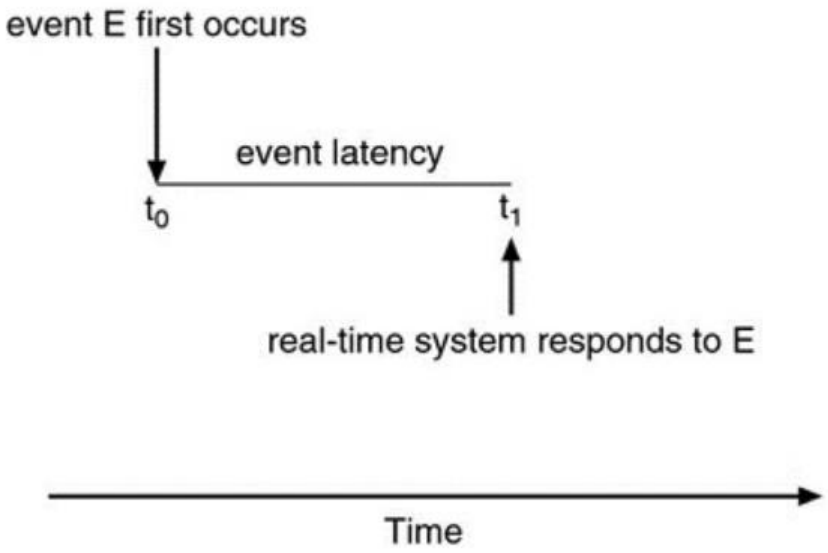
- Processor affinity – process has affinity for processor on which it is currently running
  - Per-processor ready queues provide processor affinity.
- Two forms of processor affinity
  - Soft affinity
  - Hard affinity
  - May be affected by main-memory architecture: Faster access if CPU scheduler and memory-placement algorithms are ***NUMA-aware***
- Load balancing often counteracts the benefits of processor affinity.
  - Scheduling becomes more complex.

Heterogeneous Multiprocessing

- Although mobile systems now include multicore architectures, some systems are now designed using cores that run the same instruction set, yet vary in terms of their clock speed and power management. This is known as heterogeneous multiprocessing (HMP).
  - For better power consumption
- The big.LITTLE architecture for ARM processors
  - Higher-performance ***big*** cores
  - Energy efficient ***LITTLE*** cores

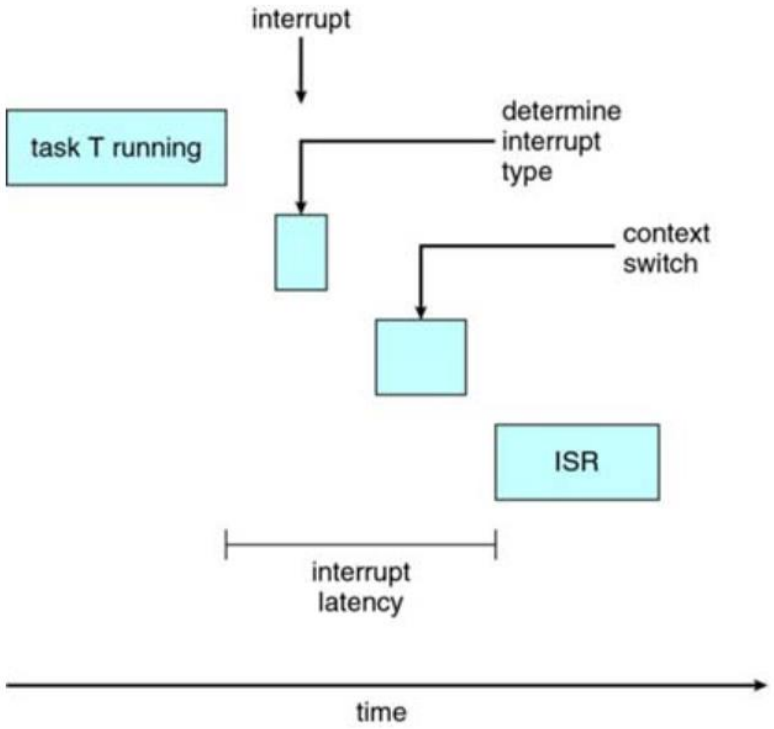
Real-Time CPU Scheduling

- Can present obvious challenges
- Soft real-time systems – no guarantee as to when critical real-time process will be scheduled
- Hard real-time systems – task must be serviced by its deadline



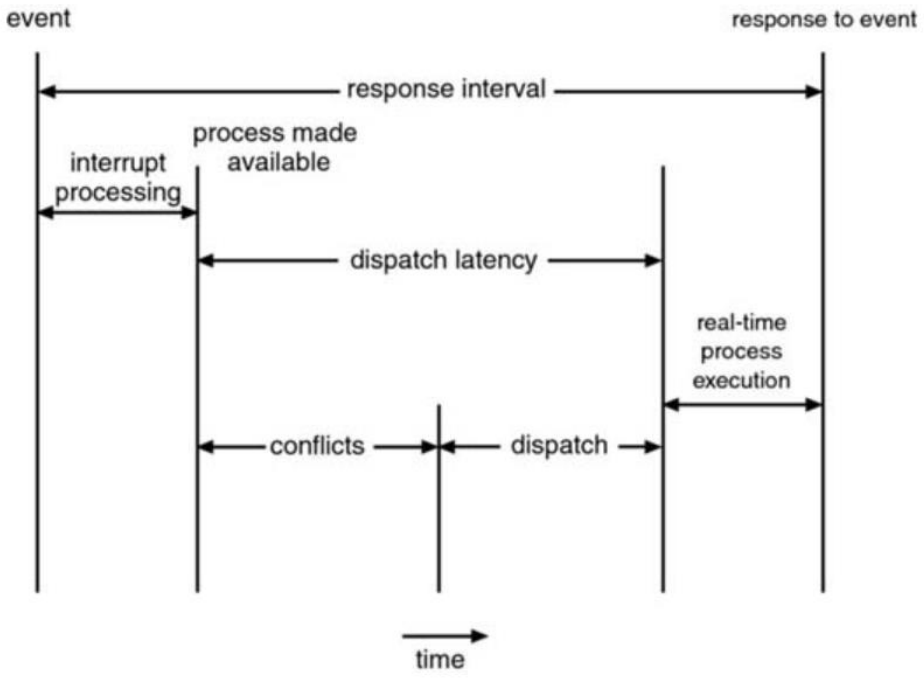
Real-Time CPU Scheduling

- Interrupt latency – time from arrival of interrupt to start of routine that services interrupt

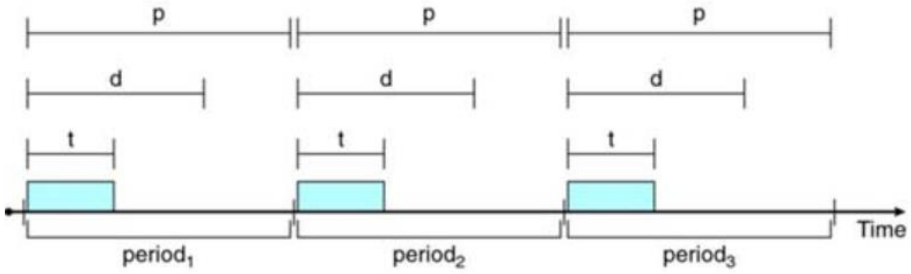


Real-Time CPU Scheduling (Cont.)

- Dispatch latency – time for schedule to take current process off CPU and switch to another
  - Conflict phase
    - Preemption of any process running in kernel mode
    - Release by low-priority process of resources needed by high-priority processes
  - Dispatch phase
    - Schedule the high-priority process onto an available CPU



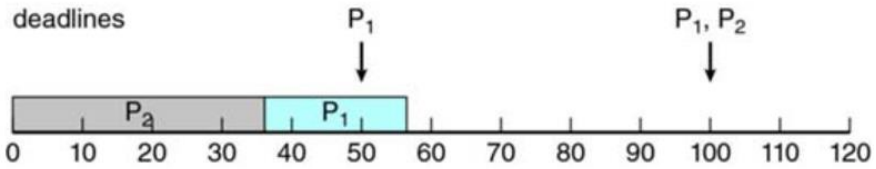
Priority-based Sche



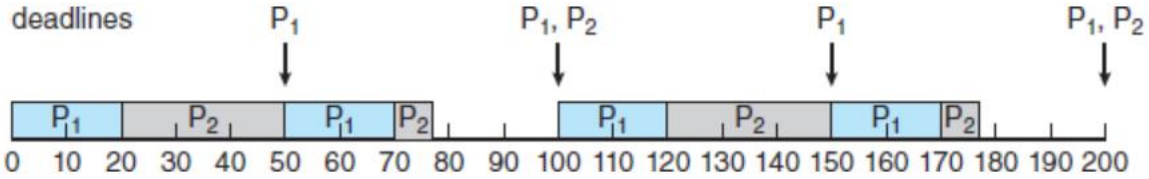
- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: periodic ones require CPU at constant intervals
  - Has processing time t, deadline d, period p
  - $0 \leq t \leq d \leq p$
  - Rate of periodic task is 1/p
- A process has to announce its deadline
  - Can be done by admission-control algorithm: Admit if possible or reject if it cannot guarantee the deadline.

Rate-Monotonic Scheduling

- A priority is assigned based on the inverse of its period
  - Shorter periods = higher priority;
  - Longer periods = lower priority
- Consider two processes P_1 and P_2
  - Periods: $P_1 = 50$ and $P_2 = 100$
  - Processing time: $t_1 = 20$ and $t_2 = 35$
  - CPU utilization: 20 / 50 +35 / 100 = 75%
- Consider the rate-monotonic scheduling
  - If P_2 is assigned a higher priority than P_1.

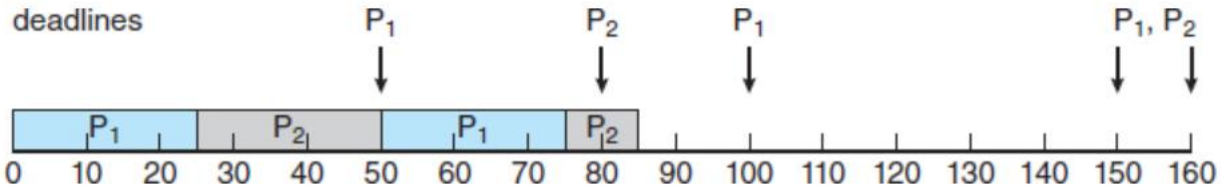


  - If P_1 is assigned a higher priority than P_2



Rate-Monotonic Scheduling

- Rate-monotonic scheduling is considered optimal
  - If a set of processes cannot be scheduled by this algorithm, it cannot be scheduled by any other algorithm that assigns static priorities.
- Consider another example
  - Periods: $P_1 = 50$ and $P_2 = 80$
  - Processing time: $t_1 = 25$ and $t_2 = 35$
  - CPU utilization: 20 / 50 +35 / 80 = 94%
  - P_1 has higher priority than P_2.

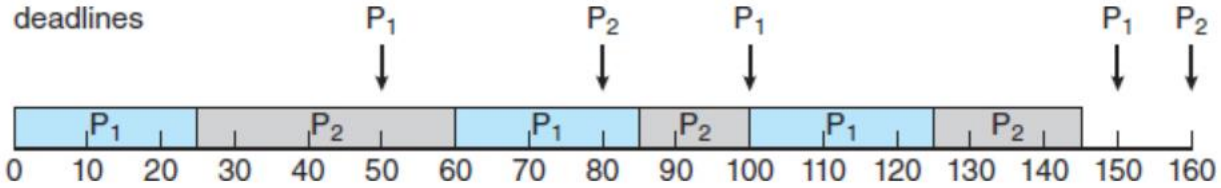


- A limitation is CPU utilization is bounded. For N processes, the worst-case CPU utilization is:

$$N(2^{1/N} - 1)$$

Earliest-Deadline-First (EDF) Scheduling

- Priorities are assigned according to deadlines:
  - the earlier the deadline, the higher the priority
  - the later the deadline, the lower the priority
- Consider previous example
  - Periods: $P_1 = 50$ and $P_2 = 80$
  - Processing time: $t_1 = 25$ and $t_2 = 35$
  - CPU utilization: 20 / 50 +35 / 80 = 94%
  - P_1 has higher priority than P_2.



- EDF is theoretically optimal.
  - 100% CPU utilization which meats all deadlines
  - Impossible due to context switching and interrupt handling

Proportional Share Scheduling

- T shares are allocated among all processes in the system
- An application receives N shares where $N < T$
- This ensures each application will receive N/T of the total processor time
- For example, $T = 100$ shares for processes A=50, B=15, and C=20 shares.
  - The processor time for A, B, and C is 50%, 15%, and 20%, respectively.
- Must work with admission-control policy to meet dealines.

Algorithm Evaluation

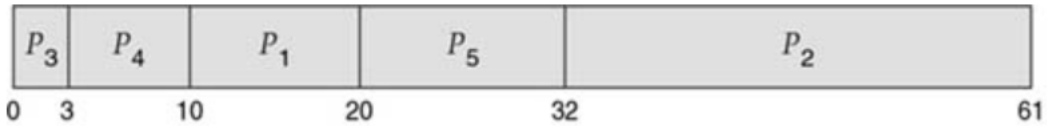
- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms, e.g.,
  - Maximizing CPU utilization under the constraint that the maximum response time is 300 milliseconds
  - Maximizing throughput such that turnaround time is (on average) linearly proportional to total execution time
- Deterministic modeling
  - Type of analytic evaluation
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload

Algorithm Evaluation

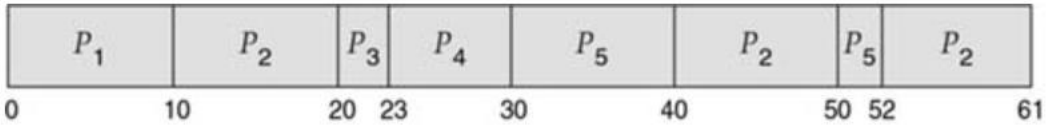
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

- Consider 5 processes arriving at time 0
- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCFS is 28ms: (0 + 10 + 39 + 42 + 49) / 5 = 28

  - Non-preemptive SFJ is 13ms: (10 + 32 + 0 + 3 + 20)/5 = 13



  - RR is 23ms: (0 + 32 + 20 + 23 + 40)/5 = 23



Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc
  - This area of study is called queueing-network analysis.

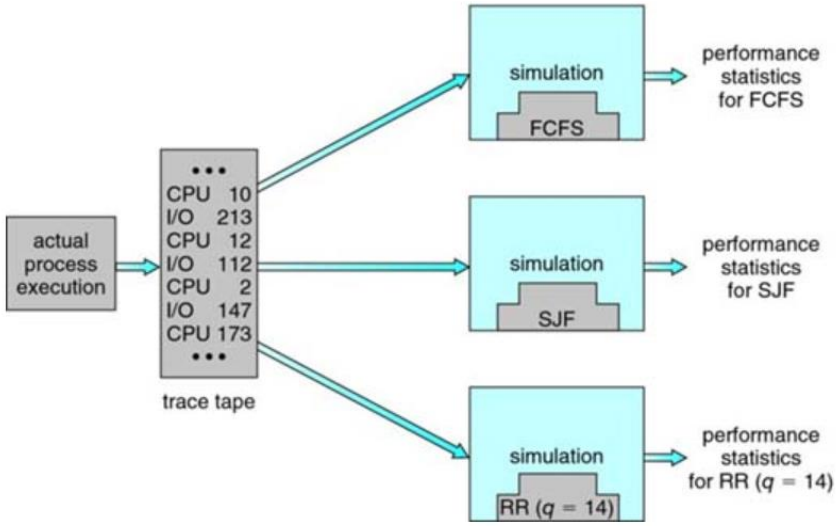
Little's Formula

- n = average queue length
- W = average waiting time in queue
- λ = average arrival rate into queue
- Little's law – in steady state, processes leaving queue must equal processes arriving, thus:

  $n = \lambda \times W$

  - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds

Simulations



- Queueing models limited
- Simulations more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to probabilities
    - Distributions defined mathematically or empirically
    - Trace tape (files) record sequences of real events in real systems

Implementation

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again environments vary