Chapter 5: CPU Scheduling

1

Chapter Topics

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

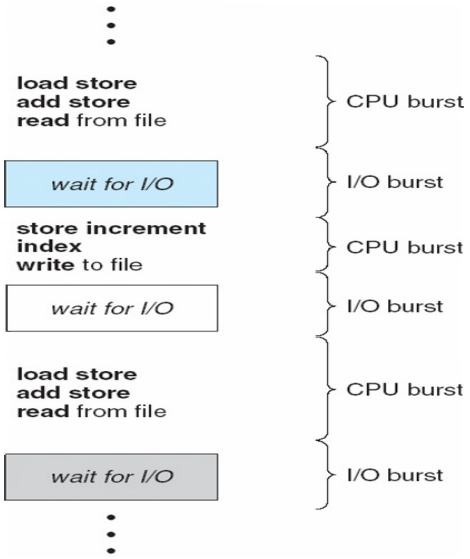
2

Multiprogramming

- The objective of multiprogramming is to have some process running all times to maximize the CPU utilization.
  - When the currently running process is waiting for certain event happen (e.g., a disk I/O), another process is selected from the ready queue and dispatched to the CPU.

3

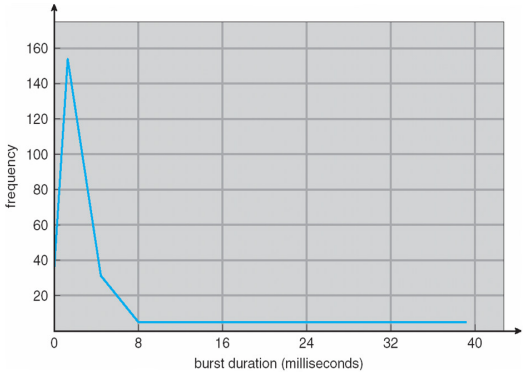
Alternating Sequence of CPU and I/O Bursts of Each Process



4

Durations of CPU Burst

- The distribution of CPU burst times has a kind of exponential distribution, with a large number of short CPU bursts and a small number of long CPU bursts.
- An I/O-bound process typically has many short CPU bursts.



Histogram of CPU burst durations

5

CPU Scheduler

- The CPU scheduler (or short-term scheduler) selects a process from the processes in memory that are ready to execute (i.e., that are in the ready queue), and allocates the CPU to the process.
- A ready queue can be implemented as a FIFO queue, a priority queue, an ordered linked list, etc.
  - The records in the queues are usually process control blocks (PCBs) of the processes.

6

Preemptive and Nonpreemptive Scheduling

- CPU scheduling decisions may take place when a process
  1. switches from the running state to the waiting state (e.g., as a result of an I/O request or an invocation of **`wait()`** to wait the termination of its child process)
  2. switches from running state to the ready state (e.g., when an interrupt occurs)
  3. switches from the waiting state to the ready state (e.g., at the completion of an I/O)
  4. terminates
- If scheduling is performed only under circumstances 1 and 4, it is called nonpreemptive scheduling; otherwise, it is called preemptive scheduling.
  - In cases 1 and 4, the CPU is not preempted from the current running process, instead it is released by the process voluntarily.

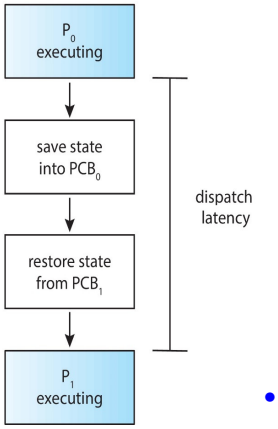
7

Preemptive and Nonpreemptive Scheduling (cont'd)

- Under nonpreemptive scheduling, once the CPU is allocated to a process, the process keeps the CPU until it voluntarily releases the CPU either by terminating or by switching to the waiting state.
- A preemptive scheduler can also select a new process to run when a process is added to the ready queue, in addition to the cases when a process terminates or changes its state from the running state to the waiting state.

8

Dispatcher



- The dispatcher module gives control of the CPU to the process selected by the CPU scheduler. The dispatching involves:
  - context switching
  - switching to user mode (to start executing the selected user process)
  - jumping to the proper location in the selected user process to (re)start that program
- **Dispatch latency:** time it takes for the dispatcher to stop one process and start running another.

9

Scheduling Criteria

- **CPU utilization:** we want to keep the CPU as busy as possible.
- **Throughput**: the number of processes that complete their execution per unit time.
- **Turnaround time:** the interval from the submission of a process to the time of completion.
- **Waiting time:** the sum of the time a process has been waiting in the ready queue.
- **Response time:** the interval from the submission of a request until the first response is produced (i.e., the time it takes to start responding), which is important for interactive applications.
  - For interactive systems, it may be more important to minimize the variance in the response time than to minimize the average response time.

10

Scheduling Algorithms

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

11

First-Come, First-Served (FCFS) Scheduling

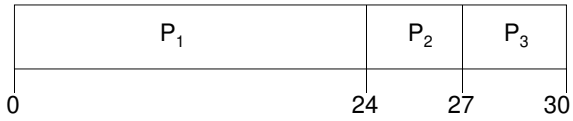
- The process that requests the CPU first is allocated the CPU first.
- FCFS scheduling can be easily implemented using a FIFO queue:
  - When a process enters the ready queue, its PCB will be linked to the tail of the queue.
  - When the CPU is free, it is allocated to the process at the head of the queue.
- The average waiting time of a process is often quite long.

12

FCFS Scheduling Example

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; for P_2 = 24; for P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17
- There is a convoy effect as the short processes wait for one big process to finish.

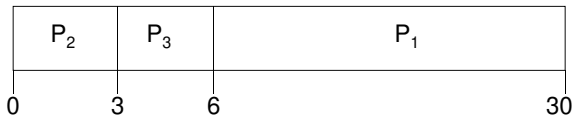
13

FCFS Scheduling Example (cont'd)

- 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; for P_2 = 0; for P_3 = 3
- Average waiting time: (6 + 0 + 3)/3 = 3
  - Much smaller than the previous case.

14

Shortest-Job-First (SJF) Scheduling

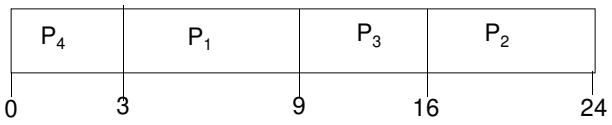
- This algorithm associates with each process the length of its next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst.
- SJF may give the minimum average waiting time for a given set of processes.
  - The real difficulty is knowing the length of the next CPU burst of each process.

15

SJF Scheduling Example

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

16

Predicting the Length of Next CPU Burst

- We can predict the length of next CPU burst of a process.
- The next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts:
  - Let t_n be the length of the nth CPU burst.
  - Let τ_{n+1} be the predicted value for the $(n+1)$th CPU burst. Then, for α, $0 \leq \alpha \leq 1$, τ_{n+1} can be defined as:

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

17

Examples of Exponential Averaging

- If $\alpha = 0$

  $\tau_{n+1} = \tau_n$

  Most recent information (i.e., t_n) has no effect.
- If $\alpha = 1$

  $\tau_{n+1} = t_n$

  Past history (represented by τ_n) is not counted.
- If we expand the formula for τ_{n+1} by substituting for τ_n, τ_{n-1}, …, τ_1, we get:

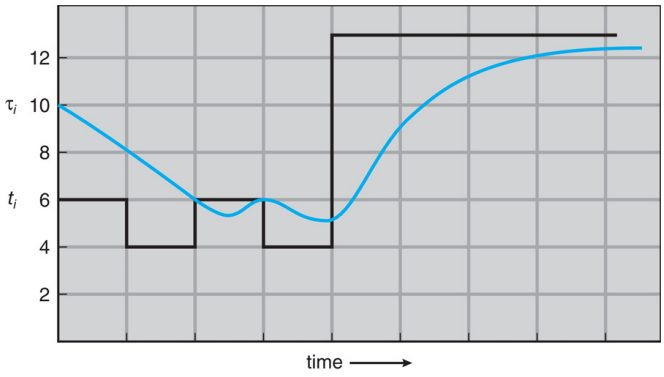
$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + (1 - \alpha)^2 \alpha\, t_{n-2} + \ldots$$
$$+ (1 - \alpha)^j \alpha\, t_{n-j} + \ldots$$
$$+ (1 - \alpha)^{n-1} \alpha\, t_1 + (1 - \alpha)^{n+1} \tau_0$$

  - Since both α and $(1 - \alpha)$ are less than 1, each successive term has less weight than its predecessor.
  - **The initial value τ_0 can be defined as a constant or as an overall system average.**

18

Example of Predicting the Length of the Next CPU Burst

$\alpha = 0.5$,
$\tau_1 = (1 - \alpha)\ \tau_0$
$= 10$



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

19

Priority Scheduling

- A priority number (an integer value) is associated with each process.
  - The smallest priority number represents the highest priority.
- The CPU is allocated to the process with the highest priority.
  - Priority Scheduling can be either preemptive or nonpreemptive.
- SJF is a priority scheduling where priority of a process is the inverse of its next CPU burst time predicted.
- The main problem is the potential starvation (i.e., indefinite waiting) of low priority processes due to a steady stream of high priority processes.
  - **Aging** is a technique of gradually increasing the priority of processes that have been waiting in the system for a long time.

20

Priority Scheduling (cont'd)

- Priorities can be defined either internally or externally:
  - Internally defined priorities use some measurable quantity (or quantities) to compute the priority; such as time limits (i.e., deadlines), memory requirement, the number of open files, and the ratio between average I/O burst to average CPU burst, etc.
  - External priorities are set by criteria outside the OS, such as the importance of the process, type and amount of the funding paid for the use of computer, the department sponsoring the work, etc.

21

Round-Robin (RR) Scheduling

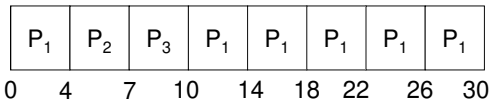
- The round-robin scheduling algorithm is designed for time-sharing systems.
  - It is similar to the FCFS scheduling, but CPU preemption is added to limit the time that a process can use the CPU continuously.
- Each process gets a small unit of CPU time, called (CPU) ***time quantum*** or (CPU) ***time slice***, which is usually set to 10–100 msec. 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 time units, then no process waits more than $(n–1)q$ time units until it gets the CPU.

22

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

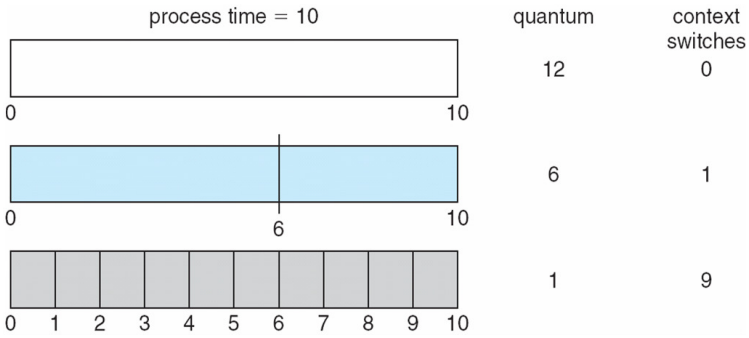


- Typically, RR results in a longer average turnaround time than SJF, but a shorter average response time.
  - If a process's CPU burst is longer than a time quantum, it should wait longer to get multiple time quantums.

23

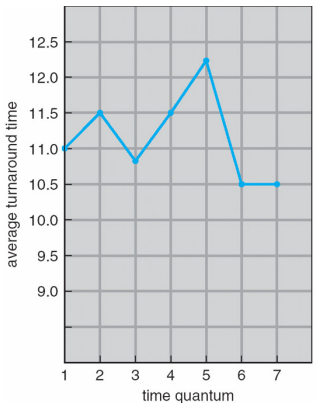
Time Quantum and Context Switch Time

- At each CPU time quantum, context switching occurs. Thus, the time quantum must be large with respect to the context switch time, otherwise the total context switching overhead is too high.



24

Turnaround Time Varies with the Time Quantum



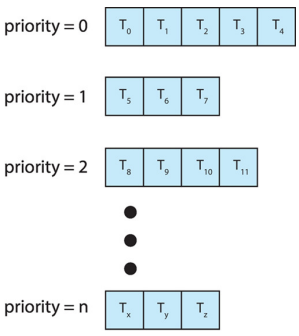
process	time
P_1	6
P_2	3
P_3	1
P_4	7

- The average turnaround time can be reduced if most processes finish their next CPU burst in a single time quantum.
  - A rule of thumb is that the time quantum should be longer than 80% of the CPU bursts.

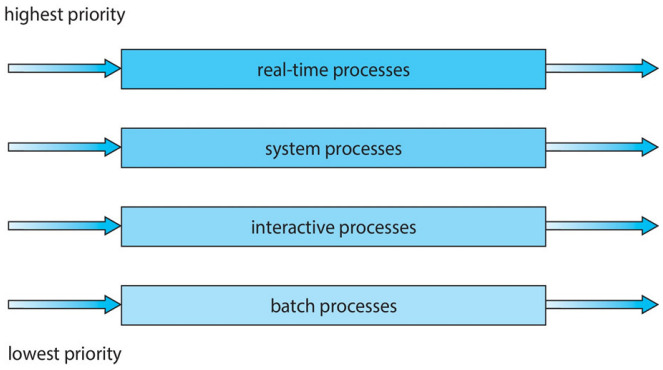
Multilevel Queue Scheduling

- The ready queue is partitioned into separate queues, and each queue has its own scheduling algorithm (e.g., round-robin, FCFS).
- There must be scheduling among the queues.

1. Fixed-priority preemptive scheduling: serve all the processes in the first-level (i.e., highest priority) queue then serve the processes in the second-level queue, and so on.
   - If we have a new process placed in a higher-level queue, while the CPU is executing a process from a lower-level queue, then the CPU is preempted from the lower level queue.
2. Time-slicing among the queues: each queue gets a certain amount (i.e., percentage) of CPU time, which it can then schedule among its processes.



Example of Multilevel Queue Scheduling



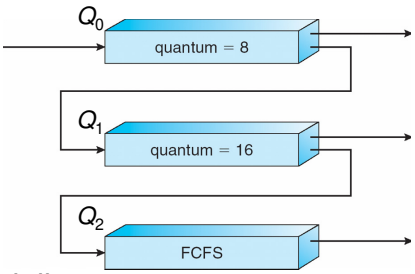
27

Multilevel Feedback Queue Scheduling

- Unlike the case of the multilevel queue scheduling, a process can move between queues.
  - The *aging* can be implemented this way, by moving up a process from a lower-priority queue to a higher-priority queue if it has been waiting in the lower-priority queue for a long time.
- A multilevel-feedback-queue scheduler is defined by the following parameters:
  - The number of queues.
  - The scheduling algorithm for each queue.
  - The method used to determine when to upgrade a process to a higher-priority queue.
  - The method used to determine when to demote a process to a lower-priority queue.
  - The method used to determine which queue a process will enter initially.

28

Example of Multilevel Feedback Queue



- Three queues:
  - Q_0: RR with time quantum 8 msec per process.
  - Q_1: RR with time quantum 16 msec per process.
  - Q_2: FCFS
- Fixed-priority preemptive Scheduling
  - A process entering the ready queue is put in queue Q_0.
  - When a process in Q_0 does not finish within the time quantum of 8 msec, it is preempted and moved to the tail of queue Q_1.
  - If Q_0 is empty, the process at the head of Q_1 is given a time quantum of 16 msec. If it does not complete, it is preempted and moved to queue Q_2.
  - Only when both Q_0 and Q_1 are empty, the processes in Q_2 are scheduled based on FCFS.

29

Thread Scheduling

- In many-to-one and many-to-many thread (mapping) models, the user-level scheduler (aka process scheduler) of the process schedules its threads to an available Light Weight Processor (LWP), which is associated with a kernel thread.
  - This scheme is known as **process-contention scope (PCS)** since the competition for the LWP is among the user threads belong to the same process.
  - Typically, PCS is done according to priorities of the user threads which are set by the programmer and are not adjusted by the user-level thread library.
- Kernel threads are scheduled by the kernel scheduler (i.e., CPU scheduler) based on **system-contention scope (SCS)** since the competition for a CPU is among all kernel threads in the system.
  - Systems using one-to-one thread (mapping) model schedule threads using only SCS.

30

Multiprocessor Scheduling

- CPU scheduling is more complex when multiple processors are available.
- In the **Asymmetric Multiprocessing** system, the master processor schedules and allocate work to the slave processors.
  - A master processor controls the system; and the slave processors either look to the master for instruction or have predefined tasks.
- In the **Symmetric Multiprocessing (SMP)** systems, each processor performs all the tasks within the OS.
  - All the processors are peers, and no master-slave relationship exists between them.
- **In the SMP systems, each processor is self-scheduling.**
  - All processes may be in a common ready queue, or each processor has its own private ready queue.
  - **The scheduler for each processor examines the ready queue (the common one or its private one) and selects a process to execute.**

31

Load Balancing

- Load balancing attempts to keep the workload evenly distributed across all processors in the SMP system.
  - **Load balancing is typically necessary on systems where each processor has its own private ready queue.**
  - On systems with a common ready queue, load balancing is often unnecessary, because whenever a processor becomes idle, it immediately takes a runnable (i.e., ready) process from the common ready queue.
- Two general approaches for loading balancing:
  - **Push migration:** A specific load-balancing task periodically checks the load on each processor, and if it finds an imbalance, evenly distributes the load by moving (or pushing) processes from overloaded processors to less-busy processors.
  - **Pull migration:** An idle processor pulls a process from the private ready queue of a busy processor to execute it.
- Linux implements both push and pull migrations:
  - For example, OS runs its load-balancing algorithm every 200 msec (push migration) or whenever the ready queue for a processor is empty (pull migration).

32

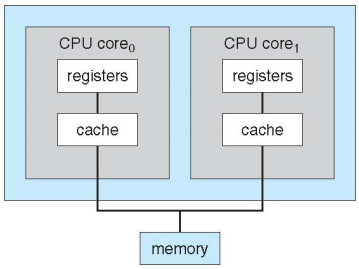
Processor Affinity

- In a multiprocessor system, if a process that has been running on one processor is migrated to another processor,
  1. the virtual blocks (i.e., instruction blocks and data blocks) of the process in the cache memory of the first processor must be invalidated, and
  2. the virtual blocks of the process should be fetched into the cache memory of the second processor .
- If a process is allowed to be running on the same processor, instead of being migrated to another processor, then the process has an affinity for the processor.
  - **Soft affinity:** OS attempts to keep a process on the same processor, but there is a chance that the process could be migrated to another processor.
  - **Hard affinity:** a process is allowed to specify (by using a system call) that it is not migrated to another processor, and OS supports it.

33

Multicore Processors

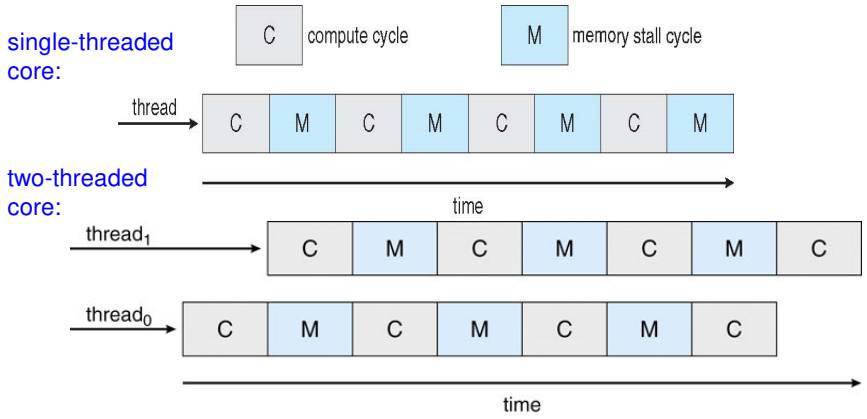
Dual-core Processor



- Multicore processor: multiple *processor cores* are placed on the same processor chip.
- Multi-threaded core: multiple hardware threads are available in a processor core, and they appear as logical processors to OS.
  - Each hardware thread has a register set used to run a thread.
  - The *UltraSPARC T1* CPU has eight cores per chip and four hardware threads per core, so there appear to be 32 logical processors.

34

Multithreaded Multicore System



- If a cache miss occurs (as the referenced word is not in the cache), we have a memory stall cycle which is the waiting time for the requested block (containing the referenced word) to be fetched from main memory into the cache memory. The execution of two threads can be interleaved within a two-threaded core.

35

Overview of Real-time Systems

- A real-time system requires that results of certain processes be produced within a specified deadline period. Those processes are called real-time processes.
- An embedded system is a computing device that is part of a larger system (e.g., home appliances, consumer digital devices, automobile, airplanes).
- A safety-critical system is a real-time system with catastrophic results in case of a missed deadline (e.g., weapon systems, antilock brake systems, flight-control systems, health-care systems).
- A hard real-time system must guarantee that real-time tasks are completed within their required deadlines.
- A soft real-time system assigns higher scheduling priority to real-time tasks than other tasks, but cannot guarantee their deadlines would be satisfied.

36

Real-time System Characteristics

- **Single purpose**: a real-time system typically serves only a single purpose, such as controlling antilock brakes or delivering music on an MP3 player.
- **Small size**: many real-time systems exist in a limited physical space. As a result, most real-time systems lack both the CPU processing power and the amount of memory available in PCs.
- **Inexpensively mass-produced**: many real-time systems are used in home appliances and consumer devices, so they need to be inexpensively mass-produced.
- **Specific timing requirements**: real-time OSs meet timing requirements by giving the highest priorities to real-time processes.

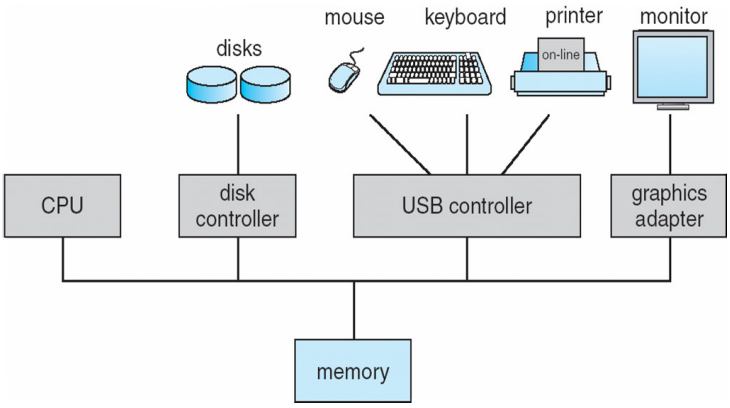
37

System-on-a-Chip

- Many real-time systems are designed using **system-on-a-chip (SOC)** strategy.
- SOC allows the CPU, memory, and attached peripheral ports (e.g., USB ports) to be contained in a single integrated circuit.
- A SOC is typically less expensive than the bus-oriented architecture.

38

Bus-Oriented System



39

Features of Real-time Kernels

- Most real-time systems do not provide some of the features common in a standard desktop system:
  - A variety of peripheral devices, such as graphical displays, CD/DVD drives, etc.
  - Protection and security mechanisms.
  - Supporting multiple users.
- Reasons include
  - Real-time systems are typically single-purpose.
  - Real-time systems often do not require interfacing with a user.
  - Many real-time systems lack sufficient space to support peripheral devices.
  - Supporting the features common in standard PCs would increase the cost of real-time systems.

40

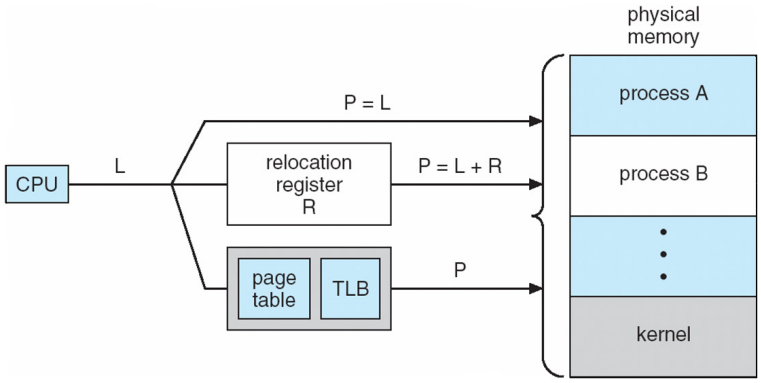
Address Translation in Real-time Systems

- Three different strategies for managing address translation for real-time systems:

1. **Real-addressing mode**: bypass logical addresses and have the CPU generate physical addresses (i.e., main memory addresses) directly.
   - The system is fast as no time is spent on address translation.
   - Relocation of the program in the memory is difficult.
   - Lack of memory protection between processes.
2. **Relocation register mode**: a relocation register is set to the memory location where a program is loaded.
   - Physical address P is generated by adding the content of the relocation register R to the logical address L (i.e., P = L + R).
   - Lack of memory protection between processes.
3. **Full virtual memory functionality**: uses page tables and a Translation Look-aside Buffer (TLB) for address mapping (from logical address to physical address).

41

Address Translation in Real-time Systems



L: logical address
P: physical address in memory
R: relocation register

42

Implementing Real-time Operating Systems

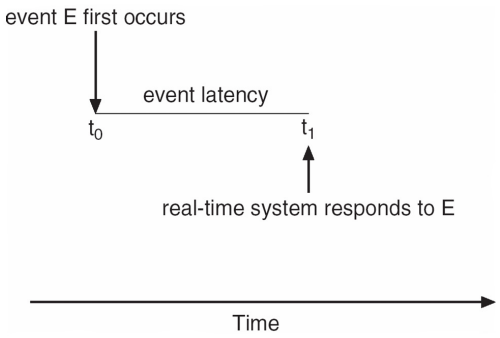
In general, real-time operating systems must provide:

1. Preemptive, priority-based scheduling:
   - Real-time processes are assigned higher priorities.
   - A process currently running on the CPU is preempted if a higher-priority process becomes available to run.
2. Preemptive kernel
   - A preemptive kernel allows the preemption of a process running in kernel mode. Otherwise, a real-time process may have to wait an arbitrary long period while another process is active in the kernel.
3. Minimizing Latency
   - Two types of latencies affect the performance of real-time systems: **interrupt latency** and **dispatch latency**.

43

Minimizing Latency

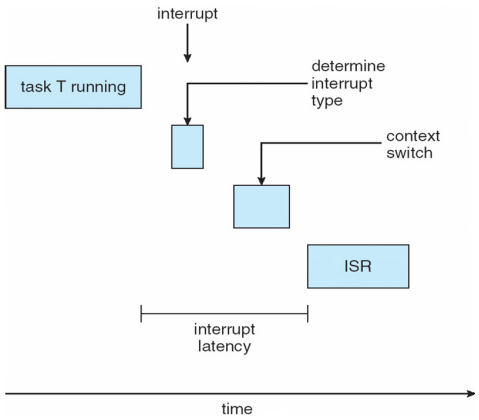
- When an event occurs, the system must respond to and service it as quickly as possible.
- **Event latency** is the amount of time from when an event occurs to when it is serviced.



44

Interrupt Latency

- Interrupt latency is the period of time from when an interrupt arrives at the CPU to when it is serviced.

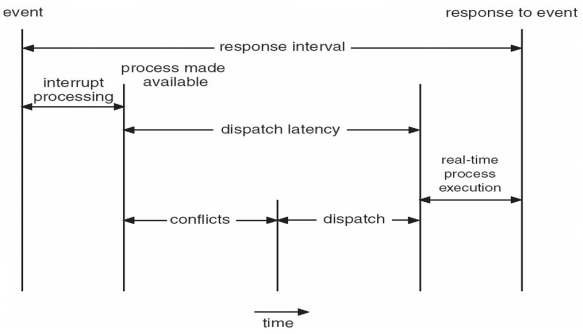


ISR: Interrupt Service Routine

45

Dispatch Latency

- Dispatch latency is the amount of time required for the scheduler to stop one process and start another.
  - Dispatching a real-time process can be triggered by a timer or a sensor input, via a corresponding interrupt.
- The conflict phase of dispatch latency has two components:
  - Preemption of any process running.
  - Release (by low-priority processes) of resources needed by a high-priority process.



46

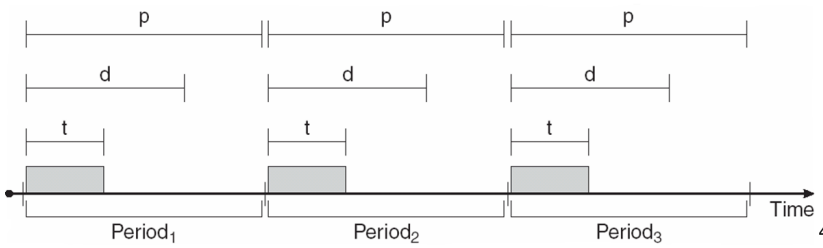
Priority-Inheritance Protocol

- Suppose that three processes *L*, *M*, and *H* have priorities in the order of *L* < *M* < *H*, and process *H* requires a resource *R* that is being locked and accessed by process *L*.
  - Ordinarily, process *H* will be blocked until process *L* finishes using resource *R* (e.g., kernel data) exclusively, and unlocks *R*.
  - However, if process *M* (that has nothing to do with *R*) becomes runnable, then it may preempt process *L*. As a result, process *M* can affect how long process *H* must wait to get the resource *R*.
  - This problem is known as **priority inversion**, and can be solved by the **Priority-Inheritance Protocol**:
- According to the priority-inheritance protocol, a process that is accessing resources needed by a higher-priority process inherits the higher priority until it is finished with the resources in question. When its use of the resources is done, its priority reverts to its original value.
  - In our example, process *L* temporarily inherits the priority of process *H* (as *H* is blocked on the locked resource *R*), thereby preventing process *M* from preempting the execution of *L*.

47

Real-time CPU Scheduling

- Periodic processes require the CPU at specified intervals (i.e., periods).
  - ***p*** is the duration of the period.
  - ***d*** is the deadline by when the process must be executed.
  - ***t*** is the processing time.



48

Real-time CPU Scheduling (cont'd)

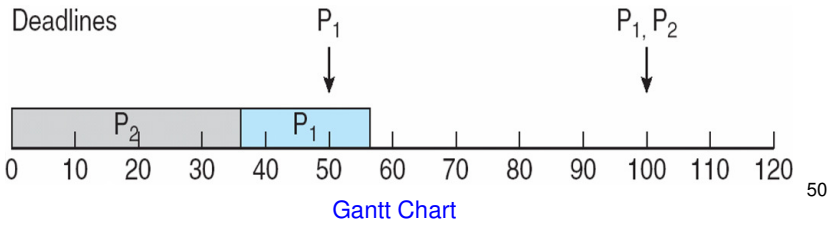
- In this form of scheduling, a process may have to announce its deadline requirements to the scheduler.
- Using an **admission control algorithm**, the scheduler either admits the process, guaranteeing that the process will complete on time, or rejects the request.
  - Admission control is okay for non-critical real-time processes.
  - e.g., play-back of a video can be rejected by an admission control algorithm (if the system is busy).

49

Scheduling of Tasks: Example

- We have two periodic processes P_1 and P_2.
- The periods of P_1 and P_2 are 50 and 100, respectively: p_1=50 and p_2=100.
- The processing times are: t_1=20 and t_2=35
- The deadline for each process requires that it completes its CPU burst (i.e., CPU phase) by the start of its next period.

If P_2 has a higher priority than P_1, such that P_2 starts First, then P_1 misses its 1st deadline, as shown below:



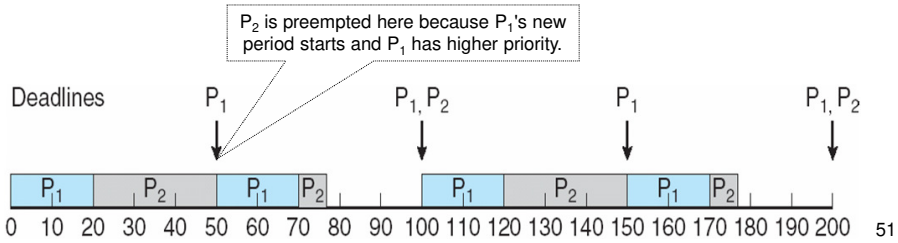
Gantt Chart

50

Rate Monotonic Scheduling (RMS)

- **A static priority is assigned to a process based on the inverse of its period:**
  - Shorter periods ⇒ higher execution rate ⇒ higher priority
  - Longer periods ⇒ lower execution rate ⇒ lower priority
- In RMS, a process with higher priority should start at its new period.

P_1 is assigned a higher priority than P_2 as P_1 has a shorter period. (p1=50 and p2=100; t1=20 and t2=35)



51

Rate-Monotonic Scheduling (RMS) (cont'd)

- There are two scheduling points:
  - At each new period of a process, RMS scheduler either starts that process or continues the currently running process depending on whose priority is higher.
  - When a process finishes its execution, RMS scheduler may start a process whose new period was passed (if its priority is higher than other processes whose new periods were also passed).
    - However, we don't need to execute the same process again if its current deadline is already satisfied. That's why the CPU would be idle during (75, 100) in the previous example.
- Rate-Monotonic scheduling (RMS) algorithm **requires that processes are periodic.** However, at each scheduling point, RMS does not consider the current deadlines of processes.

52

Rate-Monotonic Scheduling (RMS) (cont'd)

- Rate-Monotonic Scheduling is considered optimal in that if a set of processes cannot be scheduled by this algorithm, it cannot be scheduled by any other algorithm that assigns static priorities.
- In general, for *n* periodic processes to meet their deadlines, the sum of their CPU utilization cannot exceed 1:

$$\sum_{i=1}^{n} \frac{ti}{pi} \leq 1$$

- **RMS is guaranteed to work for *n* processes if**

$$\sum_{i=1}^{n} \frac{ti}{pi} \leq n(2^{1/n} - 1)$$

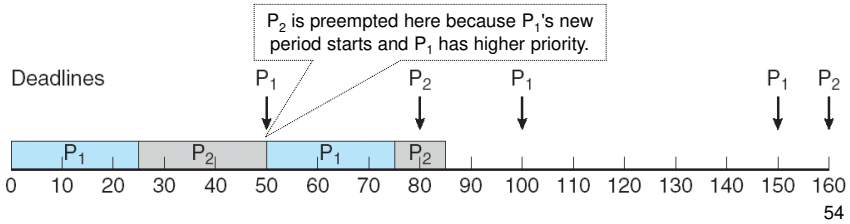
- when n =1, the maximum permitted CPU utilization is 1.
- when n =2, the maximum permitted CPU utilization is 0.83.
- when n is very large, the maximum permitted CPU utilization is $ln\ 2 \approx 0.693$.

53

Missed Deadlines with Rate Monotonic Scheduling

- We have two periodic processes P_1 and P_2:
  p_1=50 and p_2=80;
  t_1=25 and t_2=35
- The deadline for each process requires that it completes its CPU burst by the start of its next period.

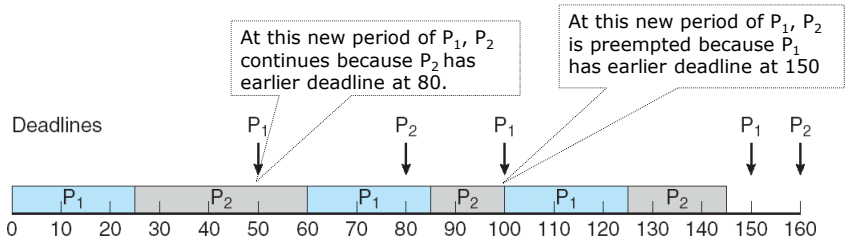
P_2 misses its first deadline, as shown below:



54

Earliest Deadline First Scheduling

- **Dynamic priorities are assigned according to deadlines: the earlier the deadline, the higher the priority**; and the later the deadline, the lower the priority.
- We have two periodic processes P_1 and P_2:
  p_1=50 and p_2=80;
  t_1=25 and t_2=35
- The deadline for each process requires that it completes its CPU burst by the start of its next period.



55

Earliest Deadline First (EDF) Scheduling (cont'd)

- There are two scheduling points:
  - At each new period of a process, EDF scheduler either starts that process or continues the currently running process depending on whose current deadline is earlier.
  - When a process finishes its execution, EDF scheduler may start a process (whose new period was passed) if its current deadline is earlier.
    - However, we don't need to execute the execute the same process again if its current deadline is already satisfied. That's why the CPU would be idle during (145, 150) in the previous example.
- Unlike the Rate-Monotonic scheduling (RMS) algorithm, **EDF scheduling does not require that processes are periodic, nor must require constant amount of CPU time per burst.**
- The only requirement is that each realtime process should announce its deadline to the scheduler when it becomes runnable.
- EDF scheduling is theoretically optimal: It can schedule processes so that each process can meet its deadline requirement, and the CPU utilization can be 100 percent.

56

Proportional Share Scheduling

- Allocating **T** shares of CPU time among all processes in the system.
  - A process can receive **N** shares where **N < T**, thus ensuring that the process will have **N**/**T** of the total CPU time.
- For example, if T=100 shares are divided among three processes (*A*, *B*, and *C*), such that *A* is assigned 50 shares, *B* is assigned 15 shares, and *C* is assigned 20 shares,
  - *A* will have 50 percent of total processor time.
  - *B* will have 15 percent of total processor time.
  - *C* will have 20 percent of total processor time.
- But it doesn't seem to consider the deadlines explicitly. So, a question is: If a process is allocated 50 shares during next 100 shares, when those 50 shares will be allocated?

57

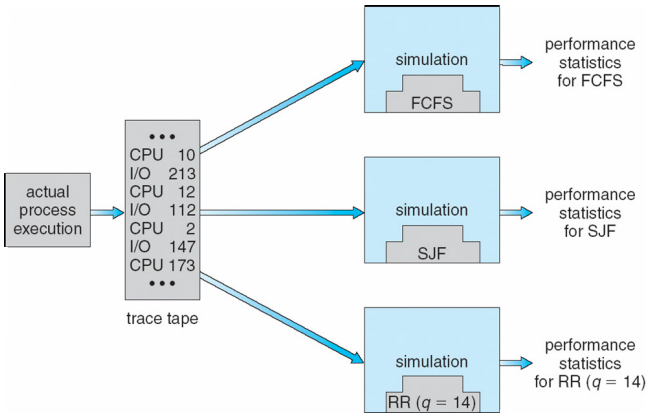
Algorithm Evaluation

- Deterministic modeling
- Queueing models
- Simulations
- Implementation

58

Evaluation of CPU Schedulers by Simulation

- A trace tape is created by monitoring the real system and recording the sequence of actual events of processes.



59