Real Time Systems1

Lecture 6

Task Classes

Precedence Constraints

Classical Uniprocessor Scheduling algorithms

Rate-monotonic Scheduling Algorithm

Task classes

- Tasks can be classified in two ways:

- 1. By the predictability of their arrival
  Periodic, aperiodic and sporadic
- 2. By the consequences of their not being executed on time.
  (Critical and non-critical tasks)

Task classes

- By the predictability of their arrival
- Periodic tasks consist of an infinite sequence of identical activities, called *instances* or *jobs*, that are regularly activated at a constant rate.
- Aperiodic tasks also consist of an infinite sequence of identical jobs (or instances); however, their activations are not regularly interleaved.
- An aperiodic task where consecutive jobs are separated by a minimum interarrival time is called a *sporadic task*.

Task classes

- **Critical and non-critical tasks**:
- Critical tasks are those whose timely execution is critical; if deadlines are missed catastrophes occur.
- Non-critical tasks: are non-critical to application.

Definitions:

- **Arrival time** *ai* is the time at which a task becomes ready for execution; it is
- also referred as *request time* or *release time* and indicated by *ri*;
- **Computation time** *Ci or Exi* is the time necessary to the processor for executing the task without interruption;
- **Absolute Deadline** *di* is the time before which a task should be completed to avoid damage to the system;

Definitions:

- **Relative Deadline** D_i is the difference between the absolute deadline and the request time: $D_i = d_i - r_i$
- **Start time** s_i is the time at which a task starts its execution;
- **Finishing time** f_i is the time at which a task finishes its execution;
- **Response time** R_i is the difference between the finishing time and the request time: $R_i = f_i - r_i$

Definitions:

- **Criticality** is a parameter related to the consequences of missing the deadline (typically, it can be hard, firm, or soft);
- **Value** v_i represents the relative importance of the task with respect to the other tasks in the system;
- **Lateness** L_i: $L_i = f_i - d_i$ represents the delay of a task completion with respect to its deadline; note that if a task completes before the deadline, its lateness is negative;

Definitions:

- **Tardiness** or *Exceeding time* E_i:
  $E_i = max(0, L_i)$ is the time a task stays active after its deadline;

- **Laxity** or *Slack time* X_i: $X_i = d_i - a_i - C_i$ is the maximum time a task can be delayed on its activation to complete within its deadline.

PRECEDENCE CONSTRAINTS

- In certain applications, computational activities cannot be executed in arbitrary order but have to respect some precedence relations defined at the design stage.
- Such precedence relations are usually described through a directed acyclic graph *G*, where tasks are represented by nodes and precedence relations by arrows.
- A precedence graph *G* induces a partial order on the task set.

PRECEDENCE CONSTRAINTS

- The notation $Ja \prec Jb$ specifies that task Ja is a *predecessor* of task Jb,
- meaning that G contains a directed path from node Ja to node Jb.
- The notation $Ja \rightarrow Jb$ specifies that task Ja is an *immediate predecessor* of Jb, meaning that G contains an arc directed from node Ja to node Jb.

PRECEDENCE CONSTRAINTS

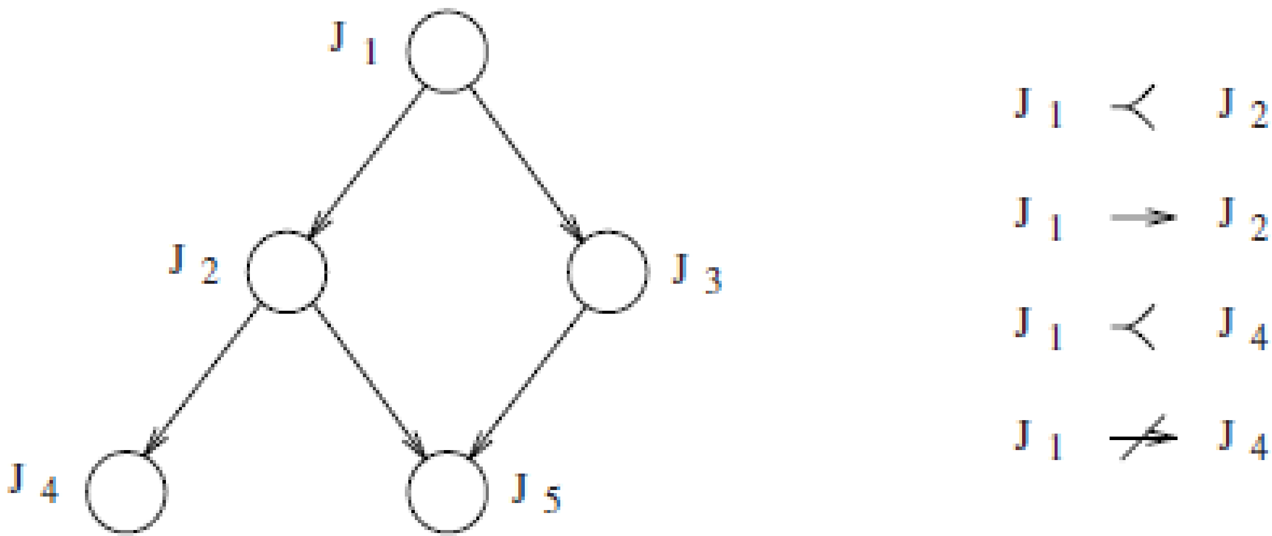

Figure 2.6 Precedence relations among five tasks.

Figure 2 illustrates a directed acyclic graph that describes the precedence constraints among five tasks.

PRECEDENCE CONSTRAINTS

- From the graph structure we observe that task $J1$ is the only one that can start executing since it does not have predecessors.
- Tasks with no predecessors are called *beginning tasks*.
- As $J1$ is completed, either $J2$ or $J3$ can start.
- Task $J4$ can start only when $J2$ is completed, whereas $J5$ must wait for the completion of $J2$ and $J3$.
- Tasks with no successors, as $J4$ and $J5$, are called *ending tasks*.

Classical Uniprocessor Scheduling algorithms

- In order to simplify the schedulability analysis, the following hypotheses are assumed on the tasks:
- **A1.** The instances of a periodic task τ_i are regularly activated at a constant rate. The interval T_i between two consecutive activations is the *period* of the task.
- **A2.** All instances of a periodic task τ_i have the same worst-case execution time C_i.
- **A3.** All instances of a periodic task τ_i have the same relative deadline D_i, which is equal to the period T_i. (Di = period of Ti)

Classical Uniprocessor Scheduling algorithms

- **A4.** All tasks in the system are independent; that is, there are no precedence relations and no resource constraints.
- In addition, the following assumptions are implicitly made:
- **A5.** No task can suspend itself, for example on I/O operations.
- **A6.** All tasks are released as soon as they arrive.
- **A7.** All overheads in the kernel are assumed to be zero

Classical Uniprocessor Scheduling algorithms
Rate-monotonic Scheduling algorithm

- It is a uniprocessor static-priority preemptive scheme. The following assumptions are made in addition to the above:
- A8 All tasks are periodic.
- A9 The relative deadline of a task is equal to its period. (Di = period of Ti)

Classical Uniprocessor Scheduling algorithms
Rate-monotonic Scheduling algorithm

- The priority of a task is inversely related to its period:
- If task Ti has a smaller period than task Tj, Ti has higher priority than Tj. Higher-priority tasks can preempt lower-priority tasks.

Classical Uniprocessor Scheduling algorithms
Rate-monotonic Scheduling algorithm

- If the total utilization of a tasks is no greater than $n*(2^{1/n} - 1)$ where n is the number of tasks to be scheduled then RM algorithm will schedule all the tasks to meet their respective deadlines.
- This condition is sufficient but not necessary if the total utilization is greater than $n*(2^{1/n} - 1)$.

Classical Uniprocessor Scheduling algorithms
Rate-monotonic Scheduling algorithm

- Example: Consider you have the following task set. Schedule them using RM algorithm:

Task	Ci or Exi	Pi or Ti
T1	2	6
T2	3	9
T3	1	15

Classical Uniprocessor Scheduling algorithms
Rate-monotonic Scheduling algorithm

- Solution: Note period of T1 is smaller than P2 and P2 is smaller than P3. So priority of T1 is the highest then T2 then T3.

- It means when T1 becomes it can preempt T2 and T3. T2 can preempt T3.

Classical Uniprocessor Scheduling algorithms
Rate-monotonic Scheduling algorithm

- The profile of T1 as the following

Task T1	Release ri	Ex1	di1
T1,1	0	2	6
T1,2	6	2	12
T1,3	12	2	18
T1,4	18	2	24
T1,5	24	2	30
T1,6	30	2	36
.			
.			
.			

Classical Uniprocessor Scheduling algorithms
Rate-monotonic Scheduling algorithm

The profile of Task2 is as the following:

Task 2	Release ri2	Ex2	di2
T2,1	0	3	9
T2,2	9	3	18
T2,3	18	3	27
T2,4	27	3	36
T2,5	36	3	45
T2,6	45	3	54
.			
.			
.			

Classical Uniprocessor Scheduling algorithms
Rate-monotonic Scheduling algorithm

- The profile of Task3 is as the following

Task3	Release ri3	Ex3	di3
T3,1	0	1	15
T3,2	15	1	30
T3,3	30	1	45
.			
.			
.			

Classical Uniprocessor Scheduling algorithms
Rate-monotonic Scheduling algorithm

- So the schedule becomes:

T1,1*		T2,1 *			T3,1*	T1,2*		Idle	T2,2*			T1,3*		idle	T3,2*	....
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

The total utilization is (Exi/p) = 2/6 + 3/9 + 1/15 = 0.7333

While 3 * $(2^{1/3} - 1)$ = 0.7788 or $(3*(\sqrt[3]{3} - 1))$

Because it is 0.7333 < 0.7788, ***so*** RM algorithm will schedule all the tasks to meet their respective deadlines.

HW:

Task	Ci or Exi	Pi or Ti
T1	3	20
T2	2	5
T3	2	10