# NPTEL Course: Real-Time Systems

## Model Questions and Answers

## 1st June, 2010

**PROF. RAJIB MALL,**
**DEPT. OF CSE,**
**IIT KHARAGPUR.**

# Contents

# Lectures 1-5: Introduction to Real-Time Systems

Q1: State whether the following statements are TRUE or FALSE. Justify your answer.

i. A hard real-time application consists of only hard real-time tasks.

**FALSE.** A hard real-time application may also contain several non-real-time tasks such as logging activities, etc.

ii. Every safety-critical real-time system contains a fail-safe state.

**FALSE.** Having fail-safe states in safety-critical real-time systems is meaningless because failure of a safety-critical system can lead to loss of lives, cause damage, etc. E.g.: a navigation system on-board an aircraft.

iii. A deadline constraint between two stimuli is a behavioral constraint on the environment of the system.

**TRUE.** It is a behavioral constraint since the constraint is imposed on the second stimulus event.

iv. Hardware fault-tolerance techniques are easily adaptable to provide software fault-tolerance.

**FALSE.** Hardware fault-tolerance is usually achieved using redundancy techniques. However, the property of statistical correlation of failures for software renders the technique ineffective.

v. A good algorithm for scheduling of hard real-time tasks tries to complete each task in the shortest possible time.

**FALSE.** A scheduling algorithm for hard real-time tasks is only concerned with completing the tasks before the deadlines. Unlike desktop

applications, there is no benefit in completing each task in the shortest possible time.

vi. All hard real-time systems usually are safety-critical in nature.

**FALSE.** Not all hard real-time systems are safety-critical in nature. E.g.: computer games, etc.

vii. It is ensured by the performance constraints on a real-time system That the environment of the system is well-behaved.

**FALSE.** Behavioral constraints on a real-time system ensure that the environment of the system is well-behaved.

viii. Soft real-time tasks do not have any associated time bounds.

**FALSE.** Soft real-time tasks also have time bounds associated with them. Instead of absolute values of time, the constraints are expressed in terms of the average response times required.

ix. The objective of any good hard real-time task scheduling algorithm is to minimize average task response times.

**FALSE.** A good hard real-time task scheduling algorithm is concerned with scheduling the tasks such that all of them can meet their respective deadlines.

x. The goal of any good real-time operating system to complete every hard real-time task as ahead of its deadline as possible.

**FALSE.** A good real-time operating system should try and complete the tasks such that they meet their respective deadlines.

Q2: With a suitable example explain the difference between the traditional notion of time and real-time.

Ans: In a real-time application, the notion of time stands for the absolute time which is quantifiable. In contrast to real time, logical time, used in most general category applications, deals with a qualitative notion of time and are expressed using event ordering relations. For example, consider the following part of the behavior of library automation software used to automate the bookkeeping activities of a college library: "After a *query book* command is given by the user, the details of all the matching books are *displayed* by the software".

## Q3: What is the difference between a performance constraint and a behavioral constraint in a real-time system?

Ans: Performance constraints are the constraints that are imposed on the response of the system. Behavioral constraints are the constraints that are imposed on the stimuli generated by the environment. Behavioral constraints ensure that the environment of a system is well-behaved, whereas performance constraints ensure that the computer system performs satisfactorily.

## Q4: Explain the important differences between hard, firm and soft real-time systems.

Ans: A hard real-time task is one that is constrained to produce its results within certain predefined time bounds. The system is considered to have failed whenever any of its hard real-time tasks does not produce its required results before the specified time bound. Unlike a hard real-time task, even when a firm real-time task does not complete within its deadline, the system does not fail. The late results are merely discarded. In other words, the utility of the results computed by a real-time task becomes zero after the deadline. Soft real-time tasks also have time bounds associated with them. However, unlike hard and firm real-time tasks, the timing constraints on soft real-time tasks are not expressed as absolute values. Instead, the constraints are expressed in terms of the average response times required.

## Q5: It is difficult to achieve software fault tolerance as compared to hardware fault tolerance. Why?

Ans: The popular technique to achieve hardware fault-tolerance is through redundancy. However, it much harder to achieve software fault-tolerance compared to hardware fault-tolerance. A few approaches have been proposed for software modeled on the redundancy techniques used in achieving hardware fault-tolerance. The reason is the statistical correlation of failures for software. The different versions of a software component show similar failure patterns, i.e., they fail due to identical reasons. Moreover, fault-tolerance using redundancy can only be applied to real-time tasks if they have large deadlines.

## Q6: What is a "fail-safe" state? Since safety-critical systems do not have a fail-safe state, how is safety guaranteed?

Ans: A fail-safe state of a system is one which if entered when the system fails, no damage would result. All traditional non-real-time systems do have one or more fail-safe states.

However, safety-critical systems do not have a fail-safe state. A safety-critical system is one whose failure can cause severe damages. This implies that the reliability requirement of a safety-critical system is very high.

## Q7: Give an example of an extremely safe but unreliable system?

Ans: Yes, document-processing software is an example of a system which is safe but may be unreliable. The software may be extremely buggy and can fail many times while using. But a failure of the software does not usually cause any significant damage of financial loss.

## Q8: List the different types of timing constraints that can occur in a real-time system?

Ans: The different timing constraints associated with a real-time system can be broadly classified into the following categories:

1) Performance constraints
2) Behavioral constraints

Each of the performance and behavioral constraints can further be classified into the following types:

1) Delay constraint
2) Deadline constraint
3) Duration constraint

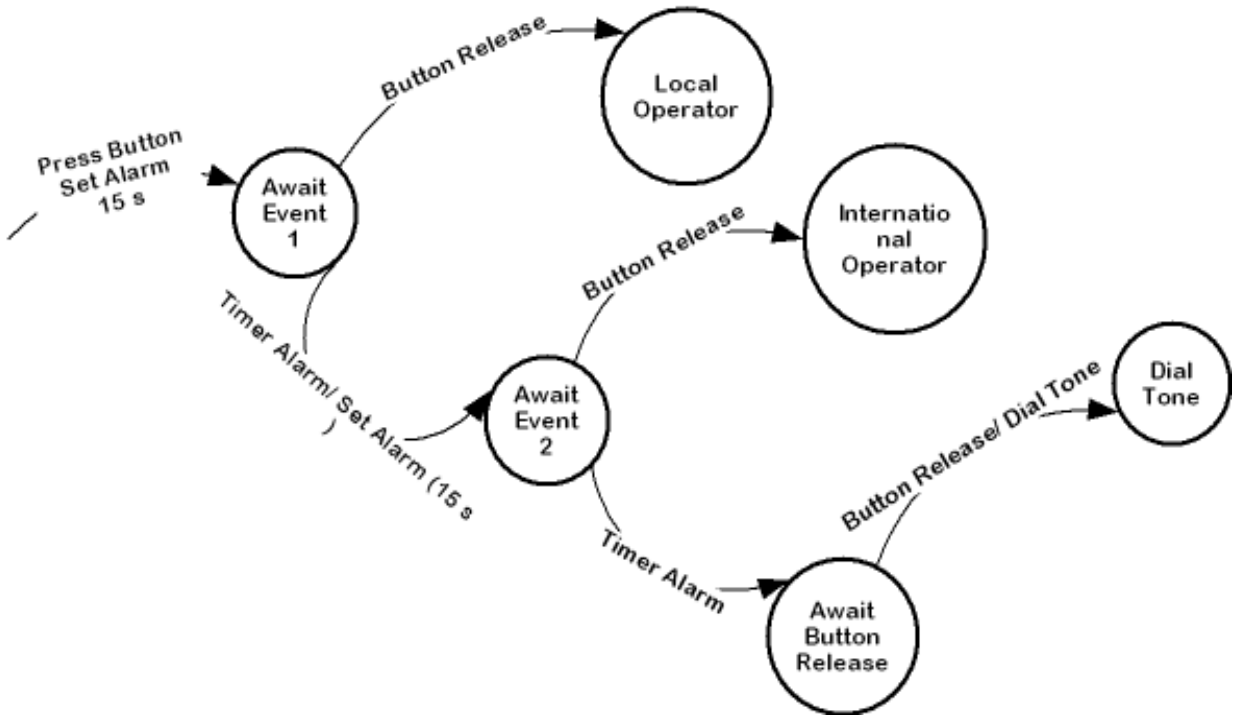
Q9: In the following, the partial behavior of a telephone system is given.

a) If you press the button of the handset for less than 15 s it connects to the local operator. If you press the button for any duration lasting between 15 to 30 s, it connects to the international operator. If you keep the button pressed for more than 30 s, then on releasing it would produce the dial tone.
b) Once the receiver of the handset is lifted, the dial tone must be produced by the system within 20 s, otherwise a beeping sound is produced until the handset is replaced.

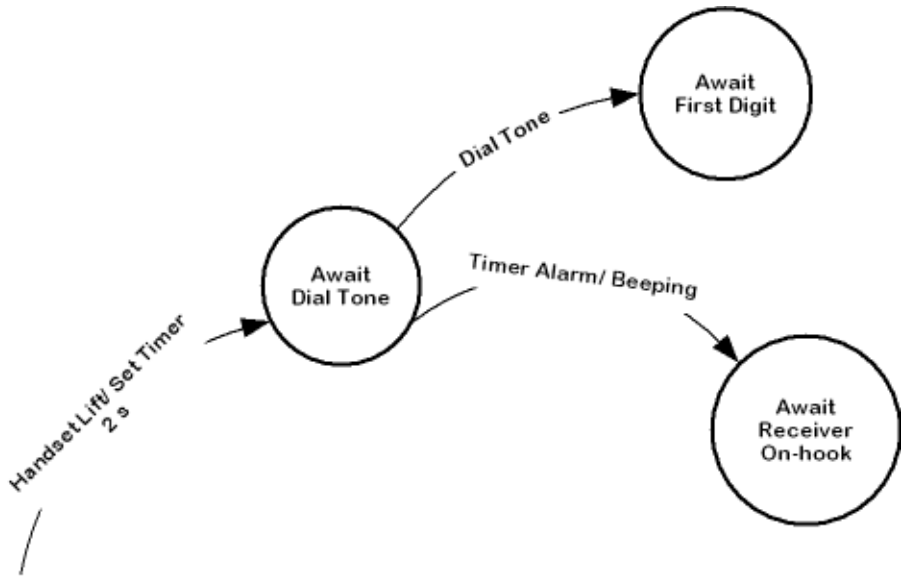
Draw the EFSM model for telephone system

Ans:

(a) The EFSM model for the problem is shown in the below figure.




(b) The EFSM model for the problem is shown in the below figure.



# Lectures 6-12: Real-Time Task Scheduling

Q1: State whether the following statements are TRUE or FALSE. Justify your answer.

i. Cyclic schedulers do not require storing a precomputed schedule unlike table-driven schedulers.

**FALSE.** A cyclic scheduler also needs to store a precomputed schedule. However, the precomputed schedule needs to be stored for only one *major cycle*. Each task in the task set which repeats identically in every major cycle.

ii. The upper bound on achievable utilization improves as the number of tasks in the system being developed increases when RMA is used for scheduling a set of hard real-time periodic tasks.

**FALSE.** Under RMA, the achievable CPU utilization falls as the number of tasks in the system increases. The utilization is maximum when there is only one task in the system. As the number of tasks reaches infinity, the utilization stabilizes at $\log_e 2$.

iii. For a non-preemptive operating system, RMA is an optimal static priority scheduling algorithm for a set of periodic real-time tasks.

**FALSE.** RMA is preemptive in nature, and hence is meaningless on a non-preemptive operating system.

iv. A pure table-driven scheduler is not as proficient as a cyclic scheduler for scheduling a set of hard real-time tasks.

**TRUE.** In a cyclic scheduler, the timer is set only once at the application initialization time, and interrupts only at the frame boundaries. But in a table-driven scheduler, a timer has to be set every time a task starts to run.

This represents a significant overhead and results in degraded system performance.

v. While scheduling a set of hard real-time periodic tasks using a cyclic scheduler, if more than one frame satisfies all the constraints on frame size then the largest of these frame sizes should be chosen.

**FALSE.** For a given task set, it is possible that more than one frame size satisfies all the three constraints. In such cases, it is better to choose the shortest frame size. This is because of the fact that the schedulability of a task increases as more number of frames becomes available over a major cycle.

vi. EDF possesses good transient overload handing capability.

**FALSE.** EDF is not good in handling transient overload conditions. This is a serious drawback in the EDF scheduling algorithm. When EDF is used to schedule a set of periodic real-time tasks, a task overshooting its completion time can cause some other task(s) to miss their deadlines.

vii. Data dependencies determine the precedence ordering among a set of tasks.

**FALSE.** Precedence ordering among a set of tasks can also be achieved using control dependence, for example, through passing of messages and events.

viii. Scheduling decisions are made only at the arrival and completion of tasks in a non-preemptive event-driven task scheduler.

**TRUE.** In event-driven scheduling, the scheduling points are defined by task completion and task arrival times. This is because during the course of execution of a task on the CPU, the task cannot be preempted.

ix. For uniprocessor systems, determining an optimal schedule for a set of independent periodic hard real-time tasks without any resource-sharing constraints under static priority conditions is an NP-complete problem.

**FALSE.** Optimal scheduling algorithms on uniprocessors already exist (e.g., RMA for static priority, and EDF for dynamic priority). However, finding an optimal schedule for a set of independent periodic hard real-time tasks without any resource-sharing constraints under static priority conditions on a multiprocessor is an NP-complete problem.

x. A set of periodic real-time tasks scheduled on a uniprocessor system using RMA scheduling show similar completion time jitter.

**TRUE.** Completion time jitters are caused by the basic nature of RMA scheduling which schedules task at the earliest opportunity at which it can run. Thus, the response time of a task depends on how many higher priority tasks arrive (or were waiting) during the execution of the task.

Q2: Explain scheduling point of a task scheduling algorithm? How the scheduling points are determined in (i) clock-driven, (ii) event-driven, (iii) hybrid schedulers?

Ans: The scheduling point of a scheduler are the points on the time line at which the scheduler makes decisions regarding which task is to be run next. In a clock-driven scheduler, the scheduling points are defined at the time instants marked by interrupts generated by a periodic timer. The scheduling points in an event-driven scheduler are generated by occurrence of certain events. For hybrid schedulers, the scheduling points are defined both through the clock interrupts and event occurrences.

Q3: What are the distinguishing characteristics of periodic, aperiodic, and sporadic real-time tasks?

Ans: A periodic task is one that repeats after a certain fixed time interval. The precise time instants at which periodic tasks recur are usually demarcated by clock interrupts. For this reason, periodic tasks are also referred to as clock-driven tasks.

A sporadic task is one that recurs at random instants. Each sporadic task is characterized by a parameter $g_i$ which implies that two instances of the sporadic task have to be separated by a minimum time of $g_i$. An aperiodic task is in many ways similar to a sporadic task. An aperiodic task can arise at random instants. In case of aperiodic tasks, the minimum separation $g_i$ between two consecutive instances can be 0. Also, the deadline for aperiodic task is expressed as either an average value or is expressed statistically.

## Q4: What is understood by jitter associated with a periodic task? Mention techniques by which jitter can be overcome.

Ans: Jitter is the deviation of a periodic task from its strict periodic behavior. The arrival time jitter is the deviation of the task from arriving at the precise periodic time of arrival. It may be caused by imprecise clocks, or other factors such as network congestions. Similarly, completion time jitter is the deviation of the completion of a task from precise periodic points. The completion time jitter may be caused by the specific scheduling algorithm employed which takes up as task for scheduling as per convenience and the load at an instant, rather than scheduling at some strict time instants.

Real-time programmers commonly handle tasks with tight completion time jitter requirements using any one of the following two techniques:

- If only one or two actions (tasks) have tight jitter requirements, these actions are assigned very high priority. This method works well only when there are a very small number of actions (tasks). When it is used in an application in which the tasks are barely schedulable, it may result in some tasks missing their respective deadlines.
- If jitter must be minimized for an application that is barely schedulable, each task needs to be split into two: one which computes the output but

does not pass it on, and one which passes the output on. This method involves setting the second task's priority to very high values and its period to be the same as that of the first task. An action scheduled with this approach will run one cycle behind schedule, but the tasks will have tight completion time jitter.

Q5: Can we consider EDF as a dynamic priority scheduling algorithm for real-time tasks?

Ans: EDF scheduling does not directly require any priority value to be computed for any task at any time, and in fact has no notion of a priority of a task. Tasks are scheduled solely on the proximity to their deadline. However, the longer a task waits in a ready queue, the higher is the chance (probability) of being taken up for scheduling. This can be considered to be a virtual priority value associated with a task which keeps increasing with time until the task is taken up for scheduling.

Q6: A real-time system consists of three tasks $T_1$, $T_2$, and $T_3$. Their characteristics have been shown in the following table.

| Task | Phase (ms) | Execution Time (ms) | Relative Deadline (ms) | Period (ms) |
|---|---|---|---|---|
| $T_1$ | 20 | 10 | 20 | 20 |
| $T_2$ | 40 | 10 | 50 | 50 |
| $T_3$ | 70 | 20 | 80 | 80 |

Suppose the tasks are to be scheduled using a table-driven scheduler. Compute the length of time for which the schedules have to be stored in the precomputed schedule table of the scheduler.

Ans: In table-driven scheduling, the size of the schedule that needs to be stored is equal to the LCM of the periods of the individual tasks. This value is called the major cycle of the set of tasks. The tasks in the schedule will automatically repeat

after every major cycle. The major cycle of a set of tasks is LCM of the periods even when the tasks have arbitrary phasing.

So, major cycle = LCM(20, 50, 80)=400 ms.

Q7: Using a cyclic real-time scheduler, suggest a suitable frame size that can be used to schedule three periodic tasks T1, T2, and T3 with the following characteristics:

| Task | Phase (ms) | Execution Time (ms) | Relative Deadline (ms) | Period (ms) |
|---|---|---|---|---|
| $T_1$ | 0 | 20 | 100 | 100 |
| $T_2$ | 0 | 20 | 80 | 80 |
| $T_3$ | 0 | 30 | 150 | 150 |

Ans: For the given task set, an appropriate frame size is the one that satisfies all the required constraints. Let F be the appropriate frame size.

Constraint 1: F>= max{execution time of a task}

⇨ F>= 30

Constraint 2: The major cycle M for the given task set *T* is

M= LCM(100, 80, 150) = 1200

*M* should be an integral multiple of F. This consideration implies that F can take on the values 30, 40, 60, 100, 120, 150, 200, 300, 400, 600, 1200 and still satisfy Constraint 1.

Constraint 3: To satisfy this constraint, we need to check whether a selected frame size F satisfies the inequality: $2F - GCD(F, p_i) <= d_i$ for each $p_i$.

Let us first try for F = 30.

$T_1$: 2*30 – GCD(30, 100) <= 100

⇨ 60 – 10 <= 100

⇨ Satisfied

$T_2$: 2*30 – GCD(30, 80) <= 80

⇨ 60 – 10 <= 80

⇨ Satisfied

$T_3$: 2*30 – GCD(30, 150) <= 150

⇨ 60 – 30 <= 150

⇨ Satisfied

Therefore, F=30 is a suitable frame size. We can carry out our computations for the other possible frame sizes, i.e. F = 40, 60, 100, 120, 150, 200, 300, 400, 600, 1200, to find out other possible solutions. However, it is better to choose the shortest frame size.

Q8: Determine whether the following set of periodic real-time tasks is schedulable on a uniprocessor using RMA.

| Task | Start Time (ms) | Processing Time (ms) | Period (ms) | Deadline (ms) |
|---|---|---|---|---|
| $T_1$ | 20 | 25 | 150 | 100 |
| $T_2$ | 40 | 7 | 40 | 40 |
| $T_3$ | 60 | 10 | 60 | 50 |
| $T_4$ | 25 | 10 | 30 | 20 |

Ans: Let us first compute the total CPU utilization achieved due to the given tasks.

$$U = \sum_{i=1}^{4} u_i = \frac{25}{150} + \frac{7}{40} + \frac{10}{60} + \frac{10}{30} = 0.84 \leq 1$$

Therefore, the necessary condition is satisfied.

The sufficiency condition is given by

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

Therefore, $0.84 \leq 4(2^{1/4} - 1)$
$= 0.84 \leq 0.76$

⇨ Not satisfied.

Although, the given set of tasks fails the Liu and Layland's test which is pessimistic in nature, we need to carry out Lehoczky's test.

We need to reorder the tasks according to their decreasing priorities.

| Task | Start Time (ms) | Processing Time (ms) | Period (ms) | Deadline (ms) |
|---|---|---|---|---|
| $T_4$ | 25 | 10 | 30 | 20 |
| $T_2$ | 40 | 7 | 40 | 40 |
| $T_3$ | 60 | 10 | 60 | 50 |
| $T_1$ | 20 | 25 | 150 | 100 |

Testing for task $T_4$: Since $e_4 <= d_4$, therefore, $T_4$ would meet its first deadline.

Testing for task $T_2$: $7 + \left\lceil \frac{40}{30} \right\rceil * 10 \leq 40$

⇨ Satisfied.

⇨ Task $T_2$ would meet its first deadline.

Testing for task $T_3$: $10 + \left\lceil \frac{60}{40} \right\rceil * 7 + \left\lceil \frac{60}{30} \right\rceil * 10 \leq 50$

⇨ Satisfied.

⇨ Task $T_3$would meet its first deadline.

Testing for task $T_1$: $25 + \left\lceil \frac{150}{60} \right\rceil * 10 + \left\lceil \frac{150}{40} \right\rceil * 7 + \left\lceil \frac{150}{30} \right\rceil * 10 \leq 100$

- ⇨ Not satisfied.
- ⇨ Therefore, task $T_1$ would fail to meet its first deadline.

Hence, the given task set is not RMA schedulable.

# Lectures 13-16: Resource Sharing and Dependencies among Real-Time Tasks

Q1: State whether the following statements are TRUE or FALSE. Justify your answer.

i. RMA is optimal for scheduling access of several hard real-time periodic tasks to a certain shared critical resource.

**FALSE.** RMA schedulers impose no constraints on the orders in which various tasks execute. Therefore, the schedules produced by RMA might violate the constraints imposed due to task dependencies.

ii. Unless a suitable resource-sharing protocol is used, even the lowest priority task in a real-time system may suffer from unbounded priority inversions.

**FALSE.** Unbounded priority inversion occurs when a higher priority task waits for a lower priority task to release a resource it needs, and meanwhile the intermediate priority tasks preempt the lower priority task from CPU usage repeatedly.

iii. Scheduling a set of real-time tasks for access to a set of non-preemptable resources using PIP results in unbounded priority inversions for tasks.

**FALSE.** PIP allows real-time tasks share critical resources without letting them incur unbounded priority inversions.

iv. A task can undergo priority inversion for some duration under PCP even if it does not require any resource.

**TRUE.** Under PCP, inheritance-related inversion can occur. In such scenarios, an intermediate priority task not needing a critical resource is kept waiting.

v. PCP is a efficient protocol to share a set of serially reusable preemptable resources among a set of real-time tasks.

**FALSE.** PCP is not well-suited for sharing a set of serially reusable preemptable resources among a set of real-time tasks. EDF or RMA are more efficient.

vi. A separate queue is maintained for the waiting tasks for each critical resource in HLP.

**FALSE.** Unlike PIP, HLP does not maintain a separate queue for each critical resource. The reason is that whenever a task acquires a resource, it executes at the ceiling priority of the resource, and the other tasks that may need this resource do not even get a chance to execute and request for the resource.

vii. HLP overcomes deadlocks while sharing critical resources among a set of real-time tasks.

**TRUE.** HLP overcomes the deadlock problem possible with PIP.

viii. Under HLP, tasks are single-blocking.

**TRUE.** When HLP is used for resource sharing, once a task gets a resource required by it, it is not blocked any further. This means that when a task acquires one resource, all the other resources required by it must be free.

ix. Under PCP, the highest priority task does not suffer any inversions when sharing certain critical resources.

**FALSE.** Even under PCP, the highest priority protocol can suffer from direct and avoidance-related inversions.

x. Under PCP, the lowest priority task does not suffer any inversions when sharing certain critical resources.

**TRUE.** In PCP, the lowest priority task does not suffer from any of direct, inheritance-related, or avoidance-related inversions.

Q2: Explain priority inversion in the context of real-time scheduling?

Ans: When a lower priority task is already holding a resource, a higher priority task needing the same resource has to wait and cannot make progress with its computations. The higher priority task remains blocked until the lower priority task releases the required non-preemptable resource. In this situation, the higher priority task is said to undergo simple priority inversion on account of the lower priority task for the duration it waits while the lower priority task keeps holding the resource.

Q3: What can be the types of priority inversions that a task might undergo on account of a lower priority task under PCP?

Ans: Tasks sharing a set of resources using PCP may undergo three important types of priority inversions: direct inversion, inheritance-related inversion, and avoidance inversion.

Direct inversion occurs when a higher priority task waits for a lower priority task to release a resource that it needs. Inheritance-related inversion occurs when a lower priority task is holding a resource and a higher priority task is waiting for it. Then, the priority of the lower priority task is raise to that of the waiting higher priority task by the inheritance clause of the PCP. As a result, the intermediate priority tasks not needing the resource undergo inheritance-related inversion.

In PCP, when a task requests a resource its priority is checked against CSC. The requesting task is granted use of the resource only when its priority is greater than CSC. Therefore, even when a resource that a task is requesting is idle, the requesting task may be denied access to the resource if the requesting task's priority is less than CSC. A task, whose priority is greater than the currently

executing task, but lesser than the CSC and needs a resource that is currently not in use, is said to undergo avoidance-related inversion.

## Q4: How are deadlocks, unbounded priority inversions, and chain blocking prevented using PCP?

Ans: Deadlocks occur only when different (more than one) tasks hold part of each other's required resources at the same time, and then they request for the resources being held by each other. But under PCP, when one task is executing with some resources, any other task cannot hold a resource that may ever be needed by this task. That is, when a task is granted one resource, all its required resources must be free. This prevents the possibility of any deadlock.

PCP overcomes unbounded priority inversions because whenever a high priority task waits for some resources which are currently being used by a low priority task, then the executing lower priority task is made to inherit the priority of the higher priority task. So, the intermediate priority tasks cannot preempt lower priority task from CPU usage. Therefore, unbounded priority inversions cannot occur in PCP.

Tasks are single blocking under PCP. That means, under PCP a task can undergo at most one inversion during its execution. This feature of PCP prevents chain blocking.

## Q5: Can PIP and PCP be considered as greedy algorithms?

Ans: In PIP, whenever a request for a resource is made, the resource will be allocated to the requesting task if it is free. However, in PCP a resource may not be granted to a requesting task even if the resource is free. This strategy in PCP helps in avoiding potential deadlocks.

## Q6: The following table shows the details of tasks in a real-time system. The tasks have zero phasing and repeat with a period of 90 ms. Determine a feasible schedule to be used by a table-driven scheduler.

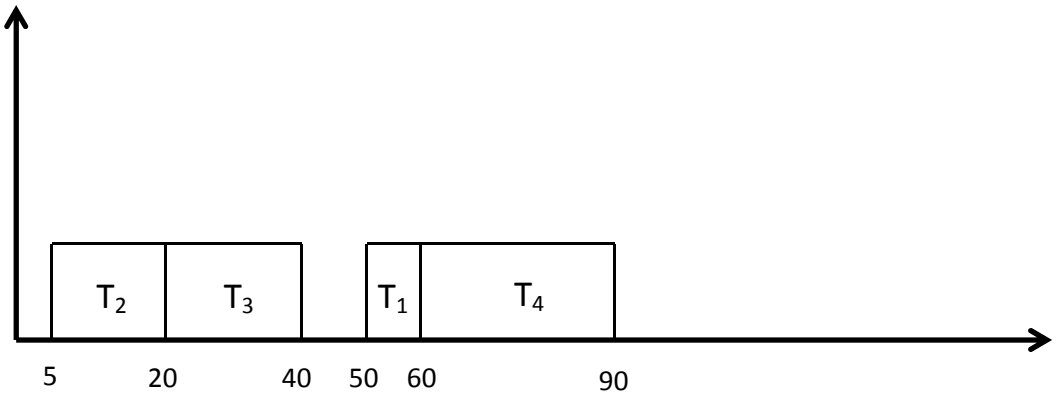
| Task | Computation Time $e_i$ (ms) | Deadline $d_i$ (ms) | Dependency |
|---|---|---|---|
| $T_1$ | 30 | 90 | - |
| $T_2$ | 15 | 40 | $T_1$, $T_3$ |
| $T_3$ | 20 | 40 | $T_1$ |
| $T_4$ | 10 | 70 | $T_2$ |

Ans:

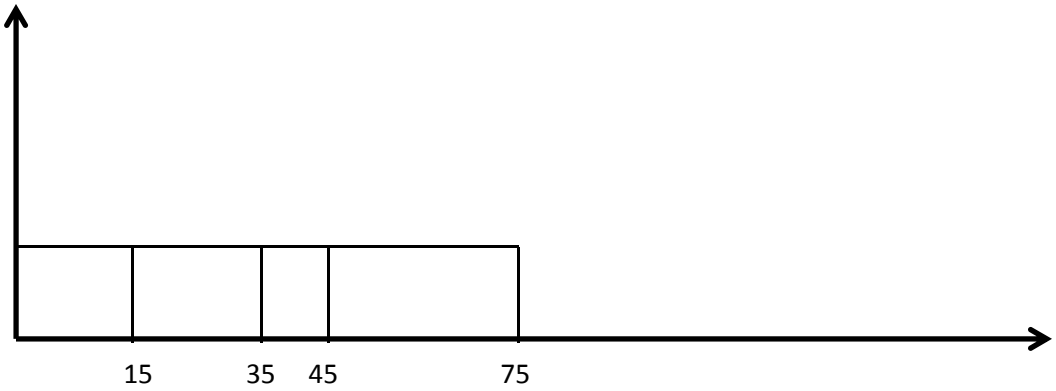
Step 1: Sort the tasks in increasing order of their deadlines.

$T_2$ $T_3$ $T_4$ $T_1$

Step 2: Schedule tasks as late as possible without violating constraints.




Step 3: Move tasks as early as possible without altering the schedule.



# Lectures 17-20: Scheduling Real-Time Tasks in Multiprocessor and Distributed Systems

Q1: State whether the following statements are TRUE or FALSE. Justify your answer.

i. By extending EDF, we can generate optimal scheduling schemes for hard real-time tasks in multiprocessor computing environments.

**FALSE.** Determining an optimal schedule for a set of real-time tasks on a multiprocessor or a distributed system is an NP-hard problem.

ii. It is possible to keep the good clocks of a distributed system having 12 clocks synchronized using distributed clock synchronization if only two of the clocks are byzantine.

**TRUE.** If less than one-third of the clocks are bad or byzantine, then we can have the good clocks approximately synchronized in distributed clock synchronization technique.

iii. Task allocation using bin-packing algorithm along with task scheduling at the individual nodes using the EDF algorithm is optimal in a distributed hard real-time computing environment.

**FALSE.** Finding an optimal schedule for a set of independent periodic hard real-time tasks without any resource-sharing constraints under static priority conditions on a multiprocessor is an NP-complete problem. Most scheduling algorithms in a distributed hard real-time computing environment are based on heuristics.

iv. Task allocation is done statically in the focused addressing and bidding algorithm in distributed real-time systems.

**FALSE.** Focused addressing and bidding algorithm used for allocating tasks in distributed real-time systems is a dynamic allocation technique.

v. Dynamic task arrivals can efficiently be handled using the focused addressing and bidding algorithm in multiprocessor-based real-time systems.

**TRUE.**

vi. The communication overhead incurred due to Buddy algorithms is less compared to focused addressing and bidding algorithms in multiprocessor real-time task scheduling.

**TRUE.**

vii. In a distributed system, allocating a set of real-time tasks can be done optimally using the bin-packing algorithm.

**FALSE.** Determining an optimal schedule for a set of real-time tasks on a multiprocessor or a distributed system is an NP-hard problem.

viii. A simple internal synchronization scheme using a time server makes the synchronization time incrementally delayed by the average message transmission time after every synchronization interval in a distributed system where the message communication time is non-zero and significant.

**FALSE.** The synchronization time will not be incrementally delayed by the average message transmission time. The synchronization time will, however, be delayed by the message jitter.

Q2: Why are algorithms which can satisfactorily schedule real-time tasks on multiprocessors not satisfactory to schedule real-time tasks on distributed systems?

Ans: A basic difference between multiprocessor and distributed systems is sharing of physical memory. Multiprocessor systems (aka. tightly-coupled systems) are characterized by the existence of a shared physical memory. In contrast, a distributed system (aka. loosely-coupled system) is devoid of any shared physical memory. In a tightly-coupled system, the interprocess communication is inexpensive and is achieved through read and writes to the shared memory. However, the same is not true for distributed systems where inter-task communication times are comparable to the task execution times. Due to this, a multiprocessor system may use a centralized scheduler/dispatcher whereas a distributed system cannot.

## Q3: What is it required to synchronize the clocks in a distributed real-time system? Compare the advantages and disadvantages of centralized and the distributed clock synchronization.

Ans: Besides the traditional use of clocks in a computer system, clocks are also used for determining timeouts and time stamping. However, different clocks in a distributed system tend to diverge since it is almost impossible to have two clocks that run exactly at the same speed. This lack of synchrony among clocks is expressed as the clock skew and determines the attendant drifts of the clocks with time. Lack of synchrony and drift among clocks makes the time stamping and timeout operations in a distributed real-time system meaningless. Therefore, to have meaningful timeouts and time stamping spanning more than one node of a distributed system, the clocks need to be synchronized.

The main problem with centralized clock distribution scheme is that it is susceptible to single point failure. Any failure of the master clock causes breakdown of the synchronization scheme. Distributed clock synchronization overcomes this handicap by periodically exchanging the clock readings of each node. However, distributed clock synchronization needs to take care of the fact that some nodes may have bad or byzantine clocks. A disadvantage of the distributed clock synchronization is the larger communication overhead that is incurred in synchronizing clocks as compared to the centralized scheme.

Q4: Modern commercial real-time operating systems use gigahertz clocks, while the clock resolution provided is rarely finer than few hundreds of milliseconds. Why?

Ans: With the current technology, it is possible to have clocks of nanosecond granularity. But still, the time granularities of modern RTOS' are of the order of microseconds or even milliseconds. The reason can be attributed to the following. The kernel maintains a software clock. After each interrupt of the hardware clock, the software clock is updated. A task can read the time using the POSIX function, clock_gettime(). So the finer the resolution, the more frequent is the hardware interrupts, and the larger is the amount of processor time servicing it. However, the response of the clock_gettime() function is not deterministic and the variation is greater than few hundreds of nanoseconds. So any software clock resolution finer than this is not meaningful.

Q5: With respect to the communication overhead and the scheduling proficiency, discuss the relative merits of the focused addressing and bidding and the buddy schemes.

Ans: The focused addressing and bidding strategy incurs a high communication overhead in maintaining the system load table at the individual processors. Window size is an important parameter in determining the communication overhead incurred. If the window size is increased, then the communication overhead decreases; however, the information at various processors would be obsolete. This may lead to a scenario where none of the focused processors bids due to status change in the window duration. If the window size is too small, then the information would be reasonably up to date at the individual processors, but the communication overhead in maintaining the status tables would be unacceptably high.

The buddy algorithm tries to overcome the high communication overhead of the focused addressing and bidding algorithm. Unlike focused addressing and bidding, in the buddy algorithm broadcast does not occur periodically at the end of every

window. A processor broadcasts only when the status of a processor changes either from overloaded to underloaded or vice versa. Further, whenever the status of a processor changes, it does not broadcast this information to all processors and limits it only to a subset of processors called its buddy set.

Q6: In a distributed system with six clocks, the maximum difference of between any two clocks is 10 ms. The individual clocks have a maximum rate of drift of $2 * 10^{-6}$. Ignore clock setup times and communication latencies.

a) What is the rate at which the clocks need to resynchronize using
   i. a simple central time server method?
b) What is the communication overhead in each of the two schemes?

Ans:

a) i) When clocks are resynchronized every $\Delta T$ s, maximum drift between any two clocks is limited to $2\rho\Delta T = \varepsilon$.

⇨ $2*2*10^{-6}*\Delta T = 10*10^{-3}$

⇨ $\Delta T = (10*10^{-3})/2*2*10^{-6}$

⇨ $\Delta T = 2500$ s.

b) i) Master time server transmits (n-1) = 5 messages per resynchronization interval.

Number of resynchronization intervals per hour = (60*60)/2500 = 1.44.

Number of messages transmitted per hour = 1.44 * 5 = 7.2 ≈ 8.

Q7: A distributed real-time system consisting of 10 clocks has up to three clocks which are byzantine. The maximum drift between the clocks has to be less than $\varepsilon$ = 1 ms. The maximum drift rate between two clocks is $\rho = 5*10^{-6}$. Compute the resynchronization interval.

Ans:

We know that

$\Delta T = \varepsilon/2n\rho$

⇨ $\Delta T = 10^{-3}/(2*10*5*10^{-6})$

⇨ $\Delta T = 10$ s.

Q8: A distributed system has 12 clocks with at best two byzantine clocks. The clocks are required to be resynchronized within 1 ms of each other. The maximum drift rate of the clocks is $6*10^{-6}$. Compute

- The rate at which the clocks need to exchange time values,
- The total number of message exchanges required per hour for synchronization.

Ans:

Time difference between two byzantine clocks = $(3*\varepsilon*2)/n$

Resynchronization needed when clocks diverge by

$2*\Delta T\rho = (n\varepsilon-6\varepsilon)/n$

⇨ $2\Delta T\rho = \varepsilon/2$

⇨ $\Delta T = 10^{-3}/2*2*6*10^{-6}$

⇨ $\Delta T = 41.67$ s.

Each node transmits (n-1) messages per resynchronization interval.

Therefore, total message transmissions per resynchronization interval = n*(n-1) = 132.

Number of resynchronization intervals per hour = 60*60/41.67 = 86.41

Number of messages transmitted per hour = 86.41*132 = 11406.

# Lectures 21-30: Real-Time Operating Systems

Q1: State whether the following statements are TRUE or FALSE. Justify your answer.

i. Real-time processes are scheduled at higher priorities than the kernel processes in RTLinux.

**TRUE.** The kernel runs as a low priority background task of RTLinux.

ii. Commercial real-time operating systems such as PSOS and VRTX support EDF scheduling of tasks.

**TRUE.** PSOS and VRTX support RMA by assigning static priorities to tasks. Since EDF requires dynamic priority computation, EDF is not supported on PSOS and VRTX.

iii. POSIX 1003.4 specifies that real-time processes are to be scheduled at priorities higher than kernel processes.

**FALSE.** POSIX.4 does not specify at what priority levels at what priority levels the kernel services are to be executed.

iv. Computation intensive tasks dynamically take on higher priorities in Unix.

**FALSE.** Under the Unix operating system, i/o bound tasks are assigned higher priorities to keep the i/o channels as busy as possible.

v. Any real-time priority level can be assigned to tasks for implementing PCP in Windows NT with ceiling values computed on these.

**FALSE.** To implement PCP in Windows NT, the real-time tasks need to have even priorities (i.e. 16, 18, …, 30) only.

vi. Task switching time on the average is larger than task preemption time.

**FALSE.** Task switching is a part of the total time taken to preempt a task. In addition, task preemption also requires comparing the priorities of the currently running task and the tasks in the ready queue.

vii. In general, segmented addressing incurs lower jitter in memory access compared to virtual addressing scheme.

**TRUE.** Memory access using virtual addressing schemes can incur large jitter depending on whether the required page is in the physical memory or has been swapped out.

viii. POSIX by ANSI/IEEE enables executable files to be portable across different Unix machines.

**FALSE.** The main goal of POSIX is application portability at the source code level.

ix. In an implementation of PCP at the user-level, half of the available priority levels are meaningfully assigned to the tasks if FIFO scheduling is not supported by the operating system among equal priority tasks.

**TRUE.** If the highest priority among all tasks needing a resource is 2*n, then the ceiling priority of the resource is 2*n+1.

x. For a real-time operating system which does not support memory protection, a procedure call and a system call are indistinguishable.

**TRUE.**

xi. Watchdog timers are used to start sensor and actuator processing tasks at regular intervals.

**FALSE.** Watchdog timers are used to detect if a task misses its deadline, and then to initiate exception handling procedures upon a deadline miss. It is also used to trigger a system reset or other corrective action if the main program hangs due to some fault condition.

Q2: List the important features that are required to be supported by a RTOS.

Ans: The following is a list of the important features that an RTOS needs to support:

- Real-time priority levels
- Fast task preemption
- Predictable and fast interrupt latency
- Support for resource-sharing among real-time tasks
- Requirements on memory management
- Support for asynchronous I/O

Q3: What is the difference between synchronous and asynchronous I/O? Which one is better suited for use in real-time applications?

Ans: Asynchronous I/O means non-blocking I/O. Under synchronous I/O system calls, a process needs to wait till the hardware has completed the physical I/O. Thus, in case of synchronous I/O, the process is blocked while it waits for the results of the system call.

On the other hand, if a process uses asynchronous I/O, then the system call will return immediately once the I/O request has been passed down to the hardware or queued in the operating system typically before the physical I/O operation has even begun. The execution of the process is not blocked because it does not need

to wait for the results of the system call. This helps in achieving deterministic I/O times in an RTOS.

## Q4: Describe an open system? How does an open system compare with a close system?

Ans: An open system is a vendor-neutral environment which allows users to intermix hardware, software, and networking solutions from different vendors. Open systems are based on open standards and are not copyrighted. The important advantages of an open system over a closed system are interoperability and portability. Other advantages are that it reduces the cost of system development and the time to market a product. It helps increase the availability of add-on software packages, enhances the ease of programming and facilitates easy integration of separately developed modules.

## Q5: What are the shortcomings of Windows NT for developing a hard real-time application?

Ans: The following are the major shortcomings of Windows NT when used to develop real-time applications:

- Interrupt processing – In Windows NT, the priority level of interrupts is always higher than that of user-level threads including the threads of real-time class. It is not possible for a user-level thread to execute at a priority higher than that of ISRs or DPCs. Therefore, even ISRs and DPCs corresponding to very low priority tasks can preempt real-time processes. As a result, the potential blocking of real-time tasks due to DPCs can be large.
- Support for resource sharing protocols –Windows NT does not provide any support to real-time tasks to share critical resources among themselves.

## Q6: What are the drawbacks in using Unix kernel for developing real-time applications?

Ans: The following are the major shortcomings in using traditional Unix for real-time application development:

- Non-preemptive kernel – Unix kernel is non-preemptive, i.e., a process running in the kernel mode cannot be preempted by other processes. A consequence of this is that even when a low priority process makes a system call, the high priority processes would have to wait until the system call by the low priority process completes. For real-time applications, this leads to priority inversion.
- Dynamic priority levels – In traditional Unix systems, real-time tasks cannot be assigned static priority values. Soon after a programmer sets a priority value for a task, the OS keeps on altering it during the course of execution of the task. This makes it very difficult to schedule real-time tasks using algorithms such as EDF or RMA.
- Insufficient device driver support – In Unix (System V), device driver runs in kernel mode. Therefore, if support for a new device is to be added, then the driver module has to be linked to the kernel modules – necessitating a system generation step.
- Lack of real-time file services – In Unix, file blocks are allocated as and when they are requested by an application. As a consequence, while a task is writing to a file, it may encounter an error when the disk runs out of space. In other words, no guarantee is given that disk space would be available when a task writes a block to a file. Traditional file writing approaches also result in slow writes because the required space has to be allocated before writing a block. Another problem with traditional file systems is that blocks of the same file may not be contiguously located on the disk. This would result in read operations taking unpredictable times, resulting in jitter in data access.
- Inadequate timer services support – In Unix, real-time timer support is insufficient for hard real-time applications. The clock resolution that is provided to applications is 10 ms, which is too coarse for many hard real-time applications.

Q7: Is it true that even in uniprocessor systems multithreading can result in faster response times compared to single-threaded tasks?

Ans: Yes, multithreading can result in faster response times even on a single processor systems. An advantage of multithreading, even for single-CPU systems, is the ability for an application to remain responsive to input. In a single threaded program, if the main execution thread blocks on a long running task, the entire application can appear to freeze. Thread creation and switching is much less costlier as compared to tasks. By moving such long running tasks to a *worker thread* that runs concurrently with the main execution thread, it is possible for the application to remain responsive to user input while executing tasks in the background.

Q8: Why is dynamically changing the priority levels of tasks important for traditional operating systems? How does this property affect real-time systems?

Ans: One major design decision in traditional operating systems is to provide acceptable response times to all the user processes. It is normally observed that I/O transfer rate is primarily responsible for the slow response time. Processors are extremely fast compared to the transfer times of I/O devices. To mitigate this problem, it is desirable to keep the I/O channels as busy as possible. This can be achieved by assigning the I/O bound tasks high priorities. To keep the I/O channels busy, any task performing I/O should not be kept waiting very long for the CPU. For this reason, as soon as the task blocks for I/O, its priority is increased by a priority recomputation rule. This gives the interactive users good response time.

However, for hard real-time systems dynamic shifting of priority values is clearly inappropriate, as it prevents tasks being constantly scheduled at high priority levels, and also prevents scheduling under popular real-time task scheduling algorithms such as EDF and RMA.

Q9: Explain the differences between a system call and a function call? What problems may arise if a system call is made indistinguishable from a function call?

Ans: Application programs invoke operating system services through system calls. Examples of system calls include operating system services for creating a process, I/O operations, etc. This is because certain operations can only be performed in the kernel mode.

In many embedded systems, kernel and user processes execute in the same address space, i.e., there is no memory protection. This makes debugging applications difficult, since a run-away pointer can corrupt the operating system code making the system 'freeze'.

Q10: Explain the requirements of a real-time file system? How is it compared to traditional file systems?

Ans: Real-time file systems overcome the problem of unpredictable read times and data access jitter by storing files contiguously on the disk. Since the file system preallocates space, the times for read and write operations are more predictable.