Programación Paralela, Concurrente y de Tiempo Real

Tiempo Real

1. Introducción al tiempo real

2. Tareas periódicas

3. Modelado de sistemas de tiempo real

4. Sincronización

5. Soporte en el sistema operativo

Material based on the tutorial by the Software Engineering Institute, Carnegie Mellon University

2. Periodic Tasks

2.1 Basic concepts

2.2 Principles of fixed priority analysis

2.3 Rate monotonic priorities

2.4 Utilization test

2.5 Response time analysis

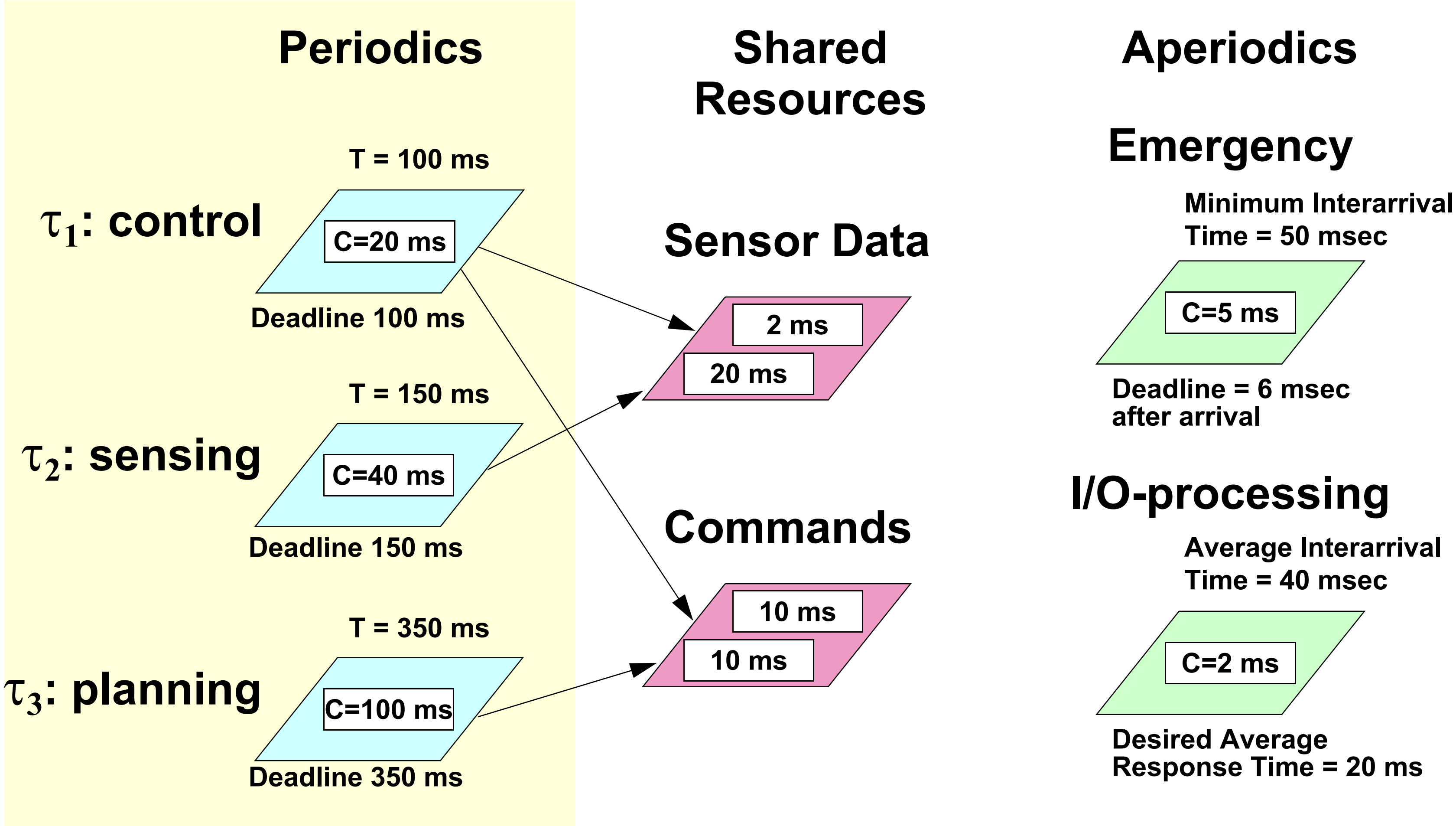
2.6 Preperiod deadlines: priority assignment

2.7 Modelling task switching

2.8 Schedulability with interrupts

2.9 Non-preemptible sections

2.1 Basic Concepts: A Sample Problem - Periodics



Notes:

In this section, we will discuss the basic principles of Rate Monotonic Analysis. We will present methods to analyze the schedulability of a set of periodic independent tasks. This kind of task set is very unrealistic, but the techniques show in this chapter are easily extended to support all the issues that appear in practical real-time systems.

This sample task set is simple and yet embodies many different types of real-time requirements. This task set is completely analyzable using the methods outlined in this presentation. For now we will concentrate on the periodic requirements and ignore any pre-period deadlines:

- Periodic task τ_1: execution time = 20 msec; period = 100 msec; deadline is at the end of each period. (we consider the actual deadline of τ_1 later)
- Periodic task τ_2: execution time = 40 msec; period = 150 msec; deadline is at the end of each period .
- Periodic task τ_3: execution time = 100 msec; period = 350 msec; deadline is at the end of each period.

Concepts and Definitions - Periodics

Periodic task

- initiated at fixed intervals
- must finish before start of next cycle

Task's CPU utilization: $U_i = C_i / T_i$

- C_i = compute time (execution time) for task τ_i
- T_i = period of task τ_i
- P_i = priority of task τ_i
- D_i = deadline of task τ_i
- ϕ_i = phase of task τ_i
- R_i = response time of task τ_i

CPU utilization for a set of tasks: $U = U_1 + U_2 + \ldots + U_n$

Notes:

A periodic task becomes ready to execute at fixed intervals. The time between task initiations is called the *period* of the task. The deadline of a task is the maximum time at which the execution of the task must have been completed, counting from the instant of task initiation. For the moment, we will consider task deadlines to coincide with the start of the next period.

C_i and T_i represent the worst-case execution time and period, respectively, for task τ_i. A task's utilization of the CPU is C_i/T_i. Total CPU utilization is the sum of the utilizations of all the tasks. The deadline of task τ_i is D_i. The priority of task τ_i is D_i. The initial phase of task τ_i is ϕ_i; the phase is the time at which the task is first activated. Since phases may drift, we will always assume the worst-case phase in the analysis.

The worst-case response time of task τ_i is R_i. This is the maximum difference between the activation time (i.e., the beginning of the task's period) and the response time.

2.2 Principles of Fixed Priority Analysis

Two concepts help building the worst-case condition under fixed priorities:

- *Critical instant*. The worst-case response time for all tasks in the task set is obtained when all tasks are activated at the same time
- *Checking only the deadlines in the worst-case busy period*.
  - for task: interval during which the processor is busy executing τ_i or higher priority tasks

Based on these concepts, several results arise:

- Optimality of rate monotonic priorities
- Utilization bound test
- Exact test

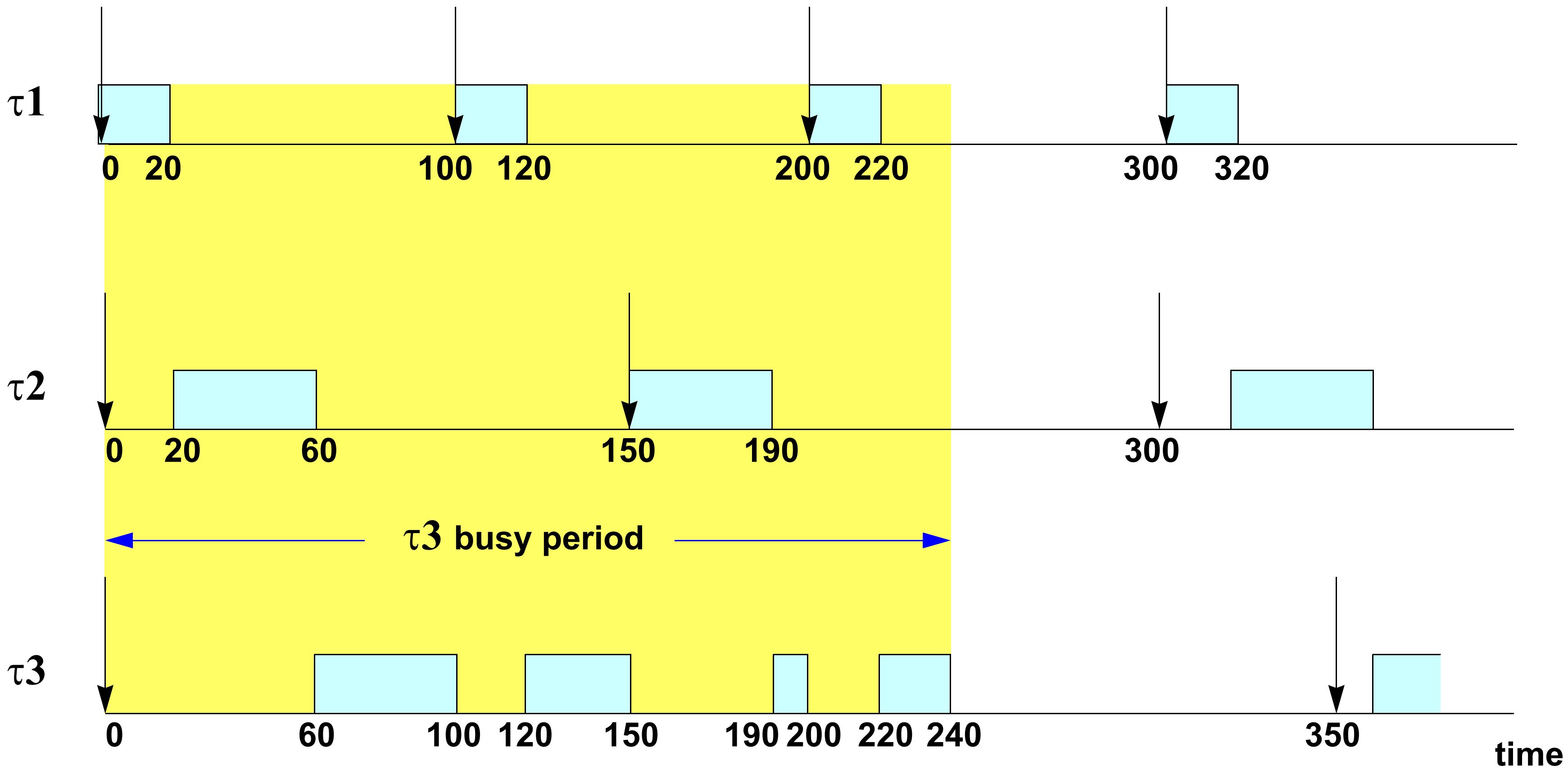
Notes:

The analysis problem that we have to solve is determining if a set of periodic independent tasks with deadlines coincident with the end of their periods is schedulable under a fixed-priority preemptive scheduler. To determine schedulability, one must determine that the deadlines will be met under all possible circumstances.

The basic RMA results for solving this problem are based on two very simple concepts, that help to build the worst-case condition for that set of tasks:

- *Critical instant*. The worst-case response time for any task in the task set is obtained when all tasks are activated at the same time, i.e., when the phase of each task is zero.
- *Checking the first deadline*. When all tasks are activated at the same time, if a task meets its first deadline, it will always meet all of its deadlines

Example of a Critical Instant



Notes:

The figure above shows how to interpret the critical instant results. It shows the execution sequence for the set of three tasks from our example. Task $\tau_1$has been assigned the highest priority, and preempts any other activity running in the system. Its response time is equal to its worst-case execution time (assuming no context switch overhead). Task τ_2 can sometimes be preempted by τ_1 and thus its worst-case response time is 60 time units, which is the response time to the first activation. Task τ_3 can be preempted both by τ_1 and τ_2. The figure shows that it is preempted multiple times before it completes its response to the first activation, at *t*=240 time units; this represents its worst-case response time.

The critical-instant and checking-the-first-deadline concepts show a very simple way to construct the worst case that can ever happen for a given task. It corresponds to the first activation, when all the other tasks are activated at the same time. If the task set meets its deadlines for the worst case, then it will always meet all deadlines. Based upon the proofs of these concepts, several analytical results have been developed that help in analyzing a system composed of periodic tasks; these results are discussed in the following slides.

2.3 Rate Monotonic Priorities

For a set of tasks with the following characteristics:

- periodic,
- independent,
- deadlines = periods,
- fixed-priority preemptive scheduling,

the optimum priority assignment is called rate monotonic:

- tasks with smaller periods get higher priorities

Optimum means that if the task set is schedulable with a given fixed-priority assignment it is also schedulable with rate monotonic priorities

Notes:

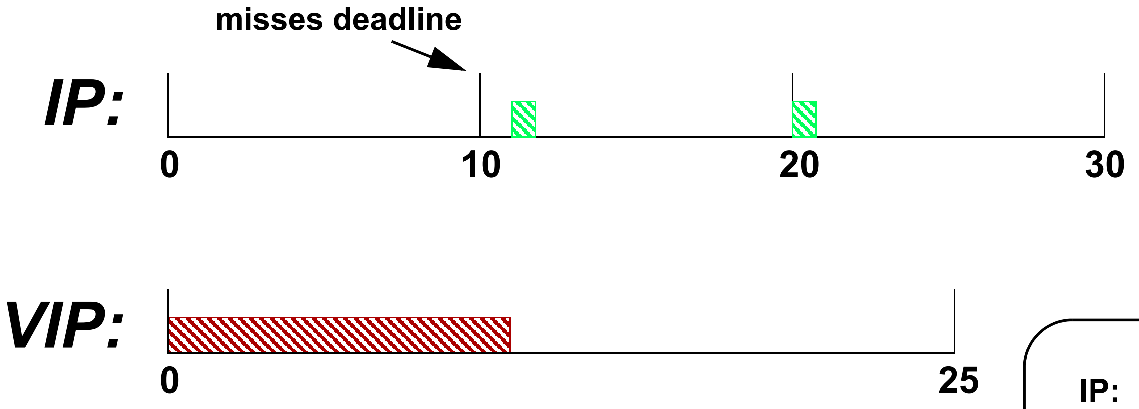
Rate Monotonic Scheduling is the name of a fixed-priority preemptive scheduling policy in which priorities are assigned ordered according to the task's rate: tasks with smaller periods get higher priorities.

Rate Monotonic priorities are optimum for sets of periodic independent tasks, when deadlines coincide with the end of the periods. Optimality means that if the task set is schedulable under any fixed priority arrangement, so it is under rate monotonic scheduling.

Notice that under the restrictions mentioned above it is not appropriate to assign priorities based upon semantic importance, but it is better to use the optimum priority assignment.

Example of Priority Assignment

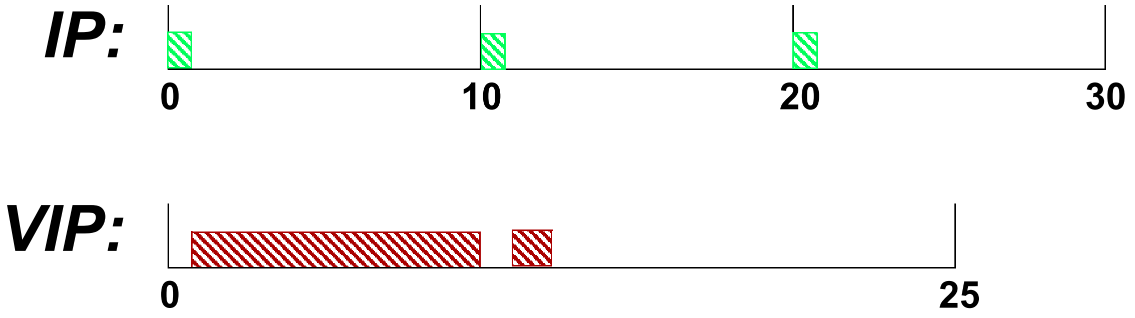
Semantic-Based Priority Assignment



IP: $U_{IP} = \frac{1}{10} = 0.10$

VIP: $U_{VIP} = \frac{11}{25} = 0.44$

Policy-Based Priority Assignment



Notes:

This example provides some motivation for using the rate monotonic scheduling algorithm. In the example we have 2 tasks, named IP and VIP, with the following descriptions:

Task IP (Important Task): C_{IP} = 1; T_{IP} = 10; U_{IP} = 0.10

Task VIP (Very Important Task): C_{VIP} = 11; T_{VIP} = 25; U_{VIP} = 0.44

Total utilization: 54%

If we assign higher priority to task VIP, then task IP will miss its deadline, even though the total utilization is only 54%. But if priorities are assigned according to the rate monotonic algorithm, then both tasks will meet their deadlines. Notice that, as VIP's period increases, total utilization decreases to 10%, since U_{VIP} approaches zero for large T_{VIP}; yet deadlines are still missed.

As an aside, note that many other scheduling policies (such as earliest deadline first, or least laxity first) result in the same priority assignment and the same schedule. The point of this example is that there a potential drawbacks to assigning priorities on the basis of semantic importance (degree of application criticality).

2.4 Utilization Bound Test

Utilization Bound Test: **A set of n independent periodic tasks, with deadlines at the end of the periods, scheduled by the rate monotonic algorithm will always meet its deadlines, for all task phasings, if**

$$\frac{C_1}{T_1} + \ldots + \frac{C_n}{T_n} \leq U(n) = n(2^{1/n} - 1)$$

U(1) = 1.0
U(2) = 0.828
U(3) = 0.779
U(4) = 0.756
U(5) = 0.743
U(6) = 0.734
U(7) = 0.728
U(8) = 0.724
U(∞) = 0.693

For harmonic task sets, the utilization bound is *U(n)=1.00* for all *n*.

Notes:

We say a set of tasks is ***schedulable*** if the task set is guaranteed to meet all its deadlines. The utilization bound test says that a task set is schedulable if its total utilization is less than a certain bound. The utilization bound test provides the basic formula underlying the theory. More complex formulas provide better bounds. This test is the application of a theorem, which was proved by Liu and Layland in “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*”, JACM*, 1973.

The utilization bound test assumes that:

1. The processor always executes the highest priority ready task (fixed-priority preemptive).
2. Task priorities are assigned according to rate monotonic policy.
3. Tasks do not synchronize with each other.
4. Each task’s deadline is at the end of its period.
5. Tasks do not suspend themselves in the middle of computations.

The test takes into account the preemption time and execution time for each task under a worst-case combination of periods and execution times. As the number of tasks goes to infinity, the bound approaches ln(2) = 0.693; in other words, any number of independent periodic tasks will meet their deadlines if the total system utilization is under 69%.

For a harmonic task set (in which the period of each task is a multiple of all higher-frequency tasks), the utilization bound is 1.0 for all task sets.

Sample Problem: Applying UB Test

	C	T	U
Task τ_1:	20	100	0.200
Task τ_2:	40	150	0.267
Task τ_3:	100	350	0.286

Total utilization is 0.200 + 0.267 + 0.286 = 0.753 < U(3) = 0.779

The periodic tasks in the sample problem are schedulable according to the UB test.

Notes:

Since we are interested in worst-case behavior, tasks utilizations are always rounded up. For example, U_2 = 40/150 = 0.2666... becomes 0.267 and U_3 = 100/350 = 0.2857143... becomes 0.286.

In this example, since the total utilization is under the bound for three tasks, all tasks are guaranteed to meet their deadlines, even under worst-case conditions. An additional 24.7% CPU capacity is available for lower-priority tasks that have no deadlines.

Remember that there are several assumptions associated with the utilization bound test:

- Zero context switch overhead.
- Deadlines are at end of period.
- No interrupts are used.
- Priorities are assigned in rate monotonic order.
- Tasks do not interact with one another.
- Tasks do not suspend themselves.

2.5 Response Time Analysis
Toward a More Precise Test

UB test has three possible outcomes.

$$0 \leq U \leq U(n) \Rightarrow Success$$

$$U(n) < U < 1{,}00 \Rightarrow Inconclusive$$

$$1{,}00 < U \Rightarrow Overload$$

UB test is conservative.

A more precise test can be applied.

Notes:

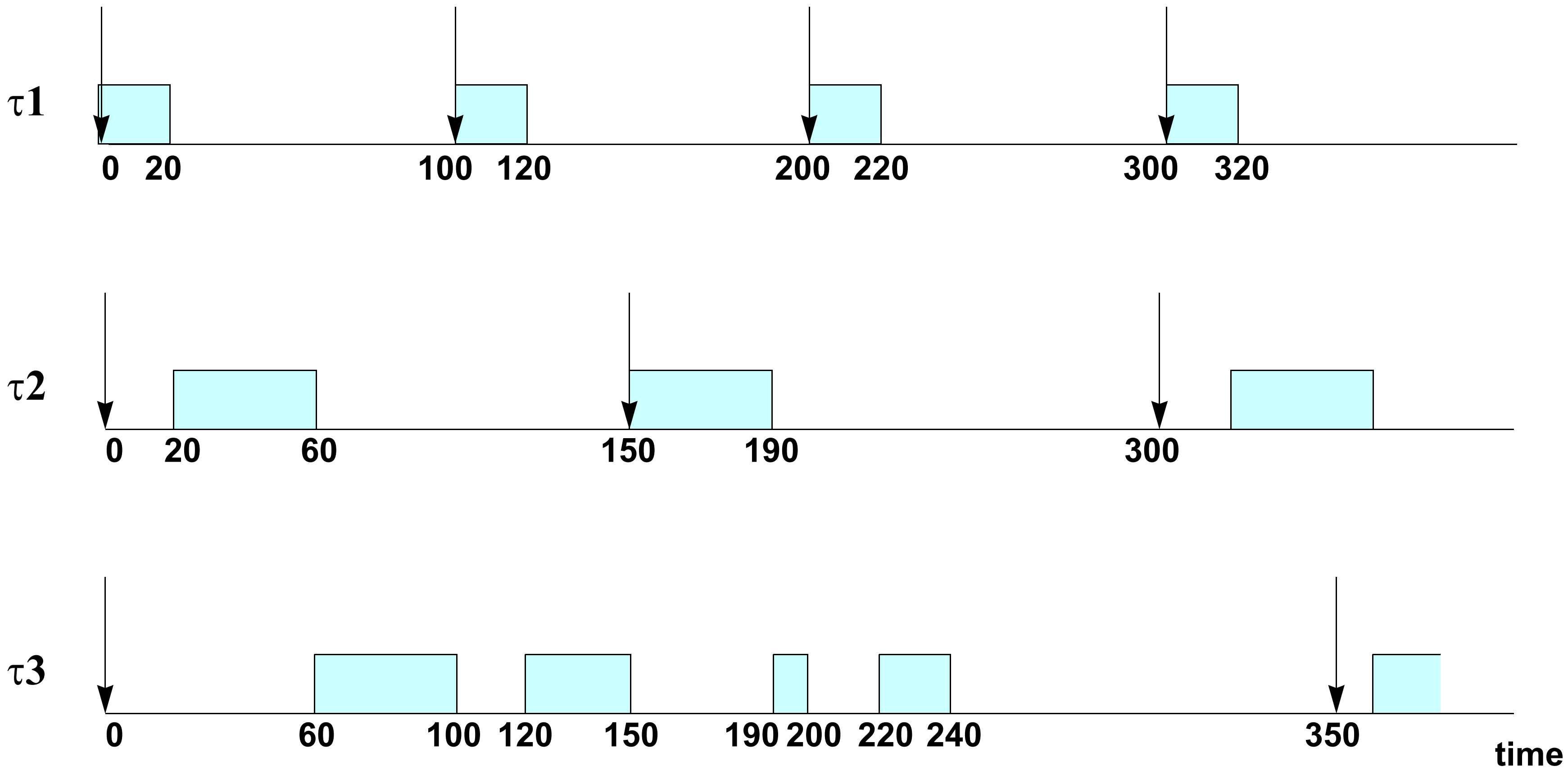
The results of applying the UB test can be:

$0 < U \le U(n)$	**success! task set is schedulable**
$U(n) < U < 1.0$	**inconclusive, task set may or may not be schedulable**
$1.0 < U$	**overload, task set exceeds capacity**

When the UB test is inconclusive, a more precise test can be applied.

The previous schedulability test (utilization bound test) is very conservative. Schedulable system utilization can often exceed 90%. In fact, for tasks with harmonic periods, where each period is evenly divisible into all longer periods (e.g. 100 ms, 200 ms, 800 ms, 1600 ms), the utilization bound is 100%.

Timeline for Sample Problem



Notes:

This is the timeline for the periodic tasks of the sample problem (shown on the previous page), in which tasks are lined up in worst-case phasing (i.e. all task are ready to execute at time t=0).

Timelines show one possible execution schedule and provide a graphical view of schedule analysis. We will draw timelines according to the following conventions:

- Tasks are arranged and numbered in rate monotonic order, highest frequency at the top.
- We assume Liu and Layland “worst-case” phasing, where all tasks start at time t=0.
- Execution time for τ_1 is plotted on its line.
- Execution time for τ_2 is then plotted on its line, accommodating preemption from τ_1’s execution; then this process is repeated for remaining tasks.
- If any task is preempted, its execution time block is divided with a hole in the middle representing the preemption.

Response time analysis (Harter, 1984; Joseph and Pandya, 1986)

For a set of *n* periodic independent tasks, with any fixed priority assignment we can apply the *Response Time Analysis (RTA):*

- Number the tasks according to priority (highest priority=τ_1, lowest priority=τ_n)
- Under a critical instant condition, the amount of work $W_i(t)$ of priority P_i or higher started before t is:

$$W_i(t) = \left\lceil \frac{t}{T_1} \right\rceil C_1 + \ldots + \left\lceil \frac{t}{T_{i-1}} \right\rceil C_{i-1} + C_i$$

Notes:

The critical-instant and check-the-first-deadline principles allow us to make an exact calculation of the worst-case response time of any given task in a periodic independent task set, independently of the priority assignment used.

To perform this calculation, we first provide an equation that allows us to obtain, at any given time t, the amount of work of priority P_i or higher that has been initiated in the system. Time t=0 corresponds to the start or the critical instant. The amount of work initiated, $W_i(t)$, is calculated by multiplying the number of activations of each applicable task, by the respective execution times.

The number of activations is obtained using a ceiling function $\lceil x \rceil$, which means rounding x to the next integer.

The important issue about this equation is that task τ_i will complete its execution when there is no pending work of priority P_i or higher, which is the same as finding the first instant t at which $W_i(t) = t$, i.e., all work has been completed. This instant can be found by using the iterative method shown in the next slide.

Response time analysis (Harter, 1984; Joseph and Pandya, 1986)

Iterative test (pseudopolynomial time):

$$a_0 = C_1 + C_2 + \ldots + C_i$$

$$a_{k+1} = W_i(a_k) = \left\lceil \frac{a_k}{T_1} \right\rceil C_1 + \ldots + \left\lceil \frac{a_k}{T_{i-1}} \right\rceil C_{i-1} + C_i$$

preemption

execution

Finish when two consecutive results are the same

Response time analysis (cont'd)

- The iteration stops when

$$a_{k+1} = a_k = R_i$$

- Task τ_i is schedulable if:

$$R_i \le D_i$$

For randomly generated task sets, the exact average utilization bound is 88%

Notes:

The response time R_i of task τ_i can be obtained by using the iterative equation that appears above. Each iteration consists of calculating the amount of work $W_i(t)$ activated at the time of the previous iteration. The iteration stops when two successive steps yield the same result, which means that the amount of work initiated until that time is already complete. If the utilization is less than 100%, the iteration is guaranteed to be finite.

The method is called the response time analysis (RTA), or the exact test, because it allows obtaining the exact worst-case response time of a given task. It can be applied by hand for small task sets, but can also be easily implemented in a computer. Besides, it works for any fixed-priority assignment, not only rate monotonic priorities, like the utilization bounds test. The iterative formula for task response time was introduced by Harter in 1984, and later by Joseph and Pandya, “Finding Response Times in a Real-Time System,” *BCS Computing Journal*, October 1986 (vol 29 no 5) pp 390-395.

The paper *The Rate Monotonic Scheduling Algorithm -- Exact Characterization and Average Case Behavior*, by Lehoczky, Sha, and Ding, Technical Report, Department of Statistics, Carnegie Mellon University, 1987 presents the fact that for randomly generated task sets, the actual utilization bound is 88% rather than the worst-case bound of 69% presented in the UB test.

Example: Applying RTA-1

Taking the sample problem, we increase the compute time of τ_1 from 20 to 40; is the task set still schedulable?

Utilization of first two tasks: 0.667 < U(2) = 0.828

- first two tasks are schedulable by utilization bound test

Utilization of all three tasks: 0.953 > U(3) = 0.779

- utilization bound test is inconclusive
- need to apply response time test

Notes:

The following data is the same as the sample problem. However, we have changed C_1 from 20 to 40 to illustrate the use of the RTA test.

Task τ_1: $C_1 = 40$; $T_1 = 100$; $U_1 = 0.4$

Task τ_2: $C_2 = 40$; $T_2 = 150$; $U_2 = 0.267$

Task τ_3: $C_3 = 100$; $T_3 = 350$; $U_3 = 0.286$

Utilization of first two tasks: 0.4 + 0.267 = 0.667 < U(2) = 0.828
Total utilization: 0.4 + 0.267 + 0.286 = 0.953 > U(3) = 0.779

When we change C_1 from 20 to 40, CPU utilization changes. Since the utilization of the first two tasks is under the utilization bound of the UB test, these tasks will always meet their deadlines. When the third task is considered, the total utilization is above the general bound, so we must look further to see if the third task can nonetheless meet its deadline.

We will now apply the response time test to task τ_3.

Example: Applying RTA-2

Use RTA test to determine if τ_3 meets its first deadline

$$a_0 = \sum_{j \le 3} C_j = C_1 + C_2 + C_3 = 180$$

$$a_1 = \sum_{j < 3} \left\lceil \frac{180}{T_j} \right\rceil C_j + C_3$$

$$= \left\lceil \frac{180}{100} \right\rceil (40) + \left\lceil \frac{180}{150} \right\rceil (40) + 100 = 260$$

$$a_2 = \left\lceil \frac{260}{100} \right\rceil (40) + \left\lceil \frac{260}{150} \right\rceil (40) + 100 = 300$$

Notes:

To apply the RTA test, we do two steps

1. Use the iterative formula to compute the response time R_3 of τ_3.
2. Compare R_3 to the first deadline for τ_3, namely T_3.

If $R_3 < T_3$, then τ_3 is schedulable according to the RTA test.

We start with an initial guess $a_0 = C_1 + C_2 + C_3$. This is labeled as the “zero-th” iteration. To compute the first iteration, a_1, we use the iteration formula:

$$a_1 = \sum_{j<3} \left\lceil \frac{a_0}{T_j} \right\rceil C_j + C_3$$

So the first iteration is
$a_1 = 180$.

Next we apply the formula to compute, a_2. Since each successive iteration is different, the computation has not yet converged and we must continue.

Example: Applying RTA-3

$$a_2 = 300$$

$$a_3 = \left\lceil \frac{300}{100} \right\rceil (40) + \left\lceil \frac{300}{150} \right\rceil (40) + 100 = 300 \Rightarrow \mathit{Done}!$$

$$R_3 = 300 < T_3 = 350$$

Task τ_3 is schedulable using the RTA test.

Notes:

We continue by applying the iteration formula for the next steps

Finally, the computed value for a_3 is the same as a_2 (300). The iteration has converged, and the worst-case response time for τ_3 is

$R_3 = 300.$

This must be compared to the first deadline for τ_3.

We have $R_3 < T_3$ since 300 < 350.

So τ_3 is schedulable using the response time test.

Notice that the result is the same obtained when we sketched the timeline for the three tasks. In practice, the response time test is a numerical way to perform the graphical check represented by the timeline.

A Summary at This Point

Utilization bound test is simple but conservative.

Response time test is more exact but also more complicated.

To this point, UB & RTA tests share the same limitations:

- all tasks run on a single processor
- all tasks periodic and non interacting
- deadlines always at the end of the period
- no interrupts
- zero context switch overhead
- tasks do not suspend themselves

In addition, the UB test has the following limitation:

- rate monotonic priorities assigned

2.6 Preperiod Deadlines: Priority Assignment

The critical-instant and checking-the-first-deadline concepts are applicable.

Rate Monotonic Priorities are no longer optimum

The Utilization Bounds Test must be modified

The response time test is applicable with no modification

Notes:

It is common in many real-time applications to set a task’s deadline to be prior to the end of its period, because a fast response is needed after an event arrives.

When some or all of the deadlines are before the end of the period, the two basic concepts of critical-instant and checking-the-first-deadline are still applicable. However, rate monotonic priorities are no longer optimum. We will discuss what the optimum priority assignment is in the next slides.

The utilization bounds test that was shown in is not applicable, but a different UB test exists for pre-period deadlines, which is also shown in a following slide.

Finally, the response test is completely applicable to the case with preperiod deadlines. This test is based on the critical instant concept and is applicable to any fixed-priority assignment. It allows obtaining the response time, which can then be checked against the actual deadline of each task.

Deadline Monotonic Priorities

For a set of periodic independent tasks, with deadlines within the period, the optimum priority assignment is the deadline monotonic assignment:

- **Priorities are assigned according to task deadlines.**
- **A task with a shorter deadline is assigned a higher priority**

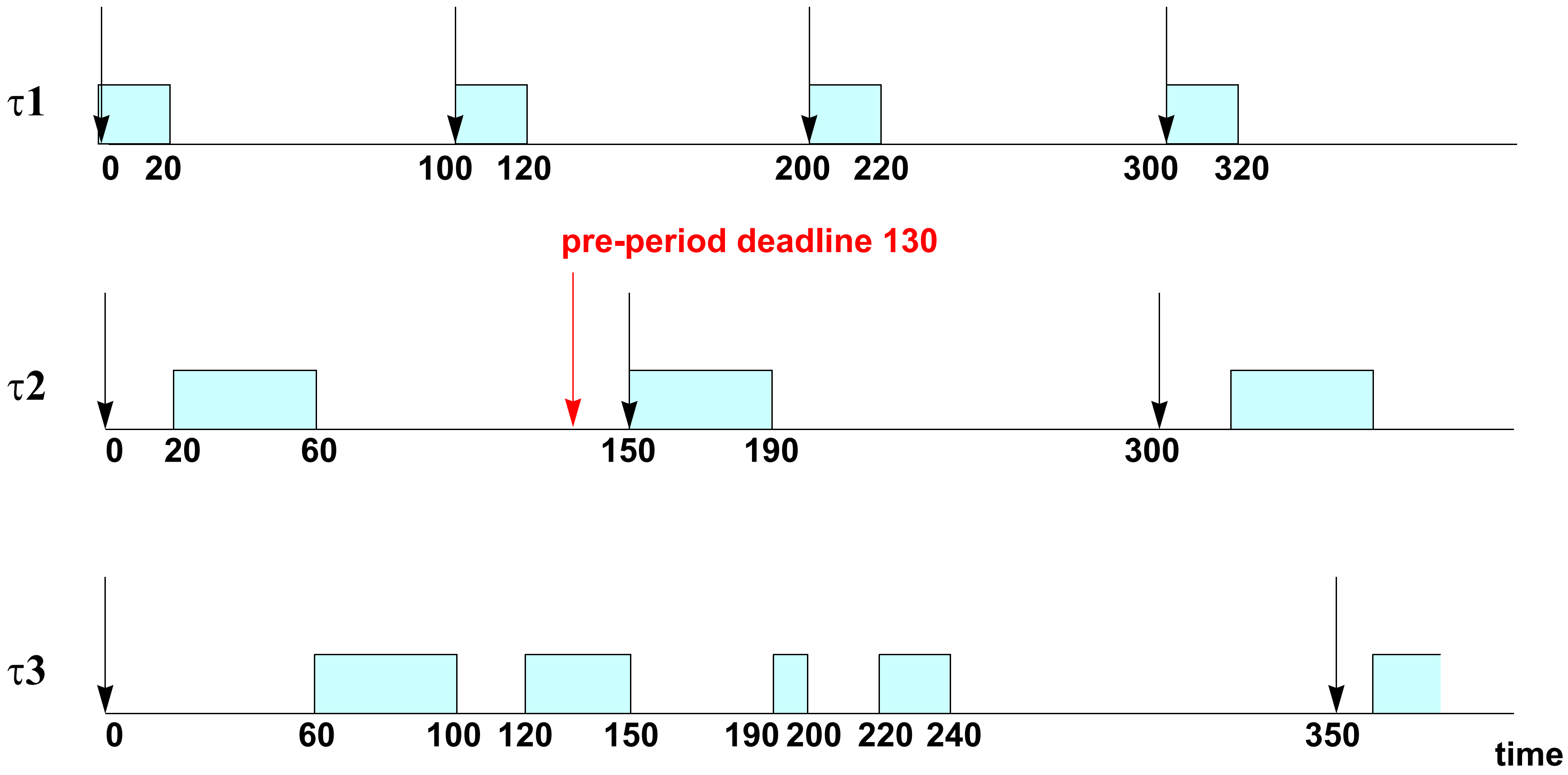
Rate Monotonic priorities are a special case of Deadline Monotonic Priorities

Notes:

Leung and Whitehead proved that for a set of periodic independent tasks with deadlines within the period, the optimum priority assignment is the deadline monotonic assignment, in which priorities are assigned according to task deadlines. The task with the shortest deadline is assigned the highest priority, the task with the next shortest deadline is assigned the next highest priority, and so on.

The rate monotonic priority assignment is a special case of the deadline monotonic assignment, because when deadlines coincide with the end of the periods there is no difference between the two priority assignments. Still, the theory is sometimes called “Rate Monotonic Analysis” in part of the literature, for historical reasons.

Sample problem with pre-period deadlines



Notes:

Response time analysis works for pre-period deadlines

- we just need to check the worst-case response time with the deadline

In the sample problem, task 2 has a deadline of 130, less than its period of 150

- the worst case response time is 60 in the timeline after a critical instant
- therefore 60<130 and task 2 is schedulable

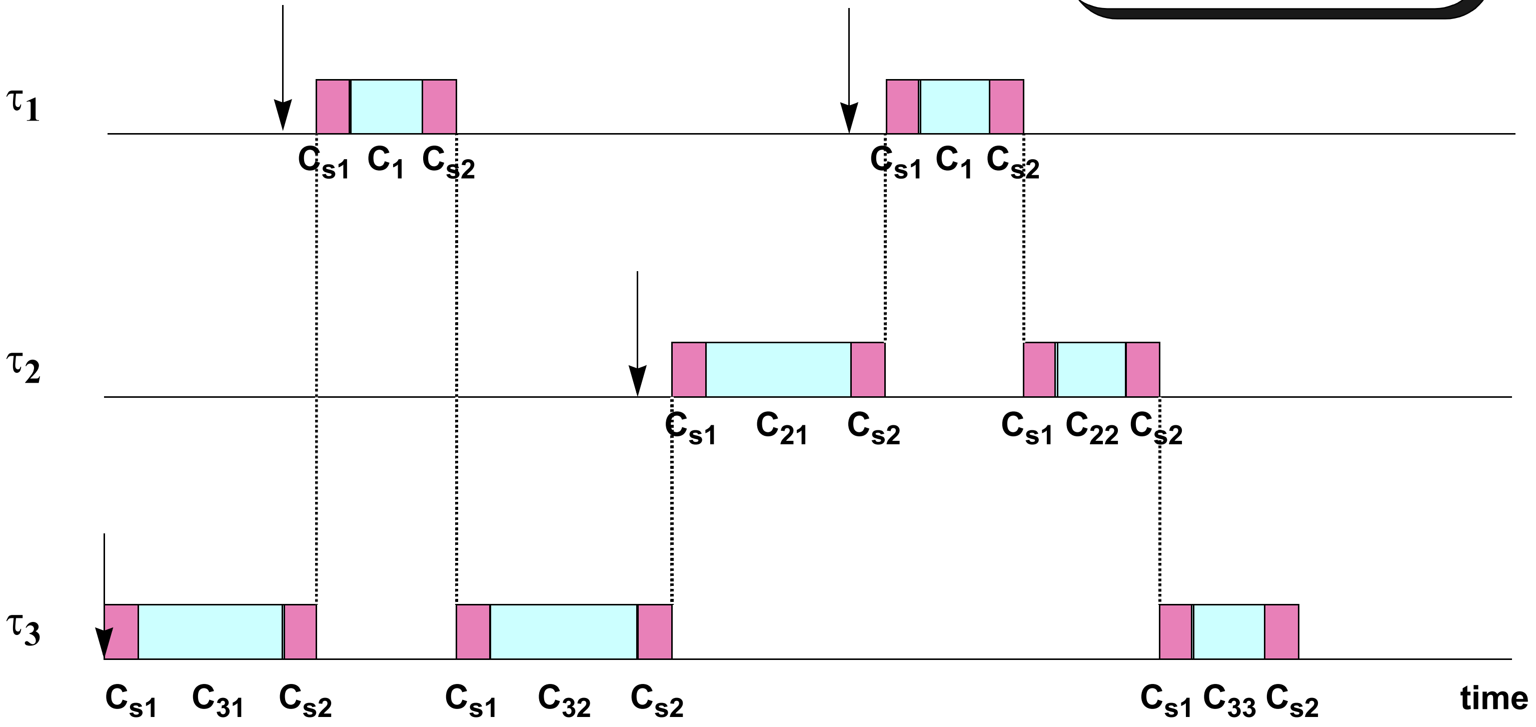
2.7. Modeling Task Switching

Two scheduling actions per preemptive task (preemption, and processor relinquishing)

$C_s = C_{s1} + C_{s2} \quad C_i \Rightarrow C_i + 2C_s$

Legend

- 🟦 Task Execution
- 🟪 Context Switch Activity



Notes:

When a task preempts a lower-priority task the execution state of the lower-priority task is saved and the execution state of the higher-priority task is established. When the higher-priority task completes its processing and relinquishes the CPU to a lower-priority task, its execution state is saved and the state of the lower-priority task is reestablished. We will assume that both kinds of context switches, preemption, and processor relinquishing, have the same duration, which we call C_s. C_s is composed of two simpler actions, saving the context of the previous task and restoring the context of the new task. These simpler actions have durations C_{s1} and C_{s2}, respectively, and so $C_s=C_{s1}+C_{s2}$.

In the figure above we can see that each time a lower priority task is preempted by a higher priority task, it is delayed by two context switches: one at the time of preemption, and one at the time the preempting task relinquishes the processor. Therefore, we model context switch time by adding two context switches ($2C_s$) to the execution time of each task.

Consider that task τ_3 is preempted by higher-priority tasks, τ_1 and τ_2. Worst case is that τ_3 will be preempted by τ_1 (and τ_2) every time τ_1 (or τ_2) is ready to run. Each preemption causes two context switches; one from τ_3 to run τ_1 and one from τ_1 back to resume τ_3. These two context switch times are charged to the preempting task.

2.8 Schedulability with Interrupts

Interrupt processing can be inconsistent with rate monotonic priority assignment:

- interrupt handler executes with high priority despite its period
- interrupt processing may delay execution of tasks with shorter periods

Effects of interrupt processing must be taken into account in schedulability model:

- The response time test works fine for arbitrary priorities
- The UB test must be modified

Notes:

Interrupt processing is common in real-time applications and often violates the optimum priority assignment. Typically, interrupt processing is higher in priority than application-level processing. So although it should be assigned a lower priority than an application-level task (i.e., when it has a longer period), it will have a higher *runtime* priority and will preempt the application-level tasks.

The response time test is capable of taking into account the effect of a non optimal priority assignment. In fact, the test works for any priority assignment.

However, the utilization bounds test must be modified to address the issue of non rate monotonic priorities. We will now explore how the basic rate monotonic schedulability model can be extended to account for interrupt processing.

To clarify some definitions;

- Runtime priority - the priority actually assigned to a task.
- Rate monotonic priority - the priority of a task, if assigned according to its rate.
- Runtime preemption - seizing of the CPU by a task with a higher runtime priority.

Example: Determining Schedulability with Interrupts

	C	T	U
Task τ_3:	60	200	0.300
Task τ_1:	20	100	0.200
Task τ_2:	40	150	0.267
Task τ_4:	40	350	0.115

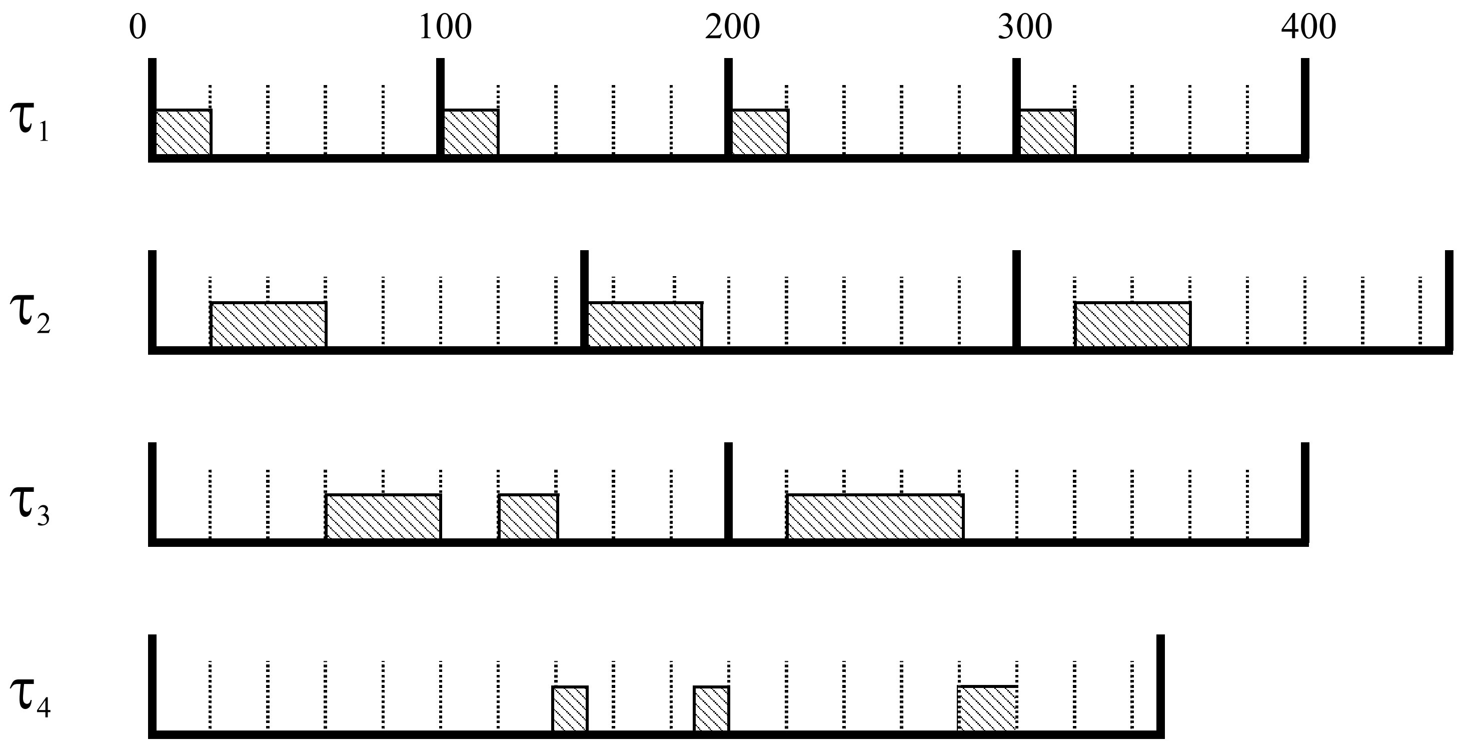
τ_3 is the interrupt handler

Notes:

We will illustrate the way we address a non optimal priority assignment with the above example.

Note that in this example, the interrupt should have a lower priority than tasks τ_1 and τ_2.

Example: Execution with Rate Monotonic Priorities

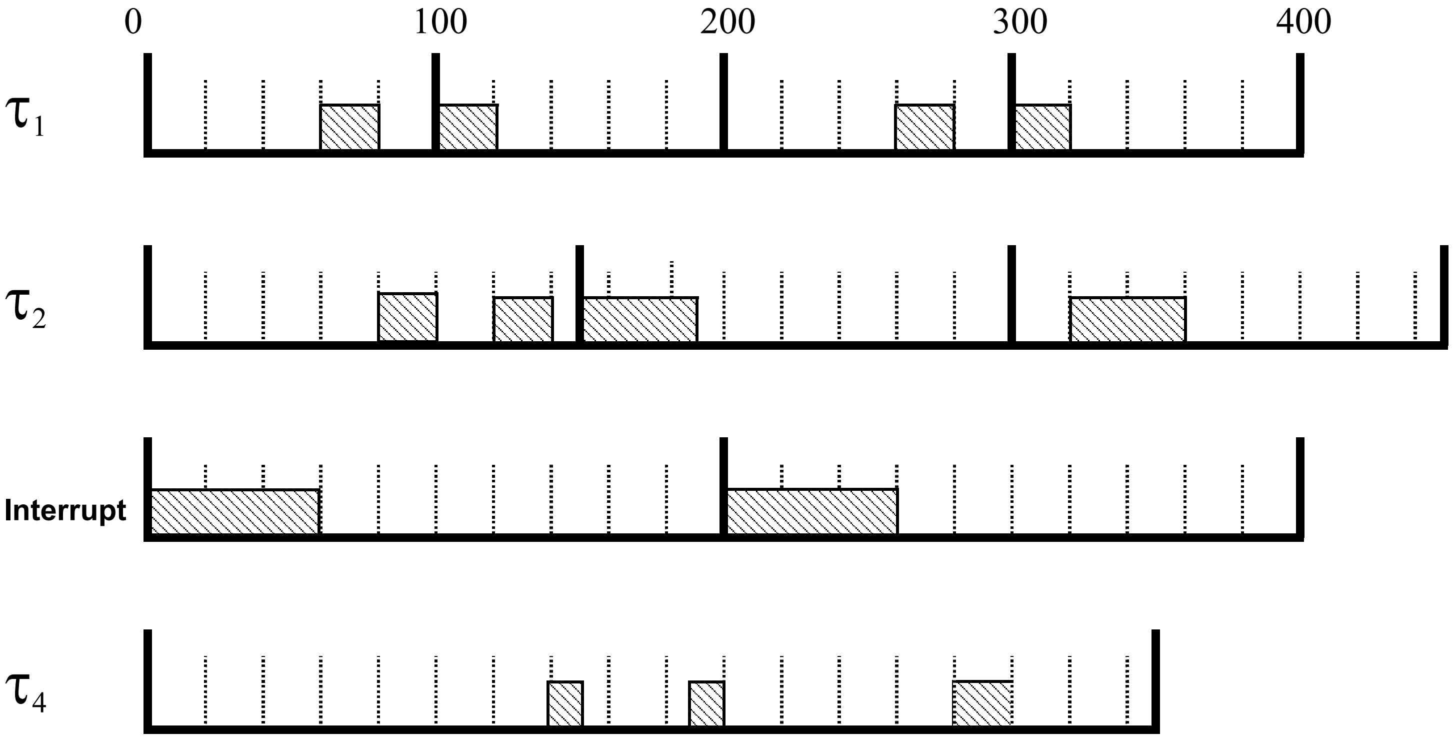


Notes:

This timeline shows how the tasks would execute under worst-case phasing **if** they ran in rate monotonic order. All tasks are clearly schedulable according to the “visual” RTA test.

Unfortunately, this timeline is **not** how the tasks execute. Since τ_3 is an interrupt handler, it executes before the other tasks, causing additional delays to τ_1 and τ_2.

Example: Execution with an Interrupt Priority



Notes:

This timeline shows the actual execution order, where τ_3 is an interrupt handler and executes with a runtime priority that is higher than the other three tasks, in spite of its longer period. Notice the impact that the interrupt has on the response time of the other three tasks. Both τ_1 and τ_2 are severely affected, being delayed by interrupt processing. However, τ_4 is not affected.

Execution is clearly inconsistent with the rate monotonic scheduling policy.

UB Test for Arbitrary Priorities

For the analysis of τ_i we can use the following sufficient test

We need to create the following two sets:

- ***H1***: singly preemptive tasks, i.e., those with periods ≥ ***D****i*
- ***Hn***: multiply preemptive tasks, i.e., those with periods < ***D****i*

The effective utilization for task τ_i is:

$$f_i = \sum_{j \in Hn} \frac{C_j}{T_j} + \sum_{k \in H1} \frac{C_k}{T_i} + \frac{C_i}{T_i}$$

The effective utilization is then checked against the utilization bound

Notes:

This utilization bound test can be applied to sets of periodic independent tasks with deadlines at the end of the period, and arbitrary priorities. When analyzing a given task $\tau_\mathbf{i}$, only that task and higher priority tasks need to be considered. The higher priority tasks are classified into two sets:

- The singly preemptive tasks set*, H1,* consists of higher priority tasks that preempt task $\tau_\mathbf{i}$ only once before its deadline at D_i.=T_i
- The multiply preemptive tasks set, *Hn,* consists of higher priority tasks that may preempt task $\tau_\mathbf{i}$ multiple times before the deadline at D_i.=T_i

This test treats event sequences in *H1* different than event sequences in *Hn*. The utilization term for tasks in *Hn* is C_j/T_j for each task $\tau_\mathbf{j}$ (the denominator is the period of $\tau_\mathbf{j}$). The utilization term for tasks in *H1* is C_k/T_i for each task $\tau_\mathbf{k}$ (the denominator is the period of $\tau_\mathbf{i}$).

The utilization bound used is the utilization bound for rate monotonic priorities:

$$U(n) = n(2^{1/n} - 1)$$

If the deadline is smaller than the period, the bound for pre-period deadlines must be used instead.

2.9 Non-Preemptible Sections: Priority Inversion

Delay to a task's execution caused by interference from lower-priority tasks is known as *priority inversion*.

Priority inversion is modeled by blocking time.

Identifying, modeling, and reducing sources of priority inversion is central to schedulability analysis.

Some sources of priority inversion:

- Non-preemptible regions of code
- FIFO (first-in-first-out) queues
- Synchronization and mutual exclusion

Notes:

Any delay to a task caused by a task with a lower priority is a priority inversion.

We model priority inversion by adding *blocking time* to the schedulable tests. By adding blocking time, we are taking into account the worst-case execution delays due to priority inversion.

Experience with RMA has shown priority inversion to be the most common cause for a task to miss its deadline.

There are many potential sources of priority inversion. This slide lists several sources of priority inversion:

- An example of priority inversion is when a higher-priority task is prevented from preempting a lower-priority task.
- Queues are also potential sources of priority inversion. If a high-priority task is waiting because a request that it made (such as an I/O call) is pending in a FIFO queue, the task may have to wait for lower-priority requests by lower-priority tasks to be satisfied.
- Synchronization caused by resource sharing also is a potential source of priority inversion. If a lower-priority task has reserved a resource that is needed by a higher-priority task, the higher-priority task is forced to wait.

Accounting for Priority Inversion

Recall that task schedulability is affected by:

- preemption by higher priority tasks
- self execution
- blocking: delays caused by lower priority tasks

The schedulability formulas are modified to add a “blocking” or “priority inversion” term to account for inversion effects.

Notes:

The schedulability tests assume that tasks are numbered in priority order. To determine whether a given task meets its deadline, one must consider three factors:

- preemption - execution time consumed by higher-priority tasks.
- execution time - from time consumed by the task itself.
- blocking - delays from execution by lower-priority tasks.

Example with Interrupt and Non-Preemptible Section

	C	T	B
Task τ_1:	20	100	10
Task τ_2:	40	150	10
Task τ_3:	60	200	10
Task τ_4:	40	350	0

Task τ_3 is an interrupt service routine

Task τ_4 has a non-preemptible non-interruptible section of 10 ms.

Notes:

In this example we have three regular tasks and an interrupt service routine that executes at the highest priority in the system. Besides, the lowest priority task has a non-preemptible and non-interruptible section of code whose worst-case execution time is 10 ms. If one of the higher priority tasks wants to execute when task τ_4 is in its non-preemptible section, the higher priority task gets delayed. This represents a priority inversion that must be taken into account in the analysis.

In this case, the blocking term B_i for the three higher priority tasks is 10 ms., because this is the worst-case delay caused by a lower priority task.

RTA Test with Blocking

In the presence of priority inversion, the RTA test for τ_i must include the blocking delay:

$$a_0 = C_1 + C_2 + \ldots + C_i$$

$$a_{k+1} = W_i(a_k) = \left\lceil \frac{a_k}{T_1} \right\rceil C_1 + \ldots + \left\lceil \frac{a_k}{T_{i-1}} \right\rceil C_{i-1} + C_i + B_i$$

preemption

execution

blocking

Notes:

The RTA test, like the UB test, may be modified to include blocking, by adding a blocking term to the formula.

In this case, no distinction is needed between multiply preemptive and singly preemptive tasks.

Example: RTA Test for τ_2

$$a_1 = \left\lceil \frac{a_0}{T_1} \right\rceil C_1 + \left\lceil \frac{a_0}{T_{int}} \right\rceil C_{int} + C_2 + B_2 = \left\lceil \frac{120}{100} \right\rceil 20 + \left\lceil \frac{120}{200} \right\rceil 60 + 40 + 10 = 150$$

$$a_2 = \left\lceil \frac{150}{100} \right\rceil 20 + \left\lceil \frac{150}{200} \right\rceil 60 + 40 + 10 = 150 \Rightarrow \text{Done}$$

Schedulable, since response time just meets the deadline

$$R_2 = 150 \le T_2 = 150$$

Example: RTA Test for τ_4

Response time for τ_4

$$a_1 = \boldsymbol{B}_4 + \boldsymbol{C}_4 + \sum_{j<4} \left\lceil \frac{a_0}{\boldsymbol{T}_j} \right\rceil \boldsymbol{C}_j = 0 + C_4 + \left\lceil \frac{160}{T_1} \right\rceil C_1 + \left\lceil \frac{160}{T_2} \right\rceil C_2 + \left\lceil \frac{160}{T_{int}} \right\rceil C_{int} = 220$$

$$a_2 = 40 + \left\lceil \frac{220}{100} \right\rceil 20 + \left\lceil \frac{220}{150} \right\rceil 40 + \left\lceil \frac{220}{200} \right\rceil 60 = 40 + (3)20 + (2)40 + (2)60 = 300$$

$$a_3 = 40 + \left\lceil \frac{300}{100} \right\rceil 20 + \left\lceil \frac{300}{150} \right\rceil 40 + \left\lceil \frac{300}{200} \right\rceil 60 = 40 + (3)20 + (2)40 + (2)60 = 300$$

The response time for τ_4 is R_4=300 which is less than the deadline T_4=350.

Therefore, τ_4 is schedulable using the RTA test.

Basic Theory: Where Are We?

We have shown how to handle:

- task context switching time: include $2C_s$ within C
- preperiod deadlines: RTA Test
- priority inversion: include as blocking B
- arbitrary priorities: RTA test or modified UB test

We still need to address:

- task interactions
- aperiodic tasks
- arbitrary deadlines
- multiprocessors
- operating system effects (other than context switch)