What Every Engineer Needs To Know About Rate-Monotonic Scheduling: A Tutorial

This paper presents the fundamentals of rate-monotonic scheduling theory for those who have had no former experience with it. It explains, with examples, the basic theorems and their extensions, including task synchronization and nonperiodic events. It also critically discusses the major shortcomings of this approach. The below article is reduced for the convenience of Real-Time Magazine. The full version can be obtained at IEEE Computer Society Press.

By Janusz Zalewski,
Professor,
Department of Computer Science,
Embry-Riddle Aeronaotical University.

Introduction

Rate-monotonic scheduling (RMS) theory has emerged in the context of task scheduling, where a finite number of periodic tasks share a single processor. RMS theory provides rules for analyzing whether or not a given set of tasks can be scheduled according to their timing characteristics. It is natural to realize that a situation in which multiple hardware modules (*boards*) share a backplane bus is logically quite similar to a Situation in which multiple tasks run on a single processor. Requests from both entities, tasks and modules, must be scheduled in order for them to obtain access to the resource they are competing for: a backplane bus or a processor, respectively.

Thus, the general results of RMS, derived for multitasking on a uniprocessor, are equally applicable to backplane bus systems – provided appropriate Substitutions of terms are made. This paper discusses the principles of rate–monotonic scheduling for multiple tasks executing on a single processor, assuming that the results can be mapped onto backplane bus Systems. In the presentation, emphasis is placed on basic concepts, and the assumption is that the reader has had no previous exposure to RMS.

Basic terms

In multiprocessor Systems where multiple processors share a single backplane, the bus protocol must allow selection of a unique bus owner for each bus transaction (bus *cycle*). There are many ways to give access to a bus in a consistent manner. The simplest, most common technique is to apply the first-come, first–served rule, granting a bus access to the processor module that requested it first. In cases when multiple requests occur at the same time (practically, within the same clock cycle), and their order cannot be distinguished, an additional policy must be applied to select a module to become a bus owner for the next bus transaction. This policy may imply random choice, selection based on physical proximity, or the use of priorities. In the latter case, that module is selected that has the highest priority level assigned to it.

However, once a module starts using a bus, it must finish its transaction. This approach has a clear disadvantage: Requests that come later but whose handling is more urgent must wait until after completion of the first request. Even if a module performing highly important activities is suddenly requesting bus access, it cannot be granted Permission to use a bus (that is, its request cannot be scheduled) until the current transaction (bus use) has completed. This problem is especially serious in case of applications with a bounded response time, known as *real–time systems*.

Scheduling tasks or scheduling bus access in real-time Systems is substantially different from traditional forms of scheduling. It is required that more urgent tasks be given priority in execution Overall othertasks that are eligible to run, not only when there is a need to select the next task to run when the tasks requested bus access at the same time, but also when one of the tasks is running and a new request is made by a task of higher priority. In contrast to the Standard meaning of *priority* (that is, the level of importance assigned to an item[1]), the term *priorityas* used here means the level of urgency assigned to a task. The Situation in which an urgent task removes the currently running taskfrom the resource is known as *preemption.*

Understanding *preemption* is quite crucial to understanding how one can shorten the time a system can react to external events. A system that requires a task to complete its execution before the next task can be scheduled to run can be much slower in responding than a system that allows a currently running task to be preempted, yielding the processor to a more–urgent task that requested access before the former task's completion. This Situation determines the basic property of real-time Systems – *responsiveness*- which is how fast, after the occurrence of a given event, the system can be expected to start handling that event.

In general, the *responsiveness* of a system can be defined as the property that is characterized by the worst-case time that elapses from the occurrence of a particular event to the start of its processing. This propetty defines how fast a system responds to events – that is, how fast it can perceive them -and is one component of a system's Overall reaction time.

In addition to responsiveness, timeliness – or how fast (that is, how timely) the events already perceived can be processed – is important. *Timeliness* is the property that is characterized by the worst-case time that is needed for processing of a perceived event.

Several questions must be answered regarding preemption via priorities to increase responsiveness. For example, if the highest–priority module can be granted bus access at any time-even being able to preempt the mod-

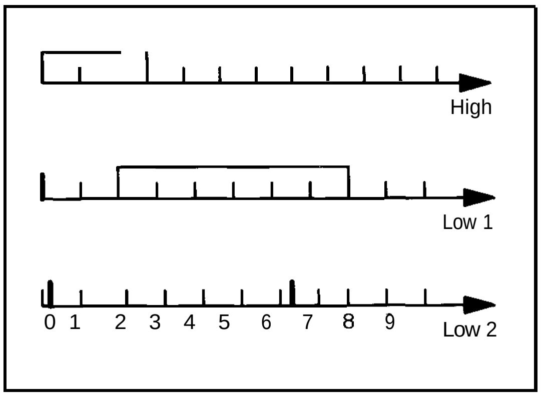


Figure 1: *Illustration of an insufficient number of priority levels.*

ule currently using a bus before its transaction has completed – there is an immediate question about fairness and starvation: How can we guarantee, if at all, that the lowest-priority modules will ever get bus access? Regarding timeliness, another question arises: If there are many simultaneous activities, how can we guarantee that all of them are completed on time? If not all can finish in a timely manner, which ones can and which ones cannot? In other words, there is a clear need to have a method of quantitative analysis of each module's (task's) behavior in terms of using the resource for which it is competing.

In summary, for a given set of tasks to be run on a single processor (or equivalently, for hardware modules competing for a backplane bus), it is necessary to ask two questions: Can the tasks run in a way that guarantees timely access to the resource for all of them? Can the tasks meet their execution deadlines? Stoyenko[2] gave the first formulation of a problem that corresponds to these two questions in terms of *schedulability*, which is the property of a set of tasks that ensures that the tasks all meet their deadlines. In the language of processor modules sharing a backplane bus, a set of bus requests made by these modules is *schedulable* if the requests are granted bus access and can complete bus transactions on time. (The definition of "on time" is predefined for each module.)

All the properties mentioned above ensure *predictability*, which is the property that is characterized by an upper bound on the Overall reaction time of each task in the system to external Stimuli. This upper bound must be known precisely in order to determine whether or not the task can meet its deadline.

RMS fundamentals: A matter of priorities

Is there a problem if priorities are not assigned at all or if multiple tasks are grouped on single priority levels because insufficient priority levels exist? Example 1 addresses this question.

Example 1

Consider three tasks and only two priority levels (one bit), as listed in Table 1. What would happen if all tasks arrived (that is, requested execution) at the same time?

Task	Priority	Execution time	Deadline
High	High (1)	2	10
Lowl	Low (0)	6	20
Low2	Low (0)	3	6

Table 1: *Illustration of insufficient priority levels.*

When Tasks High and Lowl arrive (that is, start their execution) slightly before Task Low2 (see time instant 0 in Figure 1), they must resume in this order, and there is no way, under current priority assignments, that Task Low2 can meet its deadline (marked at time instant 6, which is 6 units after this task's arrival), because it cannot preempt a task of higher or the same priority. This is because Tasks High and Lowl consume 2 + 6 = 8 time units, which is greater than Task Low2's deadline, which is 6.

Moreover, with only two priority levels, it may be difficult to find a priority assignment that solves the problem of meeting all deadlines under the worst arrival conditions. For instance, if Task Low2 is assigned the highest priority (level 1), and Tasks Lowl and High both have low priority (level 0), then starting Tasks Low2 and Lowl slightly before Task High effectively precludes Task High from meeting its deadline (simply because the execution time of Task Low2 + Task Lowl consumes 3 + 6 = 9 time units, and Task High cannot execute its 2 units before its deadline, which is 10 units, expires).

Assigning priority level 1 only to Tasks High and Low2 does solve the problem, even if all tasks run periodically, with periods equal to deadlines. Introducing an additional priority level can also help. This explains why assign– ing priorities is so impottant in real-time systems.

Knowing that the assignment of priorities to tasks (and modules) is crucial to providing better schedulability, the immediate next question is how to assign priorities – that is, on what basis? In other words, we should know the criterion of priority assignment. Lehoczky and Sha[3] gave an analysis of assigning priorities if insufficient priority levels exist. The technique they proposed is called a *constant–ratio priority grid*; it will not be described here.

If the number of priority levels is sufficient, a natural and straightforward way to assign priorities is to do so based on task importance. Example 2 gives an idea of why priorities based on task importance may not work.

Example 2

For the set of tasks in Table 2, which arrive approximately at the same time and run just once, if Task 1 comesfirst, it executesforafull6 units, effectively preventing (because of its top priority) the other two tasks from accessing the processor. Thus, with such priority assignments, the other tasks cannot meet their deadlines. However, if we change priority assignments, giving the lowest priority to the most important task, then all tasks meet deadlines without hurting the important one.

Task i	Priority (importance)	Execution time C_i	Deadline T_i
1	High	6	10
2	Medium	2	5
3	Low	2	4

***Table** 2:* *Illustration* **of** *priority assignment based on task importance (criticality).*

Rate-monotonic scheduling theoty offers a criterion for priority assignment to mutually independent tasks running periodically. The basic principle of RMS can be expressed as follows:

The shorter the task's period, the higher its priority.

Hence comes the name *rate-monotonic scheduling* for this algorithm, because it assigns priorities to tasks as a monotonicfunction of their *rates (frequencies).*

For a set of modules accessing a shared bus, the above principle means that they must be assigned different priorities, based on their prospective frequencies of accessing the bus (making bus requests). This principle, at least in its current formulation, does not seem to be useful in practice for backplane bus systems (at least, it is not known if the principle has been explicitlyapplied ever before). However, it does have far-reaching implications, which we see in subsequent sections.

Module name	Execution time (milliseconds)	*Period (milliseconds)*	Bus utilization (execution time/period)
A	5	10	1/2
B	5	20	1/4
C	10	30	1/3

Table 3: *Example of a poorly designed sys tem.*

A crucial but simple concept that forms the basis of RMS theory is that of *processor utilization* (or *bus utilization*), which is expressed by a Parameter called the *utilization factor.* The *utilization factoris* the ratio of the time a resource (processor or bus) is used to the Overall time this resource is available. It can be calculated individually for each entity competing for a resource or cumulatively as a sum of all individual utilizations. However, it should be noted that the concept of a *utilization factor* is not universal, and a system with avery low utilization factor may not guarantee schedulability, while tasks in another system with a utilization factor equal to 1 – that is, a system in which the utilization achieves 100 percent – may still be schedulable. More information on this subject is given in the section entitled "Alternative solutions."

The most visible example of a system that is poorly designed in this respect is one in which bus utilization is greater than 1 (see Table 3). Such a system reaches Saturation, and not every module can be granted bus access as requested. So, the bus utilization must be less than or equal to 1 to guarantee timely access to the bus for each processor. This Statement seems trivial but may be quite important from a backplane bus designer's or user's point of view. It may help to answer the following question: Why, for theoretical bus speeds of as high as tens of megabytes (Mbytes) per second, can we achieve only several Mbytes per second of actual speed; that is, why can we achieve a bus bandwidth utilization factor of only 0.10 to 0.25?

The above-stated requirement- that the utilization factor be less than or equal to 1 – is necessary. But is it also sufficient to ensure schedulability? This issue is discussed in this section and in the next section. For priorities based on task frequencies, a nontrivial sufficient condition for schedulability by the rate–monotonic scheduling algorithm was introduced in 1973.4

***RMS* Theorem 1**

For a set of N independent periodic tasks, where C_i and $T_i = 1, 2, \ldots$ N, are execution time and period, respectively, and assuming that task deadline equals task period, the tasks are schedulable by RMS if the following condition holds:

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

This theorem simply states that if the above condition is met, then all tasks will complete on time (by their deadline, which is equal to their period). In terms of N processors accessing a shared bus, this means that if their bus request is periodic and their operations are mutually independent (that is, they do not communicate), provided deadline is equal to period, it is sufficient to check that the above inequality holds in order to guarantee schedulability using static priority assignments.

The interpretation of this theorem is that modules' requests can be scheduled by RMS if their cumulative utilization factor is less than a cettain *upperbound* $N(2^{1/N}-1)$ which is generally much less than full bus Saturation. The value of $N(2^{1/N}-1)$ converges to In 2 $\approx$ 0.693 as the number N increases to infinity. Similarly, if the sum of utilizations is less than a certain upper bound, then all individual tasks (the whole task set) can be scheduled by RMS with guaranteed meeting of all their deadlines (provided appropriate assumptions are met). Examples 3 and 4 illustrate the meaning of Theorem 1.

Example 3

Table 4 presents a set of six tasks characterized by their execution time C_i and their period T_i. According to Theorem 1, a cumulative CPU utilization factor – the sum of C_i/T_i must be smaller than the upper bound $N(2^{1/N}$ -7) to guarantee schedulability by RMS. As we see from the table, the set of the first five tasks (or less) is schedulable by the RMS algorithm, because 0.64 < 0.743. However, the set of all six tasks may not be schedulable by the RMS algorithm, because 0.74 > 0.735. In fact, a set of any five tasks from the set of six in Table 4 is schedulable according to Theorem 1.

Task i	Execution time C_i	Period T_i	Ratio C_i/T_i	Utilization *(1...N)*	Upperbound
1	32	160	0.20	0.20	1.000
2	50	200	0.25	0.45	0.828
3	10	250	0.04	0.49	0.779
4	15	300	0.05	0.54	0.756
5	40	400	0.10	0.64	0.743
6	50	500	0.10	0.74	0.735

Table 4: *Characteristics of the six tasks discussed in Example 3.*

Example 4

In Table 5, we have another task set, consisting of eight tasks. The first seven tasks are schedulable according to Theorem 1; however, adding Task 8 causes a problem. It is unimportant to this Observation that the tasks in the table are not ordered according to their periods. If we look closer at the table, we see that the largest contributions to the cumulative CPU utilization factor (0.250 and 0.125) are from Tasks 6 and 8, respectively. A designer can then check on the possibility of shortening the execution time of Task 6 or lengthening the period of Task 8 to make them fit into the scheme and thus eliminate the problem. For instance, extending the period of Task 8 to T_8 = 100 would be a Solution (because 0.723 < 0.724).

Task i	Execution time C_i	Period T_i	Ratio C_i/T_i	Utilization *(1...N)*	Upperbound
1	10	500	0.020	0.020	1.000
2	15	300	0.050	0.070	0.828
3	12	100	0.120	0.190	0.779
4	5	150	0.033	0.223	0.756
5	20	200	0.100	0.323	0.743
6	50	200	0.250	0.573	0.735
7	25	250	0.100	0.673	0.729
8	5	40	0.125	0.798	0.724

Table 5: *Characteristics of the eight tasks discussed in Example 4.*

In the two above examples, it can be seen that Theorem 1 does not hold. Exactly what does this mean? Does it mean that the tasks are not schedulable on a processor (or that the processor modules are not schedulable to get access to the bus)? Strictly speaking, Theorem 1 does not define a necessary condition, so if the inequality from this theorem does not hold, then we cannot say whether or not a set of tasks is schedulable. Some additional criterion is needed to determine schedulability in such a case.

Sufficient and necessary condition for RMS schedulability

What condition is sufficient and necessary for RMS schedulability? This question has been answered by a stronger result of the RMS theory.[5] What follows is a sufficient and necessary condition for a set of tasks to be schedulable.

RMS Theorem 2

For a set of N independent periodic tasks, where C_i and $T_i, i = 1, 2, \ldots N$, are their execution time and period, respectively, and assuming that task deadline equals task period, the tasks are schedulable by RMS (that is, they will always meet their deadline) if and only if the following conditions hold:

$$\forall_i \min \sum_{j=1}^{i} \frac{C_j}{lT_k} \lceil lT_k / T_j \rceil \leq 1$$

where min is calculated over $(k,l) \in W_i$, and

$$W_i = \{(k,l) | 1 \leq k \leq i,\ l = 1, \ldots, \lfloor T_i/T_k \rfloor\}$$

The notations $\lfloor$*expression*$\rfloor$ and $\lceil$*expression*$\rceil$ denote the largest integer smaller than or equal to *expression* and the smallest integer larger than or equal to *expression*- that is, the floor and ceiling functions – respectively.

Task i	Execution time C_i	Period T_i	Ratio C_i/T_i	Utilization *(1...N)*	Upper-bound
1	**45**	**135**	**0.333**	**0.333**	1.000
2	50	150	0.333	0.667	0.828
3	80	360	0.222	0.889	0.779

Table 6: *Characteristics of the three tasks discussed in Example 5.*

Example 5

To see how Theorem 2 can be applied, let us analyze the schedulability of the tasks presented in Table 6. Clearly, the three tasks are not schedulable according to RMS Theorem 1, because the cumulative CPU utilization factor for the three tasks is greater than the upper bound (0.889 > 0.779).

To check the schedulability of the three tasks from Table 6, we need to use the equation of Theorem 2 for $i = 3$ only, because the first two tasks are schedulable according to Theorem 1. Thus, one of the four indices in this equation is fixed (i = 3). In the calculation of respective sums, the other index,j, will vary from 1 through $i = 3$. Knowing the values of i and j, we need to focus on the two other indices, kand l. Index k must vary from 1 through $i = 3$, and for each k, index l must vary from 1 through $\lceil T_i/T_k \rceil$. Thus, values of all four indices are given as follows:

i=3
j=1 3
k=1 . . 3
l=1 . . . $\lceil T_i/T_k \rceil$

Hencefotth, the remaining calculations are simple, though very tedious.

k =1: $\lfloor T_i/T_k \rfloor = \lfloor 360/135 \rfloor = 2$; thus, l = 1, . . . 2
k=2: $\lfloor T_i/T_k \rfloor = \lfloor 360/150 \rfloor = 2$; thus, l = 1, ... 2
k=3: $\lfloor T_i/T_k \rfloor = \lfloor 360/360 \rfloor = 1$; thus, l = 1.

After some calculations you can find the following inequalities:

	***l* =1**	***l* = 2**
***k* = 1**	> 1	≤ 1
***k* = 2**	> 1	> 1
***k* = 3**	> 1	

From the above , it stems that, for k= 1 and l = 2, the condition of Theorem 2 holds. Thus, the set of three tasks from Table 6 is schedulable by the RMS algorithm.

Unfortunately, the condition of Theorem 2 is hard to read and cumbersome to apply. Therefore, we now present another interpretation of Theorem 2 that is easier to understand.[5] The theorem simpiy states that to determine schedulability, it is sufficient and necessary to check if each task can complete its execution before its first deadline. Technically, to do this checking, a number of inequalities must be checked for all possible scheduling points for each task involved, where the scheduling points are all time instants at which any task period ends. This checking must be done for all scheduling points that fall into the first period of each task, because it is guaranteed that if a task meets its first deadline, it will always meet all other deadlines.

The following rule of thumb can be given to simplify the schedulability check by RMS:

- Step 1. Apply Theorem 1 and stop if all individual conditions are met. If not, apply Theorem 2 for all doubtful cases, as in the next steps (Steps 2a – 2c).
- Step **2a.** Determine all schedulability points by marking on a time axis all successive periods for all tasks in question, from time 0 up to the end of the first period of the lowest-frequency task.
- Step 2b. For each time instant marked in Step 2a – that is, for all schedulability points – construct an inequality that has, on its left-hand side, a sum of all possible execution times of all tasks that can be activated (possibly multiple times) before this schédulability point and, on its right-hand side, only the value of time corresponding to this schedulability point.
- Step **2c.** Check if values on the left-hand sides are smaller than or equal to their corresponding right-hand-side values. If at least one of these inequalities holds, then the set of tasks is schedulable according to RMS Theorem 2. If not, then the set of tasks is not schedulable according to RMS.

Example 6 illustrates the above rule of thumb.

Example 6

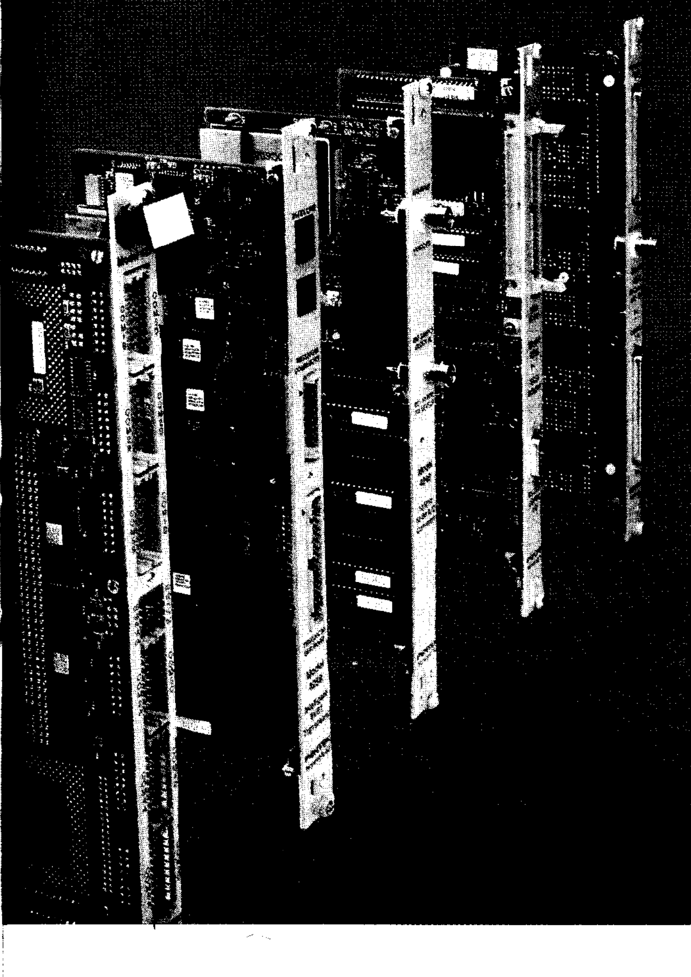
Let us analyze the set of tasks with the characteristics presented in Table 7.

Task i	Execution time C_i	Period T_i	Ratio C_i/T_i	Utilization *(1...N)*	Upper-bound
1	1	4	0.25	0.25	1.000
2	2	5	0.40	0.65	0.828
3	7	20	0.35	1.00	0.779

Table 7: *A set of tasks with utilization equal to 1.*

- Step 1. Applying Step 1, we see that the first two tasks are schedulable according to

DIGITAL SIGNAL PROCESSING DATA ACQUISITION SIGNAL ANALYSIS



If you need solutions to real-world problems in

- Multichannel spectrum analysis
- Real-time image processing
- Noise and vibration analysis
- On-line monitoring
- Process control
- Acoustics
- Digital audio
- Sonar or radar

contact the DSP experts at Pentek.
You'll be glad you did.

When it comes to DSP...
Pentek has the whole solution

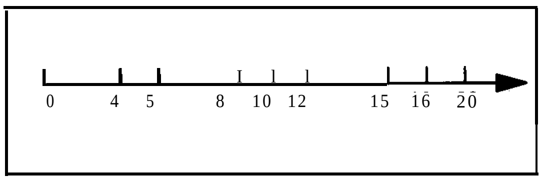


__Figure 2:__ Schedulability points for the three tasks discussed in Example 6.

RMS Theorem 1, but the third task is not. (Interestingly, the cumulative utilization factor is equal to 1. This happens in particular when tasks' periods are harmonically related.) Therefore, we have to apply Theorem 2. Let us do it with the rest of the rule of thumb given above.

- **Step 2a.** Determine all schedulability points from time 0 up to 20 (the end of the period of the lowest-frequency task). The schedulability points are

  for Task 1: 0, 4, 8, 12, 16, 20
  for Task 2: 0, 5, 10, 15, 20
  for Task 3: 0, 20

On the time axis, these schedulability points appear in the order presented in Figure 2.

- **Step 2b.** Construct inequalities for all schedulability points. The inequalities are

$$C_1 + C_2 + C_3 \leq T_1$$
$$2\ C_1 + C_2 + C_3 \leq T_2$$
$$2\ C_1 + 2\ C_2 + C_3 \leq 2\ T_1$$
$$3\ C_1 + 2\ C_2 + C_3 \leq 2\ T_2$$
$$3\ C_1 + 3\ C_2 + C_3 \leq 3\ T_1$$
$$4\ C_1 + 3\ C_2 + C_3 \leq 3\ T_2$$
$$4\ C_1 + 4\ C_2 + C_3 \leq 4\ T_1$$
$$5\ C_1 + 4\ C_2 + C_3 \leq T_3$$

- **Step 2c.** Substitute actual values for all variables, to check if at least one of the inequalities holds. The substitutions are shown below.

$$1 + 2 + 7 > 4$$
$$2 \cdot 1 + 2 + 7 > 5$$
$$2.1 + 2 \cdot 2 + 7 > 2 \cdot 4$$
$$3 \cdot 1 + 2 \cdot 2 + 7 > 2 \cdot 5$$
$$3.1 + 3 \cdot 2 + 7 > 3 \cdot 4$$
$$4.1 + 3 \cdot 2 + 7 > 3 \cdot 5$$
$$4 \cdot 1 + 4 \cdot 2 + 7 > 4 \cdot 4$$
$$5.1 + 4 \cdot 2 + 7 = 20$$

- **Conclusion.** The set of tasks from Table 7 is schedulable by Theorem 2, because one of the inequalities for the third task holds.

In Example 7, the rule of thumb is applied to the set of tasks described in Table 6 of Example 5.

Example 7

Let us analyze the set of tasks with the characteristics presented in Table 6. Are the results the same as those obtained by direct application of Theorem 2?

- **Step 1.** Applying Step 1, we see that the only task in question is again Task 3, so $i = 3$.
- **Step 2a.** Determine all schedulability points from 0 up to 360 (the end of the period of the lowest-frequency task). The schedulability points are

  for Task 1: **0,** 135, 270
  for Task 2: 0, 150, 300
  for Task 3: 0, 360

On the time axis, these schedulability points appear in the order presented in Figure 3.

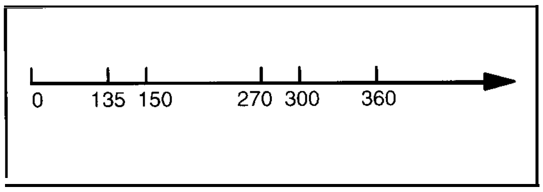


__Figure 3:__ Schedulability points for the three tasks discussed in Example 7.

- **Step 2b.** Construct inequalities for all schedulability points. The inequalities are

$$C_1 + C_2 + C_3 \leq T_1$$
$$2\ C_1 + C_2 + C_3 \leq T_2$$
$$2 \cdot C_1 + 2 \cdot C_2 + C_3 \leq 2 \cdot T_1$$
$$3 \cdot C_1 + 2 \cdot C_2 + C_3 \leq 2 \cdot T_2$$
$$3 \cdot C_1 + 3 \cdot C_2 + C_3 \leq 3 \cdot T_3$$

- **Step 2c.** Substitute actual values for all variables, to check if at least one of the inequalities holds. The Substitutions are shown below.

$$45 + 50 + 80 > 135$$
$$2 \cdot 45 + 50 + 80 > 150$$
$$2 \cdot 45 + 2 \cdot 50 + 80 = 2 \cdot 135$$
$$3 \cdot 45 + 2 \cdot 50 + 80 > 2 \cdot 150$$
$$3 \cdot 45 + 3 \cdot 50 + 80 > 360$$

- **Conclusion.** The set of tasks from Table 6 is schedulable by Theorem 2, because one of the inequalities for the third task holds. The results in Step 2c follow exactly the last column of fractions of Solution from Example 5, which confirms the applicability of the rule of thumb we used this time. One other thing worth noting that is quite unusual is that results of checks for longer periods (360, twice 150) are negative, but a check for a shorter period (twice 135) is okay.

If one understands how the above procedure works, it is possible to have more fun by finding the longest execution time a task can have with the task set still being schedulable. This is considered in Example 8.[6]

Example 8

In this example, there is a set of five tasks, only four of which have strictly determined timing characteristics (see Table 8). One of the tasks (Task 3) has a fixed period only, while it is desirable for its execution time to be the longest possible. The problem is to find this longest-possible execution time for Task 3, provided all five tasks are still schedulable by RMS.

Task i	**Exec ution time C_i**	**Peri- od T_i**	Ratio C_i/T_i	Uti- **liza-** tion ***(1...N)***	**Up- per-** bound
1	6	50	0.120	0.120	1.000
2	36	250	0.144	0.264	0.828
3	x	1000	x/1000	0.264+ x/1000	0.779
4	100	1200	0.083	0.347+ x/1000	0.756
5	120	1500	0.080	0.427+ x/1000	0.743

Table 8: *Characteristics of the five tasks cfiscussed in Example 8.*

- Step 1. Obviously, using inequalities from Theorem 1 does not give us the longest-possible time, because Theorem 1 specifies only the sufficient condition. Using Theorem 2, we have to apply the rule of thumb, if we want to make the whole calculation simpler; this is shown below.
- Step 2a. Because all schedulability points are multiples of the period of Task 1, from 0 up to 1,500, we have 30 such points to consider.
- Step 2b. Construct inequalities for all schedulability points. We have to construct 30 inequalities, one for each schedulability point. To save space, only a few of the inequalities are given here, in a general form.

$$C_1 + C_2 + C_3 + C_4 + C_5 \leq T_1 \qquad (1)$$
$$2 \cdot C_1 + C_2 + C_3 + C_4 + C_5 \leq 2 \cdot T_1 \qquad (2)$$
$$\ldots$$
$$29 \cdot C_1 + 6 \cdot C_2 + 2 \cdot C_3 + 2 \cdot C_4 + C_5 \leq 29 \cdot T_1 \qquad (29)$$
$$30 \cdot C_1 + 6 \cdot C_2 + 2 \cdot C_3 + 2 \cdot C_4 + C_5 \leq 30 \cdot T_1 \qquad (30)$$

- Step **2c.** By substituting values for all variables, we get a set of inequalities, from which we find the largest-possible value of x still meeting the condition of Theorem 2. The Substitutions are shown below.

$$6 + 36 + x + 100 + 120 > 50 \qquad (1)$$

$$7 \cdot 6 + 2 \cdot 36 + x + 100 + 120 \leq 7 \cdot 50 \qquad (7)$$

$$20 \cdot 6 + 4 \cdot 36 + x + 100 + 120 \leq 20 \cdot 50 \qquad (20)$$

$$30.6 + 6.36 + 2 \cdot x + 2 \cdot 100 + 120 \leq 30.50 \qquad (30)$$

Solving the above list of equations, from the seventh downward, we obtain

$$x + 334 = 350 \qquad (7)$$
$$.$$
$$x + 484 = 1000 \qquad (20)$$
$$\ldots$$
$$2 \cdot x + 716 = 1500 \qquad (30)$$

and we find the largest x = 516.

The technique used above is an example of applying the notion of *breakdown utilization,* which is the processor utilization at the point at which there is no additional processing capacity (that is, at the point at which there exists no single task whose execution time can be increased without making the whole task set unschedulable). More information on the use of breakdown utilization to evaluate various scheduling policies is given in a paper by Katcher, Arakawa, and Strosnider.[7]

Context switching and task dispatching

In calculations so far, it has been assumed that the change of tasks on the processor happens instantaneously; that is, it has been assumed that context switching takes no time. Every practitioner knows that this assumption is unrealistic. Each task switches context at least twice, assuming there are no preemptions. However, the duration of both context switching operations – save context and restore context – actually can be included in the task execution time. Assuming, for simplicity, that both context switching times are the same, and equal to C_S, we can add them to each task's execution time to have the extended execution time

$$C_i' = c, + 2 \times c,$$

Then, Theorems 1 and 2 can be reformulated for C'_i rather C_i than being the execution time.

Example 9

An interesting example was published by Borger, Klein, and Veltre[8] for the set of tasks presented in Table 9. This example shows the determination of the maximum context switching time C_S such that the tasks are still schedulable. A C program to do the computations, using a technique quite similar to that used in Example 8, is presented in the Appendix.

Task *i*	**Execution time** C_i	**Period** T_i	**Ratio** C_i/T_i	**Utilization** *(1...N)*	**Upperbound**
1	0.5	2.56	0.1953	0.1953	1.000
2	5.0	40.96	0.1221	0.3174	0.828
3	15.0	61.44	0.2441	0.5615	0.779
4	30.0	983.04	0.0305	0.5920	0.756
5	50.0	1024.0	0.0488	0.6408	0.743
6	1.0	1280.0	0.0008	0.6416	0.735

Table 9: *Characteristics of the six tasks discussed in Example 9.*

A slightly different Situation occurs if we want to include task dispatching time in the analysis. Normally, the task dispatcher runs at a priority higher than that of any other user task, so that its execution time should be incorporated into a task's execution time. However, when the dispatcher dispatches a low-priority task, higher–priority user tasks also suffer, because they can be blocked temporarily because of the dispatching Operation. Such blocking is effectively caused by a lower-priority task serviced by the dispatcher. Thus, blocking that results from dispatching lower-priority tasks must also be considered for higher-priority tasks. This is in addition to the increase of their execution time that results from their own dispatching. The increase in execution time of a user task that results from running the dispatcher is easy to account for, because – similar to the context switching time – it can be included in a task's execution time, as follows:

C; $+D_e$,

where D_e is the dispatcher execution time.

To account for a blocking time resulting from dispatching lower-priority tasks, one has to consider the worst case: when all tasks are activated at the same time. Because lower–priority tasks block all higher-priority tasks as a result of dispatching, the execution time of each task must be increased by a term consisting of number of lower-priority tasks, as follows:

$C''_i + (N - i)$ x **D„** ,

where $C''_i = C_i + D_e$ and D_b is the dispatcher's blocking time, where $D_b \leq D_e$. In general, D_b can be smaller than D_e if the low-priority task is activated at the same time as is the high–priority task.[8]

An example of a schedulability test for the set of tasks in Table 9, including both dispatching effects (that is, those resulting in times D_e and D_b), is given in a report by Borger, Klein, and Veltre[8] (for $D_e = D_b = 0.2$ milliseconds [ms]). A complementary view on including kernel overheads in the scheduling analysis, based on experimental findings, is presented in a paper by Burns, Wellings, and Hutcheon.[9]

Task synchronization

In addition to a negligible context switching time, another assumption in Theorems 1 and 2 that is rather impractical is that of task independence. In reality, the very essence of task existence is its interaction with other tasks, either via simple synchronization or via some way of communication (normally, message passing or shared memory). However, this interaction leads to a phenomenon known as *priority inversion.* Priority inversion occurs when a low–priority task prevents a higher-priority task from running, because the higher-priority task is blocked on a resource being used by the low-priority task.

Example 10

Consider three tasks –Task High, Task Medium, and Task Low – where the task names correspond to their priorities and Tasks High and Low need to use a common resource in an exclusive and nonpreemptive way. Assume, for example, that Task Low can run first because both other tasks are currently in a wait state (see time instant 0 in Figure 4).

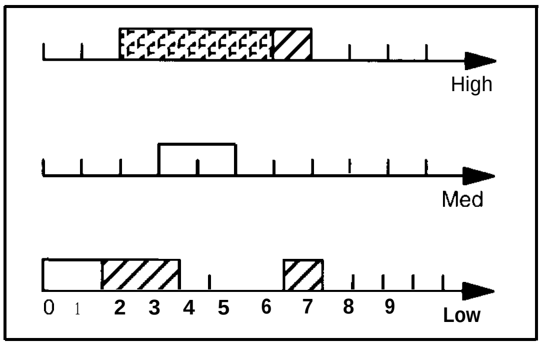


Figure 4: *Illustration of priority inversion.*

If Task Low starts using a resource (starting at time 1 and marked by slashes in Figure 4), it enters a critical section of code and normally uses some synchronization mechanism to prevent other tasks from executing the same code. (A *critical section* is a code Segment that requires exclusive execution by one task at a time until that section's completion.) If Task High wants to use the same resource a while later (starting at time 2 and marked by dots), Task Low cannot be preempted, because the resource is protected while Task Low executes. So, Task Low continues executing and Task High waits. This is called *direct blocking.*

However, Task Medium may want to run without using the resource (see time 3). Because Task Medium's priority is higher than that of Task Low, it certainly preempts Task Low and runs, thus causing Task High to wait for an extended interval of time (through times 4 and 5), even though Task Medium's priority is lower than that of Task High. This kind of blocking is called *push-through blocking.* After Task Medium has completed, Task Low resumes at time 5. Task High can execute only after Task Low has finished using the resource (at time 6). Theoretically, especially if there are more tasks of Task Medium's priority (between Task High's and Task Low's), this Situation may last indefinitely, thus blocking Task High for an unpredictable amount of time.

The Solution in the example just given is to boost the priority of Task Low, so that it temporarily becomes high, at the time Task Low is using the resource and Task High requests access to the same resource. Then, no other task, except one of a higher priority than that of Task High, can preempt Task Low, and Task Low can safely run until completion of using the resource. Such a boost is called *priority inheritance.* However, this Solution is not completely satisfactory, because deadlocks and multiple blocking may still occur. These are illustrated in Examples 11 and 12.

Example 11

In this example, assume there are two resources used by two tasks, Task High and Task Low, where task names correspond to their priorities.

If Task Low is accessing Resource 1 (starting at time 1 and marked by slashes in Figure 5) and Task High preempts it (at time 2), Task High may want to access Resource 2 (marked by backslashes) and then, before releasing Resource 2, to perform a nested access to Resource 1 (at time 3). Because Resource 1 is locked by Task Low, Task Low cannot be preempted and Task High cannot continue by using Resource 1 until Task Low finishes its own use of this resource. However, if Task Low, while still using Resource 1, makes a call to Resource 2 (at time 4), which is currently locked by Task High, a classical deadlock occurs.

This kind of Situation (that is, mutually nested calls to resources) is easy to avoid, and every practitioner would say that this is a design error. However, a second problem – multiple blocking – still may occur.

Example 12

Consider adding a second resource to be shared by the three tasks in Example 10, Tasks High, Medium, and Low. Assume that Task High needs to sequentially access Resources 1 and 2. If Task Low starts first and is accessing Resource 1 (starting from time 1 and marked by slashes in Figure 6), it can be preempted by Task Medium (at time 2), even though priority inheritance is used, because Task Low's priority is not boosted until Task High tries to get access to Resource 1. Thus, Task Medium starts running and accesses Resource 2 (marked by backslashes in the figure). Before it has completed, Task High gets activated (at time 3) and finds out that it can access neither resource (at time 4), because both resources are locked by two lower-priority tasks. This results in Task High being blocked for the duration of two resource accesses. Task High can continue only when the tasks that locked these resources complete (at times 5 and 6).

In the worst case, if a task shares *M* resources with lower-priority tasks, it can be blocked like this *M* times. An algorithm that solves such problems is called the *priority ceiling* protocol. Its basic idea is that a task can preempt another task that is currently accessing a resource and can access another resource only if the priority at which this new ac–

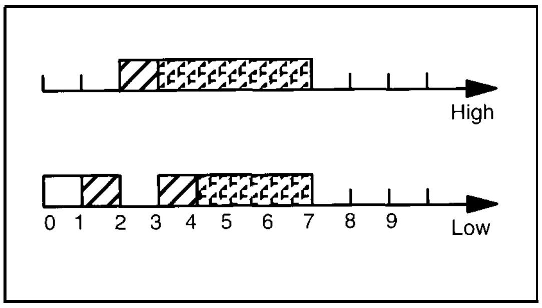


***Figure** 5: Illustration of a deadlock.*

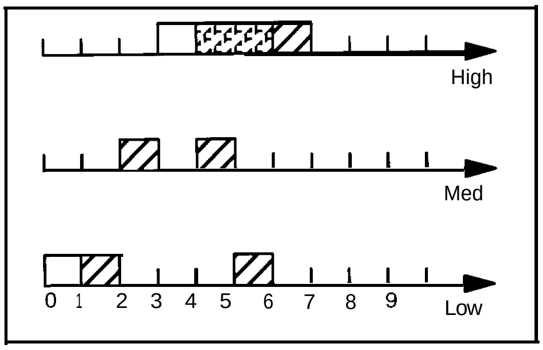


***Figure 6:** Illustration of multiple blocking.*

cess will be made is higher than the highest of the priorities to be inherited by the preempted task. If semaphores are used to control access to a resource, each Semaphore is assigned a priority (*ceiling*) equal to the highest priority of all the tasks that may access this Semaphore.

Rajkumar[10] gave examples illustrating avoidance of deadlock and of multiple blocking, for the priority ceiling protocol. A similar protocol, called the *stack resource policy*, was discussed in a paper by Baker.[11] It is left to the reader to apply the priority ceiling protocol to Examples 11 and 12, above.

It must be remembered that, in general, priority inversion cannot be eliminated, although it can be controlled by minimizing the duration of blocking due to priority inversion. Various kinds of blocking can be taken care of and included in conditions from Theorems 1 and 2,[12] as discussed below.

RMS Theorem 1A

For a set of N independent periodic tasks, where C_i, T_i, and $B_i, i = 1, 2, \ldots$ N, areexecution time, period, and worst-case blocking time, respectively, and assuming that task deadline equals task period, the tasks are schedulable by RMS if the following condition holds:

$$\forall n,\ 1 \le n \le N, \sum_{i=1}^{n} C_i/T_i + B_n/T_n \le n(2^{1/n} - 1)\ .$$

A weaker sufficient condition can be derived from the above set of inequalities. Its advantage is that it contains only one inequality rather than N inequalities.

RMS Theorem 2A

For a set of N independent periodic tasks, where C_i, T_i, and $B_i, i = 1, 2, \ldots$ N, are execution time, period, and worst-case blocking time, respectively, and assuming that task deadline equals task period, the tasks are schedulable by RMS if the following conditions hold:

$$\forall i \min \left[\sum_{j=1}^{i-1} \frac{C_j}{lT_k} \lceil lT_k/T_j \rceil + \frac{C_i}{lT_k} + \frac{B_i}{lT_k} \right] \le 1,$$

where min is calculated over $(k,l) \in W_i$, and

$$W_i = \{(k,l) | 1 \le k \le i, l = 1, \ldots, \lfloor T_i/T_k \rfloor \}$$

Note that this is only a sufficient condition, because values of B_i represent worst-case blockings. In practice, some of these inequalities may not be met and the set of tasks can be still schedulable, because actual blocking times are shorter. The rule of thumb given in the section entitled "Sufficient and necessary condition for RMS schedulability" can be applied by incorporating B_i into it.

Nonperiodic events

In practical applications, it is very unlikely that all tasks are periodically activated or that all modules request bus access periodically. Therefore, it is necessary, for any scheduling theory of practical importance, to consider nonperiodic events and their arrivals. However, a significant difficulty in analysis is caused by the fact that for most of the nonperiodic events, their rate is unbounded; that is, they may occur in bursts. Therefore, it is necessary to make certain simplifying assumptions about task arrivals, if they are nonperiodic.

The principal distinction that must be made is that between *fully nonperiodic tasks,* with no bounds on arrival times, and *sporadic tasks,* which have bounds on interarrivals *(maximum arrival rate)*. A *sporadic taskcan* be defined as a task whose subsequent arrivals *(requests for execution)* are separated by a certain minimum time.

In order to handle sporadic tasks, because they have guaranteed Separation time, the basic idea is to create a special periodic task that would take care of them. Thus, by treating sporadic tasks as if they were periodic, we put sporadic tasks into the framework of rate–monotonic scheduling for periodic tasks.

The simplest technique for treating sporadic tasks this way is presented first. In this technique, a *polling server* uses the spare processor utilization (remaining from use by all truly periodic tasks) to handle sporadic requests. The polling server algorithm is as follows:

- **Step 0.** The server is given the highest priority by making its period the shortest there is any spare capacity remaining according to Theorems 1 or 2.

- **Step 1.** At the beginning of its period, a polling server checks if any sporadic requests are pending.
- **Step** 2. If there are no unserviced requests at the moment, the server suspends until the beginning of the next period.
- Step 3. If there are unserviced requests, the server handles them until all its time (called its computation budget) is used.
- Step **3a.** If all requests are serviced within the available computation budget, the server suspends until the start of the next period.
- Step 3b. If the entire computation budget is used and there are still some unserviced requests (nonperiodic tasks pending), the server quits until the start of the next period, when it regains the whole computation budget for a new period. Regaining a computation budget is called replenishment.

One disadvantage of a polling server is that even if no event has arrived during a period, there is an overhead of checking whether there are sporadic tasks to be serviced. A more serious disadvantage is that if a Service request occurs right after the polling is done for a current period, then the request must wait until the start of the next period, even if some of the computation budget still remains unused. This means that treating urgent requests may not be possible. The Solution to this problem is to keep the unused computation budget throughout the entire period and allow all requests occurring through this period to be handled as they arrive, as long as some computation budgetstill remains. This is the technique used by a *deferred server.* As in the polling servertechnique, the replenishment of the computation budget for the deferred server occurs at the start of each period.

The problem with a deferred server is that it does notfit well into one of the theory's fundamental assumptions: that a task that is eligible for execution must run. The deferred server may not want to run, if it waits for the next nonperiodic task to arrive. This may hurt lower–priority tasks, in the sense that some of them miss their deadlines.

Another technique for handling sporadic events, which does not have the disadvantages of the polling server and the deferred server, is the *sporadic server* algorithm. The sporadic server task is given the highest priority and a computation budgetsuch that meeting the conditions of RMS Theorems 1 and 2 is still possible. However, in contrast to the other two techniques, the replenishment of a computation budget for the sporadic server is not restricted to the stat-t of a next period. The sporadic server makes replenishments when actual requests are serviced and replenishes only the amount of time used.

The detailed sporadic server algorithm is as follows:

- Step 1. If a sporadic request arrives and the server cannot handle it, because it is already busy or has no computation budget left, the request is queued for the future.
- Step 2. If a sporadic request arrives and the server can handle it, it does the following:
- Step 2a. Executes until Service completion or computation budget exhaustion.
- Step 2b. Determines the next replenishment point to occurone period afterthe start of current service.
- Step **2c.** Decreases the current computation budget by the amount used and increments its computation budget at the replenishment point by the same amount.

The sporadic server canbetter accommodate sporadic requests, because it is theoretically equivalent to a regular periodic task with a phase shift.[13]

Alternative solutions

Rate-monotonic scheduling theory is a specialcase of deadline-monotonic scheduling theory in which deadlines can be shorter than periods. (For deadlines longer than periods, no successful theory exists at the moment.) For the Situation in which priorities are based on deadlines in such a way that tasks with the shortest deadline get the highest priority (*deadline–monotonic scheduling*), a suff icient and necessary condition was proved.[14] (This should not be confused with the earliest–deadline-first scheduling policy, which is a dynamic algorithm requiring dynamic priority changes, with their readjustment at the start of eachperiod. This policy is not discussed here.) If we drop the fundamental assumption of RMS and allow deadlines to be smaller than periods, very interesting results can be obtained. These results are discussed briefly below.

In deadline-monotonic scheduling, the fundamental notion is task *response time*, which is equal to the worst-case running time of a task and includes the whole time from task activation to task completion. As a matterof principle, the execution of tasks at lower priority levels can start only if all executions at higher priority levels have been completed.

To determine schedulability, a simple equation per task needs to be solved[15,16]:

$$R_i = C_i + \sum_{j \in H} \lceil R_i / T_j \rceil \times C_j .$$

where R_i is the response time of Task iand H is a set of tasks of priorities higher than the priority of Task i. Because the response time R_i appears on both sides of this equation, the equation can be solved iteratively using the following equation for R_i at point n, which is dependent on its value at point n-l :

$$R_i^n = C_i + \sum_{j \in H} \lceil R_i^{n-1} / T_j \rceil \times C_j$$

The initial estimate, R^0_i, can be chosen as the cumulative execution time of all tasks of priorities higher than the priority of Task i. The iterations stop if $R^n_i = R^{n-1}_i$ or if $R^n_i > D_i$, where D_i is the deadline for Task i. It should be noted that, in general, the equation for R_i may have more than one solution. Also, it is possible to include a blocking component B_i in this equation, in a manner similar to that used in RMS Theorem 2A.

Example 13

To illustrate the use of both of the above equations, let us apply them to the problem from Example 5. The relevant data are repeated in Table 10, with added values for deadline (equal to period) and calculated response time.

Task i	Execution time C_i	Period T_i	Deadline D_i	Response time R_i
1	45	135	135	145
2	50	150	150	-95
3	80	360	360	270

Table 10: *Characteristics of the three tasks discussed in Examples 5 and 13.*

- **Task 1.** Because Task 1 has the highest priority, it starts first, and its response time is equal to the execution time, as follows:

$$R_1 = C_1 = 45.$$

- **Task 2.** If we put the numerical values straight into the main equation, we obtain, for Task 2.

$$R_2 = C_2 + \lceil R_2 / T_1 \rceil \times C_1$$

which is easy to solve, using a trial–and–error method, as $R_2 = 95$.

- **Task 3.** For Task 3, we need to use the iterative equation, as follows:

$$R_3^1 = C_3 + \lceil R_3^0 / T_2 \rceil \times C_2 + \lceil R_3^0 / T_1 \rceil = 185$$

$$R_3^2 = C_3 + \lceil R_3^1 / T_2 \rceil \times C_2 + \lceil R_3^1 / T_1 \rceil = 270$$

$$R_3^3 = C_3 + \lceil R_3^2 / T_2 \rceil \times C_2 + \lceil R_3^2 / T_1 \rceil = 270$$

Figure 7 illustrates the Solution for Example 13. Task 1, represented by an empty box, executes first (from time 0 to time 45). Then, Task 2, represented by a box with dots, runs for 50 time units (from time 45 to time 95). Task 3, represented by a box with slashes, is allowed to run until preempted by Task 1's second cycle (at time 135). Then, Tasks 1 and 2 both complete their second cycle and allow Task 3 to resume its first cycle (from time 230 to time 270). The Solution means that deadlines for all tasks can be set much shorter and the set of tasks would still be schedulable (see Table 10). As one can see from the computations in this example, using deadline-monotonic analysis turns out to be much easier than applying Theorem 2 of rate-monotonic scheduling theory.

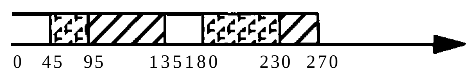


Figure 7: *Illustration of a solution in Example 13.*

The lessons learned from using deadline-monotonic scheduling are very important, so a brief discussion of them is appropriate here.

First of all, the notion of *processor(resource) utilization* happens to be much less significant in deadline-monotonic scheduling than has been claimed in rate-monotonic analysis. For example, if we choose deadlines smaller than periods but make them equal to the respective tasks' execution time, then tasks are never schedulable, no matter how low the processor utilization is. This means that we cannot base our judgment of task schedulability on how low the processor utilization is.

Second, if deadline-monotonic analysis is applied, sporadic servers are not needed, because any combination of periodic and sporadic tasks can be checked for schedulability using only the deadline-monotonic analysis test, without referring at all to rate-monotonic analysis.

Third, in deadline-monotonic scheduling, schedulability does not necessarily depend on how priorities are assigned, because the schedulability test is not related specifically to priorities. Thus, deadline-monotonic scheduling allows criticality or the importance of tasks to be considered in the assignment of priorities, without inadvertent effects on schedulability. This is in contrast to rate-monotonic scheduling, which requires priorities to be assigned strictly according to task periods.

Finally, for both theories, the assumption that all tasks start at the same time is too pessimistic. In practice, tasks are very often dependent on each other in such way that some of them must wait until data have been delivered by the others and cannot start execution before that time. This relationship among tasks is called *precedence relations.* One solution to this problem is to stretch deadlines by the time of execution of the task (or tasks) on which the current task is waiting. This solution facilitates task scheduling. The reader is referred to two papers, by Audsley, Burns, and Wellings[15] and by Burns,[17] for further details.

In conclusion, deadline–monotonic scheduling is easier and faster than rate-monotonic analysis. However, exact comparison of the two would require writing another paper besides this one. Fora more comprehensive view of what a system designer can do using deadline-monotonic analysis, the reader is referred to the paper by Audsley, Burns, and Wellings[15] mentioned above and to a paper by Audsley et al.[18]

Summary

Basic theorems of rate-monotonic scheduling have been presented and illustrated with several examples. Also, extensions – including context switching, task dispatching, and task synchronization – have been illustrated with examples, and handling nonperiodic events has been discussed. A critique of the basic assumptions of rate-monotonic scheduling has been presented. Finally, alternative solutions using deadline-monotonic scheduling have been discussed briefly.■

Dr. Janusz Zalewski is an associate Professor of Computer science at Embry-ßiddle Aeronautical University, Daytona Beach, Florida, USA. He worked at various nuclear laboratories, most recently at the Superconducting Super Collider Lab, Dallas, Texas, and Lawrence Livermore National Laboratory, Livermore, California, USA. He served on the working group designing the Multibus II Standard (IEEE Std 1296). His research interests include real-time multiprocessor Systems and safety-critical Computer systems. He received an M.Sc. in electronic engineering and Ph.D. in Computer science from Warsaw University of Technology, Poland, in 1973 and 1979, respectively.

Acknowledgments

This work was supported in part by a grant from the Advanced Research Projects Agency (ARPA) under Contract No. MDA972–92–J–1023.

The author is grateful to Daniel Mosse and Ken Tindell for their useful comments on earlier drafts of this paper.

References

1."IEEE Standard 610.12-1992: Glossary of Software Engineering Terminology," IEEE Press, New York, N.Y., 1992.

2. A.D. Stoyenko, "A Schedulability Analyzer for Real-Time Euclid," Proc. Real-Time Systems Symp., IEEE CS Press, Los Alamitos, Calif., 1987, pp. 218-227.

3. J.P. Lehoczky and L. Sha, "Performance of Real-Time Bus Scheduling Algorithms," ACM Performance Evaluation Rev., Vol. 14, No. 1, May 1986, pp. ***44-53.***

4. C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," J. ACM, Vol. 20, No. 1, Jan. 1973, pp. 46-61.

5. J. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," Proc. Real–Time Systems Symp., IEEE CS Press, Los Alamitos, Calif., 1989, pp. 166-171.

6. C.J. Paul et al., "Reducing Problem-Solving Variance to Improve Predictability," Comm. ACM, Vol. 34, No. 8, Aug. 1991, pp. 80–93.

7. D.I. Katcher, H. Arakawa, and J.K. Strosnider, "Engineering and Analysis of Fixed Priority Schedulers," IEEE Trans. Software Eng., Vol. 19, No. 9, Sept. 1993, pp. 920-934.

8. M. Borger, M. Klein, and R.A. Veltre, "Real–Time Software Engineering in Ada: Observations and Guidelines," Tech. Report CMU/SEI-89-TR-22, Software Eng. Inst., Pittsburgh, Pa., Sept. 1989.

9. A. Burns, A.J. Wellings, and A.D. Hutcheon, "The Impact of an Ada Run-Time Systems Performance Characteristics on Scheduling Models," Proc. Ada-Europe '93 Conf., Lecture Notes in Computer Sci. 688, Springer-Verlag, Berlin, Germany, 1993, pp. 240–248.

10. R. Rajkumar, Synchronization in Real-Time Systems: A Priority Inheritance Approach, Kluwer Academic Pub., Boston, Mass., 1992.

11. T.P. Baker, "Stack-Based Scheduling of Real-Time Processes," Real-Time Systems, Vol. 3, No. 1, March1 991, pp. 67-99.

12. L. Sha and J.B. Goodenough, "Real-Time Scheduling Theory and Ada," Computer, Vol. 23, No. 4, Apr. 1990, pp. 53-62.

13. B. Sprunt, L. Sha, and J.P. Lehoczky, "Aperiodic Task Scheduling for Hard Real-Time Systems," Real-Time Systems, Vol. 1, No. 1, June1989, pp. 27–60.

14. J.Y.–T. Leung and J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks," Performance Evaluation, Vol. 2, No. 4, 1982, pp. 237–250.

15. N.C. Audsley, A. Burns, and A.J. Wellings, "Deadline Monotonic Scheduling Theory and Application," Control Eng. Practice, Vol. 1, No. 1, Feb. 1993, pp. 71-78.

16. M. Joseph and P. Pandya, "Finding Response Times in a Real-Time System," Computer J., Vol. 29, No. 5, May 1986, pp. 390–395.

17. A. Burns, "Preemptive Priority Based Scheduling: An Appropriate Engineering Approach," in Principles of Real-Time Systems, SH. Son, ed., Prentice-Hall, Englewood Cliffs, N.J., 1995, pp. 225-248.

18. N. Audsley et al., "Applying New Scheduling Theory to Static Priority Preemptive Scheduling," Software Eng. J., Vol. 8, No. 5, Sept. 1993, pp. 284-292.

```
   #include <stdio.h>
main()
{
    int i,a1, a2,  a3,  a4,  a5;
        double x, cs, tmp, max_cs = 0.0. max_x = 0.0. n;
        al = 0;
        a2 = a3 = a4 = a5 = 1;
        for (x = 2.56; x <= 1280.0; x = x + 2.561  { /( compute at every schedulability pt (/
            printf("When x = %g: \n", x);
            a1++;
            if (((a1 % 16) == 1) && (a1 != 1))      /( 16 = 40.96 / 2.56 (/
                a2++;
            if (((a1 % 24) == 1) && (a1 != 1))      /( 24 = 61.44 / 2.56 (/
                a3++;
            if (((a1 % 384) == 1) && (a1 != 1))     /( 384 = 953.04 / 2.56 (/
                a4++;
            if (((a1 % 400) == 1) && (a1  != 1))    /( 400 = 1024.0 / 7.56 (/
                a5++;
            printf("%d(.5+2cs)+%d(5+2cs)+%d(15+2cs)+%d(30+2cs)+%d(50+2cs)+1+2cs<= %g\n",
                    a1, a2, a3, a4, a5, x);
            n = 2 0( (a1 + a2 + a3 + a4 + a5 + 1);
            tmp 0.5(a1 + 5.0(a2 + 15.0(a3 + 30.0(a4 + 50.0(a5  +  1.0;
            cs = (x - tmp) /  n;
            printf("cs <= %f\n\n", cs);
            if (cs > max_cs){    /( record the largest Cs and the time point(/
                max_cs = cs;
                max_x = x;
            }
        }   /( end of for (/
        printf("max_cs   %f  where x = %g\n\n", max_cs, max_x);
}
```

Appendix: *This C program does the computations for Example 9.*