An Introduction to Real-Time Scheduling

Ralf Guido Herrtwich
TR-90-035
July 30, 1990

Abstract

Until now, real-time processing techniques were used only in special computer applications such as process automation. With the advent of computer systems capable of handling time-critical data such as digital audio and video, these techniques become important for general-purpose computing as well. Real-time scheduling, i.e., assigning resources to processes in a way that takes into account the timing requirements of these processes, is the single most important technique in the construction of real-time systems. This tutorial introduces the most widely-used system models for real-time scheduling, describing resource characteristics, process parameters, and scheduling objectives. It summarizes, illustrates, and verifies essential findings about basic real-time scheduling algorithms such as earliest-deadline-first, least-laxity-first, and rate-monotonic scheduling for both sporadic and periodic processes.
1. INTRODUCTION

A system is called a real-time system if its correctness is not only determined by its output values, but also by the times at which these values become available [1]. Real-time systems are needed whenever a computation has to obey timing requirements of the system environment. Examples of traditional application areas for these systems include automatic control systems in manufacturing, process engineering, transportation, and warfare. More recent applications are computer-supported air traffic control and stock exchange. With the introduction of time-critical data such as video and audio into personal computers and workstations, real-time support becomes an essential element of general-purpose computing.

The common misconception about real-time systems is that their development has to focus on fast and efficient service. Fastness and efficiency are good qualities of any system — the main criterion for real-time systems is predictability of temporal behavior, i.e., to ensure (in practice and by proof) that the system will meet the timing requirements of its specification. Fast service is helpful to accomplish tight upper bounds on processing time, but for some applications, too fast may be just as bad as too slow — consider an audio file being played back at the maximum possible speed.

A typical objection to the predictability requirement for real-time systems is that hardware failures or software bugs can never be avoided; since they would ruin the well-planned temporal behavior anyway, it is not worth considering predictability in the first place. It is common knowledge that all guarantees about the functioning of a system are subject to limitations. It is the responsibility of system designers to decrease the odds of system failure within their capabilities and assignments. Ensuring predictable system behavior plays an important role in this process.

The main technique to achieve timing predictability is real-time scheduling, i.e., assigning system resources to processes in a way that takes their timing specifications into account. Numerous proposals for real-time scheduling algorithms have been made. Different fields such as discrete mathematics, economics, operations research, and computer science have contributed solutions. The purpose of this paper is to explain the major findings and most important approaches in real-time scheduling. It is meant as an introduction to the field, providing the necessary background to embark on studies of original papers in this area.

2. SYSTEM MODELS

The purpose of scheduling is to provide processes with the system resources they need for their execution. Any scheduling method is based on a certain system model, i.e., on assumptions about

- the available resources,
- the processes to be scheduled, and
- the scheduling objectives.

In the following subsections, we examine these issues in the context of real-time scheduling.
2.1. Resources

A resource is any system component that a process needs for its execution. Resources can be classified as active and passive resources. Active resources execute the process; examples include CPUs, network interfaces, and signal processors. Passive resources hold data of the process; examples include main memory segments, files on a disk, or frequency bands in a radio transmission. Since the primary resource for a process is, of course, a processor, most real-time systems schedule only this resource according to real-time mechanisms. In today’s systems, local “intelligence” and direct-memory-access capabilities allow other active resources to operate independently from the processor. Since in many applications they will contribute more to the temporal behavior of the system than the processor, they should be scheduled according to real-time criteria as well.

Resource usage. The need for scheduling results from limitations on resource usage. A resource needs to be used exclusively if its concurrent usage by more than one process would cause errors; a radio link is an example of an exclusive resource. If more than one process transmits data, signals get jammed and cannot be interpreted correctly. All active resources are exclusive resources. An example of a sharable resource is a read-only memory segment; many reader processes can access it correctly at the same time. If a resource needs to be used in exclusive mode - either always or sometimes - its usage has to be controlled by a scheduler. Scheduling is also needed if sharing is limited, i.e., if there are restrictions on the number of processes that can use the resource at the same time.

Resource capacity. Every resource has a certain capacity resulting from the functions it can accomplish within a certain time interval. A serial line, e.g., can have the capacity to transfer 19.2 Kbit of data every second. The purpose of a scheduler is to divide the capacity of a resource among processes in a way that meets the scheduling objectives. In the context of real-time scheduling, we are only concerned with a temporal division of resource capacity, resulting in time slices for which processes can use the resource. We assume that a potential spatial or functional division (e.g., the division of a frequency range into several bands) is already reflected in the resource granularity (e.g., that each frequency band constitutes a different resource).

1-resources and m-resources. A resource is unique if it is the only resource that can be used by a process to accomplish its task. We call such a resource a 1-resource, indicating that it is controlled by its own scheduler. The CPU in a uniprocessor system is an example of a 1-resource. If several resources can serve the needs of a process they are interchangeable. To store data, e.g., one main memory page is as good as the next. Interchangeable resources can be classified as homogeneous or heterogeneous depending on whether they accomplish a common function in the same way. In a heterogeneous multiprocessor system, the same process may be executed on different processors, but each execution will take a different length of time because the processor speeds are not the same. Correspondingly, processes will need time slices of different lengths. To take advantage of their equivalence, interchangeable resources should be managed as a group. A single scheduler combines m interchangeable resources to one m-resource (with m > 1).

---

1 It is not important in the context of this paper whether the scheduling algorithm is implemented by a separate system entity or is distributed among the processes that use a resource. When we use the term “scheduler” we mean both cases.
2.2. Processes

We call the schedulable entities of a software system processes\(^2\). For the purpose of scheduling, we do not need to distinguish between user and system processes since both require resource time slices in the same way – although perhaps with different urgency. To execute a process, exactly one active resource is needed. (In some systems, processes may require more than one active resource. Again, this is a matter of granularity that can be taken care of before the scheduler starts to work: any process requiring more than one active resource can be divided into subprocesses that require one active resource each.) In addition to its active resource, a process may require an arbitrary number of passive resources.

2.2.1. Time Parameters of Processes

To the real-time scheduler, the timing constraints of a process are described by one or more of the time parameters illustrated in Figure 1. For each process \( p \), the relation between all of its time parameters (described below) is

\[
A(p) \leq R(p) \leq S(p) \leq C(p) - E(p) \leq D(p) - E(p)
\]

Process execution. The execution time \( E \) determines how long a process needs access to a resource. At any time \( t \), a process \( p \) has a certain attained execution time \( E_{\text{att}} \) and some remaining execution time \( E_{\text{rem}} \) so that

\[
E(p) = E_{\text{att}}(p, t) + E_{\text{rem}}(p, t)
\]

Execution times depend not only on the services that a process requests from a resource, but also on the capability of a specific resource to accomplish these services. There is often no straightforward method to transform an execution time given for one resource into an execution time for another one. Even worse, execution times also depend on dynamic process behavior such as the

![Figure 1: Time parameters of a process.](image_url)

\(^2\) Depending on the particular system environment considered, they are also called tasks or threads.
alternative taken in an if-statement or the number of loops executed. Few tools are available to determine process execution times. While some calculate execution times from program code (e.g., [2]), others monitor the behavior of the running system (e.g., [3]). While the first approach promises to be more accurate, the latter approach seems to be more practical today, although not all applications offer the opportunity of test runs for monitoring purposes.

Process arrival. The arrival time $A$ of a process determines when it becomes known to the scheduler. Obviously, the scheduler cannot make a scheduling decision without knowing about a process and its requirements. Therefore, the arrival time of a process is the earliest time at which the scheduler can begin to schedule the process. If all processes have arrived before the system is started, scheduling can be complete, i.e., the scheduler runs once and its decision is final. If processes can arrive while the system is running, scheduling needs to be incremental, i.e., the arrival of a new process requires the scheduler to modify the schedule. The time it takes to schedule a process contributes to the time at which the process can be completed. Thus, scheduling itself can be time-critical.

Process start. The start time $S$ is the time at which a process obtains access to the resource. A process cannot be started before its ready time $R$, also called the time at which the process is requested. Ready times are used to ensure that the correct input is available to a process and that results are not delivered too early. They can also be used to reflect process precedence, i.e., that a process can start only if certain other processes have terminated before. A typical reason would be that one process has to wait for the delivery of data from another process. Precedence relations can also be expressed separately from the time parameters by means of a process precedence graph.

Process completion. The completion time $C$ is the time at which a process terminates. The latest useful completion time of a process is given by its deadline $D$. The deadline often results from the physical environment in which the real-time system is used. In a manufacturing system, it can result from the time at which a robot has to pick up some assembly unit from a conveyor

![Figure 2: Usefulness of process completion times given by a value function.](image-url)
belt. In a stock exchange system, it can result from the time at which the market closes. In a multimedia system, it can result from the time at which a certain video frame has to be displayed. A generalization of deadlines are value functions [4] as the one shown in Figure 2. For any potential completion time of a process, a value function gives a value that indicates the usefulness of this completion time.

**Process laxity.** The time between the ready time and the deadline of a process is called the *response time*. The response time minus the execution time of a process determines its *laxity* $L$. The laxity is the largest time for which the scheduler can safely delay the start of the process before then running it to completion without interrupts. Unlike its deadline, the laxity of a process changes over time. The longer a process has to wait for execution, the smaller its laxity gets. The laxity of a process $p$ at a time $t$ is defined as

$$L(p, t) = D(p) - R(p) - E_{rem}(p, t)$$

**Accuracy of time parameters.** Often time parameters of a process cannot be determined with complete accuracy. Therefore, it is common to describe time parameters by giving a worst-case estimate or by defining a value interval. Some models also use average values or statistical distributions. In addition, time parameters can either be *absolute* or *relative*. For example, if precedence relations between processes exist, the ready time of the second process would refer to the completion time of the first, inheriting potential uncertainties about this completion. The less accurately that time parameters of processes are given, the greater the potential waste of system resources, provided that the scheduler has to ensure deadlines by reserving a certain fraction of resource capacity for the worst case.

### 2.2.2. Special Process Characteristics

In addition to timing constraints, the scheduling of processes can be influenced by other parameters. These process characteristics have found their way into special process models.

**Process preemption.** The time slice that a scheduler assigns to a process either needs to be contiguous, or it can be divided into several fragments. A process is *preemptable* if its function is not changed by temporarily taking resources away from it. A common example for preemptable processes are those executing on the CPU. Processes using I/O devices, by way of contrast, are usually *non-preemptable*: once a printer has started to output a text file it needs to complete this task before another process can use it. Preemptability of a process determines the finality of a scheduling decision. For non-preemptable processes, the earliest time a revised schedule can become effective is the expiration of the current time slice. During this time, a *priority inversion* may occur, *i.e.*, although another process has a higher urgency, the resource cannot be taken away from its current user.

**Periodic processes.** In a large number of real-time applications, the same process recurs periodically. Examples are the periodic consumption of sensor data or the periodic delivery of video frames. The ready times of such processes are defined by their *period* $P$ (or by the reciprocal of the period, the *rate*).

$$R_{i+1}(p) = R_i(p) + P(p)$$
It is common that periodic processes have *backlog-avoiding deadlines*, i.e., that the execution of the process has to be completed before the next execution is requested.

\[ D_i(p) = R_{i+1}(p) \]

Some process models such as [5, 6] allow short-term violations of rate constraints in order to model slight irregularities in the occurrence of periodic processes.

**Imprecise computations.** Not all processes have to run to completion to produce useful results. Consider an approximation algorithm where a first calculation delivers a value that is then refined in several iterations. To obtain a not completely refined result is better than to obtain no result at all. This is reflected in the *imprecise computation model* [7] which decomposes every process into a *mandatory* and an *optional* part. The mandatory part is that portion of the process that must be executed in order to produce a result of acceptable quality. The optional part begins after the mandatory part is completed (an implicit precedence relation) and refines the result. The greater the portion of the optional part that is executed, the better the result will be. The optional part, however, can be left unfinished if necessary at the expense of result quality.

### 2.3. Scheduling Objectives

In addition to standard requirements on schedulers such as high resource utilization and low complexity, the straightforward requirement on real-time scheduling algorithms is that they find schedules where all processes meet their deadlines. Requirements are termed "hard" if not even a single deadline violation can be tolerated because it would result in an unacceptable damage or loss. Process automation systems typically have hard real-time requirements. Requirements are called "soft" if it is desirable to meet as many deadlines as possible, but a violation of deadlines can still lead to useful, although not optimal results. A video conference is an example of an application with soft real-time requirements: video frames arriving too late are simply dropped.

Depending on the process model, the objectives for scheduling may differ from "just" guaranteeing deadlines. Using value functions, the task of the scheduler is to maximize the total value. Using the imprecise computation model, the scheduler has to ensure that all mandatory process parts terminate before their deadlines and that the remaining resource capacity is used in a way that minimizes total error. Using a statistical model, some applications may only require that deadlines are guaranteed with a certain probability [9].

**Schedulability test.** To analyze if a process can be scheduled, the scheduler has to perform a *schedulability test*. When the test is successful, the scheduler is said to "guarantee" the execution of the process, i.e., it will always find a schedule that meets the process’ timing requirements (some schedulers may compute such a schedule during the schedulability test). Whether a process is schedulable depends first of all on its own time parameters, in particular, its ready time, execution time, and deadline. It also depends on its *waiting time* \( W \) that results from the competition for the resource with other processes after the process becomes ready. For each process

---

3 Sometimes the term "soft real-time system" is used to characterize a system in which timing constraints cannot be stated explicitly, but "fast" service is desired. We discourage the use of this term because it devalues real-time computing [8]: any computer system from which reasonable response times are expected — and which system would not belong in this category? — could otherwise be called a real-time system.
\[ p \], the following relation has to be fulfilled:

\[ E(p) + W(p) \leq D(p) - R(p) \]

The waiting time of \( p \) results from the scheduling algorithm used. It contains the cumulative execution times of processes with higher urgency than \( p \) (\( p \) either has to wait for their completion or is preempted by them). If scheduling is non-preemptive, the time it takes for a new schedule to become effective contributes to the waiting time.

**Optimality.** A schedulability test may turn down processes unnecessarily, i.e., although a schedule exists it cannot be found by the scheduler. Even if complete scheduling is possible, we can think of a scheduler testing the schedulability of processes one at a time. Consider an arbitrary sequence of processes with certain timing requirements. For this sequence, how "good" a scheduler is can be determined by its **guarantee ratio**:

\[
\text{guarantee ratio} = \frac{\text{number of accepted processes}}{\text{number of acceptable processes}}
\]

An algorithm is **optimal** if its guarantee ratio is 1 for all possible sequences of process requests, i.e., if it finds a schedule whenever such a schedule exists. For many scheduling scenarios, it is impossible to find optimal scheduling algorithms as we will see in Section 4.

3. INTERMISSION

Let us use a chess-board for the following game:

*We place the board on the table in front of us, marking its left vertical border as red and its lower horizontal border as blue. Let us then agree on a number \( m \). \( m \) is the minimum number of pieces we place on the board; we may place more on the board (which makes the game more interesting), and we can place more than one piece in each field. Let each of us in turns place the pieces on the board, or better yet, let our computer randomly do this for us.*

*In each move, \( m \) is the maximum number of pieces that can be moved one field down, towards the blue line. All pieces that are not moved down have to be moved one field to the left, towards the red line. The aim of the game is to move all pieces across the blue border — without a single one going across the red. Each of us, who comes up with a sequence of moves to win the game, gets one point.*

You do not need to win the game, but you should never lose...

4. SCHEDULING TECHNIQUES

Scheduling can be seen as searching — searching for a combination of time slices that allows all processes to meet their timing requirements [10]. The search space can be structured as a tree; Figure 3 illustrates this for the example of three non-preemptable processes on a 1-resource. The root of the tree represents an empty schedule. Each intermediate node is a partial schedule, extending the schedule of its parent node by one additional time slice. Any leaf of the search tree is a full schedule. Not all of these full schedules comply with the timing demands of the processes. The purpose of the scheduler usually is to search for a leaf in the tree that represents a feasible schedule.
The search space of a scheduler can be extremely large. When scheduling an $m$-resource, each node has up to 

$$\binom{n}{m}$$

subnodes, where $n$ is the number of processes. The depth of the search tree for non-preemptable processes is $n$, for preemptable processes it is given by the number of minimum-size time slices that fit into the time needed to execute all processes. Although scheduling algorithms that search the entire tree by enumeration are always optimal, their computation complexity is unacceptable: they are usually NP-hard. To make scheduling algorithms feasible, we either have to direct the search by appropriate heuristics, or have to use a method that can determine schedulability without enumeration.

4.1. Basic Real-Time Scheduling Algorithms

Any scheduling algorithm determines an order in which processes shall be executed. This is commonly termed assigning a priority to a process. In a real-time system, priorities indicate temporal urgency and should not be confused with traditional user-defined priorities that determine the importance or criticalness of the process relatively to other processes. A real-time system, however, can have user-defined priorities as well. If two processes have the same urgency, the
one with the higher user-defined priority is chosen. More advanced combinations are described in [11].

Elementary scheduling algorithms as they are used in time-sharing systems do not take the time parameters of processes into account when they determine priorities. In a first-come-first-serve or a round-robin scheduler, a process with zero laxity has to wait until all other processes ahead of it in the queue have gained access to the resource; in the general case, it will miss its deadline. Instead of choosing ready times as the primary scheduling criterion, real-time schedulers have to consider when a process needs to be completed. Hence, scheduling processes according to their deadlines or laxities seems to be more promising (see Figure 4). In the following subsections, we examine these algorithms for preemptable processes. We consider only the scheduling of the active resource a process needs and assume that requirements for passive resources are always met.

4.1.1. Earliest-Deadline-First Scheduling

The technique that has almost become a synonym for real-time scheduling is earliest-deadline-first (EDF) scheduling. As the name says, in EDF scheduling the process with the earliest deadline gets access to a resource. EDF scheduling is optimal for 1-resources [12], but not for m-resources as illustrated in Figure 5. For n processes with arbitrary ready times and deadlines, the
algorithm has a complexity of $O(n^2)$ for a 1-resource — for each ready time a comparison of deadlines is necessary.

**Proof of optimality.** Assume we have a set of $n$ processes, $p_1$ to $p_n$, with arbitrary, but fixed ready times, execution times, and deadlines. Let $s$ be a schedule that ensures that each process meets its deadline. We have to prove that in this case EDF will also find a schedule. To facilitate this proof, but without loss of generality, let us assume a certain time granularity $\tau$ for time parameters of processes. New resource assignments can only occur every $\tau$ moments, i.e., the scheduler manages time slices of the length $\tau$. Let us define the start time of the first time slice as $0$ and assume that $L$ time slices have to be assigned, numbered $0$ to $L-1$.

Let $proc$ be an array over time slices, containing the index of the process to which a time slice is assigned according to schedule $s$. We can transform $s$ into an EDF schedule by applying the following transformation (illustrated in Figure 6):
proc = schedule s;
for (i = 0; i < L; i++) {
    p = process that is ready and has the earliest deadline;
    if (proc[i] != p) {
        for (j = i + 1; proc[j] != p; j++);
        proc[j] = proc[i];
        proc[i] = p;
    }
}

If a time slice \( i \) is not assigned to the process \( p \) with the earliest deadline, the transformation "exchanges" \( i \) with the next earliest time slice of \( p \). As can be shown by induction over \( i \), the transformation leads to the same schedule as EDF.

Assume that \( i \) is the time slice taken away from \( p_i \) and given to another process \( p_j \). The number of time slices assigned to \( p_i \) and \( p_j \) within the interval \( 0 \ldots i \tau \) remains the same and no process is assigned a time slice before its ready time. Since \( p_i \) met its deadline in \( s \) we know that

\[ i \tau \leq D(p_i) \]

From EDF scheduling follows that

\[ D(p_i) \leq D(p_j) \]

so that
\[ t \leq D(p_j) \]

The exchange of time slices between \( p_i \) and \( p_j \) does, therefore, not affect the meeting of deadlines – EDF scheduling is optimal.

**Efficiency of EDF scheduling.** EDF scheduling is not only optimal, but also quite efficient in terms of process preemptions generated. In EDF scheduling, processes preempt each other when they become ready. Thus, each process will preempt any other process once, at the most. In the optimal case, not a single preemption would occur and a single process switch to start the process would suffice. Thus, an EDF scheduler will never cause more than twice as many process switches than any other feasible scheduling algorithm.

### 4.1.2. Least-Laxity-First Scheduling

*Least-laxity-first* (LLF) scheduling assigns resources to processes in the order of increasing laxity. LLF scheduling is not only optimal for 1-resources, but also for \( m \)-resources provided that all processes have identical ready times [13]. Both cases can be shown correspondingly by modifying the exchange algorithm given in the previous section.

**Comparison with EDF scheduling.** Why is LLF scheduling superior to EDF scheduling? Consider the example of a 2-resource with three processes that have identical ready times and deadlines. One process, \( p_1 \), has zero laxity, the others, \( p_2 \) and \( p_3 \), have some positive laxity as shown in Figure 7. To the EDF scheduler, all processes look the same. It will eventually prefer \( p_2 \) and \( p_3 \) to \( p_1 \). The LLF scheduler, on the other hand, takes the execution time of processes into account and can distinguish between these processes. (In fact, because of the ability to distinguish among such processes only schedulers that take execution times into account can be optimal for an \( m \)-resource [14].) Why does this difference not matter for 1-resources? If a zero laxity process is run on a 1-resource, there can be no other processes that need to be serviced in the same interval, otherwise no schedule would be feasible.

---

**Figure 7:** EDF scheduler unlike LLF scheduler cannot distinguish between processes.
Figure 8: Scheduling processes with different ready times on a 2-resource. Above: Processes to be scheduled. Middle: LLF is not optimal. Below: Feasible schedule.
Limitations of LLF scheduling. LLF scheduling is not optimal if the ready times are not the same for all processes as shown in Figure 8. No scheduling algorithm for an m-resource can be optimal that does not consider the ready times of future processes [14]. For the same reason, it is impossible to schedule optimally if the ready times of processes are unknown. We can show this by examining the scenario of the three processes in Figure 9 which shall be executed on a 2-resource. One process, p2, has zero laxity and needs to be scheduled immediately. Without loss of generality, we can assume that p2 fits exactly into a minimum-size time slice. We have to distinguish between three cases:

1. The second resource is assigned to p1 while p2 is executed. As shown in Figure 10, two new processes with zero laxity may become ready when p2 is completed. Since at this time p3 also has zero laxity and only two resources are available, one process will not meet its deadline, although a feasible schedule exists.

2. The second resource is first assigned to p3 while p2 is executed. Again, two processes with zero laxity and identical deadlines to p1 may become ready as shown in Figure 11. Again, we are short of one resource to execute three processes. There is, however, a feasible schedule.

3. The second resource is not assigned to a process as long as p2 is executed. In this case, either of the first two cases can be considered. The situation is made worse by the fact that the first time slice is not used.

All three cases together verify our claim; they not only hold for a 2-resource, but for any m-resource because extra resources can always be kept busy by introducing zero laxity processes. The cases also show that a scheduler can basically make two choices: it can execute the longer processes first at the risk that shorter and more urgent processes occur in the near future, as shown in Figure 10, or it can execute shorter processes at the risk of letting the resource idle, as shown in Figure 11.

---

**Figure 9:** Necessity of known ready times; initial scenario.
Efficiency of LLF scheduling. Although LLF scheduling may yield better results in terms of schedulability, it is less efficient than EDF scheduling; it can cause a larger number of process switches and usually requires several reevaluations of the scheduling criterion. While one process is executed, the laxities of all waiting processes decrease. At one point, the laxity of a waiting process can become smaller than the laxity of the running process. This would imply an immediate process switch. Shortly after starting the new process, however, the laxity of the preempted process would be smaller again. To prevent perpetual process switches, a minimum
time slice should be defined for which a process can keep a resource (as shown in Figure 12).

4.2. Real-Time Scheduling of Periodic Processes
Every scheduling algorithm that is applicable to sporadic processes is also feasible for periodic ones. The number of process occurrences that have to be tested for schedulability results from the time interval between the system start time 0 and the least common multiple of all process periods. After this interval, every interval looks the same, so if the processes can be scheduled in
the first interval they can also be scheduled in all following intervals. However, if the process periods are large and relatively prime, the number of process requests within that interval can become large, and more efficient methods of schedulability testing are desirable. We consider such schedulability tests for periodic processes with backlog-avoiding deadlines in the following subsections.

4.2.1. Periodic Earliest-Deadline-First Scheduling

When EDF scheduling is used for periodic preemptive processes with backlog-avoiding deadlines on a 1-resource, a very simple schedulability test is possible [15]. Using the parameters introduced in Section 2.2, a set of processes is schedulable if

$$\sum_i \frac{E(p_i)}{P(p_i)} \leq 1$$

While the necessity of this condition corresponds to intuition, its sufficiency is not at all obvious.

Proof of necessity. To show that the above condition is necessary to schedule the processes, let us consider the time interval between the system start time 0 and

$$\prod_i P(p_i)$$

Within this interval, each process $p_i$ requires a total execution time of
\[
\left[ \prod_i P(p_i) \right] \frac{E(p_i)}{P(p_i)}
\]

This results in a total execution time for all processes of

\[
P(p_2) P(p_3) \ldots P(p_n) E(p_1) + P(p_1) P(p_3) \ldots P(p_n) E(p_2) + \ldots + P(p_1) P(p_2) \ldots P(p_{n-1}) E(p_n)
\]

If this exceeds the capacity of the resource we have

\[
P(p_2) P(p_3) \ldots P(p_n) E(p_1) + \ldots + P(p_1) P(p_2) \ldots P(p_{n-1}) E(p_n) > \prod_i P(p_i)
\]

or

\[
\frac{E(p_1)}{P(p_1)} + \frac{E(p_2)}{P(p_2)} + \ldots + \frac{E(p_n)}{P(p_n)} > 1
\]

or

\[
\sum_i \frac{E(p_i)}{P(p_i)} > 1
\]

and cannot find a schedule that fits into the available time.

**Proof of sufficiency.** We show the sufficiency of the above condition by contradiction, assuming that

\[
\sum_i \frac{E(p_i)}{P(p_i)} \leq 1
\]

and that the scheduling algorithm is not feasible. If the algorithm does not find a proper schedule, there will be a deadline violation at some time \( t \). Processes can either have deadlines up to \( t \) or beyond \( t \). Processes with deadlines beyond \( t \) will be preempted by the more urgent processes at a time \( t' < t \). If no process with a deadline larger than \( t \) is executed before \( t \), let us set \( t' \) to 0, the system start time.

For each process \( p_i \), the requests that can have contributed to the deadline violation are

\[
\left[ \frac{t - t'}{P(p_i)} \right]
\]

because all other process requests would have a deadline beyond \( t \) and would not have been processed. Since in the worst case all processes can contribute to the deadline violation, the total execution time of all processes in the interval between \( t' \) and \( t \) is at most

\[
\sum_i \left[ \frac{t - t'}{P(p_i)} \right] E(p_i)
\]

In an EDF scheduler, a resource is never idle prior to a deadline violation (we show this below). Since the deadline violation occurs at \( t \) we know that the entire time since \( t' \) was used for process execution:

\[
\sum_i \left[ \frac{t - t'}{P(p_i)} \right] E(p_i) > t - t'
\]
Since \( x \geq \lfloor x \rfloor \) for every \( x \), we can write

\[
\sum_{i} \frac{t - t'}{P(p_i)} E(p_i) > t - t'
\]

or

\[
\sum_{i} \frac{E(p_i)}{P(p_i)} > 1
\]

which contradicts the assumption and shows the sufficiency.

**Addendum to the sufficiency proof.** To complete the sufficiency proof, have to show that in an EDF scheduler a resource is never idle before a deadline violation. Again, this is shown by contradiction. Assume that the first deadline violation occurs at some time \( t \) and that there is a time interval from \( t_a \) to \( t_z \) that is the last idle time prior to \( t \). None of the processes executed after \( t_z \) can have a ready time prior to \( t_z \), otherwise the idle time would have ended earlier. Also, all process invocations before \( t_z \) must have met their deadlines before \( t_a \), otherwise \( t \) would not be the first deadline violation and \( t_a \) would not be the beginning of an idle period. We can move the ready time (and deadline) of a process from some time later than \( t_z \) up to \( t_z \). Since there was no idle time between \( t_z \) and \( t_a \), there will be no idle time after the process is moved up. Besides, the deadline violation will occur at \( t \) or at some earlier time. We can repeat this for all processes, ending up with a situation where all processes are ready at \( t_z \). This situation is identical to the start of the system at time 0. For this situation, all processes have already met their deadline before \( t_a \). We can distinguish two cases:

1. \( t - t_z \geq t_a \): Since all processes met their deadlines in the shorter interval they will also meet them in the larger interval. This contradicts that \( t \) is a deadline violation.
2. \( t - t_z < t_a \): If the processes cannot meet their deadlines in the larger interval they will not meet them in the shorter interval. This contradicts that \( t \) is the first deadline violation.

Both contradictions determine that there can be no idle time prior to a deadline violation.

### 4.2.2. Rate-Monotonic Scheduling

EDF scheduling assigns process priorities *dynamically*: the priority of a process may change from request to request. By using a static priority assignment instead, where priorities are fixed once and for all, we could save some scheduling time. When preemptive processes on a 1-resource occur periodically and have backlog-avoiding deadlines, we can assign static priorities to them depending on their rates or periods. Such a *rate-monotonic* (RM) or *smallest-period-first* (SPF) scheduling is optimal among static priority schemes [15], i.e., if there is any static priority scheme that yields a feasible schedule, RM scheduling will do so as well. (Note that this does not mean that RM scheduling is as good as a dynamic priority scheme such as EDF scheduling.)

**Critical time zone.** The largest response time to which a periodic process can be subjected is called its *critical time zone*. The critical time zone of a process in a static priority scheme occurs when all processes with higher priority become ready simultaneously. We can prove this in the following way: Between all processes, we can establish a priority ordering \( p_1, p_2, \ldots, p_n \), where \( p_1 \) has the highest priority. Consider a request for \( p_j \) that occurs at time \( t_j \). Assume that between \( t_j \) and \( t_j + P(p_j) \), the next occurrence of \( p_j \), requests for a process \( p_i \) occur at \( t_i, t_i + P(p_i), t_i + 2P(p_i), \ldots \). Let \( p_i \) have a higher priority than \( p_j \). Each \( p_i \) request preempts \( p_j \) if \( p_j \) is not
completed before \( t_i \). This leads to a larger response time for \( p_j \), that results directly from the execution time of \( p_i \); the longer this time is, the later \( p_j \) is completed. The longest time occurs for simultaneous requests. Repeating this analysis for all processes that have a higher priority than \( p_i \), we have proven our claim. (These considerations also suggest to choose process rates as the criterion for priority assignment.)

**Proof of optimality.** To show that RM scheduling is optimal among static priority schemes, we can assume that we have found some other priority assignment \( q_1, q_2, \ldots, q_n \) that yields a feasible schedule. Let \( q_i \) and \( q_j \) be two adjacent processes in this ordering where \( q_i \) has the higher priority. Suppose that \( P(q_i) > P(q_j) \). In the critical time zone of \( q_j \), both \( q_i \) and \( q_j \) are executed.

\[
E(q_i) + E(q_j) \leq D(q_j) \leq D(q_i)
\]

If we change the execution sequence (i.e., the priorities) of both processes, they will still meet their deadlines. We can repeat this for all pairs of processes, ending up with an ordering \( p_1, p_2, \ldots, p_n \) that yields a feasible schedule and is equivalent to an RM ordering. Hence, if there is any feasible static schedule, RM scheduling is also feasible.

**Resource utilization.** The *utilization* \( U \) that is achieved by a schedule can be calculated as

\[
U = \sum_i \frac{E(p_i)}{P(p_i)}
\]

A set of processes is said to utilize a resource fully if there is a feasible priority assignment for it and any increase of process execution times would make the assignment infeasible. The minimal utilization factor of all process sets that fully utilize a resource defines a least upper bound on the resource utilization that can be achieved. Whenever we encounter a process set with a resource utilization below this bound, a feasible schedule can be found. Since RM scheduling is optimal it will find this schedule. Hence, knowing the least upper bound on resource utilization provides a simple schedulability test. If actual utilization is below this bound, schedulability is guaranteed. If it is beyond, this does not necessarily mean that no schedule can be found, but just that more elaborate schedulability tests are needed.

**Schedulability test for two processes.** Let us consider the resource utilization achievable by two processes \( p_1 \) and \( p_2 \). Without loss of generality, let \( p_1 \) have the smaller period and, thus, the higher priority. In the critical time zone for \( p_2 \) there are at most

\[
\left\lfloor \frac{P(p_2)}{P(p_1)} \right\rfloor
\]

requests for \( p_1 \). Let us try to adjust \( E(p_2) \) to utilize the resource fully within its critical time zone. The question is how large \( E(p_2) \) can become without violating deadlines. We have to consider two cases:

1. \( E(p_1) \) is short enough to complete all invocations of \( p_1 \) before the next request of \( p_2 \). In this case, \( E(p_1) \) fits into the time that is left if we subtract the total length of all periods of \( p_1 \) within \( p_2 \) from \( p_2 \). This is illustrated in Figure 13.

\[
E(p_1) \leq P(p_2) - P(p_1) \left\lfloor \frac{P(p_2)}{P(p_1)} \right\rfloor
\]
The largest possible value of $E(p_2)$ in this case is

$$E(p_2) = P(p_2) - E(p_1) \left[ \frac{P(p_2)}{P(p_1)} \right]$$

This results in a resource utilization of

$$U = \sum_i \frac{E(p_i)}{P(p_i)} = \frac{E(p_1)}{P(p_1)} + \frac{P(p_2) - E(p_1)}{P(p_2)} \left[ \frac{P(p_2)}{P(p_1)} \right]$$

$$= 1 + E(p_1) \left[ \frac{1}{P(p_1)} - \frac{P(p_2)}{P(p_1)} \right]$$

Since the factor of $E(p_1)$ is equal to or less than 0, utilization decreases monotonically in $E(p_1)$.

(2) The execution of the last request for $p_1$ within $P(p_2)$ overlaps with the second request for $p_2$ as illustrated in Figure 14. In this case we have

$$E(p_1) \geq P(p_2) - P(p_1) \left[ \frac{P(p_2)}{P(p_1)} \right]$$

The largest possible $E(p_2)$ uses up all time not occupied by $p_1$ before the final $p_1$ request.

$$E(p_2) = (P(p_1) - E(p_1)) \left[ \frac{P(p_2)}{P(p_1)} \right]$$
The utilization we achieve in this case is

\[
U = \sum_i \frac{E(p_i)}{P(p_i)} = \frac{E(p_1)}{P(p_1)} + \frac{(P(p_1) - E(p_1))}{P(p_2)} \frac{P(p_2)}{P(p_1)}
\]

\[
= \frac{P(p_1)}{P(p_2)} \left[ \frac{P(p_2)}{P(p_1)} \right] + E(p_1) \left[ \frac{1}{P(p_1)} - \frac{P(p_2)}{P(p_1)} \right]
\]

Since the factor of \(E(p_1)\) is equal to or greater than 0, utilization increases monotonically in \(E(p_1)\).

To find the minimum resource utilization, the largest \(E(p_1)\) from the first case and the smallest \(E(p_1)\) from the second case have to be chosen. The minimum occurs at the intersection of both cases for

\[
E(p_1) = P(p_2) - P(p_1) \left[ \frac{P(p_2)}{P(p_1)} \right]
\]

The worst utilization achievable can be calculated by inserting this value in one of the former utilization equations, e.g., the one from the first case:

\[
U = \sum_i \frac{E(p_i)}{P(p_i)} = \frac{P(p_2) - P(p_1)}{P(p_1)} \left[ \frac{P(p_2)}{P(p_1)} \right] + \frac{P(p_2) - (P(p_2) - P(p_1))}{P(p_1)} \left[ \frac{P(p_2)}{P(p_1)} \right]
\]

---

Figure 14: Schedulability test for RM scheduling: Overlapping execution time of \(p_1\).
\[
\frac{P(p_2)}{P(p_1)} = \left[ \frac{P(p_2)}{P(p_1)} \right] + 1 - \left[ \frac{P(p_1)}{P(p_2)} \right] + \left[ \frac{P(p_2)}{P(p_1)} \right] \left[ \frac{P(p_2)}{P(p_1)} \right]
\]

For brevity, let

\[ A = \frac{P(p_2)}{P(p_1)} \]

We can rewrite the above equation as

\[ U = \frac{A}{1 + \frac{1}{1 + F}} - \frac{1}{1 + F} \]

Let \( I = \lfloor A \rfloor \) be the integer part of \( A \), and \( F = A - \lfloor A \rfloor \) be the fractional part of \( A \).

Since \( U \) is monotonically increasing with \( I \), the smallest \( U \) occurs at the smallest possible \( I \). Since \( P(p_1) < P(p_2) \) and

\[ I = \left[ \frac{P(p_2)}{P(p_1)} \right] \]

the minimum \( I \) is 1. Minimizing \( U \) over \( F \), we obtain

\[ F_{\text{min}} = 2^{1/2} - 1 \]

This results in a minimum resource utilization of

\[ U_{\text{min}} = 1 - \frac{(1 - (2^{1/2} - 1))}{1 + (2^{1/2} - 1)} = 2 (2^{1/2} - 1) = 0.83 \]

Whenever two processes do not utilize the resource by less than 83\%, they are schedulable with the RM method.

**General schedulability test.** Generalizing the schedulability result for two processes, one can find that for a set of \( n \) processes with a static priority ordering the least upper bound for processor utilization is

\[ U_{\text{min}} = n (2^{1/n} - 1) \]

For large process sets, this utilization bound approaches \( \ln 2 \), i.e., the resource utilization will not exceed 69\% if \( U_{\text{min}} \) is used for schedulability testing. If all processes have the same period or all periods divide the largest period, much better utilizations can be achieved by testing the actual utilization rather than by using a worst-case estimate.

### 4.3. Scheduling Variations

Variations of the system model (special resource characteristics, new process parameters, or different scheduling objectives) can influence scheduling. Although in this paper we do not intend to describe the multitude of possible variations in detail (since many solutions discussed in the literature are highly application-specific and not transferable), let us briefly scan some of the more important findings.
Non-preemptive scheduling. Scheduling non-preemptable processes is more difficult than preemptive scheduling because no alternative to an enumeration of all potential schedules usually exists. EDF scheduling of non-preemptable processes with identical ready times remains optimal for a 1-resource [16]. This follows immediately from the preemptive EDF algorithm: a process \( p_1 \) with an earlier deadline will eventually preempt the running process \( p_2 \). Since \( p_2 \) will not be completed before \( p_1 \) is executed completely, \( p_1 \) could also be executed ahead of \( p_2 \), and the waiting time of \( p_2 \) would remain unchanged. To schedule processes with arbitrary ready times is NP-hard even on a 1-resource [17]. On an \( m \)-resource, scheduling is NP-hard even if all ready times are the same (and even if all deadlines are identical as well) [18].

Scheduling processes with precedence constraints. Precedence constraints of processes can be expressed either by defining corresponding ready times and deadlines or by considering precedence constraints independently. On a 1-resource, processes with precedence constraints can be scheduled by a latest-deadline-first (LDF) scheduler [19], regardless of whether they are preemptable or not. The algorithm assigns time slices in reverse order to processes with late deadlines first. At each time, the process that has the latest deadline is chosen from those processes which either have no successors or for which all successors have already been

---

**Figure 15**: Value functions for optional process parts in the imprecise computation model. Above: Linear increase. Middle: Increase faster than linear. Below: Increase slower than linear.
scheduled. On an \( m \)-resource, preemptive and non-preemptive scheduling are NP-hard [18]. Better solutions are possible by restricting the process precedence graph to certain tree structures.

Access to passive resources. Considering not only the active resource a process needs, but also its passive resources increases the complexity of the real-time scheduling problem. It is, therefore, no surprise that, apart from a few unrealistic cases, scheduling under passive resource constraints is again NP-hard [20]. Often other methods than time slicing can be used to provide time-critical processes with the passive resources they need. The most commonly used technique to provide access to main memory, \( e.g. \), is to "lock" pages of real-time processes in physical memory, guaranteeing their availability, preventing them from being substituted, and eliminating page faults (which is also helps to achieve a deterministic process execution time). This method decreases the utilization of memory pages, but if passive resources are plentiful such procedures are justified – provided that the schedulability test includes a check that enough passive resources are available.

Scheduling in distributed systems. Scheduling in a distributed system involves two levels:

- the local scheduling on each node, and
- a decision algorithm for assigning processes to nodes.

For local scheduling, all the above-mentioned algorithms can be used. Global process distribution is difficult even without real-time constraints (as shown by distributed load balancing algorithms). The reason is the uncertainty principle applying to distributed systems [21]: information on which a node assignment is based may be out of date. Consider the following example: A process \( p_1 \) arrives on node \( n_1 \). This node cannot guarantee \( p_1 \), but according to its state information about other nodes there is enough resource capacity available on \( n_2 \). Hence, \( n_1 \) sends \( p_1 \) to \( n_2 \). Meanwhile, another process \( p_2 \) arrives at \( n_2 \), is accepted by the scheduler (and reported to all other nodes), and takes all resource capacity. Once \( p_1 \) arrives at \( n_2 \), it again cannot be guaranteed. This is even nastier if there was a node \( n_3 \) on which \( p_1 \) could have been executed. The solution to the problem, however, is obvious: processes should never be sent to nodes without the nodes' former consent\(^4\). A bidding scheme can be used by which nodes apply for executing a process [22]. The price for obtaining an accurate scheduling decision is a potentially large (but necessarily bounded) communication delay which may contradict the timing requirements of the process to be scheduled.

Heuristic scheduling techniques. If no computationally feasible optimal scheduling algorithms can be found, a scheduler should be used that maximizes the guarantee ratio. An enumeration algorithm with strong bounding conditions can be used, provided it leads to a better than random result. The complexity of such a search algorithm can be adjusted by the number of backtracks allowed. The operational directives of the algorithm are always the same:

- Cut tree branches that lead to infeasible schedules.
- Consider the "most promising" branch first.

To decide about the most promising branch, some heuristics are needed. Earliest deadline and least laxity are, of course, promising heuristics, as can be validated by simulation [10]. The

\( ^4 \) The other possible solution – never to accept processes without the consent of all other nodes – is impractical in large distributed systems.
advantage of heuristic approaches is that the heuristic function can be arbitrary. Combined
scheduling criteria can be used to reflect the needs of a particular application. The disadvantage
(apart from not being optimal) is that the usefulness of the heuristic function always needs to be
validated by simulation.

Scheduling processes without hard deadlines. If processes have not been assigned hard dead-
lines, e.g., by specifying value functions for their completion times or by using the imprecise
computation model, the “value” of the schedule within the available execution time has to be
maximized. Again, in the general case it is impossible to obtain optimal solutions in due time,
and heuristic approximations are useful. To give an example which heuristics might help, con-
sider the scheduling of optional process parts in the imprecise computation model [23]. The
value of the completion time of these process parts increases with time. Assume that they all
reach some highest possible value \( V \) for the execution of the entire process part (see Figure 15).
If all value functions are linear, shortest-execution-time-first is a good heuristic function. If the
value functions increase faster than linear, least-attained-execution-time-first scheduling can be
used. If values increase slower than linear, most-attained-execution-time-first is a promising
solution.

Application-specific scheduling abstractions. The system model we have presented so far is
quite general. It can even accommodate cases which at first sight do not seem to be covered by it.
Consider the following example: in computer music, a single input event (e.g., a keystroke on a
synthesizer keyboard) may trigger a series of outputs which have to appear exactly at certain time
points. So far, we have only considered the entire output of a process as a whole and have
assumed that it has to be available before, but not at a certain time. By decomposing the process
into several subprocesses, it fits into our model: each output value is computed by its own pro-
cess; the ready time of the processes is the time of the input event (if values depend on each other

![Figure 16: Decomposing a process into subprocesses.](image-url)
the ready time of each subsequent process is the completion time of its predecessor), the deadline is the time at which the output has to appear minus some small time ε. These processes do not perform the actual output; this is done by other zero-laxity processes which have an execution time of ε and are started at the deadlines of the previous processes. Even if such a situation can be expressed in the "vanilla" model (as shown in Figure 16) and handled by the scheduling algorithms we have introduced, the decomposition should be made by the system and not by the user. A scheduler for an application area where special process requirements are common should provide corresponding abstractions to the user. For example, the scheduler of the FORMULA system for computer music provides a single abstraction that covers the case above [24].

Mixed scheduling. It will become more and more common that time-critical processes have to share their resources with uncritical processes. One example are CIM systems where the same machines can be used for time-critical process control and for uncritical production planning. Another example are multimedia systems where the presentation of discrete media such as text and graphics is not time-critical, but the display of continuous media such as audio and video is. In such systems, the scheduler has to operate in mixed mode. It has to fulfill two conflicting goals:

- Time-critical processes must never be subjected to an unbounded priority inversion because of uncritical processes.
- Uncritical processes should not suffer from starvation because time-critical processes are executed.

For preemptable processes, this conflict can be solved by reserving some fraction of the overall resource capacity for uncritical processes, i.e., the scheduler will not allow real-time processes to utilize resources fully⁵. It preempts uncritical processes as soon as a real-time process gets ready. For non-preemptable processes, some maximum time slice for uncritical processes needs to be defined. After this time, the process must have returned the resource or it will be aborted.

5. CONCLUSION

We have given an introduction to real-time scheduling techniques by summarizing, illustrating, and verifying essential findings of this area. Our tutorial has barely scratched the plethora of different scheduling algorithms imaginable. It should, however, have put the reader in a position from which the original papers about real-time scheduling become easier to understand. To find pointers to such papers and to obtain a more complete survey about the scheduling algorithms available, we recommend the study of [25]. This text is part of a collection, [1], which also provides a good introduction to the other techniques that are needed to build real-time systems.

⁵ In this case, the 69% resource utilization of an RM scheduler using the worst-case estimate for workload acceptance may be perfect.
ACKNOWLEDGEMENTS

This report is the belated transcript of a lecture I gave at the DASH Winter Retreat 1990. David Anderson wanted me to give this lecture; if you found it useful, thank him. A few parts of this text are taken from a book about concurrent and distributed systems that I wrote together with Guenter Hommel [21]. His influence on my work in the area of real-time scheduling is gratefully acknowledged.

REFERENCES


of Late Jobs”, Management Science 15 (1968).
18. J. D. Ullman, “Complexity of Sequence Problems”, in Computer and Job Scheduling
20. M. R. Garey and D. S. Johnson, “Complexity Results for Multiprocessor Scheduling
und echtzeitabhängige Programmsysteme, Springer-Verlag, Berlin, Heidelberg, New
22. K. Ramamritham and J. A. Stankovic, “Dynamic Task Scheduling in Distributed Hard
UCB/CSD 89/527, August 1989.
Hard Real-Time Systems, J. A. Stankovic and K. Ramamritham (editor), IEEE Computer
Society, 1988, 150-173.