An Introduction to Real-Time Operating Systems: Scheduling Theory

Clifford W. Mercer

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
cwm@cs.cmu.edu

DRAFT
Please do not distribute.

November 13, 1992

[**Reviewers:** *This paper is intended to be a combination tutorial/survey paper covering several common approaches to scheduling in real-time systems. It is intended for an audience of non-experts who understand the basics of computer programming and operating systems (at the undergraduate level). Fourth-year undergraduate students, first-year graduate students, and new hires in real-time software companies might find it useful as a first introduction to real-time systems, and it should provide a window into the major areas of the real-time systems scheduling literature.*

In the paper, the idea is to first define timing constraints and the like, and then describe several different approaches that have been used in building actual real-time systems (as opposed to concentrating on purely theoretical results). Each scheduling approach is introduced from the ground up, using straightforward language and small examples to illustrate the concepts. An evaluation of the practical considerations follows. References to further readings are given either in the introductory comments in each section or in the final paragraphs.

Taken as a whole the paper should give the reader a taste of several of the common approaches for scheduling real-time systems, and it should provide directions for further study.]

This work was supported by a National Science Foundation Graduate Fellowship.

Abstract

The functionality that real-time applications require of their operating system is much different from the functionality required by non-time-constrained time-sharing applications. Most of the differences are in the methods used for job scheduling and the systems support necessary to implement the scheduling policies. In this paper, we consider the various types of application-level real-time requirements, and we explore several different approaches to scheduling that have been used or proposed to schedule real-time activities: the cyclic executive, deterministic scheduling, capacity-based scheduling, and others. Our emphasis will be on practical methods, and we will find that these approaches have different implications for design and development, system operation, and maintenance.

1. Introduction

Real-time computer systems differ from non-real-time computer systems in that they must react to events originating in the physical world within a certain duration of time. A typical real-time system monitors and controls some external process, and the system must notice and respond to changes in the external process in a timely manner, usually on the order of tens or hundreds of milliseconds but sometimes on the order of seconds or on the order of milliseconds. If the external process is simple, then a single microcomputer which responds quickly to a few types of external events might suffice. However, many real-time systems are more complex and require more processors, more software structure, and a more sophisticated means of coordination among multiple activities; this motivates the need for methods of scheduling various activities in all but the simplest real-time systems. The scheduling is complicated by the fact that some real-time systems are such that a breakdown of the system scheduler (due to overload or poor design) can cause a catastrophic failure in the physical system, resulting in loss of life and property.

To begin our study of real-time systems and strategies for scheduling, we will first concentrate on defining the problem. We use the term "real-time system" or "real-time application" in the broadest sense to refer to the entire system including the physical environment, the software, and the computer hardware. The system is divided into several layers. At the top, we have tasks which are generic computational entities that have timing constraints and synchronization and communication relationships. These tasks are derived by some design process from the abstract timing requirements of the whole physical system. We may think of tasks as abstract computational activities or as implementation-dependent programs supported by real-time operating system. The time constraints for a task include the arrival time, the computation time, and the deadline; and several other properties serve to describe the task and the scheduling environment (these will be covered in detail in Section 2). The operating system manages hardware resources, schedules those resources, and supports the software architecture on which the tasks are implemented. The hardware layer is beneath the operating system. In this paper, we are mostly concerned with the nature of abstract timing requirements, the different ways for mapping those requirements into tasks with timing requirements, the methods for scheduling analysis based on the task specifications, and the mechanisms for scheduling the tasks in the operating system.

Two important properties of a task are the arrival characteristics and the nature of the deadline. The task arrivals may be periodic with a constant interval between successive invocations of the task, or the arrivals may be characterized by a statistical distribution where the inter-arrival time is a random variable. Tasks should complete execution before their deadlines, although there are many different ways of thinking about deadlines. In some cases, a task absolutely must complete its computation by the specified deadline; these are known as *hard* deadlines. If a task misses such a deadline, the value of completing the task after the deadline is nil. Missing a hard deadline might even result in the catastrophic failure of the system, depending on the task and the nature of its timing requirements. On the other hand, a task deadline may just specify a preference for completion time rather than an absolute requirement. In this case, the preference for completion time is called a *soft* deadline, and there is still some value to the system in completing the task after the deadline has passed.

This notion of hard and soft real-time system requirements plays an important role in the design of the system software. If the system has hard real-time requirements, the designer must go to great lengths to guarantee that no deadlines will be missed (at least under expected operating conditions). If the system timing requirements are soft, the designer can choose a more relaxed approach to system software. We will explore these issues further in Section 2.

The treatment of timing issues in Section 2 is tutorial in nature, and after we cover this background material, we can examine the approaches that have been used or proposed for managing the execution of these computations. We will discuss the cyclic executive, deterministic scheduling, capacity-based scheduling, and others. Some of the sections are tutorial in nature while other have the flavor of a survey; the sections after the first two are self-contained, depending only on the general discussion and definitions presented in the first two sections. Each section begins with some general background on the topic of the section and a discussion of historical development or motivation along with a few references to additional reading material. Subsequent discussion develops the ideas in more detail, either with a tutorial presentation or an overview of relevant work.

Section 3 describes the cyclic executive, a software structure that executes a pre-computed schedule or list of tasks repeatedly. This approach has historically been the most popular for hard real-time systems where fast, predictable execution is essential. These advantages do not come without a cost however: timing and synchronization properties are embedded in the code, a suitable schedule is often difficult to construct, and changes to the system may perturb the schedule and require a new schedule. This approach also requires that the time-constrained computations be deterministic and predictable themselves.

The basic ideas of deterministic scheduling theory are described in Section 4. Results from deterministic scheduling theory can be helpful in producing schedules for the cyclic executive, and the theory provides some insight into scheduling using software structures other than the cyclic executive. The limitation is that these theoretical results apply only to deterministic task sets and no allowance is left for variation. In this section, we explore the problem formulation for deterministic scheduling problems, and we summarize some of the general principles that emerge from the theory.

The capacity-based scheduling approach, covered in Section 5, was developed from some results in deterministic scheduling theory; this approach attempts to provide the predictable guarantees of deterministic scheduling while allowing the system designer more latitude in the organization of his software and system control structures. While some of the capacity-based work is directly applicable to practical systems, other more complex aspects of practical systems, such as interprocess communication and network communication are difficult to model and analyze.

Dynamic priority scheduling algorithms, described in Section 6, are attractive, especially since they are optimal for many simple scheduling problems. Unfortunately, they are typically not predictable under overload conditions, and this can be a drawback for real-time systems.

We consider value-function scheduling in Section 7 since this area has a long history in deterministic scheduling theory and since it has many attractive qualities. In practice, value-function scheduling is not widely used although some of the simpler, more efficient value-function scheduling techniques have been tried. A more common approach in practice is to pick a very limited value-function model where only piecewise linear functions with few segments are allowed rather than using general functions to describe task value. Value-function scheduling results have also proved useful in the recent work on scheduling imprecise computations described below.

In Section 8, we consider scheduling methods using imprecise results that have been developed recently. These methods allow for some tasks which have an optional computation as well as a mandatory computation; the tasks are usually organized such that the optional computation is a refinement on the results generated in the mandatory computation. In this framework, the mandatory computations must be completed by the task deadline and then the problem is to schedule the optional parts to optimize some scheduling criterion.

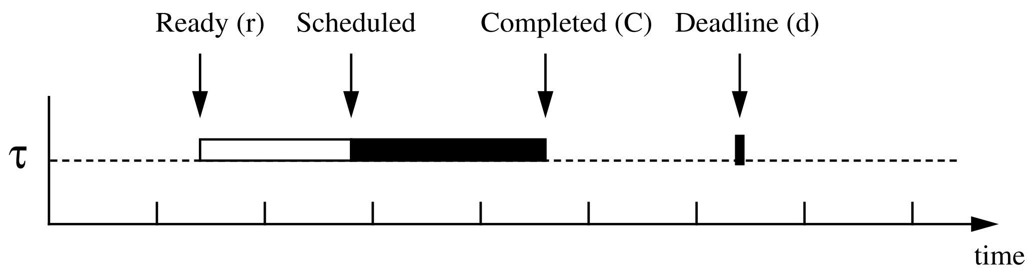

Figure 1: Schematic of a Time Constrained Computation

2. Computations and Time Constraints

Computational activities in a real-time system generally have timing constraints which specify when the computations begin, how long they last, and the deadlines for the computations. A precedence relation may be defined on the computations, indicating restrictions on the ordering of computations. The computations may also share resources (other than the processor), and these constraints on the computations must be specified and honored by the system.

2.1. Definitions and Notation

Figure 1 illustrates a computation schematically. Each computation has a ready time, r, at which the computation becomes available for scheduling. At some point after the ready time, the computation will (hopefully) be scheduled and start processing for a total duration of p. The computation will then complete at time C. A deadline, d, is typically associated with the computation as well, and the idea is to complete the computation before the deadline. Our convention with regard to this notation is that *a priori* task attributes (such as arrival time and deadline) get lower-case letters while attributes that are derived from a particular schedule (such completion time) get upper-case letters.

In actual systems there are many variations on this theme. The ready time of a computation may arise from a clock event, an external interrupt, or a software event generated by some other computation. The ready event may be an instance of a periodic computation where the same computation is activated periodically. The ready event may be aperiodic but predictable, or it may be unpredictable. The computation time may be fixed in duration or it may be variable or unpredictable. The computation itself may be preemptible, or it may form a non-preemptible critical region. The deadline is usually some fixed duration after the ready time, but the nature of the deadline may vary. Hard real-time computations take the deadline to be a *hard* deadline where the computation must be complete by the deadline time or a fatal error results. Alternatively, the deadline may just be a recommendation or preference for completion of

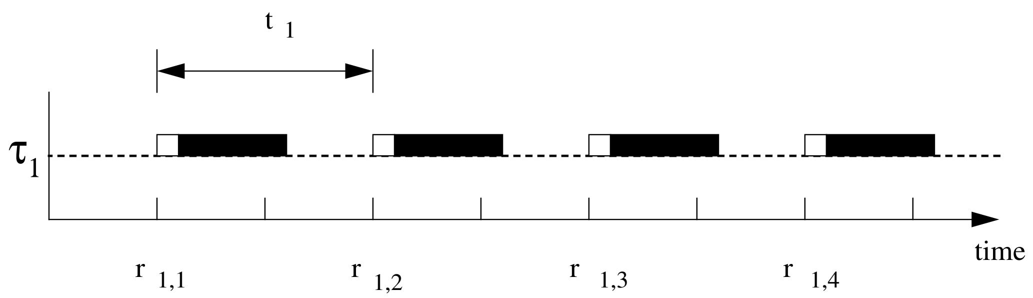

Figure 2: Periodic Task

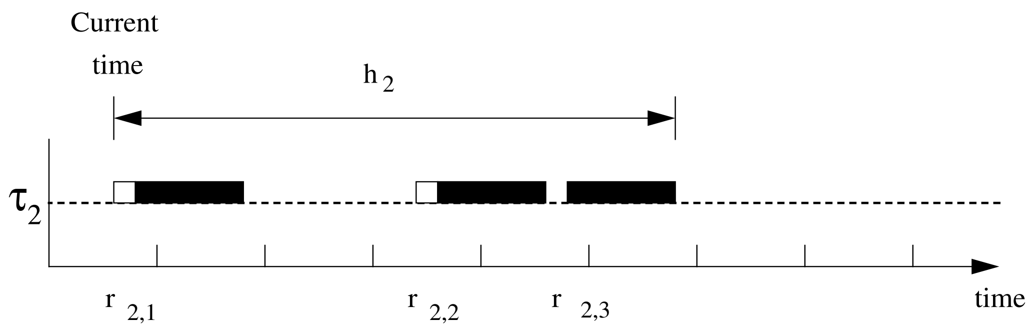

Figure 3: Aperiodic, Predictable Task

the computation, a *soft* deadline.

Since a computation may be periodic, we must sometimes distinguish between the overall activity and the periodically occurring computations. We call the overall activity a *task*, and we refer to the instantiations or individually scheduled computations of the task as *jobs*; thus a task is a stream of jobs. The jobs of a particular task are considered to be identical for the purposes of scheduling although slight variations can be indicated by a variable or stochastic computation time. We will use the word task to mean both the stream of instantiations and the individual instantiation when such usage is clear from the context.

Now we can define a periodic activity as a task where the ready times for the task instantiations are separated by a fixed duration, the period. Figure 2 shows a periodic task with periodically occurring instantiations. Task τ_1 is shown with four instantiations, each with an associated ready time, $r_{1,i}$. The ready times are separated by t_1 units where t_1 is the period of task τ_1. In this example, the computation time is constant across task instantiations, and the deadline is unspecified.

Aperiodic tasks are more difficult to specify. Some aperiodic tasks are predictable to a certain extent. For example it may be possible to predict the arrival of instantiations of an aperiodic task within some

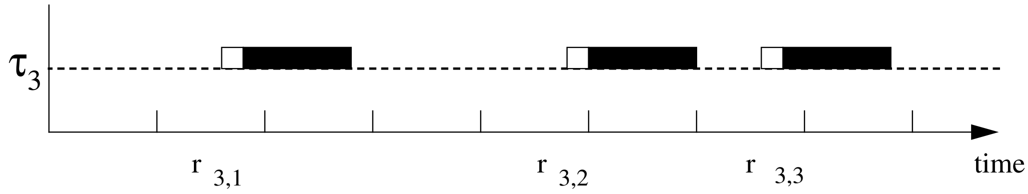

Figure 4: Aperiodic, Unpredictable Task

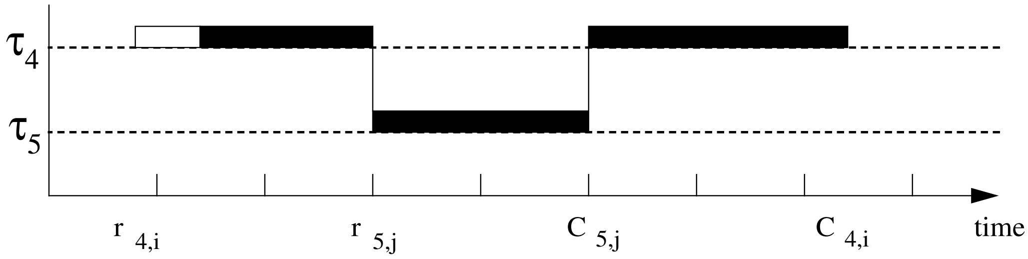

Figure 5: Preemptible Task

scheduling horizon or h_1 time units. Figure 3 shows an aperiodic task τ_2 with a scheduling horizon of duration h_2 from the current time. Within this window of h_2 time units, the ready times of instantiations of τ_2 are known, but beyond the horizon, nothing is known of the behavior of τ_2. Here again, we have assumed that the computation time is constant across instantiations in the single task, and the deadlines are left unspecified.

Another class of aperiodic tasks is almost completely unpredictable. It is common, however, to associate a minimum interarrival time for the instantiations of these unpredictable aperiodic tasks. The arrival process may be further described by a statistical arrival process. Figure 4 illustrates an aperiodic task where the arrivals are unpredictable. We may think of this as an aperiodic task with a zero-length horizon and with a statistical characterization of the arrival process. Aperiodic tasks which have hard deadlines are called sporadic (we will discuss the nature of hard and soft deadlines in the following paragraphs).

The nature of the computation time is another dimension along which tasks may vary. The computation time may be fixed or may merely be bounded in duration. The computation could also be described by a statistical distribution. Another characteristic of the computation is its preemptibility. It may be completely preemptible (that is preemptible at any point) or it may be non-preemptible. Or it may be preemptible but with one or more non-preemptible critical regions during which scheduling events are

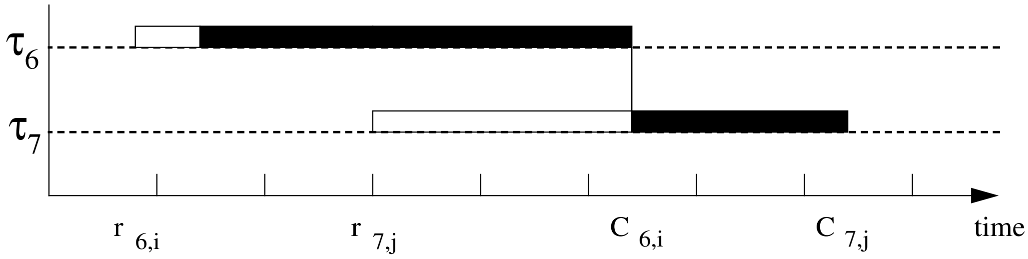

Figure 6: Non-preemptible Task

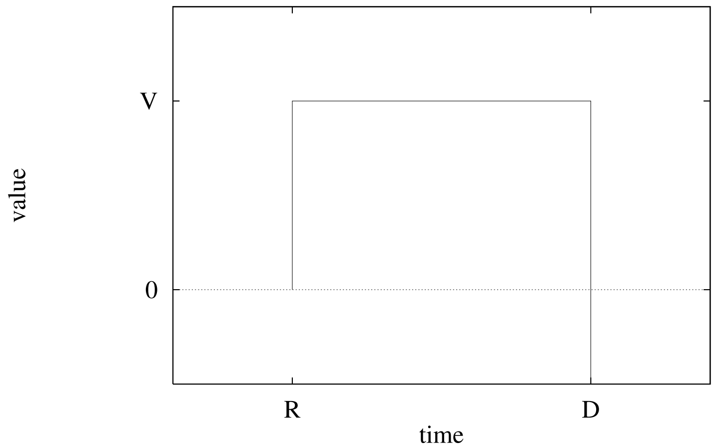

Figure 7: Hard (Catastrophic) Deadline Value Function

not allowed. Figure 5 shows an example of a preemptible task, τ_4, and its interaction with another task, τ_5. For this example, we assume that τ_4 is preemptible and has a lower priority than τ_5. τ_4 becomes ready at time $r_{4,i}$ and begins to execute immediately. At time $r_{5,j}$, τ_5 becomes ready, and since τ_5 has priority over τ_4, τ_5 preempts the ongoing execution of τ_4. After τ_5 completes, the execution of τ_4 resumes.

Figure 6 illustrates a similar case where the computation of τ_6 is non-preemptible and where τ_7 has priority over τ_6. τ_6 becomes ready at time $r_{6,i}$ and begins to execute. τ_7 becomes ready at time $r_{7,j}$, but even though τ_7 has priority over τ_6, τ_6 cannot be preempted, and τ_7 must wait until the execution of τ_6 completes. After τ_6 is finished, τ_7 can begin execution.

2.2. The Nature of Deadlines

Finally, we consider the nature of the deadlines of real-time computations. As indicated above, the deadlines may be classified as hard or soft deadlines. We can describe various types of deadlines by

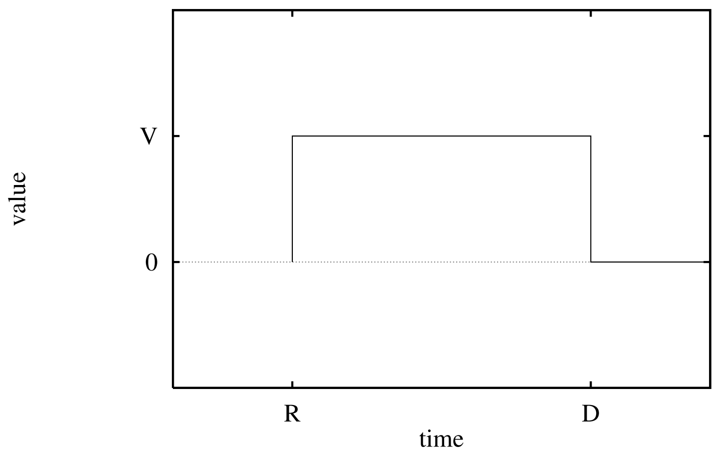

Figure 8: Hard Deadline Value Function

means of a value function. A value function is a function of time which indicates the value that completion of the task would contribute to the overall value of the system. For example, Figure 7 shows the value function of a task τ which has a hard deadline; the value drops off to $-\infty$ at $t = d$. The task becomes ready at time r, and its deadline is d. If the task is completed at time t where $r \leq t \leq d$, then the system receives some value, say measured as V. On the other hand, if the task completes after d, the value is $-\infty$, and consequently, the value of the system is $-\infty$, a catastrophic failure.

The result of missing a deadline may not be catastrophic, though. Figure 8 shows a case where completion of a task of τ would have some value until the deadline d when the value of completion of the task goes to zero. This indicates that the system will receive no benefit from completing the computation after d, and so the task should be aborted, freeing any resources it holds. In contrast to the previous case, the system can continue to operate, achieving a positive value even if this particular task is aborted and makes no contribution to that value.

Other variations on the idea of hard deadline might include a value function that ramps up to the deadline as illustrated in Figure 9. And depending on where the ramp starts, this type of value function can specify tasks which must be executed within very narrow intervals of time.

The concept of a soft deadline is illustrated in Figure 10 where the value function goes to zero after the deadline. In this case, there is some value to the system in completing the task after the deadline although this value may go to zero after some period of time. Therefore, the task should not be aborted right away as in the case of the hard deadline. The distinguishing feature would be the duration of the drop to zero value.

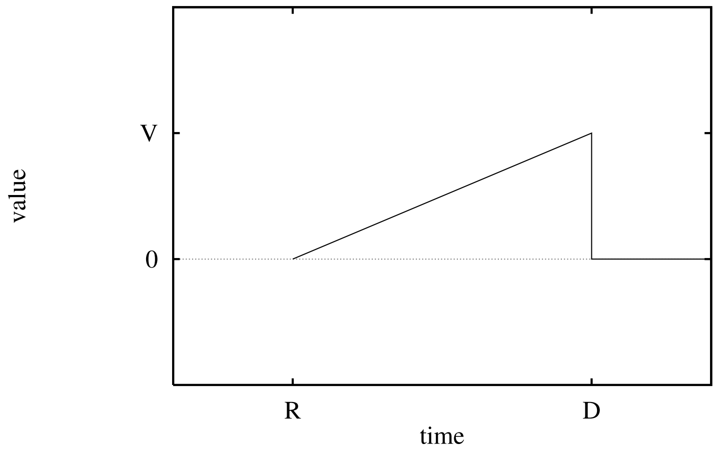

Figure 9: Ramped Hard Deadline Value Function

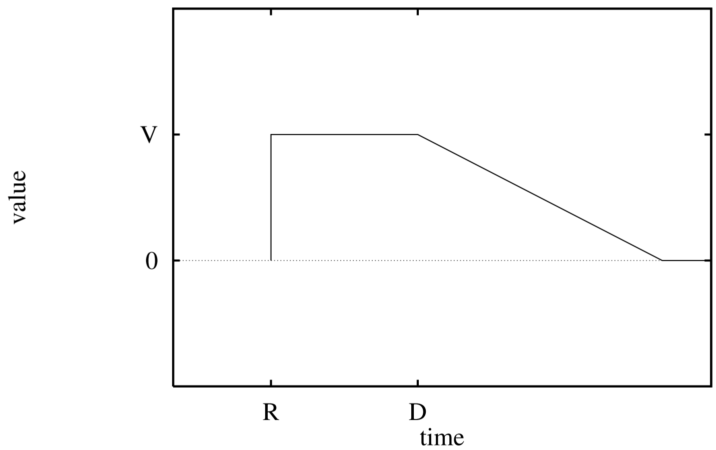

Figure 10: Soft Deadline Value Function

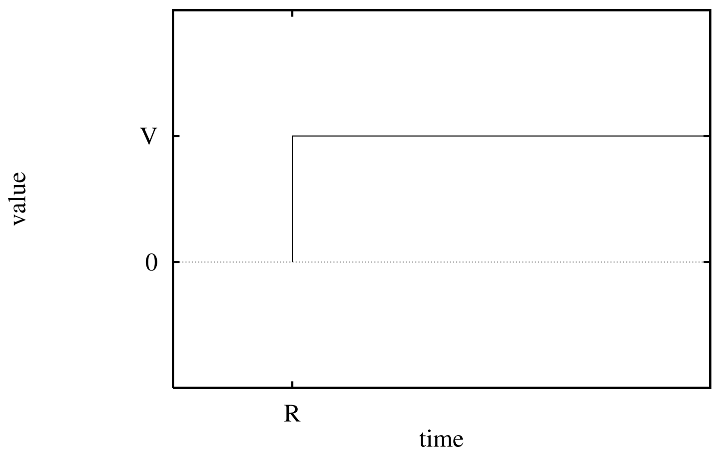

Figure 11: Non-real-time Value Function

A non-real-time task might be described by the value function shown in Figure 11. In this case, completion of the task always has a positive value associated with it. This indicates no explicit timing constraint, although in practice, few of us are willing to wait indefinitely for a computation to complete.

2.3. Example Task Set

In order to make the discussion of the scheduling algorithms more concrete, we will define an example task set with some timing requirements. Consider a home computer which is used as a communication station and workstation. The computer controls various devices in the home, supports a telephone interface to the external phone lines, provides an internal intercom service, provides multimedia communication services, and fills the role of the traditional home computer. We take as our example the task set described in Table 1 whose specification includes time constraints.

The intention of this task set is to illustrate the various conflicts that may occur among tasks with different timing requirements. We consider tasks with hard deadlines, tasks with soft timing requirements, and still other tasks with no timing requirements. In this example, the hard real-time tasks check the smoke detectors and the motion detectors for any unusual disturbances. These tasks must be performed at regular intervals, otherwise the system falls into a state where it doesn't know if there is a fire or not, a bad state to be in. This information must be checked every 15 s in a very predictable fashion. There are a few tasks which have timing constraints but which are not critical to the preservation of life; these are the digital intercom, digital phone, and digital audio tasks. One person is talking on the external phone line, and two people within the house are using the intercom. The voice packets must be transmitted

Task	t		p		Description
τ_1	15	s	2	ms	smoke detector
τ_2	15	s	2	ms	motion detector
τ_3	40	ms	5	ms	digital audio intercom
τ_4	40	ms	5	ms	digital telephone
τ_5	20	ms	12	ms	CD quality audio
τ_6	15	s	2	ms	toilet overflow detector
τ_7	1	d	60	s	download newspaper
τ_8	1	h	20	s	compile homework assignment
τ_9	1	s	1	ms	keystroke

Table 1: Example Task Set

at intervals of 40 ms, but missing a few packets is not catastrophic, merely annoying. Another person is listening to the CD quality audio which has a period of 20 ms. Lost packets are considerably more annoying here, but again, the tasks are not life-critical. Another soft real-time tasks is the toilet overflow detector. This task should be invoked every 15 s, but again the result is not (really) critical. Other activities are background activities such as downloading the news from a satellite link or compiling a programming assignment. These are not periodic tasks, but we have some idea of how often they will be invoked. We also know roughly how long the computation will take. And the processing of a keystroke is an interactive activity; the keystroke is unpredictable, but a reasonable response time is required.

These tasks must be scheduled carefully to avoid unintended behavior. For example, FCFS scheduling of these tasks would be disastrous; downloading the news would effectively shut down other activities in the system. The smoke detectors and motion detectors would be completely disabled for the duration of the download (1 minute or so). Also, the phone and intercom conversations would be disrupted. Figure 12 illustrates the scheduling sequence. Even if round robin scheduling (where tasks are preempted after a time quantum) were used, there would be potential problems. Since the smoke detector needs only a very small amount of computation during a large period, chances are that its requirement will be satisfied. But it is unlikely that the audio tasks will get the computation time they need in the exact period where it is needed (see Figure 13).

We require a scheduling policy which can ensure that the timing requirement of each task is met. In the next sections, we consider different approaches for solving the problem of scheduling these activities.

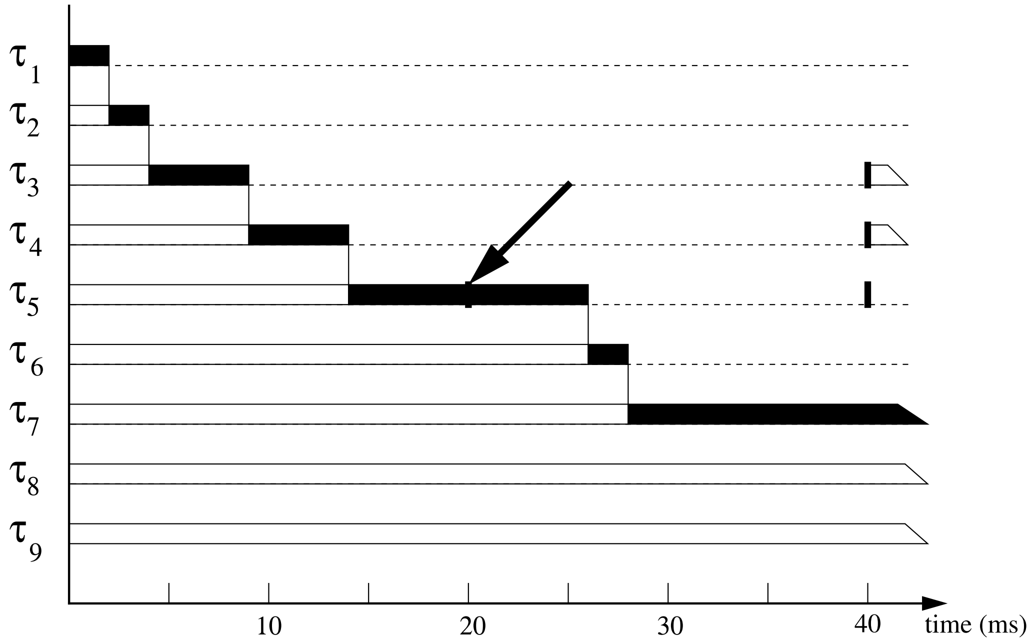

Figure 12: FIFO Scheduling of Example Task Set

3. Cyclic Executive

A cyclic executive is a supervisory control program, or executive, which dispatches the application programs of a real-time system based on a cyclic schedule constructed during the system design phase, and this schedule is executed repeatedly throughout the lifetime of the system. The schedule consists of a sequence of actions to be taken (or subroutine calls to be made) along with a fixed specification of the timing for the actions. The cyclic executive typically has several such schedules, and long-term external conditions dictate which alternative is chosen for execution (this idea of alternative *modes* is discussed further in the following paragraphs).

Since virtually all of the scheduling decisions are made at system design time, the executive is very efficient and very predictable. In general, no scheduling decisions need be made at runtime, although gross changes in external conditions may cause the executive to switch to an alternative schedule. These changes are infrequent and present more of an initialization and termination problem than an on-going scheduling problem. The major difficulty with the cyclic executive approach is that the system is very inflexible and difficult to maintain. Adding new tasks or even changing the computation time of the existing tasks can invalidate the pre-computed schedule, and these schedules can be very difficult to generate.

The cyclic executive model has been used in many real-time systems, but there is no universal

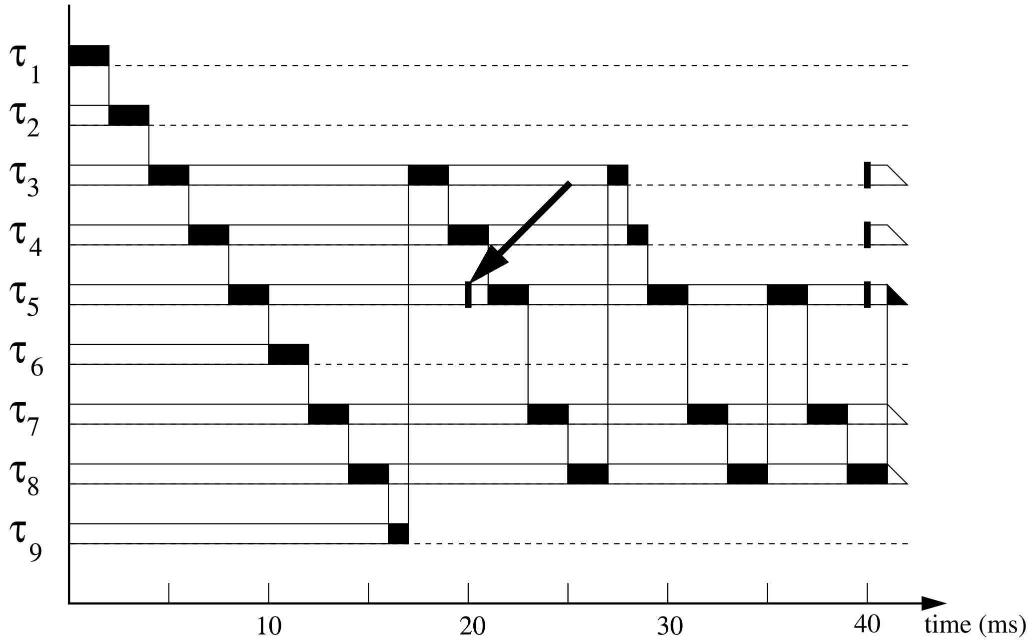

Figure 13: Round Robin Scheduling of Example Task Set

approach. Each system is built using *ad hoc* techniques tuned for the specific application domain. Hood and Grover give a detailed description of the model and variations on the central theme, and they discuss the advantages and disadvantages of the approach [15]. Their treatment of the model is very general, although a primary focus of the report is to evaluate Ada for use in implementing cyclic executives. Baker and Shaw take a more formal approach to defining the cyclic executive model, and they present a detailed analysis of the issues and problems with the approach [2]. Their article also considers many detailed issues in the use of the model in Ada systems, but their discussion of the model itself is very general. Locke describes the cyclic executive approach along with the fixed priority executive approach, and he discusses the implications and relative advantages and disadvantages between the two approaches [30]. Many systems use variations on the basic cyclic executive approach [5, 10, 37]. It is interesting to note that much of the discussion of practical real-time scheduling techniques such as the cyclic executive has been motivated by the mandated use of Ada in real-time systems and the conflict between the Ada computational model and the requirements of real-time systems.

3.1. Cyclic Scheduling

The cyclic executive provides a practical means for executing a *cyclic schedule*, but the model does not specify how the schedule is derived. The cyclic schedule is a timed sequence of computations (variously called scheduling blocks, actions, subroutine calls) which is to be repeated indefinitely, in a cyclic manner. This cyclic schedule is also known as the *major schedule* and the duration of the major schedule is called the *major cycle*. The major schedule is divided into minor schedules of equal duration, and the duration of these minor schedules is called the *minor cycle*. The minor schedules are also known as frames. The timing of the computations in the cyclic schedule is derived from the timing of the frames; the frames are initiated, in order, by a periodic clock interrupt or some similar mechanism. The individual frames are designed to execute for at most the duration of the minor cycle, but if the execution of a frame exceeds the minor cycle, a *frame overrun* is said to occur. Frame overruns may be handled in a number of different ways, but the point here is that the timing of each frame is verified only at the end of the minor cycle.. The executive has no knowledge or control of the timing of computations within a frame.

The structure of the cyclic executive as described above provides a means of executing cyclic schedules, but the problem of finding the schedule in the first place must be addressed. The cyclic executive is designed for environments where periodic processes, event-based processing, and background processing are predominant. Periodic processes are particularly well-suited to the cyclic executive and often determine the major and minor cycles of the executive. Event-based processing can be handled with the cyclic schedule in several different ways. One way is to give higher priority to the processes that are reacting to events. In this case the frames containing periodic processes must be able to tolerate the delay experienced while waiting for the event processing to finish. Another more popular idea is to allocate slots in the frames where aperiodic, event-based processes can be serviced. In this way, the interference to the cyclic scheduling structure can be reduced, but the response time for events may be longer. Another possibility is to service aperiodic events in the background, during time when the processor would otherwise be idle. This is also the way that regular background processing is managed.

Even though the major and minor cycles are derived from the timing constraints of the periodic processes, there is no standard procedure for computing the major and minor cycles or the assignment of computations to frames or the ordering of computations within frames. Deterministic scheduling theory (see Section 4) may be used to help find schedules that meet the timing constraints of all the tasks, but optimal deterministic scheduling algorithms require *a priori* knowledge of the timing parameters of all tasks, and optimal non-preemptive scheduling of computations with timing constraints is NP-hard. Preemptive scheduling is much easier, but a schedule with arbitrary preemptions may be very difficult to implement since each preemption requires that the computation of the preempted jobs be split at precisely the right place; and preemption also adds overhead. Furthermore, resource constraints and precedence constraints among the jobs might preclude preemption in particular circumstances. Some of the principles of deterministic scheduling theory may be applied to help find feasible schedules by hand, and Baker and Shaw give some constraints on the duration of the minor cycle which help to direct the search for values for the major cycle and minor cycle [2].

All of these complications tend to make schedule construction difficult. The lack of algorithms for finding feasible schedules means that designers must rely on their wits and intuition to produce a suitable schedule.

In complex systems, groups of tasks are specialized and are designed to run only under certain conditions. On a flight platform, for example, different groups of tasks will be required for different phases of flight like taking off, cruising at high altitude, and landing. These phases have different functional requirements, and different major schedules are required for each different phase. Such changes are called *mode changes*, and they involve the termination of the previously executing major schedule and the preparation and initiation of the next major schedule.

3.2. Example Schedule

To illustrate the concept of the cyclic executive, we will construct a schedule for our example task set. Since each task is required to execute at least one time during the major cycle, we will take the duration of the major cycle to be 15 s. Note that the other periods (20 ms and 40 ms) divide the major cycle evenly. We take the minor cycle to be 20 ms since that is the period of the highest frequency task. So each minor cycle will contain one execution of τ_5. The other two audio tasks, τ_3 and τ_4, must each appear in every other minor cycle, and each of the detector tasks must appear once in every 25 minor cycles (i.e. once every major cycle). The aperiodic tasks in our task set are placed in the remaining empty slots of the timeline.

We construct our timeline by first placing the computation for τ_5 in each minor schedule (Figure 14).

Now we must place τ_3 and τ_4 on the timeline. They will not both fit into the same minor schedule with τ_5, so we must place them in alternate minor schedules (see Figure 15).

And we have 17 ms of each 20 ms minor schedule consumed so far. We still have the low frequency tasks to place. Each must go in a separate minor cycle, say at the beginning of the major cycle, and then they are quiet until the next major cycle. So we have the timeline shown in Figure 16.

We now have all of the periodic tasks positioned on the timeline. It is evident that we have five unique minor schedules. The first minor schedule $\mathcal{C}_1 = \tau_5\ \tau_3\ \tau_1$, the second is $\mathcal{C}_2 = \tau_5\ \tau_4\ \tau_2$, and the third is $\mathcal{C}_3 = \tau_5\ \tau_3\ \tau_6$. Then we have the other two forms alternating for the rest of the major cycle. These last two forms are $\mathcal{C}_4 = \tau_5\ \tau_4$ and $\mathcal{C}_5 = \tau_5\ \tau_3$.

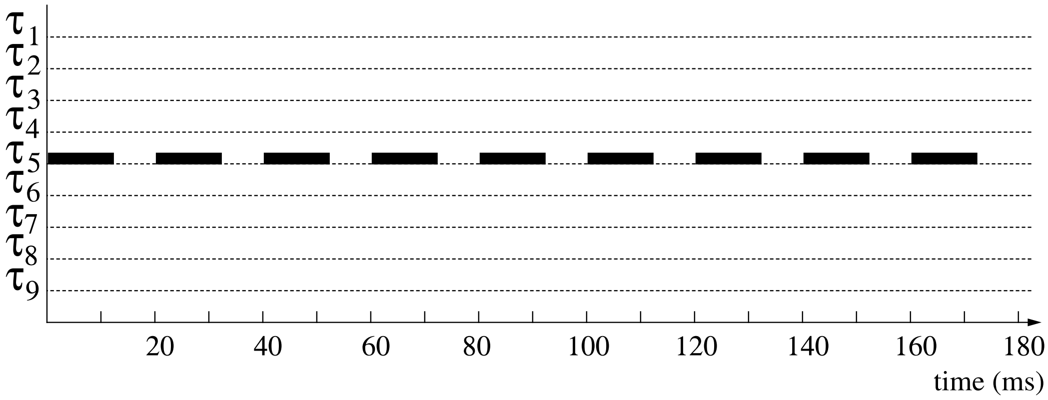

Figure 14: Timeline for Example Task Set (First Pass)

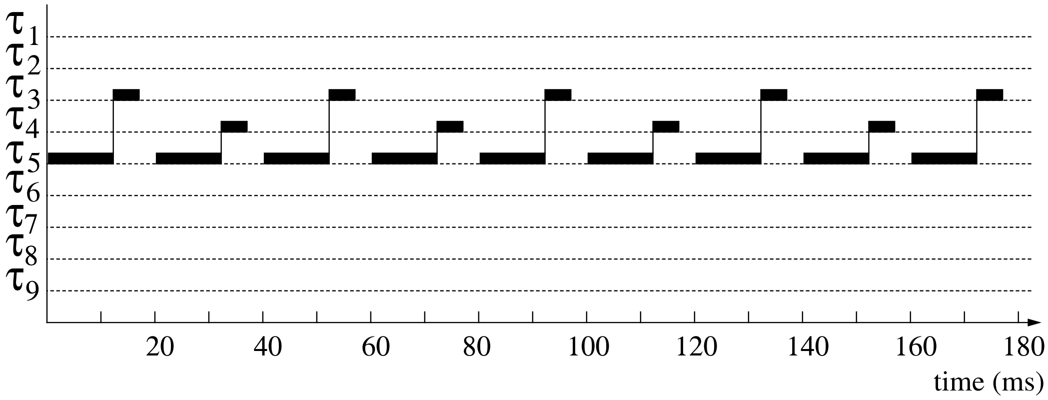

Figure 15: Timeline for Example Task Set (Second Pass)

We can now express the major schedule as a sequence of minor schedules. The major schedule is

$$\mathcal{C} = \mathcal{C}_1\mathcal{C}_2\mathcal{C}_3(\mathcal{C}_4\mathcal{C}_5)^{11}$$

where $(\mathcal{C}_4\mathcal{C}_5)^{11}$ indicates that the subsequence $\mathcal{C}_4\mathcal{C}_5$ is repeated 11 times for a total of 25 minor schedules in the major schedule. The system then repeats the schedule $\mathcal{C}$ forever (assuming we have one telephone conversation, one intercom conversation, and one CD quality audio stream going forever!).

Notice that in the final timeline (Figure 17), we have varying amounts of time left over at the end of the minor cycles. Some of this idle time has been used by the aperiodic background activities. The only restriction is that these background activities must be preemptible. That is, they must be ready to relinquish the processor immediately when the time comes to begin the next minor schedule.

This all seems well and good, but what happens if we want to add another time-constrained task to the system? Suppose we want to introduce τ_{10} with t_{10} = 40 ms and p_{10} = 4 ms. We must try to include this task in our timeline (shown in Figure 17).

But we cannot immediately put this task into either of the first two minor schedules since there is

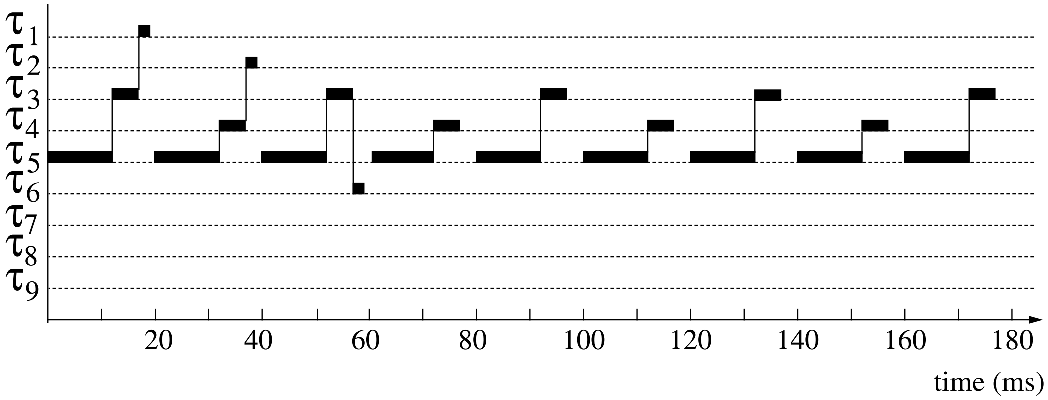

Figure 16: Timeline for Example Task Set (Third Pass)

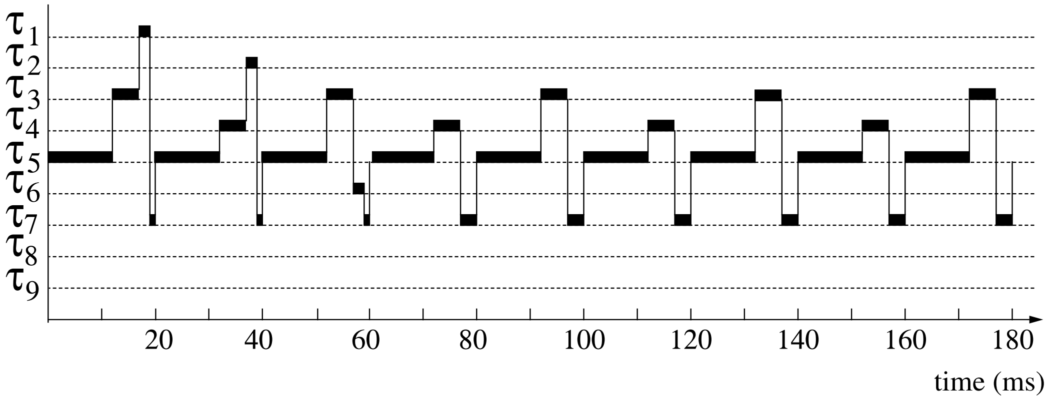

Figure 17: Timeline for Example Task Set (Final Schedule)

only 1 ms free in each. We cannot even spread the τ_{10} computation over both of the minor cycles since there is only 2 ms of idle time and we require 4 ms. The only possibility is to remove one of the low frequency tasks and service that task later. Doing that will free up a 3 ms slot, and if we are to perform the 4 ms computation, the execution must be split into a 3 ms part and a 1 ms part (or even 2 ms and 2 ms). Of course we are not guaranteed that the computation may be split in this way. There may be resources which are held over the course of the computation and which may need to be released and later re-acquired if the computation is to be split into two parts. Assuming that we can split the computation of τ_{10} into two parts, 3 ms and 1 ms, we must then place these computations into the schedule. We shift the low frequency computations as necessary to get the schedule in Figure 18.

It is clear that incorporating this additional task required a substantial redesign of the schedule. There is no general solution to the problem of creating these timelines and their generation is regarded as a kind of art. In this case, we also saw another issue which complicates the design: preemptibility of the computation. In general, the tasks are programmed without regard to concurrent programming

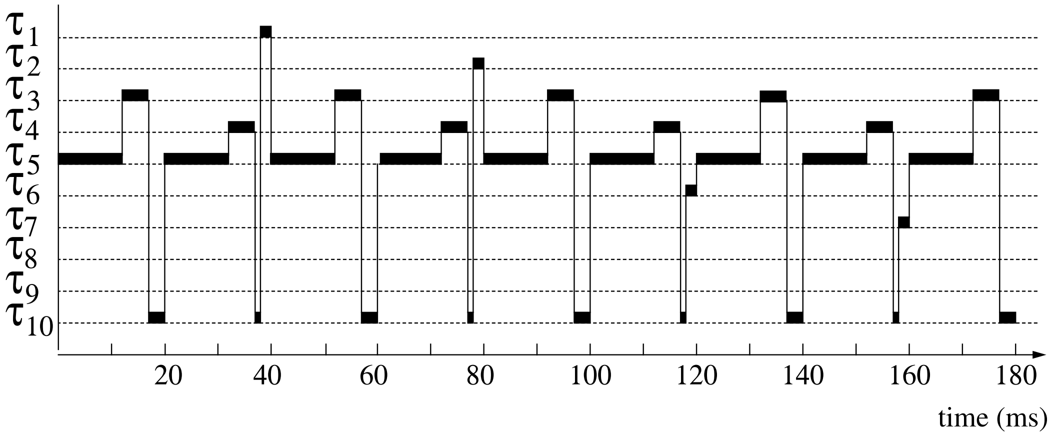

Figure 18: Timeline for Example Task Set (Modified Schedule)

issues since these periodic tasks are generally not preempted, i.e. the programmer assumes that he has exclusive control of the resources he needs for the duration of each computation. Thus, the scheduler cannot preempt the tasks at arbitrary points, and the program must be rewritten with explicit protection for critical regions if preemption is to be allowed. At any rate, requiring preemption in a particular task may complicate the programming of the task and definitely adds complexity to the overall system design.

3.3. Advantages and Disadvantages

The primary advantages of the cyclic executive approach are that it is simple to understand and simple to implement, it is efficient, and it is predictable. It is efficient because scheduling decisions are made off-line during the design process rather than during runtime. Thus context switching between computations is very fast. Context switches may be embedded in compiler-generated code or they may be specified by a table associated with the current major schedule and frame. Resource constraints and precedence constraints can also be embedded in the pre-computed schedule, so no overhead is incurred at runtime for synchronization. The timing of the schedule is easily verified at runtime by checking for frame overruns, but as long as the execution times of the frames were measured accurately during the design phase, the behavior of the system is predictable.

There are three areas where problems arise with the cyclic executive approach: design, runtime, and maintenance. Maintenance is regarded as the worst problem [15]. In the design process, scheduling and task-splitting were already identified as problem areas. The handling of frame overruns is another area where there are many choices that must be evaluated including policies such as immediate termination of the frame, suspension of the frame for later background processing, or continuation of the frame at the expense of the following frame. Mode changes also present a difficult design problem. Given that a mode change is provided in the design, after the need for a mode change is recognized during the execution of the system, the problem is to determine the appropriate time to make the change: immediately, after the current computation, after completion of the current frame, or after completion of the major schedule. Sporadic processes also present a problem. Guaranteeing fast response to sporadic processes requires a pessimistic reservation of processor resources while the alternative is slow or widely varying response time.

At runtime, the system is somewhat inflexible. It cannot generally adapt to a dynamically changing environment, at least for fine-grain changes. Long-term changes in environment can be accommodated using mode changes, allowing the system to adapt to the new conditions. Runtime efficiency may suffer if excessive resources are reserved for infrequent sporadics or if computations could not be assigned to frames so as to fully utilize the minor cycle. Such internal fragmentation in the minor cycle may result in significant unusable idle time.

System maintenance is complicated by the fact that the code may reflect job splitting and sequencing details of the schedule. Organization of the code around timing characteristics instead of around functional lines makes it that much more difficult to modify. This lack of "separation of concerns" can make program modifications very difficult, although sophisticated compilers and scheduling tools have been suggested as a means to address this problem [10]. Furthermore, the schedule that is produced during the design process is usually not documented in the code. And even though there is usually a separate document describing the details of the final schedule, details about the methods and criteria used to construct that schedule are typically omitted from the design documents.

4. Deterministic Scheduling

Deterministic scheduling, a branch of the larger field of combinatorial optimization, provides methods for constructing schedules in which the assignment of tasks to processors is known exactly for each point in time. The exact timing characteristics of the tasks must be known *a priori*. This includes arrival times, ready times, computation times, and deadlines. With this information, a scheduling algorithm can produce a precise schedule which optimizes one of several different measures. Work in deterministic scheduling theory has been in progress since the 1950's, and the topic has been of concern in operations research, industrial management, and more recently in computer science. Many of the original results were intended for use in a factory job shop or flow shop. In this situation, the time it takes to service the tasks is often very large compared with the time it takes to do the analysis and produce the schedule. In contrast, computer schedules must usually be produced in the same time frame as the execution of the actual tasks. Therefore, deterministic scheduling results may be of limited practical value in computer scheduling, especially in online computer scheduling. We will, however, explore the application of this area of scheduling theory to computer scheduling and especially to real-time computer scheduling. We will also consider some basic principles which have emerged from study in this area.

Several introductory articles and books are available which deal with the approach and results available in deterministic scheduling theory. The book by Conway *et al.* [8] is the first collection and summary of the results of scheduling theory, both deterministic and stochastic. This book is set in the language of industrial management although application to computer scheduling is straightforward. The book by Baker [1] also provides an introductory treatment of deterministic scheduling. And the survey by Gonzalez [12] provides an introductory description of computer applications of deterministic scheduling theory. Gonzalez does a nice job of introducing the topic and of briefly presenting many of the fundamental results. He does not, however, provide a clear classification of deterministic scheduling problems and results, although he includes a table of references and an indication of their applicability. The survey by Graham *et al.* [13] of deterministic scheduling results provides both an effective classification of deterministic scheduling problems and a broad survey of results. The introductory article by Blazewicz [3] also uses the classification system presented by Graham et al. [13]. Blazewicz takes a more tutorial approach to presenting scheduling results. And he also introduces areas for further research. Other comprehensive surveys in this area include the collection edited by Coffman and Bruno [7] and the survey of recent results by Lawler *et al.* [21] which is a revised version of the survey by Graham *et al.* [13].

The periodic scheduling results of Liu and Layland [28] and Serlin [39] which are summarized in [12] are classified as deterministic scheduling results and have significant practical value. The rate monotonic scheduling algorithm described by Liu and Layland [28] (and in different terminology by Serlin [39]) is of practical use in some types of process control systems. In this environment, the task requirements are periodic and hence deterministic. But rather than produce an explicit schedule, a priority assignment is all that is required to achieve the desired behavior (all deadlines are met). The major drawback of this approach is due to the number of restrictive assumptions placed on the task set, but this algorithm has served as a fruitful starting point for additional work which is aimed at relaxing the assumptions (see Section 5).

4.1. Concepts and Basic Notation

We now consider the basic concepts and notation. Graham *et al.* [13] and Blazewicz [3] describe the scheduling environment in their survey articles on deterministic scheduling. In deterministic scheduling, we assume that there are n tasks $J_j, j = 1, \ldots, n$ and m processors $M_i, i = 1, \ldots, m$. In the scheduling literature, the terms “task” and “job” are often used interchangeably, although in some cases, tasks are decomposed into separate parts called jobs. We will use the word task except where we must distinguish between jobs and tasks as collections of jobs. There may be many jobs in a single task, and each job may be composed of several operations. Each processor may work on a single task at a time, and each task may be processed by a single processor at a time. The schedule is a list of tasks along with the times when the tasks are placed on processors or taken off of processors, and a feasible schedule satisfies the timing requirements as well as the fundamental assumptions described above.

The processors are in one of two configurations: parallel or dedicated (specialized). In the case of parallel processors, we distinguish between identical processors, uniform processors, and unrelated processors. Identical processors have speeds which are constant and which do not depend on the task in service. Uniform processors have constant speeds but the speeds of individual processors may be different; the processor speed does not depend on the task. With unrelated processors, the speed depends on the task. In the case of dedicated processors, we distinguish between flow shops, open shops, and job shops. This nomenclature is taken from the industrial management literature. In the flow shop, each task is processed by all processors in the same order. In the open shop, each task is processed by all processors, but the order of processing in arbitrary. In the job shop, each task is processed by an arbitrary subset of the processors, and the order is arbitrary. But the specification of the subset and the order is fixed *a priori*.

Tasks may be characterized by their various properties as well. In the following explanation of notation, we use lower case letters for *a priori* task properties and upper case letters for task properties which arise from a particular schedule; this notation is based on that of Conway *et al.* [8] and Graham *et al.* [13]. Each task J_j has the following properties:

- the task has a vector of processing times with each element of the vector corresponding to the processing on a particular processor, $[p_{j1}, p_{j2}, \ldots, p_{jm}]$,
- the task has an arrival time or ready time, r_j,
- the task has a due date or deadline, d_j,
- the task has a weight or priority, w_j,

- the task may be preemptive or non-preemptive, depending on whether preemption is allowed in the schedules (preemption is also referred to as "task splitting"), and
- the task may be dependent or independent. Dependence between tasks is specified by means of a precedence tree or a more general precedence graph.

And now we can define what we mean by schedule a bit more precisely. A schedule is an assignment of processors to tasks. At each moment, at most one task is assigned to each processor, and at most one processor is assigned to each task. Each task is processed after its arrival time, and all tasks are completed. In addition, the precedence constraints are honored in the schedule. If the tasks are non-preemptive, then no preemptions occur in the schedule; if the tasks are preemptive, the schedule may contain a finite number of preemptions. We note that certain characteristics of the tasks, such as whether the tasks are preemptive or non-preemptive, describe the mode of service and are not properties of the individual tasks. We regard this as a weakness of current deterministic scheduling results. A more general approach of allowing preemptive and non-preemptive attributes for each task or for each operation in each task would give us a better theoretical foundation for analyzing the performance of actual systems. And we note that the property of dependence is a property of the entire task set rather than a property of the individual tasks, although the exact details of the dependences are specified on a per-task basis.

Now that we have classified the processor set and tasks along several dimensions, we can consider the performance characteristics and performance measures of individual tasks and of schedules. Each task in a schedule has

- a completion time which we denote as C_j,
- a flow time, denoted $F_j = C_j - r_j$,
- a lateness, denoted $L_j = C_j - d_j$,
- a tardiness, denoted $T_j = \max(L_j, 0)$, and
- a unit penalty U_j = if $C_j \leq d_j$ then 0, else 1.

Schedules are evaluated using

- schedule length or makespan,

$$C_{\max} = \max(C_j),$$

- mean flow time,

$$\overline{F} = \frac{1}{n}\sum_{j=1}^{n} F_j,$$

- mean weighted flow time,

$$\overline{F}_w = \frac{\sum_{j=1}^{n} w_j F_j}{\sum_{j=1}^{n} w_j},$$

- maximum lateness,

$$L_{\max} = \max(L_j),$$

- mean tardiness,

$$\overline{T} = \frac{1}{n}\sum_{j=1}^{n} T_j,$$

- mean weighted tardiness,

$$\overline{T}_w = \frac{\sum_{j=1}^{n} w_j T_j}{\sum_{j=1}^{n} w_j},$$

- number of tardy tasks,

$$\overline{U} = \sum_{j=1}^{n} U_j.$$

These properties of schedules not only provide measures for evaluating schedules, but also provide criteria for optimization in algorithms that produce schedules.

We find that, although for some simple or restricted problems an optimal solution exists, most problems, especially the more practical problems, are very difficult. Many of the optimization problems have been shown to be NP-hard; Lenstra *et al.* [25] give a brief introduction to some of the important ideas in complexity. To try to find approximate solutions to practical problems, various techniques are used which typically relax some of the assumptions that make the problem difficult. For example, it is easier to produce a schedule if preemption is allowed than if preemption is not allowed. So one technique for approximation is to allow preemption in a problem specified as non-preemptive. In this way, the basic shape of the preemptive schedule can be used to construct a schedule with no preemption. Another technique is to schedule unit length tasks. Another possibility is to use a precedence chain or precedence tree instead of a more general precedence graph. Approximation algorithms are also available which help to organize the search for a solution to a problem, avoiding the complete enumeration. And another possible technique is to just use an enumerative algorithm if the problem is sufficiently small.

4.2. Example Problem

We consider the following scheduling problem: a set of tasks are given with $\forall j, r_j = 0$ and with arbitrary processing times. Conway *et al.* show that when scheduling against a regular measure of performance (one that can be expressed as a function of task completion times) on a single processor, it is not necessary to consider schedules with preemption or with inserted idle time [8]. So the scheduling problem is reduced to constructing an ordering of the tasks.

Consider the graphical representation shown in Figure 19 of a schedule on a single processor. Each block corresponds to the computation time for a task and the lengths of the blocks correspond to the computation time. All of the blocks have unit height, so the area of each block is equal to the computation time for the associated task. And the tasks are listed in order of execution with $\tau_{[1]}$ indicating the first task to run, $\tau_{[2]}$ the second, and so on.

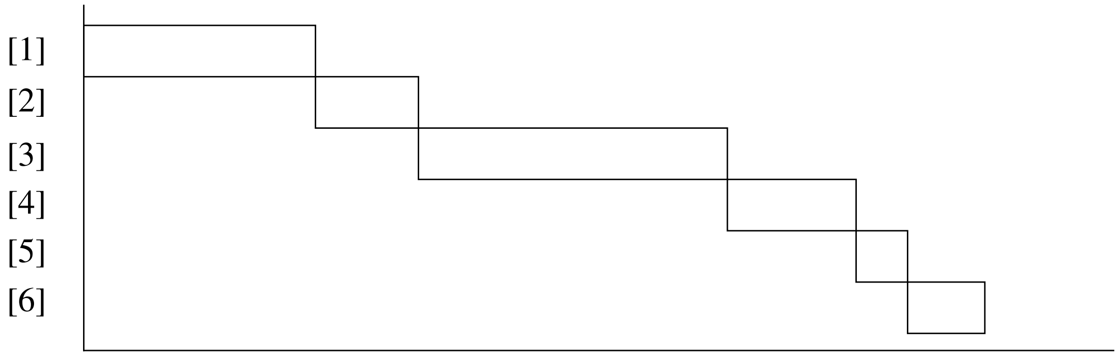

Figure 19: Arbitrary Sequencing

The total area of this graph including the area enclosed by the blocks and the area under the blocks is the sum of the flow-times. Minimizing the sum of the flow times also minimizes the mean flow time. And since the area of the blocks is the same in any schedule, we must minimize the area under the blocks to minimize mean flow time.

If we represent each block by a vector on its diagonal (as shown in Figure 20), a schedule is a sequence of such vectors laid out head to tail. We can minimize the area under the vectors by lining the vectors up to form a convex curve. This is done by putting the vector with the most negative slope first, the next most negative slope second, etc. That is, we order the jobs to satisfy

$$-1/p_{[1]} \leq -1/p_{[2]} \leq \cdots \leq 1/p_{[n]},$$

or equivalently,

$$p_{[1]} \leq p_{[2]} \leq \cdots \leq p_{[n]}.$$

Thus, we sequence the tasks in order of non-decreasing processing time; this is known as shortest-processing-time sequencing. Figure 20 shows the shortest-processing-time sequencing of the previous task set.

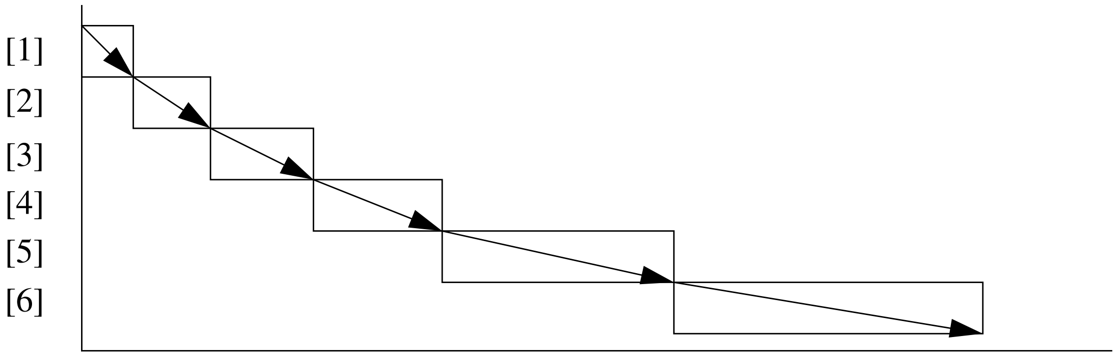

Figure 20: Shortest-processing-time Sequencing

In summary, we have illustrated the result that when scheduling jobs on a single processor, the mean flow-time is minimized by scheduling jobs in order of non-decreasing processing-time [8].

4.3. Problem Specification Notation

With this basic introduction, we now turn to a more precise discussion of the notation used to describe deterministic scheduling problems. Graham *et al.* described this notation [13], and Lawler *et al.* [21] used it subsequently as did Blazewicz [3]. The notation resembles the notation used for queueing system classification [18] and the notation used previously in the deterministic scheduling literature [8]. There are three fields in the problem specification separated by vertical lines or bars:

$$\alpha \mid \beta \mid \gamma.$$

here α is a string describing the processor environment (uniprocessor, multiprocessor, job shop, etc.), β gives other details about the scheduling environment (preemption, precedence constraints, resource constraints, etc.), and γ specifies the criterion for optimization. Each of these three fields has subfields which may or may not include a symbol. The symbol ϕ is used to indicate the case where no symbol appears in a subfield; this is usually a "default" case.

The following table gives the subfields for each of the three string and specifies the values that each subfield can take. This notation summarizes the points that were raised in the previous discussion, and examples of the notation are presented following this table.

$\alpha = \alpha_1\alpha_2$

$\alpha_1 \in \{\phi, P, Q, R, F, J, O\}$

For $\alpha_1 \in \{\phi, P, Q, R\}$, each task J_j has a single operation to be executed on processor M_i, and the processing time is p_{ij}.

$\alpha_1 = \phi$:	single processor, $p_{1j} = p_j$, the index for the processor is dropped since there is only one
$\alpha_1 = P$:	identical parallel processors, $p_{ij} = p_j(i = 1, \ldots, m)$, the index is dropped since the processing time does not depend on the actual processor used
$\alpha_1 = Q$:	uniform parallel processors, $p_{ij} = q_i p_j(i = 1, \ldots, m)$, where q_i is the *speed factor* of processor M_i which indicates its (constant) relative processing speed
$\alpha_1 = R$:	unrelated parallel processors
$\alpha_1 = F$:	flow shop, each J_j is composed of a *chain* of operations $\{O_{1j,\ldots,mj}\}$ where O_{ij} is to be processed on M_i for the computation time p_{ij}, the order in which operations are serviced is fixed by the ordering of the chain
$\alpha_1 = J$:	job shop, each J_j is composed of a *chain* of operations $\{O_{1j,\ldots,m_jj}\}$ where O_{ij} is to be processed on processor μ_{ij} for the computation time p_{ij} and $\mu_{i-1,j} \neq \mu_{ij}(i = 2, \ldots, m_j)$, the order in which operations are serviced is fixed by the ordering of the chain, but the order may be different for different tasks
$\alpha_1 = O$:	open shop, each J_j is composed of a set of operations $\{O_{1j,\ldots,mj}\}$ where O_{ij} is to be processed on M_i for the computation time p_{ij}, the order in which operations are serviced is arbitrary

$\alpha_2 \in \{\phi\} \cup \mathcal{N}$ where $\mathcal{N}$ is the set of natural numbers (positive integers)

$\alpha_2 \in \mathcal{N}$:	m is constant and equal to α_2
$\alpha_2 = \phi$:	m is variable

Note that when $\alpha_1 = \phi$, $\alpha_2 = 1$.

$\beta = \beta_1\beta_2\beta_3\beta_4\beta_5\beta_6$

$\beta_1 \in \{pmtn, \phi\}$

$\beta_1 = pmtn$:	preemption is allowed in the servicing of the tasks
$\beta_1 = \phi$:	preemption is not allowed

$\beta_2 \in \{res, res1, \phi\}$

$\beta_2 = res$:	the system includes s resources, R_h, $(h = 1, \ldots, s)$ in addition to the processor; Each task J_j needs r_{hj} units of resource R_h for the duration of its service
$\beta_2 = res1$:	the system includes a single additional resource (s = 1)
$\beta_2 = \phi$:	no additional resources are required by the tasks

$\beta_3 \in \{prec, tree, \phi\}$

$\beta_3 = prec$:	a (general) precedence relation exists between the tasks
$\beta_3 = tree$:	a precedence tree describes the precedence relation between tasks
$\beta_3 = \phi$:	there is no precedence relation for the tasks; the tasks are independent

$\beta_4 \in \{r_j, \phi\}$

$\beta_4 = r_j$:	arrival times are specified for each task
$\beta_4 = \phi$:	$r_j = 0$, (j = 1, ..., n); all tasks are released at the same time

$\beta_5 \in \{m_j \leq \overline{m}, \phi\}$

$\beta_5 = m_j \leq \overline{m}$:	$\overline{m}$ is a constant upper bound on m_j, the number of processors needed for the operations of a task (only in the job shop, $\alpha_1 = J$
$\beta_5 = \phi$:	there is no bound on m_j

$\beta_6 \in \{p_{ij} = 1, \underline{p} \leq p_{ij} \leq \overline{p}, \phi\}$

$\beta_6 = p_{ij} = 1$:	unit processing time
$\beta_6 = \underline{p} \leq p_{ij} \leq \overline{p}$:	processing time is bounded by constants $\underline{p}$ and $\overline{p}$
$\beta_6 = \phi$:	no bounds on processing time

$\gamma \in \{C_{\max}, L_{\max}, \sum C_j, \sum w_jC_j, \sum T_j, \sum w_jT_j, \sum U_j, \sum w_jU_j\}$

4.4. Problem Examples

We now consider some examples of scheduling problems specified using the notation described above. Graham *et al.* discuss these examples and others, giving a brief outline of efficient algorithms where possible [13]. The references given for the following problems were cited by Graham *et al.* as sources for the original solutions or complexity proofs.

$1\|\|L_{\max}$	On a single processor we wish to minimize the maximum lateness with no preemptions, no precedence constraints (independent tasks), and release dates at $t = 0$. Jackson's rule solves this problem: schedule the tasks using the earliest deadline first selection policy [16].
$1\|prec\|L_{\max}$	On a single processor we wish to minimize the maximum lateness, $L_{\max}$, subject to general precedence constraints on the tasks. We note that the implicit details of this specification include: there is no preemption, tasks are all released at time $t = 0$, and there are no additional resources. There is a polynomial time algorithm for this problem [20].
$1\|\|\sum w_j C_j$	On a single processor, minimize the sum of weighted completion times with no preemptions, no precedence constraints, and equal arrival times. The solution is to use Smith's rule [41] to schedule tasks in order of non-increasing ratios w_j/p_j.
$1\|\|\sum U_j$	On a single processor, minimize the number of tasks that miss their deadlines (no preemptions, no precedence constraints, and equal release times). Moore provided the solution to this problem [33]: schedule tasks in earliest-deadline-first order, removing the scheduled task with the largest processing time when the most recently added task fails to meet its deadline.
$P2\|\|C_{\max}$	On 2 identical parallel processors, minimize maximum completion time (with no preemptions, equal release times, and no precedence constraints). This problem is NP-hard [4, 26].

$P|pmtn|C_{\max}$ — On identical parallel processors, minimize maximum completion time with preemptions allowed (with equal release times and no precedence constraints). McNaughton gave a simple $\mathcal{O}(n)$ algorithm [31].

$J3|p_{ij}=1|C_{\max}$ — In a 3-machine job shop, we wish to minimize maximum completion time where tasks have unit processing time. Preemptions are not allowed and tasks are all released at time $t = 0$. This (simply specified) problem is NP-hard [25].

4.5. Scheduling in Practice

The results from deterministic scheduling theory are of limited value in practical operating system scheduling. First of all, the computation times of tasks are usually not known in advance. Secondly, the time to produce a schedule is usually large compared with the time it takes to service the tasks. For factory scheduling, this is not a problem, but it is a serious problem for short-term computer scheduling. Lastly, tasks arrive dynamically, requiring a new, updated schedule to be computed (for optimal performance). Most deterministic scheduling algorithms are not incremental in nature and do not easily accommodate new tasks.

These problems are addressed to some degree by queueing theory [8] which allows a probabilistic specification of computation times and arrival times. Queueing systems usually use a selection discipline which is easier to implement than a deterministic schedule. Selection disciplines include FCFS, shortest-processing-time (SPT), and fixed priority, and selection is efficient, usually based on a single comparison. And since the arrival times can be probabilistic, dynamic arrivals are easily accommodated by the system. This does not mean that the selection discipline will produce an optimal schedule. On the contrary, it has been shown [8] that queueing systems produce only suboptimal sequences compared with deterministic scheduling algorithms with perfect knowledge.

Despite the fact that deterministic scheduling algorithms are of limited use for online scheduling, many principles of scheduling have emerged which guide the system designer in developing an online scheduling algorithm. These principles are general although proofs of some of the claims require somewhat restrictive assumptions; they are treated in more detail by Conway *et al.* [8] and Gonzalez [12].

- Short-processing-time (SPT) sequencing is typically useful for minimizing mean flow time.

- Earliest-deadline-first scheduling tends to minimize maximum lateness.
- Schedules produced using the technique of processor-sharing are equivalent to preemptive schedules, and there is an algorithm to convert a general schedule to a preemptive schedule.
- Levels or precedence partitions are useful for constructing schedules involving tasks with precedence constraints (especially for precedence trees). The levels or partitions are ordered and each level contains tasks which are independent, forming a partial ordering of tasks.
- Multiprocessor anomalies may complicate scheduling under relaxed assumptions (such as variable computation time).
- Longest-processing-time (LPT) schedules tend to maximize mean flow time but minimize maximum finishing time on multiprocessors. (On uniprocessors, the finishing time is the same for every schedule.)
- Preemptive schedules are easier to produce than non-preemptive schedules (i.e. the complexity of the non-preemptive scheduling problem is greater than that of the preemptive scheduling problem).

5. Capacity-Based Scheduling

Several capacity-based algorithms have been developed which attempt to provide more flexibility while retaining the predictability of the cyclic executive method. In their pure form, capacity-based algorithms only require information about the amount of computation needed by a task set and the amount of computation available in the processing elements. This is in contrast to other deterministic scheduling algorithms which depend on exact timing information of tasks to make scheduling decisions. Liu and Layland's rate monotonic algorithm [28] (mentioned in Section 4) is an example of a capacity-based algorithm. Extensions to this algorithm, to be described later in this section, also retain some of the capacity-based flavor.

5.1. Rate Monotonic Analysis

Liu and Layland showed that, for a restricted class of real-time activities, it is possible to determine whether a task set is schedulable just by computing its utilization. The analysis depends on the following assumptions:

1. each task in the task set must be periodic,

Task set size (n)	Schedulable bound
1	1
2	.828
3	.780
4	.757
5	.743
6	.735
...	
∞	$\ln 2$

Table 2: Rate Monotonic Schedulable Bound

2. the deadline of a task is taken to be the end of the period for that task,
3. the computation time required of each periodic task must be constant,
4. the tasks do not communicate or synchronize with each other, and
5. the tasks can be preempted at any point in the computation; i.e. there are no critical regions in any of the computations.

To schedule a task set meeting these requirements, we use a fixed priority, preemptive scheduler. We order the tasks according to their frequency and then we assign integer priorities to the tasks based on this ordering with the highest priority assigned to the highest frequency task. This is called the rate monotonic priority assignment. We can predict whether a task set scheduled in this manner will meet all of its deadlines by computing the utilization of the task set and then comparing that utilization to a *schedulable bound.* A task set with utilization less than or equal to the schedulable bound is guaranteed to be schedulable. For task sets where the utilization is greater than the schedulable bound, there is no guarantee that the task set is schedulable. It may happen to be schedulable or it may not, and further analysis is required to make this determination. For the rate monotonic priority assignment, a task set $\tau_1, \tau_2, \ldots, \tau_n$ is schedulable if

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

where C_i is the computation time for a task of τ_i and T_i is the period for τ_i. Table 2 gives a few values for the schedulable bound for various task set sizes.

This bound is somewhat pessimistic, i.e. many task sets which have utilizations above the schedulable bound are actually schedulable. An exact criterion for determining whether a task set is schedulable was

Task	Priority	T		C		Description
τ_1	0	20	ms	12	ms	CD quality audio
τ_2	1	40	ms	5	ms	digital audio intercom
τ_3	1	40	ms	5	ms	digital telephone
τ_4	2	15	s	2	ms	smoke detector
τ_5	2	15	s	2	ms	motion detector
τ_6	2	15	s	2	ms	toilet overflow detector

Table 3: Example Task Set with Rate Monotonic Priority Assignment

given by Lehoczky *et al.* [23]. If we have a task set $\tau_1, \tau_2, \ldots, \tau_n$ that meets all of the above requirements and that is in priority order ($i < j \Rightarrow$ the priority of τ_i is higher than the priority of τ_j), then τ_i will meet all of its deadlines if and only if

$$\min_{0 \leq t \leq T_i} \sum_{j=1}^{i} \frac{C_j}{t} \left\lceil \frac{t}{T_j} \right\rceil \leq 1.$$

And if all the tasks in the task set meet this requirement, then the whole task set is schedulable. This criterion is piecewise linear and decreasing, so testing the condition at values of t which are multiples of the T_j is enough.

5.2. Example Analysis

To illustrate the use of the rate monotonic analysis, we consider our example task set. The first 6 tasks are periodic and we can assume, for now, that these tasks do not communicate or synchronize; we can assume that they use different devices. We assume that these computations can be preempted at any point and that the computation time required during each period is fixed for each task. We take the deadline of each task to be the ready time of the subsequent task in the same task. In sum, the first 6 tasks satisfy all of the conditions for the rate monotonic analysis.

First of all, we list the 6 tasks in priority order and rename them according to the new order. Using this order we can specify the rate monotonic priority assignment. The rate monotonic ordering and priority assignment is given in Table 3. Notice that priorities of different tasks with equal periods are also equal. This is not necessarily required; the rate monotonic results specify that ties may be broken arbitrarily. However, in practice, FIFO or round robin service within a priority class is preferred, especially in the lower priority classes where starvation may become an issue.

In order to determine whether this task set is schedulable or not, we first calculate the utilization and compare that with the rate monotonic schedulable bound for a task set of this size. We have

τ_i	t_{small}	$\sum_{j=1}^{i} \frac{C_j}{t_{small}} \left\lceil \frac{t_{small}}{T_j} \right\rceil$
τ_1	20.0	0.600
τ_2	20.0	0.850
τ_3	40.0	0.850
τ_4	40.0	0.900
τ_5	40.0	0.950
τ_6	80.0	0.925

Table 4: Example Task Set Exact Criterion Results

$$\sum_{i=1}^{n} \frac{C_i}{T_i} = \frac{12}{20} + \frac{5}{40} + \frac{5}{40} + \frac{2}{15000} + \frac{2}{15000} + \frac{2}{15000} = .8504$$

The schedulable bound for a task set of size 6 is (from Table 2) .735. Since the utilization is greater than the schedulable bound, we cannot conclude that the task set is schedulable. It may or may not be schedulable.

To find out whether this task set is schedulable, we are forced to use the exact criterion. The result of using the exact criterion algorithm for each task is shown in Table 4. The algorithm requires that we check for a particular condition at each period boundary of each task. For each task in the table, we have the smallest t (of the values that we check) at which the sum goes less than 1.0. So we are assured that

$$\min_{0 \leq t \leq T_i} \sum_{j=1}^{i} \frac{C_j}{t} \left\lceil \frac{t}{T_j} \right\rceil \leq \sum_{j=1}^{i} \frac{C_j}{t_{small}} \left\lceil \frac{t_{small}}{T_j} \right\rceil \leq 1.$$

And we conclude that these 6 tasks are indeed schedulable under the rate monotonic priority assignment.

But what about the rest of the tasks in our example task set? We still have three tasks that have not played any part in the analysis so far. In the cyclic executive scheduler, we used the idle time between periodic tasks to service the aperiodic tasks, and we can do the same thing with rate monotonic scheduling. There are other options for scheduling the aperiodics; these will be discussed in detail later.

5.3. Practical Extensions

The notion of schedulable bound is very powerful for measuring the performance of capacity-based scheduling algorithms in a real-time system. The central idea of these capacity-based algorithms is to determine the schedulability of a task set using a few parameters such as execution time, synchronization blocking time, communication blocking time, period and minimum interarrival time without getting into too many of the details about the actual software structure of each task.

The major drawbacks of the rate monotonic algorithm are that it assumes that tasks are independent (i. e. there is no communication or synchronization among tasks), periodic, completely preemptible and the tasks are all running on a single processor. There have been several attempts to find algorithms which fit into the framework of a rate monotonic scheduling policy to accommodate task sets which violate these assumptions.

Often, the activities in a real-time system must synchronize and communicate with each other. Unfortunately, the rate monotonic scheduling analysis techniques cannot predict the performance of a system with such tasks. This problem can be separated into two separate subproblems. One problem is the case where tasks are synchronizing on a single processor, and this includes tasks which are communicating on a single processor (using shared memory). The second problem is the case where tasks are communicating across some communication medium between two processors. Rajkumar developed synchronization algorithms and accompanying analysis techniques based on the rate monotonic scheduling paradigm in his PhD thesis [34, 35]. This work addresses the problem of uniprocessor synchronization and includes policies for managing semaphore queues and synchronization requests. The problem of end-to-end analysis with more than one processor connected by a communication medium is a topic of current research.

A special case of the synchronization problem described above is the mutual exclusion problem. Sometimes it is necessary to enforce mutual exclusion requirements, and usually this means that a task in a critical region cannot be preempted (at least not by other tasks sharing the same critical region). This issue is also addressed by the work of Rajkumar [34, 35].

The rate monotonic algorithm is limited to periodic tasks. The simplest way to support aperiodic tasks in this environment is to execute them in background mode or to periodically poll a queue of aperiodics to check for ready aperiodic tasks. The performance of aperiodics under a background server is poor since the aperiodics must wait until all of the periodic activity has completed before there is any idle time for background activity. In the case of the polling server, the performance is better, but there is still a problem if the aperiodic arrive just after the polling server checked the queue of aperiodics. When this happens, the aperiodic task is forced to wait until the next polling cycle before it has a chance to execute.

In an effort to improve the average response times of aperiodic activities while still retaining the predictability of the rate monotonic analysis, aperiodic server algorithms [24, 42, 43] have been developed and analyzed. These servers are periodic activities which provide service for aperiodic activities but are scheduled in the framework of the rate monotonic algorithm. The *deferrable server* [24, 43] has a period and a certain amount of execution time which it may allocate to aperiodic activities. This execution time can be thought of as a number of "tickets" that can be handed off to aperiodic activities which can then redeem the tickets for processor time. The deferrable server starts its periodic with a full allocation of tickets and when all the tickets have been given out, the server cannot allocate any more computation time to any more tasks. At the beginning of the next server period, the tickets are replenished, and the server can pass tickets to aperiodics. One difference between this type of periodic server and other periodic tasks in the rate monotonic framework is that typical periodic tasks are always ready to run during a period until the computation for that period is finished. On the other hand, this deferrable server only expends tickets or "runs" when an aperiodic request occurs. So it may wait until the end of its period before it demands its computation time. The *priority exchange server* works in much the same way except that computation time is traded among different priority levels. Both servers require that some capacity be reserved against the periodic tasks in the task set, but the amount of additional capacity required by the deferrable server is larger compared with that of the more complex priority exchange server.

The *sporadic server* [42] was developed to combine the best features of the deferrable server and priority exchange server. The sporadic server is the same as the deferrable server except that the ticket replenishment policy is different. Sprunt presented much of the background and motivation for the aperiodic server algorithms along with examples examples that illustrate their execution [42].

6. Dynamic Priority Scheduling

In dynamic priority scheduling, task priorities may change from request to request, or they may change dynamically over time within a single request. In static priority scheduling (also known as fixed priority scheduling), task priorities do not change once they have been assigned. We also draw a distinction between static priority scheduling and general deterministic scheduling, even though the rate monotonic algorithm is a deterministic scheduling algorithm which employs a static priority assignment. In general, deterministic scheduling algorithms produce sequences or schedules of tasks based on a complete, *a priori* definition of the task set. Priority scheduling, on the other hand, can be considered as a selection discipline for a queueing model. Priorities are used primarily in scheduling environments where tasks may arrive unpredictably, and in this case, deterministic scheduling results are not useful. The approach is to define a selection discipline which is responsible for making on-line scheduling decisions based on

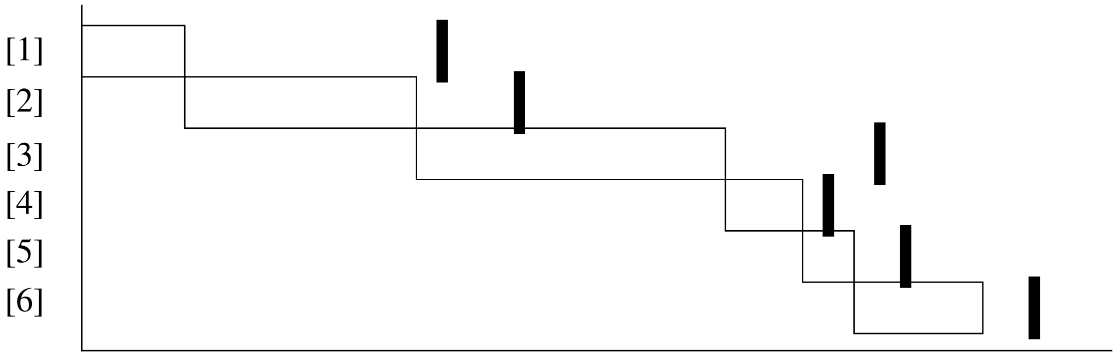

Figure 21: Almost Earliest-deadline Scheduling

the instantaneous state of the system.

6.1. Earliest-Deadline-First

One of the most popular dynamic priority scheduling algorithms is the earliest-deadline-first or earliest-deadline algorithm (also called due-date scheduling) which schedules the task with the closest deadline first. The algorithm has been discussed in many different scheduling contexts. Conway *et al.* [8] give a simple example of earliest-deadline-first in the context of a deterministic sequencing problem. They assume that tasks are all ready at time $t = 0$ and that the computations are non-preemptive. In this case, earliest-deadline-first minimizes the maximum lateness in the task set. This means that if it is possible to schedule the task set without missing any deadlines, earliest-deadline-first can find such a schedule.

In the context of a queueing system, the earliest-deadline-first algorithm is optimal in the following sense: if any algorithm can schedule a particular task set without missing any deadlines, the earliest-deadline-first algorithm can as well [9]. Dertouzos proved this using the following general technique. Consider the task set shown in Figure 21. All of the tasks meet their deadlines even though there is one pair of tasks which is not in earliest-deadline order ($\tau_{[3]}$ and $\tau_{[4]}$). The point where task $\tau_{[3]}$ is chosen to run is where the non-earliest-deadline selection is made. If task $\tau_{[4]}$ were to execute at that point instead of $\tau_{[3]}$, the computation block for $\tau_{[4]}$ would move to the left and the block for $\tau_{[3]}$ would move to the right. Moving $\tau_{[4]}$ to the left does not cause it to miss its deadline since it will complete sooner than it did before. Moving $\tau[3]$ to the right increases its completion time, but since it will complete before $\tau[4]$'s deadline and $\tau_{[4]}$'s deadline is before $\tau_{[3]}$'s deadline, $\tau_{[3]}$ will complete before its deadline as well. Thus, these two blocks are, in some sense, swapped, and this swapping operation does not cause any deadlines to be missed.

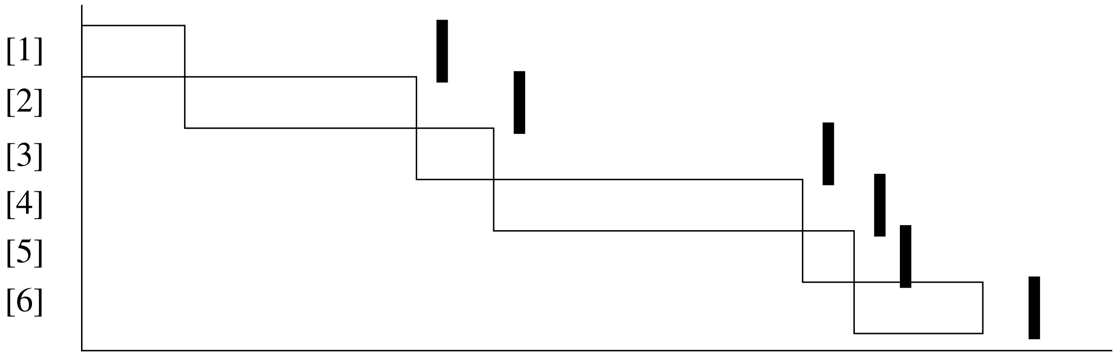

Figure 22: Earliest-deadline Scheduling

The resulting sequence is shown in Figure 22 which has the tasks renumbered according to their new order.

6.2. Least-Slack-Time

The slack-time of a task is the amount of time that the task can be delayed before it will miss its deadline. So the slack-time for task τ_i at time t is $d_i - p_i - t$. Conway *et al.* [8] consider the deterministic sequencing of tasks in order of nondecreasing slack-time (with all tasks ready at time 0 and with no preemption). They proved that least-slack-time scheduling maximizes minimum task lateness and minimum task tardiness. This is somewhat surprising; the idea of choosing the next task to execute based on information about which task seems to be in the most danger of missing its deadline is intuitively appealing. Minimum lateness, however, is not something that one would particularly want to maximize.

In spite of the limited usefulness of the above result for least-slack-time scheduling, the algorithm is optimum in the same sense as the earliest-deadline-first algorithm. Mok showed that least-slack-time scheduling is optimal in that if any algorithm can schedule a given task set with no missed deadlines, then the least-slack-time scheduling algorithm can as well [32]. The "time slice swapping" technique Mok used to prove this is the same as that used by Dertouzos in showing the optimality of earliest-deadline-first scheduling [9].

6.3. Practical Considerations

Dynamic priority schemes are not usually used in systems which require absolute predictability. The problem is that under overload, a scheduling algorithm should be able to select the most important tasks and execute them while discarding the less important tasks. However, most dynamic priority algorithms do not encode high-level importance information in the priority; the priorities typically reflect low-level, dynamically changing timing characteristics. In fact, dynamic priority scheduling was originally developed with an emphasis on assuring that even low-priority tasks eventually get their turn to execute [8]. Incorporating such a notion of fairness seems like a good idea for scheduling environments where all members of the population are equally important, but fairness has no place in scheduling to meet the deadlines of the most important tasks.

Dynamic priority schemes typically have slightly greater scheduling overhead than fixed priority schemes. This is because the range of dynamic priorities is usually greater than the range of static priorities, and dynamic priorities must also be recalculated at each decision point whereas static priorities never change and never have to be recalculated.

7. Value-Function Scheduling

Value-function scheduling policies use value functions associated with the tasks to make scheduling decisions. Each task has a value function which gives the value to the system (measured using some unit of utility) of completing a job of that task as a function of the time that the task has been active or ready. For example, value functions were used in Section 2 to illustrate various types of deadlines. The scheduler decides which job should execute by calculating the expected value of each ready task and applying some decision policy based on the calculated value.

The value-function approach seems attractive because of its expressive power, but computationally, scheduling using even the simplest types of value functions is intractable. McNaughton [31] analyzed simple models where tasks have deadlines and linear loss functions. In McNaughton's model, the loss of a task completed before its deadline is defined to be 0 while the loss after the deadline increases linearly with time. The rate of loss is a parameter of the task. With loss functions of this form, McNaughton proved under certain restricted conditions (to be explored in the following sections), a particular scheduling policy results in a schedule with minimal loss. Schild and Feldman extended the theory using similar models with quadratic loss functions [38]. Greenberger used a cost analysis for time-sharing systems [14]. These results from operations research tend to be off-line rather than on-line scheduling algorithms, although in some cases, the off-line analysis results in a simple scheduling rule that can be applied at run-time.

More recently, Jensen *et al.* explored the use of value functions in modern real-time computer operating systems [17]. They proposed and evaluated several different heuristic policies for making

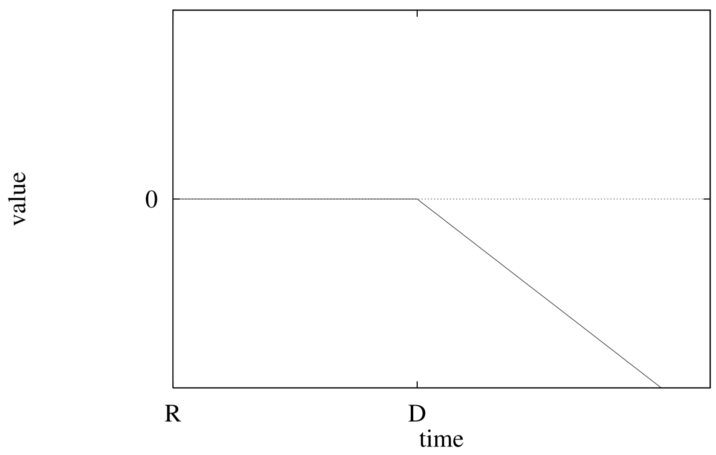

Figure 23: Linear Value Function

scheduling decisions based on the value of ready jobs. These scheduling policies were based on different heuristics and were evaluated by simulation. Tokuda *et al.* describe an implementation of several heuristics and give a performance evaluation of the their value-function approach [44] .

7.1. Value Functions

A loss function is a special case of a value function where the value of the associated task is constant until a particular point in time or deadline, and then the loss function determines the value of completing the function at a time after the deadline. In the simplest case, the loss function is linear although it could be quadratic or even a more complex function. For example, a task τ_i with deadline p_i and linear loss function $l_i(t)$ with penalty p_i would have $l_i(t)$ defined as:

$$l_i(t) = \begin{cases} 0 & \text{if } t < d_i \\ p_i(t - d_i) & \text{otherwise} \end{cases}$$

The negative of the loss function would be the value function which in this case would be defined as:

$$v_i(t) = \begin{cases} 0 & \text{if } t < d_i \\ -p_i(t - d_i) & \text{otherwise} \end{cases}$$

An example of a linear value function is shown in Figure 23.

Let a_i be the computation time required for task τ_i and let $r_i = p_i/a_i$. McNaughton showed that if m tasks are available to be scheduled and $d_1 = \cdots = d_m = 0$ and if the tasks are scheduled without splits in order of decreasing r_i, then the total cost of the schedule is minimized. That is to say that if all the tasks

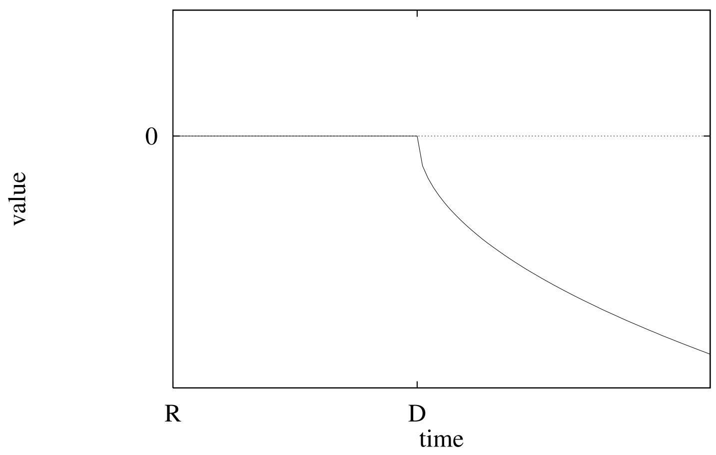

Figure 24: Quadratic Value Function

have a deadline of 0, indicating that they are all subject to their loss functions, and if the ordering is done according to his ratio rule, the loss of the schedule will be minimized.

He further proves that if a schedule is such that all tasks are scheduled in order of decreasing r_i, without splitting, with no idle time before the schedule is completed, and with no tasks completing before its deadline, then the schedule is minimal. This result has several major restrictions that limit its usefulness including the fact that no tasks are allowed to complete before their deadlines and loss functions must be linear. Generalizing along these lines leads to much harder problems as we shall see.

Loss functions can also be defined to be quadratic; in this case we have two penalty parameters for each task, p_i and q_i. The loss function can then be defined as:

$$l_i(t) = \begin{cases} 0 & \text{if } t < d_i \\ p_i(t-d_i)^2 + q_i(t-d_i) & \text{otherwise} \end{cases}$$

and the value function would be:

$$v_i(t) = \begin{cases} 0 & \text{if } t < d_i \\ -[p_i(t-d_i)^2 + q_i(t-d_i)] & \text{otherwise} \end{cases}$$

Figure 24 illustrates the form of this type of value function. This case was analyzed by Schild and Fredman [38] who proposed an exponential algorithm for computing minimal schedules.

Another formulation of the same problem was given by Lawler [19] who defined deferral cost functions with no deadlines, implying the $\forall i, d_i = 0$ restriction that McNaughton explicitly imposed. Lawler described dynamic programming and linear programming methods for solving the more general problem with nonlinear deferral cost functions and with multiple processors. The use of these types of algorithms is feasible when the time frame of the schedule is very large (much larger than the time required for the involved computation of the schedule) whereas these results are of little practical value in schedule a computer system where the computation time of the tasks are very small compared with the time required to compute a dynamic programming result.

In related work, Lawler and Moore [22] describe dynamic programming solutions to a variety of similar problems such as minimization of the weighted number of tardy jobs, maximization of weighted earliness, minimization of tardiness with respect to common relative and absolute deadlines, and minimization of weighted tardiness with respect to a common deadline. They also discuss some results for multiple processor scheduling.

More recently, real-time operating system designers have attempted to formulate scheduling heuristics based on value functions. Jensen, Locke, and Tokuda [17] proposed a canonical value function definition of the form:

$$v_i(t) = K_{i,1} + K_{i,2}t - K_{i,3}t^2 + K_{i,4}e^{-K_{i,5}t}$$

Each task τ_i has two sets of constants defining two value functions, one that is in effect before the critical time or deadline, and one that is in effect after the deadline. A few examples of the types of value functions that this formulation is designed to represent appear in Figure 25.

They simulated several scheduling algorithms with tasks defined using a value function of this form. In addition to some "classical" algorithms such as shortest-processing-time, earliest-deadline, least-slack, FIFO, random, and random-priority, they evaluated two heuristics. The first of their heuristics is exactly the scheduling policy of McNaughton, i.e. scheduling tasks in order of decreasing p_i/a_i. Their second heuristic involves using earliest-deadline with one modification; in the case of overload, the tasks with the least p_i/a_i ratio are discarded until the overload is relieved. In their simulations, tasks with different value functions were never mixed, so it is difficult to determine whether a general task set with arbitrary value functions could be effectively scheduled.

In related work, Tokuda *et al.* [44] modified the model and implemented it on the Mach operating system [36]. In this more practical work, they allowed only piecewise-linear value functions with three linear sections instead of two-section functions with the more general functions. Figure 26 shows some examples of the kind of piecewise-linear value functions that were considered. They evaluated many of the same scheduling algorithms that were used in the previous work: deadline-driven, least-slack, McNaughton's rule, and best-effort. They found that even the more efficient heuristics required a significant amount of scheduling overhead in actually executing a task set; the cost of making a single

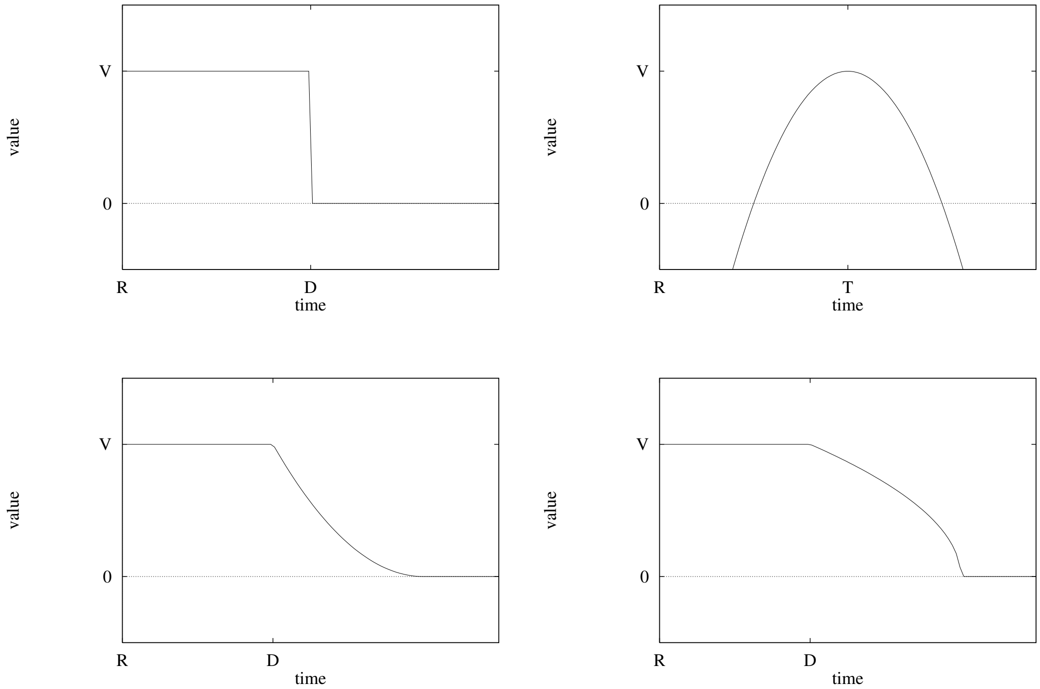

Figure 25: General Value Functions

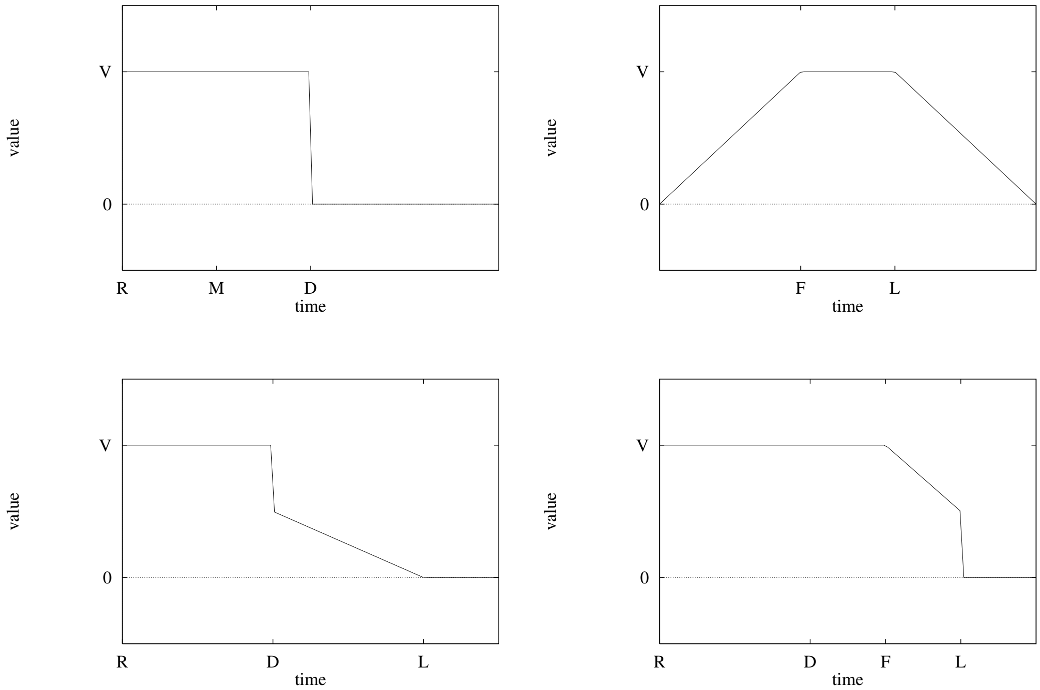

Figure 26: Piecewise-linear Value Functions

scheduling decision was sometimes more than the cost of actually executing several small tasks. This is in comparison to simple scheduling policies such as earliest-deadline for which the scheduling overhead is negligible.

7.2. Implementation Issues

It is clear from the above discussion of various value-function models, that only the very simplest models can be used efficiently; the more interesting models require exponential time for optimal solutions. Where sub-optimal heuristics have been employed, results depend on the parameters of the specific task sets that were used in simulation and the specific form of the value functions. And even in these cases, the scheduling overhead is so great that the value-function approach is unlikely to be useful in practice.

Another problem with the use of value functions in practice is determining the shape of the value function for each task in the system. With so much expressive power, finding an optimal or even a good value function is a formidable task.

8. Scheduling Tasks with Imprecise Results

The imprecise computation technique for scheduling real-time computations makes use of the fact that many computations are incremental in nature. These computations are such that a certain amount of processor time will produce a reasonably accurate result, and any additional processor time that can be allotted to the computation will refine the minimal result, increasing its accuracy. This approach to real-time scheduling is described in an introductory paper by Liu *et al.* [29] on which this section is based. Shih *et al.* [40] and Chung *et al.* [6] discuss other more detailed aspects of this work, and Lin *et al.* [27] describe a language implementation which provides constructs for real-time programming using imprecise results.

8.1. Basic Scheduling Environment

In order to use the imprecise computation technique, the system must be structured such that each task is divided into a mandatory subtask and an optional subtask. The scheduler guarantees that all of the mandatory subtasks will be completed by their deadlines, and the optional subtasks are scheduled in the remaining processor time to optimize some scheduling criterion. The optional subtasks may be aborted and discarded as appropriate; and some measure of error may be associated with any unfinished computation.

Liu *et al.* [29] classify different types of computations according to how they may be divided into subtasks as required by the imprecise computation method. The following describes their classification and uses some of their examples to illustrate the concepts. A *monotone* computation has the property that the quality of its result does not decrease as the computation proceeds, and the intermediate results of such a computation can be saved as they are produced. Examples of monotone computations include: numerical computations, statistical estimation and prediction, heuristic search, sorting, and database query processing [29]. If the mandatory computation is completed, the optional computation may be aborted and the intermediate result used at any time. This method of producing and saving intermediate results of monotone computations is called the *milestone* method.

Unfortunately, not all problems can be programmed as monotone computations. In the *sieve* method, certain parts of the total computation can be considered optional. For example, in a radar signal processing system, some tasks may require an estimate of the noise level in the received signal. The computation of the estimate may be considered optional since the previous estimates can be used in a pinch.

The *multiple version* method may be used when neither of the other two methods apply. In this case, a primary version of the task computes the precise result and an alternate version computes an imprecise (but adequate) result in a shorter period of time. The scheduler chooses between these two version based on the runtime situation. Liu *et al.* discuss these techniques in much greater detail, and they describe the limitations of each.

8.2. Formulation of Scheduling Problems

The mandatory subtasks are scheduling using traditional deterministic scheduling theory. The optional subtasks can be scheduled in many different ways. Liu *et al.* [29] describe various scheduling criteria that might be optimized such as minimization of total error, minimization of average error, minimization of the number of discarded optional tasks, minimization of the number of tardy tasks, and minimization of average response time.

9. Conclusion

In this paper, we have explored the nature of real-time computer systems in terms of the scheduling issues that arise in such systems. We introduced a computational model of tasks with timing constraint specifications for each computation, and we described some of the more practical methods that have been used to program real systems. The flavor of the methods presented in this paper is decidedly practical.

The cyclic executive has been used in many real-time systems, and deterministic scheduling theory has been used in building cyclic executives and has been applied in many other areas where time-constrained production or computation must be scheduled. The capacity-based approach to scheduling has recently been developed to alleviate some of the problems of the more inflexible design methodology based on deterministic scheduling. Dynamic priority scheduling algorithms provide the most flexibility, but they lack the predictability of deterministic scheduling and fixed priority scheduling. We have considered value-function scheduling because of the attractiveness of being able to define the importance of tasks using general, time-varying functions rather than fixed priorities, but the computational inefficiency of value-function scheduling severely inhibits useful application of this technique in practical systems. The imprecise computation approach combines some of the results of deterministic scheduling theory with simple value-function scheduling to schedule a certain class of computations more flexibly.

Acknowledgements

Thanks to: Mary Shaw, Nancy Mead, Carol Hoover, ...

10. Glossary

Many entries in this glossary contain cross-references to other entries. The cross-referenced entries may be duplicates of the original entry in which case the second entry is identified just to point out the alternative terminology; or the second entry may contain a definition of a different but related term.

abort to interrupt the execution of a task without the intention of restarting the task later.

action in the context of scheduling actions, action refers to a sequence of instructions or section of code which constitutes the computation for a task or for an instantiation of a task; the action may or may not be atomic.

aperiodic refers to events or arrivals which have no periodic pattern; the pattern may be deterministic or stochastic.

aperiodic task a task, instantiations of which may arrive in an aperiodic fashion.

aperiodic server a periodic task which is used to service aperiodic events or task instantiations; the purpose of the server is usually to reserve processor resources in the periodic framework for aperiodic tasks (see also deferrable server, sporadic server).

arrival characteristics details about the way that new tasks or computational entities become available for service in the system; this could include a period, a statistical distribution for time between adjacent arrivals, etc.

arrival process a way of describing the way that new tasks or computational entities become available for service in the system; arrivals could be periodic or statistically distributed

arrival time the time when a task becomes known to the system; may also be available for execution immediately upon its arrival.

background processing computations in a system which are not time critical and which may be afforded the lowest scheduling priority.

block in the context of scheduling blocks, block refers to a sequence of instructions or section of code which constitutes the computation for a task or for an instantiation of a task; the block may or may not be atomic.

capacity-based scheduling an approach to scheduling time-constrained tasks based on the resource utilization requirements of the tasks and the capacity of the resources.

catastrophic failure an error in a computer system (hardware or software) which leads to the destruction to the system itself or to the destruction of object or people in the physical world or both.

clock event an interesting change of the state of the clock (such as a timer expiration) which is propagated through some software mechanism to certain computational entities which have indicated interest in the state change.

completion time the absolute (as opposed to relative) time that a task completes the execution of its computation; depends on the schedule.

computation time the time that a task requires to execute on a particular processor; usually expressed as a worst case time or upper bound on the execution time.

criterion for optimization a property of a schedule that is to be minimized or maximized by a particular scheduling algorithm (see also optimization criterion).

critical region a sequence of instructions or block (or region) of code which is not to be interrupted by another task; more specifically, the task executing in the block is not to be interrupted by any task in an implicitly or explicitly specified set of tasks.

cyclic executive a supervisory program that controls the execution of application programs and schedules them using a pre-computed schedule that is repeated indefinitely.

cyclic executive model a programming model which is designed to reflect the requirements of cyclic executives, provide terminology for describing such systems, and provide some guidelines for the construction of such systems.

cyclic schedule a schedule which is intended to be repeated indefinitely.

deadline the time before which a task should have completed.

dedicated processors a collection of processors which execute independently but which may be specialized for a particular purpose; used in flow shop, job shop, and open shop scheduling analyses (see also specialized processors).

deferrable server a periodic task which is used to service aperiodic events or task instantiations according to a particular policy which attempts to minimize aperiodic response time; the purpose of the server is usually to reserve processor resources in the periodic framework for aperiodic tasks (see also aperiodic server, sporadic server).

deferral cost function a value function which describes the cost of postponing the processing of a particular task. This is similar to the loss function.

dependence refers to synchronization or communication requirements among tasks in a task set.

designer the person who determines the hardware and software components of a real-time computer system as well as the interactions between the computer system and the physical world.

deterministic scheduling the off-line sequencing or scheduling of tasks according to some optimization criterion for which all relevant information is known before the schedule is produced; parameters may include arrival time, computation time, deadline, weight, etc.

due-date scheduling a scheduling discipline which prefers tasks with earlier deadlines (due-dates) over tasks with later deadlines.

dynamic (priority) scheduling the on-line scheduling of tasks according to a task priority value which may change dynamically during the course of execution of the task or of the system (see also static priority scheduling).

earliest deadline first a scheduling discipline which prefers tasks with earlier deadlines over tasks with later deadlines.

enumeration refers to algorithms which consider each possible solution in a solution space individually; usually for optimization of some property of the solutions (see also enumeration).

enumerative algorithm refers to algorithms which consider each possible solution in a solution space individually; usually for optimization of some property of the solutions (see also enumeration).

executive a system program which provides the lowest layer of system support for application programs; a program which controls the execution of other programs (see also supervisory control program).

external interrupt a hardware or possibly software representation of an external event (see also external event).

external event an interesting change in the state of the physical world occurring at a particular point in time that results in some information generated by the physical world being received by a computer system.

FCFS scheduling first-come-first-served scheduling where tasks are serviced in order of their arrival.

feasible schedule a schedule in which all timing constraints are satisfied.

FIFO queueing first-in-first-out queueing where items are dequeued in order of the arrival in the queue.

fixed priority scheduling the on-line scheduling of tasks according to a task priority value which cannot change during the course of execution of the task or of the system; static priorities may be assigned at design time based on some appropriate scheduling analysis (see also static priority scheduling).

flight platform a real-time system (including physical world components as well as computer components) which controls the flight of some vehicle, usually containing the system itself (e.g. airplanes, missiles).

flow shop a scheduling environment where each task is processed by all of the processors in the environment and the tasks require the processors in the same order (see also job shop, open shop).

flow time the relative (as opposed to absolute) time that a task completes the execution of its computation; depends on the schedule; flow time is relative to arrival time or ready time.

frame in the cyclic executive model, the frame is the list of tasks that must be scheduled or the list of functions that must be called during a minor cycle; also known as the minor schedule (see also cyclic executive model, minor schedule).

frame overrun in the cyclic executive model, a frame overrun is a timing error which causes the computation of a frame or minor schedule to miss its deadline; the computation is not completed by the end of the minor cycle (see also frame, minor schedule, minor cycle).

hard deadline a deadline after which the value of completing a task becomes 0 or becomes $-\infty$; the task can be ignored (if the value becomes 0) or the system has experienced a catastrophic failure (if the value becomes $-\infty$); as opposed to a soft deadline.

hard real-time refers to real-time requirements, computations, systems, etc that involve hard deadlines (see also hard deadline).

horizon in defining timing constraints of tasks, arrivals of certain tasks may be predictable for the near future but unpredictable beyond that; the horizon is the time in the near future during which the arrivals are predictable.

identical processors a collection of processors which execute independently and which are identical; i.e. they have the same processing capabilities and the same processing speed (see also parallel processors, uniform processors, unrelated processors).

imprecise computation technique an approach to scheduling using imprecise computations which have a mandatory part and an optional part that refines the result produced in the mandatory part; computations with this form can be scheduled to optimize the total value of the optional parts.

imprecise result refers to an incremental computation with a mandatory part and an optional part that refines the result produced in the mandatory part; computations with this form can be scheduled to optimize the total value of the optional parts.

instantiation a single computational entity of a stream of computational entities that arise from the same arrival process.

interarrival time the duration between successive arrivals of some event.

interprocess communication the transfer of data of some kind from one computational entity to another including simple synchronization as a special case; may imply remote communication as well, depending on the context.

job a computational entity which is a single instantiation of a stream of computational entities which are generated by a particular arrival process.

job shop a scheduling environment where each task is processed by an arbitrary subset of the processors in the environment; the order of processing is arbitrary (see also flow shop, open shop).

job splitting the preemption of a task or job in a schedule, possibly causing other jobs to execute during the time between executions of the job being split.

lateness the difference between the completion time and the deadline of a task (positive or negative); the amount of time that a task overran its deadline.

least slack-time a scheduling policy which prefers tasks with smaller slack-times.

level with respect to precedence constraints, levels are collections of tasks which are independent (no precedence constraints within a level) with the property that all tasks in the levels "before" a level may all be executed before tasks in the given level, and all tasks in levels "after" the level may be executed after tasks in the given level.

load the fraction of the total capacity of a resource that is being utilized, e.g. the processor utilization of a task set is the ratio of the time the processor is active during a particular duration and the total duration (see also utilization).

longest processing time a scheduling discipline which prefers tasks with long processing time over tasks with shorter processing time; abbr. LPT.

loss function a value function which assumes non-positive values and thus describes the loss of value over time.

major cycle in the cyclic executive model, the major cycle is the duration of the major schedule that is to be repeated indefinitely (see also cyclic executive model, major schedule).

major schedule in the cyclic executive model, a major schedule is the entire list of tasks that must be scheduled; the major schedule is repeated indefinitely and is composed of smaller minor schedules or frames (see also cyclic executive model).

makespan the time between the beginning of the execution of the first task and the completion of the last task in a schedule (see also schedule length).

mandatory computation the part of a computation that is required for an imprecise result to attain its minimum precision; used in the context of scheduling with imprecise results.

maximum lateness for a particular task set and a particular schedule, the maximum lateness is the maximum amount by which any task missed its deadline.

mean flow time the average flow time of all the tasks in a schedule.

mean tardiness the average tardiness of all the tasks in a schedule.

mean weighted flow time the average flow time, weighted by task-dependent weighting factors, of all the tasks in a schedule.

mean weighted tardiness the average tardiness, weighted by a task-dependent weighting factor, of all the tasks in a schedule.

memory management a software component of a system which manages the allocation of physical and/or virtual memory.

milestone method a method for saving intermediate results in a monotone computation; used in imprecise computation technique.

minor cycle in the cyclic executive model, the minor cycle is the duration of the minor schedules or frames that are constituents of the major schedule; the major cycle is a multiple of the minor cycle (see also cyclic executive model, major schedule, minor schedule).

minor schedule in the cyclic executive model, the minor schedule is the list of tasks that must be scheduled or the list of functions that must be called during a minor cycle (see also cyclic executive model, frame).

mode refers to a set of assumptions or a set of requirements in a cyclic executive which defines one of several alternative cyclic schedules to be used for execution; changes in the environment to reflect different assumptions or different requirements require a mode change which implies the use of a different cyclic schedule.

mode change refers to a change in the set of assumptions or the set of requirements in a cyclic executive which define one of several alternative cyclic schedules to be used for execution; changes in the environment to reflect different assumptions or different requirements require a mode change which implies the use of a different cyclic schedule.

monotone computation a computation where the quality of the result does not decrease as the computation proceeds; used in imprecise computation technique.

multiple version method a method of structuring computations so that more than one algorithm is available to produce a particular results, and different algorithms have different running times and different accuracies; used in imprecise computation technique.

multiprocessor a processor environment with multiple processors.

network communication the transfer of data of some kind from one computational entity to another entity residing in a different processor on a network.

non-preemptible refers to a computation which is not susceptible to preemption or being preempted.

non-preemptive refers to scheduling algorithms, service disciplines, etc. which do not allow preemption among tasks.

$\mathcal{NP}$ the class of decision problems that can be solved in polynomial time by a non-deterministic Turing machine (see [11] or [25].

NP-complete a problem P is NP-complete if P is NP-hard and $P \in \mathcal{NP}$.

NP-hard a problem P is NP-hard if $\forall P' \in \mathcal{NP}$, P' reduces to P.

off-line refers to a computation which is done before the system that may need the result of the computation begins execution; a design-time or compile-time computation.

on-line refers to a computation which is done at the time that the system requires the result of the computation; a run-time computation.

open shop a scheduling environment where each task is processed by all of the processors in the environment; the order is arbitrary (see also flow shop, job shop).

optimization criterion a property of a schedule that is to be minimized or maximized by a particular scheduling algorithm.

optional computation the part of a computation that is optional in an imprecise result; refines the result obtained in the mandatory computation; used in the context of scheduling with imprecise results.

parallel processors a collection of processors which execute independently; usually taken to be identical unless otherwise specified (see also identical processors, uniform processors, unrelated processors).

periodic refers to events or arrivals which occur periodically, separated by a fixed duration of time.

periodic task a task, instantiations of which may arrive in periodic fashion.

physical system the physical environment in which a real-time computer system operates (in the solution of a physical-world problem).

piecewise linear refers to a continuous function which is composed of straight lines rather than sections of arbitrary-degree polynomials or exponential functions.

pre-computed schedule a schedule for which all scheduling events have been determined off-line and which has some representation suitable for executing the scheduling events during run-time.

precedence constraint a restriction on the order in which tasks may be executed; usually of the form task τ_1 must complete before task τ_2 begins execution.

precedence graph a general precedence relation with no restrictions on which tasks can have precedence constraints.

precedence partition with respect to precedence constraints, precedence partitions are collections of tasks which are independent (no precedence constraints within a level) with the property that all tasks in the levels "before" a level may all be executed before tasks in the given level, and all tasks in levels "after" the level may be executed after tasks in the given level.

precedence relation a specification of all of the precedence constraints that exist between all the tasks in a task set.

precedence tree a precedence relation where there are no cycles of dependence in a task set.

preemptible refers to a computation which is susceptible to preemption or being preempted.

preemption the interruption of an executing task in favor of a more another (usually more important) task; the second task begins executing after the state of the first is saved.

preemptive refers to scheduling algorithms, service disciplines, etc. which allow preemption among tasks.

preempt to interrupt an executing task in favor of another (usually more important) task.

priority a task attribute which defines the relative preference that tasks should receive in making scheduling decisions (see also weight).

priority assignment an assignment of priorities to tasks (see also rate monotonic priority assignment).

processor refers to a generic processing element or to a hardware device such as a microprocessor.

quantum the maximum duration that a task may execute before being preempted in favor of a waiting task or at least considered for preemption.

queueing system a collection of customers and service agents along with some specification of how each of these entities act intended to draw some conclusions or predictions of waiting time.

rate monotonic analysis the determination of whether a periodic task set can be scheduled without missing deadlines by using the rate monotonic priority assignment and a fixed priority, preemptive scheduler.

rate monotonic scheduling preemptive scheduling of periodic tasks using the rate monotonic priority assignment by which higher frequency tasks are given higher priority.

rate monotonic priority assignment an assignment of priorities to periodic tasks whereby higher frequency tasks are given higher priority.

ready time the time when a task becomes available for execution on some processor.

real-time application a program or collection of programs (usually supported by an underlying operating system or executive) that is used to solve a particular real-world problem.

real-time computer system a computer system (hardware and software) which is designed to support or to contain real-time applications.

real-time operating system the part of a real-time system which lies above the hardware and which supports a real-time programming model for the use of real-time applications (programs).

real-time system the entire collection of components that are designed to solve some physical world problem; the system includes the physical environment in which the problem is to be solved, the application software, the operating system software, and the underlying hardware.

round robin scheduling scheduling with a FIFO queue and with a quantum; executing jobs may execute for at most a quantum before being preempted and placed at the end of the queue.

schedulable refers to a task set for which a schedule can be found, specifically a schedule where no deadlines are missed can be found.

schedulable bound in rate monotonic analysis, the schedulable bound is a particular utilization bound where periodic task sets with total utilization less than the bound are guaranteed not to miss any deadlines; task sets with utilization that is greater than the bound are not guaranteed not to miss deadlines.

schedule a list of scheduling events along with the time that each event should occur and the relevant information about the action to take when the event occurs.

schedule length the time between the beginning of the execution of the first task and the completion of the last task in a schedule (see also makespan).

scheduler the logic or software within a system which controls the allocation of resources (usually processors) to tasks.

scheduling analysis the evaluation of a particular scheduling problem or scheduling technique or algorithm; typical results of the evaluation might be that all deadlines can be met under certain conditions, or that all deadlines cannot be met, or that some other optimization criterion can or cannot be optimized.

scheduling criterion a property of a schedule that is to be minimized or maximized by a particular scheduling algorithm.

scheduling events any changes in the allocation of tasks to processors; e.g. the event that a task arrives and/or becomes ready to run, the event that a task completes execution and relinquishes the processor, etc.

scheduling horizon in defining timing constraints of tasks, arrivals of certain tasks may be predictable for the near future but unpredictable beyond that; the scheduling horizon is the time in the near future during which the arrivals are predictable.

selection discipline a rule in a queueing system for choosing the next task for service.

semaphore a mechanism for synchronizing tasks.

shortest processing time a scheduling discipline which prefers tasks with short processing time over tasks with longer processing time; abbr. SPT.

sieve method a method of structuring computations that allows some computations to be skipped in certain cases; used in imprecise computation technique.

slack-time the amount of time a task can be delayed before it will miss its deadline.

soft deadline a deadline after which some value remains in completing the associated task; as opposed to a hard deadline.

soft real-time refers to real-time requirements, computations, systems, etc that involve soft deadlines.

software event an interesting change in the state of a computational entity occurring at a particular point in time that results in some information generated by the computational entity being transferred to another computational entity.

specialized processors a collection of processors which execute independently but which may be specialized for a particular purpose; used in flow shop, job shop, and open shop scheduling analyses (see also dedicated processors).

sporadic aperiodic; refers to an aperiodic task with hard deadlines (as opposed to an aperiodic task with soft deadlines).

sporadic server a periodic task which is used to service aperiodic events or task instantiations according to a particular policy which attempts to guarantee aperiodic response time; the purpose of the server is usually to reserve processor resources in the periodic framework for aperiodic tasks (see also aperiodic server, deferrable server).

starvation the situation in computer scheduling where a low priority task never reaches the head of the queue where it can execute; higher priority tasks are present indefinitely in the system.

static (priority) scheduling the on-line scheduling of tasks according to a task priority value which cannot change during the course of execution of the task or of the system; static priorities may be assigned at design time based on some appropriate scheduling analysis.

statistical distribution a mathematical function which gives the probability of occurrence of events that are described by the distribution.

supervisory control program a system program which provides the lowest layer of system support for application programs; a program supervises or controls the other programs.

synchronization a restriction on two or more computational entities in a system where the progress of the each entity is related in some way to the progress of the other(s); the entities may also be electrical or electronic rather than computational (synchronization bits on a network medium).

tardiness the difference between the completion time and the deadline of a task if that difference is positive (missed deadline) and zero otherwise.

task a computational entity in a scheduling model or in a programming model implementation; used to describe a generic computational entity or a stream of instantiations of a particular class of computational entity.

task set a collection of tasks that are intended to execute on the processor or set of processors and complete for resources.

task splitting the preemption of a task in a schedule, possibly causing other tasks to execute during the time between executions of the task being split.

time-slice swapping a proof technique where slices of computations are interchanged according to some rule which preferably improves (or at least doesn't adversely affect) some measure of performance.

timing constraints the specification of the timing attributes of a task including the arrival time, computation time, and deadline as well as other constraints such as precedence constraints, maximum blocking time, etc.

uniform processors a collection of processors which execute independently and which have constant speeds (the speed of a processor does not depend on the task it is executing) although the speeds of different processors may be different (see also parallel processors, identical processors, unrelated processors).

uniprocessor a processor environment with a single processor.

unit penalty a cost or penalty of 1 for a task missing a deadline in a particular schedule.

unrelated processors a collection of processors which execute independently and which have variable speeds, depending on the task being executed (see also parallel processors, identical processors, uniform processors).

utilization the fraction of the total capacity of a resource that is being utilized, e.g. the processor utilization of a task set is the ratio of the time the processor is active during a particular duration and the total duration (see also load).

value used in connection with the usefulness to the system or application of completing a particular task (at a particular point in time); a positive value indicates that the system would benefit from completing the task (as long as such completion doesn't preclude the completion of other tasks with higher values), a value of zero indicates that no benefit will be derived from completing the task, and a negative value indicates that it would be bad to complete the task.

value function a mathematical function that gives the value of completing an associated task as a function of the time since the task arrived in the system; used for value-function scheduling.

value-function scheduling the use of value functions for scheduling.

weight a task attribute which defines the relative preference that tasks should receive in making scheduling decisions (see also priority).

References

[1] K. R. Baker. *Introduction to Sequencing and Scheduling*. Wiley, 1974.

[2] T. P. Baker and A. Shaw. The Cyclic Executive Model and Ada. In *Proceedings of the 9th IEEE Real-Time Systems Symposium*, pages 120–129, December 1988.

[3] J. Blazewicz. Selected Topics in Scheduling Theory. *Annals of Discrete Mathematics*, 31:1–60, 1987.

[4] J. Bruno, E. G. Coffman, Jr., and R. Sethi. Scheduling Independent Tasks to Reduce Mean Finishing Time. *CACM*, 17:382–387, 1974. Cited by Graham *et al.* [13].

[5] G. D. Carlow. Architecture of the Space Shuttle Primary Avionics Software System. *CACM*, 27(9):926–936, September 1984.

[6] J. Y. Chung, J. W. S. Liu, and K.-J. Lin. Scheduling Periodic Jobs that Allow Imprecise Results. *IEEE Transactions on Computers*, 39(9):1156–1174, September 1990.

[7] E. G. Coffman, Jr. and J. L. Bruno, editors. *Computer and Job-Shop Scheduling Theory*. Wiley, 1976.

[8] R. W. Conway, W. L. Maxwell, and L. W. Miller. *Theory of Scheduling*. Addison-Wesley, 1967.

[9] M. L. Dertouzos. Control Robotics: The Procedural Control of Physical Processes. In *Proceedings of the IFIP Congress*, pages 807–813, August 1974.

[10] S. R. Faulk and D. L. Parnas. On Synchronization in Hard-Real-Time Systems. *CACM*, 31(3):274–287, March 1988.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[12] M. J. Gonzalez, Jr. Deterministic Processor Scheduling. *Computing Surveys*, 9(3):173–204, September 1977.

[13] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[14] M. Greenberger. The Priority Problem and Computer Time Sharing. *Management Science*, 12(11):888–906, July 1966.

[15] P. Hood and V. Grover. Designing Real Time Systems in Ada. Technical Report 1123-1, SofTech, Inc., January 1986.

[16] J. R. Jackson. Scheduling a Production Line to Minimize Maximum Tardiness. Technical Report Research Report 43, Management Science Research Project, UCLA, 1955. Cited by Graham *et al.* [13].

[17] E. D. Jensen, C. D. Locke, and H. Tokuda. A Time-Driven Scheduling Model for Real-Time Operating Systems. In *Proceedings of the 6th IEEE Real-Time Systems Symposium*, December 1985.

[18] L. Kleinrock. *Queueing Systems, Vol. 1: Theory*. Wiley Interscience, 1975.

[19] E. L. Lawler. On Scheduling Problems with Deferral Costs. *Management Science*, 11(2):280–288, November 1964.

[20] E. L. Lawler. Optimal Sequencing of a Single Machine Subject to Precedence Constraints. *Management Science*, 19:544–546, 1973. Cited by Graham *et al.* [13].

[21] E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Recent Developments in Deterministic Sequencing and Scheduling: A Survey, 1980. Also available in *Deterministic and Stochastic Scheduling* edited by M. A. H. Dempster, J. K. Lenstra, and A. H. G. Rinnooy Kan, Reidel, Dordrecht, 1982.

[22] E. L. Lawler and J. M. Moore. A Functional Equation and its Application to Resource Allocation and Sequencing Problems. *Management Science*, 16(1):77–84, September 1969.

[23] J. P. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pages 166–171, December 1989.

[24] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced Aperiodic Responsiveness in A Hard Real-Time Environments. In *Proceedings of 8th IEEE Real-Time Systems Symposium*, pages 261–270, December 1987.

[25] J. K. Lenstra and A. H. G. Rinnooy Kan. Computational Complexity of Discrete Optimization Problems. *Annals of Discrete Mathematics*, 4:121–140, 1979.

[26] J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker. Complexity of Machine Scheduling Problems. *Annals of Discrete Mathematics*, 4:281–300, 1977. Cited by Graham *et al.* [13].

[27] K.-J. Lin and S. Natarajan. Expressing and Maintaining Timing Constraints in FLEX. In *Proceedings of the Ninth IEEE Real-Time Systems Symposium*, pages 96–105, December 1988.

[28] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment. *JACM*, 20(1):46–61, 1973.

[29] J. W. S. Liu, K.-J. Lin, W.-K. Shih, A. C. Yu, J.-Y. Chung, and W. Zhao. Algorithms for Scheduling Imprecise Computations. *IEEE Computer*, 24(5):58–68, May 1991.

[30] C. D. Locke. Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives. *The Journal of Real-Time Systems*, 4(1):37–53, March 1992.

[31] R. McNaughton. Scheduling with Deadlines and Loss Functions. *Management Science*, 6(1):1–12, October 1959.

[32] A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. PhD thesis, Massachusetts Institute of Technology, May 1983.

[33] J. M. Moore. An n Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs. *Management Science*, 15(1):102–109, September 1968. Cited by Graham *et al.* [13].

[34] R. Rajkumar. *Task Synchronization in Real-Time Systems*. PhD thesis, Carnegie Mellon University, August 1989.

[35] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[36] R. F. Rashid, A. Tevanian, M.W. Young, D. B. Golub, R. V. Baron, D. L. Black, W. Bolosky, and J.J. Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *IEEE Transactions on Computers*, August 1988.

[37] A. E. Ritchie and L. S. Tuomenoksa. No. 4 ESS: System Objectives and Organization. *The Bell System Technical Journal*, 56(7):1017–1027, September 1977.

[38] A. Schild and I. J. Fredman. Scheduling Tasks with Deadlines and Non-linear Loss Functions. *Management Science*, 9(1):73–81, October 1962.

[39] O. Serlin. Scheduling of Time Critical Processes. In *Proceedings of AFIPS 1972 Spring Joint Computer Conference*, pages 925–932, 1972.

[40] W. K. Shih, J. W. S. Liu, and J. Y. Chung. Algorithms for Scheduling Imprecise Computations with Timing Constraints. *SIAM Journal on Computing*, 20(3):537–552, June 1991.

[41] W. E. Smith. Various Optimizers for Single-Stage Production. *Naval Research Logistics Quarterly*, 3:59–66, 1956. Cited by Graham *et al.* [13].

[42] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic Task Scheduling for Hard Real-Time Systems. *The Journal of Real-Time Systems*, 1:27–60, 1989.

[43] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments, 1990. Check citation. Unpublished Manuscript?

[44] H. Tokuda, J. W. Wendorf, and H.-Y. Wang. Implementation of a Time-Driven Scheduler for Real-Time Operating Systems. In *Proceedings of 8th IEEE Real-Time Systems Symposium*, December 1987.

Contents

List of Figures

List of Tables