Real-time in a Concurrent, Object-Oriented Programming Environment

by

Philipp Enrique Lim, Jr.

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 1996

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

The development and maintenance of most real-time systems has always been problematic. In most real-time and embedded systems, software development usually employs ad-hoc methods, utilizing assembly or machine specific languages.

Research on various algorithmic means for providing real-time program behaviour has been in existence since the 70's. Over the past two decades, real-time scheduling algorithms have gained tremendous popularity and maturity. However, a convenient means of utilizing such algorithms has lagged behind, given in part to the lack of a popular and flexible real-time programming language.

Though various real-time programming languages exist, such languages usually offer little with regard to flexibility in controlling the underlying real-time scheduling algorithm. This thesis discusses the extension and transformation of a general purpose programming language into a flexible programming language suitable for real-time program development.

Acknowledgements

I would like to thank the Institute of Computer Research for the scholarship and financial support.

Special thanks to Dr. Peter Buhr and Dr. Prabhakar Ragde for all the support, encouragement, and guidance. I remember the times filled with frustration and confusion, and the good moments too – you were there to make my "journey" much less difficult. Thank you.

Thank you to my parents, who have supported my every move in life, and who have always been there to foster me physically, emotionally, and intellectually.

Thank you to all my friends, both in Waterloo and all over the world, who have been there for me, and who have put up with my moanings, frustrations, and criticisms. This thesis would not have been completed without all your support, as well as your inspirations. In your own way, you were all an inspiration to me, and each and everyone of you has a special place in my heart.

Most of all, to God – who has blessed each and everyone of us with the potential to be the very best.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Real-time computing plays a crucial role in many important computing applications. Real-time computing, also known as time-constrained computing, is distinct from the normal computing paradigm in that the behaviour of the whole system is dictated by deadlines. The correctness of a real-time system is not only gauged by the correct logical output of a computation, but also in the timely completion of the computation.

Though real-time computing can be classified into various categories, the most general classification or distinction can be drawn between hard versus soft real-time. Many real-time practitioners define hard real-time systems as computing environments involving rigid timing constraints, such that a missed deadline spells catastrophic results. The control and monitoring of a nuclear power plant, for example, is a hard real-time system, for an untimely computational result or response can lead to meltdown or release of radioactivity. In turn, a matching definition of a soft real-time system is a computing environment also sharing timing constraints;

however, missing some or all of the constraints does not necessarily imply disastrous consequences. A looser and, perhaps, a more common definition of a hard real-time system is a computing environment in which time constraints need to be very strictly adhered to, and any lapse is unacceptable. In contrast, in a soft real-time system an occasional missed deadline is sometimes acceptable.

Whether hard or soft real-time, most industrial real-time applications are still a generation behind the current technologies offered in today's research real-time systems, in which systematic and algorithmic approaches of system development and maintenance exist. As a result, most industrial real-time application development still utilize non-real-time-specific tools, which then implies the use of ad-hoc methods, making development and maintenance tedious and difficult.

The challenge in this project is to extend an existing programming language and its implementation to a *flexible* real-time development system. To achieve this goal, a general-purpose programming language is extended to provide facilities for the effective expression of the notion of time. Furthermore, constructs are introduced that enable the explicit expression of any applicable time-constraint and time property of a program.

In the area of language implementation, various design considerations had to be evaluated, to achieve the goal of flexibility. Due to the availability of numerous scheduling algorithms, each suiting a different area or real-time application, a user must have the capability to use or implement their choice of scheduling paradigms. The task-dispatching mechanism must, therefore, be packaged and encapsulated to enable its convenient redefinition or replacement.

This thesis discusses the transformation, extension, and implementation of a concurrent and object-oriented programming language into a programming system suitable for general real-time development. Much of the discussion centres on Unix as the underlying operating system, and the actual implementation is in μC++ , a C++ dialect providing object-oriented concurrency.

The implementation for this work is done on a non-embedded μC++ environment running on top of Unix. As such, "real-time" can only be as "real" as Unix is willing to guarantee. Much of the discussion in this work, therefore, focuses on soft real-time. However, the design and implementation generalizes to cover hard real-time environments. With proper operating system support, much of the implementation can, with little change, take advantage of the underlying system's real-time guarantees. In essence, this implementation is as "real-time" as the underlying operating system.

Chapter 2

Time-Defined Delays

Time-defined delays make up the foundation for implementing real-time programs. Delaying a task means a task's execution is deferred to a later time. A time-defined delay, in turn, specifically implies that a task's execution is deferred until after a defined time frame. The task is said to "sleep", awaiting its wake-up time.

A task's ability to sleep for a specified time frame is crucial for implementing real-time programs. A time-constrained task always has a form of deadline associated with it. More sophisticated time-constrained tasks have numerous forms of deadlines (start time constraints, completion time constraints, etc.). All deadline management functions are usually implemented internally using time-defined delays.

Deadlines that require tasks to execute only after a specified time can trivially be implemented by putting the task to sleep, only to wake up when the defined time arrives. Consider a temperature monitoring system composed of a task A responsible for reading in values from various temperature sensors. It is a necessity that within 20 seconds, a temperature reading must be returned by every installed sensor. Assuming it takes no more than 2 seconds to read the values from all the sensors, this implies that task A should sleep for eighteen seconds, read values from the sensors, and go back to sleep for another eighteen seconds, only to repeat the cycle. Task A is said to be a *periodic task* due to its repetitive nature. Periodic tasks make up the majority of real-time tasks.

Deadline notification and detection are also implemented using time-defined delays. Following the temperature monitoring system described in the above paragraph, assuming that if two seconds after temperature reading initiates, an incomplete temperature reading is gathered from the facility, which results in an emergency situation. At the time task A begins reading in values from the different temperature sensors, a task B is spawned. Task B is composed of a time-defined delay statement, requesting a delay of 2 seconds, after which it checks to see if the table of temperature values has been completely updated by task A within the last two seconds. Should this condition fail, an exception is raised by task B in task A.

Chapter 3

Real-Time Scheduling

3.1 Notations and Definitions

Each task, real-time or non real-time, usually has a logical priority P associated with it. P signifies the logical importance of a task, in comparison to other tasks in the system. The lower the integer value of P, the higher its logical priority is. For instance, monitoring the temperature levels inside a power plant probably has a lower P value than clearing out the mail spool.

A real-time task is characterized by various properties; the most intrinsic is its computation time, denoted by C. Should a task be periodic (periodic tasks are tasks that re-execute after a specified time interval), as is the case with most real-time tasks, its periodicity is specified by T.

The time at which a task becomes ready for execution is denoted by r. The actual time it starts is s. Normally, a task completes its execution at time c, which is $s + C$. Being a real-time system, a real-time task also has a deadline d associated with it. The objective is to have $c \leq d$. Figure 3.1 illustrates the relationship among the various parameters.

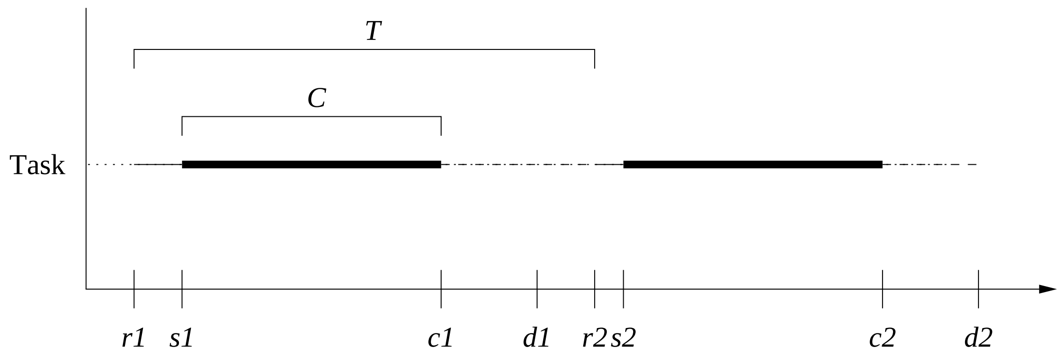

Figure 3.1: Time-Line of a Periodic Real-Time Task

From this point on, intrinsic time-attributes of tasks are referred to with capital letters. T, for instance, is a fixed attribute, which is known apriori. An intrinsic time-attribute is expressed as a time duration. In contrast, runtime dependent time-characteristics are referred to with lower-case letters. The deadline of a task during a certain run is specified by d, for example. However, a task can have a deadline D equal to T, for instance. In this case, D never changes, and is always the length of the period of the task, whereas d is the deadline of a task, based on the time it started execution. Lower-case attributes, therefore, specify absolute times.

3.2 Real-Time versus Non Real-Time Scheduling

In any time-sharing system, multiple processes or tasks exist simultaneously, necessitating a scheme to decide which ready process or task may execute. Schedulers implement the particular scheme for selection of ready processes or tasks.

Schedulers vary in objective, though the most common goal is to maintain fairness among the executions of the task set. Thus, most schedulers strive to evenly distribute execution times to all tasks, thus eliminating situations such as starvation.

A real-time system is distinct, in that its scheduler is unconcerned with the notion of "even processor allotment times" among its task set. The main objective of a real-time scheduler is to ensure tasks commence and terminate their executions to meet their specified deadlines. The fact that a task can be deprived of processor time at certain instances is usually of little consequence to a real-time system, so long as its deadlines are not compromised [51].

To illustrate the difference in real-time and non real-time scheduling paradigms, assume a system is composed of the set of tasks shown in Table 3.1. Furthermore, assume that the deadline D of each task (the time by which the computation needs to be finished by) is equal to the task's period T.

Assuming task τ_6 is the first task to arrive, a typical (non real-time) scheduler utilizing a first-come-first-served algorithm services it ahead of any other tasks in the set. Such scheduling behaviour produces disastrous results, since the execution of τ_6 effectively hogs the processor for ten seconds, thus preventing more "important" tasks from being executed.

Task	Period (T)	Computation Time (C)	Description	Priority (P)
τ_1	40 sec	1 sec	oxygen-level check	1
τ_2	10 μsec	1 μsec	temperature check	1
τ_3	30 μsec	8 μsec	pressure check	1
τ_4	15 μsec	2 μsec	oil-level check	2
τ_5	30 sec	20 μsec	water-level check	2
τ_6	60 sec	10 sec	download newsfeed	9

Table 3.1: Sample Task Set

A pre-emptive round-robin scheduling algorithm may also fail, depending on the duration of the time-slice. Tasks such as τ_2 are safe, in that they require small execution times; however, τ_3 will most likely not meet its deadline because of the size of the pre-emption cycle.

From the above examples, it is clear that normal scheduling algorithms fail, if utilized on real-time systems. Special real-time scheduling must be adopted, which takes into account the special deadline and time constraints inherent in real-time tasks.

3.3 Nature of Deadlines

As briefly described in the introduction, deadlines are usually classified according to their degree of importance. Another method of classifying deadlines is through a value function. A value function is a function of time, which indicates the value of a computation, and how such value affects the system as a whole.

Figure 3.2 illustrates a value function for a catastrophic hard real-time system. The cut-off point for the computation is the deadline. Should a computation exceed

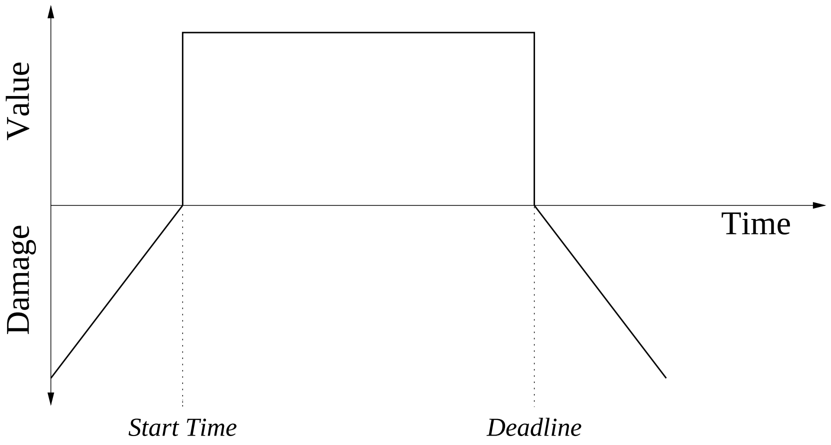

Figure 3.2: Computational Value Function for a Catastrophic Hard Real-Time Task

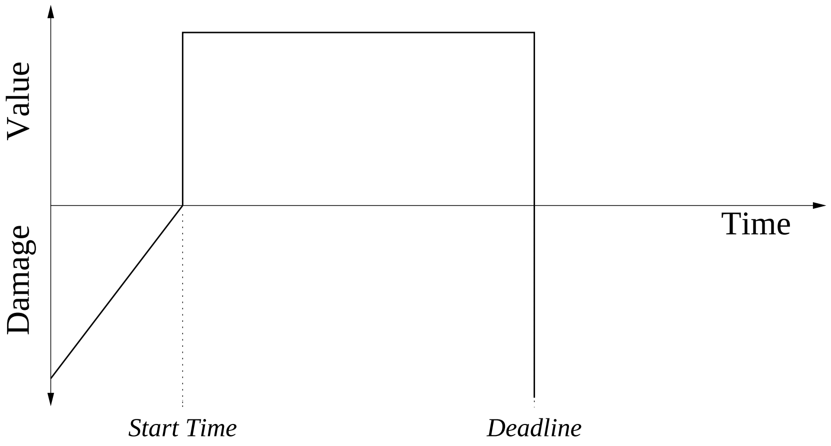

Figure 3.3: Computational Value Function for a Catastrophic Hard Real-Time Task

its deadline, some damage results. In this particular example, as time passes from the deadline, the damage caused increases. It is also worth noting that in this particular value function, should a computation initiate earlier than its scheduled start-time, damage also results. The earlier the computation is initiated prior to its scheduled start time, the more damage results. Figure 3.3 illustrates a value function in which after the deadline, the value of the function drops to ∞. In such cases, should a deadline be missed, catastrophe is instantaneous.

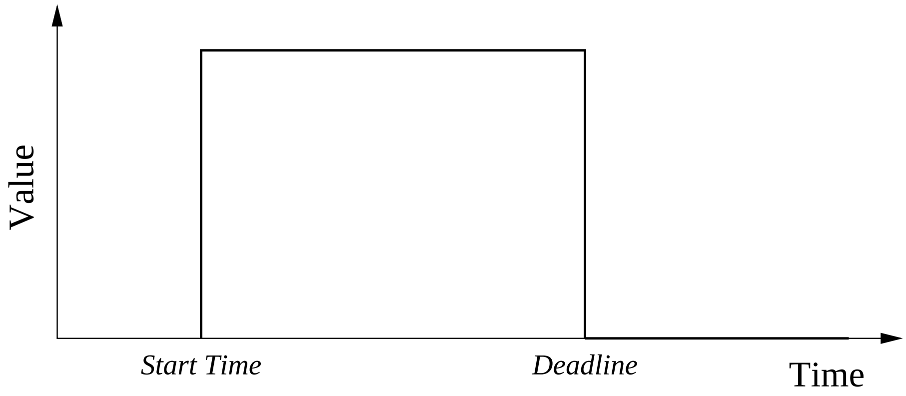

Figure 3.4: Computational Value Function for a Hard Real-Time Task

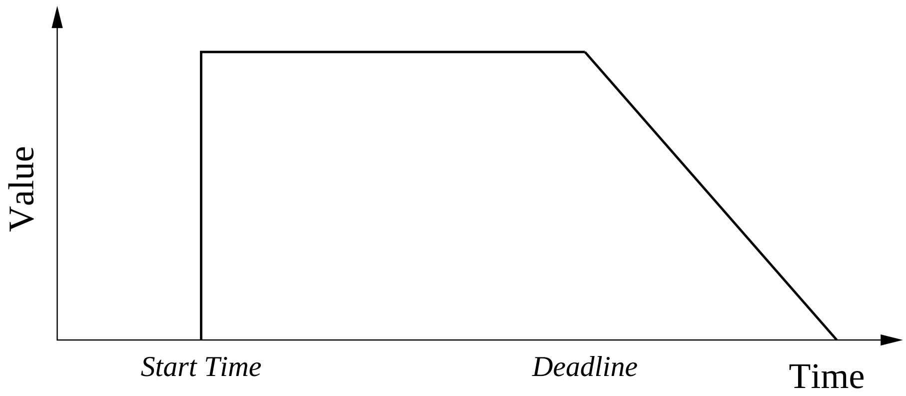

Figure 3.5: Computational Value Function for a Soft Real-Time Task

Certain tasks and systems are hard real-time, but do not necessarily result in catastrophic results should deadlines be missed. Figure 3.4 illustrates the value function for such systems. Should the deadline of a computation be missed, the computation then has no value. In contrast, a soft real-time system (figure 3.5) is a system in which tasks or computations should ideally be completed by its deadline. However, should the deadline be missed, the computation is not necessarily deemed useless. How "late" a task or computation is, indicates how much it is worth. The farther away from a deadline a computation is, the less valuable the result becomes. It is, at times, imperative to know the point in time when a computation becomes valueless.

Figure 3.6: Computational Value Function for a Non Real-Time Task

A non real-time system, in contrast, is usually characterized by the value function illustrated in figure 3.6. Observe that there is no deadline indicated in the time axis. Though a task or a computation may complete at a much later time than desired, the computation is still of value.

Value functions are used in some real-time scheduling algorithms. However, computationally, scheduling that utilizes even the simplest forms of value functions becomes unmanageable [39]. Tokuda et al [55] implemented some canonical value functions, and compared their run-time costs in comparison to various "classical" real-time scheduling algorithms. The results indicated a significant amount of scheduling overhead, and that in numerous instances, the cost of making a single scheduling decision is more costly than executing several tasks.

3.4 History

Although there has been a lot of early research in the field of real-time scheduling, most of it was not specifically geared towards real-time scheduling of computer tasks. In fact, real-time computer scheduling has its roots in job-shop scheduling, where scheduling strategies for factory, job, and machine operations were studied [15]. Job-shop scheduling provided invaluable ideas and insights, which slowly paved the way to the development of the field of hard real-time computing systems.

The main objective of a job-shop scheduler is to schedule its task set so that execution is accomplished in a minimum amount of time. Generally, individual task constraints are not taken into consideration, and the task set, as a whole, is the only concern. This approach obviously differs from the hard real-time scheduling paradigm.

Despite the apparent difference between job-shop scheduling and real-time scheduling, job-shop scheduling acted as a foundation for very basic real-time scheduling techniques. The apparent difference on whether the scheduling paradigm focused on a task-set or an individual task is often of little consequence. The bottom line is that, in a lot of cases, both paradigms lead to the timely completion of each individual task. *Earliest deadline* scheduling (Section 3.7) is a notable real-time scheduling algorithm derived from job-shop scheduling is. In a job-shop environment, where job completion times are promised to customers, the tardiness of a job set is minimized by sequencing the jobs in order of increasing due-times, i.e., perform the job with the shortest time to delivery and then the next shortest time, and so on. As it turns out, executing tasks in the order of increasing due-dates results in a higher ratio of tasks completing specified deadlines over prior traditional approaches.

3.5 Real-Time Scheduler Classification

Various taxonomies have been employed for understanding the numerous real-time scheduling algorithms available. The most common classification can be drawn between static versus dynamic scheduling. A static scheduling algorithm usually requires prior knowledge of a task's behaviour, since such knowledge is usually employed in creating a predictable system schedule or behaviour apriori (prior to runtime). A dynamic scheduling algorithm usually computes a system's behaviour on-the-fly during runtime. As a result, prior knowledge of a task's predictable behaviour is not as essential. In a dynamic scheduling environment, the future behaviour of the system is usually determined at various time-stages, based on all the tasks' characteristics and deadlines at that particular time-stage.

Static scheduling algorithms are usually very predictable and efficient. The fact that the calculations to determine the scheduling behaviour of a system is performed off-line makes the run-time cost of static scheduling very low. The inherent predictability of a static scheduling algorithm also makes it highly suitable for catastrophic and hard real-time systems. An example of a static scheduling algorithm is the rate-monotonic algorithm (see section 3.9).

Dynamic scheduling algorithms, in comparison, have a higher run-time overhead. This higher run-time overhead often results in a more flexible system, however. The addition or deletion of tasks at run-time can easily be dealt with. In addition, dynamic scheduling algorithms are more suitable for unpredictable systems, or systems with unpredictable real-time tasks. Dynamic scheduling nicely deals with the possibly changing and fluctuating resource/time constraints of such unpredictable tasks. The unpredictable nature of dynamic scheduling algorithms, on the other hand, usually makes them unusable for hard real-time system. The earliest deadline first scheduling algorithm (see section 3.7), for example, is a type of dynamic scheduling algorithm because the decision of which task executes at any point in time is determined by the deadline d of each task, which is a dynamic property.

All the scheduling algorithms presented in this thesis are preemptive in nature. Non-preemptive scheduling is much more complex than preemptive scheduling. Optimal non-preemptive scheduling of tasks with timing constraints has been shown to be NP-hard [13].

In the following sections, various real-time scheduling paradigms are discussed and analyzed. Particular attention and emphasis is given to the rate-monotonic scheduling algorithm (Section 3.9) and its derivatives, due to its high suitability for hard real-time systems, as well as its overwhelming popularity as the scheduling standard in the field of real-time computing.

3.6 Cyclic Executive

The simplest form of real-time scheduling is through a cyclic executive. In this form of scheduling, the order of task execution is crafted during the system design phase. Determining execution order involves painstaking trial-and-error or the brute-force method of re-arranging task execution orders in combination with various timing delays and specifications, to create a schedule in which all deadlines are met. After which, throughout the system's lifetime, the program merely repeats or cycles through the execution of this carefully crafted schedule.

A major advantage of a cyclic executive is its speed and predictability. The major drawback of a cyclic executive is its inflexibility. Should tasks be added to the system or should computation times change, the whole schedule may need to be regenerated, which makes maintenance extremely difficult.

3.7 Earliest Deadline Algorithm

Earliest deadline first algorithm [37], also known as due-date scheduling, orders a task set's execution by examining the deadlines of each task in the set. Tasks are ordered by deadlines, in which tasks with earlier deadlines are executed first.

As mentioned, earliest deadline first is a dynamic scheduling algorithm (a static counterpart of this algorithm is the *deadline-monotonic scheduling algorithm*, discussed in section 3.9.2.1), which implies that at certain time intervals, the tasks in the task set are examined, and the task with the earliest deadline at that particular time-frame is executed. To implement this scheme necessities the deadline information d for all tasks must be available at each time when context-switching occurs (or when "earliest-deadline" calculations are to take place). Furthermore, no precedence constraints (restrictions involving the order in which tasks must be executed) are taken into consideration, and task independence (there is no form of communication among tasks) is required.

Given that the earliest deadline first algorithm is a dynamic scheduling algorithm, it is possible to have tasks with varying timing/resource requirements.[1] During transient overloads, deadlines can, in turn, be missed in an unpredictable fashion.

The earliest deadline algorithm has been shown to be optimal, in that should a scheduling algorithm exist that can schedule the specified task set, the earliest deadline algorithm can likewise ensure that the deadline of each task in the task set is met. On the other hand, should a task set be unschedulable with the earliest deadline algorithm, no other scheduling algorithm can schedule the given task set [37].

3.8 Earliest Critical Time Algorithm

The earliest critical time first algorithm, also known as the least slack time first algorithm, is quite similar to the earliest deadline algorithm. Critical time, or slack time, is defined as the deadline of a task minus the remaining computation time required by the task. In short, it is the amount of time a task can afford to be delayed, without missing its deadline (figure 3.7 illustrates the notion of slack time). The task set is, in turn, ordered by increasing slack-time. The task with the least slack-time is run ahead of tasks with larger slack-times.

Like the earliest deadline algorithm, the deadline information of each task, d,

[1]Guaranteeing hard deadlines on tasks with varying timing/resource constraints is almost impossible. Whenever tasks of varying time and resource constraints are discussed, it is often in the context of a soft real-time environment. Dynamic scheduling algorithms are most appropriate for such task sets. Static scheduling algorithms can, however, still be employed. Section 3.9.2.2 discusses a means by which static scheduling algorithms can be adapted to deal with such tasks.

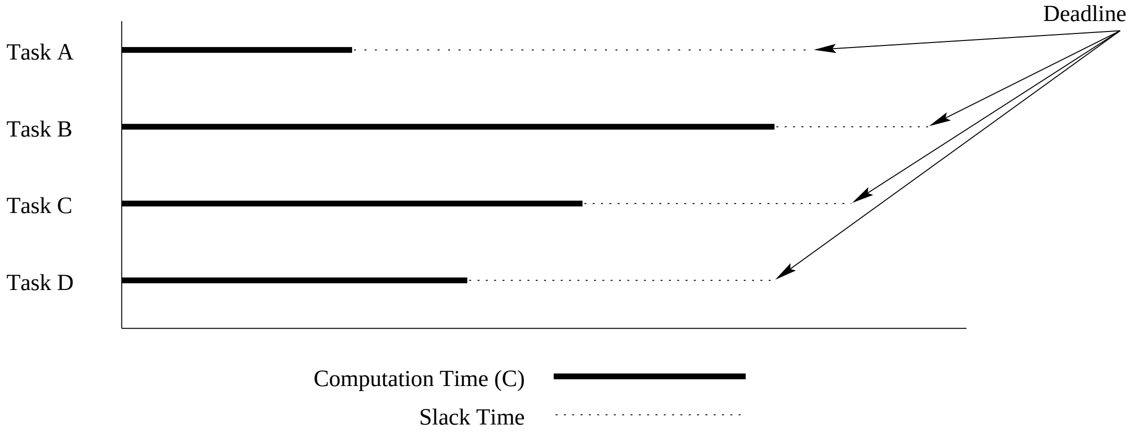

Figure 3.7: Illustration of Slack Time

must be known. Besides d the completion time requirement C of each task is also required, which allows for the calculation of the critical time.

The least slack time first algorithm has been shown to be optimal, in the same sense as the earliest deadline first algorithm. Should any scheduling algorithm be capable of scheduling a task set, so can the least slack time first algorithm.

3.9 Rate Monotonic Algorithm

Arguably, the most significant landmark in real-time scheduling is the paper by Liu and Layland in 1973. The algorithm described by Liu and Layland is geared towards finding a feasible scheduling strategy for a set of periodic tasks. Prior to this, most real-time computer scheduling was through cyclic executives (Section 3.6), which is best characterized as "brute-force," but often referred to as the "trial-and-error" method.

Liu and Layland introduced the *rate-monotonic priority assignment* or *rate-monotonic* algorithm [37, 36, 47], which soon gained popularity, and became the basis for further research in the field of real-time. It includes the following assumptions and limitations:

1. All tasks in the system are periodic.
2. All tasks in the system have a deadline that is equal to their period ($D = T$).
3. All tasks have a constant computation time (computation times do not change).
4. All tasks are independent. Thus, there is no form of communication among tasks.
5. A task can be preempted at any time.
6. There are no critical sections in any of the computations.
7. The cost of context-switching and other system overheads are ignored.

The rate-monotonic algorithm merely orders a task set in increasing order of periodic frequency. At any point in time, the ready task with the shortest period is the most eligible task to run. This form of scheduling is normally implemented with the use of priorities.

The rate-monotonic algorithm can be implemented by assigning a priority p to a task, based on a task's periodicity. Tasks with shorter periods (and thus, tasks that require more frequent executions) are assigned higher priorities than tasks of longer periods. Note that priority p is different from priority P, described in section 3.1. P

is the logical importance of a task from the perspective of a human or environment, whereas p is the priority of a task from a computer scheduler's perspective. As with P, the lower the integer value of p, the higher its priority. Taking the task set of figure 3.1 as an example, based on the rate-monotonic priority assignment scheme, the tasks would be ordered in the following sequence, from highest priority p to lowest: τ_2, τ_4, τ_3, τ_5, τ_6, and τ_1. Note that although task τ_1 is a very important task, it is given the least priority p by the rate-monotonic scheme.

In essence, in its most common implementation, the rate-monotonic algorithm can be thought of as a formalized method of assigning priorities to tasks. The complete scheduling algorithm, in turn, is comprised of a prioritized preemptive scheduler, which executes the tasks on the system based on the assigned priorities from the rate-monotonic priority assignment scheme. As a result of this, the rate-monotonic algorithm is often referred to as *fixed priority preemptive scheduling* [5]. This is a very misleading name, however, given that there are other scheduling algorithms, both real-time and non real-time, that employ fixed priority preemptive scheduling, yet, are not remotely similar to the rate-monotonic algorithm.

3.9.1 Schedulability Test

A major difference between rate-monotonic and a cyclic executive is the availability of a schedulability test, which determines whether each task in the set of tasks meets its deadlines. The availability of a schedulability test is a highly desirable feature, especially in catastrophic hard real-time systems, in which actual testing is not always possible. Rate-monotonic schedulability tests require both the periodicity T, as well as the computation time C for each task in the system.

The heaviest load on a system is at a point called the *critical instant*, which is usually at the beginning of a system's run, where all the tasks are just released and awaiting processor time. After each task has been executed once, it then becomes bound to a certain time t, in which subsequent executions of the task occur at a multiple of t. Should a task meet its deadline during the critical instant (put another way, should a task meet its first deadline), it logically follows that a task can meet its deadlines at other times during a system's run, when loads are not as heavy as during the critical instant. Based on the notion of a critical instant, Liu and Layland proved that, given a rate-monotonic priority assignment for a set of tasks, each task is guaranteed to be schedulable (each task meets its deadlines) if the following equation holds:

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

where n is the number of tasks in the system, C_i is the computation time, and T_i is the periodic time interval of a task τ_i.

If the combined processor utilizations of all the tasks in the system is no greater than $n(2^{\frac{1}{n}} - 1)$, each task is schedulable. So, if a set of three tasks are scheduled in the system, and its total processor utilization is less than or equal to 78%, the task set is guaranteed to be schedulable. Note that

$$\lim_{n \to \infty} n(2^{\frac{1}{n}} - 1) = \ln 2$$

the processor utilization bound converges to ln 2 or approximately 69.31%.

The above schedulability test described by Liu and Layland is pessimistic, in that should the condition hold, the task set is guaranteed schedulable. It is possible to exceed the processor utilization bound, and still have the task set schedulable. Lehoczky, Sha, and Ding subsequently proved an exact schedulability test for the rate-monotonic algorithm [32, 30]. Given a set of tasks ordered by decreasing priority (τ_i ... τ_j, where i is a lower priority than j), τ_i meets its deadlines if and only if

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

Should all the tasks in the task set meet the above condition, the task set is schedulable. Lehoczky et al further demonstrated that, for the average case, large task sets, which approached processor utilizations close to 90%, still met all deadlines. Of course, there are situations when task sets slightly exceeded the pessimistic utilization bound of the Liu and Layland schedulability test and are indeed unschedulable.

The rate-monotonic algorithm is proven by Liu and Layland to be an optimal static priority scheduling algorithm, in the sense that if any other static scheduling algorithm can produce a feasible schedule for a set of real-time tasks, it can likewise be scheduled by the rate-monotonic algorithm.

3.9.2 Relaxing Restrictions

Unfortunately, the rate-monotonic algorithm is highly restrictive. The requirement that task deadlines be equal to their periods ($D = T$) is highly unrealistic, for instance. Significant research has been conducted to relax the rate-monotonic's assumptions and limitations. These are discussed in the following sections.

3.9.2.1 Removing the D=T Restriction

The restriction that the deadline D of every task be equal to its period T is a very unrealistic requirement. Most real-time systems contain tasks with deadlines that are earlier than their periodicity, and though rarer, a real-time system may even have tasks whose deadlines are longer than its period.

When D $\leq$ T Leung and Whitehead [34] have defined the *deadline-monotonic scheduling algorithm*, also known as the *inverse-deadline scheduling algorithm*, which addresses situations in which a task's deadlines are earlier than or equal to its period. Deadline-monotonic is an extension of the rate-monotonic, with the slight change that task priorities p are ordered according to task deadlines, instead of task periodicities. Tasks with shorter deadlines are assigned higher priorities, in comparison to other tasks with longer deadlines. When the deadlines D and the periods T of every task are equal, the deadline-monotonic priority assignment scheme is the same as the rate-monotonic.

The deadline-monotonic priority assignment has been shown by Leung to be optimal, in the same way as the rate-monotonic priority assignment is. Should a set of tasks (with any task possibly having deadlines less than its corresponding period) be schedulable by another static priority assignment scheme, the deadline-monotonic algorithm is also capable of scheduling the task set.

Audsley developed a schedulability test for deadline-monotonic scheduling [4, 6, 3], given by:

$$\forall i : 1 \leq i \leq n : \frac{C_i}{D_i} + \frac{I_i}{D_i} \leq 1$$

where n is the number of tasks in the system, and

$$I_i = \sum_{j=1}^{i-1} \left[\left[\left\lfloor \frac{D_i - D_j}{T_j} \right\rfloor + 1 \right] C_j + \left[\left\lceil \frac{D_i}{T_j} - \left(\left\lfloor \frac{D_i - D_j}{T_j} \right\rfloor + 1 \right) \right\rceil \right] \cdot min \left(C_j, D_i - \left\lfloor \frac{D_i}{T_j} \right\rfloor T_j \right) \right].$$

The above schedulability test is sufficient, but not necessary (the system may still be schedulable, even if the above condition fails). As a result, this schedulability test can be characterized as pessimistic. An exact test is extremely costly to compute.

When D $\geq$ T Though generally not as common, there are situations in which it is desirable to have a periodic task's deadline exceed its periodicity. For example, a certain military installation may need to launch a missile every thirty minutes. The launching procedure for a missile must take, at most, thirty-five minutes to complete since onset. This implies that the deadline for the periodic task of launching missiles (which is thirty-five minutes) exceeds the task's periodicity (which is thirty minutes). As the first missile is at its last leg of its launching process (thirty minutes after the launching procedure was initiated), a subsequent missile launching must initiate (and thus, a new task must be spawned to handle the subsequent missile launching). It is important to note that since the deadlines or completion times of the periodic chores[2] exceeds the chores' period, a new task must be created for various instances or periods of the same chore. Thus, two (or possibly more) tasks are required to meet the period T. For such situations where a periodic task's deadline exceeds its periodicity, Shih, Liu, and Liu introduced the *modified rate-monotonic algorithm*[49, 36].

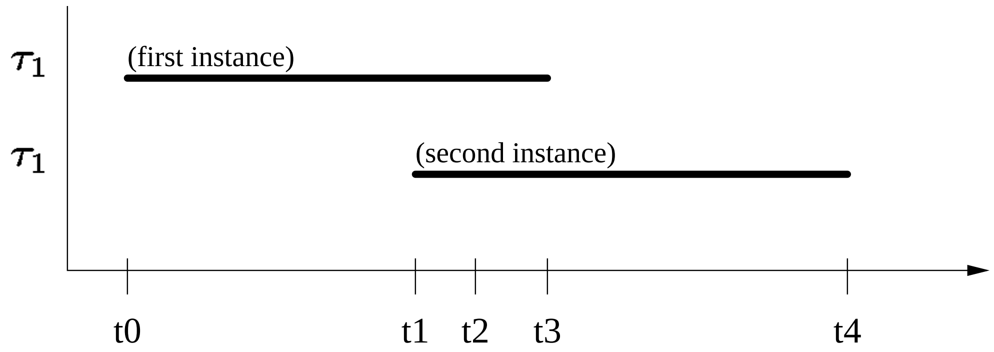

Figure 3.8: Time-Line of a Task when D > T

In the modified rate-monotonic algorithm, tasks in the system are divided into two categories: tasks which are "new" to their current period phase, and tasks which are "old" (i.e., tasks that have started earlier than the current period phase). In short, using the tasks illustrated in figure 3.8 as an example, given task τ_1 with period T and deadline D, where T=t1 < D, the first instance of τ_1 initiated at time t0 completes at t3. However, before it completes at time t3, a second instance of τ_1 is spawned at time t1. At time t2, the first instance of task τ_1 is referred to as "old", and the second instance of task τ_1 is referred to as "new."

[2]Perhaps it is clearer to use "periodic chores" instead of "periodic tasks" to refer to "tasks" possessing this property. To be consistent with the terminology adopted the real-time community, however, the latter term is adopted in this thesis.

With the modified rate-monotonic algorithm, within a task's lifetime, each task has two types of priorities assigned to them. Both types of priorities are obtained using the rate-monotonic algorithm. However, both types of priorities differ, in that the first set of priorities within a task set have priorities higher than any priorities of the second priority-type set. In short, each task τ in a task set has priorities p and $p+k$, where p is some priority determined through the rate-monotonic algorithm, and k is some positive constant for the entire task set, such that $k > \max_{0 \leq i \leq n} p_{\tau_j}$, where n is the number of tasks in the system.

The first set of priorities (the higher set of priorities) are assigned as the running priority of "old" tasks, whereas the second set of priorities (the lower set of priorities) are assigned as the running priority of "new" tasks.

Following the example task set illustrated in figure 3.8, at time t0, the first instance of τ_1 runs at priority p_1. At time t1, the second instance of τ_1 begins, thus making the first instance an "old" task. From this point on, the first instance of τ_1 now runs at priority $p_1 + k$. The second instance starts and continues to run at priority p_1, until the third instance of τ_1 arrives.

Note that, unlike the rate-monotonic algorithm, the tasks running on a modified rate-monotonic algorithm encounter a priority change (should its deadline be greater than its period). As a result of this, the modified rate-monotonic algorithm is a semi-static priority-driven algorithm. Also note that should all tasks in the system have deadlines less than their period, only the second set of priorities is utilized, and so, the modified rate-monotonic algorithm collapses to the rate-monotonic algorithm in this case.

Shih et al have shown that should the tasks' deadlines be deferred by sufficiently long times, the modified rate-monotonic algorithm should find a feasible schedule for the task set, so long as the processor utilization is less than or equal to 1. It is proven that the modified rate-monotonic algorithm is optimal for a task set, should the following conditions exist:

1. There exists $\nabla \geq 2$, such that for all i, where $0 \leq$ i $\leq$ n, $D_i \geq (\nabla - 1)T_i$.

2. The ratio between $p_{longest}$ and $p_{shortest}$ in a task set should be less than or equal to ∇.

Furthermore, it is also true that for a task set, if $D_i \geq 2T_i$ (for $0 \leq$ i $\leq$ n), then the task set is schedulable with the modified rate-monotonic algorithm if:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq \begin{cases} 1 & 2 \leq n \leq 3 \\ \frac{1}{2}\left(1 + n(2^{\frac{1}{n}} - 1)\right) & n \geq 4 \end{cases}$$

3.9.2.2 Non-periodic Tasks

Rate-monotonic requires that all tasks be periodic. However, there are many situations where some or all of the tasks in a system are non-periodic, i.e., tasks with arrival times that are non-deterministic. The problem with non-periodic tasks is that arrival times are unpredictable. Should a large set of non-periodic tasks arrive at one particular instant, the system may experience a condition known as *transient overload.* Under such situations, dynamic scheduling algorithms can manifest unpredictable behaviours (such as more important deadlines being missed as opposed to less important ones, where P defines the level of importance). Alternatively, static scheduling algorithms can be adapted to deal with non-periodic (or somewhat unpredictable) tasks. Dynamic scheduling algorithms have potential for real-time systems that are composed of non-periodic tasks. The earliest-deadline-first scheduling algorithm, for instance, has been shown to be optimal when a mixture of periodic and aperiodic tasks exist in a system.

Alternatively, static scheduling algorithms, such as the rate-monotonic algorithm and its variants, are geared towards scheduling periodic tasks. Tasks of a non-periodic nature are not addressed. Various means of dealing with such tasks (within the framework of scheduling algorithms designed for periodic tasks) [50, 17, 47] are, in turn, described in this section.

Before proceeding further, however, a distinction must be drawn between *aperiodic tasks*, which means that no indication is given as to when tasks may arrive in a system, versus *sporadic tasks*, which imply that a maximum of one instance of such a task may arrive within a specified time frame (there is a minimum inter-arrival time between sporadic tasks). Little can be guaranteed about aperiodic tasks, for there is no bound on the number of aperiodic tasks that may appear in a system at any particular time. Deadlines of sporadic tasks, on the other hand, can be guaranteed.

Polling Server The simplest means in which sporadic or aperiodic tasks can be dealt with by a periodic scheduling algorithm is by setting up a periodic task that "polls" for the existence of sporadic or aperiodic tasks during its execution, and uses its allotted periodic computation time to execute any existing sporadic or periodic tasks.

This "polling" approach, however, is non-ideal and wasteful. Should no sporadic or aperiodic task arrive during the execution time of the designated periodic polling task, valuable processing time is wasted by the designated periodic polling task spinning, waiting for a sporadic or aperiodic task arrival. In addition, should a sporadic task arrive right after the designated polling task executes, the sporadic task must wait for the next period of the polling task in order to be given processor time. Note that these deficiencies do not pose a scheduling problem because the polling server is scheduled as if it was a normal periodic task. The problems inherent in the "polling" approach merely imply that fewer aperiodic or sporadic tasks may be run.

Priority Exchange Server The priority exchange server[31] method acts similarly to the polling server mechanism. A periodic server task is allocated in the system, which is responsible for executing sporadic or aperiodic processes. Unlike the polling server approach, should there be no sporadic or aperiodic processes to service at the start of the designated server task, the server task exchanges priority with a lower priority periodic processes. This technique allows a higher priority task to execute during the server's allotted computing time (should there be no sporadic or aperiodic tasks to service), thus, resulting in no wasted processor time. At the same time, computing time is preserved, should a sporadic or aperiodic task arrive, but the sporadic or aperiodic task runs at a lower priority level than it was initially allotted. This discrepancy in priorities makes the prediction of missed deadlines extremely difficult. Deadlines of sporadic tasks are most likely missed in a highly unpredictable fashion.

Deferrable Server The deferrable server method[31, 50], like the previous methods for dealing with sporadic or aperiodic tasks, allots a periodic server task that services sporadic or aperiodic events. Unlike the priority exchange server, should the deferrable server have no sporadic or aperiodic task to execute during its allotted computing time, it does not swap priorities with a lower priority periodic task. Instead, execution of the deferrable server is postponed (the deferrable server sleeps or blocks, pending the arrival of a sporadic or aperiodic task). When a sporadic or aperiodic task does become available, the deferrable server is made ready, and executes at its originally assigned priority. Thus, deadlines are missed in a predictable fashion.

Compared to a priority exchange server, the implementation of a deferrable server is much simpler. The fact that priorities do not change makes its implementation conceptually simpler and usually less expensive. As well, it allows more non-periodic tasks to be serviced. The disadvantage of the priority exchange server is that aperiodic or sporadic tasks that arrive towards the end of the deferrable server's execution period, unlike the priority exchange server, must wait to resume execution at the next period of the deferrable server.

Sporadic Server The above methods for dealing with sporadic or aperiodic tasks have the inherent property that server execution times are replenished at the beginning of the server's period. Server execution time can be thought of as a collection of tickets, such that should a sporadic or aperiodic task arrive, the server gives the task some tickets sufficient for its execution. Should the server's tickets run out, the ready sporadic or aperiodic task(s) have to wait until the server's next period, which is when the server gets to replenish its supply of tickets.

A sporadic server[50] is different in that the replenishment of its execution time is not always at the beginning of a period. Viewed another way, its "tickets" are not replenished at fixed times, but rather, at varying points through its execution. Otherwise, its behaviour is identical to a deferrable server.

The sporadic server's approach, in essence, combines the advantages of both the priority exchange server and the deferrable server. By replenishing its execution times when sporadic or aperiodic tasks arrive, instead of at fixed times, the replenishment policy can best be described as sporadic – the same nature as the tasks it services. This approach also forces the execution of sporadic or aperiodic tasks to be more evenly distributed.

Sprunt, Sha, and Lehoczky further demonstrated that should a periodic task τ_i be schedulable in a periodic-algorithm-scheduled system, the *sporadic* task serviced by a sporadic server (having the same period and execution time as τ_i) is also schedulable[50]. As a consequence of this, hard sporadic processes can, in turn, be guaranteed in a periodic-algorithm-scheduled environment by having each sporadic server service one sporadic task in the system. The number of sporadic servers in the system would, in turn, correspond to the number of hard sporadic tasks. Softer sporadic tasks, on the other hand, can be serviced by a shared sporadic server.

3.9.2.3 Task Synchronization

Rate-monotonic (and all the other algorithms described so far) have the restriction that all tasks must be independent of each other. As a result, there is no manipulation of task-shared data structures, nor are there any forms of inter-task communication. In most real-time applications, this restriction is unacceptable.

Normally, certain tasks in a real-time system have some form of dependence, and certain tasks do manipulate shared data structures.

In the following sections, two popular algorithms that allow inter-task communication are presented. The *Basic Priority-Inheritance Protocol* and the *Priority-Ceiling Protocol* are general algorithms that can be applied on top of various scheduling algorithms (static as well as dynamic). A pre-emptive scheduling environment is required, however.

Basic Priority-Inheritance Protocol Assume a system with three tasks, τ_0, τ_1, τ_2, and corresponding priorities *p0, p1, p2*, which are in decreasing order. At time t_0, τ_2 is in a critical section holding a semaphore[3] S. At time t_1, τ_0 arrives and preempts τ_2 due to its higher priority. At time t_2, τ_0 attempts entry to the critical section occupied by τ_2. However, since τ_2 holds semaphore S, τ_0 is inhibited from proceeding; τ_0 blocks, and τ_2 resumes execution. At time t_3, τ_1 starts execution and preempts t_2. Task τ_0, the task of highest priority, does not execute while lower priority tasks do. This condition, known as *priority inversion*, can occur indefinitely.

Lampson and Redell [29] discussed the problem of priority inversion with regards to monitors. They proposed that a monitor *always* be run at a priority level that is higher than any of the priorities of the tasks calling the monitor. This approach ensures that any task running inside the monitor is allowed to execute uninterrupted

[3]For the purpose of this discussion, a semaphore is used. Note, however, that the topics discussed in this section also apply to other synchronization primitives such as locks, monitors, etc., which are used to establish mutual exclusion. Furthermore, for the sake of simplicity and generality, the terms "holds" or "locks" and "releases" are used interchangeably with the semaphore primitives P and V, respectively.

by any other task that calls the monitor, thus preventing priority inversion. This solution is very pessimistic, however, in that any task running inside a monitor always runs at an extremely high priority. A normal high priority task may, in turn, have to compete with the task running inside the monitor, even though the high priority task may not need access to the monitor at all.

Rajkumar proposed the *basic priority-inheritance protocol*[41, 42, 46], which puts a bound on the occurrence of priority inversion, in a less pessimistic manner than Lampson and Redell's solution. Rajkumar's solution is to execute a critical section at the priority of the highest blocked task waiting to enter the critical section. So, given the same sample task set, at time t_2, when τ_0 attempts entry and blocks, the priority of τ_2 (which is executing in the critical section) is bumped up to the priority of τ_0 (assuming it has the highest priority among the set of blocked tasks). At time t_3, when τ_1 arrives, τ_2 continues running and is not preempted by τ_1. τ_2 runs uninterrupted until it finishes the critical section, after which, it returns to its normal priority of *p2*. Task τ_0, the highest priority task, is then executed.

The basic priority-inheritance protocol bounds the time priority inversion occurs: should there be n lower priority tasks in the system, and the n lower priority tasks access m distinct semaphores, a task can be blocked by at most $min(n, m)$ critical sections. Despite this bound, the blocking duration for a task can still be significant, however. Suppose at time t_0, task τ_2 arrives and locks semaphore S_0. At time t_1, task τ_1 arrives, preempts τ_2, and locks semaphore S_1. At time t_2, task τ_0 arrives, needing to sequentially access both semaphore S_0 and S_1. Since both semaphores are locked by two lower priority tasks, τ_0 must wait for the duration of two critical sections (till S_1 is released by τ_1, then, S_0 is released by τ_2). This problem is known as *chain blocking*.

The most serious deficiency of the basic priority-inheritance protocol, however, is its inability to deal with the possibility of deadlocks. The prevention or escape from a deadlock situation is not addressed by the protocol. As such, the basic priority-inheritance protocol adds a significant responsibility to the programmer of ensuring that deadlock situations do not arise.

Priority-Ceiling Protocol The *priority-ceiling protocol*[41, 42, 19, 46], also proposed by Rajkumar, solves the problem of priority-inversion, without causing the problem of chain blocking and deadlocks. In addition, the priority-ceiling protocol bounds the blocking time of a task to, at most, the duration of one critical section of any lower priority task.

In the priority-ceiling protocol, each semaphore has a *priority-ceiling* associated with it, which is defined as the priority of the highest priority task that may lock this semaphore. When a task τ tries to lock a semaphore S, τ checks to see if the priority-ceiling of any of the currently locked semaphores (excluding semaphores locked by task τ) is higher-than the priority of τ. Should the condition fail (the priority ceiling of all the currently locked semaphores, except the semaphores locked by τ, is evaluated to be lower than the priority of τ), τ may lock S. Should the condition exist, however, τ is blocked and is not granted the semaphore S. In turn, the task holding the semaphore that caused this blocking (the one possessing a priority-ceiling that is higher-than or equal to the priority of τ) is given or inherits the priority of τ, if the priority associated with this semaphore is lower than that of

τ. After completing the execution of the critical section, the task that possesses this critical section returns to its previous priority-level (in a sense, inherited priorities are stacked).

Assume a task set composed of tasks $\tau_0, \tau_1, \tau_2, \tau_3$, with corresponding priorities *p0*, *p1*, *p2*, *p3* (in decreasing order). Suppose each task accesses a critical section in the following manner:

τ_0: P(S_0) ... P(S_2) ... V(S_2) ... V(S_0)

τ_1: P(S_2) ... P(S_3) ... V(S_3) ... V(S_2)

τ_2: P(S_1) ... P(S_4) ... P(S_5) ... V(S_5) ... V(S_4) ... V(S_1)

τ_3: P(S_5) ... P(S_4) ... P(S_0) ... V(S_0) ... V(S_4) ... V(S_5)

A static examination shows that the priority ceiling of the semaphore S_0 is *p0*, since this is the priority of the highest-priority task that ever accesses it. Respectively, S_1 has priority ceiling of *p2*, S_2 has *p1*, S_3 has *p1*, S_4 has *p2*, and S_5 has *p2*. Suppose the following events take place in the system (also illustrated in figure 3.9):

1. At time t_0, τ_3 arrives.

2. At time t_1, τ_3 locks S_5.

3. At time t_2, τ_2 arrives and preempts τ_3.

4. At time t_3, τ_2 attempts to lock S_1. Since a semaphore is currently locked in the system (semaphore S_5), the highest priority-ceiling among the locked semaphores is determined, and turns out to be *p2*. Since this priority value is equal to the priority of the current task executing (τ_2), τ_2 is blocked, and the task holding the lock to the semaphore with the highest priority-ceiling

among the locked semaphore set (task τ_3 in this case) inherits the priority of τ_2. The scheduler then picks up the next most eligible task to execute (the task with the highest priority).

5. At time t_4, τ_3 attempts to lock S_4. Since the set of locked semaphores, excluding those held by τ_3, is empty, τ_3 locks S_4.

6. At time t_5, τ_1 arrives and preempts τ_3.

7. At time t_6, τ_1 attempts to lock S_2. Among the set of locked semaphores (excluding those held by τ_1), the highest priority ceiling is *p2*. As a result of this, τ_1 is given the lock to S_2.

8. At time t_7, τ_1 attempts to lock S_3. Among the set of locked semaphores not held by τ_1, the highest priority ceiling is still *p2*. Therefore, τ_1 locks S_3.

9. At time t_8, τ_1 releases S_3.

10. At time t_9, τ_1 releases S_2.

11. At time t_{10}, τ_1 completes execution. The highest priority task that is eligible to run is selected by the scheduler, which is τ_3.

12. At time t_{11}, τ_3 attempts to lock S_0. Since all locked semaphores are owned by τ_3, τ_3 locks S_0.

13. At time t_{12}, τ_0 arrives and preempts τ_3.

14. At time t_{13}, τ_0 tries to lock S_0. It fails to do so, since the highest priority ceiling in the locked semaphore set is equal to the priority of τ_0. The priority

of the task holding the locked semaphore in question (task τ_3) is bumped up to priority *p0*. τ_0 blocks, and the highest priority task eligible to run is selected by the scheduler (task τ_3).

15. At time t_{14}, τ_3 releases S_0. Upon release of S_0, τ_3 returns to its previously inherited priority of *p2*, and awakens task τ_0 by "V"ing on S_0. As a result of this, the scheduler preempts τ_3 with the highest priority task eligible to run (τ_0).

16. At time t_{15}, τ_0 attempts to lock S_2. It is granted the lock, based on the rules outlined for the priority-ceiling algorithm.

17. At time t_{16}, τ_0 releases S_2.

18. At time t_{17}, τ_0 releases S_0.

19. At time t_{18}, τ_0 completes execution. The scheduler selects task τ_3, the only eligible task to run.

20. At time t_{19}, τ_3 releases S_4.

21. At time t_{20}, τ_3 releases S_5. S_5 is the semaphore responsible for τ_3 inheriting a higher priority. Since S_5 is released, τ_3 resumes its original priority of *p3*. In addition, task τ_2 is made ready. The scheduler then selects the next highest priority task eligible to run, which is τ_2. Note that τ_2 had been blocked on semaphore S_1. As a result of this, as it starts execution, it acquires the semaphore.

22. At time t_{21}, τ_2 locks S_4.

23. At time t_{22}, τ_2 locks S_5.

24. At time t_{23}, τ_2 releases S_5.

25. At time t_{24}, τ_2 releases S_4.

26. At time t_{25}, τ_2 completes its execution. Task τ_3 resumes execution.

27. At time t_{26}, τ_3 completes its execution.

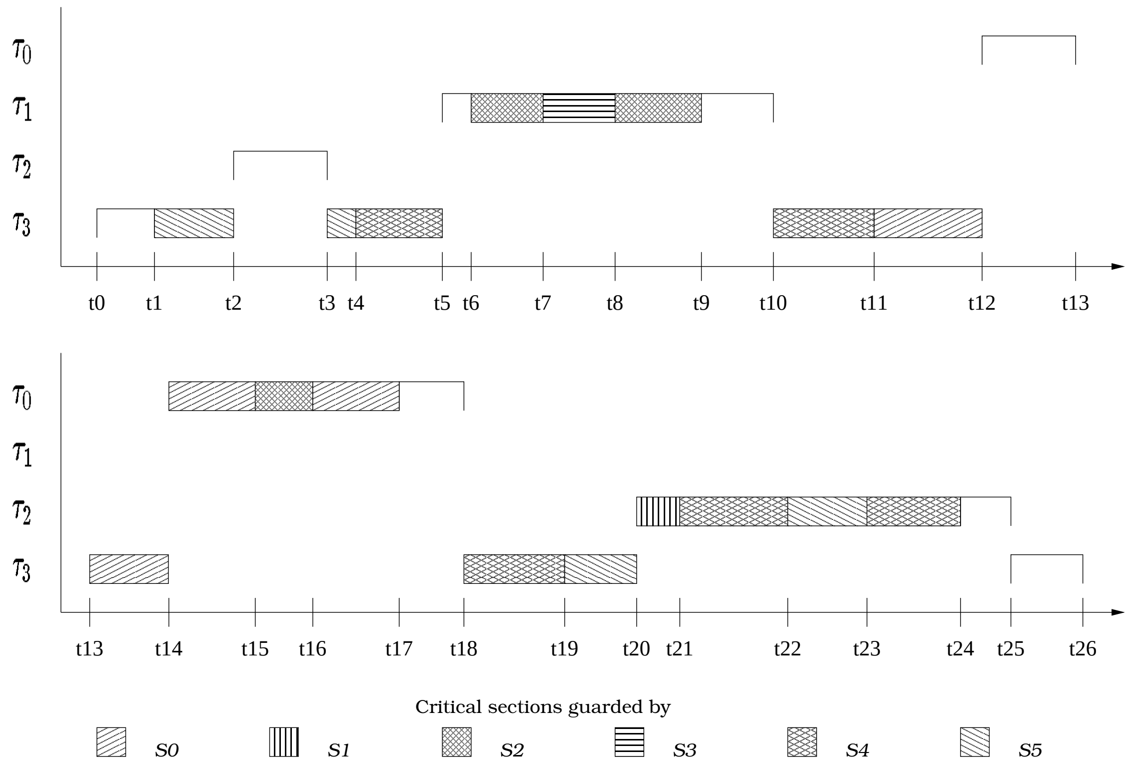

Figure 3.9: Sequence of Events Illustrating the Priority-Ceiling Protocol

The priority-ceiling protocol is a pessimistic protocol. There are situations where the priority-ceiling creates unnecessary blocking. In the above example, for instance, at time t_3, τ_2 blocks, even though it could have continued execution.

Despite such unnecessary blockings, this protocol dramatically improves the worst case blocking time of a task.

Priority-Inversion with Monitors The Priority-Inheritance algorithm has been studied and applied to problems involving avoidance synchronization (mutual exclusion). The algorithm itself, however, does not address issues involving conditional synchronization, which is an integral part of monitors. This section presents some initial new work on extending the algorithm to handle conditional synchronization.

In the priority-inheritance algorithm (and in real-time systems, in general), tasks waiting for entry into a monitor await in a prioritized entry *queue*. More specifically, each of the monitor's member routines have an entry *queue* associated with it, which is prioritized. When the monitor becomes unlocked, the next task that enters the monitor is the task with the highest priority among all the member routine's entry *queues*. Meanwhile, should a monitor be locked and a higher priority task arrives (desiring entry into the monitor), the current task executing inside the monitor "inherits" the priority of the highest priority task awaiting entry to the monitor, if the currently executing task's priority is less. This ensures that the task inside the monitor gets to execute with little interruption, thus allowing it to complete and leave the monitor, enabling the entry of the higher priority task.

Condition variables and their associated internal queues of waiting tasks are also a fundamental part of monitors [23]. Signalling a condition variable makes the task at the head of the queue eligible to run. The next task to execute in the monitor depends on the kind of monitor [10]. For this discussion it is sufficient to look at the major division of monitors into priority and no-priority monitors[4]. Priority monitors give preference to tasks that have already waited to enter the monitor and no-priority monitors do not. Hence, for a priority monitor, tasks in the monitor are considered more important than tasks waiting to enter the monitor, regardless of task priorities. This scheme makes reasoning about concurrent execution simpler, and prevents possible starvation of tasks.

No-priority monitors, such as Modula-3's monitors, behave very predictably with the priority-inheritance algorithm. Should a condition variable be signalled by a running task, the task at the head of the condition queue (or the highest priority task on a prioritized condition *queue*) is made eligible to run. This new eligible task is then inserted on the entry queue, to compete with other tasks awaiting entry to the monitor. In a no-priority monitor with non-blocking signals, the signaller continues execution after signalling, at the priority of the highest priority task awaiting entry to the monitor. In a no-priority monitor with blocking signals, the signaller is also moved out of the monitor and inserted on the entry queue, to compete with all other tasks awaiting entry to the monitor, as well as the signalled task. Since the signalled task (as well as the signaller in a no-priority monitor with blocking signals) is "moved-out" of the monitor, its priority returns to its normal or base priority, if it previously "inherited" a higher priority. As a consequence of this behaviour, priority-inversion is inhibited, in that the highest priority task is always executed before any lower priority task, despite the fact that a lower priority task may have already previously gained entrance to the monitor.

---

[4]Priority in this context is not a task's priority, but the way in which the monitor decides which task to execute next.

In a priority monitor, a task from a condition queue that is made eligible to run (by signalling the condition variable) is given preference over other tasks awaiting entry to the monitor. The next task that executes is either the signalling or signalled task (the discussion is independent of which executes), regardless of whether there are higher priority tasks waiting in the entry queues. This behaviour, in turn, creates the possibility of priority-inversion. Should a very high priority task be awaiting entry to the monitor, and a lower priority task executing in the monitor signals a condition queue whose most eligible task is also lower than the task awaiting entry to the monitor, priority-inversion results.

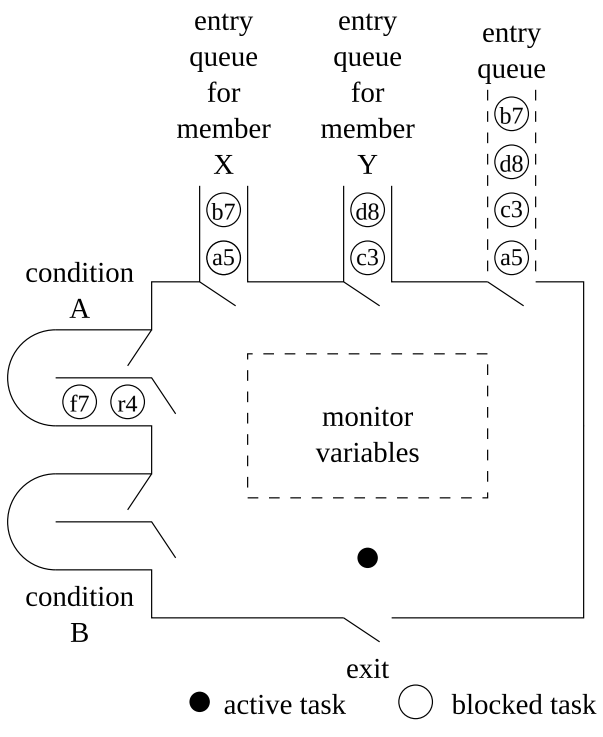

Figure 3.10: A Sample Monitor. Tasks are denoted by a letter followed by a number, which indicates its priority. Tasks awaiting entry or execution in the monitor wait on an entry queue. The head of the entry queue determines the next task to execute.

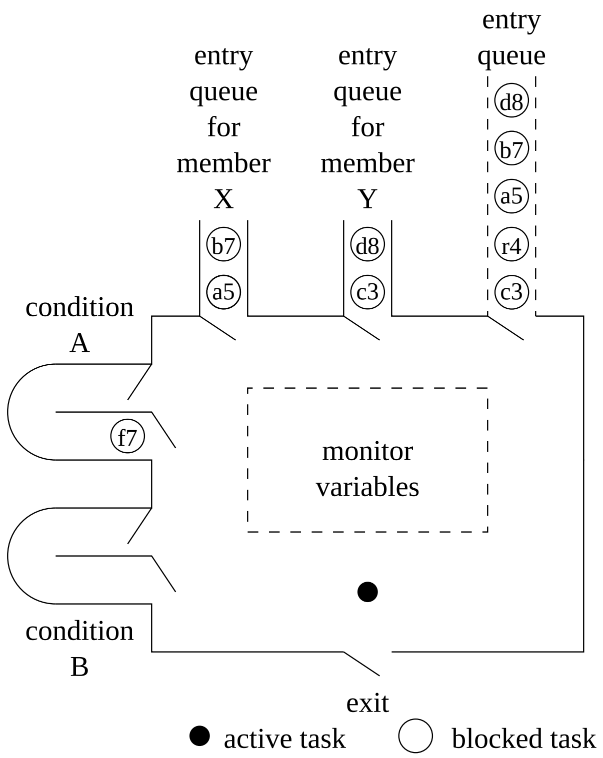

Figure 3.11: Behaviour of a No-Priority Monitor After Signalling Cond. Queue A

Depending on the kind of monitor, the execution state illustrated in figure 3.10 behaves differently when condition queue A is signalled. As shown in figure 3.11, for a no-priority monitor, when condition queue A is signalled, task *r4* is moved onto the entry queue. The entry queue continually maintains priority order, resulting in *r4* being executed only after task *c3* executes.

For a priority monitor, when condition queue A is signalled, the task at A's head is moved to a "signalled stack", as illustrated in figure 3.12. Task *r4* is then executed before any of the tasks on the entry queue, resulting in a definite priority-inversion in this example (since task *r4* executes before *c3*). The application of the priority-inheritance protocol on a priority monitor increases the original algorithm's

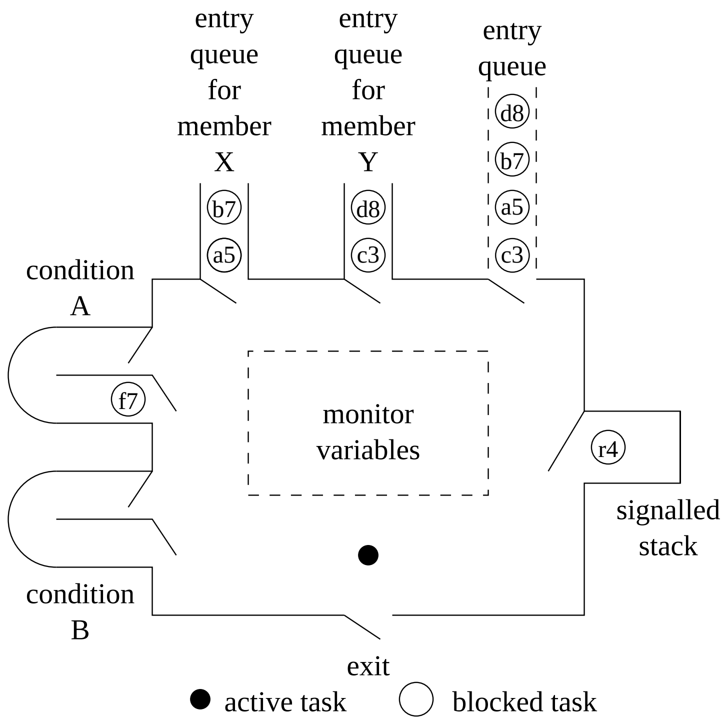

Figure 3.12: Behaviour of a Priority Monitor After Signalling Condition Queue A

bound for priority-inversion by the amount it takes to complete the execution of all the tasks in the signalled stack.

In μC++ , tasks running inside a monitor have the additional capability of being able to specifically accept any one of the monitor's member routines. Consequently, this capability brings about the possibility of bypassing higher priority tasks waiting on other routine's entry queues. When a member routine is accepted, the acceptor is moved to the signalled stack, thus causing the acceptor to block. The highest priority task waiting on the accepted member routine then executes. When a task leaves a monitor, the next task that executes within the monitor is determined by the following rule:

- If the signalled stack is not empty, the next task that executes is taken from the head of this stack.
- If the signalled stack is empty, the next task that executes is the highest priority task among all the entry queues of the member routines.

Following the same idea and principle as the priority-inheritance protocol, at any point in time, the task executing inside the monitor runs at its original priority or the priority of the highest-priority task among all the member routine's entry queues, whichever is higher. The time when priority-inversion can take place when accepting specific member routines is unbounded, since tasks can continually arrive on a member routine's entry queue, and tasks executing in the monitor can continually accept the specific member routine in question.

3.10 Multiprocessor Considerations

The discussions presented so far revolve around scheduling in a uniprocessor environment. Multiprocessor scheduling is problematic, since it has been shown by Mok and Dertouzos that the scheduling algorithms discussed so far are not optimal for multiprocessor systems.

To illustrate the poor performance of various scheduling algorithms on multiprocessor systems, consider a task set composed of tasks τ_1, τ_2, and τ_3, with corresponding periods (and deadlines) 20, 20, and 40 units; and corresponding computation time requirements of 10, 10, and 32 units. Two processors are available on the system.

With the rate-monotonic algorithm (or earliest deadline first algorithm), τ_1 and τ_2 are given highest priority, resulting in their execution on the two available processors. After τ_1 and τ_2 executes, τ_3 is ready for execution on either processor. Task τ_3 still needs 32 units of computation time. However, by this time, there is only 30 units of time left to τ_3's deadline. Consequently, task τ_3 cannot possibly meet its deadline. However, should task τ_1 and τ_2 be allocated to one processor, and τ_3 be allocated to another, there is no problem in meeting every task's deadlines.

Optimal scheduling in multiprocessor systems is known to be NP-hard [33]. As a result of this, research has been directed towards the simplification of the problem by determining optimal means of allocating tasks onto various processors, and having each processor schedule its task set, independent of the other tasks on other processors. This is often referred to as *static processor binding*. Though in principle, *dynamic processor binding* can yield better performance than static binding, the combinatorial analysis required during run-time is not feasible, especially for real-time systems.

Chapter 4

Implementation of Time-Defined Delays

4.1 The μC++ Execution Environment

In its current state, the μC++ programming language is implemented through a translator, which, in turn, produces C++ code. The program generated by the translator is then linked with a concurrent runtime library, which provides the necessary facilities for a concurrent execution environment [9].

At the heart of the μC++ runtime library is the μC++ kernel. The μC++ kernel is the main manager of resources for a μC++ program. The kernel is responsible for the scheduling and context-switching among the various μC++ tasks.

The resources managed by the μC++ kernel are abstracted into a structure called a *cluster*. A cluster is a collection of tasks and processors. The tasks in a cluster can either be blocked (or sleeping), be executing on any one of the processors within the cluster, or be ready for processor execution. Tasks that are ready for processor time are in a structure called a *ready-queue*. A cluster has one ready-queue, and any free processor within the cluster can execute any of the tasks waiting on the ready-queue. Within a μC++ program, several clusters may exist. Figure 4.1 illustrates a cluster.

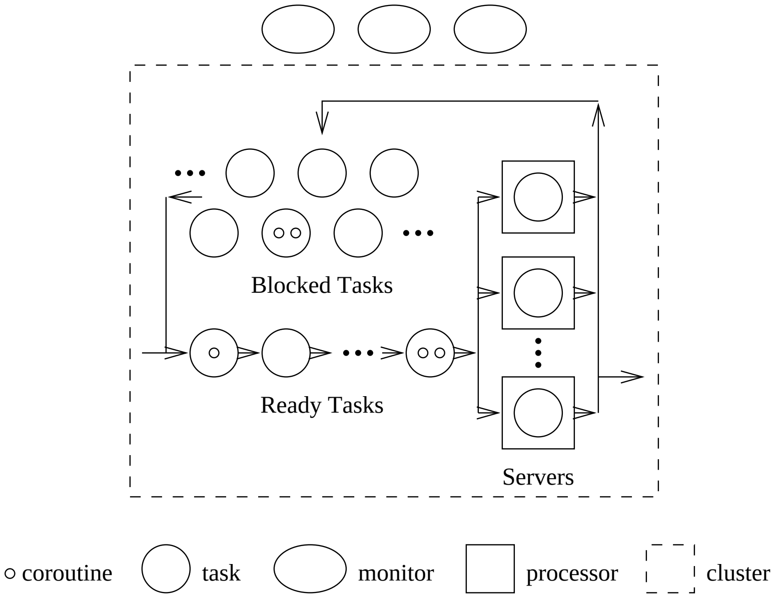

Figure 4.1: μC++ Cluster

A processor within a cluster is implemented as a Unix process (or kernel thread). On a uniprocessor version of μC++ , a single Unix process exists, which emulates the various processors within a cluster. On the multiprocessor version of μC++ , each processor within a cluster corresponds to an actual Unix process mapped on to the same memory location as other processors (or Unix processes). This memory mapping allows all the processors within a cluster to share certain data structures like the ready-queue, for example.

4.2 Timers

As discussed in chapter 2, time-defined delays are essential in real-time systems. The implementation of time-defined delays requires close interaction with one or more timer(s).

Timers are operating system entities responsible for notifying processes of the passage of time. Operating system timers, in turn, closely interact with the hardware clock. The hardware clock provides periodic interrupts to the operating system, notifying it that a certain time interval has elapsed.

4.3 Unix Signals and Alarms

In Unix, signals are used to notify processes of certain events. For example, when a user tries to suspend a Unix process, Unix sends a SIGTSTP signal to the process. The process, in turn, executes corresponding code associated with a signal. The corresponding code associated with a particular signal is referred to as a *signal handler*.

Unix offers three count-down timers for each process. One timer counts down in real-time (elapse time), one in user time (i.e., only when user code is executing), and one in user and system time (when user code is executing, and when the system is running on behalf of the process). When a count down timer reaches zero, Unix interrupts the process and has it execute a signal handler registered for the particular timer.

In the specific implementation of time-defined delays in μC++ , only the real-time timer is used, and it is called the SIGALRM timer. Unix signals provide the most efficient means by which a process can receive time notifications from the operating system, because the timer is asynchronous to the program, and, therefore, does not affect its execution until the timer expires. In fact, in the current implementation, time-defined delays are not the only functions relying on the SIGALRM timer. Scheduling (normal, as well as real-time) and other time-dependent functions all use the SIGALRM timer.

4.4 Data Structures Manipulated by the Signal Handler

The delivery of the interrupt for the SIGALRM timer is called a SIGALRM. The SIGALRM timer interacts with the application program through the execution of a signal handler. The signal handler, in turn, manipulates a set of *shared* data structures – shared among the real-time tasks, as well as the signal handler.

Whenever a SIGALRM occurs, the currently executing real-time thread is paused, pending completion of the SIGALRM signal handler. The signal handler modifies the shared data structure to affect the future behaviour of the system. After execution of the signal handler, the real-time thread resumes.

In μC++ , SIGALRMs are used for two important system functions: time-slicing and time-defined delays. Time-slicing is implemented by requesting periodic SIGALRMs, which forces a task scheduling to occur (in certain circumstances, the same task may be rescheduled).

Time-defined delays are implemented and managed by maintaining a **list of sleeping tasks** (or tasks that have requested a time-defined delay, and are awaiting wakeup) within a cluster. This list is chronologically ordered by each task's wakeup time.

As well, the SIGALRM timer must be set, to alert the system that a wakeup event must take place at a specified time. After a sleeping task is blocked, the processor is then free to execute the next ready task.

The fact that the SIGALRM timer serves more than one purpose – to alert the system that a sleeping task(s) need to be awaken, and to signal a time-slice – necessitates a **time-event data structure** that maintains and manages the set of time events in the system. Each task awaiting wakeup has a time event associated with it. In addition, time-slicing has a time event associated with it, which is not associated to any specific task. The SIGALRM timer is always set such that the outstanding SIGALRM is expected to arrive at the time specified for the event located at the head of the time-event data structure. Any change at the head of this list necessitates resetting the SIGALRM timer (either setting, resetting, or cancelling the SIGALRM timer). Figure 4.2 illustrates a time-event structure.

A race condition exists between the arrival of a previously requested SIGALRM and a change of the SIGALRM timer. This situation is handled by first manipulating the time-event structure before making the operating system call to set the SIGALRM timer. Should a SIGALRM occur while setting the next SIGALRM, the time-event structure is up to date. The time-event structure is, in turn, evaluated by the SIGALRM handler to see if the arriving SIGALRM matches the event at the head of the time-event structure. Should this not be the case, the SIGALRM is considered spurious and ignored (It turns out that other spurious SIGALRMS can occur due to Unix process to process communication in μC++ . It is, therefore, important that these situations be detected and ignored.).

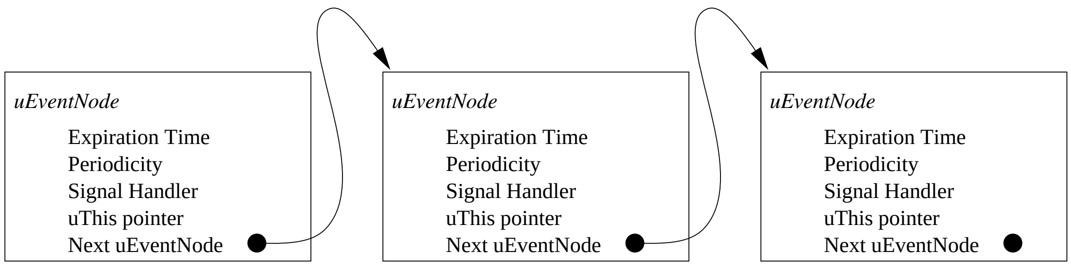

Figure 4.2: Time-Event Structure

It is important to note that the time-event structure may have several events scheduled for the same time, because several tasks can request to be awoken at the same moment. If a time-slice event happens to be scheduled for the same time as other "wake-up" event(s), however, the time-slice event is always inserted at the end of the set of events with the same time expiration. To illustrate the importance of this insertion rule, assume that a very high priority task is scheduled to be awoken at the same time as a time-slice event. If the time-slice event precedes the "wake-up" event, upon execution of the time-slice event, the most eligible task from the ready queue is scheduled. However, since the "wake-up" event has not yet been executed, the very high priority task is not on the ready queue upon execution of the time-slice, resulting in the high priority task's deferred execution till a subsequent time-slice event, despite the fact that it was eligible to execute at the time the initial time-slice took place, which results in *priority inversion* (section 3.9.2.3 describes priority inversion). Uncontrollable priority inversion is unacceptable in real-time systems; processing "wake-up" events prior to a time-slice event ensures that this form of priority inversion does not occur.

The SIGALRM handler starts by popping the first expiring event from the time-event structure to determine the nature of the event requesting the SIGALRM. Should the popped event be a periodic event (such as time-slice event), a new event for the next period is inserted in the appropriate location in the time-event structure. The popped event is, in turn, examined. If the event is for a wake-up request for a sleeping task, the `wake-up-handler` is executed. The wake-up handler removes the associated sleeping task from the list of sleeping-tasks, and inserts it in the ordered ready-queue (thus, every "wake-up" event in the time-event structure has a corresponding entry in the list of sleeping tasks). If the alarm is for a time-slice, the `time-slice-handler` is executed. The time-slice handler puts the current task executing (should one exist) at the end of the ordered ready-queue, and schedules a new task from the ready-queue.

The SIGALRM handler repeats the above steps until all the handlers for the expired events have been executed. After execution of the last handler in the set of expiring events, the SIGALRM timer is reset to the expiration time of the next event in the time-event structure (also see section 4.5). Figure 4.3 illustrates the behaviour of the SIGALRM handler.

```
Handler() {
   while (CurrentTime later than or equal to head event in
          time-event structure) {
               event = pop event in head of time-event structure
               if (event is periodic)
                    insert event in time-event structure
               case (event) {
                    wake-up event:
                            do wake-up-handler();
                            break;
                    time-slice event:
                            do time-slice-handler();
                            break;
               }
   }
   reset alarm request from Unix
}
```

Figure 4.3: Signal Handler Code for the Manipulation of the Time-Event Structure

4.5 Time-Granularity of Alarms

Each Unix system has a minimum time interval at which alarms may arrive. Should an alarm be requested that is smaller than this minimum inter-arrival time, the SIGALRM request arrives later than the requested time – at the minimum inter-arrival time.

To compensate for the possibility of a late alarm arrival (and, in turn, a late wakeup time for a time-defined delay request), when an event in the time-event structure is evaluated with an expiration time within the current time and the minimum arrival time of a SIGALRM, the event is popped from the time-event structure, and the appropriate handler is executed. The possibly premature execution of the event handler whose expiration has yet to arrive may cause problems for certain real-time applications. The underlying assumption for this default behaviour is that it is often better to be early than to be late – obviously, such philosophy does not apply to all systems. This feature can, in turn, be disabled by a simple redefinition of a compiler preprocessor constant.

4.6 The Problem Caused by Unix Alarms' Association with a Process

In Unix, timers and signals are bound to a process. On the multiprocessor version of μC++ , this implies that timers and their signals are associated with μC++ processors, because processors are implemented as Unix processes (see section 4.1 for a discussion of the μC++ runtime environment). On a cluster containing several processors, several alarms can, therefore, exist.

As further described in section 4.1, every task within a cluster has the possibility of executing on any of the processors belonging to the same cluster. As such, any of the tasks requesting time-defined delays (thus, setting the Unix timer), for example, may make such requests from any of the processors in the cluster.

The fact that tasks can issue time-defined delays from any processor in a cluster introduces a management problem: What is the most appropriate means of managing the different time requests issued on different processors, where each processor has its own timer? The previous discussion of a single time-event structure (see section 4.4) is significantly complicated, as is managing the timer that causes the SIGALRM.

A possible solution to the management problem is to designate one processor within a cluster to handle Unix alarm requests (in short, the list of sleeping tasks and time-event structure is only accessed by one process). Whenever a processor encounters a time-defined delay request from a task it is executing, an alarm request is passed on to the designated processor responsible for alarm requests. Though possible, such an implementation suffers from inefficiencies resulting from the fact that communication among processors is expensive. Statically analyzing programs to determine which tasks utilize time-defined delays, and designating such tasks to a specific processor is another option. Another option is dynamically migrating tasks to a specific processor when it requests a time-defined delay. However, such options are extremely naive. If every task on the system requests time-defined delays, the remaining non-SIGALRM-designated processors would merely be sitting idle.

Rather than attempting to centralize management of the time-event structure, a more distributed approach turns out to be better. By sharing the list of sleeping tasks and the time-event structure among processors, it is possible to eliminate the inefficiency previously described, by not having one statically designated processor handling time-defined delays, but by having the responsibility be "randomly" designated among processors. If a task requests a time-define delay, for example, an appropriate event is inserted into the shared time-event structure (as well, the list of sleeping tasks is updated). If the event that is added will expire sooner, a SIGALRM is requested by the processor running the task in question. This processor requesting the latest SIGALRM is then the designated SIGALRM processor. Should a processor on the same cluster (other than the newly designated SIGALRM processor) have an outstanding SIGALRM request for a particular task on the list of sleeping tasks, that SIGALRM will be dealt with as a spurious event because it does not match the time of the event at the beginning of the event list. When the valid or current SIGALRM does come in, the designated processor services all the expiring events on the time-event structure, and requests a new SIGALRM for the next event that will become due. In short, one processor is designated as the manager of the list of sleeping tasks and the time-event structure. There is no binding between the task requesting a time-defined delay, and the host processor it requested a delay from. Processors "randomly" take turns managing alarms and the data structures associated with them.

The above solution eliminates the bottleneck and inefficiency of having merely one processor do all the Unix alarm chores. Some form of interprocessor communication still exists, however, in that previously designated SIGALRM processors need to somehow be notified when a new SIGALRM processor is designated. Thus, the above solution is not spared from the inefficiencies inherent in interprocessor communication.

Sharing the time-event structure and/or list of sleeping tasks requires mutually exclusive access of the list(s), which introduces additional overhead. By having each processor maintain its own list of sleeping tasks and time-event structure, which creates true independence among processors, it is possible to eliminate the sharing cost. When a task requests a time-defined delay on a processor, the task becomes bound to the host processor until the task re-awakens. After the task re-awakens, the waking task is re-inserted by its host processor to the shared ready-queue, making the task free to re-execute on any of the cluster's processors.

The processor-independent nature of the last solution implies that occurrences of time-slice events can occur much more randomly, independently, and efficiently, given that each processor keeps track of and controls its own time-slice event. Furthermore, the management of SIGALRMs is more fairly distributed among processors, in much the same way as each task on the ready-queue can be randomly executed by any of the cluster's processors.

4.7 The Mutual Exclusion Problem Involving Alarms

The time-event structure/list of sleeping tasks is the primary data structure manipulated by the SIGALRM handler. Furthermore, should a SIGALRM be caused by a time-slice event, certain μC++ kernel data structures have to be read and modified.

The asynchronous and sporadic nature of SIGALRM arrivals creates problems, in that SIGALRMs can arrive while the above data structures are in the process of being modified by the μC++ kernel. It is imperative, therefore, that these structures be in a consistent state before the SIGALRM handler is allowed to execute. To ensure the consistency of shared data structures, a typical solution is to utilize a lock to provide mutual exclusion of the shared data. With the SIGALRM handler, however, utilizing locks can create problems. Assume, for a moment, that the SIGALRM handler happens to arrive and execute on the same thread as the owner of the locked data structure. Upon arrival of the SIGALRM, the task owning the lock is blocked (only to resume after the SIGALRM handler exits), and the SIGALRM handler is executed. Since the SIGALRM handler cannot acquire the lock to the shared data, it blocks, which results in a deadlock.

A solution that does not use locks for the shared data structures is possible. However, a lock-free data structure usually requires reference counts and/or shadow copies of the shared structure [22]. Therefore, such solutions are usually expensive, and imposes significant overheads on the system.

A possible solution for enforcing mutual exclusion, without the signal handler blocking, is called *roll-forwarding*[40]. Bershad[8] also posed a very similar solution to this problem. Roll-forward is a technique that defers the execution of asynchronous or sporadic interrupt code, should the code be precluded from executing at its arrival time. The deferral is due to an inconsistent data structure, or merely the fact that the operation performed by the interrupt code is undefined during the current execution. Roll-forward transfers control back to the interrupted code, and after the interrupted code completes, the deferred interrupt code is executed.

Interrupt code is usually of high priority. Since roll-forward code affects the execution of the interrupt code, roll-forwarding must be used with caution. Roll-forwarded code should be as small as possible, thus minimizing the time delay before interrupt code gets executed.

Using roll-forwarding, a general solution to the mutual exclusion problem involving Unix signals is outlined in Figure 4.4. In this solution, upon execution of the signal handler, semaphore[1] L is checked to see if the shared data structure is in use. Note that semaphore L is not P'd (it is only checked), thus, the signal handler

[1]A semaphore is used to guarantee mutual exclusion among normal routines manipulating the shared structure. If only one routine manipulates the structure, and the only mutual exclusion that needs to be guaranteed is between the routine and the signal handler, a lock-free data structure can be implemented. In fact, a flag alone, which indicates whether the routine is in the process of manipulating the shared structure, is sufficient.

never blocks. If semaphore L is unlocked, work CC (the actual work of the signal handler) is executed. If on the other hand, semaphore L is locked (some function that manipulates the shared data structure is executing when the SIGALRM is delivered), execution of CC is deferred and left for the interrupted code to execute as soon as it completes its critical section and the shared data structures are in a consistent state.

```
//RollForwardFlag is initialized to false
Signal_Handler_Function() {
   if (lock L is free) { //L guards the shared data structure
       do CC() //CC is some work utilizing the shared structure
   }
   else {
       set RollForwardFlag to true
   }
}

Outline_of_Functions_Manipulating_Shared_Structure() {
   p(L) //L is a semaphore used by all functions accessing the
        //shared data structure to guarantee mutual exclusion
      manipulate shared structure (this is the critical section)
      { //block B
          if (RollForwardFlag is true) {
             do CC()
             set RollForwardFlag to false
          }
      } //end of block B
   v(L) //release semaphore L
}
```

Figure 4.4: Roll-forward Solution with a Lock

The above solution solves the mutual exclusion problem of the shared data structure, without the signal handler blocking. However, another restriction arises: block B (starting at line 14) cannot be interrupted. To understand this requirement, assume block B can be interrupted. While executing line 15, assume RollForwardFlag evaluates to false, and right after this evaluation, a SIGALRM occurs. The signal handler evaluates L to be locked, and sets the flag RollForwardFlag to true. The signal handler exits, and the interrupted function resumes execution. Prior to interruption, the RollForwardFlag evaluates to false, so the function exits block B, and releases semaphore L. Execution of `CC` is missed.

A solution guaranteeing the atomicity of block B is to block SIGALRMs prior to entering B. After execution of block B, SIGALRM signals are resumed. When a signal is blocked, the execution of the signal handler is consequently blocked. While blocked, signals are queued, not lost. Thus, execution of the appropriate signal handler occurs after unblocking the signal.

The blocking and unblocking of signals is, however, an unsatisfactory solution. Blocking a signal is expensive, and the possibility that a function is interrupted within block B is rare.

In the μC++ implementation of time-defined delays, the above problem is circumvented by associating certain data structures with a processor, rather than sharing it among processors within a cluster (see Section 4.6). By having only one processor manipulate the critical data structures, and by integrating all manipulation of such structures inside the "kernel" (a subset of code that is executed serially and uninterrupted), the use of locks becomes unnecessary. Also, by incorporating, within the kernel, code that manipulates the critical data structures, concurrent access to the critical structures is highly minimized. The special set of code that manipulates the critical structures (the interrupt code) is the only other means by which concurrent access to the critical data structures can possibly takes place, and the the roll-forward solution nicely guarantees mutual exclusion for this scenario.

```
//RollForwardFlag is initialized to false
Signal_Handler_Function() {
   if (RollForwardFlag is true)   return;
   if (DisableInterrupts is false) {
       do Handler()
   }
   else {
       set RollForwardFlag to true
   }
}

Some_Kernel_Code() {
   some kernel chores ...
   set DisableInterrupts to true
       manipulate shared structure (this is the critical section)
   set DisableInterrupts to false
   if (RollForwardFlag is true) {
       do Handler()
   }
   some kernel chores ...
}
```

Figure 4.5: Roll-forward Solution to Kernel Interruption

The solution illustrated in Figure 4.5 (figure 4.3 in Section 4.4 defines function `Handler`) roughly illustrates roll-forwarding in the implementation of time-defined delays (and, consequently, the μC++ kernel). A SIGALRM is the only form of interrupt where roll-forwarding is utilized. Furthermore, the default action in Unix is to mask SIGALRMs until after the SIGALRM handler has completed. This default action implies that while `Handler` is executing, no further alarms can come. Consequently, the RollForwardFlag needs to be reset to false after the chores in `Handler` are executed, but before resetting the Unix alarm (line 16 of Figure 4.3).

Chapter 5

Notion of Time in Real-Time Programming

In any real-time programming environment, a time base must exist from which jobs and functions are scheduled. The very nature of time-constrained computing requires frequent access to a clock and, in turn, a need for a convenient means by which a clock can be queried [2, 24].

Queries to clocks yield time. The convenient manipulation of time is yet another essential characteristic in any time-constrained environment. Manipulating time, in turn, yields another metric that expresses a span or duration of time.

The Ada language provides a *CALENDAR* library package, which is an interface that allows for the convenient querying and manipulation of time. In addition, the Ada95 standard [25] defines an *Ada.Real-Time* package, which is a high-resolution monotonic clock library. *Ada.Real-Time* extends the *CALENDAR* library by providing additional facilities for manipulating time to the nanosecond level of granularity (in much the same way the POSIX real-time extensions does for Unix and POSIX threads). Ada's clock package arguably provides a very sophisticated level of services, in comparison to other clock facilities provided by other languages suitable for real-time development.

C and C++ on the other hand, provides a primitive interface for querying time. Though the standard C library defines various units by which time can be queried, the standard does not define any means by which time units can be easily and flexibly manipulated. Manipulation of time entails the manual manipulation of the internal representation of time, which, on most current Unix systems, is in the form of two long integers representing the number of seconds and micro or nanoseconds elapsed since the Unix epoch (defined as 1970 Jan 1 00:00:00:000000000).

In making μC++ a more suitable platform for real-time development, an interface is provided that encapsulates the internal representation of clock, time, and duration. As a result, units of time can be easily manipulated, without involving the programmer in implementation and architecture-dependent details. The μC++ Calendar package, in turn, combines the power of Ada's *CALENDAR* and *Ada.Real-Time* library packages.

5.1 Duration

`uDuration` is a class whose instances represent a span of time. By subtracting two time values, for instance, the resulting difference is a time span.

The creation and manipulation of `uDuration` values are performed by the following routines:

```
class uDuration {
  public:
    uDuration() {};
    uDuration(long seconds, long ssec);
    uDuration(long seconds);
    operator struct timeval();
    friend uDuration operator-(uDuration g1, uDuration g2);
    friend uDuration operator+(uDuration g1, uDuration g2);
    friend uDuration operator*(uDuration g1, long int g2);
    friend uDuration operator/(uDuration g1, long int g2);
    friend bool operator>(uDuration g1, uDuration g2);
    friend bool operator<(uDuration g1, uDuration g2);
    friend bool operator>=(uDuration g1, uDuration g2);
    friend bool operator<=(uDuration g1, uDuration g2);
    friend bool operator==(uDuration g1, uDuration g2);
    friend bool operator!=(uDuration g1, uDuration g2);
}
```

Usually, `uDuration` objects are created implicitly when manipulating time (see Section 5.2 regarding additional functions that provide for the generation of uDuration objects). However, `uDuration` objects can also be assigned duration values, by specifying seconds and nanoseconds in a `uDuration`'s constructor, for example:

```
uDuration x(4,447398253); //Declaration of uDuration object x,
                          //where x represents a duration of
                          //4 seconds and 447398253 nanoseconds
uDuration x();            //Declaration where x is of zero duration
uDuration x(5);           //Declaration where x is 5 seconds
```

Direct manipulation of `uDuration` objects is illustrated as follows:

```
uDuration x, y, z; int i;

x = y + z; //Addition of two uDuration objects.  Sum is a uDuration.
x = y - z; //Subtraction of two uDuration objects.  Difference
           //is a uDuration.
x = y * i; //uDuration object x is a multiple of y (i multiples).
x = i * y; //uDuration object x is a multiple of y (i multiples).
x = y / i; //uDuration object y is divided by an integer i.
```

In addition, relational comparison operators are defined for `uDuration` objects: ($>$), ($<$), ($>=$), ($<=$), ($!=$), and ($==$). All relational operators return a boolean.

5.2 Time

`uTime` is a class, whose instance represent a static value depicting a particular time. A `uTime` object can, therefore, be created by specifying various arguments such as a year, month, day, hour, minutes, seconds, and nanoseconds. The specified arguments indicate a particular time a `uTime` object represents. It is important to note, however, that the specified parameters and, in turn, instances of `uTime`, themselves, cannot represent time prior to the Unix epoch.

As with `uDuration` objects, certain mathematical and relational operations can be performed on `uTime` objects. A duration, for example, may be added to or subtracted from a `uTime` object to yield a new instance of `uTime`. In addition, two different times can be subtracted from each other to obtain a certain duration. Comparison operations on `uTime` behave similarly as those defined for `uDuration`.

```
class uTime {
  public:
    uTime();
    uTime(int yr, int mnth, int day, int hr, int min,
          long sec, long ssec);
    uTime(int mnth, int day, int hr, int min, long sec, long ssec);
    uTime(int day, int hr, int min, long sec, long ssec);
    uTime(int hr, int min, long sec, long ssec);
    uTime(int min, long sec, long ssec);
    uTime(long sec, long ssec);
    uTime(long sec);
    operator struct timeval();
    friend uTime operator+(uTime x, uDuration y);
    friend uTime operator+(uDuration y, uTime x);
    friend uTime operator-(uTime x, uDuration y);
    friend uDuration operator-( uTime g1, uTime g2 );
    friend bool operator>(uTime g1, uTime g2);
    friend bool operator<(uTime g1, uTime g2);
    friend bool operator>=(uTime g1, uTime g2);
    friend bool operator<=(uTime g1, uTime g2);
    friend bool operator==(uTime g1, uTime g2);
    friend bool operator!=(uTime g1, uTime g2);
}
```

The `uTime` constructors are overloaded so there is a choice of specifying the arguments, based on varying units of time (which are differentiated by the number of arguments). The arguments of `uTime` constructors have the following meanings:

- *yr* - the number of years since 1970;

  default value is 1970.

- *mnth* - the number of months since January [0-11];

  default value is 0.

- *day* - the day of the month [1-31];

  default value is 1.

- *hr* - the number of hours since midnight [0-23];

  default value is 0.

- *min* - the number of minutes after the hour [0-59];

  default value is 0.

- *sec* - the number of seconds after the minute [0-60];

  default value is 0.

- *ssec* - the number of nanoseconds past *sec*;

  default value is 0.

It is important to note, however, that the various arguments for `uTime`'s constructors can exceed the "logical" ranges specified between the brackets ([ ]) above. For example, the following declaration is valid:

```
uTime x(1,48,0,60,1000000000); //because there are five arguments,
                               //uTime(int day, int hr, int min,
                               //long sec, long ssec) is called.
                               //x is init. with the logical value
                               //(1970 Jan 3 0:01:01:000000000)
                               //(assuming timezone is GMT)
uTime x(818227413, 0);         //uTime(long sec, long ssec) is called.
                               //x is init. with the logical value
                               //1995 Dec 6 00:23:32:000000000
                               //(assuming timezone is EDT)
```

5.3 Clocks

Clocks are objects whose fundamental purpose is to furnish a dynamic property called time. `uClock` is a class whose instances specify a monotonically increasing value, i.e., clocks are entities that continually count up by a fixed duration amount.

The `uClock` package is designed to support the "multi-clock" environment essential in various real-time platforms. It is common for real-time systems to have various high-precision clocks, for instance, used in various time-critical applications. In its current implementation, `uClock` relies on POSIX system calls, which conveniently query various hardware clocks present on the system. In turn, the use of hardware clocks other than the default system clock requires the presence of the clock/time POSIX libraries.

The design of the `uClock` package allows for the definition of unlimited *virtual clocks*. Instances of virtual clocks are no more than independent clocks possessing their own time bases. It is possible to create a clock possessing a time other than the current system time, for instance. The only restriction is that the clock times must be greater than the Unix epoch.

```
class uClock {
  public:
    uClock(void);
    uClock(int clock_id);
    uClock(uTime t);
    void uGetTime(int &yr, int &mth, int &day, int &hr,
                  int &min, int &sec, long &ssec);
    uTime uGetTime();
    void uResetClock(uTime t);
    void uResetClock(int clock_id);
    void uResetClock();
}
```

Several overloaded constructors are available for creating instances of `uClock`:

- `uClock(void)` - `uClock` objects created with this constructor use the normal or default system-wide clock (the clock used by the Unix operating system for its normal clock or time work).
- `uClock(int clock_id)` - `uClock` objects that actually correspond to a hardware clock can also be declared. The corresponding constructor for this declaration involves an integer parameter. To declare a `uClock` object that corresponds to the normal or default system-wide clock, the *clock_id* parameter must be zero.
- `uClock(uTime t)` - `uClock` objects are usually defined by specifying a time base by which the `uClock` object functions from. The specified time base (defined by *t*), indicates the start time from which the `uClock` object counts upwards. The `uClock` object created through this method is a *virtual clock*.

From `uClock` objects, time can be extracted using `uGetTime`. `uGetTime` is overloaded, in that it may return a `uTime` object as the extracted time of the particular `uClock` object, or it may accept a set of six variables as its parameters, in which the logical time is assigned.

The base time of a `uClock` object can be changed after its creation. This is accomplished with the `uResetClock` function. The `uResetClock` member, without any parameter, simply resets the clock object to the same time as the default system clock. With a `uTime` parameter, `uResetClock` automatically transforms the `uClock` object to a virtual clock, if it is not already one. The time base of the `uClock` object is reset to the time specified in the `uTime` parameter. With an integer parameter, `uResetClock` corresponds to the hardware clock specified by the integer argument. As usual, an argument of zero sets the `uClock` object to correspond to the default system clock.

Chapter 6

Real-time Constructs

Without programming language facilities that allow for the specification of time constraints and behaviours, real-time programs are usually limited to cyclic executives, and the likelihood of encountering timing errors increases, as manual calculations are utilized. The introduction of real-time constructs is a necessity for accurately expressing time behaviour, as well as providing a means for the runtime program to evaluate whether any timing constraints have been broken. Furthermore, explicit time-constraint constructs can drastically minimize coding complexity as well as analysis. Various programming language constructs for real-time environments are discussed in [45, 38, 35, 28, 27, 26, 20, 16, 14, 43].

6.1 Construct for the Expression of Time-Defined Delays

In the Ada programming language[25, 48], a time-defined delay is expressed by either of two constructs:

`delay` *delaytime*

`delay until` *delaytime*

`delay` specifies a delay time relative to the outset of the statement declaration (duration), whereas `delay until` specifies an absolute delay time.

In μC++ , time-defined delays are implemented as function calls to the μC++ kernel. As such, the flexibility of C++ allows for the overloading of functions, which eliminates the need to have two different forms of time-delay specification:

```
void uSleep(uDuration duration);
void uSleep(uTime time);
```

With a `uDuration` parameter, `uSleep` works in the same way as Ada's *delay* statement. Given a `uDuration` parameter, the task requesting a time-defined delay sleeps for the span of time indicated by the duration parameter. A `uTime` parameter behaves the same as Ada's *delay until* statement. Since `uTime` specifies an absolute time, the task requesting a time-defined delay sleeps until the specified absolute time. Note that should the `uDuration` argument be less than or equal to a duration of zero, the task in question does not sleep and simply ignores the request. Similarly, a `uTime` parameter specifying an absolute time that has already passed results in the request being ignored.

6.2 Specification of Periodic Tasks

Without a programming language construct to specify periodicity, and without programming language facilities to express time, it is almost impossible to accurately express time specifications within a program. Specifying a periodic task in a language without proper time constructs, for example, can introduce catastrophic inaccuracies. For example, in

```
1   for ( ; ; ) {
2      // Some periodic chores
3      uDuration DelayTime = NextTime - CurrentTime;
4      uSleep(DelayTime);
5   }
```

should the task in question be context-switched out after executing line 3 (or be context-switched out after the `CurrentTime` is evaluated in line 2), the `DelayTime` would be inaccurate. As a result, the sleep time of the program is erroneous.

The above problem can be eliminated by specifying an absolute time to `uSleep` (specifying "`NextTime`" as the parameter to `uSleep`). However, with this form of periodic task specification, it is infeasible to specify other forms of deadlines. Ada only supports the periodic task specification using delays, and the system guarantees that a periodic task delays for a minimum time specified in *DelayTime*, but makes no guarantee as to when the periodic task actually gets to execute [7]. As a result, a task can request to be put to sleep for 10 seconds (and Ada would guarantee that it sleeps for at least 10 seconds), but end up executing 20 seconds later.

In μC++ , a periodic task is defined in a similar manner as a class. A periodic task has, in fact, all the properties of a class, with the addition that a task of any sort (be it periodic, aperiodic, or a normal task) possesses its own thread of control, execution-state, and mutual exclusion. Task constructors are executed by the calling thread, after which the newly created task starts execution with its own thread. A task's thread of execution initiates with `main`. Note that `main` has no parameters. Information that needs to be passed to `main` may be done through the constructor. A periodic task has the following form:

```
uPeriodicTask task-name {
   private:
      // ...
   protected:
      // ...
      void main();
   public:
      // ...
};
```

A `uPeriodicTask` type, if not derived from some other type, is implicitly derived from the task type `uPeriodicBaseTask`, as in:

```
uPeriodicTask task-name : public uPeriodicBaseTask {
   private:
      // ...
   protected:
      uDuration uPeriod;
      uTime uFirstActivateTime;
      uEvent uFirstActivateEvent;
      uTime EndTime;
      uDuration Deadline;
      // ...
   public:
      uPeriodicBaseTask(uDuration Period,
                        uCluster &cluster = uThisCluster());
      uPeriodicBaseTask(uDuration Period, uTime FirstActivate,
                        uTime EndTime, uDuration Deadline,
                        uCluster &cluster = uThisCluster());
      uPeriodicBaseTask(uDuration Period, uEvent FirstActivate,
                        uTime EndTime, uDuration Deadline,
                        uCluster &cluster = uThisCluster());
      uPeriodicBaseTask(uDuration Period, uTime FirstActivate,
                        uEvent FirstActivate, uTime EndTime,
                        uDuration Deadline,
                        uCluster &cluster = uThisCluster());
};
```

Periodic tasks may be initiated by two means. The first is by specifying a start time at which the periodic task begins execution. The second is by specifying an event (an interrupt), such that upon receipt of the event, the periodic tasks begins. If both time and event are specified, the task starts either on receipt of an event or when the specified time arrives, whichever comes first. If no time or event is specified, the periodic task may start at any time. End times may also be specified. When a specified end time arrives, the periodic task halts after the execution of the current period. The *Deadline* parameter indicates the deadline of a task, expressed as a duration from the beginning of a task's period. A 0 (zero) as an argument for any of the parameters indicates that the task in question is free from the constraints represented by the parameters (the exception is *Period*, which cannot have a zero argument). For example, should the *FirstActivate* parameter be zero, this implies that the task is scheduled for initial execution at the next available time the system can accommodate it. Finally, the *cluster* parameter specifies which cluster the task should be created in. Should this parameter be omitted, the task is created on the current cluster.

An example of a periodic task declaration is to start at a specified time and execute indefinitely (without any deadline constraints):

```
uPeriodicTask sample-task {
   protected:
      void main() {
          //periodic task body
          //...
      }
   public:
      sample-task(uDuration period, uTime time) :
                uPeriodicBaseTask(period, time, 0, 0) {};
};
```

6.3 Specification of Sporadic Tasks

Sporadic tasks are not as predictable as periodic tasks, in that the ready time of a sporadic task is not always within a constant duration, as is with periodic tasks. There is a sense of predictability, however, in that there is a minimum duration between the arrival of the same sporadic task.

Sporadic tasks are defined in a similar manner as periodic tasks, with the exception of the *Period* parameter. In the declaration of a sporadic task, the minimum inter-arrival time of the sporadic task is specified as a parameter, called *Frame*.

A `uSporadicTask` type, if not derived from some other type, is implicitly derived from the task type `uSporadicBaseTask`, as in:

```
class uSporadicTask : public uSporadicBaseTask {
   protected:
      uDuration uFrame;
      uTime uFirstActivateTime;
      uEvent uFirstActivateEvent;
      uTime EndTime;
      uDuration Deadline;
      // ...
   public:
      uSporadicBaseTask(uDuration Frame,
                        uCluster &cluster = uThisCluster());
      uSporadicBaseTask(uDuration Frame, uTime FirstActivate,
                        uTime EndTime, uDuration Deadline,
                        uCluster &cluster = uThisCluster());
      uSporadicBaseTask(uDuration Frame, uEvent FirstActivate,
                        uTime EndTime, uDuration Deadline,
                        uCluster &cluster = uThisCluster());
      uSporadicBaseTask(uDuration Frame, uTime FirstActivate,
                        uEvent FirstActivate, uTime EndTime,
                        uDuration Deadline,
                        uCluster &cluster = uThisCluster());
};
```

6.4 Specification of Aperiodic Tasks

Aperiodic tasks have no deterministic ready-time patterns. As a result, aperiodic tasks can, at most, be used in soft real-time applications.

An aperiodic task is defined as a `uRealTimeTask`. If a `uRealTimeTask` type is not derived from some other type, it is implicitly derived from the task type `uRealTimeBaseTask`, as in:

```
class uRealTimeTask : public uRealTimeBaseTask {
   private:
      // ...
   protected:
      uTime uFirstActivateTime;
      uEvent uFirstActivateEvent;
      uTime EndTime;
      uDuration Deadline;
      // ...
   public:
      uRealTimeBaseTask(uCluster &cluster=uThisCluster());
      uRealTimeBaseTask(uTime FirstActivate, uTime EndTime,
                        uDuration Deadline,
                        uCluster &cluster=uThisCluster());
      uRealTimeBaseTask(uEvent FirstActivate, uTime EndTime,
                        uDuration Deadline,
                        uCluster &cluster=uThisCluster());
      uRealTimeBaseTask(uTime FirstActivate, uEvent FirstActivate,
                        uTime EndTime, uDuration Deadline,
                        uCluster &cluster=uThisCluster());
};
```

6.5 Exception Handling

In the design and implementation of real-time programs, various timing constraints are guaranteed through the use of scheduling algorithms, as well as the utilization of exception facilities.

Exception facilities are extremely crucial in real-time systems, as they allow a system to react to an error in a deterministic fashion. Hecht [21] demonstrated, through various empirical studies, that the introduction of even the most basic fault-tolerance mechanisms into a system drastically improves the reliability of a real-time system.

μC++ is extending the basic exception constructs of C++ to deal with concurrency, as illustrated in figure 6.1. Should an exception condition be raised from within the `uTry` block, the appropriate or matching handler is executed from within the declared handlers. The future behaviour of the exception environment is defined by the following possible options for dealing with an exception situation:

- `uCatch` - deals with the exception situation in the same way C++ deals with exceptions. Should an error condition occur, program execution transfers from the point where the exception is raised, to the exception handler. After execution of the exception handler, the task in question continues execution outside of the `uTry` block.

- `uResume` - should an error condition occur, program execution transfers from the point where the exception is raised, to the exception handler [11]. Upon completion of the exception handler, program execution returns to the point where the exception is raised. In short, should an exception condition occur, control branches to the handler, and after execution of the handler, control returns to where it left off before the exception condition occurred.

As in C++ , an ellipses (...) condition indicates that the handler is executed if an exception is raised, but no appropriate exception handler is specified.

Real-time tasks (periodic, sporadic, and aperiodic tasks) may raise particular exceptions. These exceptions are:

- `uDeadlineExpiry` - A task usually has a predefined deadline. Anytime during the execution of the task body, should the deadline arrive, this exception is raised.
- `uLateStart` - A task may be required to start execution at a certain time. Should the task execute later than its specified execution time, this exception is raised.
- `uEarlyStart` - A task required to start execution at a certain time may, at times, be run earlier than its specified execution time. Should this case occur, this exception is raised.

The above exceptions may be raised and caught anywhere in the real-time task body (without the need to enclose the real-time code within a `uTry` block). If a programmer does not catch these exceptions, the default action is to terminate the program. The programmer may override the default actions by enclosing the body of the real-time task's `main` function within a `uTry` block. Should one wish to override these exceptions, it is imperative that no other code precede and supercede the `uTry` block, along with the declared handlers. Figure 6.1 illustrates this.

There are portions of code, however, that the programmer may deem as "critical" or uninterruptable, such that exceptions may not interrupt such code. This portion of uninterruptable code (uninterruptable in terms of exceptions) is labeled as *uProtected*:

```
uProtected {
    //Exceptions may not interrupt this portion of code
}
```

Should an exception condition arrive while the program is executing within the uProtected region, the execution of the exception handler is deferred until after the program exits the uProtected region.

6.6 Statement-Level Timing Constraints

Statement-level timing constraints may be specified anywhere within a program. Statement-level timing constraints are expressed within a *uBy* block, as illustrated below:

```
uBy (some_uTime_or_uDuration) {
     // ...
}
uResume (uDeadlineExpiry) {
     // ...
}
```

The `uBy` statement takes a uTime or uDuration parameter, which specifies the absolute time or length of time by which the enclosed statement needs to be executed. By the defined deadline, if the execution of the enclosed statement had not been completed, an exception is raised (section 6.5 discusses exceptions in detail, as well as the various forms of exception facilities that may be invoked).

Within a real-time task (periodic, sporadic, or aperiodic tasks), a `uBy` statement may be declared by itself, without being followed by an exception handler. The *uDeadlineExpiry* exception is automatically invoked, unless the default behaviour is overridden by an explicit declaration of an exception facility and a handler, as illustrated above.

6.7 Condition Variable Timing Constraints

In μC++ , condition variables provide a facility for waiting and synchronizing tasks. In a real-time environment, however, the time a task can wait for synchronization may need to be bounded.

Normally, a wait on a condition variable is specified by:

```
uWait ConditionVariable;
```

With timing constraints, the waiting time on a condition variable is checked to ensure that the condition variable has, indeed, been signaled by a certain time (or within a duration of time). Should this not be the case, an exception is raised. The specification of the time constraint is as follows:

```
uWait ConditionVariable by (Some_uTime_or_SomeDuration)
uCatch (uDeadlineExpiry) {
    // ...
};
```

6.8 Real-Time Programming Constraints

Real-time programming can vastly differ from programming without time constraints, in that certain operations whose execution times cannot be bounded are deemed inappropriate. The following additional constraints are listed for completeness, but not addressed in this work.

The use of dynamic data structures can lead to unbounded and unpredictable waiting times. The unpredictable nature of task arrivals and, in turn, memory requests to the operating system, coupled with the inherently sequential nature of memory management, makes memory allocation requests unpredictable (time-wise). Following the same reasoning, the dynamic creation of tasks also leads to similar unpredictable time behaviours.

Schedulability analyzers and various compile-time code analyzers are often used in real-time programs, so as to evaluate the schedulability of a program, or determine certain runtime parameters. Besides dynamic allocation of memory, the use of recursion, for instance, may also inhibit a schedulability analyzer from determining the execution time of a piece of code. Furthermore, stack-size requirements may not be accurately determinable apriori.

Certain language constructs such as a general while-loop, for example, makes a program less predictable. It has even been proposed that such constructs (that can take arbitrarily long to execute) not be made available to real-time programming languages [52]. In general, iterative computations in real-time environments must be expressible in terms of a constant-count loop.

```
uPeriodicTask sampletask {
  main() {
     // No user code allowed
     uTry {

         // Periodic Task Body
         //...

     }
     uCatch (condition1) {
         //...
     }
     uCatch (condition2) {
         //...
     }
     //...
     uResume (condition10) {
         //...
     }
     uResume (condition11) {
         //...
     }
     //...
     uCatch (uDeadlineExpiry) {
         // new default behaviour
     }
     uCatch (...) {
         //...
     }
     // No user code allowed
  }
};
```

Figure 6.1: Basic Exception Handling

Chapter 7

Real-Time Scheduler Implementation

7.1 The Notion of Priority in a Real-Time Programming Language

In adapting a general programming language to become a flexible real-time programming language, the notion of priority becomes a very handy tool, allowing for flexible implementations of various forms of scheduling paradigms [1, 12, 18]. As discussed in earlier chapters, the term *priority* has no single meaning. The priority of a task may signify its logical importance to a programmer, or may simply be a property determined by its periodic characteristics, as is the case with the rate-monotonic priority assignment scheme.

In this design and implementation of a real-time programming language, the notion of priority simply determines the order by which a set of tasks executes. As far as the real-time system is concerned, the task with the highest priority is the most eligible task to execute, and among a set of ready tasks awaiting processor time, the highest priority task is always executed first, with little regard for the possible starvation of lower priority tasks. This form of scheduling is often referred to as a *prioritized preemptive scheduling.*

In μC++ , each task's priority can be redefined and queried by the following functions, which are member routines of `uBaseTask`, from which all the different kinds of tasks inherit:

```
void uSetPriority(int priority);
void uSetActivePriority(int priority);
int uGetPriority();
int uGetActivePriority();
```

To provide the facilities for implementing various priority-changing scheduling algorithms (such as the priority-inheritance algorithm, for instance), a μC++ task can have two priorities associated with it: a base priority and an active priority. It is up to the scheduling algorithm implementor or programmer to set the appropriate priority values, or to determine whether the base priority or the active priority is the priority that will be utilized in scheduling tasks, if they are used at all.[1]

The notion of a task's priority can not only be utilized in determining which task executes next, but priorities also dictate the behaviour of various synchronization primitives such as semaphores and monitors [12]. μC++ monitors have been extended so that entry queues are prioritized.[2] The highest priority task that calls into a monitor always enters the monitor first, unless a particular entry queue is explicitly accepted, in which case, the highest priority task in the particular entry queue executes. Condition queues within a monitor are also prioritized, such

[1]Upon every task's creation, μC++ sets the base and the active priority of a task to a uniform default value, if no default priority is specified.

[2]A task's active priority is utilized by a μC++ monitor to determine a task's priority value

that the signaling of a condition queue schedules the highest priority task. Thus, both the monitor entry queues and the condition queues are prioritized, with FIFO (first-in first-out) within each priority level. With the current implementation, 32 priority levels are supported. Support for more or less priority levels can be trivially implemented by redefining the data structures in question (see section 7.2).

Prior to transforming the μC++ language implementation into a real-time "friendly" system, the monitor entry queues and the condition variable queues were scheduled using FIFO, and there was no notion of priority incorporated into the system. The addition of priorities and FIFO within the priority levels merely adds functionality, while maintaining backward compatibility to μC++ 's previous behaviour. If an application is not real-time, all the tasks are assigned an equal, default priority level. Thus, all tasks have one active priority, and FIFO scheduling dictates the system's behaviour.

7.2 Provisions for the Implementation of a Flexible Scheduler

As discussed in the introduction, one of the goals of this project is to design and implement a flexible real-time system, capable of being adapted to various real-time environments and applications. The wide availability of various forms of real-time scheduling algorithms, coupled with each algorithm's suitability for different forms of real-time applications, makes it essential that the language definition provide as few restrictions as possible on which algorithms can or cannot be utilized and implemented.

Scheduling is the mechanism by which the next task to run is chosen from a set of runnable tasks. However, this selection mechanism is closely tied to the data-structure representing the set of runnable tasks. In fact, the data-structure containing the set of runnable tasks is often designed with a particular scheduling algorithm in mind.

To achieve the goal of implementing a flexible scheduler, the ready "queue" is packaged as one entity – readily accessible and replaceable by a user. Consequently, the rules and mechanisms by which insertion and removal takes place from the ready data structure[3] is completely up to the implementor.

Four functions must be defined, which are mandatory for the ready data structure:

```
class uBaseQueueStructure {
  public:
    virtual void uAdd( TaskStructure *task ) = 0;
    virtual TaskStructure *uPop() = 0;
    virtual int uEmpty() = 0;  //is the structure empty?  True or false?
    virtual void uAddInitialize() = 0;
    virtual void uRemoveInitialize() = 0;
}
```

In turn, the "kernel" uses these four function calls to interact with the user-defined ready structure.[4]

A user can, in turn, construct any form of scheduling algorithm by modifying the behaviour of `uAdd` and `uPop`. To implement a dynamic scheduling algorithm,

[3]The term "ready queue" is no longer appropriate because the data structure may not be a queue.

[4]Operating systems such as Amoeba[54], Chorus[44], and Apertos[56] employ a similar mechanism by which the kernel can utilize external modules that are essential to the operating system.

for instance, everytime a `uAdd` or a `uPop` is performed by the kernel, an analysis of the set of runnable tasks may be performed, which may, in turn, alter the priorities of the tasks at hand. `uAddInitialize` is called by the kernel whenever a task is added to the cluster, and `uRemoveInitialize` is called by the kernel whenever a task is deleted from the cluster (section 4.4 defines *cluster*). Note that the addition (or deletion) of tasks to (or from) the cluster is not the same as adding or popping tasks from the ready structure. With a static scheduling algorithm, for example, task set analysis is only performed upon task creations, making the uAddInitialize function an ideal place to specify such analysis code. `uAdd` and `uPop` are called throughout the lifetime of the real-time tasks, but in a static scheduling algorithm, no task set analysis occurs within these functions.

A *real-time cluster* behaves just like a normal μC++ cluster, except a real-time cluster can have a special ready data-structure associated with it (the ready data-structure, in turn, has a scheduling or task dispatching policy associated with it). The ready data-structure must inherit from the *uBaseQueueStructure* class, however, and must be specified upon the creation of the real-time cluster. A real-time cluster has the following constructors:

```
class uRealTimeCluster : public uCluster {
  public:
     uRealTimeCluster( uBaseQueueStructure *Ready,
                       int size = uDefaultStackSize(),
                       const char *name = "" );
     uRealTimeCluster( uBaseQueueStructure *Ready, const char *name );
}
```

Section 7.3 discusses, in detail, the syntax for the creation of a real-time cluster.

7.3 Implementation of Deadline Monotonic Scheduling in μC++

The amalgamation of ideas discussed in the previous sections of this chapter lay the foundation for implementing many kinds of scheduling algorithms. The implementation of the *deadline monotonic* scheduling algorithm, for instance, only requires that a special ready data structure (with deadline monotonic as its task-dispatching policy) be plugged into a real-time cluster.

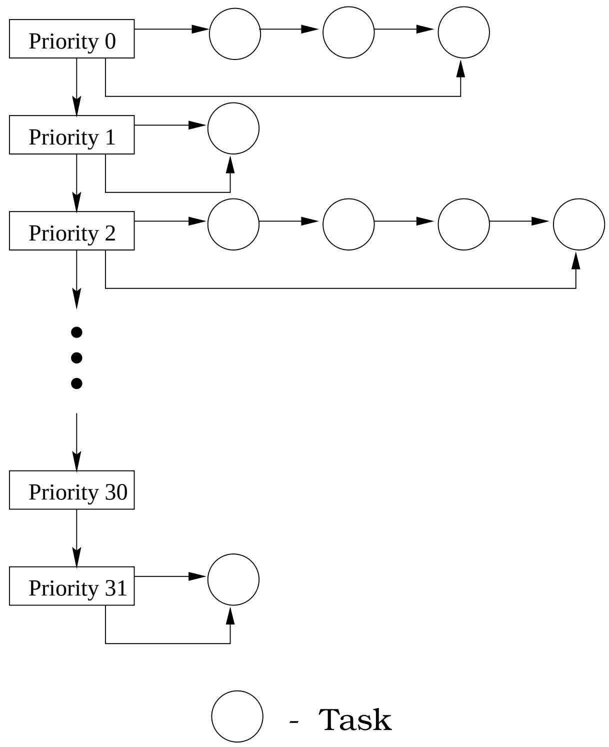

Figure 7.1: Ready-Queue Utilized in the Deadline Monotonic Implementation

The underlying ready data structure for the deadline monotonic implementation is a prioritized ready-queue, with support for 32 priority levels. The *uAdd* function adds a task to the ready-queue in a FIFO manner within a priority level. The *uPop* function returns the most eligible task with the highest priority from the ready-queue. Both *uAdd* and *uPop* utilize a constant-time algorithm for the location of the highest priority task. Figure 7.1 illustrates this prioritized ready-queue.

The `uAddInitialize` function contains the heart of the deadline monotonic algorithm. In `uAddInitialize`, each task in the ready-queue is examined, and tasks are ordered from soonest to farthest deadlines. Priorities are, in turn, assigned to every task. With the newly assigned priorities, the ready queue is re-evaluated, so as to ensure that it is in a consistent state.

On a deadline monotonic environment, tasks are not normally removed from the system. On rare occasions, should task removal be necessary, the same task set and ready queue evaluation takes place as when a new task is added. Consequently, the `uRemoveInitialize` function is identical to `uAddInitialize`.

Each time *uAddInitialize* is called, the priorities of the tasks within the cluster are manipulated. As indicated in section 7.2, this function is usually called only by the kernel. Furthermore, it is only called whenever a new task is added to the current cluster, thus requiring a re-evaluation of the task priorities.

Any real-time program wishing to utilize the deadline-monotonic algorithm must include the "uDeadlineMonotonic.h" header file in their code. A sample real-time program is illustrated in figure 7.2.

At the beginning of the `uMain::main()` routine is the creation of a real-time cluster. Note that the parameter passed to the constructor of `uRealTimeCluster` is a *uDeadlineMonotonic* class, which is a ready data-structure derived from *uBase-QueueStructure*.

```
#include<uC++.h>
#include<uDeadlineMonotonic.h>

uPeriodicTask PeriodicTask1 {
  public:
     PeriodicTask1(uDuration period, uDuration deadline,
                   uCluster &clust) : uPeriodicBaseTask(period, uTime(),
                   uTime(), deadline, clust) {};
     void main() {
          //Periodic Task Body
     } // main
};

uPeriodicTask PeriodicTask2 {
  public:
     PeriodicTask2(uDuration period, uDuration deadline,
                   uCluster &clust) : uPeriodicBaseTask(period, uTime(),
                   uTime(), deadline, clust) {};
     void main() {
          //Periodic Task Body
     } // main
};

void uMain::main() {
   uRealTimeCluster Clust(new uDeadlineMonotonic);
   uProcessor *processor;
   {
     // Note that these tasks are created, but they do not begin
     // execution till a processor is created on the "Clust" cluster.
     // This is ideal, so that "uInitialize" is run when all tasks are
     // already set-up, and no task gets to execute ahead of another
     PeriodicTask1 t1(15, 5, Clust);  //15 sec period, 5  sec deadline
     PeriodicTask2 t2(30, 10, Clust); //30 sec period, 5  sec deadline
     PeriodicTask1 t3(60, 20, Clust); //60 sec period, 20 sec deadline

     // There are no real-time tasks executing yet, since there
     // are no processors assigned to the cluster yet.

     // We now assign a processor into "Clust"
     processor = new uProcessor(Clust);
   } // wait for t1, t2, and t3 to all finish, before continuing
   delete processor;
};
```

Figure 7.2: Sample Real-Time Program

In μC++ , upon a task's creation, it is added to a cluster (either the default cluster or a user specified one), after which, the cluster's task set is analyzed, and task priorities are (re)assigned. After this priority assignment, the task is added to the ready queue, and made eligible to execute. For a static scheduling algorithm, this behaviour is sometimes unideal, in that the critical instant (see section 3.9.1 for an explanation of the critical instant) is not preserved. Upon a task's creation, it is made eligible to execute, even though its priority is not yet fully determined (as newer tasks are created, a task's priority changes – and the priority only becomes fixed and determined after all the tasks in the system have been created and analyzed). To preserve the critical instant, a possible solution (illustrated in figure 7.2) is to create a real-time cluster without initially assigning a processor to it. All the system's tasks are then created. A task is created by the current active processor on the current cluster, and the task set of the cluster where the new task is added is analyzed (and the task priorities manipulated) by the current active processor on the current cluster. After the creation of the tasks, a processor is finally assigned to the real-time cluster. This approach ensures that when the processor is put in place, the task priorities are fully determined, and the critical instant is ensured.

7.4 Testing the μC++ Real-Time Environment

To test the functionality and correctness of μC++ 's real-time features, some real-time programs were run. In particular, the implementation of the deadline monotonic algorithm was tested, as well as the implementation of the priority-inheritance

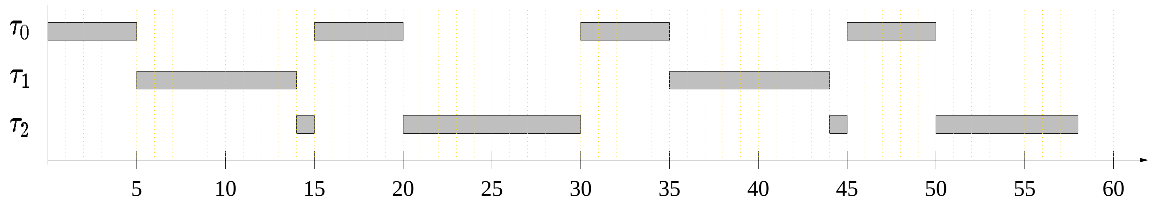

Figure 7.3: Predicted Time-Line for the Deadline Monotonic Test

protocol.

Task	Period (T)	Computation Time (C)
τ_0	15 sec	5 sec
τ_1	30 sec	9 sec
τ_2	60 sec	19 sec

Table 7.1: Task Set of the Deadline Monotonic Test Program

To test the deadline monotonic algorithm, a simple real-time program was constructed. The program creates three real-time periodic tasks, as illustrated in table 7.1. Based on the deadline monotonic algorithm, task τ_0 is given the highest priority, and task τ_2 gets the lowest priority. Schedulability tests all fail to prove that task deadlines will be met. However, carefully reasoning about the program's expected behaviour reveals that deadlines for all tasks can, indeed, be met. The reasoned or predicted time-line is illustrated in figure 7.3. When run, the program's behaviour does confirm the predicted time-line. A sample output of the program is shown in figure 7.5.

To test the functionality of the deadline monotonic, as well as the priority-inheritance protocol, the "Toilet-Going Philosophers" was implemented. In a philosopher's lifetime, a philosopher must constantly sleep, go to the toilet upon waking up, wash after going to the toilet, and go back to sleep. All philosophers must share a common restroom, which results in only one philosopher being able to go to the toilet or to wash at any one time. Philosophers, like any other person, do not wish to be interrupted while in the toilet. While washing, however, philosophers do not mind being interrupted. Philosophers must go to the toilet and wash for specified times, as indicated in table 7.2.

Task	Period (T)	Toilet Time	Washing Time
τ_0 (Philosopher 1)	20 sec	2 sec	2 sec
τ_1 (Philosopher 2)	30 sec	4 sec	3 sec
τ_2 (Philosopher 3)	40 sec	6 sec	4 sec
τ_3 (Philosopher 4)	50 sec	8 sec	5 sec

Table 7.2: Task Set of the Priority-Inheritance Test Program

Based on the deadline monotonic scheduling algorithm, task τ_0 is the highest priority task, followed by τ_1, τ_2, and τ_3 as the lowest priority task within the task set.

The predicted time-line for the behaviour of the priority-inheritance program is illustrated in figure 7.4. Note that at the 80th second since program invocation, task τ_0 becomes ready to execute. During this time, however, task τ_3 is in the toilet. Task τ_0 is not allowed to interrupt task τ_3, and graciously waits for task τ_3 to leave the toilet. In turn, task τ_3's priority is bumped up to the priority of task τ_0. As soon as task τ_3 leaves the toilet, τ_0 gets to enter the toilet. The same behaviour can be observed on the 95th second mark for tasks τ_1 and τ_2, as well as the 100th second mark for tasks τ_0 and τ_1. The program output illustrated in figure 7.6 confirms this expected behaviour.

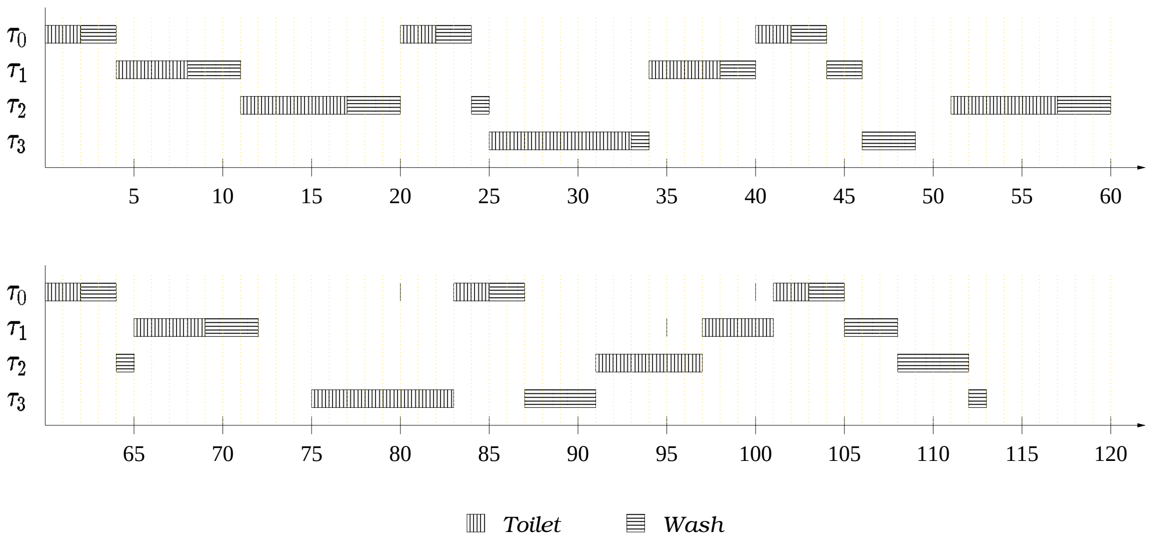

Figure 7.4: Predicted Time-Line for the Priority-Inheritance Test Program

```
Time            TaskID
------------------------------------------------------------------------
0.3354          This is Task 0 Beginning
5.112075        This is Task 0 Ending 5.108710 seconds later
5.114328        This is Task 1 Beginning
14.262883       This is Task 1 Ending 9.148541 seconds later
14.265232       This is Task 2 Beginning
15.22611        This is Task 0 Beginning
20.105209       This is Task 0 Ending 5.82588 seconds later
30.14129        This is Task 0 Beginning
35.62590        This is Task 0 Ending 5.48452 seconds later
35.114608       This is Task 1 Beginning
44.197571       This is Task 1 Ending 9.82950 seconds later
45.5562         This is Task 0 Beginning
50.64510        This is Task 0 Ending 5.58937 seconds later
57.755201       This is Task 2 Ending 43.489957 seconds later
59.997723       This is Task 0 Beginning
65.46781        This is Task 0 Ending 5.49046 seconds later
65.108272       This is Task 1 Beginning
74.193848       This is Task 1 Ending 9.85566 seconds later
74.259131       This is Task 2 Beginning
75.58465        This is Task 0 Beginning
80.115619       This is Task 0 Ending 5.57143 seconds later
90.49913        This is Task 0 Beginning
95.97519        This is Task 0 Ending 5.47595 seconds later
95.151069       This is Task 1 Beginning
104.245666      This is Task 1 Ending 9.94585 seconds later
105.41277       This is Task 0 Beginning
110.97361       This is Task 0 Ending 5.56075 seconds later
117.733971      This is Task 2 Ending 43.474826 seconds later
120.34317       This is Task 0 Beginning
125.84269       This is Task 0 Ending 5.49944 seconds later
125.143853      This is Task 1 Beginning
134.256453      This is Task 1 Ending 9.112594 seconds later
134.258466      This is Task 2 Beginning
135.34733       This is Task 0 Beginning
140.109882      This is Task 0 Ending 5.75139 seconds later
150.25800       This is Task 0 Beginning
155.110087      This is Task 0 Ending 5.84277 seconds later
155.226391      This is Task 1 Beginning
164.330029      This is Task 1 Ending 9.103628 seconds later
165.17250       This is Task 0 Beginning
170.95681       This is Task 0 Ending 5.78420 seconds later
177.877887      This is Task 2 Ending 43.619401 seconds later
...
```

Figure 7.5: Sample Program Output of the Deadline Monotonic Test

```
Time       TaskID
---------------------------------------------------------------------
0.2324    Philosopher 1 goes to TOILET (priority 1)
2.14766   Philosopher 1 leaves TOILET 2.124 sec later (priority 1)
2.17301   Philosopher 1 goes to WASH (priority 1)
4.23333   Philosopher 1 finished with WASH 2.602 sec later (priority 1)
4.26036   Philosopher 2 goes to TOILET (priority 2)
8.39601   Philosopher 2 leaves TOILET 4.135 sec later (priority 2)
8.41992   Philosopher 2 goes to WASH (priority 2)
11.4671   Philosopher 2 finished with WASH 3.471 sec later (priority 2)
11.4941   Philosopher 3 goes to TOILET (priority 3)
17.5764   Philosopher 3 leaves TOILET 6.821 sec later (priority 3)
17.6110   Philosopher 3 goes to WASH (priority 3)
20.5421   Philosopher 1 goes to TOILET (priority 1)
22.5793   Philosopher 1 leaves TOILET 2.370 sec later (priority 1)
22.6028   Philosopher 1 goes to WASH (priority 1)
24.6863   Philosopher 1 finished with WASH 2.835 sec later (priority 1)
25.9466   Philosopher 3 finished with WASH 8.335 sec later (priority 3)
25.9718   Philosopher 4 goes to TOILET (priority 4)
33.1170   Philosopher 4 leaves TOILET 8.198 sec later (priority 4)
33.1194   Philosopher 4 goes to WASH (priority 4)
34.5528   Philosopher 2 goes to TOILET (priority 2)
38.5090   Philosopher 2 leaves TOILET 3.995 sec later (priority 2)
38.5322   Philosopher 2 goes to WASH (priority 2)
40.6581   Philosopher 1 goes to TOILET (priority 1)
42.6502   Philosopher 1 leaves TOILET 1.999 sec later (priority 1)
42.6766   Philosopher 1 goes to WASH (priority 1)
44.6804   Philosopher 1 finished with WASH 2.381 sec later (priority 1)
45.6844   Philosopher 2 finished with WASH 7.152 sec later (priority 2)
49.1398   Philosopher 4 finished with WASH 16.204 sec later (priority 4)
51.4731   Philosopher 3 goes to TOILET (priority 3)
57.5066   Philosopher 3 leaves TOILET 6.334 sec later (priority 3)
57.5309   Philosopher 3 goes to WASH (priority 3)
60.1477   Philosopher 1 goes to TOILET (priority 1)
62.1691   Philosopher 1 leaves TOILET 2.213 sec later (priority 1)
62.1715   Philosopher 1 goes to WASH (priority 1)
64.1720   Philosopher 1 finished with WASH 2.545 sec later (priority 1)
64.1745   Philosopher 2 goes to TOILET (priority 2)
68.1856   Philosopher 2 leaves TOILET 4.110 sec later (priority 2)
68.1883   Philosopher 2 goes to WASH (priority 2)
71.1920   Philosopher 2 finished with WASH 3.372 sec later (priority 2)
72.1037   Philosopher 3 finished with WASH 15.506 sec later (priority 3)
75.8965   Philosopher 4 goes to TOILET (priority 4)
83.8694   Philosopher 4 leaves TOILET 7.997 sec later (priority 1)
83.8935   Philosopher 4 goes to WASH (priority 4)
83.9979   Philosopher 1 goes to TOILET (priority 1)
85.1116   Philosopher 1 leaves TOILET 2.118 sec later (priority 1)
85.1139   Philosopher 1 goes to WASH (priority 1)
87.1175   Philosopher 1 finished with WASH 2.352 sec later (priority 1)
91.1004   Philosopher 3 goes to TOILET (priority 3)
97.1026   Philosopher 3 leaves TOILET 6.221 sec later (priority 2)
97.1052   Philosopher 3 goes to WASH (priority 3)
97.2010   Philosopher 2 goes to TOILET (priority 2)
101.199   Philosopher 2 leaves TOILET 3.998 sec later (priority 1)
101.202   Philosopher 2 goes to WASH (priority 2)
101.211   Philosopher 1 goes to TOILET (priority 1)
103.214   Philosopher 1 leaves TOILET 2.319 sec later (priority 1)
103.216   Philosopher 1 goes to WASH (priority 1)
105.220   Philosopher 1 finished with WASH 2.314 sec later (priority 1)
108.222   Philosopher 2 finished with WASH 7.199 sec later (priority 2)
112.130   Philosopher 3 finished with WASH 15.252 sec later (priority 3)
113.144   Philosopher 4 finished with WASH 30.550 sec later (priority 4)
120.193   Philosopher 1 goes to TOILET (priority 1)
122.196   Philosopher 1 leaves TOILET 2.239 sec later (priority 1)
122.198   Philosopher 1 goes to WASH (priority 1)
124.197   Philosopher 1 finished with WASH 1.999 sec later (priority 1)
...
```

Figure 7.6: Sample Program Output of the Priority-Inheritance Test

Chapter 8

Future Work

Though the current implementation eliminates most of the tedious chores involved in implementation of real-time applications (such as the automated calculation of priorities for certain scheduling algorithms), the current environment still falls short, in that a form of offline schedulability analyzer has yet to be implemented. Though various research results on schedulability analysis have been presented in this thesis, automating such calculations have yet to be implemented. Implementation of schedulability analyzers are discussed in [53]. Besides a schedulability analyzer that determines the feasibility and schedulability of a task set, it is also desirable to have an analyzer which evaluates the average and worst case execution times of a program.

The current implementation does not take advantage of multiprocessor architectures, partly due to the scheduling difficulty associated with such environments. Despite this difficulty, however, restrictive environments can be constructed which could utilize the multiprocessor environment. In addition, with the use of heuristics, multiprocessor environments can still be suitable for soft real-time applications.

No programming language support has yet been defined to provide dynamic change management, which is important in "on-going" systems or systems with long and uninterruptable operations. Software upgrade and maintenance on these systems are often extremely difficult, and research on explicit programming language support for such features are scarce.

Future real-time systems are going to be more intelligent, adaptive, complex, and distributed. As such, broad areas of research have and are evolving in areas of process to processor allocations, bounding the end-to-end communication times between (distributed) resources, and integrating dynamic scheduling (which is sometimes necessary for very dynamic and adaptive systems) into static scheduling.

Chapter 9

Conclusion

In the work embodied in this project, the concurrent, object-oriented programming language μC++ was extended, making it a more suitable platform for real-time development. Changes to both the language and its implementation had to be undertaken to accomplish this goal.

Before anything, the notion of "time" was examined. The effective expression of time, along with its convenient manipulation, is an extremely essential characteristic of a real-time language. To this end, the notion of a *clock, time,* and *duration* were introduced, which allowed for the specification of various forms of "time", differentiating durations of time from absolute time values.

The introduction of a few constructs that allowed for some form of time-constraint specification was imperative. The effective specification of time constraints is a necessity for any real-time language, for it is only with the ability to do so, that the system can deterministically and concretely dictate the behaviour of the system. Constructs were introduced that allowed for the specification of various temporal scopes, which took the form of absolute deadlines, relative deadlines, and delays.

Furthermore, exception facilities were discussed, which allowed for orderly means for handling missed deadlines and other errors.

The notion of priority was introduced to tasks within the programming language. Each task's priority, in turn, dictated its scheduling behaviour. Along with the normal task dispatching scheduling policy, the notion of each task's priority dictated the behaviour of various synchronization primitives such as semaphores and monitors. Queues for monitors and condition variables, in particular, were made to be "priority-aware".

A flexible means of implementing various scheduling algorithms was possible by making the ready data-structure into a user replaceable module. The task-dispatching behaviour of the ready data-structure, in turn, dictates which task executes next. To illustrate this flexibility, the deadline monotonic scheduling algorithm was implemented.

Finally, the priority inheritance algorithm was extended and implemented. The original algorithm did not address monitor issues such as condition variables. The implemented version of the priority inheritance protocol took such issues into consideration, and upper bounds for the modified version of this algorithm were discussed.

Bibliography

[1] B. Adelberg, H. Garcia-Molina, and B. Kao. Emulating soft real-time scheduling using traditional operating system schedulers. In *Proc. IEEE Real-Time Systems Symposium*, pages 292–298, 1994.

[2] S.T. Allworth. *Introduction to Real-Time Software Design*. Macmillan Press, 1984.

[3] N.C. Audsley, A. Burns, M.F. Richardson, and A.J Wellings. Deadline monotonic scheduling theory. Technical report, University of York, 1991.

[4] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. Hard real-time scheduling: The deadline-monotonic approach. In *Real-Time Programming – IFAC Workshop*, pages 127–132, June 1991.

[5] Neil Audsley, Alan Burns, Robert Davis, Ken Tindell, and Andy Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems*, 8:173–198, 1995.

[6] Neil C. Audsley. Deadline monotonic scheduling. Technical report, University of York, 1990.

[7] T. Baker and O. Pazy. Real-time features of Ada 9x. In *Proc. IEEE Real-Time Systems Symposium*, pages 172–180, 1991.

[8] Brian N. Bershad. Mutual exclusion for uniprocessors. Technical report, Carnegie Mellon University, April 1991.

[9] Peter Buhr and Richard Stroobosscher. *μC++ Annotated Reference Manual*, 4.5 edition, 1996.

[10] Peter A. Buhr, Michel Fortier, and Michael H. Coffin. Monitor classification. *ACM Computing Surveys*, 27(1):63–107, March 1995.

[11] Peter A. Buhr, Hamish I. Macdonald, and C. Robert Zarnke. Synchronous and asynchronous handling of abnormal events in the μsystem. *Software - Practice and Experience*, 22(9):735–776, September 1992.

[12] Alan Burns and A. J. Wellings. The notion of priority in real-time programming languages. *Computer Language*, 15(3):153–162, 1990.

[13] S.C. Cheng. Scheduling algorithms for hard real-time systems: A brief survey. In John A. Stankovic and Krithi Ramamritham, editors, *Tutorial: Hard Real-Time Systems*, chapter 5, pages 150–173. IEEE Computer Society Press, 1988.

[14] Tai M. Chung and Hank G. Dietz. Language constructs and transformation for hard real-time systems. In *Proc. Second ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, June 1995.

[15] Richard W. Conway, William L. Maxwell, and Louis W. Miller. *Theory of Scheduling*. Addison-Wesley Publishing, 1967.

[16] N. Gehani and K. Ramamritham. Real-time Concurrent C: A language for programming dynamic real-time systems. *Real-Time Systems*, 3(4):377–405, December 1991.

[17] T.M. Ghazalie and T.P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, 9:31–67, 1995.

[18] David B. Golub. Operating system support for coexistence of real-time and conventional scheduling. Technical report, Carnegie Mellon University, November 1994.

[19] John B. Goodenough and Lui Sha. The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks. *Ada Letters*, VIII(7):20–31, 1988.

[20] W.A. Halang and K. Mangold. Real-time programming languages. In Michael Schiebe and Saskia Pferrer, editors, *Real-Time Systems Engineering and Applications*, chapter 4, pages 141–200. Kluwer Academic Publishers, 1992.

[21] H. Hecht and M. Hecht. Software reliability in the systems context. *IEEE Transactions on Software Engineering*, 12(1):51–58, 1986.

[22] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[23] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[24] B. Hoogeboom and W.A. Halang. The concept of time in the specification of real-time systems. In Krishna M. Kavi, editor, *Real-Time Systems: Abstractions, Languages, and Design Methodologies*, chapter 1, pages 19–38. IEEE Computer Society Press, 1992.

[25] International Standard ISO/IEC. *Ada Reference Manual*, 6.0 edition, 1995.

[26] Y. Ishikawa, H. Tokuda, and C.W. Mercer. Object-oriented real-time language design: Constructs for timing constraints. In *Proc. ECOOP/OOPSLA*, pages 289–298, October 1990.

[27] K.B. Kenny and K.J.Lin. Building flexible real-time systems using the Flex language. *IEEE Computer*, 24(5):70–78, May 1991.

[28] E. Klingerman and A.D. Stoyenko. Real-time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, pages 941–949, September 1986.

[29] B.W. Lampson and D.D. Redell. Experiences with processes and monitors in Mesa. *Communications of the ACM*, pages 105–117, February 1980.

[30] John P. Lehoczky, Lui Sha, and Y. Ding. Rate-monotonic scheduling algorithm: Exact characterization and average case behaviour. In *Proc. IEEE Real-Time Systems Symposium*, pages 166–171, 1989.

[31] John P. Lehoczky, Lui Sha, and Jay K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proc. IEEE Real-Time Systems Symposium*, pages 261–270, 1987.

[32] John P. Lehoczky, Lui Sha, J.K. Strosnider, and Hide Tokuda. Fixed priority scheduling theory for hard real-time systems. In André M. van Tilborg and Gary M. Koob, editors, *Foundations of Real-Time Computing: Scheduling and Resource Management*, chapter 1, pages 1–30. Kluwer Academic Publishers, 1991.

[33] J.Y.T. Leung and M.L. Merrill. A note on preemptive scheduling of periodic real-time tasks. *Information Processing Letters*, 11(3):115–118, 1980.

[34] J.Y.T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2:237–250, 1982.

[35] K.J. Lin and S. Natarajan. Expressing and maintaining timing constratins in FLEX. In *Proc. IEEE Real-Time Systems Symposium*, pages 96–105, 1988.

[36] C. L. Liu. Fundamentals of real-time scheduling. In Wolfgang A. Halang, editor, *Real-Time Computing*, pages 1–7. Springer-Verlag Publishers, 1994.

[37] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery*, pages 46–61, February 1973.

[38] T. Martin. Real-time programming language PEARL – concept and characteristics. In *IEEE Computer Society 2nd International Computer Software and Applications Conference*, pages 301–306, 1978.

[39] Clifford W. Mercer. An introduction to real-time operating systems: Scheduling theory. Dept. of Computer Science, Carnegie Mellon University, November 1992.

[40] David Mosberger, Peter Druschel, and Larry L. Peterson. A fast and general software solution to mutual exclusion on uniprocessors. Technical report, University of Arizona, June 1994.

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

[42] Ragunathan Rajkumar, Lui Sha, and John P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proc. IEEE Real-Time Systems Symposium*, pages 259–269, 1988.

[43] David Ripps. *An Implementaion Guide to Real-Time Programming*. Yourdon Press, 1990.

[44] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Hermann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Chorus distributed operating systems. *Computing Systems*, 1, 1988.

[45] A.E.K. Sahraoui and D. Delfieu. ZAMAN, a simple language for expressing timing constraints. In *Real-Time Programming, IFAC Workshop*, pages 19–24, 1992.

[46] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[47] Lui Sha, Mark H. Klein, and John Goodenough. Rate monotonic analysis for real-time systems. In André M. van Tilborg and Gary M. Koob, editors, *Foundations of Real-Time Computing: Scheduling and Resource Management*, chapter 5, pages 129–156. Kluwer Academic Publishers, 1991.

[48] Alan Shaw. Software clocks, concurrent programming, and slice-based scheduling. In *Proc. IEEE Real-Time Systems Symposium*, pages 14–18, 1986.

[49] Wei Kuan Shih, J.W.S. Liu, and C.L. Liu. Modified rate-monotonic algorithm for scheduling periodic jobs with deferred deadlines. *IEEE Transactions on Software Engineering*, 19(12):1171–1179, January 1994.

[50] Brinkley Sprint, Lui Sha, and John Lehoczky. Aperiodic task scheduling for hard real-time systems. *Journal of Real-Time Systems*, 1:27–60, 1989.

[51] J.A. Stankovic. Real-time computing systems: The next generation. In John A. Stankovic and Krithi Ramamritham, editors, *Tutorial: Hard Real-Time Systems*, chapter 2, pages 14–37. IEEE Computer Society Press, 1988.

[52] A.D. Stoyenko. The evolution and state-of-the-art of real-time languages. *Journal of Systems and Software*, pages 61–84, April 1992.

[53] A.D. Stoyenko. Language-independent schedulability analysis of real-time programs. In Michael Schiebe and Saskia Pferrer, editors, *Real-Time Systems*

Engineering and Applications, chapter 2, pages 41–92. Kluwer Academic Publishers, 1992.

[54] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33, December 1990.

[55] H. Tokuda, J.W. Wendorf, and H.Y.Wang. Implementation of a time-driven scheduler for real-time operating systems. In *Proc. IEEE Real-Time Systems Symposium*, December 1987.

[56] Yasuhiko Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proc. Object-Oriented Programming Systems, Languages, and Applications*, pages 414–434, 1992.