Chapter 2

Background and Motivation

This chapter discusses the requirements of real-time and multimedia applications and describes some of the problems system designers encounter in attempting to support such applications. The case is made for an operating system resource reservation approach to the problem.

2.1 Real-time and multimedia application requirements

Real-time applications require not only that computations result in logically correct answers, but that the answers are available within certain timing constraints. A logically correct answer that arrives late is considered incorrect in a real-time system [114]. Many multimedia applications have this property that late computations are useless. For example, if a video frame or audio sample arrives after the time at which it was to be displayed or played, it is no longer useful.

This section gives an overview of different kinds of timing constraints and criticality characteristics of real-time and multimedia applications. The models described here represent a compendium of the models that researchers have addressed in the literature.

2.1.1 Timing characteristics

In general, a task in a real-time system has timing constraints that specify when the computation may begin, how long it executes, and the deadline for the completion. Figure 2-1 illustrates a computation schematically. This generic model is common in the operations research literature [20]. The computation has a ready time, r, at which the computation becomes available for scheduling. At some point after the ready time, the computation will be scheduled and start processing for a total duration of C. The duration between the ready time and the start of processing is enclosed in a white box. This indicates that the task is available for scheduling but has not started yet. The black box represents the computation that completes at time E. A deadline, d, may be associated with the computation as well, and the goal is to complete the computation before the deadline. In Figure 2-1, a thick vertical line represents the deadline.

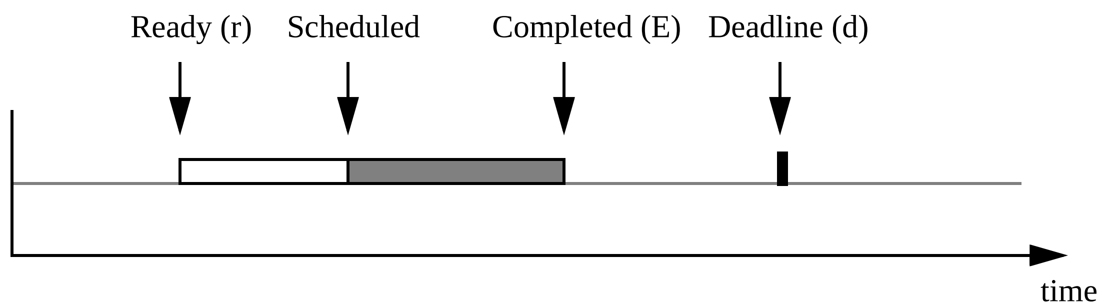

Figure 2-1: Schematic of a Time Constrained Computation

The ready time of a computation may arise from a clock event, an external interrupt, or a software event generated by some other computation. The ready event may be an instance of a periodic computation where the same computation is activated periodically. The ready event may be aperiodic but predictable, or it may be unpredictable. The computation time may be fixed or it may be variable or unpredictable. The computation itself may be preemptible, or it may form a non-preemptible critical region. The deadline is usually some fixed duration after the ready time, but the implications of missing a deadline may vary. Hard real-time computations take the deadline to be a *hard* deadline where the computation must be complete by the deadline time. Alternatively, the deadline may just be a recommendation or preference for completion of the computation, a *soft* deadline.

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

A periodic task has ready times for the task instantiations separated by a fixed period. Periodic tasks are the main focus of the original rate monotonic scheduling work [67] and the many extensions that have followed [63,108]. Figure 2-2 shows a periodic task. The figure shows four instantiations, each with an associated ready time, r_i. The ready times are separated by the period t. The computation time is represented by the black box, and the preceding white box represents the time when the task is ready by has not yet been scheduled to execute. In this example, the computation time is constant across task instantiations, and the deadline is at the end of the period.

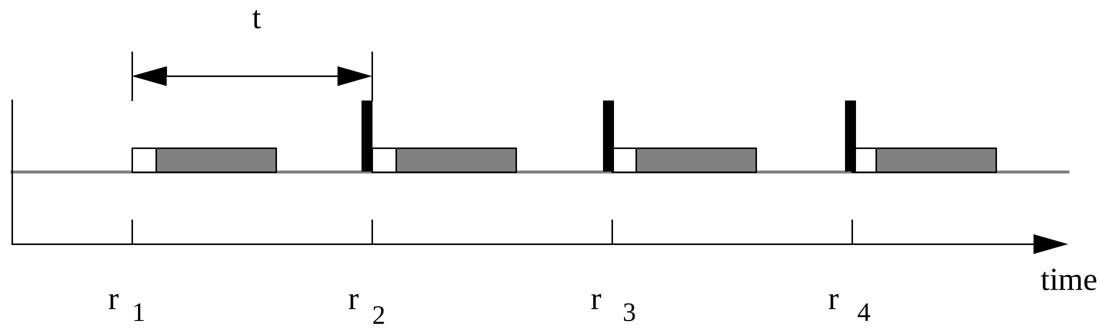

Figure 2-2: Periodic Task

Aperiodic tasks are more difficult to specify. Some aperiodic tasks are predictable to a certain extent; it may be possible to predict the arrival of instantiations of an aperiodic task within some scheduling horizon of h time units. Figure 2-3 shows an aperiodic task with a scheduling horizon of duration h from the current time. This kind of timing requirement is used in computer music, for example [5,25]. Within this window of h time units, the ready times of instantiations of the task are known, but beyond the horizon, nothing is known of the behavior. The computation time is assumed to be constant across instantiations in the single task, and the deadlines are left unspecified.

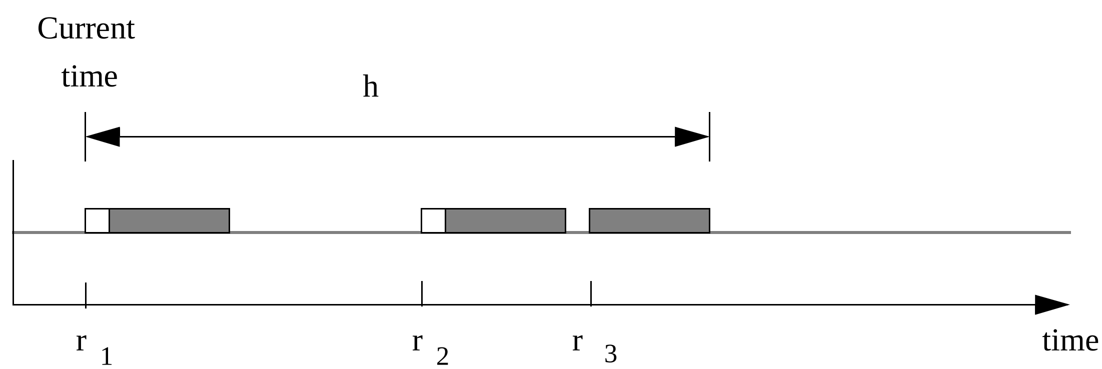

Figure 2-3: Aperiodic, Predictable Task

Another class of aperiodic tasks is almost completely unpredictable. It is common, however, to associate a minimum interarrival time for the instantiations of these unpredictable aperiodic tasks. Much work has been done on scheduling aperiodic tasks with soft deadlines [120] and on aperiodic tasks with hard deadlines, which are known as *sporadic* tasks [52,111]. Figure 2-4 illustrates an aperiodic task where the arrivals are unpredictable.

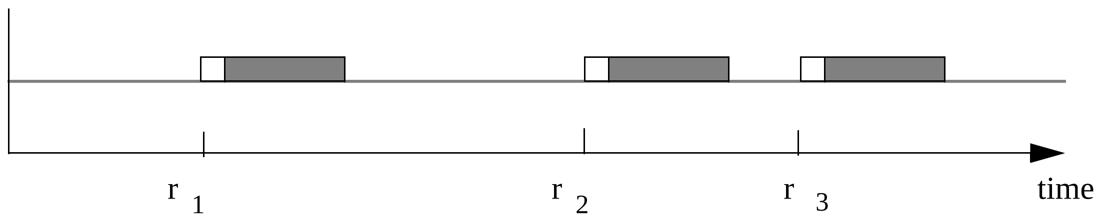

Figure 2-4: Aperiodic, Unpredictable Task

The computation time is another dimension along which tasks may vary. The computation time may be fixed or merely bounded in duration. The computation could also be described by a statistical distribution, but that case is much harder to analyze.

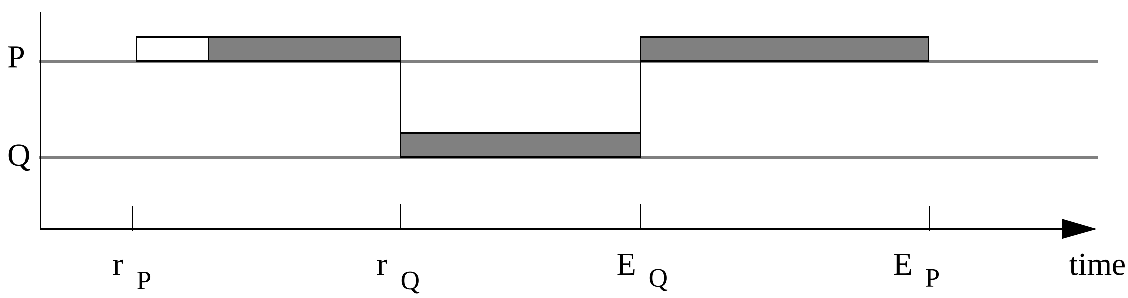

Figure 2-5: Preemptible Task

Another characteristic of the computation is its preemptibility. It may be completely preemptible (that is preemptible at any point) or it may be non-preemptible. Or it may be preemptible but with one or more non-preemptible critical regions during which scheduling events are not allowed (possibly during system calls for example). Different assumptions are made to achieve different results [67,72,82], and in particular, much work has been done on handling non-preemptible critical regions [8,16,51,97,108]. Figure 2-5 shows an example of a preemptible task, P, and its interaction with another task, Q. For this example, we assume that P is preemptible and has a lower priority than Q. P becomes ready at time r_P and begins to execute immediately. At time r_Q, Q becomes ready, and since Q has priority over P, Q preempts the ongoing execution of P. After Q completes, the execution of P resumes.

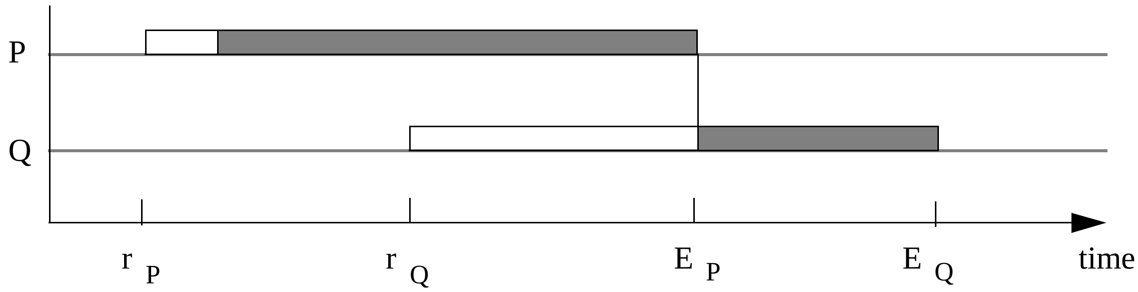

Figure 2-6: Non-preemptible Task

Figure 2-6 illustrates a similar case where the computation of P is non-preemptible and where Q has priority over P. P becomes ready at time r_P and begins to execute. Q becomes ready at time r_Q, but even though Q has priority over P, P cannot be preempted, and Q must wait until the execution of P completes. After P is finished, Q can begin execution.

2.1.2 Criticality

Deadlines may be classified as hard or soft. We can describe various types of deadlines by means of a value function. Value functions have been used for scheduling [15,55], but here they are used for purposes of exposition. A value function is a function of time that indicates the value that completion of the task would contribute to the overall value of the system. For example, Figure 2-7 shows the value function of a task that has a hard deadline; the value drops off to negative infinity at $t = d$. The task becomes ready at time *r*, and its deadline is *d*. If the task is completed at time *t* where $r \le t \le d$, then the system receives some value, *V*. On the other hand, if the task completes after *d*, the value is negative infinity, a catastrophic failure.

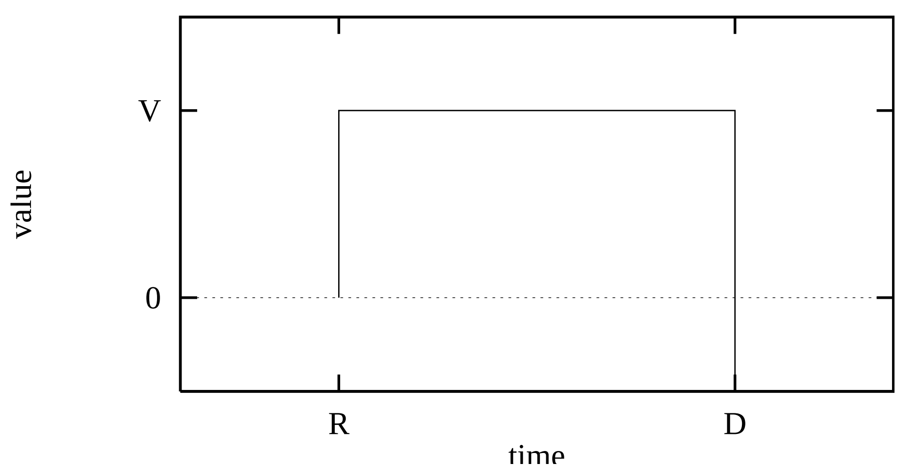

Figure 2-7: Hard (Catastrophic) Deadline Value Function

The result of missing a hard deadline may not be catastrophic. Figure 2-8 shows a case where completion of a task would have some value until the deadline *d* when the value of completion of the task goes to zero. This indicates that the system will receive no benefit from completing the computation after *d*, and so the task should be aborted, freeing any resources it holds. In contrast to the previous case, the system can continue to operate, achieving a positive value even if this particular task is aborted and makes no contribution to that value.

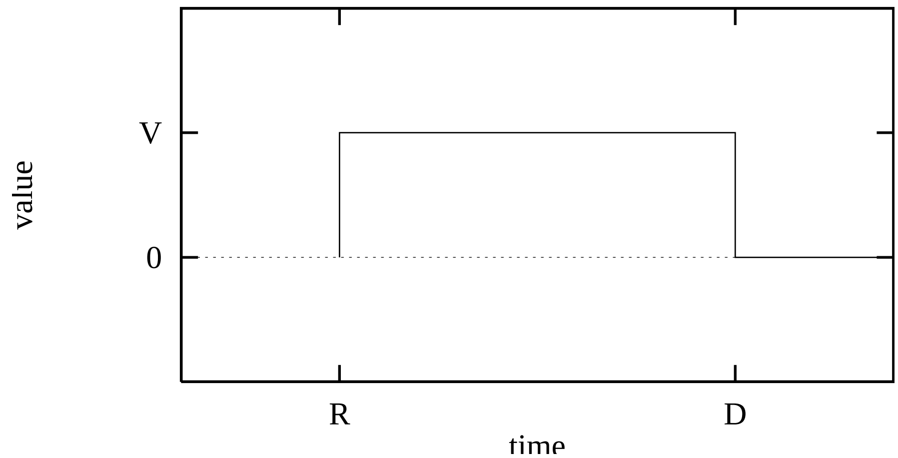

Figure 2-8: Hard Deadline Value Function

Other variations on the idea of hard deadline might include a value function that ramps up to the deadline as illustrated in Figure 2-9. And depending on where the ramp starts, this type of value function can specify tasks that must be executed within very narrow intervals of time.

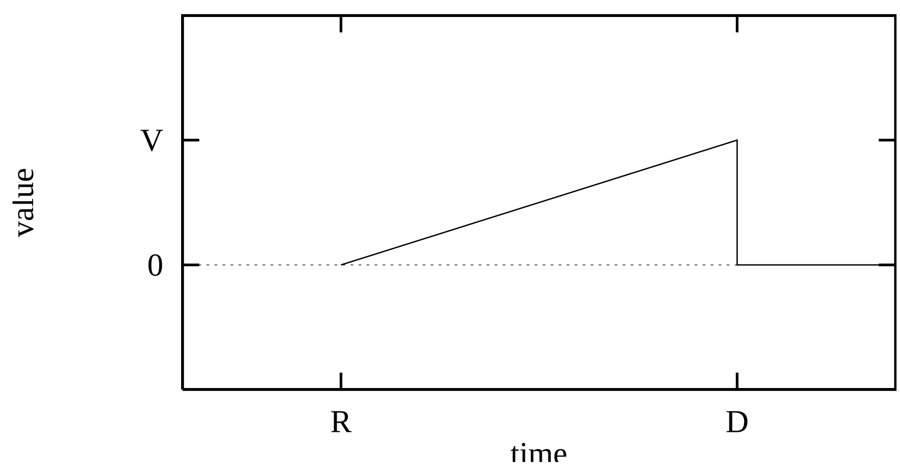

Figure 2-9: Ramped Hard Deadline Value Function

The concept of a soft deadline is illustrated in Figure 2-10 where the value function goes gradually to zero after the deadline. In this case, there is some value to the system in completing the task after the deadline, and the task should not be aborted right away as in the case of the hard deadline.

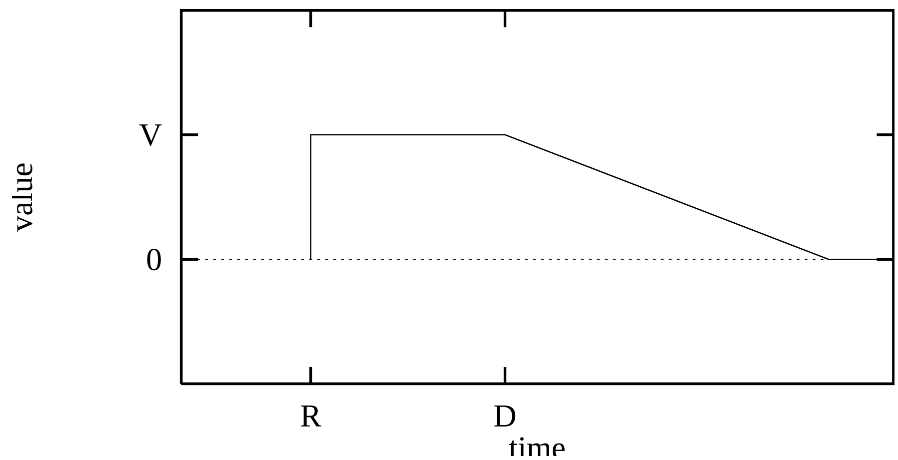

Figure 2-10: Soft Deadline Value Function

A non-real-time task might be described by the value function shown in Figure 2-11. In this case, completion of the task always has a positive value associated with it. This indicates no explicit timing constraint, although in practice, few of us are willing to wait indefinitely for a computation to complete.

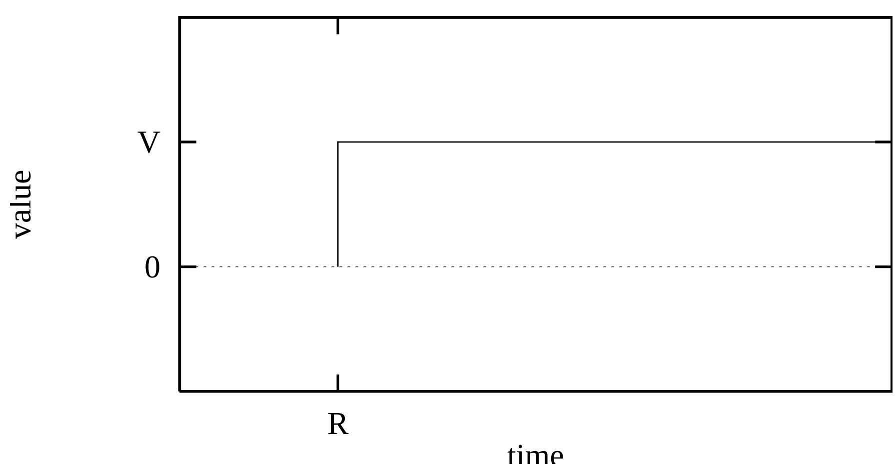

Figure 2-11: Non-real-time Value Function

2.2 Applications timing requirements

Many multimedia applications have real-time constraints. Even simple playback applications have real-time constraints that must be satisfied. For example, an audio player application might repeatedly read audio data from a file on disk and then enqueue the data for the audio device. Figure 2-12 illustrates the periodic computational requirements of such a playback application. The activity of the player is illustrated on the line labeled “P”, and the activity of the device is represented on the line labeled “D”. In each period, the applications reads, processes, and enqueues the data to a device. At the end of each period, the device reads the data out of its buffer, performs D/A conversion, and the analog signals goes to a speaker.

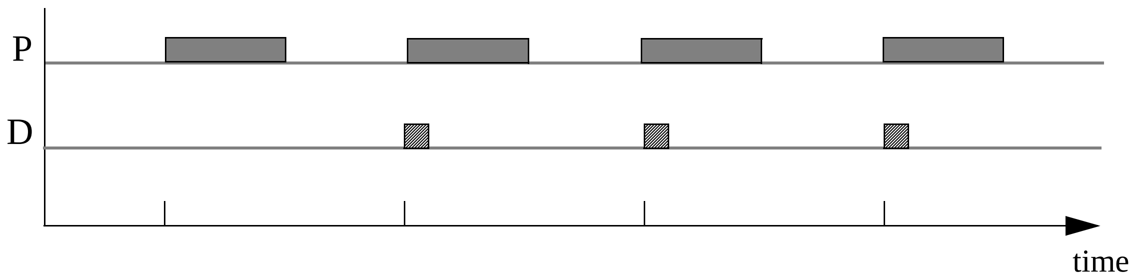

Figure 2-12: Periodic Playback Computations

A potential problem is that the computation of the audio player may be delayed so much that the device buffer empties and there are no samples for the device to convert to an analog signal. For a playback application, one solution is to introduce a large buffer and allow the playback application to execute for many periods to build up a large buffer of data ready to be played by the device.

Figure 2-13 shows a player with execution history shown on the line labeled “P”. The player buffers a number of data blocks for a device; the size of the data is indicated in the area labeled “B” between P and D. The device consumes data from the buffer, and the activity of the device is indicated on the line labeled “D”. Even if the audio player is delayed for a period or two, there will still be plenty of data in the buffer and the device will not run out of data blocks. The player can catch up with the processing that was delayed.

Another potential problem is that if some other activity on the machine is very active and manages to deprive the audio player of the resources (like processor time) for a very long time, then audio playback will be noticeably disturbed. This kind of intense competing activity can be avoided, and with large buffers audio playback will be quite smooth.

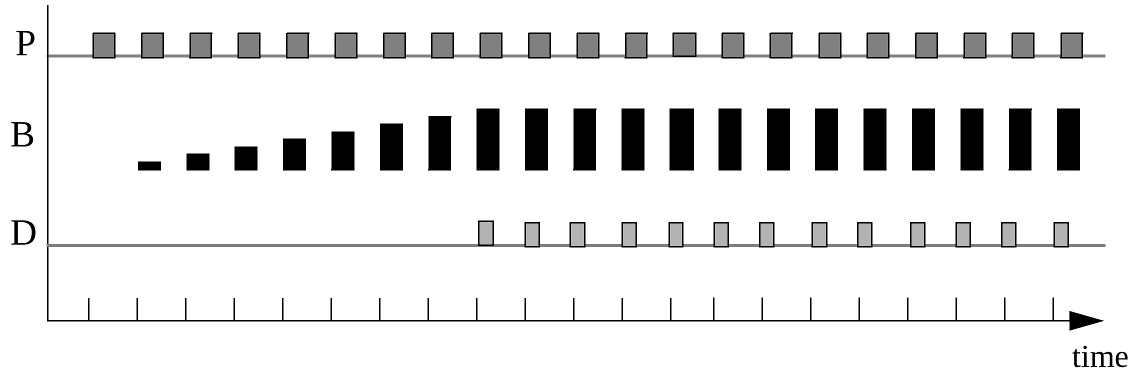

Figure 2-13: Playback Application Computing Ahead

Interactive applications cannot afford to use large buffers to smooth variations in scheduling delay. The delay introduced by large buffers is often too great to satisfy stringent end-to-end delay bounds in interactive applications like video teleconferencing. Instead, the systems must support applications that can ensure that bounds on the scheduling delay are observed. Thus the variations can be reduced and less buffering is required.

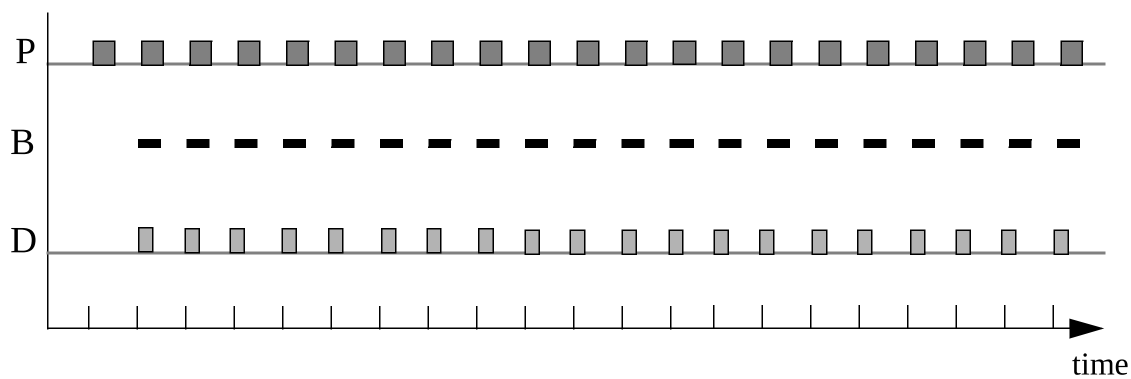

Figure 2-14: Interactive Application with Limited Workahead

Figure 2-14 shows the case where an interactive application can buffer the data generated in one period, but since there is a delay constraint, the application cannot afford to compute ahead several periods as in the previous case. The buffer must remain small. In order to make sure that the buffer is never empty when the device goes to get the next block of data, the application must make sure that the computations to enqueue the next block of data are done in time. This means it must also make sure that resources it will need from the system are available in time as well.

Other multimedia applications have timing constraints that differ from the periodic timing of the stream-oriented applications described thus far. For example, computer music applications involve performing computations to generate musical notes and other musical events. The number of notes that need to be generated at any point in time may vary widely depending on what the music calls for, so there is little periodic structure to describe the computations to be executed.

Silence-suppressed audio presents a similar problem. For audio conversations, it is not always necessary to transmit data for the silent portions of the audio streams in a two-way conversation. Again, the computational requirements become aperiodic when the amount and timing of data depend on speech patterns.

There are other applications, such as compressed video players, where a periodic execution pattern exists but where the computation time required within the periods varies. Compressed video data contains frames whose size varies according to the compression algorithm and the characteristics of the scene and how fast scenes are changing in the video source.

2.3 Quality of service management

Many multimedia applications have timing requirements and other quality of service (QOS) parameters that represent the user's desires and expectations for the performance of the applications. The complexity of providing for these timing requirements at the system level is exacerbated by the fact that the user may change those timing requirements at any time during the execution of the applications; and of course the user may create and terminate multimedia applications at any time.

2.3.1 QOS background

In recent years, researchers in the computer networking and in the telecommunications communities have been working on ways to express the QOS requirements for multimedia applications. Some of this work dealt with human perception requirements for various media [30,117], and other work focused especially on parameters and QOS architectures that are important in the context of scheduling traffic on a network [14,88]. This work can be considered an extension of the earlier work done in the telephone companies to characterize quality of service for telephone customers [100].

As researchers gained more experience with the idea of building networks that could provide quality of service levels suitable for different types of multimedia traffic, the question of how to ensure quality of service levels for end-to-end applications arose. Achieving that goal means QOS requirements must be supported in the operating system as well as the network. This observation was an initial motivation for the work described in this dissertation [76], and other system designers have started to focus on this aspect as well [3,21,46,53,83,102,126].

2.3.2 Mapping QOS parameters

In dealing with QOS management it is important to realize that there are different types of QOS parameters for different levels of the system. Applications must interact with the user in terms of user-level QOS parameters. For video, these user-level parameters might include frame resolution (width and height of each frame), number of bits/pixel, frames per second, maximum delay, and maximum jitter. For audio the user-level parameters might include sample rate, sample size, maximum delay, and maximum jitter. These are the types of parameters that might be meaningful to the user of a multimedia application. Or it might be preferable to offer QOS levels with names like: “worst”, “fair”, “good”, “better”, and “best” to simplify the interface. It would then be up to the application to translate these abstract QOS specifications to frame rates and sample rates.

Once the user-level QOS parameters are determined, they have to be mapped into system-level QOS parameters that would be meaningful for system-level resource management mechanisms. These system-level QOS parameters would describe how much time is needed on various resources. They depend on the user-level QOS parameters and on detailed computations that the application performs on data elements in the media stream.

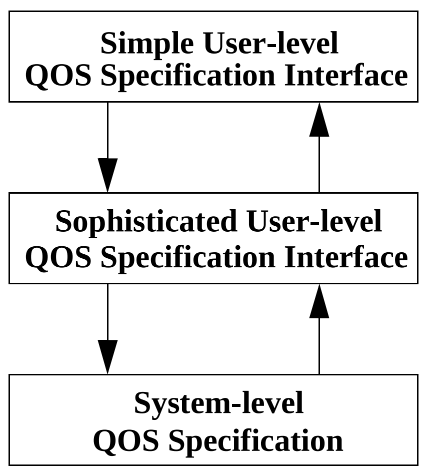

Figure 2-15: Levels of QOS Specification

Figure 2-15 summarizes these levels of QOS specification. The arrows in the figure indicate that there are mappings from one level to the next lower level and also that there are inverse mappings that come into play as well.

To allow the user to specify the QOS parameters desired at the highest level, the application must be able to map from user-level QOS parameters to system-level QOS parameters. The system-level parameters are required for the application to be able to ask for the resources it will need to execute. If the resources are unavailable, the system-level resource management mechanism should be able to communicate the fact that those parameters cannot be guaranteed. It should then initiate a negotiation to arrive at a set of system-level parameters that can be supported by the system. The inverse mapping to user-level QOS parameters should yield a QOS specification that can be tolerated by the user. Thus, the inverse mapping from system-level to user-level QOS parameters is just as important as the forward mapping.

2.3.3 QOS negotiation

The user's QOS requirements may sometimes conflict with resource usage limitations, and therefore the QOS layer may need to negotiate user requirements to resolve such conflicts. This negotiation process may be needed throughout the lifetime of certain applications since user-level QOS requirements may change over the course of execution.

The approach advocated in this dissertation for handling the complexity of a dynamic execution environment (where programs may have changing real-time requirements) is to divide the problem into two parts. One part is the dynamic negotiation of resource allocation. The second part is the resource reservation and scheduling based on the allocation. These two parts can be addressed with a layering of functionality in the system where a system-level QOS management layer handles the resource allocation policy decisions and a low-level operating system resource reservation mechanism handles the details of dynamic scheduling and usage enforcement.

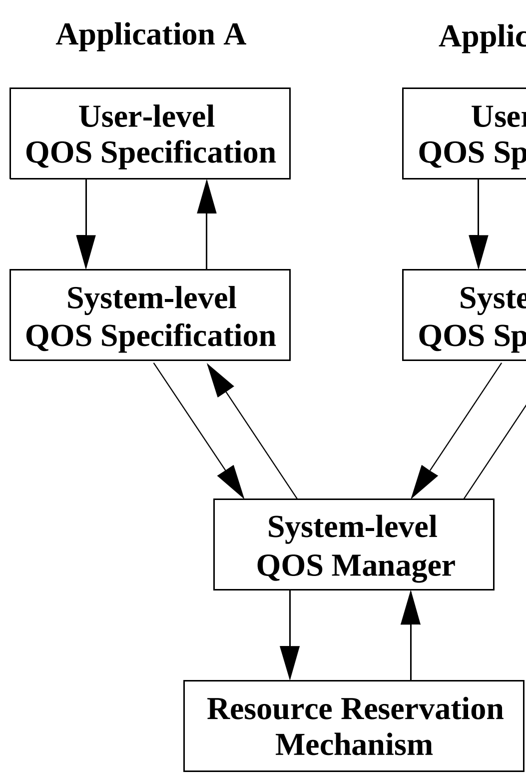

Figure 2-16: QOS Levels with QOS Manager

An illustration of the basic outline of the QOS management system appears in Figure 2-16. Each application has user-level QOS specifications. The applications map the user-level QOS parameters in to system-level QOS parameters and then negotiate with the system-level QOS manager to determine a mutually acceptable set of system-level parameters. The operating system contains a resource reservation mechanism which is used by the system-level QOS manager to actually allocate the resource capacity, schedule appropriately, and enforce the resource reservations.

The QOS management layer makes policy decisions about where resource capacity in the system should be allocated. To do this, it will depend on input from applications as they make their system-level QOS parameters known. The QOS manager may take input from user preferences expressed in the form of rules about which applications are more important than other applications, and it may take input from user interface tools designed to help the user manage resource allocation in the system. The QOS manager may also coordinate with other QOS managers on remote hosts for setting up distributed multimedia applications that require resources on several different hosts.

2.4 System design approaches

Several approaches have been used in the past to support interactive and playback types of multimedia applications. These range from hoping for the best, to dedicating expensive resources, to system support for real-time programming.

2.4.1 Specialized hardware

One approach to supporting real-time multimedia applications is to dedicate hardware to the tasks that must be performed in real-time. This relieves any contention for resources. As an example, Pandora's Box [44] was an early attempt to support multimedia applications in the context of a desktop workstation. The box contained six transputers, each one dedicated to a particular activity or class of activity including audio processing, storage/disk management, video processing, decompression, network communication, and bus management. This box was connected to a Sun workstation, to a network, and to a monitor. It coordinated graphical display from the workstation and video streams from the network or other devices, combining them and displaying the result on the monitor. The system allowed researchers to learn much about programming multimedia applications, what kinds of applications where useful, and user interface issues. However, the cost of the box was very high, and the complexity of programming the box itself was also great.

2.4.2 Time-sharing systems

A number of multimedia applications are available for personal computers and desktop workstations that run more sophisticated operating systems like UNIX and Windows NT. These systems use time-sharing scheduling policies that are not particularly well suited for meeting the timing constraints of multimedia applications.

Time-sharing schedulers are tuned to provide fair allocation of resources among users which are considered equally important. These schedulers also look at whether a processes is compute-bound or not, and they depress the priority of compute-bound processes in favor of interactive (or I/O-bound) applications which can benefit from better response times.

This kind of scheduling behavior works well in mainframe systems, but may work against multimedia applications. For example, a video application that performs a filtering computation on video frames may look compute-bound to a time-sharing scheduler and may therefore get a low priority compared to network and I/O activity on the machine. This may occur even if the video application is the most important activity to the user.

Consider a teleconferencing application that display several video streams on the screen at the same time. A fair time-sharing scheduling algorithm would give each of these streams an equal share of the processor, resulting in the same frame rate for all of the video displays. This might not be appropriate for the particular application. The user might want more control over where resources are focused, perhaps to show a higher frame rate for the person in the conference who has the floor.

In many operating systems such as UNIX [62], VMS [80] and Windows NT [24], the time-sharing scheduling policy is augmented with a fixed priority extension. The extension is usually in the form of a range of fixed "real-time" priorities. With fixed priorities, it is possible to exercise more control over how the processor is scheduled, but there are other problems. Many of these issues arise in the context of real-time operating systems as well, where fixed priority scheduling is commonly used. The next section addresses some of the difficulties of real-time programming with fixed priority schedulers.

2.4.3 Real-time operating systems

Much work in recent years focuses on how to apply real-time systems techniques to multimedia systems and applications. This includes work directed at methods for using technology available in commercial real-time operating systems as well as efforts to build research prototype operating systems.

2.4.3.1 Commercial real-time systems

Most commercial real-time operating systems support fixed priority (FP) scheduling of processes [99,104]. FP schedulers while useful for real-time scheduling, cannot by themselves support multimedia applications.

For example, fixed priority schedulers have no mechanism for dealing with overflow situations. In general, real-time operating systems do not have the mechanisms for deciding whether enough resources are available in a system to run a new application; they do not have support in the system for admission control. Furthermore, there are no mechanisms for detecting and dealing with overload situations when too many applications are allowed to run. These issues of load management are typically addressed in an ad hoc manner by subsystems specific to particular applications.

Even if a system user could determine that a particular collection of applications could run successfully on a system, the problem of determining what priority assignment should be used remains unsolved. An application designer may use multiple threads or processes in the implementation of the program, and that designer will undoubtedly know enough about the processes' computational requirements, timing constraints, and precedence relationships to assign priorities in a reasonable way. However, when running several such applications developed by different people on the same system, the question arises: How should priorities be assigned to processes in different applications in a way that will result in correct timing behavior? Without global knowledge of all processes and their timing constraints, assigning priorities appropriately is exceedingly difficult.

In practice, commercial operating systems are used mainly in embedded applications where designers carefully measure the resource requirements and coordinate scheduling based on a scheduling analysis of the specific task set designed for the application [68]. The system designers have global knowledge about resource requirements, and they use that information in the scheduling analysis to generate a priority assignment. This makes the systems rigid and difficult to maintain. Much more flexibility would be desirable.

2.4.3.2 Research prototype real-time systems

Several research prototype operating systems have applied results from real-time scheduling theory to multimedia applications [3,53,126]. The DASH kernel [3] used an admission control algorithm based on a timeline and then used earliest deadline scheduling to actually sequence the tasks. Other systems used analyses from real-time scheduling theory to guarantee timing constraints for applications. For example, YARTOS [53] uses algorithms for scheduling sporadic tasks [52] to guarantee timing constraints. In order to guarantee performance, the computational requirements of the applications must be measured and analyzed along with the timing constraints such as delay bounds. Then the application can be run with the expectation that timing constraints will be satisfied.

In RT-Mach [125], much of the work on support for multimedia applications (other than the work described in this dissertation) used the traditional rate monotonic scheduling algorithm. As with YARTOS, the computational requirements of applications were measured in advance and analyzed to ensure that timing constraints would be met. The applications could then run successfully on the system. Much of the work on RT-Mach centered on high-performance real-time threads packages [92] and QOS managers [127].

These systems took a careful approach to analyzing and guaranteeing timing requirements for multimedia applications based on real-time scheduling analyses. They also incorporated advanced real-time system features such as priority inheritance protocols [108] and inheritance protocols for deadline scheduling [16]. These features are essential for supporting strong real-time guarantees among programs that share data and interact in other ways.

2.4.4 Soft real-time system support

Several multimedia systems have used scheduling algorithms like earliest deadline first to make the system more sensitive to timing without necessarily guaranteeing that timing constraints will be met. For example, a system based on Chorus [103] proposed using deadline scheduling with no admission control [21]. Other algorithms such as lottery scheduling [133] attempt to support multimedia applications using proportional sharing of resources without real-time delay guarantees. A deterministic version of the approach called stride scheduling [134] was proposed to better support multimedia applications.

These types of systems are able to be more sensitive than time-sharing systems to the timing constraints of multimedia applications, but without effective admission control, overload cannot be prevented.

2.5 Reserve abstraction

The reserve abstraction described in this dissertation addresses several of the key problems raised above. The abstraction provides a framework for reasoning about resource reservation in an operating system. Other research efforts have focused on reservation of different resources in isolation. A framework to unify various reservation algorithms is needed.

The reserve abstraction gives resource reservation first-class status in the operating system. Reserves can be allocated independent of any particular process, and references to reserves can be passed around and bound to different processes as appropriate. For example, a real-time client might pass information about its timing constraints to a server to ensure expedited service. This is in contrast to the approach where timing constraints and resource usage requirements are associated directly with processes which makes it difficult or impossible to have one process temporarily take on the timing constraints of another process.

The key feature of the reserve abstraction is the enforcement mechanism. This prevents applications from overrunning their reservations if that would interfere with the timing requirements of other reserved activities.

The reservation framework and first-class status of reserves provide the power and flexibility to deal with the problems that arise in real-time system design and practical resource management. And the enforcement mechanism guaranteed proper resource scheduling. In combination, these aspects of the reserve abstraction offer an effective solution to the general real-time programming problem.