Preemptibility in Real-Time Operating Systems

Clifford W. Mercer and Hideyuki Tokuda
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

Real-time operating systems generally depend on some form of priority information for making scheduling decisions. Priorities may take the form of small integers or deadline times, for example, and the priorities indicate the preferred order for execution of the jobs. Unfortunately, most systems suffer from some degree of priority inversion where a high priority job must wait for a lower priority job to execute. We consider the nature of the non-preemptible code sections, called critical sections or critical regions, which give rise to this priority inversion in the context of a soft real-time operating system where average response time for different priority classes is the primary performance metric. An analytical model is described which is used to illustrate how critical regions may affect the time-constrained jobs in a multimedia (soft real-time) task set.

1 Introduction

The priority assignment in a real-time operating system represents the importance that the programmer places on each task. Because such systems must be responsive to external events, the tasks to be scheduled and their priority ordering can change very quickly. The scheduler and various mechanisms that provide input to the scheduler must be able to react quickly to the changing task mix if the priorities are to be honored. The problem is that circumstances in the actual operating system may hinder an immediate response to external events. For example, in the code for the scheduler it is often necessary to disable hardware interrupts to prevent interrupt handlers from invoking the scheduler again (schedulers are typically not re-entrant). During the time that interrupts are disabled, the system is oblivious to external events, and the external events will get no response from the system until after the interrupts have been enabled again. This part of the code where interrupts are disabled is called a critical region.

1.1 Critical Regions and Priority Inversion

Critical regions play havoc with the intended priority ordering of the tasks in the system. For the duration of the critical region, the priority structure is suspended, and the executing task essentially transforms itself into the highest priority task. Thus a low priority task may take, for some duration, the highest priority in the system. If no high priority activity becomes ready during this time, there is no problem. However, if a high priority activity arises, the corresponding external event would not cause an immediate interrupt, and the high priority (external) activity would be delayed while the lower priority internal activity executes. This is a priority inversion (using a broad interpretation of the term used in [13]) since the effective priorities of the two activities have been inverted.

Sections of code where interrupts are disabled are examples of critical regions where the effects are felt by all the tasks in the system. Critical sections protected by semaphores that are shared by a small number of tasks are another source of priority inversion on a limited scale. Intuitively, the duration of the critical sections should be small so as to avoid priority inversion and enhance response time. On the other hand, non-preemptive processing reduces context switching overhead and improves cache performance and pipeline performance. Critical sections are also needed to ensure data integrity in programs using shared memory. We now consider some of the issues involved in this tradeoff.

1.2 Necessity of Critical Regions

Interrupt handlers for various devices are sources of critical regions where interrupts are disabled for some period of time. There is a tradeoff between doing buffering

This research was supported in part by a National Science Foundation Graduate Fellowship, by the U.S. Naval Ocean Systems Center under contract number N66001-87-C-0155, by the Office of Naval Research under contract number N00014-84-K-0734, by the Federal Systems Division of IBM Corporation under University Agreement YA-278067, and by the SONY Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of NSF, NOSC, ONR, IBM, SONY, or the U.S. Government.

and other data processing in interrupt handlers and paying the overhead for scheduling a job to do the work. Some systems do character buffering, network packet buffering, and even network protocol processing in hardware interrupt handlers. 4.3 BSD [8], for example, does network protocol processing at a "software interrupt level" which has a higher priority than any of the schedulable activities in the system (and a lower priority than hardware interrupt handlers). This is an example of a critical region which is not preemptible by normal processes, which is not preemptible by other network protocol processing activities, but which allows preemption by hardware device interrupt handlers. This approach avoids scheduling events and context switches and therefore yields better throughput for equally important messages. If messages have priorities, however, this approach (with FIFO queueing) will result in very poor performance for the high priority messages.

Interrupts are also disabled during memory operations affecting the address space of a process, during interprocess communication (IPC), and during some systems calls such as I/O primitives. In these cases, disabling interrupts is a fast, simple form of mutual exclusion for system activities. But again, priority inversion will occur if low priority activities disable interrupts for long periods of time.

Critical regions in user programs are generally protected by semaphores or some other synchronization mechanism. These have a lesser impact on overall system performance, but the non-preemptible nature of such critical regions has implications among the synchronizing activities. For example, high priority and low priority activities might share a database, and the scheduling of transactions performed on that database becomes very important from the perspective of the high priority job. Among user programs, the duration of the critical region may vary widely, depending on the specific application.

1.3 Reducing the Size of Critical Regions

There are many incentives for reducing the size of the critical regions. In time-sharing systems, the critical regions where interrupts are disabled are kept as small as (conveniently) possible so as to avoid the loss of data coming in on external devices. Allowing the device interrupt handler to at least buffer the data will save the data from being lost. In real-time systems, the maximum critical region is carefully bounded so that the response time to external interrupts can be bounded [2]. For example, the scheduler in iRMX uses additional data structures and software locking to avoid disabling interrupts for long periods of time when manipulating internal lists of arbitrary length [12].

Other incentives for increased preemptibility (or, equivalently, the reduction of the size of critical regions) are derived from scheduling theory. The rate monotonic priority assignment for scheduling periodic activities depends on the complete preemptibility of the task set [9]. Extensions to this theory for allowing critical sections with bounded execution times show a reduction in real-time performance guarantees (based on a worst-case analysis) [13, 14]. This degradation increases with increases in critical region size.

1.4 Preemptibility and Packet Multiplexing

Packet switching communications systems derive much of their appeal from the efficiency of multiplexing several bursty data streams. If the size of packet is very large, other packets will be delayed while waiting for the non-preemptible handling of the large packet. The effect of allowing large packets is that the average response time is increased. For this reason, packets are often restricted in size [15]. This reasoning does not take into account, however, the case where priorities are associated with the packets. The question of whether a long, high priority packet should have the power to increase the average response time of all the packets still remains. The problems encountered with large packets also appear in the context of multimedia communication systems. For example, a network which is multiplexing voice packets, interactive data, and bulk data messages may suffer from the non-preemptibility of long facsimile transmission [3]. In the scheduling of multimedia operating systems, the reduction of critical region size is important as well.

1.5 Preemptibility and Queueing Analysis

The idea of priority queueing has been around for quite some time, and preemption is commonly associated with prioritized service. Queueing theory models typically treat the following types of service: non-preemptive, preempt-resume, and preempt-restart [7]. But these models are greatly simplified compared to actual operating systems. Most operating systems allow for some form of preemption, but no operating system provides a fully preemptible environment. More detailed models are needed to more precisely evaluate the importance of preemptibility in modern operating systems.

2 Modeling Critical Regions

In multimedia operating systems, preemptibility and response time are much more important than in time-sharing systems. Time-sharing systems tend to sacrifice predictability and response time for average throughput. Commercial real-time systems, on the other hand, sacrifice higher-level services for fast interrupt response time. We want to find a middle ground where the advantages of preemptibility are realized and where the level of sophistication of the services is comparable to workstation operating systems. The problem is one of evaluating design alternatives to determine the level of preemptibility that is most appropriate for the application domain.

2.1 Non-Preemptive and Preemptive Queueing Models

Common queueing models do not reflect the various preemptibility characteristics that appear in actual operating systems, although queueing theory results do provide some general insight. To demonstrate the limitations of queueing models for evaluating preemptibility characteristics, we consider a simple system with 2 tasks. One is a high-priority task that is meant to be periodic like a multimedia data stream, and the other is a low priority task corresponding to background activity in the system. Both tasks have a Poisson arrival process and deterministic (constant) service time. Their parameters are given in the following table.

Task	$1/\lambda$		s		Description
τ_1	40	ms	1	ms	high priority task
τ_2	*var*	ms	10	ms	low priority task

The column labeled $1/\lambda$ gives the average interarrival time for each arrival process. The interarrival time of τ_2 is variable and is computed given a target load, and we evaluate the average response time of the tasks for various loads.

We use the queueing theory results presented in [1] to evaluate the average response time of these tasks under three different system configurations:

- M/D/1 with a FIFO queueing discipline and a non-preemptible server,
- M/D/1 with a priority queueing discipline and a non-preemptible server, and
- M/D/1 with a priority queueing discipline and a preemptible server.

In Figure 1, we show the results for the case using FIFO queueing with a non-preemptible server. The tasks are divided into 2 classes which have different parameters for the arrival process and service time, but the service policy is FCFS without regard for the class. The load is the total utilization of the server; the load varies with the average interarrival time of the low priority task. We see that, under high load, the high-priority τ_1 gets caught in the queue with the lower priority activity and suffers in terms of response time.

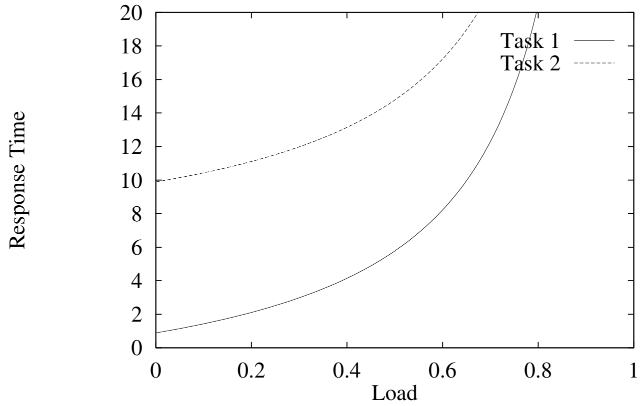

Figure 1: M/D/1, Classes, FIFO Queueing

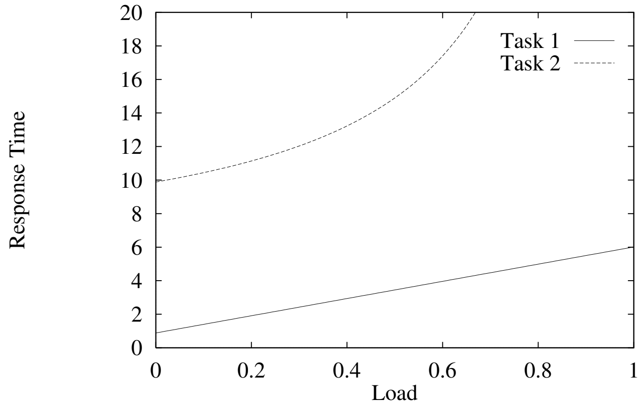

Figure 2: M/D/1, Non-Preemptive Priority Queueing

If the classes are serviced based on their priority, we can expect much better results from the high priority activities. Figure 2 shows the results for priority queueing with a non-preemptible server. In this case, the entire computation of the low priority activity is a critical region, and the high priority activity suffers only from this effect. τ_1 no longer suffers from FIFO queueing delays as in the previous case.

If the service were preemptible, we would expect an improved response time in the high priority activity. With perfect preemptibility (i.e. no critical regions) and the simplifying assumption that there is no cost in preemption, we get the results in Figure 3. This is the case with priority queueing and a preemptible service.

A major problem with these analyses is that real tasks in a real system are never completely non-preemptible or completely preemptible. Most tasks are composed of some

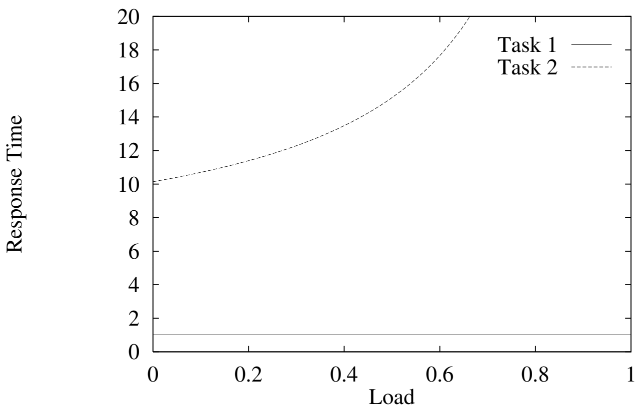

Figure 3: M/D/1, Preemptive Priority Queueing

sections of code which are preemptible and others which are non-preemptible. Our goal is to develop a model to evaluate the performance of tasks in this middle ground.

An additional problem with queueing models is that the usual arrival processes and service time distributions are not necessarily representative of workload on networked personal workstations. This is particularly true when multimedia applications are considered. We expect that the computational load on multimedia workstations will consist of a small number of periodic activities (such as audio or compressed video data streams) along with the usual mix of network file transfer activity, compilation, text formating, etc.

2.2 A Queueing Model with Critical Regions

We have developed a model which allows us to evaluate the importance of preemptibility and critical region size in the context of task sets which might be characteristic of multimedia workstations. The model approximates the queueing system of interest. We assume that

- there is a small number of periodic tasks concerned with audio or video data streams,
- the periods are synchronized, i.e. the phase offset is zero (this is a worst case assumption),
- the computation time for these tasks is fixed and the computation is preemptible (this assumption frees us from the tedious analysis of interactions between periodic tasks when the primary focus is the interaction of periodic tasks with the background load),
- the periodic tasks' busy period is shorter than the smallest period,
- only one background arrival process is present; the computation time of the background activities may contain one or more preemptible regions and non-preemptive critical regions, and
- the background arrival process is Poisson with deterministic service time.

Suppose we have a task set $\tau_1, \tau_2, \cdots, \tau_n$ where the tasks $\tau_1, \tau_2, \cdots, \tau_{n-1}$ are periodic and τ_n describes the background arrival process. For a periodic task, τ_i, we denote the period by T_i and the computation time by C_i. The background activity has a Poisson arrival process with the mean interarrival time of T_n and a deterministic service time, C_n. The critical regions in the computation time of the background activity are specified by $S_{n,k}$ which is the duration of the k^{th} segment of computation. These segments of computation may be preemptible or non-preemptible. We denote the set of preemptible segments by $\mathcal{P}_n$ and the set of non-preemptible segments by $\mathcal{N}_n$. We assume that periodic tasks are in order of increasing period, that priorities are assigned in rate monotonic fashion, and that $T_{n-1} < T_n$.

The model predicts the average response time for each priority class. The response time for a task, τ_i, has four components:

1. the task's computation time (C_i),
2. delay due to higher priority tasks ($D_{i,>}$),
3. delay due to equal priority tasks ($D_{i,=}$), and
4. delay due to lower priority tasks ($D_{i,<}$).

For periodic tasks, the delay due to lower priority tasks does not include any component from lower priority periodic tasks since the periodic tasks are all preemptible. Thus for periodic tasks, the delay due to lower priority tasks comes only from the background activity.

For a periodic task τ_i, we compute the delay due to higher priority tasks ($D_{i,>}$) by calculating the number of times each higher priority task will have an arrival that coincides with the arrival of the current task. When the arrivals are coincident, τ_i will be delayed; other interference is avoided due to the assumption that the periodic tasks' busy period is smaller than the smallest period. The delay due to higher priority tasks is given by

$$D_{i,>} = \sum_{j:T_j<T_i} C_j \frac{Ti}{lcm(T_j, T_i)}. \quad (1)$$

where *lcm* is the least common multiple.

The delay due to tasks of the same priority is based on the assumption that the tasks are synchronized. We make the assumption that τ_i will have to wait for other tasks of

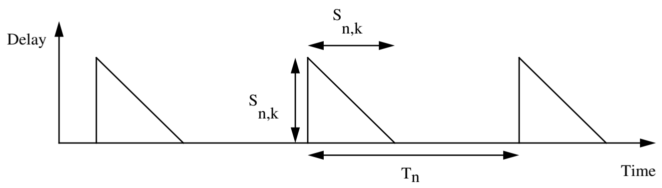

Figure 4: Delay Suffered by Higher Priority Arrival

equal period that fall before it in the (arbitrary) priority ordering. The delay due to equal priority tasks is

$$D_{i,=} = \sum_{j:T_j=T_i \wedge j<i} C_j. \qquad (2)$$

The delay due to lower priority tasks depends only on the background activity. We take the sum of expected delay due to all critical regions in the low priority task. The expected delay due to a critical region $S_{n,k}$ is calculated by integrating the delay suffered by a high priority activity contending with the critical region during the expected interarrival time of the critical region. Figure 4 shows the shape of the function of delay due to the critical region. During the critical region, the delay suffered by a higher priority task decreases linearly; if the higher priority task arrives during time that the critical region is not running, the delay is zero. The expected delay due to lower priority tasks is computed as follows:

$$D_{i,<} = \sum_{S_{n,k} \in \mathcal{N}_n} \frac{S_{n,k}^2}{2T_n}. \qquad (3)$$

The delay due to other periodic activities cannot be negotiated in general, and the design decisions of interest will revolve around the size and number of the critical regions in the background activity. So the total delay suffered by periodic task τ_i is then:

$$D_i = D_{i,>} + D_{i,=} + D_{i,<}. \qquad (4)$$

Adding the computation time for τ_i to D_i would then give us the total response time.

We use a rough estimate for the delay experienced by the background task since this measure is not a primary focus of this study. This delay is calculated using the M/D/1 result for queueing delay along with a degradation factor that slows down the execution of the low priority activity based on the periodic task load. We calculate the queueing delay as

$$D_{n,q} = (1 + \rho_p)\frac{\rho_n C_n}{2(1 - \rho_n)}. \qquad (5)$$

where ρ_p is the utilization of the periodic tasks and ρ_n is the utilization of the background task. We can then compute the total delay as the sum of the queueing delay and the computation time (degraded based on the periodic load).

$$D_n = D_{n,q} + C_n(1 + \rho_p). \qquad (6)$$

We are now in a position to analyze the simple task set described above for cases where the low priority activity is composed of one or more critical sections. This analysis will, for example, indicate whether reprogramming the low priority activity to yield at one or more preemption points will have a significant impact on the response time of the high priority activity.

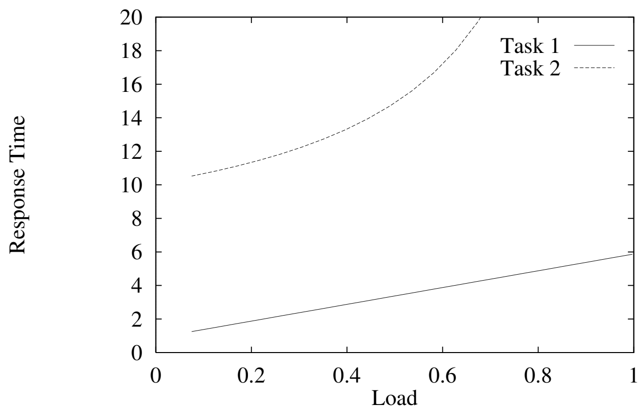

Figure 5: Limited Preemption, Priority Queueing (1 Large Critical Region)

Figure 5 shows results of this model for the case where the low priority task consists of a single critical region that lasts for the entire computation time. This case is identical to the non-preemptive priority queueing case shown in Figure 3, and the results are the same.

We now consider what would happen if the critical region were divided into two critical regions, each half the size of the original. In other words, we consider the effect of inserting a preemption point in the middle of the computation. Figure 6 shows that the average response time of the high priority task still increases linearly with the load. But with the critical region divided into two parts, the slope of the line is smaller.

A further decrease in the size of the critical regions (with resulting increase in the number of smaller regions) results in even smaller average response times for a given load, as shown in Figure 7. In this case, the critical region was divided into five equal parts.

In the next section, we summarize some of the implications of this model.

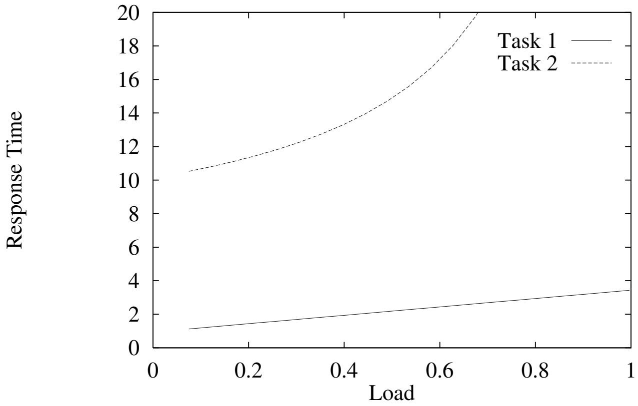

Figure 6: Limited Preemption, Priority Queueing (2 Smaller Critical Regions)

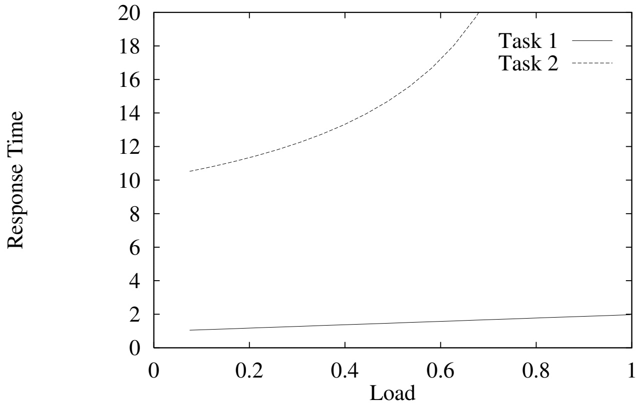

Figure 7: Limited Preemption, Priority Queueing (5 Smaller Critical Regions)

3 Implications of the Model

This model gives us a framework in which to think about the effects of critical regions on the performance of tasks in a soft real-time operating system. We will discuss several observations about the model:

- delay due to fixed-size critical regions in the background activity increases linearly with the background load (in this case only T_n varies as the load increases),
- delay due to the k^{th} critical region scales linearly with $S_{n,k}$ (with constant load across the variation),
- dividing a single critical region into 2 equal parts reduces the expected delay due to that critical region by a factor of 2,
- more generally, dividing a single critical region into m equal parts reduces the delay due to that critical region by a factor of m, and
- dividing a single critical region into 2 equal parts results in a minimal delay compared with division into 2 unequal parts.

In the following paragraphs, we will explore these implications in more detail.

Delay due to fixed-size critical regions in the background activity increases linearly with the background load. This property is evident in Figures 5-7. We can observe this from (3) in the previous section. The delay due to the k^{th} critical region is:

$$\frac{S_{n,k}^2}{2T_n}. \tag{7}$$

We consider the effect of an increasing load when the critical region size remains the same, i.e. $S_{n,k}$ is constant. If the load is increasing, then T_n is decreasing. We note that $S_{n,k}/T_n$ is the contribution of the k^{th} critical region to the load, denoted $\rho_{n,k}$. So we can re-write (7) as

$$\frac{S_{n,k}\rho_{n,k}}{2}. \tag{8}$$

From this expression, it is clear that as $\rho_{n,k}$ increases, the delay term will increase linearly.

Delay due to the k^{th} critical region scales linearly with $S_{n,k}$ (with constant load across the variation). It is clear from (8) above that when $\rho_{n,k}$ is constant and $S_{n,k}$ varies, the corresponding delay term increases linearly with increases in $S_{n,k}$.

Dividing a single critical region into 2 equal parts reduces the expected delay due to that critical region by a factor of 2. Suppose we have a critical region $S_{n,k}$. The delay term for that critical region is $S_{n,k}^2/2T_n$. If we split the critical region into two parts, each of duration $S_{n,k}/2$, the delay term becomes

$$\frac{\left(\frac{S_{n,k}}{2}\right)^2}{2T_n} + \frac{\left(\frac{S_{n,k}}{2}\right)^2}{2T_n} = \frac{S_{n,k}^2}{4T_n} = \frac{1}{2}\left(\frac{S_{n,k}^2}{2T_n}\right). \tag{9}$$

And from this we see that the delay term smaller than the original delay term by a factor of 2.

Dividing a single critical region into m equal parts reduces the delay due to that critical region by a factor of m. By extending (9), we can see that if the critical region is divided into m equal parts, the delay term becomes

$$\frac{\left(\frac{S_{n,k}}{m}\right)^2}{2T_n} + \cdots + \frac{\left(\frac{S_{n,k}}{m}\right)^2}{2T_n} = m\frac{1}{m^2}\frac{S_{n,k}^2}{2T_n} = \frac{1}{m}\left(\frac{S_{n,k}^2}{2T_n}\right). \quad (10)$$

So the expected delay term is a factor of m smaller than the original delay term. We can also see from this equation that if we consider the limit as m goes to ∞, the delay goes to zero. This indicates that the background task is fully preemptible.

$$\lim_{m \to \infty} \frac{1}{m}\left(\frac{S_{n,k}^2}{2T_n}\right) = 0. \quad (11)$$

Dividing a single critical region into 2 equal parts results in a minimal delay compared with division into 2 unequal parts. To illustrate this, we consider the case where a single critical region of duration $S_{n,k}$ is divided into two parts: $\alpha S_{n,k}$ and $(1-\alpha)S_{n,k}$. The delay term is then

$$\frac{(\alpha S_{n,k})^2}{2T_n} + \frac{[(1-\alpha)S_{n,k}]^2}{2T_n} = [\alpha^2 + (1-\alpha)^2]\left(\frac{S_{n,k}^2}{2T_n}\right). \quad (12)$$

And the expression $[\alpha^2 + (1-\alpha)^2]$ is minimized when $\alpha = 1/2$. So the delay term is minimized when the critical region is divided into 2 equal parts.

4 Application of the Model

In this section, we illustrate the use of this model in analyzing the performance in two areas: packet length in networks and protocol processing software.

4.1 Modeling Packet Length Effects

The packet length allowed by a network protocol determines the multiplexing characteristics of the network [15]. A small packet length will allow fine-grain multiplexing, ensuring fairness in a time-sharing system and promoting proper prioritized handling in protocols with priority information. A large packet length will allow a single activity to monopolize the network resources.

Limitations on packet length are particularly important in a system which integrates different types of time-constrained and non-time-constrained traffic. For example, mixing voice packets, interactive and bulk data, and facsimile packets on a single network can give rise to delay problems, especially where the voice packets are concerned [3]. In this section, we consider the delay characteristics of small high-priority voice packets which contend with large bulk data packets. And we compare the performance in that case to the situation where the packet size is limited. Arguments along these lines are the motivation for the short packet length in ATM networks where the cell (or packet) size is fixed at 53 bytes [15].

Phone-quality voice data consists of 8-bit samples with a sampling rate is 8000Hz. This generates 64kbps. A typical packetization time is about 50ms [3], so the voice packet size is 400 bytes. On a 100Mbps local area network medium, the voice packet stays on the medium for 32 μs. If we take the bulk data packet size to be 4500 bytes, a data packets stays on the medium for 360 μs. We now construct a task set with one voice task as defined above and a background data task with a variable period (to vary the load), and we consider the delay characteristics within a local area network. The specification is as follows.

Task	$1/\lambda$		s		Description
τ_1	50	ms	.032	ms	voice task
τ_2	*var*	ms	.360	ms	bulk data task

Figure 8 shows the result from this task set. The voice packet delay increases significantly as the load increases. At high loads, the delay is about 4 times the delay at low load.

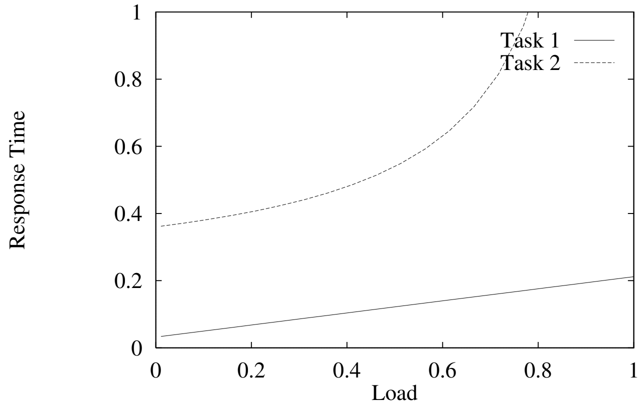

Figure 8: Voice/Data with Long Packets

We now consider the case where the maximum packet length is on the order of the voice packet length. With a maximum packet size of 400 bytes, no packet stays on the network medium for more than 32 μs. Again we vary the period of the background activity to vary the load. The revised specification is:

Task	$1/\lambda$		s		Description
τ_1	50	ms	.032	ms	voice task
τ_2	*var*	ms	.032	ms	bulk data task

In Figure 9, we can see the improvement in the response time of voice packets. The response time is almost constant while the load increases. The result is that even at high load, the voice packets are able to bypass the background bulk data activity.

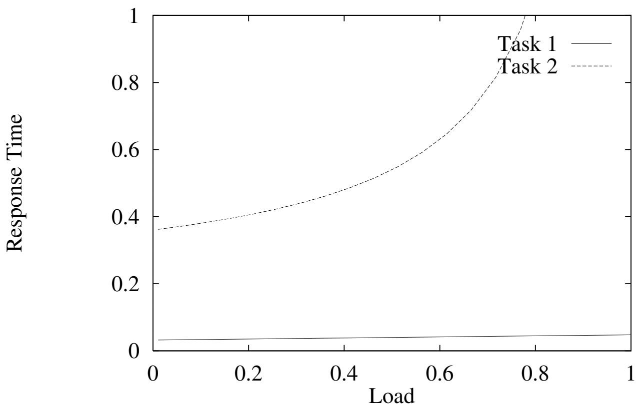

Figure 9: Voice/Data with Short Packets

This comparison provides an illustration of the reasoning behind the choice for 53-bytes fixed-size cells in ATM [15]. The small packet size is particularly important in the context of a Wide Area Network (WAN) where there may be many gateways and links between communicating endpoints. If low priority contention at each resource delayed the packet significantly, the sum of delay at all resources would accumulate very quickly.

4.2 Modeling Preemptibility in Protocol Software

Many network operating systems use non-preemptive protocol processing engines where the processing for each packet is a single critical section (to simplify coding and stream-line execution). The value of priority queueing and preemptibility in the protocol processing software of a distributed operating system was explored by means of simulation in [10]. This study considered the performance of several protocol processing techniques in terms of average response time, variance in response time, and priority inverted utilization. In this section, we will illustrate the use of this model in analyzing contention for protocol processing resources, including processor cycles. The current analysis is only in terms of average response time as predicted by the model.

We use the task set of [10] which specifies the arrival of packets to the protocol processing software. The following describes the task set which differs from the original task set in the arrival process of the background traffic. In the original task set, we use a deterministic arrival process to generate spikes of contention. In the current model, we use a Poisson arrival process for the background activity.

Task	$1/\lambda$		s		Description
τ_1	20	ms	1	ms	high priority task
τ_2	20	ms	1	ms	high priority task
τ_3	40	ms	1	ms	medium priority task
τ_4	*var*	ms	1	ms	low priority task

The protocol processing software architectures that we will evaluate are described in more detail in [10]. Briefly, they are:

- T1P – The T1P structure uses a single thread to process packets, but the packets are queued in priority order. The service is non-preemptible since there is only one thread.
- TnP – This approach uses n threads to provide preemptible service to packets which are queued in priority order [1]. The number n is the number of packet priority levels.

And we also consider two additional techniques that fall between the T1P and TnP techniques. T1P is completely non-preemptible, and TnP is completely preemptible. We want to consider techniques with limited preemption:

- T1S2P – This refers to the single-threaded approach with 2 critical regions (instead of 1 large critical region). Queueing is in priority order.
- T1S4P – In this approach, we have 4 critical regions instead of 2. Queueing is again in priority order.

Figure 10 shows the case where the protocol processing service is non-preemptible. The curve labeled "Task 1&2" is the average of the response times of the two high priority tasks. And the curves for Task 3 and Task 4 are labeled accordingly. In this case, the interference due to lower priority traffic is quite small. The average response time curve for τ_1 and τ_2 is at 1.5 ms for 20% load and only increases to about 2 ms at 100% load. The fact that the critical region size of the low priority background traffic is the same as the computation time for the high priority tasks means that the average interference will be limited.

For comparison, Figure 11 shows the case where the high priority tasks' average response time is almost constant. This case is approximated using the model with the background activity split into 10 critical regions with possible preemption points between each one. Using 10 critical regions yields relatively high preemptibility.

[1]The use of threads for protocol processing does not imply preemptive service, e.g. the x-kernel [5, 11] assigns a thread to each packet, but the protocol processing in the threads is still non-preemptive.

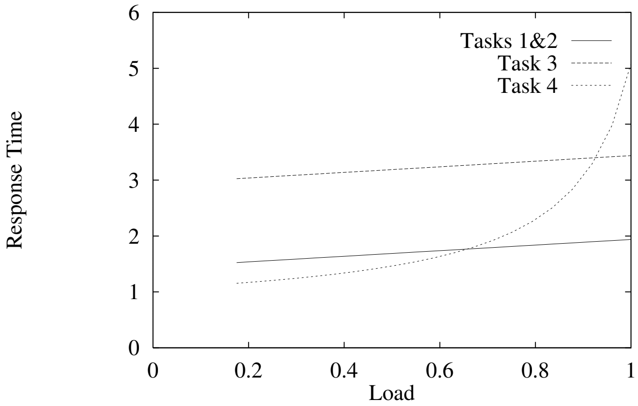

Figure 10: Protocol Processing, T1P

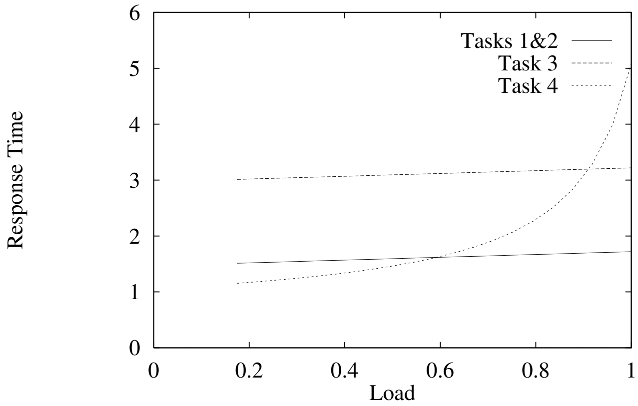

Figure 12: Protocol Processing, T1S2P

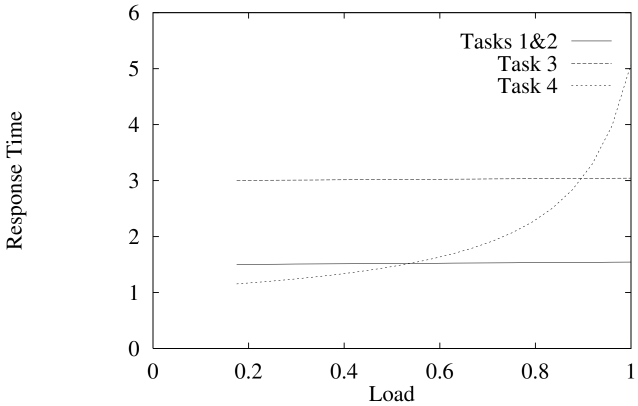

Figure 11: Protocol Processing, TnP

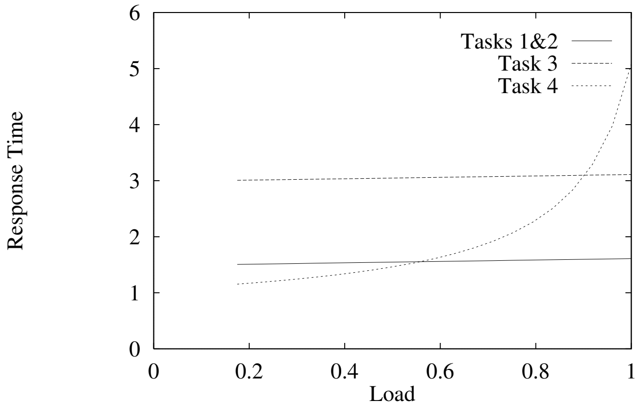

Figure 13: Protocol Processing, T1S4P

In Figure 12, we show the effect of splitting the critical region of the background task in 2 parts. This gives us a slight improvement over the case where we have a single critical region. Figure 13 shows the result when the critical region of the background task consists of 4 parts. Again, the high priority tasks' average response time is almost constant.

From this analysis, we can conclude that using priority queueing in the protocol processing software goes a long way toward improving the response time of high priority activities. However, if the computation times of the low priority tasks are on the order of the computation times of the high priority tasks, the additional benefits of preemptibility are limited. This is in contrast to the case where the low priority tasks have a large computation time compared with the high priority tasks, as was the case in the examples in Section 2.2 and in Section 4.1. Under those conditions, preemptibility is a much more important factor.

5 Discussion

The model described in this paper provides an initial assessment of the effects of preemptibility in soft real-time systems. However, an important factor has been left out. Preemptibility is not free in real systems; there is a context switch cost for each preemption. A more complete model would incorporate the cost of context switching and would provide a means to compare the response time improvement from the preemptibility and the utilization degradation due to the additional overhead of context switching. The current study provides a basis for future investigation in this area.

While this model does, in some sense, reflect a more detailed view of the system, the difficulty in finding a suitable arrival process to represent traffic in a real system limits the ability of this model to produce results that directly predict the performance of the system. This objective of this work is to provide a means to study the value of preemptibility in a general sense. In particular, we recognize that common scheduling techniques and software structuring methodologies work well in most cases. The problem in a time-constrained system is that occasional performance anomalies cannot be tolerated, at least in the high priority traffic. So we want to provide a means to evaluate the performance of particular scheduling policies and software structures under the stress of transient overload.

A better traffic arrival specification that exhibits the bursty behavior that gives rise to transient overloads would improve the usefulness of the model. Recently, some attempts have been made to characterize and model network traffic [4, 6], but it is impossible to characterize the traffic on a multimedia network without much more experience in the operation of such networks.

6 Conclusion

The importance of preemptibility in soft real-time operating systems is clear, and the model described in this paper provides some insight into the performance differences in systems which are highly preemptible vs. systems which are lazy with regard to preemptibility. The model fills the void in traditional queueing systems between completely preemptible and completely non-preemptible servers by providing a method to analyze the behavior of servers which have limited preemptibility. Several implications of the model are discussed, and the usefulness of the model in evaluating design tradeoffs is demonstrated using two example application areas.

References

[1] A. O. Allen. *Probability, Statistics, and Queueing Theory*. Academic Press, Boston, MA, 2nd edition, 1990.

[2] D. R. Cheriton, M. A. Malcolm, L. S. Melen, and G. R. Sager. Thoth, a Portable Real-Time Operating System. *CACM*, 22(2):105–115, February 1979.

[3] S. S. Gaitonde, D. W. Jacobson, and A. V. Pohm. Bounding Delay on a Multifarious Token Ring Network. *CACM*, 33(1):20–28, January 1990.

[4] R. Gusella. A Measurement Study of Diskless Workstation Traffic on an Ethernet. *IEEE Transactions on Communications*, 38(9):1557–1568, September 1990.

[5] N. C. Hutchinson and L. L. Peterson. The *x*-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[6] R. Jain and S. A. Routhier. Packet Trains – Measurements and a New Model for Computer Network Traffic. *IEEE Journal on Selected Areas in Communications*, 4(6):986–995, September 1986.

[7] L. Kleinrock. *Queueing Systems, Vol. 2: Computer Applications*. Wiley Interscience, 1976.

[8] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

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

[10] C. W. Mercer and H. Tokuda. An Evaluation of Priority Consistency in Protocol Architectures. In *Proceedings of the IEEE 16th Conference on Local Computer Networks*, pages 386–398, October 1991.

[11] S. O'Malley. Private communication.

[12] T. G. Saponas and R. B. Demuth. The Distributed iRMX Operating System: A Real-Time Distributed System. In A. K. Agrawala, K. D. Gordon, and P. Hwang, editors, *Mission Critical Operating Systems*, chapter 16, pages 208–231. IOS Press, Amsterdam, 1992.

[13] L. Sha and J. B. Goodenough. Real-Time Scheduling Theory and Ada. *IEEE Computer*, 23(4):53–62, April 1990.

[14] 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.

[15] F. Tobagi. Fast Packet Switch Architectures For Broadband Integrated Services Digital Networks. *Proceedings of the IEEE*, 78(1):133–167, January 1990.