Scheduling Real-Time Applications in an Open Environment *

Z. Deng J. W.-S. Liu
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

May 14, 1997

Abstract

This paper focuses on the problem of providing run-time support to real-time applications and non-real-time applications in an open system environment. It extends the two-level hierarchical scheduling scheme in [12] for scheduling independently developed applications. The extended scheme removes the following two restrictive requirements of the scheme in [12]: (1) real-time applications that are scheduled preemptively must consist solely of periodic tasks, and (2) applications must not share global resources (i.e., resources used by more than one applications). Consequently, the extended scheme allows us to deal with a much broader range of real-time applications.

1 Introduction

Recent advances in real-time systems technology have given us many good schemes for scheduling hard real-time applications. Examples are [1, 2, 5, 8]. A weakness shared by most existing schemes is that schedulability analysis must be done globally (i.e., by analyzing all applications in the system together) in order to validate the schedulability of each application in the system. While the necessity of global schedulability analysis imposes no serious problem when the system is closed, it does so when the system is open. Here, by *closed system*, we mean one in which all real-time applications on each processor are known. Oftentimes, the applications are developed together, and the schedulability of every combination of applications that can run at the same time can be determined *a priori*. In contrast, in an *open system*, applications may be developed and validated independently. During run-time, the user may request the start of some real-time application whose schedulability has not been analyzed together with currently executing applications. The system must determine whether to accept the request and admit the new

*This work was supported in part by NASA Grant NAG 1-613 and USAF Grant F30602-97-2-0121

application. (It usually admits such a request only when the new application and all the existing real-time applications are schedulable.) A global schedulability analysis as an acceptance test is sometimes not feasible because many characteristics of real-time applications in the system are unknown. Even when it is feasible, such an acceptance test can be costly when the applications are multi-threaded and complex.

We proposed in [12] a two-level hierarchical scheme for scheduling an open system of multi-threaded, real-time applications together with non-real-time applications on a single processor. It allows different applications to be scheduled according to different scheduling algorithms. A real-time application in the open system can be either time-driven (i.e., its tasks are scheduled in a time-driven manner) or priority-driven (i.e., its tasks are scheduled in a priority-driven manner). If the application is priority-driven, the scheduling algorithm can be preemptive or nonpreemptive, fixed-priority or dynamic priority. More importantly, the two-level scheme allows the schedulability of each application to be validated in isolation from other applications. Because it does not rely on fixed allocation of processor time or fine-grain time slicing, it is suited for applications with varying time demands and stringent timing requirements.

The two-level scheme in [12] has serious limitations. While it can deal with all time-driven and nonpreemptive real-time applications, real-time applications that are scheduled preemptively must consist solely of periodic tasks. Moreover, applications must not share global resources (i.e., resources used by more than one applications). In this paper, we present algorithms and schedulability conditions that remove these limitations. Specifically, the extended two-level scheme takes into account the following characteristics of real-time applications:

- Tasks may have nonpreemptable sections.[1]
- Periodic tasks in a priority-driven application may have release time jitters.
- A priority-driven application may have aperiodic/sporadic tasks.
- Tasks in an application may request for local resources (i.e., the resources shared only by tasks in the application) and global resources (i.e., the resources shared by tasks of different applications).

[1]When a job is in one of its nonpreemptable sections, it effectively executes at the highest priority and cannot be preempted by any jobs in the system. We will also speak of applications that are scheduled nonpreemptively. A job in such an application is never preempted by other jobs in the application, but may be preempted by jobs in other applications.

Throughout most of the paper, we assume that the (maximum) execution time and relative deadline of each job in every real-time application become known when the job is released, so are when and how long the job may use local and global resources. The summary section will discuss the case when a job may actually execute for a shorter amount of time than its execution time.

Following this introduction, Section 2 describes related work. Section 3 describes the open system architecture supported by our scheme and discusses the problem addressed here. According to our scheme, each real-time application is executed by a server. Section 4 presents a sufficient schedulability condition which, when satisfied, ensures that each server is schedulable. (We will explain shortly what we mean by a server being schedulable.) Sections 5 presents a sufficient schedulability condition of real-time applications in the open system given that all servers are schedulable. Section 6 describes the extended two-level scheduling scheme used in the open system to schedule different types of real-time applications. Because it is designed so that the condition in Section 5 is satisfied, the schedulability of each application can be determined independently from others in the system. Section 7 is a summary and presents future work.

2 Related Work

The algorithms used in our two-level scheduling scheme resemble the proportional share resource allocation scheme proposed by Ion Stoica, *et al* [13]. In essence, their scheme creates for each process a slower virtual processor whose speed is a fraction of the speed of the physical processor, as does our scheme for each application. The objective of their scheme is to provide fair sharing of the processor among multiple processes running on one processor. In contrast, our scheme deals with the scheduling of multi-threaded real-time applications in an open system. It allows the schedulability of each application to be validated based on the assumption that the application runs alone on a slower processor, and ensures that every job in the application completes by its deadline in the open system. This difference in the design objectives of the two schemes leads to different structures of the scheduler. The scheme in [13] uses a single-level scheduler that schedules the requests of processes according to their virtual deadlines. We use a two-level scheduler, namely operating system scheduler and server schedulers, to schedule real-time applications, as we will describe in Section 3.

The constant utilization server algorithm [12] used in our scheme is essentially the same as the total bandwidth server algorithm developed by Spuri and Buttazo [9]. According to both the total bandwidth server and constant utilization server algorithms, when the budget of a server is replenished to e time units at time t, the new server deadline is set to $\max\{t, d\} + e/\sigma$, where d is the current server deadline and σ is the server size (i.e., the fractional processor bandwidth allocated to the server). A server is ready for execution only when it has budget, and it consumes its budget at the rate of one per unit time when it executes. However, according to the total bandwidth server algorithm, the server budget is replenished immediately after the budget becomes zero if the ready queue of the server is not empty. In contrast, according to the constant utilization server algorithm, the budget is replenished no earlier than the current server deadline. Consequently, a constant utilization server cannot make use of any background time, while a total bandwidth server can. We use constant utilization servers to execute hard real-time applications in the open system whenever it is possible to do so and there is no advantage to complete jobs in the applications early. Thus, we leave all the processor time not required by such applications to non-real-time and soft real-time applications. We use a total bandwidth server to execute all the non-real-time applications and soft real-time applications so that the system is more responsive for them.

The total bandwidth (and constant utilization) server algorithm is similar to the preemptive fair queueing and virtual clock algorithms proposed for scheduling network traffic through ATM switches [10, 11]. A communication link with link capacity C can be thought of as a processor with speed C, and a total bandwidth server with server size U can be thought of as a real-time connection that is allocated the bandwidth UC and scheduled according to the virtual clock algorithm. Indeed, the two-level hierarchical scheduling algorithm for nonpreemptive applications can be used to schedule multiple real-time message streams on each of the connections sharing the same output link of a switch.

3 Open System Architecture

Figure 1 shows the open system architecture supported by our two-level hierarchical scheduling scheme. The system has a single processor whose speed is one. The workload on the processor consists of a variable number N of real-time applications, called $A_1, A_2, \cdots, A_N$, together with non-real-time applications. In our discussion, we often say that each *real-time application A_k is schedulable on a processor of speed σ_k*. By this statement, we mean that the all jobs in A_k meet their deadlines when the schedulability of A_k is analyzed based on the assumption that A_k executes alone on a processor of speed σ_k and is scheduled according to the algorithm Σ_k chosen by its developer. We assume that every real-time application A_k in the system is schedulable on a slow processor with speed $\sigma_k < 1$. Similarly, we say that the *real-time application A_k is schedulable in the open system* if all jobs in A_k meet their deadlines when A_k is running in the open system together with other applications and the order in which jobs in A_k are executed is determined by algorithm Σ_k.

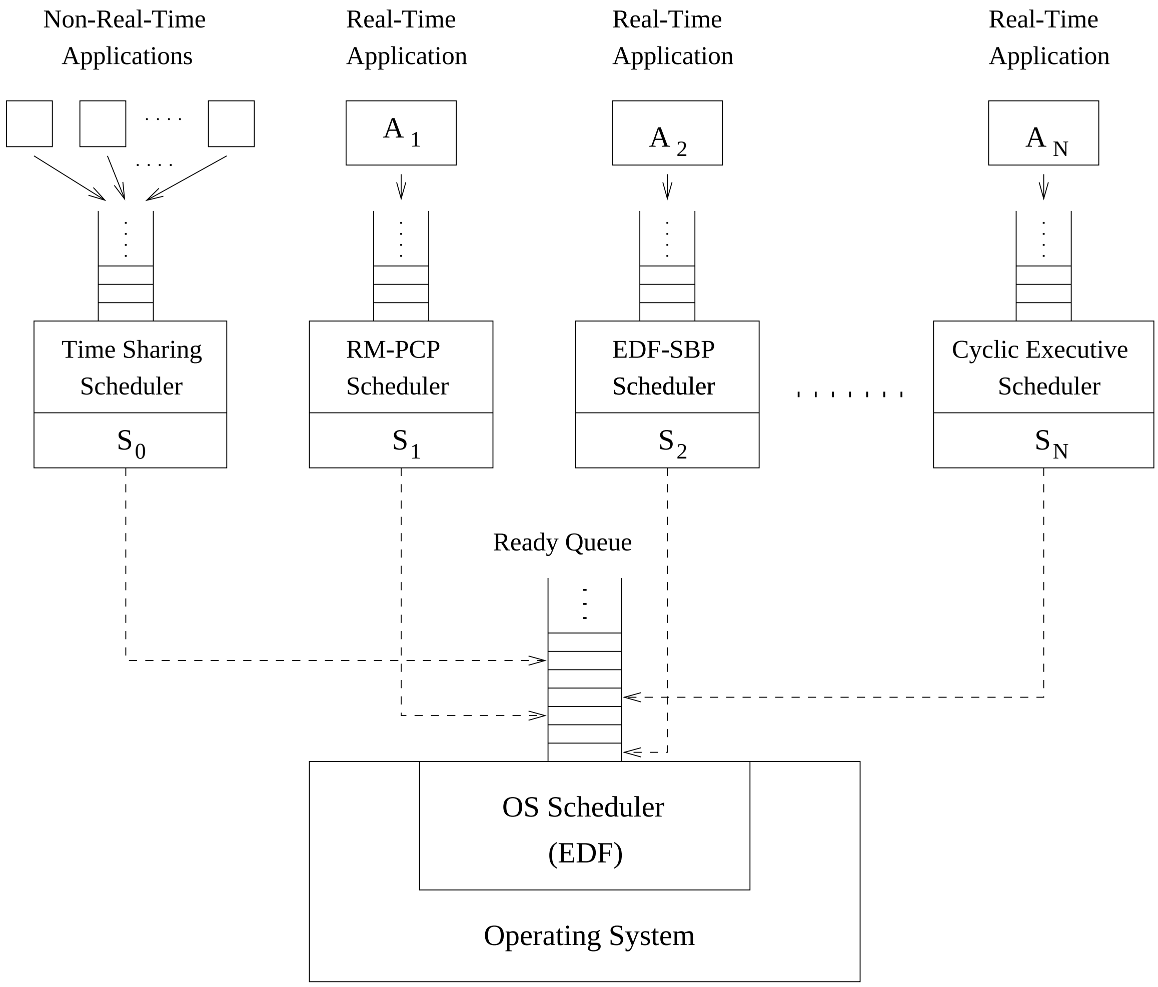


Figure 1: An Open System Architecture

3.1 Scheduling Hierarchy

In the open system, the applications are scheduled and executed according to our two-level hierarchical scheme. Specifically, all non-real-time applications are executed by a total bandwidth server which is called S_0. Each real-time application A_k is executed by a server S_k, for $k \geq 1$, which is either a constant utilization server or a total bandwidth server, depending on the characteristics of the real-time applications. The servers $S_0, S_1, \cdots, S_N$ are at the upper level (application level), as shown in Figure 1. Each server S_k, for $k \geq 1$, has a ready queue containing ready jobs of the real-time application A_k, and the server S_0's ready queue contains ready jobs of all the non-real-time applications. Each server has a scheduler associated with it, which we call *server scheduler*. The server scheduler of the server S_0 uses a time-sharing algorithm to schedule ready jobs of all non-real-time applications. The scheduling algorithm Σ_k used by the server scheduler of server S_k is the algorithm chosen for the real-time application A_k. The jobs in the ready queue of S_k are ordered according to the algorithm Σ_k. Similarly, contentions for local resources among jobs in an application are resolved according to the resource access control protocol used by the application.

At the lower level (OS level), the scheduler provided by the operating system, which we call *OS scheduler*, maintains all the servers in the system. It replenishes the server budget and sets the server deadline for every server according to the characteristics of the application(s) the server executes; Section 6 will discuss how this is done. A server is ready when its budget is nonzero and its ready queue is not empty. The OS scheduler also has a ready queue, which contains all the ready servers in the system. It schedules all the ready servers according to the EDF algorithm. Whenever a server is scheduled, it executes the job at the head of its ready queue.

Global resource contentions among applications are handled by the nonpreemptable critical section (NPS) protocol [14]. According to this protocol, whenever a job of an application A_k requests for a global resource, the request is always granted. When a job in A_k holds a global resource, it becomes nonpreemptable, and so does the server S_k, until the job releases all global resources it holds.

In summary, each scheduling decision in the open system is made in two steps. The OS scheduler first finds the server at the head of its ready queue. In turn, that server executes the job at the head of the server's ready queue. Figure 2 shows the operations of the OS scheduler. When the system starts, the OS scheduler creates the server S_0 for the non-real-time applications. The open system always admits non-real-time applications. It admits and starts a new real-time application A_k only when A_k passes an acceptance test. The new application A_k requests for admission by providing the OS scheduler the information needed to conduct the acceptance test. If the application A_k passes the test, the open system admits the application, and creates either a constant utilization server or a total bandwidth server S_k to execute A_k. We will describe in

Initiation:

- Create a total bandwidth server S_0 with size U_0 for non-real-time applications.
- Set the total server size U_t of all servers in the system to U_0.

Acceptance test and admission of A_k:

Do according to the algorithm described in Figure 5.

Maintenance of each server S_k:

Replenish the budget and set the deadline of server S_k depending on the characteristics of the application A_k and all applications executing in the open system (see Section 6).

Interaction with server scheduler of each server S_k:

- When a job J in the application A_k is released, invoke the server scheduler of S_k to insert J in S_k's ready queue.
- If the application A_k uses a preemptive scheduling algorithm, before replenishing the budget of S_k, invoke the server scheduler of S_k to update the occurrence time t_k of the next event of A_k.
- When a job J in the application A_k enters its nonpreemptable section, mark the server S_k nonpreemptable until the job J in A_k leaves its nonpreemptable section.
- When a job J in the application A_k requests for a global resource, grant J the resource and mark the server S_k nonpreemptable until the job J in A_k no longer holds any global resource.

Scheduling of all servers:

Schedule all servers on the EDF basis, except when a server S_k is marked nonpreemptable; in which case, S_k has the highest priority among all servers in the system.

Termination of a real-time application A_k:

- Destroy the server S_k.
- Decrease U_t by U_k.

Figure 2: Operations of the OS scheduler

Section 6 the information provided by the requesting application and the algorithm used by the OS scheduler to do acceptance tests and to choose the appropriate server type and server size in order to ensure the schedulability of each application. When an application A_k terminates, the OS scheduler destroys the server S_k.

When a job of a real-time application A_k is released, the OS scheduler invokes the server scheduler of the server S_k to insert the newly released job in the proper location in the server's ready queue according to the scheduling algorithm Σ_k. We assume that the time taken for inserting the newly released job into the ready queue is either negligibly small compared with the execution times of all the jobs in the system or is accounted for by including the server scheduler of S_k as a task of A_k when determining the schedulability of A_k.

Overall, the two-level hierarchical scheduler guarantees that all real-time applications are schedulable once they are accepted and all the non-real-time applications appear to be running in a slower time-sharing environment. If the server size of the server S_0 is $U_0 > 0$, the average response time of any non-real-time application is no worse than the average response time it would have in a time-sharing environment where there is no real-time application and the processor speed is U_0.

3.2 Problem Addressed and Types of Applications

The problem addressed in this paper is how the OS scheduler should choose and maintain (i.e., replenish of server budget and set server deadline) the server for each application so that the application is schedulable in the open system. To gain some insight, we consider first an application A_k that is scheduled nonpreemptively (or in a time-driven manner) and is schedulable on a processor with speed $\sigma_k < 1$. It is simple to maintain its server S_k, which is a constant utilization server of size σ_k [12]. At each replenishment time t, the OS scheduler sets the server budget to the execution time e of the job J at the head of the S_k's ready queue and the server deadline to $t + e/\sigma_k$. We have shown in [12] that for every job J in A_k, the replenishment time at which the server S_k is given budget to execute J is same as the decision time when J would be scheduled if A_k were to execute alone on a processor with speed σ_k. Moreover, if the server is schedulable, the job completes no later than the time it would complete if A_k were to execute alone on a slow processor. (By the *server being schedulable*, we mean that after each replenishment, the server budget is always exhausted at or before the server deadline.) Because the OS scheduler accepts the application A_k only when the constant utilization server S_k of size σ_k for A_k is schedulable, the application A_k is schedulable in the open system.

Unfortunately, this simple maintenance rule cannot be used for servers of preemptively scheduled applications. The example in Figure 3 explains why. The application has two jobs and uses the preemptive EDF algorithm to schedule them. The jobs are $J_1(0, a, 10a+4)$ and $J_2(a, 1, a+4)$, where a is an arbitrary positive number. (Three numbers in parenthesis next to the job name represent the release time, execution time and deadline of the job, respectively.) If this application were to execute alone on a slow processor with speed 0.25, both jobs could complete in time. On the other hand, if we were to use a constant utilization server or a total bandwidth server of size U to execute the two jobs and maintain the server in the way described above, the budget of the server and the schedule of the jobs can be as shown in Figure 3. If we want to ensure that J_2 completes by its deadline in the open system, we must have $(a+1)/U_k \leq a+4$. For arbitrary $a > 0$, this inequality holds only when U is equal to one, i.e., the application executes alone in

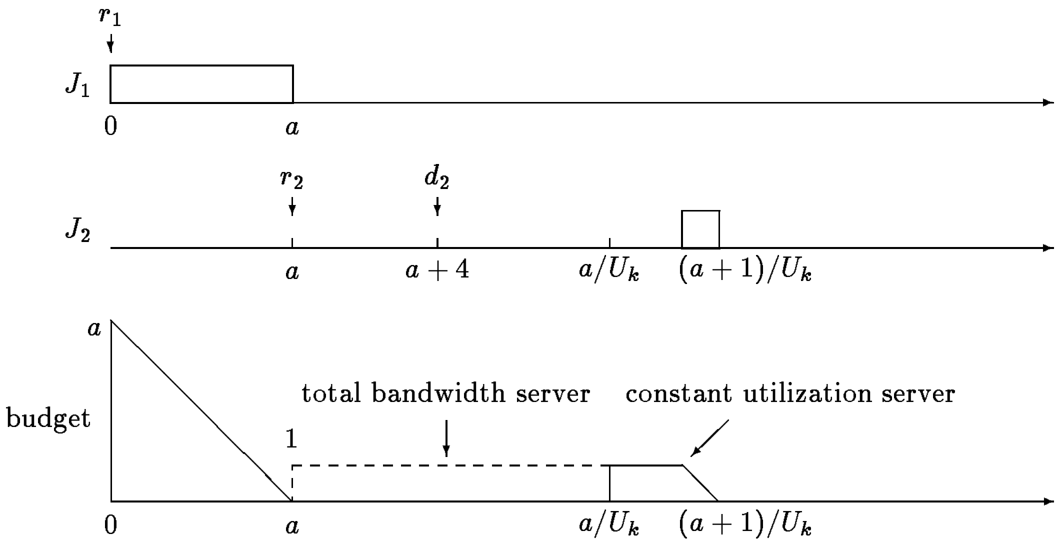


Figure 3: An Example Illustrating the Effect of Improper Server Maintenance

the open system. This, clearly, is not what we want. We note from the figure that at time 0, the server budget is set to a, the execution time of J_1. By over-replenishing the server budget, the OS scheduler allows J_1 to borrow the budget that should be saved and used later at time a to execute J_2. Thus it allows a form of priority inversion to occur.

This example gives one of the reasons that the OS scheduler chooses different types of servers and maintains the servers differently for different types of applications. For the sake of describing the algorithms used by the scheduler and showing that they work, we divide real-time applications into two broad categories: *predictable* and *nonpredictable* applications. The server scheduler of a predictable application can determine precisely at any time when the next scheduling decision for the application will be made, but the the server scheduler of a nonpredictable application cannot. Predictable applications include all applications that are scheduled in a time-driven manner and all applications in which the release times of all jobs are known *a priori*. It is not necessary for the server scheduler of nonpreemptively scheduled applications to determine when the next scheduling decision time will be. For convenience, we include the nonpreemptively scheduled application in the predictable type. Nonpredictable applications include those that have aperiodic and sporadic tasks and/or periodic tasks with release time jitters and are scheduled preemptively.

In [12], we proved the following sufficient condition under which a real-time application A_k is schedulable in the open system if no application in the open system has nonpreemptable sections or uses global resources.

Theorem 1: *A real-time application A_k that is schedulable according to some scheduling algorithm Σ_k on a slow processor with speed $\sigma_k < 1$ is also schedulable in the open system in which no application has nonpreemptable sections or uses global resources when A_k is executed by a constant utilization server S_k whose scheduler uses Σ_k, provided that all the following conditions are true.*

1. *Whenever the OS scheduler sets the budget of the server S_k, the replenished budget never exceeds the remaining execution time of the job at the head of S_k's ready queue.*
2. *During the interval (t, d) following any replenishment time t to the corresponding deadline d of the server, there would be no context switch among the jobs in the application A_k if A_k were to execute alone on the slow processor with speed σ_k.*
3. *The server S_k has server size σ_k.*
4. *The total size of all servers including S_k is equal to or less than one.*

For a predictable real-time application, it is possible to replenish the budget of its server in such a way that conditions 1 and 2 of Theorem 1 are always true, and the scheme for this type of applications that we described in [12] indeed works this way. Therefore, any predictable real-time application A_k can be made schedulable in the open system if the server S_k has server size σ_k and is schedulable. (For example, the size of the server for the application in Figure 3 should be 0.25. If at time 0, the release time a of J_2 is known, the OS scheduler can set the server budget to $a/4$ and server deadline to a so that condition 2 is satisfied. The form of priority inversion mentioned earlier cannot occur, and both jobs can complete in time.) However, for a nonpredictable real-time application, it is not always possible to meet the condition 2 of Theorem 1, since the release times of jobs are not known *a priori*. Still, as we will show in Sections 5 and 6, it is possible to maintain the server of a nonpredictable application A_k so that A_k is schedulable in the open system if we are willing to pay some penalty in processor utilization.

4 A Sufficient Conditions for the Schedulability of Servers

We now proceed to derive a sufficient schedulability condition for both predictable and nonpredictable applications in open systems. The schedulability condition is in fact two conditions: a sufficient condition under which all the servers are schedulable and a sufficient condition for any real-time application to be schedulable in the open system given that its server is schedulable. We present the former in this section and the latter in the next section.

The schedulability condition of all the servers follows directly from Lemma 2, which gives a sufficient schedulability condition for independent sporadic jobs. The jobs are preemptable but may contain nonpreemptable sections. To state the lemma, we need the following definitions and notation. We characterize each sporadic job J_i by its release time r_i, execution time e_i, deadline d_i, and the execution time L_i of its longest nonpreemptable section. The ratio $e_i/(d_i - r_i)$ is the *density* of the job J_i, and the interval $(r_i, d_i]$ is its *active interval.* J_i is said to be an *active job* at any time instant t in $(r_i, d_i]$, but is not active outside this interval. Let B_i denote the maximum execution time of the nonpreemptable sections of all jobs other than J_i, that is,

$$B_i = \max_{j \neq i}\{L_j\}.$$

B_i is the maximum blocking time the job J_i can suffer due to the nonpreemptivity of other jobs. Lemma 2 below tells us when such sporadic jobs are schedulable on the EDF basis. The proof of the lemma is in the Appendix.

Lemma 2: *A system of independent, preemptable sporadic jobs that have nonpreemptive sections is schedulable according to the EDF algorithm if at all times, the total density of all active jobs in the system is less than or equal to* $1 - \max_i\{B_i/(d_i - r_i)\}$.

Theorem 1 in [12] is for the special case when no sporadic job has nonpreemptable sections; it follows directly from the lemma. Similarly, the well-known schedulability condition of independent, preemptable periodic tasks [1] also follows straightforwardly from this lemma.

Again, we say that a constant utilization server or a total bandwidth server is *schedulable* if every time after the server budget and deadline are set, its budget is always exhausted at or before its deadline. The following theorem gives a schedulability condition for a system consisting of periodic tasks, constant utilization servers and total bandwidth servers to be schedulable according to the EDF algorithm. The theorem implies the schedulability condition of constant utilization servers and total bandwidth servers in the open system.

Theorem 3: *In a system of independent periodic tasks, whose relative deadlines are equal to their respective periods, and constant utilization servers and total bandwidth servers which execute independent aperiodic/sporadic jobs, all periodic tasks and all servers are schedulable according to the EDF algorithm if the total utilization* U_p *of all the periodic tasks and the total server size* U_s *of all the servers are such that* $U_p + U_s \leq 1 - \max_i\{B_i/\delta_i\}$ *at all times, where* B_i *is the execution time of the longest nonpreemptable section in periodic tasks and jobs executed by servers other than the periodic task* T_i *or server* S_i*, and* δ_i *is the relative deadline of the periodic task* T_i *or the shortest relative deadline of all jobs executed by the server* S_i.

To see why this theorem follows from Lemma 2, we note that each constant utilization server can be thought of as a stream of sporadic jobs, only one of which is active at any time, and the density of these sporadic jobs are all equal to the size of the server by definition. A total bandwidth server can be thought of as a constant utilization server that is allowed to execute in time intervals when no job with an earlier deadline than the server is ready to execute. Moreover, it does not consume more processor time before each server deadline than the constant utilization server. Therefore, from the point of view of schedulability, it can be thought of as a constant utilization server. (More detailed argument on this point can be found in [12].)

5 Schedulability Condition of Real-Time Applications in Open System

When no application in the system has nonpreemptable sections or uses global resources, servers of different applications can preempt each other at any time. In this case, Theorem 5 below states a sufficient condition for any real-time application to be schedulable in the open system. The proof of the theorem requires the following lemma.

Lemma 4: *If a system of n independent periodic and/or sporadic tasks T_1, T_2, $\cdots$, T_n is schedulable on a processor with speed σ according to some preemptive scheduling algorithm, it is also schedulable on a processor with speed $\alpha\sigma$ for some $\alpha > 1$ according to the same scheduling algorithm when the relative deadline D_i of every task T_i is shortened to D_i/α.*

Proof: Since the system is schedulable preemptively on a processor with speed σ, during the active interval of a job J of any task T_i, the maximum time demand t_d of the job J (i.e., the total maximum execution time of J and all jobs with equal or higher priorities than J that can possibly be active during that interval, plus any blocking time job J may experience) must be less than or equal to D_i. If the system executes on a fast processor with speed $\alpha\sigma$ for some $\alpha > 1$ according to the same scheduling algorithm, the maximum time demand of the job J during the active interval of J is $t_d/\alpha \leq D_i/\alpha$. Therefore, the tasks with the shortened relative deadline are schedulable on the fast processor. □

Theorem 5: *In an open system in which no application has nonpreemptable sections or uses global resources, a real-time application A_k consisting of n tasks that is schedulable according to some scheduling algorithm Σ_k on a slow processor with speed $\sigma_k < 1$ is also schedulable when it is executed by a constant utilization server S_k whose server scheduler uses Σ_k, provided that all the following conditions are true.*

1. *At each replenishment time, the OS scheduler never gives the server S_k more budget than the remaining execution time of the job at the head of S_k's ready queue at the time.*

2. *During the interval $(t, d - q)$ from any replenishment time t of the server S_k to q time units before the corresponding deadline d of the server, for some nonnegative constant q which is such that $t < d - q$, there would be no context switch among the jobs in the application A_k if A_k were to execute alone on the slow processor with speed σ_k.*

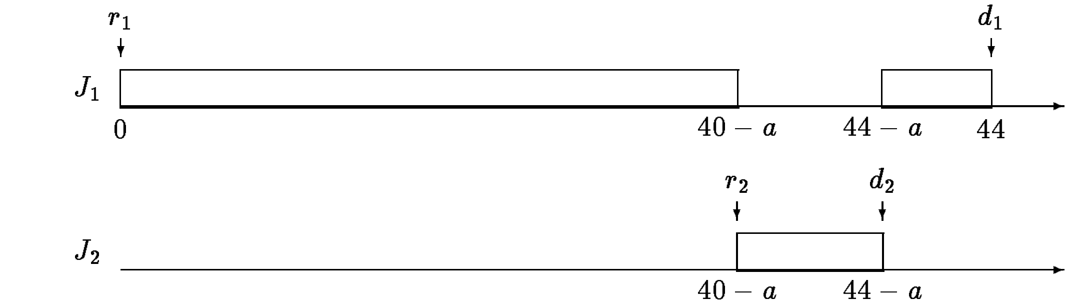
3. *The server S_k has size $U_k = max_{i=1}^{n}\{D_i/(D_i - q)\}\ \sigma_k$, where D_i is the relative deadline of i-th task in A_k.*

4. *The total size of all servers including S_k is equal to or less than one.*

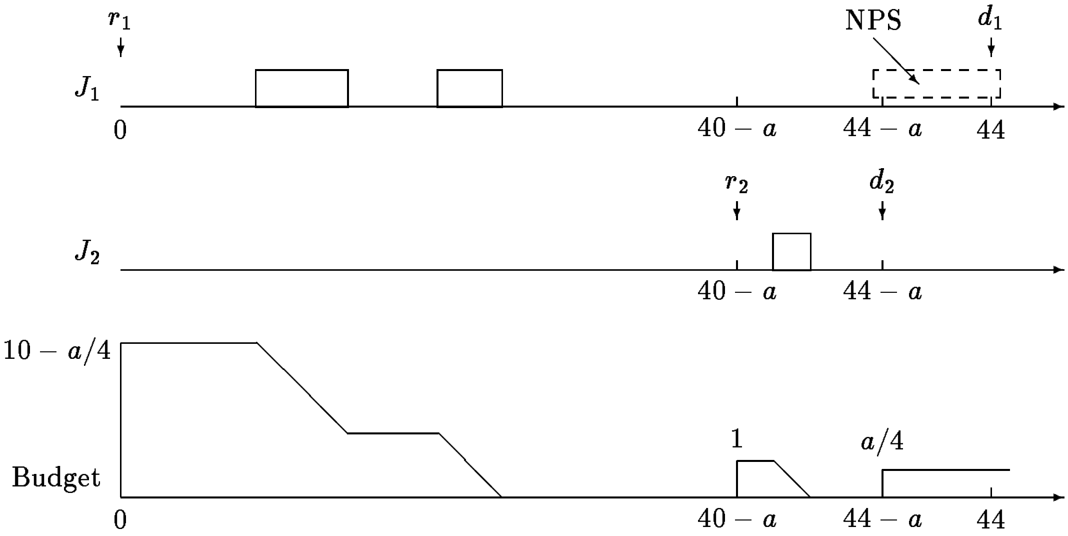
Proof: If a job with the highest priority in A_k is released (or unblocked) after the budget of the server S_k is exhausted but before the current server deadline d, the job will not be ready for execution in the open system until d when the budget of the S_k is replenished and the server becomes ready again. In effect, the ready time of the job is delayed until d. When condition 2 is true, the ready time of any job in A_k can be delayed for no more than q time units. This is equivalent to shortening the relative deadline of the job by up to q time units, or a factor of $D_i/(D_i - q)$. According to Lemma 4, the application A_k by itself would be schedulable on a processor with speed U_k given by condition 3 even if the ready time of every job in the application is delayed by up to q time units. All four conditions of Theorem 1 are satisfied if the size of server S_k is U_k and the ready time of every job released in the interval between any replenishment time and the corresponding server deadline is delayed until that server deadline. Therefore, A_k is schedulable in the open system. □

When some application in the open system has nonpreemptable sections, the execution of any jobs in one application can be delayed by the nonpreemptable sections in other applications. Clearly, this can have an adverse effect. The example in Figure 4 illustrates what the effect is. The application A_k in the example uses the EDF algorithm to schedule its two jobs, $J_1(0, 10, 44)$ and $J_2(40 - a, 1, 44 - a)$, where $0 < a < 40$. The application A_k is schedulable if it executes alone on a slow processor with speed 0.25, as shown in Figure 4(a). Now suppose that the release time of the jobs in A_k are known, the application A_k is executed by a constant utilization server S_k with server size 0.25 in the open system, and the server budget is replenished in such a way that all four conditions of Theorem 1 are satisfied. Figure 4(b) shows a possible schedule of A_k. Everything goes well until job J_2 completes and server S_k is no longer ready to execute. Before the server becomes ready again at $44 - a$, the processor executes a job J in another application whose server has a deadline later than 44, and the job J enters its nonpreemptable section right before $44 - a$. The server S_k cannot execute the remaining piece of job J_1 until the job J leaves its nonpreemptable section. If the length of the nonpreemptable section is longer than a, J_1 misses its deadline at 44. Since a can be arbitrarily small, we cannot guarantee the schedulability of the application A_k.

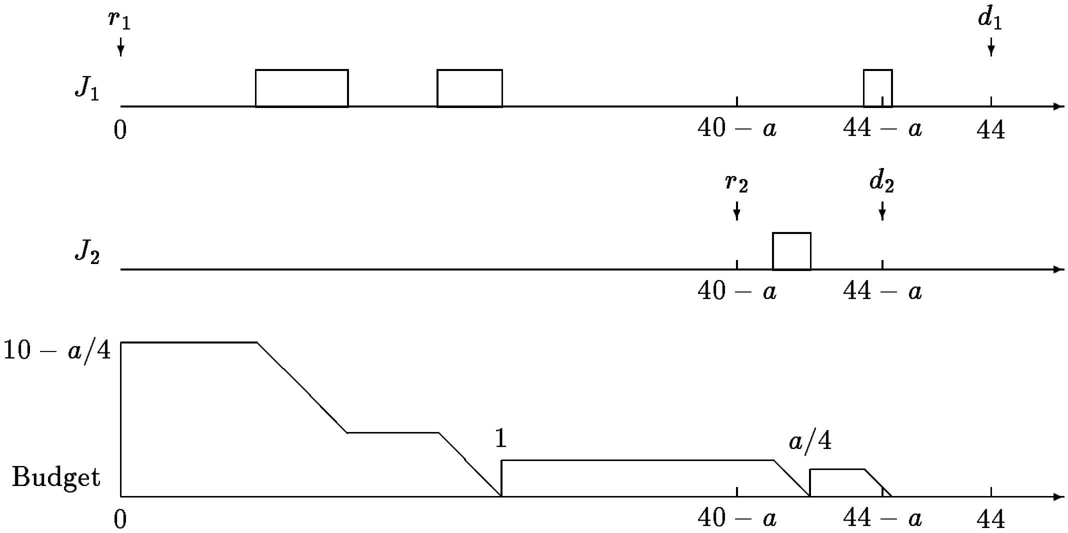
If the application A_k is executed by a total bandwidth server with server size 0.25, however, it is schedulable, as shown in Figure 4(c). When the job J_2 completes at time $44 - a$, the budget of the server S_k is replenished immediately and the deadline is set to 44. The server remains ready



(a). Slow Processor with Speed 0.25



(b). Fast Processor (executed by a constant utilization server)



(c). Fast Processor (executed by a total bandwidth server)

Figure 4: Schedules of Application A_k

to execute at this time. Consequently, the job J of the application whose server deadline is after 44 cannot execute before the job J_1 completes.

This example shows that, when some application in the open system has nonpreemptable sections, a job in an application A_k scheduled by a preemptive algorithm can potentially be blocked by a nonpreemptable section of another application each time the job is preempted if the server S_k is a constant utilization server. Consequently, the schedulability of the application A_k in the open system cannot be guaranteed. In contrast, if the server S_k is a total bandwidth server, any job of the application A_k can be blocked at most once when the job is released. In fact, a job in A_k that is released when the ready queue of the server S_k is not empty is never blocked. For this reason, if some application in the open system has nonpreemptable sections or uses global resources (which is equivalent to having nonpreemptable sections), we use total bandwidth servers to execute all the preemptive real-time applications. The following theorem gives a sufficient schedulability condition of a real-time application A_k in the open system in general. It gives us the rules for the choices and maintenance of the servers.

Theorem 6: *A real-time application A_k that is schedulable on a slow processor with speed $\sigma_k < 1$ is also schedulable in the open system when it is executed by a server S_k of size U_k provided that all the following conditions are true.*

1. *The server S_k is a total bandwidth server if A_k is scheduled by some preemptive algorithm, and is a constant utilization server if A_k is nonpreemptively scheduled (or is time-driven).*
2. *(a) If A_k is a nonpreemptive application, U_k is equal to σ_k. At each replenishment time of the server, the budget is set to the execution time of the job at the head of its ready queue.*
   (b) If the application A_k is a predictable preemptive real-time application, conditions 1, 2 and 3 of Theorem 1 are true.
   (c) If A_k is nonpredictable, conditions 1, 2 and 3 of Theorem 5 are true.
3. *The total size of all servers in the open system is no more than $1 - \max_{j\geq 1}\{B_j/\delta_j\}$, where B_j is the maximum duration of nonpreemptable sections of all applications other than application A_j and δ_j is the shortest relative deadline of all tasks in A_j.*

Proof: We first examine the case when the application A_k is scheduled by some nonpreemptive algorithm. Condition 2(a) ensures that once its server S_k becomes ready to execute the job at the head of its ready queue, it is always ready to execute until the job completes. Therefore, each job in A_k can be delayed at most once due to nonpreemptable sections of jobs in other applications for no more than B_k time units. According to Lemma 2 and Theorem 3, every job in A_k is schedulable if the total server size is no more than $1 - \max_{j\geq 1}\{B_j/\delta_j\}$.

Since the server S_k of each application A_k that is scheduled by a preemptive algorithm is a total bandwidth server, the server is always ready to execute when its ready queue is not empty. Therefore, a job can be blocked only if it is released when the server's ready queue is empty, and it can be blocked for no longer than B_k time units. The relative deadline of any job in A_k is at least δ_j time units. Therefore, according to Lemma 2 and Theorem 3, all jobs in A_k are schedulable if the total server size is no more than $1 - \max_{j \geq 1}\{B_j/\delta_j\}$. □

6 Two-Level Hierarchical Scheduling Algorithms

Earlier, Section 3 gave an overview of the two-level scheme proposed here but left out details on algorithms used for acceptance test and admission of new applications and maintenance of servers of existing applications. We are now ready to describe these algorithms. Their correctness follows directly from Theorem 6.

As we will see shortly, server schedulers must keep track of application events. An *event* of a real-time application A_k refers to one of the following:

- a job in A_k is released or completes;
- a job in A_k requests for or releases a local or a global resource; and
- a job in A_k enters or leaves a nonpreemptable section.

At any time t, the next event of A_k is the one that would have the earliest possible occurrence time at or after t if the application A_k were to execute alone on a slow processor with speed σ_k.

6.1 Acceptance Test and Admission of New Applications

Again, the open system subjects each new real-time application A_k to an acceptance test, which is described by Figure 5. If A_k passes the acceptance test, the system accepts A_k and creates a server S_k to execute A_k. Specifically, Figure 5 lists the types of information a new application A_k provides to the OS scheduler in its request for admission. In addition to the speed σ_k based on which the schedulability of A_k is validated and the algorithm Σ_k that the application uses, the new application also provides the maximum length L_k of its nonpreemptable sections if its jobs have any, as well as the shortest relative deadline δ_k of all its jobs if the application is priority-driven, or the shortest length of intervals between two consecutive events of the application if the application is time-driven. We have already mentioned these parameters in the earlier sections. Another parameter provided by the application is its jitter factor Δ_k. Δ_k is the maximum, over

When a new application A_k requests for admittance, it provides in its admission request the following information:

- the scheduling algorithm Σ_k used by A_k;
- the speed σ_k of the slow processor on which A_k is schedulable;
- the jitter factor Δ_k of A_k;
- the existence of aperiodic/sporadic tasks in A_k, if any;
- the maximum length L_k of all nonpreemptable sections or critical sections guarding global resources used by A_k; and
- the shortest relative deadline δ_k of all jobs in A_k if A_k is a priority driven application, or the shortest length δ_k between any two consecutive events of A_k if A_k is a time-driven application.

1. Find the type and the size U_k of the server S_k for A_k in the way described by in Figure 6.
2. If $U_t + U_k + \max_{1 \leq j \leq N}\{B_j/\delta_j\} > 1$, where $B_j = \max_{i \neq j}\{L_i\}$ and N is total number of applications in the system including A_k, reject A_k. Else, admit A_k, and
   - if A_k is the first application that has nonpreemptable sections or uses global resources admitted into the system, change the server type of all servers that execute preemptive applications to total bandwidth server;
   - create a server S_k of the specified type with size U_k for A_k;
   - set server budget and server deadline d to zero; and
   - increase U_t by U_k.

Figure 5: Acceptance Test and Admission of Application A_k

all the periodic tasks in A_k, the ratio of the relative deadline D_i of each periodic task T_i and the difference between the relative deadline and release time jitter $\Delta\phi_i$ of the task, that is,

$$\Delta_k = \max_{T_i \in A_k}\{\frac{D_i}{D_i - \Delta\phi_i}\}$$

Figure 6 summaries how the OS scheduler chooses the server type and calculates the server size for the new application in the step 1 of its acceptance test. If the application A_k is nonpredictable, it is impossible for the server scheduler to calculate accurately the occurrence time of the next event of A_k. However, the conditions of Theorem 5 can still be satisfied if the error in the occurrence time of the next event of A_k is no greater than q time units. (This parameter first appeared in Theorem 5 and is called the *length of scheduling quantum* in Figure 6. We will return shortly to explain this name and describe how it is used.) Therefore, the server size U_k of S_k should be at most equal to $\max_{T_i \in A_k}\{D_i/(D_i - q)\}\ \sigma_k$, which is equal to $\delta_k/(\delta_k - q)$. For the same

Server Type of S_k:

If some application (including A_k) in the system has nonpreemptable sections or uses global resources, and if A_k is scheduled by some preemptive algorithm, S_k is a total bandwidth server. Otherwise, S_k is a constant utilization server.

Server Size U_k of S_k:

If A_k is a predictable real-time application,

$$U_k = \sigma_k.$$

Otherwise,

$$U_k = \min\{\Delta_k, \delta_k/(\delta_k - q)\}\, \sigma_k,$$

where q is the length of scheduling quantum used in the open system.

Figure 6: Server Type and Server Size of S_k

reason, release time jitters of periodic tasks also make it necessary to increase the server size. If the jitter factor Δ_k is larger than $\delta_k/(\delta_k - q)$, we can treat the periodic tasks with release time jitters as sporadic tasks. This is the reason that the server size U_k is the minimum of Δ_k and $\delta_k/(\delta_k - q)$. In contrast, if the application A_k is a predictable real-time application, it is possible to calculate accurately the occurrence time of the next event of A_k at any time and to replenish the budget of the server S_k in such a way that conditions 1 and 2 of Theorem 1 are satisfied. Therefore, the server size U_k of S_k can be set to σ_k, which is the condition 3 of Theorem 1.

After determining the type and size U_k of the server S_k for the new application A_k, the OS scheduler accepts A_k if $U_t + U_k + \beta \leq 1$, where

$$\beta = \max_{1 \leq j \leq N} \{\frac{B_j}{\delta_j}\},$$

and N is the number of real-time applications including A_k in the open system. The term β in the inequality takes into account the effect of maximum blocking time that jobs of A_k may experience due to nonpreemptivity of other applications. If no application in the system has nonpreemptable sections or uses global resources, the inequality can be simplified to $U_t + U_k \leq 1$ [12]. As described in Section 5, if some application in the system has nonpreemptable sections or uses global resources, the servers of all preemptive applications must be total bandwidth servers. As long as no application in the system has nonpreemptable sections, the OS scheduler continues to use constant utilization servers for hard real-time applications in order to be more responsive for non-real-time and soft real-time applications. When the OS scheduler admits the first application that has nonpreemptable sections or uses global resources, it changes the servers of all the existing preemptive applications in the system to total bandwidth servers, as stated in Figure 5.

6.2 Server Scheduler of Nonpredictable Application

Figure 7 describes the actions taken by the OS scheduler to maintain the total bandwidth server S_k for a nonpredictable application A_k. (Descriptions for servers of predictable applications can be found in [12].) Since the precise release times of jobs in a nonpredictable application are unknown, the server scheduler of S_k is unable to obtain an accurate occurrence time of the next event of A_k. The algorithm described in Figure 8 gives an estimate. For the purpose of computing this estimate, the server scheduler maintains for each task T_j in A_k the latest release time $r_{-1}(j)$, i.e., the latest release time of all jobs in T_j prior to time t of the computation. The error in the estimate t_k is the error in the estimate of release time t' of the job that will be released first at or after time t among all jobs in A_k. The actual release time of this job is never sooner than q time units before the estimate t'. From Figures 7 and 8, we can see that the effect of using the estimate t_k computed based on the estimate t' is that the OS scheduler may give the server S_k up to qU_k units more budget than the budget computed based on the accurate occurrence time of the next event of A_k. The adverse effect of this over-budgeting is compensated for by letting the server S_k have the larger size stated in Figure 6, and thus satisfying the conditions for the schedulability of A_k in the open system.

We call q the *scheduling quantum* used by the two-level scheduler, and its value is chosen by the open system designer. Its value has no effect for a predictable application. In the other extreme, when the release times of some jobs in a preemptively scheduled application A_k are unknown, not even their ranges, the estimate t' of the next release time of some job in A_k is q units from the current replenishment time. Consequently, the amount of budget of the server S_k replenished each time is qU_k units and the deadlines of the server are q units apart. In other words, the OS scheduler maintains S_k like the server S_0 for non-real-time applications. When we know the minimum and maximum inter-arrival times of each task in the application, we may be able to get a better estimate but doing so incurs additional scheduling overhead, which is the time required to do the *for* loop in Figure 8 and has complexity $O(n)$, where n is the number of tasks in application A_k. In general, the smaller the value of q, the more frequent the server budget of S_k is replenished, and the higher the scheduling overhead. But smaller value of q also means smaller server size U_k of S_k, thus consuming less processor utilization on behalf of A_k.

Maintenance of total bandwidth server S_k whose deadline is d:

1. When a new job J_i of A_k arrives at t,

   - invoke the server scheduler of S_k to insert J_i in S_k's ready queue;
   - update the latest release time $r_{-1}(j)$ of the task T_j to which J_i belongs;
   - set J_i's remaining execution time e_i' to e_i;
   - if J_i is the only job in the ready queue,

     (a) invoke the server scheduler of S_k to estimate the occurrence time t_k of the next event of A_k as described in Figure 8,

     (b) set the server budget to $(t_k - \max\{t, d\})U_k$, and server deadline d to t_k, and

     (c) decrease the remaining execution time e_i' of J_i by $(t_k - \max\{t, d\})U_k$.

2. When a job in A_k completes,

   - if S_k's ready queue is not empty and job J_i is at the head of the ready queue,

     (a) invoke the server scheduler of S_k to estimate the occurrence time t_k of the next event of A_k as described in Figure 8,

     (b) set the server budget to $(t_k - d)U_k$, and server deadline d to t_k, and

     (c) decrease the remaining execution time e_i' of J_i by $(t_k - d)U_k$.

3. After a job J_i in A_k requests for or releases a local resource, invoke the server scheduler of S_k to change the priorities of some jobs in its ready queue if necessary and moves the job with the highest priority to the head of its ready queue.

Figure 7: Maintenance of Total Bandwidth Server S_k for a Preemptive Application A_k

Input:

n:	the number of tasks in A_k
$p^m[1 \ldots n]$:	the minimum inter-arrival time of each task in A_k
$p^M[1 \ldots n]$:	the maximum inter-arrival time of each task in A_k
$r_{-1}[1 \ldots n]$:	the latest release time of each task in A_k
q:	the length of scheduling quantum in the open system.

Calculation of the Next Event Time t_k at time t:

$t' \leftarrow \infty$;
for $(i = 1; i \leq n; i++)$ {
  $r_e \leftarrow \max\{t, r_{-1}[i] + p^m[i]\}$; /* earliest possible release time of task i */
  $r_l \leftarrow r_{-1}[i] + p^M[i]$; /* latest possible release time of task i */
  $t' \leftarrow \min\{t', \min\{r_e + q, r_l\}\}$;
}
if the ready queue of S_k is empty at time t
  $t_k \leftarrow t'$;
else {
  $J_i \leftarrow$ the job at the head of the ready queue of S_k at time t;
  $e_i'' \leftarrow$ the amount of time J_i must attain to reach the point when J_i either completes, or requests for or releases a resource, or enters or leaves a nonpreemptable section, whichever occurs the earliest;
  $t_k \leftarrow \min\{t', t + e_i''/U_k\}$;
}

Figure 8: Calculation of the Next Event Time by the Scheduler of the Server S_k of a Nonpredictable Application A_k

7 Summary

The open system architecture and the two-level hierarchical scheduling scheme described in this paper extends the scheme proposed in [12]. The extended scheme can deal with the real-time applications that have nonpreemptable sections, use global resources, have release time jitters while the original scheme cannot. With this extension, the open system architecture and the two-level hierarchical scheduling scheme can provide an isolated virtual machine that guarantees the schedulability of an arbitrary real-time application once this application is admitted into the open system.

Throughout our discussion here, we have assume that the maximum execution time of every job is known when the job is released and the job actually executes for that amount of time. In reality, a job may execute for a shorter amount of time. The system can easily reclaim the time allocated to any job executed by server S_k but unused by the job by letting the next job in S_k's ready queue execute during the remaining time if the queue is not empty. In other words, the server S_k uses the remaining budget to execute the next job and competes for processor with the current server deadline. This effectively increases the priority of the next job among the active jobs in the system. If server S_k's ready queue is empty when a job completes earlier than expected, or if it is disadvantageous to complete any job early, the OS scheduler takes back the remaining budget, thus allowing applications executed by total bandwidth servers to make use of the budget.

We are currently building a proof-of-concept prototype run-time environment and application system interfaces based on the two-level hierarchical scheme within the framework of Windows NT operating system. Specifically, we will replace the existing NT kernel scheduler by our two-level scheduler and introduce a *server* object to indirectly manage the ready *threads* (the basic scheduling entities) in the NT kernel. We will also provide a set of Real-Time Application Programming Interface (RTAPI) for the real-time applications. These applications can use RTAPI to specify their real-time attributes, for example, to register the periodic tasks in the applications.

References

[1] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment," in *J. Assoc. Comput. Mach.*, vol. 20(1), pp. 46–61, 1973.

[2] J. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Scheduling Algorithm – Exact Characterization and Average Case Behavior," in *Proceedings of the IEEE Real-Time System Symposium*, pp. 166–171, 1989.

[3] J. P. Lehoczky, L. Sha, J. K. Strosnider, and H. Tokuda, "Fixed Priority Scheduling Theory for Hard Real-Time Systems," in *Foundations of Real-Time Computing, Scheduling and Resource Management* (A. M. Tilborg and G. M. Koob, eds.), ch. 1, Kluwer Academic Publishers, 1991.

[4] J. Y.-T. Leung and J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks," *Performance Evaluation*, vol. 2, pp. 237–250, 1982.

[5] B. Sprunt, L. Sha, and J. P. Lehoczky, "Aperiodic Task Scheduling for Hard Real-Time Systems," *Real-Time Systems: The International Journal of Time-Critical Computing Systems*, vol. 1, pp. 27–60, 1989.

[6] 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.

[7] T. P. Baker, "A Stack-Based Allocation Policy for Realtime Processes," *Proceedings of IEEE 11th Real-Time Systems Symposium*, pp. 191–200, December 1990.

[8] T. M. Ghazalie and T. P. Baker, "Aperiodic Servers in a Deadline Scheduling Environment," *Real-Time Systems*, Vol. 9, No. 1, July 1995.

[9] M. Spuri and G. Buttazzo, "Scheduling Aperiodic Tasks in Dynamic Priority Systems," *Real-Time Systems*, vol. 10, pp. 179–210, 1996.

[10] L. Zhang, "VirtualClock: A New Traffic Control Algorithm for Packet-Switched Networks," *ACM Transaction on Computer Systems*, Vol. 9, No. 2, pp. 101–124, May 1991.

[11] A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm," *Proc. ACM SIGCOMM'89*, pp. 3–12.

[12] Z. Deng, J. W.-S. Liu, J. Sun, "Dynamic Scheduling of Hard Real-Time Applications in Open System Environment," Technical Report UIUCDCS-R-96-1981, Department of Computer Science, University of Illinois at Urbana-Champaign.

[13] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C. Plaxton, "A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems," *Proceedings of IEEE 17th Real-Time Systems Symposium*, pp. 288–299, December 1996.

[14] Al Mok, "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment," Ph.D. Thesis, MIT, Department of EE and CS, MIT/LCS/TR-297, May 1983.

Appendix

Proof of Lemma 1: We prove the lemma by contradiction. To do so, we suppose that a job misses its deadline at time t, and there is no missed deadline prior to t. Let t_0 be the latest time before t at which either the system idles or some job with a deadline after t executes. Suppose that during the interval $(t_0, t]$, the system executes n sporadic jobs, $J_1, J_2, \cdots, J_n$, ordered in increasing order of their deadlines. Job J_n is the one that misses its deadline. Without loss of generality, we assume that immediately prior to time t_0, the system executes some job that is in its nonpreemptable section. Let t_{-1} be the start of this nonpreemptable section. If the system executes some job that is not in its nonpreemptable section immediately prior to time t_0, we simply have t_{-1} equal to t_0.

We say that a *system event* occurs when one of the following happens: (1) a job is released or completes, (2) a job misses its deadline, (3) a job enters or leaves its nonpreemptable section. Suppose that during the interval $(t_{-1}, t]$, there are m system events, ordered in ascending order of their occurrences. Let t_i denote the time instant when the i-th event occurs, where $i = 1, 2, \cdots, m$. We must have $t_1 = t_{-1}$ and $t_m = t$. The entire interval $(t_{-1}, t]$ is partitioned into $m - 1$ disjoint subintervals, $(t_1, t_2], (t_2, t_3], \cdots, (t_{m-1}, t_m]$. By the definition of system event, in each subinterval, active jobs in the system remain unchanged, and so does the total density of all the active jobs. Let Λ_i denote the subset containing all the jobs that are active during the sub-interval $(t_i, t_{i+1}]$ for $1 \leq i \leq m - 1$ and u_i denote the total density of the jobs in Λ_i.

We note that

$$\begin{aligned}
\sum_{i=1}^{n} e_i &= \sum_{i=1}^{n} \frac{e_i}{d_i - r_i}(d_i - r_i) \\
&\leq \sum_{j=1}^{m-1} (t_{j+1} - t_j) \sum_{J_k \in \Lambda_j} \frac{e_k}{d_k - r_k} \\
&= \sum_{j=1}^{m-1} u_j (t_{j+1} - t_j)
\end{aligned}$$

Since $u_j \leq 1 - \max_i\{B_i/(d_i - r_i)\}$ for all $j = 1, 2, \cdots, m - 1$, we have

$$\begin{aligned}
\sum_{i=1}^{n} e_i &\leq \sum_{j=1}^{m-1} (t_{j+1} - t_j)(1 - \max_i\{\frac{B_i}{d_i - r_i}\}) \\
&= (t_m - t_1)(1 - \max_i\{\frac{B_i}{d_i - r_i}\}) \\
&= (t - t_{-1})(1 - \max_i\{\frac{B_i}{d_i - r_i}\}) \\
&\leq (t - t_{-1})(1 - \frac{B_n}{d_n - r_n}) \\
&= (t - t_{-1})(1 - \frac{B_n}{t - r_n})
\end{aligned}$$

At time t_{-1}, a job J with deadline later than t executes and enters its nonpreemptable section. J_n must be released after t_{-1}, that is, $r_n > t_{-1}$. Moreover, the job J remains in the nonpreemptable section at time t_0. We must have $B_n \geq t_0 - t_{-1} \geq 0$. Substituting these lower bounds into the right hand side of the expression above, we have

$$\begin{aligned}\sum_{i=1}^{n} e_i &\leq (t - t_{-1})(1 - \frac{B_n}{t - t_{-1}}) \\ &= t - t_{-1} - B_n \\ &\leq t - t_0\end{aligned}$$

However job J_n misses its deadline at time t, therefore,

$$\sum_{i=1}^{n} e_i > t - t_0,$$

which is clearly a contradiction. □