Proceedings of 9th Euromicro Workshop on Real-Time Systems

# A Scheme for Scheduling Hard Real-Time Applications in Open System Environment

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

**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. It describes a two-level hierarchical priority-driven scheme for scheduling independently developed applications. The scheme allows the developer of each real-time application to validate the schedulability of the application independently of other applications. Once a real-time application is created and accepted by the open system, its schedulability is guaranteed regardless of the behaviors of other applications that execute concurrently in the system.*

## 1 Introduction

Most existing real-time applications are implemented on stand-alone, embedded systems or on dedicated computers. Their schedulability is determined by analyzing all the applications together. With tremendous advances in hardware technologies, it is now possible to run real-time applications on fast, general purpose workstations and personal computers concurrently with non-real-time applications. A challenging problem is how to schedule an open system of complex, independently developed real-time applications and non-real-time applications. A scheduling scheme for this purpose should meet the following objectives.

1. It allows the developer of each real-time application to validate the schedulability of the tasks in the application in isolation from other applications.

2. It has a simple acceptance test according to which the operating system can determine whether to admit a new real-time application into the system without having to analyze the schedulability of all the existing applications together with the new one.

3. Once the operating system admits a real-time application into the system, it guarantees the schedulability of tasks in the application.

4. The system maintains a certain level of responsiveness for non-real-time applications.

5. It does the above without relying on fixed allocation of time/resources or fine-grain time-slicing and, consequently, is suited for applications with varying time/resource demands and stringent timing requirements.

In this paper, we describe a two-level hierarchical scheme that meets these objectives. The scheme assumes that when the operating system admits a new real-time application into the system, it creates a dedicated constant utilization server to execute the application. (We will return shortly to describe the server.) All non-real-time applications are executed by one constant utilization server. At the top level, the operating system allocates processor time to the servers, sets their deadlines, and schedules the servers according to the earliest-deadline-first (EDF) algorithm. At the low level, the scheduler of the server for each application schedules the tasks in the application according to a priority-driven algorithm chosen for the application. The scheduling algorithm for each real-time application can be either preemptive or nonpreemptive. We show here that the schedulability of any application containing arbitrary tasks can be validated independently of other applications if the application uses a nonpreemptive scheduling algorithm. If the application uses a preemptive scheduling algorithm, it can be validated independently if it consists solely of periodic tasks. Non-real-time applications are scheduled in a time-sharing fashion.

Following this introduction, Section 2 describes the system model we use in this paper and states our assumptions. It also describes the constant utilization server, which is the type of the dedicated servers we use to execute all applications. Section 3 presents a sufficient schedulability condition of the EDF algorithm when used to schedule independent, preemptable sporadic tasks in general and constant utilization servers in particular. Sections 4 presents a sufficient schedulability condition of real-time applications in the open system. Section 5 gives the algorithms which the operating system uses to maintain the servers for different types of applications so that the schedulability of each real-time

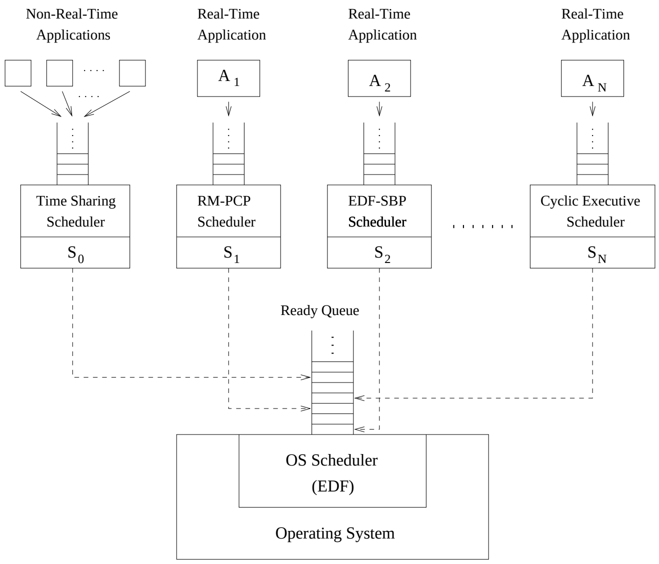


Figure 1: Open System Model

application can be determined in isolation from other applications in the system. Section 6 discusses related work, and Section 7 is a summary.

# 2 Background and Assumption

According to the model adopted in this paper, an open system has a processor with speed equal to one. The workload on the processor consists of real-time applications, denoted by $A_k$, $k = 1, 2$, and so on, and non-real-time applications. We assume that every real-time application $A_k$ would be schedulable if it were executed alone on a slow processor with speed $\sigma_k < 1$. In the open system, each real-time application $A_k$ is executed by the constant utilization server $S_k$, for $k \geq 1$, and all the non-real-time applications are executed by the constant utilization server $S_0$. As shown in Figure 1, each server has a ready queue containing application jobs that are ready to be executed by the server.

## 2.1 Constant Utilization Server

A constant utilization server is defined by its server size $U$, which is the fractional processor utilization allocated to the server. We assume that the execution time of every job in every real-time application is known after the job is released, and let the execution time of the job $J_i$ in the ready queue of a server for a real-time application be $e_i$. The execution times of jobs in non-real-time applications are unknown. These jobs are scheduled among themselves on a round-robin basis, one time slice at a time. Hence, the execution time of the job at the head of the ready queue of the server $S_0$ is equal to the length of the time slice.

Each constant utilization server becomes eligible for execution when the operating system gives it some ($> 0$) execution budget. The budget is consumed (i.e., decreased by one unit per unit of time) whenever the server executes. The server is no longer eligible for execution when its budget is exhausted (i.e., the budget becomes zero). It becomes eligible for execution again when the operating system replenishes its budget (i.e., sets its budget to some positive value again).

Specifically, the operating system replenishes the server budget and sets the server deadline of a constant utilization server of size $U$ according to the following rules. In the statement of these rules, $b$ is either equal to the execution time $e_i$ of the job $J_i$ at the head of the ready queue of the server if the scheduling algorithm of the application is nonpreemptive, or is equal to a value no greater than $e_i$ if the scheduling algorithm of the application is preemptive. We will return in Sections 4 and 5 to discuss how to choose this value in the latter case.

1. Initially, the budget of the server is zero, and the deadline $d$ is also zero.
2. When a job $J_i$ with execution time $e_i$ arrives (i.e., is released and placed in the ready queue of the server) at time $r_i$ while the ready queue is empty,
   (a) if $d \leq r_i$, set the server budget to $b$ and deadline $d$ to $r_i + b/U$;
   (b) otherwise do nothing.
3. At the deadline $d$ of the server,
   (a) if a job $J_i$ with execution time $e_i$ is waiting at the head of the ready queue, set the budget to $b$ and move the deadline to $d + b/U$;
   (b) otherwise do nothing.

The server behaves like a task with a constant utilization $U$ if its ready queue is never empty, thus the name Constant Utilization Server. This server algorithm is essentially the same as the total bandwidth server algorithm proposed by Spuri and Buttazzo [7]. (We will discuss their difference in Section 6.)

## 2.2 Scheduling Hierarchy

The applications are scheduled and executed according to a two-level hierarchical scheme. Again, at the top level, the scheduler provided by the operating system maintains all the servers. It replenishes the server budget and sets the server deadline for every server in the system, and schedules all the servers in the system according to the earliest-deadline-first (EDF) algorithm. (Hereafter, we refer to this scheduler as the OS scheduler.)

When the system starts, the operating system creates the server $S_0$ for non-real-time applications. The OS scheduler always admits non-real-time applications, but it admits a real-time application into the system only when the application meets the condition described in Section 5. When the OS scheduler admits a new real-time application $A_k$, the operating system creates a server $S_k$ with server size $U_k$ to execute $A_k$. (Section 4 will discuss the server size $U_k$ required to ensure the schedulability of the application.) When the application $A_k$ terminates, the operating system destroys the server $S_k$. We assume that the total server size of all constant utilization servers in the system is less than or equal to one at all times.

At any time, the system consists of a number of servers, as shown in Figure 1. Each server $S_k$ has a ready queue that contains ready-to-run jobs to be executed by the server. When the OS scheduler selects a server to execute, the server executes the job at the head of its ready queue. The server $S_k$ for each real-time application $A_k$ in the system also has a low-level, server scheduler, which schedules ready-to-run jobs in $A_k$ and places them in priority order in the ready queue of $S_k$. The server scheduler is a part of the application. In contrast, the operating system schedules all the non-real-time applications. The net effect is that all the non-real-time applications appear to be running in a slower time-sharing environment.

More specifically, when a job of a real-time application $A_k$ is released, the operating system invokes the server scheduler of the server $S_k$. The server scheduler then inserts the newly released job in the proper location in the server's ready queue according to the scheduling algorithm used by the server scheduler. We assume that the algorithm used by every server scheduler is a simple priority-driven algorithm. 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 as a task of $A_k$ when determining the schedulability of $A_k$.

## 3 Schedulability Condition of Sporadic Jobs With EDF Algorithm

We say that a constant utilization server is *schedulable* if every time after the server budget and deadline are set, its budget is always exhausted at or before its deadline. To state this fact in another way, we can view each server as a sporadic task in which a job with execution time equal to the server budget and deadline equal to the server deadline is released each time the server budget is replenished. The server is schedulable when every job of it completes by its deadline.

We present here a general schedulability condition that implies the schedulability condition of constant utilization servers. The general condition is for a stream of independent, preemptable sporadic jobs. Each sporadic job $J_i$ is characterized by its release time $r_i$, execution time $e_i$ and deadline $d_i$. 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*. We say that $J_i$ is an *active job* in the system at any time instant $t \in (r_i, d_i]$, but is not an active job outside this interval. Theorem 1 below gives a sufficient schedulability condition for sporadic jobs when they are scheduled on the EDF basis.

**Theorem 1:** A system of independent, preemptable sporadic jobs is schedulable according to the EDF algorithm if at any time instant, the total density of all active jobs in the system is less than or equal to one.

**Proof:** We prove the theorem 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 $t'$ 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 $(t', t]$, 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.

We call either the release of a job, or the completion of a job, or a job missing its deadline a *system event*. Suppose that during the interval $(t', t]$, there are $m$ system events, ordered in ascending order of their occurrences. Let $t_i$ denote the time instant when event $i$ occurs, where $i = 1, 2, \cdots, m$. We must have $t_1 = t'$ and $t_m = t$. The entire interval $(t', t]$ is partitioned into $m - 1$ disjoint sub-intervals, $(t_1, t_2], (t_2, t_3], \cdots, (t_{m-1}, t_m]$. By the definition of system events, in each sub-interval, active jobs in the system remain unchanged, and so does the total density of all the active jobs. Let $\Lambda_i$ 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 $\Lambda_i$.

We note that

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

Since $u_j \leq 1$ for all $j = 1, 2, \cdots, m - 1$, we have

$$\sum_{i=1}^{n} e_i \leq \sum_{j=1}^{m-1} (t_{j+1} - t_j) = t_m - t_1 = t - t'$$

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

$$\sum_{i=1}^{n} e_i > t - t'$$

which is clearly a contradiction. □

The following Corollary follows straightforwardly from Theorem 1. Our open system consists of only constant utilization servers. Because the total server size is less than or equal to one, all servers are schedulable.

**Corollary 2:** In a system of a varying number of independent, preemptable periodic tasks whose deadlines are equal to their respective periods and a varying number of constant utilization servers, 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 $U_p + U_s \leq 1$ at all times, then all periodic tasks and all servers are schedulable according to the EDF algorithm.

# 4 Schedulability Condition of Real-Time Applications

We are now ready to discuss the schedulability condition under which a real-time application $A_k$ is schedulable when it is running in an open system. As stated earlier, we assume that $A_k$ would be schedulable according to some scheduling algorithm if it were executed alone on a slow processor with speed $\sigma_k < 1$. We want to answer here the question under what condition the application $A_k$ is schedulable in the open system if its server $S_k$, working according to the constant utilization server algorithm, is schedulable.

To gain some insight, we first examine the following example. Suppose that the application $A_k$ uses the EDF algorithm to schedule its jobs, and $A_k$ has two jobs, $J_1(0, a, 10a+4)$ and $J_2(a, 1, a+4)$, where $a > 0$. (Three numbers in parenthesis represent release time, execution time and deadline of the job, respectively.) The application $A_k$ is schedulable if it executes alone on a slow processor with speed $\sigma_k$ equal to 0.25, as shown in Figure 2(a). Now suppose that the application $A_k$ is executed by the server $S_k$ with server size $U_k$ in the open system. Figure 2(b) shows a possible schedule of $A_k$. At time 0, job $J_1$ is released, the OS scheduler sets the budget of server $S_k$ to $a$ and deadline to $a/U_k$. $S_k$ may have the earliest deadline among all servers in the system during interval $(0, a)$, and it starts to execute $J_1$ immediately. At time $a$, the budget of server $S_k$ is exhausted, and job $J_1$ completes. At the same time, job $J_2$ arrives. But since the deadline of server $S_k$ is $a/U_k$, the server budget is replenished at that time. Therefore, job $J_2$ stays in the ready queue of $S_k$ waiting for the server budget to be replenished. At time $a/U_k$, the operating system sets the budget of server $S_k$ to 1 and deadline to $(a+1)/U_k$. $J_2$ is guaranteed to complete at $(a+1)/U_k$. To meet $J_2$'s deadline, we must have $(a+1)/U_k \leq a+4$, or $U_k \geq (a+1)/(a+4)$. Since $a$ is an arbitrary positive number, $J_2$ is guaranteed to meet its deadline only when $U_k = 1$. This means that no other application can be scheduled on the fast processor!

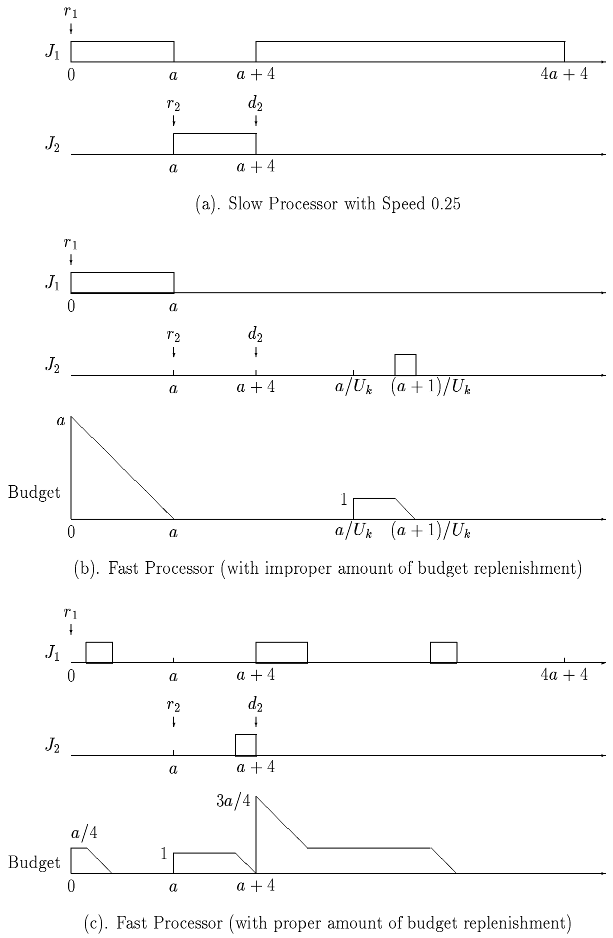


Figure 2: Schedules of Application $A_k$

This example illustrates that budget of the server $S_k$ cannot always be set to the execution time of the job at the head of $S_k$'s ready queue. Otherwise, we cannot guarantee that the application $A_k$ is schedulable in the open system even when $A_k$ is schedulable on a slower processor and the server $S_k$ is schedulable, except when the size $U_k$ of server $S_k$ is one. The following theorem states that, when the operating system replenishes the server budget in a proper manner, we can guarantee the schedulability of the application $A_k$ in the open system if the server $S_k$ for $A_k$ has size $\sigma_k < 1$ and is schedulable.

**Theorem 3:** If a real-time application $A_k$ would be schedulable according to some scheduling algorithm if it were executed alone on a slow processor with speed $\sigma_k < 1$, it is also schedulable on a fast processor with speed one when it is executed by a constant utilization server $S_k$ in the two-level scheduling hierarchy described in Section 2, provided that all the following conditions are true.

1. The server $S_k$ has server size $\sigma_k$ and is schedulable in the open system.

2. When the operating system 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.

3. During any interval $(t, d)$ between the time instant $t$ when the operating system replenishes the budget of the server $S_k$ and 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 executed alone on the slow processor with speed $\sigma_k$.

The proof of this theorem follows directly from the following lemma. Let $t_m$ denote the time when the operating system replenishes the budget of the server $S_k$ for the $m$-th time and sets its deadline to $d_m$ for $m \geq 1$. We say that a job attains $x$ units of time in an interval, or its attained time in the interval is $x$, if its remaining execution time is $x$ units less at the end of the interval than that at the beginning of the interval.

**Lemma 4:** When the conditions stated in Theorem 3 are true, the same job in $A_k$ executes and attains the same time $(d_m - t_m)\sigma_k$ on both the slow and the fast processors during any interval $(t_m, d_m)$, for all $m \geq 1$, and no other jobs in $A_k$ execute on either processor during that interval.

**Proof:** We prove the lemma by induction on the index $m$. On the fast processor, the operating system sets the budget of the server $S_k$ when the first job $J_1$ of $A_k$ arrives at $t_1$. The deadline of the server $A_k$ is set to $d_1$. According to the three conditions stated in Theorem 3, we have $e_1 \geq (d_1 - t_1)\sigma_k$, and there is no context switch among jobs in $A_k$ during the interval $(t_1, d_1)$. Since $S_k$ is schedulable, during the interval $(t_1, d_1)$, the server $S_k$ executes $J_1$ and $J_1$ attains $(d_1 - t_1)\sigma_k$ units of time on the fast processor. Similarly, if $A_k$ executes alone on the slow processor, $J_1$ also attains this amount of time in this interval. Moreover, no other jobs in $A_k$ execute on either processor during $(t_1, d_1)$.

Now suppose that during each interval $(t_m, d_m)$ for $m = 1, 2, \cdots, j$, the the same job in $A_k$ executes and attains $(d_m - t_m)\sigma_k$ units of time on both processors, and no other jobs in $A_k$ execute on either processor during the interval. At time $d_j$, every ready job $J_i$ in $A_k$ either has completed on both processors, or if not completed, has attained the same time on both processors. Therefore, on both processors at time $d_j$, either the same job in $A_k$ has the highest priority or no job in $A_k$ is ready. Let $J_x$ denote the highest priority ready job in $A_k$ at time $d_j$ or the first job in $A_k$ released after $d_j$ if there is no ready job in $A_k$ at $d_j$. Let $t'$ denote the earliest time $J_x$ is ready for execution at or after $d_j$.

The slow processor starts executing $J_x$ at $t'$. On the fast processor, the OS scheduler sets the budget and deadline of the server $S_k$ at time $t_{j+1} = t'$. The deadline is set to $d_{j+1}$. Again according to the three conditions stated in Theorem 3, we have $e'_x \geq (d_{j+1} - t_{j+1})\sigma_k$, where $e'_x$ is the remaining execution time of $J_x$, and there is no context switch among jobs in $A_k$ during the interval $(t_{j+1}, d_{j+1})$. Therefore, during the interval $(t_{j+1}, d_{j+1})$, job $J_x$ executes on the slow processor continuously and attains $(d_{j+1} - t_{j+1})\sigma_k$ units of time, and on the fast processor, the server $S_k$ executes $J_x$ and allows it to attain the same amount of time. No other jobs in $A_k$ execute on either processor during interval $(t_{j+1}, d_{j+1})$. □

Lemma 4 in essence says that when the conditions stated in Theorem 3 are true, the constant utilization server $S_k$ with size $\sigma_k$ executing on the fast processor emulates a slower processor with speed $\sigma_k$. To see why Theorem 3 follows directly, we note that during each interval $(t_m, d_m)$ for $m \geq 1$, only one job in $A_k$ executes on the slow processor and the attained time of the job is $(d_m - t_m)\sigma_k$ units. Hence, the slow processor never idles during any interval $(t_m, d_m)$, and every job in $A_k$ can complete only at the end of such an interval. According to Lemma 4, if $J_i$ executes during the intervals $(t_{m_1}, d_{m_1}), (t_{m_2}, d_{m_2}), \cdots, (t_{m_l}, d_{m_l})$ and completes at time $d_{m_l}$ on the slow processor, it also executes on the fast processor during the intervals $(t_{m_1}, d_{m_1}), (t_{m_2}, d_{m_2}), \cdots, (t_{m_l}, d_{m_l})$ and completes at or before time $d_{m_l}$ on the fast processor. Therefore, if all jobs in application $A_k$ meet their deadlines on the slow processor, they also meet their deadlines when executed by server $S_k$ on the fast processor.

We now return to the example given earlier in this section. If $A_k$ were executed on a slow processor with speed 0.25, $J_2$ would preempt $J_1$ at time $a$. Condition 3 in Theorem 3 does not hold. Indeed the application $A_k$ is not schedulable if the server budget is replenished and its deadline set as described. However, suppose that we let the OS scheduler set the server budget according to the three conditions stated in Theorem 3. At time 0, when job $J_1$ is released, the operating system sets the budget of server $S_k$ to $a/4$ and deadline to $a$. At time $a$, when job $J_2$ is released and becomes the job at the head of $S_k$'s ready queue, the server deadline just expires. The OS scheduler immediately sets the budget of $S_k$ to 1 and deadline to $a + 4$. Job $J_2$ completes at or before the server deadline at $a + 4$, thus meets its deadline. At time $a + 4$, $J_1$ becomes the job at the head of $S_k$'s ready queue with remaining execution time $3a/4$. The server budget is set to $3a/4$ and deadline is set to $4a + 4$. At or before time $4a + 4$, job $J_1$ completes. Figure 2(c) shows a possible schedule of the application.

# 5 Scheduling Algorithm for Real-Time Applications in Open System

We now describe in detail the two-level scheduling algorithm which we briefly described in Section 2. Figure 3 shows the operations of the OS scheduler. Specifically, let $U_t$ denote the total size of all the servers in the system when an application $A_k$ is created and requests for admittance into the system. Again, $A_k$ is schedulable on a slow processor with speed $\sigma_k$. The operating system admits the application into the system and creates a server $S_k$ with size $\sigma_k$ to execute the application if $U_t + \sigma_k \leq 1$. Otherwise it rejects the application. If it accepts the application, it schedules the server $S_k$ for the application together with existing servers on the EDF basis.

The ways the OS scheduler maintains the server $S_k$ and its interaction with the server scheduler depend on whether the server scheduling algorithm is preemptive or nonpreemptive and whether the jobs in the application contend for resources amongst themselves. (We assume here that applications do not contend for global resources, that is, the resources shared among jobs of different applications.) The OS scheduler replenishes the budget and sets the deadline of each server for each application so that the conditions in Theorem 3 are satisfied. It always sets the server budget to the maximum value that satisfies conditions 2 and 3 stated in Theorem 3 in order to reduce the number of times the budget and deadline of the server $S_k$ are set, thus minimizing the overall scheduling overhead. In other words, when the OS scheduler sets the budget of the server $S_k$ at time $t$, the budget is set to $\min\{e'_i, (t'-t)\sigma_k\}$, where $e'_i$ is the remaining execution time of the job $J_i$ at the head of $S_k$'s ready queue and $t'$ is the earliest possible time that a context switch could happen among jobs in application $A_k$ if $A_k$ were executed alone on the slow processor with speed $\sigma_k$.

## 5.1 Real-Time Applications with Nonpreemptive Scheduler

Figure 4 shows the actions taken by the OS scheduler to maintain the server $S_k$ for a nonpreemptive application $A_k$. The application has a stream of independent sporadic jobs $J_i, i = 1, 2, \cdots$, each of which is characterized by its release time $r_i$, execution time $e_i$, and deadline $d_i$. The release time of any job need not to be known *a priori*, but we assume that the execution time of every job in $A_k$ becomes known after it is released. The application's scheduler schedules the jobs in $A_k$ according to some nonpreemptive scheduling algorithm in such a way that $A_k$ is schedulable by itself on a slow processor with speed $\sigma_k < 1$. Whenever the server $S_k$ is scheduled, it executes the job at the head of its ready queue. The correctness of this two-level algorithm is given by Theorem 5.

**Theorem 5:** If a real-time application $A_k$ consisting of independent sporadic jobs is schedulable on a slow processor with speed $\sigma_k < 1$ by itself according to some nonpreemptive scheduling algorithm, it is also schedulable on the fast processor with speed one according to the two-level scheduling algorithm where the OS scheduler works as described in Figures 3 and 4, provided that the total server size of other existing servers is no more than $1 - \sigma_k$.

**Proof:** According Corollary 2, all the servers, including $S_k$, are schedulable. It is easy to see that all three conditions stated in Theorem 3 are true. Hence Theorem 5 is true. □

---

**Initiation:**

- Create a constant utilization server $S_0$ with size $U_0$ for non-real-time applications.
- Set the budget and deadline of server $S_0$ to infinity.
- Set the total server size $U_t$ of all servers in the system to $U_0$.

**Acceptance Test and Admission of $A_k$:**

When each new application $A_k$ requests for admittance, providing the speed $\sigma_k$ of the slow processor on which $A_k$ is schedulable in its admission request, if $U_t + \sigma_k > 1$, reject $A_k$, otherwise, admit $A_k$, and

- create a constant utilization server $S_k$ with size $\sigma_k$ for $A_k$,
- set server budget and server deadline $d$ to zero, and
- increase $U_t$ by $\sigma_k$.

**Maintenance of each server $S_k$:**

Maintain each server $S_k$ in ways described in Figures 4, 5, or 6 depending on the type of application executed by $S_k$.

**Interaction with server scheduler of each server $S_k$:**

- When every job $J_i$ in the application $A_k$ is released, invoke the server scheduler of $S_k$ to place $J_i$ in the proper location 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 application event of $A_k$.

**Scheduling of all servers:**

Schedule all servers on the EDF basis.

---

Figure 3: Operations of the OS scheduler

## 5.2 Real-Time Applications with Preemptive Scheduler

We now consider a real-time application $A_k$ that is schedulable by itself on a slow processor with speed $\sigma_k < 1$ by some preemptive scheduling algorithm (e.g., EDF, RM or DM). In this case, we require that the jobs

**Maintenance of server $S_k$:**

1. When a new job $J_i$ of $A_k$ arrives at $t$,
   (a) invoke the server scheduler of $S_k$ to place $J_i$ in the proper location in $S_k$'s ready queue, and
   (b) if the current server deadline $d \leq t$, set the server budget to $e_i$ and server deadline $d$ to $t + e_i/\sigma_k$.
2. At the deadline $d$ of the server $S_k$, if its ready queue is not empty and job $J_i$ is at the head of the ready queue, set the server budget to $e_i$ and server deadline $d$ to $d + e_i/\sigma_k$.
3. When the application $A_k$ terminates,
   (a) delete $S_k$ from the system, and
   (b) decrease $U_t$ by $\sigma_k$.

Figure 4: Maintenance of Server $S_k$ for a Nonpreemptive Application $A_k$

in each task $T_i$ in the application be released periodically. Specifically, each task $T_i$ in $A_k$ is characterized by its phase $r_i$ and period $p_i$, meaning that the $j$-th job of task $T_i$ has release time $r_i + (j-1)p_i$, and this release time is known *a priori*. However, unlike the usual periodic tasks, the jobs in each task $T_i$ may have different execution times and relative deadlines. We assume that the execution time $e_m$ of every job $J_m$ becomes known after $J_m$ is released.

Figure 5 shows the actions taken by the OS scheduler to maintain the server $S_k$ for such an application. The term *application event* in the description refers to either the release or the completion of a job in $A_k$. At any time $t$, the next application event is the application event that would have the earliest possible occurrence time after $t$ if the application $A_k$ were executed alone on the slow processor. Let $t'$ denote the earliest release time of any job of the application $A_k$ after $t$. Then at time $t$, the next application event occurs either at $t'$, if the ready queue of server $S_k$ is empty at $t$, or at $\min\{t', t + e_i'/\sigma_k\}$, if the job $J_i$ at the head of the ready queue has remaining execution time $e_i'$.

As shown in Figure 5, the maintenance of a server for a preemptive application is more complicated than that of a server for a nonpreemptive application. The added complication arises from the need for the server scheduler of each server $S_k$ to compute the occurrence time $t_k$ of the next application event in the application $A_k$ executed by $S_k$. This computation can be done in $O(N)$ time when $A_k$ contains $N$ tasks. The OS scheduler sets the budget and deadline of the server $S_k$ based on the

**Maintenance of server $S_k$:**

1. When a new job $J_i$ of $A_k$ arrives at $t$, invoke the server scheduler of $S_k$ to place $J_i$ in the proper location in $S_k$'s ready queue, and set $J_i$'s remaining execution time $e_i'$ to $e_i$. If the current server deadline $d \leq t$,
   (a) invoke the server scheduler of $S_k$ to update the occurrence time $t_k$ of the next application event of $A_k$,
   (b) set the server budget to $(t_k - t)\sigma_k$ and server deadline $d$ to $t_k$, and
   (c) decrease the remaining execution time $e_i'$ of $J_i$ by $(t_k - t)\sigma_k$.
2. At the deadline $d$ of the server $S_k$, if its 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 update the occurrence time $t_k$ of the next application event of $A_k$,
   (b) set the server budget to $(t_k - d)\sigma_k$ and server deadline $d$ to $t_k$, and
   (c) decrease the remaining execution time $e_i'$ of $J_i$ by $(t_k - d)\sigma_k$.
3. When the application $A_k$ terminates,
   (a) delete $S_k$ from the system, and
   (b) decrease $U_t$ by $\sigma_k$.

Figure 5: Maintenance of Server $S_k$ for a Preemptive Application $A_k$

occurrence time $t_k$ of the next application event. For this reason, this scheme works only when the release times of jobs are known. The correctness of this two-level algorithm for preemptive applications is given by Theorem 6, which follows straightforwardly from Corollary 2 and Theorem 3.

**Theorem 6:** If a real-time application $A_k$ consisting solely of independent tasks whose jobs are released periodically is schedulable on a slow processor with speed $\sigma_k < 1$ by itself according to some preemptive scheduling algorithm, it is also schedulable on the fast processor with speed one according to the two-level scheduling algorithm where the OS scheduler works as described in Figures 3 and 5, provided that the total server size of other existing servers is no more than $1 - \sigma_k$.

### 5.3 Resource Consideration

Oftentimes, tasks in a real-time application share logical or physical resources. In this section, we consider a preemptive application $A_k$ of tasks that share local resources. These resources are not used by tasks in applications other than $A_k$. Again we require that the jobs in each task be released periodically, so that we know their release times. Further, we assume that after a job in $A_k$ is released, we know what resources it will use, when it will request for the resources, and how long it will hold the resources. Suppose that the application $A_k$ is schedulable by some preemptive scheduling algorithm (e.g., RM, DM or EDF) and resource access protocol (e.g., PCP or SBP) when it executes alone on a slow processor with speed $\sigma_k < 1$. The question we want to answer here is how should the server for such an application $A_k$ be maintained by the OS scheduler so that the application is schedulable in the open system.

To take resource contention into consideration, the OS scheduler must react to two types of application events in addition to job releases and completions. These additional application events are requests for resource and releases of resource by jobs in $A_k$. Accordingly, the calculation of the occurrence time of the next application event after any time $t$ is changed as follows. Let $t'$ denote the earliest release time of any job of the application $A_k$ after $t$, and $J_i$ be the job at the head of the ready queue of the server $S_k$ at time $t$. Let $e_i''$ denote the amount of time $J_i$ must attain to reach the point when $J_i$ either completes, or requests for a resource or releases a resource, whichever occurs the earliest. (This can be computed when the remaining execution time $e_i'$ of the job $J_i$ is known.) At time $t$, the occurrence of the next application event is either $t'$ if the ready queue is empty, or $\min\{t', t + e_i''/\sigma_k\}$ if $J_i$ is at the head of the ready queue.

Figure 6 shows the actions taken by the OS scheduler to maintain the server $S_k$ for a preemptive application $A_k$ with local resource contention. As shown in the figure, the operations of the OS scheduler are further complicated by the need for handling resource requests by jobs in the application $A_k$. Specifically, when a job $J_i$ in the application $A_k$ requests for a resource or releases a resource, the server scheduler may need to change the priorities of some jobs in $A_k$ and sort the jobs in its ready queue according to the resource access protocol used by $A_k$. For example, if $A_k$ uses PCP algorithm, when the highest priority job $J_i$ requests for a resource which is held by job $J_j$, the server scheduler changes the priority of $J_j$ to the priority of $J_i$ and move $J_j$ to the head of its ready queue. We note that the budget of the server $S_k$ is exhausted every time a job in $A_k$ requests for a resource or releases a resource. At the deadline of the server $S_k$, the OS scheduler invokes the server scheduler to update the occurrence time $t_k$ of the next application event of $A_k$, and sets the budget and deadline of $S_k$ accordingly. Again all conditions stated in Theorem 3 are met. Hence, the following theorem stating the correctness of this two-level algorithm is true.

**Maintenance of server $S_k$:**

1. When a new job $J_i$ of $A_k$ arrives at $t$, invoke the server scheduler of $S_k$ to place $J_i$ in the proper location in $S_k$'s ready queue, and set $J_i$'s remaining execution time $e_i'$ to $e_i$. If the current server deadline $d \leq t$,
   (a) invoke the server scheduler of $S_k$ to update the occurrence time $t_k$ of the next application event of $A_k$,
   (b) set the server budget to $(t_k - t)\sigma_k$ and server deadline $d$ to $t_k$, and
   (c) decrease the remaining execution time $e_i'$ of $J_i$ by $(t_k - t)\sigma_k$.
2. At the deadline $d$ of the server $S_k$, if its 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 update the occurrence time $t_k$ of the next application event of $A_k$,
   (b) set the server budget to $(t_k - d)\sigma_k$ and server deadline $d$ to $t_k$, and
   (c) decrease the remaining execution time $e_i'$ of $J_i$ by $(t_k - d)\sigma_k$.
3. After a job $J_i$ requests for a resource or releases a resource, invoke the server scheduler to change the priorities of some jobs if necessary and move the job with the highest priority to the head of its ready queue.
4. When the application $A_k$ terminates,
   (a) delete $S_k$ from the system, and
   (b) decrease $U_t$ by $\sigma_k$.

Figure 6: Maintenance of Server $S_k$ for a Preemptive Application $A_k$ With Resource Contention

**Theorem 7:** If a real-time application $A_k$ consisting solely of independent tasks that share local resources and whose jobs are released periodically is schedulable on a slow processor with speed $\sigma_k < 1$ by itself according to some preemptive scheduling algorithm, it is also schedulable on the fast processor with speed one according to the two-level scheduling algorithm where the OS scheduler works as described in Figures 3 and 6, provided that the total server size of other existing servers is no more than $1 - \sigma_k$.

## 6 Related Work

As mentioned in Section 2, the constant utilization server algorithm is essentially the same as the total bandwidth server algorithm proposed by Spuri and Buttazzo [7]. The only difference between these two server algorithms is that according to the total bandwidth server algorithm, when a job at the head of the server's ready queue completes, the server budget is replenished immediately if the ready queue is not empty, while according to the constant utilization server algorithm, this is not done until the current server deadline. We use the constant utilization servers to execute the applications with hard deadlines in the open system. There is no benefit in completing jobs in these applications early, as long as they meet their deadlines.

The constant utilization server algorithm is also similar to the preemptive fair queueing and virtual clock algorithms proposed for network traffic scheduling [8, 9]. A processor with speed $C$ can be thought of as a communication link with link capacity $C$, and a constant utilization server $S_k$ with server size $U_k$ can be thought of as a connection with reserved bandwidth $U_k C$. The two-level hierarchical scheduling algorithm proposed in this paper can be used to schedule multiple real-time message streams on each of the connections sharing the same output link of a network switch.

## 7 Summary

This paper presented a solution to the problem of scheduling real-time applications and non-real-time applications in an open system. The solution is in the form of a two-level hierarchical algorithm. We have shown that the two-level scheme emulates the infinitesimally fine-grain, weighted round-robin algorithm. In essence, when operating system admits a real-time application which requires a fraction $\sigma$ of the processor bandwidth to meet all of the timing constraints, it provides the application with a virtual slow processor with speed $\sigma$ and protects the application from interference from other applications.

The scheduling algorithms at both levels in our two-level scheme are priority-driven. The context switch overhead of the system is equal to that incurred in a dynamic priority system, which is much smaller than that of the fine-grain round robin scheme.

To simplify our description, Figure 3 says that the budget and deadline of server $S_0$ for non-real-time applications are set to infinity. In essence, the non-real-time applications are scheduled in the background in the open system described here. We can improve the responsiveness of non-real-time applications by making $S_0$ a total bandwidth server. In that case, we would replenish its budget periodically every time slice or a few slices.

We have assumed that the OS scheduler can preempt the servers at any time. In general, the preemption of a server may be delayed because it is in a nonpreemptable section (e.g., when the application served by it is making a system call). We can account for the effect of nonpreemptivity and resource contention among applications on the schedulability of the applications in the well-known ways [4, 5].

## 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] 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.

[4] 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.

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

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

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

[8] 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.

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