monotonic algorithm and found that the average scheduling bound is usually much better than the worst case. They concluded that a good approximation to the threshold of schedulability for the rate monotonic algorithm is 88%. In fact, with the period transformation method, the utilization threshold can, in principle, be arbitrarily close to 100% [Sha 89a]. As an example of the high degree of schedulable utilization attainable with the rate monotonic algorithm, a schedulable utilization level of 99% was achieved for the Navy's Inertial Navigation System [Borger 87].

- **Stability Under Transient Overload.** Another concern for scheduling algorithms is transient overload, the case where stochastic execution times can lead to a desired utilization greater than the schedulable utilization bound of the task set. To handle transient overloads, Sha, Lehoczky, and Rajkumar describe a period transformation method for the rate monotonic algorithm that can guarantee that the deadlines of critical tasks can be met [Sha 86].
- **Aperiodic Tasks.** A real-time system often has both periodic and aperiodic tasks. Strosnider developed the *Deferrable Server* algorithm [Strosnider 88], which is compatible with the rate monotonic scheduling algorithm and provides a greatly improved average response time for soft deadline aperiodic tasks over polling or background service algorithms while still guaranteeing the deadlines of periodic tasks.
- **Resource Sharing.** Although determining the schedulability of a set of periodic tasks that use semaphores to enforce mutual exclusion has been shown to be NP-hard [Mok 83], Sha, Rajkumar, and Lehoczky [Rajkumar 89, Sha 87] have developed a priority inheritance protocol and derived a set of sufficient conditions under which a set of periodic tasks that share resources using this protocol can be scheduled using the rate monotonic algorithm.
- **Low Scheduling Overhead.** Since the rate monotonic algorithm assigns a static priority to each periodic task, the selection of which task to run is a simple function. Scheduling algorithms that dynamically assign priorities, may incur a larger overhead because task priorities have to be adjusted in addition to selecting the highest priority task to execute.

1.2.3 Scheduling Aperiodic Tasks

The scheduling problem for aperiodic tasks is very different from the scheduling problem for periodic tasks. Scheduling algorithms for aperiodic tasks must be able to guarantee the deadlines for hard deadline aperiodic tasks and provide good average response times for soft deadline aperiodic tasks even though the occurrences of the aperiodic requests are nondeterministic. The aperiodic scheduling algorithm must also accomplish these goals without compromising the hard deadlines of the periodic tasks.

Two common approaches for servicing soft deadline aperiodic requests are background processing and polling tasks. Background servicing of aperiodic requests occurs whenever the processor is idle (i.e. not executing any periodic tasks and no periodic tasks are pending). If the load of the periodic task set is high, then utilization left for background service is low, and background service opportunities are relatively infrequent. Polling consists of creating a periodic task for servicing aperiodic requests. At regular intervals, the polling task is started and services any pending aperiodic requests. However, if no aperiodic requests are pending, the polling task suspends itself until its next period and the time originally allocated for aperiodic service is not preserved for aperiodic execution but is instead used by periodic tasks. Note that if an aperiodic request occurs just after the polling task has suspended, then the aperiodic request must wait until the beginning of the next polling task period or until background processing resumes before being serviced. Even though polling tasks and background processing can provide time for servicing aperiodic requests, they have the drawback that the average wait and response times for these algorithms can be long, especially for background processing.

Figures 1-2 and 1-3 illustrate the operation of background and polling aperiodic service using the periodic task set presented in Figure 1-1. The rate monotonic algorithm is used to assign priorities to the periodic tasks yielding a higher priority for task **A**. In each of these examples, periodic tasks **A** and **B** both become ready to execute at time = 0. Figures 1-2 and 1-3 show the task execution from time = 0 until time = 20. In each of these examples, two aperiodic requests occur: the first at time = 5 and the second at time = 12.

Periodic Tasks:

	Execution Time	Period	Priority
⬜ Task A	**4**	**10**	**High**
▨ Task B	**8**	**20**	**Low**

Figure 1-1: Periodic Task Set for Figures 1-2, 1-3, 1-4, and 1-5

The response time performance of background service for the aperiodic requests shown in Figure 1-2 is poor. Since background service only occurs when the resource is idle, aperiodic service cannot begin until time = 16. The response time of the two aperiodic requests are 12 and 6 time units respectively, even though both requests each need only 1 unit of time to complete.

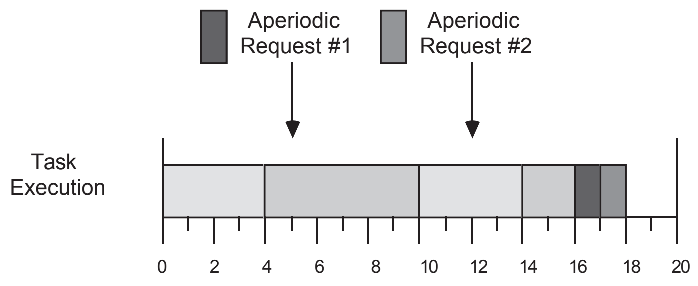

Figure 1-2: Background Aperiodic Service Example

The response time performance of polling service for the aperiodic requests shown in Figure 1-3 is better than background service for both requests. For this example, a polling server is created with an execution time of 1 time unit and a period of 5 time units which, using the rate monotonic algorithm, makes the polling server the highest priority task. The polling server's first period begins at time = 0. The lower part of Figure 1-3 shows the capacity (available execution time) of the polling server as a function of time. As can be seen from the upward arrow at time = 0 on the capacity graph, the execution time of the polling server is discarded during its first period because no aperiodic requests are pending. The beginning of the second polling period coincides with the first aperiodic request and, as such, the aperiodic request receives immediate service. However, the second aperiodic request misses the third polling period (time = 10) and must wait until the fourth polling period (time = 15) before being serviced. Also note, that since the second aperiodic request only needs half of the polling server's capacity, the remaining half is discarded because no other aperiodic tasks are pending. Thus, this example demonstrates how polling can provide an improvement in aperiodic response time performance over background service but is not always able to provide immediate service for aperiodic requests.

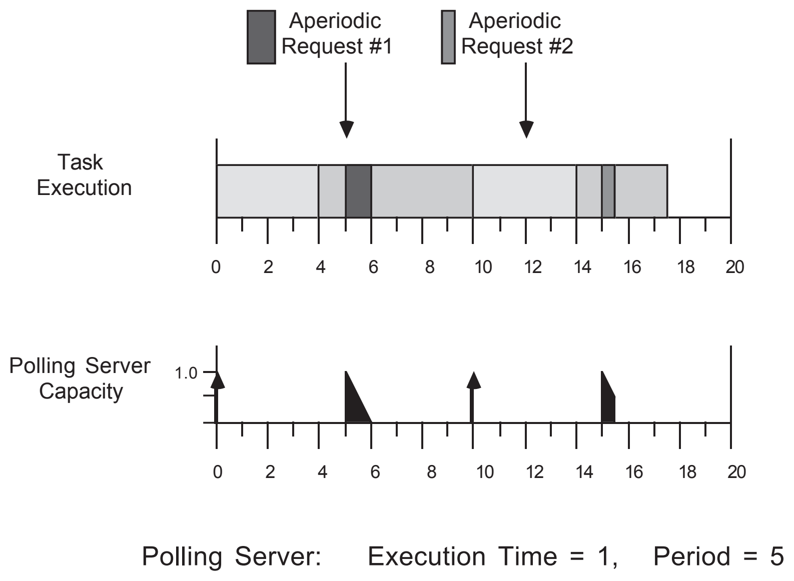

Figure 1-3: Polling Aperiodic Service Example

The *Priority Exchange* (PE) and *Deferrable Server* (DS) algorithms, introduced by Strosnider in [Strosnider 88], overcome the drawbacks associated with polling and background servicing of aperiodic requests. As with polling, the PE and DS algorithms create a periodic task (usually of a high priority) for servicing aperiodic requests. However, unlike polling, these algorithms will preserve the execution time allocated for aperiodic service if, upon the invocation of the server task, no aperiodic requests are pending. These algorithms can yield improved average response times for aperiodic requests because of their ability to provide immediate service for aperiodic tasks. In particular, the DS algorithm has been shown to be capable of providing an order of magnitude improvement in the responsiveness of asynchronous class messages for real-time token rings [Strosnider 88]. These algorithms are called *bandwidth preserving* algorithms, because they provide a mechanism for preserving the resource bandwidth allocated for aperiodic service if, upon becoming available, the bandwidth is not immediately needed. The PE and DS algorithms differ in the manner in which they preserve their high priority execution time.

The DS algorithm maintains its aperiodic execution time for the duration of the server's period. Thus, aperiodic requests can be serviced at the server's high priority at anytime as long as the server's execution time for the current period has not been exhausted. At the beginning of the DS's period, the server's high priority execution time is replenished to its full capacity.

The DS algorithm's method of bandwidth preservation is illustrated in Figure 1-4 using the periodic task set of Figure 1-1. For this example, a high priority server is created with an execution time of 0.8 time units and a period of 5 time units. At time = 0, the server's execution time is brought to its full capacity. This capacity is preserved until the first aperiodic request occurs at time = 5, at which point it is immediately serviced, exhausting the server's capacity by time = 6. At time = 10, the beginning of the third server period, the server's execution time is brought to its full capacity. At time = 12, the second aperiodic request occurs and is immediately serviced. Notice that although the second aperiodic request only consumes half the server's execution time, the remaining capacity is preserved, not discarded as in the polling example. Thus, the DS algorithm can provide better aperiodic responsiveness than polling because it preserves its execution time until it is needed by an aperiodic task.

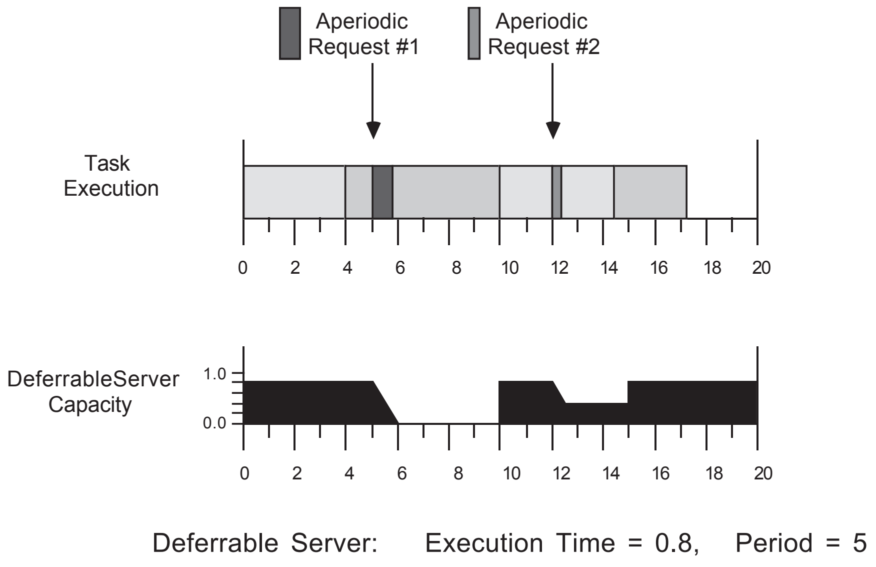

Figure 1-4: Deferrable Server Example

Unlike the DS algorithm, the PE algorithm preserves its high priority execution time by exchanging it for the execution time of a lower priority periodic task. At the beginning of the PE server's period, the server's high priority execution time is replenished to its full capacity. If the highest priority execution time available is aperiodic time (as is the case at the beginning of the PE server's period) and aperiodic tasks are pending, then the aperiodic tasks are serviced. Otherwise, the highest priority pending periodic task is chosen for execution and a priority exchange occurs. The priority exchange converts the high priority aperiodic time to aperiodic time at the assigned priority level of the periodic task. When a priority exchange occurs, the periodic task executes at the priority level of the higher priority aperiodic time, and aperiodic time is accumulated at the priority level of the periodic task. Thus, the periodic task advances its execution time, and the aperiodic time is not lost but preserved, albeit at a lower priority. Priority exchanges will continue until either the high priority aperiodic time is exhausted or an aperiodic request occurs in which case the aperiodic time is used to service the aperiodic request. Note that this exchanging of high priority aperiodic time for low priority periodic time continues until either the aperiodic time is used for aperiodic service or until the aperiodic time is degraded to the priority level of background processing (this complete degradation will occur only when no aperiodic requests arrive early enough to use the aperiodic time). Also, since the objective of the PE algorithm is to provide a low average response time for aperiodic requests, aperiodic requests win all priority ties. At all times, the PE algorithm uses the highest priority execution time available to service either periodic or aperiodic tasks.

The PE algorithm's method of bandwidth preservation is demonstrated in Figure 1-5 using the periodic task set of Figure 1-1. In this example, a high priority PE server is created with an execution time of 1 time unit and a period of 5 time units. Since the PE algorithm must manage aperiodic time across all priority levels, the capacity of the PE server as a function of time consists of three graphs: one for each priority level. The PE server's priority is priority level 1 which corresponds to the highest priority level followed by priority 2 for periodic task **A** and priority 3 for periodic task **B**. At time = 0, the PE server is brought to its full capacity, but no aperiodic tasks are pending and a priority exchange occurs between priorities 1 and 2. The PE server gains aperiodic time at priority 2 and periodic task **A** executes at priority 1. At time = 4, task **A** completes and task **B** begins. Since no aperiodic tasks are pending, another exchange takes place between priority 2 and priority 3. At time = 5, the server's execution time at priority 1 is brought to its full capacity and is used to provide immediate service for the first aperiodic request. At time = 10, the server's priority 1 execution time is brought to full capacity and is then exchanged down to priority 2. At time = 12, the server's execution time at priority 2 is used to provide immediate service for the second aperiodic request. At time = 14.5 the remaining priority 2 execution time is exchanged down to priority 3. At time = 15, the newly replenished server time at priority 1 is exchanged down to priority 3. Finally, at time = 17.5, the remaining PE server execution time at priority 3 is discarded because no tasks, periodic or aperiodic, are pending. Thus, the PE algorithm can also provide improved response times for aperiodic tasks compared to the polling algorithm. An enhancement of the PE algorithm, the Extended Priority Exchange Algorithm [Sprunt 88], has also been developed that takes advantage of the stochastic execution times of periodic tasks to provide more high priority execution time for aperiodic service.

The PE and DS algorithms differ in their complexity and in their effect upon the schedulability bound for periodic tasks. The DS algorithm is a much simpler algorithm to implement than the PE algorithm, because the DS algorithm always maintains its high priority execution time at its original priority level and never exchanges its execution time with lower priority levels as does the PE algorithm. However, the DS algorithm does pay a schedulability penalty (in terms of a lower schedulable utilization bound) for its simplicity. Both algorithms require that a certain resource utilization be reserved for high priority aperiodic service. We refer to this utilization as the server size, $\mathbf{U_s}$, which is the ratio of the server's execution time to the server's period. The server size and type (i.e. PE or DS) determine the scheduling bound for the periodic tasks, $\mathbf{U_p}$, which is the highest periodic utilization for which the rate monotonic algorithm can always schedule the periodic tasks. Below are the equations developed in [Lehoczky 87] for $\mathbf{U_p}$ in terms of $\mathbf{U_s}$ as the number of periodic tasks approaches infinity for the PE and DS algorithms:

(1)

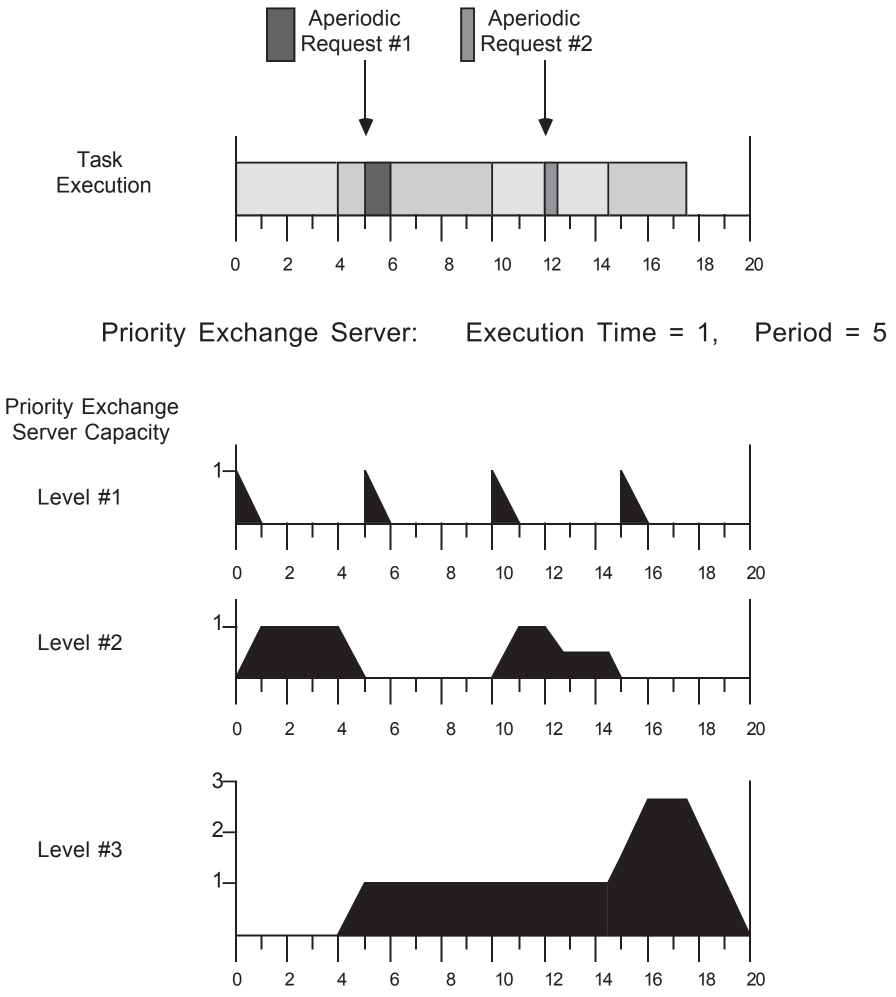

Figure 1-5: Priority Exchange Server Example

$$\textbf{PE:} \quad \mathbf{U_p} = \ln\frac{2}{\mathbf{U_s}+1}$$

$$\textbf{DS:} \quad \mathbf{U_p} = \ln\frac{\mathbf{U_s}+2}{2\mathbf{U_s}+1} \qquad (2)$$

Equations 1 and 2 show that for a given server size, $\mathbf{U_s}$ ($0 < \mathbf{U_s} < 1$), the periodic schedulability bound, $\mathbf{U_p}$, for the DS algorithm is lower than it is for the PE algorithm. These equations also imply that for a given periodic load, the server size, $\mathbf{U_s}$, for the DS algorithm is smaller than that for the PE algorithm. For example, with $\mathbf{U_p}$ = 60%, Equation 1 indicates a server size for the PE algorithm of 10% compared to a server size for the DS algorithm of 7% given by Equation 2.