6 EDF SCHEDULING FOR SHARED RESOURCES

Most scheduling algorithms are primarily concerned with cpu scheduling. When jobs are allowed to access shared resources, the accesses need to be controlled, as in any concurrent system, through the use of appropriate protocols to ensure the integrity of the resources despite potential concurrent access. The problem of accessing shared resources is well known and there is a vast literature that discusses solutions to this problem (see [8, 9, 17] for a general treatment). Solutions for the shared resource access problem usually adopt some form of semaphores [5], critical sections [2], or monitors [6].

The scheduling of accesses to resources other than the cpu is a particularly difficult problem in the context of real-time scheduling. In the presence of shared resources, the complexity of feasibility analysis becomes exponential. Specifically, the problem of deciding whether a set of periodic tasks is feasibly schedulable when semaphores are used to enforce mutual exclusion is NP-hard [11]. The difficulty arises as a result of the inability to preempt a job at arbitrary points. Suppose a job is preempted while it is accessing a resource. Clearly, the cpu can be taken away from the job and assigned to the preempting job. However, the same cannot be done for many other resources because it is quite possible that the resource state reflects partial changes done to it by the preempted job.

This chapter discusses the details of scheduling tasks which have resource constraints in addition to deadlines and periodicity constraints. For pedagogical reasons, it is assumed that shared resources are implemented by means of critical sections. To maintain the integrity of a resource, accesses to shared resources must be serialized. Hence, if a lower priority job is within a critical section and then a higher priority job tries to enter the same critical section, the higher priority job is *blocked* until the lower priority job leaves the critical section. In general, any job that needs to enter a critical section to access a resource must wait until no other job is currently within the critical section, holding the resource. Otherwise, it proceeds by entering the critical section and holds the resource. When a job leaves a critical section, the resource associated with the critical section becomes *free* and the system can then allocate it to one of the waiting jobs, if any. The protocol used to choose among waiting jobs depends on the specific algorithm used for controlling access and on the way in which priorities are assigned.

This chapter covers both priority driven algorithms as well as planning based scheduling algorithms. Specifically, Section 6.1 discusses the scheduling problem introduced by the presence of resources and Section 6.2 presents the specific issue of priority inversion resulting from the need to preserve resource consistency. Section 6.3 presents the Priority Inheritance Protocol (PIP). Section 6.4 presents the dynamic priority ceiling protocol and 6.5 briefly discuss the stack resource policy. Resource scheduling in planning mode is the subject of Section 6.6. It is interesting to note that planning mode solutions have the potential to eliminate explicit locks at runtime by scheduling tasks in such a manner as to *avoid* contention.

6.1 THE NATURE OF RESOURCES AND THE RESULTING SCHEDULING PROBLEMS

To understand the additional considerations introduced by the presence of resources, let us consider the problem of non-preemptively scheduling a set of aperiodic tasks with deadlines and resource requirements.

System resources include processors, memory, and shared data structures. In this book processing resources are referred to simply as cpus (or processors), and all the other serially reusable resources are referred to as resources.

Consider the following example: There are two processors and two copies of a resource, each of which is used only in exclusive mode. A set of jobs whose parameters are listed below is being scheduled using a dynamic priority driven approach. A job J_i's dynamic priority is denoted by Pr_i where, the smaller the value of Pr_i, the higher the priority. Assume that Pr_i is given by $(d_i + 6 * erat_i)$

Table 6.1 Job parameters for example

Job	J_1	J_2	J_3
computation time	9	10	1
resource request	either copy	either copy	both
deadline	9	74	11

where $erat_i$ is the *earliest resource available time*, i.e., the time when the resources needed by J_i will be available, given resource needs of jobs in execution and the number 6 is a weighting factor. This priority assignment function which extends EDF has good performance characteristics when planning-based approaches are used with resource constraints. The weighting factor 6 used in the priority function is chosen purely for the purposes of illustration.

The schedule produced by a greedy priority driven approach is first determined. When a processor is idle, the highest priority job which does not violate the resource constraints is assigned to the processor.

Initially, job priorities are $Pr_1 = 9$, $Pr_2 = 74$ and $Pr_3 = 11$. So J_1 has the highest priority and it is scheduled to start at time=0. Then, because one processor is still idle, another job that can start at time=0 is found. Recomputing the priorities of the remaining jobs gives $Pr_2 = 74$ and $Pr_3 = 65$. Although J_3 has the higher priority, only J_2 can start at time=0 and so it is chosen. Finally, J_3 is scheduled to start at time=10 when both copies of the resource are available. Thus jobs are scheduled according to their priority, but the algorithm is greedy about keeping the resources fully used.

Now suppose a pure priority-driven approach, one that is not greedy, is used instead. After J_1 is scheduled, the remaining job priorities are recomputed and J_3 is chosen to be executed next at time=9, followed by J_2 at time=10.

Thus, with greed, the higher priority job J_3 is delayed by one time unit while without greed, J_2 is delayed by 10 time units.

The example shows that if the resources are to be better utilized the execution of higher priority jobs may have to be delayed. This results in a form of *priority inversion* since a higher priority job is made to wait for a resource held by a lower priority job. Since the priority of a job reflects its time constraints and other characteristics of importance, it is usually desirable in real-time systems to place more emphasis on priorities than on the underutilization of resources.

The above discussion assumed that the tasks were non-preemptable. The best of both worlds can be achieved by adopting preemptive priority-driven scheduling. If this is done, then when J_1 completes execution, J_2 could have been preempted by J_3. Unfortunately, the decision to preempt may not be simple.

- There may be jobs which, once preempted, need to be restarted, losing all the computation up to the point of preemption. For example, in a communicating job, if a communication is interrupted it may have to be restarted from the beginning: the communication line represents an exclusive resource that is required for the complete duration of the job.
- A job that is preempted while *reading* a shared data structure can resume from the point of preemption provided the data structure has not been modified.
- A job that is preempted while *modifying* a shared data structure may leave it in an internally inconsistent state; one way to restore consistency is to wait for the (to be) preempted job to complete before allowing further use of the resource. An alternative is to rollback the changes made by the preempted job but, in general it is difficult to keep a record of all such changes. A rollback can add considerably to the overhead and have subsequent impact on meeting job deadlines.

Returning to our example, J_2 uses the resource in exclusive mode. So there are two possible ways in which priority driven scheduling can proceed, depending on the nature of the resource.

1. If the resource is like the communication line, J_2 can be preempted at time=9 and J_3 can begin using it immediately. This is equivalent to not having started execution of J_2 at all at time 0, so allowing J_2 to execute ahead of its turn by being greedy has not helped. But, if J_2's computation time is less than or equal to that of J_1, greed can be used. In any case, the execution of J_3 is not any further delayed than for pure priority-driven scheduling.
2. If the resource is a modifiable data structure, J_3's execution is delayed, either by the need to rollback J_2's changes or to wait for J_2 to complete execution. In either case, J_3 completes later than under pure priority-driven scheduling.

This suggests that a limited form of greed can be used in which job computation times and the nature of the resources, as well as their use, are considered when making scheduling decisions. The goal is then to ensure as much as possible that priorities are not violated or to limit the duration of priority inversion. Before describing protocols that aim to achieve this desire, the priority inversion problem is discussed in further detail.

6.2 THE PRIORITY INVERSION PROBLEM

In systems with a preemptive priority driven scheduling mechanism, a high priority job must be able to preempt a lower priority job as soon as it is ready for execution. This has been the assumption in the previous chapters. When shared resources are used, this assumption is no longer valid because if a lower priority job holds a resource and then a higher priority job tries to hold the same resource, the higher priority job is blocked until the lower priority job releases that resource. When such priority inversion occurs, a high priority job is subjected to unnecessary ***blocking***, since it must wait for the processing of a lower priority job. This blocking increases its response time and must be explicitly considered in analyzing the feasibility of a schedule since it violates the spirit of a priority-driven approach.

Moreover, the blocking time cannot be easily bounded if a specific protocol is not used. Consider the situation of Figure 6.1. Here the schedule of three preemptable jobs J_1, J_2 and J_3 is depicted when $Pr_i = d_i$, i.e., EDF is used. Here J_1 and J_3 use a shared resource, access to which is controlled by a critical section. At time $t = 3$, J_3 enters the system and begins its execution. At time $t = 5$, when J_3 is in the middle of the critical section, J_1 enters the system. Since the deadline of J_1 is earlier, a preemption occurs and J_1 starts its execution. When it needs to enter its critical section at time $t = 7$, J_1 finds that the associated resource is held by J_3. Thus, it blocks and releases the processor. Meanwhile, a third job J_2 enters the system with an earlier deadline than J_3. As per the EDF algorithm J_2 gets the processor and executes. At this point J_1 must also wait for the execution of a medium priority job. The presence of many such medium priority jobs can lead to uncontrolled blocking. In our example J_2 completes at time $t = 11$. Then J_3 is allowed to exit its critical section and at time $t = 12$ J_1 can finally resume its execution. But it is too late and it misses its deadline.

Several solutions have been proposed to deal with the problem of scheduling tasks accessing shared resources. One approach is to use a simple scheduling

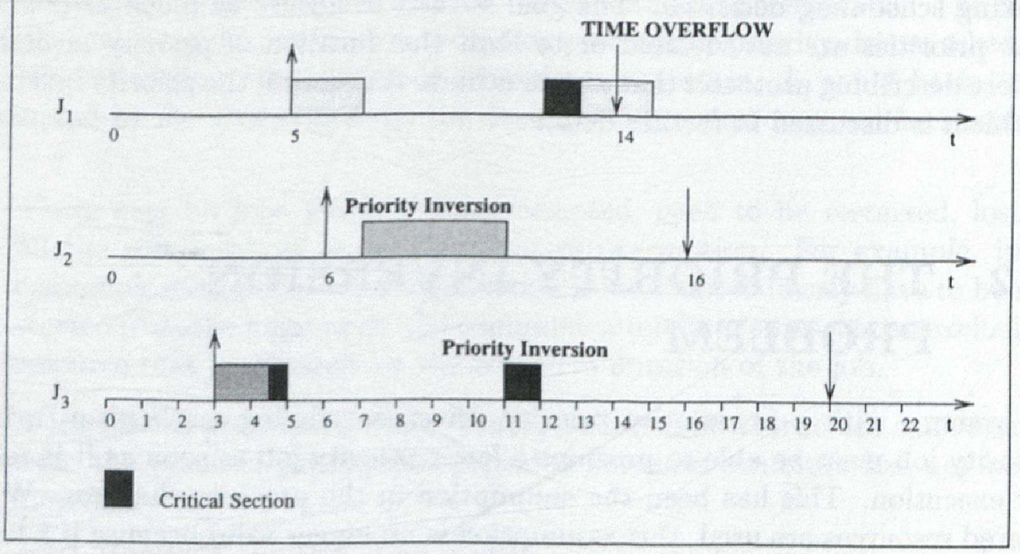

Figure 6.1 Priority inversion under EDF scheduling.

algorithm, such as one based on priorities, but embellished to control access to critical sections. Essentially, these embellishments determine when preemption can take place and who can preempt whom. The Priority Inheritance Protocol and the Priority Ceiling Protocols were developed for fixed priority systems [18]. They have been extended for EDF in [19] and in [3], respectively. In [1] Baker describes the Stack Resource Policy, a protocol suitable both for static and dynamic priority systems. The priority inheritance protocol, the dynamic priority ceiling protocol, and the stack resource protocols are described in Sections 6.3 through 6.5, respectively.

In his thesis [11], Mok proposed the *kernelized monitor*, in which the processor is assigned in time quanta of fixed length equal to the size of the largest critical section. This essentially *schedules the resource problem away* because it schedules job executions such that when one job is holding a resource, no other job can. In a way, jobs do not have to use any special protocol to ensure correct access to the resources since the way they are scheduled guarantees access correctness.

However, in a multi-processor system this is insufficient since when one job is in execution using a resource, another, needing the same resource, could also be in execution on a different processor. Hence, the multi-processor schedule must be explicitly constructed so as to exclude, in time, jobs that need the same resource for mutually exclusive use. This is what is done in the planning-based

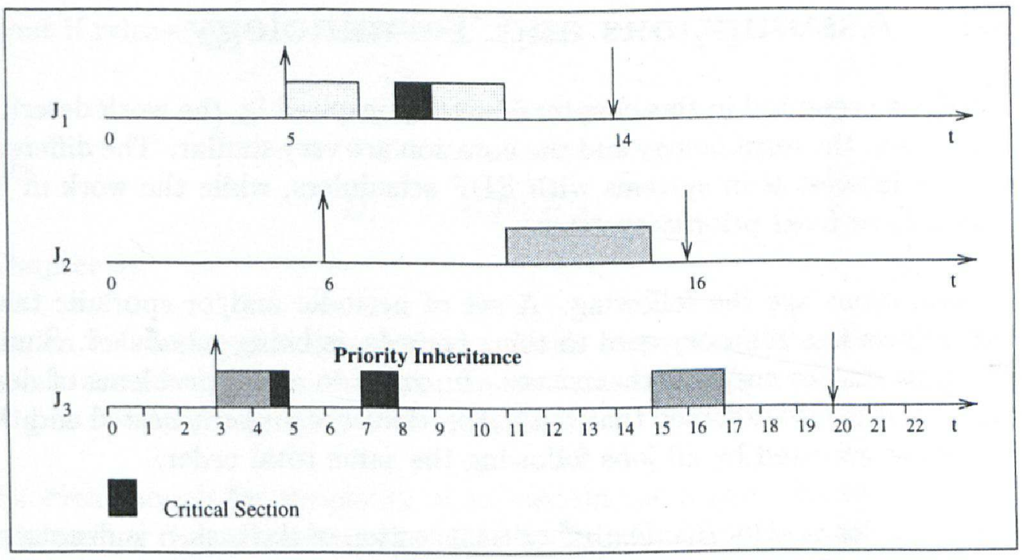
J1
0
5
14
t
J2
0
6
16
t
Priority Inheritance
J3
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 t
Critical Section

Figure 6.2 Priority inheritance under EDF scheduling.

approaches to scheduling as exemplified by the algorithm used in the Spring multi-processor kernel [14]. This algorithm is discussed in Section 6.6.

6.3 THE PRIORITY INHERITANCE PROTOCOL

The basic idea behind the Priority Inheritance Protocol (PIP) is that when a job blocks one or more higher priority jobs, it temporarily assumes the highest priority of the blocked jobs, that is, it *inherits* a higher priority. When the job exits its critical section, it resumes the original priority it had when it entered.

For instance, if the three jobs of Figure 6.1 are scheduled in this way the schedule depicted in Figure 6.2 is obtained. When the highest priority job J_1 needs to enter its critical section at time $t = 7$, J_3 inherits the priority of J_1. That is, within its critical section J_3 executes with a priority corresponding to the deadline of J_1, which is the earliest in the system. Consequently, J_2 cannot preempt J_3, as happened in Figure 6.1. As soon as J_3 exits its critical section and releases the resource, it resumes its original priority and J_1 gets both the resource and the cpu. In this way J_1 completes at time $t = 11$, J_2 at time $t = 15$ and J_3 at time $t = 17$. All the completion times being earlier than the corresponding deadlines, the schedule is feasible.

6.3.1 Assumptions and Terminology

The analysis presented in this chapter is mainly inspired by the work described in [18]. Hence, the terminology and the notation are very similar. The difference is that our interest is in systems with EDF schedulers, while the work in [18] is applicable to fixed priority systems.

The assumptions are the following. A set of periodic and/or sporadic tasks, with deadlines less than or equal to their periods, is being scheduled. Similar to [18], jobs do not suspend themselves. In order to avoid problems of deadlocked jobs, it is also assumed that critical sections are *properly* nested and that resources are accessed by all jobs following the same total order.

A resource is denoted by R_i. The j^{th} critical section of the task τ_i is denoted by $z_{i,j}$. The resource associated with $z_{i,j}$ is denoted by $R_{i,j}$. $z_{i,j} \subset z_{i,k}$ indicates that $z_{i,j}$ is entirely contained in $z_{i,k}$. Finally, the computation time of the critical section $z_{i,j}$ is denoted by $c_{i,j}$.

The assumption that critical sections are properly nested means that 1) given any pair $z_{i,j}$ and $z_{i,k}$, then either $z_{i,j} \subset z_{i,k}$, $z_{i,k} \subset z_{i,j}$, or $z_{i,j} \cap z_{i,k} = \emptyset$, and 2) the order in which the resources associated with the nested critical sections are freed is the opposite of the order in which they are acquired.

A job J is said to be *blocked* by the critical section $z_{i,j}$ of job $J_{i,h}$ (a job of τ_i) if J waits for $J_{i,h}$ to exit $z_{i,j}$ in order to continue execution, and $J_{i,h}$ has a later deadline than J. Furthermore, J is said to be blocked due to resource R, if the critical section $z_{i,j}$ blocks J and $R_{i,j} = R$.

In the description that follows the concept of *preemption levels*, originally defined by Baker in [1] is useful. The preemption level of a task τ_i is denoted by π_i. The essential property of preemption levels is that a job $J_{i,h}$ is not allowed to preempt another job $J_{j,k}$ unless $\pi_i > \pi_j$. In particular:

(6.3.1.1) if $J_{i,h}$ has higher priority than $J_{j,k}$ and arrives later, then $J_{i,h}$ must have a higher preemption level than $J_{j,k}$.

In a system with EDF scheduling, the property just mentioned is satisfied if decreasing preemption levels are assigned to tasks with increasing relative deadlines. That is:

$$\pi_i > \pi_j \quad \Leftrightarrow \quad D_i < D_j.$$

Note that if release jitter is considered, then

$$D_i < D_j$$

becomes

$$D_i - J_i < D_j - J_j$$

(see Chapter 3).

The reason for distinguishing preemption levels from priorities is that preemption levels are fixed values that can be used to statically evaluate potential blocking in dynamic priority driven systems [1].

Finally, even though for simplicity of exposition the access protocols are presented assuming that all resource accesses occur in a mutually exclusive fashion, one can envisage extensions to the protocols that can deal with resources that are used in shared mode as well.

6.3.2 Definition of the Priority Inheritance Protocol

The Priority Inheritance Protocol for EDF is defined in the following way [19]:

- When a job $J_{i,h}$ tries to enter a critical section $z_{i,j}$ and the resource $R_{i,j}$ is already held by a lower priority job $J_{j,k}$, $J_{i,h}$ waits and $J_{j,k}$ inherits the priority of $J_{i,h}$.
- The queue of jobs waiting for a resource is ordered by decreasing priority.
- Priority inheritance is transitive. That is, if a job J_3 blocks a job J_2, and J_2 blocks a job J_1, then J_3 inherits the priority of J_1 via J_2.
- At any time, the priority at which a critical section is executed is always equal to the highest priority of the jobs that are currently blocked on it.
- When a job exits a critical section it usually resumes the priority it had when it entered the critical section. The exception occurs when a new higher priority job is blocked for the critical section that contains the critical section exited by a job.
- When released, a resource is granted to the highest priority job, if any, waiting for it.

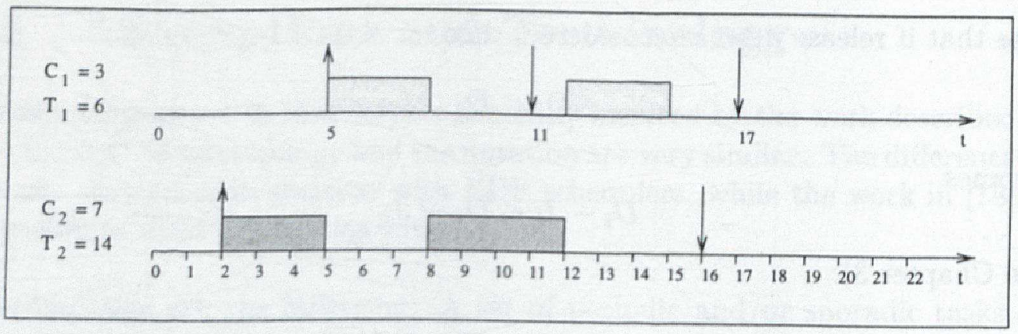

Figure 6.3 Different relative priorities under EDF scheduling.

In a manner similar to [18], in the analysis of the protocol, two types of blocking can be identified:

1. *direct* blocking, which occurs when a higher priority job tries to acquire a resource already held by a lower priority job.
2. *push-through* blocking, which occurs when a medium priority job is blocked by a lower priority job that has inherited a higher priority from a job it directly blocks.

6.3.3 Properties of the Priority Inheritance Protocol

There are only a few differences in the properties of the Priority Inheritance Protocol implemented under a fixed priority scheduling and under EDF scheduling.

In fixed priority systems the analysis is simplified by the fact that the relative priority of task instances does not change, since it is fixed for all jobs of a task. However, when an EDF scheduling mechanism is used, this is not true. In fact, at a certain time the current job of a task may have higher priority than the current job of a second task, but at a different time in the schedule the situation may be reversed with two other jobs of the same tasks. This is illustrated in Figure 6.3. At time $t = 5$, job $J_{1,1}$ is released and having the earliest deadline in the system it preempts $J_{2,1}$. However, at time $t = 11$, when the second job of τ_1 is released, preemption does not occur, because $J_{1,2}$ has a later deadline than $J_{2,1}$.

Even though it appears from the previous observation that when evaluating the possible worst case blocking for the jobs of any single task, the critical sections of all other tasks must also be considered, this is not necessary. In fact, looking at the example of Figure 6.3, even if $J_{1,2}$ has a lower priority than $J_{2,1}$, it cannot block $J_{2,1}$, since it cannot run before the completion of $J_{2,1}$. That is, jobs of τ_1 can be blocked by jobs of τ_2, but not the other way around which is a property similar to that found in fixed priority systems. The claim is more formally stated in the following two lemmas.

Lemma 6.1 *A job J_H can be blocked by a lower priority job J_L,* only if *J_L is within a critical section that can block J_H, when J_H is released.*

Proof. Suppose that when J_H is released J_L is not within a critical section which can directly block J_H or can lead to the inheritance of a priority higher than J_H. Then J_L can be preempted by J_H and it does not execute again until J_H completes, that is, it can never block J_H. □

Lemma 6.2 *A job $J_{i,h}$ of the task τ_i can be blocked by a lower priority job $J_{j,k}$ of the task τ_j, only if τ_i has a greater preemption level than τ_j, that is,* only if *$\pi_i > \pi_j$.*

Proof. By Lemma 6.1, $J_{i,h}$ can be blocked by $J_{j,k}$ only if $J_{j,k}$ is within a critical section when $J_{i,h}$ is released. Hence, $J_{j,k}$ must have been released earlier. By condition 6.3.1.1, $J_{i,h}$ must have a higher preemption level than $J_{j,k}$. □

Similar to [18], $\beta_{i,j}$ denotes the set of all critical sections of jobs of τ_j that can block jobs of the task τ_i. That is,

$$\beta_{i,j} = \{z_{j,k} : \pi_j < \pi_i \text{ and } z_{j,k} \text{ can block } J_{i,h}\}.$$

Given the assumption that critical sections are properly nested, the set $\beta_{i,j}$ is partially ordered by inclusion. The following concentrates only on the maximal elements of $\beta_{i,j}$, denoted with $\beta^*_{i,j}$.

It can now be proven that each job of τ_i may be blocked at most by one critical section of $\beta^*_{i,j}$.

Lemma 6.3 *A job $J_{i,h}$ can be blocked by a lower priority job $J_{j,k}$ for at most the duration of one critical section of $\beta^*_{i,j}$.*

Proof. By Lemma 6.1 and Lemma 6.2, in order to block $J_{i,h}$, $J_{j,k}$ must be within a critical section of $\beta^*_{i,j}$ when $J_{i,h}$ is released. Once $J_{j,k}$ exits this critical section it can be preempted by $J_{i,h}$, which later cannot be blocked by $J_{j,k}$ again. □

This property enables the computation of an upper bound on the possible blocking time experienced by any job in the schedule.

Theorem 6.1 *Under PIP and EDF scheduling, each job of a task τ_i can be blocked by at most the duration of one critical section in each of $\beta^*_{i,j}$, $1 \leq j \leq n$ and $\pi_i > \pi_j$.*

Proof. The theorem follows immediately from Lemma 6.3 and Lemma 6.2. □

The argument of Lemma 6.3 can be applied to any critical section that can block a job. Intuitively, any resource accessed by a job can cause at most one priority inversion, that is, a single blocking. If this claim is expanded, an upper bound can be found for the number of blockings a single job experiences.

Following the notation of [18], $\zeta^*_{i,j,k}$ denotes the set of all longest critical sections of τ_j associated with resource R_k which can block τ_i's jobs, either directly or via push-through blocking. That is,

$$\zeta^*_{i,j,k} = \{z_{j,h} : z_{j,h} \in \beta^*_{i,j} \text{ and } R_{j,h} = R_k\}.$$

The set of all longest critical sections associated with resource R_k that can block τ_i's jobs is then:

$$\zeta^*_{i,\cdot,k} = \bigcup_{\pi_j < \pi_i} \zeta^*_{i,j,k}.$$

The goal is to prove that each job of τ_i can be blocked by at most one critical section in $\zeta^*_{i,\cdot,k}$.

Theorem 6.2 *A job of τ_i can be blocked by at most one critical section in $\zeta^*_{i,\cdot,k}$, for each resource R_k.*

Proof. By Lemma 6.1 and Lemma 6.2, a job $J_{i,h}$ can be blocked by a lower priority job $J_{j,p}$ only if $\pi_j < \pi_i$ and $J_{j,p}$ is in the middle of a critical section of $\beta^*_{i,j}$ when $J_{i,h}$ is released. Without loss of generality, let us assume that the resource associated with the critical section is R_k. Hence $J_{j,p}$ holds R_k. Once $J_{j,p}$ exits its critical section, R_k is granted to $J_{i,h}$ in case of direct blocking, or to a higher priority job in case of push-through blocking. In either cases, $J_{i,h}$ can no longer be blocked by $J_{j,p}$, nor by other lower priority jobs with critical sections associated with R_k. □

Corollary 6.1 *Under the priority inheritance protocol and EDF scheduling, if there are m resources that can block the jobs of a task τ_i, each of these jobs can experience a maximum blocking time:*

$$B_i = \min\left(\sum_{\pi_j < \pi_i} \max\left\{c_{j,p} : z_{j,p} \in \beta^*_{i,j}\right\}, \sum_{k=1}^{m} \max\left\{c_{j,p} : z_{j,p} \in \zeta^*_{i,\cdot,k}\right\} \right). \quad (6.1)$$

Proof. It follows immediately from Theorem 6.1 and Theorem 6.2. □

6.3.4 Computation of Blocking Times

The evaluation of equation (6.1) is not as trivial as it might look. The determination of the sets $\beta^*_{i,j}$ can be done with a relatively simple procedure. However, the procedure must be carefully designed in order to take the correct critical sections into account. Once these sets are available, the computation of the sets $\zeta^*_{i,\cdot,k}$ is straightforward.

The identification of all critical sections in $\beta^*_{i,j}$ is made difficult by the existence of *push-through* blocking and *transitive* inheritance. Critical sections that cause direct blocking are easily identified, while those that can cause push-through or transitive blocking require a deeper analysis.

Transitive blocking can be caused by nested critical sections. Consider the example of Figure 6.4. There are three jobs in the schedule, J_1, J_2 and J_3, respectively, with decreasing preemption levels. J_1 accesses resource R_2. J_2 accesses two resources in a nested fashion, first R_2, and then R_1. Finally, J_3 accesses R_1. When the job J_1 is released at time $t = 8$, J_2 already holds R_2 and waits for R_1, previously held by J_3. At time $t = 10$, J_1 tries to hold R_2 and it blocks. The priority of J_1 is then transitively inherited by J_3, until it exits its critical section and releases R_1, which is immediately granted to J_2. This job has inherited the priority of J_1 through direct blocking, hence it accesses

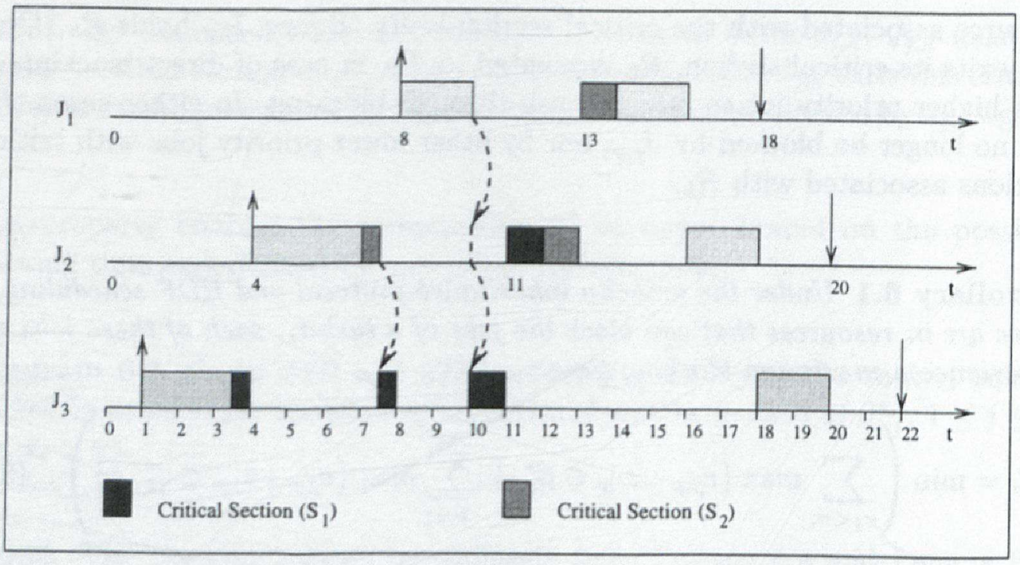

Figure 6.4 Example of transitive blocking.

R_1, and then completes its outer critical section releasing R_2, finally available to J_1 at time $t = 13$.

In general, given three tasks τ_i, τ_k and τ_j with preemption levels $\pi_i > \pi_k > \pi_j$, a job of τ_i can be blocked by a job of τ_k, which can be blocked by a job of τ_j: this can only happen if τ_k has two nested critical sections such that the outermost critical section can block τ_i's jobs, and on the innermost critical section τ_k can be blocked by τ_j. More formally:

$$\beta_{i,j} \supseteq \{z_{j,h} \in \beta_{k,j} : R_{j,h} = R_{k,p} \text{ and } z_{k,p} \subset z_{k,q} \text{ and } z_{k,q} \in \beta_{i,k}\} = \theta_{i,k,j}.$$

Note that the critical sections in $\theta_{i,k,j}$ are also considered in the determination of push-through blocking. Thus, they are not further analyzed.

The resource accesses that can cause push-through blocking can be characterized by observing that a resource R can cause this kind of blocking to a job J, only if R is accessed both by a lower priority job and by a job which has inherited or can inherit a priority equal to or higher than that of J. Unfortunately, this fact is not as useful as it is for fixed priority systems.

Furthermore, the notion of preemption levels is, in this case, not very useful. A job with high preemption level, in fact, can be subjected to push-through blocking by two jobs with lower preemption levels, as shown in Figure 6.5. Here, the job J_2 has the highest preemption level, however J_1 has the earliest

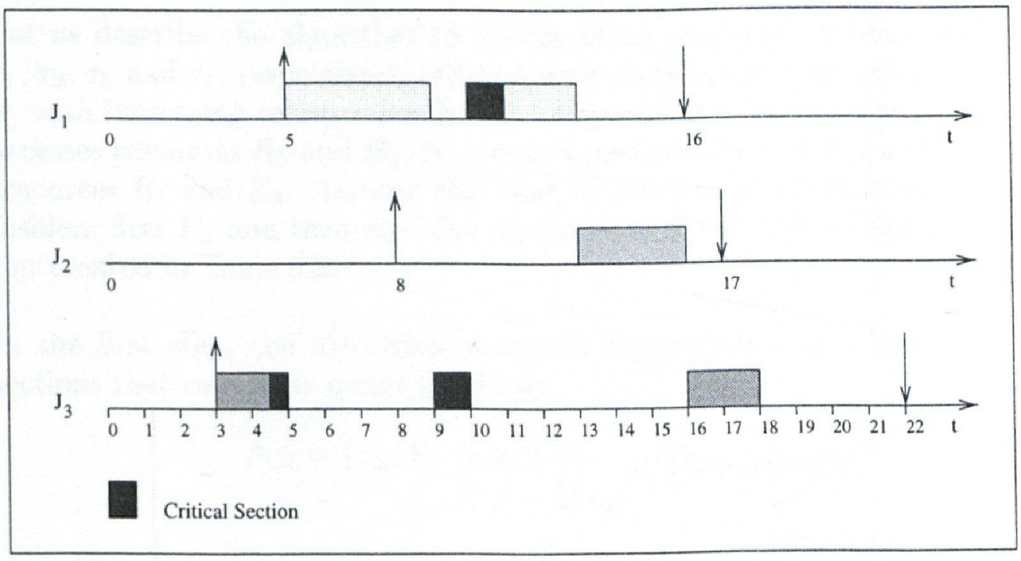
J1
0
5
16
t
J2
0
8
17
t
J3
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 t
Critical Section

Figure 6.5 Anomalous push-through blocking.

deadline in the system. Thus, when J_2 is released it cannot preempt J_1. J_1 is later blocked by J_3 on the shared resource. This blocking is also experienced by J_2.

Hence, the set of critical sections of τ_j that can block jobs of τ_i must include the critical sections that can also block jobs of any other task in the system. That is:

$$\beta_{i,j} \supseteq \{z_{j,h} \in \beta_{k,j} : \pi_k > \pi_j\} = \psi_{i,j}.$$

Note that $\forall k, \theta_{i,k,j} \subseteq \psi_{i,j}$. Hence, $\beta_{i,j}$ only takes into account the critical sections in $\psi_{i,j}$.

Let σ_i be the set of resources accessed by jobs of τ_i. The critical sections of τ_j that can directly block these jobs are those which share resources with τ_i. For each j such that $\pi_j < \pi_i$, $\beta_{i,j}$ can be finally written as:

$$\beta_{i,j} = \{z_{j,k} : R_{j,k} \in \sigma_i\} \cup \psi_{i,j}.$$

Given this formulation of the sets $\beta_{i,j}$ the algorithm shown in Figure 6.6 can be used. Without loss of generality, it is assumed that n tasks are ordered with decreasing preemption levels, that is, $\pi_i \geq \pi_j$ for all $i > j$. The complexity of the algorithm is $O(cn^3)$, where c is the maximum number of critical sections of each task in the system.

```
Algorithm BCS:

Begin
  Step 1: Direct blocking.
        for $i = 1$ to $n - 1$
            for $j = i + 1$ to $n$
                $\beta_{i,j} = \{z_{j,k} : R_{j,k} \in \sigma_i\}$
            endfor
        endfor
  Step 2: Transitive and push-through blocking.
        for $j = 2$ to $n$
            $\gamma_j = \bigcup_{h=1}^{j-1} \beta_{h,j};$
            for $i = 1$ to $j - 1$
                $\beta_{i,j} = \beta_{i,j} \cup \gamma_j$
            endfor
        endfor
End
```

Figure 6.6 Algorithm for the computation of blocking critical sections.

Let us describe the algorithm by means of an example. Assume four tasks, τ_1, τ_2, τ_3 and τ_4, respectively, ordered with decreasing preemption levels, that is, with increasing relative deadlines. Assume that τ_1 accesses resource R_3, τ_2 accesses resources R_3 and R_2, τ_3 accesses resource R_2 and R_1, and τ_4 accesses resources R_1 and R_3. Assume also that τ_3 accesses two resources in a nested fashion, first R_2 and then R_1. The key aspects of the code of the four tasks is represented in Table 6.2.

In the first step, the algorithm shown in Figure 6.6 determines the critical sections that can cause direct blocking:

$$\begin{array}{lll} \beta_{1,2} = \{z_{2,1}\} & \beta_{1,3} = \{\} & \beta_{1,4} = \{z_{4,2}\} \\ & \beta_{2,3} = \{z_{3,1}\} & \beta_{2,4} = \{z_{4,2}\} \\ & & \beta_{3,4} = \{z_{4,1}\} \end{array}$$

In the second step, the determination of the sets is completed with the addition of critical sections that can cause transitive blocking and push-through blocking:

$$\begin{array}{lll} \beta_{1,2} = \{z_{2,1}\} & \beta_{1,3} = \{z_{3,1}\} & \beta_{1,4} = \{z_{4,1}, z_{4,2}\} \\ & \beta_{2,3} = \{z_{3,1}\} & \beta_{2,4} = \{z_{4,1}, z_{4,2}\} \\ & & \beta_{3,4} = \{z_{4,1}, z_{4,2}\} \end{array}$$

Once the sets $\beta_{i,j}$ have been determined, the sets of maximal elements $\beta^*_{i,j}$ and later the sets $\zeta^*_{i,\cdot,k}$ of blocking critical sections accessed through the same resource R_k can be easily determined. The blocking time of each task in the system is then computed with equation (6.1).

An alternative to the algorithm presented in this section is to modify the procedure described in [12]. The procedure is based on the examination of all possible blockings of a job, by traversing a tree specifically built for this goal. The algorithm can find better bounds than those found by the algorithm presented in this section. However, it has an exponential complexity. In order to work for systems with EDF scheduling the procedure must be modified substituting the fixed priorities with the preemption levels of the tasks. The rest remains unchanged.

In the following, it is assumed that the blocking times have been determined as described in this section. The blocking time of the jobs of τ_i is denoted with B_i.

τ_1	τ_2	τ_3	τ_4
...	...	...	...
...	acquire(R_3)	...	acquire(R_1)
acquire(R_3)	...	acquire(R_2)	...
...	release(R_3)	...	release(R_1)
release(R_3)	...	acquire(R_1)	...
...	...	...	...
...	acquire(R_2)	...	acquire(R_3)
	...	release(R_1)	...
	release(R_2)	release(R_2)	...
	...	...	release(R_3)

Table 6.2 Structure of accesses to resources.

6.3.5 Feasibility Check

After the identification of the tasks' blocking times, the feasibility of the system under PIP must be checked. In the following theorem the tasks are assumed to be ordered by decreasing preemption levels (by increasing relative deadlines), that is, $\pi_i \geq \pi_j$ for all $i \leq j$.

Theorem 6.3 *Given a set $\mathcal{J}$ of n periodic and/or sporadic tasks, any job set generated by $\mathcal{J}$ is feasibly scheduled under EDF scheduling and Priority Inheritance Protocol,* if

$$\forall i = 1, \ldots, n \qquad \sum_{j=1}^{i-1} \frac{C_j}{D_j} + \frac{C_i + B_i}{D_i} \leq 1. \tag{6.2}$$

Proof. By induction on i, let us prove that if the first i conditions in (6.2) are true, then the subset of the first i tasks is feasibly scheduled even if the jobs can also encounter blockings normally due to jobs of tasks with lower preemption levels.

For $i = 1$ the hypothesis is true, since each job of τ_1 can be blocked at most for B_1 units of time and it is known that:

$$\frac{C_1 + B_1}{D_1} \leq 1.$$

That is, by the basic theorem relating utilization to feasibility, even if the computation time of τ_1 is increased by B_1, feasible schedules result.

Assume now that the hypothesis is true for the first $i-1$ conditions, and that also the i^{th} condition is true. Let $J_{j,k}$ be any job in the schedule S of the first i tasks. Let t be the last point in time not later than $r_{j,k}$, the release time of $J_{j,k}$, such that from time t up to the completion time $f_{j,k}$ of $J_{j,k}$, only jobs released at t or later and with deadlines less than or equal to $d_{j,k}$ are executed, except for some possible critical sections of jobs with lower preemption levels that can cause blocking. If $\mathcal{J}(t, f_{j,k})$ is the set of such jobs, the completion time of $J_{j,k}$ is:

$$f_{j,k} \leq t + B(t, f_{j,k}) + \sum_{J_{h,l} \in \mathcal{J}(t, f_{j,k})} C_{h,l},$$

where $B(t, f_{j,k})$ is the possible blocking time spent in $[t, f_{j,k}]$ executing critical sections of jobs with lower preemption levels, that is, jobs of tasks τ_p, with $p > i$.

If in $\mathcal{J}(t, f_{j,k})$ there are no jobs of τ_i, then S coincides with the schedule of the first $i-1$ tasks. By induction, this schedule is feasible, that is,

$$f_{j,k} \leq d_{j,k}.$$

If there are jobs of τ_i in $\mathcal{J}(t, f_{j,k})$, the blocking time of τ_i's jobs in $B(t, f_{j,k})$ must be taken into account. The blockings of the first $i-1$ tasks' jobs due to jobs with preemption level lower than π_i are push-through blockings for τ_i, hence they are included in B_i. If the first $i-1$ tasks are scheduled without blocking and the i^{th} task has blocking B_i, a new schedule exists in which $J_{j,k}$ does not complete earlier (note that blockings due to jobs in the same set $\mathcal{J}(t, f_{j,k})$ do not affect the completion time of $J_{j,k}$). Since the i^{th} condition in (6.2) is true, by the basic theorem relating utilization to feasibility, this new schedule is feasible. Again

$$f_{j,k} \leq d_{j,k}$$

follows. □

Similar to Chapter 3, the sufficient feasibility conditions can be checked in $O(n^2)$ time. Hence they are appropriate for use both off-line, at design time, and on-line, when tasks arrive dynamically.

6.4 THE DYNAMIC PRIORITY CEILING PROTOCOL

Priority inversion is not the only problem present in real-time systems with resources. There is another bad situation, which is not avoided by the Priority Inheritance Protocol, that can cause a high priority job to experience a large amount of blocking. This is the so called *chained blocking*, and happens because a high priority job is likely to be blocked whenever it wants to enter a critical section. If the job has several critical sections, it can be blocked for a considerable amount of time. Sha *et. al.* introduced the *Priority Ceiling Protocol* (PCP) to avoid *chained blocking* [18]. It works with a Rate Monotonic scheduler [10]. Chen and Lin [3] have extended the protocol to apply to an EDF scheduler. Later, the same authors have further extended the protocol in order to handle multiple-instance resources [4].

The main goal of these protocols is to reduce the occurrence of priority inversions beyond the reductions achieved with PIP alone. The key ideas are to prevent multiple priority inversions by means of early blocking of jobs that could cause priority inversion, and to minimize as much as possible the length of the priority inversion by allowing a temporary rise of the priority of the blocking job, that is, by using priority inheritance.

The early blocking of jobs is realized by dynamically keeping track of the *priority ceiling* of each resource, i.e., the priority of the highest priority job that may hold the resource at that time. When a job tries to hold a resource, the resource is made available only if the resource is free, and only if the priority of the job is greater than the current highest priority ceiling in the system. Such a rule can cause early blockings in the sense that a job can be blocked even if the resource it wants to access is free. This is not the case with PIP. The main advantages of early blocking are that it saves unnecessary context switches and that it affords the possibility of a simple and efficient implementation.

This access rule guarantees that any possible future job with higher priority is blocked at most once by the job which is currently holding a resource. Intuitively, chained blocking is prevented by ensuring that "among all resources needed by a future job, at most one of them is held by jobs with lower priorities at any time" [18]. This is the key to preventing multiple priority inversions experienced by a job.

Assuming an EDF priority assignment, the earlier a job's deadline, the higher its priority. Following the description given in [3], the PCP has two parts which define the priority ceiling of a resource and the handling of requests for resources:

"*Ceiling Protocol.* At any time, the priority ceiling of a resource R, $c(R)$, is equal to the original priority of the highest priority job that currently holds or will hold the resource.

Resource Allocation Protocol. A job J_i requesting a resource R is allowed to access R only if $pr_i > c(R_H)$, where pr_i is the priority of J_i and R_H is the resource with the highest priority ceiling among the resources currently held by jobs other than J_i. Otherwise, J_i waits and the job J_l which holds R_H inherits the priority of J_i until it releases R_H."

The protocol has been shown to have the following properties:

- A job can be blocked at most once before it enters its first critical section.
- It prevents deadlocks.

Of course, the former property is used to evaluate the worst case blocking times of the jobs. Given this protocol the schedulability formula of Liu and Layland [10] has been extended by Chen and Lin [3] to obtain the following condition.

Theorem 6.4 (Chen and Lin) *A set of n periodic tasks can be scheduled by EDF using the Dynamic Priority Ceiling Protocol if the following condition is satisfied:*

$$\sum_{i=1}^{n} \frac{C_i + B_i}{T_i} \leq 1,$$

where C_i is the worst case execution time, B_i is the worst case blocking length and T_i is the period of the task τ_i. □

A more precise guarantee test is also possible by considering n subtests, one for each task, as is done under the static PCP. This test is not presented here.

Compared to the PIP, the PCP has a higher processor utilization assuming the same task set, since the blocking time of each task is shorter. However, it is not suitable for task sets in which both the average number of resources accessed by a task and the average number of tasks that may access a resource are large. PCP can also unnecessarily delay jobs because the conditions it uses are sufficient, but not necessary. To rectify this, an optimal protocol has been developed in [13]. It delays jobs only if it helps to avoid potential priority inversions.

6.5 THE STACK RESOURCE POLICY

In [1], Baker introduced the Stack Resource Policy (SRP) that handles a more general situation in which multiunit resources, both static and dynamic priority schemes, and sharing of runtime stacks are all allowed. This section describes the protocol briefly.

The protocol relies on the following two conditions [1]:

(6.5.1) To prevent deadlocks, a job should not be permitted to start until the resources currently available are sufficient to meet its maximum requirements.

(6.5.2) To prevent multiple priority inversions, a job should not be permitted to start until the resources currently available are sufficient to meet the maximum requirement of any single job that might preempt it.

The key idea is that when a job needs a resource which is not available, it is blocked at the time it attempts to preempt, rather than later, when it actually may need the shared resource. As with PCP, SRP saves unnecessary context switches through earlier blocking and makes simple and efficient implementations possible. SRP derives is name because it can be easily implemented using a stack.

It should be noted that the basic premise underlying (6.5.1) also motivated the development of planning based scheduling algorithms (and they were actually developed before the SRP) in that they too do not schedule a job unless all the resources needed by it in the worst case are available. However, the details of how they make use of this requirement are different for the two types of algorithms.

The SRP uses a notion called the *resource ceiling*, which can be defined as the maximum preemption level of all the jobs that may be blocked directly by the currently available units of a resource. The protocol keeps a *system-wide* *ceiling* defined as the maximum of the current resource ceilings. A job with the earliest deadline is allowed to preempt only if its preemption level is greater than the current system-wide ceiling. This enforces the above conditions (6.5.1) and (6.5.2).

The preemption test has the effect of imposing priority inheritance (that is, an executing job that is holding a resource resists preemption as though it inherits the priority of any jobs that might need that resource). What is noteworthy is that this effect is accomplished without modifying the original priority of the job [1].

The SRP has been shown to have properties similar to those of the PCP. That is, it prevents chained blocking and is deadlock free. Furthermore, assuming n tasks ordered by decreasing preemption levels, that is, by increasing relative deadlines, Baker [1] has developed a sufficient condition to check the feasibility of a task set.

Theorem 6.5 (Baker) *A set of n tasks (periodic and sporadic) is schedulable by EDF scheduling with SRP* if

$$\forall k = 1, \ldots, n \qquad \left(\sum_{i=1}^{k} \frac{C_i}{D_i} \right) + \frac{B_k}{D_k} \leq 1.$$

□

The implementation complexity of the SRP is much lower than that of the dynamic PCP. In fact, in this case the ceilings are static, and so may be pre-computed and stored in a table. The current ceiling may be simply kept in a stack. Furthermore, the *acquire* and *release* primitives are simpler because resource requests cannot block. They don't require any blocking test or a context switch.

A concurrency control protocol similar to SRP has been independently developed for sporadic task sets [7]. In this paper, a necessary and sufficient schedulability test is derived based on the knowledge of a minimum and maximum execution time for each critical section. See the referenced paper for details.

6.6 RESOURCE SCHEDULING IN PLANNING-BASED SCHEDULERS

Recall that a planning algorithm work by attempting to constructing a schedule for a set of jobs. It starts with an empty schedule and extends it in steps, one job at a time, until, if it succeeds, it comes up with a complete feasible schedule. Here the additional considerations that enter the picture if jobs have resource constraints are presented.

In the presence of resources, the planning algorithm must compute the earliest start time, est_i, at which job J_i can begin execution after accounting for resource contention among jobs. Assume that a job can use a resource either in shared mode or in exclusive mode and holds a requested resource as long as it executes. It is assumed that jobs are non-preemptive. Given a partial schedule, then $erat_j$ the earliest time at which resource R_j is available, in shared/exclusive modes, can be determined. *erats* are computed taking into account the requirement that when a resource is held by a job in exclusive mode, no other job can be using it in exclusive or shared modes. Consider how this is accomplished.

When resources are taken into account, several data structures are required to keep track of the availability of resources. To simplify discussions, the data structures when the system has one instance of each resource are first presented. Subsequently, extensions to handle multiple instances are discussed. When only one instance exists for each resource, the planner maintains two vectors $erat^s$ and $erat^e$, to indicate the earliest resource available times of the r resources in the system for shared and exclusive modes, respectively:

$erat^s = (erat_1^s, erat_2^s, ..., erat_r^s)$ and

$erat^e = (erat_1^e, erat_2^e, ..., erat_r^e)$

Here $erat_i^s$ (or $erat_i^e$) is the earliest time when resource R_i will become available for shared (or exclusive) usage.

After the partial schedule is extended by one task, the planner updates $erat^s$ and $erat^e$ using the task's start time, computation time and resource requirements.

Here is a simple example to illustrate the computation of new $erat^s$ and $erat^e$ values: Assume a system has 5 resources, $R_1, R_2, ..., R_5$. Let current $erat^s$ and $erat^e$ be

$erat^s = (erat_1^s, erat_2^s, erat_3^s, erat_4^s, erat_5^s) = (5, 25, 10, 5, 10)$, and

$erat^e = (erat_1^e, erat_2^e, erat_3^e, erat_4^e, erat_5^e) = (5, 25, 10, 10, 15)$.

Suppose job J_i is being selected to extend the partial schedule. Assume $C_i = 10$, $r_i = 0$, and J_i requests R_1 and R_4 for exclusive use and R_5 for shared use. Then the earliest time that J_i can start is the earliest available time of the resources needed by task T. So,

$est_i = max(r_i, max_{j=1..r} erat_j^u) = \max(0, \text{MAX } (5, 10, 10\ )) = 10$ and,

J_i can start at 10.

The algorithm updates the $erat^s$ and $erat^e$ vectors as follows:

$erat^s = (erat_1^s, erat_2^s, erat_3^s, erat_4^s, erat_5^s) = (20, 25, 10, 20, 10\ )$, and

$erat^e = (erat_1^e, erat_2^e, erat_3^e, erat_4^e, erat_5^e) = (20, 25, 10, 20, 20\ )$.

Note that for R_5, both $erat_5^s$ and $erat_5^e$ need to be updated. $erat_5^s$=10 because task T uses R_5 in shared mode and it is therefore possible for some other task to utilize R_5 in parallel, in shared mode. However, $erat_5^e$=20 because another task which requires R_5 in exclusive mode cannot be permitted to execute in parallel with T.

Based on the above discussion, it is easy to observe that given a task's earliest start time, its finish time can be determined and thus the planner can decide if a task will finish by its deadline.

Now, the extensions to allow each distinct resource to have multiple instances are discussed. In this case, a vector no longer suffices to represent the two erat's. $erat^s$ and $erat^e$ have to be matrices so that earliest available time for every instance of each resource can be represented.

$$erat^s = \begin{array}{c} (erat^s_{11}, erat^s_{12}, ..., erat^s_{1n}) \\ (erat^s_{21}, erat^s_{22}, ..., erat^s_{2m}) \\ \vdots \\ \vdots \\ \vdots \\ (erat^s_{r1}, erat^s_{r2}, ..., erat^s_{rp}) \end{array}$$

and

$$erat^e = \begin{array}{c} (erat^e_{11}, erat^e_{12}, ..., erat^e_{1n}) \\ (erat^e_{21}, erat^e_{22}, ..., erat^e_{2m}) \\ \vdots \\ \vdots \\ \vdots \\ (erat^e_{r1}, erat^e_{r2}, ..., erat^e_{rp}) \end{array}$$

where n, m and p are the number of instances of resource items 1, 2 and r, respectively.

Given the representations for $erat^s$ and $erat^e$ as matrices est_i becomes:

$$est_i = max(r_i, max_{j=1..r} min_{k=1..q} erat^u_{jk})$$

where u is s when R_i is used in shared mode or e when R_i is used in exclusive mode and q is the number of instances of R_j.

The additional change when resources are included involves being resource cognizant while assigning priorities to jobs when jobs are being considered to extend the partial schedule. Two possible priority assignment policies are:

1. Minimum earliest start time first (Min_S): $Pr(J_i) = est_i$
2. Min_D + Min_S: $Pr(J_i) = d + W_1 * est_i$, where W_1 is a weighting constant.

The first policy does not consider tasks' timing constraints at all but does take into account resource availability. The second extends EDF to consider both time and resource constraints. This policy has been shown to result in good real-time performance [14] measured in terms of the planning algorithm's ability to find feasible schedules. In [16], the above planning algorithm has been extended to deal with tasks that can be parallelized, that is, mapped into subtasks that are executed concurrently on multiple processors.

6.7 SUMMARY

A spectrum of possibilities – starting from one that uses a non-preemptive job model to one that allows arbitrary preemptions – emerge when several approaches to ensure the integrity of shared resources are considered.

1. Adopt the non-preemptive job model where a job acquires and holds on to a resource throughout its execution.
   This is the approach taken by the planning based approaches. Jobs are scheduled in such a way that jobs that need the same resource are scheduled to exclude each other in time. This way, all blocking that could occur when jobs execute is avoided.
   However, if jobs are of long duration or if a job uses different resources during different parts of its execution then concurrency can be reduced and competing jobs can be made to wait for longer durations than is required for resource consistency. To alleviate this problem, a job is modeled as a set of precedence related components, with each component having its own resource requirements. Such jobs can then be scheduled using a planning based algorithm that can exploit precedence constraints of jobs.

2. Allow a job to be preempted in a controlled fashion: either after it releases the shared resources it currently holds or only if the preempting job does not require the resources needed by the preempted job.

   This characterizes the approach taken by priority-driven algorithms such as PIP, PCP, and SRP. Accesses to the resources are controlled in such a way that the duration of priority inversion is bounded and is as small as possible. As a result, feasibility checks can be undertaken to make sure that jobs meet their time constraints in spite of the blocking.

3. Allow a job to be preempted by a higher priority job at any point during its execution. The resources it currently holds are released after *undo*ing all the changes done by the preempted job. When the job resumes, it reexecutes from the beginning.
   In this case, the system must somehow be prepared to reinstate the resource's state as it existed when the preempted job began execution. This adds to the system's overheads and requires additional mechanisms for ensuring undoability. Hence, this solution is not preferred except in real-time transaction systems where such mechanisms are essential [15].

The above list shows that there is no "universal" solution to the problem of scheduling shared resource access in real-time systems. All of the approaches have their place depending on the nature of resources, jobs, and scheduling overheads that can be considered acceptable.

More experience is necessary in achieving practical realizations of the algorithms described in this chapter. In addition, a comprehensive set of solutions for the real-time scheduling of different types of resources is still being sought. In particular, good integrated scheduling and resource access policies that span processors, input/output resources, communication channels, and other resources need to be developed. The challenge lies in making them simple enough to implement and yet produce acceptable resource utilization levels.

As a final note, a protocol that shares many features with the Dynamic PIP and the SRP has been defined by Jeffay for scheduling sporadic tasks that access shared resources [7]. His model is more restrictive than that adopted by PIP and SRP. It assumes that a job can access at most one shared resource at any given time.

REFERENCES

[1] T.P. Baker, "Stack-Based Scheduling of Real-Time Processes," *Real-Time Systems Journal* 3, 1991.

[2] P. Brinch Hansen, "Structured Multiprogramming," *Communications of the ACM* 15(7), 1972.

[3] M. Chen and K. Lin, "Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems," *Real-Time Systems Journal* 2, 1990.

[4] M. Chen and K. Lin, "A Priority Ceiling Protocol for Multiple-Instance Resources," *Proc. of the Real-Time Systems Symposium*, 1991.

[5] A.N. Habermann, "Synchronization of Communicating Processes," *Communications of the ACM* 15(3), 1972.

[6] C.A.R. Hoare, "Monitors: An Operating System Structuring Concepts," *Communications of the ACM* 18(2), 1974.

[7] K. Jeffay, "Scheduling Sporadic Tasks with Shared Resources in Hard Real-Time Systems," *Proc. of the Real-Time Systems Symposium*, 1992, pp. 89-99.

[8] L. Lamport, "The Mutual Exclusion Problem: Part I - A Theory of Interprocess Communication," *Journal of the ACM* 33(2), 1986.

[9] L. Lamport, "The Mutual Exclusion Problem: Part II - Statement and Solutions," *Journal of the ACM* 33(2), 1986.

[10] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM* 20(1), 1973.

[11] A.K. Mok, Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment, Ph.D. Dissertation, MIT, 1983.

[12] R. Rajkumar, *Synchronization in Real-Time Systems: A Priority Inheritance Approach*, Kluwer Academic Publishers, Boston 1991.

[13] R. Rajkumar, L. Sha, J.P. Lehoczky and K. Ramamritham, "An Optimal Priority Inheritance Protocol for Real-Time Synchronization," in *Principles of Real-Time Systems*, Sang Son, Ed. Prentice-Hall, 1994.

[14] K. Ramamritham, J.A. Stankovic and P. Shiah, "Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems," *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184-94, April 1990.

[15] K. Ramamritham, "Real-Time Databases," *Journal of Distributed and Parallel Databases*, Volume 1, Number 2, 1993, pp. 199- 226.

[16] G. Manimaran, S. R. Murthy and K. Ramamritham, "A New Algorithm for Dynamic Scheduling of Parallelizable Tasks in Real-Time Multiprocessor Systems," *Real-Time Systems Journal*, to appear.

[17] M. Raynal, *Algorithms for Mutual Exclusion*, Cambridge (MA), MIT Press, 1986.

[18] L. Sha, R. Rajkumar and J.P. Lehoczky, Priority "Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers* 39(9), 1990.

[19] M. Spuri, "Efficient Deadline Scheduling in Real-Time Systems," Ph.D. Thesis, Scuola Superiore S. Anna, July 1995.

[20] J.A. Stankovic and K. Ramamritham, "The Spring Kernel: a New Paradigm for Real-Time Systems," *IEEE Software*, May 1991.

[21] W. Zhao, K. Ramamritham and J. Stankovic, "Preemptive Scheduling Under Time and Resource Constraints," *IEEE Transactions on Computers* 36(8), 1987.

7 PRECEDENCE CONSTRAINTS AND SHARED RESOURCES

In many hard real-time systems, due to the strict deadlines that must be met, communications among jobs are implemented in a completely deterministic manner. One approach used is to model communication requirements as precedence constraints among jobs, that is, if a job J_i has to communicate the result of its computation to another job J_j, the pair (J_i, J_j) is introduced in a partial order $\prec$, and the jobs are scheduled in such a way that if $J_i \prec J_j$ the execution of J_i precedes the execution of J_j.

Good examples of this modeling can be found in the MARS operating system [6, 7], in which the basic concept of a *real-time transaction* is described exactly in this way, and in Mok's *kernelized monitor* [8], in which a *rendez-vous* construct is used to handle similar situations. In both cases, shared resources among tasks are also considered. However, in the former work the whole schedule is statically generated, that is, is produced in advance before the system can run. The schedule is then stored in a table that, at run-time, is consulted by a dispatcher to actually schedule the jobs without any other computational effort. In the latter, although dynamic, the schedule is basically non-preemptive, or at least it can be said that the preemption points are chosen very carefully, since the processor is assigned in quantums of time of fixed length equal to the size of the largest critical section. This can often be inefficient.

This chapter describes a simple technique that can handle precedence constraints and shared resources for dynamic preemptive systems. Preemptive systems are generally much more efficient than non-preemptive ones, dynamic systems are more robust than the static solutions, and finally, the solution also has a formal basis and associated analytical formulas.