An RFB is only sent out if the sending processor estimates that there will be enough time for timely responses to it. In other words, it computes the sum of the estimated time taken by the RFB to reach its destinations, the estimated time taken by the destinations to respond with a bid, and the estimated time taken to transmit the bid to the focused processor. Call this sum t_{bid}. It also computes the latest time at which the focused processor can offload the task onto a bidder without the task deadline being missed; this time is given by the expression

$$t_{\text{offload}} = \text{task deadline} - (\text{current time} + \text{time to move the task} + \text{task-execution time})$$

If $t_{\text{bid}} \le t_{\text{offload}}$, then an RFB is sent out.

When processor p_r receives an RFB, it checks to see if it can meet the task requirements and still execute its already-scheduled tasks successfully. First, it estimates when the new task will arrive and how long it will take to be either guaranteed or rejected. This time is given by

$$\begin{aligned} t_{\text{arr}} = {} & \text{Current time} + \text{time for bid to be received by } p_s \\ & + \text{time taken by } p_s \text{ to make a decision} \\ & + \text{time taken to transfer the task} \\ & + \text{time taken by } p_r \text{ to either guarantee or reject the task} \end{aligned}$$

Next, it figures out the surplus computational time t_{surplus} that it currently has between t_{arr} and the task deadline D. This is done by first estimating the computational time already spoken for in the interval $[t_{\text{arr}}, D]$:

$$\begin{aligned} t_{\text{comp}} = {} & \text{time allotted to critical tasks in } [t_{\text{arr}}, D] \\ & + \text{time needed in } [t_{\text{arr}}, D] \text{ to run already-accepted noncritical tasks} \\ & + \text{fraction of recently accepted bids} \\ & \times \text{time needed in } [t_{\text{arr}}, D] \text{ to honor pending bids} \\ t_{\text{surplus}} = {} & D - \text{current time} - t_{\text{comp}} \end{aligned}$$

The degree to which the bid is aggressive or conservative can vary with the nature of the time estimates. If the task worst-case run times are used, the bids will be very conservative indeed. If average-case values are used instead, the bids will be less conservative.

If $t_{\text{surplus}} <$ task execution time, then no bid is sent out. If $t_{\text{surplus}} \ge$ task execution time, p_r sends out a bid to p_s. The bid contains t_{arr}, t_{surplus}, and an estimate of how long a task transferred to p_r will have to wait before it is either guaranteed or rejected.

All bids are sent to the focused processor p_s. If p_s is unable to process the task successfully, it can review the bids it gets back to see which other processor is most likely to be able to do so, and transfer the task to that processor. p_s

waits for a certain minimum number of bids to arrive or until a specified time has expired since receiving the task, whichever is sooner. It then evaluates each bid. For each bidding node p_i, p_s computes the estimated arrival time $\eta(i)$ of the task at that node. Denote by $t_{\text{surplus}}(i)$ and $t_{\text{arr}}(i)$ the t_{surplus} and t_{arr} values contained in the bid received from p_i. Then, p_s computes the following quantity for each bid:

$$t_{\text{est}}(i) = t_{\text{surplus}}(i) \frac{D - \eta(i)}{D - t_{\text{arr}}(i)} \tag{3.111}$$

Suppose $t_{\text{est}}(j)$ is the maximum such value. Then, if p_s cannot itself guarantee the task, it ships the task out to p_j.

The cardinal rule in bidding is that no new noncritical task can be allowed to cause any critical task or previously guaranteed noncritical task to miss its (guaranteed) deadline. Assessing the schedulability of a newly arrived task or responding to an RFB takes time, and can be indulged in only if doing so does not cause any guaranteed deadlines to be missed. There are two ways of determining this. The first is to introduce into the schedule a periodic task to check for schedulability; every t seconds, this task will assess the schedulability of tasks that have arrived over the last period. The second is to set a flag that indicates if the processor has the time to check the schedulability of a new task and still meet all guaranteed deadlines. If the flag is set when a new task arrives, the executing task is preempted and the processor deals with this new task. If the flag is reset, the processor is not interrupted and the new task must wait until the processor has time to handle it. A similar flag can be used to decide if there is enough time to respond to an RFB.

An issue worth discussing is how aggressive the bid should be. In many instances, worst-case task execution times are much greater than average-case times. Also, not all bids sent out by a processor are accepted. If the bidding is done too conservatively, by assuming that already-scheduled tasks will take their worst-case times and that all the bids will be successful, processors that could otherwise have successfully run the task in question may not bid. If the bidding is done too aggressively, then the probability is high that it will not be possible to honor the bid (should it be accepted). If an accepted bid cannot be honored, then a task, which might have been able to execute successfully on some other processor may miss its deadline. The designer must decide how to resolve this trade-off between aggressive and conservative bidding by finding a happy medium between them. One possible solution is for the system to adapt its bidding strategy based on experience. That is, if a processor has been unable to honor many recent bids, that processor is being too aggressive and can be made more conservative in its subsequent bidding. If, on the other hand, a processor has a lot of idle time despite declining to bid on tasks (which in hindsight it could have processed successfully), it is being too conservative and should become more aggressive.

3.4.6 The Buddy Strategy

The buddy strategy attacks the same type of problem as the FAB algorithm. Soft real-time tasks arrive at the various processors of a multiprocessor and, if an

individual processor finds itself overloaded, it tries to offload some tasks onto less lightly loaded processors. The buddy strategy differs from the FAB algorithm in the manner in which the target processors are found.

Briefly, each processor has three thresholds of loading: under (T_U), full (T_F), and over (T_V). The loading is determined by the number of jobs awaiting service in the processor's queue. If the queue length is Q, the processor is said to be in:

state U	(underloaded)	if $Q \leq T_U$
state F	(fully loaded)	if $T_F < Q \leq T_V$
state V	(overloaded)	if $Q > T_V$

A processor is judged to be in a position to execute tasks transferred from other processors if it is in state U. If it is in state V, it looks for other processors on which to offload some tasks. A processor in state F will neither accept tasks from other processors nor offload tasks onto other processors.

When a processor makes a transition out of or into state U, it broadcasts an announcement to this effect. This broadcast is not sent to all the processors; rather, its distribution is limited to a subset of the processors. This is called the processor's *buddy set*. Each processor is aware of whether any members of its buddy sets is in state U. If it is overloaded, it chooses an underloaded member (if any) in its buddy set on which to offload a task.

There are three issues worth discussing at this point. First, consider how the buddy set is to be chosen. If it is too large, the state-change broadcast will heavily load the interconnection network. If it is too small, the communication costs will be low, but the overloaded processors will be less successful in finding an underloaded processor in their buddy sets. Clearly, this applies only to multihop networks. If the interconnection network is a bus, for example, every broadcast will be seen by every processor and there is no saving in restricting delivery to a subset. If, on the other hand, a multihop network is used and the buddy set of a processor is restricted to the processors that are "close" to it (in terms of the number of hops between them), there will be a substantial saving of network bandwidth. The size of the buddy set will therefore depend on the nature of the interconnection network.

The second issue is a little more subtle. Suppose a node is in the buddy set of many overloaded processors, and that it delivers a state-change message to them, saying it is now underloaded. This can result in each of the overloaded processors simultaneously and independently dumping load on it, thus overloading that processor. To reduce the probability of this happening, we construct an ordered list of preferred processors. First we list the processors that are one hop away from the processor, then those which are two hops away, and so on. An overloaded processor searches its list in sequence, looking for underloaded processors, and sends a load to the first underloaded processor on its list. If the lists of the various processors are ordered differently, the probability of a node being simultaneously dumped on is reduced.

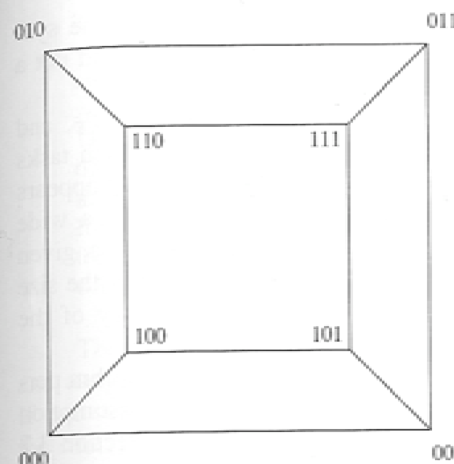

FIGURE 3.41
A three-dimensional hypercube network.

Example 3.40. Suppose the network is a three-dimensional hypercube. See Figure 3.41. Each processor in this system is connected to three other processors. Suppose the buddy set of each processor has been defined to be its immediate neighbors. Under the standard hypercube labeling, each node label will differ from its neighbors' labels in exactly one bit. Order the buddy list of each processor according to which bit is different. For example, if we give highest priority to processors differing in the LSB, then to the middle bit, and then to the MSB, the priority list of each of the processors is as shown in Table 3.2. For example, node 000 is in the buddy set of 001, 010, 100. It is the first priority node for node 001 (i.e., if node 001 is overloaded, it will check if node 000 is underloaded first. It is the second priority node for node 010 and the third priority node for node 100. Both nodes 001 and 010 will only simultaneously dump load on node 000 if (a) they are both overloaded, and (b) node 011 is not in state U. This is clearly an event that is much less probable than if 000 were the first on the list of all three of its neighbors.

For large buddy sets, if the priority lists are staggered properly, it is extremely unlikely that more than a small fraction of the processors in a buddy set will simultaneously dump a load on a given processor. Some simulation experiments indicate that buddy-set sizes of 10 to 15 are best, even for very large

TABLE 3.2
Example of priority list

Node	Priority list		
000	001	010	100
001	000	011	101
010	011	000	110
011	010	001	111
100	101	110	000
101	100	111	001
110	111	100	010

systems. Keeping the size of buddy sets constant and independent of the system size results in the state-update traffic per processor being constant and not a function of the system size.

The third issue of importance is the choice of the thresholds T_U, T_F, and T_V. In general, the greater the value of T_V, the smaller the rate at which tasks are transferred from one node to another. From simulation experiments, it appears that setting $T_U = 1, T_F = 2$, and $T_V = 3$ produces good results for a wide range of system parameters. Of course, which thresholds are best for a given system depends on the particular characteristics of that system, including the size of the buddy set, the prevailing load, and the bandwidth and topology of the interconnection network.

There is, at present, no general model of how well the buddy algorithm performs. An approximate model has been derived under the simplifying assumption that all tasks take one unit time to execute. The reader is referred to Section 3.7 for references in the technical literature.

3.4.7 Assignment with Precedence Conditions

In this section, we present a simple algorithm that assigns and schedules tasks with precedence conditions and additional resource constraints. The basic idea behind the algorithm is to reduce communication costs by assigning (if possible) to the same processor tasks that heavily communicate with one another.

The underlying task model is as follows. Each task may be composed of one or more subtasks. The release time of each task and the worst-case execution time of each subtask are given. The subtask communication pattern is represented by a task graph: that implicitly defines the precedence conditions (i.e., which subtask feeds output to which other subtask). We are also given the volume of communication between subtasks. It is assumed that if subtask s_1 sends output to s_2, this is done at the end of the s_1 execution, and must arrive before s_2 can begin executing.

Associated with each subtask is a latest finishing time (LFT). The LFT of a subtask s_i is computed as follows. Suppose that n_i is the number of paths in the task graph emanating from s_i. Let $\{v^{\ell}_{k,1}, v^{\ell}_{k,2}, \ldots, v^{\ell}_{k,m_\ell}\}$ be the nodes on the ℓth such path. Let η_ℓ be the sum of the execution times of the subtasks represented by the nodes on that path, and $D_{v^{\ell}_{k,m_\ell}}$ be the deadline of the last node on this path. We have:

$$\mathrm{LFT}(n_i) = D_{v^{\ell}_{k,m_\ell}} - \eta_\ell \tag{3.112}$$

Example 3.41. Consider the task graph shown in Figure 3.42. The execution times are as follows:

Subtask	Execution time
s_0	4
s_1	10
s_2	15
s_3	4
s_4	4

The labels within the circles are the subtask subscripts and the arc labels denote the volume of communication between the tasks. Suppose the overall deadline for this task (i.e., the deadline of subtask s_4) is 30. The LFT for s_4 is clearly 30, and for s_1, s_2, s_3 is $30 - e_4 = 26$. The LFT for s_0 is $30 - e_4 - \max\{e_1, e_2, e_3\} = 30 - 4 - 15 = 11$. Note two things from this. First, we are not accounting for any communication time here. Second, we are assuming that s_1, s_2, s_3 are run in parallel. If they were run in tandem (say, on the same processor), the LFT of s_0 would be $30-(e_1+e_2+e_3+e_4)$. If a subtask does not finish by its LFT, we are sure to miss a deadline, especially if the subtasks consume their worst-case execution times. However, meeting the LFT does not always guarantee that deadlines will all be met.

The algorithm is a trial-and-error process. We start with the set of subtasks and assign them to the processors one by one, in the order of their LFT values. If two subtasks have the same LFT value, the tie is broken by giving priority to the subtask with the greatest number of successor subtasks.

We check for feasibility with each assignment—if one assignment is not feasible, we try another, and so on. If no assignment works for this particular subtask, we backtrack and try changing the assignment of the previously assigned subtask and continue from there.

Subtasks that communicate with each other a lot are assigned to the same processor if this allows for feasible scheduling. A threshold policy is followed for this. If e_i and e_j are the execution times of subtasks s_i and s_j, respectively, and c_{ij} is the volume of communication between them, we try assigning s_i and s_j to the same processor if

$$\frac{e_i + e_j}{c_{ij}} < k_c$$

where k_c is a given parameter (to be discussed below). The idea behind this is to balance the benefits of assigning subtasks to the same processor (the communication cost is zero) against the potential benefits of assigning them to different processors (evening out the load and in some cases reducing the subtask finishing times by letting them run in parallel). The assignment is done in part by checking each pair of communicating subtasks to see if they must be assigned to the same processor.

Example 3.42. Consider again the task graph shown in Figure 3.42. Suppose $k_c = 3$. Then, $\{s_1, s_4\}$, $\{s_2, s_4\}$, and $\{s_3, s_4\}$ must all be placed on the same processor. (The

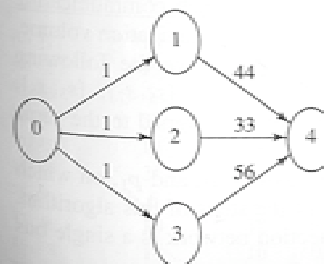

FIGURE 3.42
Task graph for calculating the latest finishing time.

side effect of this is that s_1, s_2, s_3 must be on the same processor, even though there is no communication between them.)

k_c is a tunable parameter. If k_c is 0, then intersubtask communication will simply not be taken into account when we do the assignment. If k_c is very high, then, most of the time, we will be forced to assign tasks to the same processor. This may result in the schedule becoming infeasible. In that case, we will be forced to reduce k_c adaptively to relax this constraint.

The algorithm can be informally described as follows:

1. Start with the set of tasks that need to be assigned and scheduled. Choose a value for k_c and determine, based on this value, which tasks need to be on the same processor.
2. Start assigning and scheduling the subtasks in order of precedence. If a particular allocation is infeasible, reallocate if possible. If feasibility is impossible to achieve because of the tasks that need to be assigned to the same processor, reduce k_c suitably and go to the previous step. Stop when either the tasks have been successfully assigned and scheduled, or when the algorithm has tried more than a certain number of iterations. In the former case, output the completed schedule; in the latter, declare failure.

Example 3.43. Let us now illustrate this algorithm with an example consisting of one task with eight subtasks, as specified in the table below, with task graph shown in Figure 3.43.

Subtask	Execution time	Deadline	LFT
s_0	4	–	7
s_1	10	–	24
s_2	15	22	22
s_3	4	–	26
s_4	18	–	42
s_5	3	–	42
s_6	6	–	32
s_7	3	45	45
s_8	8	40	40

The output to the "outside world" is provided by s_2, s_7, s_8, and so these are the only subtasks whose deadlines are specified. Table 3.3 shows, for each communicating pair of tasks, the ratio between the total execution time and communication volume.

If, for example, we set $k_c = 1.5$, we would have to have the following pairs assigned to the same processor: $\{s_0, s_1\}$, $\{s_0, s_2\}$, $\{s_0, s_3\}$, $\{s_5, s_7\}$, $\{s_3, s_6\}$, and $\{s_6, s_8\}$. This implies that the following tasks must all be assigned to the same processor: $s_0, s_1, s_2, s_3, s_6, s_8$.

Suppose we have a system consisting of two processors, p_0 and p_1, on which we seek to assign these tasks. The following is a list of the steps of this algorithm. We start with $k_c = 1.5$ and assume that the interconnection network is a single bus used by a two-processor system.

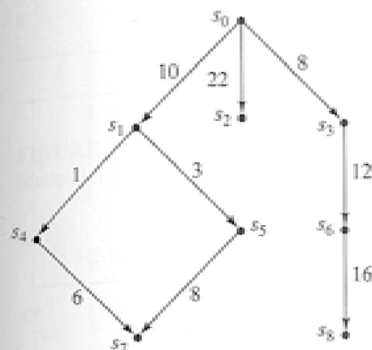

FIGURE 3.43
Task graph.

1. Assign s_0 to processor p_0 over the interval [0, 4] (assuming time 0 is when this task is released).
2. Assign s_1 to processor p_0 and schedule it over [4, 14]. (No communication costs are incurred because s_1 is on the same processor as s_0).
3. Assign s_2 to processor p_0. It is immediately clear that we cannot schedule both s_1 and s_2 so that they complete before their respective LFT values. Therefore, we need to change k_c. Let us reduce it to $k_c = 1.0$. Now, the following pairs must be assigned to the same processor: $\{s_0, s_2\}$, $\{s_5, s_7\}$, $\{s_3, s_6\}$, and $\{s_6, s_8\}$.
4. Now, assign s_1 to p_1, and schedule it over [14, 24]. Also schedule the network path between p_0 and p_1 for [4, 14] (for the communication between s_0 and s_1).
5. Assign s_2 to p_0 and schedule it over [4, 19].
6. Assign s_3 to p_0 and schedule it over [19, 23].
7. Assign s_4 to p_0 and schedule it over [23, 41].
8. Assign s_5 to p_0. This is infeasible, as we can check easily. Backtrack to the previous step.
9. Reassign s_4 to p_1 and schedule it over [24, 42].
10. Reassign s_5 to p_0 and schedule it over [27, 30].
11. Assign s_6 to p_0 and schedule it over [23, 29]. Move s_5 to [29, 32].

TABLE 3.3
Execution time to communication volume ratio.

Pair s_i, s_j	$e_i + e_j$	c_{ij}	$(e_i + e_j)/c_{ij}$
s_0, s_1	14	10	1.40
s_0, s_2	19	22	0.86
s_0, s_3	8	8	1.00
s_1, s_4	28	14	2.00
s_1, s_5	13	3	4.33
s_4, s_7	21	6	3.50
s_5, s_7	6	8	0.75
s_3, s_6	10	12	0.83
s_6, s_8	14	16	0.88

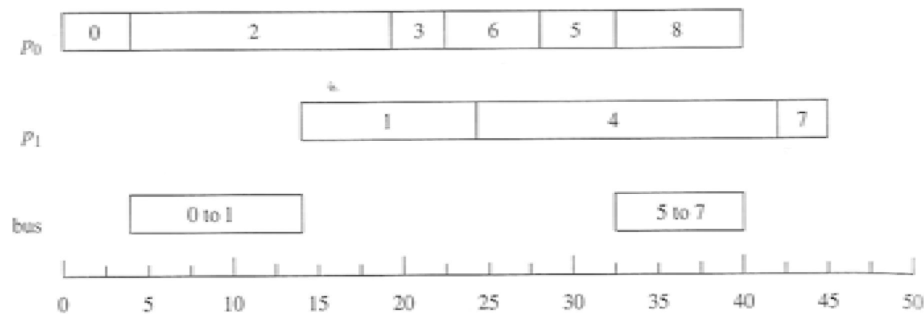

FIGURE 3.44
Example of the task assignment and scheduling algorithm.

12. The assignment of s_7 to p_0 is infeasible, so assign s_7 to p_1 and schedule it over [42,45].
13. Assign s_8 to p_0 and schedule it over [32, 40].

The schedule is shown in Figure 3.44. Note that we are scheduling each task in the task set. If the task set is periodic, then we only need to consider the tasks released during the least common multiple of the task periods.

3.5 MODE CHANGES

The workload of a real-time computer can change over time, as the mission phase changes or as processor failures dictate. Tasks may be added or deleted from the schedule, or may have their period or execution time changed. In this section, we look at how such mode changes can be accommodated under the rate-monotonic scheduling algorithm. Our objective is to allow the system to incorporate some new tasks (or delete some current ones) and still meet all the hard deadlines.

When critical sections are not involved, mode changes are quite simple to implement. Deleting a task (assuming its output is not needed by any other task) clearly does not adversely affect the schedulability of the other tasks. That is, if a task set $S = \{T_1, \ldots, T_n\}$ is RM-schedulable, $S - \{T_i\}$ is also RM-schedulable. This brings up the question of when the time allocated to the deleted task(s) can be reclaimed for use by other tasks. In order for schedulability to continue to be met, the time can only be reclaimed after the end of the period of the last iteration of the deleted task. That is, suppose we are deleting task T in Figure 3.45. T completes execution for the last time at t_0. However, its current period runs until t_1, and its deletion does not take effect until that time. This ensures that the schedulability rules we derived for the RM algorithm continue to hold throughout the mode-change process.

Adding a task is just as simple. We need only to check that it meets the RM-schedulability conditions outlined earlier in this chapter.

Example 3.44. Consider a system that is currently executing task set $\{T_1, T_2, T_3\}$ with $P_1 = 5, P_2 = 8, P_3 = 13$, and $e_1 = 1, e_2 = 3, e_3 = 4$. This task set is

FIGURE 3.45
Mode changing: task T is not deleted until t_1.

RM-schedulable since

$$e_1 \le P_1$$
$$2e_1 + e_2 \le P_1$$
$$3e_1 + 2e_2 + e_3 \le P_3$$

Assume that all these tasks have zero phasings. Suppose that at time 30 we wish to replace task T_3 by T_4, with parameters $P_4 = 14, e_4 = 5$. First, we check if the set $\{T_1, T_2, T_4\}$ is RM-schedulable. It is, because of the following inequalities:

$$e_1 \le P_1$$
$$e_1 + e_2 \le P_1$$
$$3e_1 + 2e_2 + e_4 \le P_4$$

When can we add task T_4? Under the RM algorithm, T_3 has run part of its third iteration at time 30. As a result, we cannot replace T_3 by T_4 until the end of the current T_3 period, that is, until 39. At that time, T_4 may be inducted into the task set.

If the priority ceiling protocol is used to handle the accesses to exclusive resources, the priority ceilings of the semaphores are lowered or raised as appropriate when the underlying tasks that give rise to these priorities are deleted or added. Recall that the priority ceiling of a critical section is the maximum of the priorities of all tasks that can access it.

The rule for deleting tasks is the same as when no critical sections are involved. A task may be added if the following two conditions are satisfied:

- the resultant task set is RM-schedulable, and
- when adding the task will result in the increase of the priority ceilings of any of the semaphores, those ceilings are raised before the task is added.

If the priority ceiling of a semaphore needs to be changed, the rules are:

- If the semaphore is unlocked, the ceiling is changed immediately in an indivisible action.
- If the ceiling is to be raised and the semaphore is locked, we wait until it is unlocked before raising it.
- If, as a result of some tasks being deleted, the priority celings decrease, this occurs at the time of deletion.

Example 3.45. Suppose that we have three semaphores, S_1, S_2, S_3, and four tasks T_1, T_2, T_3, T_4 in the overall task set. (As usual, we assume $T_1 \succ T_2 \succ T_3 \succ T_4$.) The following chart indicates which task may lock the semaphores:

Semaphore	Tasks
S_1	T_1, T_3
S_2	T_2, T_3, T_4
S_3	T_1, T_3, T_4

Suppose now that we wish to delete task T_2 from the task set. At the moment that T_2 is deleted, the priority ceiling of S_2 will drop from the priority of T_2 to that of T_3.

Assume that S_1 is currently locked and we wish to add task T_0 to the task set (with $T_0 \succ T_1$). Then, the priority ceilings of S_1 and S_3 will need to be raised to the priority of T_0. This may not be done, however, until the lock on S_1 is released. At that moment, the ceiling of S_1 is raised. Note that until this happens, we cannot add T_0 to the task set.

The mode change protocol has the same properties as the priority ceiling protocol. In particular, under the mode change protocol, there is no deadlock, nor can a task be blocked for more than the duration of a single outermost critical section. The proofs are similar to those derived earlier for the priority ceiling protocol, and are omitted.

3.6 FAULT-TOLERANT SCHEDULING

The advantage of static scheduling is that more time can be spent in developing a better schedule. However, static schedules must have the ability to respond to hardware failures. They do this by having a sufficient reserve capacity and a sufficiently fast failure-response mechanism to continue to meet critical-task deadlines despite a certain number of failures.

The approach to fault-tolerant scheduling that we consider here uses additional *ghost* copies of tasks, which are embedded into the schedule and activated whenever a processor carrying one of their corresponding primary or previously-activated ghost copies fails. These ghost copies need not be identical to the primary copies; they may be alternative versions that take less time to run and provide results of poorer but still acceptable quality than the primaries.

We will assume a set of periodic critical tasks. Multiple copies of each version of a task are assumed to be executed in parallel, with voting or some other error-masking mechanisms (see Chapter 7 for a detailed discussion of such mechanisms.) When a processor fails, there are two types of tasks affected by that failure. The first type is the task that is running at the time of failure, and the second type comprises those that were to have been run by that processor in the future. The use of forward-error recovery is assumed to be sufficient to compensate for the loss of the first type of task. The fault-tolerant scheduling algorithm

we describe here is meant to compensate for the second type by finding substitute processors to run those copies.

We assume the existence of a nonfault-tolerant algorithm for allocation and scheduling. This algorithm will be called as a subroutine by the fault-tolerant scheduling procedure that we will describe. We assume that the allocation/scheduling procedure consists of an assignment part Π_a and an EDF scheduling part Π_s.

We are now in a position to define our problem more precisely. Suppose the system is meant to run $n_c(i)$ copies of each version (or iteration) of task T_i, and is supposed to tolerate up to n_{sust} processor failures. The fault-tolerant schedule must ensure that, after some time for reacting to the failure(s), the system can still execute $n_c(i)$ copies of each version of task i, despite the failure of up to n_{sust} processors. These processor failures may occur in any order.

The output of our fault-tolerant scheduling algorithm will be a ghost schedule, plus one or more primary schedules for each processor. If one or more of the ghosts is to be run, the processor runs the ghosts at the times specified by the ghost schedule and shifts the primary copies to make room for the ghosts. Example 3.46 illustrates this.

Example 3.46. Figure 3.46 shows an example of a pair of ghost and primary schedules, together with the schedule that the processor actually executes if the ghost is activated. Of course, this pair of ghost and primary schedules is only feasible if, despite the ghost being activated, all the deadlines are met.

A ghost schedule and a primary schedule are said to form a *feasible pair* if all deadlines continue to be met even if the primary tasks are shifted right by the time needed to execute the ghosts. Ghosts may overlap in the ghost schedule of a processor. If two ghosts overlap, only one of them can be activated. For example, in Figure 3.47, we cannot activate both g_1 and g_2 or both g_2 and g_3. We can, however, activate both g_1 and g_3. There are two conditions that ghosts must satisfy.

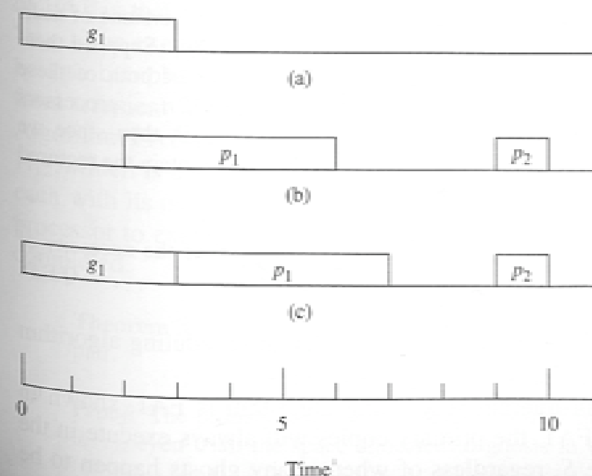

FIGURE 3.46
Schedules for Example 3.46: (a) ghost schedule; (b) primary schedule; (c) schedule if $g1$ is activated.

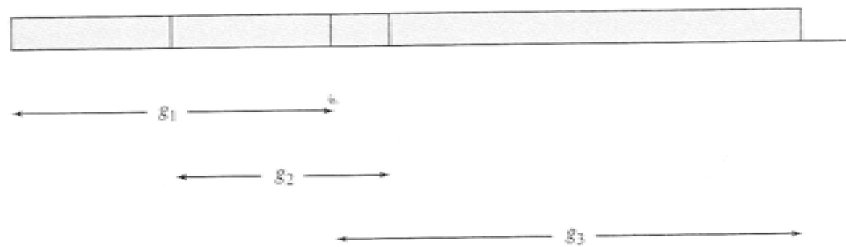

FIGURE 3.47
Overlapping ghosts.

C1. Each version must have ghost copies scheduled on n_{sust} distinct processors. Two or more copies (primary or ghost) of the same version must not be scheduled on the same processor.

C2. Ghosts are *conditionally transparent*. That is, they must satisfy the following two properties:

a. Two ghost copies may overlap in the schedule of a processor if no other processor carries a copy (either primary or ghost) of both tasks.

b. Primary copies may overlap the ghosts in the schedule only if there is sufficient slack time in the schedule to continue to meet the deadlines of all the primary and activated ghost copies on that processor.

Theorem 3.18. Conditions **C1** and **C2** are necessary and sufficient conditions for up to n_{sust} processor failures to be tolerated.

Proof

Necessity of C1. Suppose some version had ghost copies allocated to n processors $\pi_1, \ldots, \pi_n$. The failure of the processors $\pi_1, \ldots, \pi_n$, together with the failure of any processor carrying a primary copy of this task, would make the system no longer capable of running the required number of copies. Thus, if $n < n_{sust}$, the system cannot tolerate n_{sust} processor failures.

Necessity of C2a. Consider ghost copies g_1 and g_2 of the ith versions of tasks I and T, which overlap in the ghost schedule of some processor p_1. Suppose there exists some other processor p_2 that is allocated primary copies of both of these versions. There is a total of n_{sust} ghost copies of each version. If n_{sust} processors (other than p_1), carrying either the ghost or primary version, fail, then since we cannot activate both g_1 and g_2 in the schedule of p_1, we cannot maintain the required number of active copies.

Necessity of C2b. This is obvious.

Sufficiency of C1 and C2. This part is similar to the proof of the necessity of **C1** and **C2**, and is left as an exercise for the reader. Q.E.D.

Theorem 3.18 provides the conditions that the fault-tolerant scheduling algorithm must follow in producing the ghost and primary schedules.

Perhaps the simplest fault-tolerant scheduling algorithm is FA1, shown in Figure 3.48. Under algorithm FA1, the primary copies will always execute in the positions specified in schedule S, regardless of whether any ghosts happen to be

1. Run Π_a to obtain a candidate allocation of copies to processors. Denote by π_i and θ_i the primary and ghost copies allocated to processor p_i, $i = 1, \ldots, n_p$.
2. Run $\Pi_s^*(\pi_i \cup \theta_i, i)$. If the resultant schedule is found to be infeasible the allocation as produced by Π_a is infeasible; return control to Π_a in step 1. Otherwise, record the position of the ghost copies (as put out by Π_s^*) in ghost schedule G_i, and the position of the primary copies in schedule S.

FIGURE 3.48
Algorithm FA1.

1. Run Π_a to obtain a candidate allocation of copies to processors. Denote by π_i and θ_i the primary and ghost copies allocated to processor p_i, $i = 1, \ldots, n_p$. For each processor, p_i, do steps 2 and 3.
2. Run $\Pi_s^*(\pi_i \cup \theta_i, i)$. If the resultant schedule is found to be infeasible, the allocation as produced by Π_a is infeasible; return control to Π_a in step 1. Otherwise, record the position of the ghost copies (as put out by Π_s^*) in ghost schedule G_i. Assign static priorities to the primary tasks in the order in which they finish executing, i.e., if primary π_i completes before π_j in the schedule generated in this step, π_i will have higher priority than π_j.
3. Generate primary schedule S_i by running Π_{s2} on π_i, with the priorities assigned in step 2.

FIGURE 3.49
Algorithm FA2.

activated, since the ghost and primary schedules do not overlap. The drawback of FA1 is that the primary tasks are needlessly delayed when the ghosts do not have to be executed. While all the tasks will meet their deadlines, it is frequently best to complete execution of the tasks early to provide slack time to recover from transient failures (see Chapter 7). Such a needless delay does not occur in Algorithm FA2, shown in Figure 3.49. We use an additional scheduling algorithm Π_{s2}, which is a static-priority preemptive scheduler. That is, given a set of tasks, each with its own unique static priority, Π_{s2} will schedule them by assigning the processor to execute the highest-priority task that has been released but is not yet completed.

Theorem 3.19. The ghost schedule G_i and the primary schedule S_i form a feasible pair.

Proof. The primary tasks will complete no later than the time specified in $\Pi_s^*(\pi_i \cup \theta_i, i)$, even if all the space allocated to ghosts in G_i is, in fact, occupied by them. Q.E.D.

Example 3.47. Consider the case where a processor p has been allocated ghosts g_4, g_5, g_6, and primaries π_1, π_2, π_3. The release times, execution times, and deadlines are as follows:

	π_1	π_2	π_3	g_4	g_5	g_6
Release time	2	5	3	0	0	9
Execution time	~~4~~ 2	2	~~2~~ 4	2	2	3
Deadline	6	8	15	5	6	12

Suppose that there exists some processor q to which the primary copies of g_4 and g_5 have both been allocated. Then, we cannot overlap g_4 and g_5 in the ghost schedule of processor p. As a result, the ghost schedule will be as shown in Figure 3.50. It is easy to check that under these constraints, the allocation of $\pi_1, \pi_2, \pi_3, g_4, g_5, g_6$ to p is infeasible—π_2 will miss its deadline if all the ghosts are activated.

However, suppose that we have some other allocation that also allocates $\pi_1, \pi_2, \pi_3, g_4, g_5, g_6$ to p. Under this new allocation, there is no other processor to which primary or ghost copies of both g_4 and g_5 have been allocated. As a result, we can overlap g_4 and g_5 in the ghost schedule, producing the ghost schedule shown in Figure 3.51. Under these constraints, a feasible schedule can be constructed for p; see Figure 3.52.

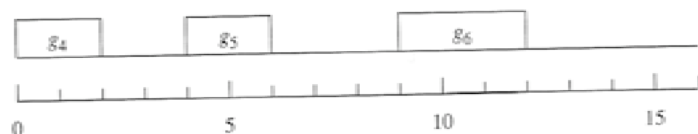

FIGURE 3.50
Ghost schedule of p if g_4 and g_5 cannot overlap.

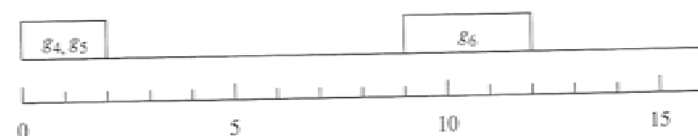

FIGURE 3.51
Ghost schedule of p if g_4 and g_5 can overlap.

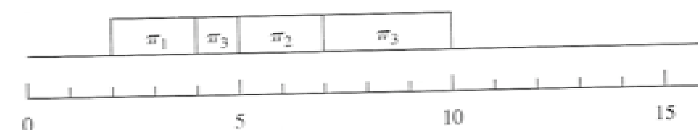

FIGURE 3.52
Feasible primary schedule of p if g_4 and g_5 can overlap.

3.7 SUGGESTIONS FOR FURTHER READING

A good, if slightly dated, introduction to the theory of scheduling is provided in [5, 6]. The RM and EDF scheduling algorithms are considered in detail in Liu and Layland [16]. The necessary and sufficient conditions for RM-schedulability are published in [13]. The case where deadlines do not equal the task periods for the RM algorithm is found in [12]. The deferred server algorithm is presented in [14]. Some practical issues relating to the RM algorithm are discussed in [23]. See [11] for additional discussions of the EDF algorithm.

Determining EDF-schedulability for sporadic tasks is presented in [3]. The MINPATH algorithm is described in [4]. The algorithm for taking precedence and exclusion constraints into account appears in [27].

The source for the primary/alternative task scheduling algorithm is [14]. IRIS tasks are studied in [8, 17, 18]. The algorithm for computing the value of π is from [19].

The next-fit algorithm for assigning RM-schedulable tasks is published in [7]. The utilization-balancing algorithm is taken from [2]. The bin-packing assignment algorithm for EDF is based on [9] and is well described in [5]. The MOS heuristic is presented in [20].

The priority ceiling protocol is introduced in [22]. A refinement of this protocol is presented in [1].

Focused addressing and bidding are discussed in [21]. The buddy strategy for assigning tasks to processors is reported in [25].

The primary/ghost algorithm for fault-tolerant scheduling was described in [8].

EXERCISES

3.1. Construct a set of periodic tasks (with release times, execution times, and periods), which can be scheduled feasibly by the EDF algorithm, but not by the RM algorithm.

3.2. Describe situations in which a task should not be preempted.

3.3. In the proof of Theorem 3.1, we considered only a two-task system. Generalize the proof to n tasks for $n > 2$.

3.4. We have been assuming in this chapter that preemption incurs no overhead. Let us now relax that assumption. Consider a two-task system where each preemption has an overhead of x. Given e_1, e_2, P_1, P_2, obtain the maximum value of x for which the task set is RM-schedulable.

3.5. This question relates to transient overloads. Consider a set of five tasks with the following characteristics.

i	e_i	a_i	P_i
1	30	5	100
2	10	5	130
3	20	15	140
4	80	10	140
5	10	10	200

Tasks T_1, T_3, T_5 are critical, but T_2, T_4 are not. Carry out a period transformation for this task set to ensure that the critical tasks always meet their deadlines.

3.6. Prove that $t(k, i)$ is the minimum t for which Equation (3.35) holds.

3.7. Prove Lemma 3.9. (Hint: This is very similar to the proof of Theorem 3.11.)

3.8. Assuming that the priority inheritance algorithm is followed, construct an example where there are N tasks, and the highest-priority task is blocked once by every one of the other $N - 1$ tasks.

3.9. Prove Theorem 3.7.

3.10. Write the time-reclamation algorithm mentioned in Section 3.2.4.

3.11. Redefine L_i so that Theorem 3.4 continues to hold when the priority ceiling algorithm is used.

3.12. Suppose a task can suspend itself. Show that if a task T_i suspends itself n times, it can be blocked by at most $n + 1$ (not necessarily distinct) members of B_i.

3.13. Prove Theorem 3.14.

3.14. Prove Theorem 3.15.

3.15. Prove Theorem 3.16.

3.16. Find the computational complexity of IRIS5.

3.17. From Equation (3.96), it follows that if the solution of problem q_i results in tasks in some set A receiving any service during an interval $(d_{i-1}, d_i]$ their reward functions must all have equal derivatives for problems $q_{i-1}, \ldots q_1$ as well. We can therefore group all tasks in A together during the solution of $q_{i-1}, \ldots q_1$. Modify IRIS5 to take advantage of this fact. Find the computational complexity of your modified algorithm.

REFERENCES

[1] Baker, T. P.: "A Stack-Based Resource Allocation Policy for Realtime Processes," *Proc. IEEE Real-Time Systems Symp.*, pp. 191–200, IEEE, Los Alamitos, CA, 1990.

[2] Bannister, J. A., and K. S. Trivedi: "Task Allocation in Fault-Tolerant Distributed Systems," *Acta Informatica* 20:261–281, 1983.

[3] Baruah, S. K., A. K. Mok, and L. E. Rosier: "Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor," *Proc. IEEE Real-Time Systems Symp.*, pp. 182–190, IEEE, Los Alamitos, CA, 1990.

[4] Bettati, R. D., Gillies, C. C. Han, K. J. Lin, C. L. Liu, J. W. S. Liu, and W. K. Shih: "Recent Results in Real-Time Scheduling," in *Foundations of Real-Time Computing: Scheduling and Resource Management* (A. van Tilborg and G. Koob, eds.), Kluwer Academic, Boston, pp. 91–128, 1991.

[5] Coffman, E. G.: *Computer and Job-Shop Scheduling Theory*, Wiley, New York, 1976.

[6] Conway, R. W., W. L. Maxwell, and L. W. Miller: *Theory of Scheduling*, Addison-Wesley, Reading, MA, 1967.

[7] Davari, S., and S. K. Dhall: "An On Line Algorithm for Real-Time Tasks Allocation," *Proc. IEEE Real-Time Systems Symp.*, pp. 194–200, IEEE, Los Alamitos, CA, 1986.

[8] Dey, J. K., J. F. Kurose, D. F. Towsley, C. M. Krishna, and M. Girkar: "Efficient On-Line Processor Scheduling for a Class of IRIS Real-Time Tasks," *SIGMETRICS 1993*, pp. 217–228, ACM, New York, 1993.

[9] Johnson, D. S.: *Near-Optimal Bin-Packing Algorithms*, PhD thesis, Massachusetts Institute of Technology, 1974.

[10] Krishna, C. M., and K. G. Shin: "On Scheduling Tasks with a Quick Recovery from Failure," *IEEE Trans. on Computers* C-35:448–455, 1986.

[11] Kurose, J. F., D. F. Towsley, and C. M. Krishna: "Design and Analysis of Processor Scheduling Policies for Real-Time Systems," in *Foundations of Real-Time Computing: Scheduling and Resource Management* (A. van Tilborg and G. Koob, eds.), Kluwer, Boston, pp. 63–90, 1991.

[12] Lehoczky, J. P.: "Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines," *Proc. IEEE Real-Time Systems Symp.*, pp. 201–210, IEEE, Los Alamitos, CA, 1990.

[13] Lehoczky, J. P., L. Sha, and Y. Ding: "The Rate Monotonic Algorithm: Exact Characterization and Average-Case Behavior," *Proc. IEEE Real-Time Systems Symp.*, pp. 166–171, IEEE, Los Alamitos, CA, 1989.

[14] Lehoczky, J. P., L. Sha, and J. K. Strosnider: "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments," *Proc. IEEE Real-Time Systems Symp.*, pp. 261–270, IEEE, Los Alamitos, CA, 1987.

[15] Liestman, A. L., and R. H. Campbell: "A Fault-Tolerant Scheduling Problem," *IEEE Trans. Software Engineering* SE-12:1089–1095, 1986.

[16] Liu, C. L., and J. W. Layland: "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, 20(1):46–61, 1973.

[17] Liu, J. W. S., K. J. Lin, W.-K. Shih, A. C. Yu, J. Y. Chung, and W. Zhao: "Imprecise Computations," *Proc. IEEE* 82:83–94, 1994.

[18] Liu, J. W. S., W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung: "Algorithms for Scheduling Imprecise Computations," in *Foundations of Real-Time Computing: Scheduling and Resource Management* (A. van Tilborg and G. Koob, eds.), Kluwer, Boston, pp. 203–250, 1991.

[19] Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery: *Numerical Recipes in FORTRAN*, Cambridge University Press, Cambridge, 1992.

[20] Ramamritham, K. J., A. Stankovic, and P.-F. Shiah: "Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems," *IEEE Trans. on Parallel and Distributed Systems*, 1:184–194, 1990.

[21] Ramamritham, K. J., A. Stankovic, and W. Zhao: "Distributed Scheduling of Tasks with Deadlines and Resource Requirements," *IEEE Trans. Computers* 38:1110–1123, 1989.

[22] Sha, L., R. Rajkumar, and J. P. Lehoczky: "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. on Computers*, 39:1175–1185, 1990.

[23] Sha, L., R. Rajkumar, and S. S. Sathaye: "Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems," *Proc. IEEE*, 82:68–82, 1994.

[24] Shih, W. K., J. W. S. Liu, and J. Y. Chung: "Algorithms for Scheduling Tasks to Minimize Total Error," *SIAM Journal of Computing* 20:537–552, 1991.

[25] Shin, K. G., and Y.-C. Chang: "Load Sharing in Distributed Real-Time Systems with State-Change Broadcasts," *IEEE Trans. Computers* 38:1124–1142, 1989.

[26] Xu, J.: "Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence, and Exclusion Relations," *IEEE Trans. Software Engineering* 19:139–154, 1993.

[27] Xu, J., and D. L. Parnas: "Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Properties," *IEEE Trans. Software Engineering* 16:360–369, 1990.