In this case, we run the EDF algorithm on the mandatory portions of each task to yield schedule S_m. If this results in an infeasible schedule, then we must stop since we can't execute even the mandatory portions of each task. Suppose that S_m is feasible. Then we adjust S_r to ensure that each task receives at least its mandatory portion of service.

Example 3.34. Consider the set of four tasks with parameters shown in the following table.

Task number	m_i	o_i	r_i	D_i
1	1	4	0	10
2	1	2	1	12
3	3	3	1	15
4	6	2	2	19

In step 1 of the IRIS1 algorithm, we run the EDF algorithm with task execution times 5, 3, 6, and 8, respectively (for tasks 1 to 4), to produce S_{r0} in Figure 3.33. It is impossible to meet the deadline of task 4. Hence, we go to step 2.

Running the EDF algorithm on the task set **M** produces the feasible schedule S_m shown in Figure 3.33. All the deadlines are met, so we can proceed to step 3. We have $a_0 = 0, a_1 = 1, a_2 = 2, a_3 = 5, a_4 = 11$. Also, $k = 4$.

Now we move to step 3 of the algorithm. Let us start with a_3. We have task T_4 scheduled in S_m over the interval $[a_3, a_4]$ and given 6 units of time. In schedule S_{r0}, T_4 is given only 5 units of time. Hence, we modify S_{r0} by adding $6-5=1$ unit of time to T_4 in the interval $[a_3, a_4)$ and taking away 1 unit from the task originally scheduled at a_3 in S_{r0}, namely T_2. This results in task T_4 being scheduled for a total of 6 units beyond a_3. The resulting schedule is S_{r1}. Let us now move to the interval $[a_2, a_3)$. T_3 is scheduled beyond that time in S_m, for a total of 3 units. In S_{r1}, T_3 has been scheduled for a total of 6 units beyond a_2. Therefore, no modifications are needed; T_3 has enough time to meet its mandatory portion. Next, we consider $[a_1, a_2)$. T_2 is scheduled for that interval in S_m. Let us consider the time given to T_2 beyond a_1 in S_{r1}. It is 2 units, which is greater than the mandatory requirement, so no modifications are needed. Finally, consider $[a_0, a_1)$. T_1 is given 1 unit in schedule S_m. In S_{r1}, T_1 is scheduled for 5 units, which exceeds the time given in S_m. So, no modifications are needed, and the optimal schedule is S_{r1}.

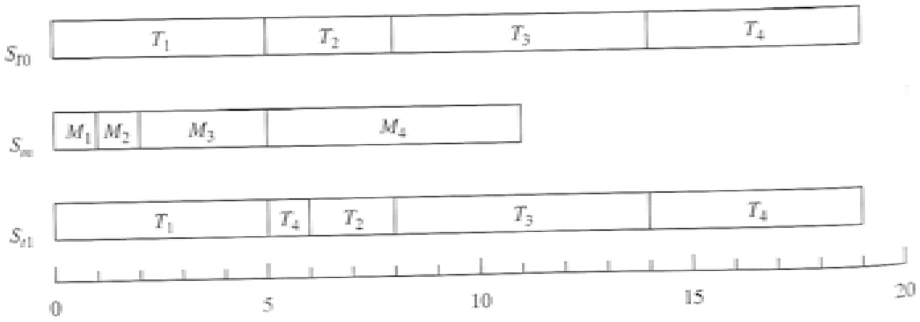

FIGURE 3.33
Schedules produced by algorithm IRIS1 for Example 3.34.

Theorem 3.14. For IRIS tasks with reward functions of the type being considered in this section, algorithm IRIS1 is optimal.

Proof. We leave the formal proof to the reader. Here, we will merely sketch the ideas behind the proof. If a feasible schedule is generated in step 1, then each task has been run to completion and we are done. If not, then from Theorem 3.13, we know that the EDF algorithm is optimal when none of the tasks has any mandatory portion; in that case, the schedule we would obtain would be S_r in step 1. But, the transformations that we do in step 3 do not change the total time for which the processor runs; it only ensures that all the mandatory portions are completed. This completes the proof sketch. Q.E.D.

3.3.2 Nonidentical Linear Reward Functions

The reward function for task T_i is given by

$$R_i(x) = \begin{cases} 0 & \text{if } x \leq m_i \\ w_i(x - m_i) & \text{if } m_i \leq x \leq m_i + o_i \\ w_i o_i & \text{if } x > m_i + o_i \end{cases} \tag{3.84}$$

Each task has a weight w_i associated with it. Assume that the tasks are numbered in nonincreasing order of weights, $w_1 \geq w_2 \geq \ldots \geq w_n$. The procedure for optimally scheduling such tasks is obvious: Always run the available task with the greatest weight, subject to the need to execute the mandatory portions of all the tasks by their respective deadlines. This is done by algorithm IRIS2, which is shown in Figure 3.34.

The idea behind the IRIS2 algorithm is the following. As with IRIS1, we check to see if we can feasibly schedule all the mandatory portions. If not, we stop right away. If we succeed, we proceed by running IRIS1 with a mandatory task set equal to the mandatory portions of the tasks, and the set of optional portions equal only to optional portion of task T_1 (the optional portions of the other tasks are considered not to exist). IRIS1 is executed. It provides as much time as possible to T_1, consistent with the need to meet the mandatory portions of all the tasks.

We now take this schedule and label as mandatory the part of the optional portion of T_1 that was scheduled by IRIS1. Next, we run the IRIS1 algorithm with this revised mandatory portion and the optional portion of T_2, and continue in this way for the remaining tasks.

Theorem 3.15. If the reward functions are as defined in this section, algorithm IRIS2 is optimal.

Proof. Once again, we will leave the formal proof as an exercise and merely provide a brief sketch. We know from Theorem 3.14 that O_n' is the maximum amount of

```
1. Set $\mathbf{M}'$ to be the set of mandatory portions of all the tasks, and $\mathbf{O}' = \emptyset$.
   Run the EDF algorithm, and let $S_m$ be the resulting schedule.

2. If $S_m$ is not feasible,
         the task set $\mathbf{T}$ is not schedulable: STOP.
   else
         $i = 1$
         do while $(1 \leq i \leq n)$
               Set $\mathbf{O}' = \mathbf{O}' \cup \{O_i\}$, and use IRIS1 to find an optimal schedule
               Define $O_i'$ to be the part of $O_i$ scheduled by IRIS1.
               Set $\mathbf{M}' = \mathbf{M}' \cup \{O_i'\}$
               $i = i + 1$
         end do
   end if
end
```

FIGURE 3.34
Algorithm IRIS2. (After Shih, Liu, and Kung [24].)

service that can be given to O_n (which is the task with the greatest weight) if all the mandatory tasks are to meet their deadlines. Similarly, we have that O_{n-1}' is the maximum amount of service that can be given to O_{n-1}, subject to the constraint that all mandatory tasks must meet their deadlines and that as much of O_n as possible should be executed. In general, for $i < n$, we have that O_i' is the maximum amount of service that can be given to O_i, subject to the constraint that all mandatory tasks must meet their deadlines and that as much of $O_{i+1}, \ldots, O_n$ as possible should be executed. The result follows from this observation. Q.E.D.

3.3.3 0/1 Reward Functions

We assume here that for any task i the reward function is given by

$$R_i(x) = \begin{cases} 0 & \text{if } x < m_i + o_i \\ 1 & \text{if } x \geq m_i + o_i \end{cases} \tag{3.85}$$

That is, we get no reward for executing the optional portion partially. If we run the optional portion to completion, we obtain one unit of reward; otherwise we get nothing.

The optimal strategy would therefore be to complete as many optional portions as possible, subject to the constraint that the deadlines of all the mandatory portions must be met. Unfortunately, when the execution times are arbitrary, the problem of obtaining an optimal schedule can be shown to be NP-complete.

Finding an efficient optimal scheduling algorithm under the 0/1 case is therefore a hopeless task. We must therefore make do with heuristics. One rather obvious heuristic is shown in Figure 3.35. The IRIS3 algorithm is based on the following reasoning. Since we get the same reward for completing the optional portion of any task, it is best to run the tasks with the shorter optional portions. Therefore, we assign weights according to the inverse of the duration of the optional portions and run IRIS2. If an optional part is not run to completion in the resulting schedule, we remove its optional portion from consideration and rerun IRIS2. We continue in this manner until each optional portion has been either scheduled to completion or dropped altogether.

```
1. Run the EDF algorithm on the set $\mathbf{M}$ of mandatory tasks.
   If $\mathbf{M}$ is not EDF-schedulable, then
         Task set $\mathbf{T}$ cannot be feasibly scheduled: STOP.
   else
         Go to step 2.
   end if

2. $\mathbf{O}$ is the set of optional portions.
   Assign $w_i = 1/o_i$ for $i = 1, \ldots, n$.
   Renumber the tasks so that their weights are in a non-ascending sequence, i.e.,
   $w_1 \geq w_2 \geq \ldots \geq w_n$.

3. Run algorithm IRIS2 on a task set composed of
         the mandatory set $\mathbf{M}$ and optional set $\mathbf{O}$ to obtain schedule $S_o$.

4. If all the optional tasks in $\mathbf{O}$ are executed to completion in $S_o$,
         Return $S_o$ and STOP.
   else
         Let $i_{\min}$ be the smallest index $i$ such that
               $o_i$ is not run to completion in $S_o$.
         Redefine $\mathbf{O} = \mathbf{O} - \{o_{i_{\min}}\}$.
         Go to step 3.
   end if

end
```

FIGURE 3.35
Algorithm IRIS3: a simple heuristic for the 0/1 case.

3.3.4 Identical Concave Reward Functions (No Mandatory Portions)

In this section, we consider tasks with identical release times and with mandatory portions of zero. We assume that the reward function of T_i is given by

$$R_i(x) = \begin{cases} f(x) & \text{if } 0 \leq x < o_i \\ f(o_i) & \text{if } x \geq o_i \end{cases} \tag{3.86}$$

where the function f is one-to-one and concave. Recall that a function $f(x)$ is concave iff for all x_1, x_2, and $0 \le \alpha \le 1$,

$$f(\alpha x_1 + [1-\alpha]x_2) \ge \alpha f(x_1) + (1-\alpha)f(x_2) \qquad (3.87)$$

Geometrically, this condition can be expressed by saying that, for any two points on a concave curve, the straight line joining them must never be above the curve. An example of a concave function is $1 - e^{-x}$.

We will also assume that the functions $f(x)$ are differentiable, and define $g(x) = df(x)/dx$. We will assume that the inverse function g^{-1} of g exists for all $i = 1, \ldots, n$. This will happen if the functions g are monotonically decreasing and we assume that they are. The tasks are numbered in nondecreasing order of their absolute deadlines (i.e., $D_1 \le D_2 \le \ldots \le D_n$). For notational convenience, define $D_0 = 0$.

Since f is a concave function, we have nonincreasing marginal returns, and so the optimum is obtained by balancing the execution times as much as possible. If all the deadlines are equal (i.e., if $D_1 = \ldots = D_n = \delta$), then the algorithm is trivial—just allocate to each task a total of δ/n of execution time before its deadline. If the deadlines are not all equal, the algorithm is a little more complicated. We will leave to the reader the problem of writing out this algorithm, IRIS4, formally. Here is an informal description.

The basic idea behind this algorithm is to equalize, as much as possible, the execution times of the tasks. The algorithm starts at the latest deadline and works backwards. In the interval $[D_{n-1}, D_n]$, only task T_n can be executed and it is allocated up to $a_n = \max\{D_n - D_{n-1}, e_i\}$ units of time in that interval. Next, we move to the interval $[D_{n-2}, D_{n-1}]$; over this interval, tasks T_{n-1} and T_n can be executed. In this interval, we also allocate time to T_{n-1} and T_n so that, in the interval $[D_{n-2}, D_n]$, the execution time these tasks receive is equalized as much as possible (subject to the obvious constraints). We then go on to the interval $[D_{n-3}, D_{n-2}]$, over which tasks T_{n-2}, T_{n-1}, T_n are available, and so on until the beginning.

Example 3.35. We have a five-task aperiodic system with deadlines $D_1 = 2$, $D_2 = 6$, $D_3 = 8$, $D_4 = 10$, and $D_5 = 20$; each task has an execution time of 8.

Let us begin with the interval (10, 20]. Only task T_5 can be scheduled in that interval, and we can give to it its entire execution time of 8. So, the allocation of execution times so far is:

T_1	T_2	T_3	T_4	T_5
0	0	0	0	8

Next, move to the interval (8, 10]. Tasks T_4, T_5 can be scheduled in that interval, but we have already given a full execution-time to T_5, so we don't consider that task here. We devote this entire interval to T_4. The execution time allocations are now:

T_1	T_2	T_3	T_4	T_5
0	0	0	2	8

Now, consider (6, 8]. T_3, T_4, T_5 are eligible to run in that interval. As before, we don't have to consider T_5. We give the 2 units to T_3 so that it is equalized with T_4. (This is the best possible balancing of the execution times.) The execution-time allocations are now:

T_1	T_2	T_3	T_4	T_5
0	0	2	2	8

Move on to (2, 6]. T_2, T_3, T_4, T_5 are eligible to run in this interval. Give 2 units to T_2 so that T_2, T_3, T_4 are each allocated 2 units. This leaves 2 units which we can allocate equally to each of these tasks over that interval. The execution-time allocations are now:

T_1	T_2	T_3	T_4	T_5
0	2.66	2.66	2.66	8

Finally, consider (0, 2]. Here, we must clearly allocate 2 units to T_1, and the final allocations are:

T_1	T_2	T_3	T_4	T_5
2	2.66	2.66	2.66	8

It is easy to check that the execution times have been balanced as much as possible, under deadline and execution time constraints. The schedule is shown in Figure 3.36.

Theorem 3.16. Algorithm IRIS4 is optimal under the conditions listed in Section 3.3.4.

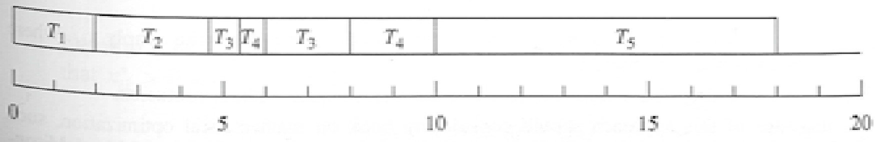

FIGURE 3.36
Example of the IRIS4 algorithm.

Proof. This has been left as an exercise for the reader. Q.E.D.

3.3.5 Nonidentical Concave Reward Functions*

As in the previous section, we consider tasks with identical release times and with mandatory portions of zero. There are n tasks in all. We assume that for any task T_i the reward function is given by

$$R_i(x) = \begin{cases} f_i(x) & \text{if } 0 \leq x < o_i \\ f_i(o_i) & \text{if } x \geq o_i \end{cases} \tag{3.88}$$

where the functions f_i are one-to-one and concave.

Let $x_{ij}(S)$ denote the service that task T_i receives in the interval $[D_{j-1}, D_j]$ under schedule S.[6] The total amount of service that task T_i receives is given by $s_S(i) = \sum_{j=1}^{i} x_{ij}(S)$.[7] The optimization problem therefore reduces to maximizing

$$P = \sum_{i=1}^{n} f_i(s(i)) \tag{3.89}$$

subject to the constraints that

$$\sum_{i=1}^{n} x_{ij} = D_j - D_{j-1} \qquad 1 \leq j \leq n \tag{3.90}$$

$$x_{ij} \geq 0 \qquad 1 \leq j \leq i \leq n \tag{3.91}$$

This is a standard constrained-maximization problem, which can be solved using Lagrange multipliers.[8] The solution to this optimization problem can be obtained by solving the following set of equations, where μ_j and v_{ij} are Lagrange multipliers.

$$-g_i(s(i)) + \mu_j + v_{ij} = 0 \qquad 1 \leq j \leq i \leq n \tag{3.92}$$

$$\sum_{i=j}^{n} x_{ij} - (D_j - D_{j-1}) = 0 \qquad j = 1, \ldots, n \tag{3.93}$$

$$x_{ij} \geq 0, \quad v_{ij}x_{ij} = 0, \quad v_{ij} \leq 0 \qquad 1 \leq j \leq i \leq n \tag{3.94}$$

[6] In this discussion, we will frequently omit S where this is convenient and write simply x_{ij}, where S can be understood from the context.

[7] As with x_{ij}, we will drop the S from $s_S(i)$, where this can be done without confusion.

[8] Readers unaware of this approach should consult any book on mathematical optimization, such as D. G. Luenberger, *Introduction to Linear and Nonlinear Programming*, Reading, MA: Addison-Wesley, 1973. If you haven't heard of Lagrange multipliers before, simply consider them to be constants and assume (3.92) to (3.94) to be true.

If $x_{ij} = 0$, from (3.94), $v_{ij} \leq 0$, and so from Equation (3.92),

$$g_i(s(i)) \leq \mu_j \qquad 1 \leq j \leq i \leq n \tag{3.95}$$

If $x_{ij} > 0$, we have $v_{ij} = 0$, and so from Equation (3.92),

$$g_i(s(i)) = \mu_j \qquad 1 \leq j \leq i \leq n \tag{3.96}$$

Let $s^*(i)$ and x^*_{ij} denote values for $s(i)$ and x_{ij} $(i, j = 1, \cdots, n)$ that satisfy Equations (3.92) to (3.94). Then an examination of these equations yields the following conclusions.

Lemma 3.13. For any i, j, if there is some k $(1 \leq k \leq i, j \leq n)$ such that $x^*_{ik} > 0$ and $x^*_{jk} > 0$, then

$$g_i(s^*(i)) = g_j(s^*(j)) \tag{3.97}$$

Proof. This follows immediately from Equation (3.96). Q.E.D.

Proof. An alternative proof that argues from first principles is as follows. Suppose that the lemma is false, and that we have some i, j such that $g_i(s^*(i)) \neq g_j(s^*(j))$ for optimal schedule S^*. Consider the case $g_i(s^*(i)) < g_j(s^*(j))$. Since $x^*_{ik}, x^*_{jk} > 0$ and the g_i are continuous functions, there exists some $\delta > 0$ such that

- $\delta \leq \min\{x^*_{ik}, x^*_{jk}\}$, and
- $g_i(s^*(i) - \delta) < g_j(s^*(j) + \delta)$.

Construct another schedule S' that is identical to S^* except that task T_i receives δ less service and task T_j receives δ more service. It is clearly possible to do this without any deadlines being missed. Denote the rewards under S^* and S' by $R(S^*)$ and $R(S')$, respectively. Then,

$$\begin{aligned} R(S') - R(S^*) &= f_i(s^*(i) - \delta) - f_i(s^*(i)) + f_j(s^*(j) + \delta) - f_j(s^*(j)) \\ &\geq -\delta\left(\max_{x \in [s^*(i)-\delta, s^*(i)]} g_i(x)\right) + \delta\left(\min_{x \in [s^*(j), s^*(j)+\delta]} g_j(x)\right) \\ &= -g_i(s^*(i) - \delta)\delta + g_j(s^*(j) + \delta)\delta \\ &> 0 \end{aligned} \tag{3.98}$$

A similar result holds for the case where $g_i(s^*(i)) > g_j(s^*(j))$.

The total reward for S' will be thus greater than that for S^*, contradicting the optimality of S^*. Q.E.D.

Lemma 3.14. For any $i, j \in \{1, \ldots, n\}$, if there is some k $(1 \leq k \leq i, j \leq n)$ such that $x^*_{ik} > 0$ and $x^*_{jk} = 0$, then

$$g_i(s^*(i)) \geq g_j(s^*(j)) \tag{3.99}$$

Proof. This follows immediately from Equations (3.95) and (3.96). Q.E.D.

Lemma 3.15. If $s^*(j) = 0$ for any $j \in \{1, \ldots, n\}$, then for all $i \in \{1, \ldots j-1\}$,

$$g_i(s^*(i)) \geq g_j(0) \tag{3.100}$$

Proof. $s^*(j) = 0$ means that $x^*_{jk} = 0$ for all k. This, together with Lemma 3.14, proves the result. **Q.E.D.**

Lemma 3.16. If there is some task T_k such that $x^*_{ki} > 0$ and $x^*_{kj} > 0$, then

$$\mu_i = \mu_j \qquad \text{for all } 1 \leq i, j \leq k \leq n \tag{3.101}$$

Proof. The proof is an immediate consequence of Equation (3.96). **Q.E.D.**

Lemma 3.17. $\mu_i \geq \mu_{i+1}$ for $1 \leq i < n$.

Proof. We prove this by contradiction. Suppose that this lemma is false, and there exists some i such that $\mu_i < \mu_{i+1}$. Then, from Lemma 3.16, we know that there is some task T_m such that $x^*_{m,i+1} > 0$ but $x^*_{m,i} = 0$. But, $x^*_{m,i} = 0$ implies from Equation (3.95) that $g_m(s^*(m)) \leq \mu_i$, and $x^*_{m,i+1} > 0$ implies from Equation (3.96) that $g_m(s^*(m)) = \mu_{i+1}$. That is, $\mu_i \geq \mu_{i+1}$, a contradiction. **Q.E.D.**

Lemma 3.18. There exists an optimal schedule S under which for all i such that $s_S(i) > 0$,

$$g_i(s_S(i)) \geq g_j(s_S(j)) \qquad \text{for all } 1 \leq i < j \leq n \tag{3.102}$$

Proof. We prove this result by construction. That is, we show that any optimal schedule can be transformed into another optimal schedule for which Equation (3.102) holds.

Suppose we are given an optimal schedule U and wish to transform it to another schedule Y for which Equation (3.102) holds. Take tasks T_i and T_j, with $i < j$ such that $s_U(i), s_U(j) > 0$. Define $v = \max\{k | x_{ik}(U) > 0\}$ and $w = \max\{k | x_{jk}(U) > 0\}$. In words this means that tasks T_i and T_j do not, under U, receive any service after time D_{v+1} and D_{w+1}, respectively. There are three cases.

Case 1. $v = w$. From Lemma 3.13, we know that $g_i(s_U(i)) = g_j(s_U(j))$. Define $x_{ik}(Y) = x_{ik}(U)$ and $x_{jk}(Y) = x_{jk}(U)$ for all $k \in \{1, \ldots, n\}$.

Case 2. $v < w$. From Equation (3.96) and Lemma 3.17, we have $g_i(s_U(i)) \geq g_j(s_U(j))$. Define $x_{ik}(Y) = x_{ik}(U)$ and $x_{jk}(Y) = x_{jk}(U)$, for all $k \in \{1, \ldots, n\}$.

Case 3. $v > w$. By shifting the execution of the tasks in time (while keeping the total time allocated to each task the same in both schedules U and Y), we can reduce Case 3 to either Case 1 or Case 2. In particular, in Schedule Y we have:

$$\begin{aligned}
x_{iv}(Y) &= x_{iv}(U) - \min(x_{iv}(U), x_{jw}(U)) \\
x_{jv}(Y) &= x_{jv}(U) + \min(x_{iv}(U), x_{jw}(U)) \\
x_{iw}(Y) &= x_{iw}(U) + \min(x_{iv}(U), x_{jw}(U)) \\
x_{jw}(Y) &= x_{jw}(U) - \min(x_{iv}(U), x_{jw}(U))
\end{aligned}$$

We are shifting some of the T_i execution from $(D_{v-1}, D_v]$ to $(D_{w-1}, D_w]$, and some of the T_j execution from $(D_{w-1}, D_2]$ to $(D_{v-1}, D_v]$.

If $x_{iv}(Y) > 0$, then we have

$$\max\{k | x_{ik}(Y) > 0\} = \max\{k | x_{jk}(Y) > 0\} = v$$

and Case 1 can now be applied. On the other hand, if $x_{iv}(Y) = 0$, then

$$\max\{k | x_{ik}(Y) > 0\} < \max\{k | x_{jk}(Y) > 0\} = v$$

and Case 2 can be applied to tasks T_i and T_j.

Thus, by repeatedly applying this construction procedure to every pair of tasks T_i, T_j for which $s_U(i) > 0$, $s_U(j) > 0$, we obtain the schedule Y. **Q.E.D.**

Assuming that the tasks are all released at time 0 and have deadlines $D_1 \leq D_2 \leq \ldots \leq D_n$, we can define n scheduling problems, $q_1, q_2, \ldots, q_n$, where q_i is the following problem (for notational convenience, assume $D_0 = 0$):

> Assuming that tasks $T_i, \ldots, T_n$ all arrive at time D_{i-1}, schedule them in the interval $[D_{i-1}, D_n]$ so that the reward is maximized.

The overall scheduling problem is therefore q_1. Solving q_n is trivial; as only task n can be scheduled in the interval $(D_{n-1}, D_n]$. We will show now how to solve q_i as a function of the solution of q_{i+1}.

Theorem 3.17. Let an optimal solution of q_{i+1} involve allocating service time $s^*_{i+1}(j)$ to task T_j $(i+1 \leq j \leq n)$. Let K be the set of tasks that receives a nonzero allocation of service time in the interval $[D_{i-1}, D_i]$ in the optimal solution to q_i. Then, an optimal solution to q_i satisfies the equation

$$\sum_{k \in K} g_k^{-1}(\mu^{(i)}) = \sum_{k \in K} s^*_{i+1}(k) + D_i - D_{i-1} \tag{3.103}$$

where $\mu^{(i)} = \mu_k^{(i)}$ for all $k \in K$.

Proof. The server does not idle while there are tasks waiting for service. Consequently,

$$\sum_{k \in K} \left(s^*_i(k) - s^*_{i+1}(k)\right) = D_i - D_{i-1}$$

$$\Rightarrow \sum_{k \in K} s^*_i(k) = \sum_{k \in K} s^*_{i+1}(k) + D_i - D_{i-1} \tag{3.104}$$

But from Lemmas 3.13 and 3.14 we know that all tasks served in the interval $(D_{i-1}, D_i]$ have the same marginal reward rate, and that other tasks in q_i have lower marginal reward rates. That is, there exists some $\mu^{(i)}$ such that $g_k(s^*_i(k)) = \mu^{(i)}$ for all $k \in K$. Thus, we have from Equation (3.104),

$$\sum_{k \in K} g_k^{-1}\left(\mu^{(i)}\right) = \sum_{k \in K} s^*_{i+1}(k) + D_i - D_{i-1} \tag{3.105}$$

Q.E.D.

We now hold all the keys to an optimal scheduling algorithm. As mentioned earlier, the solution of q_n is trivial and we will work backwards through $q_{n-1}, \ldots, q_1$. Suppose we have solved problem q_{i+1}. In the solution of q_i, we consider the set of tasks $\mathcal{T}_i = \{T_i, \ldots, T_n\}$. For notational convenience, define

- $s^*_{i+1}(j) = 0$ for all $j \leq i$, and
- $\pi(j) = \{x | g_x(s^*_{i+1}(x))$ is the jth largest of the set

$$\{g_x(s^*_{i+1}(i)), \ldots, g_x(s^*_{i+1}(n))\}\}.$$

From the foregoing results, we know that tasks $T_{\pi(1)}, \ldots, T_{\pi(y)}$ will be served in $(D_{i-1}, D_i]$ if it is possible to find some $\hat{\mu} \geq 0$ such that

$$\sum_{j=1}^{y} g^{-1}_{\pi(j)}(\hat{\mu}) = D_i - D_{i-1} + \sum_{j=1}^{y} s^*_{i+1}(j) \qquad (3.106)$$

This leaves us with the problem of obtaining y and $\hat{\mu}$. The brute-force way of doing this is to try every value of i from 1 to y, where Equation (3.106) no longer allows $\hat{\mu} > 0$. The clever(er) way of doing this is to observe that if task k is served in $(D_i, D_{i+1}]$, then $s^*_i(k) > s^*_{i+1}(k)$. In any event, for all tasks T_j such that $D_j > D_i$ we must have $s^*_i(j) \geq s^*_{i+1}(j)$, since only such tasks can be served beyond D_i. The complete algorithm is shown in Figure 3.37. Concave reward functions are probably the most realistic since they exhibit the property of nonincreasing marginal returns. The greatest gains in the accuracy of most numerical iterative algorithms, for example, come in the first few moments of execution.

```
1.  $L = \emptyset$, $x_i = 0$, $i = 1, \ldots, n$. $D_0 = 0$. $m = n$.
2.  while ($m > 0$) do
        Insert task $T_m$ into $L$.
        Define $\pi(i) = \{\alpha | g_\alpha(x_\alpha)$ is the $i$th largest among $g_\ell(x_\ell), \ell \in L\}$.
        Use binary search to find $\ell$ such that
        $\sum_{i=1}^{\ell+1} \left[ g^{-1}_{\pi(i)}(g_{\ell+1}(x_{\ell+1}) - x_{\pi(i)} \right] > D_m - D_{m-1} \geq \sum_{i=1}^{\ell} \left[ g^{-1}_{\pi(i)}(g_\ell(x_\ell)) - x_{\pi(i)} \right]$
        Solve for $\mu$ in the equation
        $\sum_{i=1}^{\ell} g^{-1}_{\pi(i)}(\mu) = \sum_{i=1}^{\ell} x_{\pi(i)} + D_m - D_{m-1}$
        We have $x_{\pi(i)} = g^{-1}_{\pi(i)}(\mu)$, $i = 1, \ldots \ell$.
        $m = m - 1$
    end while

end
```

FIGURE 3.37
Algorithm IRIS5.

3.4 TASK ASSIGNMENT

The optimal assignment of tasks to processors is, in almost all practical cases, an NP-complete problem. We must therefore make do with heuristics. These heuristics cannot guarantee that an allocation will be found that permits all tasks to be feasibly scheduled. All that we can hope to do is to allocate the tasks, check their feasibility, and, if the allocation is not feasible, modify the allocation to try to render its schedules feasible.

Heuristics typically allocate according to some simple criterion and hope that feasibility will follow as a side effect of that criterion. For example, if we keep the utilization below $n(2^{1/n} - 1)$ for all processors in a system running periodic tasks whose deadlines equal the respective periods, we know that the resulting task allocation is RM-feasible.

When checking an allocation for feasibility, we must account for communication costs. For example, suppose that $T_1 \prec T_2$. Task T_2 cannot start before receiving the task T_1 output. That is, if f_i denotes the completion time of task T_i and c_{ij} is the time to communicate from T_i to T_j,

$$r_2 \geq f_1 + c_{12} \qquad (3.107)$$

If tasks T_1 and T_2 are allocated to the same processor, then $c_{12} = 0$. If they are allocated to separate processors, c_{12} is positive and must be taken into account while checking for feasibility.

Example 3.36. Consider the situation discussed above where $\prec(2) = \{1\}$. Then, if $D_2 < f_1 + c_{12} + e_2$, the allocation is not feasible.

Sometimes an allocation algorithm uses communication costs as part of its allocation criterion.

3.4.1 Utilization-Balancing Algorithm

This algorithm attempts to balance processor utilization, and proceeds by allocating the tasks one by one and selecting the least utilized processor. The algorithm is shown in Figure 3.38.

This algorithm takes into account the possibility that we might wish to run multiple copies of the same task simultaneously for fault-tolerance. In particular, it assigns r_i copies of task T_i to separate processors. Let u^*_i and u^B_i denote the utilizations of processor p_i under an optimal algorithm that minimizes the sum of the squares of the processor utilizations and under the best-fit algorithm, respectively. If $r_1 = \ldots = r_n = r$, and there are p processors in all, it is possible to show that

$$\frac{\sum_{i=1}^{p} (u^B_i)^2}{\sum_{i=1}^{p} (u^*_i)^2} \leq \frac{9}{8} \frac{p}{p - r + 1} \qquad (3.108)$$

If $p \gg r$, this ratio tends to 1.125, which is agreeably small.

```
1. For each task $T_i$, do
        Allocate one copy of the task to each of the $r_i$ least utilized processors.
        Update the processor allocation to account for the allocation of task $T_i$.
   end do
end

(where $r_i$ is the redundancy, i.e., the number of copies of task $i$ that must be scheduled.)
```

FIGURE 3.38
Utilization-balancing algorithm.

3.4.2 A Next-Fit Algorithm for RM Scheduling

There is a utilization-based allocation heuristic that is meant specifically to be used in conjunction with the rate-monotonic scheduling algorithm. The task set has the properties that we assumed in Section 3.2.1 on RM scheduling (i.e., independence, preemptibility, and periodicity). The multiprocessor is assumed to consist of identical processors and tasks are assumed to require no resources other than processor time. Define $M > 3$ classes as follows, where M is picked by the user. Task T_i is in class $j < M$ if

$$2^{1/(j+1)} - 1 < e_i/P_i \leq 2^{1/j} - 1 \tag{3.109}$$

and in class M otherwise. Corresponding to each task class is a set of processors that is only allocated the tasks of that class.

We allocate tasks one by one to the appropriate processor class until all the tasks have been scheduled, adding processors to classes if that is needed for RM-schedulability. Example 3.37 clarifies this process.

Example 3.37. Suppose we have $M = 4$ classes. Then the following table lists the utilization bounds corresponding to each class.

Class	Bound
C_1	(0.41,1]
C_2	(0.26,0.41]
C_3	(0.19,0.26]
C_4	(0.00,0.19]

Consider the following periodic task set.

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}
e_i	5	7	3	1	10	16	1	3	9	17	21
P_i	10	21	22	24	30	40	50	55	70	90	95
$u(i)$	0.50	0.33	0.14	0.04	0.33	0.40	0.02	0.05	0.13	0.19	0.22
Class	C_1	C_2	C_4	C_4	C_2	C_2	C_4	C_4	C_4	C_4	C_3

Note: $u(i) = e_i/P_i$

Since we have at least one task in each of the four classes, let us begin by earmarking one processor for each class. In particular, let processor p_i be reserved for tasks in class C_i, $1 \leq i \leq 4$. T_1 is assigned to p_1, T_2 to p_2, and T_3 to p_4. $T_4 \in C_4$, and since $\{T_3, T_4\}$ is RM-schedulable on the same processor, we assign T_4 also to p_4. $T_5 \in C_2$, and since $\{T_2, T_5\}$ is RM-schedulable on the same processor, we assign T_5 also to p_2. $T_6 \in C_2$. However, $\{T_2, T_5, T_6\}$ is not RM-schedulable on the same processor, so we assign an additional processor p_5 to C_2 tasks and assign T_6 to p_5. $T_7 \in C_4$ and $\{T_3, T_4, T_7\}$ is RM-schedulable on the same processor, so we assign it to p_4. We proceed similarly for T_8, T_9, T_{10}. Finally, assign T_{11} to p_3. The assignments are summarized below.

Processor	Tasks
p_1	T_1
p_2	T_2, T_5
p_3	T_{11}
p_4	$T_3, T_4, T_7, T_8, T_9, T_{10}$
p_5	T_6

With this assignment, we can run the RM scheduling algorithm on each processor.

It is possible to show that this approach uses no more than N times the minimum possible number of processors, where

$$N = \begin{cases} 1.911 & \text{if there is no task with utilization in } (\sqrt{2} - 1, 0.5] \\ 2.340 & \text{otherwise} \end{cases} \tag{3.110}$$

3.4.3 A Bin-Packing Assignment Algorithm for EDF

Suppose we have a set of periodic independent preemptible tasks to be assigned to a multiprocessor consisting of identical processors. The task deadlines equal their periods. Other than processor time, tasks require no other resources.

We know that so long as the sum of the utilizations of the tasks assigned to a processor is no greater than 1, the task set is EDF-schedulable on that processor. So, the problem reduces to making task assignments with the property that the sum of the utilizations of the tasks assigned to a processor does not exceed 1.

```
Initialize i to 1. Set $U(j) = 0$, for all $j$.
while $i \leq n_T$ do
    Let $j = \min\{k | U(k) + u(i) \leq 1\}$.
    Assign the $i$th task in $L$ to $p_j$.
    $i \leftarrow i + 1$.
end while
```

FIGURE 3.39
First-fit decreasing algorithm.

We would like to minimize the number of processors needed. This is the famous bin-packing problem and many algorithms exist for solving it.

The algorithm we present here is the first fit decreasing algorithm. Suppose there are n_T tasks to be assigned. Prepare a sorted list L of the tasks so that their utilizations (i.e., $u(i) = e_i/P_i$) are in nonincreasing order. Figure 3.39 shows the algorithm.

Example 3.38. Consider the following task set:

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}
e_i	5	7	3	1	10	16	1	3	9	17	21
P_i	10	21	22	24	30	40	50	55	70	90	95
$u(i)$	0.50	0.33	0.14	0.04	0.33	0.40	0.02	0.05	0.13	0.19	0.22

Note: $u(i) = e_i/P_i$

The ordered list is $L = (T_1, T_6, T_2, T_5, T_{11}, T_{10}, T_3, T_9, T_8, T_4, T_7)$. The assignment process is summarized in the following table. The vector $\mathbf{U} = (U_1, U_2, U_3, \ldots)$ contains the total utilizations of processor p_i in U_i.

Step	Task T_i	$u(i)$	Assigned to	Post-assignment U vector
1	T_1	0.50	p_1	(0.50)
2	T_6	0.40	p_1	(0.90)
3	T_2	0.33	p_2	(0.90,0.33)
4	T_5	0.33	p_2	(0.90,0.66)
5	T_{11}	0.22	p_2	(0.90,0.88) 9
6	T_{10}	0.18	p_3	(0.90,0.88,0.18)
7	T_3	0.14	p_3	(0.90,0.88,0.32)
8	T_9	0.13	p_3	(0.90,0.88,0.45)
9	T_8	0.06	p_1	(0.96,0.88,0.45)
10	T_4	0.04	p_1	(1.00,0.88,0.45)
11	T_7	0.02	p_2	(1.00,0.90,0.45)

It is possible to show that when the number of processors required is large, the ratio

$$\frac{\text{Number of processors used by the first-fit decreasing algorithm}}{\text{Number of processors used by optimal algorithm}}$$

approaches $11/9 = 1.22$, when a large task set is used. In fact, this limit is approached quickly, so that 1.22 is a good measure even for relatively small systems.

3.4.4 A Myopic Offline Scheduling (MOS) Algorithm

Thus far, we have assumed that tasks can be preempted. The *myopic offline scheduling* (MOS) heuristic is an assignment/scheduling algorithm meant for non-preemptive tasks. This algorithm takes account not only of processing needs but also of any requirements that tasks may have for additional resources. For instance, a task may need to have exclusive access to a block of memory or may need to have control over a printer. MOS is an offline algorithm in that we are given in advance the entire set of tasks, their arrival times, execution times, and deadlines.

MOS proceeds by building up a schedule tree. Each node in this tree represents an assignment and scheduling of a subset of the tasks. The root of the schedule tree is an empty schedule. Each child of a node consists of the schedule of its parent node, extended by one task. A leaf of this tree consists of a schedule (feasible or infeasible) of the entire task set.

The schedule tree for an n_T-task system consists of $n_T + 1$ levels (including the root). Level i of the tree (counting the root as being of level 0) consists of nodes representing schedules including exactly i of the tasks.

Generating the complete tree is tantamount to an exhaustive enumeration of all possible allocations. For any but the smallest systems, it is therefore not practical to generate the complete tree; instead, we try to get to a feasible schedule as quickly as we can.

The algorithm can be informally described as follows. We start at the root node, which is an empty schedule; that is, it corresponds to no task having been scheduled. We then proceed to build the tree from that point by developing nodes. A node n is developed as follows. Given a node n, we try to extend the schedule represented by that node by one more task. That is, we pick up one of the as-yet-unscheduled tasks and try to add it to the schedule represented by node n. The augmented schedule is a child node of n.

There are two questions that must be answered. First, which task do we pick for extending an incomplete schedule? Second, when do we decide that a node is not worth developing further and turn to another node?

1. The task that we chose to extend an incomplete schedule is one that minimizes a heuristic function H. H may be any of the following functions:
   - task execution time,

- deadline,
- earliest start time (i.e., earliest time at which the resources for that task will become available after it has been released),
- laxity,[9] or
- weighted sum of any of the above.

For instance, if $H(i) = D_i$, then the next task to be chosen for scheduling will be the as-yet-unscheduled task with the earliest deadline.

2. We only develop a node if it is strongly feasible. A node is *strongly feasible* if a feasible schedule can be generated by extending the current partial schedule with any one of the as-yet-unscheduled tasks. If a node is not strongly feasible, it means that none of its descendants that are leaves can represent a feasible schedule. If we encounter a node that is not strongly feasible, we backtrack. That is, we mark that node as hopeless, and then go back to its parent, resuming the schedule-building from that point.

One difficulty with the MOS algorithm is that, if the number of tasks is very large, it can take a long time to check if a node is strongly feasible. In particular, at level i, we will need to check feasibility of extending the schedule by each of the $n_T - i$ as-yet-unscheduled tasks. As a result, the number of comparisons needed to generate one root-to-leaf path is

$$n_T + (n_T - 1) + (n_T - 2) + \cdots + 0 = \frac{n_T(n_T + 1)}{2}$$

To reduce the number of comparisons, we can replace the strong feasibility check at each node by means of a myopic procedure as follows. For each nonleaf level-i node n, this procedure picks the first $\min\{k, n_T - i\}$ as-yet-unscheduled tasks and checks to see if the schedule represented by n can be feasibly extended by each of these tasks. (The parameter k is used by the algorithm to limit the scope of the search.) If not, we mark the node as hopeless and backtrack as before. Otherwise, we develop children for that node.

Example 3.39. We have a five-task set to be scheduled on a two-processor system. The tasks are nonpreemptive. The parameters of these tasks are as follows:

	T_1	T_2	T_3	T_4	T_5
r_i	0	10	0	15	0
e_i	15	5	16	9	10
D_i	15	20	18	25	50

[9] The laxity of task T_i is given by $D_i - e_i$. It is the latest time at which T_i may be started and be guaranteed to meet its deadline.

There are no other resource requirements. Suppose we use $H(i) = r_i$. We set $k = 5$ for the myopic procedure. The tree generated by the algorithm is shown in Figure 3.40.

The root node is the empty schedule. There are three tasks with release times of 0; we pick T_1 first. A level-1 node is generated, that contains a schedule for T_1. This node is strongly feasible—any of the other tasks can be feasibly scheduled given the position that T_1 occupies in the schedule.

Next, we pick T_3 and schedule it to form a level-2 node. This, too, is strongly feasible. Then, we generate a level-3 node, which involves augmenting the previous schedule with T_5. Unfortunately, this is not strongly feasible; in particular, it would be impossible to augment this schedule with T_2. So, we backtrack to the level-2 (i.e., the parent) node. We pick T_2 rather than T_5 (the next task in order of release time) and schedule it. This results in a strongly feasible schedule.

Next, we form a level-5 node by adding T_5 to the schedule. This is not strongly feasible—T_4 cannot be added to it. So, we abandon this node, return to the parent (level-4) node, and generate a schedule by adding T_4. This is strongly feasible, and its child, formed by adding the final task to it, is a leaf node that represents a feasible scheduling of all the tasks.

The reader should run the algorithm on this set of tasks with $H(i) = D_i$ and see if it runs any faster for that function.

The running time of the algorithm depends on k and H. No definitive statements can be made about how to choose these quantities. Let us examine k. This bounds the number of tasks that the algorithm considers in determining the strong feasibility of a node. If k is too small, it is possible for us to declare a node to be strongly feasible and develop it further, only to find that none of its descendants is strongly feasible. If k is too large, we will spend a great deal of time (especially in the levels of the tree close to the root) checking the strong feasibility of nodes. In general, the tighter the constraints, the greater must be the value of k. In other words, if the task laxities are low or if many tasks use resources in addition to the processor, k must be large. It has been suggested, from extensive simulations, that $k \approx 13$ is the largest value ever required.

As far as H is concerned, a weighted sum of the deadline and earliest start time is perhaps the most promising function. Recall that the earliest start time of a task is the earliest time after the task has been released that all the nonprocessor resources needed by that task become available.

3.4.5 Focused Addressing and Bidding (FAB) Algorithm

The focused addressing and bidding (FAB) algorithm is simple enough to be an online procedure and is used for task sets consisting of both critical and noncritical real-time tasks. Critical tasks must have sufficient time reserved for them so that they continue to execute successfully, even if they need their worst-case execution times. The noncritical tasks are either processed or not, depending on the system's ability to do so.

The underlying system model is as follows. The noncritical tasks arrive at individual processors in the multiprocessor system. If a noncritical task arrives at

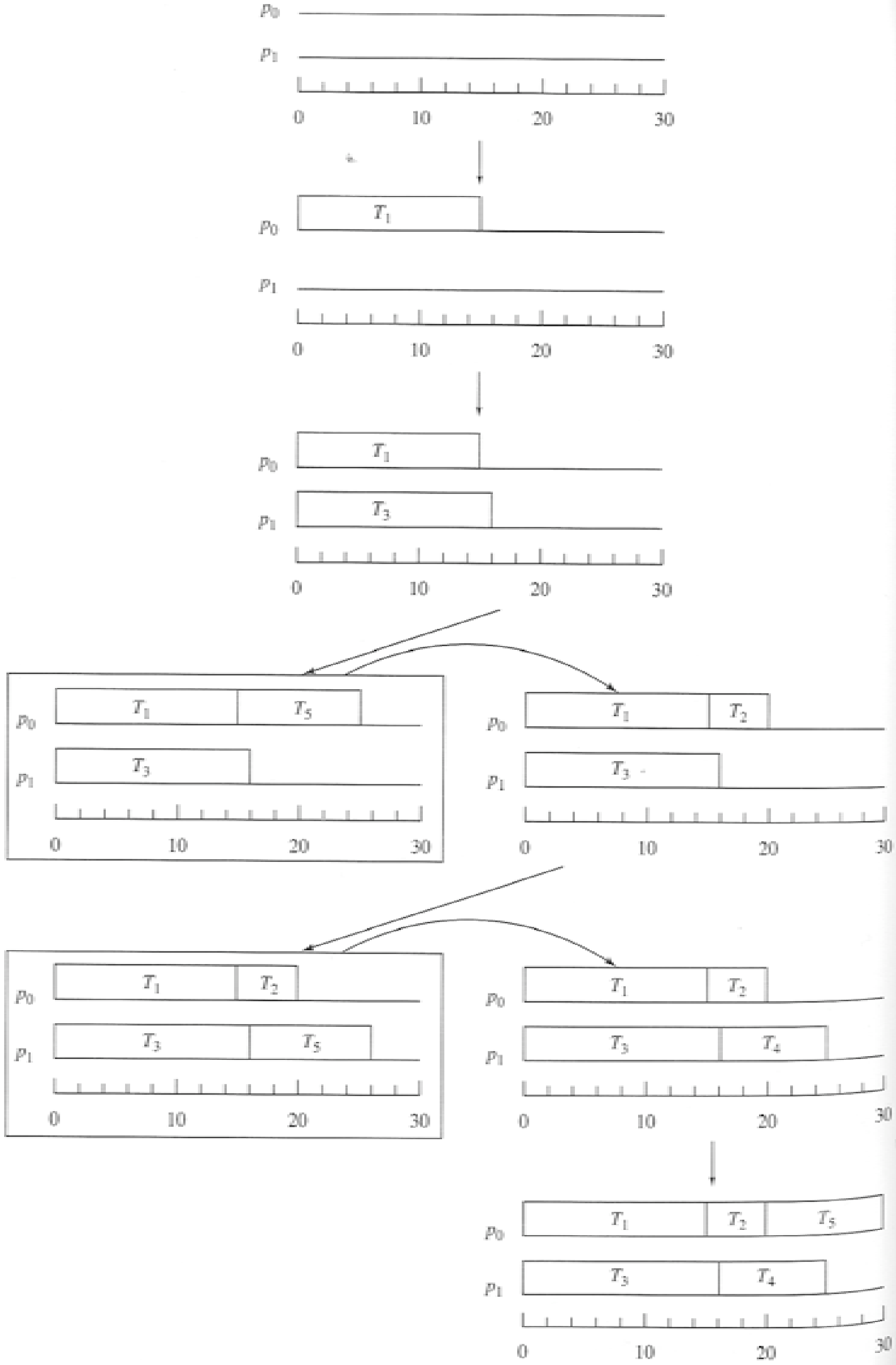

FIGURE 3.40
Example of the MOS algorithm; boxed nodes are not strongly feasible.

processor p_i, that processor checks to see if it expects to have the resources and time to execute it by the specified deadline[10] without missing any of the deadlines of the critical tasks or of the previously guaranteed noncritical tasks. If it does, p_i guarantees the successful execution of that task, adds that task to its list of tasks to be executed, and reserves time on its schedule to execute that task. Since this is a noncritical task, the guarantee can be based on the expected run time of the task rather than on the worst-case run time. In other words, we can accept that some noncritical tasks might turn out to be not executable in a timely fashion because their actual run times turn out to be much greater than anticipated.

The FAB algorithm is used when p_i determines that it does not have the resources or time to execute the task. In that case, it tries to ship that task out to some other processor in the system.

The problem of load-sharing by moving tasks from one processor to another has long been studied in general-purpose distributed systems. Many solutions have been suggested. Perhaps the simplest is a random-threshold algorithm. In this algorithm, a processor that finds its load exceeding a threshold simply sends an incoming task out to another processor, chosen at random. Another algorithm has lightly loaded processors touting for business by announcing they are lightly loaded and are willing to process excess tasks from other processors. We shall see a variant of this (adapted for real-time purposes) when we study the buddy algorithm.

The FAB algorithm is as follows. Each processor maintains a status table that indicates which tasks it has already committed to run. These include the set of critical tasks (which were preassigned statically), and any additional noncritical tasks that it may have accepted. In addition, it maintains a table of the surplus computational capacity at every other processor in the system. The time axis is divided into *windows*, which are intervals of fixed duration, and each processor regularly sends to its colleagues the fraction of the next window that is currently free (i.e., is not already spoken for by tasks). Since the system is distributed, this information may never be completely up to date.

When shopping for a processor on which to offload a task, an overloaded processor checks its surplus information and selects a processor (called the *focused processor*) p_s that it believes to be the most likely to be able to successfully execute that task by its deadline. It ships the task out to that processor. However, as we pointed out, the surplus information may have been out of date and it is possible that the selected processor will not have the free time to execute the task. As insurance against this, and in parallel with sending out the task to the focused processor p_s, the originating processor decides whether to send out requests for bids (RFB) to other lightly loaded processors. The RFB contains the vital statistics of the task (its expected execution time, any other resource requirements, its deadline, etc.), and asks any processor that can successfully execute the task to send a bid to the focused processor p_s stating how quickly it can process the task.

[10] Recall that in a real-time system, the resource and execution-time requirements of all the tasks are known in advance.