Lemma 3.12. Suppose T is not feasible and $u \leq 1$. Then $h_T(t) > t$ implies

$$t < d_{\max} \quad \text{or} \quad t < \max_{1 \leq i \leq n}\{P_i - d_i\}\frac{u}{1-u}$$

Proof. Suppose that $t > d_{\max}$. We have

$$\begin{aligned} h_T(t) &\leq \sum_{i=1}^{n} e_i \frac{t - d_i + P_i}{P_i} \\ &= t\sum_{i=1}^{n}\frac{e_i}{P_i} + \sum_{i=1}^{n}\frac{P_i - d_i}{P_i}e_i \\ &\leq \sum_{i=1}^{n}\left[\frac{e_i}{P_i}\left(t + \max_{1 \leq i \leq n}\{P_i - d_i\}\right)\right] \end{aligned} \tag{3.55}$$

If $h_T(t) > t$, we will have from (3.55),

$$\begin{aligned} t &< \sum_{i=1}^{n}\frac{e_i}{P_i}\left(t + \max_{1 \leq i \leq n}\{P_i - d_i\}\right) \\ \Rightarrow t &< \max_{1 \leq i \leq n}\{P_i - d_i\}\frac{u}{1-u} \end{aligned} \tag{3.56}$$

Q.E.D.

3.2.3 Allowing for Precedence and Exclusion Conditions*

We have assumed in the above sections that tasks are independent and are always preemptible by other tasks. We will now relax both these assumptions and present several scheduling heuristics.

Consider a set of tasks with a precedence graph, which are released at time 0. A deadline is specified for each task. It is assumed that the deadlines are chosen so that even if a task completes at its deadline, there will be enough time to execute its children in the task graph by their deadlines. If all the tasks that form a task graph are assigned to the same processor, then we can use the algorithm in Figure 3.19.

Example 3.21. Consider the task graph shown in Figure 3.20a, where the task execution times and deadlines are as follows:

Task T_i	e_i	d_i	Task T_i	e_i	D_i
1	3	6	2	3	7
3	2	20	4	5	21
5	6	27	6	6	28

Tasks are numbered so that $D_1 \leq D_2 \leq \ldots \leq D_n$.

1. Schedule task T_n in the interval $[D_n - e_n, D_n]$.
2. while `all the tasks have not been scheduled` do
   Let A be the set of as-yet-unscheduled tasks all of whose successors, if any, have been scheduled.
   Schedule task T_k, $k = \max\{m | m \in A\}$ as late as possible.
   end do
3. Move the tasks forward to the extent possible, keeping their order of execution as specified in step 2.

FIGURE 3.19
Algorithm PREC1.

The schedule as generated upon the completion of step 2 is shown in Figure 3.20b, and after moving the tasks forward in step 3 is shown in Figure 3.20c.

An interesting variation on the standard problem is scheduling with AND/OR constraints. In the standard problem, all the precedents of a task must be completed before that task can begin. In the AND/OR system, there are two types of tasks, AND tasks and OR tasks. AND tasks cannot commence computing before all their precedents have completed. OR tasks can commence after any one of their precedents has completed.

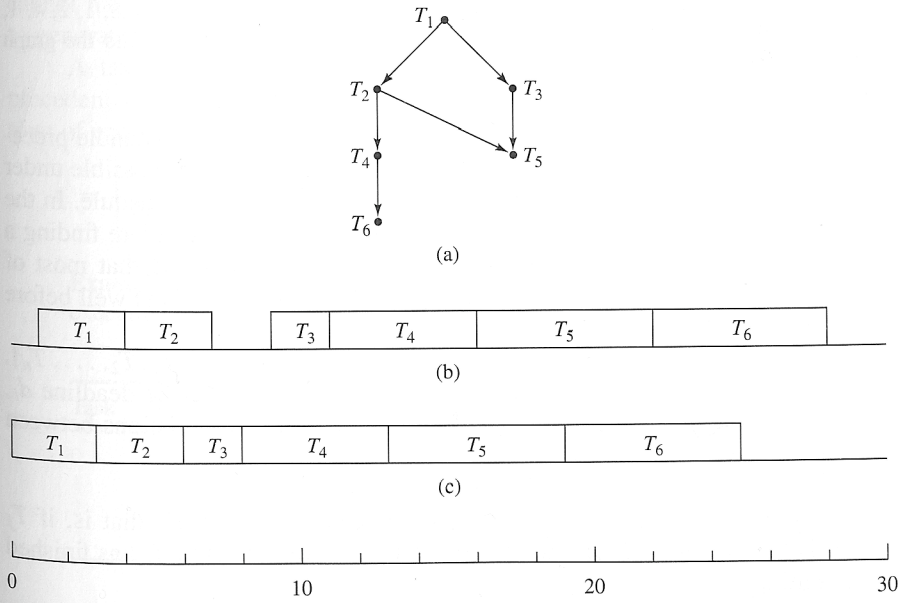


FIGURE 3.20
Example of algorithm PREC1: (a) task graph; (b) schedule after step 2 applies; (c) schedule after step 3 applies.

while A = set of all OR tasks is nonempty do:
    Choose task $T_i \in A$ none of whose precedents is an OR task.
    Find k such that $L(k) \leq L(j)$ for all $j \in \mathbf{P}_i$.
    In G, remove all edges terminating in T_i, except for the one from T_k.
    Relabel T_i as an AND task.
end do

FIGURE 3.21
Algorithm MINPATH.

Before presenting the scheduling heuristic for this problem, we first introduce some notation. Let $\mathbf{P}_i$ denote the set of all the immediate predecessors of task T_i according to the precedence graph, G. That is, if $T_k \in \mathbf{P}_i$, there is an edge from node T_k to node T_i in G. Define

$$L(i) = \begin{cases} e_i & \text{if } T_i \text{ has no precedents} \\ e_i + \max\{L(k) | T_k \in \mathbf{P}_i\} & \text{otherwise} \end{cases} \tag{3.57}$$

The minimum path algorithm, MINPATH, shown in Figure 3.21, reduces the AND/OR problem to the standard problem (consisting only of AND tasks) by suitably pruning the precedence graph. Scheduling can then be completed by using, for example, PREC1 or some other algorithm.

Example 3.22. Consider a set of eight tasks with execution times 5, 6, 8, 1, 2, 4, 1, 2, respectively. Tasks T_6, T_7, T_8 are OR tasks. The precedence graph and the graph as pruned by MINPATH are shown in Figure 3.22.

Let us now consider a more powerful and complex heuristic to handle precedence conditions. This algorithm enumerates the schedules that are possible under preemption or precedence limitations until we arrive at a feasible schedule. In the worst case, we might have to enumerate every possible schedule before finding a feasible schedule. However, simulation experiments have indicated that most of the time this algorithm finds a feasible schedule (assuming one exists) well before it has enumerated all the possible schedules.

Our task model is as follows. We have a set of tasks, $T = \{T_1, T_2, \ldots, T_n\}$. For each task T_i we are given the worst-case execution time e_i, the deadline d_i, and the release time r_i. In addition, we are given the following relations between every pair of tasks:

- T_i PRECEDES T_j is TRUE if T_i is in the precedence set of T_j, that is, if T_j needs the output of T_i and we cannot start executing T_j until T_i has finished executing.
- T_i EXCLUDES T_j is TRUE if T_j is not allowed to preempt T_i.
- T_i PREEMPTS T_j is TRUE if, when T_i is ready to run and T_j is currently running, T_j is always preempted by T_i.

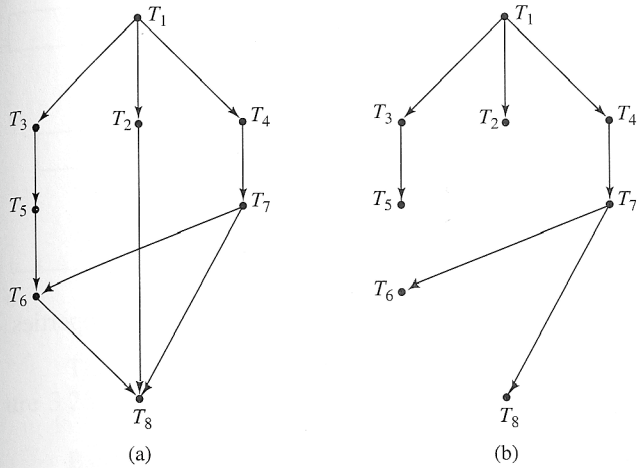


FIGURE 3.22
Task graph transformation by MINPATH: (a) original graph; (b) pruned graph. The node numbers are the task numbers.

Initially, we start with a set of PRECEDES and EXCLUDES relations as given by the set of tasks to be scheduled. The PREEMPT relation is initially empty. Clearly, some relations between a given pair of tasks are inconsistent with some other relations. For example, we cannot have both (T_i PRECEDES T_j) and (T_j PRECEDES T_i). We cannot have both (T_i PRECEDES T_j) and (T_j PREEMPTS T_i). The reader is invited to generate a few more examples of inconsistent relations.

A task is said to be *eligible* to run if it has been released and if all its precedent tasks have completed execution. We also define the *modified release time* of each task as follows.

$$r_i' = \begin{cases} r_i & \text{if no task PRECEDES } T_i \\ \max\{r_i, r_j' + e_j | T_j \text{PRECEDES} T_i\} & \text{otherwise} \end{cases} \tag{3.58}$$

Example 3.23. Figure 3.23 shows the task graph for a four-task set, where the execution and modified release times are:

Task	e_i	r_i	r_i'
0	5	0	0
1	4	1	5
2	9	10	10
3	3	11	19

This algorithm proceeds by first generating a valid initial schedule. If this solution meets all deadlines, we are done. If not, then we try to modify the schedule in order to minimize the extent to which deadlines are missed.

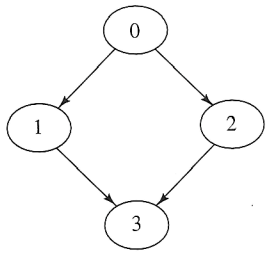


FIGURE 3.23
Example of modified release times.

A task T_i is said to be eligible to run at time t if the following properties are satisfied:

- all tasks T_j such that T_j PRECEDES T_i have completed by time t,
- T_i has not yet been completed by time t, and
- there is no as-yet-unfinished task T_k that was started before t, and such that T_k EXCLUDES T_i.

A schedule is said to be *valid* if it satisfies the following properties:

V1. The processor is not idle if there are one or more tasks that are ready to run.
V2. Exclusion, precedence, and preemption constraints are all satisfied throughout the schedule.

Within the context of these constraints, the EDF algorithm is used. If two tasks are both eligible to run (under the constraints) and have identical deadlines, the tie is broken on the basis of which one has the greater execution time. That is, an eligible task T_j will not run if there is a task T_i that is as yet unfinished but eligible to run, such that:

- T_i PREEMPTS T_j,
- $[d_i < d_j]$ and $\neg$ [T_j PREEMPTS T_i],[3] and
- $[d_i = d_j]$ and $\neg$ [T_j PREEMPTS T_i] and $e_i > e_j$.

Example 3.24. It is important to realize that validity property **V1** may not be optimal when there are tasks that cannot be preempted. To see this, consider the task set $T = \{T_1, T_2\}$, such that T_2 EXCLUDES T_1 and T_1 EXCLUDES T_2. Suppose $D_1 = 10, D_2 = 20, r_1 = 1, r_2 = 0, e_1 = 5$, and $e_2 = 10$. Assume that there are no precedence constraints. Then, when T_2 arrives at time 0, it starts executing. T_1 arrives at time 1, but cannot preempt T_2; it has to wait until T_2 finishes at time 5. By then, it is too late: T_1 simply does not have enough time to finish executing before its deadline. By contrast, if the processor is kept idle over the interval [0, 1], it can execute first T_1 and then T_2, and meet the deadlines of both tasks. See Figure 3.24.

[3] $\neg$ stands for logical NOT.

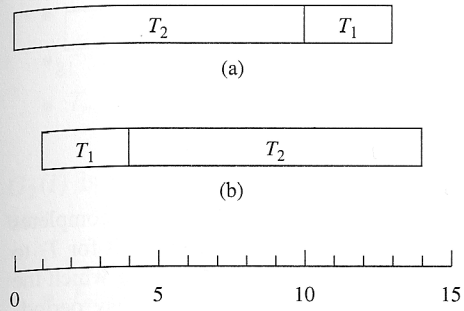


FIGURE 3.24
Example of a scheduling anomaly: (a) an infeasible schedule; (b) a reordered, and feasible, schedule.

The algorithm for generating a valid initial schedule is outlined in Figure 3.25. $f(i)$ is defined as the finishing time of task T_i in the schedule.

Example 3.25. Consider a four-task system with the following parameters:

	T_1	T_2	T_3	T_4
r_i	1	0	14	13
e_i	1	10	2	3
D_i	5	30	18	25

Suppose no precedence conditions exist, and that the only EXCLUDE relation is T_2 EXCLUDES T_1. Then, the valid initial schedule will be generated as follows. Task T_2 is released at time 0, and is scheduled to start running at that point. The next point to be examined is time 1 T_1 arrives and is prevented from preempting T_2 due to the EXCLUDES relation. The third point to be examined is time 10, when T_2 finishes. At this time, task T_1 is started and runs until 11. The processor is then idle until 13, when it starts executing T_4. T_3 arrives at 14 and, because it has an earlier deadline, it

```
t = 0
while (there are still unfinished tasks) do
    if (∃i : t = r_i' ∨ t = f(i)) then
        select for execution the highest-priority eligible task with the minimum
            deadline.
        If more than one eligible task has the same minimum deadline,
        break ties according to their execution times, giving priority
        to the one with the greatest execution time.
    end if
    t = t + 1
end while
```

FIGURE 3.25
Algorithm for generating a valid initial schedule.

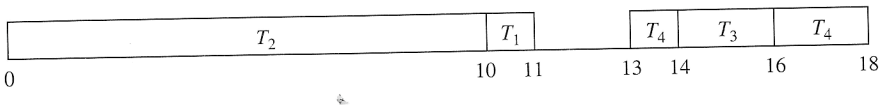


FIGURE 3.26
Valid initial schedule example.

preempts T_4 and executes to completion at time 16. T_4 then resumes and completes at 18. As currently scheduled, T_1 misses its deadline because it must wait for T_2 to finish. See Figure 3.26. A processor busy period is a time interval during which the processor is continuously busy. In the above example, the processor busy periods for the valid initial schedule are [0, 11] and [13, 18].

Our next step is to check if any tasks have missed their deadlines. If none of them has done so, we are done. If some deadlines have been missed, then we try to rectify this situation by reordering the tasks in the schedule. In this example, we can obviously make the processor idle until time 1, and then start executing T_1. T_2 can start executing after T_1 has finished.

Note that it is only useful to reorder the tasks in the same busy period as the one that missed its deadline. It is no use tinkering with the order of T_3, T_4 to try to affect T_1, T_2; these tasks arrive after both T_1 and T_2 have finished executing.

Denote by $Z(i)$ the set containing the tasks that are in the same busy period as T_i, the tasks that are scheduled before T_i, and T_i itself. We can obtain $Z(i)$ recursively as follows ($s(i)$ is the start time of T_i in the schedule):

- $T_i \in Z(i)$
- $T_k \in Z(i)$ if $\exists T_\ell \in Z(i)$ such that

$$\left([f(k) = s(\ell)] \wedge [\exists \ell' \in Z(i) : r'_{\ell'} < f(k)]\right) \vee [s(\ell) < f(k) < f(i)]$$

Example 3.26. In the valid initial schedule in Figure 3.26, $Z(1) = \{1, 2\}$, $Z(2) = \{2\}$, $Z(3) = \{3, 4\}$, and $Z(4) = \{4\}$.

Note that $f(i)$ is also the earliest possible time by which all the tasks in $Z(i)$ will finish execution.

Define the *lateness* of task T_i as $L(i) = f(i) - D_i$. If a task has a positive lateness in a schedule, it has missed its deadline according to that schedule. The lateness of a schedule is the maximum task lateness.

We now introduce two sets, $G_1(i)$ and $G_2(i)$. $G_1(i)$ is a set of tasks that cannot be preempted by T_i (because of EXCLUDES relations), but that, if moved in the schedule to execute after T_i, may reduce the maximum lateness of the system. $G_2(i)$ is a set of tasks that, if preempted by T_i, may reduce the maximum lateness. We obtain $G_1(i)$ by listing all tasks T_m satisfying all of the following properties:

- $T_m \in Z(i)$,
- $D_i < D_m$,

- $\neg(T_m$ PREEMPTS $T_i)$,
- $\neg(T_m$ PRECEDES $T_i)$, and
- T_m EXCLUDES T_i.

$G_2(i)$ is obtained by listing all tasks T_m satisfying all of the following properties:

- $T_m \in Z(i)$,
- $D_i < D_m$,
- $\neg(T_m$ EXCLUDES $T_i)$,
- $\neg(T_m$ PRECEDES $T_i)$,
- $\neg(T_m$ PREEMPTS $T_i)$, and
- There is no third task T_ℓ scheduled to run between T_i T_m, such that (T_m PRECEDES $T_\ell) \vee (T_m$ PREEMPTS $T_\ell)$.

Let us now compute a lower bound on the lateness of a valid initial schedule. Define set $K(i)$ as follows. $T_k \in K(i)$ iff each of the following is true:

- $T_k \in Z(i)$,
- $k \neq i$,
- $D_i < D_k$,
- $\neg(T_k$ PRECEDES $T_i)$,
- $\neg(T_k$ PREEMPTS $T_i)$.

If T_i is a task with the maximum lateness of any task in the system and $K(i) = \emptyset$, then we cannot improve the lateness of T_i by moving other tasks—doing so will cause lateness in one or more of the moved tasks that is at least equal to the current lateness of T_i. If $K(i) \neq \emptyset$, then we can improve on the maximum lateness by moving a task in $K(i)$. We will see examples of this shortly.

A lower bound on the lateness can be determined as follows. Suppose we have a busy period of the processor occupying the interval $[a, b]$. Note that we can never adjust the tasks to move the right endpoint of this interval back; all that we can do is to either leave it as it is (which is what will happen if we simply reorder the tasks within the busy period, without leaving any gaps), or move it further to the right (which will happen if we move a task in such a way that gaps are formed). For instance, consider the busy period [0,11] that was formed in Example 3.25. If we make the processor idle until the arrival of T_1, we are creating a gap of 1 unit in the left end of the interval [0,11] and moving the right endpoint of the busy period to 12.

Define the function $GAP(k, i)$ as follows.

$$GAP(k,i) = \begin{cases} 0 & \text{if } \neg(T_k \text{ EXCLUDES } T_i) \\ \max\{0, -s(k) + \min\{r'_\ell \mid [\ell \in Z(i)] \\ \quad \wedge [k \neq \ell] \\ \quad \wedge [s(k) < s(\ell) \leq s(i)] \\ \quad \wedge [\neg(T_k \text{PRECEDES } T_\ell)]\}\} & \text{otherwise} \end{cases} \tag{3.59}$$

$GAP(k, i)$ is the gap that would be left in the busy period if we moved T_k out to the right of T_i.

Define the function $LB(i)$ as follows.

$$LB(i) = \begin{cases} f(i) - D_i & \text{if } K(i) = \emptyset \\ f(i) + \min_{k \in K(i)}\{GAP(k,i) - D_k\} & \text{otherwise} \end{cases} \tag{3.60}$$

If $K(i) = \emptyset$, then there are no tasks that can be moved to reduce the maximum lateness of the schedule. As a result, the lateness of task T_i remains $f(i) - D_i$. If $K(i) \neq \emptyset$, then the right endpoint of $Z(i)$ moves right by $GAP(k, i)$, (i.e., it is now $f(i) + GAP(k, i)$). The lower bound of the lateness with respect to the busy period up to and including T_i is thus given by $f(i) + \min_{k \in K(i)} \{GAP(k, i) - D_k\}$.

Now, define:

$$LB_1(i) = \min\{LB(i), f(i) - D_i\} \tag{3.61}$$

$$LB_2(i) = r'_i + e_i - D_i \tag{3.62}$$

$LB_1(i)$ provides a lower bound on the lateness of the schedule under the constraints of the PREEMPT, EXCLUDE, and PRECEDE relations. $LB_2(i)$ is a lower bound defined by the task parameters and cannot be reduced. $r'_i + e_i$ is the earliest that we can execute task T_i, by definition.

It follows that a lower bound of the lateness of the schedule is given by

$$\mathcal{L} = \max\{LB_1(i), LB_2(i)\} \tag{3.63}$$

The schedule is obtained by first running the algorithm in Figure 3.25 to obtain a valid initial schedule. If this schedule either is feasible or achieves the lower bound on the lateness in the root node schedule, we are done. If not, some modifications need to be made to this schedule (i.e., some tasks have to be moved around in it), in order to reduce the lateness. Treat the valid initial schedule as the root node.

We identify a task T_j with the maximum lateness and strive to reduce it. Recall that $G_1(j)$ consists of all the tasks in the schedule that, if scheduled to run after T_j, can reduce the lateness of T_j. There are $\|G_1(j)\|$ such tasks.[4] Associate

[4] $\|A\|$ means the number of elements in set A.

a child node with each such task; there will thus be $\|G_1(j)\|$ such nodes. In the child node that corresponds to T_k being run after T_j to reduce the lateness, we can force this to happen by adding the relation T_j PRECEDES T_k.

Recall also that $G_2(j)$ consists of tasks that, if preempted by T_j, may reduce the maximum lateness; so we want to make T_j preempt such tasks where possible. We generate $\|G_2(j)\|$ additional child nodes to the root node, with one child node corresponding to each element in $G_2(j)$. Consider $T_k \in G_2(j)$. If we have some other task T_ℓ sandwiched between T_k and T_j in the schedule and if T_k is prohibited from preempting T_ℓ, we want to interchange T_ℓ and T_k by adding the relation T_ℓ PRECEDES T_k. If we have tasks T_q such that $\neg(T_k$ EXCLUDES and $T_q)$, and T_q executes between T_k and T_j, add the relation T_q PREEMPTS T_k and T_j PREEMPTS T_k.

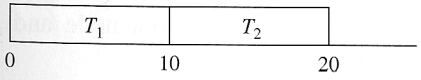
We proceed by developing the node (i.e., the schedule) that has the minimum lateness. The scheduling algorithm is shown in Figure 3.27, and Examples 3.27 and 3.28 illustrate how it works.

Example 3.27. Consider a two-task system whose parameters are:

	T_1	T_2
r_i	0	5
D_i	30	15
e_i	10	10

Note: T_1 EXCLUDES T_2.

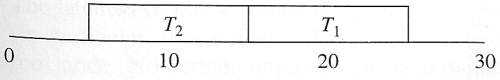
The valid initial schedule is:



Task T_2 misses its deadline and its lateness is $20-15 = 5$. $K(2) = \{1\}$. $GAP(1, 2) = 10$. The lower bound of the lateness is therefore

$$LB(2) = f(2) + \min_{k \in K(2)} \{GAP(k, 2) - D_k\} = 0$$

We develop this node further. $G_1(2) = \{1\}$, so we add the relation T_2 PRECEDES T_1. When the scheduling algorithm is run under this condition, we have the schedule shown below, which meets all the deadlines.



The algorithm puts out this schedule and stops.

1. Run the algorithm in Figure 3.25 to obtain a valid initial schedule. Compute the lower bound of the lateness of this schedule. If a feasible schedule results or the lateness equals the lower bound, output the schedule and stop. Otherwise, let the task with the maximum lateness be T_j. Define ml as the lateness of this schedule. Go to the next step.
2. Treat the valid initial schedule obtained above as the root node of a graph generated as follows. Find sets $G_1(j)$ and $G_2(j)$ with respect to the root node and create $\|G_1(j)\| + \|G_2(j)\|$ child nodes. For each node that corresponds to some task $T_k \in G_1(j)$, introduce a new relation T_j PRECEDES T_k.
   For each node corresponding to some task $T_k \in G_2(j)$, do the following.
   a. For all tasks T_ℓ with the properties that T_k EXCLUDES T_ℓ and T_ℓ executes between the execution of T_k and T_j, introduce a new relation T_ℓ PRECEDES T_k.
   b. For all tasks T_q with the properties that the relation T_k EXCLUDES T_q does NOT hold and T_q executes between T_j and T_k, introduce the new relations T_q PREEMPTS T_k and T_j PREEMPTS T_k.

   A child node also inherits all relations of its parent node.
   Recompute a valid initial schedule for each of the child nodes.
3. If steps 4 and 5 have been completed for all the child nodes, close the parent node and go to step 5. Otherwise, pick the child node T_n that has minimum lateness under the valid initial schedule and go to step 4.
4. Set $ml \leftarrow \min\{ml, \text{lateness(child node } n)\}$. If ml is no greater than the least lower bound of the lateness of all the open nodes, we have achieved the best schedule possible—output the schedule and stop. ~~Otherwise, this node can never be developed into a solution better than the currently achieved ml; close this node and go to step 3~~. See errata
5. Pick from among all the open nodes the one with the least lower bound for the lateness. If more than one open node has the least lower bound, pick the one with the smallest lateness. Define this node as the root node and go to step 2.

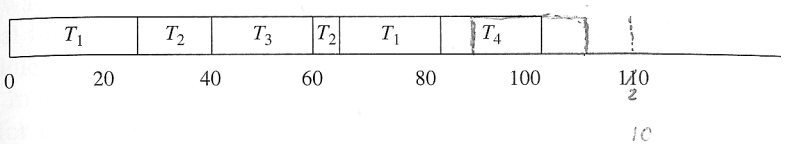
FIGURE 3.27
Scheduling algorithm.

Example 3.28. Consider now a four-task system whose parameters are:

	T_1	T_2	T_3	T_4
r_i	0	25	40	80
e_i	50	20	20	20
D_i	148	145	125	100

Note: T_1 EXCLUDES T_4.

The valid initial schedule at the root node is:



T_4 misses its deadline and the lateness of the schedule is 5. Let us calculate the lower bound of the lateness under the constraints specified here. We have the following:

$$K(4) = \{1, 2, 3\} \tag{3.64}$$

$$GAP(1,4) = -s(1) + \min\{r'(2), r'(3)\}$$
$$= 25 \tag{3.65}$$

$$GAP(2,4) = 0 \quad \text{(because } \neg(T_2 \text{ EXCLUDES } T_4)) \tag{3.66}$$

$$GAP(3,4) = 0 \quad \text{(because } \neg(T_3 \text{ EXCLUDES } T_4)) \tag{3.67}$$

$$LB(4) = f(4) + \min_{k \in K(4)} \{GAP(k,i) - D_k\}$$
$$= 105 + \min\{25 - 148, 0 - 145, 0 - 125\}$$
$$= 105 - 145 = -40 \tag{3.68}$$

$$LB_1(4) = \min\{-40, 5\} = -40 \tag{3.69}$$

$$LB_2(4) = 80 + 20 - 100 = 0 \tag{3.70}$$

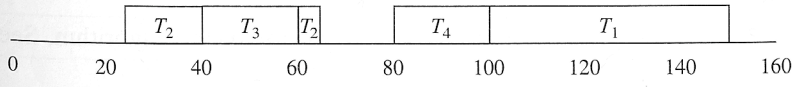
$$\mathcal{L}(4) = \max\{LB_1(4), LB_2(4)\} = 0 \tag{3.71}$$

Since $\mathcal{L}(4)$ is less than the lateness of the schedule, there could be room for improvement. We write out $G_1(4)$ and $G_2(4)$:

$$G_1(4) = \{1\} \tag{3.72}$$

$$G_2(4) = \{2, 3\} \tag{3.73}$$

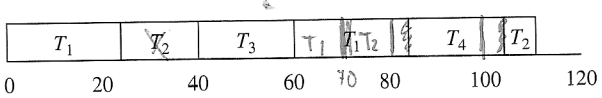
We now create three child nodes of the root node. The first of these represents an additional constraint we shall place with respect to T_1 (connected with $G_1(4) = \{1\}$). This additional constraint is T_4 PRECEDES T_1. The schedule that results is:



The lateness of this schedule is 2, which is equal to the lower bound of the lateness (in particular, it is equal to $LB_2(1)$). This lower bound is worse than that of the root node. No further improvement is possible along this path and we close this node.

Let us now turn to the second child node of the root. This corresponds to T_2 (since $2 \in G_2(4)$). We add the relations: T_1 PREEMPTS T_2, T_3 PREEMPTS T_2, and

T_4 PREEMPTS T_2. With these added relations, we run the scheduling algorithm in Figure 3.25 to obtain the following schedule:



The lateness of this schedule is 5 (T_4 again misses its deadline by 5 units; all the other tasks meet their deadlines). This node is not closed because, as before, we can make the following calculations:

$$K(4) = \{1, 2, 3\} \tag{3.74}$$

$$GAP(1,4) = -s(1) + \min\{r'(2), r'(3)\}$$
$$= 25 \tag{3.75}$$

$$GAP(2,4) = 0 \quad \text{(because } \neg(T_2 \text{ EXCLUDES } T_4)) \tag{3.76}$$

$$GAP(3,4) = 0 \quad \text{(because } \neg(T_3 \text{ EXCLUDES } T_4)) \tag{3.77}$$

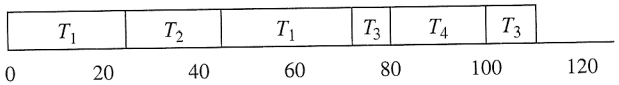
$$LB(4) = f(4) + \min_{k \in K(4)} \{GAP(k,i) - D_k\}$$
$$= 105 + \min\{25 - 148, 0 - 145, 0 - 125\}$$
$$= 105 - 145 = -40 \tag{3.78}$$

$$LB_1(4) = \min\{-40, 5\} = -40 \tag{3.79}$$

$$LB_2(4) = 80 + 20 - 100 = 0 \tag{3.80}$$

$$\mathcal{L}(4) = \max\{LB_1(4), LB_2(4)\} = 0 \tag{3.81}$$

Let us, however, turn to the third child node. This corresponds to using $3 \in G_2(4)$. We add the relations T_1 PREEMPTS T_3, T_2 PREEMPTS T_3, and T_4 PREEMPTS T_3. Under these additional relations, the algorithm in Figure 3.25 returns the following schedule:



This is feasible, and so the algorithm puts out this schedule and stops.

There is an interesting multiprocessor extension of this algorithm. See Section 3.7 for a pointer to the literature.

3.2.4 Using Primary and Alternative Tasks

Throughout this chapter, we have assumed that there must always be sufficient time for the critical tasks to execute. In order to ensure that critical tasks will complete before their deadline, we carry out a scheduling that assumes that each critical task will run to its worst-case time. Quite often, the worst-case execution time of such tasks is much greater than the average-case execution time. This results in much more time being scheduled for the tasks than is really needed. One way of retaining a high utilization of the hardware is to reclaim for less critical functions the time left unused when the critical tasks do not need all their scheduled time.

In this section, we shall consider a second approach to the problem. Suppose that for each critical task, we have two versions, a primary and an alternative. Completing either the primary or alternative version successfully results in the critical task being executed. However, the alternative is a "bare-bones" version that provides service that is just acceptable, while the primary may be capable of providing better-quality service. The alternative version has a much smaller worst-case execution time than the primary. Since only one of these versions has to execute in time to ensure acceptable service, we can avoid having to preallocate the primary for its worst-case time.[5]

Example 3.29. To illustrate, let us consider the very simple example of a one-task set, consisting of a primary and an alternative. The relative deadline is equal to the task period. The parameters are shown in the following table:

	Primary	Alternative
Worst-case run time	20	5
Average run time	7	4
Period	15	15

If only the primary version were available, this task set would be impossible to schedule; there simply wouldn't be time to complete executing the primary if it ran to its worst-case time. However, since we now have an alterative, we can set up the schedule shown in Figure 3.28. We allow 10 time units for the primary version to run in each period of 15 units; we call this the *run-time limit* of the primary version. Much of the time (since the average run time is only 7) the primary version will have completed by that time, and we can reclaim the time beyond the completion time for other activities. However, if the primary runs for more than 10, we abort it and start up the alternative task. While it does not provide results that are as good as

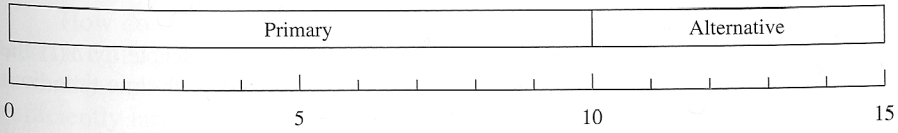


FIGURE 3.28
Example of using primary and alternative versions.

[5] In Section 3.3, this concept will be taken one step further.

the primary does, the alternative at least is guaranteed to generate acceptable output within a worst-case execution time of 5. Thus, we are assured of at least one of the two versions executing by the deadline (given that there are no failures, of course).

Assume that the set of tasks is periodic, and that the periods are in the set $\{P_m, 2P_m, 2^2P_m, \ldots, 2^iP_m\}$. Clearly, P_m is the smallest period of any task in the set. A task is said to be of *level-i* if its period is 2^iP_m, $i = 0, 1, 2, \ldots$. Assume that r is the highest level (i.e., there is no task whose period exceeds 2^rP_m). With each primary version π_i of task T_i, associate a run-time limit ℓ_i. If the primary runs beyond this run-time limit, we will abort it and turn to the corresponding alternative version.

We now present two uniprocessor scheduling algorithms, one for generating the initial schedule and another for reclaiming unused time from the initial schedule.

The initial schedule is generated as follows. First, we schedule all level-0 tasks over an interval P_m, ensuring that all alternative versions of such tasks are scheduled, and then schedule the maximum number of primary versions that will fit in the remaining time. The alternative version of a task is never scheduled to run before its primary. Call this schedule S_0.

Next, concatenate two S_0 schedules to form one schedule of length $2P_m$. Schedule all level-1 tasks in the following manner. First, schedule the alternative versions. If there is insufficient space in the schedule to fit all the alternatives, drop some of the primary versions of the level-0 tasks, as necessary. If primary versions have to be dropped, drop the ones that have the longest run-time limits (the idea is to drop as few of them as possible). Once all the level-1 alternatives have been scheduled, see if any of the level-1 primaries can be scheduled in the space available. If they fit, do so. Primaries are checked for inclusion in ascending order of their run-time limits. Then, check to see if any as-yet-unscheduled level-1 primaries have a lower run-time than any primary already scheduled. If so, drop the already-scheduled primary with the longest run-time limit and replace it with the level-1 primary. When this has been done, concatenate two copies of the resultant schedule together to form a schedule of length 2^2P_m. Schedule level-2 tasks in the same way—drop level-1 or level-0 primaries as necessary to schedule level-2 alternatives, dropping the ones with the longest run-time limit first. Continue in this way until all tasks have at least their alternative versions scheduled to run in each task period.

Example 3.30. Consider the following task set of five tasks. Denote by $\alpha(i)$ the worst-case run time of the alternative version of T_i, and by $\ell(i)$ the run-time limit of the corresponding primary version.

	T_1	T_2	T_3	T_4	T_5
$\ell(i)$	10	10	15	10	5
$\alpha(i)$	3	2	1	7	4
$P(i)$	20	20	20	40	40

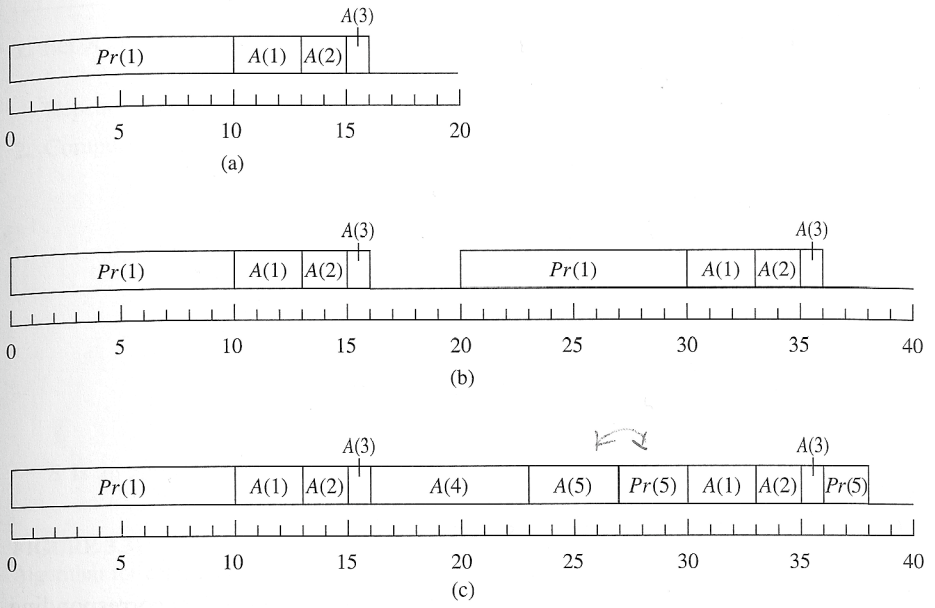


FIGURE 3.29
Example of primary and alternative-version scheduling: (a) schedule S_0; (b) two copies of schedule S_0 concatenated; (c) incorporating level-1 tasks. Pr(i) = primary of T_i; A(i) = alternative of T_i.

The level-0 tasks are T_1, T_2, T_3, and the level-1 tasks are T_4, T_5. Generate schedule S_0. After scheduling the three alternative versions, we have only 15 time units left. We pick a primary with the least run-time limit (of task T_1) and schedule it. The alternative versions are scheduled to run after this primary. See Figure 3.29a.

Next, we concatenate two copies of S_0 (see Figure 3.29b). Our first order of business is to ensure that the alternative versions of T_4 and T_5 are scheduled. This requires a total of 11 time units. We do not have 11 units free in this schedule, so we drop one of the iterations of the T_1 primary version (say the second one), and add the alternatives of T_4 and T_5. We also have enough space to add $Pr(5)$—it is executed in two parts over the intervals [27, 30] and [36, 38]. See Figure 3.29c. Here the algorithm ends.

How do we choose the run-time limit of the primary? The simplest option is to set it equal to the worst-case execution time. However, this might result in only a small number of primaries being scheduled. Another option is to pick a time sufficiently large so that with some large probability, the primary can be expected to complete within the run-time limit.

Example 3.31. Suppose the probability density function of the execution time of a primary is as shown in Figure 3.30. The density function has a long "tail." However, most of the probability is concentrated in the interval $[0, t_1]$. Hence, we might choose to set the run-time limit at t_1 rather than at the worst-case execution time.

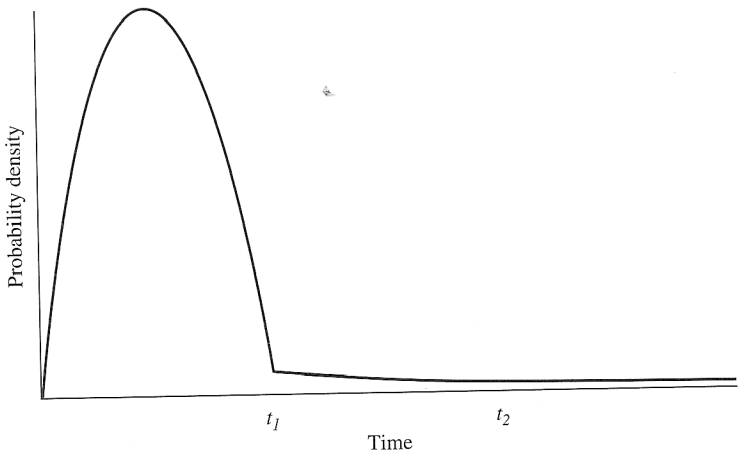


FIGURE 3.30
Probability density function of the execution time of a primary.

If a primary version completes successfully, we do not need its corresponding alternative for that period. This time can then be reclaimed. Such reclamation can result in time becoming available for other primaries, which were not part of the original schedule, to be executed. The algorithm for doing this is a simple modification of the above and is left to the reader as an exercise.

3.3 UNIPROCESSOR SCHEDULING OF IRIS TASKS

Thus far in this chapter, we have assumed that to obtain an acceptable output a task has to be run to completion. Put another way, if the task is not run to completion, we get zero reward from it (i.e., it may as well not have been run). However, there is a large number of tasks for which this is not true. These are iterative algorithms. The longer they run, the higher the quality of their output (up to some maximum run time).

Example 3.32. Figure 3.31 contains an algorithm for computing the value of π. The more times step 2 is executed, the more accurate P is as an approximation of π (subject, of course, to limitations due to finite numerical precision).

The difference between the calculated value and the actual value of π (the "error") as a function of the iteration number is provided in Table 3.1. The error is greatest for the first iteration; it diminishes rapidly after that.

Search algorithms for finding the minimum of some complicated function are also examples of iterative tasks. The longer we search the parameter space, the greater is the chance that we will obtain the optimum value or something close to it.

1. Set $A = \sqrt{2}$, $B = \sqrt[4]{2}$, $P = 2 + \sqrt{2}$.

   Repeat step 2 as many times as necessary.

2. Compute

$$A := \frac{\sqrt{A} + 1/\sqrt{A}}{2}$$

$$P := P\left(\frac{A+1}{B+1}\right)$$

$$B := \frac{B\sqrt{A} + 1/\sqrt{A}}{B+1}$$

   B is an approximation of π.

FIGURE 3.31
Algorithm for calculating π.

TABLE 3.1
Errors in calculating π

Iteration	Error
1	0.00101410035182936232807644033 9
2	0.00000000737625099203510885184 1
3	0.00000000000000000000183130608478

Example 3.33. Chess-playing algorithms evaluate the goodness of moves by looking ahead several moves. The more time they have, the further they can look and the more accurate will be the evaluation.

Tasks of this type are known as *increased reward with increased service* (IRIS) tasks. The reward function associated with an IRIS task increases with the amount of service given to it. Typically, the reward function is of the form

$$R(x) = \begin{cases} 0 & \text{if } x < m \\ r(x) & \text{if } m \le x \le o + m \\ r(o+m) & \text{if } x > o + m \end{cases} \tag{3.82}$$

where $r(x)$ is monotonically nondecreasing in x. The reward is 0 up to some time m; if the task is not executed up to that point, it produces no useful output. Tasks with this reward function can be regarded as having a mandatory and an optional component. The *mandatory* portion (with execution time m) must be completed by the deadline if the task is critical; the *optional* portion is done if time permits. The optional portion requires a total of o time to complete. In each case, the execution of a task must be stopped by its deadline d.

The scheduling task can be described as the following optimization problem:

Schedule the tasks so that the reward is maximized, subject to the requirement that the mandatory portions of all the tasks are completed.

It can be shown that this optimization problem is NP-complete when there is no restriction on the release times, deadlines, and reward functions. However, for some special cases, we do have scheduling algorithms. We now turn to these. In what follows, m_i and o_i denote the execution time of the mandatory and optional parts, respectively, of T_i.

3.3.1 Identical Linear Reward Functions

For task T_i, the reward function is given by

$$R_i(x) = \begin{cases} 0 & \text{if } x \le m_i \\ x - m_i & \text{if } m_i \le x \le o_i + m_i \\ o_i & \text{if } x > m_i + o_i \end{cases} \tag{3.83}$$

That is, the reward from executing a unit of optional work is one unit. A schedule is said to be optimal if the reward is maximized subject to all tasks completing at least their mandatory portions by the task deadline.

Theorem 3.13. The EDF algorithm is optimal if the mandatory parts of all tasks are 0.

Proof. If the mandatory portions are zero, then we can execute as little of any task as we please. It is easy to see that the reward is maximized if the processor is kept busy for as much time as possible. But this is exactly what the EDF algorithm does; if the processor is idle at some time t, this is because (a) all the previously released tasks have either completed or their deadlines have expired by time t, and (b) no other tasks have been released. **Q.E.D.**

We can use this result to develop an optimal scheduling algorithm for the case when the mandatory portions are not all zero. The tasks $T_1, \ldots, T_n$ have mandatory portions $M_1, \ldots, M_n$ and optional portions $O_1, \ldots, O_n$. Define

$$\mathbf{M} = \{M_1, \ldots, M_n\}$$
$$\mathbf{O} = \{O_1, \ldots, O_n\}$$
$$\mathbf{T} = \{T_1, \ldots, T_n\}$$

The optimal algorithm, IRIS1, is shown in Figure 3.32. Although it looks a little forbidding, the idea behind it is quite simple. First, since we receive one unit of reward for each unit of the optional portion completed for any task, the highest reward, subject to the constraint that all the mandatory portions are completed, is obtained when the processor carries out as much execution as possible.

1. Run the EDF algorithm on the task set **T** to generate a schedule S_t.
If this is feasible,
    An optimal schedule has been found: STOP.
Else,
    go to step 2.
end if

2. Run the EDF algorithm on the task set **M**, to generate a schedule S_m.
If this set is not feasible,
    T cannot be feasibly scheduled: STOP.
Else,
    Define a_i as the ith instant in S_m when either the scheduled task changes, or the processor becomes idle, $i = 1, 2, \ldots$.
        Let k be the total number of these instants.
    Define a_0 as when the first task begins executing in S_m.
    Define $\tau(j)$ as the task that executes in S_m in $[a_j, a_{j+1}]$,
    Define $L_t(j)$ and $L_m(j)$ as the total execution time given to task $\tau(j)$ in $S_t(j)$ and $S_m(j)$ respectively, after time a_j.
    Go to step 3.
end if

3. $j = k - 1$
do while $(0 \le j \le k - 1)$
    if $(L_m(j) > L_t(j))$ then
        Modify S_t by
            (a) assigning $L_m(j) - L_t(j)$ of processor time in $[a_j, a_{j+1}]$ to $\tau(j)$, and
            (b) reducing the processor time assigned to other tasks in $[a_j, a_{j+1}]$ by $L_m(j) - L_t(j)$.
        Update $L_t(1), \ldots, L_t(j)$ appropriately.
    end if
    $j = j - 1$
end do
end

FIGURE 3.32
Algorithm IRIS1.

We begin by running the EDF algorithm for the total run time of each task. Call the resulting schedule S_t. S_t maximizes the total processor busy time. If S_t is a feasible schedule, we are clearly done—we have given each task as much time as it needs to finish executing both its mandatory and optional portions, and have still met each task deadline. Suppose we do not obtain a feasible schedule; that is, some task cannot be given its full execution time and still meet its deadline.