C H A P T E R 4

Commonly Used Approaches to Real-Time Scheduling

This chapter provides a brief overview of three commonly used approaches to scheduling real-time systems: clock-driven, weighted round-robin and priority-driven. The subsequent five chapters will study in depth the clock-driven and priority-driven approaches, with particular emphasis on the latter. We need to be aware of several important facts about priority-driven scheduling. They are presented here and shown to be true, so they will not be buried in the details later. We will use these facts to motivate and support the arguments presented in later chapters. The weighted round-robin approach is used primarily for scheduling real-time traffic in high-speed switched networks. It will be described in depth in Chapter 11. We discuss here why this approach is not ideal for scheduling jobs on CPUs.

4.1 CLOCK-DRIVEN APPROACH

As the name implies, when scheduling is *clock-driven* (also called *time-driven*), decisions on what jobs execute at what times are made at specific time instants. These instants are chosen a priori before the system begins execution. Typically, in a system that uses clock-driven scheduling, all the parameters of hard real-time jobs are fixed and known. A schedule of the jobs is computed off-line and is stored for use at run time. The scheduler schedules the jobs according to this schedule at each scheduling decision time. In this way, scheduling overhead during run-time can be minimized.

A frequently adopted choice is to make scheduling decisions at regularly spaced time instants. One way to implement a scheduler that makes scheduling decisions periodically is to use a hardware timer. The timer is set to expire periodically without the intervention of the scheduler. When the system is initialized, the scheduler selects and schedules the job(s) that will execute until the next scheduling decision time and then blocks itself waiting for the expiration of the timer. When the timer expires, the scheduler awakes and repeats these actions.

4.2 WEIGHTED ROUND-ROBIN APPROACH

The round-robin approach is commonly used for scheduling time-shared applications. When jobs are scheduled on a round-robin basis, every job joins a First-in-first-out (FIFO) queue when it becomes ready for execution. The job at the head of the queue executes for at most one time slice. (A time slice is the basic granule of time that is allocated to jobs. In a time-shared environment, a time slice is typically in the order of tens of milliseconds.) If the job does not complete by the end of the time slice, it is preempted and placed at the end of the queue to wait for its next turn. When there are n ready jobs in the queue, each job gets one time slice every n time slices, that is, every *round*. Because the length of the time slice is relatively short, the execution of every job begins almost immediately after it becomes ready. In essence, each job gets $1/n$th share of the processor when there are n jobs ready for execution. This is why the round-robin algorithm is also called the processor-sharing algorithm.

The *weighted round-robin algorithm* has been used for scheduling real-time traffic in high-speed switched networks. It builds on the basic round-robin scheme. Rather than giving all the ready jobs equal shares of the processor, different jobs may be given different *weights*. Here, the weight of a job refers to the fraction of processor time allocated to the job. Specifically, a job with weight wt gets wt time slices every round, and the length of a round is equal to the sum of the weights of all the ready jobs. By adjusting the weights of jobs, we can speed up or retard the progress of each job toward its completion.

By giving each job a fraction of the processor, a round-robin scheduler delays the completion of every job. If it is used to schedule precedence constrained jobs, the response time of a chain of jobs can be unduly large. For this reason, the weighted round-robin approach is not suitable for scheduling such jobs. On the other hand, a successor job may be able to incrementally consume what is produced by a predecessor (e.g., as in the case of a UNIX pipe). In this case, weighted round-robin scheduling is a reasonable approach, since a job and its successors can execute concurrently in a pipelined fashion. As an example, we consider the two sets of jobs, $\mathbf{J}_1 = \{J_{1,1}, J_{1,2}\}$ and $\mathbf{J}_2 = \{J_{2,1}, J_{2,2}\}$, shown in Figure 4–1. The release times of all jobs are 0, and their execution times are 1. $J_{1,1}$ and $J_{2,1}$ execute on processor P_1, and $J_{1,2}$ and $J_{2,2}$ execute on processor P_2. Suppose that $J_{1,1}$ is the predecessor of $J_{1,2}$, and $J_{2,1}$ is the predecessor of $J_{2,2}$. Figure 4–1(a) shows that both sets of jobs (i.e., the second jobs $J_{1,2}$ and $J_{2,2}$ in the sets) complete approximately at time 4 if the jobs are scheduled in a weighted round-robin manner. (We get this completion time when the length of the time slice is small compared with 1 and the jobs have the same weight.) In contrast, the schedule in Figure 4–1(b) shows that if the jobs on each processor are executed one after the other, one of the chains can complete at time 2, while the other can complete at time 3. On the other hand, suppose that the result of the first job in each set is piped to the second job in the set. The latter can execute after each one or a few time slices of the former complete. Then it is better to schedule the jobs on the round-robin basis because both sets can complete a few time slices after time 2.

Indeed, the transmission of each message is carried out by switches en route in a pipeline fashion. A switch downstream can begin to transmit an earlier portion of the message as soon as it receives the portion without having to wait for the arrival of the later portion of the message. The weighted round-robin approach does not require a sorted priority queue, only a round-robin queue. This is a distinct advantage for scheduling message transmissions in ultrahigh-speed networks, since priority queues with the required speed are expensive. In

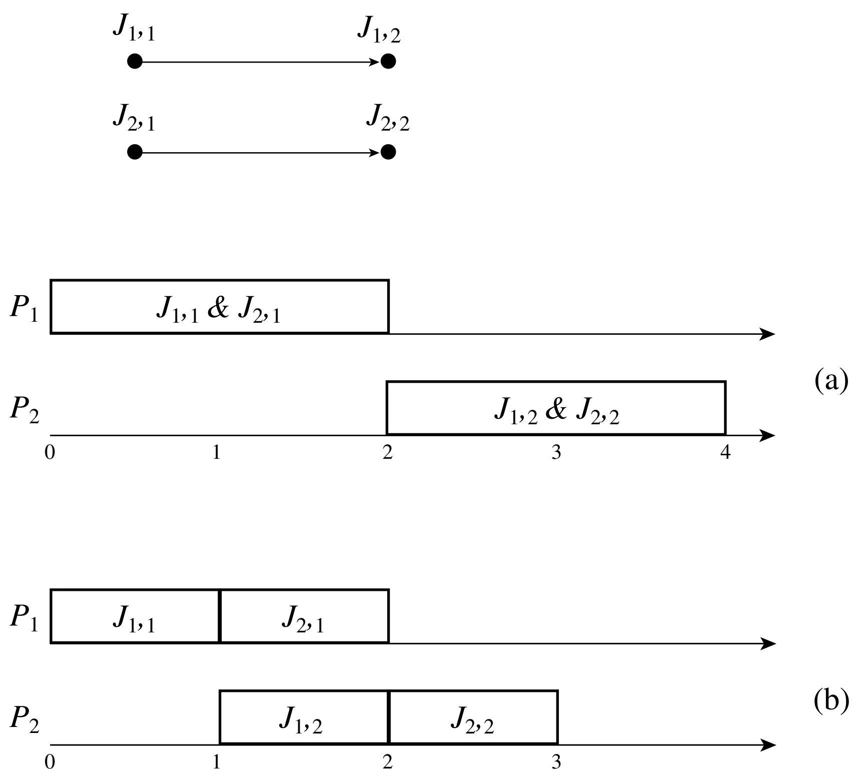


FIGURE 4–1 Example illustrating round-robin scheduling of precedence-constrained jobs.

Chapter 11 we will describe in detail the application of weighted round-robin algorithms to network scheduling.

4.3 PRIORITY-DRIVEN APPROACH

The term *priority-driven* algorithms refers to a large class of scheduling algorithms that never leave any resource idle intentionally. Stated in another way, a resource idles only when no job requiring the resource is ready for execution. Scheduling decisions are made when events such as releases and completions of jobs occur. Hence, priority-driven algorithms are *event-driven*.

Other commonly used names for this approach are *greedy scheduling*, *list scheduling* and *work-conserving scheduling*. A priority-driven algorithm is greedy because it tries to make locally optimal decisions. Leaving a resource idle while some job is ready to use the resource is not locally optimal. So when a processor or resource is available and some job can use it to make progress, such an algorithm never makes the job wait. We will return shortly to illustrate that greed does not always pay; sometimes it is better to have some jobs wait even when they are ready to execute and the resources they require are available.

The term list scheduling is also descriptive because any priority-driven algorithm can be implemented by assigning priorities to jobs. Jobs ready for execution are placed in one or

Integre Technical Publishing Co., Inc. Liu January 13, 2000 8:46 a.m. chap4 page 63

more queues ordered by the priorities of the jobs. At any scheduling decision time, the jobs with the highest priorities are scheduled and executed on the available processors. Hence, a priority-driven scheduling algorithm is defined to a great extent by the list of priorities it assigns to jobs; the priority list and other rules, such as whether preemption is allowed, define the scheduling algorithm completely.

Most scheduling algorithms used in nonreal-time systems are priority-driven. Examples include the FIFO (First-In-First-Out) and LIFO (Last-In-First-Out) algorithms, which assign priorities to jobs according their release times, and the SETF (Shortest-Execution-Time-First) and LETF (Longest-Execution-Time-First) algorithms, which assign priorities on the basis of job execution times. Because we can dynamically change the priorities of jobs, even round-robin scheduling can be thought of as priority-driven: The priority of the executing job is lowered to the minimum among all jobs waiting for execution after the job has executed for a time slice.

Figure 4–2 gives an example. The task graph shown here is a classical precedence graph; all its edges represent precedence constraints. The number next to the name of each job is its execution time. J_5 is released at time 4. All the other jobs are released at time 0. We want to schedule and execute the jobs on two processors P_1 and P_2. They communicate via a shared memory. Hence the costs of communication among jobs are negligible no matter where they are executed. The schedulers of the processors keep one common priority queue of ready jobs. The priority list is given next to the graph: J_i has a higher priority than J_k if $i < k$. All the jobs are preemptable; scheduling decisions are made whenever some job becomes ready for execution or some job completes.

Figure 4–2(a) shows the schedule of the jobs on the two processors generated by the priority-driven algorithm following this priority assignment. At time 0, jobs J_1, J_2, and J_7 are ready for execution. They are the only jobs in the common priority queue at this time. Since J_1 and J_2 have higher priorities than J_7, they are ahead of J_7 in the queue and hence are scheduled. The processors continue to execute the jobs scheduled on them except when the following events occur and new scheduling decisions are made.

- At time 1, J_2 completes and, hence, J_3 becomes ready. J_3 is placed in the priority queue ahead of J_7 and is scheduled on P_2, the processor freed by J_2.
- At time 3, both J_1 and J_3 complete. J_5 is still not released. J_4 and J_7 are scheduled.
- At time 4, J_5 is released. Now there are three ready jobs. J_7 has the lowest priority among them. Consequently, it is preempted. J_4 and J_5 have the processors.
- At time 5, J_4 completes. J_7 resumes on processor P_1.
- At time 6, J_5 completes. Because J_7 is not yet completed, both J_6 and J_8 are not ready for execution. Consequently, processor P_2 becomes idle.
- J_7 finally completes at time 8. J_6 and J_8 can now be scheduled and they are.

Figure 4–2(b) shows a nonpreemptive schedule according to the same priority assignment. Before time 4, this schedule is the same as the preemptive schedule. However, at time 4 when J_5 is released, both processors are busy. It has to wait until J_4 completes (at time 5) before it can begin execution. It turns out that for this system this postponement of the higher priority job benefits the set of jobs as a whole. The entire set completes 1 unit of time earlier according to the nonpreemptive schedule.

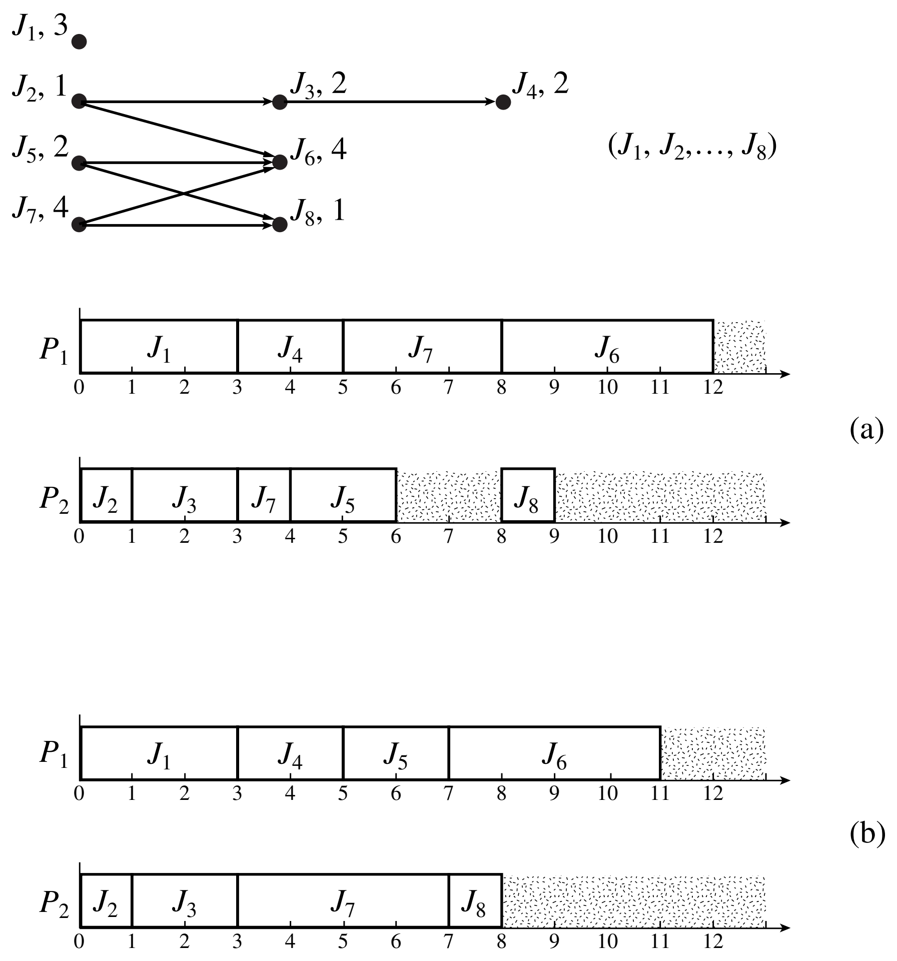


FIGURE 4–2 Example of priority-driven scheduling. (a) Preemptive (b) Nonpreemptive.

In general, however, nonpreemptive scheduling is not better than preemptive scheduling. A fundamental question is, when is preemptive scheduling better than nonpreemptive scheduling and vice versa? It would be good if we had some rule with which we could determine from the given parameters of the jobs whether to schedule them preemptively or nonpreemptively. Unfortunately, there is no known answer to this question in general. In the special case when jobs have the same release time, preemptive scheduling is better when the cost of preemption is ignored. Specifically, in a multiprocessor system, the minimum makespan (i.e., the response time of the job that completes last among all jobs) achievable by an optimal preemptive algorithm is shorter than the makespan achievable by an optimal nonpreemptive algorithm. A natural question here is whether the difference in the minimum makespans achievable by the two classes of algorithms is significant, in particular, whether the theoretical gain in makespan achievable by preemption is enough to compensate for the context switch overhead of preemption. The answer to this question is only known for the two-processor case. Coffman and

Integre Technical Publishing Co., Inc. Liu January 13, 2000 8:46 a.m. chap4 page 65

Garey [CoGa] recently proved that when there are two processors, the minimum makespan achievable by nonpreemptive algorithms is never more than 4/3 times the minimum makespan achievable by preemptive algorithms when the cost of preemption is negligible. The proof of this seemingly simple results is too lengthy to be included here.

4.4 DYNAMIC VERSUS STATIC SYSTEMS

In the above example, jobs that are ready for execution are placed in a priority queue common to all processors. When a processor is available, the job at the head of the queue executes on the processor. We will refer to such a multiprocessor system as a *dynamic system*, because jobs are *dynamically dispatched* to processors. In the example in Figure 4–2, we allowed each preempted job to resume on any processor and hence, jobs are migratable. We say that a job *migrates* if it starts execution on a processor, is preempted, and later resumes on a different processor.

Another approach to scheduling in multiprocessor and distributed systems is to partition the jobs in the system into subsystems and assign and bind the subsystems statically to the processors. Jobs are moved among processors only when the system must be reconfigured, that is, when the operation mode of the system changes or some processor fails. Such a system is called a *static system*, because the system is *statically configured*. If jobs on different processors are dependent, the schedulers on the processors must synchronize the jobs according to some synchronization and resource access-control protocol. Except for the constraints thus imposed, the jobs on each processor are scheduled by themselves.

As an example, a partition and assignment of the jobs in Figure 4–2 put J_1, J_2, J_3, and J_4 on P_1 and the remaining jobs on P_2. The priority list is segmented into two parts: (J_1, J_2, J_3, J_4) and (J_5, J_6, J_7, J_8). The scheduler of processor P_1 uses the former while the scheduler of processor P_2 uses the latter. It is easy to see that the jobs on P_1 complete by time 8, and the jobs on P_2 complete by time 11. Moreover, J_2 completes by time 4 while J_6 starts at time 6. Therefore, the precedence constraint between them is satisfied.

In this example, the response of the static system is just as good as that of the dynamic system. Intuitively, we expect that we can get better average responses by dynamically dispatching and executing jobs. In later chapters we will return to this discussion. Specifically, we will demonstrate that while dynamic systems may be more responsive on the average, their worst-case real-time performance may be poorer than static systems. More importantly, we do not yet have reliable techniques to validate the timing constraints of dynamic systems while such techniques exist for static systems. For this reason, most hard real-time systems built today are static.

4.5 EFFECTIVE RELEASE TIMES AND DEADLINES

The given release times and deadlines of jobs are sometimes inconsistent with the precedence constraints of the jobs. By this, we mean that the release time of a job may be later than that of its successors, and its deadline may be earlier than that of its predecessors. Therefore, rather than working with the given release times and deadlines, we first derive a set of effective release times and deadlines from these timing constraints, together with the given precedence constraints. The derived timing constraints are consistent with the precedence constraints.

When there is only one processor, we can compute the derived constraints according to the following rules:

Effective Release Time: The effective release time of a job without predecessors is equal to its given release time. The effective release time of a job with predecessors is equal to the maximum value among its given release time and the effective release times of all of its predecessors.

Effective Deadline: The effective deadline of a job without a successor is equal to its given deadline. The effective deadline of a job with successors is equal to the minimum value among its given deadline and the effective deadlines of all of its successors.

The effective release times of all the jobs can be computed in one pass through the precedence graph in $O(n^2)$ time where n is the number of jobs. Similarly, the effective deadlines can be computed in $O(n^2)$ time.

As an example, we look at the set of jobs in Figure 4–3. The numbers in the parentheses next to the name of each job are its given release times and deadlines. Because J_1 and J_2 have no predecessors, their effective release times are the given release times, that is, 2 and 0, respectively. The given release time of J_3 is 1, but the latest effective release time of its predecessors is 2, that of J_1. Hence, the effective release time of J_3 is 2. You can repeat this procedure and find that the effective release times of the rest of the jobs are 4, 2, 4, and 6, respectively. Similarly, J_6 and J_7 have no successors, and their effective deadlines are equal to their given deadlines, 20 and 21, respectively. Since the effective deadlines of the successors of J_4 and J_5 are larger than the given deadlines of J_4 and J_5, the effective deadlines of J_4 and J_5 are equal to their given deadlines. On the other hand, the given deadline of J_3 is equal to 12, which is larger than the minimum value of 8 among the effective deadlines of its successors. Hence, the effective deadline of J_3 is 8. In a similar way, we find that the effective deadlines of J_1 and J_2 are 8 and 7, respectively.

You may have noticed that the calculation of effective release times and deadlines does not take into account the execution times of jobs. More accurately, the effective deadline of a job should be as early as the deadline of each of its successors minus the execution time of the successor. The effective release time of a job is that of its predecessor plus the execution time of the predecessor. The more accurate calculation is unnecessary, however, when there is only one processor. Gary and Johnson [GaJo77] have shown that it is feasible to schedule any

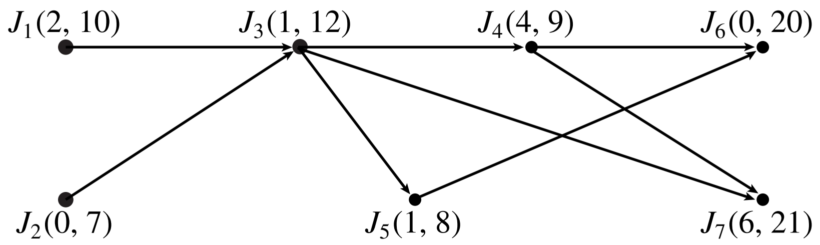


FIGURE 4–3 Example of effective timing constraints.

set of jobs on a processor according to their given release times and deadlines if and only if it is feasible to schedule the set according to their effective release times and deadlines defined above. When there is only one processor and jobs are preemptable, working with the effective release times and deadlines allows us to temporarily ignore the precedence constraints and treat all the jobs as if they are independent. Of course, by doing so, it is possible for an algorithm to produce an invalid schedule that does not meet some precedence constraint. For example, J_1 and J_3 in Figure 4–3 have the same effective release time and deadline. An algorithm which ignores the precedence constraint between them may schedule J_3 in an earlier interval and J_1 in a later interval. If this happens, we can always add a step to swap the two jobs, that is, move J_1 to where J_3 is scheduled and vice versa. This swapping is always possible, and it transforms an invalid schedule into a valid one.

Hereafter, by release times and deadlines, we will always mean effective release times and deadlines. When there is only one processor and jobs are preemptable, we will ignore the precedence constraints.

4.6 OPTIMALITY OF THE EDF AND LST ALGORITHMS

A way to assign priorities to jobs is on the basis of their deadlines. In particular, the earlier the deadline, the higher the priority. The priority-driven scheduling algorithm based on this priority assignment is called the *Earliest-Deadline-First* (*EDF*) algorithm. This algorithm is important because it is optimal when used to schedule jobs on a processor as long as preemption is allowed and jobs do not contend for resources. This fact is stated formally below.

THEOREM 4.1. When preemption is allowed and jobs do not contend for resources, the EDF algorithm can produce a feasible schedule of a set **J** of jobs with arbitrary release times and deadlines on a processor if and only if **J** has feasible schedules.

Proof. The proof is based on the following fact: Any feasible schedule of **J** can be systematically transformed into an EDF schedule (i.e., a schedule produced by the EDF algorithm). To see why, suppose that in a schedule, parts of J_i and J_k are scheduled in intervals I_1 and I_2, respectively. Furthermore, the deadline d_i of J_i is later than the deadline d_k of J_k, but I_1 is earlier than I_2 as shown in Figure 4–4.

There are two cases. In the first case, the release time of J_k may be later than the end of I_1. J_k cannot be scheduled in I_1; the two jobs are already scheduled on the EDF basis in these intervals. Hence, we need to consider only the second case where the release time r_k of J_k is before the end of I_1; without loss of generality, we assume that r_k is no later than the beginning of I_1.

To transform the given schedule, we swap J_i and J_k. Specifically, if the interval I_1 is shorter than I_2, as shown in Figure 4–4, we move the portion of J_k that fits in I_1 forward to I_1 and move the entire portion of J_i scheduled in I_1 backward to I_2 and place it after J_k. The result is as shown in Figure 4–4(b). Clearly, this swap is always possible. We can do a similar swap if the interval I_1 is longer than I_2 : We move the entire portion of J_k scheduled in I_2 to I_1 and place it before J_i and move the portion of J_i that fits in I_2 to the interval. The result of this swap is that these two jobs are now scheduled on the

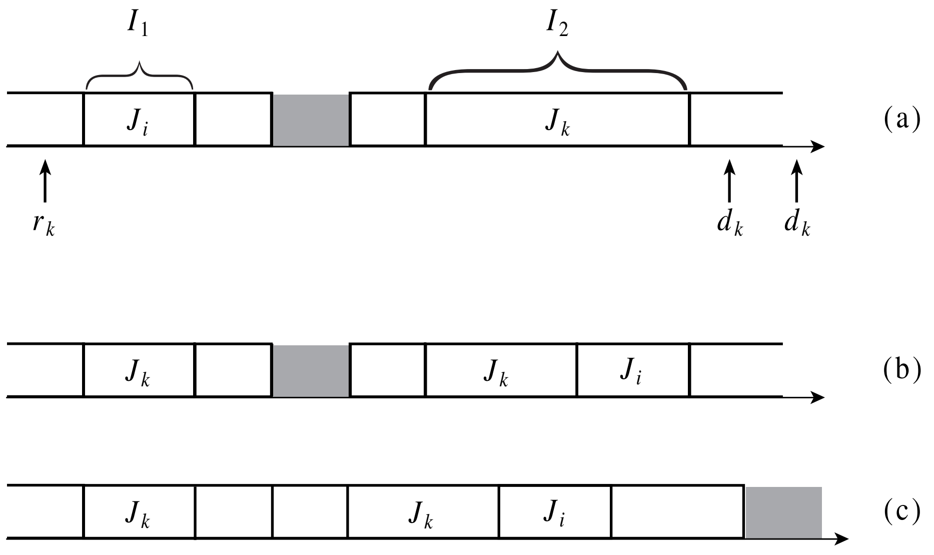


FIGURE 4–4 Transformation of a non-EDF schedule into an EDF schedule.

EDF basis. We repeat this transformation for every pair of jobs that are not scheduled on the EDF basis according to the given non-EDF schedule until no such pair exists.

The schedule obtained after this transformation may still not be an EDF schedule if some interval is left idle while there are jobs ready for execution but are scheduled in a later interval (e.g., as in the schedule in Figure 4–4(b).) We can eliminate such an idle interval by moving one or more of these jobs forward into the idle interval and leave the interval where the jobs were scheduled idle. This is clearly always possible. We repeat this process if necessary until the processor never idles when there are jobs ready for execution as in Figure 4–4(c).

That the preemptive EDF algorithm can always produce a feasible schedule as long as feasible schedules exist follows straightforwardly from the fact that every feasible schedule can be transformed into a preemptive EDF schedule. If the EDF algorithm fails to produce a feasible schedule, then no feasible schedule exists. (If a feasible schedule were to exist, it could be transformed into an EDF schedule, which contradicts the statement that the EDF algorithm fails to produce a feasible schedule.) □

When the goal of scheduling is to meet deadlines, there is no advantage to completing any job sooner than necessary. We may want to postpone the execution of hard real-time jobs for some reason (e.g., to enable soft real-time jobs, whose response times are important, to complete earlier). For this reason, we sometimes also use the *latest release time (LRT)* algorithm (or reverse EDF algorithm). This algorithm treats release times as deadlines and deadlines as release times and schedules the jobs backwards, starting from the latest deadline of all jobs, in "priority-driven" manner, to the current time. In particular, the "priorities" are based on the release times of jobs: the later the release time, the higher the "priority." Because it may leave the processor idle when there are jobs ready for execution, the LRT algorithm is not a priority-driven algorithm.

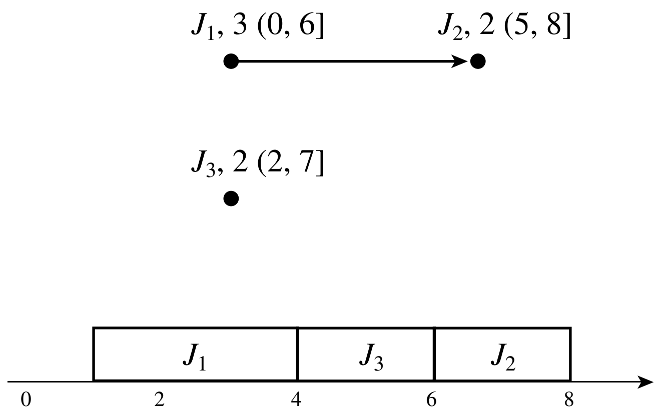


FIGURE 4–5 Example illustrating the LRT algorithm.

Figure 4–5 gives an example. In the precedence graph, the number next to the job name is the execution time of the job. Its feasible interval is given by the range of time next to its execution time. The latest deadline among all jobs is 8. Hence time starts at 8 and goes backwards to 0. At time 8, J_2 is "ready" and is scheduled. At time 7, J_3 is also "ready" to be scheduled, but because J_2 has a later release time, it has a higher priority. Consequently, J_2 is scheduled from 7 to 6. When J_2 "completes" at time 6, J_1 is "ready." However, J_3 has a higher priority and is, therefore, scheduled from 6 to 4. Finally J_1 is scheduled from 4 to 1. The result is a feasible schedule.

The following corollary states that the LRT algorithm is also optimal under the same conditions that the EDF algorithm is optimal. Its proof follows straightforwardly from the proof of Theorem 4.1.

COROLLARY 4.2. When preemption is allowed and jobs do not contend for resources, the LRT algorithm can produce a feasible schedule of a set **J** of jobs with arbitrary release times and deadlines on a processor if and only if feasible schedules of **J** exist.

Another algorithm that is optimal for scheduling preemptive jobs on one processor is the *Least-Slack-Time-First* (*LST*) algorithm (also called the *Minimum-Laxity-First* (*MLF*) algorithm) [LeWh, Mok]. At any time t, the *slack* (or *laxity*) of a job with deadline at d is equal to $d - t$ minus the time required to complete the remaining portion of the job. Take the job J_1 in Figure 4–5 as an example. It is released at time 0, its deadline is 6, and its execution time is 3. Hence, its slack is equal to 3 at time 0. The job starts to execute at time 0. As long as it executes, its slack remains at 3, because at any time t before its completion, its slack is $6 - t - (3 - t)$. Now suppose that it is preempted at time 2 by J_3, which executes from time 2 to 4. During this interval, the slack of J_1 decreases from 3 to 1. (At time 4, the remaining execution time of J_1 is 1, so its slack is $6 - 4 - 1 = 1$.)

The LST algorithm assigns priorities to jobs based on their slacks: the smaller the slack, the higher the priority. The following theorem states that the LST algorithm is also optimal. Its proof is left as an exercise at the end of this chapter.

THEOREM 4.3. When preemption is allowed and jobs do not contend for resources, the LST (MLF) algorithm can produce a feasible schedule of a set **J** of jobs with arbitrary release times and deadlines on a processor if and only if feasible schedules of **J** exist.

While the EDF algorithm does not require any knowledge of the execution times of jobs, the LST algorithm does. This is a serious disadvantage. As discussed earlier, the actual execution times of jobs are often not known until the jobs complete. Obviously, it is impossible for us to calculate the actual amounts of slack under this circumstance. We typically calculate the slack of each job based on its maximum execution time $e_i^+$when the range $[e_i^-, e_i^+]$ of execution time e_i of every job is relatively small. Furthermore, we require that the maximum (and sometimes even the actual) execution time of each sporadic or aperiodic job become known upon its arrival since this knowledge is needed for slack computation.

4.7 NONOPTIMALITY OF THE EDF AND THE LST ALGORITHMS

It is natural to ask here whether the EDF and the LST algorithms remain optimal if preemption is not allowed or there is more than one processor. Unfortunately, the answer is no.

The fact that the EDF and the LST algorithms are optimal only when preemption is allowed is illustrated by the example in Figure 4–6. The system shown in this figure has three independent, nonpreemptable jobs J_1, J_2, and J_3. Their release times are 0, 2 and 4, respectively, and are indicated by the arrows above the schedules. Their execution times are

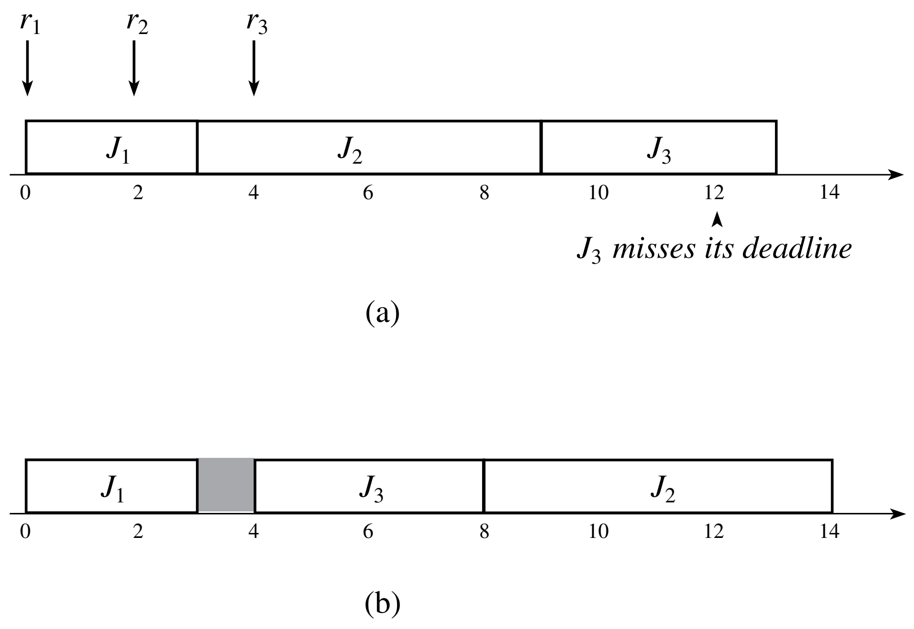


FIGURE 4–6 Example illustrating nonoptimality of the nonpreemptive EDF algorithm. (a) An EDF schedule. (b) A non-EDF schedule.

3, 6, and 4; and their deadlines are 10, 14, and 12, respectively. Figure 4–6(a) shows the schedule produced by the EDF algorithm. In particular, when J_1 completes at time 3, J_2 has already been released but not J_3. Hence, J_2 is scheduled. When J_3 is released at time 4, J_2 is executing. Even though J_3 has an earlier deadline and, hence, a higher priority, it must wait until J_2 completes because preemption is not allowed. As a result, J_3 misses its deadline. It is easy to see that the LST algorithm would produce the same infeasible schedule. The fact that these three jobs can meet their deadlines is demonstrated by the feasible schedule in Figure 4–6(b). At time 3 when J_1 completes, the processor is left idle, even though J_2 is ready for execution. Consequently, when J_3 is released at 4, it can be scheduled ahead of J_2, allowing both jobs to meet their deadlines.

We note that the schedule in Figure 4–6(b) cannot be produced by any priority-driven scheduling algorithm. By definition, a priority-driven algorithm never leaves a processor idle when there are jobs ready to use the processor. This example illustrates the fact that not only nonpreemptive EDF and LST algorithms are not optimal, but also no nonpreemptive priority-driven algorithm is optimal when jobs have arbitrary release times, execution times, and deadlines.

The example in Figure 4–7 shows that the EDF algorithm is not optimal for scheduling preemptable jobs on more than one processor. The system in this figure also contains three jobs, J_1, J_2, and J_3. Their execution times are 1, 1, and 5 and their deadlines are 1, 2, and 5, respectively. The release times of all three jobs are 0. The system has two processors. According to the EDF algorithm, J_1 and J_2 are scheduled on the processors at time 0 because they have higher priorities. The result is the schedule in Figure 4–7(a), and J_3 misses its deadline.

On the other hand, an algorithm which assigns a higher priority to J_3 in this case can feasibly schedule the jobs. An example of such algorithms is the LST algorithm. The slacks of the J_1, J_2, and J_3 in Figure 4–7 are 0, 1, and 0, respectively. Hence, this algorithm would

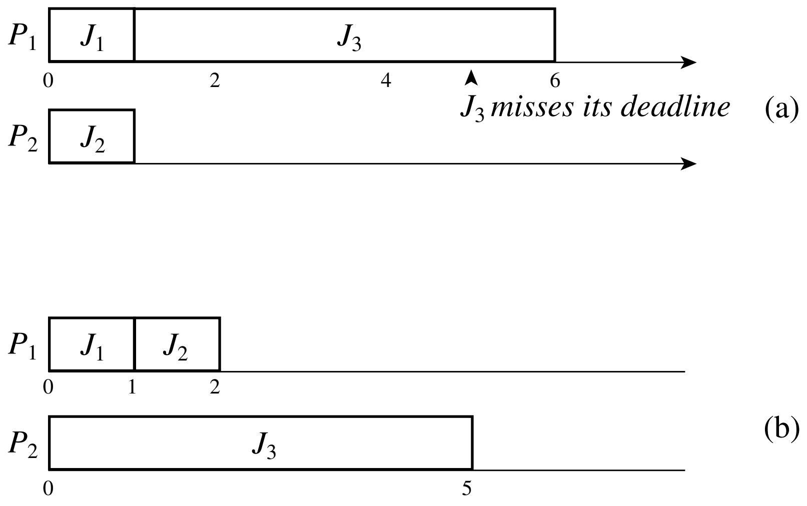


FIGURE 4–7 Example illustrating nonoptimality of the EDF algorithm for multiprocessor scheduling. (a) The EDF schedule. (b) A feasible schedule.

Integre Technical Publishing Co., Inc. Liu January 13, 2000 8:46 a.m. chap4 page 72

produce the feasible schedule in Figure 4–7(b). Unfortunately, the LST algorithm is also not optimal for scheduling jobs on more than one processor, as demonstrated by the example in Problem 4.4.

4.8 CHALLENGES IN VALIDATING TIMING CONSTRAINTS IN PRIORITY-DRIVEN SYSTEMS

Compared with the clock-driven approach, the priority-driven scheduling approach has many advantages. As examples, you may have noticed that priority-driven schedulers are easy to implement. Many well-known priority-driven algorithms use very simple priority assignments, and for these algorithms, the run-time overhead due to maintaining a priority queue of ready jobs can be made very small. A clock-driven scheduler requires the information on the release times and execution times of the jobs a priori in order to decide when to schedule them. In contrast, a priority-driven scheduler does not require most of this information, making it much better suited for applications with varying time and resource requirements. You will see in later chapters other advantages of the priority-driven approach which are at least as compelling as these two.

Despite its merits, the priority-driven approach has not been widely used in hard real-time systems, especially safety-critical systems, until recently. The major reason is that the timing behavior of a priority-driven system is nondeterministic when job parameters vary. Consequently, it is difficult to validate that the deadlines of all jobs scheduled in a priority-driven manner indeed meet their deadlines when the job parameters vary. In general, this *validation problem* [LiHa] can be stated as follows: Given a set of jobs, the set of resources available to the jobs, and the scheduling (and resource access-control) algorithm to allocate processors and resources to jobs, determine whether all the jobs meet their deadlines.

4.8.1 Anomalous Behavior of Priority-Driven Systems

Figure 4–8 gives an example illustrating why the validation problem is difficult when the scheduling algorithm is priority-driven and job parameters may vary. The simple system contains four independent jobs. The jobs are scheduled on two identical processors in a priority-driven manner. The processors maintain a common priority queue, and the priority order is J_1, J_2, J_3, and J_4 with J_1 having the highest priority. In other words, the system is dynamic. The jobs may be preempted but never migrated, meaning that once a job begins execution on a processor, it is constrained to execute on that processor until completion. (Many systems fit this simple model. For example, the processors may model two redundant data links connecting a source and destination pair, and the jobs are message transmissions over the links. The processors may also model two replicated database servers, and the jobs are queries dynamically dispatched by a communication processor to the database servers. The release times, deadlines, and execution times of the jobs are listed in the table.) The execution times of all the jobs are fixed and known, except for J_2. Its execution time can be any value in the range [2, 6].

Suppose that we want to determine whether the system meets all the deadlines and whether the completion-time jitter of every job (i.e., the difference between the latest and the earliest completion times of the job) is no more than 4. A brute force way to do so is to simulate the system. Suppose that we schedule the jobs according their given priorities, assuming

	r_i	d_i	$[e_i^-, e_i^+]$
J_1	0	10	5
J_2	0	10	[2, 6]
J_3	4	15	8
J_4	0	20	10

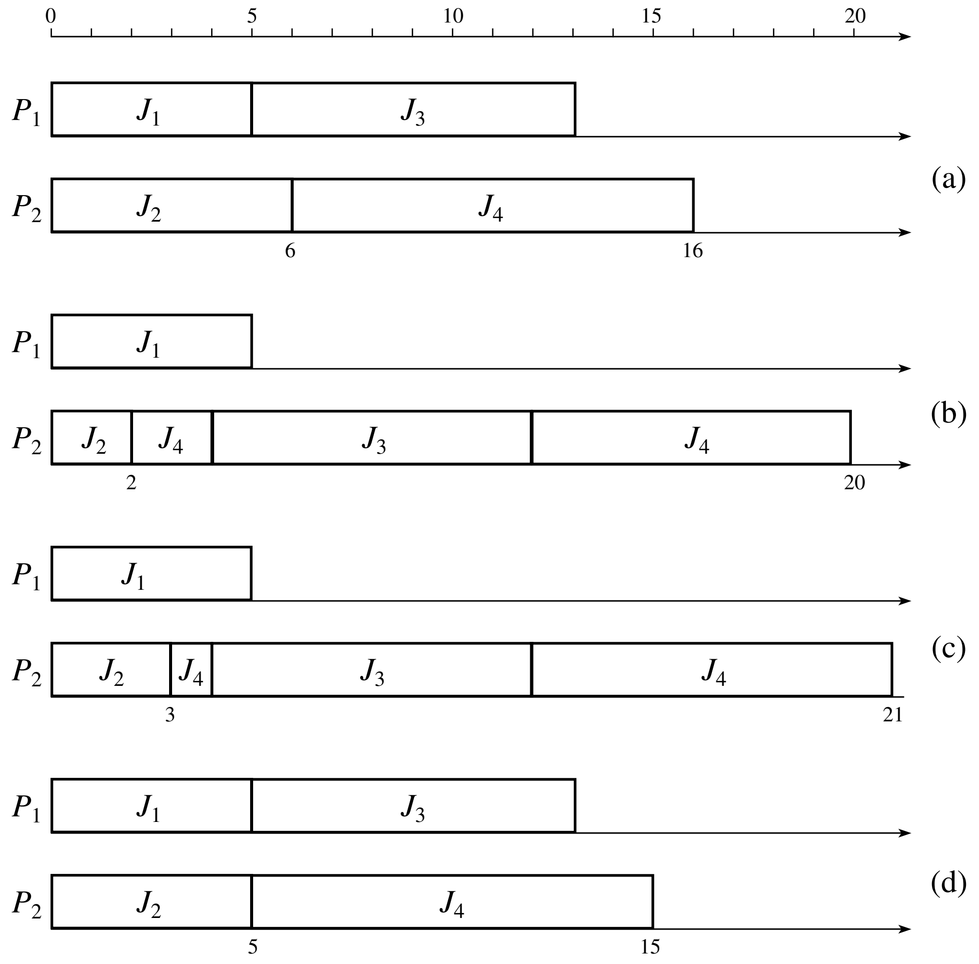


FIGURE 4–8 Example illustrating scheduling anomalies.

first that the execution time of J_2 has the maximum value 6 and then that it has the minimum value 2. The resultant schedules are shown in Figure 4–8(a) and (b), respectively. Looking at these schedules, we might conclude that all jobs meet their deadlines, and the completion-time jitters are sufficiently small. This would be an incorrect conclusion, as demonstrated by the schedules in Figure 4–8(c) and (d). As far as J_4 is concerned, the worst-case schedule is the one shown in Figure 4–8(c); it occurs when the execution time of J_2 is 3. According to this schedule, the completion time of J_4 is 21; the job misses its deadline. The best-case schedule for J_4 is shown in Figure 4–8(d); it occurs when the execution time of J_2 is 5. From this schedule, we see that J_4 can complete as early as time 15; its completion-time jitter exceeds the upper limit of 4. To find the worst-case and best-case schedules, we must try all the possible values of e_2.

The phenomenon illustrated by this example is known as a *scheduling anomaly*, an unexpected timing behavior of priority-driven systems. Graham [Grah] has shown that the completion time of a set of nonpreemptive jobs with identical release times can be later when more processors are used to execute them and when they have shorter execution times and fewer dependencies. (Problem 4.5 gives the well-known illustrative example.) Indeed, when jobs are nonpreemptable, scheduling anomalies can occur even when there is only one processor. For example, suppose that the execution time e_1 of the job J_1 in Figure 4–6 can be either 3 or 4. The figure shows that J_3 misses its deadline when e_1 is 3. However, J_3 would complete at time 8 and meet its deadline if e_1 were 4. We will see in later chapters that when jobs have arbitrary release times and share resources, scheduling anomalies can occur even when there is only one processor and the jobs are preemptable.

Scheduling anomalies make the problem of validating a priority-driven system difficult whenever job parameters may vary. Unfortunately, variations in execution times and release times are often unavoidable. If the maximum range of execution times of all n jobs in a system is X, the time required to find the latest and earliest completion times of all jobs is $O(X^n)$ if we were to find these extrema by exhaustive simulation or testing. Clearly, such a strategy is impractical for all but the smallest systems of practical interest.

4.8.2 Predictability of Executions

When the timing constraints are specified in terms of deadlines of jobs, the validation problem is the same as that of finding the worst-case (the largest) completion time of every job. This problem is easy whenever the execution behavior of the set **J** is predictable, that is, whenever the system does not have scheduling anomalies. To define predictability more formally, we call the schedule of **J** produced by the given scheduling algorithm when the execution time of every job has its maximum value the *maximal schedule* of **J**. Similarly, the schedule of **J** produced by the given scheduling algorithm when the execution time of every job has its minimum value is the *minimal schedule*. When the execution time of every job has its actual value, the resultant schedule is the *actual schedule* of **J**. So, the schedules in Figure 4–8(a) and (b) are the maximal and minimal schedules, respectively, of the jobs in that system, and all the schedules shown in the figure are possible actual schedules.

Since the range of execution time of every job is known, the maximal and minimal schedules of **J** can easily be constructed when the release-time jitters are negligible. (We assume that release times of all jobs are fixed and known for this discussion. How release-time jitters further complicate the validation problem is beyond the scope of our discussion here.) In contrast, its actual schedule is unknown because the actual values of the execution times are unknown.

The predictability of job execution (behavior) is an intuitive notion: The execution of **J** under the given priority-driven scheduling algorithm is predictable if the actual start time and actual completion time of every job according to the actual schedule are bounded by its start times and completion times according to the maximal and minimal schedules. More formally, we let $s(J_i)$ be the instant of time at which the execution of J_i begins according to the actual schedule of **J**. $s(J_i)$ is the (actual) start time of J_i. Let $s^+(J_i)$ and $s^-(J_i)$ be the start times of J_i according to the maximal schedule and minimal schedule of **J**, respectively. These start times can easily be found by constructing the maximal and minimal schedules and observing when J_i starts according to these schedules. We say that J_i is *start-time predictable* if $s^-(J_i) \leq s(J_i) \leq s^+(J_i)$. As an example, for the job J_4 in Figure 4–8, $s^-(J_4)$ is 2. $s^+(J_4)$ is 6. Its actual start time is in the range [2, 6]. Therefore, J_4 is start-time predictable.

Similarly, let $f(J_i)$ be the actual completion time (also called finishing time) of J_i according to the actual schedule of **J**. Let $f^+(J_i)$ and $f^-(J_i)$ be the completion times of J_i according to the maximal schedule and minimal schedule of **J**, respectively. We say that J_i is *completion-time predictable* if $f^-(J_i) \leq f(J_i) \leq f^+(J_i)$. The execution of J_i is *predictable*, or simply J_i is predictable, if J_i is both start-time and completion-time predictable. The execution behavior of the entire set **J** is predictable if every job in **J** is predictable. Looking at Figure 4–8 again, we see that $f^-(J_4)$ is 20, but $f^+(J_4)$ is 16. It is impossible for the inequality $20 \leq f(J_4) \leq 16$ to hold. Therefore, J_4 is not completion-time predictable, and the system is not predictable.

In general, whether a set of jobs has predictable execution behavior depends not only on the parameters of jobs in the set but also on the algorithm used to schedule the jobs. For example, while the execution of independent, preemptable but nonmigratable jobs is not predictable in general, as exemplified by the jobs in Figure 4–8, it is predictable when the priorities are assigned on the FIFO basis [Ha]. In Chapters 6–9, we will use the following fact, which is true for all priority-driven scheduling algorithms.

THEOREM 4.4. The execution of every job in a set of independent, preemptable jobs with fixed release times is predictable when scheduled in a priority-driven manner on one processor.

Proof. That the highest priority job J_1 is predictable is obvious: It always starts at its release time, and its maximum execution time is larger than its actual execution time. Suppose that all the $i - 1$ highest priority jobs $J_1, J_2, \ldots, J_{i-1}$ are predictable. We now show by contradiction that J_i, which has a lower priority than they but a higher priority than all the other jobs in the system, is also predictable.

Suppose that $s^-(J_i) \leq s(J_i) \leq s^+(J_i)$ is not true. In particular, we suppose that $s(J_i) > s^+(J_i)$. Because the scheduler never schedules a job before its release time, $s^+(J_i)$ is no earlier than the release r_i of J_i. Because the scheduling algorithm is priority-driven, every job whose release time is at or earlier than $s^+(J_i)$ and whose priorities are higher than J_i has completed by $s^+(J_i)$ according to the maximal schedule. By induction hypothesis, we know that every such job has completed by $s^+(J_i)$ according to the actual schedule as well. $s(J_i) > s^+(J_i)$ means either that the processor is left idle in the interval $[s^+(J_i), s(J_i)]$ or a job with a priority lower than J_i is scheduled in this interval. This contradicts the fact that the algorithm used to schedule the jobs is priority-driven. Hence, $s(J_i) > s^+(J_i)$ cannot be true.

A similar argument based on the actual and minimal schedules allows us to conclude that $s^-(J_i) > s(J_i)$ cannot be true. In other words, J_i is start-time predictable.

To show that J_i is also completion-time predictable, we note that since $s(J_i) \leq s^+(J_i)$, there is more available time in the interval $[s(J_i), f^+(J_i)]$ than in the interval $[s^+(J_i), f^+(J_i)]$. Moreover, the actual execution time of every job is never larger than the maximum execution time of the job. If J_i remains incomplete at $f^+(J_i)$ according to the actual schedule while it can complete by this time according to the maximal schedule, it must be delayed by one or more jobs with priorities lower than J_i executing in $[s(J_i), f^+(J_i)]$, or the processor must be left idle for some time in this interval. This contradicts the fact that the scheduling algorithm is priority-driven. We can, therefore, conclude that $f(J_i) \leq f^+(J_i)$. A similar argument based on the actual and minimal schedules tells us that $f^-(J_i)$ is never later than $f(J_i)$, or that J_i is also completion-time predictable. $\square$

Theorem 4.4 tells us that it is relatively easy to validate priority-driven, uniprocessor, static systems when jobs are independent and preemptable. Because the execution behavior of all the jobs is predictable, we can confine our attention to the maximum execution times of all the jobs and ignore the variations in execution times when we want to determine their maximum possible response times. You will see that this is indeed what we will do in most parts of Chapters 6–9. Nonpreemptivity and resource contention invariably introduce unpredictability in execution. Fortunately, it is possible to bound the additional delay suffered by every job due to these factors reasonably accurately. We will describe the methods for doing so as well.

4.8.3 Validation Algorithms and Their Performance

The validation problem has many variants, and that of validating static priority-driven systems is an important variant. Recent advances in real-time scheduling and schedulability analysis have lead to several sufficient conditions and analytical bounds. They are solutions to this variant and the subjects of discussion in Chapters 6–9. These theoretical results form a rigorous basis of good validation algorithms and tools for a wide spectrum of static systems. (A *validation algorithm* allows us to determine whether all jobs in a system indeed meet their timing constraints despite scheduling anomalies.) While there are mature validation algorithms and tools for static systems, good validation algorithms for dynamic, priority-driven systems are not yet available.

Specifically, we say that a validation algorithm is *correct* if it never declares that all timing constraints are met when some constraints may not be. The merits of (correct) validation algorithms are measured in terms of their complexity, robustness, and accuracy. A validation algorithm is good when it achieves a good balance in performance according to these conflicting figures of merit.

For example, some existing validation algorithms run in constant time or $O(n)$ time, where n is the number of tasks in the system. They are well suited for on-line acceptance tests to determine whether the system should admit a new task. More complex ones run in pseudopolynomial time but have better performance in the other dimensions.

Every rigorous validation algorithm is based on a workload model. When applied to a system, the conclusion of the algorithm is correct if all the assumptions of the model are valid for the system. A validation algorithm is said to be *robust* if it remains correct even when some assumptions of its underlying workload model are not valid. The use of a robust validation algorithm significantly reduces the need for an accurate characterization of the applications and the run-time environment and, thus, the efforts in analysis and measurement of the individual applications for the purpose of validating the workload model. We will see in later chapters that existing validation algorithms based on the periodic task model are robust to a great extent. Although the model assumes that jobs in each task are released periodically and execute for an equal amount of time, such a validation algorithm remains correct in the presence of release-time jitters, variations in job execution time, and other deviations from periodic behavior. It is only necessary for us to know the ranges of task parameters (e.g., the minimum interrelease time and maximum execution time of jobs), which are much easier to obtain and validate, either by timing analysis or measurement, than the actual values or probability distributions of the parameters.

Efficiency and robustness can be achieved easily if we are not concerned with the accuracy of the validation test. A validation algorithm is *inaccurate* when it is overly pessimistic and declares tasks unable to meet their timing constraints except when system resources are unduly underutilized. A scheduler using an inaccurate validation algorithm for an acceptance test may reject too many new tasks which are in fact acceptable. Because most validation algorithms are based on conditions that are sufficient but not necessary, they are all inaccurate to some degree, which is the price paid for the sake of robustness. The accuracy of a validation algorithm depends on whether the actual characteristics of the application systems are accurately captured by the underlying workload model. For example, validation algorithms that are based on the periodic task model are sufficiently accurate for applications, such as digital control and constant bit-rate voice and video communications, which are well characterized by the periodic task model but may have poor accuracy when used to validate applications that have widely varying processor-time demands and large release-time jitters.

4.9 OFF-LINE VERSUS ON-LINE SCHEDULING

In Section 4.1, we mentioned that a clock-driven scheduler typically makes use of a precomputed schedule of all hard real-time jobs. This schedule is computed off-line before the system begins to execute, and the computation is based on the knowledge of the release times and processor-time/resource requirements of all the jobs for all times. When the operation mode of the system changes, the new schedule specifying when each job in the new mode executes is also precomputed and stored for use. In this case, we say that scheduling is (done) off-line, and the precomputed schedules are *off-line schedules*.

An obvious disadvantage of off-line scheduling is inflexibility. This approach is possible only when the system is deterministic, meaning that the system provides some fixed set(s) of functions and that the release times and processor-time/resource demands of all its jobs are known and do not vary or vary only slightly. For a deterministic system, however, off-line scheduling has several advantages, the deterministic timing behavior of the resultant system being one of them. Because the computation of the schedules is done off-line, the complexity of the scheduling algorithm(s) used for this purpose is not important. Indeed, as we will see in the next chapter, complex heuristic algorithms are typically used to find good off-line schedules that can make nearly full use of the resources.

Competitiveness of On-Line Scheduling. We say that scheduling is done *on-line*, or that we use an *on-line scheduling algorithm*, if the scheduler makes each scheduling decision without knowledge about the jobs that will be released in the future; the parameters of each job become known to the on-line scheduler only after the job is released. The priority-driven algorithms described earlier and in subsequent chapters are on-line algorithms. In Chapter 2 we talked about the admission of each new task depending on the outcome of an acceptance test that is based on the parameters of the new task and tasks admitted earlier. Such an acceptance test is on-line.

Clearly, on-line scheduling is the only option in a system whose future workload is unpredictable. An on-line scheduler can accommodate dynamic variations in user demands and resource availability. The price of the flexibility and adaptability is a reduced ability for the scheduler to make the best use of system resources. Without prior knowledge about future jobs, the scheduler cannot make optimal scheduling decisions while a clairvoyant scheduler that knows about all future jobs can.

As a simple example, suppose that at time 0, a nonpreemptive job J_1 with execution time 1 and deadline 2 is released. An on-line scheduler has two options at time 0: It either schedules J_1 to start execution at time 0 or it postpones the execution of J_1 to some later time. Suppose that the on-line scheduler decides to schedule J_1 at time 0. Later at time $x < 1$, a job J_2 with execution time $1 - x$ and deadline 1 is released. J_2 would miss its deadline because it cannot start execution until time 1. In contrast, a clairvoyant scheduler, which knows J_2 at time 0, would schedule J_1 to start execution at time 1 and thus allow both jobs to complete in time. In the second case, the on-line scheduler decides to postpone the execution of J_1 until some later time $x < 1$. Now suppose that at time x, J_3 is released instead of J_2. The execution time of J_3 is 1, and its deadline is 2. It is impossible for the on-line scheduler to schedule both J_1 and J_3 so that they complete in time. Again, a clairvoyant scheduler, knowing the future release of J_3 at time 0, would schedule J_1 to start execution at time 0 so it can complete both J_1 and J_3 on time.

The system is said to be *overloaded* when the jobs offered to the scheduler cannot be feasibly scheduled even by a clairvoyant scheduler. When the system is not overloaded, an optimal on-line scheduling algorithm is one that always produces a feasible schedule of all offered jobs. The example above shows that *no optimal on-line scheduling algorithm exists when some jobs are nonpreemptable*. On the other hand, if all the jobs are preemptable and there is only one processor, optimal on-line algorithms exist, and the EDF and LST algorithms are examples.

During an overload, some jobs must be discarded in order to allow other jobs to complete in time. A reasonable way to measure the performance of a scheduling algorithm during an overload is by the amount of work the scheduler can feasibly schedule according to the algorithm: the larger this amount, the better the algorithm. The competitive factor of an algorithm captures this aspect of performance. To define this performance measure, we say that the *value of a job* is equal to its execution time if the job completes by its deadline according to a given schedule and is equal to zero if the job fails to complete in time according to the schedule. The *value of a schedule* of a sequence of jobs is equal to the sum of the values of all the jobs in the sequence according to the schedule. A scheduling algorithm is optimal if it always produces a schedule of the maximum possible value for every finite set of jobs. An on-line algorithm has a *competitive factor c* if and only if the value of the schedule of any finite sequence of jobs produced by the algorithm is at least c times the value of the schedule of the jobs produced by an optimal clairvoyant algorithm.

In terms of this performance measure, EDF and LST algorithms are optimal under the condition that the jobs are preemptable, there is only one processor, and the processor is not overloaded. Their competitive factors are equal to 1 under this condition. On other hand, when the system is overloaded, their competitive factors are 0. To demonstrate, let us consider two jobs. The first one is released at time 0, and its execution time is 2ε; the deadline is ε for some arbitrarily small positive number ε. At time ε, a job whose relative deadline is equal to its execution time e is released. The value achieved by the EDF or LST algorithm is 0, while the maximum possible value achievable is e.

As it turns out, the EDF and LST algorithms are not the only algorithms with poor performance when the system is overloaded. In general, all on-line scheduling algorithms perform rather poorly. The following theorem due to Baruah, *et al.* [BKMM] gives us the performance limitation of on-line scheduling when the system is overloaded.

THEOREM 4.5. No on-line scheduling algorithm can achieve a competitive factor greater than 0.25 when the system is overloaded.

Informal Proof of Theorem 4.5. To gain some insight into why this upper bound of 0.25 is true, we summarize the proof of Theorem 4.5; you can find the complete formal proof in [BKMM]. Suppose that there is an adversary of the on-line scheduler. Over time, the adversary creates two kinds of jobs and offers (i.e., releases) them to the scheduler: major jobs and jobs associated with major jobs. The relative deadline of every job is equal to its execution time. (In other words, the job has no slack; the scheduler should either schedule the job immediately after it is released, or discard it.) We name the major jobs J_i for $i = 0, 1, \ldots, max$ in increasing order of their release times and denote the execution time and release time of each job J_i by e_i and r_i, respectively. (max is some positive integer that we will define shortly.) The adversary creates a sequence of jobs associated with each major job J_i. The execution times of all the associated jobs in the sequence are equal to some small number $\varepsilon > 0$, which is negligible compared with e_i. The first job in the sequence associated with each major job J_i is released at the same time with J_i. If there is more than one associated job in a sequence, each subsequent associated job is released at the deadline of the previous associated job in the sequence. The number of associated jobs in the sequence depends on the action of the on-line scheduler. Specifically, as long as the on-line scheduler chooses to execute J_i, the adversary continues to create jobs associated with J_i until the deadline d_i of J_i. Whenever the scheduler decides to execute a job associated with J_i, the adversary stops releasing any more associated jobs.

Let us now consider a busy interval which begins at r_0 when the adversary releases the first major job J_0 and the first of the sequence of jobs associated with J_0. Depending on the action of the on-line scheduler, the adversary may release major job J_i for $i > 0$ at time r_i; the release time r_i of the ith major job and the first of its associated jobs is equal to $r_{i-1} + e_{i-1} - \varepsilon$. In other words, J_i is released at ε units of time before the deadline of J_{i-1}. It is not possible to schedule both J_{i-1} and J_i to complete in time. At the time r_i, the scheduler must choose either to discard J_{i-1} and start to execute J_i or continue to execute J_{i-1} and therefore discard J_i.

If the scheduler chooses to discard J_{i-1} and begins to execute J_i, the adversary then releases J_{i+1} at r_{i+1}. As long as the scheduler continues to discard the executing major job each time a new major job is released in order to execute the new job, the adversary continues to release the next major job ε units of time before the deadline of the executing job. This process continues until the major job J_{max} is released for some positive integer max, and the busy interval ends at the deadline of this job. In this case, the on-line scheduler discards all the jobs but J_{max}, and the total value achieved by the scheduler in this busy interval is e_{max}. In contrast, the clairvoyant scheduler would schedule the jobs associated with all the major jobs before J_{max} and then the major job J_{max} and achieve the value $\sum_{k=1}^{max} e_k$.

On the other hand, the scheduler may decide to complete J_i and, upon the completion of J_i, execute a job associated with J_{i+1} for $i < max$. In this case, the adversary stops releasing any major job after J_{i+1}. Moreover, it stops releasing jobs associated with J_{i+1} after the first associated job. (Figure 4–9 shows this situation for $i = 2$.) The value achieved in the busy interval by the on-line scheduler is approximately equal to e_i. However, the clairvoyant scheduler would execute all jobs associated with jobs J_0, $J_1, \ldots, J_{i-1}$ and then the job J_i and achieve a value of $\sum_{k=0}^{i} e_i$.

Now suppose that the execution time e_0 of the first major job J_0 is 1, and for $i > 0$, the execution time e_i of the ith major job is given by

$$e_i = ce_{i-1} - \sum_{k=0}^{i-1} e_k$$

If we perform the necessary algebraic manipulation, we will find that competitive factor of the on-line scheduler is either equal to $1/c$, if the scheduler completes J_i followed by an associate job, or is equal to the ratio of e_{max} to the sum $\sum_{k=0}^{max} e_k$, if the scheduler discards all the major jobs except J_{max}. The former is always greater than or equal to the latter for every positive integer max if c is equal to 4 or more. For c equal to 4,

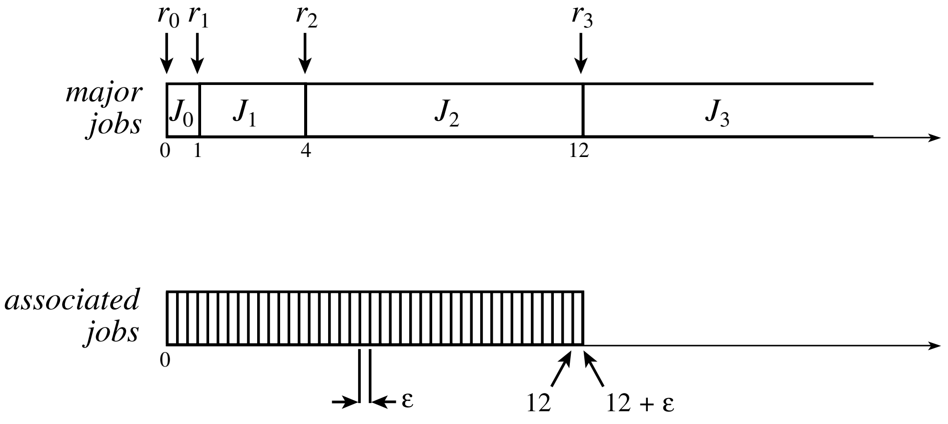


FIGURE 4–9 Example illustrating competitiveness of on-line scheduling.

the competitive factor is 0.25. The execution times of the major jobs are equal to 1, 3, 8, 20, 48, ... Figure 4–9 shows the case where the on-line scheduler executes J_2 to completion. It achieves a value of 8 while a clairvoyant scheduler, knowing that all the jobs shown in the figure will be released at all time, can achieve a value of 32. □

The system used in the proof of Theorem 4.5 is extremely overloaded. It is not surprising that the performance of on-line scheduling algorithms is poor when the system is so overloaded. Intuitively, we expect that some on-line algorithms should perform well when the system is only slightly overloaded. To state this more precisely, we let $x(t, t')$ $(t' > t)$ denote the *load ratio* of the interval $[t, t']$: It is the ratio of the total execution time of all jobs whose feasible intervals are contained in the interval $[t, t']$ to the length $t' - t$ of the interval. A system is said to have a *loading factor X* if the load ratio of the system is equal to or less than X for all intervals. Our intuition says that if the loading factor of a system is $1 + \varepsilon$ for some very small positive number ε, there should be on-line algorithms whose competitiveness factors are close to 1. Unfortunately, our intuition fails us. Baruah, *et al.* showed that the competitiveness factor of an on-line scheduling algorithm is at most equal to 0.385 for any system whose loading factor is just slightly over 1.

The results on competitiveness of on-line algorithms tell us that when scheduling is done on-line, it is important to keep the system from being overloaded using some overload management or load shedding algorithms. Most overload management algorithms take into account the criticality factors of jobs, not just their timing parameters, when choosing jobs to be discarded. We will describe a few of them in later chapters.

4.10 SUMMARY

This chapter gave a brief overview of the clock-driven, weighted round-robin and priority-driven approaches to scheduling. They are the subjects of in-depth discussion of the next few chapters. This chapter also discussed several important facts about the priority-driven approach. We need to keep them in mind at all times.

An algorithm for scheduling hard real-time jobs is optimal if it can produce a feasible schedule as long as feasible schedules of the given jobs exist, that is, when the system is not overloaded. This is the criterion of merit we use most of the time in this book. The EDF (Earliest-Deadline-First) algorithm is optimal for scheduling preemptable jobs on one processor. LST (Least-Slack-Time) algorithm is also optimal for preemptable jobs on one processor, but it requires information on the execution times of all jobs while the EDF algorithm does not. Neither algorithm is optimal when jobs are nonpreemptable or when there is more than one processor.

Another important concept is predictability of the timing behavior of jobs. The execution behavior of a system is predictable if the system exhibits no anomalies. We can conclude that the jobs in a predictable system can always meet their deadlines if the jobs meet their deadlines according to the maximal schedule of the system, that is, when every job in the system executes for as long as its maximum execution time. We have shown that when the jobs are independent and preemptable and are scheduled on one processor, their execution behavior is predictable. This fact allows us to ignore the variations in job execution times during validation and work with their maximum execution times. Indeed, this is what we will do in most parts of subsequent chapters.

In general, systems that use priority-driven scheduling have scheduling anomalies. In subsequent chapters we will discuss efficient validation algorithms that allow us to verify the timely completions of jobs despite scheduling anomalies. Such an algorithm is correct if it never concludes that some job completes in time when the job may fail to do so. The merits of correct validation algorithms are measured by their efficiency, robustness, and accuracy. These measures tell us how much time a validation algorithm takes to reach its conclusion on the schedulability of each job or the entire system, whether its conclusion remains correct when some assumptions of its underlying model are no longer valid, and whether the algorithm is overly pessimistic.

Finally, the EDF and LST algorithms are not optimal when the system is overloaded so some jobs must be discarded in order to allow other jobs to complete in time. In fact, these algorithms perform poorly for overloaded systems: Their competitiveness factors are equal to zero. Some kind of overload management algorithm should be used with these algorithms.

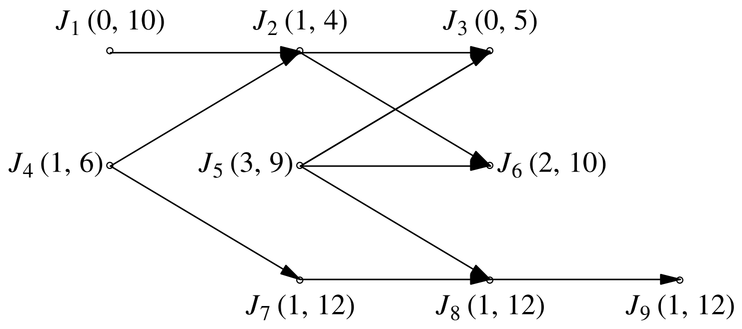
4.11 EXERCISES

4.1 The feasible interval of each job in the precedence graph in Figure 4P–1 is given next to its name. The execution time of all jobs are equal to 1.

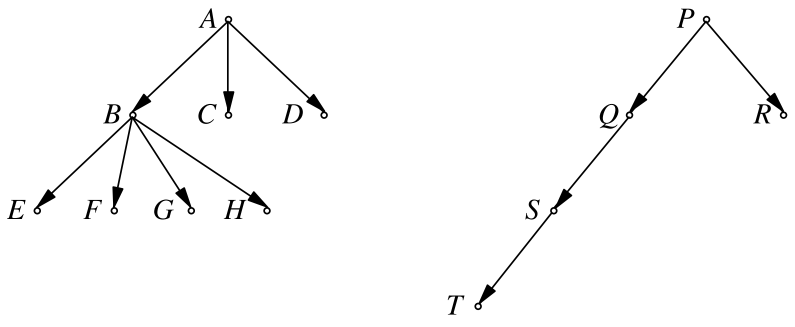
(a) Find the effective release times and deadlines of the jobs in the precedence graph in Figure 4P–1.

(b) Find an EDF schedule of the jobs

(c) A job is said to be at level i if the length of the longest path from the job to jobs that have no successors is i. So, jobs J_3, J_6, and J_9 are at level 0, jobs J_2, J_5, and J_8 are at level 1, and so on. Suppose that the priorities of the jobs are assigned based on their levels: the higher the level, the higher the priority. Find a priority-driven schedule of the jobs in Figure 4P–1 according to this priority assignment.



4.2 **(a)** The execution times of the jobs in the precedence graph in Figure 4P–2 are all equal to 1, and their release times are identical. Give a nonpreemptive optimal schedule that minimizes the completion time of all jobs on three processors. Describe briefly the algorithm you used to find the schedule.



(b) If the execution times of jobs are arbitrary rational numbers, can your algorithm be modified so that it will produce optimal preemptive schedules of such jobs? Explain your answer.

4.3 Prove Theorem 4.3 on the optimality of the LST algorithm for scheduling preemptive jobs on one processor.

4.4 Consider a system that has five periodic tasks, A, B, C, D, and E, and three processors P_1, P_2, P_3. The periods of A, B, and C are 2 and their execution times are equal to 1. The periods of D and E are 8 and their execution times are 6. The phase of every task is 0, that is, the first job of the task is released at time 0. The relative deadline of every task is equal to its period.

(a) Show that if the tasks are scheduled dynamically on three processors according to the LST algorithm, some jobs in the system cannot meet their deadlines.

(b) Find a feasible schedule of the five tasks on three processors.

(c) Parts (a) and (b) allow us to conclude that the LST algorithm is not optimal for scheduling on more than one processor. However, when all the jobs have the same release time or when they have the same deadline, the LST algorithm is optimal. Prove this statement.

4.5 [Grah] A system contains nine nonpreemptable jobs named J_i, for $i = 1, 2, \ldots, 9$. Their execution times are 3, 2, 2, 2, 4, 4, 4, 4, and 9, respectively, their release times are equal to 0, and their deadlines are 12. J_1 is the immediate predecessor of J_9, and J_4 is the immediate predecessor of J_5, J_6, J_7, and J_8. There is no other precedence constraints. For all the jobs, J_i has a higher priority than J_k if $i < k$.

(a) Draw the precedence graph of the jobs.

(b) Can the jobs meet their deadlines if they are scheduled on three processors? Explain your answer.

(c) Can the jobs meet their deadlines if we make them preemptable and schedule them preemptively. Explain your answer.

(d) Can the jobs meet their deadlines if they are scheduled nonpreemptively on four processors? Explain your answer.

(e) Suppose that due to an improvement of the three processors, the execution time of every job is reduced by 1. Can the jobs meet their deadlines? Explain your answer.

4.6 Consider a system that has two processors and uses an on-line preemptive scheduler to schedule jobs on the processors. At time 0, three independent jobs, J_1, J_2, and J_3, with execution time 1, 1, and 2, respectively, are released. Their deadlines are 2, 2, and 4, respectively. The scheduler either schedules a portion of J_3 before time 1, or it does not schedule J_3 before time 1. We now consider these two cases.

(a) In case (1), the scheduler schedules a portion of J_3 before time 1. Now suppose that two more independent jobs, J_4 and J_5, are released at time 2. Their execution times are both equal to 1, and their deadlines are equal to 2. Show that in this case, the scheduler can no longer feasibly schedule all the jobs, while a clairvoyant scheduler, which foresees the releases of J_4 and J_5, would not schedule any part of J_3 before time 1 and therefore could feasibly schedule all the jobs.

(b) In case (2), the on-line scheduler does not schedule any portion of J_3 before time 1. Show by constructing an example that there exist future jobs which the on-line scheduler will not be able to schedule feasibly, while the jobs could feasibly be scheduled if the on-line scheduler had scheduled J_3 before time 1.

4.7 Consider the set of jobs in Figure 4-3. Suppose that the jobs have identical execution time.

(a) What maximum execution time can the jobs have and still can be feasibly scheduled on one processor? Explain your answer.

(b) Suppose that the release times of J_1 and J_2 are jittery. The release time of J_1 can be as early as 0 and as late as 3, and the release time of J_2 can be as late as 1. How can you take into account this variation when you want to determine whether the jobs can all meet their deadlines?