UNIT 2
Clock-Driven Scheduling

5.1 NOTATIONS AND ASSUMPTIONS

- The clock-driven approach to scheduling is applicable only when the system is by and large deterministic. For this reason, we assume a restricted periodic task model. The following are the restrictive assumptions-
- There are *n* periodic tasks in the system. As long as the system stays in an operation mode, *n* is fixed.
- The parameters of all periodic tasks are known a priori. In particular, variations in the interrelease times of jobs in any periodic task are negligibly small.
- In other words, for all practical purposes, each job in *Ti* is released *pi* units of time after the previous job in *Ti* .
- Each job *Ji,k* is ready for execution at its release time *ri,k* .
- We refer to a periodic task *Ti* with phase φi , period *pi* , execution time *ei* , and relative deadline *Di* by the 4-tuple $(\varphi i , pi , ei , Di)$. For example, (1, 10, 3, 6) is a periodic task whose phase is 1, period is 10, execution time is 3, and relative deadline is 6.

- Therefore the first job in this task is released and ready at time 1 and must be completed by time 7; the second job is ready at 11 and must be completed by 17, and so on. Each of these jobs executes for at most 3 units of time.

- The utilization of this task is 0.3. By default, the phase of each task is 0, and its relative deadline is equal to its period. We will omit the elements of the tuple that have their default values. As examples, both (10, 3, 6) and (10, 3) have zero phase. Their relative deadlines are 6 and 10, respectively.

5.2 STATIC, TIMER-DRIVEN SCHEDULER

- The operating system maintains a queue for aperiodic jobs. When an aperiodic job is released, it is placed in the queue without the attention of the scheduler. Whenever the processor is available for aperiodic jobs, the job at the head of this queue executes.

- Whenever the parameters of jobs with hard deadlines are known before the system begins to execute, a straightforward way to ensure that they meet their deadlines is to construct a *static schedule* of the jobs off-line.

- This schedule specifies exactly when each job executes. During run time, the scheduler dispatches the jobs according to this schedule. Hence, as long as no job ever *overruns* all deadlines are surely met.

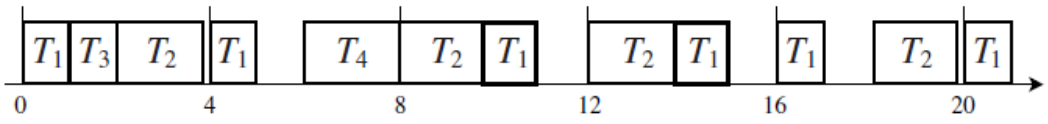

FIGURE 5–1 An arbitrary static schedule.

- As an example, we consider a system that contains four independent periodic tasks. They are $T1 = (4, 1)$, $T2 = (5, 1.8)$, $T3 = (20, 1)$, and $T4 = (20, 2)$. Their utilizations are 0.25, 0.36, 0.05, and 0.1, respectively, and the total utilization is 0.76. Some intervals, such as (3.8, 4), (5, 6), and (10.8, 12), are not used by the periodic tasks. These intervals can be used to execute aperiodic jobs.

- A straightforward way to implement the scheduler is to store the precomputed schedule as a table. Each entry *(tk , T (tk))* in this table gives a *decision time tk* , which is an instant when a scheduling decision is made, and *T (tk)*, which is either the name of the task whose job should start at *tk* or *I* . The latter indicates an idle interval during which no periodic task is scheduled.

- Upon receiving a timer interrupt at *tk* , the scheduler sets the timer to expire at *tk*+1 and prepares the task *T (tk)* for execution. It then suspends itself, letting the task have the processor and execute.When the timer expires again,the scheduler repeats this operation.

- We call a periodic static schedule a *cyclic schedule*. Again, this approach to scheduling hard real-time jobs is called the *clock-driven* or *time-driven* approach because each scheduling decision is made at a specific time, independent of events.

```
Input: Stored schedule (t_k, T(t_k)) for k = 0, 1, ... N − 1.
Task SCHEDULER:
  set the next decision point i and table entry k to 0;
  set the timer to expire at t_k.
  do forever:
    accept timer interrupt;
    if an aperiodic job is executing, preempt the job;
    current task T = T(t_k);
    increment i by 1;
    compute the next table entry k = i mod(N);
    set the timer to expire at ⌊i/N⌋H + t_k;
    if the current task T is I,
      let the job at the head of the aperiodic job queue execute;
      else, let the task T execute;
    sleep;
end SCHEDULER
```

FIGURE 5–2 A clock-driven scheduler.

5.3 GENERAL STRUCTURE OF CYCLIC SCHEDULES

5.3.1 Frames and Major Cycles

- Figure 5–3 shows a good structure of cyclic schedules. A restriction imposed by this structure is that scheduling decisions are made periodically, rather than at arbitrary times. The scheduling decision times partition the time line into intervals called *frames*.

- Every frame has length *f* ; *f* is the *frame size*. Because scheduling decisions are made only at the beginning of every frame, there is no preemption within each frame. The phase of each periodic task is a nonnegative integer multiple of the frame size.

5.3.2 Frame Size Constraints

- Ideally, we want the frames to be sufficiently long so that every job can start and complete its execution within a frame. In this way, no job will be preempted. We can meet this objective if we make the frame size *f* larger than the execution time *ei* of every task *Ti* . In other words,

$$f \geq \max_{1 \leq i \leq n} (e_i) \qquad (5.1)$$

- To keep the length of the cyclic schedule as short as possible, the frame size *f* should be chosen so that it divides *H*, the length of the hyperperiod of the system. This condition is met when *f* divides the period *pi* of at least one task *Ti* , that is,

$$\lfloor p_i/f \rfloor - p_i/f = 0 \qquad (5.2)$$

- We let *F* denote this number and call a hyperperiod that begins at the beginning of the *(kF* + 1*)*st frame, for any *k* = 0, 1, . . . , a *major cycle*.

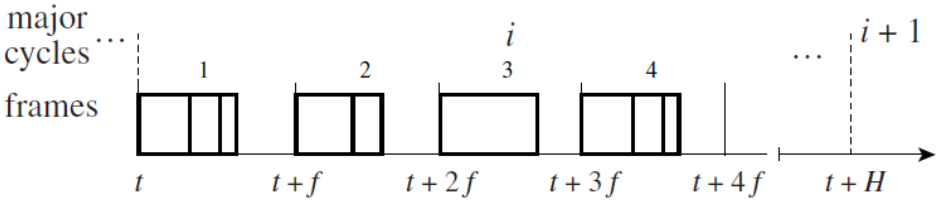

FIGURE 5–3 General structure of a cyclic schedule.

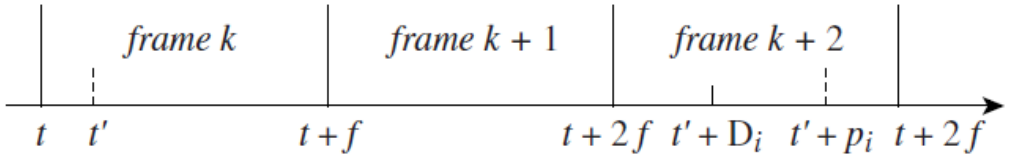

FIGURE 5–4 A constraint on the value of frame size.

- Figure 5–4 illustrates the suitable range of *f* for a task *Ti* = *(pi , ei , Di)*. When *f* is in this range, there is at least one frame between the release time and deadline of every job in the task. In this figure, *t* denotes the beginning of a frame (called the *k*th frame) in which a job in *Ti* is released, and *t* _ denotes the release time of this job.

- We need to consider two cases: *t*_ > *t* and *t* _ = *t*. If *t* _ is later than *t* , as shown in this figure, we want the *(k* +1*)*st frame to be in the interval between the release time *t* _ and the deadline *t* _+ *Di* of this job. For this to be true, we must have *t* +2 *f* equal to or earlier than *t* _ + *Di* , that is, 2 *f* − *(t* _ − *t)* ≤ *Di* .

- Because the difference *t* _ − *t* is at least equal to the greatest common divisor *gcd(pi , f)* of *pi* and *f* , this condition is met if the following inequality holds:

$$2f - gcd(p_i, f) \le D_i \qquad (5.3)$$

We refer to Eqs. (5.1), (5.2) and (5.3) as the *frame-size constraints*.

5.3.3 Job Slices

- Sometimes, the given parameters of some task systems cannot meet all three frame size constraints simultaneously. An example is the system **T** = {*(*4, 1*), (*5, 2, 7*), (*20, 5*)*}. For Eq. (5.1) to be true, we must have *f* ≥ 5, but to satisfy Eq. (5.3) we must have *f* ≤ 4. In this situation, we are forced to partition each job in a task that has a large execution time into slices (i.e., subjobs) with smaller execution times.

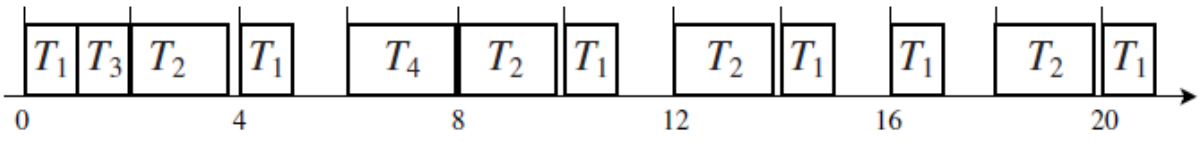

FIGURE 5–5 A cyclic schedule with frame size 2.

- For **T** = $\{(4, 1), (5, 2, 7), (20, 5)\}$, we can divide each job in $(20, 5)$ into a chain of three slices with execution times 1, 3, and 1. In other words, the task $(20, 5)$ now consists of three subtasks $(20, 1)$, $(20, 3)$ and $(20, 1)$.
- The resultant system has five tasks for which we can choose the frame size 4. Figure 5–6 shows a cyclic schedule for these tasks. The three original tasks are called T_1, T_2 and T_3, respectively, and the three subtasks of T_3 are called $T_{3,1}$, $T_{3,2}$, and $T_{3,3}$.

5.4 CYCLIC EXECUTIVES

- We use the term *cyclic executive* in a more general sense to mean a table-driven cyclic scheduler for all types of jobs in a multithreaded system.
- Similar to the scheduler in Figure 1.3, it makes scheduling decisions only at the beginning of each frame and deterministically interleaves the execution of periodic tasks. However, it allows aperiodic and sporadic jobs to use the time not used by periodic tasks.
- The pseudocode in Figure 5–7 describes such a cyclic executive on a CPU. The stored table that gives the precomputed cyclic schedule has F entries, where F is the number of frames per major cycle.
- Each entry (say the kth) lists the names of the job slices that are scheduled to execute in frame k. In Figure 5–7, the entry is denoted by $L(k)$ and is called a *scheduling block*, or simply a block. The current block refers to the list of periodic job slices that are scheduled in the current frame.

```
Input: Stored schedule: L(k) for k = 0, 1, ..., F − 1;
   Aperiodic job queue
Task CYCLIC_EXECUTIVE:
   the current time t = 0;
   the current frame k = 0;
   do forever
      accept clock interrupt at time tf;
      currentBlock = L(k);
      t = t + 1;
      k = t mod F;
      if the last job is not completed, take appropriate action;
      if any of the slices in currentBlock is not released, take appropriate action;
      wake up the periodic task server to execute the slices in currentBlock;
      sleep until the periodic task server completes;
         while the aperiodic job queue is nonempty;
         wake up the job at the head of the aperiodic job queue;
         sleep until the aperiodic job completes;
         remove the aperiodic job from the queue;
      endwhile;
      sleep until the next clock interrupt;
   enddo;
end CYCLIC_EXECUTIVE
```

FIGURE 5–7 A table-driven cyclic executive.

- In addition to scheduling, the cyclic executive also checks for overruns at the beginning of each frame. If the cyclic executive finds the periodic task server still executing the last job at the time of a clock interrupt, a *frame overrun* occurs.
- Some slice(s) scheduled in the previous frame has executed longer than the time allocated to it by the precomputed cyclic schedule. The cyclic executive takes an appropriate action to recover from this frame overrun.

5.5 IMPROVING THE AVERAGE RESPONSE TIME OF APERIODIC JOBS

- There is no advantage to completing a job with a hard deadline early. On the other hand, an aperiodic job is released and executed by the system in response to an event.
- The sooner an aperiodic job completes, the more responsive the system is. For this reason, we try to minimize the response time of each aperiodic job.

5.5.1 Slack Stealing

- A natural way to improve the response times of aperiodic jobs is by executing the aperiodic jobs ahead of the periodic jobs whenever possible. This approach, called *slack stealing*, was originally proposed for priority-driven systems. Let the total amount of time allocated to all the slices scheduled in the frame k be xk.

- The *slack* (time) available in the frame is equal to $f - xk$ at the beginning of the frame. When an aperiodic job executes ahead of slices of periodic tasks, it consumes the slack in the frame. After y units of slack time are used by aperiodic jobs, the available slack is reduced to $f - xk - y$.

- The cyclic executive can let aperiodic jobs execute in frame k as long as there is slack, that is, the available slack $f - xk - y$ in the frame is larger than 0.

- Figure 5–8 gives an illustrative example. Figure 5–8(a) shows the first major cycle in the cyclic schedule of the periodic tasks.

- Figure 5–8(b) shows three aperiodic jobs $A1$, $A2$, and $A3$. Their release times are immediately before 4, 9.5. and 10.5, and their execution times are 1.5, 0.5 and 2, respectively.

- Figure 5–8(c) shows when the aperiodic jobs execute if we use the cyclic executive shown in Figure 5–7, which schedules aperiodic jobs after the slices of periodic tasks in each frame are completed. The execution of $A1$ starts at time 7. It does not complete at time 8 when the frame ends and is, therefore, preempted. It is resumed at time 10 after both slices in the next frame complete. Consequently, its response time is 6.5.

- $A2$ executes after $A1$ completes and has a response time equal to 1.5. Similarly, $A3$ follows $A2$ and is preempted once and completes at the end of the following frame. The response time of $A3$ is 5.5. The average response time of these three jobs is 4.5.

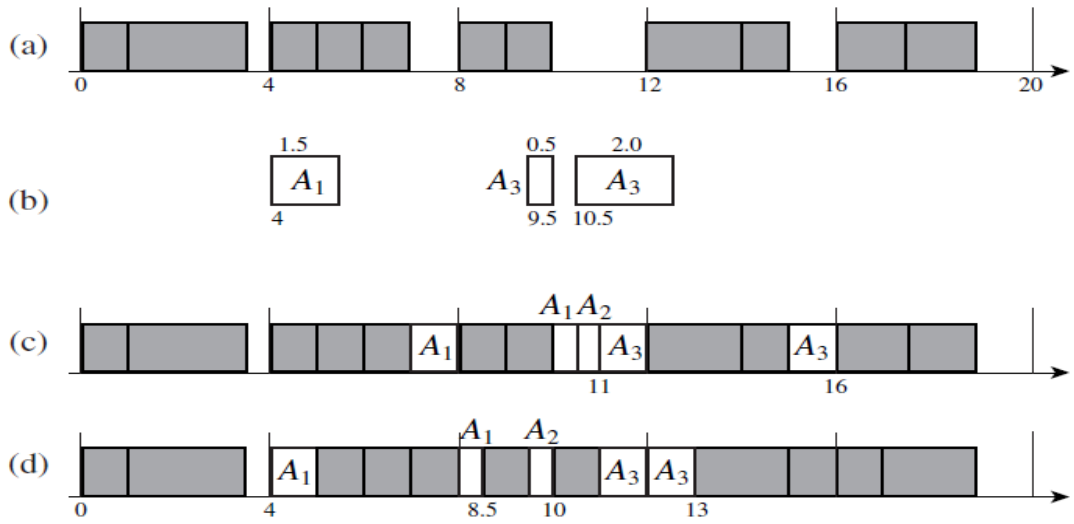

FIGURE 5–8 Example illustrating slack stealing.

- Figure 5–8(d) shows what happens if the cyclic executive does slack stealing. At time 4, the cyclic executive finds $A1$ in the aperiodic job queue, and there is 1 unit of slack. Hence it lets $A1$ execute. At time 5, there is no more slack. It preempts $A1$ and lets the periodic task server execute the job slices scheduled in the frame.

At the beginning of the next frame, the available slack is 2. It resumes $A1$, which completes at time 8.5. At the time, the first slice in the current block is executed, since the aperiodic job queue is empty.

Upon completion of the slice at time 9.5, the cyclic executive checks the aperiodic job queue again, finds *A*2 ready, and lets *A*2 execute.

When the job completes at time 10, the cyclic executive finds the aperiodic job queue empty and lets the periodic task server execute the next job slice in the current block. At time 11, it finds *A*3 and lets the job execute during the last unit of time in the frame, as well as in the beginning of the next frame. The job completes by time 13. According to this schedule, the response times of the jobs are 4.5, 0.5, and 2.5, with an average of 2.5.

5.5.2 Average Response Time

- We are often required to guarantee that the average response times of aperiodic jobs is no greater than some value. To give this guarantee, we need to be able estimate the average response time of these jobs.

- To express average response time in terms of these parameters, let us consider a system in which there are n_a aperiodic tasks. The jobs in each aperiodic task have the same interarrival-time and execution time distributions and the same response-time requirement. Suppose that the average rate of arrival of aperiodic jobs in the *i*th aperiodic task is λi jobs per unit of time.

- The sum λ of λi over all $i = 1, 2, \ldots a$ is the total number of aperiodic job arrivals per unit of time. The mean and the mean square values of the execution times of jobs in the *i*th aperiodic task are $E[\beta i]$ and $E[\beta i^2]$, respectively.

- Let u_i denote the average utilization of the *i*th task; it is the average fraction of processor time required by all the jobs in the task. *ui* is equal to $\lambda i\, E[\beta i]$. We call the sum U_A of u_i over all aperiodic tasks the *total average utilization of aperiodic tasks*; it is the average fraction of processor time required by all the aperiodic tasks in the system.

- Let U be the total utilization of all the periodic tasks. $1-U$ is the fraction of time that is available for the execution of aperiodic jobs. We call it the *aperiodic* (*processor*) *bandwidth*.

- If the total average utilization of the aperiodic jobs is larger than or equal to the aperiodic bandwidth of the system (i.e., $U_A \geq 1 - U$), the length of the aperiodic job queue and the average response time will grow without bound. Hence we consider here only the case where $U_A < 1 - U$.

- When the jobs in all aperiodic tasks are scheduled on the FIFO basis, we can estimate the average response time W (also known as waiting time) of any aperiodic job by the following expression:

$$W = \sum_{i=1}^{n_a} \frac{\lambda_i E[\beta_i]}{\lambda(1-U)} + \frac{W_0}{(1-U)^2[1 - U_A/(1-U)]} \qquad (5.4a)$$

where W_0 is given by

$$W_0 = \sum_{i=1}^{n_a} \frac{\lambda_i E[\beta_i{}^2]}{2} \qquad (5.4b)$$

- Figure 5–9 shows the behavior of the average queueing time. The average queueing time is inversely proportional to the square of the aperiodic bandwidth. It remains small for a large range of *UA* but increases rapidly and approaches infinity when *UA* approaches $(1 - U)$.

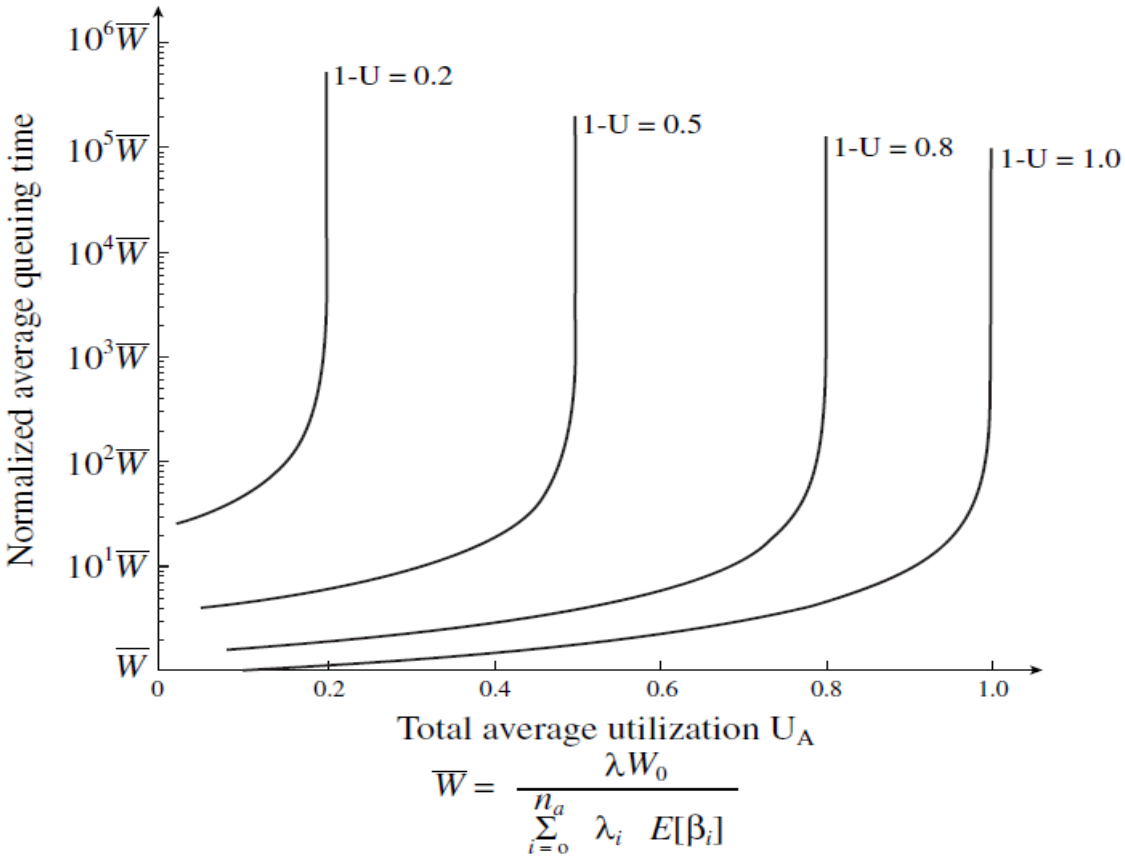

FIGURE 5–9 Average queueing time versus total average utilization.

- We can improve the average response time of some aperiodic tasks at the expense of some others by prioritizing the aperiodic tasks. The jobs in the queue for the *i*th aperiodic task are executed only when the queues of all the higher priority aperiodic tasks are empty.

- We let U_H denote the sum_$\sum_{k=1 \text{to } i-1}$ *uk* ; it is the average total utilization of all aperiodic tasks with priority higher than the *i*th aperiodic task. The average response time *Wi* of a job in the *i*th aperiodic task is given by

$$W_i = \frac{E[\beta_i]}{(1-U)} + \frac{W_0}{(1-U)^2[1-U_H/(1-U)][1-(U_H+u_i)/(1-U)]} \qquad (5.5)$$

approximately, where W_0 is given by Eq. (5.4b).

5.6 SCHEDULING SPORADIC JOBS

- Like jobs in periodic tasks, sporadic jobs have hard deadlines. Their minimum release times and maximum execution times are unknown a priori. Consequently, it is impossible to guarantee a priori that all sporadic jobs can complete in time.

5.6.1 Acceptance Test

- A common way to deal with this situation is to have the scheduler perform an acceptance test when each sporadic job is released. During an *acceptance test*, the scheduler checks whether the newly released sporadic job can be feasibly scheduled with all the jobs in the system at thetime. Here, by *a job in the system*, we mean either a periodic job or a sporadic job.

- If according to the existing schedule, there is a sufficient amount of time in the frames before its deadline to complete the newly released sporadic job without causing any job in the system to complete too late, the scheduler accepts and schedules the job. Otherwise, the scheduler rejects the new sporadic job.

- Conceptually, it is quite simple to do an acceptance test. To explain, let us suppose that at the beginning of frame t , an acceptance test is done on a sporadic job $S(d, e)$, with deadline d and (maximum) execution time e.

- Suppose that the deadline d of S is in frame l+1 and $l \geq t$. Clearly, the job must be scheduled in the lth or earlier frames. The job can complete in time only if the *current* (*total*) *amount of slack time* $\sigma c(t, l)$ in frames t, t+1, . . . l is equal to or greater than its execution time e. Therefore, the scheduler should reject S if $e > \sigma c(t, l)$.

- As we will see shortly, the scheduler may let a new sporadic job execute ahead of some previously accepted sporadic jobs. Therefore, the scheduler also checks whether accepting the new job may cause some sporadic jobs in the system to complete late.

- The scheduler accepts the new job $S(d, e)$ only if $e \leq \sigma c(t, l)$ and no sporadic jobs in system are adversely affected.

5.6.2 EDF Scheduling of the Accepted Jobs

- By virtue of its optimality, the EDF algorithm is a good way to schedule accepted sporadic jobs.The scheduler maintains a queue of accepted sporadic jobs.

- Whenever all the slices of periodic tasks scheduled in each frame are completed, the cyclic executive lets the jobs in the sporadic job queue execute in the order they appear.

- Figure 5–10 gives a pseudocode description of the modified cyclic executive that integrates the scheduling of sporadic jobs with that of aperiodic and periodic jobs.

- Figure 5–11 gives an example. The frame size used here is 4. The shaded boxes show where periodic tasks are scheduled.

```
Input: Stored schedule: L(k) for k = 0, 1, ... F − 1;
   Aperiodic job queue, sporadic-job waiting queue, and accepted-sporadic-job EDF queue;
Task CYCLIC_EXECUTIVE:
   the current time t = 0;
   the current frame k = 0;
   do forever
      accept clock interrupt at time tf;
      currentBlock = L(k);
      t = t + 1;
      k = t mod F;
      if the last job is not completed, take appropriate action;
      if any of the slices in the currentBlock is not released, take appropriate action;
      while the sporadic-job waiting queue is not empty,
         remove the job at the head of the sporadic job waiting queue;
         do an acceptance test on the job;
         if the job is acceptable,
            insert the job into the accepted-sporadic-job queue in the EDF order;
            else, delete the job and inform the application;
      endwhile;
      wake up the periodic task server to execute the slices in currentBlock;
      sleep until the periodic task server completes;
      while the accepted sporadic job queue is nonempty,
         wake up the job at the head of the sporadic job queue;
         sleep until the sporadic job completes;
         remove the sporadic job from the queue;
         endwhile;
         while the aperiodic job queue is nonempty,
         wake up the job at the head of the aperiodic job queue;
         sleep until the aperiodic job completes;
         remove the aperiodic job from the queue;
      endwhile;
      sleep until the next clock interrupt;
   enddo;
end CYCLIC_EXECUTIVE
```

FIGURE 5–10 A cyclic executive with sporadic and aperiodic job scheduling capability.

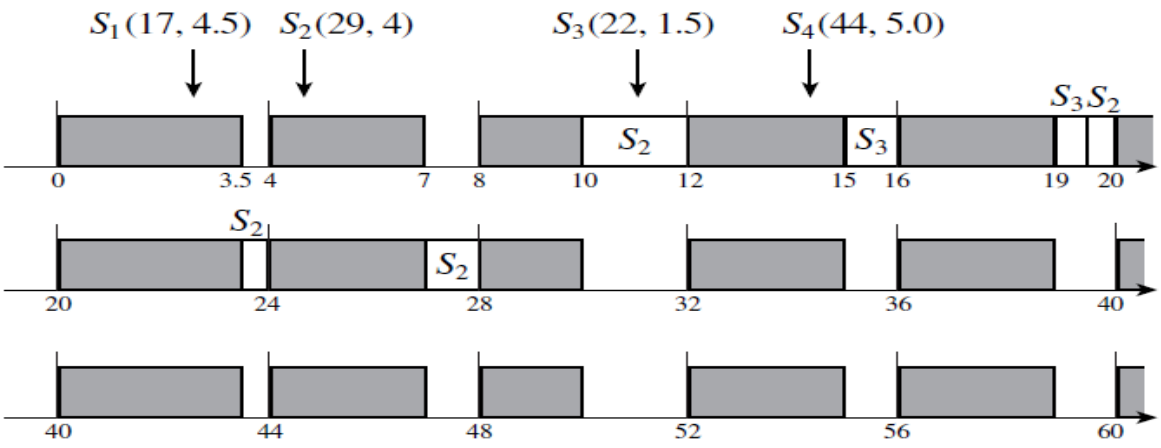

FIGURE 5–11 Example of scheduling sporadic jobs.

- Suppose that at time 3, a sporadic job *S*1*(*17, 4.5*)* with execution time 4.5 and deadline 17 is released. The acceptance test on this job is done at time 4, that is, the beginning of frame 2. *S*1 must be scheduled in frames 2, 3, and 4. In these frames, the total amount of slack time is only 4, which is smaller than the execution time of *S*1. Consequently, the scheduler rejects the job.

- At time 5, *S*2*(*29, 4*)* is released. Frames 3 through 7 end before its deadline. During the acceptance test at 8, the scheduler finds that the total amount of slack in these frames is 5.5. Hence, it accepts *S*2. The first part of *S*2 with execution time 2 executes in the current frame.

- At time 11, *S*3*(*22, 1.5*)* is released. At time 12, the scheduler finds 2 units of slack time in frames 4 and 5, where *S*3 can be scheduled. Moreover, there still is enough slack to complete *S*2 even though *S*3 executes ahead of *S*2. Consequently, the scheduler accepts *S*3. This job executes in frame 4.

- Suppose that at time 14, *S*4*(*44, 5*)* is released. At time 16 when the acceptance test is done, the scheduler finds only 4.5 units of time available in frames before the deadline of *S*4, after it has accounted for the slack time that has already been committed to the remaining portions of *S*2 and *S*3. Therefore, it rejects *S*4. When the remaining portion of *S*3 completes in the current frame, *S*2 executes until the beginning of the next frame.

- The last portion of *S*2 executes in frames 6 and 7.

5.6.3 Implementation of the Acceptance Test

The acceptance test consists of the following two steps:

1. The scheduler first determines whether the current total amount of slack in the frames before the deadline of job *S* is at least equal to the execution time *e* of *S*. If the answer is no, it rejects *S*. If the answer is yes, the second step is carried out.
2. In the second step, the scheduler determines whether any sporadic job in the system will complete late if it accepts *S*. If the acceptance of *S* will not cause any sporadic job in the system to complete too late, it accepts *S*; otherwise, it rejects *S*.

- To do an acceptance test, the scheduler needs the current total amount of slack time *σc(i, k)* in frames *i* through *k* for every pair of frames *i* and *k*. We can save computation time during run time by precomputing the *initial (total) amounts of slack σ(i, k),* for *i, k* =1, 2, . . . *F* and storing them in a slack table.

  - For any 0 < *j* < *j* _ and any *i* and *k* equal to 1, 2, . . . *F*, the initial amount of slack time in frames from frame *i* in major cycle *j* through frame *k* in major cycle *j*_ is given by

$$\sigma(i + (j-1)F, k + (j'-1)F) = \sigma(i, F) + \sigma(1, k) + (j - j' - 1)\sigma(1, F)$$

- Suppose that at the beginning of current frame *t* , there are *ns* sporadic jobs in the system. We call these jobs *S*1, *S*2, . . . , *Sns*. Let *dk* and *ek* denote the deadline and execution time of *Sk*,respectively, and *ξk* denote the execution time of the portion of *Sk* that has been completed at the beginning of the current frame.

- Suppose that the deadline *d* of the job *S(d, e)* being tested is in frame *l* + 1. The current total amount of slack time *σc(t, l)* in frames *t* through *l* can be computed from the initial amount of slack time in these frames according to

$$\sigma_c(t,l) = \sigma(t,l) - \sum_{d_k \le d} (e_k - \xi_k) \qquad (5.6a)$$

- The sum in this expression is over all sporadic jobs in the system that have deadlines equal to or less than *d*. Because the slack time available in these frames must be used to execute the remaining parts of these jobs first, only the amount leftover by them is available to *S(d, e)*.

- If *S* is accepted, the amount of slack σ before its deadline is equal to

$$\sigma = \sigma_c(t,l) - e \qquad (5.6b)$$

5.6.4 Optimality of Cyclic EDF Algorithm

- We have claimed that the cyclic EDF algorithm is good. Now is the time to ask just how good it is. To answer this question, we first compare it only with the class of algorithms that perform acceptance tests at the beginnings of frames.

- The cyclic EDF algorithm is optimal in the following sense: As long as the given string of sporadic jobs is schedulable by any algorithm in this class, the EDF algorithm can always find a feasible schedule of the jobs.

- However, the cyclic EDF algorithm is not optimal when compared with algorithms that perform acceptance tests at arbitrary times. If we choose to use an interrupt-driven scheduler which does an acceptance test upon the release of each sporadic job, we should be able to do better.

- The advantage of the interrupt-driven alternative is outweighed by a serious shortcoming: It increases the danger for periodic-job slices to complete late. The cyclic EDF algorithm for scheduling sporadic jobs is an on-line algorithm.

- At any scheduling decision time, without prior knowledge on when the future jobs will be released and what their parameters will be, it is not always possible for the scheduler to make an optimal decision.

- Therefore, it is not surprising that the cyclic EDF algorithm is not optimal in this sense when some job in the given string of sporadic jobs must be rejected.

5.7 PRACTICAL CONSIDERATIONS AND GENERALIZATIONS

So far, we have ignored many practical problems, such as how to handle frame overruns, how to do mode changes, and how to schedule tasks on multiprocessor systems.

5.7.1 Handling Frame Overruns

- A frame overrun can occur for many reasons. For example, when the execution time of a job is input data dependent, it can become unexpectedly large for some rare combination of input values which is not taken into account in the precomputed schedule.

- A transient hardware fault in the system may cause some job to execute longer than expected. A software flaw that was undetected during debugging and testing can also cause this problem.

- There are many ways to handle a frame overrun. Which one is the most appropriate depends on the application and the reason for the overrun.

- A way to handle overruns is to simply abort the overrun job at the beginning of the next frame and log the premature termination of the job. Such a fault can then be handled by some

recovery mechanism later when necessary. This way seems attractive for applications where late results are no longer useful.

- An example is the control-law computation of a robust digital controller. When the computation completes late or terminates prematurely, the result it produces is erroneous. On the other hand, as long as such errors occur infrequently, the erroneous trajectory of the controlled system remains sufficiently close to its correct trajectory.

- Another way to handle an overrun is to continue to execute the offending job. The start of the next frame and the execution of jobs scheduled in the next frame are then delayed.

- Letting a late job postpone the execution and completion of jobs scheduled after it can in turn cause these jobs to be late. This way is appropriate only if the late result produced by the job is nevertheless useful, and an occasional late completion of a periodic job is acceptable.

5.7.2 Mode Changes

- The number *n* of periodic tasks in the system and their parameters remain constant as long as the system stays in the same (operation) mode. During a *mode change*, the system is reconfigured.

- Some periodic tasks are deleted from the system because they will not execute in the new mode. When the mode change completes, the new set of periodic tasks are scheduled and executed.

Aperiodic Mode Change:

- A reasonable way to schedule a mode-change job that has a soft deadline is to treat it just like an ordinary aperiodic job. Once the job begins to execute, however, it may modify the old schedule in order to speed up the mode change.

- A periodic task that will not execute in the new mode can be deleted and its memory space and processor time are freed as soon as the current job in the task completes.

- During mode change, the scheduler continues to use the old schedule table. Before the periodic task server begins to execute a periodic job, however, it checks whether the corresponding task is marked and returns immediately if the task is marked.

- In this way, the schedule of the periodic tasks that execute in both modes remain unchanged during mode change, but the time allocated to the deleted task can be used to execute the mode-change job.

- Once the new schedule table and code of the new tasks are in memory, the scheduler can switch to use the new table.

Sporadic Mode Change:

- A sporadic mode change has to be completed by a hard deadline. There are two possible approaches to scheduling this job. We can treat it like an ordinary sporadic job and schedule it.

- This approach can be used only if the application can handle the rejection of the job. Specifically, if the mode-change job is not schedulable and is therefore rejected, the application can either postpone the mode change or take some alternate action.

- In this case, we can use mode changer described in Figure 5–12.

```
task MODE_CHANGER (oldMode, newMode):
   fetch the deleteList of periodic tasks to be deleted;
   mark each periodic task in the deleteList;
   inform the cyclic executive that a mode change has commenced;
   fetch the newTaskList of periodic tasks to be executed in newMode;
   allocate memory space for each task in newTaskList and create each of these task;
   fetch the newSchedule;
   perform acceptance test on each sporadic job in the system according to the newSchedule,
   if every sporadic job in system can complete on time according to the newSchedule,
      inform the cyclic executive to use the newSchedule;
      else,
         compute the latestCompletionTime of all sporadic jobs in system;
         inform the cyclic executive to use the newSchedule at
            max (latestCompletionTime, thresholdTime);
End Mode_Changer
```

FIGURE 5–12 A mode changer.

- As an example, when a computer-controlled bulldozer moves to the pile of hazardous waste, the mode change to slow down its forward movement and carry out the digging action should complete in time; otherwise the bulldozer will crash into the pile.

- If this mode change cannot be made in time, an acceptable alternative action is for the bulldozer to stop completely. The time required by the controller to generate the command for the stop action is usually much shorter, and the sporadic job to stop the bulldozer is more likely to be acceptable.

- On the other hand, the action to stop the bulldozer cannot be postponed. The scheduler must admit and schedule the job to activate the brakes in time. In general, the mode changer in Figure 5–12 cannot be used when a sporadic mode change cannot be rejected.

- The only alternative is to schedule each sporadic mode-change job that cannot be rejected periodically, with a period no greater than half the maximum allowed response time.

5.7.3 General Workloads and Multiprocessor Scheduling

- The clock-driven approach is applicable to workloads that are not characterizable by the periodic task model. As long as the parameters of the jobs are known a priori, a static schedule can be computed off-line.

- Once a feasible schedule is found and stored as a table, the static scheduler described in Figure 5–2 can use the table in the same manner as schedule tables of periodic tasks.

- It is conceptually straightforward to schedule tasks on several processors whenever the workload parameters are known a priori and there is a global clock. As long as the clock drifts on the processors are sufficiently small, we can use the uniprocessor schedulers.

- Sometimes, a precomputed multiprocessor schedule can be found straightforwardly from a precomputed uniprocessor schedule.

- As an example, Figure 5–13(a) shows a system containing several CPUs connected by a system bus. Each task consists of a chain of jobs, which executes on one of the CPUs and sends or receives data from one of the I/O devices via the system bus. In a system with such an architecture, the system bus is sometimes the bottleneck.

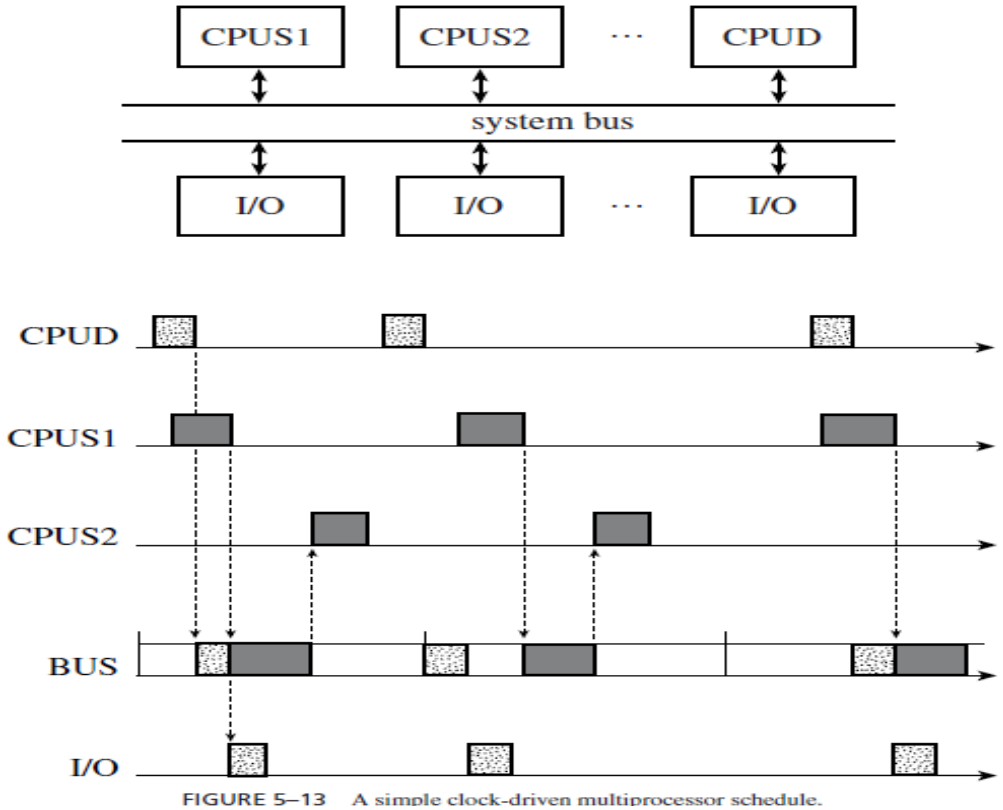

FIGURE 5–13 A simple clock-driven multiprocessor schedule.

- By the bus being the bottleneck, we mean that if there is a feasible schedule of all the data transfer activities on the bus, it is always possible to feasibly schedule the jobs that send and receive the data on the respective CPUs.

- To illustrate, Figure 5–13(b) shows a cyclic schedule of the data-transfer activities on the bus and the schedules of CPUs and I/O device interfaces that produce and consume the data.

- The shaded boxes on the time lines of CPUS1 and CPUS2 show when the CPUs execute in order to produce and consume the data that occupy the bus in time intervals shown by shaded boxes on the time line of the bus.

- Similarly, the CPUD is the producer of the data transferred to an I/O device during intervals shown as dotted boxed on the bus time line.

- We can see that the schedules of the CPUs and I/O devices can be derived directly from the schedule of the bus. Computing the schedule for the entire system is simplified to computing the schedule of the system bus.

- This example is based on the Boeing 777 Airplane Information Management System (AIMS). The system uses a table-driven system bus protocol. The protocol controls the timing of all data transfers. The intervals when the bus interface unit of each CPU must execute are determined by the schedule of the system bus in a manner illustrated by this example.

5.8 ALGORITHM FOR CONSTRUCTING STATIC SCHEDULES

The general problem of choosing a minor frame length for a given set of periodic tasks, segmenting the tasks if necessary, and scheduling the tasks so that they meet all their deadlines is NP-hard. Here, we first consider the special case where the periodic tasks contain no nonpreemptable Section and then consider the cases of nonpreemptivity.

5.8.1 Scheduling Independent Preemptable Tasks

- The iterative algorithm described below enables us to find a feasible cyclic schedule if one exists. The algorithm is called the *iterative network-flow algorithm*, or the *INF algorithm* for short. Its key assumptions are that tasks can be preempted at any time and are independent.

- Before applying the INF algorithm on the given system of periodic tasks, we find all the possible frame sizes of the system: These frame sizes met the constraints of Eqs. (5.2) and (5.3) but not necessarily satisfy Eq. (5.1).

- The INF algorithm iteratively tries to find a feasible cyclic schedule of the system for a possible frame size at a time, starting from the largest possible frame size in order of decreasing frame size. A feasible schedule thus found tells us how to decompose some tasks into subtasks if their decomposition is necessary.

- If the algorithm fails to find a feasible schedule after all the possible frame sizes have been tried, the given tasks do not have a feasible cyclic schedule that satisfies the frame size constraints even when tasks can be decomposed into subtasks.

Network-Flow Graph:

The algorithm used during each iteration is based on the well-known network-flow formulation of the preemptive scheduling problem. In the description of this formulation, it is more convenient to ignore the tasks to which the jobs belong and name the jobs to be scheduled in a major cycle of F frames $J1, J2, \ldots, JN$.

- The constraints on when the jobs can be scheduled are represented by the *network-flow graph* of the system. This graph contains the following vertices and edges; the capacity of an edge is a nonnegative number associated with the edge.

1. There is a *job vertex Ji* representing each job *Ji*, for $i = 1, 2, \ldots, N$.

2. There is a *frame vertex* named *j* representing each frame *j* in the major cycle, for $j = 1, 2, \ldots, F$.

3. There are two special vertices named *source* and *sink*.

4. There is a directed edge (Ji, j) from a job vertex *Ji* to a frame vertex *j* if the job *Ji* can be scheduled in the frame *j* , and the *capacity* of the edge is the frame size f .

5. There is a directed edge from the *source* vertex to every job vertex *Ji* , and the capacity of this edge is the execution time *ei* of the job.

6. There is a directed edge from every frame vertex to the *sink*, and the capacity of this edge is f .

- A *flow of an edge* is a nonnegative number that satisfies the following constraints:
  - It is no greater than the capacity of the edge
  - with the exception of the *source* and *sink*,
- The sum of the flows of all the edges into every vertex is equal to the sum of the flows of all the edges out of the vertex. **Figure 5–14** shows part of a network-flow graph.
- For simplicity, only job vertices *Ji* and *Jk* are shown. The label "(capacity), flow" of the each edge gives its capacity and flow.
- This graph indicates that job *Ji* can be scheduled in frames *x* and *y* and the job *Jk* can be scheduled in frames *y* and *z*.
- A *flow of a network-flow graph*, or simply a flow, is the sum of the flows of all the edges from the *source*; it should equal to the sum of the flows of all the edges into the *sink*.

There are many algorithms for finding the maximum flows of network-flow graphs. The time complexity of straightforward ones is $O((N + F)3)$. Its worst-case run time complexity is pseudopolynomial in the length of the major cycle F, but on average, it can find a maximum flow in a much shorter time.

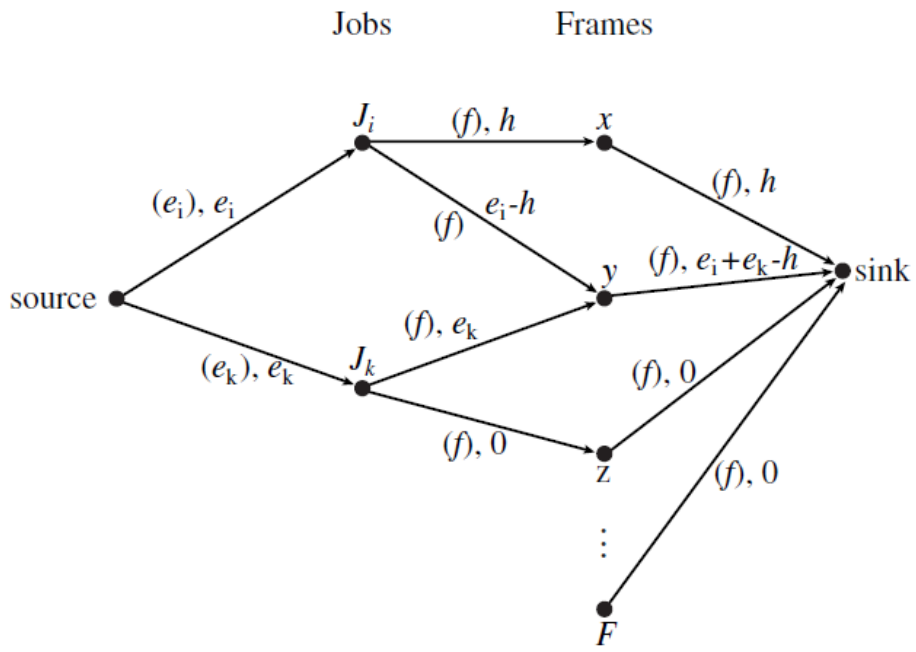

FIGURE 5–14 Part of a network-flow graph.

Maximum Flow and Feasible Preemptive Schedule:

- The maximum flow of a network-flow graph defined above is at most equal to the sum of the execution times of all the jobs to be scheduled in a major cycle. The set of flows of edges from job vertices to frame vertices that gives this maximum flow represents a feasible preemptive schedule of the jobs in the frames.

- Specifically, *the flow of an edge (Ji , j) from a job vertex Ji to a frame vertex j gives the amount of time in frame j allocated to job Ji* .

- The total amount of time allocated to a job *Ji* is represented by the total flow out of all the edges from the job vertex *Ji* . Since this amount is equal to the flow into *Ji* , this amount is *ei* .

- Since the flow of the only edge out of every frame vertex is at most *f* , the total amount of time in every frame allocated to all jobs is at most *f* .

As an example, we again look at the tasks $T_1 = (4, 1)$, $T_2 = (5, 2, 7)$, and $T_3 = (20, 5)$. The possible frame sizes are 4 and 2. The network-flow graph used in the first iteration is shown in Figure 5–15. The frame size is 4. The maximum flow of this graph is 18, which is the total execution time of all jobs in a hyperperiod. Hence the flows of edges from the job vertices to the frame vertices represent a schedule of the tasks, in particular, the schedule shown in Figure 5–7.

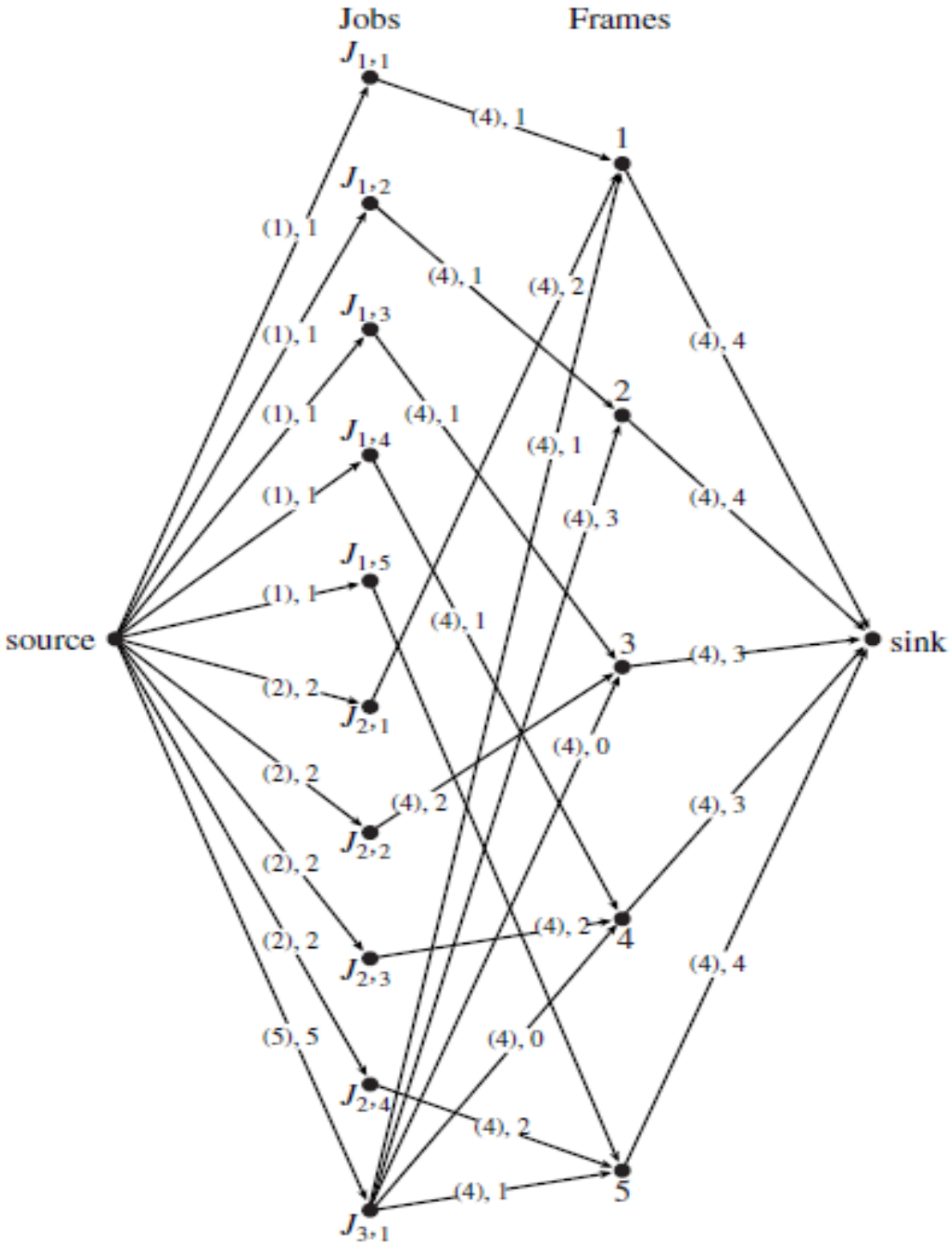

FIGURE 5–15 Example illustrating network-flow formulation.

Generalization to Arbitrary Release Times and Deadlines:

The network-flow formulation of the preemptive scheduling problem can easily be generalized to find schedules of independent jobs that have arbitrary known release times and deadlines. In this case, the release times and deadlines of all N jobs to be scheduled partition the time from the earliest release time to the latest deadline into at most $2N - 1$ intervals. The capacity of each edge into or out of an interval vertex is equal to the

length of the interval represented by the vertex. a set of flows that gives the maximum flow equal to the total execution times of all the jobs represents a preemptive schedule of the job.

5.8.2 Postprocessing

The feasible schedule found by the INF algorithm may not meet some of the constraints that are imposed on the cyclic schedule. Examples are precedence order of jobs and restrictions on preemptions. The networkflow graph for each possible frame size is generated based on the assumption that the jobs are independent.

Hence, the jobs may be scheduled in some wrong order according to a feasible schedule found by the INF algorithm. We can always transform the schedule into one that meets the precedence constraints of the jobs by swapping the time allocations of jobs that are scheduled in the wrong order.

5.9 PROS AND CONS OF CLOCK-DRIVEN SCHEDULING

Advantages:

- The clock-driven approach to scheduling has many advantages. The most important one is its conceptual simplicity.
- We can take into account complex dependencies, communication delays, and resource contentions among jobs in the choice and construction of the static schedule, making sure there will be no deadlock and unpredictable delays.
- When the workload is mostly periodic and the schedule is cyclic, timing constraints can be checked and enforced at each frame boundary.
- Context switching and communication overhead can be kept low by choosing as large a frame size as possible.
- Systems based on the clock-driven scheduling paradigm are typically *time-triggered. In a time-triggered system,* interrupts in response to external events are queued and polled periodically.
- Systems based on the clock-driven scheduling paradigm are typically *time-triggered. In a time-triggered system,* interrupts in response to external events are queued and polled periodically.

Disadvantages:

- Clock-driven approach also has many disadvantages.
- The most obvious one is that a system based on this approach is brittle: It is relatively difficult to modify and maintain.
- The release times of all jobs must be fixed. In contrast, priority-driven algorithms do not require fixed release times.
- In a clock-driven system, all combinations of periodic tasks that might execute at the same time must be known a priori so a schedule for the combination can be precomputed.
- The pure clock-driven approach is not suitable for many systems that contain both hard and soft real-time applications.