UNIT-2

Clock-Driven Scheduling

NOTATIONS AND ASSUMPTIONS:

The clock-driven approach to scheduling is applicable only when the system is by and large deterministic, except for a few aperiodic and sporadic jobs to be accommodated in the deterministic framework. For this reason, we assume a restricted periodic task model throughout this chapter. The following are the restrictive assumptions that we will remove in subsequent chapters:

1. There are *n* periodic tasks in the system. As long as the system stays in an operation mode, *n* is fixed.

2. 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 T_i is released p_i units of time after the previous job in T_i .

3. Each job $J_{i,k}$ is ready for execution at its release time $r_{i,k}$.

STATIC, TIMER-DRIVEN SCHEDULER:

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. According to the schedule, the amount of processor time allocated to every job is equal to its maximum execution time, and every job completes by its deadline. During run time, the scheduler dispatches the jobs according to this schedule. Hence, as long as no job ever *overruns* (i.e., some rare or erroneous condition causes it to execute longer than its maximum execution time), all deadlines are surely met. Because the schedule is computed off-line, we can afford to use complex, sophisticated algorithms. Among all the feasible schedules (i.e., schedules where all jobs meet their deadlines), we may want to choose one that is good according to some criteria (e.g., the processor idles nearly periodically to accommodate aperiodic jobs).

As an example, we consider a system that contains four independent periodic tasks. They are $T_1 = (4, 1)$, $T_2 = (5, 1.8)$, $T_3 = (20, 1)$, and $T_4 = (20, 2)$. Their utilizations are 0.25, 0.36, 0.05, and 0.1, respectively, and the total utilization is 0.76. It suffices to construct a static schedule for the first hyperperiod of the tasks. Since the least common multiple of all periods is 20, the length of each hyperperiod is 20. The entire schedule consists of replicated segments of length 20. Below figure shows such a schedule segment on one processor. We see that T_1 starts execution at time 0, 4, 9.8, 13.8, and so on; T_2 starts execution at 2, 8, 12, 18, and so on. All tasks meet their deadlines. 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.

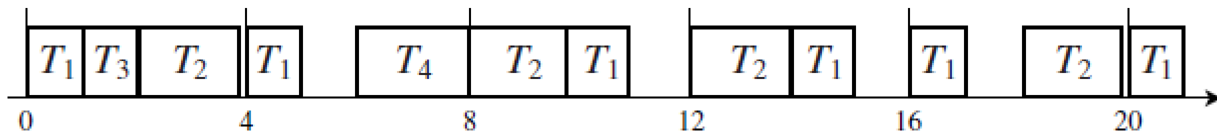


Figure -1

The pseudo code in describes the operation of such a scheduler. H is the length of the hyperperiod of the system. N is the number of entries in the schedule of each hyperperiod. The description assumes the existence of a timer. The timer, once set to expire at a certain time, will generate an interrupt at that time. This interrupt wakes up the scheduler, which is given the processor with a negligible amount of delay.

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, such as job releases and completions, in the system. It is easy to see why a clock-driven system never exhibits the anomalous timing behavior of priority-driven systems.

Pseudo code:

Input: Stored schedule $(tk , T (tk))$ 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 tk .
do forever:
accept timer interrupt;
if an aperiodic job is executing, preempt the job;
current task $T = T (tk)$;
increment i by 1;
compute the next table entry $k = i\ mod(N)$;
set the timer to expire at _i/N_H + tk ;
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

GENERAL STRUCTURE OF CYCLIC SCHEDULES:

Rather than using ad hoc cyclic schedules, such as the one in Figure -1, we may want to use a schedule that has a certain structure. By making sure that the structure has the desired characteristics, we can ensure that the cyclic schedule and the scheduler have these characteristics.

Frames and Major Cycles:

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. In other words, the first job of every task is released at the beginning of some frame.

Frame Size Constraints:

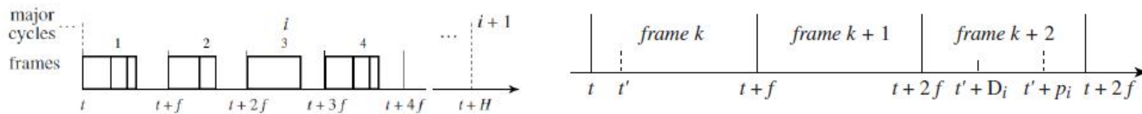
Ideally, we want the frames to be sufficiently long so that every job can start and complete its execution within a frame. In this way, no job will be preempted. We can meet this objective if we make the frame size f larger than the execution time e_i of every task T_i. In other words,

$$f \geq \max_{1 \leq i \leq n} (e_i) \qquad \text{------}\mathit{Eq\text{-}1}$$

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 p_i of at least one task T_i, that is,

$$\lfloor p_i/f \rfloor - p_i/f = 0 \qquad \text{------Eq-2}$$

for at least one i. When this condition is met, there is an integer number of frames in each hyperperiod. We let F denote this number and call a hyperperiod that begins at the beginning of the $(kF + 1)$st frame, for any $k = 0, 1, \ldots,$ a *major cycle*.



$$2f - gcd(p_i, f) \leq D_i \qquad \text{-----}\mathit{Eq\text{-}3}$$

Job Slices: 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. 1 to be true, we must have $f \geq 5$, but to satisfy Eq. 3 we must have $f \leq 4$. In this situation,

Figure-1 A cyclic schedule with frame size 2.

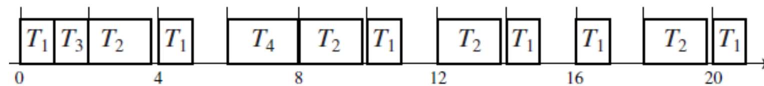
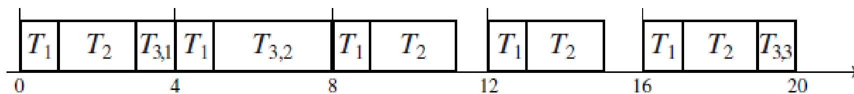


Figure 2 A preemptive cyclic schedule of $T_1 = (4, 1)$, $T_2 = (5, 2, 7)$ and $T_3 = (20, 5)$.



Example:

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-2 shows a cyclic schedule for these tasks. The three original tasks are called $T1$, $T2$ and $T3$, respectively, and the three subtasks of $T3$ are called $T3,1$, $T3,2$, and $T3,3$. To satisfy Eq.1, it suffices for us to partition each job in the task into two slices, one with execution time 3 and the other with execution time 2. However, a look at the schedule in Figure -2 shows the necessity of three slices. It

would not be possible to fit the two tasks (20, 3) and (20, 2) together with $T1$ and $T2$ in five frames of size 4. $T1$, with a period of 4, must be scheduled in each frame. $T2$, with a period of 5, must be scheduled in four out of the five frames. (The fact that the relative deadline of $T2$ is 7 does not help.) This leaves one frame with 3 units of time for $T3$. The other frames have only 1 unit of time left for $T3$. We can schedule two subtasks each with 1 unit of execution time in these frames, but there is no time in any frame for a subtask with execution time 2.

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.

From this example, we see that in the process of constructing a cyclic schedule, we have to make three kinds of design decisions: choosing a frame size, partitioning jobs into slices, and placing slices in the frames. In general, these decisions cannot be made independently. The more slices a job is partitioned into, the higher the context switch and communication overhead. Therefore, we want to partition each job into as few slices as necessary to meet the frame-size constraints

IMPROVING THE AVERAGE RESPONSE TIME OF APERIODIC JOBS:

Minimizing the response time of each aperiodic job or the average response time of all the aperiodic jobs is typically one of the design goals of real-time schedulers.

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. For the slack-stealing scheme described below to work, every periodic job slice must be scheduled in a frame that ends no later than its deadline.

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

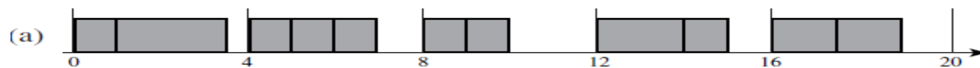


Figure (b) shows three aperiodic jobs $A1$, $A2$, and $A3$. Their release times are immediately before 4, 9.5. and 10.5, and their execution times are 1.5, 0.5 and 2, respectively.

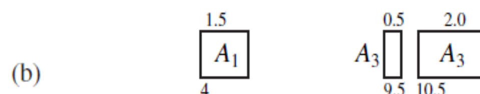


Figure (c) shows when the aperiodic jobs execute if we use the cyclic executive shown in Figure 5–7, which schedules aperiodic jobs after the slices of periodic tasks in each frame are completed. The execution of $A1$ starts at time 7. It does not complete at time 8 when the frame ends and is, therefore,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

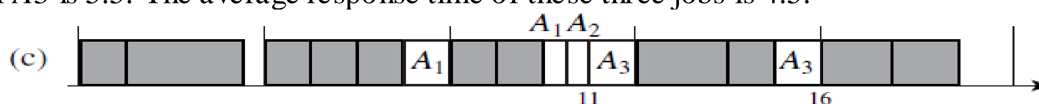
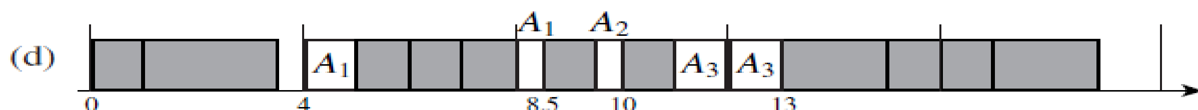


Figure (d) shows what happens if the cyclic executive does slack stealing. At time 4, the cyclic executive finds $A1$ in the aperiodic job queue, and there is 1 unit of slack. Hence it lets $A1$ execute. At time 5, there is no more slack. It preempts $A1$ and lets the periodic task server execute the 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 $A2$ ready, and lets $A2$ 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 $A3$ 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.



Average Response Time:

we are often required to guarantee that their average response time is no greater than some value. To give this guarantee, we need to be able estimate the average response time of these jobs. In general, an accurate estimate of the average response time can be found only by simulation and/or measurement. This process is time consuming and can be done only after a large portion of the system is designed and built. On the other hand, we can apply known results in queueing theory to get a rough estimate of the average response time as soon as we know some statistical behavior of the aperiodic jobs

To express average response time in terms of these parameters, let us consider a system in which there are *na* aperiodic tasks

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*.

SCHEDULING SPORADIC JOBS:

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 the time. Here, by *a job in the system*, we mean either a periodic job, for which time has already been allocated in the precomputed cyclic schedule, or a sporadic job which has been scheduled but not yet completed.

In general, more than one sporadic job may be waiting to be tested at the same time. A good way to order them is on the Earliest-Deadline-First (EDF) basis. In other words, newly released sporadic jobs are placed in a waiting queue ordered in nondecreasing order of their deadlines: the earlier the

deadline, the earlier in the queue. The scheduler always tests the job at the head of the queue and removes the job from the waiting queue after scheduling it or rejecting it.

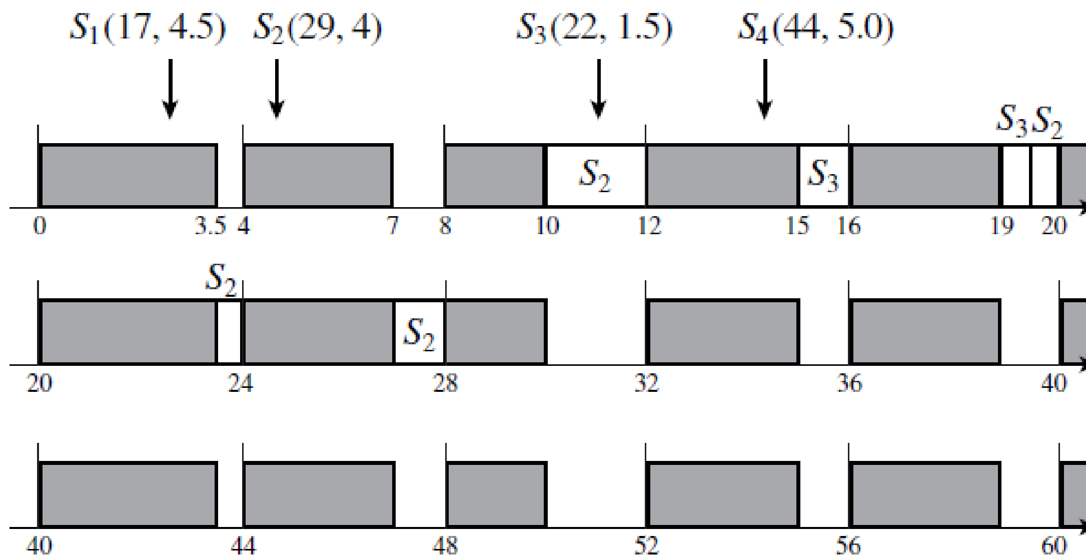
EDF Scheduling of the Accepted Jobs:

The EDF algorithm is a good way to schedule accepted sporadic jobs. For this purpose, the scheduler maintains a queue of accepted sporadic jobs in non decreasing order of their deadlines and inserts each newly accepted sporadic job into this queue in this order. 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 in the queue.

```
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.

Example: The frame size used here is 4. The shaded boxes show where periodic tasks are scheduled.



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 thecurrent 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.

Implementation of the Acceptance Test

to implement the acceptance test when accepted sporadic jobs are scheduled on the EDF basis. Specifically, we focus on an acceptance test at the beginning of frame *t* to decide whether a sporadic job *S(d, e)* should be accepted or rejected. 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*.

Optimality of Cyclic EDF Algorithm:

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 cyclic EDF algorithm for scheduling sporadic jobs is an on-line algorithm. You recall from our discussion in Section 4.8 that we measure the merit of an on-line algorithm by the value (i.e., the total execution time of all the accepted jobs) of the schedule produced by it: the higher the value, the better the 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.

PRACTICAL CONSIDERATIONS AND GENERALIZATIONS:

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

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.

Mode Changes:

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, except that it may be given the highest priority and executed ahead of other aperiodic jobs. 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 freed as soon as the current job in the task completes.

Sporadic Mode Change. A sporadic mode change has to be completed by a hard deadline. There are two possible approaches to scheduling this job

General Workloads and Multiprocessor Scheduling

It is probably obvious to you that the clock-driven approach is equally applicable to other types of workload, not just those executed on a CPU. For example, a bus arbitrator can use a scheduler

In general, searching for a feasible multiprocessor schedule is considerably more complex than searching for a uniprocessor schedule. However, since the search is done off-line, we can use

exhaustive and complex heuristic algorithms for this purpose. The next section presents a polynomial time hueristic algorithm for this purpose.

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. After presenting a polynomial time solution for this case, we then discuss how to take into account practical factors such as nonpreemptivity.

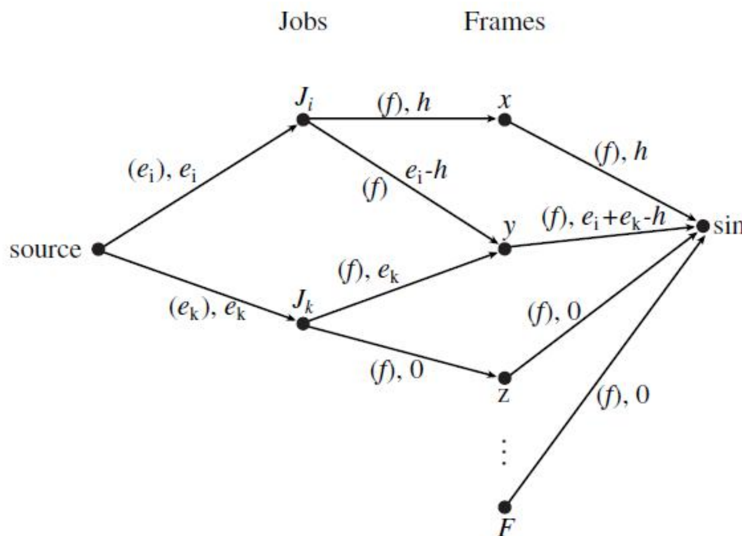
Scheduling Independent Preemptable Tasks:

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. (We note that the latter assumption imposes no restriction. Because we work with jobs' effective release times and deadlines, which were defined in Section 4.5, we can temporarily ignore all the precedence constraints.) Before applying the INF algorithm on the given system of periodic tasks, we find all the possible frame sizes of the system:

Network- Flow Graph. The algorithm used during each iteration is based on the well-known network-flow formulation [Blas] 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: (1) It is no greater than the capacity of the edge and (2) 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.

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.

Example:



Maximum Flow and Feasible Preemptive Schedule. Clearly, 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 .*

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. Precedence constraints among jobs can easily be taken into account as follows. You recall that we work with the effective release time and deadline of each job. 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. In Section 4.5 we discussed why this swapping can always be
done. We can try to transform a feasible preemptive schedule produced by the INF algorithm into one that satisfies constraints on the number of preemptions and times of preemptions. Unfortunately, there is no efficient optimal algorithm to do this transformation. This is not surprising since the problem of finding a feasible non preemptive schedule of jobs with different release times and deadlines is NP-hard, even on one processor.

PROS AND CONS OF CLOCK-DRIVEN SCHEDULING

Advantages:

- The important advantage is conceptual simplicity.
- Clock driven scheduling paradigm are time triggered. in this systems interrupts to external events are queued and polled periodically.
- Time triggered System Based on clock driven scheduling is easy to validate,test and certify.

Disadvantages:

- The system based on clock driven approach is Brittle (soft).
- The release time of all jobs must be fixed.
- In this system all combinations of periodic task that might execute at same time must be known apriori.
- The pure clock driven approach is not suitable for many systems that contain both hard and soft real time applications.