Clock-driven scheduling

Pei-Hsuan Tsai

CPS LAB

5.1 Notations and assumptions

Clock-driven approach for deterministic system

- Clock-driven approach to scheduling is applicable only when the system is deterministic, except for a few aperiodic and sporadic jobs to be accommodated in the deterministic framework.
- Restrictive assumptions of periodic task model for clock driven approaches:
  - **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. 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 $r_{i,k}$.

Notations of periodic task

- We refer to a periodic task T_i with phase φ_i , period p_i , execution time e_i , and relative deadline D_i by the 4-tuple $(\varphi_i , p_i , e_i , D_i)$.
- 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.

CPS LAB

- 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.

Other assumptions for clock-driven approaches

- There are aperiodic jobs released at unexpected time instants.
- For now we assume that there are no sporadic jobs.
- Focuses on scheduling tasks on one processor.

5.2 Static timer-driven scheduler

A queue for aperiodic tasks

- Assume that 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.
- Simply assume that they are ordered in a manner suitable for the applications in the system.
- Whenever the processor is available for aperiodic jobs, the job at the head of this queue executes.

Static schedule for hard deadline jobs

- Off-line construct a *static schedule* for jobs with hard deadlines and known before the system begins to execute.
- During run time, the scheduler dispatches the jobs according to this schedule.
- In the static schedule:
  - The amount of processor time allocated to every job is equal to its maximum execution time
  - Every job completes by its deadline.
  - All deadlines are surely met as long as no job ever *overruns*.
    - i.e., some rare or erroneous condition causes it to execute longer than its maximum execution time.

Advantage of off-line scheduling

- Because the schedule is computed off-line, we can afford to use complex, sophisticated algorithms.
- Among all the feasible schedules, we may want to choose one that is good according to some criteria
  - Feasible schedules are schedules where all jobs meet their deadlines
  - Criteria such as the processor idles nearly periodically to accommodate aperiodic jobs.

An example for periodic tasks

- 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 hyper-period of the tasks.
- Since the least common multiple of all periods is 20, the length of each hyper-period is 20.
- The entire schedule consists of replicated segments of length 20.

An arbitrary static schedule

- Figure 5–1 shows such a schedule segment on one processor.
- 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.

$T_1 = (4, 1)$, $T_2 = (5, 1.8)$, $T_3 = (20, 1)$, and $T_4 = (20, 2)$.

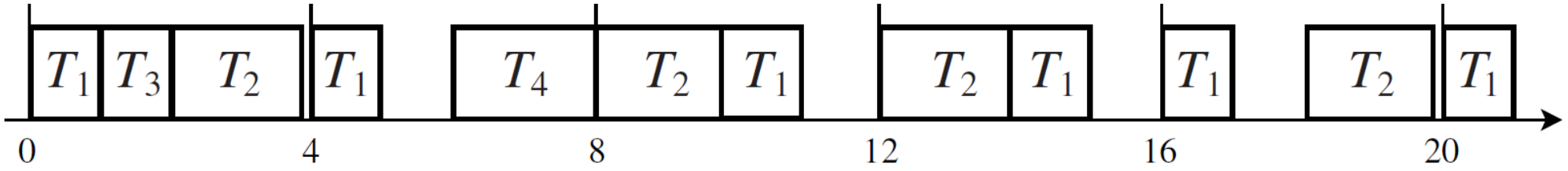


FIGURE 5–1 An arbitrary static schedule.

Not used interval

- Some intervals, such as (3.8, 4), (5, 6), and (10.8, 12), are not used by the periodic tasks.
- Make the unused intervals scattered periodically in the schedule.
- These intervals can be used to execute
  - aperiodic jobs
  - background non real-time jobs whose response times are uncritical to the performance of the system
  - some built-in self-test job that checks the status and monitors the health of the system

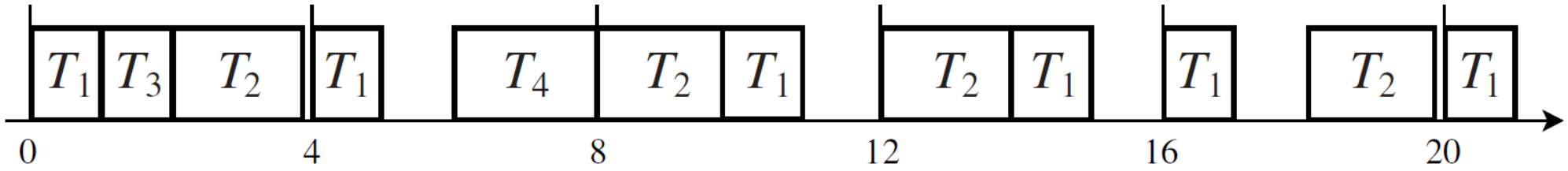


FIGURE 5–1 An arbitrary static schedule.

CPS LAB

Table to store precomputed schedule

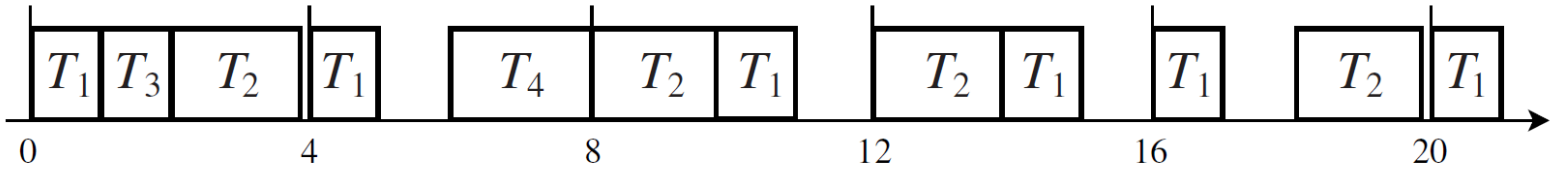


FIGURE 5–1 An arbitrary static schedule.

- The scheduler use a table to store the precomputed schedule.
- Each entry $(t_k , T(t_k))$ in this table gives
  - *decision time* t_k , which is an instant when a scheduling decision is made
  - $T(t_k)$, which is the name of the task whose job should start at t_k
  - I, which indicates an idle interval during which no periodic task is scheduled.

k	t_k	$T(t_k)$
0	0.0	T_1
1	1.0	T_3
2	2.0	T_2
3	3.8	I
4	4.0	T_1
5	5.0	I
6	6.0	T_4
7	8.0	T_2
8	9.8	T_1
9	10.8	I
10	12.0	T_2
11	13.8	T_1
12	14.8	I
13	17.0	T_1
14	17.0	I
15	18.0	T_2
16	19.8	I

Timers for scheduler

- During initialization (say at time 0), the operating system creates all the tasks that are to be executed.
- The scheduler makes use of a timer.
- After all the tasks have been created and initialized
  - At every scheduling decision time, the scheduler sets the timer.
  - The timer will expire and request an interrupt at the next decision time.
- Upon receiving a timer interrupt at t_k , the scheduler
  - sets the timer to expire at t_k+1
  - prepares the task T (t_k) for execution
  - It then suspends itself, letting the task have the processor and execute.
- When the timer expires again, the scheduler repeats this operation.

k	t_k	$T(t_k)$
0	0.0	T_1
1	1.0	T_3
2	2.0	T_2
3	3.8	I
4	4.0	T_1
5	5.0	I
6	6.0	T_4
7	8.0	T_2
8	9.8	T_1
9	10.8	I
10	12.0	T_2
11	13.8	T_1
12	14.8	I
13	17.0	T_1
14	17.0	I
15	18.0	T_2
16	19.8	I

Pseudo code of scheduler

```
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
```

- H is the length of the hyperperiod of the system.
- N is the number of entries in the schedule of each hyper-period.
- 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.

CPS LAB

Example of schedule table

- The stored table contains 17 entries.
- They are (0, T_1), (1, T_3), (2, T_2), (3.8, I), (4, T_1), . . . (19.8, I).
- Hence, the timer is set to expire at 0, 1, 2, 3.8, and so on.
- At these times, the scheduler schedules the execution of tasks T_1, T_3, T_2, and an aperiodic or background job, respectively.
- The table is used again during the next hyperperiod, and new decision times 20, 21, 22, 23.8, and so on, can be obtained from the times in the first hyperperiod as described in Figure 5–2.

k	t_k	$T(t_k)$
0	0.0	T_1
1	1.0	T_3
2	2.0	T_2
3	3.8	I
4	4.0	T_1
5	5.0	I
6	6.0	T_4
7	8.0	T_2
8	9.8	T_1
9	10.8	I
10	12.0	T_2
11	13.8	T_1
12	14.8	I
13	17.0	T_1
14	17.0	I
15	18.0	T_2
16	19.8	I

- 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.

5.3 General structure of cyclic schedules

Cyclic schedule with structure

k	t_k	$T(t_k)$
0	0.0	T_1
1	1.0	T_3
2	2.0	T_2
3	3.8	I
4	4.0	T_1
5	5.0	I
6	6.0	T_4
7	8.0	T_2
8	9.8	T_1
9	10.8	I
10	12.0	T_2
11	13.8	T_1
12	14.8	I
13	17.0	T_1
14	17.0	I
15	18.0	T_2
16	19.8	I

- Rather than using ad hoc cyclic schedules, we may want to use a schedule that has a certain structure that scheduling decisions are made periodically, rather than at arbitrary times.
- Figure 5–3 shows a good structure of cyclic schedules [BaSh].

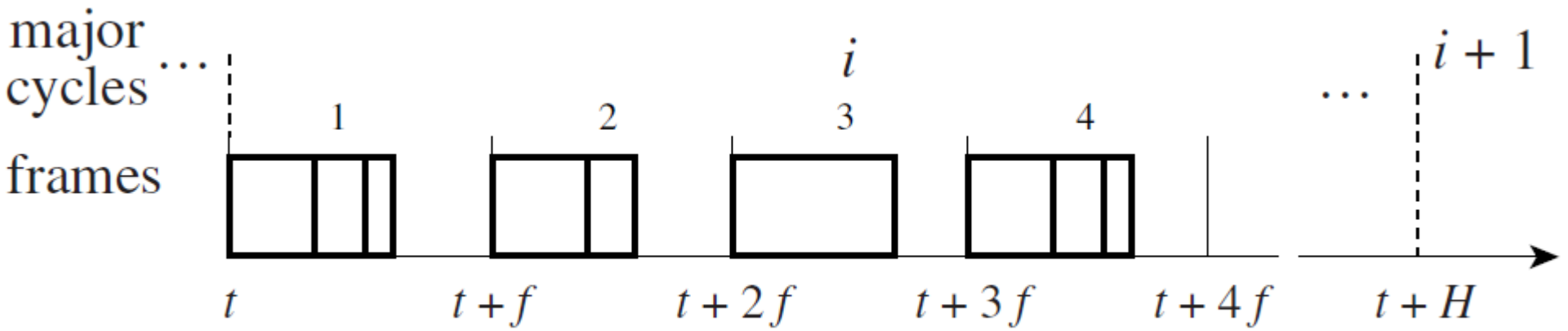


FIGURE 5–3 General structure of a cyclic schedule.

CPS LAB

5.3.1 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.

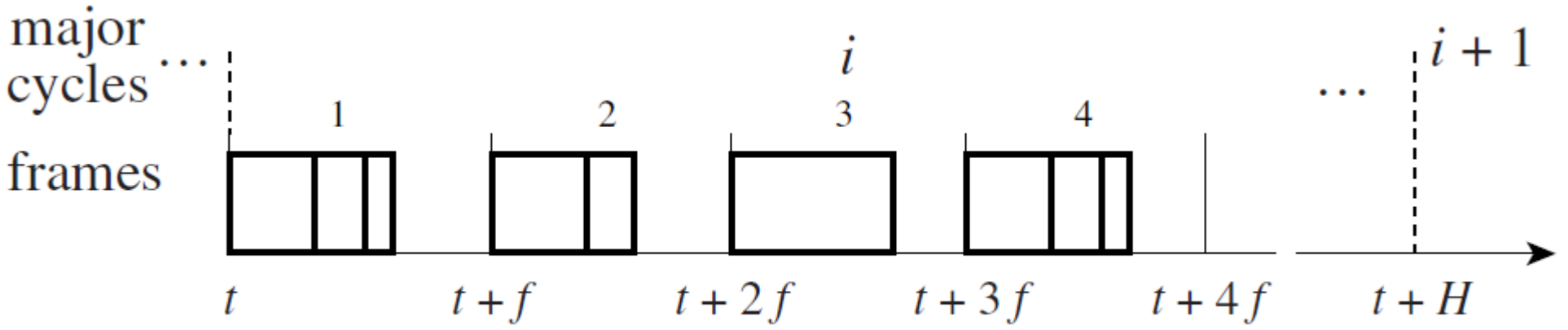


FIGURE 5–3 General structure of a cyclic schedule.

5.3.2 Frame Size Constraints - I

- 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,

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

5.3.2 Frame Size Constraints - II

- The frame size f should be chosen so that it divides H, the length of the hyper-period of the system.
- This condition is met when f divides the period p_i of at least one task T_i , that is, for at least one i .

$$\lfloor p_i / f \rfloor - p_i / f = 0$$

When this condition is met, there is an integer number of frames in each hyper-period.

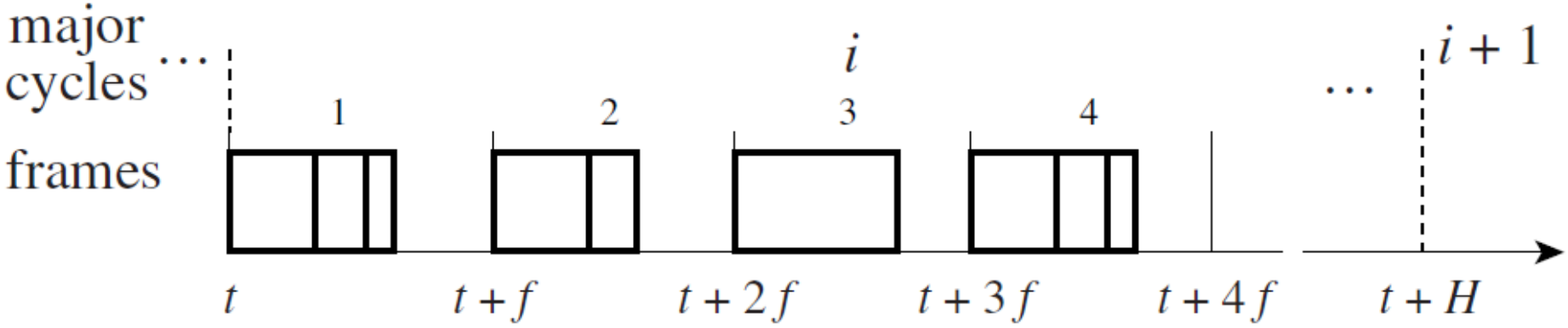


FIGURE 5–3 General structure of a cyclic schedule.

CPS LAB

5.3.2 Frame Size Constraints - III

- For the scheduler to determine whether every job completes by its deadline, we want the frame size between the release time and deadline of every job.
- Figure 5–4 illustrates the suitable range of f for a task $T_i = (p_i , e_i , D_i)$.
- In this figure, t denotes the beginning of a frame (called the k^{th} frame) in which a job in T_i is released, and t' denotes the release time of this job.
- We need to consider two cases: $t' > t$ and $t' = t$.

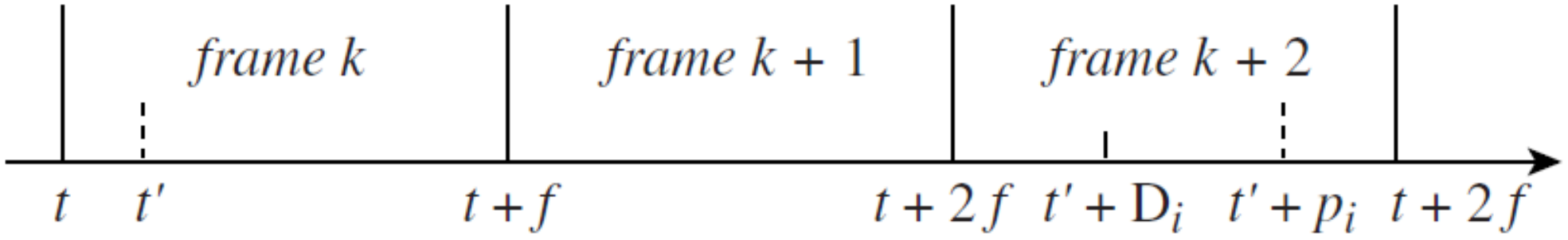


FIGURE 5–4 A constraint on the value of frame size.

CPS LAB

Job release time t' > the beginning of a frame 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 $t' + D_i$ of this job.
- For this to be true, we must have t +2 f equal to or earlier than $t + D_i$, that is, $t + 2f \leq t' + D_i$ ➔ $2f - (t' - t) \leq D_i$.
- Because the difference $t' - t$ is at least equal to the greatest common divisor $gcd(p_i , f)$ of p_i and f , this condition is met if the following inequality holds:

輾轉相除法

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

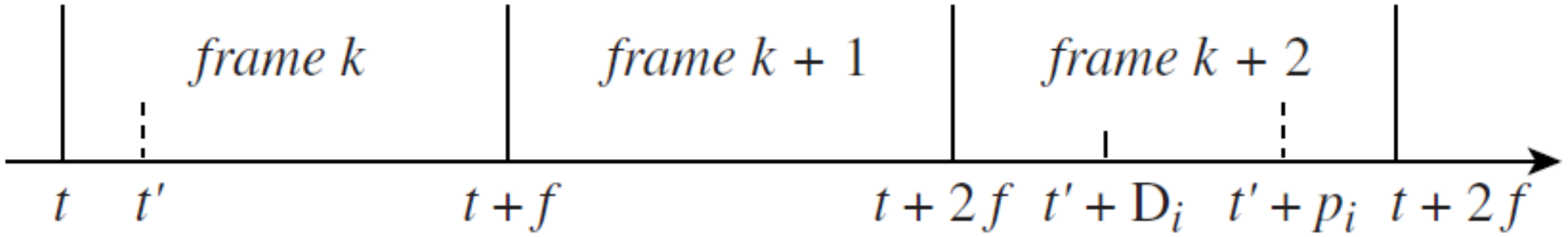
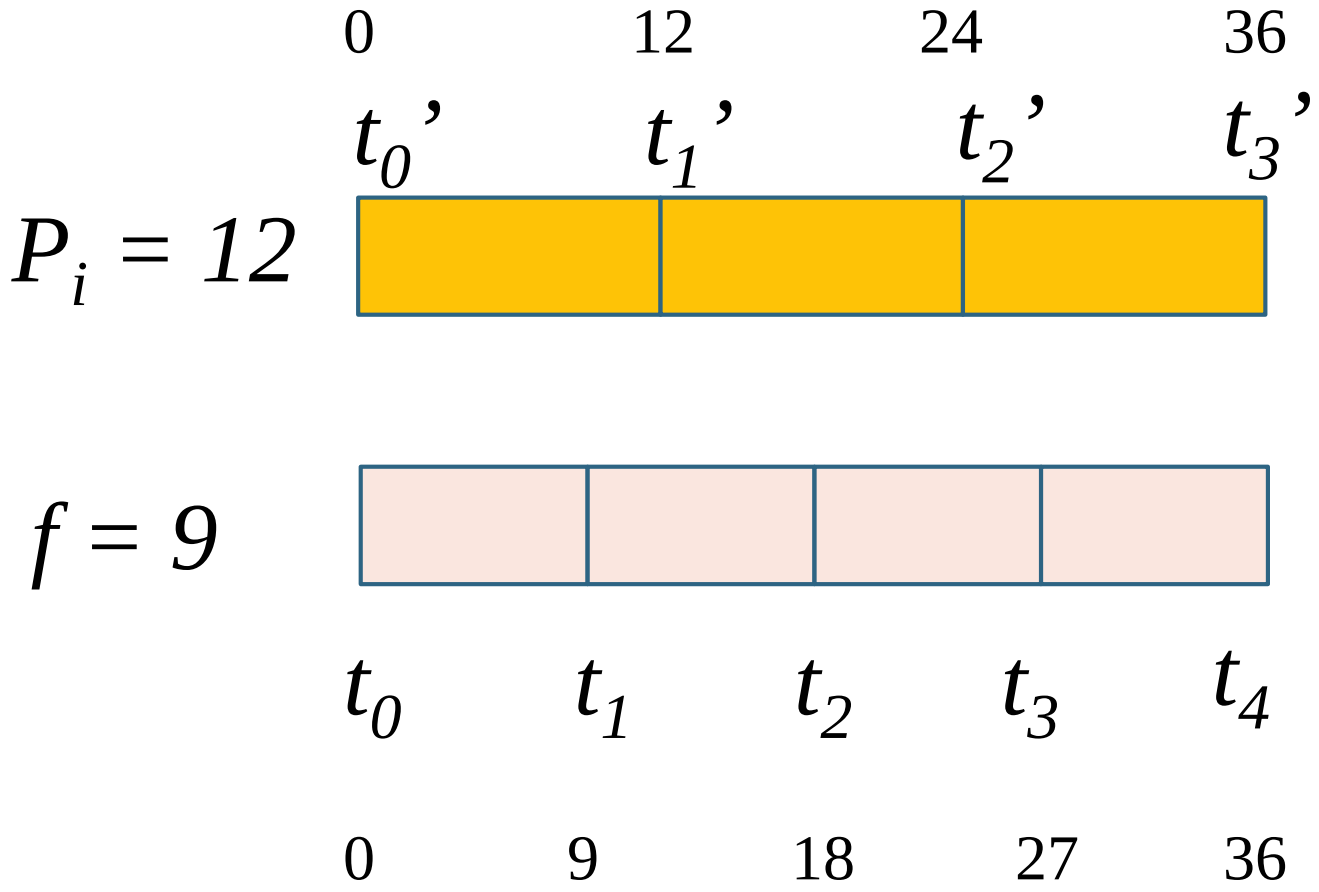


FIGURE 5–4 A constraint on the value of frame size.

CPS LAB

$t' - t$ is at least equal to the greatest common divisor $gcd(p_i , f)$



Possible $t' - t = (3 \text{ and } 6) \geq gcd(p_i, f) = 3.$

CPS LAB

t' is equal to t

- In the special case when t' is equal to t , it suffices to choose a frame size that is equal to or smaller than D_i .
- The condition $f \leq D_i$ is satisfied for all values of f that satisfy Eq. (5.3) and, hence, does not need to be considered separately.
- $2f - (t' - t) \leq D_i$ ➔ $2f \leq D_i$
- We refer to Eqs. (5.1), (5.2) and (5.3) as the *frame-size constraints*.

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

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

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

CPS LAB

Example as Figure 5-1

- For the four tasks in Figure 5–1, $T_1(4,1)$, $T_2(5,1.8)$,$T_3(20,1)$,$T_4(20,2)$.
- Eq (5.1) constrains the frame size to be no less than 2.
  - $f \geq \text{Max}(1, 1.8, 1, 2) = 2$.
- Their hyper-period length is 20;
  - Lcm(4, 5, 20, 20) = 20
- Hence, 2, 4, 5, 10 and 20 are possible frame sizes according to Eq. (5.2).
  - Factor (20) = 2, 4, 5, 10, 20.
- However, only 2 satisfies Eq. (5.3).
  - $2f - \gcd(p_i, f) \leq D_i$.
- Therefore, we an use the cyclic schedule shown in Figure 5–5.

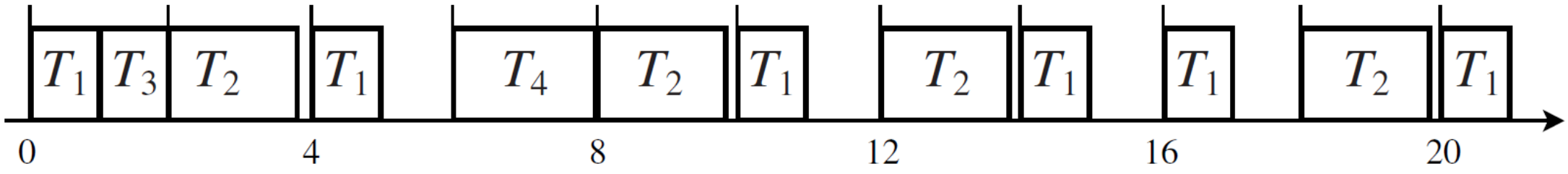


FIGURE 5–5 A cyclic schedule with frame size 2.

CPS LAB

Another example

- As another example, we consider the tasks (15, 1, 14), (20, 2, 26), and (22, 3).
- Because of Eq. (5.1), we must have $f \geq 3$;
  - $f \geq \text{Max}(1, 2, 3) = 3$
- Because of Eq. (5.2), we must have $f = 3, 4, 5, 10, 11, 15, 20$, and 22;
  - Hyper-period = LCM (15,30,22) = 330
  - Factor (330) = 3, 4, 5, 10, 11, 15, 20, 22.
- Because of Eq. (5.3), we must have $f = 3, 4$ or 5.
  - $2f - \gcd(p_i, f) \leq D_i$
- Therefore the possible choices of the frame size are 3, 4, and 5.

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 $\mathbf{T} = \{(4, 1), (5, 2, 7), (20, 5)\}$.
- For Eq. (5.1) to be true, we must have $f \geq 5$,
  - $f \geq \text{Max}\ (1, 2, 5) = 5$
- For Eq. (5.2) to be true,
  - Hyper-period = LCM(4, 5, 20) = 20
  - Factor(20) = 2, 4, 5, 10, 20.
- but to satisfy Eq. (5.3) we must have $f \leq 4$.
- 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.
  - Ex: When the job is a message transmission, we divide the message into several segments.
  - Ex: When the job is a computation, we partition the program into procedures, each of which is to be executed nonpreemptively.

Reduce the lower bound of *f* imposed by Eq. (5.1)

- For **T** = {(4, 1), (5, 2, 7), (20, 5)},
  - Divide each job in (20, 5) into a chain of three slices with execution times 1, 3, and 1.
- 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 $T_{3,3}$.

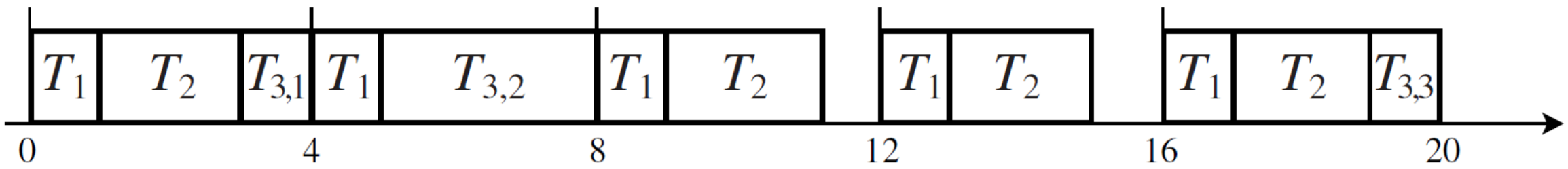


FIGURE 5–6 A preemptive cyclic schedule of $T_1 = (4, 1)$, $T_2 = (5, 2, 7)$ and $T_3 = (20, 5)$.

CPS LAB

Why we choose to decompose (20, 5) into three subtasks?

- To satisfy Eq. (5.1), $f \geq \max_{1 \leq i \leq n}(e_i)$ it suffices to partition each job into two slices, (20, 3), (20,2)

  (20,3) (20,2)這種切法不可行

- Not be possible to fit the two tasks (20, 3) and (20, 2) together with T_1 and T_2 in five frames of size 4.
- T_1, with a period of 4, must be scheduled in each frame.
- T_2, with a period of 5, must be scheduled in four out of the five frames.
- This leaves one frame with 3 units of time for T_3.
- The other frames have only 1 unit of time left for T_3.
- 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.

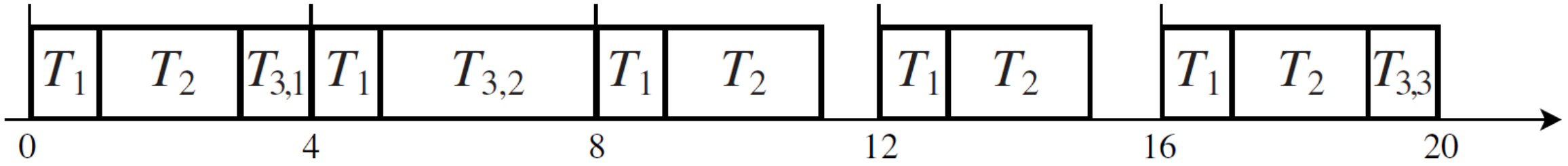


FIGURE 5–6 A preemptive cyclic schedule of $T_1 = (4, 1)$, $T_2 = (5, 2, 7)$ and $T_3 = (20, 5)$.

Process of constructing a cyclic schedule

- Three kinds of design decisions:
  - choosing a frame size,
  - partitioning jobs into slices, and
  - placing slices in the frames.
- These decisions cannot be made independently.
- The more slices a job is partitioned into, the higher the context switch and communication overhead.
- Therefore, few slices as necessary to meet the frame-size constraints.
- Unfortunately, this goal is not always attainable.
- There may not be any feasible schedule for the choices of frame size and job slices because it is impossible to pack the large job slices into the frames by their deadlines.
- In contrast, feasible schedules may exist if we choose smaller job slices.
- For this reason, we sometimes are forced to partition some jobs into more slices than needed to meet the frame size constraints.

5.4 Cyclic Executives

Structured Cyclic Schedule

- Arbitrary table-driven cyclic schedules flexible, but inefficient
  - Relies on accurate timer interrupts, based on execution times of tasks
  - High scheduling overhead
- Easier to implement if structure imposed:
  - Make scheduling decisions at periodic intervals (frames) of length f
  - Execute a fixed list of jobs with each frame, disallowing preemption except at frame boundaries
  - Require phase of each periodic task to be a non-negative integer multiple of the frame size
    - The first job of every task is released at the beginning of a frame
  - $\varphi = k * f$ where k is a non-negative integer
- Gives two benefits:
  - Scheduler can easily check for overruns and missed deadlines at the end of each frame
  - Can use a periodic clock interrupt, rather than programmable timer

Implementation : A Cyclic Executives

- Modify the clock-driven scheduler that only makes scheduling decisions at frame boundaries.
- Cyclic executives:
  - A table-driven cyclic scheduler in a multithreaded system.
  - It makes scheduling decisions only at the beginning of each frame.
  - Deterministically interleaves the execution of periodic tasks.
  - It allows aperiodic and sporadic jobs to use the time not used by periodic tasks.

Cyclic executive

- The pseudocode describes such a cyclic executive on a CPU.
- The stored table (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.
- 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 essence, the cyclic executive takes over the processor and executes at each of the clock interrupts, (at the beginning of frames).
- When it executes, the cyclic executive copies the table entry for the current frame into the current block.
- It then wakes up a job, called *periodic task server*, and lets the server execute the job slices in the current block.
- Upon the completion of the periodic task server, the cyclic executive wakes up the aperiodic jobs in the aperiodic job queue in turn and allows them to use the remaining time in the frame.
- The assumption here is that whenever the server or a job completes, the cyclic executive wakes up and executes.
- Alternatively, the system may have an aperiodic task server, which when awaked executes aperiodic jobs in the aperiodic job queue.

```
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.

Check for overrun

- The cyclic executive also checks for overruns at the beginning of each frame.
- If the last job is an aperiodic job
  - preempts the execution of the job.
  - remains in the aperiodic job queue
  - resumed whenever there is time again for aperiodic jobs.
- If the periodic task server still executes 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.

```
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.

- After checking for overruns, the cyclic executive makes sure that all the job slices scheduled in the current block are ready for execution and then wakes up the periodic task server to execute them.
- If there is still time after all the slices in the current block are completed and the aperiodic job queue is nonempty, it lets the job at the head of the aperiodic job queue execute.

```
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.

CPS LAB

5.5 Importing the average response time of aperiodic jobs

5.5.1 Slack stealing

- Aperiodic jobs are scheduled in the background after all the job slices with hard constraints scheduled in each frame are completed.
- It is not good because there is no advantage to completing a job with hard deadline early.
- The sooner an aperiodic job completes, the more responsive the system is.
- 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 is called slack stealing.

Slack stealing

- Let the total amount time allocated to all the slices scheduled in the frame k be x_k.
- The slack time available in the frame is equal to f-x_k at the beginning of the frame.
- If the aperiodic job queue is nonempty at this time, the cyclic executive can let the aperiodic jobs execute for this amount of time without causing any job miss its deadline.

An illustrative example of slack stealing

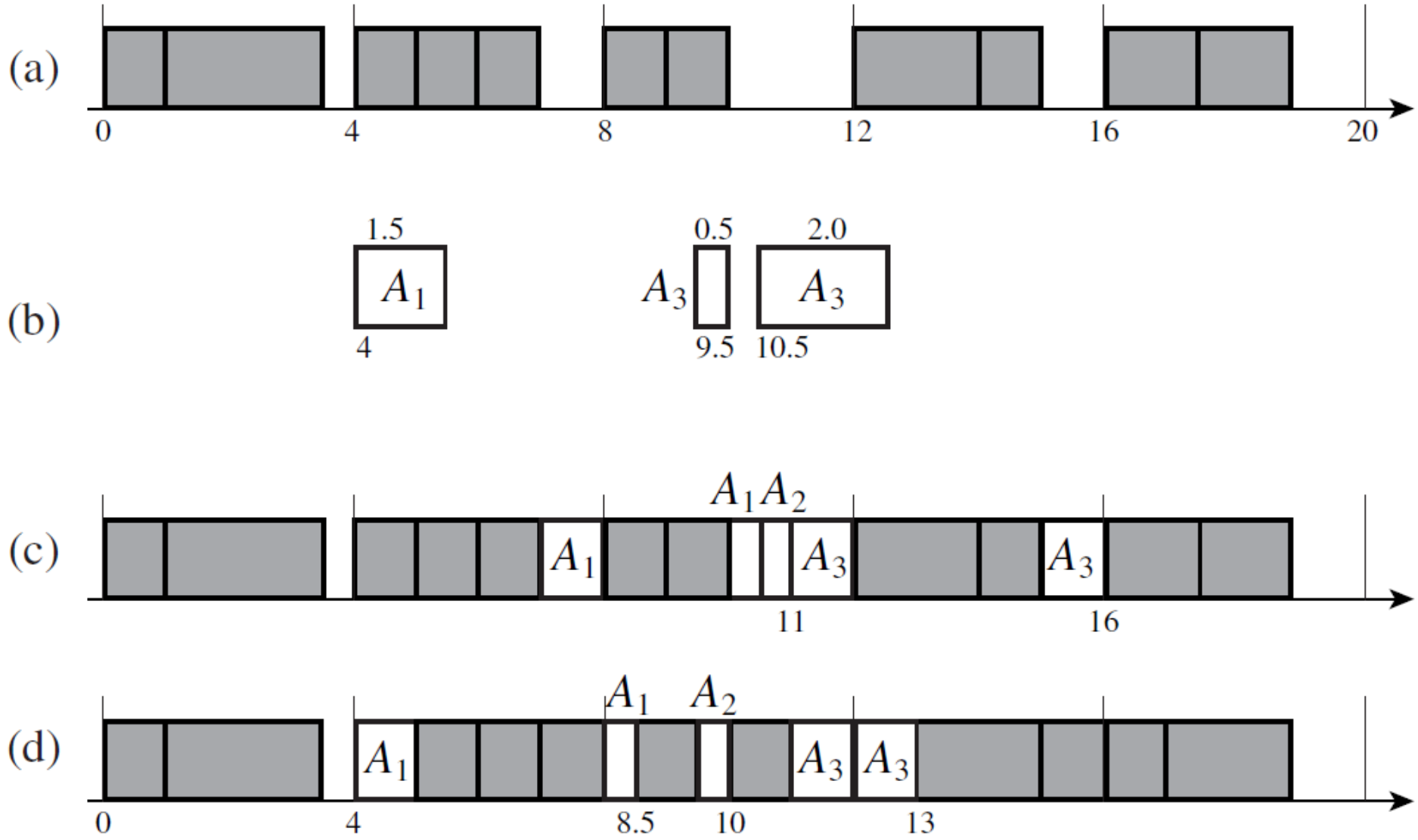


FIGURE 5–8 Example illustrating slack stealing.

(c) Response time :
A1: 10.5 -4 = 6.5
A2: 11-9.5 = 1.5
A3: 16-10.5 = 5.5
Average response time = 4.5

(d) Response time :
A1: 8.5 -4 = 4.5
A2: 10-9.5 = 0.5
A3: 13-10.5 = 2.5
Average response time = 2.5

CPS LAB

5.5.2 Average response time

- While we are not required to ensure the completion of aperiodic jobs by some specific times, 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.
- For example, a requirement of the system may be that it must respond satisfactorily as long as the average rate of arrival (i.e., releases) of aperiodic jobs is within a given limit.
- We can estimate the average response time of the system for this average arrival rate.
- We also need to know the mean and mean square values of the execution times of aperiodic jobs.
- These values can be estimated by analysis, simulation, and/or measurement of the jobs' execution by themselves.

- 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 inter arrival-time and execution time distributions and the same response-time requirement.
- Suppose that the average rate of arrival of aperiodic jobs in the ith aperiodic task is λ_i jobs per unit of time.
- The sum λ of λ_i 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 ith aperiodic task are $E[\beta_i]$ and $E[\beta_i^2]$, respectively.
- (Here $E[x]$ denotes the mean value of the random variable x.
- $E[x]$ is also called the expected value of x, hence the choice of the letter E.)

- Let u_i denote the average utilization of the ith task; it is the average fraction of processor time required by all the jobs in the task.
- u_i 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 [Klie]:

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

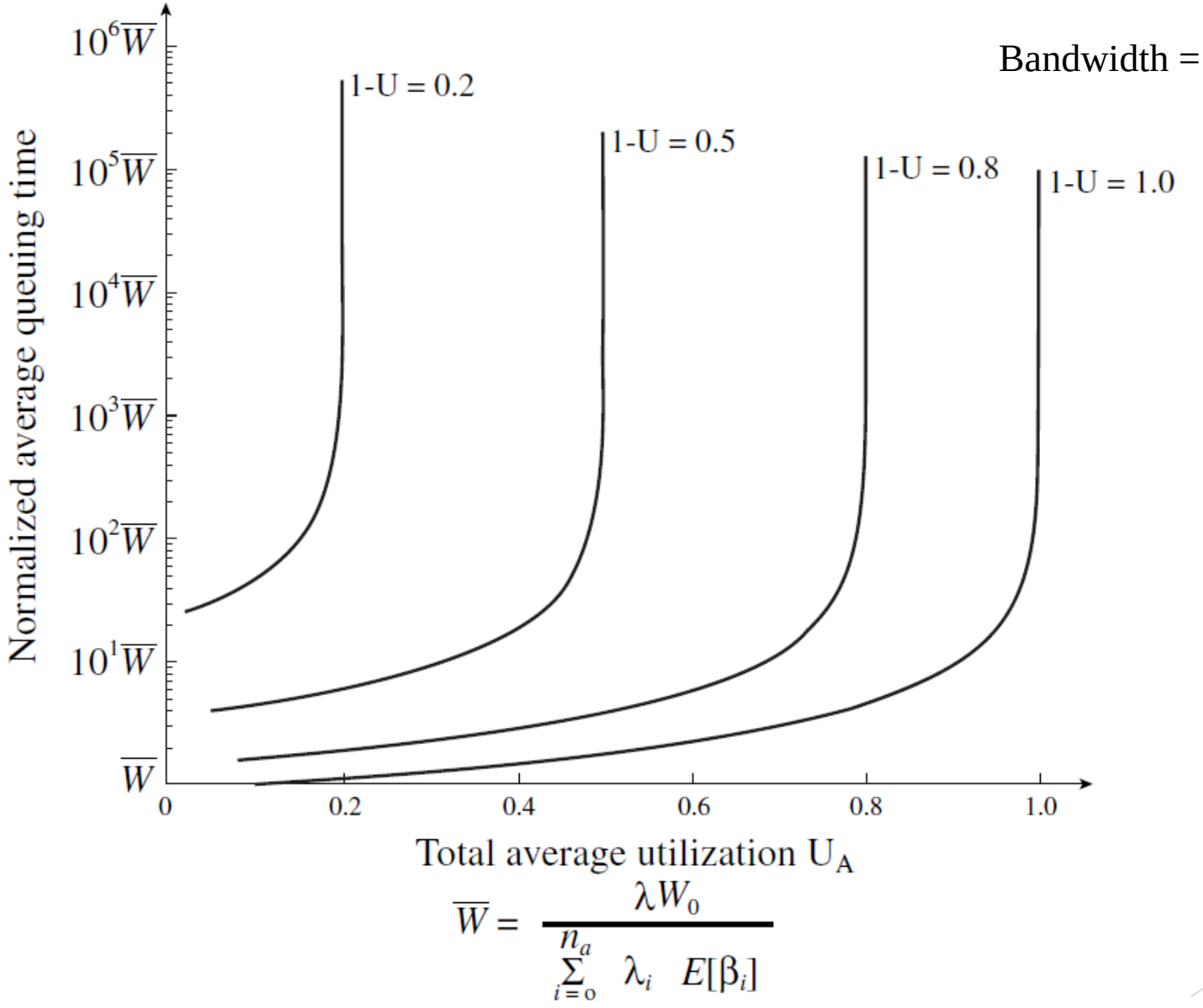
: W_0 is given by

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

- The first term in Eq. (5.4a) gives the average amount of time required by an aperiodic job to complete execution if it does not wait for any aperiodic job.
- The second term gives us the average queueing time, which is the average amount of time a job waits in the queue.

- Figure 5–9 shows the behavior of the average queueing time, normalized with respect to the average execution time $na\ i{=}1\ \lambda i\ E[\beta i\ ]/\lambda$ of all the aperiodic jobs, as a function of the total average utilization UA of aperiodic jobs for different values of aperiodic bandwidth $(1 - U)$.
- 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 $(1 - U)$.

Bandwidth = 1-U



$$\overline{W} = \frac{\lambda W_0}{\sum_{i=0}^{n_a} \lambda_i \quad E[\beta_i]}$$

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

CPS LAB

5.6 Scheduling sporadic jobs

- Like jobs in periodic tasks, sporadic jobs have hard deadlines.
- But 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

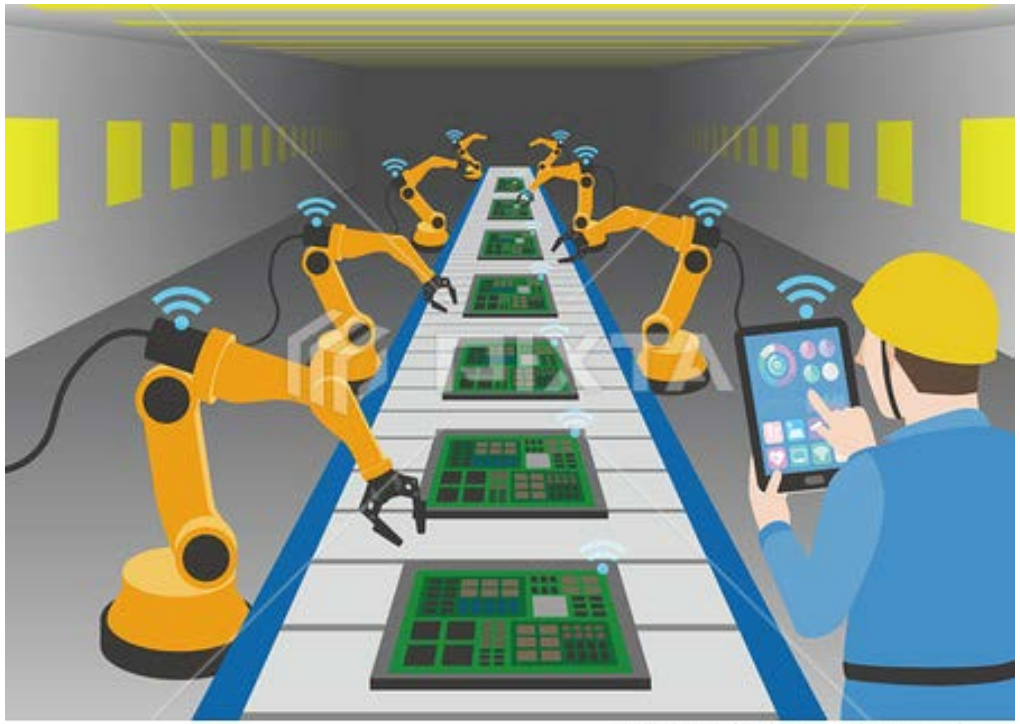
- A common way to deal with this situation is to have the scheduler perform an acceptance test when each sporadic job is released.
- During an *acceptance test*, the scheduler checks whether the newly released sporadic job can be feasibly scheduled with all the jobs in the system at the time.
- Here,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

5.6.1 Acceptance Test

- The scheduler accepts and schedules the job if according to the existing schedule,
  - there is a sufficient amount of time in the frames for the newly released sporadic job's to be completed before its deadline
  - without causing any job in the system to complete too late.
- Otherwise, the scheduler rejects the new sporadic job.
- By rejecting a sporadic job that cannot be scheduled to complete in time immediately after the job is released, the scheduler gives the application system as much time as there is to take any necessary recovery action. (應用程式能得到足夠時間執行recovery機制，而不是錯失deadline才獲通知。)

An example of quality control system

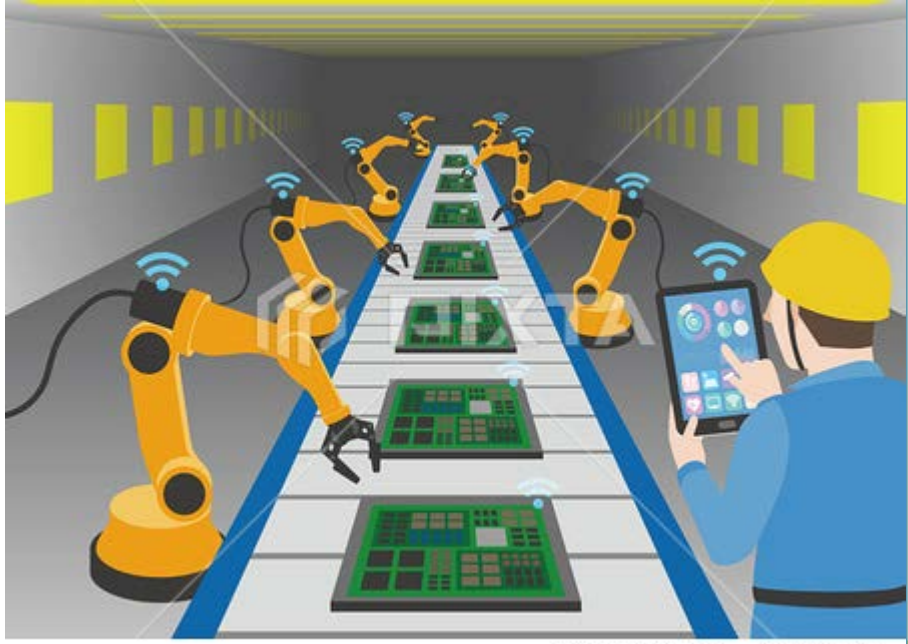
- A sporadic job that activates a robotic arm is released when a defective part is detected.
- The arm, when activated, removes the part from the conveyor belt. (輸送帶)
- This job must complete before the part moves beyond the reach of the arm.



CPS LAB

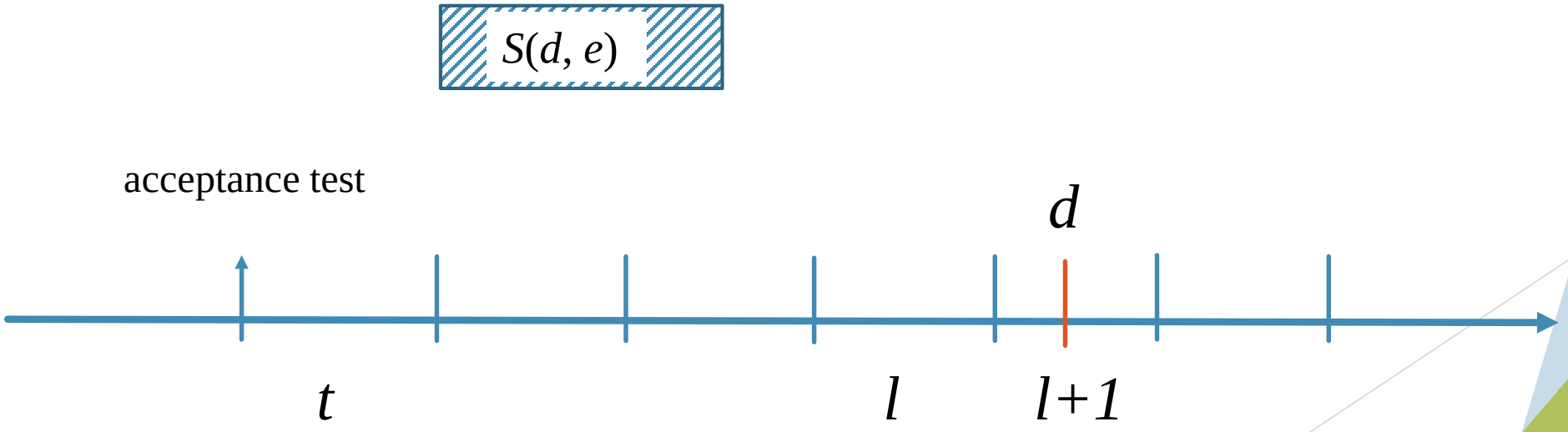
An example of quality control system

- When the job cannot be scheduled to complete in time, it is better for the system to have this information as soon as possible.
- The system can
  - slow down the belt,
  - stop the belt,
  - or alert an operator to manually remove the part.
- Otherwise, if the sporadic job were scheduled but completed too late, its lateness would not be detected until its deadline.
- By the time the system attempts a recovery action, the defective part may already have been packed for shipment, too late for simple recovery actions to be effective.



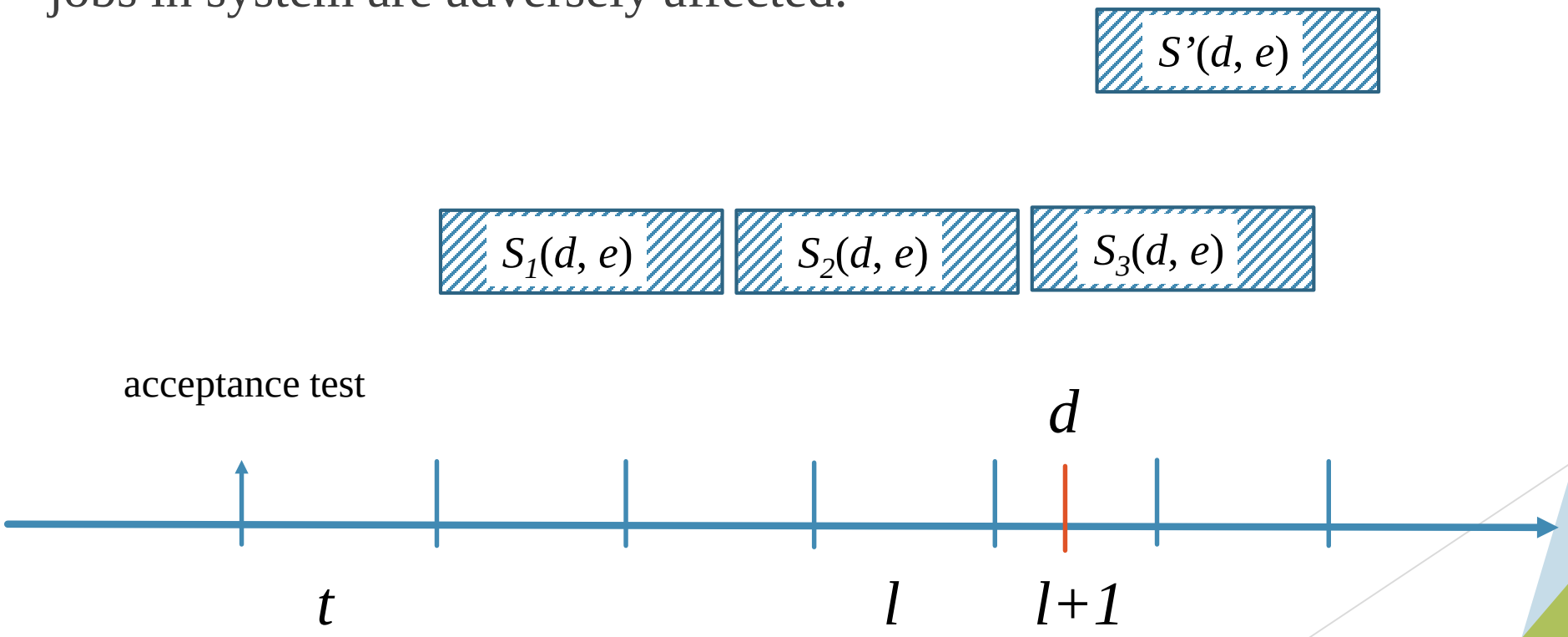
- Assumption: the maximum execution time of each sporadic job becomes known upon its release.
  - It is impossible for the scheduler to determine which sporadic jobs to admit and which to reject unless this information is available.
- The scheduler must maintain information on the maximum execution times of all types of sporadic jobs that the system may execute in response to the events it is required to handle.
- We also assume that all sporadic jobs are preemptable.
- Therefore, each sporadic job can execute in more than one frame if no frame has a sufficient amount of time to accommodate the entire job.

- 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 (i.e., frame l ends before d but frame l+1 ends after d) 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 $e > \sigma_c(t, l)$.

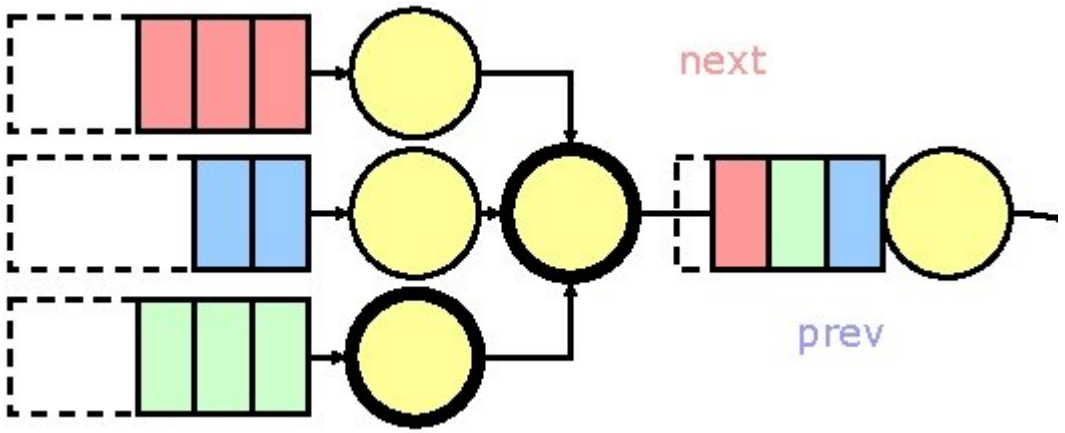


CPS LAB

- The scheduler may let a new sporadic job execute ahead of some previously accepted sporadic jobs.
- 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.



- 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 non-decreasing 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.



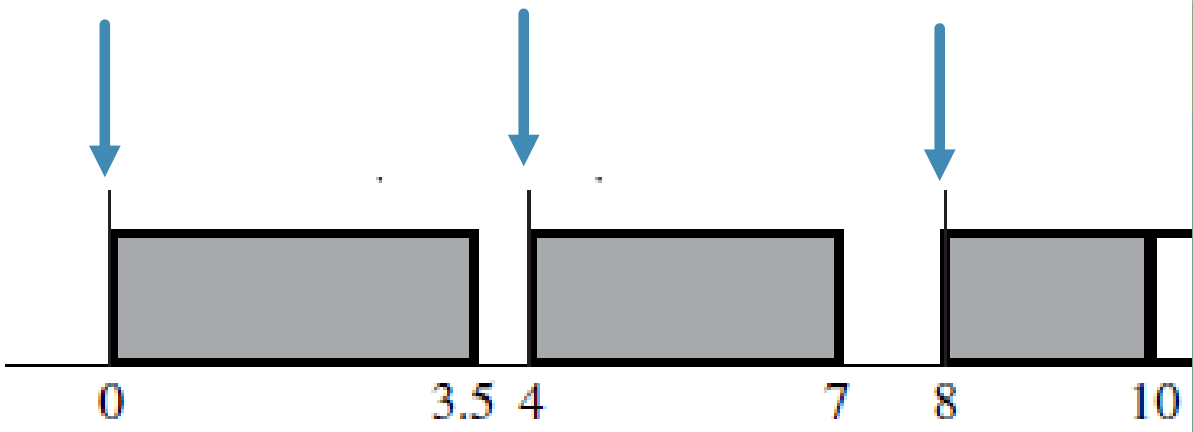
CPS LAB

5.6.2 EDF Scheduling of the Accepted Jobs

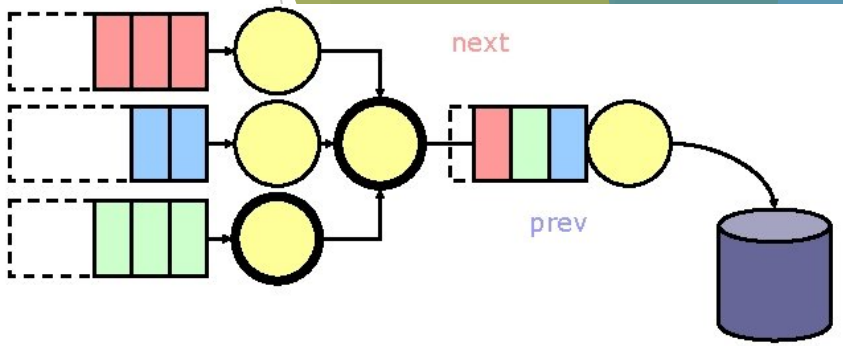
- EDF algorithm is a good way to schedule accepted sporadic jobs.
- For this purpose, the scheduler maintains a queue of accepted sporadic jobs in non decreasing order of their deadlines and inserts each newly accepted sporadic job into this queue in this order.
- Whenever all the slices of periodic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

Pseudocode of the modified cyclic executive with sporadic and aperiodic job scheduling capability

Input: Stored schedule: $L(k)$ for $k = 0, 1, \ldots F - 1$;
    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 $= L(k)$;
        $t = t + 1$;
        $k = t \bmod F$;
        if the last job is not completed, take appropriate action;
        if any of the slices in the currentBlock is not released, take appropriate action;



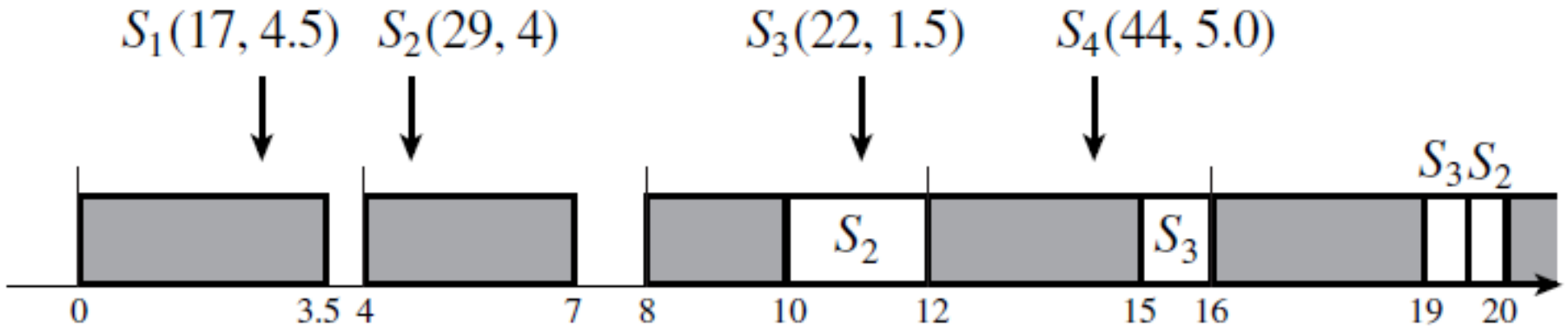
```
while the sporadic-job waiting queue is not empty,
    remove the job at the head of the sporadic job waiting queue;
    do an acceptance test on the job;
    if the job is acceptable,
        insert the job into the accepted-sporadic-job queue in the EDF order;
        else, delete the job and inform 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;
```



CPS LAB

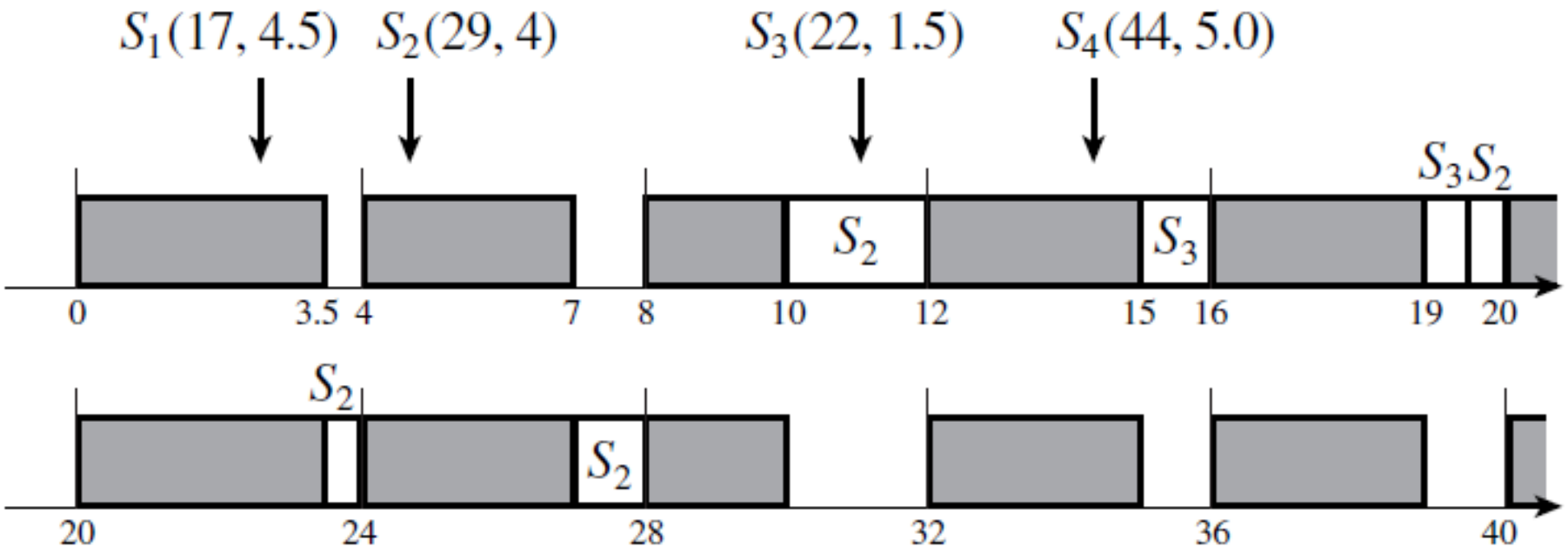
An example of scheduling sporadic jobs

- 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.



CPS LAB

- 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.



CPS LAB

- 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.

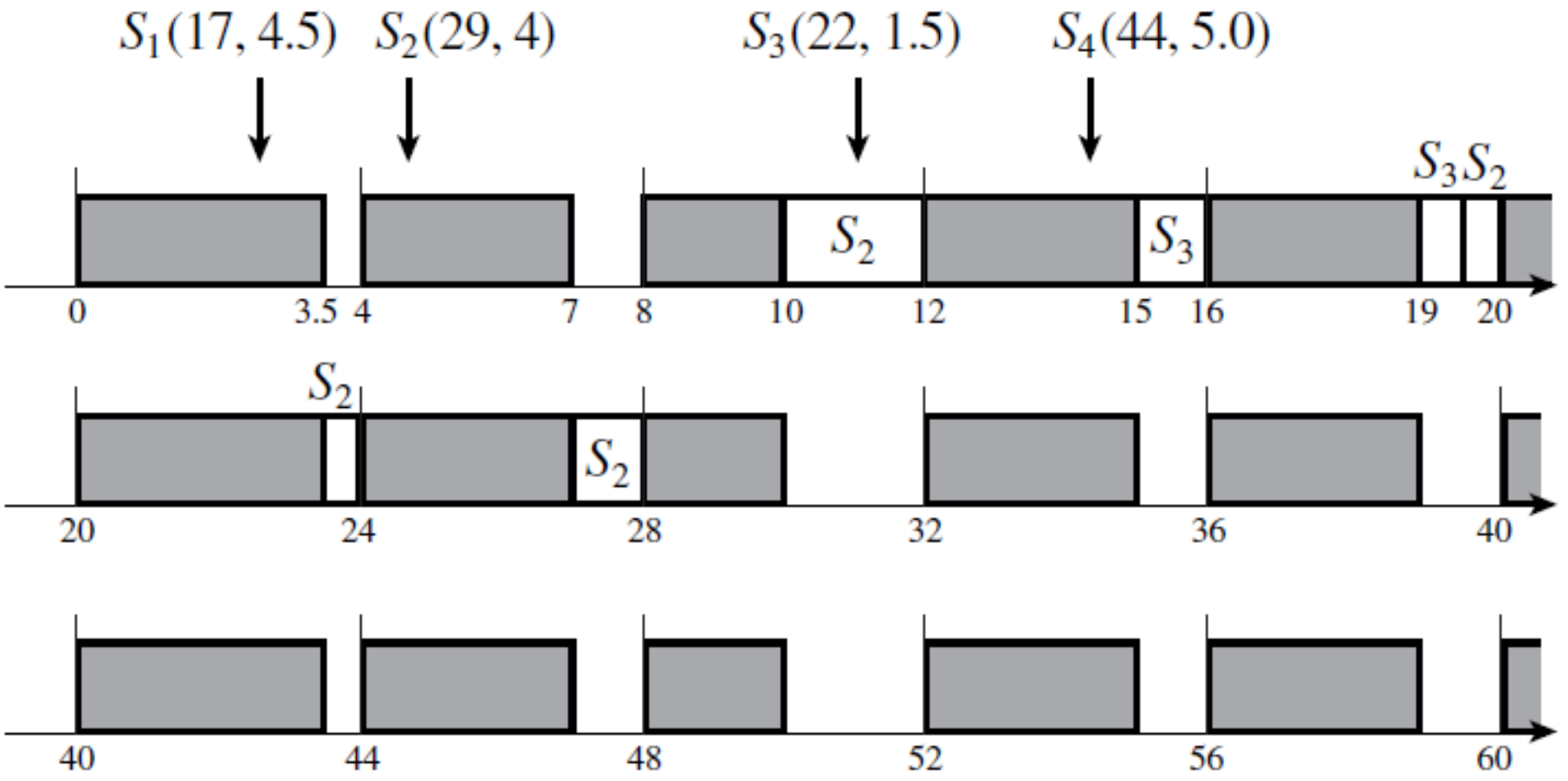


FIGURE 5–11 Example of scheduling sporadic jobs.

- 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.

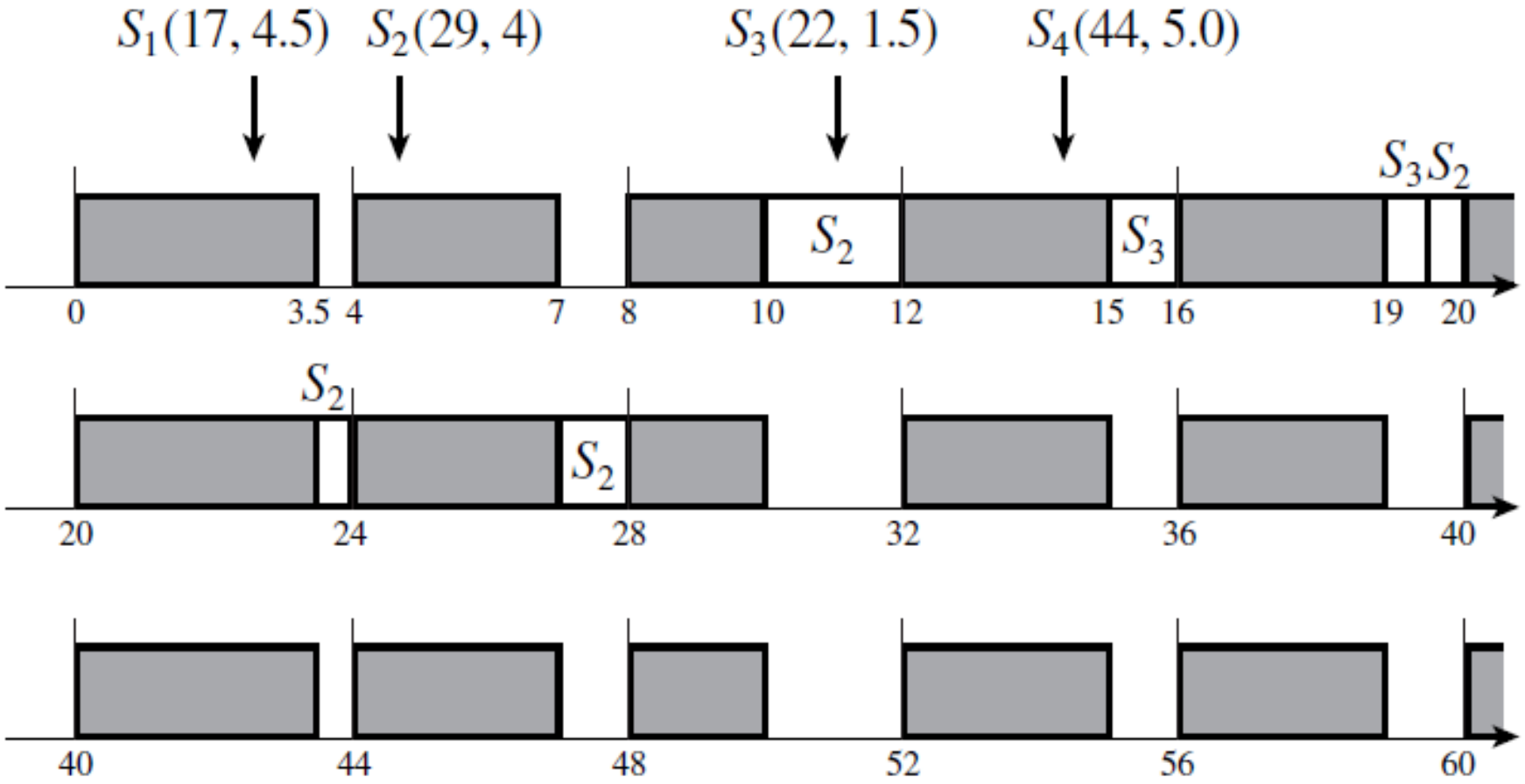


FIGURE 5–11 Example of scheduling sporadic jobs.

5.6.3 Implementation of the acceptance test with EDF basis queue.

- The acceptance test consists of the following two steps:
- Step 1: The scheduler 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.
- Step 2: The scheduler determines whether any sporadic job in the system will complete late if it accepts *S*.
  - If the answer is no, it accepts *S*.
  - If the answer is yes, it rejects *S*.
- Again, a sporadic job in the system is one that was accepted earlier but is not yet complete at the time.
- We note that there is no need to check whether any periodic job might miss its deadline because sporadic jobs in the system never affect the execution of any periodic job slice scheduled in any frame.

- To do an acceptance test, the scheduler needs the current total amount of slack time $\sigma_c(i, k)$ 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* $\sigma(i, k)$, for $i, k = 1, 2, \ldots F$ and storing them in a slack table along with the precomputed cyclic schedule at the cost of $O(F^2)$ in storage space.
- From the initial amounts of slack in the frames in the first major cyclic, initial amounts of slack in any later frames can be computed as follows.
- For any $0 < j < j'$ and any i and k equal to $1, 2, \ldots F$, 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

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

Use Figure 5-11 as an example.

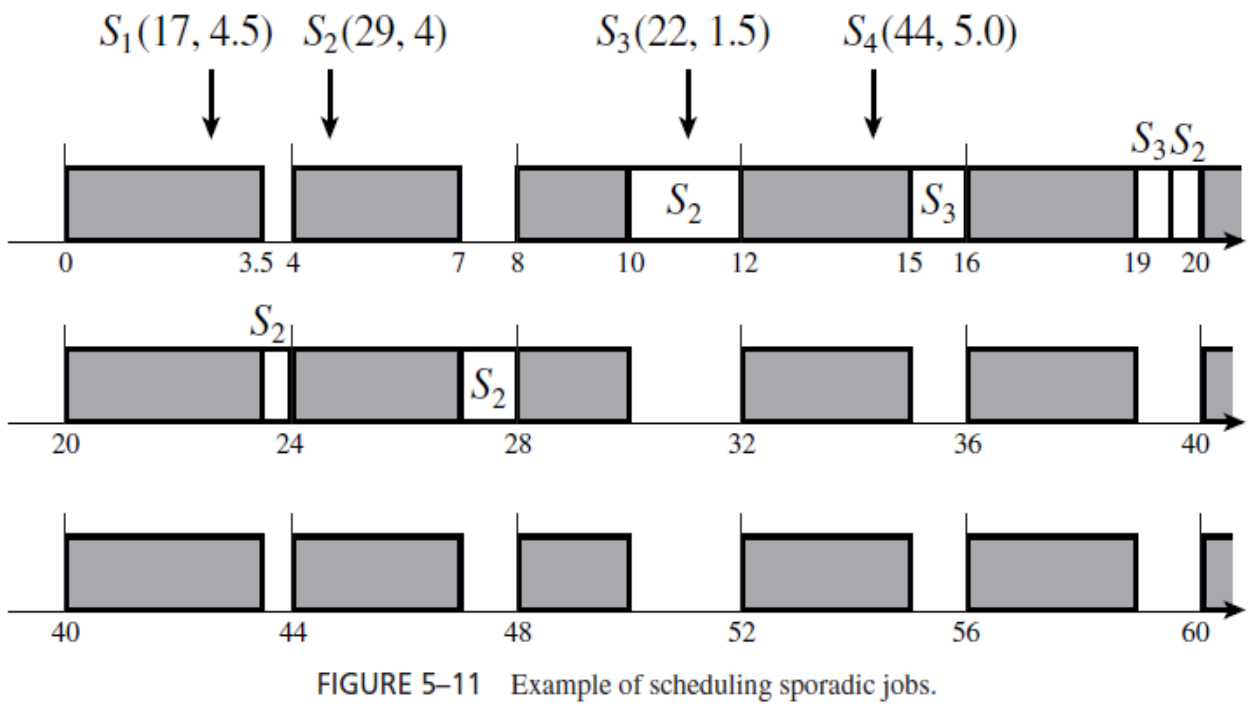


FIGURE 5–11 Example of scheduling sporadic jobs.

- For this system, F is 5.
- Only σ(i, k) for i, k = 1, 2,...,5 are stored.
- To compute the initial amount of slack time σ(3, 14) in frames 3 through 14,
  - frame 3 is in the first major cycle (i.e., j = 1),
  - frame 14 is the third major cycle (i.e., j'= 3).
- The term *i* in the above formula is equal to 3 , and σ(i, F) = σ(3, 5) = 4.
- Similarly, k = 4, and σ(1, 4) is 4.5.
- σ(1, F) is 5.5.
- σ(3, 14) = 4 + 4.5 + (3 − 1 − 1) × 5.5 = 14.

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

CPS LAB

The current total amount of slack time $\sigma_c(t, l)$ in frames t through l

- Suppose that at the beginning of current frame t, there are n_s sporadic jobs in the system.
- We call these jobs S_1, S_2,..., S_{ns} .
- d_k : the deadline of S_k
- e_k : the execution time of S_k
- ξk :the execution time of the portion of S_k 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 $\sigma_c(t, l)$ in frames t through l can be computed from the initial amount of slack time in these frames according to

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

CPS LAB

The amount of slack time $\sigma(t, l)$ before $S(d, e)$

- The sum $\sigma_c(t, l)$ 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

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

- We say that the job has σ units of slack.
- Clearly, we can accept S only if its slack σ is no less than zero.
- If S is accepted, its slack is stored for later use.

- Since the accepted jobs are scheduled on the EDF basis, the acceptance of S may cause an existing sporadic job Sk whose deadline is after d to complete too late.
- Specifically, if S is accepted, the slack σ_k of S_k is reduced by the execution time e of S.
- If the reduced amount remains to be no less than zero, S_k will not miss its deadline.
- The scheduler must consider every sporadic job that is still in the system and has a deadline after d.
- It accepts S only if the reduced slack of every such job is no less than zero.

Data maintained for the sake of acceptance tests

- The precomputed slack table whose entry $\sigma(i, k)$ gives the initial total amount of slack time in frames i through k, for every pair of $i, k = 1, 2,..., F$;
- The execution time ξ_k of the completed portion of every sporadic job S_k in the system at the beginning of the current frame t;
- The current slack σ_k of every sporadic job S_k in the system.

- Whenever a sporadic job S_k is executed in a frame, the scheduler updates ξ_k at end of the frame.
- The slack time of a sporadic job S is given by Eq. (5.6b) when the job is accepted.

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

- Later whenever a new sporadic job with an earlier deadline is accepted, the slack of S is decremented by the execution time of the new sporadic job.
- The complexity of this update operation, as well as the computation needed to accept or reject each sporadic job, is O(N_s) where N_s is the maximum number of sporadic jobs in the system.

The cyclic EDF algorithm

- At time 4, the scheduler checks whether $S_1(17, 4.5)$ can be accepted,
  - it finds that the slack $\sigma(2, 4)$ in frames before the deadline of S1 is only 4.
  - Hence, it rejects the job.

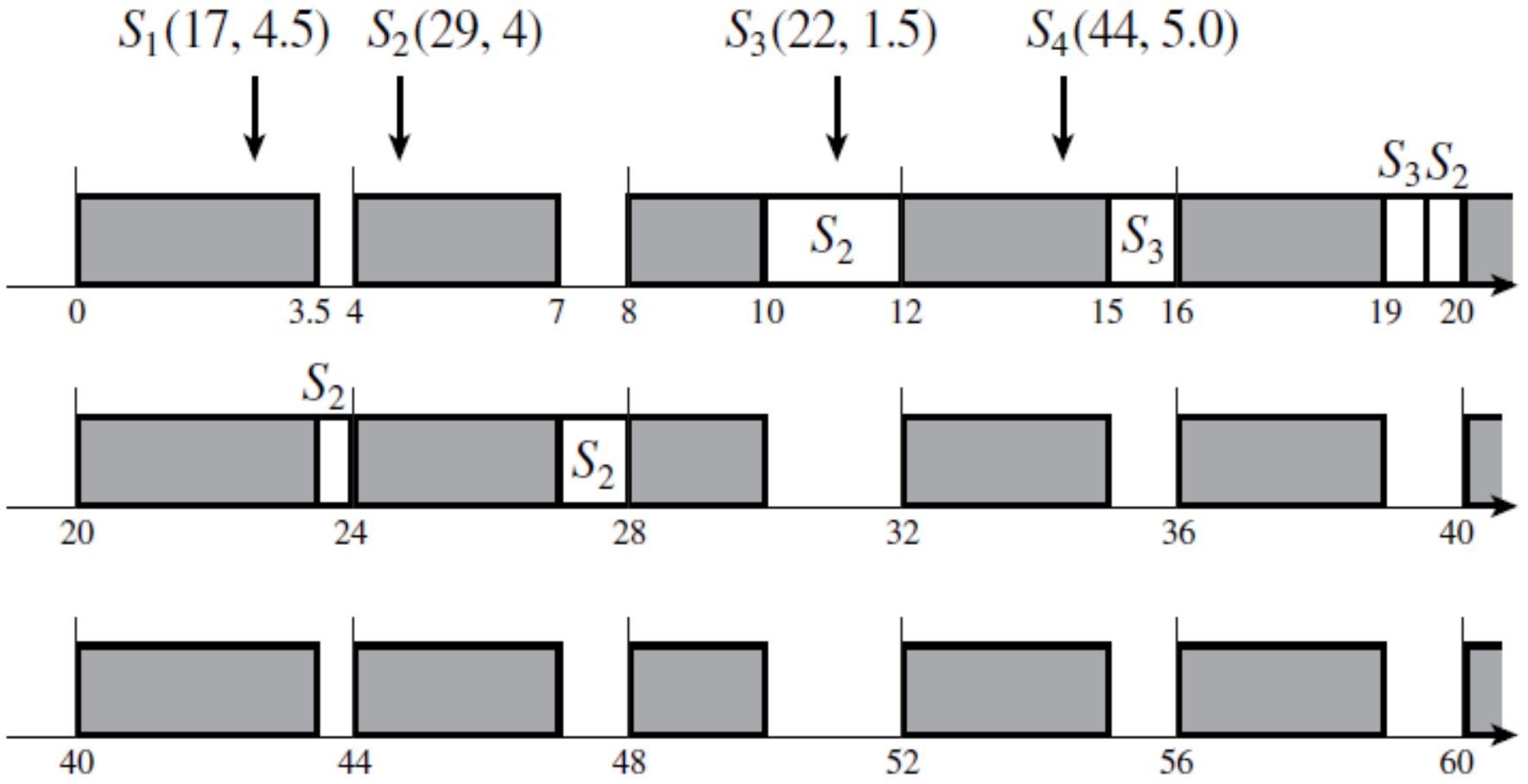


FIGURE 5–11 Example of scheduling sporadic jobs.

The cyclic EDF algorithm

- At time 8, the scheduler tests $S_2(29, 4)$,
  - the current slack $\sigma_c(3, 7)$ in frames 3 through 7 is equal to $\sigma(3, 5) + \sigma(1, 2) = 4 + 1.5 = 5.5$.
  - This amount is larger than the execution time of S_2.
  - Since there is no other sporadic job in the system, the fact that S2 passes this step suffices, and the scheduler accepts S2.
  - The slack σ_2 of S_2 is 1.5.

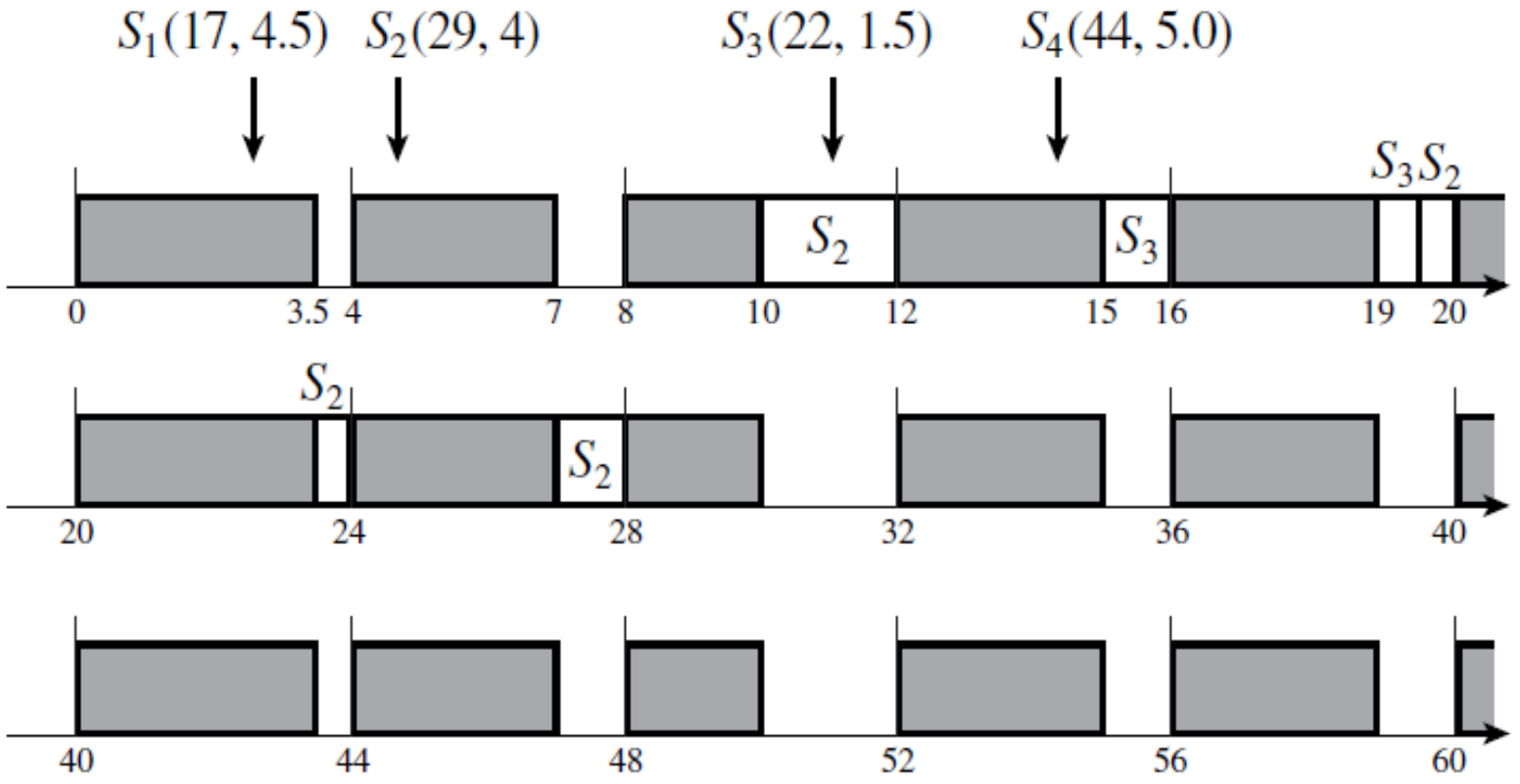


FIGURE 5–11 Example of scheduling sporadic jobs.

The cyclic EDF algorithm

- At time 12, the scheduler tests $S_3(22, 1.5)$,
  - the slack $\sigma 2$ of S_2 is still 1.5.
  - There is no sporadic job with deadline before 20, (the end of the frame before 22).
  - Hence, the sum on the right-hand side of Eq. (5.6a) is 0. $\sigma_c(t, l) = \sigma(t, l) - \sum_{d_k \le d} (e_k - \xi_k)$
  - the current slack $\sigma_c(4, 5)$ before 22 is equal to $\sigma(4, 5) = 2$.
  - Since $2 > 1.5$,
  - S_3 passes the first step of its acceptance test.
  - The acceptance of S3 would reduce σ_2 to 0.
  - Therefore, S_3 is acceptable.
  - When S_3 is accepted,
  - its slack σ_3 is 0.5, and $\sigma 2$ is 0.

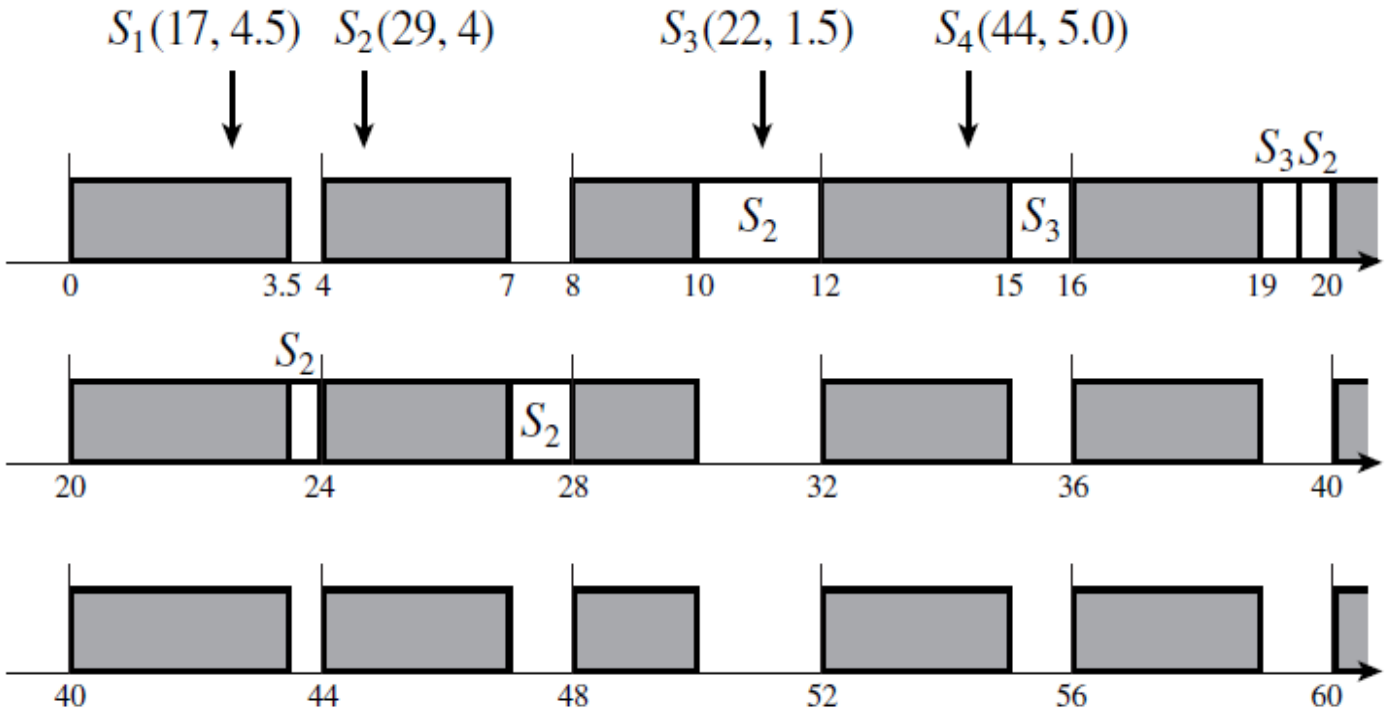


FIGURE 5–11 Example of scheduling sporadic jobs.

The cyclic EDF algorithm

- At time 16, the scheduler tests S4(44, 5.0),
  - ξ2 is 2 and
  - ξ3 is 1.0.
  - The current slack time σc(5, 11) in the frames before 44 is 7 − 2 − 0.5 = 4.5 according to Eq. (5.6a).
  - Since this slack is insufficient to complete S4 by time 44, the scheduler rejects S4.

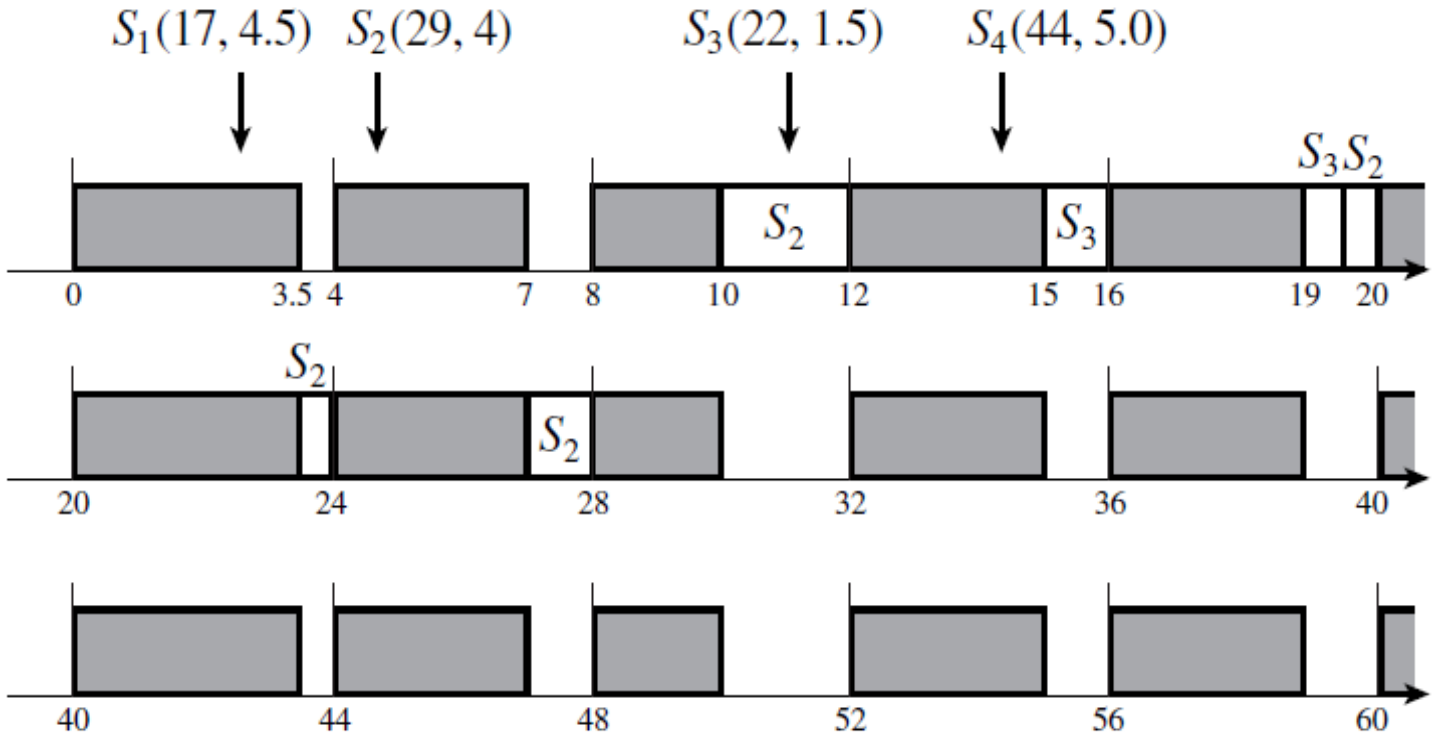


FIGURE 5–11 Example of scheduling sporadic jobs.

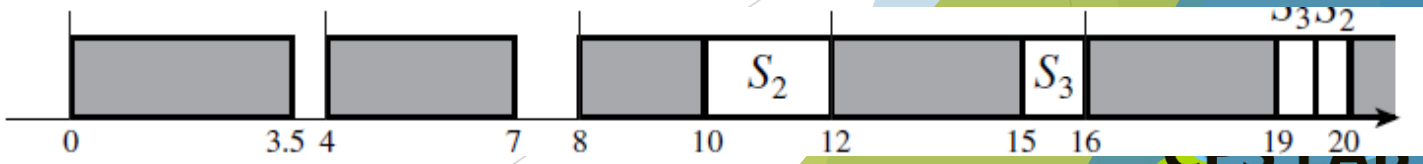
CPS LAB

5.6.4 Optimality of cyclic EDF algoritm

- Compare the cyclic EDF algorithm with 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 (i.e., all the jobs can be accepted and scheduled to complete by their deadlines) by any algorithm in this class, the EDF algorithm can always find a feasible schedule of the jobs.
- This statement follows directly from Theorem 4.1 on the optimality of the EDF algorithm.

Condition that cyclic EDF is not optimal

- However, the cyclic EDF algorithm is not optimal when compared with algorithms that perform acceptance tests at arbitrary times.
- Compared to EDF, it is better to use an interrupt-driven scheduler which does an acceptance test upon the release of each sporadic job.
- The example in Figure 5–11 illustrates this fact.
- Suppose that the scheduler were to interrupt the execution of the job slices in frame 1 and do an acceptance test at time 3 when $S_1(17, 4.5)$ is released.
- It would find an additional 0.5 units of slack in frame 1, making it possible to accept S_1.
- Because it waits until time 4 to do the acceptance test, the 0.5 unit of slack time in frame 1 is wasted.
- Consequently, it cannot accept S1.



Shortcoming of interrupt-driven scheduler

- It increases the danger for periodic-job slices to complete late.
- Because the release times of sporadic jobs are unpredictable, the execution of periodic job slices may be delayed by
  - an unpredictable number of the context switches and
  - acceptance tests in the middle of each frame.
- Because of this shortcoming, it is better to stay within the cyclic scheduling framework and make scheduling decisions, including acceptance tests, only at the beginning of the frames.

The cyclic EDF algorithm is an on-line algorithm.

- Measure the merit of an on-line algorithm by the value of the schedule produced by it: (i.e., the total execution time of all the accepted jobs)
  - 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.
- In the example in Figure 5–11, because the scheduler has already accepted and scheduled S_3 at time 12, it must reject S_4.
- The value of the schedule of these four sporadic jobs is only 5.5, instead of the maximum possible value of 9 obtained by rejecting S_3 and accepting S_4.
- If we were to modify the scheduling algorithm so that the scheduler rejects S_3, the value of the resultant schedule is again less than the maximum possible if S_4 is not released later or if its execution time is less than 1.5.

5.7 Practical considerations and generalizations

Practical problems

- How to handle frame overruns?
- How to do mode changes?
- How to schedule tasks on multiprocessor systems?

5.7.1 Handling frame overruns

- 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.

Use slack time for unfinished portion

- However, premature termination of overrun jobs may put the system in some inconsistent state, and the actions required to recover from the state and maintain system integrity may be costly.
- The unfinished portion executes as an aperiodic job during the slack time in the subsequent frame(s) or in the background whenever there is spare time.

Continue to execute the offending job.

- 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

- As stated earlier, 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.
- Periodic tasks that execute in the new mode but not in the old mode are created and added to the system.
- The periodic tasks that execute in both modes continue to execute in a timely fashion.
- When the mode change completes, the new set of periodic tasks are scheduled and executed.

Schedule mode-change job

- The work to configure the system is a mode-change job; the job is released in response to a mode change command.
- We need to consider two cases: The mode-change job has either a soft or hard deadline.

Mode change job with soft-deadline

- A soft deadline mode-change job is to treat it like an aperiodic job.
- 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.
- This scheme can be implemented by letting the scheduler or the mode-change job mark each periodic task that is to be deleted.
- During mode change, the scheduler continues to use the old schedule table.
- The scheduler 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.

How to deal with aperiodic and sporadic jobs?

- For aperiodic jobs:
  - Delay their execution until after mode change, since the deadlines of the remaining aperiodic jobs are soft, their execution can be delayed until after the mode change.
- For sporadic jobs:
  - One way to ensure their on-time completion is to defer the switchover from the old schedule table to the new schedule table until all the sporadic jobs in the system complete.
  - This option can lengthen the response time of the mode change.
  - Another option is to check whether the sporadic jobs in the system can complete in time according to the new schedule.
  - The schedule switchover is deferred only when some sporadic job cannot complete in time according to the new schedule.

A mode change task

```
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.

Mode change job with hard deadline

- There are two possible approaches to scheduling this job.
  - Treat it like an ordinary sporadic job.
- 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 the previous slide.

An alternative action using bulldozer 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.

Treat alternative action as a periodic tasks

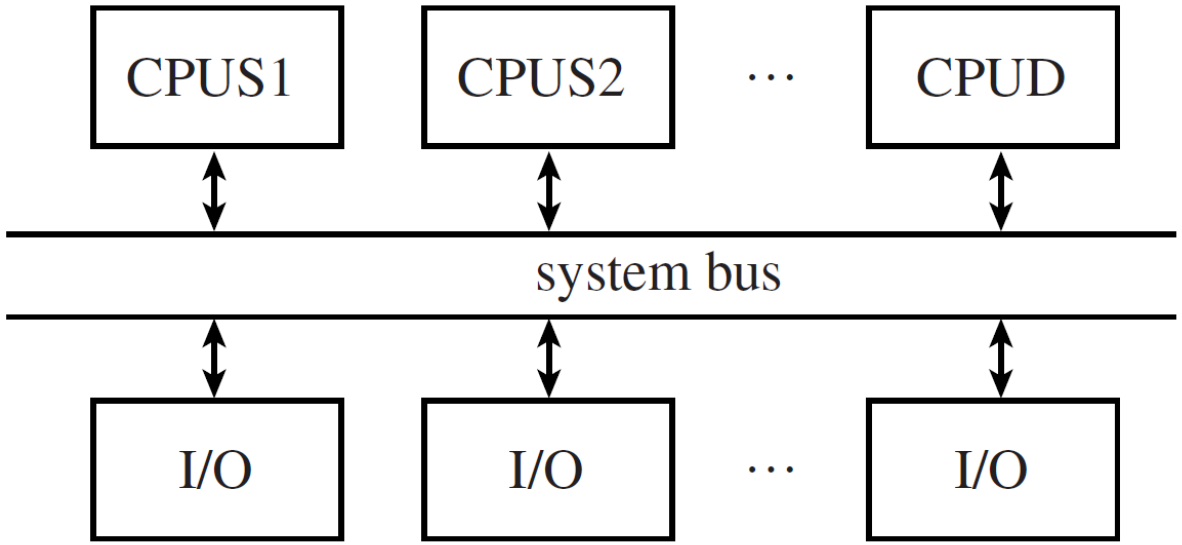
- The job that brakes in time must be admit and scheduled.
- Schedule each sporadic mode-change job that cannot be rejected periodically, with a period no greater than half the maximum allowed response time.
- The flight control system in Figure 1–3 is an example.
- The mode selection task is polled at 30 Hz.
- In this example, a mode change does not involve deletion and creation of tasks, only changes in the input data.
- The time required by the mode selection task is very small and is allocated to the task periodically.
- Leads to a reduction in the processing capacity available for periodic tasks.
- This is not a problem as long as the other periodic tasks are schedulable.

5.7.3 General Workloads and Multiprocessor Scheduling

- Whenever the workload parameters are known a priori and there is a global clock, it is conceptually straightforward to schedule tasks on several processors
- We can construct a global schedule which specifies on what processor each job executes and when the job executes.
- As long as the clock drifts on the processors are sufficiently small, we can use the uniprocessor schedulers on each processor to enforce the execution of the jobs according to the global schedule.

A precomputed multiprocessor schedule from a precomputed uniprocessor schedule.

- 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.
- The system bus is the bottleneck.
- 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.



A cyclic schedule of the data-transfer activities on the bus

- CPUS1 produces data and CPUS2 consumes the data from CPUS1.
- CPUD produces the data transferred to an I/O device.
- 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.

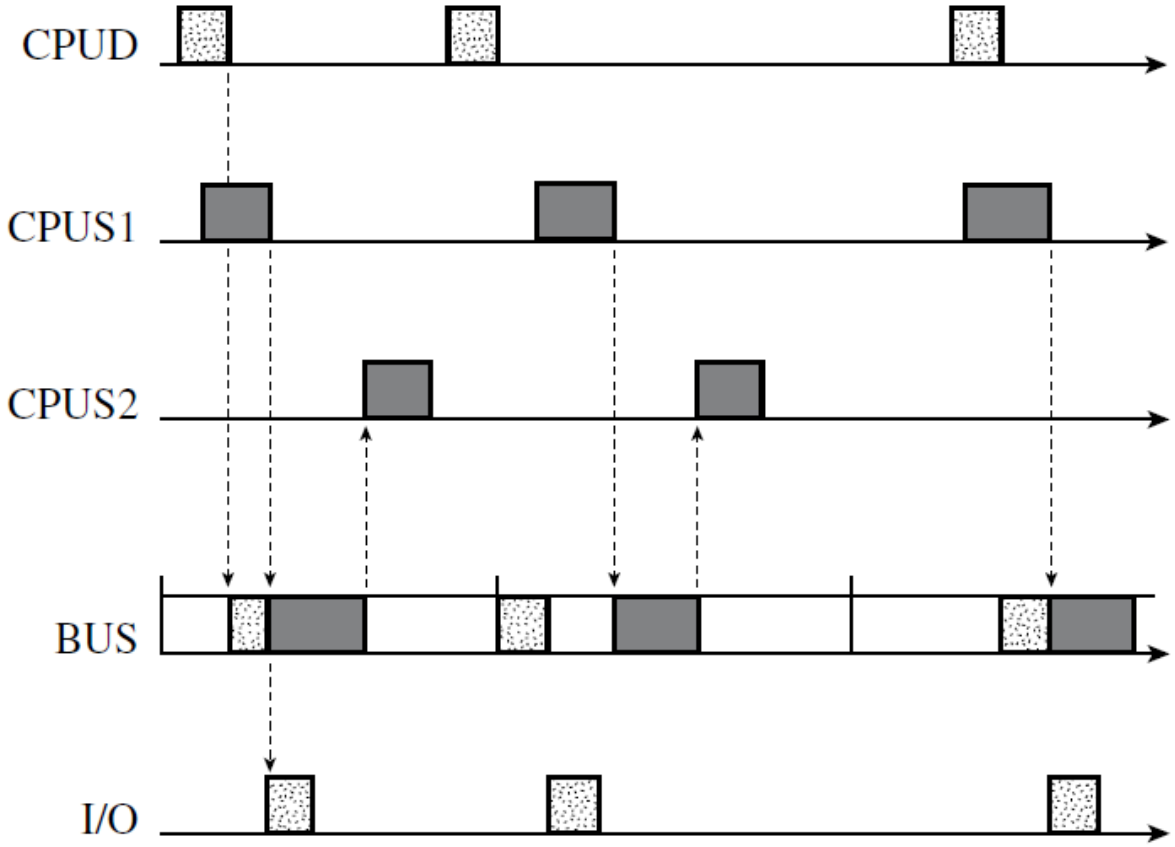


FIGURE 5–13 A simple clock-driven multiprocessor schedule.

A cyclic schedule of the data-transfer activities on the bus

- This example is based on the Boeing 777 Airplane Information Management System (AIMS) [DrHo].
- 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.
- 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.

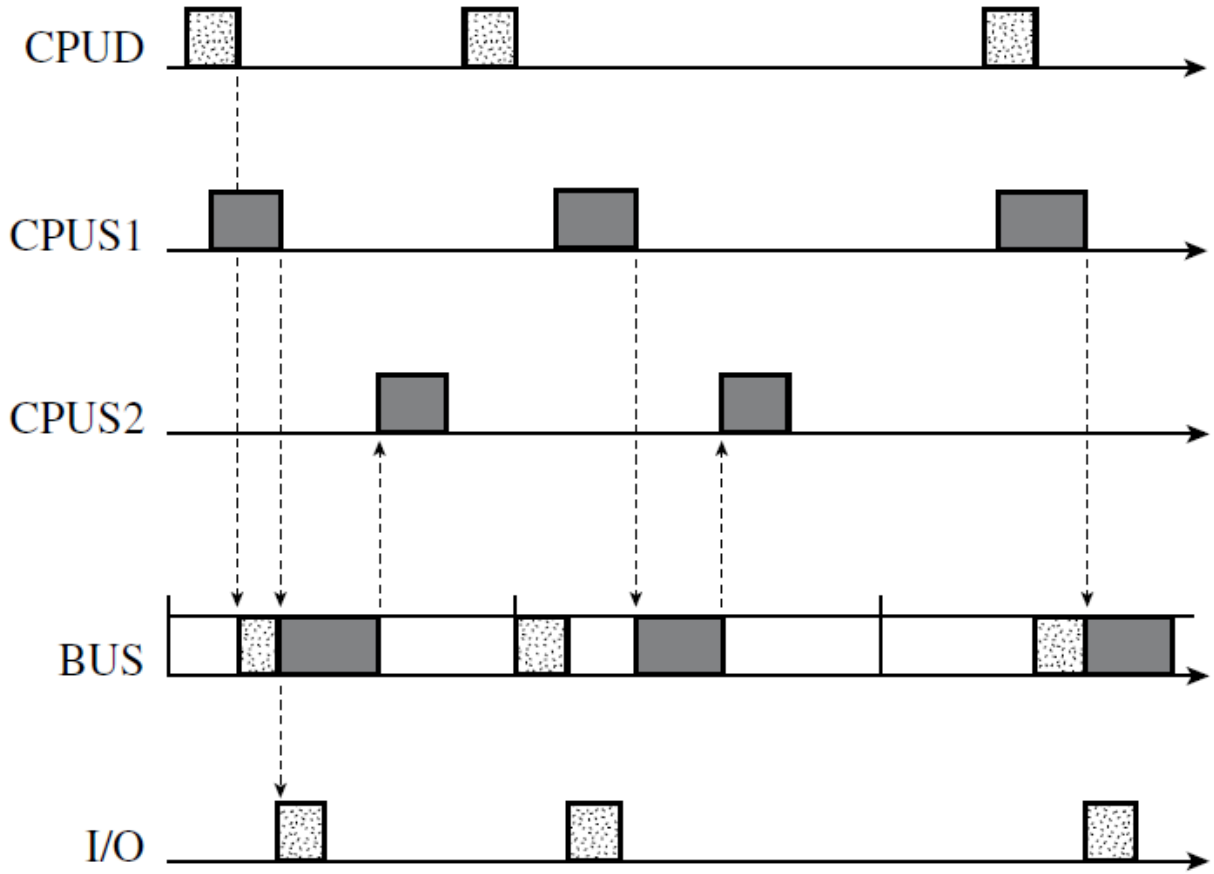


FIGURE 5–13 A simple clock-driven multiprocessor schedule.

*5.8 Algorithm for constructing static schedules

5.9 Pros and cons of clock-driven scheduling

Advantages of the clock-driven approach

- The most important advantage is simplicity.
- Complex dependencies, communication delays, and resource contentions among jobs are taken into account when construction of the static schedule,
  - ensure no deadlock and unpredictable delays.
- A stored table of start times and completion times that is interpreted by the scheduler at run time.

Minimize problems in run-time execution

- No concurrency control
  - By changing the table, jobs can be scheduled according to different static schedules in different operation modes.
- No need for synchronization mechanism
  - Precedence constraints and other types of dependency can be taken care of by the choice of the schedule.
- Satisfy tight completion-time jitter requirements.
  - Such requirements can be taken into account in the choice of the cyclic schedule.

- 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
  - so as many jobs as possible can execute from start to finish without interruption.
- This is a good scheduling strategy for applications where
  - variations in time and resource requirements are small.
- Many traditional real-time applications are examples.
  - the patient care system
  - the flight control system

Disadvantages of the clock-driven approach

- The most obvious one is that a system based on this approach is brittle(易碎的):
  - It is relatively difficult to modify and maintain.
- A new schedule construction is required when
  - changes in execution times of some tasks
  - the addition of new tasks
- Consequently, the approach is suited only for systems which are rarely modified once built.
  - e.g., small embedded controllers

Other disadvantages compared to priority driven

- In contrast to clock-driven scheduling, priority-driven algorithms do not require fixed release times.
- This relaxation of the release-time jitter requirement
  - eliminates the need for global clock synchronization
  - permits more design choices.
- As long as the inter-release times of all jobs in each periodic task are never less than the period of the task,
  - priority-driven system can guarantee the timely completion of every job.
- Many applications cannot use clock driven scheduling.
  - must be reconfigurable on-line
  - the mix of periodic tasks cannot be predicted in advance.
- Priority-driven system does not have this restriction.

5.10 Summary

Periodic jobs

- Clock-driven schedulers schedules periodic tasks according to cyclic schedule.
- Tasks information must be known a priori.
  - the number of periodic tasks
  - the parameters of the tasks
- Cyclic schedule
  - computed off-line
  - stored as a table for use by the scheduler at run time

Cyclic schedule & frame

- Cyclic schedules satisfies the frame size constraints (5.1), (5.2), and (5.3).

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

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

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

- Scheduling decisions are made at the beginning of each frame.
- At the beginning of each frame *t*, the scheduler verifies that all jobs
  - whose deadlines are earlier than *t* have indeed completed
  - which are scheduled in the current frame *t* have indeed been released
- The scheduler takes appropriate recovery action if any of these conditions are not satisfied.

CPS LAB

Aperiodic jobs

- Aperiodic jobs can be scheduled in the background after the periodic jobs scheduled in each frame have completed.
- Improve the average response time of aperiodic jobs by scheduling them ahead of the periodic job slices for as long as there is slack in the frame.

Sporadic jobs

- Similarly, sporadic jobs are also scheduled during the slack times not used by periodic job slices in the frames.
- A sporadic job *S* is accepted only when
  - the total slack time not used by periodic job slices and existing sporadic jobs in frames before its deadline is no less than the execution time of the sporadic job *S*
  - the slack of every existing sporadic job which has a later deadline than *S* is no less than execution time of *S*
- Once a sporadic job is accepted, it is scheduled among all sporadic jobs in the EDF order during the time intervals not used by periodic jobs slices.

End