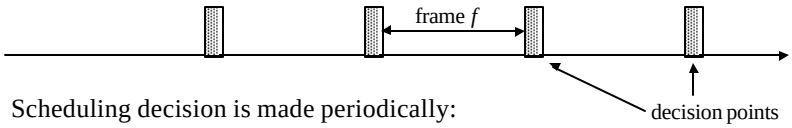
## Clock-Driven Scheduling (in-depth)

- Precompute static schedule off-line (e.g. at design time): can afford expensive algorithms.
- Idle times can be used for aperiodic jobs.
- Possible implementation:
  **Table-driven**
- Scheduling table has entries of type $(t_k, J(t_k))$ , where
  $t_k$ : decision time
  $J(t_k)$: job to start at time $t_k$
- Input: Schedule $(t_k, J(t_k))$
  $k = 0,1,\ldots,N\text{-}1$

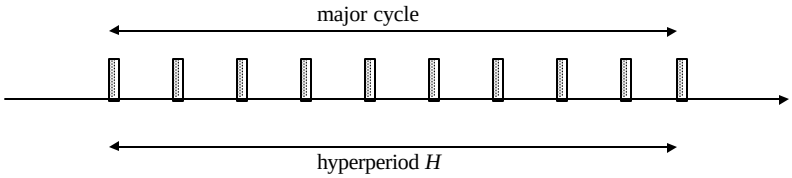
```
Task Scheduler:
 i := 0; k := 0;
 <set timer to expire at time t_0>
 BEGIN LOOP
   <wait for timer interrupt>
   i := i+1;
   k:= i mod N;
   <set timer to expire at time
    (i DIV N)*H + t_k >
   IF J(t_k-1)is empty
     THEN wakeup(aperiodic)
     ELSE wakeup(J(t_k-1))
 END LOOP
END Scheduler;
```

## Cyclic Schedules: General Structure

- Scheduling decision is made periodically:




- Scheduling decision is made periodically:
  – choose which job to execute
  – perform monitoring and enforcement operations
- **Major Cycle**: Frames in a hyperperiod.



## Frame Size Constraints

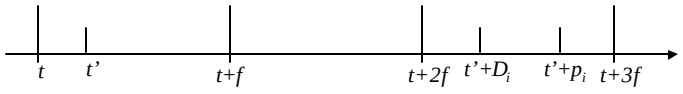
- Frames must be sufficiently long so that every job can start and complete within a single frame:

  (1) $f \geq \max(e_i)$

- The hyperperiod must have an integer number of frames:

  (2) $f \mid H \qquad (f \text{ "divides" } H)$

- For monitoring purposes, frames must be sufficiently small that between release time and deadline of every job there is at least one frame:




$$2f - (t' - t) \leq D_i$$

$$t' - t \geq \gcd(p_i, f)$$

(3) $2f - \gcd(p_i, f) \leq D_i$

## Frame Sizes: Example

- Task set:

| | | $p_i$ | $e_i$ | $D_i$ | |
|---|---|---|---|---|---|
| $T_1$ | = | 15, | 1, | 14 | |
| $T_1$ | = | 20, | 2, | 26 | $H = 660$ |
| $T_1$ | = | 22, | 3, | 22 | |

(1) $\forall i : f \geq e_i \quad \Rightarrow f \geq 3$

(2) $f \mid H \quad \Rightarrow f = 2,3,4,5,6,10,..$

(3) $\forall i : 2f - \gcd(p_i, f) \leq D_i \quad \Rightarrow f = 2,3,4,5,6$

# Slicing and Scheduling Blocks

- Slicing

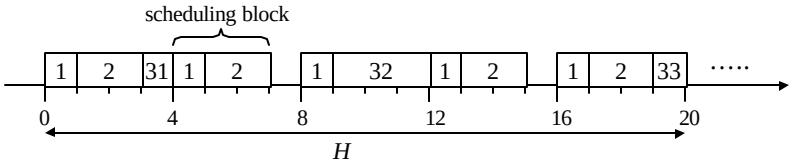
$$\begin{array}{lcl} & & p_i \quad e_i \quad D_i \\ T_1 & = & (\ 4,\ 1,\ 4\ ) \\ T_2 & = & (\ 5,\ 2,\ 5\ ) \\ T_3 & = & (\ 20,\ 5,\ 20\ ) \end{array}$$

$$\left.\begin{array}{l} (1) \Rightarrow f \geq 5 \\ (3) \Rightarrow f \leq 4 \end{array}\right\} ?!$$

slice $T_3$

$$\begin{array}{lcl} T_1 & = & (\ 4,\ 1,\ 4\ ) \\ T_2 & = & (\ 5,\ 2,\ 5\ ) \\ T_{31} & = & (\ 20,\ 1,\ 20\ ) \\ T_{32} & = & (\ 20,\ 3,\ 20\ ) \\ T_{33} & = & (\ 20,\ 1,\ 20\ ) \end{array}$$

$$\left.\begin{array}{l} (1) \Rightarrow f \geq 3 \\ (3) \Rightarrow f \leq 4 \end{array}\right\} f = 4$$




# Cyclic Executive

Input: Stored schedule: L(k) for k = 0,1,…,F-1;
Aperiodic job queue.

```
TASK CYCLIC_EXECUTIVE:
  k = 0;   /* current frame */
  BEGIN LOOP
    accept clock interrupt at time k*f;
    IF <the last job is not completed> take action;
    CurrentBlock := L(k);
    k            := k+1 mod F;
    IF <any slice in CurrentBlock is not released> take action;
    WHILE <CurrentBlock is not empty>
      execute the first slice in it;
      remove the first slice from CurrentBlock;
    END WHILE;
    WHILE <the aperiodic job queue is not empty>
      execute the first job in the queue;
      remove the just completed job;
    END WHILE;
  END LOOP;
END CYCLIC_EXECUTIVE;
```

## What About Aperiodic Jobs?

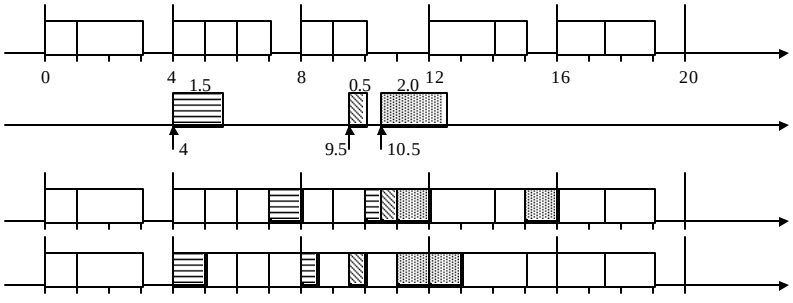
- Typically:
  - Scheduled in the background.
  - Their execution may be delayed.
- But:
  - Aperiodic jobs are typically results of external events.
- Therefore:
  - The sooner the completion time, the more responsive the system
  - Minimizing response time of aperiodic jobs becomes a design issue.
- Approach:
  - Execute aperiodic jobs ahead of periodic jobs whenever possible.
  - This is called **Slack Stealing**.

## Slack Stealing (Lehoczky *et al.*, RTSS'87)

$x_k$ Amount of time allocated to slices executed during frame $F_k$.

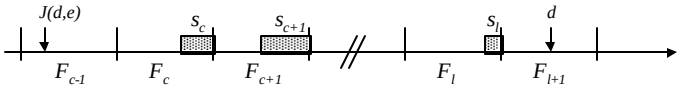
$s_k$ Slack during frame $F_k$: $s_k := f - x_k$.

- The cyclic executive can execute aperiodic jobs for $s_k$ amount of time without causing jobs to miss deadlines.
- Example:



## Sporadic Jobs

- Reminder: Sporadic jobs have hard deadlines; the release time and the execution time are not known *a priori*.
  Worst-case execution time known when job is released.
- Need **acceptance test**:




$S(c,l) = \sum_{i=c}^{l} s_i$ : Total amount of slack in Frames $F_c$, …, $F_l$.

- Acceptance Test:

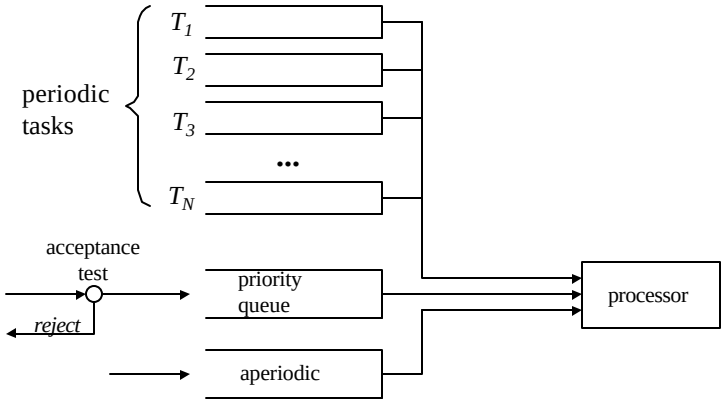
```
IF S(c,l) < e THEN
  reject job;
ELSE
  accept job;
  schedule execution;
END;
```

how?!

## Scheduling of Accepted Jobs

- Static scheduling:
  - Schedule as large a slice of the accepted job as possible in the current frame.
  - Schedule remaining portions as late as possible.
- Mechanism:
  - Append slices of accepted job to list of periodic-task slices in frames where they are scheduled.
- Problem: Early commit.
- Alternatives:
  - Rescheduling upon arrival.
  - Priority-driven scheduling of sporadic jobs.

## EDF-Scheduling of Accepted Jobs




## Acceptance Test for EDF-Scheduled Sporadic Jobs

- Sporadic Job $J$ with deadline $d$ arrives:
- Test 1: Test whether current amount of slack before $d$ is enough to accommodate $J$.
  If not, reject!
- Test 2: Test whether sporadic jobs still in system with deadlines after $d$ will miss deadline if $J$ is accepted.
  If yes, reject!
- Accept!

- (*) Define $S(J_i)$ : Amount of slack up to time $d_i$ after $J_i$ has been scheduled.
- (**) Update all $S(J_i)$ with $d_i > d$ , that is,

## Pros and Cons of Clock-Driven Scheduling

- Pros:
  - Conceptual simplicity
  - Timing constraints can be checked and enforced at frame boundaries.
  - Preemption cost can be kept small by having appropriate frame sizes.
  - Easy to validate: Execution times of slices known *a priori.*
- Cons:
  - Difficult to maintain.
  - Does not allow to integrate hard and soft deadlines.