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 \((t_k, J(t_k))\) for \(k = 0, 1, \ldots, N-1\)

```cpp
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 \((i \div N) \times H + t_k\)>
    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:
  - Frames in a hyperperiod.
  - Decision points

- 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:
  \[ f \geq \max(e_i) \]
- The hyperperiod must have an integer number of frames:
  \[ f \mid H \quad (f'' \text{ divides } H) \]
- For monitoring purposes, frames must be sufficiently small that between release time and deadline of every job there is at least one frame:
  \[
  \begin{align*}
  2f - (t' - t) & \leq D_i \\
  t' - t & \geq \gcd(p_i, f) \\
  2f - \gcd(p_i, f) & \leq D_i
  \end{align*}
  \]

Frame Sizes: Example

- Task set:
  
  \[
  \begin{array}{ccc}
  p_i & e_i & D_i \\
  T_1 & = & (15, 1, 14) \\
  T_2 & = & (20, 2, 26) \quad H = 660 \\
  T_3 & = & (22, 3, 22)
  \end{array}
  \]

- For values possible:

  \[
  \begin{align*}
  (1) & \quad \forall i : f \geq e_i & \Rightarrow f \geq 3 \\
  (2) & \quad f \mid H & \Rightarrow f = 2, 3, 4, 5, 6, 10, \ldots \\
  (3) & \quad \forall i : 2f - \gcd(p_i, f) \leq D_i & \Rightarrow f = 2, 3, 4, 5, 6
  \end{align*}
  \]
Slicing and Scheduling Blocks

- Slicing
  \[ p_i, e_i, D_i \]
  \[ T_1 = (4, 1, 4) \] \[ (1) \Rightarrow f \geq 5 \] \[ T_2 = (5, 2, 5) \] \[ (3) \Rightarrow f \leq 4 \]
  \[ T_3 = (20, 5, 20) \]

- Slice \( T_j \)
  \[ T_1 = (4, 1, 4) \]
  \[ T_2 = (5, 2, 5) \]
  \[ T_{31} = (20, 1, 20) \] \[ (1) \Rightarrow f \geq 3 \]
  \[ T_{32} = (20, 3, 20) \] \[ (3) \Rightarrow f \leq 4 \]
  \[ f = 4 \]
  \[ T_{33} = (20, 1, 20) \]

Cyclic Executive

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

**TASK CYCLIC_EXECUTIVE:**
\[ k = 0; \] /* current frame */
BEGIN LOOP
  accept clock interrupt at time \( k\cdot f \);
  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)

\[
x_k \quad \text{Amount of time allocated to slices executed during frame } F_k
\]

\[
s_k \quad \text{Slack during frame } F_k \quad s_k := f - x_k.
\]

• 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 \textit{a priori}.
  - Worst-case execution time known when job is released.

- Need \textbf{acceptance test}:
  \[
  S(c, l) = \sum_{j=c}^{l} s_j
  \]
  - Total amount of slack in Frames $F_c$, ..., $F_l$.

- Acceptance Test:
  \[
  \begin{align*}
  &\text{IF } S(c, l) < e \text{ THEN } \\
  &\hspace{1cm} \text{reject job;} \\
  &\text{ELSE } \\
  &\hspace{1cm} \text{accept job;} \\
  &\hspace{1.5cm} \text{schedule execution;} \\
  &\text{END;}
  \end{align*}
  \]

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 \textit{a priori}.

- **Cons:**
  - Difficult to maintain.
  - Does not allow to integrate hard and soft deadlines.