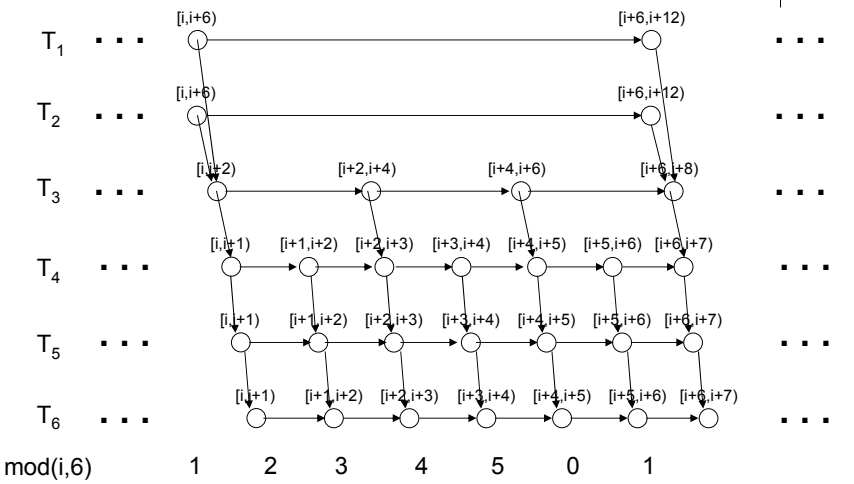
Precedence Graphs Revisited (Again)



Clock-driven Scheduling

- This approach is applicable only when the system is mostly deterministic, except for a few aperiodic and sporadic jobs
- This lecture will assume a restricted periodic task model
  1. There are ***n*** periodic tasks in the system; as long as the system remains in a particular operation mode, ***n*** is fixed
  2. The parameters of all period tasks are known a priori; in particular, variations in the inter-release times of jobs in a periodic tasks are negligibly small – i.e. 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 $r_{i,k}$

Clock-driven Scheduling

- 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 $(\varphi_i, p_i, e_i, D_i)$
- Default phase of a task is 0, default relative deadline is its period; omit elements of the tuple that have default values
- Examples:

Tuple	φ	p	e	D
(1, 10, 3, 6)	1	10	3	6
(10,3,6)	0	10	3	6
(10,3)	0	10	3	10

Clock-driven Scheduling

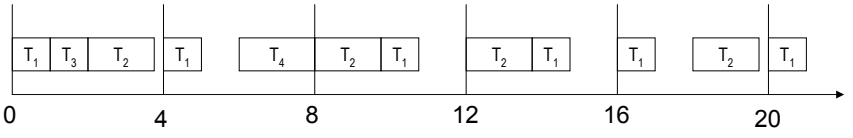
- Assume there are aperiodic jobs released at unexpected time instants
- For now, assume no sporadic jobs (recall that sporadic jobs have hard deadlines, aperiodic jobs do NOT have hard deadlines)
- Assume that the system maintains a single queue for aperiodic jobs; addition of such jobs to the queue does not require attention of the scheduler; whenever the processor is available for aperiodic jobs, the job at the head of this queue is executed

Static, Timer-driven Scheduler

- Since the parameters of jobs with hard deadlines are known before the system begins to execute, construct a static schedule of the jobs off-line; periodic static schedule == ***cyclic schedule***
- The amount of processor time allocated to every job is equal to its maximum execution time
- The static schedule guarantees that each job completes by its deadline
- The scheduler dispatches jobs according to the static schedule; as long as no job ever overruns, all deadlines are met
- Since the schedule is calculated off-line, we can employ complex, sophisticated algorithms; in particular, we can choose a feasible schedule from all possible feasible schedules that optimizes some characteristic of the system (e.g. the idle periods for the processor are nearly periodic to accommodate aperiodic jobs)

Static, Timer-driven Scheduler

- Example - consider a system with 4 independent periodic tasks: T_1 = (4, 1), T_2 = (5, 1.8), T_3 = (20, 1), and T_4 = (20, 2)
- Hyperperiod H is 20 (least common multiple of 4, 5, 20, 20)
- T1 starts execution at 0, 4, 9.8, 13.8, 16
- T2 starts execution at 2, 8, 12, 18
- T3 starts execution at 1
- T4 starts execution at 6
- Idle intervals: (3.8,4), (5,6), (10.8,12), (14.8,16), (17,18), (19.8,20)



Static, Timer-driven Scheduler

- Can execute aperiodic jobs during the idle intervals
- May be advantageous to scatter the unused intervals somewhat periodically in the schedule
- If no aperiodic jobs ready to execute during such intervals, can execute background non-realtime jobs

Static, Timer-driven Scheduler

- Implementing such a scheduler
  - Store the pre-computed schedule as a table
  - Each entry (t_k, $T(t_k)$), where t_k is a decision time (a time instant when a scheduling decision is to be made) and $T(t_k)$ is either the name of the task to start at that time or I (for idle)
  - During initialization, the system creates all the tasks that are to be executed (allocates sufficient memory for the code and data of every task and brings the code executed by the task into memory)
  - After initialization is complete, the scheduler sets the hardware timer to interrupt at the first decision time
  - Upon receipt of an interrupt at t_k, the scheduler sets the timer to expire at t_{k+1} and prepares the task $T(t_k)$ for execution; it then suspends itself, letting the task have the processor to execute

Static, Timer-driven 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 + tk;
                if the current task T is I,
                        let the job at the head of the aperiodic queue execute;
                else,
                        let the task T execute;
                sleep;
        end do.
End SCHEDULER.
```


General structure of cyclic schedules

- General consensus is that it is better to use cyclic schedules that exhibit some structure, as opposed to using totally *ad hoc* schedules – e.g. make scheduling decisions at periodic intervals, rather than at arbitrary times
- Such periodic scheduling intervals partition the timeline into frames; there is no pre-emption within frames
- The phase of each periodic task is a non-negative integer multiple of the frame size – i.e. the first job of every task is released at the beginning of a frame

Frame size constraints

- Want the frames to be sufficiently long so that every job and start and complete its execution within a frame
  - [eq 1] $f \geq \max(e_1, e_2, \ldots, e_n)$
- To minimize the length of the cyclic schedule, the frame size should divide the hyperperiod of the system; this is true if f divides the period p_i of at least one task T_i
  - [eq 2] $\exists$ i such that $\text{mod}(p_i, f) = 0$
- To make it possible for the scheduler to determine that every job completes by its deadline, the frame size should be sufficiently small such that between the release time and deadline of every job, there is at least one frame
  - [eq 3] $2*f - \gcd(p_i, f) \leq D_i$ for i = 1, 2, …, n

Frame size constraints

- Example - reconsider the system with 4 independent periodic tasks: $T_1 = (4, 1)$, $T_2 = (5, 1.8)$, $T_3 = (20, 1)$, and $T_4 = (20, 2)$
- Hyperperiod H is 20 (least common multiple of 4, 5, 20, 20)
  - Eq 1 $\Rightarrow f \geq 2$
  - Eq 2 $\Rightarrow f \in \{ 2, 4, 5, 10, 20 \}$
  - Eq 3 $\Rightarrow f = 2$
- Therefore, f = 2

Job Slices

- Sometimes, the task parameters for a system cannot meet all three frame size constraints simultaneously
- Forced to partition a large-execution-time job in a task into slices/subjobs with smaller execution times
- To construct a cyclic schedule, we need to make three kinds of design decisions:
  - Choose a frame size
  - Partition jobs into slices
  - Place slices in frames
- These decisions cannot be taken independently
- Sometimes we need to partition jobs into more slices than required by the frame size constraints in order to yield a feasible schedule

Cyclic Executives

- A table-driven cyclic scheduler for all types of jobs in a multi-threaded system
- Table that drives the scheduler has F entries, where F = H / f; each corresponding entry L(k) lists the names of the job slices that are scheduled to execute in frame k; called a scheduling block
- Cyclic executive takes over the processor and executes at the clock interrupt that signals the start of a frame
  - It determines the appropriate scheduling block for this frame
  - It executes the jobs in the scheduling block
  - It wakes up jobs in the aperiodic job queue to permit them to use the remaining time in the frame
- Major assumptions:
  - Existence of a timer
  - Each timer interrupt is handled by the executive in a bounded time

Scheduling aperiodic jobs

- Thus far, aperiodic jobs are scheduled in the background after all the job slices with hard deadlines scheduled in each frame have completed
- Delaying the execution, and hence the completion, of aperiodic jobs in preference to periodic tasks is not necessarily a good one
- There is no advantage to completing a hard RT job early
- Since an aperiodic job is released due to an event, the sooner such a job completes, the more responsive the system
- Minimizing the response times of aperiodic jobs is typically a design goal of real-time schedulers

Slack Stealing

- Execute aperiodic jobs ahead of periodic jobs whenever possible – in essence, place the “idle” periods at the beginning of each frame, rather than at the end
- Every periodic job slice must be scheduled in a frame that ends no later than its deadline
- When an aperiodic job executes ahead of slices of periodic tasks, it consumes the slack in the frame
- The cyclic executive must keep a running tally of the slack left in each frame
- As long as there is slack remaining in a frame, the cyclic executive returns to examine the aperiodic job queue after each slice completes
- Implementing such an executive:
  - Initial slack in each frame precomputed and stored in table
  - Use an interval timer to keep track of the available slack in the frame
  - At beginning of frame, interval timer set to available slack
  - Timer counts down whenever an aperiodic job executes in the frame
  - When timer expires, executive is interrupted, and preempts the executing aperiodic job

Scheduling sporadic jobs

- Sporadic jobs have hard deadlines
- Since their minimum release times and maximum execution times are not known *a priori*, impossible to guarantee that all sporadic jobs complete in time
- Have scheduler perform an acceptance test when each sporadic job is released
  - Check whether the newly released sporadic job can be feasibly scheduled with all the jobs in the system at that time
    - 'Jobs in the system' defined as periodic jobs and sporadic jobs that have been accepted but not yet completed
    - If there is sufficient slack in the frames before the new job's deadline, the new job is accepted; otherwise, it is rejected
- If more than one sporadic job is queued for acceptance test, EDF can be used for ordering this queue

Pros of Clock-driven Scheduling

- Conceptual simplicity
  - Ability to consider complex dependencies, communication delays, and resource contention among jobs when constructing the static schedule, guaranteeing absence of deadlocks and unpredictable delays
  - Entire schedule is captured in a static table
  - Different operating modes can be represented by different tables
  - No concurrency control or synchronization required
  - If completion time jitter requirements exist, these can be captured in the static schedule[s]

Pros of Clock-driven Scheduling

- When the workload is mostly periodic and the schedule is cyclic, timing constraints can be checked and enforced at each frame boundary
- Choice of frame size can minimize context switching and communication overheads
- Such systems are relatively easy to validate, test and certify

Cons of Clock-driven Scheduling

- Such systems are inflexible – precompilation of knowledge into the scheduling tables means that if anything changes materially, have to redo the table generation – as a result, best suited for systems which are rarely modified once built
- Other disadvantages:
  1. Release times of all jobs must be fixed
  2. All combinations of periodic tasks that might execute at the same time must be known *a priori* so that the combined schedule can be precomputed
  3. The treatment of aperiodic jobs is very primitive; if there is a significant amount of soft-real-time computation in the system, it is unlikely that this structure will yield acceptable response times