Real-Time Scheduling

Scheduling of Reactive Systems

[Some parts of this lecture are based on a real-time systems course of Colin Perkins

http://csperkins.org/teaching/rtes/index.html]

Reminder of Basic Notions

- Jobs are executed on processors and need resources
- Parameters of jobs
  - temporal:
    - release time – r_i
    - execution time – e_i
    - absolute deadline – d_i
    - derived params: relative deadline (D_i), completion time, response time, ...
  - functional:
    - laxity type: hard vs soft
    - preemptability
  - interconnection
    - precedence constraints (independence)
  - resource
    - what resources and when are used by the job
- Tasks = sets of jobs

Reminder of Basic Notions

- Schedule assigns, in every time instant, processors and resources to jobs
- valid schedule = correct (common sense)
- Feasible schedule = valid and all hard real-time tasks meet deadlines
- Set of jobs is schedulable if there is a feasible schedule for it

- Scheduling algorithm computes a schedule for a set of jobs
- Scheduling algorithm is optimal if it always produces a feasible schedule whenever such a schedule exists, and if a cost function is given, minimizes the cost

We have considered scheduling of individual jobs

Scheduling Reactive Systems

From this point on we concentrate on reactive systems

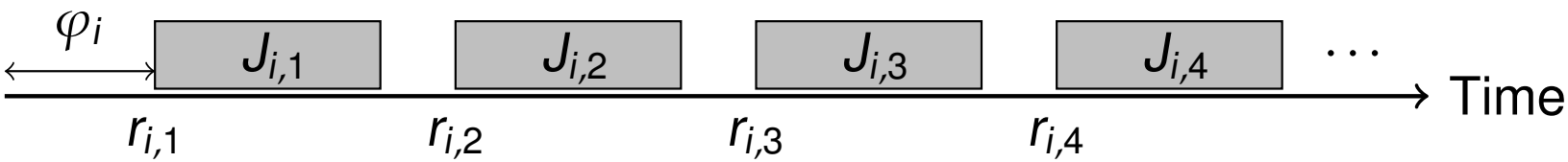
i.e. systems that run for unlimited amount of time

Recall that a task is a set of related jobs that jointly provide some system function.

- We consider various types of tasks
  - Periodic
  - Aperiodic
  - Sporadic
- Differ in execution time patterns for jobs in the tasks
- Must be modeled differently
  - Differing scheduling algorithms
  - Differing impact on system performance
  - Differing constraints on scheduling

Periodic Tasks

- A set of jobs that are executed repeatedly at regular time intervals can be modeled as a *periodic task*



- Each periodic task T_i is a sequence of jobs $J_{i,1}, J_{i,2}, \ldots J_{i,n}, \ldots$
  - The *phase* φ_i of a task T_i is *the release time* $r_{i,1}$ *of the first job* $J_{i,1}$ in the task T_i ;
    tasks are *in phase* if their phases are equal
  - The *period* p_i of a task T_i is *the minimum length of all time intervals between release times of consecutive jobs* in T_i
  - The *execution time* e_i of a task T_i is *the maximum execution time of all jobs* in T_i
  - The *relative deadline* D_i is relative deadline of all jobs in T_i

(The period and execution time of every periodic task in the system are known with reasonable accuracy at all times)

Periodic Tasks – Notation

The 4-tuple $T_i = (\varphi_i, p_i, e_i, D_i)$ refers to a periodic task T_i with phase φ_i, period p_i, execution time e_i, and relative deadline D_i

For example: jobs of $T_1 = (1, 10, 3, 6)$ are

- released at times 1, 11, 21, ...,
- execute for 3 time units,
- have to be finished in 6 time units (the first by 7, the second by 17, ...)

Default phase of T_i is $\varphi_i = 0$ and default relative deadline is $d_i = p_i$

$T_2 = (10, 3, 6)$ satisfies $\varphi = 0$, $p_i = 10$, $e_i = 3$, $D_i = 6$, i.e. jobs of T_2 are

- released at times 0, 10, 20, ...,
- execute for 3 time units,
- have to be finished in 6 time units (the first by 6, the second by 16, ...)

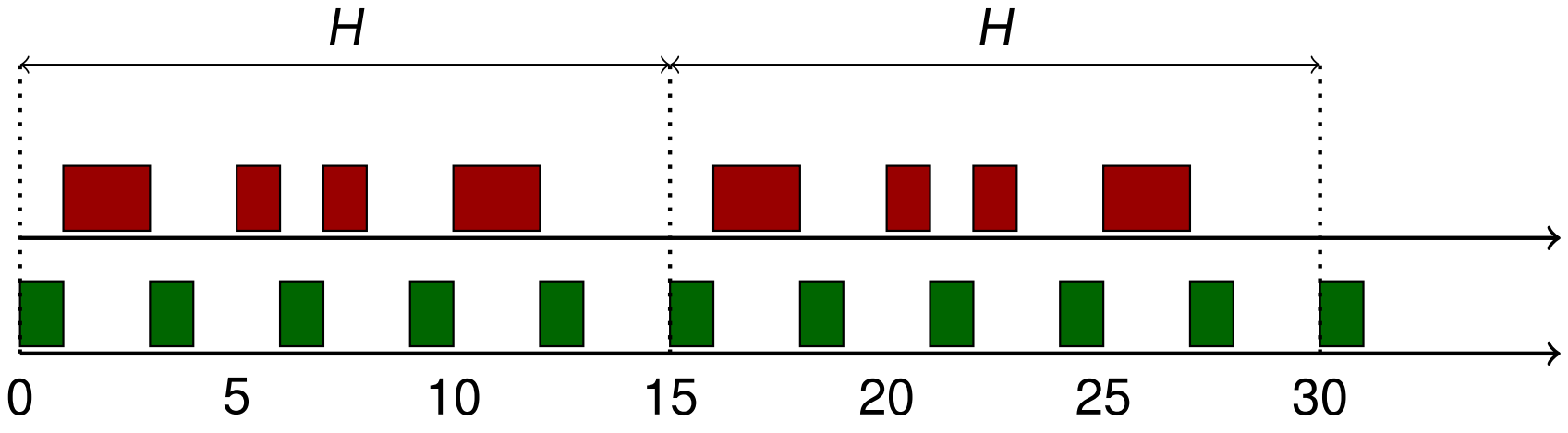
$T_3 = (10, 3)$ satisfies $\varphi = 0$, $p_i = 10$, $e_i = 3$, $D_i = 10$, i.e. jobs of T_3 are

- released at times 0, 10, 20, ...,
- execute for 3 time units,
- have to be finished in 10 time units (the first by 10, the second by 20, ...)

Periodic Tasks – Hyperperiod

The *hyper-period H* of a set of periodic tasks is the least common multiple of their periods

If tasks are in phase, then *H* is the time instant after which the pattern of job release/execution times starts to repeat



Aperiodic and Sporadic Tasks

- Many real-time systems are required to respond to external events
- The tasks resulting from such events are *sporadic* and *aperiodic* tasks
  - Sporadic tasks – hard deadlines of jobs
    e.g. autopilot on/off in aircraft
  - Aperiodic tasks – soft deadlines of jobs
    e.g. sensitivity adjustment of radar surveilance system
- Inter-arrival times between consecutive jobs are identically and independently distributed according to a probability distribution $A(x)$
- Execution times of jobs are identically and independently distributed according to a probability distribution $B(x)$

- In the case of sporadic tasks, the usual goal is to decide, whether a newly released job can be feasibly scheduled with the remaining jobs in the system
- In the case of aperiodic tasks, the usual goal is to minimize the average response time

Scheduling – Classification of Algorithms

- Static vs Dynamic
  - Static – decisions based on fixed parameters assigned to tasks/jobs before their activation
  - Dynamic – decisions based on dynamic parameters that may change during computation
- Off-line vs Online
  - Off-line – sched. algorithm is executed on the whole task set before activation
  - Online – schedule is updated at runtime every time a new task enters the system
- Optimal vs Heuristic
  - Optimal – algorithm computes a feasible schedule and minimizes cost of soft real-time jobs
  - Heuristic – algorithm is guided by heuristic function; tends towards optimal schedule, may not give one

Scheduling – Clock-Driven

- Decisions about what jobs execute when are made at specific time instants
  - these instants are chosen before the system begins execution
  - Usually regularly spaced, implemented using a periodic timer interrupt
  - Scheduler awakes after each interrupt, schedules jobs to execute for the next period, then blocks itself until the next interrupt

    E.g. the helicopter example with the interrupt every 1/180 th of a second
- Typically in clock-driven systems:
  - All parameters of the real-time jobs are fixed and known
  - A schedule of the jobs is computed off-line and is stored for use at runtime; thus scheduling overhead at run-time can be minimized
  - Simple and straight-forward, not flexible

Scheduling – Priority-Driven

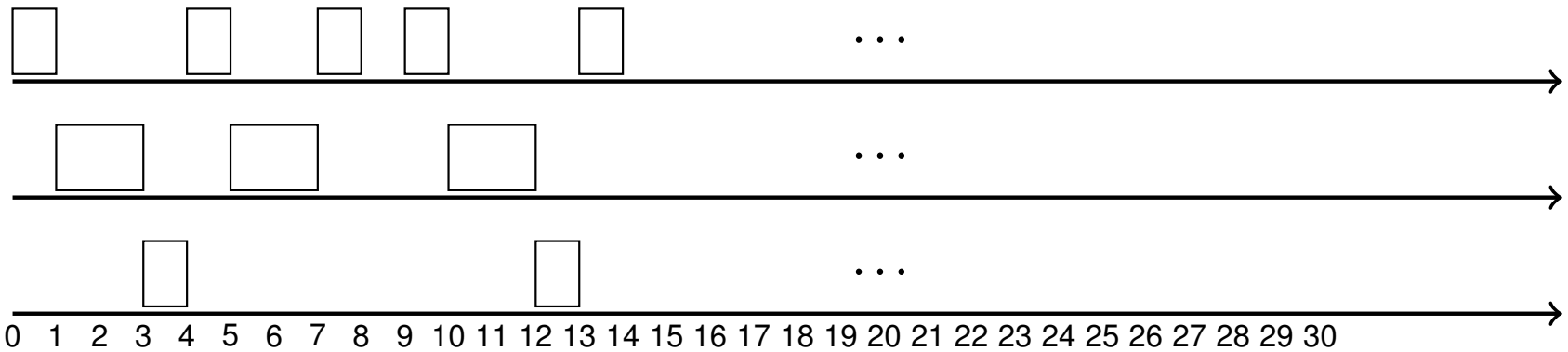
- Assign priorities to jobs, based on some algorithm
- Make scheduling decisions based on the priorities, when events such as releases and job completions occur
  - Priority scheduling algorithms are *event-driven*
  - Jobs are placed in one or more queues; at each event, the ready job with the highest priority is executed

  (The assignment of jobs to priority queues, along with rules such as whether preemption is allowed, completely defines a priority-driven alg.)
- Priority-driven algs. make *locally optimal* scheduling decisions
  - Locally optimal scheduling is often *not* globally optimal
  - Priority-driven algorithms *never* intentionally leave idle processors
- Typically in priority-driven systems:
  - Some parameters do not have to be fixed or known
  - A schedule is computed online; usually results in larger scheduling overhead as opposed to clock-driven scheduling
  - Flexible – easy to add/remove tasks or modify parameters

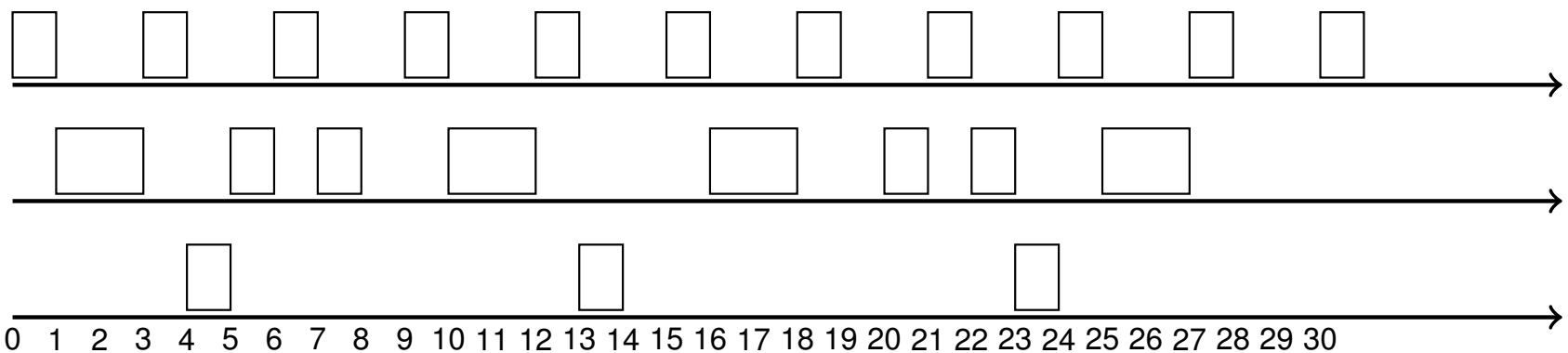
Clock-Driven & Priority-Driven Example

	T_1	T_2	T_3
p_i	3	5	10
e_i	1	2	1

Clock-Driven:



Priority-driven: $T_1 \succ T_2 \succ T_3$



Real-Time Scheduling

Scheduling of Reactive Systems

Clock-Driven Scheduling

Current Assumptions

- Fixed number, n, of periodic tasks $T_1, \ldots, T_n$
- Parameters of periodic tasks are known a priori
  - Execution time $e_{i,k}$ of each job $J_{i,k}$ in a task T_i is fixed
  - For a job $J_{i,k}$ in a task T_i we have
    - $r_{i,1} = \varphi_i = 0$ (i.e., synchronized)
    - $r_{i,k} = r_{i,k-1} + p_i$
- We allow aperiodic jobs
  - assume that the system maintains a single queue for aperiodic jobs
  - Whenever the processor is available for aperiodic jobs, the job at the head of this queue is executed
- We treat sporadic jobs later

Static, Clock-Driven Scheduler

- Construct a *static schedule* offline
  - The schedule specifies exactly when each job executes
  - The amount of time allocated to every job is equal to its execution time
  - The schedule repeats each hyperperiod

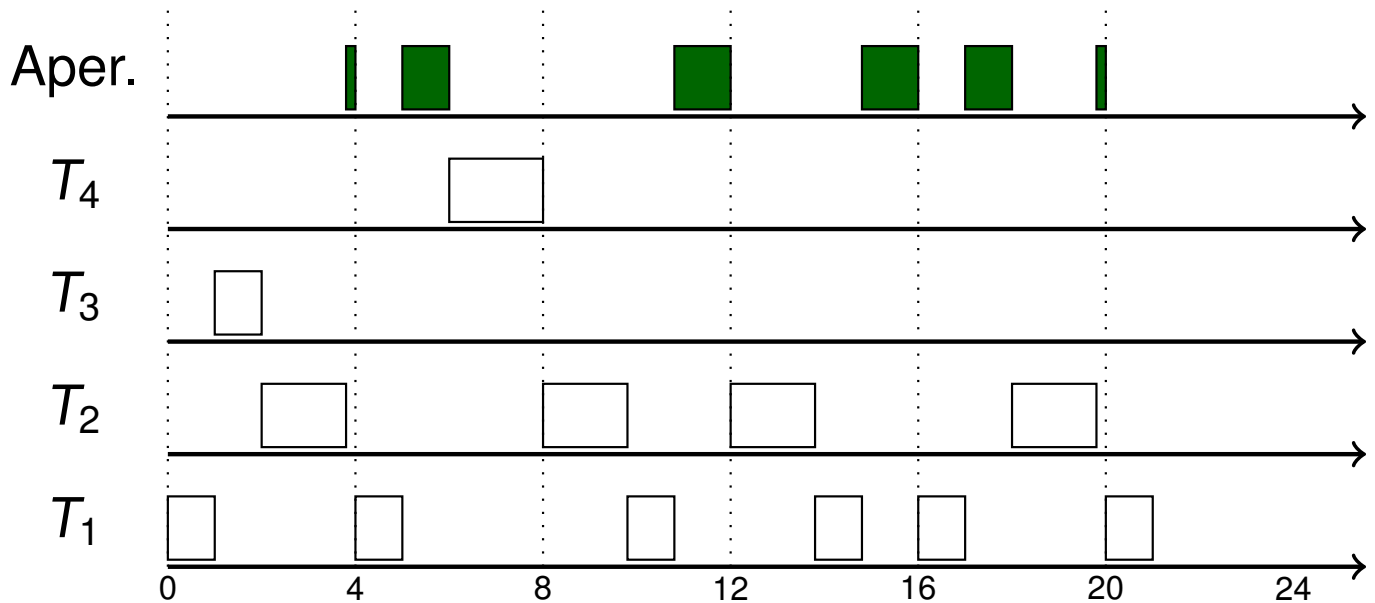
    i.e. it suffices to compute the schedule up to hyperperiod
- Can use complex algorithms offline
  - Runtime of scheduling algorithm is not relevant
  - Can compute a schedule that optimizes some characteristics of the system

    e.g. a schedule where the idle periods are nearly periodic (useful to accommodate aperiodic jobs)

Example

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

Hyperperiod $H = 20$



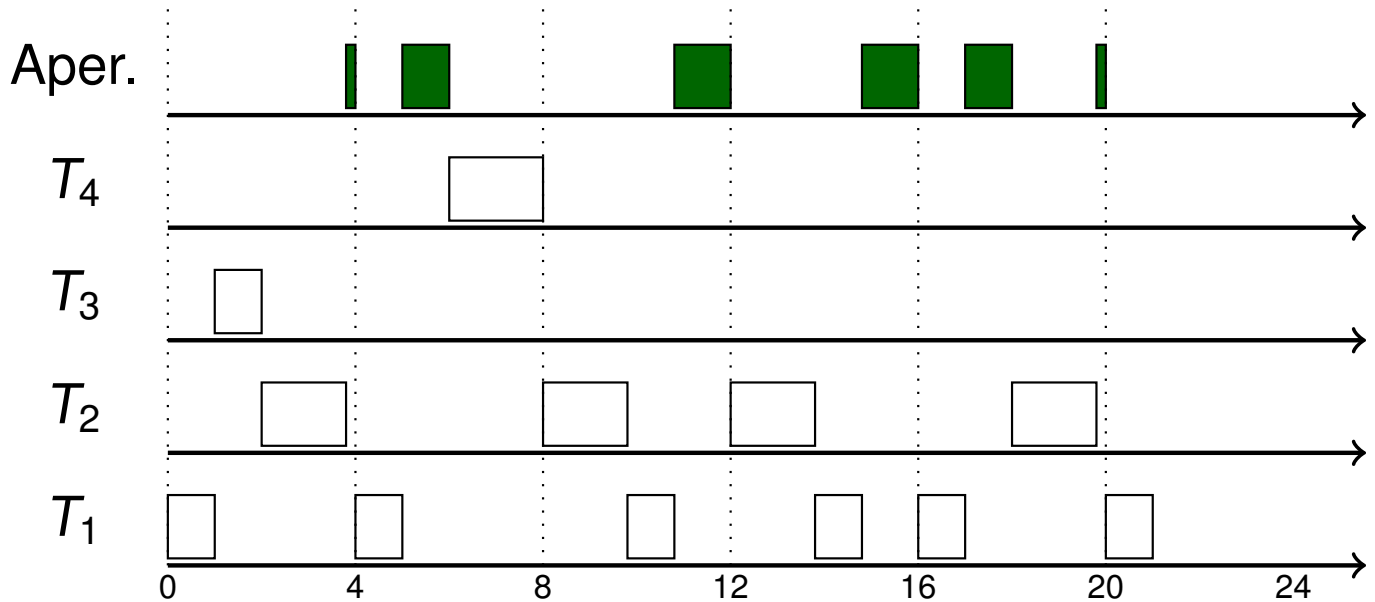
Implementation of Static Scheduler

- Store pre-computed schedule as a table
  - Each entry $(t_k, T(t_k))$ gives
    - a decision time t_k
    - scheduling decision $T(t_k)$ which is either a task to be executed, or idle (denoted by I)
- The system creates all tasks that are to be executed:
  - Allocates memory for the code and data
  - Brings the code into memory
- Scheduler sets the hardware timer to interrupt at the first decision time $t_0 = 0$
- On receipt of an interrupt at t_k:
  - Scheduler sets the timer interrupt to t_{k+1}
  - If previous task overrunning, handle failure
  - If $T(t_k) = I$ and aperiodic job waiting, start executing it
  - Otherwise, start executing the next job in $T(t_k)$

Example

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

Hyperperiod $H = 20$



t_k	0.0	1.0	2.0	3.8	4.0	5.0	6.0	$\cdots$
$T(t_k)$	T_1	T_3	T_2	I	T_1	I	T_4	$\cdots$

Implementation of Static Scheduler

Input: the table $(t_k, T(t_k))$ for $k = 0, 1, \ldots, n-1$
Task SCHEDULER:
    set the number of decisions $i := 0$ and table entry $k := 0$;
    set the timer to expire at t_0;
    do forever:
        accept timer interrupt;
        if an aperiodic job is executing, preempt it;
        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 $\lfloor i/n \rfloor * H + t_k$;
        if the current task $T = I$,
            execute the job at the head of the aperiodic queue;
        else
            let the task T execute;
        sleep;
    end do.
End SCHEDULER

Frame Based Scheduling

- 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 a structure is imposed
  - Make scheduling decisions at periodic intervals *(frames)* of length f
  - Execute a fixed list of jobs within each frame; **no preemption within frames**

How to choose the size of frames? How to compute a schedule?

To simplify further development, assume that all parameters are in $\mathbb{N}$ and choose frame sizes in $\mathbb{N}$

Frame Based Scheduling – Frame Size

0. Necessary condition for avoiding preemption of jobs is

$$f \geq \max_i e_i$$

(i.e. we want each job to have a chance to finish within a frame)

1. To minimize the number of entries in the cyclic schedule, the hyper-period should be an integer multiple of the frame size, i.e.

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

for some task T_i.

2. To allow scheduler to check that jobs complete by their deadline, at least one frame should lie between release time of a job and its deadline, i.e.

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

for all tasks T_i

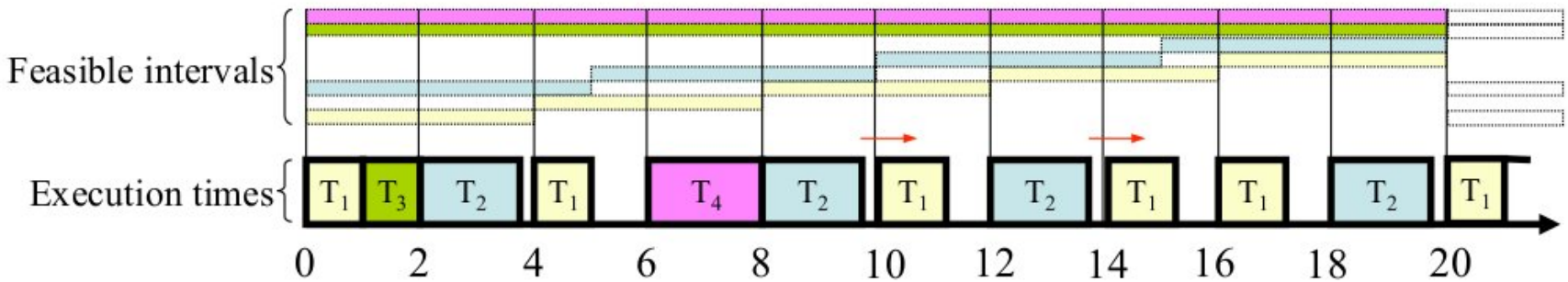
Example: $T_1 = (4, 1.0)$, $T_2 = (5, 1.8)$, $T_3 = (20, 1.0)$, $T_4 = (20, 2.0)$
Then $f \in \mathbb{N}$ satisfies 0.–2. iff $f = 2$.

Frame Based Scheduling – Frame Size – Example

Example 1

$T_1 = (4, 1.0)$, $T_2 = (5, 1.8)$, $T_3 = (20, 1.0)$, $T_4 = (20, 2.0)$
Then $f \in \mathbb{N}$ satisfies 0.–2. iff $f = 2$.

With $f = 2$ is schedulable:



Frame Based Scheduling – Job Slices

- Sometimes a system cannot meet all three frame size constraints simultaneously (and even if it meets the constraints, no non-preemptive schedule is feasible)
- Can be solved by partitioning a job with large execution time into slices with shorter execution times

  This, in effect, allows preemption of the large job

To construct a schedule, we have to make three kinds of design decisions (that cannot be taken independently):

- Choose a frame size based on constraints
- Partition jobs into slices
- Place slices into frames

Frame Based Scheduling – Job Slices – Example

- Consider $T_1 = (4, 1)$, $T_2 = (5, 2, 7)$, $T_3 = (20, 5)$
- Cannot satisfy constraints: 1. $\Rightarrow f \geq 5$ but 3. $\Rightarrow f \leq 4$
- Solve by splitting T_3 into $T_{3,1} = (20, 1)$, $T_{3,2} = (20, 3)$, and $T_{3,3} = (20, 1)$
  (Other splits exist)
- Result can be scheduled with $f = 4$

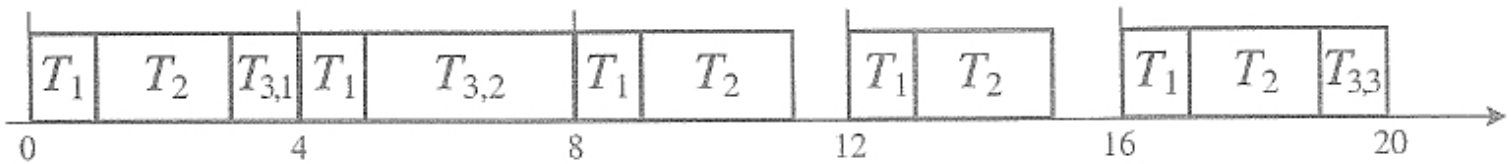


FIGURE 5–6 A preemptive cyclic schedule of $T_1 = (4, 1)$, $T_2 = (5, 2, 7)$ and $T_3 = (20, 5)$.

Frame Based Scheduling – Job Slices

Assuming that preemption is allowed in arbitrary places, jobs are independent and there are no resource contentions, there is a (pseudo)polynomial time algorithm which

- chooses an appropriate frame size f
- partitions jobs into slices
- places slices into frames

(whiteb.)

Frame Based Scheduling – Algorithm

- Compute all frame sizes satisfying conditions 1. and 2. (not necessarily 0.)
- For every frame size f construct a *network flow graph*:
  (the number of frames in one hyperperiod is F)
  - Vertices:
    - a vertex for each job J_i
    - a vertex for each frame j where $j = 1, \ldots, F$
    - *source* and *sink*
  - Edges:
    - from J_i to j of capacity f if J_i can be scheduled in the frame j
    - from *source* to J_i of capacity e_i
    - from every frame to *sink* of capacity f

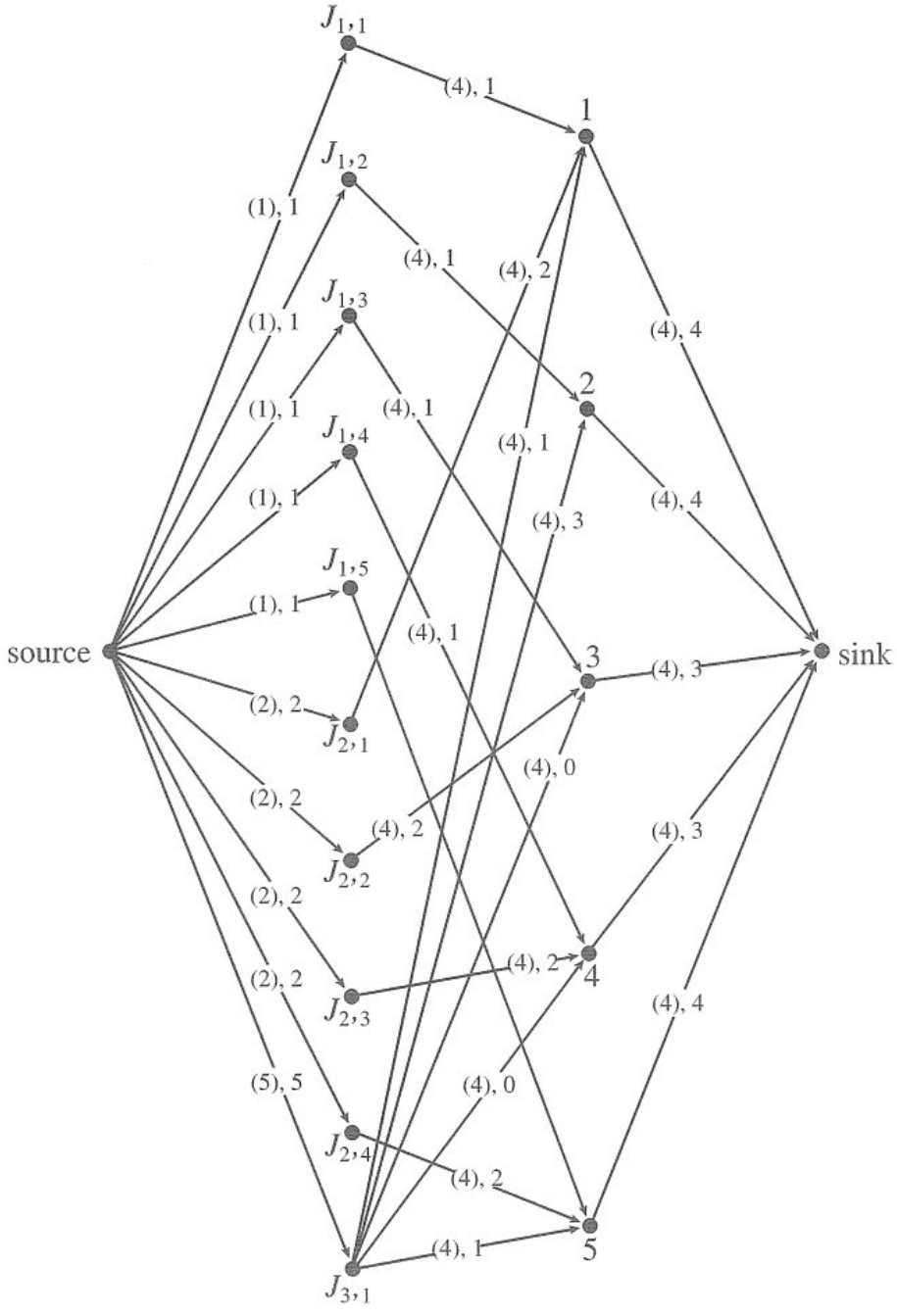
Theorem 2

Max flow assigned to every edge from source to J_i is e_i

iff

the jobs can be partitioned into slices and the slices placed into frames so that the resulting schedule is feasible. The flows assigned to edges of the form (J_i, j) determine the schedule.

Frame Based Scheduling – Network Flow Example



The example with
$T_1 = (4, 1)$, $T_2 = (5, 2, 7)$,
$T_3 = (20, 5)$ and $f = 4$

Note that

T_2 has four jobs $J_{2,1}$, $J_{2,2}$, $J_{2,3}$, $J_{2,4}$ in one hyperper.

The only job of T_3 released in one hyperper. can be placed into any frame

Frame Based Scheduling – Cyclic Executive

- Modify previous table-driven scheduler to be frame based
- Table that drives the scheduler has F entries, where $F = H/f$
  - The k-th entry $L(k)$ lists the names of the job slices that are to be scheduled in frame k ($L(k)$ is called *scheduling block*)
  - Each job slice is implemented by a procedure
- Cyclic executive executed by the clock interrupt that signals the start of a frame:
  - If an aperiodic job is executing, preempts it; if a periodic overruns, handles the overrun
  - Determines the appropriate scheduling block for this frame
  - Executes the jobs in the scheduling block
  - Executes jobs from the head of the aperiodic job queue for the remainder of the frame
- Less overhead than pure table driven cyclic scheduler, since only interrupted on frame boundaries, rather than on each job

Scheduling Aperiodic Jobs

So far, aperiodic jobs scheduled in the background after all jobs with hard deadlines

This may unnecessarily delay aperiodic jobs

Note: There is no advantage in completing periodic jobs early

Ideally, finish periodic jobs by their respective deadlines.

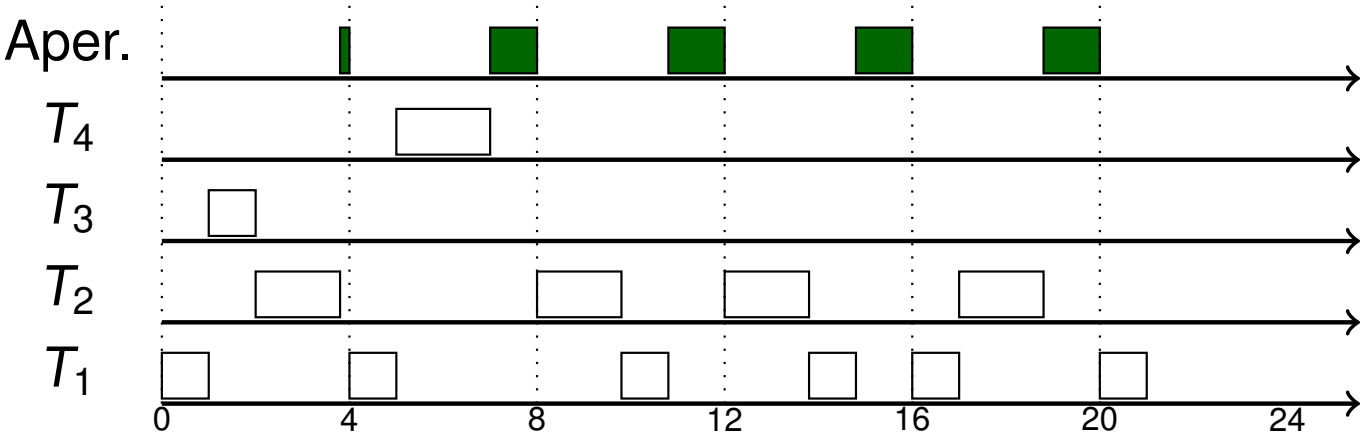
Slack Stealing:

- Slack time in a frame = the time left in the frame after all (remaining) slices execute
- Schedule aperiodic jobs ahead of periodic in the slack time of periodic jobs
  - The cyclic executive keeps track of the slack time left in each frame as the aperiodic jobs execute, preempts them with periodic jobs when there is no more slack
  - As long as there is slack remaining in a frame and the aperiodic jobs queue is non-empty, the executive executes aperiodic jobs, otherwise executes periodic
- Reduces resp. time for aper. jobs, but requires accurate timers

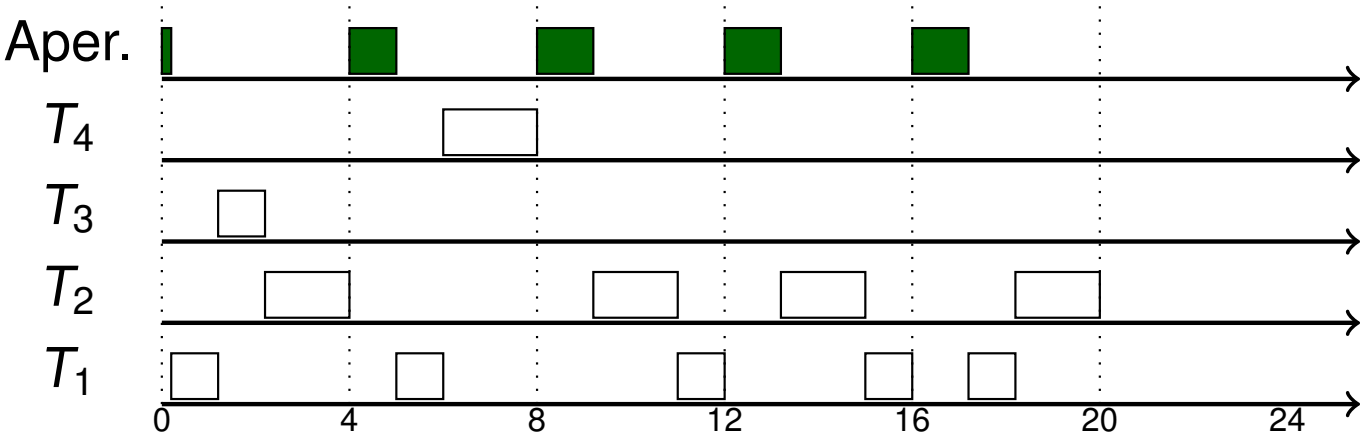
Example

Assume that the aperiodic queue is never empty.

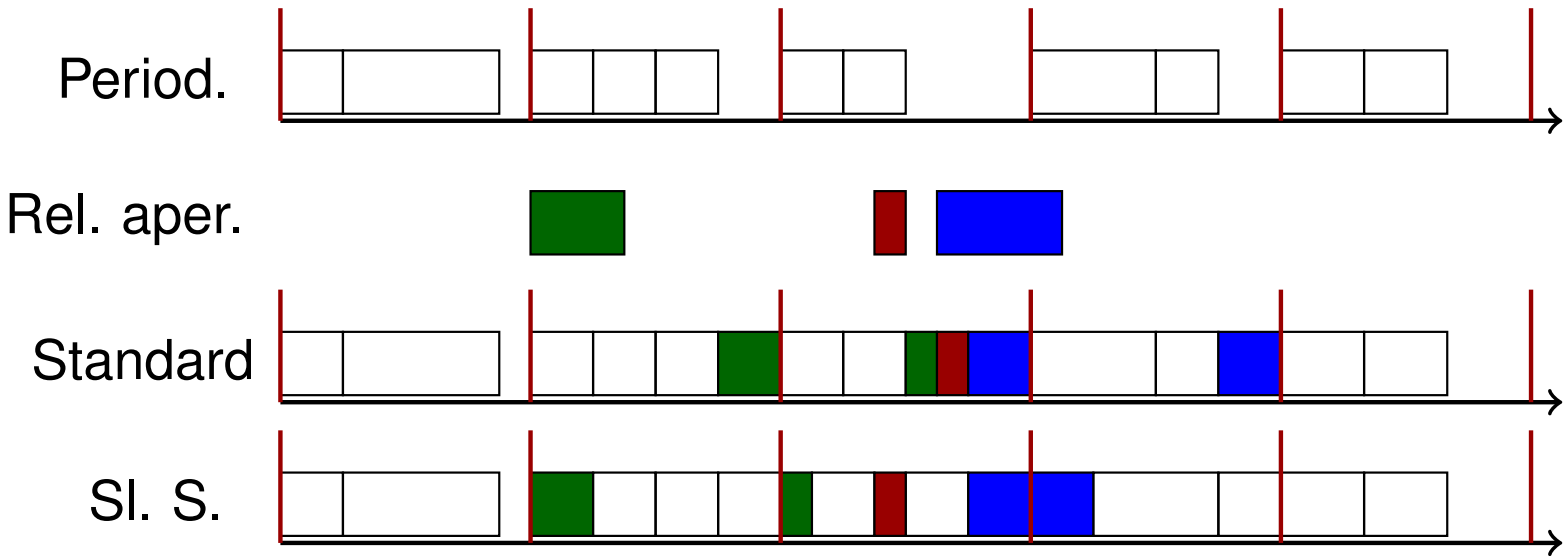
Aperiodic at the ends of frames:



Slack stealing:



Slack Stealing – cont.



Frame Based Scheduling – Sporadic Jobs

Let us allow sporadic jobs

i.e. hard real-time jobs whose release and exec. times are not known a priori

The scheduler determines whether to accept a sporadic job when it arrives (and its parameters become known)

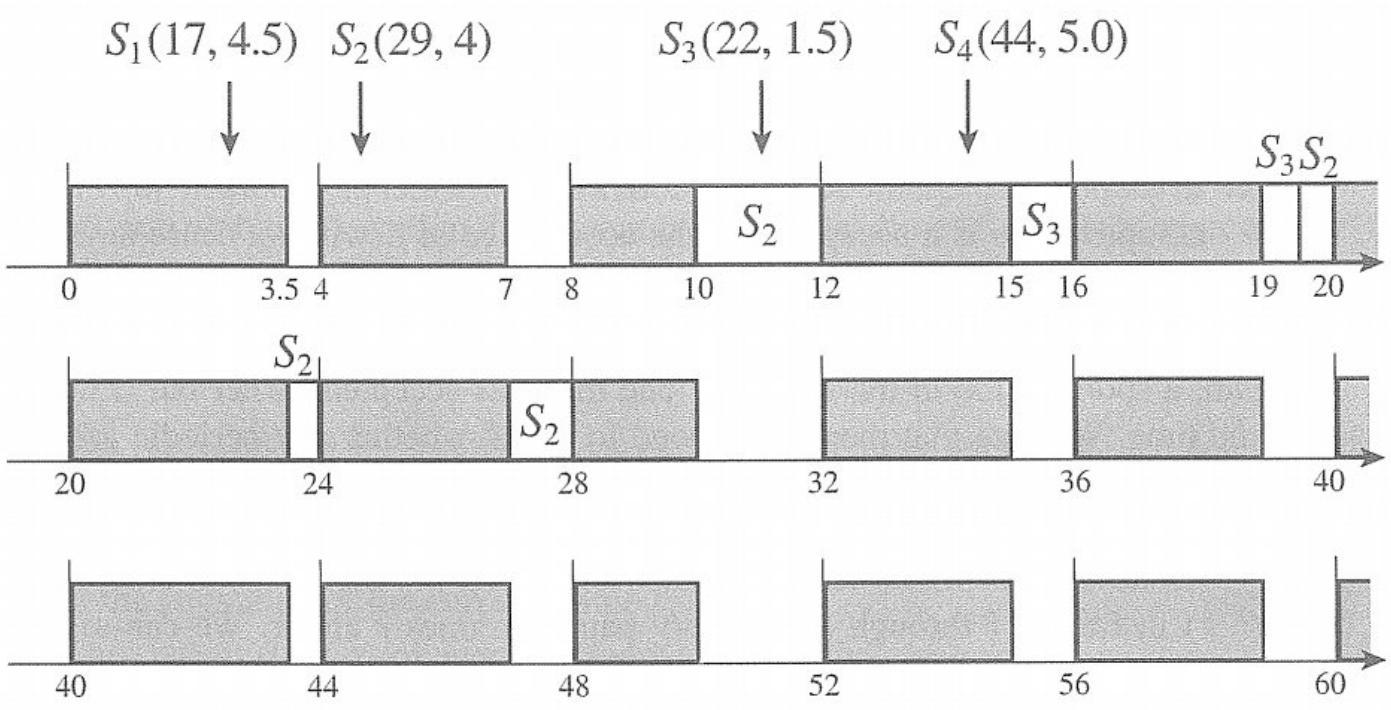
- Perform *acceptance test* to check whether the new sporadic job can be feasibly scheduled with all the jobs (periodic and sporadic) in the system at that time

  Acceptance check done at the beginning of the next frame; has to keep execution times of the parts of sporadic jobs that have already executed
- If there is sufficient slack time in the frames before the new job's deadline, the new sporadic job is accepted; otherwise, rejected
- Among themselves, sporadic jobs scheduled according to EDF

  This is optimal for sporadic jobs

Note: rejection is often better than missing deadline

e.g. a robotic arm taking defective parts off a conveyor belt: if the arm cannot meet deadline, the belt may be slowed down or stopped



- $S_1(17, 4.5)$ released at 3 with abs. deadline 17 and execution time 4.5; acceptance test at 4; must be scheduled in frames 2, 3, 4; total slack in these frames is 4, i.e. rejected
- $S_2(29, 4)$ released at 5 with abs. deadline 29 and exec. time 4; acc. test at 8; total slack in frames 3-7 is 5.5, i.e. accepted
- $S_3(22, 1.5)$ released at 11 with abs. deadline 22 and exec. time 1.5; acc. test at 12;

  2 units of slack in frames 4, 5 as S_3 will be executed *ahead of the remaining parts of* S_2 by EDF – check whether there will be enough slack for the remaining parts of S_2, accepted
- $S_4(44, 5.0)$ is rejected (only 4.5 slack left)

Handling Overruns

Overruns may happen due to failures

e.g. unexpectedly large data over which the system operates, hardware failures, etc.

Ways to handle overruns:

- Abort the overrun job at the beginning of the next frame; log the failure; recover later

  e.g. control law computation of a robust digital controller
- Preempt the overrun job and finish it as an aperiodic job

  use this when aborting job would cause “costly” inconsistencies
- Let the overrun job finish – start of the next frame and the execution jobs scheduled for this frame are delayed

  This may cause other jobs to be delayed

  depends on application

Clock-drive Scheduling: Conclusions

Advantages:

- Conceptual simplicity
  - Complex dependencies, communication delays, and resource contention among jobs can be considered when constructing the static schedule
  - Entire schedule in a static table
  - No concurrency control or synchronization needed
- Easy to validate, test and certify

Disadvantages:

- Inflexible
  - If any parameter changes, the schedule must be usually recomputed

    Best suited for systems which are rarely modified (e.g. controllers)
  - Parameters of the jobs must be fixed

    As opposed to most priority-driven schedulers

Real-Time Scheduling

Scheduling of Reactive Systems

Priority-Driven Scheduling

Current Assumptions

- Single processor
- Fixed number, n, of *independent periodic* tasks

  i.e. there is no dependency relation among jobs
  - Jobs can be preempted at any time and never suspend themselves
  - No aperiodic and sporadic jobs
  - No resource contentions

Moreover, unless otherwise stated, we assume that

- **Scheduling decisions take place precisely at**
  - release of a job
  - completion of a job

  (and nowhere else)
- Context switch overhead is negligibly small

  i.e. assumed to be zero
- There is an unlimited number of priority levels

Fixed-Priority vs Dynamic-Priority Algorithms

A priority-driven scheduler is on-line
i.e. it does not precompute a schedule of the tasks

- It assigns priorities to jobs after they are released and places the jobs in a ready job queue in the priority order

  with the highest priority jobs at the head of the queue

- At each scheduling decision time, the scheduler updates the ready job queue and then schedules and executes the job at the head of the queue

  i.e. one of the jobs with the highest priority

Fixed-priority = all jobs in a task are assigned the same priority

Dynamic-priority = jobs in a task may be assigned different priorities

Note: In our case, a priority assigned to a job does not change. There are *job-level dynamic priority* algorithms that vary priorities of individual jobs – we won't consider such algorithms.

Fixed-priority Algorithms – Rate Monotonic

Best known fixed-priority algorithm is *rate monotonic (RM)* scheduling that assigns priorities to tasks based on their periods

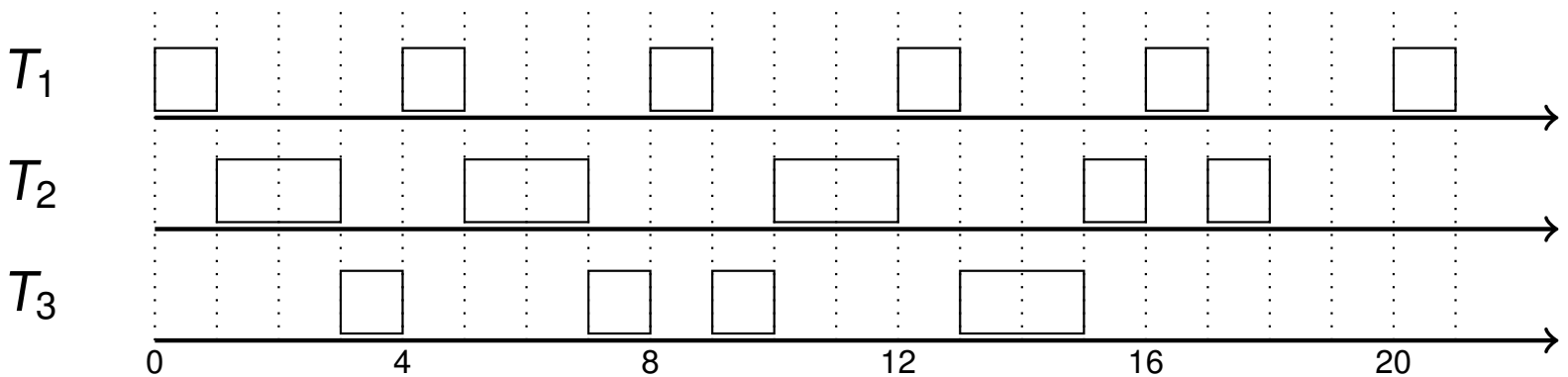
- The shorter the period, the higher the priority
- The *rate* is the inverse of the period, so jobs with higher rate have higher priority

RM is very widely studied and used

Example 3

$T_1 = (4, 1)$, $T_2 = (5, 2)$, $T_3 = (20, 5)$
with rates $1/4$, $1/5$, $1/20$, respectively

The priorities: $T_1 \succ T_2 \succ T_3$



Fixed-priority Algorithms – Deadline Monotonic

The *deadline monotonic (DM)* algorithm assigns priorities to tasks based on their *relative deadlines*

- the shorter the deadline, the higher the priority

Observation: When relative deadline of every task matches its period, then RM and DM give the same results

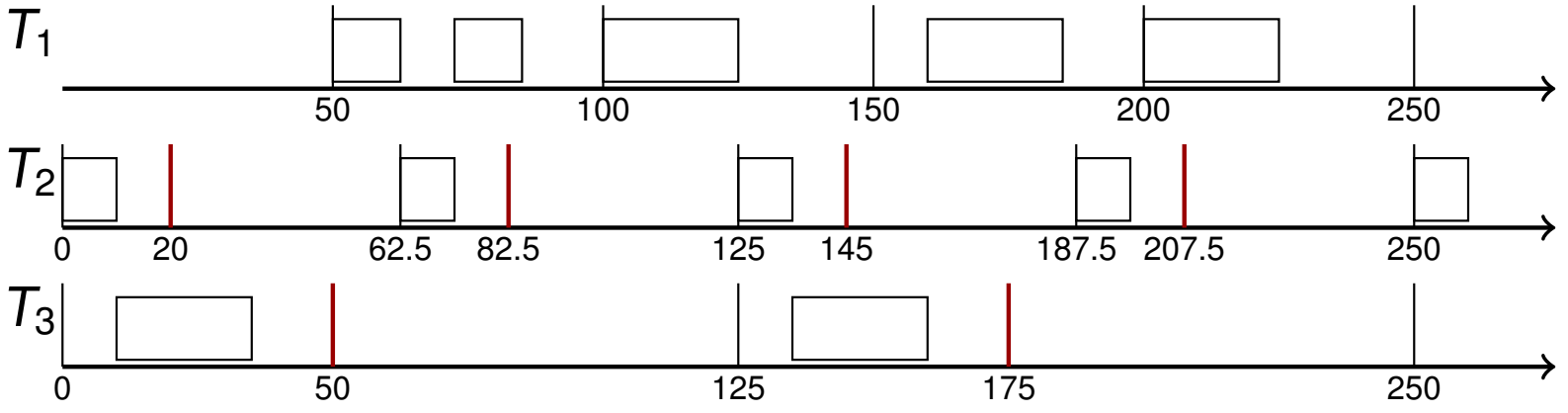
Proposition 1

When the relative deadlines are arbitrary DM can sometimes produce a feasible schedule in cases where RM cannot.

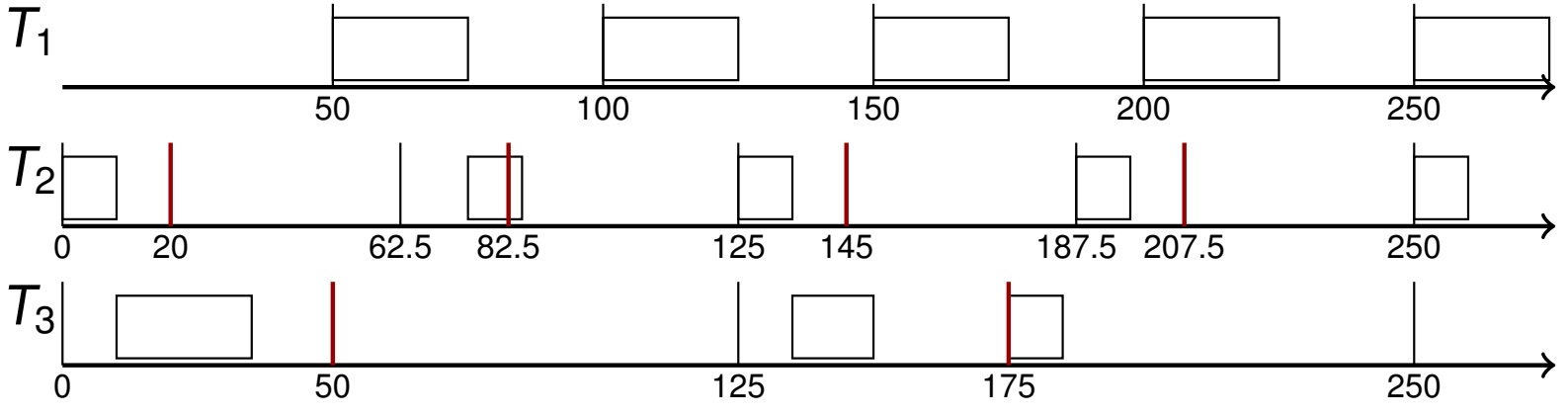
Rate Monotonic vs Deadline Monotonic

$T_1 = (50, 50, 25, 100)$, $T_2 = (0, 62.5, 10, 20)$, $T_3 = (0, 125, 25, 50)$

DM is optimal (with priorities $T_2 \succ T_3 \succ T_1$):



RM is not optimal (with priorities $T_1 \succ T_2 \succ T_3$):



Dynamic-priority Algorithms

Best known is *earliest deadline first (EDF)* that assigns priorities based on *current* deadlines

- At the time of a scheduling decision, the job queue is ordered by earliest deadline

Another one is the *least slack time (LST)*

- The job queue is ordered by least slack time

Recall that the *slack time* of a job J_i at time t is equal to $d_i - t - x$ where x is the remaining computation time of J_i at time t

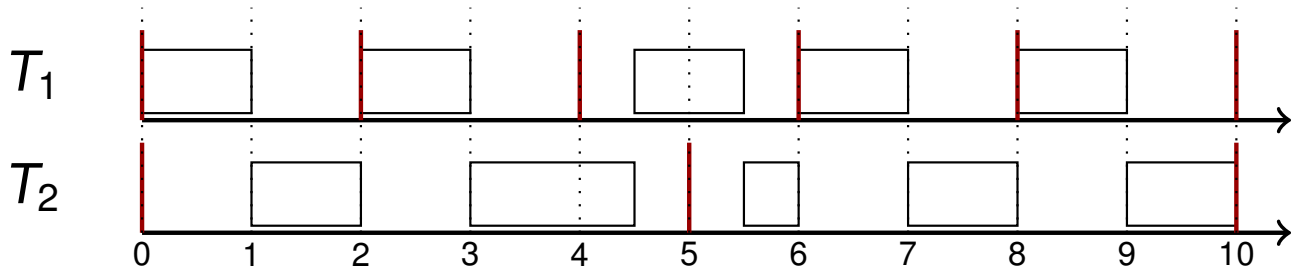
Comments:

- There is also a strict LST which reassigns priorities to jobs whenever their slacks change relative to each other – won't consider
- Standard "non real-time" algorithms such as FIFO and LIFO are also dynamic-priority algorithms

We focus on EDF here, leave some LST for homework

EDF – Example

$T_1 = (2, 1)$ and $T_2 = (5, 2.5)$



Note that the processor is 100% “utilized”, not surprising :-)

Summary of Algorithms

In what follows we consider

Dynamic-priority algorithms: EDF

Fixed-priority algorithms: RM and DM

We consider the following questions:

- Are the algorithms optimal?
- How to efficiently (or even online) test for schedulability?