Clock-driven scheduling

Also known as static or off-line scheduling

Michal Sojka

Czech Technical University in Prague,
FEE and CIIRC

November 24, 2021

Some slides are derived from lectures by Steve Goddard and James H. Anderson

Classification of scheduling algorithms

(used in real-time systems)

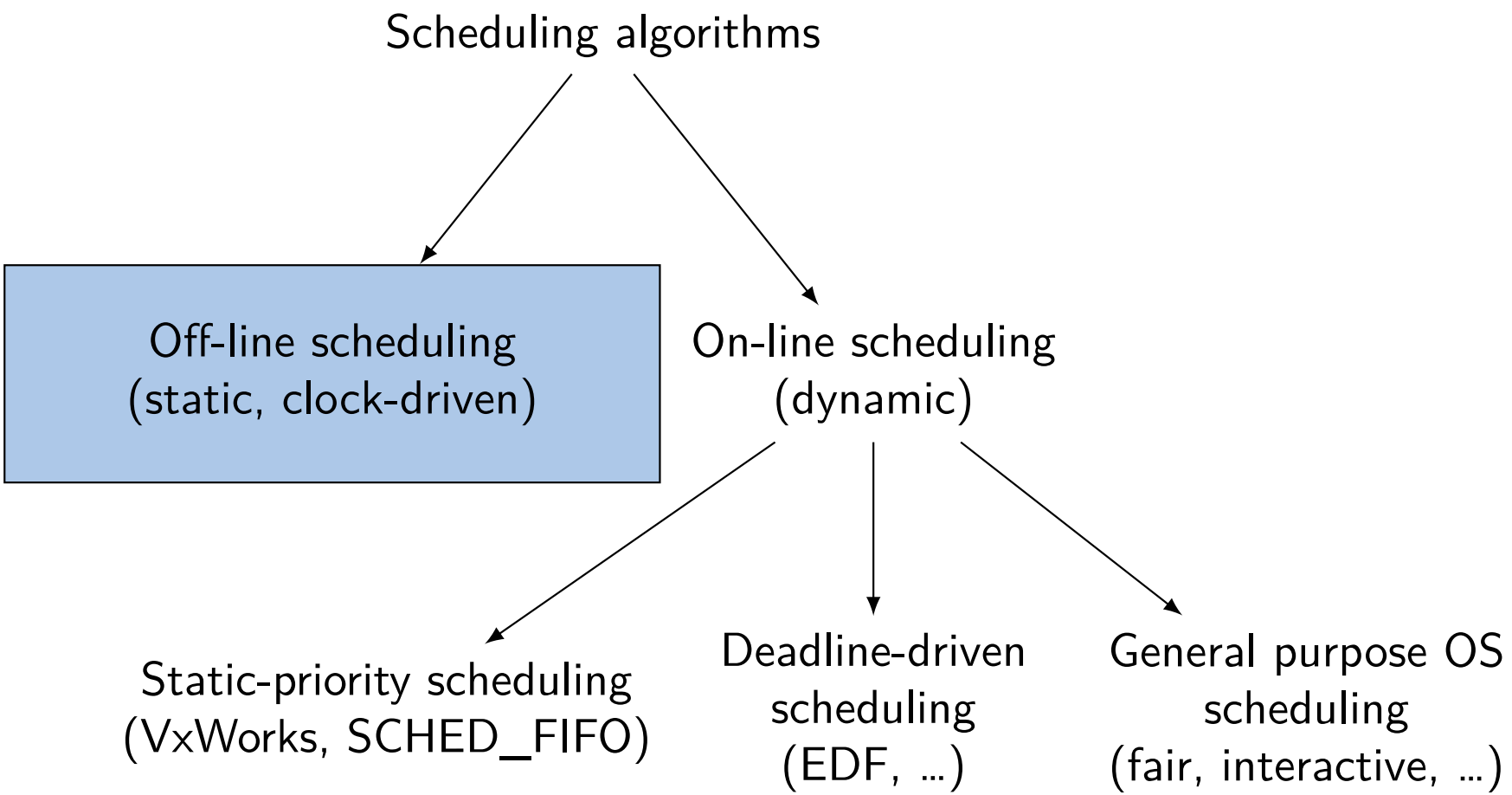


Table of contents

Outline

Motivation

Helicopter control system

Three tasks:

1. 180× per second, total computation time 1 ms
   - Read sensor data
   - Compute the control laws of the inner yaw-control loop
   - Apply the control laws results to the actuators
   - Perform internal checks
2. 90× per second, total computation time 3 ms:
   - Compute the control laws of the inner pitch-control loop
   - Compute the control laws of the inner roll- and collective-control loop
3. 30× per second, total computation time 10 ms:
   - Compute the control lows of the outer loops

How to implement this control system?
Safety-critical application ⇒ KISS

Clock-driven (or static or off-line) scheduling

(Chapter 5 from Liu)

- Decision are only made at a priory chosen time instants
- It is sufficient to have a hardware timer (no need for OS)
- Regularly spaced time instants are popular
- Schedule is computed off-line and stored for use at run-time
  - All parameters of hard real-time jobs are fixed and known
  - Scheduling overhead during run time is minimal
  - Complexity of the scheduling algorithm is not important
  - Good (optimal) off-line schedules can be found
  - **Disadvantage:** no flexibility
- Applicable only when we know all about the system in advance
  - Fixed set of tasks, fixed and known task parameters and resource requirements
  - Met by many safety-critical applications
  - Easier to certify

Clock driven (static) scheduling

For this lecture, we will consider the following periodic model:

- n periodic tasks $\tau_1, \ldots, \tau_n$; n is constant
- We assume the "rest of world" model.
- τ_i is specified with tuple (ϕ_i, T_i, C_i, D_i), where
  - ϕ_i is task phase,
  - T_i is task period,
  - C_i is execution time of τ_i and
  - D_i is relative deadline.
  - We will shorten this as (T_i, C_i, D_i), if $\phi_i = 0$ and as (T_i, C_i) if $\phi_i = 0 \wedge T_i = D_i$.
- In addition, we consider aperiodic jobs released in arbitrary instants. Later, we will look at sporadic tasks as well.
- We are interested in scheduling for **one** processor.
- Scheduling for multiple processors is covered in "Combinatorial Optimization" course.

Schedule table

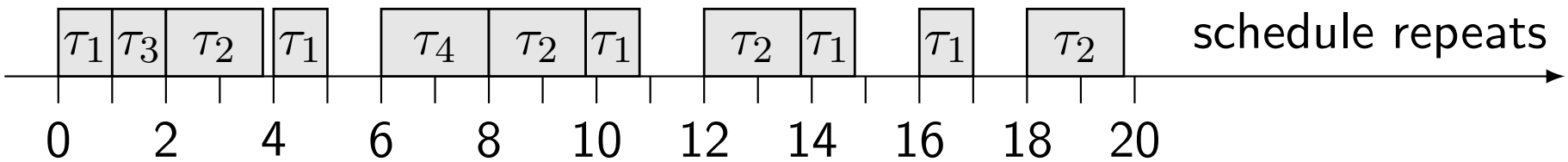
The scheduler will schedule periodic tasks according to a static schedule, which is computed offline and stored in a table.

- Every element of the table specifies decision time and scheduled task.
- If no task is to be scheduled, the special symbol $\times$ (idle) is stored there.
- We will look at a simple algorithm how to construct the schedule later.
  - Note: This algorithm need not be highly efficient (it runs off-line).
- Aperiodic jobs can be scheduled at idle intervals.

Example

System of four tasks:

	τ_1	τ_2	τ_3	τ_4
Period	4	5	20	20
Execution time	1	1.8	1	2



The schedule table will look like this:

Time	0	1	2	3.8	4	5	6	8	9.8	10.8	...	18
Task	τ_1	τ_3	τ_2	$\times$	τ_1	$\times$	τ_4	τ_1	$\times$	τ_2	...	τ_2

Example of clock-driven scheduler implementation – basic version

Scheduler – basic version without aperiodic tasks

/* H is hyperperiod, created by N "slices" (not necessarily of the same length).*/

Input: Schedule stored in table $(t_k, \tau(t_k))$ for $k = 0, 1, \ldots, N-1$

Procedure SCHEDULER:

set next scheduling point i and table index k to 0;

set timer to time t_k;

Repeat forever

wait for timer expiration (for example interrupt – instruction `hlt`, or polling)

current task $\tau := \tau(t_k)$;

$i := i + 1$;

calculate index of next table entry as $k := i \mod N$;

set timer to $\lfloor i/N \rfloor \cdot H + t_k$;

call function τ;

We assume that τ always finishes by next timer expiration.

End

End SCHEDULER

Example of clock-driven scheduler

Scheduler – called from interrupt, handles aperiodic tasks

/* H is hyperperiod, created by N "slices" (not necessarily of the same length).*/

Input: Schedule stored in table $(t_k, \tau(t_k))$ pro $k = 0, 1, \ldots, N-1$

Initialization:

set next scheduling point i and table index k to 0;
setup an interrupt handler
set timer to t_k;

In timer interrupt do:

if aperiodic job is running, preempt it;
current task $\tau := \tau(t_k)$;
$i := i + 1$;
calculate index of next table entry as $k := i \mod N$;
set timer to $\lfloor i/N \rfloor \cdot H + t_k$;
if current task τ is $\times$, **then**
activate a job from aperiodic job queue;
otherwise
activate job of task τ; ← Here we also assume task τ to finish by the next interrupt.

return from interrupt

Outline

Frame-based scheduling

Problems

- Big number of tasks $\Rightarrow$ big schedule table
- Embedded systems have limited size of memory
- Reprogramming the timer might be slow

Idea

- Divide time to constant-size frames
- Combine multiple jobs into a single frame
- Scheduling decisions are made only at frame boundaries

- Consequences:
  - There is no preemption within each frame.
  - Each job must fit into the frame
  - In addition to schedule calculation, we should check for various error conditions such as task overrun.
- Let f denote the frame size. How to select f?

Frame size constraints

1. We want big enough frames to fit every job without preempting it. This gives us

$$f \geq \max_{i=1,\ldots,n}(C_i). \tag{1}$$

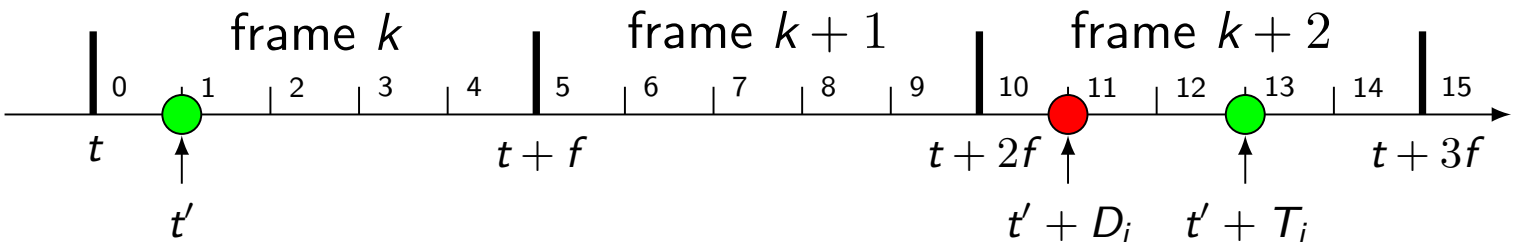
2. In order to have the table small, f should divide H. Since $H = \mathrm{lcm}(T_1, \ldots, T_n)$, f divides T_i for at least one task τ_i:

$$\left\lfloor \frac{T_i}{f} \right\rfloor = \frac{T_i}{f}. \tag{2}$$

Let $F = \frac{H}{f}$ (F is integer). Each interval H is called major cycle and each interval f minor cycle.
Each major cycle is composed from F minor cycles.

Frame size constrains (cont.)

3. We want the frame size to be sufficiently small so that there is at least one whole frame between task release time and deadline. The reasons for this are:
   - Our scheduler can only start jobs at frame boundaries. Jobs release in between have to wait for the boundary.
   - Similarly, we want the scheduler to check whether each jobs complete by its deadline. If there is no frame boundary before the deadline, we cannot prevent overruns. For example, if the deadline falls into frame $k+2$, overrun can be checked only at the beginning of frame $k+2$.



(release time of job τ_i)

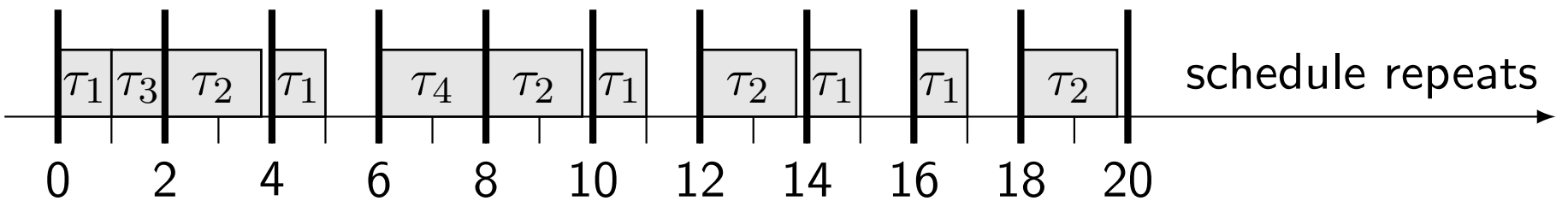
Thus:

$$2f - \gcd(T_i, f) \leq D_i. \tag{3}$$

See Liu Sect. 5.3.2 for explanation and errata_liu.pdf for additional comments.

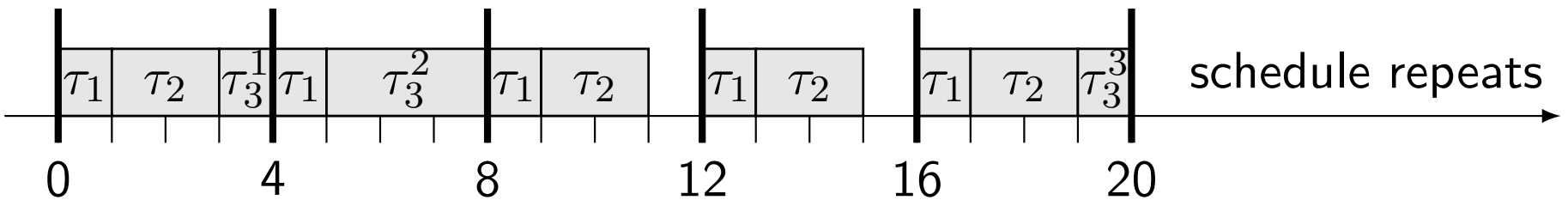
Example

- Let's have a system from the last example with four tasks: $\tau_1 = (4, 1)$, $\tau_2 = (5, 1.8)$, $\tau_3 = (20, 1)$, $\tau_4 = (20, 2)$.
- From the first constraint (1): $f \geq 2$.
- Hyperperiod is 20 so second constraint (2), tells us that f can be one of 2, 4, 5, 10 a 20.
- Third constraint (3) satisfies only $f = 2$.
- Possible cyclic schedule:



Split jobs

- What to do if frame size constrains cannot be met?
- **Example:** Consider $\tau = \{(4, 1), (5, 2, 7), (20, 5)\}$. By first constraint (1) $f \geq 5$, but by third constraint (3) $f \leq 4$!
- **Solution:** Split task $(20, 5)$ into subtasks $(20, 1)$, $(20, 3)$ and $(20, 1)$. Then, $f = 4$ works. Here is a schedule:



Summary of design decisions

- Three design decisions:
  1. choosing a frame size,
  2. partitioning jobs into slices, and
  3. placing slices in frames.
- In general, these decisions cannot be made independently.
- See next slides for an algorithm that makes these decisions for us.

Outline

Network Flow Algorithm

Algorithm for finding a static schedule

Initialization Find all possible frame sizes by constraints (2) a (3):

$$\left\lfloor \frac{T_i}{f} \right\rfloor - \frac{T_i}{f} = 0 \qquad 2f - \gcd(T_i, f) \leq D_i$$

We will ignore constraint (1)

$$f \geq \max_{i=1,\ldots,n} (C_i)$$

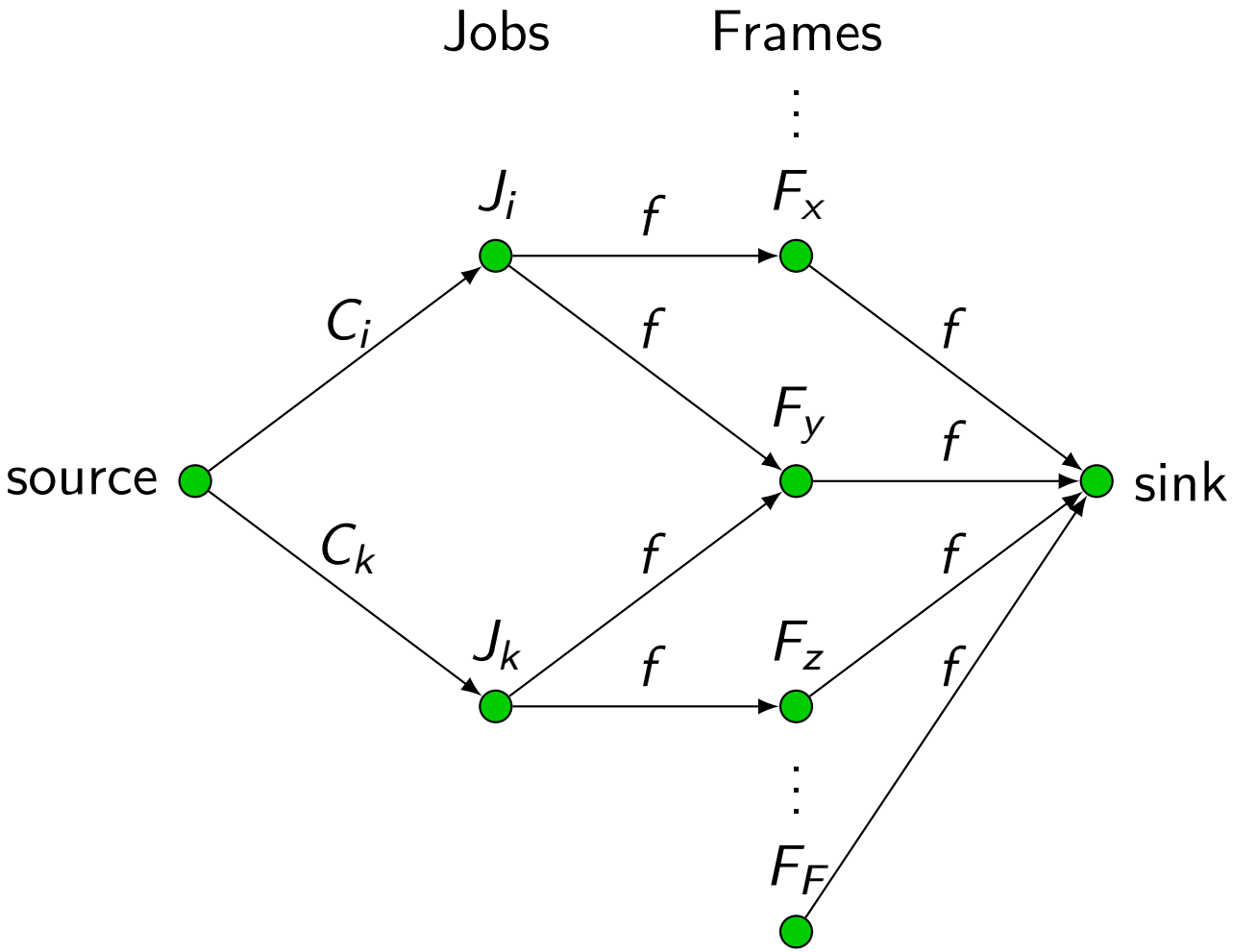
this can force us to split tasks to into subtasks.

Iterate For each frame size f construct *network flow graph* and run max-flow algorithm. If the algorithm finds a flow, we have a schedule.

Network-flow graph

- Denote all jobs in the major cycle of F frames as $J_1, J_2, \ldots, J_N$.
- **Nodes:**
  - N nodes representing jobs, denoted as $J_1, J_2, \ldots, J_N$.
  - F nodes representing frames, denoted as $F_1, F_2, \ldots, F_F$.
  - Source and sink.
- **Edges:**
  - (J_i, F_j) with capacity f, if J_i can be scheduled in frame F_j, i.e. frame F_j satisfies release-time and deadline constraints of J_i.
  - $\forall i : (source, J_i)$ with capacity C_i.
  - $\forall j : (J_j, sink)$ with capacity f.

Network-flow graph example



Finding the schedule

- Clearly, maximal flow is at most equal to

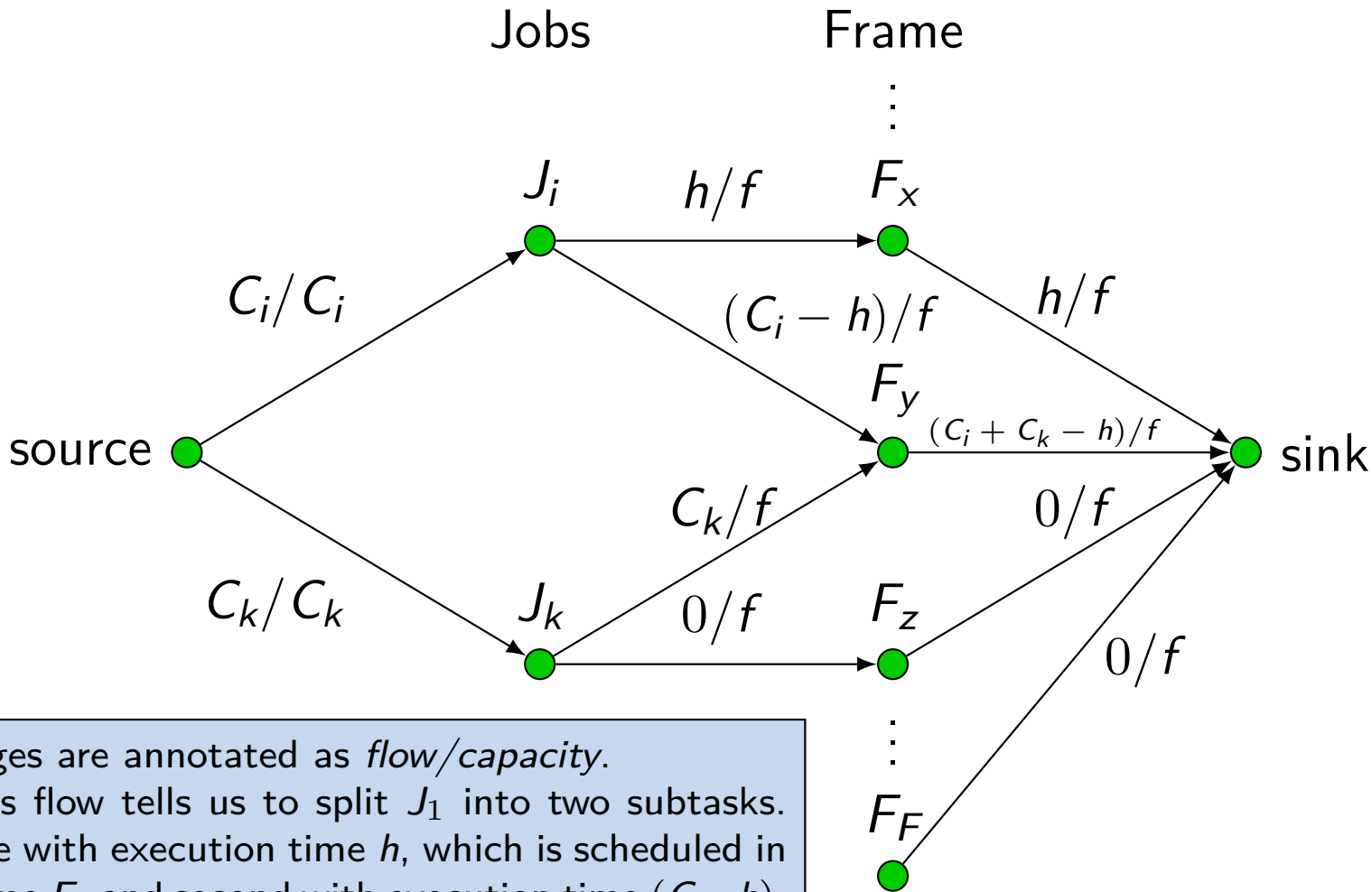
$$\sum_{i=1,\ldots,N} C_i.$$

  This is the duration of jobs that need to be scheduler during major cycle.
- If the flow found by the maximum-flow algorithm is

$$\sum_{i=1,\ldots,N} C_i,$$

  it corresponds to feasible schedule.
- If a job is "scheduled" in multiple frames, it has to be split to sub-jobs.

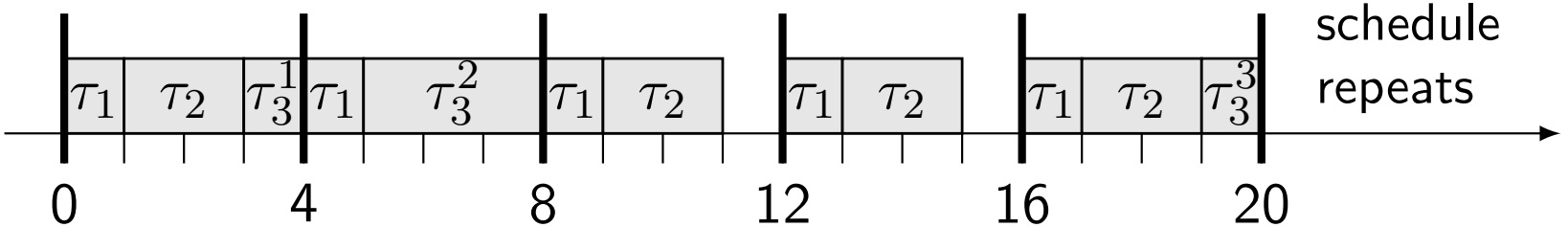
Example graph and maximum flow



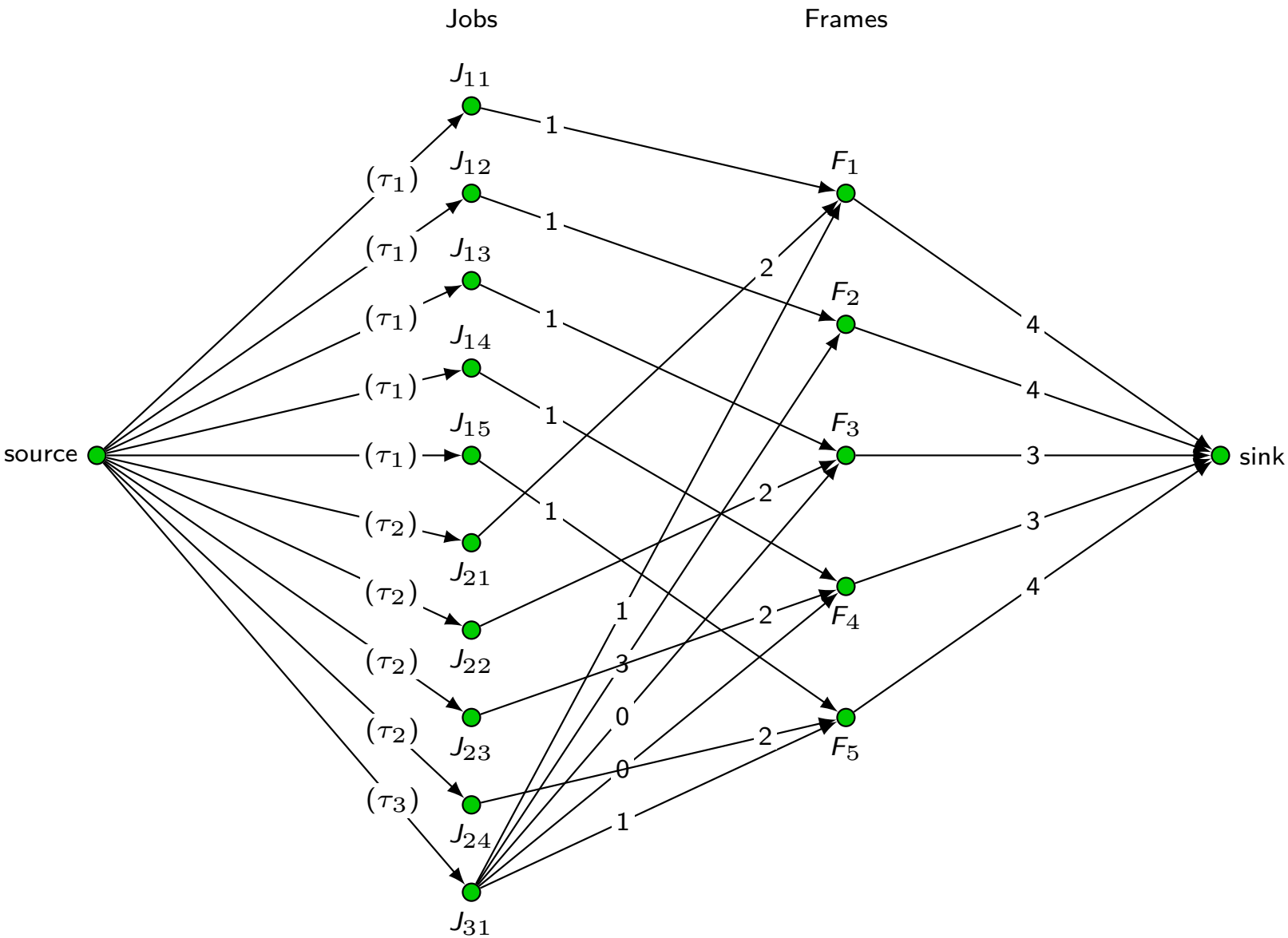
Edges are annotated as *flow/capacity*.
This flow tells us to split J_1 into two subtasks. One with execution time h, which is scheduled in frame F_x and second with execution time $(C_i - h)$, which is scheduled in frame F_y. J_k stays as one jobs as is scheduled in frame F_y.

Example

- **Example:** Consider $\tau = \{(4, 1), (5, 2, 7), (20, 5)\}$. Constraints (2) and (3) give us $f = 2$ a 4!
- We select value 4 for the first max-flow algorithm iteration.
- Maximum flow 18, which equals to the total execution time of all jobs in the hyperperiod.
- Flow of edges represent a schedule of the tasks:



Example – graph



When tasks are not independent

- Jobs with **precedence constraints** are easy to take into account.
  - Precedence constraints of "J_i precedes J_k" can be easily satisfied by setting release time r_i at or before release time r_j and setting deadline d_i at or before d_k.
  - If jobs J_i a J_k are scheduled in the wrong order, we can switch them in the schedule.
- **Restriction of preemption** (critical sections) is more ambitious.
  - Manual modification of the found schedule to remove preemptions in non-preemptive regions.
  - Unfortunately, there is no efficient optimal algorithm for this – the problem is NP-hard, even on one processor.

Outline

Cyclic Executive

- Fancy name for a scheduler that executes the frame-based schedule.

Pseudo-code for Cyclic Executive (with interrupts)

Input: Stored schedule: $L(k)$ for $k = 0, 1, \ldots, F-1$;
Initialization
    actual time t:=0; actual frame k:=0;
    set timer interrupt period to f
end
In timer interrupt do
    current_block := $L(k)$;
    $t := t + 1$; $k := t \bmod F$;
    if the last job is not completed (overrun), take appropriate action (error);
    Setup execution of jobs in current_block
        (e.g. by modifying the return address(es) on the stack)
end

Cyclic Executive (with OS)

Pseudo-code for Cyclic Executive (with OS)

Input: Stored schedule: $L(k)$ for $k = 0, 1, \ldots, F-1$;
Aperiodic job queue
Task CYCLIC_EXECUTIVE (highest priority OS task)
current time t:=0; actual frame k:=0;
repeat forever
wait until $t \cdot f$;
current_block := $L(k)$;
$t := t + 1$; $k := t \bmod F$;
if the last job is not completed (overrun), take appropriate action (error);
release all the jobs in the current_block;
wait until the jobs complete (or use a scheduling server);
while aperiodic job queue is not empty **do**
wake up the job at the head of the aperiodic job queue;
wait until the jobs complete (or use a scheduling server);
remove the aperiodic job from queue;
end
end
End CYCLIC_EXECUTIVE

Cyclic Executive implementation – Summary

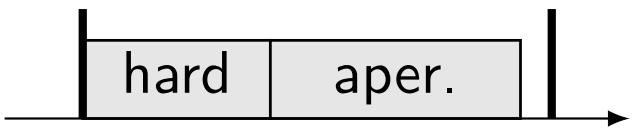
- Main program loop that calls functions (tasks) at appropriate time
  - Cannot prevent task overruns
  - Cannot preempt execution of aperiodic jobs.

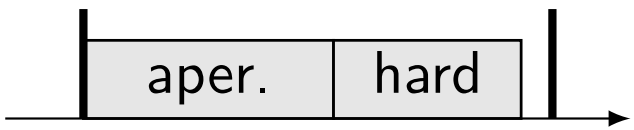
  P
- In timer interrupt or as highest priority task in an OS
  - Task overruns can be detected and handled appropriately
  - Aperiodic jobs can be preempted

Outline

Improving response times of aperiodic jobs

- Intuitively, it makes sense to give hard real-time jobs higher priority than aperiodic jobs.
- However, this may lengthen the response time of an aperiodic job.



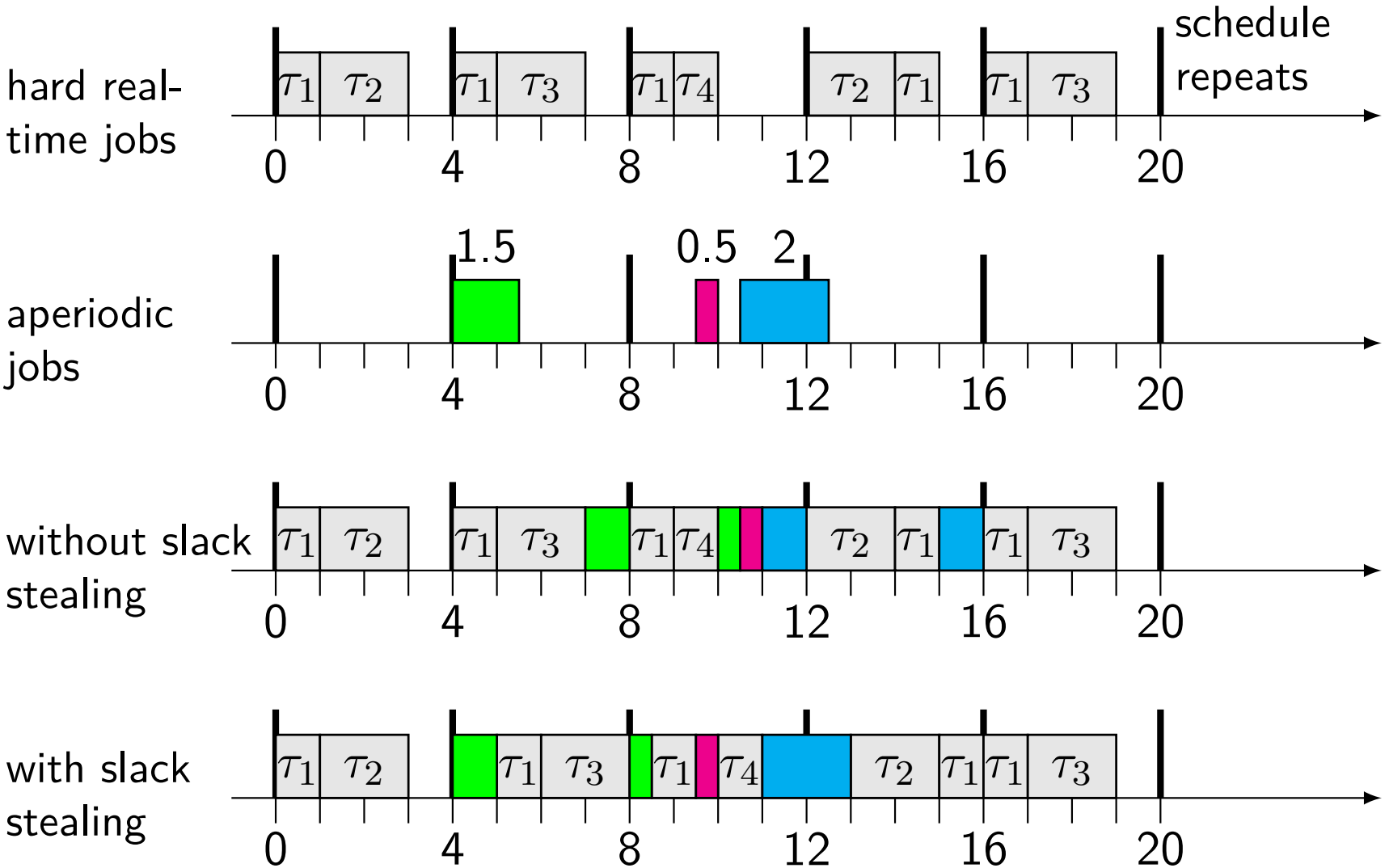


hard deadline is still met, but aperiodic job completes sooner

- Note that there is no point in completing a hard real-time job early, as long as it finishes by its deadline.

Slack Stealing – Example

(CZ: Kradení rezervy)



Slack stealing

(CZ: Kradení rezervy)

- Let the total amount of time allocated to all the jobs scheduled in frame k be x_k.
- **Definition:** slack available at the beginning of frame k is $f - x_k$.
- Scheduler change :
  - If aperiodic job queue is not empty, run aperiodic jobs in every frame with non-zero slack.

Slack stealing implementation

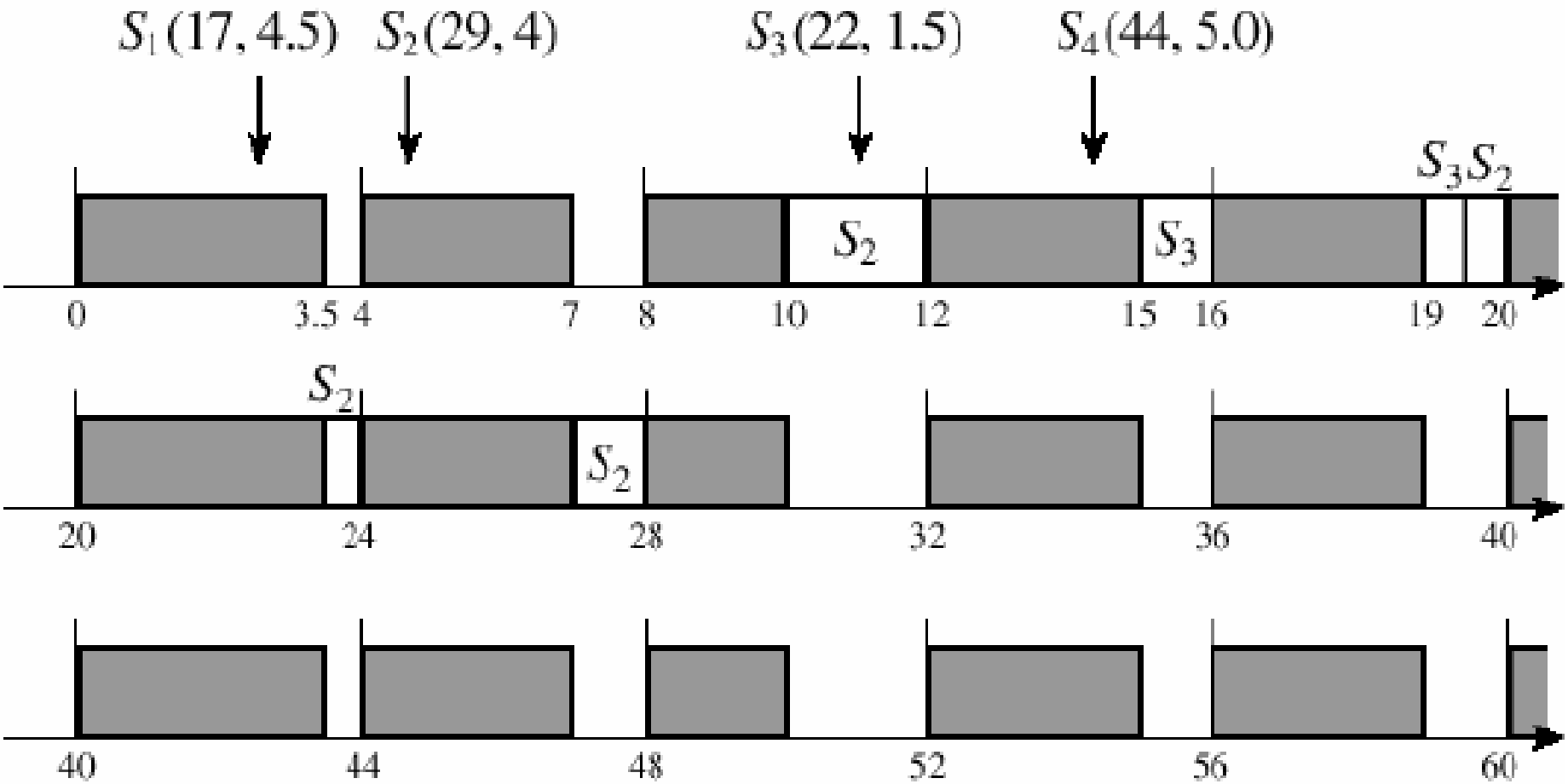
- Precomputed slack table – entries depend only on static values
- To maintain the value of remaining slack use **interval timer**.
  - Start the timer when running the aperiodic job. If the timer expires, a periodic job has to be executed.
  - **Problem:** Older operating systems do not support high-resolution timers. In VxWorks, this can be partially solved by sysClkRateSet().

Scheduling sporadic jobs

- Sporadic jobs arrive at arbitrary times.
- They have hard deadlines.
- Implies we cannot hope to schedule every sporadic job.
- When a sporadic job arrives, the scheduler performs an admission test (acceptance test) to see if the job can be completed by its deadline.
- We must ensure that a new sporadic job does not cause a previously-accepted sporadic job to miss its deadline.
- We assume sporadic jobs are prioritized on an earliest-deadline-first (EDF) basis.

Example

Frame size is 4, gray rectangles are periodic tasks
S_1 – S_4 are sporadic tasks with parameters (D_i, C_i)



Example (continued)

- S_1 is released in time 3.
  - Must be scheduled in frames 2, 3 and 4.
  - Acceptance test – at the beginning of frame 2:
    - Slack time is 4 which is less that execution time – **job is rejected**.
- S_2 is released in time 5.
  - Must be scheduled in frames 3 through 7.
  - Acceptance test – at the beginning of frame 3:
    - Slack time is 5.5 – **job is accepted**.
    - First part (2 units) executes in current frame.
- S_3 is released in time 11.
  - Must be scheduled in frames 4 and 5.
  - S_3 runs ahead of S_2.
  - Acceptance test – at the beginning of frame 4:
    - Slack time is 2 (enough for S_3 and the part of S_2) – **job is accepted**.
    - First part (1 unit) S_3 executes in current frame, followed by second part of S_2.
- S_4 is released in time 14.
  - Acceptance test – at beginning of frame 5:
    - Slack time is 4.5 (accounted for slack committed by S_2 and S_3) – **job is rejected**.
    - Remaining portion of S_3 completes in current frame, followed by the part of S_2.
    - Remaining potions of S_2 execute in the next two frames.

Admission test

- Let $\sigma(i, k)$ be initial slack in frames i through k, where $1 \leq i \leq k \leq F$. This depends only on periodic tasks and their parameters. These are known in advance.
- Assume the admission test is executed at the beginning of frame t for a just arrived sporadic task S with deadline D and execution time C. Further assume that D happens in frame $l+1$, i.e. S must be finished before end of frame l.
- The **current** total amount of slack time $\sigma_c(t, l)$ in frames t through l can be computed as

$$\sigma_c(t, l) = \sigma(t, l) - \sum_{D_k \leq D} C_k - \xi_k$$

  - We sum over earlier accepted sporadic jobs with equal or earlier deadline.
  - ξ_k denotes the execution time of S_k that has been completed at the beginning of frame t.

Admission test (cont.)

Admission test algorithm

Input: current frame t, new sporadic job $S_n = (C_n, D_n)$,
precomputed slack table σ_c,
set of already accepted jobs $\{S_k | S_k = (C_k, D_k, \xi_k, \sigma_k)\}$

if $\sigma_c(t, l) \leq C_n$ **then** reject S_n
else
    store $\sigma_n := \sigma_c(t, l) - C_n$ as slack time of job S_n;
    foreach earlier accepted sporadic job S_k with $D_k > D_n$:
        if $\sigma_k - C_n < 0$ **then**
            reject S_n
            return
        end
    end
    Insert $S_n = (C_n, D_n, 0, \sigma_n)$ into $\{S_k\}$; // *accept*
end

Execution of sporadic jobs

- Accepted sporadic jobs are executed in the same manner as aperiodic jobs in the original algorithm without slack stealing.
- **Difference:** Aperiodic job queue is FIFO ordered, whereas sporadic job queue is EDF ordered.
- Aperiodic jobs run only if sporadic job queue is empty.
- As before, slack stealing could be used when executing aperiodic jobs (in which case, some aperiodic jobs could execute when the sporadic job queue is not empty).

Outline

Practical considerations

- Handling frame overruns
  - **Main issue:** Should offending job be completed or aborted? It depends on the application.
- Mode changes
  - Changing the set of running tasks (schedule table).
  - Example: Airplane needs different sets of tasks for: taking off, cruise mode, landing
  - May be implemented as aperiodic or sporadic job that makes the change. If sporadic, it can be rejected!

Pseudocode for "mode changer"

Task Mode changer
    Fetch the list of tasks to be deleted;
    Mark each periodic task included in this list;
    Inform cyclic executive about mode change start;
    Fetch the newTaskList of periodic tasks to be executed in newMode;
    Allocate memory for each task from newTaskList and crate each of these task;
    Read new schedule;
    Perform admission test for each sporadic job in the system according to the new schedule
    If all sporadic jobs are accepted in the new schedule **then**
        Inform cyclic executive to use the new schedule;
    else
        Calculate maximum completion time among all accepted sporadic jobs. (latestCompletionTime
        Inform cyclic executive to use the new schedule in time latestCompletionTime);
    end
end Mode changed

Outline

Cyclic executives: pros and cons

Main advantage: Cyclic executives are very simple – a single table is all what is needed, no complex run queues etc.

- For example, there is no need for concurrency control and task synchronization (e.g. semaphores). In the simplest case, there are no tasks and processes and everything is just a function call.
- Can be validated, tested and certified with very high confidence.
- Certain scheduling anomalies will not occur.
- For these reasons, cyclic executives are predominant approach in many safety critical applications (e.g in airplanes, trains, etc.)

Pros and cons (continued)

Disadvantages of cyclic executives:

- **Very "fragile":** Any change, no matter how trivial, requires a new schedule table to be computed!
- Slicing one function into several smaller can be error-prone.
- Release times of all jobs must be fixed, i.e. "real-world" sporadic tasks are difficult to support.
- Temporal parameters must be essentially multiples of f.
- F can be very high.
- All combinations of period tasks that may execute together must a priori be analyzed.