Real-time systems on a distributed platform

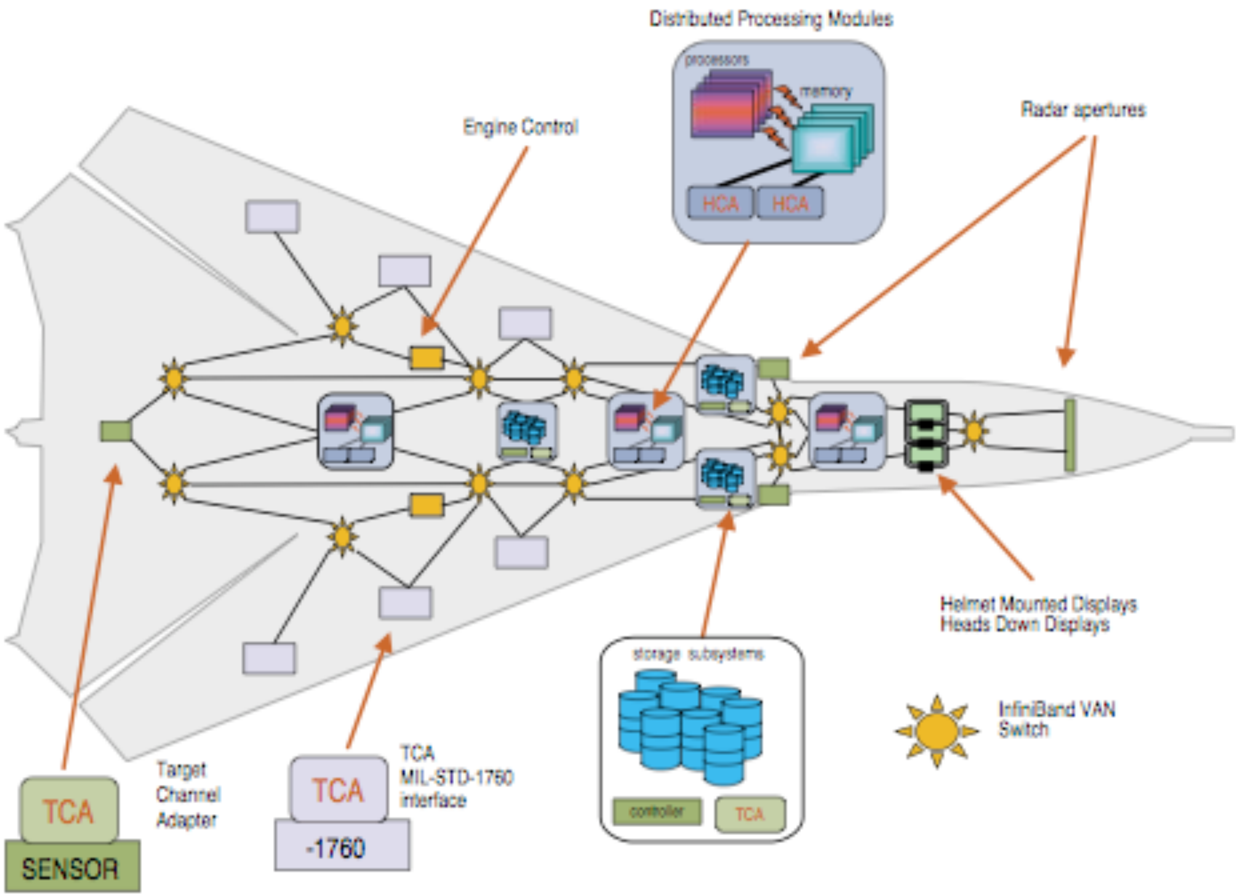
Multi-stage systems
Schedulability analysis for distributed systems
Restrictions that make analysis easier

Lecture overview

- So far we have spent a lot of time discussing small (uniprocessor) systems
- We studied the behavior of periodic tasks on uniprocessors subject to fixed and dynamic priority policies
- But many computer systems run on distributed components

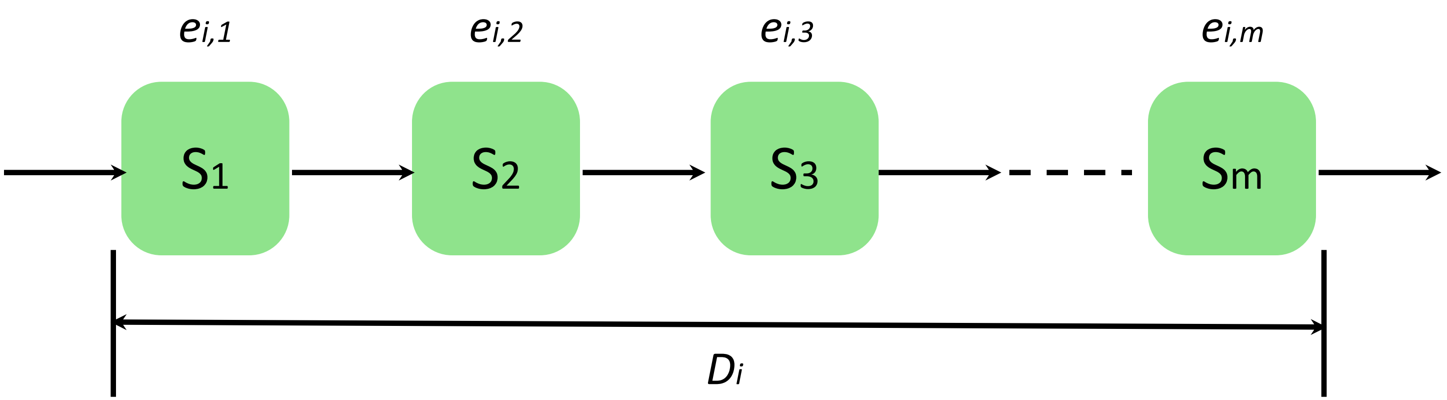
- In this lecture we will study **distributed real-time systems**
- Understand the basic elements of schedulability analysis for these systems

Example: Avionics systems
Distributed Integrated Modular Architecture (ARINC 651)



Data is processed at multiple nodes
One task in this application may have multiple stages
The entire sequence, however, has to meet a deadline

Schematic of a distributed system

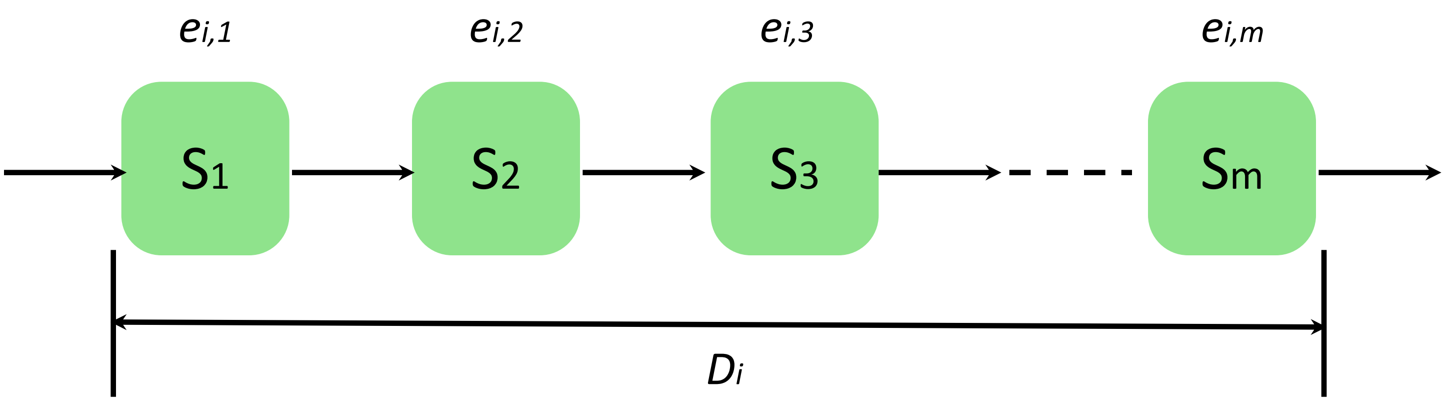


Task T_i has to be processed in ***m*** stages
The end-to-end deadline for the task is D_i
The task is periodic with period P_i
The execution time of the task at stage j is $e_{i,j}$

Deadlines in a distributed system

- Typically: **relative deadlines are greater** than the periods of the tasks
- Sometimes, relative deadlines >> periods
- Example: video transmission in aircraft might involve capturing images at **24 frames per second** (period = 1/24 = 41.7ms) but a **deadline of 250ms** is sufficient for the captured image to reach the pilot
  - Why? Human visual reaction time is about 250ms, and in all situations a total response time of 250ms (time to deliver data + reaction time) is typically sufficient
  - In this example, $D_i > 3P_i$

Applying known techniques



- Treat each stage independently

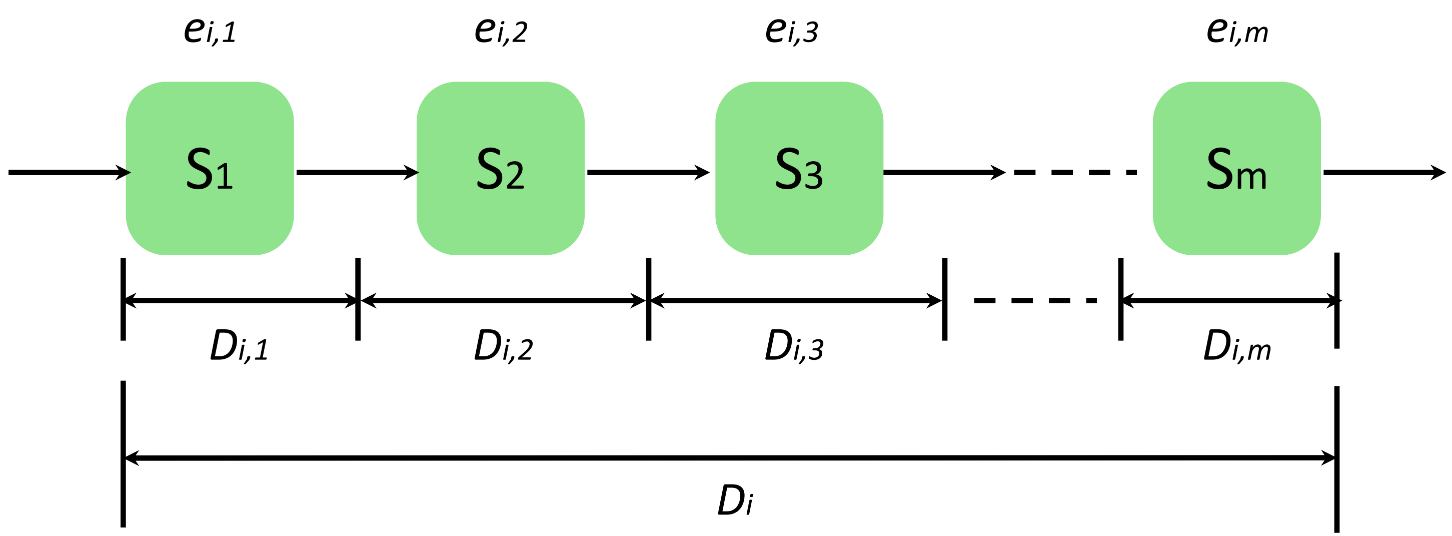
  (1) We need tasks to be periodic at each stage

  (2) For a given end-to-end deadline D_i: We need to set a relative deadline $D_{i,j}$ for stage j such that

$$\sum_{j=1}^{m} D_{i,j} = D_i$$

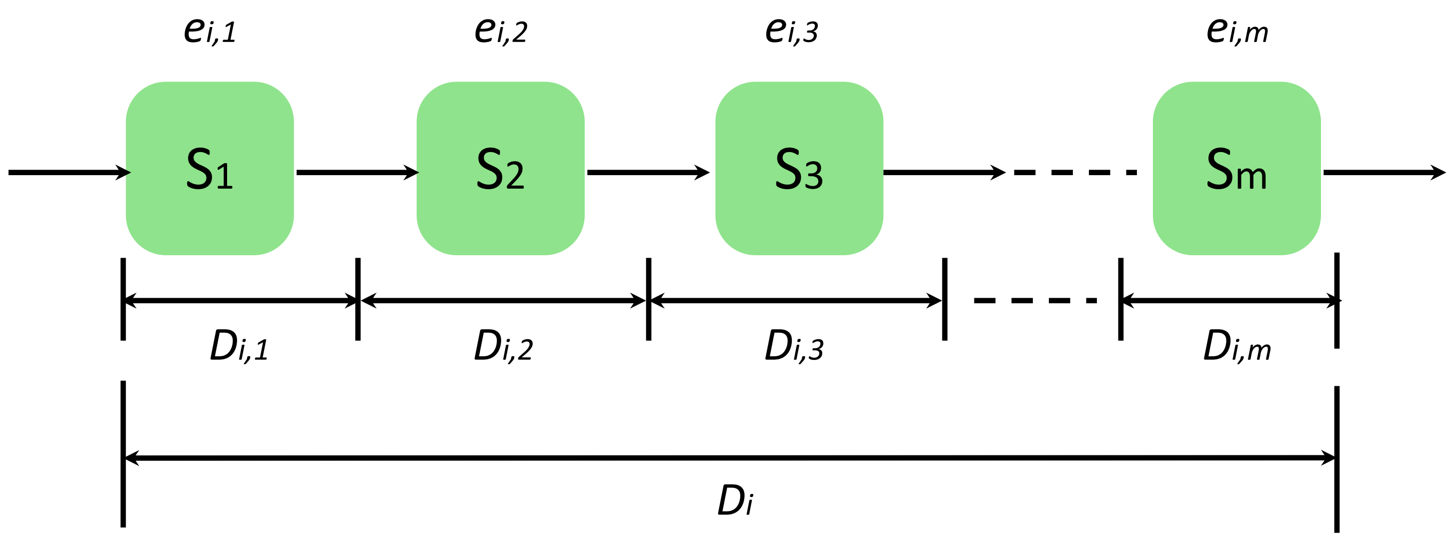
- Then we can apply known results to verify that per-stage deadlines are met

Deadline distribution



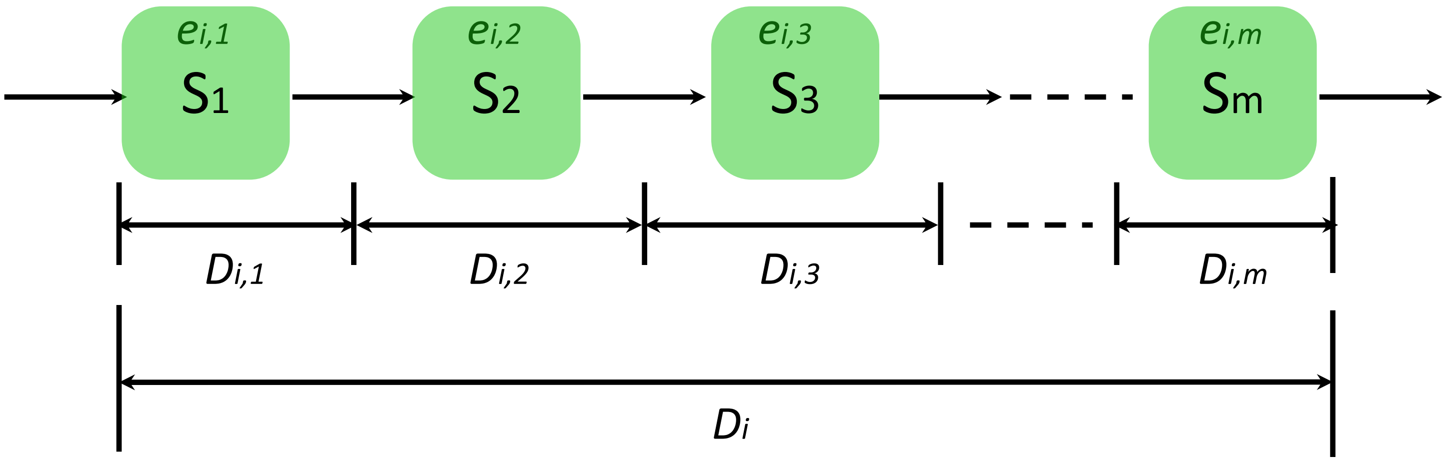
- How do we distribute the end-to-end deadline over multiple stages?
- **Hard problem:** no efficient method to determine the optimal distribution (how do we define an optimal per-stage deadline assignment?)

Deadline distribution



- We can use **heuristics**: their performance might vary based on the taskset being scheduled
- Some examples
  - Even distribution: $D_{i,j} = D_i/m$
  - Proportional distribution: $D_{i,j} = D_i \times e_{i,j}/\left(e_{i,1} + e_{i,2} + \cdots + e_{i,m}\right)$

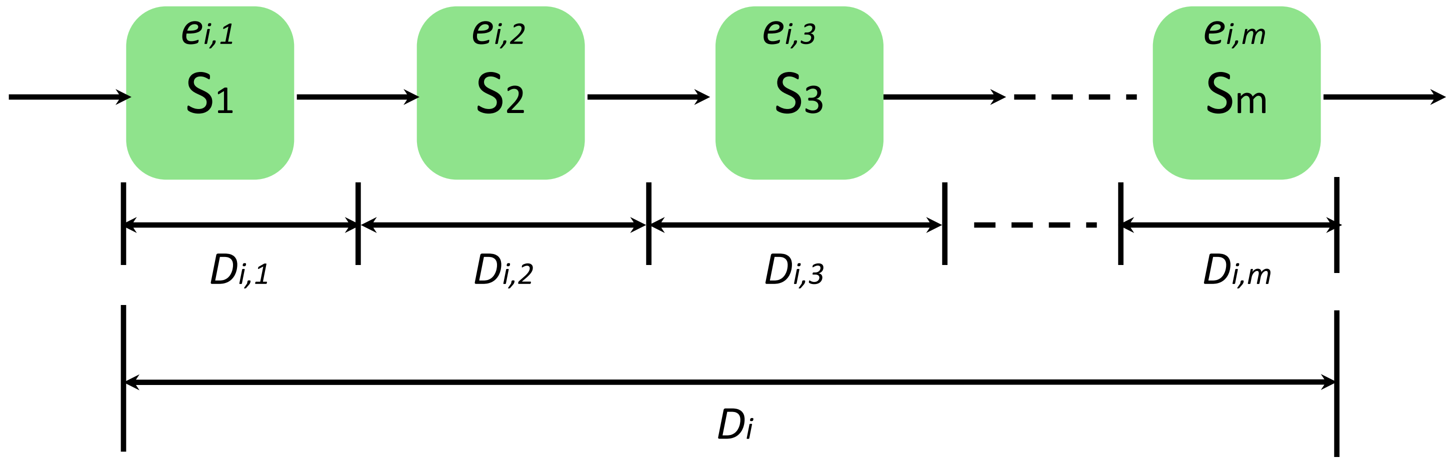
We need tasks to be periodic at each stage



Suppose $P_1 = 6$

- How do we ensure that tasks arrive at each stage periodically? Consider if:
  - job 1 of task 1 finishes at stage 1 at time 4
  - job 2 of task 1 finishes at stage 1 at time 9 (inter-arrival time of 5 at stage 2)
  - job 3 of task 1 finishes at stage 1 at time 17 (inter-arrival time of 8 at stage 2)
  - job 4 of task 1 finishes at stage 1 at time 23 (inter-arrival time of 6 at stage 2)
- **Our theory so far assumes that job arrivals are strictly periodic if we want a schedulability guarantee**
- We could ensure that **a job reaches the next stage only at the relative deadline of the previous stage**: requires extra mechanisms (overhead at the OS level)

We need tasks to be periodic at each stage



- How do we ensure that tasks arrive at each stage periodically?
- We could ensure that **a job reaches the next stage only at its relative deadline of the previous stage**: requires extra mechanisms (overhead at the OS level)
- Alternatively, we could also ensure that each job was released to the next stage only after the worst-case response time
- Compute WCRTs for each stage
- If a job completes early, buffer it and release it to the next stage only when the WCRT is reached

Synchronization for distributed real-time systems

- **Synchronization protocols** for distributed real-time systems address how tasks flow from stage to stage
- Requirements of a synchronization protocol
  - Enforce precedence constraints
  - Allow schedulability analysis
  - Low end-to-end worst-case response time
  - Low overhead
  - Low (end-to-end) average response time

Synchronization for distributed real-time systems

- **Greedy protocol (Direct Synchronization)**
- Release a job to the next stage as soon as it completes at the current stage
- Job arrivals might not be periodic (with the exception of the first stage)
  - Difficult for schedulability analysis (remember deferrable server?)
  - Higher priority tasks may arrive early: increased worst-case response time for lower priority tasks

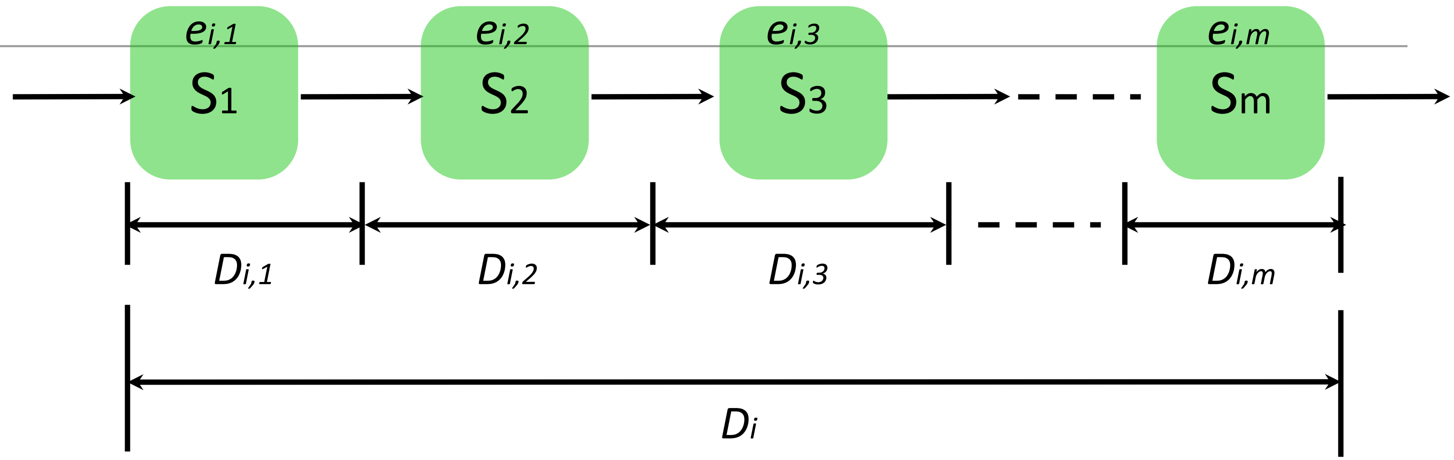
Synchronization for distributed real-time systems

- **Phase modification protocol**
- Release a job to the next stage only when the worst-case response time for the job is reached at the current stage
  - Let us suppose that the worst-case response time of task T_i at stage j is $R_{i,j}$
  - Let $\Phi_{i,1} = \Phi_i$, and $\Phi_{i,j+1} = \Phi_{i,1} + \sum_{k=1}^{j} R_{i,k}$
  - Jobs of T_i are released to stage j+1 at times $\Phi_{i,j+1}, \Phi_{i,j+1} + P_i, \Phi_{i,j+1} + 2P_i, \ldots$
- Require upper-bound on response times of tasks
- If a subtask overruns, precedence constraints might be violated!
- Require global clocks (subtle point: each stage should be time synchronized)
  - Allows schedulability analysis
  - Low worst-case response time
  - Overhead: global clock, buffering requirement

Synchronization for distributed real-time systems

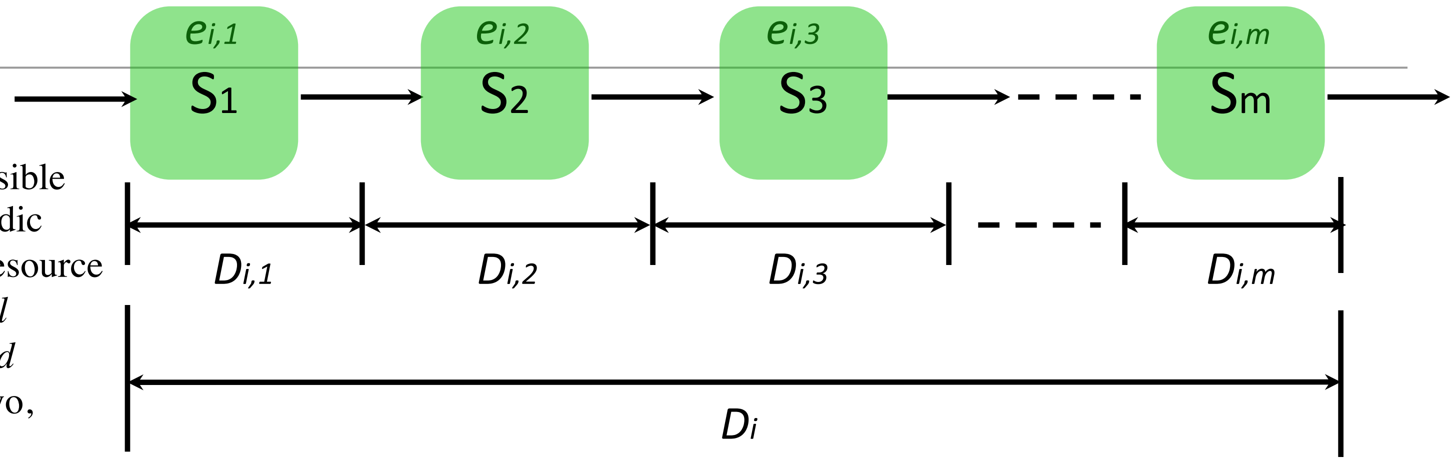
- **Release guard protocol**
- Relax the requirement on global clocks
- **Idea:** Every two consecutive instances of any subtask of T_i are released *at least* P_i time units apart
- At each stage, release an instance of a T_i only if the previous instance of T_i was released at least P_i time units earlier
- Associate each subtask with a "release guard" variable $g_{i,j}$ (earliest allowed release time for next instance of $\boldsymbol{T}_{i,j}$)
  - If $J_{i,j-1}$ finishes at or after $g_{i,j}$, release $J_{i,j}$ immediately
  - Otherwise release $J_{i,j}$ *at* $g_{i,j}$
- Release guard update rules (initially $g_{i,j} = 0$)
  - When an instance of subtask $\boldsymbol{T}_{i,j}$ is released, set $g_{i,j}$ = `current_time` + P_i
  - Update $g_{i,j}$ to `current_time` if it is an idle point (i.e., if all jobs released before `current_time` at stage j completed)

Buffering and its problems



- The difficulty with ensuring periodicity in a multi-stage (distributed) system is the need for buffering
- Most current distributed real-time systems do make use of buffering because they were designed when better tests were not known
- It is easier to build systems if we did not have to buffer
- But a challenge arises because of the loss of periodicity
- Is this a real problem? **Can we determine if tasks are schedulable even if we assume they are aperiodic?**
- Our study till date assumes workload (or utilization) is easy to compute because tasks were periodic. How does this change with aperiodic tasks?

Aperiodic workload: The Stage Delay Theorem



T. Abdelzaher et al. A feasible region for meeting aperiodic end-to-end deadlines in resource pipelines. In *International Conference on Distributed Computing Systems*, Tokyo, Japan, March 2004.

$S(t)$: set of tasks that have arrived but whose deadlines have not expired at time t (the set of *current tasks*)

$$S(t) = \{T_i : r_i \leqslant t < r_i + D_i\}$$

Synthetic (instantaneous) utilization at stage j: $U_j(t) = \sum_{T_i \in S(t)} e_{i,j}/D_i$

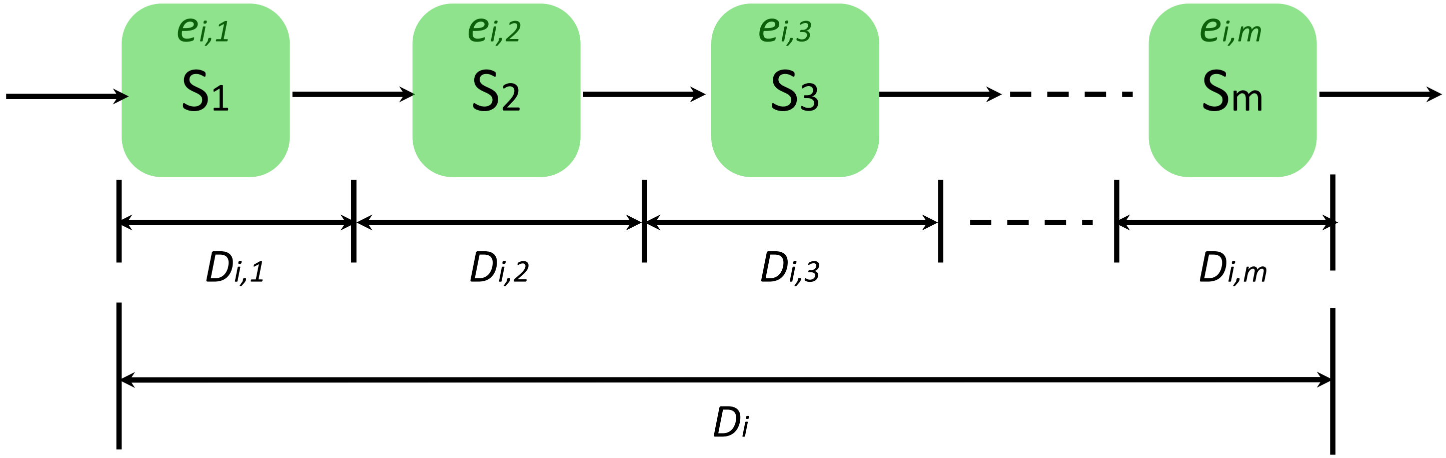
Stage Delay Theorem: Under DM, all end-to-end deadlines are met if

$$\sum_{j=1}^{m} \frac{U_j(1-U_j/2)}{1-U_j} \leqslant 1$$

U_j: any upper bound on $U_j(t)$

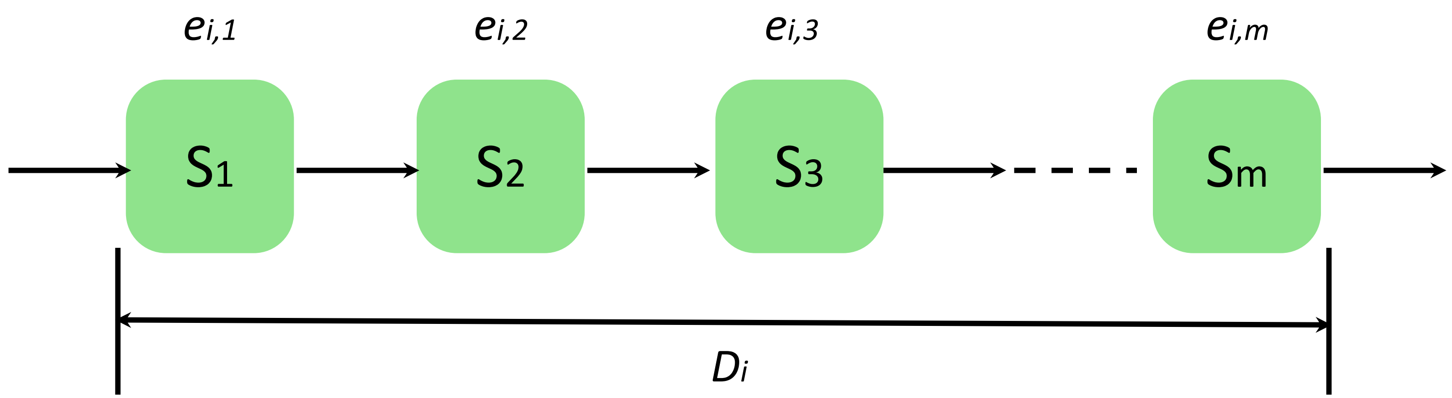
U_j is such that $U_j(t) \leqslant U_j$ for all $t \geqslant 0$

Highlights



- Many modern real-time systems are built on a distributed platform because it is not possible to perform all operations on one processor
- **Tasks thus flow through multiple stages and each instance of a task needs to meet an end-to-end deadline**
- It is possible to guarantee schedulability by setting **intermediate (or per-stage) deadlines**
- We need to identify a heuristic to set the intermediate deadlines
- Then we use the standard uniprocessor analysis for each stage
- Setting intermediate deadlines and requiring periodicity at each stage calls for buffering: **buffering adds to complexity and overhead in a system**

Real-time communication



- What about communication? How does information flow from one stage to the next?
- Several possibilities
  - Communication is instantaneous (Unlikely!)
  - Communication has bounded latency (Somewhat more likely. Add communication latency and then ensure that deadlines are met.)
  - Treat the communication channel as a stage (Most general. Better way to understand distributed systems.)

Communication media

- Data buses
- Ethernet
- ATM

- If these media support prioritized scheduling of messages (packets), we can derive latencies introduced because of communication

More details:

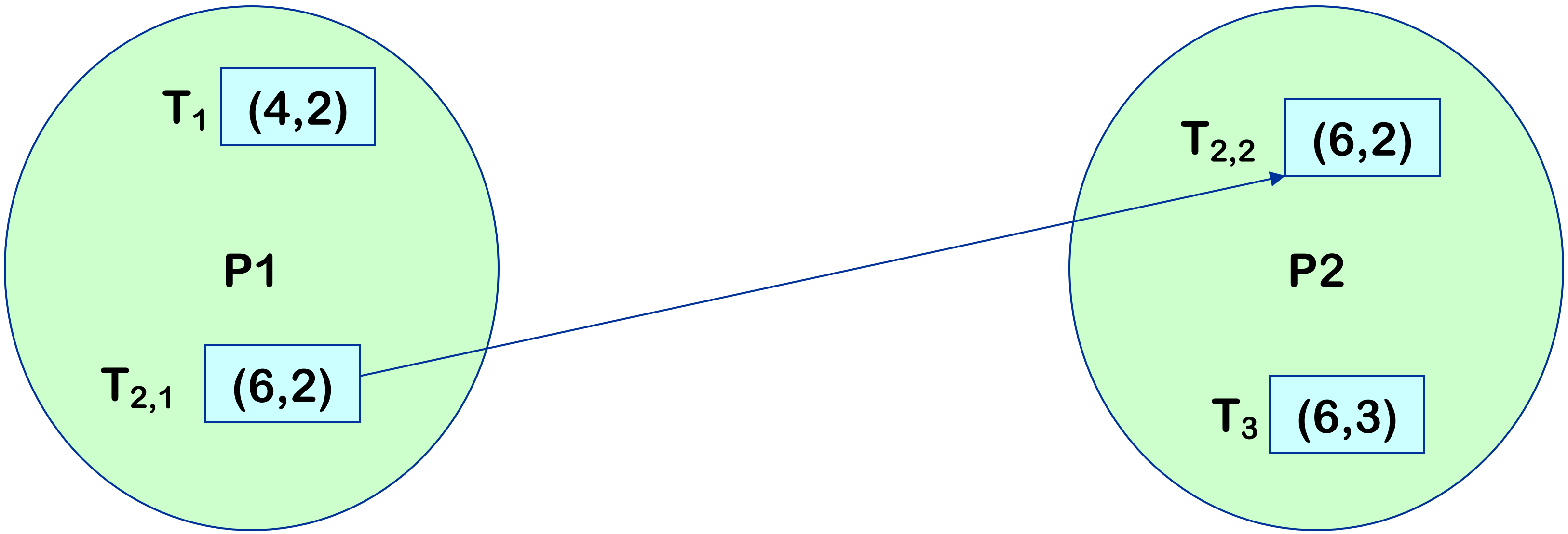
Optional reading;

Improvements to protocols are discussed.

Synchronization Protocols

- Goal: Reduce end-to-end response times (EER)
- Direct Synchronization (DS) Protocol
  - Simple and straightforward
- Phase Modification (PM) Protocol
  - Proposed by Bettati
- Release Guard Protocol
  - Proposed by Sun

Synchronization Protocol - Example



$T_{i,j}$ – j^{th} subtask of task T_i

Task T3 has a phase of 4 time units

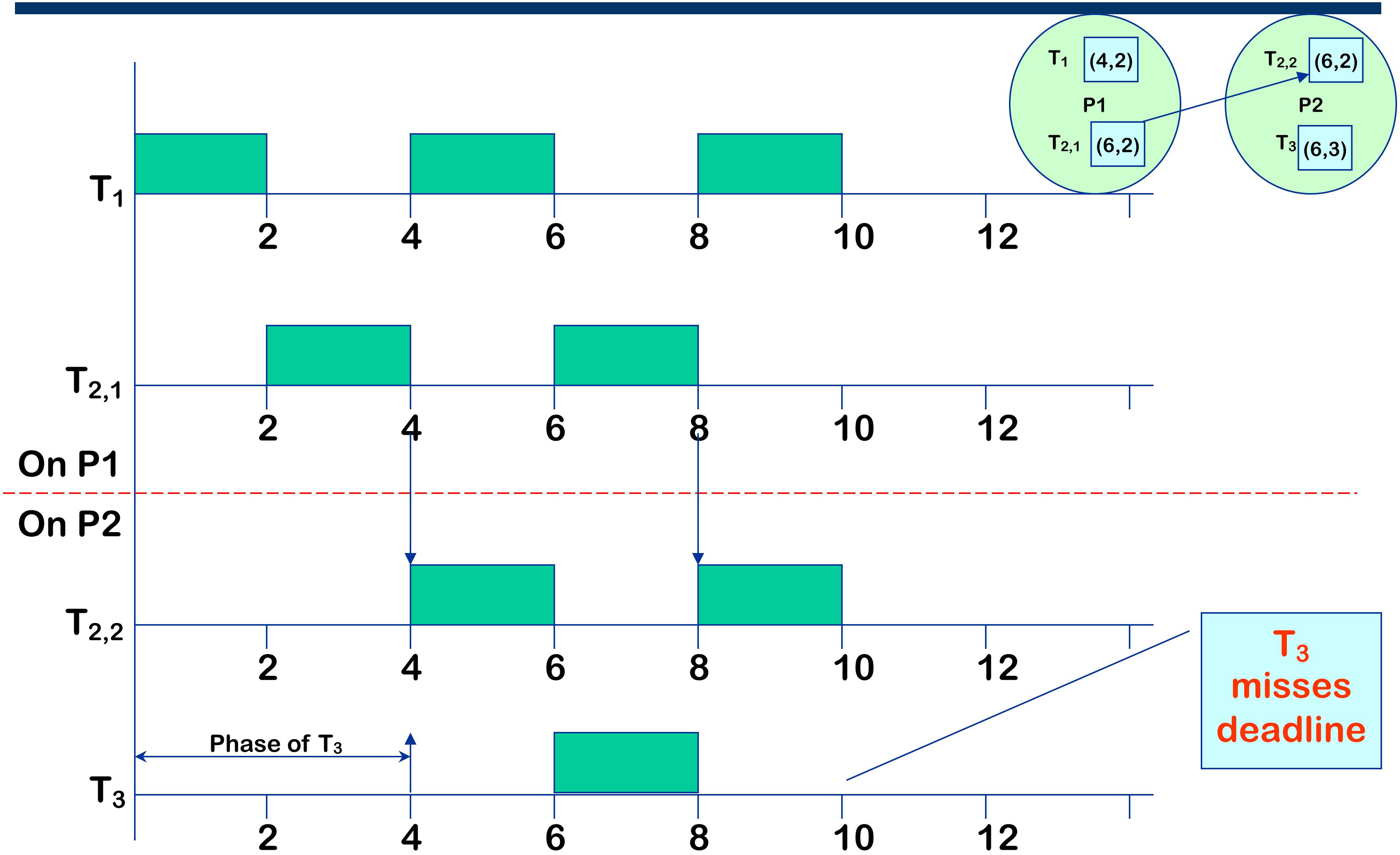
(period,execution time)

Period = relative deadline of parent task

Direct Synchronization Protocol

- Greedy strategy
- On completion of subtask
  - A synchronization signal sent to the next processor
  - Successor subtask competes with other tasks/subtasks on the next processor

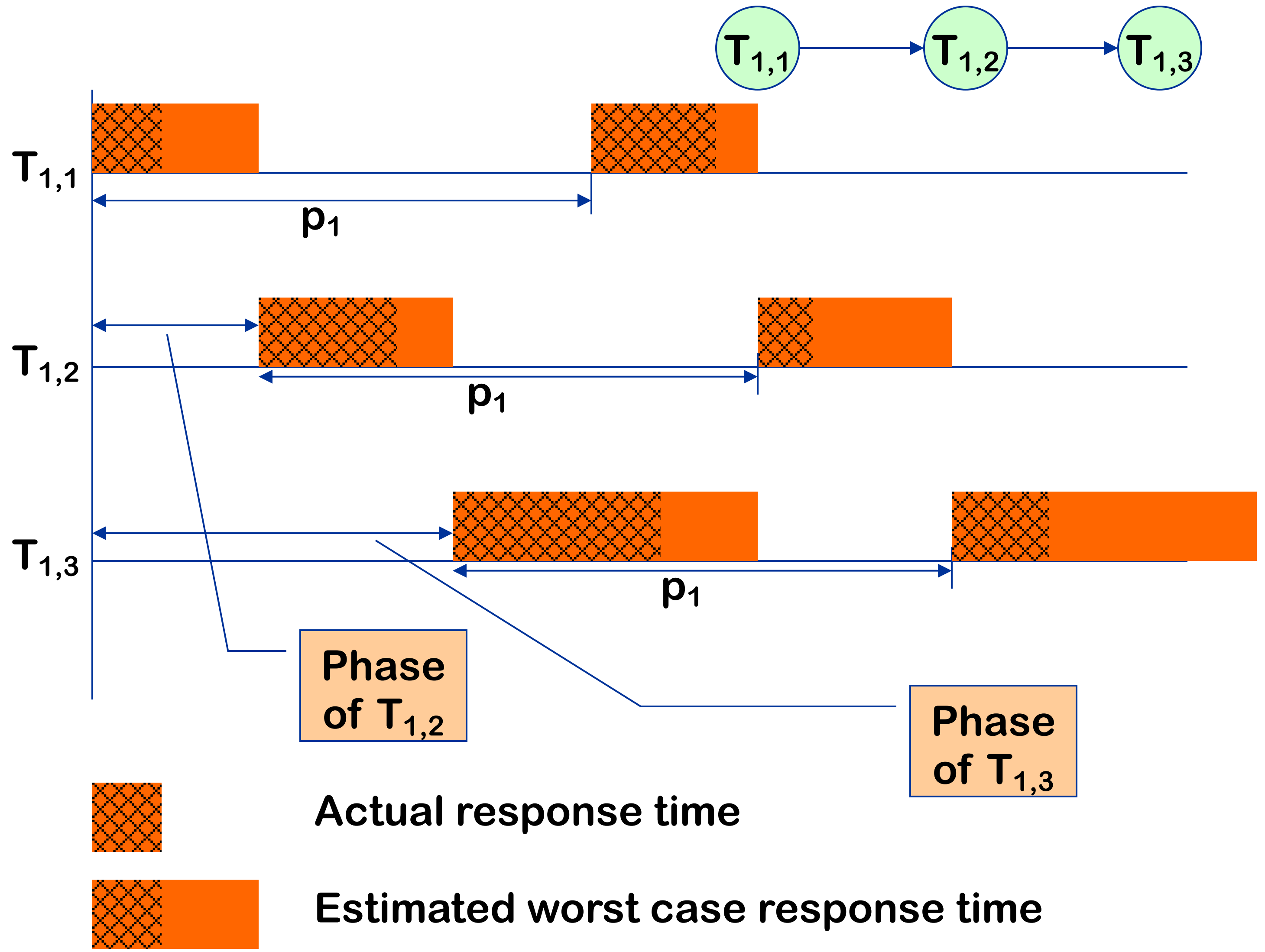
Direct Synchronization Illustrated



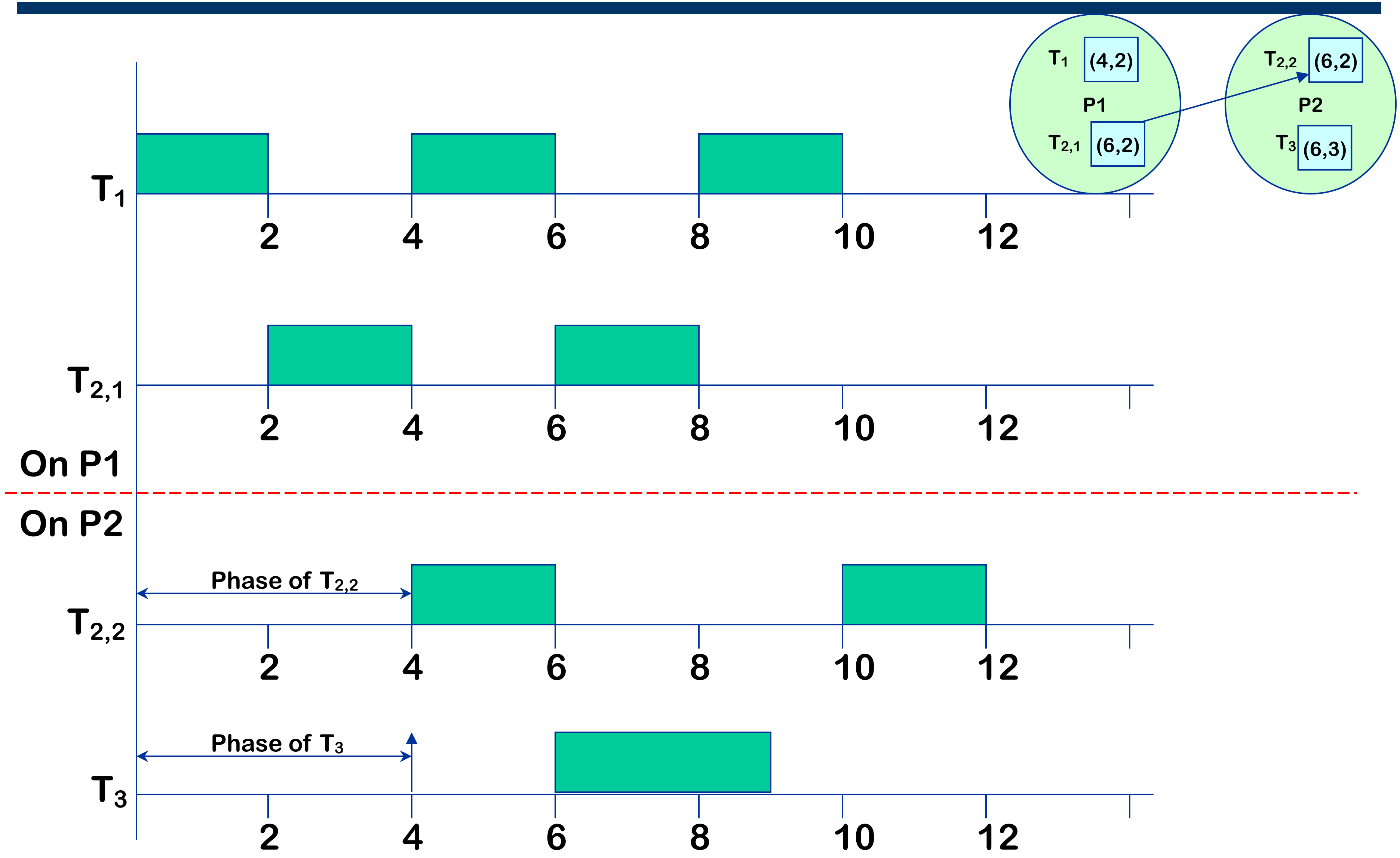
Phase Modification Protocol

- Proposed by Bettati
- Release subtasks periodically
  - According to the periods of their parent tasks
- Each subtask given its own phase
- Phase determined by subtask precedence constraints

Phase Modification Protocol Illustrated (1/2)



Phase Modification Protocol Illustrated (2/2)



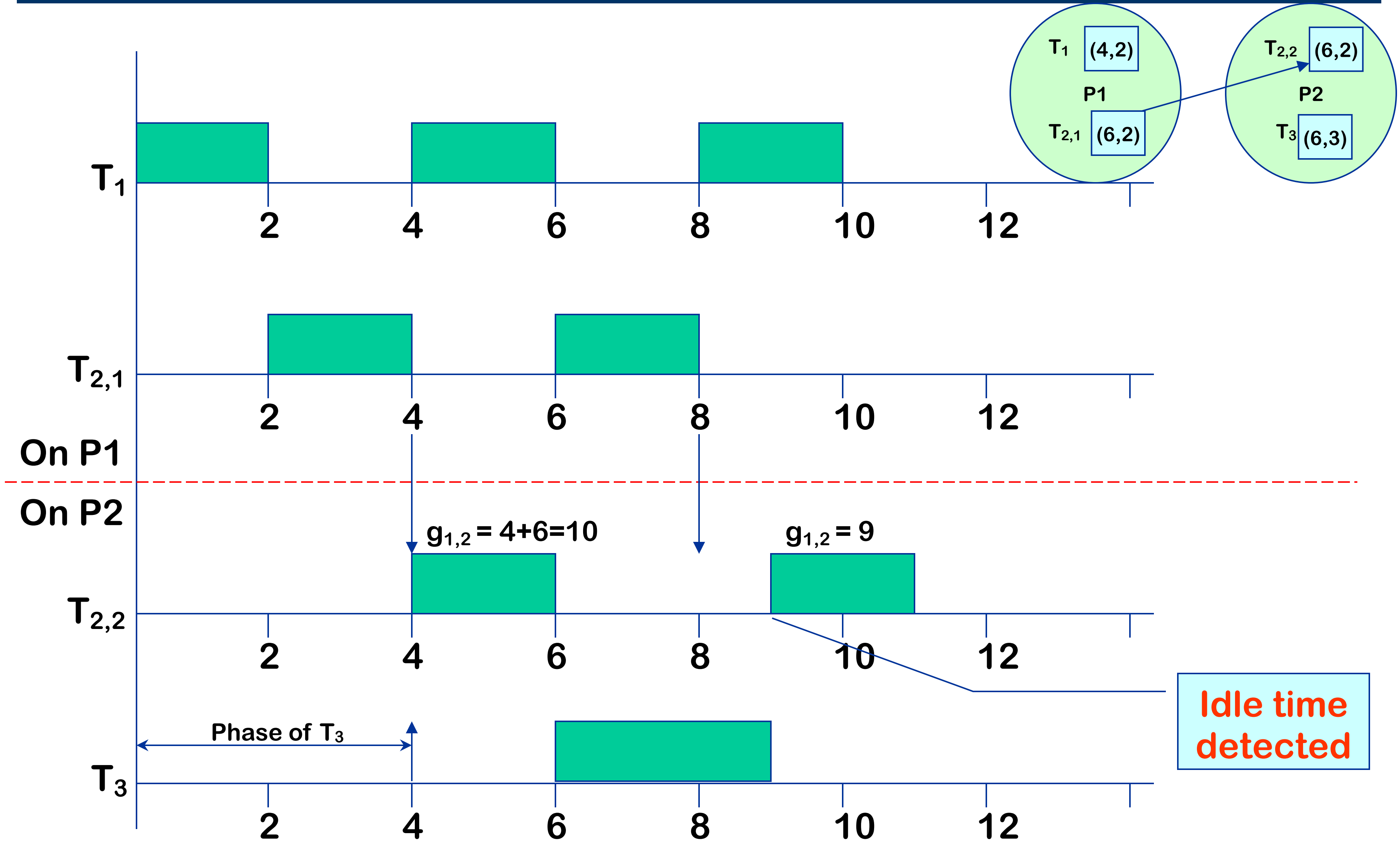
Phase Modification Protocol - Analysis

- Periodic Timer interrupt to release subtasks
- Centralized clock or strict clock synchronization
- Task overruns could cause Precedence constraint violations

Release Guard Protocol

- Proposed by Sun
- A guard variable – *release guard* - associated with each subtask
- Release guard used to control release of each subtask
  - Contains next release time of subtask
- Synchronization signals just like MPM
- Release guard updated
  - On getting synchronization signal
  - During idle time

Release Guard Protocol Illustrated



Release Guard Protocol - Analysis

- Shares the same advantages as MPM
- Upper bound on EER still the same as MPM
  - Since upper bound on release time enforced by release guard

$$\sum_{k=1}^{n_i} R_{i,k}$$

$R_{i,k}$ is the response time of the k^{th} subtask of T_i
n_i is the number of subtasks for the task T_i

- Lower bound on EER less than that of MPM
  - If there are idle times
  - Results in lower average EER