Embedded Systems Design and Modeling

Chapter 12
Scheduling

Definitions

- Scheduling:
  - The process of determining which job/task/thread/process to execute
- Considerations:
  - Preemptive vs. non-preemptive scheduling
  - Periodic vs. aperiodic tasks
  - Fixed priority vs. dynamic priority
  - Priority inversion anomalies
  - Other scheduling anomalies

Preemptive Scheduling

- Non-preemptive:
  - Once a task is started, it runs until it ends or has to do some I/O
- Preemptive:
  - A task may be stopped to run another
  - Assumes all threads have set priorities
    - either statically assigned (constant for the duration of the thread)
    - or dynamically assigned (can vary)
  - Assumes that the kernel keeps track of which threads are ready to execute
  - At any instant, the enabled thread with the highest priority is executing
  - Whenever any thread changes priority or its status, the kernel can dispatch a new thread
  - Incurs overhead and implementation complexity
  - Has better schedulability and easier analysis
- Which one? Non-preemptive has very restrictive constraints

Non-Preemptive Constraint

- Consider N tasks, with task tj getting ready every pj, and needs ej time during interval pj
- Then pj has to be >= e1 + e2 + ... + e_N
- In other words, period of every thread >= sum of computation times!
- Not practical, may have to switch to preemptive solutions

Multitasking Options

1. Cyclic executive
   - Similar to TDMA (Time Division Multiple Access) scheduling
   - Also called “static table-driven scheduling” or “timeline scheduling”
   - Requires static schedulability analysis
   - The resulting schedule or table is used at run time
2. Event-driven non-preemptive
   - Tasks are represented by functions that are handlers for events
   - Next event processed after function for previous event finishes

Multitasking Options (Cont'd)

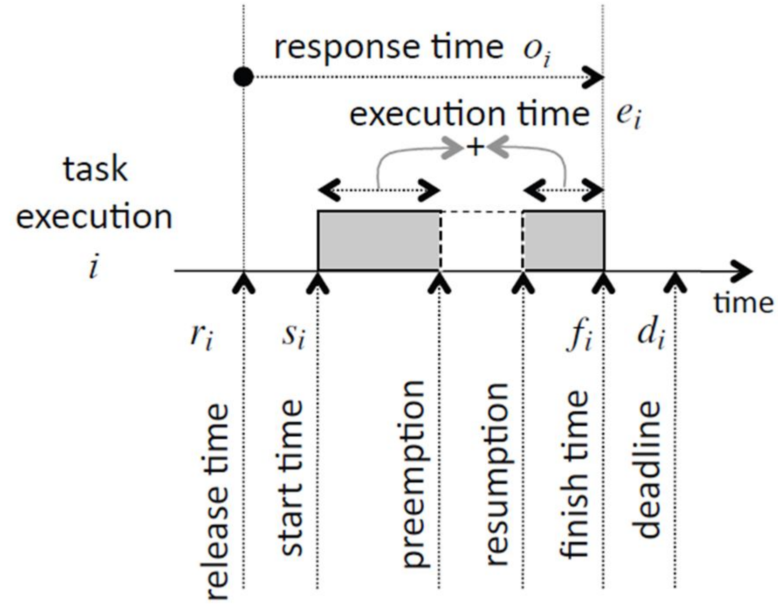
3. Thread-based
   - Supports both static and dynamic priority
   - Supports preemptive and non-preemptive
   - Requires static schedulability analysis
   - No explicit schedule constructed: at run time tasks are executed "highest priority first"
   - Flavors
     - Rate Monotonic (RM)
     - Deadline Monotonic (DM)
     - Earliest Deadline First (EDF)
     - etc.

Comparison Metrics

- Need metrics to compare performance
- Static case:
  1. First and foremost, off-line schedule has to meet all the deadlines!
  2. Secondary metric(s):
     - Maximize average earliness
     - Minimize average lateness
- Dynamic case:
  - No prior guarantee that deadline would be met
  - Metric: maximize number of arrivals that meet the deadline

Definitions

- Feasible schedule:
  - For all tasks $f_i \le d_i$
- Lateness:
  - For each task: $f_i - d_i$
  - For a feasible schedule:
    - For all tasks *lateness* ≤ 0
- Earliness:
  - For each task: $d_i - f_i$
- Makespan:
  - For each task: $\max f_i - \max r_i$



Thread-Based Scheduling

- Definition: Set of rules to determine the specific thread to be executed at a particular moment
- One possibility: Preemptive & priority driven
  - Tasks are assigned priorities
  - Statically or dynamically
  - At any instant, the highest priority task is running
  - Whenever there is a request for a task that is of higher priority than the one currently being executed, the running task is interrupted, and the newly requested task is started
- Scheduling algorithm = method to assign priorities

Task Priority Assignment Options

- Static or fixed approach
  - Priorities are assigned to tasks once for all
- Dynamic approach
  - Priorities of tasks may change from request to request
- Mixed approach
  - Some tasks have fixed priorities, others don't

Assumptions

- Run-time for each task is constant for that task, and does not vary with time and input data
- Each task must finish before the next request for it
- Tasks are independent (requests for a certain task does not depend on the initiation or completion of requests for other tasks, no data dependency)
- No task can implicitly suspend itself, e.g., for I/O
- All tasks are fully preemptible
- All kernel overheads are zero (context switching is instantaneous)

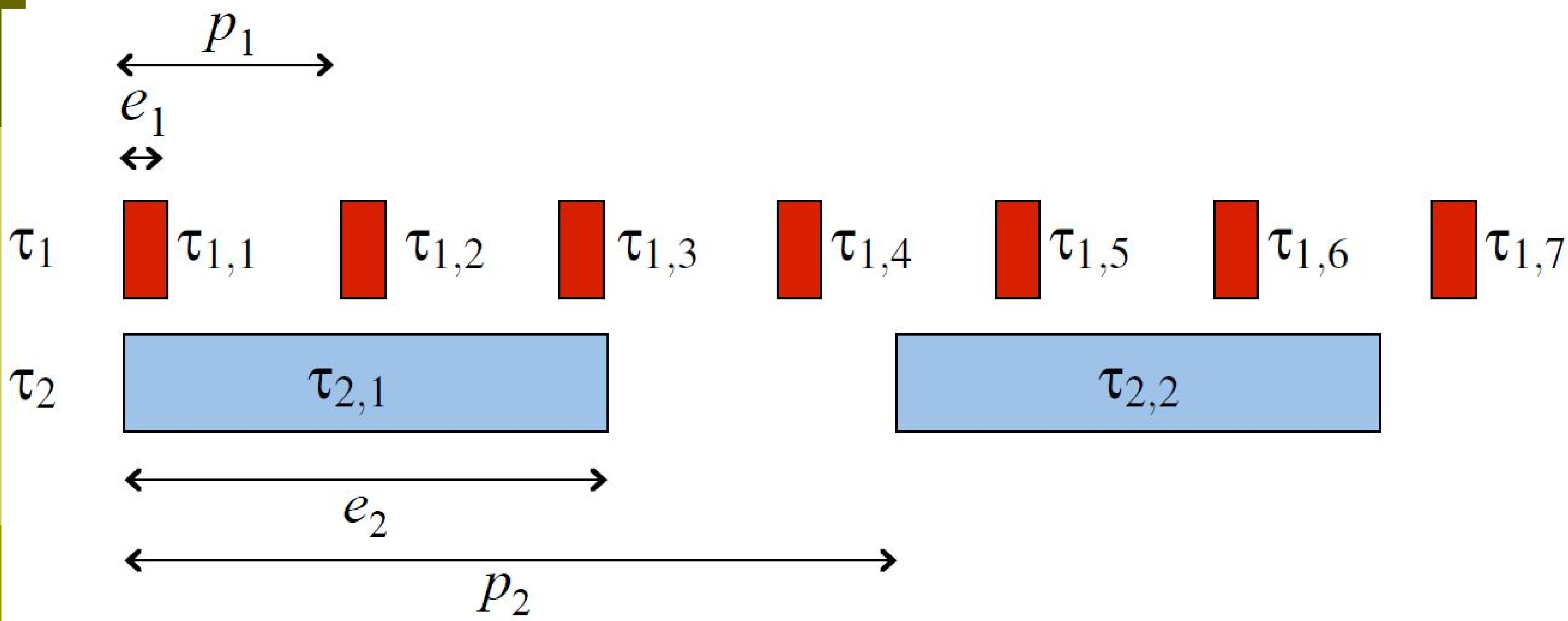
Rate Monotonic Definitions

- Consider n independent tasks: $t_1, t_2, \ldots, t_n$
- Assume request periods are $p_1, p_2, \ldots, p_n$
  - Request rate of t_i is $1/p_i$
  - $t_{i,j}$ indicates the j-th instance of the i-th task
- Run-times are $e_1, e_2, \ldots, e_n$

Rate Monotonic

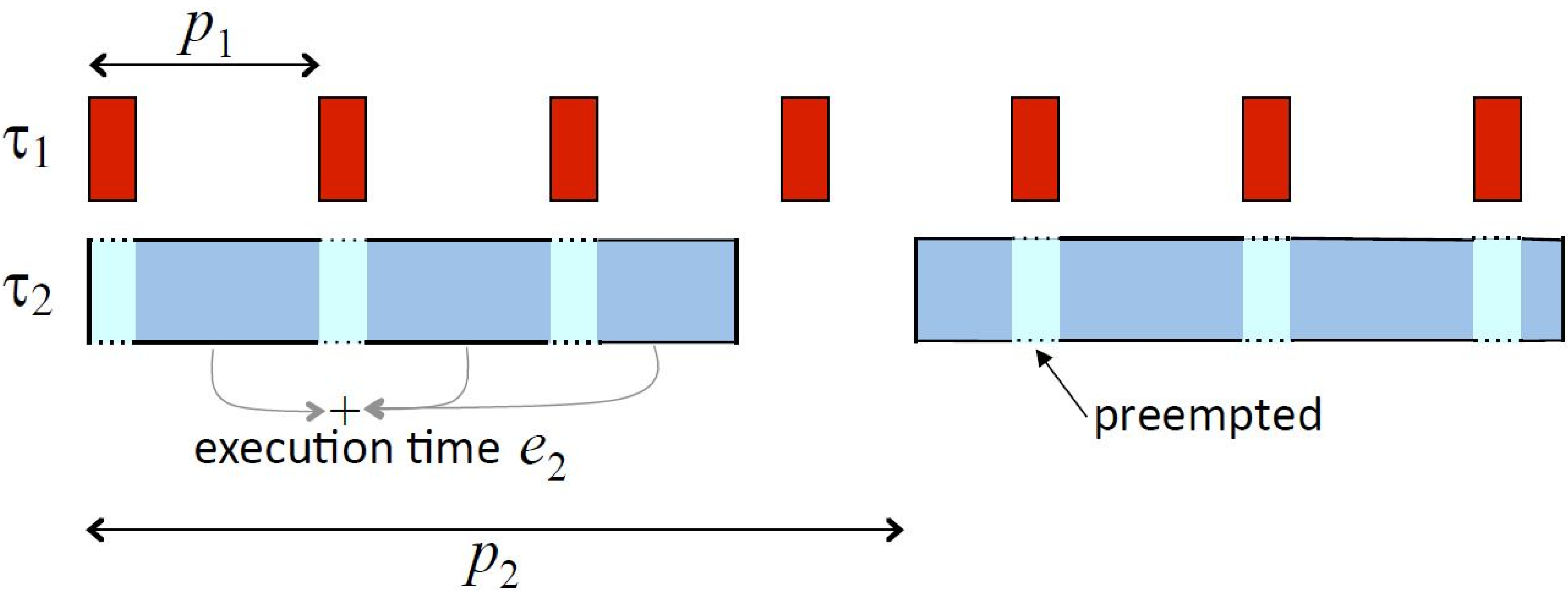
- Assign priorities according to request rates, independent of run times
  - Higher priorities for tasks with higher request rates
  - For tasks i and j, if pi < pj, Priority(i) > Priority(j)
- Called Rate Monotonic (RM) priority assignment
- It is optimal among static priority-based schemes
- Theorem 1: No other fixed priority assignment can schedule a task set if RM priority assignment can’t schedule it, i.e., if a feasible priority assignment exists, then RM priority assignment is feasible

Rate Monotonic Example 1



Not feasible with **non-preemptive** scheduler!

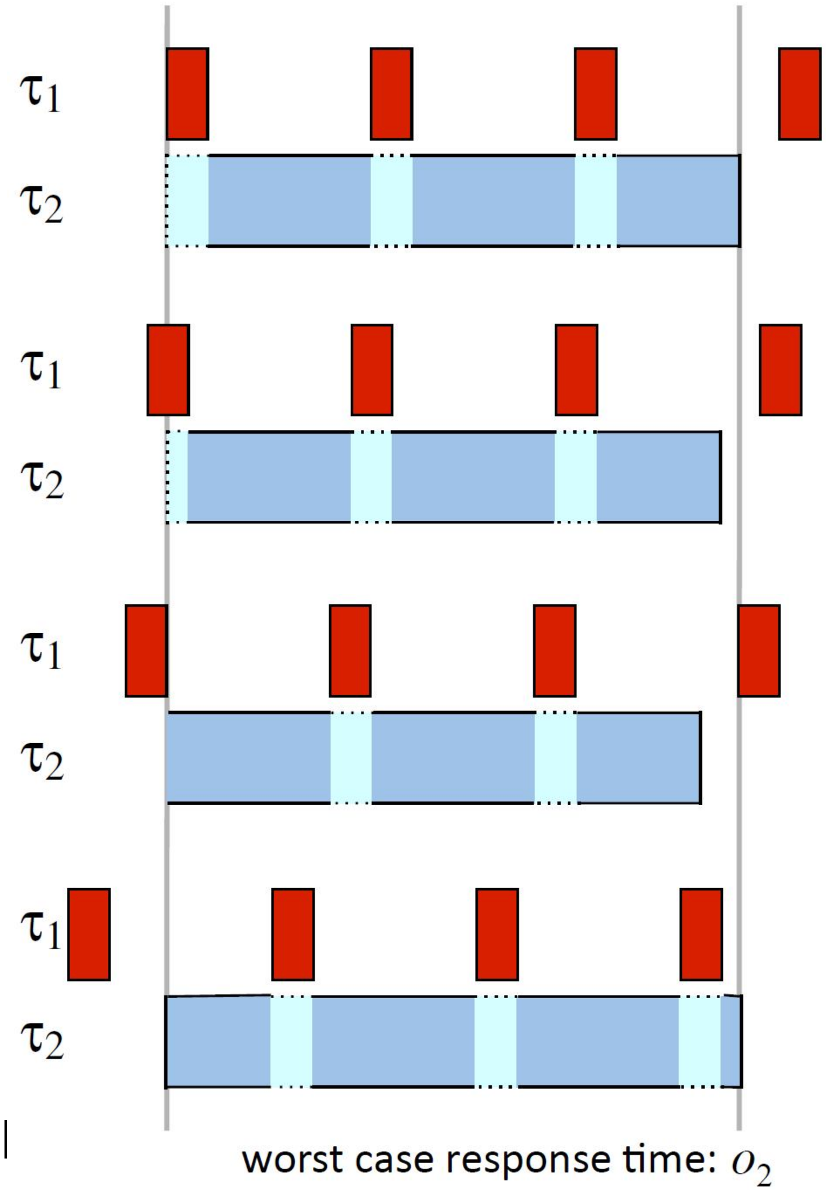
Rate Monotonic Example 2



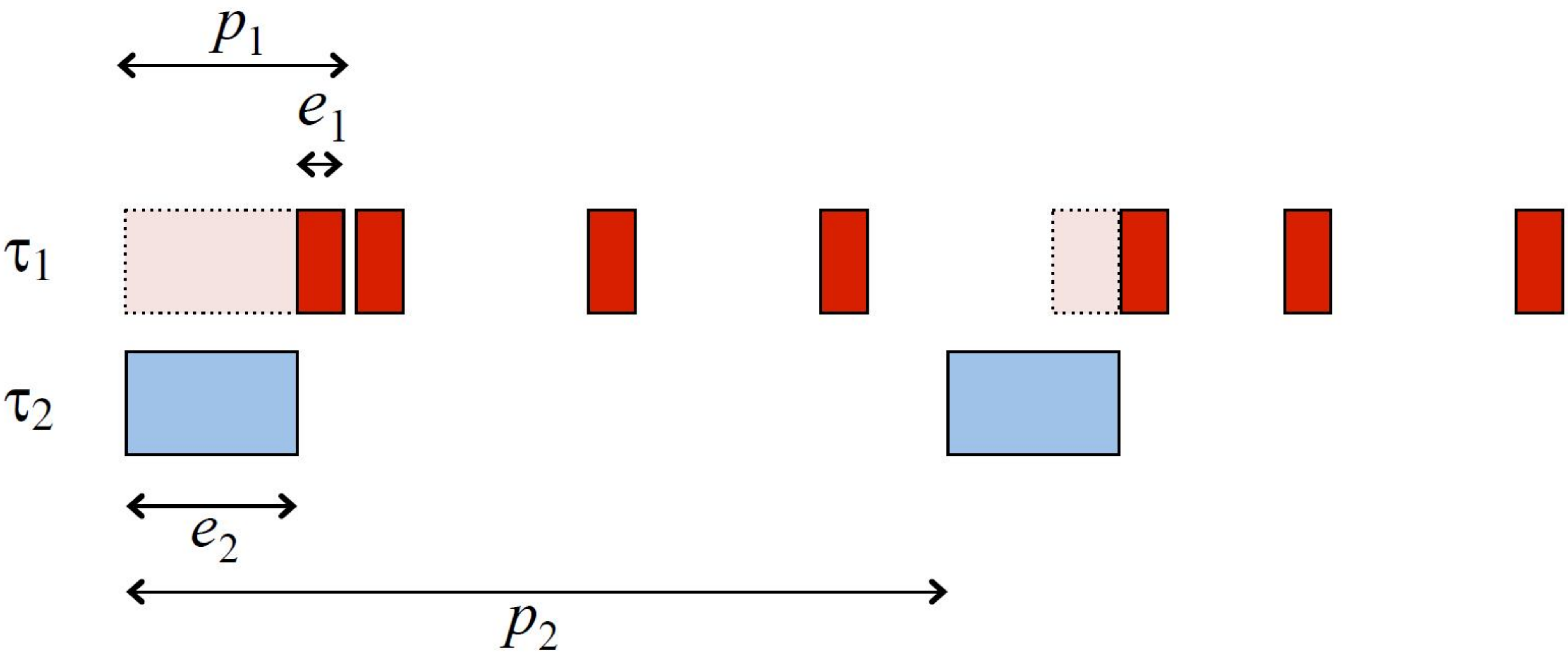
A preemptive schedule with higher priority for t1 is feasible

Rate Monotonic (worst case response time)

- Worst case response time is when the start times of tasks line up
- Reason: maximum number of context switching happens this way

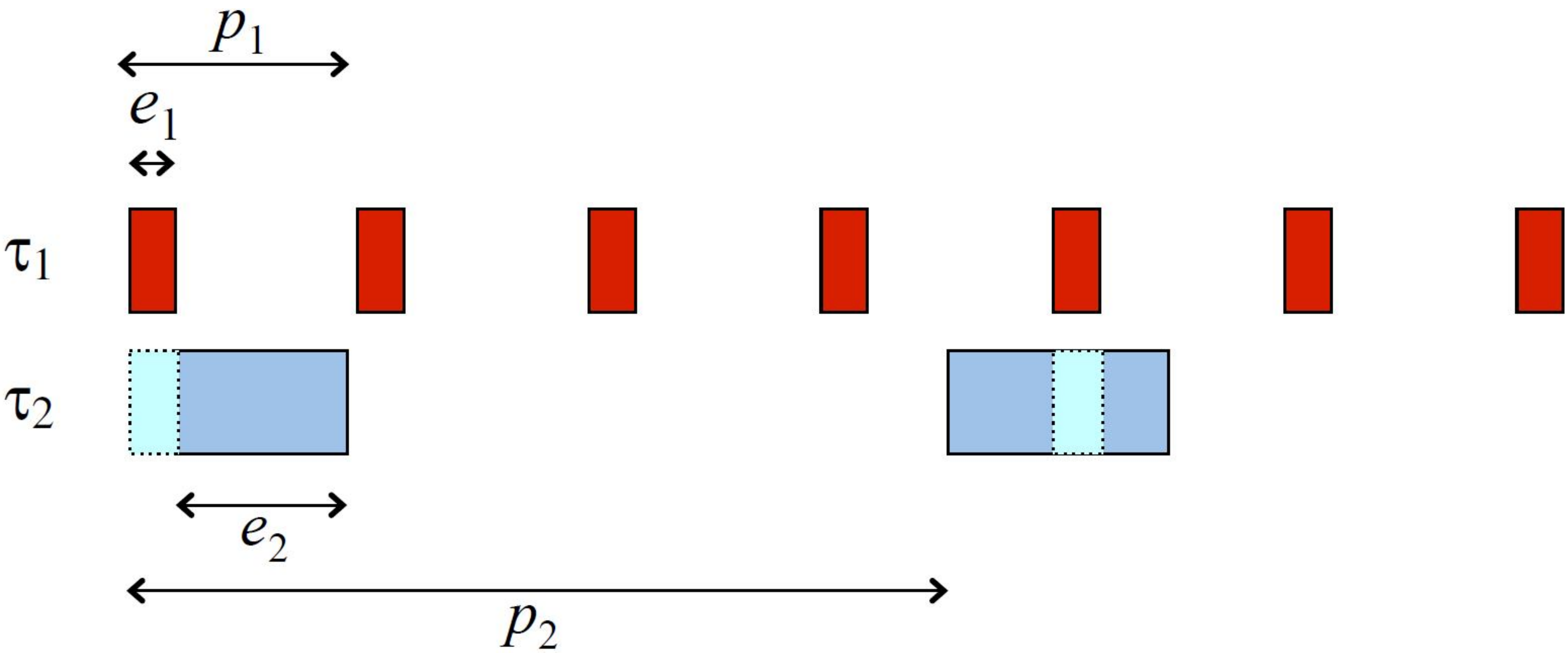


Non-Rate Monotonic



If priority given to t2 (non-RM), feasible iff e1 + e2 <= p1

Same Example with Rate Monotonic



If priority given to t1 (RM), e1 + e2 <= p1 is sufficient but not necessary

Processor Utilization Factor

- Portion of processor time spent in executing the task set
- Provides a measure of computational load on CPU due to a periodic task set
- For n tasks, t1, t2, ..., tn the utilization factor U is $U = e1/p1 + e2/p2 + ... + en/pn$
- A task set is definitely not schedulable if its processor utilization factor is greater than 1
- For a task set, U can be improved by increasing ei's or decreasing pi's as long as tasks continue to satisfy their deadlines

Utilization Factor Analysis

- There exists a minimum value of U below which task set is schedulable and above which it is not
  - This depends on scheduling algorithm and the task set
- Corresponding to a priority assignment, a set of tasks fully utilizes a processor if:
  - the priority assignment is feasible for the set
  - and, if an increase in the run time of any task in the set will make the priority assignment infeasible

Utilization Factor Analysis (Cont'd)

- The U at which this happens is called the upper bound for a task set
- The least upper bound of U is the minimum of the U's over all task sets that fully utilize the processor
  - For all task sets whose U is below this bound, there exists a fixed priority assignment which is feasible
  - U above this bound can be achieved only if the task periods pi's are suitably related

Utilization Factor Analysis (Cont'd)

- The least upper bound of U is an important characteristic of a scheduling algorithm as it allows easy verification of schedulability of a task set
  - Below this bound, a task set is definitely schedulable
  - Above this bound it may be schedulable

Upper Bound Theorem

- Theorem 2: For a set of n tasks with fixed priority assignment, the least upper bound to processor utilization factor is $U=n(2^{1/n}-1)$
- Or, equivalently, a set of n periodic tasks scheduled by RM algorithm will always meet their deadlines for all task start times if
  - $e1/p1 + e2/p2 + \ldots + en/pn <= n(2^{1/n}-1)$
- For large n, U = ln2 = 0.69 which is too low
- But, note that this is just the least upper bound
  - Task set with larger U may still be schedulable
  - If $U <= n(2^{1/n}-1)$ then it is RM schedulable
  - Otherwise, use other methods!

Example 1

- Task 1 : $e1 = 20$; $p1 = 100$
- Task 2 : $e2 = 30$; $p2 = 145$
- Is this task set schedulable?
- $U = 20/100 + 30/145 = 0.41 <= 2(2^{1/2}-1) = 0.828$
- Yes!

Example 2

- Task 1 : $e1 = 20$; $p1 = 100$
- Task 2 : $e2 = 30$; $p2 = 145$
- Task 3 : $e3 = 68$; $p3 = 150$
- Is this task set schedulable?
- $U = 20/100 + 30/145 + 68/150 = 0.86 > 3(2^{1/3}-1) = 0.779$
- Inconclusive! Need to try other ways.

Example 2 (Cont'd)

- Consider the critical instant of t3, the lowest priority task
  - t1 and t2 must execute at least once before t3 can begin executing
  - Therefore, completion time of t3 is e1+e2+e3 = 20+68+30 = 118
  - However, t1 is initiated one additional time in (0,118)
  - Taking this into consideration, completion time of t3 = 2*e1+e2+e3 = 2*20+30+68 = 138
  - 138<p3=150 so the task set is schedulable

Dynamic Priority Scheduling

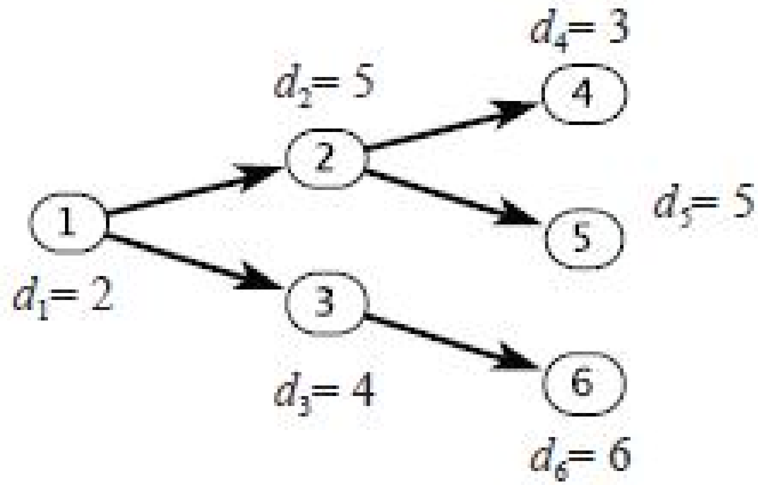
- Definition: priorities are assigned to individual instances of a task
- One of the most common algorithms of this class is EDF, or Earliest Deadline First
  - Task priorities are inversely proportional to absolute deadlines of active jobs
  - Optimal among all preemptive dynamic scheduling algorithms without precedences
  - Can be used for periodic or aperiodic
- If there exists a feasible schedule, then schedule given by EDF is also feasible

Other Dynamic Schedulings

- Latest Deadline First (LDF):
  - Schedules the last task first
  - Minimizes the maximum lateness
  - Optimal for cases with data dependency
  - But doesn't support task arrivals
- EDF with precedence (EDF*)
  - Minimizes the maximum lateness and supports task arrivals
  - Considers deadline of a task and its successors
  - Modified deadline: $d_i' = \min(d_i, \min_{j \in D(i)} (d_j' - e_j))$

Example: Comparison of EDF, LDF, EDF*

Assumption: All execution times are 1



Note: task 4 misses its deadline in EDF!

EDF	1	3	2	4	5	6
LDF	1	2	4	3	5	6
EDF*	1	2	4	3	5	6

0 2 4 6

$$d'_i = \min(d_i, \min_{j \in D(i)} (d'_j - e_j))$$

$$d'_1 = 1, \quad d'_2 = 2, \quad d'_3 = 4, \quad d'_4 = 3, \quad d'_5 = 5, \quad d'_6 = 6$$