Chapter 9
Real-Time Scheduling

Objectives

What are you about to learn?

Knowledge Objectives

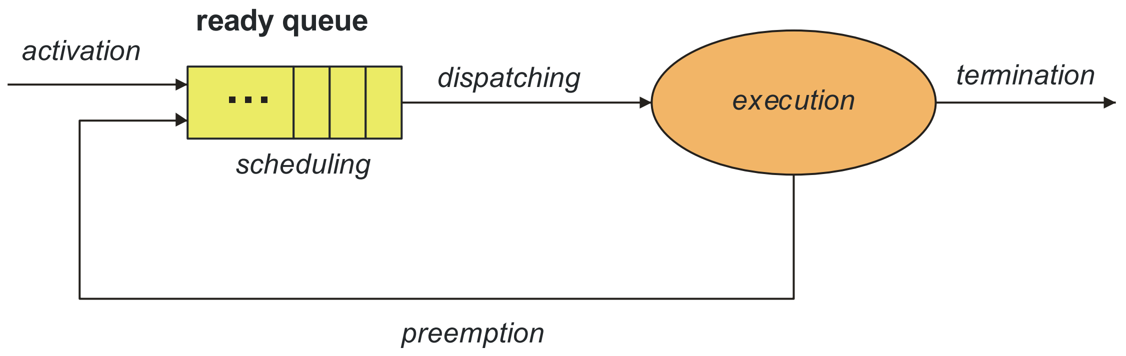
- Understand the scheduling problem for a task set with *independent*, *periodic* tasks.
- Know the different task types and scheduling parameters.
- Understand the different scheduling algorithms like RM, DM, EDF, LL, MLQS, cyclic scheduling.
- Understand how a bitmap scheduler works.

Skill Objectives

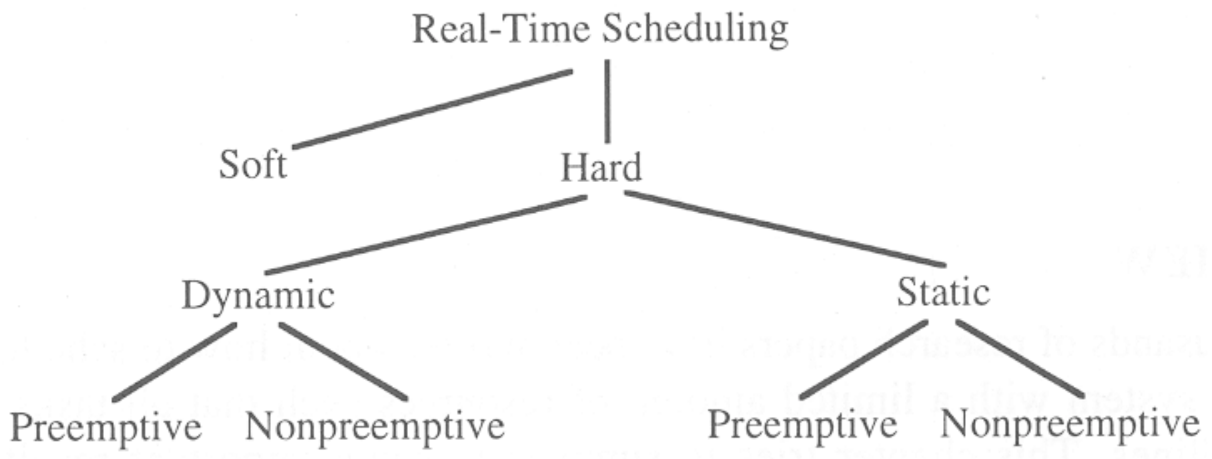
- Ability to determine if a task set is schedulable using a given scheduling algorithm.
- Ability to design a plan for a given task set and a given scheduling algorithm.
- Ability to draw a timing diagram for a task set scheduled by some scheduling algorithm.

9.1 The Scheduling Problem

The scheduling problem is concerned with allocating computing and data resources to concurrent real-time tasks, such that they meet their specified deadlines.



Classification of Scheduling Algorithms



9.1 The Scheduling Problem

Dynamic versus static scheduling:

- Dynamic (on-line) scheduler: scheduling decision are made at run time, selecting one out of a set of ready tasks.
  Dynamic schedulers are flexible and adapt to evolving tasks scenario. They consider just the current task requests. Effort to find a schedule can be large.
- Static (off-line, pre-run-time) scheduler: scheduling decisions are made at compile time. A dispatching table is generated. This requires complete prior knowledge about the task-set characteristics (maximum execution times, precedence constraints, etc.). The run-time overhead of the dispatcher is small.

Preemptive versus non-preemptive scheduling:

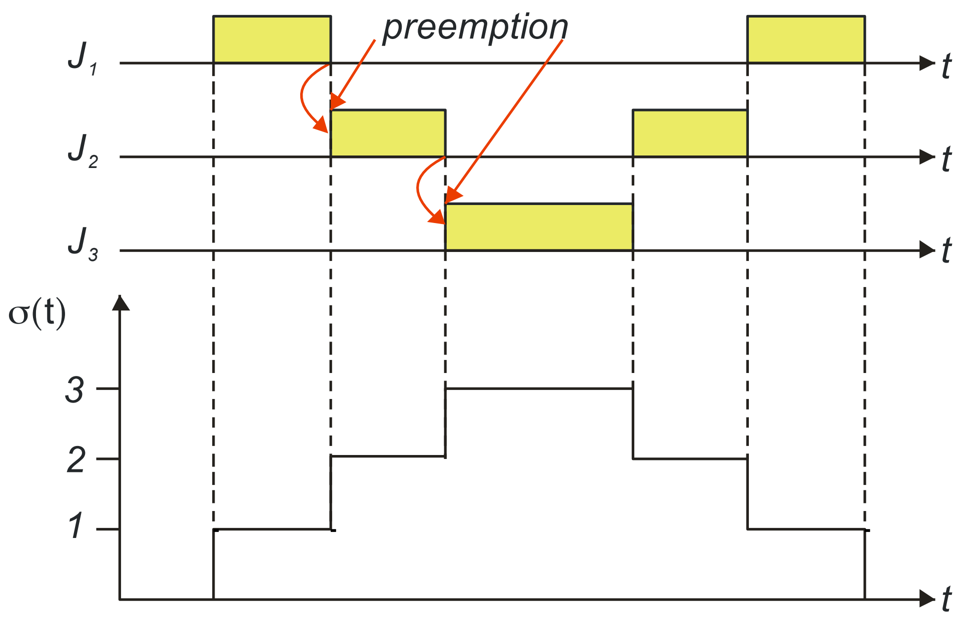
In preemptive scheduling, the currently executing task may be preempted, i.e., interrupted, if a more urgent task requires service.

In non-preemptive scheduling the currently executing task has to release allocated resources itself, it will not be interrupted by other tasks or the operating system. Typically, tasks run to completion. In some operating systems, if a task requests operating system services, the operating system can preempt the task, i.e., calls to the operating system are preemption points.

Non-preemptive scheduling is reasonable if there are many short tasks (compared to the required response time) to be executed.

Most commercial real-time operating systems employ preemptive schedulers.

9.1 The Scheduling Problem



Centralized versus distributed Scheduling:

In a dynamic distributed real-time system, it is possible to have a central scheduling instance, or to employ a distributed scheduling algorithm.

A central scheduler constitutes a single point of failure in a distributed system. Furthermore, it requires up-to-date information of the load situation of all nodes, which can consume a substantial amount of the communication channel bandwidth.

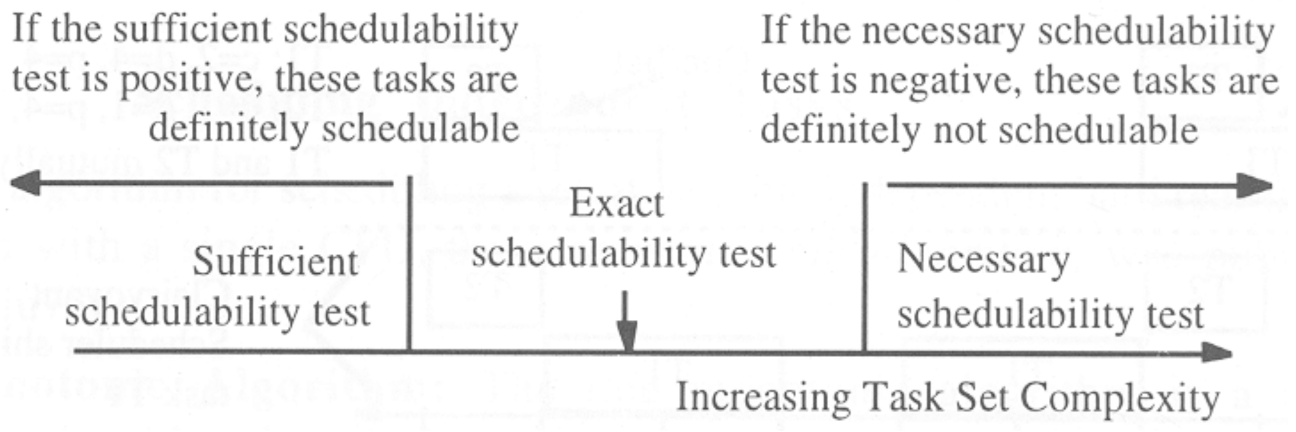
Schedulability Test

A schedulability test is a test that determines whether a set of ready tasks can be scheduled such that each task meets its deadline.

9.1 The Scheduling Problem

We distinguish between

- exact schedulability tests
- necessary schedulability tests
- sufficient schedulability tests


If the sufficient schedulability test is positive, these tasks are definitely schedulable
If the necessary schedulability test is negative, these tasks are definitely not schedulable
Exact schedulability test
Sufficient schedulability test
Necessary schedulability test
Increasing Task Set Complexity

A scheduler is optimal if it will always find a schedule provided an exact schedulability test indicates the existence of such a schedule.

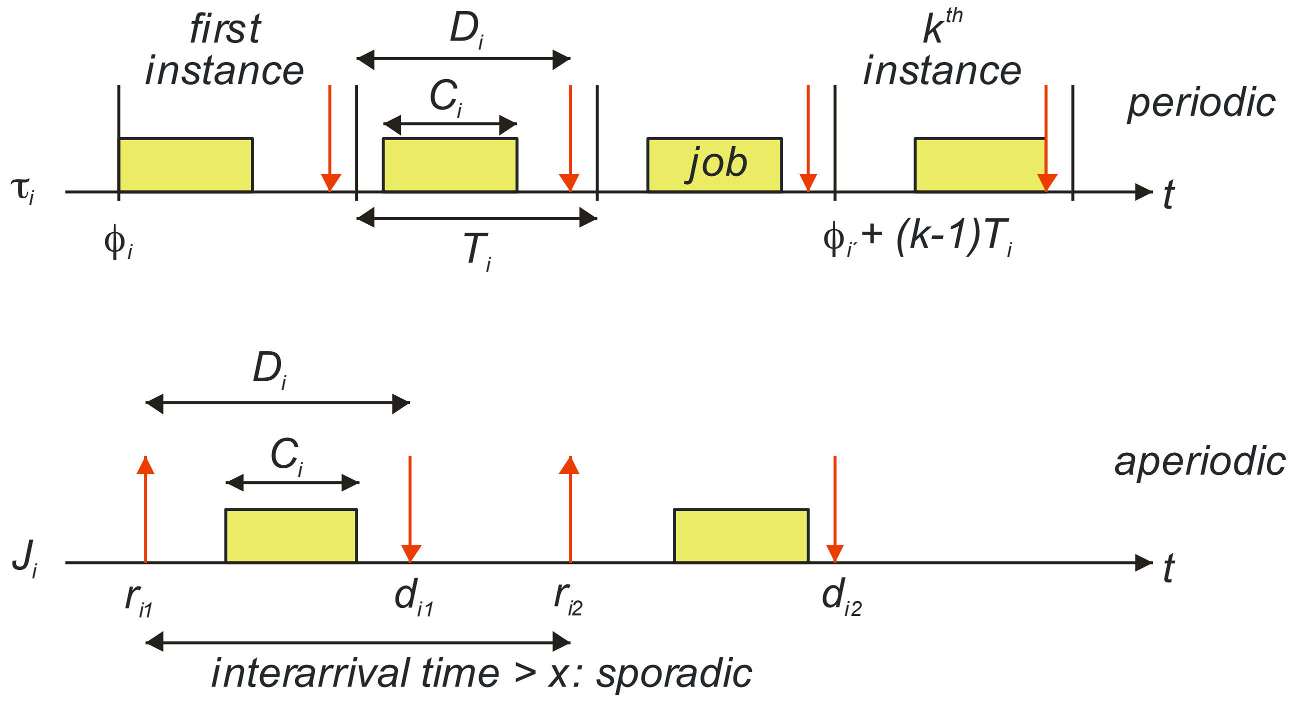
In case of task dependency, such as a shared resource, the complexity of an exact schedulability test is an NP-complete problem, and is thus computationally intractable.

9.2 The Adversary Argument

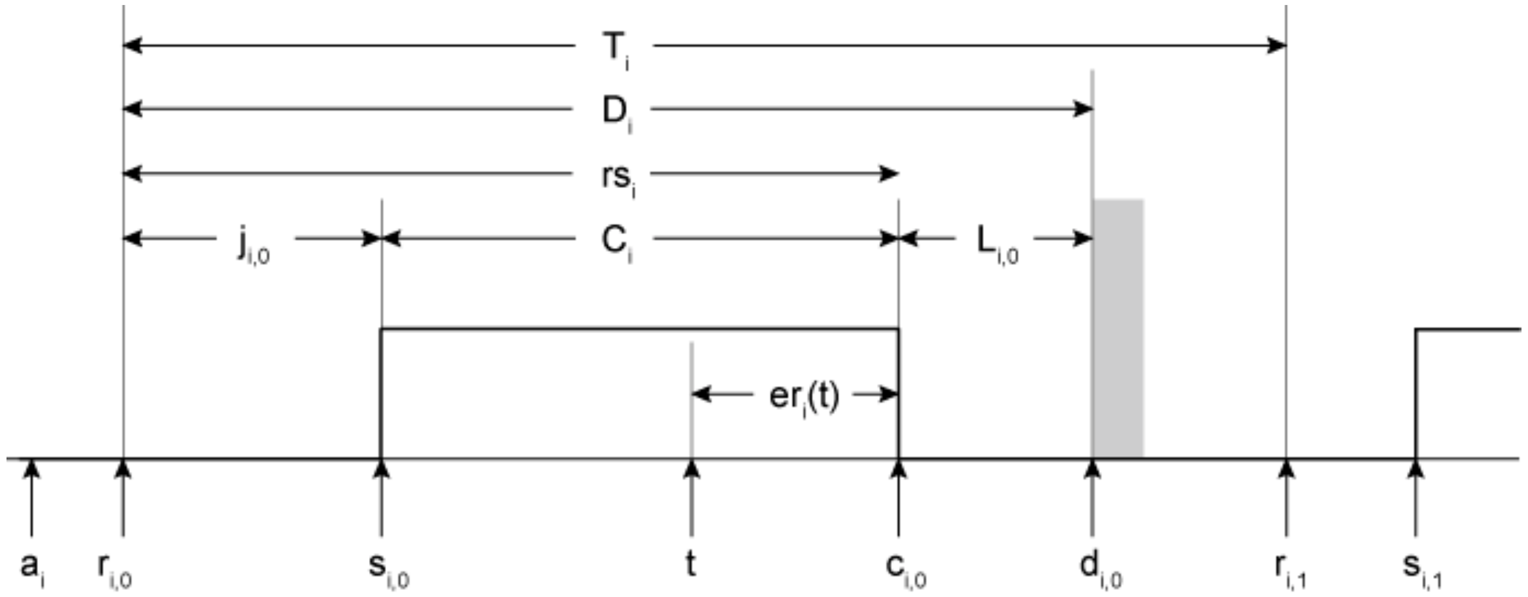
Task Types and Temporal Parameters

We distinguish between

- periodic tasks, request times are known a priori if the first request time is known
- sporadic tasks, request times are not known a priori, and there is a minimum time interval between request times
- aperiodic tasks, like sporadic tasks, but without constraints on the request times



9.2 The Adversary Argument



a_i	Arrival time, task becomes known to the system. Sometimes set equivalent to the request time r_i
r_i	Request time, release time. At this time the job is ready to execute. For periodic tasks this is also called the phase of the task. In-phase tasks have the same request times. For periodic tasks, when the first request time is known, all future request times are known as well.
s_i	Start time, the job is dispatched to execute on the computing resource.

9.2 The Adversary Argument

c_i	Completion time, the task has completed (for this period in case of a periodic task), or a job has completed (for sporadic tasks).
d_i	Deadline, the job has to be completed before this point in time. For periodic tasks, knowledge of the first deadline implies knowledge of all future deadlines. For periodic tasks, deadlines are implicitly set equivalent to the request time of the next cycle ($d_i = r_{i+1}$), if not explicitly stated otherwise.
T_i	Task period (for periodic tasks).
j_i	Reaction time, release jitter, $s_i - r_i$. This time can vary considerably depending on the current load situation of the system and the scheduling algorithm employed.
C_i	Execution time, computation time, $c_i - s_i$. The upper bound for this time we call the worst case execution time (WCET).
er_i	Remaining execution time, $c_i - t$, depends on the time of observation t. This is the computing time required to complete this job.
D_i	Relative deadline, $d_i - r_i$. This constitutes the upper bound of time between the point in time a job is released and when it has to be finished.
L_i	Laxity, $d_i - c_i$, the time left for job execution before the deadline is violated. At an arbitrary point in time of a cycle this is $l_i(t) = d_i - (t + er_i)$.
rs_i	Response time, $rs_i = e_i + j_i$, the time between the request and end of execution.

9.2 The Adversary Argument

This time can never be smaller than the minimum execution time.

There are a number of additional parameters that are helpful:

H	Hyperperiod, the smallest common multiple of all task periods $p_{i,}\ i = 1,2,3,\ldots$. The upper bound for the number of jobs inside a hyperperiod is given by $\sum_{i=1}^{n} H / T_i$
u_i	Processor utilization factor of a task, $u_i = C_i / T_i$ for periodic tasks. This value can never be larger than n for a computer with n processors, i.e., 100% for a single processor computer. It has to be always smaller than 1 in case a task cannot run on several processors at the same time.
U	Processor utilization factor of a task set, $U = \sum_{i=1}^{n} C_i / T_i$ for periodic tasks. This value can never be larger than n for a computer with n processors. Usually computing time for the operating system is not considered.
ch_i	Processor load factor of a task, $ch_i = C_i / D_i$. To meet real-time constraints, this value has to be smaller than n for a computer with n processors, or smaller than 1 in case a task cannot run on several processors at the same time.
CH	Processor load factor of a task set, $CH = \sum_{i=1}^{n} C_i / D_i$. To meet real-time constraints, this value has to be smaller than n for a computer with n processors.

9.2 The Adversary Argument

For a schedulability test of a periodic task set it suffices to examine schedules with a length of one scheduling period (hyperperiod).

Schedule Types

We distinguish between

- aperiodic task scheduling
- periodic task scheduling
- fixed priority servers, where a mix of periodic and aperiodic tasks are scheduled (hybrid task sets), and the priorities for the periodic tasks are determined at compile time
- dynamic priority servers, where a mix of periodic and aperiodic tasks are scheduled, and the priorities of the periodic tasks are determined at run-time.

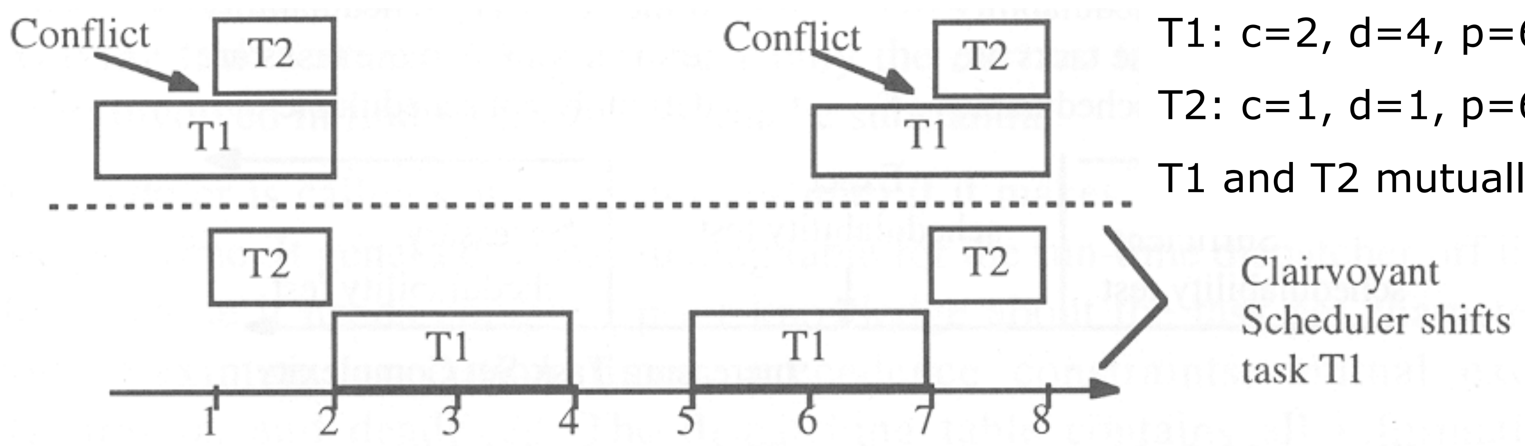
The Adversary Argument

We call a dynamic scheduler optimal if it can find a schedule where a clairvoyant scheduler, i.e., a scheduler with complete knowledge of the future can find a schedule.

The adversary argument states that, in general, it is not possible to construct an optimal totally on-line scheduler if there are mutual exclusion constraints between a periodic and a sporadic task.

Example where T2 always is requested when T1 is activated:

9.2 The Adversary Argument



T1: c=2, d=4, p=6, periodic

T2: c=1, d=1, p=6, sporadic

T1 and T2 mutually exclusive

Here: $U = 2/6 + 1/6 = 1/2 \leq 1$, so the necessary schedulability test is satisfied. Nonetheless it is not possible to construct a schedule unless the request times of T2 are taken into account.

The clairvoyant scheduler moves T1 such that the stringent deadline of T2 can be met.

9.3 Dynamic Scheduling of Periodic Task Sets

A dynamic scheduling algorithm determines on-line after the occurrence of a significant event which task out of the ready task set must be serviced next.

Scheduling Independent Tasks: Periodic Task Scheduling

There are a number of algorithms for dynamically scheduling a set of periodic independent hard real-time tasks. We can classify them according to the following scheme:

- Queue based algorithm
- Priority based algorithms
  - Algorithms with static priorities
  - Algorithms with priorities changing at run-time

The most widely employed algorithm for scheduling on a single CPU is the rate monotonic algorithm, which is based on static priority assignment.

Rate Monotonic (RM) Algorithm

The rate monotonic algorithm is a dynamic preemptive algorithm based on static task priorities.

It assigns static priorities based on the task periods. The task with the shortest period gets the highest static priority; the task with the longest period gets the lowest static priority.

At run time, the dispatcher selects the task request with the highest static priority.

9.3 Dynamic Scheduling of Periodic Task Sets

The following assumptions are made about the task set:

- The requests for all tasks of the task set $\{\tau_i\}$ for which hard deadlines exist are periodic.
- All tasks are independent of each other. There exist no precedence constraints or mutual exclusion constraints between any pair of tasks.
- The deadline interval of every task τ_i is equal to its period T_i.
- The required maximum computation time C_i of each task is known a priori and is constant.
- The time required for context switching can be ignored.
- The sum of the utilization factors u_i of the n tasks is given by

  $U = \sum_{i=1}^{n} C_i / T_i \le n(2^{1/n} - 1)$

  The term approaches $ln\ 2$, i.e., about 0.7, as n goes to infinity.

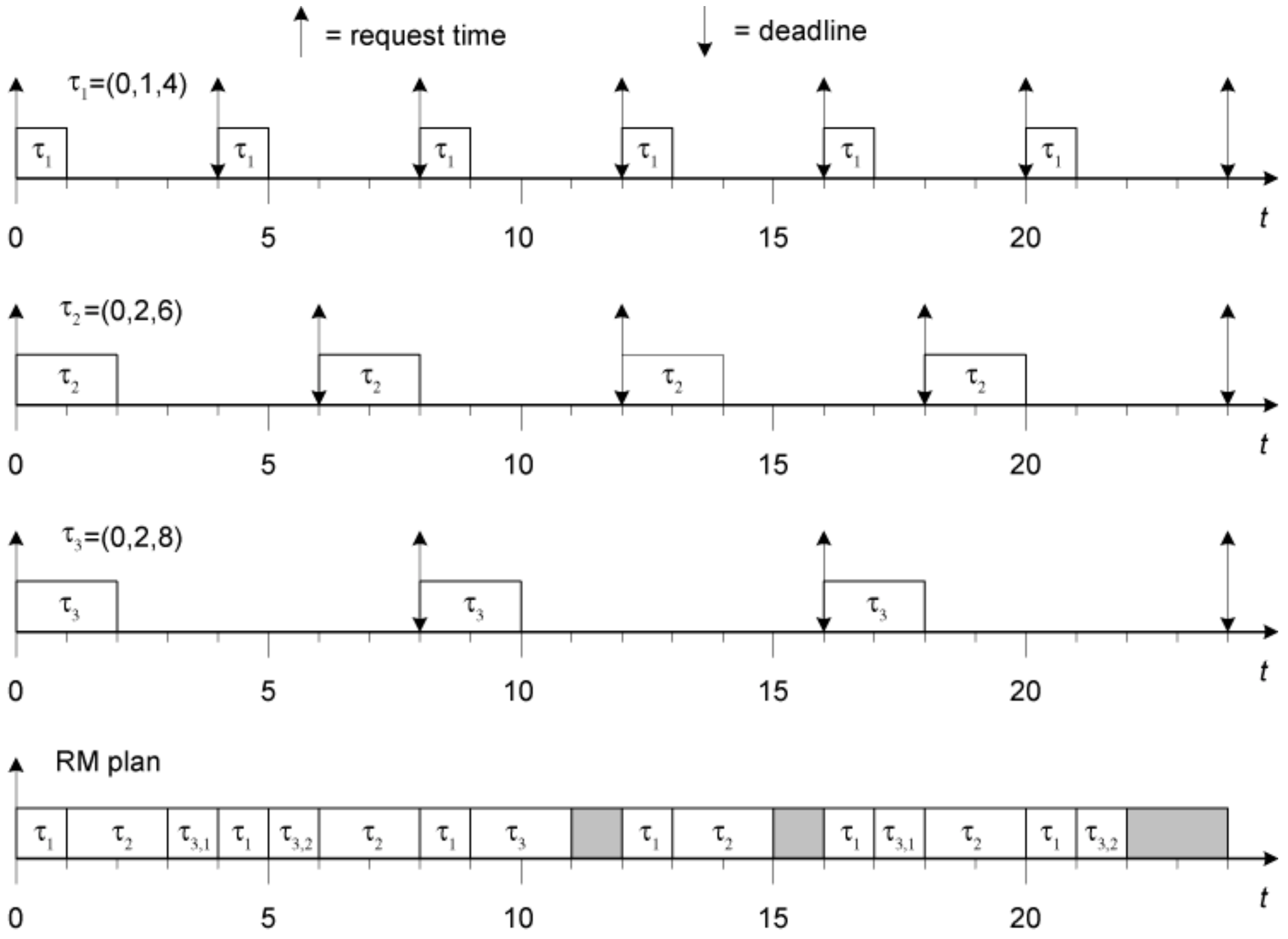
If all assumptions are satisfied the algorithm guarantees that all tasks will meet their deadline. The algorithm is optimal for single processor systems.

In case the task periods are multiples of the period of the highest priority task the utilization factor can approach the theoretical maximum of unity in a single processor system:

$$U = \sum_{i=1}^{n} C_i / T_i \le 1$$

9.3 Dynamic Scheduling of Periodic Task Sets

Example for RM plan and three tasks:

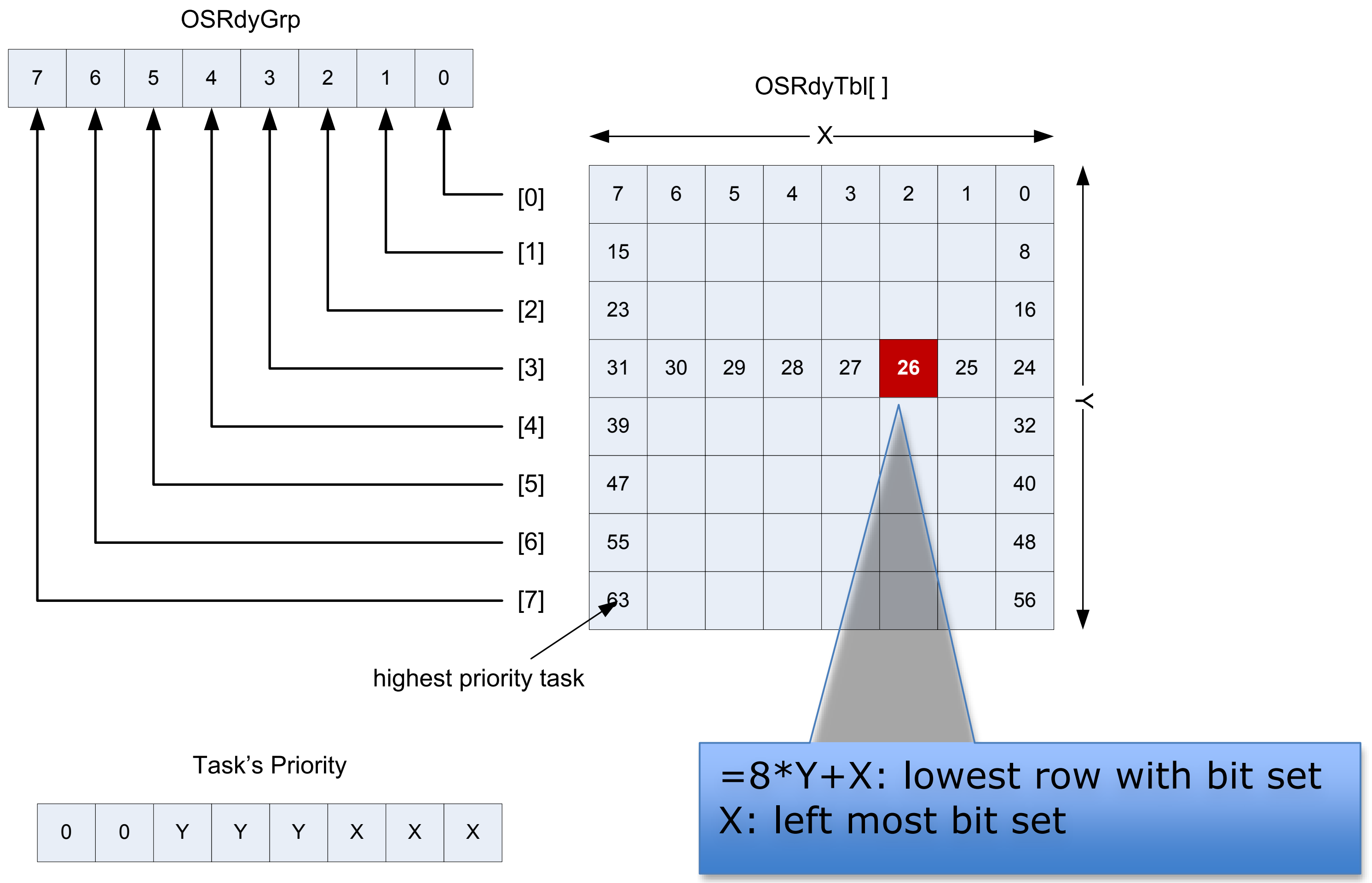


= request time
= deadline
$\tau_1=(0,1,4)$
τ_1
$\tau_2=(0,2,6)$
τ_2
$\tau_3=(0,2,8)$
τ_3
RM plan
τ_1 τ_2 $\tau_{3,1}$ τ_1 $\tau_{3,2}$ τ_2 τ_1 τ_3 τ_1 τ_2 τ_1 $\tau_{3,1}$ τ_2 τ_1 $\tau_{3,2}$
0
5
10
15
20
t

Example Implementation: OSEK Bitmap Scheduler

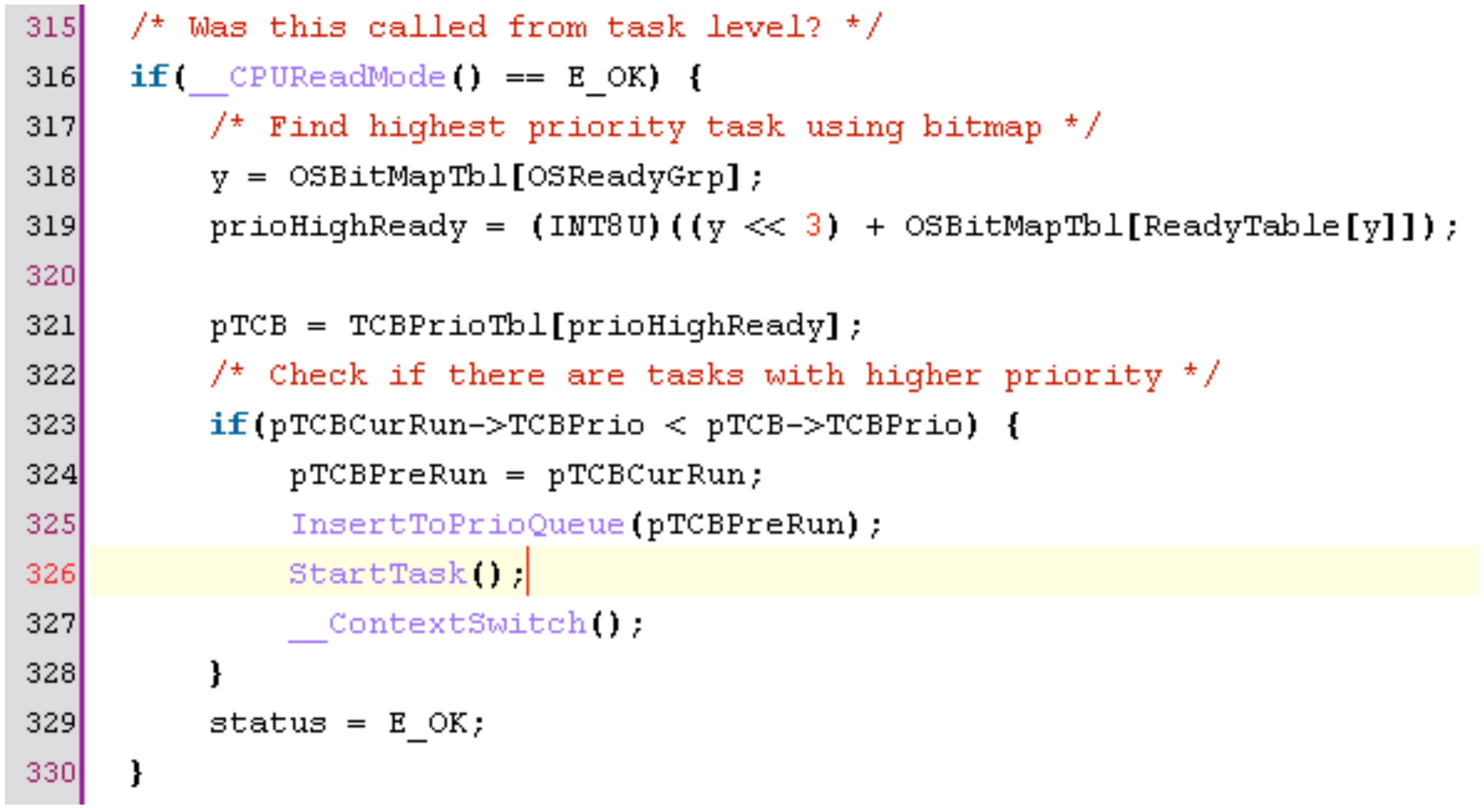
Scheduler is called often (event flag operations, semaphore operations, interrupts, mailbox and message queue operations); thus it must execute quickly.

One effective approach is the bitmap scheduler.



9.3 Dynamic Scheduling of Periodic Task Sets

The scheduler:



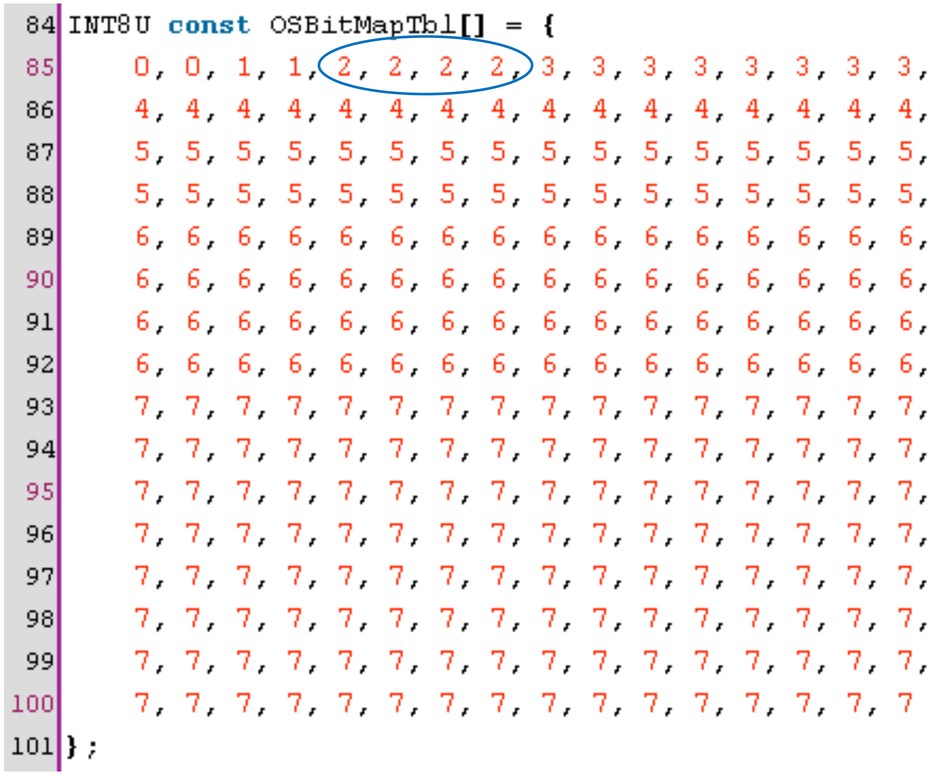
```
/* Was this called from task level? */
if(__CPUReadMode() == E_OK) {
    /* Find highest priority task using bitmap */
    y = OSBitMapTbl[OSReadyGrp];
    prioHighReady = (INT8U)((y << 3) + OSBitMapTbl[ReadyTable[y]]);

    pTCB = TCBPrioTbl[prioHighReady];
    /* Check if there are tasks with higher priority */
    if(pTCBCurRun->TCBPrio < pTCB->TCBPrio) {
        pTCBPreRun = pTCBCurRun;
        InsertToPrioQueue(pTCBPreRun);
        StartTask();
        __ContextSwitch();
    }
    status = E_OK;
}
```

OSReadyGrp holds the bit pattern of all rows in ReadyTable that have a bit set. Example for priority 26: the highest bit set in OSReadyGrp is bit #3. Other bits that could be set as well would be bit #2, #1, and #0.

9.3 Dynamic Scheduling of Periodic Task Sets

The scheduler makes use of a static table, the OSBitMapTbl[ ]. This serves to find the bit which marks the highest priority value in a single byte:



```
INT8U const OSBitMapTbl[] = {
    0, 0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3,
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
    5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
    5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
    6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
    6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
    6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
    6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7
};
```

Example when highest priority task is task with priority 26 (Bit #2 in row 3):

- X = 4 (100), 5 (101), 6 (110), or 7 (111) (Bit #2 must be set)
- Y = 3 (simply integer division of 26 / 8 == priority >> 3)

Deadline Monotonic (DM) Algorithm

The deadline monotonic algorithm is based on the same assumptions as the rate monotonic algorithm, with one exception.

The deadline monotonic algorithm drops the assumption that the deadline interval of every task τ_i is equal to its period T_i; the deadline can be shorter than the period.

The deadline monotonic algorithm assigns static priorities based on the task deadlines. The task with the shortest deadline gets the highest static priority; the task with the longest deadline gets the lowest static priority.

A sufficient condition for schedulability of a task set is then given by:

$$CH = \sum_{i=1}^{n} C_i / D_i \le n(2^{1/n} - 1)$$

9.3 Dynamic Scheduling of Periodic Task Sets

Earliest Deadline First (EDF) Algorithm

The earliest deadline first algorithm is an optimal dynamic preemptive algorithm in single processor systems, which is based on dynamic priorities.

After any significant event, the task with the earliest deadline is assigned the highest dynamic priority. After each significant event, the dispatcher selects the task request with the highest priority.

It makes the following assumptions about the task set (same as for the RM algorithm):

- The requests for all tasks of the task set $\{\tau_i\}$ for which hard deadlines exist are periodic.
- All tasks are independent of each other. There exist no precedence constraints or mutual exclusion constraints between any pair of tasks.
- The deadline interval of every task τ_i is equal to its period T_i.
- The required maximum computation time C_i of each task is known a priori and is constant.
- The time required for context switching can be ignored.

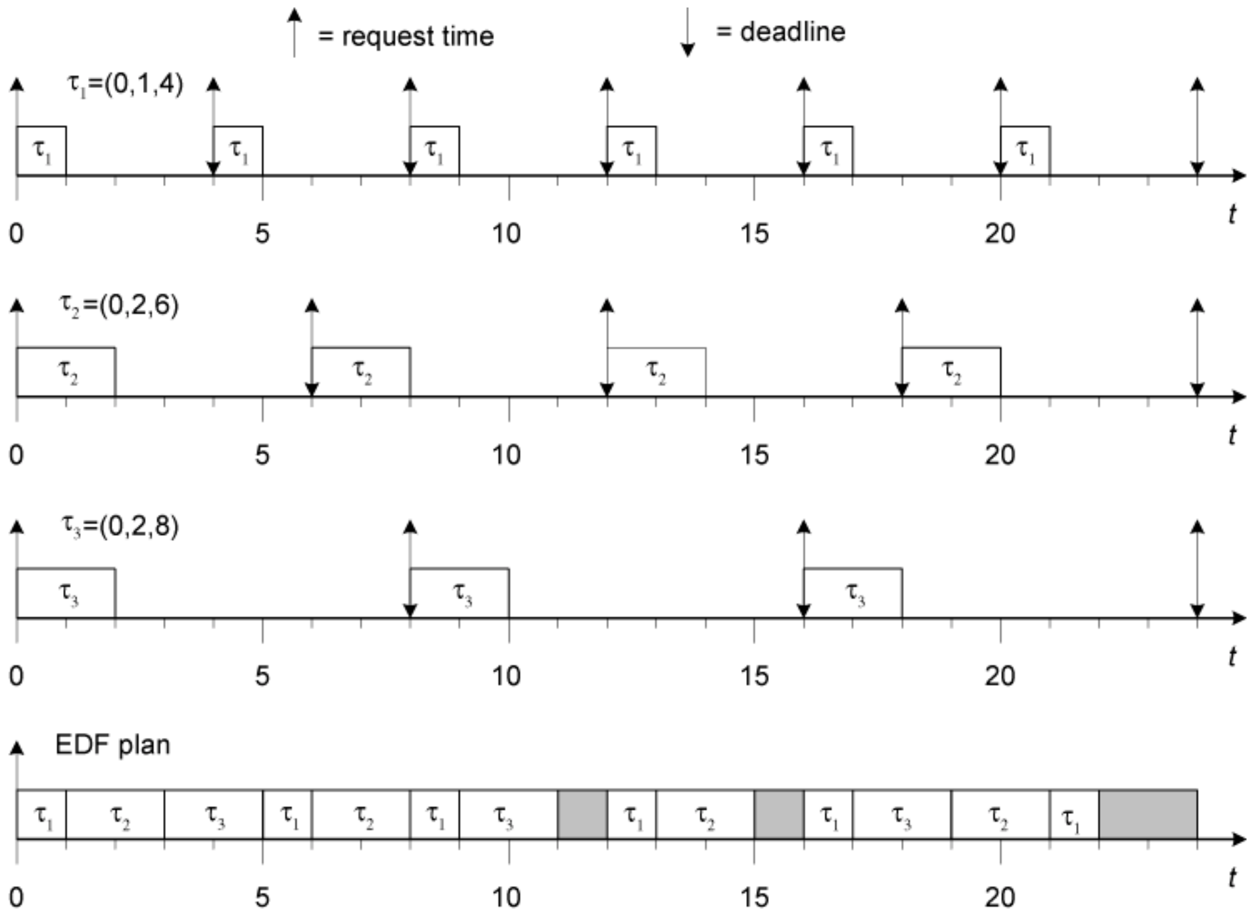
The processor utilization factor U can go up to 1, even when the task periods are not multiples of the smallest period: $U = \sum_{i=1}^{n} C_i / T_i \leq 1$

In case the deadline is smaller than the period, the following condition has to hold:

$$CH = \sum_{i=1}^{n} C_i / D_i \leq 1$$

9.3 Dynamic Scheduling of Periodic Task Sets

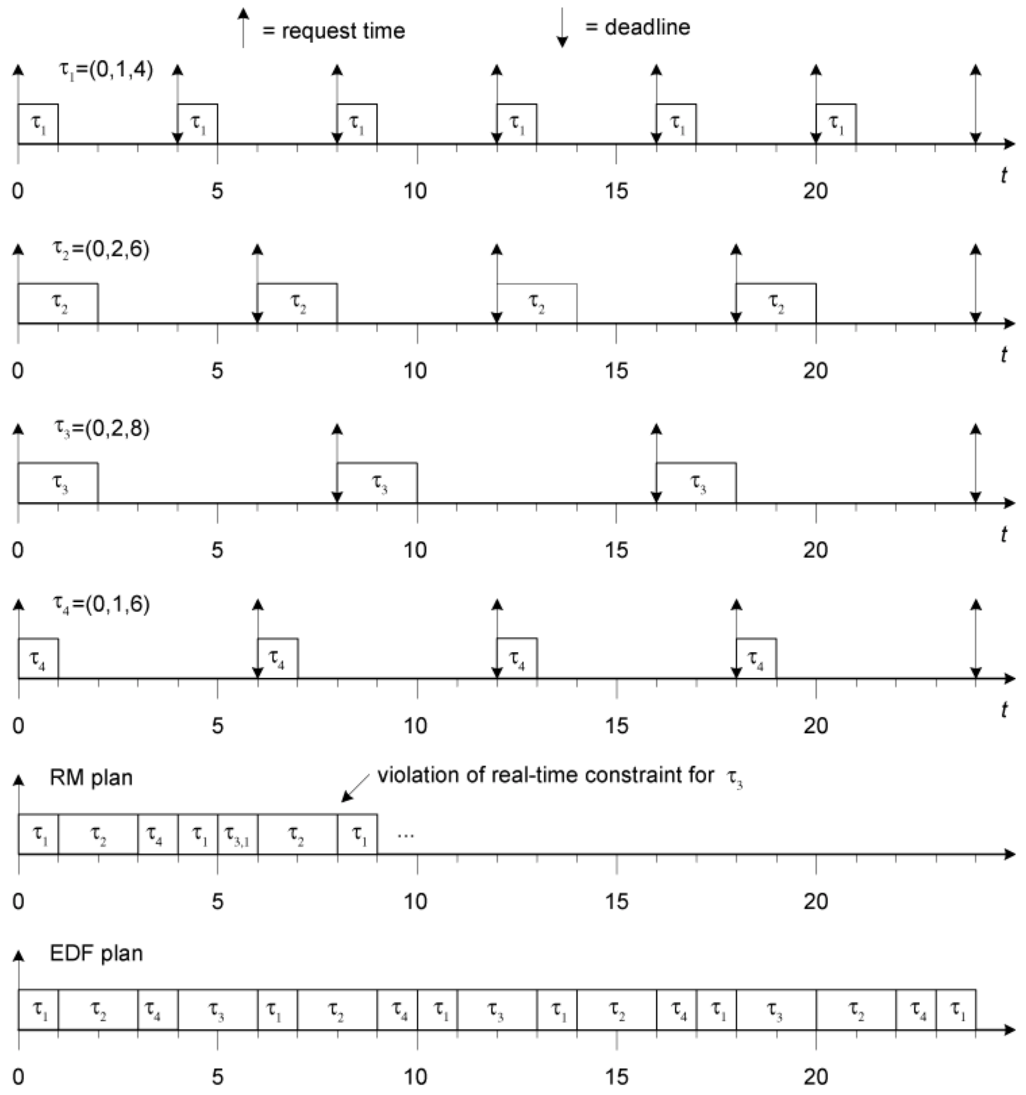
Example for an EDF plan for a task set with three tasks:



= request time
= deadline
τ_1=(0,1,4)
τ_2=(0,2,6)
τ_3=(0,2,8)
EDF plan
τ_1 τ_2 τ_3 τ_1 τ_2 τ_1 τ_3 τ_1 τ_2 τ_1 τ_3 τ_2 τ_1
0
5
10
15
20
t

9.3 Dynamic Scheduling of Periodic Task Sets

Example for comparison of performance of RM and EDF algorithm:



Least-Laxity (LL) Algorithm

The least laxity algorithm is another optimal scheduling algorithm for single processor systems. It makes the same assumptions as the EDF algorithm.

At any scheduling decision point the task with the shortest laxity l, i.e., the difference between the deadline interval D and the execution time C,

$$l = D - C$$

is assigned the highest priority.

Multilevel Priority (Queue) Scheduling (MLQS)

In a strictly priority based scheduling the question arises what to do with tasks having the same priority. A common approach is to employ a queue to contain all ready tasks for the same priority. Each higher-priority queue has to be empty before a lower priority queue is being served.

The queues can be managed in the following ways:

- FIFO, tasks are served in their order of arrival in that queue
- shortest job first, this minimized overall response time
- Round robin, each task in the queue gets a time slice called quantum, which it can consume; thereafter the next task in that queue is being served. The quantum can be different for each queue.

Scheduling Dependent Tasks

Scheduling a set of dependent tasks is of more practical relevance than scheduling independent tasks. Typically, concurrently executing tasks must exchange information and access common data resources.

In general finding a schedule for a set of tasks that use semaphores to enforce mutual exclusion is computationally intractable (NP complete problem). Doing this on-line would consume too much computational resources. The more resources are spent on scheduling, the fewer resources remain to perform the actual work.

Three possible solutions:

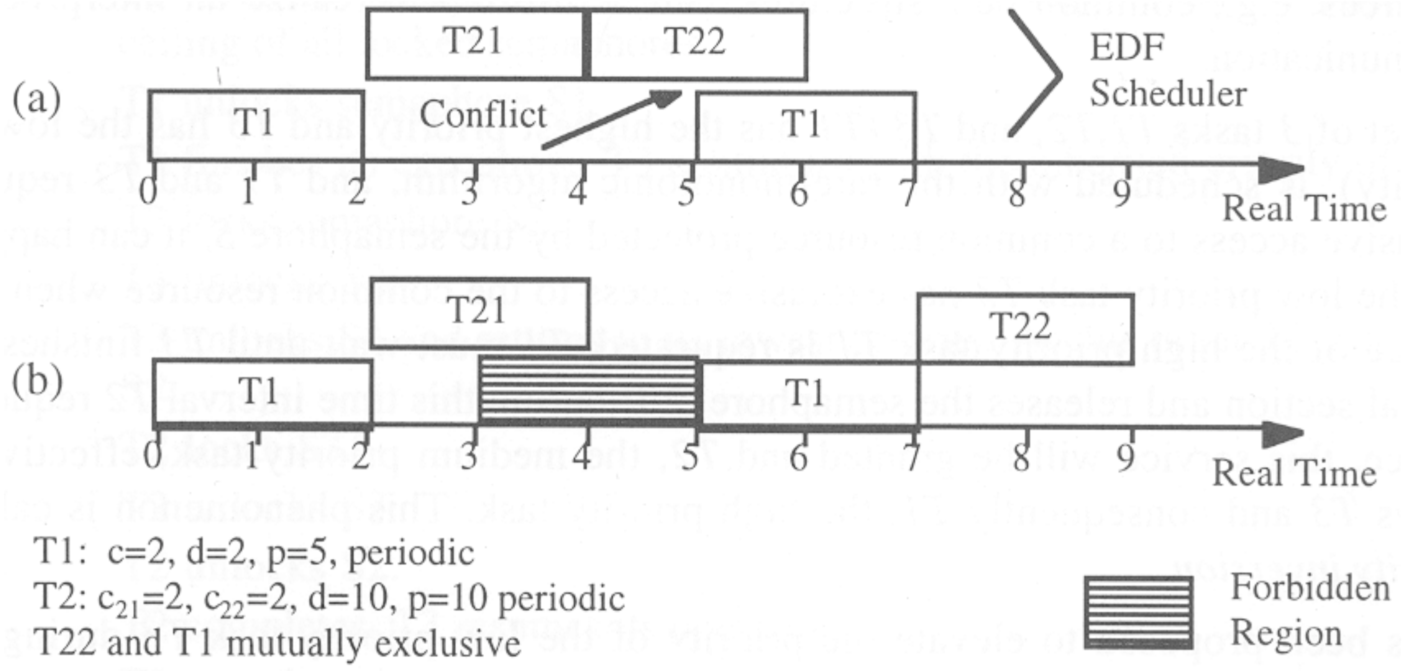
- Providing extra resources such that simpler sufficient schedulability tests and algorithms can be applied.
- Dividing the scheduling problem into two parts, such that one part can be solved offline at compile time and only the second (simpler) part must be solved at run time.
- Introducing restricting assumptions concerning the regularity of the task set.

The second and third alternatives point towards a more static solution of the scheduling problem.

9.3 Dynamic Scheduling of Periodic Task Sets

The kernelized monitor: the kernelized monitor algorithm allocates the processor time in uninterruptible quanta of duration q, where q is larger than the longest critical section of all tasks. Thus all critical sections can complete within this single uninterruptible time quantum.

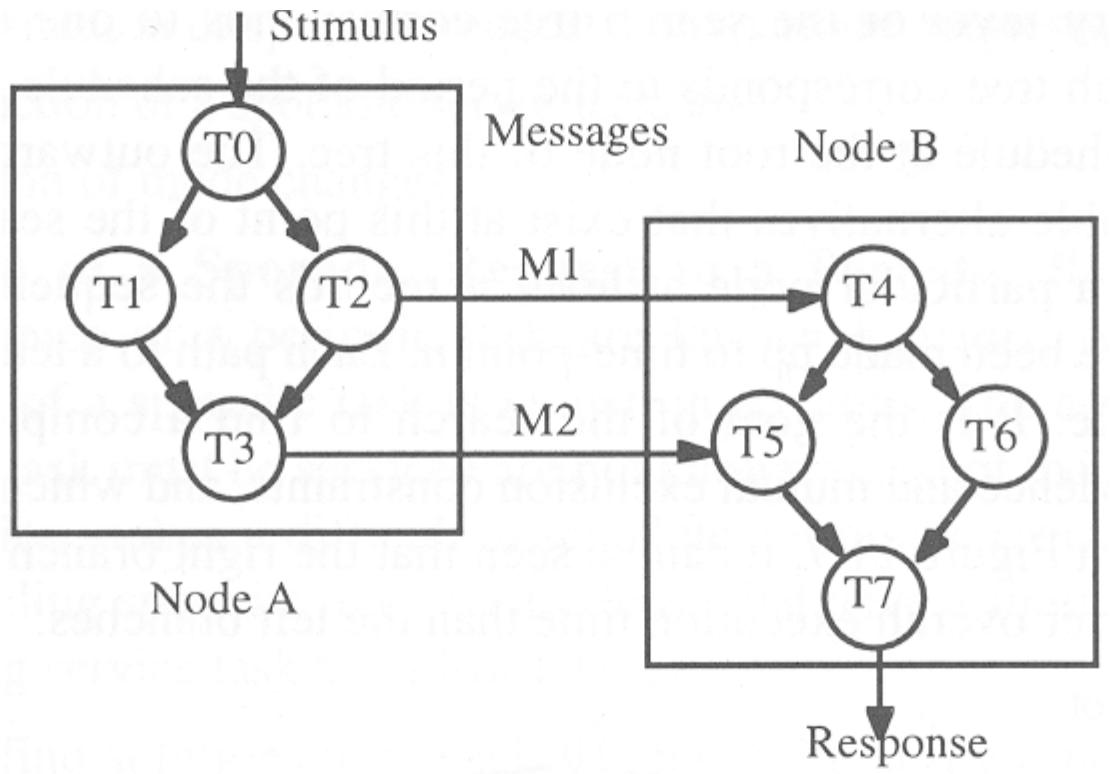
Example with q of two time units:



During compile time, all forbidden regions must be determined and passed to the dispatcher so that the dispatcher will not schedule any unwanted critical sections in the forbidden region.

9.4 Static Scheduling of Periodic Task Sets

In static or off-line scheduling, a feasible schedule of a set of tasks is calculated off-line. The precedence relations between tasks executing in the different nodes can be expressed in the form of a precedence graph:



Static Scheduling Viewed as Search

Static scheduling is based on regularity assumptions about the points in time future service request will be honoured.

The occurrence of external events is not under the control of the computer system; however, the points in time when these events be serviced can be established a priori. To this purpose, a suitable sampling rate is established for each class of events.

During system design it must be ascertained that the sum of the maximum delay times until a request is recognized by the system plus the maximum transaction response time is smaller than the specified service deadline.

The role of time: A static schedule is a periodic time-triggered schedule. The timeline is partioned into a sequence of basic granules, the basic cycle time.

There is only one interrupt in the system: a periodic clock interrupt denoting the start of a new basic granule. In a distributed system, this clock interrupt must be globally synchronized to a precision that is much better than the duration of a basic granule.

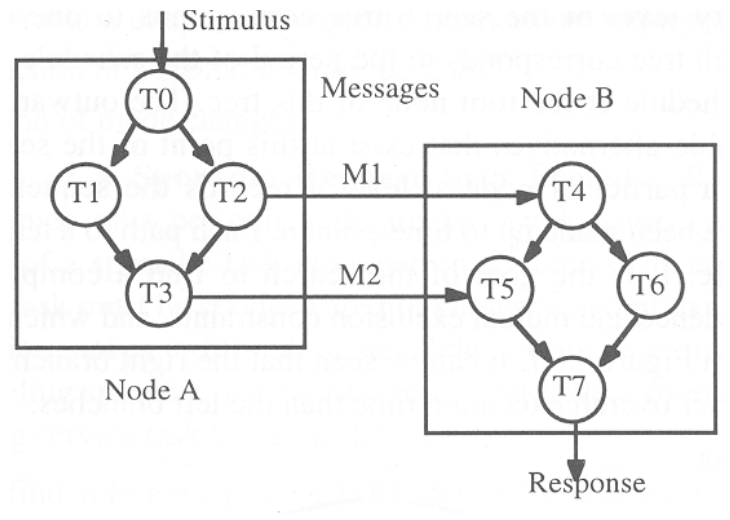
Every transaction is periodic, its period being a multiple of the basic granule. The least common multiple of all transaction periods is the schedule period.

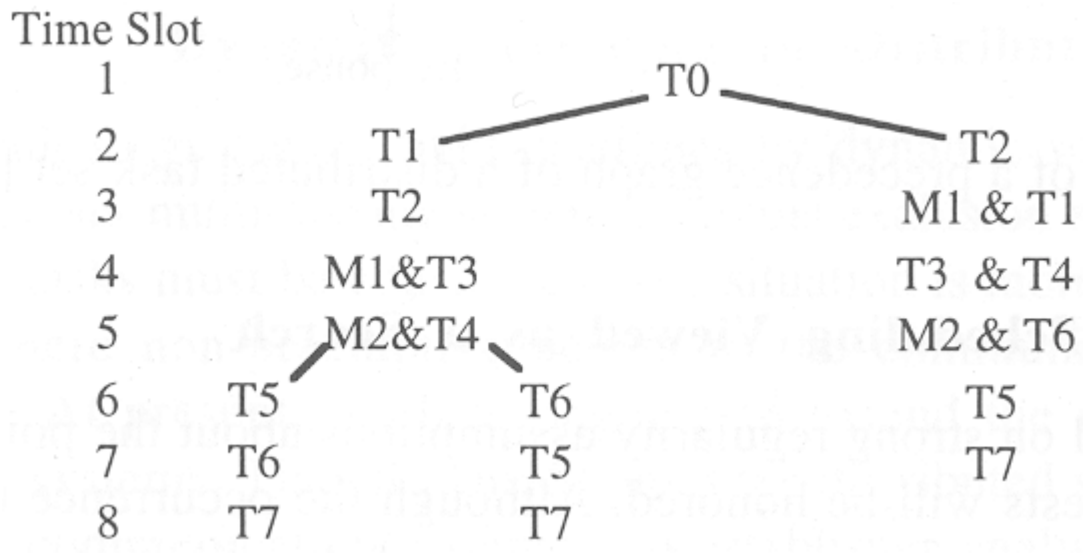
At compile time, the scheduling decision for every point of the schedule period must be determined and stored in a dispatcher table for the operating system.

At run time, the preplanned decision is executed by the dispatcher after every clock interrupt.

9.4 Static Scheduling of Periodic Task Sets

The search tree: The solution to the scheduling problem can be seen as finding a path, a feasible schedule, in a search tree by applying a search strategy. Example for the precedence graph shown on the left:





It can be seen that the right branch of the tree leads to a shorter overall execution time than the left branches.

Bratley's Algorithm

Bratley's algorithm solves the problem of finding a feasible schedule for a set of non-preemptive tasks with arbitrary (known) request times.

The algorithm starts with an empty schedule, and, at each step of the search, visits a new vertex and adds a task to the partial schedule. The algorithm uses a pruning technique to determine when a current search can be abandoned.

A branch is abandoned when

- the addition of any node to the current path causes a missed deadline
- a feasible schedule is found at the current path

The worst case complexity of this algorithm is $O(n \cdot n!)$, thus it can only be used in off-line mode, when all task parameters (including arrival times) are known. This can be the case in time-triggered systems, where tasks are activated at predefined instants by a timer process.

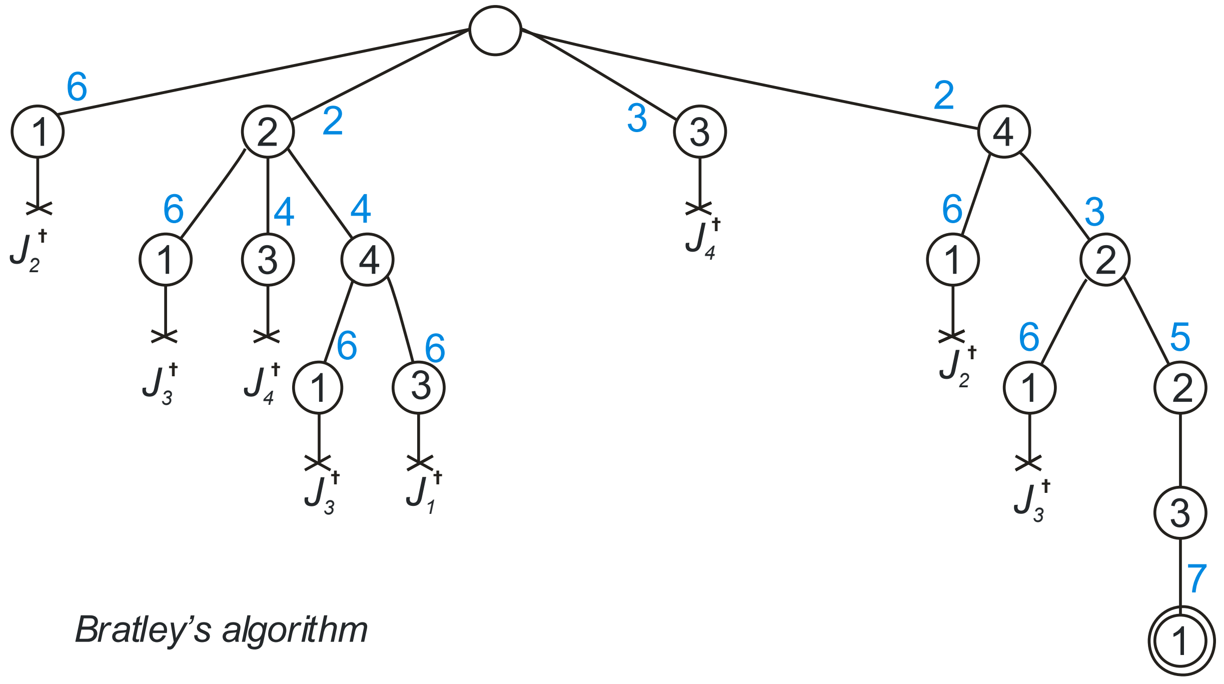
Example see next slide:

	J_1	J_2	J_3	J_4
r_i	4	1	1	0
C_i	2	1	2	2
D_i	7	5	6	4

Number in the node = scheduled task

Number outside the node = finishing time

$J_i^\dagger$ = *task that misses ist deadline*



Bratley's algorithm

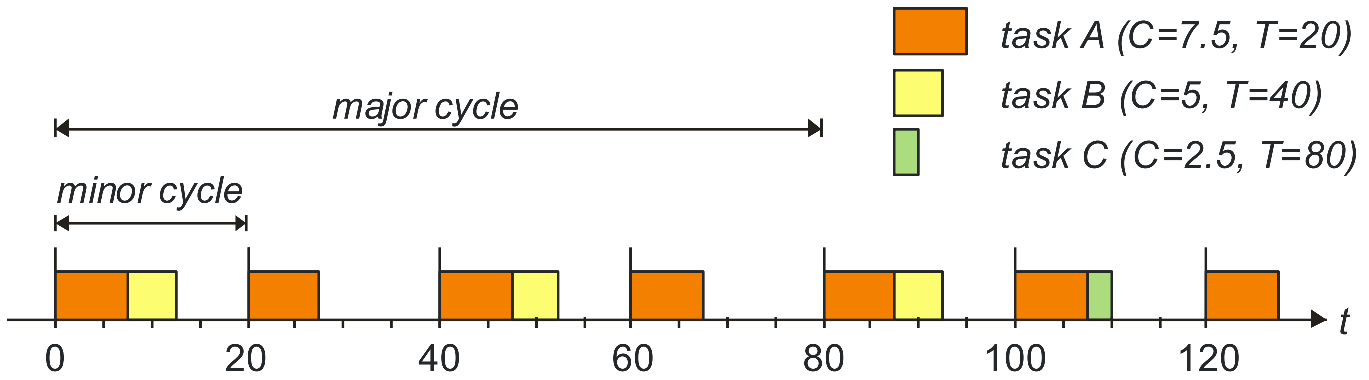
9.4 Static Scheduling of Periodic Task Sets

Timeline scheduling (cyclic scheduling)

Timeline scheduling is one of the most used approaches to handle periodic tasks in defence military systems and traffic control systems.

The time axis is divided into slices of equal length, called a minor cycle. Within a minor cycle, one or more tasks can be allocated off-line for execution. The minor cycle is the greatest common divisor (GCD) of the task periods.

A timer synchronizes the activation of the tasks at the beginning of each time slice.
The minimum period after which the schedule repeats itself is called a major cycle. The major cycle is the least common multiple of the task periods.



To guarantee a priori that a schedule is feasible on a particular processor, it is sufficient to know the task worst-case execution times and verify that the sum of the executions within each time slice is less than or equal to the minor cycle.

The major advantage of timeline scheduling is its simplicity. There is no need for preemption, a simple interrupt routine suffices.

9.4 Static Scheduling of Periodic Task Sets

On the other hand, timeline scheduling comes with a number of issues:

- It is fragile during overload conditions, e.g. if a task does not terminate at the minor cycle boundary.
- It is sensitive to application changes. The entire schedule may have to be revised. For example, if the period of task B above would have to be changed to 50 time units, the minor cycle would have to be changed to 10, and the major cycle would have to be changed to 100.
- It is difficult to handle aperiodic activities efficiently without changing the task sequence.

Increasing the Flexibility in Static Schedules

One of the weaknesses of static scheduling is the assumption of strictly periodic tasks. Even though the majority of tasks in hard real-time applications is periodic, there are also sporadic requests for service that have hard deadline requirements.

The following three methods increase the flexibility of static scheduling:

- The transformation of sporadic requests into periodic requests
- The introduction of a sporadic server task (see next chapter)
- The execution of mode changes

Transformation of a sporadic request to a periodic request: For sporadic requests there is no knowledge about future request times, but the minimum interarrival time is known in advance. This makes it difficult to schedule sporadic requests before run time.

A possible solution to this problem is the replacement of an independent sporadic task τ with laxity l by a pseudo-periodic task τ':

Parameter	Sporadic task	New pseudo-periodic task
Computation time C	C	$C'=C$
Deadline interval D	D	$D'=C$
Period T	T	$T'=min(l-1,T)$

This transformation guarantees that the sporadic task will always meet its deadline if the pseudo-periodic task can be scheduled.

9.4 Static Scheduling of Periodic Task Sets

Mode changes: Most real-time applications operate in a number of different modes. For example, the flight control system of an airplane requires a different set of services when the plane is taxiing on the ground than when the plane is in the air.

Resources can be better utilized if different schedules are employed for each of these operating modes, and only tasks are considered that are active in each mode.

Points to Remember