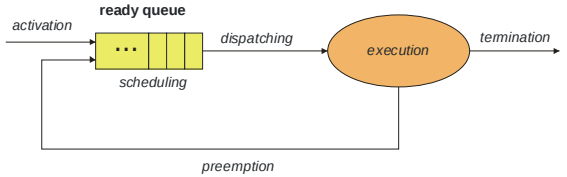
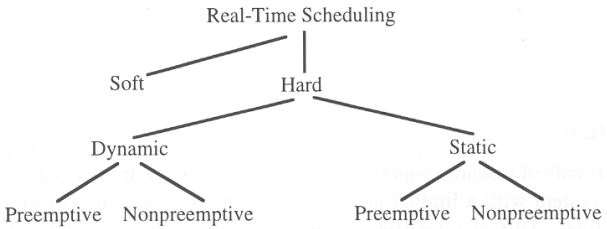
Chapter 11
Real-Time Scheduling

11.1 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



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.

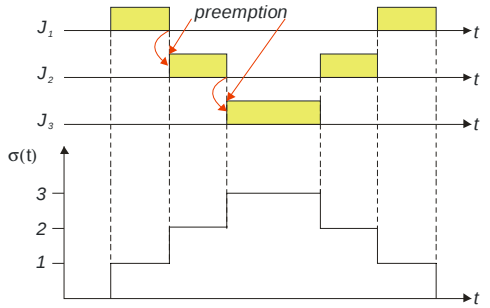
Preemptive versus non-preemptive scheduling:

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.

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.



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.

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

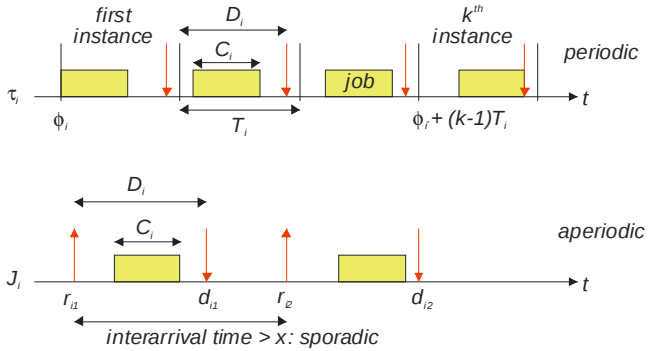
In case of task dependency, such as a shared resource, the complexity of an schedulability test is problem, and is thus computationally intractable.

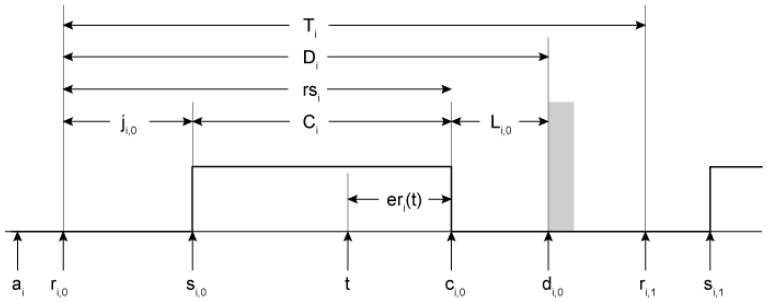
11.2 The Adversary Argument

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





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.

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. 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 $p_i, i = 1,2,3,\ldots$.

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.

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

11.3 Dynamic Scheduling

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.

The following assumptions are made about the task set:

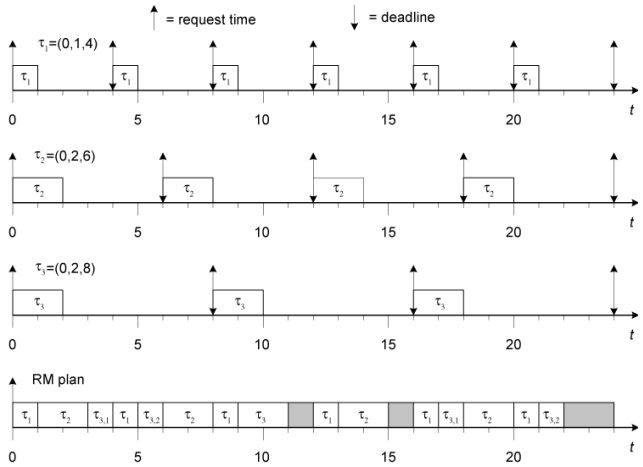
- The requests for all tasks of the task set $\{\tau_i\}$ for which hard deadlines exist are periodic.
- All tasks are independent of each other. There exist no precedence constraints or mutual exclusion constraints between any pair of tasks.
- The deadline interval of every task τ_i is equal to its 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

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

  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.

Example for RM plan and three tasks:



Earliest Deadline First (EDF) Algorithm

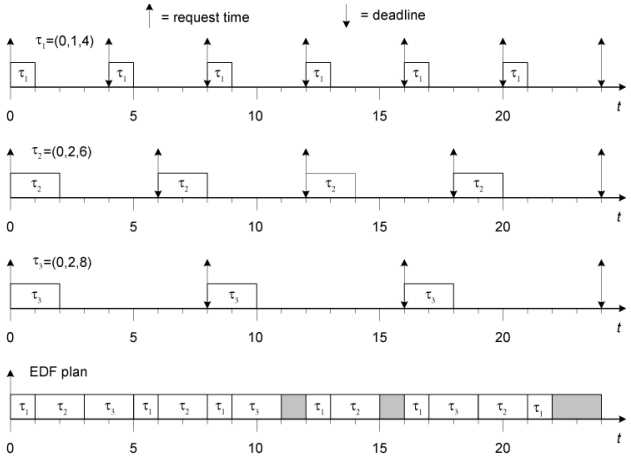
The earliest deadline first algorithm is an optimal dynamic preemptive algorithm in single processor systems, which is based on dynamic priorities.

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.

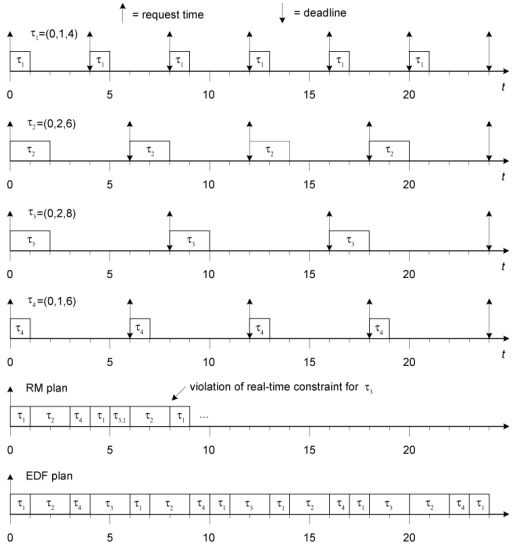
It makes the following assumptions about the task set (same as for the RM algorithm):

- 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.

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



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 off-line 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.

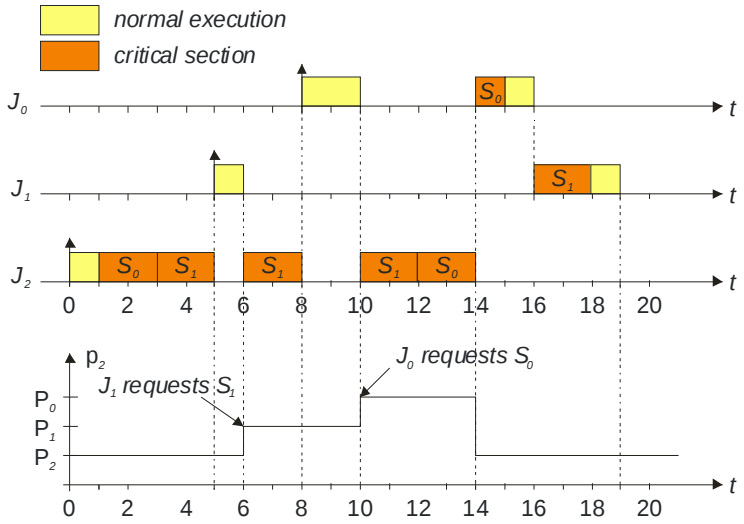
The second and third alternatives point towards a more static solution of the scheduling problem.

The Priority Inheritance Protocol

The priority inheritance protocol can be defined as follows:

- Jobs are scheduled based on their active priorities. Jobs with the same priority are executed on a first come first serve basis.
- When job J_i tries to enter a critical section $z_{i,j}$ and resource $R_{i,j}$ is already held by a lower priority job, J_i will be blocked. J_i is said to be blocked by the task that holds the resource. Otherwise, J_i enters the critical section $z_{i,j}$.
- When a job J_i is blocked on a semaphore, it transmits its active priority to the job, say J_k, that holds that semaphore. Hence, J_k resumes and executes the rest of its critical section with priority $p_k = p_i$. J_k is said to inherit the priority of J_i. In general, a task inherits the highest priority of the jobs blocked by it.
- When J_k exits a critical section, it unlocks the semaphore, and the highest-priority job, if any, blocked on that semaphore is awakened. Moreover, the active priority of J_k is updated as follows: if no other jobs are blocked by J_k, p_k is set to its nominal priority P_k, otherwise it is set to the highest priority of the jobs blocked by J_k.
- Priority inheritance is transitive; that is if a job J_3 blocks a job J_2, and J_2 blocks a job J_1, then J_3 inherits the priority of J_1 via J_2.

See the following example with three jobs with priorities $P_0 > P_1 > P_2$.



This can be simplified to a less tight condition

The Priority Ceiling Protocol

The priority ceiling protocol can be defined as follows:

- Each semaphore S_k is assigned a priority ceiling $C(S_k)$ equal to the priority of the highest-priority job that can lock it. $C(S_k)$ is a static value that can be computed off-line.
- Let J_i be the job with the highest priority among all jobs ready to run; thus J_i is assigned the processor.
- Let S^* be the semaphore with the highest priority ceiling among all the semaphores currently locked by jobs other than J_i, and let $C(S^*)$ be its ceiling.
- To enter a critical section guarded by a semaphore S_k, J_i must have a priority higher than $C(S^*)$. If $P_i \le C(S^*)$, the lock on S_k is denied and J_i is said to be blocked on semaphore S* by the job that holds the lock on S^*.
- When a job J_i is blocked on a semaphore, it transmits its priority to the job, say J_k, that holds that semaphore. Hence, J_k resumes and executes the rest of its critical section with the priority of J_i. J_k is said to inherit the priority of J_i.
- When J_k exits a critical section, it unlocks the semaphore and the highest-priority job, if any, blocked on that semaphore is awakened. Moreover, the active priority of J_k is updated as follows: if no other jobs are blocked by J_k, p_k ist set to the nominal priority P_k; otherwise, it is set to the highest priority of the jobs blocked by J_k.
- Priority inheritance is transitive; that is if a job J_3 blocks a job J_2, and J_2 blocks a job J_1, then J_3 inherits the priority of J_1 via J_2.

See the following example with three jobs with priorities $P_0 > P_1 > P_2$.

- Job J_0 sequentially accesses two critical sections guarded by semaphores S_0 and S_1
- Job J_1 accesses only a critical section guarded by semaphore S_2
- Job J_2 uses semaphore S_2 and then makes a nested access to S_1.

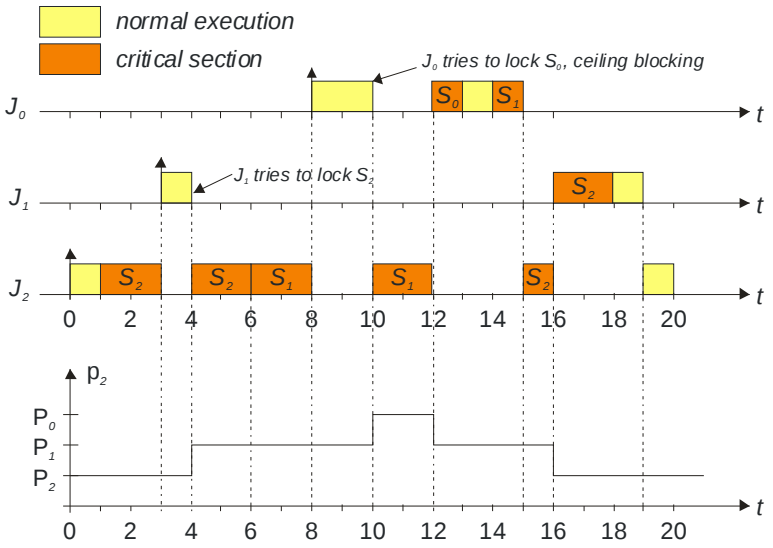
The semaphores are thus assigned the following priority ceilings:

- $C(S_0) = P_0$, since J_0 is the highest-priority task that tries to lock S_0
- $C(S_1) = P_0$, since J_0 is the highest-priority task that tries to lock S_1
- $C(S_2) = P_1$, since J_1 is the highest-priority task that tries to lock S_2

The activation times of the three jobs are as follows:

- J_0 is activated at $t = 8$
- J_1 is activated at $t = 3$
- J_2 is activated at $t = 0$

See the following diagram for the scenario.



Scenario description:

- At time 0, J_2 is activated, and since it is the only job ready to run, it starts executing and later locks semaphore S_2.
- At time 3, J_1 becomes ready and preempts J_2.
- At time 4, J_1 attempts to lock S_2, but it is blocked by the protocol because P_1 is not greater than $C(S_2)$. Then, J_2 inherits the priority of J_1 and resumes its execution.
- At time 6, J_2 successfully enters its nested critical section by locking S_1. Note that J_2 is allowed to lock S_1 because no semaphores are locked by other jobs.
- At time 8, while J_2 is executing at a priority $p_2 = P_1$, J_0 becomes ready and preempts J_2 because $P_0 > p_2$.
- At time 10, J_0 attempts to lock S_0, which is not locked by any job. However, J_0 is blocked by the protocol because its priority is not higher than $C(S_1)$, which is the highest ceiling among all semaphores currently locked by the other jobs. Since S_1 is locked by J_2, J_2 inherits the priority of J_0 and resumes its execution.
- At time 12, J_2 exits its nested critical section, unlocks S_1, and, since J_0 is awakened, J_2 returns to priority $p_2 = P_1$. At this point, $P_0 > C(S_2)$; hence, J_0 preempts J_2 and executes until completion.
- At time 15, J_0 is completed, and J_2 resumes its execution at a priority $p_2 = P_1$.
- At time 16, J_2 exits its outer critical section, unlocks S_2, and, since J_1 is awakened, J_2 returns to its nominal priority P_2. At this point, J_1 preempts J_2 and executes until completion.

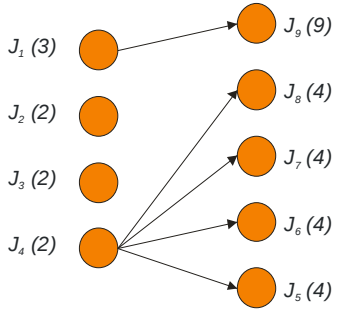
- At time 19, J_1 is completed; thus J_2 resumes its execution.

Scheduling Anomalies in Dependent Task Sets

When dealing with task sets with precedence relations executed in a multiprocessor environment we encounter some scheduling anomalies:

- adding resources (such as a processor) can make things worse
- relaxing constraints such as less precedence between tasks or lower execution time requirements can make things worse

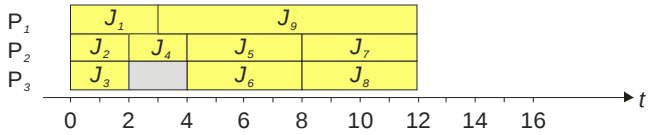
This is illustrated in the following example with nine tasks and the following precedence relationships (values in parentheses are execution times):



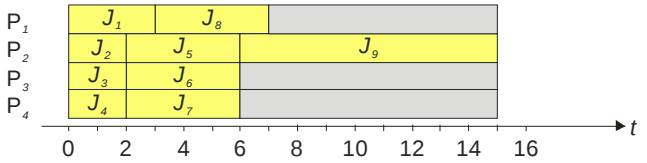
$priority(J_i) > priority(J_j) \quad \forall\, i < j$

values in parantheses are execution times

First case: optimal schedule on three processors P_1, P_2, P_3. Global completion time is 12.

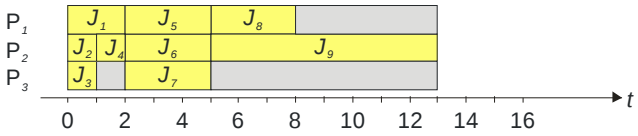


Second case: we add a processor P_4. Tasks are allocated to the first available processor. Global completion time has increased to 15.

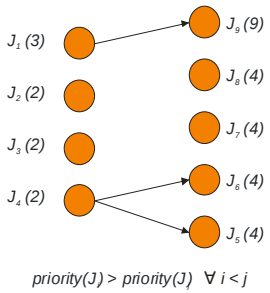


Increase to four processors
Allocation to the first available processor

Third case: we reduce all computation times by one time unit. Tasks are allocated to the first available processor. Global completion time has increased to 13.



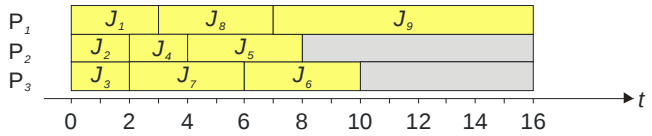
Computation time reduced by one time unit



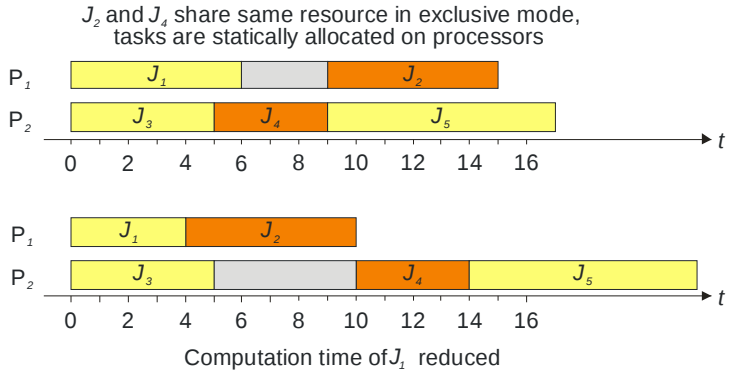
$priority(J_i) > priority(J_j) \quad \forall\, i < j$

values in parentheses are execution times

Fourth case: we remove some precedence constraints.
Global completion time has increased to 16.

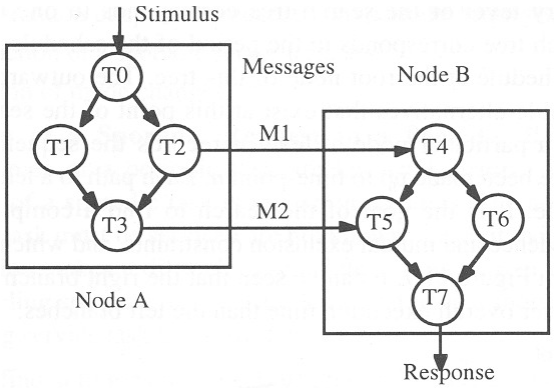


Fifth case: we share a resource in exclusive mode, and reduce computation time. Global completion time increases from 16 to 22.



11.4 Static Scheduling

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.

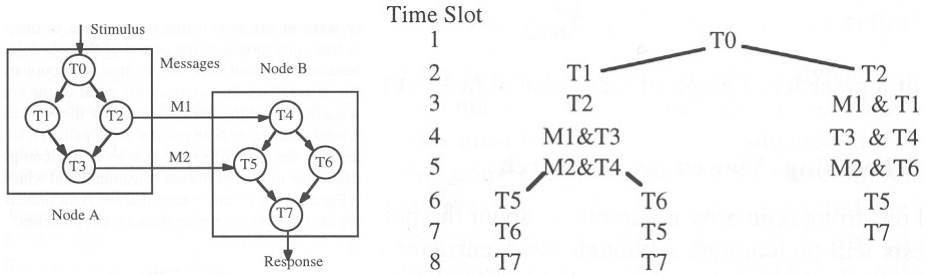
There is only one interrupt in the system: a periodic clock interrupt denoting the start of a new basic granule. In a distributed system, this clock interrupt must be globally synchronized to a precision that is much better than the duration of a basic granule.

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.

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.

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 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.

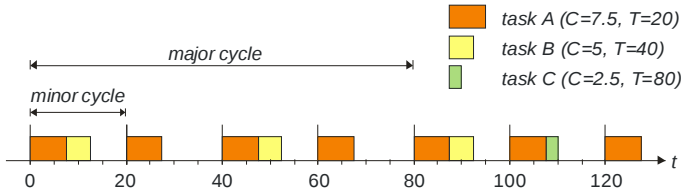
Mode changes: Most real-time applications operate in a number of different modes. For example, the flight control system of an airplane requires a different set of services when the plane is taxiing on the ground than when the plane is in the air.

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.

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.

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.

Points to Remember