Realtime Systems - Job Scheduling

1 Until now

we have:

- defined the concept of real-time systems
- identified a number of real-life systems as being real-time systems
- characterized real-time systems as systems with a large number of different interfaces to the physical nature
- setting forth specifications and demands to the real-time behaviour of the system.

Last lecture focussed on how to approach system development, i.e. analysis and design when we are dealing with real-time systems.

The CODARTS method was presented along with its different phases:

- Environmental model summarized in a System Context Diagram (SCD)
- Subsystems identified according to loaction or functional similarity.
- I/O and internal objects identified.
- Functional modelling by control and data transforms.
- Task structuring according to temporal, sequential or control cohesion.

2 Next

At this point we imagine to be at a point in the design process where a first tasks tructuring has taken place. The different transformations have all been associated to exactly one task executing in (pseudo) parallel to a number of other tasks in the system.

From task structuring we know time characteristics of the taskset $\{\tau_1, .., \tau_N\}$, i.e. the ready times $r_{i,j}$, computation times $c_{i,j}$ and deadlines $d_{i,j}$ of jobs $J_{i,j}$ (where indices i, j refer to the j th. job of the i th. task, ie. τ_i.

We know if tasks are periodic in which case we know the period T_i or aperiodic in which case we may know a minimum interarrival time $Tmin_i$.

The problem to solve is to find a principle by which to *share* the CPU time of a single CPU among the tasks in the set. Obviously such a sharing principle should not unnecessarily force any task to miss some deadline. That is if at all possible all deadlines should be met.

3 Scheduling

Sharing of CPU time among tasks is managed by a so called *scheduler*, which is a basic part of the real time execution platform. A scheduler is the part of the real time environments that either define or manage a so called *schedule*. A schedule comprises the information of which job to run at which time from now on and in all relevant future. As an example your course schedule tells you at which time you should attend which course during the whole semester. A schedule for a given taskset is said to be *feasible* if all jobs are executed completely between their readytimes and deadlines respectively.

The theory of *scheduling* deals with two questions; how to design a schedule for a given taskset and how to guarantee feasibility of the schedule. In most cases schedules are chosen according to some well known scheduling principle and then subsequently analysed for feasibility.

First schedulers may allow preemption or not. The reason for non-premptive scheduling is mainly simplicity and especially the fact, that schedulers may be written without reference to any specific CPU, i.e. *hardware independent*.

Schedulers then are categorized according to the principles for time sharing. A number of different categories exist:

- Fixed schedules: Schedules derived at pre run-time, rather simple and effective, but less flexible.
- Round robin: Fixed timeslots are granted periodically, simple and flexible but less effecitve.
- Fixed priority: CPU-time is granted to the highest priority among active tasks, simple and flexible, medium efficiency.
- Dynamic priorities: CPU-time is granted according to runtime criteria, complex, flexible and effective.

As mentioned in the above list, different scheduling disciplines may have quite different characteristics. Some are simple, that is they are easily implemented and lightly run. Some are complex and may introduce heavy computational overhead at run time. Some are very flexible so that additional tasks are easily introduced and some are not. Some are effective, i.e. they may produce feasible schedules where others may fail. We shall in the following investigate the effectiveness of different scheduling disciplines starting of with a coarse analysis.

4 Overall results.

We assume for the moment that all processes embed exactly one periodic task, where: period, computation time and deadlines are well defined for all jobs.

The notation is as follows:

- Task set $\tau_1, \tau 2, .., \tau_N$
- Jobs in task $\tau_i = \{J_{i,1}, J_{i,2}, .., J_{i,M_i}\}$

The job $J_{i,j}$ is characterized by readytime: $r_{i,j}$, computation time: $c_{i,j}$ and deadline: $d_{i,j}$.

We assume all tasks to be periodic, i.e. ready times for a given task fall equidistantly in time

$$r_{i,j+1} = r_{i,j} + T_i \tag{1}$$

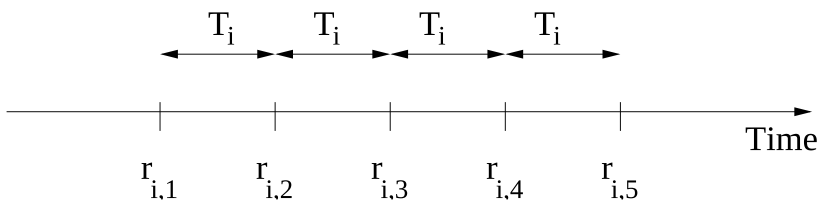

Figure 1: Equidistant ready times for a periodic task.

Also we assume constant *relative* deadlines as well as computation times, i.e. all deadlines $d_{i,j} = r_{i,j} + d_i$ and $c_{i,j} = c_i$ forall $j = 1, 2, ..$

Given the above we wish to guarantee all deadlines to be met, i.e.

$$C_{i,j} \leq d_{i,j} = r_{i,j} + d_i \tag{2}$$

First we define the time function $R(t)$ mapping any instant in time to the amount of work (in CPU time-units) remaining in the tasks ready to execute. At any instant in time $R(t)$ is the total remaining CPU-time of ready unfinished jobs.

Whenever time passes a ready time $r_{i,j}$ $R(t)$ increases discontinously by c_i, as shown in figure 2. We say that a task requests an amount of CPU time.

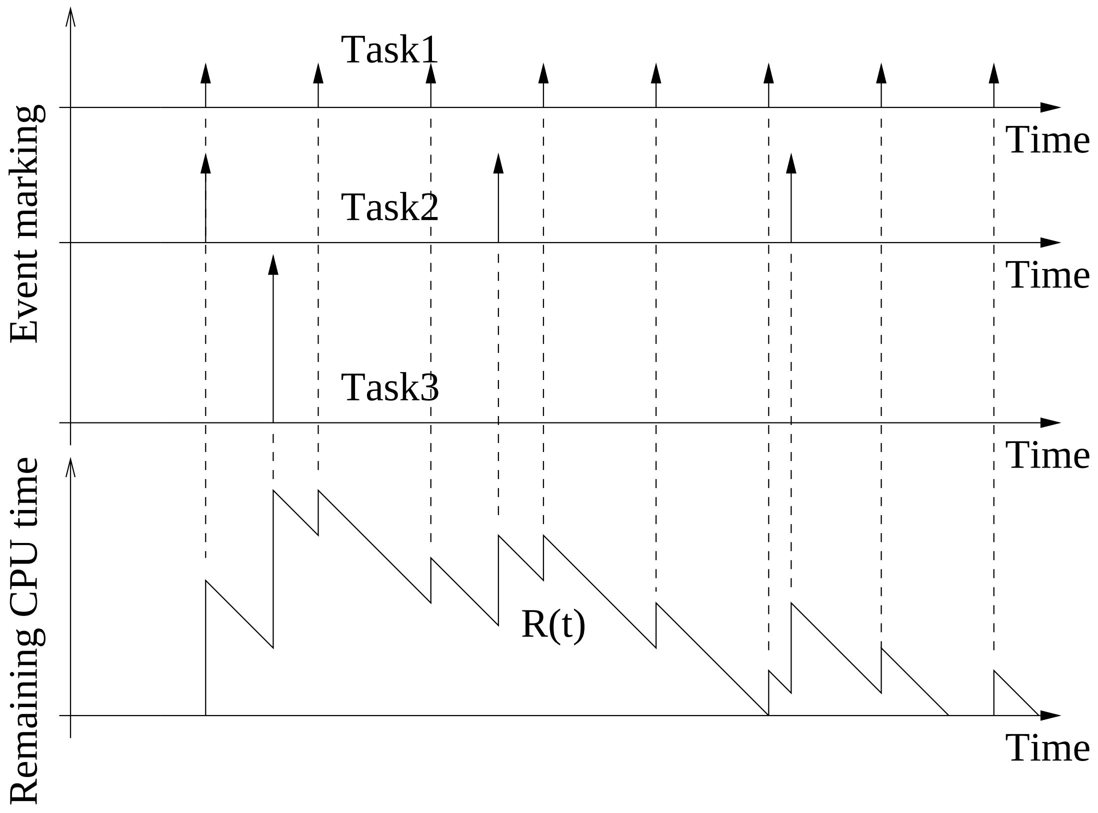

Figure 2: Remaining CPU workload R(t) as a function of time.

At all other times $R(t)$ decreases with a constant slope of 1 as long as $R(t) > 0$. This is when the CPU as executing requested work.

When $R(t)$ reaches 0 it remains zero until the next ready time comes up. In this period the CPU is idle, i.e. it runs but does not execute any requested work.

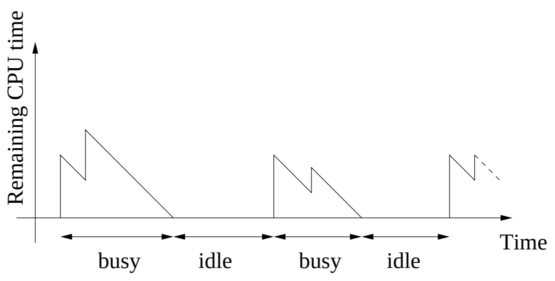

Figure 3: Interchanging busy and idle periods.

Any time interval $[a, b]$ in which $R(t) > 0$ is called a *busy period*. Any interval in which $R(t) = 0$ is called an *idle period.*. During its lifetime a CPU continously changes between being idle and busy.

Consider 1 single task τ with period T and execution time c executing alone. In order for the situation to be stable c needs to be lower than T or *utilization* $U = \frac{c}{T} < 1$. If $c > T$ the second job is ready before the 1. job has completed and is therefore scheduled later. The third job experiences even more delay and so on. The amount of unfinished work increases towards infinity.

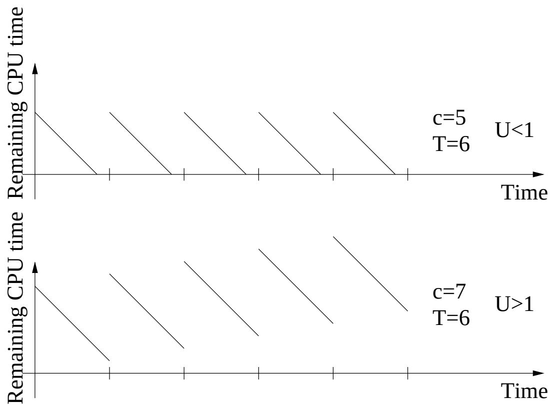

Figure 4: Utilizations below and above 1.

We shall define CPU utilization U generally as follows:

$$U = \sum_{i=1}^{N} \frac{c_i}{T_i} \tag{3}$$

Now within a busy period beginning at time 0, $R(t)$ can be found from

$$R(t) = L(t) - t \tag{4}$$

(since the CPU executes continously from 0 to t within a busy period) where $L(t)$ denotes the amount of work requested by all tasks within $[0, t]$. We may find this amount by adding the amount requested by all tasks $L_i(t)$

$$L(t) = \sum_{i=1}^{N} L_i(t) \tag{5}$$

where

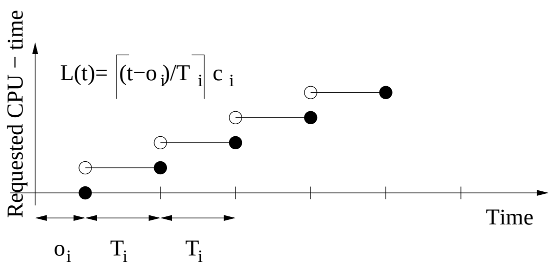

Figure 5: Work amount requested by single task.

$$L_i(t) = \lceil \frac{t - o_i}{T_i} \rceil \cdot c_i \tag{6}$$

where $\lceil x \rceil$ denotes the smallest integer above x, and o_i is the time for the 1. request from τ_i.

Since $0 \le o_i < T_i$ it may be concluded that

$$\begin{aligned} L(t) &\le \sum_{i=1}^{N} \lceil \frac{t}{T_i} \rceil \cdot c_i \le \sum_{i=1}^{N} (\frac{t}{T_i} + 1) \cdot c_i \\ &= \sum_{i=1}^{N} c_i + t \cdot \sum_{i=1}^{N} \frac{c_i}{T_i} = \sum_{i=1}^{N} c_i + t \cdot U \end{aligned} \tag{7}$$

So we may conclude that within a busy period

$$R(t) \le \sum_{i=1}^{N} c_i + (U - 1) \cdot t \tag{8}$$

Where the right hand side is a decreasing function for $U < 1$.

That is for $U < 1$ $R(t)$ will reach 0 before $Tmax$ given by

$$Tmax = \frac{\sum_{i=1}^{N} c_i}{1 - U} \tag{9}$$

so that no busy period will last longer than $Tmax$. Any relative deadline d_i less than $Tmax$ will then be met.

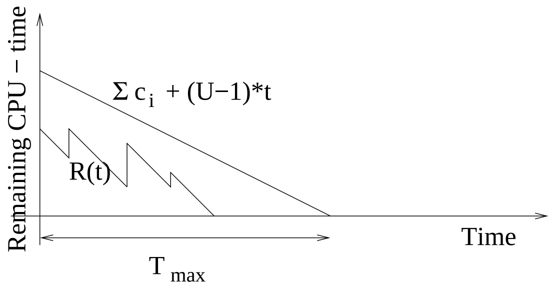

Figure 6: Stable upper bound for R(t)

However we may likewise show that

$$R(t) \geq (U - 1) \cdot t - \sum_{i=1}^{N} c_i \tag{10}$$

so that for $U > 1$ the remaining work increases towards ∞. After some time some deadlines most certainly will fail.

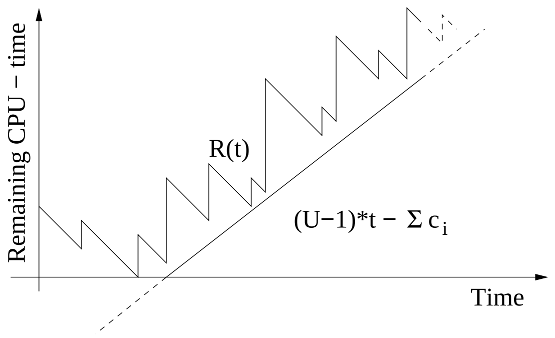

Figure 7: Unstable lower bound for R(t)

The overall conlusion is that the CPU utilization U determines whether it is possible to meet any deadlines at all or if some deadlines are certain to fail.

It is important to notice that the above analysis is valid no matter how we choose the order of execution, that is whatever scheduling algorithm we may choose to implement. However the time for switching between tasks, the so called *context switching time* is not taken into account.

The above constitutes the first coarse analysis towards answering the question of whether a given taskset is guaranteed to meet its deadlines under some specifik scheduling discipline. We are still not capable of giving a precise answer but we are approaching the goal.

4.1 Example 1

Imagine 3 tasks τ_1, τ_2, τ_3 with periods $d_1 = T_1 = 4, d_2 = T_2 = 12, d_3 = T_3 = 64$ and computation times $c_1 = 2, c_2 = 2, c_3 = 6$. In this example $U = 0.76$ so that the requirement for stability is met. However as seen on figure 8 if we apply the round-robin scheduling method, where tasks are granted CPU time on a periodic basis, with period 2, τ_1 is too late after a few periods.

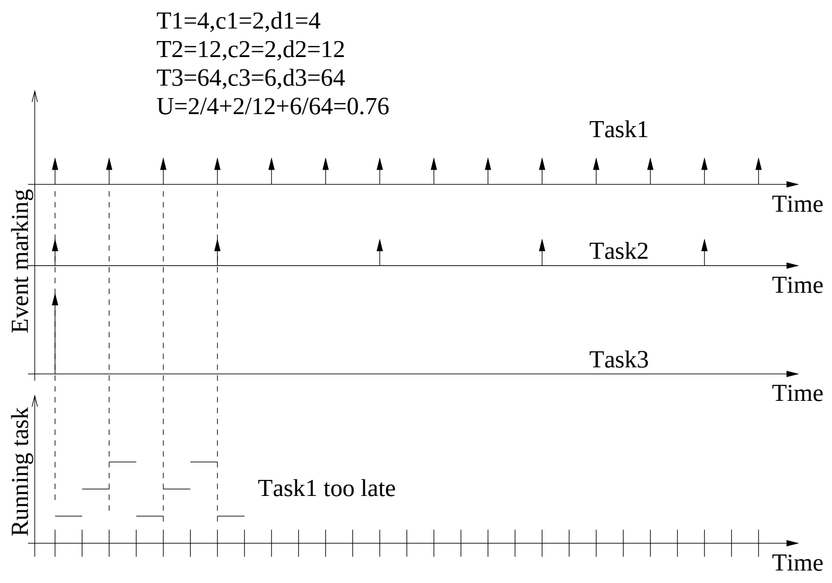

Figure 8: Taskset executed according to the round robin scheduling principle.

4.2 Example 2

Now lets try apply scheduling with fixed priority, where the taskswitching discipline is described below:

- A task is **ready** if some job allready initiated has not completed.
- Tasks are assigned different priorities, where lower number indicates higher priority.
- At all times CPU time is granted to the task which has the highest priority among the tasks that are ready.

The above example, where fixed priorities $1, 3, 2$ are assigned to tasks τ_1, τ_2, τ_3 is shown in figure 9

4.3 Example 3

Now we try to reassign priorities to be $1, 2, 3$, so that the tasks with smaller periods are assigned higher priorities. We call this scheme for priority assignment *Rate Monotonic Scheduling.* This is shown in figure 10

4.4 Example 4

It seems as if things are allright here, so we will assign priorities also according to RMA in the next example also. Here we have $d_1 = T_1 = 2, d_2 = T_2 = 3$ and $c_1 = 1, c_2 = 5/4$. The result is shown in figure 11

After a few periods τ_2 is too late even though $U = 11/12 < 1$.

We are able in an ad hoc manner to derive a fixed schedule that may outperform both round-robin and RMA but it is not easy to design and it lacks the flexibility of the former two. When we need to add an extra task to some taskset we need to redesign the schedule totally.

In the sequel we shall examine a number of different scheduling disciplines and try to establish more precise answers to the question of whether or not tasksets will meet their deadlines or not.

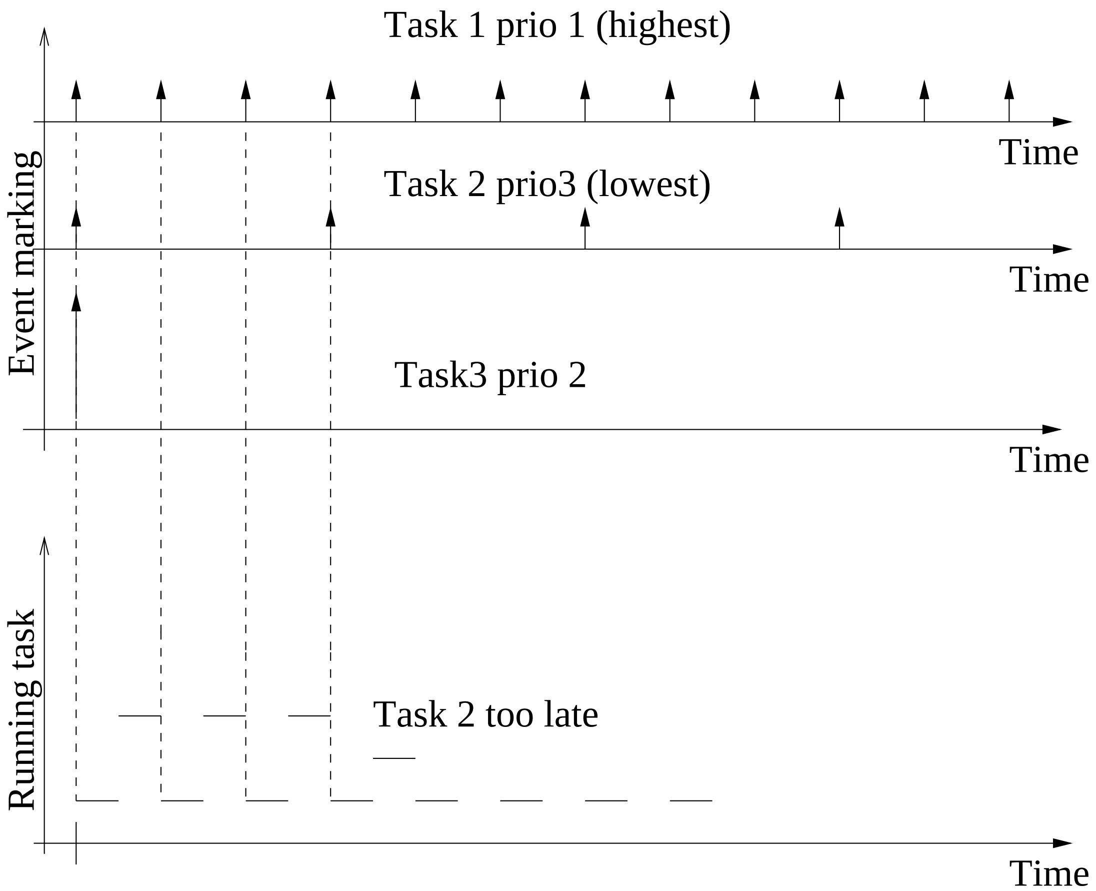

Figure 9: Taskset executed according to fixed priorities 1,3,2.

5 Fixed schedules

For fixed schedules we shall give answers only in the case where all time durations are multiples of a small common time unit T_U, that is any ready time, computation time, scheduling time, deadline as well as context switching time is given as a multiple of T_U. Furthermore ready times as well as deadlines are known absolutely.

In that case the schedule can be viewed as a table where its entry holds the state of a timeslot of length T_U . The state my assume values:

- active by task $1, .., N$

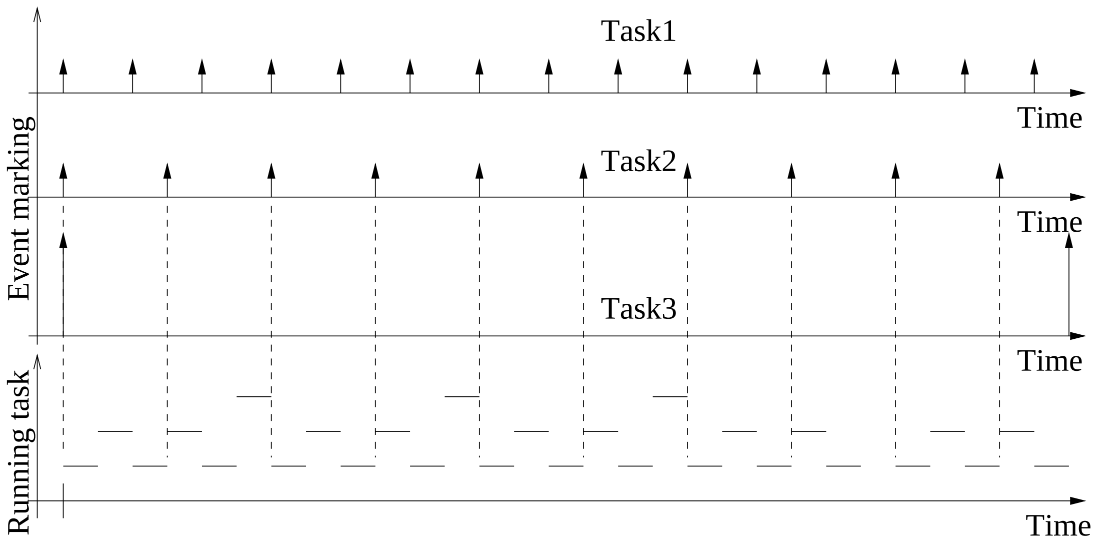

Figure 10: Taskset executed according to priorities 1,2,3

- context switching
- idle

In case of a finite number of jobs the schedule/table is finite and it may be checked by inspection.

In case of an infinite number of jobs we assume tasks to be periodic and periods be given as multiples of T_U. In this case we may need only to assign a schedule until time LCM which is the *Least Common Multiple* of task periods. If the schedule is feasible and no work remains at time LCM the same schedule is feasible for all subsequent time intervals of length LCM. The reason is that the complete pattern of ready times is periodic with period LCM. So when reaching LCM with no remaining work the situation is equivalent to the initial situation.

If we are not able to assign a feasible schedule within $[0, LCM]$ with no remaining work at LCM we may attempt to assign a feasible schedule within $[0, CM]$ with no remaining work at CM ,where CM is some common multiple of task periods. In this case the same schedule may be used in any subsequent interval of length CM.

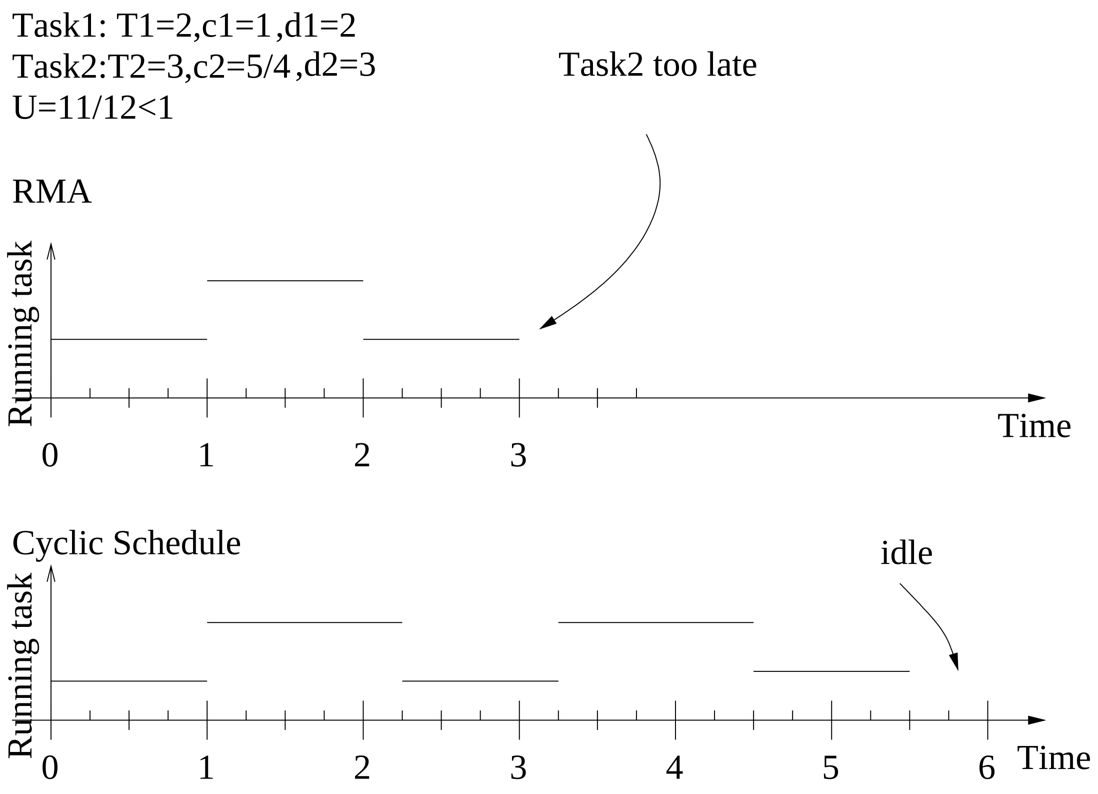

Figure 11: Taskset executed according to RMA with missed deadlines.

5.1 Example 5

We consider the taskset presented above where $T_1 = 2, T_2 = 3$ and $c_1 = 1, c_2 = 5/4$, we assume ready times to be alligned at $t = 0$, that is both tasks report ready immediately. From the numbers it is easily seen that T_U may be $1/4$ and that $LCM = 24$. We assume no context switching overhead. In figure 12 a feasible cyclic executive is shown where no work remains at LCM.

In the above example the schedule does not utilize preemption. Obviously the possibility of preempting jobs extends the possibilities of finding a feasible schedule. However one should bear in mind that every preemption adds an extra context switch to the schedule, which in some cases may cause problems.

The LCM period is some times called the *major cycle* and equals the duration of the largest period T if T is a multiple of all other periods in the set. The smallest period may be used as the so

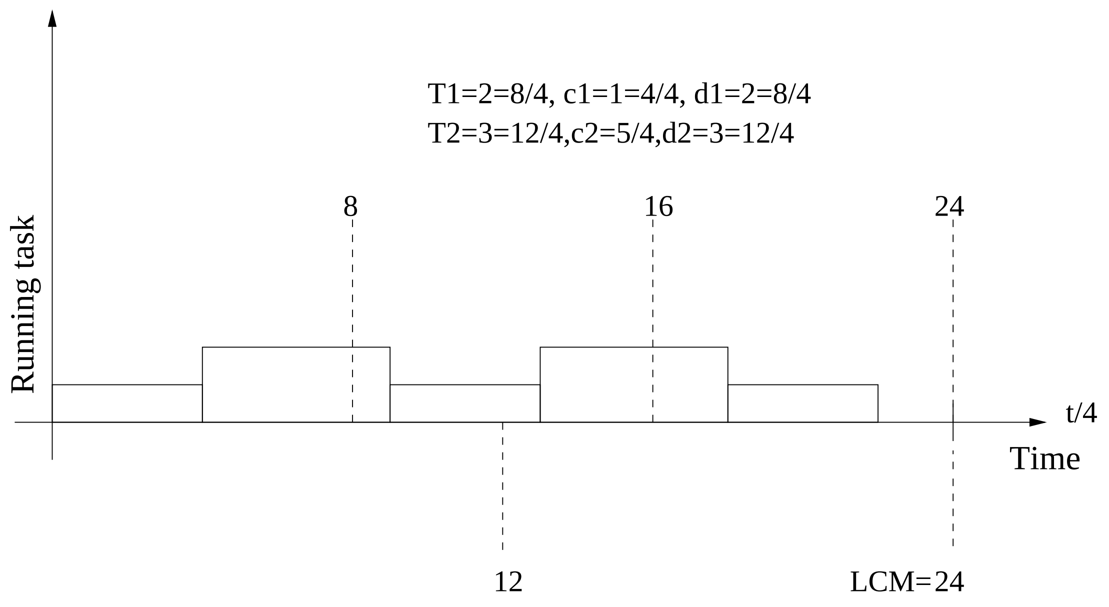

Figure 12: Cyclic exeutive in an LCM period.

called *minor cycle*. When deriving the schedule execution is first granted to the task with the lowest period within each minor cycle. Subsequently time is granted to tasks in increasing order of period until all tasks have been scheduled. The final schedule is often referred to as a *cyclic schedule* and executed by a so called *cyclic executive*, which is a table driven scheduler. Validation of the schedule is done according to the guidelines above.

5.2 Example 6

We consider a taskset of 3 tasks where $T_1 = 50mS, T_2 = 1S, T_3 = 10S$, $c_1 = 25mS, c_2 = 100mS, c_3 = 2S$, $d_1 = 25mS, d_2 = 0.5S, d_3 = 4S$. In this case the major cycle may be $10S$ whereas the minor cycle can be chosen equal to $T_1 = 50mS$. From figure (13) it is fairly easily seen that tasks τ_1, τ_2 meet their deadlines. How about τ_3 ?

6 Round robin

A round robin schedule grants time slots of length T_S to tasks in a cirkular order. Within each time slot task switching lasting T_C has to take place. If a task τ is not ready in due time the next task

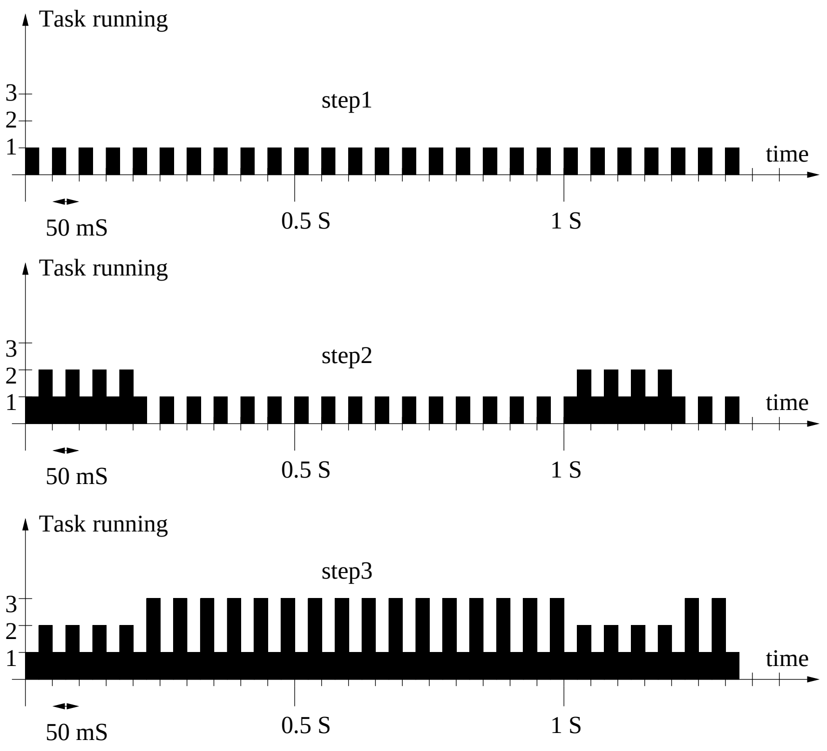

Figure 13: The steps of designing af cyclic schedule.

in order will be granted time. The task τ will, if ready then, be granted time after one rotation in the cirkular order. If a task completes before the end if a time slot task switching will take place at completion time. The work performed in a time slot relates to the remaining work as shown in figure 14

6.1 Worst case Scenario

Contrary to the case for cyclic executive we shall consider ready times and deadlines only partly known in advance. That is we know only periods and relative deadlines. However we shall assume all task to be hard realtime tasks, i.e. deadlines are less than or equal to periods. Equipped only with this knowledge the number of possible scenarios increases greatly. Therefore we aim to find a so called *worst case* scenario which may be inspected and found feasible or not. If the worst case scenario is feasible all other scenarios are guaranteed feasible as well. If not we may only state that some scenario is not feasible under the cirkumstances and no general guarantees may be given.

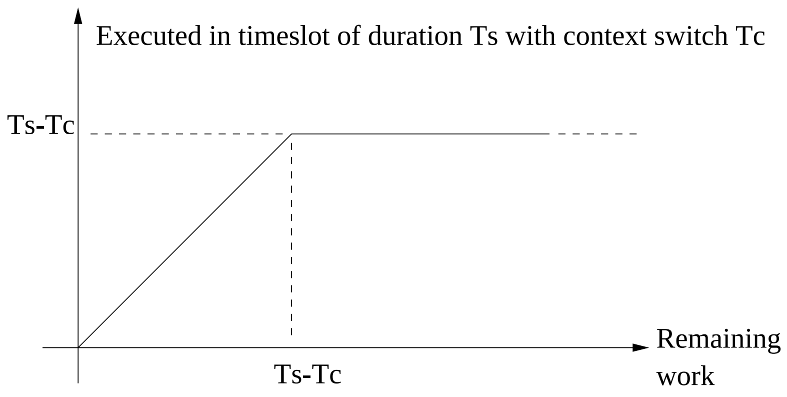

Figure 14: Relation between remaining work and performed work.

Theorem *Assume job J of task τ_i reports ready at $t = 0$ to a round robin scheduler, then J will complete no later than if at $t = 0$ the CPU is granted task τ_{i+1} (the next one in turn) and all other tasks simultaneously report ready at $t = 0$.*

As stated in the above theorem the worst case scenario for any job is when it becomes ready immediately after the CPU visited its task, and all other tasks report ready at that instant. By examining such a scenario for a given taskset and finding the round robin schedule feasible one may conclude that round robin is generally feasible for that taskset.

6.2 Example 7

Worst case scenarios for 3 tasks with the following specifications:

- τ_1: $T_1 = 3, c_1 = 1, d1 = 3$
- τ_2: $T_2 = 4, c_2 = 1, d2 = 4$
- τ_3: $T_3 = 10, c_3 = 4, d3 = 10$
- Execution order $1, 2, 3$
- Context switch time $T_C = 0$

is shown in figure 15

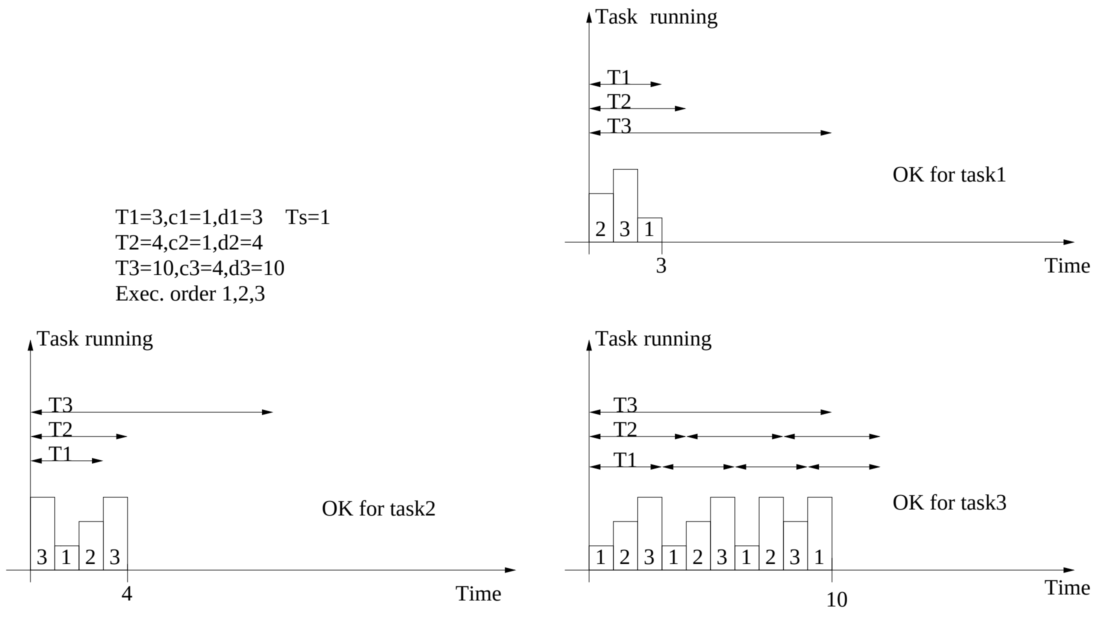

Figure 15: Analysis of taskset in preemptive round robin schedule.

In the above example the scheduler utilizes preemption which is acceptable as context switching is assumed insignificant. However one may easily imagine a non preemptive round robin scheduler. A rather simple criterion for schedulability of a hard real time task is found in such a case:

$$\sum_{i=1}^{N} c_i \leq d_j \tag{11}$$

since the worst case scenario is when the task in question needs to wait for completion of one job from all other tasks. In the above example τ_1 would in a worst case scenario have to wait $c_2 + c_3 = 5$ for the completion of jobs from τ_2 and τ_3 if executed in a non preemptive round robin schedule, which means that it would have missed its deadline even before starting to execute.

7 Summary

- The overall schedulability criterion is $U < 1$
- Schedulability depends on the scheduling policy.
- A number of different scheduling policies exist.

- Static schedules are efficient but less flexible.
- Static schedules may be constructed as cyclic schedules.
- Static schedules may be verified by inspection.
- Round robin is less efficient but highly flexible.
- Round robin may be verified inspecting worst case scenarios.
- Non preemptive version exist.