Sharif University of Technology
Electrical Engineering Complex

Real Time Embedded Systems

Research 2

Melika Rajabi
99101608

March 22, 2023

Introduction

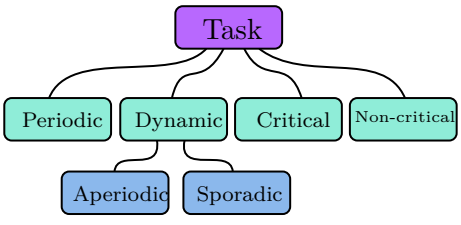
In this article we are going to discuss Scheduling algorithms. We will take into consideration, the type and number of tasks and jobs, the type and number of cores and whether we can minimize the number of cores without missing any job or not. In this regard, we are going to investigate timing diagrams, response times, sufficient and necessary conditions and so forth.

Task

Task is a real time computation unit. Each task consists of many jobs. It has some characteristics that together, build its profile. Some of these characteristics are:

- Timeliness parameters, e.g. arrival times, events rates, etc.
- Deadlines
- Resource utilization profiles
- Relative importance
- Worst case execution time
- Ready and suspension times
- Precedence and exclusion constraints

There are four types of task in real-time systems:



1 Periodic Task

In periodic tasks, jobs are released at regular intervals. It repeats itself after a fixed time interval.
A periodic task is denoted by:
$T_i =< \phi_i, p_i, e_i, D_i >$ Where:
ϕ_i: The phase of the task or the release time of the first job in the task. If the phase is not mentioned, then the release time of the first job is assumed to be zero.
p_i: The period of the task, i.e., the time interval between the release times of two consecutive jobs.
e_i: The execution time of the task. It should be calculated based on the worst case scenario.
D_i: The relative deadline of the task

2 Dynamic Task

It is a sequential program that is invoked by the occurrence of an event. An event may be generated by the processes external to the system or by processes internal to the system. Dynamically arriving tasks can be categorized on their criticality and knowledge about their occurrence times:

2.1 Aperiodic

In this type of task, jobs are released at arbitrary time intervals. Aperiodic tasks have soft deadlines or no deadlines.

2.2 Sporadic

Similar to aperiodic tasks, they repeat at random instances. The only difference is that sporadic tasks have hard deadlines.
Three tuples denote a sporadic task:
$T_i =< e_i, g_i, D_i >$ Where:
e_i: The execution time of the task
g_i: The minimum separation between the occurrence of two consecutive instances of the task.
D_i: The relative deadline of the task

3 Critical Task

Critical tasks are those whose timely executions are critical. If deadlines are missed, catastrophes occur.

4 Non-critical Task

Non-critical tasks are not critical to the application. However, they are real times tasks and hence they are useless if not completed within a deadline. The goal of scheduling these tasks is to maximize the percentage of jobs successfully executed within their deadlines.

Task Scheduling

Real-time task scheduling essentially refers to determining how the various tasks are the pick for execution by the operating system. Every operating system relies on one or more task schedulers to prepare the schedule of execution of various tasks needed to run. Each task scheduler is characterized by the scheduling algorithm it employs. A large number of algorithms for real-time scheduling tasks have so far been developed.

Before all else, we are going to be familiar with some concepts in scheduling:

Valid Schedule

A valid schedule for a set of tasks is one where at most one task is assigned to a processor at a time, no task is scheduled before its arrival time, and the precedence and resource constraints of all tasks are satisfied.

Feasible Schedule

A valid schedule is called a feasible schedule only if all tasks meet their respective time constraints in the schedule.

Proficient Scheduler

A task scheduler S_1 is more proficient than another scheduler S_2 if S_1 can feasibly schedule all task sets that S_2 can feasibly schedule, but not vice versa. S_1 can feasibly schedule all task sets that S_2 can, but there is at least one task set that S_2 cannot feasibly schedule, whereas S_1 can.

If S_1 can feasibly schedule all task sets that S_2 can feasibly schedule and vice versa, then S_1 and S_2 are called equally proficient schedulers.

Optimal Scheduler

A real-time task scheduler is called optimal if it can feasibly schedule any task set that any other scheduler can feasibly schedule. In other words, it would not be possible to find a more proficient scheduling algorithm than an optimal scheduler. If an optimal scheduler cannot schedule some task set, then no other scheduler should produce a feasible schedule for that task set.

Scheduling Points

The scheduling points of a scheduler are the points on a timeline at which the scheduler makes decisions regarding which task is to be run next. It is important to note that a task scheduler does not need to run continuously, and the operating system activates it only at the scheduling points to decide which task to run next. The scheduling points are defined as instants marked by interrupts generated by a periodic timer in a clock-driven scheduler. The occurrence of certain events determines the scheduling points in an event-driven scheduler.

Preemptive Scheduler

A preemptive scheduler is one that, when a higher priority task arrives, suspends any lower priority task that may be executing and takes up the higher priority task for execution. Thus, in a preemptive scheduler, it cannot be the case that a higher priority task is ready and waiting for execution, and the lower priority task is executing. A preempted lower priority task can resume its execution only when no higher priority task is ready.

Utilization

The processor utilization (or simply utilization) of a task is the average time for which it executes per unit time interval. In notations:
For a periodic task T_i, the utilization $u_i = \frac{e_i}{p_i}$, where:
e_i is the execution time
p_i is the period of T_i
For a set of periodic tasks $\{T_i\}$, the total utilization due to all tasks $U = \sum_{i=1}^{n} \frac{e_i}{p_i}$

Any good scheduling algorithm's objective is to feasibly schedule even those task sets with very high utilization, i.e., utilization approaching 1. Of course, on a uniprocessor, it is not possible to schedule task sets having utilization of more than 1.

Jitter

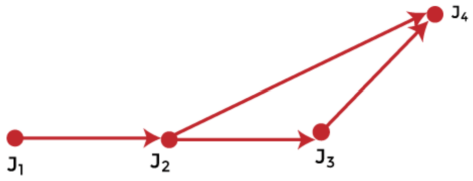
Jitter is the deviation of a periodic task from its strict periodic behavior. The arrival time jitter is the deviation of the task from the precise periodic time of arrival. It may be caused by imprecise clocks or other factors such

as network congestions.
Similarly, completion time jitter is the deviation of the completion of a task from precise periodic points. The completion time jitter may be caused by the specific scheduling algorithm employed, which takes up a task for scheduling as per convenience and the load at an instant, rather than scheduling at some strict time instants.
Jitters are undesirable for some applications. Sometimes actual release time of a job is not known. Only know that r_i is in a range $[r_i-, ri+]$. This range is known as release time jitter. Here r_i- is how early a job can be released and, r_i+ is how late a job can be released.
Similarly,only the range $[e_i-, e_i+]$ of the execution time of a job is known. Here e_i- is the minimum amount of time required by a job to complete its execution and, e_i+ is the maximum amount of time required by a job to complete its execution.

Precedence Constraint of Jobs

Jobs in a task are independent if they can be executed in any order. If there is a specific order in which jobs must be executed, then jobs are said to have precedence constraints. For representing precedence constraints of jobs, a partial order relation " $<$ " is used, and this is called precedence relation. A job J_i is a predecessor of job J_j if $J_i < J_j$, i.e., J_j cannot begin its execution until J_i completes.
J_i is an immediate predecessor of J_j if $J_i < J_j$, and there is no other job J_k such that $J_i < J_k < J_j$.
J_i and J_j are independent if neither $J_i < J_j$ nor $J_j < J_i$ is true.

An efficient way to represent precedence constraints is by using a directed graph $G = (J, <)$ where J is the set of jobs. This graph is known as the precedence graph. Vertices of the graph represent jobs, and precedence constraints are represented using directed edges. If there is a directed edge from J_i to J_j, it means that J_i is the immediate predecessor of J_j.
For example:



From the above graph, we derive the following precedence constraints:

- $J_1 < J_2$
- $J_2 < J_3$
- $J_2 < J_4$
- $J_3 < J_4$

Scheduling Algorithms

Based on schedulability, implementation (static or dynamic), and the result (self or dependent) of analysis, the scheduling algorithm are classified as follows:

Static table-driven approaches

These algorithms usually perform a static analysis associated with scheduling and capture the schedules that are advantageous. This helps in providing a schedule that can point out a task with which the execution must be started at run time.

Static priority-driven preemptive approaches

Similar to the first approach, these type of algorithms also uses static analysis of scheduling. The difference is that instead of selecting a particular schedule, it provides a useful way of assigning priorities among various tasks in preemptive scheduling.

Dynamic planning-based approaches

Here, the feasible schedules are identified dynamically (at run time). It carries a certain fixed time interval and a process is executed if and only if satisfies the time constraint.

Dynamic best effort approaches

These types of approaches consider deadlines instead of feasible schedules. Therefore the task is aborted if its deadline is reached. This approach is used widely is most of the real-time systems.

Rate-Monotonic Scheduling

An example for the first group (Fixed Priority) is the "Rate-Monotonic Scheduling". When working with RMS, one must take into account the following assumptions:

- Priorities are assigned in rank order of task period, so the highest priority is given to the task with the shortest period and the lowest priority is given to the task with the longest period.
- The requests for all tasks for which hard deadlines exist are periodic.
- Deadlines consist of runability constraints only, that is, each task must be completed before the end of its period.
- The tasks are independent. Each task does not depend on any other tasks.
- Task run time is constant (or at least has a known worst-case bound).
- Any non-periodic tasks are special (initialization or failure recovery, for example), and do not have hard critical deadlines.

Of course, these assumptions are often violated in real-world systems, i.e. some tasks can become aperiodic due to event-driven interrupts, and determining actual worst-case runtime can be quite difficult.

Advantages

- Easy to implement (most widely used)
- Low system overhead
- Optimal among other static priorities algorithms

Disadvantages

- Requires static prioritization before run-time

Sufficiency Condition: Liu & Layland
Although there isn't a straightforward way to determine for sure if a system is not RM schedulable, there is a sufficient but not necessary condition that guarantees that any system that meets this condition is RM schedulable. This check, takes into account the processor's total utilization and the number of tasks to be scheduled (n):

$$U_{task\ set} \leq n(2^{1/n} - 1)$$

First In First Out

FIFO is, in some ways, the simplest scheduling algorithm. As the name suggests, this policy is based on a queue where the first process that enters the queue is the first process that is executed. Each process runs to completion (and no process can be preempted) before the scheduler executes the next process in the queue. And because there is no prioritization in FIFO, systems using this policy often have trouble meeting deadlines, even in systems with fairly low total CPU utilization.

Sufficiency and necessity Condition
Among many limitations of FIFO, in general if any task has a period shorter than the total execution time of all tasks, then the tasks cannot be scheduled under FIFO.

Round Robin

RR scheduling is one of the most widely used static scheduling algorithms, especially in operating systems. Processes are placed in a queue. An arbitrary interval of time is defined and the scheduler cycles through the queue, running each process for the length of that time interval until it is preempted by the next process in the queue. The scheduler keeps cycling through the queue until all processes have finished executing.
Because of its lack of prioritization, RR can also fail to successfully schedule a low-CPU utilization harddeadline system.

Earliest Deadline First

EDF is an optimal single-processor scheduling algorithm. It is a dynamic, priority-driven algorithm in which highest priority is assigned to the request that has the earliest deadline, and a higher priority request always preempts a lower priority one.
Since EDF is a dynamic scheduling policy, this means priority of a task is assigned as the task arrives (rather than determined before the system begins execution) and is updated as needed. This can be a very difficult thing to implement as the scheduler must track all deadlines and the amount of CPU time used by each process thus far.

Sufficiency and necessity Condition

$$U_{task\ set} \leq 1$$

Least Laxity First (Least Slack Time First)

The laxity (also referred to as "slack time") of a process is defined as the maximum amount of time a task can wait and still meet its deadline. It is formally defined as:

$$\text{Laxity} = (d - t) - c'$$

Where d is the deadline of the task, t is the real time since the task first wished to start, and c' is the remaining execution time.
In LLF (LST) scheduling, highest priority is given to the task with the smallest laxity.
Like EDF, LLF can be used for processor utilization up to 100%. This is also an optimal scheduling algorithm for periodic real-time tasks.

When a task is executing, it can be preempted by another task whose laxity has decreased to be below that of the active task. However, a problem can arise with this scheme in which two tasks have similar laxities. One process will run for a short while and then be preempted by the other task, then priority will which back, and so on. This causes many context switches to occur during the lifetime of these tasks.

Comparing

Now that we examined several of the most popular RTOS scheduling algorithms, we will discuss the advantages and disadvantages between different algorithms.

FIFO is the simplest scheduler in that all it requires is a very basic queuing algorithm. It requires no input from the user (such as assigning priorities, an estimation of run time, or period of the tasks) as everything is handled by the RTOS. However, due to FIFO's lack of prioritization, it can fail with a schedule that has a fairly low total CPU utilization. Thus, although FIFO is an easy algorithm to understand, its limited capabilities make it ill-suited for many real-world applications.

RR, like FIFO, also has the benefit of being simple to understand and being nearly as simple to implement. It has the advantage of being fair in that all tasks will get an equal share of the CPU and no one task can hog the processor. This fairness is extremely useful on a typical multi-user system (such as a Linux server) but much less relevant for a real-time OS. Its usefulness in a general user system means RR (or some close variant) is a commonly available option.

RMS's fixed prioritization and relative ease of use makes it well-suited for use in many applications. The sufficiency condition allows users to quickly determine the schedulability of a group of tasks (although more checks will be needed if the sufficiency condition fails). However, it is not an optimal scheduling algorithm so there are cases where the total CPU utilization is less than 100% yet it won't find a schedulable solution. In addition, if the tasks aren't periodic, RMS won't be an available option.

Unlike the above schemes, EDF can schedule any set of tasks that don't exceed 100% CPU utilization. However, this comes at the cost of a more complex scheduling algorithm.

On the other hand, LLF can often handle unschedulable tasks better than EDF. Once it is clear that a process can't be completed (slack goes negative) it can stop executing the task. This can be useful if your deadlines are soft rather than hard.

There are many other scheduling algorithms, each has some advantages and some disadvantages. Each of them can be suitable for an especial purpose and aplication.

Another subject that we have to take into account is real-world difficulties. As we were analysing each algorithm, we made some assumptions that are not often the case. Now one by one we try to omit those assumptions and try to face the problems occuring and solve them.

Real-World Difficulties

Different Assignments' Deadlines

Previously, we assumed that the deadline for each task is its period. However, the deadline can be sometime before the next period arrives. In this case, EDF is still optimal. But instead of RMS, DMS (Deadline Monotonic Scheduling) is used. In this algorithm, we replace periods by deadlines everywhere in the RMS's formulas.

Resource Sharing and Priority Inversion

In a preemptive priority based real-time system, sometimes tasks may need to access resources that cannot be shared. The method of ensuring exclusive access is to guard the critical sections with binary semaphores. When a task seeks to enter a critical section, it checks if the corresponding semaphore is locked. If it is not, the task locks the semaphore and enters the critical section. When a task exits the critical section, it unlocks the corresponding semaphore.
Priority inversion occurs when a higher priority task is forced to wait for the release of a semaphore that is owned by a lower priority task. The higher priority task cannot run until the lower priority task releases its semaphore, which usually occurs when the lower priority task has run to completion.
To deal with this issue, we can use some protocols:

Priority Inheritence Protocol

It works for fixed-priority scheduling such as RMS. While T_i blocks higher priority task T_j, T_i inherits the priority of T_j. Inheritance is transitive. If T_k blocks T_i, T_k inherits the priority of T_j through T_i. This method avoids both unbounded priority inversion and deadlock (the latter only if there are no nested resources).

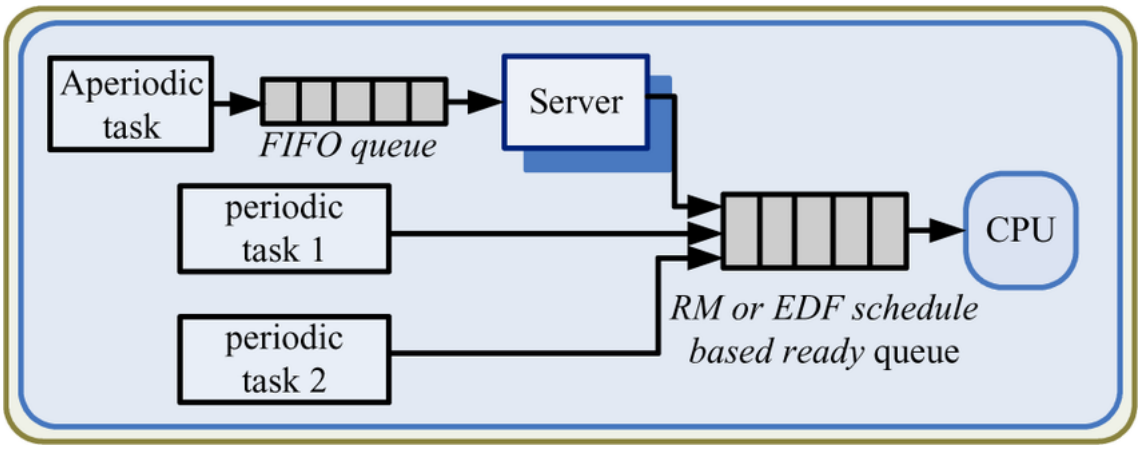
Priority Ceiling Protocol

In this protocol, each task is assigned a static priority, and each resource is assigned a ceiling priority greater than or equal to the maximum priority of all the tasks that use it. During run time, a task assumes a priority equal to the static priority or the ceiling value of its resource (whichever is larger). If a task requires a resource,

the priority of the task will be raised to the ceiling priority of the resource; when the task releases the resource, the priority is reset.

Aperiodic Tasks

As you recall, while analysing algorithms, we focused on periodic tasks. But in real world, there may occur some aperiodic tasks as well. Therefore, we have to know how to deal with other types of tasks either.
The main protocol for dealing with aperiodics is to insert them into a queue (server) so as the scheduler picks among periodic tasks or the aperiodic server. The server itself may use FIFO or another algorithm for prioritizing the entering to the main scheduler.



Picking from aperiodic server can be done in two ways:

Background server Execute aperiodics whenever the CPU is not running a periodic task. (i.e., The server has lowest priority)
Problem: response time can be very high.

Budget-based server Server is assigned budget Q_i and period p_i. The server behaves like a periodic task with $e_i = Q_i$ and period p_i. When the scheduler picks the server, if there is budget left, the server executes an aperiodic in the queue consuming its budget. When budget $= 0$, server waits until the next period, then replenish the budget to Q_i.
Problem: If the scheduler picks the server and there are no queued aperiodic tasks "Dumb" servers (Polling Server) lose budget. "Smart" servers (ex: Sporadic Server) keep the budget but modify their activation (recharge) time.

Sporadic Tasks

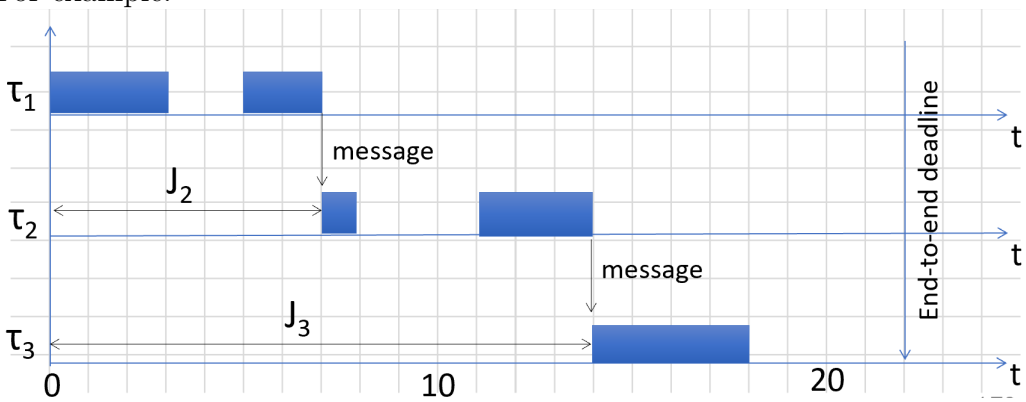
To deal with this type of tasks, we introduce two approaches:

Approach 1 Define fictitious periodic task of highest priority and of some chosen execution period. During the time this task is scheduled to run, the processor can run any sporadic task that is awaiting service. If no sporadic task awaiting service, processor is idle. Outside this time the processor attends to periodic tasks.
Problem: Wasteful

Approach 2 Whenever the processor is scheduled to run sporadic tasks, and finds no such tasks awaiting service, it starts executing other periodic tasks in order of priority. However, if a sporadic task arrives, it preempts the periodic task and can occupy a total time up to the time allotted for sporadic task.

End-to-End Delay

So far we have considered periodically-activated tasks. However, oftentimes tasks depend on communication messages. They activate once another task send a message to them. To schedule this kind of tasks, we use "Activation Jitter". We define a jitter for these tasks, which equals the worst-case response time of the activator tasks.
For example:



J_2 = worst-case response time of τ_1
J_3 = worst-case response time of τ_2

Multiprocessor Scheduling

Task scheduling on multiple processors is a far more difficult task than scheduling on a single processor. Hence, we are going to investigate two solutions for doing so:

Partitioning

Statically assign tasks among M processors.

Global Scheduling

Keep a global scheduling queue. Whenever there is a free core, pick one task from the queue and schedule it on the core.

Partitioned scheduling is preferred because there are three issues with global scheduling:

- Increases unpredictability, tasks can migrate among cores
- Much more complex to implement
- Does not necessarily perform better than partitioning
- Does not take into account the cost of preemption and migration

Conclusion

To make a long story short, an appropriate scheduler for a certain application, has to be able to handle different types and numbers of tasks present in the application. Besides it has to be practical in spite of presence of real-world difficulties. In addition, it should be practical on multiprocessor structures, if required. And in the end, it should perform all these tasks in an optimal way.

References

[1] Embedded real-time systems' class Slides, Part one, Page 51

[2] Embedded real-time systems' class Slides, Part Three

[3] javatpoint.com

[4] geeksforgeeks.org

[5] eecs.umich.edu

[6] just.edu.jo