Czech Technical University in Prague
Faculty of Electrical Engineering

ČVUT FEL katedra počítačů

Diploma Thesis

The EDF scheduler implementation in RTEMS Operating System

Martin Molnár

Supervisor: Ing. Libor Waszniowski

Study program: Computer Science and Engineering master

May 2006

Poděkování

Mé poděkování patří Ing. Liboru Waszniowskému za jeho obětavou pomoc při tvorbě diplomové práce.

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 10.5. 2005 ....

Abstract

Many real-time operating systems like RTEMS schedules tasks according to their priorities. However, the theoretical results expose that EDF scheduling brings better performance. It gave me impetus to implement EDF scheduler in RTEMS.

In fact, EDF ready queue is more difficult to implement than ready queue with fixed priority levels since ready tasks have to be order by ascending absolute deadline that changes for each instance of task and can take th evalue from wide range. I proposed and implementated the EDF ready queue as Red-Black tree with tasks being directly nodes of the tree. This structure has the linear space complexity. Time complexity of operations handling the queue is $O(log_2 n)$ in the worst case.

EDF scheduling requests for special resource access protocol. In this document I describe implementation difficulties of such protocols and suggest the solution.

Abstrakt

Mnohé operační systémy reálneho času jako RTEMS rozvrhují úlohy podle jejich priorit. Teoritické výsledky ovšem ukazují, že EDF rozvrhování přináší lepší výkonnost. To mi dalo podnět na implementaci EDF rozvrhovače pod RTEMS.

Ve skutečnosti je těžší implementovat EDF frontu připravených úloh než-li frontu s pevnými úrovněmi priorit, protože připravené úkoly musí být uspořádany podle vzestupného absolutního termínu, který se mění pro každou instanci úkolu a může nabývat hodnotu ze širokého rozsahu. Navrhnul jsem a implementoval EDF frontu jako Red-Black strom s úkoly jako listy stromu. Taková struktura má linearní prostorovou složitost. Časová složitost operací spravujících frontu je $O(log_2 n)$ pro nejhorší případ.

EDF plánování vyžaduje speciální protokol pro přístup k zdrojům. V tomto dokumentu popisuju obtížnosti implementace takových protokolů a naznačuji řešení.

Contents

List of Figures

1 Introduction

Real-time systems are computing sytems that must accept and consequently process the events coming from the environment within precise time constraints. For example, a patient care system that monitors and controls the patient's blood pressure and respiration must alarm the doctor within precise time constraint when an emergency occurs.

The software implementation of real-time system consists of a real-time operating system (RTOS) and an application software. RTOS provides an "abstraction level" that hides the hardware details from the application software. An application calls the services (CreateTask(), SemaphoreObtain(), etc.) provided by RTOS. The real-time application usually consists of autonomous processes or tasks responding to external stimuli. The system can asynchronously switch between them, directly responding to external events as they occur. This allows the system design to meet critical performance specifications.

General-computing operating systems (Linux, Windows) and real-time operating systems have common properties such as reliability and efficiency. The key difference between them is a deterministic timing behaviour in the real-time operating systems. It means that every task running in such a system must be finished in a certain time. There have been many RTOS developed, recently the most used ones to build real-time systems are VxWorks, RTEMS, QNX and OSEK.

Scheduling is a process of designing an execution order of a set of tasks with certain known characteristics on a limited set of resources like the CPU, memory, peripherals, etc. Periodic task scheduling is one of the most studied topics within the field of real-time systems, due to the large number of control applications that require cyclical activities. For that reason and for simple analysis I focuse on this class of tasks. The chapter 2 introduces other basic real-time concepts that are needed to be understand.

Several scheduling algorithms have been proposed and analysed during the past two decades. Since Rate Monotonic (RM) scheduling algorithm can be simply developed and its time complexity is $O(1)$, it is being implemented in almost all RTOSs. The RM algorithm is a simple rule that assigns fixed priorities to tasks according to their request rates. Then the task with the highest priority (the highest request rate) is execute as the first.

In [1] Giorgio C. Buttazzo compares Earliest Deadline First (EDF) to RM scheduling and refutes presumptions that RM scheduling is easier to analyse than EDF, it introduces less runtime overhead, it is more predictable in overload conditions, and causes less jitter in task execution. On the contrary, due to its properties, the EDF provides better performance overall. The resuls of Buttazzo's work, summarized in chapter 3, motivated me to implement EDF scheduling into RTEMS which is an open source real-time system with RM scheduling.

RTEMS is a well-designed, open source real-time operating system which lacks of EDF scheduler. In chapter 5, I focus on RTEMS, its feature, structure, partly RTEMS API. The most of these and the information in the appendix can not be found in official RTEMS documentation and are important to understand EDF scheduler implementation or building applications.

The development of EDF scheduler mainly relates to ready queue. In fact, EDF ready queue is more difficult to implement than ready queue with fixed priority levels since ready tasks have to be order by ascending absolute deadline that changes for each instance of task and can take the value from wide range. Several approaches I took into account and chose the most effective one. I proposed and implementated the EDF ready queue as Red-Black tree with tasks being directly nodes of the tree. This structure has linear space complexity. Time complexity of operations handling the queue is $O(log_2n)$ in the worst case. The EDF scheduler implementation is described in chapter 6.

EDF scheduling requests for special resource access protocol(RAP). Resource Access protocols are generally described in chapter 4. In chapter 7, I explain the implementation of existing RAPs in RTEMS core. Futher, I analyse RAPs supporting EDF scheduling that can be developed for RTEMS, focusing on their implementation difficulties and proposing the solution.

2 Basic Concepts

Real-time systems are computing sytems that must accept and subsequently process events coming from the environment within a precise time constraint. RTOS supports the software implementation of a real-time system. It provides an "abstraction level" that hides the hardware details from application software. An application is executed on platform provided by RTOS and does not need to access hardware directly. RTOSs emphasize the predictability, efficiency and satisfaction of timing constraints that makes them different from general-computing operating systems(Linux, Windows). The RTOS market includes many proprietary kernels(VxWorks, QNX), composite kernels (RTEMS) and real-time version of popular OS (Linux, Windows-NT).

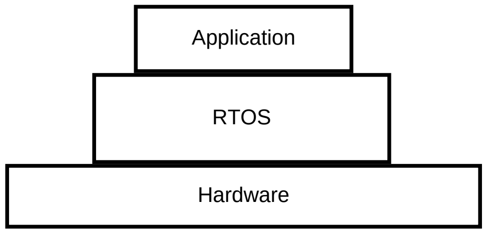


Figure 2.1: Layers of real-time system

A computation executed by the CPU in sequential fashion running is called *task* in the RTOS environment. Since only one task can run at the moment in a single processor system, there must be a predefined criterion called *scheduling policy* according to which the CPU is assigned to a task from a set of tasks. The set of rules that determine the execution order of tasks is called a *scheduling algorithm.*[2]

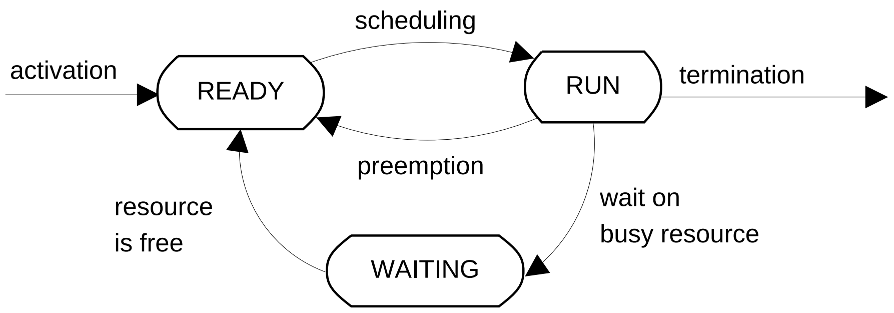


Figure 2.2: The life cycle of task in RTOS

The figure 2.2 schematically illustrates the life cycle of task in RTOS. After RTOS registers a task and assigns a memory area to it, the task is created or activated and RTOS's scheduler inserts the task into relevant position in the ready queue [1]. The task is now in the *ready* state waiting for the processor. After the processor is free, it is always assigned to the first task in ready queue which starts the running.

[1]Operating systems collects all information about the task in Task Control Block(TCB) data structure, where TCB represents a task in the system. Therefore, when talking about inserting a task into ready queue, it means inserting its TCB.

In many RTOSs a more important task that has arrived to interrupt the running task at any point,is allowed to gain the processor immediately and not to wait in the ready queue. The interrupted running task is inserted into the ready queue. The operation of taking processor from the running task, which is consequently inserted into the ready queue, is called *preemption.* With preemption is associated *context switch* that means the switching from a former to a present running task.

The processor is not the only resource a task compete for. Typically, a resource can be a file, a data structure, register of a device. Resource that can be used by more than one task is called *shared resources.* Shared resource that does not allow simultaneous access of tasks is called *exclusive resource* and a piece of task code accessing this resource exclusively is *critical section.*

RTOS provides a synchronization mechanism that guards a critical section of code. To control the access to an exclusive resource a semaphore called mutex is used. It has two operational states - locked and unlocked. Before entering the critical section the task tries to lock the appropriate mutex. In case of success it enters the critical section otherwise becomes blocked on that resource. All tasks blocked at the same resource are kept in the waiting queue associated with mutex protecting that resource. When the task leaves the critical section it unlocks the appropriate hold mutex. In consequence the first queued task is released from waiting queue of mutex and goes to a ready state. The operation locking or unlocking mutex means in the simpliest case to set a flag of mutex adequately but in order to avoid undesirable situations(deadlock, priority inversion) it has to be claused by rules defined by resource access protocol explained later.

Real-time task T_i can be characterized by following parameters[2]:

- **Arrival time** a_i: is the time at which a task becomes ready for execution; it is also referred as *request time* or *release time* and indicated by r_i
- **Computation time** C_i: is the time necessary for the processor to execute the task without interruption;
- **Deadline** d_i: is the time before which a task should be completed to avoid the damage to the system;
- **Start time** s_i: is the time at which a task starts its execution;
- **Finishing time** f_i: is the time at which a task finishes its execution

Defined parameters are illustrated in Figure 2.3.

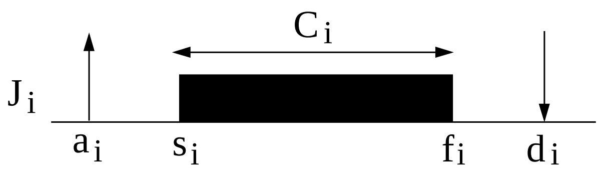


Figure 2.3: Real-time task's parameters

According to the regularity of activation, the real-time task can be *periodic* and *aperiodic*. A periodic task consists of an infinite sequence of activities, called *instances* or *jobs*, that are regularly activated at a constant rate - *period*. Next, for sake of simplicity, only periodic tasks will be considered, analyzed and their deadlines will equal to their periods. A periodic task will be denoted by τ_i and a job by $J_{i,j}$ where i index associates the task with its job and j is the job ordinal number in the sequence of activated jobs.[2]

A *schedule* is an execution order number assignment to each task from set of tasks $\tau = \tau_1, \ldots, \tau_n$. A schedule is said to be *feasible* if all tasks can be completed according to specified constraints. A set of tasks is said to be *scheduable* if there exists at least one scheduling algorithm that can produce a feasible schedule.[2]

Several scheduling algorithms have been proposed, among which the following main classes can be distinguished [2]:

- **Preemptive.** With preemptive algorithms, the running task can be interrupted at any time to assign the processor to another active task, according to a predefined scheduling policy.
- **Non-preemptive.** With a non-preemptive algorithm, a task once started, is executed by the processor until its completion. In this case all scheduling decisions are taken as a task terminates its execution.
- **Static.** Static algorithms are those in which scheduling decisions are based on fixed parameters (like task's priority in case of RM scheduling), assigned to tasks before their activation.
- **Dynamic.** Dynamic algorithms are those in which scheduling decisions are based on dynamic parameters (like task's absolute deadline in case of EDF scheduling) that may change during system evolution.
- **Off-line.** We say that a scheduling algorithm is used off-line if it is executed on the entire task set before actual task activation. The schedule generated in this way is stored in a table and later executed by a dispatcher.
- **On-line.** We say that a scheduling algorithm is used on-line if scheduling decisions are taken at runtime every time a new task enters the system or when a running task terminates.
- **Optimal.** An algorithm is said to be optimal if it minimizes some given cost function defined over the task set. When no cost function is defined and the only concern is to achieve a feasible schedule, then an algorithm is said to be optimal if it may fail to meet a deadline only if no other algorithms of the same class can meet it.
- **Heuristic.** An algorithm is said to be heuristic if it tends toward but does not guarantee to find the optimal schedule.

3 Rate Monotonic vs. EDF

In [1] G.C.Buttazzo compares EDF and RM scheduling. His results and conclusions are worth mentioning and gave me a impetus to my work. Therefore this chapter, except the Introduction section, is an abstract of Buttazzo's article.

3.1 Introduction

The Rate Monotonic scheduling (RMS) algorithm is a simple rule that assigns priorities to tasks according to their request rates. Specifically, task with higher request rate (that is, with shorter period) will have higher priority. Since periods are constant, RMS is a fixed-priority assignment and is intrinsically preemptive. [2]

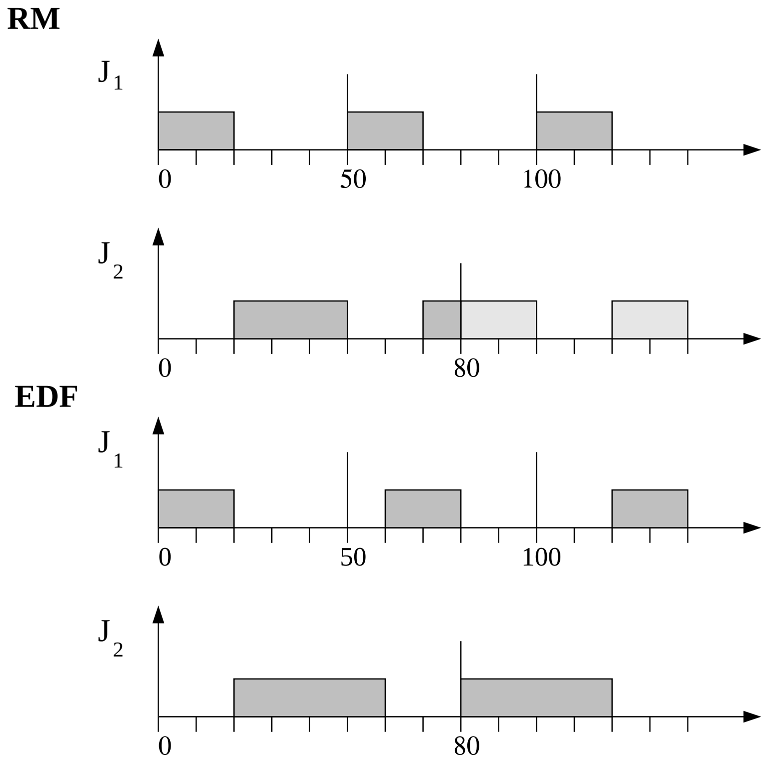


Figure 3.1: An example of RM and EDF scheduling of jobs J_1 and J_2.

Figure 3.1 illustrates RM scheduling of jobs J_1 with a 50 miliseconds period, 20 miliseconds computation time and J_2 with 80 milisecond period, 40 miliseconds computation time. Period of the job J_2 is shorter, therefore it is assigned a higher priority than to job J_1. At time 0 job J_1 starts to execute and subsequently job J_2 after its termination. At time 50 the executing J_2 is preempted by arrived higher priority task J_1. When J_1 is finished, job J_2 is resumed.

The important parameter for basic scheduability analysis of periodic tasks is *processor utilization factor U* - the fraction of processor time used by the periodic task set. The ratio $U_i = C_i/\tau_i$ is the utilization factor of task τ_i and represents the fraction of processor time used by that task. Then the utilization factor U for n periodic tasks is given by [2]:

$$U = \sum_{i=1}^{n} U_i = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{3.1}$$

For an arbitrary set of n periodic tasks, it is guaranteed that a schedule is feasible with RM algorithm if the following condition is satisfied [2]:

$$U < U_{luf} \tag{3.2}$$

,where U_{luf} is the least upper bound of the processor utilization factor and can be calculated as follows:

$$U_{luf} = n(2^{1/n} - 1) \tag{3.3}$$

For high value of n, the least upper bound converges to the value $ln2$:

$$U_{luf} = ln2 \simeq 0.69 \tag{3.4}$$

The Earliest Deadline First scheduling algorithm is a dynamic scheduling algorithm because scheduling decisions are based on absolute deadline of task changing dynamically (each job of task has different absolute deadline). The preemption is assumed - currently executing task is preempted whenever another periodic job with ealier deadline becomes active. If $J_{i,k}$ is k-th instance of task τ_i with relative deadline (period) D_i then the absolute deadline $d_{i,k}$ of job $J_{i,k}$ is

$$d_{i,k} = r_{i,k} + D_i \tag{3.5}$$

,where $r_{i,k}$ is release time of job $J_{i,k}$. [2]

Figure 3.1 illustrates EDF scheduling of tasks with the same parameters like in the previous example of RMS. Because τ_1 has earlier absolute deadline (50) that τ_2 (80), it is executed as the first. When τ_2 is assigned the processor, it is not preempted by arrived task τ_1 at time 50 since τ_2's absolute deadline is earlier.

A set of periodic tasks is scheduable with EDF if and only if processor utilization factor $U \leq 1$. Therefore, tasks may utilize the processor up to 100% and still be scheduable. Consider the previous example illustrated in Figure 3.1 when J_2's computation time is raised from 40 to 50 miliseconds. In consequence the total processor utilization factor U becomes higher. Although the schedule of jobs is feasbible with EDF scheduling, it is not with RM scheduling because of J_2's deadline miss that would occur at time 80.

There are a lot of misconceptions about the properties of these two scheduling algorithms, that for a number of reasons unfairly penalize EDF. The typical motivations that are usually given in favor of RM state that RM is easier to implement, it introduces less runtime overhead, it is easier to analyze, it is more predictable in overload conditions, and causes less jitter in task execution. In the following sections it is shown that most of the claims stated above are either false or do not hold in the general case. These two algorithms is analyzed under several perspectives, including implementation complexity, runtime over-head, schedulability analysis, robustness during transient and permanent overloads, and response time jitter. Moreover, the two algorithms are also compared with respect to other issues, including resource sharing, aperiodic task handling.[2]

3.2 Implementation complexity

The implementation details of EDF and RM scheduling can be found in chapter 6 and 5 respectively.

In fact, the EDF ready queue is more difficult to implement than ready queue with fixed priority levels since ready tasks have to be order by ascending absolute deadline that changes for each instance of task and take the value from wide range. The inserting a new task into the EDF ready queue ,implemented as balanced binary tree, is more time-consuming than inseting priority task into the ready queue, implemented as an array of priority levels. However, determining the next running task i.e. the first task is more difficult in the priority ready queue.

3.3 Runtime overhead

It is commonly believed that EDF introduces a larger runtime overhead than RM, because in EDF absolute deadlines need to be updated from a job to the other, so slightly increasing the time needed to execute the job activation primitive, whereas such a computation is not needed under RM, because the priority of task τ_i is assigned based on its period T_i.

In spite of the extra computation needed for updating the absolute deadline, however, EDF introduces less runtime overhead than RM, when context switches are taken into account. In fact, to enforce the fixed priority order given by periods, the number of preemptions that typically occur under RM is much higher than under EDF.

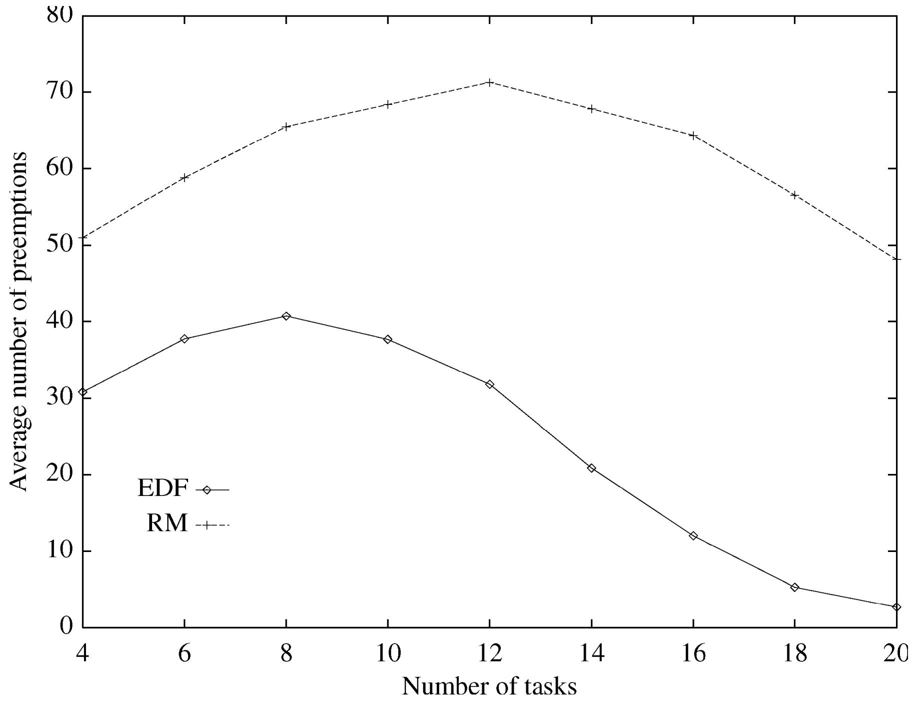


Figure 3.2: Preemptions introduced by RM and EDF as a function of the number of tasks.

Figure 3.2 shows the average number of preemptions introduced by RM and EDF as a function of the number of tasks. For each point in the graph, the average was computed over 1000 independent simulations, each running for 1000 units of time. In each simulation, periods were generated as random variables with uniform distribution in the range of 10 to 100 units of time, whereas execution times were computed to create a total processor utilization U = 0.9.

For small task sets, the number of preemptions increases because the chances for a task to be preempted increase with the number of tasks in the system. As the number of tasks gets higher, however, task execution times get smaller in the average, to keep the total processor utilization constant, hence the chances for a task to be preempted reduce. As evident from the graph, such a reduction is much more significant under EDF.

In another experiment, the behavior of RM and EDF is tested as a function of the processor load, for a fixed number of tasks. Figure 3.2 shows the average number of preemptions as a function of the load for a set of 10 periodic tasks. Periods and computation times were generated with the same criterion used in the previous experiment, but to create an average load ranging from 0.5 to 0.95.

It is interesting to observe the different behavior of RM and EDF for high processor loads. Under RM, the number of preemptions constantly increases with the load, because tasks with longer execution times have more chances to be preempted by tasks with higher priorities. Under EDF, however, increasing task execution times does not always imply a higher number of preemptions, because a task with a long period could have an absolute deadline shorter than that of a task with smaller period. In certain situations, an increased execution time can also cause a lower number of preemptions.

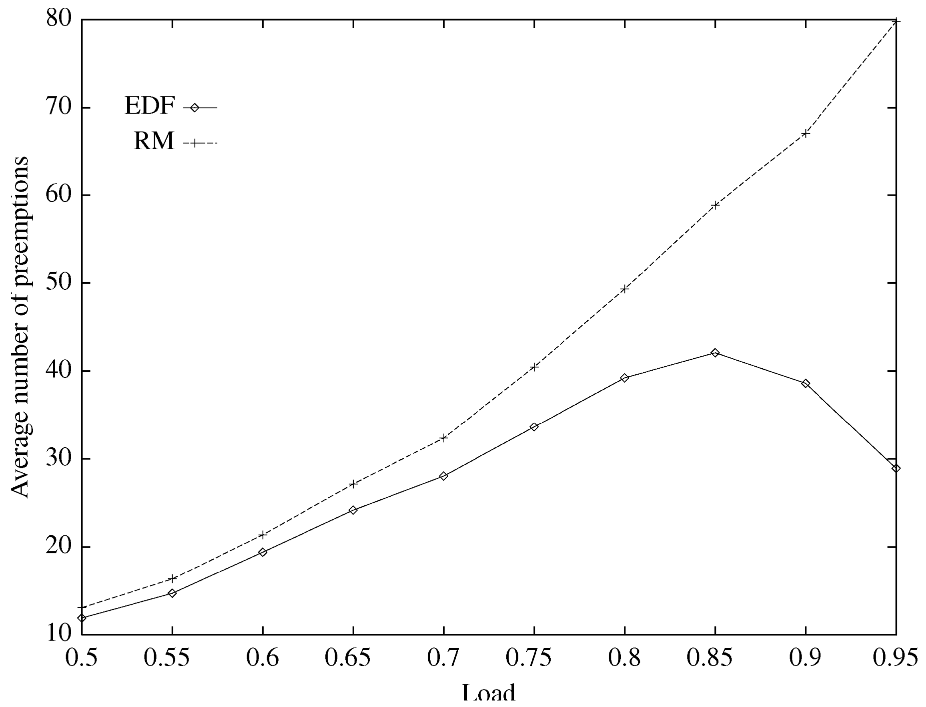


Figure 3.3: Preemptions introduced by RM and EDF on a set of 10 periodic tasks as a function of the load.

3.4 Schedulability Analysis

The schedulability analysis of periodic tasks with relative deadlines the same as period was described for EDF and RM briefly in the introduction of this chapter. However, such analysis is not exact and not always sufficient.

In the general case, exact schedulability test of periodic tasks with deadlines less than or equal to periods for RM can be peformed using the Response Time Analysis (RTA) for RM and

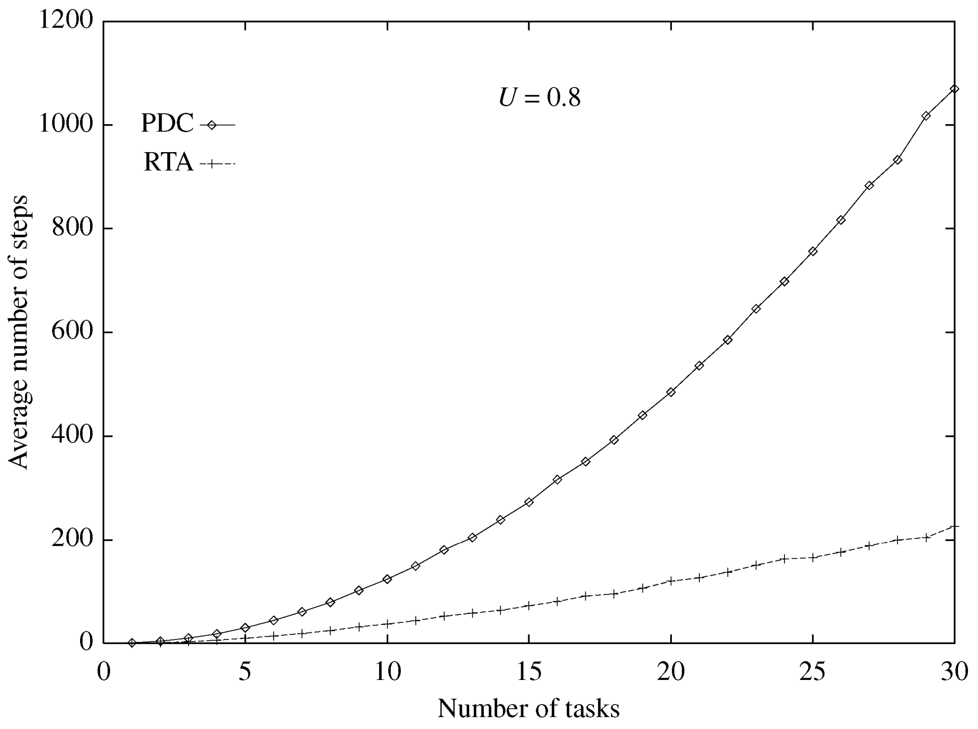


Figure 3.4: Average number of steps required for the RTA and for the PDC as a function of the number of tasks.

Processor Demand Criterion (PDC) for EDF. If relative deadlines are equal to periods, exact schedulabilty analysis can be performed in $O(n)$ under EDF, whereas is pseudo-polynomial under RM. When relative deadlines are less than periods, the analysis is pseudo-polynomial for both scheduling algorithms, although, in the average, the PDC requires more computational steps. Figure 3.4 shows the average number of steps required to run the RTA and the PDC as a function of the number of tasks. The tests were performed on randomly generated task sets with periods T_i uniformly distributed in [10, 200], relative deadlines uniformly distributed in [T_i /2, T_i], and utilization factor U = 0.8. The average was computed on 1000 samples.

3.5 Robustness During Overloads

In this section, the behavior of RM and EDF is compared during overload conditions , that is when the total demand of the task set exceeds the processor capacity. The two scheduling algorithms are considered under permanent overload situations (occurring when $U > 1$), and then under transient overload conditions, caused by sporadic execution overruns in some of the jobs.

3.5.1 Permanent Overload

Under permanent overload conditions both the behaviors of RM and EDF are predictable, but, deciding which one is better is highly application dependent. EDF automatically performs a period rescaling, and tasks start behaving as they were executing at a lower rate, whereas RM may cause a complete blocking of the lower priority tasks.

3.5.2 Transient Overload

Under RM, if the system becomes overloaded, any task, except the highest priority task, can miss its deadline, independently of its period. The situation is not better under EDF. The only difference between RM and EDF is that, under RM, an overrun in task can not cause tasks with higher priority to miss their deadlines, whereas under EDF any task could miss its deadline. However, such a property of RM can be of little use if we do not know a priori which task is going to overrun.

3.6 Jitter and Latency

In a feasible periodic task system, the computation performed by each job must start after its release time and must complete within its deadline. Due to the presence of other concurrent tasks that compete for the processor, however, a task may evolve in different ways from instance to instance; that is, the instructions that compose a job can be executed at different times, relative to the release time, within different jobs. The maximum time variation (relative to the release time) in the occurrence of a particular event in different instances of a task defines the jitter for that event. The relative response time jitter (RRJ) of a task is the maximum time variation between the response times of any two consecutive jobs. If $r_{i,k}$ denotes the response time of the k-th job of task τ_i , then the relative response time jitter (RRJ$_i$) of task τ_i is defined as

$$RRJ_i = |r_{i,k+1} - r_{i,k}| \tag{3.6}$$

whereas the absolute response time jitter (ARJ$_i$) of task τ_i is defined as

$$ARJ_i = max\ r_{i,k} - min\ r_{i,k} \tag{3.7}$$

In many control applications a high jitter can cause instability or a jerky behavior of the controlled system, hence it must be kept as low as possible.

One misconception about RM is to believe that the fixed priority assignment used in RM reduces the jitter during task execution, more than under EDF. Generally, it is not true that RM always outperforms EDF in reducing jitter nor vice-versa. A specific simulation experiment [1] has been performed to verify the jitter behavior under the two scheduling algorithms. Task sets of 10 periodic tasks were randomly generated with periods uniformly distributed in [10,200] and fixed total utilization U . The results refer to the ARJ, which has been normalized with respect to task periods. The results are reported in Figures 3.5 and 3.6, which show the jitter introduced by the algorithms for each individual task (tasks are ordered by increasing periods). It is worth observing that, when the periodic load is less than 0.7, both algorithms introduce about the same jitter, which linearly increases for tasks with longer periods. For higher loads, while RM reduces the jitter of high priority tasks at the expenses of tasks with lower priority, EDF treats tasks more evenly, obtaining a significant reduction in the jitter of the tasks with long periods for a small increase in the jitter of tasks with shorter periods.

Another parameter that it is important to minimize in control applications is the input-output latency. Assuming that a control task τ_i acquires inputs at the beginning of each instance and delivers control outputs at the end, the maximum input-output latency is defined as

$$L_i = max(f_{i,k} - s_{i,k}) \tag{3.8}$$

where $s_{i,k}$ and $f_{i,k}$ are the start time and finishing time of job $J_{i,k}$ respectively.

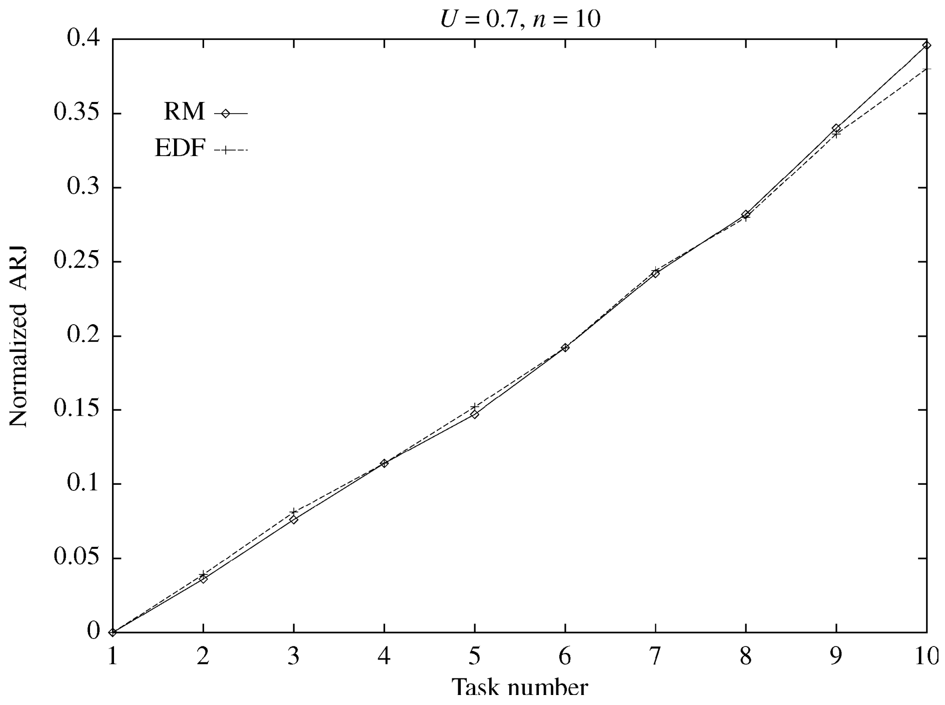


Figure 3.5: Average normalized jitter for U = 0.7.

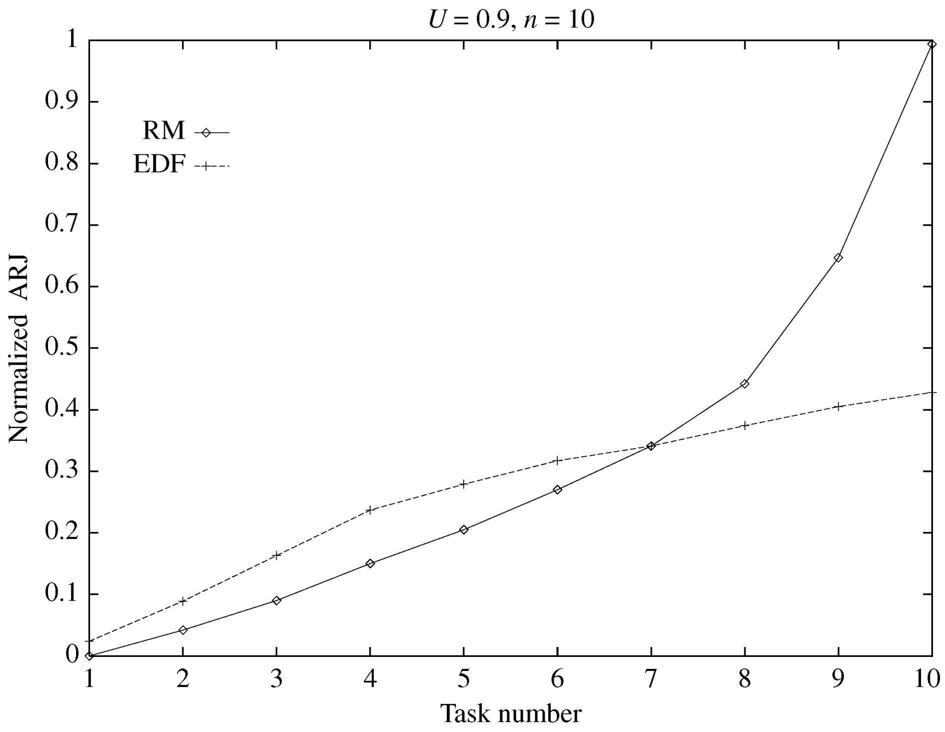


Figure 3.6: Average normalized jitter for U = 0.9.

Cervin [3] proved that EDF can always achieve a shorter input-output latency than RM, for any task. This is stated by the following theorem.

Theorem 3.6.1 . *Given a set of n periodic control tasks performing input at the beginning of each job and output at the end, the maximum inputoutput latency of each task under EDF is shorter than or equal to the corresponding maximum latency under RM.*

3.7 Resource sharing

The problem of resource sharing is discussed in a detail in chapter 4. Resource sharing is resolved by adopting specific concurrency control protocols for accessing critical sections, such as the Priority Inheritance Protocol (PIP) or the Priority Ceiling Protocol(PCP). For the reason that both PIP and PCP are well known in the real-time literature and were originally deviced for RM, some people believe that only RM can be predictably used and analyzed in the presence of shared resources. However, this is not true, because a number of protocols also exist for accessing shared resources under EDF and are described in chapter 4.

3.8 Aperiodic task handling, resource reservation

EDF allows a full processor utilization, which implies a more efficient exploitation of computational resources and minimizes the response times of the aperiodic tasks. Therefore, most resource reservation algorithms which are implemented using service mechanisms similar to aperiodic servers, have better performance under EDF.

4 Resource Access Protocols

4.1 Introduction

Resource (file, data structure, etc. other than CPU) that can be used by more than one task is called *shared resources.* Shared resource that does not allow simultaneos access of tasks is called *exclusive resource* and a piece of task's code accessing this resource exclusively *critical section.*

RTOS typically provides a general task synchronization tool, called a *semaphore*, that can be used for guarding access to a critical section. However, futher mentioned problems with priority inversion and deadlock must be solved. The access to exclusive resource is controlled by a semaphore called *mutex.* It has two operational states - locked and unlocked. Before entering a critical section, task tries to lock (obtain) an appropriate binary semaphore. In case of success it enters the critical section, otherwise becomes blocked on that resource or blocked by the task holding the resource. All tasks blocked on the same resource are kept in a waiting queue associated with mutex protecting that resource. When the task leaves a critical section it unlocks (releases) the appropriate hold mutex. In consequence, the first queued task is released from the waiting queue of mutex and goes to ready state.

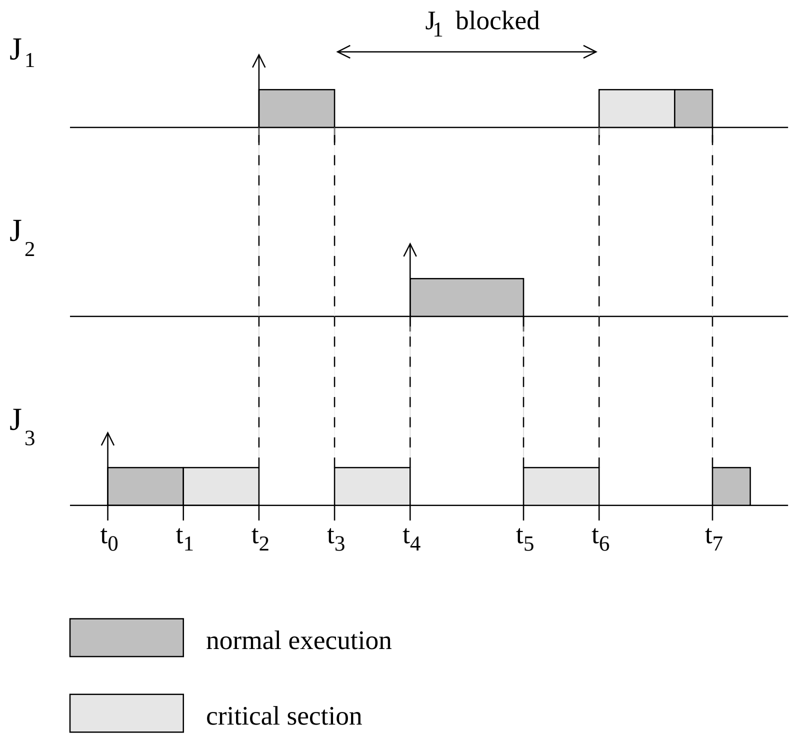


Figure 4.1: An example of priority inversion

Now, consider the example illustrated in Figure 4.1 [2]. Here, three tasks J_1, J_2, and J_3 have decreasing priorities, and J_1 and J_3 share an exclusive resource protected by a binary semaphore S. Job J_3 starts at time t_0 and at time t_1 it locks mutex S and enters the critical section in which can access the resource R. J_1 arrives at time t_2 and preempts J_3 inside its critical section. At time t_3, J_1 attepmts to use the resource R, but it is blocked on the mutex S; thus, J_3 resumes the execution inside its critical section. Now, if J_2 arrives at time t_4 it preempts J_3 (because it has a higher deadline) and increases the blocking time of J_1 by all its duration. This is a situation that, if it recurs with other medium-priority tasks, can lead to uncontrolled blocking and can cause critical deadline to be missed. A *priority inversion* is said to occur in the interval[t_3,t_6], since the highest-priority task J_1 waits for execution of lower-priority tasks(J_2 and J_3).

The priority inversion problem is solved through the use of appropriate protocols called *resource access protocols* that control accesses to any shared resource guarded by mutex. Resource access protocols can be classified as follow:

1. **Protocols for static priority systems**
   - Priority Inheritance protocol (PIP)
   - Priority Ceiling Protocols (PCPs)
     - Basic PCP, Stack-based PCP
2. **Protocols for dynamic priority systems**
   - Preemption Ceiling Protocols (PreCPs)
     - only for deadline-driven dynamic systems
     - Basic PreCP, Stack-based PreCP
   - Dynamic Priority Ceiling Protocol (DPCP) - The priority ceiling of each resource and the system ceiling are updated each time task priorities change.
   - Dynamic Priority Inheritance Protocol (DPIP)

Resource Access Protocols are discussed in a detail in the following sections.

4.2 Priority Inheritance Protocol

Rules of Basic Priority-Inheritance Protocol [2] [4]:

1. **Scheduling Rule**: Ready jobs are scheduled on the processor preemptively in a priority-driven manner according to their current priorities. At its release time t_{ri}, the current priority $\pi(t_{ri})$ of every job J_i is equal to its assigned priority π_{ni} called *nominal priority.* The job remains at this priority except under the condition stated in rule 3.
2. **Allocation Rule**:When a job J_i requests a resource R_j at time t,
   (a) if R_j is free, R_j is allocated to J_i until J_i releases the resource, and
   (b) if R_j is not free, the request is denied and J_i is blocked.
3. **Priority-Inheritance rule**: When the requesting job J_i becomes blocked at time t, the job J_l which blocks J_i inherits the current priority $\pi_i(t)$ of job J_i if the priority $\pi_i(t)$ is higher than priority $\pi_l(t)$ of the job J_l. The job J_l executes at its inherited priority. When the job J_l releases the resource R_j, the highest priority job, if any, blocked on the resource is awakened. Moreover, J_l's priority is updated as follows: if no other jobs are blocked by J_l, it is set to its nominal priority π_{ni}; otherwise it is set to the highest priority of the jobs blocked by J_l.

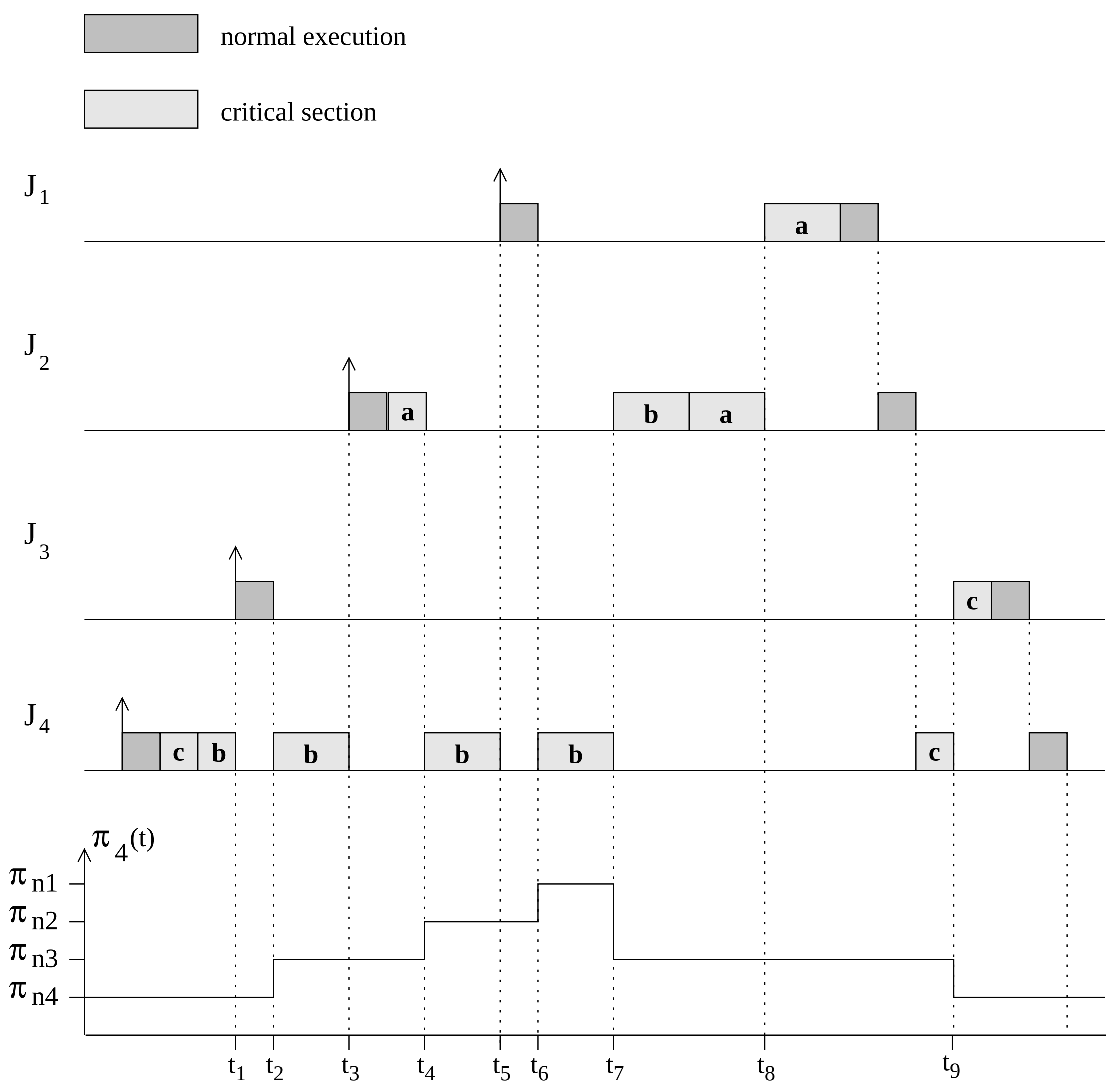


Figure 4.2: An example of priority inheritence

An example of priority inheritance is shown in Figure 4.2. Here, jobs $J_1, \ldots, J_4$ are order in descending nominal priority.

- After its start, job J_4 acquires resource R_c and R_b guarded by a mutex S_c and S_b respectively.
- At time t_1, J_4 is preempted by J_3.
- J_3 is blocked on mutex S_c at time t_2 when it tries to acquire resource R_c. J_4 resumes and inherits the priority $\pi_3(t_2) = \pi_{n3}$.
- At time t_3, J_4 is preempted by J_2, which in turn enters its critical section guarded by mutex S_a.
- At time t_4 J_2 is blocked on mutex S_b. As a consequence, J_4 resumes and inherits the priority $\pi_2(t_4)$.
- At time t_5, J_4 is preempted by J_1.

- At time t_6, J_4 tries to acquire R_a. Since S_a is locked by J_2, $J2$ inherits $\psi_1(t_6)$. However, J_2 is blocked by J_4; hence, for *transitive priority inheritance* J_4 inherits the priority $\pi_1(t_6)$ via J_2.
- When J_4 releases resource R_b at time t_7, its priority is changed to the priority $\pi_3(t_2)$. Priority π_{n1} is now inherited by J_2, which still blocks J_1 until time t_8.
- The rest of the schedule is self-explanatory.

Theorem 4.2.1 (Sha-Rajkumar-Lehoczky) *Under the PIP, a job J can be blocked for at most the duration of min(n,m) critical sections, where n is the number of lower-priority jobs that could block J and m is the number of distinct semaphores that can be used to block J.*

Although PIP bounds the priority inversion problem, the blocking duration for a job can still be substantial because of *chain blocking* (higher priority job which needs to sequentially access different resources is blocked by lower priority jobs holding them). Another problem is that the protocol does not prevent deadlocks [1][2]

4.3 Priority Ceiling Protocol

The Priority Ceiling Protocol (PCP) [2] [4] extends the priority-inheritance protocol to prevent deadlocks, chained blocking and to futher reduce the blocking time.

The main idea of this method is to extend PIP with a rule for granting a lock request on a free semaphore. To avoid multiple blocking, this rule does not allow a job to enter a critical section if there are locked semaphores that could block it. This means that, once a job enters its first crititcal section, it can never be blocked by lower-priority jobs until its completion.[2]

In order to realize this idea, each semphore S(resource R) is assigned a *priority ceiling (resource ceiling)* $\Pi(S)$ $(\Pi(R))$ equal to the priority of the highest-priority job that can lock (acquire) it. Then, a job J is allowed to enter a critical section only if its priority is higher than all priority ceilings of the semaphore currently locked by jobs other than J. At any time, the *system priority ceiling* $\widehat{\Pi}(t)$ is equal to the highest priority ceiling of the resources that are in use at the time, if some resources are in use. If all resources are free at the time, the system ceiling $\widehat{\Pi}(t)$ is equal to Ω, a nonexisting priority level that is lower than the lowest priority of all jobs.[4]
In fact, two versions of PCP have been proposed: basic and stack-based PCP(to provide stack-sharing capability). Both of them have the same worse-case performance, the longest blocking time.

4.3.1 Basic Priority-Ceiling Protocol

Rules of Basic Priority-Ceiling Protocol [4][2]:

1. **Scheduling Rule**:
   (a) At its release time t_{ri}, the current priority $\pi(t_{ri})$ of every job J_i is equal to its assigned priority π_{ni} called *nominal priority.* The job remains at this priority except under condition stated in rule 3.

[1]The deadlock does not depend on the PIP but is caused by an erroneous use of semaphores. The deadlock can be avoided by imposing a total ordering on the semaphore accesses.

   (b) Every ready job J_i is scheduled preemptively and in a priority-driven manner at its current priority $\pi_i(t)$.

2. **Allocation Rule**: Whenever a job J_i requests a resource R_j at time t, one of the following two conditions occurs:
   (a) R_j is held by another job. J_i's request fails and J_i becomes blocked.
   (b) R_j is free.
      i. If J_i's priority $\pi_i(t)$ is higher than the current priority ceiling $\widehat{\Pi}(t)$, R_j is allocated to J_i.
      ii. If J_i's priority $\pi_i(t)$ is not higher than the ceiling $\widehat{\Pi}(t)$ of the system, R_j is allocated to J_i only if J_i is the job holding the resource(s) whose priority ceiling is equal to $\widehat{\Pi}(t)$; otherwise, J_i's request is denied, and J_i becomes blocked.

3. **Priority-Inheritance rule**: When the requesting job J_i becomes blocked at time t, the job J_l which blocks J_i inherits the current priority $\pi_i(t)$ of job J_i if the priority $\pi_i(t)$ is higher than priority $\pi_l(t)$ of the job J_l. The job J_l executes at its inherited priority. When the job J_l releases the resource R_j, the highest priority job, if any, blocked on the resource is awakened. Moreover, J_l's priority is updated as follows: if no other jobs are blocked by J_l, it is set to its nominal priority π_{ni}; otherwise it is set to the highest priority of the jobs blocked by J_l.

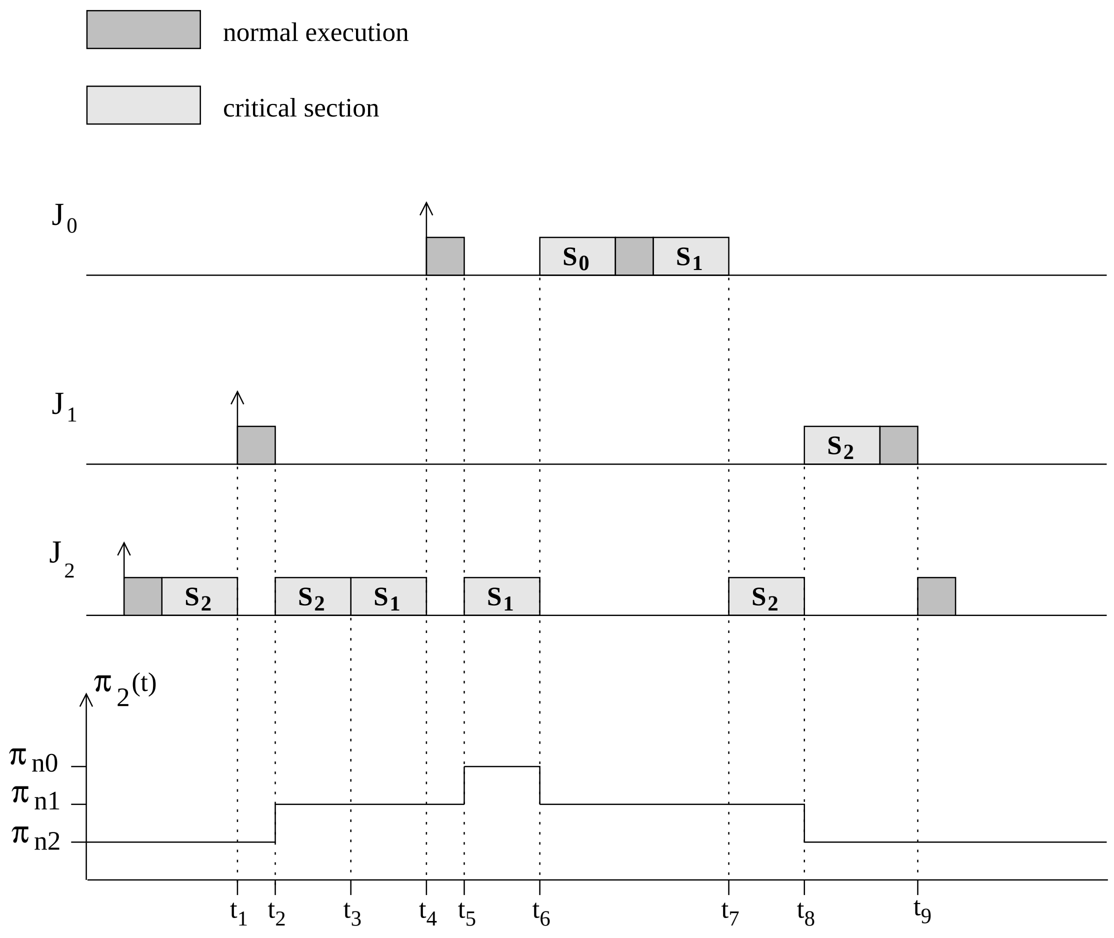


Figure 4.3: Example of basic Priority Ceiling Protocol

In order to illustrate Basic PCP, consider three jobs J_0, J_1, J_2 having decreasing priorities. The highest priority job J_0 sequentially accesses two critical sections guarded by mutexes S_0 and $S1$; J_1 uses S_2; J_2 uses S_2 and $S1$. Therefore, all mutexes are assigned the following priority ceilings: $\Pi(S_0) = \pi_{n0}$, $\Pi(S_1) = \pi_{n0}$, $\Pi(S_2) = \pi_{n1}$.
Figure 4.3 illustrates the scheduling of jobs by the basic PCP.

- At time t_0, J_0 is started and later acquires mutex S_2. The system priority ceiling $\widehat{\Pi}(t)$ is set to $\Pi(S_2) = \pi_{n1}$.
- At time t_1, J_1 preempts J_0.
- At time t_2, J_1 attempts to lock S_2, but it is blocked by the protocol because $\pi_1(t_2)$ is not higher than $\widehat{\Pi}(t_2)$. Then, J_2 inherits the priority of J_1 and resumes its execution.
- At time t_3, J_2 successfuly locks mutex S_1, the system ceiling raises to $\Pi(S_1)$.
- At time t_4, while J_2 is executing at a priority π_{n1}, J_0 becomes ready and preempts J_2 because $\pi_0(t_4) > \pi_{n1}$.
- At time t_5, J_0 atemptes to lock S_0. However, J_0 is blocked by the protocol because its priority is not higher than system ceiling. J_2 inherits the priority of J_0 and resumes its execution.
- At time t_6, J_2 unlocks S_2. Consequently J_0 is awakened, J_2's priority returns to π_{n1} and the system ceiling to $\Pi(S_2)$. At this point, now $\pi_0(t_6) > \widehat{\Pi}(t_6)$, therefore J_0 preempts J_2 and executes until completion.
- The rest of the schedule is self-explanatory.

Theorem 4.3.1 (Sha-Rajkumar-Lehoczky) *Under the basic PCP, a job J_i can be blocked for at most the duration of one critical section.[2]*

4.3.2 Stack-Based, Priority-Ceiling Protocol

Suppose that a resource in the system is the run-time stack. Thus far, we have assumed that each job has its own run-time stack. Sometimes, especially in systems where the number of jobs is large, it may be necessary for the jobs to share a common run-time stack, in order to reduce overall memory demand. However, it must be ensured deadlock-free sharing of the run-time stack among jobs and contiguous stack space of job. These requirements can be achieved by the rule that no job is ever blocked once its execution begins.[4]

Stack-based PCP extends the basic PCP in that it allows the use of multiunit resources and allows the sharing of runtime stack-based resources. The main difference between them is the time at which task is blocked. Whereas under the basic PCP a task is blocked at the time it makes its resource request and is not succesful, under the stack-based PCP a task is blocked at the time it attempts to preempt rather than to be blocked during its execution. This *early blocking* slightly reduces concurrency but saves unnecessary context switches, simplifies the implementation of the protocol, and allows the runtime sharing of stack resources. Therefore, the mentioned behaviour is called stack resource policy.[2]

Rules Definning Basic Stack-Based, Priority-Ceiling Protocol [4][2]:

1. **Update of the Current Ceiling**: Whenever all the resources are free, the ceiling of the system is Ω. The ceiling $\widehat{\Pi}(t)$ is updated each time a resource is allocated or freed.

2. **Scheduling Rule**: After a job is released, it is blocked from starting execution until its assigned priority is higher than the current ceiling $\widehat{\Pi}(t)$ of the system. At all times, jobs that are not blocked are scheduled on the processor in a priority-driven, preemptive manner according to their assigned priorities.

3. **Allocation Rule**: Whenever a job requests a resource, it is allocated the resource.

In order to illustrate stack-based PCP, consider three jobs J_0, J_1, J_2, J_3 having decreasing priorities. The highest priority job J_0 accesses critical section guarded by mutex S_0; J_1,J_3 uses S_1; and J_2 does not accesses any critical section. Therefore, all mutexes are assigned the following priority ceilings: $\Pi(S_0) = \pi_{n0}$, $\Pi(S_1) = \pi_{n1}$.

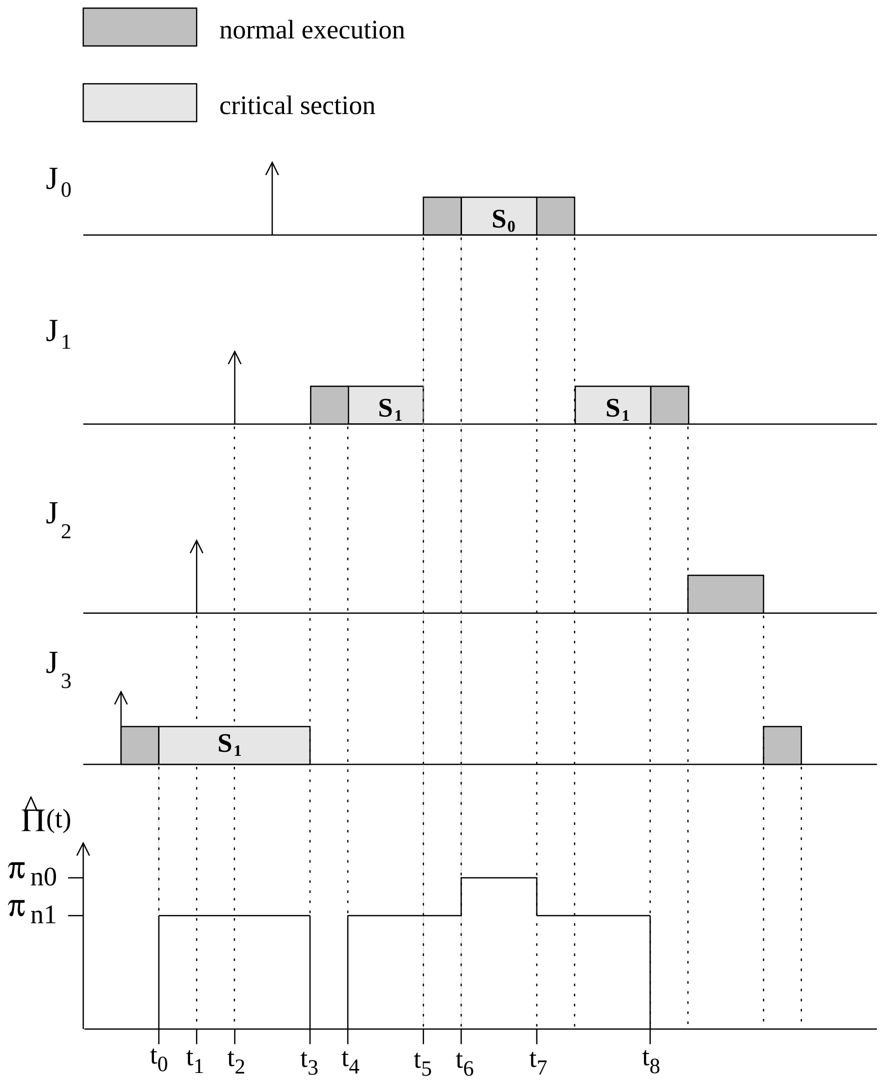


Figure 4.4: Example of stacked-based Priority Ceiling Protocol

Figure 4.4 illustrates the scheduling of jobs by the stack-based PCP.

- At time t_0, J_3 acquires mutex S_1. The system priority ceiling $\widehat{\Pi}(t)$ is set to $\Pi(S_1) = \pi_{n1}$.
- At time t_1, J_2 is released and is blocked from starting because its priority is not higher than the ceiling of the system $\widehat{\Pi}(t_1)$.

- At time t_2, for the same reason J_1 does not start execution when it is released.
- At time t_3, J_3 releases the resources guarded by mutex S_1 and the system ceiling is decreased to Ω. Consequently, J_1 starts to execute since it has the highest priority among all the jobs ready at time and its priority $\pi_1(t_3)$ is higher than the current ceiling of the system.
- At time t_4 the executing job J_1 locks mutex S_1, the system priority ceiling $\widehat{\Pi}(t)$ is set to π_{n1}.
- At time t_5, J_0 is released and preempts J_1 since it has the highest priority among all the jobs ready at time and its priority $\pi_0(t_5)$ is higher than the current ceiling of the system which is later raised to π_{n0} when J_0 locks mutex S_0 at time t_6.
- At time t_7, J_0 unlocks mutex S_0, the ceiling of the system returns back to π_{n1}.
- After J_0 terminates, J_1 resumes its execution since it has the highest priority among all the jobs ready at that time.
- The rest of the schedule is self-explanatory.

4.4 Preemption-Ceiling Protocol

In dynamic-priority deadline-driven systems, the scheduling decision is based on the dynamic priority - absolute deadline that differs in each job of task. For this class of systems, Baker [5] proposed a simpler approach to control resource accesses. The approach is based on the clever observation that the potentials of resource contentions in such dynamic-priority system do not change with the time, just as in fixed-priority systems, and hence can be analyzed statically.[4]

Besides a priority, a task τ_i is also characterized by a *preemption level* ψ_i. The preemption level is a static parameter, assigned to a task at its creation time and associated with all jobs of the task. The preemption-level assignment is a function of priority π and release time r. In general, the preemption-level assignment function must satisfy the condition:

If π_i is higher than π_k and $r_i > r_k$, then ψ_i is higher than ψ_k.[4]

Under EDF scheduling, the previous condition is satisfied if preemption levels are ordered inversely with respect to the order of relative deadlines:

$$\pi_i > \pi_k \iff D_i < D_k$$

Note that, in the static priority systems preemption level of task would equal to its assigned static priority.
A preemption-ceiling protocol makes scheduling decision based on the preemption level of the job in a manner similar to the priority-ceiling protocol. After assigning preemption levels to all the jobs, we determine the preemption ceiling of each resource. The *preemption ceiling* $\psi(R)$ of a resource R is the highest preemption level of all jobs that require the resource. The *preemption ceiling of the system* $\widehat{\Psi}(t)$ at any time t is the highest preemption ceiling of all the resources that are in use at t. If all resources are free at the time, the system preemption ceiling $\widehat{\Pi}(t)$ is equal to Ω, a nonexisting preemption level that is lower than the lowest preemption level of all jobs.[4]

Like the PCP, the PreCP also has a basic version and a stack-based version.

4.4.1 Basic Preemption-Ceiling Protocol

Rules of Basic Preemption-Ceiling Protocoli [4][2]:

1. **Scheduling Rule**: The scheduling rule is the same as the coresponding rule of the priority-ceiling protocol.
2. **Allocation Rule**: Whenever a job J requests resource R at time t, one of the following two conditions occurs:
   (a) R is held by another job. J's request fails, and J becomes blocked.
   (b) R is free.
      i. If J's preemption level $\psi(t)$ is higher than the current preemption ceiling $\widehat{\Psi}(t)$ of the system, R is allocated to J.
      ii. If J's preemption level $\psi(t)$ is not higher than ceiling $\widehat{\Psi}(t)$ of the system, R is allocated J only if J is the job holding the resource(s) whose preemption ceiling is equal to $\widehat{\Psi}(t)$; otherwise, J's request is denied, and J becomes blocked.
3. **Priority Inheritance Rule**: The priority inheritance rule is the same as the coresponding rule of the priority-ceiling protocol.

4.4.2 Stack-Based Preemption-Ceiling Protocol

Rules of Basic Stack-Based, Preemption-Ceiling Protocol [4] [2]:

1. **Update of the Current Ceiling**: Whenever all the resources are free, the preemption ceiling of the system is Ω. The preemption ceiling $\widehat{\Psi}(t)$ is updated each time a resource is allocated or freed.
2. **Scheduling Rule**: After a job is released, it is blocked from starting execution until its preemption level is higher than the current ceiling $\widehat{\Psi}(t)$ of the system and the preemption level of the executing job. At any time t, jobs that are not blocked are scheduled on the processor in a priority-driven manner according to their assigned priorities.
3. **Allocation Rule**: Whenever a job J requests for a resource R, it is allocated the resource.
4. **Priority-Inheritance Rule**: When some job is blocked from starting, the blocking job inherits the highest priority of all the blocked jobs.

Note the necessity of the priority-inheritance rule of the stack-based preemption-ceiling protocol when preemption levels are assigned on the basis of job's relative deadlines.

Theorem 4.4.1 (Baker) *Under the Stack Resource Policy, a job J_i can be blocked for at most the duration of one critical section.[2]*

5 RTEMS

5.1 Introduction

RTEMS [6], Real-Time Executive for Multiprocessor Systems, is a real-time executive (kernel) which provides a high performance environment for embedded applications including the following features:[6]

- multitasking capabilities
- homogeneous and heterogeneous multiprocessor systems
- event-driven, priority-based, preemptive scheduling
- resource access protocols (PCP,PIP)
- dynamic memory allocation
- high level of user configurability
- services to real-time applications through managers (task,timer,semaphore,etc.)
- open API(POSIX 1003.1b API including threads, RTEID/ORKID based Classic API, uITRON 3.0 API) and interface standards (TCP/IP including BSD Sockets)
- networking support: FreeBSD TCP/IP stack, UDP, TCP, BOOTP, ICMP, DHCP, PPPD, RPC, XDR, FTP server, HTTPD server, Telnet server, etc.
- filesystem support: FAT32, FAT16, and FAT12, TFTP client filesystem, NFS client
- debugging support: GNU debugger (gdb), DDD GUI interface to gdb, Debug over ethernet, Debug over serial port

The primary RTEMS source image is a collection of software that is licensed under a variety of free and open source licenses appropriate for using in embedded systems.
RTEMS has been implemented in both the Ada and C programming languages and ported to the following processor families:[6]

- arm - ARM V7 and above
- c4x - Texas Instruments C3x and C4x DSPs
- h8300 - Hitachi H8 family
- hppa1.1 - Hewlett-Packard PA-RISC
- i386 - Intel i386, i486, Pentium and above, AMD Athlon and above
- i960 - Intel i960 family
- m68k - Motorola m680x0, m683xx, CPU32, and Coldfire CPUs
- mips - MIPS ISA Levels 1 and above for 32 and 64 bit CPU models
- no_cpu - Example port to "no cpu"
- or32 - OpenCores OpenRisc32 CPU
- powerpc - IBM and Motorola PowerPC 4xx, 5xx, 6xx, 7xx, 8xx, 74xx, and 75xx

- sh - Hitachi SH1, SH2, SH3, and SH4
- sparc - SPARC V7 and above CPUs
- unix - Synthetic target CPU which allows RTEMS programs to execute natively on Linux, Solaris, FreeBSD, Cygwin, and HPUX.

In the following sections the architecture and internal structure of RTEMS will be described in such an extent to give fundamentals to the implementation of EDF scheduler. Because the most of application examples in RTEMS distribution are written using RTEMS Classic API which provides more specific features, Classic API directives are considered in the text.

5.2 RTEMS Internal Architecture

RTEMS consists of a set of layered components called *resource managers* that utilize functions such as scheduling, dispatching, and object management in the executive core and provide a set of services to a real-time application system. The executive core depends on a small set of CPU dependent routines. All these components together provide a powerful run time environment for the development of efficient real-time application systems. The following Figure 5.1 illustrates this organization:[7]

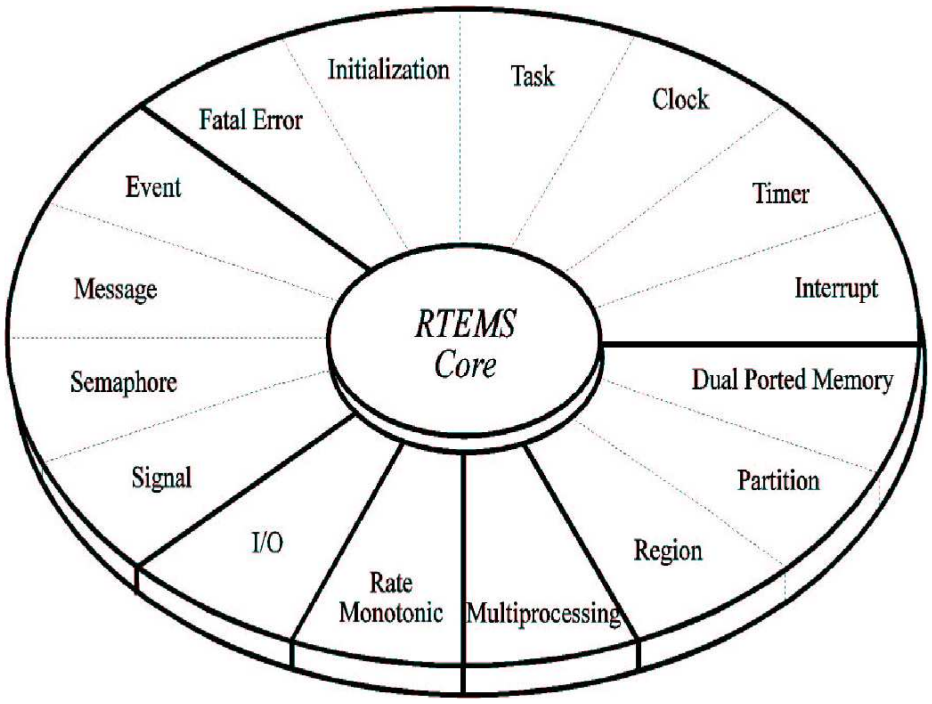


Figure 5.1: RTEMS managers

5.3 Directory structure

RTEMS has a well-organised source tree directory structure depicted in Figure 5.2 . The processor and board dependent source files are isolated from generic files. When RTEMS is configured and built, object directories and an install point will be automatically created based on the target CPU family and BSP selected. Therefore, user may have compiled rtems images with different options for particular target board and processor. [7]

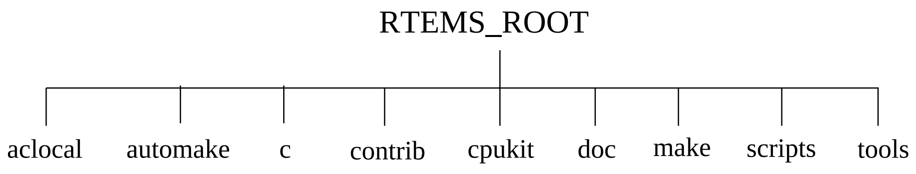


Figure 5.2: The RTEMS directory structure

Follows the description of the content of chosen directories under the root of the source tree referenced as RTEMS-ROOT:

- RTEMS_ROOT/c/src/tests/itrontests
  This directory contains the test suite for the RTEMS ITRON API.
- RTEMS_ROOT/c/src/tests/psxtests
  This directory contains the test suite for the RTEMS POSIX API.
- RTEMS_ROOT/c/src/tests/sptests
  This directory contains the test suite for the RTEMS Classic API when executing on a single processor.
- RTEMS_ROOT/cpukit/posix/
  This directory contains the RTEMS implementation of the threading portions of the POSIX API.
- RTEMS_ROOT/cpukit/rtems/
  This directory contains the implementation of the Classic API.
- RTEMS_ROOT/cpukit/itron/
  This directory contains the implementation of the ITRON API.
- RTEMS_ROOT/cpukit/score/
  This directory contains the "SuperCore" of RTEMS. All APIs are implemented as SuperCore services in principle. [1]The inline subdirectory contains inline files, src source and include header files.

5.4 Objects

RTEMS provides directives to dynamically create, delete, and manipulate a set of predefined object types. These types include tasks, message queues, semaphores, memory regions, memory partitions, timers, ports, and rate monotonic periods. All instances of object type [2] have assigned an object name and an object identifier (OID). Because of multiprocessing support, RTEMS distinguishes local and global objects.

An object identifier is a unique unsigned thirty-two bit entity composed of three parts: object class, node, and index. Figure 5.3 illustrates the structure of OID.

[1]Classic API tasks, POSIX threads, and ITRON tasks are all implemented in terms of SuperCore threads. This provides a common infrastructure and a high degree of interoperability between the APIs - services from all APIs may be used by any task/thread independent of the API used to create it.

[2]For abbreviation, futher in the text instance of object type is shortly called object.

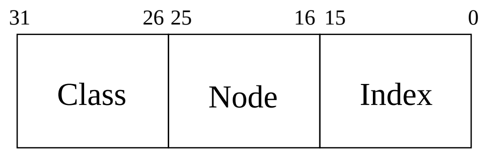


Figure 5.3: The object identifier structure

The most significant six bits are the object class followed by ten bits of the processor node number on which the object was created. The least significant sixteen bits form an identifier within a particular object type. For a programmer, it is easier to identify objects by name rather than OID which is convenient to routines inside the system. RTEMS maintains a name table for each object type and provides directives that map the object name to object's ID according to this table.

The following example shows how to create an object name:

```
rtems_object_name  my_name;

my_name = rtems_build_name( 'T', 'A', 'S', 'K' );
```

The directive rtems_build_name packs the sequence of maximum four input characters to 32 bits return value.

5.5 Task

A task or process is implemented as entity called *thread*[3]. RTOS maintains information about every thread in system in Task Control Block (TCB). In RTEMS TCB is defined by struct Thread_Control_struct in RTEMS_ROOT/cpukit/score/include/rtems/score/thread.h.

```
struct Thread_Control_struct {
  Objects_Control                      Object;
  States_Control                       current_state;
  Priority_Control                     current_priority;
  Priority_Control                     real_priority;
  unsigned32                           resource_count;
  Thread_Wait_information              Wait;
  Watchdog_Control                     Timer;
  unsigned32                           suspend_count;
  boolean                              is_global;
  boolean                              is_preemptible;
  Chain_Control                       *ready;
  .
  .
  .
};
```

Description of chosen structure fields follows:

- **Object** This is the first field in every structure of an object type. It contains fields like object name, object id, and chain node through which TCBs can be concatenated.

[3]Futher, terms process, thread, task can be interchangeable.

- **current_state** Stores the current state of task as task changes its state during its life. Note that RTEMS futher classifies some of the main states (ready, running, waiting, terminated)
- **real_priority** Nominal priority assigned to task when it is released.
- **current_priority** Actual task's priority, may be higher than nominal priority.
- **resource_count** Stores the total number of resources held by the task.
- **Wait** This structure holds necessary information when being in a wait state.
- **is_global** The field is set to TRUE if the task is global.
- **is_preemptible** The field is set to TRUE if the task is preemptible.
- **ready** The pointer indentifies priority level list of ready queue in which the task is queued.

5.6 The RTEMS chain

RTEMS chain is a kernel data structure used to build more complicated structures like queues, lists (waiting, ready queue, free object list, etc). Therefore, it is worth to explain its implementation which is also common for other operating systems.

RTEMS chain is a double-linked list of zero or more nodes starting with head and ending with tail control nodes. The chain node is defined by struct Chain_Node_struct in RTEMS_ROOT/cpukit/score/include/rtems/score/chain.h file.

```
typedef struct Chain_Node_struct Chain_Node;

struct Chain_Node_struct {
  Chain_Node *next;
  Chain_Node *previous;
};

typedef struct {
  Chain_Node *first;
  Chain_Node *permanent_null;
  Chain_Node *last;
} Chain_Control;
```

The implementation does not require special checks for manipulating the first and the last elements on the chain. To accomplish this the chain control structure is treated as two overlapping chain nodes. The permanent head of the chain overlays a node structure on first and permanent_null fields. The permanent tail of the chain overlays a node structure on permanent_null and last fields of the structure. This trick is illustrated in Figure 5.4.

RTEMS Super core provides directives for manipulating chain specified in calling argument. The operations with chain include initializing chain, inserting a node into chain, extracting a node from chain, appending a node onto the end of chain.

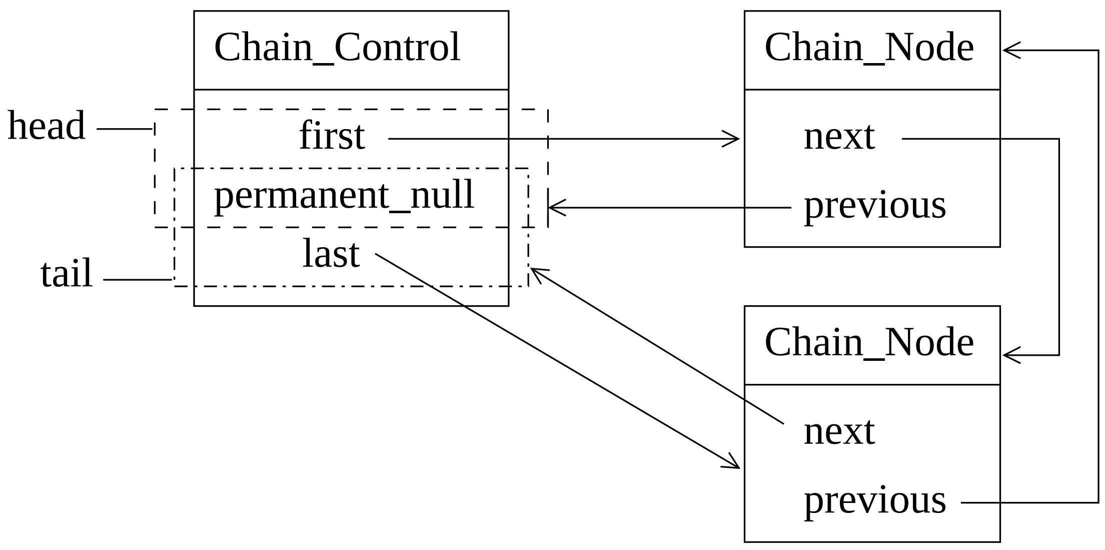


Figure 5.4: The RTEMS chain

As already mentioned above, typically, RTEMS chain is used to build a more complicated structures like queues, lists. A more complicated structure includes Chain_Node structure as the first element in its control structure. Since the included chain node and the higher level structure start at the same address, programmer can cast the pointers back and forth between them. This neat-handed technique is shown on the example of a chain of TCBs which is depicted in Figure 5.5. TCB includes the Chain_Node structure in the Object.Node field. Since this field and TCB start at the same address, a pointer to TCB can be casted to the chain node and passed as argument in a call of directive manipulating with the chain. The reverse pointer casting is possible as well.

5.7 The ready queue

RTEMS implements fixed priority-driven scheduling, preemptable by default. Tasks waiting for processor reside in the ready queue ordered by ascending priority until the execution on the processor. The scheduling algorithm guarantees that the task which is executed on the processor at any point in time is the one with the highest priority among all tasks in the ready queue.

Figure 5.5 illustrates RTEMS ready queue. The ready queue is implemented as an 256 elementary array of FIFO lists. Every element (list) of the array represents priority level. The zero level designates the highest priority. Tasks with equivalent priorities are placed on the same FIFO list associated with their priority level. Figure 5.5 depicts the stucture of RTEMS ready queue.

The executing task is the first one in the first non-empty priority-level FIFO list from the beginning of the ready queue. It would be time-consuming to linearly search for that non-empty FIFO list. Therefore, a 256-bit set is maintained in which every non-empty FIFO list has a set bit coresponding to its priority level. This bit field is handled by low-level cpu instructions and thus the first non-empty list can be found efectively.

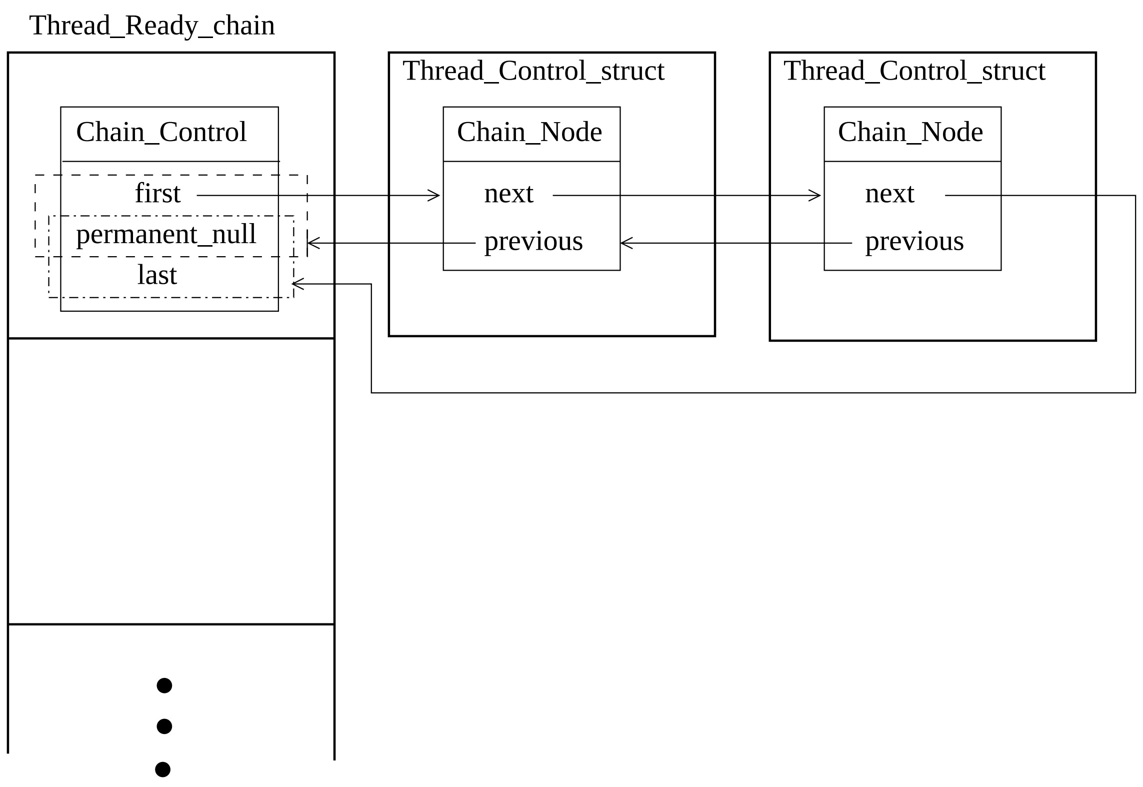


Figure 5.5: The RTEMS ready queue

Note that the executing task remains in the ready queue. A task is removed from the ready queue only when it terminates or becomes blocked. The kernel variable _Thread_Heir holds the task that is the next to run on processor. Whenever a task changes its priority, gets ready or its state is set to the requested, the _Thread_Heir variable may be updated. In RTEMS code there are test conditions whether heir task is also the executing and then context switch is performed accordingly.

5.8 Task states

Tasks in an RTEMS system must always be in one of the five allowable task states. These states are: executing, ready, waiting(or blocked), dormant, and non-existent (terminated). These states are very similar to general task states described in the in the chapter 2 and transitions between them are illustrated in Figure 5.6 [7].

A ”task” is in the non-existent state before a rtems_task_create API directive has been issued. A task enters the non-existent state from any other state in the system when it is deleted with the rtems_task_delete directive.

When a task is created calling the rtems_task_create directive it enters the dormant state. Although the task exists in the system, it cannot actively compete for system resources. It will remain in the dormant state until it is started by the rtems_task_start directive, at which time it is inserted into the ready queue and becomes ready. Then the task can be scheduled for the processor.

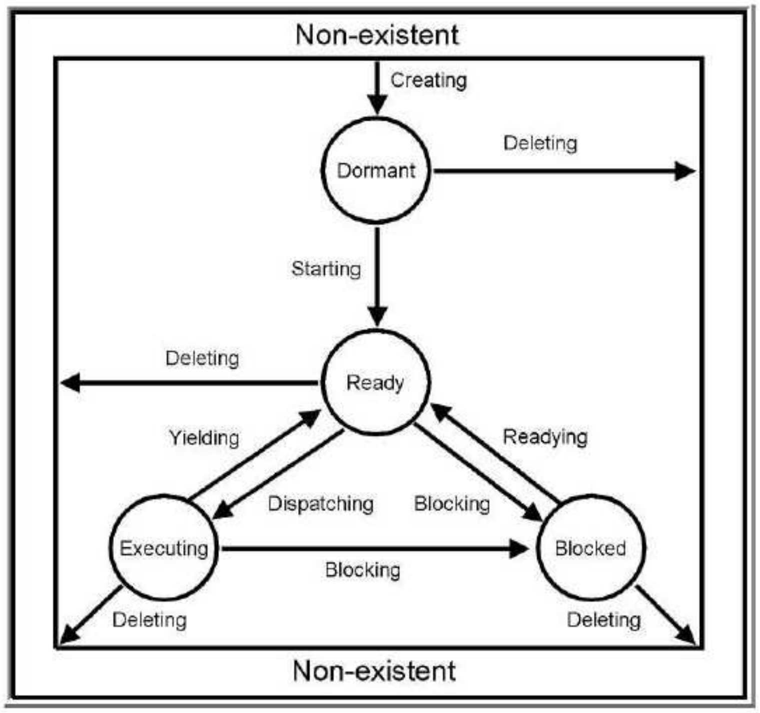


Figure 5.6: States of task in RTEMS

A task occupies the blocked state whenever it is unable to be scheduled to run. A task may block itself or be blocked by other tasks in the system. The running task blocks itself through directives like rtems_task_wake_after, rtems_task_wake_when that cause the task to wait. The only way a task can block a task other than itself is with the rtems_task_suspend directive. A task enters the blocked state due to any of the following conditions [7]:

- A task issues a rtems_task_suspend directive which blocks either itself or another task in the system.
- The running task issues a rtems_message_queue_receive directive with the wait option and the message queue is empty.
- The running task issues an rtems_event_receive directive with the wait option and the currently pending events do not satisfy the request.
- The running task issues a rtems_semaphore_obtain directive with the wait option and the requested semaphore is unavailable.
- The running task issues a rtems_task_wake_after directive which blocks the task for the given time interval. If the time interval specified is zero, the task yields the processor and remains in the ready state.
- The running task issues a rtems_task_wake_when directive which blocks the task until the requested date and time arrives.
- The running task issues a rtems_region_get_segment directive with the wait option and there is not an available segment large enough to satisfy the task's request.
- The running task issues a rtems_rate_monotonic_period directive and must wait for the specified rate monotonic period to conclude.

A task enters the ready state due to any of the following conditions [7]:

- A running task issues a rtems_task_resume directive for a task that is suspended and the task is not blocked waiting on any resource.
- A running task issues a rtems_message_queue_send, rtems_message_queue_ broadcast, or a rtems_message_queue_urgent directive which posts a message to the queue on which the blocked task is waiting.
- A running task issues an rtems_event_send directive which sends an event condition to a task which is blocked waiting on that event condition.
- A running task issues a rtems_semaphore_release directive which releases the semaphore on which the blocked task is waiting.
- A timeout interval expires for a task which was blocked by a call to the rtems_task_ wake_after directive.
- A timeout period expires for a task which blocked by a call to the rtems_task_ wake_when directive.
- A running task issues a rtems_region_return_segment directive which releases a segment to the region on which the blocked task is waiting and a resulting segment is large enough to satisfy the task's request.
- A rate monotonic period expires for a task which blocked by a call to the rtems_rate_monotonic_period directive.
- A timeout interval expires for a task which was blocked waiting on a message, event, semaphore, or segment with a timeout specified.
- A running task issues a directive which deletes a message queue, a semaphore, or a region on which the blocked task is waiting.
- A running task issues a rtems_task_restart directive for the blocked task.

Note that RTEMS internally futher classifies the waiting state according to the reason (STATES_WAITING_FOR_SEMAPHORE, STATES_WAITING_FOR_PERIOD, etc.), adds another states (e.g STATES_TRANSIENT when task is in transition between the two main states). For more details about internal states see RTEMS_ROOT/cpukit/score/include/rtems/score/states.h.

5.9 Waiting queue

There is no one general waiting queue in RTEMS kernel but several according to resource (e.g. mutex, semaphore, event, message, time) on which the tasks are blocked. Waiting queue for resources like mutex, semaphore, messages are more complicated structures including RTEMS chain. The waiting queue of mutex is described in a detail in chapter 6.

5.10 Configuration table

RTEMS must know configuration information for an application. The configuration information includes the length of clock tick, the maximum number of each RTEMS object that can be created, the application initialization tasks, and the device drivers in the application.

RTEMS provides the /cpukit/sapi/include/confdefs.h header file that contains the configuration table template that will be automatically instantiated during the compilation of application based on the setting of a number of macro in the application. Examples of macros:

```
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
#define CONFIGURE_MICROSECONDS_PER_TICK   1000 /* 1 millisecond */
#define CONFIGURE_TICKS_PER_TIMESLICE       50 /* 50 milliseconds */
#define CONFIGURE_MAXIMUM_TASKS 4
#define CONFIGURE_MAXIMUM_SEMAPHORES 3
#define CONFIGURE_EXTRA_TASK_STACK 20  /* 20B */
```

Although macros names are self-explanatory and can be found in /textsfconfdef.h file, they are documented in [7] as well.

5.11 Clock manager

The clock manager provides support for time of day and other time related capabilities. Elapsed time is measured in ticks. A tick is defined to be an integral number of microseconds which is specified by the user in the Configuration Table. For the features provided by the clock manager to be utilized, periodic timer interrupts are required. Therefore, a real-time clock or hardware timer is necessary to create the timer interrupts. [7]

When RTEMS_CLOCK_GET_TICKS_SINCE_BOOT specified in option argument of rtems_clock_get directive, it returns ticks since in the system started in the time_buffer argument. In fact, the value of _Watchdog_Ticks_since_boot kernel variable is returned.

5.12 Rate Monotonic Manager

The most of tasks in real-time systems have a periodic nature and hence their behaviour is simple predictable. A periodic task is one which must be executed at a regular interval. Therefore, periodic tasks are characterized by the length of their period and execution time.

Periodic execution cannot be accomplished by rtems_task_wake_after, rtems_task_wake_when because calling that directive only ensure that task is awaken after specified ticks and do not take into account the computation time between adjacent callings of rtems_task_wake_after directive. RTEMS supports periodic tasks by means of Rate Monotonic manager.

The rate monotonic manager provides facilities to implement periodic task releasing. The directives provided by the rate monotonic manager are [7]:

- rtems_rate_monotonic_create(rtems_name name, rtems_id *id) - Create a rate monotonic period called name and its id is returned in id argument
- rtems_rate_monotonic_ident(rtems_name name, rtems_id *id) - Get ID of a period associated with name.
- rtems_rate_monotonic_cancel(rtems_id id) - Cancel a period which can be reinitiated by the next invocation of rtems_rate_monotonic_period with id.
- rtems_rate_monotonic_delete(rtems_id id) - Delete a rate monotonic period specified by id.

- rtems_rate_monotonic_period(rtems_id id, rtems_interval length) - Conclude current/Start next period id with a length of periodic ticks.
- rtems_rate_monotonic_get_status(rtems_id id, rtems_rate_monotonic_period_status *status) - Obtain status information on period id in the rtems_rate_monotonic_period_status structure.

The usage of rate monotonic directives demonstrates the following example [7]:

```
rtems_task Periodic_task(rtems_task_argument arg)
{
   rtems_name        name_1, name_2;
   rtems_id          period_1, period_2;
   rtems_status_code status;

   name_1 = rtems_build_name( 'P', 'E', 'R', '1' );
   name_2 = rtems_build_name( 'P', 'E', 'R', '2' );
   (void ) rtems_rate_monotonic_create( name_1, &period_1 );
   (void ) rtems_rate_monotonic_create( name_2, &period_2 );
   while ( 1 ) {
      if ( rtems_rate_monotonic_period( period_1, 100 ) == TIMEOUT )
      break;
      if ( rtems_rate_monotonic_period( period_2, 40 ) == TIMEOUT )
      break;
      /*
       * Perform first set of actions between clock
       * ticks 0 and 39 of every 100 ticks.
       */

      if ( rtems_rate_monotonic_period( period_2, 30 ) == TIMEOUT )
      break;
      /*
       * Perform second set of actions between clock 40 and 69
       * of every 100 ticks. THEN ...
       *
       * Check to make sure we didn't miss the period_2 period.
       */

      if ( rtems_rate_monotonic_period( period_2, STATUS ) == TIMEOUT )
      break;
      (void) rtems_rate_monotonic_cancel( period_2 );
   }

   /* missed period so delete period and SELF */
   (void ) rtems_rate_monotonic_delete( period_1 );
   (void ) rtems_rate_monotonic_delete( period_2 );
   (void ) task_delete( SELF );
}
```

In the above example, two rate monotonic periods called name_1 and name_2 are created as a part of its initialization. The first time the loop is executed, the rtems_rate_monotonic_period directive will initiate the period_1 period for 100 ticks or period_2 for 40 ticks and return immediately. The next calling of rtems_rate_monotonic_period(period_2,30) directive breaks the main While cycle if the task missed the deadline (period) or blocks the task for remainder of the 40 tick period. In the later case after the possible blocking, period_2 period is reinitiated with 30 tick period. The calling rtems_rate_monotonic_period(period_2, STATUS) breaks the main While cycle if the task missed the deadline (period) or blocks the task for remainder of the 30 tick period and after that period is reinitiated with the next 30 ticks. The rtems_rate_monotonic_cancel(period_2) directive cancels period_2 period that would otherwise expire on the subsequent call of the rtems_rate_monotonic_period(period_2,40) directive in the beginning of the next cycle since before the task can be blocked for the remainder of the 100 tick period. Every time by the rtems_rate_monotonic_period(period_2,40) call the period_2 period will be initiated immediately and the task will not block.

6 The EDF scheduler implementation

6.1 Introduction

From the implementation point of view, scheduler consists of two parts: a ready queue and an interface. The development of scheduler mainly relates to ready queue which must meet certain criterias. An interface only provides directives manipulating with the queue.

6.2 The EDF scheduler design

When talking about the design of the EDF scheduling algorithm, we have to distinguish the case in which the algorithm is developed on top of a generic priority based operating system, from the case in which the algorithm is implemented from scratch. In both cases the main goal is to achieve computation complexity $O(1)$.

6.2.1 The design on top of fixed-priority kernel

Most of fixed-priority real-time kernels have the ready queue implemented in a way described in chapter 5 in the case of RTEMS.

When considering the development of the scheduling algorithm on top of a kernel based on a set of fixed priority levels, it is indeed true that the EDF implementation is not easy, nor efficient. In order to be scheduled by the existing fixed-priority algorithm , the job's absolute deadline have to be mapped to priority that is not always unique. For example, consider the case in which two deadlines *da* and *db* are mapped into two adjacent priority levels and a new periodic job is released with an absolute deadline *dc* , such as $da < dc < db$ (see Figure 6.1). In this situation, there is not a priority level between *da*‘s and *db*‘s priority levels that can be selected to map *dc*. This problem can only be solved by remapping *da* and *db* into two new priority levels which are not consecutive. Notice that, in the worst case, all current deadlines may need to be remapped, increasing the cost of the operation. [1]

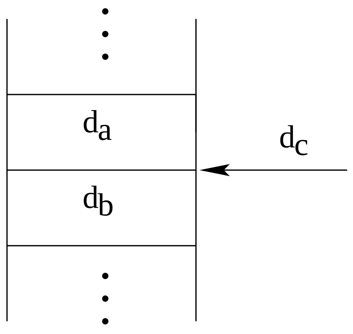


Figure 6.1: The problem of remapping deadlines

It is worth mentioning that the scheduler implementation in RTLinux [8] combines both static and dynamic priority scheduling to keep the system functionality backwards compatible. Tasks are ordered by priority and among the same static priority level the tasks with earlier deadline are executed first. Hence, each task has two scheduling attributes: priority and deadline. It is the Rate Monotonic policy, and the EDF policy at each priority level.

6.2.2 The design from scratch

Other approach is to implement the EDF ready queue ordered by increasing absolute deadlines from scratch.

An advantage of RM with respect to EDF is that, if the number of priority levels is not high, the RM algorithm can be implemented more efficiently by splitting the ready queue into several FIFO queues, one for each priority level. In this case, the insertion of a task into the ready queue[1], it can be performed in $O(1)$. Unfortunately, the same solution cannot be adopted for EDF, because the number of queues would be too large (e.g., equal to 2^{32} if the system time is stored in a 32-bits variable). [1]

Generaly, it is required that the EDF ready queue has the following features:

- dynamic size preferably growing linearly with the number of taks in the system
- ordered by ascending absolute dedlines of tasks
- to facilitate the effective enqueueing a new task into the ready queue and dequeueing the first task from the ready queue.
- to facilitate the effective selection of the next task to be run

Almost all the above features can be satisfied when implementing the ready queue as a dynamic linked list of elements representing tasks (TCBs usually). Such a linked list is a data structure expanding dynamically and linearly with the number of task, enables in a simple way to select the next running task and to manipulate (insert, delete) elements of list especially when it is a double-linked list. The only problem that appears is how to effectively find a place in the queue to insert an incomming task. The linear search through the ready queue is inadmissible because of computational complexity $O(n)$.[2]

To solve the described drawback a searching mechanism must be built upon a linked list to find an appropriate place in the ready queue for incomming task in an time-effective manner. Besides time complexity, memory requirements has to be taken into account as well. The following searching algorithms appear, all having complexity $O(log_2 n)$ of insert and delete operations:

- Brodal's data structure[9]
  - complicated data structure
  - requires temporary array
- Probabilistic Data Structures for Queues[10]
  - a variety of so called simple bottom-up sampled heaps (SBSH)
- Deterministic Data Structures - AVL, Red-Black tree (RBT)

[1]TCB represents a task in the system. Therefore, when talking about inserting a task into ready queue, it means inserting its TCB.

[2]The linear search would be simple and efficient for small number of tasks. But when using RTOS it is expected to schedule hundreds of tasks.

6.3 The Implementation

As an "implementation environment" for EDF scheduler I have chosen RTEMS real-time system because it is open source and has only fixed-priority scheduler implemented.

Although RTEMS is a fixed-priority kernel, I desists from the scheduler implementation that utilizes its priority queue because I do not consider the mapping and eventual remapping of task's absolute deadline to priority as an effective solution. However, when implementing the ready queue from scratch it can be better customized and optimized for the EDF scheduling. For that reasons, I have decided to take the design from scratch; ready tasks are placed in a dynamic linked list and a searching engine is built upon this list.

From the searching algorithms sketched in the previous section I prefered to take balanced tree (AVL, Red-Black) as searching engine for the following reasons:

- as for insert operation, its computational complexity is $O(log_2 n)$, the same as that of other algorithms

- not complicated data structure, easy to implement, do not require additional temporary data structures

- can be futher optimized for special cases as it is described in the next section

While integrating EDF scheduler into RTEMS supercore, I tried to make as few changes as possible and preserve its functionality. After applying the EDF scheduler patch on standard RTEMS core (see Appendix D) the user has a choice to build it with either default priority scheduler or the EDF scheduler. If the EDF scheduling is chosen, then an application developer must be careful of real arguments the directives like rtems_task_create are called with. This will be discussed later in the EDF interface section. I do not consider a combined scheduling of priority and EDF tasks as an advance.

The EDF scheduler implementation consists of edf_types.h, edf.h, rbtree.h header files situated in the directory RTEMS_ROOT/cpukit/score/include/rtems/score/, rbtree.c and edf.inl[3] C files in the RTEMS_ROOT/cpukit/score/src and cpukit/score/inline/rtems/score directory respectively.

6.3.1 The EDF ready queue

There are many discussions on the Internet about which one of the two balanced trees AVL and Red-Black (see [11],[12],[13],[14] for more details about Red-Black tree) is better as far as in both cases operations look-up, insertion, and deletion have the same complexity $O(log_2 n)$. The Red-black tree has height at most $2log_2(n+1)$ whereas AVL at most $1.44log_2(n+1)$. In fact, AVL tree is a special, more balanced case of Red-black tree. However, this truth does not implicate that AVL is the more efficient one because their efficiency depends on the character of input data and frequency of particular operations. I favoured Red-Black tree since in most real-time systems events (like finishing of executing task, releasing a new task) happen periodically within a few miliseconds and thus it is not really necessary to have a perfectly balanced tree every time. This conclusion ought to be confirmed by experiments but it is beyond of this work.

[3]The .inl sufix denotes a file which defines inline functions.

In the first approach, EDF ready queue is considered as a complex data structure consisting of double-linked list of ready tasks (TCBs) and searching Red-Black tree[4]. A node of tree is a special data structure comprising pointer to associated TCB in the list.

In order to reduce the memory requirements and simplify operations with the queue, besides being an element of the double-linked list, every TCB is a node of tree as well. Then TCB has to include mandatory fields of tree node control structure to facilitate building of tree. Figure 6.2 depicts the described EDF ready queue (the number illustrated inside of node is task's absolute deadline).

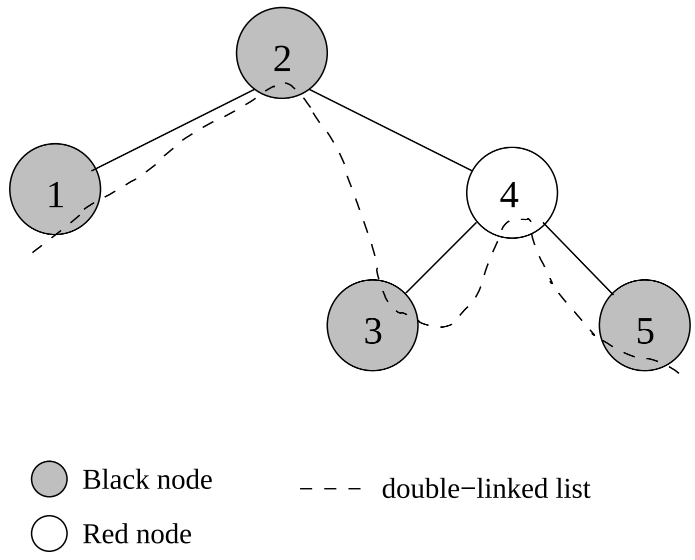


Figure 6.2: The EDF ready queue

When a new task is being inserted into the ready queue, at first it is effectively included (with complexity $O(log_2 n)$) according to its absolute deadline in an appropriate place in pre-ordered Red-Black tree. After its precedor or successor in the tree is found the task is inserted into the double-linked list. The list must be double-linked to simplify the insertion of a new TCB everywhere into the list. The task with the earliest deadline is always the first in the list. Searching for that task via tree would not be efficient. Deleting that task is trivial. The proposed EDF ready queue satisfies all features that ready queue should have.

The EDF ready queue can be futher optimized by omitting the double linked list. The reason is that we always need to know only the task with the lowest absolute deadline. Therefore it is sufficient to maintain a pointer to that task and update it whenever a new task is being inserted into the queue or the executing task terminates. Due to Red-Black tree properties this pointer called first can be updated in a simple way:

- *Insert operation* Firstly, a parent node of the incomming node (new_node) is found and while inserting the new_node into tree, the first pointer is appropriately updated:

```
if parent is not null then
    if  new_node's absolute deadline < parent's absolute deadline then
```

[4]Red-Black tree is futher called just tree for short.

```
            new_node is parent's left child
            if parent is first in queue  then new_node becomes first;
          else new_node is parent's right child
      else queue is empty
          node becoms the root of tree and first task
```

- *Delete operation* Before extracting a node (del_node) from the ready queue the first pointer is appropriately updated:

```
      if del_node is first and hence first must be updated then
         if first has right child then
         its right child becomes first
         else first's parent becomes first
```

This optimalization eliminates the need of double-linked list to keep the execution order of tasks. Thereupon it reduces memory consumption by the additional the next and the previous pointers in TCB that otherwise would be needed to build double-linked list [5] and simplifies insert and delete operations since complicated pointer updates of list's elements can be omitted.

Finally, TCB is required to include the following additional fields:

- abs_deadline - task's absolute deadline
- rel_deadline - task's relative deadline
- left,right, parent - task's left, right child and parent
- color - node's color (red or black)

The additional fields could be reduced by proclaiming the Object.Node.next to be the right field and the Object.Node.previous to be the left field. For the code clarity I did not utilized the Object structure fields for this purpose in the first EDF scheduler version.

6.3.2 edf_types.h

This header file contains definitions of basic data types for the EDF queue implementation.

```
typedef struct Thread_Control_struct EDF_Node;

typedef struct {
        EDF_Node   *root;
        EDF_Node   *first;
} EDF_Chain_Control;
```

The type EDF_Node is alias for TCB. The EDF queue is represented by the structure EDF_Chain_Control having two fields root and first. The root field is a pointer to the root node of Red-Black tree and the first is a pointer to actual executing task or task with the lowest deadline i.e. the first task in the queue.

The system has one EDF ready queue represented by _Thread_Ready_EDF_chain variable defined in the file RTEMS_ROOT/cpukit/score/src/thread.c as variable of EDF_Chain_Control type.

[5]This is ,however, not fully true, TCBs can be chained via RTEMS chain.

6.3.2.1 rbtree.c and rbtree.h

The file rbtree.c contains a code of the following directives:

- void _RBT_Insert(EDF_Chain_Control *chain,EDF_Node *node)
  Inserts TCB into RB tree.
- void _RBT_Extract(EDF_Chain_Control *chain,EDF_Node *node)
  Removes TCB from RB tree.
- boolean _RBT_Has_only_one_node(EDF_Chain_Control *chain)
  Returns TRUE if there is only one TCB in RB tree.

In a call of directives, the node parameter represents TCB and the chain parameter is a control structure involving a pointer to the root node of RB tree.

The implementation details of insert and delete operations on RB tree can be found in [13], [14]. The most of RB tree implementations require so that a null node is defined which serves as a sentinel that a leaf node have been reached. However, null nodes mean memory wastage and bring difficulty since every change of the Thread_Control_struct structure (TCB) would require a modification of the null node definition. For that reason, I prefered to implement modified version of insert and delete code not demanding a null node and ,moreover, extended by a code to update a pointer to the first TCB in the ready queue.

Simple scheme of insert operation:

```
_RBT_Insert(chain, node)
 travers tree from chain.root to find place(parent) for node in tree
 include node into tree and update the first pointer appropriately
 fixup tree properties if violated
```

Simple scheme of delete operation:

```
_RBT_Extract(chain, node)
 update the first pointer
  (find node's succesor and
   exchange it with node - this is not needed when
   deleting the first node in order)
 fixup tree properties if violated
```

6.3.3 The EDF interface

The EDF interface provides directives called directly within RTEMS core . The directives wrap concrete directives manipulating with the queue. Therefore RTEMS code is independent of the real EDF queue implementation that developer can replace with its own one he thinks is better. This architecture brings flexibility and no subsequent changes in the RTEMS core code.

The EDF interface consists of the following inline directives found in file RTEMS_ROOT/ cpukit/score/inline/rtems/score/edf.inl that directly calls functions in file RTEMS_ROOT/cpukit /score/rbtree.c:

- void _EDF_Chain_Insert(EDF_Chain_Control *chain, EDF_Node *node)
  Adjusts task's absolute deadline and inserts TCB into the EDF queue (chain) by calling _RBT_Insert directive. Absolute deadline is calculated as

the number of ticks since RTEMS boot (read from _Watchdog_Ticks_since_boot variable) + task's relative deadline

- void _EDF_Chain_Insert_Preserve_Abs_Deadline (EDF_Chain_Control *chain, EDF_Node *node)
  Inserts TCB into the EDF queue (chain) but does not adjust task's absolute deadline.
- void _EDF_Chain_Extract (EDF_Chain_Control *chain, EDF_Node *node)
  Removes node from the EDF queue (chain) by calling _RBT_Extract directive. [6]
- boolean _EDF_Chain_Has_only_one_node (EDF_Chain_Control *chain)
  Returns TRUE if there is only one TCB in the EDF queue (chain). Calls directly _RBT_Has_only_one_node directive.

There are two approaches how to activate the EDF interface calls in RTEMS core when a programmer chooses to use the EDF scheduling instead of the native priority scheduling:

1. Extend the Configuration table by parameter say sched_policy. Then the programmer can choose the type of scheduler per application basis by using macro at the beginning of application to set the value of sched_policy parameter. Depending on this value appropriate scheduler calls are activated in RTEMS code.
2. The same as above except that preprocessor macros are used. If defined the preprocessor macro EDF is defined (see /cpukit/score/include/rtems/score/edf.h) then the EDF sheduler calls are activated. Since only a selected scheduler's code is compiled during building RTEMS no conditional branching is required. Note that this is not at the expense of flexibility because a programmner may have several RTEMS cores with different options compiled.

For simplicity, I have chosen the second approach in the current scheduler version.

6.3.4 The EDF scheduler integration in RTEMS core

By now only RTEMS API supports EDF scheduling. However, it is not difficult to customize other APIs for EDF.

The only thing a programmer must remember when using the EDF scheduling is that in the call of rtems_task_create[7] relative deadline is substituted for task's priority. I do not consider a declaration of special EDF task creation directive for this purpose as a convenient solution.

The list of EDF all changes in RTEMS supercore directives: [8]

- _Thread_Initialize directive
  Passing relative deadline as the parameter instead of priority.
- _Thread_Clear_state, _Thread_Ready directives
  Depending on the state of the EDF macro (defined or undefined), TCB is inserted into the EDF or priority queue and context switch flag is updated appropriately.

[6]Since only the first node is always being removed from the chain and its pointer is included in EDF_Chain_Control structure, the node parameter in this directive call will be omitted in future version of EDF sheduler.

[7]The calls of EDF scheduler are implemented only in RTEMS API. However, it is not difficult to customize other APIs for EDF as well.

[8]RTEMS core directives are situated in RTEMS_ROOT/cpukit/score/src and RTEMS_ROOT/cpukit/score /inline/rtems/score/ directories, The names of directives are similar to files in which they are defined.

- _Thread_Create_idle directive
  Idle [9] task's priority is set to maximum priority i.e. 255 by default. EDF idle task's absolute deadline is set to maximum absolute deadline[10]. Idle task is usually created and inserted into the ready queue at RTEMS inicialization.

- _Thread_Set_priority directive
  Does nothing if EDF macro is defined.

- _Thread_Calculate_heir directive[11]
  If the EDF macro is defined, it sets the _Thread_Heir variable to the first TCB in the EDF ready queue otherwise in the priority ready queue.

- _Thread_Set_state, _Thread_Set_transient
  _Thread_Suspend directives
  If a task is ready, depending on the state of the EDF macro (defined or undefined), the task is removed from the EDF or the priority ready queue and context switch flag is updated appropriately.

- _Thread_Yield_processor directive
  If the EDF macro is defined a task is removed from and subsequently inserted back into the EDF ready queue without the change of its absolute deadline. Otherwise a task is removed from and subsequently inserted back into the priority ready queue. In both cases context switch flag is updated appropriately.

The file RTEMS_ROOT/rtems/cpukit/score/Makefile.am is a makefile for RTEMS supercore. Source and header files of EDF scheduler have to be added at the beginning of the value of THREAD_C_FILES and STD_H_FILES variable respectively to be part of the RTEMS build system.

6.4 Testing

The functionality of EDF scheduler can be demonstrated on an example of two periodic tasks. The EDF and fixed priority scheduler will produce different schedules of tasks as depicted, for example, in Figure 3.1.

For this purpose, I have developed RTEMS application that periodically releases instances of two tasks. This can be achieved by rate monotonic manager (see chapter 5). The time of start and finish of every instance is sent to the output. After the application is launched its initialization task creates and then starts two tasks Task_1 and Task_2 with almost the same body. The body of Task_1:

```
name = rtems_build_name( 'P', 'E', 'R', 'A' );
status = rtems_rate_monotonic_create( name, &period );
if ( status != RTEMS_SUCCESSFUL ) {
   printf( "rtems_monotonic_create failed with status of %d.\n", status);
   exit( 1 );
}
```

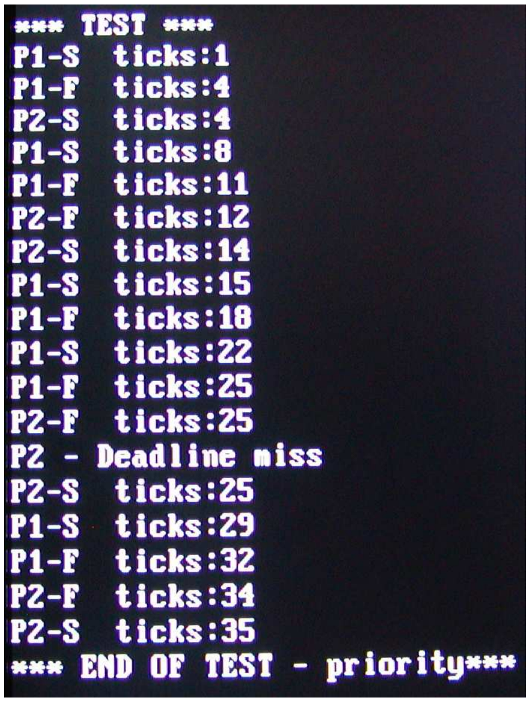
[9]Idle task is a task or more precisely active loop that runs on processor when there is no task in ready queue
[10]Since RTEMS time counter is 32-bit, maximum absolute deadline has the value 0xfffffff.
[11]_Thread_Calculate_heir directive is an inline directive defined in the file RTEMS_ROOT/cpukit/score /inline/rtems/score/thread.inl.

```
period_length = 7;
max_i = 10000; max_j = 4000;
while ( 1 ) {
   if (rtems_rate_monotonic_period(period,period_length)==RTEMS_TIMEOUT)
        puts("P1 - Deadline miss");

   rtems_clock_get(RTEMS_CLOCK_GET_TICKS_SINCE_BOOT, &start);
   sprintf(output1,"P1-S  ticks:%d",start);
   puts(output1);
   if ( start >= 30 ) break;  /* stop */
   /* active computing */
   for ( i = 1 ; i < max_i; i++)
   {
       j =  i/12 ; j++;
       for ( j = 1; j < max_j; j++)
       {
       k = j/ 3;
       k++;
       }
   }

   rtems_clock_get(RTEMS_CLOCK_GET_TICKS_SINCE_BOOT, &stop);
   sprintf(output2,"P1-F  ticks:%d",stop);
   puts(output2);
}

/* missed period so delete period and SELF */
status = rtems_rate_monotonic_delete( period );
if ( status != RTEMS_SUCCESSFUL )
{
 printf("rtems_rate_monotonic_delete failed with status of %d.\n",status);
 rtems_test_exit( 0 );
}
puts( "*** END OF TEST ***" );
rtems_test_exit( 0 );
}
```

The task Task_1 creates rate monotonic period called PERA with length $period\_length = 7$ ticks and parameters $max_i = 10000$, $max_j = 4000$. The task Task_2 creates rate monotonic period PERB with length $period\_length = 10$ ticks and parameters $max_i = 10000$, $max_j = 6000$. Parameters max_i and max_j determine loop count of outer and inner cycle of the active computation and are experimentally designated so that the computation time of Task_1 and Task_2 was 3 ticks and 5 ticks respectively. The duration of one tick is set to 10 miliseconds.

The application is named spedf.exe[12]can be found in directory RTEMS_BUILD/i386-rtems/pc386/tests after the building of RTEMS and its source files are situated in the directory RTEMS_ROOT/c/src/tests/sptests/spedf. Firstly, I have compiled the application with standard priority kernel and renamed the executive file to spedfp1.exe. Then, without any changes in the source code, the spedfe1.exe application was built with the active EDF scheduling. It was

[12]Do not be confused with .exe file suffix, it is not MSDOS executive format.



```
*** TEST ***
P1-S  ticks:1
P1-F  ticks:4
P2-S  ticks:4
P1-S  ticks:8
P1-F  ticks:11
P2-F  ticks:12
P2-S  ticks:14
P1-S  ticks:15
P1-F  ticks:18
P1-S  ticks:22
P1-F  ticks:25
P2-F  ticks:25
P2 - Deadline miss
P2-S  ticks:25
P1-S  ticks:29
P1-F  ticks:32
P2-F  ticks:34
P2-S  ticks:35
*** END OF TEST - priority***
```

Figure 6.3: The output of spedfp1.exe application

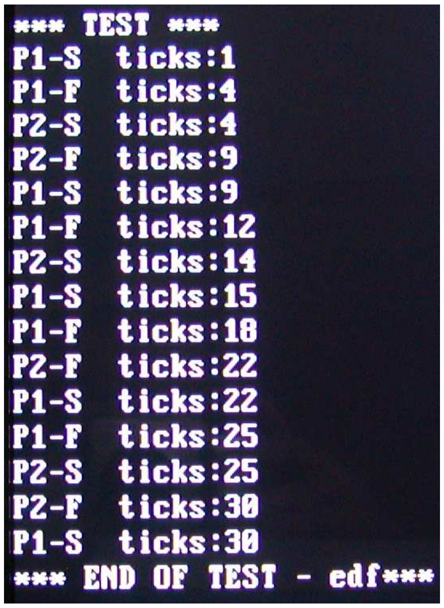
necessary to launch applications immediately after a computer restart since in PC emulator ticks values were not correct. The test applications were running on a i686 Intel Celeron 1,8GHz and sent all output to screen. See appendix C how to build and run RTEMS application.

The output of the application `spedfp1.exe` is depicted in Figure 6.3. Each output line starts with the letter P (like process) followed by task id, then follows either the letter S or F , which is an abbreviation for the start or the finish of job, and the ticks value denotes the number of ticks left since boot. Note that the Task_1 does not start at tick zero since some initial routines had been done before and the Task_2's start is postponed to the finish of first Task_1's instance. However, these phase shifts does not impact the test correctness.

From the output (Figure 6.3) of the application `spedfp1.exe` with fixed priority scheduling, it can be seen that the second instance of Task_2 starts at the time 14. Runs for 1 tick and then is preempted by higher priority job of Task_1 [13] Another opportunity to run has the Task_2's job at ticks 19,20,21 and 25 when it terminates. However, the job missed its deadline at time 24 (the start at 14 + 10 ticks period = 24). The schedule correspondes to theoretical presumptions.

Figure 6.4 illustrates the output of the `spedfe1.exe` application with EDF scheduling. The second instance of Task_2 starts at the time 14 with absolute deadline at time 24. Runs for 1 tick and then is preempted by the job of Task_1 which has earlier absolute deadline at time 22. After Task_1's job is finished at time 18, Task_2's job has earlier deadline than the next instance of Task_1 and is resumed for that reason. At time 22 the instance of Task_2 terminates before its deadline. There is no deadline miss.

[13]Task_1 has shorter period than Task_2 hence higher priority in RMS.



```
*** TEST ***
P1-S  ticks:1
P1-F  ticks:4
P2-S  ticks:4
P2-F  ticks:9
P1-S  ticks:9
P1-F  ticks:12
P2-S  ticks:14
P1-S  ticks:15
P1-F  ticks:18
P2-F  ticks:22
P1-S  ticks:22
P1-F  ticks:25
P2-S  ticks:25
P2-F  ticks:30
P1-S  ticks:30
*** END OF TEST - edf***
```

Figure 6.4: The output of spedfe1.exe application

In both cases, the task schedules correspond to theoretical presumptions. The correct functionality of EDF scheduler has been confirmed. Note that the tick values in the output of both applications can be different on various machines, since active computing may not take the same time as on the test machine. If the duration of tick is changed, it results in different output as well. However, the task scheduling is correct in any case.

7 The Resource Access Protocol Implementation

Resource Access Protocols (RAPs) are presented in chapter 4 from theoretical point of view. This chapter focus on the implementation of RAPs in RTEMS and gives proposals of their improvement.

7.1 RTEMS semaphores

RTEMS provides the synchronization tool - semaphore to control access to shared resources. The following types of semaphores are supported:

- binary semaphore
  Restricts values to 0(locked) and 1(unlocked - default). It is used to control access to exclusive resource. Allows nested access i.e. semaphore can be locked multiple times by the same semaphore holder.

- simple semaphore[1]
  Restricts values to 0(locked) and 1(unlocked - default). It is used to control access to exclusive resource. Does not allow nested access.

- counting semaphore
  It is used to control access to multiunit resource therefore it can acquire values zero to number of resource units.

RAPs discussed in this work relate to exclusive resources guarded by binary semaphore. RTEMS supports PIP,PCP for local, binary semaphores that use the priority task wait queue blocking discipline. Therefore I focus on this type of semaphore in the following explanation.

Structure *Semaphore_Control*[2] defines the control block used to manage each semaphore.

```
typedef struct {
        Objects_Control          Object;
        rtems_attribute          attribute_set;
        union {
          CORE_mutex_Control     mutex;
          CORE_semaphore_Control semaphore;
        } Core_control;
}   Semaphore_Control;
```

The *attribute_set* holds information about the character of a given semaphore (counting or binary, RAP, etc.). When using binary or simple semaphore the *mutex* field of *Core_control* is accessed. Structure *CORE_mutex_Control*[3] defines the control block used to manage each mutex.

```
typedef struct {
        Thread_queue_Control    Wait_queue;
        CORE_mutex_Attributes   Attributes;
        unsigned32              lock;
        unsigned32              nest_count;
```

[1]Both binary or simple semaphore is often called mutex.
[2]Structure *Semaphore_Control* is defined in the file RTEMS_ROOT/cpukit/rtems/include/rtems/rtems/sem.h
[3]Structure *CORE_mutex_Control* is defined in file RTEMS_ROOT/cpukit/score/include/rtems/score/coremutex.h

```
        unsigned32              blocked_count;
        Thread_Control         *holder;
        Objects_Id              holder_id;
}   CORE_mutex_Control;
```

Structure fields have the following meaning:

- Wait_queue
  Tasks blocked on semaphore wait for free resource in a waiting queue represented by structure *Thread_queue_Control*. Blocked tasks can be enqueued by FIFO or priority order.

- Attributes
  The field holds information about the priority ceiling of mutex, the waiting queue discipline, nesting behaviour, the permisission to release mutex by other task than owner.

- lock
  A value 0 indicates that the mutex is locked, whereas the value 1 that it is unlocked.

- nesting_count
  Indicates the number of nested locks.

- blocked_count
  Holds the number of tasks blocked on this mutex.

- holder, holder_id
  TCB and object ID of the task that has locked the mutex.

Semaphore's waiting queue is implemented by the universal task queue depicted in Figure 7.1. The structure *Thread_queue_Control*[4] defines the control block used to manage that queue.

```
typedef struct {
  union {
     Chain_Control Fifo;
     Chain_Control Priority[TASK_QUEUE_DATA_NUMBER_OF_PRIORITY_HEADERS];
  } Queues;
  Thread_queue_States      sync_state; /* alloc/dealloc critical section */
  Thread_queue_Disciplines discipline; /* queue discipline              */
  States_Control           state;      /* state of threads on Thread_q   */
  unsigned32               timeout_status;
}   Thread_queue_Control;
```

Note that two enqueueing disciplines are defined: fifo and priority discipline. To support PCP, PIP, the priority discipline must be used.

The priority wait queue consists of four chains used to maintain tasks by their priority. Each chain manages tasks of priorities from the given range. The number of chains equals to the number of headers and is determined by TASK_QUEUE_DATA_NUMBER_OF_PRIORITY_HEADERS macro, that is four by default. When considering 256 priority levels and 4 headers then each header includes 64 priorities. Header x manages priority range $(x * 64)$ through $((x * 64) + 63)$.

[4]Structure *Thread_queue_Control* is defined in file RTEMS_ROOT/cpukit/score/include/rtems/score/tqdata.h.

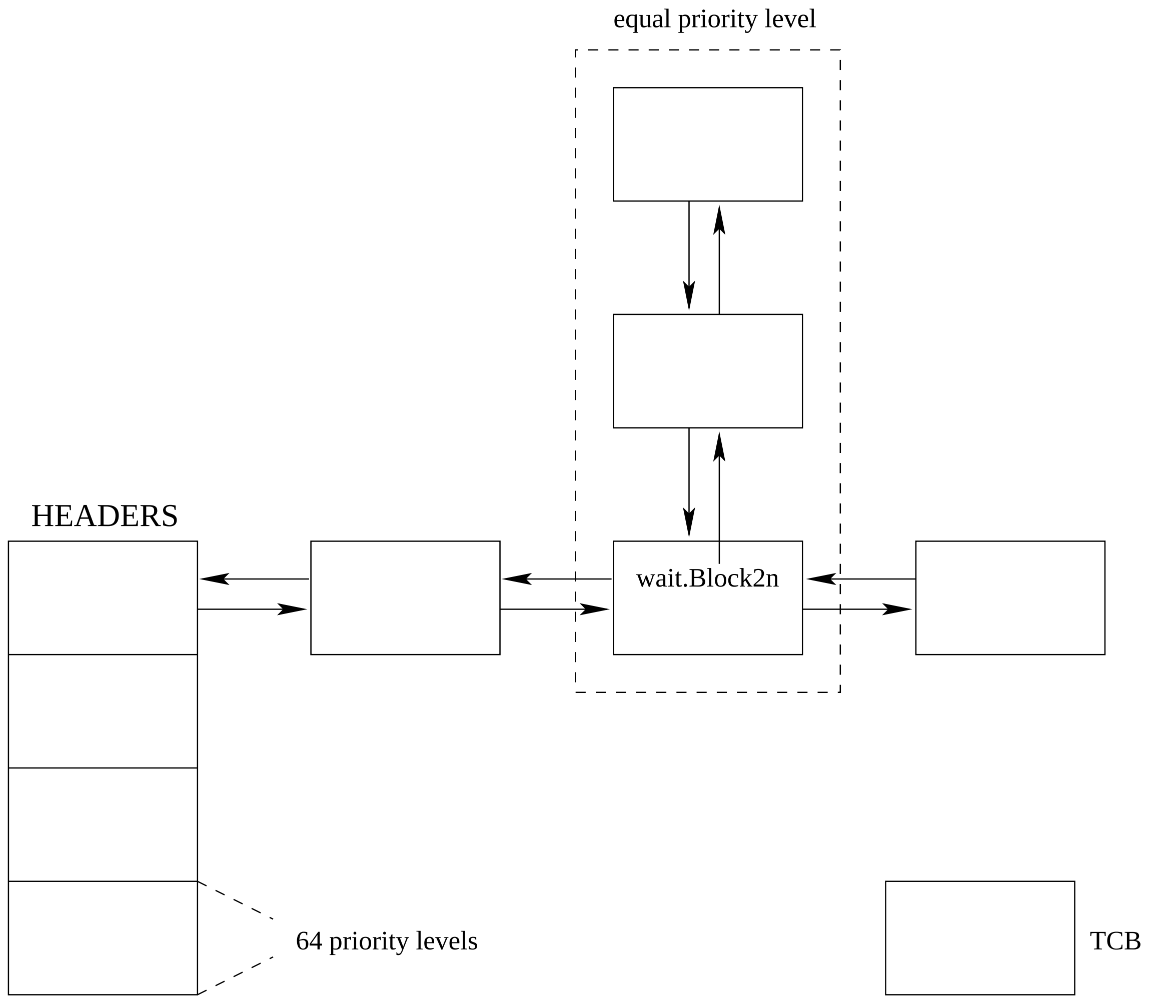


Figure 7.1: The task queue

A newly blocked task is assigned an header that indicates the FIFO queue into which then the task should be inserted. Before enqueuing a proper place must be found for that task in the queue to preserve its ordering by the priority. If the task's priority is more than half way through the priority range it is in, then the search is performed from the rear of the chain. This halves the search time to find the insertion point. If there is already a task in the chain with the same priority a newly blocked task is inserted into FIFO chain defined by the Wait.Block2n field in TCB of that task. This futher reduces searching time since tasks with equal priorities can be simply skipped. The Wait.queue field in TCB refers to the mutex waiting queue in that task is situated.

7.2 RTEMS RAPs implementation

RTEMS implements both PCP and PIP. There are minimal differences in their implementation and those are stated in the next explanation.

RTEMS supports PIP,PCP for local, binary semaphores that use the priority task wait queue blocking discipline. Binary semaphore is created by calling rtems_semaphore_create(name, 1, RTEMS_PRIORITY| RTEMS_BINARY_SEMAPHORE|RTEMS_PRIORITY_CEILING, 10, id), where name is the name of mutex, followed by the number of resource units, attributes, the priority ceiling of mutex and semaphore identifier returned in id. The semaphore can be deleted by calling rtems_semaphore_delete(id) directive.

When the executing task tries to acquire a mutex by calling rtems_semaphore_obtain directive one of the following situations can happen:

1. Mutex is unlocked
   - The executing task locks the mutex, the holder and the holder_id fields of mutex structure are set to that task, the nest_count field to 1 and the resource_count field in TCB is increased.
   - In case of PCP RAP, task's priority ,if lower, is raised to the ceiling of mutex, otherwise it is evaluated as error if task's priority is higher than ceiling.
2. Mutex is locked
   - If the executing task is a holder of mutex and nested behaviour is allowed then the nest_count is increased. If nested behaviour is not allowed it is evaluated as error.
   - If RTEMS_WAIT option was specified in the call of directive then the executing task is blocked and inserted into the waiting queue of mutex. The blocked_count field of mutex structure is increased, the Wait.queue field of TCB is set to the waiting queue where the task waits for the mutex to become available. In case of PIP RAP, the mutex holder's priority, if lower, is raised to the executing task's priority. In case of PCP RAP, the executing task's priority ,if lower, is raised to the ceiling of mutex.

When the executing task releases held mutex by calling rtems_semaphore_release directive:

- the nest_count, the resource_count and the blocked_count fields are decreased, the holder and the holder_id fields are zeroed ;
- if it is the task's last mutex (the resource_count field from TCB of the task is zero), its priority is changed to nominal priority;
- after the mutex is released, the first task , if any, in the waiting queue of the mutex is dequeued, obtains the mutex and is inserted back into the ready queue.

From the previous explanation it is obvious, that PIP and PCP in RTEMS do not match the theory represented in chapter 4. According to the basic PCP theory when a task becomes blocked on mutex, the mutex holder inherits its priority if higher. However,in case of RTEMS PCP implementation, task's priority is raised to the ceiling immediately as the task acquires the mutex. Hence RTEMS PCP can not be the basic PCP. Even it is not the stacked-based PCP since a task is not prevented from starting if there is a chance to be blocked on resource.

When examining the source code of RTEMS RAPs I found another difference related to the change of task's priority after releasing mutex. To be sure, I have developed the application (spmutextest) which creates a task that acquires and subsequently releases mutexes with middle and high ceiling. Firstly, task obtains mutex with the middle priority ceiling and its priority is changed from low to middle level and later to high level when it locks mutex with the high ceiling. After releasing the lastly locked mutex task's priority is not changed to the middle priority as it would be theoretically expected. However, task's priority is altered only when the task releases all of the binary semaphores it holds. Then its priority is directly restored to the normal value. Before that it remains at the high priority even if task is not already in a critical section with high priority ceiling.

This behaviour seems to be a RTEMS PIP, PCP design flaw. Therefore, I sent a report about it to RTEMS mailing list. Another RTEMS developer confirmed this behaviour that he considers a design flaw as well. Then a response came from Joel Sherill, the RTEMS maintainer: "This is the designed and documented behavior. To implement otherwise would require maintaining information on the set of held resources and analyzing that set each time a mutex is released. The current design is an engineering tradeoff assuming that programmer do not nest lots of PCP/PIP mutexes and do not hold them for lengthy periods of time.

I recently thought about this and think the "resources held set" could be cheaply maintained but how you determine the appropriate priority when you release a mutex is still painful. Even if you maintain the set as a stack/LIFO and imposed and enforced the rule that PCP/PIP mutexes were released in the opposite order of acquisition, I still think you would have to scan the set when you released a mutex to find out what the highest priority to inherit is.

Any thoughts on an efficient way to implement the set management required is appreciated. I have trouble seeing it as something that is constant order execution time."

7.3 New RTEMS RAP designs

7.3.1 Improvements of existing RTEMS RAPs

Some developers argue that it is easy to imagine applications that violate the previous assumption that programmer do not nest lots of PCP/PIP mutexes and do not hold them for lengthy periods of time. However, the correct PIP,PCP imlementations meet the difficulty how to effectively determine task's current priority when the task releases a mutex.

Considering PCP, task's priority is equal to the highest ceiling of the mutexes held by task. There are two possible approaches how to effectively determine task's priority.

The first approach is to maintain an ordered resource (mutex) set for each task. Then a ceiling of the first mutex in the set is task's current priority. The complexity of such structure is $O(log_n)$ (balanced tree) or $O(n)$ (linear linked-list).

The second approach could be to enforce the similar mutex release rule like in OSEK operating system that PCP mutexes are released in the opposite order of acquisition. For each task the resource held set with stack access policy is maintained. Resource set is implemented as dynamic list(chain) of held mutexes. The element of the list - mutex control structure is extened by field say release_priority which holds task's priority after releasing that mutex.

For better explanation, consider the following example depicted in Figure 7.2 :

- Task's normal priority is 9.
- Task acquires mutex with the ceiling equal to 4. Then, put that mutex's control structure with release priority equalling to task's current priority on the stack and change task's current priority to 4.
- Task acquires mutex with the ceiling equal to 8. Then, put that mutex's control structure with release priority equalling to task's current priority on the stack. Task's current priority is not changed because it is higher than 8.

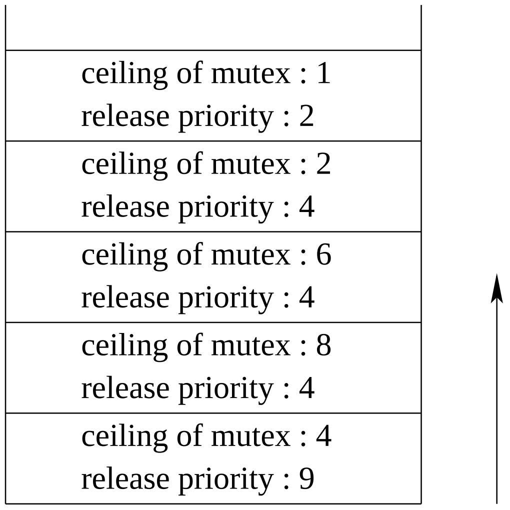


Figure 7.2: A stack of mutexes

It‘s obvious how to release mutexes and change task‘s priority. At first, according to the mutex release rule the mutex with the ceiling equal to 1 is removed from the top of stack and task’s priority is changed to 2. The attempt to release other mutex is evaluated as error.

This proposal was refused because of restriction accompanied by the mutex release rule. RTEMS developers appealed to a scenario where small critical section is protected by a ”fine-grained” mutex. Part of the critical code section is acquisition of a ’coarse’ mutex:

```
lock(fine_grained)
    do_critical_work()
    lock(coarse)
unlock(fine_grained)
do_coarsely_protected_work()
lock(fine_grained)
    unlock(coarse)
    do_critical_cleanup()
unlock(fine_grained)
```

However, I think it is possible to write/rewrite programs that obey the above mentioned mutex realese rule.

It is hard to argue which approach should be taken. Since I do not expect that task would hold tens of mutexes, I prefer the first approach to avoid the restriction the second approach have. For each task, an ordered mutex set, implemented as linked-list, is maintained. The space and time complexity of that approach is $O(n)$ which is acceptable for low n. For larger number of mutexes, the balanced tree (see Figure 6.2) or a structure similar to the task queue (see Figure 7.1) could be taken into account.

7.3.2 RAP implementation for EDF

RAPs for dynamic priority systems are Preemption Ceiling Protocol, Dynamic Priority Ceiling Protocol and Dynamic Priority Inheritance Protocol. Dynamic versions of PCP is not efficient nor simple to implement since inherited priorities, the priority ceilings of each resources and the system ceiling must be updated each time task priorities change. The similar drawback brings DPIP.

Basic PreCP can be simply implemented by utilizing the existing basic PCP implementation when task's preemption levels are used instead of their priorities. Because of distinct basic PCP implementation in RTEMS, basic PreCP must be implemented from scratch or existing RTEMS PCP modified. In fact, both cases have the same complexity.

Another way is to implement stacked-based PreCP that applies Stack Resource Policy proposed by Baker[5] for EDF scheduling especially. Such implementation design is described in [15]. However, I think it has increased memory requirements and its time complexity is two times higher than stated in the article. The efficient implementation should be still considered.

One of SRP properties is early blocking, it means that the execution of a job is delayed instead of being blocked while requesting shared resources. Early blocking and selecting the highest-priority job among the jobs that will not be blocked is in the comptence of scheduler. It is important to note that for that reason, the design and implementation of scheduler and SRP are close-knit and can not be proceeded separately like in the case of other protocols.

In general, efficient stacked-based PreCP implementation seems to be a big challenge.

8 Conclusion

To make the EDF scheduler implementation and resources access protocols better to understand, I explained, at the beginning of this work, the basic real-time concepts, scheduling algorithms and gave a comprehensive look at resource access protocols.

Giorgio C. Buttazzo in [1] compares EDF scheduling to RM and refutes presumptions that RM is easier to analyze than EDF, it introduces less runtime overhead, it is more predictable in overload conditions, and causes less jitter in task execution. His results are noteworthy and hence I presented them briefly in my work. Buttazzo's work motivated me to implement EDF scheduling into RTEMS.

RTEMS is a well-designed, open source real-time operating system which lacks of EDF scheduler. In chapter 5, I focused on RTEMS, its feature, structure, internals (the ready queue, clock manager, rate monotonic manager, etc.), partly RTEMS API. The most of these and the information in the appendix can not be found in official RTEMS documentation and are important to understand EDF scheduler implementation or building applications.

RTEMS has a priority based kernel. I analyzed EDF scheduler implementation designs and decided not to take a standard approach that means to implement EDF scheduling algorithm by utilizing already existing priority ready queue. I prefered to implement special EDF queue from scratch and so have free hands to design and optimize it. I showed that EDF scheduler integration brings only few changes into original RTEMS source code. Therefore this approach is acceptable in priority based kernels. A developer can activate EDF scheduling or use priority scheduling by default. A remarkable solution in this way is provided by S.Ha.R.K real-time system, which is a dynamic configurable kernel architecture designed for supporting hard, soft, and non real-time applications with interchangeable scheduling algorithms. S.Ha.R.K has a modular interface for the specification of scheduling algorithms realized like modules.

From several proposals of EDF ready queue I decided to implement it as Red-Black tree with tasks being directly tree nodes and ordered by asceding absolute deadline. This structure facilitates to find a proper place in the queue for new task with complexity $O(log_2 n)$, simple enqueuing and dequeuing of task and the effective selection of the next task to be run. I futher optimized ready queue structure due to the fact that every time only the task with the lowest deadline must be known. The correct functionality of EDF scheduler was confirmed by the application I developed for this purpose. The test was performed on the set of periodic tasks. RTEMS provides the mechanism for periodic task releasing and hence I did not have to implement it.

The EDF scheduling requests for especial resource access protocol and what's more, one of them, SRP, must be designed together with scheduler. When designing that RAP I found out that existing RTEMS RAPs for priority scheduling cannot be modified and used for the EDF case since their implementation does not match the theory. For that reason I only explained RTEMS RAPs implementation, their shortcomings and proposed the solution. The implementation of RAP for EDF scheduling is the future work and should be cooperated with RTEMS developers who have large experience with applications in a real enviroment and specific requirements for algorithm efficiency.

9 References

[1] Giorgio C. Buttazzo. Rate monotonic vs. edf: Judgment day. *Real-Time Systems*, 29(1):5–26, 2005.

[2] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 1997.

[3] A.Cervin. Integrated control and real-time scheduling. *Doctoral Dissertation, ISRN LUTFD2/TFRT-1065-SE, Department of Automatic Control, Lund.*, 2003.

[4] Jane W.S. Liu. *Hard Real-Time Computing Systems*. Prentice Hall, 2000.

[5] T. P. Baker. Stack-based scheduling of realtime processes. *Journal of Real-Time Systems*, pages 67–99, 1991.

[6] RTEMS Homepage. `http://www.rtems.org`.

[7] RTEMS documentation sets - RTEMS Applications C User's Guide. `http://www.rtems.org/onlinedocs.html`.

[8] Patricia Balbastre and Ismael Ripoll. Integrated Dynamic Priority Scheduler for RTLinux. *Department of Computer Engineering (DISCA),Universidad Polit ´cnica de Valencia*, 2001.

[9] Gerth S. Brodal. Worst-case efficient priority queues. *Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 52–58, 1996.

[10] R. Sridhar K. Rajasekar and C. Pandu Rangan. Probabilistic data structures for priority queues. *Proceedings of the 6th Scandinavian Workshop on Algorithm Theory*, pages 143–154, 1998.

[11] Red-black trees. `http://sage.mc.yu.edu/kbeen/teaching/algorithms/resources/red-black-tree.html`.

[12] Data structures and algorithms: Red-black trees. `http://www.cs.auckland.ac.nz/software/AlgAnim/red_black.html`.

[13] Youjip Won. Red black tree. `cdrom://red-black_tree_docs/Won-rbtree.pdf`, 2005.

[14] Margarida Jacome. Red black Tree. `cdrom://red-black_tree_docs/Jacome-rbtree.pdf`, 2005.

[15] Sangchul Han Moonju Park Yookun Cho. An efficient job selection scheme in real-time scheduling under the stack resource policy. *The 11th International Workshop on Parallel and Distributed Real-Time Systems*, 2003.

[16] RTEMS Documentation Sets - Getting started with RTEMS. `http://www.rtems.org/onlinedocs.html`.

[17] RTEMS FAQ. `http://www.rtems.org/onlinedocs.html`.

[18] BOCHS. `http://bochs.sourceforge.net`.

[19] Bochs - creating hard disk image. `http://www.rtems.com/phpwiki/index.php/Bochs#Hard_Disk_Image`.

A List of abbreviation

ARJ Average Response Jitter

DPCP Dynamic Priority Ceiling Protocol

DPIP Dynamic Priority Inheritance Protocol

EDF Earliest Deadline First

FPS Fixed Priority Scheduling

OID Object Identifier

PCP Priority Ceiling Protocol

PDC Processor Demand Criterion

PIP Priority Inheritance Protocol

RAP Resource Access Protocol

RBT Red-Black Tree

RMS Rate Monotonic Scheduling

RM Rate Monotonic

RRJ Relative Response Jitter

RTA Response Time Analysis

RTOS Real-Time Operating System

RTEMS Real-Time Operating System for Multiprocessor Systems

SRP Stack Resource Policy

TCB Task Control Block

B Building RTEMS

The information about building RTEMS can be also found in [16] and [17] but I would like to share my experiences.

The following text describes steps how to build RTEMS from sources for the i386 target platform. The building environment is GNU/Linux Debian operating system on the i686 platform.

B.1 Preparation

RTEMS and tools sources are avaiable to download from ftp://ftp.rtems.org/pub or CVS repository. I got the developmental RTEMS 4.7 snapshot rtems-ss-20030417.

To build RTEMS the correct and patched versions of tools [1] have to be built and installed before. In snapshot's directory the file TOOL_VERSION contains tools names and their versions used to build the given snapshot. However, you can try the lastest tool versions. Tools sources and their patches can be found in build_tools and c_tools RTEMS repository directories.

It is a good practise to build tools and rtems in other place than in the directory they will be installed to. Therefore, I created subdirectories build_tools and c_tools in the top-level rtems directory and placed there tool sources and patches. Then I built the tools in separate build directory and installed them to mentioned subdirectories. RTEMS is installed to rtems subdirectory and its sources and a patch are placed in the sources subdirectory. Although this directory structure is not mandatory it is transparent and recommended.

B.2 Building GNU Toolset

B.2.1 Building the autoconf tool

Building autoconf(version 2.57):

1. ~/rtems/build_tools/sources/autoconf-2.57$ patch -p0 <patchfile
   applies the patch patchfile(if any RTEMS patch file is available), the number after -p option tells the patch program to strip off 0 slashes from the pathname. If a patch prompts for a file to patch, you may need to adjust this number.

2. ~/rtems/build/autoconf-2.57$ ../../build_tools/sources/autoconf-2.57/configure –prefix=$HOME/rtems/build_tools/autoconf/
   executes configure script from the directory ~/rtems/build_tools/sources/autoconf-2.57/, that creates a file containing configuration options for building autoconf in the current directory. The prefix variable specifies the install directory.

3. ~/rtems/build/autoconf-2.57$ make
   executes the building process, object, executable and library files are generated in the current directory.

4. ~/rtems/build/autoconf-2.57$ make install
   executes the instalation process, built executable files and libraries are copied to the directory specified in prefix variable during the configuration phase.

[1]The tools like binutils, gcc are called cross tools because they have to be built specially for target platform.

B.2.2 Building the automake tool

Building automake (version 1.7.6)

1. ~/rtems/build_tools/sources/automake-1.7.6$ patch -p0 <patchfile
   applies the patch patchfile(if any RTEMS patch file is available), the number after -p option tells patch program to strip off 0 slashes from the pathname. If a patch prompts for a file to patch, you may need to adjust this number.

2. ~/rtems/build/automake-1.7.6$ ../../build_tools/sources/automake-1.7.6/configure –prefix=$HOME/rtems/build_tools/automake/
   executes configure script from the directory ~/rtems/build_tools/sources/automake-1.7.6/, that creates file containing configuration options for building autoconf in the current directory. The prefix variable specifies the install directory.

3. ~/rtems/build/automake-1.7.6$ make
   executes the building process, object, executable and library files are generated in the current directory.

4. ~/rtems/build/automake-1.7.6$ make install
   executes the instalation process, built executable files and libraries are copied to the directory specified in prefix variable during the configuration phase.

B.2.3 Building the binutils cross tools

Binutils package contains target dependent utilities like ld, as, etc. Building binutils (version 2.13.2.1):

1. ~/rtems/c_tools/sources/binutils-2.13.2.1$ patch -p0 <patchfile
   applies the patch patchfile(if any RTEMS patch file is available), the number after -p option tells patch program to strip off 0 slashes from the pathname. If a patch prompts for a file to patch, you may need to adjust this number.

2. ~/rtems/build/binutils-2.13.2.1$ ../../c_tools/sources/binutils-2.13.2.1/configure –prefix=$HOME/rtems/c_tools/binutils/ –target=i386-rtems
   executes configure script from the directory ~/rtems/c_tools/sources/binutils-2.13.2.1/, that creates file containing configuration options for building autoconf in the current directory. The prefix variable specifies the install directory and the target variable specifies the target platform.

3. ~/rtems/build/binutils-2.13.2.1$ make
   executes the building process, object, executable and library files are generated in the current directory.

4. ~/rtems/build/binutils-2.13.2.1$ make install
   executes the instalation process, built executable files and libraries are copied to the directory specified in prefix variable during the configuration phase.

B.2.4 Building GCC

It is preferable to link embedded applications with the newlib library instead of libc since the newlib is optimized for this type of applications.

The C Library is built as a subordinate component of gcc. Because of this, the newlib source directory must be available inside the gcc source tree. This is normally accomplished using a symbolic link :

```
~/rtems/c_tools/sources/gcc-3.2.2$ ln -s ../newlib-1.11.0 newlib
```

If any RTEMS patch file for newlib is available, it has to be applied.

Before building gcc-3.2.2 and newlib-1.11.0, binutils-2.13.2.1 must be installed and the directory containing those executables must be in PATH environment variable. This can be accomplished issuing the command:

```
export PATH=~/rtems/c_tools/binutils/bin:$PATH
```

Building gcc (version 3.2.2):

1. ~/rtems/c_tools/sources/gcc-3.2.2$ patch -p0 <patchfile
   applies the patch patchfile(if any RTEMS patch file is available), the number after -p option tells patch program to strip off 0 slashes from the pathname. If a patch prompts for a file to patch, you may need to adjust this number.
2. Since GCC configuration requires more options , it is better to put them in a shell script (it is also useful as the history of options gcc was built with):

   ```
   #!/bin/bash

   ../../c_tools/sources/gcc-3.2.2/configure --prefix=~/rtems/c_tools/gcc\
           --target=i386-rtems \
           --with-gnu-ld \
           --with-gnu-as \
           --with-newlib \
           --enable-languages=c,c++ \
           --enable-threads \
           --verbose
   ```

   This shell script is then invoked from the directory ~/rtems/build/gcc-3.2.2. For more information about configuration options, invoke the configure script with the –help option.
3. ~/rtems/build/gcc-3.2.2$ make
   executes the building process, object,executable and library files are generated in the current directory.
4. ~/rtems/build/gcc-3.2.2$ make install
   executes the instalation process, built executable files and libraries are copied to the directory specified in prefix variable during the configuration phase.

B.3 Building RTEMS

In order to compile RTEMS, the toolset have to be specified in PATH variable. It is important to have the RTEMS toolset first in your path to ensure that the intended version of all tools is used. This can be accomplished issuing the command:

```
export PATH=$HOME/rtems/build_tools/autoconf/bin/:$HOME/rtems/ build_tools/
automake/bin/:$HOME/rtems/c_tools/gcc/bin:$HOME/rtems/c_tools/binutils/
bin:$PATH
```

It is also necessary to clear the list of directories in which gcc looks for header files by the command unset CPATH.

Building RTEMS (snapshot 20030417 of development version 4.7):

1. ~/rtems/sources/rtems-4.7$ ./bootstrap
   The bootstrap script automatically generates configuration files within RTEMS's source tree.

2. Since RTEMS configuration requires more options , it is better to put them in a shell script (it is also useful for the history of options RTEMS was built with):

   ```
   #!/bin/bash

   ../../sources/rtems-4.7/configure  \
           --target=i386-rtems --prefix=/home/martin/rtems/rtems-4.7 \
           --enable-networking --enable-rdbg \
           --enable-cxx --enable-tests --disable-docs \
           --enable-rtemsbsp=pc386  --build=i386-pc-linux-gnu \
           --disable-itron --enable-posix
   ```

   This shell script is then invoked from the directory ~/rtems/build/rtems-4.7. For more information about configuration options, invoke the configurescript with the –help option.

3. ~/rtems/build/rtems-4.7$ make
   executes the building process, object, executable and library files are generated in the current directory.

4. ~/rtems/build/rtems-4.7$ make install
   executes the instalation process, built executable files and libraries are copied to the directory specified in prefix variable during the configuration phase.

C Building and running applications

The following text decribes the steps how to build RTEMS application. The description is intended for beginners, another useful information about API can be found in [7], [16], [17].

The easiest way to learn how to write RTEMS application is to have a look at sample or test application in the RTEMS_ROOT/c/src/tests/ directory. A programmer should be aware that:

- application is required to define at least one initialization task.
- the upper limit on the number of resources (e.g. semaphores) is specified by a programmer.
- certain directives are functioning only when supporting RTEMS manager is involved in the build process of application.

Now, consider the test application spedf described in chapter 6. Its source files can be found in the rtems/sources/rtems/c/src/tests/sptests/spedf directory. The application consits of two C source files init.c and tasks.c. Initialization task specified in init.c creates two tasks defined in tasks.c. The files Makefile.am and Makefile.in are customized general makefiles for test applications and the following variables have to set specifically in both files:

- TEST = spedf
  sets the name of output application (spedf.exe).
- MANAGERS = io rate_monotonic
  specifies managers to be compiled in.
- C_FILES = init.c task.c
  specifies C source files to be involved in build process.
- H_FILES = system.h
  specifies header files to be involved in build process.

The simpliest way to build an application is to include it into RTEMS build system together with other RTEMS test applications. In order to spedf's makefiles are taken into account, do the following:

- Append the spedf subdirectory to the SUBDIRS variable in Makefile.am and Makefile.in files situated in the higher level ∼/rtems/c/src/tests/sptests directory.
- Append the line spedf/Makefile to explicit makefile list in the file ∼/rtems/c/src/tests/sptests/configure.ac.

Then build RTEMS according to instructions in Appendix B. If a change in application is made later on, only the make phase must be rerun. It is important to note that –enable-tests option must be specified in configuration options for RTEMS in order to test applications are compiled during RTEMS build. The output spedf.exe application can be found in the ∼/rtems/build/rtems-4.7/i386-rtems/c/pc386/tests/sptests/spedf/o-optimize directory.

Figure C.1 illustrates the RTEMS application build process. Note that RTEMS kernel is a library which is linked together with application object file to output file. Hence RTEMS is sometimes called an *executive*.

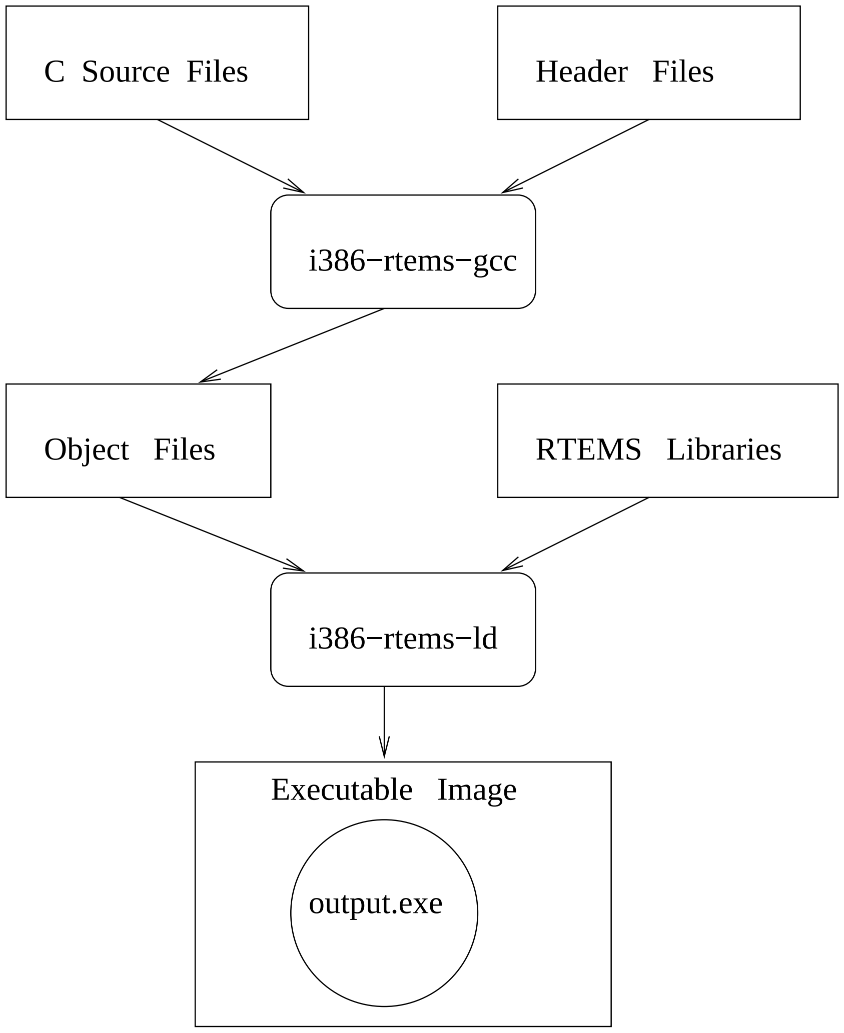


Figure C.1: The application building

RTEMS applications run on target hardware they were built for. While developing the application, it is often more practical to test aplication in hardware emulator. Such an emulator for Intel x86 platform is BOCHS. It includes emulation of the Intel x86 CPU, common I/O devices, and a custom BIOS.

Bochs can be build from source or installed from binary packages. For Debian Linux, the following BOCHS packages are available:

```
bochs - IA-32 PC emulator
bochs-doc - Bochs upstream documentation
bochs-sdl - SDL plugin for Bochs
bochs-svga - SVGA plugin for Bochs
bochs-term - Terminal (ncurses-based) plugin for Bochs
bochs-wx - WxWindows plugin for Bochs
bochs-x - X11 plugin for Bochs
bochsbios - BIOS for the Bochs emulator
bximage - Disk Image Creation Tool for Bochs
grub-disk - GRUB bootable disk image
```

```
sb16ctrl-bochs - control utility for Bochs emulated SB16 card
vgabios - VGA BIOS software for the Bochs and Qemu emulated VGA card
```

At least, bochs, bochs-term, bochsbios, vgabios packages have to be installed in order to emulator runs in text mode. For more information about BOCHS see [18] from where you can download source packages.

Bochs emulator is invoked by the command (seeman bochs for more options):

bochs -f configfile

Here I include my bochs configuration file:

```
config_interface: textconfig
display_library: term
romimage: file=/usr/share/bochs/BIOS-bochs-latest,address=0xf0000 megs:32
vgaromimage: file=/usr/share/vgabios/vgabios.bin
#romimage: file=~/rtems/bochs/share/bochs/BIOS-bochs-latest,address=0xf0000
megs: 32
#vgaromimage: file=~/rtems/bochs/share/bochs/VGABIOS-lgpl-latest
floppya: 1_44=~/rtems/apps/grubboot.img, status=inserted
ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14
ata1: enabled=0, ioaddr1=0x170, ioaddr2=0x370, irq=15
ata2: enabled=0, ioaddr1=0x1e8, ioaddr2=0x3e0, irq=11
ata3: enabled=0, ioaddr1=0x168, ioaddr2=0x360, irq=9
ata0-master: type=disk, path="~/rtems/apps/harddisk.img", cylinders=20,
heads=16, spt=63, translation=auto, biosdetect=auto, model="Generic 1234"
ata0-slave: type=cdrom, path="/dev/cdrom", status=inserted

boot: disk

ips: 1000000
clock:sync=none, time0=local
floppy_bootsig_check: disabled=0
log: /dev/stdout
panic: action=ask
error: action=report
info: action=report
debug: action=ignore
debugger_log: -
vga_update_interval: 300000
keyboard_serial_delay: 250
keyboard_paste_delay: 100000
mouse: enabled=1
private_colormap: enabled=0
```

To get more information about BOCHS config file see man bochsrc. The boot=disk parameter determines boot device while ata0-master: type=disk, path=" /rtems/apps/harddisk.img" specifies the virtual hard disk containing data stored in the harddisk.img file. How to create harddisk.img file is described in [19]. This image contains RTEMS application, grub loader installed in master boot record and grub's configuration file. My grub config:

```
default 3
```

```
timeout 5
color cyan/blue white/blue

title spedfp1 10000/4000 10000/6000 (7,10)priority
root  (fd0)
kernel /spedfp1.exe
boot

title spedfp2 10000/4000 10000/6000 (9,11)priority
root  (fd0)
kernel /spedfp2.exe
boot

title spedfe1 10000/4000 10000/6000 (7,10)edf
root  (fd0)
kernel /spedfe1.exe
boot

title spedfe2 10000/4000 10000/6000 (9,11)edf
root  (fd0)
kernel /spedfe2.exe
```

After invoking bochs emulator you can simple choose the application to run from grub menu.

D Applying EDF patch

Standard RTEMS distribution do not have a support for EDF scheduling. This feature can be added by applying the EDF patch by invoking the patch program in RTEMS source directory:

```
patch -p1 < edf-rtems-ss-20030417.diff
```

Note that now the EDF patch is only applied to RTEMS API. However, it is not difficult to customize other APIs for EDF that ought to be tested subsequently.

The patch includes the spedf application to test the functionality of EDF scheduler.

E CD Contents

The enclosed CD is bootable, after boot you can choose an application to run from grub menu. The CD has the following contents:

- **boot**
  The directory contains files needed by grub loader.

- **edf-rtems-ss-20030417.diff**
  This file is RTEMS EDF scheduler patch. See Appendix D how to apply the patch file.

- **exe**
  The directory contains RTEMS application that can be selected to run from grub menu.

- **spmutextest**
  The directory contains source files of the spmutextest application described in chapter 7. This application is not included in the EDF patch. See Appendix C how to build this application.

- **red-black_tree_docs**
  The directory contains documentation about RED-Black tree.

- **rtems**
  My directory for building RTEMS and applications. Its structure is described in Appendix B. The apps subdirectory contains .boshrc configuration file, the harddisk.img hard disk image and the floppy.img floppy image that can be copied to boot diskette by dd command.

- **diplom_work**
  The directory contains all documentation files (including images) of diplom work.