Enriching Enea OSE for Better Predictability Support

Master of Science Thesis

Naveed Ul Mustafa
naveedum@kth.se

Stockholm, August 2011

Examiner:
Ingo Sander
KTH
ingo@kth.se

Supervisor:
Mehrdad Saadatmand
ENEA&MDH
mehrdad.saadatmand@mdh.se

TRITA-ICT-EX-2011:194
Abstract

A real-time application is designed as a set of tasks with specific timing attributes and constraints. These tasks can be categorized as periodic, sporadic or aperiodic, based on the timing attributes that are specified for them which in turn define their runtime behaviors. To ensure correct execution and behavior of the task set at runtime, the scheduler of the underlying operating system should take into account the type of each task (i.e., periodic, sporadic, aperiodic). This is important so that the scheduler can schedule the task set in a predictable way and be able to allocate CPU time to each task appropriately in order for them to achieve their timing constraints.

ENEA OSE is a real-time operating system with fixed priority preemptive scheduling policy which is used heavily in embedded systems, such as telecommunication systems developed by Ericsson. While OSE allows for specification of priority levels for tasks and schedules them accordingly, it can not distinguish between different types of tasks.

This thesis work investigates mechanisms to build a scheduler on top of OSE, which can identify three types of real-time tasks and schedule them in a more predictable way. The scheduler can also monitor behavior of task set at run-time and invoke violation handlers if time constraints of a task are violated.

The scheduler is implemented on OSE5.5 soft kernel. It identifies periodic, aperiodic and sporadic tasks. Sporadic and aperiodic tasks can be interrupt driven or program driven. The scheduler implements EDF and RMS as scheduling policy of periodic tasks. Sporadic and aperiodic tasks can be scheduled using polling server or background scheme.

Schedules generated by the scheduler deviate from expected timing behavior due to scheduling overhead. Approaches to reduce deviation are suggested as future extension of thesis work. Usability of the scheduler can be increased by extending the scheduler to support other scheduling algorithm in addition to RMS and EDF.
Deep Thanks to Allah……
To my loving parents.....with respect and gratitude.....
# Contents

1 Introduction .................................................. 9  
1.1 Background .................................................. 9  
1.2 Relation With The CHESS Project .......................... 9  
   1.2.1 Description of The CHESS Project .................. 9  
   1.2.2 Place of Thesis Work Within The CHESS Project .... 10  
   1.2.3 Problem Statement .................................. 10  
1.3 Contributions .............................................. 10  
1.4 Thesis Phases ............................................... 11  
1.5 Time Frame ................................................. 11  
1.6 Structure of Thesis Report ............................... 11  

2 Real-Time Operating System ................................. 12  
2.1 Real-Time Systems .......................................... 12  
   2.1.1 Hard and Soft Real-Time Systems .................... 13  
2.2 RTOS ....................................................... 13  
2.3 General Components of an RTOS ........................... 14  
2.4 Tasks ....................................................... 15  
   2.4.1 Task States ........................................... 16  
   2.4.2 Periodic, Aperiodic and Sporadic Tasks ............ 17  
2.5 Priorities ................................................... 17  
2.6 Timers ...................................................... 17  
2.7 Scheduling .................................................. 18  
   2.7.1 Categories of Scheduling Algorithms ............... 18  
   2.7.2 Scheduling Algorithms for Periodic Tasks ........ 19  
   2.7.3 Scheduling of Sporadic and Aperiodic Tasks ....... 20  
2.8 Synchronization Objects ................................... 21  
   2.8.1 Semaphores .......................................... 21  
   2.8.2 Mutex ................................................. 22  
   2.8.3 Monitors ............................................. 22  
   2.8.4 Message Queues ..................................... 22  
2.9 Interrupt Handling ........................................ 23
3 Introduction to OSE

3.1 Processes

3.1.1 Process Types

3.1.2 Process Categories

3.1.3 Scheduling Policies

3.1.4 Process States

3.1.5 Summary of Process Scheduling in OSE

3.2 Message Passing

3.2.1 Signals

4 Desired Capabilities of The Scheduler

4.1 Implementation of Periodic and Sporadic Tasks

4.2 Support for Fixed Size of Task Set

4.3 Specifying Non-Functional Parameters

4.4 Fixed Priority Preemptive Scheduling

4.5 Monitoring of Execution Time Overruns and Deadlines Misses

4.6 Support for Absolute Delay

4.7 Single Event Dependency

5 Evaluation of OSE for Implementation of The Scheduler

5.1 Evaluation for Implementation of Periodic and Sporadic Tasks

5.2 Evaluation for Supporting Fixed Size Task Set

5.3 Evaluation for Specification of Non Functional Parameters

5.4 Evaluation for Fixed Priority Preemptive Scheduling

5.5 Evaluation for Monitoring of Execution Time Overruns and Deadline Misses

5.6 Evaluation for Supporting Absolute Delays

5.7 Evaluation for Supporting Single Event Dependency

5.8 Summary of Evaluation of OSE

6 Overview and Evaluation of Commonly Used RTOSes

6.1 FreeRTOS

6.1.1 Tasks

6.1.2 Communication and Synchronization

6.2 Evaluation of FreeRTOS

6.3 Windows Embedded CE

6.3.1 Threads

6.3.2 Communication and Synchronization

6.4 Evaluation of Windows Embedded CE

6.5 QNX

6.5.1 Processes
6.5.2 Inter-Process Communication .......................... 48
6.6 Evaluation of QNX ........................................... 48

7 Design of The Scheduler ................................. 51
  7.1 System Overview .......................................... 51
  7.2 Assumptions ............................................... 51
  7.3 System Components ....................................... 52
    7.3.1 Process Creator .................................. 52
    7.3.2 Scheduler .......................................... 55
    7.3.3 Sporadic Queue Holder ............................ 56
    7.3.4 Aperiodic Queue Holder ............................ 56
    7.3.5 Monitor .............................................. 57
  7.4 System Signals ........................................... 57
  7.5 Priority Assignment ..................................... 59
  7.6 System Operation ....................................... 59
    7.6.1 Scheduling of Periodic Tasks ...................... 60
    7.6.2 Scheduling of Sporadic and Aperiodic Tasks ..... 60
    7.6.3 Monitoring ......................................... 62
    7.6.4 Summary .............................................. 63
  7.7 Design Features ......................................... 64
  7.8 Design Limitations ..................................... 65

8 Implementation and Results .......................... 66
  8.1 Implementation .......................................... 66
    8.1.1 Constraints Data Structure ....................... 66
    8.1.2 Pseudo-Code of Process Body ..................... 68
    8.1.3 Pseudo-Code of Process Creator .................. 68
    8.1.4 Pseudo-Code of Sporadic Queue Holder ........... 69
    8.1.5 Pseudo-Code of Aperiodic Queue Holder .......... 70
    8.1.6 Pseudo-Code of Monitor ............................ 72
    8.1.7 Pseudo-Code of Scheduler .......................... 72
  8.2 Formatting of Log Files ............................... 74
    8.2.1 Formatting of Scheduler_log.txt File ............ 75
    8.2.2 Formatting of Monitor_log.txt File ............... 75
  8.3 Verification of Results ................................ 76
    8.3.1 Case 1: ............................................. 76
    8.3.2 Case 2: ............................................. 77

9 Conclusions and Future Work ........................ 81
  9.1 Conclusion ............................................. 81
    9.1.1 Scheduling Overhead .............................. 81
List of Figures

2.1 High level view of an RTOS ........................................ 14
2.2 Abstract presentation of a task system .......................... 15
2.3 Task state transition diagram for general real-time system .. 16
2.4 Earliest Deadline First Scheduling ............................... 19
2.5 Rate Monotonic Scheduling Algorithm ........................... 20
2.6 Comparison of different scheduling schemes for sporadic and aperiodic tasks ........................................... 20

3.1 Process states in OSE ............................................. 27
3.2 An overview of process scheduling in OSE .................... 28

4.1 Abstract presentation of a task system .......................... 32

6.1 Task states in FreeRTOS ........................................... 40

7.1 Process creator ..................................................... 54
7.2 Scheduler .......................................................... 55
7.3 Sporadic queue holder ............................................. 56
7.4 Aperiodic queue holder ........................................... 57
7.5 Monitor component ............................................... 58
7.6 Design of the scheduler ........................................... 61
7.7 Sequence diagram to demonstrate operation of the scheduler 63

8.1 Expected schedule for task set given in case 1 ............... 77
8.2 Scheduler generated by the scheduler for task set given in case 1 78
8.3 Expected schedule for task set given in case 2 with budget consumption and repletion ......................................... 79
8.4 Schedule generated by the scheduler for task set given in case 2 with budget consumption and repletion ....................... 80
List of Tables

6.1 Evaluation of FreeRTOS ........................................... 40
6.2 Evaluation of Windows Embedded CE ............................. 44
6.3 Evaluation of QNX .................................................. 49

7.1 Priority assignment to OSE processes ......................... 59
Chapter 1

Introduction

1.1 Background

This thesis is performed at Mälardalen University Sweden (MDH) and ENEA AB in scope of the CHESS Project [1]. MDH is a leading Swedish university with its School of Innovation, Design and Engineering (IDT) having extensive research projects in embedded and real-time systems. ENEA is a global software and services company focusing on solutions for communication driven products [2]. Operating System E (OSE) is a Real-Time Operating System (RTOS) developed by ENEA. Many systems, e.g. mobile phones, automobiles and medical devices are using OSE [3].

1.2 Relation With The CHESS Project

1.2.1 Description of The CHESS Project

Current component based runtime environments mainly focus on preserving the functional properties of specified components. One challenge in designing real time systems is to preserve non-functional properties (e.g. latency, memory usage, fault tolerance, predictability and reliability) and verify this preservation throughout the whole process (i.e. from specification down to the modeling, transformation and code generation).

The CHESS project is a collaboration among leading research & development, academic and industrial organizations from all across Europe. The CHESS project focuses on capturing (i.e. specification and preservation of) non-functional properties[1]. Development of tools and techniques to support

---

\(^1\)Composition with Guarantee for High-integrity Embedded Software Component Assembly
description, verification and preservation of non-functional properties from the perspective of multiple industrial domains (space, railways, telecommunications and automotives) is the focus of the CHESS project.

1.2.2 Place of Thesis Work Within The CHESS Project

Within the CHESS project, execution platforms for above mentioned industrial domains are to be identified and adapted to support preservation of non-functional requirements. One of these domains is telecommunications. OSE by ENEA is identified as the execution platform of interest for this domain. Therefore, OSE is required to be extended to provide better support for predictability.

1.2.3 Problem Statement

The main goal of this thesis is to develop a scheduler on top of OSE real-time scheduler with better predictability support that provides the following features:

1. Enables users to specify periodic and sporadic tasks with their respective specifications such as the period, deadline, Minimum Inter-Arrival Time (MIAT) and Worst Case Execution Time (WCET).

2. Extends OSE scheduler to schedule task set consisting of periodic, sporadic and aperiodic tasks.

3. Provides interfaces to monitor system behavior such as exceeding execution time, missing a deadline and violating MIAT.

1.3 Contributions

The goal of thesis work is achieved by

1. Study of concepts related with real-time operating systems.

2. In depth study of ENEA OSE and its evaluation for implementation of scheduler.

3. Study of FreeRTOS, Windows Embedded CE and QNX and their evaluation for implementation of scheduler. This provides the base to make comparison of OSE with above mentioned RTOSes for implementation.

4. Design of the scheduler for implementation on OSE.
5. Implementation of the scheduler.

6. Verification of the scheduler’s timing behavior.

1.4 Thesis Phases

The thesis work is organized into two phases. Phase I focuses on literature review, including study of all related material like documents about the CHESS project, books with content about RTOSes, research papers of interest, preparation of the initial (alpha) report and a three day basic OSE training course at ENEA AB. The duration of phase I is eight weeks. Phase II consists of design and implementation of the scheduler, verification of timing behavior of the scheduler, refinement and updates of the thesis report and presentation of the thesis work.

1.5 Time Frame

The time limit for this thesis work is 22 weeks in which all literature review, implementation, reports and presentation is to be completed.

1.6 Structure of Thesis Report

Phase I consists of chapters 1 to 6. Chapter 1 introduces the thesis work. Chapter 2 contains concepts and terminologies related with an RTOS (such as OSE). Chapter 3 provides introduction to OSE which is an example of RTOS.

Chapter 4 describes the capabilities, which the scheduler need to have in order to meet the criteria laid out in problem statement. Chapter 5 evaluates OSE for implementation of the scheduler.

Chapter 6 provides overview of commonly used RTOSes like FreeRTOS, Windows Embedded CE and QNX. It also presents summary of evaluation of these RTOSes for implementation of the scheduler.

Phase II consists of chapters 7 to 9. Chapter 7 presents design of the scheduler to be implemented on top of OSE. Chapter 8 provides pseudocode for implementation of the scheduler. It also present a discussion on results generated by the scheduler. Chapter 9 contains concluding remarks and suggests a direction for future work.
Chapter 2

Real-Time Operating System

This chapter starts with a description of real-time systems and their types in section 2.1. Many real time systems rely on an RTOS to provide basic services. RTOS and its components are described in section 2.2 and 2.3. RTOS uses concept of multitasking to achieve a sense of concurrency. Tasks and priority assignment to them is discussed in section 2.4 and 2.5. Section 2.6 describes timers. Algorithms for scheduling of tasks are discussed in section 2.7. Tasks need to communicate with each other and execute in a synchronous way. Synchronization objects supported by an RTOS are described in section 2.8. Chapter concludes with a description of interrupt handling in section 2.9.

2.1 Real-Time Systems

Today, more and more systems are developed, which need to interact with their environment in a timely manner. They need to read input data or signals, apply the processing algorithm and generate a result (or output) within specified time limits. Such systems can be a part of a larger system such as an embedded system, or they can exist independently.

Systems which are required to handle external events with timing constraints are called real-time systems.[4]

An artificial heart pacemaker is a medical device, which is example of a (real-time) system interacting with its environment [5]. It has electrodes which act as sensors. Electrical pulses (i.e. input signals) delivered to artificial pacemaker by electrodes, are used to regulate heart beat (output). It is critical for an artificial heart pacemaker to read not only pulses in a timely manner but also process them and generate output electrical pulses within specified time constraints. Failure of pacemaker to send electrical pulses to
Two terms are of special interest with respect to real-time systems: timely fashion and external events. External events are input signals or occurrences generated due to interaction of the system with its environment. Such events are to be handled by the real-time system. For example, in the above mentioned example of pacemaker, signals coming from electrodes act as an external event. If heart rhythm is detected to be abnormal, pacemaker responds to the event by generating output electrical pulses to heart.

Response in timely fashion means that system must take actions against corresponding events within specified time limits. In real-time systems, accuracy of results and meeting the time limits are equally important. For example, a late response by pacemaker can be catastrophic for a person.

Other examples of real-time systems from modern industrial domain include robots, space missions, chemical and nuclear plants [6].

2.1.1 Hard and Soft Real-Time Systems

In real-time systems, it is a challenge to guarantee that all events will be responded within specified time constraints. For example, an important event may be responded in a timely manner at the expense of an event with relatively low importance missing its deadline. Keeping this fact in mind, real-time systems can be divided into two categories: soft real-time systems and hard real-time systems.

The time point by which a task in real-time system has to complete its execution is called deadline. If the result produced after deadline is still useful, system is categorized as a soft-real time system [7]. Live video streaming application is a popular example of a soft real-time system. Although response times of video frames are important in such an application, but missing a deadline does not lead to catastrophic result.

On the other hands, a system in which results produced after deadline are totally unacceptable and useless is termed as a hard real-time system [8]. An example of a hard real-time system is railway traffic signaling system. If such a system misses to send a signal to one of two trains, which need to pass a single railway route, it could lead to loss of many human lives.

2.2 RTOS

Real-time systems with small size of code and low level of complexity can be developed as a single control loop. An operating system may not be needed. Such real-time systems are categorized as bare metal. An operating system
is not a compulsory part of a real time system. Functionality provided by RTOS can be implemented by application program itself [9].

However, an RTOS provides re-usability and portability. Therefore, design time is reduced. In addition, by using an RTOS an application designer does not need to implement scheduling policies, synchronization and communication mechanisms. Hence, designer can focus more on application level development.

2.3 General Components of an RTOS

Figure 2.1 presents general components of an RTOS. As depicted in Figure 2.1, every RTOS has a core component called kernel, and some optional or configurable modules (for example, modules for support of a device I/O, debugging, networking and file system).

A real-time kernel is a part of RTOS and provides all necessary services for embedded (or real-time) applications (for example, resource management, Inter-Process Communication IPC and scheduling) [10]. Because of memory size limitation in real-time systems, kernel is required to be as small as possible.

Main components of a kernel include

1. **Scheduler**: It is part of kernel, which allocates CPU time to multiple tasks based on a pre-defined scheduling policy (e.g. priority based
2. **Preemptive scheduling or round robin scheduling**.

2. **Synchronization and Communication Objects**: Tasks may use shared resources. Therefore, they need to communicate with each other to access resources in a synchronized way. Synchronization objects serve this purpose. Such objects include semaphores, mutexes, critical sections, message queues, signals, and mailboxes.

3. **Background Services**: These services are needed for interrupt handling, garbage collection, collection of timing information, and resource management.

2.4 **Tasks**

A real-time system is usually modeled as a set of concurrent tasks. Each task has a set of constraints namely release time ($\phi$), period ($P$), execution or computation time ($C$), and deadline ($D$). A task is represented as $\tau(\phi, P, C, D)$, as depicted in Figure 2.2.

![Figure 2.2: Abstract presentation of a task system](image)

Task’s constraints include

- **Release Time $\phi$**: It is the time instant at which a task becomes available for execution. After release time a task can be scheduled, whenever its all data and control dependencies are met [11].

- **Period $P$**: Period is an important parameter in case of periodic tasks. Period is defined as the time interval after which next instance of periodic task is released. This constraint is replaced by MIAT in case of sporadic tasks.

- **Execution/Computation time $C$**: Time required by a task for execution, when it has all resources it needs including CPU, is termed as execution time [11, 12].
• Response time: It is defined as an interval between time instant when a task is released and its completion [11].

• Absolute Deadline \( D \): Absolute deadline of a task is the instant of time from start of system, by which its execution is required to be completed [11, 12].

• Relative Deadline \( d \): Maximal allowable response time of a task is its relative deadline \( d \). In other words, relative deadline is duration between release time and absolute deadline.

2.4.1 Task States

A task cannot be allocated the CPU for an infinite time. Hence a task cannot always be in running state on a CPU. This leads to logical result that a task can have more than one state, as depicted in Figure 2.3.

![Figure 2.3: Task state transition diagram for general real-time system](image)

Task states are

1. **Ready**: A task is in the ready state when it has all the resources needed for execution. Yet it is waiting for scheduler to allocate it CPU time.

2. **Running**: A task is in running state when it is allocated CPU time.

3. **Suspended**: In suspended state, a task waits for arrival of an event or availability of a shared resource.
2.4.2 Periodic, Aperiodic and Sporadic Tasks

In a real-time system, various type of functions are to be implemented. Release time for some functions is known in advance and they have a periodic behavior. Such functions can be modeled by periodic tasks. Periodic task is one which has a fixed interval of time between two consecutive activations. This fixed interval is called the period of a task [12]. For example, update of temperature on a display screen with regular interval is a periodic task.

Not all functions can be represented by periodic tasks. The reason is that arrival time is not a known priori for some external events. Furthermore, they may not release periodically at regular intervals. Such functions can be modeled by aperiodic or sporadic tasks. Aperiodic tasks are those which are activated at arbitrary time instants with soft deadlines and there is no regular interval between two activations [6, 11]. A button pressed to generate the result of addition on the display screen of a calculator is an example of an aperiodic task. In this case if the result is displayed with some delay, performance is degraded but result is still acceptable.

A sporadic task has two parameters: hard deadline and MIAT [11, 13, 14]. MIAT is the time duration which must be maintained between two consecutive releases of a sporadic task. Second activation of a sporadic task can occur only after MIAT is elapsed relative to previous activation. An example of a sporadic task is detection of fire and corresponding action taken by a home safety system. Release time of such a task is not known in advance. Furthermore, it has a hard deadline to meet. Otherwise, missing a deadline can result in damage of property and human life.

2.5 Priorities

In a real-time system not all tasks are equally important. Some tasks are more critical as compared to others. In order to represent this fact, tasks are assigned priorities. These priorities are mainly used by scheduling algorithms to allocate CPU time to multiple tasks. The more important or critical a task is, the higher is its priority.

2.6 Timers

In real-time systems, the notion of time is very important. Timers are events scheduled with predefined timeout values in future [4]. A timer can be categorized as physical timer or logical timer. A physical timer is derived from
physical timer chip, whereas logical timer is a software event scheduled using RTOS system call.

2.7 Scheduling

In RTOS, a given functionality may be achieved by dividing it into several tasks. All tasks cannot use CPU at the same time. They must be managed in such a way that each task gets its share of CPU execution time according to some predefined protocol. Scheduler is responsible for this management [9]. Scheduler distributes CPU execution time among multiple tasks using a scheduling algorithm.

2.7.1 Categories of Scheduling Algorithms

Scheduling algorithms can be divided into two main categories: off-line scheduling and on-line scheduling.

**Off-Line Scheduling:** In this category, schedule is computed before the system begins to execute. Release times and resource requirements of all tasks is known in advance. [11]. Such algorithms provide maximum usage of system resources. However, they can be used only if number of tasks in a system is already known and no task is created dynamically.

Due to off-line computation of schedule, scheduling overhead is relatively low. Drawback of off-line scheduling is the lack of flexibility. Whenever a new task is to be included, schedule needs to be computed again and updated. Furthermore, only periodic tasks can be scheduled using this approach. Use of round robin algorithm with the number of tasks known in advance is an example of off-line scheduling.

**On-Line Scheduling:** On-line schedules are not computed before starting a system. Time constraints of each task becomes known to scheduler only when task is released, hence all scheduling decisions are made at runtime [11]. On-line scheduling is flexible as compared to off-line scheduling. It can handle periodic, aperiodic and sporadic tasks. However, due to runtime computation of schedule, preemption overhead is relatively high. All priority driven scheduling algorithms (e.g. First In First Out algorithm with priorities assigned based on release times or Longest Execution Time First algorithm with priorities assigned based on execution times) belong to this category.
2.7.2 Scheduling Algorithms for Periodic Tasks

Periodic tasks can be scheduled using priority based preemptive scheduling algorithms, for example, Earliest Deadline First (EDF) algorithm or Rate Monotonic Scheduling (RMS) algorithm.

In EDF, tasks are scheduled according to their deadlines. The earlier the deadline of a task is, the higher is its priority [16]. Deadline of all tasks are calculated at each timer tick and a task with the nearest/earliest deadline is selected [9]. Figure 2.4 presents a set of three tasks T1, T2 and T3 which are scheduled according to EDF algorithm.

T1: Release time = 1, Execution time = 8, Relative deadline = 15
T2: Release time = 2, Execution time = 3, Relative deadline = 16
T3: Release time = 3, Execution time = 6, Relative deadline = 12

![Figure 2.4: Earliest Deadline First Scheduling](image)

In RMS, tasks are assigned priorities based on their period. Lower the period of a task, higher is its priority. Figure 2.5 presents schedule of following three tasks using RMS algorithm.

T1: Period = 9, Release time = 0, Execution time = 5,
T2: Period = 3, Release time = 0, Execution time = 1,
T3: Period = 6, Release time = 0, Execution time = 3,


2.7.3 Scheduling of Sporadic and Aperiodic Tasks

Release time of aperiodic and sporadic tasks can not be determined in advance. Hence they can not be scheduled in same fashion as periodic tasks.

Different schemes are available to schedule sporadic and aperiodic tasks along with periodic ones. Figure 2.6 from [22] presents comparison of different scheduling schemes for sporadic and aperiodic tasks. Background and polling server schemes give better performance and are easy to implement as compared to other schemes such as slack stealer.

In background scheme, idle time of scheduler is used for scheduling of aperiodic and sporadic tasks. Whenever, there is no ready periodic task
to be executed, scheduler checks if sporadic or aperiodic task is ready. If so, sporadic or aperiodic task is scheduled until either periodic task becomes ready or all sporadic or aperiodic tasks are executed.

In polling scheme, a periodic task is designated as a polling server. Execution time of the polling server is known as its execution budget. Whenever polling server is released, its budget is replenished to its execution time. On release of polling server, ready sporadic or aperiodic task is scheduled for time interval equal to execution budget. From that point on, execution budget reduces at rate of 1 per unit time. Scheduled aperiodic or sporadic task is preempted, when execution budget of polling server becomes zero.

If polling server is released and there is no ready sporadic or aperiodic task, then budget of polling server immediately becomes zero. More detailed discussion on scheduling of aperiodic and sporadic tasks can be found in [11].

2.8 Synchronization Objects

In real-time systems, tasks need to cooperate and communicate with each other to perform a common system level function. RTOS provides various mechanisms, which help in synchronization and communication among multiple tasks. Most common of these synchronization objects are semaphores, events, mutexes, monitors and message queues. They are discussed in following subsections.

2.8.1 Semaphores

In real-time systems, multiple tasks may need to share a resource. To maintain integrity of data, only one task should access the shared resource at a time. When a task is using a shared resource, it should not be interrupted by other tasks. Such resources, which are accessed in mutually exclusive manner, are known as serially reusable resources. Parts of application code which use these resources are known as critical sections [8]. Critical sections can be implemented using semaphores.

“A semaphore is a kernel object that one or more tasks can acquire or release for synchronization” [16]. Semaphore $S$ is implemented by two procedures [8].

1. Wait($S$): if $S > 0$ then $S = S - 1$ else suspend the running task.

2. Signal($S$): if no task is suspended on $S$ then $S = S + 1$ else resume one task.
A semaphore can be initialized with any non negative number. An example of using printing service, where only one user can access a printer can be modeled by use of semaphore.

2.8.2 Mutex

A mutex is a special binary semaphore with two states: locked and unlocked. It is much more powerful as compared to semaphore because of following features [17].

1. It will have an owner unlike semaphores. Mutex is created in an unlocked state. Owner of mutex is the task which acquires it. Only the task that owns the mutex can release it. On the other hand, binary semaphore can be released by any task that did not originally acquire it. This can lead to potential problems in maintaining synchronization.

2. A task owning a mutex cannot be deleted (unlike semaphore).

2.8.3 Monitors

Semaphores are difficult to handle and are error prone. For example, suppose a semaphore is acquired by only one task in system and that task is deleted. Now all other tasks blocked on that semaphore will be waiting.

Monitors provide relatively a clean approach to achieve synchronization among multiple tasks in a system. A monitor encapsulates shared data within atomic procedures. Any other task which want to access shared data must request service of monitor task. To request monitor service, a lock is used. A monitor has a single lock, and a task that needs service from monitor must acquire it exclusively [16].

Suppose that access to a printing service is encapsulated by a monitor task $T_M$. All other tasks which need to use this service cannot access the printer directly. They must send their request with data to be printed to $T_M$. Only that task (among multiple tasks requesting access) which has ownership of monitor lock will be served while other will wait for lock to be free.

2.8.4 Message Queues

Semaphores, mutexes and monitors provide synchronization among multiple tasks. However, they can not be used for data exchange between two tasks. Message queues are used for this purpose [9].

Message queues are an important mean of communication among multiple tasks in a real-time system. One or multiple messages can be placed in a
message queue. Messages are placed at the end of a queue and removed from its front [15]. Several operations like blocking read, blocking write, non blocking read and non blocking write can be applied on message queues. In some RTOSes like OSE, messages can be used for both synchronization and data exchange. More detailed discussion about message queues can be found in [4].

2.9 Interrupt Handling

An important feature of real-time systems is that they interacts with their environment. Therefore, interrupt handling is an important concept in such systems. An interrupt is an asynchronous event that disrupts the normal execution flow [4]. In other words, an interrupt is an asynchronous signal asserted by environment (e.g, a peripheral device) [9].

Whenever an interrupt occurs, RTOS calls the corresponding Interrupt Service Routine (ISR). When ISR is completed and a higher priority task is ready, RTOS makes a context switch. Otherwise, task which was interrupted is scheduled on CPU.
Chapter 3

Introduction to OSE

Facts and figures in this chapter are excerpts from [19], unless specified otherwise.

In chapter 2, different concepts related to RTOS are described. This chapter presents overview of OSE.

OSE is a distributed real-time, fully preemptive OS, optimized to provide high rates of data throughput. It is based on message passing architecture. It does not rely on semaphore and shared memory for communication among processes. Two main entities of the message based architecture of OSE are processes and messages [18].

3.1 Processes

Process is the most fundamental building block since it is through a process that scheduler allocates CPU time.

3.1.1 Process Types

In OSE, there are four types of processes:

1. **Interrupt Process:**
   Interrupt processes are invoked in response to a hardware or software interrupt. They execute without blocking and can be preempted only by higher priority interrupt processes. Interrupt processes always have higher priority than prioritized and background processes. They can be used to model hardware driven sporadic and aperiodic tasks. Priority number assigned to an interrupt process at time of creation
is vector number of interrupt for which it is registered. Whenever an interrupt with this vector number arrives, interrupt process is invoked. Interrupt process can also be invoked by other processes by signaling it’s fast semaphore or sending a signal to it.

2. **Timer Interrupt Process:**
   Timer interrupt processes can be used to model periodic tasks. They execute periodically and at priority level of system-timer process. Period of execution is specified at time of creation. At each execution of system-tick function, kernel checks that which timer interrupt processes are ready to run. Frequency of calling system-tick function depends on defined length of tick. To model a periodic task with frequency and priority higher then system-timer process, an ordinary interrupt process tied to external higher frequency timer can be used.

3. **Prioritized Process:**
   Prioritized processes are most commonly used processes. Their priority is assigned at time of creation. However, it can be manipulated later on by using system calls. Prioritized processes are implemented usually in form of an infinite loop and waiting for a signal to start its execution.

4. **Background Process:**
   Background processes are scheduled only if there is no interrupt process, timer interrupt process or prioritized process ready to run. Main feature of background processes is that they have no priority value. An important parameter for background process is time slice which is specified at time of creation. Round robin policy is used to schedule multiple ready background processes. Such processes can be used for measuring statistics such as time interval for which no process is scheduled (besides background process itself).

5. **Phantom Process:**
   Phantom processes are not real processes. They are never scheduled and have no priority. They have no code associated with them and hold only data. They are used to implement redirection tables in link-handlers.
3.1.2 Process Categories

There are two categories of processes in OSE depending upon their life time and termination behavior.

Dynamic Process

Dynamic process is one which can be freely created and killed at run time. It enables the system to run multiple instances of same code. Number of instances is not needed to be known at compile time. Dynamic process also helps to save stack memory as stack is set free when process is killed.

Static Process

Static processes exist for the life time of program. They are created and started automatically when application starts. They cannot be killed during program execution. If a static process is terminated for any reason, system error handler is invoked with fatal error.

3.1.3 Scheduling Policies

OSE uses following scheduling policies.

1. Fixed-Priority scheduling
2. Preemptive scheduling
3. Periodic scheduling
4. Round-robin scheduling

3.1.4 Process States

Processes in OSE can have one of following three states.

1. Running
2. Ready
3. Waiting

These states are depicted in Figure 3.1.
3.1.5 Summary of Process Scheduling in OSE

Figure 3.2 presents an overview of processes and scheduling principles in OSE.

3.2 Message Passing

In many cases a process needs to communicate with other processes. In OSE, communication can be achieved by using signals, fast semaphores or semaphores. However, recommended means of communication in OSE is message passing.

3.2.1 Signals

In OSE, signal is an independent non-shared entity. Signal is a message, containing information which one process wishes to convey to another process. A signal has following attributes:

1. Identity
2. Sender
3. Owner
4. Addressee
5. Data

Data attribute is optional and individual to all signals.

![Diagram of process scheduling in OSE](image)

**Figure 3.2:** An overview of process scheduling in OSE

Four steps of passing message between two processes in OSE are signal allocation, hunting target process, sending the signal and receiving the signal.

1. **Signal Allocation:**
   Before sending a signal, a process must allocate and initialize it. Signal buffers are allocated from pool memory. The allocation is a fast and constant-time operation. Signal allocation can be performed by using `alloc` system call. More details on `alloc` system call are available in [21].

2. **Finding Process Identifier (PID) of Target Process:**
   It is necessary for a process to know PID of target process. If second process is defined as static process and in same load module, then PID
of that process is available as global variable. Another way to find PID is to use hunt system call. Hunting works for both dynamic and static processes. To perform hunting, a process needs to know name of target process. If target process is found, hunt-signal and PID of sought process is sent back to calling process by kernel. Hence it appears that hunt-signal is sent by sought process. Calling process in hunting can also provide its own hunt-signal, which is sent back to the calling process on successful hunting. Otherwise, standard hunt-signal is sent back by kernel. Details of hunt system call are available in [21].

3. Sending the Signal:
After allocation and initialization of signal and successfully finding PID of target process, signal can be sent to target process with send system call. After calling send system call, sending process loses ownership of the signal. If addressee process has a redirection table, signal is redirected to the intended process. [21] provides details of send system call, creation and setting of redirection tables for a given process.

4. Receiving the Signal:
Every process has an in-queue for signals. A process can receives signals by using receive system call. A process lists signals of its interest (among all those available in the in-queue) as a parameter of receive system call. If an appropriate signal is not available, calling process is pended i.e. a context switch occurs. On reception of one of the desired signals, execution of receiving process can continue. Details of receive system call are available in [21].

Attaching to A Process
If a process sends signal to another process, but target process is terminated then signal is simply freed by kernel. In such situation, caller process is never updated about the fact that target process is terminated.

To help with this problem, a process can subscribe for death/termination notice of target process. This can be achieved by using attach system call. In attach system call, calling process specify the process to attach with. Furthermore, calling process can provide its own signal which is sent back by kernel when target process is terminated. Otherwise, kernel sends back a standard signal to calling process upon termination of target process.
Summary of Message Passing

Message passing consists of four steps.

1. Allocating the signal
2. Hunting
3. Sending the signal
4. Receiving the signal

A calling process can subscribe for death notice of target process by using `attach` system call.
Chapter 4

Desired Capabilities of The Scheduler

Content of this chapter is based on deliverables from a work module of the CHESS project.

As stated in chapter 1, outcome of this thesis work is a scheduler on top of OSE. Such scheduler provides support for defining tasks with all non-functional parameters like period, WCET and deadline. It also provides support to monitor system behavior.

To achieve this goal, the scheduler should have certain capabilities which are discussed in following sections.

4.1 Implementation of Periodic and Sporadic Tasks

Real-time applications can be decomposed into a number of separate tasks, each performing a certain function within a single thread of control. Task release can be either time-triggered or event-triggered. Two important task release patterns are periodic task release and sporadic task release. In periodic case, tasks must be released at beginning of each period. In sporadic case, time between two releases of same task must not be smaller than its MIAT. Thus the scheduler should provide means to implement periodic and sporadic tasks. Both program-driven and interrupt-driven sporadic tasks must be supported. must enforce periodic releases for periodic tasks.

• must enforce MIAT between two consecutive releases of a sporadic task.
4.2 Support for Fixed Size of Task Set

To generate a valid schedule, a given task set should be tested for schedulability. To predict statically that a task set is schedulable or not, number of task N in set should be fixed and must not change over time. To keep N as a fixed number, dynamic creation of tasks should not be allowed. Therefore, the scheduler

- must provide means to implement task set where all tasks are created in a system initialization phase, and no more tasks are dynamically created thereafter.

- must provide means to implement task set where all tasks do not terminate.

4.3 Specifying Non-Functional Parameters

For each task in the task set following parameters must be specified.

\( N \): Number of tasks in a system

For each \( i \) in 1,.., \( N \), the \( i \)th task is represented by \((\phi_i, T_i, C_i, D_i)\):

1. \( \phi_i \): Release time of the task
2. \( T_i \): Period or MIAT
3. \( C_i \): WCET
4. \( D_i \): Relative deadline

![Figure 4.1: Abstract presentation of a task system](image)

To implement a task set, the scheduler must provide means to associate WCET, period, MIAT and deadline with each task.
4.4 Fixed Priority Preemptive Scheduling

To predict order of scheduling for a given task set, fixed priority preemptive scheduling must be used. Association between tasks and their priorities should be static so that schedule remains predictable. Otherwise, if priorities keep changing dynamically, scheduling cannot be predicted as priorities are not known.

EDF is a type of priority based preemptive scheduling, where priorities are assigned on base of deadlines. This implies that the scheduler

- must support fixed-priority preemptive scheduling. The association between threads and priorities must be static.
- may optionally support earliest-deadline-first scheduling.

4.5 Monitoring of Execution Time Overruns and Deadlines Misses

Response time of a task can be affected by non-task entities like interrupt or scheduling overhead. Hence there is potential of violation of execution times and deadlines. In order to detect execution time overruns and deadline misses, the scheduler must support execution time monitoring.

4.6 Support for Absolute Delay

While scheduling the tasks on a single processor, scheduler needs to suspend itself for fixed duration of time. Therefore, the scheduler should have capability to specify and enforce absolute delays.

4.7 Single Event Dependency

A task may depend on a result produced by another task or an event. Blocking of one task on a large number of other tasks can lead to potential synchronization problems and non-predictability of schedule. Therefore, the scheduler should allow blocking of a task on at most single event.
Chapter 5

Evaluation of OSE for Implementation of The Scheduler

In chapter 4, desired capabilities of the scheduler are discussed. This chapter perform an evaluation of OSE with perspective of implementing the scheduler. In following sections, possible implementation approaches are suggested for achieving above mentioned capabilities.

5.1 Evaluation for Implementation of Periodic and Sporadic Tasks

As discussed in chapter 3, OSE has four type of processes with a message passing architecture. OSE provides no process types as periodic, sporadic or aperiodic.

The scheduler can use following approaches to implement periodic tasks through processes and signals.

1. Using timer interrupt process with time slice parameter set to period of task.
2. Using prioritized process waiting on a signal released by scheduler periodically.
3. Using prioritized process which is started and stopped by scheduler periodically.

For sporadic process scheduling, either background scheme or polling server scheme can be used, as discussed in chapter 2. In either case sporadic task
can be implemented using a prioritized process. In background scheme that prioritized process is scheduled by scheduler whenever no periodic task is ready. In Polling scheme, same prioritized process is scheduled by scheduler when polling server is released and it has non zero budget.

In case of a sporadic task, time-stamp can be obtained every time it is scheduled. Such time-stamp can be checked on next release of sporadic task to ensure MIAT between two consecutive invocations.

5.2 Evaluation for Supporting Fixed Size Task Set

In OSE, tasks can be created in dynamic or static way. While designing the scheduler, fixed size for task set can be achieved by

1. creating OSE processes against user defined tasks using static approach. Such processes start automatically when application get started. It is illegal to kill a static process at run time. Hence, it is ensured that number of tasks in system remains fixed.

2. creating OSE processes against user defined tasks using dynamic approach. However, all such processes should be created before start of scheduler. These processes do not start automatically with application. Scheduler can start them when needed. To keep the number of tasks fixed, scheduler should not kill any process at run-time.

5.3 Evaluation for Specification of Non Functional Parameters

In OSE, only priority can be specified at time of process creation. No direct support is provided to associate WCET, period, MIAT and release time with processes.

While designing the scheduler, non-functional parameters of a task can be specified in a parameter file. This file can be read and saved into a user defined structure. Scheduler can use this structure to know time constraints for each task and applying them while scheduling the corresponding processes.
5.4 Evaluation for Fixed Priority Preemptive Scheduling

OSE provides support for fixed priority preemptive scheduling. However, EDF is not supported. The scheduler can use preemptive scheduling to implement EDF or any other priority based scheduling policy like RMS.

5.5 Evaluation for Monitoring of Execution Time Overruns and Deadline Misses

To provide support for monitoring, following approaches can be employed by the scheduler

1. Using Event Handlers: Scheduler keeps record of execution time and completion time of every task. Whenever, a task exceeds its execution time or misses the deadline, scheduler triggers an event. Event handling support of OSE can be employed to take appropriate action on triggering of each event.

2. Using Background Process for Monitoring: Scheduler keeps record of execution time and completion time of every task. However, it does not trigger events on violation of execution time or deadline. Instead, scheduler sends a signal to a background process. The background process can handle those signals to extract information and perform appropriate action.

Advantage over handler approach is that, the background process executes only when no other process is ready. Hence, it does not affect the scheduling of user defined tasks. However, it increases time between a violation and execution of relevant action against violation.

5.6 Evaluation for Supporting Absolute Delays

OSE provides API for self suspension of a process for given amount of time relative to point of invocation of API. The scheduler can use this API along with time stamping to implement absolute delays.

1. Get current absolute time using API call for time stamping. Suppose time stamping API returns $x$ time units.
2. Subtract it from desired absolute time point, \( y \), until which scheduler needs to delay itself (i.e. \( y - x \));

3. Use OSE API for suspension of a process with argument set to result of step 2.

### 5.7 Evaluation for Supporting Single Event Dependency

In OSE, a process can be blocked on any number of event or signals. Single event dependency can be achieved if the scheduler observes the self imposed restriction of not making any process in whole design to wait on more than one event or signal. This should be considered as a design constraint and not as an implementation constraint.

### 5.8 Summary of Evaluation of OSE

All desired capabilities discussed in chapter 4 can be implemented in a prototype scheduler on top of OSE scheduler. Some features can be implemented by using multiple approaches, for example, implementation of periodic tasks or monitoring mechanisms. In such cases, an approach can be selected depending upon implementation complexity and performance.
Chapter 6

Overview and Evaluation of Commonly Used RTOSes

In Chapter 3, Introduction to OSE is provided while evaluation of OSE for implementation of the scheduler is presented in Chapter 5. To get a view of implementation feasibility of the scheduler on other commonly used RTOSes, similar evaluation is performed for FreeRTOS, Windows Embedded CE and QNX.

In this chapter, section 6.1 and 6.2 presents overview of FreeRTOS and its evaluation for implementation of scheduler. Section 6.3 and 6.4 provides overview of Windows Embedded CE and summary of its evaluation. Section 6.5 and 6.6 presents overview and evaluation of QNX.

6.1 FreeRTOS

Facts presented in this section are excerpts from [23] unless stated otherwise.

FreeRTOS is a popular portable, open source, royalty free, small real-time kernel. It is ideally suited for embedded real-time applications that use small to medium size microcontrollers with 32 K bytes and 512 K bytes of flash memory and 16Kbytes and 256Kbytes of RAM.

6.1.1 Tasks

FreeRTOS organizes application as independent threads of execution, which are named as tasks. Each task is a small program in its own with an entry point and will normally run forever in an infinite loop. Priorities are assigned to these tasks which may or may not be unique. There is no restriction by
FreeRTOS on assignment of priorities.

An idle task is created automatically with priority of zero (lowest priority) and can be used for background tasks by implementing an idle hook/callback function.

Scheduling Policies

FreeRTOS provides following scheduling policies.

1. Pre-emptive scheduling
2. Cooperative scheduling

Task States

Tasks can exist in one of four states;

1. Ready
2. Running
3. Suspended
4. Blocked

Figure 6.1 shows state transitions of a task in FreeRTOS.

6.1.2 Communication and Synchronization

In FreeRTOS, communication among various tasks in an application is performed through queues. Queues are objects, which are not owned by or assigned to any particular task. A queue can hold a finite number of fixed size data items.

For synchronization among multiple tasks, in addition to queues, FreeRTOS provides binary semaphores, counting semaphores, recursive semaphores and mutexes.

6.2 Evaluation of FreeRTOS

Table 6.1 presents evaluation of FreeRTOS for desired capabilities discussed in chapter 4.
**Figure 6.1:** Task states in FreeRTOS

**Table 6.1: Evaluation of FreeRTOS**

<table>
<thead>
<tr>
<th>Desired Capability</th>
<th>Evaluation Status</th>
<th>Comments (if any)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation of periodic tasks</td>
<td>Feasible implicitly</td>
<td>Task cannot be given identity as a periodic task at the time of creation. However, periodic tasks can be implemented in form of a task which executes in an infinite loop but delays itself for a fixed period of time after every iteration.</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Desired Capability</th>
<th>Evaluation Status</th>
<th>Comments (if any)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation of Sporadic tasks</td>
<td>Feasible</td>
<td>Task cannot be given identity as a sporadic task at the time of creation. However, sporadic behavior can be modeled by making the body of a task to wait on a semaphore to start its execution. When semaphore is available, task executes its body and then again wait for semaphore. To model interrupt driven sporadic tasks, semaphore can be releases by an Interrupt Service Routine (ISR). Program driven sporadic tasks can be modeled by releasing required semaphore by another task instead of ISR.</td>
</tr>
<tr>
<td>Fixed Size of task set</td>
<td>Feasible</td>
<td>In FreeRTOS, non-functional parameters cannot be specified at the time of task creation. However, we can pass a void pointer to implementing function of a task while creating a task through xTaskCreate() API. We can use this pointer argument to hold a pointer to a structure having WCET, deadline, Period/MIAT and release time as its data members.</td>
</tr>
<tr>
<td>Specifying non-functional parameters</td>
<td>Feasible</td>
<td>In FreeRTOS tasks can be created dynamically, but it is not a requirement [23]. It is possible to create all needed tasks statically before scheduler is started.</td>
</tr>
<tr>
<td>Fixed Priority Preemptive Scheduling</td>
<td>Feasible</td>
<td>FreeRTOS provides support for fixed-priority preemptive scheduling</td>
</tr>
</tbody>
</table>

"Continued on next page"
Table 6.1 – continued from previous page

<table>
<thead>
<tr>
<th>Desired Capability</th>
<th>Evaluation Status</th>
<th>Comments (if any)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitoring</td>
<td>Feasible implicitly</td>
<td>In FreeRTOS explicit support is provided neither for monitoring of execution time and deadline nor for invocation of overrun handlers, in case WCET and/or deadline are violated. However, WCET and deadline can be monitored by obtaining the time stamp at start and end of a task through <code>xTaskGetTickCount()</code> API call, taking difference of two values and comparing it against specified WCET or deadline. If it is found that overrun occurs, a (higher priority) overrun handler task (e.g. which simply prints “overrun occurs”) can be unblocked by releasing a binary semaphore using <code>xSemaphoreGive()</code> API call.</td>
</tr>
<tr>
<td>Absolute Delay</td>
<td>Feasible explicitly</td>
<td>In FreeRTOS, absolute delays can be specified and implemented using <code>vTaskDelayUntil()</code> API function [23].</td>
</tr>
<tr>
<td>Single Event dep-</td>
<td>Feasible explicitly</td>
<td>Although, in FreeRTOS a task can be blocked on multiple events, semaphores and mutexes but this is not compulsory. A self imposed restriction can be observed in application development that all tasks should block (when needed) at the most on a single event.</td>
</tr>
<tr>
<td>endency</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6.3  Windows Embedded CE

Facts presented in this section are excerpts from [24] unless stated otherwise.

Windows embedded CE is a real-time, component based, multi threaded operating system that supports preemptive multitasking. It is designed for devices that require a minimum size, integration of multiple processor architectures and support for real-time operations. Windows CE supports four processor architectures ARM, Microprocessor without Interlocked Pipeline Stages (MIPS), SH4 and x86.

6.3.1  Threads

Execution unit in Windows Embedded CE is a thread. Each thread has its own context (stack, priority, access rights) and is executed in a process container. Each process contains at least one thread called primary thread. All process threads have a shared memory.

Scheduling

Scheduler is a component of Windows CE responsible for managing thread execution. The scheduler has following scheduling principles.

- Time sliced multitasking
- 256 priority levels
- Preemptive multitasking
- Round robin scheduling of threads with the same priority level
- One level of priority inversion is supported

6.3.2  Communication and Synchronization

For synchronization among threads and safe access of shared resources, Windows Embedded CE provides following synchronization and communication objects.

- Critical section
- Mutex
• Semaphore
• Events
• Point to point message queues

6.4 Evaluation of Windows Embedded CE

Table 6.2 presents evaluation of Windows Embedded CE for desired capabilities discussed in chapter 4.

Table 6.2: Evaluation of Windows Embedded CE

<table>
<thead>
<tr>
<th>Desired Capability</th>
<th>Evaluation Status</th>
<th>Comments (if any)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation of periodic tasks</td>
<td>Feasible implicitly</td>
<td>In Windows CE, threads cannot be given identity of periodic thread at the time of creation. However, periodic task can be modeled by a thread which performs a desired functionality in an infinite loop but suspends itself (by calling sleep function) after completing every iteration.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 6.2 – continued from previous page

<table>
<thead>
<tr>
<th>Desired Capability</th>
<th>Evaluation Status</th>
<th>Comments (if any)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation of Sporadic</td>
<td>Feasible</td>
<td>In windows CE, threads cannot be given identity of sporadic thread at the time of creation. However, to represent an interrupt-driven sporadic task, Interrupt Service Thread (IST) can be used. IST is a regular system thread that has a high enough priority for handling the task of processing a specific interrupt. In an interrupt-driven sporadic task, event on which IST is waiting, can be released by the kernel after complete execution of ISR. To represent a program-driven sporadic task, a regular thread can be used, which implements the desired functionality and then waits on an event in an infinite loop. Required event can be released by another task in application.</td>
</tr>
<tr>
<td>tasks</td>
<td>implicitly</td>
<td></td>
</tr>
<tr>
<td>Fixed Size of task set</td>
<td>Feasible</td>
<td>Threads in Windows CE can be created statically in initialization phase by using CreateThread( ) function. It is possible to develop a system with fixed number of active threads.</td>
</tr>
<tr>
<td></td>
<td>explicitly</td>
<td></td>
</tr>
<tr>
<td>Specifying non-functional</td>
<td>Feasible</td>
<td>In Windows Embedded CE, non-functional parameters cannot be specified explicitly while creating a thread. However, we can use optional argument named “lpParameter” of CreateThread function to hold a pointer to structure which contains non-functional parameters as its data members.</td>
</tr>
<tr>
<td>parameters</td>
<td>implicitly</td>
<td></td>
</tr>
</tbody>
</table>
Table 6.2 – continued from previous page

<table>
<thead>
<tr>
<th>Desired Capability</th>
<th>Evaluation Status</th>
<th>Comments (if any)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Priority</td>
<td>Feasible</td>
<td>In Windows CE, fixed priority preemptive scheduling is used with priority levels</td>
</tr>
<tr>
<td>Preemptive</td>
<td>explicitly</td>
<td>from 0 to 255 where 0 is the highest priority level. Priorities are set at a</td>
</tr>
<tr>
<td>Scheduling</td>
<td></td>
<td>thread level statically. Scheduler in Windows CE supports only one level of priority</td>
</tr>
<tr>
<td></td>
<td></td>
<td>inversion.</td>
</tr>
<tr>
<td>Absolute Delay</td>
<td>Approximation</td>
<td>In Windows CE, threads can be delayed for the duration of time relative to point</td>
</tr>
<tr>
<td></td>
<td>is feasible</td>
<td>of invocation using Sleep API but there is no support to specify an absolute delay</td>
</tr>
<tr>
<td></td>
<td></td>
<td>until a point in time. Hence, absolute delays can be only approximated.</td>
</tr>
<tr>
<td>Single Event dependency</td>
<td>Feasible</td>
<td>This is an implementation requirement. While developing system using Windows CE,</td>
</tr>
<tr>
<td></td>
<td>explicitly</td>
<td>this restriction can be easily respected by ensuring that no threads depend on</td>
</tr>
<tr>
<td></td>
<td></td>
<td>more than one event, if any.</td>
</tr>
</tbody>
</table>

6.5 QNX

*Facts presented in this section are excerpts from [25] unless stated otherwise.*

QNX operating system is ideal for real-time applications. It provides multitasking, priority driven preemptive scheduling and fast context switching, all essential ingredients of a real-time system.

QNX is remarkably flexible. Developers can scale up or down the operating system depending upon their needs. QNX has high degree of flexibility and modularity due to two principles.

- Microkernel architecture
• Message based inter process communication

6.5.1 Processes
In QNX, process is unit of execution. Life cycle of process is consisted of four phases.

1. Creation
2. Loading
3. Execution
4. Termination

Process Scheduling
In QNX, following scheduling methods are provided.

1. FIFO scheduling
2. Round-robin scheduling
3. Adaptive scheduling

Process States
In QNX, process can exist in one of following states.

1. Ready
2. Blocked
3. Held
4. Wait-blocked
5. Dead
6.5.2 Inter-Process Communication

In QNX, communication among processes can be achieved by following objects.

1. Messages: They are primarily used for synchronous communication.
2. Proxies: They are special form of messages and primarily used for event notification.
3. Signals: They are used for asynchronous communication.
4. Semaphores

6.6 Evaluation of QNX

Table 6.3, on next page, presents evaluation of QNX for desired capabilities discussed in chapter 4.
<table>
<thead>
<tr>
<th>Desired Capability</th>
<th>Evaluation Status</th>
<th>Comments (if any)</th>
</tr>
</thead>
</table>
| Implementation of periodic tasks| Feasible implicitly| In QNX, process cannot be given identity of a periodic process at the time of creation. However, periodicity can be achieved by sleep () or delay () primitives, which suspend the calling process for time specified in seconds or milliseconds. But problem is that this suspension period is over once
  - Specified real-time is elapsed
  - Or a signal is received whose action is to terminate the process or call a signal handler [20]. Hence, periodicity can be achieved using sleep () or delay () primitives provided that no signal is generated for periodic process until the suspension period is over. An alternative is to use a timer to enforce periodicity. |
| Implementation of Sporadic tasks| Feasible implicitly| In QNX, process cannot be given identity of a periodic process at the time of creation. However, interrupt-driven sporadic tasks can be implementing by attaching proxy to a process using qnx_proxy_attach () primitive and then triggering that proxy from within a specific interrupt service routine using trigger () primitive. |

Continued on next page
<table>
<thead>
<tr>
<th>Desired Capability</th>
<th>Evaluation Status</th>
<th>Comments (if any)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Size of task set</td>
<td>Feasible explicitly</td>
<td>In QNX, it can be ensured that all processes are created in a static way, and no process creates other processes in an iterative manner.</td>
</tr>
<tr>
<td>Specifying non-functional parameters</td>
<td>Non-feasible</td>
<td>In QNX WCET and other non-functional parameters cannot be specified.</td>
</tr>
<tr>
<td>Fixed Priority Preemptive Scheduling</td>
<td>Feasible explicitly</td>
<td>QNX fulfills the requirement of priority driven preemptive scheduling at the system level. However, at process levels other scheduling policies are also supported, but those scheduling policies are affective only if more than one process of same priority exists in the system.</td>
</tr>
<tr>
<td>Absolute Delay</td>
<td>Approximation is feasible</td>
<td>In QNX, relatives delay can be implemented by using delay ( ) and sleep ( ) primitives but absolute delays that are specified by a point in time can only be approximated.</td>
</tr>
<tr>
<td>Single Event dependency</td>
<td>Feasible explicitly</td>
<td>This is an implementation requirement which can be fulfilled easily by observing a self-imposed restriction that each process should be blocked and released by at most only one process/event.</td>
</tr>
</tbody>
</table>
Chapter 7

Design of The Scheduler

7.1 System Overview

The Scheduler developed in this thesis work schedules a given set of task, \( S \), for single core according to selected scheduling policy. \( S \) can contain three kind of tasks: Periodic tasks, Aperiodic tasks and Sporadic tasks. It is assumed that \( S \) is already tested for schedulability. Task parameters, such as period and execution time, are provided as parameter files to scheduler.

Scheduling policy is selectable and not fixed. The Scheduler provides support for RMS and EDF scheduling algorithms. Support for monitoring of task constraints is also provided. For example, task deadlines can be monitored by examining a log file generated by the scheduler.

There are some assumptions made about the scheduler and input task set. After providing discussion of these assumption in section 7.2, components of the scheduler are described in section 7.3. These components coordinate with each other by passing messages. Different type of messages employed in design are described in section 7.4. Priority assignment to various OSE processes in system is discussed in section 7.5. Section 7.6 presents operation of the scheduler. Chapter concludes by presenting features and limitations of design in section 7.7 and 7.8, respectively.

7.2 Assumptions

Following assumptions are made about task set.

1. **Schedulable task set**: It is assumed that input task set is already tested for schedulability. This assumption is driven by nature of CHESS project. Another module of the project tests the task set for schedulability and then provides the task set as an input to the scheduler.
2. **Single core system:** The scheduler is designed for single core, i.e. tasks are executed on one processor. Hence at any time only one task is in running state.

3. **No Inter-task communication:** It is assumed that all tasks in task set are independent. Hence, execution of any task do not depend on results generated by any other task in task set. A task can execute whenever it is ready and granted the scheduling slot.

4. **Hard deadlines for sporadic tasks and soft deadlines for aperiodic tasks:** Sporadic tasks are assumed to have hard deadlines. So it is important for them to meet their deadlines. However, soft deadlines are assumed for aperiodic tasks.

5. **FIFO ordering of sporadic and aperiodic tasks:** Sporadic tasks are placed in sporadic queue in such a way that they are supposed to meet their deadlines if they are scheduled in FIFO order. Same assumption is true about aperiodic tasks. If this assumption is not true and sporadic and aperiodic tasks are placed in random order in the queues then other approaches for selection of tasks from queue can be applied. One approach is to select a sporadic/aperiodic task which has earliest deadline.

### 7.3 System Components

The system is consisted of the following components:

1. Process Creator
2. Scheduler
3. Sporadic Queue Holder
4. Aperiodic Queue Holder
5. Monitor

#### 7.3.1 Process Creator

Task set, \( S \) to be scheduled is specified by providing two files for each task in \( S \).

1. **Parameter File:** A file with .prm extension provides task parameter. File contains one parameter at a line which are listed in following order.
• Release time
• Period
• Execution time
• Relative deadline
• Type of task

For sporadic tasks, period can be replaced with MIAT. In case of aperiodic tasks, period or MIAT is not needed. Hence it can be specified as 0.

Type of task can have four legal values: 0 for periodic tasks, 1 for sporadic tasks, 2 for aperiodic tasks and 3 for polling server.

2. **Body File**: A file with .c extension contains the body of task.

In other words, .prm file of a task contains its non-functional specification while .c file contains functional specification. Process creator reads the parameters for each task from its .prm file into a data structure, called “constraints”. It also creates a prioritized OSE process with priority level 1 against each user defined task. Following is the content from a sample .prm file of a user defined task.

```
2030 //Release time of task.
6 //Period or MIAT parameter.
2 //Execution time of task.
4 //Relative Deadline.
0 //Type of task; This is a periodic task.
```

In addition to above mentioned OSE processes for user defined tasks, four additional OSE processes are also created to implement the scheduler.

1. Scheduler Process
2. Sporadic Queue Holding Process
3. Aperiodic Queue Holding Process
4. Monitor Process
All created OSE processes have priority level of 1 except scheduler which is created with priority level 0. None of created OSE processes is started by process creator except scheduler. Task parameters and PIDs for all processes are then passed to scheduler. Figure 7.1 presents structure and functionality of process creator.

![Figure 7.1: Process creator](image)

**Structure of Task Body:**

All tasks, independent of their type and parameters, have same body structure. In a while loop, they keep waiting for their scheduling turn. Once processor is assigned to a task, it executes it body, sends the completion acknowledgment to scheduler and then again waits for its turn. Following is the pseudo-code for task body.

```plaintext
while 1
    begin
        wait for scheduling turn;
```
7.3.2 Scheduler

Scheduler is a prioritized OSE process with highest priority level of 0. After receiving “constraints” structure and PIDs of all OSE processes created by process creator, it schedules tasks according to selected scheduling algorithm. The scheduler provides options to select between RMS or EDF algorithm.

To schedule sporadic and aperiodic tasks, scheduler uses two approaches:

1. Background scheduling of sporadic and aperiodic tasks
2. Use of polling server for scheduling sporadic and aperiodic tasks

Figure 7.2 shows function performed by scheduler.

![Scheduler Diagram]

**Figure 7.2:** Scheduler
7.3.3 Sporadic Queue Holder

Sporadic queue holder is a prioritized OSE process which keeps record of released sporadic tasks by using a queue. Each element of queue contains two parameters for a sporadic task: PID corresponding to the given task and release time of sporadic task.

Whenever a new sporadic task is released, sporadic queue holder is notified. Upon receiving notification, it updates the queue by placing PID and release time for new sporadic task into queue. Queue elements are ordered in increasing order of release times of tasks. Figure 7.3 presents the sporadic queue holder.

![Figure 7.3: Sporadic queue holder](image)

7.3.4 Aperiodic Queue Holder

Aperiodic queue holder is a prioritized OSE process which keeps record of released aperiodic tasks by using a queue. Each element of queue contains two parameters relevant to periodic task: PID corresponding to given task and release time of aperiodic task.

Whenever a new aperiodic task is released, aperiodic queue holder is notified. Upon receiving notification, it updates the queue by placing PID and release time for new aperiodic task into queue. Queue elements are ordered in increasing order of release times of tasks. Figure 7.4 presents the aperiodic queue holder.
Monitor component is responsible for observing whether provided constraints for each task in task set $S$ are met or not. For example, if MIAT parameter specified in case of a sporadic task is violated then monitor records this violation and saves the information in a log file.

Whenever a task completes its execution, scheduler sends an information packet to monitor. Such packet contains start time, completion time, desired deadline, desired execution time, desired MIAT, actual execution time and actual MIAT of completed task. Monitor extracts this information from packet and saves the relevant monitoring statements in log file. Figure 7.5 shows functionality of the monitor.

7.4 System Signals

To achieve scheduling of tasks in reliable way, different kind of signals are defined and used by system. These signals play two important roles:

1. Carry required data from one component to other
2. Ensure synchronous execution of all components

These signals are described below.

1. **start_exe** sig: Start execution signal. This signal is sent by scheduler to a process to be scheduled. Target process can start execution only if it has received start_exe sig signal.
2. \textbf{comp\_sig}: Completion signal. This signal is sent as an acknowledgment to scheduler by a process which has completed its execution.

3. \textbf{aper\_update\_sig}: Update signal for aperiodic queue holder. Aperiodic tasks can be released at run-time. Whenever an aperiodic task is released, either by an interrupt process or a prioritized process, this is notified to aperiodic queue holder by sending aper\_update\_sig to it. This signal is also used as a response to scheduler by aperiodic queue holder on receiving start\_exe\_sig from scheduler.

4. \textbf{spor\_update\_sig}: Update signal for sporadic queue holder. Sporadic tasks can be released at run-time. Whenever a sporadic task is released, either by an interrupt process or a prioritized process, this is notified to sporadic queue holder by sending spor\_update\_sig to it. This signal is also used as a response to scheduler by sporadic queue holder on receiving start\_exe\_sig from scheduler.

5. \textbf{qupdate\_confirm\_sig}: Queue update confirmation signal. A confirmation signal is sent back to sender of update signal, after receiving aper\_update\_sig (in case of aperiodic queue holder) or spor\_update\_sig (in case of sporadic queue holder). Confirmation signal informs sender whether queue is updated successfully or not. In case of failure, sender can send aper\_update\_sig again with same content after waiting for a finite amount of time.

6. \textbf{monitor\_info\_sig}: Monitoring information signal. This signal is sent by scheduler to Monitor, every time a task completes its execution. Monitor uses the information contained in this signal to determine if a completed task has met its constraints, such as deadline and WCET.
7.5 Priority Assignment

In OSE, there are 32 priority levels. Priority 0 is considered highest while 31 is considered as lowest priority level. In system under discussion, process creator creates one OSE process against each task in the input task set. All such processes are assigned a priority level of 1. Similarly, sporadic queue holder process and aperiodic queue holder process have priority level of 1. However, scheduler process has priority level of 0 which is highest possible priority level.

Reason for assigning priority level 0 to scheduler is to make it non pre-emptable by any other prioritized OSE process. However, it can be pre-empted by interrupt and timer interrupt OSE processes.

As execution of all other OSE processes is controlled by scheduler, therefore they all have same priority level of 1. Monitor is a background OSE process and hence has lowest priority level. This ensures that monitoring is performed only when no task is ready and scheduler is idle. This reduces affect of monitoring on scheduling of tasks. Table 7.1 shows summary of priority assignment to OSE processes.

<table>
<thead>
<tr>
<th>Process</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheduler</td>
<td>0</td>
</tr>
<tr>
<td>Processes created against task set</td>
<td>1</td>
</tr>
<tr>
<td>Aperiodic Queue Holder</td>
<td>1</td>
</tr>
<tr>
<td>Sporadic Queue Holder</td>
<td>1</td>
</tr>
<tr>
<td>Monitor</td>
<td>Lowest Priority, Background process</td>
</tr>
</tbody>
</table>

7.6 System Operation

To understand operation of scheduler, consider Figure 7.6. As described in previous section, all OSE processes are created by Process creator but not started by it. Process creator starts the Scheduler process and passes “constraints” structure along with PIDs of all OSE processes.
7.6.1 Scheduling of Periodic Tasks

Scheduler examines the “type” parameter of all tasks to identify periodic tasks among the task set. Periodic tasks are scheduled according to specified scheduling algorithm, for example RMS.

Scheduler sends start_exe_sig to the task which has highest priority according to selected scheduling algorithm. Then scheduler waits for receiving comp_sig back from target OSE process but with a finite waiting time called “timeout”. If comp_sig is received before timeout is expired, it implies that target task has completed. Hence scheduler, schedules the next ready periodic task. If comp_sig is not received within timeout duration, then target task is preempted and scheduler schedules one of the ready higher priority tasks.

Target task on receiving start_exe_sig from scheduler, starts its execution. When execution is complete, target task sends comp_sig back to scheduler to notify that its execution is completed.

7.6.2 Scheduling of Sporadic and Aperiodic Tasks

Sporadic and aperiodic tasks can be scheduled by using one of two approaches.

Background Scheme

One approach to schedule sporadic and aperiodic tasks is to use time slots in which scheduler is idle and no periodic task is ready to run. In such case, scheduler first makes query to sporadic queue holder to find if there is any ready sporadic task in queue. This is achieved by sending start_exe_sig from scheduler to sporadic queue holder. spor_update_sig is sent back by sporadic queue holder to scheduler, indicating availability status of sporadic task. If there is a ready sporadic task, scheduler schedules it until either it completes its execution or a periodic task becomes ready. If sporadic task is completed and no periodic task is ready to run, scheduler again makes query to sporadic queue holder to find if there are any more sporadic tasks waiting in the queue.

If sporadic task queue is empty and no periodic task is ready to run, scheduler makes query to aperiodic queue holder. This is achieved by sending start_exe_sig to aperiodic queue holder. Availability status of aperiodic task is communicated back to scheduler by sending aper_update_sig back to scheduler from aperiodic queue holder. If aperiodic queue is not empty and aperiodic task at head of the queue is ready to run, scheduler schedules that
If there is no sporadic and aperiodic task to execute and scheduler is idle, it waits for arrival of next ready periodic, sporadic or aperiodic task.

Polling Server Scheme

An alternative approach to schedule sporadic and aperiodic tasks is to use polling server. Polling server is a periodic task like any other periodic task. It has a period $P_s$ and execution time $E_s$. Execution time of polling server is known as its budget.

Polling server is scheduled along with all other periodic tasks according to selected scheduling algorithm. However, when polling server gets the chance
to execute, scheduler makes query to sporadic and aperiodic queues to find if there is any ready sporadic or aperiodic task. If sporadic or aperiodic task is ready to run, scheduler schedules that task and budget of polling server keeps declining per unit time. If sporadic or aperiodic task completes its execution before budget is expired, scheduler picks next ready sporadic or aperiodic task to execute. This sequence continues until either there is no sporadic or aperiodic task or budget of server is expired or a higher priority periodic task becomes ready to execute.

At start of each period of polling server, its budget is set equal to its execution time. If at that time point, no sporadic or aperiodic task is ready to run then budget immediately declines to zero. Otherwise budget decrease one per unit time.

**Interrupt and Program Driven Sporadic and Aperiodic Tasks**

To schedule a sporadic task, it is necessary that its corresponding PID and release time is placed in sporadic process queue maintained by sporadic queue holder. This can be achieved by sending a spor_update_sig to sporadic queue holder containing release time and PID of OSE process corresponding to the task. Signal, spor_update_sig, can be sent to sporadic queue holder either by an interrupt OSE process or a prioritized OSE process. In first case, target sporadic task becomes interrupt driven while in second case it behaves as program driven sporadic task.

7.6.3 Monitoring

On completion of a task, independent of its type, scheduler sends monitor_info_sig to Monitor. Monitor is background OSE process. Hence, it can execute only when there is no periodic, sporadic or aperiodic task ready to run.

Monitor continuously checks its input message queue for monitor_info_sig. This message carries following information to Monitor regarding completed task.

- start time of sender task
- completion time of sender task
- specified deadline parameter for sender task
- specified MIAT parameter for sender task
- specified execution time for sender task
Monitor uses this information to make decision if completed task has met its parameters or violated them. In any case, Monitor records the information in a log file.

Same discussion is valid for achieving interrupt and program driven behavior for aperiodic tasks.

### 7.6.4 Summary

Operation of the scheduler is summarized by sequence diagram of Figure 7.7. In this diagram, task set consists of two periodic tasks $T_1$ and $T_2$, one sporadic task $T_3$ and an aperiodic task $T_4$.

![Sequence diagram to demonstrate operation of the scheduler](Image)

**Figure 7.7**: Sequence diagram to demonstrate operation of the scheduler

Figure 7.7 shows a valid sequence of execution when RMS is used as a
scheduling policy for periodic tasks, while sporadic and aperiodic tasks are scheduled using background scheme. Sequence diagram is easy to understand if following parameters are considered for each task. Please note that constraints are specified using the convention of .prm files mentioned in section 7.3.1.

Periodic task: \( T_1(0, 12, 3, 8, 0) \)
Periodic task: \( T_2(0, 4, 1, 3, 0) \)
Sporadic task: \( T_3(0, 15, 2, 6, 1) \)
Aperiodic task: \( T_1(0, 0, 1, 6, 2) \)

As is evident from sequence diagram, monitor gets the chance to execute only when no other process is in ready state.

### 7.7 Design Features

The scheduler designed in this thesis work has following salient features.

1. **User friendly creation of tasks:** A task is created by specifying two files:
   - Parameter file (.prm) containing constraints of tasks such as release time, period, execution time, relative deadline and type of task.
   - Body file (.c) containing body of task to be executed.

   User does not need to worry about implementation details and monitoring of different task constraints.

2. **Monitoring:** Design supports monitoring of task constraints as they execute on processor. If some constraints are violated and actual behavior is deviated from expected one, then these facts are recorded in a log file.

3. **Scalability:** The scheduler is scalable in the sense that it can support any number of tasks in the task set. Only limitation is that task set should be already tested for schedulability.

4. **Flexibility:** The scheduler provides flexibility at two levels
   - Two scheduling policies, EDF and RMS, are supported. Scheduling policy can be easily selected by using “constant.h” file which contains different system constants.
• Two schemes are supported for scheduling of sporadic and aperiodic tasks: Background scheme and Polling scheme. User can select desired scheme through “constants.h” file.

5. **Interrupt and program driven sporadic and aperiodic tasks:** Sporadic tasks can be released by any OSE interrupt process through its interrupt service routine as well as by any prioritized OSE process in the system. Same support is provided for aperiodic tasks.

### 7.8 Design Limitations

The scheduler suffers from some limitations which are listed below.

1. **Scheduling overhead dependent on size of task set:** Every time a task completes its execution or it is preempted, scheduler consumes a finite amount of time in finding the next task to be executed and time duration assigned to it for execution. Time consumed in making this decision is dependent on size of task set. As size of task set increase, scheduling overhead also increases.

2. **Accumulation of scheduling overhead:** As scheduler consumes non-zero time between scheduling of two consecutive tasks, scheduling overhead gets accumulated over time. Hence deviation of generated schedule from expected one becomes significant over time. Lower priority tasks suffer more from this accumulation of scheduling overhead as compared to higher priority tasks. However, when a higher priority task preempts a lower priority task, it brings the schedule back to desired time point. In this way, accumulation of overhead does not grow to infinity. Therefore, a point is never reached where all tasks start missing their deadlines.

3. **Starvation suffered by Monitor:** As Monitor is implemented as a background process, it can suffer from starvation to an extent that it never gets the chance to execute. For example, if task set is consisted of task with such periods and execution times that scheduler remains always busy in scheduling the tasks and have no idle time, then Monitor will never be scheduled. Severity of problem can be reduced by implementing Monitor as a prioritized process instead of background process. In this way, it will be scheduled by the scheduler along with other prioritized processes. However, this will affect the scheduling of user defined tasks.
Chapter 8
Implementation and Results

The scheduler, described in chapter 7, is implemented on OSE5.5 soft kernel. This chapter provides pseudo-code of implementation in section 8.1. Section 8.2 provides description of log files generated by the scheduler. Section 8.3 verifies the result of implementation by comparing the schedules generated by the scheduler with expected time behavior of a given set of tasks.

8.1 Implementation

As discussed in chapter 7, process creator is a component which reads parameter of the user defined tasks from .prm files and save them in an array of structure “constraints”. Following subsection describes this data structure.

8.1.1 Constraints Data Structure

This data structure has two types of data members: first set of data members are used to hold parameters of the task while second set of data members is used in monitoring of time constraints. Listing 1 describes these data members. Comments provide a short description about each data member.

Listing 1: Constraints Data Structure

```c
struct constraints
{
    PROCESS PID; //Process ID contains ID of OSE Prioritized
                   //process created against a user defined task.
    OSTICK Rel,   //Contains specified Release time of first
                  //first
```
instance of user defined Task.

Period_or_MIAT, //Contains specified Period of user defined
//task (in case if user defined task is
//periodic) or MIAT (in case if user defined
//task is sporadic.)

Exe_Time, //Contains specified WCET of user defined task.

Rel_DL, //Contains specified Relative Deadline of user
//defined task.

N_Rel, //Used to contain time point at which user
//defined task is expected to be released
//next time.

Rem_Exe_Time; //Used to keep record of remaining execution
//time of user defined task. Initially it is
//equal to WCET of task and then decreases
//to zero until task is completed.

int type; //Used to contain type of task specified
//by user:
//It can have 4 legal values.
//0: Periodic task
//1: Sporadic task
//2: Aperiodic task
//3: Polling Server

OSTICK start_time, //Used to contain time stamp at which a task
//is actually scheduled by the scheduler.

completion_time, //Used to contain time stamp at which a task
//completes its execution.

actual_exe_time, //Used to contains actual execution time
//consumed by the task. It may or may not
//be equal to specified WCET.

prev_Rel, //Used to keep record of release time of last
//instance of the task (one preceding the
//current active instance).

current_Rel; //Used to contain release time of current
//active instance of the task.

};
8.1.2 Pseudo-Code of Process Body

All processes created against user defined tasks have same general body structure, independent of type of the corresponding task. Process body is a piece of code which is written by user for a particular functionality. For example, body of a process may read the data from peripherals and apply some algorithm on it. Listing 2 presents pseudo-code for body of one of such processes.

**Listing 2:** Process Body

```plaintext
while 1
  begin
    wait for start_exe.sig;    //Wait until start_exe.signal is received
    //from scheduler and then start execution of
    //process body.
    statement 1;               //First statement of user defined body
    //of process.
    statement 2;
    statement 3;
    statement n;              //Last statement of user defined body of
    //process.
    Send comp_sig back to scheduler;  //Send comp_signal back to
    //scheduler to acknowledge completion of
    //process body.
  end
```

8.1.3 Pseudo-Code of Process Creator

Process Creator reads N .prm files containing parameters for N tasks. It also reads N .c files containing code for body of task. Listing 3 shows pseudo-code for process creator.

**Listing 3:** Process Creator

```plaintext
struct constraints stock[N];  //Define an array named “stock”
  //of type “constraints” to contain
  //timing parameters of all tasks, in
```
for $i := 1$ to $N$ step 1
begin
    create_process($P_i$);
    //Create a prioritized OSE
    //process against user defined
    //task $T_i$ in task set.
    stock[$i$] = File$_i$.prm
    //Read parameters of task
    //from file File$_i$.prm
    //into $i$th element of array “stock”.
end
scheduler ← stock
//Send array “stock” to the scheduler.

8.1.4 Pseudo-Code of Sporadic Queue Holder

Sporadic queue holder contains PID and release times of processes corresponding to ready sporadic tasks. Listing 4 describes implementation of sporadic queue holder.

Listing 4: Sporadic Queue Holder

Spor.Q[L]
//Make a queue of length L to
//hold PIDs and release times
//of sporadic tasks.
while 1
begin
    receive(start_exe_sig);
    //Wait until start_exe_sig
    //signal is received from the scheduler.
    do
        begin
            receive_w_tmo(0, spor_update_sig)
            //Check the signal
            //queue for spor_update_sig signal
            //and read this signal from
            //queue, if available.
            Spor.Q[L] ← spor_update_sig.PID
            //Extract the PID of sporadic
            //process from signal and
            //place it in queue.
            Spor.Q[L] ← spor_update_sig.release_time
while $\text{Signal\_Queue! = empty}$

//If signal queue was not empty
//receive the next signal from queue
//otherwise, break the do loop.

alloc($\text{Sig}$);

//Allocate a signal of type
//spor_update_sig.

if $\text{Spor\_Q! = Empty}$

begin

$\text{Sig}\_\text{PID} \leftarrow \text{Spor\_Q}_0\_\text{PID}$

//then pick the PID

$\text{Sig}\_\text{release} \_\text{time} \leftarrow \text{Spor\_Q}_0\_\text{release} \_\text{time}$

//and release time from head of
//queue and place in signal $\text{Sig}$.

$\text{Spor\_Q}_i \leftarrow \text{Spor\_Q}_{i+1}$

//Update the queue of sporadic
//processes, after removing first
//element from queue.

end

else

begin

$\text{Sig}\_\text{PID} \leftarrow 0$

//initialize all fields
//of signal with zero.

$\text{Sig}\_\text{release} \_\text{time} \leftarrow 0$

end

scheduler$\leftarrow \text{Sig}$

//Send the signal $\text{Sig}$ to the scheduler.

end

8.1.5 Pseudo-Code of Aperiodic Queue Holder

Aperiodic queue holder contains PID and release times of processes corresponding to ready aperiodic tasks. Listing 5 describes implementation of aperiodic queue holder.

Listing 5: Aperiodic Queue Holder
Aper_Q[L] //Make a queue of length L to
//hold PIDs and release times
//of aperiodic tasks.

while 1
begin
receive(start_exe_sig); //Wait until start_exe_sig
//signal is received from the scheduler.
do
begin
receive_w_tmo(0,aper_update_sig) //Check the signal
//queue for aper_update_sig signal
//and read this signal from
//queue, if available.

Aper_Q[L] ← aper_update_sig.PID //Extract the PID of aperiodic
//process from signal and
//place it in queue.

Aper_Q[L] ← aper_update_sig.release_time //Extract the release time of aperiodic
//process from signal and
//place it in queue.
end
while Signal_Queue! = empty //If signal queue was not empty,
//receive the next signal from queue
//otherwise, break the do loop.
alloc(Sig); //Allocate a signal of type
//aper_update_sig.
if Aper_Q! = Empty //If queue of aperiodic processes
//is not empty
begin
    Sig.PID ← Aper_Q[0].PID //then pick the PID
    Sig.release_time ← Aper_Q[0].release_time
    //and release time from head of
    //queue and place in signal Sig.
    Aper_Q = Aper_Q[1] //Update the queue of aperiodic
    //processes, after removing first
    //element from queue.
end
else //If queue of aperiodic processes
    //is empty then
begin
Sig.PID ← 0 //initialize all fields of signal with zero.
Sig.release_time ← 0
end
scheduler ← Sig //Send the signal Sig to the scheduler.

8.1.6 Pseudo-Code of Monitor

Monitor component receives monitor_info_sig from scheduler on completion of task, extracts the information from signal and then determines if a task has violated its constraints or not. Results are written in a monitoring log file. Listing 6 describes the implementation of monitor process.

Listing 6: Monitor

while 1
begin
Sig = receive(monitor_info_sig); //Wait until monitor_info_sig signal is received from scheduler in Sig.
V1, V2, ...Vn ← Sig //Extract all information from signal into variables.
Analyze(V1, V2, ...Vn) //Analyze the information.
“Monitor_log.txt” ← Result_analysis //Write results of analysis in monitoring log file.
end

8.1.7 Pseudo-Code of Scheduler

Scheduler is core component of whole design. After receiving array of constraints structure from process creator, it schedules the set of user defined tasks according to selected policy. Periodic tasks can be scheduled either using EDF or RMS. Sporadic and aperiodic tasks are scheduled using background scheme or polling server scheme. Listing 7 provides pseudo-code for scheduler when background scheme is used for scheduling of sporadic and aperiodic tasks with RMS used for scheduling of periodic tasks.
Scheduler ← Stock

//Receive array of constraints structure,
//called “stock”, from Process Creator.

PROCESS PID_Schd;

//Variable to contain PID of process
//to be scheduled by the scheduler.

OSTICK Timeout;

//Variable to contain time value.

while 1

begin

PID_Schd=NULL;

//Initially set the variable equal to NULL.

if Scheduling_policy == RMS

begin

PID_Schd=Next_Proc()

//Find the PID of periodic process to be
//scheduled next according to RMS.

Timeout=Find_timeout()

//Find available time interval for
//scheduling, after which a higher priority
//periodic task will become ready.

end

if PID_Schd == NULL

begin

Spor_QHolder←start_exe_sig

//Make query to Sporadic Queue holder.

Sig1 = receive(spor_update_sig)

//Wait until

//spor_update_sig signal
//is received by the scheduler
//from sporadic queue holder.

if Sig1.PID! = 0

begin

PID_Schd=Sig1.PID

//Get PID of Process
//to be scheduled in PID_Schd.

schedule(PID_Schd)

//Schedule the process.

end

elsif Sig1.PID == 0

begin

Sig2 = receive(aper_update_sig)

//Wait until

//aper_update_sig signal
//is received by the scheduler
//from aperiodic queue holder.

if Sig2.PID! = 0

begin

//If aperiodic queue is not empty,

end

this

else

begin

Sig3 = receive(othr_update_sig)

//Wait until

//othr_update_sig signal
//is received by the scheduler
//from other queue holder.

if Sig3.PID! = 0

begin

//If other queue is not empty,

end

end


begin
  PID_Schd=Sig2.PID  //Get PID of Process
  //to be scheduled in PID_Schd.
schedule(PID_Schd)   //Schedule the process.
end

elsif Sig2.PID == 0
  //If aperiodic queue is empty,
delay();              //Wait until next periodic task
  // becomes ready.
end

---

8.2 Formatting of Log Files

The scheduler produces two log files for a given task set.

1. **Scheduler_log.txt**: This log file contains the listing of time points at which tasks in task set are scheduled for execution or they complete their execution or they are preempted. Scheduler_log.txt is written every time a task is scheduled for execution, completed or preempted.

2. **Monitor_log.txt**: This log file contains the listing generated by Monitor. Monitor_log.txt file is written only when a task is completed.

Following subsections describe formatting of both log files and their interpretation. Log files are generated for following task set.

- **T_1(0, 10, 2, 5, 3)**: Sporadic Task with release time=0, period=10, WCET or Budget=2, Relative Deadline=5, Task type=3 (polling server).
- **T_2(0, 5, 2, 4, 0)**: Periodic Task with release time=0, period=5, WCET=2, Relative Deadline=4, Task type=0 (periodic task).
- **T_3(0, 5, 2, 4, 1)**: Sporadic Task with release time=0, MIAT=5, WCET=2, Relative Deadline=4, Task type=1 (sporadic task). Two instances of Sporadic task T_3 are released at time 0;
- **T_4(0, 0, 2, 7, 2)**: Aperiodic Task with release time=0, period= 0 (don’t care), WCET=2, Relative Deadline=7, Task type=2 (aperiodic task).

Aperiodic tasks are scheduled using RMS. Polling server is used for scheduling of aperiodic and sporadic tasks.
8.2.1 Formatting of Scheduler.log.txt File

Listing 8 is an excerpt from a Scheduler.log.txt file. Comments provide a short description of listing.

Listing 8: Scheduler.log.txt File

********New Session of Scheduler****** //indicates that a new session of scheduling, possibly with new task set is started

- task PID=65603 //task modeled by process with PID 65603
  - Scheduled for 5 ticks at ticks=8768 //is scheduled at 8768 tick

- task PID=65603 //task modeled by process with PID 65603
  - Completed at ticks=8770 //completes its execution at 8770 ticks

- task PID=65604 //task modeled by process with PID 65604
  - Scheduled with budget= 2 ticks at ticks=8770 //is scheduled at 8770 tick
    //with budget of 2 ticks for polling server

- task PID=65604 //task modeled by process with PID 65604
  - Not completed at ticks=8772 //is preempted at 8772 ticks

- Remaining Execution Time in ticks=1 //Remaining execution time of preempted task is 1 tick

- task PID=65603 //task modeled by process with PID 65603
  - Scheduled for 5 ticks at ticks=8773 //is scheduled at 8768 tick

8.2.2 Formatting of Monitor.log.txt File

Listing 9 is an excerpt from a Monitor.log.txt file. Comments provide a short description of listing.

Listing 9: Monitor.log.txt File

********New Session of Monitor****** //indicates that a new session of monitoring, possibly with new task set is started

- PID =65603 //Monitoring for task modeled by process with PID 65603 is started
- Type of task =0 //Task is periodic task
- start time in ticks=8768 //Task is started at 8768 ticks
- specified deadline in ticks=8772 //Specified deadline of task is 8772 ticks
8.3 Verification of Results

Following sections verify the schedules generated by the scheduler in two cases:

1. Case 1: Scheduling Policy for periodic tasks is RMS, while sporadic and aperiodic tasks are scheduled using background scheme.

2. Case 2: Scheduling Policy for periodic tasks is RMS, while sporadic and aperiodic tasks are scheduled using polling server scheme.

8.3.1 Case 1:

Schedule is generated by the scheduler for following task set.

- $T_1(0, 10, 2, 5, 0)$; Periodic Task with release time=0, period=10, WCET=2, Relative Deadline=5, Task type=0 (periodic task).

- $T_2(0, 5, 2, 4, 0)$; Periodic Task with release time=0, period=5, WCET=2, Relative Deadline=4, Task type=0 (periodic task).

- $T_3(0, 5, 2, 4, 1)$; Sporadic Task with release time=0, MIAT=5, WCET=2, Relative Deadline=4, Task type=1 (sporadic task). Two instances of Sporadic task $T_3$ are released at time 0;

- $T_4(0, 0, 2, 7, 2)$; Aperiodic Task with release time=0, period= 0 (don’t care), WCET=2, Relative Deadline=7, Task type=2 (aperiodic task).

Expected Schedule

Figure 8.1 shows the expected schedule for given set of tasks.
Figure 8.1: Expected schedule for task set given in case 1

Schedule Generated by The Scheduler

Figure 8.2 shows the schedule generated by the scheduler. Generated schedule is given in Appendix A in form of a listing. Scheduler releases first task at 3438 system ticks. To make this schedule more easy to compare with one in Figure 8.1, subtraction of 3438 ticks is performed at every time point.

Comparison

In schedule generated by the scheduler, as shown in Figure 8.2, second invocation of sporadic task \( T_3 \) consumes less execution time than specified WCET. Hence, it affects the scheduling time point for aperiodic task \( T_4 \) which is scheduled at time instant 9.7 instead of 17 as compared to Figure 8.1.

8.3.2 Case 2:

Schedule is generated by the scheduler for the following task set.

- \( T_1(0, 10, 2, 5, 3) \): Sporadic Task with release time=0, period=10, WCET or Budget=2, Relative Deadline=5, Task type=3 (polling server).
Figure 8.2: Scheduler generated by the scheduler for task set given in case 1

- $T_2(0, 5, 2, 4, 0)$; Periodic Task with release time=0, period=5, WCET=2, Relative Deadline=4, Task type=0 (periodic task).

- $T_3(0, 5, 2, 4, 1)$; Sporadic Task with release time=0, MIAT=5, WCET=2, Relative Deadline=4, Task type=1 (sporadic task). Two instances of Sporadic task $T_3$ are released at time 0;

- $T_4(0, 0, 2, 7, 2)$; Aperiodic Task with release time=0, period= 0 (don’t care), WCET=2, Relative Deadline=7, Task type=2 (aperiodic task).

Expected Schedule

Figure 8.3 shows the expected schedule for given set of tasks.

Schedule Generated by The Scheduler

Figure 8.4 shows the schedule generated by the scheduler for task set given in case 2. Generated schedule is given in Appendix B in form of a listing.
Figure 8.3: Expected schedule for task set given in case 2 with budget consumption and repletion

Sporadic and aperiodic tasks are scheduled using polling scheme. Task $T_1$ act as polling server with maximum budget equal to 2.

Comparison

In Figure 8.4, sporadic task $T_3$ is scheduled at time 3 instead of 2. This is because of scheduler overhead. As scheduling of task $T_3$ is delayed by one time unit, it affects scheduling time of all waiting sporadic and aperiodic tasks.
Figure 8.4: Schedule generated by the scheduler for task set given in case 2 with budget consumption and repletion
Chapter 9

Conclusions and Future Work

9.1 Conclusion

The scheduler, presented in chapter 7 and 8 is capable of creating OSE processes with functional and non functional parameters defined by user. Scheduling is performed with target of meeting all specified constraints. If some constraints are not met, then monitor is invoked to take appropriate action (like logging the violation statistics).

Schedules generated by the scheduler deviate from expected behavior as demonstrated by comparison of Figure 8.4 with Figure 8.3. Potential reasons of deviation are discussed below with suggested improvements.

9.1.1 Scheduling Overhead

Scheduler consumes a finite amount of time in making selection of next task to be scheduled. Time consumed by scheduler, reduces the available time duration for scheduling of a task. Suppose that a higher priority task $T_1$ completes it execution at time instant $t_1$, with next release scheduled at time point $t_2$. Scheduler can allocate CPU time of $t_2 - t_1$ units to a lower priority task $T_2$ with execution time equal to $t_2 - t_1$. However, scheduler consumes $\Delta$ units of time in making selection of task $T_2$ from the task set. Hence, time interval available for scheduling of task $T_2$ is only $t_2 - t_1 - \Delta$. As a result, Task $T_2$ misses its deadline. It also affects scheduling of all other tasks having priority lower than task $T_2$. This scheduling overhead accumulates over time. As a result, deviation of generated schedule from expected one can be unacceptable.

Scheduling overhead can be reduced by improving implementation of queue management in aperiodic and sporadic queue holders. Current implementation uses FIFO order for insertion and deletion of elements from
queue. Time consumed in updating the queue every time an element is removed, can be significantly reduced by implementing circular queues. This will also reduce overall scheduler overhead. Hence, less number of tasks will miss their deadlines.

9.1.2 Resolution of Timer

Resolution level of timer is critical in scheduling tasks. Whenever scheduler allocate $\omega$ unit of time to a task, it gets time stamp before suspending itself. When suspension time, $\omega$, is over scheduler returns from the task and gets the second time stamp. Two time stamps values are used to keep record of remaining execution time of task. However, it is possible that scheduled task get preempted just before its completion. In this situation, usage of second time stamp in calculating remaining execution time will lead to incorrect result. Lower the resolution of timer, more difference between actual and calculated value of remaining execution time. This can cause deviation of generated schedules from expected behavior.

Problem can be solved by setting length of system tick to minimum of 1 ms. However, while implementing design on soft kernel tick length can not be reduced lower then 10 ms. Therefore, schedules generated by the scheduler suffer more from deviation.

9.1.3 Preemption by System Processes and Interrupts

The scheduler is implemented on top of OSE scheduler as prioritized process. Although it is treated as a highest priority prioritized process in system, it always has lower priority than interrupt and timer interrupt processes. Furthermore, the scheduler can be preempted by OSE system processes. These preemption are another source of deviation.

Situation can be improved if scheduler is implemented by modifying OSE kernel, instead of implementing as a prioritized process.

9.2 Future Work

This section describes a set of future tasks that can improve performance of the scheduler.

1. The scheduler uses either polling scheme or background scheme for scheduling sporadic and aperiodic tasks. As higher priority is assumed for sporadic tasks, aperiodic tasks are scheduled only if there is no ready
sporadic task. This increases response time of aperiodic tasks. Response time can be reduced by scheduling aperiodic and sporadic tasks with different schemes. Polling server can be used to schedule sporadic tasks, while background scheme can be dedicated to schedule aperiodic tasks. With this design, aperiodic tasks can be scheduled whenever no periodic task or polling server is ready to execute. Scheduling of aperiodic task is not delayed until completion of all sporadic tasks.

2. Current design uses FIFO order to select among multiple ready sporadic tasks in the queue. Same is the case for aperiodic tasks. Deadline miss rate can be reduced by making selection dependent on deadlines. In other words, EDF can also be applied on queue of sporadic and aperiodic tasks.

3. Current design places the sporadic tasks and aperiodic tasks in two separate queues, assuming that sporadic tasks are more critical than aperiodic ones. In situation where sporadic and aperiodic tasks are equally critical, a single queue can be used. Tasks waiting in queue can be ordered with respect to their deadlines or release times.

4. The scheduler does not provide support for priority inheritance or priority ceiling protocol. Problem of priority inversion can be reduced by providing support for these protocols.

5. The scheduler supports EDF and RMS as scheduling policies. Other scheduling algorithms can be supported to increase usability of the scheduler.
Bibliography


[22] Dr. Radek Pelnek. Class Lecture, Topic “Scheduling of Aperiodic Tasks” IA158, Department of Information Technologies, Faculty of Informatics, Masaryk University, Czech Republic, Spring 2011.


Appendix A

Listing for schedule generated by the scheduler for task set given in section 8.3.1

Scheduling Policy: RMS
Scheduling scheme for sporadic and aperiodic tasks: Background Scheme
T1 (0,10,2,5,0) PID: 65596
T2 (0,5,2,4,0) PID: 65595
T3 (0,5,2,4,1) PID: 65597
T4 (0,0,2,7,2) PID: 65598
// ————Results———
task 65596 is scheduled at 3438 ticks for 5 ticks
  task 65596 is completed 3440 ticks
  task 65595 is scheduled at 3440 ticks for 3 ticks
  task 65595 is completed 3442 ticks
  task 65597 is scheduled at 3442 ticks for 1 ticks
  task 65597 is not completed at 3443 ticks
  PID: 65597 rem Exe Time=1
  task 65596 is scheduled at 3443 ticks for 5 ticks
  task 65596 is completed 3445 ticks
  task 65597 is resumed at 3445 ticks for 3 ticks
  task 65597 is completed 3446 ticks
  task 65597 is scheduled at 3447 ticks for 1 ticks
  task 65597 is completed 3447 ticks
  task 65598 is scheduled at 3447 ticks for 1 ticks
  task 65598 is not completed at 3448 ticks
  PID: 65598 rem Exe Time=1
  task 65596 is scheduled at 3448 ticks for 5 ticks
task 65596 is completed 3450 ticks
task 65595 is scheduled at 3450 ticks for 3 ticks
task 65595 is completed 3452 ticks
task 65598 is resumed at 3452 ticks for 1 ticks
task 65598 is not completed at 3453 ticks
PID: 65598 rem Exe Time=1
task 65596 is scheduled at 3453 ticks for 5 ticks
task 65596 is completed 3455 ticks
task 65598 is resumed at 3455 ticks for 3 ticks
task 65598 is completed 3456 ticks
task 65596 is scheduled at 3458 ticks for 5 ticks
task 65596 is completed 3460 ticks
task 65595 is scheduled at 3460 ticks for 3 ticks
task 65595 is completed 3462 ticks
task 65596 is scheduled at 3463 ticks for 5 ticks
task 65596 is completed 3465 ticks
task 65596 is scheduled at 3468 ticks for 5 ticks
task 65596 is completed 3470 ticks
task 65595 is scheduled at 3470 ticks for 3 ticks
task 65595 is completed 3472 ticks
task 65596 is scheduled at 3473 ticks for 5 ticks
task 65596 is completed 3475 ticks
task 65596 is scheduled at 3478 ticks for 5 ticks
task 65596 is completed 3480 ticks
task 65595 is scheduled at 3480 ticks for 3 ticks
task 65595 is completed 3482 ticks
task 65596 is scheduled at 3483 ticks for 5 ticks
task 65596 is completed 3485 ticks
task 65596 is scheduled at 3488 ticks for 5 ticks
task 65596 is completed 3490 ticks
task 65595 is scheduled at 3490 ticks for 3 ticks
task 65595 is completed 3492 ticks
task 65596 is scheduled at 3493 ticks for 5 ticks
task 65596 is completed 3495 ticks
task 65596 is scheduled at 3498 ticks for 5 ticks
task 65596 is completed 3500 ticks
Appendix B

Listing for schedule generated by the scheduler for task set given in section 8.3.2

Scheduling Policy: RMS
Scheduling scheme for sporadic and aperiodic tasks: Polling Server Scheme
T1 (0,10,2,5,3) PID: 65596
T2 (0,5,2,4,0) PID: 65595
T3 (0,5,2,4,1) PID: 65597
T4 (0,0,2,7,2) PID: 65598

//—Results—

<table>
<thead>
<tr>
<th>Task</th>
<th>Start Time</th>
<th>Execution Time</th>
<th>Completion Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>3004</td>
<td>5</td>
<td>3006</td>
</tr>
<tr>
<td>T2</td>
<td>3006</td>
<td>2</td>
<td>3008</td>
</tr>
<tr>
<td>T3</td>
<td>3016</td>
<td>1</td>
<td>3017</td>
</tr>
<tr>
<td>T4</td>
<td>3024</td>
<td>5</td>
<td>3029</td>
</tr>
</tbody>
</table>

PID: 65596
PID: 65597

88
task 65596 is completed 3026 ticks
task 65597 is resumed at 3026 ticks with budget= 2 ticks
task 65597 is completed 3027 ticks
task 65598 is scheduled at 3027 ticks with budget= 1 ticks
task 65598 is not completed at 3028 ticks
PID: 65598 rem Exe Time=1

89