Institute of Software Technology
Department of Programming Languages and Compilers
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master Thesis Nr. 3556

Simulation of Multi-core Scheduling in Real-Time Embedded Systems

Md. Golam Hafiz Khan

Course of Study : INFOTECH

Examiner : Prof. Dr. rer. nat./HarvardUniv. Erhard Plödereder
Supervisor : Dipl.-Inf. Mikhail Prokharau

Date of Submission : April 7, 2014

CR-Classification : D.4.1, D.4.8, C.4.d, I.6.8.i

Abstract

In real-time systems the correctness of a system depends not only on the logical correctness of the running program but also on the time at which the logically correct output is produced. Therefore, in such a system it is necessary to provide the right computational result within a strict time limit called the deadline of a task. In hard real-time systems the deadline of a task must not be missed, whereas in soft real-time systems it can be missed occasionally. In recent years the trend has been observed which shows a shift from single-core to multi-core architectures for real-time systems. The main point of this thesis is to study a few promising multi-core scheduling algorithms both from the partitioned and the global approaches to multi-core scheduling and implement some of them into the existing simulation software. To represent the partitioned approach, Partitioned EDF has been implemented with the capability of specification of a resource access protocol for each core. The partitioned approach requires heuristics for task partitioning, the problem known to be NP-hard in the strong sense. For this reason, the implementation of Partitioned EDF requires manual task partitioning of the system in order to be able to utilize the maximum processing power. Proportionate Fair abbreviated as Pfair is the only known optimal way to schedule a set of periodic tasks on multi-core systems that falls into the global approach of multi-core scheduling. Therefore, to represent the global approach, several variants of Pfair scheduling algorithm have been selected for the implementation into the existing system. To be truly useful in practice, a real-time multi-core scheduling algorithm should support access to shared resources using some resource access protocol. For this reason, the Flexible Multiprocessor Locking Protocol abbreviated as FMLP has been studied and implemented to simulate shared resource access on multi-core systems. This resource access protocol can be used by scheduling algorithms representing both the partitioned and global approaches, but it only supports such variants of those algorithms which allow non-preemptive execution. A variant of Global EDF termed Global Suspendable Non-preemptive EDF was implemented prior to implementing FMLP. The existing simulator provided a set of single-core and some basic multi-core scheduling algorithms for scheduling real-time task sets. No schedulability analysis was implemented in the previous work. So, as part of this thesis, the schedulability analysis for single core scheduling algorithms has been implemented. A schedulability analysis for the partitioned approach of multi-core scheduling has also been provided for systems where a single-core scheduling algorithm runs on each partition. The updated simulation software also supports self-suspension of tasks for a specified duration.

Acknowledgement

Foremost, I would like to thank Prof. Dr. rer. nat./Harvard Univ. Erhard Plödereder for the opportunity to do my master's thesis at the Department of Programming Languages and Compilers, Institute of Software Technology, University of Stuttgart.

Many thanks to my supervisor, Dipl.-Inf. Mikhail Prokharau for his enthusiasm, wisdom, cordial supervision and valuable advice that helped me a lot throughout my thesis work. I am earnestly grateful to my supervisor for the fruitful discussions, providing significant guidelines for solving difficult problems in an easier way, important corrections throughout writing of my thesis and educating me how to be able to write.

I also want to thank the people who work at this department and who provided instant support for any difficulty I faced.

I am very grateful to my father and mother for the continuous support starting from the childhood to today. If not for their encouragement, care, love, passion, and their aimed assistance, I would not be here writing my master's thesis.

Finally, I would like to finish by expressing my acknowledgement and love to all of my friends for their support in all the ways during my master's study.

Contents

1 | Introduction

A *real-time* system is a system where the correctness of the system not only depends on the *logical* correctness of the output but also on the time when the correct output is produced. Thus, a *real-time* system has two concepts of correctness: *logical* and *temporal* [55]. The logical correctness means that the produced output is correct and the temporal correctness means that the system meets the timing constraint, in other words the output is produced at the right time [55].

Nowadays, the computational complexity of real-time applications is on the increase. Thus, computational requirements for systems using them grow very fast. Examples of such complex systems are automated tracking systems, teleconference systems, etc. These applications have to follow some timing constraints in order to ensure performance, responsiveness and safety. The processing requirements of these systems may exceed the capacity of a single processor and thus the necessity for multi-core processors has come to the front. Additionally, multiprocessors are more cost-effective than a single processor of the same processing speed [57].

The main goal of this thesis work is to extend the existing simulator by Munk [41] to provide a few additional multiprocessor scheduling policies. This thesis work also deals with the schedulability analysis of single core scheduling policies and also of multi-core partitioned policies where single core scheduling algorithms are used for each core. A task may suspend itself while accessing resources or wait for an event to occur. A task may need to execute non-preemptively in the preemptive scheduling algorithms where a higher priority task is in ready state. This thesis also discusses task self-suspension and non-preemptive execution where a task may contain non-preemptive sections for execution.

1.1 Motivation

In May 2004, Intel canceled the next version of the Pentium P4 processor named as *Tejas* because of excessive power consumption and started to move towards multiprocessor systems [22]. Therefore, to solve the problem of high power consumption by single core processors and to increase the processing speed, research has been started to move away from increasing the computational speed of a single processor and increase the number of cores on a single chip. A noticeable trend was started in 2009 to use multi-core processors in hard real-time embedded systems [22].

The invention of multiprocessors on a single chip has improved the availability of high processing speed in a cost effective manner. The hardware for high processing speed has been developed by introducing multi-core systems but the theory necessary for the software architecture of such systems has not been fully developed [13]. In order to run software on multi-core systems, the most important issue is to schedule the task set on each of the cores in an effective and efficient manner. The scheduler for multi-core systems has to dispatch the task on each and every core in such a manner that it can extract the maximum level of parallelism out of the processor architecture, without affecting the correct functionality of the software and the system. Therefore, appropriate multiprocessor scheduling algorithms are necessary to utilize the expensive processing power.

1.2 Previous Work

In the previous work [41], a simulation software has been developed to simulate and visualize task sets for real-time embedded systems. The input parameters for the simulator are file based and the output is visual. The simulator simulates and visualizes simultaneously in such a manner that it can be thought of as real time simulation and visualization system. [41].

The simulator provides the required functionality so that it can be paused and resumed at any time. The simulator also analyzes the internal events and notifies the visualization thread accordingly, e.g. a deadline miss of a hard real-time task, a task changing its priority, etc. The visualization runs in a separate thread and updates the visualization context according to the triggered event from the simulation thread. The user can configure the event notifications of the simulator in order to customize the type of notifications they wants to see [41].

The software supports well known single core scheduling algorithms with various resource access protocols. From the category of multi-core scheduling algorithms *partitioned deadline monotonic scheduler* (P-DMS) and *global earliest deadline first* (G-EDF) scheduling have been implemented.

1.3 Contributions

In the beginning of this chapter, the contribution of this thesis work has been briefly described. In this section, the contribution is discussed in greater detail.

The target of this thesis is to study some promising multi-core real-time scheduling approaches and extend the existing simulator to support several multi-core scheduling algorithms considering the use of shared resources and the impact of resource blocking.

The new version of the existing simulator has to be improved by extending it with the following features:

- The existing software does not support schedulability analysis for single-core scheduling policies. The software should support the schedulability analysis for single core scheduling algorithms as well as partitioned scheduling algorithms where single core scheduling algorithms are used to schedule task sets.
- A task may self-suspend, so this feature should be implemented in the simulation software.
- The following multi-core scheduling policies must be supported for multi-core scheduling:
    - Scheduling Algorithms
        - Partitioned Earliest Deadline First (P-EDF)
        - Proportionate Fairness (Pfair)
    - Resource Access Protocol
        - Flexible Multi-processor Locking Protocol
- Interrupt handling should be supported by multi-core scheduling algorithms
- Clusters and grouped of tasks should also be supported.

As a part of this thesis work the existing visualization of the software should be enhanced as necessary for the specified features. Graphical library provided by Eclipse is used for the existing software and the same library should be used for the improvement of the graphical part and the visualization. As the existing software is written in Java in order to support multiple platforms [41], Java is selected as a programming language to develop the new features.

1.4 Organization

The rest of this thesis is organized as follows. Chapter 2 introduces and defines the terms related to real-time embedded systems in order to establish a common notation. Chapter 3 presents the project management and software engineering methods, applied for this thesis. In chapter 4, the design of the existing software is presented along with the changes done as part of this thesis work. Chapter 5 discusses the necessary theories related to the schedulability analysis of single core scheduling policies and chapter 6 deals with the general concepts of self-suspension, non-preemptive execution and interrupt handling. In chapter 7 and chapter 8, the general scheduling techniques for multiprocessor systems and some selected multiprocessor scheduling algorithms are discussed. Chapter 9 deals with the implementation details of the new features. The validation and testing of the newly implemented features is discussed in chapter 10 along with the results. Chapter 11 concludes with a discussion of the further work.

2 | Definitions

There are some common terms related to real-time embedded systems, which will be used throughout this thesis. This chapter defines those terms so that they are understood before they are used.

2.1 An Embedded System

An *embedded system* is an information processing or computer system enclosed in a fixed context which is not a general-purpose workstation like desktop or laptop computer, dedicated to predefined functionality, not directly visible to the user [41]. According to Barr [8],

> "an embedded system is a combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function."

The noticeable term is *dedicated function*, that contrasts to the general purpose computer which can be used to program and execute any kind of application. An embedded system ranges from portable devices like digital watches or GPS devices to large complex systems like modern cars, trains, aircraft, MRI etc.

2.2 A Real-Time System

Nowadays many embedded systems must meet real-time constraints. They have to finish a task execution within a predefined fixed time limit, which is defined as deadline of the task. Such a system is referred to as *real-time system.* According to Burns and Wellings [17], a real time system is defined as

> "any information processing activity or system which has to respond to externally generated input stimuli within a finite and specified delay"

The correctness of a real-time system not only depends on the correctness of the functionality, provided by the system, it also depends on the time at which the correct results are available. Real-time systems are divided into three categories:

Hard Real-Time Systems: In a *hard real-time system*, a task is not allowed to miss its deadline. The violation of hard real-time constraint can cause massive disaster such as loss of life or property [42]. Typical examples of hard real-time systems are industrial process control systems, medical implants such as pacemakers, robots, controllers for automotive systems, air-traffic controllers, etc.

Soft Real-Time Systems: In a *soft real-time system*, a task is allowed to miss its deadline occasionally but meet it on average, which is the opposite of a hard real-time system. As soft real-time constraints are less critical, they can be violated. The violation of soft real-time constraints does not directly impact system safety and the system can keep running. However, this violation of deadlines is not desirable because it may degrade the quality of service of the system [42].

Firm Real-Time Systems: A firm *real-time system* can be placed somewhere between hard and soft real-time system. It is neither soft nor hard real-time system. A task can miss its deadline until an upper bound has been reached. It is supposed to keep a certain level of quality of service to the end user [42].

2.3 Classification of Multiprocessor Systems

Multiprocessor systems can be classified into three different categories from the scheduling perspective [22].

Homogeneous Multiprocessor Systems: In this category of multiprocessor systems, the processors are identical. The rate of execution for all tasks on each processor is the same.

Heterogeneous Multiprocessor Systems: In this category of multiprocessor systems, the processors are not identical. The rate of execution of tasks depends on both the processor and the task. Furthermore, all tasks might not be able to execute on all processors.

Uniform Multiprocessor Systems: In this category of multiprocessor systems the task execution rate only depends on the speed of the processor. A processor of speed 2 will execute a task exactly at the double speed of what was executed on a processor of speed 1.

2.4 Tasks

In a real-time system, there are several processes with timing constraints; each of these processes is referred to as a task. The functionality of a real-time system is provided by the execution of these processes or tasks. A task can be invoked any number of times either finite or infinite. Each new invocation of a task is known as a job [31].

In other words, “a task is a software entity or program intended to process some specific input or to respond in a specific manner to events conveyed to it” [44], quoted by Munk [41]

A real-time task has a sequence of commands which includes execution that can be preemptive or non-preemptive, feature resource use and suspension delay. A real-time system may contain arbitrary n tasks T_i and the task set denoted as τ is a set of all tasks in the system where $\tau = \{T_1, T_2, \cdots T_n\}$. Each task has arbitrarily many attributes, the main three attributes are *worst-case execution time*, C_i, *period* or *minimum inter-arrival time*, P_i and *deadline*, D_i.

The release time of a task denoted as r_i is the time when the first job of a task is released. A periodic task set can be classified in two different categories based on the arrival time of the tasks. A task set is termed *synchronous* if there are some points where all tasks arrive simultaneously [22]. A task set is termed asynchronous if task arrival times are separated by fixed offsets and there is no simultaneous arrival time of all tasks [22]. Another way of defining it is: “a periodic task set is said to be synchronous if all the tasks release their first jobs at the same time otherwise asynchronous”[25].

2.4.1 Periodicity

A period is an important parameter of a task by which the task releases its jobs. Real-time tasks are categorized in three different classes based on the frequency of releasing jobs by a task.

Periodic: A periodic task arrives repeatedly with a certain regular interval, this interval is known as task period. A periodic task must be executed periodically. This means, a periodic task T_i with a period P_i, releases a new job at every P_i time units. A periodic task T_i therefore generates an infinite sequence of jobs at every P_i time units.

Sporadic: A sporadic task arrives irregularly with each arrival separated by at least a predefined time unit, this predefined time unit is called minimum inter-arrival time. A sporadic task T_i with *minimum inter-arrival time* P_i, generates two successive jobs by a delay of at least P_i time units. A sporadic task T_i therefore also generates an infinite sequence of jobs at least every P_i time units.

Aperiodic: An aperiodic task does not have any period or minimal inter-arrival time constraint. It can release a job at any time without caring about the delays between two successive job releases. This is the most difficult type of tasks that needs to be scheduled in a real-time system.

2.4.2 Deadlines

A deadline is another important parameter of a real-time tasks. The deadline D_i of a task T_i specifies the time when the job of this task must finish its execution. Therefore, D_i is the deadline of a task that defines the real-time behavior of that task. There are two types of deadlines to mention.

Relative Deadlines: A real-time job, one invocation of a task, needs to finish its execution by a predefined time interval after its invocation, the duration of this time interval is called *relative deadline* and denoted D_i.

Absolute Deadlines: The *absolute deadline* is the time by which the job must complete its execution. The absolute deadline is denoted d_i and defined as $d_i = r_i + D_i$ where r_i and D_i are the release time and relative deadline respectively.

There are three different types of tasks depending on the relation between deadline and period or minimum inter-arrival time.

Implicit Deadlines: A task has an implicit deadline if the deadline of the task is equal to its period or minimum inter-arrival time. A task T_i with a period P_i and deadline D_i is an implicit deadline task if $D_i = P_i$.

Constrained Deadlines: A task has a constrained deadline if the deadline of the task is equal to or less than its period or minimum inter-arrival time. A task T_i with a period P_i and deadline D_i is a constrained deadline task if $D_i \leq P_i$.

Arbitrary Deadlines: There is no constraint on a deadline for arbitrary deadline tasks, deadlines can be less than, equal to or greater than the period of the task. A task T_i with a period P_i and deadline D_i is an arbitrary deadline task if $D_i \leq P_i$ or $D_i > P_i$.

2.4.3 State Transitions

A task may be in any of the six states, *non-existing*, *created*, *ready*, *running*, *blocked* or *terminated*. The transition from one state to another state has been discussed in [41, section 2.3] using the state transition diagram. In this section, the state transition diagram is given, the solid arrow represents a valid state transition, and dashed arrow represents a state transition in case of any exception.

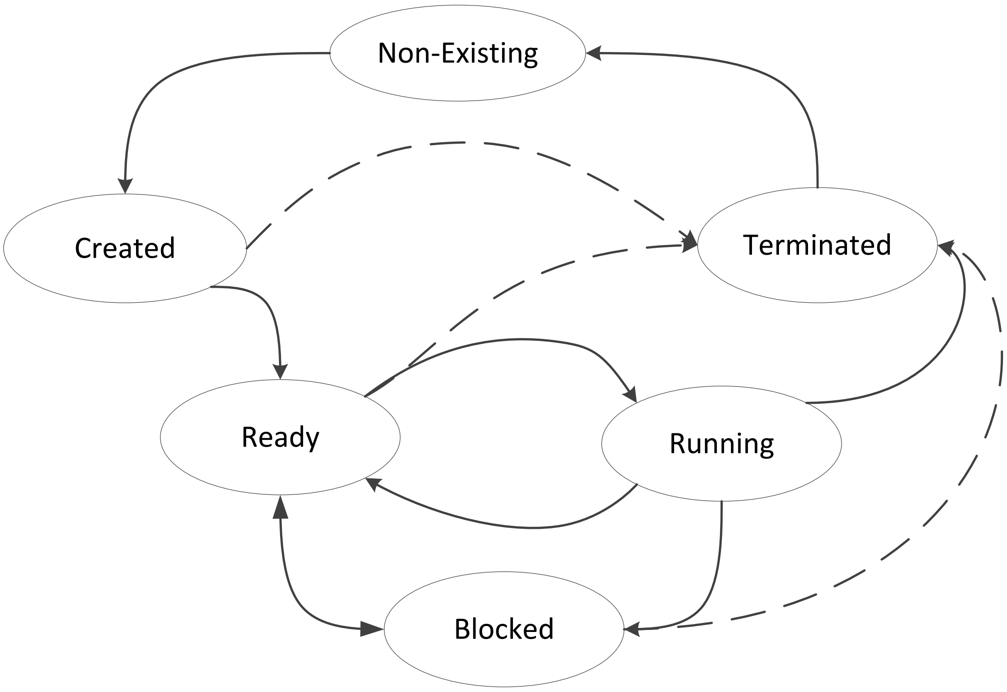


Figure 2.1: State transitions of a real-time task

2.5 Schedulability Tests

A schedulability test or analysis checks whether all the real-time tasks in a real-time system can meet their deadlines or not. There are three different schedulability tests:

Sufficient: A schedulability test is termed *sufficient* if all the tasks in the task set are considered to be schedulable by the test are in fact schedulable [23]. If the test is passed then the task set is schedulable. If the test is failed, then the task set may be schedulable or not, but not necessarily.

Necessary: A schedulability test is referred to as *necessary* if all tasks in the task set are considered to be unschedulable by the test are in fact not schedulable [23]. If the test is failed, then the task set will not be schedulable. If the test is passed then the task set may be schedulable but not necessarily.

Exact: A schedulability test which is both necessary and sufficient is referred to as an *exact* schedulability test [23]. The task set is schedulable if it passes the exact test; task set is unschedulable if the test is failed.

3 Project Management and Software Engineering

In this thesis work, along with theoretical work a previous version of simulation software had to be understood and was to be improved by developing some predefined new features. Therefore, as a part of this master thesis a piece software was going to be developed. In order to have quality software in time, the development of the software was done using project management with some software engineering steps.

Software development gives structure to the development of software products. Among the recent software development methodologies, agile methodologies are considered to be more fast and highly adaptable to the changes. The important aspect of agile methodologies contrasting them to the classical software development methodologies is that they do not focus on the long development cycle but rather do the work in short iterative light weight cycles. Agile methodologies are developed to be used in small software projects where the number of team members is limited [53]. As the development team comprises only one developer, a light weight software development methodology is appropriate, thus the agile software development methodology has been selected for the development of this project.

3.1 Project Management

Agile project management methods provide less time consuming procedures which can be used to manage agile software development. In order to manage the software development a well known agile project management method called *Scrum* is selected for the project. Scrum is a framework for managing an iterative and incremental development of a software project. There are three core roles in Scrum: *Product Owner*, *Development Team* and *Scrum Master*. The product owner represents the stakeholder, in this project the supervisor gets this responsibility. The development team contains only one member thus the other two roles go to that member. There are three phases of Scrum: *planning and design*, *sprint cycle* and *project closure* [53].

The product owner is responsible to create a feature list and prioritize the features, this list is known as *product backlog*. A specification is given by the product owner, i.e. supervisor, which contains all tasks that have to be done. A number of meetings have been scheduled with the supervisor to discuss the requirements and to plan and design accordingly. From these meetings the product backlog was created for the next phase known as sprint cycles.

Scrum provides an innovative feature which is the middle phase, the sprint cycle. The sprint is a short time period where a list of tasks will be taken to be completed by the specified time period. For this project the sprint duration was set to four weeks, a set of tasks was taken from the product backlog in order to complete a specific functionality and deliver it to the customer, i.e. the supervisor. If the developed product meets the requirements then it is marked as a completed task, otherwise the modification goes to the next sprint.

The final phase of Scrum is known as project closure where the project finishes by wrapping all the completed tasks into a single deliverable software unit that is delivered to the stakeholder, i.e. the supervisor.

3.2 Software Engineering

Incremental software development consists of development of the initial software, its delivery to the customers for their observation and comments, and its improvement by going through several versions until a complete fully functional system has been developed. Incremental software development is a fundamental part of agile methodologies which is considered to be better than the waterfall model [53]. In this approach specification, development, validation and verification are combined into one activity rather than separated.

The specification document of this thesis only listed some topics related to real-time scheduling problems which had to be understood properly by studying the available literature and implemented into the existing simulator. In the beginning, the implementation algorithms were not well defined and came from the literature study. In order to cope with these changing requirements incremental approach was selected as a software development methodology.

3.2.1 System Prototyping

System prototyping is the process of developing an initial version of the software, generally a part of the software. Rapid software development is used to develop a system prototype in order to check the requirements specified by the customer and the feasibility decisions of the software design. Then the software prototype is presented to the customer in order to show the functionality they desire and to collect their feedback. The customer satisfaction or changes of the prototype are therefore necessary to avoid frequent changes when the system gets bigger. Thus, the change requirements by the customer after the system delivery are reduced by noticeable amount[53].

The customer may get new ideas from the demonstration of the prototype and should be able to understand the effectiveness and efficiency of the software. Therefore, they can propose to substitute some old requirements with valuable new requirements.

The software prototypes are developed by using several Scrum sprints, and the results are presented to the customer in order to collaborate and gather customer feedback.

3.2.2 Incremental Delivery

Incremental delivery is the process where predefined tasks have been completed in one increment and delivered to the customer. This allows to have collaboration with the customer, they can test the software to see whether it meets their needs, if not they can place a change request for incremental delivery which can be incorporated into the system in the next increment at a relatively low cost[53].

In this project work, the Scrum sprints are considered as an increment and at the end of each sprint the output is delivered to the customer, i.e. the supervisor. The tasks are selected in each sprint in such a way that independent functionality is developed in every sprint that can be presented to the customer. The good thing about incremental delivery is that this presentable functionality is available after a certain time duration, i.e. sprint duration that satisfies the customer.

3.2.3 Continuous Integration

As a part of software development using the agile approach, during the development continuous integration has to be carried out. The continuous integration involves building the whole software frequently even though a negligible code segment has been changed. Software testing and documentation generation are included in this build process [53].

4 | Design

In the previous work, Munk [41, chapter-6] has presented a system model for the simulation software. This thesis is based on the previous work so the previous system design is being used with some extensions and modifications. In this chapter, the previous software design will be presented in brief along with the modifications in detail.

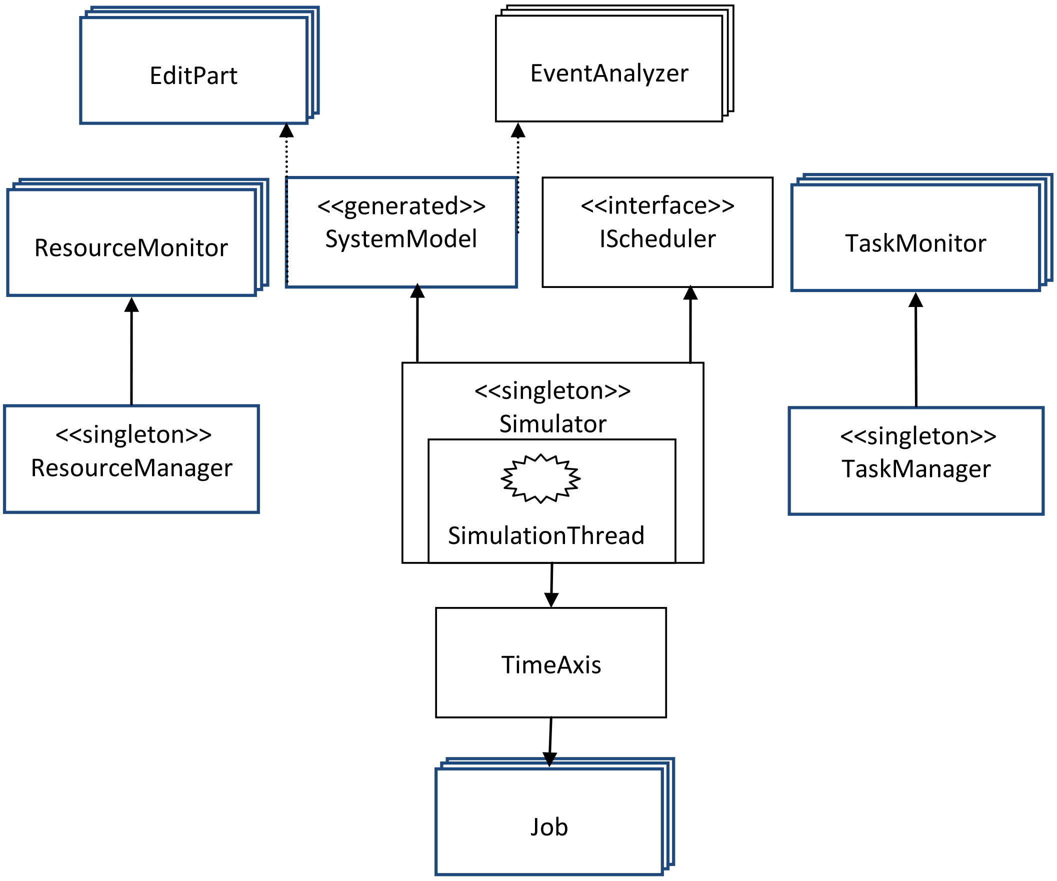


Figure 4.1: An overview of the system, software design [41]

Figure 4.1 shows an overview of the simulation software where classes are represented by boxes, references are represented by solid arrows and notifications are represented by dotted arrows [41]. The basic software design is the same as the previous design; no main block is added to the system design. There are some enhancements in the existing software blocks in order to support new features. The software components that are blue colored in the design diagram have been changed by adding new features to support schedulability analysis, task self suspension, non-preemptive execution and finally for multi-core scheduling simulation. As mentioned in [41], this diagram is a coarse overview so the UML standard has not been followed. In the following sections, the modified parts of the architecture will be discussed to clarify the improvements that have been done to the existing system.

4.1 System Model

Munk [41] quoted from Shannon [52], a simulation is "the process of designing a model of a real system and conducting experiments with this model for the purpose of understanding the behavior of the system and / or evaluating various strategies for the operation of the system." Thus, a simulation model is a necessary part of the simulation. The system model has to contain all necessary information in order to generate a schedule. The system needs to be designed in such a way that it is neither oversimplified nor too detailed whereas the quality of simulation result depends on the abstraction result of the system model [52].

In the previous work [41], a *meta-model* has been developed that tells how to describe the system model and its elements. This meta-model is named *system model.* The root element of the system model is the class `systemModel` that has a name and a description of String type. A system model consists of multiple `Tasks`, `Cores` and `Resources` [41].

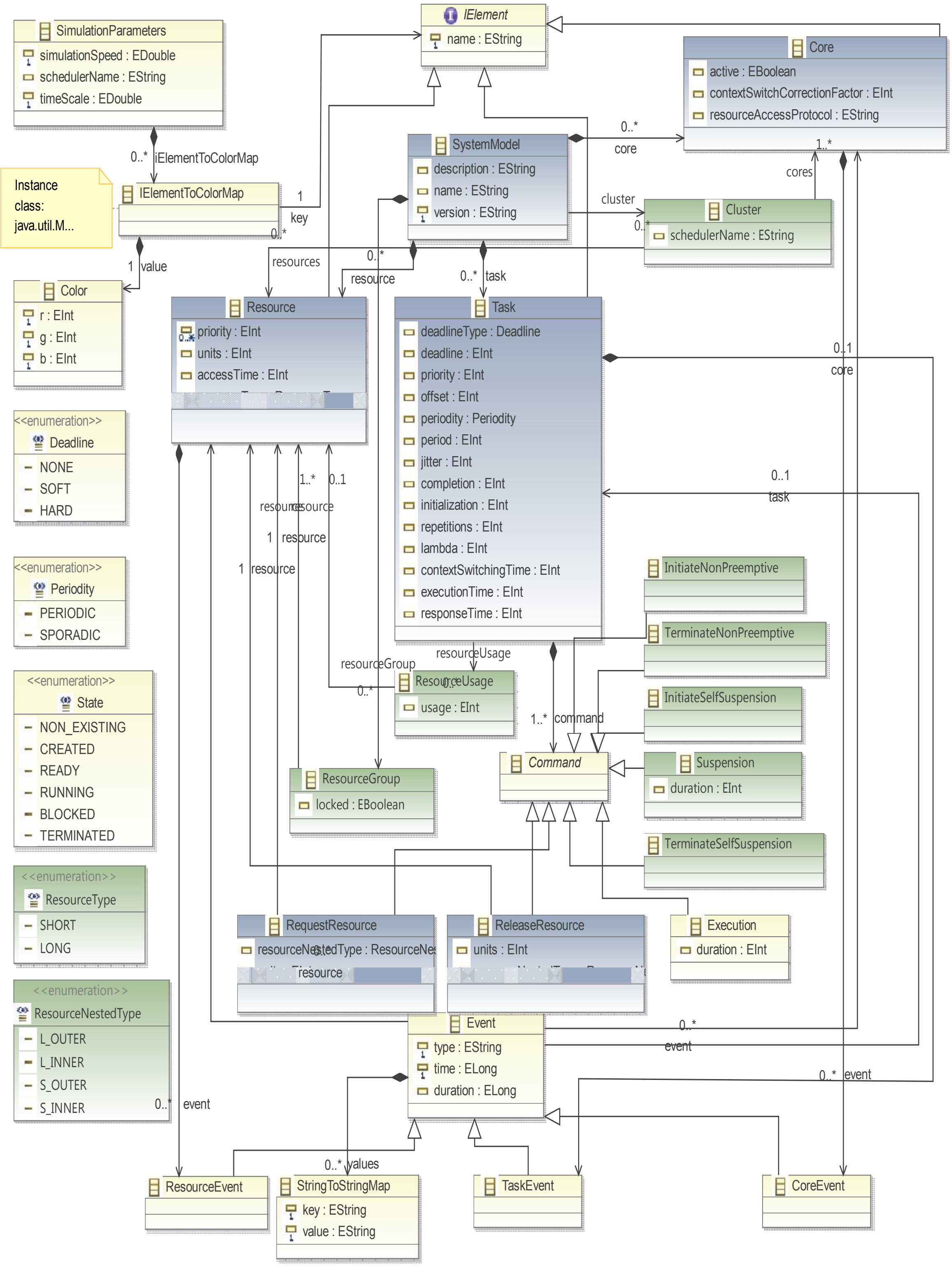


Figure 4.2: An overview of the system model

Figure-4.2 shows the UML representation of the system model. The system model contains all the model classes from the previous work that were added or modified as a part of this thesis work. The model classes colored yellow in the diagram are from the previous work without any modification. The model classes colored green, are newly added, and those colored blue are from the previous work with some necessary modification. The system model is designed and defined in the EMF framework, therefore the source code is generated by its framework [41].

The `systemModel` class is modified by adding two new attributes to it. A system model may also contain multiple instances of `ResourceGroup` and `Cluster`. The attribute `ResourceGroup` is necessary for *flexible multiprocessor locking protocol*, which is a resource access protocol for multiprocessor scheduling and the `Cluster` attribute is used to specify the clusters to group the task sets into m clusters where each cluster will be scheduled under a specified scheduling algorithm.

4.1.1 Tasks

A task should have some properties like periodicity, deadline type, deadline, execution time etc. All these basic properties have been included in the `Task` model of the previous work.

The execution time of a job, i.e one invocation of a task is not specified in an attribute. The total execution of a task is split into multiple commands [41]. Therefore, in order to get the total execution time of a task, every time one needs to go through a loop to sum up the execution time of all `Execution` commands. The total execution time of a task is necessary to perform the schedulability analysis. For this purpose an attribute named `executionTime` has been added to the `Task` model. This attribute stores the total execution time of the task.

In order to perform schedulability analysis, a response time analysis is done which calculates the response time of each task. Once the response time of a task is calculated, it is stored for future use, e.g. to check that the response time calculation conforms to the simulation output. For this purpose an attribute named `responseTime` has been added to the `Task` model. The calculated response time of a task will be stored in this attribute.

Response time analysis using blocking time requires to calculate the blocking time, which is different for various resource access protocols. In order to calculate blocking time for *simple priority inheritance* or *priority ceiling protocol*, the resource usage information is necessary. A task should keep the information of all resources whether it uses them or not. For this purpose an attribute named `resourceUsage` has been added that may contain multiple instances of `ResourceUsage` model class. Table 4.1 lists all the attributes of `Task` model with a brief description of each attribute, for more information see [41, sec 6.1.1]. A horizontal line is placed to separate the existing attributes and newly added attributes, the notation which will also be used for other tables where the attributes of a model class are listed.

Properties	Default Value	Description
deadlineType	NONE	The deadline of a task can be NONE, SOFT or HARD
deadline	0	The relative deadline of the task
priority	0	The priority of the task
offset	0	The release offset of the first creation of the task
periodity	PERIORIC	A task can be PERIODIC or SPORADIC
period	0	The time of the task's period or minimum inter-arrival time for sporadic task
jitter	0	The maximum variation of a task release time, earlier or later
completion	0	Duration of the completion phase, for future use
initialization	0	Duration of the initialization phase for future use
repetitions	-1	The number, how many times a task will repeat its creation; -1 represents infinite repetitions
lambda	0	The lambda parameter of the Poisson distribution, used for sporadic tasks
contexSwitching-Time	0	The time necessary for context switching before the task starts executing
executionTime	0	The execution time, i.e WCET of the task, it has to execute for each invocation
responseTime	0	The time necessary for a job, to complete its execution
resourceUsage	0	The list of resources a task uses

Table 4.1: The attributes of the task model with a default value and short description

4.1.1.1 Commands

A task may contain arbitrary sequences of Commands. A command is an abstract class inherited by its subclasses RequestResource, ReleaseResource, and Execution. The execution command has an attribute named duration that the task needs to execute which is known as WCET. The execution time of a task needs to be split for only execution and execution while using a resource. The previous work only supports these three commands. As a part of FMLP implementation, nested resource access should be understood by the simulator. A new enumeration model class has been created for this purpose named ResourceNestedType which contains L_OUTER, L_INNER, S_OUTER, S_INNER. A RequestResource or ReleaseResource should be marked as of the nested type. For this purpose a new attribute is added to these models named resourceNestedType which contains any value from the enumeration model accordingly.

The previous work does not support self suspension and non-preemptive execution. In order to enable the system to support these two features, some new commands have been introduced.

Self Suspension: three new commands have been introduced to support self suspension, `InitiateSelfSuspension`, `TerminateSelfSuspension`, and `Suspension`. All these three commands inherit the `Command` abstract class. These three commands work as execution with resource lock. When a task wants to suspend itself then these three commands come in the following sequence: `InitiateSelfSuspension`, `Suspension`, `TerminateSelfSuspension`. The suspension command has an attribute name `suspension`, that stores the time duration for which the task needs to be in suspension mode.

Non-Preemptive Execution: two new commands have been introduced to support a task executing its non-preemptive section, so that a higher priority task cannot preempt the task while it is executing its non-preemptive section. `InitiateNonPreemptive` and `TerminateNonPreemptive` are the two new commands inheriting the `Command` abstract class. When a task starts its non-preemptive execution it sets a `InitiateNonPreemptive` in the command sequence and a `TerminateNonPreemptive` command when it finishes non-preemptive execution. There will be an `Execution` command placed between these two commands which specifies the time duration of non-preemptive execution. This feature is necessary to implement FMLP, a resource access protocol for multiprocessor scheduling.

4.1.1.2 Resource Usage

A `ResourceUsage` model class has two attributes, one is `resource` of type `Resource` class and another is `usage` of type integer. The value of this usage attribute can be either 0 or 1. A task has the `resourceUsage` of all resources that are necessary to calculate blocking time and thus response time.

4.1.2 Cores

A `Core` is a part of the processor. A processor i.e. a processing chip may have multiple cores which is known as multi-core system. A system model prepared for single core scheduling will have one core and for multi-core scheduling a system model will have more than one core, thus a system model may contain arbitrary number of cores.

In the existing system model the `Core` model has two attributes `active` and `contextSwitchCorrectionFactor`. The short description will be given in the attributes overview table, for more information see [41, sec-6.1.2]

In partitioned scheduling approach of multi-core scheduling, a different resource access protocol can be specified for each core. i.e one core can run using one resource access protocol, whereas another core may use different resource access protocol. In order to support this feature, an attribute named `resourceAccessProtocol` is added to the `Core` model. The use of this attribute will be discussed in 9.4 on page 80.

The attributes of the `Core` model are listed in table-4.2 along with short description of each attribute and their default values.

Properties	Default Value	Description
`active`	`false`	Option to deactivate a core, for future use
`contextSwitch-CorrectionFactor`	0	A correction factor of task context switch timings
`resourceAccess-Protocol`	`NULL`	The resource access protocol that will be used for this core with the specified scheduling algorithm for partitioned multi-core scheduling algorithm

Table 4.2: The attributes of the core model with default value and short description

4.1.3 Resources

A unit of a resource can only be exclusively accessed by one task at a time as defined in [41, sec-2.5].

In the existing system model the `Resource` model has two attributes `priority` and `units`. Some resource access protocols need to specify a ceiling priority, for this reason a resource has this `priority` attribute. There may be some resource with multiple units, the resource model has the `units` attribute to specify how many units of that resource there are. More about these two attributes is found in [41, sec-6.1.1]

In order to perform the response time analysis, blocking time is necessary for calculation. The `accessTime` attribute of a resource is needed to calculate the blocking time. The `accessTime` of a resource is the maximum access time among all tasks using this resource. For FMLP, it is necessary to mark a resource as short or long. The `resourceType` attribute tells whether a resource is short or long.

The attributes of the `Resource` model are listed in table-4.3 along with short description of each attribute and their default values.

Properties	Default Value	Description
`priority`	`empty list`	A list of priority ceiling values in the order of available units
`units`	1	The number of total available units
`accessTime`	0	The maximum access time of all task used this resource
`resourceType`	LONG	The resource type either SHORT or LONG

Table 4.3: The attributes of the resource model with default value and short description

4.1.4 Events

There are arbitrarily many `Events` generated during the simulation related to tasks, resources and cores. The examples of such events are: a task is created, a task is scheduled on a specific core, the priority of a task changes during the simulation, a task is blocked to access a resource etc. The visualization of the simulation is event based, when an event occurs in the simulation, it triggers the simulation event and the visualization updates accordingly, thus the events work as an interface between the simulation and visualization [41]. Every model class contains an attribute of event class to allow a different visualization kind depending on the event type [41].

A new event type can be created without modifying the model as the `type` attribute of `Event` class is a string. Existing simulator triggers a `CORE_USE_EVENT` when a task is scheduled on a core in order to update the visualization. In the implementation of GSN-EDF, it requires to update the visualization when a task is linked to a core in addition with the task scheduled on a core, for this reason a new event is created as `TASK_LINKED_EVENT` in order to update the visualization when a task is linked to a processor. The system introduces two different running types `RUNNING` and `NON_PREEMPTIVE_RUNNING` for execution to update the visualization correctly for preemptive and non-preemptive execution accordingly.

4.2 Simulation Parameters

The simulation model is loaded from an `XML` file along with some other information for the simulator to run. The class `SimulationParameters` is modeled with the Eclipse EMF which is not a part of the system model. This class has some important attributes: those will be taken from the user, the scheduler name that will do the task scheduling as `schedulerName`, simulation speed as `simulationSpeed` and the time scale for the visualization as `timeScale` [41]. As a part of this thesis work nothing has been changed in this class.

4.3 Simulator

The simulator is the central element of the simulation software and the simulation kernel is necessary to access from other classes like scheduler, visualization etc. In order to support this, the class `Simulator` is designed to develop according to the singleton design pattern. Therefore, the number of instances of this class is restricted to one which allows to access this class from other places without creating an instance [41].

The simulation and visualization need to run separately in order to keep them independent. The simulator class contains an inner class `SimulationThread` by extending the Java `Thread` class which is responsible for the simulation. The simulator class contains the methods `start()`, `play()`, `pause()`, and `stop()` to control the `SimulationThread` and thus, simulation [41].

The class `TimeAxis` holds a map between integral points of time *t* and a list of events at that time. The list of events represented by the abstract class `Job`. Thus, the events are generally generated by the `Jobs`. In general, the `Jobs` are the implementation of the `Command` model and some other events like deadline check, to block a task etc. Each implementation of a `Job` has to implement `executeJob()`, where the code segment is placed to do the necessary work for this `Job` and `isDispatchNecessary()` to tell the simulator whether a dispatch is necessary or not. Therefore, the unnecessary calls to dispatch method are avoided [41].

A job can be added to the time axis by calling its `addJob(final long time, final IJob job)` function. The simulation thread calls the function `executeJobs(final long now)` to execute all the jobs at that point and returns the `boolean` value stating whether a dispatch is necessary or not. If a dispatch is necessary then the scheduler gets a dispatch call and thus executes the `dispatch()` method of the respective scheduler to schedule the task set at that point of time. Thereafter, the simulator gets the next time where there are some jobs to process by the function `getNextTimeStep()` of `TimeAxis` class.

A scheduler interface `IScheduler` is presented in [41, figure 6.4]. Any combination of scheduling policy and resource access protocol must implement this interface. The `initialize()` method is to initialize the variables. In order to request and release a resource `requestResource()` and `releaseResource()` methods will be used respectively. The scheduling decision is done inside the `dispatch()` method. Whenever a scheduling decision is necessary, the dispatch method of the scheduler is called. The `stateChangeRequest()` method is used to receive the notification when the state of a task is changed.

4.4 Task Monitor

The responsibility of dispatch() method of a scheduler is not only taking the scheduling decision but also starting the task execution, preempting a task if necessary etc. In order to provide this functionality, necessary methods are provided by the TaskMonitor class. A task is linked to a task monitor where necessary attributes are defined to maintain the life cycle of a task via the TaskManager class. The TaskManager is a singleton class that maintains a map between the tasks and their monitors [41].

Task Monitor provides functions to change the state of a task, create the jobs to the time axis, etc. The important thing is done here, it creates the events accordingly which is necessary for the visualization thread to update the view part[41].

The newly implemented features, e.g. self-suspension, non-preemptive execution, FMLP requires to add some attributes and functions in the TaskMonitor class. The details of these attributes and functions will be discussed in detail in their respective implementation sections.

4.5 Resource Monitor

Like TaskMonitor, a resource is linked to a ResourceMonitor via the ResourceManager class, which is also a singleton class that keeps track of resources with its associated resource monitor. ResourceMonitor class provides necessary methods to have functionality of locking a resource when necessary, and releasing a resource when its use is complete. The necessary events are generated from this class required for the visualization of resource events[41].

In order to implement FMLP, resources need to be grouped, one group may contain one or more resources. In FMLP, a lock needs to be granted on a resource group rather than a lock on a single resource. As a resource monitor only keeps the information for one resource, the locking is not possible inside this class. ResourceManager keeps the mapping between a resource and a resource monitor for all resources, thus the information for all resources can be obtained in this class by adding new attributes and the necessary functions are placed here to lock and unlock a group required by the FMLP. This will be discussed in detail in the section on FMLP implementation.

4.6 Exception Handling

The advantages of using Java and Eclipse RCP include the availability of built-in feature for exception handling. A view which is called “Error Log”, lists all the status messages. The class StatusManager can be used to add new status messages in the above list. It supports different severity levels, i.e., information, warning and error. The exceptions that are caught in the main simulation loop called `SimulationThread`, that is simulation exceptions, that arise during simulation are reported by the `StatusManager` class [41].

The class `SimulationException` is introduced in order to differentiate between the Java exceptions and exceptions that are provided by the simulation. The simulation exceptions may be an illegal state transition, resource release for more than available units etc. The `SimulationException` inherits from the `java.lang.Exception` to serving as Java exception. The main simulation exceptions come from the `TaskMonitor` or `ResourceMonitor` classes as they serve the purpose of changing the state of a task and requesting or releasing a resource [41]. Simulation exceptions may be thrown from some other places like newly developed Pfair schedulers when a task violates its lag constraint.

As a part of this thesis work, all new implementations use the existing exception handling introduced by the previous work by Munk [41].

5 | Schedulability Analysis

Many safety-critical hard real-time systems involve several distinct functionalities where each functionality has a hard real-time constraint that must not be violated [17]. It is quite challenging to develop such a system where no critical task misses its deadline. In preemptive scheduling algorithms the order of task execution changes dynamically so it is very difficult to predict in advance whether all tasks will meet their deadline or not.

In multitasking real-time systems, schedulability analysis is formally employed to prove in advance that a task set that will be scheduled on a system will conform to its deadlines. In this chapter, the necessary theory for schedulability analysis of single-processor algorithms implemented by Munk [41] will be discussed. The implementation will be discussed in chapter 9.

A schedulability analysis can be either direct or indirect. An indirect schedulability test does not calculate the delay between arrival time and completion time known as response time, but determine the schedulability of a given task set for the given system parameters. Direct schedulability test will explicitly calculate the worst-case response time of the tasks to determine the schedulability. Direct schedulability analysis is more accurate than indirect schedulability analysis but results in high computing costs [58].

Schedulability Analysis for Fixed Priority Preemptive Scheduling: As discussed above, there are two different approaches for schedulability analysis [28]. The earlier schedulability analysis which calculates the total processor utilization i.e. the percentage of processor used by all the tasks is known as *utilization based schedulability test*. The later work on schedulability analysis focuses on *response time analysis*, i.e. the time duration by which a task completes its execution [35, 5]. These two approaches will be used for *fixed priority preemptive scheduling* schedulability analysis.

Schedulability Analysis for Dynamic Priority Preemptive Scheduling: The most common *dynamic priority scheduling* algorithm, *earliest deadline first* which was introduced by Liu and Layland [38]. Dertouzos [24] proves that *earliest deadline first* is optimal among all other uni-processor scheduling algorithms. The schedulability analysis for implicit task set scheduling under EDF scheduler can be done using utilization based schedulability analysis mentioned in [38]. Demand bound analysis will be done for constrained deadline task sets.

5.1 Utilization Based Schedulability Analysis

The utilization based schedulability test is the most common and less costly approach in terms of computing cost [58]. In this analysis a task set can only be scheduled if the total utilization of all tasks is lower than a predefined bound [58]. This utilization bound differs for different algorithms used to schedule the task set [38].

5.1.1 Rate Monotonic Scheduling

Rate monotonic scheduler is a *fixed priority preemptive* scheduling algorithm. A static priority is assigned to each task according to the rule *"The shorter the period, the higher the priority".* [38]

In order to perform the utilization based schedulability test for RMS the processor load needs to be measured i.e. the processor utilization [38]

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i} \tag{5.1}$$

A necessary but not sufficient condition for RMS schedulability test is that the total processor utilization is not more than one, i.e. $U \leq 1$. A simple utilization based sufficient schedulability test for RMS with the following conditions:

- all tasks are independent, there is no communication or synchronization between the tasks, if it does exist this test does not consider that.
- every task in the task set should have implicit deadline i.e. deadline is equal to period $D_i = P_i$.
- the task set should be synchronous.

A real-time system is schedulable under RMS if the total utilization of the task set does not exceed the utilization bound according to Liu and Layland [38].

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{1/n} - 1) \tag{5.2}$$

This is a sufficient test but not necessary [4].

5.1.2 Deadline Monotonic Scheduling

For a task set, deadline equaling period is not always realistic, sometimes a task set may have tasks with deadlines being less than or equal to period. Deadline monotonic scheduler is also a *fixed priority preemptive* scheduling algorithm. The priority assignment differs from RMS in such a way *"The shorter the relative deadline, the higher the priority"*, called deadline monotonic priority assignment.

Audsley et al. [4] observe that a task may experience worst case interference I_i from the higher priority tasks in the same task set while scheduling using DMS.

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{D_i}{P_j} \right\rceil C_j \tag{5.3}$$

The entire task set will be scheduled using DMS, if [4]

$$\text{for all task } i, \frac{C_i}{D_i} + \frac{I_i}{D_i} \leq 1 \tag{5.4}$$

where C_i/D_i is the processor utilization by task i, until its deadline D_i and I_i/D_i is the maximum degree of interference (utilization) it experiences from higher priority tasks until deadline D_i. This test checks that the processor utilization of any task T_i plus the degree of interference it experienced from higher priority tasks before the deadline of the task T_i is not more than 1. This is also a sufficient test but not necessary [4].

5.1.3 Earliest Deadline First

A task set is schedulable under *earliest deadline first* scheduling algorithm if the total processor utilization by all the tasks in this task set does not exceed 100% [38]. Thus, a set of periodic tasks is schedulable under EDF if the following conditions hold

- all tasks in the task set are independent.
- deadline is equal to period i.e. $D_i = P_i$ for all tasks.
- the task set is synchronous.
- and utilization:

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1 \tag{5.5}$$

which is an exact schedulability test for EDF without considering the synchronization and communication between tasks.

5.2 Response Time Analysis

This analysis finds out the worst-case response time of each task and compares it to its deadline to determine whether a task can meet its deadline or not. The response time analysis can test the schedulability of a task set with arbitrary deadline tasks [56]. This section describes the schedulability analysis for an arbitrary deadline task set.

5.2.1 Exact Schedulability Test

In this section, the response time analysis considers each task to be independent, only accounts for the execution time, does not care about the blocking time while accessing shared resources. Therefore, this test is both necessary and sufficient schedulability test. The response time analysis for i^{th} process is given by the following equation

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{P_j} \right\rceil \times \ C_j \tag{5.6}$$

where $\sum\limits_{j \in hp(i)} \lceil R_i/P_j \rceil \times C_j$ is the worst-case interference by the higher priority tasks. In equation 5.6, the worst-case response time R_i comes in both left and right hand sides. Note that, the right hand side of equation 5.6 is a monotonically non-decreasing function of R_i [23]. This equation is a fixed point iteration where the iteration starts with an initial value of R_i which is $R_i^0 = C_i$. From this equation the smallest R_i needs to be found satisfying the equation which can be solved by a recurrence relation

$$W_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{W_i}{P_j} \right\rceil \times \ C_j \tag{5.7}$$

starts from the initial value and continues until W_i stabilize, meaning $W_i^n = W_i^{n+1}$ or $W_i > D_i$ where D_i is the relative deadline of i^{th} task [23].

5.2.2 Sufficient Schedulability Test

In the previous section on the exact schedulability test by response time analysis, it is assumed that all tasks are independent, no synchronization or communication between tasks is present. Synchronization and communication add another factor that may increase the response time for some tasks i.e. a task may be blocked to access shared resources. Therefore, equation-5.6 can be rewritten as follows by adding blocking time B_i

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{P_j} \right\rceil \times \ C_j \tag{5.8}$$

where B_i is the blocking time i.e. a task is blocked on accessing some shared resources that need to be calculated. The worst case blocking time B_i needs to be calculated which is different for different resource access protocols. In order to calculate blocking time for *default* protocol and *simple priority inheritance* protocol for resource access, the worst-case blocking time can be calculated by the following equation

$$B_i = \sum_{k=1}^{K} usages(k, i) \times CS(k) \tag{5.9}$$

The worst-case blocking time calculation is different for *Priority Ceiling Protocol* which can be calculated by the following equation

$$B_i = \max(usages(k, i) \times CS(k)) \tag{5.10}$$

In the above two equations (equation 5.9 and 5.10), K is the number of resources and $CS(k)$ is the worst-case access time of resource k. The usage of a resource k is defined as, $usage(k, i) = 1$ if resource k is used by a higher priority task including the task itself and a lower priority task, otherwise $usage(k, i) = 0$

Like equation 5.6, equation 5.8 is also a fixed point iteration where the iteration starts with a different initial value which is $R_i^0 = B_i + C_i$ and continue until it stabilizes.

A scheduler requires to switch between jobs while scheduling a set of tasks, which is known as *context switching* [30]. Context switching overhead refers to the overhead when there is preemption. The processor needs to save the context of the current job, load the context of the next job and resume it [30]. Thus, the cost for context switching may lengthen the response time. The following equation calculates response time by also considering context switching time.

$$R_i = CS1 + B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{P_j} \right\rceil \times \ (CS1 + CS2 + C_j) \tag{5.11}$$

where $CS1$ is the WCET cost of context switch to the process and $CS2$ is the WCET cost of context switch away from the process.

5.3 Demand Bound Analysis

A utilization based necessary and sufficient schedulability analysis is presented by Liu and Layland [38] where all the tasks have implicit deadlines, i.e the relative deadlines are equal to their period, which has been presented in section 5.1.3. In real-time systems, the assumption that all task deadlines will be equal to their period is not always realistic. For such a system where the deadline of a task is not implicit, the schedulability analysis is not possible by the utilization based schedulability analysis.

The earlier research on exact schedulability analysis for the EDF scheduling algorithm with arbitrary deadlines was described by Leung and Merrill [36]. They showed that a periodic task set will be schedulable using the EDF algorithm if and only if the absolute deadlines of all the tasks in the period $[\,0, \max\{s_i + 2H\}\,]$ has been meet. Where s_i is the start time of a task T_i and $\min\{s_i\} = 0$, H is the least common multiple of the periods of the tasks. Baruah et al. [9] improved this schedulability analysis by introducing a processor demand function. According to them, a task set, including sporadic tasks, is schedulable if and only if $\forall\ t > 0,\ h(t) \leq t$, here t is the time and $h(t)$ is processor demand function defined as:

$$h(t) = \sum_{i=0}^{n} \max\{0, 1 + \left\lfloor \frac{t - D_i}{P_i} \right\rfloor\}\ C_i \tag{5.12}$$

In the above demand function, the time value t that is not bound and is more than 0 (zero) can be bound to a certain value. Now the lower bound and upper bound will be discussed first and then they will be combined.

Upper Bound: A task set is schedulable If and only if $U \leq 1$ and $\forall t\ L_a, h(t) \leq t$. Where the upper bound L_a is defined as: [10, 9, 49]

$$L_a = \max_{1<i<n}\{P_i - D_i\}\frac{U}{1 - U} \tag{5.13}$$

Lower Bound: A task set is schedulable if and only if $U \leq 1$ and $\forall t\ L_b, h(t) \leq t$ where L_b is the synchronous busy period of the task set. The interval between 0 (zero) and the first time the process is idle is called busy period. The length of the synchronous busy period L_b can be computed by the following equation to stop at the first idle time

$$W(0) = \sum_{i=1}^{n} C_i \tag{5.14}$$

$$W(k) = \sum_{i=1}^{n} \left\lceil \frac{W(k-1)}{P_i} \right\rceil \tag{5.15}$$

This iteration stops when the first idle time is found which is at $W(k)$ when $W(k-1) = W(k)$.

Combining Upper Bound and Lower Bound: Now the upper bound and lower bound can combined to a single equation to check schedulability of a task set. Thus, a task set is schedulable if and only if $U \leq 1$ and $\forall t \in P, h(t) \leq t$. Where the P is defined as a collection of absolute deadlines by the following equation

$$P = \{d_k |\ d_k = kP_i + D_i \wedge d_k < \min(L_a, L_b), k \in N\} \tag{5.16}$$

The schedulability analysis using demand bound analysis for EDF is necessary and sufficient thus it is an exact schedulability test without considering synchronization between tasks.

5.4 Summary

The following table lists all discussed schedulability analysis techniques with respective single core scheduling algorithms for which they can be used on different types of task sets except the sufficient response time analysis that is not listed in the table, the table only lists appropriate schedulability tests for various single core scheduling algorithms, applied only to independent task sets.

		Deadline Equals Period	**Deadline Less Than Period**
RMS	Sufficient	$U = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{1/n} - 1)$	
	Exact	Response Time Analysis	
DMS	Sufficient	$\frac{C_i}{D_i} + \frac{I_i}{D_i} \leq 1$	
	Exact	Response Time Analysis	
EDF	Sufficient		
	Exact	$U = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$	Demand Bound Analysis

Table 5.1: Schedulability analysis summary

6 Self Suspension, Non-Preemptive Execution and Interrupt Handling

6.1 Self Suspension

In many real-time embedded systems, tasks may invoke some significant suspension intervals during communication, synchronization and on accessing external resources like I/O. The intervals of these suspensions are mostly independent of the processors on which these tasks are being scheduled. Traditional real-time systems consider the self suspension time as a part of execution time [34], and thus, the task holds the processor though it does not need it for computation. In this section, the basic theory on self-suspension of real time tasks will be discussed for implementation into the simulator.

In many real-time systems when a task interacts with external devices or waits for some events to occur it may introduce self suspension delays. The examples of such devices are magnetic disc, network card, etc. When a task initiates to use a device and sends the commands to process, there may be some predefined time intervals that the device will use to process the data to get the result. After sending the command to the device, the task may self-suspend itself by the predefined time duration, which is termed self-suspension time.

The recent research on scheduling tasks with self-suspension delays [47, 48] shows that the feasibility problem of scheduling periodic tasks containing self-suspension delays is *NP-hard* in the strong sense. In real-time systems such self-suspension delays impact schedulability analysis quite negatively when deadline misses are not allowed [39].

Thus, the sequence of execution times of a task can be split by adding suspension delays between the execution times. Thus, the execution time C_i of each task T_i can be represented as a sequence of execution times and suspension delays [34]

$$C_i = (C_i^1, E_i^1, C_i^2, E_i^2, C_i^3, E_i^3, \cdots, E_i^{m_i-1}, C_i^{m_i}) \tag{6.1}$$

where there are m_i non-suspending computation time chunks separated by $m_i -$

1 suspension delays. There are some constraints on using the defined model for self suspension described in [34]. The most important constraint is that a task will contain multiple execution chunks with different suspension behavior which is supported by the existing simulator. As the self-suspension behavior is defined by the system model, the self suspension is defined for the first time when the model is being prepared; the self-suspension cannot happen at unknown time. This problem has also been solved in the course of development of the simulation that is necessary for FMLP implementation.

When a task self suspends itself, the scheduler removes it from the ready queue, places it into the blocked queue, and takes the next eligible task for scheduling.

The schedulability analysis for a single core scheduling algorithm, which is a part of this thesis work, will not work if a task in a task set is going to be scheduled on the system that contains self-suspension delays. Thus, the schedulability analysis including self-suspension delays can be treated as future work.

6.2 Non-Preemptive Execution

The term *non-preemptive execution* refers to the situation when in a preemptive scheduling algorithm a task in the task set may have a finite time duration that should be executed non-preemptively even though there is a higher priority ready task that is waiting for its execution. FMLP requires a version of G-EDF where jobs can be non-preemptable for a short duration of time [14]. In order to implement FMLP, the existing simulator needs to be updated to support a scenario where a task may have non-preemptive section which should be executed non-preemptively. As the execution of a non-preemptive section in a preemptive scheduling algorithm is supported by the simulator for the implementation of GSN-EDF which is required for the FMLP implementation, it will also be available for any preemptive scheduling algorithm supported by the existing system and its improvement is part of this thesis work.

A higher priority jobs can be blocked by a lower priority job when it is executing its non-preemptive section and such blocking is termed *non-preemptive blocking.*

6.3 Interrupt Handling

Interrupt is an asynchronous notification to the processor and it may occur at any time, possibly between two instructions. If an interrupt is detected by a processor, it pauses the currently scheduled task and instead runs a designated *interrupt service routine* (ISR). Therefore the interrupted task faces undesirable delay and the response time is increased. Most interrupts are maskable. However, there are some routines to check the system health that are non-maskable, known as *non maskable interrupts* (NMIs). In multiprocessor interrupt handling, some interrupts may be designated to a processor on which they will be processed. In other cases an interrupt can be processed by any processor [16].

Interrupts are categorized into four different groups, *device interrupts* (DIs), *timer interrupts* (TIs), *cycle-stealing interrupts* (CSIs), and *inter-processor interrupts* (IPIs) [16].

In the uni-processor scheduling for real-time systems interrupt handling methods were discussed by Liu and Layland [38]. In static priority scheduling algorithms the priorities of interrupts are set to higher priority values and thus processed instantly. In EDF, a dynamic priority scheduling algorithm, “schedulability can be tested by treating time lost to processing interrupts as a source of blocking” [32].

In the existing simulation software, the interrupt requests are handled in the same way Liu and Layland [38] described it in their paper. For fixed priority scheduling algorithms, the task which is treated as an interrupt gets the highest priority. In the Earliest Deadline First scheduling algorithm, the deadline of the interrupt task is given a small value which is very close to its WCET and thus it is executed as a higher priority task.

In the multiprocessor scheduling algorithms the interrupts are handled in the same way as they are handled in the single-core scheduling algorithms. There is a problem of handling interrupts for the Pfair scheduling algorithms where the priority is dynamic and the deadline of a task must be equal to its period. Therefore in Pfair, it is a bit difficult to make the priority of the interrupt higher as it is done for other scheduling algorithms. The interrupt task will also progress in proportion to its weight.

7 Scheduling on Multiprocessor Systems

Nowadays real-time embedded systems are increasingly used in many different environments where the technology is changing rapidly. As a result the number of processors on a single chip is growing every year for more powerful and rapid computation.

In recent years, chip designers prefer to increase the number of processors on a single chip rather than increasing the processor speed because of cost minimization and some other related factors [33, 50, 59]. Thus, task execution parallelism is necessary to use the maximum processing speed of multiprocessor systems.

This technological modification of the processor architecture has stimulated more research in the field of multiprocessor scheduling in recent years. There are many mature algorithms for single core scheduling that can feasibly schedule a task set on a single processor system [17, 18, 40] . On the other hand, real-time multiprocessor scheduling is a more difficult problem than uni-processor scheduling [37]. Real-time multiprocessor scheduling research is still ongoing with many open problems.

The correctness of a *real-time* system not only depends on the correctness of the functionality provided by the system, it also depends on the time at which the correct results are available. Thus, a *real-time* system should function properly within a predefined time. For example, in a safety critical system, the system must be shut down as soon as any indicator is reacting in an unusual way. If the system is not shut down within a very short time period, it may cause life risk.

7.1 Taxonomy of Multiprocessor Scheduling Algorithms

Multiprocessor scheduling can be regarded as trying to solve two problems: the *allocation problem* that will define on which processor a task should execute and the *priority problem* that will define when and in what order each task should execute with respect to other tasks. Multiprocessor scheduling algorithms can be classified into three different categories according to the available degree of inter-processor migration and three other categories according to when the priority is being changed [19].

7.1.1 Allocation

Traditionally inter-processor migration has been restricted in real-time system. In many systems, the cost of context switching from one processor to another processor is prohibited. The traditional real-time scheduling theory does not have techniques, tools and results so that it could analyze the system in detail to allow migration.

Recent development in multiprocessors on a single chip and very fast communication over small areas solves the issue of context switching cost described earlier. So the system designer does not need to care about inter-processor migration solely due to implementation consideration. The result of recent experiments also shows that scheduling algorithms that provide inter-processor migration are competitive in comparison to scheduling algorithms that do not allow migration [54].

Scheduling algorithms fall into one of the following three different categories according to the degree of migration [22].

7.1.1.1 No Migration

Every task is assigned to a processor on which it can be scheduled. In this category, migration among processes is not allowed and the technique can also be named *partitioned*. There are m different subsets of task sets if the available numbers of processors are m. A subset of tasks is assigned to a processor and needs to schedule only on that processor.

7.1.1.2 Task Level Migration

The jobs of a task may execute on different processors; however, each job must execute entirely on a single processor. So, the run-time context of a job needs to be maintained only on one processor, whereas the task-level context migration is possible. Task level migration is also known as *restricted migration*.

7.1.1.3 Job Level Migration

There is no restriction on inter-processor migration. A job of a task can be migrated and moved from one processor to another; however, parallel execution of a job is restricted so that a job cannot be scheduled on more than one processor at a time. Task level migration is also referred to as *full migration*.

7.1.2 Priority

A scheduling algorithm falls into any of the following three different categories according to the complexity of the priority scheme [22].

7.1.2.1 Fixed Task Priority

A unique fixed priority is assigned to each task and this priority will be applied to all of its jobs. Fixed job priority is also referred to as *static priority.*

7.1.2.2 Fixed Job Priority

The different jobs of a task may have different priorities but each job will have the same static priority.

7.1.2.3 Dynamic Priority

As the name suggests, a job may have different priority at different points of time. There is no restriction on priorities that can be assigned to jobs.

7.1.3 Work-conserving and non-work-conserving

A scheduling algorithm is said to be *work-conserving*, if the processor is not idle at any time where one or more tasks are in *ready* states and waiting to get the processor to be executed, otherwise it is said to be *non-work-conserving.* The multiprocessor scheduling algorithms in the class of partitioned approach are not work-conserving because they may allow an idle processor while there are some ready tasks on other processors and waiting to get the processor but the idle processor cannot schedule them [22].

7.2 Schedulability, Feasibility and Optimality

A task set τ is said to be feasible with respect to a system if there exists a scheduler that can schedule all possible sequences of jobs released by the tasks belonging to τ on that system so that no task misses its deadline [22]. Therefore, the feasibility of a task set τ is not dependent on a particular scheduling algorithm used to schedule the jobs released by the tasks in τ [42].

A task set τ is said to be schedulable with respect to a scheduling algorithm S, if any job generated by the tasks in τ does not miss its deadline in the schedule generated by S [42]. In other words, a task set τ is said to be schedulable with respect to a scheduling algorithm if the worst-case response time of all the jobs generated by the tasks in τ is less than or equal to their respective deadlines [22].

The optimality of a real-time scheduling algorithm can be derived by combining these two terms, schedulability and feasibility [42]. Thus, a real-time scheduling algorithm S is said to be optimal with respect to a system and a task model,if it can schedule any task set that is generated according to the task model and is feasible on the system [22]. Therefore, an optimal scheduling algorithm is an algorithm that can schedule a task set for which a scheduling solution exists [42].

7.3 Multiprocessor Scheduling Approaches

There are two main approaches that are traditionally considered for scheduling on multiprocessors, they are *partitioned* and *global* scheduling. Scheduling algorithms where task migration is restricted are known as *partitioned scheduling* and where task migration is not restricted are known as *global scheduling.* In multiprocessor scheduling, sometimes these two approaches are combined which is called *hybrid scheduling* [6, 7, 19, 25].

7.3.1 Partitioned Scheduling

In this approach, tasks are partitioned among available processors. It means, a task is assigned to a specific processor and is always scheduled only on that processor. Therefore, the multiprocessor system becomes a collection of independent uni-processor systems and a uni-processor scheduling algorithm will run for each core to schedule the tasks assigned to that processor. Unlike in global scheduling, in partitioned approach tasks cannot migrate from one processor to another, as this is prohibited for partitioned approach. For example, in multiprocessor scheduling with partitioned EDF (P-EDF), a uni-processor scheduling algorithm, *earliest deadline first* is used to schedule the task set on each processor independently [22].

The following advantages have been observed for partitioned scheduling algorithms over global scheduling algorithms for scheduling on multiprocessors [22]

- A task may overrun its worst-case execution time, this situation only affects the tasks on the same processor.
- There is no penalty in terms of migration cost as each task only runs on the same processor.
- A priority queue is maintained for each processor, so the queue length is small and queue access is faster in comparison to global priority queue.
- The main advantages of using this approach for multiprocessor scheduling is that it reduces the multiprocessor scheduling to single processor scheduling, when the assignment of all tasks to processors has been done.

The following disadvantages have been observed for partitioned scheduling algorithms over global scheduling

- The main disadvantage of this approach is allocation of the tasks to the processors. Finding the optimal assignment of tasks to processors is a bin-packing problem which is NP-Hard in the strong sense.
- Partitioned approach is not work-conserving, i.e. a processor may remain idle while there may be some ready task to be scheduled on another processor and missing its deadline.

The performance of the partitioned scheduling algorithms depends on the bin packing algorithms used to partition the tasks among the processors of the multiprocessor system. Truly, the bin packing algorithms cannot ensure a task partitioning that can achieve total utilization of more than $(m + 1)/2$ on a multiprocessor system where the number of processors is m [3]. Therefore, in the worst-case scenario, slightly more than 50% of the processing capacity of the multiprocessor system is used by the partitioned scheduling algorithms for the task execution while the other almost 50% remain unused [42].

7.3.2 Global Scheduling

In this approach, all the ready tasks are placed in a global priority queue and the scheduler selects the highest priority task from the queue for its execution. A task can be dispatched to any available processor even after preemption. Unlike partitioned scheduling, a task is not fixed onto a processor and task (job) migration is permitted in global scheduling. For example, the global version of EDF named global EDF (G-EDF) selects the m highest priority (earliest deadline) ready tasks to execute on the m processors of the multiprocessor system at time t [22].

The following advantages have been observed for global scheduling algorithms over partitioned scheduling algorithms for scheduling on multiprocessors [22]

- There are typically fewer preemptions because the scheduler needs to preempt a task when there is no idle processor but a task needs to be scheduled.
- When a task used less time than its worst-case behavior then the time can be used by all tasks, not a subset of tasks as in partitioned scheduling.
- This approach is more appropriate for open systems where load balancing / task allocation is not necessary for taskset changes.
- Some global scheduling algorithms are work-conserving, i.e. there will be no idle processor if there is a ready task needed to be scheduled.

The following disadvantages have been observed for global scheduling algorithms over partitioned scheduling algorithms for scheduling on multiprocessors

- Global scheduling uses a single queue for ready tasks. The queue length is long and queue access is time consuming.

- A job can be frequently migrated from one processor to another processor, this leads to a high migration overhead of the system.

In 2001, Andersson et al. [3] develop an utilization bounds for periodic task-sets with implicit deadlines. According to them, the maximum utilization bound for any global fixed job priority algorithm is $(m + 1)/2$. Thus, the G-EDF and its variants have the same utilization bound as partitioned EDF [42].

7.3.3 Hybrid Scheduling

In global scheduling, depending on the hardware architecture, a task may migrate from one processor to another frequently which leads to a very high migration overhead. This job migration results in excessive communication loads and cache misses which increase the worst case execution time that would not occur in the fully partitioned or non-migration scheduling approaches. On the other hand, in partitioned scheduling, the total available processing capacity is fragmented and a large amount of processing capacity is unused and remains idle. In fully partitioned approach the maximum utilization bound is just about 50% of the total processing capacity thus the utilization is very low [22].

Furthermore, there are some systems that can be scheduled under fully partitioned or fully global scheduling approach because, some tasks are not permitted to migrate while some other are permitted to migrate [43].

The recently developed more general hybrid scheduling approach is cluster based scheduling which can be categorized as a generalization of partitioned and global scheduling protocols [43]. In this approach, tasks are statically assigned to clusters and then inside a cluster tasks are scheduled globally. There are two forms of cluster based scheduling; clusters can be physical or virtual. In physical cluster-based scheduling the processors are statically assigned to each cluster and each cluster will contain a subset of processors from the available multi-processor system. In virtual cluster-based scheduling the processors are dynamically mapped to each cluster, there is a one-to-many relationship between cluster and multiprocessors[43].

Thus, clustering is an approach where processors are partitioned into a smaller number of subsystems to which tasks are allocated. Therefore, the issue of capacity fragmentation in partitioned approaches is improved and the number of processors in each cluster are fewer than on the total system which reduces the global queue length and the migration overhead. The processors in the same cluster may share the same cache which reduces the penalty in terms of increasing worst-case execution time despite migrations.

7.4 Resource Sharing

In the previous section, the scheduling approaches for multiprocessors have been discussed in detail where the tasks are independent from each other. In this section resource access protocol will be discussed for both *partitioned* and *global* approaches in order to allow accesses to shared resources by the tasks.

In 1988, a multiprocessor variant of the *priority ceiling protocol* named *MPCP* has been introduced by Rajkumar et al. [46], which is applicable with the fixed priority scheduling algorithms of partitioned approaches. The global shared resources receive a priority ceiling values that are strictly higher than those of any other tasks in the system. When a task tries to acquire a global resource that is currently locked by any other task then the requested task is blocked and waits in a priority queue for the maintenance of blocked tasks on that resource. As a result, the lower priority local tasks can continue execution. At the time of global resource being released, a task will be taken from the head of the queue waiting to acquire this resource, will have the lock on the resource and resume with the ceiling priority of the resource. MPCP restricts nested access to the globally shared resources, i.e. local and global critical section nesting is not allowed. A bounded blocking time and a sufficient schedulability test is provided by MPCP, the schedulability test is based on utilization bound provided by Liu and Layland [38] [22].

Another multiprocessor variant of *priority ceiling protocol* named MDPCP has been described by Chen et al. [20]. A sufficient schedulability analysis for Partitioned EDF with MDPCP resource access protocol is provided by them for a multiprocessor system [20].

A multiprocessor resource access protocol MSRP has been introduced by Gai et al., which is based on the *stack resource protocol* for uni-processor resource access protocol. MSRP can be used with either fixed priority scheduling algorithms or EDF for partitioned systems. The significant difference between MPCP and MSRP is that when a resource is blocked on a global resource in MSRP, it busy-waits and becomes non-preemptive which is known as *spin-lock*. A FIFO queue is used for each global resource in order to store the blocked task on this resource so that the correct task resumes when the resource is released. In MSRP, the spin-lock wastes processing time just by making the processor idle which can be used by other tasks [22].

Devi et al. [26] proposed two simple methods for short non-nested access to shared data structures by considering the problems of accessing shared resources under global EDF, a scheduling approach for global multiprocessor scheduling. The methods are *spin-based queue lock* and *lock free synchronization* [22].

Block et al. [14] introduced a multiprocessor resource access protocol which is known as *flexible multiprocessor locking protocol* abbreviated as FMLP. FMLP can be used with both partitioned and global scheduling algorithms. A variant of both partitioned EDF and global EDF is necessary to use FMLP which is discussed in detail in section 8.3.2 on page 69.

8 | Multiprocessor Scheduling Policies

In chapter-7, the general concepts for scheduling on multiprocessor systems have been discussed. In this section, a few scheduling algorithms both from partitioned and global approach will be discussed in detail.

The existing simulation software developed by Munk [41] supports multi-core scheduling algorithms along with single-core scheduling algorithms. He explained how to add a new scheduler to the SAVORS system [41, sec. 8.11]. New scheduling policies and resource access protocols can be added to the existing installation of the software as an additional plug-in. All newly implemented multi-core scheduling algorithms were added to the `de.unistuttgart.iste.ps.savors.scheduler.multicore` plug-in. A multi-core scheduling algorithm will be available to the software after adding it to the plug-in extension.

This simulation software supports only homogeneous multiprocessor systems, defined in Chapter-2 [41]. The simulation software does not support task parallelism, meaning a task cannot run on more than one processors at a time [41]. Therefore, the newly implemented multi-core algorithms only work for homogeneous multiprocessor systems without task parallelism.

8.1 Partitioned Earliest Deadline First

In the Partitioned EDF multiprocessor scheduling algorithm the task-set needs to be divided into smaller disjoint sets of tasks. There are several heuristic methods for this partitioning process. Once the task-set is partitioned, on each processor each task set will be scheduled by a uni-processor scheduling algorithm *Earliest Deadline First.*

Therefore, the multiprocessor scheduling problem under partitioned approach can be divided into solving two different parts: partition the task set into m subsets of tasks, so that each subset is schedulable where m is number of processors, and run a uni-processor scheduler for each core. In this case the uni-processor scheduling algorithm is EDF. Several polynomial time heuristics are present to partition the task set on multiprocessor systems based on bin-packing approaches which are known to be NP-hard in the strong sense [21, 27]. Three of them are going to be described here in brief, details with the algorithms can be found in [55, section 2.1.2]

Heuristic Methods for Task Partitioning

Next Fit: this method maintains an ordering among the processors and assigns tasks to each processor. Since many tasks can be assigned on that processor in such a way, the partition is schedulable using a desirable single core scheduling algorithm. The detailed procedure can be found in [55, figure 2.3]

First Fit: the next fit method is achieved by considering that a task will be assigned to the first available processor to which it fits and is known as *first fit*, described in [55, figure 2.4].

Best Fit: in this method, each task is assigned to a processor that the assigned task can fit on and the remaining unused processing capacity is minimized. Srinivasan [55] described this method in section 2.4 with EDF as uni-processor scheduling algorithm.

The *next fit* partitioning method has a time complexity of $\mathcal{O}(n)$ times the uni-processor schedulability analysis, whereas the *first fit* and *best fit* partitioning methods have the time complexity of $\mathcal{O}(mn)$ times the uni-processor schedulability analysis time.

In P-EDF, when the task set partitioning has been done, a uni-processor dynamic priority scheduling algorithm EDF will be used to schedule the task sets assigned to that core.

8.2 Proportionate Fairness

Baruah et al. introduced the *Proportionate Fair* (Pfair) scheduling algorithm in their paper in 1996[12]. The algorithms in this class of *Pfair* scheduling are applicable to schedule periodic task sets with implicit deadlines. *Pfair* scheduling algorithms are based on the concept of *fluid scheduling*, where each task executes or make progress proportionate to its utilization or task weight. A task weight is the ratio of the execution time and the period and termed W_i i.e. $W_i = C_i/P_i$.

Liu, first discussed the periodic scheduling problems in 1969[37]. A task set of n tasks and m resources, where each task has an execution time C_i and a period P_i by which the task is periodic. A scheduler that schedules the task set on m processors, allocates exactly C_i time units to each task in every P_i time units [37].

A decision problem that a task set can be scheduled on the given number of processors can be solved efficiently. A task set cannot be scheduled if the total weight of all tasks is more than the available number of processors i.e. $\sum_{i=0}^{n-1} W_i > m$. A task set will be schedulable if $\sum_{i=0}^{n-1} W_i \leq m$ and resource sharing is allowed [10].

Baruah et al. mentioned that they have solved the periodic scheduling problems by appointing an even stonger fairness constraint in [12]. They mention the approach as proportionate progress where each task is scheduled on resources in proportion to its weight. At any time t, a task T_i should have executed either $\lfloor W_i \cdot t \rfloor$ or $\lceil W_i \cdot t \rceil$. They named this approach *proportionate fairness* or *P-fairness*. The P-fairness is a strict and stronger condition than periodic scheduling because any proportionate scheduling is periodic where the inverse is not always true [12]. They proved that any periodic task set has a P-fair schedule if $\sum_{i=0}^{n-1} W_i \leq m$ holds. P-Fair scheduling is optimal for periodic task sets with implicit deadlines [12].

P-Fairness

- In *P-fair* scheduling algorithms, scheduling decisions need to be taken at every point of time t, starting from 0. The time interval between t and t+1 (including t but excluding $t + 1$) is called slot t, $t \in N$.
- The symbol ϕ is referred to as fair scheduling problems with m processors and n tasks.
- Every task T_i has an integer period P_i, an integer execution time C_i and a rational weight W_i, where $T_i, P_i > 1$ and $0 < W_i < 1$.
- Although the loss of generality Baruah et al. [12] limits their attention to the case
$$\sum_{i=0}^{n-1} W_i = m \tag{8.1}$$
In order to maintain this constraint they suggest to use a dummy task for the unused processor time with larger execution time and period [12].

Some Definitions:

- A schedule S for an instant ϕ is a function from $\tau \times N$ to $\{0, 1\}$ where $\sum_{T_i \in \tau} S(T_i, t) \leq m, t \in N$. Informally $S(T_i, t) = 1$ if and only if the task T_i is scheduled at slot t by the scheduler S, otherwise $S(T_i, t) = 0$ [12].
- Lag is the most important term in P-fair scheduling which calculates the difference between how much resource a task should receive in the interval $[0, t)$ and how much it actually received. The lag of a task T_i at time t with respect to a scheduler S is denoted $Lag(S, T_i, t)$ and is defined as
$$lag(S, T_i, t) = W_i \cdot t - \sum_{i \in [0,t)} S(T_i, i) \tag{8.2}$$
- A schedule S is said to be P-fair if and only if at any time t, the absolute value of its lag constraint is strictly less than 1 i.e.
$$\forall\ T_i, t : T_i \in \tau, t \in N : -1 < lag(S, T_i, t) < 1 \tag{8.3}$$

Existence of P-fair Scheduling: The correctness proof of *P-fair* scheduling algorithm depends on the existence of P-fair schedule of the resource (processor) sharing problem. Regarding resource sharing scheduling instance ϕ, the earliest and latest possible slot denoted $earliest(T_i, j)$ and $latest(T_i, j)$ respectively, during which the j^{th} time, $j \in N$, of task T_i may be scheduled. The closed expression of $earliest(T_i, j)$ and $latest(T_i, j)$ are $earliest(T_i, j) = min\ t : t \in N : W_i \cdot (t + 1) - (j + 1) > -1$ and $latest(T_i, j) = max\ t : t \in N : W_i \cdot t - j < 1$ [12].

$$earliest(T_i, j) = \lfloor j/W_i \rfloor \tag{8.4}$$

$$latest(T_i, j) = \lceil (j + 1)/W_i \rceil - 1 \tag{8.5}$$

Observe that, $earliest(T_i, j) < latest(T_i, j)$ where $T_i \in \tau, j \in N$. Let the least common multiple of all tasks in the task set be L i.e $L = lcm_{T_i \in \tau}\ P_i$. Baruah et al. [12, Lemma 3.1] proved a lemma in their paper that a scheduling instance ϕ has a P-fair schedule if and only if there is a schedule S i.e.

$$\forall\ T_i, t : T_i \in \tau, t \in (0, L] : -1 < lag(S, T_i, t) < 1 \tag{8.6}$$

8.2.1 Algorithm PF

In this section, the original P-fair scheduling algorithm PF will be discussed. As a P-fair scheduling algorithm, PF was the first optimal multiprocessor scheduling algorithm [12]. Algorithm PF assigns a dynamic priority for each task at every integral point of time $t \geq 0$ and then schedules m highest priority tasks to m available processors in slot t with an arbitrary tie breaking mechanism.

PF uses the earliest-pseudo-deadline-first mechanism to prioritize the sub-tasks and several tie breaking rules to break the tie when several sub-tasks have the same pseudo-deadline.

Some definitions:

- The characteristic string is an important parameter for P-fair scheduling. The characteristic string of a task T_i is an infinite string over the sequence of $\{-, 0, +\}$ which is denoted $\alpha(T_i)$ [12]. At every integral point of time t the characteristic string of a task can be calculated by the following equation

$$\alpha_t(T_i) = \sin(W_i \cdot (t + 1) - \lfloor W_i \cdot t \rfloor - 1),\ t \in N. \tag{8.7}$$

At time t, the characteristic sub-string of a task T_i is a finite string

$$\alpha_t(T_i, t) = \alpha_{t+1}(T_i)\alpha_{t+2}(T_i) \cdots \alpha_{t'}(T_i) \tag{8.8}$$

where $t^{'} = \min i : i > t : \alpha_i(T_i) = 0$. [12]. Informally, the characteristic string of a task T_i at time t is a finite sequence of string over $-, 0, +$ starting from the characteristic string of time t to until it finds a 0 in its characteristic string [12].

- Regarding P-fair scheduling a task will be in any of three different states *ahead*, *behind* or *punctual* based on its executed time. With respect to a schedule S, at time t, a task T_i is said to be *ahead* if and only if $lag(S, T_i, t) < 0$, *behind* if and only if $lag(S, T_i, t) > 0$ and *punctual* if and only if the task is neither ahead nor behind [12].
- In P-fair scheduling all tasks will be placed in three different categories examining their executed time and the characteristic string at time t. The three different categories are *urgent*, *tnegru* and *contending*. A task T_i will be placed in *urgent* category if and only if it is behind and the characteristic string $\alpha_t(T_i) \neq -$, the task T_i will be placed in *tnegru* category if and only if it is ahead and $\alpha_t(T_i) \neq +$ and the task T_i will be placed in *contending* category if and only if it is neither urgent nor tnegru [12].

With the above terms and definitions the original P-fair scheduling algorithm (PF) can be presented based on [12], derived from which it was implemented as a multi-core scheduler as part of this thesis work. This algorithm is referred to as *Algorithm PF* in [12].

At any time t, the Algorithm PF has to make a decision which m sub-tasks from the task set of n tasks need to be scheduled. According to Baruah et al. [12, Lemma 4.4], all urgent tasks must be scheduled in the current time slot, i.e. slot t in order to maintain P-fairness for the scheduler. According to Baruah et al. [12, Lemma 4.3], in order to preserve the P-fairness, no tnegru tasks will be scheduled in the current time slot t. As PF generates P-fair scheduling, at time t, it must schedule all the urgent tasks and no tnegru task.

There are n tasks in the task set and m resources to schedule those tasks. The first step is to do the task categorization, i.e. n tasks will be divided into urgent, tnegru and contending tasks. The assumption is that there are n_0, n_1 and n_2 urgent, tnegru and contending tasks respectively. As mentioned before, the scheduler needs to schedule all urgent tasks at time t. If the number of urgent tasks are more than the available resources, i.e. $n_0 > m$, it would be impossible for the algorithm PF to schedule all the urgent tasks and thus it will violate the lag constraint. The assumption is that $\sum_{T_i \in \tau} W_i = m$, see equation-8.1, therefore to get a schedule instance ϕ correctly the scheduler should allocate exactly m resources in every time slot. But if $n_1 > n - m$ then the scheduler is bound to schedule some tnegru task. Baruah et al. [12] showed that the two above discussed problems will never happen in PF scheduler, thus $n_1 \leq n - m$ and $n_0 \leq m$.

At each time t, the algorithm PF will schedule all the urgent tasks and if there are free resources after executing all urgent tasks then the scheduler executes $m - n_0$

tasks for the contending tasks according to the total order by which contenting tasks are prioritized. The algorithm PF is summarized below in [12].

Algorithm PF
1. All urgent tasks must be scheduled. 2. No tnegru task is scheduled. 3. The remaining resources must be allocated to the highest priority contending tasks according to total order.

Table 8.1: Algorithm PF

the PF algorithm uses characteristic strings to order the contending tasks moving forward from time t. At every time t, the total order $\succeq$ to prioritize tasks in the contending task set can be defined as: If T_i and T_j are two different contending tasks then $T_i \succeq T_j$ if and only if $\alpha(T_i,t) \geq \alpha(T_j,t)$. The lexicographical comparison is done between the characteristic sub-strings of $\alpha(T_i,t)$ and $\alpha(T_j,t)$ where the priorities of the string characters are defined as $- < 0 < +$ [12].

8.2.1.1 The Comparison Algorithm

The PF algorithm needs a comparison algorithm to compare two tasks to determine the total order. Baruah et al. [12] have presented two different implementations of the characteristic sub-string comparison algorithm. The first, they called `NaiveCompare` and proved its correctness. The second, they called `Compare` and proved its equivalence to the first algorithm.

8.2.1.1.1 A Naive Implementation This section discusses the naive implementation of the characteristic sub-string comparison algorithm which will be used by the PF algorithm to compare two contending tasks in order to determine the total order. Let's assume two contending tasks are T_i and T_j at any time t. The goal is to determine whether [12]:

(i) $\alpha(T_i,t) < \alpha(T_j,t)$

(ii) $\alpha(T_i,t) > \alpha(T_j,t)$

(iii) $\alpha(T_i,t) = \alpha(T_j,t)$

8.2.1.1.2 An Efficient Implementation In this section an efficient algorithm will be discussed to compare the characteristic sub-string named `Compare` that has been presented in [12, sec 6.2] by Baruah et al.. The comparison algorithm `Compare` is a polynomial time algorithm with the same input output behavior as `NaiveCompare`.

Algorithm 1 A naive implementation of the characteristic sub-string comparison algorithm

```
function NaiveCompare($a_0, b_0, c_0, a_1, b_1, c_1$)
    int $a_0, b_0, c_0, a_1, b_1, c_1$;
    do
        $c_0, c_1 := c_0 - a_0, c_1 - a_1$
        do
            $c_0, c_1 := c_0 + b_0, c_1 + b_1$
        while $(c_0 < 0 \wedge c_1 < 0)$
    while $(c_0 > 0 \wedge c_1 > 0)$
    if $c_0 = 0 \wedge c_1 = 0$ then return TIE
    end if
    if $c_0 \geq 0 \wedge c_1 \leq 0$ then return 0
    else if $c_0 \leq 0 \wedge c_1 \geq 0$ then return 1
    end if
end function
```

Algorithm 2 An efficient implementation of the characteristic sub-string comparison algorithm

```
function Compare($a_0, b_0, c_0, a_1, b_1, c_1$)
    int $a_0, b_0, c_0, a_1, b_1, c_1$;
    if $\min(a_0, a_1) > \min(b_0, b_1)$ then
        return Compare($b_1, a_1, -c_1, b_0, a_0, -c_0$)
    end if
    if $\lceil c_0/a_0 \rceil > \lceil c_1/a_1 \rceil$ then return 0
    else if $\lceil c_0/a_0 \rceil < \lceil c_1/a_1 \rceil$ then return 1
    end if
    $c_0, c_1 := c_0 - a_0 \cdot \lceil c_0/a_0 \rceil, c_1 - a_1 \cdot \lceil c_1/a_1 \rceil$;
    if $c_0 = 0 \wedge c_1 = 0$ then return TIE
    else if $c_0 \neq 0 \wedge c_1 = 0$ then return 0
    else if $c_0 = 0 \wedge c_1 \neq 0$ then return 1
    end if
    if $\lfloor b_0/a_0 \rfloor > \lfloor b_1/a_1 \rfloor$ then return 0
    else if $\lfloor b_0/a_0 \rfloor < \lfloor b_1/a_1 \rfloor$ then return 1
    end if
    return Compare($a_0 - (b_0 \mod a_0), b_0 \mod a_0, c_0 + (b_0 \mod a_0), a_1 - (b_1 \mod a_1), b_1 \mod a_1, c_1 + (b_1 \mod a_1)$)
end function
```

8.2.2 Algorithm PD

Baruah et al. [11] defined an algorithm to solve the periodic scheduling problem that has a running time $\mathcal{O}(\min\{m \log n, n\})$. They named the algorithm *Algorithm PD*, the letters "PD" refer to the word *pseudo-deadline* which is very close to the term *quasi-deadline* used for *Algorithm PF*. They proved in [11] that Algorithm PD is correct and also generates a P-fair schedule for each feasible periodic scheduling instance ϕ.

Algorithm PD is developed based on the correctness of Algorithm PF, by looking up the constant time tie-breaking rules which is sufficient to preserve the P-fairness constraint. The main aim is to limit the tie-breaking rules to check only for a fixed number of future pseudo-deadlines. Algorithm PF used a finer "resolution" of tie breaking rules. Thus, these two algorithms may produce different scheduling decisions but they are closely related [11].

Like Algorithm PF, Algorithm PD also used the task categorization into three different categories namely *urgent*, *tnegru* and *contending* at time t but it orders the contending tasks in a different way which will be discussed in the following sections starting with definitions.

Some definitions:

- The task sets are divided into two different groups, *heavy* and *light*. A task T_i, that has a weight W_i is heavy if the weight of the task is more than or equal to $1/2$ (i.e $W_i > 1/2$) and the task is light if the weight of the task is less than $1/2$ (i.e $W_i < 1/2$). A task of weight $1/2$ cannot go to both categories at the same time, it can go either to the heavy of light task group [11].

- Baruah et al. [11] defined another type of characteristic string $\beta(T_i)$ which is closely related to the characteristic string $\alpha(T_i)$ defined in algorithm PF. For a light task $\beta(T_i) = \alpha(T_i)$ and for a heavy task $\beta(T_i) = \alpha(T_i')$ where T_i' is assumed to be a task of weight $W_i' = 1 - W_i$. They introduced $\beta(T_i)$ to generate a schedule for T_i from a schedule for T_i'. They did it reversely by allocating a resource in exactly those slots where T_i' is not scheduled.

- A task T_i will have a pseudo-deadline at slot t if $\beta(T_i) = 0$ or $\beta(T_i) = +$. A pseudo-deadline corresponds to the quasi-deadline. A pseudo-deadline is the last slot by which a light task T_i must get the resource a given number of times and a heavy task T_i must deny the resource a given number of times to preserve the P-fairness. If a task has a pseudo-deadline at any time slot i where $i > t$ and there is no pseudo-deadline between i and t, it is denoted $\delta(T_i, t)$ [11].

- An integer value k is defined for each task T_i which is determined from the execution time of the task C_i and period of the task P_i. For a light task, the integer value k is defined as $k = \lfloor P_i/C_i \rfloor$ and for a heavy task the integer value k is defined as $k = \lfloor P_i/(P_i - C_i) \rfloor$.

Baruah et al. [11] defined a total order $\sqsupseteq$ on tuples $\{N, \{0, +\}, N\}$ by which the contending tasks will be prioritized and sorted according to the priority. The total order is defined as:

$$\begin{aligned}(d_1, s_1, k_1) \sqsupseteq (d_2, s_2, k_2) \iff & (d_1 < d_2) \vee \\ & ((d_1 = d_2) \wedge (s_1 = +) \vee (s_2 = 0)) \vee \\ & ((d_1 = d_2) \wedge (s_1 = s_2) \vee (k_1 < k_2))\end{aligned} \tag{8.9}$$

The total order $\sqsupseteq$ defined in equation 8.9 produces an ordering on task T_i and T_j at any time t [11]

$$T_i \trianglerighteq T_j \iff (\delta(T_i, t), \beta_{\delta(T_i,t)}(T_i), T_i.k) \sqsupseteq (\delta(T_j, t), \beta_{\delta(T_j,t)}(T_j), T_j.k) \tag{8.10}$$

The equation 8.10 will order the task set according to the total order $\trianglerighteq$. There is a tie between task T_i and T_j if and only if $T_i \trianglerighteq T_j$ and $T_j \trianglerighteq T_i$ hold. Such tie can be broken arbitrarily [11]. At every slot t, Algorithm PD will assign m resources to the m highest priority tasks. The tasks will be categorized in seven categories, where lower categories have the higher priorities. The tasks will be categorized by the following rules, listed in the table [11]

Task categorization according to the algorithm PD
1. All urgent tasks (all tasks must be scheduled).
2. Heavy contending tasks having characteristic string + at time $t+1$ i.e $\alpha_{t+1}(T_i) = +$. In this category, a task T_i receives higher priority than T_j if and only if $T_j \trianglerighteq T_i$.
3. Light contending tasks having characteristic string + at time $t+1$ i.e $\alpha_{t+1}(T_i)$. In this category, a task T_i has higher priority than T_j if and only if $T_i \trianglerighteq T_j$.
4. Heavy contending tasks having characteristic string 0 at time $t+1$ i.e $\alpha_{t+1}(T_i) = 0$.
5. Light contending tasks having characteristic string 0 at time $t+1$ i.e $\alpha_{t+1}(T_i) = 0$.
6. Rest of the heavy contending tasks will be placed in this category. In this category, a task T_i receives higher priority than T_j if and only if $T_j \trianglerighteq T_i$.
7. Rest of the light contending tasks will be placed in this category. In this category, a task T_i has higher priority than T_j if and only if $T_i \trianglerighteq T_j$.

Table 8.2: Task categorization used by the algorithm PD

8.3 Multiprocessor Resource Access Protocol

The recent movement from uni-processor systems to multi-processor systems stimulates the research of analysis techniques to support real-time applications on multiprocessor systems. In order to support real-time applications on multiprocessor systems, the multiprocessor scheduling algorithms and resource access protocols are necessary to maintain the timing constraint of all tasks scheduled on the system [15]. The scheduler should restrict in some way to guarantee such constraints. In lock based resource sharing, processor capacity or utilization is lost when a task is blocked to access a resource that is used by another task. To minimize such loss of processing time an efficient locking protocol is necessary.

In order to access a resource on uni-processor systems, some efficient resource access protocols exist that ensure that each job of a task blocks at most once [51, 6, 45]. Resource access on multiprocessor systems is much more complex than on uni-processor systems. In 2007, Block et al. [14] presented a resource access protocol for multiprocessor systems named *Flexible Multiprocessor Locking Protocol* abbreviated as FMLP. FMLP can be used for both the partitioned and global scheduling approaches.

In this section the *Flexible Multiprocessor Locking Protocol* will be discussed as implemented with Global EDF scheduler. In order to implement the resource access protocol with Global-EDF, it is not sufficient to implement FMLP with the original version of G-EDF. In the original version of the G-EDF scheduling algorithm jobs can be preempted at any time. To implement FMLP it is necessary to have the functionality that a job can execute non-preemptively for a fixed duration of time. In this section a variant of G-EDF scheduling algorithm will be presented named *Global Suspendable Non-Preemptive EDF* abbreviated as GSN-EDF which is necessary for the FMLP implementation.

8.3.1 The GSN-EDF Algorithm

In this section the variant of G-EDF will be presented with necessary modifications. At first, we need to discuss some terms and definitions.

- A job T_i^j (j^{th} job of i^{th} task) is eligible for its execution at its release time $r(T_i^j)$ and should finish its execution before its absolute deadline $d(T_i^j)$. A job is *pending* at time t if and only if $t \geq r(T_i^j)$ and the job has not finished its execution. A pending job can be in any of three different states: *suspended*, *preemptable* or *non-preemptable*. A suspended job, also known as blocked, cannot be scheduled on any processor. A preemptable job can be scheduled on any processor and can be preempted by any higher priority job at any time. A non-preemptive job will continue its execution until it finishes its non-preemptive section. A job can only become non-preemptive or suspended when it is being scheduled on some processor. A preemptive or non-preemptive job is named *runnable.* The state transition of a job from suspended to preemptive is known as *resumed* [14].

- A task should have a unique priority which is required by the FMLP. In G-EDF the task sets are prioritized according to their absolute deadline, if two tasks have the same absolute deadline then the tie is broken arbitrarily. Block et al. [14] used $Y(T_i^j) = (d(T_i^j), i)$ where i denotes the task number and $Y(T_i^j)$ is the priority of task T_i^j. They defined the priority between two tasks T_i^j and T_c^d as $Y(T_i^j) > Y(T_c^d)$ if and only if $d(T_i^j) < d(T_c^d)$ or $d(T_i^j) = d(T_c^d) \wedge i < c$.

The GSN-EDF algorithm takes care of the situation that a job can only be blocked by any other non-preemptive jobs when T_i^j is either released or resumed. The above mentioned blocking durations are reasonably constrained. A task T_i^j is said to be non-preemptively blocked at time t if and only if T_i^j is one of the m highest priority runnable jobs but instead of T_i^j a lowest priority non-preemptive job is scheduled [14].

In order to allow jobs to have a non-preemptive section, the naive modification to the GSN-EDF algorithm is necessary. At any time t, the non-preemptive jobs need to be scheduled first. If there are q non-preemptive pending jobs at time t, then these jobs are scheduled at t additionally, if there are k preemptive pending jobs then the $\min(k, m - q)$ highest priority preemptive jobs are scheduled at t.

At any time t, the scheduler selects m highest priority tasks and schedules them on m processors. However, in GSN-EDF [14] used another term that a runnable job is either *linked* to a processor or *unlinked.* According to G-EDF when a job is scheduled on a processor, in GSN-EDF that job T_i^j is linked to a processor at time t. Therefore, at any time t, m highest priority tasks are linked to m processors. The most important event to notice here, if a task T_i^j is linked to a processor, but not scheduled and instead another task is scheduled then task T_i^j is said to be non-preemptively blocked. Consequently, if a task T_a^b is scheduled on a processor but is unlinked and continues its execution though it is not one of m highest priority tasks at time t; the reason is the task is non-preemptable [14].

Block et al. [14, figure 2] have defined the pseudo-code for the GSN-EDF algorithm which has been taken into consideration for the implementation of the GSN-EDF in the existing system. The algorithm is straightforward and is given below for better understanding.

Algorithm 3 Pseudo-code that defines the GSN-EDF algorithm [14]

```
$T_i^j$ is released or resumed at time $t$
$T_a^b$ := lowest-priority linked job (if exists);
if fewer than $m$ jobs are linked then
    $k$ := index of any unlinked processor;
    $T_i^j$ is linked to and scheduled on processor $k$
else if $Y(T_i^j) > Y(T_a^b)$ then
    $k$ := processor $T_a^b$ is linked to;
    $T_a^b$ becomes unlinked;
    $T_i^j$ becomes linked to processor $k$;
    if $T_a^b$ is scheduled and preemptable then
        $T_a^b$ stops being scheduled;
        $T_i^j$ is scheduled on processor $k$;
    end if
end if
$T_i^j$ changes from non-preemptable to preemptable at time $t$
$k$ := processor $T_i^j$ is scheduled on;
$T_a^b$ := job linked to processor $k$;
if $T_a^b$ exists $\wedge$ $T_a^b \neq T_i^j$ then
    $T_i^j$ stops being scheduled
    $T_a^b$ is scheduled on processor $k$;
end if
$T_i^j$ becomes suspended or completes at time $t$
$k$ := processor $T_i^j$ is scheduled on;
$T_i^j$ stops being scheduled;
$T_a^b$ := job linked to processor $k$;
if $T_a^b$ exists $\wedge$ $T_a^b \neq T_i^j$ then
    $T_a^b$ is scheduled on processor $k$;
else
    $T_x^y$ := highest-priority runnable unlinked job;
    $T_i^j$ becomes unlinked;
    if $T_x^y$ exists $\wedge T_x^y$ is not scheduled then
        $T_x^y$ is linked to and scheduled on processor $k$;
    else if $T_x^y$ exists then
        $q$ := processor $T_x^y$ is scheduled on;
        $T_r^e$ := job linked to processor $q$;
        if $T_r^e$ exists then
            $T_r^e$'s link changes from processor $q$ to $k$;
            $T_r^e$ is scheduled on processor $k$;
        end if
        $T_x^y$ becomes linked to processor $q$,
    end if
end if
```

8.3.2 Flexible Multiprocessor Locking Protocol

In this section the *flexible multiprocessor locking protocol* will be discussed. Block et al. [14] used the term “flexible” because it can be used under both partitioned and global scheduling approaches. In this protocol, resources are grouped as short or long which is specified by the user based on the resource worst case access time.

One important observation while implementing multiprocessor locking protocol is how to respond to a resource request which cannot be satisfied immediately. If a task is suspended in such a case, this will impact the blocking time negatively. Block et al. [14] suggested to improve the blocking time by introducing *busy-wait* for short resources where jobs busy-wait non-preemptively. In this case, at any time t the number of jobs that can be blocked on the same resources are $(m-1)$ but the important situation to notice here, the processor may be idle for some short time duration which leads to the waste of valuable processing speed [14].

In order to achieve a high degree of parallelism, they suggest a balance between busy-waiting and suspensions in three different ways [14]:

- A resource can be obtained only for a *long* or *short* duration and busy-waiting can only be employed with short resources. As mentioned earlier a resource can be treated as being in either long or short group which is specified by the user. For resource nesting, a long resource cannot be nested inside a short resource which is a constraint imposed by Block et al. [14].
- The time duration a job spends for busy-waiting is minimized by executing the short resources non-preemptively.
- In order to deal with short, non-nestable resources efficiently, resources are grouped in an effective way which helps to minimize the overhead of deadlock avoidance.

8.3.2.1 Resource Request Rules

In FMLP, the fundamental unit of resource locking is *resource groups*. A resource group is formed by either short or long resources together. The minimum number of resources that a resource group may contain is one. A resource group is protected by a lock which is termed as *group lock*. A resource group of short resources is protected by a non-preemptive queue lock and semaphore is used for long resource groups. Two resources r_1, r_2 belong to the same resource group, if there is a job that has nested requests for a resource, request for r_1 is nested inside the request for r_2 and both r_1 and r_2 are either short or long [14].

The non-nestable resources are grouped individually in FMLP. Thus, FMLP improves parallelism by handling the common case of non-nested resource access efficiently. In regard to the type of resource request, some important terms *outermost request*, *inner request* and *non-nestable resource* are defined. A request for a non-nestable long resource may contain requests for short resources whereas requests for short resources contained in long requests are considered to be short outermost requests [14]. Block et al. [14] refer to the long (short) outermost and inner requests as *l-outermost(s-outermost)* and *l-inner(s-inner)* respectively, in order to avoid confusion.

Short Resource Request: When a job T_i^j requests a short resource r_1, which is an s-outermost resource request, the job must acquire the group lock of the resource group that contains r_1. In order to acquire such group lock the job has to become non-preemptive until it releases the group lock. In a queue lock all other jobs blocked on the same resource group will busy wait in a FIFO order. Any inner resource request will be granted immediately as the inner resources are also in the same resource group by definition. The resource group lock will be released when the s-outermost resource use has been completed [14].

Long Resource Request: When a job T_i^j request a long resource r_1, which is a l-outermost resource request, the job must acquire the group lock of the resource group that contains r_1. In a semaphore lock all other jobs are blocked on the same resource group, pushed into a FIFO queue and suspended. The task T_i^j that holds the group lock of the resource group that contains r_1 will inherit the priority of the highest task that blocked on the same resource group and task T_i^j is scheduled preemptively. Any inner resource request for a long resource will be granted immediately as the inner long resources are also in the same resource group by definition. If the task T_i^j has an inner request for a short resource inside the long resource request then the short resource may be an s-outermost request or an s-inner request, in such case the task has to follow the short resource request protocol described earlier. The resource group lock will be released when the s-outermost resource use has been completed and the priority of the task is restored to its original priority. At this point the first blocked job from the FIFO queue will be taken and resumed [14].

8.3.2.2 Blocking under GSN-EDF with FMLP

A job T_i^j may experience delays due to a shared resource access which is currently not available, locked by any other job. Such delays come from busy-waiting and suspensions or blocking and are not from the preemption happened for a higher priority job being executed instead of that. In GSN.EDF, at any time t, m highest priority jobs are linked to m processors [14]. In normal behavior, a job should be scheduled on the processor to which it is linked. Therefore, the job T_i^j is said to be blocked at time t, if the job is linked to a processor but not scheduled which means the job is either blocked or busy waits. The important point should be noted here, under GSN-EDF a job may be blocked by both lower or higher priority tasks in contrast to uni processor where only lower priority tasks experience blocking [14]. There are three different types of blocking under GSN-EDF which come from three different sources.

- "*Busy-wait blocking* occurs when a job must busy-wait in order to acquire a short resource" [14]. The maximum total amount of time that a job of a task

T_i can busy-wait is denoted $BW(T_i)$ by [14].

- "*Non-preemptive blocking* occurs when a preemptable pending job T_i^j is one of the m highest-priority pending jobs, but is not scheduled because a lower-priority non-preemptable job is scheduled instead (i.e. T_i^j is linked but not scheduled)". The maximum total amount of non-preemptive blocking time that a job of a task T_i can face is denoted $NPB(T_i)$ [14].
- "Direct blocking occurs when a preemptable pending job T_i^j is one of the m highest-priority jobs and it issues a request for an outermost long resource r_1 from Group g, but is suspended because some other job holds a resource from Group g." The maximum total amount of time for which a job of a task T_i can be directly blocked is denoted $BB(T_i)$ [14].

Therefore, the total blocking time that any job of a task T_i suffers from, is denoted $B(T_i)$, which is the sum of the three different blocking times described above, defined as [14]

$$B(T_i) = BW(T_i) + NPB(T_i) + DB(T_i) \tag{8.11}$$

Block et al. [14, Theorem 3] presented and proved a theorem that FMLP is *deadlock-free.*

9 | Implementation

In the previous chapters the theoretical background of this thesis' goal, what should be implemented and the necessary modifications of the system model have been discussed in detail. This chapter focuses on the implementation of the new features that have been integrated into the existing system.

9.1 Schedulability Analysis

The necessary theory for the schedulability analysis of different scheduling algorithms has been studied in detail and discussed in chapter 5. A schedulability analysis runs offline, i.e before starting the simulation, the schedulability analysis module checks the system model and decides whether the task set can be scheduled on the specified processor by the desired scheduling algorithm using the mentioned resources.

Munk [41, section-6.3.4] presented the UML diagram for the scheduler interface `IScheduler` that has to be implemented by each combination of scheduling algorithms and resource access protocols. In this scheduler interface, a function name `checkSystemModel()` is intended to check whether the system model is valid or not. In order to maintain the software design, the schedulability analysis function calls are placed inside this function. After the system model validation, the schedulability analysis is done and delivers the result to the user along with different types of notifications `INFO`, `WARNING`, `ERROR` or `FATAL` according to the enumeration `Severity`. In case of a `FATAL` error the simulation cannot be started [41].

All the new classes or the new functions in the existing classes for schedulability analysis are placed inside the package `de.unistuttgart.iste.ps.savors.scheduler.systemModelCheckProvider`. The system model contains `Tasks` model which has been updated by adding the attribute `executionTime`, which is an optional field. It can be provided by the user in the system model but does not have to. In case it is not provided, the function `updateTasksByExecutionTime(SystemModel systemModel)` has been implemented to update this field for all tasks which is necessary for schedulability analysis. Thus, it is important to check that the values of this field are valid for all tasks.

Listing 9.1: Update all tasks in the task model by setting up the `executionTime` attribute

```
    // go through all tasks and set the execution time
    for (Task task : systemModel.getTask()) {
        int executionTime = 0;
        // go through all commands, take only Execution command and
            sum all Execution commands of current task
        for (Command command : task.getCommand()) {
            // check if the command is execution type then we
            if (command instanceof ExecutionImpl) {
                //add this execution value to the executionTime
                    variable
                     executionTime += ((ExecutionImpl) command).
                         getDuration();
            }
        }
        // set the task execution time
        task.setExecutionTime(executionTime);
    }
```

9.1.1 Utilization Based Analysis

There are two functions written for utilization based schedulability analysis, `utilizationBasedCheck()` and `utilizationBasedCheckDeadlineLessThanPeriod()`, for simplicity the parameters of the functions have not been given. As the function names show the difference between these two functions, they will be clarified further when used.

In order to avoid exceptions in the schedulability analysis functions where the period or deadline of a task is zero, as provided by the system model, the schedulability analysis will not be done and informs the user accordingly.

Listing 9.2: Checking the period and deadline of every task is not zero, if zero then schedulability analysis will not be done

```
    // check necessary condition so that the schedulability test do
        not produce errors, like periods, deadlines are not zero
    if (task.getPeriod() <= 0 || task.getDeadline() <= 0) {
        // produce message that utilization based test is not
            possible and return
        reporter.addFinding(Severity.INFO, Messages.
            PeriodOrDeadlineisZero);
        return false;
    }
```

In general, given the assumption for RMS that the deadline of any task is equal to its period, the task set is synchronous. If the task set follows the assumption, then the schedulability analysis can be done by using the function `utilizationBasedCheck()` from the class `FixedPriorityPreemptiveSystemModelCheckProvider`, if utilization based test fails for RMS then response time analysis has to be done. If the task set that is going to be scheduled using RMS, does not follow the assumption, e.g. some task deadlines are not equal to its period or task set is asynchronous then, the schedulability analysis cannot be done using utilization based schedulability analysis, so response time analysis is necessary.

In DMS, the deadline of a task may be equal to or less than its period, the utilization based schedulability for DMS is done by the function `utilizationBasedCheckDeadlineLessThanPeriod()` from the same class. If utilization based Schedulability fails for DMS then response time analysis is necessary.

For utilization based schedulability analysis for the dynamic priority scheduling algorithm EDF, the function `utilizationBasedCheck()` is called from the class `DynamicPrioritySystemModelCheckProvider`. The schedulability analysis of EDF can be done using this function when the deadlines of all tasks are implicit. If there is a task having a deadline less than its period or the utilization based schedulability analysis for EDF has failed, then demand bound analysis has to be done to check the schedulability for EDF.

In schedulability analysis a necessary condition is defined that utilization cannot be more than 1. If this necessary condition fails, utilization is more than 1, then a `FATAL` error is triggered and simulation cannot be started. In all other cases, if the utilization based test is successful then no message will be triggered. If any test fails then a message will be triggered to inform the user that the test has failed and provide information about the next step which is going to be done.

Listing 9.3: Checking the necessary condition for schedulability

```
    if (utilization > 1.0)
    {
        reporter.addFinding(Severity.FATAL, Messages.
           UtilizationMoreThanOne);
        return false;
    }
```

9.1.2 Response Time Analysis

In order to perform response time analysis, the function `responseTimeAnalysis()` is provided in three different classes to serve three different purposes.

The exact schedulability analysis has been done by the `responseTimeAnalysis()` function, which is written in the `FixedPriorityPreemptiveSystemModelCheckProvider`. If the utilization based schedulability analysis fails then the response time analysis function from this class will be called which does not consider any synchronization between tasks, i.e the blocking time for shared resource use will not be considered.

Listing 9.4: Exact schedulability analysis without task synchronization for shared resource access inside `CheckSystemModel()`

```
    if(FixedPriorityPreemptiveSystemModelCheckProvider.
       utilizationBasedCheck(reporter, systemModel))
    {
        // necessary and sufficient test, consider only execution
        FixedPriorityPreemptiveSystemModelCheckProvider.
           responseTimeAnalysis(reporter, systemModel);
    }
```

The version of response time analysis that supports the synchronization and communication between tasks is a sufficient schedulability analysis which is not necessary. The implementation of this version of response time analysis varies for different resource access protocols. For the *simple priority inheritance* protocol a version of this function is implemented in class `SimplePriorityInheritanceSystemModelCheckProvider` and for *Priority Ceiling Protocol* a version of this function is implemented in class `PriorityCeilingSystemModelCheckProvider` where the calculation of blocking time is different for different resource access protocols. The following listings show the calculation of blocking times and thus how response time analysis is done from the `CheckSystemModel()` function.

Listing 9.5: Blocking time calculation for *simple priority inheritance*

```
    // calculate the blocking time of each task
    int blockingTime = 0;
    for(ResourceUsage resourseUsage : taskWithPriority.getTask().
       getResourceUsage())
    {
        blockingTime += resourseUsage.getResource().getAccessTime()
            * resourseUsage.getUsage();
    }
```

Listing 9.6: Blocking time calculation for *priority ceiling protocol*

```
    // calculate the blocking time of each task
    int blockingTime = 0;
    for(ResourceUsage resourseUsage : taskWithPriority.getTask().
       getResourceUsage())
    {
        blockingTime = Math.max(blockingTime, resourseUsage.
           getResource().getAccessTime() * resourseUsage.getUsage()
           );
    }
```

Listing 9.7: Schedulability analysis with task synchronization for shared resource access inside `CheckSystemModel()`

```
    if(FixedPriorityPreemptiveSystemModelCheckProvider.
       utilizationBasedCheck(reporter, systemModel))
    {
        // necessary and sufficient test, consider only execution
        FixedPriorityPreemptiveSystemModelCheckProvider.
           responseTimeAnalysis(reporter, systemModel);
    }
    // sufficient test, also consider blocking time
    PriorityCeilingSystemModelCheckProvider.responseTimeAnalysis(
        reporter, systemModel);
```

In case a task may violate its deadline as found by the response time analysis, this will be notified to the user by `WARNING` message.

9.1.3 Demand Bound Analysis

In order to perform the demand bound analysis for a task set that will be scheduled under EDF, the function `demandBoundCheck()` from the class `DynamicPrioritySystemModelCheckProvider()` can be used.

Listing 9.8: Exact schedulability analysis for EDF without task synchronization for shared resource access inside `CheckSystemModel()`

```
    // at first call the utilization based schedulability check
    if(DynamicPrioritySystemModelCheckProvider.
        utilizationBasedCheck(reporter, systemModel))
    {
        // if the utilization based schedulability check fails then
            demand bound analysis is done
        DynamicPrioritySystemModelCheckProvider.demandBoundCheck(
            reporter, systemModel);
    }
```

The important piece of information to be listed here is that the demand bound analysis is an exact schedulability analysis for dynamic priority scheduling algorithm EDF that does not consider the blocking time occurred during shared resource usage.

9.2 Self Suspension

There are three new model classes that have been created for self suspension `InitiateSelfSuspension`, `Suspension`, `TerminateSelfSuspension` which have been presented in section 4.1.1.1 on page 29. In order to implement the self suspension functionality for a task, these three subclass instances of `Command` model must come in the order they are listed here. A listing of the system model which contains a task that will have self suspension is given below. The task W will execute for one time unit and then suspend for one time unit. When it completes the self suspension of one unit, it will again execute for one time unit.

Listing 9.9: Representing self-suspension in the system model

```
    <task name="W" deadlineType="HARD" deadline="4" offset="0"
      period="4" executionTime="2" repetitions="-1">
     <command xsi:type="model:Execution" duration="1"/>
     <command xsi:type="model:InitiateSelfSuspension"/>
     <command xsi:type="model:Suspension" duration="1"/>
     <command xsi:type="model:TerminateSelfSuspension"/>
     <command xsi:type="model:Execution" duration="1"/>
    </task>
```

There are two new jobs associated with the above mentioned two model classes. They are `InitiateSelfSuspensionJob` and `TerminateSelfSuspensionJob` related to `InitiateSelfSuspension` and `TerminateSelfSuspension` respectively. Task monitor is responsible to create the next job on the `TimeAxis`. Like other jobs, task monitor also creates the task job `InitiateSelfSuspensionJob` on the `TimeAxis` but `TerminateSelfSuspensionJob` is not created from the `TaskMonitor`. When the job `InitiateSelfSuspensionJob` is executed from the `TimeAxis`, the `execute()` function of the `InitiateSelfSuspensionJob` is executed. This function changes the state of the task from `running` to `blocked`, and calls the function `createTerminateSelfSuspensionJobTask(long time)` of `TaskMonitor` class which is responsible for creating `TerminateSelfSuspensionJob`. This function takes the value from the `Suspension` model, which tells how many time units the task wants to be in suspension. Then it calculates the time, meaning the time in `TimeAxis`, where the `TerminateSelfSuspensionJob` should be created. This function is also responsible to change the blocking type to `SELF_SUSPEND`, and update the `commandNumber` attribute of `TaskMonitor` correctly.

Listing 9.10: Creation of `TerminateSelfSuspensionJob` from `TaskMonitor`

```
    // check the command is a type of TerminateSelfSuspension
      final Command command2 = task.getCommand().get(
        commandNumber);
      if (command2 instanceof TerminateSelfSuspension) {
        // create TerminateSelfSuspensionJob
        IJob terminateSelfSuspensionJob = new
          TerminateSelfSuspensionJob(task);
        // add the job to the time axis
              Simulator.getInstance().getTimeAxis().
                addJob(time+suspensionTime,
                terminateSelfSuspensionJob);
              // update the command number
              commandNumber++;
      }
```

When a task initiates a self suspension, a state transition is done from *running* to *blocked*, the task is blocked without any resource access. For such blocking a new blocking type is introduced for visualization which is `SELF_SUSPEND`. When the task completes its self suspension, a state transition from *blocked* to *ready* is done. Execution of any self suspension job requires a dispatch. The important method of the `IJob` interface is `isDispatchNecessary()` which tells the simulator whether a dispatch is necessary or not. Both jobs for self suspension return true to tell the simulator that a dispatch is necessary.

9.3 Non-preemptive Execution

Like self-suspension, two new model classes `InitiateNonPreemptive` and `TerminateNonPreemptive` have been added to support non-preemptive execution. The execution time will be split by only execution, execution with resource access and non-preemptive execution as explained in section 4.1.1.1 on page 29. Non-preemptive execution should be started with `InitiateNonPreemptive Command` and ended with `TerminateNonPreemptive Command`. A listing of the system model which contains a task that has a non-preemptive section for execution is given below. The task $I2$ will preemptively execute for two time units and then non-preemptively execute for two time units and finally it will execute for one time unit preemptively. When the task executes preemptively any higher priority task can preempt it. In contrast, when non-preemptive task is executed no higher priority task can preempt it.

Listing 9.11: Representing non-preemptive section in the system model

```
    <task name="I2" deadlineType="HARD" deadline="10"  offset="0" period="500" repetitions="-1">
    <command xsi:type="model:Execution" duration="2"/>
      <command xsi:type="model:InitiateNonPreemptive"/>
      <command xsi:type="model:Execution" duration="2"/>
      <command xsi:type="model:TerminateNonPreemptive"/>
      <command xsi:type="model:Execution" duration="1"/>
  </task>
```

There are two new jobs `InitiateNonPreemptiveJob` and `TerminateNonPreemptiveJob` for the above described two models respectively. These jobs are created and sent to `TimeAxis` from the `TaskMonitor` class except it updates the `commandNumber` attribute. The `commandNumber` attribute is updated from the functions `initiateNonPreemptiveRequestSuccessful()` and `terminateNonPreemptiveRequestSuccessful()` of `TaskMonitor` class, they are called from the `execute()` method of `InitiateNonPreemptiveJob` and `TerminateNonPreemptiveJob` respectively.

A new `boolean` type attribute `nonPreemptiveExecution` is added to the `TaskMonitor` class which is updated by the above mentioned two non-preemptive jobs. This attribute is checked by the scheduler to know whether a task is executing its non-preemptive section or not. This `nonPreemptiveExecution` attribute is set to `true` from the `InitiateNonPreemptiveJob` which means the respective task is executing its non-preemptive section. It is set to `false` from `TerminateNonPreemptiveJob` to indicate that non-preemptive execution has been finished. When a dispatch is necessary, the `dispatch()` method of the scheduler is called, it checks inside the dispatch whether a task is non-preemptive or not. If a task is non-preemptive then the scheduler does not perform the preemption, although there is a higher priority ready task. How exactly the scheduler checks non-preemptive execution is explained below:

Listing 9.12: Checking non-preemptive execution inside the dispatch method of the scheduler taken from *global-EDF*

```
// check the priority of the ready and compare it with running task
if (runningTaskDeadline > inspectTaskDeadline
// This condition is added for non-preemptive access, if non-preemptive job will not be preempted, simply keep running
&& !TaskManager.getMonitorForTask(runningTask).isNonPreemptiveExecution())
{
    // Preemption!
    // preempt the running task
    TaskManager.getMonitorForTask(runningTask).makeReady();

    // Preempted task is immediately put on the ready queues,
    readyQueues.add(runningTask, runningTaskDeadline,   false);
    coreTaskAssignment.put(core, inspectTask);
    // start the selected task
    TaskManager.getMonitorForTask(inspectTask).run(core);

    assignmentFound = true;
}
```

9.4 Partitioned EDF

In section 8.1, some heuristic methods for task partitioning have been discussed. In practice, to implement EDF into the existing system the tasks partitioning part has to be done by the user explicitly. In the system model, the task partitioning has to be done by the user as part of preparing the system model for the simulator. The task partitioning is described below with the help of the system model listing.

Listing 9.13: Task partitioning in the system model for multiprocessor partitioned scheduling approach

```
<task name="1.I1" ...>
    <command xsi:type="model:Execution" duration="1"/>
</task>
<task name="1.I2" ... >
    <command xsi:type="model:Execution" duration="5"/>
    ...
</task>
<task name="2.A" ...>
    <command xsi:type="model:Execution" duration="2"/>
    ...
</task>
<task name="2.B" ...>
    <command xsi:type="model:Execution" duration="3"/>
</task>
```

```
    <core name="Core 1"/>
    <core name="Core 2" resourceAccessProtocol="Stack Resource
       Protocol (Baker's)"/>
```

The tasks are partitioned to the processors by their names. The task name will start with an integer number and a dot. The integer number represents the core on which the task will be executed. In the implementation of P-EDF, the tasks are restricted in such a way that two different tasks cannot share a single resource from two different processors. For this reason resources are also bound to a core from which they can be accessed. The resource will be named in the same way as the task name where the integer number indicates the core to which a resource will be linked and the resource will only be allowed to be accessed by the tasks linked to that core.

A resource access protocol can be specified in the simulation model with the scheduler name. If no name is specified then the default resource access protocol will be used. In order to specify the resource access protocol for each core, the `resourceAccessProcol` attribute of `Core` model will be used as discussed in section 4.1.2 on page 30. Any resource access protocol supported by the EDF uni-processor scheduling algorithm can be used for a core. If no resource access protocol is specified for a core then the default resource access protocol will be used for that core. The following listing shows the way partitioned EDF initializes an EDF algorithm for each core with specified resource access protocol for that core.

Listing 9.14: P-EDF initializes a single core EDF algorithm for each core

```
// assign an EDF algorithm with specified resource access protocol
   for each core
for (Core core : Simulator.getInstance().getSystemModel().getCore()
   ) {
    if (core.getResourceAccessProtocol() == null || core.
       getResourceAccessProtocol().equals("")) {
        EDF scheduler = new EDF();
        scheduler.initialize(core);
        coreScheduler.add(scheduler);
    } else if (core.getResourceAccessProtocol().equals(Messages.SRP
       )) {
        EDFWithSRP scheduler = new EDFWithSRP();
        scheduler.initialize(core);
        coreScheduler.add(scheduler);
    }
}
```

At this point, the partitioned EDF scheduler needs to run a single core EDF scheduler for each core with the specified resource access protocol. The P-EDF scheduler does this in its `initialize()` method and keeps track of schedulers on each core in `coreScheduler` attribute which is a Java array list of `IScheduler`. When a dispatch is necessary, P-EDF calls the dispatch method of the associated scheduler of each core from its own `dispatch()` method.

The P-EDF algorithm validates the system model from the `checkSystemModel()` function. It checks the task names and resource names to see whether the names are given according to the convention. It also checks that a resource is not accessed from a task that is not linked to this core. If the two above described system model validations fail then the simulator triggers an `ERROR` message and the simulation cannot be started. The schedulability analysis for each core is done by using the single core schedulability analysis methods for EDF.

9.5 Proportionate Fairness

In contrast to all other scheduling algorithms that have been implemented, the class of P-fair scheduling algorithms contains job level dynamic priority scheduling algorithms. As discussed in section 8.2 on page 58, the variants of P-fair scheduling algorithms use the same logic to categorize tasks into three different categories *urgent*, *tnegru* and *contending*. An abstract class `Pfair` has been created by implementing the `IScheduler` interface. This abstract class serves as a super class for all the P-fair variants, namely PF, PD and PD^2 scheduling algorithms.

P-fair scheduling algorithm needs to have a dispatch at every integer time t which is not possible with the existing simulation kernel. With the existing system, a scheduler gets a dispatch when it is necessary to get a dispatch determined from the executed jobs, e.g. the scheduler gets a dispatch when a task is ready, a task is blocked on any resource, etc. In order to get a dispatch at every time t, a new job is added named `DispatchJob`. The `Pfair` scheduler adds a `DispatchJob` to the `TimeAxis` at time $t + 1$ from its `dispatch()` method to get a dispatch at the next integral point of time when it gets a dispatch at time t.

Listing 9.15: Generating a dispatch job from `Pfair` for every time unit t

```
    // need to check whether there is a task to schedule at the
       next point or not
    // create the Dispatch job to dispatch at next integer time
    if (!TaskManager.areAllTasksNonExisting() && time !=
       previousDispatchTime) {
        DispatchJob dispatchJob = new DispatchJob();
        Simulator.getInstance().getTimeAxis().addJob(time + 1,
           dispatchJob);
    }
```

The functionality of `DispatchJob` is nothing but to generate a dispatch on that point of time by the simulator. Every job has a function `isDispatchNecessary()` which is called from the time axis to decide whether a dispatch is necessary or not. This function returns true and thus the `dispatch()` method is called.

The abstract class `Pfair` implements the function `stateChangeRequest(long time, Task task, State newState)` which will be used by all sub-classes. `Pfair` has some `protected` attributes that will be updated from this class and will be used from its subclasses. `Pfair` uses a Java `HashMap` instance `characteristicStringOfTask` to store the characteristic string of each task at time t. The following listing shows the calculation of characteristic sting done by `Pfair` according to Baruah et al. [12].

Listing 9.16: Calculation of characteristic string in `Pfair`

```java
// calculate characteristic string for each task at this point of
    time
for (Task task : Simulator.getInstance().getSystemModel().getTask()
    ) {
     double csValue = 0;
     String cString;
     double taskWeight = (double) task.getExecutionTime() / (double)
          task.getPeriod();
     // equation to calculate characteristic string on each point by
          Baruah et. al.
     // alpha(x) = sign(x.w * (t + 1) - Floor(x.w * t) - 1)
     csValue = Math.sin((taskWeight * (double) ((double) time + 1.0)
         ) - Math.floor(taskWeight * (double) time) - 1.0);
     if (csValue < 0.0)
          cString = Messages.NegativeString;
     else if (csValue > 0.0)
          cString = Messages.PositiveString;
     else
          cString = Messages.NeutralString;
   // update the hashmap to store the characteristic string
     characteristicStringOfTask.put(task, cString);
}
```

The most important attributes `urgetTasks`, `tnegruTasks` and `contendingTasks` are Java lists of type `Task` that contains the tasks of three different categories. In the `dispatch()` function of this `Pfair` class, the characteristic strings and lags of all tasks are calculated and based on these two values the tasks are categorized and stored accordingly.

Listing 9.17: Categorize tasks into `urgetTasks`, `tnegruTasks` and `contendingTasks`

```java
     // check all the tasks that need to be scheduled and add to
         urgent, tnegru, or contending task list
     for (Task task : schedulingTasks) {
          if (lagMulPeriodOfTask.get(task) > 0 && !
              characteristicStringOfTask.get(task).equals(   Messages.
              NegativeString)) {
               //  the task is urgent
               urgentTasks.add(task);
          } else if (lagMulPeriodOfTask.get(task) < 0 && !
              characteristicStringOfTask.get(task).equals(Messages.
              PositiveString)) {
               //  the task is tengru
               tnegruTasks.add(task);
          } else {
               // the task is contending
```

```
                contendingTasks.add(task);
            }
        }
```

In order to prove that a generated schedule is P-fair, it must be proven to obey the lag constraint. At each dispatch the lag of every task should be checked so that it does not violate the constraint. `Pfair` provides a function `checkLagConstraint()` to check the lag constraint which is called at the beginning of every dispatch.

All variants of Pfair scheduling algorithm are able to access shared resources by default resource access protocol.

9.5.1 Algorithm PF

The class `PF` has been implemented by extending the abstract class `Pfair` to implement the algorithm PF. `PF` calls the `initialize()` method from the super class inside its own `initialize()` method, in order to initialize all the variables of the super-class. If necessary, it will initialize its own variables inside its own initialize method after invoking the super-class method. In this case all variables are declared in the super-class. At every integral point of time *t*, when a dispatch is necessary, `PF` first calls the `dispatch()` method from the super-class which categorizes all the tasks into urgent, tnegru and contending task lists and stores them accordingly.

At this point, *algorithm PF* needs to sort the contending tasks according to the total order by using any of the two comparison algorithms described in section 8.2.1.1 on page 62. Both algorithms have been implemented inside `PF` class as functions, namely `naiveCompare(Task task0, Task task1)` and `compare(int a0, int b0, int c0, int a1, int b1, int c1)`. These two algorithms have been describes in algorithm 1 on page 63 and algorithm 2 on page 63 respectively. In the implementation of the algorithm 1 is slightly different in the function parameters, instead of six parameters it takes the two tasks it needs to compare and calculate all six original parameters inside the function.

Listing 9.18: Calculating six parameters for the implementation of algorithm 1, `NaiveCompare` [12]

```
    // the values need to be passed inside the function, calculated
        inside here
    int a0, b0, c0, a1, b1, c1;
    // a0 = x.p - x.e
    a0 = task0.getPeriod() - task0.getExecutionTime();
    a1 = task1.getPeriod() - task1.getExecutionTime();
    // b0 = x.e
    b0 = task0.getExecutionTime();
    b1 = task1.getExecutionTime();
    // c0 = x.p * lag(S,x,t) + 2 * x.e - x.p
    // we have lag * period, we used that instead of {x.p * lag(S,x
       ,t)}
    c0 = lagMulPeriodOfTask.get(task0) + 2 * task0.getExecutionTime
       ()   - task0.getPeriod();
    c1 = lagMulPeriodOfTask.get(task1) + 2 * task1.getExecutionTime
       ()   - task1.getPeriod();
```

Both of these functions take two tasks `task0` and `task1` as input, the order of the parameters is important to decide the tasks' priority. Let's assume `task0` is the first parameter and `task1` is the second parameter. The functions compare and return the result of comparison in the following way, 0 => [`task0` > `task1`], 1 => [`task0` < `task1`], 2 => [`task0` = `task1`] and finally -1 => [`an error occurred during comparison`], which should not happen for these algorithms. For safety, in case of any bug inside the code this error value is returned. The return values are slightly different than in the algorithm for the implementation purposes, however, the functionality is the same.

In order to insert the tasks from the `contendingTasks` list to a list where the task will be arranged according to the priority in descending order, `Pfair` uses a Java `LiknedList` instance, `contendingTaskLinkedList`. In this linked list, by comparing two tasks, if `task0` has higher priority than `task1` then `task0` is placed before task `task1`, if `task1` has higher priority than `task0` then `task1` is placed before task `task0` and finally if both tasks have the same priority then the tie can be broken arbitrarily, the order of these two tasks can be arbitrary.

When the contending tasks are prioritized and sorted according to the priorities, the `PF` can schedule tasks according to the algorithm PF described in table 8.1 on page 62. All the urgent tasks will be scheduled from the `urgentTasks` task list. If the number of urgent tasks n_0 is less than m, the number of processors, then the first $m - n_0$ tasks will be scheduled from the contending task linked list. In P-fair, the lag of every task is important and needs to update the lag of all tasks at every integral point of time t to the respective variable. In P-fair implementation, in order to have an integer number, the lag will be stored multiplied by its period in a Java `HashMap` named `lagMulPeriodOfTask`. At every dispatch the lag value for each task needs to be updated according to Baruah et al. [12]. The implementation for the lag update is given by the following listings:

Listing 9.19: Updating lag multiplied by period value when a task is scheduled [12]

```
    // update task lag, if a task is scheduled then add
    lagMulPeriodOfTask.put(runTask, lagMulPeriodOfTask.get(runTask)
        - (runTask.getPeriod() - runTask.getExecutionTime()));
```

Listing 9.20: Updating lag multiplied by period value when a task is not scheduled [12]

```
    // update task lag, if a task is scheduled then add
    lagMulPeriodOfTask.put(task, lagMulPeriodOfTask.get(task) +
       task.getExecutionTime());
```

9.5.2 Algorithm PD

Like PF, the class PD has been implemented by extending the abstract class Pfair for the implementation of algorithm PD. Algorithm PD uses the attributes and methods from its super class and also it has some local attributes, six instances of Java HashMap to arrange the contending task list in six different categories mentioned in table 8.2 on page 65. These local variables are initialized after the call to super class `initialize()` method. When a dispatch is necessary it first calls the super-class `dispatch()` method and gets the urgent, tnegru and contending tasks in three different lists. As in PF, it is necessary to sort the contending tasks according to the total order which is different for PD.

PD calculates the pseudo-deadline d_i, characteristic string $\beta_t(T_i)$ and the integer value k for each task at every integral point of time t and uses a Java HashMap instance `contendingTaskWithPDAttributes` to store this value which maps to the corresponding task T_i. These three values for each task will be called PD attributes in this thesis, those are necessary to determine the total order. The calculation of PD attributes is different for heavy tasks and light tasks.

Listing 9.21: Calculate PD attributes for heavy tasks

```
        // calculate the pseudo-deadline, characteristic string and k for each heavy task and add to the hash map
        characteristicString = calculateCharacteristicString(csTime, 1.0 - taskWeight);
        while(characteristicString != Messages.NeutralString && characteristicString != Messages.PositiveString)
        {
                csTime = csTime + 1;
                characteristicString = calculateCharacteristicString(csTime, 1.0 - taskWeight);
        }
        // at position 0, stores $d_i$
        pdAttributes.add(csTime);
        // at position 1, stores $\beta_(T_i)$
        pdAttributes.add(characteristicString);
        // at position 2, stores $T_i.k$
        pdAttributes.add(Math.floor(contendingTask.getPeriod()/(contendingTask.getPeriod() - contendingTask.getExecutionTime())));
        // add put into the hash map
        contendingTaskWithPDAttributes.put(contendingTask, pdAttributes);
```

Listing 9.22: Calculate PD attributes for light tasks

```
    // calculate the pseudo-deadline, characteristic string and k for each light task and add to the hash map
     characteristicString = calculateCharacteristicString(csTime, taskWeight);
        while(characteristicString != Messages.NeutralString && characteristicString != Messages.PositiveString)
        {
                csTime = csTime + 1;
                characteristicString = calculateCharacteristicString(csTime, taskWeight);
        }
        // at position 0, stores d_i
        pdAttributes.add(csTime);
        // at position 1, stores β(T_i)
        pdAttributes.add(characteristicString);
        // at position 2, stores T_i.k
        pdAttributes.add(Math.floor(contendingTask.getPeriod()/contendingTask.getExecutionTime()));
        //add put into the hash map
        contendingTaskWithPDAttributes.put(contendingTask, pdAttributes);
```

The algorithm PD now checks the contending tasks from the list `contendingTasks` and places them in the appropriate category according to the rules listed in table 8.2 on page 65. When the contending tasks are categorized into six different categories, they are combined to the `contendingTaskLinkedList` according to the priority, highest priority category will be added first as mentioned by [11] where lowest numbered category has the higher priority.

At this point algorithm PD follows the same procedure as algorithm PF, it schedules all urgent tasks n_0 and if $n_0 < m$ then first $m - n_0$ tasks from the contending task linked list. It follows the same rule to update the lag and period multiplication values of the `HashMap` instance `lagMulPeriodOfTask` as `PF`.

9.6 Global Suspendable Non-Preemptive EDF

This section describes the implementation of the variant of Global-EDF scheduling algorithm known as *global suspendable non-preemptive EDF*, or abbreviated GSN-EDF. This algorithm needs a simulator that supports task suspension which is also known as blocking. Also, a task is allowed to execute some period of time non-preemptively, no higher priority tasks can preempt a task while it is executing its non-preemptive section. The existing simulator support task suspension which is termed blocking on shared resource access. The implementation of the non-preemptive section is discussed in section 9.3 on page 79, therefore, now the current simulator also supports non-preemptive execution in a preemptive scheduling algorithm.

In GSN-EDF, the scheduler needs to take scheduling decisions when a task changes states from the following three cases; at any time t, when a task (i) is released or resumed, (ii) changes from non-preemptable to preemptable and (iii) becomes suspended or completes, described in algorithm 3 on page 68 [14, figure 2].

When a task is released or resumed, a state transition is triggered from different states to *ready* state, according to a valid state transition. Similarly, when a task is blocked or completed, a state transition is triggered from different states to *blocked* or *terminated* state respectively, according to a valid state transition. For such state transitions, the simulator works accordingly and the scheduler gets a dispatch if it is necessary.

There is no state transition in the system when a task changes from non-preemptable to preemptable. It would be better to have such a state transition in the system. As there is no such state transition in the existing simulator, a new state is not introduced in order to keep the original state transition working as it is described by Munk [41] and to avoid modifications in the core of the simulator, the simulation kernel. However this situation is handled in a different way. A new `boolean` type attribute `nonPreemptiveToPreemptive` is added to the `TaskMonitor` class, which is initially `false` and set to true when a task transits from non-preemptive to preemptive execution. When a task executes the `TerminateNonPreemptiveJob`, the `execute()` function sets this `nonPreemptiveToPreemptive` to `true` and the `isDispatchNecessary()` function returns true to tell the simulator that a dispatch is necessary. This attribute `nonPreemptiveToPreemptive` is checked from the dispatch routine to determine that the task changes from non-preemptable to preemptable and processes according to the GSN-EDF algorithm.

Priority Queue: it serves the purpose of a queue to store all ready tasks according to the task priority. The mechanism used to break a tie, where two tasks have the same priority, is FIFO. As discussed in section 8.3.1 on page 66, GSN-EDF requires unique priority, if there is a tie with the deadline values then the lowest numbered task, the serial number of the task is considered here, gets higher priority. A variation of the existing `add(final Task task, final int priority, final boolean tail)` function, `addInTaskOrder(final Task task, final int priority, boolean lowestIndexedTaskHashighestPriority)` has been implemented which adds the task to the priority queue according to the priority definition in GSN-EDF [14].

The implementation of GSN-EDF uses two instances of Java HashMap, `linkedTaskOnCore` and `scheduledTaskOnCore` to keep track of tasks linked on processors and scheduled on processors respectively. At any time t, it is necessary to know the linked and scheduled tasks on processors. It is also necessary to visualize linked tasks on processors along with scheduled tasks on processors. The event of tasks running on core is triggered from the `TaskMonitor`, which it knows from the state transitions. As there is no state transition for task linked on core, the event for task linked on processor is done from the `dispatch()` method of GSN-EDF scheduler.

In GSN-EDF, two instances of priority queue `linkedJobReadyQueue` and `unLinkedJobReadyQueue` are used to store two types of tasks, linked tasks and unlinked runnable tasks respectively. `linkedJobReadyQueue` stores the tasks that are linked to any processor, the length of this queue is equal to the number of processors. `unLinkedJobReadyQueue` stores all other ready tasks and non-preemptively running tasks not linked to any processor.

The GNS-EDF algorithm is described in a way that would be easier to implement if the `dispatch()` method is called on each state transition whenever a dispatch is necessary. However, the existing simulation software executes all `Jobs` at any time t and calls the dispatch routine if necessary. In order to handle such a difference, a Java `ConcurrentLinkedQueue` instance `readyTasks` is used to store all ready tasks and also the same type instance `blockedOrTerminatedTasks` to store all blocked or terminated tasks in order to process them according to the algorithm when it gets a dispatch call. There are concurrent accesses on the above mentioned two task lists, if Java `List` is used instead, they will get concurrent modification exception from the java exception class `java.util.ConcurrentModificationException`. Although `ConcurrentLinkedQueue` instances are used to store ready tasks and blocker or terminated tasks, these two instances will be referred to as ready task lists and blocked or terminated task lists from now for simplicity.

T_i^j is released or resumed: When a task is ready, then it is inserted into the list `readyTasks` from the `stateChangeRequest()` method. At first it checks that if there is any core, that is not linked to any tasks then it takes a task from ready tasks, linked and scheduled on that core, this process continues until all cores are linked by a task. When a task is linked to a core, it must be pushed into the priority queue `linkedJobReadyQueue`. If there are more ready tasks in the list then the remaining ready tasks are processed according to the section 1, line 1 to line 13 of algorithm 3.

T_i^j changes from non-preemptive to preemptive: If a task is finished its non-preemptive execution and needs to execute preemptively then the dispatch routine executes the section 2, line 14 to line 19 of algorithm 3.

T_i^j becomes suspended or finished: A task may be suspended at any time and resumes later on. In order to schedule a task that is suspended and going to be resumed, GSN-EDF checks on which core it was scheduled previously and tries to schedule on that core. Therefore, GSN-EDF uses a Java `HashMap` instance `preemptedTaskOnCore` to store the scheduled core when a task is preempted. When a task is blocked or terminated, it is inserted into the list `blockedOrTerminatedTasks` from the `stateChangeRequest()` method. These tasks are processed according to the section 3, line 20 to line 39 of algorithm 3.

There is a call to the `dispatch()` method itself inside this method for the implementation of GSN-EDF. The dispatch method starts the processing with ready tasks, then non-preemptive to preemptive tasks and finishes by processing blocked or terminated tasks. Meanwhile, a task may be in ready state from blocked state. Therefore, the dispatch method checks at the end if there is any ready task then calls itself.

Listing 9.23: Dispatch method calls itself to process ready tasks

```
     // Need to check there are ready tasks that were not processed, if any then process them
        if(!readyTasks.isEmpty())
        {
                // dispatch again for the remaining ready tasks
                this.dispatch(time);
        }
```

9.7 Flexible Multiprocessor Locking Protocol

The class `GSNEDFWithFMLP` is introduced by extending the class `GSNEDF` for the implementation of multiprocessor scheduling algorithm, *global suspendable non-preemptive EDF* with the *flexible multiprocessor locking protocol.* The `GSNEDFWithFMLP` class only implements the `requestResource()` and `releaseResource()` methods, all other required methods will be inherited from the super class.

As required by the FMLP implementation a resource needs to be marked `short` or `long` which is done by the user in the system model. In order to make it simple in the implementation, the system model also needs to specify each `RequestResource` or `ReleaseResource Command` by `s-outermost`, `l-outermost`, `s-inner` or `l-inner` in the `resourceNestedType` attribute. In the following listing a system model for task A is given that uses resource and the resource nested type is specified with each request.

Listing 9.24: Resource nested type in `RequestResource` and `ReleaseResource`

```
      <task name="1.I2" deadlineType="HARD" deadline="10"  offset="0" period="500" repetitions="-1">
      <command xsi:type="model:Execution" duration="1"/>
            <command xsi:type="model:RequestResource" resource="/0/@resource.0" resourceNestedType="L_OUTER"/>
      <command xsi:type="model:Execution" duration="1"/>
            <command xsi:type="model:RequestResource" resource="/0/@resource.1" resourceNestedType="L_INNER"/>
      <command xsi:type="model:Execution" duration="1"/>
      <command xsi:type="model:ReleaseResource" resource="/0/@resource.1" resourceNestedType="L_INNER"/>
      <command xsi:type="model:ReleaseResource" resource="/0/@resource.0" resourceNestedType="L_OUTER"/>
    </task>
```

FMLP needs the resource group based on the rules described in section 8.3.2.1 on page 69. The resource group should be done by the user and must provide the information in the system model. In the following listing three resource groups are provided. First group contains two resources, and the rest two groups contains one resource in each group.

Listing 9.25: Providing resource group in the system model

```
   <resourceGroup resource="/0/@resource.0 /0/@resource.1"/>
    <resourceGroup resource="/0/@resource.2"/>
    <resourceGroup resource="/0/@resource.3"/>
```

In order to access a resource, the FMLP requires a *group lock*. In order to *lock* a resource, ResourceMonitor provides a method lock() and to *unlock* it provides another method unlock(). For FMLP locking a variants of these two methods fMLPLock() and fMLPUnlock() have been written. As resource monitor only monitors one resource, it is not possible to get the group lock from resource monitor, because a resource group may contain more than one resource. In order to grant a FMLP locking, ResourceMonitor checks some necessary information about the resource group in the ResourceManager class which can be accesses by any other ResourceMonitor. The following listing shows when a group lock is granted then how ResourceMonitor update ResourceManager attributes.

Listing 9.26: Update ResourceManager on FMLP group lock

```
  // check the group is free
    if(!resourceGroupObj.isLocked()){
      // lock the group
      ResourceManager.addGroupLock(resourceGroupObj,
         lockingTask);
      return resourceGroupObj;
    }   else {
        // add the task to the waiting list of this group
        ResourceManager.addTaskToBlockedList(
          resourceGroupObj, lockingTask);
        }
```

If the resource group can be locked then the addGroupLock(resourceGroupObj, lockingTask) method from ResourceManager is called which is responsible to update its attribute resourceGroupMapsLockedTask that maps a resource group and the locking task. If the resource cannot be locked then the addTaskToBlockedList(resourceGroupObj, lockingTask) method from the same class will be called which insert the task in to the blocked task list which is maintained as FIFO queue. ResourceManager maintain an attribute blockedTasksOnResourceGroup which maps between the resource group and the FIFO queue that contains the tasks blocked on that resource group.

The following listing shows when a FMLP group lock is released then the task monitor tells task manager to release the group lock.

Listing 9.27: Update ResourceManager on FMLP group lock release

```
  // check the group is locked
    if(resourceGroupObj.isLocked()){
      // unlock the group lock
      ResourceManager.removeGroupLock(resourceGroupObj,
         lockingTask);
      return resourceGroupObj;
```

```
        }
```

In FMLP resource request method, when a task needs to request for a resource it checks the resource nested type, if it is a *s-outermost* request then it sets the non-preemptive execution and requests for the resource. Any inner resource request get the immediate access to the resources as it has the group lock. When a *s-outermost* request gets blocked on any resource group then the task busy-waits. If a *l-outermost* request blocks then the current task holding this resource group will get the highest priority among all tasks blocks on that group.

In FMLP resource release method, when a task releases a resource and it is *s-outermost*, then non-preemptive execution is turned off and the first task from the *busy-wait* FIFO queue gets the group lock of this group and starts execution. If the release request is *l-outermost* then revert back the priority of the task, it inherited from the highest priority task blocked in this group and the first task from the FIFO queue, blocked on this group, gets the resource and starts executing.

9.8 Clusters and Group of Task

Clustering is a hybrid approach of scheduling on multiprocessors. For each cluster, a specified scheduling algorithm is used to schedule the group of tasks assigned to that cluster. A cluster may contain an arbitrary number of cores. A single-core scheduling algorithm may be used where the cluster only contains one core. On the other hand, a multi-core scheduling algorithm is needed where the number of cores is more than one in a cluster.

As in case of implementation of the partitioned approach, the tasks are grouped and assigned to a cluster which requires manual task partitioning. The task partitioning must be done by the user through the system model. The tasks and resources are linked to a cluster by their names, ending with a dot followed by an integer number that specifies the cluster to which they are linked. The scheduler name and resource access protocol must be provided for each cluster.

Thus, the cluster scheduler runs the respective scheduler for each cluster with the specified resource access protocol. As P-EDF scheduler, the cluster scheduler also does this in its `initialize()` method and keeps track of the scheduler for each cluster in a `clusterScheduler` attribute. If a dispatch is necessary then the cluster scheduler calls the `dispatch()` method of the respective scheduler associated with the cluster that requires the dispatch.

The system model is validated by the `checkSystemModel()` function that determines whether the system model is properly prepared for the cluster scheduler or not, i.e. it checks the task names, resource names, scheduler for the cluster, etc.

10 | Validation, Test and Results

10.1 Validation and Test

In software engineering, *verification* is the process of checking a piece of software to confirm that it meets its specification and that the developed result includes all the functionality specified by the customer, *verification* refers to the confirmation that all functional and non-functional requirements are correctly implemented. Software testing is the procedure to show that the software does its specified work and to find the errors before it is used [53]. A testing process has two goals: (i) *validation testing* that determines whether the software satisfies the specified requirements and (ii) *defect testing* where the test cases are designed and checked to find out about the defects. Thus, software testing is part of a wider process - software verification and validation [53]. The eventual aim of the software testing is to make sure with confidence that the software is ready for the intended purpose.

Sommerville [53] mentioned in his book that software testing goes through three testing phases *development testing*, *release testing* and *user testing*. This section presents the testing procedures applied to test the developed software.

10.1.1 Development Testing

Development testing is a testing phase where the testing takes place during the development of the software to find the bugs and errors. System developers and programmers are responsible to carry out the testing in this phase [53].

Development testing includes all testing activities that are carried out by the team developing the system. The tester of the software is usually the programmer who developed the software. Development testing may consist of three phases: *unit testing*, *component testing* and *system testing* [53].

10.1.1.1 Unit Testing

Unit testing is the process of testing program components or units, i.e methods, objects and their individual functionality. Thus, the simplest types of program components are distinct methods or functions. Unit testing is carried out by testing of these methods individually by different sets of input parameters and comparing the results to the expected values [53].

10.1.1.2 Component Testing

Software components are usually composite components which consist of several individual interacting objects or functions. Thus, component testing tests the functionality provided by the software component where several individual software units or methods are integrated to provide the specified functionality [53].

During the development phase of the current version of the simulation software, each specific functionality has been tested as part of development testing. The individual functionality has been tested carefully. Among the tested are some important functionality tests that are going to be listed here in this section.

The implementation of self-suspension requires a function `createTerminateSelfSuspensionJobTask`, provided in the `TaskMonitor` class which is responsible to create the `TerminateSelfSuspensionJob` which is called from the `execute()` function of `InitiateSelfSuspensionJob`. It has been tested that it provides the correct functionality, i.e. creates the `Job` at correct time on the `TimeAxis`.

There are several functions provided for the schedulability analysis, each of them has been tested by loading system model and checking whether it is schedulable or not. The result was checked according to the expectations. If the result was not correct then the errors were located and fixed accordingly.

The sub-string comparison algorithms described in algorithm 1 on page 63 and algorithm 2 on page 63 have been implemented as a part of PF contending task comparison functionality. These two functions were tested with the respective inputs, i.e. two tasks from the example given by Baruah et al. [12] at time t and the output was compared. The attribute calculation for comparing two contending tasks according to algorithm PD was tested by manually calculating the result and checking whether the task comparison is correct or not.

The GSN-EDF algorithm has three sections for the implementation: a task released or resumed, a task finished its non-preemptive section and a task terminated or blocked. All these three sections have been tested independently to see whether the functionality of the algorithm was implemented successfully.

As a part of the FMLP implementation, a task may be required to be non-preemptive for a short time duration which depends on the accessing resource type. If the task is going to request a resource which is *s-outermost* then at first it should update the execution type from preemptive execution to non-preemptive execution. This functionality has been tested by creating a scenario.

10.1.1.3 System Testing

During development, system testing is done by integrating the software components. A version of the software is created by integrating the software components and then tests of the integrated system are done as part of development testing which is termed system testing. The responsibilities of system testing are to check that all components are integrated correctly, they are compatible and correctly interact with each other, and also transfer correct data at the right time [53].

The incremental software development follows a short development cycle to complete some functionality, testing and deploy it to the customer which was followed as a software engineering approach for this project. The incremental software development is managed by Scrum, a project management approach for software development. Each increment is managed by a sprint. Thus, during the sprint, developed functionality is integrated, tested and presented to the customer, i.e. the supervisor.

The test cases are written in such a way where the steps, inputs, preconditions, post-conditions and expected results are listed for each test case. During the test the test document is followed and the actual output is listed as actual result. When the testing has been completed, the expected outputs and actual outputs are compared. If the actual output is different from expected output then the source code is analyzed to find the bug and fix it.

The system testing has been done for the schedulability analysis, self-suspension functionality, algorithm PF, algorithm PD, GSN-EDF algorithm and FMLP implementation individually by following the examples presented in the respective literature as well as a lot of self created examples.

10.2 Results

This section discusses the results of the simulations that have been implemented as part of this thesis work. The results are going to be presented graphically, as generated by the software with a short description for each.

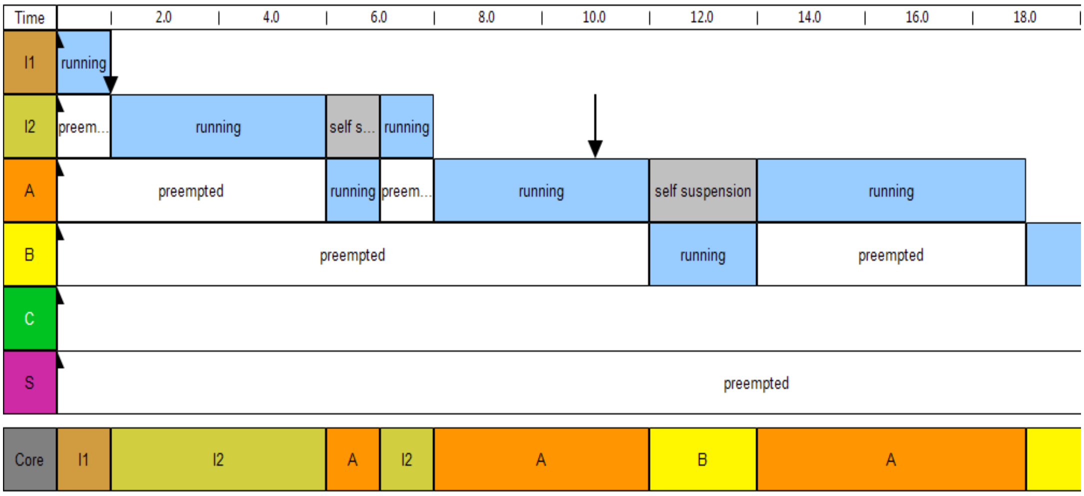


Figure 10.1: The simulation result of a task set scheduled under RMS where some tasks contain self-suspension delays.

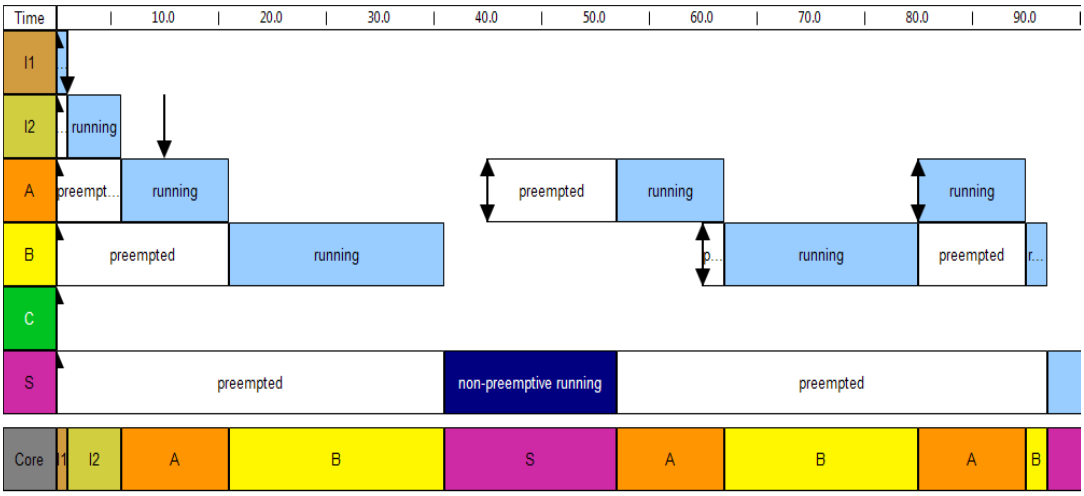


Figure 10.2: The simulation result of a task set scheduled under EDF where a task contains a non-preemptive section.

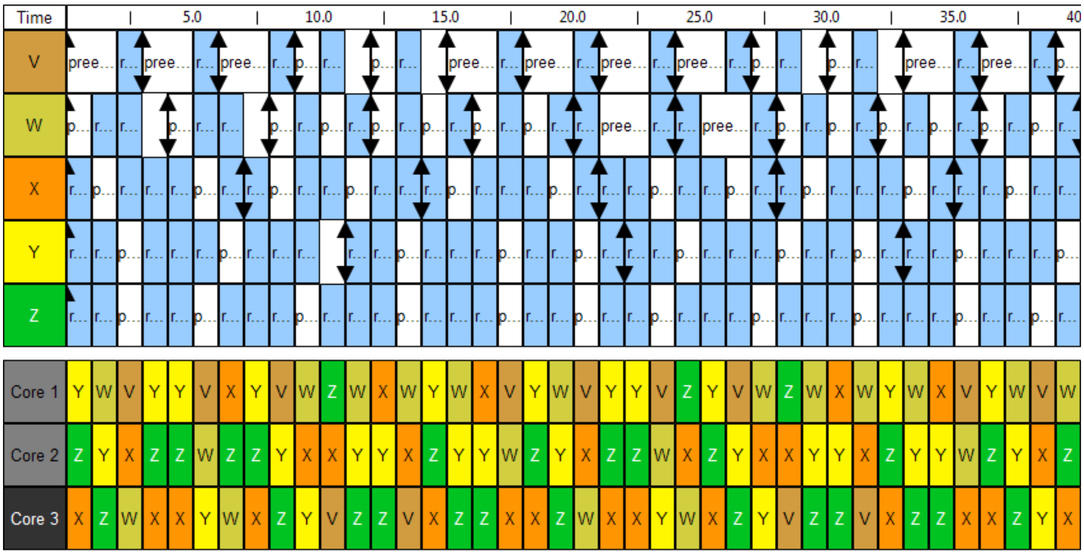


Figure 10.3: An example task set taken from Baruah et al. [12] is scheduled using algorithm PF. The task set only contains execution without using any resource.

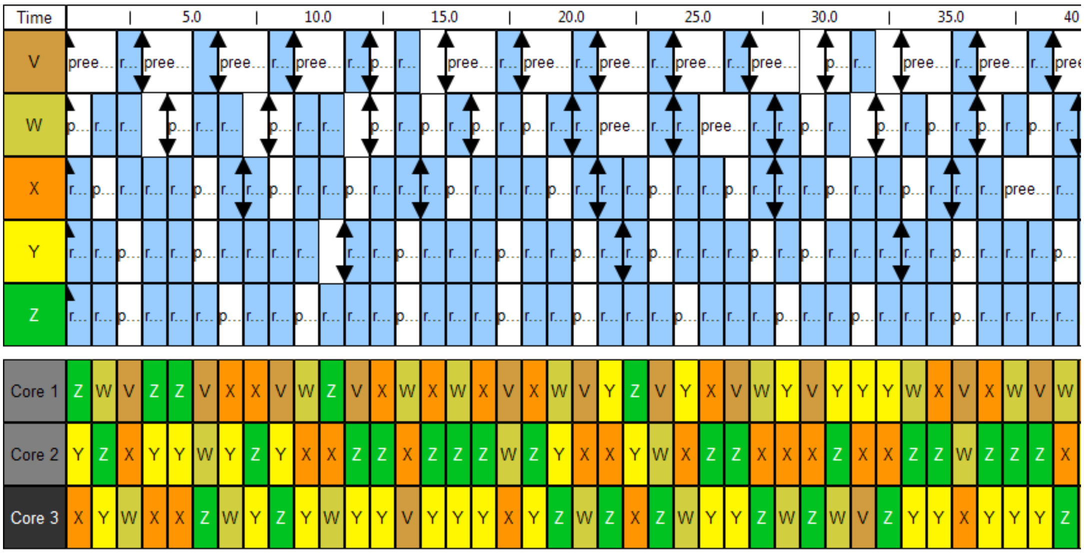


Figure 10.4: The same task set scheduled under PD, which has been scheduled under PF in the previous simulation, figure 10.3.

The following two figures show the simulation result of a task set scheduled under Pfair scheduling algorithm PF where some tasks in the task set get blocked to access shared resources and thus violate the Pfair lag constraint.

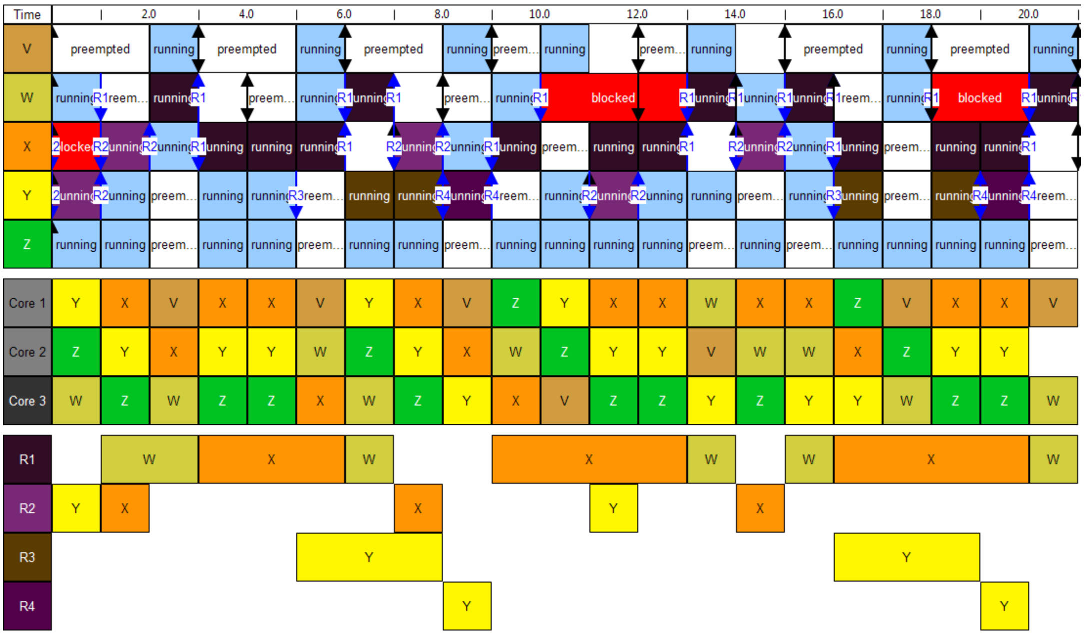
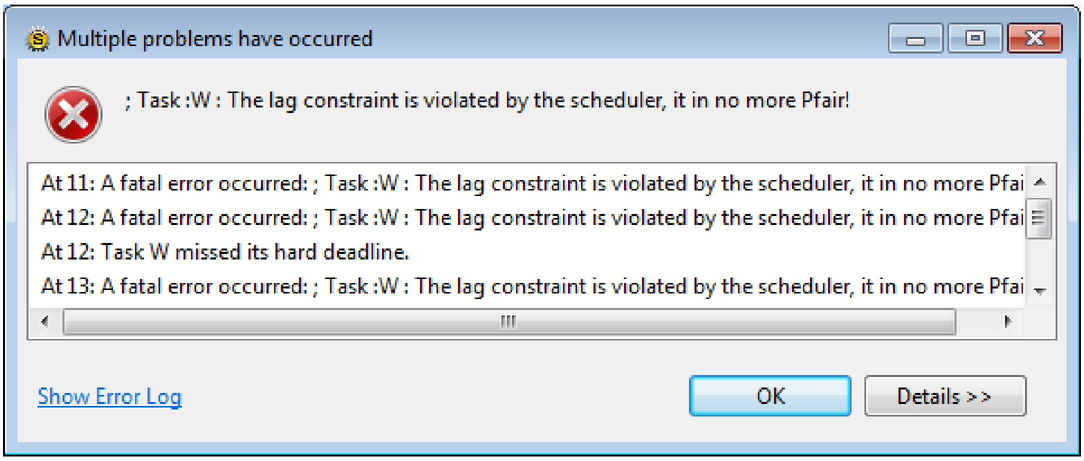


Figure 10.5: The simulation result of a task set scheduled under PF with shared resources.



; Task :W : The lag constraint is violated by the scheduler, it in no more Pfair!

```
At 11: A fatal error occurred: ; Task :W : The lag constraint is violated by the scheduler, it in no more Pfai
At 12: A fatal error occurred: ; Task :W : The lag constraint is violated by the scheduler, it in no more Pfai
At 12: Task W missed its hard deadline.
At 13: A fatal error occurred: ; Task :W : The lag constraint is violated by the scheduler, it in no more Pfai
```

Figure 10.6: The notification of the violation of Pfair lag constraint due to task being blocked to access shared resources.

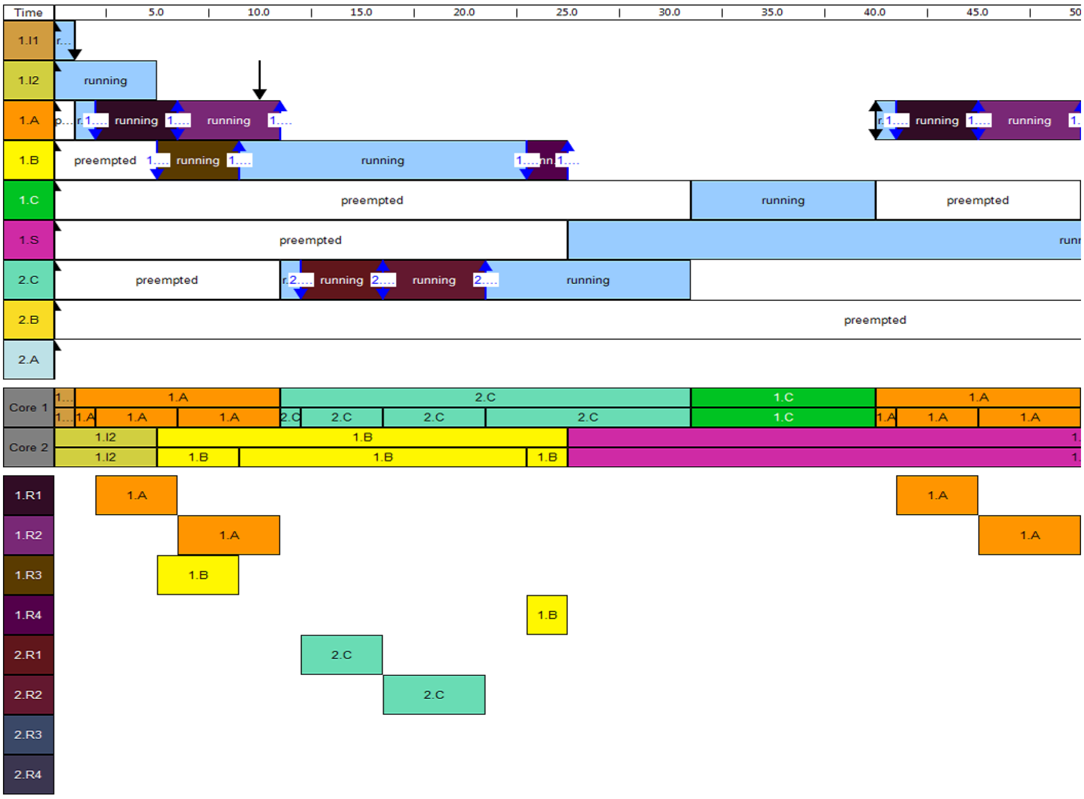


Figure 10.7: The simulation result of a task set scheduled under GSN-EDF with the default resource access protocol.

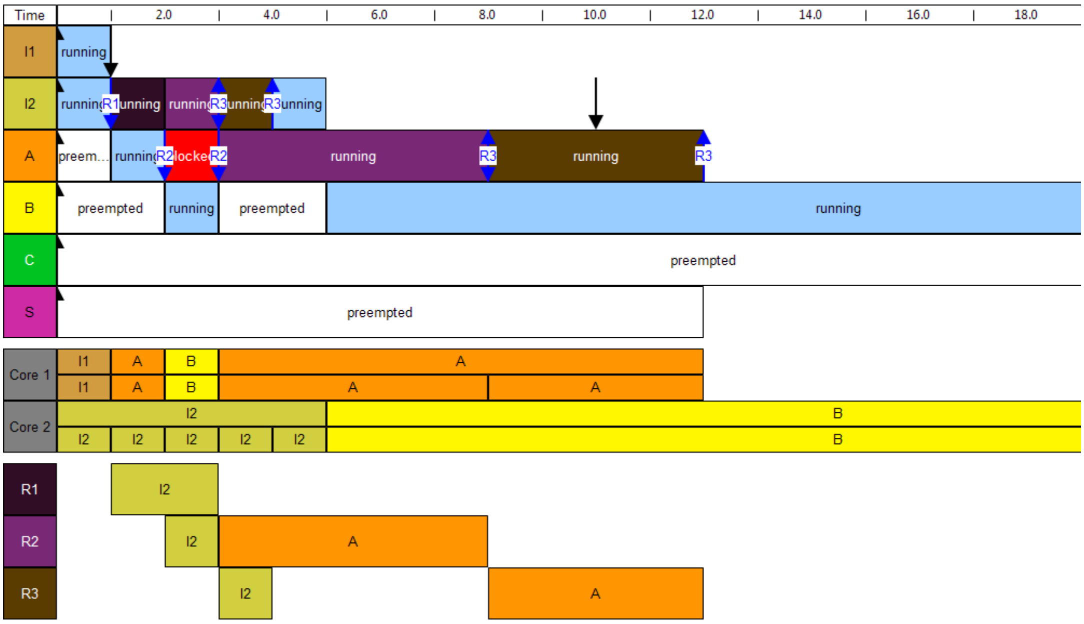


Figure 10.8: The simulation result of a task set scheduled under GSN-EDF with FMLP where a task gets blocked while accessing a shared resource that is a long resource. The resource is free but the group to which this resource belongs to is locked by another task, thus, the requested task is blocked.

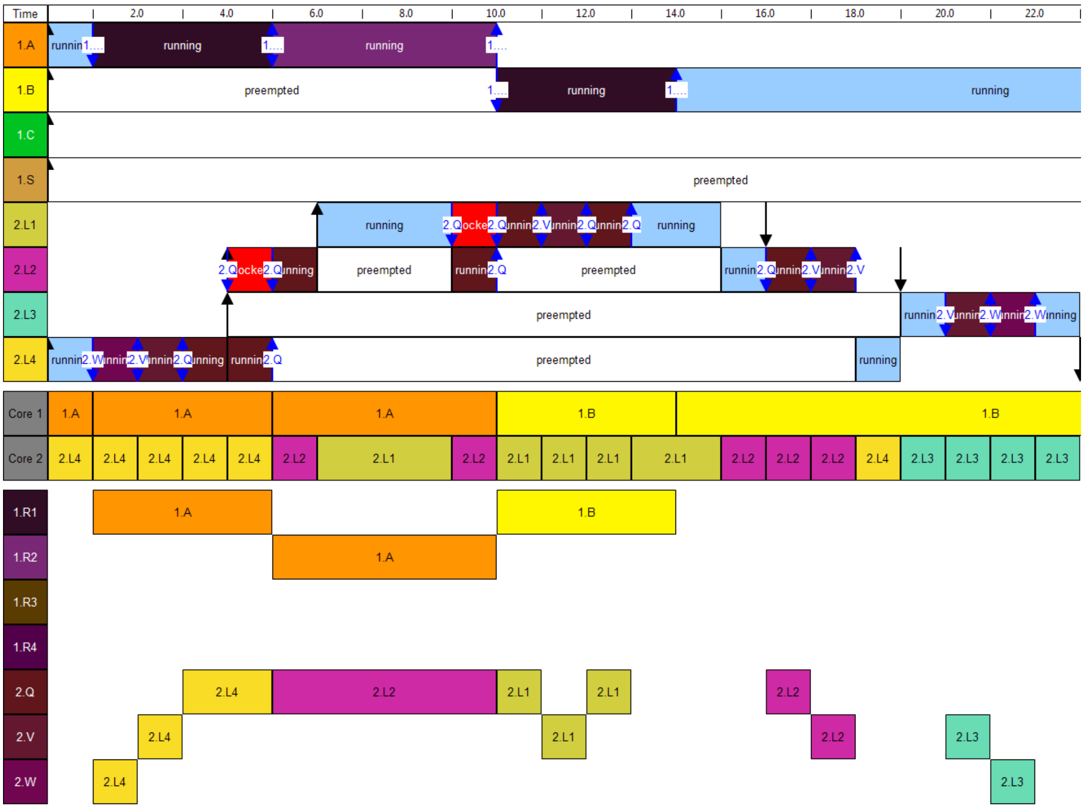


Figure 10.9: The simulation result of a task set scheduled under P-EDF with the default resource access protocol.

11 | Conclusion and Future Work

11.1 Conclusion

In this thesis work, available literature related to scheduling on multiprocessors and multiprocessor scheduling policies for real-time embedded systems have been studied in detail. Furthermore, the concepts related to scheduling on multiprocessors and some promising multiprocessor scheduling policies have been presented with their implementation on the existing system which can be considered the main part of this thesis work.

In the the previous work [41], a simulation software for task scheduling and visualization was developed which focuses on but is not limited to scheduling policies primarily used in real-time embedded systems. The existing software does not provide schedulability analysis and self-suspension of tasks which is a part of this thesis work. Additionally, the newly implemented multiprocessor scheduling policies have been integrated into the existing software. Furthermore, an extensive effort has been given to understanding the existing software in a comprehensive manner. The necessary parts of existing software used by the implementation part of this thesis work have been discussed in brief to give an outline of the base simulation kernel.

As mentioned earlier, the existing simulation software supports some well known single core and a few multi-core scheduling algorithms but the schedulability analysis is absent. The necessary theory and its implementation on schedulability analysis for single core scheduling policies has been discussed in detail both for fixed priority and dynamic priority scheduling algorithms.

In multi-core scheduling, a task may self-suspend itself when it accesses a resource and knows the time duration required by the resource to respond, in such scenario it self-suspends itself by the predefined duration required by the resource to respond. The existing software does not support the feature of task self-suspension which has been implemented in this thesis work. The concept of self-suspension is simple but it creates some scheduling anomalies by a very low utilization of the processor that happen in the dynamic priority scheduling policies [39]. Furthermore, the schedulability analysis is more difficult than task self-suspension itself. Therefore, the system does not provide schedulability analysis where there is a task containing self suspension in the task set.

There are two main approaches for scheduling on multiprocessors that have been discussed in detail with their pros and cons with respect to other mechanisms. Another approach has been discussed that combines the above two approaches termed hybrid approach. A noticeable term for multiprocessor scheduling algorithms describes whether an algorithm is work conserving or not. The scheduling algorithms in the class of partitioned approach are not work-conserving whereas some global approaches for scheduling algorithms are work-conserving.

The partitioned EDF has been discussed in detail with some heuristic methods to partition the task set into m subsets where the number of processors is m. The partitioning approaches are known as bin-packing problems which is NP-Hard in the strong sense. They cannot guarantee more than about 50% use of the processor. The implementation does not provide any automatic task partitioning method which should be done by the user manually, so that for each processor a single-core EDF algorithm runs to schedule the task set.

The Pfair scheduling approach introduced by Baruah et al. [12] is currently the only known optimal scheduling approach for scheduling on multiprocessors. The optimality of the Pfair approach is only restricted to periodic task sets with implicit deadlines. The Pfair variants of multiprocessor scheduling algorithms PF and PD have been discussed in detail with their implementation. They both use the same method to prioritize tasks based on the task deadline; the only difference is the way they break the tie where two jobs of different tasks have the same deadline. The implementation has been done in such a way, the common functionality is placed inside an abstract class `Pfair` which is extended by the variants of Pfair scheduling algorithms and implement the differing parts in their subclasses. The Pfair variants of scheduling algorithms can use resources by default resource access protocol but there may arise some scheduling anomalies while accessing a shared resource, e.g. a task is blocked on a resource and cannot execute for some subsequent slots and thus violates its lag constraint.

Like single-core scheduling algorithms, a multi-core scheduling algorithm must have a resource access protocol where there are some resources shared by multiple tasks at any time t. In order to synchronize the resource usage by the tasks scheduled on a multiprocessor system, a multiprocessor resource access protocol must be used with the scheduling algorithms. The flexible multiprocessor locking protocol has been discussed in detail including its implementation with GSN-EDF, a variant of global EDF. Block et al. [14] used the term “flexible” to mean that it can be used for both partitioned and global approaches. The FMLP requires a version of Global-EDF namely GSN-EDF which has been implemented as a prerequisite to implement it. The GSN-EDF implementation includes the enhancement of the existing simulator so that a task can execute for a specified duration non-preemptively, which can be used by any other scheduling algorithms by specifying the non-preemptive section in the system model. A multiprocessor system model can be scheduled under GSN-EDF with FMLP.

As a part of visualization, some small changes have been introduced that are required by the new implementation. The scheduling view has been updated for GSN-EDF algorithm to display which tasks are linked to which cores in addition displaying the task scheduled on each core.

The system has been tested using the specified testing approach. As a part of development testing unit testing, component testing and system testing have been done during the development. Although release testing requires a separate team to test the entire system, there is no available separate team for that. Therefore, the release testing has also been done by the developer.

11.2 Future Work

The present version of the simulation software does not support the schedulability analysis for EDF, the dynamic priority single core scheduling algorithm, by considering the synchronization between tasks for shared resource access. Neither does it support the schedulability analysis where a task in the task set contains self-suspension delays or non-preemptive sections for execution. The simulator provides multi-core scheduling simulation algorithms both from partitioned and global approaches of multiprocessor scheduling policies. It only provides the schedulability analysis for the partitioned multi-core scheduling algorithms but not for the class of global multi-core scheduling algorithms. Therefore, the schedulability analysis can be improved for the cases where the present version of the software does not provide a solution.

Of the partitioned approaches of multi-core scheduling policies P-DMS was implemented in the previous work [41] and P-EDF has been implemented in this thesis work where the task partitioning is manual, done by the user through the provided system model. A literature study can be employed to search for the promising partitioned multi-core scheduling algorithms and the automatic task partitioning techniques to implement in the simulator.

In this thesis work, the Pfair variants of dynamic priority scheduling algorithms PF [12] and PD [11] have been implemented which provide a framework that can be used to implement other variants of Pfair scheduling approaches. Therefore, other promising Pfair variants of scheduling algorithms can be studied from the available literature and implemented in order to enhance the current version of the software. The variant of Pfair scheduling algorithm known as ERfair is a work-conserving scheduling algorithm which can be considered for the implementation [1, 2].

Once the system model has been loaded into the simulator, the elements of the system model, e.g. a task and its attributes, a resource and its attributes etc., cannot be modified. Therefore a graphical user interface can be provided to view or modify the elements of the system model.

The input of the simulator is file based. Thus, an XML file is prepared and provided as a system model to the simulator. Therefore, all the elements in the system model need to be provided explicitly. The automatic task generation functionality can be provided for the simulator so that an arbitrary number of tasks, resources and cores can be generated from the necessary configuration for the simulation. The necessary configuration for the task, resource and core generation can be provided by the user explicitly before the system model generation.

The performance evaluation is still missing in the updated version of the simulator, which may be an attractive feature that will compare the performance of the different scheduling algorithms. A task set can be scheduled under two different scheduling algorithms and then the performance can be evaluated according to some predefined metric displaying the results to the user, it can be either in a text or graphical format.

Bibliography

[1] James H Anderson and Anand Srinivasan. Early-release fair scheduling. In *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*, pages 35–43. IEEE, 2000.

[2] James H Anderson and Anand Srinivasan. Mixed pfair/erfair scheduling of asynchronous periodic tasks. In *Real-Time Systems, 13th Euromicro Conference on, 2001.*, pages 76–85. IEEE, 2001.

[3] Björn Andersson, Sanjoy Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. In *Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE*, pages 193–202. IEEE, 2001.

[4] Neil C Audsley, Alan Burns, Mike F Richardson, and Andy J Wellings. Real-time scheduling: the deadline-monotonic approach. In *in Proc. IEEE Workshop on Real-Time Operating Systems and Software*. Citeseer, 1991.

[5] Neil C Audsley, Alan Burns, Robert I Davis, Ken W Tindell, and Andy J Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems*, 8(2-3):173–198, 1995.

[6] Theodore P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, 1991.

[7] Theodore P Baker. A comparison of global and partitioned edf schedulability tests for multiprocessors. In *In International Conf. on Real-Time and Network Systems*. Citeseer, 2005.

[8] M. Barr. Embedded systems glossary, 2007. URL http://www.barrgroup.com/Embedded-Systems/Glossary.

[9] Sanjoy K Baruah, Aloysius K Mok, and Louis E Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 182–190. IEEE, 1990.

[10] Sanjoy K Baruah, Louis E Rosier, and Rodney R Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2(4):301–324, 1990.

[11] Sanjoy K Baruah, Johannes E Gehrke, and C Greg Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Parallel Processing Symposium, International*, pages 280–280. IEEE Computer Society, 1995.

[12] Sanjoy K Baruah, Neil K Cohen, C Greg Plaxton, and Donald A Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[13] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *Parallel and Distributed Systems, IEEE Transactions on*, 20(4):553–566, 2009.

[14] Aaron Block, Hennadiy Leontyev, Björn B Brandenburg, and James H Anderson. A flexible real-time locking protocol for multiprocessors. In *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, pages 47–56. IEEE, 2007.

[15] Björn B Brandenburg and James H Anderson. Optimality results for multiprocessor real-time locking. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 49–60. IEEE, 2010.

[16] Björn B Brandenburg, Hennadiy Leontyev, and James H Anderson. An overview of interrupt accounting techniques for multiprocessor real-time systems. *Journal of Systems Architecture*, 57(6):638–654, 2011.

[17] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages.* Addison Wesley, MA, 4th edition, 2009.

[18] G. Buttazzo. *Hard Real-Time Computing Systems Predictable Scheduling Algorithms and Applications.* Springer, Berlin, Germany., 2nd edition, 2005.

[19] John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James Anderson, and Sanjoy Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. *Handbook on Scheduling Algorithms, Methods, and Models, pages*, pages 30–1, 2004.

[20] Chia-Mei Chen, Satish K Tripathi, and Alex Blackmore. A resource synchronization protocol for multiprocessor real-time systems. In *Parallel Processing, 1994. Vol. 1. ICPP 1994. International Conference on*, volume 3, pages 159–162. IEEE, 1994.

[21] Sadegh Davari and Sudarshan K Dhall. An on line algorithm for real-time tasks allocation. In *RTSS*, pages 194–200, 1986.

[22] Robert I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):1–44, October 2011.

[23] Robert I Davis, Attila Zabos, and Alan Burns. Efficient exact schedulability tests for fixed priority real-time systems. *Computers, IEEE Transactions on*, 57(9):1261–1276, 2008.

[24] ML Dertouzos. Control robotics: the procedural control of physical processes," information processing 74, 1974.

[25] Umamaheswari C Devi. *Soft real-time scheduling on multiprocessors.* PhD thesis, University of North Carolina, 2006.

[26] UmaMaheswari C Devi, Hennadiy Leontyev, and James H Anderson. Efficient synchronization under global edf scheduling on multiprocessors. In *Real-Time Systems, 2006. 18th Euromicro Conference on*, pages 10–pp. IEEE, 2006.

[27] Sudarshan K Dhall and CL Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[28] Colin J. Fidge. Real-time schedulability tests for preemptive multitasking. *Real-Time Systems*, 14(1):61–93, 1998.

[29] Paolo Gai, Giuseppe Lipari, and Marco Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE*, pages 73–83. IEEE, 2001.

[30] Mike Holenderski. Real-time system overheads: a literature overview. *Computer Science Report*, 8:26, 2008.

[31] Philp L. Holman. *On the Implementation of Pfair-scheduled Multiprocessor Systems.* Phd thesis, Chapel Hill, 2004.

[32] Kevin Jeffay and Donald Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *Real-Time Systems Symposium, 1993., Proceedings.*, pages 212–221. IEEE, 1993.

[33] J. T. Kao and A. P. Chandrakasan. Dual-threshold voltage techniques for low-power digital circuits. *IEEE Journal of Solid-State Circuits*, 35:1009–1018, July 2000.

[34] Karthik Lakshmanan and Ragunathan Rajkumar. Scheduling self-suspending real-time tasks with rate-monotonic priorities. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, pages 3–12. IEEE, 2010.

[35] John P Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *RTSS*, volume 90, pages 201–209, 1990.

[36] Joseph Y-T Leung and ML Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information processing letters*, 11(3):115–118, 1980.

[37] C. L. Liu. Scheduling algorithms for multiprocessors in a hard real-time environment. *JPL Space Programs Summary*, 37–60:28–31, 1969.

[38] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20 (1):46–61, 1973.

[39] Cong Liu and James H Anderson. Task scheduling with self-suspensions in soft real-time multiprocessor systems. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 425–436. IEEE, 2009.

[40] J. W. S. Liu. *Real Time Systems.* Prentice Hall, Englewood Cliffs, NJ., 2000.

[41] Peter Munk. Visualization of scheduling in real-time embedded systems. Master thesis, University of Stuttgart, 2013.

[42] Geoffrey Nelissen, Dragomir Milojevic, and Joël Goossens. *Efficient Optimal Multiprocessor Scheduling Algorithms for Real-Time Systems.* PhD thesis, PhD thesis, Université Libre de Bruxelles, 2013.

[43] Farhang Nemati. *Partitioned Scheduling of Real-Time Tasks on Multi-core Platforms.* School of Innovation, Design and Engineering, Mälardalen University, 2010.

[44] Nimal Nissanke. *Realtime systems.* Prentice-Hall, Inc., 1997.

[45] Ragunathan Rajkumar. *Synchronization in real-time systems: A Priority Inheritance Approach.* Kluwer Academic Publishers, 1991.

[46] Ragunathan Rajkumar, Lui Sha, and John P Lehoczky. Real-time synchronization protocols for multiprocessors. In *RTSS*, pages 259–269, 1988.

[47] Frederic Ridouard, Pascal Richard, and Francis Cottet. Negative results for scheduling independent hard real-time tasks with self-suspensions. In *Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International*, pages 47–56. IEEE, 2004.

[48] Frédéric Ridouard, Pascal Richard, Francis Cottet, and Karim Traore. Some results on scheduling tasks with self-suspensions. *Journal of Embedded Computing*, 2(3):301–312, 2006.

[49] Ismael Ripoll, Alfons Crespo, and Aloysius K Mok. Improvement in feasibility testing for real-time tasks. *Real-Time Systems*, 11(1):19–39, 1996.

[50] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand. Leakage current mechanisms and leakage reduction techniques in deep-submicrometer cmos circuits. *Proceedings of the IEEE*, 91(2):305–327, February 2003.

[51] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers*, 39(9):1175–1185, 1990.

[52] Robert E Shannon. Introduction to the art and science of simulation. In *Proceedings of the 30th conference on Winter simulation*, pages 7–14. IEEE Computer Society Press, 1998.

[53] Ian Sommerville. *Software Engineering.* Addison Wesley, 9 edition, 2011.

[54] A. Srinivasan, P. Holman, J. Anderson, and S. Baruah. The case for fair multiprocessor scheduling. *Proceedings of the 11th International Workshop on Parallel and Distributed Real-time Systems*, 2003.

[55] Anand Srinivasan. *Efficient and Flexible Fair Scheduling of Real-time Tasks on Multiprocessors.* Phd thesis, Chapel Hill, 2003.

[56] Ken Tindell and John Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and microprogramming*, 40(2):117–134, 1994.

[57] David A Wood and Mark D Hill. Cost-effective parallel computing. Technical report, DTIC Document, 1994.

[58] Jianjia Wu, Jyh-Charn Liu, and Wei Zhao. On schedulability bounds of static priority schedulers. In *Real Time and Embedded Technology and Applications Symposium, 2005. RTAS 2005. 11th IEEE*, pages 529–540. IEEE, 2005.

[59] K.-S. Yeo and K. Roy. *Low Voltage, Low Power VLSI Subsystems.* McGraw-Hill, Inc, New York, NY, USA, 1 edition, 2005.

List of Figures

List of Tables

Listings

Abbreviations

CSIs	Cycle-stealing Interrupts
DIs	Device Interrupts
DMS	Deadline Monotonic Scheduling
EDF	Earliest Deadline First
EMF	Eclipse Modeling Framework
FIFO	First In First Out
FMLP	Flexible Multiprocessor Locking Protocol
G-EDF	Global Earliest Deadline First
GPS	Global Positioning System
GSN-EDF	Global Suspendable Non-Preemptive EDF
IPIs	Inter-processor Interrupts
ISR	Interrupt Service Routine
MPCP	Multiprocessor Priority Ceiling Protocol
MRI	Magnetic Resonance Imaging
MSRP	Multiprocessor Stack Resource Protocol
NMIs	Non Maskable Interrupts
P-DMS	Partitioned Deadline Monotonic Scheduling
P-EDF	Partitioned Earliest Deadline First
Pfair	Proportionate Fairness
RCP	Rich Client Platform
RMS	Rate Monotonic Scheduler
SAVORS	Simulation And Visualization of Real-time Scheduling
TIs	Timer Interrupts
UML	Unified Modeling Language
WCET	Worst Case Execution Time