technische universität
dortmund

Master thesis

Real-Time Synchronization on Multiprocessors through Suspension or Spinning?

Shabnam Tabatabaian
September 2018

Supervisors:
Prof. Dr. Jian-Jia Chen
Dipl.-Inf. Georg von der Brüggen

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12 (Eingebettete Systeme)
http://ls12-www.cs.tu-dortmund.de

Acknowledgements

Foremost, I would like to thank my professor Prof. Chen, who has given me the opportunity to work on this amazing topic. I also would like to express my gratitude towards my supervisor Dipl.-Inf. Georg von der Brüggen for his continuous support, insightful comments and feedback throughout all the six months of my master thesis. Further I would like to thank all my friends and family, who helped me during my thesis.

Last but not least, I am particularly gratitude to my parents for their endless love and support from the very beginning of my life. I am also thankful of my wonderful sister, Shadan, who made my life colorful! A special thanks to my beloved, Milad, without whose support and encouragement this work would never have been possible.

Abstract

For many real-time applications, it is often too restrictive to assume that tasks are executed only in a uniprocessor system and all of them are completely independent, because tasks will often compete for shared resources. The access to the shared resource, called *critical section* of the task execution, is mutually exclusive to prevent *race conditions*. In fact, it is not possible that two jobs access the shared resource at the same time. As shared resources must be serially executed to achieve *mutual exclusion*, the execution of critical sections inevitably causes some delay due to *priority inversion*, which means, a task is prevented from executing due to another task with a lower priority that holds a shared resource, also called *pi-blocking*. To avoid unnecessary blocking of high-priority tasks due to unrelated shared resources, there exists some locking protocols. Two well-known of these locking protocols for systems with one processor, also called uniprocessor systems, are *Priority Inheritance Protocol (PIP)* [1] and the *Priority Ceiling Protocol (PCP)* [1]. Although the PIP could solve the problem of priority inversion, it might cause a deadlock if there are multiple resources. Through the use of PCP proposed by Sha et. al [1], deadlock can never occur.

Resource sharing becomes more complex when considering multiprocessor platforms instead of uniprocessor platforms, because in multiprocessor systems the *task-suspending* and *task-spinning* may come into play. Under spinning approach, tasks busy-wait in a tight loop or called "spin" to the processor, while it is waiting for accessing a shared resource. Under suspending approach, tasks relinquish their processor and suspend during the time waiting for a shared resource accessing. There exist some protocols, which support both mechanisms, e.g. *Multiprocessor Slack Resource Policy (MSRP)* [2] as a spin-based multiprocessor locking protocol as well as *Distributed Priority Ceiling Protocol (DPCP)* [3] and *Multiprocessor Priority Ceiling Protocol (MPCP)* [4] under suspending approach. Although, many suspension-based and spin-based locking protocols have been designed and analyzed in the last years, it has been relatively little investigated, that under which settings which one of the suspension-based and the spin-based locking protocols are in general preferable. Therefore, this tradeoff has to be analyzed and some break-even configurations should be determined. The only result in this area is published in [5], which concludes that "blocking by suspending is rarely preferable to spinning (provided spinning can be done in-cache, which we assume)". However, the analysis in [5] is conducted only considering the *Flexible Multiprocessor Locking Protocol (FMLP)*.

In this thesis other locking protocols than FMLP will be taken into consideration and finally the conditions that favor spin-based locking protocols over suspension-based locking protocols and the other way around, will be specified. To this end, first a test-bed based on the platform *LITMUS*^{RT} [6] [7] will be set up and then those protocols will be tested under different settings of properties such as number of processors, number of resource accesses per task, period of each task and etc. to find the break-even configurations.

Contents

Acknowledgements i
Abstract ii
List of Figures v
List of Tables vi
List of Abbreviations vii
List of Notations viii
1 Introduction 1
1.1. Task Scheduling 3
1.1.1. Fixed Priority Algorithm 7
1.1.2. Dynamic Priority Algorithm 8
1.2. Concurrent Resource Access 8
1.2.1. Lock-Free 9
1.2.2. Locking 10
1.3. Synchronization 12
1.4. Thesis Goal 14
1.5. Prior and Related Works 15
1.6. Thesis Structure 16
2 Multiprocessor Real-Time Systems 17
2.1. Real-Time Task Model 18
2.2. Real-Time Resource Model 19
2.3. Real-Time Scheduling 19
2.3.1. Partitioned Scheduling 20
2.3.2. Global Scheduling 23
2.4. Synchronization Mechanism 24
2.4.1. Spinning 25
2.4.2. Suspending 27
2.5. Synchronization Protocols 28
2.5.1. Distributed Priority Ceiling Protocol 28
2.5.2. Distributed FIFO Locking Protocol 30
2.5.3. Multiprocessor Priority Ceiling Protocol 32
2.5.4. Multiprocessor Priority Ceiling Protocol Virtual Spinning 34

3 Test-Bed 35
3.1. Hardware Platform 35
3.2. Basics of LITMUS^{RT} 35
3.2.1. Installation 37
3.2.2. Modification 37
3.2.3. Task States 40
3.2.4. Verifications 41
3.2.5. Measuring of Test Parameters 45
4 Evaluation 46
4.1. Experimental Set-Up 46
4.1.1. Scheduling Algorithm 46
4.1.2. Partition Algorithm 47
4.1.3. Resource Assignment Rule 48
4.1.4. Task-Set Configuration Parameters 48
4.2. Results Comparison and Analyzing 49
4.2.1. Schedulability Evaluation 50
4.2.2. Maximum Response Time Evaluation 54
4.2.3. Average Response Time Evaluation 59
5 Summary 63
5.1. Conclusion 63
5.2. Future Work 64
References 65
Appendix: List of Graphs 71

List of Figures

Figure 1.1: The basic structure of multiprocessing systems 2
Figure 1.2: Global scheduling 4
Figure 1.3: Partitioned scheduling 5
Figure 1.4: Semi-Partitioned scheduling 6
Figure 1.5: Cluster-based scheduling 7
Figure 1.6: Priority Inversion 9
Figure 1.7: Deadlock 11
Figure 1.8: Global resource queues (τis are the tasks, which are waiting for a global resource) 12
Figure 1.9: Priority boosting 14
Figure 2.1: Typical parameters of a real-time task 18
Figure 2.2: Illustration of the first-fit, best-fit and worst-fit decreasing bin-packing heuristics .. 23
Figure 2.3: The difference between pi-blocking and s-blocking 26
Figure 2.4: Example for DPCP (redrawn from [34]) 30
Figure 2.5: Example for DFLP (redrawn from [57]) 32
Figure 2.6: Example for MPCP (redrawn from [34]) 33
Figure 2.7: Example for MPCP-VS (redrawn from [34]) 34
Figure 3.1: Task-state transition diagram 40
Figure 3.2: DPCP's behavior implemented on LITMUS^{RT} 41
Figure 3.3: DFLP's behavior implemented on LITMUS^{RT} 42
Figure 3.4: MPCP's behavior implemented on LITMUS^{RT} 43
Figure 3.5: MPCP-VS's behavior implemented on LITMUS^{RT} 44
Figure 4.1: Example of RM scheduling 47
Figure 4.2: Schedulability results obtained by executing of tasks with low utilization 52
Figure 4.3: Schedulability results obtained by executing of tasks with medium utilization 54
Figure 4.4: Maximum response time results obtained by executing of tasks with low utilization 55
Figure 4.5: Maximum response time results obtained by executing of tasks with medium utilization 57
Figure 4.6: Maximum response time results obtained by executing of tasks with the ratio of the total cs length to the total non-cs length per task-set equal to 10% 58
Figure 4.7: Average response time results obtained by executing of tasks with low utilization . 59
Figure 4.8: Average response time results obtained by executing of tasks with medium utilization 61
Figure 4.9: Average response time results obtained by executing of tasks with the ratio of the total cs length to the total non-cs length per task-set equal to 10% 62

List of Tables

Table 2.1: Sample of a task-set related to the examples showed in Figure 2.4, Figure 2.6 and Figure 2.7 30
Table 2.2: Sample of a task-set related to the example showed in Figure 2.5..... 32
Table 3.1: Definition of parameters of the customized tasks allocation tool, rtspin 39
Table 4.1: Example of implicit-deadline periodic task-set..... 47
Table 4.2: Experimental set-up 50

List of Abbreviations

CPU	Central Processing Unit
cs	critical section
DFLP	Distributed Priority Ceiling Protocol
DM	Deadline Monotonic
DPCP	Distributed Priority Ceiling Protocol
EDF	Earliest Deadline First
FMLP	Flexible Multiprocessor Locking Protocol
G-EDF	Global Earliest Deadline First
G-FP	Global Fixed Priority
LITMUS^{RT}	LInux Testbed for MUltiprocessor Scheduling in Real-Time systems
LST	Least Slack Time
MPCP	Multiprocessor Priority Ceiling Protocol
MPCP-VS	Multiprocessor Priority Ceiling Protocol Virtual Spinning
MSRP	Multiprocessor Slack Resource Policy
PCP	Priority Ceiling Protocol
Pfair	proportionate fair
P-FP	Partitioned Fixed Priority
pi-blocking	priority inversion blocking
PIP	Priority Inheritance Protocol
RM	Rate Monotonic
RPC	Remote Procedure Calls
RTOS	Real Time Operating System
s-blocking	Spin blocking
WCET	Worst-Case Execution Time

List of Notations

Notation	Interpretation	Constraint/Definition
T	A task-set	$T = \{\tau_1, \dots, \tau_N\}$
N	Number of tasks per task-set	$N = \left\lceil \frac{50}{m} \right\rceil . m$
τ_i	The i^{th} task	$i \in \{1, \dots, N\}$
J_{x_i}	The x^{th} job of task τ_i	$x_i \geq 1$
C_i	Worst-case execution time of task τ_i	$C_i > 0$
p_i	Period of task τ_i	$p_i \geq C_i$
D_i	Relative deadline of task τ_i	$D_i \geq C_i$
d_i	Absolute deadline of task τ_i	
a_i	Arrival or release time of task τ_i	
f_i	Finishing time of task τ_i	
s_i	Start time of task τ_i	
R_i	Response time of task τ_i	$R_i = f_i - a_i$
U_i	Utilization of task τ_i	$U_i = \frac{C_i}{p_i}$
U_i^C	Total utilization of critical sections of task τ_i	
U_i^{nonC}	Total utilization of non-critical sections of task τ_i	
$Y(\tau_i)$	Function of base priority assigning to tasks τ_i	$Y(\tau_i) = i$
$y(\tau_i, t)$	Function of effective priority assigning to tasks τ_i at time t	
m	Number of processors	$m \in \{2,4,8,16\}$
P_j	The j^{th} processor	$j \in \{1, \dots, m\}$
S_j	Set of assigned tasks (items) to processor P_j (j^{th} bin)	$j \in \{1, \dots, m\}$
K	Number of resource accesses per task (Number of critical sections per task)	$K \in \{1, 2, 3, 4, 5\}$
n	Number of shared resources in a task-set	$n = \left\lceil \frac{K \,.\, 50}{m} \right\rceil$
r_q	The q^{th} resource	$q \in \{1, \dots, n\}$
$R_{i,q,v}$	The v^{th} time that the task τ_i requests the resource r_q	$v \geq 1, 1 \leq i \leq N, 1 \leq q \leq n$
A_i	Agent of task τ_i	
$A_{j,i}$	Agent of task τ_i related to the resource located on processor P_j	
F_q	FIFO-Queue of the resource r_q	
B	Number of Bins	
V	Bin capacity	
x_i	The i^{th} item	
a_i	Size of item x_i	
V_j	Remain capacity of j^{th} bin (processor P_j)	$V_j = V - \sum_{x_i \in S_j} a_i$

1 Introduction

Real-time systems are computing systems for which the reaction to events within a limited time is very important. In other words, if a correct result is not delivered within precise time constraints, deemed deadline, they might be useless or even dangerous. Therefore, the name “real-time” is opted for such systems. As a consequence, these systems must support dual notations of correctness in order to result a correct behavior, which are *logical correctness*, i.e., the results are correct, and *temporal correctness*, i.e., the results are delivered on time. Indeed, high reactivity and high dependability are more important than performance in the real-time environment [8].
With the emerge of the real-time systems, they have started to be used widely in the industry in recent years. A *Real Time Operating System (RTOS)* is an operating system, which is developed for real-time embedded systems and applications. One of the most important features of RTOS is the timing considerations so that the most important operations are assigned higher priorities, therefore they will be under investigation sooner than other operations. With the help of RTOS, priorities can be changed directly and the real-time system can process data fast, so that the attained results are usable for responding to another process, which is happened at the same time, as in transaction processing [9]. RTOS is able to response rapidly enough to external events in a predetermined and predictable manner. Beside the immediately respond feature of RTOSs, the basic supports for scheduling, resource management and I/O can be provided through RTOSs. Moreover, single-use specialized systems have been changed through RTOSs to the operating systems with a wide variety of general-purpose (such as real-time variants of Linux). Communication and synchronization between different tasks are two other most important things that RTOS focuses on to gain the target of the application [9].
In the following, the basic issue of the structure of a real-time kernel will be described shortly. The kernel of an operating system is the most internal part of the system that is connected to the hardware of the physical machine directly. The basic activities of a kernel are *Process Management*, *Interrupt Management* and *Process Synchronization* [8].

- **Process management** is the main service, which should be provided by any operating system. It supports different activities, for instance process creation and termination, task scheduling, context switching, and other relevant functions.
- **Interrupt management** is a mechanism for handling the interrupt requests generated by any peripheral device, such as the mouse, the keyboard, ports, or any specific sensor interface.
- **Process synchronization** is another important service of the kernel, which means that the kernel should provide a mechanism for supporting synchronization and communication. Semaphores in classical operating systems were responsible to do this support, but they usually encounter the problem of priority inversion (see section 1.2.1), which causes unbounded blocking on tasks’ execution and makes system unreliable for hard real-time

tasks. In this regard, the kernel of a real-time system should provide proper types of semaphores, which can provide predictability by supporting resource access protocols (such as Priority Inheritance [1] or Priority Ceiling [1]) to manage priority inversion.

In the end of the last century, the uniprocessor systems could not support the raising request of computer capacity any longer. Hence, the multiprocessor concept was suggested and developed by several companies. Nowadays, such multiprocessor chips are used in many real-time systems and afterward, RTOSs are not only designed for uniprocessor platforms but also improved to be compatible for multiprocessor platforms.

A **Multiprocessor Operating System** is a type of operating systems designed for a single computer system with at least two central processing units (CPU), which communicate with each other by sharing the computer bus, memory and other peripheral devices. These types of systems are suitable for very high-speed processing of a large volume of data and usually used in satellite control, weather forecasting and other similar environments. In chapter 2, an overview of current multiprocessor-capable RTOSs will be given to provide a context for the work of the thesis on LITMUS^{RT} [6] [7]. Figure 1.1 depicts the basic structure of multiprocessing systems.

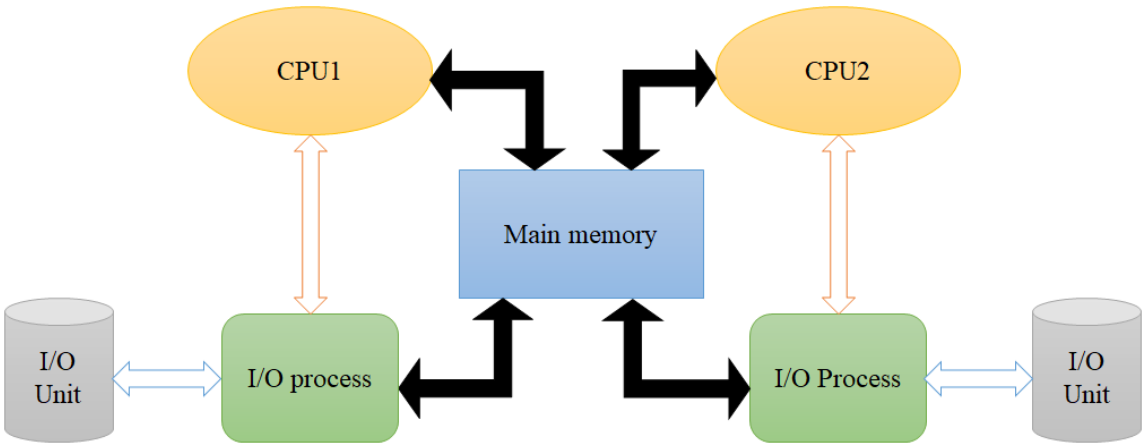


Figure 1.1: The basic structure of multiprocessing systems

One of the essential advantages of the multiprocessor systems is that they finish more tasks in a shorter period of time. Further, multiprocessor systems demonstrate more reliability in the situations of failure in comparison with the uniprocessor systems. In such situation, the multiprocessor system will not stop the processing, rather it will only slow the speed of the system down. To this extent, multiprocessor systems are taken into consideration to reach optimal scheduling algorithms as well as resource synchronization protocols suitable for different cases. A scheduling algorithm can be either reasonable to use if it could take the most advantage of the system, while reducing the response time, or labeled as an inadequate scheduling algorithm if it wastes the processors utilization. Likewise, a qualified resource synchronization protocol can play an effective role for the system's performance in different aspects. However, the recognition, if an algorithm or a protocol is adequate enough, differs from a real-time application to another and depends on various factors; for example, some systems are trying to reduce the average response

time, while for another system deadline meeting is important. In this regard, depending on diverse systems, different scheduling algorithms and protocols should be used.

A *Locking* system is one of the most important things in the computer system for employing an effective multiprocessing operating system. The main goal of locking is the serialization of the access to the protected resource by multiple tasks. A locking scheme is important, because it protects the access to the resources shared among multiple processors. After all, the concentration on the resource synchronization and locking concept can help researchers to exploit the best performance of the multiprocessor's computing ability as much as possible whenever they use it in the real-time system.

In the rest of this chapter, some basic concepts of real-time systems such as task scheduling, locking protocols and resource synchronization are explained that will be used throughout the thesis. After that, the goal of this master thesis and some prior works will be illustrated.

1.1. Task Scheduling

A real-time system usually consists of a recurrent task-set denoted $T = \{\tau_1, \dots, \tau_n\}$ [8]. A process or task τ_i is a recurrently executed program that given a sequence of inputs and produces a set of outputs. The recurrency of a task means that, the task releases an infinite number of task instances, which are referred as *jobs*. The x^{th} job of task τ_i is denoted J_{x_i}. Each task τ_i is characterized by its arrival time or *period* denoted p_i, its *relative deadline* denoted D_i and its *worst-case execution time (WCET)* denoted C_i. The period of a task indicates how often the task will be released. The relative deadline of a task is the time length between the arrival time and the absolute deadline, which is the latest time each job of the task should finish its execution at. The worst execution time of a task is the maximum time length between the start of the task execution and its finish time. Additionally, task *utilization* is the portion of the processor bandwidth that task requires [10].

Nowadays, most of the real-time systems are *Multi-Tasking* systems, i.e., the execution tasks are competing against each other for shared resources. Therefore, scheduling decisions are needed to determine: (i) when should a task be scheduled? (ii) which task should be scheduled? and (iii) how tasks should be scheduled? Scheduling is the process of the decision-making, which is used on a regular basis in many systems [11]. One thing that the task scheduling in the domain of real-time systems deals with, is the access to shared resources through tasks over given time periods and its target is to make one or more objectives optimal such as getting maximum performance from the system, minimizing the average response time, etc. The shared resources can be either external objects like buses, accelerators, GPUs, storage, etc. or can be logical shared resources, that is a piece of code executed by the processor. There exist three fundamental scheduling approaches under the multiprocessor environment: *global* [12] [13], *partitioned* [12] [13] and *semi-partitioned* [14].

Under **Global Scheduling**, all tasks are scheduled through one global scheduler, which uses a single ready queue during run-time as can be seen in Figure 1.2. This scheduling approach allows all jobs of tasks to migrate among multiple processors, i.e., a job of a task that starts its execution

on a processor, may be finish it on another processor. Under this scheduling approach, a system with *m* processors selects at any time at most *m* tasks with highest priority to schedule them on the *m*-processors platform.

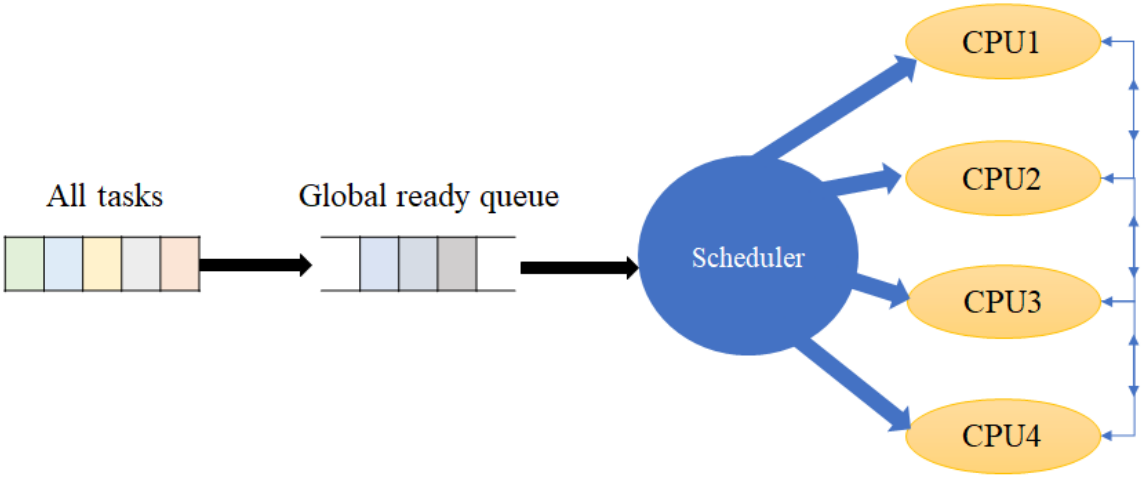


Figure 1.2: Global scheduling

Global scheduling can offer some advantages. For instance, either in adaptive systems where requirements of tasks are changed during runtime for responding to environmental changes or by using open systems in which tasks can be added to or removed from the system dynamically, global scheduling is a proper approach due to dynamic allocating tasks to the processors, so it is not needed to consider the complex task mapping for such systems [15]. Furthermore, through the use of this scheduling approach, the tasks should be preempted and do context switches only when there are no idle processors. Nonetheless, the opportunity of the tasks' migration among processors in this approach causes high overhead, which is one of the worst disadvantages of global scheduling. Another disadvantage of this method is that the scheduling protocols such as Rate Monotonic (RM) [16] or Earliest Deadline First (EDF) [16], which are known as the *optimal scheduling algorithms* on uniprocessor systems, are not optimal anymore on multiprocessor platforms [17]. A scheduling algorithm is labeled optimal if it can make a task-set schedulable, whenever there exists any scheduling approach that can make a task-set schedulable [10]. However, for global scheduling there are some efficient analysis [18] and also many works have provided new scheduling approaches like the proportionate fair (pfair) scheduling approach [19], which are optimal under particular circumstance, such as no preemption and migration. But unfortunately, they usually produce high run-time overheads [20]. The literature on these optimal scheduling algorithms is extensive and a comprehensive review is beyond the scope of this dissertation. Instead, the interested reader can look up on [21] [22].

Under **Partitioned Scheduling**, the system allocates tasks to fixed processors during design time and all jobs of each task will be executed on the same processor to which the task is assigned. Per processor there exists a local ready queue and, on each processor, a uniprocessor scheduling algorithm like Rate Monotonic (RM) [16] or Earliest Deadline First (EDF) [16] will be applied

for scheduling the tasks independently as it is shown in Figure 1.3. By using this type of scheduling approach, different or identical scheduling algorithms may be used on each processor.

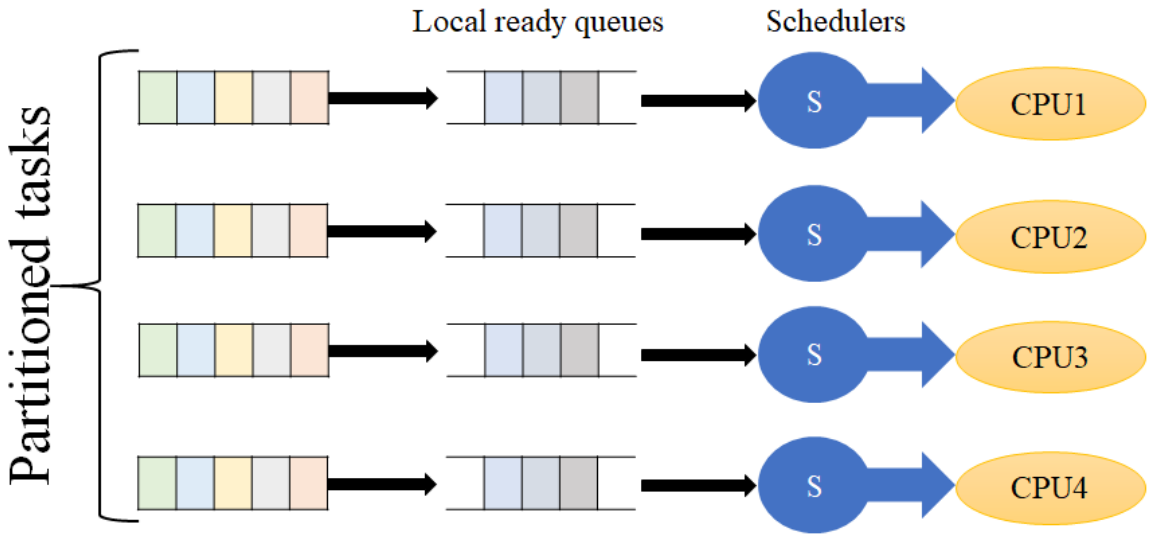


Figure 1.3: Partitioned scheduling

The simple implementation of partitioned scheduling is one of the most important advantage of this approach, which makes it more popular than other approaches. Another advantage of using partitioned scheduling is its run-time efficiency, because the task migration from a processor to another is not allowed in this method. Nevertheless, since partitioning is in fact a *Bin-Packing* problem (see section 2.3.1) and therefore NP-hard in the strong sense [23], we cannot find an optimal way to assign tasks to the processors in a polynomial time. Hence, we have to take advantage of heuristic algorithms to do the task mapping process (i.e., assignment of tasks to processors). The other weak point of partitioned scheduling is that processors will not be utilized fully some times. According to [24] and [25], if the utilization of a task-set is higher than $\frac{m+1}{2}$, partitioning over *m* processors is not possible, i.e., if some task-sets have the total utilization higher than 50% (even just a little bit higher than), tasks might miss their deadlines [26]. Despite these mentioned disadvantages of partitioned scheduling approach, the usage of this approach is still preferred by most of the real-time operating systems [10] for reasons of its uniprocessor legacy, trivial implementation complexity and POSIX-compliant real-time [27].

Both global and partitioned scheduling will be described more in detail in section 2.3.

Beside global and partitioned scheduling, another approach of scheduling termed *semi-partitioned* scheduling has been introduced, which combines partitioned and global scheduling on the same platform to benefit from the advantages of both approaches and to minimize the disadvantage of each.

Under **Semi-Partitioned Scheduling**, on one hand similar to partitioned scheduling, a separate scheduler and local ready queue is allocated to each processor for scheduling the partitioned tasks on each processor. On the other hand, similar to global scheduling, the partitioned tasks can

migrate from the ready queue of one processor to the ready queue of another as depicted in Figure 1.4.

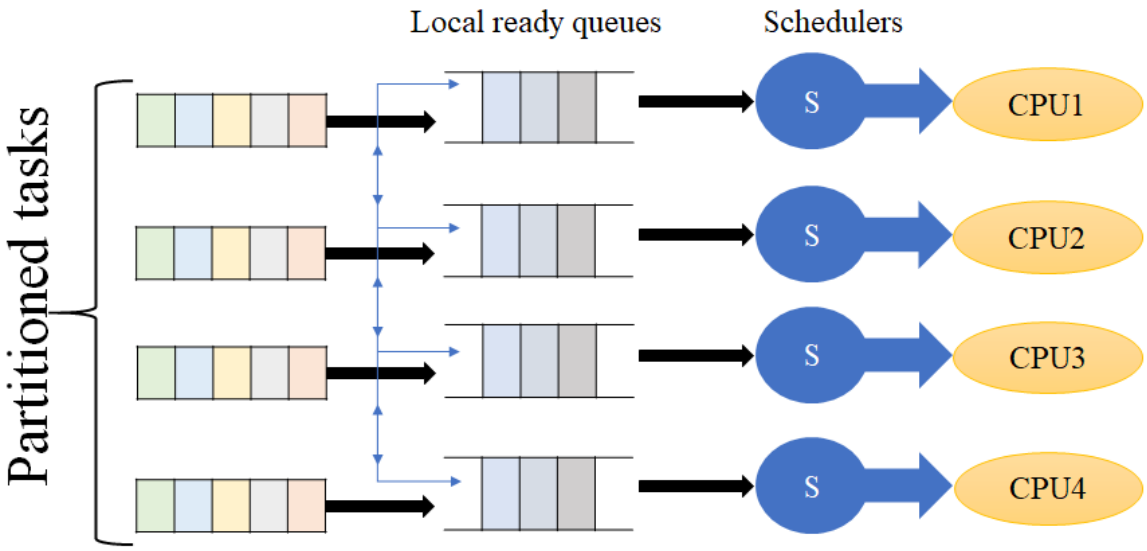


Figure 1.4: Semi-Partitioned scheduling

The semi-partitioned scheduling helps to utilize processors in a better way to compared with partitioned scheduling through the use of the remaining capacity on each processor to schedule the tasks that could not fit on any processor through the migration opportunity. It means that, whenever a task could not fit on any processor, its execution has to be split among multiple processors (jobs of the task are executed on different processors) [10]. According to [28] the utilization bound of task-sets on each processor can be increased as high as the utilization bound of Liu and Layland's RM scheduling [16] for an arbitrary task-set. One of the famous types of semi-partitioned approached is deemed *cluster-based* scheduling. Under a cluster-based approach, processors are divided to some clusters, such that each cluster involves a set of processors and then the tasks are allocated to the clusters and as Figure 1.5 shows, the tasks of each cluster are scheduled globally [29]. Cluster-based scheduling could be considered as a partitioned scheduling if the number of clusters is equal to the number of system processors and on the contrary, a cluster-based scheduling maps to global scheduling when the system consists of only one cluster. In Figure 1.5 a 2-cluster system is illustrated where each cluster includes 3 processors. There are two main categories for the cluster-based scheduling, *physical* and *virtual*. By using the physical type of the cluster-based approach, each cluster is allocated to a fixed set of processors, whiles under the virtual type, the clusters are assigned to the processors dynamically. The discussion about these two types is irrelevant to this thesis and omitted here in the interest of brevity. Instead, for a comprehensive overview of the state-of-the-art in types of cluster-based scheduling, the interested readers are referred to [30] [29].

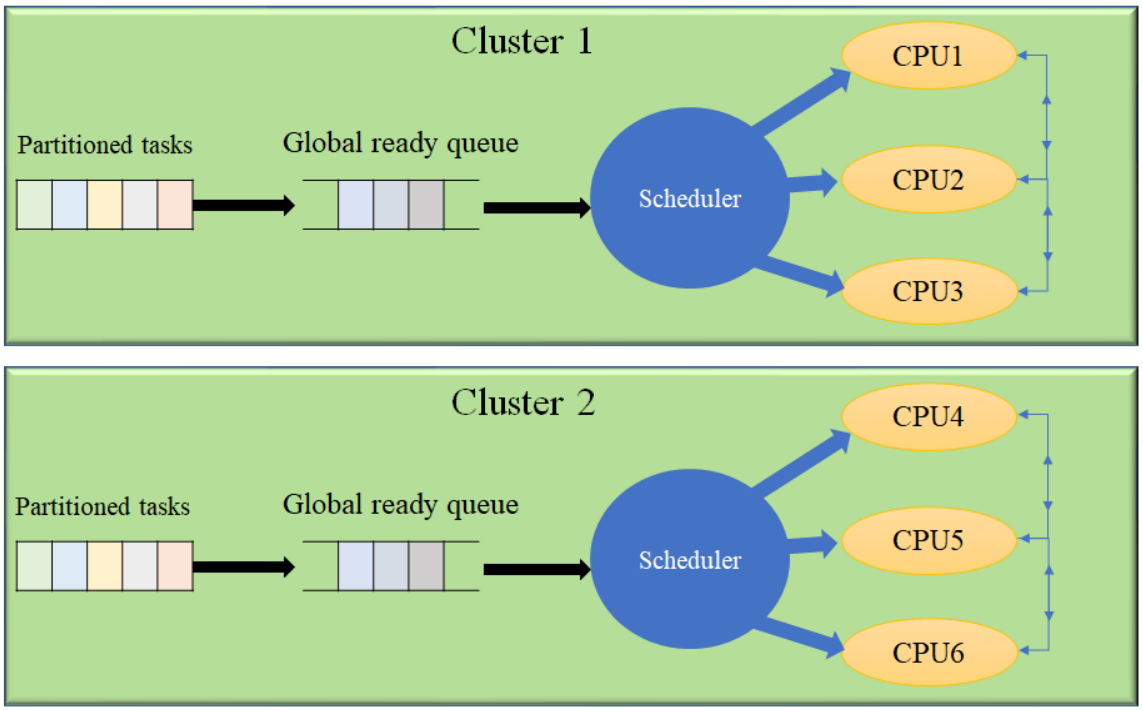


Figure 1.5: Cluster-based scheduling

For each fundamental scheduling approach mentioned above, there are two types of scheduling algorithms. One of them is deemed *Fixed Priority Algorithms* such as Rate Monotonic (RM) [16] and Deadline Monotonic (DM) Algorithm [31], and the other one is called *Dynamic Priority Algorithms* such as Earliest Deadline First (EDF) [16] and Least Slack Time (LST) [32] Algorithm. In two following subsections, these two types of scheduling approaches will be discussed shortly.

1.1.1. Fixed Priority Algorithm

At present, this type of scheduling algorithm is the most widely used approach and it is also the distinct focus of this thesis.

Under this class of scheduling, the tasks will be ordered at time zero (before execution of the system) according to a priority list [11]. This priority list will not be altered during the process of each task execution and at any time point, in which a processor is free, the task at the top of the queue of ready tasks should be picked up as the next task for processing. In fixed priority scheduling, it should be guaranteed that at any time, the runnable task with highest priority is executed [16]. In this regard, if the priority of the current running task is lower than the priority of a task, which is released right now (i.e., an internal or external event is occurred, and the dispatcher handle it), the task with low priority will be preempted (if preemption is allowed), and the task with higher priority will start to execute. If during the execution of the high priority task, a task with medium priority arrives, the high priority task will continue its running, and after some later time, when it finished its computation, the medium priority task can start to run. The task with lower priority can resume its execution only when all tasks with higher priority have completed

their jobs, and then the low priority task can keep on running unless a task with higher priority arrives.

In this master thesis, the **Partitioned Fixed Priority** (P-FP) scheduling algorithm is adopted due to its advantages, which are:

1. Comparing to the global scheduling (described in the next section), partitioned based scheduling has lower overheads due to limited migration.
2. By exploiting P-FP approach, each processor of the system can be considered separately.
3. There exist many well-established algorithms based on uniprocessor.
4. P-FP can support all synchronization protocols, which are used for this thesis.
5. It is simple to understand P-FP and deal with it.

More detail about the partitioned fixed priority scheduling algorithm, specially the Rate Monotonic (RM) [16], which is used in this thesis, will be given in section 4.1.1.

1.1.2. Dynamic Priority Algorithm

Under a dynamic priority scheduling algorithm, a sequence of decision is required to be taken to be able to calculate the priorities during the execution of the system. If this class of scheduling is used in a real-time system, whenever a processor is ready to use, the scheduler will decide which task can be executed as the next one. This decision may depend on some available information such as the current time, the tasks waiting for processing and etc., but such decision should be taken without need to the knowledge about the future tasks [33].

Dynamic scheduling is very useful for many real-time applications, which should be more flexible in dealing with practical issues, such as scheduling decision, when the environment changes or a part of the system fails and etc. For instance, robotic systems require the support of dynamic scheduling, because in such systems control section must adapt to a dynamic environment [33]. Thus, by using this type of scheduling algorithm, the adaption to the dynamic changing progress can be supported easily. However, producing a well-defined policy to afford all dynamic changing in an optimal manner can be very hard depending on the difficulty of a specific problem [32]. There exist three basic steps for dynamic priority method: (i) feasibility checking, (ii) schedule construction and (iii) dispatching. It is not necessary to apply all these three steps and the use of each step depends on the kind of application for which the system is designed, the programming model adopted and the scheduling algorithm used [33].

Since in this thesis the dynamic priority approach is not used, the discussion about this approach will be omitted here for the sake of brevity and the interested reader can be referred to [33] for a detailed discussion of this approach.

1.2. Concurrent Resource Access

Most scheduling approaches assume that tasks are independent, thus they are never delayed by actions of other tasks. But the assumption of independent task execution is not always true,

especially when these approaches are supposed to be used in an embedded system. Since the number of available resources such as queues, buffers or I/O devices in the embedded systems are constrained, it is needed to deal with the resource sharing through the tasks [10]. For example, if a message must be transmitted by a task, but the required network device is already in use, then the task cannot progress to running until the shared resource becomes available. Such behavior can put the temporal correctness of real-time tasks at risk and result in application failure [34]. Therefore, in such system to avoiding the data corruption, it is necessary to handle the simultaneously shared resource accessing. There are two main solutions for this problem, which are discussed in the following subsections.

1.2.1. Lock-Free

Lock-free is one of the solutions of resource access handling. In a lock-free protocol [35], the task's trying on achieving a shared resource will be continued, until the task gains its intended resource. One of the advantages of using lock-free protocols is that the support by the operating system is not a requirement of using this method. Moreover, we do not need to worry about *priority inversion* due to lack of the usage of locks [10].

Priority Inversion phenomenon or in the specific definition for real-time resource sharing denoted as **priority inversion blocking** (pi-blocking) happens when a lower priority task prevents a higher priority task from executing for an unbounded amount of time [8]. Such inversions can cause problem in the schedulability analysis, because in the concept of most scheduling approaches, additionally to assumption of the task independently, it is also assumed that, a task always carry on to run whenever it has the highest priority. Thus, this assumption will be violated, if priorities are inverted. In such wise, the response time of a task with inverted priority can be longer than the worst-case response-time bound assuming independence [10]. Figure 1.6 presents a simple case of priority inversion.

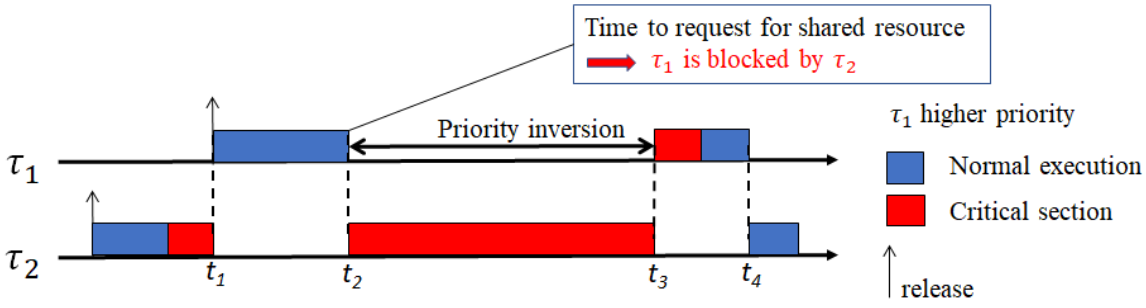


Figure 1.6: Priority Inversion

One of the disadvantages of using lock-free, which makes the usage of this method not suitable for real-time applications, is that the number of retries of the tasks for accessing the shared resources cannot be bounded easily. In this regard, predictability, which is the most essential specification of the real-time systems, cannot be afforded by using lock-free protocols [10]. In this

thesis, therefore lock-free based protocols will not be considered and the interested reader can be referred to [36] for an in-depth discussion.

1.2.2. Locking

Beside lock-free protocols, there is another alternative approach, named locking, to handle the concurrent access to shared resources, i.e. *critical section* of the task execution, and achieving the *mutual exclusive* access [10].

Critical section or **critical region** is a protected part of a program, where shared resources of a system such as data structures, peripheral devices, network connections, etc. are required to be accessed. This protection of the section is needed to prevent from concurrently access to the shared resources, otherwise it can lead to unexpected or erroneous behavior. Actually, these protected sections are not allowed to be executed by more than one process at the same time [37].

Mutual exclusive means that two or more events cannot occur concurrently. Mutual exclusion is on one hand, the requirement that a task should not be allowed to execute its critical section, while another task is in its critical section on the same resource and on the other hand, it is the most important property of concurrency control, which helps to prevent *race condition* phenomena [38].

A **race condition** or **race hazard** is a behavior in the system, which means that the output of a program depends on the sequence or timing of other events, i.e., if this sequence does not happen in the intended order, then the system encounters bugs [39].

The task blocking may be happened either owing to asking for a resource that a lower priority task locked it already, or when a task with lower priority prevents the task from being scheduled due to being non-preemptable. This intimates that a task can be blocked even though it did not ask for any resource [10] [34]. By usage of the locks to manage the access to the shared resources, whenever a task asks for a resource, the task has to lock the requested resource before using or rather holding it. Among the tasks, which request a same resource, the resource will be used by the one that holds it and then the task can enter to its critical section, where it uses the resource [15]. On this wise, the predictability will be afforded, which is essential in real-time environment. Therefore, the focus of this thesis is on locking protocols. However, the usage of the locks causes priority inversions (see Figure 1.6), which should be taken into account, because they can be reason of the unpredictable delays. To this end, blocking on the real-time systems needs to be bounded. Additional problem of using locks is *deadlock* phenomenon, which can result in system crash [40].

Deadlock is raised by nested resource locking. As it is illustrated in Figure 1.7, if a task requests a resource, while holds another one and at the same time another task asks for the holed resource by the first task, while holds the intended resource by the first task, then system encounters deadlock and none of the tasks can resume their execution [40]. Due to the complexity of deadlock consideration, nested resource locking is not the experience case in this thesis.

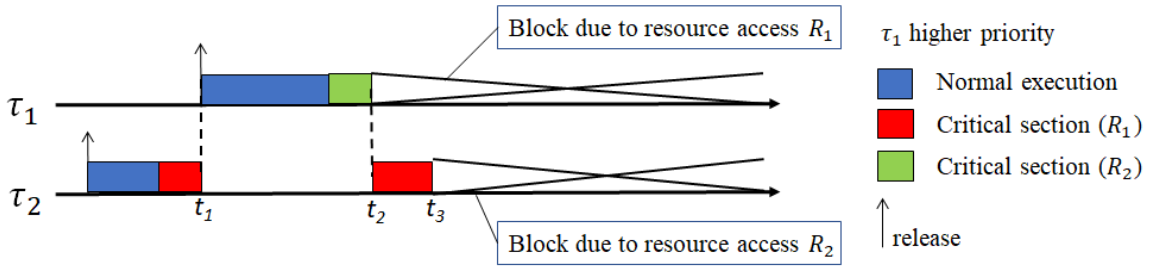


Figure 1.7: Deadlock

Tasks may cause blocking, whenever they use local or global resources. In case of accessing a global resource, there are two main techniques to identify if a task has permit to access a global resource: (i) ordering or queueing the pending requests of multiple tasks, (ii) leave the pending requests of multiple tasks unorder. Through the use of the queueing method, each global resource has a unique global queue, which contains the requests of the tasks waiting for that resource. Figure 1.8 depicts this method in a system with 3 processors and 2 global resources. The requests of the tasks on each queue can be ordered based on their priority or based on first-come first-serve manner (FIFO) or based on a combination of both. On one hand, the priority-based queues are proper to service the high priority tasks as soon as possible, because whenever a task with higher priority is enqueued, it is placed ahead of lower priority tasks. On the other hand, the priority-based queues are not suitable for the lower priority tasks, since they may suffer *starvation*, due to adding always the higher priority tasks in the same resource queue before a lower priority task. By using the FIFO-based queues, due to its first-come first-serve policy, the lower priority tasks can be placed ahead of higher priority tasks in the queue if they came before and thus they can acquire the resource sooner. However, considering the worst-case scenario, in which the size of the queue is not bounded, a higher priority task must wait for all lower priority tasks [10]. In [41] it is shown that none of the queueing techniques dominates the other. In this thesis, the FIFO-based queueing technique is used in the experiments.

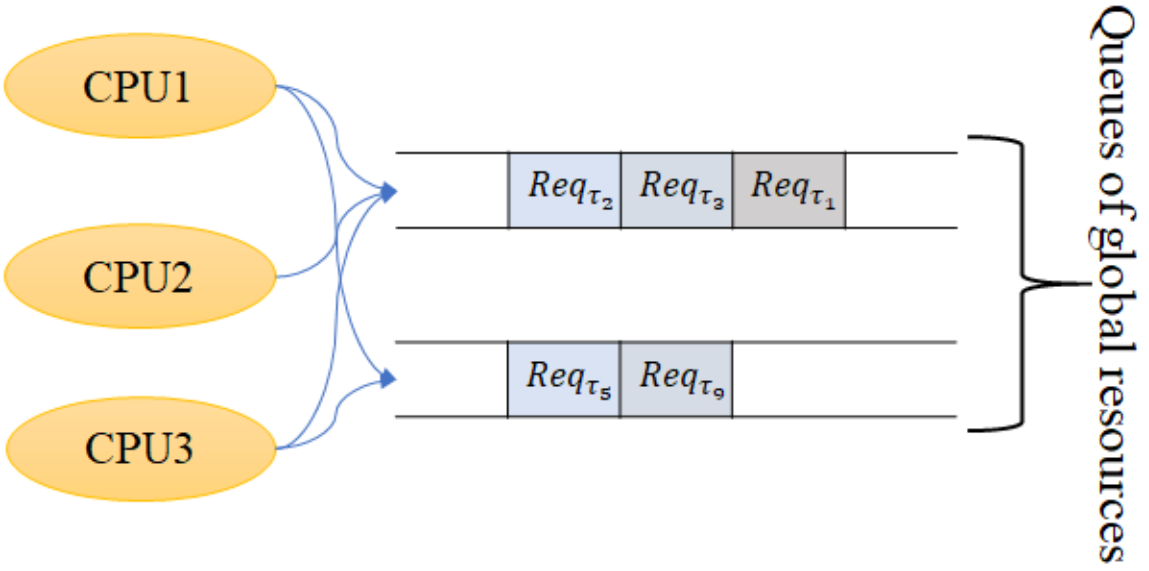


Figure 1.8: Global resource queues ($\tau_i s$ are the tasks, which are waiting for a global resource)

1.3. Synchronization

As discussed in section 1.2, tasks are not always independent from each other but may share some resources of the system. Thus, the handling of the resource sharing is one of the essential components in RTOS. To this end, as argued above, we can use the locking method. However, the use of this method causes two big problems, i.e., priority inversion and deadlock. In such manner, resource synchronization protocols are required to solve these problems. In fact, the locking protocols provide the consistency in the real-time system through the synchronization of the access to the shared resources. Two well-known of these protocols for uniprocessor systems are *Priority Inheritance Protocol (PIP)* [1] and the *Priority Ceiling Protocol (PCP)* [1], which are described briefly as follow:

- **Priority Inheritance Protocol:** The priority inheritance protocol (PIP) introduced by Sha et. al [1] is a solution to solve the problem of unbounded priority inversion phenomenon under uniprocessor systems. This protocol modifies the priority of those tasks that cause blocking, in order to avoid unbounded priority inversion. In particular, if one or more tasks are blocked by task τ_i with lower priority, the priority of task τ_i is raised temporarily to the priority of task τ_j (i.e., task τ_i *inherits* the priority of task τ_j), which has the highest priority among the blocked tasks. That means, no task with priority lower than the priority of task τ_j can preempt task τ_i, thus the blocking duration of the higher priority tasks will not be extended. Although the PIP could solve the problem of priority inversion, it might cause a deadlock if there are multiple resources [8].
- **Priority Ceiling Protocol:** The priority ceiling protocol (PCP) proposed by Sha et. al [1] solved not only the problem of unbounded priority inversion, but also the problem of deadlock. Actually, the PCP is the extension of the PIP to a rule for granting a lock request

> on a free resource. According to this rule, a task is not allowed to enter its critical section, if it could be blocked by requesting a locked resource, thus multiple blocking is avoided. Under this protocol, each resource has a *priority ceiling*, which is equal to the highest priority of all the tasks that require the resource. Once a task is allocated a resource, its priority is temporarily increased to the priority ceiling of that resource. Then, a task can gain a free resource only if its priority is higher than all priority ceilings of the resources currently locked by other tasks. This means, whenever a task acquires a shared resource and starts to execute its critical section, no task can block it until its completion. Through the use of the PCP as the shared resource access protocol on a uniprocessor system of preemptive, priority-driven tasks, deadlock can never occur and a task can be blocked for at most the duration of "one" critical section [8].

The resource synchronization becomes more complex when considering multiprocessor platforms instead of uniprocessor platforms. In a multiprocessor platform there are two types of blocking: (i) direct blocking and (ii) remote blocking. If the execution of a task suffers delayed because of being blocked through a lower priority tasks on the same processor or on the different one, the direct blocking and remote blocking is happened respectively. Both types of delay should be taken into consideration because they effect on the tasks' response time [10].

When a task requests a resource, it does not matter local or global, it is not allowed to lock the resource immediately, i.e., when a task is waiting to obtaining its required resource, first, the task is blocked on the resource and then, as soon as the task is eligible to access the resource, it locks the resource. In a multiprocessor platform the performance of a task, which is remotely blocked, may be either spinning to the processor (a preemptive/non-preemptive busy-wait) or being suspended and releasing the processor. Both spin-based and suspension-based protocols have some advantages and disadvantages, thus neither of them is completely preferable to the other, i.e., some task-sets can be schedulable only by exploiting the spin-based protocol or just under the suspension-based protocol. One of the differences between spin-based and suspension-based protocols is that under a spin-based protocol, the processor bandwidth will be wasted through the task, which is spinning to the processor, for the whole waiting time to acquire a remotely held resource. But, under a suspension-based protocol, the task waiting for the resource will be suspended and by releasing the processor, the other tasks are allowed to use it. In such situation, these running tasks on the processor may issue more resource requests, meanwhile the task is suspended due to wait for a remotely held resource. On this wise, in the worst-case, extra delay will be added to the suspended task by these new requests under a suspension-based protocol. Another difference is that by using spin-based protocols, context switch overheads will be smaller in comparison with the situation, in which the suspension-based protocols are in use [10].

In a multiprocessor platform, resource sharing protocols handle the access to the shared resources. The first real-time locking protocols for multiprocessors were proposed in [3] [4], which are actually two extensions of the uniprocessor locking protocol, i.e. priority ceiling protocol (PCP) [1], for distributed- and shared-memory multiprocessors under P-FP scheduling, called the distributed PCP (DPCP) [3] and the multiprocessor PCP (MPCP) [4] respectively. In this thesis

not only the MPCP is investigated, but the DPCP is also included because neither one of the MPCP and DPCP dominates the other. One of the properties of these both multiprocessor PCP variants is the usage of the *priority boosting* technique.

Priority boosting is a technique that increases the task's priority, i.e., the priority of a resource-holding task becomes higher than (without respecting to its "primitive" priority) the priority of the other non-resource-holding tasks [34]. The goal of this technique is the speeding up of the completion of the resource requests. This means that, a resource-holding task can continue to execute until the completion of its critical section without being worry about the preemption by a newly-released task with higher priority and as a result, the unbounded priority inversion can be prevented easily [34]. Figure 1.9 exhibits the act of priority boosting. In this figure the higher priority task τ_1, which is released at time point t_2, cannot preempt the lower priority task τ_2, which is in the middle of the execution of its critical section at the same time point, because of the use of priority boosting technique here. Although, in such manner the priority boosting behaves like a non-preemptive execution, the priority-boosted tasks are allowed to preempt the other priority-boosted tasks, unlike non-preemptive execution.

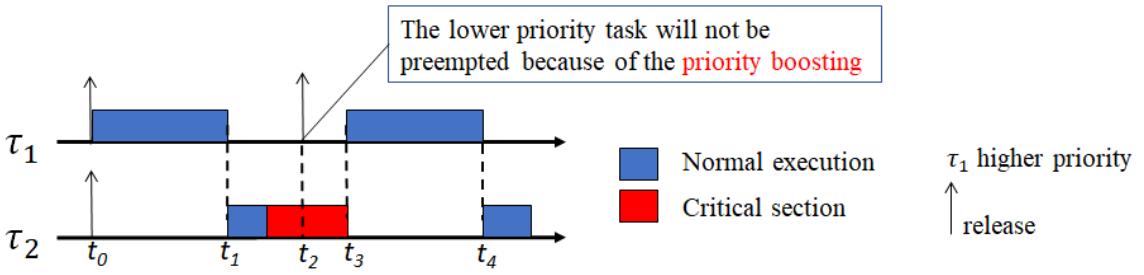


Figure 1.9: Priority boosting

Besides the priority boosting, sharing of the same queuing discipline is another attribute of the MPCP and DPCP, which implies that, if there are multiple requests for a same resource at the same time, the requests are ordered in decreasing priority [34].

In addition to the similarities between the MPCP and DPCP, they are different from each other definitely and the main difference between these two protocols is the execution position of the resource requests. These protocols will be reviewed in detail in section 2.5.

1.4. Thesis Goal

Unfortunately, the topic of multiprocessor real-time synchronization has been somewhat neglected in comparison to real-time scheduling and unlike scheduling, there does not exist so many works on this subject, even though, both of these topics are equally-important.

As argued in previous subsection, to solve the problems of priority inversion and deadlock, which are produced through the use of locking method, we need to use the resource synchronization protocols and in respect to the multiprocessor environment, two types of resource synchronization

protocols, spin-based and suspension-based protocol can be in use. Although, many suspension-based locking protocols and spin-based locking protocols have been designed and analyzed in the last years, it has been relatively little investigated empirically that, under which settings which one of the suspension-based and the spin-based locking protocols are in general preferable.
Despite the fact that, both approaches have some advantages and disadvantages, the choosing of one of them under different situations depends on the problem objectives, which should be optimized in that circumstance. It means that, sometimes the use of the spin-based locking protocols can provide better results in respect to a specific objective in comparison with the usage of the suspension-based locking protocols and vice versa. In this regard, such tradeoff should be investigated and some break-even settings has to be found and analyzed.
The effort to find this tradeoff leads to the motivations of the topic of this thesis. To this end, the proper multiprocessor locking protocols must be analyzed and categorized under two groups: suspension-based and spin-based. After that, a testbed should be determined and set up based on the appropriate settings to analyze the intended locking protocols under different parameters. All these steps will be discussed in the following chapters.

1.5. Prior and Related Works

In a multiprocessor real-time environment, there has been relatively little work on the topic of synchronization. To handle the synchronizing accesses to shared resources, locking is one of the most popular method, for which there exist two basic options: spin-based locking, and suspension-based locking. There are not many experimental works on multiprocessor-based evaluation of these basic techniques that focus on multiprocessor real-time systems. Some of them are outlined below.
In the work of Wieder and Brandenburg [41] both preemptive and non-preemptive spin-based protocols have been investigated for both ordered and unordered techniques. In this work they considered both FIFO and priority-based ordering for the ordered variant. They used also a de-queueing policy (see section 2.4.1) to avoid the transitive arrival blocking problem which is the result of the conjunction with FIFO-based queues and preemptive spin-based protocols. In their work, in order to bound the maximum cumulative blocking, which is imposed to tasks to achieve tighter bounds, a Mixed-Integer Linear Program (ILP) technique has been used too.
Another work of Wieder and Brandenburg with Biondi‘s collaboration [42] has introduced a new blocking analysis based on graph abstraction for nested non-preemptive spin-based protocols, which is itself based on FIFO-based queues.
The work in [43] has investigate the spin-based multiprocessor real-time locking protocols for replicated resources, where multiple replicas may be used by tasks.
Again, Brandenburg et al. in [5] have studied lock-free, wait-free and both spin-based and suspension-based protocols from the lock-based synchronization protocols on the LITMUS^{RT} platform [6] [7]. In this study, it is concluded that, the non-blocking algorithms can be preferable if all critical sections are short whiles, wait-free or spin-based protocols are favorable for larger

and more complex shared data. Moreover, they have also concluded that, blocking by suspending is rarely preferable to spinning and the use of suspension-based protocols are not recommended for partitioned systems.
Among the above-mentioned works, the work of Brandenburg et al. in [5] has the most relation to the subject of this thesis. However, the observation by him in that work could be resulting only from the properties of the FMLP and not reflect the spinning and suspension in general. Therefore, in this thesis other multiprocessor locking protocols are under consideration.

1.6. Thesis Structure

The thesis is divided into 5 chapters. Chapter 1 gives a background describing some basic information about the real-time environment and also in this chapter some concepts of the multiprocessor real-time systems related to the subject of this thesis and the problem and the existing solutions, e.g. scheduling and synchronization protocols are introduced. Chapter 2 describes the multiprocessor real-time systems, in particular the real-time locking protocols for the sharing of global and partitioned resources and also the spin-based and suspension-based synchronization mechanisms under partitioned scheduling will be studied in this chapter. The used test-bed and its installation and modification will be introduced in chapter 3. Chapter 4 presents the process and the explanation of the testing parameters as well as the results of the testing and also focuses on the analyzing and the evaluation of these results. Finally, the conclusion of current research in this thesis and the future works are summarized in chapter 5.

2 Multiprocessor Real-Time Systems

In the end of the last century, the uniprocessor systems could not support the raising request of computer capacity any longer. Hence, the multiprocessor concept was suggested and developed by several companies. A multiprocessor system consists of two or more independent processors (cores) on a single chip. The system connects processors to a single shared memory through a shared bus. Figure 1.1 depicts the basic structure of multiprocessing systems.

After the availability of the multiprocessors in the last several years, these systems were not taken into consideration for reasons of their high cost, high power consumption and less performance improvement. That is to say that, it was not worth to suffer the high cost and energy consuming, while the performance of the multiprocessor real-time system was not improved impressively. However, in virtue of the rapidly growing of the chip technology over the last few years, multiprocessor chips become more popular than before because of their faster computing speed, lower price, lower power consumption and smaller volume.

Todays, multi-core architectures are the most famous technology, which are used in desktop computing industry and are identified as the de facto processors generally [44]. The use of the multiprocessor systems since several years proves that, this kind of real-time systems are more reliable in the situations, which are the failure situations for uniprocessor. Nevertheless, the performance of using multiprocessors varies for applications with different nature as well as different software implementation. Since the concurrency is the main property of the multiprocessor architecture, the system has to benefit appropriate algorithms for dividing the software into tasks (threads) as well as tasks partitioning on processors in order to have the high performance. The lack of such fairly distribution in an application can cause some problems like a heavy work by just one task, hence the significantly improving of the performance cannot be fulfilled by the usage of the multiprocessor technology. Since real-time systems are basically multi-threaded, they can adapt to the multiprocessor technology easier than single-threaded and sequential programs as well. It means that, critical functionality can be separated on different processors and the multiprocessor system can take the advantage of concurrency by running the independent tasks concurrently in order to improve performance. Additionally, because of the location of all cores on the same chip (in the most multiprocessor systems) and also sharing the same memory, such systems benefit from a very fast cores communication [44].

There exist two architecture models to design a multiprocessor system; in the first model, named *homogenous*, the system includes identical processors, i.e., the performance of all processors is the same. The disadvantage of this model is heat and power consumption problems, which is disappeared in the second model of multiprocessor architecture. In the second model, the processors do not have the same performance, thus each task can be allocated to the suitable processor, which means that the tasks with the requirement of lower performance can be local on

the processors with lower performance; in this regard the energy consumption will be decreased [44].

In the rest of this chapter, first real-time task and resource model will be introduced. Next, two basic approaches of multiprocessor scheduling will be reviewed and afterwards the multiprocessor synchronization mechanisms and some relative locking protocols will be discussed.

2.1. Real-Time Task Model

A real-time system usually consists of a recurrent task-set denoted $T = \{\tau_1, \dots, \tau_n\}$ [8]. A process or task τ_i is a recurrently executed program. The recurrency of a task means that, the task releases an infinite number of task instances, which are referred as *jobs*. Generally, each real-time task τ_i executed on a homogeneous, real-time system with multiple processor, is characterized by the parameters as follow [8]:

- **Arrival or release time** (a_i) is the time at which a task becomes ready for execution;
- **Period** (p_i) indicates how often the task will be released;
- **Worst-case execution time** (C_i) is the maximum time necessary to the processor for executing the task without interruption;
- **Absolute deadline** (d_i) is the latest time each job of the task should finish its execution at;
- **Relative deadline** (D_i) is the time length between the arrival time and the absolute deadline of the task: $D_i = d_i - a_i$;
- **Start time** (s_i) is the time at which the task starts its execution;
- **Finishing time** (f_i) is the time at which the task finishes its execution;
- **Response time** (R_i) is the time length between the finishing time and the arrival time of the task: $R_i = f_i - a_i$;
- **Utilization** (U_i) is the portion of the processor bandwidth that task requires [10]: $U_i = \frac{C_i}{p_i}$.

Some of the parameters defined above are illustrated in Figure 2.1.

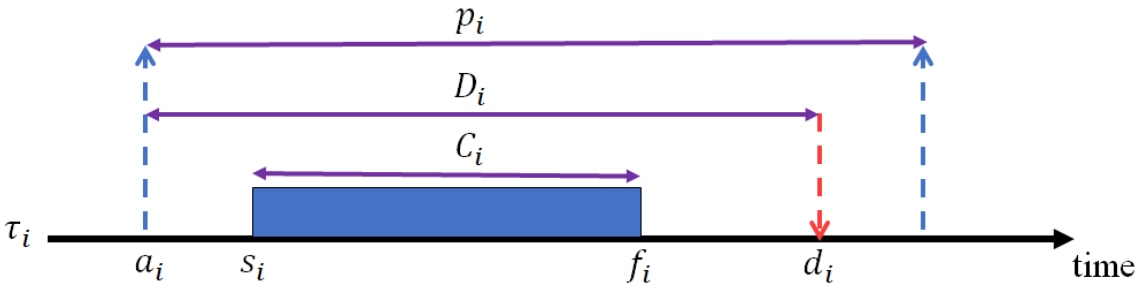


Figure 2.1: Typical parameters of a real-time task

2.2. Real-Time Resource Model

In a multiprocessor real-time system, there are usually some resources such as buses, GPUs, storage, a piece of code and etc., which are shared among different tasks. Each task can access one or more shared resources during its execution time. As a result, it is necessary to introduce the resource model of the real-time system. A resource model defines the parameters relative to a set of shared resources, which are controlled by locking protocols and used by a task as well [8]. The resource sharing among tasks is formalized as follow: The system with m processors includes n shared resources $r_1, r_2, \dots, r_n$. When a resource r_q is requested by a task τ_i, a request $R_{i,q,v}$ ($v \geq 1$) is issued by task τ_i. Such request indicates the v^{th} time that task τ_i requests resource r_q since it has been launched. Upon the holding of resource r_q by task τ_i, request $R_{i,q,v}$ is satisfied and it is not complete until resource r_q is released. Acquisition delay happens for task τ_i, when request $R_{i,q,v}$ cannot be satisfied once it is created, i.e., it is not possible for task τ_i to proceed its computation due to waiting for $R_{i,q,v}$ to be satisfied. While a task τ_i is running, requests can be issued at any time and when a request $R_{i,q,v}$ is being executed, it is said that τ_i is in a critical section. Further, it is assumed that, the order of requests for each resource can be arbitrary and there is not a minimum separation between consecutive requests as well. In light of the resource model used in this thesis, the task should remain scheduled on a processor during the use of a resource, i.e. during its critical section. In this regard, the task starts to execute its critical section as soon as allocated a processor and completes its execution without any preemption.

If a request $R_{i,q,w}$ is issued before that another satisfied request $R_{i,q,v}$ is complete, where $v < w$, then $R_{i,q,w}$ is called as an *inner request*, which is nested within the *outer request* $R_{i,q,v}$. In contrary, to the inner request, an outermost request is not nested within any other requests [34]. All resource requests in this resource model are supposed to be an outermost request, which means that each task requests at most one resource at any time.

In the case of using distribution-based protocols to protect resource accesses, it is necessary to be determined which resource is local to which processor. Indeed, in order to analysis, such resource assignment must be fixed in advance according to an assignment rule. In this thesis, resources are assigned to processors according to the resource assignment rule (1).

2.3. Real-Time Scheduling

The description in this section is intended to provide the needed background for the presentation of the thesis work. Therefore, the section focuses on examples and the description of the discussed approaches, but not on proofs, which can be found in the references.

One of the responsibilities of a scheduler[1] in a multiprocessor real-time system with m processors is the assigning of (at most) m of n pending tasks to m processors, in particular when $n > m$.

[1] Refer to [65] for a formal concept of “schedule” and “scheduling algorithm”.

Schedulers are divided into several categories. One of them is called *static scheduler* (also known as table-driven scheduler). A task-set-specific allocation policy, which is already computed during the design of the system, will be ordained by the static scheduler. Such schedulers can be implemented easily and the process of their validation is straightforward, because the validator needs to check just the pre-computed table. In this regard, static schedulers are identified as an ideal choice for safety-critical systems, although they are not suitable widely for sporadic workloads (with unpredictable inter-arrival delays) due to their inflexibility [34]. Another category of the schedulers is named *dynamic schedulers*. The scheduling rules are evaluated online by dynamic schedulers and the scheduling decisions are based on the current system state such as the set of pending tasks and their parameters, but not on the pre-computed information [34]. This thesis is restricted to the use of the static schedulers.

As it is mentioned before, a static scheduler assigns each task a priority according a scheduling policy. Let $Y(\tau_i)$ be the function of priority assigning to the tasks τ_i. To have an explicit evaluation, the unique priority for each task is required and it is also necessary to order the tasks such that task τ_i placed before task τ_j if $Y(\tau_i) < Y(\tau_j)$. Based on the prioritization function Y, one of the fundamental classes of priority-based schedulers, called *fixed-priority (FP)* scheduler [34], can be described so that all jobs of a task have the same and constant priority. Formally, $Y(j_{x_i}) = Y(j_{y_i})$ for any two jobs J_{x_i} and J_{y_i} of a task τ_i.

In order to apply the fixed-priority scheduling to a shared-memory multiprocessor system, there exists two fundamental approaches: (i) scheduling of all available processors by exploiting one scheduler (with a single, shared ready queue), (ii) each processor is scheduled independently (with a local ready queue) [34]. The former approach is deemed *global scheduling*, and the latter one is named *partitioned scheduling*. The partitioned scheduling case is considered next and the global scheduling will be reviewed afterwards.

2.3.1. Partitioned Scheduling

Dhall and Liu are the first researchers, who studied the partitioned scheduling in the context of real-time systems [17]. Under partitioned scheduling, each subdivision (called a partition in this case) involves only one processor and the task assigning to each partition has already done statically. A uniprocessor scheduling protocol, such as RM [16] and EDF [16] schedules the tasks within each processor. For each task there is a dedicated processor so that the task can run its jobs on that processor. The jobs of a processor are scheduled through a local ready queue associated to the processor (see Figure 1.3).

Partitioned scheduling is one of the multiprocessor real-time scheduling approaches that is used most commonly in practice. One reason for this wealth of choices is that scheduling analyses from the uniprocessor domain, which are sufficiently comprehended and already verified, are capable to be used in multiprocessor systems too. In principle, the use of different scheduling policies on distinct processors is also possible, however, in this thesis just the “pure” FP partitioned scheduler, namely partitioned FP (P-FP), will be considered, because the partitioning with different

scheduling policies is currently not supported in LITMUS^{RT} platform [6] [7] and also the main interest of the thesis is to compare locking protocols under consideration of spinning or suspending, but not the evaluation of different scheduling algorithms, which can be investigated as a future work. Another advantage of the partitioned scheduling is its run-time efficiency due to the lack of tasks migration overhead. Nevertheless, the task preventing from migration among processors causes the reduction of the utilization bound; for instance, it has been indicated in [45] that some task-sets can be schedulable only under the allowing of the task migration among processors. Another disadvantage of partitioned scheduling algorithm is the challenge of fairly task allocation (partitioning) onto processors such that no processor is overloaded. Strictly speaking, this task assignment problem is a *bin-packing* problem and as it is known, the bin-packing problem is NP-hard in the strong sense, thus it is not generally realistic to find an optimal distribution of tasks among processors in polynomial time [44].

Bin-packing problem is one of the fundamental problems in partitioning domain and the complexity of this problem is NP-hard in the strong sense [23] as well. The target of this problem is the distribution of available items to the minimum number of bins without overflowing the capacity of any bin. A bin capacity V and a set of n items $x_1, \dots, x_n$ with respective sizes $a_1, \dots, a_n$ are given as the inputs of bin-packing problem and the output is to determine if there is a solution to assign item to bins such that the items must fit into B bins (B should be minimized) [34].

The relation between bin-packing problem and the partition problem in the context of partitioned scheduling is that, the items and their corresponding sizes in bin-packing problem are the tasks and their utilizations in task partition problem respectively. Each processor of the multiprocessor real-time system is corresponding to a bin and the effective capacity V of each bin or rather each processor is dependent on the scheduling policy in use; for instance, through the use of the EDF scheduling algorithm [16], the capacity of each processor is equal to 1.0. In this case, to be able to partition a task-set, a bin-packing problem should be solved so that an implicit-deadline task-set is feasible on m processors under partitioned scheduling if and only if there is a "packing" of all tasks into m "bins" [34]. Such situation denotes the capability of the bin-packing decision problem to be reduced to task-set partitioning in polynomial time (by scaling item sizes and bin sizes), which indicates that is intractable to find an optimal task assignment in all cases, unless $P = NP$ [23]. In [17] it is formally proved that, the complexity of the task assignment problem is NP-hard in the strong sense through a solution, which can solve the 3-partition problem [23]. From a practical point of view, there are some bin-packing heuristics for finding valid task assignments. Reviewing all the bin-packing heuristics is somewhat tedious and irrelevant to the scope of the thesis and the interested reader is referred to [46] instead. Here just the top three popular bin-packing heuristics are discussed.

The packing of large items (in relation to the bin size) is generally more difficult than packing the smaller items because it is more convenient to place smaller items in the remaining capacity of partially used bins compared to large items. On this wise, packing items in order of decreasing size, such that $a_1 \geq a_2 \geq \cdots \geq a_n$, is more preferable and easier to partition them off-line [34].

After the ordering of items according to decreasing of their size, one of the following placement heuristics can be applied to partitioned a preordered task-set (set of items). In the following description of each heuristic, S_j is the set of assigned items to the j^{th} bin[2] and the remaining capacity of the j^{th} bin is calculated in this way: $V_j = V - \sum_{x_i \in S_j} a_i$. Bins are ordered by their index.

- **First-fit decreasing:** By using the first-fit heuristic, an item x_i will be placed in the first bin (index-based order) where it fits. That is to say, if $V_j \geq a_i$ and $V_l < a_i$ for each l element of $\{1, \dots, j-1\}$, then S_j contains the item x_i. If no such j could be found, then x_i is added to the next empty bin [34]. According to [47], for a large number of bins, this heuristic uses at most 1.22 times the number of bins used by an optimal solution [47]. If the number of bins is less than 4, then the number of required bins for this heuristic is at most 1.5 times the number of bins used by an optimal solution [48].
- **Best-fit decreasing:** Through the use of the best-fit heuristic, at first all existence bins are considered and then the one with minimum remaining capacity after placing item x_i will be selected, i.e., if $L = \{l | V_l \geq a_i\}$ and $V_l \geq V_j \ \forall l \in L$, then $x_i \in S_j$. If no such j could be found, then item x_i is added to the next empty bin [34]. Similar to the first-fit heuristic, at most 1.22 times and 1.5 times the number of bins used by an optimal solution will be required by this heuristic for a large [47] and small [48] number of bins respectively. From an academic point of view, the best-fit heuristic is the best one in comparison with other bin-packing heuristics [49], because its deviation is usually little from an optimal solution (considering the number of required bins). However, in practice, this heuristic suffers the high overhead and consumes too much time to search the entire set of bins for a bin with smallest remaining capacity and still big enough for item x_i. Therefore, the first-fit decreasing is generally faster than the best-fit heuristic.
- **Worst-fit decreasing:** In contrast to the best-fit heuristic, the worst-fit heuristic puts the item x_i in the bin with largest free capacity, i.e., if $L = \{l | V_l \geq a_i\}$ and $V_l \leq V_j \ \forall l \in L$, then $x_i \in S_j$. Like two previous strategies, if no such j could be found, a newly-added bin is allocated to item x_i [34]. At most 1.25 times the number of bins used by an optimal solution will be required by this heuristic [47]. This signifies that, the worst-case performance of both first-fit and best-fit strategy is partly better than the worst-fit decreasing heuristic in term of storage utilization. However, this heuristic is proper for packing the bins with approximately the same weight or fill them with items of approximately the same value in particular when the number of bins is fixed.

Figure 2.2 shows a solution for each above-mentioned heuristic to pack 5 items $x_{i=1}^{5} = \{x_1, x_2, x_3, x_4, x_5\}$ in order of decreasing size like $a_{i=1}^{5} = \{8,6,3,3,1\}$ into bins of size 10. The example of Figure 2.2 (especially in the case of placing the last item x_5) implies that, the decision

[2] Initially, the value of j is equal to 1, i.e., there exist just one bin at the beginning, but this can be increased, whenever an extra empty bin is required.

for the bin to place the next item x_i in it, is depend on items' size a_i and the choice of bin-packing heuristic as well.

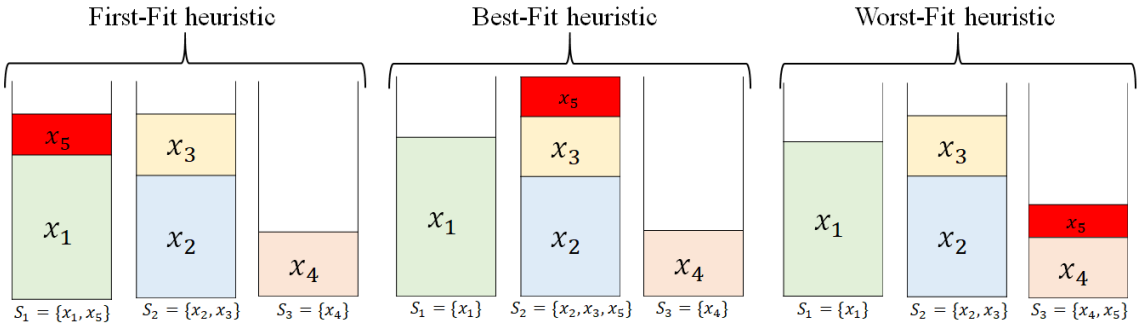


Figure 2.2: Illustration of the first-fit, best-fit and worst-fit decreasing bin-packing heuristics

Among the aforementioned heuristics, an interpreted version of worst-fit decreasing heuristic is used in the experimental phase of the thesis to partition the tasks on the processors of the real-time system. More details on this will be given in section 4.1.2.

2.3.2. Global Scheduling

Under global scheduling algorithms, a single system-level scheduler is supposed to schedule the tasks, and the multiprocessor system allows each task to run on any processor. Such multiprocessor systems use a single global queue storing ready tasks. At any point of time, among all ready tasks of the global queue, the scheduler selects at most m tasks, which have the highest priority, for execution on a multiprocessor system with m processors. Another property of the global scheduling is that, tasks can start their execution on one processor and finish it on another processor as well as be preempted on one processor and resumed on another processor, i.e., tasks can migrate among processors whenever it is required [44]. *Global EDF (G-EDF)* [50] is an example of a global scheduling algorithm. Although EDF [16] is an optimal scheduling algorithm under uniprocessor, G-EDF is not optimal for multiprocessor systems. Actually, it is not necessary for the global scheduling algorithms to be optimal, however the multiprocessor researchers have developed some optimal new multiprocessor scheduling algorithms. For instance, *proportionate fair (Pfair)* scheduling approaches are such optimal algorithms for multiprocessor systems [51] [19]. Nevertheless, the performance of this scheduling algorithm approaches is not very impressive because the number of preemptions and migrations might be increased significantly by using this particular class of scheduling algorithm, which causes high run-time overhead. However, there exists researches attempt to decrease this overhead in the multiprocessor scheduling algorithms; e.g., the work by Levin et al. [52].

Another popular approach of global scheduling algorithm is *Global Fixed Priority scheduling (G-FP)*. Under fixed-priority global scheduling, just the lowest-priority scheduled tasks will be affected through preemptions, which means that, the m^{th} highest-priority scheduled task $\tau_m$will be preempted by a just now released task τ_i if $Y(\tau_i) < Y(\tau_m)$. The most important beneficial of

global scheduling is that, the algorithmic capacity loss inherent in partitioned scheduling can be resolved somehow as a single, shared ready queue is used by all processors conceptually. Thus, the problem of task assignment is not the issue any more, which is the source of capacity loss under partitioned scheduling [34].
In this thesis global scheduling will not be considered because LITMUS^{RT} [6] [7] does not support G-FP scheduling currently. Therefore, for a comprehensive overview of the state-of-the-art in global fixed priority scheduling, the interested readers are referred to [19].

2.4. Synchronization Mechanism

In multiprocessors systems, tasks are usually not independent and they may share resources. This means that, in order to adapt the existing systems to be able to executed on a multi-core platform, synchronization protocols are required.
If all tasks in a task-set are independent from each other, the scheduler schedules always the *m* highest-priority tasks, i.e., whenever the scheduler assigns a processor to a task, the task can start and continue to execution until its completion without any preemption. But the task independence condition is not always true as in most real-time systems multiple tasks need to use the shared resources of the system at the same time. For instance, limited physical I/O facilities or program state such as queues, buffers, and other data structures might be shared among some tasks. Sometimes it is possible that tasks are independent at the application level, even though synchronization is still required, because in RTOS kernel there is a significant amount of shared state such as memory maps, device state, task information, and ready queues, so that the system should synchronize the parallel access to such shared resources in order to minimize the inconsistencies and conflicts. The locking approaches [10] can usually afford this goal. Other alternatives to achieve this goal are the non-blocking algorithms [35], however, in comparison with lock-based synchronization, in which the in-place updates are allowed, through the use of the non-blocking approaches, additional memory is usually required and also there exists significant copying or retry overheads, and just shared data structures (not devices) can use such approaches for synchronization as well [34]. With regard to synchronization, this thesis focuses only on locking approach. It means that, when there is dependency between tasks and an occupied shared resource is required by another task, then the resource cannot be assigned to the task and the task cannot complete its execution; in such situation, the state of the task is changed to *waiting* until the resource becomes available, even if this task is one of the *m* highest-priority tasks.
Indeed, in the environment of multiprocessor real-time system, the need of synchronization or rather locking protocols is appeared when some tasks located on multiple processors (not on a same processor) request for a shared resource at the same time; this situation is not the case in the existing uniprocessor synchronization techniques. There exist two categories for shared resources: *global* and *local* resources. The resource located on j^{th} processor called local resource if it can be accessed only through the tasks partitioned on processor P_j, else the resource called global. The advantage of the local resources is that, system can manage them optimally by

exploiting the uniprocessor locking protocols in each partition, i.e., local resources reduce to a simpler uniprocessor problem and multiprocessor locking protocols are used just for global resources. However, through the use of the global scheduling, since it is unknown in advance that each processor contains which set of tasks, all shared resources should be global [34]. In this thesis it is assumed that each resource r_q is a global resource.

When it is necessary to block a task, it can be done either by spinning or by being suspended. These two types of lock-based synchronization mechanisms will be described in the following subsections.

2.4.1. Spinning

Whenever an occupied shared resource is required by a task, the task cannot progress towards its completion, thus it must wait until the shared resource becomes accessible. One way to implement such waiting on a multiprocessor is *spinning*.

Under a spin-based protocol, the processor is kept by a task, which asks for a shared resource that is currently locked by another task, thus the task spins to the processor. Strictly speaking, in spin-based protocols, a simple delay loop is executed by the waiting task τ_i (i.e., the task performs busy-waiting or "spinning") until the access to the desired resource is achieved by the waiting task τ_i. In such situation, task τ_i is still scheduled during spinning, in this wise, the task does not suffer pi-blocking (see section 1.2.1). However, the spinning task causes a type of delay, named *s-blocking*, which is happened due to task's own busy-waiting, i.e., the perform of spinning increases task's execution time [34].

By using the spin-based locking protocols, when a task τ_i is waiting for a global resource or rather spins to the processor, the priority of the task will be increased temporally to a priority higher than its base priority ($Y(\tau_i)$). In other words, it is obvious that, when a higher-priority task is released, the lower-priority task τ_i, which holds already the higher-priority task's desired resource, cannot continue its execution, while it is keeping its base priority ($Y(\tau_i)$). According to the priority inversion phenomena (see section 1.2.1), in order to guarantee the progress of the resource-holding task τ_i with the lower priority in such situation, the task's base-priority that is assigned at the beginning of the scheduling by the scheduling policy, can be raised temporarily by a locking protocol to the *effective priority* of task τ_i, such that the locking protocol calculates the effective priority of task τ_i according to the resource usage of τ_i at time t and is denoted like $y(\tau_i, t)$. The equality of the base priority and the effective priority of a task τ_i at time t, i.e. $y(\tau_i, t) = Y(\tau_i)$, intimates that task τ_i does not occupy any shared resource at time t [34].

Spinning can be both preemptive and non-preemptive. Both these types of spinning mechanism are outlined below.

- **Non-preemptive spinning:** Traditional spin-based protocols raise the priority of a task to the highest priority on the related processor, hence it makes the task non-preemptive. In other words, a place-holder in the global queue of the intended resource is provided for the task and the task busy-waits till acquiring the resource. Afterwards, as soon as the

resource is available, i.e., not locked by any other task, and also the task has placed at the head of the queue, the resource can be locked by the task [10]. Under this type of spinning, the newly-released tasks with higher priority might suffer from priority inversion or rather pi-blocking (see section 1.2.1), which is caused by the lower-priority spinning task. In contrast, the spinning task as well as the other tasks with the priority lower than the priority of the spinning task suffer from s-blocking. It means that, when a non-preemptive spinning task τ_i cannot be preempted by a higher-priority task τ_j, then τ_j incurs pi-blocking (but not s-blocking), whereas τ_i incurs s-blocking (but not pi-blocking) [34]. Figure 2.3 illustrates the difference between pi-blocking and s-blocking through an example of three tasks scheduled under P-FP and partitioned on two processors in this way that the first processor P_1 contains task τ_1 and the second processer P_2 includes tasks τ_2 and τ_3 with the condition $Y(\tau_2) > Y(\tau_3)$. In this example task τ_3 becomes a non-preemptive spinning task at time point 3, due to waiting for a shared resource, which is already in use by task τ_1. Thus, task τ_3 does not incur pi-blocking, since no priority is inverted (it is still scheduled), but the response time of task τ_3 will be increased because of the s-blocking. In contrast, task τ_2, which has a priority higher than the priority of task τ_3 and is released after task τ_3, suffers from the delay caused by the pi-blocking until the time point 7, when task τ_3 becomes preemptable again.

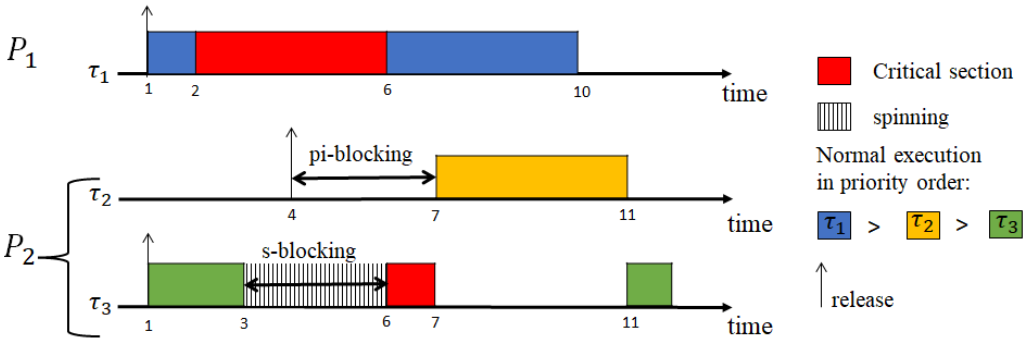


Figure 2.3: The difference between pi-blocking and s-blocking

- **Preemptive spinning:** Another type of spin-based protocols are preemptive spin-based protocols. Under a preemptive spin-based protocol, in which tasks spin preemptively, which they do in the considered spin-based protocol in this thesis and will be discussed in section 2.5.4, a local task can preempt spinning tasks. In other words, whenever a task spins, for which a request in the global resource queue is considered, it may be preempted on its processor by the other local tasks. To manage the request (ask for a resource) of the spinning task upon its preemption, there are three policies: (i) *de-queuing* the request from the global resource [41], (ii) *skipping* the request [53] and (iii) *the classic policy* (neither dequeuing nor skipping). The **de-queuing policy** removes (dequeues) the request of the task from the global resource queue upon preemption of the spinning task and then enqueues the request to the global resource queue as soon as the preempted task is allowed to spin again. Under this policy, the preempted task may suffer from extra waiting-time

due to the later requests of the additional tasks for the same global resource. Wieder and Brandenburg in [41] have investigated the preemptive spin-based protocols with FIFO-based queues, using the de-queueing policy upon preemptions. The **skipping policy** does not remove the request of the task from the global resource queue but selects the next eligible task in the queue instead of the preempted task located at the head of the queue. Similar to the de-queuing policy, under the skipping policy, the preempted task may suffer from extra waiting-time due to the later requests of the additional tasks for the same global resource. In [53] the preemptive spin-based protocols based on a skipping policy is considered by Takada and Sakamura. The **classic policy** (which is used in this thesis) neither dequeues nor skips a task upon preemption. Instead the task remains at the head of the queue and can gain the access to the global resource as soon as the resource becomes available [10].

2.4.2. Suspending

Another way to implement the waiting situation, in which a task requesting a locked resource must wait until the shared resource becomes available, on a multiprocessor is *suspending*.
In suspension-based protocols, once a task is suspended, the processor occupied by the waiting task will be released, so the other tasks have a chance to execute on the processor and the waiting job can attain the processor when the resource becomes available. In other words, if a task requests a resource that is shared across processors and in the meanwhile, another task locked the resource, then the task becomes suspend.
Under suspension-based locking protocols, whenever a task is waiting for a resource and therefore is suspend and releases the processor, a placeholder in the queue related to the resource will be assigned to the task and the task remain there until it acquires the resource. In such manner, the other tasks related to this processor can be executed. If the task is located at the head of the resource queue and the resource is not occupied, then the task is eligible for accessing the resource and the resource is ready to become lock. At this point, the resource locking is performed according to the locking protocol policy, which is in use by the real-time system, and the raising of the task priority depends on the policy as well [10].
In recent years, many suspension-based locking protocols are presented in domain of multiprocessor real-time systems. Nonetheless, it is still hard to distinguish which one of these protocols is the best one, because such decision varies according to the different settings of the real-time systems. As a result, neither suspension-based locking protocol is completely preferable to the others. For the sake of consistency, only the suspension-based locking protocols relevant to this work are investigated in section 2.5.

2.5. Synchronization Protocols

As argued in previous section, the handling of the resource sharing is one of the essential components in RTOS. For this purpose, resource synchronization protocols are required. Resource sharing becomes more complex when considering multiprocessor platforms instead of uniprocessor platforms, because in multiprocessor systems the task-suspending and task-spinning may come into play. Therefore, before starting the experimental evaluations, it is required to determine some multiprocessor locking protocols among all protocols implemented in the used test-bed, i.e. LITMUS^{RT} [6] [7], which are appropriate for this work. The most appropriate protocols for this work are three suspension-based locking protocols *Distributed Priority Ceiling Protocol (DPCP)* [3] and *Distributed FIFO Locking Protocol (DFLP)* [54] for resource distributed-based partitioned scheduling as well as *Multiprocessor Priority Ceiling Protocol (MPCP)* [4] for global resource partitioned scheduling, and a spin-based locking protocol *Multiprocessor Priority Ceiling Protocol Virtual Spinning (MPCP-VS)* [55].
In the following subsections, firstly two resource distributed- as well as suspension-based protocols, i.e. DFLP and DPCP, are introduced. Furthermore, the other suspension-based protocol MPCP is explained and finally, the spin-based protocol MPCP-VS is reviewed.

2.5.1. Distributed Priority Ceiling Protocol

The Distributed Priority Ceiling Protocol (DPCP) is a suspension-based synchronization locking protocol, which is developed for distributed-based multiprocessor systems with shared resources. Rajkumar et al. were the first, who introduced this protocol in [3] and later the DPCP was described in greater detail in [56].
DPCP has been proposed for fixed-priority partitioned rate monotonic scheduling (RM) [16] and its primary responsibility is to pass message between processors. To this end, the *remote procedure calls* (RPC) are used by the protocol. Through the use of the DPCP, non-critical sections of a task are executed on the same processor assigned to the task, but if a resource required by the task is located on a processor other than the task's allocated processor, then the tasks executes its related critical section on the other processor. By exploiting the DPCP, the priority of the task becomes temporary the highest priority among all priorities in the system, while it is executing its critical section on a processor other than its assigned processor [10].
Since the DPCP is designed for distributed-memory multiprocessors, by the use of this protocol, it is assumed that, each resource is local to a specific processor and it can be accessed only from that processor. In this work it is assumed that, r_q is assigned to or rather accessible from processor P_j according to the resource assignment rule (1). Due to the use of RPCs by the DPCP, the remote procedure call model is implemented by this protocol. Based on this model, resources are accessible by a task indirectly, i.e., a request $R_{i,q,v}$ (i.e., task τ_i requests resource r_q for the v^{th} time) for a resource located on processor P_j is submitted at first to an *agent*. Actually, a local agent $A_{j,i}$ is responsible to perform requests of task τ_i related to the resources assigned to

processor P_j. In other words, if $r_q = j$, i.e., resource r_q is local to processor P_j, then whenever task τ_i needs to access resource r_q, it must delegate its request $R_{i,q,v}$ to agent $A_{j,i}$. Each agent will be active upon receiving a request. Whenever an agent $A_{j,i}$ is active, task τ_i becomes suspended and remains in this state until the agent conducts request $R_{i,q,v}$ completely. Recall that, the priority boosting phenomena (see section 1.3) occurs during the execution of requests for global resources. Additionally, the effective priority of the task (see section 2.4.1) will not be changed during the execution of requests for global resources, because the requests are not carried out by the tasks themselves, rather by a local agent. An agent $A_{j,i}$ has been assigned a base priority according to the formula: $Y(A_{j,i}) = i - N$, where N is the number of tasks in a task-set. Such assignment formula guarantees that, even the lowest priority agent has a higher priority than the task with highest-priority. The resource access of each agent is conducted according to the rules of the PCP introduced by Sha et. al. in [1], but in DPCP there is a difference, which is that the priority ceiling of global resources is defined with respect to agent priorities instead of the task priority. In this regard, through the use of the PCP it is still guaranteed that, whenever some requests for a resource is submitted by multiple tasks at the same time, those requests are performed in order of the issuing tasks' base priorities since they are reflected in the agents' base priorities [34].

An example for resource sharing under the DPCP is illustrated in Figure 2.4, which is according to an implicit-deadline periodic task-set presented in Table 2.1. This example contains four tasks partitioned over two processors and also includes two shared resources local to processor 1. Thus, four agents are needed, which reside on processor 1 too. Agent $A_{1,3}$ becomes active at time 1 due to request $R_{3,1,1}$, i.e., task τ_3 submits its first request for resource r_1 on its local agent $A_{1,3}$. Therefore, task τ_3 enters to the suspension state until time 5, at which agent $A_{1,3}$ has completed request $R_{3,1,1}$. Shortly after the activation of $A_{1,3}$, $A_{1,4}$ is invoked at time 2 due to request $R_{4,1,1}$, but cannot start to perform $R_{4,1,1}$, because $A_{1,3}$ has a higher priority than $A_{1,4}$. However, in the case of agent $A_{1,2}$ activated by $R_{2,1,1}$ at time 4, $A_{1,2}$ is eligible to try for locking r_1, because its priority is higher than $A_{1,3}$. Nonetheless, this effort will fail, since r_1 is still occupied by $A_{1,3}$. Therefore, agent $A_{1,2}$ suspends. Note that, although the highest priority task τ_1 is released before $A_{1,2}$, it cannot start to run until time 7, because the priority of all active agents at the release time of τ_1 exceeds the priority of τ_1. When r_1 becomes accessible again (i.e., $A_{1,3}$ released r_1), although $A_{1,2}$ was activated later than $A_{1,4}$, $A_{1,2}$ obtains r_1 because of its higher priority. When $A_{1,2}$ completed $R_{2,1,1}$, it is activated again due to another request $R_{2,2,1}$. However, $A_{1,2}$ cannot run until time 10, while another agent $A_{1,1}$ with higher priority is executing (on the same processor). At time 10 both τ_1 and $A_{1,2}$ are ready to run on processor 1, but only one of them can be scheduled at the same time. In this case, $A_{1,2}$ starts its execution and τ_1 is preempted, because the priority of $A_{1,2}$ exceeds the priority of τ_1. As a result, the highest priority task of this example has to tolerate pi-blocking (see section 1.2.1) for a total of six-time units (during [2,7] and [10,11]) before it finishes its execution [34].

Note that, tasks do not suffer any s-blocking (see section 2.4.1) under the DPCP, because there is neither task-spinning nor agent-spinning.

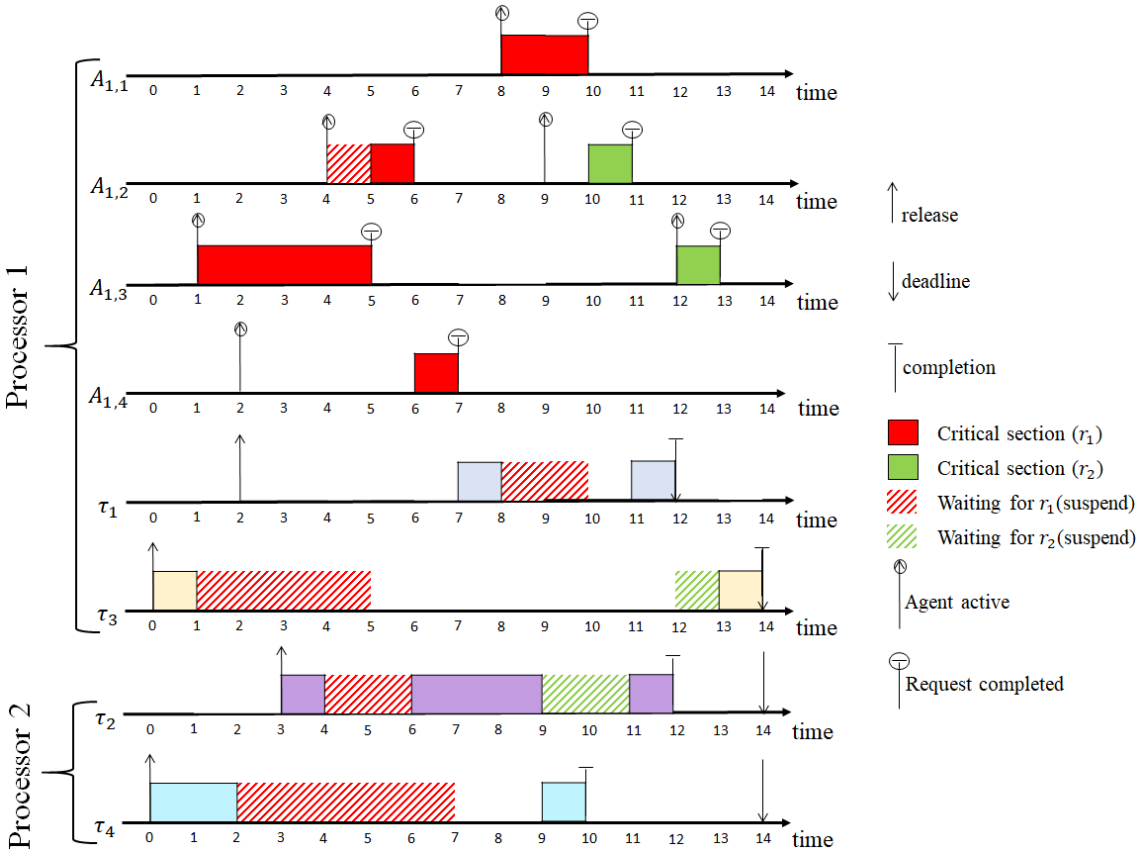


Figure 2.4: Example for DPCP (redrawn from [34])

	Period	Release time	Length of 1th non-cs	Length of 2th non-cs	Length of 3th non-cs	Length of cs (resource 1)	Length of cs (resource 2)
τ_1	10	2	1	1	-	2	-
τ_2	11	3	1	3	1	1	1
τ_3	14	0	1	0	1	4	1
τ_4	14	0	2	1	-	1	-

Table 2.1: Sample of a task-set related to the examples showed in Figure 2.4, Figure 2.6 and Figure 2.7

2.5.2. Distributed FIFO Locking Protocol

The Distributed FIFO Locking Protocol (DFLP) is another suspension-based synchronization locking protocol for distributed-based multiprocessor systems with shared resources, which is

based on simple FIFO queues in order to avoid starvation [54]. Similar to the DPCP [3], resources are assigned to the processors and priority-boosted agents mediate resource accesses, but unlike the DPCP, the PCP is not used by the DFLP. Under the DFLP, each request $R_{i,q,v}$ (i.e., task τ_i requests resource r_q for the v^{th} time) is ordered with a per-resource FIFO queue denoted F_q. Each agent A_q serves all requests related to resource r_q in FIFO order. When a request $R_{i,q,v}$ is issued, task τ_i suspends and $R_{i,q,v}$ enters to F_q. If $R_{i,q,v}$ locates on the head of F_q, then agent A_q becomes active to process $R_{i,q,v}$ and remains active as long as there is a request in F_q. After completion of request $R_{i,q,v}$, it is removed from F_q and task τ_i resumes its execution. Agents are scheduled preemptively in order of increasing lock-request time with respect to each agent's request currently being processed, i.e., priority of an agent processing an earlier-issued request is higher than priority of another agent serving a later-issued request [54].

Figure 2.5 shows an example for resource sharing under the DFLP. This example is according to an implicit-deadline periodic task-set presented in Table 2.2. For similarity, only the highest priority task is considered for each processor. This example contains two shared resources local to processor 4. Thus, two agents A_1 and A_2 are needed, which also reside on processor 4. Agent A_2 becomes active due to a request $R_{3,2,1}$ at time 1 and therefore task τ_4 is preempted immediately. Recall that, agents are subject to priority boosting. At time 2, agent A_1 is activated by an issued request $R_{2,1,1}$. Since the priority of A_2 is higher than the priority of A_1 at that time (remember that under DFLP, priority of an agent processing an earlier-issued request is higher than the priority of another agent serving a later-issued request), A_2 will not be preempted by A_1. Another request for resource r_1 is issued at time 3 by task τ_1, thus $R_{1,1,1}$ is appended to F_1. After finishing the process of $R_{3,2,1}$ by agent A_2, A_1 is scheduled and starts to serve first $R_{2,1,1}$ and next $R_{1,1,1}$ due to the use of FIFO queues by DFLP. Since both agents are local to processor 4, task τ_4 has to tolerate pi-blocking throughout [1,10], while A_1 and A_2 serve the requests [57].

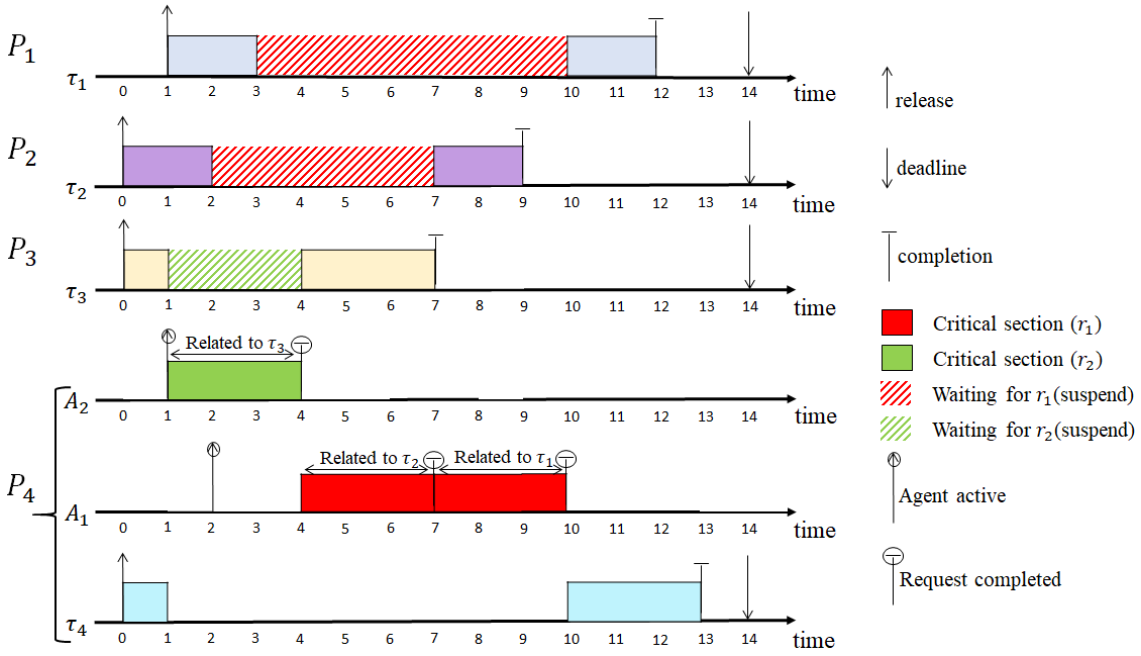


Figure 2.5: Example for DFLP (redrawn from [57])

	Period	release	1th non-cs	2th non-cs	cs (resource 1)	cs (resource 2)
τ_1	13	1	2	2	3	-
τ_2	14	0	2	2	3	-
τ_3	14	0	1	3	-	3
τ_4	14	0	4	-	-	-

Table 2.2: Sample of a task-set related to the example showed in Figure 2.5

2.5.3. Multiprocessor Priority Ceiling Protocol

The Multiprocessor Priority Ceiling Protocol (MPCP) is another suspension-based synchronization protocol, which is proposed by Rajkumar [4] for shared-memory platforms. Similar to the DPCP synchronization protocol, the MPCP is also another extension of PCP [1] and so it raises the priority of a task higher than any priority in the system, while the task runs its global critical sections. However, when a task with higher priority enters its global critical section, it preempts the running lower priority task (it does not matter if the lower priority task executes its critical or non-critical section). In this case, a further incrementation of the boosted priority will be applied to the global ceiling of the resource. The MPCP synchronization protocol is an appropriate protocol for partitioned fixed-priority (P-FP) rate monotonic scheduling (RM) [16] and the global resources. Moreover, the implementation of the MPCP is more efficient in comparison with the DPCP protocol, because the DPCP suffers overhead of remote execution of global critical sections and communication delays [10].

Unlike the DPCP, under MPCP it is not required to use RPCs as well as local agents, because in MPCP protocol it is defined that tasks can access their intended resources from any processor. In fact, the requests of each task are processed by the task itself on the same processor assigned to the task. In particular, while the task holds a resource r_q, the priority of the task during the task's request execution (critical section) is boosted to the highest priority of any remote task that shares the resource r_q. Since it is not necessary for the effective priority of tasks (see section 2.4.1) to be unique, the existence of two or more tasks with highest effective priority at the same time is possible. In such situation, the earlier-boosted task is the one, which has precedence to avoid unnecessary preemptions. If multiple tasks request a resource r_q at the same time, r_q assigned to the tasks in order of decreasing base priority. Under MPCP, no priority inheritance is used for global resources. Nonetheless, the executions of critical sections that delay higher-priority tasks, are guaranteed to be completed rapidly through the use of priority boosting [34].

An example for resource sharing under the MPCP is illustrated in Figure 2.6, which is according to the implicit-deadline periodic task-set presented in Table 2.1. The depicted scenario is the same as previously discussed in the example of DPCP showed in Figure 2.4. However, in this example there is no agent in use, because in contrast to the distributed-based locking protocols, e.g. DPCP and DFLP, the MPCP does not require local agent any more since global resources are accessible by tasks directly. In the example presented in Figure 2.6, task τ_4 requests resource r_1 at time 2, but cannot access the resource since r_1 has been already hold by τ_3. Similarly, τ_2 requests also r_1 at time 4 but has to suspend like task τ_4, until it can gain r_1. In this example, task τ_1 has the highest base-priority, nevertheless, it cannot be scheduled until time 5, at which the priority of τ_3 returns to the base-priority of the task. At time 6, again r_1 is requested by the other task τ_1, thus there are multiple requests for r_1 at this moment (recall that, r_1 was being requested by τ_4). Although the request of task τ_4 for r_1 was issued much earlier at time 2, $R_{1,1,1}$ will be satisfied before $R_{4,1,1}$ since tasks gain a resource in order of decreasing priority. Therefore, τ_4 remains suspended until time 8 and then preempts τ_2 immediately due to holding a global resource [34].

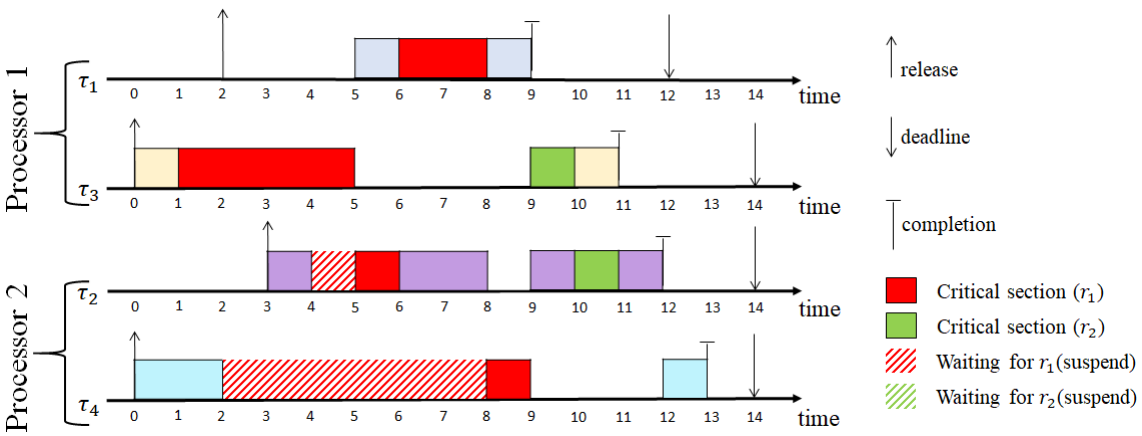


Figure 2.6: Example for MPCP (redrawn from [34])

2.5.4. Multiprocessor Priority Ceiling Protocol Virtual Spinning

The Multiprocessor Priority Ceiling Protocol Virtual Spinning (MPCP-VS) is a variant of the MPCP based on a request rule named "virtual spinning" and proposed by Lakshmanan et al. [55]. MPCP-VS is actually one of the preemptive spin-based synchronization protocols designed for multiprocessor systems.

As the name of this protocol implies, tasks are spinning virtually or rather preemptively instead of spinning non-preemptively to the processor while waiting for a global resource to become available. In other words, under MPCP-VC, the other local (lower-priority) tasks can be scheduled and allowed to run their non-critical sections during the waiting time of the higher priority task. In fact, the task, which is waiting for a global resource, suspends under the MPCP [4] and thus lets the other local tasks execute. However, these running tasks will be blocked whenever they issue a request and try to lock a resource. They remain blocked as long as the suspended task is spinning to the processor virtually. The word "spinning" used for the name of this protocol denotes the busy-waiting and therefore there is no context switch for the spin task [34].

An example for resource sharing under the MPCP-VS is showed in Figure 2.7, which is according to the implicit-deadline periodic task-set presented in Table 2.1. The depicted scenario is the same as the one previously discussed in the example of DPCP illustrated in Figure 2.4. In the example of the MPCP-VS, the virtual spinning happens on processor 2. Task τ_4 requests resource r_1 at time 2 and therefore it has to wait for accessing r_1 since the resource is being used by the other task τ_3 at this moment. Since τ_4 spins preemptively, the other task τ_2 located on the same processor has a chance to run. However, τ_2 can execute until time 4, when it requests resource r_1. In such situation, although τ_2 has higher base-priority than τ_4, it must suspend and is not permitted to issue its request due to the uncompleted request of task τ_4. τ_4 resumes its execution at time 6, when it finishes the execution of its critical section and releases r_1 [34]. However, τ_2 cannot gain r_1 yet, since r_1 is held by τ_1, which has the highest priority in this example.

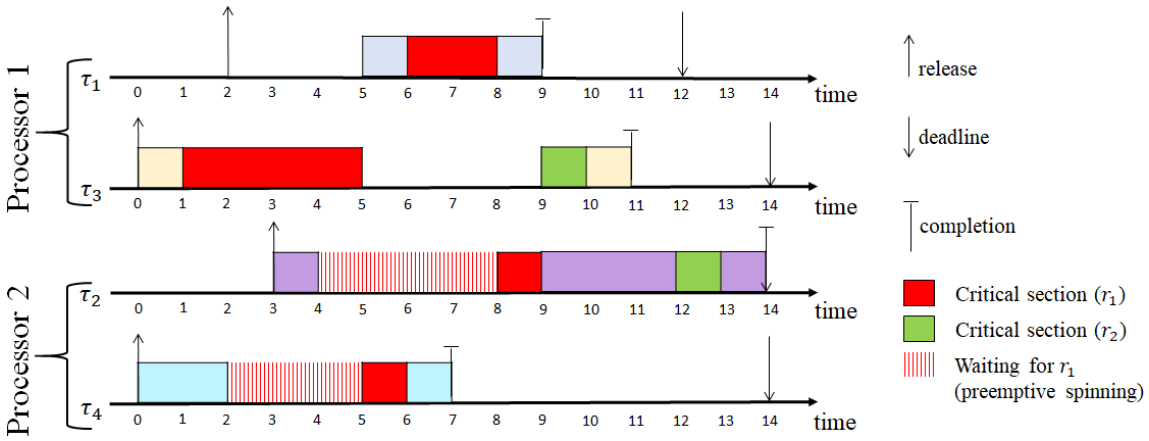


Figure 2.7: Example for MPCP-VS (redrawn from [34])

3 Test-Bed

To progress the work of this thesis, it is required to choose a proper experimental platform, which can support the necessary scheduling and resource synchronization protocols. Additionally, to have an evaluation with a high accuracy it is important that, the test-bed provides a circumstance to trace the features and parameters of the examination. As a result, among all available real-time system test-beds such as RTEMS [58], QNX Neutrino [59] and so on, LITMUS^{RT} [6] [7] is selected to use as the experiment platform for the thesis. There are several reasons for choosing this test-bed: First of all, LITMUS^{RT} is a well-established experimental open source code and evaluation platform in the real-time research community, which is downloadable for free from its official website: `https://www.litmus-rt.org`. Second, it provides a plugin interface, to activate and to change different scheduling policy and locking protocols dynamically at run time, which is not supported by the Linux kernel. Moreover, by having this test-bed, it is not required to implement the protocols interested for the thesis as it supports a variety of scheduling and resource synchronization protocols.
The rest of this chapter introduces the used hardware and the used test-bed LITMUS^{RT}.

3.1. Hardware Platform

The hardware platform used during the experiments of this master thesis is a cache-coherent SMP, which contains two 64-bit Intel Xeon Processor E5-2650Lv4 with the speed of running at 1.7 GHz. The cache capacity and the main memory capacity of the hardware are 35 MB and 64 GB respectively.

3.2. Basics of LITMUS^{RT}

Since the central work of this thesis is the comparison of implemented multiprocessor synchronization mechanisms, an actual RTOS should be used. In this regard, an extension of Linux called **LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime systems (LITMUS^{RT}) [6] [7] is used.
LITMUS^{RT} is a native real-time Linux version and a widely used open-source UNIX-like kernel that is focused on extending the stock Linux kernel with multiprocessor real-time scheduling policies and locking protocols [34]. Strictly speaking, LITMUS^{RT} is one of the common useful experimental platforms for applied real-time systems research as it provides abstractions and interfaces within the kernel that simplify the prototyping of multiprocessor real-time scheduling and synchronization algorithms.

LITMUS^{RT} was originally launched at the University of North Carolina at Chapel Hill under the direction of James H. Anderson[3] and has benefited from contributions by a number of researchers over the years. Now, the maintainer and main developer behind LITMUS^{RT} is Björn Brandenburg[4] of the Max Planck Institute for Software Systems (MPI-SWS). The first public version of LITMUS^{RT}, named LITMUS^{RT} 2007.1 and released in May 2007[5], was based on Linux 2.6.20 and described in detail in [60]. The version of LITMUS^{RT} used in this thesis is 2017.1 (the latest version) and is based on Linux 4.9.30.

The main reason that makes Linux an unsuitable experimental platform is that, Linux does not support adding, removing, or switching scheduling classes at runtime. From a design point of view, the Linux scheduling framework is structured as a hierarchy of scheduling classes, i.e., the policy used for a particular process type is encapsulated by each scheduling class. In other words, whenever a scheduling decision is made, each scheduling class is polled in the hierarchy in top-to-bottom order until a pending task is found. In such wise, a new scheduling class at the top of the hierarchy is introduced by LITMUS^{RT} such that all task except the LITMUS^{RT} tasks can be only scheduled when no real-time workload is present. In fact, instead of implementation of a particular policy by the LITMUS^{RT} (unlike the regular Linux scheduling classes), all scheduling decisions are deferred to the active real-time scheduler plugin. This allows developer to program scheduler plugins through a simple, real-time-specific interface that changes only rarely instead of interfacing with the full Linux kernel, which is more complex and changes frequently between versions. In contrast to the standard Linux, global scheduling policies can be supported very well in LITMUS^{RT} in this way that the Linux system call interface is augmented further with additional real-time-specific system calls. Notably, some additional calls for real-time task and job control (e.g., to configure task parameters, to wait for the next job release, to obtain the job sequence number, etc.) and for invoking real-time locking protocols (e.g., to acquire resource handles, to lock resources, etc.) are determined. As an additional benefit of LITMUS^{RT}, the specific support of migration for global and clustered scheduler plugins is factored out into a common "migration path" in the LITMUS^{RT} scheduling class [34].

Synchronous releases in LITMUS^{RT}. To gain reasonable results from the examinations, we have to consider the worst-cases as far as it is possible. To this end, we need to take the critical instant of each task into account, that is the time at which the release of the task will yield the largest response time and it occurs when the task is released simultaneously with higher priority tasks. So, the task-set should be able to release its tasks synchronously. The synchronously released of a task-set happens when all tasks in the task-set release their first job exactly at the same time (which is, by default, time 0). Although it is approximately possible to use standard Linux system calls for a synchronous release, the achievement of high precision with this approach is difficult. Another remarkable thing during task-set setup is that the task's release must be accomplished

[3] `http://www.cs.unc.edu/~anderson/`

[4] `http://www.mpi-sws.org/~bbb`

[5] The versioning template is *yearOfRelease.perYearSequenceNumber*. For example, version 2007.2 denotes the second release of LITMUS^{RT} in 2007 and version 2008.1 denotes the first release in 2008.

after the finishing of all processes' initialization phase (not before that). In such manner, it is important to detect accurately if all processes are ready to start the real-time execution. To this end, LITMUS^{RT} provides an additional synchronous release API. After the end of a process's initialization phase and its transmission into real-time mode, a system call, deemed `wait_for_ts_release()` can be invoked by the process to participate in a synchronous release. By calling this method, the process will be suspended until the synchronous release occurs. Essentially, the number of waiting real-time processes are exported as a virtual file in the proc file system and they are monitored by a non-real-time process. Whenever all real-time processes have "checked in", a system call, named `release_at()` is invoked, which causes a synchronous task-set release. The use of this real-time plugin's method ensures that all tasks are released at the same time [34].

The details of the design and the implementation of LITMUS^{RT} is somewhat tedious and is not the subject of this thesis, thus it is omitted here in the interest of brevity. The interested reader is referred to [34] instead.

In the following, the installation of LITMUS^{RT} and some necessary modifications and verifications related to the LITMUS^{RT} as well as some procedures of parameter measuring through LITMUS^{RT} will be described.

3.2.1. Installation

In order to work with LITMUS^{RT} it is generally required to compile the entire LITMUS^{RT} kernel from the source, and then installing *liblitmus* and *Feather-Trace*. Although, all of the necessary compilation and installation guide can be found on the official website of LITMUS^{RT}[6], there exist some points to be noticed during the installation of LITMUS^{RT} on the Linux system (Ubuntu 16.04 is used in this thesis).

First of all, since the tracing of feature for each protocol is needed to evaluate the examinations, it is recommended to install the Ubuntu into a real computer rather than a virtual machine, which is not transparent with regard to timing. In fact, providing the high accuracy of the timing is very important in this thesis, because the investigated protocols are used for real-time system and sensitive to the timing.

Second, in order to do development and use the trace tools of *Feature-Trace-Tool* (see section 3.2.5), the option "`TRACE() debugging`" should be enabled during the configuration of the kernel. However, it is recommended to disable it whenever it does not need to debug, because it is a high-overhead debug tracing tool.

3.2.2. Modification

Under LITMUS^{RT} kernel there is a useful tasks allocation tool called *rtspin*. This tool can be found under the userspace portion of LITMUS^{RT} named *liblitmus*, which includes both libraries for

[6] `https://www.litmus-rt.org/installation.html#adjustingconfigurationoptionsforlitmusrt`

writing real-time tasks and utilities such as *setsched* for managing the system's schedulers dynamically. To meet the requirements of the thesis examinations, it is essential to modify the tasks allocation tool at first. By exploiting the original task allocation tool, just one critical section can be set for each task and also it is not possible to specify the position of that critical section (it is set randomly). Further, the critical section cannot be executed on a processor differ from the one assigned to the task. This is to say that, the distributed-based protocols' behavior cannot be simulated. In order to dominate all restrictions mentioned above, the pattern of task should be customized according to the requirements of the thesis examinations. To this end, the original task allocation tool is modified in this way that for each task the number of critical sections inside the task and the length of each section (both critical and non-critical sections) is definable as well as the resource id, which will be used by the critical section. Moreover, the modified version of the tasks allocation tool, *rtspin*, can support the distributed-based protocols as it is possible to define migration option for each task. So, a comprehensive example of the new task allocation format is as follow:

```
./rtspin -p 1 -q 9 -X DPCP -x 3:5:7:10 -L 2:1:2 -M 2:-1:2 -Q 2:3:4 -o 2
-w 25 130 5 &
```

Each parameter of this new structure is defined in the Table 3.1.

Parameter	Description
-p	Processor id (number), which the task is assigned to initially
-q	Task's priority (highest = 1, lowest = 511)
-X	locking protocol, by which the task access to the assigned shared resources is protected
-x	Number and length (in milli seconds) of the non-critical sections of the task (the first part, before the first colon indicates the number of non-critical sections and the length of the sections are divided by further colons)
-L	Number and length (in milli seconds) of the critical sections of the task (the first part, before the first colon indicates the number of critical sections and the length of the sections are divided by further colons)
-M	Number of migrations and the processor ids, the task migrates to in order to access the distributed shared resources assigned to that processors (the first part, before the first colon indicates the number of migrations and the processor ids are divided by further colons) Note: -1 means no migration is needed (i.e., both task and shared resource are assigned to the same processor), other number indicates the processor id
-Q	Number of the shared resources used by the task and their ids (the first part, before the first colon indicates the number of used shared resources and their ids are divided by further colons) Note: each critical section associated exact to one resource (nested critical section is beyond the scope of this thesis)
-o	The release time of the task in milli seconds (also known as phase). Its value is zero by default
-w	wait for a synchronous release
The last three numbers	They indicate the WCET (in milli seconds), period of the task (in milli seconds) and the whole run time (in seconds) respectively.
&	By using the trailing "&" at the end of each task format, the process started in the background and it is useful for scripting the creation of multiple waiting tasks.

Table 3.1: Definition of parameters of the customized tasks allocation tool, rtspin

Following the manual described in the Table 3.1, the sample task allocation mentioned before can be explained as follows:

The sample task with priority 9 is assigned to the processor with id 1 and will be released at 2ms. The task contains three non-critical sections and two critical sections. Each critical section is executed after the end of the execution of a non-critical section. In such manner, the sample task runs first a non-critical section for 5ms, second a critical section for 1ms, third again another non-critical section for 7ms, after that another critical section for 2ms and at the end another non-critical section for 10ms. Since a distributed locking protocol, DPCP, is in use to protect the task's

accesses to the assigned shared resources, the destination processor should be determined, where the task must migrate to once it runs a critical-section; thus, the task does not need to migrate to any processor for executing its first critical section as the intended shared resource is located on the same processor that the task is local to. However, the task must run its second critical section on the processor with id 2. The whole execution time of this task is 25ms, the period is 130ms and the run time is 5s.

3.2.3. Task States

To deal with the locking protocols implemented on the test-bed LITMUS^{RT} [6] [7], different states of a task are needed to define at first.

Whenever a task is released, it is called an **active** task. Upon the activation of a task, it can transit from one state to another until it is complete and **terminated**. These different task states and the task transitions are illustrated in Figure 3.1.

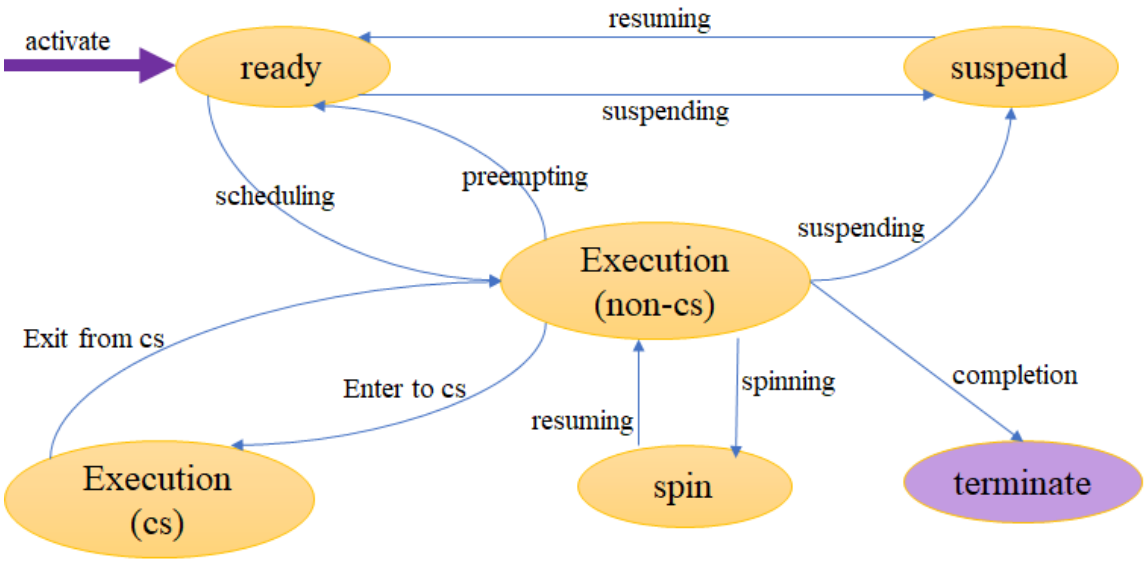


Figure 3.1: Task-state transition diagram

As it depicted in Figure 3.1, the state of an active task can be one of the following states: A task waiting for the processor is called a **ready** task. A ready task is available for **execution**. A scheduled task executes either its non-critical section or its critical section at any time. While a task runs its non-critical sections, it can be preempted, otherwise, depending on the locking protocols, the task cannot be sometimes preempted during its critical section. It is assumed that, pending tasks are ready to execute unless a locking protocol forces them to wait for accessing a resource. Tasks can wait to acquire a resource either as **spinning** or **suspending**, which is depended on the type of the locking protocol that controls the resource accesses. However, tasks cannot be suspended or spinning while being non-preemptable. The task resumption can occur after the task was in the waiting state, i.e. spinning or suspending [34].

3.2.4. Verifications

Before starting the real testing to compare the synchronization mechanisms, some verification pertained to the interested locking protocols implemented on the test-bed LITMUS^{RT} [6] [7] should be done. For the following two reasons, some simple test-cases are designed to test the correctness of these protocols' behavior. For one thing, it is needed to verify if these protocols operate as expected under using of the modified rtspin tool (the tasks allocation tool) described in section 3.2.2. For another, conduction of such sample experiments leads to be more familiar with the way the test-bed works. After running of each sample task-set on LITMUS^{RT} [6] [7], the results will be depicted through `st-draw()` tool.

In the following, all verification conducted for four locking protocols related to the work of this thesis are explained.

Verification for DPCP. The sample task-set to verify the behavior of the DPCP [3] is shown as follow, which is designed based on the same task-set presented in Table 2.1. The reason for selecting such task-set is that such example is simple and still comprehensive.

```
./rtspin -p 0 -q 1 -X DPCP -x 2:1:1   -L 1:2   -M 1:-1    -Q 1:1   -o 2
-w 4 10 2 &
./rtspin -p 1 -q 2 -X DPCP -x 3:1:3:1 -L 2:1:1 -M 2:0:0   -Q 2:1:2 -o 3
-w 7 11 2 &
./rtspin -p 0 -q 3 -X DPCP -x 3:1:0:1 -L 2:4:1 -M 2:-1:-1 -Q 2:1:2
-w 7 14 2 &
./rtspin -p 1 -q 4 -X DPCP -x 2:2:1   -L 1:1   -M 1:0     -Q 1:1
-w 4 14 2 &
```

The result of the task-set schedule is presented in Figure 3.2.

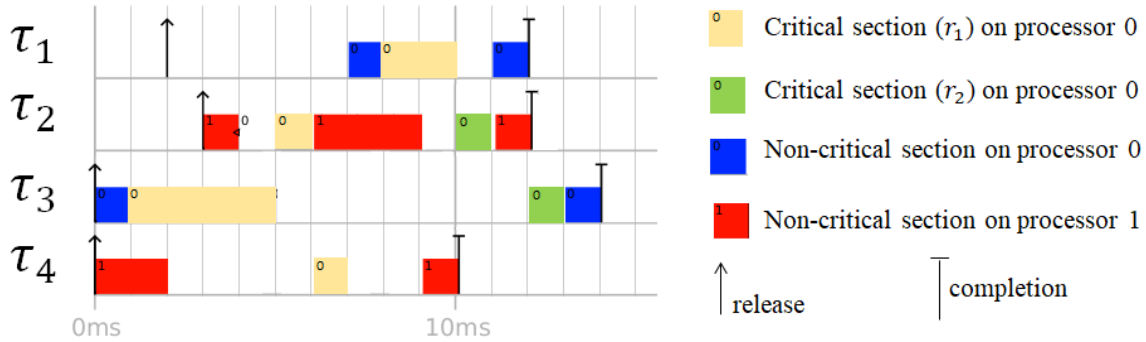


Figure 3.2: DPCP's behavior implemented on LITMUS^{RT}

After comparing Figure 3.2 with Figure 2.4, it will be clear that, the DPCP implemented under LITMUS^{RT} [6] [7] acts as it is expected (i.e., it performs exact same as the example corresponding to Figure 2.4). As Figure 3.2 shows, the task migration is implemented correct, i.e., they migrate at correct times to the correct processors. Additionally, the result confirm that the priority ceiling

policy is applied with respect to agent priorities, as it is defined under DPCP. Thus, the DPCP has been verified on both aspects: PCP [1] behavior and the support of migration.

Verification for DFLP. The experiment used to verify the behavior of the DFLP [54], must support that, it is possible to investigate first, whether all requests related to a resource are served in FIFO order, and second, whether the requests for multiple resources are scheduled preemptively in order of increasing lock-request time. Moreover, the task migration and priority-boosted agents should be considerable under the experiment too, as DFLP, same as DPCP, is a distributed-based locking protocol. To this end, a sample task-set is designed as follow, which includes four tasks partitioned over two processors. Note that, the resource partition is according to the assignment rule (1) and the task partition is conducted using the worst-fit decreasing heuristic (see section 2.3.1) based on processor utilization.

```
./rtspin -p 0 -q 1 -X DFLP -x 1:1                                    -o 4
-w 1 12 2 &
./rtspin -p 0 -q 2 -X DFLP -x 3:1:1:1 -L 2:1:4 -M 2:-1:-1 -Q 2:1:2
-w 8 16 2 &
./rtspin -p 1 -q 3 -X DFLP -x 2:1:1   -L 1:2   -M 1:0     -Q 1:1   -o 5
-w 4 20 2 &
./rtspin -p 1 -q 4 -X DFLP -x 3:1:1:1 -L 2:1:1 -M 2:0:0   -Q 2:2:1 -o 3
-w 5 25 2 &
./rtspin -p 1 -q 5 -X DFLP -x 2:1:2   -L 1:1   -M 1:0     -Q 1:2   -o 5
-w 4 30 2 &
```

The result of the task-set schedule after running on the test-bed, is presented in Figure 3.3.

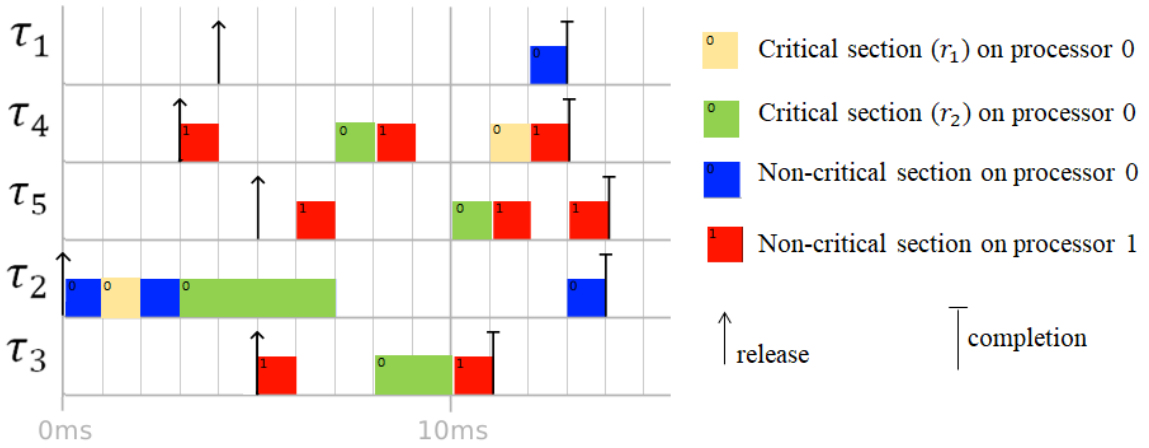


Figure 3.3: DFLP's behavior implemented on LITMUS^{RT}

As Figure 3.3 shows, the highest priority task τ_1, is released at time 4. However, it cannot start its execution immediately as another task τ_2 is running its critical section on the same processor, i.e., agent A_2 is carrying out request $R_{2,2,1}$. Therefore, the priority boosting of agents is confirmed. At time 7, at which τ_2 releases resource r_2, three requests for r_2 (first $R_{4,2,1}$ at time 4, second $R_{3,2,1}$ at time 6 and third $R_{5,2,1}$ at time 7) were already put in the queue of r_2 (i.e. F_2). As the result

shows, these three requests are served exact in the same order they were in F_2, i.e., they are served in FIFO order. Additionally, since all these requests are issued by the tasks local to a different processor from the r_2's processor, the task migration can be verified for the implementation of the DFLP. In order to verify the last feature of the DFLP, that the requests for multiple resources should be scheduled in order of increasing lock-request time, it is sufficient to focus on the time 10, when two requests $R_{4,1,1}$ and $R_{5,2,1}$ can be performed (i.e., both resources r_1 and r_2 are free). In this situation, $R_{5,2,1}$ is served before $R_{4,1,1}$, as $R_{5,2,1}$ was issued earlier than $R_{4,1,1}$. As a consequence, all the properties of DPCP have been verified.

Verification for MPCP. In order to verify the behavior of the MPCP [4], it is necessary to consider two issues: (i) the non-critical sections should not be able to preempt a task that is executing its critical section (no matter what their priority is), (ii) a task should be able to interrupt another task even during the execution of its critical section if and only if, the preempted task has the lower ceiling than the another one. The sample task-set designed as follow makes it possible to investigate both issues. Note that, tasks are partitioned over two processors using the worst-fit decreasing heuristic (see section 2.3.1) based on processor utilization.

```
./rtspin -p 0 -q 2 -X MPCP -x 3:2:1:1 -L 2:1:2 -Q 2:2:2 -o 2  -w 7  35
2 &
./rtspin -p 0 -q 3 -X MPCP -x 3:2:1:2 -L 2:3:4 -Q 2:3:1 -o 7  -w 12 40
2 &
./rtspin -p 0 -q 4 -X MPCP -x 2:1:1   -L 1:3   -Q 1:3         -w 5  45
2 &
./rtspin -p 1 -q 1 -X MPCP -x 2:1:2   -L 1:3   -Q 1:1   -o 15 -w 6  30
2 &
./rtspin -p 1 -q 5 -X MPCP -x 3:5:4:1 -L 2:5:3 -Q 2:2:2       -w 18 50
2 &
```

The result of the task-set schedule after running on the test-bed, is presented in Figure 3.4.

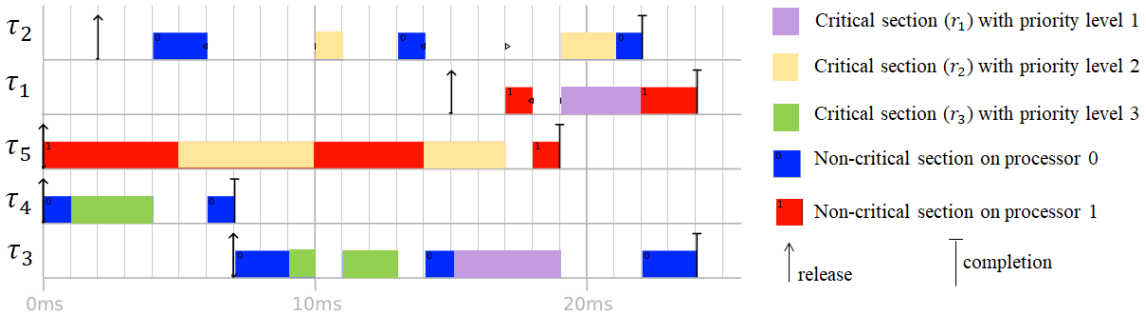


Figure 3.4: MPCP's behavior implemented on LITMUS^{RT}

As Figure 3.4 shows, at time 2, task τ_2 is released, which has the highest priority among the tasks assigned to the processor 0. However, it cannot interrupt the critical section executed by τ_3 on the same processor (the same situation happens also at time 15). Therefore, the feature that, critical

sections always having the higher priority than non-critical sections, is confirmed. At time 10, when τ_5 releases resource r_2, τ_2 resumes its execution and enters to the critical section. So, τ_3 is preempted by τ_2, although τ_3 was running its critical section. Such preemption happens because the τ_2's critical section (r_2) has the higher ceiling than τ_3's critical section (r_3). Such situation is an example exhibits that, a critical section can be preempted by other critical section when it has the higher ceiling than the other one, which is under executing. To verify that the critical section with higher ceiling cannot be preempted by other critical section with lower priority ceiling, it is sufficient to focus on the behavior of the task-set schedule at time 17 in Figure 3.4. At this moment, τ_5 releases r_2, which is already requested by τ_2. However, τ_2 cannot interrupt the execution of τ_3, because τ_3 occupies at this time resource r_3, which has the ceiling priority equal to τ_1's priority. Note that, τ_1 is the highest priority task among all the tasks in the sample task-set. Hence, τ_2 cannot preempt τ_3 any more. As a result, all the properties of the MPCP have been verified.

Verification for MPCP-VS. Since MPCP-VS [55] is the preemptive-spinning variant of the MPCP, considering the verification of MPCP above, here we focus to verify the preemptive-spinning behavior of the MPCP-VS implemented under LITMUS^{RT} [6] [7]. To this end, a simple task-set designed as follow was run on the test-bed and the result of the task-set schedule is depicted in Figure 3.5.

```
./rtspin -p 0 -q 1 -X MPCP_VS -x 2:3:2 -L 1:2 -Q 1:1 -w 7 16 2 &
./rtspin -p 0 -q 2 -X MPCP_VS -x 2:2:2 -L 1:1 -Q 1:2 -w 5 16 2 &
./rtspin -p 1 -q 3 -X MPCP_VS -x 2:1:2 -L 1:6 -Q 1:1 -w 9 16 2 &
```

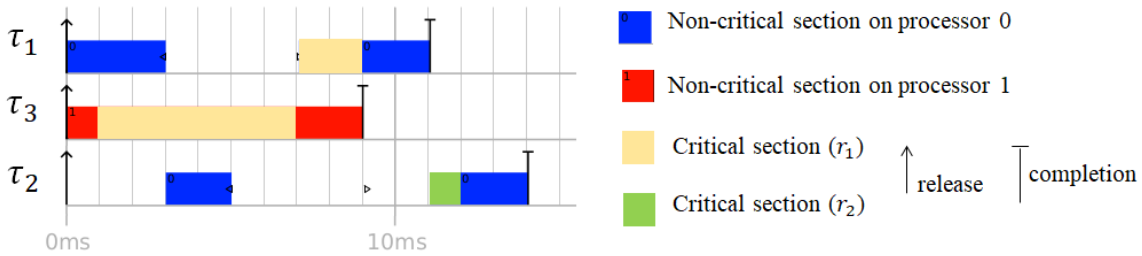


Figure 3.5: MPCP-VS's behavior implemented on LITMUS^{RT}

According to the result presented in Figure 3.5, the virtual spinning happens on processor 0. Task τ_1 requests resource r_1 at time 3 and therefore it has to wait for accessing r_1 since the resource is being used by the other task τ_3 at this moment. Since τ_1 spins preemptively, the other task τ_2 located on the same processor can start to run. However, τ_2 can execute until time 5, at which it requests a resource r_2. In such situation, although r_2 is free, τ_2 must suspend, i.e., it is not permitted to issue its request due to the uncompleted request of task τ_1. In this regard, the MPCP-VS has been verified on spinning preemptively aspect.

3.2.5. Measuring of Test Parameters

In this thesis, there are three main indicators to show the performance of suspension-based and spin-based protocols in practice and compare them with each other: *schedulability*, *maximum response time* and *average response time*.

These parameters were measured and recorded using *Feather-Trace-Tool*, a low-overhead tracing framework [61].

Schedulability. If the deadline of all tasks of a task-set are met, then the task-set is *schedulable*. In other words, the deadline of each task in the task-set must not be more than the worst-case response time of the task. In order to recognize the schedulability of a task-set, a schedulability test can be used, which is a test, whose result indicates if a task-set is schedulable under a specific setting of the system [10].

Response Time. The length of the interval between the task's arrival and its finishing time is named the *response time* of the task. In the domain of the real-time systems, it is common to consider the maximum response time (the worst-case response time) of tasks in order to explore the schedulability of the system [10]. However, the average response time of a task is sometimes needed to be taken into consideration, specially under the circumstances related to the workload balancing.

For tracing and exporting the status information of tasks execution, LITMUS^{RT} provides some devices. One of them, which is used in this thesis, is called `st-trace-schedule()` device. By exploiting this infrastructure, all scheduling events can be recorded, i.e., it records, which tasks are scheduled at what point and corresponding job releases and deadlines. The purpose of this tool is to record a scheduling event whenever a task is dispatched (switched to), is preempted (switched away), suspends (blocks) or a task resumes (wakes up) [62]. Furthermore, the primary parameters of each task, which are release time, deadline, and finish time, are recorded as well as the time of a synchronous task system release (if any). The `st-trace-schedule()` device exports structured data, which is not human-readable. Therefore, a tool can be used to plot data in order to obtain a visual depiction of the recorded schedule or to convert them to a CSV file. For example, a trace can be rendered as a PDF file through the use of the pycairo[7]-based `st-draw()` tool or a CSV file with relevant per-job statistics can be produced by the tool `st-job-stats()`. All these tools are available in feather-trace-tool. The reader interested for more information about the feather-trace-tool is referred to the website of the LITMUS^{RT}[8].

[7] `http://cairographics.org/pycairo/`

[8] `https://www.litmus-rt.org/tutorial/manual.html#tracingaschedule`

4 Evaluation

As it mentioned before, the goal of this thesis is to analyze conditions that favor spin-based locking protocols over suspension-based locking protocols and the other way around and also to explore such tradeoff in detail. To this end, after preparing the required conditions as they explained in previous chapters, necessary experiments are conducted through the use of the LITMUS^{RT} [6] [7] to compare spin-based and suspension-based synchronization mechanisms as provided via the protocols mentioned in section 2.5. The results of these experiments are discussed in this chapter as well as the settings of the examinations conducted during the work of this thesis.

4.1. Experimental Set-Up

Before presenting the evaluation of the experiments, the required foundation should be established as well as the basic settings for the experiments. In this regard, the relevant scheduling and partition algorithms are reviewed. Then, the formalized resource assignment rule and task-set configuration are explained in the rest of this section. So, with respect to the background materials presented in the previous chapters, limitations of the work presented in this thesis has been defined in the following subsections (they are also summarized in Table 4.2).

4.1.1. Scheduling Algorithm

In this work, the *Partitioned Fixed-Priority (P-FP) Rate-Monotonic (RM)* scheduling algorithm is used. By working with a fixed-priority scheduling, tasks are usually indexed in order of decreasing priority [34], which is adopt in this work as well. In such wise, the priority function $Y(\tau_i) = i$ can be a simple and useful function for FP scheduling. Now the important point is to find a solution of priority assigning to tasks. One of these solutions is the *rate-monotonic (RM)* priority assignment introduced by Liu and Layland [16]. RM scheduling indexes tasks in order of increasing period, i.e., for each two tasks τ_i and τ_j with period p_i and p_j respectively, if $p_i \leq p_j$, then $i < j$. In other words, the shorter the task period is, the higher is the task priority. An example of RM scheduling on a uniprocessor system is depicted in Figure 4.1 according to an implicit-deadline periodic task-set presented in Table 4.1. In this example, all four tasks are scheduled in order of decreasing task priority and they are released at the same time (at time 0). At time 4, the second job of task τ_1, J_{2_1}, preempts the first job of task τ_3, J_{1_3}, as the priority of task τ_1 is higher than the priority of task τ_3. Likewise, task τ_3 cannot be scheduled again until the completion of J_{2_1} at time 6. The lowest priority task τ_4 cannot be scheduled until all previously released higher-priority tasks have been completed, when is at time 7.

	WCET	Period
τ_1	1	4
τ_2	1	5
τ_3	3	9
τ_4	3	18

Table 4.1: Example of implicit-deadline periodic task-set

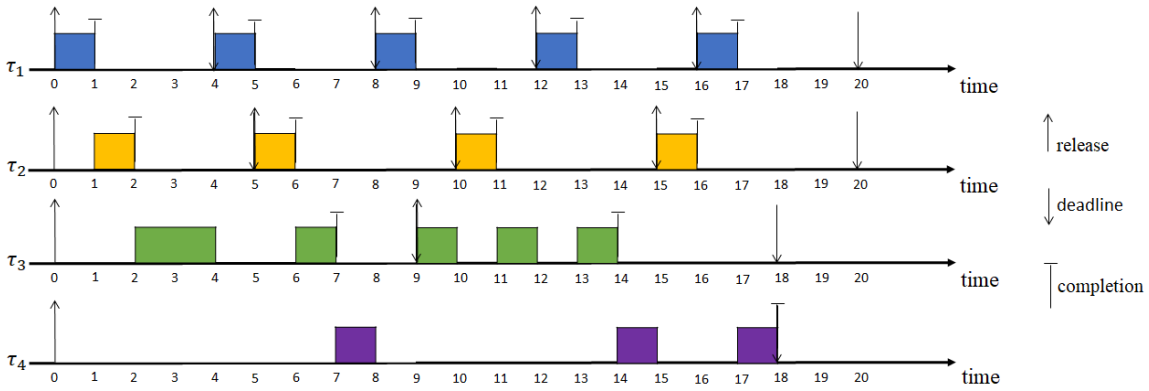


Figure 4.1: Example of RM scheduling

Liu and Layland [16] proved that, RM scheduling is in fact optimal for an implicit-deadline periodic task-set with respect to FP schedulers on a uniprocessor. In other words, if a task-set is schedulable under some FP schedulers, then it is also schedulable under RM scheduling. Their theorem says:

"A set of n independent, preemptable periodic Tasks $\tau = \{T_1, \dots, T_n\}$ *with relative deadlines equal to their respective periods can be scheduled on a processor according to the RM algorithm if its total utilization is at most* $n(2^{\frac{1}{n}} - 1)$*"* [16].

4.1.2. Partition Algorithm

Among the heuristics argued in section 2.3.1, an interpreted version of non-optimal worst-fit heuristic have been used by the thesis for partitioning a task-set on the multiprocessor system.

Recall that, the goal of the bin-packing heuristics is to minimize the number of bins. To this end, the items are assigned to partially used bins whenever possible and a new bin will be in used only when it is absolutely required. This approach is a slight mismatch to the problem of partitioning a task-set onto a given multiprocessor system with the fix number of processors *m*. This means that, the act of adding new processor (new bin) is not possible under the multiprocessor platform. Moreover, in a multiprocessor system it is not worst to leave one or more processors completely unallocated to real-time tasks, especially in the context of energy-aware systems. In this regard, the aforementioned bin-packing heuristics are interpreted as follows: It is assumed that, there exist initially *m* empty bins. Whenever an additional bin is required, it means that, the heuristic failed to partition the task-set. Given the fact that, under both first-fit and best-fit heuristics, an idle

processer is allocated a task only if the other used processors are already fully allocated, this interpretation has little impact on these heuristics. In contrast, by using this interpretation, the worst-fit heuristic provides the spreading out of the total utilization among all processors more-or-less evenly [34]. For the following two reasons, it is preferred to spread out the load of the system among all processors. For one thing, there is more benefit to use all available processors instead of leaving them idle. For another, since overhead accounting causes the inflation of task utilizations, it is useful to leave some free capacity on each processor to compensate for such inflation [34]. In regard to these reasons, the worst-fit decreasing heuristic based on processor utilization is chosen to partition tasks among processors in the examinations of the thesis.

4.1.3. Resource Assignment Rule

As mentioned in previous chapter, four locking protocols implemented on the test-bed LITMUS^{RT} [6] [7] are used to conduct the evaluations. There are two distributed-based protocols, i.e. DPCP [3] and DFLP [54], among these protocols. Therefore, in the case of using distributed-based protocols to protect resource accesses, it should be determined, which resource is local to which processor. In other words, it is necessary to assign one or more available shared resources to processors of the system according to a resource assignment rule, which must be fixed prior in order to analysis.

The assignment rule used in the thesis is defined as follow: After assigning priority to each task and applying the task partitioning algorithm, resource r_q is local to processor P_j, if the first request for resource r_q is issued by one of the tasks allocated to the processor P_j. By this assumption that S_j is the set of assigned tasks to the processor P_j, if $\tau_i \in S_j$ and there is a request $R_{i,q,1}$, then the resource r_q is assigned to the processor P_j, i.e.

$$r_q = \{j \mid \exists R_{i,q,1}, \tau_i \in S_j\} \quad (1)$$

4.1.4. Task-Set Configuration Parameters

To evaluate the candidate locking protocols, it is required to configure several reasonable task-sets at first. For the following two reasons, it is important to be careful in choosing the construction of the task-set. For one thing, the performance of the locking protocols is examined based on their ability to handle the task-sets. For another, the results of the examination may vary for different choices of task-set, i.e., the performance of protocols dependent not only on the task partition, but also on the task-set. In other words, a task-set must be neither too simple nor too complicated, so the benefits of the synchronization mechanism can be highlighted as well as possible to analyze. In this regard, the approach proposed by Emberson et al. [63] is used to generate the task periods over the maximum number of resource accesses per task $K \in \{2, 4, 8, 16\}$. For low (5%) and medium (25%) task utilization 200 task-sets are generated (100 task-set for each of them) and 40 task-sets for high (70%) task utilization. Depending on the number of processors $m \in \{2, 4, 8, 16\}$,

each task-set contains $N = \left\lceil \frac{50}{m} \right\rceil \times m$ tasks and all processors are assigned equal number of tasks. The distribution of periods is from 1ms to 10ms. The approach generates implicit-deadline task-set, i.e., the task relative deadline equals the task period. The number of shared resources in a task-set is determined through the use of the formula $n = \left\lceil \frac{50 \times K}{m} \right\rceil$. The ratio of the total length of critical sections to the total length of non-critical sections varies for different value of $\alpha \in \{1.11, 1.25, 2, 5, 10\}$. For example, if $\alpha = 1.25$, then the ratio of the critical sections to the non-critical sections equals $\frac{1}{1.25} = 0.8$, which means that 80% of the task total execution time is the total length of critical section. In other words, the larger value of α is, the shorter is the length of critical sections. Each critical section uses just one of the *n* resources. The WCETs of the non-critical-sections and critical-sections of task τ_i are calculated according to the task period p_i and the total utilization of task critical sections denoted U_i^C and the total utilization of task non-critical sections denoted U_i^{nonC} respectively, which are generated through the task generator [63]. In this regard, the worst-case execution time of task τ_i is defined as $C_i = p_i \times \left(U_i^C + U_i^{nonC}\right)$.

4.2. Results Comparison and Analyzing

The results presented in this section, are from a set of experiments, which are conducted under the test-bed LITMUS^{RT} [6] [7]. These experiments compare spin-based and suspension-based synchronization mechanisms together as provided via different suspension-based protocols (MPCP, DPCP, DFLP) and a spin-based protocol MPCP-VS. Note that, the performance of a protocol depends not only on its theory but also its implementation. Additionally, the performance of a protocol may be varied by implementing on different platforms and by running it on machines with different configuration as well. In this regard, all the analyses and evaluations conducted in this section are based on the current implementation of each protocol on the current version of the LITMUS^{RT} (version 2017.1) running on the platform with the configuration mentioned in section 3.1.

Considering almost all possible combinations of parameters in the experimental set-up, it results 960 configurations, which lead to 155 graphs to present the results of the execution of all these test-cases. Although only some representative example graphs are present here, the complete set of graphs can be found in Appendix. The experimental set-up parameters, which are already fully-described in previous section, are also summarized in Table 4.2.

The evaluation (comparison) is based on schedulability (i.e. number of tasks missing deadline), maximum and average response-time. As the result, the most beneficial synchronization mechanism for each situation is discussed. Additionally, for each locking protocol 240 randomly-generated task-sets are used, which are generated through the task-set generator described in section 4.1.4.

Scheduling Algorithm	Priority Fixed-Partitioned Rate Monotonic (P-FP RM)
Task-Partition Algorithm	Worst-fit decreasing based on processor utilization
Resource-Distribution Algorithm	Based on the resource assignment rule (1), i.e., after assigning priority to each task and applying the task partitioning algorithm, resource r_q is local to processor P_j, if the first request for r_q is issued by one of the tasks allocated to P_j
m = Number of Processors	{2, 4, 8, 16}
K = Number of resource accesses per task (Number of critical sections per task)	{1, 2, 3, 4, 5}
Period of each task	[1ms, 10ms]
Ratio of the total length of critical sections (cs) to the total length of non-critical sections (non-cs)	{10%, 20%, 50%, 80%, 90%}
Average utilization of each task	{0.05, 0.25, 0.7}
Number of tasks per processor	$\left\lfloor \frac{50}{Number\ of\ processors} \right\rceil = \left\lfloor \frac{50}{m} \right\rceil = \{25, 13, 6, 3\}$
N = Number of tasks per task-set	$\left\lfloor \frac{50}{Number\ of\ processors} \right\rceil . Number\ of\ processors = \left\lfloor \frac{50}{m} \right\rceil . m = \{48,50,52\}$
n = Number of shared resources per task-set	$\left\lceil \frac{Number\ of\ resource\ access\ per\ task\ .50}{Number\ of\ processors} \right\rceil = \left\lceil \frac{K\ .50}{m} \right\rceil$

Table 4.2: Experimental set-up

4.2.1. Schedulability Evaluation

The subgraphs in this section illustrate results obtained by executing of tasks with low and medium utilization as well as the number of resource accesses per task $K = 1$ to 5 resources on $m = \{2, 4, 8, 16\}$ processors. The x-axis of each subgraph gives the ratio of the total length of critical sections (cs) to the total length of non-critical sections (non-cs). Each graph is resulted from the running of 20 task-sets. In this section, number of tasks missing a deadline per task-set is concerned.

Figure 4.2 contains subgraphs, which represent results obtained by executing of tasks with low utilization. Subgraph (a) in this figure shows results obtained for $m = 2$ and $K = 1$. Without regard to exceptions, there are several factors to notice after analyzing this graph. First, by increasing the ratio of the total cs length to the total non-cs length roughly until 50%, the number of tasks missing a deadline per task-set are decreased, but afterwards the number of tasks missing a deadline per task-set tends to be higher. Second, by using spin-based variant of the MPCP, the number of tasks missing a deadline are less than in comparison to the situation when we use suspension-based variant of the MPCP. Third, by using spin-based variant of the MPCP, the

number of tasks missing a deadline are more than the situation when we use the DPCP or the DFLP.

Subgraph (b) in Figure 4.2 demonstrates results obtained for $m = 4$ and $K = 3$. Again, there are several interesting things to note. First, by using the spin-based variant of the MPCP, the number of tasks missing deadline is approximately as large as the number of tasks missing deadline using the suspension-based variant of the MPCP. Second, by using DPCP or DFLP, we have more tasks, which missed their deadline, in comparing with the situation when we use the MPCP (both spin- or suspension-based variant) generally.

Subgraph (c) and (d) in Figure 4.2 are approximately same, which display results obtained for $m = 8, \mathrm{K} = 4$ and $m = 16, K = 5$ respectively. Without regard to exceptions, there are several things to notice after analyzing both graphs. First, if the ratio of the total cs length to the total non-cs length is between 20% and 80%, the number of tasks missing deadline per task-set is more than the number of tasks missing deadline resulted by using other ratios (90% or 10%). Second, by using the DPCP, the DFLP and suspension-based variant of the MPCP, there are more tasks missing deadline in comparing with the situation when we use the spin-based MPCP.

All in all, after comparing all subgraphs in Figure 4.2 with each other, without regard to exceptions it can be found out first, by increasing the number of processors and task resource accesses, the total number of tasks missing deadline tends to be fewer and second, if the used system has 2 or 4 processors and the average utilization of each task equals 5% and each task is allowed to access just 3 or fewer resources, it is better to use the suspension-based locking protocols, else applying a spin-based locking protocol can be a proper solution to reach fewer tasks missing deadline. Other results that were obtained but not shown here (instead shown in Appendix) support these conclusions.

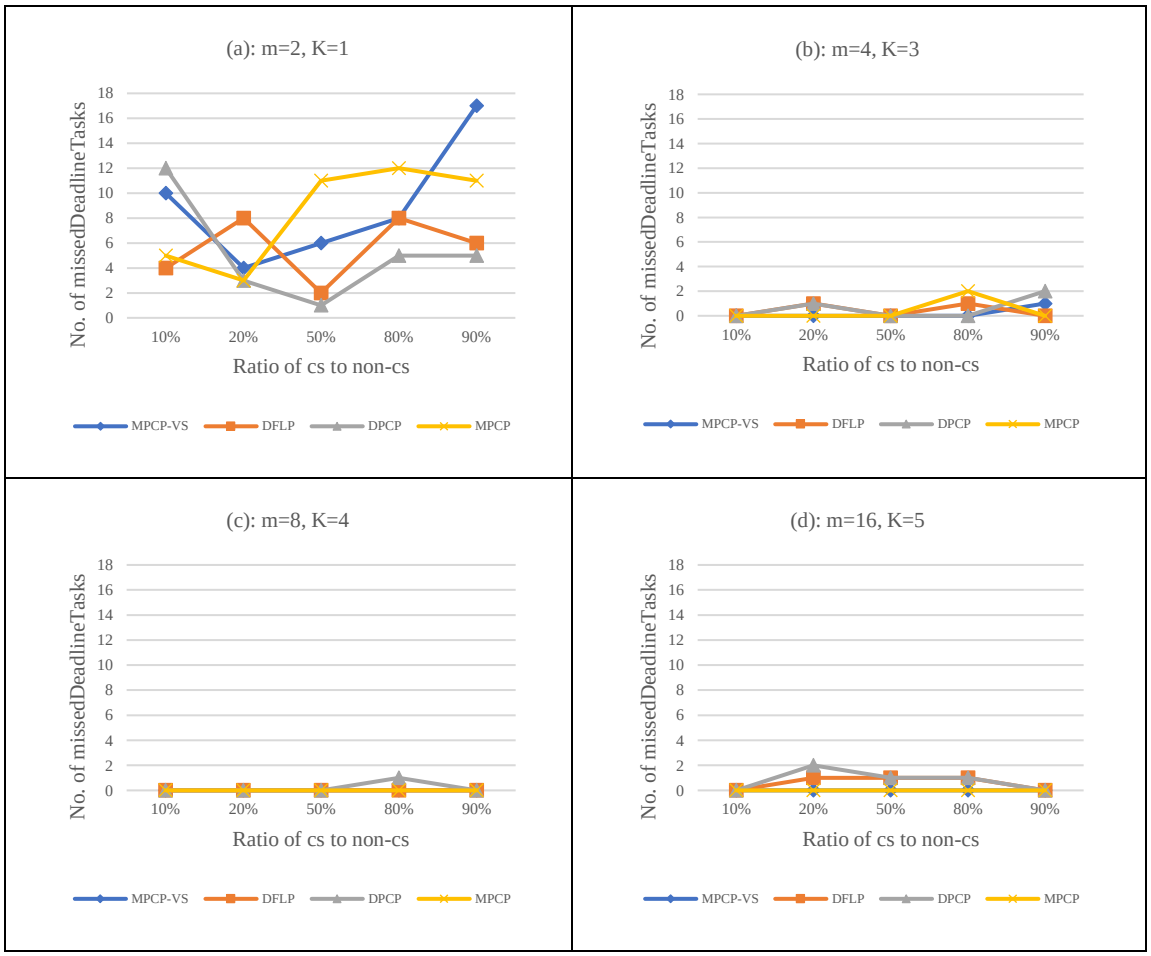


Figure 4.2: Schedulability results obtained by executing of tasks with low utilization

The deliberation of the number of tasks missing deadline per task-set will be continued in Figure 4.3 but here the utilization is increased to 25%.

Subgraph (a) in Figure 4.3 presents results obtained for $m = 2$ and $K = 2$. Without regard to exceptions, there are several factors to notice after analyzing this graph. First, by increasing the ratio of the total cs length to the total non-cs length, the number of tasks missing deadline per task-set is not changed drastically. Second, the number of tasks missing deadline by using all the locking protocols is roughly close to each other but not exactly the same. Third, by increasing the ratio of the total cs length to the total non-cs length from 50%, the usage of the spin-based variant of the MPCP leads to the fewer number of tasks missing deadline in comparison to using a suspension-based locking protocol.

Subgraph (b) in Figure 4.3 shows results obtained for $m = 4$ and $K = 3$. Without regard to exceptions, there are several factors to note. First, again here same as in subgraph (a), the number of tasks missing deadline per task-set is not changed violently by increasing the ratio of the total cs length to the total non-cs length. Second, the number of tasks missing deadline using the DPCP or the DFLP locking protocol is generally fewer than the number of tasks missing deadline using the MPCP locking protocol (both spin- or suspension-based variant).

Subgraph (c) in Figure 4.3 shows results obtained for $m = 8$ and $K = 4$. Without regard to exceptions, there are several factors to notice after analyzing this graph. First, by increasing the ratio of the total cs length to the total non-cs length, there is a slight increase in the number of tasks missing deadline per task-set. Second, the spin-based locking protocol does not seem to be a good choice in comparison with the suspension-based protocols, because the number of tasks missing deadline using the suspension-based protocols is fewer than the number of tasks missing deadline using the spin-based protocol generally.

Subgraph (d) in Figure 4.3 shows results obtained for $m = 16$ and $K = 5$. Without regard to exceptions, there are several factors to notice after analyzing this graph. First, by increasing the ratio of the total cs length to the total non-cs length, the number of tasks missing deadline per task-set is increased drastically. Second, the number of tasks missing deadline using the spin-based MPCP is roughly same as the number of tasks missing deadline using the suspension-based protocols DPCP and DFLP. Third, through the use of the suspension-based protocol MPCP, the number of tasks missing deadline is fewer than the number of tasks missing deadline through the use of the other protocols.

All in all, after comparing all subgraphs in Figure 4.3 with each other, it can be realized that, the number of tasks missing deadline per task-set is not changed intensely by increasing the number of processors and task resource accesses, unless the number of used processors is more than 8. This means that, in such situation, the total number of tasks missing deadline tends to be fewer. Moreover, if the used system has 2 processors and the utilization of each task equals 25%, it is suitable to use the spin-based locking protocol, else applying one of the suspension-based locking protocols can be a proper solution to achieve fewer tasks missing deadline. Other results that were obtained but not shown here (instead shown in Appendix) support these conclusions.

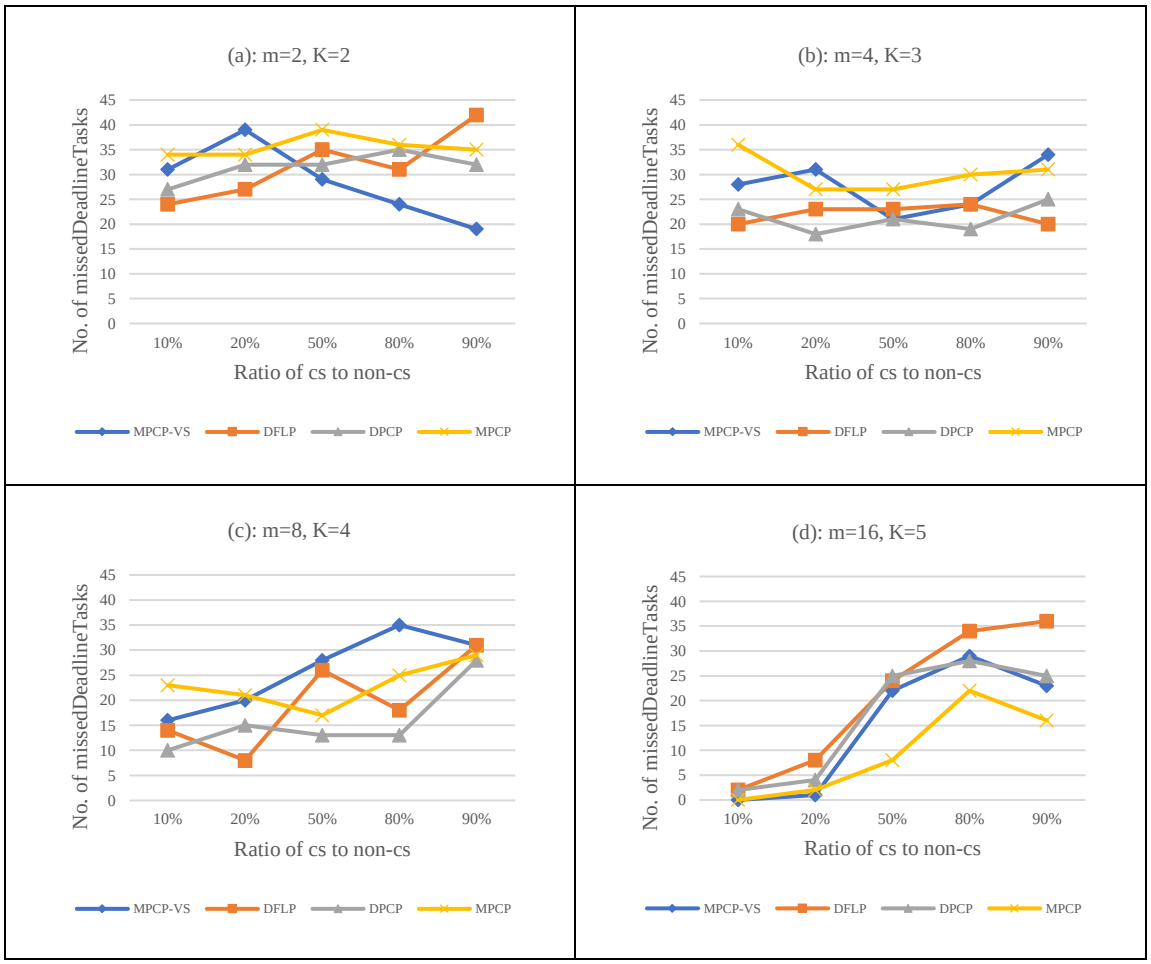


Figure 4.3: Schedulability results obtained by executing of tasks with medium utilization

4.2.2. Maximum Response Time Evaluation

In this section, the factor maximum response time per task-set is taken into consideration.

The subgraphs in Figure 4.4 and Figure 4.5 illustrate results obtained by executing of tasks with low and medium utilization as well as the number of resource accesses per task $K = 1$ to 4 resources on $m = \{2, 4, 8, 16\}$ processors. The x-axis of each subgraph in those figures gives the ratio of the total length of critical sections (cs) to the total length of non-critical sections (non-cs). Each graph is resulted from the running of 20 task-sets.

The results of all subgraphs in Figure 4.4, besides graph (a), are roughly the same. However, by downscaling the graphs (b), (c) and (d), it is figured out that, the maximum response time per task-set is decreased by increasing the number of processors and task resource accesses. Moreover, if a system with 4 or fewer processors and tasks utilization equal to 5% is used, it is better to use one of the suspension-based locking protocols (except the MPCP), else applying a spin-based locking protocol can be a proper solution to reach less maximum response time for each task-set.

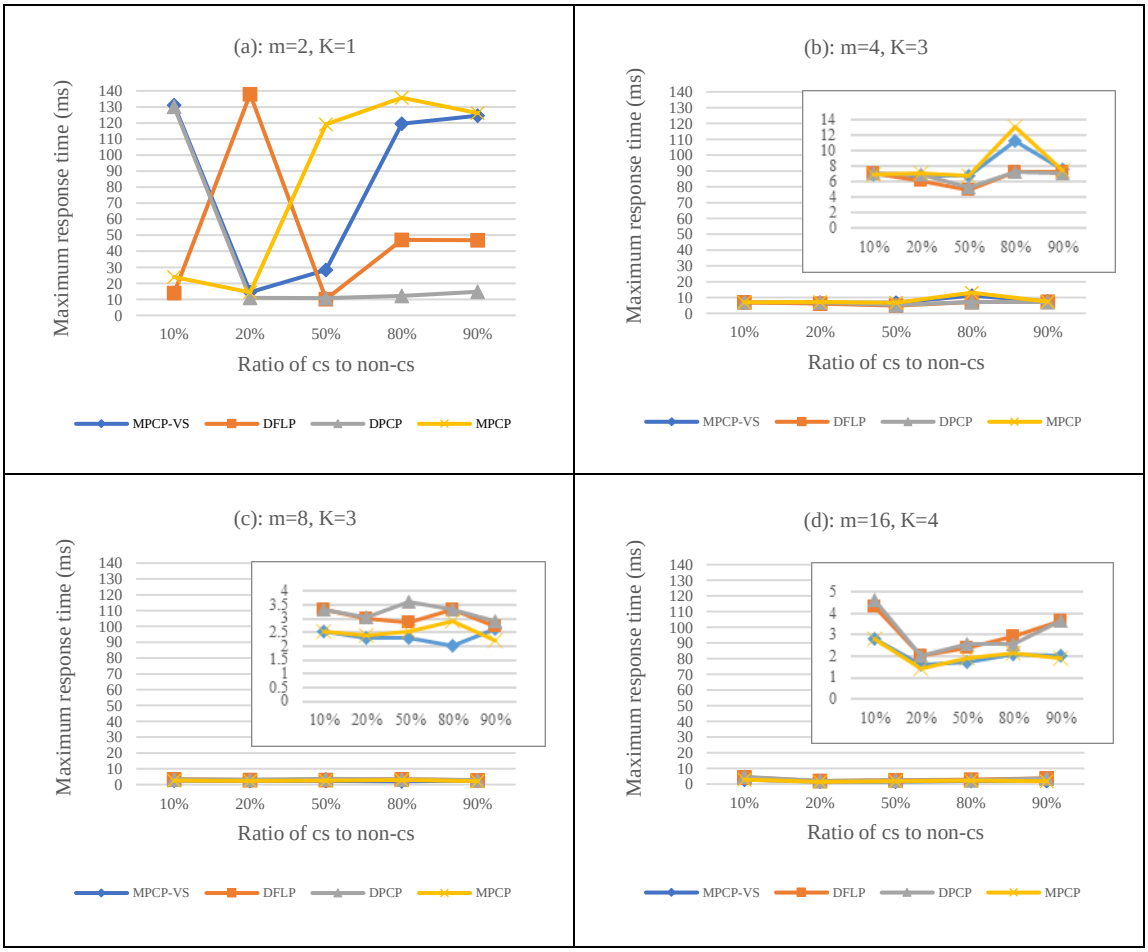


Figure 4.4: Maximum response time results obtained by executing of tasks with low utilization

The consideration of maximum response time per task-set will be continued in Figure 4.5 but here the utilization is increased to 25%.

Subgraph (a) in Figure 4.5 presents results obtained for $m = 2$ and $K = 1$. Without regard to exceptions, there are several factors to notice after analyzing this graph. First, by increasing the ratio of the total cs length to the total non-cs length, if the suspension-based MPCP is used, then the maximum response time per task-set is increased lowly, otherwise it tends to decrease. Second, under the circumstance of using the DFLP or the DPCP, by increasing the ratio of the total cs length to the total non-cs length to 20%, the maximum response time is less than the maximum response time resulting from the usage of spin-based MPCP.

Subgraph (b) in Figure 4.5 shows results obtained for $m = 4$ and $K = 2$. There are several factors to notice. First, if one of the suspension-based protocols is used, by increasing the ratio of the total cs length to the total non-cs length, the maximum response time per task-set is not changed drastically. Second, under the circumstance of using DFLP or DPCP, by increasing the ratio of the total cs length to the total non-cs length to 50%, the maximum response time is less than the maximum response time resulting from the usage of the MPCP (both spin- or suspension-based variant). However, by continuing the increment of the ratio of the total cs length to the total non-cs length from 50%, the use of spin-based the MPCP results less maximum response time.

The outcomes in subgraph (c) and (d) are almost the same, which illustrate results obtained for $m = 8, K = 3$ and $m = 16, \mathrm{K} = 4$ respectively. There are several factors to notice after analyzing both graphs. First, if one of the suspension-based locking protocols is used, by increasing the ratio of the total cs length to the total non-cs length, the maximum response time per task-set increased, otherwise it tends to decrease. Second, under the circumstance of using suspension-based locking protocols, by increasing the ratio of the total cs length to the total non-cs length to approximately 25%, the maximum response time is less than the maximum response time resulting from the usage of the spin-based variant of the MPCP. But for the ratio of the total cs length to the total non-cs length more than 25%, the use of spin-based MPCP results less maximum response time.

All in all, after comparing all subgraphs in Figure 4.5 with each other, it is discovered that, the maximum response time per task-set is decreased by increasing the number of processors. Furthermore, if there is a system with 4 or more processors and task utilization equal to 25%, then it is better to use a spin-based locking protocol generally, else applying one of the suspension-based locking protocols can be a proper solution to reach less maximum response time for each task-set. Other results that were obtained but not shown here (instead shown in Appendix) support these conclusions.

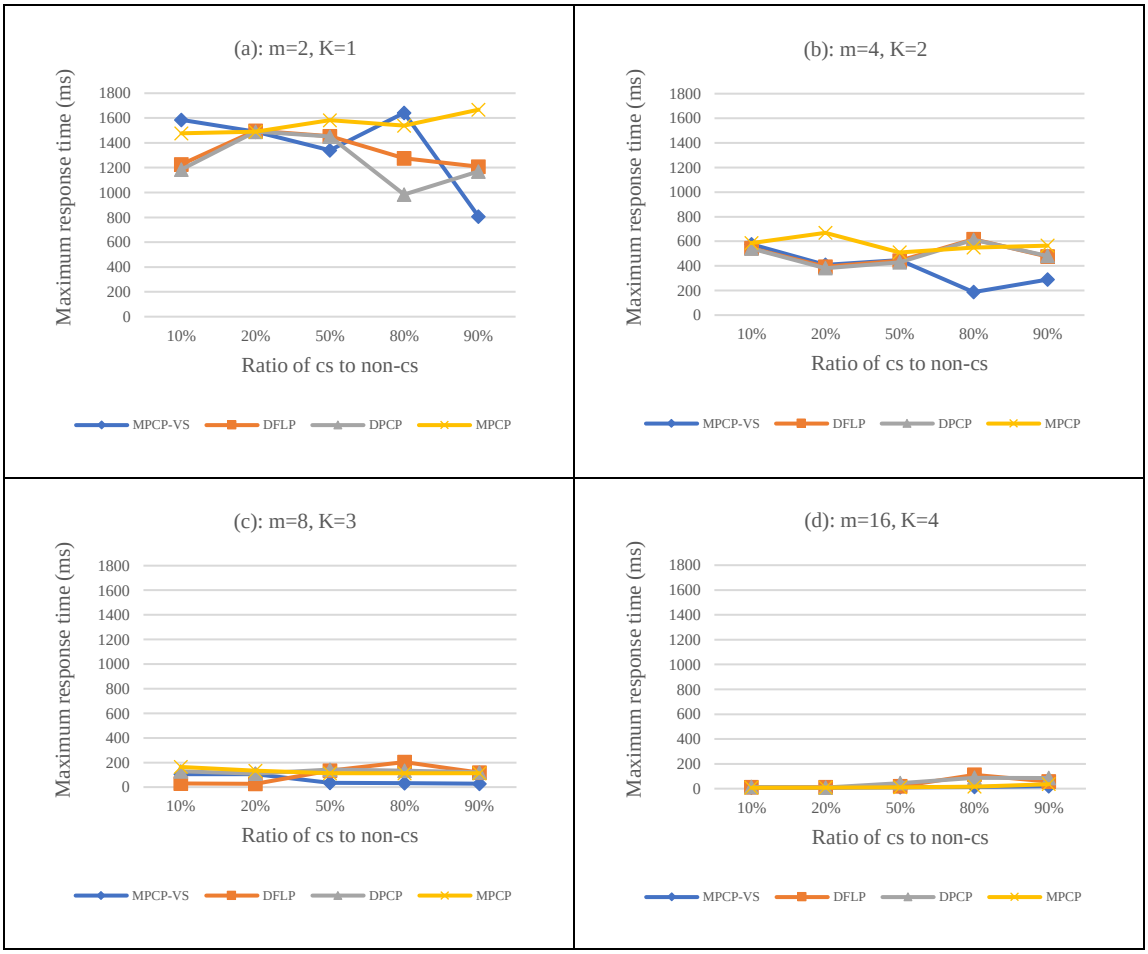


Figure 4.5: Maximum response time results obtained by executing of tasks with medium utilization

The subgraphs in Figure 4.6 illustrate results obtained by executing of tasks with the number of resource accesses per task $K = 1$ to 5 resources and the ratio of the total cs length to the total non-cs length per task-set equals 10%. The x-axis of each subgraph in this figure gives the percentage of the task utilization. Each graph is resulted from the running of 12 task-sets.

Subgraph (a) in Figure 4.6 presents results obtained for $m = 2$ and $K = 1$. There are several factors to note. First, as the utilization becomes higher, the maximum response time per task-set tends to increase no matter which type of locking protocols is used. Second, the longest maximum response time per task-set will be obtained if the spin-based variant of the MPCP is used as the locking protocol. Third, to gain the shortest maximum response time, we can use both DPCP and DFLP, because the maximum response times of each task-set for both protocols are approximately the same.

Subgraph (b) in Figure 4.6 shows results obtained for $m = 4$ and $K = 3$. There are several factors to note. First, as the utilization becomes higher, the maximum response time per task-set tends to increase. Second, the maximum response time using spin-based variant of the MPCP is shorter than the maximum response time using suspension-based variant of the MPCP; however, the shortest maximum response time per task-set is obtained using the DPCP and the DFLP.

The outcomes in subgraph (c) and (d) are almost the same, which illustrate results obtained for $m = 8, K = 4$ and $m = 16, K = 5$ respectively. There are several factors to notice after analyzing both graphs. First, as the utilization becomes higher, the maximum response time per task-set tends to increase. Second, the maximum response time using spin-based variant of the MPCP is shorter than the maximum response time using suspension-based variant of the MPCP. Third, if the number of resource accesses is more than 3, then the shortest maximum response time per task-set is obtained using spin-based variant of the MPCP.

All in all, after comparing all subgraphs in Figure 4.6 with each other, it can be found out that, the maximum response time per task-set will be decreased by increasing the number of processors. Additionally, if the used system has 4 or more processors, while the ratio of the total cs length to the total non-cs length per task-set is equal to 10%, it is better to use a spin-based locking protocol generally, else applying one of the suspension-based locking protocols can be a proper solution to reach less maximum response time for each task-set. Other results that were obtained but not shown here (instead shown in Appendix) support these conclusions.

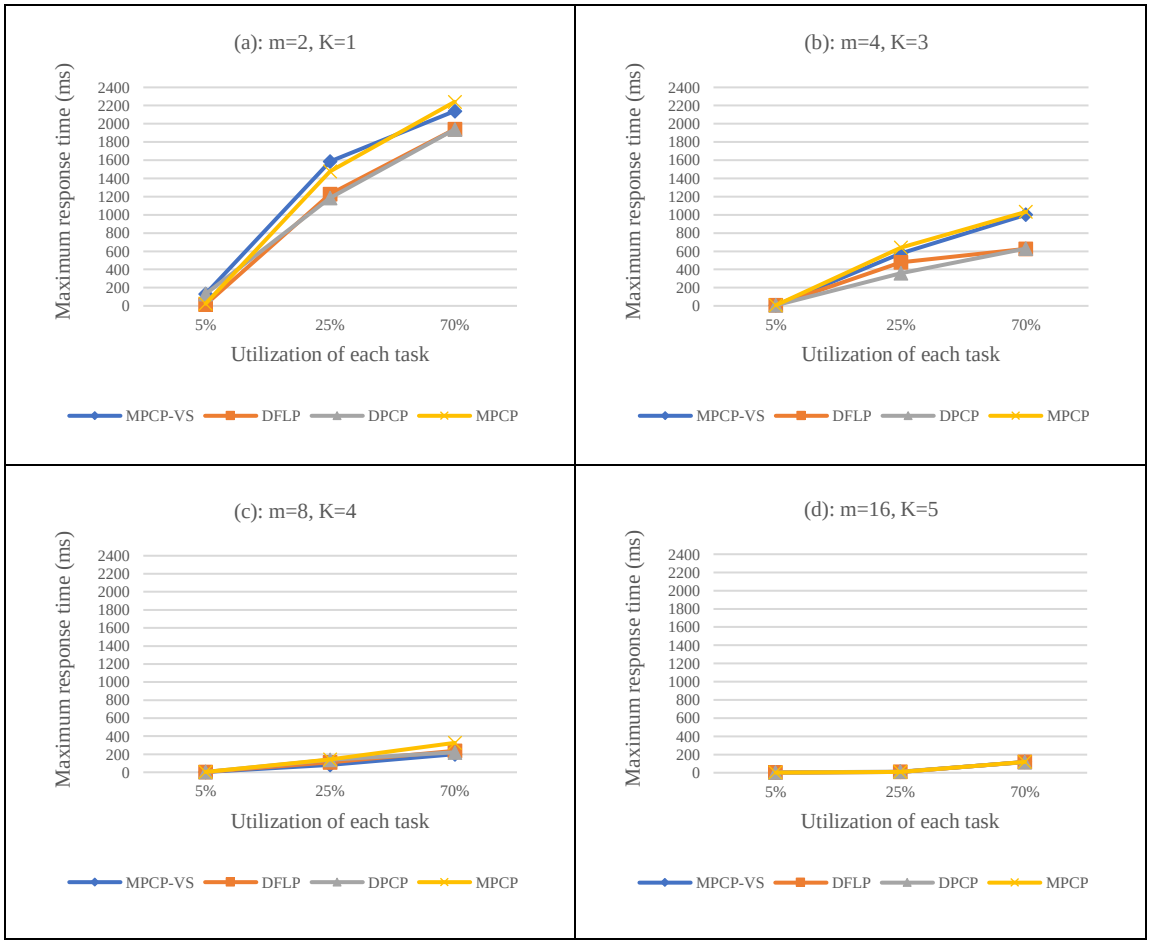


Figure 4.6: Maximum response time results obtained by executing of tasks with the ratio of the total cs length to the total non-cs length per task-set equal to 10%

4.2.3. Average Response Time Evaluation

In this section, average response time per task-set is taken into consideration.

The subgraphs in Figure 4.7 and Figure 4.8 illustrate results obtained by executing of tasks with low and medium utilization as well as the number of resource accesses per task $K = 1$ to 5 resources on $m = \{2, 4, 8, 16\}$ processors. The x-axis of each subgraph in those figures gives the ratio of the total length of critical sections (cs) to the total length of non-critical sections (non-cs). Each graph is resulted from the running of 20 task-sets.

The results of all subgraphs in Figure 4.7, besides graph (a), are roughly the same. However, by downscaling of the graphs (b), (c) and (d), it is realized that, the average response time per task-set will be decreased by increasing the number of processors. Moreover, if a system with more than 4 processors and the task utilization equal to 5% is in use, it is better to use a spin-based locking protocol generally, else applying one of the suspension-based locking protocols, specially the DPCP and the DFLP, can be a proper solution to reach less average response time for each task-set. Other results that were obtained but not shown here (instead shown in Appendix) support these conclusions.

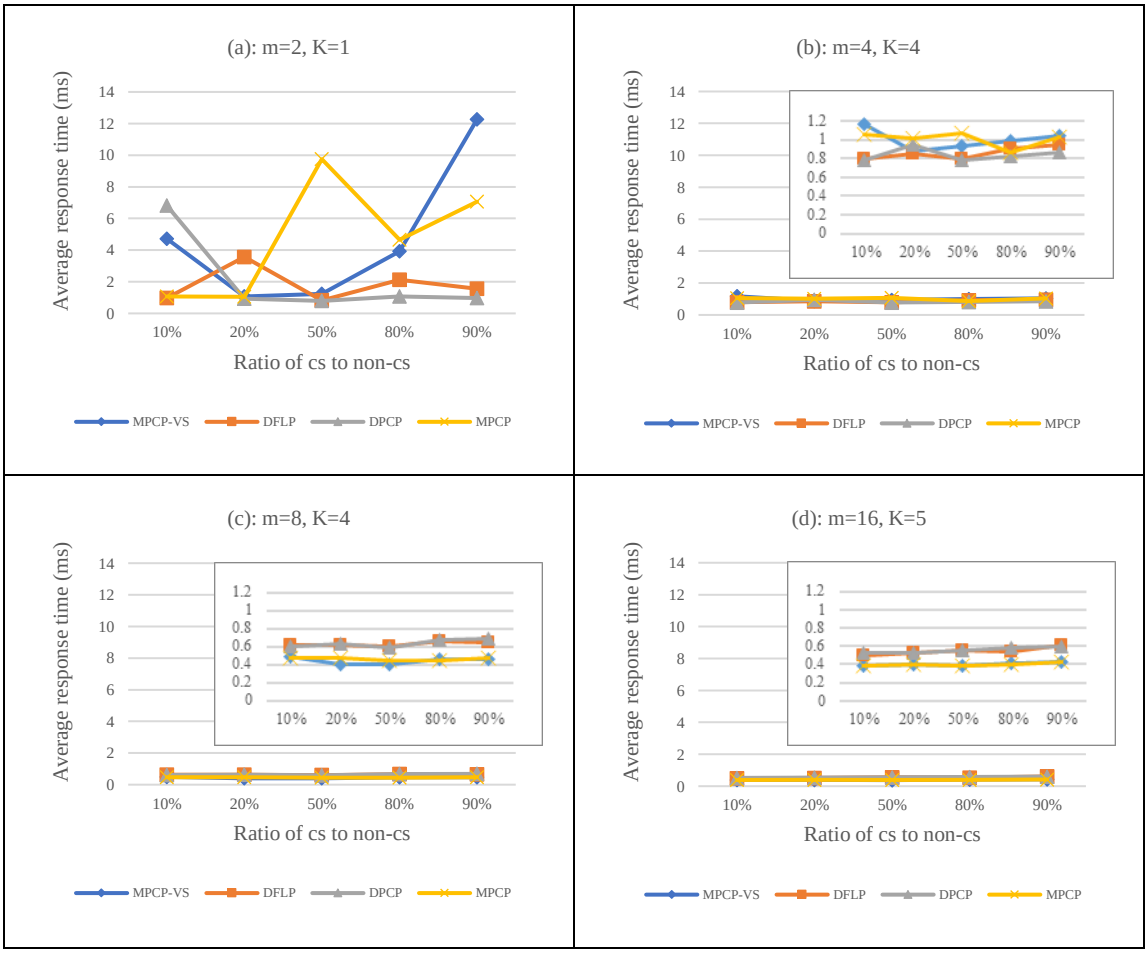


Figure 4.7: Average response time results obtained by executing of tasks with low utilization

The investigation of average response time per task-set will be continued in Figure 4.8 but here the utilization is increased to 25%.

Subgraph (a) in Figure 4.8 presents results obtained for $m = 2$ and $K = 1$. Without regard to exceptions, there are several factors to notice after analyzing this graph. First, the longest average response time per task-set is obtained if the suspension-based variant of the MPCP is used as the locking protocol. Second, if one of the suspension-based protocols is used, by increasing the ratio of the total cs length to the total non-cs length, the average response time per task-set is increased, otherwise the average response time tends to decrease.

Subgraph (b) in Figure 4.8 presents results obtained for $m = 4$ and $K = 2$. There are several factors to note. First, if the suspension-based variant of the MPCP is used as the locking protocol, then the longest average response time per task-set is obtained. Second, if one of the suspension-based protocols is used, by increasing the ratio of the total cs length to the total non-cs length, the average response time per task-set stays roughly the same, otherwise without regard to particulars, the average response time tends to decrease.

It seems that, the results of subgraph (c) and (d) in Figure 4.8 are the same. However, by downscaling them, it is realized that, if the spin-based variant of the MPCP is used as the locking protocol and the ratio of the total cs length to the total non-cs length is equal to or more than 50%, then the shortest average response time per task-set is obtained. Moreover, if one of the suspension-based protocols is used, by increasing the ratio of the total cs length to the total non-cs length, the average response time per task-set is increased totally, otherwise the average response time tends to decrease.

All in all, after comparing all subgraphs in Figure 4.8 with each other, it is discovered that, the average response time per task-set will be decreased by increasing the number of processors. In addition, if the used system has 4 or more processors and the task utilization equals 25%, it is generally better to use a spin-based locking protocol, else applying one of the suspension-based locking protocols can be a proper solution to reach less average response time for each task-set. Other results that were obtained but not shown here (instead shown in Appendix) support these conclusions.

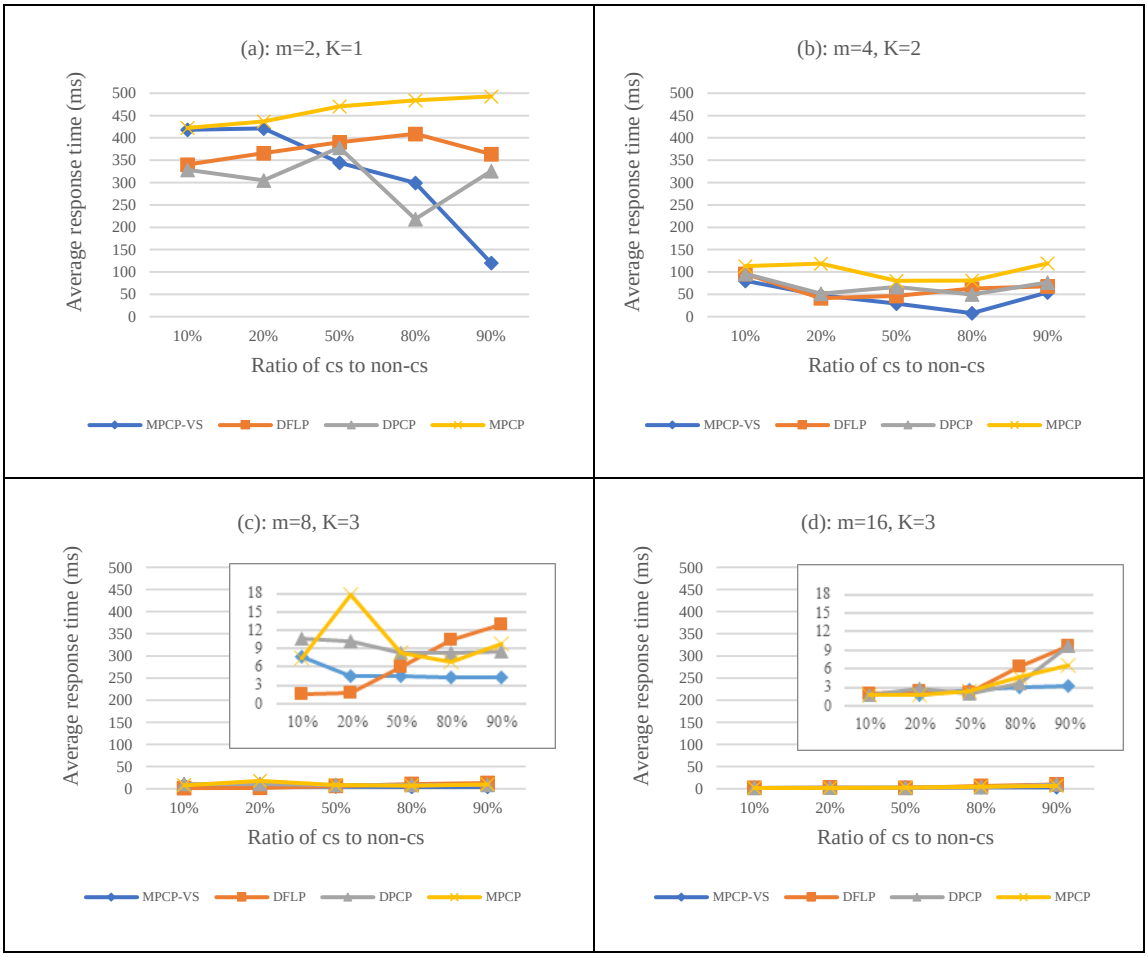


Figure 4.8: Average response time results obtained by executing of tasks with medium utilization

The subgraphs in Figure 4.9 illustrate results obtained by executing of tasks with the number of resource accesses per task $K = 1$ to 5 resources and the ratio of the total cs length to the total non-cs length per task-set equal to 10%. The x-axis of each subgraph in this figure gives the percentage of the task utilization. Each graph is resulted from the running of 12 task-sets.

The outcomes in subgraph (a) and (b) are almost the same, which illustrate results obtained for $m = 2, K = 1$ and $m = 4, K = 3$ respectively. There are several factors to notice after analyzing both graphs. First, as the utilization becomes higher, the average response time per task-set tends to increase, no matter which type of locking protocols is used. Second, the increment of the number of the resource accesses results the decrement of the average response time per task-set. Third, the average response time using the spin-based variant of the MPCP is shorter than the average response time using the suspension-based variant of the MPCP; Nonetheless, the shortest average response time per task-set is obtained using the DPCP or the DFLP, because the average response times of each task-set for both protocols are roughly identical.

The outcomes in subgraph (c) and (d) are also almost the same, which illustrate results obtained for $m = 8, K = 5$ and $m = 16, K = 2$ respectively. There are several factors to notice after analyzing both graphs. First, as the utilization is increased to 25%, the average response time per

task-set is not changed drastically, but if the utilization is more than 25%, the average response time tends to increase strongly. Second, by increasing the number of resource accesses, there is a slight decrease in the average response time per task-set. Third, the average response time using the spin-based variant of the MPCP is shorter than the average response time using the suspension-based variant of the MPCP; However, the shortest average response time per task-set is obtained using the DPCP or the DFLP, because the average response time of each task-set for both protocols are approximately the same.

All in all, after comparing all subgraphs in Figure 4.9 with each other, first it is realized that, the average response time per task-set will be decreased by increasing the number of processors. Second, if a system with more than 8 processors is in use, while the ratio of the total cs length to the total non-cs length per task-set is equal to 10%, it is better to use a spin-based locking protocol generally, else applying one of the suspension-based locking protocols can be a proper solution to reach less average response time for each task-set. Other results that were obtained but not shown here (instead shown in Appendix) support these conclusions.

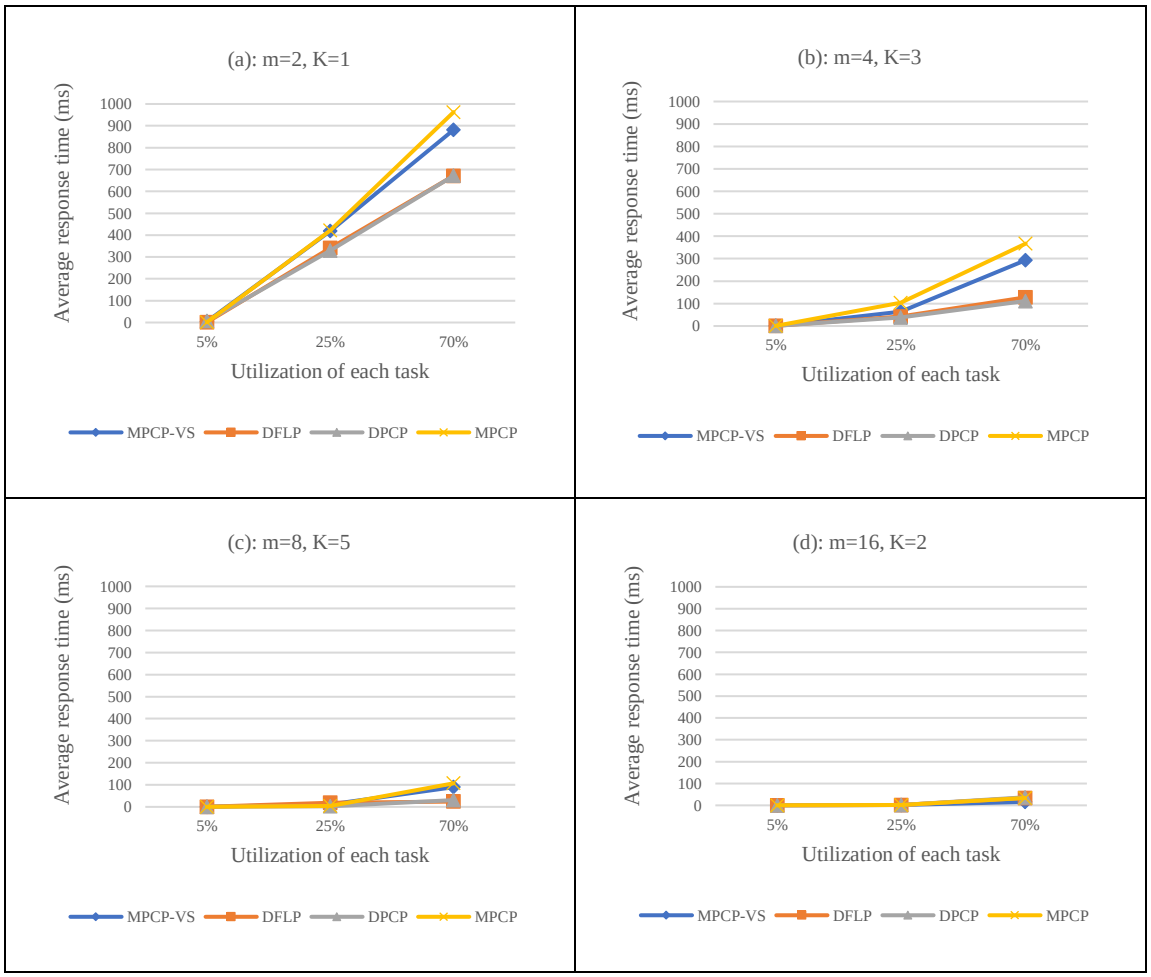


Figure 4.9: Average response time results obtained by executing of tasks with the ratio of the total cs length to the total non-cs length per task-set equal to 10%

5 Summary

With the emerge of multiprocessor technologies, they have started to be used widely in embedded platforms. Multiprocessor systems are referred as a set of processing units, which share usually some resources available in the system. As there are usually constraints on the number of resources in such systems, an undisciplined use of locking synchronization mechanism can severely degrade the performance of the system. Generally, there are two locking synchronization mechanisms, deemed spinning and suspending. Both approaches have their own positive and negative factors. For instance, under spinning mechanism, system encounters CPU time-wasting, i.e., the processor bandwidth is wasted through the task spinning to the processor, while, it is not the issue in the case of suspension mechanism, where the task waiting for the resource releases the processor and allows the other tasks to use it. However, by using suspension-based protocols, system suffers higher context switch overheads in comparison with the situation, in which the spin-based protocols are in use. Hence, in order to improve the performance of the multiprocessor systems, it is a matter of deciding under which circumstance, which synchronization mechanisms should be used. Such decision is the conclusion of this master thesis, which is argued next. Furthermore, some future works are discussed in the last section of this chapter.

5.1. Conclusion

In this work, the performance of both locking synchronization approaches was compared under different settings. The result evaluation study was designed with the aim to introduce the conditions under which suspension mechanism is better than spin mechanism, and vice versa. Therefore, all the conclusions drawn in this section are based on the analyses and evaluations discussed in previous chapter. In general speaking, the work in this thesis leads to conclude that, neither one of the spin-based nor suspension-based protocols is completely preferable to the other. In other words, the results show that there is no single “best” locking synchronization mechanism in practice. Rather, locking protocol performance is highly workload-dependent (e.g. task utilization, number of resources, number of processors, etc.).
The major conclusions of the study in this thesis are summarized as follows:

- Withstanding the schedulability evaluation argued in previous chapter, on one hand, suspension-based protocols are generally preferable for a task-set with low utilization (e.g. 5%) and number of resource accesses per task fewer than 4 resources running on a system with 4 or fewer processors, otherwise schedulability is likely to be poor under suspending approach and instead a spin-based protocol performs better. On the other hand, for a task-set with medium utilization (e.g. 25%), spinning approach is preferable only in the case that tasks run on a 2-processor system.

- Withstanding the analyses of the results related to the factor maximum response time, in general, spinning approach is preferable in the case that system contains 4 or more processors. In particular, suspension-based protocols perform as well or better than spin-based protocols for most tested workloads running on a system with fewer than 4 processors.
- Withstanding the analyses of the results pertained to the factor average response time, suspension-based blocking should be avoided under the case that task-set has low or medium utilization and the used system has 4 or more processors as well. Moreover, under the situation, where the ratio of the total length of critical sections to the total length of non-critical sections equals 10%, the use of spinning-based blocking will most likely not lead to appreciably less average response time in comparison with the situation, in which the suspension approach is in use, unless the number of processors is more than 8.
- Last but not least, distributed-based protocols, e.g. DPCP and DFLP, perform well for many workloads and outperform MPCP in the case of using suspension mechanism. Furthermore, regardless of exceptions, the spinning variant of MPCP shows a comparable performance as that of the suspension variant of MPCP with a better performance in some special cases.

5.2. Future Work

Although some conclusions are drawn in the concept of finding trade-off between suspending and spinning, there are still several directions conceivable for future work.

First, as the performance of resource sharing protocols highly depends on how the tasks are partitioned, other partition algorithms like *Resource-oriented partitioned scheduling* [64] can be used for task partitioning and then repeating the process of testing explained in this study. Besides changing task partition algorithm, different resource assignment rules in the case of distributed-based protocols can be applied.

Second, this study can be repeated again to compare performance of suspension-based protocols with non-preemptive spin-based locking protocols such as *Multiprocessor Slack Resource Policy (MSRP)* [2], instead of preemptive variant of spinning.

Finally, such trade-off between locking synchronization mechanisms can be determined based on dynamic scheduling too.

References

[1] L. Sha, R. Rajkumar and J.P. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization", *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175-1185, September 1990. doi: 10.1109/12.57058. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=57058&isnumber=2066.

[2] P. Gai, G. Lipari and M. Di Natale, "Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip", in *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001) (Cat. No.01PR1420)*, London, 2001. doi: 10.1109/REAL.2001.990598. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=990598&isnumber=21338.

[3] R. Rajkumar, L. Sha and J.P. Lehoczky, "Real-time synchronization protocols for multiprocessors", in *Proceedings. Real-Time Systems Symposium*, Huntsville, AL, 1988. doi: 10.1109/REAL.1988.51121. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=51121&isnumber=1866.

[4] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors", in *Proceedings.,10th International Conference on Distributed Computing Systems*, Paris, 1990. doi: 10.1109/ICDCS.1990.89257. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=89257&isnumber=2912.

[5] Björn B. Brandenburg, John M. Calandrino and Aaron Block, "Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin?", in *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, St. Louis, MO, 2008. doi: 10.1109/RTAS.2008.27. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4550805&isnumber=4550765.

[6] John M. Calandrino, Hennadiy Leontyev and Aaron Block, "LITMUS^RT : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers", in *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, Rio de Janeiro, 2006. doi: 10.1109/RTSS.2006.27. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4032341&isnumber=4032321.

[7] Brandenburg, B.B. and McKenney, J.H., "Scheduling and Locking in Multiprocessor Real-time Operating Systems", 2011.

[8] G. Buttazzo, Hard Real-Time Computing Systems, 3 ed., vol. 24, Springer US.

[9] John A. Stankovic and R. Rajkumar, "Real-Time Operating Systems", *Real-Time Systems*, vol. 28, no. 2-3, pp. 237-253, 2004. doi: 10.1023/B:TIME.0000045319.20260.73.

[10] S. Afshar, "Lock-Based Resource Sharing for Real-Time Multiprocessors," 2017.

[11] M. Pinedo, Scheduling : Theory, Algorithms, and Systems / M. Pinedo., 3 ed., Springer-Verlag, 2018.

[12] Andrea Bastoni, Bjorn B. Brandenburg and James H. Anderson, "An Empirical Comparison of Global, Partitioned, and Clustered Multiprocessor EDF Schedulers", in *31st IEEE Real-Time Systems Symposium*, San Diego, 2010. doi: 10.1109/RTSS.2010.23. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5702214&isnumber=5702212.

[13] Umamaheswari C. Devi, "Soft Real-Time Scheduling on Multiprocessors," Doctoral Dissertation, Chapel Hill, NC, 2006.

[14] Andrea Bastoni, Bjorn B. Brandenburg and James H. Anderson, "Is Semi-Partitioned Scheduling Practical?", in *23rd Euromicro Conference on Real-Time Systems*, Porto, 2011. doi: 10.1109/ECRTS.2011.20. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6001775&isnumber=6001639.

[15] Felipe Cerqueira, Manohar Vanga and Björn B. Brandenburg, "Scaling global scheduling with message passing", in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Berlin, 2014. doi: 10.1109/RTAS.2014.6926008. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6926008&isnumber=6925980.

[16] C.L. Liu and James W. Layland, "Scheduling Algorithms for Multiprogramming in Hard-Real-Time Environment", *Journal of the ACM*, vol. 20, no. 1, pp. 46-61, January 1973. doi: 10.1145/321738.321743. Available: http://doi.acm.org/10.1145/321738.321743.

[17] Sudarshan K. Dhall and C. L. Liu, "On a Real-Time Scheduling Problem", *Operations Research*, vol. 26, no. 1, pp. 127-140, February 1978. doi: 10.1287/opre.26.1.127.

[18] T.P. Baker, "Multiprocessor EDF and deadline monotonic schedulability analysis", in *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*, Mexico, 2003. doi: 10.1109/REAL.2003.1253260. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1253260&isnumber=28040.

[19] Marko Bertogna and Michele Cirinei, "Response-Time Analysis for Globally Scheduled Symmetric Multiprocessor Platforms", in *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, Tucson, 2007. doi: 10.1109/RTSS.2007.31. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4408300&isnumber=4408278.

[20] Robert I. Davis and Alan Burns, "A survey of hard real-time scheduling for multiprocessor systems", *ACM Computing Surveys (CSUR)*, vol. 43, no. 4, pp. 35:1--35:44, October 2011. doi: 10.1145/1978802.1978814. Available: http://doi.acm.org/10.1145/1978802.1978814.

[21] Sanjoy Baruah, "Techniques for Multiprocessor Global Schedulability Analysis", in *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, Tucson, 2007. doi: 10.1109/RTSS.2007.35. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4408297&isnumber=4408278.

[22] Sanjoy Baruah and Theodore Baker, "Schedulability analysis of global EDF", *Real-Time Systems,* vol. 38, no. 3, p. 223–235, April 2008. doi: s11241-007-9047-9. Available: https://doi.org/10.1007/s11241-007-9047-9.

[23] Michael R. Garey and David S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, vol. 24, 1982.

[24] Dong-Ik Oh and T.P. Bakker, "Utilization Bounds for N-Processor Rate Monotone Scheduling with Static Processor Assignment", *Real-Time Systems,* vol. 15, no. 2, p. 183–192, September 1998. doi: A:1008098013753. Available: https://doi.org/10.1023/A:1008098013753.

[25] B. Andersson, S. Baruah and J. Jonsson, "Static-priority scheduling on multiprocessors", in *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001) (Cat. No.01PR1420)*, London, 2001. doi: 10.1109/REAL.2001.990610. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=990610&isnumber=21338.

[26] J. M. López, J. L. Díaz and D. F. García, "Utilization Bounds for EDF Scheduling on Real-Time Multiprocessor Systems", *Real-Time Systems,* vol. 28, no. 1, p. 39–68, October 2004. doi: B:TIME.0000033378.56741.14. Available: https://doi.org/10.1023/B:TIME.0000033378.56741.14.

[27] "Erika enterprise OS," June 2016. [Online]. Available: http://erika.tuxfamily.org/drupal/.

[28] Nan Guan, Martin Stigge and Wang Yi, "Fixed-Priority Multiprocessor Scheduling with Liu and Layland's Utilization Bound", in *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium,* Stockholm, 2010. doi: 10.1109/RTAS.2010.39. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5465974&isnumber=5465951.

[29] Insik Shin and Arvind Easwaran, "Hierarchical Scheduling Framework for Virtual Clustering of Multiprocessors", in *2008 Euromicro Conference on Real-Time Systems,* Prague, 2008. doi: 10.1109/ECRTS.2008.28. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4573114&isnumber=4573090.

[30] John M. Calandrino, James H. Anderson and Dan P. Baumberger, "A Hybrid Real-Time Scheduling Approach for Large-Scale Multicore Platforms", in *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*, Pisa, 2007. doi: 10.1109/ECRTS.2007.81. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4271698&isnumber=4271665.

[31] N.C.Audsley, A.Burns and A.J.Wellings, "Deadline monotonic scheduling theory and application", *Control Engineering Practice,* vol. 1, no. 1, pp. 71 - 78, February 1993. doi: https://doi.org/10.1016/0967-0661(93)92105-D. Available: http://www.sciencedirect.com/science/article/pii/096706619392105D.

[32] Krishna, Real Time Systems, McGraw-Hill Education (India) Pvt Limited, 2010, 2010.

[33] Joseph Mattai, Real-Time Systems: Specification, Verification, and Analysis, Mathai Joseph, Ed., New Jersey: Prentice Hall PTR, 1995.

[34] Bjorn B. Brandenburg, "Scheduling and Locking in Multiprocessor Real-time Operating Systems," University of North Carolina at Chapel Hill, Chapel Hill, 2011.

[35] UmaMaheswari C. Devi, Hennadiy Leontyev and James H. Anderson, "Efficient Synchronization Under Global EDF Scheduling on Multiprocessors", in *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, Washington, 2006. doi: 10.1109/ECRTS.2006.10. Available: https://doi.org/10.1109/ECRTS.2006.10.

[36] P. Tsigas and Yi Zhang, "Non-blocking data sharing in multiprocessor real-time systems", in *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99 (Cat. No.PR00306)*, Hong Kong, 1999. doi: 10.1109/RTCSA.1999.811240. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=811240&isnumber=17590.

[37] Michel Raynal, Concurrent Programming: Algorithms, Principles, and Foundations, 1 ed., Springer-Verlag Berlin Heidelberg, 2013.

[38] E. W. Dijkstra, "Solution of a Problem in Concurrent Programming Control", *Communications of the ACM*, vol. 8, no. 9, p. 569, September 1965. doi: 10.1145/365559.365617. Available: http://doi.acm.org/10.1145/365559.365617.

[39] S.H. Unger, "Hazards, critical races, and metastability", *IEEE Transactions on Computers*, vol. 44, no. 6, pp. 754 - 768, Jun 1995. doi: 10.1109/12.391185. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=391185&isnumber=8861.

[40] Susan Davidson, Insup Lee and Victor Fay Wolfe, "Deadlock prevention in concurrent real-time systems", *Real-Time Systems*, vol. 5, no. 4, pp. 305-318, 1993. doi: 10.1007/BF01088833. Available: https://doi.org/10.1007/BF01088833.

[41] Alexander Wieder and Björn B. Brandenburg, "On Spin Locks in AUTOSAR: Blocking Analysis of FIFO, Unordered, and Priority-Ordered Spin Locks", in *2013 IEEE 34th Real-Time Systems Symposium*, Vancouver, 2013. doi: 10.1109/RTSS.2013.13. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6728860&isnumber=6728845.

[42] Alessandro Biondi, Björn B. Brandenburg and Alexander Wieder, "A Blocking Bound for Nested FIFO Spin Locks", in *2016 IEEE Real-Time Systems Symposium (RTSS)*, Porto, 2016. doi: 10.1109/RTSS.2016.036. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7809864&isnumber=7809829.

[43] Catherine E. Nemitz, Kecheng Yang and Ming Yang, "Multiprocessor Real-Time Locking Protocols for Replicated Resources", in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, Toulouse, 2016. doi: 10.1109/ECRTS.2016.29. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7557868&isnumber=7557852.

[44] Farhang Nemati, "Resource Sharing in Real-Time Systems on Multiprocessors," Mälardalen University, Västerås, 2012.

[45] Björn B. Brandenburg and James H. Anderson, "A Comparison of the M-PCP, D-PCP, and FMLP on LITMUSRT", in *International Conference On Principles Of Distributed Systems*, 2008. doi: https://doi.org/10.1007/978-3-540-92221-6_9.

[46] Edward G. Coffman Jr., János Csirik, Gábor Galambos, Silvano Martello and Daniele Vigo, "Bin Packing Approximation Algorithms: Survey and Classification", in *Handbook of Combinatorial Optimization*, P. M. Pardalos, D. Du and R. L. Graham, Eds., New York, Springer New York, 2013, pp. 455-531.doi: 10.1007/978-1-4419-7997-1_35. Available: https://doi.org/10.1007/978-1-4419-7997-1_35.

[47] D. S. Johnson, "Near-optimal bin packing algorithms," Massachusetts Institute of Technology, Massachusetts, 1973.

[48] D. Simchi-levi, "New worst-case results for the bin-packing problem", *Naval Research Logistics - NAV RES LOG*, vol. 41, pp. 579-585, 1994. doi: 10.1002/1520-6750(199406)41:43.0.CO;2-G.

[49] C. Kenyon, "Best-fit Bin-packing with Random Order", in *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, Atlanta, 1996. Available: http://dl.acm.org/citation.cfm?id=313852.314083.

[50] T. P. Baker, "A Comparison of Global and Partitioned EDF Schedulability Tests for multiprocessors", in *In International Conf. on Real-Time and Network Systems*, 2005.

[51] S. K. Baruah, N. K. Cohen, C. G. Plaxton and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation", *Algorithmica*, vol. 15, no. 6, p. 600–625, 01 Jun 1996. doi: 10.1007/BF01940883. Available: https://doi.org/10.1007/BF01940883.

[52] G. Levin, S. Funk and C. Sadowski, "DP-FAIR: A Simple Model for Understanding Optimal Multiprocessor Scheduling", in *2010 22nd Euromicro Conference on Real-Time Systems*, Brussels, 2010. doi: 10.1109/ECRTS.2010.34. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5562894&isnumber=5562824.

[53] H. Takada and K. Sakamura, "Predictable spin lock algorithms with preemption", in *Proceedings of 11th IEEE Workshop on Real-Time Operating Systems and Software*, Seattle, 1994. doi: 10.1109/RTOSS.1994.292571. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=292571&isnumber=7232.

[54] B. B. Brandenburg, "A Note on Blocking Optimality in Distributed Real-Time Locking Protocols," 2012.

[55] K. Lakshmanan, D. de Niz and R. Rajkumar, "Coordinated Task Scheduling, Allocation and Synchronization on Multiprocessors", in *2009 30th IEEE Real-Time Systems Symposium*, Washington, 2009. doi: 10.1109/RTSS.2009.51. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5368127&isnumber=5368116.

[56] R. Rajkumar, Synchronization in Real-Time Systems, 1 ed., vol. 151, Springer US, 1991.

[57] B. B. Brandenburg, "Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling", in *2013 IEEE 19th Real-Time and Embedded Technology and*

Applications Symposium (RTAS), Philadelphia, 2013. doi: 10.1109/RTAS.2013.6531087. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6531087&isnumber=6531071.

[58] "Real-time executive for multiprocessor systems (RTEMS)," [Online]. Available: https://www.rtems.org/.

[59] "QNX Neutrino RTOS," [Online]. Available: http://blackberry.qnx.com/en/products/neutrino-rtos/neutrino-rtos.

[60] B. B. Brandenburg, J. M. Calandrino, H. Leontyev, A. D. Block, U. Devi and J. Anderson, "LITMUS RT: A status report", in *Proceedings of the 9th Real-Time Linux Workshop*, 2007.

[61] B. Brandenburg and J. Anderson, "Feather-trace: A light-weight event tracing toolkit", in *In Proceedings of the Third International Workshop on Operating Systems Platformsfor Embedded Real-Time Applications*, 2007.

[62] "Linux Testbed for Multiprocessor Scheduling in Real-Time Systems," [Online]. Available: https://www.litmus-rt.org/.

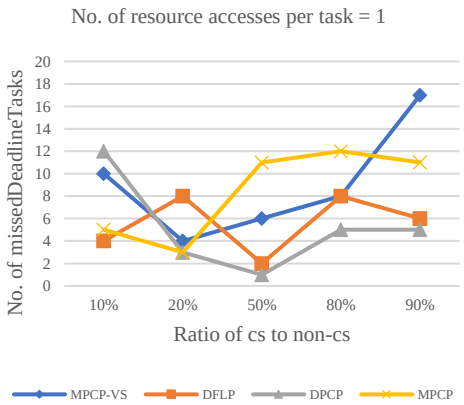
[63] P. Emberson, R. Stafford and R.I. Davis, "Techniques For The Synthesis Of Multiprocessor Tasksets", in *WATERS workshop at the Euromicro Conference on Real-Time Systems*, 2010.

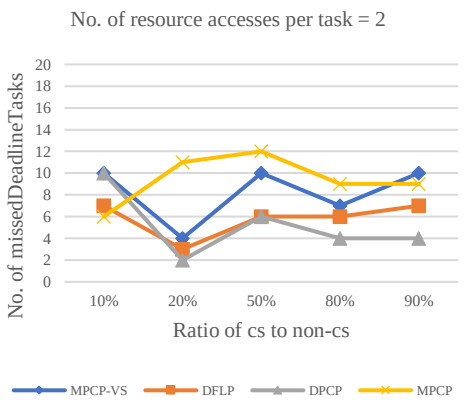
[64] Wen-Hung Huang, M. Yang and J.-J. Chen, "Resource-Oriented Partitioned Scheduling in Multiprocessor Systems: How to Partition and How to Share?", in *2016 IEEE Real-Time Systems Symposium (RTSS)*, Porto, 2016. doi: 10.1109/RTSS.2016.020. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7809848&isnumber=7809829.

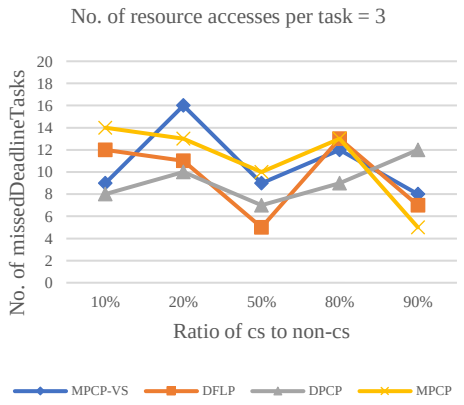
[65] Nathan Wayne Fisher, "The Multiprocessor Real-Time Scheduling of General Task Systems," University of North Carolina at Chapel Hill, 2007.

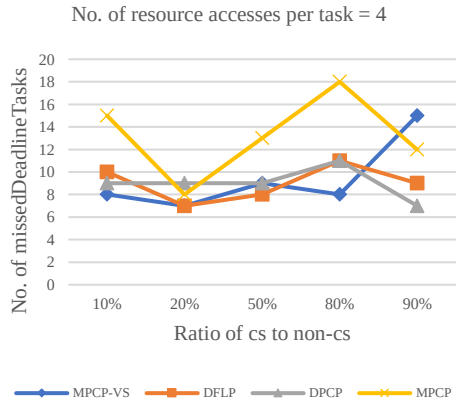
Appendix: List of Graphs

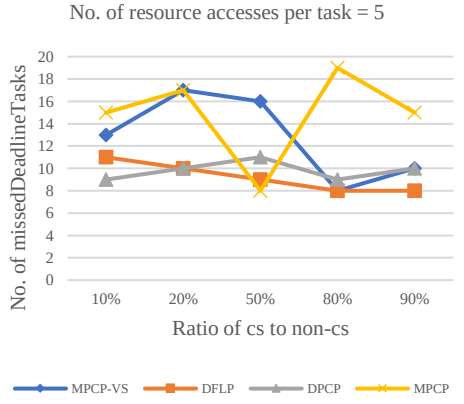
Utilization = 5%, Number of processors = 2



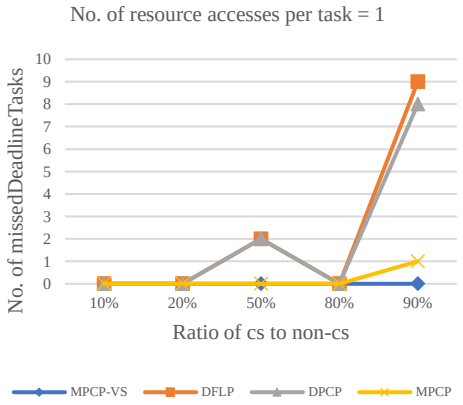


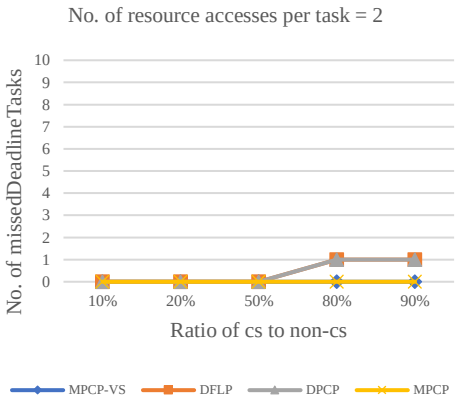


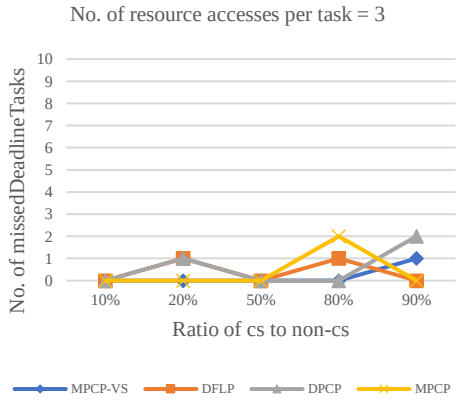


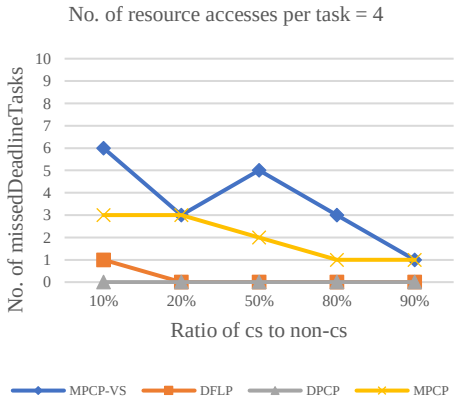


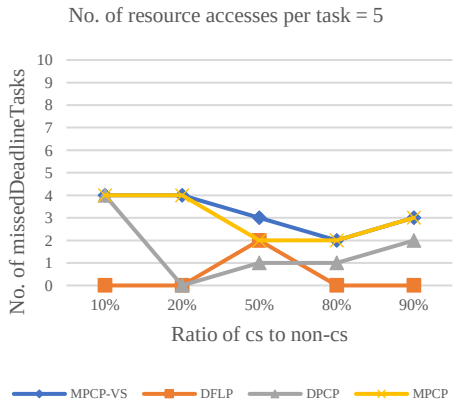
Utilization = 5%, Number of processors = 4



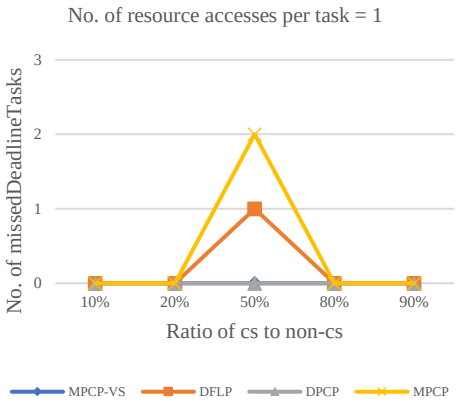


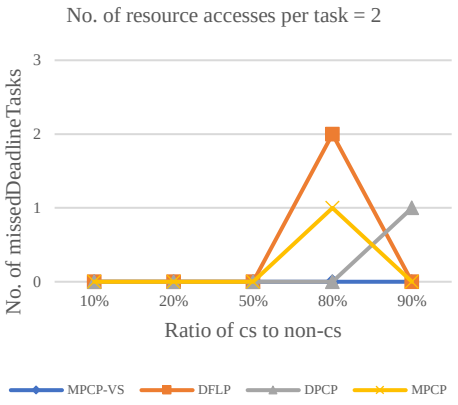


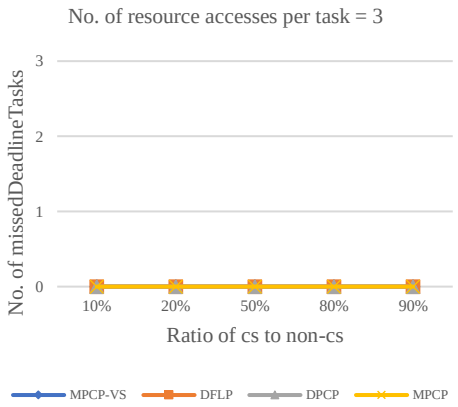


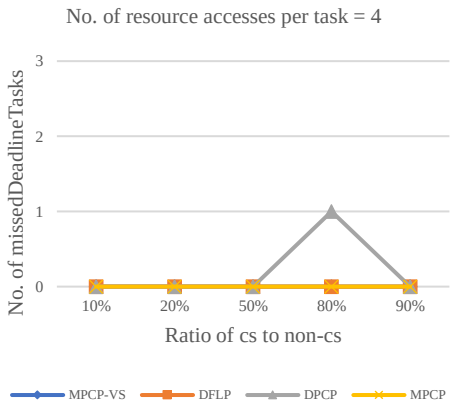


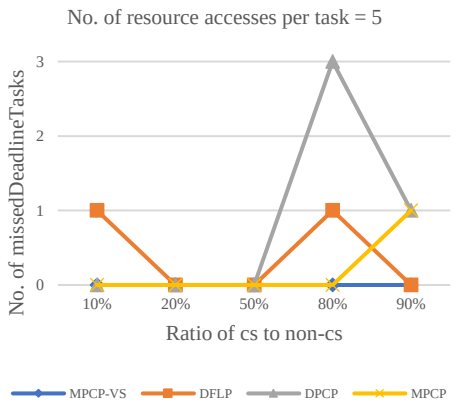
Utilization = 5%, Number of processors = 8



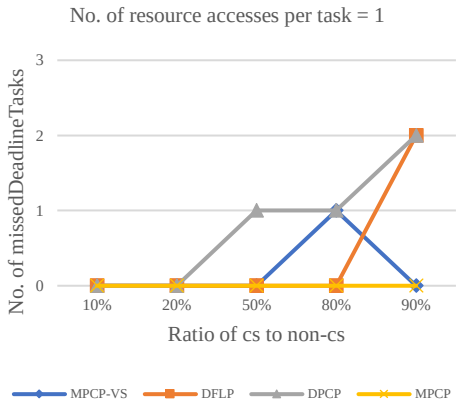


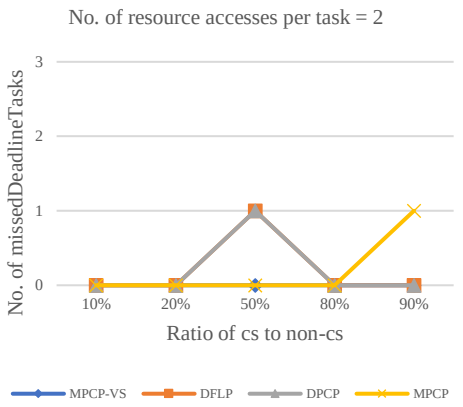


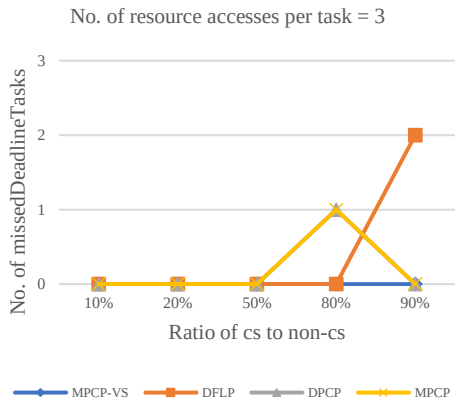


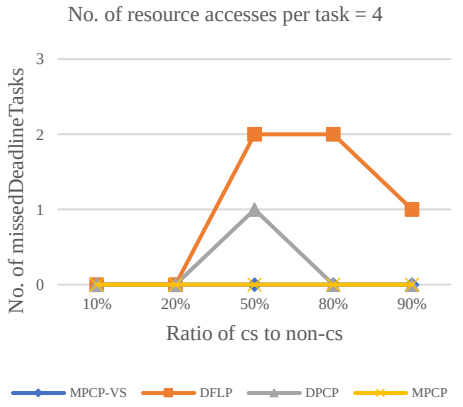


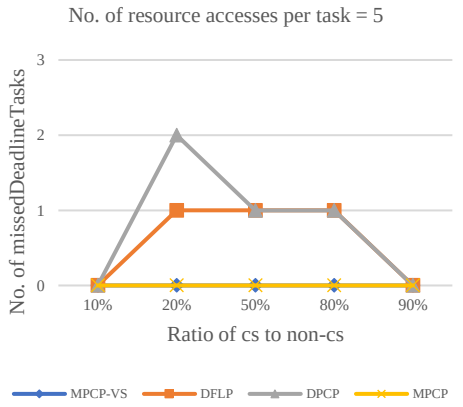
Utilization = 5%, Number of processors = 16



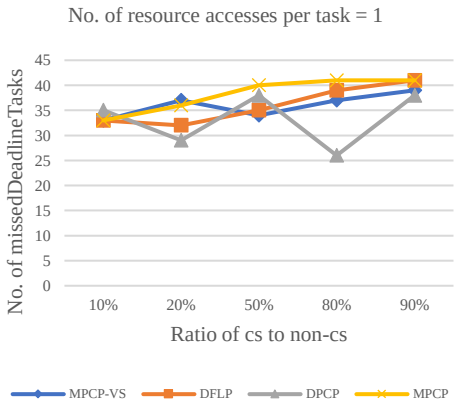


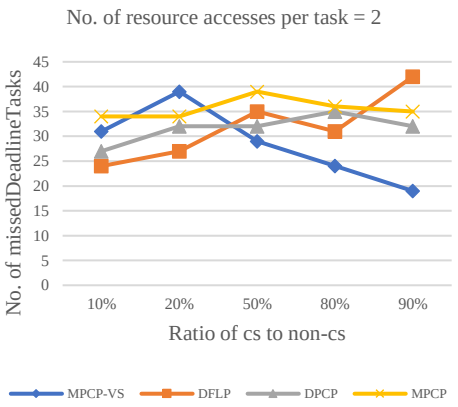


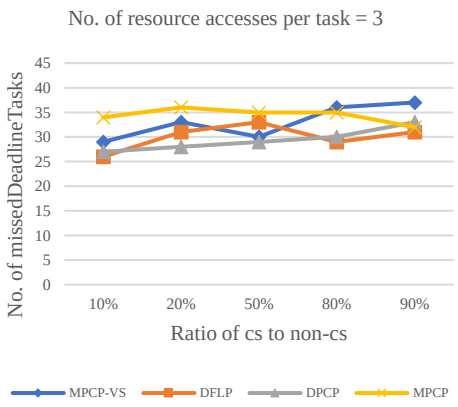


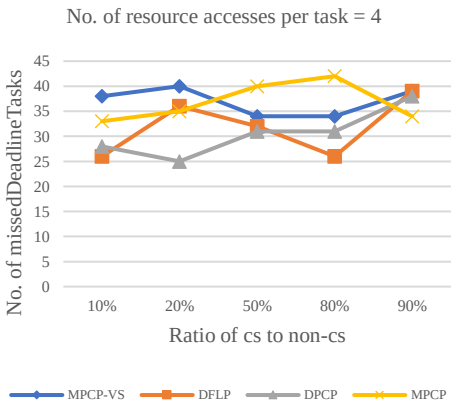


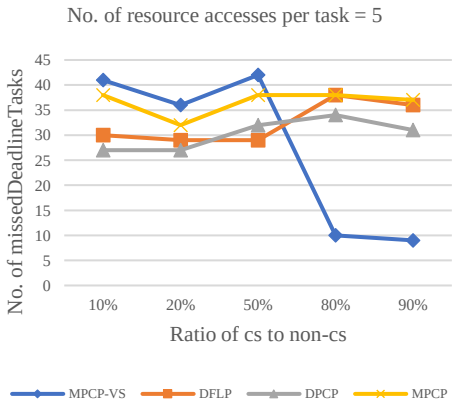
Utilization = 25%, Number of processors = 2



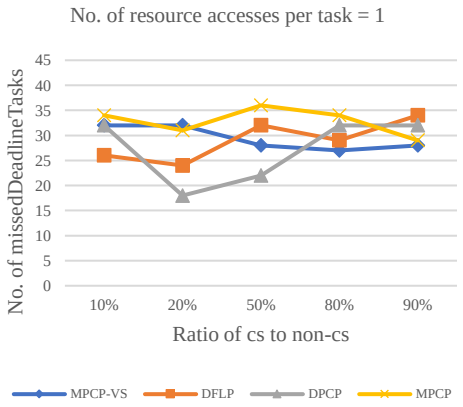


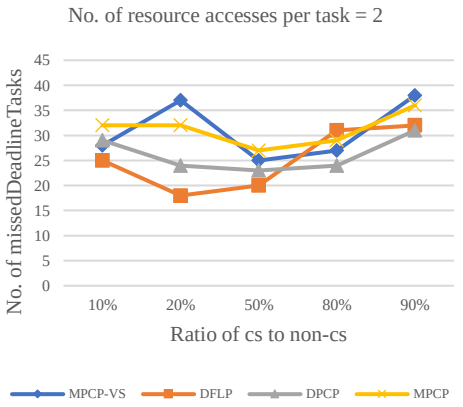


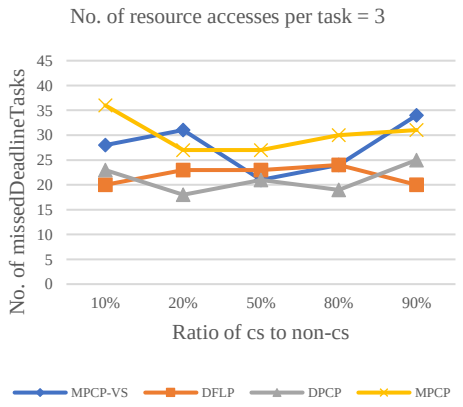


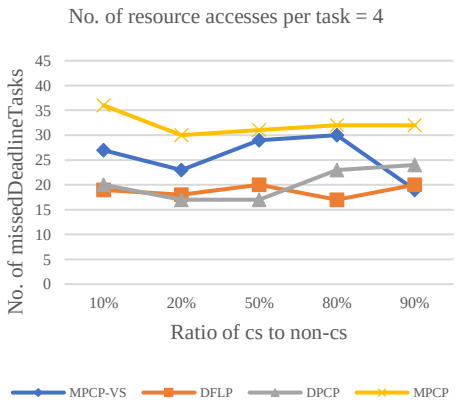


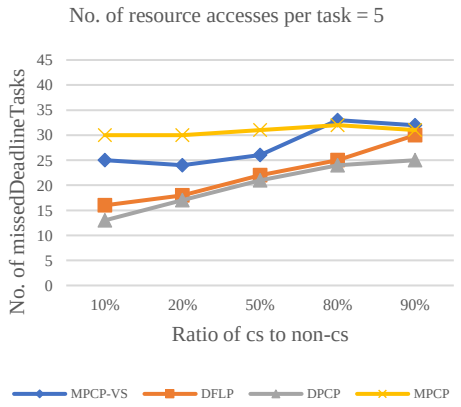
Utilization = 25%, Number of processors = 4



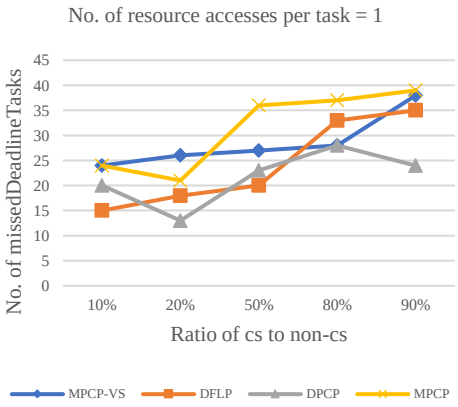


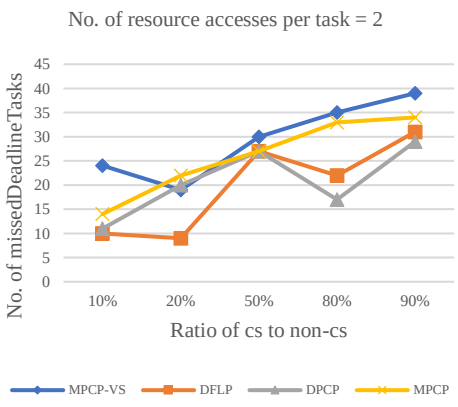


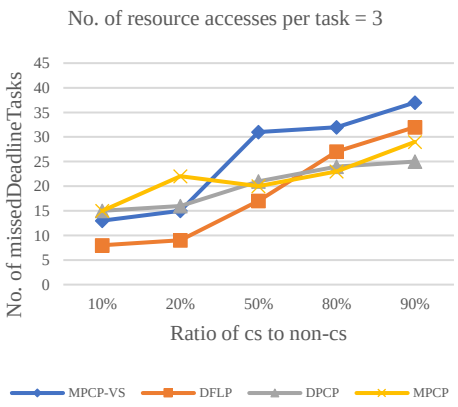


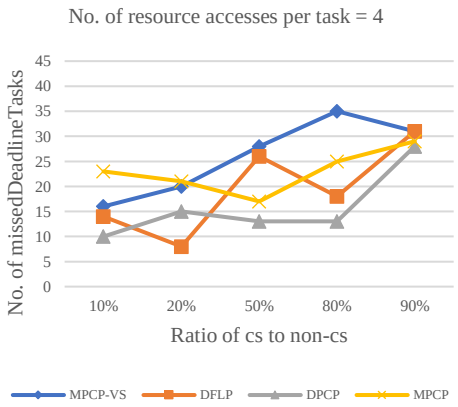


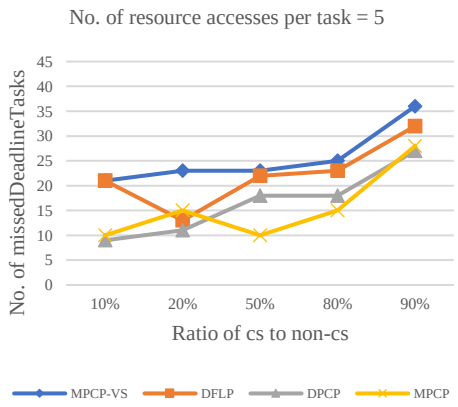
Utilization = 25%, Number of processors = 8



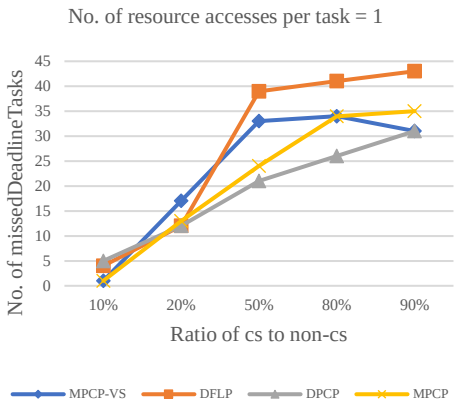


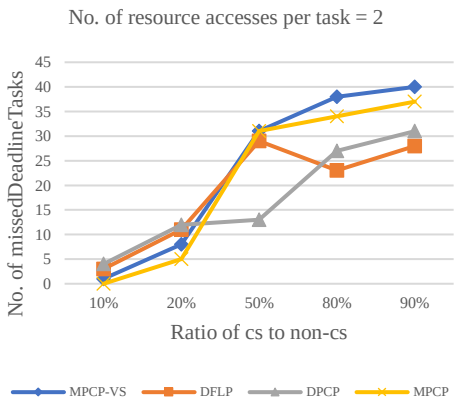


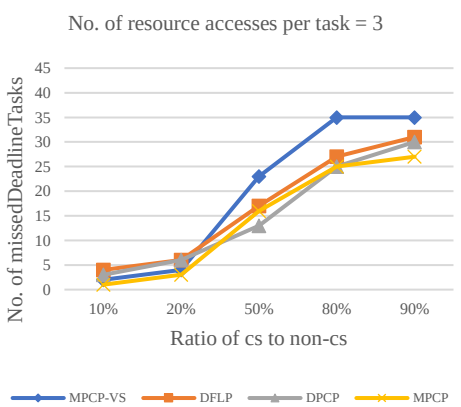


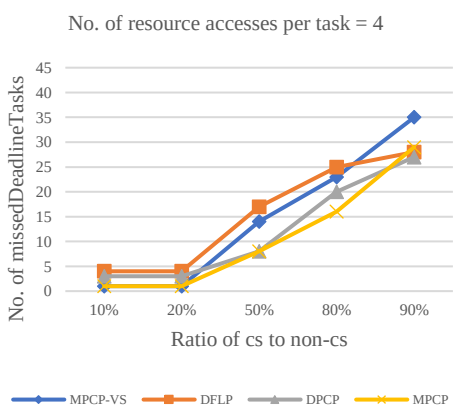


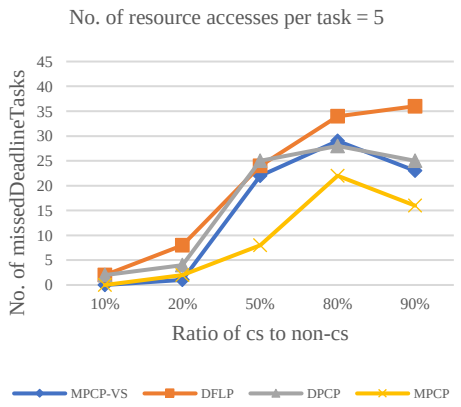
Utilization = 25%, Number of processors = 16



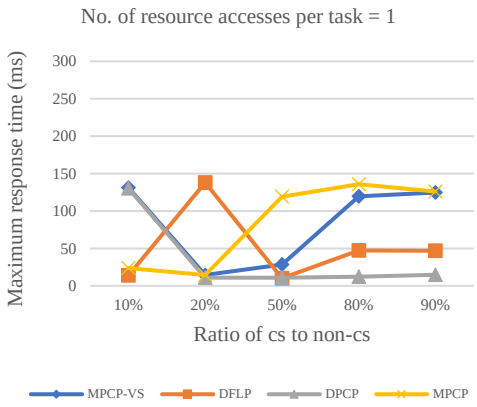


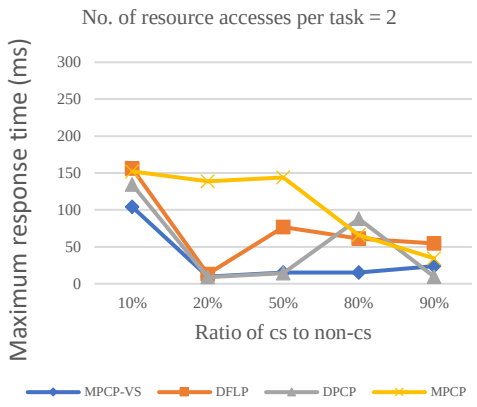


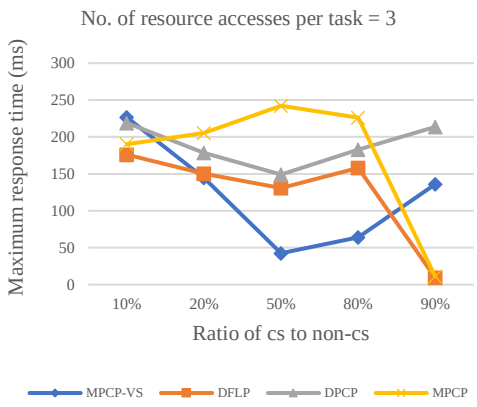


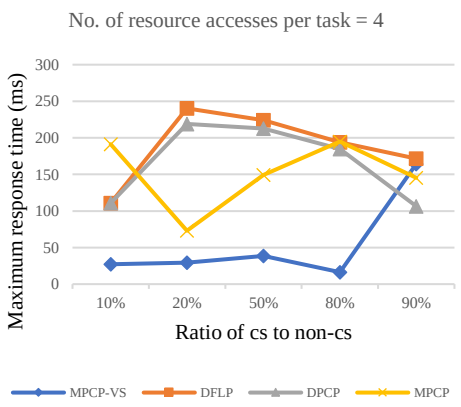


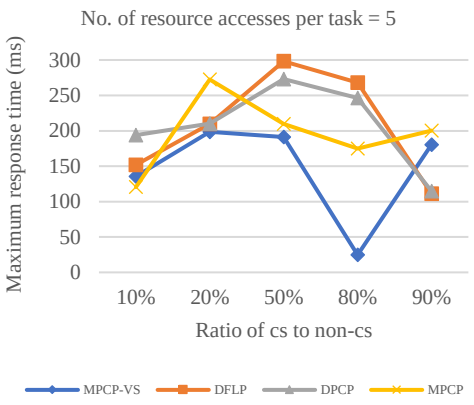
Utilization = 5%, Number of processors = 2



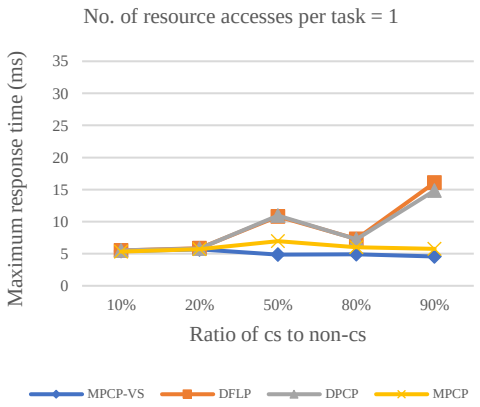


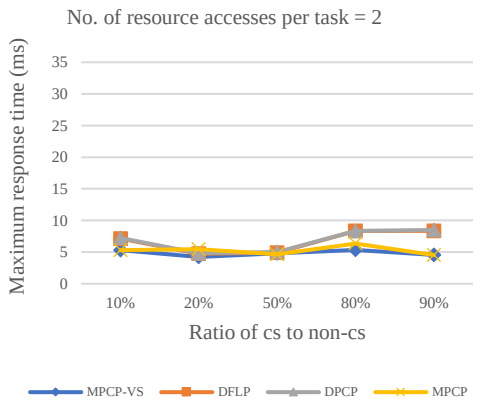


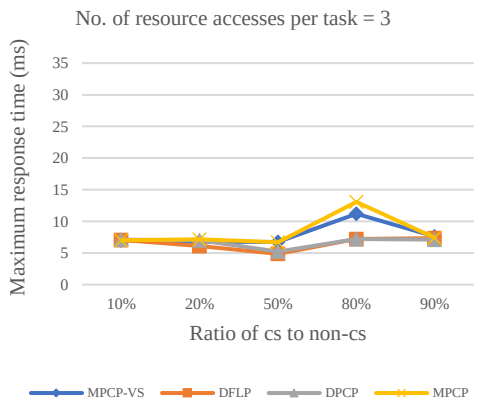


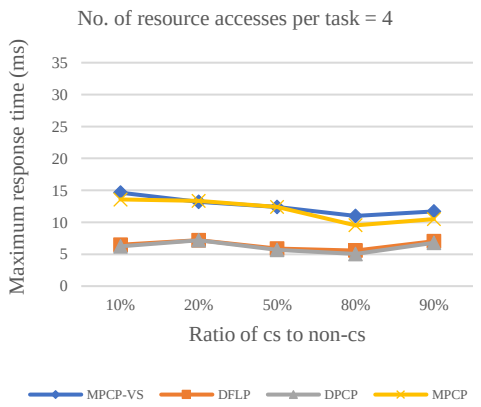


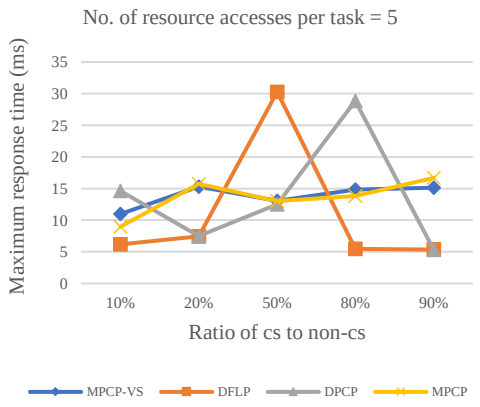
Utilization = 5%, Number of processors = 4



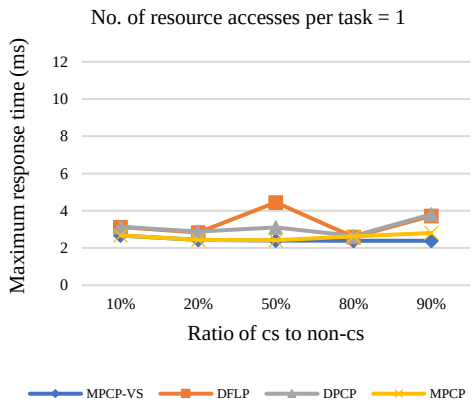


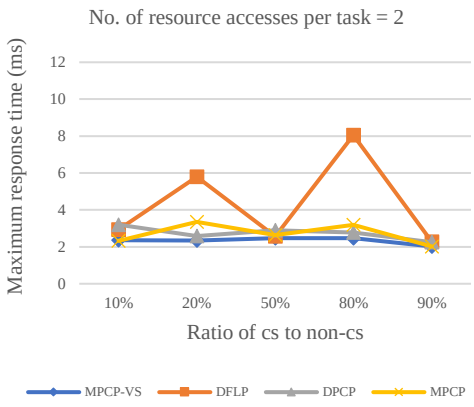


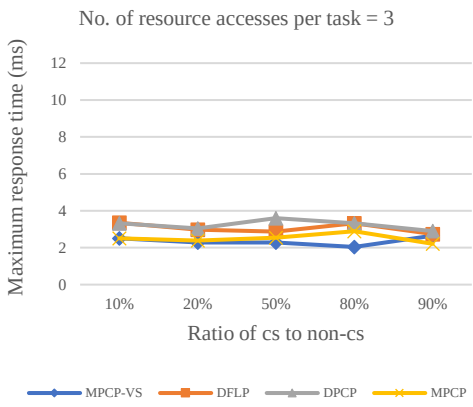


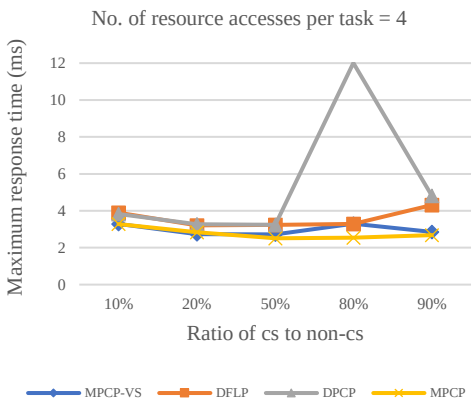


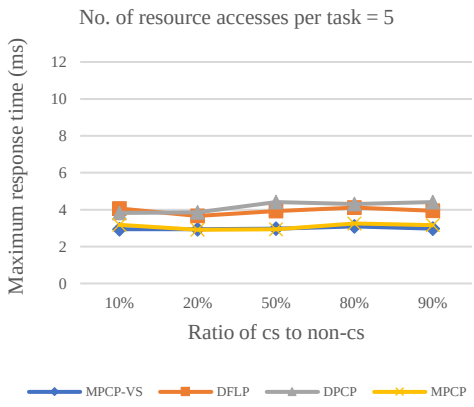
Utilization = 5%, Number of processors = 8



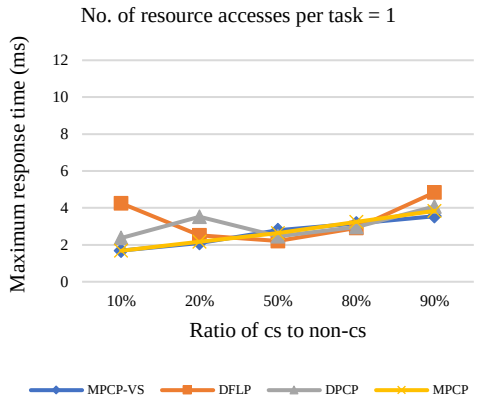


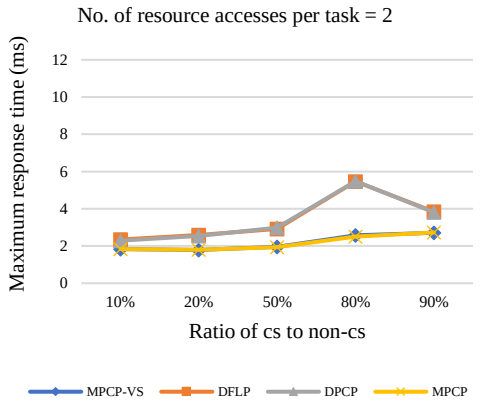


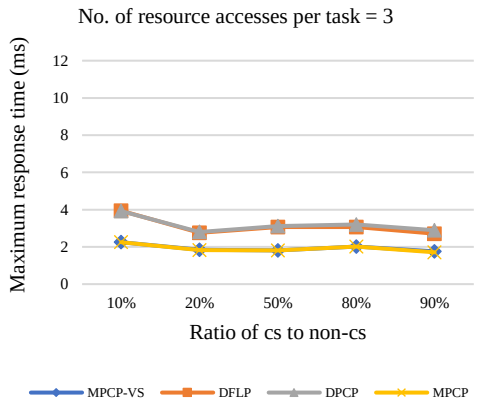


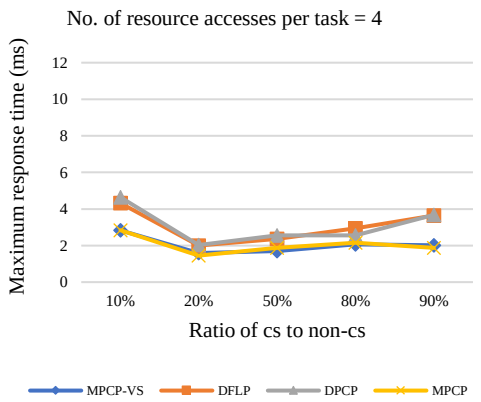


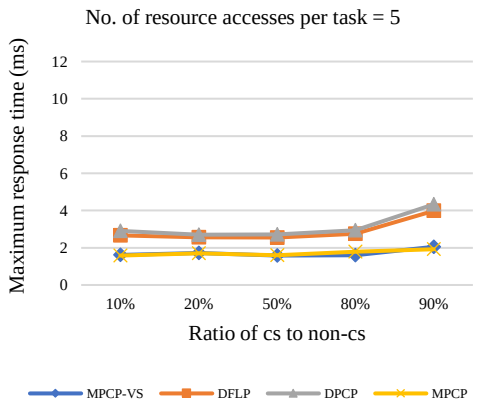
Utilization = 5%, Number of processors = 16



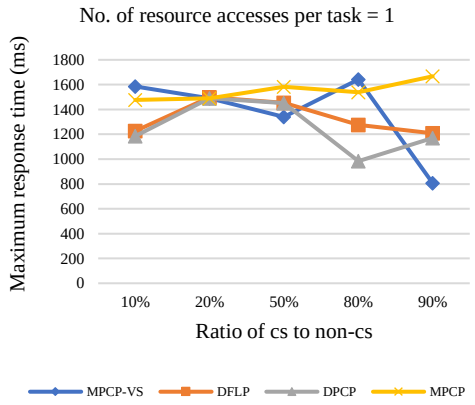


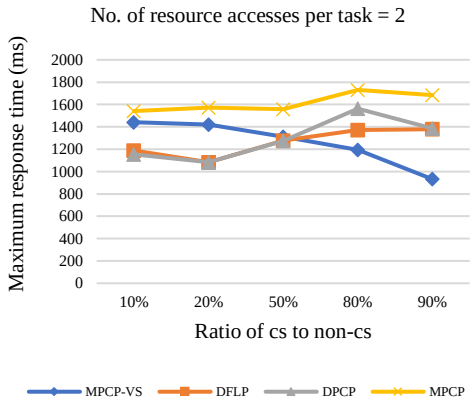


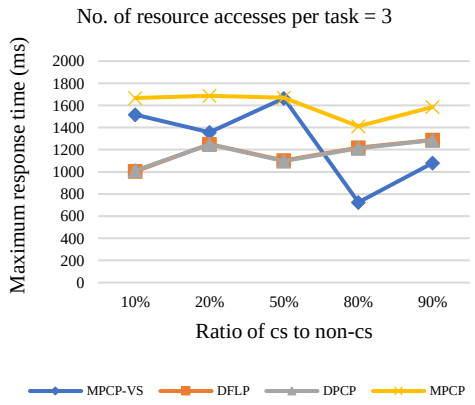


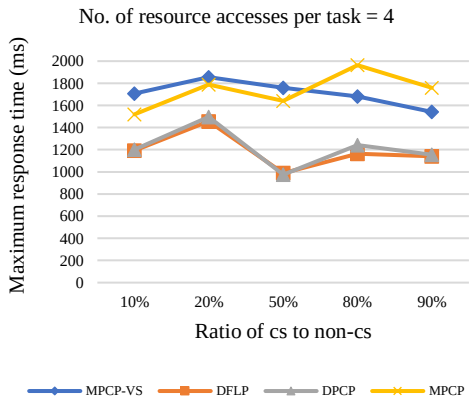


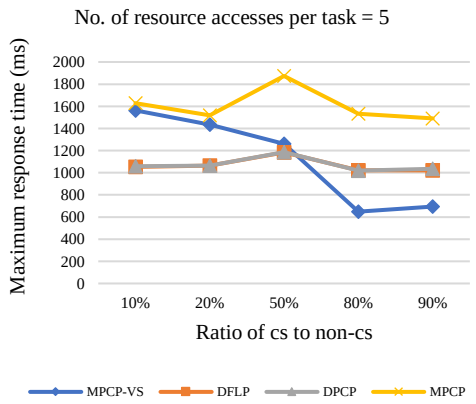
Utilization = 25%, Number of processors = 2



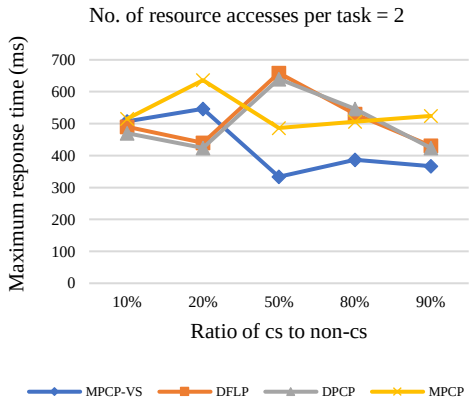


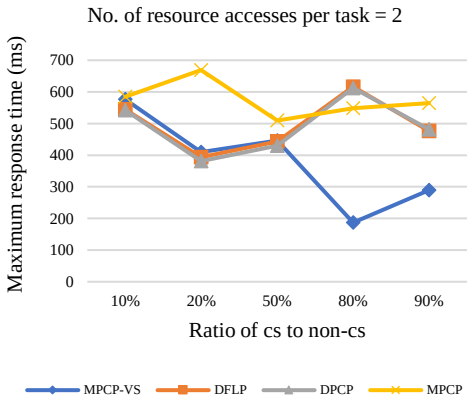


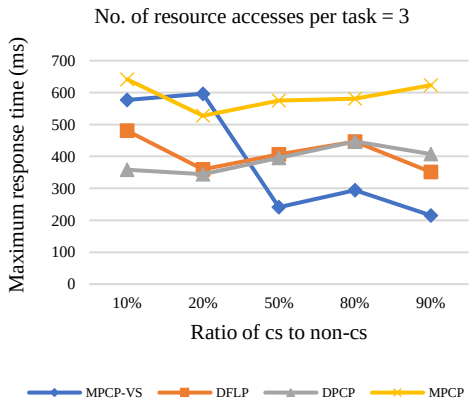


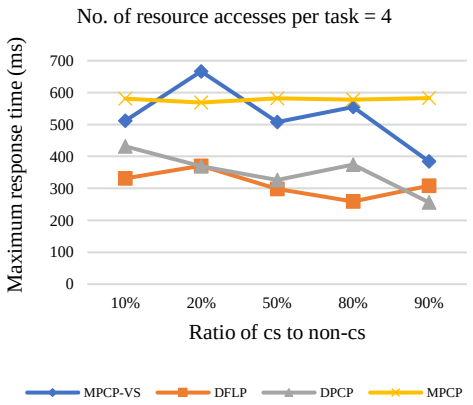


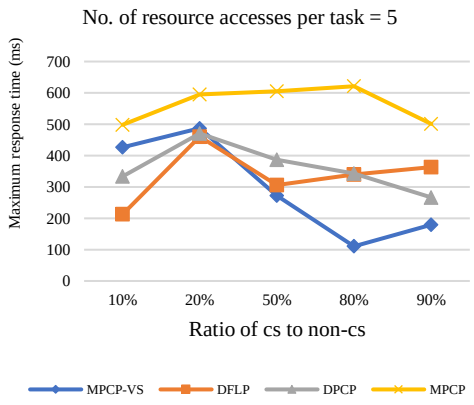
Utilization = 25%, Number of processors = 4



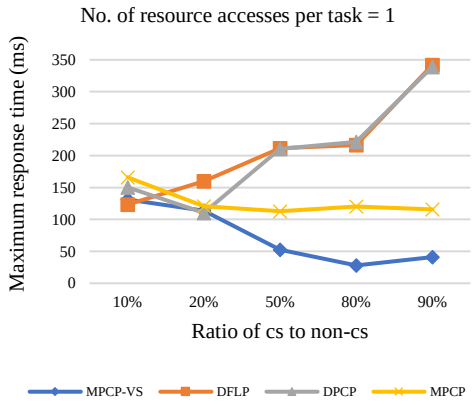


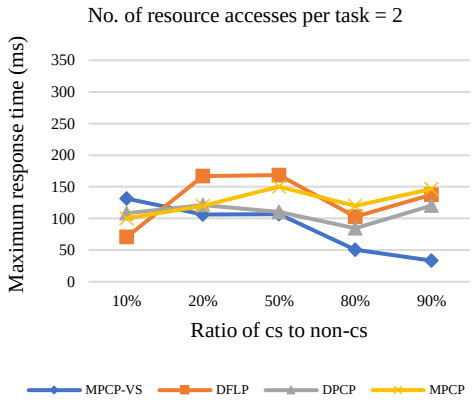


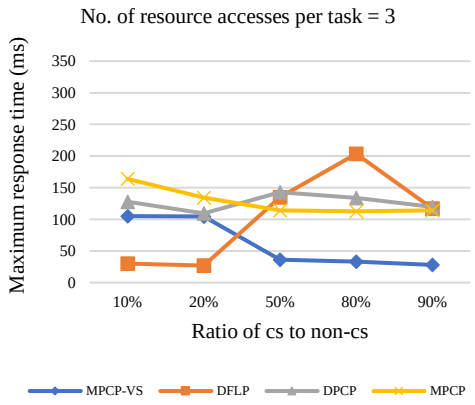


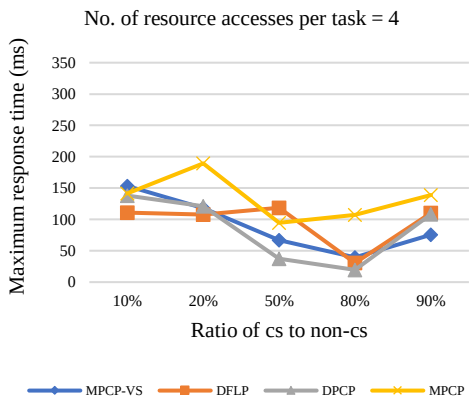


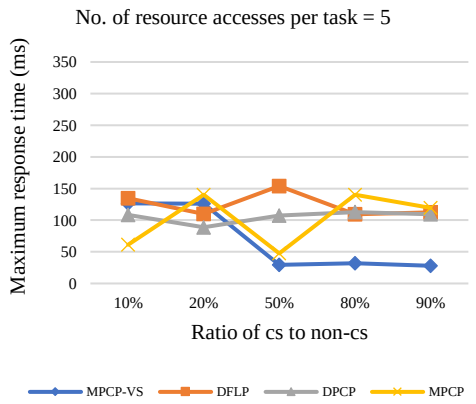
Utilization = 25%, Number of processors = 8



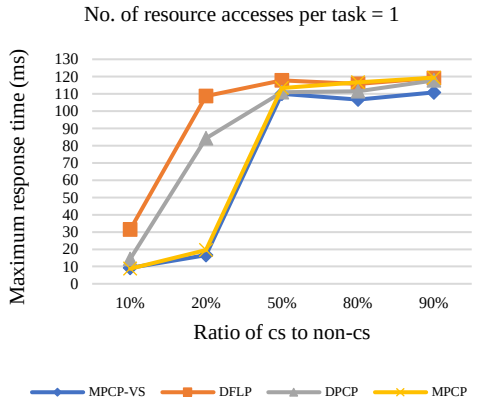


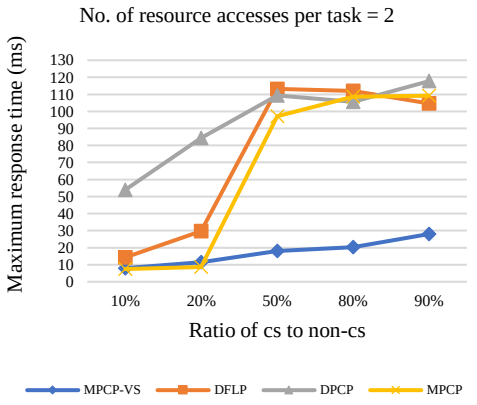


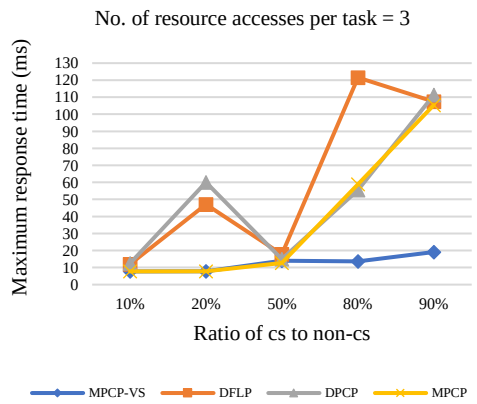


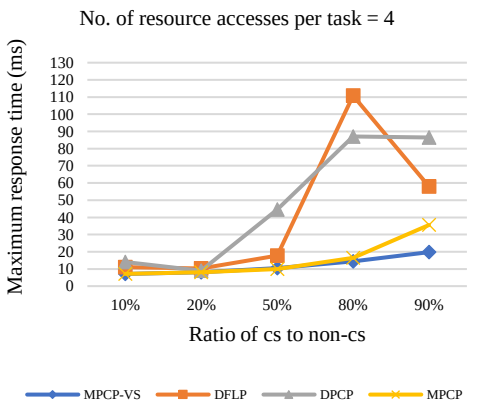


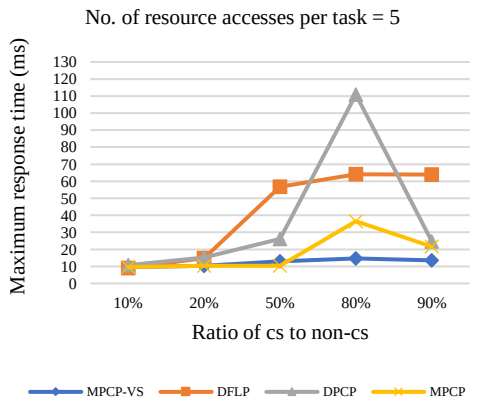
Utilization = 25%, Number of processors = 16



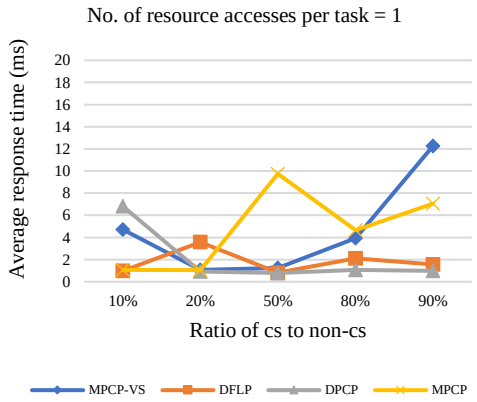


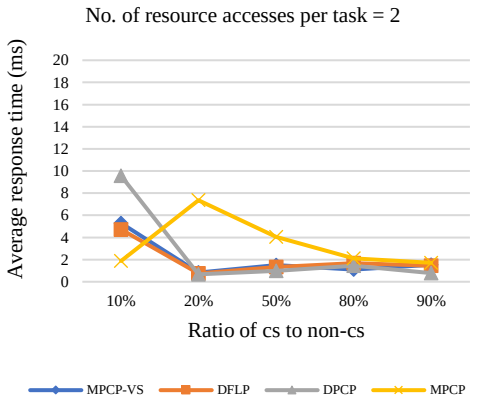


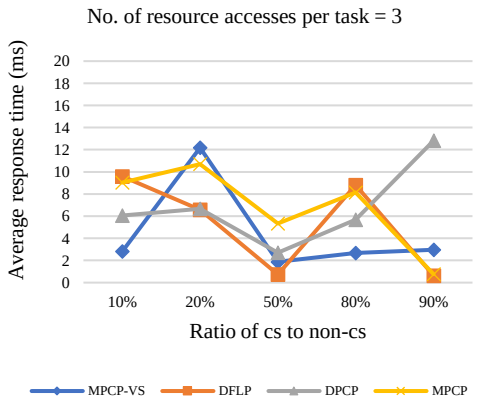


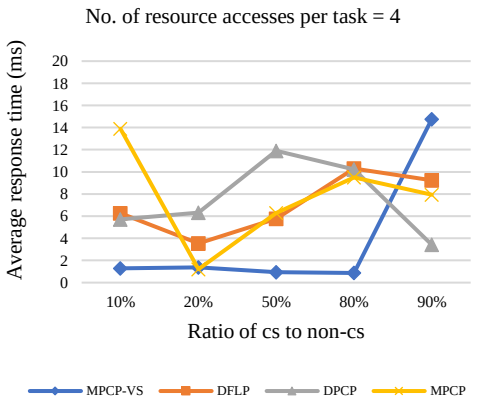


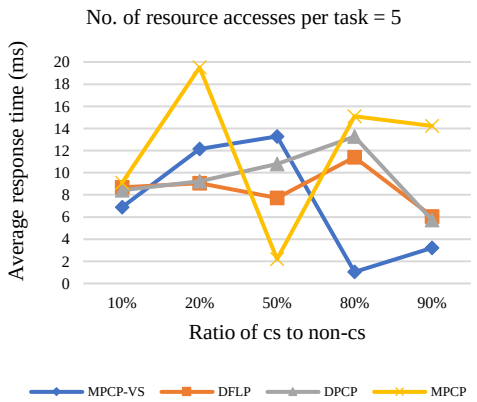
Utilization = 5%, Number of processors = 2



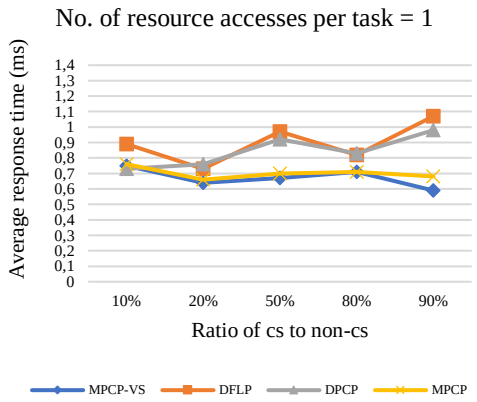


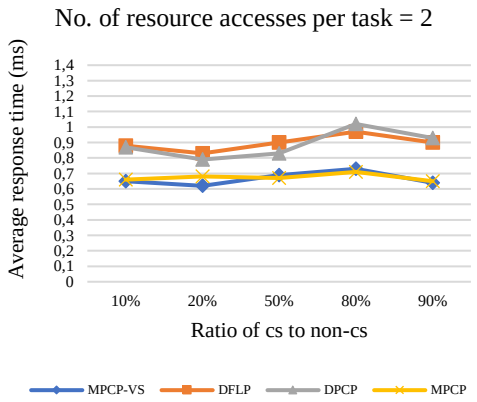


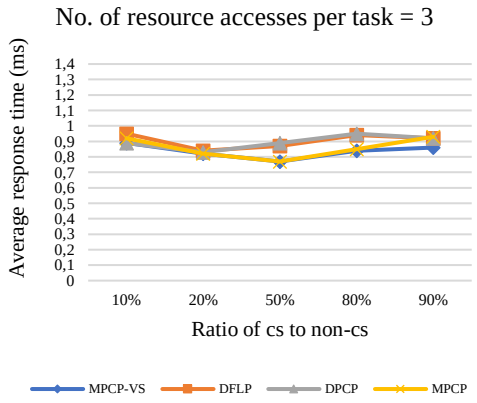


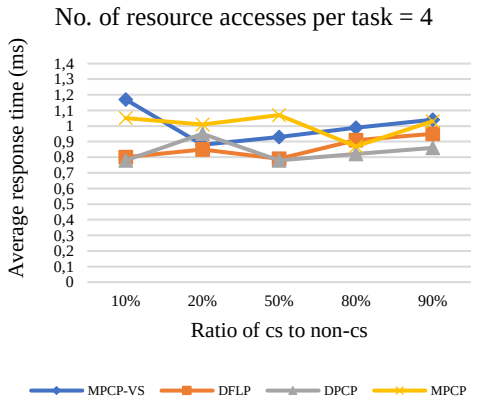


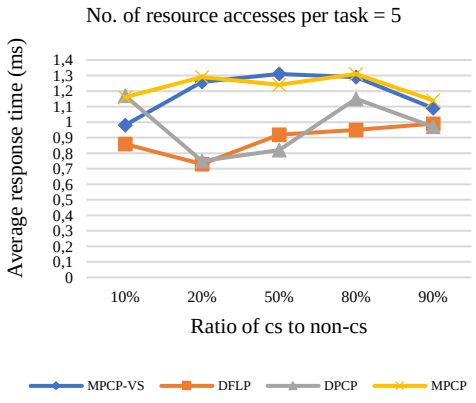
Utilization = 5%, Number of processors = 4



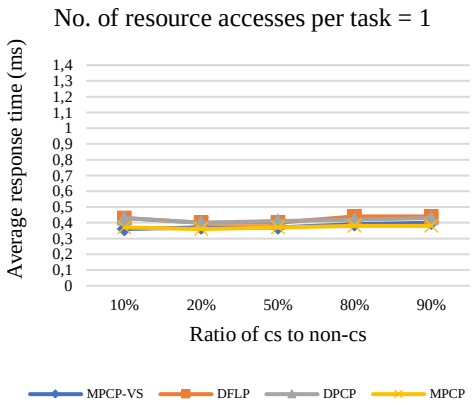


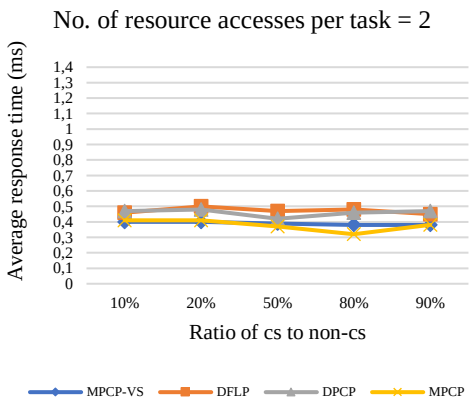


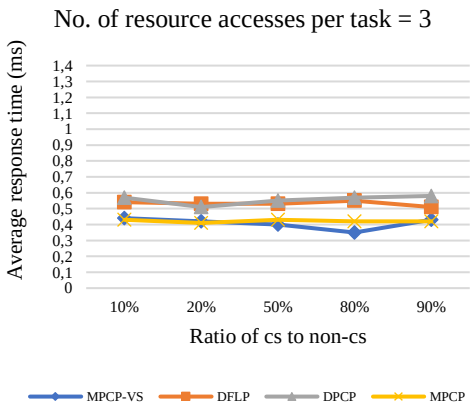


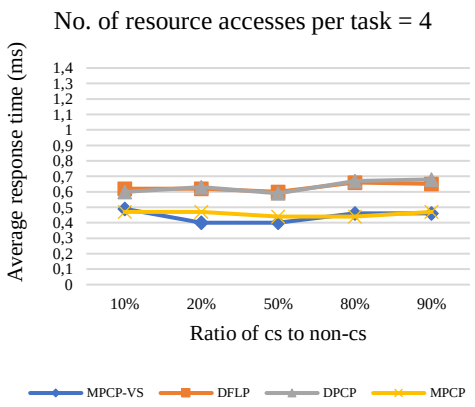


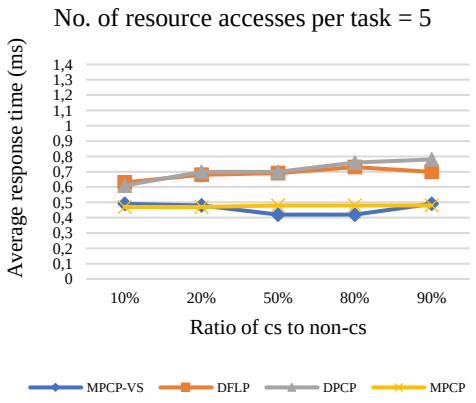
Utilization = 5%, Number of processors = 8



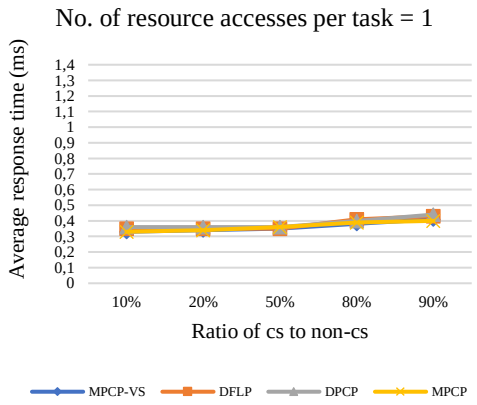


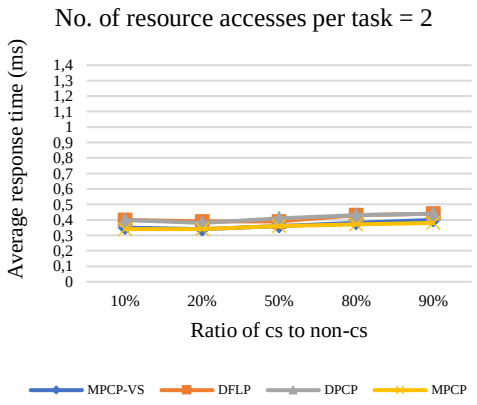


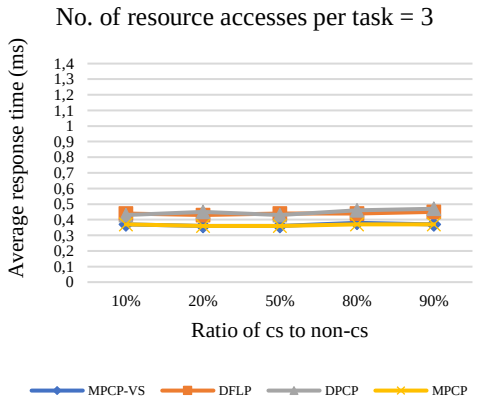


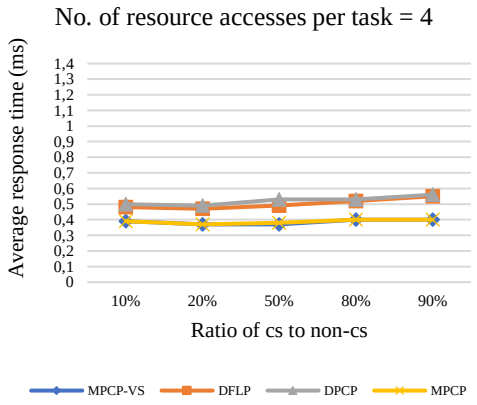


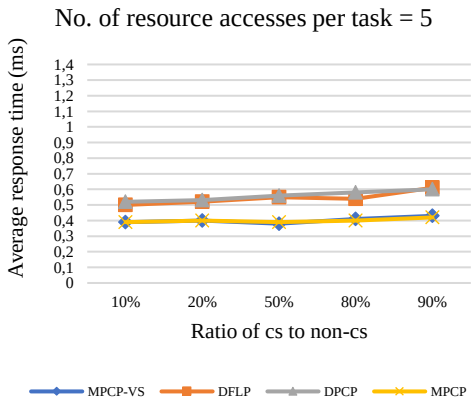
Utilization = 5%, Number of processors = 16



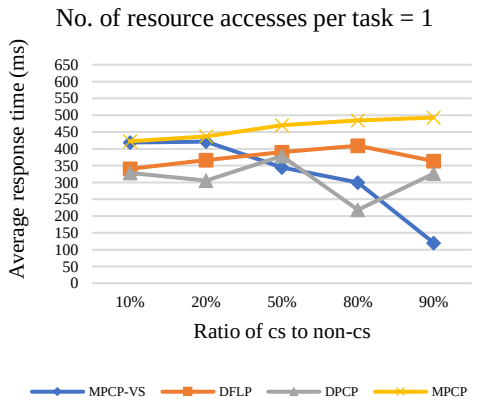


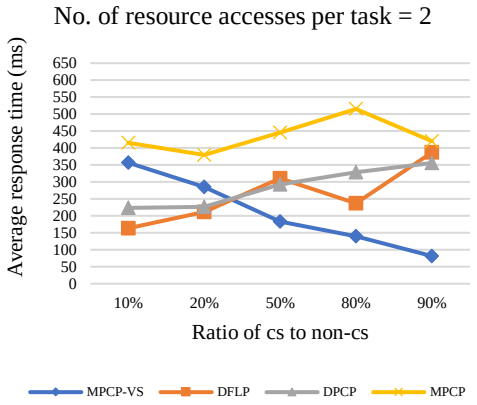


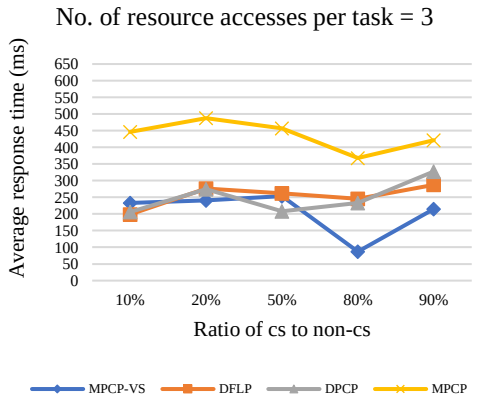


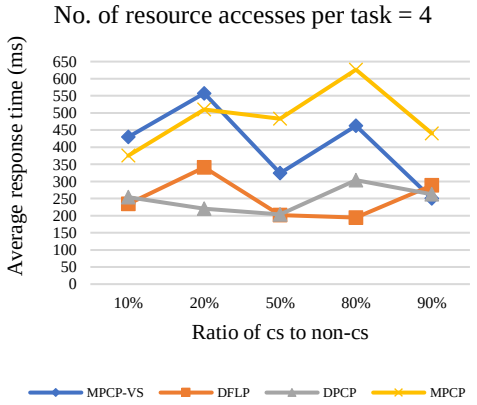


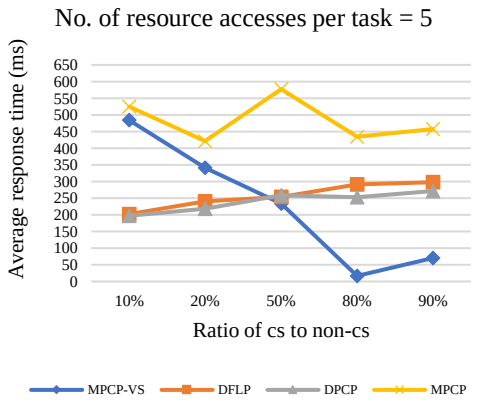
Utilization = 25%, Number of processors = 2



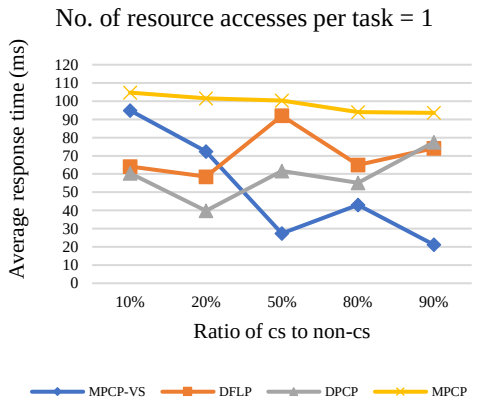


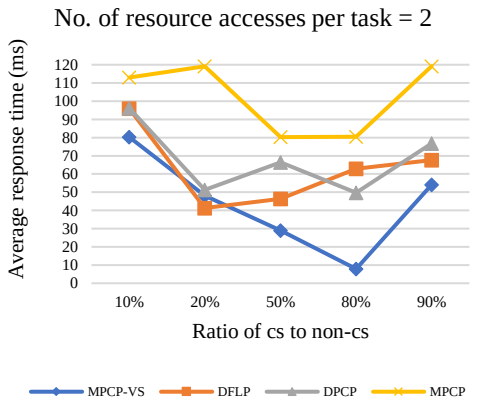


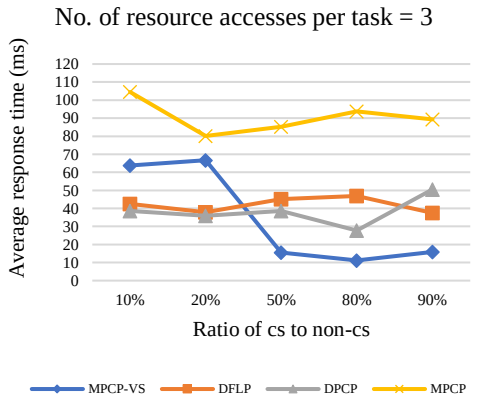


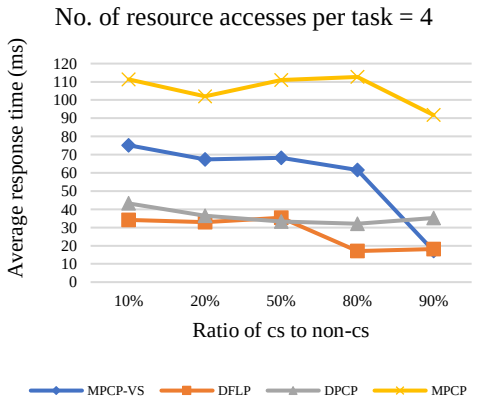


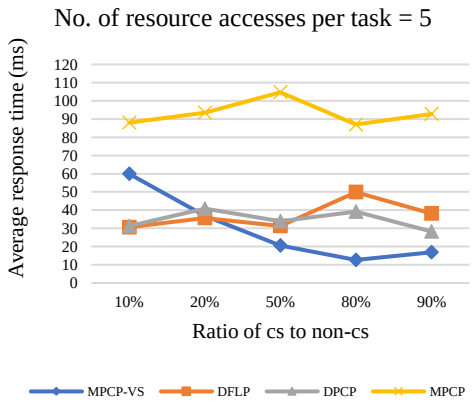
Utilization = 25%, Number of processors = 4



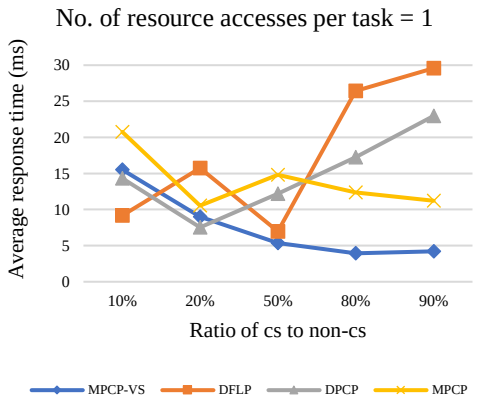


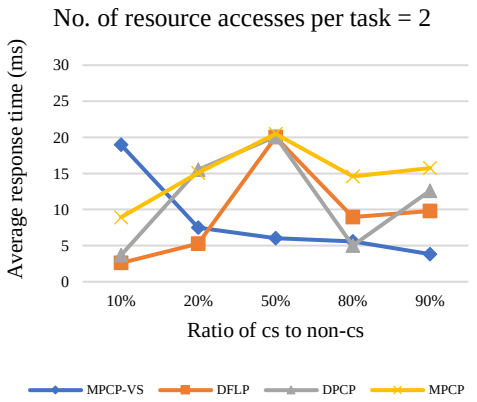


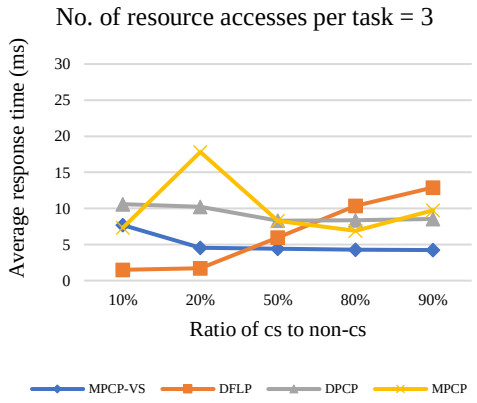


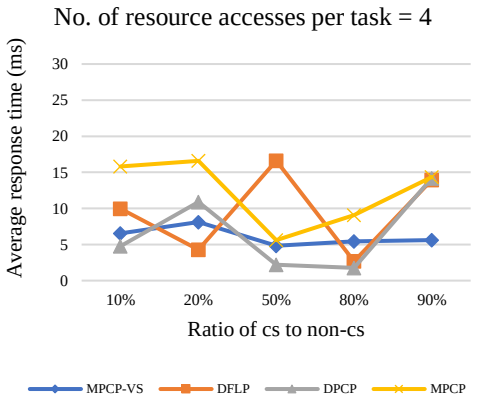


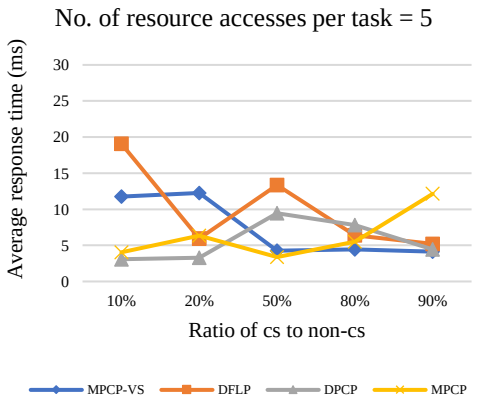
Utilization = 25%, Number of processors = 8



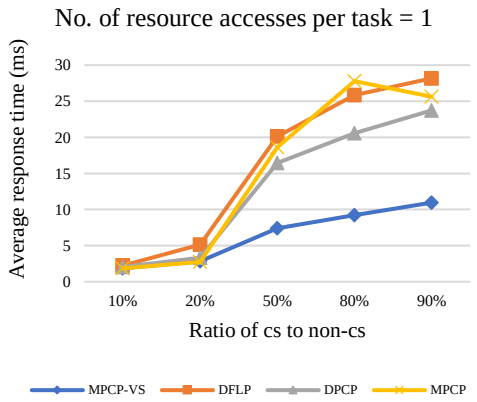


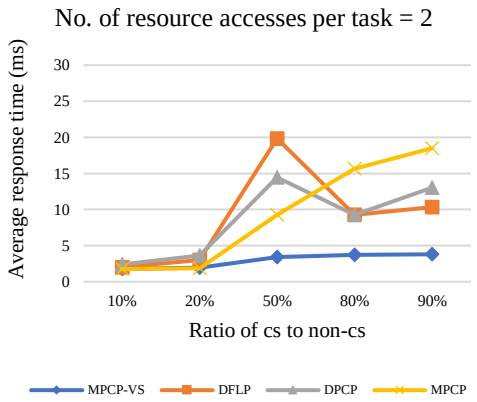


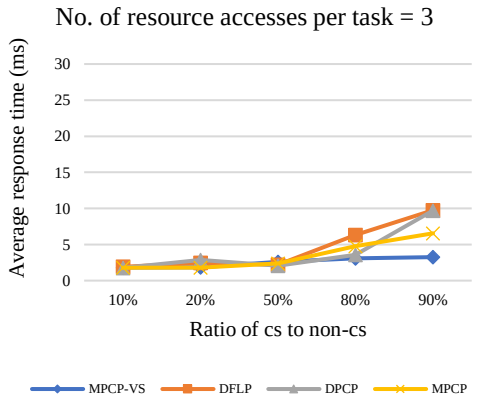


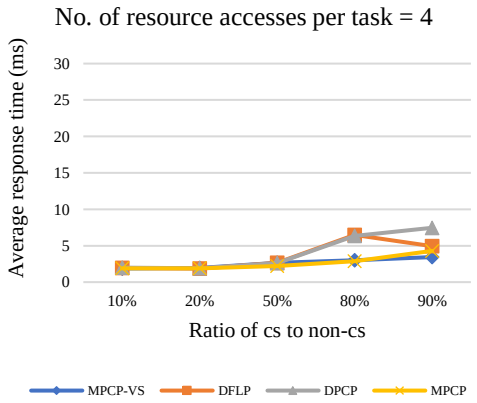


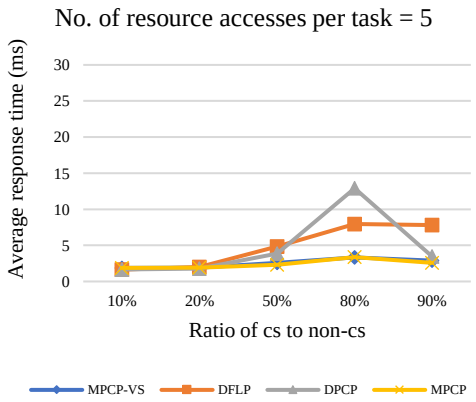
Utilization = 25%, Number of processors = 16



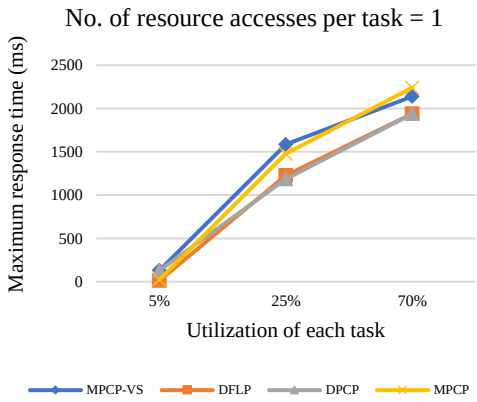


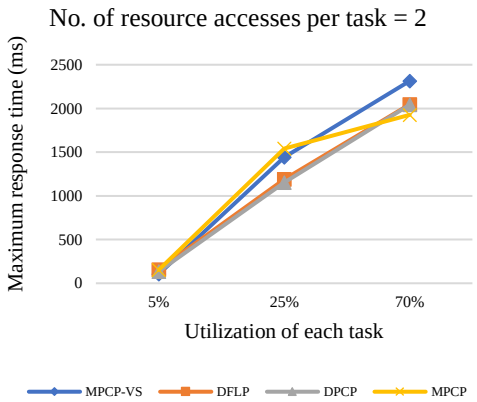


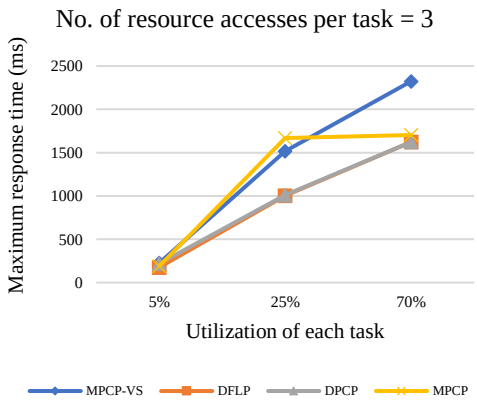


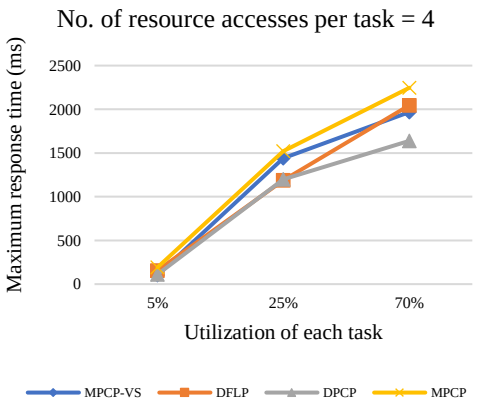


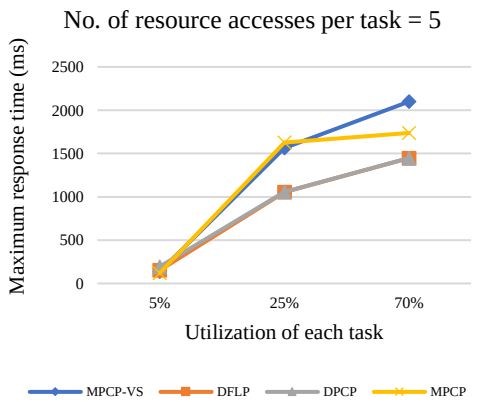
Ratio of the total cs length to the total non-cs length = 10%, Number of processors = 2



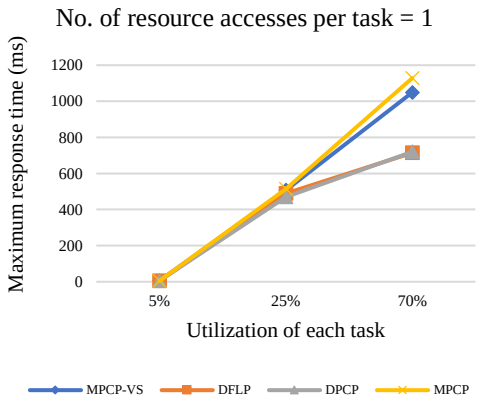


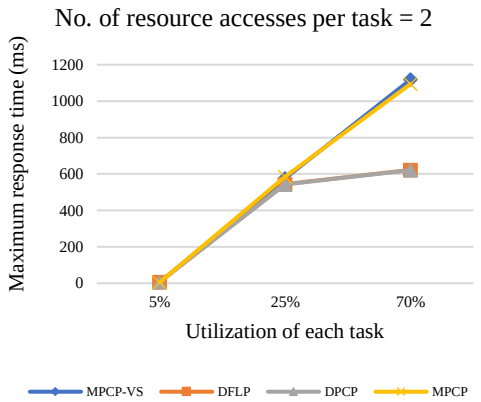


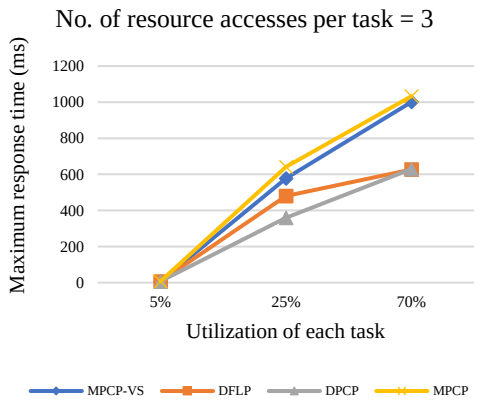


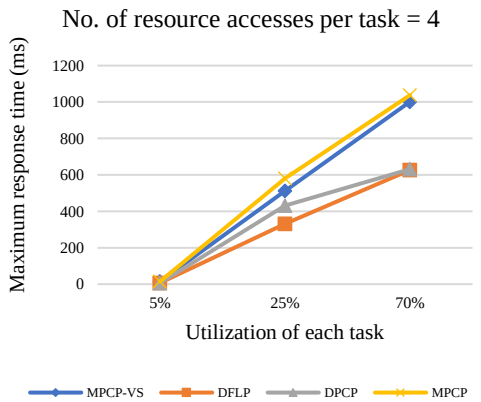


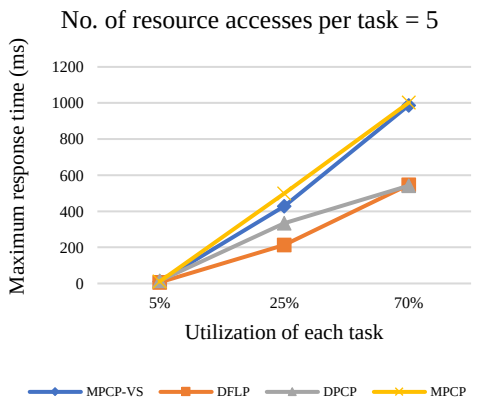
Ratio of the total cs length to the total non-cs length = 10%, Number of processors = 4



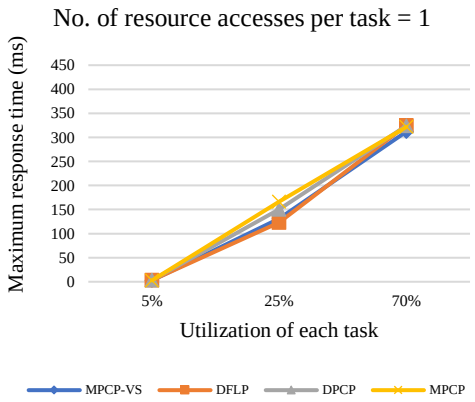


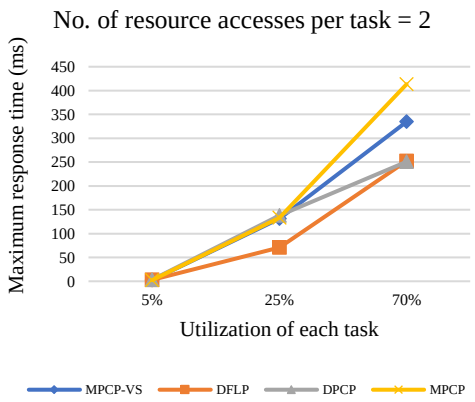


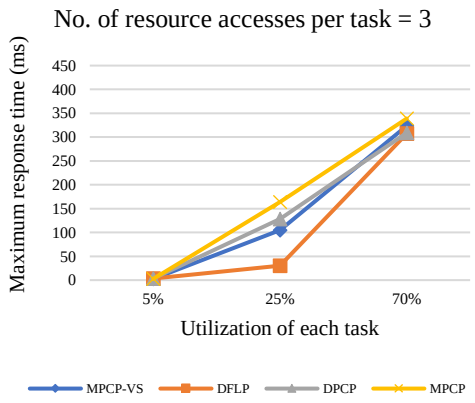


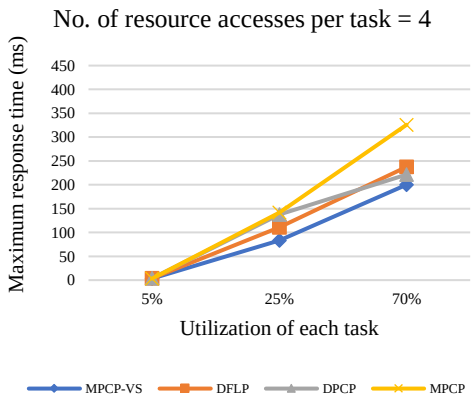


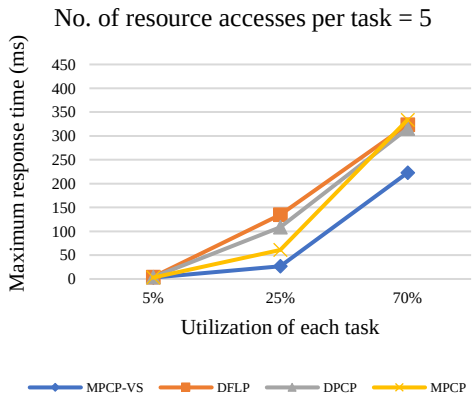
Ratio of the total cs length to the total non-cs length = 10%, Number of processors = 8



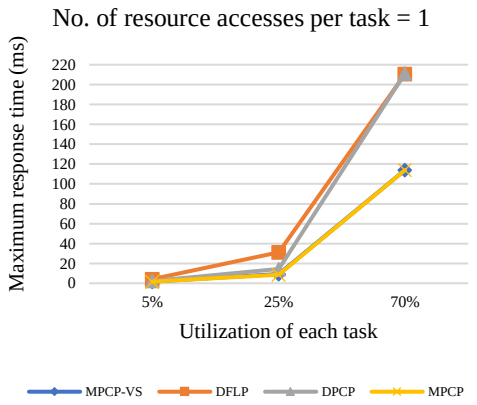


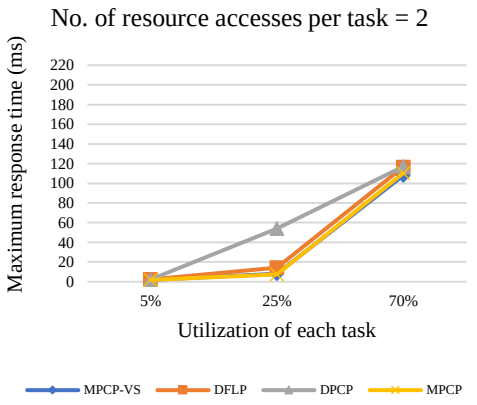


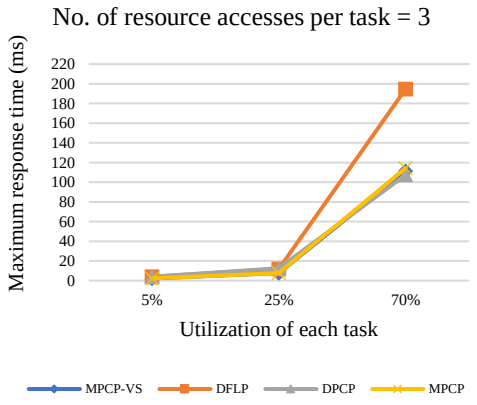


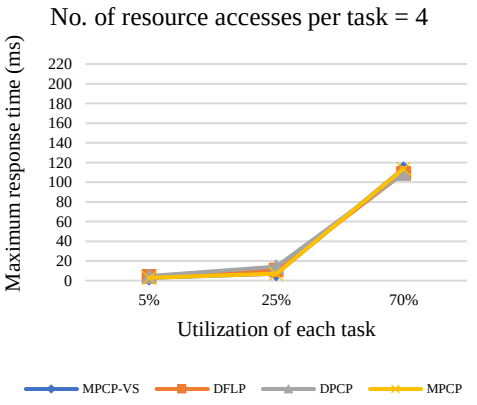


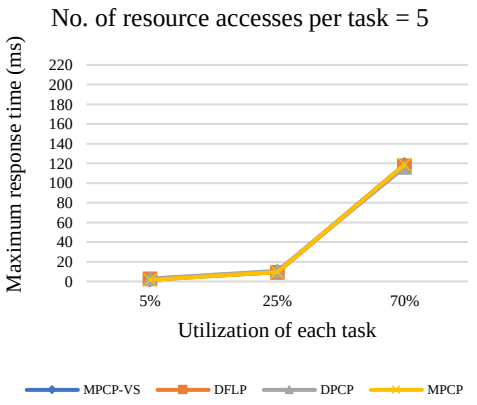
Ratio of the total cs length to the total non-cs length = 10%, Number of processors = 16



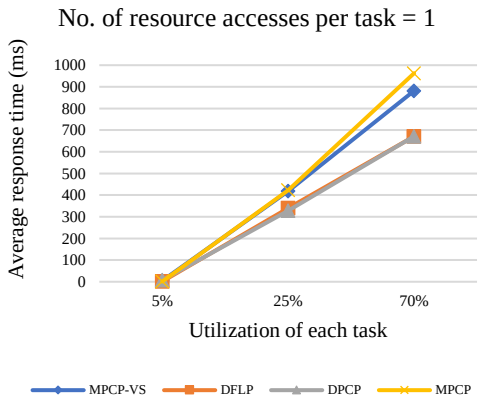


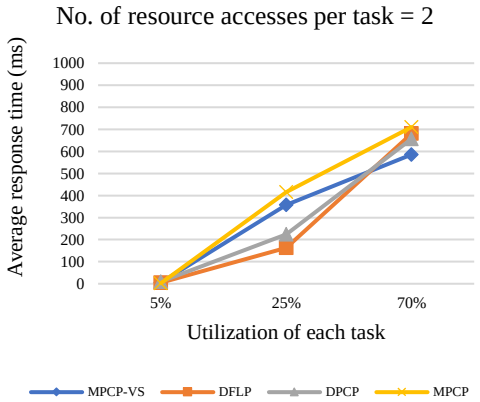


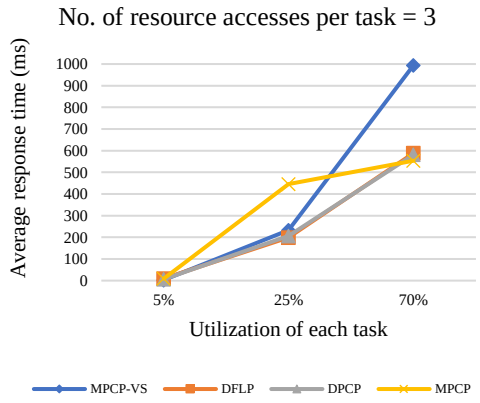


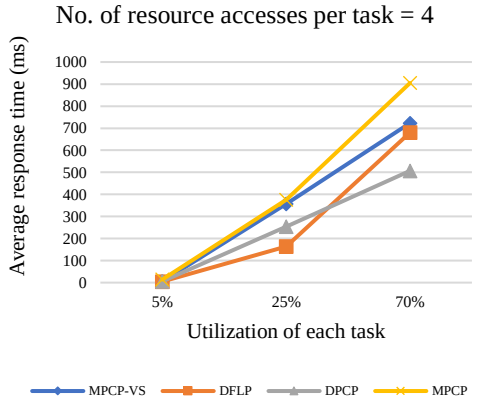


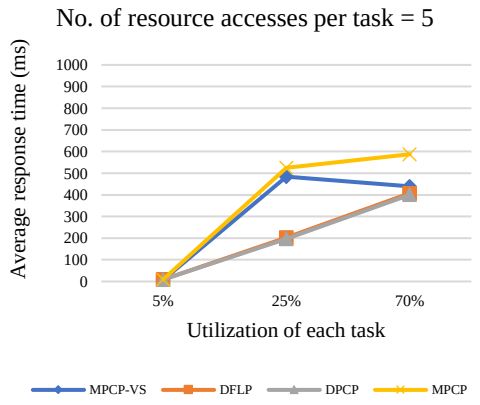
Ratio of the total cs length to the total non-cs length = 10%, Number of processors = 2



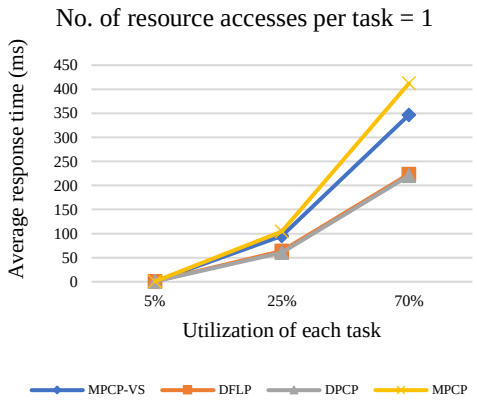


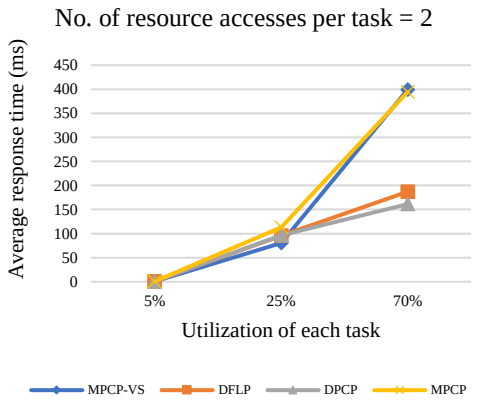


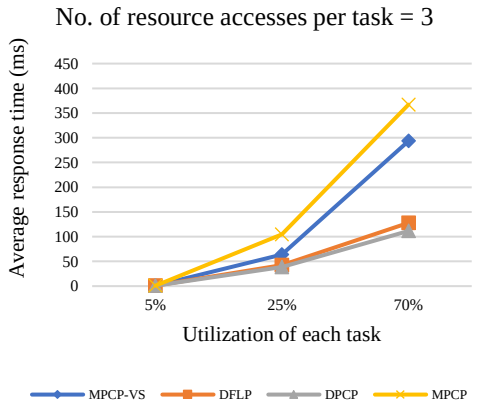


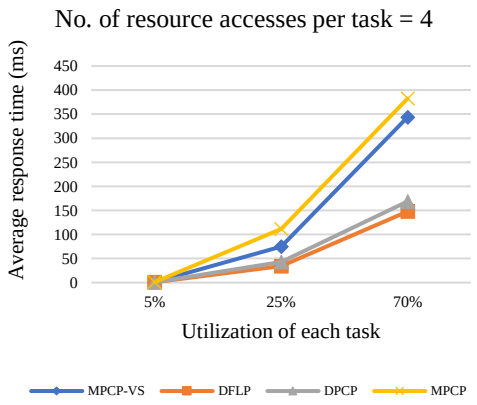


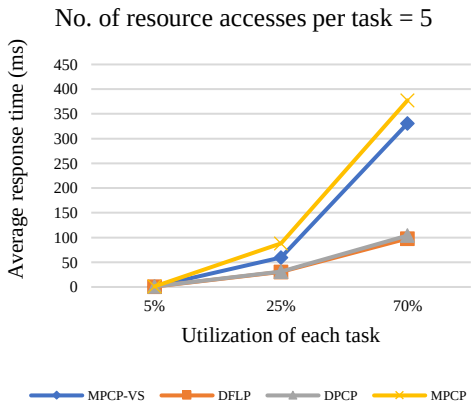
Ratio of the total cs length to the total non-cs length = 10%, Number of processors = 4



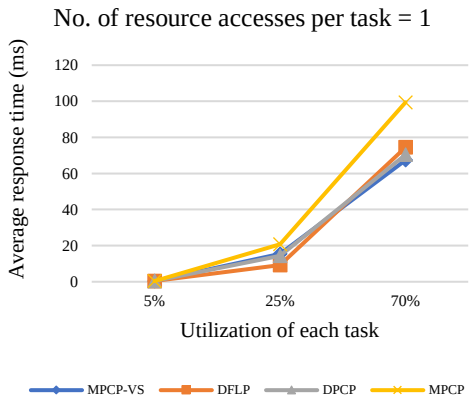


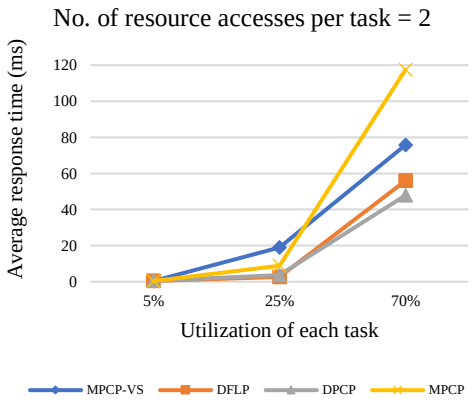


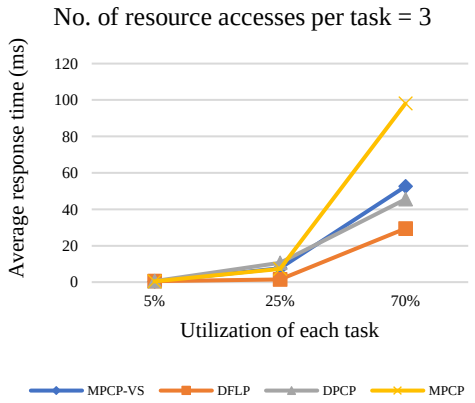


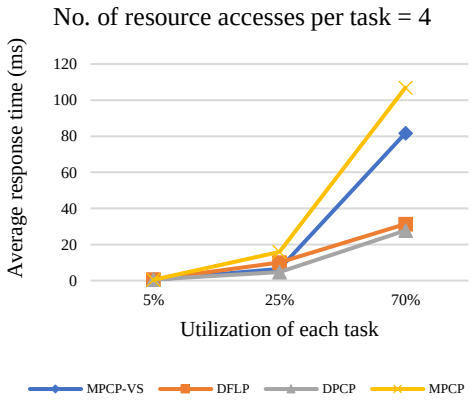


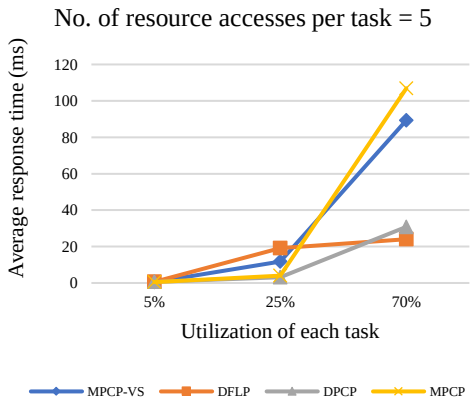
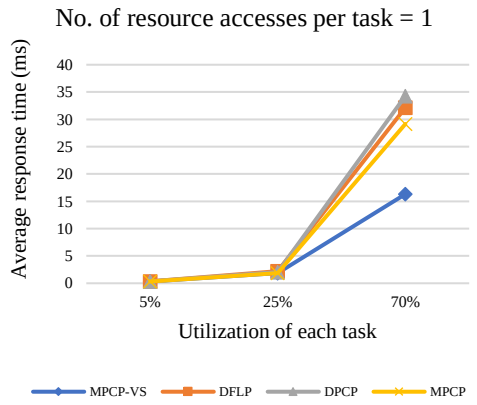
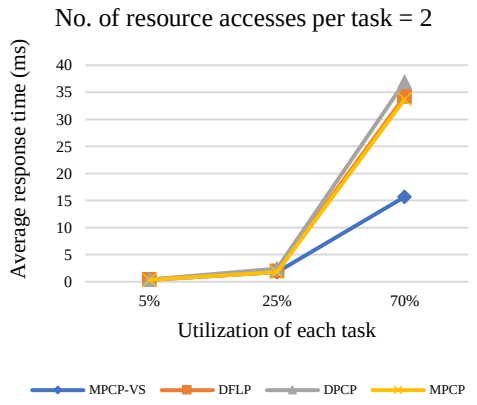
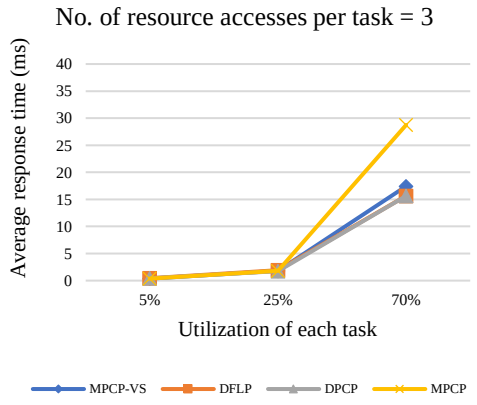
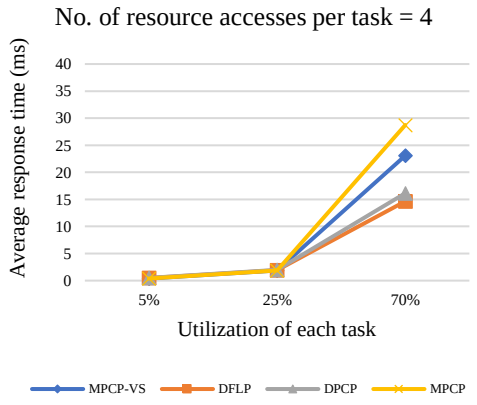
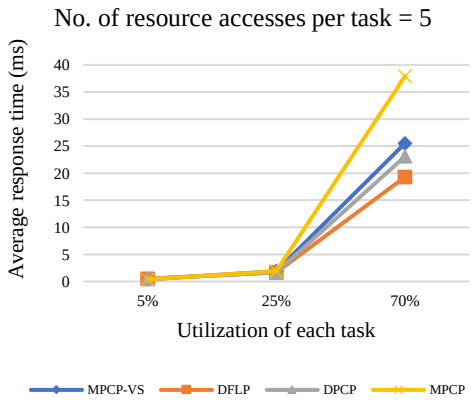
Ratio of the total cs length to the total non-cs length = 10%, Number of processors = 8











Ratio of the total cs length to the total non-cs length = 10%, Number of processors = 16