Chapter 1
Introduction

1.1 Introduction to Real-Time Operating System

In our daily life, embedded systems have become an essential part of our life. It can be a mobile phone, a smartcard, a music player, a router or any other electronics devices which are rapidly changing our lives. An embedded system is defined as a group of several devices or parts such as computer software, hardware, and extra mechanical components intended to accomplish a specific function on the fly on time basis. Due to time constraint, they are also called real-time systems. For that, the real-time operating system (RTOS) becomes the foundation of the Embedded Systems. These real-time systems are lacking in memory and processing power. For this reason, memory management is essential and has been a topic of research since the inception of real-time systems.

To define a real-time operating system, **it is a type of an operating system which provides support to the real-time applications by giving an accurate result within a specific time duration. This time duration is called deadline. [11]**

Table 1.1: Difference between General Purpose OS and RTOS

	RTOS	**General Purpose OS**
Determinism	Deterministic	Non-deterministic
Preemptive kernel	All kernel operations are preempt-able	Not Necessary
Priority Inversion	Have mechanisms to prevent priority inversion	No such mechanism is present
Task Scheduling	Scheduling is time-based	Scheduling is process based
Application	Typically used for embedded applications	Generally used for desktop PC or other general purpose PCs

The table 1.1 shows the fundamental dissimilarities between the general purpose operating system and a real-time operating system [86]. Differentiation is prepared based on various characteristics like determinism, kernel, priority inversion, job scheduling and latency. Real-time Operating Systems have implemented in such a way that the scheduling of events is extremely deterministic. A real-time system may have one or multiple real-time tasks. The fundamental dissimilarity between a real-time and non-real-time task is that a real-time task is always considered by the time limit which informs the system that the given task must be finished within the specified time.

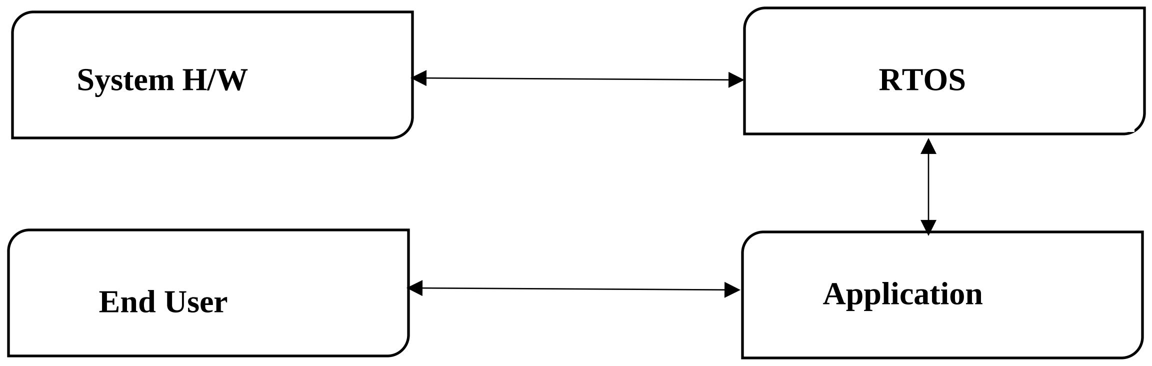

Figure 1.1: Overview of embedded system for real-time applications [21]

The real-time system can be categorized into three different categories on the basis of its criticality of meeting the deadline [44]:

1) Hard real-time systems: A real-time task/system is considered to be hard, if not generating the outcomes within the given deadline may create terrible significances on the system under control. For example, automotive systems, nuclear-plant controllers etc.
2) Firm real-time systems: A real-time task/system is considered to be firm, if generating the outcomes after its deadline is of no use for the system, but it may not create any destruction or catastrophes. For example, railway ticket reservation system.
3) Soft real-time system: A real-time task/system is considered to be soft if the outcome of the system is still useful even after missing the deadline with slight performance degradation. For example, multimedia applications on the mobile phone.

1.2 Features of RTOS

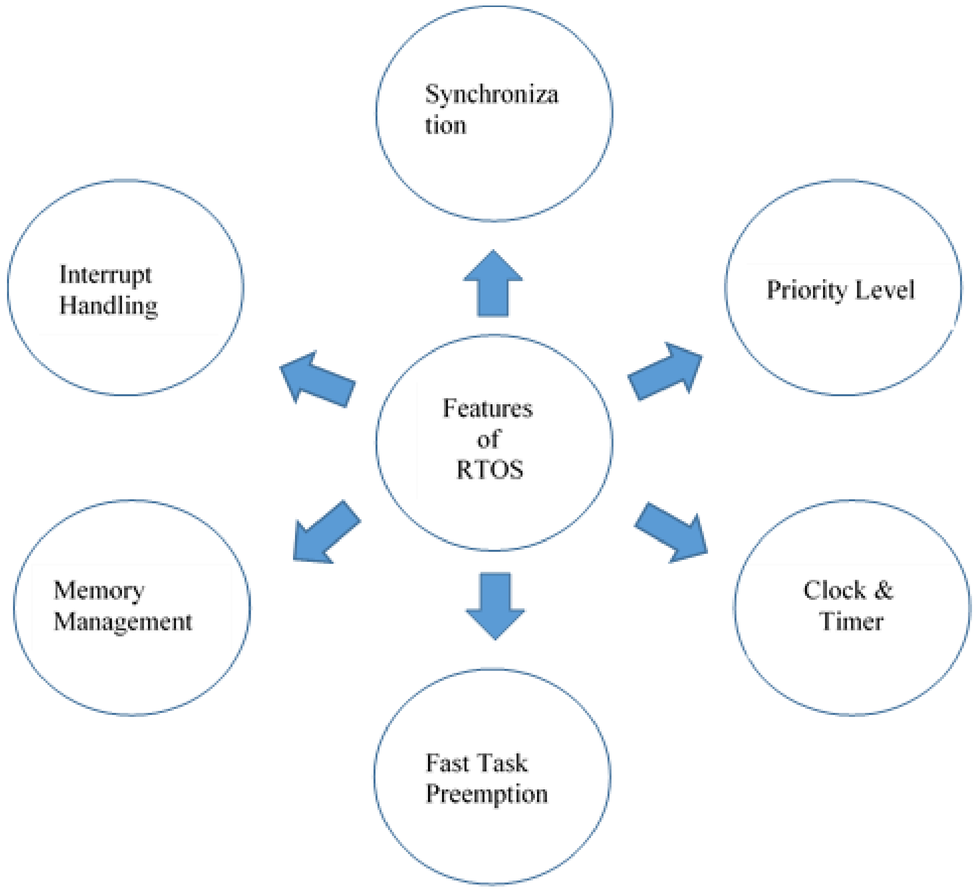

Figure 1.2: Features of RTOS [21] [44]

- **Synchronization:**

  Synchronization is essential for tasks of the real-time system to share mutually exclusive resources. Here, predictable inter-task communication and synchronization strategies are mandatory for multithreading communication in an appropriate manner [27] [61].

- **Interrupt Handling:**

  For handling interrupt, an Interrupt Service Routine is required. Here, interrupt latency is considered as the time duration between an interrupt generation and the execution of the related Interrupt Service Routine [36].

- **Timer and clock:**
  The Clock and timer services with suitable determination are important aspects of each real-time operating system.

- **Real-Time Priority Levels:**
  Each real-time operating system must have the provision of real-time priority levels which are allocated by the developers and the operating system cannot modify it [33].

- **Fast Task Preemption:**
  For the effective procedure of a real-time application, whenever a high priority critical task comes to the critical region, and if a running task has low priority, then the high priority task should be allocated to the CPU immediately.

- **Memory Management:**
  Real-time operating systems for huge and standard sized applications are predictable to offer virtual memory [25]. They not only have to achieve the demands of memory but to also have to serve the memory requests of non-real-time applications as well like different types of editors, browsers, etc. A Real-time Operating System usually has a small memory size with only essential features for an application [42] [43].

1.3 Memory Management

There are two types of memory management carried out in real-time operating system using a stack and using a heap. At the time of context switching of tasks, the stack management is used whereas for dynamically allocating memory to the tasks, the heap management is used [77] [97].

The memory management of the real-time operating system is categorized as static memory management and dynamic memory management [92]. Figure 1.3 shows the classification of Memory Management Technique.

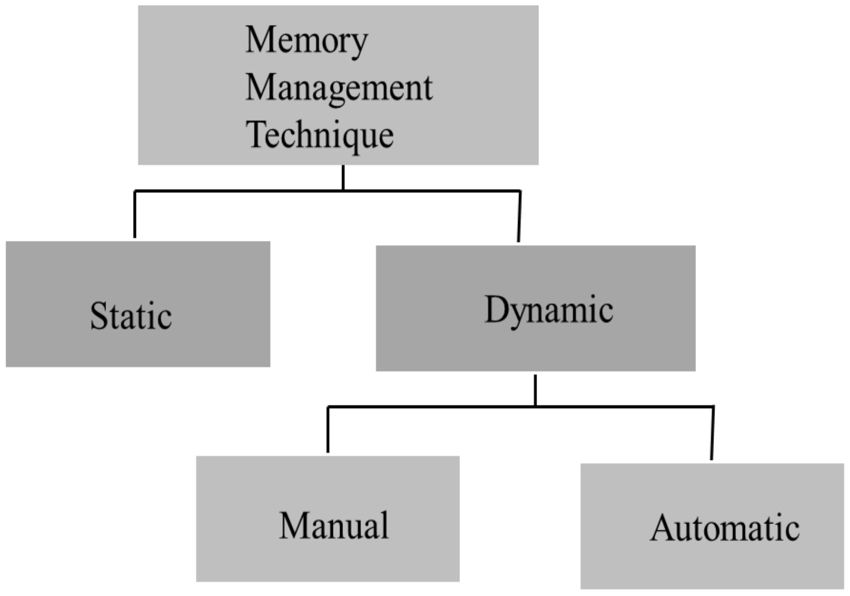

Figure 1.3: Classification of Memory Management Technique [92]

The following table Table1.2 [21] [82] shows the fundamental differences between static and dynamic memory management.

Table 1.2: The Fundamental difference between static and dynamic memory management

Static Memory management		Dynamic Memory Management
1	Memory allocation is done at compile or design time.	Memory allocation is done at runtime or during execution.
2	Static memory allocation is a fix process which means requisite memory for a specific process is already identified, and after allocating memory, no modifications can be done during the execution.	Dynamic memory allocation needs memory manager to maintain which portion of the memory is allocated and which one is unallocated. So, when a process requests memory, it can allocate and deallocate memory whenever needed.
3	Allocation and deallocation of memory are not performed during execution.	Memory bindings are established and demolished during execution.
4	Extra memory space required.	Less memory space required.

Further, Dynamic memory management is classified into two categories, Manual and automatic [21] [92].

1. Manual memory management

In manual memory management, the developer has direct control over the memory allocation and deallocation. Typically, this is carried out either by heap management functions or by the language constructs that affect the stack. The advantage of manual memory management is it is simple for developers to comprehend and use. However, the disadvantage is that memory management errors are very ordinary.

2. Automatic memory management

In Automatic memory management, the task of memory manager is to manage memory by reusing the memory blocks which are out of scope from the program objects. These automatic memory allocators are also called garbage collectors. The benefit of using it is it removes the majority of the memory management errors. The demerit is many times it misses the deallocation of unusable memory blocks which may lead to memory leakage problem. It also consumes a large amount of the processor time and generally has non-deterministic performance.

1.4 Problem Statement

To design and develop a dynamic memory allocator for real-time operating systems which can reduce the memory fragmentation and satisfy the maximum number of memory block request in the consistent execution time.

1.5 Motivation

The majority of the operating systems use dynamic memory allocation for processing which requires communicating explicitly with memory allocator component. Though memory allocation algorithms have been analyzed and worked upon broadly since 1960, it has been observed that there is less attention given to the multiprocessor architecture and real-time operating system as well. Most of the algorithms have been designed such that, they are applicable for the general-purpose operating system and do not fulfill the necessities of the real-time systems [37].

Moreover, the limited allocators have been designed to support real-time systems but they are not completely scalable for the multiprocessors. As the 21st century is an era of high-performance computing, the demand for multi-core architecture has gained momentum. NUMA architecture based systems are the outcome of this tendency and offer an organized scalable design [50] [62]. However, these dynamic memory allocators are not capable of performing on a multiprocessor architecture and do not comply with real-time system requirements as well. Researchers have proved that the existing memory allocators for any operating systems which support NUMA architecture are not suitable for the real-time applications [71] [81]. Hence, there is a need to have a dynamic memory allocator which can perform well on SMP and NUMA based soft real-time systems, with better execution time and less fragmentation. This research is carried out in the same direction to achieve the aforementioned goal.

Memory is an important aspect when implementing real-time applications because of its costly management in terms of time and memory. Therefore, worst-case execution time and utilization of memory are the prime aspects of real-time system designers. To increase the performance of the real-time system, a cost-effective memory manager is required [79] [84]. As a result, the majority of the real-time systems use static memory allocator for random allocation and de-allocation of memory blocks. It means the allocation of memory is done at the compilation time or during initialization of program before the application arrives into core real-time stage where the entire physical memory is accessible as a single block and can be used as per requirement. Due to the unavailability of automatic memory management, an issue regarding excessive use of memory space arises as the memory which is used for keeping objects cannot be simply reclaimed. The developers have to employ and maintain their private pools of memory to decrease the excessive usage of memory space and reclaim the memory space. The real-time systems have been growing in size and complexity with the multi-core and multiprocessor architecture based systems and hence they require the flexible use of the existing resources comprising memory as well [107]. The static memory and pool of memory are not applicable in the era of the multiprocessor system and the implementation of real-time applications will gradually have to use dynamic memory management allocator to achieve the predictable performance and flexibility [95].

Dynamic memory allocator has been the most significant and essential part in the general-purpose software field because it is more proficient and flexible than the static memory allocator.

Since last five decades, the research scope of the dynamic memory allocation algorithms has been analyzed for its significance and attractiveness. Earlier, a considerable amount of research work has been carried out on the issue of the good average response time of Dynamic Memory Allocation algorithms to check their speed and efficiency in allocation and deallocation of memory blocks. Along with the ways to decrease fragmentation in memory, in case of dynamic memory allocators, there are numerous faster and competent algorithms available for the general purpose operating system, for instance, DLmalloc [53], tcmalloc [99] and Hoard [6]. However, the worst-case execution time (WCET) of these algorithms has not been analyzed thoroughly. For real-time systems, no significant analysis has been carried out for dynamic memory allocation algorithms. Due to this, the majority of the application designers of real-time systems usually ignore the dynamic memory allocators. The reason why the designers are worried is that the worst-case execution time of dynamic memory allocator routines is not constrained [88] [98]. As the lifespan of a real-time application is typically longer than the lifespan of a general-purpose application, not only WCET of the dynamic storage algorithm but also memory utilization efficiency should be considered for the algorithms [106]. Throughout the long lifespan of the application, dynamic memory allocator creates holes in memory, which cannot be reclaimed because of their small size. These holes result in slow offensive response time or they lead to failure in meeting the deadlines. Such holes are called memory fragmentation.

1.6 Objectives of Memory Management Algorithm

The research in dynamic memory management for real-time systems is one of the areas which is still unconquered because real-time applications impose different requirements on memory allocators than the general-purpose applications. The most significant requirement in real-time systems is the investigation of scheduling which is used to decide if the response time of real-time application can be bound to fulfill the timing restriction of the execution. This investigation should consider the impression of multiprocessor architecture settings like concurrency, lock contention, cache misses and traffic on the bus. The effects of all these problems on NUMA architecture systems, related to dynamic memory management could be described as follows:

1) **Reduce memory fragmentation**: As stated above that the period of real-time applications is normally longer than that of simple applications. It may be running from one day or one month while the period of simple application is very short. During the lifespan of a real-time application, the systems may randomly release memory segments of any size, which may cause holes (free memory block) in the memory [7]. In addition to this, these holes are used to be of very small size and hence they cannot be reused. For this, reducing fragmentation of memory can be considered as the main objective.

2) **Restricted execution time**: Applications of real-time systems should meet their timing criteria. For this, the dynamic memory management algorithm should be designed in such a way so that the application can be completed within the given deadline.

3) **Increase node-based locality**: In NUMA based architectures; the memory is shared in the system so that its processors can access remote memory. Here, the time required to access memory will be high as compared to access the local memory. For this reason, it would cause performance degradation of the system. Hence, the node-based locality which tries to access the local memory is the main features to improve the system performance [35] [47].

4) **Reduce false sharing**: Due to real-time systems are cache coherent in nature [72] [78], it introduces "False sharing". When several processors try to access different data objects inside similar cache or memory block, this problem may arise [8]. This false sharing may lead to performance degradation of the application when working on multiprocessor shared memory architecture.

5) **Reduce memory access to the remote nodes**: The time consumption for accessing a remote node is very high [100] which is almost twice than for the simple one, the application developer need to consider this point and develop it in such a way that it can reduce the memory access of remote node.

6) **Reduce lock conflicts**: Applications which are executing on the NUMA based systems and which run concurrently cause the problem of lock conflicts. This issue should be addressed to maintain the performance of the system.

1.7 Research Contributions

For last four-five decades, the majority of the operating systems have started using dynamic memory allocation algorithm. In this, every time the memory allocation needs explicit communication with an allocator. The memory allocation algorithms have been analyzed broadly since 1960, but inadequate devotion has been given to the real-time properties. Majority of the algorithms are for the general-purpose operating system and do not fulfill the requirements of real-time systems. Moreover, the limited allocators which provide support to real-time systems are not properly scalable for multiprocessors. The demand for high-performance computational processing has been increased gradually which lead to development of multi-cores. NUMA architecture based systems are a portion of this tendency and offer an organized scalable design. In summary, there are a few dynamic memory allocators are available, but they do not perform properly on multiprocessor architecture and do not satisfy real-time system requirement as well. The existing memory allocators for any operating systems which support NUMA architecture are not suitable for the real-time applications. So there is a requirement of a dynamic storage allocator which can perform well on both SMP as well as NUMA based soft real-time systems. This allocator must give better execution time and less fragmentation. Through this research, the following research contributions have been made.

1. A Dynamic memory allocator **DmRT** has been designed and implemented for Symmetric Multiprocessor System which provides consistent and optimum execution time along with less memory fragmentation and satisfies maximum number of memory requests as compared to the other existing allocators.
2. A dynamic memory allocator **DmRT** for NUMA (Non-Uniform Memory Access) architecture based real-time operating system has also been designed and implemented. It provides consistent and optimum execution time, less memory fragmentation and it also serves a maximum number of the memory requests.

3. There are so many simulators available to simulate different test cases for scheduling in a real-time operating system, e.g. Litmus-RT, Mark3, rtsim, etc., but no simulator is available for simulating memory management algorithm for RTOS so far. Therefore, MemSimRT has been designed to simulate various memory allocators for both SMP as well as NUMA architecture based RTOS. This will be useful for all existing allocators as well as the allocators which would be proposed in future. The MemSimRT can be downloaded using the following QRcode

1.8 Applications

The main requirement of any real-time operating system is to finish the task within the given deadline. If it misses the deadline then either it would lead to catastrophic events or may lead to failure in performance. To accomplish this criterion, it is mandatory that the task should have resources in time whenever it requires. Here, resources may be anything CPU, Input/Output or memory etc.

To execute a process it must have CPU in time as and when needed, but at the same time, it should have an on-demand memory as well. To provide memory on time, the strong and perfect memory allocator will help the task to achieve the deadline. Hence, a memory allocator is designed which can run on Symmetric Multiprocessor (SMP) architecture as well as Non-uniform Memory access (NUMA) architecture based real-time operating system.

1.9 Organization of Thesis

The remaining chapters of the thesis are organized into six chapters as given below:

Chapter 2. Literature Review: This chapter explores the dynamic storage algorithm models provided by general-purpose and real-time systems, including their policies and mechanisms. Furthermore, the conventional memory management algorithms have also been investigated, with specific emphasis on managing the memory blocks. This chapter also highlights hybrid memory management algorithms which perform better on multiprocessor architecture systems as compared to the conventional dynamic storage allocators.

Chapter 3. DmRT for SMP: This chapter focuses on providing and implementing a more efficient dynamic memory management algorithm for SMP architecture. Also it includes the results of this allocator and compares it with the existing allocators.

Chapter 4. DmRT for NUMA: This chapter discusses an efficient and dynamic memory management algorithm for NUMA architecture especially with its results analysis in context of existing allocators.

Chapter 5. Memory Management Simulator: MemSimRT: This chapter introduces a simulator of memory management algorithm which has been designed for the real-time operating system.

Chapter 6. Conclusions and Future Work: The conclusions derived based on the results of this research work are given in this chapter. Additionally, some directions for further possible research are also presented.