Memory Management in Real-Time Operating System

A Synopsis

Submitted in partial fulfillment of the

requirements for the award of the degree

of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE & ENGINEERING

By

SHAH VATSALKUMAR HASMUKHBHAI

FOTE/878

Guided By,

Dr. APURVA SHAH

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

FACULTY OF TECHNOLOGY & ENGINEERING

THE MAHARAJA SAYAJIRAO UNIVERSITY OF BARODA

VADODARA-390002 (INDIA)

JUNE 2018

ABSTRACT

The memory allocation algorithms have been analyzed and worked upon broadly, but there is less attention given to the multiprocessor architecture and real-time operating system as well. Most of the algorithms are applicable for the general-purpose operating system and do not fulfill the necessities of real-time systems. Moreover, limited allocators designed to support real-time systems which are not completely scalable for multiprocessors. In the 21st century, as we have entered into 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. However, existing dynamic memory allocators are not capable of performing on a multiprocessor architecture and do not comply with real-time system requirements as well. Researches have proved that the existing memory allocators for any operating systems which support NUMA architecture are not suitable for real-time applications. 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 of a dynamic memory allocator for real-time systems. 1. Dynamic memory allocator ***DmRT*** *for symmetric multiprocessing (SMP) and Non-Uniform Memory Access (NUMA) architecture based real-time operating system has been designed and implemented which provides consistent and optimum execution time, less memory fragmentation, as well as satisfying a maximum number of the memory request, compared to other existing allocators. 2. There are so many simulators available to simulate different test cases for scheduling in a real-time operating system like Litmus-RT, Mark3, rtsim, etc., but till date, none of the simulators is available for simulating memory management algorithm for RTOS. Hence,* ***MemSimRT*** *has been designed to simulate various memory allocators for both SMP as well as NUMA architecture based RTOS.*

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1
Introduction

1.1 Introduction to Real-Time Operating System

RTOS denotes “Real-time Operating System” which is basically a type of an operating system which provides support to the real-time applications by giving an accurate result within the time limit [4][17]. Real-time Operating System can be mainly classified into two categories: 1) hard real-time system and 2) soft real-time system depends on how rigorously it follows the task accomplishment deadline.

Table 1.1: Difference between General Purpose OS and RTOS [4]

	RTOS	**General Purpose OS**
Determinism	Deterministic	Non-deterministic
Preemptive kernel	All kernel operations are preemptable	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
Latency	Have their worst-case latency defined	Latency is not of a concern Purpose OS
Application	Typically used for embedded applications	General purpose OS is used for desktop PCs or other general purpose PCs

A real-time system can be categorized into three different categories on the basis of its criticality [17]:

- **Hard:** A real-time task/system is considered to be hard if generating the outcomes after its deadline may create terrible significances on the system under control. For example, automotive systems, and nuclear-plant governing systems, etc.

- **Firm:** 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 does not create any destruction. For example, railway ticket reservation system.
- **Soft:** A real-time task/system is considered to be soft if generating the outcomes after its deadline still provides usefulness for the system, however affecting a performance degradation. For example, multimedia applications on the mobile phone.

1.2 Features of RTOS [4] [17]

There are various features of RTOS like Synchronization, Interrupt Handling, Timer and clock, Real-Time Priority Levels, Fast Task, Preemption and Memory Management among them our focus is towards Memory Management.

Real-time operating system for huge and standard sized application are predictable to offer virtual memory, not only to achieve the demands of memory but to provide the memory request of non-real-time applications as well such as different types of editors, browsers, etc. A real-time operating system normally has small memory size by comprising only the essential features for an application [12].

1.3 Memory Management

Generally, memory management of Real-time operating system can be categorized as static memory management and dynamic memory management [35]. Table 1.2 [4] shows the fundamental difference between static memory management 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 execution.	Dynamic memory allocation needs memory manager to maintain which portion of the memory is allocated and which portion of the memory is unallocated. Due to this when a process requests memory, it can allocate memory and when the task is done then deallocate it.
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.

1.4 Problem Statement

Since last four to five decades, the majority of the operating systems have been used 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 real-time systems [37]. Moreover, limited allocators designed to support real-time systems which are not completely scalable for multiprocessors. In the 21st century, as we have entered into 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 indicating that a few dynamic memory allocators are available. 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. Researches have proved that the existing memory allocators for any operating systems which support NUMA architecture are not suitable for real-time applications. 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 of a dynamic memory allocator for real-time systems.

1.5 Objectives of Memory Management Algorithm

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

1) Reduce memory fragmentation
2) Restricted execution time
3) Increase node-based locality
4) Reduce false sharing
5) Reduce memory access to the remote nodes
6) Reduce lock conflicts

1.6 Research Contributions

1. Dynamic memory allocator **DmRT** has been designed and implemented for symmetric multiprocessing system which provides consistent and optimum execution time, less memory fragmentation, as well as satisfying a maximum number of the memory request, compare to other existing allocators.
2. As per the need of high-performance computing, a dynamic memory allocator **DmRT** for NUMA architecture based real-time operating system has been designed and implemented

which provides consistent and optimum execution time, less memory fragmentation as well as satisfying a maximum number of the memory request.

3. There are so many simulators available to simulate different test cases for scheduling in a real-time operating system like Litmus-RT, Mark3, rtsim, etc. but till date, no such simulator is available for simulating memory management algorithm for RTOS. Hence MemSimRT has been designed to simulate various memory allocators for both SMP as well as NUMA architecture based RTOS.

   Download MemSimRT using following QR code

Chapter 2
Literature Review

2.1 Dynamic Memory Management Algorithms

Memory management is the key feature of the real-time operating system. This section describes certain memory management algorithms for general-purpose as well as the real-time operating system. There are conventional as well as unconventional algorithms for dynamically allocation/deallocation of memory. All traditional algorithms can be considered as conventional algorithms. 1) Sequential Fit Algorithm 2) Buddy Allocators 3) Doug Lea(DLmalloc) 4) Half-Fit 5) TLSF 6) tcmalloc 7) Hoard 8) Smart Memory Allocators

1) Sequential Fit Algorithm

It can be categorized into four types [4] [17] [37]: Best Fit, Next Fit, First Fit and Worst-Fit.

a. Best Fit: Its name itself suggest that each time the allocator tries to search out the smallest unallocated memory block which is big enough to fulfill the application’s request.
b. Next Fit: The array of unallocated blocks is to be found from the location where the previous search suspended, returning the next memory block which is big enough to fulfill the request.
c. First fit: The array of unallocated blocks is to be found from scratch, returning the first memory block which is big enough to fulfill the request.
d. Worst fit: The array of unallocated blocks is searched, returning biggest existing unallocated memory block.

The time complexity of this algorithm is O(n).

2) Buddy Allocator Algorithm

This algorithm [31] uses an array of link lists of the unallocated blocks. Each list for allowable block size. For example list of 2N block size like 200 kb, 400 kb, 800 kb so on. The buddy allocator finds the smallest block which is large enough to hold the request from the list of unallocated blocks. If the unallocated block list is empty, then it will search a block from another list which is larger than a request, then select and split the block [6][10]. A block must be divided into the same size blocks, i.e., 400 kb block will split into two 200 kb blocks. In the same way, the block may be merged with its adjacent block of the same size, and this is possible if the adjacent block has not been divided into the smaller block.

3) Dlmalloc

This algorithm was proposed by Doug Lea in 1996. Later on, its extended version has been designed by Gloger in 2006 [20] and by Free Software Foundation in 2012. This algorithm occupies a huge number of static size arrays known as small-bins to allocate a small memory block. Bins occupy unallocated blocks which are having sizes not more than 256 bytes. Every bin comprises unallocated blocks of equal size. If demanded memory block's size is not more than 256 bytes, the algorithm tries to find for existing blocks in the bins using best-fit policy or large enough to satisfy the request.

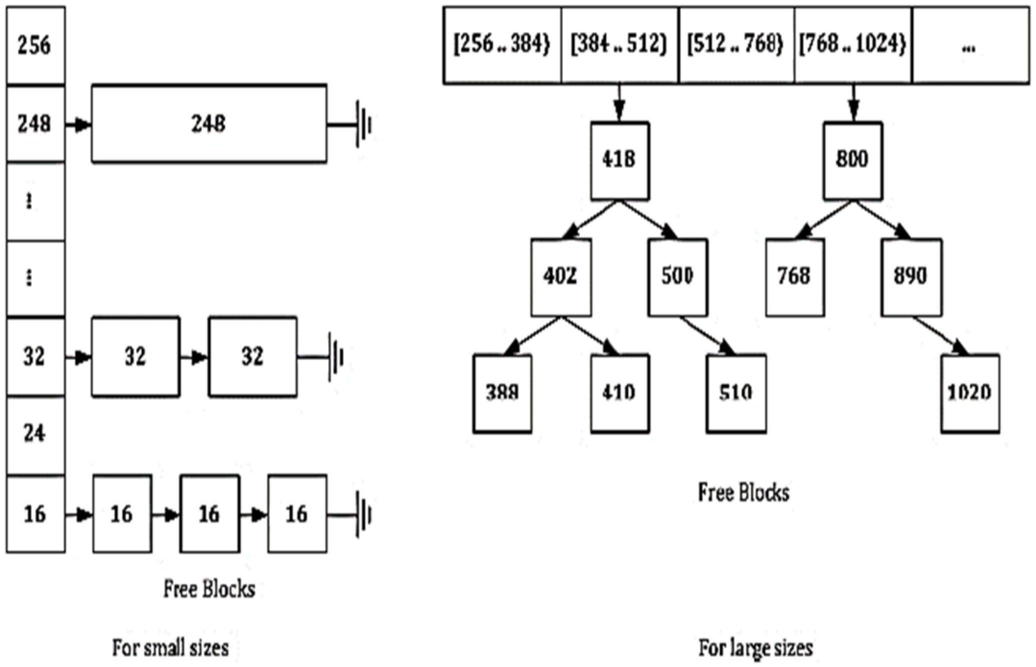


Figure 2.1: Organization of DLmalloc algorithm [37]

If the demanded memory block's size is larger than 256 bytes and smaller than some fixed value (normally 256 Kb), then algorithm tries to search existing blocks in an array known as tree-bin, which is having a tree structure for the memory block. As shown in Figure 2.1 tree-bins accumulate a collection of the bin. Nodes in the tree structure act as a small-bin, comprising the blocks of equal size. Any demand of block size beyond the fixed value, the algorithm transfers the requests to the operating system through some specific system call.

4) Half-Fit

This algorithm has been designed and implemented by Ogasawra [29] [30] in 1995. This algorithm uses bitmapped fit strategy and achieving execution time in constant manner. As it is using bitmap policy for allocating release block, it is slow. Use of bitmap is nothing but only to maintain the status of unoccupied lists. Its time complexity of time is O(1).

This algorithm maintains a segregated list of a single level. In this list, unallocated blocks of different size are connected. It takes unallocated blocks of the required size from unallocated block list through which request will be satisfied. Figure 2.2 shows this example.

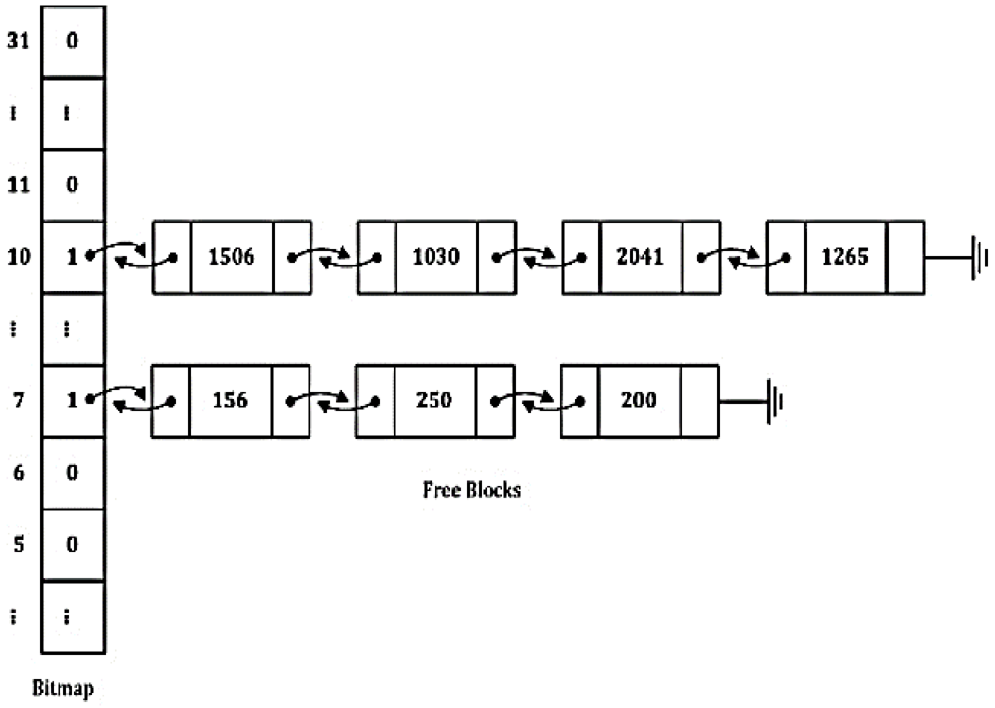


Figure 2.2: Organization of Half-fit algorithm [29] [37]

It has specific allocation/deallocation methodology to avoid searching using bitmaps because of its constant execution time. If the requested size of the memory block is r, then index i may be computed by this equation [30]:

$$i = \begin{cases} 0 & if\ r\ is\ 1 \\ \lfloor \log_2(r-1) \rfloor + 1 & otherwise \end{cases} \quad 2.1$$

Where i, specifies the unallocated memory block lists whose width vary from 2^i to $2^{i+1} - 1$. After computing the value of i, an unallocated block is occupied from the unallocated block list indexed by i. If there is no unallocated block in the list, the subsequent unallocated block list will be searched.

If the size of an assigned memory block is more than the demanded memory block size, an unallocated block from the unallocated block list will be split into two distinct memory blocks of sizes r1 and r2 before assigning for allocation then the remaining memory block r2 will be put into the matching unallocated block list. For deallocation, released memory block will be directly merged with neighboring memory block if the corresponding block is free/unallocated.

This algorithm is suitable for real-time operating systems because of its constant time complexity.

5) TLSF

TLSF is one of the best available dynamic memory allocation algorithm stands for two-level segregated fit algorithm, unlike segregated list allocator, this algorithm having two level of segregated lists of unallocated memory blocks in which each list maintain the unallocated blocks of predefined size range [25][26][28].

The first-level of list (FLI) splits unallocated memory blocks into various parts which are apart by the power of two like 2, 4, 8, 16 onwards. The secondary level known as second-level lists splits each first level list by a user-defined variable known as Second Level Index. TLSF structures are shown in Figure 2.3.

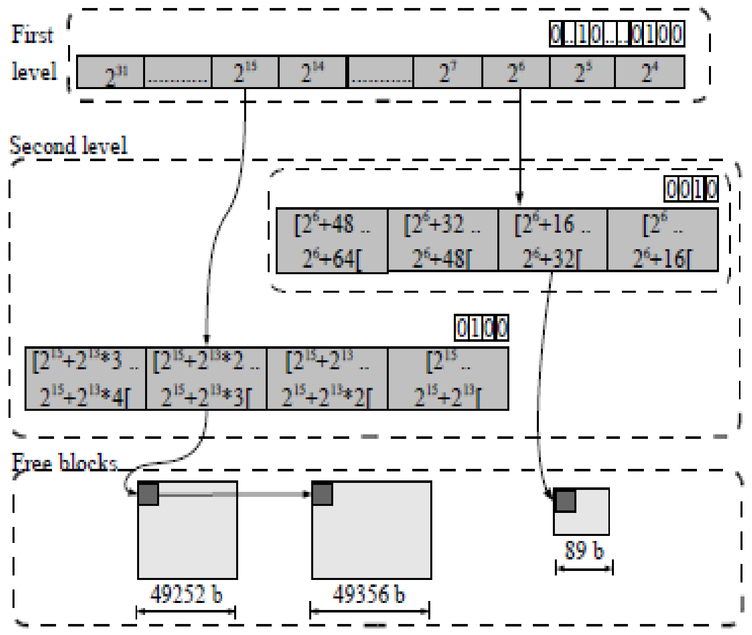


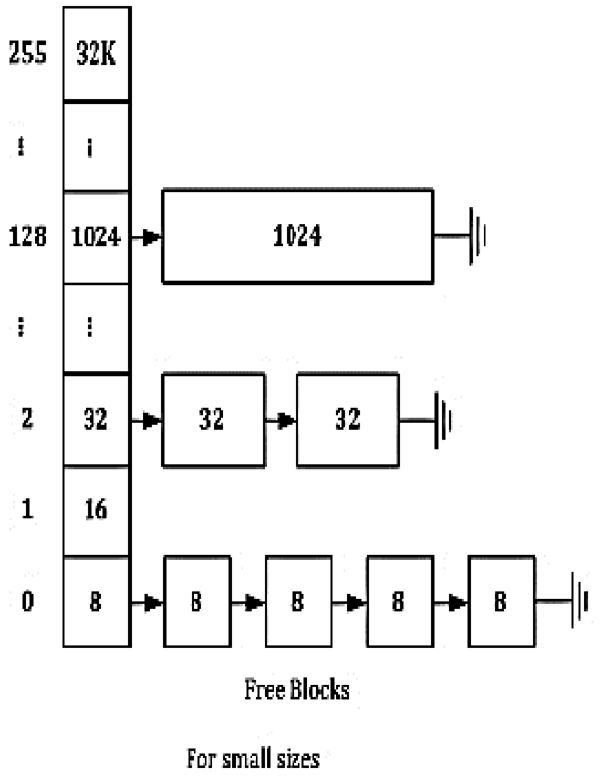
Figure 2.3: Simple TLSF Structure [25][26]

This algorithm provides bounded execution/response time. This algorithm is best suitable for the real-time operating system.

6) tcmalloc

This algorithm is developed and implemented by Sanjay Ghemawt in 2010 [36]. It is an extremely accessible algorithm which associates both global heap structure and threads private heap multiprocessor architecture. To allocate small memory block which size range from 4 bytes to 32 Kb, this allocator allocates private local heap to each thread. Hence, small size memory block allocation does not require synchronization mechanism for the thread.

To allocate large memory blocks which ranges from 32 Kb to 1 Mb, this allocator maintains a global heap structure which is collectively used by all available threads. As this global heap is shared by threads, some kind of synchronization mechanism should be used to offer mutual exclusion. Hence, it employs spin-lock mechanism. If any applications demand a huge memory block whose size is beyond 1Mb, then allocator forwards the request to the existing operating system using a system call or APIs. Figure 2.4 shows the structure of tcmalloc allocator. The time complexity of this allocator is O(1).



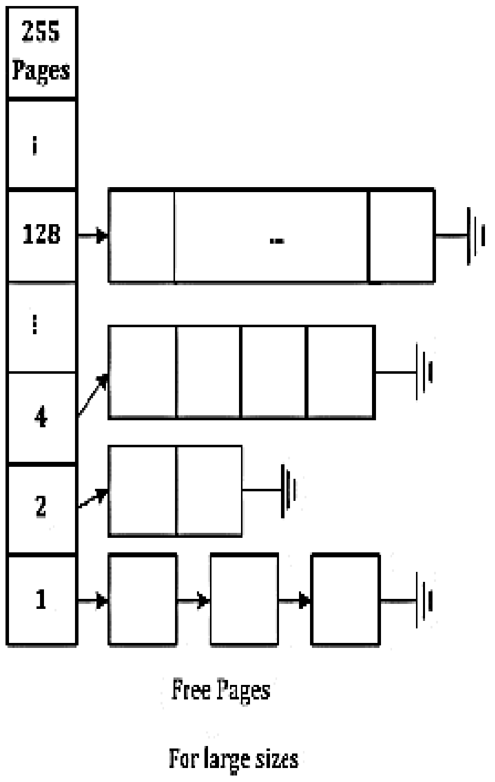


Figure 2.4: Organization of the tcmalloc algorithm [36]

7) Hoard

This algorithm was designed by Bergar in 2000 [17], and it is popular due to its speed and scalable in the environment of the multiprocessor system. This allocator also employs a segregated class mechanism, but unlike tcmalloc it maintains private heap to each processor and to avoid heap conflict it maintains a global heap. Other than private processor heap and global heap, it also maintains private heap per thread to allocate smaller size memory blocks which size less than 256 bytes. Threads which are executing on the same processor can also share private processor heap.

Hence, to allocate any blocks whose size is less than 256 bytes, it first searches it into a heap of the thread if it fails, then it searches into private processor heap, and if it is also full, then it searches into a global heap. So its time complexity is O(n), where n is the number of chunks of the memory block which is known as super-block., Figure 2.5 shows the structure of this algorithm.

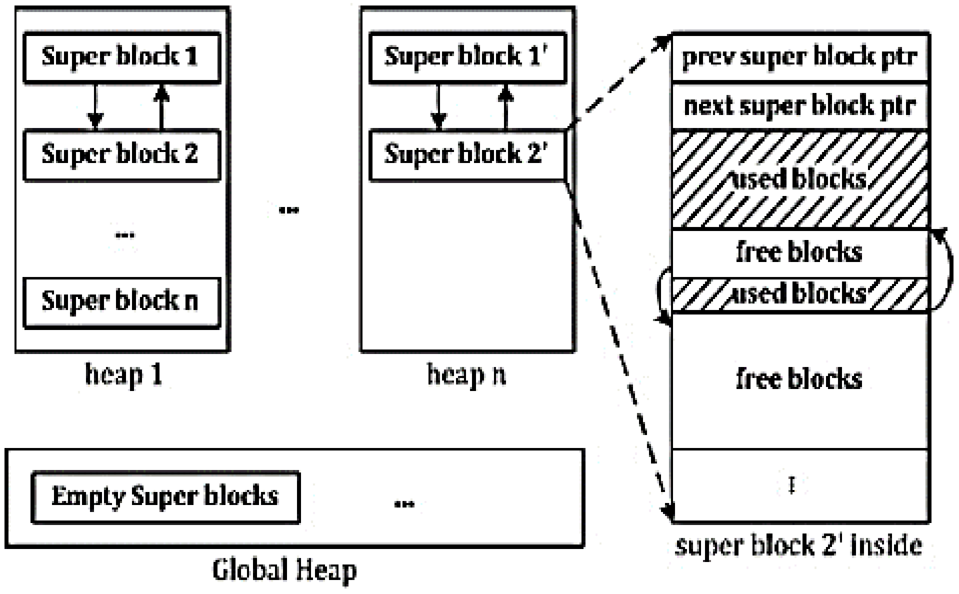


Figure 2.5: Organization of Hoard algorithm [2]

8) Smart Memory Allocator Algorithm

This algorithm has been proposed by Ramakrishna M, Jisung Kim, Woohyong Lee and Youngki Chung in 2008 [47]. This is a custom type of dynamic memory algorithm having the best response time and less memory fragmentation. This algorithm divides memory blocks into two categories. One is short-lived, and another is long-lived memory blocks. The short-lived memory blocks are allocated in the direction of lowest level to highest level from heap while long-lived memory blocks are allocated from highest level to lowest level [47]. The used space of heap grows from highest level to lowest level as well as lowest level to highest level. Initially, entire heap memory is unallocated, and there is only one unallocated memory block for each short as well as long-lived memory. The heap space is divided equally into two blocks. When the heap grows from both sides, the virtual border between these two can easily modify according to the dynamic memory request.

As this algorithm predicting memory object life scope, it can easily allocate memory block from either short-lived or long-lived memory pool [9]. So it can have best response time with lower fragmentation. It is implemented with a lookup table and hierarchical bitmaps which are improved version of the multilevel segregated mechanism.

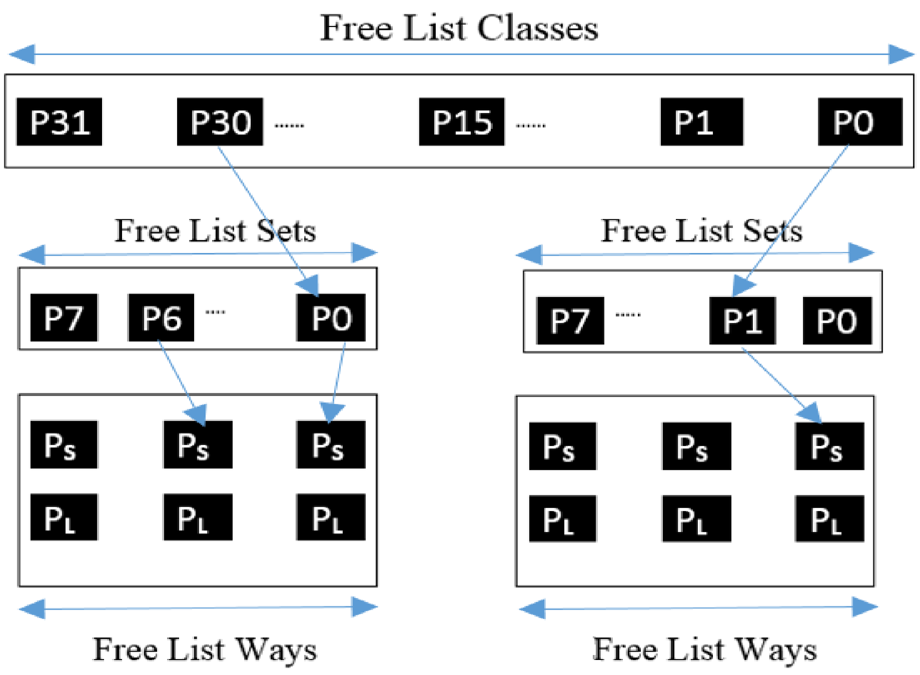


Figure 2.6: Structure of Smart Memory Allocator [41] [47]

2.2 Summary

Table 2.1: Summary of all Allocators

Memory Management Algorithms	Parameters		
	Allocation	Fragmentation	NUMA Support
Sequential fit	O(n)	Acceptable	No
Buddy System	$O(\log_n 2)$	Unacceptable	No
Doug Lea(DLmalloc)	O(m)	Acceptable	No
tcmalloc	O(1)	Acceptable	Yes
Hoard	O(n)	Acceptable	No
Half-fit	O(1)	Unacceptable	No
TLSF	O(1)	Acceptable	No
Smart Memory Allocator	$O(\log_2 n)$	Unacceptable	No

Table 2.1 shows in the worst-case, the time complexity of the allocators as well as fragmentation by the allocator is acceptable or not and whether it provides support to NUMA architecture or not. In the table, n is the heap size, m is the tree's depth. Concerning real-time systems segregated Fit, tcmalloc and Half-fit are the only algorithms which satisfactorily provides the desired objectives also provide a constant execution time.

Chapter 3
DmRT for SMP & NUMA

In this chapter, a new dynamic memory allocator for the real-time operating system will be discussed. It has been proposed, designed and implemented for Symmetric multiprocessing (SMP) NUMA (Non-uniform memory access) architecture. And its name is **DmRT** stand for Dynamic memory manager for Real-Time systems. This allocator has been designed to achieve constant and minimum execution time, low fragmentation and satisfying a maximum number of request for the memory block. Furthermore, the DmRT has been compared with the existing dynamic memory allocators of the real-time operating system.

All the design principals such as strategies, policies, and mechanisms will be explained then the structure of DmRT, and its results will be discussed in this section.

3.1 Design Principals

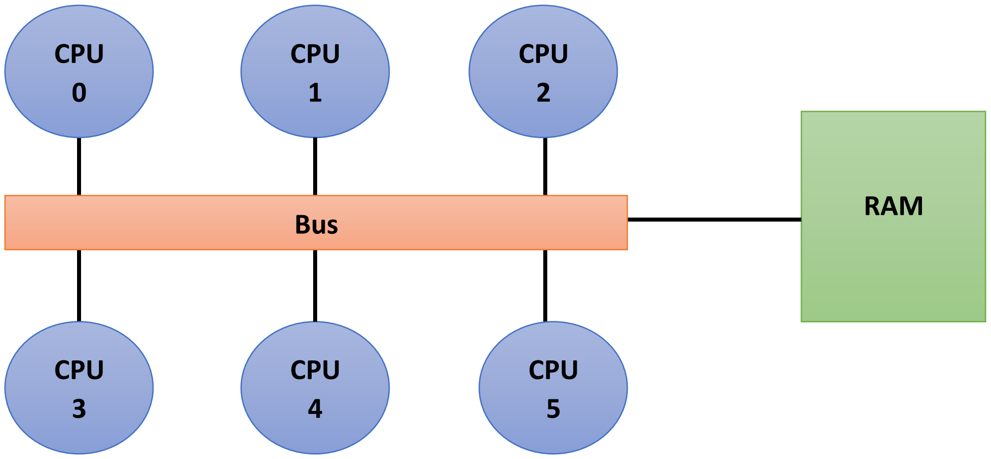


Figure 3.1: SMP Architecture

As shown in Figure 3.1, Symmetric multiprocessing (SMP) comprises a multiprocessor computer hardware and software architecture in which more than one identical processors are connected to a common or rather shared main memory, and all processors have full access to all resources like input and output devices, and are managed by a single operating system instance that treats all processors equally, reserving none for special purposes. Nowadays, the majority of the multiprocessor systems use the SMP architecture. While in the multi-core processors, the SMP architecture applies to the cores, treating them as separate processors.

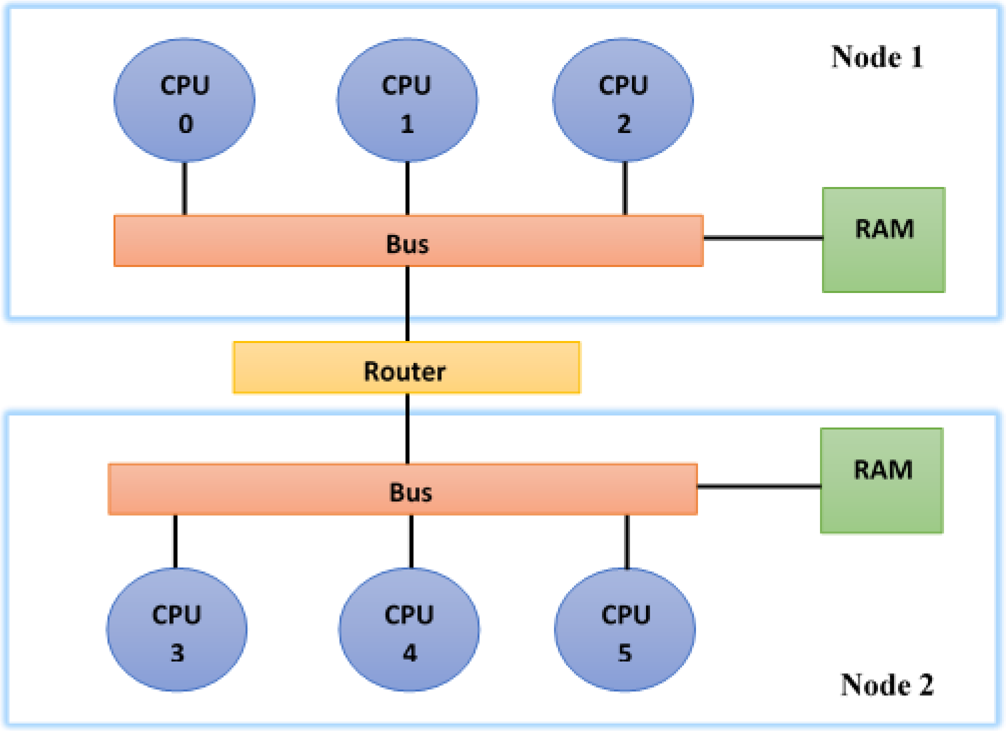


Figure 3.2: NUMA Architecture

NUMA stands for Non-uniform memory access (NUMA) is nothing but one type of design for a computer memory which is used in multiprocessing in which the access time of memory depends on the memory location relative to the processor. In NUMA architecture, a processor can read/write from its local memory faster than non-local memory, i.e., the local memory of other processor or shared memory between processors. The benefits of NUMA are limited to particular workloads, notably on servers where the data is often strongly associated with certain tasks or users.

In high-performance computing generation, NUMA is the future of SMP, but its architecture is clumsier than the Symmetric multiprocessor. Figure 3.2 shows simple NUMA architecture where only two nodes are available in which each node contains more than one processor and they all are sharing one memory. They may have their local memory as well. The complex architectures are also available which can have 4 or 8 nodes. 4 nodes architecture has been considered for this proposed memory allocator, but it is merely easy to scale it up to 8 nodes.

There are different strategies for different size of block in which has been explained in this section.

3.1.1 Multiple strategies for different sizes of blocks (for SMP & NUMA)

As discussed earlier, various strategies have been used for allocating the different size of blocks to achieve advantages of all policies, strategies, and mechanisms.

I. A small block whose size of memory block < 512 bytes
II. A normal block whose size of memory block < threshold (Some predefined size, i.e., 2Mb)
III. A large block whose size for request exceeding the threshold or (Some predefined size)

3.1.2 Search Policies and Mechanisms (for SMP & NUMA)

After defining the strategies, now its turn to which policies and mechanisms will be used to implement these strategies.

I. For small blocks, the best-fit policy is used which have been implemented by exact-fit mechanisms to reduce the fragmentation in small sizes of blocks generated by rounding up the request size of the memory block.
II. For normal blocks, the good-fit policy is used which has been implemented by segregated lists, which use an array of unallocated block lists.
III. For large blocks, the worst-fit policy is used.

3.1.3 Arrangement of blocks (for SMP & NUMA)

DmRT implements the exact-fit mechanism to increase the efficiency of small memory block allocation and to decrease internal fragmentation. It also implements the segregated-fit mechanisms to employ a good-fit and a first-fit policy for searching the nearest segregated size class. Thus it can ignore the requirement of a thorough search. Actually, two types of bitmaps have been used to keep track of unallocated blocks in the implementation of DmRT. Furthermore, this allocator is predictable as it has employed segregated list with bitmap policies and it provides confined execution time.

Among the bitmaps, use of one bitmap is to keep track of small memory blocks, and this bitmap is employed as a two-dimensional array for holding unallocated memory blocks according to the memory block size. In this proposed memory allocator (DmRT), for effective memory allocation, the block size is set apart 4 bytes from 4 bytes to 512 bytes. To check whether a specific size of a memory block is unallocated or not, two mechanisms have been employed. One is in two bitmaps total 64-bits and second is maintaining a pointer array to hold unallocated blocks as shown in Figure 3.3.

The second type of bitmap comprises a two-dimensional bitmap array directing to unallocated memory blocks. The primary bitmap which is indexed by i, specifies unallocated memory blocks whose sizes available between 2^i to $2^{i+1} - 1$, and the secondary bitmap which is indexed by j, splits each primary level range in similar width of a number of ranges. For the easiness, the number of ranges in the secondary level is signified as the power of two: $\mathbf{2}^{\boldsymbol{range}}$. For this allocator, the default value of the ***range*** is taken as 6. The variable *range* splits the primary level ranges in an equal number of ranges. For example, if value of *range* is 4 then there are 16 segregated lists inside the provided size ranges, if value of *range* is 5 then there are 32 segregated lists inside the provided size ranges and so on and if value of *range* is 1 then the allocator accomplishes unallocated blocks as powerfully as the binary buddy allocator.

The value of the *range* is crucial to specify the performance of the allocator. Because it is important to decide the minimum size of the memory block. If the value of the *range* is big, then it causes more consumption of memory space for the storing information like extra bits and pointers. Conversely, If the value of the *range* is too small, then it increases the internal fragmentation accordingly.

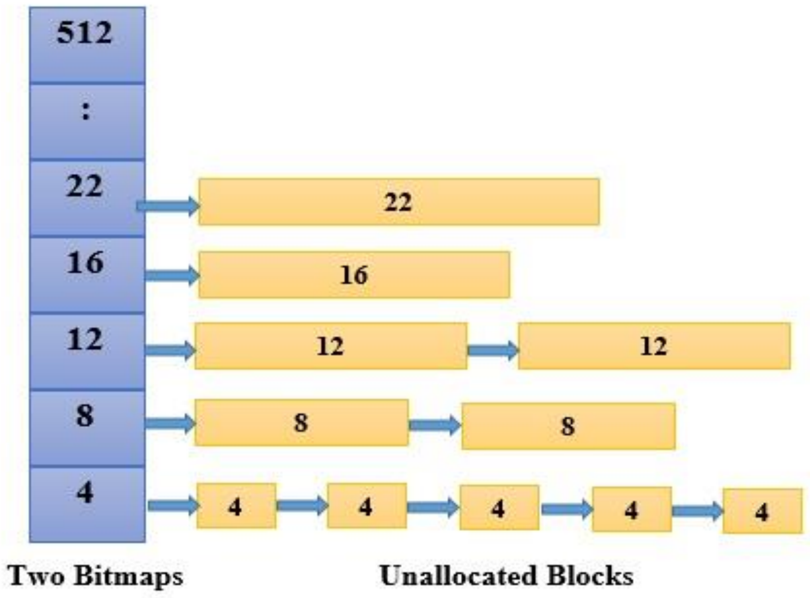


Figure 3.3: DmRT Structure for Small Block Allocation

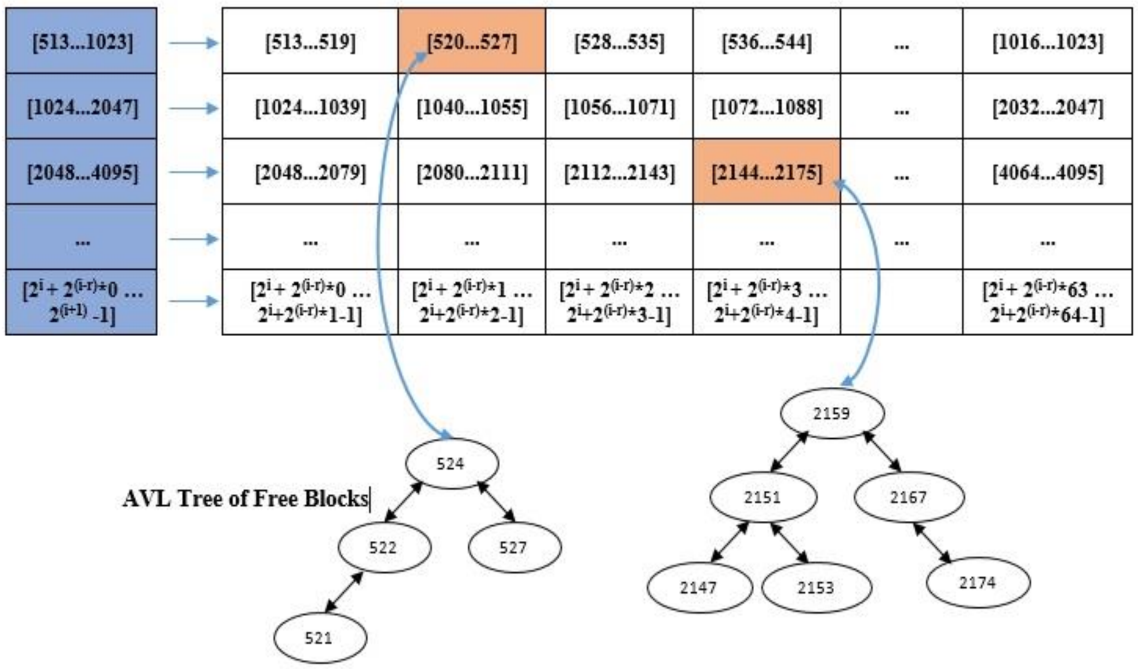


Figure 3.4: DmRT Structure for Normal Block Allocation

Therefore, the index i denotes the existing maximum size of a memory block: $2^{i+1} - 1$, while the number of segregated lists in the provide sizes can be defined by the number of ranges: 2^{range}. Furthermore, a specific segregated list can be identified by the value of index I (i, j), and the value of index I specify whether the list (i, j) encompasses any unallocated blocks or not. Hence, all bitmaps do not comprise unallocated memory blocks, but they specify the probable availability of a particular size of the memory block. All pointers to unallocated memory blocks are kept in two-dimensional pointer array which is known as matrix.

As discussed, for this proposed allocate, the value of *range* is set to 6 by default, each and every component of the array indicates to a list which has unallocated memory blocks of sizes in a range from $2^i + 2^{(i-range)} \times j$ to $2^i + 2^{(i-\mathrm{range})} \times (j+1) - 1$.

According to the employment of this allocator, it employs two-dimensional bitmap arrays, which needs a 64-bit variable for the primary bitmap and two-time 64-bit variables for the secondary bitmaps, therefore total 66 variables of 64-bit to specify the unallocated block lists are required. Also in each secondary level range, all available memory blocks arranged in AVL tree inorder to balance the tree structure.

The Primary Level is intended to accomplish the time of execution in a constant manner for allocation of memory blocks. Every segregated list keep the specific size of unallocated memory blocks, and the proposed algorithm can find an unallocated block by an index computed using equation 3.1. The Primary Level is designed using bitmaps and singular linked lists which contain small sizes of memory blocks, and also it is designed using a bitmap, arrays of pointers to unallocated blocks and doubly-linked lists for normal sizes of memory blocks. Sharing a single global heap between more than one threads having a tendency to raise the possibility of lock conflicts. To decrease this, every thread of the application should have a private thread heap.

$$\mathrm{ISL}\ (\mathrm{PI}, \mathrm{SI}) = \begin{cases} PI = \lfloor \log_2 RB \rfloor & where\ PI \in [9,31] \\ SI = \left\lfloor \dfrac{(RB - 2^{PI})}{2^{PI-range}} \right\rfloor & where\ SI \in [0, 63] \end{cases} \qquad 3.1$$

3.1.4 Strategy for selecting Remote Memory

We have proposed one memory allocator which can work on NUMA based architecture for the real-time operating system. Figure 3.5 shows a schematic diagram of NUMA based architecture for RTOS. As shown in the figure, there are total four nodes where each node having two processors, each processor within a node are connected with a bus, and all nodes are connected with shared memory. Also, each processor having their own local (private) memory.

Whenever any processor requires any memory block it will first check into its local memory, if the required memory block is available then it will allocate the same block from the local memory and if not then it will try to access from the shared memory, if the memory block is there in shared memory then it will allocate but if it is not there then it will ask for another processor which is lightly loaded in terms of memory. Now, what is lightly loaded processor? Let’s check it.

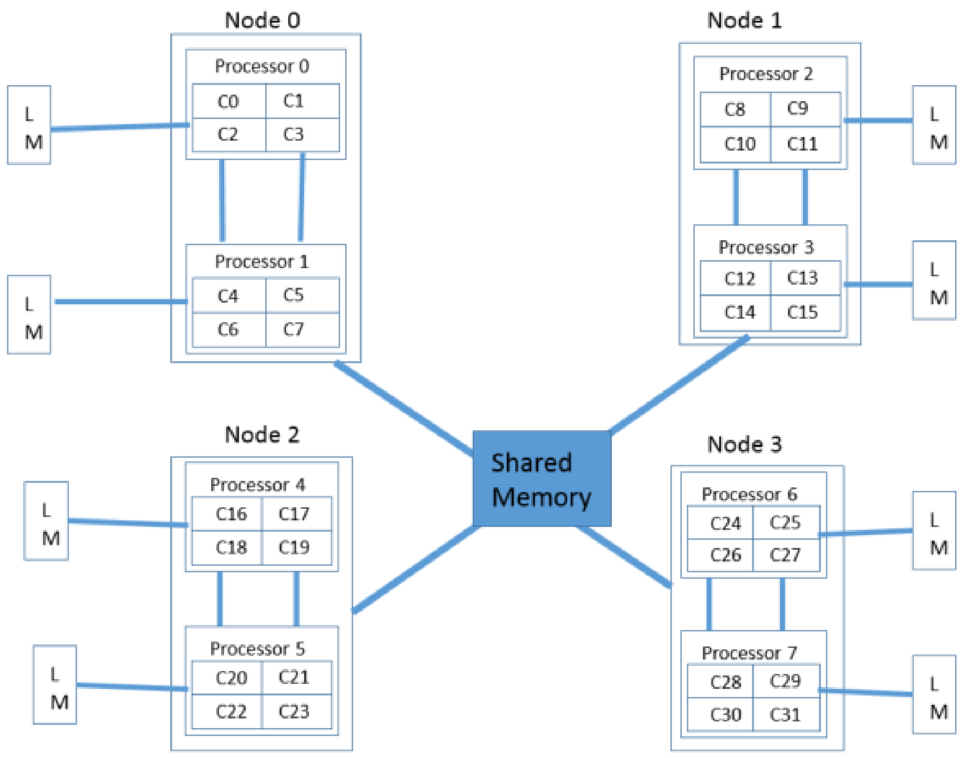


Figure 3.5: Complex NUMA Structure (4 Nodes)

According to memory utilization, each processor will be categorized into four categories.

1) Ideal

2) Heavily Loaded

3) Normal Loaded

4) Lightly Loaded

The first step is to calculate the load average for memory utilization for all processors using following equation [32] [38].

$$Mem_{u\_avg} = \frac{Mem_{u1} + Mem_{u2} + Mem_{u3} + \cdots + Mem_{un}}{n} \qquad 3.2$$

The second step is to find the upper and lower threshold value for memory utilization using following equation.

$$T_U = H \times Mem_{u\_avg}$$
$$T_L = L \times Mem_{u\_avg} \qquad 3.3$$

Where, T_U = upper limit of threshold,
T_L = lower limit of threshold,
U and L are constants. (U >1 and L< 1)

In the proposed algorithm, U and L are set to be 1.3 and 0.7 respectively which interpret if memory utilization is 30% above the Mem_{u_avg}, it is heavily loaded. And if memory utilization is 70% of the Mem_{u_avg}, it is lightly loaded; otherwise, it is normally loaded.

Hence, Light weight Memory <= 35% of Threshold value

Heavy weight Memory >= 65% of Threshold value

Average Memory node > 35% to < 65%

Ideal Memory < 10%   Threshold value

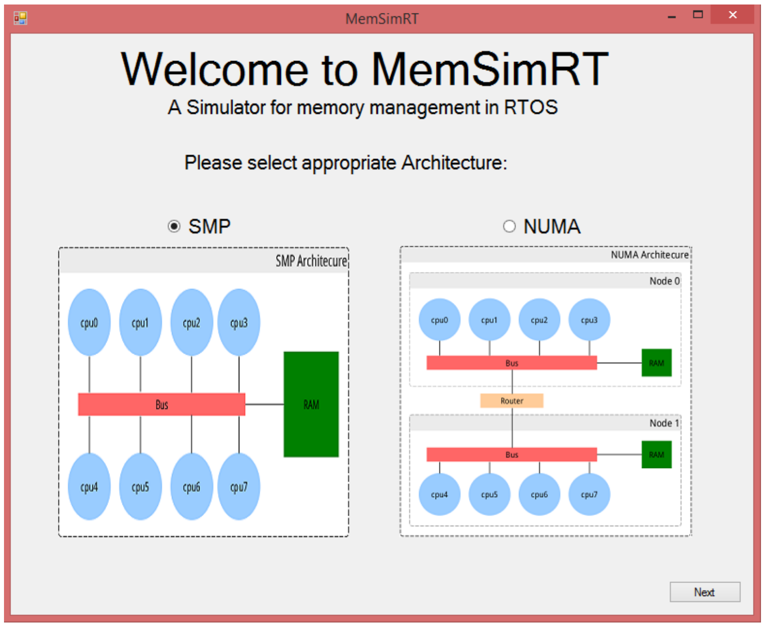
And then select appropriate processor’s memory for allocating memory.

Chapter 4
Results & MemSimRT

4.1 MemSimRT

There are so many simulators available to simulate different test cases for scheduling in a real-time operating system like Litmus-RT, Mark3 etc. But till date, no such simulator is available for simulating memory management algorithm for RTOS. So MemSimRT has been designed to simulate various memory allocators for both SMP as well as NUMA architecture based RTOS. Its front end created in C# while back-end developed using python.

Download MemSimRT using this QRcode:


MemSimRT

Welcome to MemSimRT

A Simulator for memory management in RTOS

Please select appropriate Architecture:

[✓ SMP] NUMA

Figure 4.1: The welcome screen of MemSimRT

Figure 5.1 shows the welcome screen of MemSimRT. So it is the Home screen of the simulator. As per our dynamic memory allocator, it has two alternatives. One is SMP, i.e., Symmetric multiprocessor and second is NUMA, i.e., Non-uniform memory access based architecture. Basically, in this simulator, NUMA is designed for eight processors, but it can be modified as per requirement by slightly changing the script.

4.2 Results

There are five different test cases.

1. SMP

Case 1: Existing allocators and DmRT allocate from Local Memory

As it is a Symmetric MultiProcessor architecture, all processors will share the same memory which is known as local memory for them. And whenever any request for the memory block is raised then, the memory manager will search and allocate memory block from the same local memory.

2. NUMA

Case 2: Existing from Local and DmRT Follow Local → Shared → Ideal

Existing allocators means Dlmalloc, tcmalloc and TLSF will allocate the memory block from local memory while DmRT will first try to allocate block from local memory; if it fails then it will attempt the same from shared memory and still if it will get failure then it will find ideal memory which has been discussed earlier and then it will allocate block from it. As DmRT tries to find memory block from three different types of memory, its execution time will be more than the other allocators, but it provides consistent execution time. And also it will satisfy a maximum number of the request as well as it will have less fragmentation due to proposed allocator structure.

Case 3: Existing allocators from Local and DmRT from Ideal

In this case, all existing allocator will allocate memory block from Local memory only. While DmRT first finds the idle memory and then it will allocate memory block from it. Here, existing allocators allocating blocks only from local memory that's why it can have less number of request satisfaction while DmRT will have a maximum number of request satisfaction. Also, other parameters will be best due to its structure.

Case 4: Existing allocators and DmRT both from Ideal

In this case, existing allocators and DmRT both will first find idle memory and then allocate a block from it. As existing allocators and DmRT, both allocate memory from ideal memory, execution time will be moreover same, but still, DmRT will have a maximum number of request satisfaction and less fragmentation.

Case 5: Existing allocators and DmRT follow Local → Shared → Ideal

In this case, existing allocators and DmRT both will first try to allocate memory block from local; if they fail then they will try to allocate the same block from shared memory and still got the failure then they will find idle memory and try to allocate same memory block from it. Though both existing allocators and DmRT follow the same path from allocating memory, proposed allocator defeats all of them in each parameter.

In each case, there are three different test categories have been selected.

a. Best case, i.e. test has been taken for 100 memory blocks request.
b. Average case, i.e. test has been taken for 1000 memory blocks request.
c. Worst case, i.e. test has been taken for 2000 memory blocks request.

There are three main parameters are considered for the results.

Parameter 1: Execution time. It should be consistent and minimum.

Parameter 2: Fragmentation. It should be as low as possible.

Parameter 3: Number of requests satisfied: It should be as high as possible.

Here, total four memory management algorithms have been compared.

a. Dlmalloc b. tcmalloc c. TLSF d. DmRT

All tests have been done on MemSimRT.

Table 4.1: Results of All Allocators in All Test cases

Parameter		Execution Time				Fragmentation				Request Satisfied			
Test No.	**Cases**	**Dlmalloc**	**Tcmalloc**	**TLSF**	**DmRT**	**Dlmalloc**	**tcmalloc**	**TLSF**	**DmRT**	**Dlmalloc**	**Tcmalloc**	**TLSF**	**DmRT**
Case 1	**Best Case**	287.8581	330.3003	268.598	234.6128	43.6472	29.684	22.4791	17.5031	56.6156	62.5883	81.5737	87.6169
	Average Case	1904.826	2890.503	1461.272	1067.995	52.3926	35.157	27.0205	22.0902	45.458	57.4617	74.9894	83.109
	Worst Case	3204.577	4352.133	2153.912	1847.152	60.4389	43.6719	32.0433	26.9948	34.903	52.783	70.9776	77.3786
Case 2	**Best Case**	326.2426	410.8068	290.2026	374.3901	43.5037	36.6849	21.5874	10.5141	57.5068	62.3301	76.6088	94.6614
	Average Case	2013.324	2988.474	1522.335	2303.212	52.5491	44.4944	29.5867	15.6241	45.2403	54.2546	65.6095	87.4181
	Worst Case	3202.561	4348.651	2049.025	3361.854	61.4645	43.3702	36.5828	20.5572	35.5009	50.4204	61.5822	83.7309
Case 3	**Best Case**	338.0589	420.5202	296.4418	245.4583	45.2763	33.6326	24.0237	15.4697	57.7885	64.1904	76.9458	89.9526
	Average Case	2054.716	2995.241	1539.964	1115.835	53.8153	44.9067	30.2955	19.5226	46.4232	54.5453	65.9168	82.8598
	Worst Case	3283.047	4288.159	2064.704	1785.676	60.4389	43.6719	32.0433	26.9948	34.903	52.783	70.9776	77.3786
Case 4	**Best Case**	374.8572	444.5905	319.6948	249.566	35.2178	26.3902	19.2872	14.5781	67.5358	72.1999	83.1096	89.1851
	Average Case	2110.58	3240.527	1530.277	1149.484	43.0248	35.5497	26.3408	19.7854	55.5939	64.722	73.3219	81.1776
	Worst Case	3277.428	4467.102	2172.658	1860.321	52.6015	43.6602	35.9747	25.1212	45.1788	53.4233	65.3064	74.1789
Case 5	**Best Case**	528.5204	636.5573	480.8449	385.8492	31.4948	23.8764	17.5356	10.4119	71.7431	78.1612	86.8777	93.7361
	Average Case	2928.034	4166.439	2541.517	2252.019	38.895	31.6255	22.913	15.0111	62.0389	70.6678	79.9134	87.5092
	Worst Case	4231.555	5107.635	3684.495	3367.702	47.3835	38.2598	30.8472	19.1314	52.5833	61.8889	74.2487	83.9386

As shown in the table, DmRT performs better concerning all other allocators (Dlmalloc, tcmalloc and TLSF) in all cases. Only in Case 2 where all existing allocators allocate memory from Local memory only and DmRT allocates from Local, Shared and Ideal in which execution time is more than Dlmalloc and TLSF but in same case number of request satisfied is maximum than others as well fragmentation is minimum than other allocators.

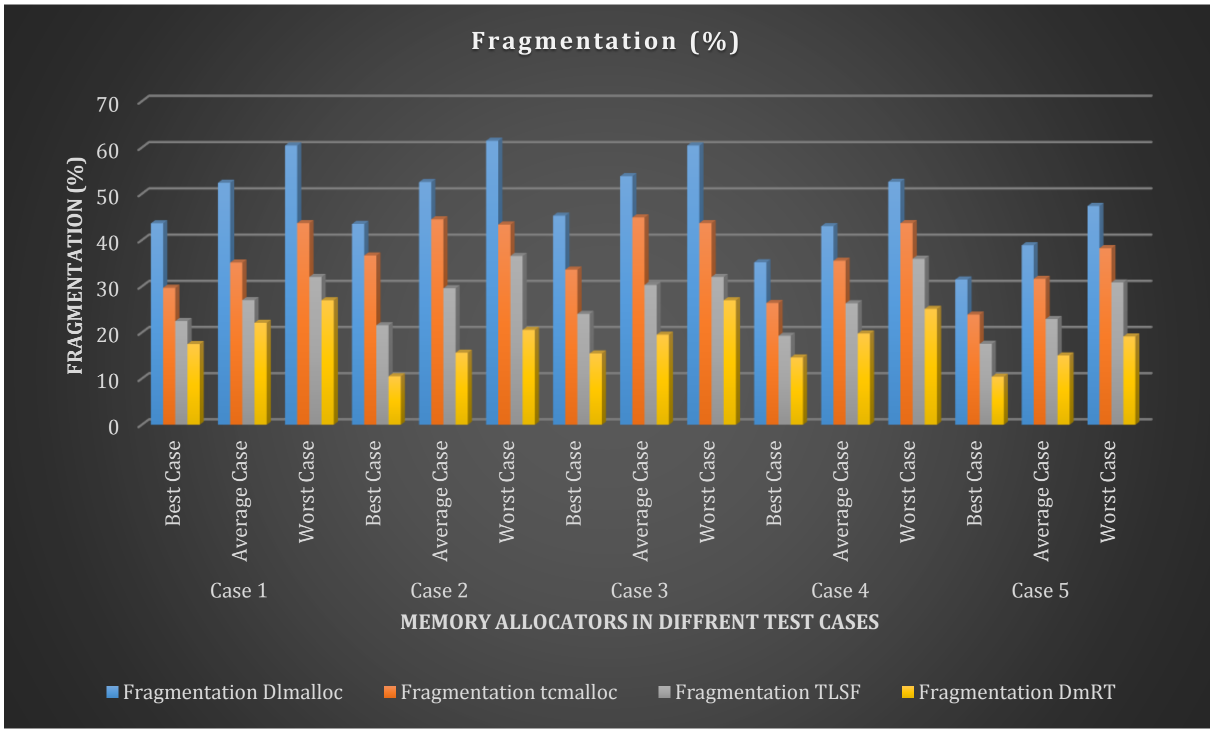


Figure 4.2: Fragmentation in % of all Allocators in All Test cases

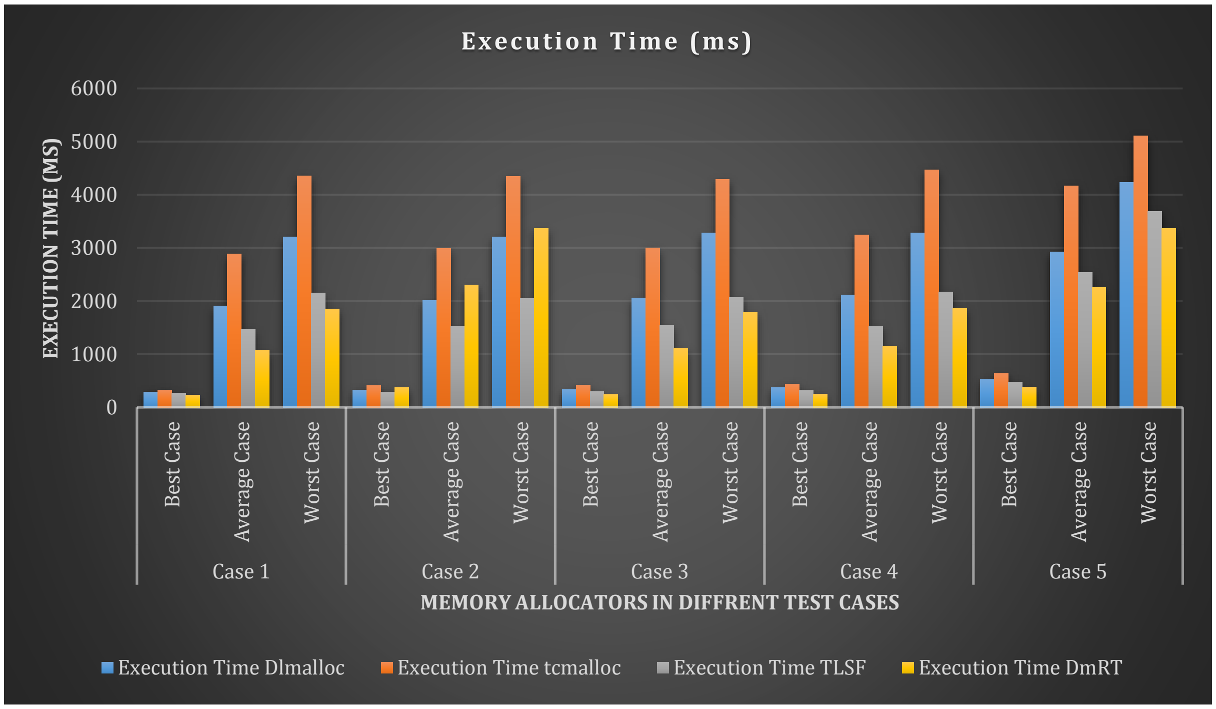


Figure 4.3: Execution time in (ms) of all Allocators in All Test cases

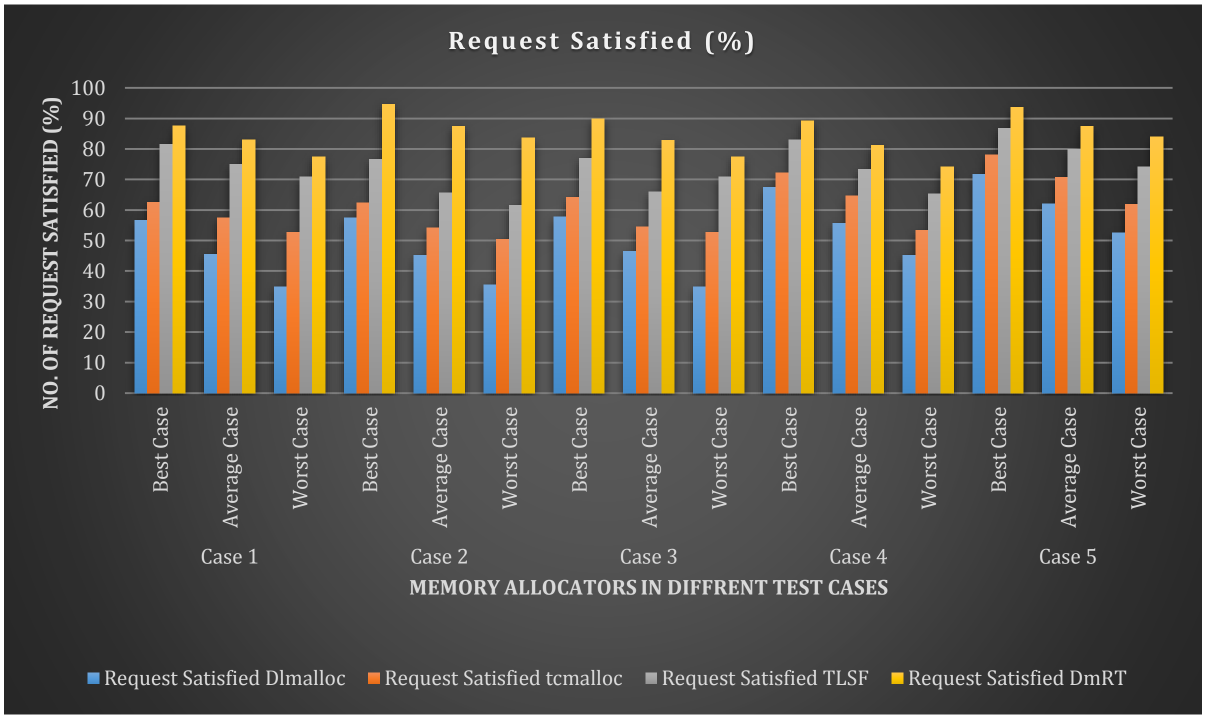


Figure 4.4: No. of Request Satisfied in % of all Allocators in All Test cases

References

[1]. Bays, C. (1977). A comparison of next-fit, first-fit, and best-fit. Commun. ACM, 20(3): (pp. 191–192).

[2]. Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson, P. R. (2000). Hoard: a scalable memory allocator for multithreaded applications. SIGPLAN Not., 35(11): (pp. 117–128).

[3]. Christian Del Rosso. (2005). Dynamic Memory Management for Software Product Family Architectures in Embedded Real-Time Systems. Fifth Working {IEEE} / {IFIP} Conference on Software Architecture (pp. 211-212)

[4]. Dipti Diwase, Shraddha Shah, Tushar Diwase and Priya Rathod. (2012). Survey Report on Memory Allocation Strategies for Real-time Operating System in Context with Embedded Devices. International Journal of Engineering Research and Applications, Vol. 2, Issue 3, (pp.1151-1156).

[5]. Edge, J. (2009). Perfcounters added to the mainline. http://lwn.net/Articles/336542/.

[6]. Ferreira, T., Matias, R., Macedo, A., and Araujo, L. (2011). An experimental study on memory allocators in multicore and multithreaded applications. In Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2011 12th International Conference on, (pp. 92 –98).

[7]. FSF, F. s. f. (2012a). Glibc, the gnu c library. ”http://www.gnu.org/software/libc/libc.html”.

[8]. FSF, F. s. f. (2012b). The gnu c++ library manual. “http://gcc.gnu.org/onlinedocs/libstdc++”.

[9]. Gergov, J. (1996). Approximation algorithms for dynamic storage allocation. In Algorithms — ESA ’96, volume 1136, pages 52–61. Springer Berlin / Heidelberg.

[10]. Gloger, W. (2006). ptmalloc2. ”http://www.malloc.de/en/”.

[11]. Hans-Georg Eßer. (2011) Combining memory management and filesystems in an operating systems course. Proceedings of the 16th Annual {SIGCSE} Conference on Innovation and Technology in Computer Science Education, Darmstadt, Germany.

[12]. Hasan, Y. and Chang, M. (2005). A study of best-fit memory allocators. Computer Languages, Systems & Structures, 31(1): (pp. 35 – 48).

[13]. Hasan, Y., Chen, W.-M., Chang, J. M., and Gharaibeh, B. M. (2010). Upper bounds for dynamic memory allocation. IEEE Trans. Comput., 59(4): (pp. 468–477).

[14]. Hewlett-Packard Corporation (2012). HP Pro-Liant DL980 G7 server with HP PREMA Architecture PREMA Architecture. Technical Whitepaper.

[15]. Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., and Toshiba Corporation (2011). Advanced configuration and power interface specification.

[16]. Hirschberg, D. S. (1973). A class of dynamic memory allocation algorithms. Commun. ACM, 16(10): (pp. 615–618).

[17]. Jane W. S. Liu. (2000). “Real-time System”, 1st Edition published by Person Education.

[18]. Johnstone, M. S. and Wilson, P. R. (1998). The memory fragmentation problem: solved? SIGPLAN Not., 34(3): (pp. 26–36).

[19]. Knuth, D. (1997). The art of computer programming: Fundamental Algorithms, volume 1. addison-Wesley, 2 edition.

[20]. Lea, D. (1996). A memory allocator. ”http://g.oswego.edu/dl/html/malloc.html”. Unix/Mail December, 1996.

[21]. Lei Liu, Mengyao Xie, Hao Yang. (2017). Memos: Revisiting Hybrid Memory Management in Modern Operating System. CoRR abs/1703.07725

[22]. Lei Liu, Yong Li, Chen Ding, Hao Yang, Chengyong Wu. (2016). Rethinking Memory Management in Modern Operating System: Horizontal, Vertical or Random? IEEE Trans. Computers 65(6): (pp. 1921-1935)

[23]. Linus Torvalds, e. (2011). Source codes of linux kernel v3.0.4. ”http://lxr. linux.no/linux+v3.0.4/”.

[24]. Marchand, A., Balbastre, P., Ripoll, I., Masmano, M., and Crespo, A. (2007). Memory resource management for real-time systems. In Real-Time Systems, 2007. ECRTS '07. 19th Euromicro Conference, (pp. 201 – 210).

[25]. Masmano (2012). The lastest version of TLSF source. http://wks.gii.upv.es/tlsf/files/src/TLSF-2.4.6.tbz2.

[26]. Masmano, M., Ripoll, I., and Crespo, A. (2003). Dynamic storage allocation for real-time embedded systems. Proc. of Real-Time System Simposium WIP.

[27]. Masmano, M., Ripoll, I., Balbastre, P., and Crespo, A. (2008a). A constant-time dynamic storage allocator for real-time systems. Real-Time Systems, 40(2): (pp. 149–179).

[28]. Masmano, M., Ripoll, I., Real, J., Crespo, A., and Wellings, A. (2008b). Implementation of a constant-time dynamic storage allocator. Software: Practice and Experience, 38(10): (pp. 995–1026).

[29]. Ogasawara, T. (1995). An algorithm with constant execution time for dynamic storage allocation. In RTCSA '95: Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications, pages 21–25, Washington, DC, USA. IEEE Computer Society.

[30]. Ogasawara, T. (2009). Numa-aware memory manager with dominant-threadbased copying gc. SIGPLAN Not., 44(10): (pp. 377–390).

[31]. Page, I. and Hagins, J. (1986). Improving the performance of buddy systems. Computers, IEEE Transactions on, C-35(5): (pp. 441 –447).

[32]. Paul Werstein, Hailing Situ, Zhiyi Huang. (2006). "Load Balancing in a Cluster Computer", Seventh International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'06).

[33]. Puaut, I. (2002). Real-Time Performance of Dynamic Memory Allocation Algorithms. In ECRTS '02: Proceedings of the 14th Euromicro Conference on Real-Time Systems, (pp. 41–49), Washington, DC, USA. IEEE Computer Society.

[34]. Puaut, I. and Hardy, D. (2007). Predictable paging in real-time systems: A compiler approach. In Real-Time Systems, 2007. ECRTS '07. 19th Euromicro Conference on, (pp. 169 –178).

[35]. Robart L. Budzinski, Edward S. Davidson. (1981). A Comparison of Dynamic and Static Virtual Memory Allocation Algorithms" IEEE Transactions on software Engineering, Vol. SE-7, NO. 1.

[36]. Sanjay Ghemawat, P. M. (2010). Tcmalloc: Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[37]. Seyeon Kim. (2013). Node-oriented dynamic memory management for real-time systems on ccNUMA architecture systems. University of York, UK.

[38]. Vatsal Shah, Kanu Patel. (2012). Load Balancing algorithm by Process Migration in Distributed Operating System. International Journal of Computer Science and Information Technology & Security (IJCSITS), ISSN: 2249-9555, Vol. 2, No.6.

[39]. V Shah, A Shah. (2017). Critical Analysis for Memory Management Algorithm for NUMA based Real-time Operating System. IEEE Xplore.

[40]. V Shah, A Shah. (2018). Proposed Memory Allocation Algorithm for NUMA based Soft Real-time Operating System. International Conference On Emerging Technologies In Data Mining And Information Security (IEMIS 2018)

[41]. Vatsal Shah, Apurva Shah. (2016). An Analysis and Review on Memory Management Algorithms for Real-time Operating System. International Journal of Computer Science and Information Security (IJCSIS), Vol. 14, No. 5.

[42]. Vee, V.-Y. and Hsu, W.-J. (1999). A scalable and efficient storage allocator on shared memory multiprocessors. In Proceedings of the 1999 International Symposium on Parallel Architectures, Algorithms and Networks, ISPAN '99, Washington, DC, USA. IEEE Computer Society.

[43]. Wellings, A. J., Malik, A. H., Audsley, N. C., and Burns, A. (2010). Ada and cc-numa architectures what can be achieved with ada 2005? Ada Lett., 30(1): (pp. 125–134).

[44]. Wilson, P. R., Johnstone, M. S., Neely, M., and Boles, D. (1995b). Dynamic Storage Allocation: A Survey and Critical Review. In IWMM ’95: Proceedings of the International Workshop on Memory Management, (pp. 1–116), London, UK. Springer-Verlag.

[45]. Wilson, P., Johnstone, M., Neely, M., and Boles, D. (1995a). Memory allocation policies reconsidered. Technical report, Technical report, University of Texas at Austin Department of Computer Sciences.

[46]. XiaoHui Sun, JinLin Wang, xiao chan. (2007). “An Improvement of TLSF Algorithm”.

[47]. Youngki Chung, Ramakrishna M, Jisung Kim and Woohyong Lee. (2008). Smart Dynamic Memory Allocator for embedded systems. Proceedings of 23rd International Symposium on Computer and Information Sciences, ISCIS '08.

[48]. Zorn, B. and Grunwald, D. (1992). Empirical measurements of six allocation-intensive c programs. SIGPLAN Not., 27(12): (pp .71–80).

[49]. Zorn, B. and Grunwald, D. (1994). Evaluating models of memory allocation. ACM Trans. Model. Comput. Simul., 4(1): (pp. 107–131).

Publications

1. Vatsalkumar H. Shah, Dr. Apurva Shah, (May, 2016). ***"An Analysis and Review on Memory Management Algorithms for Real-time Operating System"*** *published in International Journal of Computer Science and Information Security, Volume 14, Issue 5, (pp. 236-240)* (Web of Science Thomson Reuters, Scopus, DOAJ)

2. Vatsalkumar H. Shah, Dr. Apurva Shah, (December, 2017). ***"Critical Analysis for Memory Management Algorithm for NUMA based Real-time Operating System".*** *In Proceedings of IEEE Conference, 2017 (International Conference on Intelligent Sustainable Systems 2017.)****.*** (pp. 323-327). (INSPEC, Scopus Indexed)

3. Vatsalkumar H. Shah, Dr. Apurva Shah, (February, 2018). ***"Proposed Memory Allocation Algorithm for NUMA based Soft Real-time Operating System".*** As a book chapter in *Advances in Intelligent Systems and Computing (AISC), Springer Series*. (ISI, DBLP, EI-Compendex, SCOPUS) *(To be published)*

4. Vatsalkumar H. Shah, Dr. Apurva Shah, (June, 2018). ***"Memory Allocator for SMP & NUMA based Soft Real-time Operating System".*** As a book chapter in *Advances in Intelligent Systems and Computing (AISC), Springer Series*. (ISI, DBLP, EI-Compendex, SCOPUS) *(To be published)*