Chapter 2
Literature Review

2.1 Dynamic Memory Management

It is one of the significant procedures in the present world of engineering to accomplish tasks generated during execution of programme in any of the high-level programming. It is about to arrange unallocated or allocated memory blocks which have a smaller life scope than their parent task, job or process. It is highly challenging to fulfill the criteria of timing restriction of real-time applications. The reason behind this is any real-time application requires the prediction of worst-case execution time (WCET) in dynamic memory management [96]. Furthermore, searching the ideal location for allocating a memory block is an NP-hard problem if certain memory blocks are previously allocated [30] [38] [94]; and if a memory management algorithm cannot satisfy the request of memory block, it will lead to fragmentation, though the total size of available memory might be more than the requested block of memory size.

If dynamic memory management is used on multiprocessor architecture, some new complications are raised like false sharing, synchronization between threads. The reason behind this is that the needs of real-time applications vary from one architecture to another [31] [32].

In this section, the above issues will be explained based on the research work that has been carried out so far along with the literature review of memory management algorithms. Section 2.2 introduces some essential problems in memory management like fragmentation, false sharing and heap [93]. In Section 2.3, a range of traditional and nontraditional memory management algorithms have been discussed.

2.1.1 Jargons

In 1995, Wilson [116] and his team have given certain terms or jargons that are commonly used in the area of memory management. The following text discusses a few jargons used like Strategy, Policy and Mechanism.

1) Strategy:

A strategy is a fundamental methodology used in any memory allocation algorithm. It considers different configurations in the program and defines a variety of suitable rules for memory blocks allocation dynamically. The goals of an allocation algorithm can be described as being a correspondent in significance to the allocation algorithm's strategy, For example, "reducing lock contentions", "increasing data locality" etc. These all strategies can be accomplished by policies.

2) Policy:

A policy is a conclusion method for allocating blocks of memory dynamically. It decides precisely from where an allocated block will be removed or at what place in memory an unallocated block will be put in. For example, one policy defines like: "each time discover the minimum block of memory which is large enough to fulfill the memory request". These selected policies are employed by different mechanisms. Some policies are Best-Fit, Exact-Fit, First-Fit, Next-Fit, Good-Fit, and Worst-Fit.

3) Mechanism:

A mechanism is nothing but the implantation of policy. It is a group of different algorithms and data structures. For example, "use a singly linked-list and find the location of unallocated memory block list from where the previous request was fulfilled; the unallocated blocks are inserted at the end of the singly linked-list". Normally, the mechanism may be defined as Sequential Fit, Segregated Fit, Buddy Systems, Indexed Fit and Bitmapped Fit [64] [67].

These all jargons and their definitions are significant for accepting and planning a dynamic memory management algorithm. For instance, for a specific strategy, various policies can cause diverse effects. If a policy creates better locality with huge fragmentation, an application designer has to select an alternative policy under the similar strategy, which can generate low fragmentation. Any policy can be employed by a variety of mechanisms. If a specific policy achieves a good result, but its implementation is not well-organized, designers can employ other policy by selecting an altered or alternative mechanism [101]. Hypothetically, low fragmentation is one of the major characteristics of any dynamic memory allocator which can be achieved by a robust allocation policy. The allocation policy is the selection of memory block from the unallocated memory block list. It can be accomplished by two methods [44] [89] [109]: a) Splitting b) Merging

a) Splitting:

In this allocation policy, large size memory block will be split into a number of small size memory blocks, and it will use a large separated memory block to fulfill a specific memory block request. Normally, the remaining memory blocks will be traced as unallocated memory blocks, and will be used to fulfill upcoming memory block requests [49].

b) Merging:

Merging is used when any applications, jobs or tasks release allocated memory blocks. When applications release any memory blocks, a virtual component called as memory manager finds whether any neighboring memory blocks are released, unallocated or not. And if they are released then it coalesces these memory blocks into a single large size memory block [49]. This is required as a large size memory block is more convenient than two small-sized memory blocks.

Merging process can be divided into two different classes. Primarily, immediate merging tries to merge the unallocated memory blocks instantly whenever a block is released. However, immediate merging is costly because any released memory block will be merged together by continually and regularly coalescing the neighboring unallocated memory blocks. Secondary, deferred merging simply notes a released memory block as “unallocated” or “released” without combining. Because most of the applications frequently generate similar size memory blocks of short life span this type of memory management algorithm maintains memory blocks of a specific size on an unallocated list and reclaims them without merging and splitting. Due to this, if an application demands the similar sized block of memory immediately after one memory block is freed, the memory block request can be fulfilled by simple manipulation; this may enhance if some specific size of memory blocks are frequently allocated and unallocated. But the disadvantage of differed merging technique is that it has unbounded response time which causes fragmentation.

2.2 Fundamental Issues

To design any memory allocator, two important things must be considered, first is the time efficiency and second is the space efficiency. Without making a negotiation between these two, it is hardly possible to develop a memory allocator which is having low fragmentation and low response time for most of the applications [17] [22]. For example, the memory allocator designed

by Kingsley [48] is the best example of simple segregated storage algorithms. In this allocator, each request of the memory block is converted into a nearest large number which is a power of two. In its allocation phase, it tries to remove memory block from the array, and in deallocation phase, it tries to insert memory block into an array. Due to the simple design of this algorithm, its performance is very efficient and fast and it doesn't try to merge the adjacent memory blocks. Hence, the internal fragmentation is very high. So, an important criterion to design a dynamic memory allocator is to maintain a balance between time and space efficiency.

For memory allocators of multiprocessor architecture, several other problems arise like false sharing, heap conflicts, boundless increasing heap problems and conventional fragmentation.

2.2.1 Fragmentation

Among the various issues, the most critical problem of a memory allocation algorithm is memory fragmentation. Randell had categorized fragmentation in two classes [90]: internal and external. These are generated due to splitting and merging of memory blocks as explained earlier.

1) External Fragmentation

External Fragment will be generated when the demanded block of memory cannot be served even though the total unallocated memory is more than demanded memory block size. While allocating and deallocating a memory block, external fragmentation is created due to a large number of small-sized unallocated memory blocks which are known as 'holes'. The large number of small size unallocated memory blocks are not adjacent to each other, hence cannot be combined, and they are too minor to fulfill any demand.

2) Internal Fragmentation

Internal fragmentation is generated when the memory management algorithm provides a large-sized unallocated memory block to fulfill the demand, instead of the actual demanded block size. The remaining part of the allocated memory block will have no use. That's why this scenario is known as internal fragmentation. The remaining memory block remains included in an assigned memory block. In certain memory allocation algorithms, internal fragmentation is allowed up to a certain limit for improved performance or easiness. In contrast to external fragmentation, internal fragmentation is exceptional to the respective implementation of a memory management algorithm

and so it must be analyzed according to the application. Hence, it is difficult to discover a common analysis of internal fragmentation anywhere.

2.2.1.1 Fragmentation Measurement Parameters

For comparison of memory management algorithms, some unit of measure is required. Normally, the cost of time and space are used as a unit of measure. The cost of time means the execution time of process and cost of space means fragmentation in memory. Through various ways, the fragmentation in memory can be demonstrated. For example, there are 15 unallocated memory blocks of size 8Kb and 20 unallocated memory blocks of size 2Kb, and if an application demands 10 unallocated memory blocks of size 8Kb and 15 unallocated memory blocks of size 2Kb. In this situation, there should not be any fragmentation as the demanded memory blocks are available but if some application requires 15 unallocated memory blocks of size 16Kb then it would create a problem and demand will not satisfied because of the huge amount of fragmentation.

In his work, Johnstone [46] proposed four different unit of measure to define the amount of fragmentation.

Method 1: The fragmentation can be measured as the total memory usage by the memory allocator over the amount of memory demanded by the application, aggregated at all points over the time. This method of fragmentation measurement is quite simple but this method hides the spikes in the utilization of memory and due to these spikes fragmentation may create problems.

Method 2: The fragmentation can be measured as the total memory usage by the memory allocator above the highest memory needed by the application at the point of highest utilization of memory. The issue with this method is that the point of highest utilization of memory cannot normally be measured at runtime.

Method 3: The fragmentation can be measured as the highest amount of memory used by the memory allocator over the actual memory needed by the application at the time of maximum memory used by the memory allocator. The problem with this method is that

it will create high fragmentation, even if the applications use small amount of memory more than the size of required memory.

Method 4: The fragmentation can be measured as the highest amount of memory usage by the memory allocator over the highest amount of memory usage by the application. The drawback of this method is that when an application uses small memory, it can show low fragmentation.

Each method stated above is used for measurement of fragmentation generated by the application. The issue with these all measures is that none of the methods differentiated between unallocated memory and allocated memory for its self-arrangement of data, such as maintaining unallocated blocks. In this research, the internal fragmentation and external fragmentation have been considered with space used by data structures. The following equation [46] has been used to compute the exact amount of fragmentation (*frag*):

$$frag = \frac{h-a}{h} \qquad 2.1$$

Where, h is the actual amount of memory provided by the allocator, and points out the amount of allocated memory requested by the application.

2.2.2 False Sharing

The application which includes multiple threads has so many challenges like race conditions between threads, different types of conflict such as contentions, debugging of threads and the most obvious problem, the deadlock conditions. But uniprocessor system cannot cop up in the modern and faster trends of multiprocessor environment. So, memory management algorithm must be smart enough to resolve the issues created due to simultaneous execution of threads which are demanding different or same memory block at the same time.

Sharing resources between parallel executing thread is the most obvious condition in any application with multithreading. Now, the conflict between threads always happens when two or more thread try to access shared resources. But this is not the only reason when the conflict arises, it may arise when all threads try to read or write different resources if multiple resources are

sharing same memory or cache. For example, thread1 modifies one object, say dummy_ob1, and at the same time, thread2 modifies another object dummy_ob2. Let us consider that these objects dummy_ob1 and dummy_ob2 exist in the same cache memory but the threads are executing on two different processors. If of these two objects, one is updated, then cache memory will be put in invalidate condition due to cache coherent rules and because of this situation, the performance of the application will be degraded [40]. This scenario is known as false sharing.

Due to the growing needs of a multi-processor system, the tendency of adding more cache memory size leads to the problem of false sharing [57]. It is also hard to remove false sharing automatically. To address this either the compilers adjust some logic of simultaneous looping or designer should use such data structure to avoid this problem [45]. This technique may break the association between objects of data and false sharing. Though the issue cannot be entirely removed as there is some impact of data structure which is based on an array; still, a better-planned approach can address this issue [6].

2.2.3 Types of Heap

The memory used by the memory allocator for allocation of a block is called heap. In a multiprocessor environment, a single heap will not be work [18] [87]. This section explains different types of the heap with its pros and cons [108]. Of these, anyone can be used according to its application.

1) Sequential Single Heap

Sequential single heap used by the memory management algorithm generally has lower fragmentation and faster execution time. Here, the heap is secured by a universal lock which leads the sequential accesses of memory and creates lock conflict.

2) Synchronized Single Heap

The usefulness of a synchronized single heap is to decrease the sequential accesses of memory and heap conflict. It is typically carried out by a synchronized data-structure such as a B-

tree or an unallocated memory block list with locks. But, its cost of accessing memory is comparatively high. Also, false sharing remains as it is.

3) Absolute Reserved Heap

An Absolute Reserved heap specifies that an isolated heap is assigned to every thread and these threads are distinct from other existing threads. For this, why every thread cannot use other reserved heap for the memory operations like read or write. Therefore, a memory management algorithm having various heaps can decrease the majority of the lock conflicts on the reserved heap, and become scalable. Inappropriately, it creates reserved heap to develop without any bounds. For example, two threads have a producer-consumer association where a thread T1 acts as a producer assigning memory Mem1 and another thread T2 acting as a consumer frees Mem1. In this, Mem1 must be further added into T2's heap.

4) Reserved heaps with rights

In contrast to the memory manager which uses absolute reserved heaps, the memory managers with rights provide unallocated memory blocks to the heap. However, in a producer-consumer association based applications, a round-robin way, the memory manager can remove the generated false sharing. It still generates boundless memory usage.

5) Reserved heaps with thresholds

The memory manager generates an order of heaps, and multiple threads can share some of the heap excluding the reserved heaps. The heap, which is shared, may demonstrate heap conflicts, in rare case. So, the memory managers may be effective and scalable. If reserved heaps cross the threshold value, some part of unallocated memory will be migrated to the shared heap. The cost of managing memory, especially the cost of time, may be huge because so many memory processes are required for this.

Majority memory allocators have a specific issue of boundless heap increments due to multiple heaps, where memory usage cannot be unbounded by any policy even if the requisite memory is fixed. It is known as blowup occurrence [6]. Some of the memory management algorithms like Hoard [6] and Vee [115] demonstrate restricted memory usage as they provide the reserved heaps with thresholds.

2.3 Memory Management Algorithms

Since 1963, several memory management algorithms have been designed and implemented. Majority of the current memory management algorithms are modifications of the traditional allocators. Furthermore, these traditional memory management algorithms are ordinary and very simple to use in minor devices [19]. There are two different approaches to analyze Worst-Case Execution Time of memory allocators: 1) Static Worst-Case Execution Time analysis 2) Worst Case Complexity analysis. But, without prior information of the allocation or deallocation demands, it is difficult to use Static Worst-Case Execution Time analysis [88] [91].

Now, different memory management algorithm such as sequential and segregated fit, buddy allocators, indexed/bitmapped fit and current memory management algorithms will be discussed.

2.3.1 Sequential Fit

Sequential fit allocators can be categorized into four categories: 1) Best-Fit 2) First-Fit 3) Next-Fit and 4) Worst-Fit. Figure 2.1 [100] demonstrates the variances between allocators of sequential fit. It consists of five unallocated memory blocks of dissimilar sizes with six allocated memory blocks. The sizes of the memory blocks are specified in the caption of every memory block which has indicators to represent a doubly link list.

For Example, one application demands 2Kb of unallocated memory. Here, as per their policies, the best-fit will provide the 2Kb unallocated memory block, 3Kb of unallocated memory block will be provided by First-fit and Next-fit whereas 5Kb unallocated memory block will be provided by Worst-fit. This has been demonstrated in Figure 2.1.

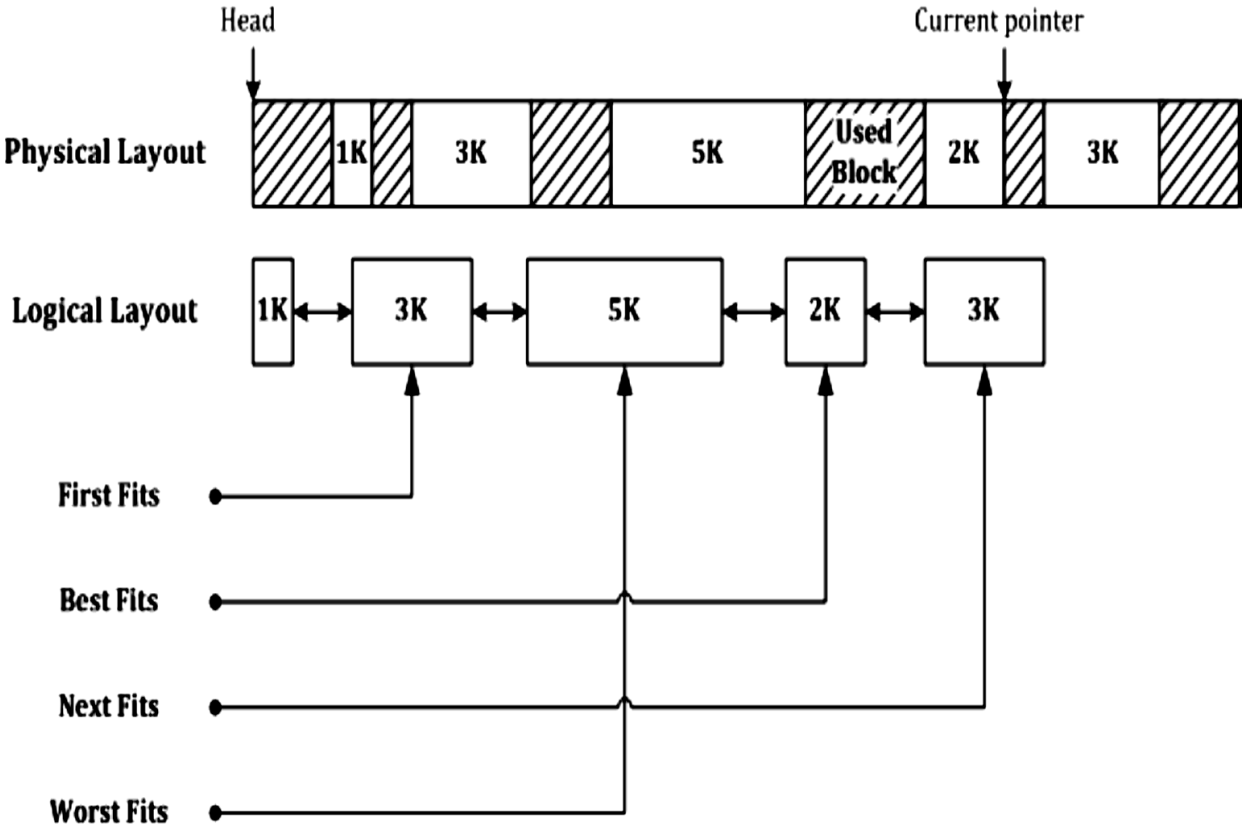

Figure 2.1: Structure for Sequential-Fit Allocators

2.3.1.1 Best-fit

The allocator is normally implemented using either a doubly or circular linked list. However, it is not only a single list of available unallocated memory blocks, but it has arrays of different category and sizes.

Generally, Implementation of best-fit policy can be done in various ways such as address-order, First-in-First-out and Last-in-First-out. These memory management algorithms are implemented based on in-depth searching algorithms. They also demonstrate proper and decent usage of memory but they also lead to some fragmentation [51]. Though, it is not an important issue [5] [117].

In best-fit allocator, the algorithm discovers for an unallocated memory block, which is big enough to fulfill the demand of the application, sequentially from the top of the unallocated block list till it finds an appropriate memory block. If the memory manager discovers an appropriate

block, the searching process ends, and it provides the memory block. If the memory manager discovers more than one appropriate memory blocks, the selection of memory block relies on the implementation of the allocator. If the size of the discovered memory block is bigger than the size demanded, the memory block will be split into two parts and the remaining part will be pushed into the unallocated block list, otherwise the memory manager will fail to allocate demanded block. In deallocation of the blocks, released blocks are combined with adjacent free memory blocks.

2.3.1.2 First-fit

The memory management algorithms which are employing First-fit policy are also most general sequential fit mechanisms [9] [51]. They try to discover the first unallocated memory block which is big enough to fulfill the demand of an application. Implementation of the First-fit policy can be done in various ways such as address-order, First-in First-out or Last-in First-out mechanisms. This one also, same as best-fit, can find a block in O(n) time complexity with the help of an array which is a doubly linked list [103] [104].

The First-fit allocator finds an unallocated block sequentially from the top of unallocated block lists until the demanded block is not found. If there is no suitable block found, it shows failure. If the size of the searched memory block is bigger than the demanded size, it is split into two parts and the remaining part will be pushed into the unallocated block list. In the deallocation of the blocks, released blocks are combined with adjacent free memory blocks.

The main issue with the first-fit policy is that repeated splitting happens on the top of the unallocated block list which creates so many small memory blocks adjacent on the top of the list. Due to these small memory blocks the searching time is increased as it requires going through entire list every time resulting into high fragmentation. Therefore, the traditional first-fit policy is not appropriate for an application which allocates and deallocates different sizes of memory blocks repeatedly. But, like best-fit, it may be more appropriate if implemented via additional classy data structures.

2.3.1.3 Next-fit

The allocation algorithms which employ next-fit policy are one of the variations of algorithms having a first-fit policy which has a moving pointer for memory blocks allocation [51]. The allocators preserve a pathway of the pointer which stores the position of the block where the previous search was successful. This pointer will be used as the starting point for the subsequent search. Implementation of the next-fit policy can be done in various ways such as address-order, first-in-first-out and last-in-first-out mechanisms.

Theoretically, the next-fit decreases the aggregate search time for a singular link list; but it tends to get inferior zone because it examines each unallocated memory block before searching the similar memory block due to the moving pointer.

As the pointer moves over the unallocated memory block lists frequently and is expected to store the objects in memory with different size and periods from different stages of the application's runtime. Therefore, it generates higher fragmentation than other sequential fit memory management algorithms. The allocators having a next-fit policy with last-in first-out mechanism have considerably lesser fragmentation than next-fit allocators implemented by address-order [117]. Next-fit is the finest solution to reduce the average response time in a mutually shared memory of SMP architecture [1] [3] [70].

Regarding allocation and deallocation of the memory blocks, the allocators having a next-fit policy are nearly identical to the first-fit allocators except the beginning point of finding.

In summary, sequential fit memory management algorithms are developed with the help of a linear list, which contains doubly or circular linked lists with various policies in like first-in-first-out or last-in-first-out. The best-fit as well as the first-fit are created using first-in-first-out or an address-ordered policy appears to work fine. Conversely, scalability is a significant issue with sequentially fit allocators because unallocated blocks increase the cost of searching.

Sequential fit algorithms can be collectively used, like half-fit, optimal-fit, or worst-fit. For example, in worst-fit policy, it searches for the biggest unallocated memory block which is big enough to fulfill the demand of an application as it tries to create the unallocated block as large as it can to avoid small fragmentations.

In the worst-case, the algorithms allocate memory blocks in O(n) time complexity, where n is the heap's size. These algorithms are not feasible and not adequate for any real-time systems.

2.3.2 Segregated unallocated block Lists

Majority of the latest memory management allocators, like tcmalloc, Tow Level Segregated Fit, DougeLea malloc, Hoard etc. implement segregated unallocated block list mechanisms which use an array of unallocated memory block lists. Due to this, the searching time for the demanded block is reduced but it also creates little internal fragmentations. The allocator uses the size of the memory block to maintain and separate out the different size block. Each block size is a power of two apart that's why each class of size keeps unallocated memory blocks of a specific size.

The allocators round up the demanded block size to the nearest class of size. The memory allocator searches for an unallocated block of the demanded size, which is big enough to fulfill the demand of an application, in a specific sized class or of marginally lesser size which is bigger than any smaller sized class.

While in deallocating of memory block, the memory management algorithm pushes an unallocated memory block into the unallocated block list for the given size on release of the allocated memory block. Wilson, in [117], specifies that the allocators having segregated unallocated block lists can be categorized into two classes: 1) simple segregated storage 2) segregated fit.

2.3.2.1 Simple Segregated Storage

It is one of the simple memory management algorithms which use an array of unallocated memory block lists. In this, there is no requirement of splitting and merging of unallocated blocks. These features differentiate between simple segregated storage from buddy systems. The main benefit of this method is that no captions are needed. As caption exhibits overheads, this algorithm

reduces memory usage – the captions typically raise memory usage by some percentage [120] – which is most significant when the average demanded size is very less.

As there is no need of splitting or merging of blocks and keeping of captions, these allocators are faster; particularly when the memory blocks of a provided size are demanded frequently in a short time span. It allocates the demanded block in O(1) time complexity.

The issue with this allocator is that it makes large external fragmentation, which is relative to the highest amount of memory used by the allocator (Mem_used) times the maximum block size (Mem_max) demanded by the application. It also generates internal fragmentation.

2.3.2.2 Segregated Fit

These allocators use arrays of unallocated block lists. Every array keeps unallocated blocks within a specific size class. This memory management algorithm is faster than a single unallocated block list for the majority of the cases, as it finds the unallocated block list for the suitable sized class when the application demands specific memory. It, then, selects an array within a specific size class and finds an unallocated memory block from the array with a sequential fit mechanism. If the unallocated block is not available, the allocators try to discover for a bigger block in the adjoining array continuously until it finds a bigger block. The bigger block will be split into two parts, the requested and the remainder. The remainder part will be pushed onto a specific array. This memory management algorithm is classified as 1) Exact Lists 2) Strict Size Classes with Rounding and 3) Size Classes with Range Lists.

1) **Exact Lists**: In this mechanism, the memory management algorithm requires having lists with an enormous number of unallocated block and each list has all possible block size. The memory managers use the exact lists inside the classes of small size to decrease a huge number of unallocated block lists.

2) **Strict Size Classes with Rounding**: According to this mechanism, the memory management algorithm rounds up the demanded size to the nearest sizes in the class of size, though it wastes

some of the memory space as internal fragmentation. The most significant benefit of this allocator is that it can hold all memory blocks of the same size on a single size list.

3) **Size Classes with Range Lists**: This mechanism allows unallocated block lists to maintain blocks of slightly different sizes. This is an extensively used mechanism for memory allocations.

Summarizing all approaches, the segregated unallocated block lists can be used with other mechanisms like first-fit or best-fit to find a specific unallocated block list. If the memory manager discovers a specific size list, it examines for an unallocated block in the list according to these mechanisms. In the worst-case, this memory management algorithm accomplishes the task in O(1) time complexity but due to large external fragmentation, these algorithms are not appropriate for the real-time system.

2.3.3 Buddy Allocators

Buddy allocators are specific derivations of segregated unallocated block lists with the help of size class which rounds up to nearest value. In this allocator, the entire area of the heap is hypothetically distributed into two regions. These regions are, in turn, divided into two smaller areas, and so on so forth. This scenario is known as simple buddy situation. In worst-case, the standard algorithm of buddy allocator has O($\log_2 n$) time complexity, where n is the highest heap size.

The significant benefit of this allocator is that it holds all available memory blocks on a size list that are of precisely of the same size. For example, if a size of an unallocated block list is 2Kb, then each block available in the list will be of the size 2Kb. The only difference between this allocator and the other segregated lists allocators is that there is a limited splitting and merging of unallocated blocks with the help of a specific equation. For example, a power of two equation is as under:

$$i = \lfloor \log_2 range \rfloor \qquad 2.2$$

Here, the parameter ***range*** is used to calculate a specific index which signifies a specific unallocated block list within the size class to be used for either pushing an unallocated block or finding for an unallocated block.

For allocating a memory block in a simple binary buddy, the algorithm searches an unallocated block inside a specific array acquired by equation 2.2. If the unallocated block is not available, then the allocators attempt to discover a bigger memory block in the nearby array sequentially till searching a bigger block is found. If an unallocated block has been discovered in some upper range array, the unallocated block will be removed from the unallocated block list and recursively divided into the size of power of two logarithmically to be of lesser size. However, the size of the allocated block would be still big enough to fulfill the demand. The residual blocks which are created due to splitting are pushed into the corresponding unallocated block lists.

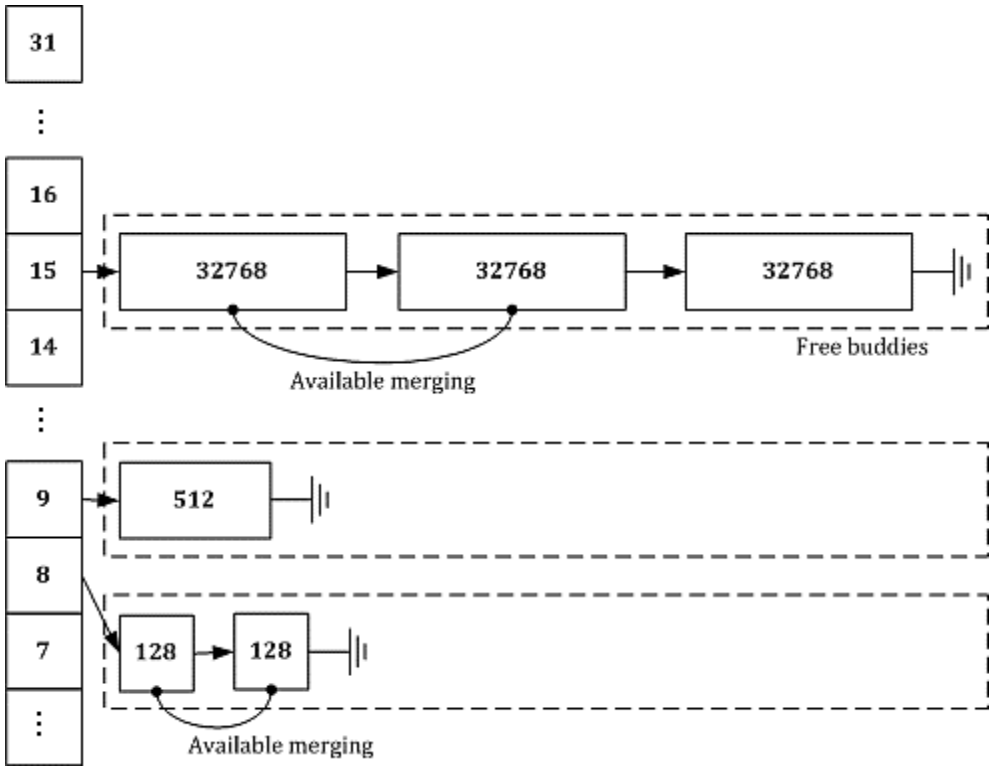

Figure 2.2: Structure of Buddy Allocator [100]

When an application releases an allocated memory block, it uses equation 2.2 to find the specific array inside size classes. The array found by the equation 2.2 is analyzed to check if it keeps neighboring memory blocks which are already unallocated and if it is so, then allocator

combines the released block with the neighboring memory blocks to make a new memory block of exactly dual the size. Subsequently, this procedure sequentially repeats until all unallocated block which encounters the criteria mentioned earlier. This simplest allocator is known as the binary buddy system. Wilson [117] categorized buddy systems into four categories as: 1) Binary Buddy 2) Fibonacci Buddy 3) Weighted Buddy and 4) Double Buddy.

1) **Binary buddy:** It is the simplest form of buddy systems. According to this allocator, all available memory blocks' size is a power of two, where every size is split into two identical fragments and combined into one dual size. Due to these features pointer manipulations become easy. For this, it has usually been considered as an allocator for a real-time system. In the worst case, this allocator performs the task in $O(\log_2\frac{m}{n})$ time complexity, where m is the highest heap size, and n is the highest allocated size of memory used by the application. The issue with these allocators is that internal fragmentation is comparatively very high around 25% [51].

2) **Fibonacci Buddy:** In 1997, Knuth [51] had proposed an allocator with the help of Fibonacci series numbers as the block sizes of buddy rather than a power of two, which lead to reduction in internal fragmentation related to binary buddies [39]. As all block sizes are a Fibonacci number, i.e., the addition of the two preceding numbers, one memory block can only be split if sizes are within the Fibonacci numbers as well. An issue with this allocator is that the residual block is probably unusable if the application allocates numerous unallocated blocks of the similar sizes [117].

3) **Weighted Buddy:** These memory management algorithms [83] [102] allow handling the classes of size in two ways. The size of all available memory blocks is 2^n and $3*2^n$ for all n. Hence, the size classes are comprised of the powers of two and three times a power of two among each pair of successive sizes, for example, 2, 3, 4, 6, 8, and so on. One significant benefit of this allocator is that, here, the mean internal fragmentation is lesser than the other existing buddy systems. On the contrary, a major issue with these allocators is that it creates large external fragmentation than other existing buddy systems [14].

4) **Double Buddy:** In 1986, Hagnis and Page [83] proposed allocator and named it double buddy systems. It is one of the implementations of weighted buddy allocator. These algorithms decrease the fragmentation related to weighted buddies. Here, the splitting criterion is the main difference between double buddy and weighted buddy. All unallocated memory blocks can be split in half, precisely creating all memory blocks in the size of power of two which is similar to the binary buddies.

In the worst-case, the time complexity of all algorithms discussed above are $O(\log_2 n)$ but because of large internal fragmentation [110], these algorithms are not appropriate for the real-time system.

2.3.4 Indexed Fit and Bitmapped Fit

In the sequential fit mechanism, for searching an unallocated block, linear search is required. It is also required in segregated fit for passing through an array inside a size class. An indexed fit mechanism is used to increase the speed of discovering an unallocated block of memory management algorithms. It can be used with other mechanisms to create a hybrid mechanism with different types of data structures. These mechanisms are used in detail indexing to maintain track of unallocated memory blocks inside size-based policies. Overall, all of these memory management allocators are the modifications of the indexed fit mechanism because the majority of the allocators maintain a record about which parts of the memory have been used or which are still unallocated [15] [24].

For example, the bitmapped fit mechanism is inherited from the indexed fit mechanism. This memory management allocator maintains a record about which portions of arrays are in use and which portion is free. Each bit is mapped to an unallocated memory block or array, based on their mechanisms. A bitmapped fit structure is used very less because, traditionally, it is very slow in finding an unallocated block. Now, with the latest processors, based on bit search instructions, it consumes limited clock cycles of the processor to find a block.

The most significant benefit of a bitmap scheme is that it can be developed with very less amount of memory. For instance, it can be one or two words in some of the application. If the development of a memory management algorithm needs a huge amount of memory, then there is a maximum possibility of interleaved data structure across the nodes in a Non-Uniform Memory Access architecture. To improve locality of searching on the node itself, a bitmap per node can be used.

Best examples of an indexed fit with bitmapped fit structures are Half-fit [80] and Two-Level Segregated Fit [59]. They use bitmaps to maintain to find which areas are unallocated. Fast fit [105] is also one of the best examples of an indexed fit mechanism which implements a Cartesian tree.

2.3.5 Hybrid Allocators

As stated earlier that a single memory management allocator has numerous drawbacks. Hence, the majority of the latest memory management allocators have combined various allocators to increase the speed of searching and inserting memory block [54] [55]. Majority of the algorithms used by latest allocators are a mixture of sequential fit, buddy system and segregated fit algorithms. In this section, some of the most extensively used hybrid memory management allocators in the general-purpose operating system as well as the real-time operating system.

2.3.5.1 Doug Lea(DLmalloc)

This algorithm was developed by Doug Lea in 1996,s extended by Gloger in 2006 [23] [34] and third time modified by the Free Software Foundation in 2012 [23] [28] [29]. In these all extended version of the memory management algorithms, the most important strategy and policy persist as it is. However, in terms of improvements in the latest extension, they implemented bitmaps to discover available unallocated blocks instead of the sequential searching which was being used in its earlier version. Presently, it implements a mixture of various mechanisms, based on the demanded size. It also employs the good-fit policy together with the segregated size-classes mechanism. DLmalloc allocator is designed and developed to use two different data structure,

based on the memory blocks' size. Figure 2.3 [52] [100] [112] demonstrates the structure of DLmalloc.

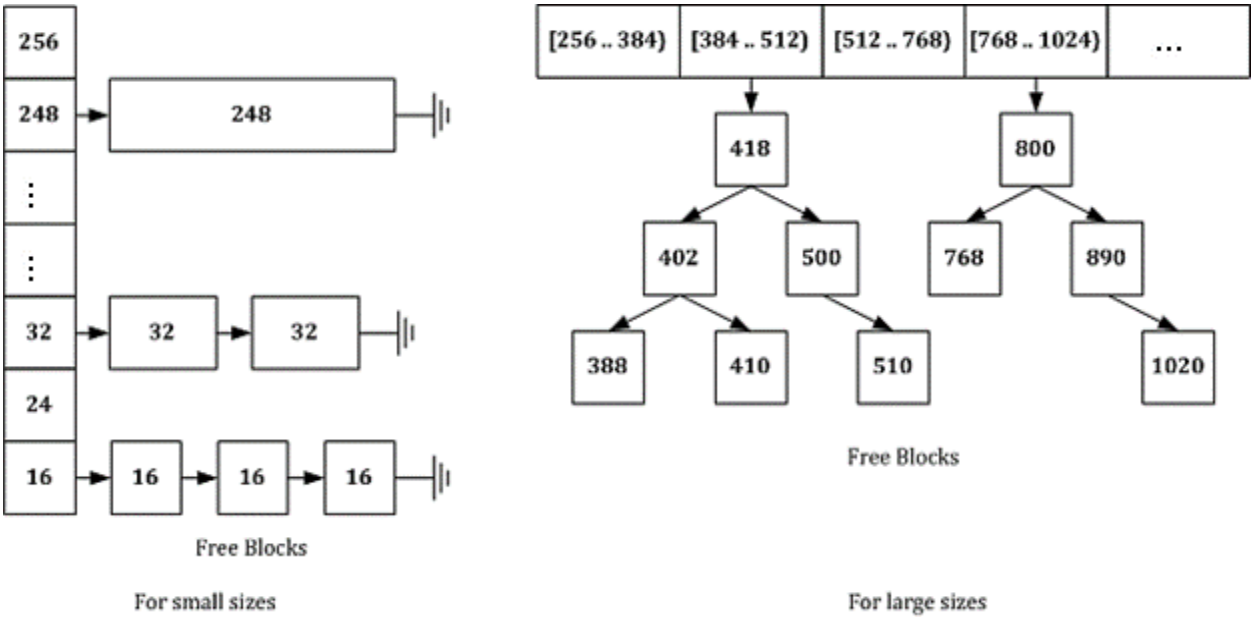

Figure 2.3: Structure of DLmalloc

To allocate a small memory block, this algorithm occupies a huge number of static size arrays known as small-bins. Bins occupy unallocated blocks which are having sizes not more than 256 bytes. Every bin comprised of 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 the best-fit policy or finds blocks which are large enough to satisfy the request.

If the size of the demanded memory block is larger than 256 bytes and lesser than some fixed value (normally 256KB), the algorithm tries to search existing blocks in an array known as tree-bin, which is having a tree structure for the memory blocks. As shown in figure 2.3, tree-bins consist of a collection of the bin. Nodes in the tree structure act as small-bin, comprising of the blocks of equal size. For any kind of demands of block size beyond the fixed value, the algorithm transfers the demands to the operating system through some specific system call.

In the worst-case, to find a small block whose size less than 256 bytes, this memory management allocator takes O(1) time and for a larger block whose size is greater than 256 bytes, it takes O(m) times, where m is the depth of the tree.

2.3.5.2 Half-Fit

This algorithm, designed and implemented by Ogasawara 1995 [80], uses bitmapped fit strategy and achieves execution time in a constant manner. As it uses bitmap policy for allocating a released block, it is slow. Here, the bitmap is used, only, to maintain the status of unoccupied lists. The time complexity of half-fit 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 to satisfy the request. Figure 2.4 [80] [100] [112] shows the structure of Half-fit.

This algorithm has a 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 the index i may be computed by this equation [79]:

$$i = \begin{cases} 0 & if\ r\ is\ 1 \\ \lfloor \log_2(r-1) \rfloor + 1 & otherwise \end{cases} \quad 2.3$$

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 block list indexed by i. If the 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 blocks list will be split into two distinct memory blocks of sizes r1 and r2 before allocation and the remaining memory block r2 will be put into the matching unallocated block list.

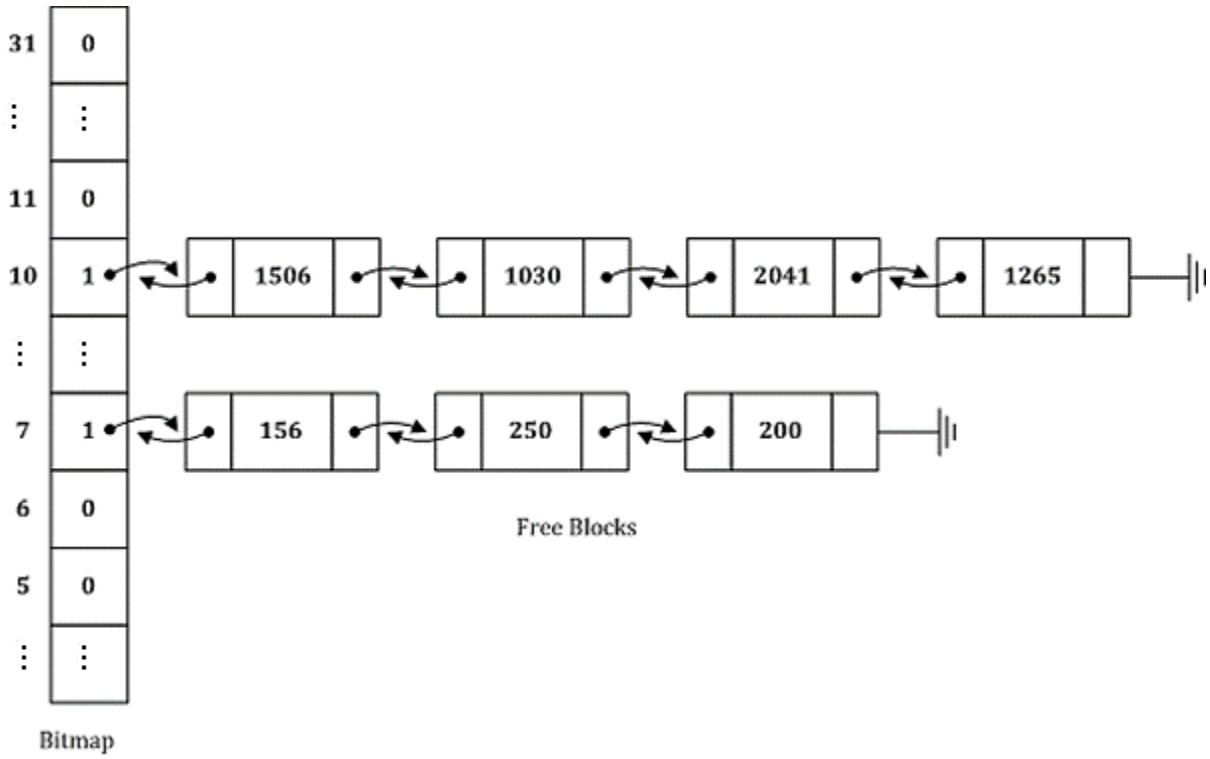

Figure 2.4: Structure for Half-Fit

For the deallocation process, the released memory block will be directly merged with neighboring memory block if it is unallocated. After an unallocated memory block is combined, the combined memory block's size is r, and this new memory block is pushed onto the top of the unallocated block list indexed by i. To find the value of i, the algorithm uses the following equation 2.4 [79]:

$$i = \lfloor \log_2 r \rfloor \qquad 2.4$$

As adjacent memory blocks are linked by doubly link list, merging with neighboring unallocated blocks can be done in O(1) time.

This algorithm can be used for real-time operating systems because of its constant time complexity. Furthermore, it demonstrates the best performance as compared to the binary buddy allocators. This allocator also provides the best response time in a worst-case scenario. But, it is not an ideal algorithm for the real-time operating system due to its internal fragmentation which is created due to its merging policy.

2.3.5.3 TLSF

TLSF (Two-Level Segregated Fit) [13] [58] [63] [67] is one of the best available dynamic memory allocation algorithms. This allocator is an enhancement of Half-fit allocator [60] [79]. This is the best allocator for the real-time operating system. Unlike a segregated list allocator, this algorithm has two levels of segregated lists of unallocated memory blocks; in which each list maintains the unallocated blocks of predefined size range to decrease internal fragmentation.

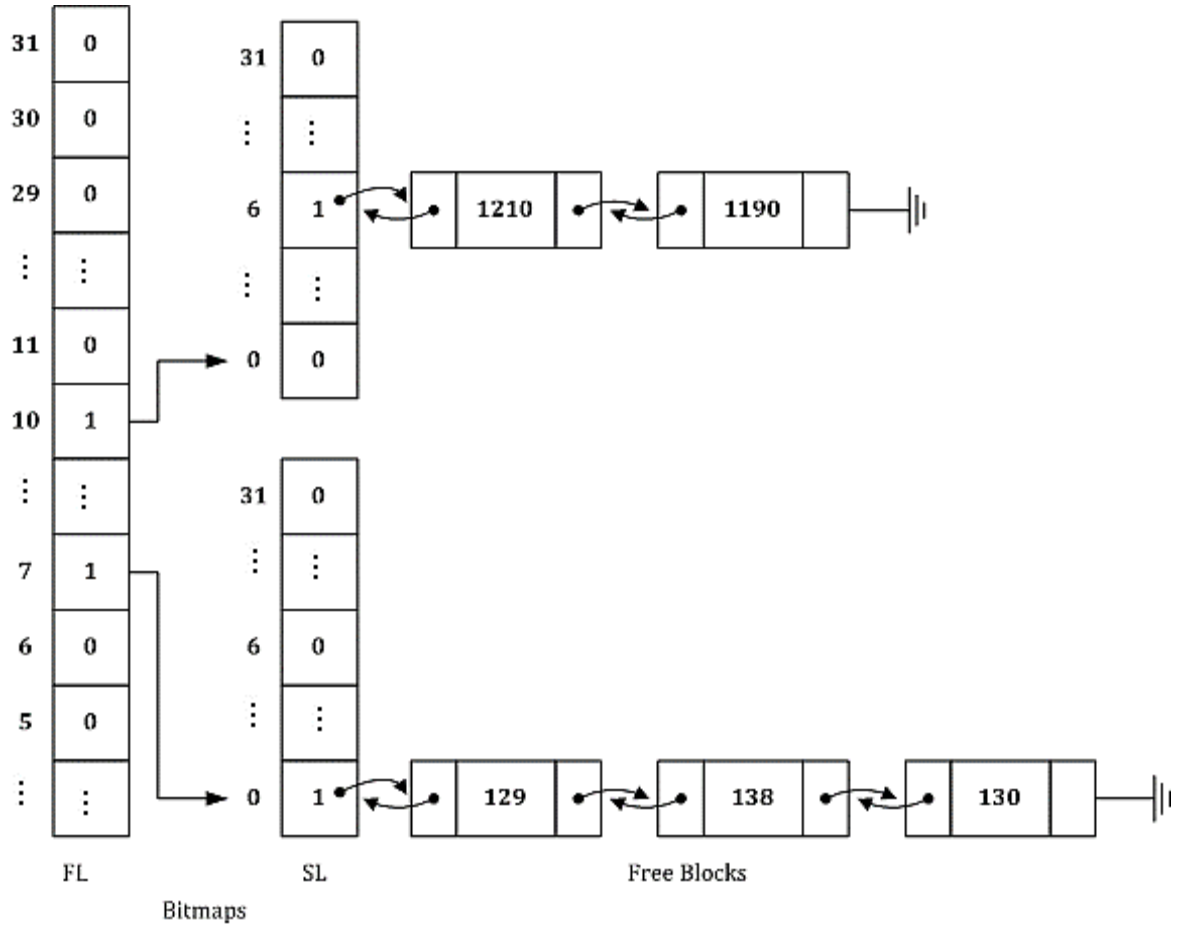

Figure 2.5: Structure of TLSF

The first-level of the list (FL) splits unallocated memory blocks into parts which are apart from each other by the power of two like 2, 4, 8, 16… and so on. The secondary array, known as second-level lists, splits each of the first level lists by a user-defined variable known as Second Level Index. TLSF structure is shown in Figure 2.5 [58] [100].

The primary goal of TLSF is to deliver restricted response time in both the procedures of memory allocation and deallocation irrespective of the size of memory. For allocating a memory block, TLSF uses equations 2.5 [68] [69] [118], where the specified size of a block computes the indexes of the two arrays pointing to the related segregated list.

$$\mathrm{i}(f,s) = \begin{cases} f = \lfloor \log_2(r + 2^{\lfloor \log_2(r) \rfloor - \mathrm{SLI}} - 1) \rfloor \\ s = \left\lfloor \dfrac{(r + 2^{\lfloor \log_2(r) \rfloor - \mathrm{SLI}} - 1 - 2^{i})}{2^{f-SLI}} \right\rfloor \end{cases} \quad 2.5$$

First-level of list (FL) f can be computed as the location of the previous bit set of the size, where the bit is set to 1. This index specifies blocks of memory according to their size. Each FL location indicates to a specific size class. For example, FL3 specifies size class width from 8 bytes to 16 bytes; FL4 specifies size class width from 16 bytes to 32 bytes. The second-level of the list can be calculated by the same equation. Each position indicates a memory block within similar sizes. These equations can be proficiently employed by the latest processors' instructions set only.

If the provided size of a memory demand is r, the index i (f, s), which is used to take the top of the unallocated block list keeping the nearest class list, is computed by the above-mentioned equation 2.5. If an unallocated block is discovered from the unallocated block list indexed by i, only that particular block will be fetched from the top of the unallocated block list and returned. If searching of the unallocated memory block at index i fail, then the subsequent unallocated block list whose index is nearby i will be inspected. Then, an unallocated memory block from the nearest unallocated block list of bigger sizes will be split into two memory blocks of sizes r and r′. The residual memory block of size r′ is pushed onto the related unallocated block list.

For the deallocation of a memory block, released memory blocks are directly merged with the neighboring unallocated memory blocks. If the neighboring blocks are unallocated, the neighboring blocks will be fetched from the segregated list and combined with the present memory block. The new block will be pushed onto the top of the unallocated block list indexed by j. For the calculation of j, TLSF uses the following equations 2.6 [68]:

$$\mathrm{j}\,(f,s) = \begin{cases} f = \lfloor \log_2 r \rfloor \\ s = \left\lfloor \dfrac{(r - 2^f)}{2^{f-SL}} \right\rfloor \end{cases} \qquad 2.6$$

As TLSF explores the demanded memory block's size to compute the suitable location indexed by the FL and SL depending on equations 2.5 and 2.6, it can be accomplished in O(1) time complexity.

Here, TLSF uses a shared heap which creates heap conflict in multi-threaded environments and it is not scalable also. Although, the improved version of TLSF does not have a constant execution time, its policy can still be accomplished in a constant time. In the modern type of TLSF [65], multiple memory pools are used along with an extra pool of memory which will be generated when available pools of memory are occupied by system calls.

The issue which may arise here is that if the last allocated memory block in each pool of memory requires to be free and the memory manager has already used the multiple pools of memory where the later releases the last allocated block, then the pool of memory that has the last allocated memory block will be put in the unallocated state with TLSF trying to combine neighboring memory pools while deallocating blocks of memory. Under this policy, the time of the merging process can rise linearly; therefore if there are N unallocated memory pools, it traverses N times to combine each other sequentially. Hence, under this policy, TLSF achieves O(n) time complexity [66].

2.3.5.4 tcmalloc

This algorithm is developed and implemented by Sanjay Ghemawat in 2010 [99]. It is an algorithm which associates both global heap structure and thread's private heap multiprocessor architecture. To allocate small memory block which ranges from 4 bytes to 32 Kb size, 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 is required to offer mutual exclusion. For this, it employs a spin-lock mechanism. If any of the 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 2.6 [99] [100] [112] shows the structure of tcmalloc allocator. The time complexity of this allocator is O(1).

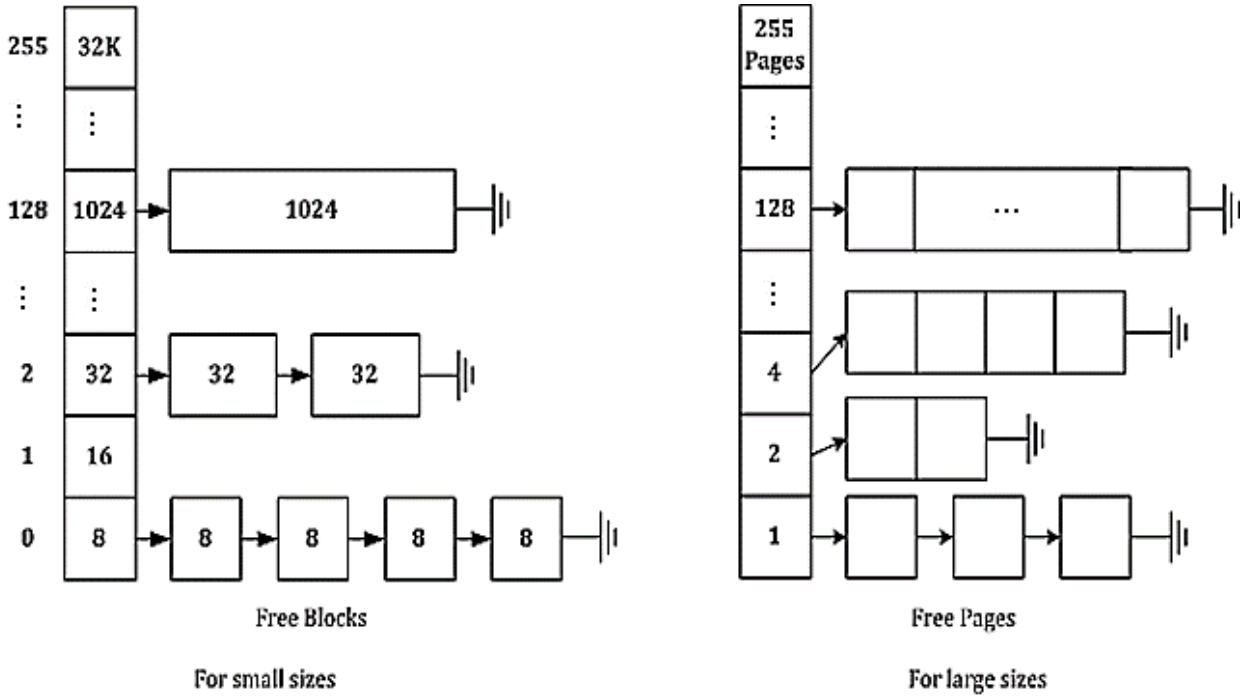

Figure 2.6: Structure of tcmalloc

If the available size of a demanded memory block is less than 32Kb, the memory manager examines whether it can fulfill the demand from the thread's private heap. If this fails, tcmalloc examines the global heap with the help of a lock to achieve synchronization. If the unallocated block list is empty, the subsequent unallocated block list will be inspected, and so on. If the global heap has an unallocated memory block of adequately big size, it will be split into two memory blocks, one block is remainder and another one is served. The remaining unallocated memory block will be pushed onto one of the unallocated block lists in the global heap.

In the deallocation of the memory blocks, if any thread releases a small memory block, then it will be pushed onto the list in the private thread heap of the related size. If the size of an unallocated block list goes beyond a specific threshold, for example, 2MB, then the memory management algorithm moves some of the unallocated blocks back to the global heap.

The tcmalloc uses the demanded size of the memory block to compute the suitable index in the global heap or private thread heap. So, this operation can be accomplished in O(1) time complexity. The main disadvantage of this allocator is that it has boundless memory consumption and it does not address to false sharing [113].

2.3.5.5 Hoard

This algorithm has been designed by Bergar in 2000 [6] and it is popular due to its speed and scalability in the multiprocessor environment.

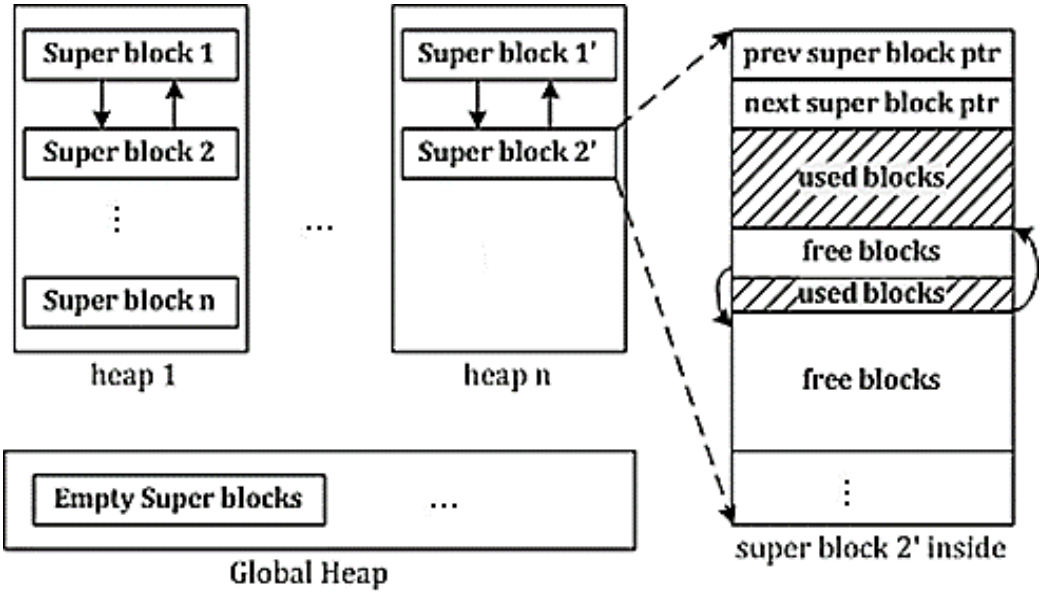

Figure 2.7: Structure of Hoard

This allocator also uses a segregated class mechanism, but unlike tcmalloc, it maintains a private heap for each processor and also maintains a global heap to avoid heap conflicts. Other than private processor heap and global heap, it also maintains one private heap per thread to allocate smaller size memory blocks whose size is less than 256 bytes. Threads which are executing

on the same processor can also share private processor heap. Figure 2.7 [6] [100] [112] shows the structure of this algorithm.

This algorithm, to allocate any blocks, whose size is less than 256 bytes, it first searches it into a heap of the thread and if it fails, then it searches into private processor heap. In a private processor heap, the memory manager allocates memory blocks from the system in a large piece, called superblock, which is an array of some sets of memory blocks comprised of unallocated block list. If private processor heap is completely searched, then it searches into a global heap. In such scenario, the thread locks the private processor heap which will be unlocked when the memory manager moves a new superblock to the private processor heap from the global heap.

This allocator has the time complexity of O(n) for allocation of a memory block as it requires to discover consecutively for the thread's superblock. Here n is superblock which is the number of chunks of the memory block. As mentioned earlier, there can be heap conflicts between different threads of the shared private processor heap; this allocator addresses it with double private processor heaps corresponding to a number of processors in the system. This allocator also uses a specific mapping function between processors and threads. However, the cache misses arises, here, due to the disruption of node oriented data locality.

2.3.5.6 Smart Memory Allocator Algorithm

This algorithm has been proposed by Ramakrishna M, Jisung Kim, Woohyong Lee and Youngki Chung in 2008 [26] [119]. 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 the other is long-lived memory blocks. The short-lived memory blocks are allocated in the direction of lowest level to highest level from heap whereas the long-lived memory blocks are allocated from highest level to lowest level [10] [12].

The used space of heap grows from highest level to lowest level as well as lowest to highest also. Initially, the entire heap memory is unallocated, and there will be one unallocated memory block for both the categories, short and long-lived memory [2] [4]. The heap space is divided

equally into two blocks. When the heap grows from both the sides, the virtual border between these two can easily be modified according to the dynamic memory request.

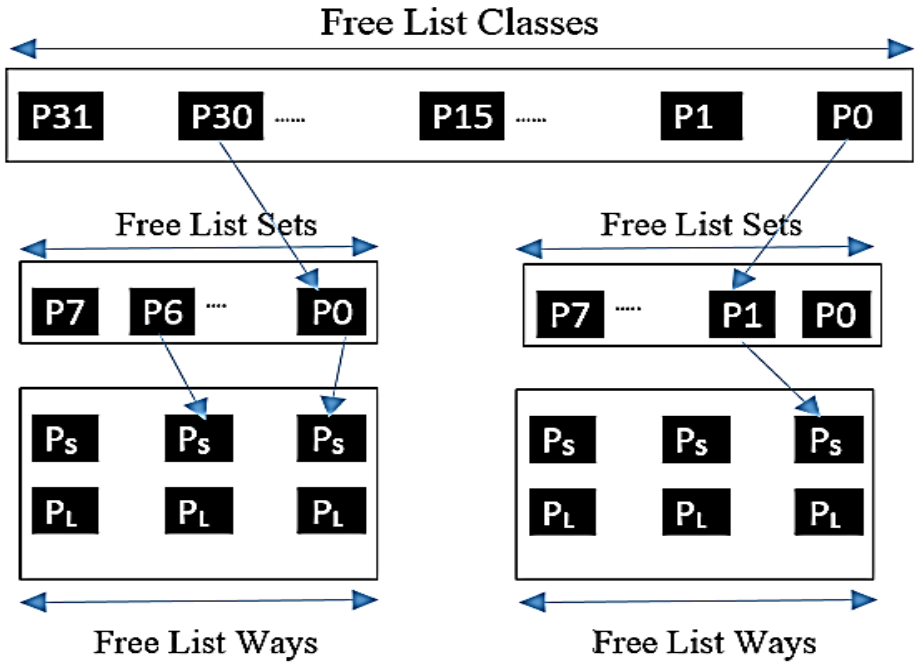

Figure 2.8: Structure of Smart Memory Allocator [114] [119]

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

2.4 Summary

Dynamic memory allocation algorithms are important for the applications of this computing era. These, all allocators use all available memory resources more efficiently. The recent general-purpose operating systems offer the dynamic memory allocation, but since they are not enhanced, they may lead to issues like costly searching of memory block and fragmentation.

In last five decades, many memory management algorithms have been proposed. Each algorithm has its own pros and cons, but they try to decrease fragmentations or lower the execution time.

Table 2.1 shows the worst-case time complexity of the allocators as well as the fragmentation that occurs is acceptable or not along with the provision to support the NUMA architecture. The parameters in the table are: n is the heap size and m is the tree's depth. In context of real-time systems, the segregated Fit, tcmalloc and Half-fit are the only algorithms which satisfactorily meet the desired objectives and also provide a constant execution time.

Table 2.1: Summary of existing Memory Allocators of RTOS

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($\log_n 2$)	Unacceptable	No

Furthermore, the buddy allocators having O($\log_2$ n) time complexity is affordable for real-time systems, but they cause large fragmentation. The other allocators are not acceptable for the real-time systems because of their boundless execution time. Moreover, only tcmalloc allocator provides support to NUMA architecture system.