UNLV | University Libraries
University of Nevada, Las Vegas

UNLV Theses, Dissertations, Professional Papers, and Capstones

8-1-2012

Non-Blocking Concurrent Operations on Heap

Mahesh Acharya
University of Nevada, Las Vegas, maheshmhs@yahoo.com

Follow this and additional works at: https://digitalscholarship.unlv.edu/thesesdissertations

 Part of the Computer Engineering Commons, and the Computer Sciences Commons

Repository Citation

Acharya, Mahesh, "Non-Blocking Concurrent Operations on Heap" (2012). *UNLV Theses, Dissertations, Professional Papers, and Capstones*. 1651.
https://digitalscholarship.unlv.edu/thesesdissertations/1651

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Theses, Dissertations, Professional Papers, and Capstones by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.

NON-BLOCKING CONCURRENT OPERATIONS ON HEAP

by

Mahesh Acharya

Bachelor of Technology in Computer Science Engineering,
Uttar Pradesh Technical University, India
2008

A thesis submitted in partial fulfillment of
the requirements for the

Master of Science Degree in Computer Science

School of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
August 2012

© Mahesh Acharya, 2012
All Rights Reserved

THE GRADUATE COLLEGE

We recommend the thesis prepared under our supervision by

Mahesh Acharya

entitled

Non-Blocking Concurrent Operations on Heap

be accepted in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
School of Computer Science

Ajoy K. Datta, Committee Chair

Lawrence L. Larmore, Committee Member

Juyeon Jo, Committee Member

Venkatesan Muthukumar, Graduate College Representative

Ronald Smith, Ph. D., Vice President for Research and Graduate Studies
and Dean of the Graduate College

August 2012

Abstract

We present a non-blocking implementation for the concurrent Heap data structure in the asynchronous shared memory system. The system may be a traditional one with a single processor with multiple threads, or it may be the one with multiple processors each possibly with multiple threads. Our implementation supports *readMin*, *deleteMin*, and *insert* operations on the heap. The deleteMin and insert operations are non-blocking operations, whereas the readMin is wait-free. One easy approach of using a heap in multi-threading environment could be by locking the entire heap before performing any operation. This would make the implementation very slow to have any practical application.

The above inefficiency could be reduced by using the fine-grained locking mechanism as shown by Herlihy and Shavit[16]. But, this would still be a blocking implementation, where some threads may be waiting forever for some other threads holding the locks that might have crashed or failed.

Israeli and Rappoport[20] gave a non-blocking implementation which supports deleteMin and insert operations using some primitives which they claim could be built using transactional memory. Moreover, their implementation has a limitation on the range of values a node in the heap can have. We propose a different implementation which supports all three operations using a primitive, called Double-Compare-And-Set(DCAS). DCAS is a synchronization operation supported in hardware by Motorola 68K family of processors [19]. It was also going to be supported by Sun's canceled Rock processor[4]. DCAS takes two memory locations as input, and writes new values to them only if they match input expected values. Our algorithm has no limitation on the range of numbers a node can have.

Acknowledgements

I would like to express my deepest sincere gratitude to my adviser Dr. Ajoy K. Datta for giving me a unique opportunity to work on such an important topic. His continuous guidance, invaluable suggestions, affectionate encouragement and generous help are greatly acknowledged. His keen interest on the topic and enthusiastic support on my effort was a source of inspiration to carry out the study. I also consider myself fortunate to work as his TA for past two years.

I would also like to thank Dr. Ju-Yeon Jo, Dr. Lawrence L. Larmore and Dr. Venkatesan Muthukumar for their time in reviewing my report and their willingness to serve on my committee.

I also thank my friends and family for their unconditional support.

Mahesh Acharya

University of Nevada, Las Vegas
August 2012

Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the use of multi-core processors being ubiquitous, concurrent data structures have been getting the interest of many researchers. These data structures allow many threads to access the same data at the same time. This calls for the proper management of the contention between the threads so that the operations are correct and the data structure stays valid. Traditionally, these concurrency related issues used to be handled by using locks but use of locks has many issues as explained in section 2.2.1. To resolve these issues various approaches have been made for different data structures like linked lists, queues, hash tables etc.

Heap is one of such fundamental data structure which is also a priority queue. The highest priority element always stays at the root of the heap. If we assign higher priority to smaller element then the heap formed is called *min-heap* but if we assign higher priority to larger element then the heap formed is called *max-heap*. Through out this thesis when we say heap we are referring to the former one called min-heap unless otherwise stated. Changing our min-heap implementation to max-heap implementation shouldn't be any difficult. And a stable heap has the invariant that the parent has no lower priority than any of its children. The heap supports three operations insert,delete and read maximum priority element. Insert and delete operations insert and delete an item respectively in the heap where as read returns the element with maximum priority.

Each of these operations are in fact a sequence of many instructions. This may make the heap inconsistent during insertion or deletion. During insertion, a node that is bubbling up may have its priority lower than that of its parent where as a node bubbling down due to deletion may have higher priority than its children. Such situations may arise due to simultaneous non-blocking operations in the heap.

Any node that violates the order invariant can create confusions to subsequent insertions and deletions and terminate them prematurely leaving the heap in inconsistent state. So any imple-

mentation of concurrent operations on heap should consider these situations properly to make them work correctly.

1.1 Contribution

In this thesis, we present an implementation of operations for concurrent heap structure on asynchronous shared memory system. Our implementation offers all readMin, deleteMin and insert operations on the heap structure. The readMin operation is wait free and returns the minimum element from the heap, deleteMin operation deletes the minimum element from the heap and returns it where as insert operation inserts an item into the heap at proper location. Both deleteMin and insert operations are non-blocking operations.

To make these operations non-blocking we used the approach of helping other threads if they try to block a thread. To increase the performance, we help other thread only if that thread is preventing us from progress. A thread which might have gone sleeping might have been helped accomplish the task or might have been progressed further. Our implementation also makes the woken up thread to proceed from the correct new position rather than having to go through complex operations of figuring out the current state which is the case in the implementation by Israeli and Rappoport[20]. We have presented the implementation in a detailed pseudo-code in order to enable anyone with hardware support for DCAS to write a code in a programming language.

1.2 Outline

In Chapter 2 we give an overview of shared memory system. Our focus will be mainly on multi-threaded system and multi-core systems. We will briefly explain why more and more systems are going towards multi-core systems. We will also include different approaches for synchronization among threads. Lock Based synchronization approach and its limitations are also explained in the chapter along with other lock free systems. Lock free systems are explained in terms of obstruction free, wait free and non-blocking implementations.

While we know of many different data structures for sequential operations, we need to use them with caution for concurrent operations. In Chapter 3 we will briefly illuminate various approaches people have come forward with for implementing data structures for asynchronous shared memory systems. In this chapter we will also highlight a very popular transformation called universal transformation which can be used to transform a sequential algorithm to a non-blocking concurrent algorithm. We will also include works done by various researchers in some of the popular data structures like linked lists, binary search trees and heap.

In Chapter 4 we will discuss how we represent our data. Heap is constructed from arrays. One array holds the data while another array holds the information about the nodes of the heap. Each thread also have some variables to store information. Different bits of the information word are allocated to represent different things like the node id, current operation and the thread id. This chapter also has the listing of different global values we use for our algorithm. It also contains the details about CAS and DCAS the primitive hardware operations which are used in our implementation.

Chapter 5 consists of implementations of various procedures which we call subsidiary operations. These procedures are created to implement the main procedures in chapter 6. Chapter 6 explains the details of readMin, deleteMin and insert operations in the heap. Finally we conclude the thesis with chapter 7 which contains both the conclusions and recommendations for future work.

Chapter 2

Background

Unlike in old days, computers now generally consists of multiple processors. On the basis of number of processors we can classify the computer system into uni-processor or multi-processor system. Again, computers can be constructed out of several separate computation nodes, where each node (which could be a computer system in its own) is physically separated from others. These type of computer systems are called distributed systems[1].

These computer systems need a mechanism to communicate between one another. It could be done either by shared memory system or message passing system. Message passing system is the one where communication is done by exchange of information packages via some form of network. Details of message passing system are beyond the scope of this thesis while shared memory system is explained in the section 2.1.

2.1 Shared Memory System

Shared memory system is one in which processes(or threads) or the processors communicate via a shared memory. Figure 2.1 shows a shared memory system where three processors P_1, P_2 and P_3 access the same memory. Again, it it important to note that the shared memory could be centrally

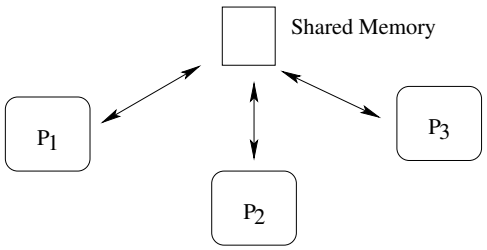

Figure 2.1: A shared memory system with 3 processors P_1, P_2 and P_3

located or distributed into different computation nodes and connected via some form of network.

On the basis of how the shared memory is distributed, we have two different kind of architectures: UMA(uniform memory access) and NUMA(non-uniform memory access). In UMA, the shared memory appears to be at equal distance from each processor and hence have the same response time. However in NUMA architecture, a shared memory might appear to be closer to one processor while another shared memory might appear closer to another processor and hence they may not have the same response time.

In order to exploit concurrency concurrency in a computational workload, multi-threaded system or multi-core system or both can come into help.

2.1.1 Multi Threaded System

When a program needs memory access to proceed further, that access must reference RAM whose cycle time is tens of times slower than that of a processor, then a single threaded processor can do nothing but wait until the required data is received. Thus, a lot of times single threaded processors spend a huge amount of time doing nothing.

If it is such latencies that is preventing a CPU pipeline busy, then a single pipeline should be able to complete more than one concurrent task in less time then it would take to run the tasks serially. This leads us to run more than one thread at a time. Such execution requires more than one program counter and more than one set of programmable registers. But replicating these resources is still less than replicating an entire processor. Kevin[22] a Principal Architect at MIPS Technologies, mentions that in the MIPS32 34K processor, which implements the multi-threading architecture, an increase in area of 14% can buy an increase of throughput of 60% relative to a comparable single-threaded core. However, multi-threading a single processor can only improve performance up to the level where the execution units are saturated.

Intel terms this technology of supporting multi-threading by duplicating sections of processor *hyper-threading*[7]. Intel states[25] that hyper-threading makes a single physical processor appear as two logical processors where the physical execution resources are shared and the architecture state is duplicated for the two logical processors. This enables the operating system or the user program to schedule threads as they would in multiple physical processors.

Multi-threading isn't limited to hyper-threading. Even without addition of processor components multi-threading is still relevant. An operating system can implement time-division multiplexing to make the processor switch between threads.

2.1.2 Multi Core System

A multi-core processor[9] is a single computing component with two or more independent actual processors called cores. Multiple cores can run multiple instructions at the same time increasing overall speed for programs that support parallel computing. To further elaborate why multi-core processing is so important we need to see why it was really invented.

The Need of Multi-core Processors

In 1965 Gorden Moore first drew a line representing the number of components per integrated circuits for minimum cost per component developed between 1959 and 1964. He extrapolated the trend to 1975 to show that the number of components per chip would double per year[27]. In 1975, it came to the realization that the doubling would happen every two year rather than every year. The rule finally settled with the prediction that the doubling would happen every 18 months although Moore still doesn't agree with the change[8, 2]. As the transistor size decreased the clock speed increased[23]. Higher frequency meant faster computers.

With transistor density increasing, the heat was now becoming a major problem. It was getting very difficult to increase the clock speed without having to deal with the heat. But we are limited by the following equations[24].

$$Power = Capacitance \times Voltage^2 \times Frequency$$

$$Frequency \propto Voltage$$

so we have,

$$Power \propto Voltage^3$$

So 10% reduction in voltage would bring 10% reduction in frequency, 30% reduction in power and less than 10% reduction in performance. Goeff Lowney[24],Intel Fellow gives us the following observation between single core to dual core.

Parameters	single core	dual core
Area	1	2
Voltage	1	0.85
Frequency	1	0.85
Power	1	1
Performance	1	1.8

Table 2.1: Comparision between single core and dual core chips.

The observation alone is sufficient argument to see why there is so much buzz about multi-cores for improving performance. So the industry is halting the clock rate improvements and is instead doubling the number of cores every 18 months[29]. The multi-core systems, distribute the work load to several processors. So there can be as many processes running simultaneously as the number of processors. But the gain in performance depends largely on how the algorithms are implemented. It is driven by the fact that the larger the fraction of the code that can be parallelized larger is the gain and it is what Amdahl's law [6, 17] confirms to.

2.1.3 Beyond Multi-Core System

Kevin[22] points out that no matter how many cores have been used its always easy to imagine adding another while only a limited class of problems can make practical use of them. Hyperthreading and quad-core processors now already leading the market, research on *many-core*[29] computing is well underway.

2.2 Synchronization of Processes

With multi-core architectures more and more processes and threads were being executed concurrently. It increased the overall speed of execution. However, there was a problem. The concurrent execution started sharing resources. And such sharing some times created *race condition*[30] and started behaving in unexpected manner. The asynchronous concurrent processes had to be synchronized when they were accessing the shared resource.

2.2.1 Lock Based System

Mutual exclusion has traditionally been a very safe approach for achieving synchronization among processes that share certain resources. Use of lock is one well known approach of achieving the mutual exclusion. However this simple solution is not quite a good solution when it comes to multi-core architecture. Some of the prominent drawbacks of using lock based systems are as follows:

- Deadlock: Say we have two processes A and B running concurrently. Both of them use resources r_1 and r_2. A locks r_1 while B locks r_2. A now waits for B to release its lock on r_2 while B waits for A to release lock on r_1 before each releasing the lock they hold themselves. They will end up waiting for ever. Deadlock has occurred.

- Priority Inversion: Say we have three processors A, B and C with priorities low, medium and high respectively. A and C share a resource and synchronize by mutual exclusion. If A holds

the shared resource while B preempts it C is made to wait for B although B has actually lower priority then C. This may not be what the scheduler actually wanted. It was the "Priority Inversion" which was frequently resetting "The Pathfinder" in Mars in 1997[21].

- Delays: A process which is enjoying the shared resource may take a long time to complete. It will cause other processes waiting for the resource to delay. The waiting processes may end up waiting for ever if the one holding the resource dies.

Using mutual exclusion makes the algorithm sequential in nature. It can't be worst than having multiple cores at hand and having to perform tasks sequentially.

2.2.2 Lock Free Systems

Lock free implementations don't rely on mutual exclusion so they are free from the above mentioned limitations of mutual exclusion. Some properties associated with progress conditions[28] in failure prone asynchronous systems are explained.

Obstruction Free Implementations

In an obstruction free implementation a process completes its operation if allowed to execute alone for sufficiently long time. This property is useful in those systems where conflict is rare.

Wait Free Implementations

In wait free implementations, irrespective of how many other processes are active or what their state is, any process that has invoked an operation eventually completes unless it fails itself.

Non-Blocking Implementations

Non-Blocking Implementation comes in between above two properties. It is stronger than obstruction freedom but weaker than wait freedom. It guarantees that at least one process will make progress after finite number steps. Some of the advantages(see here.[13]) of this synchronization are

- allows synchronized code to run anywhere(including interrupt handlers)
- is deadlock free
- aids fault tolerance
- eliminates interference between synchronization and scheduling

- can reduce interrupt latency
- aid portability
- increase total system throughput

Still non-blocking implementation is not without limitations. Some of the limitations are (see here. [13])

- Conceptual Complexity
- Performance: Non-blocking algorithms must guarantee that an object is always in a state where progress can be made even if the current process dies.
- Unfamiliarity

Chapter 3

Literature

With big computer organization pushing towards multi-core and multi processing architectures, it is no surprising that research is being done in concurrent data structure. Some of the common data structures that have gained much interest are doubly linked list, B-trees and AVL trees. Doubly linked lists are commonly used for other data structures like stack, queues, hash tables and skip lists. B-trees come into practice where indexing of large data is common. AVL trees on the other hand are used where performance on search operations is sought.

Designing the concurrent counterparts of these algorithms is more challenging because of the nature that asynchronous threads work upon the shared memory and it is very essential to ensure the thread safety. At the same time it is also highly desirable to support progress of as many threads as possible and reduce the interference among them.

Hillel[18] points out three ways of dealing with the problem. First and probably the worst is the use of lock whose appalling drawbacks we have seen before. However, with fine grained locking mechanism, it may not be as sinister as it looks. Second is the lock free implementations which we will explore in detail in this thesis and the third one is the use of transactional memory(TM).

Transactional memory is an approach extended from the popular approach used in database transactions. Transaction works on the principle that either all of the operations within a transaction complete or non of them. High level transactional operations on data items are translated into low level primitive operations on memory locations by software implementation of transactional memory. But Hillel[18] confirms that coarse grained implementations of TM aren't scalable enough where as fine grained implementations require non trivial amount of extra work.

Improvement in TM could be achieved by implementation with disjoint access parallelism (transactions on disconnected data do not interfere) and invisible reads(reads that do not write to the memory) which are wait free. But Hillel[18] has further shown that no TM implementation can be

disjoint access parallel and have invisible, wait free read only transactions.

3.1 Universal Transformations

Herlihy[14] has a translation protocol that takes as input the sequential algorithm and produces equivalent non-blocking concurrent algorithm. His first transformation does the copying of the entire object, making necessary changes to it and trying to replace the old object by CAS operation. CAS is explained later in section 4.2. Figures 3.1 through 3.3 elaborate on how only one succeeds at a time.

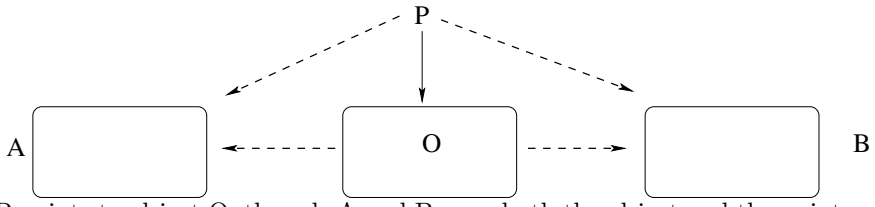

Figure 3.1: P points to object O, threads A and B copy both the object and the pointer. They make changes to their own copy of O

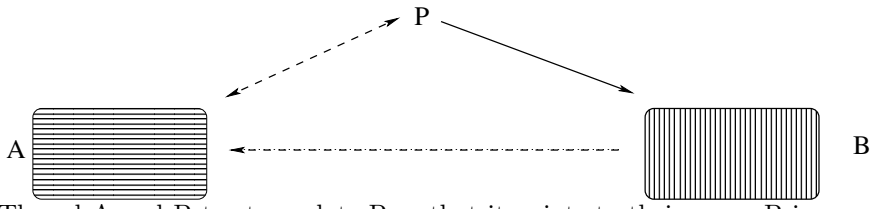

Figure 3.2: Thread A and B try to update P so that it points to their copy. B is successful but A fails. A then copies the pointer and the new object and makes modification to this copy.

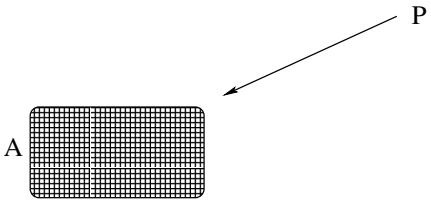

Figure 3.3: Thread A is now successful in making the pointer point to its copy.

For large objects, copying the whole object made this an impractical solution. He then proposed another transformation protocol. Each data structure is made up of blocks connected by pointers. Only the blocks which are modified or contain the pointers to the blocks which are to be modified need to be copied. Again it has some drawbacks, first there was still a lot of copying, second programmer had to go extra mile to break the structure into proper blocks and third for some data structure like priority queue implemented as linked list no decomposition performs well.

3.2 Linked Lists

A linked list is a dynamic data structure where a node has unique neighbors. Linked list can be singly linked list where each node contains reference to the next node or a doubly linked list where the node also has the reference to the previous node. Insertion and deletion can take place at any position in the list. However, these insertions and deletions in concurrent environment can make the list incorrect.

When we try to delete a node in the middle of a list we update the next pointer of the previous node to point to the next node of the node to be deleted. During insertion, we make the new node point to the next node and make the previous node point to the new node.

If we run concurrent insertions and deletions as in Figure 3.4 where a thread say T_1 tries to delete node B while another thread say T_2 tries to insert node C. So T_1 makes node A point to node D while T_2 makes node B point to node C. After both operations are complete we will have node A point to node D thus loosing the newly inserted node C. In Figure 3.5, thread T_1 tries to delete node B while thread T_2 tries to delete node C. When T_1 makes node A point to node C, T_2 makes node B point to node D. Finally, we will have a linked list where node C is still not deleted.

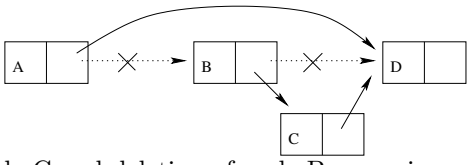

Figure 3.4: Insertion of node C and deletion of node B occurring concurrently in a linked list

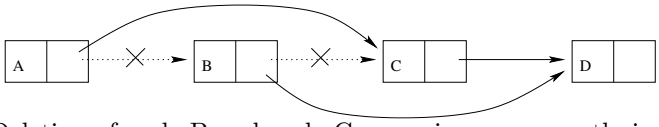

Figure 3.5: Deletion of node B and node C occurring concurrently in a linked list

To handle such situation Herlihy [15] proposed an general approach. Reference counter is placed for each pointer to the elements of the list. Deletion is allowed only when the count reaches to zero. To overcome the overhead of maintaining the reference counts during a visit, Massalin and Pu[26] restricted the delete operation to safe situation. And they defined the delete operation to be safe if the deleted node's pointers continue to point to the nodes that eventually take it back to the main list where CAS detects change and retries again. So they mark the node for deletion. If the previous node is not marked, they sit on the previous node and delete it. For better performance, one can traverse from head to tail deleting all the marked nodes.

3.3 Binary Search Trees

A binary search tree(BST) implements an abstract dictionary and supports three operations *Find(k)*, *Insert(k)* and *Delete(k)* operations. A BST is said to be leaf oriented if every internal node has exactly two children and all keys currently in the dictionary are stored in the leaves of the tree. A leaf oriented BST has the property that for each internal node x, all descendants of the left child of x have keys that are strictly less than the key of x and all descendants of the right child of x have keys that are greater than or equal to the key of x. Insertion in a BST replaces a leaf by a sub-tree of three nodes while deletion deletes the leaf and its parent while making its sibling the child of former grandparent.

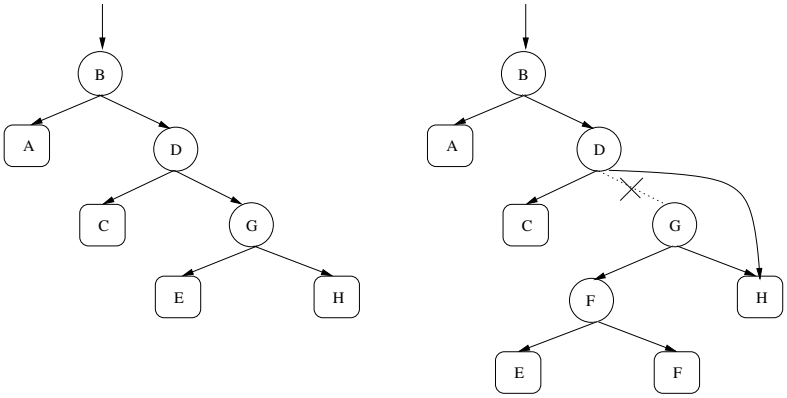

Figure 3.6: BST before and after concurrent insertion and deletion of F and E respectively

In Figure 3.6 some thread say T_1 performs deletion of E by making its sibling H as the child of its grand parent D. At the same time another thread say T_2 performs insertion of by replacing the leaf E with a sub-tree of size three. But the concurrent operations of T_1 and T_2 create the BST where the insertion of F is lost. Similarly in Figure 3.7 thread T_1 performs deletion of C by making G the

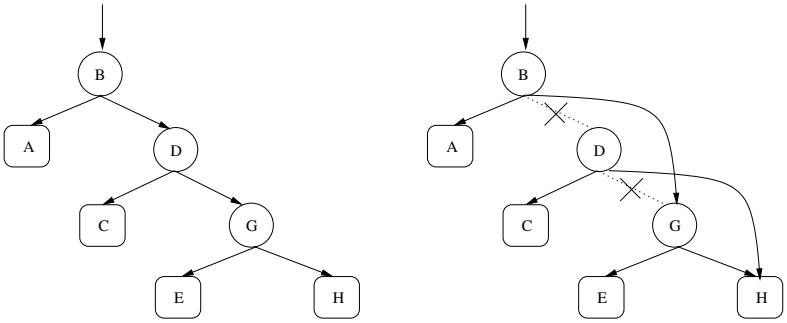

Figure 3.7: BST before and after concurrent deletion of C and E

child of B while thread T_2 performs another deletion by making H the child of D. These operations don't actually perform the deletion of E.

Although there have been several implementations of BST that use locks Ellen, Fatourou, Ruppert and Breugel [11] claim that their is the first complete implementation of a non-blocking BST in an asynchronous shared memory system using CAS. They stored a separate auxiliary field along with each key. When a leaf node is to be deleted its parent node is marked. Once a node is marked any child pointers can't change.

3.4 Heap

A heap is a complete binary tree in which each node has a priority not less than any nodes in the sub-tree with itself as the root of the sub-tree. It supports three operation insert(k), delete(), and read(). As explained in chapter 1 heap could be either min heap or max heap. However, we will concentrate only on min heap.

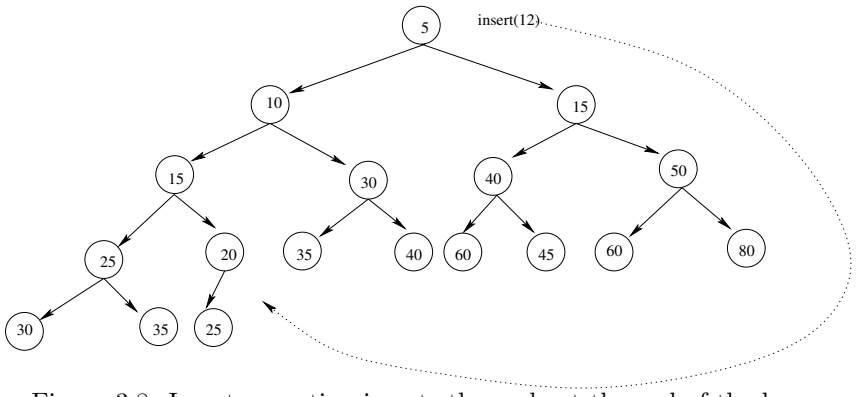

Figure 3.8: Insert operation inserts the node at the end of the heap

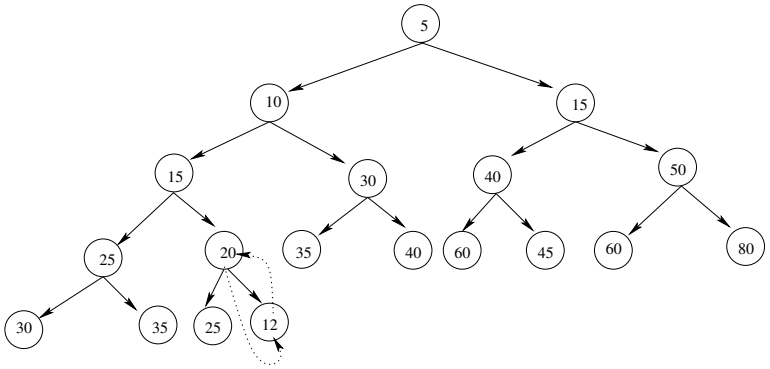

Figure 3.9: Bubble up of the newly inserted node in the heap

The insert(k) operation inserts an item with value k into the heap. For convenience we will assume the priority to be equal to the value. In figure 3.8, insert(12) operation is invoked and the item is inserted at the end of the heap as shown in figure 3.9. Since 12 is smaller than 20, the node

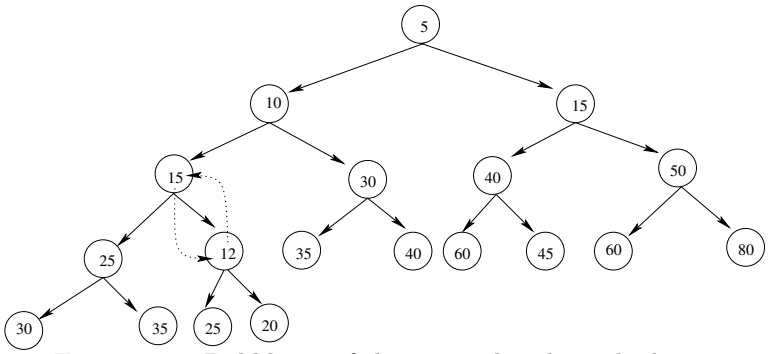

Figure 3.10: Bubble up of the inserted node in the heap

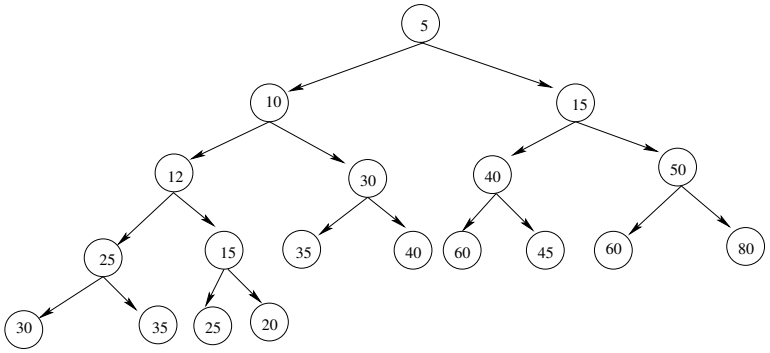

Figure 3.11: Final heap after insert operation completes.

with value 12 starts to bubble up until it is not smaller than its parent. Figure 3.10 and Figure 3.11 show the bubble up of the node with value 12.

The delete() operation deletes the element at the root of the heap, which is in-fact the one with smallest element in the heap. In the figure 3.12 delete operation deletes the element at the root of the heap. The last element from the heap is then brought to the root as in figure 3.12. At this point the root no longer has the minimum element. So the node at the root starts to bubble down until non of its children are smaller than itself. This bubble down process is shown in figure 3.13 through figure 3.16.

Similarly read() operation returns the minimum priority element of the heap. As this element is stored at the root of the heap, it returns the element at the root of the heap as in figure 3.17.

So far we have explained how a heap would work in a sequential operation. But the case is different when we are dealing with concurrent operations. If the original heap was as in figure 3.12 and we have two threads T_1 performing *delete()* operation and T_2 performing read() operation. After T_1 deletes the root element and copies the last element which is 25 in this case to the root position as in figure 3.13, T_2 comes along and reads and returns 25 as the minimum element. Since these two operations were overlapping we might assume either T_1 or T_2 completed first. If T_1 had completed before T_2, T_2 would return 10. And if T_2 had completed first, it would return 5. In either case 25

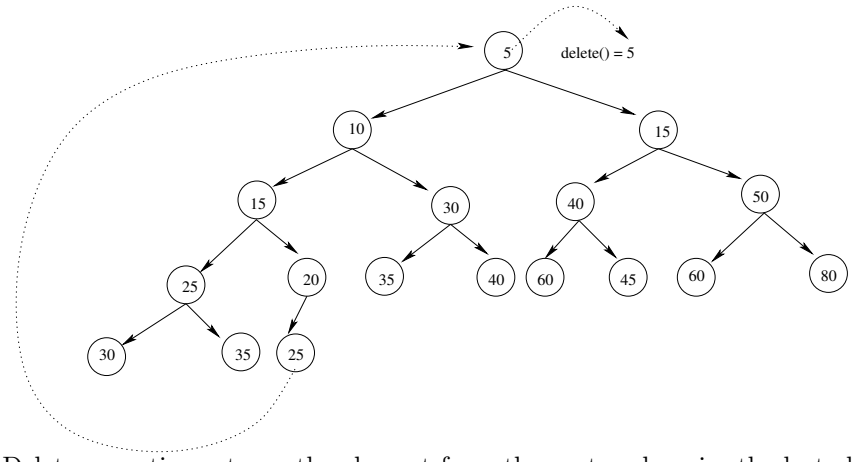

Figure 3.12: Delete operation returns the element from the root and copies the last element to the root.

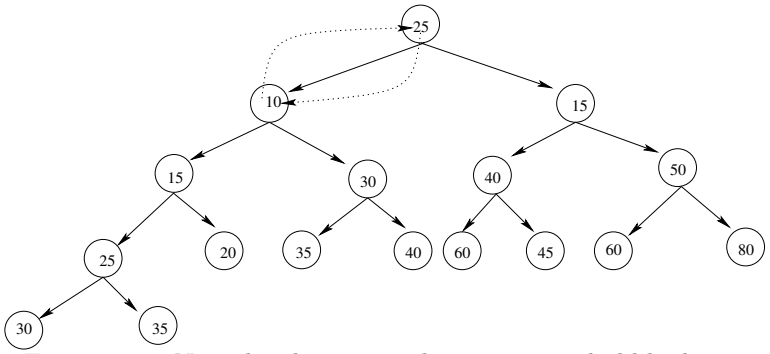

Figure 3.13: Now the element at the root starts bubble down

would not be the correct answer. So this is obviously not linearizable.

We need a mechanism to let threads communicate with each other whenever they are trying to work in the same heap. Classical approach would be to lock the heap. Whenever a thread tries to access the heap, it should first acquire a lock. And if it succeeds in acquiring the lock, it can proceed further with any update in the heap. And it releases the lock only when it is done with the heap. If it fails to acquire the lock, then we can make it spin around the lock or implement some wait signal approach. To improve the performance we might have different kinds of locks for read and write operations. However, we would be loosing our valuable resources. Say for example when T_1 is in the middle of delete() operation as in figure 3.12, another thread T_3 could be waiting for insert(35) operation. From this point they would never interfere each other. We are really making T_3 wait for nothing.

Herlihy and Shavit[16] tried to improve the approach by using fine grained locks. Rather than locking the whole heap, they suggested using the locks at the nodes. Locking nodes and its children or its parent whichever is necessary during bubble down or bubble up but not locking the whole heap.

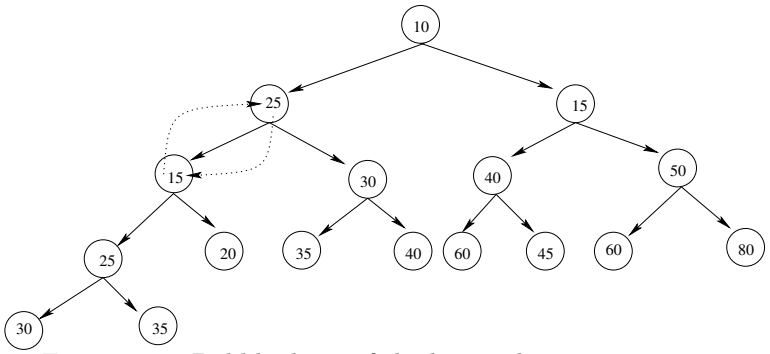

Figure 3.14: Bubble down of the larger element continues.

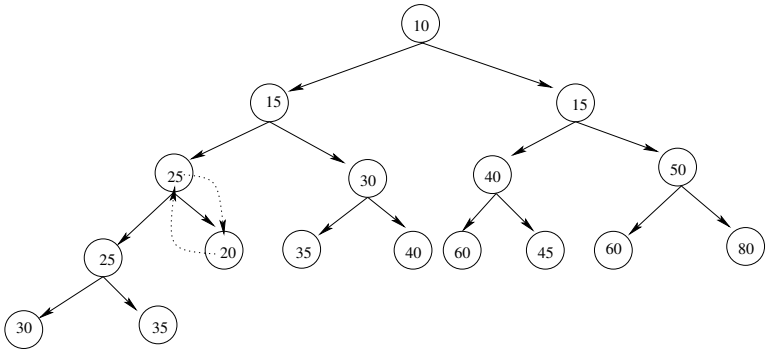

Figure 3.15: Bubble down still needs to be carried on.

This approach certainly improves the performance as two threads which are working at different branches of the heap never have to wait for each other. But as they are using locks, this approach isn't free from having to wait for another thread which holds the lock and is too slow or dies while still holding the lock. As an example we can refer to the figure 3.18. Here threads T_1 and T_2 are middle of the operation delete() and are currently undergoing bubble down. However, at this point T_2 has to wait for T_1 as T_1 holds the lock needed by T_2. But if T_1 takes a long time to complete or crashes while still holding the lock and T_2 may be waiting for ever.

Israeli and Rappoport[20] gave a non-blocking implementation which supports deleteMin and insert operations using some strong atomic primitives which they claim could be built using transactional memory. They have suggested an approach of using some bits from the value to contain the information required for stating the operation currently being carried out with that node. With such information stored, even if the process dies, another thread which comes across this node can help it with the operation. In the figure 3.18 even if T_1 crashes while trying to perform some operation on the node, T_2 can help it by observing the bits allocated for representing the information of the node. So a thread has to write this information before performing the actual operation. This algorithm has made a significant effort towards making the algorithm a non-blocking one. However, they haven't given any algorithm for read() operation. And we have provided a wait free approach

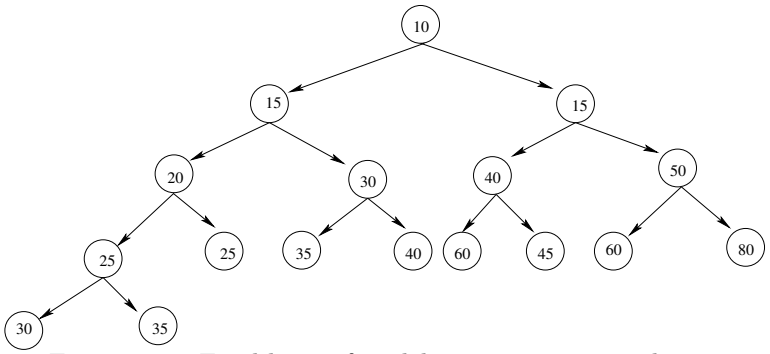

Figure 3.16: Final heap after delete operation completes.

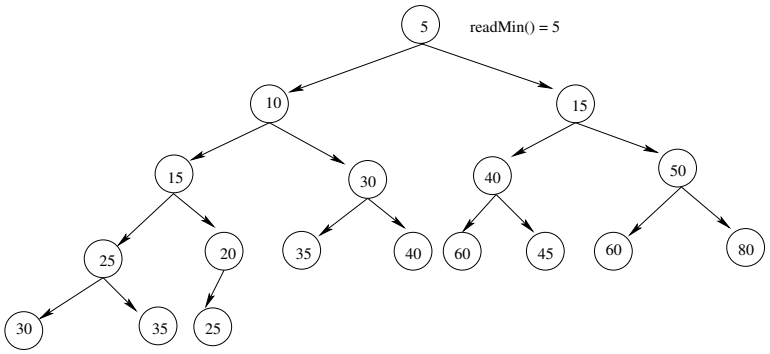

Figure 3.17: readMin operation returns the element at root.

for this operation. Also they have reserved certain bits from the value to store the information about the operation on the node. This lowers the range of values that can be stored in the node. We avoided this constraint by storing these extra information in separate word.

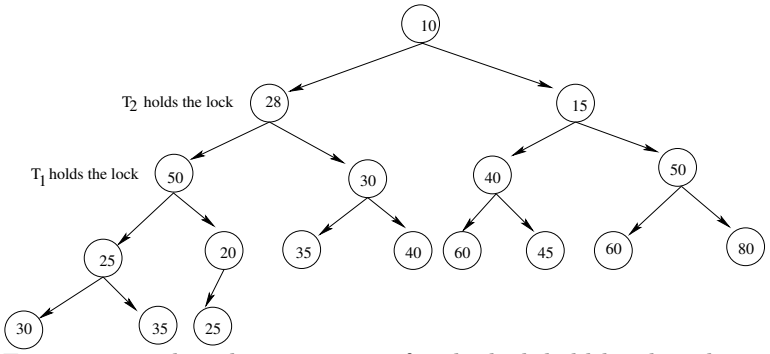

Figure 3.18: Thread T_2 is waiting for the lock held by thread T_1.

Chapter 4

Model

We have considered the asynchronous shared memory model as the system where the data structure will be used. The system should also support Compare and Set(CAS) and Double Compare and Set(DCAS) primitive operations by hardware. The data structure can be accessed by multiple threads at the same time. We will discuss in this chapter how the heap is represented and what information is stored at each node.

4.1 Data Representation

We are implementing heap using arrays. The root of the heap at index 1 of the array. All nodes of the heap have two words stored per node. One for the value(priority) and the other for the information. To make it easier to handle two words at each location in array, we use two arrays. One of the array is called heap which stores the value while the other array stores information. Now for accessing the information associated with the node we use *node.info*. If the node is located at the index 5, node.info would point to 5th index of the array containing information about the node.

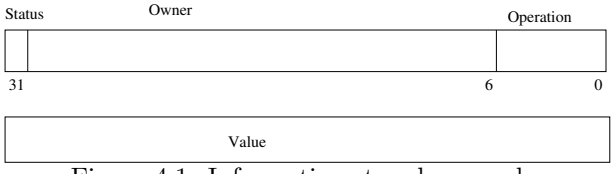

Figure 4.1: Information stored per node

As shown in the figure 4.1 the information word contains three values. The lowest 6 bits indicate the operation associated with the node. If these bits value is equal to zero then no operation is being carried out. Then next 25 bits indicate the owner of the node. Here the owner represents the thread

id. If these bits value is equal to zero then no thread has owned the node. The most significant bit is the status bit. This bit is set to 1 for all nodes which are in the heap and set to 0 if the node gets deleted.

Each thread also has certain information stored with it. Associated with each thread are three things: information, value and size of the heap. The word in information holds three values. Just like the information in the node, information field has operation(6 least significant bits) followed by node id which occupies next 25 bits and hold the index of the node in the heap on which the thread is working upon and most significant bit represents the status. Value 1 in the status represents that the thread is still interested in the heap while 0 indicates that the thread is done using the heap.

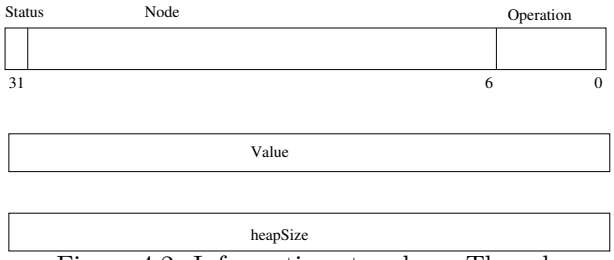

Figure 4.2: Information stored per Thread

Now the value field stores the value the thread is either going to insert or the value it has deleted. This is required because other thread which want to help this thread in inserting its value in the heap should be able to figure out which value was to be inserted. Similarly if they help in deleting a node, it should be able to figure out which value was deleted. Finally, *heapSize* stores the size of the heap just before it started deleting the node. As there could be many threads working concurrently in the heap, the size of the heap keeps on changing. So other threads which want to help it, should know which node it was planning to swap to the root.

Thread with ID = 5 will have its information stored at owner[5]. Its value will be stored at value[5] and accessed as 5.value while its heapSize is stored at heapSize[5] and accessed as 5.heapSize(). While a thread can access any values of from any other threads, it can also write to any arrays of other threads except the heapSize array. Thread with id = 5 can write to any location in owner and value array but in heapSize array it can only write to heapSize[5] location.

4.1.1 Global Variables

Operations

1: $GETMINVAL \leftarrow 000001$

2: $COPYLAST \leftarrow 000010$

3: $HELPDOWN \leftarrow 000011$

4: $ADDVAL \leftarrow 000100$

5: $HELPUP \leftarrow 000101$

6: $SWAPUP \leftarrow 000110$

7: $SWAPDOWN \leftarrow 000111$

Nodes

1: $UNLOCKEDNODE \leftarrow 10000000000000000000000000000000$

2: $EMPTYNODE \leftarrow 00000000000000000000000000000000$

Threads

1: $OWNERDONE \leftarrow 00000000000000000000000000000000$

Status

1: $ACTIVE \leftarrow 1$

2: $PASSIVE \leftarrow 0$

3: $DONE \leftarrow 0$

Child ID

1: $LEFT \leftarrow 0$

2: $RIGHT \leftarrow 1$

4.2 Primitive Operations

4.2.1 Compare and Set

CAS(Compare and Set)[19] sometimes called Compare-and-Swap instruction compares the value in a memory location with a value in a data register and copies a second data register into a memory location if the compared values are equal. After CAS has read a value from the memory no other instruction can change that memory location until CAS is done with writing at the location. In single processor system, the operation is protected from interrupts while in multi processor system all other processing wishing to access the global pointer must wait until the CAS completes.

Procedure 1 CAS(*addr, expectedValue, newValue)

```
if *addr = expectedValue then
  *addr ← newValue
  return true
end if
return false
```

4.2.2 Double Compare and Set

DCAS[19] is commonly called double compare and set and sometimes referred to as CAS2. The DCAS instruction is similar to the CAS instruction except that it performs two comparisons and updates two variables when the results of the comparisons are equal. If the results of both comparisons are equal, DCAS copies new values into the destination addresses and returns true. If the result of either comparison is not equal then it returns false. Thus we see that this operation is a stronger than CAS(Compare and Set) which is available in almost all modern architectures. Unlike CAS, DCAS is not very common with modern architectures. It is supported by Motorola 68k family of processors. A canceled Rock Processor[4] from Sun would also have supported DCAS.

Procedure 2 DCAS(*addr0, expVal0, newVal0, *addr1,expVal1,newVal1)

```
if ((*addr0 = expVal0) and (*addr1 = expVal1))  then
  *addr0 = newVal0
  *addr1 = newVal1
  return true
end if
return false
```

Hardware implementation of DCAS had been found to be somewhat slow so an alternative approach for software implementation for DCAS was also suggested by Brian[3]. He suggested the use of lock known to the operating system to implement DCAS. If a process which has a lock looses CPU due to context switch, the operating system rolls back the process out of DCAS procedure and releases the lock. However Michael[12] points out some of the issues with this approach. First it needs to use the locks. Second, one needs to be more careful when asynchronized reads need to be supported because it is likely that readers might see intermediate values which could later be rolled back.

Execution of these instruction as one atomic block makes it a strong primitive and also might make it costly in terms of execution speed. Still Simon[10] points out that DCAS doesn't provide any sufficient additional power over CAS to implement a non-blocking data structure so he suggested the use of further strong primitives like multi-word compare and swap. However, we have managed to implement heap using DCAS operations only.

Chapter 5

Subsidiary Operations

We have three operations readMin, deleteMin and insert to be performed in the heap. These operations are long and complex. So we have added a number of other operations which are used to implement these operations. The necessity to create these subsidiary operations is justified as these operations need to be called multiple times within the main operations.

5.1 getParent

This procedure is used to get the parent node of a node. In an array based implementation where the root node is at heap[1],child of any node at heap[i] is at heap[2*i] and heap [2*i+1]. So we can figure out the parent once a child node is given.

Procedure 3 getParent(node)

parent $\leftarrow \lfloor \frac{node}{2} \rfloor$
if parent < 1 **then**
  return null
else
  return parent
end if

5.2 getChild

Index of the left of right child for a node with given index is returned unless the child is outside the range of the heap size. If it is outside the heapSize it returns false.

Procedure 4 getChild(node,heap,childId)

```
if childId = LEFT then
  child ← 2× node
else
  child ← 2×node + 1
end if
heapSize ← heap.size
if child > heapSize then
  return null
else
  return child
end if
```

5.3 getOwner

This method extracts the operation field (from bit 0 to bit 5)from the info field and returns the same.

Procedure 5 getOperation(info)

```
Operation ← info & 31
return operation
```

5.4 setOwner

This method return the info word with the operation field replaced by the *operation*. For this we first set the bits representing operation by 0 and then we use bit wise or operation on it with the operation.

Procedure 6 setOperation(info,operation)

```
newInfo ←info & 4,294,967,232
newInfo ← newInfo | operation
return newInfo
```

5.5 getOperation

This method is used to return the value of the bits indicating owner field. The bits representing owner are the 6^{th} to 30^{th} bits.

Procedure 7 getOperation(info)

```
nodeOwner ← info ≪ 1
nodeOwner ← nodeOwner ≫ 6
return nodeOwner
```

5.6 setOperation

This method is used to change the value of the bits indicating owner field. The bits representing owner are the 6^{th} to 30^{th} bits.

Procedure 8 setOwner(info,operation)

```
owner ← owner ≪ 6
info ← info & 2,147,483,711
info ← info |owner
return info
```

5.7 getNode

This method returns the node which the thread is actually working upon. While a thread may also own two nodes at the same time, it is sufficient in our algorithm to store only the first one.

Procedure 9 getNode(ownerInfo)

```
node ← ownerInfo ≪ 1
node ← node ≫ 6
return node
```

5.8 setNode

This method updates the info field of the thread by replacing the bits 6^{th} through 30^{th}.

Procedure 10 setNode(ownerInfo,node)

```
node ← ownerInfo ≪ 6
info ← info & 2,147,483,711
info ← info |node
return info
```

5.9 getStatus

This method is used to both return whether the thread is done with the heap or not. This information is stored in the most significant bit of the info field.

Procedure 11 getStatus(info)

status ← info ≫ 31
return status

5.10 swapValues

This method takes two nodes, their corresponding info fields and the corresponding values. First the info field of the nodes are changed as in Figure 5.2 to reflect that it is in the process of swapping and its owners field are also changed at the same time.

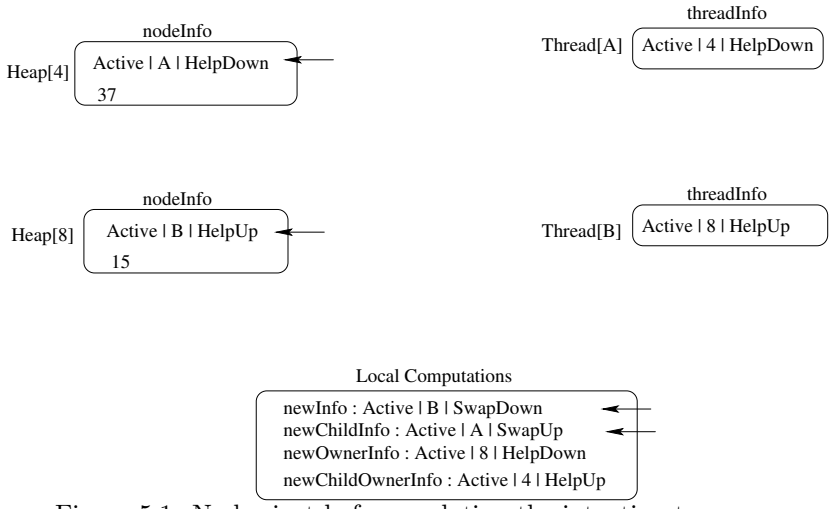

Figure 5.1: Nodes just before updating the intention to swap

Then the values are swapped. Above two are performed only if other threads haven't swapped the values yet. This checking is performed by comparing the values. Now the values being swapped, The respective info fields of the owners are changed as seen in Figure 5.4 so that they point to the new nodes. Finally the intention to swap the node is removed by changing the SWAPUP and SWAPDOWN field by HELPUP and HELPDOWN field respectively.

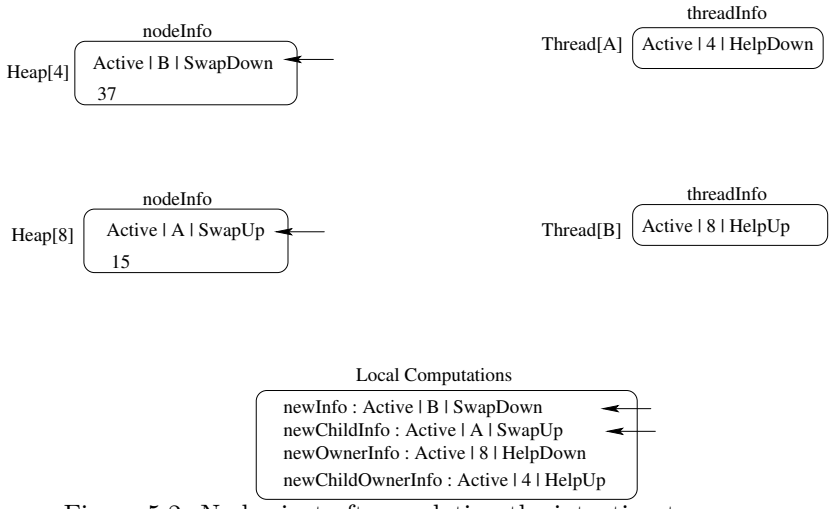

Figure 5.2: Nodes just after updating the intention to swap

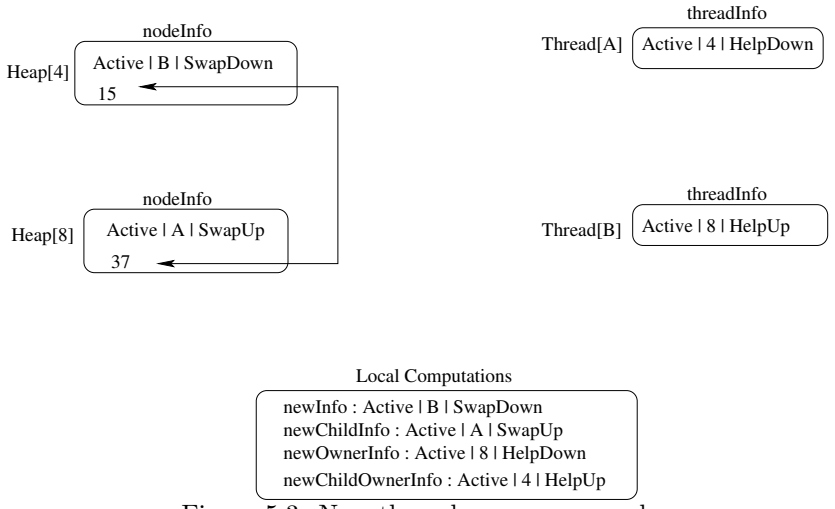

Figure 5.3: Now the values are swapped

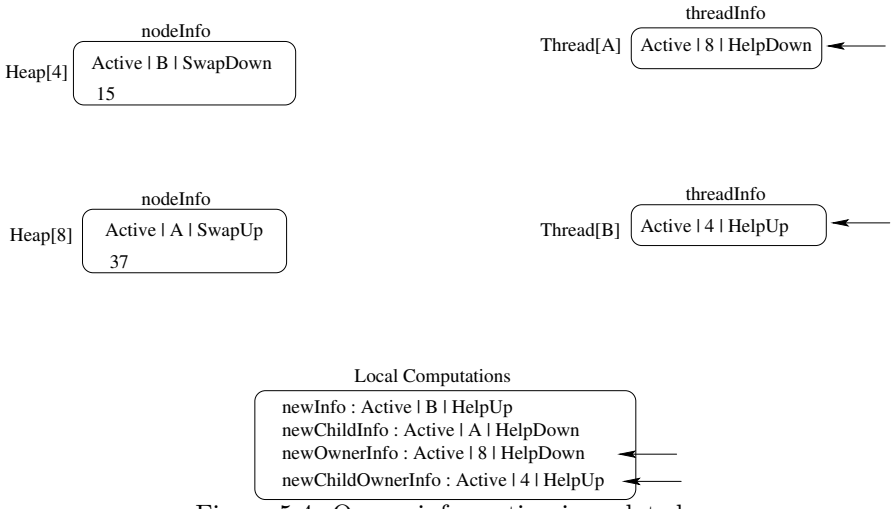

Figure 5.4: Owner information is updated.

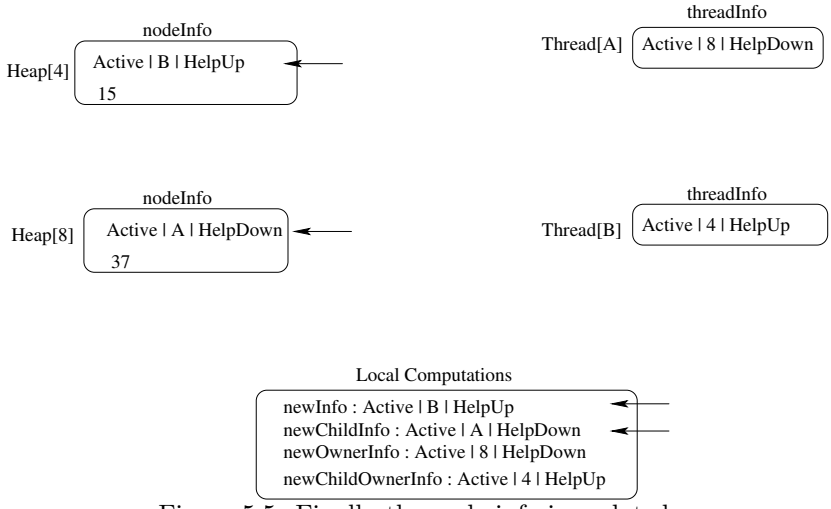

Figure 5.5: Finally the node info is updated

Procedure 12 swapValues(node,childNode,nodeInfo,childInfo,nodeValue,childValue)

```
owner ← getOwner(nodeInfo)
childOwner ← getOwner(childInfo)
newInfo ← setOperation(childInfo,SWAPDOWN)
newChildInfo ← setOperation(nodeInfo,SWAPUP)
if owner >0 then
  ownerInfo ← owner.info
  newOwnerInfo ← setNode(ownerInfo,childNode)
end if
if childOwner >0 then
  childOwnerInfo ← childOwner.info
  nChildOwnerInfo ← setNode(childOwnerInfo,node)
end if
if nodeValue >childValue then
  DCAS(node.info,nodeInfo,newInfo,childNode.info,childInfo,newChildInfo)
  DCAS(node.value,nodeValue,childValue,childNode.value,childValue,nodeValue)
end if
if owner >0 and childOnwer >0  then
  DCAS(owner.info,ownerInfo,newOwnerInfo,childOwner.info,childOwnerInfo,nChildOwnerInfo)
  childInfo ← setOperation(newChildInfo,HELPDOWN)
  info ← setOperation(newInfo,HELPUP)
else
  if childOwner <1 then
    CAS(owner.info,ownerInfo,newOwnerInfo)
    info ← UNLOCKEDNODE
  else
    CAS(childOwner.info,childOwnerInfo,nChildOwnerInfo)
    childInfo ← UNLOCKEDNODE
  end if
end if
DCAS(node.info,newInfo,info,childNode.info,newChildInfo,childInfo)
```

5.11 helpDown

This method is used for bubbling down the larger values. It finds out the minimum values in the sub-trees rooted at both its children. Refer to section 6.1 for details on how to find the minimum in a sub-tree. If it is the smaller than the the minimum values from both sub-trees, it knows that it doesn't need to propagate any further. So it stops here and modifies the corresponding information. Refer to Figure 5.6 through Figure 5.9 for few examples.

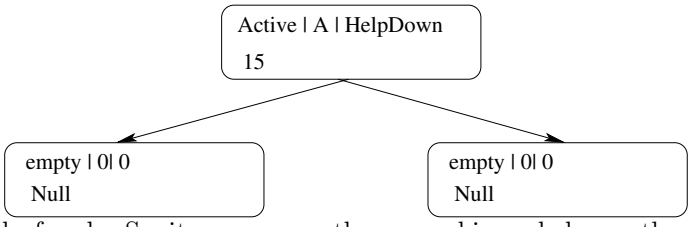

Figure 5.6: It is the leaf node. So, it can remove the ownership and change the owner information.

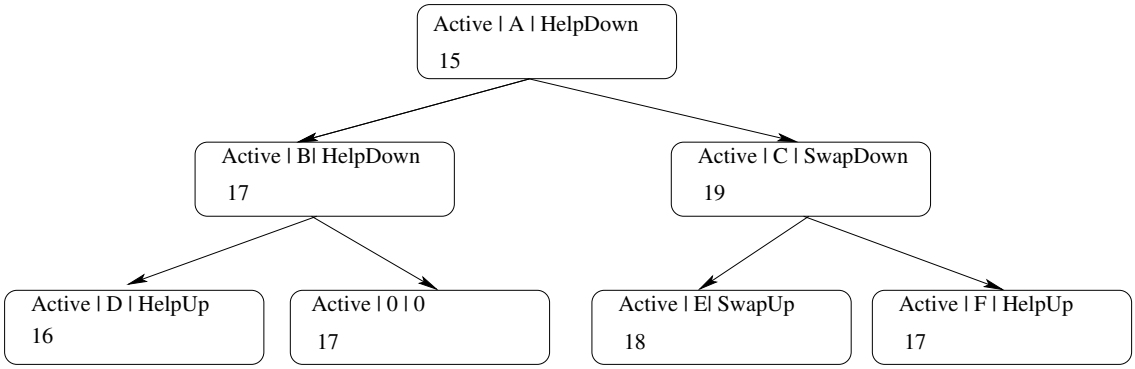

Figure 5.7: It finds out that it has the smallest value in its entire sub-tree. So, it can remove the ownership and change the owner information.

Otherwise, it marks the child with which it needs to work with. If the operation in that node is HELPDOWN or SWAPDOWN, it helps the node otherwise, it simply swaps the child node with itself.

Procedure 13 helpDown(node,nodeInfo)

```
ownerInfo ← owner.info
ownerStatus ← getStatus(ownerInfo)
nodeVal ← node.value
nodeInfo ← node.info
leftChild ← getChild(node, LEFT)
if leftChild = null then
  DCAS(owner.info,ownerInfo,OWNERDONE,node.info,nodeInfo,UNLOCKEDNODE)
  return
end if
leftMin ← readMin(leftChild)
leftChildInfo ← leftChild.info
rightChild ← getChild(node, RIGHT)
if rightChild ≠ null then
  rightChildInfo ← rightChild.info
  rightMin ← readMin(rightChild)
else
  rightMin ← POSITIVEINFINITY
end if
if (nodeVal ≤ leftMin) and (nodeVal ≤ rightMin) then
  DCAS(owner.info,ownerInfo,OWNERDONE,node.info,nodeInfo,UNLOCKEDNODE)
  return
else
  if leftMin ≤ rightMin then
    minNode ← leftChild
  else
    minNode ← rightChild
  end if
end if
childInfo ← minNode.info
childValue ← minNode.value
childOwner ← getOwner(childInfo)
childOperation ← getOperation(childInfo)
if childOperation = HELPDOWN or SWAPDOWN then
  helpNode(minNode,childInfo)
else
  swapValues(node, minNode, nodeInfo, childInfo, nodeValue, childValue)
end if
```

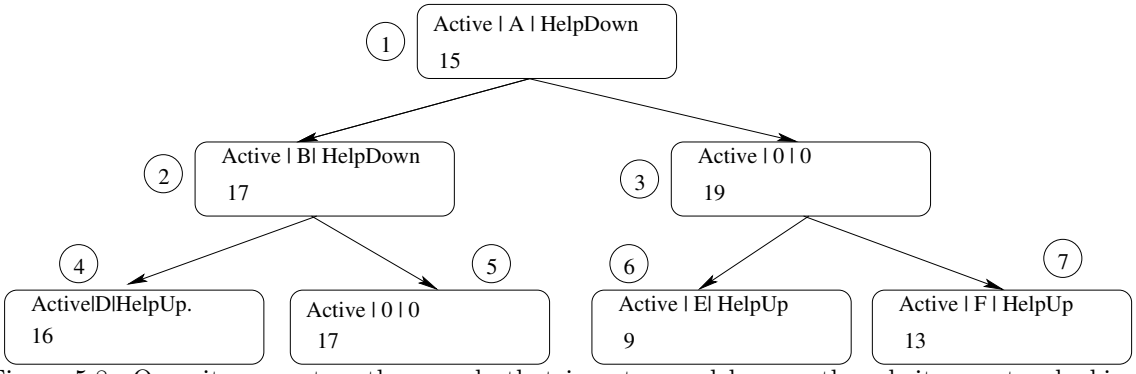

Figure 5.8: Once it encounters the a node that is not owned by any thread, it can stop looking for minimum value in the sub-tree. Here again, it can remove the ownership and change the owner information.

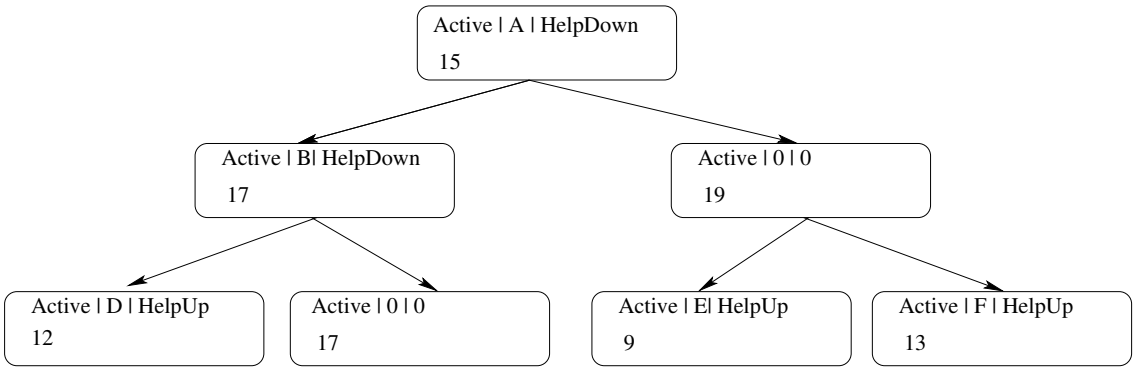

Figure 5.9: It finds out that minimum lies in the left sub-tree. So it has to help the left child.

5.12 helpUp

This method is used whenever a node needs to bubble up. For this purpose, parent of the node is identified. If the *helpUp* operation is found in the root as in Figure 5.10 then information of the root and that of the owner are updated to reflect the completion of job. If the parent is not owned by

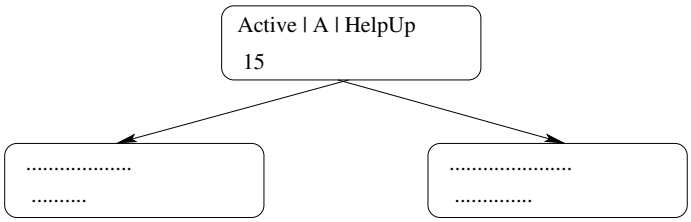

Figure 5.10: a case with helpUp operation on a root.

anybody or if it is owned and the operation in the info field is *HELPDOWN* then the parent value is compared with its value and two nodes are swapped if the parent value is smaller than it value. However, if the parent value is less than its value, completion status is updated in the owner of the field and the one being helped up. In Figure 5.11 we see that node 2 finds out that its parent(node 1) is locked with *helpDown* operation but parents value is smaller than its value. So, it can safely update the info of node 2 and that of its owner B to reflect the completion of job. In the same figure node 7 finds out that its parent(node 3) is not owned but its own value is smaller than its parent's. So node 7 will start the swap operation with its parent.

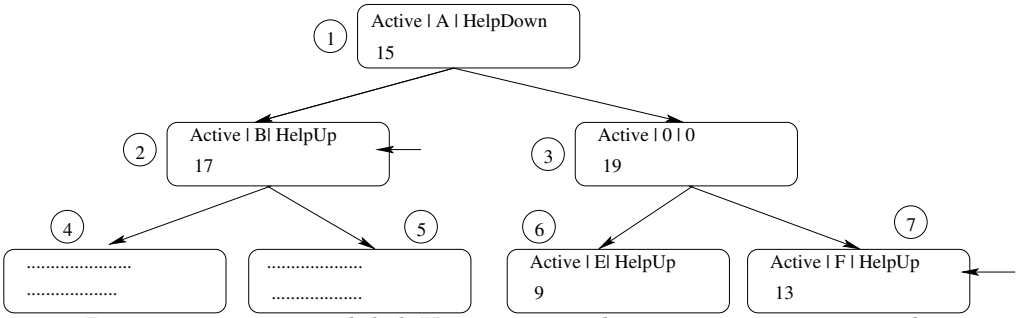

Figure 5.11: a case with helpUp operation where a swapping is required.

But if the parent is owned by some node and operation to be performed is *HELPUP*, then the node tries to help the parent node. In Figure 5.12 node 2 will need to perform *swapValues* operation with its parent but node 7 will have to help its parent(node 3) because the parent is owned and has to swap the values with node 6.

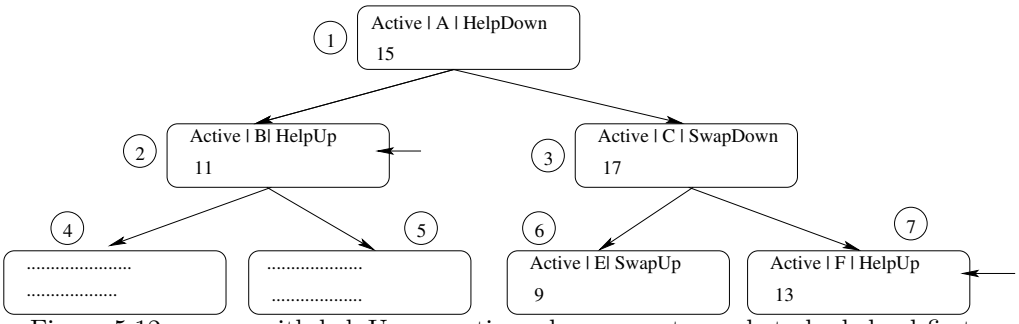

Figure 5.12: a case with helpUp operation where parents needs to be helped first.

Procedure 14 helpUp(node,nodeInfo)

```
 1: parent ← getParent(node)
 2: if parent = null then
 3:    DCAS(owner.info,ownerInfo,OWNERDONEINFO,node.info, nodeInfo,NODEDONEINFO)
 4: end if
 5: parentValue ← parent.value
 6: nodeValue ← node.value
 7: parentInfo ← parent.info
 8: nodeInfo ← node.info
 9: parentOwner ← getOwner(parentInfo)
10: parentOwnerInfo ← parentOwner.info
11: ownerInfo ← owner.info
12: parentOperation ← getOperation(parentInfo)
13: if (parentOwner = UNLOCKEDNODE) or (parentOperation = HELPDOWN) then
14:    if parentValue > nodeValue  then
15:       swapValues(parent,node,parentInfo,nodeInfo,parentValue,nodeValue)
16:    else
17:       DCAS(owner.info,ownerInfo,OWNERDONEINFO,
18:       node.info, nodeInfo,NODEDONEINFO)
19:    end if
20: else
21:    helpNode(parent,parentInfo)
22: end if
```

5.13 helpNode

This method takes node and info of the node as two arguments. It checks what operations need to be helped for the node and it performs tasks accordingly. During deleteMin and insert operations a several intermediate steps are required. Write to a shared variable during such intermediate steps is visible to others and other nodes should be able to figure out how exactly to help the node which is in one of those intermediate stages. We have already seen the details of helpUp and helpDown operations. This *helpNode* operation helps the node which is stuck in any of the intermediate stages. Now we discuss helping for each such intermediate stages.

GETMINVAL

A helper thread may find the operation *GETMINVAL* in the root node. A helper thread say thread B could be the one performing or helping one of the children of the root or a new thread trying to perform delete operation on the heap. If the operation field of the root has the value *GETMINVAL*, it indicates that a thread say thread A is in the middle deleteMin operation. And it has successfully owned the root node and decreased the heap size. Also it has copied the value of the root node as the minimum value to its private location. Also the node has already copied the size of the heap just before decreasing the heap size. However, this value is written to a place where only the thread initiating the deleteMin operation can write and any other threads can read. This is done to allow the helper threads to copy that last node to the root and bubble down.

Now, the next immediate job thread B has to perform is to own that last node for thread A. Thread B reads the heapSize value from thread A to identify the node it has to own. If that node is not owned by anybody, it changes the information of the last node to make to reflect the new owner and also the next operation to be performed. Just to make sure that it is not too late for thread B to help thread A, B also checks that nothing has been changed to the root. However, if the last node is already owned by some other thread say thread C and thread C is going to perform helpUp operation on the last node then thread B releases thread C of its duty and owns the last node for thread A. It doesn't make the heap inconsistent because finally this value will bubble down from the root to its correct place. However, if thread C is performing some other operation say swap or helpDown thread B helps thread C.

COPYLAST

A helper thread may find the operation *COPYLAST* in the last node. A helper thread say thread B finding the the operation in last node to be COPYLAST indicates that thread B could be trying to insert a new node in the heap or is trying to help some other thread say thread A. Thread A on the other hand is in the middle of the operation deleteMin. By this time both the root node and the last node have been owned by thread A.

Now thread B helps thread A by copying the value from the last node to the root and writing this information in the thread A's shared memory. This is done only if no other thread or thread A itself hasn't done this. Having done this thread B now turns the last node to be an empty node and changes the information in the root node so that any other thread trying to own root know that the next thing they need to do is help thread A with the operation helpDown.

HELPDOWN

A helper thread may need to help another thread which needs to bubble down the heap. In this case helper thread follows the helpDown as shown inf Procedure 5.11.

ADDVAL

When some thread say thread B discovers that an operation on a node by another thread say A is ADDVAL, thread B knows that thread A was in the middle of insert operation and has already owned the node and also increased the. heap size.

Now, thread B can help thread A by copying the value thread A intended to insert into the node value. Once this insertion of the value in to the node is done, thread B updates the information of the node and that of thread A to reflect that the next job to be performed is helpUp.

Thread B could have gone sleeping or somehow taken a longer period in between these steps. At the same time many other threads could have tried to help thread A. Many of them may be successful in inserting the value into the thread but only one of them will succeed in updating the information of the node and thread A.

HELPUP

A helper thread may need to help another thread which needs to bubble up in the heap. In this case helper thread follows the helpUp as shown inf Procedure 5.12.

SWAPDOWN

A helper thread say thread B if finds out that the node is owned by some other thread say thread A and operation in the node is *SWAPDOWN*, then it is clear that thread A is undergoing bubble up and it is at the moment in the middle of swapping the node owned by some other thread say thread C. By this time the owners of the nodes have swapped their ownership.

So thread B has to find out which child of the node was actually previously owned by thread A(before swapping the ownership with thread C). The child node which was previously owned by thread A has the operation *SWAPUP*. Thread B then tries to swap the values of those nodes. However some other threads might have already helped in swapping the values. Thread B only helps in swapping the values if the parent node has higher value than its child node. Otherwise thread B assumes that some other thread has already helped in swapping the values.

Once the values are swapped, now the information of the thread A and thread C are changed to reflect the correct node they now point to. Finally, the information of the corresponding nodes are updated to reflect that the parent now should continue helpUp (bubble up) operation and child should continue helpDown (bubble down) operation.

SWAPUP

This is very much similar to the swapDown operation.A helper thread say thread B if finds out that the node is owned by some other thread say thread A and operation in the node is SWAPUP, then it is clear that thread A is undergoing bubble down and it is at the moment in the middle of swapping the node owned by some other thread say thread C. By this time the owners of the nodes have swapped their ownership.

Thread B then tries to swap the values of the node with that of its parent node. However some other threads might have already helped in swapping the values. Thread B only helps in swapping the values if the parent node has higher value than its child node. Otherwise thread B assumes that

some other thread has already helped in swapping the values.

Once the values are swapped, now the information of the thread A and thread C are changed to reflect the correct node they now point to. Finally, the information of the corresponding nodes are updated to reflect that the parent now should continue helpUp (bubble up) operation and child should continue helpDown (bubble down) operation.

Procedure 15 helpNode(node,nodeInfo)

```
nodeOwner ← getOwner(nodeInfo)
nodeOperation ← getOperation(nodeInfo)
nodeValue ← node.value
if (nodeOperation = GETMINVAL  then
  goto GETMINVAL
end if
if (nodeOperation = COPYLAST  then
  goto COPYLAST
end if
if (nodeOperation = HELPDOWN  then
  goto HELPDOWN
end if
if (nodeOperation = SWAPDOWN  then
  goto SWAPDOWN
end if
if (nodeOperation = SWAPUP  then
  goto SWAPUP
end if
if (nodeOperation = ADDVAL  then
  goto ADDVAL
end if
if (nodeOperation = HELPUP  then
  goto HELPUP
end if
GETMINVAL:
heapSize ← nodeOwner.heapSize
lastNode ← HEAP[heapSize]
lastInfo ← lastNode.info
lastOperation ← getOperation(lastInfo)
lastOwner ← getOwner(lastInfo)
nNodeInfo ← setOwner(lastInfo,nodeOwner)
nNodeInfo ← setOperation(nNodeInfo,COPYLAST)
if lastOperation = HELPUP then
  lastOwnerInfo ← lastOwner.info
  DCAS(lastNode.info,lastInfo,nNodeInfo,
  lastOwner.info,lastOwnerInfo,OWNERDONE)
else
  if lastInfo = UNLOCKEDNODE then
    DCAS(lastNode.info,lastInfo,nNodeInfo,node.info,nodeInfo,nodeInfo)
  else
    helpNode(lastNode,lastInfo)
  end if
end if
return
```

```
COPYLAST:
rootValue ← nodeOwner.minValue
rootInfo ← root.info
nodeOwner ← getOwner(nodeInfo)
ownerInfo ← nodeOwner.info
if getOperation(ownerInfo) = GETMINVAL then
  nOwnerInfo ← setOperation(OwnerInfo,HELPDOWN)
  DCAS(nodeOwner.info,OwnerInfo,nOwnerInfo,root.value,rootValue,nodeValue)
end if
if getOperation(rootInfo) = GETMINVAL then
  nRootInfo ← setOperation(rootInfo,HELPDOWN)
  DCAS(root.info,rootInfo,nRootInfo,node.info,nodeInfo,EMPTYNODE)
end if
return
HELPDOWN:
helpDown(node,nodeInfo)
return
ADDVAL:
value ← nodeOwner.value
DCAS(node.info,nodeInfo,nodeInfo,node.value,nodeValue,value)
newNodeInfo ← setOperation (info,HELPUP)
newOwnerInfo ← setOperation (ownerInfo,HELPUP)
newOwnerInfo ← setNode (newOwnerInfo,node)
DCAS(self.info,ownerInfo,newOnwerInfo,node.info,info,newNodeInfo)
return
HELPUP:
helpUp(node,nodeInfo)
return
SWAPDOWN:
leftChild ← getChild(node,LEFT)
childInfo ← leftChild.info
childOperation ← getOperation(childInfo)
if childOperation = SWAPUP then
  childNode ← leftChild
else
  rightChild ← getChild(node,RIGHT)
  childInfo ← rightChild.info
  childOperation ← getOperation(childInfo)
  if childOperation = SWAPUP then
    childNode ← rightChild
  else
    return
  end if
end if
childValue ← child.value
if nodeValue >childValue then
  DCAS(node.value,nodeValue,childValue,childNode.value,childValue,nodeValue)
end if
```

```
newInfo ← setOperation(nodeInfo,HELPUP)
newChildInfo ← setOperation(childInfo,HELPDOWN)
ownerInfo ← nodeOwner.info
childOwnerInfo ← childOwner.info
ownerNode ← getNode(ownerInfo)
childOwnerNode ← getNode(childOwnerInfo)
newOwnerInfo ← setNode(ownerInfo,childNode)
nchildOwnerInfo ← setNode(newOwnerInfo,node)
if ownerNode = childNode and childOwnerNode = node then
  DCAS(nodeOwner.info,ownerInfo,newOwnerInfo,
  childOwner.info,childOwnerInfo,nChildOwnerInfo)
end if
DCAS(node.info,nodeInfo,newInfo,childNode.info,childInfo,newChildInfo)
return
SWAPUP:
parent ← getParent(node)
parentInfo ← parent.info
parentOperation ← getOperation(parentInfo)
if parentOperation ≠ SWAPDOWN then
  return
end if
parentValue ← parent.value
if nodeValue <parentValue then
  DCAS(node.value,nodeValue,parentValue,parent.value,parentValue,nodeValue)
end if
newInfo ← setOperation(nodeInfo,HELPDOWN)
newParentInfo ← setOperation(parentInfo,HELPUP)
ownerInfo ← nodeOwner.info
parentOwnerInfo ← parentOwner.info
ownerNode ← getNode(ownerInfo)
parentOwnerNode ← getNode(parentOwnerInfo)
newOwnerInfo ← setNode(ownerInfo,parent)
nParentOwnerInfo ← setNode(parentOwnerInfo,node)
if ownerNode = parent and parentOwnerNode = node then
  DCAS(nodeOwner.info,ownerInfo,newOwnerInfo,
  parentOwner.info,parentOwnerInfo,nParentOwnerInfo)
end if
DCAS(node.info,nodeInfo,newInfo,parentNode.info,parentInfo,newParentInfo)
return
```

Chapter 6

Operations

So far we have discussed the data representation, hardware primitives and subsidiary procedures. Now we are ready to implement the main operations readMin, deleteMin and insert. These operations are the ones that will be visible to the users of the data structure.

6.1 readMin operations

This method finds the likely minimum value propagating towards root in the sub-tree rooted at the given node. If the node is not locked, it returns the value held by the node. Even if the node is locked and the operation being performed is HELPUP, it implies that the node is being propagated up, so it is safe to return this value as the minimum value of the sub-tree rooted at the node. This doesn't violate the linearizability property.

However, if the node is locked and the operation is not HELP UP then minimum values are found

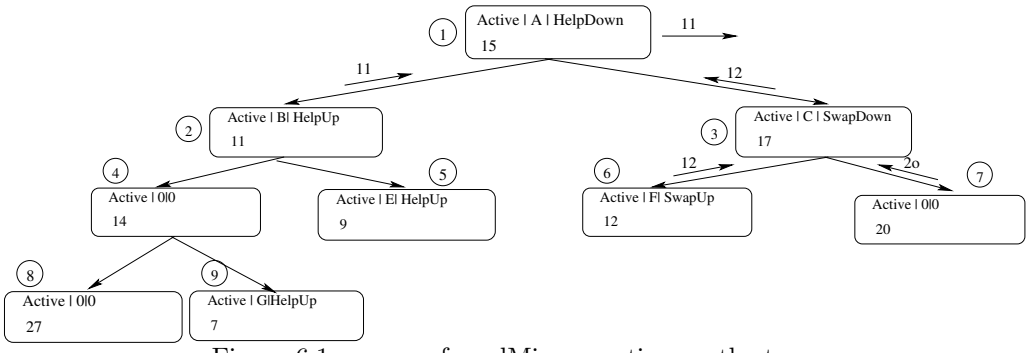

Figure 6.1: a case of readMin operation on the tree

in both sub-trees. Then the minimum value among itself and sub-trees rooted at its children is returned as the minimum value. If the node is locked and the operation is not HELPUP, it is likely

that some minimum value is residing below it and the node was just in the process of bubbling down. So we had to look at its children too.

In Figure 6.1 Node 1 is locked so it seeks minimum values from both its child nodes(ie node 2 and node 3). Node 2 is owned by thread B with operation helpUp so it doesn't need to check for any values in its children and returns 11. Although we see that node 9 has the lowest value 7, it was valid for node 2 to return 11 as it can be assumed that addition of node 9 is done later than the readMin operation. At the same time node 3 seeks minimum from its children. Although both its children are owned by different threads, node 6 can return its value 12 because the value is propagating upwards. Same is the case with node 7. Now node 3 upon receiving values from its children computes the minimum value among its own value and those from the children. So it sends 12 as the minimum value to its parent node 1. Node 1 finally return 11 as the min value.

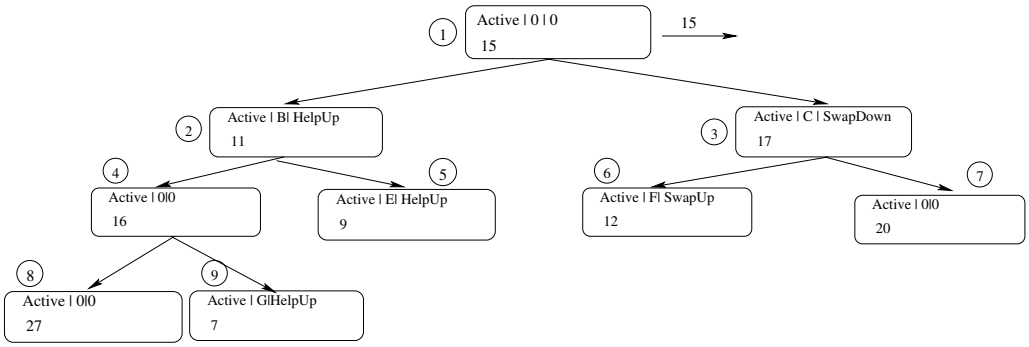

Figure 6.2: another case of readMin operation on the tree

Again in Figure 6.2 readMin operation at node 1 would return its own value ie 15 although there are so many smaller values in its sub-tree. But is is okay for node 1 to return its own value it is because node 1 is not owned by any other threads. All the smaller values which are in its sub-tree are still propagating upwards which means their insertion is not complete yet. So, it can be assumed that those insertion happened after the read min operation.

We see that the readMin operation is both *non-blocking* and *wait free*.

Procedure 16 readMin(node)

```
info ← node.info
value ← node.value
operation ← getOperation(info)
if (info = UNLOCKEDNODE) or (operation = HELPUP or SWAPUP) then
  return value
else
  leftChild ← getChild(node,LEFT)
  if leftChild = null then
    return value
  end if
  leftVal ← readMinVal(leftChild)
  rightChild ← getChild(node,RIGHT)
  if rightChild = null then
    rightVal ← POSITIVEINFINITY
  else
    rightVal ← readMinVal(rightChild)
  end if
  if value ≤ leftVal and value ≤ rightVal then
    return value
  else
    if leftVal ≤ rightVal then
      return leftVal
    else
      return rightVal
    end if
  end if
end if
```

6.2 insert operation

This method is used for adding a new node with a given value. It prepares the info field for the node, and gets the heap size. Using DCAS operation it updates the heap size and the info field. Now again it makes changes to the operation section of the info field and uses DCAS to update the value of the newly added node and the info field. Now the node has been added. Another DCAS is required to update the info field of the thread wishing to insert the node. Now the newly inserted nodes needs to bubble up. Until the owner is done, it helps the node to propagate up. Sometimes even

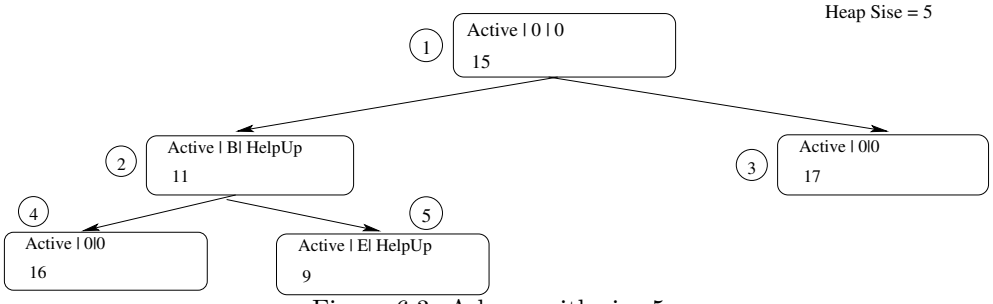

Figure 6.3: A heap with size 5.

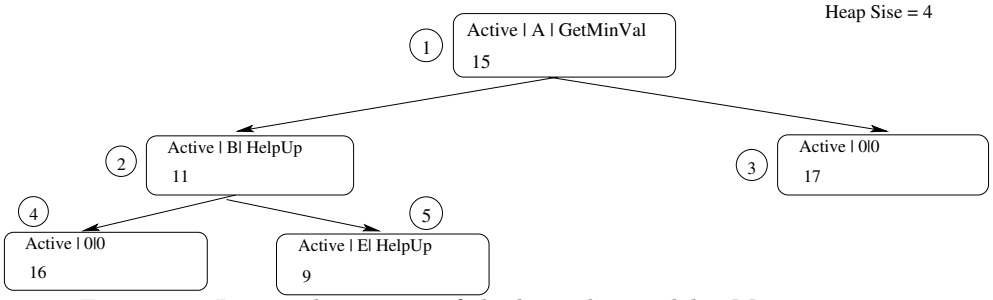

Figure 6.4: Intermediate stage of the heap during deleteMin operation.

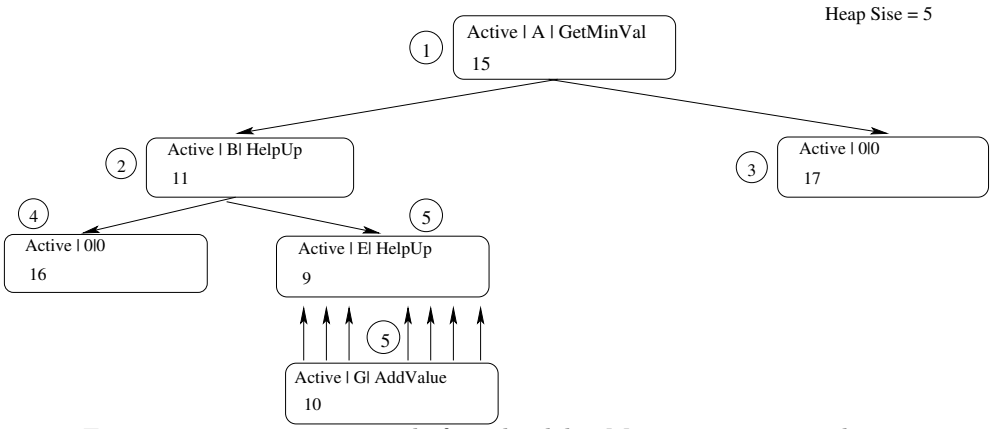

Figure 6.5: insert operation before the deleteMin operation completes.

after increasing the heapSize, it may not be a good idea to insert the node immediately to the last position. We will need to check if the position is empty. Consider the following case as in the Figure 6.3. Here the heap size is 5. Then comes thread A which starts performing deleteMin operation on the heap. The heap size changes to 4. However, it has not finished the operation deleteMin. The resultant heap is shown in Figure 6.4. In the mean time another thread say thread G starts its insert operation as in Figure 6.5. It increments its heap size to 5. G creates a separate node but it doesn't replace the node 5 which has not been free yet. So it needs to help the operation by E or the operation by A(which will soon copy the node to the root) before replacing the node by its node. Again, there could be couple of more threads lined up to replace the node 5. Only one of them succeeds. Others will keep on helping the one which successfully owns the node.

Procedure 17 insert(heap,value)

```
info ← setStatus(BLANKINFO,ACTIVE)
info ← setOperation(info,ADDVALUE)
info ← setOwner(info,self)
while TRUE do
  size ← heap.Size
  newSize ← size+1
  node ← HEAP[newSize]
  nodeInfo ← node.info
  nodeStatus ← getStatus(nodeInfo)
  if nodeStatus = ACTIVE then
    helpNode(node,nodeInfo)
  else
    nodeValue ← node.value
    ownerInfo ← self.info
    if DCAS(heap.size,size,newSize,node.Info,BLANKNODE,info) then
      break
    end if
  end if
end while
DCAS(node.info,info,info,node.value,nodeValue,value)
newNodeInfo ← setOperation (info,HELPUP)
newOwnerInfo ← setOperation (ownerInfo,HELPUP)
newOwnerInfo ← setNode (newOwnerInfo,node)
DCAS(self.info,ownerInfo,newOnwerInfo,node.info,info,newNodeInfo)
while TRUE do
  ownerInfo ← self.info
  ownerStatus ← getStatus(ownerInfo)
  nodeInfo ← node.info
  if ownerStatus = DONE then
    break
  else
    helpUp(node,nodeInfo)
  end if
end while
```

6.3 deleteMin Operation

This method is used to delete the highest priority element from the heap. If the heap doesn't contain any elements it returns error. So it first tries to own the root of the heap. If the heap is owned by some other thread it tries to help that thread. After trying to help the thread it again tries to own the root. Claiming the ownership of the root and decreasing the heap size is done atomically. At each trial, it also copies the size of the heap before decrement and the value of the root to its own space which can't be modified by any other thread. If the snapshot of the heap just before Thread A tries to perform deleteMin operation was as in Figure 6.6 then after owning the root the snapshot would be as shown in the Figure 6.7.

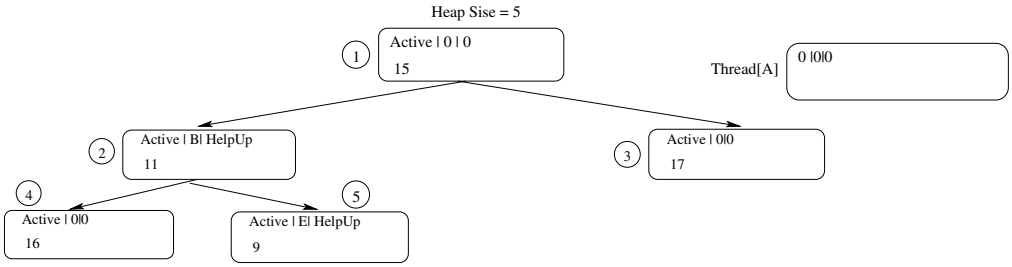

Figure 6.6: snapshot of a heap before deleteMin operation.

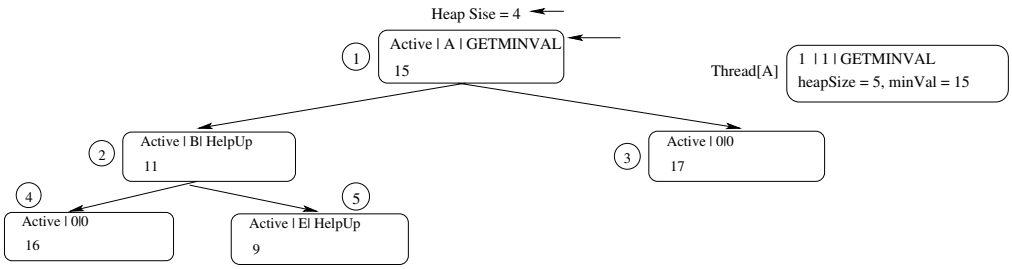

Figure 6.7: snapshot of a heap after decreasing the heap size.

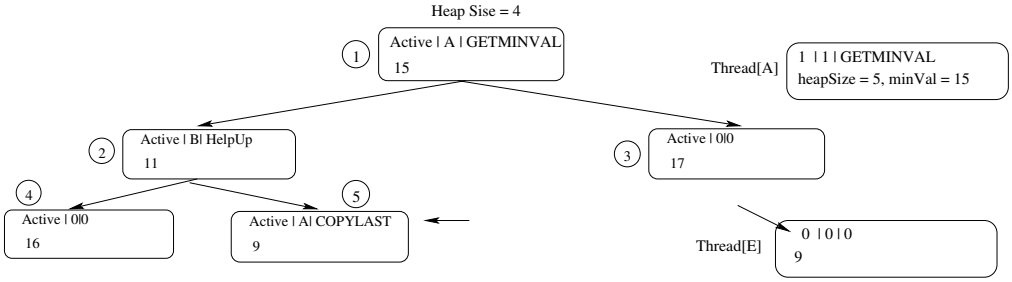

Figure 6.8: snapshot of a heap after getting the ownership of the last node.

Now the next job for the thread A is to try claiming the ownership of the last node. If the last node is owned by some other thread and the operation in the last node is *HELPUP*, then the

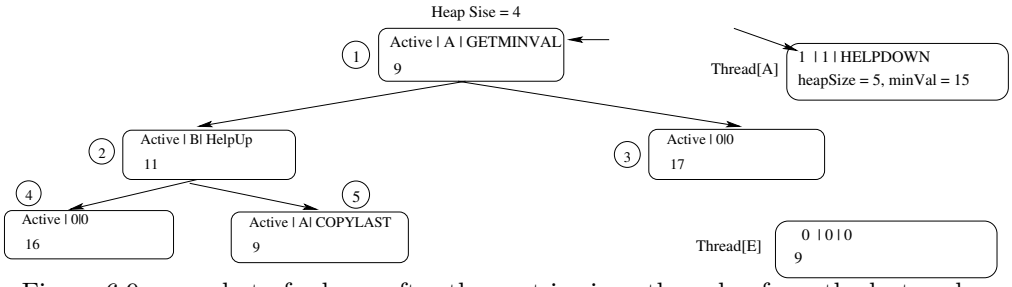

Figure 6.9: snapshot of a heap after the root is given the value from the last node.

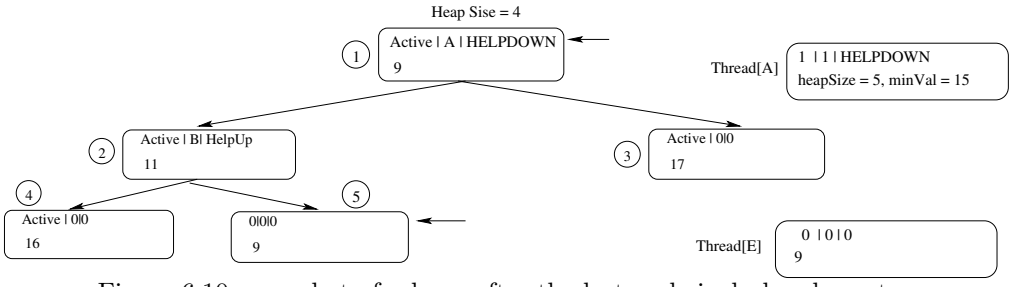

Figure 6.10: snapshot of a heap after the last node is declared empty.

information of the owner of the last node is changed to *DONE* and it owns the last node. Snapshot of the heap at this point could be something like Figure 6.8. The job gets easier if the last node is not owned by anybody. In that case it just has to change get the ownership of the last node.However, if the last node is already owned by some other threads then it has to help that thread before it can get the ownership.

Now, the value held by the last node is copied to the root node. Also the information stored by the thread is changed so that its operation field reflects *HELPDOWN*. Snapshot of the heap at this point might appear something like the Figure 6.9.

The thread now clears the information of the last node to reflect that it is an empty node and at the same time it also changes the information stored at the root that next operation it needs to under go is HELPDOWN. Now the normal business of helpDown carries on as described in section 5.11.

Procedure 18 deleteMin(heap)

```
while true do
  heapSize ← heap.size
  if heapSize < 1 then
    return error
  end if
  self.info ← setOperation(setNode(UNLOCKEDNODE,root),GETMINVAL)
  rootInfo ← root.info
  if rootInfo=UNLOCKEDNODE then
    nRootInfo ← setOwner(rootInfo,self)
    nRootInfo ← setOperation(nRootInfo,GETMINVAL)
    minValue ← root.value
    selfInfo ← self.info
    self.heapSize← heapSize
    self.value ← value
    if DCAS(heap.size,heapSize,heapSize-1,root.info,rootInfo,nRootInfo) then
      break
    end if
  else
    helpNode(root,rootInfo)
  end if
end while
lastNode ← HEAP[heapSize]
while true do
  lastInfo ← lastNode.info
  lastOperation ← getOperation(lastInfo)
  selfInfo ← self.info
  if lastOperation = COPYLAST or getOperation(selfInfo) = HELPDOWN or getSta-
  tus(selfInfo) = DONE then
    break
  end if
  lastOwner ← getOwner(lastInfo)
  nNodeInfo ← setOwner(lastInfo,self)
  nNodeInfo ← setOperation(nNodeInfo,COPYLAST)
  lastValue ← lastNode.value
  if lastOperation = HELPUP then
    lastOwnerInfo ← lastOwner.info
    if   (DCAS(lastNode.info,lastInfo,nNodeInfo,lastOwner.info,lastOwnerInfo,OWNERDONE)
    then
      break
    end if
  else
    if lastInfo = UNLOCKEDNODE then
      if (CAS(lastNode.info,lastInfo,nNodeInfo) then
        break
      end if
    else
      helpNode(lastNode,lastInfo)
    end if
  end if
end while
```

```
if getOperation(selfInfo) = GETMINVAL then
  nSelfInfo ← setOperation(selfInfo,HELPDOWN)
  DCAS(self.info,selfInfo,nSelfInfo,root.value,minValue,lastValue)
  nRootInfo ← setOperation(rootInfo,HELPDOWN)
  DCAS(root.info,rootInfo,nRootInfo,lastNode.info,lastInfo,EMPTYNODE)
end if
while true do
  ownerInfo ← self.info
  ownerStatus ← getStatus(ownerInfo)
  if ownerStatus = DONE then
    break
  else
    node ← getNode(ownerInfo)
    helpDown(node,self)
  end if
end while
return self.value
```

Chapter 7

Conclusion and Future Work

The implementation we have is a non-blocking approach for deleteMin and insert operations while the one for readMin is wait free. This algorithm could be very useful in cases which requires more number of reads. Since DCAS is a very expensive operation, it might not be a good idea to use this implementation where the size of heap is very small. We believe that this implementation could be very effective where the size of heap is very large and there are many threads concurrently changing the heap. In a large heap, these threads might be working at different places and they may not interfere with each other. If they don't interfere with one another then the business of helping is reduced which makes each thread to complete its job faster. Occasional helping required can be dealt without much work.

However few question still remain to be solved. We will explain some of them in detail.

Does *ABA[5]* problem exist? What is its impact? As we use CAS and DCAS, its obvious that ABA problem does exist. Does it have any undesirable consequence? We haven't found any cases where ABA problem had any undesirable situation. However, more cases need to be explored and some formal proof showing the harmless nature of the problem needs to shown. However, if the problem is found to have some negative impact it can be resolved by including some serial number or some other counter information in the info field of the node. Since this number doesn't need to be changed during the whole operation it won't have any negative impact on the performance. Creating counter by a thread can be done offline and each thread has its own counter value so there won't be any contention for it.

Can all the operations be made wait free? We see that once the threads have owned any node, they help other nodes only if they obstruct its path of progress. Such helping can't go for ever. So

its definitely a wait free. However, before a thread can own a node, it may end up helping other threads for ever. This raises the possibilities that it may starve for ever. Practically, this situation may never arise but theoretically, it can happen. So if we can make a queue of threads trying to own the root or the last nodes, then we can make the implementation wait free. Implementing wait free queue is again another synchronization problem but this problem has been solved.

Bibliography

[1] Hakȧn Sundell. *Efficient and Practical Non-Blocking Data Structures*. PhD thesis, Chalmers University of Technology and Göteborg University,Sweden, 2004.

[2] Anonymous. *1965 - Moore's Law Predicts the Future of Integrated Circuits*. World Wide Web, http://www.computerhistory.org/semiconductor/timeline/1965-Moore.html, November 2007.

[3] Brian N. Bershad. Practical considerations for non-blocking concurrent objects. In *In Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 264–274, 1993.

[4] Wikipedia contributors. *Double compare and swap*. Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/wiki/Double_compare-and-swap, November 2011.

[5] Wikipedia contributors. *ABA problem*. Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/wiki/ABA_problem, March 2012.

[6] Wikipedia contributors. *Amdahl's law*. Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/wiki/Amdahl's_law, March 2012.

[7] Wikipedia contributors. *Hyper-threading*. Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/wiki/Hyper-threading, April 2012.

[8] Wikipedia contributors. *Moore's law*. Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/wiki/Moore's_law, March 2012.

[9] Wikipedia contributors. *Multi-core processor*. Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/wiki/Multi-core_processor, March 2012.

[10] Simon Doherty, David L. Detlefs, Lindsay Groves, Christine H. Flood, Victor Luchangco, Paul A. Martin, Mark Moir, Nir Shavit Guy, and L. Steele. Dcas is not a silver bullet for non-blocking algorithm design. In *In Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 216–224. ACM Press, 2004.

[11] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *PODC*, pages 131–140, 2010.

[12] Michael Greenwald. *Software Implementation of DCAS*. World Wide Web, http://www-dsg.stanford.edu/papers/non-blocking-osdi/node12.html, September 1996.

[13] Michael Greenwald. Two-handed emulation: how to build non-blocking implementations of complex data-structures using dcas. *Information Processing Letters*, 2002.

[14] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15:745–770, 1993.

[15] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13:124–149, 1993.

[16] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[17] Mark D. Hill and Michael R. Marty. Amdahls law in the multicore era. *IEEE COMPUTER*, 2008.

[18] Eshcar Hillel. *Concurrent Data Structures: Methodologies and Inherent Limitations*. Ph.D. thesis, The Technion - Israel Institute of Technology, February 2011.

[19] Freescale Semiconductor Inc. *MOTOROLLA MC68030: Enhanced 32 bit microprocessor user's manual*. Prentice Hall, 3 edition, 1992.

[20] Amos Israeli and Lihu Rappoport. Efficient wait-free implementation of a concurrent priority queue. In *7th International Workshop on Distributed Algorithms*, pages 27–29. Springer-Verlag, Sep 1993.

[21] Mike Jones. *What really happened on Mars?* World Wide Web, http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/mars_pathfinder.html, December 1997.

[22] Kevin D. Kissell. *Demystifying multithreading and multi-core*. World Wide Web, http://www.eetimes.com/design/automotive-design/4004762/Demystifying-multithreading-and-multi-core, September 2007.

[23] Carlo Kopp. *Moore's Law - Is the End Upon Us?* World Wide Web, http://www.ausairpower.net/OSR-0700.html, July 2000.

[24] Geoff Lowney. Why intel is designing multi-core processors. Presentation Slides, July 2006.

[25] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):4–15, February 2002.

[26] Henry Massalin and Calton Pu. A lock-free multiprocessor os kernel, 1991.

[27] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38, April 1965.

[28] Michel Raynal. On the implementation of concurrent objects. In *Dependable and Historic Computing*, pages 453–478, 2011.

[29] John Shal, Krste Asanovi, David A. Patterson, Kurt Keutzerand Tim Mattson, and Katherine Yelick. The manycore revolution: Will the hpc community lead or follow? *SciDAC Review*, September 2009.

[30] Andrew S. Tanenbaum. *Modern operating systems (3. ed.)*. Pearson Education, 2008.

Vita

Graduate College
University of Nevada, Las Vegas

Mahesh Acharya

Degrees:

Master of Science in Computer Science 2012
University of Nevada Las Vegas

Thesis Title: Non-Blocking Concurrent Operations on Heap

Thesis Examination Committee:

Chairperson, Dr. Ajoy K. Datta, Ph.D.
Committee Member, Dr. Ju-Yeon Jo, Ph.D.
Committee Member, Dr. Lawrence L. Larmore,Ph.D.
Graduate Faculty Representative, Dr. Venkatesan Muthukumar, Ph.D.