Efficient Lock-free and Wait-free Algorithms for Operating System Kernels

Henning Dietmar Weiss

Kongens Lyngby 2010
IMM-BSC-2010-01

Technical University of Denmark
DTU Informatics
Department of Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

In this thesis a memory manager based on NBMalloc was implemented in the experimental multi-core operating system fenixOS. It is able to allocate blocks of memory in a lock-free manner. Unit tests for improving the quality and maintainability of NBMalloc have been created for all of it's non-trivial functions.

The prior code of the deque and the garbage collector was analyzed and the implementation of the scanning function was completed. Besides, minor bugs in the deque could be identified and corrected. Tests of the deque and the garbage collector were performed in a semi-random fashion and a unit testing framework in fenixOS was created.

Preface

This thesis was prepared at Department of Informatics at the Technical University of Denmark in partial fulfillment of the requirements for acquiring the B.Sc. degree in engineering.

The thesis deals with memory management issues of operating systems on multicore architectures. The main focus is on algorithms for lock-free memory allocation and a lock-free deque data structure and efficient ways of their implementation in an experimental operating system.

Lyngby, June 2010

Henning Dietmar Weiss

Acknowledgements

I would like to thank my supervisor Sven Karlsson for giving me guidance in conducting the work on lock-free structures and helping me with various discussions about the performance issues connected to it.

I would also like to thank the FenixOS team for many insightful discussions of the implementation of the operating system.

I would also like to thank my family for the care and support throughout the writing of my thesis.

Contents

Chapter 1

Introduction

Modern computer hardware will drastically change the way we write software in the next decade. This change is caused by the increasing number of CPU that are available in computers. Even today it is not uncommon for a consumer grade PC to have four CPU. To take advantage of this trend, software designers need to design their software with parallelization in mind from the beginning.

Concurrency situations occur on large multi-processor systems as well as in multi-process systems. A major challenge with such concurrency is the sharing of variables. The traditional way of approaching this problem is to define processes to be mutual exclusive when entering a critical memory region [8]. This region encapsulates the part of the code which makes use of shared variables. Mutexes and similar locking mechanisms are used to prevent other processes from accessing the shared variables simultaneously. However, lock-based approaches do not represent an optimal solution due to the risks of deadlock, starvation and priority inversion.

Lock-free approaches are a remedy. The processes can access shared variables in a concurrent fashion. This allows greater concurrency and threads competing for a shared resource without having their execution indefinitely postponed by mutual exclusion. The criterion of wait-freeness adds an additional requirement compared to lock-free approaches. They have also to guarantee per-thread progress.

However, the guarantee of system-wide progress is achieved on the expense of greater complexity of data structures and algorithms involved. The issue of complexity is indeed problematic with regard to both implementation costs and overall efficiency in non-highly concurrent environments. Lock-free approaches are only efficient for data structures that are used by many processes at once. Otherwise the performance gain may be lost and unnecessary complexity is added.

An operating system usually contains many internal data structures that keep track of it's internal state. Consequently, these data structures are subject to extreme concurrency. The double linked list is one of the most commonly used structures. Therefore, a specific variant of the double linked list will be covered in greater detail in section 2.2.1.

Besides, operating systems provide memory allocators. They allow programs to request blocks of memory for their own use. Because this is frequently done by many different threads, it is desirable that a high degree of concurrency is achieved. Efficient memory management has a significant effect of the system performance.

In this thesis the lock-free *NBmalloc* memory allocator and the deque data structure will be employed and their performance in comparison to their lock based counterparts will be considered. The implementation of those components will be done in an experimental OS to investigate the potential of lock-free approaches in modern operating system infrastructures.

Chapter 2

Background

2.1 Operating System

2.1.1 Overview

The installation of an operating system, or OS for short, is needed for making a computer operational. The OS is therefore an integral part of the software stack, with the notable exception of embedded devices. Even tough the term "operating system" is widely used, it is still difficult to give a precise definition everyone can agree on. In the following section an overview of some of the components that modern OS provide will be given.

An OS is an abstraction layer to the hardware, see Fig. 2.1. Abstraction allows them to use the interface that the OS provides, instead of having to consider the hardware of the users computer directly. If the programmer had to do that, he would need to write drivers for all possible configurations that the users of his program could possibly have. For a moderately complex application that takes advantage of modern hardware, this would require a monumental effort. The components the program uses might include graphic card, sound card, USB connectors, CD and DVD drives etc. But even very basic hardware like the CPU and RAM require some extra logic to work in the convenient way programmers

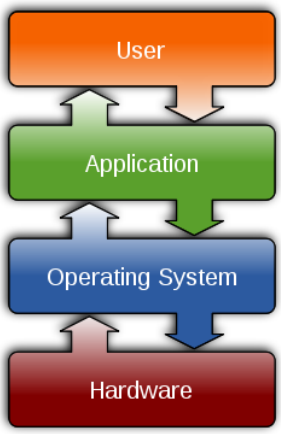

Figure 2.1: Abstraction in an operating system [11]

have come to expect. External storage medium usually have file systems. These can become quite complex as well and must provide the identical on-disk format.

In addition of being very time consuming, an essential amount of redundant work is being done by each application developer if they would not rely on an OS. Development time and costs are saved by discouraging the developers to interact with the hardware directly and using the abstraction layer that the OS provides instead. This results in better hardware drivers (exploiting more efficiently the particular hardware features) and cost efficient end user programs due to their improved portability.

OS enable features that would not be possible without them. Since they are the major control mechanisms, they can enforce complex mechanisms on programs like security policies. One notable mechanism is memory protection. Memory protection allows the OS to prevent a program from reading or writing places in memory. Since programs have to request the read or write through the OS, it can decide if it is safe for a program to do so. Thus, programs can be restricted to areas of memory that are not critical for other programs. Restrictions of this kind prevents the alteration memory regions that the OS or other programs rely on. If this feature were not in place, faulty or malicious programs could cause the operating system to crash.

One of the more user visible features is multi-tasking. Allowing multiple pro-

grams to run simultaneously on one CPU is expected by any modern computer[1]. Additionally, it is expected that the OS can utilize multi-cores in an efficient manner if the computer has several CPU.

2.1.2 FenixOS

FenixOS is an experimental operating system written in C++. Its founder and lead architect Sven Karlsson is also the supervisor of my bachelor project. The motivation behind FenixOS was that all popular operating systems are based on POSIX with the exception of MS Windows. POSIX stands for "Portable Operating System Interface for Unix" and is more than two decades old. The thought is that computers and how we use them has changed over time, perhaps allowing a fresh start with a new vision.

FenixOS is at an early stage of development. It is based on monolithic design. It also moved some components, that are traditionally in the kernel, out of kernel space as processes. Moving complex processes out to user space has the advantage of making the system more stable. If these processes crash, they are isolated from the kernel, allowing the OS to restart them. If the same processes where in the kernel, they might crash the entire OS and requiring a reboot.

When starting my project, the OS was able to boot and print to a text terminal. It had a simple memory allocator that could allocate pages. My project started from a branch of the main trunk, in which Sven had already implemented an almost complete lock-free garbage collector and lock-free deque.

The components that where covered by other members of the team include the whole file system infrastructure, kernel debugging, scheduling and the porting of the entire OS to other hardware platforms. An libc port was also required. The libc is the standard C library. It provides important functionality like string manipulation and memory management. The reason for it being important for an OS is that many existing C programs rely on the libc. Porting a libc to FenixOS would have the benefit of being able to compile and run many existing applications in its environment.

[1] with the exception of the iPad

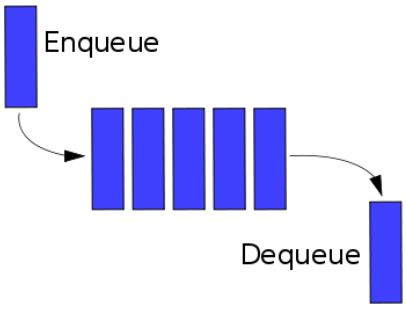

Figure 2.2: Queue [11]

2.2 Data Structures

Data structures are used by an OS to keep track of its internal state. Some states change often and are critical to the overall performance of the software stack. One of the data structures dealing with states is the deque, which I will take a closer look at in the following section.

2.2.1 Deque

The expression deque is a shorthand for "double ended queue". It uses a double linked list as base, but only the head and tail of it can be modified through it's provided operations, Fig. 2.2. A double linked list is a data structure that has the operations *insertBefore*, *insertAfter*, *Delete*, *Read*, *Next*, *Prev*, *First* and *Last*. Most of these operations can modify the list in arbitrary positions. In contrast, the deque only provides the *pushLeft*, *pushRight*, *PopLeft* and *PopRight* operations. The term push means pushing (or adding) an element to the queue and pop means popping (or removing) an element. Left refers to the head and right to the tail. These terms are derived from thinking about the queue as a list that has a horizontal orientation. This list expands "to the right". Where as the ordinary queue can only be used as FIFO (first in, first out), the deque allows greater flexibility. It could implement a LIFO (last in, first out) or any variation of inserting elements on either end and retrieving them from either end, making it a very versatile data structure for queuing. Its main utility is when elements are expected to be kept in the order in which where enqueued and all necessary modifications can be done by modifying either the head or the

tail.

A typical application of deques is the scheduling of processes. A scheduler is essentially an algorithm that chooses the next task that is to be executed. In a primitive scheduling algorithm, a process would add itself to deque to indicate its desire to be scheduled. The scheduler would then stop a running task at regular intervals, push it onto the queue, pop a different task off the deque and schedules it to be run.

2.2.2 Concurrency and data structures

In concurrency theory there are several problems related to the access of data structures by several threads at the same time. The main problem is that the structures can be left in an inconsistent state or that the threads read wrong information out from them. The traditional way of solving this problem is through various locking mechanism. These are based on the premise that before entering a so called critical region, a thread checks if it has permission to do so. A critical region is essentially a piece of code (or pieces) that may not be executed by several processes in parallel. When this occurs it may result in an inconsistent state. By setting a lock, the process signals other processes that it is currently in the critical region. Before the other processes enter, they will check for the lock in the same fashion and will wait until they have clearance.

The waiting can be done by so called busy looping, which essentially causes the process to enter an infinite loop that checks over and over again if it is allowed to enter. In more sophisticated systems the process will enter sleep mode and there will be some kind of callback function that wakes up the process when it is allowed to enter. This saves a lot of resources, but requires the appropriate infrastructure to be in place.

Another problem with the mutual exclusion is that certain other border conditions can occur, such as deadlocks or starvation. Deadlocks can occur when process A locks resource X and then waits or resource Y, while process B has already locked resource Y and waits for X. This will result in a state where neither of the processes can redeem the situation, resulting in a state which is “dead”. Starvation is similar in that one process is in a “dead” state, because other processes manage to allocate the lock faster. This causes the “starving” process to wait indefinitely for clearance to the critical region.

2.2.3 Lock-free data structures

The term lock-free data structures describes data structures which are non-blocking. They do not employ locks or mutual exclusion of any kind to ensure consistency in the data structure. Essentially the critical operations are identified in the algorithms, and then measures are taken to ensure that a consistent state is maintained. The critical part in this case describes the situation when a process assumes a value to be have a certain value, which has been changed by a different thread in the mean time. The values become inconsistent when it replaces the new value with its modified value that is based on non current information. To prevent this, the algorithm needs to ensure that it can be certain that the value has not changed until the precise time of writing. This is achieved through an atomic operation that the CPU needs to support directly.

In the case of the AMD64 platform it is called lock_cmpxchg, which is an assembly instruction. In general this instruction is called Compare-and-Swap(CAS). It needs three arguments: the memory location to write to, the assumed value and the new value to write. If the assumed value matches the value at the location it will write the new value and indicate success. Otherwise it will take no action and indicate failure. The related function Fetch-and-Add(FAA) is also used in the algorithms described below. It represents an optimization, since it can easily be implemented with the CAS instruction. It takes the target memory address and a value as parameter and will cause the target variable to be incremented or decremented by the given value.

An extension of CAS is CAS2 (Double-Compare-and-Swap) which is like CAS designed as an atomic primitive proposed to support concurrent techniques. Similar to CAS, CAS2 writes new values to two memory locations if they match the expected values.

The two atomic operations CAS and CAS2 allow to design concurrent data structures without the use of mutual exclusion. A simple and common design pattern is the following. An algorithm starts by reading a shared variable into a local temporary. It then uses the value of that temporary to make all of its calculations. Finally it tries to change the variable with the CAS instruction, using the temporary as old value and its calculated value as new value. This allows the algorithm to detect if the value has been changed from what it assumes the value to be. When the CAS instruction was unsuccessful, the algorithm can simply start from the beginning by re-reading the variable, since it has not modified global state yet. The algorithm can continue in this fashion until it successfully changed the target variable.

2.2.4 The ABA problem

The ABA problem refers to a condition that can occur when a thread reads a shared variable with the intention of modifying it later. If the value of that variable is changed to a different value in the mean time, the algorithm can detect this change by using the CAS operation. However, this mechanism fails if the particular value is changed twice. The first change might modify the variable's value to a different one, but the second modification could change it back to the value that the initial thread expected. The thread would then not be able to detect that a change took place. This is problematic since global state could be modified, without the thread being aware of it.

One example of where this might cause problems is in a simple stack. A process might read the first and the second element. The process is suspended and a different process is run in the mean time. This second process could delete the top two elements, but add the first element again, or an element of the same value, to the top again. When the first process resumes execution, it will check if the top element is the same. This causes the first process to believe nothing has changed. When the second element is referenced as if it still where in the stack, it might cause corruption of the internal state or a null pointer reference.

One method of solving this problem is versioning. Each shared variable then gets an associated version field which is changed atomically with the data field. The version field is incremented by one on each read. This allows a thread to detect changes, even if the data field is unchanged.

2.3 Memory Management

2.3.1 Reference Counting

When designing complicated data structures there is a need for dynamic memory management. The reason for this is that traditionally a data structure is a number of nodes that are connected to each other by links. Nodes are essentially pieces of memory that hold data and links are pointers, addresses of memory. When a node is deleted, meaning that it's memory is given marked for later use, it is important to keep track of the links. The rationale behind this is that when a different node still depends on the node that it is about to be deleted, it might put the node into an inconsistent state. So the algorithm needs some way to keep track of how many nodes still need a node to be in existence, so it

does not free memory that is still being referred to by other nodes. One method of dealing with this problem is reference counting.

Reference counting means that the number of active references to a node is stored. This makes it possible to wait until all references to a node are released, before it is freed.

This means that the design of an algorithm does not need to take care of managing the deallocation of the node. It can instead mark a node as deleted and move on with it's normal operation. The mark makes will have the influence that other decisions that the algorithm makes will not rely on that node. As soon as the last reference is released from that particular node it can be freed and it's memory can be reused.

This can be used to implement a form of automatic memory management, also called garbage collection.

2.3.2 Performance issues

There are several challenges that must be overcome when implementing an efficient lock-free memory manager.

One of the major problems is fragmentation. Fragmentation is a problem that refers to pieces of memory that cannot be allocated. Memory is usually managed in blocks. The manager has a fixed size (or sizes) of blocks that can be handed out. When a process requests blocks that are less then that size, the memory manager will need to fulfill that request by giving a block that is larger. This means that the amount that is left of the block is unable to be used by a different process. When this happens, a lot memory is "fragmented" into smaller blocks which are too small to fulfill any requests the process might make.

Another problem is false sharing. It refers to a performance degrading usage pattern when sharing caches between different CPUs. For instance, if a CPU attempts to access data that will never be altered by another CPU and that data shares a cache block with data that is altered, the first CPU might get mistakenly forced to reload the cache in spite of the fact that the block considered is unchanged.

2.4 Summary

In this chapter the background of lock-free data structures was considered. Besides the problem of concurrency, the implementation of lock-free and wait-free algorithms demands special attention to the design of efficient data structures allowing a high performance by avoiding fragmentation and unnecessary caching. The following chapter deals with potential solutions to the memory management issues associated with lock-free algorithms.

Chapter 3

Theory

3.1 Memory Management

As outlined in chapter 2, an OS has primarily the two functions of providing a virtual machine and to manage the sharing of computer resources. The most critical task with regard to reliability and performance is memory management. This chapter deals with the choice of a memory allocator concept for FenixOS.

3.2 Hazard Pointers

One problem that occurs with lock-free algorithms is the reclamation of dynamic objects. Dynamic objects are pieces of memory that are allocated while the algorithm is running. It can be quite problematic to manage references to these objects when it comes to deletion. Parts of the algorithm may still have references to these objects. If the object is freed without taking the pending references into account, it might cause null pointers and inconsistent states.

An effective method of solving this problem is Hazard Pointers, or HP for short. HP are references to shared memory. Instead of referencing the memory by

pointer or reference directly, a thread allocates a HP to it. When the allocation of a HP was successful, it will be added to a list that indexes all HP of the current thread. The HP gives the guarantee that the memory will be available. If the thread did not have that guarantee, it would have to check the availability each time, the lock-freeness of the algorithm would not be preserved. When a thread is done accessing this memory location, the hazard pointer should be released.

Instead of deleting nodes, a thread will "retire" them. This means that the memory is not reclaimed right away. Instead the memory is added to the so called rlist. It is not possible to allocate a new HP to this memory after this happened. HP that where allocated before the memory was freed are still functioning, as per the guarantee described above. The rlist is thread local list of retired nodes. When the list grows to a certain size threshold by retiring a node, the scan function will be run. This function will iterate through the entire list of HPs and add them to a temporary list. The scan function will iterate through each element in this list, trying to find a matching entry in the rlist. If no match for an entry is found, the HP is removed from the rlist and the node is marked for reclamation. If this is not the case, it is kept in the rlist for future reclaiming.

The problem with HP is that they do not support global references. This makes the implementation of many algorithms difficult, since they might rely on references that are inside nodes. One simple example is the linked list. Each node contains a pointer to the next node. This results in references that are not thread local.

An extension of the Hazard Pointer method is [5] called *BEWARE&CLEANUP*. It makes it possible to have global references and guarantees prevention of dangling pointers. The algorithm follows the same ideas as Hazard Pointers. It keeps a list of retired nodes and runs a cleanup method when a certain threshold is hit.

3.3 Hoard

Hoard is a memory allocator for multi-processor systems. It is a lock-based allocator, but I will introduce it here because NBMalloc is based on it. Hoard is well known for its ability to allocate memory in a highly efficient manner. It is very fast, scales well and has a low level of fragmentation (see section 2.3.2).

The central concept of hoard is that each CPU has its own heap. Those heaps contain Superblocks. Superblocks that are not owned by any processor are

contained in the global heap. Before trying to allocate new Superblocks, a CPU will always check the global heap for vacant Superblocks before trying to allocate new Superblocks. Each Superblock contains a fixed number of blocks. The blocks represent the actual memory that can be allocated.

When a process requests memory from Hoard, Hoard will first search through the local heap of that CPU on which the process is running on. It will start by searching through the list of Superblocks, starting with the fullest first. This ensures that the fullest blocks are always filled up first. This reduces the number of Superblocks that are needed by each local heap. When an empty block is found, it is marked as in use and its address is returned to the process. If no empty Superblock is found in the local heap, Hoard will try to find one in the global heap. If the global heap contains a Superblock, it is moved to the local heap of the corresponding processor and a free block is returned. If the global heap does not contain such a Superblock, Hoard allocates new memory from the operating system (with *mmap* or similar mechanisms) and proceeds as above.

When a block is freed, Hoard will find the Superblock this blocks belongs to. It then marks the block as allocatable. If this causes the number of free blocks to hit a certain threshold, the Superblock will be moved to the global heap. This is done to prevent heap blowup.

To speed up the searching through and moving of Superblocks, Hoard groups them in so called fullness groups. Fullness groups represent the amount of blocks left in a Superblock. The maximum fullness group might contain all Superblocks of a certain CPU that have no blocks free. When searching for a free block, Hoard can skip this group and start by searching through the next least empty group. When allocating and freeing blocks, this property is maintained by checking if a Superblock has changed fullness group by the modification taken. If so, the Superblock is moved to the according fullness group or to the global heap.

Hoard has several parameters. The number of blocks every Superblock contains, the number of fullness groups, the size of each block and the threshold where blocks are moved back to the global heap. Many of these parameters are highly application specific and must be adjusted to the particular use case.

3.4 Deque

The lock-free dequeue from [8] is based on a lock-free double linked list. The dequeue can be thought of as an extension to the linked list functionality. Since

we can add elements dynamically we require a lock-free mechanism to allocate storage for these nodes. This is provided by NBMalloc, which I covered in a previous section.

One major problem with lock-free lists occurs when a node is being inserted. The node that is left of that node could be deleted before the insert operation is done. If this happens the inserted node is effectively also deleted from the list, since the next pointer of the node to the left of the deleted node now points to the element right of the inserted node. The solution to this problem proposed in the paper is to mark the next pointer of the node previous to the deleted node. This is done by keeping the address of the element pointers together with a boolean flag. This allows the manipulation of the flag and the next address atomically by using a single CAS operation. When the flag is set, the element the pointer is pointing to is about to be deleted.

The algorithm will maintain valid next pointers through the entire list. Previous pointers will be updated when the marked nodes are removed from the list. This makes it possible to traverse through deleted elements. When traversing backward through valid nodes on the list, it is necessary to check if a node is still member of the list. This can be done by checking the node to the left of that node next pointer. If it points to the node it is still a member, if it doesn't it has been deleted.

As mentioned above, the algorithm will only ensure next pointers to be reliable. I will now give an overview of the basic steps of the algorithm for insertion and deletion. It is necessary to apply the helper function *correctPrev* to ensure the lock-freeness of the algorithm.

When an insertion of a node is done, the algorithm will first set the according previous and next pointer of that node. This is a non intrusive action with regards to the list. Assuming that the new node called C is inserted between node A and B. Then the next pointer of A is updated to point to C. This is done by CAS. If the CAS fails it means that a different element has been inserted after A or that B has been deleted. In this case the algorithm simply updates it's assumption about the list and tries again from the beginning. If the CAS was successful, the previous pointer of B will be updated. This shows that there is a moment between the updating of A and B where iterating through the previous pointers would yield B-A instead of the correct sequence B-C-A. *insert illustration*

When deleting a node B from between nodes A and C, following steps are made. First the *A's* next pointer is marked, indicating a deletion. Second *C's* previous pointer is marked. Third *A's* next pointer is updated to point to C, resetting the mark in the process. Finally *C's* previous pointer is updated to point to A.

The deleted node B is now no longer part of the list and it's memory can be freed accordingly.

In addition to the list operations, the algorithm also provides the four dequeue functions. They are *pushLeft(), pushRight(), popLeft() and popRight().* These functions will use the same insertion and the deletion function as the underlying list, but will operate on the head or the tail as their target node.

The helper function *correctPrev* is invoked on many operations that modify the list. It, as the name suggests, updates the previous pointers of the list. *fill in some details*

When the algorithms in the dequeue fail, it is sometimes not optimal to try repeating the operation immediately after it failed. There may be another thread that is modifying nodes related to the operation that is in progress. In this case a thread will wait for a predefined while. This is called backing-off. For every failed the wait time will increase exponentially. This back-off approach is used by many of the functions of the algorithm.

3.5 NBMalloc

NBMalloc is a lock-free memory allocator based on Hoard. Similar to Hord, NBMalloc uses Blocks, Superblocks and fullness groups to optimize searching and moving, see figure 3.1.

Size classes is a new concept in NBMalloc. The idea is that a Superblock contains a fixed number of blocks. All blocks in this Superblock have usually the same size. Size classes are fixed groups in which all allocation requests must fit. If a process tries to allocate a block that is larger than the highest size class, NBMalloc cannot allocate it. In this case the process should use a different mechanism of allocating memory, like requesting pages from the operating system directly.

A process passes the size of the block of memory if it wants to allocate to *malloc().* NBMalloc then maps this size to the next biggest size class. This has the effect that a process will get a bigger chunk of memory than it requested if the size of the block does not match the size class, but has the advantage of not needing a size class for every conceivable size of blocks being allocated.

One major difference in the structure between Hoard and NBmalloc is the Flat-Set data structure. This structure contains a fixed size array of pointers to

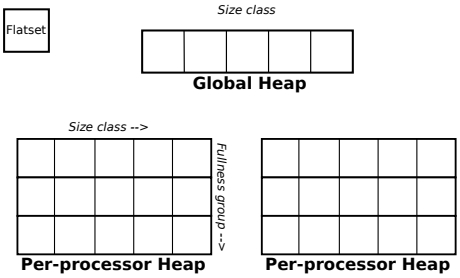

Figure 3.1: Overview of heap organization

Superblocks, which are initially null pointers.

The global heap contains a FlatSet for each size class, see figure 3.1. The per processor heaps have a two dimensional array of FlatSets. The first dimensions is the size class and the second the fullness group.

The *move_info_t* data structure is responsible to keep track of the movement of Superblocks. It is contained in each Superblock. The move operation in NBmalloc needs to make sure that a Superblock is accessible at any time during a move, since other external processes are not aware of movements taking place and expect the blocks to be accessible. Additionally other movements on the same Superblock might be detected and dealt with. It involves four steps. Each step has different outcome possibilities.

The first step of the movement is done by modifying the *move_info_t* structure of the Superblock that we want to move. It is checked that no other movements is in progress. If we detect a movement that is in progress, we help the movement to finish before continuing with our move. If this is not the case we mark the *move_info_t:result* indicating a move that is in progress and the *move_info_t:to* to point to our destination.

The second step involves taking the to location from the first step and updating the to location with our Superblock. This is done by CAS. If the CAS failed, it is checked if the target location has become occupied in the mean time. If this is the case we clear the *move_info_t* and return failure. Otherwise we move to step 3.

The third and fourth step are there for clearing the *move_info_t:from* and updating *move_info_t* to the new from and setting a *move_info_t:result* indicating that no move is progress, since we successfully made the move.

All modifications to *move_info_t* need to be done through hazard pointers. This is required because it cannot be changed by a single CAS operation, since it is larger then one machine word.

Chapter 4

Results

4.1 Beware & Cleanup

A large part of the garbage collection was implemented by Sven Karlsson before I started my project. The functionality is found in

- `Reclaimable.hpp`,
- `ReferenceCountable.hpp` and
- `TaggablePointer.hpp`.

The following paragraphs will cover Sven's implementation in slight detail because the algorithm does not converge much from what I discussed in section "Lock-Free Garbage Collector". Afterwards I will discuss the additions made.

TaggablePointer is the interface to the pointers used in *B&C*. It holds a pointer and a flag in one word which are both modified through CAS. They are used as links between nodes in the deque.

ReferenceCountable is, as the name implies, the implementation of reference counting. The classes that inherit its interface have the operations *getRefer-*

ence() and *releaseReference()*. They are used to dereference an object that is about to be accessed by incrementing a counter of references and releasing a reference by decreasing the counter respectively.

Reclaimable is the base class of *B&C* managed objects. It contains *scan()* and the "cleanup" functions.

My contribution was to finish *scan()*. It was in a unfinished state, lacking a lot of logic. I also implemented the quicksort algorithm in three utility functions. It is used to sort the elements in the *plist*, as suggested as optimization in the literature.

Finishing *scan()* finishes the GC. I have not made any extensive stress testing or unit testing to check it's validity. What I have done is an analysis of the code, comparing it to the pseudo code of the paper and checking for any discrepancy. Additionally I checked the functionality of the GC by using it in the Deque implementation, which I will discuss in a following section.

4.2 Deque

The deque was in an almost complete and functioning state when I started my project. I mainly read and understood the entire code and fixed two *TaggablePointers* that where erroneously released. I will describe the overall structure of the implementation and the differences between it and the algorithm described in the paper.

The deque implementation is contained in `SafeHeadPointer.hpp`, `SafeHead.hpp`, `SafeLinkage.hpp` and `SafeLink.hpp`.

`SafeHeadPointer.hpp` is the interface that the user can manipulate. It contains *pushLeft()/Right()* and *popLeft()/Right()*. These are only function calls to the corresponding functions in *SafeHead* by the same name. As of now only *pushRight()* and *popLeft()* are implemented.

`SafeLinkage.hpp` is the base class of the nodes of the list. It contains a *TaggablePointer* to the next and the previous elements. Notably it also contains *correctPrev()*.

The *SafeHead* represents the head and *SafeLink* the nodes of the deque. Both `SafeHead.hpp` and `SafeLink.hpp` are children of `SafeLinkage.hpp`.

One performance advantage that is gained from having an object oriented design is that less dereferencing needs to be done. A good example of this is *correctPrev()*. This function needs three nodes to operate on. When calling this function, the address of the object itself is passed implicitly as the *this* pointer. Since the address will not change while we are in it's function call, we have the address of the active node without needing to call a dereference function.

It can be concluded that the deque is a high performing lock-free data structure implemented in FenixOS. The fixing of some minor bugs conducted in the framework of this project has helped to make it fully functional.

4.3 NBMalloc

I have implemented a memory allocator based on the NBMalloc paper [4]. The paper contains a pseudo code, on which the code is strongly oriented. One major difference is the adaption to object oriented design. The encapsulation trough object orientation allows to clearly group the data with its corresponding functions. This makes the code easier to read and understand. It also allows several simplifications with regard to the parameter lists of many functions. The functions often need the data structure on which to operate. By having a member function that modifies member variables this is not necessary. All of the above are widely known advantages of object oriented design, it is just stated here for motivating my choice of design. Especially since c++ allows both the object oriented and the imperative approach.

On the top level algorithm exposes *malloc()* and *free()* for use by other processes. These two functions behave like their libc counter parts. *malloc()* takes a size argument, that indicates how large it should be. It returns the address of the beginning of that block if it was successful, or 0 on failure it failed. *free()* takes the address to be freed as argument. It frees the block of memory so that it can be reused by NBMalloc.

The first major simplification of the algorithm was to remove the three version fields from *flat-set_info*, *superblock_ref_t* and *block_ref_t*.

The algorithm suggested in the literature uses the *moveInfo* data structure. It consists of five variables. This makes it not possible to update the *moveInfo* field atomically. In my implementation I have fitted *moveInfo* into one 64 bit variable. This is possible by omitting the *new_pos_value* and the *cur_pos_value* field. These two variables are used in the original algorithm to detect changes in the intermediate pointer structures that point to Superblocks. Since the versions

have been eliminated, those two fields are not required. This leaves us with only three remaining fields. By saving the *to* and *from* field as indices into an array of Flatsets, and the corresponding index into the FlatSet:set it is possible to further reduce the amount of memory required. This makes it possible to fit the two Superblock locations in few bytes each. The result field is only 2 bits wide. This allows the entire *moveInfo* to easily fit into 64 bit, making it possible to modify it with the CAS operation. Because modifications can be done in this fashion, it makes the use of hazard pointers to access *moveInfo* unnecessary, reducing the complexity of the algorithm even further.

4.4 Unit Testing

Developing in a kernel environment can be challenging due to the lack of infrastructure. Conveniences like debuggers and stack traces have to be implemented before they are available. The implementation of complex software can easily become cumbersome if the only debug tool is the programming language itself and a primitive print function.

In unit testing a developer will define requirements which a function or any part of the system has to fulfill. The expectations of the return value of the function or the state it modifies can be expressed as a test function in the target programming language. Each of these functions return a truth value, indicating if the test has passed or not. Tests can be grouped together to a test suite. When a test suite is run, the user will receive an overview over passed and failed tests, see Fig. 4.1. The advantage of this methodology is that the developer can ensure that a function still behaves in the manner he expects, even if complex changes have been made. This can take a lot of guesswork out of the process of finding misbehavior in a program. Additionally it allows a quick and up-to-date overview of the projects status, provided that tests of all components in question have been made.

Unit testing has several advantages towards regular development. It is possible to test in a bottom-up fashion. The developer ensures that the basic functions are working as expected, before building on top of them. It is also possible to test in a top-down fashion. This means that a test is written for the top most component of a program. This could be an interface or the upper function of an algorithm.

My implementation of the unit testing framework is modelled after *junit/cppunit*. It is contained in the two classes *FenixOSTest.hpp* and *FenixOSTestSuite.hpp*.

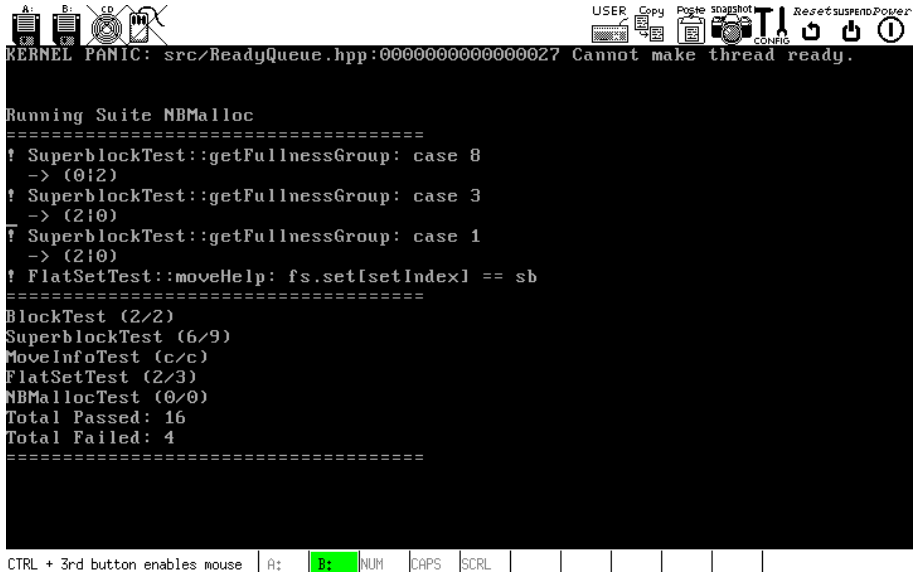

Figure 4.1: Test suite

FenixOSTest.hpp contains the base class that all tests should inherit from. The function *runTest()* has to be overridden and should contain all test functions to be run. Each test function should not return a value and can contain a arbitrary number of tests. When a test in a function passes, *pass()* should be called, otherwise *fail()* should be called. Those two functions increase their respective counters in the test class, to reflect how many passes and fails have been generated. On a fail the test will print additional information on the console about the details concerning the failure. Before each test function, *setUp()* and *tearDown()* are called. They can be used to manipulate the global state to reflect the test environment appropriate for the test. After all tests have been run, *report()* can be called. It prints the total number of passed and the total number of tests. *FenixOSTest.hpp* also contains some convenience macros that are used to obtain function names, line numbers and boolean test expressions as string values. This makes it easier to create tests and allows *fail()* to display more information about what went wrong.

A test suite is essentially a container for the test classes. Tests are added to it by calling *addTest()* with the test as parameter. The function *run()* can be called to run all tests functions of all the tests the suite contains. A report of the entire test run can then be generate with *report()*.

Chapter 5

Conclusion

The first thing I achieved was the implementation of a memory manager based on NBMalloc in the fenixOS environment. It is able to allocate blocks of memory in a lock-free manner. To improve the quality and maintainability of NBMalloc, I have created unit tests for all of it's non-trivial functions. The tests only cover non-concurrent access to the data structures.

The deque based on [8] and the garbage collector based on [5], was implemented by Sven Karlsson prior to the start of my project. I analyzed his code of both implementations, comparing them with the algorithms described in their corresponding paper. The garbage collector was completed with the exception of the *scan()* function. I successfully implemented it. With the exception of two minor bugs, the code of the deque was quite complete and optimal. I have tested the deque, and thereby indirectly the garbage collector. The tests I have done involved adding and removing elements to the deque in a semi-random fashion. This has been done with a dozen or so elements and requires further testing to ensure the correct functioning when dealing with a more massive data set.

Additionally I created a unit testing framework in fenixOS. It allowed me to divide my algorithms into smaller units and test their functionality separately. Having a test suite in fenixOS was a major convenience factor with regard to being able to rapidly deploy and test new functionality in this environment. It made it much easier for me to discover misbehavior in deep nested functions of

my code.

What is left to be done is the evaluation of the correctness of all the implemented components in a heavily concurrent environment with parallel access to all of the data structures. Benchmarks of the deque and the memory allocator need to be created. This will enable a quantitative performance comparison with high performing lock-based data structure counterparts, allowing me to draw conclusions about the real world utility of lock-free data structures.

Throughout my project I have learned about many aspects of software engineering. First I have gotten more familiar with reading and writing code in the C++ programming language. Most of the code in fenixOS is written in this language. Many advanced features of C++ are used to optimize the speed of code execution. Firstly, this required me to familiarize myself with many of these concepts to understand the design and inner workings of the operating system. Secondly, I got deeper understanding of the software engineering process in general and the development of complex systems that span many different abstraction layers through working on the source code of an operating system and being exposed to many different aspects of the development process. This has been partly through my peers, through the weekly status meetings of the fenixOS team. Thirdly, I learned how to implement complex algorithms from a partial specification. In my case this specification was in the form of scientific papers. The learning process involved how to alter the design to fit to the target programming language and working around the limitations of the target environment. Fourthly, I learned how programming languages and OSes manage memory. Through my work I have gained a deeper understanding of automatic memory management, e.g. garbage collection and explicit memory management including the challenges that must be overcome to achieve high performance.

Finally I would like to thank Sven Karlsson for creating fenixOS and giving me the opportunity to write the thesis about it. This has been overall a very enjoyable and educational project, with a steep learning curve. I would like to conclude with one saying that describes this very well: “a steep learning curve simply means that you learn a lot in a short period of time”.

Bibliography

[1] E D Berger, K S McKinley, R D Blumofe, P R Wilson Hoard: A Scalable Memory Allocator for Multithreaded Applications, 2000.

[2] T Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein Introduction to Algorithms.

[3] B Eckels, Thinking in C++ Volume 1+2, 2001.

[4] A Gidenstamm, M Papatriantafilou, P Tsigas : NBmalloc: Allocating Memory in a Lock-Free Manner, Algorithmica, DOI 10.1007/s00453-008-9268-x, Springerlink.com, Jan 2009

[5] A Gidenstamm, M Papatriantafilou, H Sundell, P Tsigas : Efficient and Reliable Lock-free Memory Reclamation Based Reference Counting, IEEE Trans. on Parallel and Distr. Systems, Vol. 20, No. 8, Aug 2009

[6] M Michael, Hazard Pointers: Safe Memory Reclaimation for Lock-free Objects, 2004.

[7] B Stroustrup The C++ Programming Language.

[8] H Sundell, P Tsigas, Fast and Lock-free Concurrent Priority Queues for Multi-Thread Systems, Technical report no. 2003-01, Chalmers University of Technology, Göteborg 2003

[9] H Sundell, P Tsigas : Lock-free deques and doubly linked lists, J. Parallel Distr. Comput. 68 (2008) 1008-20

[10] Tanenbaum, Modern Operating Systems 2nd Ed, 2007.

[11] Wikipedia, en.wikipedia.org/wiki/Memory_management, 22 Jun 2010

Appendix A

Project Planning

Project planning is part of every project management, which relates to the use of

- schedules for planning,
- risk analysis,
- subsequent reporting of project progress, and
- revising the timeplan.

Initially, the project scope was analyzed and the necessary steps for completing the project were defined. The durations for the various tasks were listed and grouped as well as the logical dependencies between tasks were identified. Due to the small size of the project, it does not make sense to use time on finding the critical path or to reserve float time in the schedule as it is needed in projects with certain team sizes.

For each activity, the necessary resources in terms of time were estimated which gives the total project cost. Progress was measured against the timeplan throughout the project work.

In the following, the inputs to project planning phase and concept proposal writing are outlined. The outputs of the project planning phase include the project requirements and project schedule.

A.1 Scheduling

A.1.1 Work Plan

1. Finish the linked-list implementation.
    - Refactor the code to use better variable and method names
    - Finish function *scan()*
    - Test/Debug with static memory management
    - Refactoring of variable names
2. Implement the lock-free memory manager.
    - Study paper on NBMalloc
    - Talk with Sven about the placement of this functionality
    - Create required data structures and function stubs
    - Implement $\text{get}_{\text{any}}()$, insert() and move()
    - Implement the super block operations and helper functions
    - Implement malloc() and free() and provide a public interface
    - Test/Debug
3. Perform performance evaluations comparing with lock based approaches.
    - Find lock based dequeue for comparison
    - Meet with Sven and/or Stavrock for discussion on evaluation methods
    - Conceive test scenarios
    - Represent results graphically
    - Evaluate results
4. Make the memory use of the existing garbage collector dynamic, so that:
    - Memory for internal data structures is allocated and de-allocated,
    - Possibly on a page basis, as needed by the algorithms.
    - Identify which parts need to be changed

- Modify the identified components
- Test/Debug
- Compare the performance of this approach to the previous static algorithm

5. If any time is left, investigate lock-free approaches to hash tables:
   - Hash maps and sets
   - Research Hash Tables/Maps and/or Sets
   - Implement
   - Test/Debug
   - Compare the performance to their lock-based version

A.1.2 Initial Timeplan

1. Finish dequeue (01.03 - 19.03) (3 weeks)
2. Implement GC (22.03 - 09.04) (3 weeks)
3. Performance Evaluation (12.04 - 16.04) (1 week)
4. Make memory use dynamic (19.04 - 07.05) (3 weeks)
5. Additional data structures (time left)
6. Writing report (24.05 - 25.06) (4 weeks)

A.1.3 Revised Timeplan

1. Finish NBMalloc (26.04 - 14.05) (2 weeks)
2. Finish dequeue + do performance evaluation on memory manager and dequeue (15.05 - 04.06) (3 week)
3. Write report + clean up and finish code (05.06 - 25.06) (3 weeks)

A.2 Risk Analysis

Scenario: Failure to implement lock-free dequeue

- Risk: Low
- Description: I am unable to implement a lock-free dequeue in the given time frame
- Consequence: Someone else has to overtake the development of this functionality in fenixOS. I will concentrate on a different area of my project.
- Prevention: Regular status meetings to keep development on track. Using additional time on development.

Scenario: The dequeue does not perform to our expectations

- Risk: Low-Medium
- Description: It might be that a lock-free dequeue is slower performance wise than a lock-based one
- Consequence: Use a lock-based dequeue instead
- Prevention: Use good coding practices, not letting performance be an afterthought

Scenario: Failure to implement lock-free memory manager

- Risk: Low
- Description: I am unable to implement a lock-free memory manager in the given time frame
- Consequence: Someone else has to overtake the development of this functionality
- Prevention: Regular status meetings to keep development on track. Using additional time on development

Scenario: The memory manager does not perform to our expectations

- Risk: Low-Medium
- Description: It might be that a lock-free memory manager is slower performance wise than a lock-based one
- Consequence: Use a lock-based memory manager instead
- Prevention: Use good coding practices, not letting performance be an afterthought

Scenario: Failure to make the garbage collector dynamic

- Risk: Medium

- Description: I am unable to make the garbage collector allocate memory dynamically
- Consequence: I will be unable to fulfill the goal objective
- Prevention: Using additional time on development

Scenario: Unexpected events (illness, death, Armageddon)

- Risk: Low
- Description: Unexpected events that might prevent me working on the project
- Consequence: I will be unable to fulfill my goal objectives
- Prevention: Planning a 'iron reserve' in terms of time and energy

Acronyms

Abbreviation	Meaning
CAS	compare and swap
CAS2	compare and swap with double words
CPU	central processing unit
deque	double-ended queue, also known as DEQ
FAA	fetch and add
FIFO	first in, first out
GC	garbage collector
HP	hazard pointer
LIFO	last in, first out
OS	operating system
POSIX	portable operating system interface for Unix
RAM	random access memory
USB	universal serial bus